Understanding Deep Learning

Application in Rare Event Prediction

Chitta Ranjan

Understanding Deep Learning
Application in Rare Event Prediction

Chitta Ranjan, Ph.D.

First Edition

Chitta Ranjan, Ph.D.
Director of Science, ProcessMiner Inc.
cranjan@processminer.com
`https://medium.com/@cran2367`

ISBN: 9798586189561

Printed in the United States of America.

Dedicated to my parents,
Dr. Radha Krishna Prasad and Chinta Prasad.

With love, Bharti.

Preface

Deep learning is an art. But it has some boundaries. Learning the boundaries is essential to develop working solutions and, ultimately, push them for novel creations.

For example, cooking is an art with some boundaries. For instance, most cooks adhere to: salt stays out of desserts. But chefs almost always add salt to desserts because they know the edge of this boundary. They understand that while salt is perceived as a condiment it truly is a flavor enhancer; when added to a dessert it enhances the flavor of every ingredient.

Some chefs push the boundaries further. Claudia Fleming, a distinguished chef from New York, went to an extreme in her *pineapple* dessert. Each component in it is heavily salted. Yet the dessert is not salty. Instead, each flavor feels magnified. The salt makes the dessert an extraordinary dish.

The point is that an understanding of the constructs of food allows one to create extraordinary recipes. Similarly, an understanding of deep learning constructs enables one to create extraordinary solutions.

This book aims at providing such a level of understanding to a reader on the primary constructs of deep learning: multi-layer perceptrons, long- and short-term memory networks from the recurrent neural network family, convolutional neural networks, and autoencoders.

Further, to retain an understanding, it is essential to develop a mastery of (intuitive) visualizations. For example, Viswanathan Anand, a chess grandmaster, and a five-time world chess champion is quoted in *Mind Master: Winning Lessons from a Champion's Life*, "chess players visualize the future and the path to winning." There, he alludes that a player is just as good as (s)he can visualize.

Likewise, the ability to intuitively visualize a deep learning model is essential. It helps to see the flow of information in a network, and its transformations along the way. A visual understanding makes it easier to build the most appropriate solution.

This book provides ample visual illustrations to develop this skill. For example, an LSTM cell, one of the most complex constructs in deep learning is visually unfolded to vividly understand the information flow within it in Chapter 5.

The understanding and visualizations of deep learning constructs are shrouded by their (mostly) abstruse theories. The book focuses on simplifying them and explain to a reader **how** and **why** a construct works. While the "how it works" makes a reader learn a concept, the "why it works" helps the reader unravel the concept. For example, Chapter 4 explains how dropout works followed by why it works?

The teachings in the book are solidified with implementations. This book solves a rare event prediction problem to exemplify the deep learning constructs in every chapter. The book explains the problem formulation, data preparation, and modeling to enable a reader to apply the concepts to other problems.

This book is appropriate for graduate and Ph.D. students, as well as researchers and practitioners.

To the practitioners, the book provides complete illustrative implementations to use in developing solutions. For researchers, the book has several research directions and implementations of new ideas, e.g., custom activations, regularizations, and multiple pooling.

In graduate programs, this book is suitable for a one-semester introductory course in deep learning. The first three chapters introduce the field, a working example, and sets up a student with TensorFlow. The rest of the chapters present the deep learning constructs and the concepts therein.

These chapters contain exercises. Most of them illustrate concepts in the respective chapter. Their practice is encouraged to develop a stronger understanding of the subject. Besides, a few advanced exercises are marked as optional. These exercises could lead a reader to develop novel ideas.

Additionally, the book illustrates how to implement deep learning networks with TensorFlow in Python. The illustrations are kept verbose and, mostly, verbatim for readers to be able to use them directly. The link to the code repository is also available at `https://github.com/cran2367/deep-learning-and-rare-event-prediction`.

My journey in writing this book reminded me of a quote by Dr. Frederick Sanger, a two-time Nobel laureate, "it is like a voyage of discovery, seeking not for new territory but new knowledge. It should appeal to those with a good sense of adventure."

I hope every reader enjoys this voyage in deep learning and find their adventure.

Chitta Ranjan

Acknowledgment

A trail of learning experiences ultimately leads to writing a book. I have drawn from many people during my experiences to whom I am very thankful.

I thank Dr. Kamran Paynabar and Dr. Jhareswar Maiti who instilled in me the passion for learning and discovery during my academic life.

I also want to thank Dr. David Hardtke, Dr. Ronald Menich, and Karim Pourak for giving me an intellectually rich environment at my past and current work. The professional and sometimes philosophical conversations with them gave me insightfulness to write this book.

My father, Dr. Radha Krishna Prasad, who is an inspiration in my life helped me get started with the book. His guidance during the writing invaluably enriched it.

Besides, bringing this book to its completion would not have been possible without the support of my wife, Bharti. I am grateful for her immeasurable support and motivation when writing was difficult.

I also thank Dr. Shad Kirmani, eBay Inc., Zhen Zhu, Stanford University, and Jason Wu, who helped improve the theoretical content in the book. I am also grateful to Professor Mary Orr and Grace Gibbs for their inputs.

Lastly, I want to thank my family and friends for their perennial encouragement.

Website

This book provides several implementations of deep learning networks in TensorFlow. Additionally, video lectures are provided for most of the sections in the book. The website provides the link to the code repository, links to the lectures, and other resources useful for both readers and instructors.

`www.understandingdeeplearning.com`

Contents

This page is intentionally left blank.

Chapter 1

Introduction

Data is important to draw inferences and predictions. As obvious as it may sound today, it has been a collective work over the centuries that has brought us this far. Francis Bacon (1561-1626) proposed a Baconian method to propagate this "concept" in the 16th century. His book *Novum Organum* (1620) advanced Aristotle's *Organon* to advocate data collection and analysis as the basis of knowledge.

Centuries have passed since then. And today, data collection and analysis has unarguably become an important part of most processes.

As a result, data corpuses are multiplying. Appropriate use of these abundant data will make us potently effective. And a key to this is **recognizing predictive patterns from data** for better decision making.

Without this key, data by itself is a dump. But, at the same time, drawing the valuable predictive patterns from this dump is a challenge that we are facing today.

> "We are drowning in information while starving for knowledge. The world henceforth will be run by synthesizers, people able to put together the right information at the right time, think critically about it, and make important choices wisely." – E.O. Wilson, *Consilience: The Unity of Knowledge* (1998).

John Naisbitt stated the first part of this quote in Megatrends (1982) which was later extended by Dr. Edward Osborne Wilson in his book *Consilience: The Unity of Knowledge* (1998). Both of them have emphasized the importance of data and the significance of drawing patterns from it.

Humans inherently learn patterns from data. For example, as a child grows she learns touching a hot cup will burn. She would learn this after doing it a few times (collecting data) and realizing the touch burns (a pattern). Over time, she learns several other patterns that help her to make decisions.

However, as problems become more complex humans' abilities become limited. For example, we might foretell today's weather by looking at the morning sun but cannot predict it for the rest of the week.

This is where Artificial Intelligence (AI) comes into the picture. AI enables an automatic derivation of predictive patterns. Sometimes the patterns are interpretable and sometimes otherwise. Regardless, these automatically drawn patterns are usually quite predictive.

In the last two decades, AI has become one of the most studied fields. Some of the popular texts in AI are, *Pattern recognition and machine learning* by Bishop, C. Bishop 2006, *The elements of statistical learning* by Friedman, J., Hastie, T., and Tibshirani, R. Hastie, Tibshirani, and Friedman 2009, and *Deep Learning* by LeCun, Y., Bengio, Y., and Hinton, G. LeCun, Bengio, and G. Hinton 2015.

In this book, we will go a little further than them to understand the constructs of deep learning. A rare event prediction problem is also solved side-by-side to learn the application and implementation of the constructs.

Rare event prediction is a special problem with profound importance. **Rare events are the events that occur infrequently**. Statistically, if an event constitutes less than 5% of the data set, it is categorized as a rare event. In this book, even rarer events that **occur less than 1%** are discussed and modeled.

Despite being so rare when these events occur, their consequences can be quite dramatic and often adverse. Due to which, such problems are sometimes also referred to as adverse event prediction.

Rare event problem has been categorized under various umbrellas.

For instance, “mining needle in a haystack,” “chance discovery,” “exception mining,” and so on. The rare event prediction problem in this book is, however, different from most of these categories. It is predicting a rare event in advance. For example, predicting a tsunami before it hits the shore.

Rare event prediction is predicting a rare event in advance.

In the next section, a few motivating rare event examples are posed. Thereafter, a dialogue on machine learning versus deep learning approaches and the reasoning for selecting deep learning is made in § 1.3. Lastly, a high-level overview of the rest of the book is given in § 1.4.

1.1 Examples of Application

Rare event problems surround all of us. A few motivating examples are presented below.

Before going further, take a moment and think of the rare event problems that you see around. What are their impacts? How would it be if we could predict them? Proceed with the thought in mind.

1.1.1 Rare Diseases

There are 5,000 to 8,000 known rare diseases. Based on World Update Report (2013)[1] by WHO, 400 million people worldwide of which 25 million in the US are affected by a rare disease.

Some rare diseases are chronic and can be life-threatening. An **early detection** and diagnosis of these diseases will significantly improve the patients’ health and may save lives.

[1] `https://www.who.int/medicines/areas/priority_medicines/Ch6_19Rare.pdf`

The International Rare Diseases Research Consortium (IRDiRC) was launched in 2011 at the initiative of the European Commission and the US National Institutes of Health to foster international collaboration in rare diseases research. However, despite these developments, rare disease diagnosis and early detection is still a major challenge.

1.1.2 Fraud Detection

Digital frauds, such as credit card and online transactions, are becoming a costly problem for many business establishments and even countries. Every year billions of dollars are siphoned in credit card frauds. These frauds have been growing year after year due to the growth in online sales. A decade ago the estimated loss due to online fraud was \$4 billion in 2008, an increase of 11% from \$3.6 billion in 2007.

The fraud's magnitude is large in dollars but constitutes a fraction of all the transactions. This makes it extremely challenging to detect. For example, a credit card fraud data set provided by Kaggle has 0.172% of the samples labeled as fraud[2].

An **early detection** of these frauds can help in a timely prevention of the fraudulent transactions.

1.1.3 Network Intrusion Detection

Attacks on computer systems and computer-networks are not unheard-of. Today most organizations, including hospitals, traffic control, and sensitive government bodies, are run on computers. Attacks on such systems can prove to be extremely fatal.

Due to its importance, this is an active field of research. To bolster the research, KDD-CUP'99 contest provided a network intrusion data set[3] from Defense Advanced Research Projects Agency (DARPA), an agency of the United States Department of Defense responsible for the development of emerging technologies for use by the military. The data included a wide variety of intrusions simulated in a military network

[2]`https://www.kaggle.com/mlg-ulb/creditcardfraud/`

[3]`http://kdd.ics.uci.edu/databases/kddcup99/kddcup99`

environment. Few examples of network attacks were: denial-of-service (dos), surveillance (probe), remote-to-local (r2l), and user-to-root (u2r). Among which, the u2r and r2l categories are intrinsically rare but fatal.

The objective in such systems are quick or **early detection** of such intrusions or attacks to prevent any catastrophic breach.

1.1.4 Detecting Emergencies

Audio-video surveillance is generally accepted by society today. The use of cameras for live traffic monitoring, parking lot surveillance, and public gatherings, is getting accepted as they make us feel safer. Similarly, placing cameras for video surveillance of our homes is becoming common.

Unfortunately, most such surveillance still depends on a human operator sifting through the videos. It is a tedious and tiring job—monitoring for events-of-interest that rarely occur. The sheer volume of these data impedes easy human analysis and, thus, necessitates automated predictive solutions for assistance.

TUT Rare Sound events 2017, a development data set[4], provided by the Audio Research Group at Tampere University of Technology is one such data set of audio recordings from home surveillance systems. It contains labeled samples for baby crying, glass breaking, and gunshot. The last two events being the rare events of concern (housebreak) requiring **early detection**.

1.1.5 Click vis-à-vis churn prediction

Digital industries, such as Pandora, eBay, and Amazon, rely heavily on subscriptions and advertisements. To maximize their revenue, the objective is to increase the clicks (on advertisements) while minimizing customers' churn.

A simple solution to increasing customer clicks would be to show more advertisements. But it will come at the expense of causing cus-

[4]`http://www.cs.tut.fi/sgn/arg/dcase2017/challenge/task-rare-sound-event-detection`

tomer churn. A churn is a customer leaving the website which could be in the form of exiting a transaction page or sometimes even unsubscribing. Churns, therefore, cost dearly to the companies.

To address this problem, companies attempt at a targeted marketing that maximizes the click probability while minimizing the churn probability by modeling customer behaviors.

Typically, customer behavior is a function of their past activities. Due to this, most research attempts at developing predictive systems that can **early predict** the possibility of click and churn to better capitalize on the revenue opportunities. However, both of these events are "rare" which makes it challenging.

1.1.6 Failures in Manufacturing

Today most manufacturing plants whether continuous or discrete run 24x7. Any downtime causes a significant cost to the plant. For instance, in 2006 the automotive manufacturers in the US estimated production line downtime costs at about $22,000 per minute or $1.3 million per hour (Advanced Technology Services 2006). Some estimates ran as high as $50,000 per minute.

A good portion of these downtimes is caused due to a *failure*. Failures occur in either machine parts or the product. For example, predicting and preventing machine bearing failure is a classical problem in the machining and automotive industries.

Another common problem on the product failures are found in pulp-and-paper industries. These plants continuously produce paper sheets. Sometimes sheet breakage happens that causes the entire process to stop. According to Bonissone et. al. (2002) Bonissone, Goebel, and Y.-T. Chen 2002, this causes an expected loss of $3 billion every year across the industry.

More broadly, as per a report by McKinsey in 2015 Manyika 2015, predictive systems have the potential to save global manufacturing businesses up to $630 billion per year by 2025.

These failures are a rare event from a predictive modeling point-of-view. For which, an **early prediction** of the failures for a potential prevention is the objective.

Outside of these, there are several other applications, such as stock market collapse, recession prediction, earthquake prediction, and so on.

None of these events should occur in an ideal world. We, therefore, **envision to have prediction systems to predict them in advance to either prevent or minimize the impact**.

1.2 A Working Example

A paper sheet-break problem in paper manufacturing is taken from Ranjan et al. 2018 as a working example in this book. The methods developed in the upcoming chapters will be applied on a sheet-break data set.

The problem formulation in the upcoming chapter will also refer to this problem. The sheet-break working example is described and used for developing a perspective. However, the formulation and developed methods apply to other rare event problems.

1.2.1 Problem Motivation

Paper sheet-break at paper mills is a critical problem. On average, a mill witnesses more than one sheet-break every day. The average is even higher for fragile papers, such as paper tissues and towels.

Paper manufacturing is a continuous process. These sheet-breaks are unwanted and costly hiccups in the production. Each sheet-break can cause downtime of an hour or longer. As mentioned in the previous section, these downtime cause loss of millions of dollars at a plant and billions across the industry. Even a small reduction in these breaks would lead to significant savings.

More importantly, fixing a sheet-break often requires an operator to enter the paper machine. These are large machines with some hazardous sections that pose danger to operators' health. Preventing sheet-break via predictive systems will make operators' work condition better and

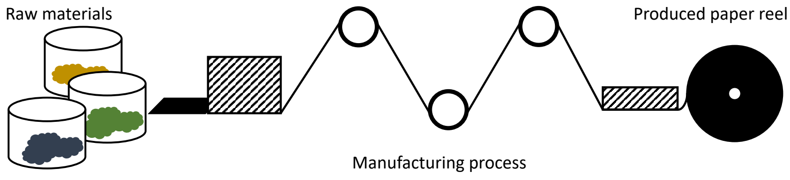


Figure 1.1. *A high-level schematic outline of a continuous paper manufacturing process.*

the paper production more sustainable.

1.2.2 Paper Manufacturing Process

Paper manufacturing machines are typically half a mile long. A condensed illustration of the machine and the process is given in Figure 1.1. As shown, the raw materials get into the machine from one end. They go through manufacturing processes of forming, press, drying, and calendaring in the same order. Ultimately, a large reel of the paper sheet is yielded at the other end. This process runs continuously.

In this continuous process, sometimes sheet tears are called **sheet-breaks** in the paper industry. There are several possible causes for a break. They could be instantaneous or gradual. Breaks due to instantaneous causes like something falling on the paper are difficult to predict. But gradual effects that lead to a break can be caught in advance. For example, in the illustrative outline in Figure 1.1 suppose one of the rollers on the right starts to rotate asynchronously faster than the one on its left. This asynchronous rotation will cause the sheet to stretch and eventually break. When noticed in time, an imminent break can be predicted and prevented.

1.2.3 Data Description

The sheet-break data set in consideration has observations from several sensors measuring the raw materials, such as the amount of pulp fiber, chemicals, etc., and the process variables, such as blade type, couch

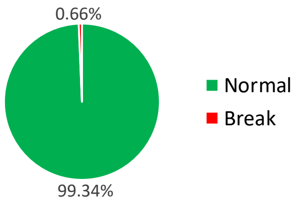


Figure 1.2. *Class distribution in paper sheet-break data.*

vacuum, rotor speed, etc.

The data contains the process status marked as: normal or break. The class distribution is extremely skewed in this data. Shown in Figure 1.2, the data has only 0.66% positive labeled samples.

1.3 Machine Learning vs. Deep Learning

Machine Learning is known for its simplicity especially in regards to its interpretability. Machine learning methods are, therefore, usually the first choice for most problems. Deep learning, on the other hand, provides more possibilities. In this section, these two choices are debated.

As noticed in § 1.2.3 we have an imbalanced binary labeled multivariate time series process. For over two decades, imbalanced binary classification has been actively researched. However, there are still several open challenges in this topic. Krawczyk 2016 discussed these challenges and the research progresses.

One major open challenge for machine learning methods is an apparent absence of a robust and simple modeling framework. The summarization of ML methods by Sun, Wong, and Kamel 2009 in Figure 1.3 makes this more evident.

As seen in the figure, there are several disjoint approaches. For each of these approaches, there are some unique methodological modifications. These modifications are usually not straightforward to implement.

Another critical shortcoming of the above methods is their limitations with multivariate time series processes.

To address this, there are machine learning approaches for multi-

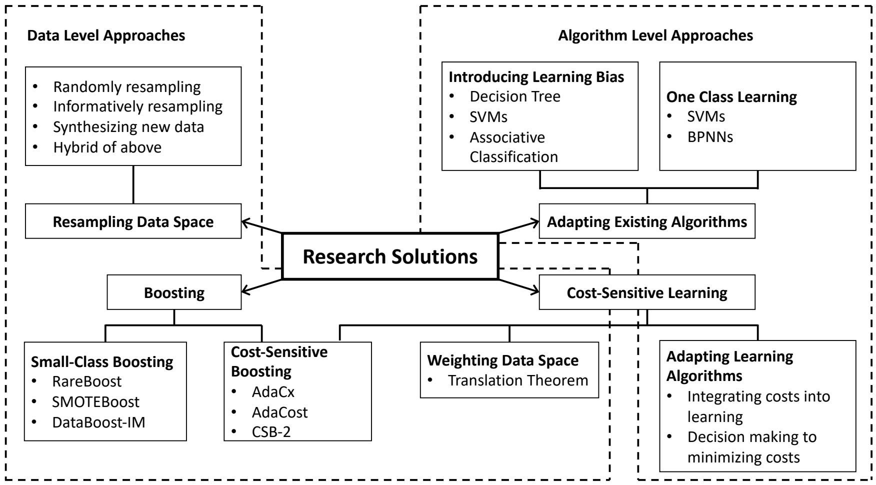


Figure 1.3. *A summary of research solutions in machine learning for imbalanced data classification (Sun, Wong, and Kamel 2009).*

variate time series classification. For example, see Batal et al. 2009; Orsenigo and Vercellis 2010; Górecki and Łuczak 2015. Unfortunately, these methods are not directly applicable to imbalanced data sets.

In sum, most of the related machine learning approaches are solving a part of an, "*imbalanced multivariate time series problem.*" A robust and easy-to-implement solution framework to solve the problem is, therefore, missing in machine learning.

Deep learning, on the other hand, provides a better possibility.

It is mentioned in § 2.2.3 that traditional oversampling and data augmentation techniques do not work well with extremely rare events. Fortunately, in the rest of the book, it is found that deep learning models do not necessarily require data augmentation.

Intuitively this could be logical. Deep learning models are inspired by the mechanisms of human brains. We, humans, do not require over-sampled rare events or objects to learn to distinguish them. For example, we do not need to see several Ferrari cars to learn how one looks like.

Similarly, deep learning models might learn to distinguish rare events from a few samples present in a large data set. The results in this book empirically support this supposition. But this is still a conjecture and the true reason could be different.

Importantly, there are architectures in deep learning that provide a simpler framework to solve a complex problem such as an imbalanced multivariate time series.

Given the above, deep learning methods are developed for rare event prediction in the subsequent chapters of this book.

1.4 In this Book

In the next chapter, a rare event problem is framed by elucidating its underlying process, the problem definition, and the objective. Thereafter, before getting to deep learning constructs, a reader is set up with TensorFlow in Chapter 3. Both these chapters can be skipped if the reader intends to only learn the deep learning theories.

The book then gets into deep learning topics starting with deconstructing multi-layer perceptrons in Chapter 4. They are made of *dense* layers—the most fundamental deep learning construct. More complex and advanced long- and short- term memory (LSTM) constructs are in Chapter 5. Convolutional layers, considered as the workhorse of deep learning, are in Chapter 6. Lastly, an elemental model in deep learning, autoencoders, are presented in Chapter 7.

The chapters uncover the core concepts behind these constructs. A simplified mathematics behind dense and other layers, state information flow in LSTM, filtration mechanism in convolution, and structural features in autoencoder are presented.

Some of the concepts are even rediscovered. For example,

- Chapter 4 draws parallels between a simple regression model and a multi-layer perceptron. It shows how nonlinear activations differentiates them by implicitly breaking down a complex problem into small pieces. The essential gradient properties of activations are laid. Moreover, customized activation is also implemented for research illustrations. Besides, the *dropout* concept which enhances MLP (and most other networks) is explained along with answering how to use it?

- LSTM mechanisms are one of the most mystical in deep learning. Chapter 5 deconstructs an LSTM cell to visualize the state information flow that preserves the memory. The secret behind it is the gradient transmission which is explained in detail. Moreover, the variants of LSTM available in TensorFlow such as *backward* and *bi-directional* are visually explained. Besides, LSTMs have a rich history with several variants. They are tabulated to compare against the variant in TensorFlow. This gives a perspective of the LSTM commonly in-use versus the other possibilities.

- Convolutional networks use *convolution* and *pooling* operations. They are simple operations yet make convolutional networks one of the most effective models. Chapter 6 explains both concepts with visual exemplifications and theoretical clarifications. The chapter goes into detail to explain convolutional and pooling properties such as parameter sharing, filtration, and invariance. Besides, the need for pooling to summarize convolution is statistically concretized. The statistics behind pooling also answer questions like why max-pool generally outperforms others, and what other pooling statistics are viable?

- Autoencoders are constructs for unsupervised, and semi-supervised learning. Chapter 7 explains them by drawing parallels with principal component analysis in machine learning. The comparison makes it easier to understand their internal mechanisms. Autoencoders' ability to learn specific tasks such as denoising or feature learning for classification by incorporating regularization are also presented. Besides, classifiers trained on encodings are developed.

Every chapter explains the concepts along with illustrating model implementations in TensorFlow. The implementation steps are also explained in detail to help a reader learn model building.

This book acknowledges that there is no perfect model. And, therefore, the book aims at showing **how** to develop the models to enable a reader to build his own best model.

> *Think of deep learning as an art of cooking. One way to cook is to follow a recipe. But when we learn* ***how*** *the food, the spices, and the fire behave, we make our creation. And an understanding of the* ***how*** *transcends the creation.*
>
> *Likewise, an understanding of the* ***how*** *transcends deep learning. In this spirit, this book presents the deep learning constructs, their fundamentals, and how they behave. Baseline models are developed alongside, and concepts to improve them are exemplified.*

Besides, the enormity of deep learning modeling choices can be overwhelming. To avoid that, we have a few rules-of-thumb before concluding a chapter. It lists the basics of building and improving a deep learning model.

In summary, deep learning offers a robust framework to solve complex problems. It has several constructs. Using them to arrive at the best model is sometimes difficult. The focus of this book is to understand these constructs to have a direction to develop effective deep learning models. Towards the end of the book, a reader would understand every construct and how to put them together.

Chapter 2

Rare Event Prediction

2.1 Rare Event Problem

The previous chapter emphasized the importance of rare event problems. This chapter formulates the problem by laying out the underlying statistical process, problem definition, and objectives. Moreover, the challenges in meeting the objectives are also discussed. The working example in § 1.2 is referred to during the formulation.

2.1.1 Underlying Statistical Process

First, the statistical process behind a rare event problem is understood. This helps in selecting an appropriate approach (see the discussion in § 1.3).

The process and data commonalities in the rare event examples in § 1.1 are

- time series,
- multivariate, and
- imbalanced binary labels.

Consider our working example of a sheet-break problem. It is from a continuous paper manufacturing process that generates a data stream.

This makes the underlying process a **stochastic time series**.

Additionally, this is a **multivariate** data streamed from multiple sensors placed in different parts of the machine. These sensors collect the process variables, such as temperature, pressure, chemical dose, etc.

Thus, at any time t, a vector of observations $\boldsymbol{x}_t$ is recorded. Here, $\boldsymbol{x}_t$ is a vector of length equal to the number of sensors and x_{it} the reading of the i-th sensor at time t. Such a process is known as a **multivariate time series**.

In addition to the process variables $\boldsymbol{x}_t$, a binary label y_t denoting the status of the process is also available. A positive y_t indicates an occurrence of the rare event. Also, due to the rareness of positive y_t's, the class distribution is imbalanced.

For instance, the labels in the sheet-break data denote whether the process is running normal ($y_t = 0$), or there is a sheet-break ($y_t = 1$). The samples with $y_t = 0$ and $y_t = 1$ is referred to as *negatively* and *positively* labeled data in the rest of the book. The former is the majority class and the latter minority.

Putting them together, we have an **imbalanced multivariate stochastic time series** process. Mathematically represented as, $(y_t, \boldsymbol{x}_t), t = 1, 2, \ldots$ where $y_t \in \{0, 1\}$ with $\sum \mathbb{1}\{y_t = 1\} \ll \sum \mathbb{1}\{y_t = 0\}$ and $\boldsymbol{x}_t \in \mathbb{R}^p$ with p being the number of variables.

2.1.2 Problem definition

Rare event problems demand an early detection or prediction to prevent the event or minimize its impact.

In literature, detection and prediction are considered as different problems. However, in the problems discussed here, early detection eventually becomes a prediction. For example, early detection of a condition that would lead to a sheet-break is essentially predicting an imminent sheet-break. This can, therefore, be formulated as a "prediction" problem.

An early prediction problem is predicting an event in advance. Suppose the event prediction is needed k time units in advance. This k should be chosen such that the prediction gives sufficient time to take an action against the event.

Mathematically, this can be expressed as estimating the probability of $y_{t+k} = 1$ using the information at and until time t, i.e.,

$$\Pr[y_{t+k} = 1 | \boldsymbol{x}_{t^-}] \tag{2.1}$$

where $\boldsymbol{x}_{t^-}$ denotes $\boldsymbol{x}$ before time t, i.e., $\boldsymbol{x}_{t^-} = \{\boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \ldots\}$.

Equation 2.1 also shows that this is a classification problem. Therefore, prediction and classification are used interchangeably in this book.

2.1.3 Objective

The objective is to **build a binary classifier to predict a rare event in advance**. To that end, an appropriate loss function and accuracy measures are selected for predictive modeling.

Loss function

There are a variety of loss functions. Among them, binary cross-entropy loss is chosen here.

Cross-entropy is intuitive and has the appropriate theoretical properties that make it a good choice. Its gradient lessens the vanishing gradients issue in deep learning networks. Moreover, from the model fitting standpoint, cross-entropy approximates the Kullback-Leibler divergence. Meaning, minimizing cross-entropy yields an approximate estimation of the "true" underlying process distributions[1].

It is defined as,

$$\begin{aligned} \mathcal{L}(\theta) = -y_{t+k} \log(\Pr[y_{t+k} = 1 | \boldsymbol{x}_{t-}, \theta]) - \\ (1 - y_{t+k}) \log(1 - \Pr[y_{t+k} = 1 | \boldsymbol{x}_{t-}, \theta]) \end{aligned} \tag{2.2}$$

where θ denotes the model.

[1]Refer to a recent paper at NIPS 2018 Zhang and Sabuncu 2018 for more details and advancements in cross-entropy loss.

Entropy means randomness. The higher the entropy the more the randomness. More randomness means a less predictable model, i.e., if the model is random it will make poor predictions.

Consider an extreme output of an arbitrary model: an absolute opposite prediction, e.g., estimating $\Pr[y = 1] = 0$ when $y = 1$. In such a case, the loss in Equation 2.2 will be, $\mathcal{L} = -1 * \log(0) - (1-1) * \log(1-0) = -1 * -\infty - 0 * 0 = +\infty$.

On the other extreme, consider an oracle model: makes absolute true prediction, i.e. $\Pr[y = 1] = 1$ when $y = 1$. In this case, the cross-entropy loss will become, $\mathcal{L} = -1 * \log(1) - (1-1) * \log(1-1) = 0$.

During model training, any arbitrary model is taken as a starting point. The loss is, therefore, high at the beginning. The model then trains itself to lower the loss. This is done iteratively to bring the cross-entropy loss from $+\infty$ towards 0.

Accuracy measures

Rare event problems have extremely imbalanced class distribution. The traditional *misclassification* accuracy metric does not work here.

This is because more than 99% of our samples are negatively labeled. A model that predicts everything, including all the minority $< 1\%$ positive samples, as negative is still $> 99\%$ accurate. Thus, a model that cannot predict any rare event would appear accurate. The area under the ROC curve (AUC) measure is also unsuitable for such extremely imbalanced problems.

In building a classifier, if there is any deviation from the usual it is useful to fall back to the *confusion matrix* and look at other accuracy measures drawn from it. A confusion matrix design is shown in Table 2.1.

The "actuals" and the "predictions" are along the rows and columns of the matrix. Their values are either negative or positive. As mentioned before, negative corresponds to the normal state of the process and is the majority class. And, positive corresponds to the rare event and is the minority class.

The actual negative or positive samples predicted as the same go on the diagonal cells of the matrix as *true negative* (TN) and *true positive* (TP), respectively. The other two possibilities are if an actual negative is predicted as a positive and the vice versa denoted as *false positive* (FP) and *false negative* (FN), respectively.

In rare event classifiers, the goal is inclined towards maximizing the true positives while ensuring it does not lead to excessive false predictions. In light of this goal, the following accuracy measures are chosen and explained vis-à-vis the confusion matrix.

- **recall**: the percentage of positive samples correctly predicted as one, i.e., $\frac{\text{TP}}{\text{TP} + \text{FN}}$. It lies between 0 and 1. A high recall indicates the model's ability to accurately predicting the minority class. A recall equal to one means the model could detect all of the rare events. However, this can also be achieved with a dummy model that predicts everything as a rare event. To counterbalance this, we also use f1-score.

- **f1-score**: a combination (harmonic mean) of precision[2] and recall. This score indicates the model's overall ability in predicting most of the rare events with as few false alerts as possible. It is computed as, $\frac{2}{\left(\frac{\text{TP}}{\text{TP}+\text{FN}}\right)^{-1} + \left(\frac{\text{TP}}{\text{TP}+\text{FP}}\right)^{-1}}$. The score lies between 0 and 1 with higher the better. If we have the dummy model favoring high recall by predicting all samples as positive (a rare event), the f1-score will counteract it by getting close to zero.

- **false positive rate (fpr)**: lastly, it is also critical to measure the false positive rate. Fpr is the percentage of false alerts, i.e., $\frac{\text{FP}}{\text{FP} + \text{TN}}$. An excess of false alerts makes us insensitive to the predictions. It is, therefore, imperative to keep the fpr as close to zero as possible.

[2]A ratio of the true positives overall predicted positives. The ratio lies between 0 to 1 with higher the better. This measure shows the model performance w.r.t. high true positives and low false positives. High precision is indicative of this and vice versa.

Table 2.1. Confusion matrix.

		Predicted	
		Negative	**Positive**
Actual	**Negative**	True Negative (TN)	False Positive (FP)
	Positive	False Negative (FN)	True Positive (TP)

** Negative: Normal process and the majority class.*
*** Positive: Rare event and the minority class.*

2.2 Challenges

The step after formulating a problem is to identify the modeling challenges. Challenges, if identified, enable a better modeling direction. It tells a practitioner the issues to address during the modeling.

A few acute challenges posed by a rare event problem are,

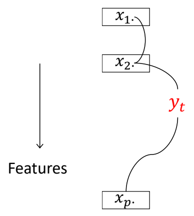
- high-dimensional multivariate time series process,
- early prediction, and
- imbalanced data.

2.2.1 High-dimensional Multivariate Time Series

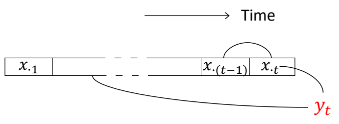
This is a mouthful and, hence, broken down to its elements for clarity. Earlier, § 2.1.1 mentioned that a rare event process is a multivariate time series. A multivariate process has multiple features (variables). Rare event problems typically have 10s to 100s of features which categorizes them as a high-dimensional process.

A **high-dimensional** process poses modeling challenges due to "spatial" relationships between the features. This is also known as *cross-correlations* in *space*. The term "space" is used because the features mentioned here are spread in a *space*.

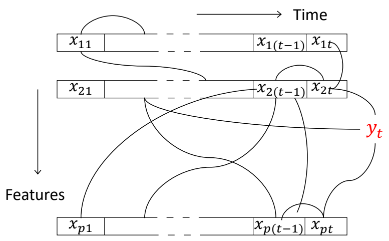
While this space is in a mathematical context, for an intuitive understanding, think of the sensors placed at different locations in space on a paper manufacturing machine and how they correlate with each other.



(a) *Space.*



(b) *Time.*



(c) *Space and time.*

Figure 2.1. *Spatio-temporal relationships.*

Figure 2.1a shows a visual illustration of spatial correlations and dependencies of features with the response. Since every feature can be related to each other, the possible spatial relationships increase exponentially with the number of features, p.

The issue with excess spatial relationships with large p is that they may induce spurious dependencies in the model. For example, in the figure x_1 and x_2 are correlated and x_2 is related to y. But the model may select x_1 instead of x_2 or both x_1 and x_2.

Such spurious dependencies often cause high model variance and overfitting, and therefore, should be avoided.

Time series, also referred to as **temporal processes**, pose another critical challenge. Temporal features exhibit correlations with themselves (autocorrelation[3]) and long-short term dependencies with

[3]The correlation of a variable with a lagged value of itself. For example, x_t being correlated with its previous value x_{t-1}.

the response.

For illustration, Figure 2.1b shows x_{t-1} and x_t are autocorrelated. It is important to isolate and/or account for these dependencies in the model to avoid high model variance.

Additionally, we see that x_t and an early observation close to x_1 are related to y_t indicative of short- and long-term dependencies, respectively. While estimating the short-term dependencies are relatively simpler, long-term dependencies are quite challenging to derive.

But long-term dependencies are common and should not be ignored. For example, in a paper manufacturing process, a few chemicals that are fed at an early stage of the production line affects the paper quality hours later. In some processes, such long-term relationships are even a few days apart.

A major issue in drawing these long-term dependencies is that any prior knowledge on the lag with which a feature affects the response may be unavailable. In absence of any prior knowledge, we have to include all the feature lags in a model. This blows up the size of the model and makes its estimation difficult.

A high-dimensional and time series process are individually challenging. If they are together, their challenges get multiplied.

Figure 2.1c is visually illustrating this complexity. As we can see here, in a high-dimensional time series process the features have, a. relationships with themselves, b. long- and short-term dependencies, and c. cross-correlations in space and time. Together, all of these are called **spatio-temporal relationships**.

As shown in the figure, the dependencies to estimate in a spatio-temporal structure grows exponentially with more features and a longer time horizon. Modeling such processes is, therefore, extremely challenging.

2.2.2 Early Prediction

As mentioned in § 2.1.2, we need to predict an event in advance. Ideally, we would like to predict it well in advance. However, the farther we are

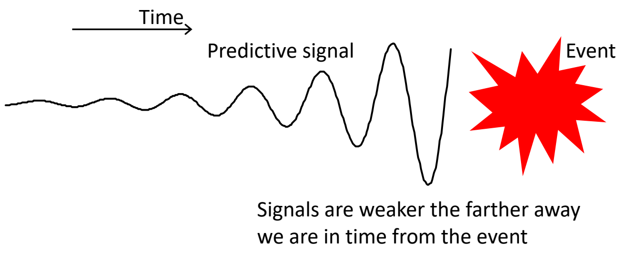

Time
Predictive signal
Event
Signals are weaker the farther away we are in time from the event

Figure 2.2. *Challenge with early prediction.*

from an event the weaker are the predictive signals and, therefore, results in poor predictions.

Visually illustrated in Figure 2.2, the red zagged mark indicates the occurrence of an event and the horizontal axis is time. As shown, the predictive signal will be the most dominant closest to the event. The farther we are in a time the weaker is the signal.

This is generally true for any event irrespective of whether it is rare or not. However, due to the dramatic adverse impact of a rare event, it is critical to be able to predict it well in advance. And the challenge is the more in advance prediction we want the harder it gets for a model.

2.2.3 Imbalanced Data

A rare event problem by definition has imbalanced data. The issue with imbalanced data sets is that the model is heavily skewed towards the majority class. In such situations, learning the underlying patterns predictive of the rare event becomes difficult.

A typical resolution to address the imbalanced data issue is oversampling. Oversampling, however, does not work in most rare event problems. This is because, a. extreme imbalance in class distribution, and b. time series data.

The former typically make resampling approaches inapplicable because it requires excessive resampling to balance the data. This causes the model to get extremely biased.

The latter prohibits the usage of more sophisticated oversampling techniques, such as SMOTE. This is because samples "interpolation" for data synthesis done in SMOTE takes into account the spatial aspect of the process but not temporal (refer to Appendix L). Due to this, the synthesized data does not necessarily accentuate or retain the underlying predictive patterns.

The temporal aspect of our problem also prohibits the use of most other data augmentation approaches. In problems like image classification, techniques such as reorientation and rotation augment the data.

But with temporal features, any such augmentation distorts the data and, consequently, the inherent dependencies. Time series data augmentation methods using slicing and time warping are available but they do not work well with multivariate time series.

In the rest of the book, appropriate modeling directions to address these challenges are discussed.

Chapter 3

Setup

TensorFlow and the working example is set up in this chapter. The chapter begins with the reasoning for choosing TensorFlow followed by laying out the steps to set it up.

After this setup, the chapter describes the paper sheet-break problem, a data set for it, and a few basic preprocessing steps. The following sections are intentionally succinct to keep the setup process short.

3.1 TensorFlow

There are a bunch of platform choices for deep learning. For example, Theano, PyTorch, and TensorFlow. Among them, the book uses the recent TensorFlow 2x. The section begins with its reasoning and then the installation steps in Ubuntu, Mac, and Windows.

Why TensorFlow 2x?

TensorFlow 2x was released in 2019 and is expected to change the landscape of deep learning. It has made,

- model building simpler,
- production deployment on any platform more robust, and

- enables powerful experimentation for research.

With these, TF 2x is likely to propel deep learning to mainstream applications in research and industry alike.

TF 2x has Keras API integrated into it. Keras is a popular high-level API for building and training deep learning models. In regards to TF 2x and Keras, it is important to know,

- TF 1.10+ also supports Keras. But in TF 2x Keras is also integrated with the rest of the TensorFlow platform. TF 2x is providing a single high-level API to reduce confusion and enable advanced capabilities.
- `tf.keras` in TF 2x is different from the `keras` library from `www.keras.io`. The latter is an independent open source project that precedes TF 2x and was built to support multiple backends, viz. TensorFlow, CNTK, and Theano.

It is recommended to read TensorFlow 2018 and TensorFlow 2019 from the TensorFlow team for more details and benefits of TF 2x. In summary, they state that TF 2x has,

- brought an ease-of-implementation,
- immense computational efficiency, and
- compatibility with most mobile platforms, such as Android and iOS, and embedded edge systems like Raspberry Pi and edge TPUs.

Achieving these was difficult before. As TF 2x brings all of these benefits, this book chose it.

Fortunately, the installation has become simple with TF 2x. In the following, the installation prerequisites, the installation, and testing steps are given.

Note: Using Google Colab environment is an alternative to this installation. Google Colab is generally an easier way to work with TensorFlow. It is a notebook on Google Cloud with all the TensorFlow requisites pre-installed.

3.1.1 Prerequisites

TensorFlow is available only for Python 3.5 or above. Also, it is recommended to use a virtual environment. Steps for both are presented here.

Install Python

Anaconda

Anaconda with Jupyter provides a simple approach for installing Python and working with it.

Installing Anaconda is relatively straightforward. Follow this link `https://jupyter.org/install` and choose the latest Python.

System

First, the current Python version (if present) is looked for.

```
$ python --version
Python 3.7.1
```

or,

```
$ python3 --version
Python 3.7.1
```

It this version is less than 3.5, Python can be installed as shown in the listing below.

Listing 3.1. Install Python.

```
$ brew update
$ brew install python # Installs Python 3
$ sudo apt install python3-dev python3-pip
```

Install Virtual Environment

An optional requirement is using a virtual environment. Its importance is in the next section, § 3.1.2. It can be installed as follows[1].

[1]Refer: `https://packaging.python.org/guides/installing-using-pip-and-virtual-environments/`

Mac/Ubuntu

```
$ python3 -m pip install --user virtualenv
```

Windows

```
py -m pip install --user virtualenv
```

Now the system ready to install TensorFlow.

3.1.2 TensorFlow 2x Installation

Instantiate and activate a virtual environment

The setup will begin by creating a virtual environment.

Why is the virtual environment important? A virtual environment is an isolated environment for Python projects. Inside a virtual environment one can have a completely independent set of packages (dependencies) and settings that will not conflict with anything in other virtual environment or with the default local Python environment.

This means that different versions of the same package can be used in different projects at the same time on the same system but in different virtual environments.

Mac/Ubuntu

Create a virtual environment called `tf_2`.

```
$ virtualenv --system-site-packages -p python3 tf_2
$ source tf_2/bin/activate  # Activate the
    virtualenv
```

The above command will create a virtual environment `tf_2`. In the command,

- `virtualenv` will create a virtual environment.
- `-system-site-packages` allow the projects within the virtual environment `tf_2` to access the global site-packages. The default

setting does not allow this access (`-no-site-packages` were used before for this default setting but now deprecated.)

- `-p python3` is used to set the Python interpreter for `tf_2`. This argument can be skipped if the `virtualenv` was installed with Python 3. By default, that is the python interpreter for the virtual environment. Another option for setting Python 3.x as an interpreter is `$ virtualenv -system-site-packages -python=python3.7 tf_2`. This gives more control.

- `tf_2` is the name of the virtual environment created. Any other name of a user's choice can also be given. The physical directory created at the location of the virtual environments will bear this name. The (`/tf_2`) directory will contain a copy of the Python compiler and all the packages installed afterward.

Anaconda

With Anaconda, the virtual environment is created using Conda as,

```
$ conda create -n tf_2
$ conda activate tf_2 # Activate the virtualenv
```

The above command will also create a virtual environment `tf_2`. Unlike before, pre-installation of the virtual environment package is not required with Anaconda. The in-built `conda` command provides this facility.

Understanding the command,

- `conda` can be used to create virtual environments, install packages, list the installed packages in the environment, and so on. In short, `conda` performs operations that `pip` and `virtualenv` do. While `conda` replaces `virtualenv` for most purposes, it does not replace `pip` as some packages are available on `pip` but not on `conda`.

- `create` is used to create a virtual environment.

- `-n` is an argument specific to `create`. `-n` is used to name the virtual environment. The value of `n`, i.e. the environment name, here is `tf_2`.

- Additional useful arguments: similar to `--system-site-packages` in `virtualenv`, `--use-local` can be used.

Windows

Create a new virtual environment by choosing a Python interpreter and making a `.\tf_2` directory to retain it.

```
C:\> virtualenv --system-site-packages -p python3 ./
   tf_2
```

Activate the virtual environment:

```
C:\> .\tf_2\Scripts\activate
```

Install packages within a virtual environment without affecting the host system setup. Start by upgrading pip:

```
(tf_2) C:\> pip install --upgrade pip
(tf_2) C:\> pip list  # show packages installed
   within the virtual environment
```

After the activation in any system, the terminal will change to this `(tf_2) $`.

Install TensorFlow

Upon reaching this stage, TensorFlow installation is a single line.

```
(tf_2) $ pip install --upgrade tensorflow
```

For a GPU or any other version of TensorFlow replace `tensorflow` in this listing with one of the following.

- `tensorflow` âĂŤLatest stable release (2.x) for CPU-only (recommended for beginners).
- `tensorflow-gpu` âĂŤLatest stable release with GPU support (Ubuntu and Windows).
- `tf-nightly` âĂŤPreview build (unstable). Ubuntu and Windows include GPU support.

3.1.3 Testing

Quick test

An instant test for the installation through the terminal is,

```
(tf_2) $ python -c "import tensorflow as tf; x =
    [[2.]]; print('tensorflow version', tf.
    __version__); print('hello, {}'.format(tf.matmul
    (x, x)))"
tensorflow version 2.0.0
hello, [[4.]]
```

The output should have the TensorFlow version and a simple operation output as shown here.

Modeling test

More elaborate testing is done by modeling a simple deep learning model with MNIST (`fashion_mnist`) image data.

```
import tensorflow as tf
layers = tf.keras.layers
import numpy as np
print(tf.__version__)
```

The `tf.__version__` should output `tensorflow 2.x`. If the version is older, check the installation, or the virtual environment should be revisited.

In the following, the `fashion_mnist` data is loaded from the TensorFlow library and preprocessed.

```
mnist = tf.keras.datasets.fashion_mnist
(x_train, y_train), (x_test, y_test) = mnist.
    load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

A simple deep learning model is now built and trained on the data.

```
model = tf.keras.Sequential()
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
model.compile(optimizer='adam',
 loss='sparse_categorical_crossentropy',
 metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)
```

Note that this model is only for demonstration and, therefore, trained on just five epochs.

Lastly, the model is evaluated on a test sample.

```
predictions = model.predict(x_test)
predicted_label = class_names[np.argmax(predictions
    [0])]
print('Actual label:', class_names[y_test[0]])
print('Predicted label:', predicted_label)
# Actual label: Ankle boot
# Predicted label: Ankle boot
```

The prediction output confirms a complete installation.

3.2 Sheet Break Problem Dataset

An anonymized data from the paper manufacturing plant for sheet break is taken from link[2] provided in Ranjan et al. 2018.

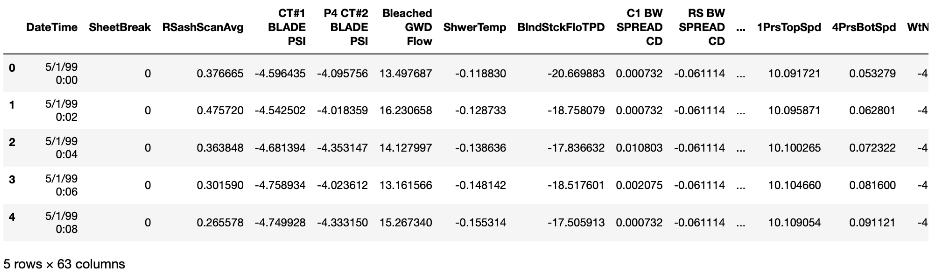
It contains data at two minutes frequency with the time information present in the `DateTime` column. The system's status with regards to *normal* versus *break* is present in the `SheetBreak` column with the corresponding values as 0 and 1.

These remaining columns include timestamped measurements for the predictors. The predictors include raw materials, such as the amount of pulp fiber, chemicals, etc., and the process variables, such as blade type, couch vacuum, rotor speed, etc.

The steps for loading and preparing this data is shown in Listing 3.2 below.

Listing 3.2. Loading data.

[2] `http://bit.ly/2uCIJpG`



	DateTime	SheetBreak	RSashScanAvg	CT#1 BLADE PSI	P4 CT#2 BLADE PSI	Bleached GWD Flow	ShwerTemp	BlndStckFloTPD	C1 BW SPREAD CD	RS BW SPREAD CD	...	1PrsTopSpd	4PrsBotSpd	WtN
0	5/1/99 0:00	0	0.376665	-4.596435	-4.095756	13.497687	-0.118830	-20.669883	0.000732	-0.061114	...	10.091721	0.053279	-4
1	5/1/99 0:02	0	0.475720	-4.542502	-4.018359	16.230658	-0.128733	-18.758079	0.000732	-0.061114	...	10.095871	0.062801	-4
2	5/1/99 0:04	0	0.363848	-4.681394	-4.353147	14.127997	-0.138636	-17.836632	0.010803	-0.061114	...	10.100265	0.072322	-4
3	5/1/99 0:06	0	0.301590	-4.758934	-4.023612	13.161566	-0.148142	-18.517601	0.002075	-0.061114	...	10.104660	0.081600	-4
4	5/1/99 0:08	0	0.265578	-4.749928	-4.333150	15.267340	-0.155314	-17.505913	0.000732	-0.061114	...	10.109054	0.091121	-4

5 rows × 63 columns

Figure 3.1. *A snapshot of the sheet-break data.*

```
 1  import pandas as pd
 2
 3  df = pd.read_csv("data/processminer-sheet-break-rare
        -event-dataset.csv")
 4  df.head(n=5)  # visualize the data.
 5
 6  # Hot encoding
 7  hotencoding1 = pd.get_dummies(df['Grade&Bwt'])
 8  hotencoding1 = hotencoding1.add_prefix('grade_')
 9  hotencoding2 = pd.get_dummies(df['EventPress'])
10  hotencoding2 = hotencoding2.add_prefix('eventpress_'
        )
11
12  df = df.drop(['Grade&Bwt', 'EventPress'], axis=1)
13
14  df = pd.concat([df, hotencoding1, hotencoding2],
        axis=1)
15
16  # Rename response column name for ease of
        understanding
17  df = df.rename(columns={'SheetBreak': 'y'})
```

A snapshot of the data is shown in Figure 3.1.

The data contains two categorical features which are hot-encoded in Line 7-10 followed by dropping the original variables. The binary response variable `SheetBreak` is renamed to `y`.

The processed data is stored in a `pandas` data frame `df`. This will be further processed with *curve shifting* and *temporalization* for modeling in the next chapters.

Deactivate Virtual Environment

Before closing, deactivate the virtual environment.

Mac/Ubuntu

```
(tf_2) C:\> deactivate
```

Windows

```
(tf_2) $ deactivate  # don't exit until you're done
   using TensorFlow
```

Conda

```
(tf_2) $ source deactivate
```

or,

```
(tf_2) $ source deactivate
```

This concludes the TensorFlow setup. Now we will head to model building.

Chapter 4

Multi-layer Perceptrons

4.1 Background

Perceptrons were built in the 1950s. And they proved to be a powerful classifier at the time.

A few decades later, researchers realized stacking multiple perceptrons could be more powerful. That turned out to be true and multi-layer perceptron (MLP) was born.

A single perceptron works like a neuron in a human brain. It takes multiple inputs and, like a neuron emits an electric pulse, a perceptron emits a binary pulse which is treated as a response.

The **neuron**-like behavior of perceptrons and an MLP being a **network** of perceptrons perhaps led to the term *neural networks* come forth in the early days.

Since its creation, neural networks have come a long way. Tremendous advancements in various architectures, such as convolutional neural networks (CNN), recurrent neural networks (RNN), etc., have been made.

Despite all the advancements, MLPs are still actively used. It is the "hello world" to deep learning. Similar to *linear regression* in machine learning, MLP is one of the immortal methods that remain active due to its robustness.

It is, therefore, logical to start exploring deep learning with multi-layer perceptrons.

Multi-layer perceptrons are complex nonlinear models. This chapter unfolds MLPs to simplify and explain its fundamentals in § 4.2. The section shows that an MLP is a collection of simple regression models placed on every node in each layer. How they come together with non-linear activations to deconstruct and solve complex problems becomes clearer in this section.

An end-to-end construction of a network and its evaluation is then given in § 4.3-4.4. § 4.3 has granular details on data preparation, viz. curve shifting for early prediction, data splitting, and features scaling. Thereafter, every construction element, e.g., layers, activations, evaluation metrics, and optimizers, are explained in § 4.4.

Dropout is a useful technique (not limited to multi-layer perceptrons) that resolves **co-adaptation** issue in deep learning. The co-adaptation issue is explained in § 4.5.1. How dropout addresses it and regularizes a network is in § 4.5.2. The use of dropout in a network is then illustrated in § 4.5.3.

Activation functions elucidated in § 4.7 are one of the most critical constructs in deep learning. Network performances are usually sensitive to activations due to their vanishing or exploding gradients. An understanding of activations is, therefore, essential for constructing any network. § 4.7.1 and 4.7.2 explain vanishing and exploding gradients issues, and their connection with activations. The story of activations laying discoveries such as non-decaying gradient, saturation region, and self-normalization is in § 4.7.3 and 4.7.4.

Besides, a few customizations in TensorFlow implementations are sometimes required to attempt novel ideas. § 4.8.1 shows an implementation of a new *thresholded exponential linear unit* (`telu`) activation. Moreover, metrics such as f1-score and false positive rate useful for evaluating imbalanced classifiers are unavailable in TensorFlow. They are custom implemented in § 4.8.2. This section also clarifies that metrics available outside TensorFlow such as in `sklearn` cannot be directly used during model training.

Lastly, deep learning networks have several configurations and nu-

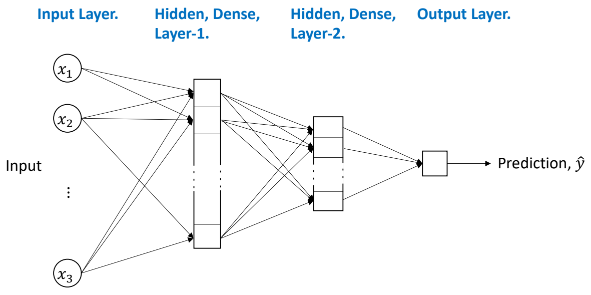


Figure 4.1. *A high-level representation of a multi-layer perceptron.*

merous choices for them, e.g., number of layers, their sizes, activations on them, and so on. To make a construction simpler, the chapter concludes with a few rules-of-thumb in § 4.10.

4.2 Fundamentals of MLP

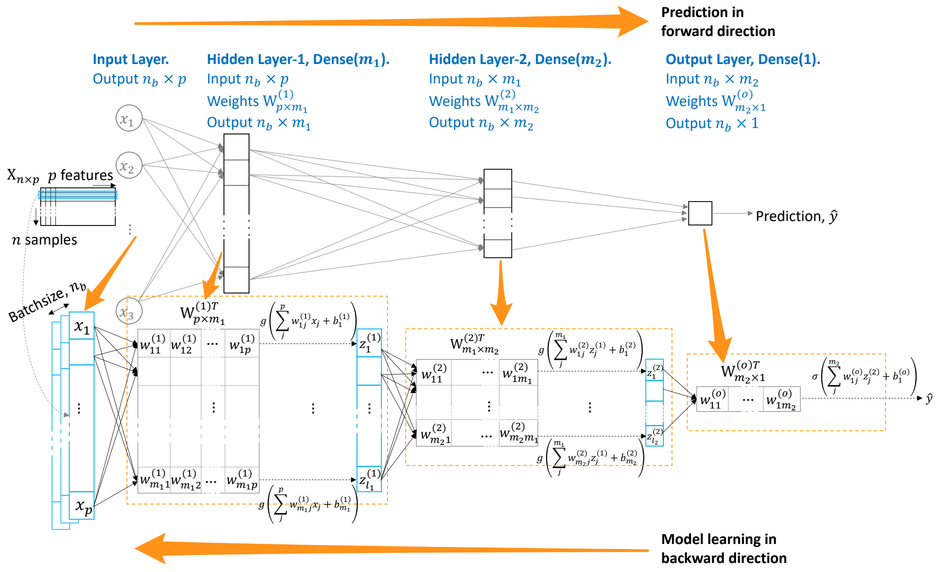
Multi-layer perceptrons are possibly one of the most visually illustrated neural networks. Yet most of them lack a few fundamental explanations. Since MLPs are the foundation of deep learning, this section attempts at providing a clearer perspective.

A typical visual representation of an MLP is shown in Figure 4.1. This high-level representation shows the feed-forward nature of the network. In a feed-forward network, information between layers flows in only forward direction. That is, information (features) learned at a layer is not shared with any prior layer[1].

The abstracted network illustration in Figure 4.1 is unwrapped to its elements in Figure 4.2. Each element, its interactions, and implementation in the context of TensorFlow are explained step-by-step in the following.

1. The process starts with a data set. Suppose there is a data set

[1]Think of a bi-directional LSTM discussed in § 5.6.4 as a counter example to a feed-forward network.



Prediction in forward direction
Input Layer.
Output $n_b \times p$
Hidden Layer-1, Dense(m_1).
Input $n_b \times p$
Weights $W^{(1)}_{p\times m_1}$
Output $n_b \times m_1$
Hidden Layer-2, Dense(m_2).
Input $n_b \times m_1$
Weights $W^{(2)}_{m_1\times m_2}$
Output $n_b \times m_2$
Output Layer, Dense(1).
Input $n_b \times m_2$
Weights $W^{(o)}_{m_2\times 1}$
Output $n_b \times 1$
$X_{n\times p}$ p features
n samples
Batchsize, n_b
Prediction, $\hat{y}$
$g\left(\sum_j^p w^{(1)}_{1j}x_j + b^{(1)}_1\right)$
$g\left(\sum_j^{m_1} w^{(2)}_{1j}z^{(1)}_j + b^{(1)}_1\right)$
$\sigma\left(\sum_j^{m_2} w^{(o)}_{1j}z^{(2)}_j + b^{(o)}_1\right)$
Model learning in backward direction

Figure 4.2. *An unwrapped visual of a multi-layer perceptron. The input to the network is a batch of samples. Each sample is a feature vector. The hidden layers in an MLP are* `Dense`*. A* `Dense` *layer is characterized by a weight matrix W and bias b. They perform simple affine transformations (dot product plus bias: $XW + b$). The affine transforms extract features from the input. The transforms are passed through an activation function. The activations are nonlinear. Its nonlinearity enables the network to implicitly divide a complex problem into arbitrary sub-problems. The outputs of these sub-problems are put together by the network to infer the final output $\hat{y}$.*

shown at the top left, $X_{n \times p}$, with n samples and p features.

2. The model ingests a randomly selected batch during training. The batch contains random samples (rows) from X unless otherwise mentioned. The batch size is denoted as n_b here[2].

3. By default, the samples in a batch are processed independently. Their sequence is, therefore, not important.

4. The input batch enters the network through an input layer. Each node in the input layer corresponds to a sample feature. Explicitly defining the input layer is optional but it is done here for clarity.

5. The input layer is followed by a stack of hidden layers till the last (output) layer. These layers perform the "complex" interconnected nonlinear operations. Although perceived as "complex," the underlying operations are rather simple arithmetic computations.

6. A hidden layer is a stack of computing nodes. Each node extracts a feature from the input. For example, in the sheet-break problem, a node at a hidden-layer might determine whether the rotations between two specific rollers are out-of-sync or not[3]. A node can, therefore, be imagined as solving one arbitrary sub-problem.

7. The stack of output coming from a layer's nodes is called a *feature map* or *representation*. The size of the feature map, also equal to the number of nodes, is called the layer size.

8. Intuitively, this feature map has results of various "sub-problems" solved at each node. They provide predictive information for the next layer up until the output layer to ultimately predict the response.

9. Mathematically, a node is a perceptron made of weights and bias parameters. The weights at a node are denoted with a vector $\boldsymbol{w}$ and a bias b.

[2]Batch size is referred to as `None` or `?` in `model.summary()`, e.g., see Figure 4.4.

[3]Another example from the face-recognition problem about solving an arbitrary sub-problem at a node is determining whether eyebrows are present or not.

10. All the input sample features go to a node. The input to the first hidden layer is the input data features $\boldsymbol{x} = \{x_1, \ldots, x_p\}$. For any intermediate layer it is the output (feature map) of the previous layer, denoted as $\boldsymbol{z} = \{z_1, \ldots, z_m\}$, where m is the size of the prior layer.

11. Consider a hidden layer l of size m_l in the figure. A node j in the layer l performs a feature extraction with a dot product between the input feature map $\boldsymbol{z}^{(l-1)}$ and its weights $\boldsymbol{w}_j^{(l)}$, followed by an addition with the bias b_j. Generalizing this as,

$$z_j^{(l)} = \sum_i^{m_{l-1}} z_i^{(l-1)} w_{ij}^{(l)} + b_j, \; j = 1, \ldots, m_l \tag{4.1}$$

where $z_i^{(l-1)}, i = 1, \ldots, m_{l-1}$ is a feature outputted from the prior layer $l-1$ of size m_{l-1}.

12. The step after the linear operation in Equation 4.1 is applying a nonlinear *activation* function, denoted as g. There are various choices for g. Among them, a popular activation function is *rectified linear unit* (`relu`) defined as,

$$g(z) = \begin{cases} z, & \text{if } z > 0. \\ 0, & \text{otherwise.} \end{cases} \tag{4.2}$$

As shown in the equation, the function is nonlinear at 0.

13. The operations in Equation 4.1 and 4.2 can be combined for every nodes in a layer as,

$$\boldsymbol{z}^{(l)} = g(\boldsymbol{z}^{(l-1)} W^{(l)} + \boldsymbol{b}^{(l)}) \tag{4.3}$$

where $\boldsymbol{z}^{(l)}$ is the feature map, $W^{(l)} = [\boldsymbol{w}_1^{(l)}; \ldots; \boldsymbol{w}_{m_l}^{(l)}]$ is the stack of weights of all m_l nodes in the layer, $\boldsymbol{z}^{(l-1)}$ is the input to the layer which is $\boldsymbol{x}$, if $l = 1$, and $\boldsymbol{b}^{(l)} = \{b_1^{(l)}, \ldots, b_{m_l}^{(l)}\}$ is the bias.

14. Equation 4.3 is applied to the batch of n_b input samples. For a `Dense` layer this is a matrix operation shown in Equation 4.4 below,

$$Z^{(l)}_{n_b \times m_l} = g(Z^{(l-1)}_{n_b \times m_{l-1}} W^{(l)}_{m_{l-1} \times m_l} + \boldsymbol{b}^{(l)}_{m_l}). \tag{4.4}$$

The output $Z^{(l)}_{n_b \times m_l}$ of the equation is the g-activated *affine* transformation of the input features.

The operation here is called a ***`tensor`*** *operation. Tensor is a term used for any multi-dimensional matrix. Tensor operations are computationally efficient (especially in GPUs), hence, most steps in deep learning layers use them instead of iterative loops.*

15. It is the nonlinear activation in Equation 4.4 that **dissociates** the feature map of one layer from another. Without the activation, the feature map outputted from every layer will be just a **linear** transformation of the previous. This would mean the subsequent layers are **not** providing any additional information for a better prediction. An algebraic explanation of this is given in Equation A.1 in the Appendix.

16. Activation functions, therefore, play a **major** role. An appropriate selection of activation is critical. In addition to being nonlinear, an activation function should also have a non-decaying gradient, saturation region, and a few other criteria discussed in § 4.7.

17. The operation in Equation 4.4 is carried forward in each layer till the output layer to deliver a prediction. The output is delivered through a different activation denoted as σ. The choice of this activation is dictated by the response type. For example, it is *sigmoid* for a binary response.

18. The model training starts with randomly initializing the weights and biases at the layers. The response is predicted using these

parameters. The prediction error is then propagated back into the model to update the parameters to a value that reduces the error. This iterative training procedure is called *backpropagation*.

19. Put simply, backpropagation is an **extension** of the iterative stochastic gradient-descent based approach to train multi-layer deep learning networks. This is explained using a single-layer perceptron, also otherwise known as, **logistic regression**. A gradient-descent based estimation approach for logistic regression is shown in p.120-121 in Hastie, Tibshirani, and Friedman 2009[4]. The estimation equation in the reference[5] is rephrased to a simplified context here,

$$\begin{aligned} \theta^{new} &\leftarrow \theta^{old} - \eta \nabla_\theta \\ &\leftarrow \theta^{old} - \eta X^T (y - \hat{y}) \end{aligned} \tag{4.5}$$

where η is a multiplier[6], X is a random sample, ∇_θ is the first-order gradient of the loss with respect to the parameter θ, and θ is the weight and bias parameters. As shown, the gradient ∇_θ for the logistic regression contains $(y - \hat{y})$ which is the **prediction error**. This implies that the prediction error is propagated back to update the model parameters θ.

20. This estimation approach for logistic regression is extended in *backpropagation* for a multi-layer perceptron. In backpropagation, this process is repeated on every layer. It can be imagined as updating/learning one layer at a time in the reverse order of prediction.

21. The learning is done iteratively over a user-defined number of epochs. An epoch is a learning period. Within each epoch, the stochastic gradient-descent based learning is performed iteratively over randomly selected batches.

[4]The approach in Hastie, Tibshirani, and Friedman 2009 also used the second-derivative or the Hessian matrix. However, in most backpropagation techniques only first derivatives are used.

[5]Equation 4.26 in Hastie, Tibshirani, and Friedman 2009.

[6]Equal to the Hessian in Newton-Raphson algorithm.

22. After training through all the epochs the model is expected to have learned the parameters that have minimal prediction error. This minimization is, however, for the training data and is not guaranteed to be the global minima. Consequently, the performance of the test data is not necessarily the same.

The fundamentals enumerated above will be referred to during the modeling in the rest of the chapter.

Deep learning models are powerful because it breaks down a problem into smaller sub-problems and combines their results to arrive at the overall solution. This fundamental ability is due to the presence of the nonlinear activations.

Intuitively, ***backpropagation*** *is an extension of stochastic gradient-descent based approach to train deep learning networks.*

The next section starts the preparation for modeling.

4.3 Initialization and Data Preparation

4.3.1 Imports and Loading Data

Modeling starts with the ritualistic library imports. Listing 4.1 shows all the imports and also a few declarations of constants, viz. random generator seeds, the data split percent, and the size of figures to be plotted later.

Listing 4.1. Imports for MLP Modeling.

```python
import tensorflow as tf

from tensorflow.keras import optimizers
from tensorflow.keras.models import Model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import AlphaDropout

import pandas as pd
import numpy as np

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

from imblearn.over_sampling import SMOTE
from collections import Counter

import matplotlib.pyplot as plt
import seaborn as sns

# user-defined libraries
import datapreprocessing
import performancemetrics
import simpleplots

from numpy.random import seed
seed(1)

from pylab import rcParams
rcParams['figure.figsize'] = 8, 6

SEED = 123 #used to help randomly select the data
    points
DATA_SPLIT_PCT = 0.2
```

A few user-defined libraries: `datapreprocessing`, `performancemetrics`, and `simpleplots` are loaded. They have custom functions for preprocessing, evaluating models, and visualizing results, respectively. These libraries are elaborated in this and upcoming chapters (also refer to Appendix B and C).

Next, the data is loaded and processed the same way as in Listing 3.2 in the previous chapter. The listing is repeated here to avoid any confusion.

Listing 4.2. Loading data for MLP Modeling.

```
# Read the data
df = pd.read_csv("data/processminer-sheet-break-rare
    -event-dataset.csv")

# Convert Categorical column to hot dummy columns
hotencoding1 = pd.get_dummies(df['Grade&Bwt'])
hotencoding1 = hotencoding1.add_prefix('grade_')
hotencoding2 = pd.get_dummies(df['EventPress'])
hotencoding2 = hotencoding2.add_prefix('eventpress_'
    )

df=df.drop(['Grade&Bwt', 'EventPress'], axis=1)

df=pd.concat([df, hotencoding1, hotencoding2], axis
    =1)

# Rename response column name for ease of
    understanding
df=df.rename(columns={'SheetBreak':'y'})
```

4.3.2 Data Pre-processing

The objective, as mentioned in Chapter 2, is to predict a rare event in advance to prevent it, or its consequences.

From a modeling standpoint, this translates to teaching the model to identify a *transitional* phase that would lead to a rare event.

For example, in the sheet-break problem, a transitional phase could be the speed of one of the rollers (in Figure 1.1) drifting away and rising in comparison to the other rollers. Such asynchronous change stretches the paper sheet. If this continues, the sheet's tension increases and ultimately causes a break.

The sheet break would typically happen a few minutes after the drift starts. Therefore, if the model is taught to identify the start of the drift it can predict the break in advance.

One simple and effective approach to achieve this is *curve shifting*.

Curve Shifting

Curve Shifting here should not be confused with *curve shift* in Economics or *covariate shift* in Machine Learning. In Economics, a curve shift is a phenomenon of the Demand Curve changing without any price change. Covariate shift or data shift in ML implies a change in data distribution due to a shift in the process. Here it means aligning the predictors with the response to meet a certain modeling objective.

For early prediction, curve shifting moves the labels early in time. Doing so, the samples before the rare event get labeled as one. These prior samples are assumed to be the transitional phase that ultimately leads to the rare event.

Providing a model with these positively labeled transitional samples teaches it to identify the "harbinger" of a rare event in time. This, in effect, is an early prediction.

For early prediction, teach the model to identify the transitional phase.

Given the time series sample, $(y_t, \boldsymbol{x}_t), t = 1, 2, \ldots$, curve shifting will,

1. label the k prior samples to a positive sample as one. That is, $y_{t-1}, \ldots, y_{t-k} \leftarrow 1$, if $y_t = 1$.
2. Followed by dropping the sample $(y_t, \boldsymbol{x}_t)$ for the $y_t = 1$ instance.

Due to the relabeling in the first bullet, the model learns to predict the rare-event up to $t + k$ times earlier.

Besides, the curve shifting drops the original positive sample to avoid teaching the model to predict the rare event when it has already happened. Referring to Figure 2.2, the signal is highest when the event has occurred. Keeping these samples in the training data will be overpowering on the transitional samples. Due to this, it is likely that the model does not learn predicting the event ahead.

The `curve_shift` function in the `datapreprocessing` library performs this operation. It labels the samples adjacent to the positive labels as one. The number of adjacent samples to label is equal to the argument `shift_by`. A negative `shift_by` relabels the preceding samples[7]. Moreover, the function drops the original positive sample after the relabeling.

Listing 4.3. Curve-shifting.

```
 1 # Sort by time.
 2 df['DateTime'] = pd.to_datetime(df.DateTime)
 3 df = df.sort_values(by='DateTime')
 4 
 5 # Shift the response column y by 2 rows to do a 4-
     min ahead prediction.
 6 df = datapreprocessing.curve_shift(df, shift_by=-2)
 7 
 8 # Drop the time column.
 9 df = df.drop(['DateTime'], axis=1)
10 
11 # Converts df to numpy array
12 X = df.loc[:, df.columns != 'y'].values
13 y = df['y'].values
14 
15 # Axes lengths
16 N_FEATURES = X.shape[1]
```

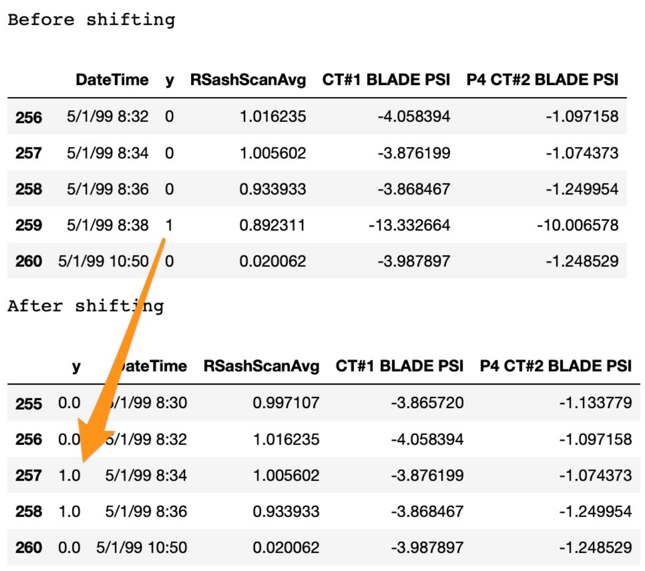
Line 6 in Listing 4.3 applies the curve shift with `shift_by=-2`. This relabels two samples prior to a sheet break as positive, i.e., the transitional phase leading to a break. Since the samples are at two minutes interval, this shift is of four minutes. Thus, the model trained on this curve-shifted data can do up to 4-minute ahead sheet break prediction.

While this is reasonable for this problem, the requirements could be different for different problems. The `shift_by` parameter should be set accordingly. Furthermore, for advanced readers, the curve shift definition is given in Appendix B for details and customization, if needed.

The effect of the curve shifting is visualized in Figure 4.3. The figure shows sample 259 is originally a positive sample. After applying `curve_shift` with `shift_by=-2`, the preceding two samples 257-258 are relabeled as `1`. And, the original positive sample 259 is dropped.

Thereafter, the `DateTime` column is not needed and, therefore, dropped.

[7] A positive `shift_by` relabels the succeeding samples to $y_t = 1$ as one.



```
Before shifting
```

	DateTime	y	RSashScanAvg	CT#1 BLADE PSI	P4 CT#2 BLADE PSI
256	5/1/99 8:32	0	1.016235	-4.058394	-1.097158
257	5/1/99 8:34	0	1.005602	-3.876199	-1.074373
258	5/1/99 8:36	0	0.933933	-3.868467	-1.249954
259	5/1/99 8:38	1	0.892311	-13.332664	-10.006578
260	5/1/99 10:50	0	0.020062	-3.987897	-1.248529

```
After shifting
```

	y	DateTime	RSashScanAvg	CT#1 BLADE PSI	P4 CT#2 BLADE PSI
255	0.0	5/1/99 8:30	0.997107	-3.865720	-1.133779
256	0.0	5/1/99 8:32	1.016235	-4.058394	-1.097158
257	1.0	5/1/99 8:34	1.005602	-3.876199	-1.074373
258	1.0	5/1/99 8:36	0.933933	-3.868467	-1.249954
260	0.0	5/1/99 10:50	0.020062	-3.987897	-1.248529

Figure 4.3. *An illustration of Curve Shifting. In this example, a dataframe is curve shifted (ahead) by two time units (`curve_shift(df, shift_by=-2)`). After the shift, the updated dataframe has the labels of two prior samples to a positive sample updated to 1 and the original positive sample is dropped. This procedure can also be interpreted as treating the samples prior to a positive sample as transitional phases. Using `curve_shift()`, these transitional samples are re-labeled as positives.*

The dataframe is partitioned into the features array `X` and the response `y` in lines 12-13. Lastly, the shape of the features array is recorded in `N_FEATURES`. This becomes a global constant that will be used in defining the `input_shape` during modeling.

Data Splitting

The importance of splitting a data set into *train*, *valid*, and *test* sets is well known. It is a necessary modeling tradition for the right reasons, which are briefly described below.

With the split data,

1. a model is trained on the *train* set, and
2. the model's performance is validated on the *valid* set.
3. Steps 1-2 are repeated for a variety of models and/or model configurations. The one yielding the best performance on the *valid* set is chosen as the final model. The performance of the final model on the *test* set is then recorded.

The *test* set performance is a "robust" indicator. While the *train* set performance is unusable due to a usual overfitting, the *valid* set is used for model selection and, therefore, is biased towards the selected model. Consequently, only the *test* set performance gives a reliable estimate.

In Listing 4.4, the data is randomly split into a *train*, *valid*, and *test* making 64%, 16%, and 20% of the original data set, respectively.

Listing 4.4. Data splitting.

```
# Divide the data into train, valid, and test
X_train, X_test, y_train, y_test =
    train_test_split(np.array(X),
                     np.array(y),
                     test_size=DATA_SPLIT_PCT,
                     random_state=SEED)
X_train, X_valid, y_train, y_valid =
    train_test_split(X_train,
                     y_train,
                     test_size=DATA_SPLIT_PCT,
                     random_state=SEED)
```

A common question in splitting a time series is: should it be at random or in the order of time?

Splitting time series in the order of time is appropriate when,

1. the data set is arranged as time-indexed tuples $(response : y_t,\ features : \boldsymbol{x}_t)$. And,
2. the model has temporal relationships, such as $y_t \sim y_{t-1} + \ldots + x_t + x_{t-1} + \ldots$. That is, the y_t is a function of the prior y's and x's.

In such a model, maintaining the time order is necessary for the data splits.

However, suppose the temporal (time-related) features are included in the tuple as $(response : y_t,\ features : \{y_{t-1}, \ldots, \boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \ldots\})$. This makes a self-contained sample with the response and (temporal) features. With this, the model does not require to keep track of the time index of the samples.

For multivariate time series, it is recommended to prepare such self-contained samples. In MLP, the temporal features can be added during the data preprocessing. But learning the temporal patterns are left to recurrent and convolutional neural networks in the upcoming chapters where temporally contained samples are used.

Here, the data samples are modeled in their original form $(y_t, \boldsymbol{x}_t)$, where y_t is one in the transitional phase and zero, otherwise. No additional time-related features are to be modeled. Therefore, the model is agnostic to the time order. And, hence, the data set can be split at random.

Features Scaling

Virtually every problem has more than one feature. The features can have a different range of values. For example, a paper manufacturing process has temperature and moisture features. Their units are different due to which their values are in different ranges.

These differences may not pose theoretical issues. But, in practice, they cause difficulty in model training typically by converging at local minimas.

Feature scaling is, therefore, an important preprocessing step to address this issue. Scaling is generally linear[8]. Among the choices of linear scaling functions, the standard scaler shown in Listing 4.5 is appropriate for the unbounded features in our problem.

Listing 4.5. Features Scaling.

```
# Scaler using the training data.
scaler = StandardScaler().fit(X_train)

X_train_scaled = scaler.transform(X_train, scaler)
X_valid_scaled = scaler.transform(X_valid, scaler)
X_test_scaled = scaler.transform(X_test, scaler)
```

As shown in the listing, the `StandardScaler` is fitted on the *train* set. The fitting process computes the means ($\bar{\boldsymbol{x}}_{train}$) and standard deviation ($\boldsymbol{\sigma}_{train}$) from the *train* samples for each feature in $\boldsymbol{x}$. This is used to transform each of the sets as per Equation 4.6.

$$\boldsymbol{x} \leftarrow \frac{\boldsymbol{x} - \bar{\boldsymbol{x}}_{train}}{\boldsymbol{\sigma}_{train}} \tag{4.6}$$

Another popular scaling method is `MinMaxScaler`. However, they work better in and became more popular through the image problems in deep learning. A feature in an image has values in a fixed range of $(0, 255)$. For such bounded features, `MinMaxScaler` is quite appropriate.

🔔 *`StandardScaler` is appropriate for unbounded features such as sensor data. However, `MinMaxScaler` is better for bounded features such as in image detection.*

[8]There are a few nonlinear feature scaling methods to deal with feature outliers. However, it is usually recommended to deal with the outliers separately and work with linear scaling to not disturb the original distribution of the data.

4.4 MLP Modeling

In this section, a multi-layer perceptron model is constructed step-by-step. Each modeling step is also elucidated conceptually and programmatically.

4.4.1 Sequential

TensorFlow provides a simple-to-implement API for constructing deep learning models. There are three general approaches,

- sequential,
- functional, and
- model sub-classing.

The ease of their use is in the same order. Most of the modeling requirements are covered by *sequential* and *functional*.

Sequential is the simplest approach. In this approach, models that have a linear stack of layers and the layers communicate sequentially are constructed. Models in which layers communicate non-sequentially (for example, residual networks) cannot be modeled with a sequential approach. Functional or model sub-classing is used in such cases.

MLPs are sequential models. Therefore, a sequential model is initialized as shown in Listing 4.6.

Listing 4.6. Creating a Sequential object.

```
model = Sequential()
```

The initialization here is creating a `Sequential` object. `Sequential` inherits the `Model` class in TensorFlow, and thereby, inherits all the training and inference features.

4.4.2 Input Layer

The model starts with an input layer. No computation is performed at this layer. Still, this plays an important role.

An input layer can be imagined as a "gate" to the model. The gate has a defined shape. This shape should be consistent with the input sample. This layer has two functions,

- not allow a sample to get through if its shape is not consistent, and
- communicate the shape of the inputted batch to the next layer.

Listing 4.7. Input layer in a Sequential model.

```
model.add(Input(shape=(N_FEATURES, )))
```

The input layer is added to the `model` in Listing 4.7. It takes an argument `shape` which is a tuple of the shape of the input. The tuple contains the length of each axes of the input sample and an empty last element.

Here the input has only one axis for the features (shown in Figure 4.2) with a length `N_FEATURES` defined in Listing 4.3. In the case of multi-axes inputs, such as images and videos, the tuple will have more elements.

The last (empty) element corresponds to the batch size. The batch size is defined during the model fit and is automatically taken by the model. The empty element in the tuple can be seen as a placeholder.

Explicitly defining the input layer is optional. In fact, it is common to define the input shape in the first computation layer, e.g. as `Dense(..., input_shape=(N_FEATURES, ))`. It is still explicitly defined here for clarity.

4.4.3 Dense Layer

A dense layer is one of the primary layers in deep learning. It is used in MLPs and most other deep learning architectures.

Its primality can be attributed to its simplicity. A linearly activated dense layer is simply an *affine* transformation of the inputs. It will be explained in the future chapters that such affine transformations make model construction for different architectures, such as LSTM autoencoder, easier.

Moreover, as opposed to most other layers, a (non)linear dense layer provides a simple structure to find a relationship between the features and response in the same space.

An MLP is a stack of dense layers. That is, from hidden to output all are dense layers[9].

The number of hidden layers is a model configuration. As a rule-of-thumb, it is recommended to begin with two hidden layers as a baseline. They are added to the model in Listing 4.8.

Listing 4.8. Hidden Layers in a Sequential model.

```
model.add(Dense(32, activation='relu', name='
    hidden_layer_1'))
model.add(Dense(16, activation='relu', name='
    hidden_layer_2'))
```

The size of a layer is the first argument. The number of nodes (denoted as `units` in TensorFlow) in the layer is the same as its size (see Figure 4.2).

The size is a configuration property. It should be set around half of the number of input features. As a convention the size should be taken from a geometric series of 2: a number in $\{1, 2, 4, 8, 16, 32, \ldots\}$.

The input sample has 69 features, therefore, the first dense layer is made of size 32. This also means the input to the second layer has 32 features, and thus, its size is set as 16.

Following these conventions are optional but help in streamlined model construction. These conventions are made keeping in account the insensitivity of deep learning models towards minor configuration changes.

Deep learning models are generally insensitive to minor changes in a layer size. Therefore, it is easier to follow a rule-of-thumb for configuring layer sizes.

Activation is the next argument. This is an important argument

[9]An MLP with only **one** dense layer (the output layer) is the same as a logistic regression model.

as the model is generally sensitive to any ill selection of activation. `relu` is usually a good first choice for the hidden layers.

Appropriate choice of activation is essential because models are sensitive to them. ***Relu*** *activation is a good default choice for hidden layers.*

The `name` argument, in the end, is optional. It is added for better readability in the *model summary* shown in Figure 4.4. And, if needed, to draw layer attributes, such as layer weights, using the name property.

4.4.4 Output Layer

The output layer in most deep learning networks is a dense layer. This is due to dense layer's *affine* transformation property which is usually required at the last layer. In an MLP, it is a dense layer by design.

The output layer should be consistent with the response's size just like the input layer must be consistent with the input sample's size.

In a classification problem, the size of the output layer is equal to the number of classes/responses. Therefore, the output dense layer has a unit size in a binary classifier (`size=1`) in Listing 4.9.

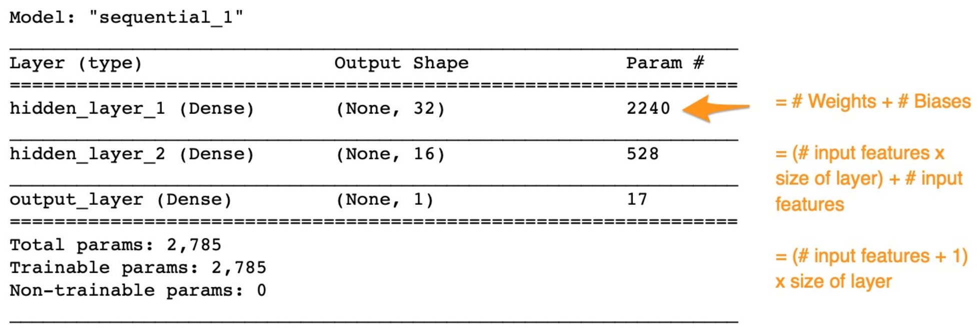
Listing 4.9. Output Layer in a Sequential model.

```
model.add(Dense(1, activation='sigmoid', name='
    output_layer'))
```

Also, the activation on this layer is dictated by the problem. For regression, if the response is in $(-\infty, +\infty)$ the activation is set as `linear`. In binary classifiers it is `sigmoid`; and `softmax` for multi-class classifiers.

4.4.5 Model Summary

At this stage, the model network has been constructed. The model has layers from input to output with hidden layers in between. It is useful to visualize the summary of the network.



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
hidden_layer_1 (Dense)       (None, 32)                2240
_________________________________________________________________
hidden_layer_2 (Dense)       (None, 16)                528
_________________________________________________________________
output_layer (Dense)         (None, 1)                 17
=================================================================
Total params: 2,785
Trainable params: 2,785
Non-trainable params: 0
_________________________________________________________________
```

= # Weights + # Biases
= (# input features x size of layer) + # input features
= (# input features + 1) x size of layer

Figure 4.4. *MLP Baseline Model Summary.*

Listing 4.10 displays the model summary in a tabular format shown in Figure 4.4.

Listing 4.10. Model Summary.

```
model.summary()
```

In the summary, `Model: "sequential_‘x’"` tells this is a `Sequential` TensorFlow model object. The layers in the network are displayed in the same order as they were added.

Most importantly, the shape of each layer and the corresponding number of parameters are visible. The number of parameters in a dense layer can be derived as: $weights = input\ features \times size\ of\ layer$ plus $biases = input\ features$. However, a direct visibility of the layer-wise and an overall number of parameters help to gauge the shallowness or massiveness of the model.

The end of the summary shows the total number of parameters and its breakdown as *trainable* and *non-trainable* parameters. All the weight and bias parameters on the layers are trainable parameters. They are trained (iteratively estimated) during the model training.

Examples of non-trainable parameters are the network topology configurations, such as the number of hidden layers and their sizes. Few other non-trainable parameters are in layers such as `Dropout` and `BatchNormalization`. A `Dropout` parameter is preset while `BatchNormalization` parameters, e.g., the batch mean and variance, are not "trained" but derived during the estimation.

In fact, in some cases, an otherwise trainable `Dense` layer is deliberately made non-trainable by
`model.get_layer(layerName).trainable = False`. For example, upon using a pre-trained model and training only the last layer.

However, in most models like here, all the parameters are trainable by default.

4.4.6 Model Compile

So far, the model is constructed and visually inspected in the model summary. This is like setting the layers of a cake on a baking tray and inspecting it before placing it in the oven. But before it goes in the oven there is an oven configuration step: set the right mode and temperature.

Similarly, in TensorFlow, before the model is fed in the machine for training there is a configuration step called *model compilation*.

This is done using `model.compile` in Listing 4.11.

Listing 4.11. Model Compile.

```
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy',
                       tf.keras.metrics.Recall(),
                       performancemetrics.F1Score(),
                       performancemetrics.
                          FalsePositiveRate()]
                )
```

The `model.compile` function has two purposes. First, verify any network architecture flaws, such as inconsistencies in input and output of successive layers. An error is thrown if there is any inconsistency.

And, second, define the three primary arguments: `optimizer`, `loss`, and `metrics` (there are other optional arguments not covered here). Each of these essential arguments is explained in the following.

- `optimizer`. A variety of optimizers are available with Tensor-

Flow[10]. Among them, `adam` is a good first choice. Quoted from its authors in Kingma and Ba 2014,

> "(Adam) is designed to combine the advantages of two recently popular methods: AdaGrad (Duchi et al., 2011 Duchi, Hazan, and Singer 2011), which works well with sparse gradients, and RMSProp (Tieleman & Hinton, 2012 G. Hinton et al. 2012), which works well in on-line and non-stationary settings..."

Almost every optimizer developed for deep learning are gradient-based. *Adam*, among them, is computationally and memory efficient. Moreover, it is less sensitive to its hyper-parameters. These attributes make *adam* an appropriate choice for deep learning.

The original adam paper (Kingma and Ba 2014) is simple-to-read and is recommended for more details.

`Adam` is a robust optimizer and, hence, a default choice.

- `loss`. The choice of a loss function depends on the problem. For example, in regression `mse` is a good choice while `binary_crossentropy` and `categorical_crossentropy` work for binary and multi-class classifiers, respectively.

  A list of loss functions for different needs are provided by TensorFlow[11]. For the binary classification problem here, `binary_crossentropy` loss is taken. This loss function is shown in Equation 2.2 and succinctly expressed again as

$$\mathcal{L}(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\Big(y_i \log(p_i) + (1 - y_i)\log(1 - p_i)\Big) \tag{4.7}$$

[10] `https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/optimizers`

[11] `https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/losses`

where $y_i \in 0, 1$ are the true labels and $p_i = \Pr[y_i = 1]$ is the predicted probability for $y_i = 1$. Looking closely at the equation, we can see that the loss is minimum when there are perfect predictions, i.e., if $p_i \to 1 | y_i = 1$ **and** $p_i \to 0 | y_i = 0$.

Also, it can be noted in Equation 4.7 that the cross-entropy loss is continuous and differentiable. Thus, it works well with the gradient descent based backpropagation estimation methods.

Cross entropy loss is continuous and differentiable and, hence, works well with gradient-based back-propagation method in deep learning.

- `metrics`. Unlike the `optimizer` and `loss`, metrics are not directly related to model training. Rather, the metrics are for evaluating the model performance during training and inferencing.

  Metrics are unrelated to training because their values, whether good or poor, do not impact the model parameter update during the training. The training focuses only on optimizing (generally minimizing) the loss function. The metrics are an outcome of this process.

  In regression, `RootMeanSquaredError()` is an appropriate metric. In classification, the `accuracy` that shows the percent of correct classification is generally used. However, for imbalanced data sets the objective is to have a high *recall* and *f1-score*, and a low *false positives rate* (fpr) (described in § 2.1.3).

  TensorFlow provides an in-built definition for recall (`Recall()`) but falls shy for the latter two. Therefore, custom f1-score and fpr metrics are constructed and elaborated in § 4.8.2. Also, the inability to use outside functions becomes clearer there.

The model compilation is the first stage-gate towards model completion. A successful model compile is a moment to cherish before heading to the next step: model fit.

4.4.7 Model Fit

Model fitting is a step towards the "moment of truth." This is the time when the model performance is seen.

Listing 4.12. Model Fit.

```
history = model.fit(x=X_train_scaled,
                    y=y_train,
                    batch_size=128,
                    epochs=100,
                    validation_data=(X_valid_scaled,
                        y_valid),
                    verbose=1).history
```

The primary arguments in `model.fit` is `x`, `y`, `batch_size`, and `epochs`.

- `x`: It is the training features X in a numpy array or pandas data frame. It is recommended to be scaled. In this model, `StandardScaler` scaled features are used.

- `y`: It is the response $\boldsymbol{y}$ aligned with the features X, i.e., the i-th row in X should correspond to the i-th element in $\boldsymbol{y}$. It is a one-dimensional array for a single class (response) model. But `y` can be a multi-dimensional array for a multi-class (response) model.

- `batch_size`: The model trains its parameters batch-by-batch using a batch gradient descent based approach. This approach is an extension of stochastic gradient descent (SGD) which uses only one sample at a time. While the batch size can be set to one like in SGD, deep learning optimization with one sample at a time is noisy and can take longer to converge. Batch gradient descent overcomes this issue while still having the benefits of SGD.

  Similar to the hidden layer sizes, the model is not extremely sensitive to minor differences in batch size. Therefore, it is easier to choose the batch size from the geometric series of 2. For balanced data sets, it is suitable to have a small batch, e.g., 16. However, for imbalanced data sets, it is recommended to have a larger batch size. Here it is taken as 128.

- `epochs`: Epochs is the number of iterations of model fitting. The number of epochs should be set such that it ensures model convergence. One indication of model convergence is a stable loss, i.e., no significant variations or oscillations. In practice, it is recommended to start with a small epoch for testing and then increase it. For example, here the epochs were set as 5 first and then made 100. The model training can also be resumed from where it was left by rerunning `model.fit`.

If `model.fit` *is rerun without re-initializing or redefining the model, it starts from where the previous fit left. Meaning, the model can be incrementally trained over separate multiple runs of* `model.fit`.

An additional but optional argument is `validation_data`. Validation data sets can be either passed here or an optional argument `validation_split` can be set to a float between 0 and 1. Although validation is not mandatory for model fitting, it is a good practice. Here, the validation data sets are passed through the `validation_data` argument.

Listing 4.12 executes the model fit. The `fit()` function returns a list `history` containing the model training outputs in each epoch. It contains the values for the loss, and all the metrics mentioned in `model.compile()`. This list is used to visualize the model performance in the next section.

4.4.8 Results Visualization

Visualizing the results is a natural next step after model fitting. Visualizations are made for,

- **loss**. The progression of loss over the epochs tells about the model convergence. A stable and decreasing loss for the training data shows the model is converging.

  The model can be assumed to have converged if the loss is not changing significantly towards the ending epochs. If it is still decreasing, the model should be trained with more epochs.

An oscillating loss, on the other hand, indicates the training is possibly stuck in local minima. In such a case, a reduction in the optimizer learning rate and/or a change in batch size should be tried.

A stabilized decreasing loss is also desirable in the validation data. It shows the model is robust and not overfitted.

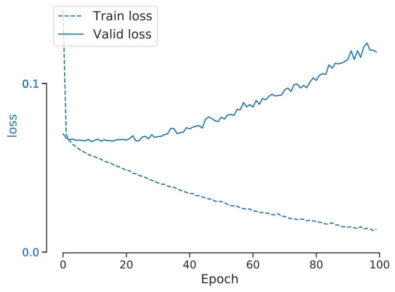
- **metrics**. Accuracy metrics on the validation data is more relevant than training. The accuracy-related metrics should improve as the model training progresses through the epochs. This is visualized in the metrics' plots.

  If the metrics have not stabilized and still improving, the model should be trained with more epochs. Worse, if the metrics are deteriorating with the epochs then the model should be diagnosed.

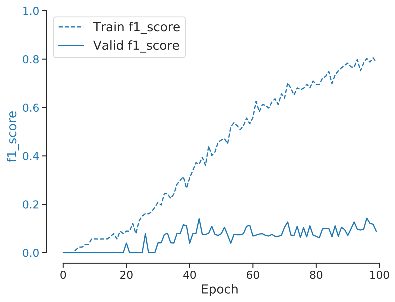
A few custom plotting functions are defined in Appendix C to visualize the results. Using them the results are plotted (and saved to a file) in Listing 4.13.

Listing 4.13. MLP baseline model results.

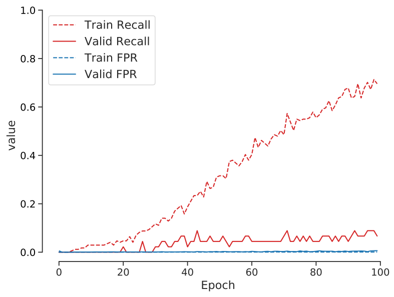
```
# Plotting loss and saving to a file
plt, fig =
  simpleplots.plot_metric(history,
    metric='loss')
fig.savefig('mlp_baseline_loss.pdf',
  bbox_inches='tight')

# Plotting f1score and saving to a file
plt, fig =
  simpleplots.plot_metric(history,
    metric='f1_score', ylim=[0., 1.])
fig.savefig('mlp_baseline_f1_score.pdf',
  bbox_inches='tight')

# Plotting recall and fpr, and saving to a file
plt, fig =
  simpleplots.plot_model_recall_fpr(history)
fig.savefig('mlp_baseline_recall_fpr.pdf',
  bbox_inches='tight')
```



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure 4.5. *MLP baseline model results.*

The loss is shown in Figure 4.5a. The loss is stable and almost plateaued for the training data. However, the validation loss is increasing indicating possible overfitting.

Still the performance metrics of the validation set in Figure 4.5b-4.5c are reasonable. F1-score is more than 10%. The recall is around the same and the false positive rate is close to zero.

Completing a baseline model is a major milestone. As mentioned before, the multi-layer perceptron is the "hello world" to deep learning. A fair performance of the baseline MLP shows there are some predictive patterns in the data. This is a telling that with model improvement and different networks a usable predictive model can be built.

The next section will attempt at some model improvements for multi-layer perceptrons while the upcoming chapters will construct more intricate networks, such as recurrent and convolutional neural networks.

4.5 Dropout

A major shortcoming of the baseline model was overfitting. Overfitting is commonly due to a phenomenon found in large models called **co-adaptation**. This can be addressed with **dropout**. Both, the co-adaptation issue and its resolution with dropout, are explained below.

4.5.1 What is Co-Adaptation?

If all the weights in a deep learning network are learned together, it is usual that some of the nodes have more predictive capability than the others.

In such a scenario, as the network is trained iteratively these powerful (predictive) nodes start to suppress the weaker ones. These nodes usually constitute a fraction of all. But over many iterations, only these powerful nodes are trained. And the rest stop participating.

This phenomenon is called co-adaptation. It is difficult to prevent with the traditional $\mathcal{L}_1$ and $\mathcal{L}_2$ regularization. The reason is that they also regularize based on the predictive capability of the nodes. As a result, the traditional methods become close to deterministic in choosing and rejecting weights. And, thus, a strong node gets stronger and the weak get weaker.

A major fallout of co-adaptation is: expanding the neural network size does not help.

If co-adaptation is severe, enlarging the model does not add to learning more patterns. Consequently, neural networks' size and, thus, capability get limited.

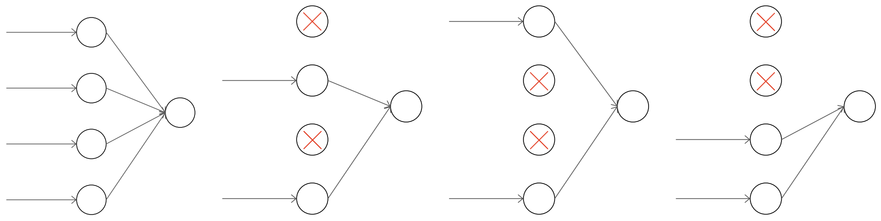
This had been a serious issue in deep learning for a long time. Then, in around 2012, the idea of *dropout*—a new regularization approach—emerged.

Dropout resolved co-adaptation. And naturally, this revolutionized deep learning. With dropout, deeper and wider networks were possible.

One of the drivers for the deep learning successes experienced today is attributed to dropout.

4.5.2 What Is Dropout?

Dropout changed the approach of learning weights. Instead of learning all the network weights together, dropout trains a subset of them in a batch training iteration.



(a) *Learning all weights.*

(b) *Learning a subset of all weights with dropout.*

Figure 4.6. *Illustrating the difference in weight learning with and without Dropout. Without dropout (left), weights on all the connections are learned together. Due to this, if a few connections are stronger than others (because of initialization or the nature of the data) the other connections become dormant. With dropout (right), connections are randomly dropped during each training iteration. As a result, no connection gets a chance to maintain dominance. Consequently, all the weights are appropriately learned.*

Figure 4.6a-4.6b illustrates the model weights training (updates) during a batch iteration. Here a simple example to train weights of four nodes is shown. The usual training without dropout is in Figure 4.6a. In this, all the nodes are active. That is, all the weights will be trained together.

On the other hand, with dropout, only a subset of nodes are kept active during batch learning. The three illustrations in Figure 4.6b correspond to three different batch iterations (refer to the iteration levels in Figure 4.14). In each batch iteration, half of the nodes are switched off while the remaining are learned. After iterating through all the batches the weights are returned as the average of their batch-wise estimations.

This technique acts as network regularization. But familiarity with traditional methods might make dropout appear not a regularization. Yet, there are some commonalities.

Like $\mathcal{L}_1$ regularization pushes the small weights to zero, dropout pushes a set of weights to zero. Still, there is an apparent difference: $\mathcal{L}_1$ does a data-driven suppression of weights while dropout does it at random.

Nevertheless, dropout is a regularization technique. It is closer to an $\mathcal{L}_2$ regularization. This is shown mathematically by Pierre and Peter in Baldi and Sadowski 2013. They show that under linearity (activation) assumptions the loss function with dropout (Equation 4.8 below) has the same form as $\mathcal{L}_2$ regularization.

$$\mathcal{L} = \frac{1}{2}\Big(t - (1-p)\sum_{i=1}^{n} w_i x_i\Big)^2 + \underbrace{p(1-p)\sum_{i=1}^{n} w_i^2 x_i^2}_{\text{Regularization term.}} \tag{4.8}$$

where p is the dropout rate.

The dropout rate is the fraction of nodes that are dropped at a batch iteration. For example, p is 0.5 in the illustration in Figure 4.6b.

The regularization term in Equation 4.8 has a penalty factor $p(1-p)$. The factor $p(1-p)$ is maximum when $p = 0.5$. Therefore, the dropout regularization is the largest at $p = 0.5$.

Dropout is a regularization technique equivalent to $\mathcal{L}_2$ regularization under linearity assumptions.

A dropout rate $p = 0.5$ *is an ideal choice for maximum regularization.*

Therefore, a dropout rate of 0.5 is usually a good choice for hidden layers. If a model's performance deteriorate with this dropout rate it is usually better to increase the layer size instead of reducing the rate.

Dropout, if applied on the input layer, should be kept small. A rule-of-thumb is 0.2 or less. Dropout at the input layer means not letting a subset of the input features into the training iteration. Although arbitrarily dropping features is practiced in a few bagging methods in machine learning, it generally does not bring significant benefit to deep learning. And, therefore, dropout at the input layer should be avoided.

Dropout at the input layer is better avoided.

These inferences drawn from Equation 4.8 are pertinent to dropout's practical use. Although the equation is derived with linearity assumptions, the results apply to nonlinear conditions.

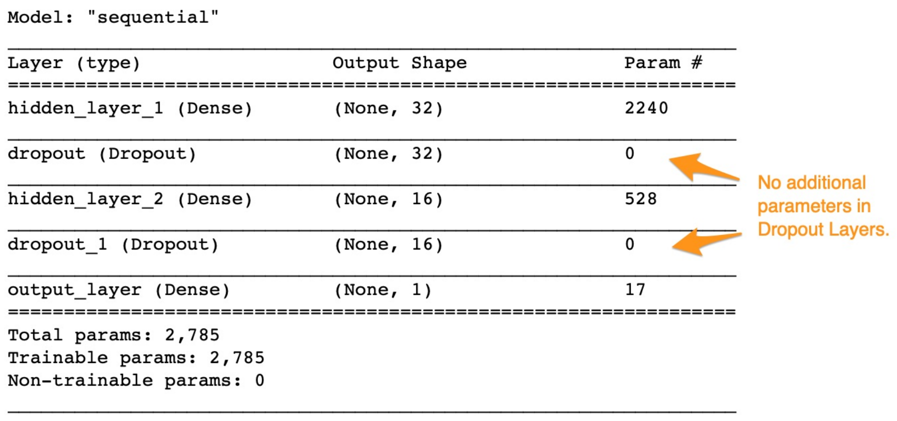
4.5.3 Dropout Layer

Dropout is implemented in `layers` class in TensorFlow. The primary argument in a `Dropout` layer is the `rate`. Rate is a float between 0 and 1 that defines the fraction of input units to drop (p in Equation 4.8).

As shown in Listing 4.14, dropout is added as a layer after a hidden layer. A dropout layer does not have any trainable parameter (also seen in the model summary in Figure 4.7).

Listing 4.14. MLP with dropout.

```
model = Sequential()
model.add(Input(shape=(N_FEATURES, )))
model.add(Dense(32, activation='relu',
  name='hidden_layer_1'))
model.add(Dropout(0.5))
model.add(Dense(16, activation='relu',
  name='hidden_layer_2'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid',
  name='output_layer'))

model.summary()
```



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
hidden_layer_1 (Dense)       (None, 32)                2240
_________________________________________________________________
dropout (Dropout)            (None, 32)                0
_________________________________________________________________
hidden_layer_2 (Dense)       (None, 16)                528
_________________________________________________________________
dropout_1 (Dropout)          (None, 16)                0
_________________________________________________________________
output_layer (Dense)         (None, 1)                 17
=================================================================
Total params: 2,785
Trainable params: 2,785
Non-trainable params: 0
_________________________________________________________________
```

Figure 4.7. *MLP with Dropout Model Summary.*

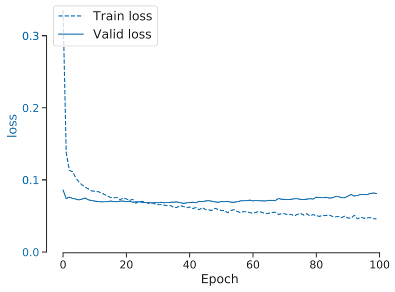
```python

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy',
                       tf.keras.metrics.Recall(),
                       performancemetrics.F1Score(),
                       performancemetrics.
                           FalsePositiveRate()]
             )
```

The results of the model are shown in Figure 4.8a-4.8c. Dropout did improve validation loss. Unlike the previous model, the validation loss is virtually non-increasing.

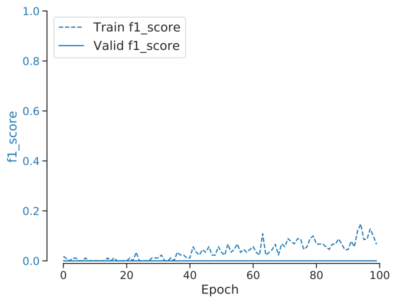
While dropout addressed the overfitting issue, it made the model non-predictive. This is shown in Figure 4.8b and 4.8c where the f1-score and recall are nearly zero.

Dropout sometimes causes this. This phenomenon is typical because a sub-model is learned at a time and the sub-model may not be sufficient to make an accurate prediction.

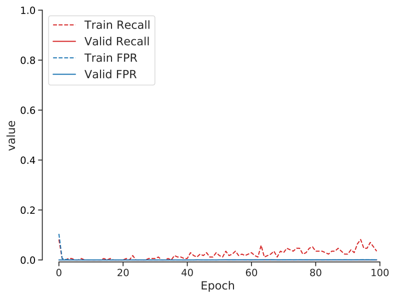
A common resolution to this is increasing the network size: increase the layers and/or their sizes. Moreover, a dropout layer has another argument `noise_shape` to add noise to the inputs. Adding noise can make the model more robust and, therefore, improve accuracy.



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure 4.8. *MLP with Dropout Model Results.*

4.6 Class Weights

A rare event problem has very few positively labeled samples. Due to this, even if the classifier is misclassifying the positive labels, their effect on the loss function is minuscule.

Remember the loss function in Equation 4.7, it gives equal importance (weights) to the positive and negative samples. To overcome this, we can overweight the positives and underweight the negative samples. A binary cross-entropy loss function will then be,

$$\mathcal{L}(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\Big(w_1 y_i \log(p_i) + w_0(1-y_i)\log(1-p_i)\Big) \tag{4.9}$$

where $w_1 > w_0$.

The class-weighting approach works as follows,

- The model estimation objective is to minimize the loss. In a perfect case, if the model could predict all the labels perfectly, i.e. $p_i = 1|y_i = 1$ **and** $p_i = 0|y_i = 0$, the loss will be **zero**. Therefore, the best model estimate is the one with the loss closest to zero.

- With the class weights, $w_1 > w_0$, if the model misclassifies the positive samples, i.e. $p_i \to 0|y_i = 1$, the loss goes **farther away from zero** as compared to if the negative samples are misclassified. In other words, the model training penalizes misclassification of positives more than negatives.

- Therefore, the model estimation strives to correctly classify the minority positive samples.

In principle, any arbitrary weights such that $w_1 > w_0$ can be used. But a rule-of-thumb is,

- w_1, positive class weight = number of negative samples / total samples.

- w_0, negative class weight = number of positive samples / total samples.

Using this thumb-rule, the class weights are defined in Listing 4.15. The computed weights are, `{0: 0.0132, 1: 0.9868}`.

Listing 4.15. Defining Class Weights.

```python
class_weight = {0: sum(y_train == 1)/len(y_train),
                1: sum(y_train == 0)/len(y_train)}
```

The model with class weights is shown in Listing 4.16. Except the class weights argument in `model.fit`, the remaining is the same as the baseline model in § 4.4.

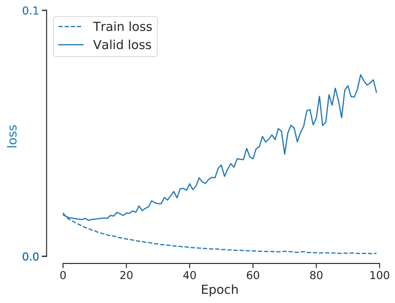
Listing 4.16. MLP Model with Class Weights.

```python
model = Sequential()
model.add(Input(shape=(N_FEATURES, )))
model.add(Dense(32, activation='relu',
  name='hidden_layer_1'))
model.add(Dense(16, activation='relu',
  name='hidden_layer_2'))
model.add(Dense(1, activation='sigmoid',
  name='output_layer'))

model.summary()

model.compile(optimizer='adam',
            loss='binary_crossentropy',
            metrics=['accuracy',
                     tf.keras.metrics.Recall(),
                     performancemetrics.F1Score(),
                     performancemetrics.
                        FalsePositiveRate()]
           )

history = model.fit(x=X_train_scaled,
                    y=y_train,
                    batch_size=128,
                    epochs=100,
                    validation_data=(X_valid_scaled,
                      y_valid),
                    class_weight=class_weight,
                    verbose=0).history
```

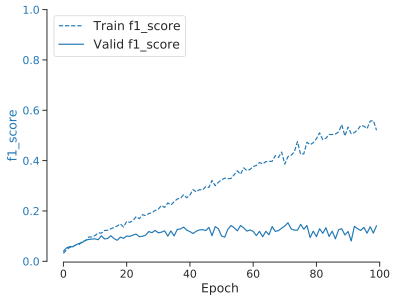
The results of the model with class weights are in Figure 4.9a-4.9c. While the training loss is well-behaved, the validation loss is going upwards. But here it is not necessarily due to overfitting.

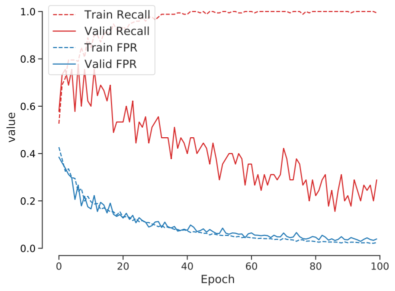
It is usual to see such behavior upon manipulating the class weights. Here the validation recall (true positives) is high at the beginning and then decreases along with the false positive rate (false positives). But because the weights of the positive class are higher when both recall and fpr decrease, the validation loss increases faster (effect of lessening true positives are higher) than the reduction (effect of lessening false positives).

Despite the awkward behavior of the loss, the recall improved significantly. But at the same time, the false positive rate rose to around 4% which is more than desired. This performance can be adjusted by changing the weights.



(a) *Loss.*





(b) *F1-score.* (c) *Recall and FPR.*

Figure 4.9. *MLP with Class Weights Model Results.*

4.7 Activation

Activation functions are one of the primary drivers of neural networks. An activation introduces the non-linear properties to a network.

It is shown in Appendix A that a network with *linear* activation is equivalent to a simple regression model. It is the non-linearity of the activations that make a neural network capable of learning non-linear patterns in complex problems.

But there are a variety of activations, e.g., `tanh`, `elu`, `relu`, etc. Does choosing one over the other improve a model?

Yes. If appropriately chosen, an activation can significantly improve a model. Then, how to choose an appropriate activation?

An appropriate activation is the one that does not have **vanishing** and/or **exploding** gradient issues. These issues are elaborated next.

4.7.1 What is Vanishing and Exploding Gradients?

Deep learning networks are learned with backpropagation. Backpropagation methods are gradient-based. A gradient-based parameter learning can be generalized as

$$\theta_{n+1} \leftarrow \theta_n - \eta\nabla_\theta \tag{4.10}$$

where n is a learning iteration, η is a learning rate, and ∇_θ is the gradient of the loss $\mathcal{L}(\theta)$ with respect to the model parameters θ.

The equation shows that gradient-based learning iteratively estimates θ. In each iteration, the parameter θ is moved "closer" to its optimal value θ^*.

However, whether the gradient will truly bring θ closer to θ^* will depend on the gradient itself. This is visually illustrated in Figure 4.10a-4.10c.

The gradient, ∇, guides the parameter θ to its optimal value. An apt gradient is, therefore, critical for the parameter's journey to the optimal.

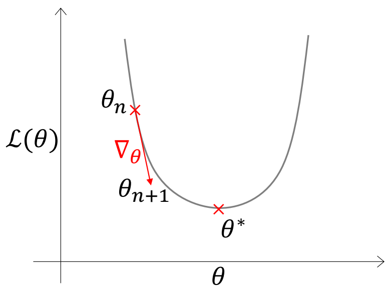
In these figures, the horizontal axis is the model parameter θ, the vertical axis is the loss $\mathcal{L}(\theta)$ and θ^* indicates the optimal parameter at the lowest point of loss.

A stable gradient is shown in Figure 4.10a. The gradient has a magnitude that brings θ_n closer to θ^*.

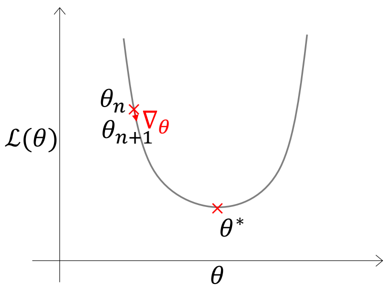
But if the gradient is too small the θ update is negligible. The updated parameter θ_{n+1}, therefore, stays far from θ^*.

The gradient when too small is called the vanished gradient. And this phenomenon is referred to as a **vanishing gradient** issue.

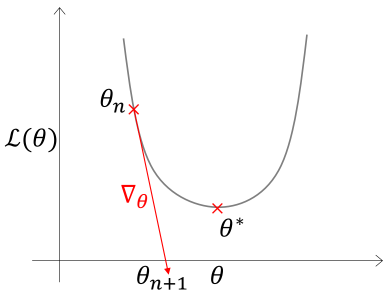
On the other extreme, sometimes the gradient is massive. Depicted in Figure 4.10c, a large gradient moves θ farther away from θ^*.



(a) *Stable gradient.*



(b) *Vanished gradient.*



(c) *Exploded gradient.*

Figure 4.10. *Stable gradient vs. vanishing and exploding gradient.*

This is the **exploding gradient** phenomenon. A large gradient is referred to as an exploded gradient and it makes reaching θ^* rather elusive.

4.7.2 Cause Behind Vanishing and Exploding Gradients

This is explained with the help of expressions in Figure 4.11. The figure is showing the gradient expressions for the layers of the network constructed in this chapter. The expressions are derived in Appendix D.

From the expressions in the figure, the gradients are,

- **Chain multiplied**. The gradients are computed using the chain rule[12]. That is, the partial gradients are successively multiplied.

- **Of the activations**. The gradient ∇_θ is the chain multiplication of the partial gradients of the (layer) activations. In the figure, the $\sigma(\cdot)$ and $g(\cdot)$ are the activation at the output and hidden layers, respectively.
- **Dependent on the feature map**. Each part of the chain multiplication has the layer's input features. The first layer has the input features $\boldsymbol{x}$ and the subsequent layers have the feature maps $\boldsymbol{z}$.

Due to these characteristics, the gradients can,

- **Vanish**. A layer's gradient is a chain of partial gradients multiplied together. At each network depth, another gradient term is included in the chain.

  As a result, the deeper a network, the longer the chain. These gradients are of the activations. If the *activation gradients* are smaller than 1, the overall gradient rapidly gets close to zero. This causes the gradient vanishment issue.
- **Explode**. If the *activation gradients'* or the feature maps' magnitudes are larger than 1, the gradient quickly inflates due to the chain multiplication. Excessive inflation leads to what is termed as gradient explosion.

The vanishing and exploding gradients issues are a consequence of the activation gradient and feature map. The feature map effect is addressed using batch normalization in Ioffe and Szegedy 2015. More work has been done on activations to address these issues and is discussed next.

4.7.3 Gradients and Story of Activations

The vanishing and exploding gradient issues were becoming a bottleneck in developing complex and large neural networks. They were

[12]Chain rule: $f'(x) = \frac{\partial f}{\partial u}\frac{\partial u}{\partial v}\cdots\frac{\partial z}{\partial x}$.

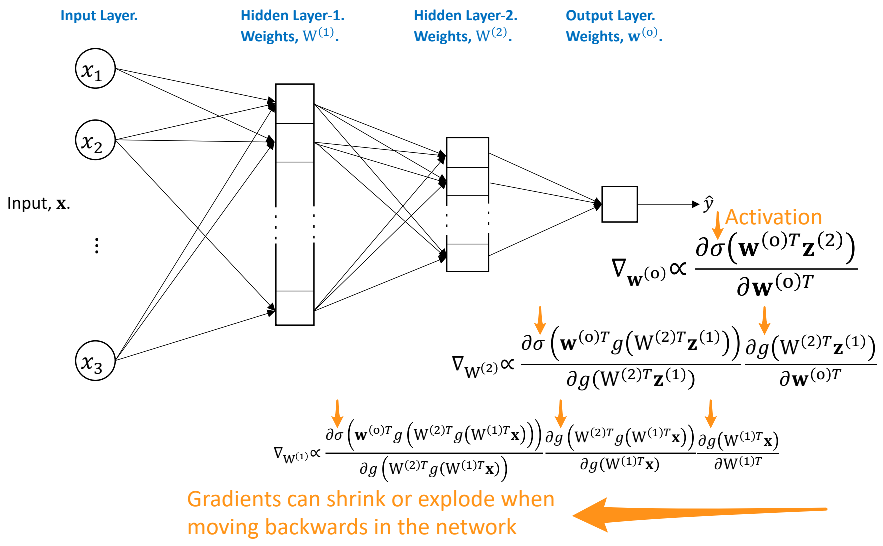

Input Layer.
Hidden Layer-1. Weights, $W^{(1)}$.
Hidden Layer-2. Weights, $W^{(2)}$.
Output Layer. Weights, $\mathbf{w}^{(o)}$.
x_1
x_2
x_3
Input, $\mathbf{x}$.
$\hat{y}$
Activation
$\nabla_{\mathbf{w}^{(o)}} \propto \frac{\partial \sigma\left(\mathbf{w}^{(o)T}\mathbf{z}^{(2)}\right)}{\partial \mathbf{w}^{(o)T}}$
$\nabla_{W^{(2)}} \propto \frac{\partial \sigma\left(\mathbf{w}^{(o)T} g\left(W^{(2)T}\mathbf{z}^{(1)}\right)\right)}{\partial g\left(W^{(2)T}\mathbf{z}^{(1)}\right)} \frac{\partial g\left(W^{(2)T}\mathbf{z}^{(1)}\right)}{\partial \mathbf{w}^{(o)T}}$
$\nabla_{W^{(1)}} \propto \frac{\partial \sigma\left(\mathbf{w}^{(o)T} g\left(W^{(2)T} g\left(W^{(1)T}\mathbf{x}\right)\right)\right)}{\partial g\left(W^{(2)T} g\left(W^{(1)T}\mathbf{x}\right)\right)} \frac{\partial g\left(W^{(2)T} g\left(W^{(1)T}\mathbf{x}\right)\right)}{\partial g\left(W^{(1)T}\mathbf{x}\right)} \frac{\partial g\left(W^{(1)T}\mathbf{x}\right)}{\partial W^{(1)T}}$
Gradients can shrink or explode when moving backwards in the network

Figure 4.11. *Illustration to explain vanishing and exploding gradients. Model training uses the backpropagation technique, which indeed relies on gradients. Gradients are a product of partial derivatives. Each layer adds another partial derivative. Therefore, as backpropagation runs down the lower layers the partial derivatives get stacked up. Depending on their small or large magnitudes, the overall gradient can vanish or explode.*

first resolved to some extent with the rectified linear unit (`relu`) and `leaky-relu` in Maas, Hannun, and Ng 2013.

`Relu` activation is defined as,

$$g(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.11}$$

The gradient of `relu` is 1 if $x > 0$ and 0 otherwise. Therefore, when a `relu` unit is "activated," i.e., the input in it is anything greater than zero, its derivative is 1. Due to this, the **gradients vanishment do not happen** for the positive-valued units in an arbitrarily deep network. These units are sometimes also called *active* units.

Additionally, `relu` is nonlinear at $x = 0$ with a saturation region for $x < 0$. A saturation region is where the gradient is zero. The saturation region dampens the activation variance if it is too large in the lower layers. This helps in learning lower level features, e.g., more abstract distinguishing patterns in a classifier.

However, only one saturation region, i.e., zero gradient in one region of x, is desirable. More than one saturation region, like in `tanh` activation (has two saturation region shown in Figure 4.12b), make the variance too small causing gradients vanishment.

A saturation region is the part of a function where the gradient becomes zero. Only one saturation region is desirable.

Theoretically, `relu` resolved the vanishing gradient issue. Still, researchers were skeptical about `relu`. Because without any negative outputs the `relu` activations' means (averages) are difficult to control. Their means can easily stray to large values.

Additionally, `relu`'s gradient is zero whenever the unit is not active. This was believed to be restrictive because the gradient-based backpropagation does not adjust the weights of units that never activate initially. Eventually causing cases where a unit never activates.

To alleviate this, the `relu` authors further developed `leaky-relu`. Unlike `relu`, it has a scaled-down output, $0.01x$ when $x < 0$. The activations are visualized for comparison in Figure 4.12a.

As seen in the figure, the `leaky-relu` has a small but non-zero output for $x < 0$. But this yields a non-zero gradient for every input. That is, it has no saturation region. This did not work in favor of `leaky-relu`.

To resolve the issues in `relu` and `leaky-relu`, `elu` was developed in Clevert, Unterthiner, and Hochreiter 2015. The `elu` activation is defined as,

$$g(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(\exp x - 1), & \text{otherwise.} \end{cases} \tag{4.12}$$

`Elu`'s gradient is visualized in Figure 4.12b. In contrast to the `leaky-relu`, `elu` has a saturation in the negative region. As mentioned before, the saturation results in small derivatives which decrease the variance and, therefore, the information is well-propagated to the next layer.

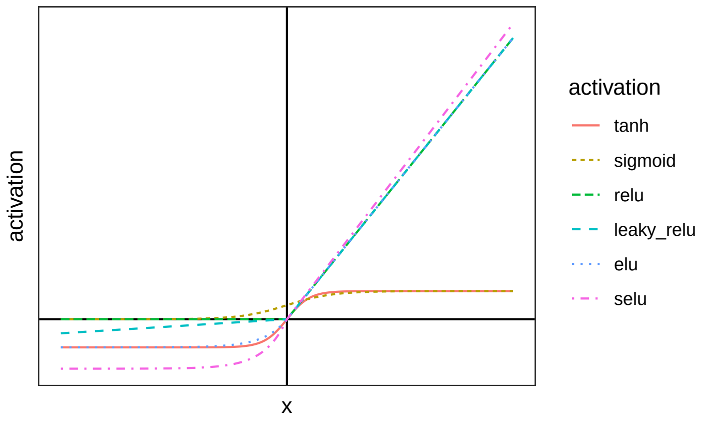
With this property and the non-zero negative outputs, `elu` could enable faster learning as they bring the gradient closer to the natural gradient (shown in Clevert, Unterthiner, and Hochreiter 2015).

However, despite the claims by `elu` or `leaky-relu`, they did not become as popular as `relu`. `Relu`'s robustness made it a default activation for most models.

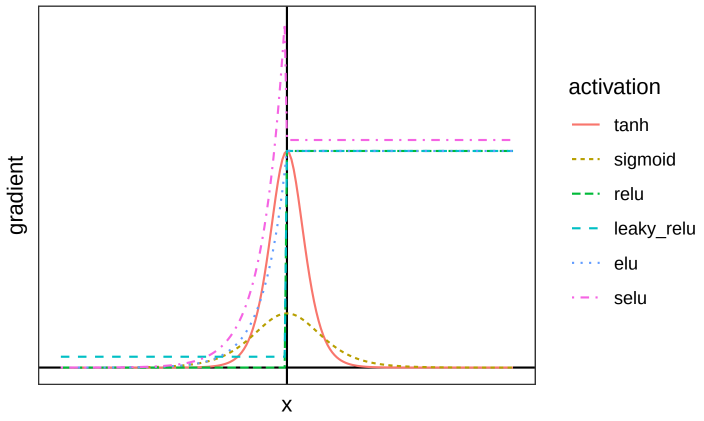
In any case, `relu` and other activations developed thus far could not address the gradient explosion issue. Batch normalization, a computation outside of activation done in a `BatchNormalization` layer, was typically used to address it. Until Scaled Exponential Linear Unit (`selu`) was developed in Klambauer et al. 2017.

`Selu` can construct a self-normalizing neural network. This addresses both the vanishing and exploding gradient issues at the same time.

A `selu` activation, shown in Equation 4.13 below, appears to be a



(a) *Activation output.*



(b) *Activation gradient.*

Figure 4.12. *Activations comparison. The top chart compares the shape of activations $g(x)$ and the bottom compares their gradients $\frac{\partial g(x)}{\partial x}$. An ideal activation for most hidden layers has, 1. nonlinearity which makes a network nonlinear to solve complex problems, 2. a region where the gradient is ≥ 1 and $< 1 + \delta$, where δ is small, to avoid gradient vanishment and explosion, respectively, and 3. a saturation region where the gradient becomes 0 to reduce variance.*

minor change in `elu` in Equation 4.12 with a λ factor.

$$g(x) = \begin{cases} \lambda x, & \text{if } x > 0 \\ \lambda\alpha(\exp x - 1), & \text{otherwise} \end{cases} \tag{4.13}$$

where, $\lambda > 1$.

But Klambauer et al. 2017 proved that the simple change brought an important property of *self-normalization* that none of the predecessors had.

4.7.4 Self-normalization

The activations of a neural network are considered self-normalized if their means and variances across the samples are within predefined intervals. With `selu`, if the input mean and variance are within some intervals, then the outputs' mean and variance will also be in the same respective intervals. That is, the normalization is transitive across layers.

This means that `selu` successively normalizes the layer outputs when propagating through network layers. Therefore, if the input features are normalized the output of each layer in the network will be automatically normalized. And, this normalization **resolves the gradient explosion issue**.

To achieve this, `selu` initializes normalized weight vectors such that $\sum w_i = 0$ and $\sum w_i^2 = 1$ for every layer. The weights are randomly drawn from a truncated normal distribution. For this initialization the best values for the parameters λ and α in Equation 4.13 are derived as 1.0507 and 1.6733 in Klambauer et al. 2017.

The development of `selu` in Klambauer et al. 2017 outlines the generally desired properties of an activation. Among them, the presence of negative and positive values, and a (one) saturation region are the first candidates.

Figure 4.12a and 4.12b display the presence/absence of these properties among the popular activations. Only `elu` and `selu` have both the properties. However, `selu` went beyond `elu` with two additional attributes,

- **Larger gradient**. A gradient larger than one. This increases the variance if it is too small in the lower layers. This would make learning low-level features in deeper networks possible.

  Moreover, the gradient is larger around $x = 0$ compared to `elu` (see Figure 4.12b). This reduces the noise from weaker nodes and guides them to their optimal values faster.

- **Balanced variance**. A fixed point where the variance damping (due to the gradient saturation) is equalized by variance inflation (due to greater than 1 gradient). This controls the activations from vanishing or exploding.

`Selu` properties are irresistible. It performs better in some data sets and especially in very deep networks. But in shallow networks such as the baseline network in this chapter, `selu` might not outperform others.

4.7.5 `Selu` Activation

The MLP model with `selu` activation is shown in Listing 4.17. As shown in the listing, a `selu` activated model requires,

- `kernel_initializer='lecun_normal'`[13]. This initializes the weights by sampling the weight vectors from a truncated normal distribution with mean 0 and standard deviation as

  $$\frac{1}{\sqrt{m_{l-1}}}$$

  where m_{l-1} is the number of input units (size of the previous layer).

- `AlphaDropout()`[14]. Standard dropout randomly sets inputs to zero. This does not work with `selu` as it disturbs the activations'

[13]`https://www.tensorflow.org/api_docs/python/tf/initializers/lecun_normal`

[14]`https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/AlphaDropout`

mean and variance. Therefore, a new "alpha dropout" was introduced in Klambauer et al. 2017 in which the inputs are randomly set to α'. α' is set so that the activations' original mean and variance are preserved. This ensures the self-normalizing property of `selu`.

Empirically, an alpha dropout rate of 0.05 or 0.1 leads to a good performance.

Listing 4.17. MLP with `selu` Activation.

```
model = Sequential()
model.add(Input(shape=(N_FEATURES, )))
model.add(Dense(32, activation='selu',
  kernel_initializer='lecun_normal'))
model.add(AlphaDropout(0.1))
model.add(Dense(16, activation='selu',
  kernel_initializer='lecun_normal'))
model.add(AlphaDropout(0.1))
model.add(Dense(1, activation='sigmoid'))

model.summary()

model.compile(optimizer='adam',
            loss='binary_crossentropy',
            metrics=['accuracy',
                     tf.keras.metrics.Recall(),
                     performancemetrics.F1Score(),
                     performancemetrics.
                        FalsePositiveRate()]
           )

history = model.fit(x=X_train_scaled,
                    y=y_train,
                    batch_size=128,
                    epochs=100,
                    validation_data=(X_valid_scaled,
                      y_valid),
                    verbose=0).history
```

4.8 Novel Ideas Implementation

4.8.1 Activation Customization

Activations is an active research field in Deep Learning and still at its nascent stage. It is common among researchers to attempt novel activation ideas. To enable this, custom activation implementation is shown here.

Activations can be defined as a conventional python function. For such definitions, their gradient should also be defined and registered to TensorFlow[15].

However, the gradient definition is usually not required if the activation is defined using TensorFlow functions. TensorFlow has derivatives predefined for its in-built functions. Therefore, explicit gradient declaration is not required. This approach is, therefore, simpler and is practically applicable in most activation definitions.

Here a custom activation, *Thresholded Exponential Linear Unit* (`telu`), is defined in Equation 4.14.

$$g(x) = \begin{cases} \lambda x, & \text{if } x > \tau \\ 0, & \text{if } -\tau \le x \le \tau \\ \lambda\alpha(\exp x - 1), & \text{if } x < -\tau \end{cases} \tag{4.14}$$

With this activation, weak nodes smaller than τ will be deactivated. The idea behind thresholding small activations is applying regularization directly through the `telu` activation function.

This custom activation is a modification of `selu`. The activation is defined in Listing 4.18. The first input argument `x` to activation is a tensor. Any subsequent argument is the hyperparameters, that can be defined and set as required. Here, the `threshold`, τ, is the hyperparameter.

Listing 4.18. Custom Activation `telu` definition.

[15] See `tf.RegisterGradient` at `https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/RegisterGradient`

```python
1  from tensorflow.keras import backend as K
2  def telu(x, threshold=0.1):
3      '''
4      Thresholded Exponential linear unit.
5  
6        Arguments:
7        x: Input tensor.
8        alpha: A scalar = 1.6732632, slope
9               of negative section.
10       scale: A scalar = 1.05070098, to
11              keep the gradient > 1 for
12              x > 0.
13 
14       Returns:
15       The thresholded exponential linear
16       activation:
17       scale * x, if x > threshold,
18       0, if -threshold < x < threshold
19       scale * alpha * (exp(x)-1), if
20         `x < -threshold`.
21     '''
22     x_ = tf.keras.activations.selu(x)
23 
24     # Create a tensor of same shape as x
25     # with the threshold in each cell.
26     threshold_ = tf.math.scalar_mul(threshold,
27       K.ones_like(x_))
28 
29     # Creates an identity tensor which
30     # is one if the abs(x) > threshold.
31     threshold_multiplier =
32       K.cast(tf.keras.backend.less(threshold_,
33         K.abs(x_)),
34         dtype='float32')
35 
36     return tf.math.multiply(x_,
37       threshold_multiplier)
```

The activation definition in Listing 4.18 is using in-built TensorFlow functions for every operation. The input `x` is, first, passed through `selu(x)` in line 22 and then thresholded. The thresholding is done by,

- defining a tensor of the same shape as `x` which has a value `1`, if the corresponding absolute of `x` element is greater than the `threshold` and `0`, otherwise.
- Followed by an element-wise multiplication between the vectors using `tf.math.multiply` in line 36.

The custom activation must be tested by defining TensorFlow tensors and passing them through the function. The testing is shown in Listing 4.19.

Listing 4.19. Testing the custom activation `telu`.

```
# Testing TELU
test_x = tf.convert_to_tensor([-1., 1.1, 0.01],
  dtype=tf.float32)

# Sanity test 1: telu output should be
# equal to selu if threshold=0.
tf.print('TELU with threshold=0.0:',
  telu(test_x, threshold=0.))
tf.print('SELU for comparison: ',
  tf.keras.activations.selu(test_x))

# Output:
# TELU with threshold=0.0: [-1.11133075 1.15577114 0.0105070099]
# SELU for comparison:     [-1.11133075 1.15577114 0.0105070099]

# Sanity test 2: telu should make
# activations < threshold.
tf.print('TELU default setting: ',
 telu(test_x))  # default threshold = 0.1

# Output:
# TELU default setting:    [-1.11133075 1.15577114 0]
```

First, a sanity test is done by setting the threshold as 0.0. In this setting, the output should be equal to the output of a `selu` activation. Second, `telu` functionality is validated by setting the threshold as 0.1.

With this threshold, the input smaller than the threshold is made 0. The outputs are shown under comments in the listing.

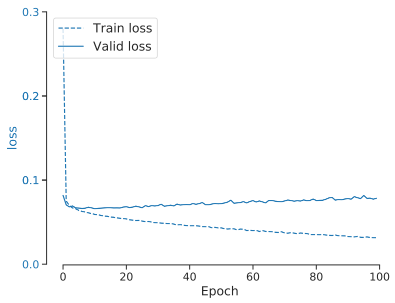
The activation definition passed both tests. In general, it is important to run various tests on custom functions before moving forward. Because the farther we go with custom functions, the harder it becomes to pinpoint the problem if any arises.

In Listing 4.20, an MLP model is trained with the custom `telu` activation.

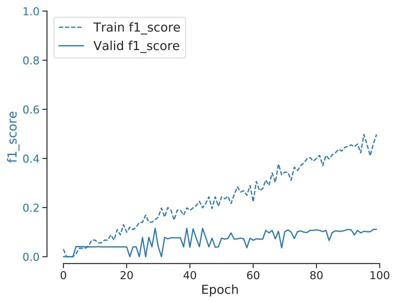
Listing 4.20. MLP model with the custom `telu` activation.

```
model = Sequential()
model.add(Input(shape=(N_FEATURES, )))
model.add(Dense(32, activation=telu))
model.add(Dense(16, activation=telu))
model.add(Dense(1, activation='sigmoid'))

model.summary()

model.compile(optimizer='adam',
            loss='binary_crossentropy',
            metrics=['accuracy',
                     tf.keras.metrics.Recall(),
                     performancemetrics.F1Score(),
                     performancemetrics.
                        FalsePositiveRate()]
           )

history = model.fit(x=X_train_scaled,
                    y=y_train,
                    batch_size=128,
                    epochs=100,
                    validation_data=(X_valid_scaled,
                      y_valid),
                    verbose=0).history
```

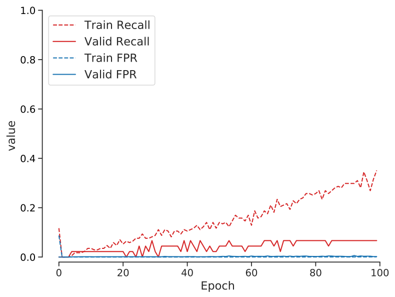
`Telu` activation performed at par with the baseline. While the baseline model had increasing validation, `telu` resolved the increasing validation loss issue without sacrificing the accuracy.



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure 4.13. *MLP with `telu` activation results.*

The development and results from a custom `telu` activation here demonstrates that there is significant room for research in activation.

4.8.2 Metrics Customization

Looking at suitably chosen metrics for a problem tremendously increases the ability to develop better models. Although a metric does not directly improve model training but it helps in a better model selection.

Several metrics are available outside TensorFlow such as in `sklearn`. However, they cannot be used directly during model training in TensorFlow. This is because the metrics are computed while processing batches during each training epoch.

Fortunately, TensorFlow provides the ability for this customization. The custom-defined metrics `F1Score` and `FalsePositiveRate` are provided in the user-defined `performancemetrics` library. Learning the programmatic context for the customization is important and, therefore, is elucidated here.

The TensorFlow official guide shows the steps for writing a custom metric[16]. It instructs to create a new subclass inheriting the `Metric` class and work on the following definitions for the customization,

- `__init__()`: All the state variables should be created in this method by calling `self.add_weight()` like:
  `self.var = self.add_weight(...)`.
- `update_state()`: All updates to the state variables should be done as `self.var.assign_add(...)`.
- `result()`: The final result from the state variables is computed and returned in this function.

Using these instructions, the `FalsePositiveRate()` custom metric is defined in Listing 4.21. Note that `FalsePositives` metric is already present in TensorFlow. But drawing the false positive rate (a ratio) from it during training is not direct and, therefore, the `FalsePositiveRate()` is defined.

Listing 4.21. Custom FalsePositiveRate Metric Definition.

```python
class FalsePositiveRate(tf.keras.metrics.Metric):
    def __init__(self, name='false_positive_rate',
      **kwargs):
        super(FalsePositiveRate, self).
        __init__(name=name, **kwargs)
        self.negatives = self.add_weight(name='
            negatives',
          initializer='zeros')
        self.false_positives = self.add_weight(
          name='false_negatives',
          initializer='zeros')
```

[16]tensorflow/python/keras/metrics.py at `shorturl.at/iqDS7`

```

    def update_state(self,
      y_true, y_pred, sample_weight=None):
        '''
        Arguments:
        y_true  The actual y. Passed by
                default to Metric classes.
        y_pred  The predicted y. Passed
                by default to Metric classes.

        '''
        # Compute the number of negatives.
        y_true = tf.cast(y_true, tf.bool)

        negatives = tf.reduce_sum(tf.cast(
          tf.equal(y_true, False), self.dtype))

        self.negatives.assign_add(negatives)

        # Compute the number of false positives.
        y_pred = tf.greater_equal(
            y_pred, 0.5
        )  # Using default threshold of 0.5 to
           # call a prediction as positive labeled.

        false_positive_vector =
          tf.logical_and(tf.equal(y_true, False),
            tf.equal(y_pred, True))
        false_positive_vector = tf.cast(
            false_positive_vector,
          self.dtype)
        if sample_weight is not None:
            sample_weight = tf.cast(sample_weight,
              self.dtype)
            sample_weight = tf.broadcast_weights(
              sample_weight, values)
            values = tf.multiply(
                false_positive_vector,
              sample_weight)

        false_positives = tf.reduce_sum(
            false_positive_vector)

            self.false_positives.assign_add(
                false_positives)

    def result(self):
        return tf.divide(self.false_positives,
          self.negatives)
```

How do metrics computation and customization work?

A model training is done iteratively. The iterations happen at multiple levels. The iteration levels for multi-layer perceptron training are laid out in Figure 4.14.

The topmost iteration level is *epochs*. Within an epoch, a model is trained iteratively over randomly selected batches. The batches are the second level of iteration.

Multiple samples are present within a batch. The model can be trained by processing one sample at a time. But they are processed together as a batch (as shown in Equation 4.4) for computational efficiency. The `sample` is, therefore, grayed in the figure indicating it a logical iteration instead of an actual one.

During batch processing, all the model parameters—weights and biases—are updated. Simultaneously, the *states* of a metric are also updated. Upon processing all the batches in an epoch, both the estimated parameters and computed metrics are returned. Note that all these operations are enclosed within an epoch and no values are communicated between two epochs.

The meaning of the metrics state, metrics computation, and the programmatic logic in Listing 4.21 for defining `FalsePositiveRate` are enumerated below.

- The class `FalsePositiveRate()` is inheriting the `Metric` class in TensorFlow. As a result, it automatically has the `__init__()`, `update_state()`, and `result()` definitions. These definitions will be overwritten during the customization.

- `__init__()` is the entry gate to a metric class. It initializes the

epoch
batch
sample

Figure 4.14. *Levels of iteration in MLP training. The iterations are interpreted up to the sample level. However, computationally all samples in a batch are processed together using tensor operations. Hence, the `sample` is grayed here to indicate it is only a logical iteration level.*

state variables. **State variables** are used for metric computation.

- The false-positive rate is the ratio of false positives over the negatives. Therefore, `false_positives` and `negatives` become the state variables.
- The state variables are prefixed with `self.`. A `self` variable can be accessed and updated from any definition in the class.
- The states—false positives and negatives—are initialized to zero in `__init__()`. That is, at the onset of an epoch all the states are reset to their initial values. In some problems, the initial value can be other than zero.
- After `__init__()`, the program enters `update_state()`. This function can be imagined as the metric processing unit. It is run for every batch in an epoch. Inside this function, the metric states are updated from the outcome derived from a batch.
- `update_state()` has the actual labels `y_true` and the predictions `y_pred` for the samples in the batch as default arguments.
- Line 23-28 in the function are computing and updating the `self.negatives` state variable. The original labels `y_true` in the inputted data are numeric 0 and 1. They are first converted into boolean in Line 23. The `negatives` in the batch are then computed by finding all the `False` (the negative samples) in the boolean `y_true` vector followed by converting it back to numeric 0/1 and summing the now numeric vector.

- The `negatives` computed in Line 25 is the number of negative samples in the batch under process. `negatives` is a local variable in its enclosing function `update_state()`. It is added to the metric state variable `self.negatives` (a class variable) in Line 28.
- Similarly, the false positives in the batch is computed in Line 31-51. False positives is derived by comparing the predictions `y_pred` with the actuals `y_true`.
- To get to the false positives, the first step is to convert `y_pred` to boolean. `y_pred` is the output of a *sigmoid* activated output layer. Thus, it is a continuous number in $(0, 1)$.
- `y_pred` is converted to boolean using a default threshold of 0.5[17]. The predictions in `y_pred` greater than the threshold are made `True` and `False`, otherwise.
- A false positive is when the actual label `y_true` is `False` but the prediction `y_pred` is `True`. That is, the model incorrectly predicted a negative sample as positive. This logical comparison is done in Line 36.
- The output of the logical comparison is a boolean vector `false_positive_vector` which is converted to numeric 0/1 in Line 39.
- The `false_positives` in the batch is then computed by summing the vector in Line 49.
- The possibility of a not `None` sample weight is accounted for in Line 26-29.
- Same as before, the `self.false_positives` state is then updated in Line 51.
- After the epoch has iterated through all the batches the metric state variables `self.negatives` and `self.false_positives` will have stored the totals for the entire data set.

[17]This is a default value used in practice. If needed, it can be changed to any other desired value.

- The state `self.negatives` has the total negative samples in the data set. This is a constant and, therefore, `self.negatives` will remain the same in all epochs. `self.false_positives`, on the other hand, is a result of the goodness of model fitting. And so it changes (ideally reduces) over successive epochs.

- Finally, the program goes to `result()` and yields the metric's value. This is the "exit" gate of the metric class. Here the false positive rate is computed by dividing the metric states `self.false_positives` and `self.negatives`.

- **Important**: a metric state should be additive only. This is because the `update_state()` can only increment or decrement the state variable. For instance, if the false-positive rate—a ratio and, thus, non-additive—was created as the state variable and updated in `update_state()` it would yield $\sum_{i\in\texttt{batches}} \frac{\texttt{false positives}_i}{\texttt{negatives}_i}$ instead of the desired $\frac{\sum_{i\in\texttt{batches}} \texttt{false positives}_i}{\sum_{i\in\texttt{batches}} \texttt{negatives}_i}$. In general, any non-additive computation should, therefore, be done in `result()`.

The state in metric customization should be additive only.

Similarly, the `F1Score` custom metric is defined in Listing 4.22.

Listing 4.22. Custom F1Score Metric Definition.

```
class F1Score(tf.keras.metrics.Metric):
    def __init__(self, name='f1_score', **kwargs):
        super(F1Score, self).__init__(name=name, **
           kwargs)
        self.actual_positives =
          self.add_weight(name='actual_positives',
                          initializer='zeros')
        self.predicted_positives =
          self.add_weight(name='predicted_positives',
                          initializer='zeros')
        self.true_positives =
          self.add_weight(name='true_positives',
                          initializer='zeros')

    def update_state(self,
      y_true, y_pred, sample_weight=None):
        '''
        Arguments:
        y_true   The actual y. Passed by default
                 to Metric classes.
        y_pred   The predicted y. Passed by
                 default to Metric classes.

        '''
        # Compute the number of negatives.
        y_true = tf.cast(y_true, tf.bool)

        actual_positives = tf.reduce_sum(
            tf.cast(tf.equal(y_true, True), self.dtype))
        self.actual_positives.assign_add(actual_positives)

        # Compute the number of false positives.
        y_pred = tf.greater_equal(
            y_pred, 0.5
        )  # Using default threshold of 0.5 to call a prediction as positive labeled.

        predicted_positives = tf.reduce_sum(
            tf.cast(tf.equal(y_pred, True),
                    self.dtype))
        self.predicted_positives.assign_add(predicted_positives)

        true_positive_values =
          tf.logical_and(tf.equal(y_true, True),
                         tf.equal(y_pred, True))
        true_positive_values =
          tf.cast(true_positive_values, self.dtype)

        if sample_weight is not None:
            sample_weight =
              tf.cast(sample_weight, self.dtype)
            sample_weight =
              tf.broadcast_weights(sample_weight,
                 values)
            values =
              tf.multiply(true_positive_values,
                 sample_weight)

        true_positives = tf.reduce_sum(
           true_positive_values)

        self.true_positives.assign_add(
           true_positives)

    def result(self):
        recall =
          tf.math.divide_no_nan(
            self.true_positives,
            self.actual_positives)
        precision =
          tf.math.divide_no_nan(
            self.true_positives,
            self.predicted_positives)
        f1_score =
          2 * tf.math.divide_no_nan(
            tf.multiply(recall, precision),
            tf.math.add(recall, precision))

        return f1_score
```

4.9 Models Evaluation

Several models were built in this chapter. Out of them, one model should be selected as the *final* model. The performance of the final model is evaluated on the test data. This is a traditional process of model building and selection. For this purpose, the data set was initially

Table 4.1. MLP models comparison. The red highlighted values indicate an undesirable or poor result.

	Validation			
Model	**Loss**	**F1-score**	**Recall**	**FPR**
Baseline	Increasing	0.13	0.08	0.001
Dropout	Non-increasing	0.00	0.00	0.000
Class weights	Increasing	0.12	0.31	0.102
`selu`	Non-increasing	0.04	0.02	0.001
`telu` (custom)	Non-increasing	0.12	0.08	0.001

split in train, valid, and test.

The selection is made by comparing the validation results. These results are summarized in Table 4.1. The baseline model has higher accuracy measures but increasing validation loss indicating potential overfitting. Dropout resolves the overfitting but the accuracy goes to zero. Class weights boosted the accuracy but also has a high false-positive rate. The `selu` activation attempted after that had significantly lower f1-score than the baseline.

The custom activation `telu` performed relatively better than the others. It has non-increasing validation loss and close to the baseline accuracy. Therefore, `telu` activated model is selected as the final model.

This does not mean `telu` will be the best choice in other cases. The process of building multiple models and selection should be followed for every problem.

The final selected model is evaluated using `model.evaluate()` function. The `evaluate` function applies the trained model on test data and returns the performance measures defined in `model.compile()`.

The evaluation and test results are shown in Figure 4.23.

Listing 4.23. MLP final selected model evaluation.

```
# Final model
model.evaluate(
            x=X_test_scaled,
            y=y_test,
            batch_size=128,
             verbose=1)

# loss: 0.0796 - accuracy: 0.9860 -
# recall_5: 0.0755 - f1_score: 0.1231 -
# false_positive_rate: 0.0020
```

Multi-layer perceptrons are elementary deep learning models. Any reasonable result from MLPs act as a preliminary screening that there are some predictive patterns in the data. This lays down a path for further development with different network architectures.

4.10 Rules-of-thumb

This chapter went through a few MLP model constructions. Even in those few models, several modeling constructs and their settings were involved. In practice, there are many more choices. And they could be overwhelming.

Therefore, this chapter concludes with some thumb-rules to make an initial model construction easier.

- **Number of layers**. Start with two hidden layers (this does not include the last layer).
- **Number of nodes (size) of intermediate layers**. A number from a geometric series of 2: 1, 2, 4, 8, 16, 32, ... The first layer should be around half of the number of input data features. The next layer size is half of the previous and so on.
- **Number of nodes (size) of the output layer**.
  - **Classification**. If binary classification then the size is one. For a multi-class classifier, the size is the number of classes.
  - **Regression**. If a single response then the size is one. For multi-response regression, the size is the number of responses.
- **Activation**.
  - **Intermediate layers**. Use `relu` activation.

  - **Output layer**. Use `sigmoid` for binary classification, `softmax` for a multi-class classifier, and `linear` for regression.

- **Dropout**. Do not add dropout in a baseline MLP model. It should be added to a large or complex network. Dropout should certainly be not added to the input layer.

- **Data Preprocessing**. Make the features X as numeric by converting any categorical columns into one-hot-encoding. Then, perform feature scaling. Use `StandardScaler` if the features are unbounded and `MinMaxScaler` if bounded.

- **Split data to train, valid, test**. Use `train_test_split` from `sklearn.model_selection`.

- **Class weights**. Should be used with caution and better avoided for extremely imbalanced data. If used in a binary classifier, the weights should be: 0: number of 1s / data size, 1: number of 0s / data size.

- **Optimizer**. Use `adam` with its default learning rate.

- **Loss**.
  - **Classification**. For binary classification use `binary_crossentropy`. For multiclass, use `categorical_crossentropy` if the labels are one-hot-encoded, otherwise use `sparse_categorical_crossentropy` if the labels are integers.
  - **Regression**. Use `mse`.

- **Metrics**.
  - **Classification**. Use `accuracy` that shows the percent of correct classifications. For imbalanced data, also include `Recall`, `FalsePositiveRate`, and `F1Score`.
  - **Regression**. Use `RootMeanSquaredError()`.

- **Epochs**. Set it as 100 and see if the model training shows decreasing loss and any improvement in the metrics over the epochs.

- **Batch size**. Choose the batch size from the geometric progression of 2. For imbalanced data sets have larger value, like 128, otherwise, start with 16.

For advanced readers,

- **Oscillating loss**. If the oscillating loss is encountered upon training then there is a convergence issue. Try reducing the learning rate and/or change the batch size.

- **Oversampling and undersampling**. Random sampling will work better for multivariate time series than synthesis methods like `SMOTE`.

- **Curve shifting**. If the time-shifted prediction is required, for example, in an early prediction use curve shifting.

- **Selu activation**. `selu` activation has been deemed as better for large networks. If using `selu` activation then use `kernel_initializer='lecun_normal'` and `AlphaDropout()`. In `AlphaDropout`, use the rate as `0.1`.

4.11 Exercises

1. The chapter mentioned few important properties. In their context, show,

   (a) Why a linearly activated MLP model is equivalent to linear regression? Refer to Appendix A.

   (b) Why is a neural network of a single layer, unit sized layer, and `sigmoid` activation the same as logistic regression?

   (c) How the loss function with dropout under linear activation assumption (Equation 4.8) contains an $\mathcal{L}_2$ regularization term? Refer to Baldi and Sadowski 2013.

   Also, show how the dropout approach is similar to an ensemble method? Refer to Srivastava et al. 2014.

2. **Temporal features**. In (multivariate) time series processes, temporal patterns are naturally present. These patterns are expected to be predictive of the response. Temporal features can be added as predictors to learn such patterns.

   Add the following set of features (one set at a time) to the *baseline* and *final* model of your data. Discuss your findings.

   (a) Up to three lag terms for y and each x's. That is, create samples like,

   $$(\text{response: } y_t, \text{predictors: } y_{t-1}, y_{t-2}, y_{t-3}, \boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \boldsymbol{x}_{t-2}, \boldsymbol{x}_{t-3})$$

   (b) Lag terms up to 10. Does increasing the lags have any effect on the model performance?

   (c) The first derivative of x's, i.e., a sample tuple will be,

   $$(\text{response: } y_t, \text{predictors: } \boldsymbol{x}_t, \boldsymbol{x}_t - \boldsymbol{x}_{t-1})$$

   (d) Second derivatives of x's, i.e.

   $$(\text{response: } y_t, \text{predictors: } \boldsymbol{x}_t, (\boldsymbol{x}_t - \boldsymbol{x}_{t-1}) - (\boldsymbol{x}_{t-1} - \boldsymbol{x}_{t-2}))$$

   (e) Both, first and second derivatives together as predictors.

   (f) Does the derivatives add to the predictability? Why?

   (g) What is the limitation in manually adding features?

(h) (Optional) Frequency domain features. For temporal processes, the features in the frequency domain are known to have good predictive signals. They are more robust to noise in the data. Add frequency domain features as predictors. Discuss your findings.

(i) (Optional) The MLP model constructed in the chapter does not have any temporal features. By itself, the MLP cannot learn temporal patterns. Why did the MLP model without temporal features could still work?

3. **Accuracy metric—Diagnostics Odds Ratio.** Imbalanced data problems need to be evaluated with several non-traditional metrics. A diagnostic odds ratio is one of them.

   (a) Explain diagnostic odds ratio and its interpretation in the context of imbalanced class problems.

   (b) Build a custom metric for the diagnostics odds ratio.

   (c) Add it to the *baseline* and *final* models. Discuss the results.

4. **Batch normalization.** It is mentioned in § 4.7 that extreme feature values can cause vanishing and exploding gradient issues. Batch normalization is another approach to address them.

   (a) Explain the batch normalization approach. Describe it alongside explaining Algorithm 1 in Ioffe and Szegedy 2015.

   (b) Add `BatchNormalization` in TensorFlow to the *baseline* and *final* models. Discuss your results.

   (c) Train the models without feature scaling. Can batch normalization work if the input features are not scaled?

   (d) (Optional) Define a custom layer implementing Algorithm 1 in Ioffe and Szegedy 2015[18]

5. **Dropout.** Understand dropout behavior.

   (a) Dropout does not necessarily work well in shallow MLPs. Build an MLP with more layers and add dropout layers.

[18]The noise and gaussian dropout layers definition here, `https://github.com/tensorflow/tensorflow/blob/v2.1.0/tensorflow/python/keras/layers/noise.py` is helpful.

(b) Build a two hidden layer MLP with larger layer sizes along with dropout.

(c) Discuss the results of (a) and (b).

(d) Srivastava et al. 2014 state that a multiplicative Gaussian noise (now known as Gaussian Dropout) can work better than a regular dropout. Explain the reasoning behind this theory. Repeat (a)-(c) with Gaussian dropout and discuss the results.

6. **Activation.**

(a) `Selu` has one of the best properties among the existing activations. It is believed to work better in deeper networks. Create an MLP network deeper than the baseline and use `selu` activation. Discuss the results.

(b) (Optional) *Thresholded exponential linear unit* (`telu`) is a new activation developed in this chapter. It performed better compared to others. This shows that there is room for developing new activations that might outperform the existing ones.

In this spirit, make the following modification in `telu`.

The threshold τ in Equation 4.14 is fixed. Make τ adaptive by making it proportional to the standard deviation of its input x. Change $\tau = 0.1\sigma_x$ in Equation 4.14 and build customized activation.

The idea is to adjust the threshold based on the input variance.

Apply this customized activation on the *baseline* model and discuss the results.

(c) (Optional) Can the results be further improved with the activation customization? Define your custom activation and test it.

Chapter 5

Long Short Term Memory Networks

5.1 Background

"Humans don't start their thinking from scratch every second. As you read this essay, you understand each word based on your understanding of previous words. You don't throw everything away and start thinking from scratch again. Your thoughts have persistence."

– in Olah 2015.

Sequences and time series processes are like essays. The order of words in an essay and, likewise, the observations in sequences and time series processes are important. Due to this, they have temporal patterns. Meaning, the previous observations (the memory) has an effect on the future.

Memory persistence is one approach to learn such temporal patterns. Recurrent neural networks (RNN) like long- and short-term memory networks were conceptualized for this purpose.

RNNs constitute a very powerful class of computational models capable of learning arbitrary dynamics. They work by keeping a memory of patterns in sequential orders. It combines knowledge from the past memories with the current information to make a prediction.

RNN development can be traced back to Rumelhart, G. E. Hinton, and R. J. Williams 1985. Several RNN variants have been developed since then. For example, Elman 1990, Jordan 1990, and time-delay neural networks by Lang, Waibel, and G. E. Hinton 1990.

However, the majority of the RNNs became obsolete because of their inability to learn long-term memories. This was due to the vanishing gradient issue (explained in the RNN context in § 5.2.8).

The issue was addressed with the development of *long short term memory* (LSTM) networks by Hochreiter and Schmidhuber 1997. LSTMs introduced the concept of cell state that holds the long- and short-term memories.

This ability was revolutionary in the field of RNNs. A majority of the success attributed to RNNs comes from LSTMs. They have proven to work significantly better for most of the temporal and sequential data.

At its inception, LSTMs quickly set several records. It outperformed real-time recurrent learning, back-propagation through time, Elman-nets, and others, popular at the time. LSTM could solve complex and long time-lag tasks that were never solved before.

Developments in and applications of LSTMs continued over the years. Today, they are used by Google and Facebook for text translation (Yonghui Wu et al. 2016; Ong 2017). Apple and Amazon use it for speech recognition in Siri (C. Smith 2016) and Alexa (Vogels 2016), respectively.

Despite the successes, LSTMs have also faced criticisms from some researchers. There has been debate around the need for LSTMs after the development of transformers and attention networks (Vaswani et al. 2017).

Regardless of the varied beliefs, LSTMs and other RNNs still stand as major pillars in deep learning. In addition to their superiority in various problems, there is an immense scope of new research and development.

This chapter begins by explaining the fundamentals of LSTM in § 5.2. LSTMs are one of the most complex constructs in deep learning.

This section attempts at deconstructing it and visualizing each element for a clearer understanding.

The subsections in § 5.2 show an LSTM cell structure (§ 5.2.2), the state mechanism that causes memory persistence (§ 5.2.3), the operations behind the mechanism (§ 5.2.4), the model parameters (§ 5.2.6), and the training iterations (§ 5.2.7). Importantly, LSTMs are capable of persisting long-term memories due to a stable gradient. This property is articulated in § 5.2.8.

Moreover, LSTM networks have an intricate information flow which is seldom visualized but paramount to learn. § 5.3 explains LSTM cell operations and information flow in a network with expanded visual illustrations. The subsections, therein, elucidates *stateless* and *stateful* networks (§ 5.3.2), and the importance of (not) returning sequence outputs (§ 5.3.3).

The chapter then exemplifies an end-to-end implementation of a (baseline) LSTM network in § 5.4 and 5.5. Every step from data preparation, *temporalization*, to network construction are shown. A few network improvement techniques such as *unrestricted* network, *recurrent dropout*, *go-backwards*, and *bi-directional* are explained and implemented in § 5.6.1-5.6.5.

LSTMs have a rich history of development. It has matured to the current level after several iterations of research. § 5.7 walks through a brief history of recurrent neural networks and the evolution of LSTMs.

Lastly, the networks are summarized in § 5.8 and a few rules-of-thumb for LSTM networks are given in § 5.9.

5.2 Fundamentals of LSTM

LSTMs are one of the most abstruse theories in elementary deep learning. Comprehending the fundamentals of LSTM from its original paper(s) can be intimidating.

For an easier understanding, it is deconstructed to its elements and every element is explained in this section. This begins with a typical neural network illustration in Figure 5.1.

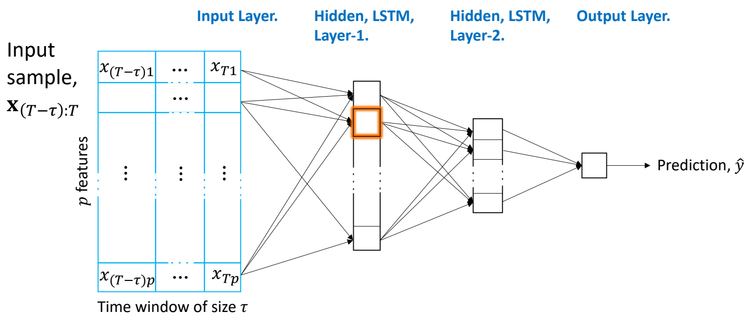


Figure 5.1. *A high-level representation of an LSTM network. The input is a time-window of observations. An LSTM network is designed to learn spatio-temporal relationships from these time-window inputs. The orange highlighted box represents an LSTM cell in the LSTM layer. Each cell learns a distinctive spatio-temporal feature from the inputs.*

5.2.1 Input to LSTM

The input to an LSTM layer is a time-window of observations. In Figure 5.1, it is denoted as $\boldsymbol{x}_{(T-\tau):T}$. This represents p-dimensional observations in a window of size τ.

This window of observations serves as an input sample. The window allows the network to learn the spatial and temporal relationships (remember Figure 2.1c in Chapter 2).

5.2.2 LSTM Cell

The hidden layers in Figure 5.1 are LSTM. The nodes in a layer is an LSTM *cell*—highlighted in orange. A node in LSTM is called a *cell* because it performs a complex biological cell-like multi-step procedure.

This multi-step procedure is enumerated in § 5.2.4. Before getting there, it is important to know the distinguishing property that the cell mechanism brings to LSTM.

The cell mechanism in LSTM has an element called *state*. A cell state can be imagined as a *Pensieve* in *Harry Potter*.

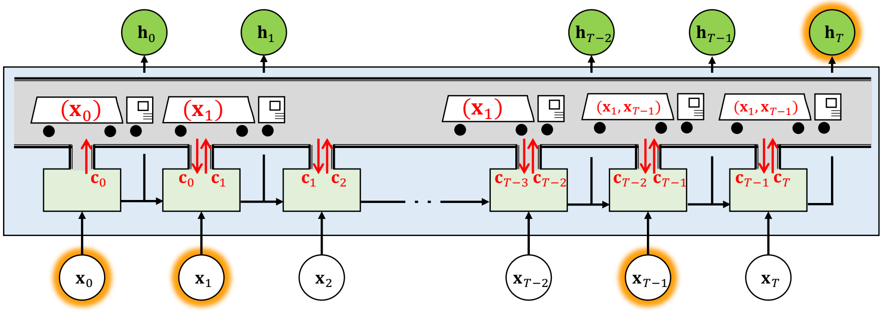
"I use the Pensieve. One simply siphons the excess thoughts from one's mind, pours them into the basin, and examines them at one's leisure. It becomes easier to spot patterns and links; you understand when they are in this form."
–Albus Dumbledore explaining a Pensieve in *Harry Potter and the Goblet of Fire.*

Like a Pensieve, sans magic, a cell state preserves memories from current to distant past. Due to the cell state, it becomes easier to spot patterns and links by having current and distant memories. And, this makes the difference for LSTMs. The state and its mechanism are elaborated in detail next.

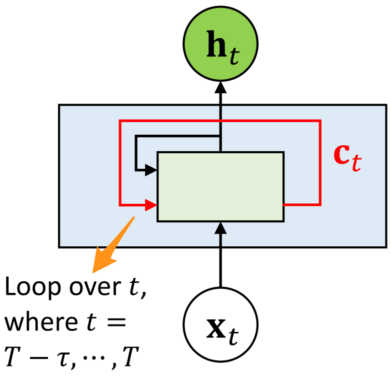
5.2.3 State Mechanism

The cell state mechanism is explained with the help of an intuitive illustration in Figure 5.2a. In the figure, the blue-shaded larger box denotes an LSTM cell. The cell operations are deconstructed inside the box and explained below.

- The input sample to a cell is a time-window of observations $\boldsymbol{x}_{(T-\tau):T}$. For simplicity, $T-\tau$ is replaced with 0 in the figure. The observations are, thus, shown as $\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$.
- The cell sequentially processes the time-indexed observations.
- The iterations are shown as green boxes sequentially laid inside the deconstructed cell.
- A green box takes in one time-step $\boldsymbol{x}_t$. It performs some operations to compute the cell state, $\boldsymbol{c}_t$, and the output, $\boldsymbol{h}_t$.
- Like the other RNNs, the hidden output $\boldsymbol{h}_t$ is transmitted to the next iteration and, also, returned as a cell output. This is shown with branched arrows with horizontal and vertical branches carrying $\boldsymbol{h}_t$. The horizontal branch goes to the next green box (iteration) and the vertical branch exits the cell as an output.
- Differently from the other RNNs, an LSTM cell also transmits the cell state $\boldsymbol{c}_t$.



(a) *Illustration of an unwrapped LSTM cell mechanism showing the time-step iterations. The input to a cell is the time-window of observations* $\boldsymbol{x}_{(T-\tau):T}$. *The index* $(T-\tau)$ *is replaced with* 0 *for simplicity. The LSTM cell acts as a memory lane in which the cell states carry the long- and short-term memories through an imaginary truck of information.*



(b) *A condensed form of the LSTM cell mechanism. In actuality, the cell states* $(\boldsymbol{h}_t, \boldsymbol{c}_t)$ *are re-accessed iteratively for* $t = (T-\tau), \ldots, T$. *A succinct representation of this is shown with the looping arrows.*

Figure 5.2. *An unwrapped and condensed (wrapped) illustration of LSTM cell state mechanism.*

- Imagine the iterations along the time-steps, $t = 0, \ldots, T$, in a cell as a drive down a lane. Let's call it a "memory lane." A green box (the time-step iteration) is a station on this lane. And, there is a "truck of information" carrying the cell state, i.e., the memory.

- The truck starts from the left at the first station. At this station, the inputted observation $\boldsymbol{x}_0$ is assessed to see whether the information therein is relevant or not. If yes, it is loaded on to the truck. Otherwise, it is ignored.

- The *loading* on the truck is the cell state. In the figure's illustration, $\boldsymbol{x}_0$ is shown as important and loaded to the truck as a part of the cell state.

- The cell state $\boldsymbol{c}_t$ as truckloads are denoted as $(\boldsymbol{x}.)$ to express that the state is some function of the $\boldsymbol{x}$'s and not the original $\boldsymbol{x}$.

- The truck then moves to the next station. Here it is unloaded, i.e., the state/memory learned thus far is taken out. The station assesses the unloaded state alongside the $\boldsymbol{x}$ available in it.

- Suppose this station is t. Two assessments are made here. First, is the information in the $\boldsymbol{x}_t$ at the station relevant? If yes, add it to the state $\boldsymbol{c}_t$.

  Second, in the presence of $\boldsymbol{x}_t$ is the memory from the prior $\boldsymbol{x}$'s still relevant? If irrelevant, forget the memory.

  For example, the station next to $\boldsymbol{x}_0$ is $\boldsymbol{x}_1$. Here $\boldsymbol{x}_1$ is found to be relevant and added in the state. At the same time, it is found that $\boldsymbol{x}_0$ is irrelevant in the presence of $\boldsymbol{x}_1$. And, therefore, the memory of $\boldsymbol{x}_0$ is taken out of the state. Or, in LSTM terminology, $\boldsymbol{x}_0$ is forgotten.

- After the processing, the state is then loaded back on the truck.

- The process of loading and unloading the truck-of-information is repeated till the last $\boldsymbol{x}_T$ in the sample. Further down the lane, it is shown that $\boldsymbol{x}_2, \boldsymbol{x}_{T-2}$ and $\boldsymbol{x}_T$ contain irrelevant information. They are ignored while $\boldsymbol{x}_{T-1}$ was added as its information might be absent in the state $\boldsymbol{c}_{T-2}$ learned before reaching it.

- The addition and omission of the timed observations of a sample makes up the **long-term** memory in $\boldsymbol{c}_t$.

- Moreover, the intermediate outputs $\boldsymbol{h}_t$ has a **short-term** memory.

- Together, they constitute all the long- and short-term memories that lead up to deliver the final output $\boldsymbol{h}_T$ at the last station.

If you are still thinking of Harry Potter, the truck is the Pensieve.

Using the cell state, LSTM becomes capable of preserving memories from the past. **But, why was this not possible with the RNNs before LSTM?**

Because the gradients vanish quickly for the $\boldsymbol{h}_t$'s. As a result, long-term memories do not persist. Differently, the cell states $\boldsymbol{c}_t$ in LSTM has stabilized gradient (discussed in § 5.2.8) and, thus, keeps the memory.

5.2.4 Cell Operations

An LSTM cell behaves like a living cell. It performs multiple operations to learn and preserve memories to draw inferences (the output).

Consider a cell operation in an LSTM layer in Figure 5.3. The cell processes one observation at a time in a timed sequence window $\{\boldsymbol{x}_{T-\tau}, \boldsymbol{x}_{T-\tau+1}, \ldots, \boldsymbol{x}_T\}$.

Suppose the cell is processing a time-step $\boldsymbol{x}_t$. The $\boldsymbol{x}_t$ flows into the cell as input, gets processed along the paths (in Figure 5.3) in the presence of the previous output $\boldsymbol{h}_{t-1}$ and cell state $\boldsymbol{c}_{t-1}$, and yields the updated output $\boldsymbol{h}_t$ and cell state $\boldsymbol{c}_t$.

The computations within the cell as implemented in TensorFlow from Jozefowicz, Zaremba, and Sutskever 2015 are given below in Equation 5.1a-5.1f,

$$i_t = \texttt{hard-sigmoid}(\boldsymbol{w}_i^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_i^{(h)}\boldsymbol{h}_{t-1} + b_i) \tag{5.1a}$$
$$o_t = \texttt{hard-sigmoid}(\boldsymbol{w}_o^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_o^{(h)}\boldsymbol{h}_{t-1} + b_o) \tag{5.1b}$$
$$f_t = \texttt{hard-sigmoid}(\boldsymbol{w}_f^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_f^{(h)}\boldsymbol{h}_{t-1} + b_f) \tag{5.1c}$$
$$\tilde{c}_t = \texttt{tanh}(\boldsymbol{w}_c^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_c^{(h)}\boldsymbol{h}_{t-1} + b_c) \tag{5.1d}$$
$$c_t = f_t c_{t-1} + i_t \tilde{c}_t \tag{5.1e}$$
$$h_t = o_t \texttt{tanh}(c_t) \tag{5.1f}$$

where,

- i_t, o_t, and f_t are input, output, and forget gates,
- `hard-sigmoid` is a segment-wise linear approximation of `sigmoid` function for faster computation. It returns a value between 0 and 1, defined as,

$$\texttt{hard-sigmoid}(x) = \begin{cases} 1 & , x > 2.5 \\ 0.2x + 0.5 & , -2.5 \leq x \leq 2.5 \\ 0 & , x < -2.5 \end{cases}$$

- $\tilde{c}_t$ is a temporary variable that holds the relevant information in the current time-step t,
- c_t and h_t are the cell state and outputs, and
- $\boldsymbol{w}_{\cdot}^{(x)}$, $\boldsymbol{w}_{\cdot}^{(h)}$, and $b_{\cdot}$ are the weight and bias parameters.

The intuition behind processing operations in Equation 5.1a-5.1f are broken into four steps and described below.

- **Step 1. Information.**

  The first step is to learn the information in the time-step input $\boldsymbol{x}_t$ alongside the cell's prior learned output $\boldsymbol{h}_{t-1}$. This learning is done

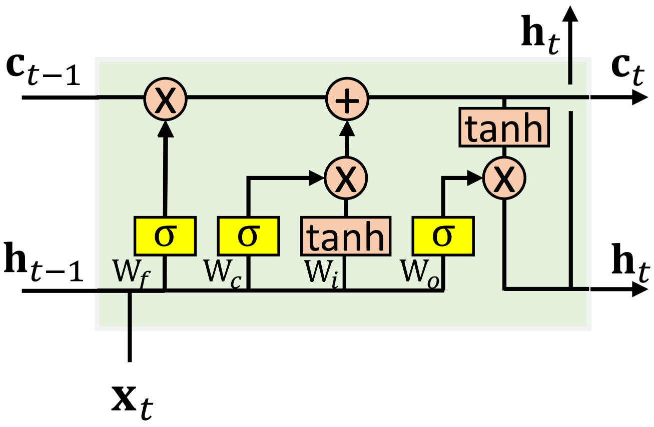
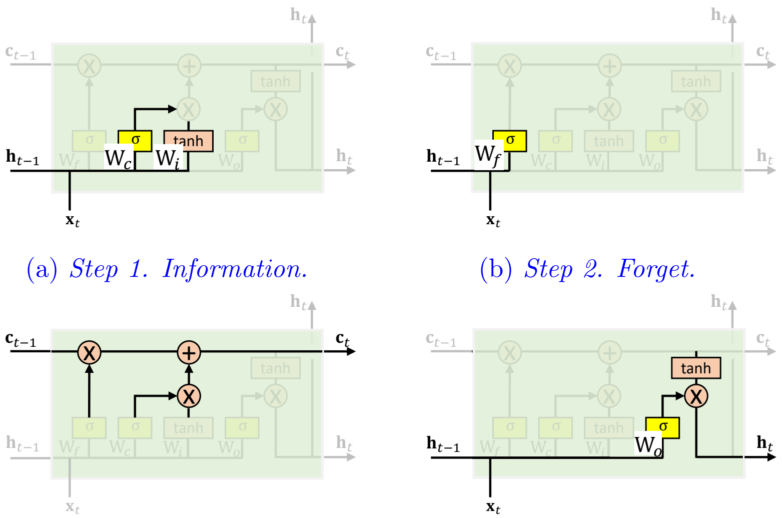


Figure 5.3. *The inside of an LSTM cell. The cell consists of three gates, input (i), output (o), and forget (f), made of sigmoid activation (σ) shown with yellow boxes. The cell derives relevant information through* **tanh** *activations shown with orange boxes. The cell takes the prior states ($\mathbf{c}_{t-1}$, $\mathbf{h}_{t-1}$), runs it through the gates, and draws information to yield the updated ($\mathbf{c}_t$, $\mathbf{h}_t$). Source Olah 2015.*


(a) *Step 1. Information.*

(b) *Step 2. Forget.*

(c) *Step 3. Memory.*

(d) *Step 4. Output.*

Figure 5.4. *Operations steps in an LSTM cell. Source Olah 2015.*

in two sub-steps given in Equation 5.1d and 5.1a and succinctly expressed below.

$$\tilde{c}_t = g\big(\boldsymbol{w}_c^{(h)}\boldsymbol{h}_{t-1} + \boldsymbol{w}_c^{(x)}\boldsymbol{x}_t + b_c\big)$$
$$i_t = \sigma\big(\boldsymbol{w}_i^{(h)}\boldsymbol{h}_{t-1} + \boldsymbol{w}_i^{(x)}\boldsymbol{x}_t + b_i\big)$$

The first equation finds the relevant information in $\boldsymbol{x}_t$. The equation applies a `tanh` activation. This activation has negative and positive values in $(-1, 1)$. The reason for using `tanh` is in step 3.

It is possible that $\boldsymbol{x}_t$ has information but it is redundant or irrelevant in the presence of the information already present with the cell from the previous $\boldsymbol{x}$'s.

To measure the relevancy, i_t is computed in the second equation. It is activated with `sigmoid` to have a value in $(0, 1)$. A value closer to 0 would mean the information is irrelevant and vice-versa.

- **Step 2. Forget.**

  Due to the new information coming in with $\boldsymbol{x}_t$, some of the previous memory may become immaterial. In that case, that memory can be *forgotten.*

  This forgetting decision is made at the forget gate in Equation 5.1c.

$$f_t = \sigma\big(\boldsymbol{w}_f^{(h)}\boldsymbol{h}_{t-1} + \boldsymbol{w}_f^{(x)}\boldsymbol{x}_t + b_f\big)$$

  The expression is `sigmoid` activated which yields an indicator between 0 and 1. If the indicator is close to zero, the past memory is forgotten. In this case, the information in $\boldsymbol{x}_t$ will replace the past memory.

  If the indicator is close to one, it means the memory is still pertinent and should be carried forward. But this does not necessarily indicate that the information in $\boldsymbol{x}_t$ is irrelevant or will not enter the memory.

- **Step 3. Memory.**

The previous steps find the information in $\boldsymbol{x}_t$, its relevance, and the need for the memory. These are concocted together to update the cell memory in Equation 5.1e.

$$c_t = f_t c_{t-1} + i_t \tilde{c}_t$$

The first component determines whether to carry on the memory from the past. If the forget gate is asking to forget, i.e., $f_t \rightarrow 0$, the cell's memory is bygone.

The second component in the equation, $i_t \tilde{c}_t$, is from step 1. This is a product of the relevance indicator, i_t, and the information, $\tilde{c}_t$. Ultimately, the product contains the relevant information found at t. Its value lies between -1 and $+1$. Depending on the different scenarios given in Table 5.1, they become close to zero or $|1|$.

As depicted in the table, the best scenario for $\boldsymbol{x}_t$ is when it has *relevant* information. In this case, both the magnitude of $|\tilde{c}_t|$ and i_t will be close to 1, and the resultant product will be far from 0. This scenario is at the bottom-right of the table.

The $i_t \tilde{c}_t$ component also makes the need for using a `tanh` activation in Equation 5.1d apparent.

Imagine using a positive-valued activation instead. In that case, $i_t \tilde{c}_t$ will be always positive. But since the $i_t \tilde{c}_t$ is added to state c_t in each iteration in $0, 1, \ldots, T$, a positive $i_t \tilde{c}_t$ will move c_t in an only positive direction. This can cause the state c_t to inflate.

On the other hand, `tanh` has negative to positive values in $(-1, 1)$. This allows the state to increase or decrease. And, keeps the state tractable.

Still, `tanh` is not mandatory. A positive-valued activation can also be used. This chapter implements `relu` activation in § 5.5 and 5.6, where it is found to work better than `tanh` when the input is scaled to Gaussian(0,1).

- **Step 4. Output.**

  At the last step, the cell output h_t is determined in Equation 5.1f. The output h_t is drawn from two components. One of them is the

Table 5.1. Scenarios of information present in a time-step $\boldsymbol{x}_t$ and its relevance in presence of the past memory.

		Information magnitude, $\lvert\tilde{c}_t\rvert$, close to,	
		0	**1**
Information relevance, i_t, close to,	**0**	No information in $\boldsymbol{x}_t$.	$\boldsymbol{x}_t$ has redundant information already present in the memory.
	1	Model (weights) inconsistent.	$\boldsymbol{x}_t$ has relevant new information.

output gate o_t that acts as a scale with value in $(0, 1)$. The other is a `tanh` activated value of the updated cell state c_t in step 3.

$$
\begin{aligned}
o_t &= \sigma\big(\boldsymbol{w}_o^{(h)}\boldsymbol{h}_{t-1} + \boldsymbol{w}_o^{(h)}\boldsymbol{x}_t + b_o\big) \\
h_t &= o_t \tanh(c_t)
\end{aligned}
$$

The h_t expression behaves like short-term memory. And, therefore, h_t is also called a short state in TensorFlow.

These steps are shown in Figure 5.4a-5.4d. In each figure, the paths corresponding to a step are highlighted. Besides, the order of step 1 and 2 are interchangeable. But the subsequent steps 3 and 4 are necessarily in the same order.

5.2.5 Activations in LSTM

The activations in Equation 5.1d for $\tilde{c}_t$ and 5.1f for emitting h_t correspond to the `activation` argument in an `LSTM` layer in TensorFlow. By default, it is `tanh`. These expressions act as learned features and, therefore, can take any value. With `tanh` activation, they are in $(-1, 1)$. Other suitable activations can also be used for them.

On the other hand, the activations in Equation 5.1a-5.1c for input, output, and forget gates are referred to as the argument `recurrent_activation` in TensorFlow. These gates act as scales. Therefore, they are intended to stay in $(0, 1)$. Their default is, hence, `sigmoid`. For most purposes, it is essential to keep `recurrent_activation` as `sigmoid` (explained in § 5.2.8).

The `recurrent_activation` should be `sigmoid`. The default `activation` is `tanh` but can be set to other activations such as `relu`.

5.2.6 Parameters

Suppose an LSTM layer has m cells, i.e., the layer size equal to m. The cell mechanism illustrated in the previous section is for one cell in an LSTM layer. The parameters involved in the cell are, $\boldsymbol{w}_{\cdot}^{(h)}, \boldsymbol{w}_{\cdot}^{(x)}, b_{\cdot}$, where $\cdot$ is c, i, f, and o.

A cell intakes the prior output of all the other sibling cells in the layer. Given the layer size is m, the prior output from the layer cells will be an m-vector $\boldsymbol{h}_{t-1}$ and, therefore, the $\boldsymbol{w}_{\cdot}^{(h)}$ are also of the same length m.

The weight for the input time-step $\boldsymbol{x}_t$ is a p-vector given there are p features, i.e., $\boldsymbol{x}_t \in \mathbb{R}^p$. Lastly, the bias on a cell is a scalar.

Combining them for each of c, i, f, o, the total number of parameters in a cell is $4(m + p + 1)$.

In the LSTM layer, there are m cells. Therefore, the total number of parameters in a layer are,

$$n\_parameters = 4m(m + p + 1) \tag{5.2}$$

It is important to note here that **the number of parameters is independent of the number of time-steps processed by the cell**. That is, they are independent of the window size τ.

This implies that the parameter space does not increase if the window size is expanded to learn longer-term temporal patterns. While this might appear an advantage, in practice, the performance deteriorates after a certain limit on the window size (discussed more later in § 5.6.5).

An LSTM layer has $4m(m + p + 1)$ parameters, where m is the size of the layer and p the number of features in the input.

The number of LSTM parameters are independent of the sample window size.

5.2.7 Iteration Levels

A sample in LSTM is a window of time-step observations. Due to this, its iterations level shown in Figure 5.5 goes one level further than in MLPs (in Figure 4.14). In LSTMs, the iterations end at a time-step.

Within a time-step, all the cell operations described in § 5.2.4 are executed. But, unlike MLPs, the cell operations cannot be done directly with the tensor operation.

The cell operations in a time-step are sequential. The output of a time-step goes as an input to the next. Due to this, the time-steps are processed one-by-one. Furthermore, the steps within a time-step are also in order.

Because of the sequential operations, LSTM iterations are computationally intensive. However, samples within a batch do not interact in *stateless* LSTMs (default in TensorFlow) and, therefore, a batch is processed together using tensor operations.

5.2.8 Stabilized Gradient

The concept of keeping the memory from anywhere in the past was always present in RNNs. However, before LSTMs the RNN models were unable to learn long-term dependencies due to vanishing and exploding gradient issues.

```
epoch
 └─batch
    └─sample
       └─time-step
          ├─step-1 information
          ├─step-2 forget
          ├─step-3 memory
          └─step-4 output
```

Figure 5.5. *LSTM training iteration levels.*

This was achieved with the introduction of a cell state in LSTMs. The cell state stabilized the gradient. This section provides a simplified explanation behind this.

The output of a cell is h_T. Suppose the target is y_T, and we have a square loss function,

$$\mathcal{L}_T = (y_T - h_T)^2.$$

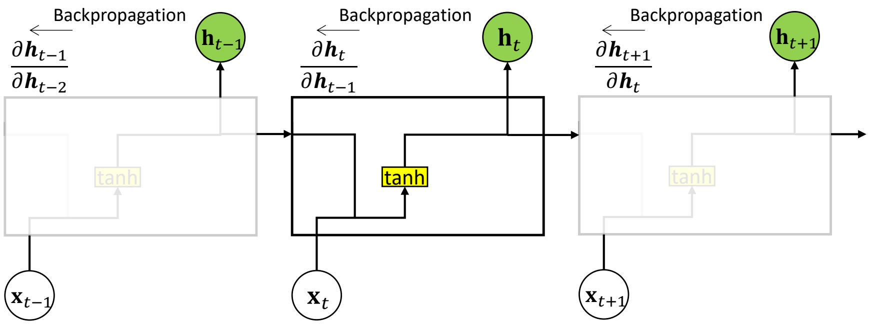
During the model estimation, the gradient of the loss with respect to a parameter is taken. Consider the gradient for a weight parameter,

$$\frac{\partial}{\partial w}\mathcal{L}_T = -2\underbrace{(y_T - h_T)}_{error}\frac{\partial h_T}{\partial w}.$$

The term $(y_T - h_T)$ is the model error. During model training, the need is to propagate this error for model parameter update.

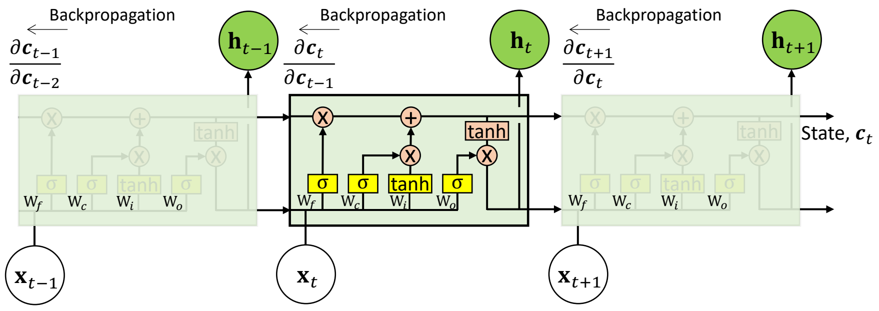
Whether the error appropriately propagates or not depends on the derivative $\frac{\partial h_T}{\partial w}$. If $\frac{\partial h_T}{\partial w}$ vanishes or explodes, the error gets distorted and model training suffers.

This was the case with the simple RNNs (refer to Figure 5.6a). In a simple RNN, there is no cell state. The cell output h_T is a function of the prior time-step output h_{T-1}.



Backpropagation
$\frac{\partial \boldsymbol{h}_{t-1}}{\partial \boldsymbol{h}_{t-2}}$
$\mathbf{h}_{t-1}$
tanh
$\mathbf{x}_{t-1}$
Backpropagation
$\frac{\partial \boldsymbol{h}_{t}}{\partial \boldsymbol{h}_{t-1}}$
$\mathbf{h}_{t}$
tanh
$\mathbf{x}_{t}$
Backpropagation
$\frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_{t}}$
$\mathbf{h}_{t+1}$
tanh
$\mathbf{x}_{t+1}$

(a) *Simple RNN. The error propagates along the gradient of the hidden outputs* $\frac{h_t}{h_{t-1}}$ *which can explode or vanish.*



Backpropagation
$\frac{\partial \boldsymbol{c}_{t-1}}{\partial \boldsymbol{c}_{t-2}}$
$\mathbf{h}_{t-1}$
Backpropagation
$\frac{\partial \boldsymbol{c}_{t}}{\partial \boldsymbol{c}_{t-1}}$
$\mathbf{h}_{t}$
Backpropagation
$\frac{\partial \boldsymbol{c}_{t+1}}{\partial \boldsymbol{c}_{t}}$
$\mathbf{h}_{t+1}$
X
+
tanh
X
X
σ
σ
tanh
σ
W_f
W_c
W_i
W_o
State, $\boldsymbol{c}_t$
$\mathbf{x}_{t-1}$
$\mathbf{x}_{t}$
$\mathbf{x}_{t+1}$

(b) *LSTM backpropagation. The error propagates along the gradients of the cell states* $\frac{c_t}{c_{t-1}}$ *which are stable.*

Figure 5.6. *Illustration of backpropagation in a simple RNN (top) and LSTM (bottom).*

$$h_T \propto g(h_{T-1}).$$

Therefore, the derivative $\frac{\partial h_T}{\partial w}$ will be,

$$\frac{\partial h_T}{\partial w} \propto \frac{\partial h_T}{\partial h_{T-1}} \underbrace{\frac{\partial h_{T-1}}{\partial h_{T-2}} \cdots \frac{\partial h_{T-\tau+1}}{\partial h_{T-\tau}}}_{\text{can explode or vanish}} \frac{\partial h_{T-\tau}}{\partial w}. \tag{5.3}$$

As shown in Equation 5.3, the derivative $\frac{\partial h_T}{\partial w}$ is on the mercy of the chain product. A chain product is difficult to control. Since $\frac{\partial h_t}{\partial h_{t-1}}$ can take any value, the chain product can **explode** or **vanish**.

On the contrary, consider the LSTM Equations 5.1e-5.1f defined in § 5.2.4,

$$\begin{aligned} c_t &= f_t c_{t-1} + i_t \tilde{c}_t \\ h_t &= o_t \texttt{tanh}(c_t). \end{aligned}$$

Unlike a simple RNN, LSTM emits two outputs (refer to Figure 5.6b) in each time-step: a) a slow state c_t which is the cell state or the long-term memory, and b) a fast state h_t which is the cell output or the short-term memory.

Computing the derivative for LSTM from the expression $h_t = o_t \texttt{tanh}(c_t)$,

$$\frac{\partial h_T}{\partial w} \propto \frac{\partial h_T}{\partial c_T} \frac{\partial c_T}{\partial c_{T-1}} \frac{\partial c_{T-1}}{\partial c_{T-2}} \cdots \frac{\partial c_{T-\tau+1}}{\partial c_{T-\tau}} \frac{\partial c_{T-\tau}}{\partial w}. \tag{5.4}$$

Assume the forget gate f_t used in $c_t = f_t c_{t-1} + i_t \tilde{c}_t$ is inactive throughout the window $(T - \tau) : T$, i.e., $f_t = 1$, $t = T - \tau, \ldots, T$. Then,

$$c_t = c_{t-1} + i_t \tilde{c}_t.$$

In this scenario, $\frac{\partial c_t}{\partial c_{t-1}} = 1$. And, consequently, $\frac{\partial h_T}{\partial w}$ in Equation 5.4 becomes,

$$\frac{\partial h_T}{\partial w} \propto \frac{\partial h_T}{\partial c_T} \underbrace{\prod 1}_{\text{stabilizes the gradient}} \frac{\partial c_{T-\tau}}{\partial w} \tag{5.5}$$

This derivative in Equation 5.5 is now stable. It does not have a chain multiplication of elements that can take any value. Instead, it is replaced with a chain multiplication of 1's.

The cell state, thus, enables the error $(y_T - h_T)$ to propagate down the time-steps. In the scenario when the forget gate is closed $(f_t = 0)$ at some time-step t in $(T - \tau) : T$, the derivative becomes zero. This scenario means the memory/information before t is irrelevant. Therefore, learning/error propagation before t is not needed. In this case, the derivative desirably becomes zero.

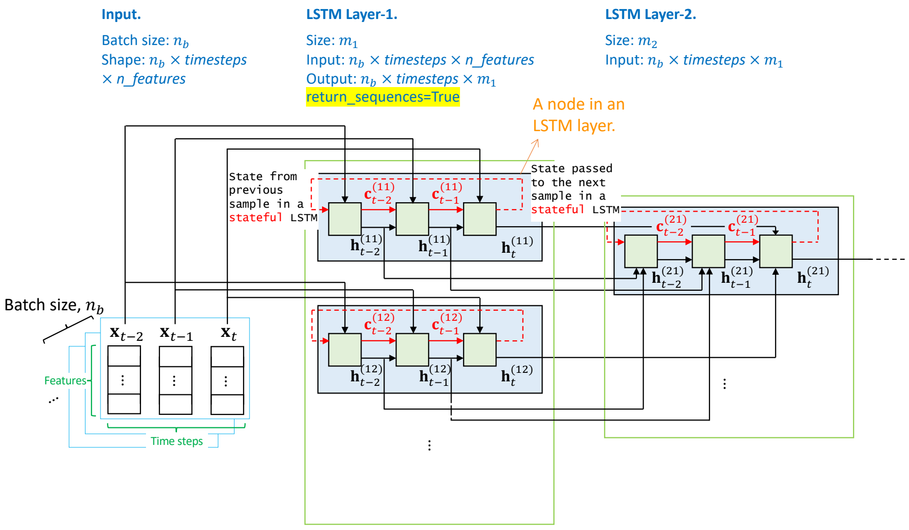
It is important to note that this stabilizing property in LSTM is achieved due to the additive auto-regression like $c_t = c_{t-1} + i_t \tilde{c}_t$, expression for the cell state. The additive expression makes the cell state the **long-term** memory.

The additive nature of the cell state c_t provides a stabilized gradient that enables the error to propagate down the time-steps. Due to this, long-term information can be preserved in c_t.

Besides, the stabilizing property is achieved if the forget gate takes value in $(0, 1)$. It is, therefore, essential that `recurrent_activation` is `sigmoid` as mentioned in § 5.2.5.

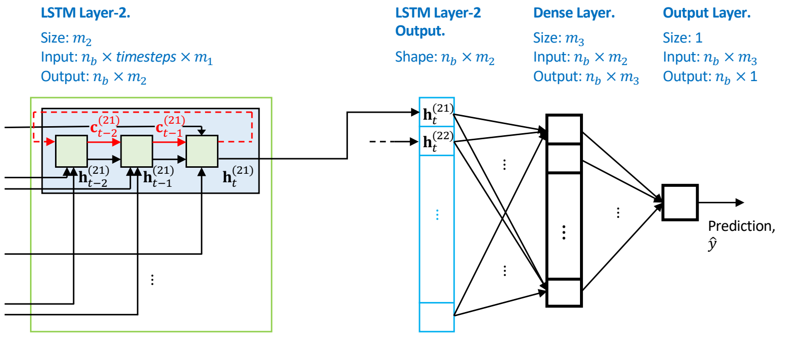
5.3 LSTM Layer and Network Structure

The previous section illustrated the LSTM fundamentals by deconstructing an LSTM cell in Figure 5.2. The cell input-output and the operations

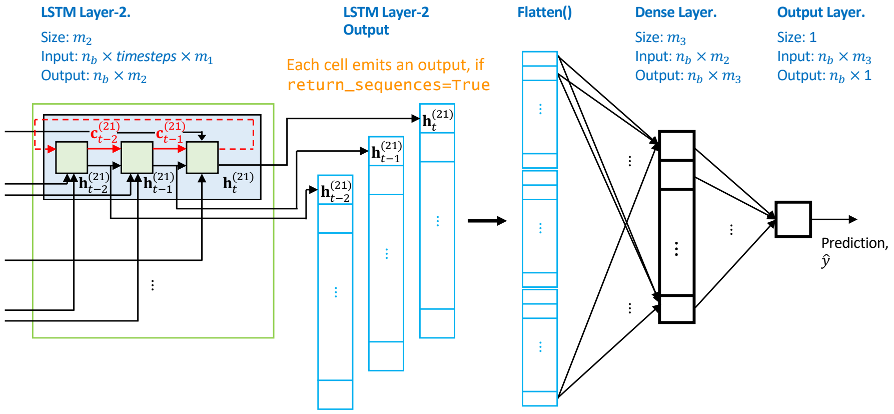

Input.
Batch size: n_b
Shape: n_b × timesteps × n_features
LSTM Layer-1.
Size: m_1
Input: n_b × timesteps × n_features
Output: n_b × timesteps × m_1
return_sequences=True
LSTM Layer-2.
Size: m_2
Input: n_b × timesteps × m_1
A node in an LSTM layer.
State from previous sample in a stateful LSTM
State passed to the next sample in a stateful LSTM
Batch size, n_b
Features
Time steps
$\mathbf{x}_{t-2}$ $\mathbf{x}_{t-1}$ $\mathbf{x}_t$
$\mathbf{c}_{t-2}^{(11)}$ $\mathbf{c}_{t-1}^{(11)}$
$\mathbf{h}_{t-2}^{(11)}$ $\mathbf{h}_{t-1}^{(11)}$ $\mathbf{h}_t^{(11)}$
$\mathbf{c}_{t-2}^{(12)}$ $\mathbf{c}_{t-1}^{(12)}$
$\mathbf{h}_{t-2}^{(12)}$ $\mathbf{h}_{t-1}^{(12)}$ $\mathbf{h}_t^{(12)}$
$\mathbf{c}_{t-2}^{(21)}$ $\mathbf{c}_{t-1}^{(21)}$
$\mathbf{h}_{t-2}^{(21)}$ $\mathbf{h}_{t-1}^{(21)}$ $\mathbf{h}_t^{(21)}$

(a) *LSTM network input and hidden layers. The input is a batch of time-window of observations. This makes each sample in a batch a 2D array and the input batch a 3D array. The time-window is arbitrarily takes as three for illustration. The cells in blue boxes within the hidden LSTM layer is unwrapped to their time-step iterations shown with green boxes. The connected arcs show the transmission of time-indexed information between the layers. The first LSTM layer is emitting sequences (`LSTM(...,return_sequences=True)`). These sequences have the same notional time order as the input and are processed in the same order by the second LSTM layer. If the model is stateful, the cell state from the prior batch processing is preserved and accessed by the next batch.*

Figure 5.7. *LSTM Network Structure. Part I.*



(b1) *Restricted LSTM network. In a restricted network, the last LSTM layer emits only the final hidden output. As shown above, the second LSTM layer returns only the last $\boldsymbol{h}_t$'s from each cell which makes up the feature map vector for input to the output dense layer (`(LSTM(..., return_sequences=False)`).*



(b2) *Unrestricted LSTM network. In an unrestricted network, the hidden outputs at each time-step, i.e., a sequence of outputs $\{\boldsymbol{h}_{t-\tau}, \ldots, \boldsymbol{h}_t\}$, are returned (`(LSTM(..., return_sequences=False)`). This makes up a 2D feature map of shape, (layer size, time-steps).*

Figure 5.7. *LSTM Network Structure. Part II.*

therein were explained. In this section, the view is zoomed out of the cell and the network operations at the layer level are explained.

Figure 5.7 here brings an LSTM layer's internal and external connections into perspective. It provides visibility on the layer's input-output mechanism.

The illustrations provide an understanding of stacking layers around an LSTM layer, and the way they interact. Their references to TensorFlow modeling is also given in this section.

Earlier, Figure 5.1 showed an abstraction of LSTM network. The network layers are expanded in Figure 5.7a-5.7b2. The expanded view is split into two parts for clarity. Figure 5.7a shows the left part of the network—from input to the middle. And Figure 5.7b1-5.7b2 show its continuation till the output in two major LSTM modes, viz., `return_sequences` is `True` versus `False`.

In the figures, the blue-shaded boxes in the LSTM layers are the layer's cells. And, as before, the green boxes within a cell is representative of a time-step iteration for input processing.

5.3.1 Input Processing

As also mentioned in § 5.2.1, LSTM takes a time-window of observations as an input sample. A τ sized time-window of p-dimensional observations $\boldsymbol{x}$ denoted as $\boldsymbol{x}_{(T-\tau):T}$ is a two-dimensional array. A batch of such samples is, thus, a three-dimensional array of shape: (n_batch, $timesteps$, $n\_features$).

The first LSTM layer in Figure 5.7a takes in samples from the input batch. A sample is shown as a two-dimensional array with features and time-steps along the rows and columns, respectively.

Each of the time-steps is processed sequentially. For illustration, this is shown by connecting each input time-step with the corresponding time-step iteration in the cell.

Similarly, every cell in the layer takes all the time-step inputs. The cells transmit the cell states and hidden states within themselves to perform their internal operations (described in four steps in § 5.2.4).

The interesting part is the way the states are transmitted outside. There are two major transmission modes to understand in LSTM: **stateful** and **return sequences**. These modes allow building a stateful or stateless LSTM, and/or (not) return sequences. They are described next.

5.3.2 Stateless versus Stateful

Stateless

An LSTM layer is built to learn temporal patterns. A default LSTM layer in TensorFlow learns these patterns in a time-window of observations $\boldsymbol{x}_{(T-\tau)} : T$ presented to it. This default setting is called a *stateless* LSTM and is enforced with `LSTM(..., stateful=False)`.

It is called *stateless* because the cell states are transmitted and contained within the time-window $(T - \tau) : T$. This means a long-term pattern will be only up to τ long.

Also, in this setting, the model processes each time-window independently. That is, there is no interaction or learning between two time-windows. Consequently, the default input sample shuffle during model training is allowed.

Stateful

A stateless model constrains LSTM to learn patterns only within a fixed time-window. It does not provide visibility beyond the window.

But LSTMs were conceived to learn any long-term patterns. One might desire to learn patterns as long back in the past as the data goes.

The *stateful* mode in LSTM layers enables this with `LSTM(..., stateful=True)`. In this mode, the last hidden and cell states of a batch go back in the LSTM cell as the initial hidden and cell states for the next batch.

This is shown with dashed lines exiting and entering a cell in Figure 5.7a. Note again that this reuse of states happens in stateful mode only. Also, in this mode, the batches are not shuffled, i.e., `model.fit(..., shuffle=False)`. This is to maintain the time order of the samples to use the learning from the chronologically previous sample.

This mode appears tempting. It promises to provide a useful power of learning exhaustive long-term patterns. Still, it is and should **not** be a preferred choice in most problems.

Stateful LSTM is appropriate if the time series/sequence data set is stationary. In simple words, it means if the relationships stay the same from the beginning to the end (in time). For example, text documents.

However, this is not true in many real-world problems. And, therefore, a stateful LSTM is not the default choice.

5.3.3 Return Sequences vs Last Output

Return sequences

The cells in an LSTM layer can be made to return sequences as an output by setting `LSTM(..., return_sequences=True)`. In this setting, each cell emits a sequence of length same as the input. Therefore, if a layer has l cells, the output shape is (n_batch, $timesteps$, l).

Sequences should be returned when the temporal structure needs to be preserved. This requirement is usually when

- the model output is a sequence. For example, in sequence-to-sequence models for language translation. Or,
- the subsequent layer needs sequence inputs. For example, a stack of LSTM (or convolutional) layers to sequentially extract temporal patterns at different levels of abstraction.

The illustration in Figure 5.7a is sequentially extracting temporal patterns using two LSTM layers. For this, the first LSTM layer (`LSTM Layer-1`) has to return a sequence.

This is shown in the figure in which the time-step iterations (green boxes within a cell) emit an output that is transmitted to the corresponding time-step iteration in the cell of the next LSTM layer.

In this network, returning sequences from the last LSTM layer

(`LSTM Layer-2`) becomes a choice. Figure 5.7b2 shows this choice. In this setting, a cell emits a sequence of outputs. The output is a sequence $\{\boldsymbol{h}_{T-2}, \boldsymbol{h}_{T-1}, \boldsymbol{h}_T\}$ with the same time-steps as the input sample $\{\boldsymbol{x}_{T-2}, \boldsymbol{x}_{T-1}, \boldsymbol{x}_T\}$. This output is flattened to a vector before sending to the output dense layer[1].

The last LSTM layer emitting sequences is termed as *unrestricted* LSTM network. Because the model is not restricting the output layer to use only the last outputs of the LSTM cells. These networks are larger but have the potential to yield better results.

Return last output

In this setting, a cell in an LSTM layer emits only the last time-step output $\boldsymbol{h}_t$. This is done by setting `LSTM(..., return_sequences=False)`. The output shape is $(n\_batch,\, l)$.

The last time-step output $\boldsymbol{h}_T$ is an amalgamation of information present in all the cell states $\{\boldsymbol{c}_T, \boldsymbol{c}_{T-1}, \ldots, \boldsymbol{c}_{T-\tau}\}$ and the prior cell outputs
$\{\boldsymbol{h}_{T-1}, \boldsymbol{h}_{T-2}, \ldots, \boldsymbol{h}_{T-\tau}\}$.

This is usually required in the following scenarios,

- The encoder in an LSTM autoencoder. The encodings are generally a vector. In some LSTM autoencoders, the encoder LSTM layer emits the last output vector as the encodings.

- Sequence to scaler model. A classifier is a good example of such models in which the input is a sequence and the output is a class (a scaler).

Figure 5.7b1 illustrates this setting in the last layer of the LSTM network. It is called a *restricted* LSTM network because the last layer's output is restricted. As shown in the figure, only the last time-step output $\boldsymbol{h}_T$ (from the last green box) is emitted and sent to the next layer.

[1]Flattening is, although, optional in TensorFlow because the dense layer automatically takes shape based on the input.

5.4 Initialization and Data Preparation

5.4.1 Imports and Data

We get started with importing the libraries. LSTM related classes are taken from `tensorflow` library. Also, the user-defined libraries, viz. `datapreprocessing`, `performancemetrics`, and `simpleplots`, are imported.

Listing 5.1. LSTM library imports.

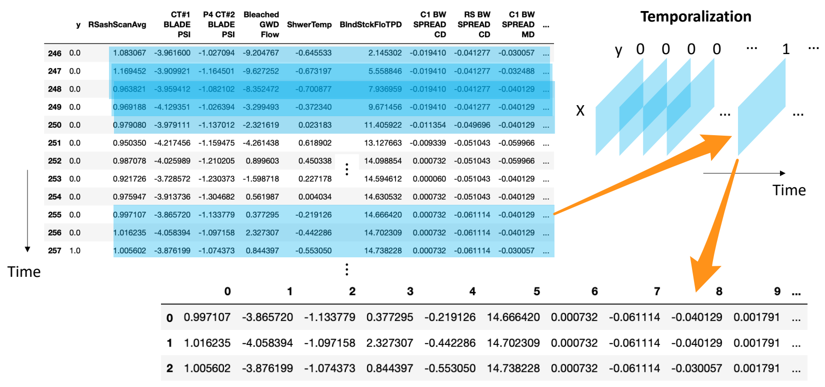
```python
import pandas as pd
import numpy as np

import tensorflow as tf
from tensorflow.keras import optimizers
from tensorflow.keras.models import Model
from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Input
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import LSTM

from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Bidirectional

from tensorflow.python.keras import backend as K

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

import matplotlib.pyplot as plt
import seaborn as sns

# user-defined libraries
import utilities.datapreprocessing as dp
import utilities.performancemetrics as pm
import utilities.simpleplots as sp

from numpy.random import seed
seed(1)
```

```

SEED = 123  # used to help randomly select the data
    points
DATA_SPLIT_PCT = 0.2

from pylab import rcParams
rcParams['figure.figsize'] = 8, 6
plt.rcParams.update({'font.size': 22})
```

Next, the data is read and the basic pre-processing steps are performed.

Listing 5.2. Data loading and pre-processing.

```
df = pd.read_csv("data/processminer-sheet-break-rare
    -event-dataset.csv")
df.head(n=5)  # visualize the data.

# Convert Categorical column to Dummy
hotencoding1 = pd.get_dummies(df['Grade&Bwt'])
hotencoding1 = hotencoding1.add_prefix('grade_')
hotencoding2 = pd.get_dummies(df['EventPress'])
hotencoding2 = hotencoding2.add_prefix('eventpress_'
    )

df = df.drop(['Grade&Bwt', 'EventPress'], axis=1)

df = pd.concat([df, hotencoding1, hotencoding2],
    axis=1)

# Rename response column name for ease of
    understanding
df = df.rename(columns={'SheetBreak': 'y'})

# Shift the response for training the model early
    prediction.
df = curve_shift(df, shift_by=-2)

# Sort by time and drop the time column.
df['DateTime'] = pd.to_datetime(df.DateTime)
df = df.sort_values(by='DateTime')
df = df.drop(['DateTime'], axis=1)
```



	y	RSashScanAvg	CT#1 BLADE PSI	P4 CT#2 BLADE PSI	Bleached GWD Flow	ShwerTemp	BlndStckFloTPD	C1 BW SPREAD CD	RS BW SPREAD CD	C1 BW SPREAD MD	...
246	0.0	1.083067	-3.961600	-1.027094	-9.204767	-0.645533	2.145302	-0.019410	-0.041277	-0.030057	...
247	0.0	1.169452	-3.909921	-1.164501	-9.627252	-0.673197	5.558846	-0.019410	-0.041277	-0.032488	...
248	0.0	0.963821	-3.959412	-1.082102	-8.352472	-0.700877	7.936959	-0.019410	-0.041277	-0.040129	...
249	0.0	0.969188	-4.129351	-1.026394	-3.299493	-0.372340	9.671456	-0.019410	-0.041277	-0.040129	...
250	0.0	0.979080	-3.979111	-1.137012	-2.321619	0.023183	11.405922	-0.011354	-0.049696	-0.040129	...
251	0.0	0.950350	-4.217456	-1.159475	-4.261438	0.618902	13.127663	-0.009339	-0.051043	-0.059966	...
252	0.0	0.987078	-4.025989	-1.210205	0.899603	0.450338	14.098854	0.000732	-0.051043	-0.059966	...
253	0.0	0.921726	-3.728572	-1.230373	-1.598718	0.227178	14.594612	0.000060	-0.051043	-0.040129	...
254	0.0	0.975947	-3.913736	-1.304682	0.561987	0.004034	14.630532	0.000732	-0.051043	-0.040129	...
255	0.0	0.997107	-3.865720	-1.133779	0.377295	-0.219126	14.666420	0.000732	-0.061114	-0.040129	...
256	0.0	1.016235	-4.058394	-1.097158	2.327307	-0.442286	14.702309	0.000732	-0.061114	-0.040129	...
257	1.0	1.005602	-3.876199	-1.074373	0.844397	-0.553050	14.738228	0.000732	-0.061114	-0.030057	...

	0	1	2	3	4	5	6	7	8	9	...
0	0.997107	-3.865720	-1.133779	0.377295	-0.219126	14.666420	0.000732	-0.061114	-0.040129	0.001791	...
1	1.016235	-4.058394	-1.097158	2.327307	-0.442286	14.702309	0.000732	-0.061114	-0.040129	0.001791	...
2	1.005602	-3.876199	-1.074373	0.844397	-0.553050	14.738228	0.000732	-0.061114	-0.030057	0.001791	...

Figure 5.8. *Testing data temporalization.*

5.4.2 Temporalizing the Data

From an LSTM modeling standpoint, a usual two-dimensional input, also referred to as planar data, does not directly provide time-windows. But the time-windows are required to extract spatio-temporal patterns.

Therefore, the planar data is *temporalized*. In temporalization, a time-window spanning observations in $t : t - lookback$ is taken at a time point t and placed at index t in a three-dimensional array.

This is visualized in Figure 5.8. An example of real data is shown in the figure. Here a time-window of predictors of size three (`lookback=3`) at time index `257` is taken out and stacked in a 3-dimensional array at the same index `257`.

No reorientation in the response `y` is made. It is only ensured that the indexes of `y` are synchronized with the time-window stack of predictors which is now a 3-dimensional array.

With temporalized input, the model has access to the current and the past predictor observations $\boldsymbol{x}_{t:(t-lookback)}$ that led to the observed response y_t. This access enables the model to learn the temporal patterns.

Data temporalization is essential to learn temporal patterns.

The `lookback` is also referred to as `timesteps` to imply the prior time-steps the model is looking at for learning the patterns.

Data is temporalized in line 3 in Listing 5.3. The shape of the temporalized data is: $(samples, timesteps, features)$. A lookback (or time-steps) of `5` is chosen. This implies the model will look at up to the past 5 observations. This equates to a time-window of 10 minutes in the sheet-break data set.

Listing 5.3. Data Temporalization.

```
1 # Temporalize the data
2 lookback = 5
3 X, y = temporalize(X=input_X, y=input_y, lookback=
      lookback)
```

5.4.3 Data Splitting

At this stage, the data is split into train, valid, and test. Fortunately, the `train_test_split()` function in `sklearn` can be used directly on higher-dimensional arrays. Irrespective of the array dimension, the function does the split along the first axis. This is done in Listing 5.4.

`sklearn.model_selection.train_test_split()` *is agnostic to the shape of the input array. It always samples w.r.t. the array's first dimension.*

Listing 5.4. Temporalized data split.

```
X_train, X_test, y_train, y_test =
    train_test_split(X, y,
                     test_size=DATA_SPLIT_PCT,
                     random_state=SEED)
X_train, X_valid, y_train, y_valid =
    train_test_split(X_train, y_train,
                      test_size=DATA_SPLIT_PCT,
                      random_state=SEED)

TIMESTEPS = X_train.shape[1]  # equal to the
    lookback
N_FEATURES = X_train.shape[2]  # the number of
    features
```

5.4.4 Scaling Temporalized Data

The 3-dimensional data is scaled using a udf, `scale()`, in Listing 5.5.

Listing 5.5. Scaling temporalized data.

```
# Fit a scaler using the training data.
scaler = StandardScaler().fit(dp.flatten(X_train))
X_train_scaled = dp.scale(X_train, scaler)
X_valid_scaled = dp.scale(X_valid, scaler)
X_test_scaled = dp.scale(X_test, scaler)
```

Besides, it is not preferred to scale the initial 2-dimensional data before temporalization because it will lead to the leakages in the time-window during the data split.

5.5 Baseline Model—A Restricted Stateless LSTM

Similar to the previous chapter, it is always advisable to begin with a baseline model. A restricted stateless LSTM network is taken as a baseline. In such a network, every LSTM layer is stateless and the final layer has a *restricted* output, i.e.,
`LSTM(..., stateful=False, return_sequences=False)`.

In the following, the baseline model will be constructed step-by-step.

5.5.1 Input layer

As mentioned before, the input layer in LSTM expects 3-dimensional inputs. The input shape should be: (*batch size*, *timesteps*, $features$).

A stateless LSTM does not require to explicitly specify the batch size (a stateful LSTM does require the batch size as mentioned in Appendix F). Therefore, the input shape is defined as follows in Listing 5.6.

Listing 5.6. LSTM input layer.

```
model = Sequential()
model.add(Input(shape=(TIMESTEPS, N_FEATURES),
                name='input'))
```

The input shape can also be provided as an argument to the first LSTM layer defined next. However, similar to the previous chapter this is explicitly defined for clarity.

5.5.2 LSTM layer

As a rule-of-thumb, two hidden LSTM layers are stacked in the baseline model. The `recurrent_activation` argument is left as its default `sigmoid` while the output `activation` is set as `relu` (refer to § 5.2.5). The `relu` activation came into existence after LSTMs. Therefore, they are not in the legacy LSTM definitions but can be used on the output.

Listing 5.7. LSTM layers.

```
model.add(
    LSTM(units=16,
         activation='relu',
         return_sequences=True,
         name='lstm_layer_1'))
model.add(
    LSTM(units=8,
         activation='relu',
         return_sequences=False,
         name='lstm_layer_2'))
```

The first LSTM layer has `return_sequences` set as `True`. This layer, therefore, yields the hidden outputs for every time-step. Consequently, the first layer output is (*batch size*, *timesteps*, 16), where 16 is the layer size.

Since this is a restricted LSTM network, the last LSTM layer is set with `return_sequences` as `False`. Therefore, it returns the output from only the last time-step. Thus, the layer output is of shape: (*batch size*, 8), where 8 is the layer size.

5.5.3 Output layer

The output layer should be a `Dense` layer in an LSTM network and most other networks, in general.

Why? The output layer should be dense because it performs an affine transformation on the output of the ultimate hidden layer. The purpose of complex hidden layers, such as LSTM and Convolutional, is to extract predictive features. But these abstract features do not necessarily translate to the model output y. A dense layer's affine transformation puts together these features and translates them to the output y.

The output layer should be a `Dense` *layer for most deep learning networks.*

We add a `Dense` layer of size 1. As also mentioned in §4.4.4, the size is based on the number of responses. For a binary classifier, like here, the size is one with `sigmoid` activation. If we have a multi-class classifier, the size should be the number of labels with `softmax` activation.

Listing 5.8. LSTM network output layer.

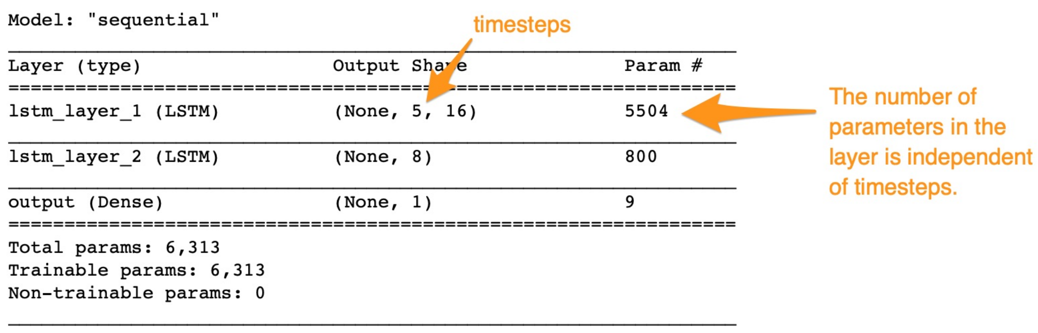
```
model.add(Dense(units=1,
                activation='sigmoid',
                name='output'))
```

5.5.4 Model Summary

At this stage, the structure of the baseline LSTM model is ready. Before moving forward, the model structure should be glanced at using the `model.summary()` function.

Listing 5.9. LSTM baseline model summary.

```
model.summary()
```


timesteps
The number of parameters in the layer is independent of timesteps.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_layer_1 (LSTM)          (None, 5, 16)             5504
_________________________________________________________________
lstm_layer_2 (LSTM)          (None, 8)                 800
_________________________________________________________________
output (Dense)               (None, 1)                 9
=================================================================
Total params: 6,313
Trainable params: 6,313
Non-trainable params: 0
_________________________________________________________________
```

Figure 5.9. *LSTM baseline model summary.*

```

# Number of parameters = 4l(p + l + 1),
# l = layer size, p = number of features.
4*16*(n_features + 16 + 1) # Parameters in
    lstm_layer_1
# 5504
```

The summary in Figure 5.9 shows the number of parameters in each layer. For self-learning, it can be computed by hand using Eq. 5.2. For example, the number of parameters in the first LSTM layer is $4 \times 16(n\_features + 16 + 1) = 5504$, where 16 is the layer size.

5.5.5 Compile and Fit

The model `compile()` and `fit()` arguments are explained in § 4.4.6 and 4.4.7. They are directly applied here.

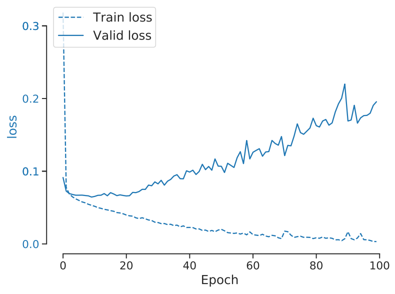
Listing 5.10. LSTM baseline model compile and fit.

```
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=[
                  'accuracy',
                  tf.keras.metrics.Recall(),
                  pm.F1Score(),
                  pm.FalsePositiveRate()
              ])
```

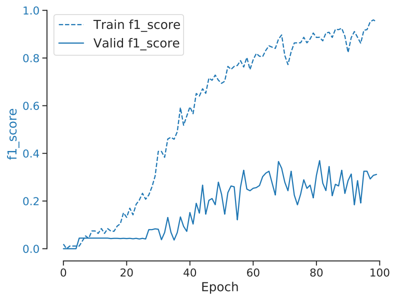
```

history = model.fit(x=X_train_scaled,
                    y=y_train,
                    batch_size=128,
                    epochs=100,
                    validation_data=(X_valid_scaled,
                                     y_valid),
                    verbose=0).history
```

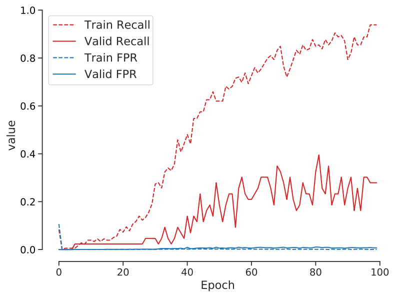
The results are shown in Figure 5.10a-5.10c.

The accuracy metrics for the baseline LSTM model is already better than the best achieved with MLP. This was expected from an LSTM model because it is capable of drawing the temporal patterns. A few model improvements are attempted next to further improve accuracy.



(a) *Loss.*





(b) *F1-score.* (c) *Recall and FPR.*

Figure 5.10. *LSTM baseline model results.*

5.6 Model Improvements

5.6.1 Unrestricted LSTM Network

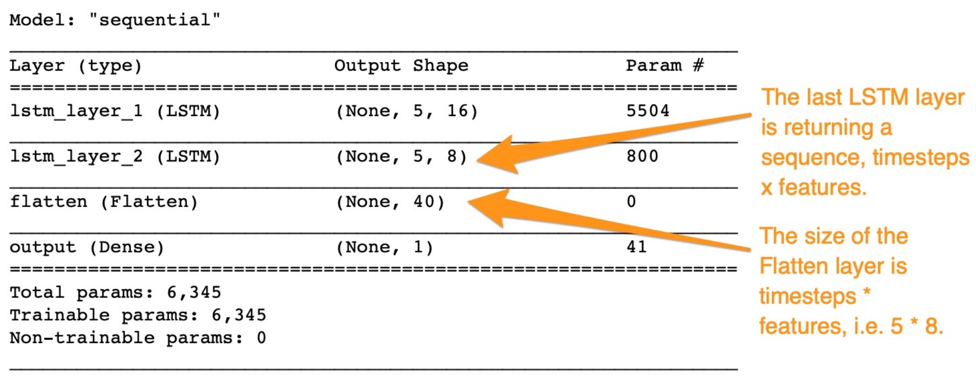
§ 5.3.3 discussed two choices in LSTM networks: return sequences vs return last output at the final LSTM layer. The latter is typically the default choice and used in the baseline restricted model above. However, the former can likely improve the model.

The former choice enables the ultimate LSTM layer to emit a sequence of hidden outputs. Since the last LSTM layer is not restricted to emitting only the final hidden output, this network is called an *unrestricted* LSTM network. A potential benefit of this network is the presence of more intermediate features.

Based on this hypothesis, an unrestricted network is constructed in Listing 5.11.

Listing 5.11. Unrestricted LSTM model.

```
model = Sequential()
model.add(Input(shape=(TIMESTEPS, N_FEATURES),
                name='input'))
model.add(
    LSTM(units=16,
         activation='relu',
         return_sequences=True,
         name='lstm_layer_1'))
model.add(
    LSTM(units=8,
         activation='relu',
         return_sequences=True,
         name='lstm_layer_2'))
model.add(Flatten())
model.add(Dense(units=1,
                activation='sigmoid',
                name='output'))

model.summary()

model.compile(optimizer='adam',
              loss='binary_crossentropy',
```



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_layer_1 (LSTM)          (None, 5, 16)             5504
_________________________________________________________________
lstm_layer_2 (LSTM)          (None, 5, 8)              800
_________________________________________________________________
flatten (Flatten)            (None, 40)                0
_________________________________________________________________
output (Dense)               (None, 1)                 41
=================================================================
Total params: 6,345
Trainable params: 6,345
Non-trainable params: 0
_________________________________________________________________
```

The last LSTM layer is returning a sequence, timesteps x features.

The size of the Flatten layer is timesteps * features, i.e. 5 * 8.

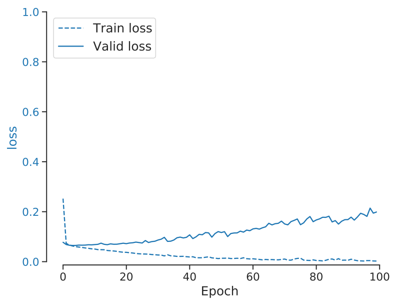
Figure 5.11. *LSTM full model summary.*

```
                      metrics=[
                          'accuracy',
                          tf.keras.metrics.Recall(),
                          pm.F1Score(),
                          pm.FalsePositiveRate()
                      ])
history = model.fit(x=X_train_scaled,
                    y=y_train,
                    batch_size=128,
                    epochs=100,
                    validation_data=(X_valid_scaled,
                                     y_valid),
                    verbose=0).history
```

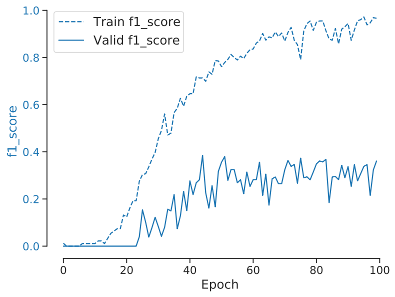
In the construction, the ultimate LSTM layer `lstm_layer_2` is set with `return_sequences=True`. The model summary in Figure 5.11 is showing its effect.

Unlike the baseline model, `lstm_layer_2` is now returning 3-dimensional outputs of shape (*batchsize*, 5, 8), where 5 and 8 are the time-steps and the layer size, respectively. This 3D tensor is flattened before passing on to the output `Dense` layer (also depicted in Figure 5.7b2).

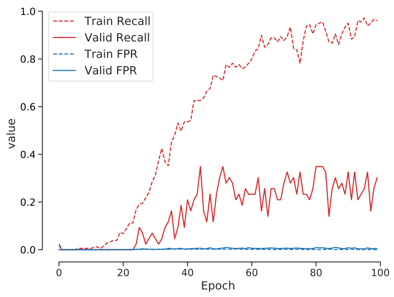
In this network, the parameters in the output `Dense` layer increases from 9 (= 8 weights + 1 bias) in the baseline network (Figure 5.9) to 41 (= 40 weights + 1 bias).



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure 5.12. *Unrestricted LSTM model results.*

The results are shown in Figure 5.12a-5.12c. The accuracy improved with the unrestricted model. For the given problem, the temporal intermediate features seem to be predictive but this cannot be generalized.

5.6.2 Dropout and Recurrent Dropout

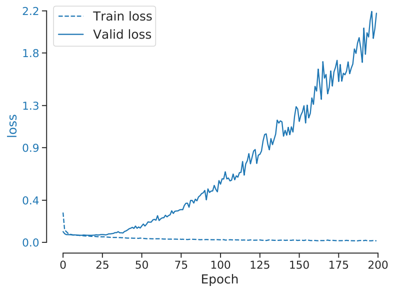
Dropout is a common technique used for deep learning network improvement. However, it does not always work with RNNs including LSTMs. Gal and Ghahramani 2016 made an extreme claim stating,

> "Dropout is a popular regularization technique with deep networks where network units are randomly masked during training (dropped). But the technique has never been applied successfully to RNNs."

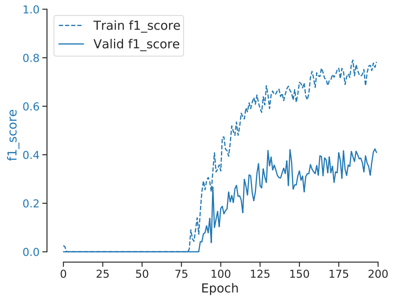
True to the claim, a regular dropout does not always work with LSTMs. However, there is another type of dropout available in RNNs called *recurrent dropout*. In this technique, a fraction of inputs to the recurrent states is dropped. Both these dropouts are applied together in Listing 5.12 and are found to improve the model.

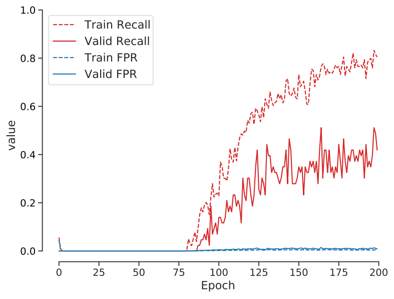
Listing 5.12. Unrestricted LSTM with regular and recurrent dropout.

```
model = Sequential()
model.add(Input(shape=(TIMESTEPS, N_FEATURES),
                name='input'))
model.add(
    LSTM(units=16,
         activation='relu',
         return_sequences=True,
         recurrent_dropout=0.5,
         name='lstm_layer_1'))
model.add(Dropout(0.5))
model.add(
    LSTM(units=8,
         activation='relu',
         return_sequences=True,
         recurrent_dropout=0.5,
         name='lstm_layer_2'))
model.add(Flatten())
model.add(Dropout(0.5))
model.add(Dense(units=1,
                activation='sigmoid',
                name='output'))

model.summary()

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=[
                  'accuracy',
                  tf.keras.metrics.Recall(),
                  pm.F1Score(),
                  pm.FalsePositiveRate()
              ])
history = model.fit(x=X_train_scaled,
                    y=y_train,
                    batch_size=128,
                    epochs=200,
                    validation_data=(X_valid_scaled,
                                     y_valid),
                    verbose=0).history
```



(a) *Loss.*





(b) *F1-score.* (c) *Recall and FPR.*

Figure 5.13. *Unrestricted LSTM with dropout and recurrent dropout results.*

The results are shown in Figure 5.13a-5.13c. The accuracy was further improved. However, increasing validation loss is still observed. Besides, this model has trained over 200 epochs (as opposed to 100 in previous models) for the metrics to stabilize.

`Recurrent_dropout` is more applicable to LSTM networks than the regular dropout.

5.6.3 Go Backwards

Researchers at Google published a paper by Sutskever, Vinyals, and Le 2014. This paper is now one of the most popular papers on LSTMs. In this paper, they had an interesting finding. They quote,

> "...we found that reversing the order of the words in all source sentences (but not target sentences) improved the LSTM's performance markedly, because doing so introduced many short-term dependencies between the source and the target sentence which made the optimization problem easier."

Such a model can be made by setting `go_backwards=True` in the **first** LSTM layer. This processes the input sequence backward.

Note that only the first LSTM layer should be made backward because only the inputs need to be processed backward. Subsequent LSTM layers work on arbitrary features of the model. Making them backward is meaningless in most cases.

Listing 5.13. LSTM network with input sequences processed backward.

```
model = Sequential()
model.add(Input(shape=(TIMESTEPS, N_FEATURES),
                name='input'))
model.add(
    LSTM(units=16,
         activation='relu',
         return_sequences=True,
         go_backwards=True,
         name='lstm_layer_1'))
model.add(
    LSTM(units=8,
         activation='relu',
         return_sequences=True,
         name='lstm_layer_2'))
model.add(Flatten())
model.add(Dense(units=1,
                activation='sigmoid',
                name='output'))

```

```
model.summary()

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=[
                  'accuracy',
                  tf.keras.metrics.Recall(),
                  pm.F1Score(),
                  pm.FalsePositiveRate()
              ])
history = model.fit(x=X_train_scaled,
                    y=y_train,
                    batch_size=128,
                    epochs=100,
                    validation_data=(X_valid_scaled,
                                     y_valid),
                    verbose=0).history
```

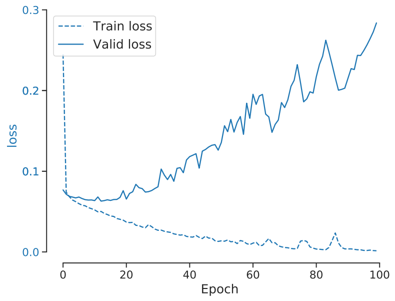
Backward sequence processing worked quite well for sequence-to-sequence model. Perhaps because the output was also a sequence and the initial elements in the input sequence were dependent on the last elements of the output sequence. Therefore, reversing the input sequence brought the related input-output elements closer.

However, as shown in the results in Figure 5.14a-5.14c, this approach fared close to or less than the baseline in our problem.

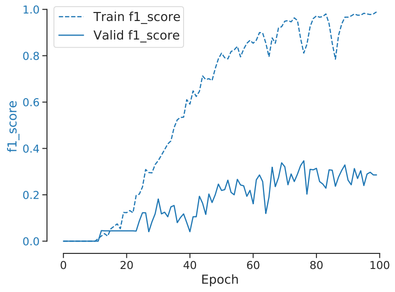
`Backward` LSTM models are effective on sequences in which the early segments have a larger influence on the future. For example, in language translations because the beginning of a sentence usually has a major influence on how the sentence ends.

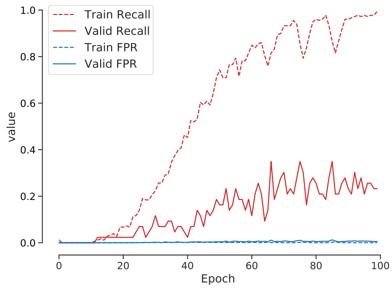
5.6.4 Bi-directional

A regular LSTM, or any RNN, learn the temporal patterns in a forward direction—going from past to the future. Meaning, at any time-step the cell state learns only from the past. It does not have visibility of the



(a) *Loss.*





(b) *F1-score.* (c) *Recall and FPR.*

Figure 5.14. *LSTM with input processed backward results.*

future. This concept is clearer in Figure 5.2a. As shown in the figure, the truck of memories (cell state) moves from left-to-right.

Schuster and Paliwal 1997 made a significant contribution by proposing a bi-directional RNN. A bi-directional RNN adds a mirror RNN layer to the original RNN layer. The input sequence is passed as is to the original layer and in reverse to the mirror.

This enables the cell states to learn from all the input information, i.e., both the past and the future. This is illustrated in Figure 5.15. Similar to a regular LSTM layer, the information from the past flows into the cell state from left-to-right in the top lane. Additionally, the information from the future flows back to the cell states right-to-left in the bottom lane.

This ability to have collective information from the past and the future make a bi-directional LSTM layer more powerful. A popular

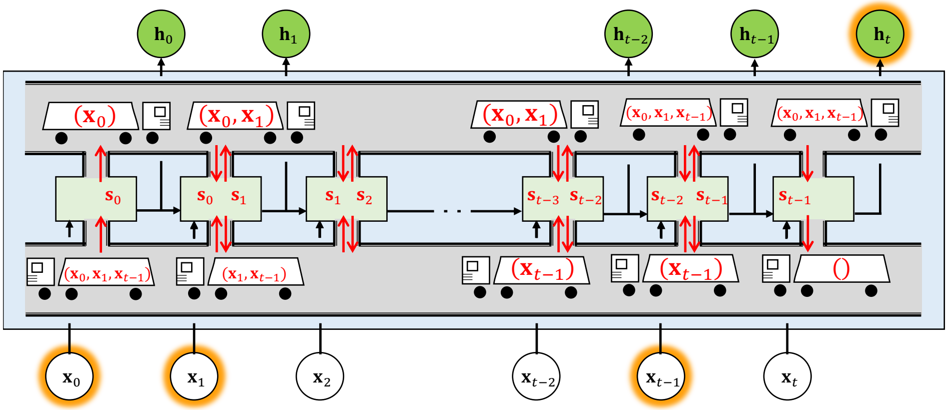


Figure 5.15. *LSTM bi-directional cell state mechanism.*

work by Graves and Schmidhuber 2005 on bi-directional LSTMs show that they significantly outperformed the traditional LSTMs in speech recognition. Graves and Schmidhuber 2005 also quote an important requirement for using a bi-directional LSTM,

> ...for temporal problems like speech recognition, relying on knowledge of the future seems at first sight to violate causality—at least if the task is online.
>
> ...However, human listeners do exactly that. Sounds, words, and even whole sentences that at first mean nothing are found to make sense in the light of the future context.

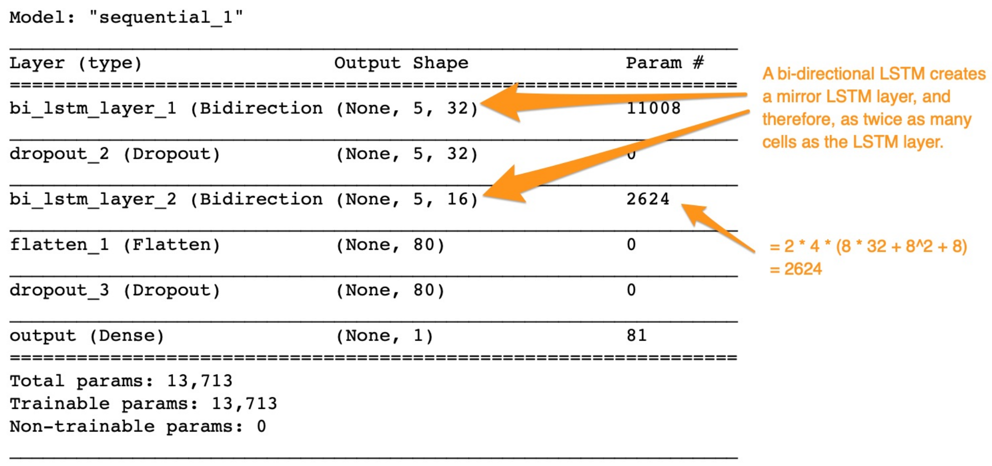
Therefore, we must recognize whether the problem is truly online, i.e., requiring an output for every new input, or semi-online when the output is needed at end of some input segment. Bi-directional LSTM does not apply to the former but could significantly improve performance in the latter.

The sheet-break problem at hand can be treated as semi-online. A prediction can be made after a window of sensor observations are made. In fact, from an LSTM standpoint, most problems are either offline (e.g., text documents) or semi-online (e.g., time series). A bi-directional LSTM network is, therefore, built in Listing 5.14.

Listing 5.14. Bi-directional LSTM network.

```python
model = Sequential()
model.add(Input(shape=(TIMESTEPS, N_FEATURES),
                name='input'))
model.add(
    Bidirectional(
        LSTM(units=16,
             activation='relu',
             return_sequences=True),
        name='bi_lstm_layer_1'))
model.add(Dropout(0.5))
model.add(
    Bidirectional(
        LSTM(units=8,
             activation='relu',
             return_sequences=True),
        name='bi_lstm_layer_2'))
model.add(Flatten())
model.add(Dense(units=1,
                activation='sigmoid',
                name='output'))

model.summary()

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=[
                  'accuracy',
                  tf.keras.metrics.Recall(),
                  pm.F1Score(),
                  pm.FalsePositiveRate()
              ])

history = model.fit(x=X_train_scaled,
                    y=y_train,
                    batch_size=128,
                    epochs=100,
                    validation_data=(X_valid_scaled,
                                     y_valid),
                    verbose=0).history
```

In TensorFlow, a bi-directional LSTM layer is made by wrapping



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
bi_lstm_layer_1 (Bidirection (None, 5, 32)             11008
_________________________________________________________________
dropout_2 (Dropout)          (None, 5, 32)             0
_________________________________________________________________
bi_lstm_layer_2 (Bidirection (None, 5, 16)             2624
_________________________________________________________________
flatten_1 (Flatten)          (None, 80)                0
_________________________________________________________________
dropout_3 (Dropout)          (None, 80)                0
_________________________________________________________________
output (Dense)               (None, 1)                 81
=================================================================
Total params: 13,713
Trainable params: 13,713
Non-trainable params: 0
_________________________________________________________________
```

A bi-directional LSTM creates a mirror LSTM layer, and therefore, as twice as many cells as the LSTM layer.

= 2 * 4 * (8 * 32 + 8^2 + 8)
= 2624

Figure 5.16. *LSTM bi-directional network summary.*

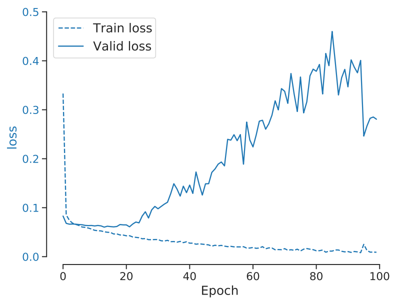
an `LSTM` layer within a `Bidirectional` layer. The `Bidirectional` layer creates a mirror layer for any RNN layer passed as an argument to it (an LSTM here).

As shown in Figure 5.16, this results in twice the number of parameters for a bi-directional layer compared to the ordinary LSTM layer. Expressed as

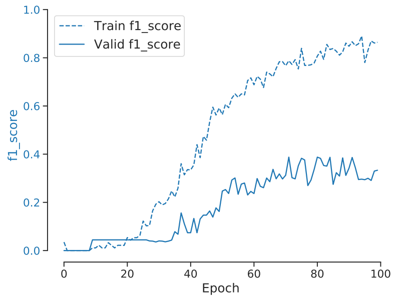
$$n\_parameters = 2 \times 4l(p + l + 1) \tag{5.6}$$

where l and p are the size of the layer and number of features, respectively. Bi-directional networks, therefore, require a sufficient amount of training data. Absence of which may render it less effective.

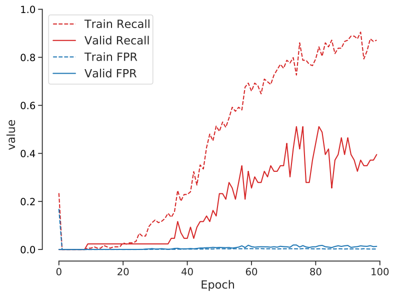
The results of the bi-directional network are shown in Figure 5.17a-5.17c. The accuracy is found to be higher than the prior models. This could be attributed to the bi-directional LSTM's ability to capture temporal patterns both retrospectively (backward) and prospectively (forward).



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure 5.17. *Bi-directional LSTM results.*

Bi-directional LSTMs learn temporal patterns both retrospectively and prospectively.

5.6.5 Longer Lookback/Timesteps

LSTMs are known to learn long-term dependencies. In the previous models, the lookback period was set as 5. A hypothesis could be that a longer lookback could capture more predictive patterns and improve accuracy. This is tried in this section.

First, the data is re-prepared with a longer lookback of 20 in Listing 5.15 by setting the `lookback = 20`.

Listing 5.15. Data re-preparation with longer lookback.

```python
lookback = 20  # Equivalent to 40 min of past data.
# Temporalize the data
X, y = dp.temporalize(X=input_X,
                      y=input_y,
                      lookback=lookback)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=DATA_SPLIT_PCT, random_state=
       SEED)
X_train, X_valid, y_train, y_valid =
   train_test_split(
    X_train, y_train, test_size=DATA_SPLIT_PCT,
       random_state=SEED)

X_train = X_train.reshape(X_train.shape[0],
                          lookback,
                          n_features)
X_valid = X_valid.reshape(X_valid.shape[0],
                          lookback,
                          n_features)
X_test = X_test.reshape(X_test.shape[0],
                        lookback,
                        n_features)

# Initialize a scaler using the training data.
scaler = StandardScaler().fit(dp.flatten(X_train))

X_train_scaled = dp.scale(X_train, scaler)
X_valid_scaled = dp.scale(X_valid, scaler)
X_test_scaled = dp.scale(X_test, scaler)
```

The unrestricted-LSTM network is then trained with the longer lookback data in Listing 5.16.

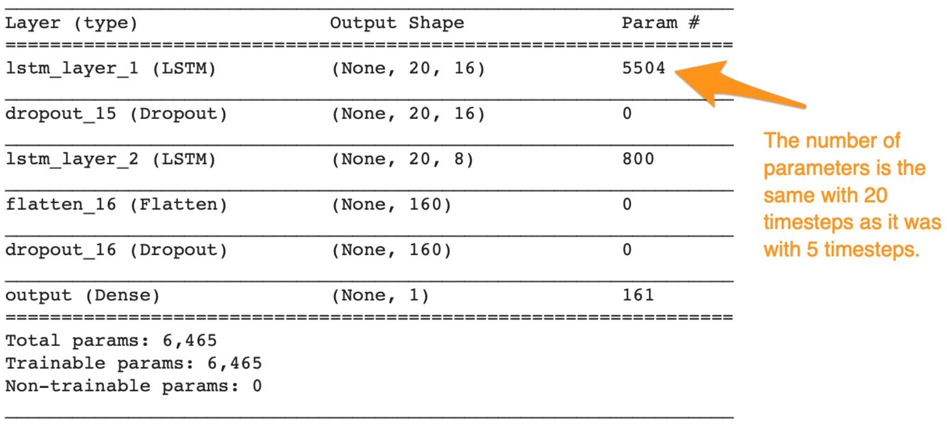
Listing 5.16. LSTM model with longer lookback.

```python
timesteps = X_train_scaled.shape[1]

model = Sequential()
model.add(Input(shape=(timesteps, N_FEATURES),
                name='input'))
```

```
model.add(
    LSTM(units=16,
         activation='relu',
         return_sequences=True,
         recurrent_dropout=0.5,
         name='lstm_layer_1'))
model.add(Dropout(0.5))
model.add(
    LSTM(units=8,
         activation='relu',
         return_sequences=True,
         recurrent_dropout=0.5,
         name='lstm_layer_2'))
model.add(Flatten())
model.add(Dropout(0.5))
model.add(Dense(units=1,
                activation='sigmoid',
                name='output'))

model.summary()

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=[
                  'accuracy',
                  tf.keras.metrics.Recall(),
                  pm.F1Score(),
                  pm.FalsePositiveRate()
              ])
history = model.fit(x=X_train_scaled,
                    y=y_train,
                    batch_size=128,
                    epochs=200,
                    validation_data=(X_valid_scaled,
                                     y_valid),
                    verbose=0).history
```

It was mentioned in § 5.2.6 that the number of parameters in an LSTM layer does not increase with the lookback. That is now evident in the model summary in Figure 5.18.

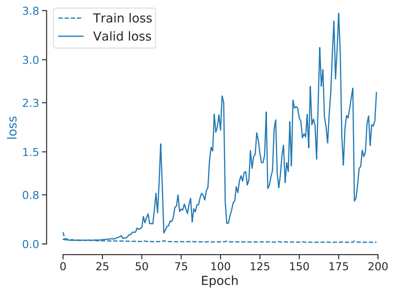
The results are in Figure 5.19a-5.19c. The performance deteriorated



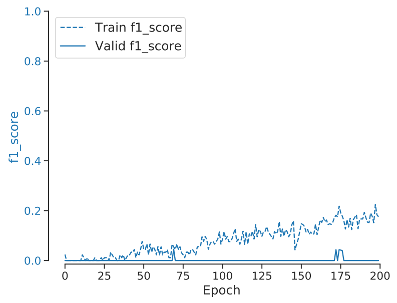
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_layer_1 (LSTM)          (None, 20, 16)            5504
_________________________________________________________________
dropout_15 (Dropout)         (None, 20, 16)            0
_________________________________________________________________
lstm_layer_2 (LSTM)          (None, 20, 8)             800
_________________________________________________________________
flatten_16 (Flatten)         (None, 160)               0
_________________________________________________________________
dropout_16 (Dropout)         (None, 160)               0
_________________________________________________________________
output (Dense)               (None, 1)                 161
=================================================================
Total params: 6,465
Trainable params: 6,465
Non-trainable params: 0
_________________________________________________________________
```

The number of parameters is the same with 20 timesteps as it was with 5 timesteps.

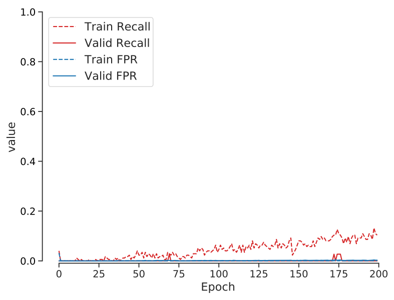
Figure 5.18. *LSTM with longer lookback network summary.*



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure 5.19. *Unrestricted-LSTM model with longer lookback results.*

with a longer lookback. These results confirm the statement by Jozefowicz, Zaremba, and Sutskever 2015. They stated that if the lookback period is long, the cell states fuse information across a wide window. Due to this, the extracted information gets smeared. As a result, protracting the lookback does not always work with LSTMs.

Increasing the lookback may smear the features learned by LSTMs and, therefore, cause poorer performance.

5.7 History of LSTMs

Until the early 1990s, RNNs were learned using real-time recurrent learning (RTRL, Robinson and Fallside 1987) and back-propagation through time (BPTT, R. J. Williams and Zipser 1995). But they had critical limitations: vanishing and exploding gradient.

These limitations were due to the error propagation mechanism in a recurrent network. In the early methods, the errors were progressively multiplied as it traveled back in time. Due to this, the resultant error could exponentially increase or decrease. Consequently, the backpropagated error can quickly either vanish or explode.

The gradient explosion was empirically addressed by capping any high values. But the vanishing gradient issue was still an unsolved problem. It was casting doubt on whether RNNs can indeed exhibit significant practical advantages.

Then, long short term memory (LSTM) was developed by Hochreiter and Schmidhuber 1997. They stated,

> "...as the time lag increases, (1) stored information must be protected against perturbation for longer and longer periods, and—especially in advanced stages of learning—(2) more and more already correct outputs also require protection against perturbation."
>
> – p7 in Hochreiter and Schmidhuber 1997.

Table 5.2. History of LSTM.

1997	Hochreiter & Schmidhuber • Memory state. .
1999	Gers, Schmidhuber, & Cummins • Memory state, and • Forget gate.
2002	Gers, Schraudolph & Schmidhuber • Memory state, • Forget gate, and • Peeping hole.
2007-2008	Graves, Fernández, & Schmidhuber, Graves & Schmidhuber • Memory state, • Forget gate, • Peeping hole, and • Multi-dimensional.
2012-2013	Graves 2012, Graves 2013 • Memory state, • Forget gate, • Peeping hole, • Multi-dimensional, and • refined formulation
2015	Jozefowicz, Zaremba, & Sutskever • Memory state, • Forget gate, • Multi-dimensional, • refined formulation, and • *implemented in TensorFlow.*
2017	Baytas, Xiao, Zhang, et. al. • Handle irregular time-intervals (time aware), ⋮

They recognized that the vanishment problem is due to error propagation. If only the error could be left untouched during the time travel, the vanishment and explosion would not occur.

Hochreiter and Schmidhuber 1997 implemented this idea by enforcing *constant* error flow through what they called "constant error carousals (CECs)." Their CEC implementation was done with the help of adding a recurring cell state. An abstract and simplified representation of their formulation is shown below.

$$\begin{aligned}
i_t &= f(\boldsymbol{w}_i \boldsymbol{y}_{t-1} + b_i)\\
o_t &= f(\boldsymbol{w}_o \boldsymbol{y}_{t-1} + b_o)\\
\tilde{c}_t &= g(\boldsymbol{w}_c \boldsymbol{y}_{t-1} + b_c)\\
c_t &= c_{t-1} + i_t \tilde{c}_t\\
y_t &= o_t h(s_t)
\end{aligned}$$

The first three expressions are called *input* gate, *output* gate, and *state* gate, respectively. The last two expressions are the *cell state* and *cell output*, respectively. In this section, it is okay to ignore the equation details. Here the focus is on the formulation and their key differentiating elements highlighted in red.

The key element in Hochreiter and Schmidhuber's formulation above is the cell state c_t. The cell state acts as long-term memory.

It has an additive expression instead of multiplicative. The expression can also be seen as computing the delta, $\Delta c_t = i_t \tilde{c}_t$, at each time-step and adding it to the cell state c_t. While it is true that additive c_t does not necessarily result in a more powerful model, the gradients of such RNNs are better behaved as they do not cause vanishment (explained in § 5.2.8).

But this approach has another issue. The additive cell state expression does not forget a past. It will keep the memories from all the time-steps in the past. Consequently, Hochreiter and Schmidhuber's LSTM will not work if the memories have limited relevance in time.

Hochreiter and Schmidhuber worked around this by performing an *apriori* segmentation of time series into subsequences such that all time-steps in the subsequence are relevant. But such an apriori processing is a methodological limitation.

Gers, Schmidhuber, and Cummins 1999 addressed this by bringing *forget* gates into the formulation. They stated, "any training procedure for RNNs which is powerful enough to span long time lags must also address the issue of **forgetting** in short-term memory."

In Gers, Schmidhuber, and Cummins 1999, it is emphasized that the cell state c_t tends to grow linearly during a time series traversal. If a continuous time series stream is presented, the cell states may grow in an unbounded fashion. This causes saturation of the output squashing function $h(c_t)$ at the end[2].

Gers et. al. countered this with *adaptive* forget gates in Gers, Schmidhuber, and Cummins 1999. These gates learn to reset the cell states (the memory) once their contents are out-of-date and, hence, useless. This is done by a multiplicative forget gate activation f_t. f_t can also be seen as a weight on the prior memory shown in their high-level formulation below.

$$\begin{aligned}
i_t &= f(\boldsymbol{w}_i \boldsymbol{y}_{t-1} + b_i) \\
o_t &= f(\boldsymbol{w}_o \boldsymbol{y}_{t-1} + b_o) \\
f_t &= f(\boldsymbol{w}_f \boldsymbol{y}_{t-1} + b_f) \\
\tilde{c}_t &= g(\boldsymbol{w}_c \boldsymbol{y}_{t-1} + b_c) \\
c_t &= f_t c_{t-1} + i_t \tilde{c}_t \\
y_t &= o_t h(c_t)
\end{aligned}$$

After forget gates, Gers and Schmidhuber were back on this. They, along with Schrandolph, devised what they called as a *peeping hole* in Gers, Schraudolph, and Schmidhuber 2002. The name may sound creepy but the approach was scientific.

In the LSTMs, thus far, each gate receives connections from the input and the output of the cells. But there is no direct connection between the gates and the cell state (memory) they are supposed to control. The resulting lack of essential information (cell state) may harm a network's

[2]**Saturation** meaning: At a saturation point of a function, any change in its input does not change the output. That is, $y = f(x) = f(x + \Delta x)$ if $x, x + \Delta x \in$ saturation region.

performance (Gers, Schraudolph, and Schmidhuber 2002).

As a remedy, Gers et. al. added weighted "peephole" connections from the cell states to the input, output, and forget gates shown below.

$$\begin{aligned}
i_t &= f(\boldsymbol{w}_i^{(y)}\boldsymbol{y}_{t-1} + \boldsymbol{w}_i^{(c)}\boldsymbol{c}_{t-1} + b_i) \\
o_t &= f(\boldsymbol{w}_o^{(y)}\boldsymbol{y}_{t-1} + \boldsymbol{w}_o^{(c)}\boldsymbol{c}_{t-1} + b_o) \\
f_t &= f(\boldsymbol{w}_f^{(y)}\boldsymbol{y}_{t-1} + \boldsymbol{w}_f^{(c)}\boldsymbol{c}_{t-1} + b_f) \\
\tilde{c}_t &= g(\boldsymbol{w}_c^{(y)}\boldsymbol{y}_{t-1} + b_c) \\
c_t &= f_t c_{t-1} + i_t \tilde{c}_t \\
y_t &= o_t h(c_t)
\end{aligned}$$

The next wave of LSTM progress can be attributed to Alex Graves for his work in 2005–2015. He along with Santiago Fernandex and Jurger Schmidhuber developed the foundation of multi-dimensional RNNs in Graves, Fernández, and Schmidhuber 2007.

To avoid confusion for statisticians, the multi-dimension here refers to the number of input sample axes and not the features. For example, a time series has one axes while an image has two axes.

Until this work, LSTM/RNNs were applicable only to single-axes sequence problems, such as speech recognition. Applying RNNs to data with more than one spatio-temporal axes was not straightforward. Graves et. al. (2007) laid down the formulation for multi-dimension/axes sequences.

The multi-dimensional extension was a significant leap that made RNNs, in general, and LSTMs, specifically, applicable to multivariate time series, video processing, and other areas.

This work was carried forward by Graves and Schmidhuber in Graves and Schmidhuber 2009 that won the ICDAR handwriting competition in 2009.

Then in 2012-2013 Graves laid a refined LSTM version in Graves 2012; Graves 2013 that we are familiar with today. His formulation for multi-dimensional sequences is shown below.

$$
\begin{aligned}
i_t &= f(W_i^{(x)}\boldsymbol{x}_t + W_i^{(h)}\boldsymbol{h}_{t-1} + W_i^{(c)}\boldsymbol{c}_{t-1} + b_i)\\
o_t &= f(W_o^{(x)}\boldsymbol{x}_t + W_o^{(h)}\boldsymbol{h}_{t-1} + W_o^{(c)}\boldsymbol{c}_{t-1} + b_o)\\
f_t &= f(W_f^{(x)}\boldsymbol{x}_t + W_f^{(h)}\boldsymbol{h}_{t-1} + W_f^{(c)}\boldsymbol{c}_{t-1} + b_f)\\
\tilde{c}_t &= g(W_c^{(x)}\boldsymbol{x}_t + W_c^{(h)}\boldsymbol{h}_{t-1} + b_c)\\
c_t &= f_t c_{t-1} + i_t \tilde{c}_t\\
h_t &= o_t g(c_t)
\end{aligned}
$$

In this formulation, Graves included the features $\boldsymbol{x}$ present along the axes of a multi-dimensional sample. This refined version was also a simplification of the previous versions in Hochreiter and Schmidhuber 1997, and Gers, Schmidhuber, and Cummins 1999; Gers, Schraudolph, and Schmidhuber 2002 (might not be apparent here because the formulations above are simplified representations).

While the previous works had a complex memory block concept with byzantine architecture (not shown here for clarity), Graves new formulation had a simpler memory cell.

Jozefowicz and Sutseker from Google, and Zaremba from FaceBook took forward Graves formulation that ultimately led to the current LSTM implementation in TensorFlow. They explored the LSTM variants in Jozefowicz, Zaremba, and Sutskever 2015 and recommended the formulation by Graves 2012 but without the peephole.

$$
\begin{aligned}
i_t &= f(W_i^{(x)}\boldsymbol{x}_t + W_i^{(h)}\boldsymbol{h}_{t-1} + b_i)\\
o_t &= f(W_o^{(x)}\boldsymbol{x}_t + W_o^{(h)}\boldsymbol{h}_{t-1} + b_o)\\
f_t &= f(W_f^{(x)}\boldsymbol{x}_t + W_f^{(h)}\boldsymbol{h}_{t-1} + b_f)\\
\tilde{c}_t &= g(W_c^{(x)}\boldsymbol{x}_t + W_c^{(h)}\boldsymbol{h}_{t-1} + b_c)\\
c_t &= f_t c_{t-1} + i_t \tilde{c}_t\\
h_t &= o_t g(c_t)
\end{aligned}
$$

In Jozefowicz et. al. (2015) and, consequently, in TensorFlow the LSTM's hidden state is a tuple $(\boldsymbol{h}_t, \boldsymbol{c}_t)$. The cell state, $\boldsymbol{c}_t$, is called a "slow" state that addresses the vanishing gradient problem, and $\boldsymbol{h}_t$ is called a "fast" state that allows the LSTM to make complex decision over short periods of time.

The developments in LSTMs and RNNs have continued. Time aware LSTM by Baytas et al. 2017 was proposed to handle irregular time intervals between the time-steps in a sequence. A novel memory cell called Legendre memory unit (LMU) was developed recently for RNNs by Voelker, Kajić, and Eliasmith 2019. Moreover, a time-segment LSTM and temporal inception by Ma et al. 2019 show interesting applications.

In sum, LSTM is an *enhanced* RNN. LSTM can learn long-term memories and also dispose of them when they become irrelevant. This is achieved due to the advancements and refinements over the years.

5.8 Summary

LSTM models showed to work better than MLPs. This was expected because they can learn temporal patterns. The baseline restricted LSTM model beat the best MLP model in the previous chapter. The unrestricted LSTM proved to perform even better. Adding a recurrent dropout for regularization further improved and stabilized the model.

Inspired from other works on sequence modeling, backward and bi-directional LSTM models were developed. The backward model performed below the baseline. They work better for sequence-to-sequence problems like language translations. However, the bi-directional model outperformed the others. This could be attributed to a bi-directional LSTM's ability to capture temporal patterns both retrospectively and prospectively.

Lastly, owing to the expectation from LSTMs to learn even longer-term patterns a wider time-window of inputs are used. This is done by re-preparing the data by increasing the lookback from 5 to 20. However, contrary to the expectation the performance degraded. Primarily due to LSTM cell state's limitation in fusing temporal patterns from wide time-windows. Stateful LSTMs is an alternative to learning exhaustively long-term patterns. Their implementation is shown in Appendix F as they are not directly applicable to the non-stationary time series process here.

Besides, the LSTM models constructed here faced the issue of increasing validation loss. This is further touched upon in the exercises. Finally, the chapter is concluded with a few rules-of-thumb.

5.9 Rules-of-thumb

- The thumb-rules for the number of layers, number of nodes and activation functions for the intermediate and output layers are the same as that for MLPs in § 4.10.
- **Data Processing**. The initial data processing, e.g., converting the data to numeric is the same as in MLP thumb-rules in § 4.10. Additionally,
  - **Temporalize**. The data temporalization into 3-dimensional arrays of shape, (*batch size*, *timesteps*, *features*), is necessary.
  - **Split**. Randomly split the temporalized data using `train_test_split` from `sklearn.model_selection`. The data split should be done after temporalization to avoid observations to leak between train, valid and test sets. Besides, as discussed in § 4.3.2, the temporalized data windows are self-contained. Therefore, random sampling of the time series is applicable.
  - **Scale**. Scaling the temporalized 3D data is facilitated with custom-defined functions in § 5.4.4. Fit a `StandardScaler` on the train set and transform the valid and test sets.
- **Restricted vs unrestricted LSTM**. It is preferable to work with unrestricted LSTM. It will typically provide better accuracy. This is done as follows,
  - **Return sequences**. Set the argument `return_sequences=True` in all the LSTM layers, including the last in the stack.
  - **Flatten**. If the layer next to the last LSTM layer is `Dense()`, add a `Flatten()` layer. The `Flatten()` is a transformation layer converting the 3-dimensional (*batch size*, *timesteps*, *features*) output from the LSTM layer with a time-steps axis into a 2-dimensional array (*batch size*, *timesteps * features*)

- **Stateful LSTM**. It should not be used as a baseline. A stateful LSTM requires a deeper understanding of the process and problem formulation. For example, whether the process is stationary. If unclear, it is better to avoid stateful LSTM.

- **Dropout**. In LSTM we have two choices for Dropout, viz. the regular dropout using the `Dropout()` layer and recurrent dropout using the `recurrent_dropout` argument in the `LSTM` layer. Among them it is preferred to start with the recurrent dropout and its rate as `0.5` in each LSTM layer, `LSTM(..., recurrent_dropout=0.5)`.

- **Go backward**. The `go_backwards` argument in an LSTM layer allows processing the input sequences in the reverse order. This brings the long-term patterns closer while moving the short-term patterns backward. This switch is useful in some problems, such as language translation.

  The *backward* utility depends on the problem. This setting should be set to `False` in the baseline.

  To test whether it brings any improvement, only the **first** LSTM layer should be toggled to go backward by setting `LSTM(..., go_backwards=True)`.

- **Bi-directional**. A bi-directional LSTM can learn patterns both retrospectively (from the past) and prospectively (from the future). This makes it stronger than regular LSTMs. However, bi-directional LSTM has double the number of parameters. It should not be used in the baseline but must be attempted to validate if it brings any improvement.

  An LSTM layer can be made bi-directional by wrapping it as `bidirectional(LSTM(...))`. Similarly, any other RNN layer can also be made bi-directional using the `bidirectional()` wrapper.

5.10 Exercises

1. LSTM is said to be an enhanced RNN. It brought in the concept of cell state for long-term memory.

   (a) Explain how a simple RNN works and differentiate it with LSTMs.

   (b) In LSTMs, is it possible to identify which features are preserved or forgotten from the memory?

2. **Activation**. In this chapter, `relu` activation is used on the LSTM outputs. But `tanh` is its native activation.

   (a) Explain why `relu` is still applicable as LSTM output activation.

   (b) Train the baseline and bi-directional model with `tanh` activation and discuss the results.

   (c) Train the baseline and bi-directional models with ELU and SELU activations. Do they address the increasing validation loss issue? Discuss your findings.

3. **Peephole LSTM**. Peephole LSTMs are a useful variant. They have proved to work better than others in Graves 2013. Refer to the Peephole LSTM expressions in § 5.7.

   (a) Use the peephole LSTM in TensorFlow[3].

   (b) Implement the peephole LSTM as your custom LSTM cell.

   (c) Discuss the results.

4. **Gated Recurring Unit (GRU)**. GRU proposed in Cho et al. 2014 is an alternative to LSTM. Jozefowicz, Zaremba, and Sutskever 2015 found that GRU outperformed LSTM in many cases. The formulation for GRU is,

[3]Refer to `https://bit.ly/2JnxcyM`

$$
\begin{aligned}
r_t &= \sigma(W_r^{(x)}\boldsymbol{x}_t + W_r^{(h)}\boldsymbol{h}_{t-1} + b_r)\\
z_t &= \sigma(W_z^{(x)}\boldsymbol{x}_t + W_z^{(h)}\boldsymbol{h}_{t-1} + b_z)\\
\tilde{h}_t &= \tanh(W_h^{(x)}\boldsymbol{x}_t + W_h^{(h)}\boldsymbol{h}_{t-1} + b_h)\\
h_t &= z_t h_{t-1} + (1 - z_t)\tilde{h}_t
\end{aligned}
$$

(a) Implement GRU in TensorFlow. Replace the LSTM layer with the GRU layer. Discuss the results.

(b) Jozefowicz, Zaremba, and Sutskever 2015 found that initializing LSTM with forget gate bias as 1 improved the performance. Explain the premise of initializing the forget bias as 1.

(c) Run the baseline and bi-directional LSTM model with forget bias initialized to 1 (set `unit_forget_bias` as `True`). Compare the results with the GRU model.

5. (Optional) **A higher-order cell state**. The cell state in traditional LSTM has a first-order autoregressive (AR-1) like structure. That is, the cell state c_t is additive with the prior c_{t-1}. Will a higher-order AR function work better?

   (a) Build a simple custom LSTM cell with a second-order autoregressive expression for cell state. The formulation is given below. This is a dummy LSTM cell in which cell state at a time t will always have a part of the cell state learned at $t-2$.

$$
\begin{aligned}
i_t &= \texttt{hard-sigmoid}(\boldsymbol{w}_i^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_i^{(h)}\boldsymbol{h}_{t-1} + b_i)\\
o_t &= \texttt{hard-sigmoid}(\boldsymbol{w}_o^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_o^{(h)}\boldsymbol{h}_{t-1} + b_o)\\
f_t &= \texttt{hard-sigmoid}(\boldsymbol{w}_f^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_f^{(h)}\boldsymbol{h}_{t-1} + b_f)\\
\tilde{c}_t &= \texttt{tanh}(\boldsymbol{w}_c^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_c^{(h)}\boldsymbol{h}_{t-1} + b_c)\\
c_t &= b + 0.5c_{t-2} + f_t c_{t-1} + i_t\tilde{c}_t\\
h_t &= o_t\texttt{tanh}(c_t)
\end{aligned}
$$

   where b is a bias parameter.

(b) Build an adaptive second order AR cell state with the help of an additional gate. The formulation is,

$$
\begin{aligned}
i_t &= \texttt{hard-sigmoid}(\boldsymbol{w}_i^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_i^{(h)}\boldsymbol{h}_{t-1} + b_i)\\
o_t &= \texttt{hard-sigmoid}(\boldsymbol{w}_o^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_o^{(h)}\boldsymbol{h}_{t-1} + b_o)\\
f_t &= \texttt{hard-sigmoid}(\boldsymbol{w}_f^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_f^{(h)}\boldsymbol{h}_{t-1} + b_f)\\
g_t &= \texttt{hard-sigmoid}(\boldsymbol{w}_g^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_g^{(h)}\boldsymbol{h}_{t-1} + b_g)\\
\tilde{c}_t &= \texttt{tanh}(\boldsymbol{w}_c^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_c^{(h)}\boldsymbol{h}_{t-1} + b_c)\\
c_t &= b + g_t c_{t-2} + f_t c_{t-1} + i_t \tilde{c}_t\\
h_t &= o_t \texttt{tanh}(c_t)
\end{aligned}
$$

(c) The purpose of this exercise is to learn implementing new ideas for developing an RNN cell. An arbitrary formulation is presented above. How did this formulation work? Will a similar second-order AR formulation for GRUs work? Can you propose a formulation that can outperform LSTM and GRU cells?

Chapter 6

Convolutional Neural Networks

"...We are suffering from a plethora of surmise, conjecture, and hypothesis. The difficulty is to detach the framework of fact—of absolute undeniable fact—from the embellishments..."

– Sherlock Holmes, *Silver Blaze*.

6.1 Background

High-dimensional inputs are common. For example, images are high-dimensional in space; multivariate time series are high-dimensional in both space and time. In modeling such data, several deep learning (or machine learning) models get drowned in the excess confounding information.

Except for convolutional networks. They specialize in addressing this issue. These networks work by **filtering the essence** out of the excess.

Convolutional networks are built with simple constructs, viz. *convolution*, and *pooling*. These constructs were inspired by studies in neuroscience on human brain functioning. And, it is the simplicity of these constructs that make convolutional networks robust, accurate, and efficient in most problems.

> "Convolutional networks are perhaps the greatest success story of biologically inspired artificial intelligence."
>
> – in Goodfellow, Bengio, and Courville 2016.

Convolutional networks were among the first working deep networks trained with back-propagation. They succeeded while other deep networks suffered from gradient issues due to their computational and statistical inefficiencies. And, both of these properties are attributed to the inherent simplistic constructs in convolutional networks.

This chapter describes the fundamentals of convolutional constructs, the theory behind them, and the approach to building a convolutional network.

It begins with a figurative illustration of the convolution concept in § 6.2 by describing a convolution as a simple filtration process. Thereafter, the unique properties of convolution, viz. parameter sharing, (use of) weak filters, and equivariance to translation, are discussed and illustrated.

The convolution equivariance makes a network sensitive to input variations. This hurts the network's efficiency. It is resolved by pooling discussed in § 6.4. Pooling regularizes the network and, also, allows modulating it between equivariance and invariance. These pooling attributes are explained in § 6.4.1 and § 6.4.2, respectively. The sections show that a pooled convolutional network is regularized and robust.

These sections used single-channel inputs for the illustrations. The convolution illustration is extended to multi-channel inputs such as a colored image in § 6.5.

After establishing the concepts, the mathematical kernel operations in convolution is explained in § 6.6. A few convolutional variants, viz. padding, stride, dilation, and 1×1 convolutions are then given in § 6.7.

Next, the elements of convolutional networks are described in § 6.8, e.g., input and output shapes, parameters, etc. Moreover, the choice between `Conv1D`, `Conv2D`, and `Conv3D` layers in TensorFlow is sometimes confusing. The section also explains their interchangeability and compatibility with different input types.

At this point, the stage is set to start multivariate time series modeling with convolutional networks. The model construction and interpretations are in § 6.9. Within this section, it is also shown that convolutional networks are adept at learning long-term temporal dependencies in high-dimensional time series. A network in § 6.9.4 learned temporal patterns as wide as 8 hours compared to only up to 40 minutes with LSTMs in the previous chapter.

Furthermore, a flexible modeling of multivariate time series using a higher-order `Conv2D` layer is given in § 6.10.

After this section, the chapter discusses a few advanced topics in relation to pooling.

Pooling in a convolutional network summarizes a feature map. The summarization is done using *summary statistics*, e.g., average or maximum. § 6.11 delves into the theory of summary statistics intending to discover the statistics that are best for pooling. The section theoretically shows that maximum likelihood estimators (MLEs) make the best pooling statistic.

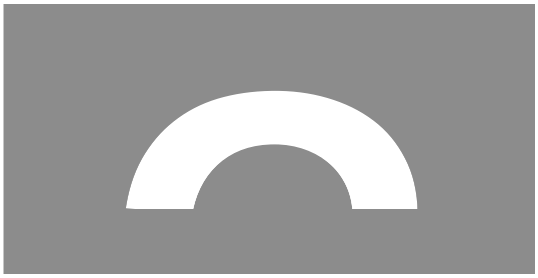
The theory also helps uncover the reason behind max-pool's superiority in § 6.12.1. Importantly, it provides several other strong pooling statistic choices in § 6.13. A few futuristic pooling methods such as adaptive and multivariate statistics are also discussed in § 6.14. A brief history of pooling is then laid in § 6.15.

Lastly, the chapter concludes with a few rules-of-thumb for constructing convolutional networks.

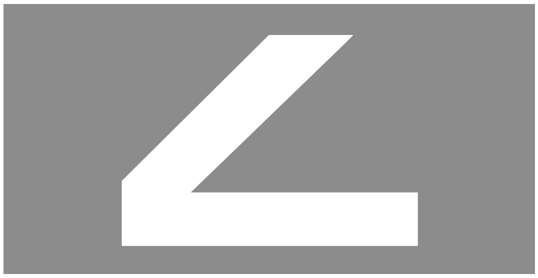
6.2 The Concept of Convolution

The concept of convolution is one of the simplest in deep learning. It is explained in this section with the help of an arbitrary image detection problem but the concept applies similarly to other problems.

Figure 6.1. *An image of the letter "**2**." The problem is to detect the image as **2**. Visually this is straightforward to a human but not to a machine. An approach for the detection is determining filters with distinctive shapes that match the letter and filtering the image (the convolution process) through them.*



(a) *Semi-circle filter.*



(b) *Angle filter.*

Figure 6.2. *Examples of filters for convolution to detect letters in an image. The filters—semi-circle and angle—together can detect letter 2 in an image.*

Problem. Consider an image of the letter "**2**" in Figure 6.1. The problem is to detect the image as "**2**." This is a straightforward task for a human. But not necessarily to a machine. The objective and challenge are to train a machine to perform the task. For which, one approach is *filtration*.

Filtration. There are several ways to perform the letter "**2**" detection. One of them is to learn distinctive *filters* and filter an image through them.

The filters should have shapes that distinguish "**2**." One such set of filters is a *semi-circle* and an *angle* shown in Figure 6.2a and 6.2b. A presence of these shapes in an image would indicate it has "**2**."

The presence of a shape in an image can be ascertained by a filtration-

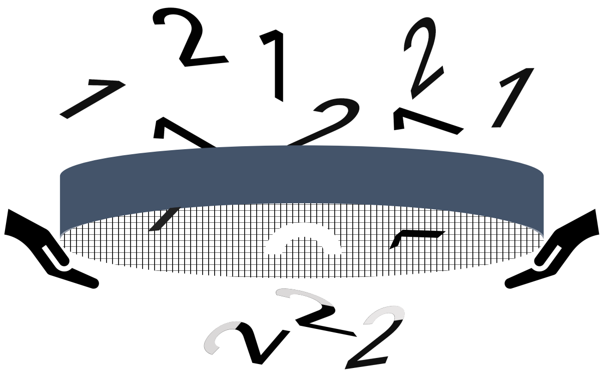
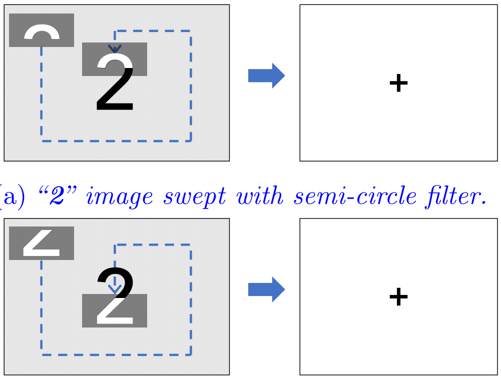


Figure 6.3. *An illustration of convolution as a filtration process through a sieve. The sieve has a semi-circular hole corresponding to a filter. Several letters are sifted through the sieve. The letters that have a semi-circular shape in them falls through it. Consequently, the* ***2****'s pass through the filter while the* ***1****'s do not.*



(a) *"****2****" image swept with semi-circle filter.*

(b) *"****2****" image swept with angle filter.*

Figure 6.4. *Formal implementation of the filtration process in convolution is sweeping the input as illustrated above. An image with "****2****" in it is swept with the semi-circle filter (top) and the angle filter (bottom). The filtration process of detecting the presence of the filter shapes is accomplished by sweeping the entire image to find a match.*

like process. Imagine there is a bucket of images of **1**'s and **2**'s. These images are sifted through a sieve which has a semi-circular hole as shown in Figure 6.3.

While (a part of) **2**'s could pass through the sieve, the **1**'s could not go through. The filtration through the sieve indicated the presence of semi-circle in the **2**'s. Similarly, another filtration through the *angle* filter could be performed to infer an image contains "**2**."

This filtration process is performed in **convolution**. A difference is that the filtration process in convolution appears like *sweeping* an image instead of sifting.

Sweeping. Formal implementation of filtration is sweeping an image with a filter.

Figure 6.4a and 6.4b illustrate the sweeping process in which the entire image is swept by the *semi-circle* and *angle* filters, respectively. In both the sweeps, the respective filter shapes were found and indicated with a + symbol in the output. These outputs enable detection of "**2**."

Convolution. Sweeping an input mathematically translates to a convolution operation.

In deep learning, convolution is performed with discrete kernels (details in § 6.6). The kernel corresponds to a filter and convolving an input with it indicates the presence/absence of the filter pattern.

A convolution operation is equivalent to sweeping an input with a filter in search of the filter's pattern.

Importantly, the pattern's location in the input is also returned. These locations are critical. Without their knowledge, an image with semi-circle and angle strewed anywhere in it will also be rendered as "**2**" (an illustration is in Figure 6.11 in § 6.4.2).

In sum, a convolution operation filters distinguishing patterns in an input. The inputs are typically high-dimensional. The filtration by convolution displays the spirit of extracting the **essence** (patterns) from the **excess** (high-dimensional input).

The illustrations in this section might appear ordinary but remember for a machine it is different. The foremost challenge is that the filters are unknown. And, identifying appropriate filters by a machine is non-trivial. Convolutional networks provide a mechanism to **automatically** learn the filters (see § 6.9). Besides, convolution exhibits some special properties discussed next.

🔔 *Earlier, filters were derived through feature engineering. Convolutional networks automated the filters learning.*

6.3 Convolution Properties

A convolution process sweeps the entire input. Sweeping is done with a filter smaller than the input. This gives the property of *parameter sharing* to a convolutional layer in a deep learning network.

Moreover, the filters are small and, hence, called *weak*. This enables sparse interaction. Lastly, during the sweep the location of filter patterns in the input is also returned which brings the *equivariance* property to convolution.

This section explains the benefits, downsides, and intention behind these convolution properties.

6.3.1 Parameter Sharing

The property of parameter sharing is best understood by contrasting a dense layer with convolutional.

Suppose a dense layer is used for the image detection problem in the previous section. A dense layer would yield a filter of the **same shape and size** as the input image with letter "**2**" as shown in Figure 6.5.

Figure 6.5. *An illustration of a filter in a dense layer to detect letter "2." The filter size is the same as that of the input. Consequently, the parameter space is significantly large. This is also referred to as a strong filter because it can alone detect the letter "2." Although the filter is strong, the excess amount of parameters in it make a dense layer statistically inefficient. Due to statistical inefficiency, a dense layer would require a large number of samples to automatically learn a filter.*

Compare the sizes of dense layer filter in Figure 6.5 with either of the convolutional filters, semi-circle, and angle, in Figure 6.2a and 6.2b. The latter are clearly smaller.

The convolutional filter is **smaller** than the input. But to cover the entire input, it sweeps through it from top to bottom and left to right. This is called *parameter sharing*.

A filter is a kernel that is made of some parameters. When the same filter is swept on different parts of the input, it is referred to as the parameters are shared. This is in contrast to a dense layer filter which is as big as the input and, hence, does not share parameters.

What is the benefit of parameter sharing? Parameter sharing makes a convolutional network statistically efficient. Statistical efficiency is the ability to learn the model parameters with as few samples as possible.

Due to parameter sharing, a convolutional layer can work with small-sized filters—smaller than the input. The small filters have fewer parameters compared to an otherwise dense layer. For instance, if the input

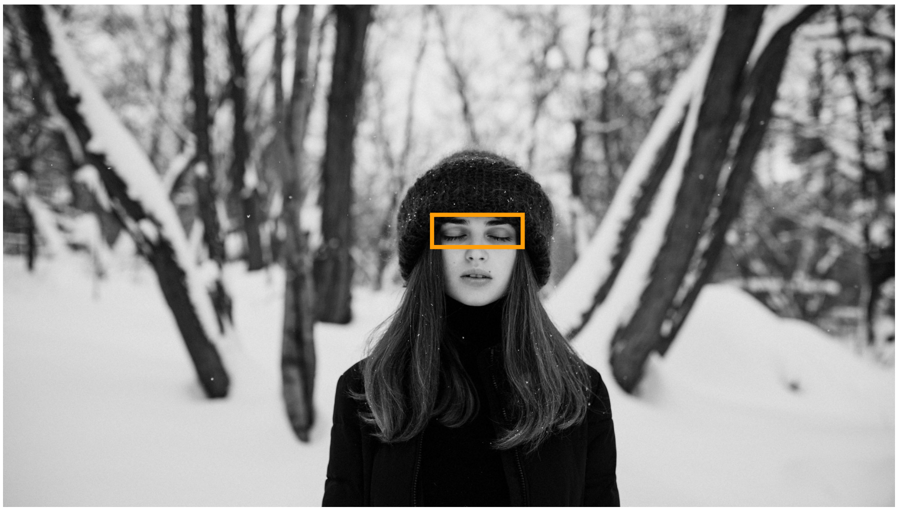


Figure 6.6. *An example of a typical-sized* 2400 × 1600 *image. A convolutional filter to detect a pair of eyes is shown with an orange box. The filter is significantly smaller than the image. Therefore, it is termed as a weak filter. The filter is essentially a set of parameters. This filter sweeps—convolves—the entire image in search of a pair of eyes. This is called parameter sharing. Besides, the convolution process also yields the location of the pair of eyes. This ability to detect positional information is called equivariance. –Photo by Elizaveta Dushechkina on Unsplash.*

image has $m \times n$ pixels, a filter in the dense layer (the weight matrix) is also $m \times n$ but a convolutional filter will be $m' \times n'$, where $m' << m$ and $n' << n$.

The benefit is evident if m and n are large. For instance, a typical-sized image shown in Figure 6.6 is 2400×1600 pixels. In this image, a convolutional filter to detect eyes is 20×60 with $20 * 60 = 1,200$ parameters. This size is significantly smaller than a dense layer weight parameter which is equal to the input size of $2400 * 1600 = 3,840,000$.

Since there are fewer parameters, a convolutional network can learn them with a relatively smaller number of samples. Due to this property, convolutional networks are sometimes also referred to as regularized versions of multilayer perceptrons.

Parameter sharing makes convolutional network statistically efficient, i.e., the network parameters can be learned with fewer samples.

6.3.2 Weak Filters

The statistical efficiency is boosted due to the filter re-usability brought by *weak filters.*

Convolutional layer deploy filters smaller than the input. These are referred to as *weak filters.* The name is because one weak filter by itself is not sufficient. One needs the help of other weak filters for inferencing. For instance, the illustrative example of the letter "**2**" detection required two (weak) filters: a semi-circle and an angle.

The use of weak filters brings an important attribute of *filter reuse* to convolutional nets. **Filter re-usability** is the ability to use a filter for detecting a pattern in multiple objects. For example, the semi-circle filter can be reused to distinguish "**0**," "**2**," "**3**," "**6**," "**8**," and "**9**" from other letters as shown in Figure 6.7. In a letters detection problem, where the letters can be in 0-9, the filter reuse becomes extremely beneficial. Instead of learning strong filters for every letter, a set of weak filters collaborate to detect multiple objects.

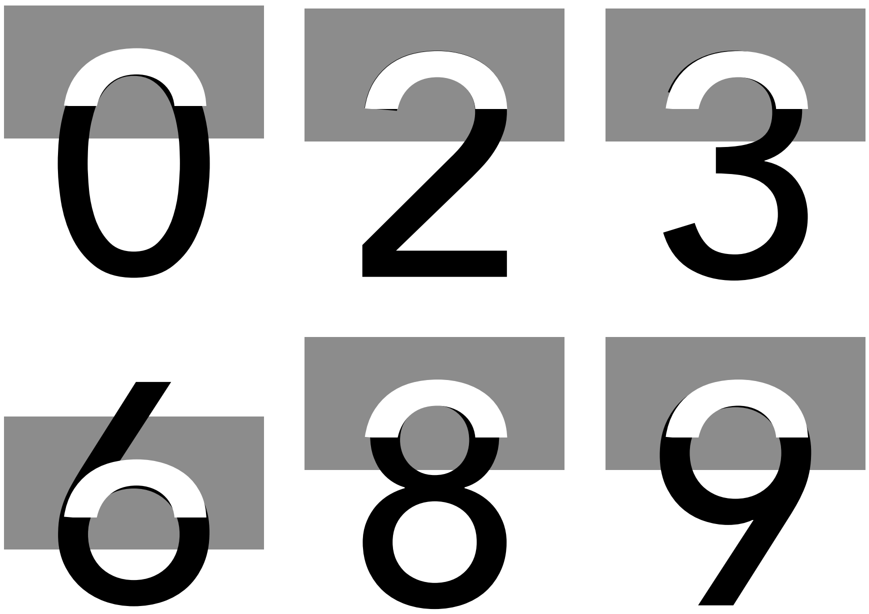


Figure 6.7. *A weak filter semi-circle can detect a distinctive pattern in multiple letters, viz. "**0**," "**2**," "**3**," "**6**," "**8**," and "**9**." In a letters detection problem where the letters can be in 0-9, the semi-circle can separate six of the ten letters. Additional filters, for example, an inverted semi-circle to further separate "**0**," "**3**," "**6**," and "**8**," can be employed. These filters collaborate to infer the letters accurately with fewer parameters to learn. These weak filters and the possibility of their reuse make convolutional networks statistically efficient.*

The benefit of weak filters is that they are reused. A set of weak filters collaborate to detect multiple objects.

A strong filter is capable of detecting an object just by itself. However, it comes at the cost of excessive parameters. Moreover, a strong filter by definition cannot be re-used to detect any other object. An example of a strong filter for detecting "**2**" is in Figure 6.5. This filter cannot be used for any other object. Besides, strong filters are sensitive to distortions (noise) in samples. For example, if the letter "**2**" is written slightly differently, the filter will not work.

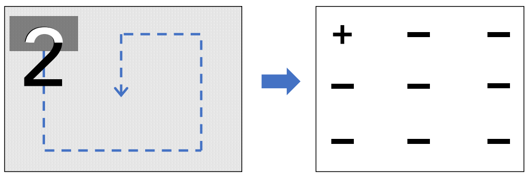
Weak filters, on the other hand, are small in size, reusable, and robust to noise. Due to this, they can be learned with fewer samples. Moreover, due to the reusability and robustness of a weak filter, it is learned from samples of multiple objects. For instance, the semi-circle filter will be automatically learned in a convolutional network using samples of "**0**," "**2**," "**3**," "**6**," "**8**," and "**9**." As a result, the presence of weak filters boosts the statistical efficiency of convolutional networks.

A collection of weak filters make a convolutional network strong. Weak filters are small filters with fewer parameters and also allows filter re-usability. These attributes boost the networks' statistical efficiency.

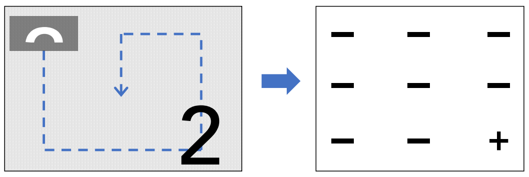
6.3.3 Equivariance to Translation

Convolution exhibits the property of *equivariance to translation*. This means if there is any variation in the input, the output changes in the same way. Due to this property, convolution also preserves the positional information of objects in the input. This is explained below.

As mentioned in the previous § 6.2, convolution is a process of sweeping an input with a filter. At every (sweep) stride, the filter processes the input at the location and emits an output. This is illustrated in



(a) *Letter "2" on the top-left of image.*



(b) *Letter "2" on the bottom-right of image.*

Figure 6.8. *Illustration of the equivariance property of convolution. Convolution is equivariant to translation, which implies if the input changes, the output changes in the same way. For example, "2" is at different positions: top-left (top) and bottom-right (bottom). The convolution output changed at the same locations showing its equivariance.*

Figure 6.8a and 6.8b.

In the figures, the letter "**2**" is placed at the top-left and the bottom-right, respectively. The images are convolved (swept) with the semi-circle filter. The convolution output is numeric but for simplicity, the output is represented as +, if there is a match with the filter, and −, otherwise.

Figure 6.8a and 6.8b show that based on the location of the letter the convolution output changes. In Figure 6.8a, the output has a + at the top-left and the rest are −. But as the position of "**2**" changed to the bottom-right in Figure 6.8b the convolution output also changed to + at the same bottom-right location.

This shows that convolution maps the object location variation in the input. In effect, convolution preserves the information of the object's location.

Equivariance to translation is a property due to which if there is a variation in the input, the convolution output changes in the same way.

Is location preservation important? Not always. In several problems, the objective is to determine the presence or absence of an object or pattern in the input. Its location is immaterial. In such problems, preserving the location makes the model over-representative. An over-representative model has more features than needed and, consequently, excess parameters. This hurts the statistical efficiency. To resolve this, *pooling* is used in conjunction with a convolutional layer.

Due to the equivariance property, the convolutional layer preserves the location information of an object or pattern in an input. This makes the network over-representative that counteracts its statistical efficiency. Adding a pooling layer addresses the issue.

6.4 Pooling

Pooling brings *invariance* to a convolutional network. If a function is invariant, its output is unaffected by any translational change in the input.

A pooling operation draws a summary statistic from the convolution output. This replaces the over-representative convolution output with a single or a few summary statistics. Pooling, therefore, further regularizes the network and maintains its statistical efficiency (illustrated in § 6.4.1).

However, just like equivariance, an invariance to translation is also sometimes counterproductive. Invariance makes the network blind to the location of patterns in the input. This sometimes leads to a network confuse the original input with its distortions (illustrated in Figure 6.10 and 6.11 in § 6.4.2).

Pooling provides a lever to modulate the network between equivariance and invariance. Somewhere between the two is usually optimal.

Thereby, a pooling layer complements a convolutional layer. Moreover, pooling does not have trainable parameters[1]. And, therefore, it does not add a computational overhead.

A pooling layer provides a mechanism to regularize a convolutional network without adding computational overhead.

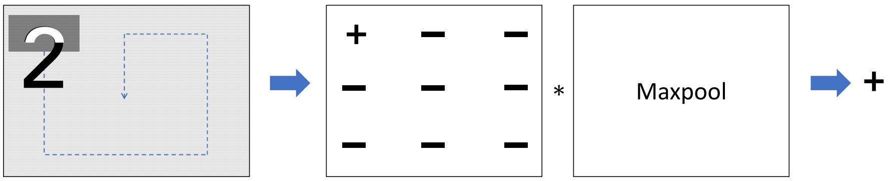
In the following, the first invariance as a means for regularization is explained. It is followed by showing pooling as a tool for modulating a network between equivariance and invariance.

6.4.1 Regularization via Invariance

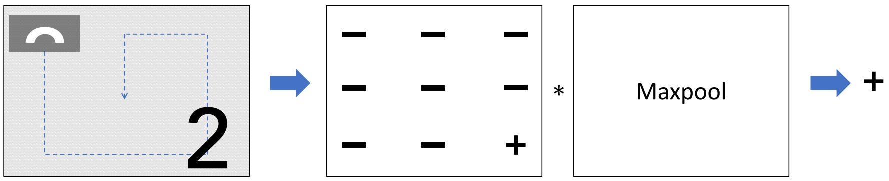
Invariance is the opposite of equivariance. If a function is invariant, its output is unaffected by any translational change in the input.

Pooling brings *invariance* to a convolutional network. A pooling

[1]Unknown parameters to estimate.



(a) *Letter "2" on the top-left of image.*



(b) *Letter "2" on the bottom-right of image.*

Figure 6.9. *Illustration of invariance property of pooling. This illustration shows an extreme level of pooling, called global pooling. In this, the network becomes absolutely invariant. Meaning, the location information of the object is not preserved. Instead, the network focuses only on determining the presence or absence of an object in an input. For example, although the letter "2" is present in top-left and bottom-right in the top and bottom figures, the pooling output remains the same. Here `MaxPool` is used for illustration but the pooling behavior stays the same for any other pooling. This indifference of the output to the translations in the input is called invariance.*

operation summarizes the convolutional output into a statistic(s), called a *summary statistic*. For example, the maximum summary statistic is returned in `MaxPool`. In doing so, the granular information about an object's location is lost.

This causes the network to become invariant to the location of an object in the input. The phenomenon is illustrated in Figure 6.9a and 6.9b. This illustration is a continuation of the one in Figure 6.8 in § 6.3.3 by applying `MaxPool` after the convolution.

As shown in the figures, the maxpool operand takes in the output from convolution and emits the maximum value. Consequently, despite the letter "**2**" being at the top-left or bottom-right in Figure 6.9a and 6.9b, respectively, the output from pooling is the same +.

In effect, pooling regularized the network by reducing the spatial size of the representation (the feature map). This reduces the parameters and, thereby, improves the statistical and computational efficiency. It also improves the network's *generalizability*, i.e., its applicability to more variety of inputs.

> "Invariance to local translation can be a useful property if we care more about whether some feature is present than exactly where it is."
>
> –Goodfellow, Bengio, and Courville 2016.

However, the efficiency improvement is achieved at the cost of losing the location information. Especially, in the illustration here the location information is completely lost. This is because *global pooling* was used in the illustration. Global pooling is the case when the pooling operand has the same size as the convolution output.

Global pooling is extreme pooling. It makes the network absolutely invariant. In simple words, this means that the network becomes indifferent to the location of an object in the input.

However, pooling is usually not used in this extremity. Instead, it is used as a means to modulate a network between equivariance and invariance. This is discussed next.

Pooling regularizes a network by making it invariant, i.e., indifferent, to the location of an object in the input.

6.4.2 Modulating between Equivariance and Invariance

Pooling is used to modulate a network between equivariance and invariance. Both the extremities have downsides.

Equivariance, on one hand, preserves the location information but makes the network over-representative. This hurts the network's computational and statistical efficiencies.

Invariance, on the other hand, regularizes the model to improve its efficiency but loses the location information. This makes the network unable to differentiate between original and manipulated inputs, thereby, hurting its reliability and general applicability.

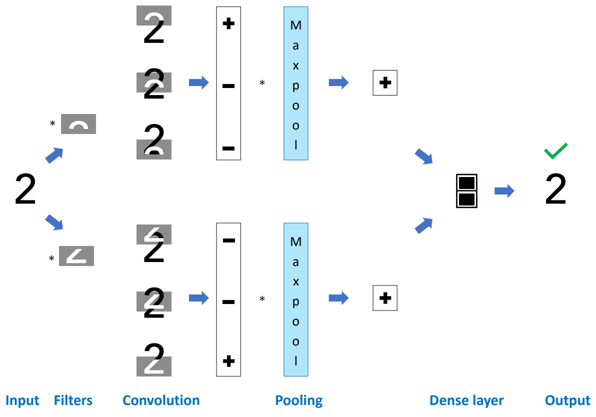
A network is optimal somewhere between the two extremities. Pooling leads to equivariance or invariance depending on whether the pool size is equal to **1** or equal to the convolutional feature map, respectively. Changing the pool size in this range modulates a network between them[2].

At its minimum size of **1**, the pooling output is the same as its input. At its maximum size, pooling makes the network invariant to location translations. The invariance effect is illustrated in Figure 6.10a and 6.10b.

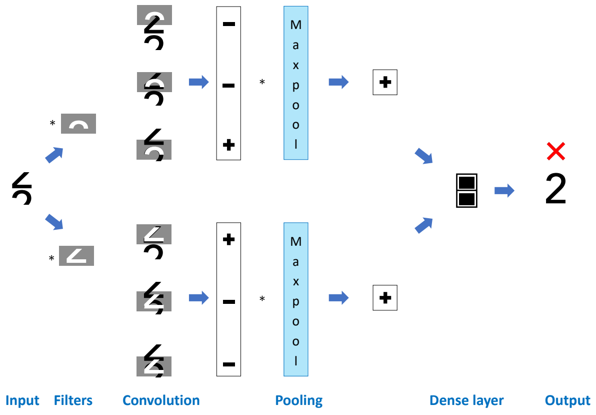
The network in both figures has global pooling. An image of "**2**" and an unknown inscription are analyzed using semi-circle and angle filters in Figure 6.10a and 6.10b, respectively.

The inscription also contains a semi-circle and an angle but in the opposite order compared to "**2**." But the network could not distinguish the inscription from "**2**." The network detected both the inputs, "**2**" and the inscription, as "**2**." The latter (Figure 6.10b) shows that the network

[2]Convolution and pooling, both, are performed along the spatial axes of an input. Therefore, the pool size is defined for the spatial axes. More details are in § 6.8.

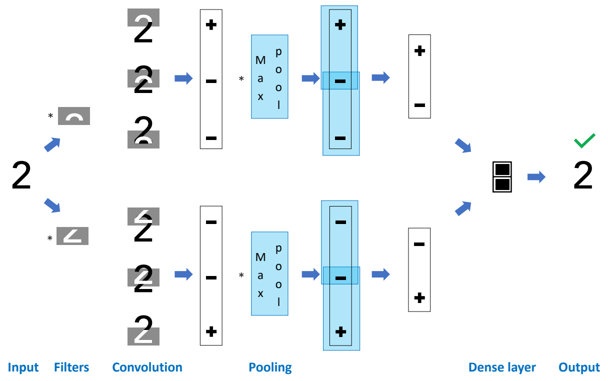


(a) *Letter "2" correctly identified as "2."*

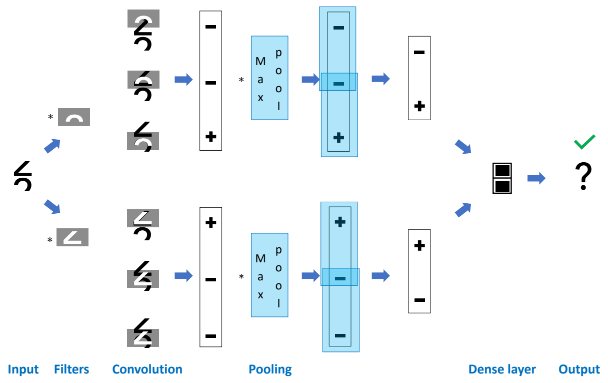


(b) *An inscription incorrectly identified as "2."*

Figure 6.10. *Limitation of invariance is illustrated with global pooling, i.e., pool size equal to its input. This is a pooling extremity that makes the network invariant to location translations. For example, a deformed inscription in the bottom figure has a semi-circle and an angle but not like in a letter "2" is still incorrectly identified as "2."*



(a) *Letter "2" correctly identified as "2."*



(b) *An inscription correctly identified as unknown.*

Figure 6.11. *Pooling modulates the network from invariance to equivariance by reducing the pool size. Doing this preserves the location information and also regularizes the network by reducing the spatial features. With appropriate modulation—reducing pool size to two—the network here is not confused by a manipulative inscription in the bottom figure. Instead, it could correctly identify the top as "2" and the bottom as unknown.*

got confused by the manipulated inscription and incorrectly identified it as "**2**."

This is due to the network's invariance to the location. Although the network's convolutional layer recognized the location of the patterns semi-circle and angle in the inscription as the opposite of "**2**," the pooling layer summarized the convolution output into a single value. As a result, the invariant network could be easily manipulated by an arbitrary distortion.

The issues with equivariance and invariance are resolved by choosing the pool size between **1** and the size of the feature map illustrated in Figure 6.11a and 6.11b.

The network in this illustration has the pool size reduced to two (which is in between the maximum size of 3 and the minimum size of 1). The pooling yields a summary of the feature map within the pool window. Similar to a convolution process, the pool window sweeps the feature map and emits the summary statistic (the maximum, here) at each stride.

A smaller pool size leads to the preservation of some of the location information of patterns. For example, in Figure 6.11a the pooling layer preserved the location information of the semi-circle and angle in the input. Due to which, the network could recognize the letter "**2**" as "**2**" because the semi-circle and angle locations are at the top and bottom of the image, respectively. On the other hand, the location of these patterns is the opposite of the inscription in Figure 6.11b. The network could not recognize this and, therefore, correctly labeled the inscription as *unknown*.

The amount of regularization can be modulated by changing the pool size in pooling. Larger the pool size, the more the regularization, and vice-versa.

A large pool size for higher regularization makes the network invariant to the location of patterns in the input. This makes the network easily confused by manipulative distortions in the input.

The pool size should be chosen such that it provides regularization while preserving the location information.

In sum, a pooling layer provides a mechanism to modulate the network between equivariance and invariance. An appropriately chosen pool size between **1** and the feature map size leads to,

- **a regularized network**. Pooling achieves parameter reduction while preserving location information. This makes the network computationally and statistically efficient. And,

- **a network's robustness** to manipulations by location translations. That is, the network is likely to correctly distinguish the original from its manipulative distortion.

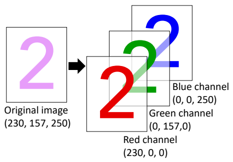
Pooling summarizes the information in a sample. Maximum is one such *summary statistic* and `MaxPool` is one of the most popular pooling methods. More details on summary statistics and the reason for maxpool's popularity is explored towards the end of this chapter in § 6.11 and 6.12. Additionally, deeper insights into the theory behind pooling and a few effective choices are provided there.

6.5 Multi-channel Input

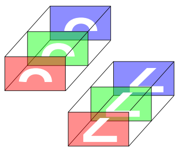
So far the convolutional network constructs were explained using monochrome images as examples. However, most data sets have polychrome, i.e., colorful images.

Despite the varied colors in an image, every color is made from a palette comprising of a few primary colors, e.g., red-green-blue (RGB). Each constituent of the palette is called a *channel*. Therefore, a color image from RGB palette has red, green, and blue channels.

For instance, a multi-channel version of the grayscale image in Figure 6.1 is shown in Figure 6.12a.



(a) *Letter 2 multi-channel: RGB.*



(b) *Multi-channel semi-circle and angle filters.*

Figure 6.12. *Multi-channel image and filters.*

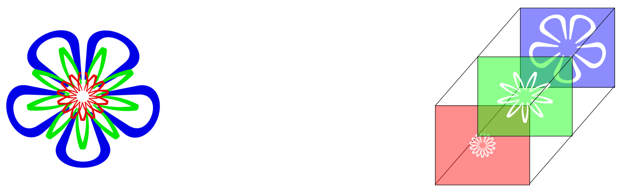
Letter "**2**" in Figure 6.12a is a polychrome image from the RGB palette. As shown, the purple-colored "**2**" is a composition of red, green, and blue shades. More specifically, $(Red : 230, Green : 157, Blue : 250)$. Decomposing them into individual channels yields images in each channel, red $(230, 0, 0)$, green $(0, 157, 0)$, and blue $(0, 0, 250)$. The colors in the image are true to scale.

The importance of visualizing an input as a composition of channels is to understand that it requires the filters (convolutional or pooling) to have the same number of channels. For example, Figure 6.12b shows the semi-circle and angle filters with three channels for red, green, and blue, respectively.

However, the example should not be misunderstood as the filter shapes must be the same in each channel. On the contrary, every channel in a filter can have a different shape.

As an example, Figure 6.13a has a flower with different features (shapes) in different channels. For such inputs, the filters take different shapes (learned in a convolutional network) in different channels as shown in Figure 6.13b.

Moreover, the channels in images are usually independent. However, it is not necessary for other data types such as a multivariate time series.



(a) *Multi-channel data with different features in different channels.* (b) *Multi-channel filter with different shapes in different channels.*

Figure 6.13. *Multi-channel image with different characteristic features in different channels.*

Channels in data can be interpreted in various ways. It is the palette constituents in images but it is the features in a multivariate time series. Therefore, one generic understanding of channels is: every channel provides **different information** about a sample.

A convolutional filter has the same number of channels as in the input.

The shape of a filter can be different in different channels.

6.6 Kernels

Convolution is a filtration process. A filter is made of kernel(s). Rather, a kernel is a mathematical representation of a filter.

Kernels can be either a continuous or a discrete function. Among them, discrete kernels are commonly used in convolutional networks due to their flexibility. This choice is similar to fitting an empirical distribution in statistical analysis. Discrete kernels make convolutional networks robust to the distribution of input features.

Suppose, f is a discrete kernel and g is a mono-channel image, then the convolution of f and g is expressed as,

$$f * g = \left[\sum_{u=-\infty}^{+\infty} \sum_{v=-\infty}^{+\infty} f(u,v) g(x-u, y-v) \right], \forall x, y. \tag{6.1}$$

Here, f is a two-dimensional kernel because g is two-dimensional. The dimensions here are *spatial*. For statisticians, the term dimension can be confused with the number of features. Therefore, in this chapter, *axis* is used to denote a spatial dimension.

As mentioned in § 6.2, a convolution operation involves sweeping the input. The sweeping occurs along each axes of the input. Therefore, because the input g has two axes the convolution operation in Equation 6.1 is a double summation along both.

A mono-channel image is a two-axes $m \times n$ matrix. For illustration, suppose the image is a 5×5 matrix,

$$G = \begin{bmatrix} g_{11} & g_{12} & g_{13} & g_{14} & g_{15} \\ g_{21} & g_{22} & g_{23} & g_{24} & g_{25} \\ g_{31} & g_{32} & g_{33} & g_{34} & g_{35} \\ g_{41} & g_{42} & g_{43} & g_{44} & g_{45} \\ g_{51} & g_{52} & g_{53} & g_{54} & g_{55} \end{bmatrix}$$

A discrete kernel in convolutional networks is essentially a tensor. For the two-axes G, a $k \times l$ tensor is taken as a kernel. Consider such an arbitrary 3×5 kernel,

$$F = \begin{bmatrix} f_{11} & f_{12} & f_{13} & f_{14} & f_{15} \\ f_{21} & f_{22} & f_{23} & f_{24} & f_{25} \\ f_{31} & f_{32} & f_{33} & f_{34} & f_{35} \end{bmatrix}$$

Remember from § 6.2 that a convolution operation is like filtration. Mathematically, this is equivalent to determining the similarity between the kernel F and the input G via a convolution operation.

Referring back to Figure 6.8, a convolution operation is simply sweeping a kernel over an image and an output is returned at every stride. The output is a **dot product** of the kernel and a section of the image.

The operation can be expressed by rewriting Equation 6.1 as,

$$F * G = \left[\sum_{u=1}^{k} \sum_{v=1}^{l} f_{uv} g_{x-u+1, y-v+1} \right], \forall x, y \tag{6.2}$$

where $x = k, \ldots, m$, and $y = l, \ldots, n$. Note that x and y range from k and l onward to keep the indexes $x - u + 1$ and $y - v + 1$ positive for all values of u and v. The convolution happens iteratively for every value of x and y. Every iteration is a dot product between the kernel f and a section of input g at (x, y).

The higher the dot product, the stronger is the similarity between the section and the kernel. The more instances of high dot products indicate a significant part of the input G is similar to the kernel F.

In the following, the convolution operation that was figuratively illustrated earlier is illustrated again with tensor (matrix) operations.

The letter "**2**" is represented as a matrix G in Equation 6.3, and *semi-circle* and *angle* filters as kernel matrices F_s and F_a, respectively, in Equation 6.4 and 6.5. The non-zero entries are highlighted in yellow and pink in G and F's, respectively.

$$G = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{6.3}$$

$$F_s = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{6.4}$$

$$F_a = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{6.5}$$

Here the G is a 5×5 matrix and $F_{.}$'s are 3×5 tensors, i.e., $m = 5, n = 5, k = 3$ and $l = 5$. Therefore, the range of the iterators x, y in Equation 6.2 are, $x \in \{3, 4, 5\}$ and $y \in \{5\}$. The convolution operation occurs for every (x, y) pairs $(3, 5), (4, 5), (5, 5)$ as shown below.

$$F_{.} * G = \Bigg[\underbrace{\sum_{u=1}^{3} \sum_{v=1}^{5} f_{uv} g_{3-u+1,5-v+1}}_{A\,(x=3,y=5)}, \underbrace{\sum_{u=1}^{3} \sum_{v=1}^{5} f_{uv} g_{4-u+1,5-v+1}}_{B\colon (x=4,y=5)}, \\ \underbrace{\sum_{u=1}^{3} \sum_{v=1}^{5} f_{uv} g_{5-u+1,5-v+1}}_{C\colon (x=5,y=5)} \Bigg] \tag{6.6}$$

The operation for F_s is shown below in parts (A), (B), and (C) for (x, y) pairs $(3, 5), (4, 5), (5, 5)$, respectively.

$$A = \begin{matrix} 0*0+0*0+1*1+0*0+0*0+ \\ 0*0+1*1+0*0+1*1+0*0+ \\ 0*0+0*0+0*0+0*1+0*0 \end{matrix} = 3,$$

$$B = \begin{matrix} 0*0+0*1+1*0+0*1+0*0+ \\ 0*0+1*0+0*0+1*1+0*0+ \\ 0*0+0*0+0*1+0*0+0*0 \end{matrix} = 1, \text{ and}$$

$$C = \begin{matrix} 0*0+0*0+1*0+0*1+0*0+ \\ 0*0+1*0+0*1+1*0+0*0+ \\ 0*0+0*1+0*1+0*1+0*0 \end{matrix} = 0$$

Putting (A), (B), and (C) together, the convolution output for F_s is

$$F_s * G = [3, 1, 0] \,. \tag{6.7}$$

Similarly, the convolution output for F_a is,

$$F_a * G = [1, 2, 5] . \tag{6.8}$$

Earlier in Figure 6.10a and 6.11a in § 6.4.2, the convolution outputs corresponding to Equation 6.7 and 6.8 were represented as $F_s * G = [+, -, -]$ and $F_a * G = [-, -, +]$ for simplicity. However, as shown in this section, the convolution output is a continuous real number instead of binary.

So far, the kernels are illustrated for mono-channel inputs. But inputs can be multi-channel as shown in § 6.5 in which case a filter has as many channels as the input. A multi-channel filter is made up of multiple kernels—one kernel for each channel. The convolution is then expressed as

$$F * G = \left[\sum_{u=1}^{k} \sum_{v=1}^{l} \sum_{c=1}^{n_c} f_{uvc} g_{x-u+1, y-v+1, c} \right], \forall x, y \tag{6.9}$$

where n_c denotes the number of channels.

In summary, a few things to note in the convolution operation are,

- The cross-sectional size of a kernel is smaller than the input, i.e., $k \leq m$ and $l \leq n$, but they have the same number of channels; sometimes also referred to as the *depth*.
- The dot product is between the kernel and a section of the input's cross-section at (x, y) through its entire depth.
- Irrespective of the depth, the dot product still yields a scalar.
- Consequently, the convolution output shape does not change with the number of channels in the input. This is an important deduction that helps understand the importance of 1×1 convolution in § 6.7.4.

Besides, Equation 6.9 is applicable on a two-axes multi-channel inputs, e.g., a color image. A summation is added (removed) for inputs with additional (lesser) axes, e.g., a video has three axes while time series has a single axis.

A filter is mathematically a kernel. A discrete kernel, which is simply a matrix, is typically used in convolutional networks.

A filter has a kernel for every channel in the input.

The shape of a convolution output does not change with the number of channels in the input.

6.7 Convolutional Variants

6.7.1 Padding

In the previous section, convolution was illustrated using an arbitrary input and kernel of sizes 5×5 and 3×5, respectively. It was observed in Equations 6.7 and 6.8 that the 5×5 input reduced to a 3×1 output.

Consider a network with several such layers. At every layer the features size will shrink considerably.

The shrinking of the feature map is not a problem if the network is shallow or the input is high-dimensional. The feature reduction (regularization) is, in fact, beneficial as it reduces the dimension.

However, a rapid reduction in the feature map size prohibits the addition of any subsequent layer or makes the higher-level layers futile. It is because we can quickly run out of the features available for the next convolutional layer. This issue easily becomes an impediment against constructing deep networks.

Feature maps shrink considerably in traditional convolution. It becomes an issue in deep networks as we quickly run out of features.

$$G' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(a) *Zero-padding shown in gray.*

$$F_s * G' = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 3 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 2 & 0 \\ 1 & 1 & 3 & 1 & 1 \end{bmatrix}$$

(b) *Convolution output from the "semi-circle" kernel, F_s.*

$$F_a * G' = \begin{bmatrix} 1 & 1 & 3 & 1 & 1 \\ 0 & 2 & 1 & 2 & 1 \\ 1 & 1 & 2 & 2 & 0 \\ 1 & 2 & 5 & 2 & 1 \\ 0 & 2 & 1 & 1 & 0 \end{bmatrix}$$

(c) *Convolution output from the "angle" kernel, F_a.*

Figure 6.14. *Illustration of padding in convolutional layers on a 5×5 image of "2." Convolving the input with kernels F_s and F_a of size 3×5 yielded a 3×1 output in Equation 6.7 and 6.8. This significantly downsized the feature map. The precipitous downsizing prohibits stacking multiple layers as the network quickly runs out of features. This is resolved with padding. Padding works by appending zeros on the periphery of the input. Convolving the padded input (top) with F_s and F_a again results in a 5×5 outputs (bottom) which are of the same size as the input. Padding, thus, prevents the feature map from diminishing and, hence, enables constructing deeper networks.*

The issue can be resolved with a padded convolutional layer. In padding, the size of the input sample is artificially increased to augment the outputted features map size.

Padding was originally designed to make the size of the feature map equal to the size of the input. It is done by appending 0's on the periphery of input. Figure 6.14a illustrates padding of the 5×5 matrix G for the letter "**2**" by adding rows and columns of 0's to form an artificially enlarged input G'. Convolving the enlarged G' with F_s and F_a resulted in an 5×5 output in Figure 6.14b and 6.14c. The outputs are of the **same** size as the original 5×5 input G. It is, thus, called "same" padding.

"Same" padding employs the **maximum** amount of padding. More than this is pointless as it will only add rows and/or columns of zeros in the output. The default convolutional layer with **no** padding is referred to as "valid" padding.

The amount of padding can be fine-tuned between "valid" and "same" padding to optimize the feature size. However, such a fine-tuning should generally be avoided as feature size reduction can be achieved better with pooling.

"Same" padding is useful to construct deep networks. It prevents the feature maps from diminishing allowing high-level layers to extract predictive patterns.

Padding can be done in TensorFlow by setting the `padding` argument as `Conv(..., padding="same")`. Convolution layer, otherwise, has no padding, by default, and can be explicitly set as `Conv(..., padding="valid")`.

6.7.2 Stride

Stride in the context of convolution virtually means the same as a stride in plain English. Stride means the distance between two steps. During a convolution, every iteration in (x, y) in Equation 6.9 is a step. By default, x and y are iterated with steps of one. In this case, the stride is 1. However, the stride can be increased to skip some steps.

For instance, in the example of convolving a 5×5 input with 3×5 kernels in § 6.6, x is in $\{3, 4, 5\}$. It means the convolution operation is taking steps in the direction of x as $3 \rightarrow 4$ and $4 \rightarrow 5$. The stride here is 1, which is the lowest possible.

But if the input size is large, convolving every step in the input is computationally intensive. In such cases, it could be useful to skip steps by setting the stride greater than 1. In the above example, a `stride=2` reduces the convolution iterations from 3 ($x = \{3, 4, 5\}$) to 2 ($x = \{3, 5\}$).

In large inputs, a stride of 2 significantly reduces the computation time but generally does not affect the accuracy. Despite the benefit, a larger stride (> 2) is often not used as it can adversely affect the network's performance.

A stride of 2 can help reduce computation when the input size is large. However, a larger stride should be avoided.

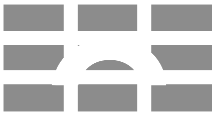
In TensorFlow, the `strides` argument is a tuple of size $1, 2$, or 3 for `Conv1D`, `Conv2D`, and `Conv3D`, respectively. The tuple specifies the stride along each spatial axes, e.g., height, or width. If a single number is given in the argument, the same stride is applied along all the axes.

6.7.3 Dilation

Dilation means to spread out or expand. This is precisely the purpose of dilation in convolution. When a filter is dilated, it expands by adding empty spaces between its elements.

An illustration of the dilated *semi-circle* and *angle* filters are in Figure 6.15a and 6.15b. In its implementation, the filters' kernels are interspersed with 0's.

A dilated kernel expands its coverage while having fewer elements. Due to lesser elements, a dilated filter reduces the computation. It is


(a) *Dilated "semi-circle" filter.*

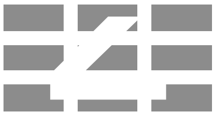

(b) *Dilated "angle" filter.*

Figure 6.15. *Illustration of dilated filters. In dilated convolutional layers, the filters are expanded by uniformly adding rows and columns of zero along its spatial axes. This helps in increased coverage of the input with fewer parameters.*

also shown to improve accuracy in certain problems. For example, F. Yu and Koltun 2016 exemplified the benefits of dilated convolution in semantic segmentation problem.

However, replacing a regular filter with a dilated filter sometimes reduces accuracy. To address this, an un-dilated layer and dilated layer are put together in a network such that the latter is larger than the former. This approach achieves an increased coverage of the input without a significant increase in computation.

A dilated convolution layer used together with a regular un-dilated convolution layer increases the coverage of input that typically improves the accuracy without a significant increase in computation.

The dilation argument in Tensorflow is `dilation_rate`. Similar to `strides`, this is also a tuple of size $1, 2$, or 3 for `Conv1D`, `Conv2D`, and `Conv3D`, respectively, that specifies the dilation along each spatial axes. And, if a single number is given, the same dilation is applied along all the axes. Also, note that the term `dilation_rate` should **not** be confused to be a float between 0 and 1. Instead, it a positive integer, i.e., *dilation_rate* $\in \mathbb{I}^{+}$. The integer denotes the number of zero rows/-columns interspersed in a kernel.

Besides, dilation should not be confused with stride. Both reduce computation for large inputs. But a dilated convolutional layer has expanded filters. On the other hand, increasing stride does not change the filter size. It only skips some steps.

6.7.4 1x1 Convolution

There is a special and popular convolutional layer that has filters of size 1×1. This means the filter is unlikely to have a pattern to detect in the input. This is contrary to the filtration purpose of convolutional layers. Then, what is its purpose?

Its purpose is different from conventional convolutional layers. Conventional convolutions' purpose is to detect the presence of certain patterns in the input with a filtration-like approach. Unlike them, a 1×1 convolution's purpose is only to **amalgamate** the input channels.

Figure 6.16 illustrates a 1×1 convolution that aggregates the color channels of an image to translate it into a grayscale. As shown in the figure, the 1×1 convolution kernel has the same depth, i.e., channels, as the input. The 1×1 filter moves systematically with a stride of one across the input without any padding (dilation is not applicable in a 1×1 filter) to output a feature map with the same height and width as the input **minus** the depth.

A 1×1 filter does not attempt to learn any pattern. Instead, it is a linear projection of the input. It significantly reduces the dimension of the features and, consequently, the parameters in a network. Besides, some researchers consider a 1×1 convolution as "pooling" of the channels. It is because a pooling layer does not summarize features along the channels but a 1×1 convolution does exactly that.

Moreover, while a single 1×1 convolution reduces the features depth to 1, multiple 1×1 convolution cells in a layer bring different summaries of the channels. The number of cells can be adjusted to increase or decrease the resultant feature map depth.

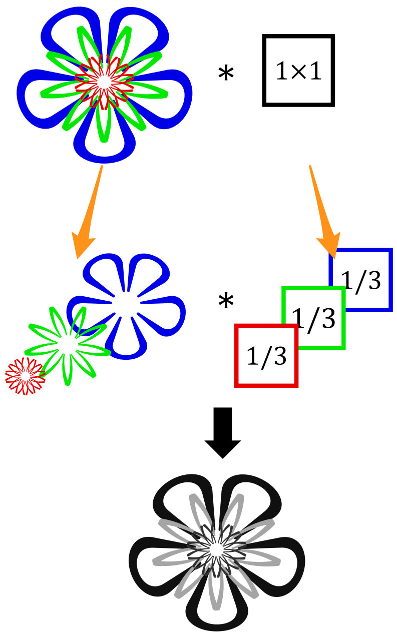


Figure 6.16. *Illustration of an application of a* 1×1 *convolution. The input is a color image with red, blue, and green channels. The channels are intended to be summarized to reduce the input's dimension. A* 1×1 *convolution can be used here to transform the image into a mono-channel grayscale image. For this, a* 1×1 *convolutional kernel with a weight parameter* $1/3$ *is taken. The image and the kernel are expanded here to show the original channels which get amalgamated into a single channel grayscale output. In a convolutional layer, the weight parameter is learned during the network training. A layer can have multiple* 1×1 *convolution cells. Each cell will yield a new transformed channel, e.g., different color schemes in images such as gray, sepia, chrome, vivid, etc. A* 1×1 *convolutional layer can, thus, be used to increase or reduce the channels.*

A 1×1 convolutional layer does not look for shape or object patterns. Instead, its purpose is to amalgamate the signals in the input channels. It can be imagined as the "pooling" of channels.

1×1 convolution plays a critical role in constructing deep networks where the feature maps have to be modulated at different stages. It was initially investigated in Lin, Q. Chen, and Yan 2013. Afterward, Szegedy et al. 2015; He et al. 2016 used them to develop notably deep networks.

6.8 Convolutional Network

6.8.1 Structure

This section exemplifies the structure of a convolutional network. Figure 6.17 illustrates an elementary convolutional network. The components of the network are as follows,

- **Grid-like input**. Convolutional layers take grid-like inputs. The input in the figure is like an image, i.e., it has two axes and three channels each for blue, red, and green.
- **Convolutional layer**. A layer comprises filters. A filter is made up of kernels. It has one kernel for each input channel. The size of a layer is essentially the number of filters in it which is a network configuration. Here, five illustrative filters, viz., diagonal stripes, horizontal stripes, diamond grid, shingles, and waves, are shown in the convolutional layer. Each of them has blue, red, and green channels to match the input.
- **Convolutional output**. A filter sweeps the input to tell the presence/absence of a pattern and its location in the input. In the figure, the outputs corresponding to each filter are shown with a black square of the same pattern. It must be noted that the colored channels in the input are absent in the layer's output. This is

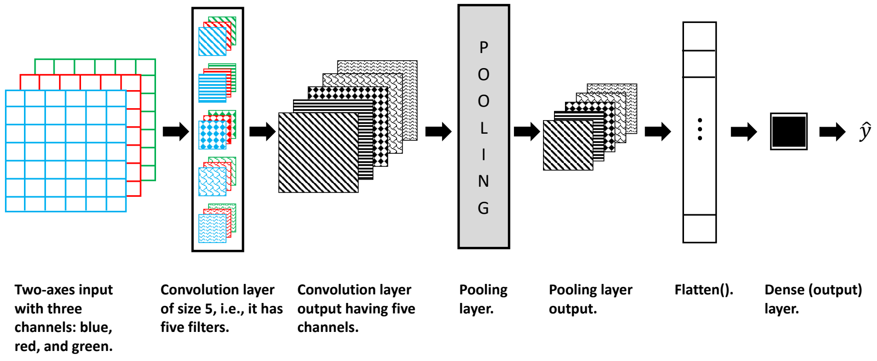

P
O
O
L
I
N
G
$\hat{y}$
Two-axes input with three channels: blue, red, and green.
Convolution layer of size 5, i.e., it has five filters.
Convolution layer output having five channels.
Pooling layer.
Pooling layer output.
Flatten().
Dense (output) layer.

Figure 6.17. *Convolutional networks have a grid-like input. In this illustration, the input is like an image, i.e., it has two axes and three channels for blue, red, and green. The convolutional layer has five illustrative filters, viz., diagonal stripes, horizontal stripes, diamond grid, shingles, and waves. Each filter has as many channels as the input which are blue, red, and green here. The convolutional output from a filter aggregates the information from all the channels. As a result, the input's original channels are relinquished. Instead, each filter's output becomes a channel for the next layer. The output from each filter is, therefore, shown in a respective same pattern black square as a channel. They form a feature map with five channels which is the input to the pooling layer. This layer summarizes and reduces the spatial features but maintains the channel structure. More sets of (higher level) convolutional and pooling layers can be stacked with a caution that sufficient features are passed on after every layer. Towards the end of the network, the output of the convolutional/pooling layer is flattened and followed by a dense output layer.*

because during the convolution operation the information across the channels is aggregated. Consequently, the original channels in the input are relinquished. Instead, the output of each filter becomes a channel for the next layer.

- **Pooling layer**. A convolutional layer is conjoined with a pooling layer. In some texts, e.g., Goodfellow, Bengio, and Courville 2016, a convolutional layer is defined as a combination of a convolutional and pooling layer. The pooling layer summarizes the spatial features, which are the horizontal and vertical axes in the figure.

- **Pooling output**. Pooling reduces the sizes of the spatial axes due to a data summarization along the axes. This makes the network invariant to minor translations and, consequently, robust to noisy inputs. It is important to note that the pooling occurs only along the spatial axes. Therefore, the number of channels remains intact.

- **Flatten**. The feature map thus far is still in a grid-like structure. The flatten operation vectorizes the grid feature map. This is necessary before passing the feature map on to a dense output layer.

- **Dense (output) layer**. Ultimately, a dense layer maps the convolution derived features with the response.

The purpose of a convolutional network is to automatically learn predictive filters from data. Multiple layers are often stacked to learn from low- to high-level features. For instance, a face recognition network could learn the edges of a face in the lower layers and the shape of eyes in the higher layers.

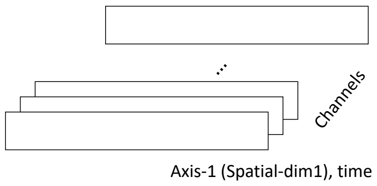
The purpose of a convolutional network is to automatically learn the filters.

6.8.2 `Conv1D`, `Conv2D`, and `Conv3D`

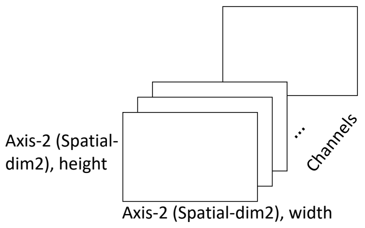
In Tensorflow, the convolutional layer can be chosen from `Conv1D`, `Conv2D`, and `Conv3D`. The three types of layers are designed for inputs with one, two, or three spatial axes, respectively. This section explains when each of them is applicable and their interchangeability.

Convolutional networks work with grid-like inputs. Such inputs are categorized based on their axes and channels. Table 6.1 summarizes them for a few grid-like data, viz. time series, image, and video.

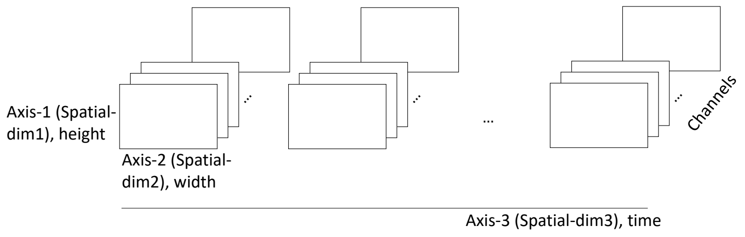
A (univariate) time series has a single spatial axis corresponding to the time. If it is multivariate then the features make the channels. Irrespective of the number of channels, a time series is modeled with `Conv1D` as it has only one spatial dimension.



(a) *Input shape for `Conv1D`.*



(b) *Input shape for `Conv2D`.*


Axis-1 (Spatial-dim1), height
Axis-2 (Spatial-dim2), width
...
Channels
Axis-3 (Spatial-dim3), time

(c) *Input shape for `Conv3D`.*

Figure 6.18. *`Conv` layer for different types of input shapes.*

Images, on the other hand, have two spatial axes along their height and width. Videos have an additional **spatial** axis oxymoronically along **time**. `Conv2D` and `Conv3D` are, therefore, applicable to them, respectively. The channels in them are the palette colors such as red, green, and blue.

Table 6.1. Axes and channels in grid-like inputs to convolutional networks.

	Time series	**Image**	**Video**
Axis-1 (Spatial dim1)	Time	Height	Height
Axis-2 (Spatial dim2)	-	Width	Width
Axis-3 (Spatial dim3)	-	-	Time
Channels	Features (one in univariate time series)	Colors	Colors
Conv'x'd	`Conv1D`	`Conv2D`	`Conv3D`
Input Shape	(samples, time, features)	(samples, height, width, colors)	(samples, height, width, time, colors)
`kernel_size`[3]	An integer, t, specifying the time window	An integer tuple, (h, w), specifying the height and width window.	An integer tuple, (h, w, t), specifying the height, width, and time window.
Kernel shape	(t, features)	(h, w, colors)	(h, w, t, colors)

`Conv1D`, `Conv2D`, and `Conv3D` are used to model inputs with one, two, and three spatial axes, respectively.

The `Conv‘x’D` selection is independent of the channels. There could be any number of channels of arbitrary features. Regardless, the `Conv‘x’D` is chosen based on the number of spatial axes only.

Inputs to `Conv1D`, `Conv2D`, and `Conv3D` are structured as N-D tensors of shape `(samples, time_steps, features)`, `(samples, height, width, channels)`, and `(samples, height, width, channels)`, respectively. The first axis is reserved for samples for almost every layer in TensorFlow. The shape of a sample is defined by the rest of the axes (shown in Figure 6.18a-6.18c). Among them, the last axis corresponds to the channels (by default) in any of the `Conv‘x’D` layers[4] and the rest are the spatial axes.

The `kernel_size` argument in `Conv‘x’D` determines the spatial dimension of the convolution kernel. The argument is a tuple of integers. Each element of the tuple corresponds to the kernel's size along the respective spatial dimension. The depth of the kernel is fixed and equal to the number of channels. The depth is, therefore, not included in the argument.

`Conv` layers are agnostic to the number of channels. They differ only by the shape of the input's spatial axes.

Besides, one might observe that a `Conv2D` can be used to model the inputs of `Conv1D` by appropriately reshaping the samples. For example, a time series can be reshaped as `(samples, time, 1, features)`. Similarly, a `Conv3D` can be used to model the inputs of both `Conv1D` and `Conv2D` by reshaping a time series as `(samples, 1, 1, time, features)`

[4]The position of the channel is set with `data_format` argument. It is `channels_last` (default) or `channels_first`.

and an image as `(samples, height, width, 1, colors)`[5]. Essentially, due to their interchangeability, a universal `Conv3D` layer could be made to work with a variety of inputs. The three variants are, still, provided in TensorFlow for convenience.

Additionally, it is worth to learn that a network can be formulated differently by moving the features on the channels to a spatial axis. For example, a multivariate time series can be modeled like an image with a `Conv2D` by reshaping it from `(samples, time, features)` to `(samples, time, features, 1)`. This approach is shown in § 6.10. And, similarly, the channels of an image can be put on a spatial axis as `(sample, height, width, colors, 1)` and modeled with a `Conv3D`.

In short, the input types and the convolutional layer variants are **not** rigidly interlocked. Instead, it is upon the practitioner to formulate the problem as appropriate.

The choice of `Conv1D`, `Conv2D`, and `Conv3D` are not tightly coupled with specific input types. They can be used interchangeably based on a problem.

6.8.3 Convolution Layer Output Size

It is essential to understand the size of a convolutional layer output in a different scenario to construct a network. A typical un-padded convolutional layer, for example, downsizes the feature map. Constructing deep networks without keeping the track of output downsizing may result in an ineffective network.

The output size of a filter in a convolutional layer is,

$$\boldsymbol{o} = \left\lfloor \frac{\boldsymbol{i} - \boldsymbol{k} - (\boldsymbol{k} - 1)(\boldsymbol{d} - 1) + 2\boldsymbol{p}}{\boldsymbol{s}} \right\rfloor + 1 \qquad (6.10)$$

where,

[5]In such a restructuring, the `kernel_size` along the unit axes is also made `1`.

- $\boldsymbol{i}$ Size of input's spatial axes,
- $\boldsymbol{k}$ Kernel size,
- $\boldsymbol{s}$ Stride size,
- $\boldsymbol{d}$ Dilation rate,
- $\boldsymbol{p}$ Padding size,

and, each parameter in the equation is a tuple of the same size as the number of spatial axes in the input. For instance, in the convolution example between G and $F_{s,a}$ in § 6.6,

- $\boldsymbol{i}$ $= (5, 5)$, size of the spatial axes of input G,
- $\boldsymbol{k}$ $= (3, 5)$, size of the kernel $F_.$,
- $\boldsymbol{s}$ $= (1, 1)$, the default single stride,
- $\boldsymbol{d}$ $= (1, 1)$, the default no dilation, and
- $\boldsymbol{p}$ $= (0, 0)$, the default no padding.

The output size is computed as $\boldsymbol{o} = \left\lfloor \frac{(5,5)-(3,5)-((3,5)-1)((1,1)-1)+2*(0,0)}{(1,1)} \right\rfloor + 1 = (3, 1)$.

Furthermore, a convolutional layer with l filters has an $(\boldsymbol{o}, l)$ tensor as the output where l corresponds to the channels. Extending the example in § 6.6, suppose a convolutional layer has the semi-circle, F_s, and angle, F_a, filters. The output will then be a $(3, 1, 2)$ tensor where the last tuple element 2 correspond to channels—one from each filter.

It is worth noting that the original channels in the input are not included in the output size computation. As mentioned in the previous § 6.8.1, the convolution operation aggregates the channels due to which the original channels are lost in the output. This also implies that a network construction by tracking the output downsizing is unrelated to the input channels.

6.8.4 Pooling Layer Output Size

The output size of the pooling layer is governed by similar parameters as the convolutional. Analogous to the kernel size, a pooling layer has pool size. Strides and padding are also choices in pooling. Dilation, however, is not available in pooling.

Based on them, the pooling output size for a channel is,

$$\boldsymbol{o} = \left\lfloor \frac{\boldsymbol{i} - \boldsymbol{k} + 2\boldsymbol{p}}{\boldsymbol{s}} \right\rfloor + 1 \tag{6.11}$$

where,

- $\boldsymbol{i}$ Size of input's spatial axes,
- $\boldsymbol{k}$ Pool size,
- $\boldsymbol{s}$ Stride size, and
- $\boldsymbol{p}$ Padding size.

If there are l channels, the pooling output is an $(\boldsymbol{o}, l)$ tensor. Essentially, the pooling happens independently for every channel, i.e., the values in the channels are not merged.

Moreover, note that "same" padding does not result in downsampling. Still, the pooling operation brings the invariance attribute to the network and, hence, useful.

6.8.5 Parameters

The number of parameters in a `Conv` layer can be simply expressed as the `((filter volume) + 1) * (number of filters)`.

A filter has weight parameters equal to its volume plus a bias parameter. The *volume* of a filter is the product of the spatial axes (kernel size, $\boldsymbol{k}$) and the depth (the number of channels, c, in the input). For example, in § 6.6 the kernel shape $\boldsymbol{k}$ is $(3, 5)$ and number of channels c for the grayscale input is 1. Therefore, the filter volume is $3 * 5 * 1 = 15$. If the input was in RGB, i.e., $c = 3$, the filter volume becomes $3*5*3 = 45$.

The number of filters is equal to the parameter `filters` set in the definition of a `Conv` layer.

Unlike the `Conv` layer, most of the pooling layers such as `MaxPool` and `AveragePooling` in TensorFlow does not have any trainable parameters.

6.9 Multivariate Time Series Modeling

The rare event prediction problem explored in this book is a multivariate time series. This section proceeds with modeling it with convolutional networks.

6.9.1 Convolution on Time Series

Before modeling, the filters and convolution operation in the context of multivariate time series is briefly explained.

A multivariate time series structure is illustrated in Figure 6.19a. The figure shows an illustrative example in which the x-, y-, and z-axis, show the time, the features, and the features' values, respectively[6].

The time series in the figure has three features with rectangular-, upward pulse-, and downward pulse-like movements. The features are placed along the depth which makes them the channels. A filter for such a time series is shown in Figure 6.19b.

The convolution operation between the filter and the time series is shown in Figure 6.20. As time series has only one spatial axis along time, the convolution sweeps it over time. At each stride, a similarity between the filter and a section of time series is emitted (not shown in the figure). The convolution variants, viz. padding, stride ($>$1), dilation, and 1×1, work similarly along the time axis.

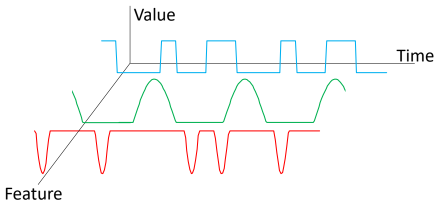
6.9.2 Imports and Data Preparation

Like always, the modeling starts with importing the required libraries, including the user-defined ones.

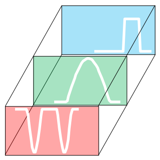
Listing 6.1. Imports for the convolutional network.

```python
import pandas as pd
import numpy as np

```

[6]The z-axis with the values should not be confused as a spatial axis as it appears in images. The axis for values of pixels in an image is hidden for simplicity. It can be imagined as an additional axis perpendicular to the image.



(a) *Multivariate time-series.*



(b) *Multivariate time-series filter.*

Figure 6.19. *An illustrative example of a multivariate time series (top) and a filter (bottom). The time series has a single spatial axis along time shown on the x-axis. A univariate time series has a single channel while the features in a multivariate time series become its channels shown on the y-axis. The value of the features with time is shown along the z-axis. This axis should not be confused as a spatial axis like in an image. A time series filter with the same number of channels, i.e., features, is shown in the bottom figure.*

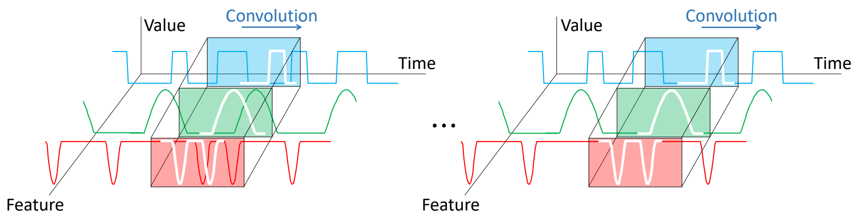


Figure 6.20. *Illustration of a convolution operation on a multivariate time series. Similar to the convolution in images, the filter is swept across the time series input. Since there is only one spatial axis along the time, the sweep happens over it. At each stride, the filter looks for a match between itself and the input. For example, a complete match is detected as the filter comes towards the right.*

```python
import tensorflow as tf
from tensorflow.keras import optimizers
from tensorflow.keras.models import Model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Conv1D
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPool1D
from tensorflow.keras.layers import AveragePooling1D
from tensorflow.keras.layers import MaxPool2D
from tensorflow.keras.layers import ReLU
from tensorflow.keras.layers import Flatten
from tensorflow.python.keras import backend as K

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

from collections import Counter

import matplotlib.pyplot as plt
import seaborn as sns

```

```python
# user-defined libraries
import utilities.datapreprocessing as dp
import utilities.performancemetrics as pm
import utilities.simpleplots as sp

from numpy.random import seed
seed(1)

SEED = 123  # used to help randomly select the data
    points
DATA_SPLIT_PCT = 0.2

from pylab import rcParams
rcParams['figure.figsize'] = 8, 6
plt.rcParams.update({'font.size': 22})
```

The tensor shape of a multivariate time series in a convolutional network is the same as in an LSTM network. The *temporalization* procedure discussed in § 5.4.2 is also applied here. The data preprocessing steps for converting categorical features to one-hot encoding and curve-shifting is in Listing 6.2.

Listing 6.2. Data pre-processing.

```python
df = pd.read_csv("data/processminer-sheet-break-rare
    -event-dataset.csv")
df.head(n=5)  # visualize the data.

# Hot encoding
hotencoding1 = pd.get_dummies(df['Grade&Bwt'])
hotencoding1 = hotencoding1.add_prefix('grade_')
hotencoding2 = pd.get_dummies(df['EventPress'])
hotencoding2 = hotencoding2.add_prefix('eventpress_'
    )

df = df.drop(['Grade&Bwt', 'EventPress'], axis=1)

df = pd.concat([df, hotencoding1, hotencoding2],
    axis=1)

# Rename response column name for ease of
    understanding
```

```
df = df.rename(columns={'SheetBreak': 'y'})

# Shift the response column y by 2 rows to do a 4-
   min ahead prediction.
df = dp.curve_shift(df, shift_by=-2)

# Sort by time and drop the time column.
df['DateTime'] = pd.to_datetime(df.DateTime)
df = df.sort_values(by='DateTime')
df = df.drop(['DateTime'], axis=1)

# Converts df to numpy array
input_X = df.loc[:, df.columns != 'y'].values
input_y = df['y'].values
```

6.9.3 Baseline

As usual, modeling starts with constructing a baseline model. The baseline models the temporal dependencies up to a lookback of 20, which is the same as the LSTM models in the previous chapter. The *temporalization* is done with this lookback in Listing 6.3. The resultant data in `X` is a `(samples, timesteps, features)` array.

Listing 6.3. Data temporalization and split

```
lookback = 20
X, y = dp.temporalize(X=input_X,
                      y=input_y,
                      lookback=lookback)

# Divide the data into train, valid, and test
X_train, X_test, y_train, y_test =
    train_test_split(np.array(X),
                     np.array(y),
                     test_size=DATA_SPLIT_PCT,
                     random_state=SEED)
X_train, X_valid, y_train, y_valid =
    train_test_split(X_train,
                     y_train,
                     test_size=DATA_SPLIT_PCT,
                     random_state=SEED)
```

```python

# Scaler using the training data.
scaler = StandardScaler().fit(dp.flatten(X_train))

X_train_scaled = dp.scale(X_train, scaler)
X_valid_scaled = dp.scale(X_valid, scaler)
X_test_scaled = dp.scale(X_test, scaler)

# Axes lengths
TIMESTEPS = X_train_scaled.shape[1]
N_FEATURES = X_train_scaled.shape[2]
```

The baseline is a simple network constructed with the `Sequential()` framework in Listing 6.9.3.

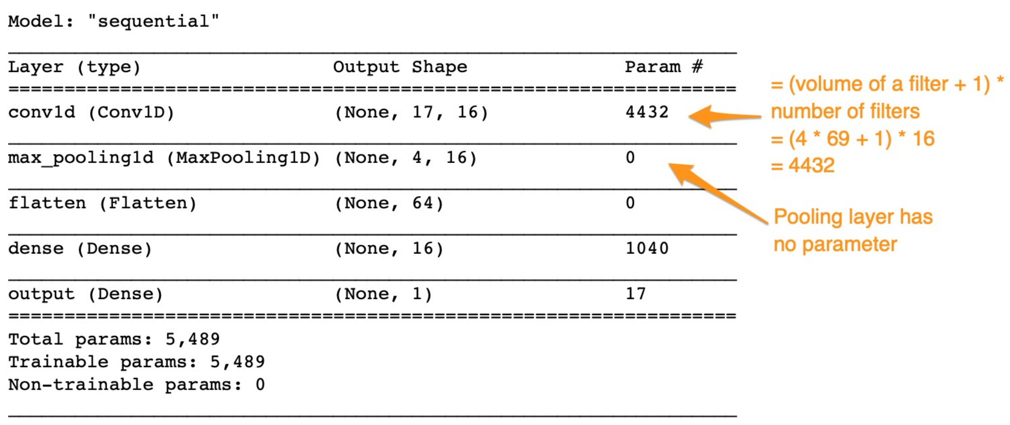
Listing 6.4. Baseline convolutional neural network

```python
model = Sequential()
model.add(Input(shape=(TIMESTEPS,
                       N_FEATURES),
                name='input'))
model.add(Conv1D(filters=16,
                 kernel_size=4,
                 activation='relu',
                 padding='valid'))
model.add(MaxPool1D(pool_size=4,
                    padding='valid'))
model.add(Flatten())
model.add(Dense(units=16,
                activation='relu'))
model.add(Dense(units=1,
                activation='sigmoid',
                name='output'))
model.summary()
```

The network's components are as follows.

Input layer

The shape of an input sample is defined in the `Input()` layer. An input sample here is a `(timesteps, n_features)` tensor due to the *temporalization* during the data processing.



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d (Conv1D)              (None, 17, 16)            4432
_________________________________________________________________
max_pooling1d (MaxPooling1D) (None, 4, 16)             0
_________________________________________________________________
flatten (Flatten)            (None, 64)                0
_________________________________________________________________
dense (Dense)                (None, 16)                1040
_________________________________________________________________
output (Dense)               (None, 1)                 17
=================================================================
Total params: 5,489
Trainable params: 5,489
Non-trainable params: 0
_________________________________________________________________
```

= (volume of a filter + 1) * number of filters
= (4 * 69 + 1) * 16
= 4432
Pooling layer has no parameter

Figure 6.21. *Baseline convolutional network summary.*

Conv layer

The network begins with a `Conv1D` layer. The `kernel_size` is set as `4` and the number of `filters` is `16`. Therefore, there will be `16` convolutional filters each being a `(4, n_features)` tensor.

Here `n_features=69` and, therefore, each filter will have 4×69 weights plus 1 bias parameters. Combining this for all the 16 filters, the convolutional layer contains $(4 * 69 + 1) * 16 = 4,432$ trainable parameters.

The output size of the layer along the spatial axis can be computed using Equation 6.10 as $\boldsymbol{o} = \left\lfloor \frac{(20)-(4)-((3)-1)((1)-1)+2*(0)}{(1)} \right\rfloor + 1 = (17)$. The output size along the channels will be equal to the number of filters in the convolutional layer, i.e., 16. Therefore, the output shape is `(None, 17, 16)` as shown in the model summary in Figure 6.21. Here, `None` corresponds to the `batch_size`.

The number of parameters is computed as per § 6.8.5 as $(4*69+1)*16 = 4,432$, where 4 is the kernel size and 69 is the number of channels, and 16 is the number of filters in the layer.

The `activation` in the `Conv` layer is set to `relu`. Similar to other networks, `relu` is a good choice in a baseline model.

Pooling layer

Max-pooling is one of the most popular pooling. The reason behind its popularity is explained in 6.12.1. `MaxPool`, thus, becomes an obvious choice for the baseline.

Similar to convolutional layers, a pooling layer is chosen from `MaxPool1D`, `MaxPool2D`, and `MaxPool3D` based on the number of spatial axes in its input. Here the output of the `Conv` layer which is the input to the `MaxPool` layer has one spatial axis and, therefore, `MaxPool1D` is used.

The `pool_size` is set to `4` which results in an output of size $\boldsymbol{o} = \left\lfloor \frac{(17)-(4)+2*(0)}{(4)} \right\rfloor + 1 = (4)$ along its (single) spatial axis. Also, the output will have the same number of channels as the number of filters in its input, i.e., 16. Therefore, as also shown in Figure 6.21, the output shape is `(None, 4, 16)` where `None` corresponds to the `batch_size`.

Besides, the pooling layer has no trainable parameters. A few pooling methods such as in Kobayashi 2019b have trainable parameters which can be considered for model improvement.

Flatten layer

This is merely an operational layer. It, naturally, has no trainable parameters. The flattening layer is required to create a feature vector out of the multi-axes tensor outputted from the `MaxPool` layer.

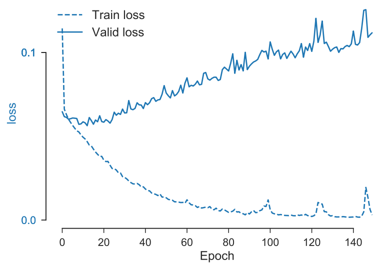
Dense layer

A penultimate `Dense` layer with `16` units is added to reduce the dimension of the features from `64` to `16` before passing them on to the final output dense layer. Since here we have a binary classification problem, the output dense layer requires a single unit with `sigmoid` activation.

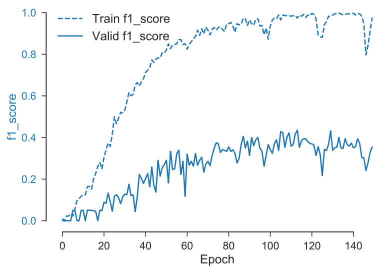
Model fitting and results

The model fit is performed similarly to the other networks discussed in the previous chapters.

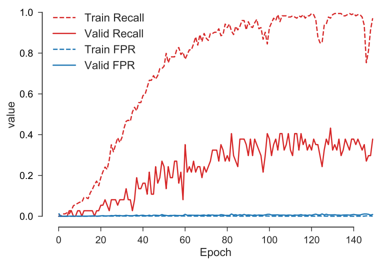
```
# Model fitting
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=[
                  'accuracy',
                  tf.keras.metrics.Recall(),
                  pm.F1Score(),
                  pm.FalsePositiveRate()
              ])
history = model.fit(x=X_train_scaled,
                    y=y_train,
                    batch_size=128,
                    epochs=150,
                    validation_data=(X_valid_scaled,
                        y_valid),
                    verbose=0).history
```



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure 6.22. *Baseline convolutional network results.*

The results are summarized in Figure 6.22a-6.22c. The validation recall and FPR are close to 0.4 and 0, respectively. Also, f1-score is close to 0.4. These performance results from the baseline convolutional network are already close to the best output in the previous chapter with LSTM models.

It is important to remind ourselves that the models developed here are not for comparison because the performances differ with problems. Instead, the essential finding here is that a convolutional network can work as well or possibly even better than a recurrent neural network (e.g., LSTM) in some temporal data. This means convolutional networks are capable of learning temporal patterns by treating them as spatial.

Convolutional networks have a strong potential to learn temporal patterns. They perform even better than recurrent neural networks in some problems.

6.9.4 Learn Longer-term Dependencies

Recurrent neural networks such as LSTMs are intended to learn long-term dependencies. The amount of the long-term dependencies can be increased by making the `lookback` higher during the data *temporalization.*

In principle, LSTMs should work with any short or long `lookback` because the gradient of the long-term dependencies cannot vanish (see § 5.2.8 in the previous chapter). But, in practice, it does not work because the learned dependencies can get smeared.

This phenomenon was reported in Jozefowicz, Zaremba, and Sutskever 2015. The smearing occurs because LSTMs and similar recurrent neural networks fuse the state information causing them to become confounded in the long-term.

A convolutional network, on the other hand, works like a filtration process with small filters. These filters sweep the input. They are agnostic to the input's size. That means, from a network's accuracy standpoint the filtration with convolution is largely indifferent to the `lookback`.

A network is constructed following this principle in Listing 6.5 where the `lookback=240` as opposed to 20 in the baseline.

Listing 6.5. Convolutional network with longer-term dependencies

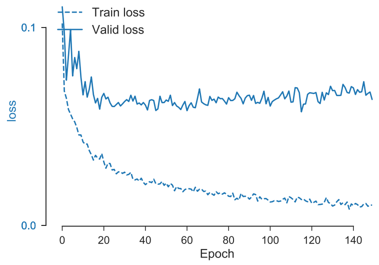
```
# Data Temporalize
lookback = 240  # Value 20 in baseline. Increased to
    40 and 240 to learn longer-term dependencies.
X, y = dp.temporalize(X=input_X,
                      y=input_y,
                      lookback=lookback)
X_train, X_test, y_train, y_test =
  train_test_split(np.array(X),
                   np.array(y),
                   test_size=DATA_SPLIT_PCT,
                   random_state=SEED)
X_train, X_valid, y_train, y_valid =
  train_test_split(X_train,
                   y_train,
                   test_size=DATA_SPLIT_PCT,
                   random_state=SEED)
# Initialize a scaler using the training data.
scaler = StandardScaler().fit(dp.flatten(X_train))

X_train_scaled = dp.scale(X_train, scaler)
X_valid_scaled = dp.scale(X_valid, scaler)
X_test_scaled = dp.scale(X_test, scaler)

TIMESTEPS = X_train_scaled.shape[1]
N_FEATURES = X_train_scaled.shape[2]

# Network construction
model = Sequential()
model.add(Input(shape=(TIMESTEPS,
                       N_FEATURES),
                name='input'))
model.add(Conv1D(filters=16,
                 kernel_size=4,
                 activation='relu'))
model.add(Dropout(0.5))
model.add(MaxPool1D(pool_size=4))
model.add(Flatten())
model.add(Dense(units=16,
                         activation='relu'))
model.add(Dense(units=1,
                         activation='sigmoid',
                         name='output'))
model.summary()

model.compile(optimizer='adam',
                      loss='binary_crossentropy',
                      metrics=[
                            'accuracy',
                            tf.keras.metrics.Recall(),
                            pm.F1Score(),
                            pm.FalsePositiveRate()
                      ])
history = model.fit(x=X_train_scaled,
                              y=y_train,
                              batch_size=128,
                              epochs=150,
                              validation_data=(X_valid_scaled,
                                                           y_valid),
                              verbose=0).history
```

The results are in Figure 6.23a-6.23c. The accuracy performance increased compared to the baseline. This implies longer-term dependencies exist in the problem.

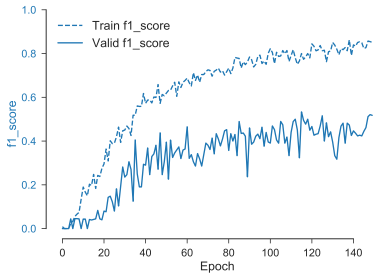
The presence of longer-term dependencies fits with the fact that the process settings in manufacturing processes such as paper manufacturing have lagged effects. For example, the density of pulp going into the machine impacts the paper strength a few hours later at the other end of the machine.

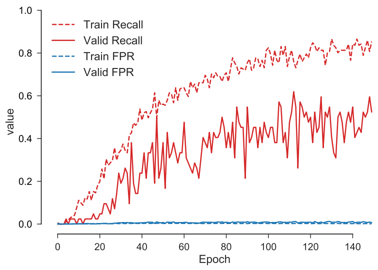
Convolutional networks can capture longer-term patterns in time series processes.

At this point, a question arises: why do not we make the `lookback` even longer? Although a longer `lookback` is usually better for accurate predictions, it significantly increases the computation because the convolutional feature map size rises dramatically. The increase in computation with `lookback` is not specific to convolutional networks. It is



(a) *Loss.*





(b) *F1-score.* (c) *Recall and FPR.*

Figure 6.23. *Results of a convolutional neural network from learning longer-term dependencies.*

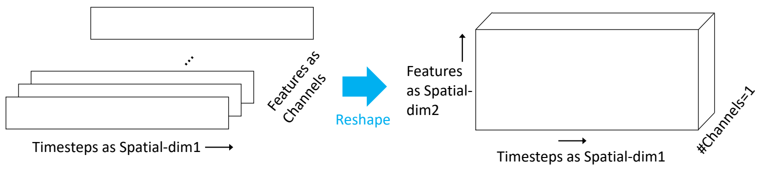


Figure 6.24. *A reshaped multivariate time series that resembles structure of an image with a single channel.*

a general problem in most deep learning networks.

6.10 Multivariate Time Series Modeled as Image

In the previous § 6.9, convolutional neural networks were constructed following the conventional approach of placing the temporal axis along a spatial axis and the multivariate features as channels.

Another approach is placing the features along a second spatial axis. This can be done by reshaping the input samples from `(timesteps, features)` to `(timesteps, features, 1)` tensors. The reshape is illustrated in Figure 6.24. The reshaped time series appears in the shape of an image with a single channel. Hence, this approach is termed as modeling a multivariate time series as an image.

In the reshaped time series sample, the `timesteps` and `features` are the spatial axes with one channel. Due to the two spatial axes, a convolutional network is constructed with `Conv2D`.

In the following, first, a `Conv2D` network equivalent to the baseline `Conv1D` network in § 6.9.3 is constructed to show their interchangeability in § 6.10.1. Thereafter, another network is constructed with `Conv2D` which is intended to learn the **local** temporal and spatial dependencies in and between the features in § 6.10.2.

6.10.1 Conv1D and Conv2D Equivalence

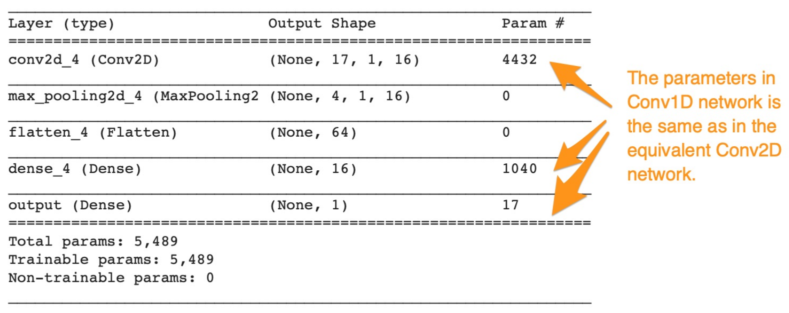
This section shows the equivalence between Conv1D and Conv2D by modeling the multivariate time series like an image.

In Listing 6.6, a convolutional network equivalent to the baseline network in § 6.9.3 is constructed using Conv2D. At its top, a lambda function to reshape the original samples is defined. It changes the X from (samples, timesteps, features) to (samples, timesteps, features, 1) tensor.

Listing 6.6. A Conv2D network equivalent to the baseline Conv1D network.

```
 1  # Equivalence of conv2d and conv1d
 2  def reshape4d(X):
 3    return X.reshape((X.shape[0],
 4                      X.shape[1],
 5                      X.shape[2],
 6                      1))
 7
 8  model = Sequential()
 9  model.add(Input(shape=(TIMESTEPS,
10                         N_FEATURES,
11                         1),
12                  name='input'))
13  model.add(Conv2D(filters=16,
14                   kernel_size=(4, N_FEATURES),
15                   activation='relu',
16                   data_format='channels_last'))
17  model.add(MaxPool2D(pool_size=(4, 1)))
18  model.add(Flatten())
19  model.add(Dense(units=16,
20                  activation='relu'))
21  model.add(Dense(units=1,
22                  activation='sigmoid',
23                  name='output'))
24  model.summary()
```

Thereafter, a network with Conv2D is defined. The network is made equivalent to the Conv1D network by setting the kernel_size in line 14 as (4, n_features). This kernel covers the entire feature axis which



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 17, 1, 16)         4432
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 4, 1, 16)          0
_________________________________________________________________
flatten_4 (Flatten)          (None, 64)                0
_________________________________________________________________
dense_4 (Dense)              (None, 16)                1040
_________________________________________________________________
output (Dense)               (None, 1)                 17
=================================================================
Total params: 5,489
Trainable params: 5,489
Non-trainable params: 0
_________________________________________________________________
```

The parameters in Conv1D network is the same as in the equivalent Conv2D network.

Figure 6.25. *Summary of a `Conv2D` equivalent to the baseline `Conv1D` network shows their interchangeability.*

makes the network the same as the `Conv1D` network.

They become the same because the features are the channels in the `Conv1D` network. Therefore, the `Conv` kernel spans all the channels. Setting the `Conv2D` kernel width equal to the number of features replicated this behavior. This is also confirmed by comparing the parameters in each layer shown in the model summary in Figure 6.25 with the baseline model summary in Figure 6.21. All of them are the same.

The result of the `Conv2D` network here is not shown as they resemble the baseline results in § 6.9.3.

The purpose of this section is to show the interchangeability of `Conv1D` and `Conv2D` in constructing convolutional networks. A practitioner can choose between either.

`Conv2D` networks provide more flexibility. Using it, one can construct models equivalent to a `Conv1D` network as well as try other architectures by treating a multivariate time series as an image. This is shown in the next section.

6.10.2 Neighborhood Model

A benefit of treating multivariate time series as an image for modeling is: a network can be constructed to learn the **local** temporal and spatial dependencies called a *neighborhood model*. These models require fewer convolutional parameters. One such network is developed in Listing 6.7.

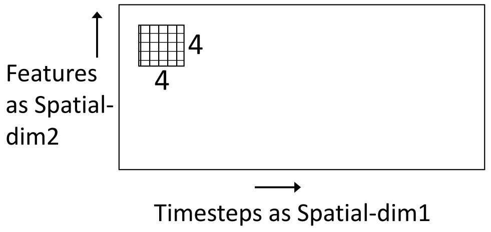
Listing 6.7. Neighborhood model for multivariate time series

```python
def reshape4d(X):
  return X.reshape((X.shape[0],
                    X.shape[1],
                    X.shape[2],
                    1))

# Neighborhood Model
model = Sequential()
model.add(Input(shape=(TIMESTEPS,
                       N_FEATURES,
                       1),
                name='input'))
model.add(Conv2D(filters=16,
                 kernel_size=(4, 4),
                 activation='relu',
                 data_format='channels_last',
                 name='Conv2d'))
model.add(MaxPool2D(pool_size=(4, 4),
                    name='MaxPool'))
model.add(Flatten(name='Flatten'))
model.add(Dense(units=16,
                activation='relu',
                name='Dense'))
model.add(Dense(units=1,
                activation='sigmoid',
                name='output'))
model.summary()
```

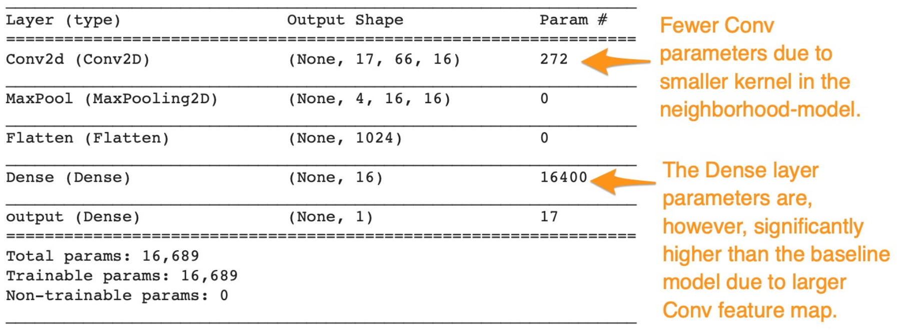
The `kernel_size` for the convolutional layer is set as `(4, 4)`. As visually illustrated in Figure 6.26a, the kernel can **only** learn the **local** dependencies within a 4×4 span and, hence, referred to as a *neighborhood model*.

Besides, as the reshaped time series has a single channel, the convolutional kernel size becomes `(4, 4, 1)` as opposed to `(4, n_features, 1)` in the previous § 6.10.1. Consequently, as shown in Figure 6.26b, the convolutional parameters reduce significantly.

However, due to the smaller kernel, the convolutional feature map becomes larger than in the baseline. This causes the penultimate dense layer parameters to increase.



(a) *A* 4×4 *kernel.* .



```
Layer (type)                 Output Shape              Param #
=================================================================
Conv2d (Conv2D)              (None, 17, 66, 16)        272
_________________________________________________________________
MaxPool (MaxPooling2D)       (None, 4, 16, 16)         0
_________________________________________________________________
Flatten (Flatten)            (None, 1024)              0
_________________________________________________________________
Dense (Dense)                (None, 16)                16400
_________________________________________________________________
output (Dense)               (None, 1)                 17
=================================================================
Total params: 16,689
Trainable params: 16,689
Non-trainable params: 0
_________________________________________________________________
```

Fewer Conv parameters due to smaller kernel in the neighborhood-model.

The Dense layer parameters are, however, significantly higher than the baseline model due to larger Conv feature map.

(b) *Neighborhood model summary.*

Figure 6.26. *In the top figure, the horizontal and vertical axes are time (i.e., the temporal dimension) and features (i.e., the spatial dimension), respectively. The* 4×4 *kernel spans both the axes and learns the local spatio-temporal dependencies. The constructed convolutional network summary (bottom) shows a significant reduction in the convolutional layer parameters compared to the baseline.*

A neighborhood model is expected to perform well if the interacting features are ordered or grouped, i.e., spatial dependencies are expected to be local. In such a scenario, it can have a similar performance as the baseline with fewer convolutional parameters that improve efficiency.

Besides, a neighborhood model also provides the flexibility to span a longer or the entire time-steps if long-term dependencies need to be learned. This is, otherwise, difficult if the time series is in its original shape as in the baseline model.

In sum, modeling a multivariate time series as an image brings more flexibility to construct efficient networks to learn short or wide spatial dependencies **and** short or long temporal dependencies based on the problem. The conventional modeling approach, on the other hand, only allows changing the length of the temporal dependencies.

6.11 Summary Statistics for Pooling

The strength of a convolutional network is its ability to simplify the feature extraction process. In this, pooling plays a critical role by weeding the extraneous information.

A pooling operation summarizes features into a *summary statistic*. It, therefore, relies on the statistic's efficiency. Whether the statistic preserves the relevant information or loses them depends on its efficiency.

What is an efficient summary statistic?

A summary statistic is a construct from *principles of data reduction* (Casella and Berger 2002). It is defined as,

> *a summary statistic summarizes a set of observations to preserve the* ***largest*** *amount of information as* ***succinctly*** *as possible.*

An efficient summary statistic is, therefore, one that concisely contains the most information of a sample. For example, the sample mean, or maximum. Other statistics, such as the sample skewness, or sample

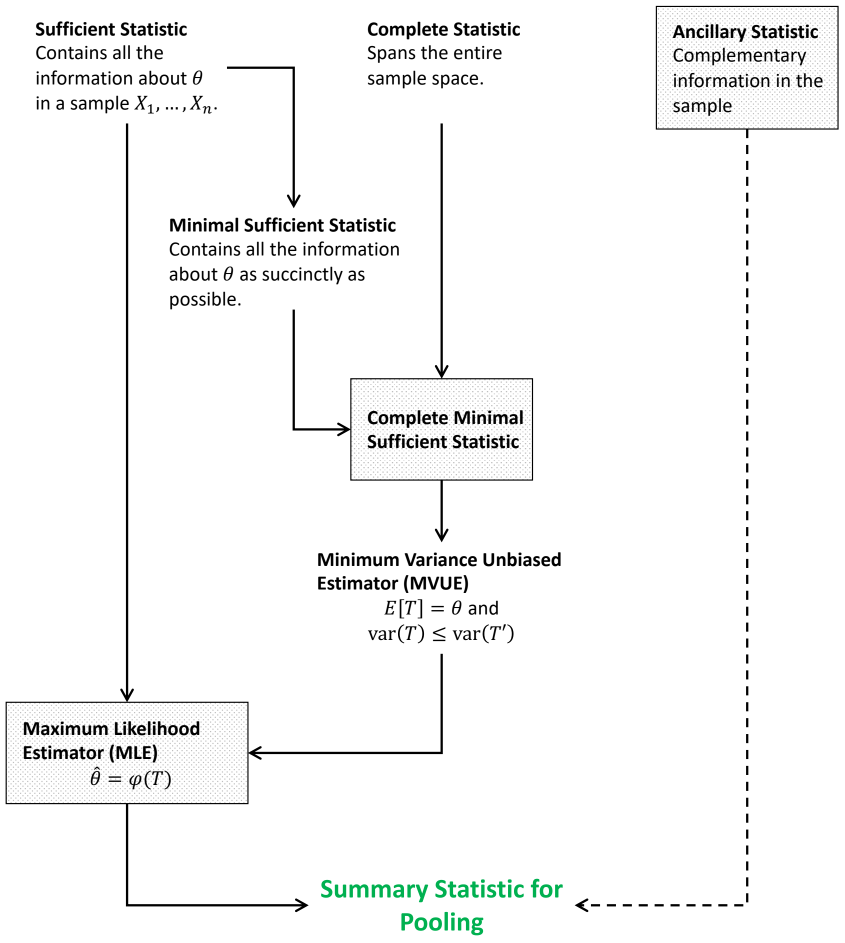

Sufficient Statistic
Contains all the information about θ in a sample $X_1, \ldots, X_n$.
Complete Statistic
Spans the entire sample space.
Ancillary Statistic
Complementary information in the sample
Minimal Sufficient Statistic
Contains all the information about θ as succinctly as possible.
Complete Minimal Sufficient Statistic
Minimum Variance Unbiased Estimator (MVUE)
$E[T] = \theta$ and
$\text{var}(T) \leq \text{var}(T')$
Maximum Likelihood Estimator (MLE)
$\hat{\theta} = \varphi(T)$
Summary Statistic for Pooling

Figure 6.27. *A summary statistic for pooling has roots in sufficient, complete, and ancillary statistics. A statistic that is both sufficient and complete provides a minimum variance unbiased estimator (MVUE). An MVUE's properties make it an efficient statistic. For some distributions, the maximum likelihood estimator (MLE), e.g., sample maximum, is an MVUE and, hence, becomes the best pooling statistic. Moreover, ancillary statistics, such as sample range, extracts information complementary to the MLE. They can be used as an additional pooling statistic (see Appendix J).*

size do not contain as much relevant information and, therefore, not efficient for pooling.

This section lays out the theory of summary statistics to learn about efficient statistics for pooling.

> "An experimenter might wish to summarize the information in a sample by determining a few key features of the sample values. This is usually done by computing (summary) statistics—functions of the sample."
>
> –Casella and Berger 2002.

Learning the dependence of pooling on the efficiency of summary statistics and the theory behind them is rewarding. It provides answers to questions like,

- Currently, max-pool and average-pool are the most common. Could there be other equally or more effective pooling statistics?
- Max-pool is found to be robust and, hence, better than others in most problems. What is the cause of max-pool's robustness?
- Can more than one pooling statistic be used together? If yes, how to find the best combination of statistics?

This section goes deeper into the theory of extracting meaningful features in the pooling layer. In doing so, the above questions are answered. Moreover, the theory behind summary statistics also provides an understanding of appropriately choosing a single or a set of statistics for pooling.

Pooling operation computes a summary statistic and its efficacy relies on the efficiency of the statistic.

An efficient summary statistic is one that contains the most information in as few values as possible, e.g., the sample mean and variance.

In the following, summary statistics applicable to pooling from three categories: (minimal) sufficient statistics, complete statistics, and ancillary statistics are explained. First, a few definitions are given in § 6.11.1. Then, sufficient statistics are shown to contain all the sample information in § 6.11.2. Next, § 6.11.3 shows complete statistics span the entire sample and a complete sufficient statistic is the minimum variance unbiased statistic (MVUE). It is further shown that a distribution's maximum likelihood estimator (MLE), e.g., average and maximum, is complete, sufficient, and MVUE.

This means, MLEs span the entire sample, contains all the information as succinctly as possible and is efficient. Hence, they make the best pooling statistic. Moreover, ancillary statistics such as sample range are shown to have complementary information in § 6.11.4 which can improve a network if used as an additional pooling statistic.

The findings in this section are used later in § 6.12 and 6.13 to uncover discoveries such as the reason behind max-pool's superiority, the effect of nonlinear activation on pooling, and the MLEs of common distributions for pooling.

6.11.1 Definitions

The feature map outputted by a convolutional layer is the input to a pooling layer. The feature map is a random variable $\boldsymbol{X} = \{X_1, \ldots, X_n\}$ where n is the feature map size[7].

An observation of the random variable is denoted as $\boldsymbol{x} = \{x_1, \ldots, x_n\}$. Describing properties of random variables is beyond the scope of this book but it suffices to know that their true underlying distribution and parameters are unknown[8].

The distribution function, i.e., the *pdf* or *pmf*[9], for the random variable $\boldsymbol{X}$ is denoted as f. The distribution has an underlying unknown parameter θ. The θ characterizes the observed $\boldsymbol{x}$ and, therefore, should

[7]The variables are denoted in block letters to denote they are random variables.

[8]Refer to Chapter 5 in Casella and Berger 2002 to learn the properties of random variables.

[9]Pdf or pmf refers to probability density function or probability mass function for continuous or discrete distributions, respectively.

be estimated.

A summary statistic of $f(\boldsymbol{X})$ is an estimate of θ. The statistic is a function of the random variable denoted as $T(\boldsymbol{X})$ and computed from the sample observations as $T(\boldsymbol{x})$. The sample mean, median, maximum, standard deviation, etc. are examples of the function T.

The goal is to determine T's that contain the most **information** of the feature map, achieve the most **data reduction**, and are the most **efficient**. These T's are the best choice for pooling in convolutional networks.

6.11.2 (Minimal) Sufficient Statistics

"A *sufficient* statistic for a distribution parameter θ is a statistic that, in a certain sense, captures all the information about θ contained in the sample."

–Casella and Berger 2002.

The concept of *sufficient* statistics lays down the foundation of data reduction by summary statistics. It is formally defined as follows.

Definition 1. *Sufficient Statistic*. *A statistic $T(\boldsymbol{X})$ is a sufficient statistic for θ if the sample conditional distribution $f(\boldsymbol{X}|T(\boldsymbol{X}))$ does not depend on θ.*

The definition can be interpreted as the conditional distribution of $\boldsymbol{X}$ given $T(\boldsymbol{X})$, i.e., $f(\boldsymbol{X}|T(\boldsymbol{X}))$, is independent of θ. This implies that in presence of the statistic $T(\boldsymbol{X})$ any remaining information in the underlying parameter θ is not required.

A sufficient statistic can replace the distribution parameter θ.

It is possible only if $T(\boldsymbol{X})$ contains all the information about θ available in $\boldsymbol{X}$. Therefore, $T(\boldsymbol{X})$ becomes a *sufficient* statistic to represent the sample in place of θ.

For example,

- **Mean**. The sample mean, $T(\boldsymbol{X}) = \bar{X} = \frac{\sum_i X_i}{n}$, is a sufficient statistic for a sample from a **normal** or **exponential** distribution.
- **Maximum**. The sample maximum, $T(\boldsymbol{X}) = X_{(n)}$, where $X_{(n)} = \max_i X_i, i = 1, \ldots, n$ is the n-th order statistic[10], is a sufficient statistic in a (truncated) **uniform** distribution or approximately in a **Weibull** distribution if its shape parameter is large.

The average-pool (`AvgPool`) and max-pool (`MaxPool`) indirectly originated from here. They are commonly used pooling methods. Between them, `MaxPool` is more popular. But, why? It is answered shortly in § 6.12.1. Before getting there, summary statistics are explored in the context of pooling.

Mean and maximum are sufficient statistics which indirectly led to the origin of ***`AvgPool`*** *and* ***`MaxPool`****.*

The *sufficiency* of a statistic is proved using the Factorization Theorem.

Theorem 1. ***Factorization Theorem****. A statistic $T(\boldsymbol{X})$ is sufficient if and only if functions $g(t|\theta)$ and $h(\boldsymbol{x})$ can be found such that $f(\boldsymbol{x}|\theta) = g(T(\boldsymbol{x})|\theta)h(\boldsymbol{x})$.*

The proofs for the sufficiency of the sample mean and maximum for normal and uniform distributions, respectively, are in Casella and Berger 2002 Chapter 6. However, it is worthwhile to look at sufficient statistics for a normal distribution to realize there are multiple sufficient statistics for a distribution.

Proposition 1. *If $X_1, \ldots, X_n$ are iid normal distributed $N(\mu, \sigma^2)$, the sample mean $\bar{x} = \frac{\sum_i^n x_i}{n}$ and sample variance $s^2 = \frac{\sum_i^n (x_i - \bar{x})^2}{(n-1)}$ are the sufficient statistics for μ and σ^2, respectively.*

Proof. The parameters for a normal distribution are $\theta = (\mu, \sigma^2)$. The joint pdf of the sample $\boldsymbol{X} = X_1, \ldots, X_n$ is,

[10] An order statistic denoted as $X_{(i)}$ is the i-th largest observation in a sample. Therefore, $X_{(n)}$ is the maximum of a sample.

$$\begin{aligned} f(\boldsymbol{x}|\mu,\sigma^2) &= \prod_i^n (2\pi\sigma^2)^{-1/2} \exp\big(-(x_i-\mu)^2/(2\sigma^2)\big) \\ &= (2\pi\sigma^2)^{-n/2} \exp\Big(-\sum_i^n (x_i-\mu)^2/(2\sigma^2)\Big) \end{aligned} \tag{6.12}$$

The pdf depends on the sample $\boldsymbol{x}$ through the two statistics $T_1(\boldsymbol{x}) = \bar{x}$ and $T_2(\boldsymbol{x}) = s^2$.

Thus, using the Factorization Theorem we can define $h(\boldsymbol{x}) = 1$ and

$$\begin{aligned} g(\boldsymbol{t}|\theta) &= g(t_1, t_2|\mu,\sigma^2) \\ &= (2\pi\sigma^2)^{-n/2} \exp\Big(-\big(n(t_1-\mu)^2 + (n-1)t_2\big)/(2\sigma^2)\Big) \end{aligned}$$

We can now express the pdf as

$$f(\boldsymbol{x}|\mu,\sigma^2) = g(T_1(\boldsymbol{x}), T_2(\boldsymbol{x})|\mu,\sigma^2)h(\boldsymbol{x}).$$

Hence, by Factorization Theorem, $T(\boldsymbol{X}) = (T_1(\boldsymbol{X}), T_2(\boldsymbol{X})) = (\bar{X}, S^2)$ is a sufficient statistic for (μ,σ^2) in this normal model. $\square$

Proposition 1 shows that a sample from normal distribution has more than one sufficient statistic, $\bar{x}$, and s^2. Similarly, a uniform distribution has the sample maximum $\max_i x_i$ and minimum $\min_i x_i$ as its sufficient statistics.

This tells that sufficient statistics are not unique in a distribution. There can be many. In fact, the entire ordered sample $T(\boldsymbol{X}) = \boldsymbol{X} = (X_{(1)}, \ldots, X_{(n)})$ is also a sufficient statistic.

Of course $T(\boldsymbol{X}) = \boldsymbol{X}$ is not much of a data reduction. But, out of the several sufficient statistics, which is better than the other?

The answer lies in the defined purpose of a summary statistic. The purpose is to achieve as much data reduction as possible without loss of information about the parameter θ.

Therefore, a sufficient statistic that achieves the most data reduction while retaining all the information about θ is preferable. Such a statistic is formally called a *minimal sufficient statistic.*

Definition 2. ***Minimal Sufficiency.*** *A sufficient statistic $T(\boldsymbol{X})$ is called a minimal sufficient statistic if for any other sufficient statistic $T'(\boldsymbol{X})$, $T(\boldsymbol{X})$ is a function of $T'(\boldsymbol{X})$, i.e., $T(\boldsymbol{X}) = f(T'(\boldsymbol{X}))$ for any $\boldsymbol{X} \in \mathcal{X}$.*

This can be interpreted as if any sufficient statistic $T'(\boldsymbol{X})$ can be reduced to $T(\boldsymbol{X})$, it means $T(\boldsymbol{X})$ provides more data reduction without losing information. For example, $T(\boldsymbol{X}) = \max_i X_i$ and $T'(\boldsymbol{X}) = X_1, \ldots, X_n$ are sufficient statistics where $T(\boldsymbol{X}) = \max_i X_i = \max T'(\boldsymbol{X})$.

Thus, $T(\boldsymbol{X})$ has the information of θ more succinctly than any other $T'(\boldsymbol{X})$. And, therefore, $T(\boldsymbol{X})$ becomes *minimally sufficient.*

Mathematically, minimal sufficiency can be proved using the following theorem.

Theorem 2. *Suppose there exists a statistic $T(\boldsymbol{x})$ such that for every two samples $\boldsymbol{x}$ and $\boldsymbol{y}$ the ratio of their pdfs $\frac{f(\boldsymbol{x}|\theta)}{f(\boldsymbol{y}|\theta)}$ is a constant independent of θ if and only if $T(\boldsymbol{x}) = T(\boldsymbol{y})$. Then $T(\boldsymbol{X})$ is a minimal sufficient statistic for θ.*

Using the theorem, the minimal sufficient statistics for $X \sim Normal$ and $X \sim Uniform$ are shown in the following propositions.

Proposition 2. *The sample mean $\bar{x} = \frac{\sum_i^n x_i}{n}$ and sample variance $s^2 = \frac{\sum_i^n (x_i - \bar{x})^2}{(n-1)}$ are the **minimal** sufficient statistics for μ and σ^2, respectively, if $X_1, \ldots, X_n$ are iid normal $N(\mu, \sigma^2)$.*

Proof. Suppose $\boldsymbol{x}$ and $\boldsymbol{y}$ are two samples, and their sample mean and variances are $(\bar{x}, s_{\boldsymbol{x}^2})$ and $(\bar{y}, s_{\boldsymbol{y}^2})$, respectively.

Using the pdf expression in Equation 6.12, the ratio of pdfs of $\boldsymbol{x}$ and $\boldsymbol{y}$ is,

$$
\begin{aligned}
\frac{f(\boldsymbol{x}|\mu,\sigma^2)}{f(\boldsymbol{y}|\mu,\sigma^2)} &= \frac{(2\pi\sigma^2)^{-n/2}\exp\Big(-\big(n(\bar{x}-\mu)^2+(n-1)s_{\boldsymbol{x}^2}\big)/(2\sigma^2)\Big)}{(2\pi\sigma^2)^{-n/2}\exp\Big(-\big(n(\bar{y}-\mu)^2+(n-1)s_{\boldsymbol{y}^2}\big)/(2\sigma^2)\Big)} \\
&= \exp\Big(\big(-n(\bar{x}^2-\bar{y}^2)+2n\mu(\bar{x}-\bar{y})-(n-1)(s_{\boldsymbol{x}^2-s_{\boldsymbol{y}^2}})\big)/(2\sigma^2)\Big).
\end{aligned}
$$

The ratio is a constant, i.e., independent of μ and σ, if and only if $\bar{x}=\bar{y}$ and $s_{\boldsymbol{x}^2=s_{\boldsymbol{y}^2}}$. Thus, by Theorem 2, $(\bar{(X)}, S^2)$ is minimal sufficient statistic for (μ,σ^2). $\square$

Proposition 3. *The sample maximum* $\max_i X_i$ *and minimum* $\min_i X_i$ *are the* ***minimal*** *sufficient statistics for* θ *if* $X_1,\ldots,X_n$ *are iid uniform in the interval* $(-\theta,\theta)$*, and* $-\infty<\theta<\infty$*.*

Proof. The joint pdf of $\boldsymbol{X}$ from $U(-\theta,\theta)$ is,

$$
\begin{aligned}
f(\boldsymbol{x}|\theta) &= \prod_i^n \frac{1}{2\theta}\mathbb{1}(|x_i|<\theta) \\
&= \frac{1}{(2\theta)^n}\mathbb{1}(\max_i x_i<\theta)\cdot\mathbb{1}(\min_i x_i>-\theta)
\end{aligned}
$$

Therefore, a ratio of pdfs of two samples $\boldsymbol{x}$ and $\boldsymbol{y}$ is,

$$
\frac{f(\boldsymbol{x}|\theta)}{f(\boldsymbol{y}|\theta)} = \frac{\mathbb{1}(\max_i x_i<\theta)\cdot\mathbb{1}(\min_i x_i>-\theta)}{\mathbb{1}(\max_i y_i<\theta)\cdot\mathbb{1}(\min_i y_i>-\theta)}
$$

The ratio is a constant independent of θ if and only if $\max_i x_i = \max_i y_i$ and $\min_i x_i = \min_i y_i$.

Therefore, by Theorem 2, $T(\boldsymbol{X}) = (\max_i X_i, \min_i X_i)$ is a minimal sufficient statistic for θ.

$\square$

In sum, sufficient statistics provide all information about a sample. However, there are many sufficient statistics and most of them do not result in data reduction. A minimal sufficient statistic, on the other hand, preserves the information and provides as much data reduction as possible. Therefore, among the several choices of sufficient statistics, *minimal sufficient statistic(s)* should be taken for pooling.

A minimal sufficient statistic such as mean and maximum has ***all*** *the information about underlying distribution parameter θ present in a feature map as succinctly as possible.*

Moreover, any one-to-one mapping of a minimal sufficient statistic is also a minimal sufficient statistic. This is important knowledge. Based on this, a pooling statistic can be scaled to stabilize a network without affecting the statistic's performance. For example, one should be pooling with $\sum_i X_i/n$ and $\sqrt{\sum_i (X_i - \bar{X})^2/n}$ instead of $\sum_i X_i$ and $\sum_i (X_i - \bar{X})^2/n$, respectively.

Identifying the best one-to-one mapping is, however, not always straightforward. The approach to finding the best-mapped statistic is formalized by connecting minimal sufficient statistics with the maximum likelihood estimator (MLE) through the theory of complete statistics in the next section.

6.11.3 Complete Statistics

The many choices with minimal sufficient statistics sometimes confuse a selection. This section introduces complete statistics which narrows the pooling statistic choice to only the **maximum likelihood estimator** of the feature map distribution.

A complete statistic is a bridge between minimal sufficient statistics and MLEs. MLEs derived from complete minimal statistics have the essential attributes of unbiasedness and minimum variance along with the minimality and completeness properties. MLEs, therefore, become the natural choice for pooling. Thereby, removing most of the ambiguity around pooling statistic selection.

In the following, these attributes and the path that leads to the relationship between complete minimal statistics and the MLE is laid out.

Definition 3. ***Completeness****. Let $f(t|\theta)$ be a family of pdfs or pmfs for a statistic $T(\boldsymbol{X})$. The family of probability distributions is called complete if for every measurable, real-valued function g, $E_\theta(g(T)) = 0$ for all $\theta \in \Omega$ implies $g(T) = 0$ with respect to θ, i.e., $P_\theta(g(T) = 0) = 1$ for all θ. The statistic T is boundedly complete if g is bounded.*

In simple words, it means a probability distribution is complete if the probability of a statistic $T(\boldsymbol{X})$ from an observed sample $\boldsymbol{X} = X_1, \ldots, X_n$ in the distribution is always non-zero.

This is clearer by considering a discrete case. In this case, completeness means $E_\theta(g(T)) = \sum g(T)P_\theta(T = t) = 0$ implies $g(T) = 0$ because by definition $P_\theta(T = t)$ is non-zero.

For example, suppose $X_1, \ldots, X_n$ is observed from a normal distribution $N(\mu, 1)$, and there is a statistic $T(\boldsymbol{X}) = \sum X_i$. Then, the $P_\mu(T(\boldsymbol{X}) = 0)$ is not equal to 0 for all μ. Therefore, $E_\mu(g(T)) = \int g(T)P_\mu(T) = 0$ implies $g(T) = 0$ for all μ. Therefore, $T = \sum X_i$ is complete.

This is an important property because it confirms that a statistic T, if complete, will span the whole sample space. Simply put, the statistic will contain *some* information from every observed sample X_i of the distribution for any parameter θ. And, therefore, the statistic is called **complete**.

A complete statistic contains some information about every observation from a distribution.

The importance of the *completeness* property is understood better by differentiating it with a sufficient statistic.

A minimal sufficient statistic contains **all** the information about θ, it does not necessarily **span** the whole sample space.

For example, suppose $X_1, \ldots, X_n$ is iid $Uniform(-\theta, \theta)$ then $T(\boldsymbol{X}) =$

$(X_{(1)}, X_{(n)})$, where $X_{(1)} = \min_i X_i$ and $X_{(n)} = \max_i X_i$ is a sufficient statistic. But it is **not** complete because $E(X_{(n)} - X_{(1)}) = c$, where c is a constant independent of θ. Therefore, we can define $g(T) = X_{(n)} - X_{(1)} - c$ but $E(X_{(n)} - X_{(1)} - c) = 0$ does not necessarily imply $X_{(n)} - X_{(1)} - c$ is always 0 because $E(X_{(n)} - X_{(1)}) \neq c$ for $\theta' \neq \theta$.

However, for the $Uniform$ distribution, $T = X_{(n)}$ is sufficient and complete. The proof is in § 6.2 in Casella and Berger 2002. It means $T = X_{(n)}$ spans the whole sample space.

For a normal distribution $N(\mu, \sigma^2)$, $T = (\sum_i X_i, \sum_i X_i^2)$ is both sufficient and complete. Meaning, the T has all the information about μ, σ^2 in a sample $X_1, \ldots, X_n$ as well as spans the whole sample space.

On a side note, a complete *statistic* is a misleading term. Instead of a statistic, completeness is a property of its family of distribution $f(t|\theta)$ (see Casella and Berger 2002 p.285). That means, when a statistic's distribution is complete it is called a *complete statistic*.

Next, the following Theorem 3 and 4 establish a relation between a complete statistic and a minimal sufficient statistic.

Theorem 3. ***Bahadur's theorem***[11]. *If T is a boundedly complete sufficient statistic and finite-dimensional, then it is minimal sufficient.*

A boundedly complete statistic in Theorem 3 implies the arbitrary function g in Definition 3 is bounded. This is a weak condition which is almost always true. Therefore, a complete sufficient statistic in most cases are also minimal.

The reverse, however, is always true as stated in Theorem 4.

Theorem 4. ***Complete Minimal Sufficient Statistic***[12]. *If a minimal sufficient statistic exists, then any complete sufficient statistic is also minimal.*

A complete minimal sufficient statistic has both **completeness** and **minimality** attributes. The statistic, therefore, **spans** the entire sample space, draws information from there, and yields **all** the information

[11]See Bahadur 1957.

[12]See § 6.2 in Casella and Berger 2002 and § 2.1 in Schervish 2012.

about the feature map distribution parameter θ as **succinctly** as possible. These attributes might appear enough, but are they?

They are not. Consider a complete minimal sufficient statistic $U = \sum_i X_i$ for a normally distributed feature map, $N(\mu, \sigma^2)$. Its expected value is $E(U) = n\mu$, which makes it biased. Using such a statistic in pooling can also make a convolutional network biased.

The biasedness in T is removed in $U = \frac{\sum_i X_i}{n}$. But there are other unbiased statistics as well, e.g., $U' = (X_{(1)} + X_{(n)})/2$. Which among them is better for pooling? The one with a smaller variance.

Compare the variances of U and U': $var(U) = \sigma^2/n$ and $var(U') = \sigma^2/2$. Clearly, $var(U) < var(U')$, if $n > 2$. It means if suppose U' is used in pooling, its value will swing significantly from sample to sample. This makes the network training and inferencing unstable. U, on the other hand, will have smaller variations that bring model stability.

Unbiasedness and small variation, therefore, in a pooling statistic makes a convolutional network efficient. This brings us to another type of statistic called minimum variance unbiased estimator (MVUE). It is defined as,

Definition 4. ***Minimum Variance Unbiased Estimator (MVUE).*** *A statistic T is a minimum variance unbiased estimator if T is unbiased, i.e., $E(T) = \theta$ and $var(T) \leq var(T')$ for all unbiased estimator T' and for all θ. Due to unbiasedness and small variance, it is statistically efficient.*

An MVUE is of particular interest due to its efficiency. Using an MVUE instead of any other statistic is analogous to using scaled input in a deep learning network. Just like an unscaled input, a biased and/or high variance pooling statistic makes the network unstable.

Identification of an MVUE provides an efficient statistic. Also, to our benefit, MVUE is unique. This is vital because the lookout for the best pooling statistic is over once the MVUE is found for a feature map. The uniqueness property is given in Theorem 5 below.

Theorem 5. ***MVUE is unique.*** *If a statistic T is a minimum variance unbiased estimator of θ then $T$is unique.*

Proof. Suppose T' is another MVUE $\neq T$. Since both T and T' are MVUE, their expectations will be θ, i.e., $E(T) = E(T') = \theta$, and the lowest variance denoted as δ, i.e., $var(T) = var(T') = \delta$.

We define an unbiased estimator combining T and T' as,

$$T^* = \frac{1}{2}(T + T').$$

For T^*, we have

$$E(T^*) = \theta$$

and,

$$\begin{aligned}
var(T^*) &= var\left(\frac{1}{2}T + \frac{1}{2}T'\right) \\
&= \frac{1}{4}var(T) + \frac{1}{4}var(T') + \frac{1}{2}cov(T, T') \\
&\leq \frac{1}{4}var(T) + \frac{1}{4}var(T') + \\
&\frac{1}{2}(var(T)var(T'))^{1/2} && \text{Cauchy-Schwarz inequality} \\
&= \delta && \text{As, } var(T) = var(T') = \delta.
\end{aligned}$$

But if the above inequality is strict, i.e., $var(T^*) < \delta$, then the minimality of δ is contradicted. So we must have equality for all θ.

Since the inequality is from Cauchy-Schwarz, we can have equality iff,

$$T' = a(\theta)T + b(\theta).$$

Therefore, the covariance between T and T' is,

$$\begin{aligned}
cov(T, T') &= cov(T, a(\theta)T + b(\theta)) \\
&= cov(T, a(\theta)T) \\
&= a(\theta)var(T)
\end{aligned}$$

but from the above equality $cov(T, T') = var(T)$, therefore, $a(\theta) = 1$. And, since

$$\begin{aligned} E(T') &= a(\theta)E(T) + b(\theta) \\ &= \theta + b(\theta) \end{aligned}$$

should be θ, $b(\theta) = 0$.

Hence, $T' = T$. Thus, T is unique.

$\square$

Due to the uniqueness and efficiency properties, the MVUE statistic is an ideal statistic in pooling and, therefore, should be identified. As shown in Figure 6.27, a complete minimal sufficient statistic leads to the MVUE as per Theorem 6 below.

Theorem 6. ***Lehmann-Scheffé***[13]. *Let T be a complete (minimal) sufficient statistic and there is any unbiased estimator U of θ. Then there exists a unique MVUE, which can be obtained by conditioning U on T as $T^* = E(U|T)$. The MVUE can also be characterized as a unique unbiased function $T^* = \varphi(T)$ of the complete sufficient statistic.*

A complete (minimal) sufficient statistic does not guarantee unbiasedness and low variance by itself. However, Theorem 6 tells that a one-to-one function of it is an MVUE and is, of course, complete and minimal.

Therefore, an MVUE statistic T^*, if found, has both efficiency and complete minimal statistic properties. These properties make it supreme for pooling. However, a question remains, how to find the T^*?

Theorem 7 leads us to the answer in Corollary 1.

Theorem 7. ***MLE is a function of sufficient statistic.*** *If T is a sufficient statistic for θ, then the maximum likelihood estimator (MLE) (if it exists) $\hat{\theta}$ is a function of T, i.e., $\hat{\theta} = \varphi(T)$.*

[13]See Lehmann and Scheffé 1950 and Lehmann and Scheffé 1955.

Proof. Based on the Factorization Theorem 1,

$$f(\boldsymbol{x}|\theta) = g(T(\boldsymbol{x}|\theta))h(\boldsymbol{x}).$$

An MLE is computed by finding the θ that maximizes the likelihood function $L(\theta|\boldsymbol{x}) = f(\boldsymbol{x}|\theta)$.

If MLE is unique then $h(\boldsymbol{x})$ is a constant or equal to 1 without loss of generality. Therefore,

$$\begin{aligned}\hat{\theta} &= \arg\max_{\theta} f(\boldsymbol{x}|\theta)\\ &= \arg\max_{\theta} g(T(\boldsymbol{x}|\theta))\end{aligned}$$

which is clearly a function of the sufficient statistic $T(\boldsymbol{X})$. □

Corollary 1. *If MLE $\hat{\theta}$ is unbiased and a complete minimal sufficient statistic T exist for parameter θ, then $\hat{\theta} = \varphi(T)$, and it is the unique minimum variance unbiased estimator (MVUE). Thereby, the statistic $\hat{\theta}$ contains all the information about the parameter θ, spans the whole sample space, and is efficient.*

Proof. Based on Theorem 7, the maximum likelihood estimator (MLE) $\hat{\theta}$ is a function of a sufficient statistic T, i.e., $\hat{\theta} = \varphi(T)$.

If T is complete sufficient and $\hat{\theta}$ is unbiased, then based on Theorem 6, $\hat{\theta} = \varphi(T)$ and is an MVUE. Therefore, being a function of complete statistic, $\hat{\theta}$ spans the whole sample space. Also, it is efficient based on the definition of MVUE in Definition 4.

Based on Theorem 3 and Theorem 4 a complete statistic in most cases is minimal, and if a minimal statistic exist then any complete statistic is always minimal. Therefore, if T is complete minimal, $\varphi(T)$ is also complete minimal and, therefore, $\hat{\theta} = \varphi(T)$ will have all the information about θ as succinctly as possible.

Finally, as per Theorem 5 an MVUE is unique. Therefore, $\hat{\theta}$ will be unique. □

At this stage, Theorem 3-7 come together to forge a spectacular relationship in Corollary 1. This is a pivotal result because a maximum likelihood estimator is available for most of the distributions applicable to the convolutional feature maps inputted to pooling.

MLE's properties elucidated in Corollary 1 make it an obvious choice in pooling. It is shown in § 6.13 that *average* and *maximum* pooling statistics are indeed MLEs. In fact, a parameterization combining the average and maximum in Boureau, Ponce, and LeCun 2010 to obtain a pooling statistic between the two is shown to be the MLE for Weibull distribution.

6.11.4 Ancillary Statistics

A (complete) minimal sufficient statistic itself or in the form of MLE retains all the information about the parameter θ. It eliminates all the extraneous information in the sample and takes only the piece of information related with θ.

Therefore, it might be suspected that no more information remains to draw from the sample. Except that there is more.

There is still information remaining in the sample that is independent of θ. For example, sample range or interquartile range. Such statistics are called *ancillary statistics*.

Ancillary statistics, defined below, have information complementary to minimal sufficient statistics. They can, therefore, act as a strong companion to a minimal sufficient based MLE statistic in pooling.

Definition 5. ***Ancillary Statistic.*** *A statistic $S(\boldsymbol{X})$ whose distribution does not depend on the parameter θ is called an ancillary statistic.*

As mentioned above, the sample range or interquartile range are examples of ancillary statistics. Both are special cases of a range statistic $R = X_{(k)} - X_{(l)}$, where $X_{(k)}$ and $X_{(l)}$ are order statistics with $k, l \in \{1, \ldots, n\}$. Proposition 4 shows that a range statistic is an ancillary statistic for any location model, e.g., normal and uniform distribution.

Proposition 4. *If $X_1, \ldots, X_n$ are iid observations from a location model where cdf is denoted as $F(x - \theta), -\infty < \theta < \infty$, e.g., Uniform and*

Normal, then any range statistic $R = X_{(k)} - X_{(l)}$ is an ancillary statistic, where $X_{(k)}$ and $X_{(l)}$ are order statistics with $k, l \in \{1, \ldots, n\}$. The sample range $R = X_{(n)} - X_{(1)} = \max_i X_i - \min_i X_i$ and inter-quartile range $R = Q_3 - Q_1$ are special cases of a range.

Proof. We have $X \sim F(X - \theta)$. We replace X with Z such that $X = Z + \theta$. Thus, the cdf of a range statistic $R = X_{(k)} - X_{(l)}$ becomes,

$$\begin{aligned}
F_R(r|\theta) &= P_\theta(R \leq r) \\
&= P_\theta(X_{(k)} - X_{(l)} \leq r) \\
&= P_\theta((Z_{(k)} + \theta) - (Z_{(l)} + \theta) \leq r) \\
&= P_\theta(Z_{(k)} - Z_{(l)} + \theta - \theta \leq r) \\
&= P_\theta(Z_{(k)} - Z_{(l)} \leq r)
\end{aligned}$$

The cdf of R, therefore, does not depend on θ. Hence, the range statistic R is an ancillary statistic. $\square$

As per the proposition, a range statistic can be used in conjunction with any other minimal sufficient statistic in pooling. However, the combination should be chosen carefully. They are sometimes dependent. For example, in a Uniform model, the minimal sufficient statistics are $T_1(\boldsymbol{X}) = \max_i X_i$ and $T_2(\boldsymbol{X}) = \min_i X_i$, and an ancillary statistic is $S(\boldsymbol{X}) = \max_i X_i - \min_i X_i$. Clearly, $S(\boldsymbol{X})$ is a function, and, thereof dependent, of $T_1(\boldsymbol{X})$ and $T_2(\boldsymbol{X})$.

A complete minimal statistic, however, is independent of any ancillary statistic as per Theorem 8.

Theorem 8. ***Basu's Theorem***[14]. *If $T(\boldsymbol{X})$ is complete and minimal sufficient statistic, then $T(\boldsymbol{X})$ is independent of every ancillary statistic $S(\boldsymbol{X})$.*

Therefore, a minimum variance unbiased MLE based off of a complete minimal statistic is always independent of an ancillary statistic.

[14] See Basu 1955.

This property reinforces the support for using MLE as the primary pooling statistic. And, if needed, an ancillary statistic can be directly included due to their mutual independence. For illustration, Appendix J develops a Convolutional Network with *maximum* (MLE) pool and *sample range* (ancillary statistic) pool put together in parallel.

🔔 *The ancillary statistic in pooling can draw additional relevant information from the convolutional feature map to improve a network's performance.*

6.12 Pooling Discoveries

A convolutional network mainly comprises of three operations of convolution, activation, and pooling. Among them, pooling plays a key role in *extracting the essence from the excess* to improve the computational and statistical efficiencies.

There are a variety of pooling statistics developed over the years and discussed in § 6.15. Despite the variety, max-pooling remains popular due to its superior performance in most data sets.

The upcoming § 6.12.1 puts forward a plausible reason behind max-pool's superiority. The expounded reasoning also uncovers an inherent fallacy of distribution distortion in the *convolution* $\rightarrow$ *activation* $\rightarrow$ *pooling* structure in traditional networks.

Remedial architectures from Ranjan 2020 to address the fallacy by preserving the features map distribution is in § 6.12.2. The distribution preservation leads to presenting maximum likelihood estimators (MLEs) for pooling in § 6.13 (based on Corollary 1).

The section also shows a unifying theory behind max- and average-pooling, and their combination as mixed pooling in the form of MLE statistic of a Weibull distribution. Thereafter, a few advanced pooling techniques based off of summary statistics to adopt adaptive pooling and address spatial relationships are laid in § 6.14.

Lastly, the history of pooling discussed in § 6.15 ties together the literature with the rest of this section.

6.12.1 Reason behind Max-Pool Superiority

The realization of max-pooling importance traces back to biological research in Riesenhuber and Poggio 1998. Riesenhuber and Poggio 1999 provided a biological explanation of max-pool superiority over average. Popular work by deep learning researchers have also advocated for max-pooling in Yang et al. 2009; Boureau, Ponce, and LeCun 2010; Saeedan et al. 2018.

Yang et al. 2009 reported significantly better classification performance on several object classification benchmarks using max-pooling compared to others.

A theoretical justification was provided in Boureau, Ponce, and LeCun 2010. They provided a theory supporting max-pool assuming the input to pooling as Bernoulli random variables. But the Bernoulli assumption is an extreme simplification. A traditional features map input to pooling is a continuous random variable while Bernoulli is discrete.

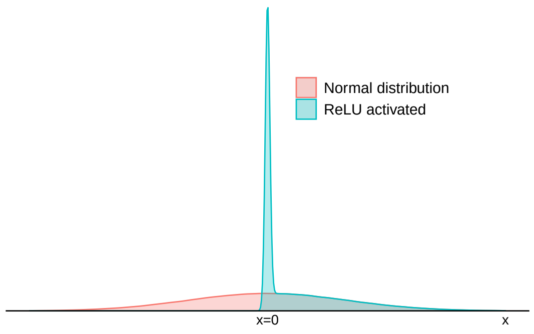
The reason for max-pool's superiority lies in the understanding distribution of feature maps, or rather the distorted distribution and its MLE statistic.

A feature map is a continuous random variable. A random variable follows a distribution. The MLE of the distribution, if known, is the best pooling statistic. The question is, what is the distribution of the feature map?

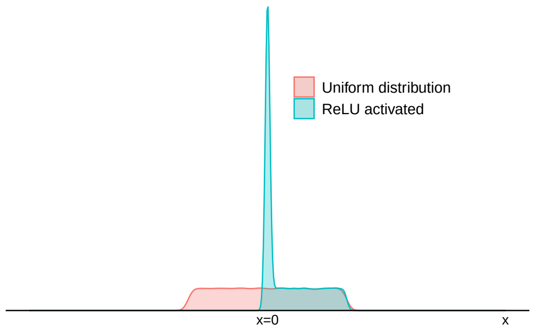
It depends on the problem. However, determining the distribution is not difficult. A bigger issue is that a feature map's distribution is already distorted before pooling. This is caused by a nonlinear activation of convolution output.

A nonlinear activation **distorts** the original distribution. Fitting known distributions on them become difficult. This is illustrated in Figure 6.28a. The figure shows the effect of ReLU activation on a normally distributed feature map.

As shown in the figure, the original distribution warps due to ReLU activation. It becomes severely skewed and does not follow a known



(a) *Normal Distribution before and after ReLU activation.*



(b) *Uniform Distribution before and after ReLU activation.*

Figure 6.28. *The feature map outputted from a convolutional network typically follows a distribution. The true distribution is, however, distorted by a nonlinear activation. For example, ReLU activated normal and uniform distribution shown here are severely skewed. Due to this, the feature map becomes biased. Therefore, most summary statistics other than the maximum becomes unusable for pooling. For example, the sample average yields an overestimate of the mean, the minimum statistic remains zero irrespective of the true smallest value, and a range statistic becomes =(maximum − zero). In effect, a nonlinear activation before pooling restricts its ambit, i.e., only a few summary statistics in pooling remain usable.*

distribution. If it is still assumed as a normal distribution, the sample mean (the MLE) will be an overestimate of the true mean. The overestimation undermines the average statistic for pooling. Similarly, other statistics, if used, in pooling such as the sample variance (an ancillary statistic[15]) will be underestimated.

The average statistic is overestimated under the normality assumption due to the distortion. This explains the reason behind average-pooling unfitness in some problems.

How does the maximum pooling remain effective in presence of the distortion? There are two plausible reasons based on distribution assumptions described below.

- **Uniform distribution**. The maximum statistic is the MLE of a uniform distribution. As shown in Figure 6.28b, the distortion does not affect the sample maximum. Although an activated feature map is no longer uniform if it was originally uniformly distributed the maximum statistic remains undisturbed for pooling.

- **Weibull distribution**. An activated feature map can be fitted with a Weibull distribution. It is quite a flexible distribution. It has various shapes for different values of its scale λ and shape k parameters. A few illustrations for different (λ, k) are shown in Figure 6.34 in § 6.13.4. The section also presents Weibull's MLE in Equation 6.25 which becomes equal to the sample maximum for large k.

Under these conditions, the sample maximum becomes the best pooling statistic. Perhaps, most of the problems are close to one of them and, hence, max-pooling is popular.

Max-pool is robust to the distortions in feature map distribution caused by nonlinear activation. Therefore, it works better than other types of pooling in most problems.

[15]Sample variance is an ancillary statistic as well as the MLE of normal distribution's variance parameter

The above reasons are conjectures. The exact theoretical reasoning behind max-pool's superiority is still elusive. And, as the theory behind pooling evolves, a better pooling statistic backed with its theoretical efficiency might be discovered.

6.12.2 Preserve Convolution Distribution

The premise of max-pool's superiority also uncovered a traditional convolutional network fault: convolution feature map distribution distortion before pooling. Distortion of feature map distribution is detrimental to pooling because the information is lost or masked.

This fault is present due to the structure *convolution → activation → pooling* in traditional networks. The structure follows the deep learning paradigm of nonlinear activation following a trainable layer.

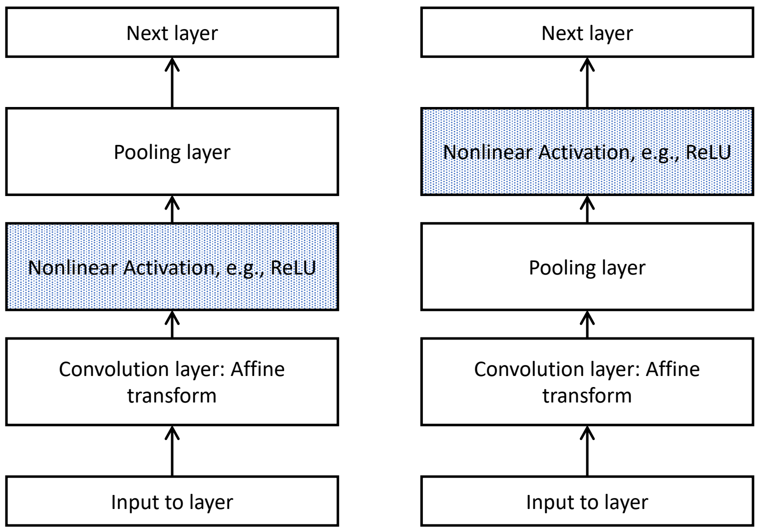
Nonlinear activation on convolutional layer output before pooling is an architectural fault.

But following this paradigm is not necessary for convolutional networks. A pooling layer is not trainable[16]. Therefore, nonlinear activation before or after pooling does not affect the overall nonlinearity of a convolutional network. Thus, their positions can be swapped to follow a structure *convolution → pooling → activation* to preserve the feature map's distribution while not impacting the network's nonlinearity.

Ranjan 2020 made the above argument and proposed the swapping. Swapping the order of pooling and nonlinear activation remedies the distribution distortion fallacy. The difference between the traditional and a swapped architecture is shown in Figure 6.29a and 6.29b.

Due to the swapping, the information derived in the convolution operation remains intact for pooling. Importantly, the convolutional feature map's distribution remains undisturbed. This brings a few major improvements,

[16]A pooling layer is fundamentally not trainable. However, some trainable pooling methods are proposed by researchers, e.g., Kobayashi 2019a.


Next layer
Pooling layer
Nonlinear Activation, e.g., ReLU
Convolution layer: Affine transform
Input to layer
Next layer
Nonlinear Activation, e.g., ReLU
Pooling layer
Convolution layer: Affine transform
Input to layer

(a) *Traditional Convolutional Network.*

(b) *Swapped Pooling-Activation Network.*

Figure 6.29. *A traditional convolutional network (left) has a nonlinear activation, e.g., ReLU, before the pooling layer. The nonlinear activation distorts the distribution of the feature map outputted by the convolutional layer. Due to this, only a maximum summary statistic in MaxPool works in most problems. Swapping the order of pooling and activation (right) remedies the issue of distortion. In this network, the feature map's distribution is undisturbed which allows the use of a variety of summary statistics, including combinations of sufficient and ancillary statistics. For example, (mean, standard deviation), or (maximum, range).*

- **MLE statistic based on distribution**. As per Corollary 1 in § 6.11.3, the maximum likelihood estimator (MLE) makes the *best* statistic for pooling. Feature maps typically follow distributions from known families. If undisturbed, their MLEs can be used in pooling.
- **Usability of ancillary statistics**. A variety of ancillary statistics such as standard deviation, range, and IQR, become informative if the distribution is preserved. Moreover, a combination of MLE and ancillary statistics can also be pooled.

A swapped pooling and nonlinear activation architecture in the convolutional network allows the use of MLEs and ancillary statistics in pooling.

Both improvements have far-reaching effects. Especially, the possibility of pooling MLEs discussed in detail next.

6.13 Maximum Likelihood Estimators for Pooling

A feature map follows a distribution. The distribution differs with samples. For example, an object with sharp edges at the center of an image will have a different feature map distribution compared to an object with smudgy edges or located at a corner (shown in Figure 6.35a and 6.35b in § 6.14).

Regardless, the distribution's maximum likelihood estimator (MLE) makes the most efficient pooling statistic as described in § 6.11. A few distributions that feature maps typically follow and their MLEs are, therefore, given below.

Besides, a profound theoretical support for the parametric combination of the sample mean and maximum proposed by Boureau, Ponce, and LeCun 2010 as the optimal choice for pooling is given in § 6.13.4. The section shows that Boureau, Ponce, and LeCun 2010's parameterization is the MLE of a Weibull distribution.

6.13.1 Uniform Distribution

A uniform distribution belongs to the symmetric location probability distribution family. A uniform distribution describes a process where the random variable has an arbitrary outcome in a boundary denoted as (α, β) with the same probability. Its pdf is,

$$f(x) = \begin{cases} \dfrac{1}{\beta - \alpha} & \text{, if } \alpha < x < \beta \\ 0 & \text{, otherwise} \end{cases} \tag{6.13}$$

Different shapes of the uniform distribution are shown in Figure 6.30 as examples. Feature maps can follow a uniform distribution under some circumstances, such as if the object of interest is scattered in an image.

However, uniform distributions relevance lies in it being the maximum entropy probability distribution for a random variable. This implies, if nothing is known about the distribution except that the feature map is within a certain boundary (unknown limits) and it belongs to a certain class then the uniform distribution is an appropriate choice.

Besides, the maximum likelihood estimator of uniform distribution is,

$$\hat{\beta} = \max_i X_i \tag{6.14}$$

Therefore, if the feature map is uniformly distributed or distribution is unknown, $\max_i X_i$ is the *best* pooling statistic. The latter claim also reaffirms the reasoning behind max-pools superiority in § 6.12.1.

6.13.2 Normal Distribution

A normal distribution, a.k.a. Gaussian, is a continuous distribution from the exponential location family. It is characterized by its mean μ and standard deviation σ parameters. Its pdf is defined below and

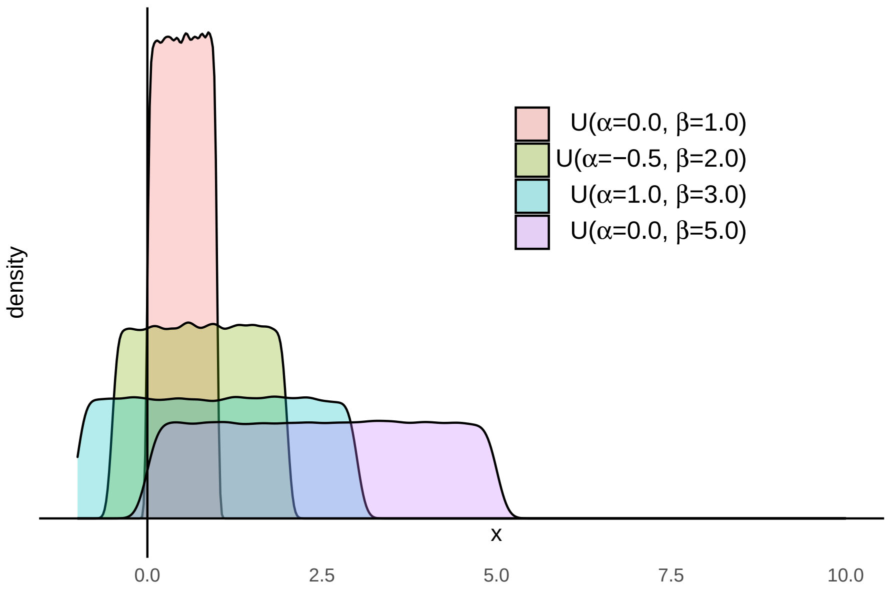


Figure 6.30. *In a uniform distribution, $X \sim U(\alpha, \beta)$, $X \in (\alpha, \beta)$, $\alpha, \beta \in \mathbb{R}$, and $\beta > \alpha$, the probability of the feature X having any value in (α, β) is equal $(= \frac{1}{\beta - \alpha})$ and zero, otherwise. It is also maximum entropy probability distribution, which implies if nothing is known about a feature map's distribution except that it is bounded and belongs to a certain class then uniform distribution is a reasonable default choice. The maximum likelihood estimate (MLE) of a uniform distribution is the maximum order statistic, i.e., $\hat{\beta} = X_{(n)} = \max_i X_i$.*

Figure 6.31 shows a few examples.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\Big(- \frac{(x-\mu)^2}{2\sigma^2}\Big) \tag{6.15}$$

The MLEs of the normal distribution are

$$\hat{\mu} = \frac{\sum_i X_i}{n}, \tag{6.16}$$

$$\hat{\sigma}^2 = \frac{\sum_i (X_i - \bar{X})^2}{n-1}. \tag{6.17}$$

A normal distribution supports $-\infty < x < \infty$, i.e., $x \in \mathbb{R}$ and is symmetric. But most nonlinear activated feature map either distorts the symmetry or bounds it, e.g., ReLU lower bounds the feature map at 0.

Due to the distortion, activated feature maps are unlikely to follow a normal distribution. Therefore, a restructured *convolution → pooling → activation* architecture described in § 6.12.2 becomes favorable.

A normal distribution is a plausible distribution for most data samples barring some special cases such as if the object in an image is at the corners. Besides, normality offers two MLE statistics for pooling that provide both signal and spread information.

6.13.3 Gamma Distribution

A gamma distribution is an asymmetrical distribution also from the exponential family. It has two parameters: shape k and scale θ. They characterize the lopsidedness and spread of the distribution.

Its pdf is,

$$f(x) = \frac{1}{\Gamma(k)\theta^k} x^{k-1} \exp -\frac{x}{\theta} \tag{6.18}$$

where, $x > 0$, and $k, \theta > 0$.

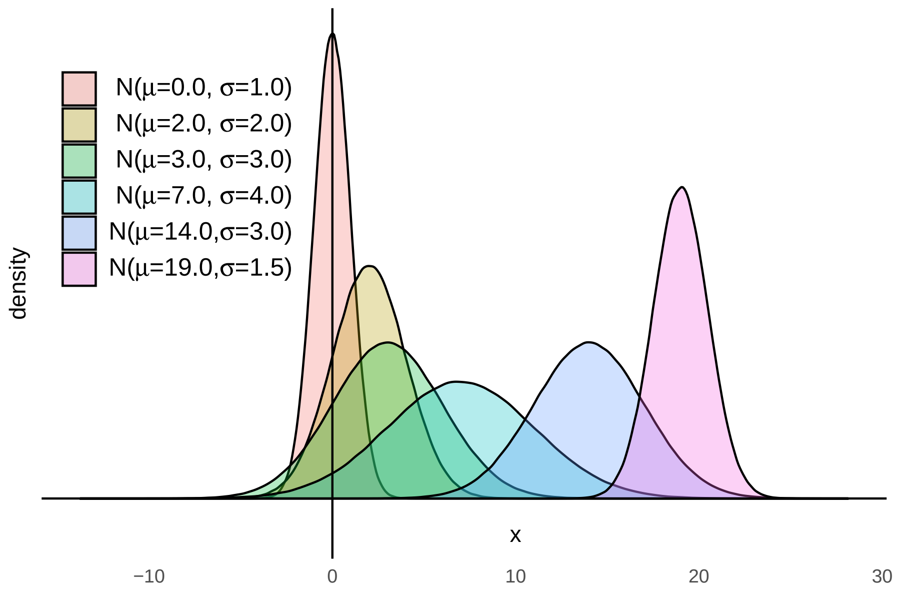


Figure 6.31. *A normal distribution, $X \sim N(\mu, \sigma)$, $X \in \mathbb{R}$, $\mu \in \mathbb{R}, \sigma \in \mathbb{R}^+$, is the symmetric distribution from the exponential family. A convolutional feature map follows a normal distribution if and only if it can take any value in $(-\infty, \infty)$ and the probability $P(X = x)$ is symmetric. That is, a nonlinear activated convolutional output is unlikely to be normal. A linearly activated (or inactivated) convolutional output can be assumed to be normal in many data sets. The center μ and the spread σ differs by the data. The MLEs for them are, $\hat{\mu} = \bar{X} = \frac{\sum_i X_i}{n}$ and $\hat{\sigma}^2 = S^2 = \frac{\sum_i (X_i - \bar{X})^2}{(n-1)}$.*

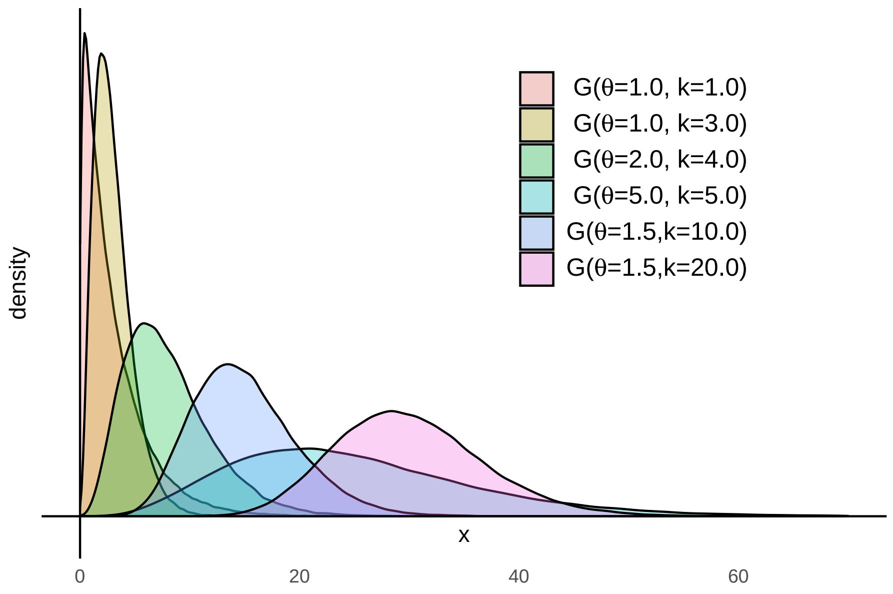


Figure 6.32. *A gamma distribution, $X \sim G(k, \theta)$, $X > 0$, and $k, \theta > 0$, is one of the asymmetric distribution from the exponential family. Gamma distribution is applicable on an activated convolutional output x such that $f(x) > 0$ or constrained convolution where $x > 0$. Gamma distribution provides more flexibility due to its asymmetry. The exponential distribution is a special case of Gamma distribution if $k = 1$, shown with $G(1, 1)$ above. This is an extreme case found when the object is partially present or at corners of the input. In other cases, the shape and scale differ with data and are characterized by k and θ, respectively. Closed-form MLEs do not exist for Gamma, however, mixed type log-moment estimators exist that have similar efficiency as MLEs presented as,*

$\hat{\theta} = \frac{1}{n(n-1)}\left(n\sum_i x_i \log(x_i) - \sum_i \log(x_i)\sum_i x_i\right)$ *and*

$\hat{k} = \frac{n\sum_i x_i}{n\sum_i x_i \log(x_i) - \sum_i \log(x_i)\sum_i x_i}$.

The distribution can take a variety of forms as shown in Figure 6.32. In fact, like uniform distribution, the gamma distribution is also the maximum entropy probability distribution. This makes gamma distribution applicable to feature maps in most data samples.

However, its support is $x > 0$ which constrains its applicability only to positive-valued feature maps. There are several ways to constrain the feature maps, e.g., using a positive activation such as ReLU (or Null-ReLU defined in Appendix G), or with kernel and bias constraints set to non-negative in the convolutional layer and non-negative input[17]. Besides, if needed, a three-parameter Gamma distribution is also available with a location parameter c and pdf $f(x|c, \theta, k) = \frac{1}{\Gamma(k)\theta^k}(x-c)^{k-1} \exp -\frac{(x-c)}{\theta}$ where $c \in (-\infty, \infty)$.

Assuming the feature map is positive, i.e., $c = 0$, gamma distribution applies to them in most cases. If $k = 1$, $G(\theta, 1)$ is an exponential distribution also shown in Figure 6.32. Another example in Figure 6.35b in § 6.14 shows an image with an object at a corner yields an exponentially distributed feature map. The MLE for exponential is the sample mean,

$$\hat{\theta} = \frac{\sum_i x_i}{n}. \tag{6.19}$$

Under other circumstances, the gamma distribution is flexible to take different shapes. However, a closed-form MLE does not exist for them. Fortunately, mixed type log-moment estimators exist that have similar efficiency as MLEs (Ye and N. Chen 2017). They are,

$$\hat{\theta} = \frac{1}{n^2}\left(n\sum_i x_i \log(x_i) - \sum_i \log(x_i) \sum_i x_i\right) \tag{6.20}$$

$$\hat{k} = \frac{n\sum_i x_i}{n\sum_i x_i \log(x_i) - \sum_i \log(x_i) \sum_i x_i}. \tag{6.21}$$

[17]E.g. Input scaled with `sklearn.preprocessing.MinMaxScaler()` and Convolutional layer defined as
`Conv1D(..., kernel_constraint=tf.keras.constraints.NonNeg(), bias_constraint=tf.keras.constraints.NonNeg(),...)`.

Based on the listed complete sufficient statistic for the gamma distribution in Table 6.2 as per Theorem 9, the estimators $\hat{\theta}$ and $\hat{k}$ are complete sufficient statistics. However, the statistics are biased. The bias-corrected statistics are (Louzada, P. L. Ramos, and E. Ramos 2019),

$$\tilde{\theta} = \frac{n}{n-1}\hat{\theta} \tag{6.22}$$

$$\tilde{k} = \hat{k} - \frac{1}{n}\left(3\hat{k} - \frac{2}{3}\left(\frac{\hat{k}}{1+\hat{k}}\right) - \frac{4}{5}\frac{\hat{k}}{(1+\hat{k})^2}\right). \tag{6.23}$$

These unbiased estimators are a function of the complete sufficient statistics. Therefore, according to Theorem 6 $\tilde{\theta}$ and $\tilde{k}$ are MVUE. In effect, they exhibit the same properties expected from MLEs for pooling.

Note that, in practice, $\log(x)$ in the statistics can be replaced with $\log(x+\epsilon)$ where ϵ is a small constant in $\mathbb{R}^+$, e.g., $\epsilon \leftarrow 1e-3$. This adds a small bias to the estimates but also makes them stable. This correction addresses the possibility of feature maps taking extremely small values.

6.13.4 Weibull Distribution

A Weibull distribution is also an asymmetrical distribution from the exponential family. It has similar parameters as a gamma distribution, scale λ and shape k, and also a similar form. Like a gamma distribution, Weibull is also lopsided and has a spread characterized by k and λ. But they differ in their precipitousness. This is clearer from the pdf given as,

$$f(x) = \frac{k}{\lambda}\left(\frac{x}{\lambda}\right)^{k-1} \exp -\left(\frac{x}{\lambda}\right)^{k} \tag{6.24}$$

where, $x \geq 0$, and $\lambda, k > 0$.

Ignoring the constants in the pdf of gamma and Weibull distributions (in Equation 6.18 and Equation 6.24, respectively), one can note that the difference is $\exp(-x/\theta)$ versus $\exp -(x/\lambda)^k$. This implies that Weibull distribution precipitates rapidly if $k > 1$ and slowly if $k < 1$, while the

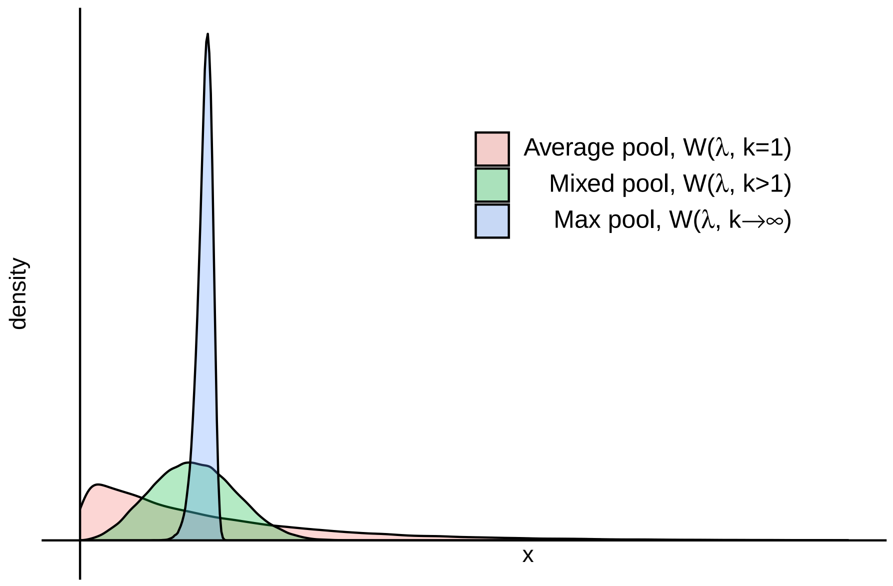


Figure 6.33. *Pooling has typically stayed with max or average statistics. A better pooling, however, is hypothesized to be somewhere between them. Boureau, Ponce, and LeCun 2010 parameterized pooling statistic as* $f(x) = \left(\frac{1}{n}\sum_i x_i^k\right)^{\frac{1}{k}}$*, which gives the average for* $k = 1$ *and the max for* $k \to \infty$*. A value in between for* k *gives a pooling statistic that is a "mixture" of average and max. This parameterization comes from the assumption that* X *is Weibull distributed, i.e.,* $X \sim W(\lambda, k)$*, and the pooling statistic is the MLE for* λ*.*

gamma distribution is indifferent. If $k = 1$, both gamma and Weibull are equal and reduces to an exponential distribution.

The ability to adjust the precipitousness makes Weibull distribution a sound choice for fitting feature maps. Because the separability of features defines the precipitousness. For example, a strong feature map sharply distinguishes an object's features and will have a quickly dropping off pdf, i.e., a large k.

Therefore, the MLE of Weibull distribution given below is quite appropriate for pooling.

$$\hat{\lambda} = \left(\frac{1}{n} \sum_i x_i^k \right)^{\frac{1}{k}} \tag{6.25}$$

assuming k is known.

Interestingly, the parametrization given in Boureau, Ponce, and LeCun 2010 is the MLE of Weibull distribution. This is interesting because in their or other prior work a connection between pooling and Weibull distribution was not established.

Instead, Boureau, Ponce, and LeCun 2010 found that the optimal pooling statistic is somewhere between an average- and max-pooling. They, therefore, posed the pooling statistic as in Equation 6.25 that continuously transitions from average- to max-pooling as follows,

- if $k = 1$, then average-pooling,
- if $k \to \infty$, then max-pooling, and
- if $1 < k < \infty$, then mixed-pooling.

The Weibull distribution in these conditions are shown in Figure 6.33. In the figure, it is noticed that under the max-pool condition the distribution precipitates quickly indicating a distinguishing feature map. The average-pool is for $k = 1$ when Weibull becomes an Exponential distribution for which the MLE is indeed the sample mean (Equation 6.19). A mixed pooling is in between the two and so is the shape of its distribution.

In the above, the shape parameter k is assumed to be known. Fitting a Weibull distribution to feature maps can become more informative if the k is also derived from the features (data). The distribution's form for different (λ, k) are shown in Figure 6.34 for illustration.

Unfortunately, a closed-form MLE for k does not exist. It can be estimated using numerical methods by solving,

$$\frac{\sum_i x_i^k \log(x_i)}{\sum_i x_i^k} - \frac{1}{k} - \frac{1}{n}\sum_i \log(x_i) = 0 \tag{6.26}$$

Moreover, Weibull distribution can also be fitted if feature maps are lower bounded at τ, where $\tau > 0$. Then the MLE for $\hat{\lambda}$ is,

$$\hat{\lambda} = \left(\frac{1}{\sum_i \mathbb{1}(x_i > \tau)}\sum_i (x_i^k - \tau^k)\mathbb{1}(x_i > \tau)\right)^{\frac{1}{k}} \tag{6.27}$$

and k can be estimated by solving,

$$\frac{\sum_i (x_i^k \log(x_i) - \tau^k \log(\tau))}{\sum_i (x_i^k - \tau^k)} - \frac{1}{\sum_i \mathbb{1}(x_i > \tau)}\sum_i \log(x_i)\mathbb{1}(x_i > \tau) = 0 \tag{6.28}$$

where $\mathbb{1}(x_i > \tau)$ is an indicator function equal to 1, if $x_i > \tau$, and 0, otherwise.

These expressions are derived from a three-parameter Weibull defined as, $f(x) = \frac{k}{\lambda}\left(\frac{x-\tau}{\lambda}\right)^{k-1} \exp -\left(\frac{x-\tau}{\lambda}\right)^{k}$, where $\tau < x < \infty$ (R. L. Smith 1985; Hirose 1996)[18]. They are helpful to accommodate some types of activation or dropout techniques.

Lastly, like gamma, Weibull distribution also supports only positive x. Therefore, approaches, such as positive-constrained inputs, kernel, and bias, discussed in § 6.13.3 can be used.

[18] A similar three-parameter is also for gamma distribution in R. L. Smith 1985 but closed-form estimators are unavailable.

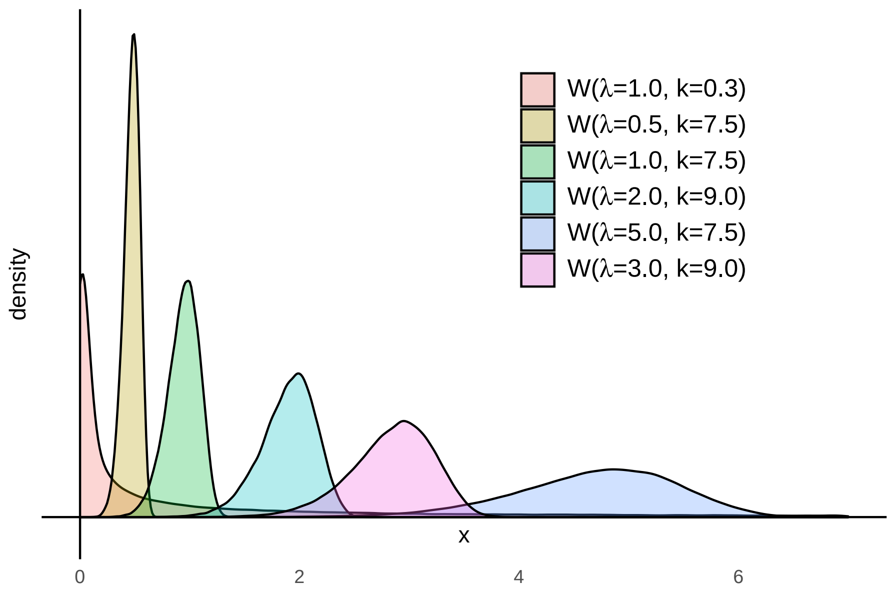


Figure 6.34. *Weibull distribution, $X \sim W(\lambda, k)$, $X \in \mathbb{R}^+$, and $\lambda, k \in \mathbb{R}^+$, is another asymmetric distribution from the exponential family. Similar to a Gamma distribution, Weibull provides flexibility for different shapes and scales. Moreover, the Exponential distribution is also a special case of Weibull if $k = 1$. MLE is available for λ in Weibull, $\hat{\lambda} = \left(\frac{1}{n}\sum_i x_i^k\right)^{\frac{1}{k}}$. Closed-form MLE for k is, however, unavailable and can be estimated via numerical methods.*

.

6.14 Advanced Pooling

The possibility of fitting distributions uncovered a myriad of pooling statistics. It also made possible using advanced techniques, such as an adaptive selection of distribution.

Such techniques have significance because optimal pooling depends on the characteristics of feature maps in convolutional networks and the data set. Automatically determining the optimal pooling is challenging and discussed in expanse in Lee, Gallagher, and Tu 2016; Kobayashi 2019b.

Moreover, MLEs are learned to be the appropriate pooling statistic. But they are unavailable for some distributions.

Besides, convolutional feature maps are likely to be dependent in most data sets because nearby features are correlated. However, pooling statistics are developed assuming their independence.

This section provides a few directions to address these challenges.

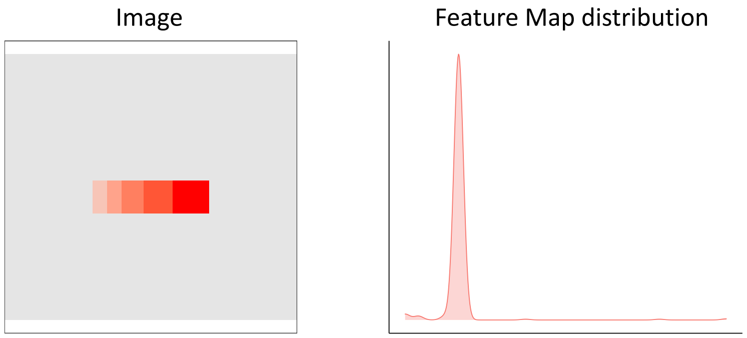
6.14.1 Adaptive Distribution Selection

Feature maps for different samples in a data set can differ significantly. For illustration, Figure 6.35a and 6.35b show images with an object at the center and corner, respectively. The images are filtered through a Sobel filter (§ 12.6.2 in McReynolds and Blythe 2005). Also shown in the figures is the feature map distribution yielded by the filter.

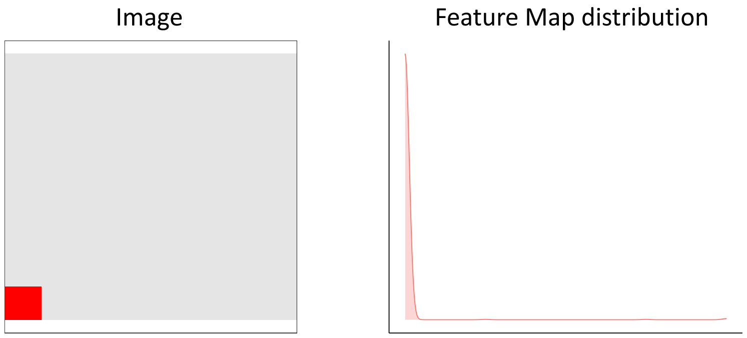
The former image results in a peaked distribution that can be from a normal, gamma, or Weibull while the latter results in an exponential-like distribution.

If the distribution is known, using MLEs for pooling is fitting the distribution to the feature map.

This is straightforward with normal and gamma distribution as closed-form estimators exist for their parameters. For Weibull, MLE is available for scale λ but not for the shape k. Although k can be numerically estimated by solving Equation 6.26 for k, it is computationally intensive. However, k need not be estimated with precision. Instead, k can be assumed to belong in a finite discrete set of positive real numbers, i.e.,



(a) *Object with color gradient.*



(b) *Object on a corner.*

Figure 6.35. *An image containing an object with a color gradient (top) and another image with an object at a corner (bottom) are convolved with a Sobel kernel that detects the color gradient changes. The outputted convolution feature map has a gamma (top) and exponential (bottom) distribution, respectively. Their respective MLEs should be used in pooling.*

$\mathcal{K} = \{k | k \in \mathbb{R}^+\}$ and $|\mathcal{K}|$ is small. The $k \in \mathcal{K}$ that yields the maximum likelihood should be chosen, i.e.,

$$\arg_k \max \prod_i f(x_i | \hat{\lambda}, k) \tag{6.29}$$

where f is the pdf of Weibull in Equation 6.24, $\hat{\lambda}$ is estimated using Equation 6.25, and $x_i, i = 1, \ldots, n$ is the feature map.

Deriving pooling statistics by fitting distributions is effective. But it works on an assumption that the distribution is known. Since, exponential distributions, viz. normal, gamma, and Weibull, provide reasonable flexibility, the assumption is not strong. Still further improvement can be made by adaptively choosing a distribution, i.e., let the distribution be data-driven. An approach could be to fit various distributions and select the one which yields the maximum likelihood.

6.14.2 Complete Statistics for Exponential Family

We have learned that MLEs are the best pooling statistic. But their closed-form expressions are sometimes unknown.

We know that MLEs are a function of complete statistic(s). In absence of an MLE expression, complete statistic(s) can be used in pooling.

Most feature maps follow a distribution from exponential family and, fortunately, complete statistics for any distribution from the family is available based on Theorem 9 below.

Theorem 9. ***Exponential family complete sufficient statistics***[19]. *Let $X_1, \ldots, X_n$ be iid observations from an exponential family with pdf or pmf of form*

$$f(x|\boldsymbol{\theta}) = h(x)c(\boldsymbol{\theta}) \exp\left(\sum_{j=1}^{k} w_j(\boldsymbol{\theta}) t_j(x)\right) \tag{6.30}$$

[19]Based on Theorem 8.1 in Lehmann and Scheffé 1955 and Theorem 6.2.10 in Casella and Berger 2002.

Table 6.2. List of Complete Sufficient Statistics based on Theorem 9 for Exponential family distributions.

Distribution	**Pdf**, x, $\boldsymbol{\theta}$	**Complete sufficient statistics**, $T(\boldsymbol{X})$
Normal	$\frac{1}{\sqrt{2\pi\sigma^2}} \exp - \frac{(x-\mu)^2}{2\sigma^2}$, $x \in (-\infty, \infty)$, $\boldsymbol{\theta} = (\mu, \sigma^2)$	$\left(\sum_{i=1}^n X_i, \sum_{i=1}^n X_i^2\right)$
Exponential	$\lambda \exp(-\lambda x)$, $x \in [0, \infty)$, $\boldsymbol{\theta} = \lambda$	$\sum_{i=1}^n X_i$
Gamma	$\frac{\beta^\alpha}{\Gamma(\alpha)} \exp - \Big(\beta x - (\alpha - 1) \log x\Big)$, $x \in (0, \infty)$, $\boldsymbol{\theta} = (\alpha, \beta)$	$\left(\sum_{i=1}^n X_i, \sum_{i=1}^n \log X_i\right)$
Weibull	$\frac{k}{\lambda} \exp - \Big((\frac{x}{\lambda})^k - (k-1) \log \frac{x}{\lambda}\Big)$, $x \in [0, \infty)$, $\boldsymbol{\theta} = (\lambda, k)$	$\left(\sum_{i=1}^n X_i^k, \sum_{i=1}^n \log X_i\right)$

where $\boldsymbol{\theta} = (\theta_1, \theta_2, \ldots, \theta_k)$. *Then the statistic*

$$T(\boldsymbol{X}) = \left(\sum_{i=1}^n t_1(X_i), \sum_{i=1}^n t_2(X_i), \ldots, \sum_{i=1}^n t_k(X_i)\right) \tag{6.31}$$

is complete if $\{(w_1(\boldsymbol{\theta}), \ldots, w_k(\boldsymbol{\theta})) : \boldsymbol{\theta} \in \Theta\}$ *contains an open set in* $\mathbb{R}^k$. *Moreover, it is also a sufficient statistic for* θ.

It can be noted that the MLEs for normal, gamma and Weibull distributions are indeed a function of the complete statistics listed in

Table 6.2 based on the theorem. Similarly, complete statistic(s) for any other distribution from the exponential family can be determined for pooling.

6.14.3 Multivariate Distribution

Most of the pooling approaches assume feature maps are independent. The independence assumption is, however, false since close by image features are correlated. For example, if a filter detects an edge then it is likely to find the next pixel as an edge as well. Consequently, the feature map it yields will have dependence.

Addressing the dependence is challenging with traditional pooling methods. Only a few techniques address it. Saeedan et al. 2018 is one of them which uses the concept developed in Weber et al. 2016 in the field of image processing for detail-preservation for downscaling in pooling.

The dependence can be addressed by making the features independent, e.g., removing autocorrelation, before fitting distributions (pooling). Features dependence can also be addressed by fitting multivariate distributions, e.g., multivariate normal and Wishart distributions.

6.15 History of Pooling

Pooling is an important construct of a convolutional network. Without pooling, the network is statistically and computationally inefficient, and virtually dysfunctional.

The concept has roots in biological constructs. Many computer vision architectures including pooling in convolutional networks have been inspired by studies of the visual cortex on multi-stage processing of simple and complex cells in Hubel and Wiesel 1962. Translational invariance is achieved by the complex cells that aggregate local features in a neighborhood.

One of the earliest neural networks with the pooling concept is the Neocognitron (Fukushima 1986). This network employed a combination of “S-cells” and “C-cells” which acted as activation and pooling, respectively. The “C-cells” become active if at least one of the inputs from the “S-cells” is active. This is similar to a binary gate that makes the network robust to slight deformations and transformations.

LeCun, Boser, et al. 1990 introduced the idea of parameter sharing with convolution and network invariance via subsampling by taking an average of the local features. Average-pooling was further used in LeCun, Bottou, et al. 1998.

Max-pooling was put forward soon after in Riesenhuber and Poggio 1999. The paper discusses the biological functioning of the visual cortex and lays two idealized pooling mechanisms, linear summation (‘SUM’) with equal weights (to achieve an isotropic response), and a nonlinear maximum operation (‘MAX’), where the strongest afferent determines the postsynaptic response. They are average and max-pooling, respectively.

Riesenhuber and Poggio 1999 compared average- and max- pooling from a biological visual cortex functioning standpoint. They explained that responses of a complex cell would be invariant as long as the stimulus stayed in the cell’s receptive field. However, it might fail to infer whether there truly is a preferred feature somewhere in the complex cell’s receptive field. In effect, the feature specificity is lost. However, in max-pooling the output is the most active afferent and, therefore, signals the best match of the stimulus to the afferent’s preferred feature. This premise in Riesenhuber and Poggio 1999 explained the reason behind max-pool’s robustness over the average.

Max-pool was further used and empirical evidence of its efficacy was found in Gawne and Martin 2002; Lampl et al. 2004; Serre, Wolf, and Poggio 2005; Ranzato, Boureau, and Cun 2008. Using max-pool, Yang et al. 2009 reported much better classification performance on multi-object or scene-classification benchmarks compared to average-pool. However, no theoretical justification behind max-pool’s outperformance was yet given.

Boureau, Ponce, and LeCun 2010 perhaps provided the earliest theoretical support behind max-pool. They assumed feature maps as Bernoulli random variables that take values 0 or 1. Under the assumption, they expressed the mean of separation and variance of max-pooled features.

Their expressions show that max-pool does better class separation than average. However, the justification was based on an extreme simplification of Bernoulli distribution while feature maps are continuous in most problems. To which, Ranjan 2020 recently provided more general proof from a statistical standpoint.

Besides, the possibility of the optimal pooling lying in between average- and max- pooling was seeded in Boureau, Ponce, and LeCun 2010. They, themselves, also provided a parameterization to combine both as $\sum_i \frac{\exp(\beta \boldsymbol{x}_i + \alpha)}{\sum_j \exp(\beta \boldsymbol{x}_j + \alpha)}$ which is equivalent to average or max if $\beta \to 0$ and $\beta \to \infty$, respectively, and $\alpha = 0$. A more sophisticated approach to estimate the α, β from features, i.e., trainable mixing parameters, based on maximum entropy principle was developed in Kobayashi 2019b.

The mixing idea was taken in D. Yu et al. 2014; Lee, Gallagher, and Tu 2016 to propose mixed-pooling as $f_{mix}(\boldsymbol{x}) = \alpha \cdot f_{max}(\boldsymbol{x}) + (1 - \alpha) \cdot f_{avg}(\boldsymbol{x})$. Lee, Gallagher, and Tu 2016 also devised a trainable mixing called as gated-pooling in which the weight $\alpha = \sigma(\boldsymbol{\omega}^T \boldsymbol{x})$ where σ is a sigmoid function in $[0, 1]$ and $\boldsymbol{\omega}$ is learned during training. A further extension, called Tree-pooling, was also proposed by them that automatically learned the pooling filters and mixing weights to responsively combine the learned filters.

Although pooling is typically used for invariance, a variant called stochastic pooling was proposed in Zeiler and Fergus 2013 for regularization of convolutional networks. Stochastic pooling works by fitting a multinomial distribution to the features and randomly drawing a sample from it.

Most of the pooling techniques worked by assuming the independence of features. However, it is a strong assumption because nearby features are usually correlated. Ranjan 2020 discussed this issue and provided a few directions, e.g., fitting a multivariate distribution and using its summary statistic or removing feature dependency before computing a statistic for pooling. Prior to this, T. Williams and Li 2018 and Saeedan et al. 2018 have worked in a similar direction.

T. Williams and Li 2018 proposed a wavelet pooling that uses a second-level wavelet decomposition to subsample features. This approach was claimed to resolve the shortcomings of the nearest neighbor interpolation-like method, i.e., local neighborhood features pooling, such as edge halos, blurring, and aliasing Parker, Kenyon, and Troxel 1983.

Saeedan et al. 2018 developed a "detail-preserving" pooling drawn from the approach for image processing in Weber et al. 2016 called detail-preserving image downscaling (DPID). DPID calculates a weighted average of the input, but unlike traditional bilateral filters (Tomasi and Manduchi 1998), it rewards the pixels that have a larger difference to the downscaled image. This provides a customizable level of detail magnification by allowing modulation of the influence of regions around edges or corners.

Kobayashi 2019a proposed fitting Gaussian distribution on the local activations and aggregate them into sample mean μ_X and standard deviation σ_X. They devised the pooling statistic as $\mu_X + \eta\sigma_X$ where $\mu_X = \frac{\sum_i X_i}{n}$ and $\sigma_X = \sqrt{\frac{\sum_i (X_i - \bar{X})^2}{(n-1)}}$. Since the activations are not full Gaussian, they fit half-Gaussian and inverse softplus-Gaussian. Still, the activations are unlikely to follow these distributions due to the phenomenon of distribution distortion presented in Ranjan 2020. Moreover, the $\mu_X + \eta\sigma_X$ is not a *complete* statistic due which it does not have the minimum variance.

Besides, He et al. 2015b developed a spatial pyramid pooling. It is mainly designed to deal with images of varying size, rather than delving in to different pooling functions or incorporating responsiveness.

There is another school of thought that rallies against pooling altogether. For example, Springenberg et al. 2015 proposed an "all Convolutional Net" which replaced the pooling layers with repeated convolutional layers. To reduce the feature map size, they used larger stride in some of the convolutional layers to achieve a similar performance as with the pooling layers. Variational autoencoders (VAEs) or generative adversarial networks (GANs) are also found to work better in absence of pooling layers. However, this could be due to the usage of lossy statistics[20] for pooling. A use of MLEs or complete sufficient statistics as proposed in Ranjan 2020 would work well in VAEs or GANs.

[20]Summary statistics that lose the information of the original data.

6.16 Rules-of-thumb

- **Baseline network**. Construct a simple sequential baseline model with *convolution* → *pooling* → *activation* → *flatten* → *dense* → *output* layer structure. Note to swap activation and pooling layers.
- **Convolution layer**
  - `Conv1D` **vs.** `Conv2D` **vs.** `Conv3D`. A `Conv‘x’D` is chosen based on the number of spatial axes in the input. Use `Conv1D`, `Conv2D`, and `Conv3D` for inputs with 1, 2, and 3 spatial axes, respectively. Follow Table 6.1 for details.
  - **Kernel**. The `kernel_size` argument is a tuple in which each element represents the size of the kernel along a spatial axis. The size can be taken as the $\sqrt{\text{spatial axis size}}/2$.
  - **Filters**. The number of `filters` can be taken as a number from the geometric series of 2 close to n_features/4.
  - **Activation**. Use a `linear` activation. A nonlinear activation will be added after the pooling layer.
  - **Padding**. Use `valid` padding in a shallow network. A shallow network is defined as one in which the feature map size does not reduce significantly before the output layer. In deep networks with several convolutional layers, use the `same` padding at least in some of the convolutional layers.
  - **Dilation**. Do not use dilation in a baseline model. In deep networks, undilated and dilated convolutional layers can be paired.
  - **Stride**. Use the default `stride=1` in a baseline model. The stride can be increased to `2` if the input dimension is high. However, in most problems, a stride larger than `2` is not recommended.
- **Pooling layer**
  - `Pool1D` **vs.** `Pool2D` **vs.** `Pool3D`. Use the pooling layer consistent with the `Conv` layer. For example, `Pool1D` with `Conv1D`, and so on.

  - **Pooling statistic**. Use maximum statistic for pooling via the `MaxPool` layer.
  - **Pool size**. The square root of the feature map size (along the spatial axes) can be taken.
  - **Padding**. Follow the same principle as for convolution padding. Use `valid` padding in a shallow and `same` in a deep network.

- **Activation**. Use a nonlinear `relu` activation layer after a pooling layer to follow *convolution* → *pooling* → *activation* structure.

- **Dense layer**. After the stack of *convolution* → *pooling* → *activation* bundled layers, place a dense layer to downsize the feature map. Use a `Flatten()` layer before the dense layer to vectorize the multi-axes feature map. The number of units can be set equal to a number in the geometric series of 2 closest to the length of the flattened features divided by 4.

- **Output layer**. The output layer is dense. The number of units is equal to the number of labels in the input. For example, in a binary classifier, the number of units is equal to 1. The other configurations of the output layer follow the same rules-of-thumb as in § 4.10.

6.17 Exercises

1. **Long-term dependencies**. Recurrent neural networks, e.g., LSTMs are meant to learn long-term dependencies. However, it is found that in some high-dimensional multivariate time series problems LSTMs perform poorly if the time-steps (lookback) is increased. A convolutional neural network, on the other hand, works better even with long lookbacks.

   (a) Refer to Jozefowicz, Zaremba, and Sutskever 2015 and explain the reason behind LSTM and other RNNs limitation in learning long-term dependencies.

   (b) Explain why convolutional networks still work well with long time-steps?

   (c) Train and evaluate the baseline convolutional network in § 6.9.3 with a `TIMESTEPS` equal to `120` and `240`, and report the increase in the model accuracy, parameters, and runtime. Interpret your findings.

   (d) Plotting the feature maps and filters provide some interpretation of the model. Refer to Appendix I to plot them and report your interpretations.

2. **`Conv1D`, `Conv2D`, and `Conv3D`**. § 6.8.2 explained the different scenarios in which `Conv1D`, `Conv2D`, or `Conv3D` could be used. Next, § 6.10.1 explained that they are top-down interchangeable. Also, as shown in § 6.10.2, the interchangeability provides more modeling choices when a higher level `Conv‘x’D` is used.

   (a) Refer to § 6.10.1 and construct a convolutional network using `Conv3D` layer that is equivalent to the baseline model in § 6.9.3.

   (b) Construct another convolutional network using `Conv3D` layer equivalent to the neighborhood model in § 6.10.2.

   (c) Explain how a `Conv3D` layer can replace a `Conv2D` layer in a convolutional network for images?

3. **1×1 Convolution**. A 1×1 convolutional layer summarizes the information across the channels.

   (a) Explain how is this conceptually similar to and different from pooling?

   (b) Refer to Appendix H to construct a network with 1×1 convolutional layers. Present the improvement in model performance.

   (c) Plot the feature maps outputted by the 1×1 convolutional layer (refer Appendix I) and present your interpretations.

4. **Summary Statistics.** § 6.11 expounded the theory of minimal sufficient and complete statistics which led to a maximum likelihood estimator (MLE) shown as the best pooling statistic in § 6.13. Additionally, ancillary statistics were put forward as complementary pooling statistics to enhance a network's performance.

   (a) Prove that the maximum statistic $\max_i X_i$ is a complete minimal sufficient statistic for a uniform distribution. Also, show that it is the MLE for uniform as mentioned in § 6.13.1.

   (b) Derive the MLEs for normal, gamma, and Weibull distributions given in § 6.13.2-6.13.4.

5. **Pooling statistics**. It is mentioned in § 6.12 the feature map distribution distortion is caused by nonlinear activations. A possibility of using pooling statistics other than average and maximum arises by addressing the distortion.

   (a) Train the baseline model with max- and average pooling. Then swap activation and pooling layers, and make the convolutional layer activation as `linear` as described in § 6.12.2 in the same baseline network. Train with max- and average-pooling again. Compare the results for both pooling before and after the swap.

   (b) Construct a network by referring to Appendix J with a maximum and range $(\max - \min)$ pooling statistics in parallel. The range statistic is an *ancillary* statistic that complements the maximum statistic with regards to the information drawn from a feature map. Present and discuss the results.

(c) Explain why swapping the activation and pooling layers make use of ancillary statistics such as range and standard deviation possible?

(d) (Optional) Construct a network with average and standard deviation pooling statistics in parallel. They are the MLEs of a normal distribution. If feature maps are normally distributed, they pool the most relevant information. Present and discuss the results.

(e) (Optional) Construct a network with Weibull distribution MLE as the pooling statistic. Train the network with the shape parameter k in $\{0.1, 1, 10, 100\}$. Present the results and explain the effect of k.

Chapter 7

Autoencoders

7.1 Background

An **autoencoder** is a reconstruction model. It attempts to reconstruct its inputs from itself as depicted below,

$$\boldsymbol{x} \rightarrow \underbrace{f(\boldsymbol{x}) \rightarrow \boldsymbol{z}}_{encoding} \rightarrow \underbrace{g(\boldsymbol{z}) \rightarrow \hat{\boldsymbol{x}}}_{decoding} . \tag{7.1}$$

Clearly, an autoencoder is made of two modules: an encoder and a decoder. As their names indicate, an encoder f encodes input $\boldsymbol{x}$ into encodings $\boldsymbol{z} = f(\boldsymbol{x})$, and a decoder g decodes $\boldsymbol{z}$ back to a closest reconstruction $\hat{\boldsymbol{x}}$.

Training a model to predict (construct) $\hat{\boldsymbol{x}}$ from $\boldsymbol{x}$ sometimes appear trivial. However, an autoencoder does not necessarily strive for a perfect reconstruction. Instead, the goal could be dimension reduction, denoising, learning useful features for classification, pre-training another deep network, or something else.

Autoencoders, therefore, fall in the category of **unsupervised** and **semi-supervised** learning.

They were conceptualized in the late 80's. These early works were inspired by principal component analysis. PCA, which was invented more than a century ago (Pearson 1901), has remained a popular feature representation technique in machine learning for dimension reduction and feature representation. Autoencoders developed in Rumelhart, G. E. Hinton, and R. J. Williams 1986; Baldi and Hornik 1989 provided a neural network version of PCA.

Over the past two decades, autoencoders have come far ahead. Sparse encoding with feature space larger than the input was developed in Ranzato, Poultney, et al. 2007; Ranzato, Boureau, and Cun 2008. Furthermore, denoising autoencoders based on corrupted input learning in Vincent, Larochelle, Bengio, et al. 2008; Vincent, Larochelle, Lajoie, et al. 2010 and contractive penalty in Rifai, Vincent, et al. 2011; Rifai, Mesnil, et al. 2011 came forward.

In this chapter, some of the fundamental concepts behind this variety of autoencoders are presented. It begins with explaining an autoencoder vis-à-vis a PCA in § 7.2. A simpler explanation is provided here by drawing parallels. Thereafter, the family of autoencoders and their properties are presented in § 7.3.

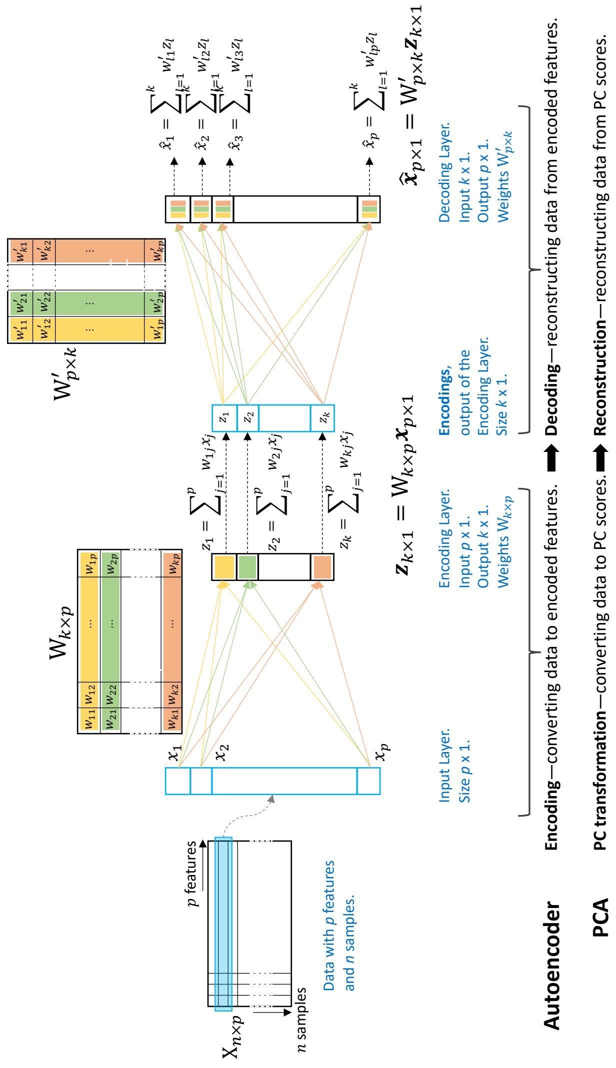
The chapter then switches to using autoencoders for rare event prediction. An anomaly detection approach is used in § 7.4. Thereafter, a sparse autoencoder is constructed and a classifier is trained on the sparse encodings in § 7.5.

A sparse LSTM and convolutional autoencoders are also constructed in § 7.6 for illustration. Towards the end of the chapter, § 7.7 presents autoencoder customization in TensorFlow to incorporate sparse covariance and orthogonal weights properties. The customization example here is intended to help researchers implement their ideas. Lastly, the chapter concludes with a few rules-of-thumb for autoencoders.

7.2 Architectural Similarity between PCA and Autoencoder

For simplicity, a linear single layer autoencoder is compared with principal component analysis (PCA).

There are multiple algorithms for PCA modeling. One of them is estimation by minimizing the reconstruction error (see § 12.1.2 in Bishop



$X_{n\times p}$
p features
n samples
Data with p features and n samples.
Input Layer. Size p x 1.
$W_{k\times p}$
$z_1 = \sum_{j=1}^{p} w_{1j}x_j$
$z_2 = \sum_{j=1}^{p} w_{2j}x_j$
$z_k = \sum_{j=1}^{p} w_{kj}x_j$
$\mathbf{z}_{k\times 1} = W_{k\times p}\boldsymbol{x}_{p\times 1}$
Encoding Layer. Input p x 1. Output k x 1. Weights $W_{k\times p}$
Encodings, output of the Encoding Layer. Size k x 1.
$W'_{p\times k}$
$\hat{x}_1 = \sum_{l=1}^{k} w'_{l1}z_l$
$\hat{x}_2 = \sum_{l=1}^{k} w'_{l2}z_l$
$\hat{x}_3 = \sum_{l=1}^{k} w'_{l3}z_l$
$\hat{x}_p = \sum_{l=1}^{k} w'_{lp}z_l$
$\hat{\boldsymbol{x}}_{p\times 1} = W'_{p\times k}\mathbf{z}_{k\times 1}$
Decoding Layer. Input k x 1. Output p x 1. Weights $W'_{p\times k}$
Autoencoder
Encoding—converting data to encoded features.
Decoding—reconstructing data from encoded features.
PCA
PC transformation—converting data to PC scores.
Reconstruction—reconstructing data from PC scores.

Figure 7.1. *Illustration of PCA and Autoencoder relationship.*

2006). Following this algorithm gives a clearer understanding of the similarities between a PCA and an autoencoder.

Figure 7.1 visualizes a single-layer linear autoencoder. As shown at the bottom of the figure, the encoding process is similar to the principal component transformation.

PC transformation is projecting the original data on the principal components to yield reduced-dimension features called principal scores. This is similar to the *encoding* process in an autoencoder.

Conversely, an autoencoder's *decoding* process is similar to reconstructing the data from the Principal Scores.

In the following, their relationship is further elaborated by showcasing the key autoencoder components and their counterpart in PCA.

7.2.1 Encoding—Projection to Lower Dimension

PCA is a covariance decomposition expressed as $\Sigma = W \Lambda W^T$, where Σ is the covariance matrix of X, W is the eigenvectors matrix, and Λ is an eigenvalues diagonal matrix.

The W matrix is an orthogonal basis, also known as the principal components. The transformation of the original data X into principal features is,

$$Z = XW \tag{7.2}$$

where W is a $p \times k$ matrix with eigenvectors in each column. The eigenvectors are the top k principal components. The k is chosen as less than the original dimension p of X, i.e., $k < p$. This dimension reduction can also be seen as an **encoding**.

A similar process occurs in the encoding stage of an autoencoder. Look closely at the encoder section in Figure 7.1.

The encoding (dense) layer has k nodes. A colored cell in the layer is a computing node with p weights. The weights can also be denoted as a $p \times k$ matrix W. The weights on each node are the equivalent of an eigenvector in PCA.

The encoding output is,

$$Z = g(XW) \tag{7.3}$$

where g is an activation function. If the activation is linear, the encoding in autoencoder becomes equivalent to the principal scores in PCA (Equation 7.2).

7.2.2 Decoding—Reconstruction to Original Dimension

Reconstructing the data is also called **decoding**. The original data can be reconstructed (estimated) from the principal scores as,

$$\hat{X} = ZW^T. \tag{7.4}$$

Similarly, the decoder in an autoencoder reconstructs the data. As shown in Figure 7.1, the data is reconstructed as,

$$\hat{X} = ZW' \tag{7.5}$$

where W' is a $k \times p$ weight matrix. Unlike in encoding, the activation in decoding depends on the range of the input. For example, if the input $\boldsymbol{x} \in \mathbb{R}$ then a linear activation but if $\boldsymbol{x} \in \mathbb{R}^+$ then a ReLU-like activation. Since the inputs are usually scaled to belong in $\mathbb{R}$, it is safe to assume that the decoding has linear activation.

Note that the W in Equation 7.4 for PCA is the same as the encoding weights in Equation 7.2. But a different weight W' in Equation 7.5 is for a decoder. This is because the encoding and decoding weights are not necessarily the same in autoencoders.

But they can be *tied* to become the same. This unison is shown in Figure 7.1 using colors in the W matrices. When the encoder and decoder weights are tied, we have

$$W' = W^T \tag{7.6}$$

which means the rows of the encoder weights become equal to the columns of decoder weights.

If the encoder and decoder weights are ***tied*** *and the encoder has* ***linear*** *activation, the autoencoder becomes equivalent to a PCA model. Conversely, a nonlinearly activated encoder is a nonlinear extension of PCA.*

7.3 Autoencoder Family

There are several types of autoencoders. Table 7.1 summarizes the properties of the most common autoencoders. The rest of this section briefly describes them along with their applications.

7.3.1 Undercomplete

An undercomplete autoencoder has a smaller encoding dimension than the input. A simple example of such an autoencoder is,

$$X_{\cdot\times p} \rightarrow \underbrace{f(X_{\cdot\times p}W^{(e)}_{p\times k})}_{encoder} \rightarrow Z_{\cdot\times k} \rightarrow \underbrace{g(Z_{\cdot\times k}W^{(d)}_{k\times p})}_{decoder} \rightarrow \hat{X}_{\cdot\times p}. \tag{7.7}$$

Here the input X and the encoding Z are p- and k-dimensional, respectively, and $k < p$.

In learning a smaller representation, an undercomplete autoencoder gathers the most salient features of the data. The learning process is simply minimizing a loss function

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}W^{(e)})W^{(d)})) \tag{7.8}$$

where L is a loss function, for example, mean squared error, that penalizes dissimilarity between $\boldsymbol{x}$ and $\hat{\boldsymbol{x}} = g(f(\boldsymbol{x}W^{(e)})W^{(d)})$.

Undercomplete autoencoders are more common. Perhaps because it has roots in PCA. A linearly activated (single or multilayer) undercomplete autoencoder reduces to

Table 7.1. Types of autoencoders and their properties.

	Undercomplete autoencoder	Regularized overcomplete	Sparse autoencoder	Denoising autoencoder	Contractive autoencoder	Well-posed autoencoder
(Near) Orthogonal weights, $\lambda \|\|W^T W - I\|\|_F^2$						X
Unitnorm weights, $\|\|W\|\|_2 = 1$	X	X				X
Sparse encoding covariance, $\lambda \|\|\Omega_z (1 - I)\|\|_F^2$						X
Sparse encoding, $\lambda \|\|Z\|\|_1$		X	X			
Small derivative, $\lambda \|\|\frac{\partial z}{\partial \boldsymbol{x}}\|\|_F^2$		X			X	
Small encoding, $k < p$	X					X
Corrupted input, $\boldsymbol{x} \leftarrow \boldsymbol{x} + \epsilon$				X		

$$X_{\cdot\times p} \to X_{\cdot\times p}W^{(e)}_{p\times k} \to Z_{\cdot\times k} \to Z_{\cdot\times k}W^{(d)}_{k\times p} \to \hat{X}_{\cdot\times p},$$

which is the same as PCA if $W^{(e)}$ has eigenvectors.

These roots bring the dimension reduction ability in undercomplete autoencoder. Their encodings are, therefore, used in data compression and data transformation. For example, classifiers on high-dimensional data are sometimes trained on its encodings (a data transformation).

7.3.2 Overcomplete

While undercomplete autoencoder does dimension reduction and learns the most important features, it sometimes fails to learn the true underlying process. It fails when the encoder and decoder are given too much capacity, i.e., a lot to learn in a small dimension when k is small.

An overcomplete autoencoder overcomes the issue by allowing the encoding dimension to be equal to or larger than the input. This eases the over capacity issue.

A large encoding may appear counter-intuitive to the common understanding of autoencoders. It does not offer dimension reduction! On the contrary, we have more. Even worse, an overcomplete autoencoder can easily learn to become an identity transform, i.e., a trivial model with $W^{(e)} = I_{p\times p}$.

To avoid triviality, an overcomplete autoencoder is regularized. A nonlinear and regularized overcomplete autoencoder can effectively learn the underlying data distribution due to its larger capacity.

A simple overcomplete autoencoder is similar to an undercomplete autoencoder in Equation 7.7 except $k \geq p$. The difference is in the loss function. Overcomplete autoencoders **necessarily** have regularization included in them. For example,

$$L(\boldsymbol{x}, \hat{\boldsymbol{x}}) + \lambda_1 ||\boldsymbol{z}||_1 + \lambda_2 \left|\left| \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} \right|\right|_F^2 \tag{7.9}$$

where $L(\boldsymbol{x}, \hat{\boldsymbol{x}})$ is typically a mean squared error to keep the model output $\hat{\boldsymbol{x}}$ close to the input $\boldsymbol{x}$.

The second term $\lambda_1 ||\boldsymbol{z}||_1$ encourages sparse encodings $\boldsymbol{z}$. This is discussed more in § 7.3.5 on **sparse autoencoders**. Besides, it is argued in Goodfellow, Bengio, and Courville 2016 that the L_1-norm[1] sparsity penalty is not a "regularization" term. Instead, it is a way to approximately train a generative model.

The third term $\left|\left|\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}\right|\right|_F^2$ makes the derivative of the encodings with respect to the input smaller. This forces the autoencoder to learn a function that is less sensitive to minor changes in $\boldsymbol{x}$. An autoencoder with derivative regularization is called a **contractive autoencoder** discussed in § 7.3.4.

With these regularization properties taken from other autoencoders, an overcomplete autoencoder learns useful abstraction of the data distribution even if its capacity is large enough to learn a trivial identity function.

Besides, remember that Equation 7.9 is only an example of overcomplete model loss function. An overcomplete autoencoder can work with any one of the two or other types of regularization as well.

7.3.3 Denoising Autoencoder (DAE)

Observing noisy data is common in several problems. One application of autoencoders is to denoise the data. A specially designed *denoising* autoencoder serves this purpose.

A simple denoising autoencoder appears as

$$\boldsymbol{x} + \boldsymbol{\epsilon} \rightarrow f(\boldsymbol{x} + \boldsymbol{\epsilon}) \rightarrow \boldsymbol{z} \rightarrow g(\boldsymbol{z}) \rightarrow \hat{\boldsymbol{x}}, \tag{7.10}$$

where $\boldsymbol{\epsilon}$ is a noise added to the input.

As shown in Equation 7.10, a denoising autoencoder deliberately adds noise denoted with ϵ in the input. The model is then trained to reconstruct the original $\boldsymbol{x}$ from its noisy version $\boldsymbol{x} + \boldsymbol{\epsilon}$. The loss function

[1]An L_1-norm is $||\boldsymbol{z}||_1 = \sum_i |z|_i$.

is, therefore, defined as

$$L(\boldsymbol{x}, g(f(\boldsymbol{x} + \boldsymbol{\epsilon}))). \tag{7.11}$$

In minimizing the loss, a denoising autoencoder learns to skim denoised data from noise. It is shown in Bengio et al. 2013; Alain and Bengio 2014 the model implicitly learns the structure of the data distribution $p_{data}(\boldsymbol{x})$.

Besides, a denoising autoencoder can be constructed as an overcomplete high-capacity autoencoder. It is because its loss function becomes

$$\mathbb{E}(||\boldsymbol{x} - \hat{\boldsymbol{x}}||^2) + \sigma^2 \mathbb{E}\Big[\Big|\Big|\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}\Big|\Big|_F^2\Big] + o(\sigma^2) \tag{7.12}$$

if we have mean squared error criterion and the noise is additive, $\boldsymbol{x} \leftarrow \boldsymbol{x} + \boldsymbol{\epsilon}$, and gaussian, $\epsilon \sim N(0, \sigma)$ (refer to Alain and Bengio 2014).

7.3.4 Contractive Autoencoder (CAE)

A contractive autoencoder is trained to make encodings robust to noisy perturbations in the input. To encourage robustness, the *sensitivity* of $\boldsymbol{z}$ to any perturbations in the input is penalized during model training.

How to measure the sensitivity? The derivative of encodings with respect to the input makes an appropriate measure.

If the input and encoding are $\boldsymbol{x} \in \mathbb{R}^p$ and $\boldsymbol{z} \in \mathbb{R}^k$, respectively, then the derivative $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}$ is a $k \times p$ matrix. The matrix of derivatives is called a Jacobian denoted by J where

$$J_{ij} = \frac{\partial z_i}{\partial x_j}, \, i = 1, \ldots, k, \, j = 1, \ldots, p. \tag{7.13}$$

The overall sensitivity of a model encodings is estimated by the Frobenius-norm of J, which in simple terms is the sum of squares of every derivative J_{ij}. The norm is expressed as $||J||_F^2 = \sum_{ij} \left(\frac{\partial z_i}{\partial x_j}\right)^2$.

Consequently, the CAE loss function with the penalty on derivatives is

$$L(\boldsymbol{x}, \hat{\boldsymbol{x}}) + \lambda \Big|\Big| \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} \Big|\Big|_F^2 . \tag{7.14}$$

The objective of a contractive autoencoder is similar to a denoising autoencoder: make the encodings robust to noise. Alain and Bengio 2014 theoretically show that they are equivalent if denoising autoencoder is trained by adding small Gaussian noise to the input.

However, they achieve robustness differently. Contractive autoencoder forces the encodings $\boldsymbol{z}$ to resist perturbations in the input. On the other hand, a denoising autoencoder forces this on the inferences $\hat{\boldsymbol{x}}$.

In doing so, a contractive autoencoder learns **better** encodings to use in other tasks such as classification. Rifai, Vincent, et al. 2011 achieved state-of-the-art classification errors on a range of datasets using contractive autoencoders learned features to train MLPs.

> "The best classification accuracy usually results from applying the contractive penalty (for robustness to input perturbations) to the encodings $\boldsymbol{z}$ rather than to the inferences $\hat{\boldsymbol{x}}$,"
>
> –Goodfellow, Bengio, and Courville 2016.

Rifai, Vincent, et al. 2011 found that the Jacobian norm penalty in contractive autoencoder loss (Equation 7.14) carve encodings that correspond to a lower-dimensional non-linear manifold. Simply put, a manifold is a house of data. The objective of autoencoders is to find the shape of the smallest such house. In doing so, its encodings learn the essential characteristics of the data.

In simple terms, a manifold is a house of data. An autoencoder finds the shape of the smallest such house.

The encodings capture the local directions of the variations along the manifold while being more invariant to a majority of directions orthogonal to the manifold.

Think of this as the manifold house has data on its walls. Variations along the walls should be captured. But variations perpendicular to a wall (and on the outside) is noise. The model should, therefore, be invariant to them.

Rifai, Mesnil, et al. 2011 went further to add a penalty on the second derivative of encodings $\frac{\partial^2 \boldsymbol{z}}{\partial \boldsymbol{x}^2}$ called the Hessian to develop a higher-order contractive autoencoder. While the first derivative Jacobian penalty ensures robustness to small noise, the Hessian penalty extends this robustness to larger noise.

From a manifold perspective, the Hessian penalty penalizes curvature. This means the manifold house will have smoother walls. Consequently, the noisier data falls out of it easily.

With its benefits, a practical issue with the contractive autoencoders is that although the Jacobian or Hessian is cheap to compute in single hidden layer autoencoders, it becomes expensive in deeper autoencoders. Rifai, Vincent, et al. 2011 addressed the issue by separately training a series of single-layer autoencoders stacked to make a deep autoencoder. This strategy makes each layer locally contractive and, consequently, the deep autoencoder contractive.

7.3.5 Sparse Autoencoder

A sparse autoencoder imposes sparsity on the encodings $\boldsymbol{z}$. The sparse encodings are typically drawn to perform another task, such as classification.

For instance, suppose we have labeled data $(\boldsymbol{x}, y)$ from a high-dimensional process. Learning a classifier on high-dimensional data is challenging. However, a sparse autoencoder can extract the relevant information along with reducing the classification features dimension. An example of such a model is

$$\begin{tikzcd}
\boldsymbol{x} \arrow[r] & f(\boldsymbol{x}) \arrow[r] & \boldsymbol{z} \arrow[r] \arrow[d] & g(\boldsymbol{z}) \arrow[r] & \hat{\boldsymbol{x}} \\
& & {} \arrow[r] & h(\boldsymbol{z}) \arrow[r] & \hat{y}
\end{tikzcd} \tag{7.15}$$

where $\hat{\boldsymbol{x}}$ is the reconstructed $\boldsymbol{x}$ and $\hat{y}$ is the predicted label from the encodings $\boldsymbol{z}$, and f, g, h, denote encoder, decoder, and classifier, respectively.

Its loss function is expressed as

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))) + \lambda||\boldsymbol{z}||_1 \tag{7.16}$$

where the penalty is an L_1-norm $\lambda||\boldsymbol{z}||_1 = \lambda\sum_i |z_i|$.

The encoding size can be smaller, equal, or larger than the input. It can vary based on the problem. Regardless, the sparsity penalty in a sparse autoencoder extracts essential statistical features in the input data.

This property is guaranteed because the sparsity penalty results in the autoencoder approximate a generative model. It is straightforward to show that the sparsity penalty is arrived at by framing a sparse autoencoder as a generative model that has latent variables (refer to § 14.2.1 in Goodfellow, Bengio, and Courville 2016). The latent variables here are the encodings $\boldsymbol{z}$.

Sparsity penalty in an autoencoder is not a "regularization" term. It is a way of approximating a generative model.

Approximating a generative model with an autoencoder ensures the encodings are useful, i.e., they describe the latent variables that explain the data.

Besides, the loss function in Equation 7.16 is if the autoencoder is trained independent of the classification task. The loss function can be modified to include classification. For example, the classifier and autoencoder in Equation 7.15 can be trained simultaneously by including the classification error in the loss as

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))) + L'(y, h(\boldsymbol{z})) + \lambda||\boldsymbol{z}||_1 \tag{7.17}$$

where $L'(y, h(\boldsymbol{z}))$ penalizes differences between the actual and predicted labels such as cross-entropy.

The former approach has two-stage learning. The encodings are learned in the first stage by minimizing Equation 7.16 independent of the classification task. The encodings act as latent features to train a classifier in the second stage. This approach is tractable and, therefore, easier to train. However, the encodings can sometimes be ineffective in the classifier.

The simultaneous learning approach by minimizing loss in Equation 7.17 ensures the autoencoder learns the latent features such that they are capable of classifying. However, it lacks tractability.

Sparse autoencoders learn relevant information in a reduced dimension useful in other tasks such as classification.

7.4 Anomaly Detection with Autoencoders

Anomaly detection is unarguably one of the best approaches in rare event detection problems. Especially, if the event is so rare that there are insufficient samples to train a classifier. Fortunately, an (extremely) rare event often appears as an anomaly in a process. They can, therefore, be detected due to their abnormality.

Petsche et al. 1996 have one of the early works in deep learning which developed anomaly detectors for rare event detection. They developed an “autoassociator” to detect an imminent motor failure.

The “autoassociator” was essentially a reconstructor. Petsche et al.

1996 showed that the autoassociator has a small reconstruction error on measurements recorded from healthy motors but a larger error on those recorded from faulty motors.

This difference in the reconstruction errors: small for normal conditions and large for a rare anomaly forms the basis of anomaly detection models for rare events.

This section presents the anomaly detection approach for rare event detection using an autoencoder. First, the approach is explained in § 7.4.1. Then an autoencoder construction and application for rare event prediction is given in § 7.4.2-7.4.5.

7.4.1 Anomaly Detection Approach

If an event is extremely rare, we can use an **anomaly detection approach** to predict the rare event. In this approach, the rare event is treated as anomalies. The model is trained to detect the anomaly.

Given a data set $(\boldsymbol{x}_t, y_t), t = 1, \ldots, T$, where y_t corresponds to the event and $\boldsymbol{x}_t \in \mathbb{R}^p$ are the process variables at time t with p dimensions. The problem is to detect a rare event, if one occurred, at a time t' when $t' > T$.

At time $t' > T$, the event $y_{t'}$ is unknown. For example, if a bank fraud happened at t' it may remain undetected until reported by the account holder. By that time, a loss is likely to be already incurred. Instead, if the fraud was detected at t', the fraudulent transaction could be withheld.

The anomaly detection works using a *reconstruction model* approach for detection. The model learns to reconstruct the process variables $\boldsymbol{x}_t$ given itself, i.e., the model is $\hat{\boldsymbol{x}}_t|\boldsymbol{x}_t$.

The approach can be broken down as follows.

1. **Training**
   (a) Divide the process data into two parts: majority class (negatively labeled), $\{\boldsymbol{x}_t, \forall t | y_t = 0\}$, and minority class (positively labeled), $\{\boldsymbol{x}_t, \forall t | y_t = 1\}$.
   (b) The majority class is treated as a normal state of a process.

The normal state is when the process is event-less.

(c) Train a reconstruction model on the normal state samples $\{\boldsymbol{x}_t, \forall t | y_t = 0\}$, i.e., ignore the positively labeled minority data.

2. **Inferencing**

   (a) A well-trained reconstruction model will be able to accurately reconstruct a new sample $\boldsymbol{x}_{t'}$ if it belongs to the normal state. It will, therefore, have a small reconstruction error $||\boldsymbol{x}_{t'} - \hat{\boldsymbol{x}}_{t'}||_2^2$.

   (b) However, a sample during a rare-event would be abnormal for the model. The model will struggle to reconstruct it. Therefore, the reconstruction error will be large.

   (c) Such an instance of high reconstruction error is called out as a rare event occurrence.

7.4.2 Data Preparation

As usual, we start model construction by loading the libraries in Listing 7.1.

Listing 7.1. Load libraries for dense reconstruction model

```
import numpy as np
import pandas as pd

%tensorflow_version 2.x
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Layer
from tensorflow.keras.layers import InputSpec
from tensorflow.keras.callbacks import
    ModelCheckpoint
from tensorflow.keras.callbacks import TensorBoard
from tensorflow.keras import regularizers
from tensorflow.keras import activations
from tensorflow.keras import initializers
```

```python
from tensorflow.keras import constraints
from tensorflow.keras import Sequential
from tensorflow.keras import backend as K
from tensorflow.keras.constraints import UnitNorm
from tensorflow.keras.constraints import Constraint
from tensorflow.python.framework import tensor_shape

from numpy.random import seed
seed(123)

from sklearn import datasets
from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import scipy

# User-defined library
import utilities.datapreprocessing as dp
import utilities.reconstructionperformance as rp
import utilities.simpleplots as sp
```

In the anomaly detection approach, an autoencoder is trained on the "normal" state of a process which makes the majority class. In the dataset, those are the `0` labeled samples. They are, therefore, separated for training and a `scaler` is fitted on it. The rest of the validation and test sets are scaled with it.

Listing 7.2. Data preparation for rare event detection

```python
# The data is taken from https://arxiv.org/abs
    /1809.10717. Please use this source for any
    citation.

df = pd.read_csv("data/processminer-sheet-break-rare
    -event-dataset.csv")
df.head(n=5)  # visualize the data.

# Hot encoding
hotencoding1 = pd.get_dummies(df['Grade&Bwt'])
hotencoding1 = hotencoding1.add_prefix('grade_')
hotencoding2 = pd.get_dummies(df['EventPress'])
hotencoding2 = hotencoding2.add_prefix(
    'eventpress_')

df = df.drop(['Grade&Bwt', 'EventPress'],
             axis=1)

df = pd.concat([df, hotencoding1, hotencoding2],
               axis=1)

# Rename response column name for ease of
  understanding
df = df.rename(columns={'SheetBreak': 'y'})

# Sort by time.
df['DateTime'] = pd.to_datetime(df.DateTime)
df = df.sort_values(by='DateTime')

# Shift the response column y by 2 rows to do a 4-
  min ahead prediction.
df = dp.curve_shift(df, shift_by=-2)

# Drop the time column.
df = df.drop(['DateTime'], axis=1)

# Split the data and scale

DATA_SPLIT_PCT = 0.2
SEED = 123
df_train, df_test =
    train_test_split(df,
                     test_size=DATA_SPLIT_PCT,
                     random_state=SEED)
df_train, df_valid =
    train_test_split(df_train,
                     test_size=DATA_SPLIT_PCT,
                     random_state=SEED)

df_train_0 = df_train.loc[df['y'] == 0]
df_train_1 = df_train.loc[df['y'] == 1]
df_train_0_x = df_train_0.drop(['y'], axis=1)
df_train_1_x = df_train_1.drop(['y'], axis=1)

df_valid_0 = df_valid.loc[df['y'] == 0]
```

```
df_valid_1 = df_valid.loc[df['y'] == 1]
df_valid_0_x = df_valid_0.drop(['y'], axis=1)
df_valid_1_x = df_valid_1.drop(['y'], axis=1)

df_test_0 = df_test.loc[df['y'] == 0]
df_test_1 = df_test.loc[df['y'] == 1]
df_test_0_x = df_test_0.drop(['y'], axis=1)
df_test_1_x = df_test_1.drop(['y'], axis=1)

scaler = StandardScaler().fit(df_train_0_x)
df_train_0_x_rescaled =
    scaler.transform(df_train_0_x)
df_valid_0_x_rescaled =
    scaler.transform(df_valid_0_x)
df_valid_x_rescaled =
    scaler.transform(df_valid.drop(['y'],
                                 axis = 1))

df_test_0_x_rescaled =
    scaler.transform(df_test_0_x)
df_test_x_rescaled =
    scaler.transform(df_test.drop(['y'],
                                 axis = 1))
```

7.4.3 Model Fitting

An undercomplete autoencoder is constructed here. Its configurations are as follows,

- A single layer encoder and decoder.
- The encoding size is taken as approximately half of the input features. The size is chosen from the geometric series of 2 which is closest to the half.
- The encoding layer is `relu` activated.
- The encoding weights are regularized to encourage orthogonality (see § 7.7.3).
- The encodings are regularized to encourage a sparse covariance (see § 7.7.4).

- The encoding and decoding weights are constrained to have a unit norm. The encoding weights are normalized along the **rows** axis while the decoding weights on the **columns** axis.
- The decoding layer is `linear` activated. It is mandatory to have it linearly activated because it reconstructs the input that ranges in $(-\infty, \infty)$.
- The loss function is mean squared error that is appropriate because $\boldsymbol{x}$ is unbounded.
- The input `x` and output `y` in the `.fit()` function are the same in an autoencoder as the objective is to reconstruct $\boldsymbol{x}$ from itself.

Listing 7.3. Autoencoder model fitting

```
# Autoencoder for rare event detection

input_dim = df_train_0_x_rescaled.shape[1]

encoder = Dense(units=32,
                activation="relu",
                input_shape=(input_dim,),
                use_bias = True,
                kernel_regularizer=
                  OrthogonalWeights(
                    weightage=1.,
                    axis=0),
                kernel_constraint=
                  UnitNorm(axis=0),
                activity_regularizer=
                  SparseCovariance(weightage=1.),
                     name='encoder')

decoder = Dense(units=input_dim,
                activation="linear",
                use_bias = True,
                kernel_constraint=
                  UnitNorm(axis=1), name='decoder')

autoencoder = Sequential()
autoencoder.add(encoder)
autoencoder.add(decoder)

autoencoder.summary()
autoencoder.compile(metrics=['accuracy'],
                    loss='mean_squared_error',
                    optimizer='adam')

history = autoencoder.fit(x=df_train_0_x_rescaled,
                    y=df_train_0_x_rescaled,
                    batch_size=128,
                    epochs=100,
                    validation_data=
                        (df_valid_0_x_rescaled,
                         df_valid_0_x_rescaled),
                    verbose=0).history
```

7.4.4 Diagnostics

A few simple diagnostics are done on the fitted autoencoder. In Listing 7.4, it is shown that the dot product of encoding weights are nearly an identity matrix. That shows the effect of the `OrthogonalWeights()` regularizer.

The listing also shows that the weights have an exact unit-norm. A unit-norm constraint is relatively simpler to impose than orthogonality. Unit-norm does not significantly impact learning. It is simply normalizing any estimated set of weights. Unlike orthogonality, this constraint can, therefore, be a hard-constraint[2].

Listing 7.4. Near orthogonal encoding weights

```
# Near orthogonal encoding weights
w_encoder = autoencoder.get_layer('encoder').
   get_weights()[0]
print('Encoder weights dot product\n',
      np.round(np.dot(w_encoder.T, w_encoder), 2))

# Encoder weights dot product
```

[2]A hard-constraint is strictly applied while a soft-constraint is encouraged to be present in a model.

```
#   [[ 1.   0.  -0.  ...  -0.   0.  -0.]
#   [ 0.   1.   0.  ...   0.  -0.   0.]
#   [-0.   0.   1.  ...  -0.   0.   0.]
#   ...
#   [-0.   0.  -0.  ...   1.  -0.  -0.]
#   [ 0.  -0.   0.  ...  -0.   1.   0.]
#   [-0.   0.   0.  ...  -0.   0.   1.]]

# Encoding weights have unit norm
w_encoder = np.round(autoencoder.get_layer('encoder'
   ).get_weights()[0], 2).T
print('Encoder weights norm, \n',
      np.round(np.sum(w_encoder ** 2, axis = 1), 1))

# Encoder weights norm,
#   [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
#     1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
#     1. 1. 1. 1. 1. 1.]
```

After this, Listing 7.5 shows the covariance sparsity of the learned encodings. That ensures the encodings have less redundant information.

Listing 7.5. Sparse covariance of encodings

```
# Nearly-uncorrelated encoded features
encoder_model = Model(inputs=autoencoder.inputs,
                      outputs=autoencoder.get_layer(
                         'encoder').output)
encoded_features = pd.DataFrame(encoder_model.
   predict(df_train_0_x_rescaled))

print('Encoded feature correlations\n',
      np.round(encoded_features.corr(), 2))

# Encoded feature correlations
#          0      1      2   ...     29     30     31
# 0     1.00  -0.01  -0.08   ...   0.04   0.03  -0.12
# 1    -0.01   1.00   0.05   ...  -0.01   0.06  -0.02
# 2    -0.08   0.05   1.00   ...   0.11  -0.02   0.10
# 3     0.00  -0.03  -0.04   ...   0.17   0.04  -0.02
# ...
# 28   -0.02   0.09   0.15   ...  -0.10  -0.16  -0.03
# 29    0.04  -0.01   0.11   ...   1.00  -0.01  -0.08
# 30   0.03   0.06 -0.02   ...  -0.01   1.00   0.09
# 31  -0.12  -0.02   0.10   ...  -0.08   0.09   1.00
# [32 rows x 32 columns]
```

7.4.5 Inferencing

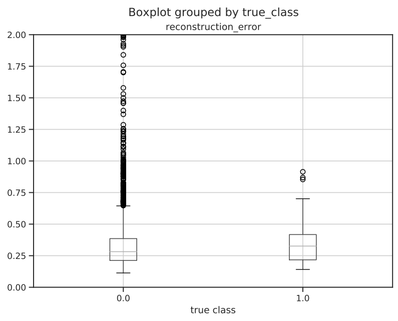
The inferencing for rare event prediction using an autoencoder is made by classifying the reconstruction errors as high or low. A high reconstruction error indicates the sample is anomalous to the normal process and, therefore, inferred as a rare event.

Listing 7.6. Autoencoder inferencing for rare event prediction

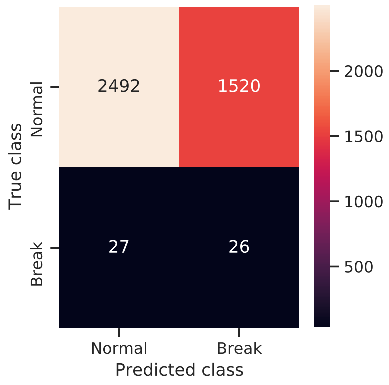
```
# Inferencing
error_vs_class_valid =
    rp.reconstructionerror_vs_class(
        model=autoencoder,
        sample=df_valid_x_rescaled,
        y=df_valid['y'])

# Boxplot
plt, fig = rp.error_boxplot(
    error_vs_class=error_vs_class_valid)

# Prediction confusion matrix
error_vs_class_test =
    rp.reconstructionerror_vs_class(
        model=autoencoder,
        sample=df_test_x_rescaled,
        y=df_test['y'])
conf_matrix, fig =
    rp.model_confusion_matrix(
        error_vs_class=error_vs_class_test,
        threshold=errors_valid1.quantile(0.50))
```

To determine the threshold of high vs low, boxplot statistics of the reconstruction error is determined for the positive and negative samples in the validation set. The boxplots are shown in Figure 7.4a.

From them, the 50-percentile of the positive sample errors is taken as the threshold for inferring a test sample as a rare event. The confusion matrix from test set predictions is shown in Figure 7.4b.



(a) *Boxplot.*



(b) *Confusion matrix.*

Figure 7.2. *Dense undercomplete autoencoder inferencing results. The boxplot of reconstruction errors for positive and negative samples in the validation set is at the top. The bottom figure shows the confusion matrix of the test inferences made based on the error threshold from the validation set.*

The test recall is as high as $\sim 50\%$ but at the cost of a high false-positive rate $\sim 38\%$. The f1-score is, therefore, as small as $\sim 3\%$.

A high false-positive rate may be undesirable in some problems. Another approach of using encodings learned from an autoencoder in a feed-forward classifier typically addresses the issue and shown in the next section.

7.5 Feed-forward MLP on Sparse Encodings

In this section, first, an overcomplete sparse autoencoder is constructed in § 7.5.1. Its encodings are then used in a feed-forward MLP classifier in § 7.5.2. This section shows that sparse autoencoders can learn useful features and help improve classification tasks.

7.5.1 Sparse Autoencoder Construction

A sparse autoencoder described in § 7.3.5 is constructed here. The model is overcomplete with the encoding dimension equal to the input dimension.

The sparsity regularization is imposed on the encodings by setting `activity_regularizer=tf.keras.regularizers.L1(l1=0.01)` in the model construction in Listing 7.7.

Listing 7.7. Sparse Autoencoder to derive predictive features

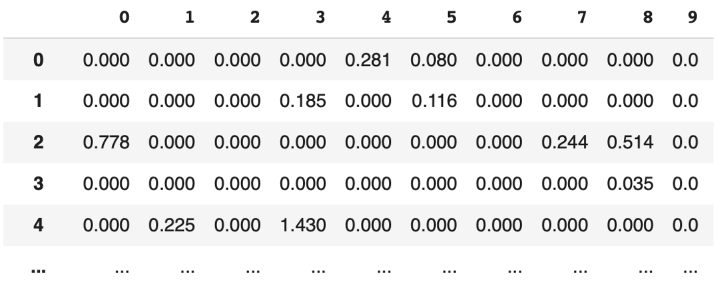
```python
# Sparse Autoencoder for Rare Event Detection
input_dim = df_train_0_x_rescaled.shape[1]

encoder = Dense(units=input_dim,
                activation="relu",
                input_shape=(input_dim,),
                use_bias = True,
                kernel_constraint=
                  UnitNorm(axis=0),
                activity_regularizer=
                  tf.keras.regularizers.L1(
                      l1=0.01),
                name='encoder')

decoder = Dense(units=input_dim,
                activation="linear",
                use_bias = True,
                kernel_constraint=
                  UnitNorm(axis=1), name='decoder')

sparse_autoencoder = Sequential()
sparse_autoencoder.add(encoder)
sparse_autoencoder.add(decoder)

sparse_autoencoder.summary()
sparse_autoencoder.compile(
    metrics=['accuracy'],
    loss='mean_squared_error',
    optimizer='adam')

history = sparse_autoencoder.fit(
    x=df_train_0_x_rescaled,
    y=df_train_0_x_rescaled,
    batch_size=128,
    epochs=100,
    validation_data=(df_valid_0_x_rescaled,
                     df_valid_0_x_rescaled),
    verbose=0).history
```

This regularization makes the autoencoder's loss function as in Equation 7.16. Due to the sparsity penalty in loss, the encoder weights shown in Listing 7.8 are not a trivial identity function even though the autoencoder is overcomplete (refer to the issue of trivial encodings in § 7.3.2).

Listing 7.8. Encoder weights are not identity in an overcomplete sparse autoencoder

```
# Weights on encoder
w_encoder = np.round(np.transpose(
    autoencoder.get_layer('encoder').get_weights()
        [0]), 3)

# Encoder weights
#   [[ 0.088  0.063  0.009 ...  0.096 -0.261  0.103]
#   [ 0.076  0.227  0.071 ...  0.083 -0.007  0.094]
#   [ 0.055 -0.1   -0.218 ... -0.019 -0.067  0.011]
#   ...
#   [ 0.039 -0.042  0.078 ... -0.216 -0.066  0.178]
#   [ 0.088 -0.169  0.084 ... -0.333 -0.109  0.171]
#   [ 0.057  0.081 -0.086 ...  0.26   0.201 -0.1  ]]
```

Moreover, the effect of the encodings sparsity penalty is visible in Figure 7.3. The figure shows several encodings are pushed to zero. In doing so, the autoencoder learns only the essential data features.



	0	1	2	3	4	5	6	7	8	9
0	0.000	0.000	0.000	0.000	0.281	0.080	0.000	0.000	0.000	0.0
1	0.000	0.000	0.000	0.185	0.000	0.116	0.000	0.000	0.000	0.0
2	0.778	0.000	0.000	0.000	0.000	0.000	0.000	0.244	0.514	0.0
3	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.035	0.0
4	0.000	0.225	0.000	1.430	0.000	0.000	0.000	0.000	0.000	0.0
...	...	...	...	...	...	...	...	...	...	...

Figure 7.3. *Encodings of the sparse autoencoder.*

The sparse encodings did prove to improve the model's performance. The boxplots in Figure 7.4a show a better separation of positive and negative samples compared to the previous model. The confusion matrix in Figure 7.4b also shows an improvement.

With the sparse autoencoder, the recall increased to $\sim$ 60%, the false-positive rate decreased to $\sim$ 34%, and f1-score increased to $\sim$ 4%. The false-positive rate is still high. The next section shows a feed-forward MLP on the sparse encodings to resolve the issue.

7.5.2 MLP Classifier on Encodings

The encodings learned in sparse autoencoders typically make excellent input features to a feed-forward classifier in deep learning.

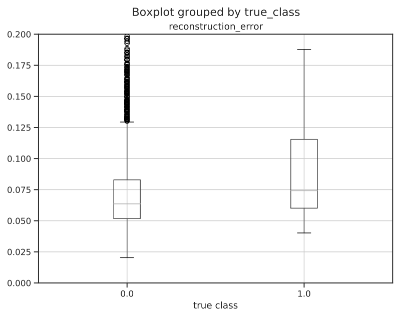
In this section, a two-stage approach: 1. learn a sparse encoder, and 2. train a classifier on the encodings, is demonstrated.

Listing 7.9 first derives the encodings of the training, validation, and test $\boldsymbol{x}$'s by passing them through the sparse encoder learned in the previous § 7.5.1.

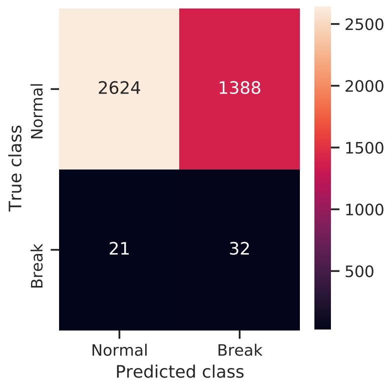
An MLP model with the same architecture as the baseline MLP in § 4.4 is constructed except that the network is trained on the encodings.

Listing 7.9. MLP feed-forward classifier on sparse encodings for rare event prediction

```
# Classifier on the sparse encodings

```



(a) *Boxplot.*



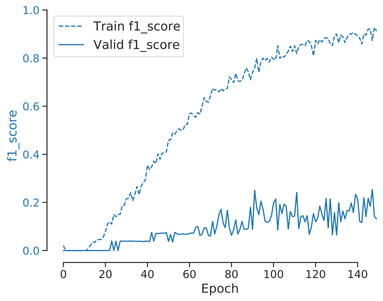
(b) *Confusion matrix.*

Figure 7.4. *Sparse overcomplete autoencoder inferencing results. The boxplot (top) shows the sparse autoencoder could better separate positive and negative samples. The confusion matrix (bottom) confirms it with an improvement in the accuracy measures.*

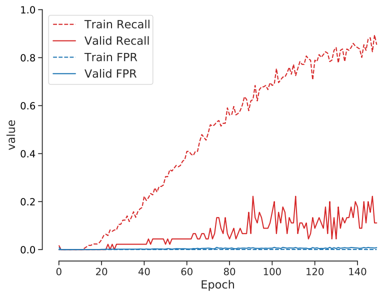
```
# Data preparation
import utilities.performancemetrics as pm
df_train_x = df_train.drop(['y'], axis=1)
df_train_y = df_train['y'].values

df_train_x_rescaled = scaler.transform(df_train_x)

df_valid_y = df_valid['y'].values
df_test_y = df_test['y'].values

```

```
# Obtain encodings to use as features in classifier
encoder_model =
    Model(inputs=sparse_autoencoder.inputs,
          outputs=sparse_autoencoder.get_layer('
             encoder').output)
X_train_encoded_features =
    encoder_model.predict(df_train_x_rescaled)
X_valid_encoded_features =
    encoder_model.predict(df_valid_x_rescaled)
X_test_encoded_features =
    encoder_model.predict(df_test_x_rescaled)

# Model
classifier = Sequential()
classifier.add(Input(
    shape=(X_train_encoded_features.shape[1], )))
classifier.add(Dense(units=32,
                     activation='relu'))
classifier.add(Dense(units=16,
                     activation='relu'))
classifier.add(Dense(units=1,
                     activation='sigmoid'))

classifier.compile(optimizer='adam',
            loss='binary_crossentropy',
            metrics=['accuracy',
                     tf.keras.metrics.Recall(),
                     pm.F1Score(),
                     pm.FalsePositiveRate()]
           )

history = classifier.fit(
    x=X_train_encoded_features,
    y=df_train_y,
    batch_size=128,
    epochs=150,
    validation_data=(X_valid_encoded_features,
                     df_valid_y),
    verbose=0).history
```

The model's f1-score, recall, and FPR are shown in Figure 7.5a and 7.5b. These accuracy measures when compared with the MLP model results on original data in Figure 4.5b and 4.5c show a clear improvement. The f1-score increased from $\sim 10\%$ to $\sim 20\%$, recall increased from $\sim 5\%$ to $\sim 20\%$, and false-positive remained close to zero.



(a) *F1-score.*



(b) *Recall and FPR.*

Figure 7.5. *MLP feed-forward classifier on sparse encodings results.*

7.6 Temporal Autoencoder

A dense autoencoder encodes flat data. Sometimes it is required to encode them as sequences or time series. Sequence constructs such as LSTM and Convolution can be used to build autoencoders for them.

However, there are a few differences in this autoencoder construction compared to a dense autoencoder. They are discussed below along with constructing LSTM and convolutional autoencoders.

7.6.1 LSTM Autoencoder

An LSTM autoencoder can be used for sequence reconstruction, sequence-to-sequence translation (Sutskever, Vinyals, and Le 2014), or feature extraction for other tasks. The simple sparse autoencoder is constructed here in Listing 7.10 which can be modified based on the objective.

In this model,

- an overcomplete autoencoder is modeled.
- The model is split into `encoder` and `decoder` modules.

- The encoder has the same number of units as the number of input features (overcompleteness).
- The LSTM layer output in the encoder is flattened (in line 18). This is to provide encoded vectors. This is optional.
- The encoded vector is L_1-regularized in line 22 for sparsity.
- The encoder and decoder modules end with a linearly activated `Dense` layer. The ending dense layer is not for feature extraction. Instead, its task is similar to **calibration**. Complex layers such as LSTM and convolution yield features that may be in a different space than the original data. A linear dense layer brings these features back to the original space for reconstruction.
- The ending `Dense` layer in the decoder is applied directly to the LSTM sequence outputs. Flattening the sequences is unnecessary because the dense layer automatically estimates a weights tensor compatible with the shape of the sequences.

Listing 7.10. A sparse LSTM autoencoder

```
 1  # LSTM Autoencoder Model
 2  # Encoder
 3
 4  inputs = Input(shape=(TIMESTEPS,
 5                        N_FEATURES),
 6                 name='encoder-input')
 7
 8  x = LSTM(units=N_FEATURES,
 9           activation='tanh',
10           return_sequences=True,
11           name='encoder-lstm')(inputs)
12
13  # Shape info needed to build Decoder Model
14  e_shape = tf.keras.backend.int_shape(x)
15  latent_dim = e_shape[1] * e_shape[2]
16
17  # Generate the latent vector
18  x = Flatten(name='flatten')(x)
19  latent = Dense(units=latent_dim,
20                 activation='linear',
               activity_regularizer=
                  tf.keras.regularizers.L1(l1=0.01),
               name='encoded-vector')(x)

# Instantiate Encoder Model
encoder = Model(inputs=inputs,
                outputs=latent,
                name='encoder')
encoder.summary()

# Decoder
latent_inputs = Input(shape=(latent_dim,),
                      name='decoder_input')

x = Reshape((e_shape[1], e_shape[2]),
            name='reshape')(latent_inputs)

x = LSTM(units=N_FEATURES,
         activation='tanh',
         return_sequences=True,
         name='decoder-lstm')(x)

output = Dense(units=N_FEATURES,
               activation='linear',
               name='decoded-sequences')(x)

# Instantiate Decoder Model
decoder = Model(inputs=latent_inputs,
                outputs=output,
                name='decoder')
decoder.summary()

# Instantiate Autoencoder Model using Input and
    Output
autoencoder = Model(inputs=inputs,
                    outputs=decoder(inputs=encoder(
                       inputs)),
                    name='autoencoder')
autoencoder.summary()
```

Linearly activated dense layer in encoder and decoder is necessary for calibration.

7.6.2 Convolutional Autoencoder

An autoencoder that encodes data using convolutions is called a **convolutional autoencoder**. Unlike most other autoencoders, a convolutional autoencoder uses different types of convolution layers for encoding and decoding. This and other specifications of a convolutional autoencoder constructed in Listing 7.11 are given below.

- The encoder and decoder modules are encapsulated within definitions. The encapsulation approach makes it easier to construct more complex modules. Moreover, the benefit is in visualizing the network structure as shown in Figure 7.6.
- A sparsity constraint on the encodings (for a sparse autoencoder) is added in the encoder in line 44.
- The decoder in a convolutional autoencoder cannot be made with `Conv` layers. A convolution layer is meant for extracting abstract features. In doing so, it downsizes the input. A decoder, on the other hand, is meant to reconstruct the input by upsizing (encoded) features. This is possible with `ConvTranspose` layers, also known as the **deconvolutional** layer (refer Shi, Caballero, Theis, et al. 2016).
- Pooling should strictly be not used in the decoder. Pooling is for data summarization. But decoding is for reconstruction. Any summarization with pooling can obstruct reconstruction.
- Decoding with (transpose) convolutional layers sometimes causes feature inflation. A batch normalization layer is usually added to stabilize the reconstruction.
- Similar to an LSTM autoencoder and for the same reason, a `Dense` layer is added at the end of both encoder and decoder. Its purpose is to **calibrate** the encoding and decoding.

🔔 *`Pooling` and `Conv` layers should **not** be used in a decoder of convolutional autoencoders.*

🔔 *`ConvTranspose` is used in place of a `Conv` layer in a decoder of convolutional autoencoders.*

Listing 7.11. A sparse convolutional autoencoder

```
inputs = Input(shape=(TIMESTEPS,
                      N_FEATURES),
               name='encoder_input')

def encoder(inp):
  '''
  Encoder.

  Input
  inp    A tensor of input data.

  Process
  Extract the essential features of the input as
  its encodings by filtering it through
  convolutional layer(s). Pooling can also
  be used to further summarize the features.

  A linearly activated dense layer is added as the
  final layer in encoding to perform any affine
  transformation required. The dense layer is not
  for any feature extraction. Instead, it is only
  to make the encoding and decoding connections
  simpler for training.

  Output
  encoding   A tensor of encodings.
  '''

  # Multiple (conv, pool) blocks can be added here
  conv1 = Conv1D(filters=N_FEATURES,
                      kernel_size=4,
                      activation='relu',
                      padding='same',
                      name='encoder-conv1')(inp)
    pool1 = MaxPool1D(pool_size=4,
                         strides=1,
                         padding='same',
                         name='encoder-pool1')(conv1)

    # The last layer in encoding
    encoding = Dense(units=N_FEATURES,
                        activation='linear',
                        activity_regularizer=
                         tf.keras.regularizers.L1(l1
                             =0.01),
                        name='encoder')(pool1)

    return encoding

def decoder(encoding):
    '''
    Decoder.

    Input
    encoding    The encoded data.

    Process
    The decoding process requires a transposed
    convolutional layer, a.k.a. a deconvolution
    layer. Decoding must not be done with a
    regular convolutional layer. A regular conv
    layer is meant to extract a downsampled
    feature map. Decoding, on the other hand,
    is reconstruction of the original data from
    the downsampled feature map. A regular
    convolutional layer would try to extract
    further higher level features from
    the encodings instead of a reconstruction.

    For a similar reason, pooling must not be
    used in a decoder. A pooling operation is
    for summarizing a data into a few summary
```

```
  statistics which is useful in tasks such as
  classification. The purpose of decoding is
  the opposite, i.e., reconstruct the original
  data from the summarizations. Adding pooling
  in a decoder makes it lose the variations
  in the data and, hence, a poor reconstruction.

  If the purpose is only reconstruction, a
  linear activation should be used in decoding.
  A nonlinear activation is useful for
  predictive features but not for reconstruction.

  Batch normalization helps a decoder by
  preventing the reconstructions
  from exploding.

  Output
  decoding    The decoded data.

  '''

  convT1 = Conv1DTranspose(filters=N_FEATURES,
                           kernel_size=4,
                           activation='linear',
                           padding='same')(encoding)

  decoding = BatchNormalization()(convT1)

  decoding = Dense(units=N_FEATURES,
                   activation='linear',
                   name='decoder')(decoding)

  return decoding

autoencoder = Model(inputs=inputs,
                    outputs=decoder(encoder(inputs)))

autoencoder.summary()

autoencoder.compile(loss='mean_squared_error',
                    optimizer = 'adam')
```

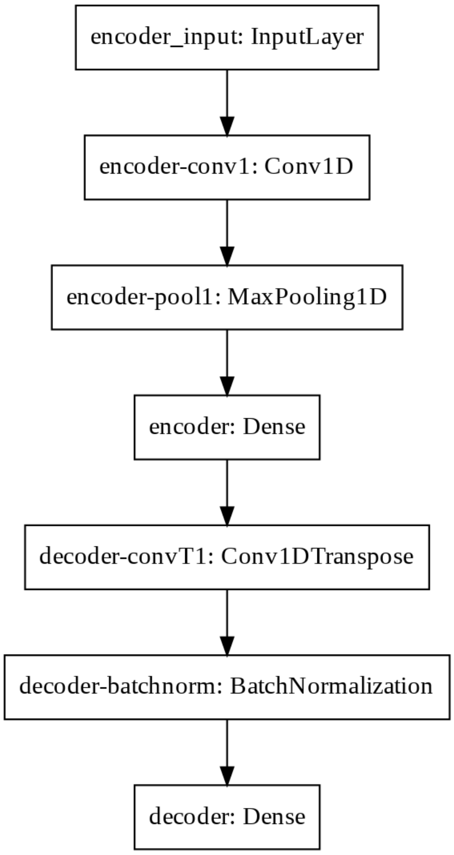

encoder_input: InputLayer
encoder-conv1: Conv1D
encoder-pool1: MaxPooling1D
encoder: Dense
decoder-convT1: Conv1DTranspose
decoder-batchnorm: BatchNormalization
decoder: Dense

Figure 7.6. *A baseline convolutional autoencoder.*

```

history = autoencoder.fit(x=X_train_y0_scaled,
                          y=X_train_y0_scaled,
                          epochs=100,
                          batch_size=128,
                          validation_data=
                              (X_valid_y0_scaled,
                               X_valid_y0_scaled),
                          verbose=1).history
```

A convolutional autoencoder can be used for image reconstruction, image denoising, or, like the other autoencoders, feature extraction for other tasks (Shi, Caballero, Huszár, et al. 2016).

In this chapter, three different programmatic paradigms in TensorFlow is used for constructing a dense, an LSTM, and a convolutional autoencoder. The paradigms are interchangeable for simple models. However, the functional approach used for the convolutional autoencoder in Listing 7.11 is preferable.

7.7 Autoencoder Customization

Autoencoders have proven to be useful for unsupervised and semi-supervised learning. Earlier, § 7.3 presented a variety of ways autoencoders can be modeled. Still, there is significant room for new development.

The section is intended for researchers seeking new development. It presents an autoencoder customization idea in § 7.7.1. A customized autoencoder is then constructed in § 7.7.2-7.7.4.

7.7.1 Well-posed Autoencoder

A mathematically well-posed autoencoder is easier to tune and optimize. The structure of a well-posed autoencoder can be defined from its relationship with principal component analysis.

As explained in the previous § 7.2.1 and 7.2.2, a linearly activated autoencoder approximates PCA. And, conversely, autoencoders are a nonlinear extension of PCA. In other words, an autoencoder extends PCA to a nonlinear space. Therefore, an Autoencoder should ideally have the properties of PCA. These properties are,

- **Orthonormal weights**. It is defined as follows for encoder weights,

$$W^T W = I, \text{and} \tag{7.18}$$

$$\sum_{j=1}^{p} w_{ij}^2 = 1, \; i = 1, \ldots, k \tag{7.19}$$

where I is a $p \times p$ identity matrix, p is the number of input features, and k is the number of nodes in an encoder layer.

- **Independent features**. The principal component analysis yields independent features. This can be seen by computing the covariance of the principal scores $Z = XW$,

$$\begin{aligned}\text{cov}(Z) &\propto (XW)^T(XW)\\ &= W^T X^T XW\\ &\propto W^T W \Lambda W^T W\\ &= \Lambda\end{aligned}$$

  where Λ is a diagonal matrix of eigenvalues and W are the eigenvectors. But autoencoder weights are **not** necessarily eigenvectors. Therefore, this property is not present by default and can be incorporated by adding a constraint,

$$\text{correlation}(Z_{encoder}) = I. \tag{7.20}$$

- **Tied layer**. An autoencoder typically has an hour-glass like symmetric structure[3]. In such a network, there is a mirror-layer in the decoder for every layer in the encoder. These layers can be tied by their weights and activations as

$$W^{(-l)} = (W^{(l)})^T \tag{7.21}$$
$$f^{(-l)} = (f^{(l)})^{-1} \tag{7.22}$$

  where $W^{(l)}$ and $f^{(l)}$ are the weights and activation on the l-th encoder layer, and $W^{(-l)}$ and $f^{(-l)}$ are on its mirror layer in the decoder. The weights transpose relationship in Equation 7.21 comes from Equation 7.6. The activations' inverse relationship is required due to their nonlinearity.

[3]An autoencoder does not necessarily have an hour-glass like structure. The decoder can be structured differently than the encoder.

Tying the weights without the activations inverse relationship can cause poor reconstruction. Consider a simple autoencoder flow to understand this issue:

$$X \rightarrow \underbrace{f^{(l)}(XW^{(l)})}_{encoder} \rightarrow Z \rightarrow \underbrace{f^{(-l)}(ZW^{(-l)})}_{decoder} \rightarrow \hat{X}.$$

In this flow, if $W^{(-l)}$ becomes $W(^{(l)})^T$ then $f^{(-l)}$ must become the inverse of $f^{(l)}$. Otherwise, the model is improperly posed. For example, if $f^{(-l)}$ is the same as $f^{(l)}$ then the model will be expected to yield the estimation of the input X using the same weights learned for encoding and the same nonlinear activation but $X \rightarrow f^{(l)}(XW^{(l)}) \rightarrow Z \rightarrow f^{(l)}(Z(W^{(l)})^T) \not\rightarrow \hat{X}$

Therefore, layers can only be tied in presence of a nonlinear activation and its inverse. However, defining such activation is nontrivial. In its absence, layers can be tied only if they are linearly activated in which case we lose the multilayer and nonlinear benefits of autoencoders.

7.7.2 Model Construction

In this section, an autoencoder with sparse encoding representation covariance (the features independence property in § 7.7.1) and orthonormal weights regularization is constructed and fitted on a random data. The custom definitions for these properties and the regularization effects are then illustrated in § 7.7.3 and 7.7.4.

The model is constructed and fitted on random data. The encoder layer has `kernel_regularizer`, `kernel_constraint`, and `activity_regularizer` defined. These are discussed in the next sections.

Listing 7.12. An illustrative example of a regularized autoencoder.

```
# Generate Random Data for Testing
n_dim = 5

# Generate a positive definite
# symmetric matrix to be used as
# covariance to generate a random data.
cov = datasets.make_spd_matrix(n_dim,
                               random_state=None)

# Generate a vector of mean for generating the
    random data.
mu = np.random.normal(loc=0,
                      scale=0.1,
                      size=n_dim)

# Generate the random data, X.
n = 1000
X = np.random.multivariate_normal(mean=mu,
                                  cov=cov,
                                  size=n)

# Autoencoder fitted on random data
nb_epoch = 100
batch_size = 16
input_dim = X.shape[1]
encoding_dim = 4
learning_rate = 1e-3

encoder = Dense(units=encoding_dim,
                activation="relu",
                input_shape=(input_dim, ),
                use_bias = True,
                kernel_regularizer=
                    OrthogonalWeights(
                        weightage=1.,
                        axis=0),
                kernel_constraint=
                    UnitNorm(axis=0),
                activity_regularizer=
                    SparseCovariance(
                        weightage=1.), name='encoder
                            ')

decoder = Dense(units=input_dim,
                activation="linear",
                use_bias = True,
                kernel_constraint=
                         UnitNorm(axis=1), name='decoder'
                             )

autoencoder = Sequential()
autoencoder.add(encoder)
autoencoder.add(decoder)

autoencoder.compile(metrics=['accuracy'],
                         loss='mean_squared_error',
                         optimizer='sgd')
autoencoder.summary()

autoencoder.fit(X, X,
                    epochs=nb_epoch,
                    batch_size=batch_size,
                    shuffle=True,
                    verbose=0)
```

7.7.3 Orthonormal Weights

Orthonormality of encoding weights is an essential property. Without it the model is ill-conditioned, i.e., a small change in the input can cause a significant change in the model.

As shown in Equation 7.18 and 7.19, this property has two parts: a. **orthogonality**, and b. **unit norm**.

The latter is easily incorporated in an encoder layer using `kernel_constraint=UnitNorm(axis=0)` constraint[4].

The former property can be incorporated with a custom regularizer defined in Listing 7.13. This is used in `kernel_regularizer` that acts as a soft constraint. Meaning, the estimated weights will be only **nearly** orthogonal.

The weights can be made strictly orthogonal by an input covariance decomposition. But due to decomposition computational complexity, a regularization method is preferred.

[4]In a decoder layer, the same constraint should be applied on the columns as `kernel_constraint=UnitNorm(axis=1)`.

Listing 7.13. A custom constraint for orthogonal weights.

```python
# Orthogonal Weights.
class OrthogonalWeights (Constraint):
    def __init__(self,
                 weightage = 1.0,
                 axis = 0):
        self.weightage = weightage
        self.axis = axis

    def weights_orthogonality(self,
                              w):
        if(self.axis==1):
            w = K.transpose(w)

        wTwminusI = K.dot(K.transpose(w), w) -
            tf.eye(tf.shape(w,
                    out_type=tf.float32)[1])

        return self.weightage * tf.math.sqrt(
            tf.math.reduce_sum(tf.math.square(
                wTwminusI)))

    def __call__(self, w):
        return self.weights_orthogonality(w)
```

The learned encoder and decoder weights for the autoencoder in Listing 7.12 is shown in Listing 7.14.

Listing 7.14. Encoder and decoder weights on an illustrative regularized autoencoder

```python
w_encoder = np.round(autoencoder.get_layer('encoder'
    ).get_weights()[0], 3)
w_decoder = np.round(autoencoder.get_layer('decoder'
    ).get_weights()[1], 3)
print('Encoder weights\n', w_encoder.T)
print('Decoder weights\n', w_decoder.T)

# Encoder weights
#  [[-0.301 -0.459  0.56  -0.033  0.619]
#  [-0.553  0.182 -0.439 -0.644  0.231]
#  [-0.036  0.209 -0.474  0.61   0.599]
#   [-0.426 -0.683 -0.365  0.288 -0.367]]
# Decoder weights
#   [[-0.146 -0.061  0.436 -0.72   0.517]
#   [-0.561  0.199 -0.431 -0.633  0.242]
#   [-0.059  0.681 -0.345  0.613  0.194]
#   [ 0.086 -0.769 -0.22   0.198 -0.56 ]]
```

The (near) orthogonality of encoding weights is shown in Listing 7.15. The weights are nearly orthogonal with small non-zero off-diagonal elements in $W^T W$. As mentioned earlier, the orthogonality is added as a soft-constraint due to which $W^T W \approx I$ instead of strict equality.

Listing 7.15. Encoder weights dot product show their near orthogonality. The orthogonality regularization is not applied on the decoder—its dot product is therefore not diagonal heavy

```
w_encoder = autoencoder.get_layer('encoder').
    get_weights()[0]
print('Encoder weights dot product\n',
      np.round(np.dot(w_encoder.T, w_encoder), 2))

w_decoder = autoencoder.get_layer('decoder').
    get_weights()[1]
print('Decoder weights dot product\n',
      np.round(np.dot(w_decoder.T, w_decoder), 2))

# Encoder weights dot product
#   [[1.   0.   0.   0.  ]
#   [0.   1.   0.01 0.  ]
#   [0.   0.01 1.   0.  ]
#   [0.   0.   0.   1.  ]]
# Decoder weights dot product
#   [[ 1.    0.46 -0.52 -0.49]
#   [ 0.46  1.   -0.02 -0.37]
#   [-0.52 -0.02  1.   -0.44]
#   [-0.49 -0.37 -0.44  1.  ]]
```

The orthogonality regularization is not added to the decoder. Due to that, the decoder weights dot product is not diagonally dominant.

Besides, the unit-norm constraint is added on both encoder and decoder weights. Their norms are shown in Listing 7.16.

Listing 7.16. Norm of encoder and decoder weights

```
print('Encoder weights norm, \n',
      np.round(np.sum(w_encoder ** 2,
                      axis = 0),
               2))
print('Decoder weights norm, \n',
      np.round(np.sum(w_decoder ** 2,
                      axis = 1),
               2))

# Encoder weights norm,
#   [1. 1. 1. 1.]
# Decoder weights norm,
#   [1. 1. 1. 1.]
```

It is important to note that decoder unit-norm is applied along the columns. Also, if the orthogonality regularization is to be added on decoder, it should be such that $WW^T \approx I$ unlike in encoder (where it is $W^TW \approx I$).

7.7.4 Sparse Covariance

Independence of the encoded features (uncorrelated features) is another desired property. It is because correlated encoding means we have redundant information spilled over the encoded dimensions.

However, unlike in PCA, weights orthogonality does not necessarily result in independent features. This is because $Z^TZ = W^TX^TXW \neq I$ even if $W^TW = I$. Orthogonal principal components, on the other hand, leads to independent features because they are drawn from the matrix decomposition of the input, i.e., $X^TX = W\Lambda W^T$ where Λ is a diagonal eigenvalues matrix.

If the orthogonal encoder weights were drawn by covariance decomposition of the input, the features will be independent. However, due to the computational complexity, the (near) feature independence is incorporated by regularization.

An approach to incorporate this is in Listing 7.17. Similar to weight orthogonality, the feature independence is incorporated as a soft constraint to have a sparse covariance.

Listing 7.17. A custom regularization constraint for encoded feature covariance sparsity

```python
# Near-Independent Features
class SparseCovariance (Constraint):

    def __init__(self, weightage=1.0):
        self.weightage = weightage

    # Constraint penalty
    def uncorrelated_feature(self, x):
        if(self.size <= 1):
            return 0.0
        else:
            output = K.sum(K.square(
                self.covariance -
                tf.math.multiply(self.covariance,
                                 tf.eye(self.size)))
                                    )
            return output

    def __call__(self, x):
        self.size = x.shape[1]
        x_centered = K.transpose(tf.math.subtract(
            x, K.mean(x, axis=0, keepdims=True)))

        self.covariance = K.dot(
            x_centered,
            K.transpose(x_centered)) / \
                tf.cast(x_centered.get_shape()[0],
            tf.float32)

        return self.weightage * self.
           uncorrelated_feature(x)
```

The covariance of the encoded random data is shown in Listing 7.18. As expected, the covariance is sparse. Moreover, similar to PCA behavior, most of the input data variability is taken by a small number of encodings. In this example, the last encoding dimension has more than 99% of the variance.

Listing 7.18. The covariance of encoded features

```
encoder_model = Model(inputs=autoencoder.inputs,
                      outputs=autoencoder.get_layer(
                         'encoder').output)
encoded_features = np.array(encoder_model.predict(X)
    )
print('Encoded feature covariance\n',
      np.round(np.cov(encoded_features.T), 3))

# Encoded feature covariance
#   [[ 0.01 -0.     0.    -0.02]
#   [-0.     0.     0.    -0.02]
#   [ 0.     0.     0.02 -0.02]
#   [-0.02 -0.02 -0.02  4.36]]
```

7.8 Rules-of-thumb

- **Autoencoder Construction**.
  - Add unit-norm constraint on the weights. This prevents ill-conditioning of the model.
  - Add a linearly activated dense layer at the end of the encoder and decoder for calibration in most autoencoders.
  - The activation on the decoder output layer should be based on the range of the input. For example, `linear` if the input $\boldsymbol{x}$ is in $(-\infty, \infty)$ (scaled with `StandardScaler`) or `sigmoid` if $\boldsymbol{x}$ is in $(0, 1)$ (scaled with `MinMaxScaler`).
- **Sparse autoencoder**.
  - A sparsity constraint should be added to the encoder's output. Typically, it is a dense layer. The sparsity can be added as `activity_regularizer=tf.keras.regularizers.L1(l1=0.01)`.
  - The encoding size should be equal to the original data dimension (overcomplete). The sparsity penalty ensures that the encodings are useful and not trivial.
  - Sparse encodings are best suited for use in other tasks such as classification.

- **Denoising autoencoder**.
  - Unlike sparse autoencoders, denoising autoencoders regularize the decoder output to make them insensitive to minor changes in the input.
  - Train a denoising autoencoder by adding small gaussian noise to the input. Ensure that the loss function minimizes the difference of the original data $\boldsymbol{x}$ with the decodings of the noisy data $g(f(\boldsymbol{x}+\boldsymbol{\epsilon}))$ where $\boldsymbol{\epsilon}$ is Gaussian(0,σ).
  - They are useful for denoising or reconstruction objectives. But their encodings are typically not useful for classification tasks.
- **LSTM autoencoder**.
  - Use `tanh` activation in the LSTM layers in both encoder and decoder.
  - Works better for translation tasks, for example, English to Spanish text translation. Typically, they do not work well for data reconstruction.
- **Convolutional autoencoder**.
  - Encoder module has a stack of `Conv` and `Pooling` layers. They perform summarization of the useful features of the data.
  - Decoder module has a stack of `ConvTranspose` and `BatchNormalization` layers.
  - Decoder module should **not** have `Conv` or `Pooling` layers.

7.9 Exercises

1. At the beginning of the chapter it is mentioned that autoencoders were conceptualized with inspiration from an older concept of principal component analysis (PCA) in statistics.

   (a) A PCA model is linear. It was mentioned in § 4.2 that a dense layer network is equivalent to a linear model if the activations on each layer are linear (proved in Appendix A). Extending this to a dense autoencoder, does it become the same as a principal component analysis (PCA) model if the activations are linear? Refer to § 7.7.1.

   (b) Under what conditions an autoencoder becomes equivalent to PCA? Refer to § 7.2 and § 7.7.1.

2. In § 7.6.1 and 7.6.2 it is mentioned that a linearly activated dense layer should be added at the end of encoder and decoder modules.

   (a) What is the benefit of the linear dense layers?

   (b) When is it not essential?

   (c) Is it more essential in decoder than encoder? Explain.

3. The chapter provides examples of shallow autoencoders. In building a deep autoencoder what would you do in the following scenario?

   (a) In constructing a Sparse autoencoder would you consider adding sparsity penalty on the activations (output) of the intermediate layers in the encoder versus only on the last encoder layer?

   (b) An autoencoder is essentially a special case of a feed-forward network where both the predictors and the response are the same. Looking at an autoencoder from this perspective sometimes makes it difficult to separate encoder and decoder. Essentially, to tell the boundary. This becomes even harder in a deep network. How would you distinguish where an encoder ends and a decoder begins?

4. In § 7.7.1 a few aspects of regularizing autoencoder by constraining encoder and decoder are discussed. Based on them,

   (a) Should you consider regularizing the encoder or decoder functions f and g, respectively? Why?

   (b) If following the structure of the principal component of analysis, what is the difference in constraining weights on encoder and decoder to be unit-norm in a dense autoencoder? Why?

   (c) Why is it difficult to strictly enforce weights orthogonality in an autoencoder? Why is it present by default in PCA? Refer to § 7.7.3.

   (d) Orthogonal weights in PCA leads to independent features. However, an autoencoder even with orthogonal encoder weights does not guarantee independent encodings. Why? The answer to this question also answers Q 1a.

5. The goal of an autoencoder is to learn the essential properties of the data while training to reconstruct the input. There are different types of regularization available to improve learning. Broadly, regularization can be applied to either the encoding $f(\boldsymbol{x})$ or the decoding $g(f(\boldsymbol{x}))$. Refer to § 7.3 and answer the following.

   (a) Among them, when is regularizing the encoding $f(\boldsymbol{x})$ better?

   (b) When is regularizing the decoder $g(f(\boldsymbol{x}))$ better?

   (c) (Optional) Refer to § 14.2.1 in Goodfellow, Bengio, and Courville 2016 to show that a sparse autoencoder approximates a generative model and that the sparsity penalty arrives as a result of this framework.

6. (Optional) The encoder module in an autoencoder can be incorporated in a classifier network. There are several ways it can be incorporated. Appendix K provides a flexible implementation to try different approaches.

   (a) Run the model in the appendix to train a classifier by transferring the encoder weights learned in autoencoder training to a classifier.

(b) Make the transferred encoder weights trainable. Note the change in the number of trainable parameters compared to in Q 6a. Train the model.

Bibliography

[AB14] Guillaume Alain and Yoshua Bengio. “What regularized autoencoders learn from the data-generating distribution”. In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 3563–3593.

[Adv06] Inc. Advanced Technology Services. *Downtime Costs Auto Industry $22k/Minute - Survey*. Mar. 2006. URL: `https://news.thomasnet.com/companystory/downtime-costs-auto-industry-22k-minute-survey-481017`.

[Bah57] RR Bahadur. “On unbiased estimates of uniformly minimum variance”. In: *Sankhyā: The Indian Journal of Statistics (1933-1960)* 18.3/4 (1957), pp. 211–224.

[Bas55] Dev Basu. “On statistics independent of a complete sufficient statistic”. In: *Sankhyā: The Indian Journal of Statistics (1933-1960)* 15.4 (1955), pp. 377–380.

[Bat+09] Iyad Batal et al. “Multivariate time series classification with temporal abstractions”. In: *Twenty-Second International FLAIRS Conference*. 2009.

[Bay+17] Inci M Baytas et al. “Patient subtyping via time-aware LSTM networks”. In: *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 2017, pp. 65–74.

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural machine translation by jointly learning to align and translate”. In: *arXiv preprint arXiv:1409.0473* (2014).

[Ben+13] Yoshua Bengio et al. “Generalized denoising auto-encoders as generative models”. In: *Advances in neural information processing systems.* 2013, pp. 899–907.

[BGC02] Piero Bonissone, Kai Goebel, and Yu-To Chen. “Predicting wet-end web breakage in paper mills”. In: *Working Notes of the 2002 AAAI symposium: Information Refinement and Revision for Decision Making: Modeling for Diagnostics, Prognostics, and Prediction.* 2002, pp. 84–92.

[BH89] Pierre Baldi and Kurt Hornik. “Neural networks and principal component analysis: Learning from examples without local minima”. In: *Neural networks* 2.1 (1989), pp. 53–58.

[Bis+95] Christopher M Bishop et al. *Neural networks for pattern recognition.* Oxford university press, 1995.

[Bis06] Christopher M Bishop. *Pattern recognition and machine learning.* springer, 2006.

[Bou+11] Y-Lan Boureau, Nicolas Le Roux, et al. “Ask the locals: multi-way local pooling for image recognition”. In: *2011 International Conference on Computer Vision.* IEEE. 2011, pp. 2651–2658.

[BPL10] Y-Lan Boureau, Jean Ponce, and Yann LeCun. “A theoretical analysis of feature pooling in visual recognition”. In: *Proceedings of the 27th international conference on machine learning (ICML-10).* 2010, pp. 111–118.

[BS13] Pierre Baldi and Peter J Sadowski. “Understanding dropout”. In: *Advances in neural information processing systems.* 2013, pp. 2814–2822.

[CB02] George Casella and Roger L Berger. *Statistical inference.* Vol. 2. Duxbury Pacific Grove, CA, 2002.

[Cho+14] Kyunghyun Cho et al. “Learning phrase representations using RNN encoder-decoder for statistical machine translation”. In: *arXiv preprint arXiv:1406.1078* (2014).

[CJK04] N Chawla, N Japkowicz, and A Kolcz. “Special issue on class imbalances”. In: *SIGKDD Explorations* 6.1 (2004), pp. 1–6.

[CN11] Adam Coates and Andrew Y Ng. "Selecting receptive fields in deep networks". In: *Advances in neural information processing systems.* 2011, pp. 2528–2536.

[CUH15] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and accurate deep network learning by exponential linear units (elus)". In: *arXiv preprint arXiv:1511.07289* (2015).

[Dai+17] Jifeng Dai et al. "Deformable convolutional networks". In: *Proceedings of the IEEE international conference on computer vision.* 2017, pp. 764–773.

[Den+09] Jia Deng et al. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition.* Ieee. 2009, pp. 248–255.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of machine learning research* 12.Jul (2011), pp. 2121–2159.

[Elm90] Jeffrey L Elman. "Finding structure in time". In: *Cognitive science* 14.2 (1990), pp. 179–211.

[ESL14] Joan Bruna Estrach, Arthur Szlam, and Yann LeCun. "Signal recovery from pooling representations". In: *International conference on machine learning.* 2014, pp. 307–315.

[Fuk86] Kunihiko Fukushima. "A neural network model for selective attention in visual pattern recognition". In: *Biological Cybernetics* 55.1 (1986), pp. 5–15.

[Gam17] John Cristian Borges Gamboa. "Deep learning for time-series analysis". In: *arXiv preprint arXiv:1701.01887* (2017).

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* MIT press, 2016.

[GFS07] Alex Graves, Santiago Fernández, and Jürgen Schmidhuber. "Multi-dimensional recurrent neural networks". In: *International conference on artificial neural networks.* Springer. 2007, pp. 549–558.

[GG16] Yarin Gal and Zoubin Ghahramani. “A theoretically grounded application of dropout in recurrent neural networks”. In: *Advances in neural information processing systems.* 2016, pp. 1019–1027.

[GŁ15] Tomasz Górecki and Maciej Łuczak. “Multivariate time series classification with parametric derivative dynamic time warping”. In: *Expert Systems with Applications* 42.5 (2015), pp. 2305–2312.

[GM02] Timothy J Gawne and Julie M Martin. “Responses of primate visual cortical V4 neurons to simultaneously presented stimuli”. In: *Journal of neurophysiology* 88.3 (2002), pp. 1128–1135.

[Gra12] Alex Graves. “Sequence transduction with recurrent neural networks”. In: *arXiv preprint arXiv:1211.3711* (2012).

[Gra13] Alex Graves. “Generating sequences with recurrent neural networks”. In: *arXiv preprint arXiv:1308.0850* (2013).

[GS05] Alex Graves and Jürgen Schmidhuber. “Framewise phoneme classification with bidirectional LSTM and other neural network architectures”. In: *Neural networks* 18.5-6 (2005), pp. 602–610.

[GS09] Alex Graves and Jürgen Schmidhuber. “Offline handwriting recognition with multidimensional recurrent neural networks”. In: *Advances in neural information processing systems.* 2009, pp. 545–552.

[GSC99] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. “Learning to forget: Continual prediction with LSTM”. In: (1999).

[GSS02] Felix A Gers, Nicol N Schraudolph, and Jürgen Schmidhuber. “Learning precise timing with LSTM recurrent networks”. In: *Journal of machine learning research* 3.Aug (2002), pp. 115–143.

[Gul+14] Caglar Gulcehre et al. “Learned-norm pooling for deep feedforward and recurrent neural networks”. In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases.* Springer. 2014, pp. 530–546.

[He+15a] Kaiming He et al. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE international conference on computer vision.* 2015, pp. 1026–1034.

[He+15b] Kaiming He et al. “Spatial pyramid pooling in deep convolutional networks for visual recognition”. In: *IEEE transactions on pattern analysis and machine intelligence* 37.9 (2015), pp. 1904–1916.

[He+16] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016, pp. 770–778.

[Hin+12] Geoffrey Hinton et al. “COURSERA: Neural Networks for Machine Learning”. In: *Lecture 9c: Using noise as a regularizer* (2012).

[Hir96] Hideo Hirose. “Maximum likelihood estimation in the 3-parameter Weibull distribution. A look through the generalized extreme-value distribution”. In: *IEEE Transactions on Dielectrics and Electrical Insulation* 3.1 (1996), pp. 43–55.

[How+17] Andrew G Howard et al. “Mobilenets: Efficient convolutional neural networks for mobile vision applications”. In: *arXiv preprint arXiv:1704.04861* (2017).

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference, and prediction.* Vol. Second Edition. Springer Science & Business Media, 2009.

[HW62] David H Hubel and Torsten N Wiesel. “Receptive fields, binocular interaction and functional architecture in the cat’s visual cortex”. In: *The Journal of physiology* 160.1 (1962), p. 106.

[IS15] Sergey Ioffe and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *arXiv preprint arXiv:1502.03167* (2015).

[Jap+00] Nathalie Japkowicz et al. “Learning from imbalanced data sets: a comparison of various strategies”. In: *AAAI workshop on learning from imbalanced data sets*. Vol. 68. Menlo Park, CA. 2000, pp. 10–15.

[JHD12] Yangqing Jia, Chang Huang, and Trevor Darrell. “Beyond spatial pyramids: Receptive field learning for pooled image features”. In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. 2012, pp. 3370–3377.

[Jor90] Michael I Jordan. “Attractor dynamics and parallelism in a connectionist sequential machine”. In: *Artificial neural networks: concept learning*. 1990, pp. 112–127.

[JZS15] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. “An empirical exploration of recurrent network architectures”. In: *International conference on machine learning*. 2015, pp. 2342–2350.

[KB14] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[Kla+17] Günter Klambauer et al. “Self-normalizing neural networks”. In: *Advances in neural information processing systems*. 2017, pp. 971–980.

[Kob19a] Takumi Kobayashi. “Gaussian-Based Pooling for Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. 2019, pp. 11216–11226.

[Kob19b] Takumi Kobayashi. “Global feature guided local pooling”. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2019, pp. 3365–3374.

[Kra16] Bartosz Krawczyk. “Learning from imbalanced data: open challenges and future directions”. In: *Progress in Artificial Intelligence* 5.4 (2016), pp. 221–232.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[Lam+04] Ilan Lampl et al. “Intracellular measurements of spatial integration and the MAX operation in complex cells of the cat primary visual cortex”. In: *Journal of neurophysiology* 92.5 (2004), pp. 2704–2713.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), p. 436.

[LCY13] Min Lin, Qiang Chen, and Shuicheng Yan. “Network in network”. In: *arXiv preprint arXiv:1312.4400* (2013).

[LeC+90] Yann LeCun, Bernhard E Boser, et al. “Handwritten digit recognition with a back-propagation network”. In: *Advances in neural information processing systems.* 1990, pp. 396–404.

[LeC+98] Yann LeCun, Léon Bottou, et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[LGT16] Chen-Yu Lee, Patrick W Gallagher, and Zhuowen Tu. “Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree”. In: *Artificial intelligence and statistics.* 2016, pp. 464–472.

[LRR19] Francisco Louzada, Pedro L Ramos, and Eduardo Ramos. “A note on bias of closed-form estimators for the gamma distribution derived from likelihood equations”. In: *The American Statistician* 73.2 (2019), pp. 195–199.

[LS50] EL Lehmann and H Scheffé. “Completeness, Similar Regions, and Unbiased Estimation. I”. In: vol. 10. 4. JSTOR 25048038. 1950, 305fffdfffdfffd–340. DOI: 10.1007/978-1-4614-1412-4_23.

[LS55] EL Lehmann and H Scheffé. “Completeness, Similar Regions, and Unbiased Estimation. II”. In: vol. 15. 3. JSTOR 25048243. 1955, pp. 219–236. DOI: 10.1007/978-1-4614-1412-4_24.

[LWH90] Kevin J Lang, Alex H Waibel, and Geoffrey E Hinton. “A time-delay neural network architecture for isolated word recognition”. In: *Neural networks* 3.1 (1990), pp. 23–43.

[Ma+19] Chih-Yao Ma et al. “TS-LSTM and temporal-inception: Exploiting spatiotemporal dynamics for activity recognition”. In: *Signal Processing: Image Communication* 71 (2019), pp. 76–87.

[Mal89] Stephane G Mallat. “A theory for multiresolution signal decomposition: the wavelet representation”. In: *IEEE transactions on pattern analysis and machine intelligence* 11.7 (1989), pp. 674–693.

[Man15] James Manyika. *The Internet of Things: Mapping the value beyond the hype.* McKinsey Global Institute, 2015.

[MB05] Tom McReynolds and David Blythe. *Advanced graphics programming using OpenGL.* Elsevier, 2005.

[MHN13] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. “Rectifier nonlinearities improve neural network acoustic models”. In: *Proc. icml.* Vol. 30. 1. 2013, p. 3.

[Nag+11] Jawad Nagi et al. “Max-pooling convolutional neural networks for vision-based hand gesture recognition”. In: *2011 IEEE International Conference on Signal and Image Processing Applications (ICSIPA).* IEEE. 2011, pp. 342–347.

[NH10] Vinod Nair and Geoffrey E Hinton. “Rectified linear units improve restricted boltzmann machines”. In: *ICML.* 2010.

[Ola15] Christopher Olah. “Understanding lstm networks”. In: (2015).

[Ong17] Thuy Ong. *Facebook’s translations are now powered completely by AI.* Aug. 2017. URL: `https://www.theverge.com/2017/8/4/16093872/facebook-ai-translations-artificial-intelligence`.

[OV10] Carlotta Orsenigo and Carlo Vercellis. “Combining discrete SVM and fixed cardinality warping distances for multivariate time series classification”. In: *Pattern Recognition* 43.11 (2010), pp. 3787–3794.

[Pea01] Karl Pearson. “LIII. On lines and planes of closest fit to systems of points in space”. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2.11 (1901), pp. 559–572.

[Pet+96] Thomas Petsche et al. “A neural network autoassociator for induction motor failure prediction”. In: *Advances in neural information processing systems.* 1996, pp. 924–930.

[PKT83] J Anthony Parker, Robert V Kenyon, and Donald E Troxel. “Comparison of interpolating methods for image resampling”. In: *IEEE Transactions on medical imaging* 2.1 (1983), pp. 31–39.

[Ran+07] Marc’Aurelio Ranzato, Christopher Poultney, et al. “Efficient learning of sparse representations with an energy-based model”. In: *Advances in neural information processing systems.* 2007, pp. 1137–1144.

[Ran+18] Chitta Ranjan et al. “Dataset: rare event classification in multivariate time series”. In: *arXiv preprint arXiv:1809.10717* (2018).

[Ran20] Chitta Ranjan. “Theory of Pooling”. In: *PrePrint, ResearchGate* (Nov. 2020). DOI: `10.13140/RG.2.2.23408.07688`. URL: `https://doi.org/10.13140/RG.2.2.23408.07688`.

[RBC08] Marc’Aurelio Ranzato, Y-Lan Boureau, and Yann L Cun. “Sparse feature learning for deep belief networks”. In: *Advances in neural information processing systems.* 2008, pp. 1185–1192.

[RF87] AJ Robinson and Frank Fallside. *The utility driven dynamic error propagation network.* University of Cambridge Department of Engineering Cambridge, 1987.

[RHW85] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation.* Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *nature* 323.6088 (1986), pp. 533–536.

[Rif+11a] Salah Rifai, Grégoire Mesnil, et al. “Higher order contractive auto-encoder”. In: *Joint European conference on machine learning and knowledge discovery in databases.* Springer. 2011, pp. 645–660.

[Rif+11b] Salah Rifai, Pascal Vincent, et al. “Contractive auto-encoders: Explicit invariance during feature extraction”. In: *Icml*. 2011.

[RP98] Maximilian Riesenhuber and Tomaso Poggio. “Just one view: Invariances in inferotemporal cell tuning”. In: *Advances in neural information processing systems*. 1998, pp. 215–221.

[RP99] Maximilian Riesenhuber and Tomaso Poggio. “Hierarchical models of object recognition in cortex”. In: *Nature neuroscience* 2.11 (1999), pp. 1019–1025.

[Sae+18] Faraz Saeedan et al. “Detail-preserving pooling in deep networks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 9108–9116.

[Sch12] Mark J Schervish. *Theory of statistics*. Springer Science & Business Media, 2012.

[Shi+16a] Wenzhe Shi, Jose Caballero, Ferenc Huszár, et al. “Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 1874–1883.

[Shi+16b] Wenzhe Shi, Jose Caballero, Lucas Theis, et al. “Is the deconvolution layer the same as a convolutional layer?” In: *arXiv preprint arXiv:1609.07009* (2016).

[Smi16] Chris Smith. *iOS 10: Siri now works in third-party apps, comes with extra AI features*. June 2016. URL: `https://bgr.com/2016/06/13/ios-10-siri-third-party-apps/`.

[Smi85] Richard L Smith. “Maximum likelihood estimation in a class of nonregular cases”. In: *Biometrika* 72.1 (1985), pp. 67–90.

[SP10] Thomas Serre and Tomaso Poggio. “A neuromorphic approach to computer vision”. In: *Communications of the ACM* 53.10 (2010), pp. 54–61.

[SP97] Mike Schuster and Kuldip K Paliwal. “Bidirectional recurrent neural networks”. In: *IEEE transactions on Signal Processing* 45.11 (1997), pp. 2673–2681.

[Spr+15] Jost Tobias Springenberg et al. “Striving for simplicity: The all convolutional net”. In: *ICLR* (2015).

[Sri+14] Nitish Srivastava et al. “Dropout: a simple way to prevent neural networks from overfitting”. In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. “Sequence to sequence learning with neural networks”. In: *Advances in neural information processing systems*. 2014, pp. 3104–3112.

[SWK09] Yanmin Sun, Andrew KC Wong, and Mohamed S Kamel. “Classification of imbalanced data: A review”. In: *International Journal of Pattern Recognition and Artificial Intelligence* 23.04 (2009), pp. 687–719.

[SWP05] Thomas Serre, Lior Wolf, and Tomaso Poggio. “Object recognition with features inspired by visual cortex”. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05)*. Vol. 2. Ieee. 2005, pp. 994–1000.

[SZ15] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *ICLR* (2015).

[Sze+15] Christian Szegedy et al. “Going deeper with convolutions”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[Ten18] TensorFlow. *Standardizing on Keras: Guidance on High-level APIs in TensorFlow 2.0*. Dec. 2018. URL: `https://medium.com/tensorflow/standardizing-on-keras-guidance-on-high-level-apis-in-tensorflow-2-0-bad2b04c819a`.

[Ten19] TensorFlow. *What’s coming in TensorFlow 2.0*. Jan. 2019. URL: `https://medium.com/tensorflow/whats-coming-in-tensorflow-2-0-d3663832e9b8`.

[TM98] Carlo Tomasi and Roberto Manduchi. “Bilateral filtering for gray and color images”. In: *Sixth international conference on computer vision (IEEE Cat. No. 98CH36271)*. IEEE. 1998, pp. 839–846.

[Vas+17] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[Vin+08] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, et al. “Extracting and composing robust features with denoising autoencoders”. In: *Proceedings of the 25th international conference on Machine learning*. 2008, pp. 1096–1103.

[Vin+10] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, et al. “Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion.” In: *Journal of machine learning research* 11.12 (2010).

[VKE19] Aaron Voelker, Ivana Kajić, and Chris Eliasmith. “Legendre Memory Units: Continuous-Time Representation in Recurrent Neural Networks”. In: *Advances in Neural Information Processing Systems*. 2019, pp. 15544–15553.

[Vog16] Werner Vogels. *Bringing the Magic of Amazon AI and Alexa to Apps on AWS*. Nov. 2016.

[Web+16] Nicolas Weber et al. “Rapid, detail-preserving image downscaling”. In: *ACM Transactions on Graphics (TOG)* 35.6 (2016), pp. 1–6.

[WH18] Yuxin Wu and Kaiming He. “Group normalization”. In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 3–19.

[WL18] Travis Williams and Robert Li. “Wavelet pooling for convolutional neural networks”. In: *International Conference on Learning Representations*. 2018.

[Wu+16] Yonghui Wu et al. “Google’s neural machine translation system: Bridging the gap between human and machine translation”. In: *arXiv preprint arXiv:1609.08144* (2016).

[WZ95] Ronald J Williams and David Zipser. “Gradient-based learning algorithms for recurrent”. In: *Backpropagation: Theory, architectures, and applications* 433 (1995).

[Xie+17] Saining Xie et al. “Aggregated residual transformations for deep neural networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2017, pp. 1492–1500.

[Yan+09] Jianchao Yang et al. “Linear spatial pyramid matching using sparse coding for image classification”. In: *2009 IEEE Conference on computer vision and pattern recognition.* IEEE. 2009, pp. 1794–1801.

[YC17] Zhi-Sheng Ye and Nan Chen. “Closed-form estimators for the gamma distribution derived from likelihood equations”. In: *The American Statistician* 71.2 (2017), pp. 177–181.

[YK16] Fisher Yu and Vladlen Koltun. “Multi-scale context aggregation by dilated convolutions”. In: *ICLR* (2016).

[Yu+14] Dingjun Yu et al. “Mixed pooling for convolutional neural networks”. In: *International conference on rough sets and knowledge technology.* Springer. 2014, pp. 364–375.

[ZF13] Matthew D Zeiler and Rob Fergus. “Stochastic pooling for regularization of deep convolutional neural networks”. In: *ICLR* (2013).

[ZS18] Zhilu Zhang and Mert Sabuncu. “Generalized cross entropy loss for training deep neural networks with noisy labels”. In: *Advances in neural information processing systems.* 2018, pp. 8778–8788.

Appendix A

Importance of Nonlinear Activation

It is mentioned a few times in this book that a nonlinear activation is essential for the nonlinearity of a deep learning model. Specifically, this is emphasized in § 4.7 in Chapter 4. In this appendix, it is shown that a linear activation makes a multi-layer network a simple linear regression model.

In the following, Equation A.1 applies the activation successively from input to the output in an illustrative two-layer network in Chapter 4. It shows that if the activation is linear, i.e., $g(x) = x$, then any multi-layer network becomes equivalent to a linear model.

$$\begin{aligned}\hat{y} &= g(\mathbf{w}^T\mathbf{z}^{(2)}) && \text{(A.1a)}\\ &= g(\mathbf{w}^T g(W^{(2)T} g(W^{(1)T}\mathbf{x}))) && \text{(A.1b)}\\ &= \mathbf{w}^T W^{(2)T} W^{(1)T}\mathbf{x}, \text{if activation } g \text{ is linear.} && \text{(A.1c)}\\ &= \tilde{W}^T\mathbf{x} && \text{(A.1d)}\end{aligned}$$

where, $\tilde{W}^T = \mathbf{w}^T W^{(2)T} W^{(1)T}$.

Appendix B

Curve Shifting

Curve shifting is used to learn relationships between the variables at a certain time with an event at a different time. The event can be either from the past or the future.

For a rare event prediction, where the objective is to predict an event in advance, the event is shifted back in time. This approach is similar to developing a model to predict a transition state which ultimately leads to an event.

In Listing B.1, a user-defined function (UDF) for curve-shifting a binary response data is shown. In the UDF, an input argument `shift_by` corresponds to the time units we want to shift `y`. `shift_by` can be a positive or negative integer.

Listing B.1. Curve Shifting.

```
import numpy as np

def sign(x):
    return (1, -1)[x < 0]


def curve_shift(df, shift_by):
    '''
    This function will shift the binary labels in a
        dataframe.
    The curve shift will be with respect to the 1s.
    For example, if shift is -2, the following
        process
    will happen: if row n is labeled as 1, then
    - Make row (n+shift_by):(n+shift_by-1) = 1.
    - Remove row n.
    i.e. the labels will be shifted up to 2 rows up.

    Inputs:
    df       A pandas dataframe with a binary
        labeled column.
             This labeled column should be named as
                 'y'.
    shift_by An integer denoting the number of rows
        to shift.

    Output
    df       A dataframe with the binary labels
        shifted by shift.
    '''

    vector = df['y'].copy()
    for s in range(abs(shift_by)):
        tmp = vector.shift(sign(shift_by))
        tmp = tmp.fillna(0)
        vector += tmp
    labelcol = 'y'
    # Add vector to the df
    df.insert(loc=0, column=labelcol + 'tmp', value=
        vector)
    # Remove the rows with labelcol == 1.
    df = df.drop(df[df[labelcol] == 1].index)
    # Drop labelcol and rename the tmp col as
        labelcol
    df = df.drop(labelcol, axis=1)
    df = df.rename(columns={labelcol + 'tmp':
        labelcol})
    # Make the labelcol binary
    df.loc[df[labelcol] > 0, labelcol] = 1

    return df
```

`curve_shift` assumes the response is binary with `(0, 1)` labels, and for any row `t` where `y==1` it,

1. makes the `y` for rows `(t+shift_by):(t+shift_by-1)` equal to 1. Mathematically, this is $y_{(t-k):t} \leftarrow 1$, if $y_t = 1$ and k is the `shift_by`. And,

2. remove row `t`.

Step 1 shifts the curve. Step 2 removes the row when the event (sheet-break) occurred. As also mentioned in § 2.1.2, we are not interested in teaching the model to predict an event when it has already occurred.

The effect of the curve shifting is shown using Listing B.2.

Listing B.2. Testing Curve Shift.

```
import pandas as pd
import numpy as np

'''Download data here:
https://docs.google.com/forms/d/e/1
    FAIpQLSdyUk3lfDl7I5KYK_pw285LCApc-
    _RcoC0Tf9cnDnZ_TWzPAw/viewform
'''
df = pd.read_csv("data/processminer-sheet-break-rare
    -event-dataset.csv")
df.head(n=5)  # visualize the data.

# Hot encoding
hotencoding1 = pd.get_dummies(df['Grade&Bwt'])
hotencoding1 = hotencoding1.add_prefix('grade_')
hotencoding2 = pd.get_dummies(df['EventPress'])
hotencoding2 = hotencoding2.add_prefix('eventpress_'
    )

df = df.drop(['Grade&Bwt', 'EventPress'], axis=1)

df = pd.concat([df, hotencoding1, hotencoding2],
    axis=1)

df = df.rename(columns={'SheetBreak': 'y'})  #
    Rename response column name for ease of
    understanding
```

```
'''
Shift the data by 2 units, equal to 4 minutes.

Test: Testing whether the shift happened correctly.
'''
print('Before shifting')  # Positive labeled rows
    before shifting.
one_indexes = df.index[df['y'] == 1]
display(df.iloc[(one_indexes[0]-3):(one_indexes
    [0]+2), 0:5].head(n=5))

# Shift the response column y by 2 rows to do a 4-
    min ahead prediction.
df = curve_shift(df, shift_by = -2)

print('After shifting')  # Validating if the shift
    happened correctly.
display(df.iloc[(one_indexes[0]-4):(one_indexes
    [0]+1), 0:5].head(n=5))
```

The outputs of the listing are visualized in Figure 4.3 in Chapter 4.

Appendix C

Simple Plots

The result plots in every chapter are made using the definitions in Listing C.1.

Listing C.1. Simple plot definitions.

```python
##############################
##### Plotting functions #####
##############################

import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

def plot_metric(model_history, metric,
    ylim=None, grid=False):
    sns.set()

    if grid is False:
        sns.set_style("white")
        sns.set_style("ticks")

    train_values = [
        value for key, value in model_history.items
            ()
        if metric in key.lower()
    ][0]
    valid_values = [
        value for key, value in model_history.items
            ()
        if metric in key.lower()
    ][1]

    fig, ax = plt.subplots()

    color = 'tab:blue'
    ax.set_xlabel('Epoch', fontsize=16)
    ax.set_ylabel(metric, color=color, fontsize=16)

    ax.plot(train_values, '--', color=color,
        label='Train ' + metric)
    ax.plot(valid_values, color=color,
        label='Valid ' + metric)
    ax.tick_params(axis='y', labelcolor=color)
    ax.tick_params(axis='both',
        which='major', labelsize=14)

    if ylim is None:
        ylim = [
            min(min(train_values),
                min(valid_values), 0.),
            max(max(train_values),
                max(valid_values))
        ]
    plt.yticks(np.round(np.linspace(ylim[0],
        ylim[1], 6), 1))
    plt.legend(loc='upper left', fontsize=16)

    if grid is False:
        sns.despine(offset=1, trim=True)

    return plt, fig


def plot_model_recall_fpr(model_history, grid=False)
   :
    sns.set()

    if grid is False:
        sns.set_style("white")
        sns.set_style("ticks")

    train_recall = [
        value for key, value in model_history.items
            ()
        if 'recall' in key.lower()
    ][0]
    valid_recall = [
        value for key, value in model_history.items
            ()
        if 'recall' in key.lower()
    ][1]

    train_fpr = [
        value for key, value in model_history.items
            ()
        if 'false_positive_rate' in key.lower()
    ][0]
    valid_fpr = [
        value for key, value in model_history.items
            ()
        if 'false_positive_rate' in key.lower()
    ][1]

    fig, ax = plt.subplots()

    color = 'tab:red'
    ax.set_xlabel('Epoch', fontsize=16)
    ax.set_ylabel('value', fontsize=16)
    ax.plot(train_recall, '--', color=color, label='
        Train Recall')
    ax.plot(valid_recall, color=color, label='Valid
        Recall')
    ax.tick_params(axis='y', labelcolor='black')
    ax.tick_params(axis='both', which='major',
        labelsize=14)
    plt.legend(loc='upper left', fontsize=16)

    color = 'tab:blue'
    ax.plot(train_fpr, '--', color=color, label='
        Train FPR')
```

```
    ax.plot(valid_fpr, color=color, label='Valid FPR
        ')
    plt.yticks(np.round(np.linspace(0., 1., 6), 1))

    fig.tight_layout()
    plt.legend(loc='upper left', fontsize=16)

    if grid is False:
        sns.despine(offset=1, trim=True)

    return plt, fig
```

Appendix D

Backpropagation Gradients

Think of the two-layer neural network shown in Figure 4.2 illustrated in Chapter 4. We used a `binary_crossentropy` loss for this model shown in Equation 4.7. Without any loss of generality (w.l.o.g.), it can be expressed for a single sample as,

$$\mathcal{L}(\theta) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}) \tag{D.1}$$

where, $\hat{y}$ is the prediction for y, i.e. the $\Pr_\theta[y = 1]$ (denoted by p in Eq. 4.7) and θ is the set of all parameters $\{W^{(1)}, W^{(2)}, \mathbf{w}^{(o)}\}$. Here the bias parameters are assumed as 0 w.l.o.g.

The parameter update in an iterative estimation vary for different optimizers such as `adam` and `sgd` in TensorFlow. However, as they are all Gradient Descent based, the update rule generalizes as,

$$\theta \leftarrow \theta - \eta \frac{\partial \mathcal{L}}{\partial \theta} \tag{D.2}$$

where η is a learning parameter.

As seen in the equation, the gradient guides the parameter estimation to reach its optimal value.

The gradient expression for a weight parameter can be derived as,

$$\frac{\partial \mathcal{L}}{\partial W^T} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W^T} \tag{D.3}$$

A derivative of the weight transpose is used for mathematical convenience. Besides, $\frac{\partial \mathcal{L}}{\partial \hat{y}}$ will always be the same and, therefore, can be ignored to express,

$$\frac{\partial \mathcal{L}}{\partial W^T} \propto \frac{\partial \hat{y}}{\partial W^T} \tag{D.4}$$

Additionally, the relationship between the layers' inputs and outputs are,

$$\begin{aligned} \hat{y} &= \sigma(\mathbf{w}^{(o)T}\mathbf{z}^{(2)}) && \text{(D.5a)} \\ \mathbf{z}^{(2)} &= g(W^{(2)T}\mathbf{z}^{(1)}) && \text{(D.5b)} \\ \mathbf{z}^{(1)} &= g(W^{(1)T}\mathbf{x}) && \text{(D.5c)} \end{aligned}$$

where σ is the activation on the output layer and g on the hidden layers. Note that g can be different across layers but shown to be the same here for simplicity.

Using Equation D.4-D.5, we can express the gradients for each weight parameter as,

Output Layer.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}^{(o)T}} \propto \frac{\partial \sigma(\mathbf{w}^{(o)T}\mathbf{z}^{(2)})}{\partial \mathbf{w}^{(o)T}} \tag{D.6}$$

Hidden Layer-2.

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial W^{(2)T}} &\propto \frac{\partial \sigma(\mathbf{w}^{(o)T}\mathbf{z}^{(2)})}{\partial W^{(2)T}} \\ &\propto \frac{\partial}{\partial W^{(2)T}} \sigma(\mathbf{w}^{(o)T} g(W^{(2)T}\mathbf{z}^{(1)})) \\ &\propto \frac{\partial \sigma(\mathbf{w}^{(o)T} g(W^{(2)T}\mathbf{z}^{(1)}))}{\partial g(W^{(2)T}\mathbf{z}^{(1)})} \frac{\partial g(W^{(2)T}\mathbf{z}^{(1)})}{\partial W^{(2)T}}\end{aligned} \tag{D.7}$$

Hidden Layer-1.

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial W^{(1)T}} &\propto \frac{\partial \sigma(\mathbf{w}^{(o)T}\mathbf{z}^{(2)})}{\partial W^{(1)T}} \\ &\propto \frac{\partial}{\partial W^{(1)T}} \sigma(\mathbf{w}^{(o)T} g(W^{(2)T} g(W^{(1)T}\mathbf{x}))) \\ &\propto \frac{\partial \sigma(\mathbf{w}^{(o)T} g(W^{(2)T} g(W^{(1)T}\mathbf{x})))}{\partial g(W^{(2)T} g(W^{(1)T}\mathbf{x}))} \frac{\partial g(W^{(2)T} g(W^{(1)T}\mathbf{x}))}{\partial g(W^{(1)T}\mathbf{x})} \frac{\partial g(W^{(1)T}\mathbf{x})}{\partial W^{(1)T}}\end{aligned} \tag{D.8}$$

Appendix E

Data Temporalization

Temporal models such as LSTM and convolutional networks are a bit more demanding than other models. A significant amount of time and attention goes into preparing the data that fits them.

First, we will create the three-dimensional tensors of shape: (*samples*, *timesteps*, *features*) in Listing E.1. *Samples* mean the number of data points. *Timesteps* is the number of time steps we look back at any time t to make a prediction. This is also referred to as the `lookback` period. The *features* are the number of features the data has, in other words, the number of predictors in multivariate data.

Listing E.1. Data temporalization

```
def temporalize(X, y, lookback):
    '''
    Inputs
    X          A 2D numpy array ordered by time of
       shape: (n_observations x n_features)
    y          A 1D numpy array with indexes aligned
       with X, i.e. y[i] should correspond to X[i].
        Shape: n_observations.
    lookback   The window size to look back in the
       past records. Shape: a scalar.

    Output
    output_X   A 3D numpy array of shape: ((
       n_observations-lookback-1) x lookback x
```

```
        n_features)
    output_y  A 1D array of shape: (n_observations-lookback-1), aligned with X.
    '''
    output_X = []
    output_y = []
    for i in range(len(X) - lookback - 1):
        t = []
        for j in range(1, lookback + 1):
            # Gather the past records upto the lookback period
            t.append(X[[(i + j + 1)], :])
        output_X.append(t)
        output_y.append(y[i + lookback + 1])
    return np.squeeze(np.array(output_X)), np.array(output_y)


def flatten(X):
    '''
    Flatten a 3D array.

    Input
    X             A 3D array for lstm, where the array is sample x timesteps x features.

    Output
    flattened_X  A 2D array, sample x features.
    '''
    flattened_X = np.empty(
        (X.shape[0], X.shape[2]))  # sample x features array.
    for i in range(X.shape[0]):
        flattened_X[i] = X[i, (X.shape[1] - 1), :]
    return flattened_X


def scale(X, scaler):
    '''
    Scale 3D array.

    Inputs
    X             A 3D array for lstm, where the
        array is sample x timesteps x features.
    scaler        A scaler object, e.g., sklearn.
        preprocessing.StandardScaler, sklearn.
        preprocessing.normalize

    Output
    X             Scaled 3D array.
    '''
    for i in range(X.shape[0]):
        X[i, :, :] = scaler.transform(X[i, :, :])

    return X
```

Additional helper functions, `flatten()` and `scale()`, are defined to make it easier to work with the tensors.

Testing

Since temporalization is an error-prone transformation, it is important to test the input tensors as shown in Listing E.2.

Listing E.2. Testing data temporalization.

```
"""### Temporalized data scale testing"""

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# Sort by time and drop the time column.
df['DateTime'] = pd.to_datetime(df.DateTime)
df = df.sort_values(by='DateTime')
df = df.drop(['DateTime'], axis=1)

input_X = df.loc[:, df.columns != 'y'].values  #
    converts df to numpy array
input_y = df['y'].values

n_features = input_X.shape[1]  # number of features

# Temporalize the data
lookback = 5
```

```
X, y = temporalize(X=input_X,
                   y=input_y,
                   lookback=lookback)

X_train, X_test, y_train, y_test = train_test_split(
    np.array(X),
    np.array(y),
    test_size=0.2,
    random_state=123)
X_train, X_valid, y_train, y_valid =
   train_test_split(
    X_train,
    y_train,
    test_size=0.2,
    random_state=123)

# Initialize a scaler using the training data.
scaler = StandardScaler().fit(flatten(X_train))

X_train_scaled = scale(X_train, scaler)

'''
Test: Check if the scaling is correct.

The test succeeds if all the column means
and variances are 0 and 1, respectively, after
flattening.
'''
print('==== Column-wise mean ====\n', np.mean(
   flatten(X_train_scaled), axis=0).round(6))
print('==== Column-wise variance ====\n', np.var(
   flatten(X_train_scaled), axis=0))

# ==== Column-wise mean ====
#  [-0.  0.  0. -0. -0. -0. -0.  0. -0. -0.  0. -0.
    -0.  0.  0.  0.  0.  0.
#  -0. -0. -0. -0.  0.  0. -0. -0.  0.  0. -0.  0.
   0.  0.  0.  0. -0.  0.
#   0.  0. -0.  0.  0. -0. -0.  0. -0.  0.  0.  0.
   0. -0. -0. -0.  0.  0.
#   0.  0.  0. -0. -0.  0. -0. -0. -0. -0.  0.  0.
    -0.  0.  0.]
```

```
# ==== Column-wise variance ====
#  [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
    1.  1.  1.  1.  1.  1.  1.  1.
#   1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
    1.  1.  1.  1.  1.  1.  1.  1.
#   1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
    1.  1.  1.  0.  0.]
```

The temporalization output is shown in Figure 5.8 in Chapter 5.

Besides, the `scale()` function is tested and the outputs are shown in the listing. As expected, the mean and variances become 0 and 1 after using a `StandardScaler()` on the temporalized data.

Appendix F

Stateful LSTM

A typical *stateless* LSTM cell illustrated in Chapter 5 processes only as much "past" in the data as defined by the timesteps. This could be restrictive because:

> "Ideally, we want to expose the network to the entire sequence and let it learn the inter-dependencies, rather than we define those dependencies explicitly in the framing of the problem.
>
> ...
>
> This is truly how the LSTM networks are intended to be used."
>
> – Jason Brownlee

It is, in fact, learned in temporal modeling with LSTMs and convolutional neural networks in Chapter 5 and 6 that a larger lookback (timesteps) typically improves a model's accuracy. This is logical because there are lagged dependencies. And, a larger timestep allows the model to look farther in the past to have more context for prediction.

If going farther back in time improves accuracy then can we go all the way in the past? The answer is yes. This can be done with *stateful* LSTMs.

Using a stateful LSTM is a simple approach. This approach requires the input samples to be ordered by time. Also, unlike typical model fitting that resets the model every training iteration, with the stateful LSTM it is reset every epoch.

The implementation in this appendix shows that a stateful LSTM cell processes the entire input training data sequentially and learns the dependencies from anywhere in the past.

However, as lucrative as the stateful LSTM appears, it does not always work. This approach tends to work better if the data is stationary. For example, text documents. The writing pattern does not change significantly and therefore, the process is stationary.

But most time series processes are non-stationary. The dependencies in them are confounded due to the non-stationarity. Therefore, a window of timesteps in which the process is assumed to be stationary tends to work better. For the same reason, a large time step should be carefully chosen in non-stationary processes.

Stateful LSTM is suitable if the data is stationary, i.e., the patterns do not change with time.

Implementing a stateful LSTM is different from traditional models. In the following, the implementation steps are given.

Data Preparation

In a stateful LSTM network, it is necessary to have the size of the input data as a multiple of the batch size. The data preparation is thus slightly different. In Listing F.1 the number of samples for train, valid, and test is taken as a multiple of the batch size which is closest to their original size.

Listing F.1. Stateful LSTM model data preparation.

```
# Time ordered original data.
lookback_stateful = 1
# Temporalize the data
X, y = temporalize(X=input_X,
                   y=input_y,
                   lookback=lookback_stateful)

batch_size = 128

# Train, valid and test size set
# to match the previous models.
train_size = 13002
valid_size = 3251
test_size = 3251

X_train_stateful, y_train_stateful =
    np.array(
        X[0:int(train_size / batch_size) *
            batch_size]),
            np.array(
        y[0:int(train_size / batch_size) *
            batch_size])
X_valid_stateful, y_valid_stateful = np.array(
    X[int(train_size / batch_size) *
      batch_size:int((train_size + valid_size) /
          batch_size) *
      batch_size]), np.array(
          y[int(train_size / batch_size) *
            batch_size:int((train_size +
                valid_size) / batch_size) *
            batch_size])
X_test_stateful, y_test_stateful = np.array(
    X[int((train_size + test_size) / batch_size) *
        batch_size:]), np.array(
        y[int((train_size + test_size) /
            batch_size) * batch_size:])

X_train_stateful =
    X_train_stateful.reshape(
        X_train_stateful.shape[0],
        lookback_stateful,
        n_features)
X_valid_stateful =
    X_valid_stateful.reshape(
        X_valid_stateful.shape[0],
        lookback_stateful,
        n_features)
X_test_stateful =
    X_test_stateful.reshape(
        X_test_stateful.shape[0],
        lookback_stateful,
        n_features)

scaler_stateful =
    StandardScaler().fit(flatten(
        X_train_stateful))

X_train_stateful_scaled =
    scale(X_train_stateful,
          scaler_stateful)

X_valid_stateful_scaled =
    scale(X_valid_stateful,
          scaler_stateful)
X_test_stateful_scaled =
    scale(X_test_stateful,
          scaler_stateful)
```

The question is, why the batch size is required in a stateful model?

It is because when the model is stateless, TensorFlow allocates a tensor for the states of size `output_dim` based on the number of LSTM cells. At each sequence processing, this state tensor is reset.

On the other hand, TensorFlow propagates the previous states for each sample across the batches in a stateful model. In this case, the structure to store the states is of shape `(batch_size, output_dim)`. Due to this, it is necessary to provide the batch size while constructing the network.

Stateful Model

A stateful LSTM model is designed to traverse the entire past in the data for the model to self-learn the distant inter-dependencies instead of limiting it in a lookback window. This is achieved with a specific training procedure shown in Listing F.2.

The LSTM layer is made stateful by setting its argument `stateful=True`.

Listing F.2. Stateful LSTM model.

```python
# Stateful model.

timesteps_stateful =
    X_train_stateful_scaled.shape[1]
n_features_stateful =
    X_train_stateful_scaled.shape[2]

model = Sequential()
model.add(
    Input(shape=(timesteps_stateful,
                 n_features_stateful),
          batch_size=batch_size,
          name='input'))
model.add(
    LSTM(8,
         activation='relu',
         return_sequences=True,
         stateful=True,
         name='lstm_layer_1'))
model.add(Flatten())
model.add(Dense(units=1,
                activation='sigmoid',
                name='output'))

model.summary()

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=[
                  'accuracy',
                  tf.keras.metrics.Recall(),
                  performancemetrics.F1Score(),
                  performancemetrics.
                      FalsePositiveRate()
              ])
```

Unlike stateless LSTM, the cell states are preserved at every training iteration in a stateful LSTM. This allows it to learn the dependencies between the batches and, therefore, long-term patterns in significantly long sequences. However, we do not want the state to be transferred from one epoch to the next. To avoid this, we have to manually reset the state after each epoch.

A custom operation during training iterations can be performed by overriding the definitions in `tf.keras.callbacks.Callback`[1]. The `Callback()` class has definitions to perform operations at the beginning and/or end of a `batch` or `epoch` for both `test` and `train`. Since we require to reset the model states at the end of every epoch, we override the `on_epoch_end()` in Listing F.3 with `model.reset_states()`.

Listing F.3. Custom Callback() for Stateful LSTM model.

```
class ResetStatesCallback(
    tf.keras.callbacks.Callback):

    def on_epoch_end(self, epoch, logs={}):
        self.model.reset_states()
```

We now train the model in Listing F.4. In the `model.fit()`, we set the argument `callbacks` equal to our custom defined `ResetStatesCallback()`. Also, we set `shuffle=False` to maintain the time ordering of the samples during the training.

Listing F.4. Stateful LSTM model fitting.

```
history = model.fit(
    x=X_train_stateful_scaled,
    y=y_train_stateful,
    callbacks=[ResetStatesCallback()],
    batch_size=batch_size,
    epochs=100,
    shuffle=False,
    validation_data=(X_valid_stateful_scaled,
                     y_valid_stateful),
    verbose=0).history
```

The results from stateful LSTM on the sheet-break time series is poorer than the other LSTM models in Chapter 5. As alluded to earlier, a potential reason is that the process is non-stationary. Due to this, the dependencies change over time and are difficult to learn.

[1] /api_docs/python/tf/keras/callbacks/Callback

Appendix G

Null-Rectified Linear Unit

Rectified Linear Units (`relu`) is one of the most common activation functions. It is expressed as,

$$g(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise.} \end{cases} \tag{G.1}$$

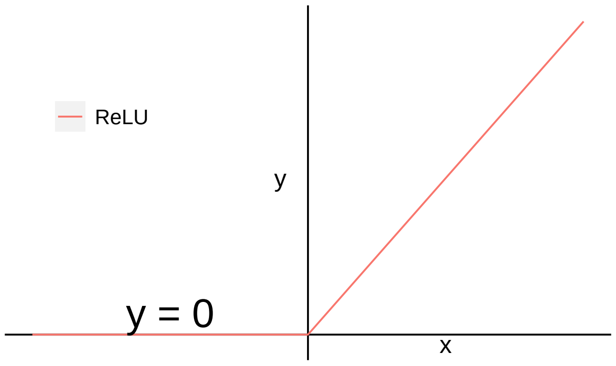
As shown in the equation, `relu` makes any non-positive x as 0. This brings nonlinearity to the model but at the cost of feature manipulation.

Such manipulation affects the pooling operation in convolutional networks (see § 6.12.2 in Chapter 6). The manipulation distorts the original distribution of the features which makes some pooling statistics, e.g., average, inefficient.

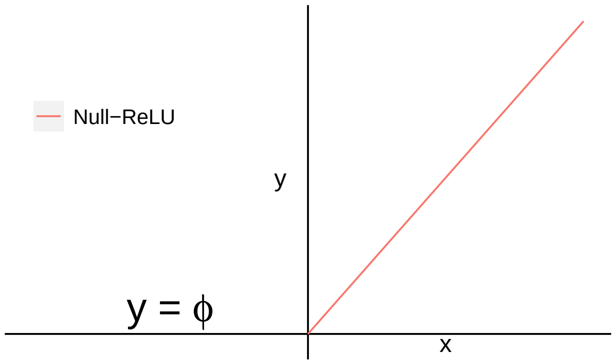
A resolution mentioned in § 6.13.3 is replacing `relu` activation with null-`relu`, which is defined as,

$$g(x) = \begin{cases} x, & \text{if } x > 0 \\ \phi, & \text{otherwise} \end{cases} \tag{G.2}$$

where ϕ denotes *null*. Unlike Equation G.1, null-`relu` acts as dropping the non-positive x's instead of treating them as 0. Both are visualized in Figure G.1a and G.1b, respectively.



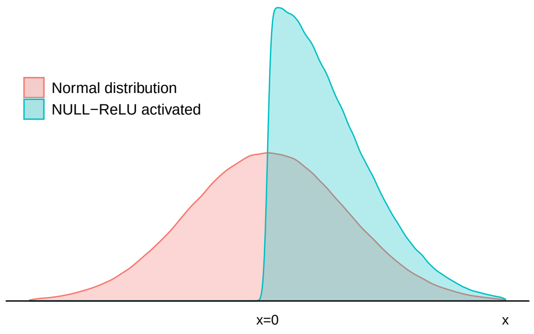
(a) *Traditional* ***relu*** *activation.*



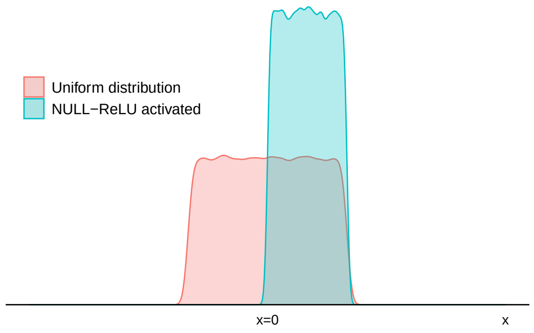
(b) *A variant null-****relu*** *activation.*

Figure G.1. *The traditional Rectified Linear Unit (****relu****) activation (top) transforms any non-positive x to zero. This is a nonlinear operation essential in most deep learning layers. However, in between a convolutional and pooling operation, the* ***relu*** *transformation can have an unwanted effect. The pooling attempts to draw a summary statistic from convolution output. But* ***relu*** *brings artificial* 0*'s that subvert summary information. A Null-****relu*** *activation (bottom) mitigates this by replacing the non-positive x's with ϕ (null). Unlike* 0*s in* ***relu****, nulls do not add any artificial information; they only mask the non-positive x's.*

Null-`relu`'s impact on the activated features for normal and uniform distributions are shown in Figure G.2a and G.2b, respectively. As opposed to the `relu` effect shown in Figure 6.28a and 6.28b, the activated feature distributions are still known—half-gaussian and uniform. Therefore, efficient pooling statistics such as in Kobayashi 2019a can be used.



(a) *Normal Distribution after null-`relu` activation.*



(b) *Uniform Distribution after null-`relu` activation.*

Figure G.2. *The distribution of the feature map is distorted by a `relu` activation. While any nonlinear activation distorts the original distribution, `relu`'s is severe because it artificially adds zeros for every non-positive x. Due to this, the activated x's distribution becomes extremely heavy at zero. Such distributions do not belong to known or well-defined distribution families. A variant of `relu` called null-`relu` transforms the non-positive x to null (ϕ). This is equivalent to throwing the non-positive x's instead of assuming them to be zero. If the original distribution is normal (top) or uniform (bottom), the null-`relu` activated are half-Gaussian and Uniform, respectively. Therefore, it has a less severe effect on distribution.*

Appendix H

1×1 Convolutional Network

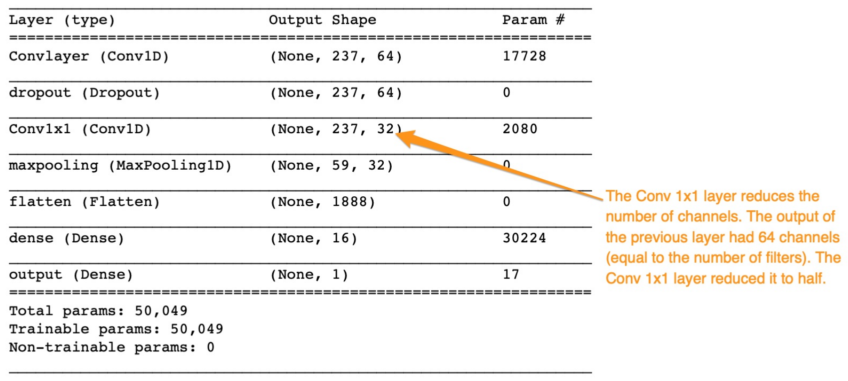
§ 6.7.4 explains 1×1 convolution layers and their purpose. An illustrative convolutional network with 1×1 convolutional layer is shown in Listing H.

A regular convolution layer has a kernel size larger than 1 to learn spatial features. Differently, a 1×1 convolution layer has a kernel size equal to 1. Its primary purpose is to reduce the channels for network dimension reduction. Sometimes they are also used to learn features from the information in the channels. For this, multiple 1×1 layers are stacked in parallel.

The illustrative example in Listing H has a single 1×1 convolutional layer. As shown in Figure H.1, the layer resulted in a reduction of the channels from 64 to 32. The network is built and trained on temporalized data with `lookback=240`. The results are shown in Figure H.2a-H.2c.

```
## 1x1 convolutional network

model = Sequential()
model.add(Input(shape=(TIMESTEPS,
                       N_FEATURES),
                name='input'))
model.add(Conv1D(filters=64,
                 kernel_size=4,
                 activation='relu',
                 name='Convlayer'))
```

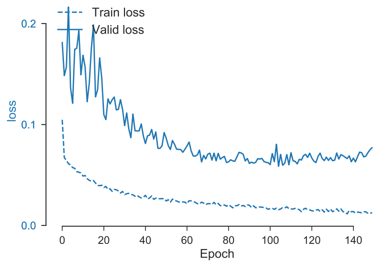
```
model.add(Dropout(rate=0.5,
                  name='dropout'))
model.add(Conv1D(filters=32,
                 kernel_size=1,
                 activation='relu',
                 name='Conv1x1'))
model.add(MaxPool1D(pool_size=4,
                    name='maxpooling'))
model.add(Flatten(name='flatten'))
model.add(Dense(units=16,
                activation='relu',
                name='dense'))
model.add(Dense(units=1,
                activation='sigmoid',
                name='output'))
model.summary()

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=[
                  'accuracy',
                  tf.keras.metrics.Recall(),
                  pm.F1Score(),
                  pm.FalsePositiveRate()
              ])
history = model.fit(x=X_train_scaled,
                    y=y_train,
                    batch_size=128,
                    epochs=150,
                    validation_data=(X_valid_scaled,
                                     y_valid),
                    verbose=0).history
```



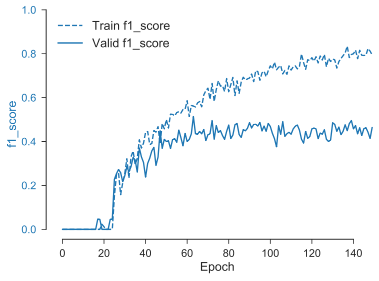
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Convlayer (Conv1D)           (None, 237, 64)           17728
_________________________________________________________________
dropout (Dropout)            (None, 237, 64)           0
_________________________________________________________________
Conv1x1 (Conv1D)             (None, 237, 32)           2080
_________________________________________________________________
maxpooling (MaxPooling1D)    (None, 59, 32)            0
_________________________________________________________________
flatten (Flatten)            (None, 1888)              0
_________________________________________________________________
dense (Dense)                (None, 16)                30224
_________________________________________________________________
output (Dense)               (None, 1)                 17
=================================================================
Total params: 50,049
Trainable params: 50,049
Non-trainable params: 0
_________________________________________________________________
```

The Conv 1x1 layer reduces the number of channels. The output of the previous layer had 64 channels (equal to the number of filters). The Conv 1x1 layer reduced it to half.

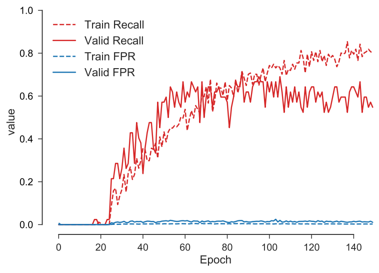
Figure H.1. *Summary of a 1 × 1 convolutional network.*



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure H.2. *1 × 1 convolutional network results.*

Appendix I

CNN: Visualization for Interpretation

A benefit of convolutional networks is that they have filters and feature representations which can be visualized to understand the spatio-temporal patterns and dependencies. These visuals help diagnose a model.

In this appendix, illustrations for filters and feature representation visualization are presented. The visuals are for the network constructed in Appendix H.

Filter Visualization

We can look at the layers in a `model` by printing `model.layers` as shown in Listing I.1. Every convolutional layer makes a set of filters. A layer weights (the filters) can be taken using the layer index or the name.

For example, the convolutional layer weights in the model in Listing H can be fetched as `model.layers[0].get_weights()` because it is the first layer which has index 0, or using a named call as `model.get_layer('Convlayer').get_weights()` where `Convlayer` is the user-defined name for the layer.

Listing I.1. CNN model layers.

```
model.layers
# [<tensorflow.python.keras.layers.convolutional.
    Conv1D at 0x149fd59b0>,
# <tensorflow.python.keras.layers.core.Dropout at 0
    x149fd5780>,
# <tensorflow.python.keras.layers.convolutional.
    Conv1D at 0x14b123358>,
# <tensorflow.python.keras.layers.pooling.
    MaxPooling1D at 0x14d77d048>,
# <tensorflow.python.keras.layers.core.Flatten at 0
    x149d86198>,
# <tensorflow.python.keras.layers.core.Dense at 0
    x14cd69c50>,
# <tensorflow.python.keras.layers.core.Dense at 0
    x14c990a90>]
```

Listing I.2 fetches the convolutional filters and scales them in (0,1) for visualization. They are then plotted in Figure I.1. The plots have a clearer interpretation of image problems. In such problems, the filters have shapes that correspond to certain patterns in the objects.

The filter visuals here can be interpreted differently. Each filter is of shape (4, 69) corresponding to the kernel size and the input features, respectively. The plot shows which feature is active in a filter.

Besides, there are a total of 64 filters in the convolutional layer (see Listing H). Out of them, 16 filters are shown in the figure.

Listing I.2. CNN filter plotting.

```
## Plot filters

# retrieve weights from the first convolutional
    layer
filters, biases = model.layers[0].get_weights()
print(model.layers[0].name, filters.shape)
# Convlayer (4, 69, 64)

# normalize filter values to 0-1 so we can visualize
     them
f_min, f_max = filters.min(), filters.max()
filters = (filters - f_min) / (f_max - f_min)

```

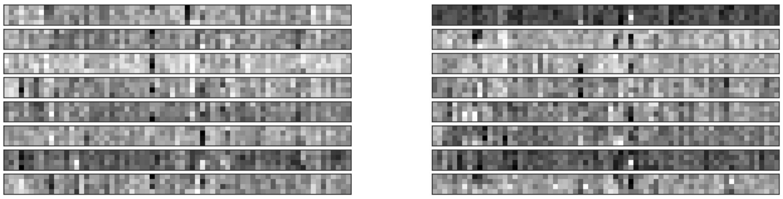


Figure I.1. *Convolutional layer filters visualization.*

```python
from matplotlib import pyplot
from matplotlib.pyplot import figure
figure(num=None, figsize=(20,10), dpi=160, facecolor
    ='w', edgecolor='k')
# plot first 16 filters
n_filters, ix = 16, 1
for i in range(n_filters):
    # get the filter
    f = filters[:, :, i]
    # plot each channel separately

    # specify subplot and turn of axis
    ax = pyplot.subplot(n_filters, 2, ix)
    ax.set_xticks([])
    ax.set_yticks([])
    # plot filter channel in grayscale
    pyplot.imshow(f[:, :], cmap='gray')
    ix += 1
# show the figure
pyplot.show()
```

Feature Representation Visuals

Visualizing the feature representations helps to learn the model's reactions to positive and negative samples.

In Listing I.3, a few true positive and true negative samples are taken based on the model inferences. Out of all the true positives, the ones with high probabilities are taken for better diagnosis.

The feature representation outputted by the first convolutional layer in Listing H is taken here. The steps for fetching the feature representations (map) are shown in Listing I.3.

As was shown in the model summary in Figure H.1, the output of the convolution layer is a feature representation of shape 237×64.

We visualize these 237×64 outputs for the true positive and true negative samples in Figure I.2a-I.2b.

Listing I.3. Convolutional network feature representation plotting.

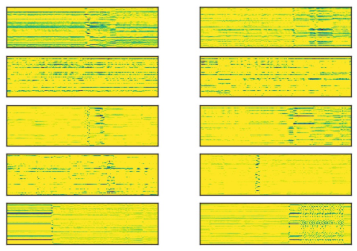
```python
## Plot feature map
# Take out a part of the model to fetch the feature
    mapping
# We are taking the feature mapping from the first
    Convolutional layer
feature_mapping = Model(inputs=model.inputs, outputs
    =model.layers[0].output)

prediction_valid = model.predict(X_valid_scaled).
    squeeze()

top_true_positives = np.where(
    np.logical_and(prediction_valid > 0.78, y_valid
        == 1))[0]

top_true_negatives = np.where(
    np.logical_and(prediction_valid < 1e-10, y_valid
         == 0))[0]

# Plotting
from matplotlib import pyplot
from matplotlib.pyplot import figure
figure(num=None, figsize=(4, 4), dpi=160, facecolor=
    'w')


def plot_feature_maps(top_predictions):

    n_feature_maps, ix = 10, 1

    for i in range(n_feature_maps):

         samplex = X_valid_scaled[top_predictions[i], :, :]
         samplex = samplex.reshape((1, samplex.shape[0], samplex.shape[1]))

         feature_map = feature_mapping.predict(samplex).squeeze()

         ax = pyplot.subplot(np.round(n_feature_maps / 2), 2, ix)
         ax.set_xticks([])
         ax.set_yticks([])

         # plot filter channel in grayscale
         pyplot.imshow(np.transpose(1 - (feature_map - feature_map.min()) /
                                     (feature_map.max() - feature_map.min())),
                       cmap='viridis')
         ix += 1

    # show the figure
    pyplot.show()

plot_feature_maps(top_true_positives)

plot_feature_maps(top_true_negatives)
```

In the figure, the yellow indicates the feature is active while the opposite for green. At a high level, it can be interpreted that most of the features are activated for true positives but not for true negatives. Meaning, the activation of these features distinguishes a positive (sheet break) from a normal process (no sheet break).

However, the true positive feature map on the top-left in Figure I.2a does not follow this interpretation. To further diagnose, subsequent layers should be visualized.

These visualizations help diagnose the model. The diagnosis can help in model improvement, new model development, or root cause analysis.

A different set of samples can be chosen, e.g., false positives, or false negatives, to diagnose the model to identify what happens when it is unable to correctly predict.


(a) *True positives.*

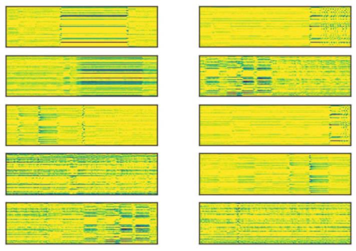

(b) *True negatives.*

Figure I.2. *Convolutional network feature representation plotting.*

Appendix J

Multiple (Maximum and Range) Pooling Statistics in a Convolution Network

The concept of using summary statistics for pooling described in Chapter 6 opened possibilities for new ways of pooling. One of them is using ancillary statistics in parallel with a sufficient statistic.

In this appendix, a network is constructed in Listing J with *maximum* pooling (a sufficient statistic) and *range* pooling (an ancillary statistic) in parallel.

The network structure is shown in Figure J.1. As shown here, the ReLU (nonlinear) activation is added after the pooling. This is essential as described in § 6.12.2. Otherwise, if activation is added between convolutional and pooling layers (following the tradition), the *range* statistic becomes the same as the *maximum* whenever the feature representations have any negative value.

```
## Multiple-pooling layer convolutional network
x = Input(shape=(TIMESTEPS, N_FEATURES))

conv = Conv1D(filters=16,
              kernel_size=4,
              activation='linear',
```

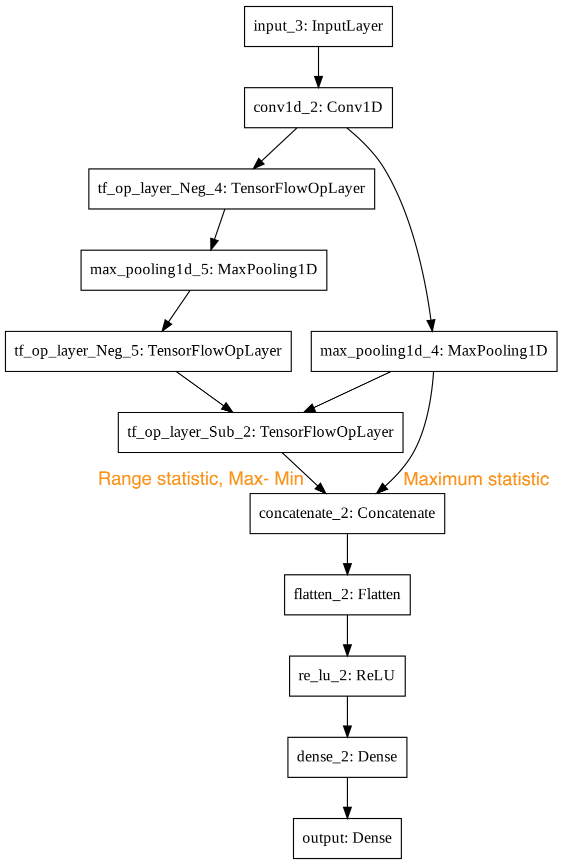

input_3: InputLayer
conv1d_2: Conv1D
tf_op_layer_Neg_4: TensorFlowOpLayer
max_pooling1d_5: MaxPooling1D
tf_op_layer_Neg_5: TensorFlowOpLayer
max_pooling1d_4: MaxPooling1D
tf_op_layer_Sub_2: TensorFlowOpLayer
Range statistic, Max- Min
Maximum statistic
concatenate_2: Concatenate
flatten_2: Flatten
re_lu_2: ReLU
dense_2: Dense
output: Dense

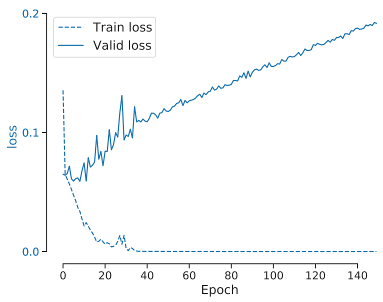
Figure J.1. *A convolutional network with maximum and range pooling placed in parallel.*

```
                     padding='valid')(x)

# left operations
max_statistic = MaxPool1D(pool_size=4,
                                       padding='valid')(conv)

# right operations
# 1. negative of feature map
range_statistic = tf.math.negative(conv)
# 2. apply maxpool to get the min statistics
range_statistic = MaxPool1D(pool_size=4,
                                 padding='valid')(
                                      range_statistic)
# 3. negative of negative in step (1) to revert to
    original
range_statistic = tf.math.negative(range_statistic)
# 4. subtract with max_statistic to get the
# range statistic max(x) - min(x)
range_statistic = tf.math.subtract(max_statistic,
                                            range_statistic)

# Concatenate the pool
concatted =
tf.keras.layers.Concatenate()([max_statistic,
                                               range_statistic])

features = Flatten()(concatted)

features = ReLU()(features)

# 128 nodes for lookback = 20 or 40.
dense = Dense(units=256,
                     activation='relu')(features)

predictions = Dense(units=1,
                              activation='sigmoid',
                              name='output')(dense)

model = Model(inputs=x,
                     outputs=predictions)

# Plot the network structure
```

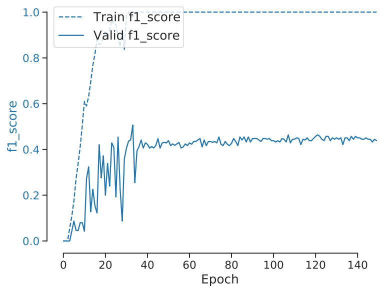
```
tf.keras.utils.plot_model(model,
                          show_shapes=False,
                          dpi=600)

# Train the model
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=[
                  'accuracy',
                  tf.keras.metrics.Recall(),
                  pm.F1Score(),
                  pm.FalsePositiveRate()
              ])
history = model.fit(x=X_train_scaled,
                    y=y_train,
                    batch_size=128,
                    epochs=150,
                    validation_data=(X_valid_scaled,
                                     y_valid),
                    verbose=1).history
```

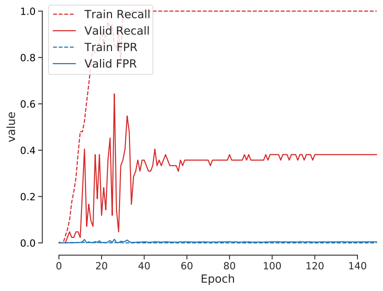
The network construction in Listing J shows a work-around with existing TensorFlow functions to get the range statistic. A custom range-pooling layer can also be built.

The results from training the network on a temporalized data with `lookback=240` shown in Figure J.2a-J.2c have a more stable accuracy performance than every other model. The stability could be attributed to the informative features provided by the parallel pooling.



(a) *Loss.*





(b) *F1-score.*

(c) *Recall and FPR.*

Figure J.2. *Convolutional network with range and maximum pooling results.*

Appendix K

Convolutional Autoencoder-Classifier

One of the applications of autoencoders is learning encodings that are used to train a classifier. In this appendix, an illustrative example is presented with the steps.

Sparse Convolutional Autoencoder

A sparse autoencoder is preferred for use in a classifier. The sparsity is applied on the encodings as shown in Listing K.1. The implementation approach here is different from the implementations in Chapter 7 to show another way to construct an autoencoder.

Listing K.1. Sparse convolutional autoencoder.

```python
# Sparse CNN Autoencoder

inputs = Input(shape=(TIMESTEPS,
                      N_FEATURES),
               name='encoder_input')

def encoder(inp):
  '''
  Encoder.

```

```
    Input
    inp   A tensor of input data.

    Process
    Extract the essential features of the input as
    its encodings by filtering it through convolutional
    layer(s). Pooling can also be used to further
    summarize the features.

    A linearly activated dense layer is added as the
    final layer in encoding to perform any affine
    transformation required. The dense layer is not
    for any feature extraction. Instead, it is only
    to make the encoding and decoding connections
    simpler for training.

    Output
    encoding  A tensor of encodings.
    '''

    # Multiple (conv, pool) blocks can be added here
    conv1 = Conv1D(filters=N_FEATURES,
                   kernel_size=4,
                   activation='relu',
                   padding='same',
                   name='encoder-conv1')(inp)
    pool1 = MaxPool1D(pool_size=4,
                      strides=1,
                      padding='same',
                      name='encoder-pool1')(conv1)

    # The last layer in encoding
    encoding = Dense(units=N_FEATURES,
                     activation='linear',
                     activity_regularizer=
                      tf.keras.regularizers.L1(l1=0.01),
                     name='encoder-dense1')(pool1)

    return encoding

```

```python
def decoder(encoding):
  '''
  Decoder.

  Input
  encoding    The encoded data.

  Process
  The decoding process requires a transposed
  convolutional layer, a.k.a. a deconvolution
  layer. Decoding must not be done with a
  regular convolutional layer. A regular conv
  layer is meant to extract a downsampled
  feature map. Decoding, on the other hand,
  is reconstruction of the original data
  from the downsampled feature map. A
  regular convolutional layer would try to
  extract further higher level features from
  the encodings instead of a reconstruction.

  For a similar reason, pooling must not be
  used in a decoder. A pooling operation is
  for summarizing a data into a few summary
  statistics which is useful in tasks such
  as classification. The purpose of
  decoding is the opposite, i.e., reconstruct
  the original data from the summarizations.
  Adding pooling in a decoder makes it lose
  the variations in the data and,
  hence, a poor reconstruction.

  If the purpose is only reconstruction, a
  linear activation should be used in
  decoding. A nonlinear activation is useful
  for predictive features but not for
  reconstruction.

  Batch normalization helps a decoder by
  preventing the reconstructions
  from exploding.

  Output
  decoding    The decoded data.

  '''

  convT1 = Conv1DTranspose(filters=N_FEATURES,
                           kernel_size=4,
                           activation='linear',
                           padding='same')(encoding)

  decoding = BatchNormalization()(convT1)

  decoding = Dense(units=N_FEATURES,
                   activation='linear')(decoding)

  return decoding

autoencoder = Model(inputs=inputs,
                    outputs=decoder(encoder(inputs))
                       )

autoencoder.summary()
autoencoder.compile(loss='mean_squared_error',
   optimizer = 'adam')

history =
    autoencoder.fit(x=X_train_y0_scaled,
                    y=X_train_y0_scaled,
                    epochs=100,
                    batch_size=128,
                    validation_data=
                        (X_valid_y0_scaled,
                         X_valid_y0_scaled),
                    verbose=1).history
```

Convolutional Classifier Initialized with Encoder

A classifier can be (potentially) enhanced with an autoencoder. Listing K.2 constructs a feed-forward convolutional classifier with an encoder attached to it.

Listing K.2. Convolutional feed-forward network initialized with autoencoder.

```
# CNN Classifier initialized with Encoder

def fully_connected(encoding):
  conv1 = Conv1D(filters=16,
                 kernel_size=4,
                 activation='relu',
                 padding='valid',
                 name='fc-conv1')(encoding)
  pool1 = MaxPool1D(pool_size=4,
                    padding='valid',
                    name='fc-pool1')(conv1)
  flat1 = Flatten()(pool1)

  den = Dense(units=16,
              activation='relu')(flat1)

  output = Dense(units=1,
                 activation='sigmoid',
                 name='output')(den)

  return(output)

encoding = encoder(inputs)
classifier =
    Model(inputs=inputs,
          outputs=fully_connected(
              encoding=encoder(inputs)))
```

The encoder can be used to have either a,

- **Pre-trained classifier**. A trained encoder can be used as a part of a feed-forward classifier network. Or,
- **Encoded features as input**. The features produced by an encoder used as input to a classifier.

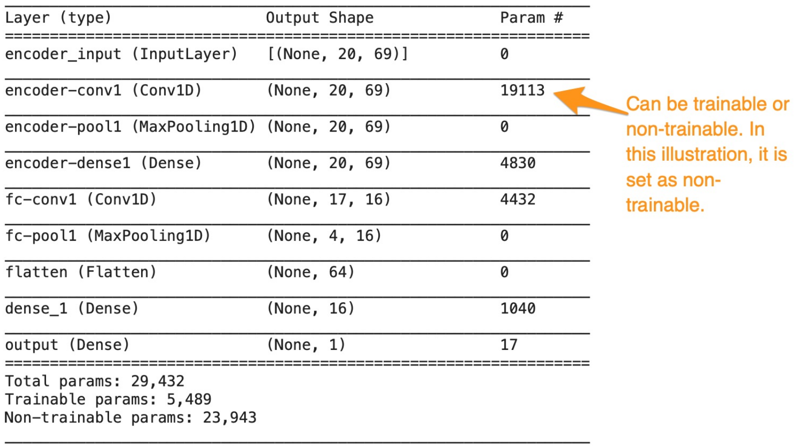
Corresponding to the two approaches, an argument `retrain_encoding` is defined in Listing K.3.

The argument when set to `False` results in the classifier using the **encoded features as input**. This is achieved by making the layers in the encoder section of the model as non-trainable in line 13 in the listing (Listing K.3). This is also shown in Figure K.1.

The argument, `retrain_encoding`, when set to `True` uses the encoding weights to initialize the model and retrain them while learning the classifier.

Listing K.3. Training an autoencoder-classifier.

```
 1  # Classifier layer initialized with encoder
 2  retrain_encoding = False
 3
 4  for classifier_layer in classifier.layers:
 5    for autoencoder_layer in autoencoder.layers:
 6      if classifier_layer.name == autoencoder_layer.
          name:
 7        # Set the weights of classifier same as the
 8        # corresponding autoencoder (encoder) layer
 9        classifier_layer.set_weights(
10            autoencoder_layer.get_weights())
11
12        if retrain_encoding == False:
13          classifier_layer.trainable = False
14        print(classifier_layer.name +
15              ' in classifier set to ' +
16              autoencoder_layer.name +
17              ' in the encoder' +
18              'is trainable: ' +
19              str(classifier_layer.trainable))
20
21  classifier.summary()
22  classifier.compile(optimizer='adam',
23                     loss='binary_crossentropy',
24                     metrics=[
25                         'accuracy',
26                         tf.keras.metrics.Recall(),
27                         pm.F1Score(),
28                         pm.FalsePositiveRate()
29                     ])
30
31  history = classifier.fit(x=X_train_scaled,
32                           y=y_train,
33                           batch_size=128,
```



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
encoder_input (InputLayer)   [(None, 20, 69)]          0
_________________________________________________________________
encoder-conv1 (Conv1D)       (None, 20, 69)            19113
_________________________________________________________________
encoder-pool1 (MaxPooling1D) (None, 20, 69)            0
_________________________________________________________________
encoder-dense1 (Dense)       (None, 20, 69)            4830
_________________________________________________________________
fc-conv1 (Conv1D)            (None, 17, 16)            4432
_________________________________________________________________
fc-pool1 (MaxPooling1D)      (None, 4, 16)             0
_________________________________________________________________
flatten (Flatten)            (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 16)                1040
_________________________________________________________________
output (Dense)               (None, 1)                 17
=================================================================
Total params: 29,432
Trainable params: 5,489
Non-trainable params: 23,943
_________________________________________________________________
```

Can be trainable or non-trainable. In this illustration, it is set as non-trainable.

Figure K.1. *Model summary of convolutional autoencoder-classifier using the encodings as classifier input.*

```
                                epochs=100,
                                validation_data=
                                      (X_valid_scaled,
                                       y_valid),
                                verbose=1).history
```

Appendix L

Oversampling

Oversampling techniques artificially increase the minority positive class samples to balance the data set. One of the most fundamental oversampling techniques is randomly selecting samples from the minority class with replacement. The random selection process is repeated multiple times to get as many minority samples as required for data balancing.

However, in an extremely unbalanced data, creating a large number of duplicates for the minority class may yield a biased model.

A more methodical approach called Synthetic Minority Over-sampling Technique (SMOTE) can address this.

SMOTE

In this approach, data is synthesized by randomly interpolating new points between the available (real) minority samples.

This procedure is illustrated in Figure L.1. In the figure, the $+$'s are the "rare" minority class and o's are the majority. SMOTE synthesizes a new minority sample between the existing samples. These interpolations work as follows,

- the interpolated point is drawn randomly from anywhere on the line (in the two-dimensional visual in Figure L.1, otherwise, a hyper-plane in general) that connects two samples denoted as $\mathbf{x}_1$

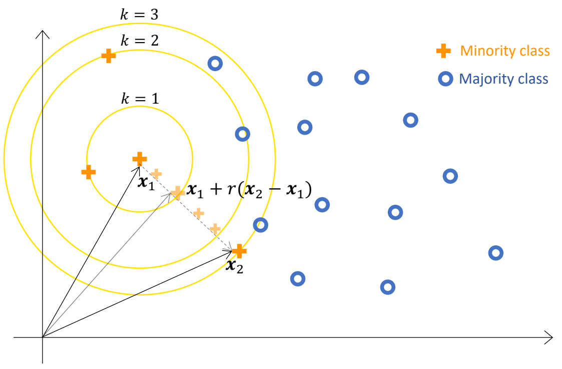


Figure L.1. *Illustration of SMOTE oversampling. SMOTE synthesizes a new sample (shown in translucent orange +) by interpolating between two samples.*

and $\mathbf{x}_2$.

- For this, a random number between 0 and 1 is generated. The new point is then synthesized as, $\mathbf{x}_1 + r(\mathbf{x}_2 - \mathbf{x}_1)$, shown as a translucent + in the figure.

- Note that, any interpolated sample between $\mathbf{x}_1$ and $\mathbf{x}_2$ will lie on the line connecting them (shown as small translucent +'s in the figure).

SMOTE is available in Python as `imblearn.over_sampling.SMOTE()` [1]. Its primary arguments are,

- `k_neighbors`. Denoted as k in Figure L.1, it is the number of nearest neighbors SMOTE will use to synthesize new samples. By default, $k = 5$. A lower k will have lesser noise but also less robust, and vice-versa for a higher k.

- `random_state` is used to control the randomization of the algorithm. It is useful to set this to reproduce the sampling.

[1] `https://imbalanced-learn.readthedocs.io/en/stable/generated/imblearn.over_sampling.SMOTE.html`

Listing L.1. SMOTE Oversampling.

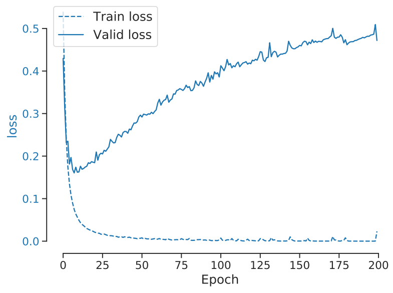
```python
smote = SMOTE(random_state=212, k_neighbors=1)
X_train_scaled_resampled, y_train_resampled =
  smote.fit_resample(X_train_scaled, y_train)
print('Resampled dataset shape %s' %
  Counter(y_train_resampled))
```

Listing L.1 shows the `SMOTE` declaration. Here, `k_neighbors=1` because the data is noisy and a larger k will add more noise. However, a small k comes at the cost of making the model biased, and therefore, potentially poorer inferencing accuracy.

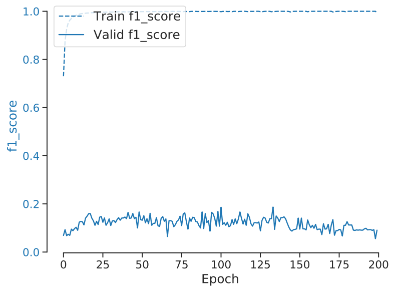
A model is built and trained with the oversampled data in Listing L.2. The results are shown in Figure L.2a-L.2c.

Listing L.2. MLP Model with SMOTE Oversampled Training Data.

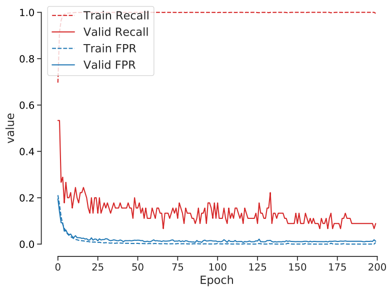
```python
model = Sequential()
model.add(Input(shape=(N_FEATURES, )))
model.add(Dense(32, activation='relu'))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.summary()

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy',
                       tf.keras.metrics.Recall(),
                       performancemetrics.F1Score(),
                       performancemetrics.
                          FalsePositiveRate()]
             )

history = model.fit(x=X_train_scaled_resampled,
                    y=y_train_resampled,
                    batch_size=128,
                    epochs=200,
                    validation_data=(X_valid_scaled,
                      y_valid),
```



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure L.2. *MLP with SMOTE oversampling model results.*