

# Practical Deep Learning at Scale with MLflow

Bridge the gap between offline experimentation  
and online production



**Yong Liu**

Foreword by Dr. Matei Zaharia, Chief Technologist, Databricks, and Co-Creator of MLflow



# Practical Deep Learning at Scale with MLflow

Bridge the gap between offline experimentation and online production

Yong Liu

**Packt**>

BIRMINGHAM—MUMBAI

# Practical Deep Learning at Scale with MLflow

Copyright © 2022 Packt Publishing

*All rights reserved.* No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Publishing Product Manager:** Dhruv Jagdish Kataria

**Senior Editor:** Tazeen Shaikh

**Content Development Editor:** Manikandan Kurup

**Technical Editor:** Devanshi Ayare

**Copy Editor:** Safis Editing

**Project Coordinator:** Farheen Fathima

**Proofreader:** Safis Editing

**Indexer:** Rekha Nair

**Production Designer:** Jyoti Chauhan

**Marketing Coordinators:** Shifa Ansari and Abeer Riyaz Dawe

First published: July 2022

Production reference: 1160622

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80324-133-3

[www.packt.com](http://www.packt.com)

*To my father and the memory of my mother for their sacrificial love,  
prayers, and life-long support.*

*- Yong Liu*

# Foreword

I am thrilled to introduce this book on Practical Deep Learning at Scale with MLflow by Dr. Yong Liu. Deep learning has been revolutionizing many areas of computing in the past decade, but good resources for using it in production applications remain scarce. At the same time, practitioners have realized that designing machine learning (ML) applications to be operable, maintainable, and updateable is one of the hardest parts of using ML in production, leading to the new field of MLOps. Dr. Liu tackles these issues head-on by showing you how to build robust and maintainable deep learning applications using MLflow, a widely-used open source MLOps framework, and multiple state-of-the-art methods and software tools.

Dr. Liu brings a wealth of experience in production machine learning that shines through in every chapter of the book. He has been working in large-scale computing since his Ph.D., he has built large-scale production ML applications at Microsoft, Maana, and Outreach, and he has published multiple research papers on deep learning. This means that each chapter recommends practical approaches that have worked in multiple organizations. Dr. Liu also presents all his material clearly to tell you the tradeoffs in each decision, illustrates all the ideas through runnable code and surveys multiple open source and commercial tools for each task.

As one of the original creators of MLflow, I was very excited that Dr. Liu chose MLflow as the MLOps framework for this book. When we started MLflow in 2018, there was no widely used open-source MLOps framework, so we designed a highly extensible framework that can be integrated with a wide variety of other tools and services and customized to each organization's workflow. We've been thrilled with the fast growth of the MLflow open source community since then and with the powerful integrations that the community has contributed to libraries including PyTorch, SHAP, Delta Lake, and others. Dr. Liu's team was one of the early users of MLflow, so he is an expert on how to use the framework in practice. I hope that you enjoy learning from his experience and building groundbreaking applications using the latest techniques in deep learning.

*Dr. Matei Zaharia*

*Chief Technologist, Databricks, and Co-Creator of MLflow*

# Contributors

## About the author

**Yong Liu** has been working in big data science, machine learning, and optimization since his doctoral student years at the **University of Illinois at Urbana-Champaign (UIUC)** and later as a senior research scientist and principal investigator at the **National Center for Supercomputing Applications (NCSA)**, where he led data science R&D projects funded by the National Science Foundation and Microsoft Research. He then joined Microsoft and AI/ML start-ups in the industry. He has shipped ML and DL models to production and has been a speaker at the Spark/Data+AI summit and NLP summit. He has recently published peer-reviewed papers on deep learning, linked data, and knowledge-infused learning at various ACM/IEEE conferences and journals.

*I want to thank my wife and my two teenage kids for their support and encouragement during the time of writing this book. I am also grateful for those collaborators, team members, and mentors at Outreach Corporation whom I have learned a lot from.*

## About the reviewers

**Dr. Pavel Dmitriev** received a B.S. degree in applied mathematics from Moscow State University in 2002, and a Ph.D. degree in computer science from Cornell University in 2008. He previously worked as an engineer and a data scientist at Yahoo and Microsoft. He is currently a vice president of data science at Outreach where he works on enabling data-driven decision-making in sales through machine learning and experimentation. Pavel's research was presented at a number of international conferences such as KDD, ICSE, WWW, CIKM, BigData, and SEAA. A certified yoga and meditation instructor, he actively works on improving physical and mental well-being in corporations through classes and workshops.

**Hong Yung (Joey) Yip** is a Ph.D. candidate in computer science at the **Artificial Intelligence Institute (AIISC)**, University of South Carolina. His research interests are the areas of knowledge-infused learning, which intertwines AI and knowledge graphs to enhance neural networks in performance, interpretability, and explainability for dynamic and real-time domains. He has co-authored and published at top venues (WWW, ISWC, and IEEE). He has previously interned at the National Library of Medicine, Bethesda MD, on developing scalable approaches for biomedical vocabulary alignment, and with Outreach Corporation, Seattle WA, on conceptualizing a Sales Engagement Graph framework for temporal pattern discovery and contextual understanding in sales processes.

# Table of Contents

## Preface

---

## Section 1 – Deep Learning Challenges and MLflow Prime

### 1

#### Deep Learning Life Cycle and MLOps Challenges

---

Technical requirements	4	Understanding DL model challenges	17
Understanding the DL life cycle and MLOps challenges	5	Understanding DL code challenges	18
Implementing a basic DL sentiment classifier	7	Understanding DL explainability challenges	19
Understanding DL's full life cycle development	9	Summary	22
Understanding MLOps challenges	12	Further reading	23
Understanding DL data challenges	15		

### 2

#### Getting Started with MLflow for Deep Learning

---

Technical requirements	26	Implementing our first DL experiment with MLflow autologging	30
Setting up MLflow	26	Exploring MLflow's components and usage patterns	35
Setting up MLflow locally using miniconda	27	Exploring experiments and runs in MLflow	35
Setting up MLflow to interact with a remote MLflow server	29		



Exploring MLflow models and their usages	39	Summary	44
Exploring MLflow code tracking and its usages	43	Further reading	45

## Section 2 – Tracking a Deep Learning Pipeline at Scale

### 3

#### Tracking Models, Parameters, and Metrics

---

Technical requirements	50	Understanding the open provenance tracking framework	54
Setting up a full-fledged local MLflow tracking server	51	Implementing MLflow model tracking	55
Tracking model provenance	53	Tracking model metrics	63
		Tracking model parameters	66
		Summary	68
		Further reading	69

### 4

#### Tracking Code and Data Versioning

---

Technical requirements	72	Tracking data versioning in Delta Lake	91
Tracking notebook and pipeline versioning	72	An example of tracking data using MLflow	93
Pipeline tracking	78		
Tracking locally, privately built Python libraries	88	Summary	95
		Further reading	96

## Section 3 – Running Deep Learning Pipelines at Scale

### 5

#### Running DL Pipelines in Different Environments

---

Technical requirements	100	Running local code remotely in the cloud	109
An overview of different execution scenarios and environments	101	Running remotely in the cloud with remote code in GitHub	118
Running locally with local code	104	Summary	122
Running remote code in GitHub locally	107	Further reading	123

### 6

#### Running Hyperparameter Tuning at Scale

---

Technical requirements	126	Creating the Ray Tune trainable for the DL model	137
Understanding automatic HPO for DL pipelines	127	Creating the Ray Tune HPO run function	142
Types of hyperparameters and their challenges	127	Running the first Ray Tune HPO experiment with MLflow	145
How HPO works and which ones to choose	130	Running HPO with Ray Tune using Optuna and HyperBand	147
Creating HPO-ready DL models with Ray Tune and MLflow	134	Summary	151
Setting up Ray Tune and MLflow	136	Further reading	151

---

## Section 4 - Deploying a Deep Learning Pipeline at Scale

### 7

#### Multi-Step Deep Learning Inference Pipeline

---

Technical requirements	156	Implementing language detection preprocessing logic	169
Understanding patterns of DL inference pipelines	156	Implementing caching preprocessing and postprocessing logic	171
Understanding the MLflow Model Python Function API	159	Implementing response composition postprocessing logic	172
Implementing a custom MLflow Python model	162	Implementing an inference pipeline as a new entry point in the main MLproject	175
Implementing preprocessing and postprocessing steps in a DL inference pipeline	169	Summary	178
		Further reading	179

### 8

#### Deploying a DL Inference Pipeline at Scale

---

Technical requirements	182	Step 1: Build a local SageMaker Docker image	194
Understanding different deployment tools and host environments	182	Step 2: Add additional model artifacts layers onto the SageMaker Docker image	195
Deploying locally for batch and web service inference	185	Step 3: Test local deployment with the newly built SageMaker Docker image	197
Batch inference	185	Step 4: Push the SageMaker Docker image to AWS Elastic Container Registry	199
Model as a web service	188	Step 5: Deploy the inference pipeline model to create a SageMaker endpoint	202
Deploying using Ray Serve and MLflow deployment plugins	190	Step 6: Query the SageMaker endpoint for online inference	205
Deploying to AWS SageMaker – a complete end-to-end guide	193	Summary	208
		Further reading	208

## Section 5 – Deep Learning Model Explainability at Scale

### 9

#### Fundamentals of Deep Learning Explainability

---

Technical requirements	214	Problem type: what is the machine learning problem type	219
Understanding the categories and audience of explainability	215	Objectives type: what is the motivation or goal to explain	219
Audience: who needs to know	216	Method type: what is the specific post-hoc explanation method used	220
Stage: when to provide an explanation in the DL life cycle	217	Exploring the SHAP Explainability toolbox	221
Scope: which prediction needs explanation	217	Exploring the Transformers Interpret toolbox	226
Input data format: what is the format of the input data	218	Summary	228
Output data format: what is the format of the output explanation	218	Further reading	229

### 10

#### Implementing DL Explainability with MLflow

---

Technical requirements	232	Creating and logging an MLflow pyfunc explainer	242
Understanding current MLflow explainability integration	232	Deploying an MLflow pyfunc explainer for an EaaS	246
Implementing a SHAP explanation using the MLflow artifact logging API	235	Using an MLflow pyfunc explainer for batch explanation	248
Implementing a SHAP explainer using the MLflow pyfunc API	241	Summary	253
		Further reading	253

---

#### Index

---

#### Other Books You May Enjoy

---



# Preface

Starting from AlexNet in 2012, which won the large-scale ImageNet competition, to the BERT pre-trained language model in 2018, which topped many **natural language processing (NLP)** leaderboards, the revolution of modern **deep learning (DL)** in the broader **artificial intelligence (AI)** and **machine learning (ML)** community continues. Yet, the challenges of moving these DL models from offline experimentation to a production environment remain. This is largely due to the complexity and lack of a unified open source framework for supporting the full life cycle development of DL. This book will help you understand the big picture of DL full life cycle development, and implement DL pipelines that can scale from a local offline experiment to a distributed environment and online production clouds, with an emphasis on hands-on project-based learning to support the end-to-end DL process using the popular open source MLflow framework.

The book starts with an overview of the DL full life cycle and the emerging **machine learning operations (MLOps)** field, providing a clear picture of the four pillars of DL (data, model, code, and explainability) and the role of MLflow in these areas. A basic transfer learning-based NLP sentiment model using PyTorch Lightning Flash is built in the first chapter, which is further developed, tuned, and deployed to production throughout the rest of the book. From there onward, it guides you step-by-step to understand the concept of MLflow experiments and usage patterns, using MLflow as a unified framework to track DL data, code and pipeline, model, parameters, and metrics at scale. We'll run DL pipelines in a distributed execution environment with reproducibility and provenance tracking, and tune DL models through **hyperparameter optimization (HPO)** with Ray Tune, Optuna and HyperBand. We'll also build a multi-step DL inference pipeline with preprocessing and postprocessing steps, deploy a DL inference pipeline for production using Ray Serve and AWS SageMaker, and finally, provide a DL Explanation-as-a-Service using **SHapley Additive exPlanations (SHAP)** and MLflow integration.

By the end of this book, you'll have the foundation and hands-on experience to build a DL pipeline from initial offline experimentation to final deployment and production, all within a reproducible and open source framework. Along the way, you will also learn the unique challenges with DL pipelines and how we overcome them with practical and scalable solutions such as using multi-core CPUs, **graphical processing units (GPUs)**, distributed and parallel computing frameworks, and the cloud.

## Who this book is for

This book is written for data scientists, ML engineers, and AI practitioners who want to master the full life cycle of DL development from inception to production using the open source MLflow framework and related tools such as Ray Tune, SHAP, and Ray Serve. The scalable, reproducible, and provenance-aware implementations presented in this book ensure you build an enterprise-grade DL pipeline successfully. This book will support anyone building powerful DL cloud applications.

## What this book covers

*Chapter 1, Deep Learning Life Cycle and MLOps Challenges*, covers the five stages of the full life cycle of DL and the first DL model in this book using the transfer learning approach for text sentiment classification. It also defines the concept of MLOps along with the three foundation layers and four pillars, and the roles of MLflow in these areas. An overview of the challenges in DL data, model, code, and explainability are also presented. This chapter is designed to bring everyone to the same foundational level and provides clarity and guidelines on the scope of the rest of the book.

*Chapter 2, Getting Started with MLflow for Deep Learning*, serves as an MLflow primer and a first hands-on learning module to quickly set up a local filesystem-based MLflow tracking server or interact with a remote managed MLflow tracking server in Databricks, and perform a first DL experiment using MLflow auto logging. It also explains some foundational MLflow concepts through concrete examples such as experiments, runs, metadata about and the relationship between experiments and runs, code tracking, model logging, and model flavor. Specifically, we underline that experiments should be first-class entities that can be used to bridge the gap between the offline and online production life cycle of DL models. This chapter builds the foundational knowledge of MLflow.

*Chapter 3, Tracking Models, Parameters, and Metrics*, covers the first in-depth learning module on tracking using a fully-fledged local MLflow tracking server. It starts with setting up a local fully-fledged MLflow tracking server that runs in Docker Desktop, with a MySQL backend store and a MinIO artifact store. Before implementing tracking, this chapter provides an open provenance tracking framework based on the open provenance model vocabulary specification, and presents six types of provenance questions that could be implemented by using MLflow. It then provides hands-on implementation examples on how to use MLflow model-logging APIs and registry APIs to track model provenance, model metrics, and parameters, with or without auto logging. Unlike other typical MLflow API tutorials, which only provide guidance on using the APIs, this chapter instead focuses on how successfully we can use MLflow to answer the provenance questions. By the end of this chapter, we could answer four out of six provenance questions, and the remaining two questions can only be answered when we have a multi-step pipeline or deployment to production, which are covered in the later chapters.

*Chapter 4, Tracking Code and Data Versioning*, covers the second in-depth learning module on MLflow tracking. It analyzes the current practices on the usage of notebooks and pipelines in the ML/DL projects. It recommends using VS Code notebooks and shows a concrete DL notebook example that can be run either interactively or non-interactively with MLflow tracking enabled. It also recommends using MLflow's **MLproject** to implement a multi-step DL pipeline using MLflow's entry points and pipeline chaining. A three-step DL pipeline is created for DL model training and registration. In addition, it also shows the pipeline level tracking and individual step tracking through the parent-child nested run in MLflow. Finally, it shows how to track public and privately built Python libraries and data versioning in **Delta Lake** using MLflow.

*Chapter 5, Running DL Pipelines in Different Environments*, covers how to run a DL pipeline in different environments. It starts with the scenarios and requirements for executing DL pipelines in different environments. It then shows how to use MLflow's **command-line interface (CLI)** to submit runs in four scenarios: running locally with local code, running locally with remote code in GitHub, running remotely in the cloud with local code, and running remotely in the cloud with remote code in GitHub. The flexibility and reproducibility supported by MLflow to execute a DL pipeline also provide building blocks for **continuous integration/continuous deployment (CI/CD)** automation when needed.

*Chapter 6, Running Hyperparameter Tuning at Scale*, covers using MLflow to support HPO at scale using state-of-the-art HPO frameworks such as Ray Tune. It starts with a review of the types and challenges of DL pipeline hyperparameters. Then, it compares three HPO frameworks Ray Tune, Optuna, and HyperOpt, and provides a detailed analysis of the pros and cons and their integration maturity with MLflow. It then recommends and shows how to use Ray Tune with MLflow to do HPO tuning for the DL model we have been working on in this book so far. Furthermore, it covers how to switch to other HPO search and scheduler algorithms such as Optuna and HyperBand. This enables us to produce high-performance DL models that meet the business requirements in a cost-effective and scalable way.

*Chapter 7, Multi-Step Deep Learning Inference Pipeline*, covers creating a multi-step inference pipeline using MLflow's custom Python model approach. It starts with an overview of four patterns of inference workflows in production where a single trained model is usually not enough to meet the business application requirements. Additional preprocessing and postprocessing steps are needed. It then presents a step-by-step guide to implementing a multi-step inference pipeline that wraps the previously fine-tuned DL sentiment model with language detection, caching, and additional model metadata. This inference pipeline is then logged as a generic MLflow **PyFunc** model that can be loaded using the common MLflow PyFunc load API. Having an inference pipeline wrapped as an MLflow model opens doors for automation and consistent management of the model pipeline within the same MLflow framework.



*Chapter 8, Deploying a DL Inference Pipeline at Scale*, covers deploying a DL inference pipeline into different host environments for production usage. It starts with an overview of the landscape of deployment and hosting environments including batch inference and streaming inference at scale. It then describes the different deployment mechanisms such as MLflow built-in model serving tools, custom deployment plugins, and generic model serving frameworks such as Ray Serve. It shows examples of how to deploy a batch inference pipeline using MLflow's Spark **user-defined function (UDF)**, and how to serve a DL inference pipeline as a local web service using either MLflow's built-in model serving tool or Ray Serve's MLflow deployment plugin, `mlflow-ray-serve`. It then describes a complete step-by-step guide to deploying a DL inference pipeline to a managed AWS SageMaker instance for production usage.

*Chapter 9, Fundamentals of Deep Learning Explainability*, covers the foundational concepts of explainability and exploration of using two popular explainability tools. It starts with an overview of the eight dimensions of explainability and **explainable AI (XAI)**, then provides concrete learning examples to explore the usage of SHAP and Transformers-interpret toolboxes for an NLP sentiment pipeline. It emphasizes that explainability should be lifted to be the first-class artifact when developing a DL application since there are increasing demands and expectations for model and data explanation in various business applications and domains.

*Chapter 10, Implementing DL Explainability with MLflow*, covers how to implement DL explainability using MLflow to provide **Explanation-as-a-Service (EaaS)**. It starts with an overview of MLflow's current capability to support explainers and explanations. Specifically, the existing integration with SHAP in MLflow APIs does not support DL explainability at scale. Therefore, it provides two generic ways of using MLflow's artifact logging APIs and **PyFunc** APIs for the implementation. Examples are provided for implementing SHAP explanation, which logs the SHAP value in a bar chart in an MLflow tracking server's artifact store. A SHAP explainer can be logged as an MLflow Python model, and then loaded as either a Spark UDF for batch explanation or as a web service for online EaaS. This provides maximal flexibility within a unified MLflow framework for implementing explainability.

---

## To get the most out of this book

The majority of the code in this book can be implemented and executed using the open source MLflow tool, with a few exceptions where a 14-day full Databricks trial is needed (sign up at <https://databricks.com/try-databricks>) along with an AWS Free Tier account (sign up at <https://aws.amazon.com/free/>). The following lists some major software packages covered in this book:

- MLflow 1.20.2 and above
- Python 3.8.10
- Lightning-flash 0.5.0
- Transformers 4.9.2
- SHAP 0.40.0
- PySpark 3.2.1
- Ray[tune] 1.9.2
- Optuna 2.10.0

The complete package dependencies are listed in each chapter's `requirements.txt` file or the `conda.yaml` file in this book's GitHub repository. All code has been tested to run successfully in a macOS or Linux environment. If you are a Microsoft Windows user, it is recommended to install **WSL2** to run the bash scripts provided in this book: <https://www.windowscentral.com/how-install-wsl2-windows-10>. It is a known issue that the MLflow CLI does not work properly in the Microsoft Windows command line.

Starting from *Chapter 3, Tracking Models, Parameters, and Metrics* of this book, you will also need to have Docker Desktop (<https://www.docker.com/products/docker-desktop/>) installed to set up a fully-fledged local MLflow tracking server for executing the code in this book. AWS SageMaker is needed in *Chapter 8, Deploying a DL Inference Pipeline at Scale*, for the cloud deployment example. VS Code version 1.60 or above ([https://code.visualstudio.com/updates/v1\\_60](https://code.visualstudio.com/updates/v1_60)) is used as the **integrated development environment (IDE)** in this book. Miniconda version 4.10.3 or above (<https://docs.conda.io/en/latest/miniconda.html>) is used throughout this book for creating and activating virtual environments.

**If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.**

Finally, to get the most out of this book, you should have experience in programming in Python and have a basic understanding of popular ML and data manipulation libraries such as pandas and PySpark.

## Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow>. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

## Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: [https://static.packt-cdn.com/downloads/9781803241333\\_ColorImages.pdf](https://static.packt-cdn.com/downloads/9781803241333_ColorImages.pdf).

## Conventions used

There are a number of text conventions used throughout this book.

**Code in text:** Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "For learning purposes, we have provided two example `mlruns` artifacts and the `huggingface` cache folder in the GitHub repository under the `chapter08` folder."

A block of code is set as follows:

```
client = boto3.client('sagemaker-runtime')
response = client.invoke_endpoint(
    EndpointName=app_name,
    ContentType=content_type,
    Accept=accept,
    Body=payload
)
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
loaded_model = mflow.pyfunc.spark_udf(  
    spark,  
    model_uri=logged_model,  
    result_type=StringType())
```

Any command-line input or output is written as follows:

```
mflow models serve -m models:/inference_pipeline_model/6
```

**Bold:** Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "To execute the code in this cell, you can just click on **Run Cell** in the top-right drop-down menu."

#### Tips or Important Notes

Appear like this.

## Get in touch

Feedback from our readers is always welcome.

**General feedback:** If you have questions about any aspect of this book, email us at [customercare@packtpub.com](mailto:customercare@packtpub.com) and mention the book title in the subject of your message.

**Errata:** Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit [www.packtpub.com/support/errata](http://www.packtpub.com/support/errata) and fill in the form.

**Piracy:** If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [copyright@packt.com](mailto:copyright@packt.com) with a link to the material.

**If you are interested in becoming an author:** If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit [authors.packtpub.com](http://authors.packtpub.com).

## Share Your Thoughts

Once you've read *Practical Deep Learning at Scale with MLflow*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

# Section 1 - Deep Learning Challenges and MLflow Prime

In this section, we will learn about the five stages of the full life cycle of **deep learning (DL)**, and understand the emerging field of **machine learning operations (MLOps)** and the role of MLflow. We will provide an overview of the challenges in the four pillars of a DL process: data, model, code, and explainability. Then, we will learn how to set up a basic local MLflow development environment and run our first MLflow experiment for a **natural language processing (NLP)** model built on top of **PyTorch Lightning Flash**. Finally, we will explain the foundational MLflow concepts such as experiments, runs, and many more, through this first MLflow experiment example.

This section comprises the following chapters:

- *Chapter 1, Deep Learning Life Cycle and MLOps Challenges*
- *Chapter 2, Getting Started with MLflow for Deep Learning*



# 1

# Deep Learning Life Cycle and MLOps Challenges

The past few years have seen great success in **Deep Learning (DL)** for solving practical business, industrial, and scientific problems, particularly for tasks such as **Natural Language Processing (NLP)**, image, video, speech recognition, and conversational understanding. While research in these areas has made giant leaps, bringing these DL models from offline experimentation to production and continuously improving the models to deliver sustainable values is still a challenge. For example, a recent article by VentureBeat (<https://venturebeat.com/2019/07/19/why-do-87-of-data-science-projects-never-make-it-into-production/>) found that 87% of data science projects never make it to production. While there might be business reasons for such a low production rate, a major contributing factor is the difficulty caused by the lack of experiment management and a mature model production and feedback platform.



This chapter will help us to understand the challenges and bridge these gaps by learning the concepts, steps, and components that are commonly used in the full life cycle of DL model development. Additionally, we will learn about the challenges of an emerging field known as **Machine Learning Operations (MLOps)**, which aims to standardize and automate ML life cycle development, deployment, and operation. Having a solid understanding of these challenges will motivate us to learn the skills presented in the rest of this book using MLflow, an open source, ML full life cycle platform. The business values of adopting MLOps' best practices are numerous; they include faster time-to-market of model-derived product features, lower operating costs, agile A/B testing, and strategic decision making to ultimately improve customer experience. By the end of this chapter, we will have learned about the critical role that MLflow plays in the four pillars of MLOps (that is, data, model, code, and explainability), implemented our first working DL model, and grasped a clear picture of the challenges with data, models, code, and explainability in DL.

In this chapter, we're going to cover the following main topics:

- Understanding the DL life cycle and MLOps challenges
- Understanding DL data challenges
- Understanding DL model challenges
- Understanding DL code challenges
- Understanding DL explainability challenges

## Technical requirements

All of the code examples for this book can be found at the following GitHub URL: <https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow>.

You need to have Miniconda (<https://docs.conda.io/en/latest/miniconda.html>) installed on your development environment. In this chapter, we will walk through the process of installing the PyTorch lightning-flash library (<https://github.com/PyTorchLightning/lightning-flash>), which can be used to build our first DL model in the *Implementing a basic DL sentiment classifier* section. Alternatively, you can sign up for a free Databricks Community Edition account at <https://community.cloud.databricks.com/login.html> and use a GPU cluster and a notebook to carry out the model development described in this book.

In addition to this, if you are a Microsoft Windows user, we recommend that you install WSL2 (<https://www.windowscentral.com/how-install-wsl2-windows-10>) so that you have a Linux environment to run the command lines that are present in this book.

## Understanding the DL life cycle and MLOps challenges

Nowadays, the most successful DL models that are deployed in production primarily observe the following two steps:

1. **Self-supervised learning:** This refers to the pretraining of a model in a data-rich domain that does not require labeled data. This step produces a pretrained model, which is also called a **foundation model**, for example, BERT, GPT-3 for NLP, and VGG-NETS for computer vision.
2. **Transfer learning:** This refers to the fine-tuning of the pretrained model in a specific prediction task such as text sentiment classification, which requires labeled training data.

One ground-breaking and successful example of a DL model in production is the *Buyer Sentiment Analysis* model, which is built on top of BERT for classifying sales engagement email messages, providing critical fine-grained insights into buyer emotions and signals beyond simple activity metrics such as reply, click, and open rates (<https://www.prnewswire.com/news-releases/outreach-unveils-groundbreaking-ai-powered-buyer-sentiment-analysis-transforming-sales-engagement-301188622.html>). There are different variants regarding how this works, but in this book, we will primarily focus on the **Transfer Learning** paradigm of developing and deploying DL models, as it exemplifies a practical DL life cycle.

Let's walk through an example to understand a typical core DL development paradigm. For example, the popular BERT model released in late 2018 (a basic version of the BERT model can be found at <https://huggingface.co/bert-base-uncased>) was initially pretrained on raw texts (without human labeling) from over 11,000 books from BookCorpus and the entire English Wikipedia. This pretrained language model was then fine-tuned to many downstream NLP tasks, such as text classification and sentiment analysis, in different application domains such as movie review classifications by using labeled movie review data (<https://huggingface.co/datasets/imdb>). Note that sometimes, it might be necessary to further pretrain a foundation model (for example, BERT) within the application domain by using unlabeled data before fine-tuning to boost the final model performance in terms of accuracy. This core DL development paradigm is illustrated in *Figure 1.1*:

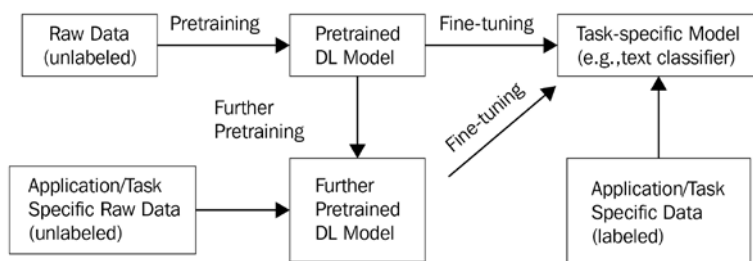


Figure 1.1 – A typical core DL development paradigm

Note that while *Figure 1.1* represents a common development paradigm, not all of these steps are necessary for a specific application scenario. For example, you might only need to do fine-tuning using a publicly available pretrained DL model with your labeled application-specific data. Therefore, you don't need to do your own pretraining or carry out further pretraining using unlabeled data since other people or organizations have already done the pretraining step for you.

#### DL over Classical ML

Unlike classical ML model development, where, usually, a feature engineering step is required to extract and transform raw data into features to train an ML model such as decision tree or logistic regression, DL can learn the features automatically, which is especially attractive for modeling unstructured data such as texts, images, videos, audio, and speeches. DL is also called *representational learning* due to this characteristic. In addition to this, DL is usually data- and compute-intensive, requiring **Graphics Process Units (GPUs)**, **Tensor Process Units (TPU)**, or other types of computing hardware accelerators for at-scale training and inference. Explainability for DL models is also harder to implement, compared with traditional ML models, although recent progress has now made that possible.

## Implementing a basic DL sentiment classifier

To set up the development of a basic DL sentiment classifier, you need to create a virtual environment in your local environment. Let's assume that you have **miniconda** installed. You can implement the following in your command-line prompt to create a new virtual environment called `dl_model` and install the PyTorch `lightning-flash` package so that the model can be built:

```
conda create -n dl_model python==3.8.10
conda activate dl_model
pip install lightning-flash[all]
```

Depending on your local machine's memory, the preceding commands might take about 10 minutes to finish. You can verify the success of your installation by running the following command:

```
conda list | grep lightning
```

If you see output similar to the following, your installation was successful:

```
lightning-bolts    0.3.4                pypi_0    pypi
lightning-flash   0.5.0                pypi_0    pypi
pytorch-lightning 1.4.4                pypi_0    pypi
```

Now you are ready to build your first DL model!

To begin building a DL model, complete the following steps:

1. Import the necessary `torch` and `flash` libraries, and import `download_data`, `TextClassificationData`, and `TextClassifier` from the `flash` subpackages:

```
import torch
import flash
from flash.core.data.utils import download_data
from flash.text import TextClassificationData,
TextClassifier
```

2. To get the dataset for fine-tuning, use `download_data` to download the `imdb.zip` file, which is the public domain binary sentiment classification (positive/negative) dataset from **Internet Movie Database (IMDb)** to a local data folder. The IMDb ZIP file contains three CSV files:

- `train.csv`
- `valid.csv`
- `test.csv`

Each file contains two columns: `review` and `sentiment`. We then use `TextClassificationData.from_csv` to declare a `datamodule` variable that assigns the "review" to `input_fields`, and the "sentiment" to `target_fields`. Additionally, it assigns the `train.csv` file to `train_file`, the `valid.csv` file to `val_file`, and the `test.csv` file to the `test_file` properties of `datamodule`, respectively:

```
download_data("https://pl-flash-data.s3.amazonaws.com/
imdb.zip", "./data/")
datamodule = TextClassificationData.from_csv(
    input_fields="review",
    target_fields="sentiment",
    train_file="data/imdb/train.csv",
    val_file="data/imdb/valid.csv",
    test_file="data/imdb/test.csv"
)
```

3. Once we have the data, we can now perform fine-tuning using a foundation model. First, we declare `classifier_model` by calling `TextClassifier` with a backbone assigned to `prajjwall/bert-tiny` (which is a much smaller BERT-like pretrained model located in the Hugging Face model repository: <https://huggingface.co/prajjwall/bert-tiny>). This means our model will be based on the `bert-tiny` model.
4. The next step is to set up the trainer by defining how many epochs we want to run and how many GPUs we want to use to run them. Here, `torch.cuda.device_count()` will return either 0 (no GPU) or 1 to  $N$ , where  $N$  is the maximum number of GPUs you can have in your running environment. Now we are ready to call `trainer.finetune` to train a binary sentiment classifier for the IMDb dataset:

```
classifier_model = TextClassifier(backbone="prajjwall/
bert-tiny", num_classes=datamodule.num_classes)
trainer = flash.Trainer(max_epochs=3, gpus=torch.cuda.
```

```
device_count())
trainer.finetune(classifier_model, datamodule=datamodule,
strategy="freeze")
```

### DL Fine-Tuning Time

Depending on your running environment, the fine-tuning step might take a couple of minutes on a GPU or around 10 minutes (if you're only using a CPU). You can reduce `max_epochs=1` if you simply want to get a basic version of the sentiment classifier quickly.

- Once the fine-tuning step is complete, we will test the accuracy of the model by running `trainer.test()`:

```
trainer.test()
```

The output of the test should look similar to the following screenshot, which indicates that the final model accuracy is about 52%:

```
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
Testing: 99%|██████████| 616/625 [00:05<00:00, 119.48it/s]-----
DATALOADER:0 TEST RESULTS
{'test_accuracy': 0.520799994468689, 'test_cross_entropy': 0.8747543096542358}
-----
Testing: 100%|██████████| 625/625 [00:05<00:00, 112.81it/s]
Out[56]: [{'test_accuracy': 0.520799994468689,
  'test_cross_entropy': 0.8747543096542358}]
```

Figure 1.2 – The test results of our first DL model

The test result shown in the preceding diagram indicates that we have a basic version of the model, as we only fine-tuned the foundation model for three epochs and haven't used any advanced techniques such as hyperparameter tuning or better fine-tuning strategies. However, this is a great accomplishment since you now have a working knowledge of how the core DL model paradigm works! We will explore more advanced model training techniques in later chapters of this book.

## Understanding DL's full life cycle development

By now, you should have your first DL model ready and should feel proud of it. Now, let's explore the full DL life cycle together to fully understand its concepts, components, and challenges.

You might have gathered that the core DL development paradigm revolves around three key artifacts: *Data*, *Model*, and *Code*. In addition to this, *Explainability* is another major artifact that is required in many mission-critical application scenarios such as medical diagnoses, the financial industry, and decision making for criminal justice. As DL is usually considered a black box, providing explainability for DL increasingly becomes a key requirement before and after shipping to production.

Note that *Figure 1.1* is still considered offline experimentation if we are still trying to figure out which model works using a dataset in a lab-like environment. Even in such an offline experimentation environment, things will quickly become complicated. Additionally, we would like to know and track which experiments we have or have not performed so that we don't waste time repeating the same experiments, whatever parameters and datasets we have used, and whatever kind of metrics we have for a specific model. Once we have a model that's good enough for the use cases and customer scenarios, the complexity increases as we need a way to continuously deploy and update the model in production, monitor the model and data drift, and then retrain the model when necessary. This complexity further increases when at-scale training, deployment, monitoring, and explainability are needed.

Let's examine what a DL life cycle looks like (see *Figure 1.3*). There are five stages:

1. Data collection, cleaning, and annotation/labeling.
2. Model development (which is also known as offline experimentation). The core DL development paradigm in *Figure 1.1* is considered part of the *model development* stage, which itself can be an iterative process.
3. Model deployment and serving in production.
4. Model validation and A/B testing (which is also known as online experimentation; this is usually in a production environment).
5. Monitoring and feedback data collection during production.

Figure 1.3 provides a diagram to show that it is a continuous development cycle for a DL model:

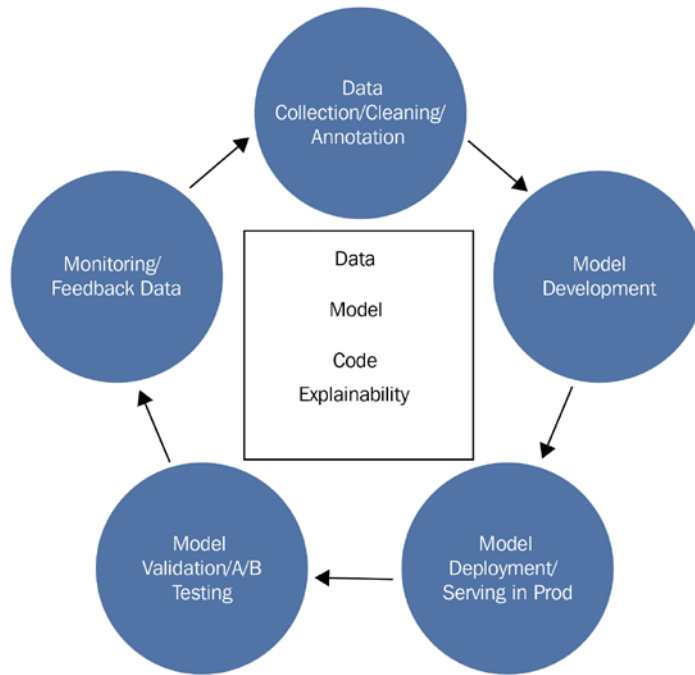


Figure 1.3 – The full DL development life cycle

In addition to this, we want to point out that the backbone of these five stages, as shown in *Figure 1.3*, essentially revolves around the four artifacts: data, model, code, and explainability. We will examine the challenges related to these four artifacts in the life cycle in the following sections. However, first, let's explore and understand MLOps, which is an evolving platform concept and framework that supports the full life cycle of ML. This will help us understand these challenges in a big-picture context.



## Understanding MLOps challenges

MLOps has some connections to DevOps, where a set of technology stacks and standard operational procedures are used for software development and deployment combined with IT operations. Unlike traditional software development, ML and especially DL represent a new era of software development paradigms called **Software 2.0** (<https://karpathy.medium.com/software-2-0-a64152b37c35>). The key differentiator of Software 2.0 is that the behavior of the software does not just depend on well-understood programming language code (which is the characteristic of Software 1.0) but depends on the learned weights in a neural network that's difficult to write as code. In other words, there exists an inseparable integration of the code, data, and model that must be managed together. Therefore, MLOps is being developed and is still evolving to accommodate this new Software 2.0 paradigm. In this book, MLOps is defined as an operational automation platform that consists of three foundation layers and four pillars. They are listed as follows:

- Here are the three foundation layers:
  - Infrastructure management and automation
  - Application life cycle management and **Continuous Integration and Continuous Deployment (CI/CD)**
  - Service system observability
- Here are the four pillars:
  - Data observability and management
  - Model observability and life cycle management
  - Explainability and **Artificial Intelligence (AI)** observability
  - Code reproducibility and observability

Additionally, we will explain MLflow's roles in these MLOps layers and pillars so that we have a clear picture regarding what MLflow can do to build up the MLOps layers in their entirety:

- **Infrastructure management and automation:** This includes, but is not limited to, *Kubernetes* (also known as k8s) for automated container orchestration and *Terraform* (commonly used for managing hundreds of cloud services and access control). These tools are adapted to manage ML and DL applications that have deployed models as service endpoints. These infrastructure layers are not the focus of this book; instead, we will focus on how to deploy a trained DL model using MLflow's provided capabilities.

- **Application life cycle management and CI/CD:** This includes, but is not limited to, *Docker* containers for virtualization, container life cycle management tools such as Kubernetes, and *CircleCI* or *Concourse* for **CI** and **CD**. Usually, CI means that whenever there are code or model changes in a GitHub repository, a series of automatic tests will be triggered to make sure no breaking changes are introduced. Once these tests have been passed, new changes will be automatically released as part of a new package. This will then trigger a new deployment process (CD) to deploy the new package to the production environment (often, this will include human approval as a safety gate). Note that these tools are not unique to ML applications but have been adapted to ML and DL applications, especially when we require GPU and distributed clusters for the training and testing of DL models. In this book, we will not focus on these tools but will mention the integration points or examples when needed.
- **Service system observability:** This is mostly for monitoring the hardware/clusters/CPU/memory/storage, operating system, service availability, latency, and throughput. This includes tools such as *Grafana*, *Datadog*, and more. Again, these are not unique to ML and DL applications and are not the focus of this book.
- **Data observability and management:** This is traditionally under-represented in the DevOps world but becomes very important in MLOps as data is critical within the full life cycle of ML/DL models. This includes *data quality monitoring*, *outlier detection*, *data drift* and *concept drift detection*, *bias detection*, *secured and compliant data sharing*, *data provenance tracking* and *versioning*, and more. The tool stacks in this area that are suitable for ML and DL applications are still emerging. A few examples include **DataFold** (<https://www.datafold.com/>) and **Databand** (<https://databand.ai/open-source/>). A recent development in data management is a unified lakehouse architecture and implementation called **Delta Lake** (<http://delta.io>) that can be used for ML data management. MLflow has native integration points with Delta Lake, and we will cover that integration in this book.
- **Model observability and life cycle management:** This is unique to ML/DL models, and it only became widely available recently due to the rise of MLflow. This includes tools for model training, testing, versioning, registration, deployment, serialization, model drift monitoring, and more. We will learn about the exciting capabilities that MLflow provides in this area. Note that once we combine CI/CD tools with MLflow training/monitoring, user feedback loops, and human annotations, we can achieve **Continuous Training**, **Continuous Testing**, and **Continuous Labeling**. MLflow provides the foundational capabilities so that further automation in MLOps becomes possible, although such complete automation will not be the focus of this book. Interested readers can find relevant references at the end of this chapter to explore this area further.

- Explainability and AI observability:** This is unique to ML/DL models and is especially important for DL models, as traditionally, DL models are treated as black boxes. Understanding why the model provides certain predictions is critical for societally important applications. For example, in medical, financial, juridical, and many human-in-the-loop decision support applications, such as civilian and military emergency response, the demand for explainability is increasingly higher. MLflow provides native integration with a popular explainability framework called SHAP, which we will cover in this book.
- Code reproducibility and observability:** This is not entirely unique to ML/DL applications. However, DL models face some special challenges as the number of DL code frameworks are diverse and the need to reproduce a model is not entirely up to the code alone (we also need data and execution environments such as GPU clusters). In addition to this, notebooks are commonly used in model development and production. How to manage the notebooks along with the model run is important. Usually, GitHub is used to manage the code repository; however, we need to structure the ML project code in a way that's reproducible either locally (such as on a local laptop) or remotely (for example, in a Databricks' GPU cluster). MLflow provides this capability to allow DL projects that have been written once to run anywhere, whether this is in an offline experimentation environment or an online production environment. We will cover MLflow's MLproject capability in this book.

In summary, MLflow plays a critical and foundational role in MLOps. It fills in the gaps that DevOps traditionally does not cover and, thus, is the focus of this book. The following diagram (*Figure 1.4*) shows the central roles of MLflow in the still-evolving MLOps world:

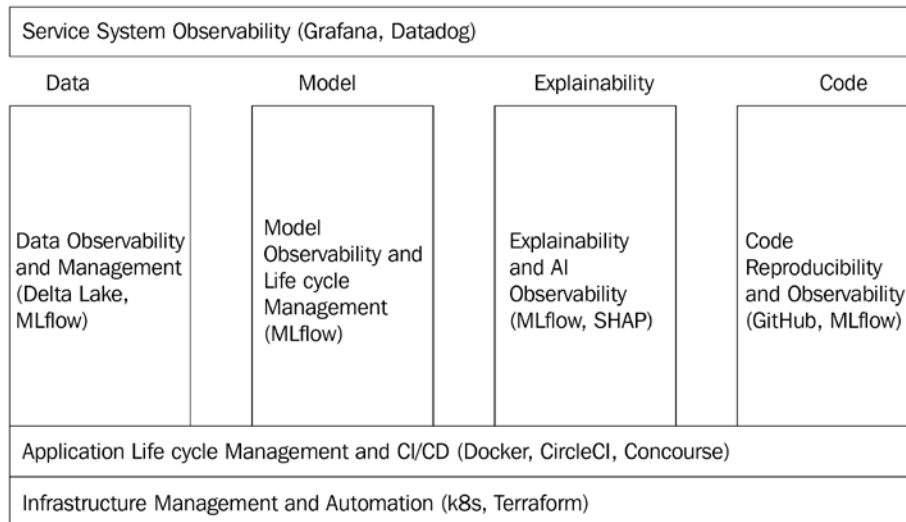


Figure 1.4 – The three layers and four pillars of MLOps and MLflow's roles

While the bottom two layers and the topmost layer are common within many software development and deployment processes, the middle four pillars are either entirely unique to ML/DL applications or partially unique to ML/DL applications. MLflow plays a critical role in all four of these pillars in MLOps. This book will help you to confidently apply MLflow to solve the issues of these four pillars while also equipping you to further integrate with other tools in the MLOps layers depicted in *Figure 1.4* for full automation depending on your scenario requirements.

## Understanding DL data challenges

In this section, we will discuss the data challenges at each stage of the DL life cycle, as illustrated in *Figure 1.3*. Essentially, DL is a data-centric AI, unlike symbolic AI where human knowledge can be used without lots of data. The challenges for data in DL are pervasive in all stages of the full life cycle:

- **Data collection/cleaning/annotation:** One of DL's first successes began with **ImageNet** (<https://www.image-net.org/>), where millions of images are collected and annotated according to the English nouns in the WordNet database (<https://wordnet.princeton.edu/>). This led to the successful development of pretrained DL models for computer vision such as VGG-NETS ([https://pytorch.org/hub/pytorch\\_vision\\_vgg/](https://pytorch.org/hub/pytorch_vision_vgg/)), which can perform state-of-the-art image classification and is widely used for industrial and business applications. The main challenge of this kind of large-scale data collection and annotation is the unknown bias, which is hard to measure in this process (<https://venturebeat.com/2020/11/03/researchers-show-that-computer-vision-algorithms-pretrained-on-imagenet-exhibit-multiple-distressing-biases/>). Another example is the sales engagement platform **Outreach** (<https://www.outreach.io/>), where we can classify a potential buyer's sentiment. For instance, we might start by collecting email messages of 100 paid organizations to train a DL model. Following this, we would need to collect email messages from more organizations, either due to an accuracy requirement or expanded language coverage (such as from English only to other languages such as German and French). These many iterations of data collection and annotation will generate quite a lot of datasets. There is a tendency to just name the version of the dataset with hardcoded version numbers as part of a dataset filename such as the following:

```
MyCoolAnnotatedData-v1.0.csv
MyCoolAnnotatedData-v2.0.csv
MyCoolAnnotatedData-v3.0.csv
MyCoolAnnotatedData-v4.0.csv
```

This seems to work until some changes are required in any one of the vX.0 datasets due to the need to correct annotation errors or remove email messages because of customer churn. Also, what happens if we need to combine several datasets together or perform some data cleaning and transformation to train a new DL model? What if we need to implement data augmentation to artificially generate some datasets? Evidently, simply changing the names of the files is neither scalable nor sustainable.

- **Model development:** We need to understand that the bias in the data we use to train/pretrain a DL model will reflect in the prediction when applying the model. While we do not focus on de-biasing data in this book, we must implement data versioning and data provenance as first-class artifacts when training and serving a DL model so that we can track all model experiments. When fine-tuning a pretrained model for our use cases, as we did earlier, we also need to track the versioning of the fine-tuning dataset we use. In our previous example, we use a variant of the BERT model to fine-tune the IMDb review data. While, in our first example, we did not care about the versioning or source of the data, this is important for a practical and real application. In summary, DL models need to link to a particular version of datasets using a scalable approach. We will provide solutions to this topic in this book.
- **Model deployment and serving in production:** This is for deploying into the production environment to serve online traffic. DL model serving latency is of particular importance and is interesting to collect at this stage. This might allow you to adjust the hardware environment used for inference.
- **Model validation and A/B testing:** The data we collect at this stage is mostly for user behavior metrics in the online experimentation environment (<https://www.slideshare.net/pavel/ab-testing-ai-global-artificial-intelligence-conference-2019>). Online data traffic also needs to be characterized in order to understand whether there is a statistical difference in the input to the model between offline experimentation and online experimentation. Only if we pass the A/B testing and validate that the model indeed works better than its previous version in terms of user behavior metrics do we roll out to production for all users.
- **Monitoring and feedback loops:** In this stage, note that the data will need to be continuously collected to detect data drift and concept drift. For example, in the buyer sentiment classification example discussed earlier, if buyers start to use terminology that is not encountered in the training data, the performance of the model could suffer.

In summary, data tracking and observability are major challenges in all stages of the DL life cycle.

## Understanding DL model challenges

In this section, we will discuss DL model challenges. Let's look at the challenges at each stage of the DL life cycle, as depicted in *Figure 1.3*:

- **Data collection/cleaning/annotation:** While the data challenge has already been stated, the challenge of linking data to the model of interest still exists. MLflow has native integration with Delta Lake so that any trained model can be traced back to a particular version within Delta Lake.
- **Model development:** This is the time for trying lots of model frameworks, packages, and model selections. We need to track all the packages we use, along with the model parameters, hyperparameters, and model metrics in all experiments we run. Without a scalable and standardized way to track all experiments, this becomes a very tangled space. This not only causes trouble in terms of not knowing which experiments have been done so that we don't waste time doing them again, but it also creates problems when tracking which model is ready to be deployed or has already been deployed. Model serialization is another major challenge as different DL frameworks tend to use different ways to serialize the model. For example, `pickle` (<https://github.com/cloudpipe/cloudpickle>) is usually used in serializing the model written in Python. However, `TorchScript` (<https://pytorch.org/docs/stable/jit.html>) is now highly performant for PyTorch models. In addition, Open Neural Network Exchange or ONNX (<https://onnx.ai/>) tries to provide more framework-agnostic DL serialization. Finally, we need to log the serialized model and register the model so that we can track model versioning. MLflow is one of the first open source tools to overcome these challenges.
- **Model deployment and serving in production:** An easy-to-use model deployment tool that can tie into the model registry is a challenge. MLflow can be used to alleviate that, allowing you to load models for production deployment with full provenance tracking.
- **Model validation and A/B testing:** During online validation and experimentation, model performance needs to be validated and user behavior metrics need to be collected. This is so that we can easily roll back or redeploy a particular version of the models. A model registry is critical for at-scale online model production validation and experimentation.
- **Monitoring and feedback loops:** Model drifting and degradation over time is a real challenge. The visibility of model performance in production needs to be continuously monitored. Feedback data can be used to decide whether a model needs to be retrained.

In summary, DL model challenges in the full life cycle are unique. It is also worth pointing out a common framework that can assist the model development and online production back-and-forth is of great importance, as we don't want to use different tools just because the execution environment is different. MLflow provides this unified framework to bridge such gaps.

## Understanding DL code challenges

In this section, we will discuss DL code challenges. Let's look at how these code challenges are manifested in each of the stages described in *Figure 1.3*. In this section, and within the context of DL development, code refers to the source code that's written in certain programming languages such as Python for data processing and implementation, while a model refers to the model logic and architecture in its serialized format (for example, pickle, TorchScript, or ONNX):

- **Data collection/cleaning/annotation:** While data is the central piece in this stage, the code that does the query, **extraction/transformation/loading (ETL)**, and data cleaning and augmentation is of critical importance. We cannot decouple the development of the model from the data pipelines that provide the data feeds to the model. Therefore, data pipelines that implement ETL need to be treated as one of the integrated steps in both offline experimentation and online production. A common mistake is that we use different data ETL and cleaning pipelines in offline experimentation, and then implement different data ETL/cleaning pipelines in online production, which could cause different model behaviors. We need to version and serialize the data pipeline as part of the entire model pipeline. MLflow provides several ways to allow us to implement such multistep pipelines.
- **Model development:** During offline experiments, in addition to different versions of data pipeline code, we might also have different versions of notebooks or use different versions of DL library code. The usage of notebooks is particularly unique in ML/DL life cycles. Tracking which model results are produced by which notebook/model pipeline/data pipeline needs to be done for each run. MLflow does that with automatic code version tracking and dependencies. In addition, code reproducibility in different running environments is unique to DL models, as DL models usually require hardware accelerators such as GPUs or TPUs. The flexibility of running either locally, or remotely, on a CPU or GPU environment is of great importance. MLflow provides a lightweight approach in which to organize the ML projects so that code can be written once and run everywhere.

- **Model deployment and serving in production:** While the model is serving production traffic, any bugs will need to be traced back to both the model and code. Thus, tracking code provenance is critical. It is also critical to track all the dependency code library versions for a particular version of the model.
- **Model validation and A/B testing:** Online experiments could use multiple versions of models using different data feeds. Debugging any experimentation will require not only knowing which model is used but also which code is used to produce that model.
- **Monitoring and feedback loops:** This stage is similar to the previous stage in terms of code challenges, where we need to know whether model degradation is due to code bugs or model and data drifting. The monitoring pipeline needs to collect all the metrics for both data and model performance.

In summary, DL code challenges are especially unique because DL frameworks are still evolving (for example, **TensorFlow**, **PyTorch**, **Keras**, **Hugging Face**, and **SparkNLP**). MLflow provides a lightweight framework to overcome many common challenges and can interface with many DL frameworks seamlessly.

## Understanding DL explainability challenges

In this section, we will discuss DL explainability challenges at each of the stages described in *Figure 1.3*. It is increasingly important to view explainability as an integral and necessary mechanism to define, test, debug, validate, and monitor models across the entire model life cycle. Embedding explainability early will make subsequent model validation and operations easier. Also, to maintain ongoing trust in ML/DL models, it is critical to be able to explain and debug ML/DL models after they go live in production:

- **Data collection/cleaning/annotation:** As we have gathered, explainability is critical for model prediction. The root cause of any model's trustworthiness or bias can be traced back to the data used to train the model. Explainability for the data is still an emerging area but is critical. So, what could go wrong and become a challenge during the data collection/cleaning/annotation stage? For example, let's suppose we have an ML/DL model, and its prediction outcome is about whether a loan applicant will pay back a loan or not. If the data collected has certain correlations between age and the loan payback outcome, this will cause the model to use age as a predictor. However, a loan decision based on a person's age is against the law and not allowed even if the model works well. So, during data collection, it could be that the sampling strategy is not sufficient to represent certain subpopulations such as different loan applicants in different age groups.



A subpopulation could have lots of missing fields and then be dropped during data cleaning. This could result in underrepresentation following the data cleaning process. Human annotations could favor the privileged group and other possible unconscious biases. A metric called **Disparate Impact** could reveal the hidden biases in the data, which compares the proportion of individuals that receive a positive outcome for two groups: an unprivileged group and a privileged group. If the unprivileged group (for example, persons with age > 60) receives a positive outcome (for example, loan approval) less than 80% of the proportion of the privileged group (persons with age < 60), this is a disparate impact violation based on the current common industry standard (a four-fifths rule). Tools such as **Dataiku** could help to automate the disparate impact and subpopulation analysis to find groups of people who may be treated unfairly or differently because of the data used for model training.

- **Model development:** Model explainability during offline experimentation is very important to not only help understand why a model behaves a certain way but also help with model selection to decide which model to use if we need to put it into production. Accuracy might not be the only criteria to select a winning model. There are a few DL explainability tools, such as SHAP (please refer to *Figure 1.5*). MLflow integration with SHAP provides a way to implement DL explainability:

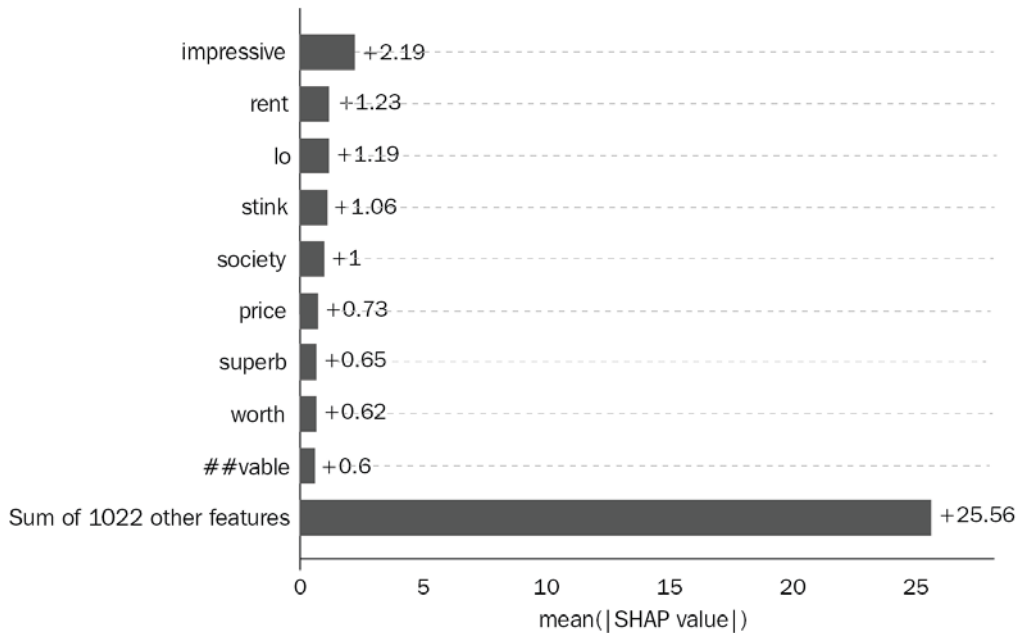


Figure 1.5 – NLP text SHAP Variable Importance Plot when using a DL model

Figure 1.5 shows that this NLP model's prediction results' number one feature is the word *impressive*, followed by *rent*. Essentially, this breaks the black box of the DL model, giving much confidence to the usage of DL models in production.

- **Model deployment and serving in production:** During the production stage, if the explainability of the model prediction can be readily provided to users, then not only will the usability (user-friendliness) of the model be improved, but also, we can collect better feedback data as users are more incentivized to give more meaningful feedback. A good explainability solution should provide point-level decisions for any prediction outcome. This means that we should be able to answer why a particular person's loan is rejected and how this rejection compares to other people in a similar or different age group. So, the challenge is to have explainability as one of the gated deployment criteria for releasing a new version of the model. However, unlike accuracy metrics, it is very difficult to measure explainability as scores or thresholds, although certain case-based reasoning could be applied and automated. For example, if we have certain hold-out test cases where we expect the same or similar explanations regardless of the versions of the model, then we could use that as a gated release criterion.
- **Model validation and A/B testing:** During online experimentation and ongoing production model validation, we would need explainability to understand whether the model has been applied to the right data or whether the prediction is trustworthy. Usually, ML/DL models encode complex and non-linear relationships. During this stage, it is often desirable to understand how the model influences the metrics of user behavior (for example, a higher conversion rate on a shopping website). Influence sensitivity analysis could provide insights regarding whether a certain user feature such as a user's income has a positive or negative impact on the outcome. If during this stage, we found, for some reason, that higher incomes cause a negative loan approval rate or a lower conversion rate, then this should be automatically flagged. However, automated sensitivity analysis during model validation and A/B testing is still not widely available and remains a challenging problem. A few vendors such as TruEra provide potential solutions to this space.

- **Monitoring and feedback loops:** While model performance metrics and data characteristics are of importance here, explainability can provide an incentive for users to provide valuable feedback and user behavior metrics to identify drivers and causes of model degradation if there are any. As we know, ML/DL models are prone to overfitting and cannot generalize well beyond their training data. One important explainability solution during model production monitoring is to measure how feature importance shifts across different data splits (for example, pre-COVID versus post-COVID). This can help data scientists to identify where degradation in model performance is due to changing data (such as a statistical distribution shift) or changing relationships between variables (such as a concept shift). A recent example provided by TruEra (<https://truera.com/machine-learning-explainability-is-just-the-beginning/>) illustrates that a loan model changes its prediction behavior due to changes in people's annual income and loan purposes before and after the COVID periods. This explainability of **Feature Importance Shift** greatly helps to identify the root causes of changes in model behavior during the model production monitoring stage.

In summary, DL explainability is a major challenge where ongoing research is still needed. However, MLflow's integration with SHAP now provides a ready-to-use tool for practical DL applications, which we will cover in our advanced chapter later in this book.

## Summary

In this opening chapter, we implemented our first DL model by following the pretrain plus fine-tuning core DL development paradigm using PyTorch `lightning-flash` for a text sentiment classification model. We learned about the five stages of the full life cycle of DL. We defined the concept of MLOps along with the three foundation layers and four ML/DL pillars, where MLflow plays critical roles in all four pillars (data, model, code, and explainability). Finally, we described the challenges in DL data, model, code, and explainability.

With the knowledge and first DL model experience gained in this chapter, we are now ready to learn about and implement MLflow in our DL model in the following chapters. In the next chapter, we will start with the implementation of a DL model with MLflow autologging enabled.

---

## Further reading

To further your knowledge, please consult the following resources and documentation:

- *On the Opportunities and Risks of Foundation Models* (Stanford University): <https://arxiv.org/abs/2108.07258>
- *MLOps: not as Boring as it Sounds*: <https://itnext.io/mlops-not-as-boring-as-it-sounds-eaebe73e3533>
- *AI is Driving Software 2.0... with Minimal Human Intervention*: <https://www.datasciencecentral.com/profiles/blogs/ai-is-driving-software-2-0-with-minimal-human-intervention>
- *MLOps: Continuous delivery and automation pipelines in machine learning* (Google): <https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning>
- *Deep Learning Development Cycle* (Salesforce): <https://metamind.readme.io/docs/deep-learning-dev-cycle>
- *MLOps – The Missing Piece In The Enterprise AI Puzzle*: <https://www.forbes.com/sites/janakirammsv/2021/01/05/mlopsthe-missing-piece-in-the-enterprise-ai-puzzle/?sh=3d5c89dd24ad>
- *MLOps: What It Is, Why It Matters, and How to Implement It*: <https://neptune.ai/blog/mlops>
- *Explainable Deep Learning: A Field Guide for the Uninitiated*: <https://arxiv.org/abs/2004.14545>
- *Machine learning explainability is just the beginning*: <https://truera.com/machine-learning-explainability-is-just-the-beginning/>
- *AI Fairness — Explanation of Disparate Impact Remover*: <https://towardsdatascience.com/ai-fairness-explanation-of-disparate-impact-remover-ce0da59451f1>
- *Datasheets for Datasets*: <https://arxiv.org/pdf/1803.09010.pdf>



# 2

# Getting Started with MLflow for Deep Learning

One of the key capabilities of MLflow is to enable **Machine Learning (ML)** experiment management. This is critical because data science requires reproducibility and traceability so that a **Deep Learning (DL)** model can be easily reproduced with the same data, code, and execution environment. This chapter will help us get started with how to implement DL experiment management quickly. We will learn about MLflow experiment management concepts and capabilities, set up an MLflow development environment, and complete our first DL experiment using MLflow. By the end of this chapter, we will have a working MLflow tracking server showing our first DL experiment results.

In this chapter, we're going to cover the following main topics:

- Setting up MLflow
- Implementing our first MLflow logging-enabled DL experiment
- Exploring MLflow's components and usage patterns

## Technical requirements

To complete the experiment in this chapter, we will need the following tools, libraries, and GitHub repositories installed or checked out on our computer:

- VS Code: The version we use in this book is August 2021 (that is, version 1.60). We use VS Code for our local code development environment. This is the recommended way for local developments. Please refer to [https://code.visualstudio.com/updates/v1\\_60](https://code.visualstudio.com/updates/v1_60).
- MLflow: Version 1.20.2. In this chapter, in the *Setting up MLflow* section, we will walk through how to set up MLflow locally or remotely. Please refer to <https://github.com/mlflow/mlflow/releases/tag/v1.20.2>.
- Miniconda: Version 4.10.3. Please refer to <https://docs.conda.io/en/latest/miniconda.html>.
- PyTorch lightning-flash: 0.5.0. Please refer to <https://github.com/PyTorchLightning/lightning-flash/releases/tag/0.5.0>.
- The GitHub URL for the code in this chapter: You can find this at <https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLflow/tree/main/chapter02>.

## Setting up MLflow

**MLflow** is an open source tool that is primarily written in Python. It has over 10,000 stars tagged in its GitHub source repository (<https://github.com/mlflow/mlflow>). The benefits of using MLflow are numerous, but we can illustrate one benefit with the following scenario: Let's say you are starting a new ML project, trying to evaluate different algorithms and model parameters. Within a few days, you run hundreds of experiments with lots of code changes using different ML/DL libraries and get different models with different parameters and accuracies. You need to compare which model works better and also allow your team members to reproduce the results for model review purposes. Do you prepare a spreadsheet and write down the model name, parameters, accuracies, and location of the models? How can someone else rerun your code or use your trained model with a different set of evaluation datasets? This can quickly become unmanageable when you have lots of iterations for different projects. MLflow can help you to track your experiments, compare your model runs and allow others to reproduce your results easily, reuse your trained models for review purposes, and even deploy your model to production with ease.

Sound exciting? Well, let's set up MLflow so that we can explore its components and patterns. MLflow allows both a local setup and a cloud-based setup. We will walk through both of these setup scenarios in the following sections.

## Setting up MLflow locally using miniconda

First, let's set up MLflow in a local development environment. This allows quick prototyping and helps you to get familiar with the basic functionality of the MLflow tool. Additionally, it allows you to interact with a remote MLflow cloud server when required. Follow these instructions to set up MLflow.

Assuming you already have a virtual conda environment created from *Chapter 1, Deep Learning Life Cycle and MLOps Challenges*, you are ready to install MLflow in the same virtual environment:

```
pip install mlflow
```

The preceding command will install the latest version of MLflow. If you want to install a specific version of MLflow, you can use the following:

```
pip install mlflow==1.20.2
```

As you can see, I have installed MLflow version 1.20.2. By default, MLflow will use the local filesystem to store all of the experiment artifacts (for example, a serialized model) and metadata (parameters, metrics, and more). If a relational database is needed as MLflow's backend storage, additional installation and configuration are required. For now, let's use the filesystem for storage. You can verify your MLflow installation locally by typing the following into the command line:

```
mlflow --version
```

Then, it will show the installed MLflow version, as follows:

```
mlflow, version 1.20.2
```

This confirms that we have installed version 1.20.2 of MLflow on our local development environment. Additionally, you can launch the MLflow UI locally to see the MLflow tracking server UI, as follows:

```
mlflow ui
```



Following this, you will see that the UI web server is running:

```

± lmain ? :3 x1 → mlflow ui
[2021-09-05 19:20:52 -0700] [62537] [INFO] Starting gunicorn 20.1.0
[2021-09-05 19:20:52 -0700] [62537] [INFO] Listening at: http://127.0.0.1:5000 (62537)
[2021-09-05 19:20:52 -0700] [62537] [INFO] Using worker: sync
[2021-09-05 19:20:52 -0700] [62539] [INFO] Booting worker with pid: 62539

```

Figure 2.1 – Starting the MLflow UI in a local environment

Figure 2.1 shows the local MLflow UI website: <http://127.0.0.1:5000/>. If you click on this URL, you will see the following MLflow UI showing up in your browser window. Since this is a brand new MLflow installation, there is only one **Default** experiment with no runs under it yet (please refer to Figure 2.2):

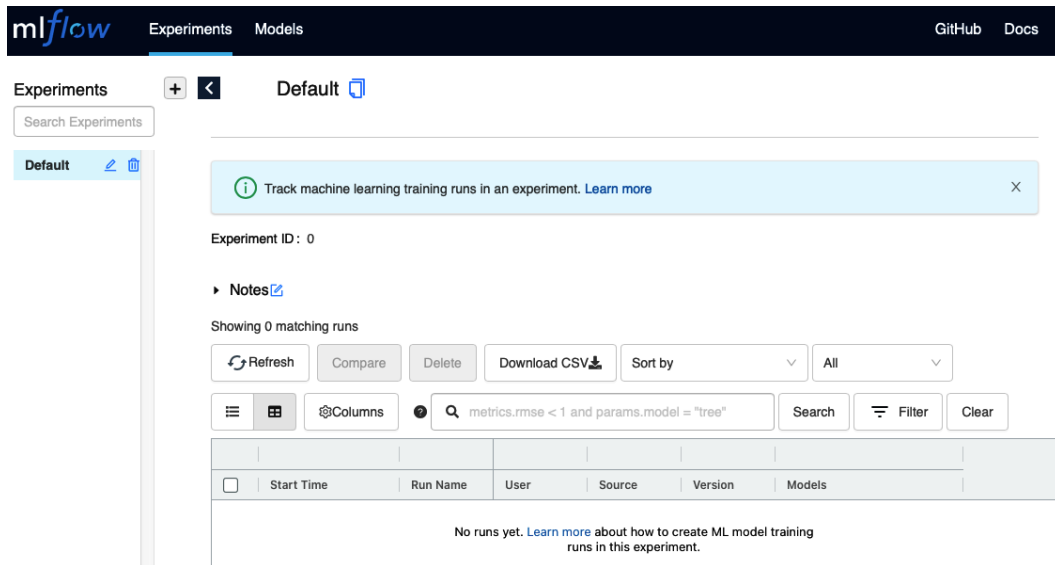


Figure 2.2 – The MLflow Default Experiments UI web page

Seeing the default MLflow UI page up and running concludes the successful setup of MLflow locally with a local working MLflow tracking server.

## Setting up MLflow to interact with a remote MLflow server

In a corporate production environment, MLflow is usually hosted on a cloud server, which could be self-hosted or one of the Databricks' managed services in one of the cloud providers (such as AWS, Azure, or Google Cloud). In those cases, there is a requirement to set up your local development environment so that you can run your ML/DL experiment locally but interact with the MLflow server remotely. Next, we will describe how to do this using environment variables with the help of the following three steps:

1. In a bash shell command-line environment, define three new environment variables if you are using a Databricks-managed MLflow tracking server. The first environment variable is `MLFLOW_TRACKING_URI`, and the assigned value is `databricks`:

```
export MLFLOW_TRACKING_URI=databricks
export DATABRICKS_HOST=https://*****
export DATABRICKS_TOKEN=dapi*****
```

2. The second environment variable is `DATABRICKS_HOST`. If your Databricks managed website looks like `https://dbc-*.cloud.databricks.com/`, then that's the value of the `DATABRICKS_HOST` variable (replace `*` with your actual website string).
3. The third environment variable is `DATABRICKS_TOKEN`. Navigate to your Databricks-managed website at `https://dbc-*.cloud.databricks.com/#setting/account`, click on **Access Tokens**, and then click on **Generate New Token**. You will see a pop-up window with a **Comment** field (which can be used to record why this token will be used) and expiration date, as shown in *Figure 2.3*:

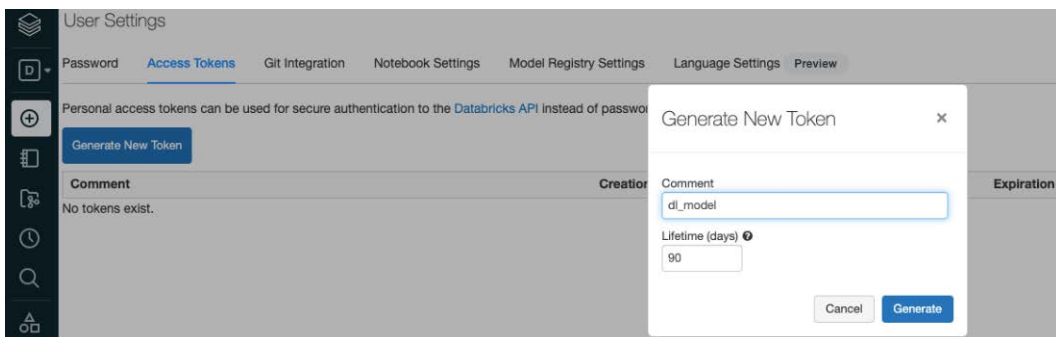


Figure 2.3 – Generating a Databricks access token

Click on the **Generate** button and a pop-up window similar to *Figure 2.4* will appear. It will ask you to copy that token. This token will need to be copied and assigned to the `DATABRICKS_TOKEN` environment variable as the value:

## Generate New Token

Your token has been created successfully.

```
dapid5011ac928a9652e4fe03201fe50ca33
```

⚠ Make sure to copy the token now. You won't be able to see it again.

Done

Figure 2.4 – Copying the generated token that will be used for the environment variable

Once you have these three environment variables set up, you will be able to interact with the Databricks-managed MLflow server in the future. Note that the access token has an expiration date for security reasons, which can be revoked at any time by the administrator, so make sure you have the environment variable updated accordingly when the token is refreshed.

In summary, we have learned how to set up MLflow locally to interact with a local MLflow tracking server or a remote MLflow tracking server. This will allow us to implement our first MLflow tracking-enabled DL model in the next section so that we can explore MLflow concepts and components in a hands-on way.

## Implementing our first DL experiment with MLflow autologging

Let's use the DL sentiment classifier we built in *Chapter 1, Deep Learning Life Cycle and MLOps Challenges*, and add MLflow autologging to it to explore MLflow's tracking capabilities:

1. First, we need to import the MLflow module:

```
import mlflow
```

This will provide MLflow **Application Programming Interfaces (APIs)** for logging and loading models.

2. Just before we run the training code, we need to set up an active experiment using `mlflow.set_experiment` for the current running code:

```
EXPERIMENT_NAME = "dl_model_chapter02"
mlflow.set_experiment(EXPERIMENT_NAME)
experiment = mlflow.get_experiment_by_name(EXPERIMENT_NAME)
print("experiment_id:", experiment.experiment_id)
```

This sets an experiment named `dl_model_chapter02` to be the current active experiment. If this experiment does not exist in your current tracking server, it will be created automatically.

#### Environment Variable

Note that you might need to set the tracking server URI using the `MLFLOW_TRACKING_URI` environment variable before you run your first experiment. If you are using a hosted Databricks server, implement the following:

```
export MLFLOW_TRACKING_URI=databricks
```

If you are using a local server, then set this environment variable to empty or the default localhost at port number 5000 as follows (note that this is our current section's scenario and assumes you are using a local server):

```
export MLFLOW_TRACKING_URI= http://127.0.0.1:5000
```

3. Next, add one line of code to enable autologging in MLflow:

```
mlflow.pytorch.autolog()
```

This will allow the default parameters, metrics, and model to be automatically logged to the MLflow tracking server.

#### Autologging in MLflow

Autologging in MLflow is still in experiment mode (as of version 1.20.2) and might change in the future. Here, we use it to explore the MLflow components since it only requires one line of code to automatically log everything of interest. In the upcoming chapters, we will learn about and implement additional ways to perform tracking and logging in MLflow. Also, note that currently, autologging in MLflow for PyTorch (as of version 1.20.2) only works for the PyTorch Lightning framework, not any arbitrary PyTorch code.

- Use the Python context manager with statement to start the experiment run by calling `mlflow.start_run`:

```
with mlflow.start_run(experiment_id=experiment.
    experiment_id, run_name="chapter02"):
    trainer.finetune(classifier_model,
                    datamodule=datamodule,
                    strategy="freeze")
    trainer.test()
```

Notice that all lines of code underneath the `with` block are the regular DL model fine-tuning and testing steps. We only enable automatic MLflow logging so that we can observe the metadata that is being tracked/logged by the MLflow tracking server.

- Next, you can run the entire code of `first_dl_with_mlflow.py` (the full code can be viewed in this chapter's GitHub at [https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter02/first\\_dl\\_with\\_mlflow.py](https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter02/first_dl_with_mlflow.py)) using the following command line:

```
python first_dl_with_mlflow.py
```

On a non-GPU macOS laptop, the entire run takes less than 10 minutes. You should have an output on your screen, as follows:

```
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
experiment_id: 1

| Name          | Type                               | Params
-----|-----|-----
0 | train_metrics | ModuleDict                         | 0
1 | val_metrics   | ModuleDict                         | 0
2 | model         | BertForSequenceClassification     | 4.4 M
-----|-----|-----
258      Trainable params
4.4 M    Non-trainable params
4.4 M    Total params
17.545   Total estimated model params size (MB)
Epoch 2: 100%|██████████| 6250/6250 [02:15<00:00, 45.97it/s, loss=0.851, v_
Testing: 99%|██████████| 621/625 [00:09<00:00, 61.16it/s]-----
-
DATALOADER:0 TEST RESULTS
{'test_accuracy': 0.6236000061035156, 'test_cross_entropy': 0.6444308757781982}
-----|-----
Testing: 100%|██████████| 625/625 [00:09<00:00, 63.51it/s]
```

Figure 2.5 – DL model training/testing with MLflow autologging enabled

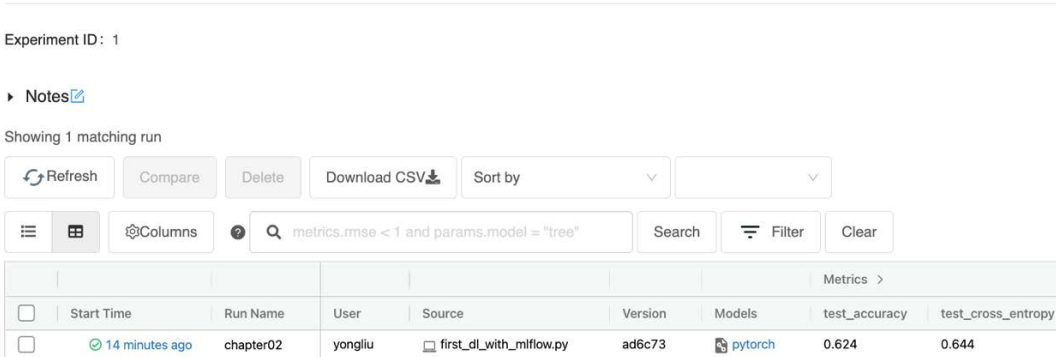
If you are running this for the first time, you will see that the experiment with the name of `dl_model_chapter02` does not exist. Instead, MLflow automatically creates this experiment for you:

```
INFO: 'dl_model_chapter02' does not exist. Creating a new experiment
experiment_id: 1
```

Figure 2.6 – MLflow automatically creates a new environment if it does not exist

- Now, we can open the MLflow UI locally to see what has been logged in the local tracking server by navigating to `http://127.0.0.1:5000/`. Here, you will see that a new experiment (`dl_model_chapter02`) with a new run (**Run Name** = `chapter02`) has been logged:

`dl_model_chapter02` 

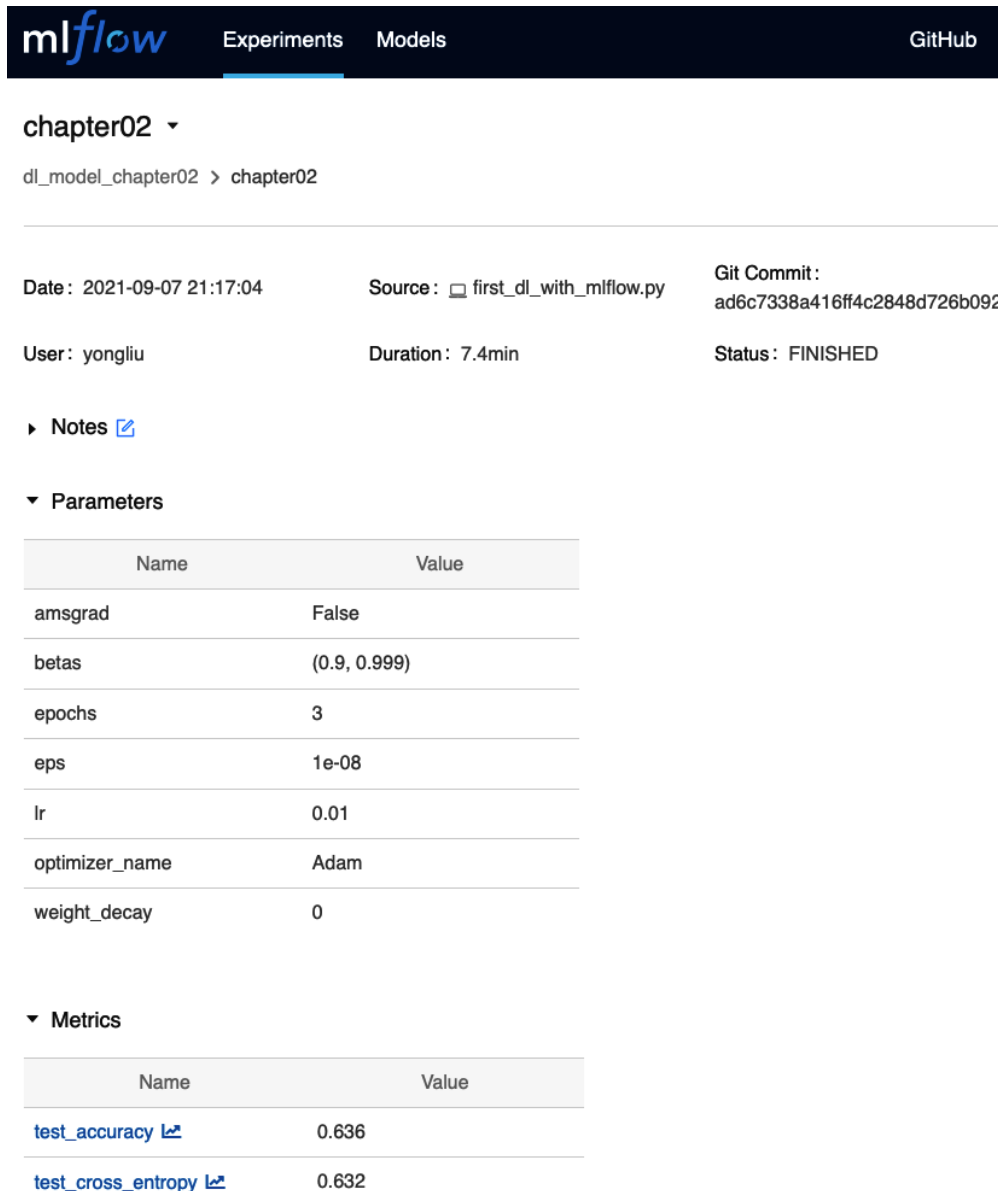


The screenshot shows the MLflow tracking server UI. At the top, it displays "Experiment ID: 1". Below this, there is a "Notes" section. The main area shows "Showing 1 matching run". There are several action buttons: "Refresh", "Compare", "Delete", "Download CSV", and "Sort by". A search bar contains the query "metrics.rmse < 1 and params.model = 'tree'". Below the search bar is a table with the following data:

	Start Time	Run Name	User	Source	Version	Models	test_accuracy	test_cross_entropy
<input type="checkbox"/>	14 minutes ago	chapter02	yongliu	first_dl_with_mlflow.py	ad6c73	pytorch	0.624	0.644

Figure 2.7 – The MLflow tracking server UI shows a new experiment with a new run

Now, click on the hyperlink of the **Start Time** column in *Figure 2.7*. You will see the details of the logged metadata of the run:



The screenshot shows the MLflow interface for an experiment run. At the top, there is a navigation bar with the MLflow logo, 'Experiments' (selected), 'Models', and 'GitHub'. Below the navigation bar, the experiment name 'chapter02' is displayed with a dropdown arrow. The breadcrumb path is 'dl\_model\_chapter02 > chapter02'. The main content area displays metadata details for the run, organized into sections: 'Date', 'Source', 'Git Commit', 'User', 'Duration', and 'Status'. Below these are sections for 'Notes', 'Parameters', and 'Metrics', each with a table of data.

**mlflow** Experiments Models GitHub

chapter02 ▾

dl\_model\_chapter02 > chapter02

---

Date: 2021-09-07 21:17:04      Source: [first\\_dl\\_with\\_mlflow.py](#)      Git Commit: ad6c7338a416ff4c2848d726b092

User: yongliu      Duration: 7.4min      Status: FINISHED

▶ Notes [🔗](#)

▾ Parameters

Name	Value
amsgrad	False
betas	(0.9, 0.999)
epochs	3
eps	1e-08
lr	0.01
optimizer_name	Adam
weight_decay	0

▾ Metrics

Name	Value
<a href="#">test_accuracy</a> <a href="#">🔗</a>	0.636
<a href="#">test_cross_entropy</a> <a href="#">🔗</a>	0.632

Figure 2.8 – The MLflow run UI shows the metadata details about the experiment run

If you can view this screen in your own local environment, then congratulations! You just completed the implementation of MLflow tracking for our first DL model! In the next section, we will explore central concepts and components in MLflow using our working example.

## Exploring MLflow's components and usage patterns

Let's use the working example implemented in the previous section to explore the following central concepts, components, and usages in MLflow. These include experiments, runs, metadata about experiments, artifacts for experiments, models, and code.

### Exploring experiments and runs in MLflow

**Experiment** is a first-class entity in the MLflow APIs. This makes sense as data scientists and ML engineers need to run lots of experiments in order to build a working model that meets the requirements. However, the idea of an experiment goes beyond just the model development stage and extends to the entire life cycle of the ML/DL development and deployment. So, this means that when we do retraining or training for a production version of the model, we need to treat them as *production-quality* experiments. This unified view of experiments builds a bridge between the offline and online production environments. Each experiment consists of many runs where you can either change the model parameters, input data, or even model type for each run. So, an experiment is an umbrella entity containing a series of runs. The following diagram (*Figure 2.9*) illustrates that a data scientist could carry out both offline experiments and online production experiments across multiple stages of the life cycle of ML/DL models:

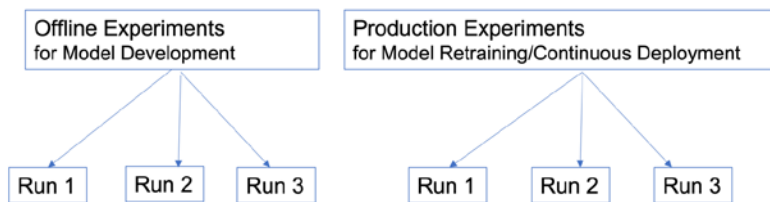


Figure 2.9 – Experiments across the offline and online production life cycles of ML/DL models



As you can see from *Figure 2.9*, during the model development stage, a data scientist could run multiple runs of the same experiment or multiple experiments depending on the project scenarios. If it is a small ML project, having all runs under one single offline experiment could be enough. If it is a complex ML project, it is reasonable to design different experiments and conduct runs under each experiment. A good reference for designing ML experiments can be found at <https://machinelearningmastery.com/controlled-experiments-in-machine-learning/>. Then, during the model production phase, it is desirable to set up production-quality experiments, as we need to perform model improvement and continuous deployment with model retraining. A production experiment will provide a gated accuracy check to prevent regression of the new model. Often, this is achieved by running automatic model evaluation and validation against a hold-out test dataset to check whether a new model still meets the release bar in terms of accuracy.

Now, let's explore the MLflow experiments in a hands-on way. Run the following MLflow command line to interact with the tracking server:

```
mlflow experiments list
```

If your `MLFLOW_TRACKING_URI` environment variable points to a remote tracking server, then it will list all the experiments that you have read access to. If you want to see what's in the local tracking server, you could set `MLFLOW_TRACKING_URI` to nothing (that is, empty), as follows (note that you don't need to do this if you have never had this environment variable in your local user profile; however, doing this will make sure you use a local tracking server):

```
export MLFLOW_TRACKING_URI=
```

Prior to your first implementation of the DL model with MLflow autologging enabled, the output of listing all your experiments should look similar to *Figure 2.10* (note that this also depends on where you run the command line; the following output assumes you run the command in your local folder where you can check the code for *Chapter 2* on GitHub):



```

main U:1 ?:10 x | → mlflow experiments list
-----
Experiment Id  Name           Artifact Location
-----
0             Default       file:///Users/yongliu/opensource_code/Practical-Deep-Learning-at-Scale-with-MLFlow/chapter02/mlruns/0

```

Figure 2.10 – The default MLflow experiment list in a local environment

*Figure 2.10* lists the three columns of the experiment property: **Experiment Id** (an integer), **Name** (a text field that can be used to describe the experiment name), and **Artifact Location** (by default, this is located in the `mlruns` folder underneath the directory where you execute the MLflow commands). The `mlruns` folder is used by a filesystem-based MLflow tracking server to store all the metadata of experiment runs and artifacts.

### The Command-Line Interface (CLI) versus REST APIs versus Programming Language-Specific APIs

MLflow provides three different types of tools and APIs to interact with the tracking server. Here, the CLI is used so that we can explore the MLflow components.

So, let's explore a specific MLflow experiment, as follows:

1. First, create a new experiment using the MLflow CLI, as follows:

```
mlflow experiments create -n dl_model_chapter02
```

The preceding command creates a new experiment named `dl_model_chapter02`. If you have already run the first DL model with MLflow autologging in the previous section, the preceding command will cause an error message, as follows:

```
mlflow.exceptions.MlflowException: Experiment 'dl_model_chapter02' already exists.
```

This is to be expected, and you have done nothing wrong. Now if you list all the experiments in the local tracking server, it should include the newly created experiment, as shown here:



```
± [main U:1 7:10 x] → mlflow experiments list
Experiment Id  Name      Artifact Location
-----
0  Default  file:///Users/yongliu/opensource_code/Practical-Deep-Learning-at-Scale-with-MLFlow/chapter02/mlruns/0
1  dl_model_chapter02  file:///Users/yongliu/opensource_code/Practical-Deep-Learning-at-Scale-with-MLFlow/chapter02/mlruns/1
```

Figure 2.11 – The new MLflow experiments list after creating a new experiment

2. Now, let's examine the relationship between experiments and runs. If you look carefully at the URL of the run page (*Figure 2.8*), you will see something similar to the following:

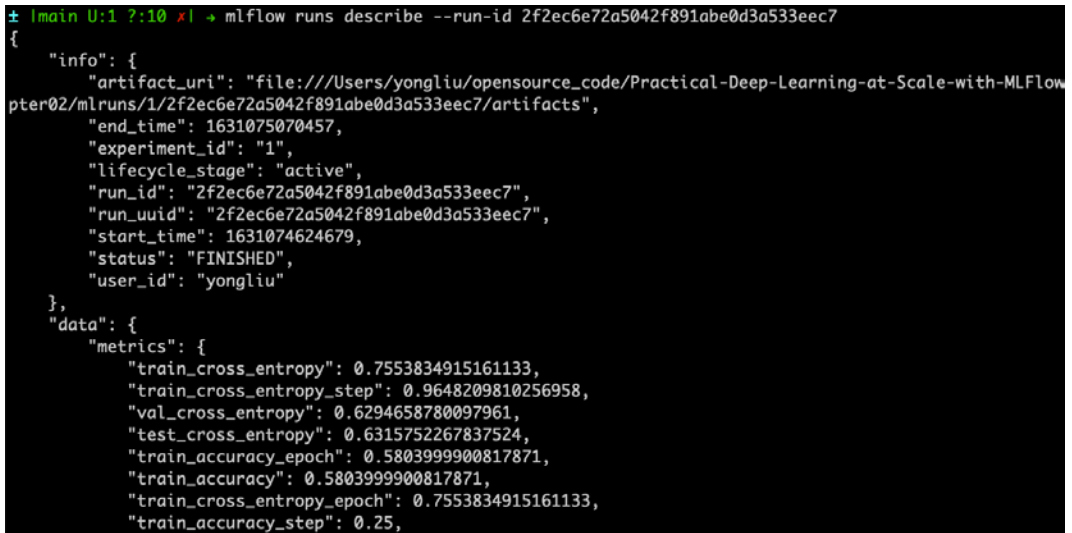
```
http://127.0.0.1:5000/#/experiments/1/
runs/2f2ec6e72a5042f891abe0d3a533eec7
```

As you might have gathered, the integer after the `experiments` path is the experiment ID. Then, after the experiment ID, there is a `runs` path, followed by a GUID-like random string, which is the run ID. So, now we understand how the runs are organized under the experiment with a globally unique ID (called a run ID).

Knowing a run's globally unique ID is very useful. This is because we can retrieve the run's logged data using `run_id`. If you use the `mlflow runs describe --run-id <run_id>` command line, you can get the list of metadata that this run has logged. For the experiment we just ran, the following shows the full command with the run ID:

```
mlflow runs describe --run-id
2f2ec6e72a5042f891abe0d3a533eec7
```

The output snippets of this command line are as follows (*Figure 2.12*):

A terminal window showing the command `mlflow runs describe --run-id 2f2ec6e72a5042f891abe0d3a533eec7` and its output in JSON format. The output includes metadata such as artifact URI, end time, experiment ID, lifecycle stage, run ID, run UUID, start time, status, and user ID. It also includes a 'data' section with metrics like train cross entropy, validation cross entropy, test cross entropy, train accuracy, and train accuracy epoch.

```
± |main U:1 ? :10 x| → mlflow runs describe --run-id 2f2ec6e72a5042f891abe0d3a533eec7
{
  "info": {
    "artifact_uri": "file:///Users/yongliu/opensource_code/Practical-Deep-Learning-at-Scale-with-MLFlow
pter02/mlruns/1/2f2ec6e72a5042f891abe0d3a533eec7/artifacts",
    "end_time": 1631075070457,
    "experiment_id": "1",
    "lifecycle_stage": "active",
    "run_id": "2f2ec6e72a5042f891abe0d3a533eec7",
    "run_uuid": "2f2ec6e72a5042f891abe0d3a533eec7",
    "start_time": 1631074624679,
    "status": "FINISHED",
    "user_id": "yongliu"
  },
  "data": {
    "metrics": {
      "train_cross_entropy": 0.7553834915161133,
      "train_cross_entropy_step": 0.9648209810256958,
      "val_cross_entropy": 0.6294658780097961,
      "test_cross_entropy": 0.6315752267837524,
      "train_accuracy_epoch": 0.5803999900817871,
      "train_accuracy": 0.5803999900817871,
      "train_cross_entropy_epoch": 0.7553834915161133,
      "train_accuracy_step": 0.25,
```

Figure 2.12 – The MLflow command line describes the run in the JSON data format

Note that *Figure 2.12* presents all the metadata about this run in JSON format. This metadata includes parameters used by the model training; metrics for measuring the accuracy of the model in training, validation, and testing; and more. The same data is also presented in the MLflow UI in *Figure 2.8*. Note that the powerful MLflow CLI will allow very convenient exploration of the MLflow logged metadata and artifacts as well as enabling shell script-based automation, as we will explore in the upcoming chapters.

## Exploring MLflow models and their usages

Now, let's explore how the DL model artifacts are logged in the MLflow tracking server. On the same run page, as shown in *Figure 2.8*, if you scroll down toward the bottom, you will see the Artifacts section (*Figure 2.13*). This lists all the metadata regarding the model and the serialized model itself:

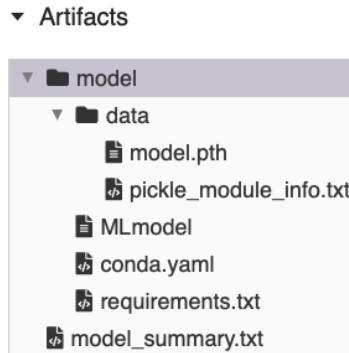


Figure 2.13 – The model artifacts logged by MLflow

### The MLflow Tracking Server's Backend Store and Artifact Store

An MLflow tracking server has two types of storage: first, a backend store, which stores experiments and runs metadata along with params, metrics, and tags for runs; and second, an artifact store, which stores larger files such as serialized models, text files, or even generated plots for visualizing model results. For the purpose of simplicity, in this chapter, we are using a local filesystem for both the backend store and the artifact store. However, some of the more advanced features such as model registry are not available in a filesystem-based artifact store. In later chapters, we will learn how to use a model registry.

Let's look at the list of artifacts, one by one:

- `model_summary.txt`: At the root folder level, this file looks similar to the following output if you click on it. It describes the model metrics and the layers of the DL model (please refer to *Figure 2.14*):

	Name	Type	Params
0	train_metrics	ModuleDict	0
1	train_metrics.accuracy	Accuracy	0
2	val_metrics	ModuleDict	0
3	val_metrics.accuracy	Accuracy	0
4	model	BertForSequenceClassification	4.4 M
5	model.bert	BertModel	4.4 M
6	model.bert.embeddings	BertEmbeddings	4.0 M
7	model.bert.embeddings.word_embeddings	Embedding	3.9 M
8	model.bert.embeddings.position_embeddings	Embedding	65.5 K
9	model.bert.embeddings.token_type_embeddings	Embedding	256
10	model.bert.embeddings.LayerNorm	LayerNorm	256
11	model.bert.embeddings.dropout	Dropout	0
12	model.bert.encoder	BertEncoder	396 K
13	model.bert.encoder.layer	ModuleList	396 K
14	model.bert.encoder.layer.0	BertLayer	198 K
15	model.bert.encoder.layer.0.attention	BertAttention	66.3 K
16	model.bert.encoder.layer.0.attention.self	BertSelfAttention	49.5 K
17	model.bert.encoder.layer.0.attention.self.query	Linear	16.5 K
18	model.bert.encoder.layer.0.attention.self.key	Linear	16.5 K
19	model.bert.encoder.layer.0.attention.self.value	Linear	16.5 K
20	model.bert.encoder.layer.0.attention.self.dropout	Dropout	0
21	model.bert.encoder.layer.0.attention.output	BertSelfOutput	16.8 K
22	model.bert.encoder.layer.0.attention.output.dense	Linear	16.5 K
25	model.bert.encoder.layer.0.intermediate	BertIntermediate	66.0 K
26	model.bert.encoder.layer.0.intermediate.dense	Linear	66.0 K
27	model.bert.encoder.layer.0.output	BertOutput	65.9 K
28	model.bert.encoder.layer.0.output.dense	Linear	65.7 K
29	model.bert.encoder.layer.0.output.LayerNorm	LayerNorm	256
30	model.bert.encoder.layer.0.output.dropout	Dropout	0
31	model.bert.encoder.layer.1	BertLayer	198 K
32	model.bert.encoder.layer.1.attention	BertAttention	66.3 K

Figure 2.14 – The model summary file logged by MLflow

Figure 2.14 provides a quick overview of what the DL model looks like in terms of the number and type of neural network layers, the number and size of the parameters, and the type of metrics used in training and validation. This is very helpful when the DL model architecture is needed to be shared and communicated among team members or stakeholders.

- The `model` folder: This folder contains a subfolder, called `data`, and three files called `MLmodel`, `conda.yaml`, and `requirements.txt`:
  - `MLmodel`: This file describes the flavor of the model that MLflow supports. **Flavor** is MLflow-specific terminology. It describes how the model is saved, serialized, and loaded. For our first DL model, the following information is stored in an `MLmodel` file (Figure 2.15):

```
artifact_path: model
flavors:
  python_function:
    data: data
    env: conda.yaml
    loader_module: mlflow.pytorch
    pickle_module_name: mlflow.pytorch.pickle_module
    python_version: 3.8.10
  pytorch:
    model_data: data
    pytorch_version: 1.9.0
run_id: 2f2ec6e72a5042f891abe0d3a533eec7
utc_time_created: '2021-09-08 04:23:49.531445'
```

Figure 2.15 – Content of the `MLmodel` file for our first DL model run with MLflow  
Figure 2.15 illustrates that this is a PyTorch flavor model with `run_id` that we have just run.

- `conda .yaml`: This is a conda environment definition file used by the model to describe our dependencies. *Figure 2.16* lists the content logged by MLflow in the run we just completed:

```
channels:  
- conda-forge  
dependencies:  
- python=3.8.10  
- pip  
- pip:  
  - mlflow  
  - av==8.0.3  
  - boto3==1.18.31  
  - cloudpickle==1.6.0  
  - datasets==1.2.1  
  - lightning-flash==0.5.0  
  - psutil==5.8.0  
  - pydeprecate==0.3.1  
  - rouge-score==0.0.4  
  - scikit-learn==0.24.2  
  - sentencepiece==0.1.96  
  - timm==0.4.12  
  - torch==1.9.0  
  - torchvision==0.10.0  
  - transformers==4.9.2  
name: mlflow-env
```

Figure 2.16 – The content of the `conda.yaml` file logged by MLflow

- `requirements.txt`: This is a Python `pip`-specific dependency definition file. It is just like the `pip` section in the `conda .yaml` file, as shown in *Figure 2.16*.
- `data`: This is a folder that contains the actual serialized model, called `model.pth`, and a description file, called `pickle_module_info.txt`, whose content for our first DL experiment is as follows:

```
mlflow.pytorch.pickle_module
```

This means the model is serialized using a PyTorch-compatible pickle serialization method provided by MLflow. This allows MLflow to load the model back to memory later if needed.

### Model Registry versus Model Logging

The MLflow model registry requires a relational database such as MySQL as the artifact store, not just a plain filesystem. Therefore, in this chapter, we will not explore it yet. Note that a model registry is different from model logging in that, for each run, you want to log model metadata and artifacts. However, only for certain runs that meet your production requirements, you may want to register them in the model registry for production deployment and version control. In later chapters, we will learn how to register models.

By now, you should have a good understanding of the list of files and metadata about the model and the serialized model (along with the `.pth` file extension in our experiment, which refers to a PyTorch serialized model) logged in the MLflow artifact store. In the upcoming chapters, we will learn more about how the MLflow model flavor works and how to use the logged model for model registry and deployment. MLflow model flavors are model frameworks such as PyTorch, TensorFlow, and scikit-learn, which are supported by MLflow. Interested readers can find more details about the current built-in model flavors supported by MLflow from the official MLflow documentation site at <https://www.mlflow.org/docs/latest/models.html#built-in-model-flavors>.

## Exploring MLflow code tracking and its usages

When exploring the metadata of the run, we can also discover how the code is being tracked. As shown in the MLflow UI and the command-line output in JSON, the code is tracked in three ways: a filename, a Git commit hash, and a source type. You can execute the following command line:

```
mlflow runs describe --run-id 2f2ec6e72a5042f891abe0d3a533eec7
| grep mlflow.source
```

You should be able to find the following segments of JSON key-value pairs in the output:

```
"mlflow.source.git.commit":
"ad6c7338a416ff4c2848d726b092057457c22408",
"mlflow.source.name": "first_dl_with_mlflow.py",
"mlflow.source.type": "LOCAL"
```

Based on this `ad6c7338a416ff4c2848d726b092057457c22408` Git commit hash, we can go on to find the exact copy of the Python code we used: [https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLflow/blob/ad6c7338a416ff4c2848d726b092057457c22408/chapter02/first\\_dl\\_with\\_mlflow.py](https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLflow/blob/ad6c7338a416ff4c2848d726b092057457c22408/chapter02/first_dl_with_mlflow.py).



Note that, here, the source type is `LOCAL`. This means that we execute the MLflow-enabled source code from a local copy of the code. In later chapters, we will learn about other source types.

#### LOCAL versus Remote GitHub Code

If the source is a local copy of the code, there is a caveat regarding the Git commit hash that you see in the MLflow tracking server. If you make code changes locally but forget to commit them and then immediately start an MLflow experiment tracking run, MLflow will only log the most recent Git commit hash. We can solve this problem in one of two ways:

1. Commit our code changes before running the MLflow experiment.
2. Use remote GitHub code to run the experiment.

Since the first method is not easily enforceable, the second method is preferred. Using remote GitHub code to run a DL experiment is an advanced topic that we will explore in later chapters.

So far, we have learned about the MLflow tracking server, experiments, and runs. Additionally, we have logged metadata about runs such as parameters and metrics, examined code tracking, and explored model logging. These tracking and logging capabilities ensure that we have a solid ML experiment management system, not only for model development but also for model deployment in the future, as we need to track which runs produce the model for production. *Reproducibility* and *provenance-tracking* are the hallmarks of what MLflow provides. In addition to this, MLflow provides other components such as **MLproject** for standardized ML project code organization, a model registry for model versioning control, model deployment capabilities, and model explainability tools. All of these MLflow components cover the whole life cycle of ML/DL development, deployment, and production, which we will examine in more depth in future chapters.

## Summary

In this chapter, we learned how to set up MLflow to work with either a local MLflow tracking server or a remote MLflow tracking server. Then, we implemented our first DL model with MLflow autologging enabled. This allowed us to explore MLflow in a hands-on way to understand a few central concepts and foundational components such as experiments, runs, metadata about experiments and runs, code tracking, model logging, and model flavor. The knowledge and first-round experiences gained in this chapter will help us to learn more in-depth MLflow tracking APIs in the next chapter.

## Further reading

To further your knowledge, you can consult the following resources and documentation:

- The MLflow *Command-Line Interface* documentation: <https://www.mlflow.org/docs/latest/cli.html>
- The MLflow PyTorch autologging documentation: <https://www.mlflow.org/docs/latest/tracking.html#pytorch-experimental>
- The MLflow PyTorch model flavor documentation: [https://www.mlflow.org/docs/latest/python\\_api/mlflow.pytorch.html#module-mlflow.pytorch](https://www.mlflow.org/docs/latest/python_api/mlflow.pytorch.html#module-mlflow.pytorch)
- *MLflow and PyTorch — Where Cutting Edge AI meets MLOps*: <https://medium.com/pytorch/mlflow-and-pytorch-where-cutting-edge-ai-meets-mlops-1985cf8aa789>
- *Controlled Experiments in Machine Learning*: <https://machinelearningmastery.com/controlled-experiments-in-machine-learning/>



# Section 2 – Tracking a Deep Learning Pipeline at Scale

In this section, we will learn how to use MLflow to track **deep learning (DL)** pipelines to answer various provenance-tracking questions related to data, model, and code (including both notebook and pipeline code). We will start with setting up a local full-fledged MLflow tracking server that will be used frequently in the rest of this book. A provenance tracking framework that includes six types of provenance questions will be presented to guide our implementation. Then, we will learn how to track model provenance, metrics, and parameters using MLflow to answer these provenance questions. We will also learn how to choose an appropriate notebook and pipeline framework to implement DL code that's extensible and trackable. We will then implement a multi-step DL training/testing/registration pipeline using MLflow's **MLproject**. Finally, we will learn how to track public and privately built Python libraries and data versioning using **Delta Lake**.

This section comprises the following chapters:

- *Chapter 3, Tracking Models, Parameters, and Metrics*
- *Chapter 4, Tracking Code and Data Versioning*



# 3

## Tracking Models, Parameters, and Metrics

Given that MLflow can support multiple scenarios through the life cycle of DL models, it is common to use MLflow's capabilities incrementally. Usually, people start with MLflow tracking since it is easy to use and can handle many scenarios for reproducibility, provenance tracking, and auditing purposes. In addition, tracking the history of a model from cradle to sunset not only goes beyond the data science experiment management domain but is also important for model governance in the enterprise, where business and regulatory risks need to be managed for using models in production. While the precise business values of tracking models in production are still evolving, the need for tracking a model's entire life cycle is unquestionable and growing. For us to be able to do this, we will begin this chapter by setting up a full-fledged local MLflow tracking server.

We will then take a deep dive into how we can track a model, along with its parameters and metrics, using MLflow's tracking and registry APIs. By the end of this chapter, you should feel comfortable using MLflow's tracking and registry APIs for various reproducibility and auditing purposes.

In this chapter, we're going to cover the following main topics:

- Setting up a full-fledged local MLflow tracking server
- Tracking model provenance
- Tracking model metrics
- Tracking model parameters

## Technical requirements

The following are the requirements you will need to follow the instructions provided in this chapter:

- Docker Desktop: <https://docs.docker.com/get-docker/>.
- PyTorch lightning-flash: 0.5.0: <https://github.com/PyTorchLightning/lightning-flash/releases/tag/0.5.0>.
- VS Code with the Jupyter Notebook extension: <https://github.com/microsoft/vscode-jupyter/wiki/Setting-Up-Run-by-Line-and-Debugging-for-Notebooks>.
- The following GitHub URL for the code for this chapter: <https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLflow/tree/main/chapter03>.
- WSL2: If you are a Microsoft Windows user, it is recommended to install WSL2 to run the Bash scripts provided in this book: <https://www.windowscentral.com/how-install-wsl2-windows-10>.

## Setting up a full-fledged local MLflow tracking server

In *Chapter 2, Getting Started with MLflow for Deep Learning*, we gained hands-on experience working with a local filesystem-based MLflow tracking server and inspecting the components of the MLflow experiment. However, there are limitations with a default local filesystem-based MLflow server as the model registry functionality is not available. The benefit of having a model registry is that we can register the model, version control the model, and prepare for model deployment into production. Therefore, this model registry will bridge the gap between offline experimentation and an online deployment production scenario. Thus, we need a full-fledged MLflow tracking server with the following stores to track the complete life cycle of a model:

- **Backend store:** A relational database backend is needed to support MLflow's storage of metadata (metrics, parameters, and many others) about the experiment. This also allows the query capability of the experiment to be used. We will use a MySQL database as a local backend store.
- **Artifact store:** An object store that can store arbitrary types of objects, such as serialized models, vocabulary files, figures, and many others. In a production environment, a popular choice is the AWS S3 store. We will use **MinIO** (<https://min.io/>), a multi-cloud object store, as a local artifact store, which is fully compatible with the AWS S3 store API but can run on your laptop without you needing to access the cloud.

To make this local setup as easy as possible, we will use the `docker-compose` (<https://docs.docker.com/compose/>) tool with one line of command to start and stop a local full-fledged MLflow tracking server, as described in the following steps. Note that Docker Desktop (<https://docs.docker.com/get-docker/>) must be installed and running on the machine before you can follow these steps. Docker helps build and share containerized applications and microservices. The following steps will launch the local MLflow tracking server inside your local Docker container:

1. Check out the `chapter03` code repository for your local development environment: <https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLflow/tree/main/chapter03>.
2. Change directory to the `mlflow_docker_setup` subfolder, which can be found under the `chapter03` folder.
3. Run the following command:

```
bash start_mlflow.sh
```



If the command is successful, you should see an output similar to the following on your screen:

```
Creating mlflow_s3      ... done
Creating mlflow_db     ... done
Creating mlflow_server ... done
Creating create_mlflow_bucket ... done
Creating mlflow_nginx ... done
```

Figure 3.1 – A local full-fledged MLflow tracking server is up and running

- Go to `http://localhost/` to see the MLflow UI web page. Then, click the **Models** tab in the UI (Figure 3.2). Note that this tab would not work if you only had a local filesystem as the backend store for the MLflow tracking server. Hence, the MLflow UI's backend is now running on the Docker container service you just started, not a local filesystem. Since this is a brand-new server, there are no registered models yet:

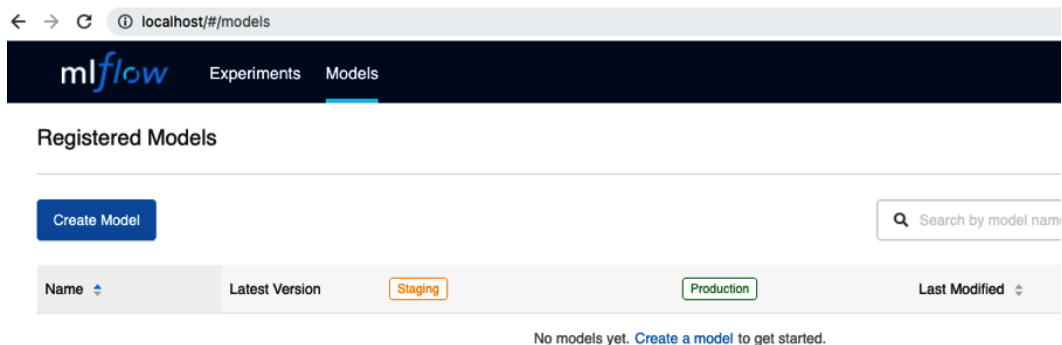


Figure 3.2 – MLflow model registry UI

- Go to `http://localhost:9000/`, and the following screen (Figure 3.3) should appear for the MinIO artifact store web UI. Enter `minio` for **Access Key** and `minio123` for **Secret Key**. These are defined in the `.env` file, under the `mlflow_docker_setup` folder:

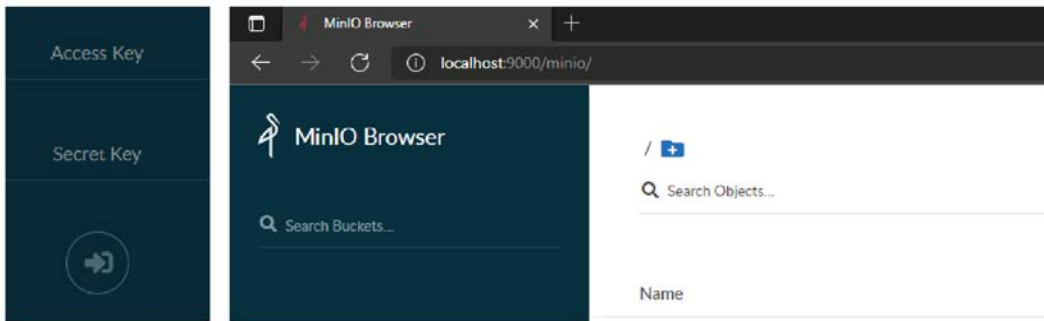


Figure 3.3 – MinIO Web UI login page and browser page after logging in

At this point, you should have a full-fledged local MLflow tracking server running successfully! If you want to stop the server, simply type the following command:

```
bash stop_mlflow.sh
```

The Docker-based MLflow tracking server will stop. We are now ready to use this local MLflow server to track model provenance, parameters, and metrics.

## Tracking model provenance

**Provenance** tracking for digital artifacts has been long studied in the literature. For example, when you're using a piece of patient diagnosis data in the biomedical industry, people usually want to know where it comes from, what kind of processing and cleaning has been done to the data, who owns the data, and other history and lineage information about the data. The rise of ML/DL models for industrial and business scenarios in production makes provenance tracking a required functionality. The different granularities of provenance tracking are critical for operationalizing and managing not just the data science offline experimentation, but also before/during/after the model is deployed in production. So, what needs to be tracked for provenance?

## Understanding the open provenance tracking framework

Let's look at a general provenance tracking framework to understand the big picture of why provenance tracking is a major effort. The following diagram is based on the **Open Provenance Model Vocabulary Specification** (<http://open-biomed.sourceforge.net/opmv/ns.html>):

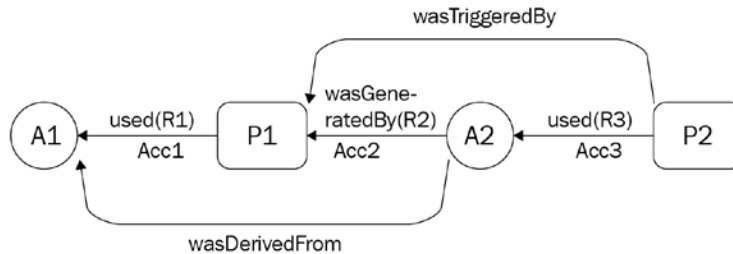


Figure 3.4 – An open provenance tracking framework

In the preceding diagram, there are three important items:

- **Artifacts:** Things that are produced or used by processes (**A1** and **A2**).
- **Processes:** Actions that are performed by using or producing artifacts (**P1** and **P2**).
- **Causal relationships:** Edges or relationships between artifacts and processes, such as *used*, *wasGeneratedBy*, and *wasDerivedFrom* in the preceding diagram (**R1**, **R2**, and **R3**).

Intuitively, this **open provenance model (OPM)** framework allows us to ask the following 5W1H (five Ws and one H) questions, as follows:

Provenance Type	Questions	Model Provenance Example
What	What process generated this artifact?	What model framework or model version was used?
When	When was this artifact generated?	What time was this model produced?
Why	Why was this artifact generated?	Why was this model promoted to production?
Where	Where was this artifact located in the workflow?	Which step was this model located in the model pipeline?
Who	Who generated or used this artifact?	Who ran the model pipeline?
How	How was this artifact generated?	How was the model trained and with what parameters?

Figure 3.5 – Types of provenance questions

Having a systematic provenance framework and a set of questions will help us learn how to track model provenance and provide answers to these questions. This will motivate us when we implement MLflow model tracking in the next section.

## Implementing MLflow model tracking

We can use an MLflow tracking server to answer most of these types of provenance questions if we implement both MLflow logging and registry for the DL model we use. First, let's review what MLflow provides in terms of model provenance tracking. MLflow provides two sets of APIs for model provenance:

- **Logging API:** This allows each run of the experiment or a model pipeline to log the model artifact into the artifact store.
- **Registry API:** This allows a centralized location to track the version of the model and the stages of the model's life cycle (**None**, **Archived**, **Staging**, or **Production**).

### Difference between Model Logging and Model Registry

Although every run of the experiment needs to be logged and the model needs to be saved in the artifact store, not every instance of the model needs to be registered in the model registry. That's because, for many early exploratory model experimentations, the model might not be good. Thus, it is not necessarily registered to track the version. Only when a model has good offline performance and becomes a candidate for promoting to production do we need to register it in the model registry to go through the model promotion process.

Although MLflow's official API documentation separates logging and registry into two components, we will refer to them together as model tracking functionality in MLflow in this book.

We already saw MLflow's auto-logging for the DL model we built in *Chapter 2, Getting Started with MLflow for Deep Learning*. Although auto-logging is powerful, there are two issues with the current version:

- It does not automatically register the model to the model registry.
- It does not work out of the box for the logged model to work directly with the original input data (in our case, an English sentence) if you just follow MLflow's suggestion to use the `mlflow.pyfunc.load_model` API to load the logged model. This is a limitation that's probably due to the experimental nature of the current auto-logging APIs in MLflow.

Let's walk through an example to review MLflow's capabilities and auto-logging's limitations and how we can solve them:

1. Set up the following environment variables in your Bash terminal, where your MinIO and MySQL-based Docker component is running:

```
export MLFLOW_S3_ENDPOINT_URL=http://localhost:9000
export AWS_ACCESS_KEY_ID=minio
export AWS_SECRET_ACCESS_KEY=minio123
```

Note that `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` are using the same values that were defined in the `.env` file, under the `mlflow_docker_setup` folder. This is done to make sure that we are using the MLflow server that we set up previously. Since these environmental variables are session-based, we can also set up the following environment variables in the notebook's code, as follows:

```
os.environ["AWS_ACCESS_KEY_ID"] = "minio"
os.environ["AWS_SECRET_ACCESS_KEY"] = "minio123"
```

```
os.environ["MLFLOW_S3_ENDPOINT_URL"] = "http://
localhost:9000"
```

The preceding three lines of code can be found in this chapter's notebook file, just after importing the required Python packages. Before you execute the notebook, make sure that you run the following commands to initialize the virtual environment, `dl_model`, which now has additional required packages defined in the `requirements.txt` file:

```
conda create -n dl_model python==3.8.10
conda activate dl_model
pip install -r requirements.txt
```

If you set up the `dl_model` virtual environment in the previous chapters, you can skip the first line on creating a virtual environment called `dl_model`. However, you still need to activate `dl_model` as the currently active virtual environment and then run `pip install -r requirements.txt` to install all the required Python packages. Once the `dl_model` virtual environment has been set up successfully, you may proceed to the next step.

2. To follow along with this model tracking implementation, check out the `dl_model_tracking.ipynb` notebook file in VS Code by going to this chapter's GitHub repository: [https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLflow/blob/main/chapter03/dl\\_model\\_tracking.ipynb](https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLflow/blob/main/chapter03/dl_model_tracking.ipynb).

Note that, in the fourth cell of the `dl_model_tracking.ipynb` notebook, we need to point it to the correct and new MLflow tracking URI that we just set up in the Docker and define a new experiment, as follows:

```
EXPERIMENT_NAME = "dl_model_chapter03"
mlflow.set_tracking_uri('http://localhost')
```

3. We will still use the auto-logging capabilities provided by MLflow but we will assign the run with a variable name, `dl_model_tracking_run`:

```
mlflow.pytorch.autolog()
with mlflow.start_run(experiment_id=experiment.
experiment_id, run_name="chapter03") as dl_model_
tracking_run:
    trainer.finetune(classifier_model,
datamodule=datamodule, strategy="freeze")
    trainer.test()
```

`dl_model_tracking_run` allows us to get the `run_id` parameter and other metadata about this run programmatically, as we will see in the next step. Once this code cell has been executed, we will have a trained model logged in the MLflow tracking server with all the required parameters and metrics. However, the model hasn't been registered yet. We can find the logged experiment in the MLflow web UI, along with all the relevant parameters and metrics, at <http://localhost/#/experiments/1/runs/37a3fe9b6faf41d89001eca13ad6ca47>. You can find the model artifacts in the **MinIO** storage backend. Go to <http://localhost:9000/minio/mlflow/1/37a3fe9b6faf41d89001eca13ad6ca47/artifacts/model/> to see the storage UI, as shown here:

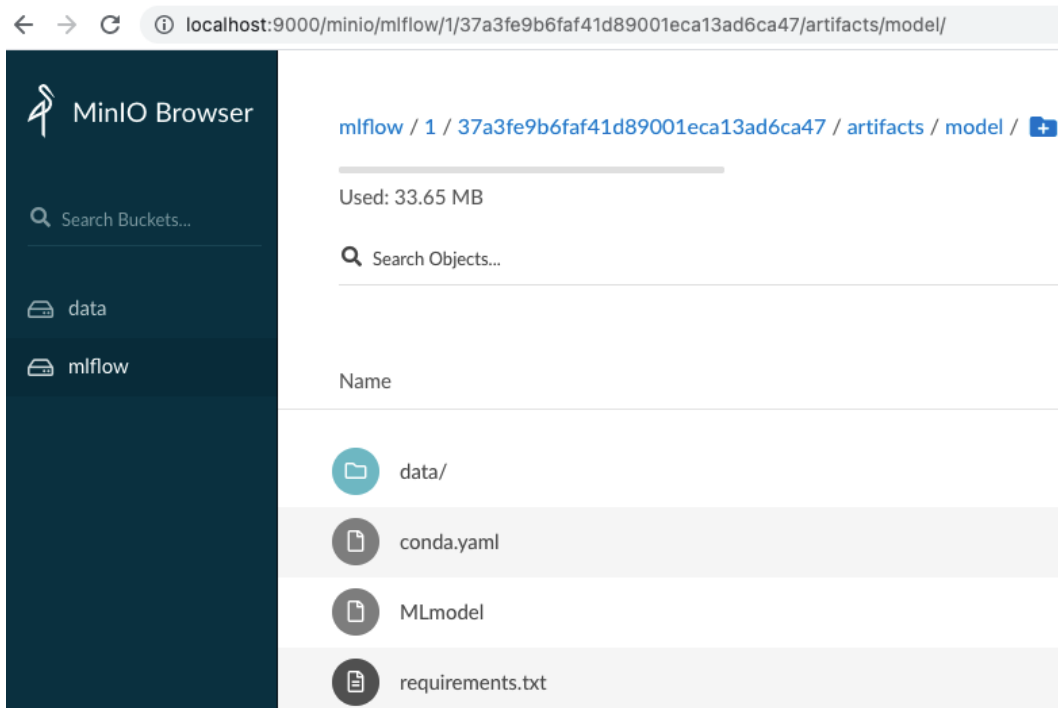


Figure 3.6 – Model artifacts logged in the MinIO storage backend

The folder structure is similar to what we saw in *Chapter 2, Getting Started with MLflow for Deep Learning*, when we used a plain filesystem to store the model artifacts. However, here, we are using a **MinIO** bucket to store these model artifacts.

- Retrieve the `run_id` parameter from `dl_model_tracking_run`, as well as other metadata, as follows:

```
run_id = dl_model_tracking_run.info.run_id
print("run_id: {}; lifecycle_stage: {}".format(run_id,
        mlflow.get_run(run_id).info.lifecycle_stage))
```

This will print out something like the following:

```
run_id: 37a3fe9b6faf41d89001eca13ad6ca47; lifecycle_
stage: active
```

- Retrieve the logged model by defining the logged model URI. This will allow us to reload the logged model at this specific location:

```
logged_model = f'runs:{run_id}/model'
```

- Use `mlflow.pytorch.load_model` and the following `logged_model` URI to load the model back into memory and make a new prediction for a given input sentence, as follows:

```
model = mlflow.pytorch.load_model(logged_model)
model.predict({'This is great news'})
```

This will output a model prediction label, as follows:

```
['positive']
```

#### **mlflow.pytorch.load\_model versus mlflow.pyfunc.load\_model**

By default, and in the MLflow experiment tracking page's artifact section, if you have a logged model, MLflow will recommend using `mlflow.pyfunc.load_model` to load back a logged model for prediction. However, this only works for inputs such as a pandas DataFrame, NumPy array, or tensor; this does not work for an NLP text input. Since auto-logging for PyTorch lightning uses `mlflow.pytorch.log_model` to save the model, the correct way to load a logged model back is to use `mlflow.pytorch.load_model`, as we have shown here. This is because MLflow's default design is to use `mlflow.pyfunc.load_model` with standardization and a known limitation that can only accept input formats in terms of numbers. For text and image data, it requires a tokenization step as a preprocessing step. However, since the PyTorch model we saved here already performs tokenization as part of the serialized model, we can use the native `mlflow.pytorch.load_model` to directly load the model that accepts text as inputs.



With that, we have successfully logged the model and loaded the model back to make a prediction. If we think this model is performing well enough, then we can register it.

7. Let's register the model by using the `mlflow.register_model` API:

```
model_registry_version = mlflow.register_model(logged_model, 'nlp_dl_model')
print(f'Model Name: {model_registry_version.name}')
print(f'Model Version: {model_registry_version.version}')
```

This will produce the following output:

```
Successfully registered model 'nlp_dl_model'.
2021/10/21 20:14:42 INFO mlflow.tracking._model_registry.client: Waiting up to 300 seconds for model
version to finish creation.           Model name: nlp_dl_model, version 1

Model Name: nlp_dl_model
Model Version: 1

Created version '1' of model 'nlp_dl_model'.
```

Figure 3.7 – Model registration success message

This shows that the model has been successfully registered as version 1 in the model registry, under the name `nlp_dl_model`.

We can also find this registered model in the MLflow web UI by clicking [http://localhost/#/models/nlp\\_dl\\_model/versions/1](http://localhost/#/models/nlp_dl_model/versions/1):

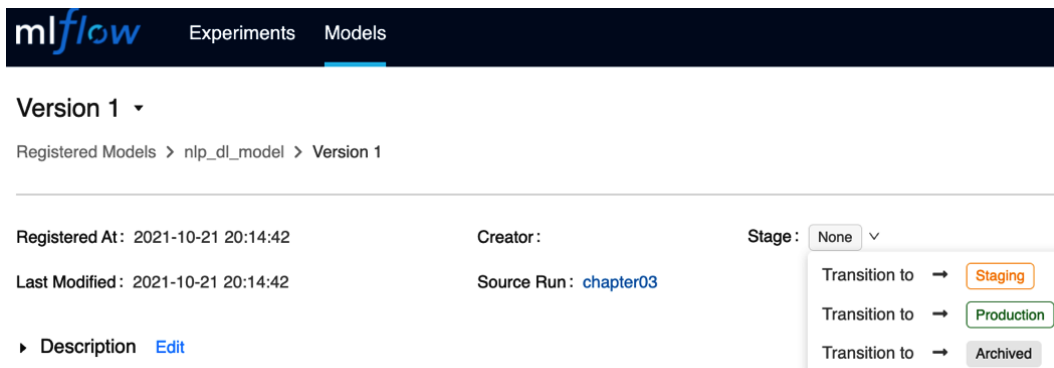


Figure 3.8 – MLflow tracking server web UI showing the newly registered model  
By default, a newly registered model's stage is **None**, as shown in the preceding screenshot.

By having a model registered with a version number and stage label, we have laid the foundation for deployment to staging (also known as pre-production) and then production. We will discuss how to perform model deployment based on registered models later in this book.

At this point, we have solved the two issues we raised at the beginning of this section regarding the limitations of auto-logging:

- How to load a logged DL PyTorch model using the `mlflow.pytorch.load_model` API instead of the `mlflow.pyfunc.load_model` API
- How to register a logged DL PyTorch model using the `mlflow.register_model` API

#### Choices of MLflow DL Model Logging APIs

For DL models, the auto-logging for PyTorch only works for **PyTorch lightning** frameworks. There are other DL frameworks, such as **TensorFlow**, **Keras**, **fastai**, and **MXNet**, that are also supported by the corresponding MLflow auto-logging APIs. For other PyTorch frameworks such as **Hugging Face**, we can use MLflow's `mlflow.pyfunc.log_model` to log the model, especially when we need to have multi-step DL model pipelines. We will implement such custom MLflow model flavors later in this book. If you don't want to use auto-logging for PyTorch, then you can directly use `mlflow.pytorch.log_model`. PyTorch's auto-logging uses `mlflow.pytorch.log_model` inside its implementation (see the official MLflow open source implementation here: [https://github.com/mlflow/mlflow/blob/290bf3d54d1e5ce61944455cb302a5d6390107f0/mlflow/pytorch/\\_pytorch\\_autolog.py#L314](https://github.com/mlflow/mlflow/blob/290bf3d54d1e5ce61944455cb302a5d6390107f0/mlflow/pytorch/_pytorch_autolog.py#L314)).

If we don't want to use auto-logging, then we can use MLflow's model logging API directly. This also gives us an alternative way to simultaneously register the model in one call. You can use the following line of code to both log and register the trained model:

```
mlflow.pytorch.log_model(pytorch_model=trainer.model, artifact_path='dl_model', registered_model_name='nlp_dl_model')
```

Note that this line of code does not log any parameters or metrics of the model.

With that, we have not only logged many experiments and models in the tracking server for offline experimentation but also registered performant models for production deployment in the future with version control and provenance tracking. We can now answer some of the provenance questions that we posted at the beginning of this chapter:

Provenance Type	Model Provenance Question	Answers Based on MLflow Model Tracking and Registry
What	What model framework or model version was used?	Version 1 is currently being used.
When	What time was this model produced?	2021.10.21 ( <a href="#">see Figure 3.8</a> )
Why	Why was this model promoted to production?	We will answer this when we discuss how to deploy models later in this book
Where	Which step was this model located in the model pipeline?	This is currently a single-step pipeline. We will discuss multi-step pipelines later in this book.
Who	Who ran the model pipeline?	yongliu
How	How was the model trained and with what parameters?	Take a look at the MLflow tracking page for run_id = 37a3fe9b6faf41d89001eca13ad6ca47

Figure 3.9 – Answers to model provenance questions

The why and where provenance questions are yet to be fully answered but will be done so later in this book. This is because the why provenance question for the production model can only be tracked and logged when the model is ready for deployment, where we need to add comments and reasons to justify the model's deployment. The where provenance question can be answered fully when we have a multiple-step model pipeline. However, here, we only have a single-step pipeline, which is the simplest case. A multi-step pipeline contains explicitly separate modularized code to specify which step performs what functionality so that we can easily change the detailed implementation of any of the steps without changing the flow of the pipeline. In the next two sections, we will investigate how we can track metrics and the parameters of models without using auto-logging.

## Tracking model metrics

The default metric for the text classification model in the PyTorch `lightning-flash` package is **Accuracy**. If we want to change the metric to **F1 score** (a harmonic mean of precision and recall), which is a very common metric for measuring a classifier's performance, then we need to change the configuration of the classifier model before we start the model training process. Let's learn how to make this change and then use MLflow's non-auto-logging API to log the metrics:

1. When defining the classifier variable, instead of using the default metric, we will pass a metric function called `torchmetrics.F1` as a variable, as follows:

```
classifier_model = TextClassifier(backbone="prajjwal1/
bert-tiny", num_classes=datamodule.num_classes,
metrics=torchmetrics.F1(datamodule.num_classes))
```

This uses the built-in metrics function of `torchmetrics`, the `F1` module, along with the number of classes in the data we need to classify as a parameter. This makes sure that the model is trained and tested using this new metric. You will see an output similar to the following:

```
{'test_cross_entropy': 0.785443127155304, 'test_f1':
0.5343999862670898}
```

This shows that the model training and testing were using the F1 score as the metric, not the default accuracy metric. For more information on how you can use `torchmetrics` for customized metrics, please consult its documentation site: <https://torchmetrics.readthedocs.io/en/latest/>.

2. Now, if we want to log all the metrics to the MLflow tracking server, including the training, validation, and testing metrics, we need to get all the current metrics by calling the trainer's callback function, as follows:

```
cur_metrics = trainer.callback_metrics
```

Then, we need to cast all the metric values to `float` to make sure that they are compatible with the MLflow `log_metrics` API:

```
metrics = dict(map(lambda x: (x[0], float(x[1])),
cur_metrics.items()))
```

3. Now, we can call MLflow's `log_metrics` to log all the metrics in the tracking server:

```
mlflow.log_metrics(metrics)
```

You will see the following metrics after using the F1 score as the classifier's metric, which will be logged in MLflow's tracking server:

```
{'train_f1': 0.5838666558265686,
 'train_f1_step': 0.75,
 'train_cross_entropy': 0.7465656399726868,
 'train_cross_entropy_step': 0.30964696407318115,
 'val_f1': 0.5203999876976013,
 'val_cross_entropy': 0.8168156743049622,
 'train_f1_epoch': 0.5838666558265686,
 'train_cross_entropy_epoch': 0.7465656399726868,
 'test_f1': 0.5343999862670898,
 'test_cross_entropy': 0.785443127155304}
```

Using MLflow's `log_metrics` API gives us more control with additional lines of code, but if we are satisfied with its auto-logging capabilities, then the only thing we need to change is what metric we want to use for the model training and testing processes. In this case, we only need to define a new metric to use when declaring a new DL model (that is, use the F1 score instead of the default accuracy metric).

4. If you want to track multiple model metrics simultaneously, such as the F1 score, accuracy, precision, and recall, then the only thing you need to do is define a Python list of metrics you want to compute and track, as follows:

```
list_of_metrics = [torchmetrics.Accuracy(),
                   torchmetrics.F1(num_classes=datamodule.num_classes),
                   torchmetrics.Precision(num_classes=datamodule.num_
classes),
                   torchmetrics.Recall(num_classes=datamodule.num_
classes)]
```

Then, in the model initialization statement, instead of passing a single metric to the `metrics` parameter, you can just pass the `list_of_metrics` Python list that we just defined, above the `metrics` parameter, as follows:

```
classifier_model = TextClassifier(backbone="prajjwal/  
bert-tiny", num_classes=datamodule.num_classes,  
metrics=list_of_metrics)
```

No more changes need to be made to the rest of the code. So, in the `dl_model-non-auto-tracking.ipynb` notebook ([https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter03/dl\\_model-non-auto-tracking.ipynb](https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter03/dl_model-non-auto-tracking.ipynb)), you will notice that the preceding line is commented out by default. However, you can uncomment it and then comment out the previous one:

```
classifier_model = TextClassifier(backbone="prajjwal1/  
bert-tiny", num_classes=datamodule.num_classes,  
metrics=torchmetrics.F1(datamodule.num_classes))
```

Then, when you run the rest of the notebook, you will get the model testing reports, along with the following metrics, in the notebook's output:

```
{'test_accuracy': 0.6424000263214111, 'test_  
cross_entropy': 0.6315688490867615, 'test_f1':  
0.6424000263214111, 'test_precision': 0.6424000263214111,  
'test_recall': 0.6424000263214111}
```

You may notice that the numbers for accuracy, F1, precision, and recall are the same. This is because, by default, `torchmetrics` uses a **micro-average** method, which computes a single scalar average score for all the classes by counting total true positives, false negatives, and false positives. Scikit-learn has an average option called **binary** that outputs only the score for the positive label when it is a binary classification model ([https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1\\_score.html#](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html#)). However, `torchmetrics` does not support a **binary** average method for a binary classification model. The only alternative is to use a `none` method, which computes the metric for each class and returns the metric for each class, even for a binary classification model. So, this does not produce a single scalar number. However, you can always call scikit-learn's metrics API to compute an F1-score or other metrics based on the binary average method by passing two lists of values. Here, we can use `y_true` and `y_predict`, where `y_true` is the list of ground truth label values and `y_predict` is the list of model predicted label values. This can be a good exercise for you to try out as this is a common practice for all ML models, not special treatment for a DL model.

## Tracking model parameters

As we have already seen, there are lots of benefits of using auto-logging in MLflow, but if we want to track additional model parameters, we can either use MLflow to log additional parameters on top of what auto-logging records, or directly use MLflow to log all the parameters we want without using auto-logging at all.

Let's walk through a notebook without using MLflow auto-logging. If we want to have full control of what parameters will be logged by MLflow, we can use two APIs: `mlflow.log_param` and `mlflow.log_params`. The first one logs a single pair of key-value parameters, while the second logs an entire dictionary of key-value parameters. So, what kind of parameters might we be interested in tracking? The following answers this:

- **Model hyperparameters:** **Hyperparameters** are defined before the learning process begins, which means they control how the learning process learns. These parameters can be turned and can directly affect how well a model trains. In a DL model, the list of hyperparameters includes the backbone language model, learning rate, loss function, the optimizer to be used, and many more. MLflow's auto-logging does not automatically log all the hyperparameters, so this is an opportunity for us to directly use MLflow's `log_params` API to record them in the experiment.
- **Model parameters:** These parameters are learned during the model training process. For a DL model, these usually refer to the neural network weights that are learned during training. We don't need to log these weight parameters individually since they are already in the logged DL model.

Let's log these hyperparameters using MLflow's `log_params` API, as follows:

```
params = {"epochs": trainer.max_epochs}
if hasattr(trainer, "optimizers"):
    optimizer = trainer.optimizers[0]
    params["optimizer_name"] = optimizer.__class__.__name__
if hasattr(optimizer, "defaults"):
    params.update(optimizer.defaults)
params.update(classifier_model.hparams)
mlflow.log_params(params)
```

Note that here, we log the maximal number of epochs, the trainer's first optimizer's name, the optimizer's default parameters, and the overall classifier's hyperparameters (`classifier_model.hparams`). The one-line piece of code `mlflow.log_params(params)` logs all the key-value parameters in the `params` dictionary to the MLflow tracking server. If you see the following hyperparameters in the MLflow tracking server, then it means it works!

▼ Parameters

Name	Value
amsgrad	False
backbone	prajjwal1/bert-tiny
betas	(0.9, 0.999)
enable_ort	False
epochs	3
eps	1e-08
learning_rate	0.01
loss_fn	None
lr	0.01
metrics	F1()
multi_label	False
num_classes	2
optimizer	<class 'torch.optim.adam.Adam'>
optimizer_kwargs	None
optimizer_name	Adam
scheduler	None
scheduler_kwargs	None
serializer	None
weight_decay	0

Figure 3.10 – MLflow tracking server web UI showing the logged model hyperparameters



Notice that this list of parameters is more than what the auto-logger logs as we added additional hyperparameters to log in the experiment. If you want to log any other customized parameters, you can follow the same pattern in your experiment. The complete notebook, without the use of auto-logging, can be checked out in this chapter's GitHub repository at [https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter03/dl\\_model-non-auto-tracking.ipynb](https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter03/dl_model-non-auto-tracking.ipynb).

If you have reached this point in this chapter, then you have successfully implemented an MLflow tracking model and its metrics and parameters!

## Summary

In this chapter, we set up a local MLflow development environment that has full support for backend storage and artifact storage using MySQL and the MinIO object store. This will be very useful for us when we develop MLflow-supported DL models in this book. We started by presenting the open provenance tracking framework and asked model provenance tracking questions that are of interest. We worked on addressing the issues of auto-logging and successfully registered a trained model by loading a trained model from a logged model in MLflow for prediction using the `mlflow.pytorch.load_model` API. We also experimented on how to directly use MLflow's `log_metrics`, `log_params`, and `log_model` APIs without auto-logging, which gives us more control and flexibility over how we can log additional or customized metrics and parameters. We were able to answer many of the provenance questions by performing model provenance tracking, as well as by providing a couple of the questions that require further study of using MLflow to track multi-step model pipelines and their deployment.

We will continue our learning journey in the next chapter and learn how to perform code and data tracking using MLflow, which will give us additional power to answer data and code-related provenance questions.

## Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- MLflow Docker setup reference: <https://github.com/sachua/mlflow-docker-compose>
- MLflow PyTorch autologging implementation: [https://github.com/mlflow/mlflow/blob/master/mlflow/pytorch/\\_pytorch\\_autolog.py](https://github.com/mlflow/mlflow/blob/master/mlflow/pytorch/_pytorch_autolog.py)
- MLflow PyTorch model logging, loading, and registry documentation: [https://www.mlflow.org/docs/latest/python\\_api/mlflow.pytorch.html](https://www.mlflow.org/docs/latest/python_api/mlflow.pytorch.html)
- MLflow parameters and metrics logging documentation: [https://www.mlflow.org/docs/latest/python\\_api/mlflow.html](https://www.mlflow.org/docs/latest/python_api/mlflow.html)
- MLflow model registry documentation: <https://www.mlflow.org/docs/latest/model-registry.html>
- Digging into big provenance (with SPADE): <https://queue.acm.org/detail.cfm?id=3476885>
- How to utilize torchmetrics and lightning-flash: <https://www.exxactcorp.com/blog/Deep-Learning/advanced-pytorch-lightning-using-torchmetrics-and-lightning-flash>
- Why are precision, recall, and F1 score equal when using micro averaging in a multi-class problem? <https://simonhessner.de/why-are-precision-recall-and-f1-score-equal-when-using-micro-averaging-in-a-multi-class-problem/>



# 4

# Tracking Code and Data Versioning

DL models are not just models – they are intimately tied to the code that trains and tests the model and the data that's used for training and testing. If we don't track the code and data that's used for the model, it is impossible to reproduce the model or improve it. Furthermore, there have been recent industry-wide awakenings and paradigm shifts toward a **data-centric AI** (<https://www.forbes.com/sites/gilpress/2021/06/16/andrew-ng-launches-a-campaign-for-data-centric-ai/?sh=5cbacdc574f5>), where the importance of data is being lifted to a first-class artifact in building ML and, especially, DL models. Due to this, in this chapter, we will learn how to track code and data versioning using MLflow. We will learn about the different ways we can track code and pipeline versioning and how to use Delta Lake for data versioning. By the end of this chapter, you will be able to understand and implement tracking techniques for both code and data with MLflow.

In this chapter, we're going to cover the following main topics:

- Tracking notebook and pipeline versioning
- Tracking locally, privately built Python libraries
- Tracking data versioning in Delta Lake

## Technical requirements

The following are the technical requirements for this chapter:

- VS Code with the Jupyter Notebook extension: <https://github.com/microsoft/vscode-jupyter/wiki/Setting-Up-Run-by-Line-and-Debugging-for-Notebooks>.
- The code for this chapter, which can be found in this book's GitHub repository: <https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/tree/main/chapter04>.
- Access to a Databricks instance so that you can learn how to use Delta Lake to enable versioned data access.

## Tracking notebook and pipeline versioning

Data scientists usually start by experimenting with Python notebooks offline, where interactive execution is a key benefit. Python notebooks have come a long way since the days of **Jupyter** notebooks (<https://jupyter-notebook.readthedocs.io/en/stable/>). The success and popularity of Jupyter notebooks are undeniable. However, there are limitations when it comes to using version control for Jupyter notebooks since Jupyter notebooks are stored as JSON data with mixed output and code. This is especially difficult if we're trying to track code using MLflow as we're only using Jupyter's native format, whose file extension is `.ipynb`. You may not be able to see the exact Git hash in the MLflow tracking server for each run using a Jupyter notebook either. There are a lot of interesting debates on whether or when a Jupyter notebook should be used, especially in a production environment (see a discussion here: <https://medium.com/mlops-community/jupyter-notebooks-in-production-4e0d38803251>). There are multiple reasons why we shouldn't use Jupyter notebooks in a production environment, especially when we need reproducibility in an end-to-end pipeline fashion, where unit testing, proper code versioning, and dependency management could be difficult with a lot of notebooks. There are some early innovations around scheduling, parameterizing, and executing Jupyter notebooks in a workflow fashion using the open source tool **papermill** by Netflix (<https://papermill.readthedocs.io/en/latest/index.html>). However, a recent innovation by Databricks and VS Code makes notebooks much easier to be version controlled and integrated with MLflow. Let's look at the notebook characteristics that were introduced by these two tools:

- **Interactive execution:** Both Databricks's notebooks and VS Code's notebooks can run the same way as traditional Jupyter notebooks, in a cell-by-cell execution mode. By doing this, you can immediately see the output of the results.

- **File format:** Both Databricks's notebooks and VS Code's notebooks are stored as plain-old Python code with a `.py` file extension. This allows all the regular Python code linting (code format and style checking) to be applied to a notebook.
- **Special symbols for rendering code cells and Mark down cells:** Both Databricks and VS Code leverage some special symbols to render Python files as interactive notebooks. In Databricks, the special symbols to delineate code into different executable cells are as follows:

```
# COMMAND -----

import mlflow
import torch
from flash.core.data.utils import download_data
from flash.text import TextClassificationData,
TextClassifier
import torchmetrics
```

The code below the special `COMMAND` line will be rendered as an executable cell in the Databricks web UI portal, as follows:



Figure 4.1 – Databricks executable cell

To execute the code in this cell, you can just click **Run Cell** via the top-right drop-down menu.

To add a large chunk of text to describe and comment on the code in Databricks (also known as Markdown cells), you can use the `# MAGIC` symbol at the beginning of the line, as follows:

```
# MAGIC %md
# MAGIC ##### Notebooks for fine-tuning a pretrained
language model to do text-based sentiment classification
```

This is then rendered in the Databricks notebook as a Markdown comment cell, as follows:

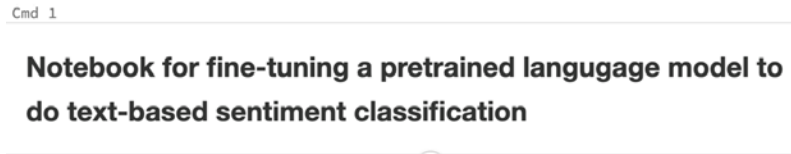


Figure 4.2 – Databricks Markdown text cell

In VS Code, a slightly different set of symbols is used for these two types of cells. For a code cell, the `# %%` symbols are used at the beginning of the cell block:

```
# %%
download_data("https://pl-flash-data.s3.amazonaws.com/
imdb.zip", "./data/")
datamodule = TextClassificationData.from_csv(
    input_fields="review",
    target_fields="sentiment",
    train_file="data/imdb/train.csv",
    val_file="data/imdb/valid.csv",
    test_file="data/imdb/test.csv"
)
```

This is then rendered in VS Code's editor, as follows:

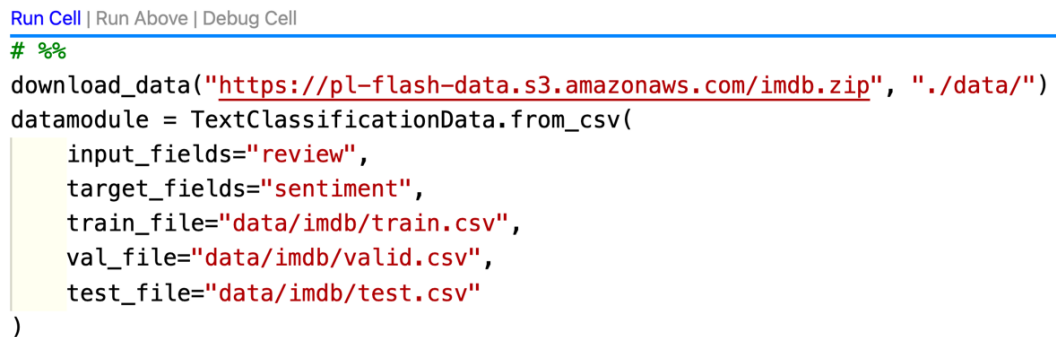


Figure 4.3 – VS Code code cell

As you can see, there is a **Run Cell** button before the block of code that you can click to run the code block interactively. If you click the **Run Cell** button, the code block will start executing in the side panel of the editor window, as shown here:

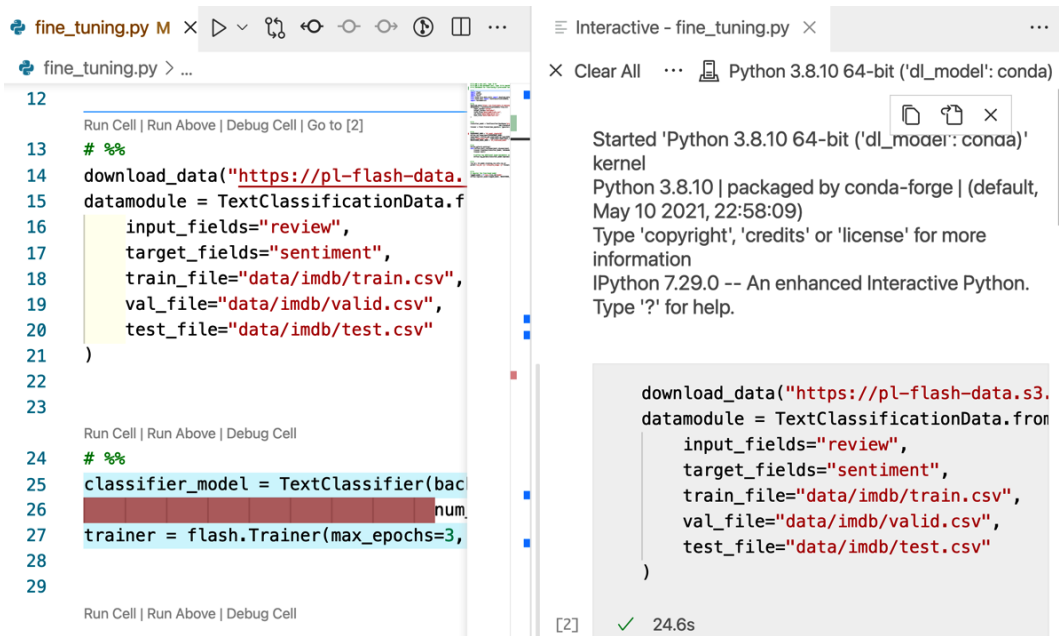


Figure 4.4 – Running code interactively in VS Code

To add a Markdown cell that contains comments, add the following to the beginning of the line, as well as the necessary symbols:

```
# %% Notebook for fine-tuning a pretrained language model
and sentiment classification
```

This will ensure that the text is not an executable code block in VS Code.

Given the advantages of Databricks and VS Code notebooks, we suggest using either for version tracking. We can use GitHub to track the versioning of either type of notebook since they use a regular Python file format.

#### Two Ways to Use Databricks Notebook Version Control

For a managed Databricks instance, a notebook version can be tracked in two ways: by looking at the revision history on the side panel of the notebook on the Databricks web UI, or by linking to a remote GitHub repository. Detailed descriptions are available in the Databricks notebook documentation: <https://docs.databricks.com/notebooks/notebooks-use.html#version-control>.



While the Databricks web portal provides excellent support for notebook version control and integration with MLflow experimentation tracking (see this chapter's callout boxes on *Two Ways to Use Databricks Notebook Version Control* and *Two Types of MLflow Experiments in Databricks Notebooks*), there is one major drawback of writing code in the Databricks notebook web UI. This is because the web UI is not a typical **integrated development environment (IDE)** compared to VS Code, where code style and formatting tools such as **flake8** (<https://flake8.pycqa.org/en/latest/>) and **autopep8** (<https://pypi.org/project/autopep8/>) can easily be enforced. This can have a major impact on code quality and maintainability. Thus, it is highly recommended that you use VS Code to author notebook code (either a Databricks notebook or a VS Code notebook).

#### Two Types of MLflow Experiments in Databricks Notebooks

For a managed Databricks web portal instance, there are two types of MLflow experiments you can perform: workspace and notebook experiments. A workspace experiment is mainly for a shared experiment folder that is not tied to a single notebook. Remote code execution can write to a workspace experiment folder if needed. On the other hand, a notebook scope experiment is tied to a specific notebook and can be found directly on one of the top-right menu items called **Experiment** in the notebook page on the Databricks web portal. For more details, please look at the Databricks documentation website: <https://docs.databricks.com/applications/mlflow/tracking.html#experiments>.

Using this chapter's VS Code notebook, `fine_tuning.py`, which can be found in this chapter's GitHub repository ([https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter04/notebooks/fine\\_tuning.py](https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter04/notebooks/fine_tuning.py)), you will be able to run it interactively in the VS Code editor and log the experiment in the MLflow Docker server that we set up in *Chapter 3, Tracking Models, Parameters, and Metrics*. As a reminder, note that to run this notebook in VS Code successfully, you will need to set up your virtual environment, called `dl_model`, as described in the `README.md` file in this chapter's GitHub repository. It consists of the following three steps:

```
conda create -n dl_model python==3.8.10
conda activate dl_model
pip install -r requirements.txt
```

If you run this notebook cell-by-cell from beginning to end, your experiment page will look as follows:

The screenshot shows the MLflow Experiments page for an experiment named 'chapter04'. The breadcrumb path is 'dl\_model\_chapter04 > chapter04'. The experiment details are as follows:

Date: 2021-10-31 20:48:59	Source: <code>ipykernel_launcher.py</code>	User: yongliu
Duration: 8.5min	Status: FINISHED	Lifecycle Stage: active

Figure 4.5 – Logged experiment page after running a VS Code notebook interactively

You may immediately notice a problem in the preceding screenshot – **Source: `ipykernel_launcher.py`**. This is not the source code file we just ran; that is, the `fine_tuning.py` file. This is because VS Code notebooks are not natively integrated into the MLflow tracking server for source file tracking; it can only show the **ipykernel** (<https://pypi.org/project/ipykernel/>) that VS Code uses to execute a VS Code notebook (<https://github.com/microsoft/vscode-jupyter>). Unfortunately, this is a limitation that, at the time of writing, cannot be addressed by running VS Code notebooks *interactively* for experiment code tracking. Databricks notebooks running inside a hosted Databricks web UI have no such problem as they have native integration with the MLflow tracking server that's bundled in the Databricks web portal.

However, since the VS Code notebooks are just Python code, we can run the notebooks in the command line *non-interactively*, as follows:

```
python fine_tuning.py
```

This will log the actual source code's filename and the Git commit hash in the MLflow experiment page without any issues, as shown here:

The screenshot shows the MLflow Experiments page for an experiment named 'chapter04'. The breadcrumb path is 'dl\_model\_chapter04 > chapter04'. The experiment details are as follows:

Date: 2021-11-06 16:56:17	Source: <code>fine_tuning.py</code>	Git Commit: 661ffeda5ae53cff3623f2fcc8227d822e877e2d
User: yongliu	Duration: 7.0min	Status: FINISHED
Lifecycle Stage: active		

Figure 4.6 – Logged experiment page after running a VS Code notebook in the command line

The preceding screenshot shows the correct source filename (**Source:** `fine_tuning.py`) and the correct git commit hash (**661ffeda5ae53cff3623f2fcc8227d822e877e2d**). This workaround does not require us to change the notebook's code and could be very useful if our initial interactive notebook debugging is done and we want to get a complete run of the notebook, along with proper code version tracking in the MLflow tracking server. Note that all the other parameters, metrics, and models are tracked properly, regardless of whether we run the notebook interactively.

## Pipeline tracking

Having discussed notebook code tracking (version and filename), let's turn to the topic of pipeline tracking. Before we discuss pipeline tracking, however, we will discuss the definition of a pipeline in the ML/DL life cycle. Conceptually, a pipeline is a multi-step data processing and task workflow. However, the implementation of such a data/task workflow can be quite different. A pipeline can be defined as a first-class Python API in some ML packages. The two most well-known pipeline APIs are as follows:

- `sklearn.pipeline.Pipeline` (<https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html>): This is widely used for building tightly integrated multi-step pipelines for classical machine learning or data **extract, transform, and load (ETL)** pipelines using **pandas DataFrames** (<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html>).
- `pyspark.ml.Pipeline` (<https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.Pipeline.html>): This is a PySpark version for building simple and tightly integrated multi-step pipelines for machine learning or data ETL pipelines using **Spark DataFrames** (<https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.DataFrame.html>).

However, when we're building a DL model pipeline, we need to use multiple different Python packages at different steps of the pipeline, so a one-size-fits-all approach using a single pipeline API doesn't usually work. In addition, neither of the aforementioned pipeline APIs have native support for the current popular DL packages, such as **Huggingface** or **PyTorch-Lightning**, which require additional integration work. Although some open source DL pipeline APIs exist such as **Neuraxle** (<https://github.com/Neuraxio/Neuraxle>), which tries to provide a sklearn-like pipeline interface and framework, it is not widely used. Furthermore, using these API-based pipelines means that you'll be locked in when you need to add more steps to the pipeline, which could reduce your flexibility to extend or evolve a DL pipeline when new requirements arise.

In this book, we will take a different approach to define and build a DL pipeline that's based on MLflow's **MLproject** (<https://www.mlflow.org/docs/latest/projects.html#mlproject-file>) structure. This will give you the most flexibility to build a multi-step pipeline that can be tracked using MLflow. At the same time, for each step, you will be allowed to use the most appropriate DL or data processing packages without being locked in. Let's walk through this by breaking the single file-based Python notebook, `fine_tuning.py`, into a multiple-step pipeline. This pipeline can be visualized as a three-step flow diagram, as shown here:

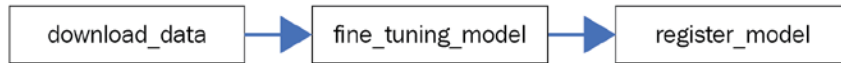


Figure 4.7 – A three-step DL pipeline

This three-step flow is as follows:

1. Download the data to a local execution environment
2. Fine-tune the model
3. Register the model

These modular steps may seem to be overkill for our current example, but the power of having a distinctive functional step is evident when more complexities are involved, or when changes are needed at each step. Each step can be modified without them affecting the other steps if we define the parameters that need to be passed between them. Each step is a standalone Python file that can be executed independently with a set of input parameters. There will be a main pipeline Python file that can run the whole pipeline or a sub-section of the pipeline's steps. In the `MLproject` file, which is a standard YAML file without the file extension, we can define four entry points (`main`, `download_data`, `fine_tuning_model`, and `register_model`), their required input parameters, their types and default values, and the command line to execute each entry point. In our example, these entry points will be provided in a Python command-line execution command. However, you can invoke any kind of execution, such as a batch shell script, if needed for any particular steps. For example, the following lines in the `MLproject` file for this chapter (<https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter04/MLproject>) describe the name of the project, the `conda` environment definition filename, and the main entry point:

```

name: dl_model_chapter04
conda_env: conda.yaml
entry_points:
  main:
  
```

```
parameters:
  pipeline_steps:
    description: Comma-separated list of dl pipeline steps
to execute
    type: str
    default: all
  command: "python main.py --steps {pipeline_steps}"
```

Here, the name of the project is `dl_model_chapter04`. `conda_env` refers to a local conda environment's YAML definition file, `conda.yaml`, which is located in the same directory as the `MLproject` file. The `entry_points` section lists the first entry point, called `main`. In the `parameters` section, there is one parameter called `pipeline_steps`, which allows the user to define a comma-separated list of DL pipeline steps to execute. This parameter is of the `str` type and its default value is `all`, which means that all the pipeline steps will run. Lastly, the `command` section lists how to execute this step in the command line.

The rest of the `MLproject` file defines the other three pipeline step entry points by following the same syntactic convention as the `main` entry point. For example, the following lines in the same `MLproject` file define the entry point of `download_data`:

```
download_data:
  parameters:
    download_url:
      description: a url to download the data for fine tuning
a text sentiment classifier
      type: str
      default: https://pl-flash-data.s3.amazonaws.com/imdb.
zip
    local_folder:
      description: a local folder to store the downloaded
data
      type: str
      default: ./data
    pipeline_run_name:
      description: an mlflow run name
      type: str
      default: chapter04
  command:
```

```
"python pipeline/download_data.py --download_url
{download_url} --local_folder {local_folder} \
--pipeline_run_name {pipeline_run_name}"
```

The `download_data` section, similar to the main entry point, also defines the list of parameters, types, and default values, as well as the command line to execute this step. We can define the rest of the steps in the same manner as we did in the `MLproject` file that we just checked out from this book's GitHub repository. For more details, take a look at the full content of that `MLproject` file.

After defining the `MLproject` file, it becomes clear that we have defined a multi-step pipeline in a declarative way. This is like a specification for the pipeline that says each step's name, what input parameters it expects, and how to execute them. Now, the next step is to implement the Python function to execute each step of the pipeline. So, let's look at the core implementation of the main entry point's Python function, which is called `main.py`. The following lines of code (not the entire Python code in `main.py`) illustrate the core component of implementing the entire pipeline with just one step in the pipeline (`download_data`):

```
@click.command()
@click.option("--steps", default="all", type=str)
def run_pipeline(steps):
    with mlflow.start_run(run_name='pipeline', nested=True) as
        active_run:
            download_run = mlflow.run(".", "download_data",
                parameters={})

if __name__ == "__main__":
    run_pipeline()
```

This main function snippet contains a `run_pipeline` function, which will be run when the `main.py` file is executed in the command line. There is a parameter called `steps`, which will be passed to this function when it's provided. In this example, we are using the `click` Python package (<https://click.palletsprojects.com/en/8.0.x/>) to parse command-line arguments. The `run_pipeline` function starts an MLflow experiment run by calling `mlflow.start_run` and passing two parameters (`run_name` and `nested`). We have used `run_name` before – it's the descriptive phrase for this run. However, the `nested` parameter is new, which means that this is a parent experiment run. This parent experiment run contains some child experiment runs that will be hierarchically tracked in MLflow. Each parent run can contain one or more child runs. In the example code, this contains one step of the pipeline run, called `download_data`, which is invoked by calling `mlflow.run`. This is the key MLflow function to invoke an MLproject's entry point programmatically. Once `download_data` has been invoked and the run has finished, the parent run will also finish, thus concluding the pipeline's run.

#### Two Ways to Execute an MLproject's Entry Point

There are two ways to execute an MLproject's entry point. First, you can use MLflow's Python API, known as `mlflow.run` ([https://www.mlflow.org/docs/latest/python\\_api/mlflow.projects.html#mlflow.projects.run](https://www.mlflow.org/docs/latest/python_api/mlflow.projects.html#mlflow.projects.run)). Alternatively, you can use the MLflow's command-line interface tool, called `mlflow run`, which can be called in a command-line shell environment to execute any entry point directly (<https://www.mlflow.org/docs/latest/cli.html#mlflow-run>).

Now, let's learn how to implement each step in the pipeline generically. For each pipeline step, we put the Python files in a `pipeline` folder. In this example, we have three files: `download_data.py`, `fine_tuning_model.py`, and `register_model.py`. Thus, the relevant files for successfully building an MLflow supported pipeline project are as follows:

```
MLproject
conda.yaml
main.py
pipeline/download_data.py
pipeline/fine_tuning_model.py
pipeline/register_model.py
```

For the implementation of each pipeline step, we can use the following Python function templates. A placeholder section is reserved for implementing the actual pipeline step logic:

```
import click
import mlflow

@click.command()
@click.option("input")
def task(input):
    with mlflow.start_run() as mlrn:
        # Implement pipeline step logic here
        mlflow.log_parameter('parameter', parameter)
        mlflow.set_tag('pipeline_step', __file__)
        mlflow.log_artifacts(artifacts, artifact_path="data")

if __name__ == '__main__':
    task()
```

This template allows us to standardize the way we implement the pipeline step task. The main idea here is that for each pipeline step task, it needs to start with `mlflow.start_run` to launch an MLflow experiment run. Once we've implemented specific execution logic in the function, we need to log some parameters using `mlflow.log_parameter`, or some artifacts in the artifact store using `mlflow.log_artifacts`, that can be passed to and used by the next step of the pipeline. This is called **pipeline chaining**, and it allows multiple steps of a single pipeline or even different pipelines to share data and artifacts. We also want to set a tag to indicate which step is executed using `mlflow.set_tag`.

For example, in the `download_data.py` step, the core implementation is as follows:

```
import click
import mlflow
from flash.core.data.utils import download_data

@click.command()
@click.option("--download_url")
@click.option("--local_folder")
@click.option("--pipeline_run_name")
def task(download_url, local_folder, pipeline_run_name):
```



```
with mlflow.start_run(run_name=pipeline_run_name) as mrun:
    download_data(download_url, local_folder)
    mlflow.log_param("download_url", download_url)
    mlflow.log_param("local_folder", local_folder)
    mlflow.set_tag('pipeline_step', __file__)
    mlflow.log_artifacts(local_folder, artifact_
path="data")

if __name__ == '__main__':
    task()
```

In this `download_data.py` implementation, the task is to download the data for model building from a remote URL to a local folder (`download_data(download_url, local_folder)`). Once we've done this, we will log a few parameters, such as `download_url` and `local_folder`. We can also log the newly downloaded data into the MLflow artifact store using `mlflow.log_artifacts`. For this example, this may not seem necessary since we only want to execute the next step in a local development environment. However, for a more realistic scenario in a distributed execution environment where each step could be run in different execution environments, this is very desirable since we only need to pass the artifact URL path to the next step of the pipeline to use; we don't need to know how and where the previous step was executed. In this example, when the `mlflow.log_artifacts(local_folder, artifact_path="data")` statement is called, the downloaded data folder is uploaded to the MLflow artifact store. However, we will not use this artifact path for the downstream pipeline step in this chapter. We will explore how we use this kind of artifact store to pass artifacts to the next step in the pipeline later in this book. Here, we will use the log parameters to pass the downloaded data path to the next step of the pipeline (`mlflow.log_param("local_folder", local_folder)`). So, let's look at how we can do that by extending `main.py` so that it includes the next step, which is the `fine_tuning_model` entry point, as follows:

```
with mlflow.start_run(run_name='pipeline', nested=True)
as active_run:
    download_run = mlflow.run(".", "download_data",
parameters={})
    download_run = mlflow.tracking.MlflowClient().get_
run(download_run.run_id)
    file_path_uri = download_run.data.params['local_
folder']
```

```
fine_tuning_run = mlflow.run(".", "fine_tuning_
model", parameters={"data_path": file_path_uri})
```

We use `mlflow.tracking.MlflowClient().get_run` to get the `download_run` MLflow run object and then use `download_run.data.params` to get `file_path_uri` (in this case, it is just a local folder path). This is then passed to the next `mlflow.run`, which is `fine_tuning_run`, as a key-value parameter (`parameters={"data_path": file_path_uri}`). This way, the `fine_tuning_run` pipeline step can use this parameter to prefix its data source path. This is a very simplified scenario to illustrate how we can pass data from one step to the next. Using the `mlflow.tracking.MlflowClient()` API, which is provided by MLflow ([https://www.mlflow.org/docs/latest/python\\_api/mlflow.tracking.html](https://www.mlflow.org/docs/latest/python_api/mlflow.tracking.html)), makes accessing a run's information (parameters, metrics, and artifacts) straightforward.

We can also extend the `main.py` file with the third step of the pipeline by adding the `register_model` step. This time, we need the logged model URI to register a trained model, which depends on `run_id` of the `fine_tuning_model` step. So, in the `fine_tuning_model` step, we need to get the `run_id` property of `fine_tuning_model` run and then pass it through the input parameter for the `register_model` run, as follows:

```
fine_tuning_run_id = fine_tuning_run.run_id
register_model_run = mlflow.run(".", "register_model",
parameters={"mlflow_run_id": fine_tuning_run_id})
```

Now, the `register_model` step can use `fine_tuning_run_id` to locate the logged model. The core implementation of the `register_model` step is as follows:

```
with mlflow.start_run() as mrun:
    logged_model = f'runs:{mlflow_run_id}/model'
    mlflow.register_model(logged_model, registered_model_
name)
```

This will register a fine-tuned model at the URI defined by the `logged_model` variable to an MLflow model registry.

If you have followed these steps, then you should have a working pipeline that can be tracked by MLflow from end to end. As a reminder, a prerequisite is to have the local full-fledged MLflow server set up, as shown in *Chapter 3, Tracking Models, Parameters, and Metrics*. You should have set up the virtual environment, `dl_model`, in the previous section. To test this pipeline, check out this chapter's GitHub repository at <https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/tree/main/chapter04> and run the following command:

```
python main.py
```

This will run the entire three-step pipeline and log the pipeline's `run_id` (which is the parent run) and each step's run as the child runs in the MLflow tracking server. The last few lines of the console screen's output will display something as follows when it has finished running (you will see lots of outputs on the screen when you run the pipeline):

```
'pipeline_step': 'pipeline/fine_tuning_model.py'}>, info=<RunInfo: artifact_uri='s3://mlflow/2/5ba38e059695485396e709b809e9bb8d/artifacts', end_time=1637536183598, experiment_id='2', lifecycle_stage='active', run_id='5ba38e059695485396e709b809e9bb8d', run_uuid='5ba38e059695485396e709b809e9bb8d', start_time=1637535738320, status='FINISHED', user_id='yongliu'>>
2021-11-21 15:09:47,042 finished mlflow pipeline run with a run_id = f8f21fdf8fff4fd6a400eeb403b776c8
```

Figure 4.8 – Console output of running the pipeline with MLflow `run_ids`

This shows the pipeline's `run_id`, which is `f8f21fdf8fff4fd6a400eeb403b776c8`; the last step is the `run_id` property of `fine_tuning_model`, which is `5ba38e059695485396e709b809e9bb8d`. If we go to the MLflow tracking server's UI web page by clicking on <http://localhost>, we should be able to see the following nested experiment runs in the `dl_model_chapter04` experiment folder, as follows:

<input type="checkbox"/> ↓ Start Time	Duration	Run Name	Source	Version
<input type="checkbox"/> 7 minutes ago	7.6min	pipeline	main.py	d0d416
<input type="checkbox"/> 19 seconds	3.3s	-	Practical-Deep-Learning-at-Scale-with-MLFlow#chapter04:register_model	d0d416
<input type="checkbox"/> 7 minutes ago	7.4min	-	Practical-Deep-Learning-at-Scale-with-MLFlow#chapter04:fine_tuning_mode	d0d416
<input type="checkbox"/> 7 minutes ago	7.7s	-	Practical-Deep-Learning-at-Scale-with-MLFlow#chapter04:download_data	d0d416

Figure 4.9 – A pipeline being run with nested three-step child runs in the MLflow tracking server

The preceding screenshot shows the pipeline run, along with the source `main.py` file and the nested run of the three steps of the pipeline. Each step has a corresponding entry point name defined in `MLproject` with a GitHub commit hash code version of `d0d416`. If you click on the `register_model` run page, you will see the following information:

dl\_model\_chapter04 &gt; Run 4b311303e0bf4d6f8b2674ca8a79d04b

**Run 4b311303e0bf4d6f8b2674ca8a79d04b**

<b>Date:</b> 2021-11-21 15:09:43	<b>Source:</b> [📄] Practical-Deep-Learning-at-Scale-with-MLFlow#chapter04:register_model	<b>Git Commit:</b> d0d416ae68d78d92973c1bca4e774fda8
<b>Entry Point:</b> register_model	<b>User:</b> yongliu	<b>Duration:</b> 3.3s
<b>Status:</b> FINISHED	<b>Lifecycle Stage:</b> active	<b>Parent Run:</b> <a href="#">f8f21fdf8fff4fd6a400eeb403b776c8</a>

▼ **Run Command**

```
mlflow run file:///Users/yongliu/opensource_code/Practical-Deep-Learning-at-Scale-with-MLFlow#chapter04 -v d0d416ae68d78d92973c1bca4e774fda835a20bf -e register_model -b local -P mlflow_run_id=5ba38e059695485396e709b809e9bb8d -P registered_model_name=dl_finetuned_model
```

Figure 4.10 – Entry point register\_model's run page on the MLflow tracking server

The preceding screenshot shows not only some of the familiar information we have seen already, but also some new information such as **Parent Run: f8f21fdf8fff4fd6a400eeb403b776c8**, **Entry Point: register\_model**, and a fully populated **Run Command** cell that's automatically generated by MLflow. **Run Command** contains the file's location URL (a string that starts with `file:///`), the GitHub hash code version, the entry point (`-e register_model`), the execution environment, which is a local dev environment (`-b local`), and the expected parameters for the `register_model` function (`-P`). We will learn how to use MLflow's `MLproject` to run commands to execute tasks remotely later in this book. Here, we just need to understand that the source code is referred to through the entry point (`register_model`), not the filename itself, since the reference is declared as an entry point in the `MLproject` file.

If you saw the output shown in *Figure 4.9* and *Figure 4.10* in your MLflow tracking server, then it's time to celebrate – you have successfully executed a multi-step DL pipeline using MLflow!

In summary, to track a multi-step DL pipeline in MLflow, we can use `MLproject` to define entry points for each pipeline step and a main pipeline entry point. In the main pipeline function, we implement methods so that data can be passed between pipeline steps. Each pipeline step then uses the data that's been shared, as well as other input parameters, to execute a specific task. Both the main pipeline-level function and each step of the pipeline are tracked using the MLflow tracking server, which produces a `parent_run_id` to track the main pipeline run and multiple MLflow nested runs to track each pipeline's step. We introduced a template for each pipeline step to implement this task in a standard way. We also explored the powerful pipeline chaining that's done through MLflow's `run` parameter and artifact store to learn how to pass data between pipeline steps.

Now that you know how to track notebooks and pipelines, let's learn how to track Python libraries.

## Tracking locally, privately built Python libraries

Now, let's turn our attention to tracking locally, privately built Python libraries. For publicly released Python libraries, we can explicitly specify their released version, which is published in PyPI, in a requirements file or a `conda.yaml` file. For example, this chapter's `conda.yaml` file (<https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter04/conda.yaml>) defines the Python version and provides a reference to a requirements file, as follows:

```
name: dl_model
channels:
  - conda-forge
dependencies:
  - python=3.8.10
  - pip
  - pip:
    - -r requirements.txt
```

The Python version is defined as `3.8.10` and is being enforced. This `conda.yaml` file also refers to a `requirements.txt` file, which contains the following versioned Python packages as a `requirements.txt` file, which is located in the same directory as the `conda.yaml` file:

```
ipykernel==6.4.1
lightning-flash[all]==0.5.0
mlflow==1.20.2
transformers==4.9.2
boto3==1.19.7
pytorch-lightning==1.4.9
datasets==1.9.0
click==8.0.3
```

As we can see, all these packages are being tracked explicitly using their published PyPI (<https://pypi.org/>) version number. When you run the MLflow MLproject, MLflow will use the `conda.yaml` file and the referenced `requirements.txt` file to create a conda virtual environment dynamically. This ensures that the execution environment is reproducible and that all the DL model pipelines can be run successfully. You may have noticed that such a virtual environment was created for you the first time you ran the previous section's MLflow pipeline project. You can do this again by running the following command:

```
conda env list
```

This will produce a list of conda virtual environments in your current machine. You should be able to find a virtual environment starting with a `mlflow-` prefix, followed by a long string of alphanumeric characters, as follows:

```
mlflow-95353930ddb7b60101df80a5d64ef8bf6204a808
```

This is the virtual environment that's created by MLflow dynamically, which follows the dependencies that are specified in `conda.yaml` and `requirements.txt`. Subsequently, when you log the fine-tuned model, `conda.yaml` and `requirements.txt` will be automatically logged in the MLflow artifact store, as follows:

▼ Artifacts

The screenshot shows the MLflow artifact store interface. On the left, a file explorer shows a directory structure: `model` (expanded) contains `data`, `MLmodel`, `conda.yaml` (selected), `requirements.txt`, and `model_summary.txt`. On the right, the details for the selected `conda.yaml` file are shown. The full path is `s3://mlflow/2/5ba38e059695485396e709b809e9bb8d/artifacts/model/conda.yaml` and its size is 437B. The file content is as follows:

```
channels:
- conda-forge
dependencies:
- python=3.8.10
- pip
- pip:
  - mlflow
  - aiohttp==3.8.1
  - av==8.0.3
  - boto3==1.19.7
  - cloudpickle==2.0.0
  - datasets==1.9.0
  - hydra-core==1.1.1
  - ipython==7.29.0
  - lightning-flash==0.5.0
```

Figure 4.11 – Python packages are being logged and tracked in the MLflow artifact store

As we can see, the `conda .yaml` file was automatically expanded to include the content of `requirements.txt`, as well as other dependencies that conda decides to include.

For privately built Python packages, which means the Python packages that are not published to PyPI for public consumption and references, the recommended way to include such a Python package is by using `git+ssh`. Let's assume that you have a privately built project called `cool-dl-utils`, that the organization you work for is called `cool_org`, and that your project's repository has been set up in GitHub. If you want to include this project's Python package in the `requirements` file, you need to make sure that you add your public key to your GitHub settings. If you want to learn how to generate a public key and load it into GitHub, take a look at GitHub's guide at <https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account>. In the `requirements.txt` file, you can add the following line, which will reference a specific GitHub hash (`81218891bbf5a447103884a368a75ffe65b17a44`) and the Python `.egg` package that was built from this private repository (you can also reference a `.whl` package if you wish):

```
cool-dl-utils @ git+ssh://git@github.com/cool_org/cool-dl-  
utils.git@81218891bbf5a447103884a368a75ffe65b17a44#egg=cool-dl-  
utils
```

If you have a numerically released version in your privately built package, you can also directly reference the release number in the `requirements.txt` file, as follows:

```
git+ssh://git@github.com/cool_org/cool-dl-utils.git@2.11.4
```

Here the release number of `cool-dl-utils` is `2.11.4`. This allows MLflow to pull this privately built package into the virtual environment to execute `MLproject`. In this chapter, we don't need to reference any privately built Python packages, but it is worth noting that MLflow can leverage the `git+ssh` approach to do that.

Now, let's learn how to track data versioning.

## Tracking data versioning in Delta Lake

In this section, we'll learn how data is tracked in MLflow. Historically, data management and versioning are usually considered as being different from machine learning and data science. However, the advent of data-centric AI is playing an increasingly important role, particularly in DL. Therefore, it is critical to know what and how data is being used to improve the DL model. In the first data-centric AI competition, which was organized by Andrew Ng in the summer of 2021, the requirements to become a winner were not about changing and tuning a model, but rather improving the dataset of a fixed model (<https://https-deeplearning-ai.github.io/data-centric-comp/>). Here is a quote from the competition's web page:

*"The Data-Centric AI Competition inverts the traditional format and asks you to improve a dataset, given a fixed model. We will provide you with a dataset to improve by applying data-centric techniques such as fixing incorrect labels, adding examples that represent edge cases, applying data augmentation, and so on."*

This paradigm shift highlights the importance of data in deep learning, especially supervised deep learning, where labeled data is important. An implied underlying assumption is that different data will produce different model metrics, even if the same model architecture and parameters are used. This requires us to diligently track the data versioning process so that we know which version of the data is being used to produce the winning model.

There are several emerging frameworks for tracking data versioning in the ML/DL life cycle. One of the early pioneers in this domain is **DVC** (<http://dvc.org>). It uses a set of GitHub-like commands to pull/push data as if they are code. It allows the data to be stored remotely in S3, or Google Drive, among many other popular stores. However, the data that's stored in the remote store becomes hashed and isn't human-readable. This becomes a locked-in problem since the only way to access the data is through the DVC tool and configuration. In addition, it is hard to track how the data and its schema have been changed. While it is possible to integrate MLflow with DVC, its usability and flexibility are not as desirable as we want. Thus, we will not deep dive into this approach in this book. If you are interested in this, we suggest that you utilize the *Versioning data and models in ML projects using DVC and AWS* reference at the end of this chapter to find more details about using DVC.



The recently open sourced and open format-based **Delta Lake** (<https://delta.io/>) is a practical solution for integrated data management and version control in a DL/ML project, especially since MLflow can directly support such integration. This is also the foundational data management layer, called **Lakehouse** (<https://databricks.com/blog/2020/01/30/what-is-a-data-lakehouse.html>), which unifies both data warehouse and streaming data into one data foundation layer. It supports both schema change tracking and data versioning, which is ideal for a DL/ML data use scenario. Delta tables are based on the open standard file format called **Parquet** (<https://parquet.apache.org/>), which is widely supported for large-scale data storage.

#### Delta Table in Databricks

Note that this section assumes that you have access to a Databricks service, which allows you to experiment with the Delta Lake format in the **Databricks File System (DBFS)**. You can get a trial account for the community version by going to the Databricks portal: <https://community.cloud.databricks.com/login.html>.

Note that this section requires you to use **PySpark** to manipulate the data through both reading/writing data from/into storage such as S3. Delta Lake has a capability called **Time Travel** that can automatically version the data. By passing a parameter such as a timestamp or a version number, you can read any historical data for that particular version or timestamp. This makes reproducing and tracking the experiments much easier as the only temporal metadata about the data is the version number or timestamp of the data. There are two ways to query the Delta table:

- `timestampAsOf`: This lets you read the Delta table, as well as read a version that has a specific timestamp. The following code shows how the data can be read using `timestampAsOf`:

```
df = spark.read \
    .format("delta") \
    .option("timestampAsOf", "2020-11-01") \
    .load("/path/to/my/table")
```

- `versionAsOf`: This defines the numerical value of the Delta table's version. You also have the option to read a version that has a specific version, starting with version 0. The following PySpark code reads the data with the `versionAsOf` option defined as version 52:

```
df = spark.read \  
    .format("delta") \  
    .option("versionAsOf", "52") \  
    .load("/path/to/my/table")
```

Having this kind of timestamped or versioned access is a key advantage to tracking any file version using a Delta table. So, let's look at a concrete example of this in MLflow so that we can track the IMDb dataset we have been using.

## An example of tracking data using MLflow

For the IMDb datasets we have been using to fine-tune the sentiment classification model, we will upload these CSV files into Databricks' data store or any S3 bucket that you can access from your Databricks portal. Once you've done that, follow these steps to create a Delta table that supports versioned and timestamped data access:

1. Read the following CSV files into a DataFrame (assuming that you uploaded the `train.csv` file into the `FileStore/imdb/` folder in Databricks):

```
imdb_train_df = spark.read.option('header', True). \  
    csv('dbfs:/FileStore/imdb/train.csv')
```

2. Write the `imdb_train_df` DataFrame in DBFS as a Delta table, as follows:

```
imdb_train_df.write.format('delta').option("mergeSchema", \  
    "true").mode("overwrite").save('/imdb/training.delta')
```

3. Read the `training.delta` file back into memory using the following command:

```
imdb_train_delta = spark.read.format('delta').load('/ \  
    imdb/training.delta')
```

- Now, look at the history of the Delta table via the Databricks UI. You click on the **History** tab once you've read the Delta table from storage into memory:

```
train_delta = spark.read.format('delta').load('/imdb/train.delta')
```

---

imdb\_train\_delta: pyspark.sql.dataframe.DataFrame

chema Details **History**

---

Filter

	version ▲	timestamp ▲	userId ▲	user
1	0	2021-11-22T03:39:22.000+0000	100004	yong

Figure 4.12 – The train\_delta table's history with a version and a timestamp column

The preceding screenshot shows that the version is **0** and that the timestamp is **2021-11-22**. This is the value that we can use to access the versionized data when passing the version number or timestamp to a Spark DataFrame reader.

- Read the versioned imdb/train\_delta file using the following command:

```
train_data_version = spark.read.format("delta").
option("versionAsOf", "0").load('/imdb/train.delta')
```

This will read version 0 of the train.delta file. If we had other versions of this file, we could pass a different version number.

- Read the timestamped imdb/train\_delta file using the following command:

```
train_data_timestamped = spark.read.format("delta").
option("timestampAsOf", "2021-11-22T03:39:22").load('/
imdb/train.delta')
```

This will read the timestamped data. At the time of writing, this is the only timestamp we have, which is fine. If we had more timestamped data, we could pass a different version to it.

- Now, if we need to log this data version in the MLflow tracking experiment run, we can just log the path of the data, the version number, and/or the timestamp using `mlflow.log_parameter()`. This will log these as part of the experiment's parameter key-value list:

```
mlflow.log_parameter('file_path', '/imdb/train.delta')
mlflow.log_parameter('file_version', '0')
```

```
mlflow.log_parameter('file_timestamp', '2021-11-22T03:39:22')
```

The only requirement for using a Delta table is that the data needs to be stored in a form of storage that supports Delta tables, such as Lakehouse, which is supported by Databricks. This is of great value for enterprise ML/DL scenarios since we can track data versioning alongside code and model versioning.

In summary, Delta Lake provides a simple yet powerful way to version data. MLflow can easily log these version numbers and timestamps as part of the experiment's parameter lists to track the data, as well as all the other parameters, metrics, artifacts, code, and models consistently.

## Summary

In this chapter, we took a deep dive into how we can track code and data versions in an MLflow experiment run. We started by reviewing the different types of notebooks: Jupyter notebooks, Databricks notebooks, and VS Code notebooks. We compared them and recommended that VS Code should be used to author a notebook due to its IDE support, as well as its Python styling, autocompletion, and many more rich features.

Then, after reviewing the limitations of existing ML pipeline API frameworks, we discussed how to create a multi-step DL pipeline using MLflow's **MLproject** framework. We showed a step-by-step approach to creating a three-step DL pipeline using MLproject and how to implement a pipeline function to orchestrate the necessary tasks. We also provided a Python implementation template to help you implement each pipeline task. When running a pipeline with MLflow, we can track the entire pipeline's progress with a parent `run_id`, and then use a child `run_id` for each pipeline step. The flexibility to do pipeline chaining and tracking by passing parameters or artifact store locations to the next step can be done using `mlflow.run()` and `mlflow.tracking.MlflowClient()`. We successfully ran the end-to-end three-step pipeline using the MLflow nested run tracking capability. This will also open doors for us to extend the use of MLproject for running different steps in a distributed way in future chapters.

We also learned how to track privately built Python packages using the `git+ssh` approach. We then used the Delta Lake approach to gain versioned and timestamped access to data. This allows data to be tracked in two ways using a version number or a timestamp. MLflow can then log these version numbers or timestamps as a parameter during the MLflow experiment run. Since we are entering the data-centric AI era, being able to track data versioning is critical for reproducibility and time travel.

With that, we've finished learning how to comprehensively track code, data, and models using MLflow. In the next chapter, we will learn how to scale out our DL experiment in a distributed way.

## Further reading

For more information about the topics that were covered in this chapter, take a look at the following resources:

1. MLflow notebook experiment tracking in Databricks: <https://docs.databricks.com/applications/mlflow/tracking.html#create-notebook-experiment>
2. *Building Multistep Workflows*: <https://www.mlflow.org/docs/latest/projects.html#building-multistep-workflows>
3. *End-to-end ML pipelines with MLflow projects*: <https://dztlab.github.io/ml/2020/08/09/mlflow-pipelines/>
4. Installing a privately built Python package: <https://medium.com/@ffreitasalves/pip-installing-a-package-from-a-private-repository-b57b19436f3e>
5. *Versioning data and models in ML projects using DVC and AWS*: <https://medium.com/analytics-vidhya/versioning-data-and-models-in-ml-projects-using-dvc-and-aws-s3-286e664a7209>
6. *Introducing Delta Time Travel for Large Scale Data Lakes*: <https://databricks.com/blog/2019/02/04/introducing-delta-time-travel-for-large-scale-data-lakes.html>
7. *How We Won the First Data-Centric AI Competition: Synaptic-AnN*: <https://www.deeplearning.ai/data-centric-ai-competition-synaptic-ann/>
8. *Reproduce Anything: Machine Learning Meets Data Lakehouse*: <https://databricks.com/blog/2021/04/26/reproduce-anything-machine-learning-meets-data-lakehouse.html>
9. *DATABRICKS COMMUNITY EDITION: A BEGINNER'S GUIDE*: <https://www.topcoder.com/thrive/articles/databricks-community-edition-a-beginners-guide>

# Section 3 – Running Deep Learning Pipelines at Scale

In this section, we will learn how to run **deep learning (DL)** pipelines in different execution environments and perform hyperparameter tuning, or **hyperparameter optimization (HPO)**, at scale. We will start with an overview of the scenarios and requirements for executing DL pipelines in different environments. We will then learn how to use MLflow's **command-line interface (CLI)** to run in four different execution scenarios in a distributed environment. From there on, we will learn how to choose the best HPO framework by comparing **Ray Tune**, **Optuna**, and **HyperOpt** for tuning hyperparameters of a DL pipeline. Finally, we will concentrate on how to implement and run HPO for DL at scale using state-of-the-art HPO frameworks such as Ray Tune and MLflow.

This section comprises the following chapters:

- *Chapter 5, Running DL Pipelines in Different Environments*
- *Chapter 6, Running Hyperparameter Tuning at Scale*



# 5

# Running DL Pipelines in Different Environments

It is critical to have the flexibility of running a **deep learning (DL)** pipeline in different execution environments such as local or remote, on-premises, or in the cloud. This is because, during different stages of the DL development, there may be different constraints or preferences to either improve the velocity of the development or ensure security compliance. For example, it is desirable to do small-scale model experimentation in a local or laptop environment, while for a full hyperparameter tuning, we need to run the model on a cloud-hosted GPU cluster to get a quick turn-around time. Given the diverse execution environments in both hardware and software configurations, it used to be a challenge to achieve this kind of flexibility within a single framework. MLflow provides an easy-to-use framework to run DL pipelines at scale in different environments. We will learn how to do that in this chapter.



In this chapter, we will first learn about the different DL pipeline execution scenarios and their execution environments. We will also learn how to run the different steps of the DL pipeline in different execution environments. Specifically, we will cover the following topics:

- An overview of different execution scenarios and environments
- Running locally with local code
- Running remote code in GitHub locally
- Running local code remotely in the cloud
- Running remotely in the cloud with remote code in GitHub

By the end of this chapter, you will be comfortable setting up the DL pipelines to run either locally or remotely with different execution environments.

## Technical requirements

The following technical requirements are needed for completing the learning in this chapter:

- The code in this chapter can be found at the following GitHub URL: <https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/tree/main/chapter05>.
- Installation of the Databricks **command-line interface (CLI)** tool to access the Databricks platform remote execution of DL pipelines: <https://github.com/databricks/databricks-cli>.
- Access to a Databricks instance (must be the Enterprise version, as the Community version does not support remote execution) for learning how to run DL pipelines remotely on a cluster in Databricks.
- A full-fledged MLflow tracking server when running locally. This MLflow tracking server setup is the same as in previous chapters.

## An overview of different execution scenarios and environments

In our previous chapters, we mainly focused on learning how to track DL pipelines using MLflow's tracking capabilities. Most of our execution environments are in a local environment, such as a local laptop or desktop environment. However, as we already know, the DL full life cycle consists of different stages where we may need to run the DL pipelines either entirely, partially, or as a single step in a different execution environment. Here are two typical examples:

- When accessing data for model training purposes, it is not uncommon to require the data to reside in an enterprise-security and privacy-compliant environment, where both the computation and the storage cannot leave a compliant boundary.
- When training a DL model, it is usually desirable to use a remote GPU cluster to maximize the efficiency of model training, where a local laptop usually does not have the required hardware capability.

Both cases require a carefully defined execution environment that might be needed in one or multiple stages of the DL lifecycle. Note that this is not just a requirement to be flexible when moving from the development stage to a production environment, where the execution hardware and software configuration could be understandably different. It is also a requirement to be able to switch running environments during development stages or in different production environments without making major changes to the DL pipelines.

Here, we classify the different scenarios and execution environments into the following four scenarios, based on the different combinations of the location of the source code of DL pipelines and target execution environments, as shown in the following table:

Scenarios	Target execution environment (local)	Target execution environment (remote)
Source code (local)	DL development (for example, testing small changes locally)	DL development (for example, training a DL model)
Source code (remote in GitHub)	DL development or production regression testing with new data or model	DL development or production (for example, deployment of a new pipeline)

Figure 5.1 – Four different scenarios of DL pipeline source codes and target execution environments

*Figure 5.1* describes how either in development or production environments, we could encounter the possibilities of using either local or remote code to run in a different execution environment. Let's examine them one by one as follows:

- **Local source code running in a local target environment:** This usually happens at the development stage, where modest computing power in a local environment is adequate to support quick prototyping or test runs for small changes in an existing pipeline. This is mostly the scenario we have been using in previous chapters for our MLflow experiments when learning how to track pipelines.
- **Local source code running in a remote target environment:** This usually happens at the development stage or re-training of an existing DL model, where a GPU or other types of hardware accelerators, such as **Tensor Processing Units (TPUs)** or **field-programmable gate arrays (FPGAs)**, are needed to perform computational and data-intensive model training or debugging prior to merging the GitHub repository (using local code change first).
- **Remote source code running in a local target environment:** This usually happens when we don't have any changes in the code but the data has changed, either during the development stage or the production stage. For example, during the DL development stage, we could change the data with newly augmented training data either through some data augmentation techniques (for example, using **AugLy** to augment existing training data: <https://github.com/facebookresearch/AugLy>) or newly annotated training data. During the production deployment step, we often need to run a regression test to evaluate a to-be-deployed DL pipeline against a hold-out regression testing dataset, so that we don't deploy a degraded model if the model performance accuracy metric does not meet the bar. In this case, the hold-out testing dataset is not usually big, so the execution can be done on the deployment server locally instead of launching to a remote cluster in a Databricks server.
- **Remote source code running in a remote target environment:** This can happen in the development stage or production stage, where we want to use a fixed version of the DL pipeline code from GitHub to run in a remote GPU cluster to do model training, hyperparameter tuning, or re-training. Such large-scale execution can be time-consuming, and a remote GPU cluster could be very useful.

Given the four different scenarios, it would be desirable to have a framework to be able to run the same DL pipeline with minimal configuration changes under these conditions. Prior to the arrival of MLflow, it took quite a lot of engineering and manual efforts to support these scenarios. MLflow provides an MLproject framework that supports all these four scenarios through the following three configurable mechanisms:

1. **Entry points:** We can define one or multiple entry points to execute different steps of a DL pipeline. For example, the following is an example to define a main entry point:

```
entry_points:
  main:
    parameters:
      pipeline_steps: { type: str, default: all }
    command: "python main.py -pipeline_steps {pipeline_
steps}"
```

The entry point's name is `main`, which, by default, will be used when executing an MLflow run without specifying an entry point for an MLproject. Under this `main` entry point, there is a list of parameters. We can define the parameter's type and default value using a short syntax, as follows:

```
parameter_name: {type: data_type, default: value}
```

We can also use a long syntax, as follows:

```
parameter_name:
  type: data_type
  default: value
```

Here, we define only one parameter, called `pipeline_steps`, using the short syntax format with a `str` type and a default value of `all`.

2. **Software and library dependencies:** We can use one conda `.yaml` configuration file or a Docker image to define the software and library dependencies that can be used by the MLproject's entry points. Note that a single MLproject can either use a conda `yaml` file or a Docker image, but not both at the same time. Depending on the DL pipeline dependencies, sometimes using a conda `.yaml` file over a Docker image is preferred, since it is much more lightweight and easier to make changes without requiring additional Docker image storage locations and loading a large Docker image into memory in a resource-limited environment. However, a Docker image does sometimes have advantages if there are any Java packages (`.jar`) that are needed at runtime. If there are no such JAR dependencies, then it is preferred to have a conda `.yaml` file to specify the dependencies. Furthermore, as of MLflow version 1.22.0, running Docker-based projects on Databricks is not yet supported by the MLflow command line. If there are indeed any Java package dependencies, they can be installed using **init scripts** (for example, see the official documentation at <https://docs.databricks.com/clusters/init-scripts.html#example-install-postgresql-jdbc-driver>). Thus, we will use conda `.yaml` configuration files to define execution environment dependencies in this book.
3. **Hardware dependencies:** We can use a cluster configuration JSON file to define the execution target backend environment, be it a GPU, CPU, or other types of clusters. This is only needed when the target backend execution environment is non-local, either in a Databricks server or a **Kubernetes (K8s)** cluster.

Previously, we learned how to use MLproject to create a multiple-step DL pipeline running in a local environment in *Chapter 4, Tracking Code and Data Versioning*, for tracking purposes. Now, we are going to learn how to use MLproject for supporting the different running scenarios outlined previously.

## Running locally with local code

Let's start with the first running scenario using the same **Natural Language Processing (NLP)** text sentiment classification example as the driving use case. You are advised to check out the following version of the source code from the GitHub location to follow along with the steps and learnings: <https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/tree/26119e984e52dadd04b99e6f7e95f8dda8b59238/chapter05>. Note that this requires a specific Git hash committed version, as shown in the URL path. That means we are asking you to check out a specific committed version, not the main branch.

Let's start with the DL pipeline that downloads the review data to local storage as a first execution exercise. Once you check out this chapter's code, you can type the following command line to execute the DL pipeline's first step:

```
mlflow run . --experiment-name='dl_model_chapter05' -P
pipeline_steps='download_data'
```

If we don't specify an entry point, it defaults to `main`. In this case, this is our desired behavior since we want to run the `main` entry point to start the parent DL pipeline.

The `dot` means the current local directory. This tells MLflow to use the code in the current directory as the source to execute the project. If this command line runs successfully, you should be able to see the first two lines of output in the console as follows, which also reveal where the target execution environment is:

```
2022/01/01 19:15:37 INFO mlflow.projects.utils: === Created
directory /var/folders/51/whxjy4r92dx18788yp1lycyr0000gp/T/
tmp3qj2kws2 for downloading remote URIs passed to arguments of
type 'path' ===
2022/01/01 19:15:37 INFO mlflow.projects.backend.local: ===
Running command 'source /Users/yongliu/opt/miniconda3/bin/./
etc/profile.d/conda.sh && conda activate mlflow-95353930ddb7b
60101df80a5d64ef8bf6204a808 1>&2 && python main.py --pipeline_
steps download_data' in run with ID 'f7133b916a004c508e227f00d5
34e136' ===
```

Note that the second output line shows `mlflow.projects.backend.local`, which means the target running environment is local. You may wonder where we define the local execution environment in our initial command line. It turns out that by default, the value for the parameter called `--backend` (or `-b`) is `local`. So, if we spell out the default values, the `mlflow run` command line will look like the following:

```
mlflow run . -e main -b local --experiment-name='dl_model_
chapter05' -P pipeline_steps='download_data'
```

Note that we also need to specify `experiment-name` in the command line or through an environment variable named `MLFLOW_EXPERIMENT_NAME` to define the experiment in which this project will run. Alternatively, you can specify an `experiment-id` parameter, or an environment variable named `MLFLOW_EXPERIMENT_ID`, to define the experiment integer ID that already exists. You only need to define either the ID or the name of the environment, but not both. It is common to define a human-readable experiment name and then query the experiment ID for that experiment in other parts of the code so that they will not be out of sync.

**MLflow Experiment Name or ID for Running an MLproject**

To run an MLproject either using the CLI or the `mlflow.run` Python API, if we don't specify `experiment-name` or `experiment-id` through either an environment variable or a parameter assignment, it will default to the Default MLflow experiment. This is not desirable, as we want to organize our experiments into clearly separated experiments. In addition, once an MLproject starts running, any child runs will not be able to switch to a different experiment name or ID. So, the best practice will be always to specify an experiment name or an ID before launching an MLflow project run.

Once you finish the run, you will see the output as in the following lines:

```
2022-01-01 19:15:48,249 <Run: data=<RunData: metrics={},
params={'download_url': 'https://p1-flash-data.s3.amazonaws.
com/imdb.zip',
'local_folder': './data',
'mlflow_run_id': 'f9f74ebd80f246d58a5f7a3bfb3fc635',
'pipeline_run_name': 'chapter05'}, tags={'mlflow.gitRepoURL':
'git@github.com:PacktPublishing/Practical-Deep-Learning-at-
Scale-with-MLFlow.git',
'mlflow.parentRunId': 'f7133b916a004c508e227f00d534e136',
```

Note that this is a nested MLflow run since we first launch a main entry point that starts the whole pipeline (that's why there is `mlflow.parentRunId`), and then under this pipeline, we run one or multiple steps. Here, the step we run is called `download_data`, which is another entry point defined in the MLproject, but is invoked using the `mlflow.run` Python API, as follows, in the `main.py` file:

```
download_run = mlflow.run(".", "download_data", parameters={})
```

Note that this also specifies which code source to use (`local`, since we specified a `dot`), and by default, a local execution environment. That's why you should be able to see the following lines in the console output:

```
'mlflow.project.backend': 'local',
'mlflow.project.entryPoint': 'download_data',
```

You should also see a few other details of the run parameters for this entry point. The last two lines of the command line output should look like the following:

```
2022-01-01 19:15:48,269 finished mlflow pipeline run with a
run_id = f7133b916a004c508e227f00d534e136
2022/01/01 19:15:48 INFO mlflow.projects: === Run (ID 'f7133b91
6a004c508e227f00d534e136') succeeded ===
```

If you see this, you should feel proud that you have successfully run a pipeline with one step to completion.

While this is something we have done before without knowing some of the details, the next section will allow us to run remote code in a local environment, where you will see the increasing flexibility and power of MLproject.

## Running remote code in GitHub locally

Now, let's see how we run remote code from a GitHub repository on a local execution environment. This allows us to precisely run a specific version that has been checked into the GitHub repository using the commit hash. Let's use the same example as before by running a single `download_data` step of the DL pipeline that we have been using in this chapter. In the command line prompt, run the following command:

```
mlflow run https://github.com/PacktPublishing/Practical-
Deep-Learning-at-Scale-with-MLFlow#chapter05 -v
26119e984e52dadd04b99e6f7e95f8dda8b59238 --experiment-
name='dl_model_chapter05' -P pipeline_steps='download_data'
```

Notice the difference between this command line and the one in the previous section. Instead of a *dot* to refer to a local copy of the code, we are pointing to a remote GitHub repository (`https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow`) and the folder name (`chapter05`) that contains the MLproject file we want to reference. The `#` symbol denotes the relative path to the root folder, according to MLflow's convention (see details on the MLflow documentation at this website: <https://www.mlflow.org/docs/latest/projects.html#running-projects>). We then define a version number by specifying the Git commit hash using the `-v` parameter. In this case, it is this version we have in the GitHub repository:

```
https://github.com/PacktPublishing/
Practical-Deep-Learning-at-Scale-with-MLFlow/
tree/26119e984e52dadd04b99e6f7e95f8dda8b59238/chapter05
```



**Hidden Bug of Running an MLflow Project with GitHub's Main Branch**

When we omit the `-v` parameter in the MLflow run, MLflow will assume we want to use the default `main` branch of a GitHub project. However, MLflow's source code has a hardcoded reference to the `main` branch of a GitHub project as `origin.refs.master`, requiring the existence of a `master` branch in the GitHub project. This does not work in newer GitHub projects such as this book's project, since the default branch is called `main`, not `master` anymore, due to the recent changes introduced by GitHub (see details here: <https://github.com/github/renaming>). So, at the time of writing this book, in the MLflow version 1.22.0, there is no way to run a default `main` branch of a GitHub project. We need to specifically declare the Git commit hash version when running an MLflow project in the GitHub repository.

So, what happens when you use the code in a remote GitHub project repository when running an MLflow project? It becomes clear when you see the first line of the following console output:

```
2021/12/30 18:57:32 INFO mlflow.projects.utils: === Fetching
project from https://github.com/PacktPublishing/Practical-Deep-
Learning-at-Scale-with-MLFlow#chapter05 into /var/folders/51/
whxjy4r92dx18788yp11ycyr0000gp/T/tmpdyzaalye ===
```

This means that MLflow, on behalf of the user, starts to clone the remote project to a local temporary folder called `/var/folders/51/whxjy4r92dx18788yp11ycyr0000gp/T/tmpdyzaalye`.

If you navigate to this temporary folder, you will see that the entire project content from GitHub has been cloned to this folder, not just the folder containing the ML project you want to run.

The rest of the console output is as we have seen when using the local code. Once you finish the run with the `download_data` step, you should be able to find the downloaded data in the temporary folder under `chapter05`, since we define the local destination folder as a `./data` relative path in the ML project file:

```
local_folder: { type: str, default: ./data }
```

MLflow automatically converts this to an absolute path, and it becomes a relative path to the cloned project folder under `chapter05`, since that's where the `MLproject` file resides.

This capability to reference a remote GitHub project and run it in a local environment, whether this local environment is your laptop or a virtual machine in the cloud, is powerful. This enables automation through **continuous integration and continuous deployment (CI/CD)** since this can be directly invoked in a command line, which can then be scripted into a CI/CD script. The tracking part is also precise, since we have the Git commit hash logged in the MLflow tracking server, which allows us to know exactly which version of the code was executed.

Note in both the scenarios we just covered, the execution environment is a local machine where the MLflow run command was issued. The MLflow project runs to completion *synchronously*, meaning it is a blocking call and it will run to completion and show you the progress in the console output in real time.

However, there are additional running scenarios we need to support. For example, sometimes the machine where we issue the MLflow project run command is not powerful enough to support the computation we need, such as training a DL model with many epochs. Another scenario could be if the data to be downloaded or accessed for training is multiple gigabytes and you don't want to download it to your local laptop for model development. This requires us to be able to run the code in a remote cluster. Let's look at how we can do that in the next section.

## Running local code remotely in the cloud

In previous chapters, we ran all our code in a local laptop environment, and limited our DL fine-tuning step to only three epochs due to the limited power of a laptop. This serves the purpose of getting the code running and testing quickly in a local environment but does not serve to build an actual high-performance DL model. We really need to run the fine-tuning step in a remote GPU cluster. Ideally, we should only change some configuration and still issue the MLflow run command line in a local laptop console, but the actual pipeline will be submitted to a remote cluster in the cloud. Let's see how we can do this for our DL pipeline.

Let's start with submitting code to run in a Databricks server. There are three prerequisites:

- **An Enterprise Databricks server:** You need to have access to an Enterprise-licensed Databricks server or a free trial version of the Databricks server (<https://docs.databricks.com/getting-started/try-databricks.html#sign-up-for-a-databricks-free-trial>) in the cloud. The Community version of Databricks does not support this remote execution.

- **The Databricks CLI:** You need to set up the Databricks CLI where you issue the MLflow project run commands. To install it, simply run the following command:

```
pip install databricks-cli
```

We also include this dependency in the `requirements.txt` file of `chapter05` when you check out the code for this chapter.

- **Access tokens for accessing the Databricks server:** There are two ways to set up the tokens: using an environment variable, or using the Databricks command-line tool to generate a `.databrickscfg` file in your local home folder. You don't need both, but if you do have both, the one defined using environment variables will take a higher precedence when being picked up by the Databricks command line. The approach of using environment variables and generating access tokens is described in the *Setting up MLflow to interact with a remote MLflow server* section of *Chapter 1, Deep Learning Life Cycle and MLOps Challenges*. Note these environment variables can be set up in the command line directly or can be put into your `.bash_profile` file if you are using a macOS or Linux machine.

Here, we describe how we can use the Databricks command-line tool to generate a `.databrickscfg` file:

1. Run the following command to set up the token configuration:

```
databricks configure --token
```

2. Follow the prompt to fill in the remote Databricks host URL and the access token:

```
Databricks Host (should begin with https://):  
https://?????  
Token: dapi??????????
```

3. Now, if you check your local home folder, you should find a hidden file called `.databrickscfg`.

If you open this file, you should be able to see something like the following:

```
[DEFAULT]  
host = https://???????  
token = dapi?????????  
jobs-api-version = 2.0
```

Note that the last line indicates the remote job submission and execution API version that the Databricks server is using.

Now that you have the access set up correctly, let's see how we can run the DL pipeline remotely in the remote Databricks server using the following steps:

1. Since we are going to use the remote Databricks server, the local MLflow server we set up before no longer works. This means that we need to disable and comment out the following lines in the `main.py` file, which are only useful to the local MLflow server setup (check out the latest version of the code for `chapter05` from GitHub to follow the steps, at <https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLflow.git>):

```
os.environ["MLFLOW_TRACKING_URI"] = http://localhost
os.environ["MLFLOW_S3_ENDPOINT_URL"] = http://
localhost:9000
os.environ["AWS_ACCESS_KEY_ID"] = "minio"
os.environ["AWS_SECRET_ACCESS_KEY"] = "minio123"
```

Instead, we should use the following environment variable that can be defined in a `.bash_profile` file or directly executed in the command line:

```
export MLFLOW_TRACKING_URI="databricks"
```

This will use the MLflow tracking server on the Databricks server. If you don't specify this, it will default to a localhost but will fail since there is no localhost version of MLflow on the remote Databricks server. So, make sure you have this set up correctly. Now, we are ready to run our local code remotely.

2. Now, run the following command line to submit the local code to the remote Databricks server to run. We will just start with the `download_data` step, as follows:

```
mlflow run . -b databricks --backend-config cluster_spec.
json --experiment-name='/Shared/dl_model_chapter05' -P
pipeline_steps ='download_data'
```

You will see this time that the command line has two new parameters: `-b databricks`, which specifies the backend as a Databricks server, and `--backend-config cluster_spec.json`, which details the cluster specification. The content of this `cluster_spec.json` file is as follows:

```
{
  "new_cluster": {
    "spark_version": "9.1.x-gpu-ml-scala2.12",
    "num_workers": 1,
    "node_type_id": "g4dn.xlarge"
```

```
}  
}
```

This `cluster_spec.json` file is typically located in the same folder in which the `MLproject` file is located and needs to be predefined so that the `MLflow run` command can pick it up. The example we give here only defines a minimal set of parameters needed to create a job cluster on Databricks using AWS's GPU virtual machine as a single node, but you can create a much richer cluster specification if necessary (see the following *Cluster Specification for Databricks* box for more details).

#### Cluster Specification for Databricks

When submitting jobs to Databricks, it requires the creation of a new job cluster, which is different from an interactive cluster that you already have, where you can run an interactive job by attaching a notebook. A cluster specification is defined by minimally specifying the Databricks runtime version, which in our current example is `9.1.x-gpu-ml-scala2.12`, the number of worker nodes, and the node type ID, as shown in our example. It is recommended to use the **long-term support (LTS)** version of the Databricks runtime (<https://docs.databricks.com/release-notes/runtime/9.1ml.html>). The cluster node type depends on the cloud provider. Here, we use AWS's single GPU node (`g4dn.xlarge`) for learning purposes. There are many other configurations that you can define in this cluster specification, including storage and access permission, and `init` scripts. The easiest way to generate a working cluster specification JSON file is to use the Databricks portal UI to create a new cluster, where you can select the Databricks runtime version, cluster node types, and other parameters (<https://docs.databricks.com/clusters/create.html>). Then, you can get the JSON representation of the cluster by clicking on the JSON link on the top right of the **Create Cluster** UI page (see *Figure 5.2*).

Create Cluster

New Cluster Cancel Create Cluster DBU / hour: 1.42 ? 1 Workers:16 GB Memory, 4 Cores  
1 Driver:16 GB Memory, 4 Cores

Policy ? UI | JSON

Unrestricted | ?

Cluster name  
dl\_model

Cluster mode ?  
Standard | ?

Databricks runtime version ?  
Runtime: 9.1 LTS ML (GPU, Scala 2.12, Spark 3.1.2) | ? NVIDIA EULA ?

Autopilot options

Enable autoscaling ?

Enable autoscaling local storage ?

Terminate after  minutes of inactivity ?

Worker type ? Workers

g4dn.xlarge 16 GB Memory, 1 GPU | ? 1 ?

New Configure separate pools for workers and drivers for flexibility. [Learn more](#)

Driver type

Same as worker 16 GB Memory, 1 GPU | ?

Figure 5.2 - An example of creating a cluster on Databricks

Also notice that the `experiment-name` parameter in the preceding command no longer just takes an experiment name string but needs to include an absolute path in the Databricks workspace. This is different from the local MLflow tracking server. This convention must be followed to make this remote job submission work. Note that if you want to have several levels of subfolder structures, such as the following, then each subfolder must already exist in the Databricks server:

```
/rootPath/subfolder1/subfolder2/my_experiment_name
```

This means that the `rootPath`, `subfolder1`, and `subfolder2` folders must already exist. If not, the command line will fail since it cannot create the parent folder automatically on the Databricks server. That last string, `my_experiment_name`, can be automatically created if it does not already exist since that's the actual experiment name that will host all the experiment runs. Note that, in this example, we are using the command-line parameter to specify the experiment name, but it is also possible to use the environment variable to specify it, as follows:

```
export MLFLOW_EXPERIMENT_NAME=/Shared/dl_model_chapter05
```

3. Once this command is executed, you will see a much shorter console output message this time compared with the previous run in a local environment. This is because when executing code this way, it runs *asynchronous*, which means the job is submitted to the remote Databricks server and immediately returns to the console without waiting. Let's look at the first three lines of the output:

```
INFO: '/Shared/dl_model_chapter05' does not exist.  
Creating a new experiment  
2022/01/06 17:35:32 INFO mlflow.projects.  
databricks: === Uploading project to DBFS path /  
dbfs/mlflow-experiments/427565/projects-code/  
f1cbec57b21eabfca52f417f8482054bbea22be  
9205b5bbde461780d809924c2.tar.gz ===  
2022/01/06 17:35:32 INFO mlflow.projects.  
databricks: === Finished uploading project to /  
dbfs/mlflow-experiments/427565/projects-code/  
f1cbec57b21eabfca52f417f8482054bbea22be  
9205b5bbde461780d809924c2.tar.gz ===
```

The first line means that the experiment does not exist in the Databricks server, so it is being created. If you run this a second time, this will not show up. The second and third lines describe the process where MLflow packages the MLproject as a `.tar.gz` file and uploads it to the Databricks file server. Note that, unlike a GitHub project where it needs to check out the entire project from the repository, here, it only needs to package the `chapter05` folder since that's where our MLproject resides. This can be confirmed by looking at the job running logs in the Databricks cluster, which we will explain (where to get the job URL and how to look for the logs) in the next few paragraphs.

### Synchronous and Asynchronous Running of MLproject

The official MLflow run CLI does not support a parameter to specify the running of an MLflow project in asynchronous or synchronous mode. However, the MLflow run Python API does have a parameter called `synchronous`, which by default is set to be `True`. When using MLflow's CLI to run an MLflow job using Databricks as the backend, the default behavior is asynchronous. Sometimes, synchronous behavior of the CLI run command is desirable during CI/CD automation when you need to make sure the MLflow run completes successfully before moving to the next step. This cannot be done with the official MLflow run CLI, but you can write a wrapper CLI Python function to call MLflow's Python API with synchronous mode set to `True` and then use your own CLI Python command to run the MLflow job in synchronous mode. Also, note that `mlflow.run()` is the high-level fluent (object-oriented) API for the `mlflow.projects.run()` API. We use the `mlflow.run()` API extensively in this book for consistency. For details on the MLflow run Python API, see the official documentation page: [https://www.mlflow.org/docs/latest/python\\_api/mlflow.projects.html#mlflow.projects.run](https://www.mlflow.org/docs/latest/python_api/mlflow.projects.html#mlflow.projects.run).

The next few lines of the output look similar to the following:

```
2022/01/06 17:48:31 INFO mlflow.projects.databricks: ===
Running entry point main of project . on Databricks ===
2022/01/06 17:48:31 INFO mlflow.projects.databricks: ===
Launched MLflow run as Databricks job run with ID 279456.
Getting run status page URL... ===
2022/01/06 17:48:31 INFO mlflow.projects.databricks: ===
Check the run's status at https://???.cloud.databricks.
com#job/168339/run/1 ===
```

These lines describe that the job has been submitted to the Databricks server and the job run ID and the job URL are shown in the last line (replace `???` with your actual Databricks URL to make this work for you). Notice that the MLflow run ID is 279456, which is different from the ID you see in the job URL (168339). This is because the job URL is managed by the Databricks job management system and has a different way to generate and track each actual job.



- Click the job URL link (<https://???.cloud.databricks.com#job/168339/run/1>) and check the status of this job, which will show the progress and standard output and error logs (see *Figure 5.3*). Usually, this page will take a few minutes to start showing the running progress since it needs to create a brand new cluster based on `cluster_spec.json` before it can start running the job.

MLflow Run for . (Run id: 279456) ✕ Delete

**Started:** 2022-01-06 17:48:31 PST  
**Duration:** 3m 55s  
**Status:** Succeeded  
**Run ID:** 279456  
**Task:** shell-command-task  
 ◦ **Dependent Libraries:**  
   ▪ mlflow==1.20.2 (PyPI)  
**Cluster:** [job-168339-run-1 \(Terminated\)](#) - [View Spark UI](#) / [Logs](#) / [Metrics](#)

### Output

**Spark driver logs**  
 Auto-fetch data

**Recent log files**  
[stdout \(25973 bytes\)](#)

**Standard output**

```
mlflow, version 1.20.2
sending incremental file list
f1cbec57b21eabfca52f417f8482054bbea22be9205b5bbde461780d809924c2.tar.gz

sent 3,033 bytes received 35 bytes 2,045.33 bytes/sec
total size is 2,882 speedup is 0.94
mlflow-project/
mlflow-project/MLproject
mlflow-project/README.md
mlflow-project/cluster_spec.json
mlflow-project/conda.yaml
mlflow-project/main.py
mlflow-project/pipeline/
mlflow-project/pipeline/download_data.py
mlflow-project/pipeline/fine_tuning_model.py
mlflow-project/pipeline/register_model.py
mlflow-project/requirements.txt
Collecting package metadata (repopdata.json): ...working... done
```

Figure 5.3 – MLflow run job status page with standard output

*Figure 5.3* shows the job was successfully finished (**Status: Succeeded**) and the standard output, which shows the content of the `chapter05` folder was uploaded and extracted in the **Databricks File System (DBFS)**. As mentioned previously, only the MLproject we want to run was packaged, uploaded, and extracted in the DBFS, not the entire project repository.

On the same job status page, you will also find the standard errors section, which shows the logs describing the pipeline step we wanted to run: `download_data`. These are not errors but just informational messages. All Python logs are aggregated here. See *Figure 5.4* for details:

Standard error

```

2022/01/07 01:49:46 INFO mlflow.utils.conda: === Creating conda environment mlflow-95353930ddb7b60101df80a5d64ef8bf6204a808 ===
2022/01/07 01:52:08 INFO mlflow.projects.utils: === Created directory /tmp/tmps_xc6t_e for downloading remote URIs passed to arguments of
type 'path' ===
2022/01/07 01:52:08 INFO mlflow.projects.backend.local: === Running command 'source /databricks/conda/bin/./etc/profile.d/conda.sh && conda
activate mlflow-95353930ddb7b60101df80a5d64ef8bf6204a808 1>&2 && python main.py --pipeline_steps download_data' in run with ID
'261f9b38e670487c8c6b4c29c977da6d' ===
2022-01-07 01:52:10,134 pipeline experiment_id: 427565
2022-01-07 01:52:10,134 pipeline active steps to execute in this run: ['download_data']
2022/01/07 01:52:11 INFO mlflow.projects.utils: === Created directory /tmp/tmpde_gtb2r for downloading remote URIs passed to arguments of
type 'path' ===
2022/01/07 01:52:11 INFO mlflow.projects.backend.local: === Running command 'source /databricks/conda/bin/./etc/profile.d/conda.sh && conda
activate mlflow-95353930ddb7b60101df80a5d64ef8bf6204a808 1>&2 && python pipeline/download_data.py --download_url https://pl-flash-
data.s3.amazonaws.com/imdb.zip --local_folder ./data --pipeline_run_name chapter05' in run with ID '73c2fe1495c94c27bee752cc72caf07c' ===
2022-01-07 01:52:13,926 Downloading data from https://pl-flash-data.s3.amazonaws.com/imdb.zip

```

Figure 5.4 – MLflow job information logged on the job status page

Figure 5.4 shows the log that's very similar to what we see when we run in the local interactive environment, but now these runs were executed in the cluster we specified when we submitted the job. Note that the pipeline experiment ID is 427565 in Figure 5.4. You should be able to find the successfully completed MLflow DL pipeline runs in the integrated MLflow tracking server on the Databricks server, using the experiment ID 427565 in the following URL pattern:

`https://[your databricks hostname]/#mlflow/experiments/427565`

If you see the familiar tracking results as we have seen in previous chapters, give yourself a big hug since you just completed a major learning milestone in running local code in a remote Databricks cluster!

Furthermore, we can run multiple steps of the DL pipeline using this approach without changing any code in the individual step's implementation. For example, if we want to run both the `download_data` and `fine_tuning_model` steps of the DL pipeline, we can issue the following command:

```

mlflow run . -b databricks --backend-config cluster_spec.json
--experiment-name='/Shared/dl_model_chapter05' -P pipeline_
steps='download_data,fine_tuning_model'

```

The output console will show the following short messages:

```

2022/01/07 15:22:39 INFO mlflow.projects.databricks: ===
Uploading project to DBFS path /dbfs/mlflow-experiments/427565/
projects-code/743cadfec82a55b8c76e9f27754cfdd516545b155254e990c
2cc62650b8af959.tar.gz ===
2022/01/07 15:22:40 INFO mlflow.projects.databricks: ===
Finished uploading project to /dbfs/mlflow-experiments/427565/
projects-code/743cadfec82a55b8c76e9f27754cfdd516545b155254e990c
2cc62650b8af959.tar.gz ===

```

```
2022/01/07 15:22:40 INFO mlflow.projects.databricks: ===  
Running entry point main of project . on Databricks ===  
2022/01/07 15:22:40 INFO mlflow.projects.databricks: ===  
Launched MLflow run as Databricks job run with ID 279540.  
Getting run status page URL... ===  
2022/01/07 15:22:40 INFO mlflow.projects.databricks: ===  
Check the run's status at https://?????.cloud.databricks.  
com#job/168429/run/1 ===
```

You can then go to the job URL page shown in the last line of the console output and wait until it creates a new cluster and completes both steps. You should then be able to find both steps in the experiment folder logged in the MLflow tracking server, using the same experiment URL (since we use the same experiment name):

```
https://[your databricks hostname]/#mlflow/experiments/427565
```

Now that we know how to run local code in a remote Databricks cluster, we will learn how to run the code from a GitHub repository in a remote Databricks cluster.

## Running remotely in the cloud with remote code in GitHub

The most reliable way to reproduce a DL pipeline is to point to a specific version of the project code in GitHub and then run it in the cloud without invoking any local resources. This way, we know the exact version of the code as well as using the same running environment defined in the project. Let's see how this works with our DL pipeline.

As a prerequisite and a reminder, the following three environment variables need to be set up before you issue the MLflow run command to complete this section of the learning:

```
export MLFLOW_TRACKING_URI=databricks  
export DATABRICKS_TOKEN=[databricks_token]  
export DATABRICKS_HOST='https://[your databricks host name/'
```

We already know how to set up these environment variables from the last section. There is potentially one more setup needed, which is to allow your Databricks server to access your GitHub repository if it is non-public (see the following *GitHub Token for Databricks to Access a Non-Public or Enterprise Project Repository* box).

### GitHub Token for Databricks to Access a Non-Public or Enterprise Project Repository

To allow Databricks to access the project repository in GitHub, there is another token that's needed. This can be generated by going to your personal GitHub page (<https://github.com/settings/tokens>) and then following the steps described on this page (<https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token>). You can then follow the instructions on the Databricks documentation website to set it up: <https://docs.databricks.com/repos.html#configure-your-git-integration-with-databricks>.

Now, let's run the project using the specific version in the GitHub repository for the full pipeline on the remote Databricks cluster:

```
mlflow run https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow#chapter05 -v 395c33858a53bcd8ac217a962ab81e148d9f1d9a -b databricks --backend-config cluster_spec.json --experiment-name='/Shared/dl_model_chapter05' -P pipeline_steps='all'
```

We will then see the output as brief as six lines. Let's look at what the important information on each line shows and how this works:

1. The first line shows where the content of the project repository was downloaded to locally:

```
2022/01/07 17:36:54 INFO mlflow.projects.utils: ===
Fetching project from https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow#chapter05
into /var/folders/51/whxjy4r92dx18788yp11ycyr0000gp/T/tmpzcepn5h5 ===
```

If we go to the temporary directory shown in this message on the local machine where we execute this command, we see that the entire repository is already downloaded to this folder: `/var/folders/51/whxjy4r92dx18788yp11ycyr0000gp/T/tmpzcepn5h5`.

2. The next two lines show the project content was zipped and uploaded to a DBFS folder on the Databricks server:

```
2022/01/07 17:36:57 INFO mlflow.projects.databricks: === Uploading project to DBFS path /
```

```
dbfs/mlflow-experiments/427565/projects-code/
fba3d31e1895b78f40227b5965461faddb
61ec9df906fb09b161f74efaa90aa2.tar.gz ===
2022/01/07 17:36:57 INFO mlflow.projects.
databricks: === Finished uploading project to /
dbfs/mlflow-experiments/427565/projects-code/
fba3d31e1895b78f40227b5965461faddb61ec
9df906fb09b161f74efaa90aa2.tar.gz ===
```

If we use the local command-line tool of Databricks, we can list this `.tar.gz` file as if it is a local file (but in fact, it is located remotely on the Databricks server):

```
databricks fs ls -l dbfs:/mlflow-experiments/427565/
projects-code/fba3d31e1895b78f40227b5965461faddb61ec
9df906fb09b161f74efaa90aa2.tar.gz
```

You should see output similar to the following, which describes the attributes of the file (size, owner/group ID, and whether it is a file or directory):

```
file 3070 fba3d31e1895b78f40227b5965461faddb61ec
9df906fb09b161f74efaa90aa2.tar.gz 1641605818000
```

3. The next line shows that it starts to run the main entry point for this GitHub project:

```
2022/01/07 17:36:57 INFO mlflow.projects.databricks: ===
Running entry point main of project https://github.com/
PacktPublishing/Practical-Deep-Learning-at-Scale-with-
MLFlow#chapter05 on Databricks ===
```

Note the difference when we run the local code (it was a *dot* after the project, which means the current directory on the local system). Now, it lists the full path of the GitHub repository location.

4. The last two lines are like the previous section's output, where it lists out the job URL:

```
2022/01/07 17:36:57 INFO mlflow.projects.databricks: ===
Launched MLflow run as Databricks job run with ID 279660.
Getting run status page URL... ===
2022/01/07 17:36:57 INFO mlflow.projects.databricks: ===
Check the run's status at https://?????.cloud.databricks.
com#job/168527/run/1 ===
```

5. If we click the job URL in the last line of the console output, we will be able to see the following content on that website (*Figure 5.5*):

MLflow Run for <https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow#chapter05> (Run id: 279660) ✕ Delete

**Started:** 2022-01-07 17:36:57 PST  
**Duration:** 15m 31s  
**Status:** Succeeded  
**Run ID:** 279660  
**Task:** shell-command-task  
 ◦ **Dependent Libraries:**  
 • mlflow==1.20.2 (PyPi)  
**Cluster:** job-168527-run-1 (Terminated) - [View Spark UI / Logs / Metrics](#)

Figure 5.5 – MLflow run job status page using the code from the GitHub repository

*Figure 5.5* shows the end status of this job. Notice that the title of the page now says **MLflow Run for <https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow#chapter05>**, instead of **MLflow Run for .** as shown in the previous section when using local code to run.

The status of the job shows this was run successfully and you will also see that the results are logged in the experiment page as before, with all three steps finished. The model is also registered in the model registry as expected, in the Databricks server under the following URL:

`https://[your_databricks_hostname]/#mlflow/models/dl_finetuned_model`

In summary, the mechanism of how this approach works is shown in the following diagram (*Figure 5.6*):

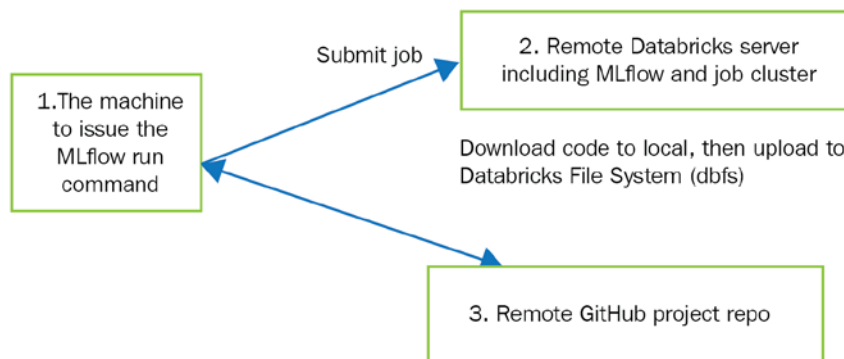


Figure 5.6 – Summary view of running remote GitHub code in a remote Databricks cluster server

Figure 5.6 shows that there are three different locations (a machine where we issue the MLflow run command, a remote Databricks server, and a remote GitHub project). When an MLflow run command is issued, the remote GitHub project source code is cloned to the machine where the MLflow run command was issued, and then uploaded to the remote Databricks server with a job submitted to execute the multiple steps of the DL pipeline. This is an asynchronous execution, and the status of the job needs to be monitored based on the job URL created.

#### Running an MLflow Project on Other Backends

Right now, Databricks supports two types of remote running backend environments: Databricks and K8s. However, as of MLflow version 1.22.0 (<https://www.mlflow.org/docs/latest/projects.html#run-an-mlflow-project-on-kubernetes-experimental>), running MLflow projects on K8s is still in experimental mode and is subject to change. If you are interested in learning more about this, refer to the reference in the *Further reading* section to explore an example provided. There are also other third-party provided backends (also called community plugins) such as `hadoop-yarn` (<https://github.com/criteo/mlflow-yarn>). Due to the availability of Databricks in all major cloud providers and its maturity in supporting enterprise security-compliant production scenarios, this book currently focuses on learning about running MLflow projects remotely in a Databricks server.

## Summary

In this chapter, we have learned how to run a DL pipeline in different execution environments (local or remote Databricks clusters) using either local source code or GitHub project repository code. This is critical not just for reproducibility and flexibility in executing a DL pipeline, but also provides much better productivity and future automation possibility using CI/CD tools. The power to run one or multiple steps of a DL pipeline in remote resource-rich environments gives us the speed to execute large-scale computation and data-intensive jobs that are typically seen in production-quality DL model training and fine-tuning. This allows us to do hyperparameter tuning or cross-validation of a DL model if necessary. We will start to learn how to run large-scale hyperparameter tuning in the next chapter as our natural next step.

## Further reading

- MLflow run projects parameters (for both command line and Python API): <https://www.mlflow.org/docs/latest/projects.html#running-projects>
- MLflow run command line (CLI) documentation: <https://www.mlflow.org/docs/latest/cli.html#mlflow-run>
- MLflow run projects on Databricks: <https://www.mlflow.org/docs/latest/projects.html#run-an-mlflow-project-on-databricks>
- An example of running an MLflow project on K8s: <https://github.com/SameeraGrandhi/mlflow-on-k8s/tree/master/examples/LogisticRegression>
- Running MLflow projects on Azure: <https://docs.microsoft.com/en-us/azure/machine-learning/how-to-train-mlflow-projects>





# 6

# Running Hyperparameter Tuning at Scale

**Hyperparameter tuning** or **hyperparameter optimization (HPO)** is a procedure that finds the best possible deep neural network structures, types of pretrained models, and model training process within a reasonable computing resource constraint and time frame. Here, hyperparameter refers to parameters that cannot be changed or learned during the ML training process, such as the number of layers inside a deep neural network, the choice of a pretrained language model, or the learning rate, batch size, and optimizer of the training process. In this chapter, we will use HPO as a shorthand to refer to the process of hyperparameter tuning and optimization. HPO is a critical step for producing a high-performance ML/DL model. Given that the search space of the hyperparameter is very large, efficiently running HPO at scale is a major challenge. The complexity and high cost of evaluating a DL model, compared to classical ML models, further compound the challenges. Therefore, we will need to learn state-of-the-art HPO approaches and implementation frameworks, implement increasingly complex and scalable HPO methods, and track them with MLflow to ensure a reproducible tuning process. By the end of this chapter, you will be comfortable with implementing scalable HPO for DL model pipelines.

In this chapter, first, we will give an overview of the different automatic HPO frameworks and applications of DL model tuning. Additionally, we will understand what to optimize and when to choose what frameworks to use. We will compare three popular HPO frameworks: **HyperOpt**, **Optuna**, and **Ray Tune**. We will show which of these is the best choice for running HPO at scale. Then, we will focus on learning how to create HPO-ready DL model codes that can use Ray Tune and MLflow. Following this, we will show how we can switch to using different HPO algorithms easily with Optuna as a primary example.

In this chapter, we'll cover the following topics:

- Understanding automatic HPO for DL pipelines
- Creating HPO-ready DL models using Ray Tune and MLflow
- Running the first Ray Tune HPO experiment with MLflow
- Running Ray Tune HPO with Optuna and HyperBand

## Technical requirements

To understand the examples in this chapter, the following key technical requirements are needed:

- Ray Tune 1.9.2: This is a flexible and powerful hyperparameter tuning framework (<https://docs.ray.io/en/latest/tune/index.html>).
- Optuna 2.10.0: This is an imperative and define-by-run hyperparameter tuning Python package (<https://optuna.org/>).
- The code for this chapter can be found in the following GitHub URL, which also includes the `requirements.txt` file that contains the preceding key packages and other dependencies: <https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/tree/main/chapter06>.

# Understanding automatic HPO for DL pipelines

Automatic HPO has been studied for over two decades since the first known paper on this topic was published in 1995 (<https://www.sciencedirect.com/science/article/pii/B9781558603776500451>). It has been widely understood that tuning hyperparameters for an ML model can improve the performance of the model – sometimes, dramatically. The rise of DL models in recent years has triggered a new wave of innovation and the development of new frameworks to tackle HPO for DL pipelines. This is because a DL model pipeline imposes many new and large-scale optimization challenges that cannot be easily solved by previous HPO methods. Note that, in contrast to the model parameters that can be learned during the model training process, a set of hyperparameters must be set before training.

## Difference between HPO and Transfer Learning's Fine-Tuning

In this book, we have been focusing on one successful DL approach called **Transfer Learning** (please refer to *Chapter 1, Deep Learning Life Cycle and MLOps Challenges*, for a full discussion). The key step of a transfer learning process is to fine-tune a pretrained model with some task- and domain-specific labeled data to get a good task-specific DL model. However, the fine-tuning step is just a special kind of model training step that also has lots of hyperparameters to optimize. That's where HPO comes into play.

## Types of hyperparameters and their challenges

There are several types of hyperparameters that you can use for a DL pipeline:

- **DL model type and architecture:** In the case of transfer learning, choosing which pretrained models to use is one possible hyperparameter. For example, there are over 27,000 pretrained models in the **Hugging Face** model repository (<https://huggingface.co/models>), including **BERT**, **RoBERTa**, and many more. For a particular prediction task, we might want to try a few of them to decide which is the best one to use.

- **Learning- and training-related parameters:** These include different types of optimizers such as **stochastic gradient descent (SGD)** and **Adam** (you can view a list of PyTorch optimizers at <https://machinelearningknowledge.ai/pytorch-optimizers-complete-guide-for-beginner/>). It also includes the associated parameters such as learning rate and batch size. It is recommended that, when applicable, the following parameters should be first tuned in their order of importance for a neural network model: learning rate, momentum, mini-batch size, the number of hidden layers, learning rate decay, and regularization (<https://arxiv.org/abs/2003.05689>).
- **Data and pipeline configurations:** A DL pipeline can include data processing and transformation steps that could impact model training. For example, if we want to compare the performance of a classification model for an email message with or without the signature text body, then a hyperparameter for whether to include an email signature is needed. Another example is when we don't have enough data or variations of data; we could try to use various data augmentation techniques that will lead to different sets of input for the model training (<https://neptune.ai/blog/data-augmentation-nlp>).

As a reminder, not all hyperparameters are tunable or require tuning. For example, it is not necessary for the **number of epochs** in a DL model to be tuned. This is because training should stop when the accuracy metric stops improving or does not hold any promise to do better than other hyperparameter configurations. This is called early stopping or pruning and is one of the key techniques underpinning some recent state-of-the-art HPO algorithms (for more discussions on early stopping, please refer to <https://databricks.com/blog/2019/08/15/how-not-to-scale-deep-learning-in-6-easy-steps.html>).

Note that all these three categories of hyperparameters can be mixed and matched, and the configuration of the entire hyperparameter space can be very large. For example, if we want to choose the type of pretrained model we want to use as a hyperparameter (for example, the choice could be **BERT** or **RoBERTa**), two learning-related parameters (such as the learning rate and batch size), and two different data augmentation techniques for NLP texts (such as random insertion and synonym replacement), then we have five hyperparameters to optimize. Note that each hyperparameter can have quite a few different candidate values to choose from, and if each hyperparameter has 5 different values, then we will have a total of  $5^5 = 3125$  combinations of hyperparameters to try. In practice, it is very common to have dozens of hyperparameters to try, and each hyperparameter could have dozens of choices or distributions to sample from. This quickly leads to a curse of dimensionality problem (<https://insaid.medium.com/automated-hyperparameter-tuning-988b5aeb7f2a>). This high-dimensional search space challenge is compounded by the expensive training and evaluation costs of DL models; we know that even 1 epoch of a tiny BERT, which we tried in the previous chapters, with a tiny set of training and validation dataset can take 1–2 mins. Now imagine a realistic production-grade DL model with HPO that could take hours, days, or even weeks if not executed efficiently. In general, the following is a list of the main challenges that require the application of high-performance HPO at scale:

- The high-dimensional search space of hyperparameters
- The high cost of model training and evaluation time for increasingly large DL models
- Time-to-production and deployment for DL models used in production

#### Performing Model Training and HPO Simultaneously

It is possible to change the hyperparameters dynamically during the training process. This is a hybrid approach that does model training and HPO simultaneously, such as **Population-Based Training (PBT)** (<https://deepmind.com/blog/article/population-based-training-neural-networks>). However, this does not change the fact that when starting a new epoch of training, a set of hyperparameters needs to be predefined. This PBT is one of the innovations that tries to reduce both the cost of searching for high-dimensional hyperparameter space and the training cost of a DL model. Interested readers should consult the *Further reading* section to dive deeper into this topic.

Now that we understand the general challenges and categories of hyperparameters to optimize, let's look at how HPO works and how to choose a framework for our usage.

## How HPO works and which ones to choose

There are different ways to understand how HPO works. The classical HPO methods include grid search and random search, where a set of hyperparameters are chosen with a range of candidate values. Each one is run independently to completion, and then we pick the best hyperparameter configuration from the set of trials we run, given the best model performance metric we found. Although this type of search is easy to implement and might not even require a sophisticated framework to support it, it is inherently inefficient and might not even find the best configuration of hyperparameters due to the non-convex nature of HPO. The term non-convex means that multiple local minimal or maximal points exist, and an optimization method might not be able to find a global optimal (that is, minimum or maximum). Put simply, a modern HPO needs to do two things:

- The adaptive sampling of hyperparameters (also known as **Configuration Selection** or **CS**): This means it needs to find which set of hyperparameters to try by taking advantage of prior knowledge. This is mostly about using different variants of Bayesian optimization to adaptively identify new configurations based on previous trials in a sequential way. This has been proven to outperform traditional grid search and random search methods.
- The adaptive evaluation of the performance of a set of hyperparameters (also known as **Configuration Evaluation** or **CE**): These approaches focus on adaptively allocating more resources to promising hyperparameter configurations while quickly pruning the poor ones. Resources can be in different forms such as the size of the training dataset (for example, only using a small fraction of the training dataset) or the number of iterations (for example, only using a few iterations to decide which ones to terminate without running to convergence). There is a family of methods called multi-armed bandit algorithms, such as the **Asynchronous Successive Halving Algorithm** (ASHA). Here, all trials start with an initial budget, then the worst half is removed, the budget is adjusted for the remaining ones, and this repeats until only one trial is left.

In practice, we want to select a suitable HPO framework using the following five criteria:

- Callback integration with MLflow
- Scalability and support of GPU clusters
- Ease of use and flexible APIs
- Integration with cutting edge HPO algorithms (**CS** and **CE**)
- Support of DL frameworks

In this book, three frameworks have been compared, and the results are summarized in *Figure 6.1*:

Frameworks	Ray Tune	Optuna	HyperOpt
	v1.9.2	v2.10.0	v0.2.7
<b>Callback integration with MLflow</b>	The <code>MLflowLoggerCallback</code> <code>@mlflow_mixin</code> decorator.	<code>MLflowCallback</code> (an experimental feature).	No callback support.
<b>Scalability and support of GPU clusters</b>	Natively supports running in parallel and GPU clusters using Ray.	Requires an external relational DB for parallelization.  No explicit support for GPU clusters.	Requires SparkTrial or MongoDB for parallel running.  No explicit support for GPU clusters.
<b>Ease of use and flexible APIs</b>	Define-and-run distributed computing APIs to support both functional and class-level APIs.	Define-by-run APIs allow dynamic search space configuration.	Define-and-run APIs and only supports the minimization of a given metric.
<b>Integration with cutting edge HPO algorithms (CS or CE)</b>	CS (called Suggest): Many cutting-edge search algorithms can even use Optuna and HyperOpt.  CE (called Scheduler): ASHA, HyperBand, PBT, and more.	CS (called Sampler): Random search, TPE, and CMA-ES.  CE (called Pruner): ASHA, HyperBand, MedianPruner, and ThresholdPruner.	CS: Random search, TPE, and Adaptive TPE.  CE: No pruning.
<b>Support of DL frameworks</b>	Supports all major DL frameworks and is easy to extend to any framework.	Supports all major DL frameworks.	Not made specifically for DL frameworks.

Figure 6.1: Comparison of Ray Tune, Optuna, and HyperOpt

As you can see from *Figure 6.1*, the winner is **Ray Tune** (<https://docs.ray.io/en/latest/tune/index.html>), when compared to **Optuna** (<https://optuna.org/>) and **HyperOpt** (<https://hyperopt.github.io/hyperopt/>). Let's explain the five criteria, as follows:

- **Callback integration with MLflow:** Optuna's support of the MLflow callback is still an experimental feature, while HyperOpt does not support callback at all, leaving additional work for users to manage the MLflow tracking for each trial run.



Only Ray Tune supports both the Python mixin decorator and callback integration with MLflow. Python mixin is a pattern that allows a standalone function to be mixed in whenever needed. In this case, the MLflow functionality is automatically mixed in during model training through the `mlflow_mixin` decorator. This can turn any training function into a Ray Tune trainable function, automatically configuring MLflow and creating a run in the same process as each Tune trial. You can then use the MLflow API inside the training function and it will automatically get reported to the correct run. Additionally, it supports MLflow's autologging, which means that all of the MLflow tracking information will be logged into the correct trial. For example, the following code snippet shows that our previous DL fine-tuning function can be turned into a `mlflow_mixin` Ray Tune function, as follows:

```
@mlflow_mixin
def train_dl_model():
    mlflow.pytorch.autolog()
    trainer = flash.Trainer(
        max_epochs=num_epochs,
        callbacks=[TuneReportCallback(
            metrics, on='validation_end')])
    trainer.finetune()
```

Note that when we define the trainer, we can add `TuneReportCallback` as one of the callbacks, which will pass the metrics back to Ray Tune, while the MLflow autologging does its job of logging all the tracking results simultaneously. In the next section, we will show you how to turn the previous chapter's example of fine-tuning the DL model into a Ray Tune trainable.

- **Scalability and support of GPU clusters:** Although Optuna and HyperOpt support parallelization, they both have dependencies on some external databases (relational databases or MongoDB) or SparkTrials. Only Ray Tune supports parallel and distributed HPO through the Ray distributed framework natively, and it is also the only one that supports running on a GPU cluster among these three frameworks.

- **Ease of use and flexibility of the APIs:** Among all the three frameworks, only Optuna supports **define-by-run** APIs, which allows you to dynamically define the hyperparameters in a Pythonic programming style, including loops and branches ([https://optuna.readthedocs.io/en/stable/tutorial/10\\_key\\_features/002\\_configurations.html](https://optuna.readthedocs.io/en/stable/tutorial/10_key_features/002_configurations.html)). This is in contrast to the **define-and-run** APIs, which both Ray Tune and HyperOpt support, where the search space is defined by a predefined dictionary prior to evaluating the objective function. These two terms, **define-by-run** and **define-and-run**, were actually coined by the DL framework's development community. In the early days, when TensorFlow 1.0 was initially released, a neural network needed to be defined first and then lazily executed later, which is called define-and-run. These two phases, 1) the construction of the neural network phase and 2) the evaluation phases, are sequentially executed, and the neural network structure cannot be changed after the construction phase. The newer DL frameworks, such as TensorFlow 2.0 (or the eager execution version of TensorFlow) and PyTorch, support the **define-by-run** neural network computation. There are no two separate phases for constructing and evaluating neural networks. Users can directly manipulate the neural networks while doing the computation. While the **define-by-run** API provided by Optuna can be used to directly define the hyperparameter search space dynamically, it does have some drawbacks. The main problem is that the parameter concurrence is not known until runtime, which could complicate the implementation of the optimization method. This is because knowing the parameter concurrence beforehand is well supported for many sampling methods. Thus, in this book, we prefer using **define-and-run** APIs. Also, note that Ray Tune can support the **define-by-run** API through integration with Optuna (you can see an example in Ray Tune's GitHub repository at [https://github.com/ray-project/ray/blob/master/python/ray/tune/examples/optuna\\_define\\_by\\_run\\_example.py#L35](https://github.com/ray-project/ray/blob/master/python/ray/tune/examples/optuna_define_by_run_example.py#L35)).
- **Integration with cutting-edge HPO algorithms (CS and CE):** On the CS side, among these three frameworks, HyperOpt has the least active development to support or integrate with the latest cutting-edge HPO sampling and search methods. Its primary search method is **Tree-Structured Parzen Estimators (TPE)**, which is a Bayesian optimization variant that's especially effective for a mixed categorical and conditional hyperparameter search space. Similarly, Optuna's primary sampling method is TPE. On the contrary, Ray Tune supports all cutting-edge searching methods, including the following:
  - DragonFly (<https://dragonfly-opt.readthedocs.io/en/master/>), which is a highly scalable Bayesian optimization framework

- BlendSearch (<https://microsoft.github.io/FLAML/docs/Use-Cases/Tune-User-Defined-Function/#hyperparameter-optimization-algorithm>) from Microsoft Research

In addition, Ray Tune also supports TPE through integration with Optuna and HyperOpt.

On the CE side, HyperOpt does not support any pruning or schedulers to stop the non-promising hyperparameter configuration. Both Optuna and Ray Tune support quite a few pruners (in Optuna) or schedulers (in Ray Tune). However, only Ray Tune supports PBT. Given the active development community and flexible API developed by Ray Tune, it is possible for Ray tune to continue to integrate and support any emerging schedulers or pruners in a timely fashion.

- **Support of DL frameworks:** HyperOpt is not specifically designed or integrated with any DL frameworks. This does not mean you cannot use HyperOpt for tuning DL models. However, HyperOpt does not offer any pruning or scheduler support to perform early stopping for unpromising hyperparameter configuration, which is a major disadvantage for HyperOpt to be used for DL model tuning. Both Ray Tune and Optuna have integration with popular DL frameworks such as PyTorch Lightning and TensorFlow/Keras.

In addition to the major criteria that we just discussed, Ray Tune also has the best documentation, extensive code examples, and a vibrant open source developer community, which is why we prefer to use Ray Tune for our learning in this chapter. In the following sections, we will learn how to create HPO-ready DL models with Ray Tune and MLflow.

## Creating HPO-ready DL models with Ray Tune and MLflow

To use Ray Tune with MLflow for HPO, let's use the fine-tuning step in our DL pipeline example from *Chapter 5, Running DL Pipelines in Different Environments*, to see what needs to be set up and what code changes we need to make. Before we start, first, let's review a few key concepts that are specifically relevant to our usage of Ray Tune:

- **Objective function:** An objective function can be either to minimize or maximize some metric values for a given configuration of hyperparameters. For example, in the DL model training and fine-tuning scenarios, we would like to maximize the F1-score for the accuracy of an NLP text classifier. This objective function needs to be wrapped as a trainable function, where Ray Tune can do HPO. In the following section, we will illustrate how to wrap our NLP text sentiment model.

- **Function-based APIs and class-based APIs:** A function-based API allows a user to insert Ray Tune statements into the model training function (called trainable in Ray Tune) such as `tune.report` for reporting model metrics ([https://docs.ray.io/en/latest/tune/api\\_docs/trainable.html#function-api](https://docs.ray.io/en/latest/tune/api_docs/trainable.html#function-api)). A class-based API requires the model training function (trainable) to be a subclass of `tune.Trainable` ([https://docs.ray.io/en/latest/tune/api\\_docs/trainable.html#trainable-class-api](https://docs.ray.io/en/latest/tune/api_docs/trainable.html#trainable-class-api)). A class-based API provides more control of how Ray Tune controls the model training processing. This might be very helpful if you start writing a new piece of architecture for a neural network model. However, when using a pretrained foundation model for fine-tuning, it is much easier to use a function-based API since we can leverage packages such as PyTorch Lightning Flash to do HPO.
- **Trials:** Each trial is a run of a specific configuration of hyperparameters. This can be executed by passing the trainable function into `tune.run`, where Ray Tune will orchestrate the HPO process.
- **Search space:** This is a set of configurations where each hyperparameter will be assigned a way in which to sample from certain distributions (for example, log uniform distribution sampling can use `tune.loguniform`) or from some categorical variables (for example, `tune.choice(['a', 'b', 'c'])` can allow you to choose these three choices uniformly). Usually, this search space is defined as a Python dictionary variable called `config`.
- **Suggest:** This is the search algorithm or CS algorithm that you need to choose for selecting the best trial. Ray Tune provides integration to many popular open source search algorithms and can automatically convert the search space defined in Ray Tune into the format that the underlying optimization algorithms expect. A list of available search algorithms can be found through the `tune.suggest` API ([https://docs.ray.io/en/latest/tune/api\\_docs/suggestion.html#tune-search-alg](https://docs.ray.io/en/latest/tune/api_docs/suggestion.html#tune-search-alg)).
- **Scheduler:** This is also called CE, as mentioned earlier. While the `tune.suggest` API provides the optimization algorithms for searching, it does not offer the early stopping or pruning capability to halt the obviously unpromising trials after just a few iterations. Since early stopping or pruning can significantly speed up the HPO process, it is highly recommended that you use a scheduler in conjunction with a searcher. Ray Tune provides many popular schedulers through its `scheduler` API (`tune.schedulers`), such as ASHA, HyperBand, and more. (Please visit [https://docs.ray.io/en/latest/tune/api\\_docs/schedulers.html#trial-schedulers-tune-schedulers](https://docs.ray.io/en/latest/tune/api_docs/schedulers.html#trial-schedulers-tune-schedulers).)

Having reviewed the basic concepts and APIs of Ray Tune, in the next section, we will be setting up Ray Tune and MLflow to run HPO experiments.

## Setting up Ray Tune and MLflow

Now that we understand the basic concepts and APIs of Ray Tune, let's see how we can set up Ray Tune to perform HPO for the fine-tuning step of our previous NLP sentiment classifier. You might want to download this chapter's code (<https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter06/>) to follow along with these instructions:

1. Install Ray Tune by typing the following command into your conda virtual environment, `dl_model_hpo`:

```
pip install ray[tune]==1.9.2
```

2. This will install Ray Tune in the virtual environment where you will launch the HPO runs for your DL model fine-tuning. Note that we have also provided the complete `requirements.txt` file in this chapter's GitHub repository (<https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter06/requirements.txt>), where you should be able to run the following installation command:

```
pip install -r requirements.txt
```

3. The complete instructions in the `README.md` file, which are in the same folder, should give you more guidance if you need to know how to set up a proper virtual environment.
4. For the MLflow setup, assuming you already have a full-fledged MLflow tracking server set up, the only thing you need to pay attention to is making sure that you have the environment variables set up correctly to access the MLflow tracking server. Run the following in your shell to set them up. Alternatively, you can overwrite your environmental variables by calling `os.environ["environmental_name"]=value` in the Python code. As a reminder, we have shown the following environment variables that can be set in the command lines per Terminal session:

```
export MLFLOW_TRACKING_URI=http://localhost
export MLFLOW_S3_ENDPOINT_URL=http://localhost:9000
export AWS_ACCESS_KEY_ID="minio"
export AWS_SECRET_ACCESS_KEY="minio123"
```

5. Run the step of `download_data` to download the raw data to the local folder under the `chapter06` parent folder:

```
mlflow run . -P pipeline_steps='download_data'
--experiment-name dl_model_chapter06
```

When the preceding execution is done, you should be able to find the IMDB data under the `chapter06/data/` folder.

Now we are ready to create an HPO step to fine-tune the NLP sentiment model we built earlier.

## Creating the Ray Tune trainable for the DL model

There are multiple changes that we need to make to allow Ray Tune to run HPO to fine-tune the DL model that we developed in previous chapters. Let's walk through the steps, as follows:

1. First, let's identify the list of possible hyperparameters (both tunable and non-tunable) in our previous fine-tuning code. Recall that our fine-tuning code looks similar to the following (only the key lines of code are shown here; the complete code can be found in `chapter05` in the GitHub repository at [https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter05/pipeline/fine\\_tuning\\_model.py#L19](https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter05/pipeline/fine_tuning_model.py#L19)):

```
datamodule = TextClassificationData.from_csv(
    input_fields="review",
    target_fields="sentiment",
    train_file=f"{data_path}/imdb/train.csv",
    val_file=f"{data_path}/imdb/valid.csv",
    test_file=f"{data_path}/imdb/test.csv")

classifier_model = TextClassifier(
    backbone= "prajjwall/bert-tiny",
    num_classes=datamodule.num_classes,
    metrics=torchmetrics.F1(datamodule.num_classes))

trainer = flash.Trainer(max_epochs=3)
```

```
trainer.finetune(classifier_model,
                 datamodule=datamodule, strategy="freeze")
```

The preceding code has four major pieces:

- `datamodule` variable: This defines the data sources for training, validation, and testing. There is a `batch_size` parameter with a default value of 1, which is not shown here, but it is one of the most important hyperparameters to tune. For more details, please see the explanation in the `lightning-flash` code documentation ([https://github.com/PyTorchLightning/lightning-flash/blob/450902d713980e0edefcfd2d2a2a35eb875072d7/flash/core/data/data\\_module.py#L64](https://github.com/PyTorchLightning/lightning-flash/blob/450902d713980e0edefcfd2d2a2a35eb875072d7/flash/core/data/data_module.py#L64)).
- `classifier_model`: This defines a classifier with the exposed parameters through the `TextClassifier` API of `lightning-flash`. There are multiple hyperparameters in the input arguments that could be tuned, including `learning_rate`, the backbone foundation model, `optimizer`, and more. You can see the complete list of input arguments in the `lightning-flash` code documentation for the `TextClassifier` API (<https://github.com/PyTorchLightning/lightning-flash/blob/450902d713980e0edefcfd2d2a2a35eb875072d7/flash/text/classification/model.py#L44>).
- `trainer`: This defines a trainer variable that can be used for fine-tuning. Here, there are a few hyperparameters that need to be set, but not necessarily tuned, such as `num_epochs`, as discussed earlier.
- `trainer.finetune`: This does the actual finetuning (transfer learning). Note that there is also a possible hyperparameter **strategy** that could be tuned.

For learning purposes, we will pick `learning_rate` and `batch_size` as the two hyperparameters to tune, as these two are the most important hyperparameters to optimize for a DL model. Once you finish this chapter, you should be able to easily add additional hyperparameters to the list of candidates for optimization.

2. Ray Tune requires a trainable function to be passed into `tune.run`. This means we need to create a trainable function. By default, a trainable function only takes one required input parameter, `config`, which contains a dictionary of key-value pairs of hyperparameters and other parameters for identifying an execution environment such as an MLflow tracking URL. However, Ray Tune provides a wrapper function, called `tune.with_parameters`, which allows you to pass along additional arbitrary parameters and objects (<https://docs.ray.io/en/latest/tune/tutorials/overview.html#how-can-i-pass-further-parameter-values-to-my-trainable>). First, let's create a function called `finetuning_dl_model` to encapsulate the logic that we just examined regarding the fine-tuning step, using a `mlflow_mixin` decorator. This allows MLflow to be initialized automatically when this function is called:

```
@mlflow_mixin
def finetuning_dl_model(config, data_dir=None,
                        num_epochs=3, num_gpus=0):
```

This function takes a `config` dictionary as input where a list of hyperparameters and MLflow configurations can be passed in. Additionally, we add three additional arguments to the function signature: `data_dir` for the location of the directory, `num_epochs` for the maximum number of epochs for each trial to run, and `num_gpus` for the number of GPUs for each trial to use if there is any.

3. In this `mlflow_mixin` decorated function, we can use all the MLflow tracking APIs if necessary, but as of MLflow version 1.22.0, since MLflow's autologging support no longer is an experimental feature, but a mature production quality feature (<https://github.com/mlflow/mlflow/releases/tag/v1.22.0>), we should just use autologging in our code, as follows:

```
mlflow.pytorch.autolog()
```

This is efficient and requires no change. However, the `batch_size` hyperparameter is not automatically captured by autologging, so we need to add one more logging statement after the fine-tuning is done, as follows:

```
mlflow.log_param('batch_size', config['batch_size'])
```



4. In the rest of the implementation body of the `finetuning_dl_model` function, the majority of the code is the same as before. There are a few changes. In the `datamodule` variable assignment statement, we add `batch_size=config['batch_size']` to allow the mini-batch size of the training data to be tunable, as shown here:

```
datamodule = TextClassificationData.from_csv(  
    input_fields="review",  
    target_fields="sentiment",  
    train_file=f"{data_dir}/imdb/train.csv",  
    val_file=f"{data_dir}/imdb/valid.csv",  
    test_file=f"{data_dir}/imdb/test.csv",  
    batch_size=config['batch_size'])
```

5. When defining the `classifier_model` variable, instead of using the default values of the set of hyperparameters, now we need to pass in the `config` dictionary to assign these values:

```
classifier_model = TextClassifier(  
    backbone=config['foundation_model'],  
    learning_rate=config['lr'],  
    optimizer=config['optimizer_type'],  
    num_classes=datamodule.num_classes,  
    metrics=torchmetrics.F1(datamodule.num_classes))
```

6. Next, we need to modify the trainer assignment code. Here, we need to do two things: first, we need to define a metrics key-value dictionary to pass from PyTorch Lightning to Ray Tune. The key in this metrics dictionary is the name to be referenced in the Ray Tune trial run, while the value of the key in this dictionary is the corresponding metric name reported by PyTorch Lightning.

### Metric Names in the PyTorch Lightning's Validation Step

When passing the metrics to Ray Tune, first, we need to know the metric names used in PyTorch Lightning during the validation step since HPO only uses validation data for evaluation, not the hold-out test datasets. It turns out PyTorch Lightning has a hardcoded convention to prefix all metrics with the corresponding training, validation, and testing step names and an underscore. A metric named `f1` will be reported in PyTorch Lightning as `train_f1` during the training step, `val_f1` during the validation step, and `test_f1` during the testing step. (You can view the PyTorch Lightning code logic at <https://github.com/PyTorchLightning/lightning-flash/blob/8b244d785c5569e9aa7d2b878a5f94af976d3f55/flash/core/model.py#L462>). In our example, we can pick `cross_entropy` and `f1` as the metrics during the validation step, which are named `val_cross_entropy` and `val_f1`, to pass back to Ray Tune as `loss` and `f1`, respectively. That means, in Ray Tune's trial run, we reference these two metrics as simply `loss` and `f1`.

So, here we define two metrics that we want to pass from the PyTorch Lightning validation step, `val_cross_entropy` and `val_f1`, to Ray Tune as `loss` and `f1`, respectively:

```
metrics = {"loss": "val_cross_entropy", "f1": "val_f1"}
```

Now, we can pass this metrics dictionary to the trainer assignment, as follows:

```
trainer = flash.Trainer(max_epochs=num_epochs,
                        gpus=num_gpus,
                        progress_bar_refresh_rate=0,
                        callbacks=[TuneReportCallback(metrics,
                                                    on='validation_end')])
```

Notice that the metrics dictionary is passed through `TuneReportCallback` when the `validation_end` event happens. This means that when the validation step is done in PyTorch Lightning, it will automatically trigger the Ray Tune report function to report the list of metrics back to Ray Tune for evaluation. The supported list of valid events for `TuneReportCallback` to use can be found in Ray Tune's integration with the PyTorch Lightning source code ([https://github.com/ray-project/ray/blob/fb0d6e6b0b48b0a681719433691405b96fbae104/python/ray/tune/integration/pytorch\\_lightning.py#L170](https://github.com/ray-project/ray/blob/fb0d6e6b0b48b0a681719433691405b96fbae104/python/ray/tune/integration/pytorch_lightning.py#L170)).

7. Finally, we can call `trainer.finetune` to execute the fine-tuning step. Here, we can pass `finetuning_strategies` as one of the tunable hyperparameters to the argument list:

```
trainer.finetune(classifier_model,
                 datamodule=datamodule,
                 strategy=config['finetuning_strategies'])
```

8. This completes the changes to the original function of fine-tuning the DL model. Now we have a new `finetuning_dl_model` function that's ready to be wrapped in `tune.with_parameters` to become a Ray Tune trainable function. It should be called as follows:

```
trainable = tune.with_parameters(finetuning_dl_model,
                                data_dir, num_epochs, num_gpus)
```

9. Note that there is no need to pass the `config` parameter, as it is implicitly assumed that it's the first parameter of `finetuning_dl_model`. The other three parameters need to be passed to the `tune.with_parameters` wrapper. Also, make sure this statement to create a trainable object for Ray Tune is placed outside of the `finetuning_dl_model` function.

In the next section, it will be placed inside Ray Tune's HPO running function called `run_hpo_dl_model`.

## Creating the Ray Tune HPO run function

Now, let's create a Ray Tune HPO run function to do the following five things:

- Define the MLflow runtime configuration parameters including a tracking URI and an experiment name.
- Define the hyperparameter search space using Ray Tune's random distributions API ([https://docs.ray.io/en/latest/tune/api\\_docs/search\\_space.html#random-distributions-api](https://docs.ray.io/en/latest/tune/api_docs/search_space.html#random-distributions-api)) to sample the list of hyperparameters we identified earlier.
- Define a Ray Tune trainable object using `tune.with_parameters`, as shown toward the end of the previous subsection.
- Call `tune.run`. This will execute the HPO run and return Ray Tune's experiment analysis object when it has been completed.
- Log the best configuration parameters when the entire HPO run is finished.

Let's walk through the implementation to see how this function can be implemented:

1. First, let's define the hyperparameter's config dictionary, as follows:

```
mlflow.set_tracking_uri(tracking_uri)
mlflow.set_experiment(experiment_name)
```

This will take `tracking_uri` and `experiment_name` of MLflow as the input parameters and set them up correctly. If this is the first time you're running this, MLflow will also create the experiment.

2. Then, we can define the config dictionary, which can include both tunable and non-tunable parameters, and the MLflow configuration parameters. As discussed in the previous section, we will tune `learning_rate` and `batch_size` but will also include other hyperparameters for bookkeeping and future tuning purposes:

```
config = {
    "lr": tune.loguniform(1e-4, 1e-1),
    "batch_size": tune.choice([32, 64, 128]),
    "foundation_model": "prajjwal1/bert-tiny",
    "finetuning_strategies": "freeze",
    "optimizer_type": "Adam",
    "mlflow": {
        "experiment_name": experiment_name,
        "tracking_uri": mlflow.get_tracking_uri()
    },
}
```

As you can see from the config dictionary, we called `tune.loguniform` to sample a log uniform distribution between  $1e-4$  and  $1e-1$  to select a learning rate. For the batch size, we called `tune.choice` to select one of three distinct values uniformly. For the rest of the key-value pairs, they are non-tunable since they do not use any sampling methods but are needed to run the trials.

3. Define the trainable object using `tune.with_parameters` with all of the extra parameters except for the config parameter:

```
trainable = tune.with_parameters(
    finetuning_dl_model,
    data_dir=data_dir,
    num_epochs=num_epochs,
    num_gpus=gpus_per_trial)
```

In the next statement, this will be called the `tune.run` function.

4. Now we are ready to run the HPO by calling `tune.run`, as follows:

```
analysis = tune.run(
    trainable,
    resources_per_trial={
        "cpu": 1,
        "gpu": gpus_per_trial
    },
    metric="f1",
    mode="max",
    config=config,
    num_samples=num_samples,
    name="hpo_tuning_dl_model")
```

Here, the objective is to find the set of hyperparameters that maximizes the F1-score among all of the trials, so the mode is `max` and the metric is `f1`. Note that this metric name, `f1`, is from the `metrics` dictionary that we defined in the previous `finetuning_dl_model` function, where we mapped PyTorch Lightning's `val_f1` to `f1`. This `f1` value is then passed to Ray Tune at the end of each trial's validation step. The `trainable` object is passed to `tune.run` as the first parameter, which will be executed as many times as the parameter of `num_samples` allows. Following this, `resources_per_trial` defines the CPU and GPU to use. Note that in the preceding example, we haven't specified any search algorithms. This means it will use `tune.suggest.basic_variant` by default, which is a grid search algorithm. There is also no scheduler defined, so, by default, there is no early stopping, and all trials will be run in parallel with the maximum number of CPUs allowed on the execution machine. When the run finishes, an `analysis` variable is returned, which contains the best hyperparameters found, along with other information.

5. Log the best configuration of the hyperparameters found. This can be done by using the returned `analysis` variable from `tune.run`, as follows:

```
logger.info("Best hyperparameters found were: %s", analysis.
    best_config)
```

That's it. Now we can give it a try. If you download the complete code from this chapter's GitHub repository, you should be able to find the `hpo_fineting_model.py` file under the `pipeline` folder.

With the preceding change, now we are ready to run our first HPO experiment.

## Running the first Ray Tune HPO experiment with MLflow

Now that we have set up Ray Tune, MLflow, and created the HPO run function, we can try to run our first Ray Tune HPO experiment, as follows:

```
python pipeline/hpo _ finetuning _ model.py
```

After a couple of seconds, you will see the following screen, *Figure 6.2*, which shows that all 10 trials (that is, the values that we set for `num_samples`) are running concurrently:

```
== Status ==
Current time: 2022-02-05 19:59:44 (running for 00:00:09.23)
Memory usage on this node: 16.3/32.0 GiB
Using FIFO scheduling algorithm.
Resources requested: 10.0/12 CPUs, 0/0 GPUs, 0.0/11.12 GiB heap, 0.0/5.56 GiB objects
Result logdir: /Users/yongliu/ray_results/hpo_tuning_dl_model
Number of trials: 10/10 (10 RUNNING)
+-----+-----+-----+-----+-----+
| Trial name          | status | loc           | batch_size | lr      |
+-----+-----+-----+-----+-----+
| finetuning_dl_model_32854_00000 | RUNNING | 127.0.0.1:26756 | 32 | 0.000165628 |
| finetuning_dl_model_32854_00001 | RUNNING | 127.0.0.1:26757 | 32 | 0.01546 |
| finetuning_dl_model_32854_00002 | RUNNING | 127.0.0.1:26755 | 32 | 0.000232962 |
| finetuning_dl_model_32854_00003 | RUNNING | 127.0.0.1:26752 | 32 | 0.00162333 |
| finetuning_dl_model_32854_00004 | RUNNING | 127.0.0.1:26758 | 128 | 0.0144146 |
| finetuning_dl_model_32854_00005 | RUNNING | 127.0.0.1:26754 | 32 | 0.000620787 |
| finetuning_dl_model_32854_00006 | RUNNING | 127.0.0.1:26749 | 128 | 0.00999589 |
| finetuning_dl_model_32854_00007 | RUNNING | 127.0.0.1:26750 | 32 | 0.00318296 |
| finetuning_dl_model_32854_00008 | RUNNING | 127.0.0.1:26753 | 64 | 0.00347588 |
| finetuning_dl_model_32854_00009 | RUNNING | 127.0.0.1:26751 | 64 | 0.067672 |
+-----+-----+-----+-----+-----+
```

Figure 6.2 – Ray Tune running 10 trials in parallel on a local multi-core laptop

After approximately 12–14 mins, you will see that all the trials have finished and the best hyperparameters will be printed out on the screen, as shown in the following (your results might vary due to the stochastic nature, the limited number of samples, and the use of grid search, which does not guarantee a global optimal):

```
Best hyperparameters found were: {'lr': 0.025639008922511797,
'batch_size': 64, 'foundation_model': 'prajjwal1/bert-
tiny', 'finetuning_strategies': 'freeze', 'optimizer_type':
'Adam', 'mlflow': {'experiment_name': 'hpo-tuning-chapter06',
'tracking_uri': 'http://localhost'}}
```

You can find the results for each trial under the result log directory, which, by default, is in the current user's `ray_results` folder. From *Figure 6.2*, we can see that the results are in `/Users/yongliu/ray_results/hpo_tuning_dl_model`.

You will see the final output of the best hyperparameters on your screen, which means you have completed running your first HPO experiment! You can see that all 10 trials are logged in the MLflow tracking server, and you can visualize and compare all 10 runs using the parallel coordinates plot provided by the MLflow tracking server. You can produce such a plot by going to the MLflow experiment page and selecting the 10 trials you just finished and then clicking on the **Compare** button near the top of the page (see *Figure 6.3*). This will bring you to the side-by-side comparison page with the plotting options being displayed at the bottom of the page:

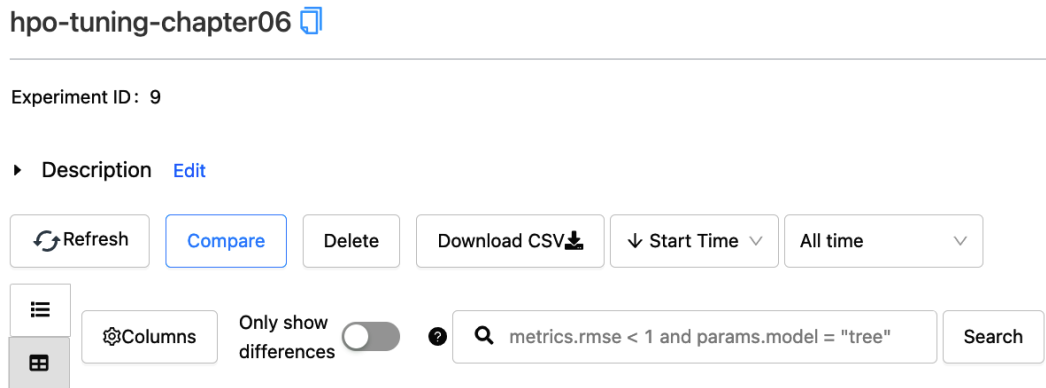


Figure 6.3 – Clicking Compare to compare all 10 trial runs on the MLflow experiment page

You can click on the **Parallel Coordinates Plot** menu item, which allows you to select the parameters and metrics to plot. Here, we select `lr` and `batch_size` as the parameters and `val_f1` and `val_cross_entropy` as the metrics. The plot is shown in *Figure 6.4*:

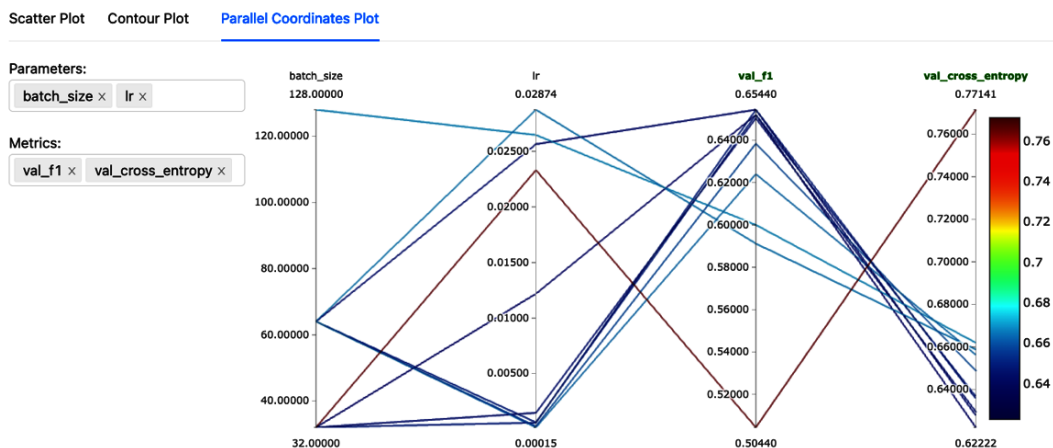


Figure 6.4 –Parallel Coordinates Plot for comparing the HPO trial results

As you can see in *Figure 6.4*, it is very easy to see that **batch\_size** of 128 and **lr** of 0.02874 produce the best **val\_f1** score of 0.6544 and **val\_cross\_entropy** (the loss value) of 0.62222. As mentioned earlier, this HPO run did not use any advanced search algorithms and schedulers, so let's see whether we can do better with more experiments in the following sections using early stopping and pruning.

## Running HPO with Ray Tune using Optuna and HyperBand

Now, let's do some experiments with different search algorithms and schedulers. Given that Optuna is such a great TPE-based search algorithm, and ASHA is a great scheduler that does asynchronous parallel trials with early termination of the unpromising ones, it would be interesting to see how many changes we need to do to make this work.

It turns out the change is very minimal based on what we have already done in the previous section. Here, we will illustrate the four main changes:

1. Install the **Optuna** package. This can be done by running the following command:

```
pip install optuna==2.10.0
```

This will install Optuna in the same virtual environment that we had before. If you have already run `pip install -r requirements.text`, then Optuna has already been installed and you can skip this step.



2. Import the relevant Ray Tune modules that integrate with Optuna and the ASHA scheduler (here, we use the HyperBand implementation of ASHA) as follows:

```
from ray.tune.suggest import ConcurrencyLimiter
from ray.tune.schedulers import AsyncHyperBandScheduler
from ray.tune.suggest.optuna import OptunaSearch
```

3. Now we are ready to add the search algorithm variable and scheduler variable to the HPO execution function, `run_hpo_dl_model`, as follows:

```
searcher = OptunaSearch()
searcher = ConcurrencyLimiter(searcher, max_concurrent=4)
scheduler = AsyncHyperBandScheduler()
```

Note that the `searcher` variable is now using Optuna, and we set the maximal number of concurrent runs to 4 for this `searcher` variable to try at any given time during the HPO search process. The scheduler is initialized with the HyperBand scheduler.

4. Assign the searcher and scheduler to the corresponding parameters of the `tune.run` call, as follows:

```
analysis = tune.run(
    trainable,
    resources_per_trial={
        "cpu": 1,
        "gpu": gpus_per_trial
    },
    metric="f1",
    mode="max",
    config=config,
    num_samples=num_samples,
    search_alg=searcher,
    scheduler=scheduler,
    name="hpo_tuning_dl_model")
```

Note that `searcher` is assigned to the `search_alg` parameter, and `scheduler` is assigned to the `scheduler` parameter. That's it. Now we are ready to run HPO with Optuna under the unified Ray Tune framework, with all of the MLflow integration that's already been provided by Ray Tune.

We have provided the complete Python code in the `hpo_finetuning_model_optuna.py` file under the `pipeline` folder. Let's run this HPO experiment as follows:

```
python pipeline/hpo_finetuning_model_optuna.py
```

You will immediately notice the following in the console output:

```
[I 2022-02-06 21:01:27,609] A new study created in memory with
name: optuna
```

This means that we are now using Optuna as the search algorithm. Additionally, you will notice that there are four concurrent trials in the status output displayed on the screen. As time goes by, some trials will be terminated after one or two iterations (epochs) before completion. This means ASHA is at work and has eliminated those unpromising trials to save computing resources and speed up the searching process. *Figure 6.5* shows one of the outputs during the run where three trials were terminated with only one iteration. You can find `num_stopped=3` in the status output (the third line in *Figure 6.5*), where it says `Using AsyncHyperBand: num_stopped=3`. This means that `AsyncHyperBand` terminated these three trials before they were completed:

```
Current time: 2022-02-06 21:10:14 (running for 00:08:47.23)
Memory usage on this node: 18.8/32.0 GiB
Using AsyncHyperBand: num_stopped=3
Bracket: Iter 64.000: None | Iter 16.000: None | Iter 4.000: None | Iter 1.000: 0.6442999988794327
Resources requested: 4.0/12 CPUs, 0/0 GPUs, 0.0/9.77 GiB heap, 0.0/4.88 GiB objects
Current best trial: 01ddc54e with fi=0.6503999829292297 and parameters={'lr': 0.0009599443695046438, 'batch_size': 128,
periment_name': 'hpo-tuning-chapter06', 'tracking_uri': 'http://localhost'}}
Result logdir: /Users/yongliu/ray_results/hpo_tuning_dl_model
Number of trials: 10/10 (4 RUNNING, 6 TERMINATED)
```

Trial name	status	loc	batch_size	lr	iter	total time (s)
finetuning_dl_model_fa500d0e	RUNNING	127.0.0.1:54653	128	0.000170111		
finetuning_dl_model_fd0f1ed6	RUNNING	127.0.0.1:54656	128	0.000216298		
finetuning_dl_model_168bd138	RUNNING	127.0.0.1:54652	128	0.000423303		
finetuning_dl_model_229ccb26	RUNNING	127.0.0.1:54651	128	0.000369355		
finetuning_dl_model_019dccee	TERMINATED	127.0.0.1:54660	128	0.00969524	1	153.858
finetuning_dl_model_01b9f736	TERMINATED	127.0.0.1:54659	128	0.0286883	3	375.601
finetuning_dl_model_01ccc316	TERMINATED	127.0.0.1:54661	32	0.00220535	3	420.799
finetuning_dl_model_01ddc54e	TERMINATED	127.0.0.1:54657	128	0.000959944	3	372.249
finetuning_dl_model_5fb8cda8	TERMINATED	127.0.0.1:54658	32	0.00885667	1	163.147
finetuning_dl_model_c4744d3a	TERMINATED	127.0.0.1:54655	32	0.000972929	1	151.947

Figure 6.5 – Running HPO with Ray Tune using Optuna and AsyncHyperBand

At the end of the run, you will see the following results:

```
2022-02-06 21:11:59,695 INFO tune.py:626 -- Total run time:
632.10 seconds (631.91 seconds for the tuning loop).
2022-02-06 21:11:59,728 Best hyperparameters found were: {'lr':
0.0009599443695046438, 'batch_size': 128, 'foundation_model':
'prajjwall1/bert-tiny', 'finetuning_strategies': 'freeze',
```

```
'optimizer_type': 'Adam', 'mlflow': {'experiment_name':  
'hpo-tuning-chapter06', 'tracking_uri': 'http://localhost'}}
```

Notice that the total run time was only 10 minutes. Compared with the previous section that used grid search without early stopping, this saves 2–4 minutes. Now, this might seem brief, but remember that we are only using a tiny BERT model here with only 3 epochs. In a production HPO run, using a large pretrained foundation model with 20 epochs is not uncommon, and the speed of searching will be significant with a good search algorithm combined with a scheduler such as the Asynchronous HyperBand scheduler. The integration of MLflow provided by Ray Tune comes for free, as we can now switch to a different search algorithm and/or a scheduler under a single framework.

While this section only shows you how to use Optuna within the Ray Tune and MLflow framework, replacing Optuna with HyperOpt is a simple drop-in change. Instead of initializing a searcher with OptunaSearch, we can use HyperOptSearch (you can see an example at [https://github.com/ray-project/ray/blob/d6b0b9a209e3f693afa6441eb284e48c02b10a80/python/ray/tune/examples/hyperopt\\_conditional\\_search\\_space\\_example.py#L80](https://github.com/ray-project/ray/blob/d6b0b9a209e3f693afa6441eb284e48c02b10a80/python/ray/tune/examples/hyperopt_conditional_search_space_example.py#L80)), and the rest of the code is the same. We leave this as an exercise for you to explore.

#### Using Different Search Algorithms and Schedulers with Ray Tune

Note that not all search algorithms can work with any scheduler. What search algorithms and schedulers you choose depends on the model complexity and evaluation cost. For a DL model, since the cost of running one epoch is usually high, it is very desirable to use a modern search algorithm such as TPE, Dragonfly, and BlendSearch, coupled with an ASHA type scheduler such as the HyperBand scheduler that we use. For more detailed guidance on which search algorithms and schedulers to use, you should consult the following documentation on the Ray Tune website: <https://docs.ray.io/en/latest/tune/tutorials/overview.html#which-search-algorithm-scheduler-should-i-choose>.

Now that we understand how to use Ray Tune and MLflow to do highly parallel and efficient HPO for DL models, this builds the foundation for us to do more advanced HPO experiments at scale in the future.

---

## Summary

In this chapter, we covered the fundamentals and challenges of HPO, why it is important for the DL model pipeline, and what a modern HPO framework should support. We compared three popular frameworks – Ray Tune, Optuna, and HyperOpt – and picked Ray Tune as the winner for running state-of-the-art HPO at scale. We saw how to create HPO-ready DL model code using Ray Tune and MLflow and ran our first HPO experiment with Ray Tune and MLflow. Additionally, we covered how to switch to other search and scheduler algorithms once we have our HPO code framework set up, using the Optuna and HyperBand schedulers as an example. The learnings from this chapter will help you to competently carry out large-scale HPO experiments in real-life production environments, allowing you to produce high-performance DL models in a cost-effective way. We have also provided many references in the *Further reading* section at the end of this chapter to encourage you to study further.

In our next chapter, we will continue learning how to build preprocessing and postprocessing steps for a model inference pipeline using MLflow, which is a typical scenario in a real production environment after having an HPO-tuned DL model that's ready for production.

## Further reading

- *Best Tools for Model Tuning and Hyperparameter Optimization*: <https://neptune.ai/blog/best-tools-for-model-tuning-and-hyperparameter-optimization>
- Comparison between Optuna and HyperOpt: <https://neptune.ai/blog/optuna-vs-hyperopt>
- *How (Not) to Tune Your Model with Hyperopt*: <https://databricks.com/blog/2021/04/15/how-not-to-tune-your-model-with-hyperopt.html>
- *Why Hyper parameter tuning is important for your model?*: <https://medium.com/analytics-vidhya/why-hyper-parameter-tuning-is-important-for-your-model-1ff4c8f145d3>
- *The Art of Hyperparameter Tuning in Deep Neural Nets by Example*: <https://towardsdatascience.com/the-art-of-hyperparameter-tuning-in-deep-neural-nets-by-example-685cb5429a38>
- *Automated Hyperparameter tuning*: <https://insaid.medium.com/automated-hyperparameter-tuning-988b5aeb7f2a>

- *Get better at building PyTorch models with Lightning and Ray Tune*: <https://towardsdatascience.com/get-better-at-building-pytorch-models-with-lightning-and-ray-tune-9fc39b84e602>
- *Ray & MLflow: Taking Distributed Machine Learning Applications to Production*: <https://medium.com/distributed-computing-with-ray/ray-mlflow-taking-distributed-machine-learning-applications-to-production-103f5505cb88>
- *A Novice's Guide to Hyperparameter Optimization at Scale*: <https://wood-b.github.io/post/a-novices-guide-to-hyperparameter-optimization-at-scale/>
- A Databricks notebook to run Ray Tune and MLflow on a Databricks cluster: <https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/6762389964551879/1089858099311442/7376217192554178/latest.html>
- *A Brief Introduction to Ray Distributed Objects, Ray Tune, and a Small Comparison to Parsl*: <https://cloud4scieng.org/2021/04/08/a-brief-introduction-to-ray-distributed-objects-ray-tune-and-a-small-comparison-to-parsl/>

# Section 4 – Deploying a Deep Learning Pipeline at Scale

In this section, we will learn how to implement and deploy a multi-step inference pipeline for production usage. We will start with an overview of four patterns of inference workflows in production. We will then learn how to implement a multi-step inference pipeline with preprocessing and postprocessing steps around a fine-tuned **deep learning (DL)** model using MLflow **PyFunc** APIs. With a ready-to-deploy MLflow PyFunc-compatible DL inference pipeline, we will learn about different deployment tools and hosting environments to decide which tool to use for a specific deployment scenario. We will then implement and deploy a batch inference pipeline using MLflow's Spark **user-defined function (UDF)**. From there on, we will focus on deploying a web service using either MLflow's built-in model serving tool or Ray Serve's MLflow deployment plugin. Finally, we will show a complete step-by-step guide to deploying a DL inference pipeline to a managed AWS SageMaker instance for production usage.

This section comprises the following chapters:

- *Chapter 7, Multi-Step Deep Learning Inference Pipeline*
- *Chapter 8, Deploying a DL Inference Pipeline at Scale*



# 7

# Multi-Step Deep Learning Inference Pipeline

Now that we have successfully run **HPO (Hyperparameter Optimization)** and produced a well-tuned DL model that meets the business requirements, it is time to move to the next step towards using this model for prediction. This is where the model inference pipeline comes into play, where the model is used for predicting or scoring real-world data in production, either in real time or batch mode. However, an inference pipeline usually does not just rely on a single model but needs preprocessing and postprocessing logic that is not necessarily seen during the model development stage. Examples of preprocessing steps include detecting the language locale (English or some other languages) before passing the input data to the model for scoring. Postprocessing could include enriching the predicted labels with additional metadata to meet the business application's requirements. There are also patterns of ML/DL inference pipelines that could even involve an ensemble of models to solve a real-world business problem. Many ML projects often underestimate the efforts needed to implement a production inference pipeline, which could result in degradation of the model's performance in production or in the worst case, failure of the entire project. Thus, it is important to learn how to recognize the pattern of different inference pipelines and implement them properly before we deploy the model into production.



By the end of this chapter, you will be able to use MLflow to confidently implement preprocessing and postprocessing steps for a multi-step inference pipeline that is ready to be used in production in future chapters.

In this chapter, we're going to cover the following main topics:

- Understanding patterns of DL inference pipelines
- Understanding the MLflow Model Python Function API
- Implementing a custom MLflow Python model
- Implementing preprocessing and postprocessing steps in a DL inference pipeline
- Implementing an inference pipeline as a new entry point in the main ML project

## Technical requirements

The following are the technical requirements for this chapter:

- The GitHub code for this chapter: <https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/tree/main/chapter07>
- A full-fledged local MLflow tracking server, as described in *Chapter 3, Tracking Models, Parameters, and Metrics*.

## Understanding patterns of DL inference pipelines

As the model development enters the stage of implementing an inference pipeline for the upcoming production usage, it is important to understand that having a well-tuned and trained DL model is only half the success story for business AI strategy. The other half includes deploying, serving, monitoring, and continuously improving the model after it goes into production. Designing and implementing a DL inference pipeline is the initial step toward the second half of the story. While the model has been trained, tuned, and tested on curated offline datasets, now it needs to handle prediction in two ways:

- **Batch inference:** This usually requires some scheduled or ad hoc execution of an inference pipeline for some offline batch of observational data. The turnaround time for producing prediction results is daily, weekly, or other schedules.

- **Online inference:** This usually requires a web service for real-time execution of an inference pipeline that produces prediction results for input data in under a second or even less than 100 milliseconds depending on the user scenarios.

Note that because the execution environment and data characteristics could be different from the offline training and testing environment, there will be additional preprocessing or postprocessing steps around the core model logic developed during the model training and tuning steps. While it should be emphasized that any sharable data preprocessing steps should be used in both the training pipeline and inference pipeline, it is unavoidable that some business logic will come into play, which will allow the inference pipeline to have additional preprocessing and postprocessing logic. For example, a very common step in a DL inference pipeline is to use caching to store and return prediction results based on a recently seen input so that an expensive model evaluation does not need to be invoked. This step is not needed for a training/testing pipeline during the model development stage.

While the pattern for inference pipelines is still emerging, it is now commonly known that there are at least four patterns in a real-world production environment:

- **Multi-step pipeline:** This is the most typical usage of the model in production, which includes a linear workflow of preprocessing steps before the model logic is invoked and some postprocessing steps after the model evaluation results are returned. While this is conceptually simple, the implementation can still be varied. We will see how we can do this efficiently in this chapter using MLflow.
- **Ensemble of models:** This is a more complex scenario where multiple different models can be used. These could be the same types of models with different versions for A/B testing purposes or different types of models. For example, for a complex conversational AI chatbot scenario, an intent classification model of the user query to classify user intents into a specific category is required. Then a content relevance model is also required to retrieve relevant answers to present to the user based on the detected user intent.
- **Business logic and model:** This usually involves additional business logic on how and where the input to the model should come from, such as querying from an enterprise database for user information and validation or retrieving precomputed additional features from a feature store before invoking a model. In addition, postprocessing business logic could also transform the prediction results into some application-specific logic and store the results in some backend storage. While this could be as simple as a linear multi-step pipeline, it can also quickly become a **DAG (Directed Acyclic Graph)** with multiple fan-in and fan-out parallel tasks before and after the model has been invoked.

- **Online learning:** This is one of the most complex inference tasks in production where a model is constantly learning and updating its parameters such as reinforcement learning.

While it is necessary to understand the big picture of the complexity of inference pipelines in production, the purpose of this chapter is to learn how we can create reusable building blocks of inference pipelines that could be used in multiple scenarios through the powerful and generic MLflow Model API, which can encapsulate preprocessing and postprocessing steps alongside a trained model. Interested readers are encouraged to learn more about the model pattern in production from this post (<https://www.anyscale.com/blog/serving-ml-models-in-production-common-patterns>) and other references in the *Further reading* section.

So, what's the MLflow Model API and how do you use that to implement preprocessing and postprocessing logic for a multi-step inference pipeline? Let's find out in the next section.

#### **Multi-Step Inference Pipeline as an MLflow Model**

Previously, in *Chapter 3, Tracking Models, Parameters, and Metrics*, we introduced the flexible loosely coupled multi-step pipeline implementation using MLflow **MLproject** so that we could execute and track a multi-step training pipeline explicitly in MLflow. However, during inference time, it is desirable to implement lightweight preprocessing and postprocessing logic alongside a trained model that's already logged in the model repository. The MLflow Model API provides a mechanism to wrap a trained model with preprocessing and postprocessing logic and then save the newly wrapped model as a new model that encapsulates the inference pipeline logic. This unifies the way to load an original model or an inference pipeline model using MLflow Model APIs. This is critical for flexible deployment using MLflow and opens doors for creative inference pipeline building.

## Understanding the MLflow Model Python Function API

The MLflow Model (<https://www.mlflow.org/docs/latest/models.html#id25>) is one of the core components provided by MLflow to load, save, and log models in different flavors (for example, a **scikit-learn** or a **PyTorch** model flavor). A model flavor is an MLflow defined standard format that explicitly specifies a directory of arbitrary files and a description file called **MLmodel**. As a reminder and an example, *Figure 7.1* shows what we have saved after fine-tuning our example NLP sentiment classifier in the MLflow artifact store and the content of the MLmodel file:

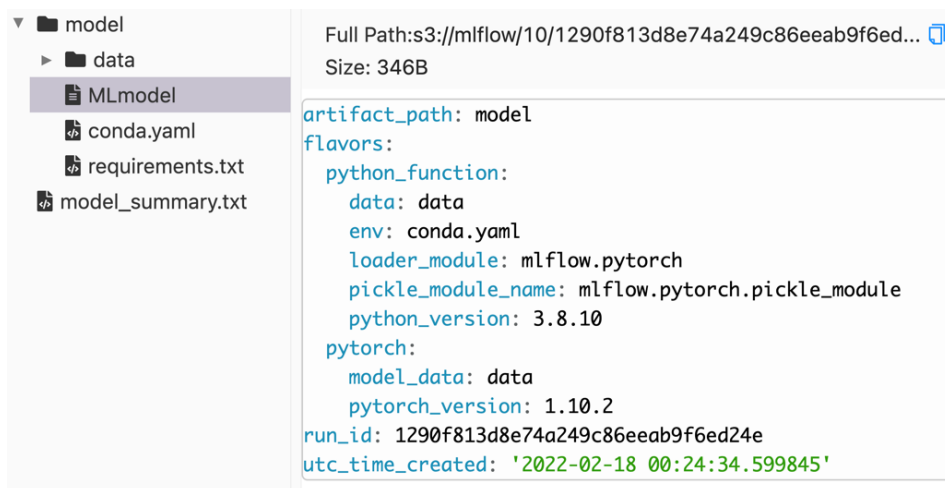


Figure 7.1 – MLmodel content for a fine-tuned PyTorch model

As can be seen from *Figure 7.1*, the flavor of this model is PyTorch. There are also a few other metadata about the model, such as the conda environment, which defines the dependencies for running the model, and many others. Given this self-contained information, it should be enough to allow MLflow to load the model back using the `mlflow.pytorch.load_model` API as follows:

```

logged_model = f'runs:{run_id}/model'
model = mlflow.pytorch.load_model(logged_model)

```

This will allow loading the model that was logged by an MLflow run with `run_id` back to memory and doing inference. Now imagine we have the following scenario where we need to add some preprocessing logic to check the language type of the input text. This requires loading a language detector model (<https://amitnness.com/2019/07/identify-text-language-python/>) such as the **FastText** language detector (<https://fasttext.cc/>), or Google's **Compact Language Detector v3** (<https://pypi.org/project/gclد3/>). Additionally, we also want to check whether there is any cached prediction for the exact same input. If it exists, then we should just return the cached result without invoking the expensive model prediction part. This is very typical preprocessing logic. For postprocessing, a common scenario is to return the prediction along with some metadata about the model URIs so that we can debug any potential prediction issue in production. Given this preprocessing and postprocessing logic, the inference pipeline now looks like the following figure:



Figure 7.2 – Multi-step inference pipeline

As can be seen from *Figure 7.2*, these five steps include the following:

- One original fine-tuned model for prediction (a PyTorch DL model)
- One additional language detection model that was not part of our previous training pipeline
- Cache operations (check cache and store to cache) for improving response performance
- One response message composition step

Rather than splitting these five steps into five different entry points in an **ML project** (recall that an entry point in an **ML project** can be arbitrary execution code in Python or other executables), it is much more elegant to compose this multi-step inference pipeline in a single entry point, since these steps are closely related to the model's prediction step. In addition, the advantage of encapsulating these closely related steps into a single inference pipeline is that we can save and load the inference pipeline as an MLmodel artifact. MLflow provides a generic way to implement this multi-step inference pipeline as a new Python model, without losing the flexibility of adding additional preprocessing and postprocessing capability if needed as shown in the following figure:

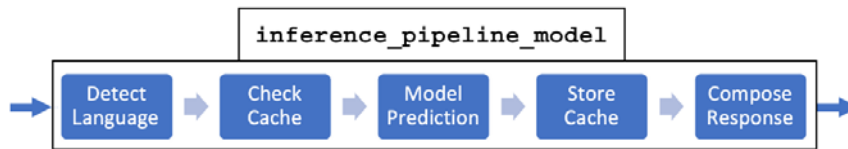


Figure 7.3 – Encapsulate the multi-step preprocessing and postprocessing logic into a new MLflow Python model

As can be seen from *Figure 7.3*, if we encapsulate the preprocessing and postprocessing logic into a new MLflow model called `inference_pipeline_model`, then we can load this entire inference pipeline as if it is just another model. This will also allow us to formalize the input and output format (called **Model Signature**) for the inference pipeline so that whoever wants to consume this inference pipeline will not need to guess what the format of the input and output is.

The mechanism to implement this at a high level is as follows:

1. First, create a custom MLflow pyfunc (Python function) model ([https://www.mlflow.org/docs/latest/python\\_api/mlflow.pyfunc.html#creating-custom-pyfunc-models](https://www.mlflow.org/docs/latest/python_api/mlflow.pyfunc.html#creating-custom-pyfunc-models)) to wrap the existing trained model. Specifically, we need to go beyond the built-in model flavors (<https://www.mlflow.org/docs/latest/models.html#built-in-model-flavors>) provided by MLflow and implement a new Python class that inherits from `mlflow.pyfunc.PythonModel` ([https://www.mlflow.org/docs/latest/python\\_api/mlflow.pyfunc.html#mlflow.pyfunc.PythonModel](https://www.mlflow.org/docs/latest/python_api/mlflow.pyfunc.html#mlflow.pyfunc.PythonModel)), defining `predict()` and, optionally, the `load_context()` methods in this new Python class.

In addition, we can specify the **Model Signature** (<https://mlflow.org/docs/latest/models.html#model-signature>) by defining the schema of a model's inputs and outputs. These schemas can be either column-based or tensor-based. It is highly recommended to implement these schemas for automatic input validation and model diagnosis in a production environment.

2. Then implement the preprocessing and postprocessing logic within this MLflow pyfunc. These could include caching, language detection, a response message, and any other logic that's needed.
3. Finally, implement the entry point in the ML project for the inference pipeline so that we can invoke the inference pipeline as if it is a single model artifact.

Now that we understand the fundamentals of MLflow's custom Python model to represent a multi-step inference pipeline, let's see how we can implement it for our NLP sentiment classification model with the preprocessing and postprocessing steps described in *Figure 7.3* in the following sections.

## Implementing a custom MLflow Python model

Let's first describe the steps to implement a custom MLflow Python model without any extra preprocessing and postprocessing logic:

1. First, make sure we have a trained DL model that's ready to be used for inference purposes. For the sake of learning in this chapter, we include the training pipeline **MLproject** in this chapter, so that we can easily produce a fine-tuned DL model. To run the training pipeline, make sure you have the virtual environment set up for this chapter by following the README file in this chapter's GitHub repository and *set up the environment variables* accordingly (<https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter07/README.md>). Then, in the command line, run the following command to generate a fine-tuned model in the local MLflow tracking server:

```
mlflow run . --experiment-name dl_model_chapter07 -P
pipeline_steps=download_data,fine_tuning_model
```

Once this is done, you will have a fine-tuned DL model logged in the MLflow tracking server. Now, we will use the logged model URI as the input for the inference pipeline since we will wrap it and save it as a new MLflow model. The logged model URI is something like the following, where the long random alphanumeric string is the `run_id` of the `fine_tuning_model` MLflow run, which you can find in the MLflow tracking server:

```
runs:/1290f813d8e74a249c86eeab9f6ed24e/model
```

2. Once you have a trained/fine-tuned model, we are ready to implement a new custom MLflow Python model as follows. You may want to check out the **VS Code** notebook for `basic_custom_dl_model.py` in the GitHub repo ([https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter07/notebooks/basic\\_custom\\_dl\\_model.py](https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter07/notebooks/basic_custom_dl_model.py)) to follow through the steps outlined here:

```
class InferencePipeline(mlflow.pyfunc.PythonModel):
    def __init__(self, finetuned_model_uri):
        self.finetuned_model_uri = finetuned_model_uri
```

```
def sentiment_classifier(self, row):
    pred_label = self.finetuned_text_classifier.
predict({row[0]})
    return pred_label

def load_context(self, context):
    self.finetuned_text_classifier = mlflow.pytorch.
load_model(self.finetuned_model_uri)

def predict(self, context, model_input):
    results = model_input.apply(
        self.sentiment_classifier, axis=1,
        result_type='broadcast')
    return results
```

Let's see what we have implemented. First, the `InferencePipeline` class inherits from the `MLflow.pyfunc.PythonModel` module, and implements four methods as follows:

- `predict`: This is a method that's required by `mlflow.pyfunc.PythonModel`, which returns the prediction result. Here, the `model_input` parameter is a pandas `DataFrame`, which contains a column with input text that needs to be classified. We leverage the pandas `DataFrame`'s `apply` method to run a `sentiment_classifier` method to score each row of the `DataFrame`'s text and the result is a `DataFrame` with each row being the predicted label. Since our original fine-tuned model does not accept a pandas `DataFrame` as input (it accepts a list of text strings as input), we need to implement a new classifier as a wrapper to the original model. That's the `sentiment_classifier` method. The other `context` parameter is the MLflow context to describe where the model artifact is stored. Since we will pass an MLflow logged model URI, this `context` parameter is not used in our implementation, as the logged model URI contains everything MLflow needs to load a model.
- `sentiment_classifier`: This is a wrapper method to allow each row of the input pandas `DataFrame` to be scored by calling the fine-tuned DL model's prediction function. Note that we are wrapping the first element of the row into a list so that the DL model can correctly use it as an input.



- `init`: This is a standard Python constructor method. Here, we use it to pass in a previously fine-tuned DL model URI, `finetuned_model_uri`, so that we can load it in the `load_context` method. Note that we do not want to directly load the model in the `init` method since it will cause a serialization issue (if you want to try, you will find out serializing a DL model naively is not a fun experience). Since the fine-tuned DL model is already serialized and deserialized through the `mlflow.pytorch` APIs, we should not reinvent the wheel here. The recommended way is to load the model in the `load_context` method.
  - `load_context`: This method is called when loading an MLflow model with the `mlflow.pyfunc.load_model` API. This is executed immediately after the Python model is constructed. Here, we load the fine-tuned DL model by using the `mlflow.pytorch.load_model` API. Note that whatever models are loaded in this method can use their corresponding deserializing methods. This will open doors for loading other models such as a language detection model, which could contain native code (for example, C++ code) that cannot be serialized using Python serialization protocols. This is one of the nice features provided by the MLflow model API framework.
3. Now that we have an MLflow custom model that can accept a column-based input, we can also define the model signature as follows:

```
input = json.dumps(['name': 'text', 'type': 'string'])
output = json.dumps(['name': 'text', 'type': 'string'])
signature = ModelSignature.from_dict({'inputs': input,
                                     'outputs': output})
```

This signature defines an input format with one named column called `text` with a datatype of `string`, and an output format with one named column called `text` with a datatype of `string`. The `mlflow.models.ModelSignature` class is used to create this signature object. This will be used when we log the new custom model in MLflow, as we will see in the next step.

4. Next, we can log this new custom model in MLflow as if this is a generic MLflow `pyfunc` model using the `mlflow.pyfunc.log_model` API as follows:

```
MODEL_ARTIFACT_PATH = 'inference_pipeline_model'
with mlflow.start_run() as dl_model_tracking_run:
    finetuned_model_uri =
    'runs:/1290f813d8e74a249c86eeab9f6ed24e/model'
    inference_pipeline_uri = f'runs:/{dl_model_tracking_
run.info.run_id}/{MODEL_ARTIFACT_PATH}'
    mlflow.pyfunc.log_model(
        artifact_path=MODEL_ARTIFACT_PATH,
```

```
conda_env=CONDA_ENV,  
python_model=InferencePipeline(  
    finetuned_model_uri),  
signature=signature)
```

The preceding code will log a model in the MLflow tracking server with a top-level folder named `inference_pipeline_model`, since we define the `MODEL_ARTIFACT_PATH` variable with this string value and assign this value to the `artifact_path` parameter of the `mlflow.pyfunc.log_model` method. The other three parameters we assign are the following:

- `conda_env`: This is to define the conda environment where this custom model will run. Here, we can pass the absolute path of the `conda.yaml` file in the root folder of this chapter defined by the `CONDA_ENV` variable (details of this variable can be found in the source code of this `basic_custom_dl_model.py` notebook on GitHub).
- `python_model`: Here, we call the new `InferencePipeline` class we just implemented and pass in the parameter of `finetuned_model_uri`. This way, the inference pipeline will load the correct fine-tuned model for prediction purposes.
- `signature`: We also pass the signature for both input and output we just defined and assign it to the `signature` parameter so that model input and output schema can be logged and enforced for validation purposes.

As a reminder, make sure you replace the `'runs:/1290f813d8e74a249c86eeab9f6ed24e/model'` value for the `finetuned_model_uri` variable with your own fine-tuned model URI generated in *step 1* so that the code will correctly load the original fine-tuned model.

- If you follow through the **VS Code** notebook for `basic_custom_dl_model.py` and run it cell by cell up to *step 4*, you should be able to find a newly logged model in the **Artifacts** section of the MLflow tracking server as shown in the following screenshot:

▼ Artifacts

Full Path: s3://mlflow/10/18a62e35629a4f35956033103dfea863/artifacts/inference\_pip... [Register Model](#)

### MLflow Model

The code snippets below demonstrate how to make predictions using the logged model. You can also [register it to the model registry](#) to version control

#### Model schema

Input and output schema for your model. [Learn more](#)

Name	Type
<b>Inputs (1)</b>	
text	string
<b>Outputs (1)</b>	
text	string

#### Make Predictions

Predict on a Spark DataFrame: [🔗](#)

```
import mlflow
logged_model = 'runs:/18a62e35629a4f35956033103dfea863/inference_pipeline_model'

# Load model as a Spark UDF.
loaded_model = mlflow.pyfunc.spark_udf(spark, model_uri=logged_model)

# Predict on a Spark DataFrame.
columns = list(df.columns)
df.withColumn('predictions', loaded_model(*columns)).collect()
```

Figure 7.4 – Inference MLflow model with model schema and a root folder of `inference_pipeline_model`

As can be seen from *Figure 7.4*, the root folder name (top left of the screenshot) is `inference_pipeline_model`, which is the `artifact_path` parameter's assigned value when calling `mlflow.pyfunc.log_model`. Note, if we do not specify the `artifact_path` parameter, by default it will be just `model`. You can confirm this by just looking at *Figure 7.1* earlier in this chapter. Also note that now there is a **Model schema** section as shown in *Figure 7.4*, which is new. This describes both the input and output format as we defined before. In fact, if we click the `MLmodel` file under the `inference_pipeline_model` folder, we can see the full content as follows:

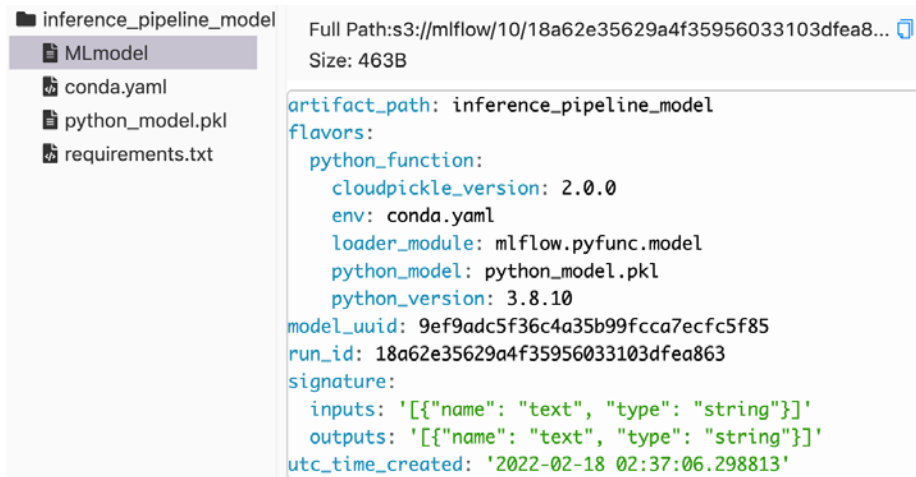


Figure 7.5 – The content of `inference_pipeline_model`'s `MLmodel` file

As can be seen from *Figure 7.5*, the content of the `MLmodel` file now contains a `signature` section near the bottom, a new section compared with *Figure 7.1*. However, there are some more important differences in terms of the model flavor. The flavor of `inference_pipeline_model` is a generic `mlflow.pyfunc.model` model, not a `mlflow.pytorch` model anymore. In fact, if you compare *Figure 7.5* with *Figure 7.1*, which is our PyTorch fine-tuned DL model, there is a section about `pytorch` and its `model_data` and `pytorch_version`, which has now completely disappeared in *Figure 7.5*. For MLflow, it has no knowledge of the original model, which is a PyTorch model, but just a generic MLflow `pyfunc` model as the newly wrapped model. This is great news since now we only need one generic MLflow `pyfunc` API to load the model, regardless of how complex the wrapped model is and how many more preprocessing and postprocessing steps are inside this generic `pyfunc` model when we implement it in the next section.

6. We now can load `inference_pipeline_model` using the generic `mlflow.pyfunc.load_model` to load the model and do prediction with an input `pandas DataFrame` as follows:

```
input = {"text": ["what a disappointing movie", "Great
movie"]}
input_df = pd.DataFrame(input)
with mlflow.start_run():
    loaded_model = \
mlflow.pyfunc.load_model(inference_pipeline_uri)
    results = loaded_model.predict(input_df)
```

Here, `inference_pipeline_uri` is the URI produced in *step 4* as the unique identifier for `inference_pipeline_model`. For example, an `inference_pipeline_uri` value could look as follows:

```
'runs:/6edf6013d2454f7f8a303431105f25f2/inference_pipeline_model'
```

Once the model is loaded, we can just call the `predict` function to score the `input_df` DataFrame. This calls the `predict` function of our newly implemented `InferencePipeline` class, as described in *step 2*. The results will look something like the following:

```
print(results)
      text
0  negative
1  positive
```

Figure 7.6 – Output of the inference pipeline in a pandas DataFrame format

If you see the prediction results like in *Figure 7.6*, then you should feel proud that you have just implemented a working custom MLflow Python model that has enormous flexibility and power to enable us to implement preprocessing and postprocessing logic without changing any of the logging and loading model parts, as we will see in the next section.

#### Creating a New Flavor of MLflow Custom Model

As shown in this chapter, we can build a wrapped MLflow custom model using an already trained model for inference purposes. It should be noted that it is also possible to build an entirely new flavor of MLflow custom model for training purposes. This is needed when you have a model that's not yet supported by the built-in MLflow model flavors. For example, if you want to train a brand new **FastText** model based on your own corpus but as of MLflow version 1.23.1, there is no **FastText** MLflow model flavor yet, then you can build a new **FastText** MLflow model flavor (see reference: <https://medium.com/@pennyqxr/how-save-and-load-fasttext-model-in-mlflow-format-37e4d6017bf0>). Interested readers can also find more references in the *Further reading* section at the end of this chapter.

## Implementing preprocessing and postprocessing steps in a DL inference pipeline

Now that we have a basic generic MLflow Python model that can do prediction on an input pandas DataFrame and produce output in another pandas DataFrame, we are ready to tackle the multi-step inference scenario described before. Note that while the initial implementation in the previous section might not look earth-shaking, this opens doors for implementing preprocessing and postprocessing logic that was not possible before while maintaining the capability of using the generic `mlflow.pyfunc.log_model` and `mlflow.pyfunc.load_model` to treat the entire inference pipeline as a generic `pyfunc` model, regardless of how complex the original DL model is and how many additional preprocessing and postprocessing steps there are. Let's see how we can do this in this section. You may want to check out the VS Code notebook for `multistep_inference_model.py` from GitHub ([https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter07/notebooks/multistep\\_inference\\_model.py](https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter07/notebooks/multistep_inference_model.py)) to follow through the steps in this section.

In *Figure 7.3*, we depicted two preprocessing steps prior to the model prediction, and two postprocessing steps after the model prediction. So where and how do we add the preprocessing and postprocessing logic while keeping this entire inference pipeline as a single MLflow model? It turns out the main changes will happen in the `InferencePipeline` class implemented in the previous section. Let's walk through the implementation and changes step by step in the following subsections.

### Implementing language detection preprocessing logic

Let's first implement the language detection preprocessing logic:

1. To detect the language type of the input text, we can use Google's **Compact Language Detector v3**. Note that this language detector is a neural network model that contains native code (the core implementation is in C++) with a Python binding so that we can use it in a Python environment (<https://github.com/google/cld3>). As this model cannot be serialized by using Python serialization protocols such as pickle, it would be a major challenge to figure out how to package this in an MLflow `pyfunc` model. The good news is that MLflow's `load_context` method allows us to load this model without worrying about serialization and deserialization. We only need to add two lines of code in the `load_context` method in the `InferencePipeline` class as follows to load the language detector model:

```
import gld3
self.detector = gld3.NNetLanguageIdentifier()
```

```
min_num_bytes=0,  
max_num_bytes=1000)
```

The preceding two lines are added into the `load_context` method, along with the preexisting statement that loads the fine-tuned DL model for sentiment classification. This will allow the language detector to be loaded as soon as the initialization of the `InferencePipeline` class is done. This language detector will use up to the first 1,000 bytes of the input to determine the language type. Once this language detector is loaded, then we can use it to detect the language in a preprocessing method.

2. In a preprocessing method for language detection, we will accept each row of the input text, detect the language, and return the language type as a string as follows:

```
def preprocessing_step_lang_detect(self, row):  
    language_detected = \  
self.detector.FindLanguage(text=row[0])  
    if language_detected.language != 'en':  
        print("found Non-English Language text!")  
    return language_detected.language
```

The implementation is straightforward. We also add a printout to see if we see any non-English text in the input to the console. If your business logic requires you to implement any preemptive actions when dealing with some specific language, then you can add more logic in this method. Here, we just return the language type detected.

3. Then, in the `sentiment_classifier` method that scores each row of the input, we can just add one line prior to the prediction to first detect the language as follows:

```
language_detected = self.preprocessing_step_lang_  
detect(row)
```

Later, we pass along the `language_detected` variable to the response as we will see in the postprocessing logic implementation.

And that's all it takes to implement the language detection as a preprocessing step in the inference pipeline.

Now let's see how to implement the other step: cache, which requires both preprocessing (detecting if there are any preexisting matched prediction results for the same input) and postprocessing (storing a key-value pair of input and prediction results in the cache).

## Implementing caching preprocessing and postprocessing logic

Let's see how we can implement caching in the `InferencePipeline` class:

1. We can add a new statement to initialize the cache store in the `init` method, as this has no problem being serialized or deserialized:

```
from cachetools import LRUCache
self.cache = LRUCache(100)
```

This will initialize a Least Recently Used cache with 100 objects stored.

2. Next, we will add a preprocessing method to detect if any input is in the cache:

```
def preprocessing_step_cache(self, row):
    if row[0] in self.cache:
        print("found cached result")
        return self.cache[row[0]]
```

If it finds the exact input row as a key already in the cache, then it returns the cached value.

3. In the `sentiment_classifier` method, we can add the preprocessing step to check the cache and if it finds the cache, then it will immediately return the cached response without invoking the expensive DL model classifier:

```
cached_response = self.preprocessing_step_cache(row)
if cached_response is not None:
    return cached_response
```

This preprocessing step should be placed as the first step in the `sentiment_classifier` method, before doing language detection and model prediction. This can significantly speed up real-time prediction responses when there are many duplicated inputs.

4. Also in the `sentiment_classifier` method, we need to add a postprocessing step to store new input and prediction responses in the cache:

```
self.cache[row[0]] = response
```

That's it. We have successfully added caching as a preprocessing and postprocessing step in the `InferencePipeline` class.



## Implementing response composition postprocessing logic

Now let's see how we can implement the response composition logic as a postprocessing step after the original DL model prediction is invoked and the result is returned. Just returning a prediction label of `positive` or `negative` usually is not enough, as we would like to know which version of the model was used and what language was detected for debugging and diagnosis in the production environment. The response to the caller of the inference pipeline will no longer be a plain string, but rather a serialized JSON string. Follow these steps to implement this postprocessing logic:

1. In the `init` method of the `InferencePipeline` class, we need to add a new `inference_pipeline_uri` parameter, so that we can capture this generic MLflow `pyfunc` model's reference for provenance tracking purposes. Both the `finetuned_model_uri` and `inference_pipeline_uri` parameters will be part of the response's JSON object. The `init` method now looks like the following:

```
def __init__(self,
              finetuned_model_uri,
              inference_pipeline_uri=None):
    self.cache = LRUCache(100)
    self.finetuned_model_uri = finetuned_model_uri
    self.inference_pipeline_uri = inference_pipeline_uri
```

2. In the `sentiment_classifier` method, add a new postprocessing statement to compose a new response based on the language detected, predicted label, and the model metadata including both `finetuned_model_uri` and `inference_pipeline_uri`:

```
response = json.dumps({
    'response': {
        'prediction_label': pred_label
    },
    'metadata': {
        'language_detected': language_
detected,
    },
    'model_metadata': {
        'finetuned_model_uri': self.
finetuned_model_uri,
        'inference_pipeline_model_uri': self.
inference_pipeline_uri
    },
})
```

Note that we use `json.dumps` to encode a nested Python string object into a JSON formatted string, so that the caller can easily parse out the response using JSON tools.

3. In the `mlflow.pyfunc.log_model` statement, we need to add a new `inference_pipeline_uri` parameter when calling the `InferencePipeline` class:

```
mlflow.pyfunc.log_model(  
    artifact_path=MODEL_ARTIFACT_PATH,  
    conda_env=CONDA_ENV,  
    python_model=InferencePipeline(finetuned_model_uri,  
    inference_pipeline_uri),  
    signature=signature)
```

This will log a new inference pipeline model with all the additional processing logic we implemented. This completes the implementation of the multi-step inference pipeline depicted in *Figure 7.3*.

Note that once the model is logged with all these new steps, to consume this new inference pipeline, that's to say, to load this model, requires zero code changes. We can load the newly logged model the same way as before:

```
loaded_model = mlflow.pyfunc.load_model(inference_pipeline_uri)
```

If you have followed through the steps up until now, you should also run the VS Code notebook for `multistep_inference_model.py` cell by cell up to *step 3* described in this subsection. Now we can try to use this new multi-step inference pipeline to test it out. We can prepare a new set of input data where there are duplicates and a non-English text string as follows:

```
input = {"text": ["what a disappointing movie", "Great movie",  
    "Great movie", "很好看的电影"]}  
input_df = pd.DataFrame(input)
```

This input includes two duplicated entries (`Great movie`) and one Chinese text string (the last element in the input list, where the meaning of the Chinese text is the same as `Great Movie`). Now we can just load the model and call `results = loaded_model.predict(input_df)` as before. And during the execution of this `predict` statement, you should see the following two statements in the console output:

```
found cached result
found Non-English language text.
```

This means that our caching and language detector works!

We can also print out the results to double-check whether our multi-step pipeline works or not using the following code:

```
for i in range(results.size):
    print(results['text'][i])
```

This will print out the full content for each row of the response. Here, we display the output for the last one (which has the Chinese text) as an example:

```
{
  "response": {
    "prediction_label": [
      "negative"
    ]
  },
  "metadata": {
    "language_detected": "zh"
  },
  "model_metadata": {
    "finetuned_model_uri": "runs:/1290f813d8e74a249c86eeab9f6ed24e/model",
    "inference_pipeline_model_uri": "runs:/9690b4cc873a4ebf8640390ec6baa0e1/inference_pipeline_model"
  }
}
```

Figure 7.7 – JSON response for the Chinese text string input using the multi-step inference pipeline

As can be seen in *Figure 7.7*, `prediction_label` is included in the response (which is negative). Since we have been using **TinyBERT** for the English language only, this incorrect prediction is expected. If we switch to a multilingual pretrained language model such as **bert-base-multilingual-uncased** (<https://huggingface.co/bert-base-multilingual-uncased>) as the foundation model during model training and fine-tuning, then supporting inference for multiple languages is possible. In fact, the multilingual version of BERT supports 102 world languages. If we look at the `language_detected` field under the `metadata` section in the JSON response, we see the string "zh", which represents the Chinese language. This is what the language detector produced in the preprocessing step. Additionally, the `model_metadata` section includes both the original `finetuned_model_uri` and `inference_pipeline_model_uri`. These are MLflow tracking server-specific URIs that we can use to uniquely trace and identify which fine-tuned model and inference pipeline was used for this prediction result. This is very important for provenance tracking and diagnosis analysis in the production environment. Comparing this complete JSON output with the earlier prediction label output in *Figure 7.6*, this has much richer contextual information for the consumer of the inference pipeline to use.

If you see the JSON output in your notebook run like *Figure 7.7*, give yourself a round of applause, because you have just completed a big milestone in implementing a multi-step inference pipeline that can be reused and deployed into production for realistic business scenarios.

## Implementing an inference pipeline as a new entry point in the main MLproject

Now that we have successfully implemented a multi-step inference pipeline as a new custom MLflow model, we can go one step further by incorporating this as a new entry point in the main **MLproject** so that we can run the following entire pipeline end to end (*Figure 7.8*). Check out this chapter's code from GitHub to follow through and run the pipeline in your local environment.



Figure 7.8 – End-to-end pipeline using MLproject

We can add the new entry point `inference_pipeline_model` into the `MLproject` file. You can check out this file on the GitHub repository (<https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLflow/blob/main/chapter07/MLproject>):

```
inference_pipeline_model:
  parameters:
    finetuned_model_run_id: { type: str, default: None }
  command: "python pipeline/inference_pipeline_model.py
  --finetuned_model_run_id {finetuned_model_run_id}"
```

This entry point or step can be invoked either standalone or as part of the entire pipeline depicted in *Figure 7.8*. As a reminder, make sure you have set up the environment variables as described in the README file of this chapter for the MLflow tracking server and backend storage URIs before you execute the MLflow run commands. This step logs and registers a new `inference_pipeline_model`, which itself contains multi-step preprocessing and postprocessing logic. The following command can be used to run this step at the root level of the `chapter07` folder, if you know the `finetuned_model_run_id`:

```
mlflow run . -e inference_pipeline_model --experiment-
name dl_model_chapter07 -P finetuned_model_run_
id=07b900a96af04037a956c74ef691396e
```

This will not only log a new `inference_pipeline_model` in the MLflow tracking server but will also register a new version of `inference_pipeline_model` in the MLflow model registry. You can find the registered `inference_pipeline_model` in your local MLflow server with the following link:

```
http://localhost/#/models/inference\_pipeline\_model/
```

As an example, a registered `inference_pipeline_model` version 6 is shown in the following screenshot:

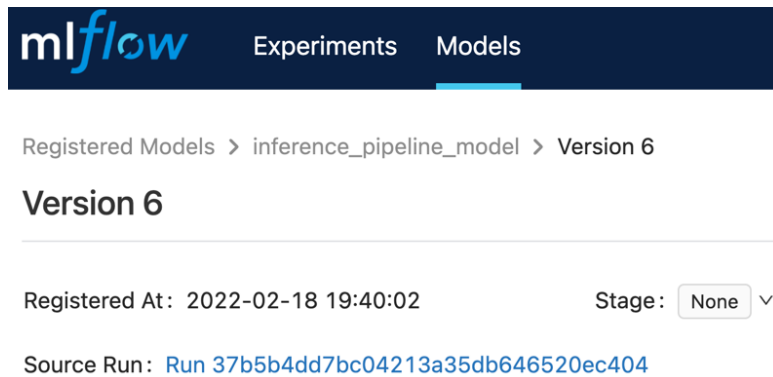


Figure 7.9 – A registered `inference_pipeline_model` at version 6

You can also run the entire end-to-end pipeline depicted in *Figure 7.8* as follows:

```
mlflow run . --experiment-name dl_model_chapter07
```

This will run all the steps in this end-to-end pipeline and finish with a logged and registered `inference_pipeline_model` in the model registry.

The implementation of the Python code for `inference_pipeline_model.py`, which is executed when the entry point `inference_pipeline_model` is invoked, is basically copying the `InferencePipeline` class we implemented in the VS Code notebook for `multistep_inference_model.py` with a couple of small changes as follows:

- Adding a task function to be executed as a parameterized entry point for this step:

```
def task(finetuned_model_run_id, pipeline_run_name):
```

What this function does is starting a new MLflow run to log and register a new inference pipeline model.

- Turning on the model registration while logging as follows:

```
mlflow.pyfunc.log_model(
    artifact_path=MODEL_ARTIFACT_PATH,
    conda_env=CONDA_ENV,
    python_model=InferencePipeline(
        finetuned_model_uri,
        inference_pipeline_uri),
    signature=signature,
    registered_model_name=MODEL_ARTIFACT_PATH)
```

Note that we assign to `registered_model_name` the value of `MODEL_ARTIFACT_PATH`, which is `inference_pipeline_model`. This enables the model to be registered under this name in the MLflow model registry, as seen in *Figure 7.9*.

The complete code for this new entry point can be found in the GitHub repository: [https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter07/pipeline/inference\\_pipeline\\_model.py](https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter07/pipeline/inference_pipeline_model.py).

Note that we also need to add a new section in the `main.py` file to allow the `inference_pipeline_model` entry point to also be callable from within the `main` entry point. The implementation is straightforward, just like adding other steps previously as described in *Chapter 4, Tracking Code and Data Versioning*. Interested readers should check out the `main.py` file from GitHub to take a look at the implementation: <https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter07/main.py>.

This concludes the implementation of adding a new entry point in the **MLproject** so that we can run the multi-step inference pipeline creation, logging, and registering using the MLflow run command tool.

## Summary

In this chapter, we covered a very important topic on creating a multi-step inference pipeline using MLflow's custom Python model approach, namely `mlflow.pyfunc.PythonModel`.

We discussed four patterns of inference workflow in production where usually a single trained model is not enough to complete the business application requirements. It is highly likely some preprocessing and postprocessing logic is not seen during the model training and development stage. That's why MLflow's `pyfunc` approach is an elegant approach to implementing a custom MLflow model that can wrap a trained DL model with additional preprocessing and postprocessing logic.

We successfully implemented an inference pipeline model that wraps our DL sentiment classifier with language detection using Google's Compact Language Detector, caching, and additional model metadata in addition to the prediction label. We went one step further to incorporate the inference pipeline model creation step into the end-to-end model development workflow so that we can produce a registered inference pipeline model with one MLflow run command.

The skills and lessons learned in this chapter will be critical for anyone who wants to implement a real-world inference pipeline using the MLflow `pyfunc` approach. This also opens doors for supporting flexible and powerful deployment into production scenarios, which we will cover in the next chapter.

## Further reading

- *MLflow Models* (MLflow documentation): <https://www.mlflow.org/docs/latest/models.html#>
- *Implementing the statsmodels flavor in MLflow*: <https://blog.stratio.com/implementing-the-statsmodels-flavor-in-mlflow/>
- *InferLine: ML inference Pipeline Composition Framework*: <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-76.pdf>
- *Batch Inference vs Online Inference*: <https://mlinproduction.com/batch-inference-vs-online-inference/>
- *Lessons from building a small MLOps pipeline*: <https://www.nestorsag.com/blog/lessons-from-building-a-small-ml-ops-pipeline/>
- *Text summarizer on Hugging Face with MLflow*: <https://vishsubramanian.me/hugging-face-with-mlflow/>





# 8

# Deploying a DL Inference Pipeline at Scale

Deploying a **deep learning (DL)** inference pipeline for production usage is both exciting and challenging. The exciting part is that, finally, the DL model pipeline can be used for prediction with real-world production data, which will provide real value to the business scenarios. However, the challenging part is that there are different DL model serving platforms and host environments. It is not easy to choose the right framework for the right model serving scenarios, which can minimize deployment complexity but provide the best model serving experiences in a scalable and cost-effective way. This chapter will cover the topics as an overview of different deployment scenarios and host environments, and then provide hands-on learning on how to deploy to different environments, including local and remote cloud environments using MLflow deployment tools. By the end of this chapter, you should be able to confidently deploy an MLflow DL inference pipeline to various host environments for either batching or real-time inference services.

In this chapter, we're going to cover the following main topics:

- Understanding the landscape of deployment and hosting environments
- Deploying locally for batch and web service inference
- Deploying using Ray Serve and MLflow deployment plugins
- Deploying to AWS SageMaker – a complete end-to-end guide

## Technical requirements

The following items are required for this chapter's learning:

- GitHub repository code for this chapter: <https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/tree/main/chapter08>.
- Ray serve and mlflow-ray-serve plugin: <https://github.com/ray-project/mlflow-ray-serve>.
- AWS SageMaker: You will need to have an AWS account. You can create a free AWS account easily through the free signup website at <https://aws.amazon.com/free/>.
- AWS **command-line interface (CLI)**: <https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html>.
- Docker Desktop: <https://www.docker.com/products/docker-desktop/>.
- Complete the example in *Chapter 7, Multi-Step Deep Learning Inference Pipeline*, of this book. This will give you a ready-to-deploy inference pipeline to use in this chapter.

## Understanding different deployment tools and host environments

There are different deployment tools in the MLOps technology stack that have different target use cases and host environments for deploying different model inference pipelines. In *Chapter 7, Multi-Step Deep Learning Inference Pipeline*, we learned the different inference scenarios and requirements and implemented a multi-step DL inference pipeline that can be deployed into a model hosting/serving environment. Now, we will learn how to deploy such a model to a few specific model hosting and serving environments. This is visualized in *Figure 8.1* as follows:

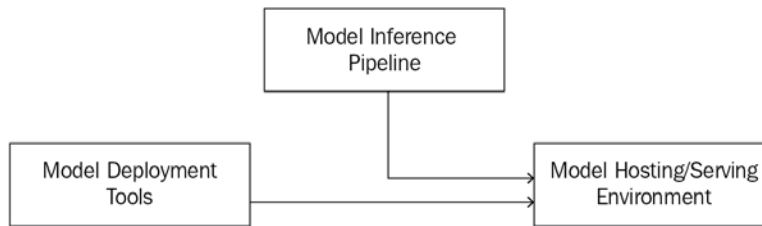


Figure 8.1 – Using model deployment tools to deploy a model inference pipeline to a model hosting and serving environment

As can be seen from *Figure 8.1*, there can be different deployment tools for different model hosting and serving environments. Here, we list the three typical scenarios as follows:

- **Batch inference at scale:** If we want to do batch inference at a regular schedule, we can use the PySpark **user defined function (UDF)** to load an MLflow model flavor to do this, since we can leverage Spark's scalable computational approach on a distributed cluster (<https://mlflow.org/docs/latest/models.html#export-a-python-function-model-as-an-apache-spark-udf>). We will show an example of how to do this in the next section.
- **Streaming inference at scale:** This usually requires an endpoint that hosts the **Model as a Service (MaaS)**. There exist quite a few tools and frameworks for production-grade deployment and model serving. We will compare a few tools in this section to understand how they work and how well they integrate with MLflow before we start learning how to do this type of deployment in this chapter.
- **On-device model inference:** This is an emerging area called **TinyML**, which deploys ML/DL models in a resource-limited environment such as mobile, sensor, or edge device (<https://www.kdnuggets.com/2021/11/on-device-deep-learning-pytorch-mobile-tensorflow-lite.html>). Two popular frameworks are PyTorch Mobile (<https://pytorch.org/mobile/home/>) and TensorFlow Lite (<https://www.tensorflow.org/lite>). This is not the focus of this book. You are encouraged to check out some further reading for this area at the end of this chapter.

Now, let's look at what kind of tools are available for deploying model inference as a service, especially those tools that have support for MLflow model deployment.

There are three types of model deployment and serving tools, as follows:

- **MLflow built-in model deployment:** This comes out of the box from MLflow releases, which includes deployments to a local web server, AWS SageMaker, and Azure ML. There is also a managed MLflow on Databricks that supports model serving in public review as of this writing, which we will not cover in this book since this is well presented in the official Databricks documentation (interested readers should look up the official documentation on this Databricks feature at this website: <https://docs.databricks.com/applications/mlflow/model-serving.html>). However, we will show you how to use the MLflow built-in model deployment to deploy to local and remote AWS SageMaker in this chapter.
- **MLflow custom deployment plugins:** MLflow provides an API for deploying to custom serving environments through MLflow deployment plugins (<https://mlflow.org/docs/latest/plugins.html#deployment-plugins>). Examples include `mlflow-torchserv` (<https://github.com/mlflow/mlflow-torchserve>), `mlflow-ray-serve` (<https://github.com/ray-project/mlflow-ray-serve>), and `mlflow-triton-plugin` (<https://github.com/triton-inference-server/server/tree/v2.17.0/deploy/mlflow-triton-plugin>). We will show how to use the `mlflow-ray-serve` plugin for deployment in this chapter.
- **Model serving tools that can deploy MLflow model flavors:** These are usually generic model serving frameworks that support many types of models, including MLflow model flavors. Examples include **Seldon MLServer** (<https://docs.seldon.io/projects/seldon-core/en/latest/servers/mlflow.html>), **Ray Serve** (<https://docs.ray.io/en/latest/serve/ml-models.html#integration-with-model-registries>) and **Triton Inference Server**; only two MLflow model flavors – **Open Neural Network Exchange (ONNX)** and **Triton** – are supported at the time of writing (<https://developer.nvidia.com/nvidia-triton-inference-server>). We will show you how to use **Ray Serve** together with the `mlflow-ray-serve` plugin to deploy the MLflow Python model. Note that, although in this book we show how to use an MLflow customized plugin to deploy with a generic ML serve tool such as Ray Serve, it is important to note that a generic ML serve tool can do much more with or without an MLflow customized plugin.

### Optimize DL Inference through Specialized Inference Engines

There are some special MLflow model flavors such as **ONNX** (<https://onnx.ai/>) and **TorchScript** (<https://huggingface.co/docs/transformers/v4.17.0/en/serialization#torchscript>) that are specially designed for DL model inference runtime. We can convert a DL model into an ONNX model flavor (<https://github.com/microsoft/onnxruntime>) or a TorchScript server (<https://pytorch.org/serve/>). As both ONNX and TorchScript are still evolving and are specifically designed for the original DL model part, but not the entire inference pipeline, we are not covering them in this chapter.

Now that we have a good understanding of the varieties of the deployment tools and model serving frameworks, let's learn how to do the deployment in the following sections with concrete examples.

## Deploying locally for batch and web service inference

For development and testing purposes, we usually need to deploy our model locally to verify it works as expected. Let's see how to do it for two scenarios: batch inference and web service inference.

### Batch inference

For batch inference, follow these instructions:

1. Make sure you have completed *Chapter 7, Multi-Step Deep Learning Inference Pipeline*. This will produce an MLflow `pyfunc` DL inference model pipeline URI that can be loaded using standard MLflow Python functions. The logged model can be uniquely located by the `run_id` and model name as follows:

```
logged_model = 'runs:/37b5b4dd7bc04213a35db646520ec404/inference_pipeline_model'
```

The model can also be identified by the model name and version number using the model registry as follows:

```
logged_model = 'models:/inference_pipeline_model/6'
```

2. Follow the instructions under the *Batch inference at-scale using PySpark UDF function* section of this README.md file (<https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter08/README.md>) to set up the local virtual environment, a full-fledged MLflow tracking server, and a few environment variables so that we can execute the code on your local environment.
3. Load the model with the MLflow `mlflow.pyfunc.spark_udf` API to create a PySpark UDF function as follows. You may want to check out the `batch_inference.py` file from GitHub to follow through ([https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter08/batch/batch\\_inference.py](https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter08/batch/batch_inference.py)):

```
loaded_model = mlflow.pyfunc.spark_udf(spark, model_
uri=logged_model, result_type=StringType())
```

This will wrap the inference pipeline as a PySpark UDF function with a return result type of `String`. This is because our model inference pipeline has a model signature requiring the output as a string type column.

4. Now, we can apply the PySpark UDF function to the input `DataFrame`. Note that the input `DataFrame` must have a `text` column with a `string` data type since that's what the model signature requires:

```
df = df.withColumn('predictions', loaded_model())
```

Because our model inference pipeline has defined a model signature, we don't need to specify any column parameters if it finds the `text` column in the input `DataFrame`, which is `df` in this example. Note that we can read a large volume of data using Spark's `read` API, which supports different data format reading, such as CSV, JSON, Parquet, and many more. In our example, we read the `test.csv` file from the IMDB dataset. This will leverage Spark's powerful distributed computation on a cluster if we have a large volume of data. This enables us to do batch inference at scale effortlessly.

5. To run the batch inference code from end to end, you should check out the complete code provided in the repository at this location: [https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter08/batch/batch\\_inference.py](https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter08/batch/batch_inference.py). Make sure you replace the `logged_model` variable with your own `run_id` and model name or the registered model name and version before you run the following command in the `batch` folder:

```
python batch_inference.py
```

6. You should see the output in *Figure 8.2* on the screen:

```

± main U:1 7:8 *1 + python batch_inference.py
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
22/03/06 21:31:53 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform...
found Non-English language text. (0 + 1) / 1]
found Non-English language text.
found cached result
found Non-English language text.
found Non-English language text.
found Non-English language text.
found Non-English language text.
found Non-English language text.
found Non-English language text.
found Non-English language text.
found Non-English language text.
found Non-English language text.
found Non-English language text.
found Non-English language text.
-----RECORD 0-----
text      | Turgid dialogue, feeble characterization - Harvey Keitel a judge? He plays mo...
predictions | {"response": {"prediction_label": ["positive"]}, "metadata": {"language_detec...
-----RECORD 1-----
text      | The worst movie in the history of cinema. I don't know if it was trying to be...
predictions | {"response": {"prediction_label": ["positive"]}, "metadata": {"language_detec...
-----RECORD 2-----
text      | "I mean, you just have to love the Italian film industry. Someone came up wit...
predictions | {"response": {"prediction_label": ["positive"]}, "metadata": {"language_detec...
-----RECORD 3-----
text      | "I'm a big fan of Kevin Spacey's work, but this is a sub-standard film. If yo...
predictions | {"response": {"prediction_label": ["positive"]}, "metadata": {"language_detec...
-----RECORD 4-----
text      | An exquisite film. They just don't make them like this any more! We eavesdrop...
predictions | {"response": {"prediction_label": ["positive"]}, "metadata": {"language_detec...
-----RECORD 5-----
text      | "In an otherwise good review, loleralacartelort7890 says ""The truth is that ...
predictions | {"response": {"prediction_label": ["positive"]}, "metadata": {"language_detec...
-----RECORD 6-----
text      | "Talk about a dream cast - just two of the most wonderful actors who ever app...
predictions | {"response": {"prediction_label": ["positive"]}, "metadata": {"language_detec...
-----RECORD 7-----
text      | "Previous comments encouraged me to check this out when it showed up on TCM, ...
predictions | {"response": {"prediction_label": ["positive"]}, "metadata": {"language_detec...
-----RECORD 8-----
text      | "While the idea is more original than most Sci-Fi movies, the execution is, a...
predictions | {"response": {"prediction_label": ["positive"]}, "metadata": {"language_detec...
-----RECORD 9-----
text      | The story of a Volcano erupting downtown L.A. sounds like a nice plot for a d...
predictions | {"response": {"prediction_label": ["positive"]}, "metadata": {"language_detec...
only showing top 10 rows

```

Figure 8.2 – Batch inference using PySpark UDF function

As can be seen from *Figure 8.2*, the multi-step inference pipeline we loaded worked correctly and even detected non-English texts and duplicates, although the language detector probably produced some false positives. The output is a two-column DataFrame where the JSON response of the model prediction is saved in the `predictions` column. Note that you can use the same code provided in `batch_inference.py` in a Databricks notebook and process a very large volume of input data with a Spark cluster by changing the input data and the logged model location.

Now that we know how to do batch inference at scale, let's see how to deploy to a local web service for the same model inference pipeline.



## Model as a web service

We can deploy the same logged model inference pipeline to a web service locally and have an endpoint that accepts HTTP requests with an HTTP response.

The local deployment is quite straightforward with just one command line. We can deploy a logged model or a registered model using the model URI as in the previous batch inference, as follows:

```
mlflow models serve -m models:/inference_pipeline_model/6
```

You should be able to see the following:

```
2022/03/06 21:50:19 INFO mlflow.models.cli: Selected backend
for flavor 'python_function'
2022/03/06 21:50:21 INFO mlflow.utils.conda: === Creating conda
environment mlflow-a0968092d20d039088e2875ad04bbaa0f3a75206 ===
± |main U:1 ? :8 X| done
Solving environment: done
```

This will create the conda environment using the logged model so that it will have all the dependencies to run. After the conda environment is created, you should see the following:

```
2022/03/06 21:52:11 INFO mlflow.pyfunc.backend: === Running
command 'source /Users/yongliu/opt/miniconda3/bin/./etc/
profile.d/conda.sh && conda activate mlflow-a0968092d20d039
088e2875ad04bbaa0f3a75206 1>&2 && gunicorn --timeout=60 -b
127.0.0.1:5000 -w 1 ${GUNICORN_CMD_ARGS} -- mlflow.pyfunc.
scoring_server.wsgi:app'
[2022-03-06 21:52:12 -0800] [97554] [INFO] Starting gunicorn
20.1.0
[2022-03-06 21:52:12 -0800] [97554] [INFO] Listening at:
http://127.0.0.1:5000 (97554)
[2022-03-06 21:52:12 -0800] [97554] [INFO] Using worker: sync
[2022-03-06 21:52:12 -0800] [97561] [INFO] Booting worker with
pid: 97561
```

Now, the model is deployed as a web service and ready to accept HTTP requests for model prediction. Open a different Terminal window and type the following command to invoke the model web service to get a prediction response:

```
curl http://127.0.0.1:5000/invocations -H 'Content-Type: application/json' -d '{
  "columns": ["text"],
  "data": [{"This is the best movie we saw."}, {"What a movie!"}]
}'
```

We can see the following prediction response immediately:

```
[{"text": "{\"response\": {\"prediction_label\": [\"positive\"]}, \"metadata\": {\"language_detected\": \"en\"}, \"model_metadata\": {\"finetuned_model_uri\": \"runs:/07b900a96af04037a956c74ef691396e/model\", \"inference_pipeline_model_uri\": \"runs:/37b5b4dd7bc04213a35db646520ec404/inference_pipeline_model\"}}"}, {"text": "{\"response\": {\"prediction_label\": [\"positive\"]}, \"metadata\": {\"language_detected\": \"en\"}, \"model_metadata\": {\"finetuned_model_uri\": \"runs:/07b900a96af04037a956c74ef691396e/model\", \"inference_pipeline_model_uri\": \"runs:/37b5b4dd7bc04213a35db646520ec404/inference_pipeline_model\"}}"}]
```

If you have followed the steps so far and saw the prediction results, you should feel very proud that you just deployed a DL model inference pipeline into a local web service! This is great for testing and debugging, and the behavior of the model will not change on a production web server, so we should make sure it works on a local web server.

So far, we have learned how to use the built-in MLflow deployment tool. Next, we will see how to use a generic deployment tool, Ray Serve, to deploy an MLflow inference pipeline.

## Deploying using Ray Serve and MLflow deployment plugins

A more generic way to do deployment is to use a framework such as Ray Serve (<https://docs.ray.io/en/latest/serve/index.html>). Ray Serve has several advantages, such as DL model frameworks agnostics, native Python support, and supporting complex model composition inference patterns. Ray Serve supports all major DL frameworks and any arbitrary business logic. So, can we leverage both Ray Serve and MLflow to do model deployment and serve? The good news is that we can use the MLflow deployment plugins provided by Ray Serve to do this. Let's walk through how to use the `mlflow-ray-serve` plugin to do MLflow model deployment using Ray Serve (<https://github.com/ray-project/mlflow-ray-serve>). Before we begin, we need to install the `mlflow-ray-serve` package:

```
pip install mlflow-ray-serve
```

Then, we need to start a single node Ray cluster locally first using the following two commands:

```
ray start --head
serve start
```

This will start a Ray cluster locally, and you can access its dashboard from your web browser at `http://127.0.0.1:8265/#/` as follows:

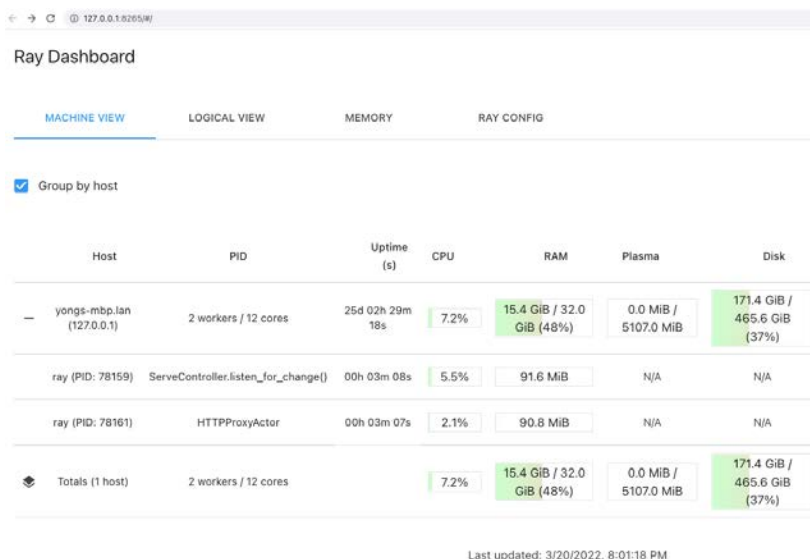


Figure 8.3 – A locally running Ray cluster

Figure 8.3 shows a locally running Ray cluster. You can then issue the following command to deploy `inference_pipeline_model` into Ray Serve as follows:

```
mlflow deployments create -t ray-serve -m
runs:/63f101fb3700472ca58975488636f4ae/inference_pipeline_model
--name dl-inference-model-on-ray -C num_replicas=1
```

This will show the following screen output:

```
2022-03-20 20:16:46,564      INFO worker.py:842 -- Connecting to
existing Ray cluster at address: 127.0.0.1:6379
2022-03-20 20:16:46,717      INFO api.py:242 -- Updating
deployment 'dl-inference-model-on-ray'. component=serve
deployment=dl-inference-model-on-ray
(ServeController pid=78159) 2022-03-20 20:16:46,784      INFO
deployment_state.py:912 -- Adding 1 replicas to deployment
'dl-inference-model-on-ray'. component=serve deployment=dl-
inference-model-on-ray
2022-03-20 20:17:10,309      INFO api.py:249 -- Deployment
'dl-inference-model-on-ray' is ready at `http://127.0.0.1:8000/
dl-inference-model-on-ray`. component=serve deployment=dl-
inference-model-on-ray
python_function deployment dl-inference-model-on-ray is created
```

This means that an endpoint at `http://127.0.0.1:8000/dl-inference-model-on-ray` is ready to serve an online inference request! You can test this deployment using the Python code provided at `chapter08/ray_serve/query_ray_serve_endpoint.py` as follows:

```
python ray_serve/query_ray_serve_endpoint.py
```

This will show results on the screen as follows:

```
2022-03-20 21:16:45,125      INFO worker.py:842 -- Connecting to
existing Ray cluster at address: 127.0.0.1:6379
[{'name': 'dl-inference-model-on-ray', 'info':
Deployment(name=dl-inference-model-on-ray, version=None, route_
prefix=/dl-inference-model-on-ray)}]
{
  "columns": [
    "text"
  ],
}
```

```
"index": [
  0,
  1
],
"data": [
  [
    "{\"response\": {\"prediction_label\": [\"negative\"]}, \"metadata\": {\"language_detected\": \"en\"}, \"model_metadata\": {\"finetuned_model_uri\": \"runs:/be2fb13fe647481eafa071b79dde81de/model\", \"inference_pipeline_model_uri\": \"runs:/63f101fb3700472ca58975488636f4ae/inference_pipeline_model\"}}"
  ],
  [
    "{\"response\": {\"prediction_label\": [\"positive\"]}, \"metadata\": {\"language_detected\": \"en\"}, \"model_metadata\": {\"finetuned_model_uri\": \"runs:/be2fb13fe647481eafa071b79dde81de/model\", \"inference_pipeline_model_uri\": \"runs:/63f101fb3700472ca58975488636f4ae/inference_pipeline_model\"}}"
  ]
]
}
```

You should see the inference model response as expected. If you followed through up to this point, congratulations on your successful deployment using the `mlflow-ray-serve` MLflow deployment plugin! If you no longer need this Ray Serve instance, you can stop it by executing the following command line:

```
ray stop
```

This will stop all running Ray instances on your local machine.

### Deployment Using MLflow Deployment Plugins

There are several MLflow deployment plugins. We just showed how to use `mlflow-ray-serve` to deploy a generic MLflow Python model, `inference_pipeline_model`. This opens doors to deploying to many target destinations where you can launch a Ray cluster in any cloud provider. We will not cover more details in this chapter as it's beyond the scope of this book. If you are interested, refer to the Ray documentation on how to launch cloud clusters (AWS, Azure, and **Google Cloud Platform (GCP)**): <https://docs.ray.io/en/latest/cluster/cloud.html#:~:text=The%20Ray%20Cluster%20Launcher%20can,ready%20to%20launch%20your%20cluster>. Once there is a Ray cluster, you can follow the same procedure to deploy an MLflow model.

Now that we know several ways to deploy locally and could further deploy to the cloud using Ray Serve if desirable, let's see how we can deploy to a cloud-managed inference service, AWS SageMaker, in the next section, since it is widely used and can provide a good lesson on how to deploy in a realistic scenario.

## Deploying to AWS SageMaker – a complete end-to-end guide

AWS SageMaker has a cloud-hosted model service managed by AWS. We will use AWS SageMaker as an example to show you how to deploy to a remote cloud provider for hosted web services that can serve real production traffic. AWS SageMaker has a suite of ML/DL-related services including supporting annotation and model training and many more. Here, we show how to **bring your own model (BYOM)** for deployment. This means that you have a model inference pipeline trained outside of AWS SageMaker, and now just need to deploy to SageMaker for hosting. Follow the next steps to prepare and deploy a DL sentiment model. A few prerequisites are required:

- You must have Docker Desktop running in your local environment.
- You must have an AWS account. You can create a free AWS account easily through the free signup website at <https://aws.amazon.com/free/>.

Once you have these requirements, activate the `dl-model-chapter08` conda virtual environment to follow through a few steps for deploying to SageMaker. We breakdown these steps into six subsections as follows:

1. Build a local SageMaker Docker image
2. Add additional model artifacts layers onto the SageMaker Docker image

3. Test local deployment with the newly built SageMaker Docker image
4. Push the SageMaker Docker image to AWS Elastic Container Registry
5. Deploy the inference pipeline model to create a SageMaker endpoint
6. Query the SageMaker endpoint for online inference

Let's start with the first step to build a local SageMaker Docker image.

## Step 1: Build a local SageMaker Docker image

We intentionally start with a local build without pushing to the AWS so that we can learn how to add additional layers on top of this basic image and verify everything locally before incurring any cloud cost:

```
mlflow sagemaker build-and-push-container --build --no-push -c
mlflow-dl-inference
```

You will see a lot of screen outputs and at the end, it will show something like the following:

```
#15 exporting to image
#15 sha256:e8c613e07b0b7ff33893b694f7759a10
d42e180f2b4dc349fb57dc6b71dcab00
#15 exporting layers
#15 exporting layers 8.7s done
#15 writing image sha256:95bc539b021179e5e87087
012353ebb43c71410be535ef368d1121b550c57bd4 done
#15 naming to docker.io/library/mlflow-dl-inference done
#15 DONE 8.7s
```

If you see the image name `mlflow-dl-inference`, that means you have successfully created a SageMaker-compatible MLflow-model-serving Docker image. You can verify this by running the following command:

```
docker images | grep mlflow-dl-inference
```

You should see output like the following:

```
mlflow-dl-inference          latest
95bc539b0211    6 minutes ago    2GB
```

## Step 2: Add additional model artifacts layers onto the SageMaker Docker image

Recall that our inference pipeline model builds on top of a fine-tuned DL model and we load the model through the MLflow PythonModel API's `load_context` function ([https://www.mlflow.org/docs/latest/python\\_api/mlflow.pyfunc.html#mlflow.pyfunc.PythonModel](https://www.mlflow.org/docs/latest/python_api/mlflow.pyfunc.html#mlflow.pyfunc.PythonModel)) without serializing the fine-tuned model itself. This is partly because MLflow cannot serialize the PyTorch DataLoader (<https://pytorch.org/docs/stable/data.html#single-and-multi-process-data-loading>) properly using pickle since the DataLoader does not implement pickle serialization as of this writing. This does give us an opportunity to learn how we can deploy when some of the dependencies cannot be serialized properly, especially when dealing with a real-world DL model.

### Two Ways to Allow a Docker Container to Access an MLflow Tracking Server

There are two ways to allow a Docker container such as `mlflow-dl-inference` to access and load a fine-tuned model at runtime. The first method is to allow the container to include the MLflow tracking server URL and access token. This may cause some security concerns in an enterprise environment since the Docker image now contains some security credentials. The second method is to directly copy all the referenced artifacts to create a new Docker image that's self-sufficient. At runtime, it does not have to know where the original MLflow tracking server is located since it has all model artifacts locally. This self-contained approach eliminates any concerns of security leaking. We use this second approach in this chapter for deployment.

In this chapter, we will copy the referenced fine-tuned model into a new Docker image that's built on top of the basic `mlflow-dl-inference` Docker image. This will make a new self-contained Docker image without relying on any external MLflow tracking server. To do this, you need to either download the fine-tuned DL model from a model tracking server to your current local folder, or you can just run our MLproject's pipeline locally using the local filesystem as the MLflow tracking server backend. Follow the *Deploy to AWS SageMaker* section in the `README.md` file to reproduce the local MLflow runs to prepare a fine-tuned model and `inference-pipeline-model` in the local folder. For learning purposes, we have provided two example `mlruns` artifacts and the `huggingface` cache folder in the GitHub repository in the `chapter08` folder (<https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/tree/main/chapter08>), so that we can start building a new Docker image right away by using these existing artifacts.



To build a new Docker image, we need to create a Dockerfile as follows:

```
FROM mlflow-dl-inference
ADD mlruns/1/meta.yaml /opt/mlflow/mlruns/1/meta.yaml
ADD mlruns/1/d01fc81e11e842f5b9556ae04136c0d3/ /opt/mlflow/
mlruns/1/d01fc81e11e842f5b9556ae04136c0d3/
ADD tmp/opt/mlflow/hf/cache/dl_model_chapter08/csv/ /opt/
mlflow/tmp/opt/mlflow/hf/cache/dl_model_chapter08/csv/
```

The first line means that it starts with the existing `mlflow-dl-inference` Docker image, and the following three lines of `ADD` will copy one `meta.yaml` file and two folders to the corresponding locations in the Docker image. Note that if you already have produced your own runs by following the `README` file, then you do not need to add the third line. Note that, by default, when the Docker container starts, it automatically goes to this `/opt/mlflow/` working directory so everything needs to be copied to this folder for easy access. Also, note that the `/opt/mlflow` directory requires superuser permission, so you need to be prepared to enter your local machine's admin password (usually, on your own laptop, that's your own password).

#### Copy Privately Built Python Packages into Docker Images

It is also possible to copy privately built Python packages into Docker images so that we can directly reference them in the `conda.yaml` file without going outside of the container itself. For example, we can copy a private Python wheel package, `cool-dl-package-1.0.py3-none-any.whl`, to the `/usr/private-wheels/cool-dl-package/cool-dl-package-1.0-py3-none-any.whl` Docker folder, and then we can point to this path in the `conda.yaml` file. This allows MLflow model artifacts to load these locally accessible Python packages successfully. In our current example, we don't use this approach since we haven't used any privately built Python packages. This is useful for future reference if you are interested in exploring this.

Now, you can run the following command to build a new Docker image in the `chapter08` folder as follows:

```
docker build . -t mlflow-dl-inference-w-finetuned-model
```

This will build a new Docker image, `mlflow-dl-inference-w-finetuned-model`, on top of `mlflow-dl-inference`. You should see the following output (only the first and last couple of lines are presented for brevity):

```
[+] Building 0.2s (9/9) FINISHED
=> [internal] load build definition from Dockerfile
                                                                    0.0s
.....
=> => naming to docker.io/library/mlflow-dl-inference-w-
finetuned-model
```

Now, you have a new Docker image named `mlflow-dl-inference-w-finetuned-model`, which contains the fine-tuned model. Now, we are ready to deploy our inference pipeline model using this new Docker image, which is SageMaker compatible.

## Step 3: Test local deployment with the newly built SageMaker Docker image

Before we deploy to the cloud, let's test the deployment locally with this new SageMaker Docker image. MLflow provides a convenient way to test this locally using the following command:

```
mlflow sagemaker run-local -m runs:/
dc5f670efa1a4eac95683633ffcfdd79/inference_pipeline_model -p
5555 -i mlflow-dl-inference-w-finetuned-model
```

This command will start running the `mlflow-dl-inference-w-finetuned-model` Docker container locally and deploy the inference pipeline model with a `dc5f670efa1a4eac95683633ffcfdd79` run ID into this container.

### Fix a Potential Docker Error

Note that you may encounter a Docker error saying **The path `/opt/mlflow/mlruns/1/dc5f670efa1a4eac95683633ffcfdd79/artifacts/inference_pipeline_model` is not shared from the host and is not known to Docker**. You can configure shared paths from [Docker | Preferences... | Resources | File Sharing](#) to fix this Docker error.

We already provided this inference pipeline model in the GitHub repository, so this should work out-of-the-box when you check out the repository in your local environment. The port for web service is 5555. Once the command is running, you will see a lot of outputs on the screen, and finally, you should see the following:

```
[2022-03-18 01:47:20 +0000] [552] [INFO] Starting gunicorn
20.1.0
[2022-03-18 01:47:20 +0000] [552] [INFO] Listening at:
http://127.0.0.1:8000 (552)
[2022-03-18 01:47:20 +0000] [552] [INFO] Using worker: gevent
[2022-03-18 01:47:20 +0000] [565] [INFO] Booting worker with
pid: 565
[2022-03-18 01:47:20 +0000] [566] [INFO] Booting worker with
pid: 566
[2022-03-18 01:47:20 +0000] [567] [INFO] Booting worker with
pid: 567
[2022-03-18 01:47:20 +0000] [568] [INFO] Booting worker with
pid: 568
[2022-03-18 01:47:20 +0000] [569] [INFO] Booting worker with
pid: 569
[2022-03-18 01:47:20 +0000] [570] [INFO] Booting worker with
pid: 570
```

This means that the service is up and running. You might see a few warnings about the PyTorch version not being compatible, but they can be safely ignored. Once this service is up and running, you can then test against it in a different Terminal window by issuing a `curl` web request as follows, like we have tried before:

```
curl http://127.0.0.1:5555/invocations -H 'Content-Type:
application/json' -d '{
  "columns": ["text"],
  "data": [{"This is the best movie we saw."}, {"What a
movie!"}]
}'
```

Note that the port number is 5555 for the localhost. You should then see the response as follows:

```
{
  "text": "{\\\"response\\\": {\\\"prediction_label\\\":
    [\\\"positive\\\"]}, \\\"metadata\\\": {\\\"language_detected\\\":
    \\\"en\\\"}, \\\"model_metadata\\\": {\\\"finetuned_model_
    uri\\\": \\\"runs:/d01fc81e11e842f5b9556ae04136c0d3/
    model\\\", \\\"inference_pipeline_model_uri\\\": \\\"runs:/
    dc5f670efala4eac95683633ffcfdd79/inference_pipeline_
    model\\\"}}\", {
  \"text\": \"{\\\"response\\\": {\\\"prediction_label\\\":
    [\\\"negative\\\"]}, \\\"metadata\\\": {\\\"language_detected\\\":
    \\\"en\\\"}, \\\"model_metadata\\\": {\\\"finetuned_model_uri\\\": \\\"runs:/
    d01fc81e11e842f5b9556ae04136c0d3/model\\\", \\\"inference_pipeline_
    model_uri\\\": \\\"runs:/dc5f670efala4eac95683633ffcfdd79/
    inference_pipeline_model\\\"}}\"}]
```

You may wonder how this is different from the previous section's local web service for the inference model. The difference is that this time, we are using a SageMaker container locally, while previously, it was just a local web service without a Docker container. Having the SageMaker container tested locally is very important so that you don't waste time and money deploying a failed model service to the cloud.

Next, we are ready to deploy this container to AWS SageMaker.

## Step 4: Push the SageMaker Docker image to AWS Elastic Container Registry

Now, you can push your newly built `mlflow-dl-inference-w-finetuned-model` Docker image to AWS **Elastic Container Registry (ECR)** with the following command. Make sure you have your AWS access token and access ID set up correctly (the real one, not the local development one). Once you have your access key ID and token, run the following command to set up your access to the real AWS:

```
aws configure
```

Answer all the questions after executing the command and you will be ready to go. Now, you can run the following command to push the `mlflow-dl-inference-w-finetuned-model` Docker image to the AWS ECR:

```
mlflow sagemaker build-and-push-container --no-build --push -c
mlflow-dl-inference-w-finetuned-model
```

Make sure you don't build a new image with the `--no-build` option included in the command since we just want to push the image, not build a new one. You will see the following output, which shows the image is being pushed to the ECR. Note that in the following output, the AWS account is masked with `xxxxxx`. You will see your account number showing in the output. Make sure you have the permission to write to the AWS ECR store:

```
2022/03/18 17:36:05 INFO mlflow.sagemaker: Pushing image to ECR
2022/03/18 17:36:06 INFO mlflow.sagemaker: Pushing docker
image mlflow-dl-inference-w-finetuned-model to xxxxxx.dkr.
ecr.us-west-2.amazonaws.com/mlflow-dl-inference-w-finetuned-
model:1.23.1
Created new ECR repository: mlflow-dl-inference-w-finetuned-
model
2022/03/18 17:36:06 INFO mlflow.sagemaker: Executing: aws ecr
get-login-password | docker login --username AWS --password-
stdin xxxxxx.dkr.ecr.us-west-2.amazonaws.com;
docker tag mlflow-dl-inference-w-finetuned-model xxxxxx.dkr.
ecr.us-west-2.amazonaws.com/mlflow-dl-inference-w-finetuned-
model:1.23.1;
docker push xxxxxx.dkr.ecr.us-west-2.amazonaws.com/mlflow-dl-
inference-w-finetuned-model:1.23.1
Login Succeeded
The push refers to repository [xxxxxx.dkr.ecr.us-west-2.
amazonaws.com/mlflow-dl-inference-w-finetuned-model]
447db5970ca5: Pushed
9d6787a516e7: Pushed
1.23.1: digest: sha256:f49f85741bc2b82388e85c79f6621f4
d7834e19bdf178b70c1a6c78c572b4d10 size: 3271
```

Once this is done, if you go to the AWS website (for example, if you use the `us-west-2` region data center, the URL is `https://us-west-2.console.aws.amazon.com/ecr/repositories?region=us-west-2`), you should find your newly pushed image in the ECR with a folder named `mlflow-dl-inference-w-finetuned-model`. You will then find the image in this folder as follows (*Figure 8.4*):

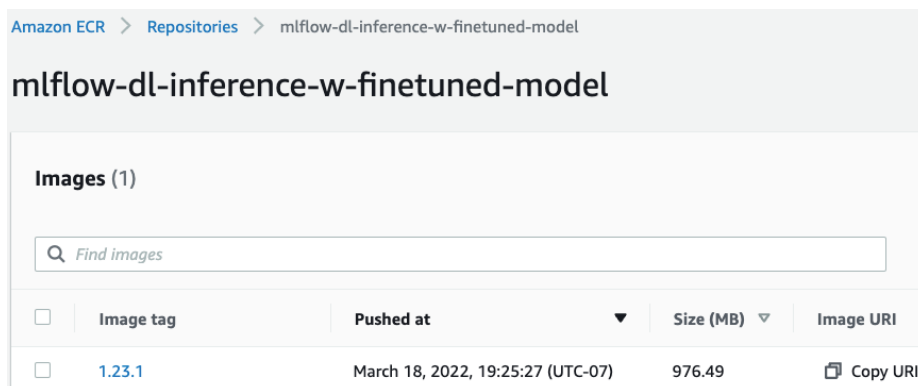


Figure 8.4 – AWS ECR repositories with `mlflow-dl-inference-w-finetuned-model` image tag `1.23.1`

Note that the image tag number **1.23.1** in *Figure 8.4* is the MLflow version we used. This image has a full URI, which you can get using the `Copy URI` option. It will look as follows (with the AWS account masked with `xxxxxx`):

```
xxxxxx.dkr.ecr.us-west-2.amazonaws.com/mlflow-dl-inference-w-  
finetuned-model:1.23.1
```

You will need this image URI to deploy to SageMaker in the next step. Let's now deploy to SageMaker to create an inference endpoint.

## Step 5: Deploy the inference pipeline model to create a SageMaker endpoint

Now, it is time to deploy the inference pipeline model to SageMaker using this image URI we just pushed to the AWS ECR registry. We have included the `sagemaker/deploy_to_sagemaker.py` code in the `chapter08` folder in the GitHub repository. You will need to use the correct AWS role for the deployment. You can create a new `AWSSageMakerExecutionRole` role in your account and assign two permissions policies to this role, `AmazonS3FullAccess` and `AmazonSageMakerFullAccess`. In a real-world scenario, you might want to tighten the permission to a more restricted policy, but for learning purposes, this will work fine. The following figure shows the screen after the role is created:

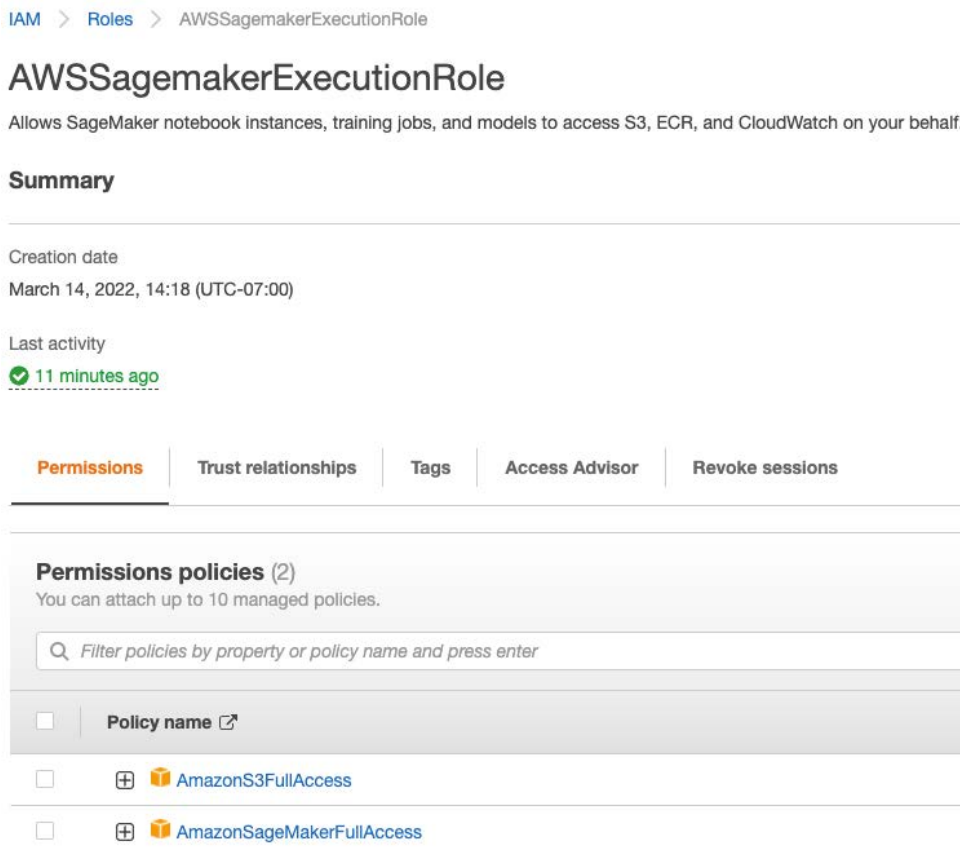


Figure 8.5 – Create a role that can be used for deployment in SageMaker

You also need to create an S3 bucket for SageMaker to upload the model artifacts and deploy them to SageMaker. In our example, we created a bucket called `dl-inference-deployment`. When we execute the deployment script, as shown here, the model to be deployed will be first uploaded to the `dl-inference-deployment` bucket and then deployed to SageMaker. We have provided the complete deployment script in the `chapter08/sagemaker/deploy_to_sagemaker.py` GitHub repository so you can download and execute it as follows (as a reminder, before you run this script, make sure you reset the environment variable of `MLFLOW_TRACKING_URI` to empty, as in `export MLFLOW_TRACKING_URI=`):

```
sudo python sagemaker/deploy_to_sagemaker.py
```

This script executes the following two tasks:

1. Makes a copy of the local `mlruns` under the `chapter08` folder to a local `/opt/mlflow` folder so that SageMaker deployment code can pick up the `inference-pipeline-model` to upload. Because the `/opt` path is usually restricted, here we use `sudo` (superuser) to do this copy. This will prompt you to type in your user password on your laptop.
2. Uses the `mlflow.sagemaker.deploy` API to create a new SageMaker endpoint, `dl-sentiment-model`.

The code snippet is as follows:

```
mlflow.sagemaker.deploy(  
    mode='create',  
    app_name=endpoint_name,  
    model_uri=model_uri,  
    image_url=image_uri,  
    execution_role_arn=role,  
    instance_type='ml.m5.xlarge',  
    bucket = bucket_for_sagemaker_deployment,  
    instance_count=1,  
    region_name=region  
)
```

The parameters need some explanations so that we fully understand all the preparation work that is needed:

- `model_uri`: This is the inference pipeline model's URI. In our example, it is `runs:/dc5f670efa1a4eac95683633ffcfdd79/inference_pipeline_model`.



- `image_url`: This is the Docker image we uploaded to the AWS ECR. In our example, it is `xxxxxx.dkr.ecr.us-west-2.amazonaws.com/mlflow-dl-inference-w-finetuned-model:1.23.1`. Note that you need to replace the masked AWS account number, `xxxxxx`, with your actual account number.
- `execution_role_arn`: This is the role we created to allow SageMaker to do the deployment. In our example, it is `arn:aws:iam::565251169546:role/AWSSageMakerExecutionRole`. Again, you need to replace `xxxxxx` with your actual AWS account number.
- `bucket`: This is the S3 bucket we created to allow SageMaker to upload the model and then do the actual deployment. In our example, it is `dl-inference-deployment`.

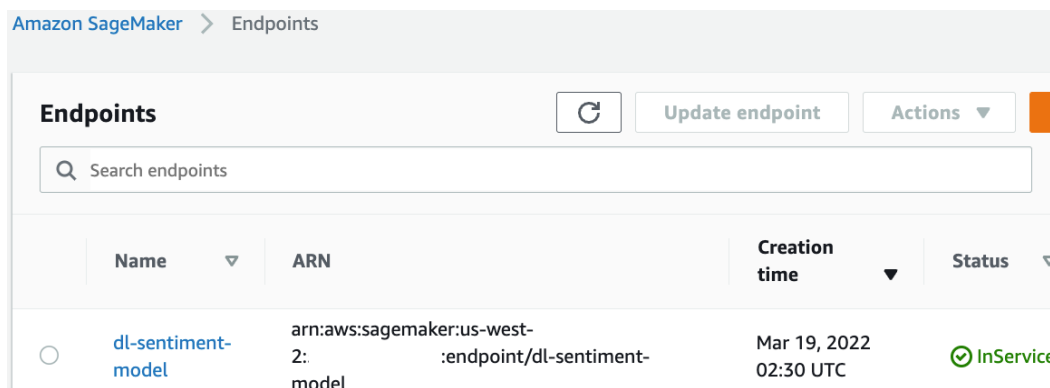
The rest of the parameters are self-explanatory.

After you execute the deployment script, you will see the following output (where `xxxxxx` is the masked AWS account number):

```
2022/03/18 19:30:47 INFO mlflow.sagemaker: Using the python_
function flavor for deployment!
2022/03/18 19:30:47 INFO mlflow.sagemaker: tag response:
{'ResponseMetadata': {'RequestId': 'QMAQRCTJT36TXD2H',
'HostId': 'DNG57U3DJrhLcsBxa39zsjulUH9VB56FmGkxAiMYN+2fhc/
rRukWe8P3qmBmvRYbMj0sW3B2iGg=', 'HTTPStatusCode':
200, 'HTTPHeaders': {'x-amz-id-2':
'DNG57U3DJrhLcsBxa39zsjulUH9VB56FmGkxAiMYN+2fhc/
rRukWe8P3qmBmvRYbMj0sW3B2iGg=', 'x-amz-request-id':
'QMAQRCTJT36TXD2H', 'date': 'Sat, 19 Mar 2022 02:30:48 GMT',
'server': 'AmazonS3', 'content-length': '0'}, 'RetryAttempts':
0}}
2022/03/18 19:30:47 INFO mlflow.sagemaker: Creating new
endpoint with name: dl-sentiment-model ...
2022/03/18 19:30:47 INFO mlflow.sagemaker: Created model with
arn: arn:aws:sagemaker:us-west-2:xxxxxx:model/dl-sentiment-
model-model-qbca2radrxitkujn3ezubq
2022/03/18 19:30:47 INFO mlflow.sagemaker: Created
endpoint configuration with arn: arn:aws:sagemaker:us-
west-2:xxxxxx:endpoint-config/dl-sentiment-model-config-
r9ax3wlhrfisxkacyycj8a
2022/03/18 19:30:48 INFO mlflow.sagemaker: Created endpoint
with arn: arn:aws:sagemaker:us-west-2:xxxxxx:endpoint/
dl-sentiment-model
2022/03/18 19:30:48 INFO mlflow.sagemaker: Waiting for the
```

```
deployment operation to complete...
2022/03/18 19:30:48 INFO mlflow.sagemaker: Waiting for endpoint
to reach the "InService" state. Current endpoint status:
"Creating"
```

This may take several minutes (sometimes more than 10 minutes). You may see some warning messages regarding PyTorch version compatibility as you saw when doing local SageMaker deployment testing. You can also go directly to the SageMaker website and you will see the status of the endpoints starting with **Creating**, and then eventually turning to a green-colored **InService** status as follows:



Name	ARN	Creation time	Status
dl-sentiment-model	arn:aws:sagemaker:us-west-2: endpoint/dl-sentiment-model	Mar 19, 2022 02:30 UTC	InService

Figure 8.6 – AWS SageMaker dl-sentiment-model endpoint InService

If you see the **InService** status, then congratulations! You have successfully deployed a DL inference pipeline model into SageMaker and you can now use it for production traffic!

Now that the status of the service is **InService**, you can query it using the command line in the next step.

## Step 6: Query the SageMaker endpoint for online inference

To query the SageMaker endpoint, you can use the following command line:

```
aws sagemaker-runtime invoke-endpoint --endpoint-name
'dl-sentiment-model' --content-type 'application/json;
format=pandas-split' --body '{"columns":["text"], "data":
[["This is the best movie we saw."], ["What a movie!"]]'
response.json
```

You will then see the output as follows:

```
{
  "ContentType": "application/json",
  "InvokedProductionVariant": "dl-sentiment-model-model-qbca2radrxitkujn3ezubq"
}
```

The actual prediction results are stored in a local `response.json` file, which can be viewed by running the following command to show the content of the response:

```
cat response.json
```

This will display the content as follows:

```
[{"text": "{\"response\": {\"prediction_label\": [\"positive\"]}, \"metadata\": {\"language_detected\": \"en\"}, \"model_metadata\": {\"finetuned_model_uri\": \"runs:/d01fc81e11e842f5b9556ae04136c0d3/model\", \"inference_pipeline_model_uri\": \"runs:/dc5f670efala4eac95683633ffcfdd79/inference_pipeline_model\"}}"}, {"text": "{\"response\": {\"prediction_label\": [\"negative\"]}, \"metadata\": {\"language_detected\": \"en\"}, \"model_metadata\": {\"finetuned_model_uri\": \"runs:/d01fc81e11e842f5b9556ae04136c0d3/model\", \"inference_pipeline_model_uri\": \"runs:/dc5f670efala4eac95683633ffcfdd79/inference_pipeline_model\"}}"}]
```

This is the expected response pattern from our inference pipeline model! It is also possible to run the query against the SageMaker inference endpoint using Python code, which we have provided in the `chapter08/sagemaker/query_sagemaker_endpoint.py` file in the GitHub repository. The core code snippet uses **Boto3** and the SageMakerRuntime client's `invoke_endpoint` to query, as follows:

```
client = boto3.client('sagemaker-runtime')
response = client.invoke_endpoint(
    EndpointName=app_name,
    ContentType=content_type,
    Accept=accept,
    Body=payload
)
```

The parameters for `invoke_endpoint` need some explanation:

- `EndpointName`: This is the inference endpoint name. In our example, it is `dl-inference-model`.
- `ContentType`: This is the MIME type of the input data in the request body. In our example, we use `application/json; format=pandas-split`.
- `Accept`: This is the desired MIME type of the inference in the response body. In our example, we expect the `text/plain` string type.
- `Body`: This is the actual text that we want to predict the sentiment using the DL model inference service. In our example, it is `{"columns": ["text"], "data": [{"text": "This is the best movie we saw."}, {"text": "What a movie!"}]}`.

The full code is provided in the GitHub repository, and you can run it in the command line as follows:

```
python sagemaker/query_sagemaker_endpoint.py
```

You will see the following output on your Terminal screen:

```
Application status is: InService
[{"text": "{\"response\": {\"prediction_label\": [\"positive\"]}, \"metadata\": {\"language_detected\": \"en\"}, \"model_metadata\": {\"finetuned_model_uri\": \"runs:/d01fc81e11e842f5b9556ae04136c0d3/model\", \"inference_pipeline_model_uri\": \"runs:/dc5f670efala4eac95683633ffcfdd79/inference_pipeline_model\"}}"}, {"text": "{\"response\": {\"prediction_label\": [\"negative\"]}, \"metadata\": {\"language_detected\": \"en\"}, \"model_metadata\": {\"finetuned_model_uri\": \"runs:/d01fc81e11e842f5b9556ae04136c0d3/model\", \"inference_pipeline_model_uri\": \"runs:/dc5f670efala4eac95683633ffcfdd79/inference_pipeline_model\"}}"}]
```

This is what we expect from our inference pipeline model's response! If you have followed this chapter up to here, congratulate yourself on successfully deploying our inference pipeline model into production in a remote cloud host, AWS SageMaker! When you are done following the lessons in this chapter, make sure to delete the endpoint so that it doesn't incur unnecessary costs.

Let's summarize what we've learned in this chapter.

## Summary

In this chapter, we have learned different ways to deploy an MLflow inference pipeline model for both batch inference and online real-time inference. We started with a brief survey on different model serving scenarios (batch, streaming, and on-device) and looked at three different categories of tools for MLflow model deployment (the MLflow built-in deployment tool, MLflow deployment plugins, and generic model inference serving frameworks that could work with the MLflow inference model). Then, we covered several local deployment scenarios, using the PySpark UDF function to do batch inference and MLflow local deployment for web service. Afterward, we learned how to use Ray Serve in conjunction with the `mlflow-ray-serve` plugin to deploy an MLflow Python inference pipeline model into a local Ray cluster. This opens doors to deploy to any cloud platform such as AWS, Azure ML, or GCP, as long as we can set up a Ray cluster in the cloud. Finally, we provided a complete end-to-end guide on how to deploy to AWS SageMaker, focusing on a common scenario of BYOM, where we have a trained inference pipeline model that's built outside of AWS SageMaker and now needs to be deployed to AWS SageMaker for a hosting model service. Our step-by-step guide should provide you with the confidence to deploy an MLflow inference pipeline model for real production usage.

Note that the landscape of deploying DL inference pipeline models is still evolving, and we just learned some foundational skills. You are encouraged to explore more from the *Further reading* section for more advanced topics.

Now that we know how to deploy and host a DL inference pipeline, we will learn how to do model explainability in the next chapter, which is of great importance for trustworthy and interpretable model prediction results in many real-world scenarios.

## Further reading

- *An Introduction to TinyML*: <https://towardsdatascience.com/an-introduction-to-tinyml-4617f314aa79>
- *Performance Optimizations and MLFlow Integrations – Seldon Core 1.10.0 Released*: <https://www.seldon.io/performance-optimizations-and-mlflow-integrations-seldon-core-1-10-0-released/>
- *Ray & MLflow: Taking Distributed Machine Learning Applications to Production*: <https://medium.com/distributed-computing-with-ray/ray-mlflow-taking-distributed-machine-learning-applications-to-production-103f5505cb88>

- 
- *Managing your machine learning lifecycle with MLflow and Amazon SageMaker:* <https://aws.amazon.com/blogs/machine-learning/managing-your-machine-learning-lifecycle-with-mlflow-and-amazon-sagemaker/>
  - *Deploy A Locally Trained ML Model In Cloud Using AWS SageMaker:* <https://medium.com/geekculture/84af8989d065>
  - *PyTorch vs TensorFlow in 2022:* <https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2022/>
  - *Try Databricks: Free Trial or Community Edition:* <https://docs.databricks.com/getting-started/try-databricks.html#free-trial-or-community-edition>
  - *MLOps with MLflow and Amazon SageMaker Pipelines:* <https://towardsdatascience.com/mlops-with-mlflow-and-amazon-sagemaker-pipelines-33e13d43f238>
  - *PyTorch JIT and TorchScript:* <https://towardsdatascience.com/pytorch-jit-and-torchscript-c2a77bac0fff>
  - *ML Model Serving Best Tools:* <https://neptune.ai/blog/ml-model-serving-best-tools>
  - *Deploying Machine Learning models to production — Inference service architecture patterns:* <https://medium.com/data-for-ai/deploying-machine-learning-models-to-production-inference-service-architecture-patterns-bc8051f70080>
  - *How to Deploy Large-Size Deep Learning Models into Production:* <https://towardsdatascience.com/how-to-deploy-large-size-deep-learning-models-into-production-66b851d17f33>
  - *Serving ML models at scale using Mlflow on Kubernetes:* <https://medium.com/artefact-engineering-and-data-science/serving-ml-models-at-scale-using-mlflow-on-kubernetes-7a85c28d38e>
  - *When PyTorch meets MLflow:* <https://mlops.community/when-pytorch-meets-mlflow/>
  - *Deploy a model to an Azure Kubernetes Service Cluster:* <https://docs.microsoft.com/en-us/azure/machine-learning/how-to-deploy-azure-kubernetes-service?tabs=python>
  - *ONNX and Azure Machine Learning: Create and accelerate ML models:* <https://docs.microsoft.com/en-us/azure/machine-learning/concept-onnx>



# Section 5 – Deep Learning Model Explainability at Scale

In this section, we will learn about the foundational concepts of explainability and **explainable artificial intelligence (XAI)** and how to implement **deep learning (DL)** explainability with MLflow. We will start with an overview of the eight dimensions of explainability and then learn how to use **SHapley Additive exPlanations (SHAP)** and **Transformers Interpret** to perform explainability for a **natural language processing (NLP)** pipeline. Furthermore, we will learn and analyze the current MLflow integration with SHAP to understand the trade-offs and avoid potential implementation problems. Then, we will show how to implement SHAP using MLflow's logging APIs. Finally, we will learn how to implement a SHAP explainer as an MLflow Python model and then load it as either a Spark UDF for batch explanation or as a web service for online **Explanation-as-a-Service (EaaS)**.

This section comprises the following chapters:

- *Chapter 9, Fundamentals of Deep Learning Explainability*
- *Chapter 10, Implementing DL Explainability with MLflow*





# 9

# Fundamentals of Deep Learning Explainability

Explainability is providing selective human-understandable explanations for a decision provided by an automated system. In the context of this book, during the full life cycle of **deep learning (DL)** development, explainability should be emphasized as a first-class artifact, along with the other three pillars: data, code, and model. This is because different stakeholders and regulators, model developers, and final consumers of the model output may have different needs to understand how the data is used and why the model produces certain predictions or classifications. Without such understanding, it will be difficult to gain the trust of the consumers of the model output or to diagnose what could have gone wrong when model output results drift. This also means that explainability tools should be employed not only for explaining prediction results from a deployed model in production or during offline experimentation, but also for understanding the data characteristics and differences between the datasets used in offline model training and the ones encountered in online model operation.

In addition, in many highly regulated industries, such as autonomous driving, medical diagnosis, banking, and finance, there is also a legal mandate that demands the **right to explanation** (<https://academic.oup.com/idpl/article/7/4/233/4762325>) for any individual to get an explanation for an output of the algorithm. Finally, a recent survey showed that over 82% of CEOs believe that AI-based decisions must be explainable to be trusted as enterprises accelerate their investment in developing and deploying AI-based initiatives (<https://cloud.google.com/blog/topics/developers-practitioners/bigquery-explainable-ai-now-ga-help-you-interpret-your-machine-learning-models>). Therefore, it is important to learn the fundamentals of explainability and the related tools so that we know when to use what tools for what audience to provide a relevant, accurate, and consistent explanation.

By the end of this chapter, you will be confident to know what a good explanation is and what tools exist for different explainability purposes and will gain hands-on experience in using two explainability toolboxes for explaining DL sentiment classification models.

In this chapter, we're going to cover the following main topics:

- Understanding the categories and audience of explainability
- Exploring the SHAP Explainability toolbox
- Exploring the Transformers Interpret toolbox

## Technical requirements

The following requirements are necessary to complete the learning in this chapter:

- SHAP Python library: <https://github.com/slundberg/shap>
- Transformers Interpret Python library: <https://github.com/cdpierser/transformers-interpret>
- Captum Python library: <https://github.com/pytorch/captum>
- Code from the GitHub repository for this chapter: <https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/tree/main/chapter09>

## Understanding the categories and audience of explainability

As this chapter's opening texts imply, explainability for a DL system becomes increasingly critical, sometimes even mandatory, in highly regulated industries such as financial, legal, governmental, and medical application domains. An example lawsuit partially due to the lack of ML explainability is the case of **B2C2 v Quoine** (<https://www.scl.org/articles/12130-explainable-machine-learning-how-can-you-determine-what-a-party-knew-or-intended-when-a-decision-was-made-by-machine-learning>), where automated AI trading algorithms mistakenly placed an order with 250 times the market price for bitcoin trading. The recent successful applications of DL models in production stimulate active and abundant research and development in the explainability area due to the need to understand why and how a DL model works. You may have heard of the term **explainable artificial intelligence (XAI)**, which was started by the **US Defense Advanced Research Projects Agency (DARPA)** in 2015 for its XAI program with the goal of enabling end users to better understand, trust, and effectively manage AI systems (<https://onlinelibrary.wiley.com/doi/epdf/10.1002/ai12.61>). However, the concept of explainability goes way back to the early days of expert systems in the 1980s or even earlier (<https://wires.onlinelibrary.wiley.com/doi/full/10.1002/widm.1391>), and the recent surge of attention on the topic of explainability just highlights how important it is.

So, what's an explanation? It turns out that this is still an active research topic in the ML/DL/AI community. From a practical purpose, a precise definition of explanation depends on who wants the explanations for what purpose at what time across the ML/DL/AI life cycle (<https://dl.acm.org/doi/abs/10.1145/3461778.3462131>). So, explainability can be defined as *the capability to provide an audience-appropriate, human-understandable interpretation of why and how a model provides certain predictions*. This may also include the data explainability aspect, where and how the data was used through provenance tracking, what the data characteristics are, or whether it has changed due to unexpected events. For example, sales and marketing emails changed due to an unexpected COVID outbreak (<https://www.validity.com/resource-center/disruption-in-email/>). Such data changes will unexpectedly change the distribution of model prediction results. We need to take into account such data changes when explaining the model drift. This means the complexity of the explanations needs to be tailored and selective to the receiving audience without overwhelming information. For example, a complex explanation with many technical jargons such as *activation* might not work as well as a simple text summary with business-friendly terms. This further shows that explainability is also a **Human-Computer Interface/Interaction (HCI)** topic.

To get the big picture of what the explainability categories and corresponding audiences look like, we consider the eight dimensions of explanations shown in *Figure 9.1*:

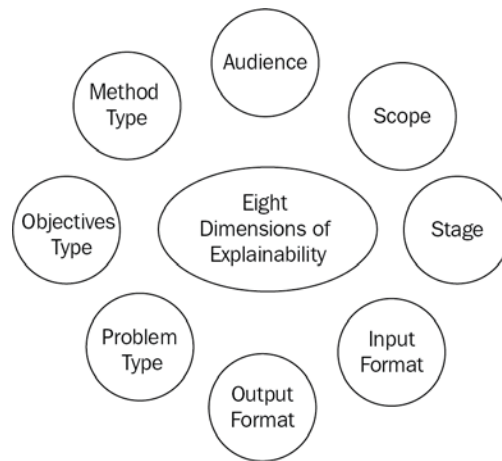


Figure 9.1 – Eight dimensions to understand explainability

As can be seen from *Figure 9.1*, the complexity of explainability can be understood from eight dimensions. This is not necessarily an exhaustive categorization, but rather a guide to understanding different perspectives from HCI, the full life cycle of AI/ML/DL, and different technical approaches. In the following discussion, we will highlight the dimensions and their inter-relationships that are most relevant to DL applications, since the focus of this chapter is on DL explainability.

## Audience: who needs to know

As pointed out recently by a study (<https://dl.acm.org/doi/abs/10.1145/3461778.3462131>), it is important to understand who needs to know what kind of explanations at what stage across an AI project life cycle. This will also affect the explanation output formats. An earlier study (<https://arxiv.org/pdf/1702.08608.pdf>) also points out that depending on whether a domain expert is involved in a real application task (for example, a medical doctor in a diagnosis of cancer), the cost of validating an explanation could also be high since it requires an actual human in a real work environment.

For current practical DL projects, we need to tailor our methods and presentations of explanations depending on the target audience, such as data scientists, ML engineers, business stakeholders, **User Experience (UX)** designers, or end users, as there is no one-size-fits-all approach.

## Stage: when to provide an explanation in the DL life cycle

A **stage** usually refers to when the explanations can be provided during the model development life cycle. For a model such as a decision tree, since it is a white-box model, we say we can provide **ante-hoc** explainability. However, currently, most DL models are mostly treated as black-box models even though self-explaining DL models are being gradually developed with ante-hoc explainability (<https://arxiv.org/abs/2108.11761>). Therefore, for current practical DL applications, **post-hoc** explainability is needed. In addition, when the model development stages are in training, validation, or production, the explainability scope can be global, cohort, or local, even using the same post-hoc explainability tools (<https://towardsdatascience.com/a-look-into-global-cohort-and-local-model-explainability-973bd449969f>).

## Scope: which prediction needs explanation

**Scope** refers to whether we can provide the explanation for all predictions, a subset of the predictions, or just one specific prediction, even if we use the same post-hoc tool for a black-box DL model. The most common global explainability is to describe **feature importance** and allow users to know which feature is the most impactful one for the overall model performance. Local explainability is about **feature attribution** for a specific prediction instance. The difference between feature attribution and feature importance is that feature attribution not only quantifies the ranking and magnitude of the feature impact, but also the direction of the impact (for example, whether a feature is positively or negatively affecting the prediction).

Many of the post-hoc tools for DL models are very good at local explainability. Cohort explainability is useful for identifying potential model bias for some specific groups such as age or race groups. For a DL model, if we want to have a global explanation, we often need to use a surrogate model such as a decision tree model to emulate the behavior of a DL model (<https://towardsdatascience.com/explainable-ai-xai-methods-part-5-global-surrogate-models-9c228d27e13a>). However, this approach does not always work well as it is very difficult to know whether the surrogate model is approximating the predictions of the original black-box model well enough. So, in practice, local explainability tools for DL models are often used, such as **SHapley Additive exPlanations (SHAP)**, which we will explain in the method dimension.

## Input data format: what is the format of the input data

**Input data format** refers to what kind of input data we are dealing with when developing and using the model. While a simple model might only focus on a single type of input data format such as text, many complex models might require using a mix of structured tabular data plus unstructured data such as images or texts. In addition, there is also a separate need to understand the input data hidden bias (during model training and validation) or drifting (during production). As such, this is quite a complex topic. The data explanation can also be used for monitoring data outliers and drifting during production. This is applicable to all types of ML/DL models.

## Output data format: what is the format of the output explanation

**Output explanation format** refers to how we present the explanations to our target audience. Often, an image explanation might be a bar chart showing the feature importance with the top few features and their scores, or a saliency map that highlights the spatial support of a particular class in each image for image-related ML problems. For a textual output, it could be an English sentence to say why a credit application is rejected because of a few factors that are understandable to the applicants. **Natural language processing (NLP)** model explainability could also be through interactive exploration that uses salience maps, attention, and other rich visualization (see examples in Google's **Language Interpretability Tool (LIT)**: <https://ai.googleblog.com/2020/11/the-language-interpretability-tool-lit.html>). As there is no silver bullet for explainability of these complex output formats, it is critical to meet the needs, experiences, and expectations of the audience that asks for the explanation.

## **Problem type: what is the machine learning problem type**

**Problem type** refers to all kinds of ML/AI problems broadly, but for practical purposes, current commercially successful problems are mostly around classification, regression, and clustering. Reinforcement learning and recommendation systems also see increasingly successful adoption in the industry. DL models are now often used in all these types of problems or are at least being evaluated as a potential candidate model.

## **Objectives type: what is the motivation or goal to explain**

**Objectives type** refers to the motivation of using explainability in AI/ML projects. It has been argued that the number one objective of explainability is to gain trust by providing a sufficient understanding of the AI system behavior and uncovering vulnerabilities, biases, and flaws of the system. An additional motivation is to infer the causal relationship from the input and output prediction. Other objectives include improving the model accuracy through a better understanding of the inner workings of the AI/ML systems, and justifying the model behavior and decisions through transparent explanations when potentially severe consequences are involved. It is even possible to reveal unknown insights and rules that are based on explanations (<https://www.tandfonline.com/doi/full/10.1080/10580530.2020.1849465>). Overall, it is very desirable to break the black box so that when being used in a real production system, the AI/ML models and systems can be used with confidence.



## Method type: what is the specific post-hoc explanation method used

**Method type (post-hoc)** refers to post-hoc methods that are very relevant to the DL models. There are two major categories of post-hoc methods: perturbation-based and gradient-based. Recent work has started to unify these two approaches, although it is not yet widely applicable for practical usage (<https://teamcore.seas.harvard.edu/publications/towards-unification-and-robustness-perturbation-and-gradient-based>). The following is a brief discussion on these two types of methods:

- Perturbation-based methods leverage perturbations of individual instances to construct interpretable local approximations using linear models to explain the predictions. The most popular perturbation-based methods include **Local Interpretable Model-Agnostic Explanations (LIME)**, (<https://arxiv.org/pdf/1602.04938.pdf>), SHAP, and variants of LIME and SHAP such as BayesLIME and BayesSHAP, TreeSHAP, and many more (<https://towardsdatascience.com/what-are-the-prevailing-explainability-methods-3bc1a44f94df>). LIME can be used for tabular, image, and textual input data and is model agnostic. That's to say, LIME can be used for any type of classifiers (tree-based or DL models) regardless of the algorithms being used. SHAP uses principles from cooperative game theory to identify the contribution of different features to the prediction in order to quantify the impact of each feature. SHAP produces a so-called shapely value, which is the average of all the marginal contributions to all possible coalitions or combinations of different features. It works well for many types of models, including DL models, although the computational time could be much faster for tree-based models such as XGBoost or LightGBM (<https://github.com/slundberg/shap>).
- Gradient-based methods, such as SmoothGrad (<https://arxiv.org/abs/1706.03825>) and Integrated Gradients (<https://towardsdatascience.com/understanding-deep-learning-models-with-integrated-gradients-24ddce643dbf>), leverages gradients computed with respect to input dimensions of individual instances to explain model predictions. They can be applied to both image and textual input data, although sometimes, textual input could suffer a manipulation or adversary attack (<https://towardsdatascience.com/limitations-of-integrated-gradients-for-feature-attribution-ca2a50e7d269>), which will change the feature importance undesirably.

Note that there are additional types of methods such as counterfactual (<https://christophm.github.io/interpretable-ml-book/counterfactual.html>) or prototype-based methods (<https://christophm.github.io/interpretable-ml-book/proto.html>), which we will not cover in this book.

Having discussed the many dimensions of explainability, it is important to know that XAI is still an emerging area ([https://fairlyaccountable.org/aaai-2021-tutorial/doc/AAAI\\_slides\\_final.pdf](https://fairlyaccountable.org/aaai-2021-tutorial/doc/AAAI_slides_final.pdf)) and it is sometimes even difficult to find agreement among different explainability methods when applying to the same dataset or models (see a recent study on the topic of disagreement problems in explainable ML from the practitioners' perspective: <https://arxiv.org/abs/2202.01602>). In the end, it does require some experimentation to find out which explainability provides the human validated explanations that are meeting the requirements for a specific prediction task in the real world.

In the next two sections of this chapter, we will focus on providing some hands-on experiments using some popular and emerging toolkits to learn how to do explainability.

## Exploring the SHAP Explainability toolbox

For our learning purpose, let's review some popular explainability toolboxes while experimenting with some examples. Based on the number of GitHub stars (16,000 as of April 2022, <https://github.com/slundberg/shap>), SHAP is the most widely used and integrated open source model explainability toolbox. It is also the foundation explanation tool that is integrated with MLflow. Here, we would like to run a small experiment to get some hands-on experience on how this works. Let's use a sentimental analysis NLP model to explore how SHAP can be used for explaining the model behavior:

1. Set up the virtual environment on your local environment after checking out this chapter's code from GitHub. Running the following command will create a new virtual environment called `dl-explain`:

```
conda env create -f conda.yaml
```

This will install SHAP and its related dependencies such as `matplotlib` in this virtual environment. Once this virtual environment is created, activate this virtual environment by running the following command:

```
conda activate dl-explain
```

Now, we are ready to run the experiment with SHAP.

2. You can check out the `shap_explain.ipynb` notebook to follow through with the exploration. The first step in this notebook is to import the relevant Python libraries:

```
import transformers
import shap
from shap.plots import *
```

These imports will allow us to use the Hugging Face transformers pipeline API to get a pre-trained NLP model and SHAP functions.

3. We then create `dl_model` using the transformers pipeline API for `sentiment_analysis`. Note this is a pretrained pipeline so we can use this without additional finetuning. The default transformer model used in this pipeline is `distilbert-base-uncased-finetuned-sst-2-english` (<https://huggingface.co/distilbert-base-uncased-finetuned-sst-2-english>):

```
dl_model = transformers.pipeline(
    'sentiment-analysis', return_all_scores=True)
```

This will produce a model ready to predict positive or negative sentiment for an input sentence.

4. Try this `dl_model` with two input sentences and see whether the output makes sense:

```
dl_model(
    ["What a great movie! ...if you have no taste.",
     "Not a good movie to spend time on."])
```

This will produce an output of the labels and probability scores for each sentence as follows:

```
[[{ 'label': 'NEGATIVE', 'score': 0.00014734962314832956 },
  { 'label': 'POSITIVE', 'score': 0.9998526573181152 }],
 [{ 'label': 'NEGATIVE', 'score': 0.9997993111610413 },
  { 'label': 'POSITIVE', 'score': 0.00020068213052581996 }]]
```

It seems that the first sentence was predicted with a high probability to be `POSITIVE`, and the second sentence was predicted with a high probability to be `NEGATIVE`. Now, if we take a deep look at the first sentence, we may think the model prediction was incorrect, as there is a subtle negative emotion in the second part of the sentence (`no taste`). So, we want to know why the model made such a prediction. This is where model explainability comes into play.

5. Now, let's use the SHAP API, `shap.Explainer`, to get the Shapley values for the two sentences we are interested in explaining:

```
explainer = shap.Explainer(dl_model)
shap_values = explainer(["What a great movie! ...if you
have no taste.", "Not a good movie to spend time on."])
```

6. Once we have `shap_values`, we can visualize the Shapley values using different visualization techniques. The first one is to use `shap.plot.text` to visualize the first sentence's Shapley values when the prediction label is POSITIVE:

```
shap.plots.text(shap_values[0, :, "POSITIVE"])
```

This will produce the plot as follows:

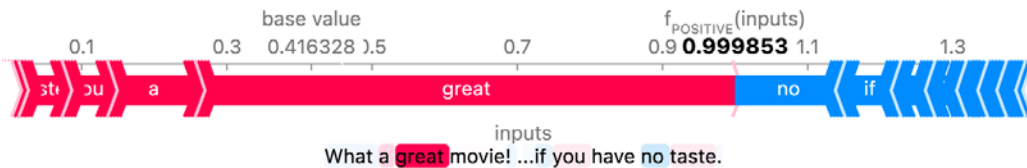


Figure 9.2 – SHAP visualization for sentence 1 with a positive prediction

As can be seen in *Figure 9.2*, the word `great` has a very large SHAP value that dominates the influence of the final prediction, while the word `no` has less effect on the final prediction. This results in the final prediction result of POSITIVE. So, what about the second sentence with a NEGATIVE prediction? Running the following command will produce a similar plot:

```
shap.plots.text(shap_values[1, :, "NEGATIVE"])
```

This command creates the following plot:

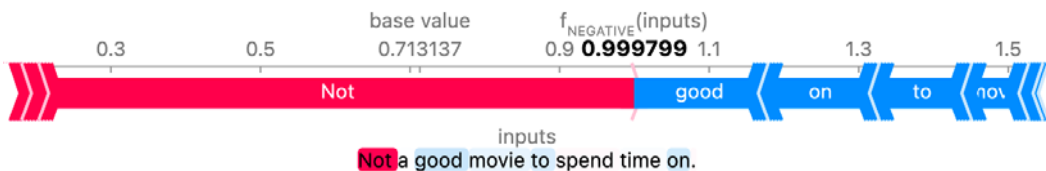


Figure 9.3 – SHAP visualization for sentence 2 with a negative prediction

As can be seen from *Figure 9.3*, the word `Not` has a strong influence on the final prediction, while the word `good` has a very small influence, resulting in the final prediction of a NEGATIVE sentiment. This makes a lot of sense, which is a good explanation of the model's behavior.

7. We can also visualize `shap_values` using different plots. A common one is the bar plot, which plots the feature contribution to the final prediction. Running the following command will produce a plot for the first sentence:

```
bar(shap_values[0, :, 'POSITIVE'])
```

This will produce a bar chart as follows:

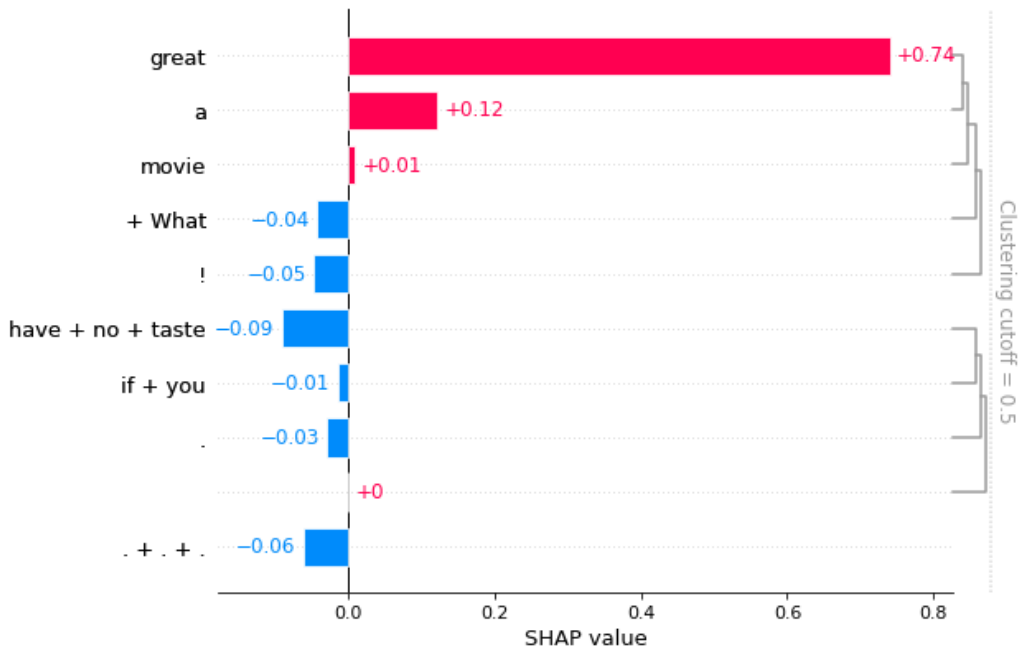


Figure 9.4 – SHAP bar chart for sentence 1 with a positive prediction

As can be seen from *Figure 9.4*, the chart ranks the most important features from top to bottom, where the top ones with a positive influence on the final prediction are plotted on the positive side of the  $x$  axis, while the negative contribution is plotted on the negative side of the  $x$  axis. The  $x$  axis is the value of each token or word's SHAP value with a sign (+ or -). This clearly shows the word `great` is a strong positive factor that impacts the final prediction, while `have no taste` has some negative effect but not enough to change the direction of the final prediction.

Similarly, we can plot a bar chart for the second sentence as follows:

```
bar(shap_values[1, :, 'NEGATIVE'])
```

This will produce the following bar chart:

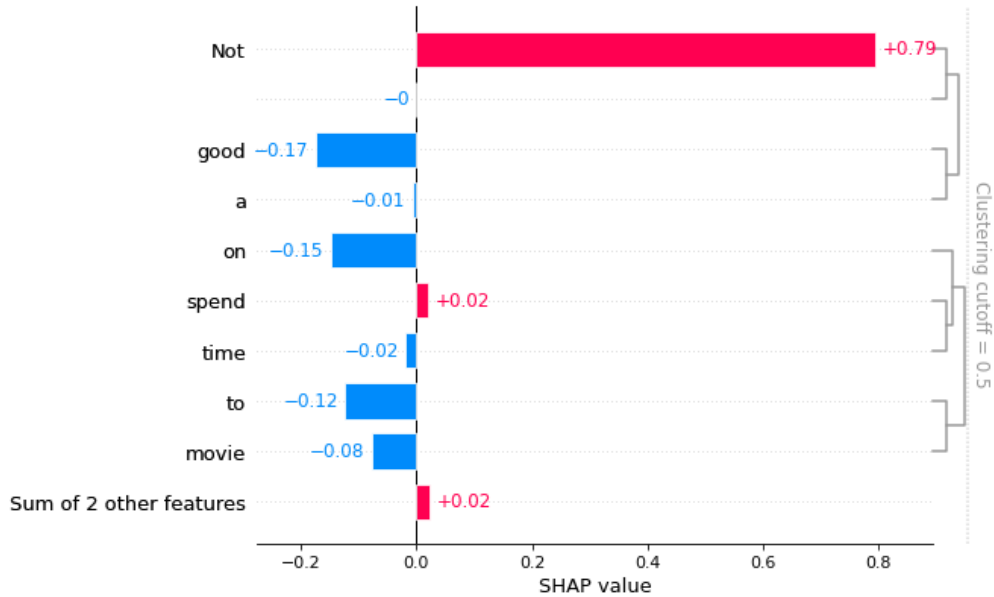


Figure 9.5 – SHAP bar chart for sentence 2 with a negative prediction

As can be seen from *Figure 9.5*, the word `Not` has a strong contribution to the final prediction, while the word `good` is second. These two words have the opposite effect on the final prediction, but apparently, the word `Not` is much stronger and has a much larger SHAP value.

If you have followed along with this example and seen the SHAP charts in your notebook, congratulations! This means you have successfully run the SHAP Explainability tool to explain the DL transformer model for the NLP text sentiment analysis.

Let's further explore another popular explainability tool to see how they perform different explanations.

## Exploring the Transformers Interpret toolbox

As we already reviewed in the first section of this chapter, there are two major methods: perturbation-based and gradient-based post-hoc explainability tools. SHAP belongs to the perturbation-based family. Now, let's look at a gradient-based toolbox called **Transformers Interpret** (<https://github.com/cdpierse/transformers-interpret>). This is a relatively new tool, but it is built on top of a unified model interpretability and understanding library for PyTorch called **Captum** (<https://github.com/pytorch/captum>), which provides a unified API to use either perturbation or gradient-based tools (<https://arxiv.org/abs/2009.07896>). Transformers Interpret further simplifies the API of Captum so that we can quickly explore gradient-based explainability methods to get some hands-on experience.

To get started, first make sure you already have the `dl-explain` virtual environment set up and activated, as described in the previous section. Then, we can use the same Hugging Face transformer sentiment analysis model to explore some NLP sentiment classification examples. Then, we can perform the following steps to learn how to use Transformers Interpret to do the model explanation. You may want to check out the `gradient_explain.ipynb` notebook to follow the instructions:

1. Import relevant packages into the notebook as follows:

```
from transformers import
    AutoModelForSequenceClassification, AutoTokenizer
from transformers_interpret import
    SequenceClassificationExplainer
```

This will use Hugging Face's transformer model and tokenizer, as well as the explainability function from `transformers_interpret`.

2. Create the model and the tokenizer using the same pre-trained model as previous section, which is the `distilbert-base-uncased-finetuned-sst-2-english` model:

```
model_name = "distilbert-base-uncased-finetuned-sst-2-english"
model = AutoModelForSequenceClassification.from_pretrained(model_name)
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

Now that we have the model and tokenizer, we can create an explainability variable using the `SequenceClassificationExplainer` API.

3. Create an explainer and give an example sentence to get the word attribution from the explainer:

```
cls_explainer = SequenceClassificationExplainer(model,
tokenizer)
word_attributions = cls_explainer("Not a good movie to
spend time on.")
```

4. We can also get the prediction label before we check the word attributions by running the following command:

```
cls_explainer.predicted_class_name
```

This will produce a result of `Negative`, which means the prediction is a negative sentiment. So, let's see how the explainer provides an explanation for this prediction.

5. We can just display the `word_attributions` value, or we can visualize it. The value of `word_attributions` is as follows:

```
[(['CLS'], 0.0),
 ('not', 0.48004693667974174),
 ('a', -0.18190486629629668),
 ('good', -0.0005904323790053206),
 ('movie', 0.1484645593783762),
 ('to', 0.6242280981881405),
 ('spend', 0.38801686343155),
 ('time', 0.2720354453568708),
 ('on', 0.20949497995981806),
 ('.', 0.23730623499309145),
 ('[SEP]', 0.0)]
```

Figure 9.6 – Layered integrated gradient word attribution values with a negative prediction

As can be seen from *Figure 9.6*, using the layered integrated gradient method, which is the current explainer's default method implemented in the Transformers Interpret library, the word `not` contributed positively to the final prediction result, which is a negative sentiment. This makes sense. Notice that several other words, such as `to spend time on`, also have a strong positive influence on the final prediction. Given the cross-attention mechanism, it seems the model is trying to extract `not to spend time on` as the main attribution to the final prediction. Note we can also visualize these word attributions as follows:

```
cls_explainer.visualize("distilbert_viz.html")
```



This will produce the follow plot:

Legend: <span style="color: red;">■</span> Negative <span style="color: gray;">□</span> Neutral <span style="color: green;">■</span> Positive				
True Label	Predicted Label	Attribution Label	Attribution Score	Word Importance
0	NEGATIVE (1.00)	NEGATIVE	2.18	[CLS] not a good movie to spend time on . [SEP]

Figure 9.7 – Layered integrated gradient word attribution values with a negative prediction

As can be seen in *Figure 9.7*, it highlights the word importance of `not` `to` `spend` `time` `on` to positively impact the final negative prediction.

Now that we have experimented with both perturbation and gradient-based explainability methods, we have successfully completed our hands-on exploration of using the explainability tool for post-hoc local explanation.

Next, we will summarize what we learned in this chapter.

## Summary

In this chapter, we reviewed explainability in AI/ML through an eight-dimension categorization. Although this is not necessarily a comprehensive or exhaustive overview, this does give us a big picture of who to explain to, different stages and scopes to explain, various kinds of input and output formats of the explanation, common ML problems and objectives types, and finally, different post-hoc explainability methods. We then provided two concrete exercises to explore the SHAP and Transformers Interpret toolboxes, which can provide perturbation and gradient-based feature attribution explanations for NLP text sentiment DL models.

This gives us a solid foundation for using explainability tools for DL models. However, given the active development of XAI, this is only the beginning of using XAI in DL models. Additional explainability toolboxes such as TruLens (<https://github.com/truera/trulens>), Alibi (<https://github.com/SeldonIO/alibi>), Microsoft Responsible AI Toolbox (<https://github.com/microsoft/responsible-ai-toolbox>), and IBM AI Explainability 360 Toolkit (<https://github.com/Trusted-AI/AIX360>) are all in active development and worthy of investigation and future learning. Additional links are also provided in the *Further reading* section to help you continue to learn this topic.

Now that we know the fundamentals of explainability, in the next chapter, we will learn how to implement explainability in the MLflow framework so that we can provide a unified way to support explanation within the MLflow framework.

## Further reading

- *New frontiers in Explainable AI*: <https://towardsdatascience.com/new-frontiers-in-explainable-ai-af43bba18348>
- *Towards a Rigorous Science of Interpretable Machine Learning*: <https://arxiv.org/pdf/1702.08608.pdf>
- *The Toolkit Approach to Trustworthy AI*: <https://opendatascience.com/the-toolkit-approach-to-trustworthy-ai/>
- *A Framework for Learning Ante-hoc Explainable Models via Concepts*: <https://arxiv.org/abs/2108.11761>
- *Demystifying Post-hoc Explainability for ML models*: <https://spectra.mathpix.com/article/2021.09.00007/demystify-post-hoc-explainability>
- *A Look Into Global, Cohort and Local Model Explainability*: <https://towardsdatascience.com/a-look-into-global-cohort-and-local-model-explainability-973bd449969f>
- *What Are the Prevailing Explainability Methods?* <https://towardsdatascience.com/what-are-the-prevailing-explainability-methods-3bc1a44f94df>
- *Explainable Artificial Intelligence: Objectives, Stakeholders, and Future Research Opportunities*: <https://www.tandfonline.com/doi/full/10.1080/10580530.2020.1849465>



# 10

## Implementing DL Explainability with MLflow

The importance of **deep learning (DL)** explainability is now well established, as we learned in the previous chapter. In order to implement DL explainability in a real-world project, it is desirable to log the explainer and the explanations as artifacts, just like other model artifacts in the MLflow server, so that we can easily track and reproduce the explanation. The integration of DL explainability tools such as SHAP (<https://github.com/slundberg/shap>) with MLflow can support different implementation mechanisms, and it is important to understand how these integrations can be used for our DL explainability scenarios. In this chapter, we will explore several ways to integrate the SHAP explanations into MLflow by using different MLflow capabilities. As tools for explainability and DL models are both rapidly evolving, we will also highlight the current limitations and workarounds when using MLflow for DL explainability implementation. By the end of this chapter, you will feel comfortable implementing SHAP explanations and explainers using MLflow APIs for scalable model explainability.

In this chapter, we're going to cover the following main topics:

- Understanding current MLflow explainability integration
- Implementing SHAP explanations using the MLflow artifact logging API
- Implementing SHAP explainers using the MLflow pyfunc API

## Technical requirements

The following requirements are necessary to complete this chapter:

- MLflow full-fledged local server: This is the same one we have been using since *Chapter 3, Tracking Models, Parameters, and Metrics*.
- The SHAP Python library: <https://github.com/slundberg/shap>.
- Spark 3.2.1 and PySpark 3.2.1: See the details in the README.md file of this chapter's GitHub repository.
- Code from the GitHub repository for this chapter: <https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/tree/main/chapter10>.

## Understanding current MLflow explainability integration

MLflow has several ways to support explainability integration. When implementing explainability, we refer to two types of artifacts: explainers and explanations:

- An explainer is an explainability model, and a common one is a SHAP model that could be different kinds of SHAP explainers, such as **TreeExplainer**, **KernelExplainer**, and **PartitionExplainer** (<https://shap.readthedocs.io/en/latest/generated/shap.explainers.Partition.html>). For computational efficiency, we usually choose **PartitionExplainer** for DL models.
- An explanation is an artifact that shows some form of output from the explainer, which could be text, numerical values, or plots. Explanations can happen in offline training or testing, or can happen during online production. Thus, we should be able to provide an explainer for offline evaluation or an explainer endpoint for online queries if we want to know why the model provides certain predictions.

Here, we give a brief overview of the current capability as of MLflow version 1.25.1 (<https://pypi.org/project/mlflow/1.25.1/>). There are four different ways to use MLflow for explainability as follows:

- Use the `mlflow.log_artifact` API ([https://www.mlflow.org/docs/latest/python\\_api/mlflow.html#mlflow.log\\_artifact](https://www.mlflow.org/docs/latest/python_api/mlflow.html#mlflow.log_artifact)) to log relevant explanation artifacts such as bar plots and Shapley values arrays. This gives maximum flexibility for logging explanations. This can be used either offline as batch processing or online when we automatically log a SHAP bar plot for a certain prediction. Note that logging an explanation for each prediction during online production scenarios is expensive, so we should provide a separate explanation API for on-demand queries.
- Use the `mlflow.pyfunc.PythonModel` API ([https://www.mlflow.org/docs/latest/python\\_api/mlflow.pyfunc.html#mlflow.pyfunc.PythonModel](https://www.mlflow.org/docs/latest/python_api/mlflow.pyfunc.html#mlflow.pyfunc.PythonModel)) to create an explainer that can be logged and loaded with MLflow's `pyfunc` methods, `mlflow.pyfunc.log_model` for logging and `mlflow.pyfunc.load_model` or `mlflow.pyfunc.spark_udf` for loading an explainer. This gives us maximum flexibility to create customized explainers as MLflow generic `pyfunc` models and can be used for either offline batch explanation or online as an **Explanation as a Service (EaaS)**.
- Use the `mlflow.shap` API ([https://www.mlflow.org/docs/latest/python\\_api/mlflow.shap.html](https://www.mlflow.org/docs/latest/python_api/mlflow.shap.html)). This has some limitations. For example, the `mlflow.shap.log_explainer` method only supports scikit-learn and PyTorch models. The `mlflow.shap.log_explanation` method only supports `shap.KernelExplainer` (<https://shap-lrjball.readthedocs.io/en/latest/generated/shap.KernelExplainer.html>). This is very computationally intensive, as the computing time grows exponentially with respect to the number of features; thus, it is not feasible to compute explanations for even a moderate size dataset (see a posted GitHub issue <https://github.com/mlflow/mlflow/issues/4071>). The existing examples provided by MLflow are for classical ML models in scikit-learn packages such as linear regression or random forest, with no DL model explainability examples (<https://github.com/mlflow/mlflow/tree/master/examples/shap>). We will show in later sections of this chapter that this API currently does not support the transformers-based SHAP explainers and explanations, thus we will not use this API in this chapter. We will highlight some of the issues as we walk through our examples in this chapter.

- Use the `mlflow.evaluate` API ([https://www.mlflow.org/docs/latest/python\\_api/mlflow.html#mlflow.evaluate](https://www.mlflow.org/docs/latest/python_api/mlflow.html#mlflow.evaluate)). This can be used for evaluation after the model is already trained and tested. This is an experimental feature and might change in the future. It supports MLflow `pyfunc` models. However, it has some limitations in that the evaluation dataset label values must be numeric or Boolean, all feature values must be numeric, and each feature column must only contain scalar values (<https://www.mlflow.org/docs/latest/models.html#model-evaluation>). Again, existing examples provided by MLflow are only for classical ML models in scikit-learn packages (<https://github.com/mlflow/mlflow/tree/master/examples/evaluation>). We could use this API to just log the classifier metrics for an NLP sentiment model, but the explanation part will be skipped automatically by this API because it requires a feature column containing scalar values (an NLP model input is a text input). Thus, this is not applicable to the DL model explainability we need. So, we will not use this API in this chapter.

Given that some of these APIs are still experimental and are still evolving, users should be aware of the limitations and workarounds to successfully implement explainability with MLflow. For DL model explainability, as we will learn in this chapter, it is quite challenging to implement using MLflow as the MLflow integration with SHAP is still a work-in-progress as of MLflow version 1.25.1. In the following sections, we will learn when and how to use these different APIs to implement explanations and log and load explainers for DL models.

## Implementing a SHAP explanation using the MLflow artifact logging API

MLflow has a generic tracking API that can log any artifact: `mlflow.log_artifact`. However, the examples given in the MLflow documentation usually use scikit-learn and tabular numerical data for training, testing, and explaining. Here, we want to show how to use `mlflow.log_artifact` for an NLP sentimental DL model to log relevant artifacts, such as Shapley value arrays and Shapley value bar plots. You can check out the Python VS Code notebook, `shap_mlflow_log_artifact.py`, in this chapter's GitHub repository ([https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter10/notebooks/shap\\_mlflow\\_log\\_artifact.py](https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter10/notebooks/shap_mlflow_log_artifact.py)) to follow along with the steps:

1. Make sure you have the prerequisites, including a local full-fledged MLflow server and the conda virtual environment, ready. Follow the instructions in the README.md (<https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter10/README.md>) file in the Chapter 10 folder to get these ready.
2. Make sure you activate the `chapter10-dl-explain` virtual environment as follows before you start running any code in this chapter:

```
conda activate chapter10-dl-explain
```

3. Import the relevant libraries at the beginning of the notebook as follows:

```
import os
import matplotlib.pyplot as plt
import mlflow
from mlflow.tracking import MlflowClient
from mlflow.utils.file_utils import TempDir
import shap
import transformers
from shap.plots import *
import numpy as np
```



4. The next step is to set up some environment variables. The first three environment variables are for the local MLflow URIs, and the fourth is for disabling a Hugging Face warning that arises due to a known Hugging Face tokenization issue:

```
os.environ["AWS_ACCESS_KEY_ID"] = "minio"  
os.environ["AWS_SECRET_ACCESS_KEY"] = "minio123"  
os.environ["MLFLOW_S3_ENDPOINT_URL"] = "http://  
localhost:9000"  
os.environ["TOKENIZERS_PARALLELISM"] = "False"
```

5. We will also need to set up the MLflow experiment and show the MLflow experiment ID as an output on the screen:

```
EXPERIMENT_NAME = "dl_explain_chapter10"  
mlflow.set_tracking_uri('http://localhost')  
mlflow.set_experiment(EXPERIMENT_NAME)  
experiment = mlflow.get_experiment_by_name(EXPERIMENT_  
NAME)  
print("experiment_id:", experiment.experiment_id)
```

If you have been running the notebook, you should see an output like the following:

```
experiment_id: 14
```

This means the MLflow experiment ID for the experiment name `dl_explain_chapter10` is 14. Note that, you could also set the MLflow tracking URI as an environment variable as follows:

```
export MLFLOW_TRACKING_URI=http://localhost
```

Here, we use MLflow's `mlflow.set_tracking_uri` API to define the URI location instead. Either way is fine.

6. Now we can create a DL model to classify a sentence into either positive or negative sentiment using Hugging Face's transformer pipeline API. Since this is already fine-tuned, we will focus on how to get the explainer and explanation for the model, rather than focusing on how to train or finetune a model:

```
dl_model = transformers.pipeline('sentiment-analysis',  
return_all_scores=False)  
explainer = shap.Explainer(dl_model)  
shap_values = explainer(["Not a good movie to spend time  
on.", "This is a great movie."])
```

The code snippets create a sentiment analysis model, `dl_model`, and then create a SHAP explainer for this model. Then we provide a list of two sentences for this explainer to get the `shap_values` object. This will be used for logging in MLflow.

Given the `shap_values` object, we can now start a new MLflow run and log both the Shapley values and the bar plot that we saw in the previous chapter (*Chapter 9, Fundamentals of Deep Learning Explainability*). The first line of code makes sure all active MLflow runs are ended. This is useful if we want to rerun this block of code multiple times interactively:

```
mlflow.end_run()
```

Then we define two constants. One, `artifact_root_path`, is for the root path in the MLflow artifact store, which will be used to store all the SHAP explanation objects. The other, `shap_bar_plot`, is for the artifact filename, which will be used for the bar plot figure:

```
artifact_root_path = "model_explanations_shap"
artifact_file_name = 'shap_bar_plot'
```

7. We then start a new MLflow run, under which we will generate and log three SHAP files into the MLflow artifact store under the path `model_explanations_shap`:

```
with mlflow.start_run() as run:
    with TempDir() as temp_dir:
        temp_dir_path = temp_dir.path()
        print("temp directory for artifacts: {}".format(temp_dir_path))
```

We also need to have a temporary local directory, as shown in the preceding code snippet to first save the SHAP files, and then log those files to the MLflow server. If you have run the notebook up to this point, you should see a temporary directory in the output like the following:

```
temp directory for artifacts: /var/folders/51/
whxjy4r92dx18788yp11ycyr0000gp/T/tmpgw520wu1
```

8. Now we are ready to generate the SHAP files and save them. The first one is the bar plot, which is a little bit tricky to save and log. Let's walk through the following code to understand how we do this:

```
try:
    plt.clf()
    plt.subplots_adjust(bottom=0.2, left=0.4)
```

```

shap.plots.bar(shap_values[0, :, "NEGATIVE"],
               show=False)
plt.savefig(f"{temp_dir_path}/{artifact_file_name}")
finally:
    plt.close(plt.gcf())
mlflow.log_artifact(f"{temp_dir_path}/{artifact_file_
name}.png", artifact_root_path)

```

Note that we are using `matplotlib.pyplot`, which was imported as `plt` to first clear the figure using `plt.clf()` and then create a subplot with some adjustments. Here, we define `bottom=0.2`, which means the position of the bottom edge of the subplots is at 20% of the figure height. Similarly, we adjust the left edge of the subplot. Then we use the `shap.plots.bar` SHAP API to plot the bar plot for the first sentence's feature contribution to the prediction, but with the `show` parameter to be `False`. This means, we will not see the plot in the interactive run, but the figure is stored in the `pyplot plt` variable, which can then be saved using `plt.savefig` to a local temporary directory with the filename prefix `shap_bar_plot.pyplot` will automatically add the file extension `.png` to the file once it is saved. So, this will save a local image file called `shap_bar_plot.png` in the temporary folder. The last statement calls MLflow's `mlflow.log_artifact` to upload this PNG file to the MLflow tracking server's artifact store in the root folder, `model_explanations_shap`. We also need to make sure that we close the current figure by calling `plt.close(plt.gcf())`.

9. In addition to logging the `shap_bar_plot.png` to the MLflow server, we also want to log the Shapley `base_values` array and `shap_values` array as NumPy arrays into the MLflow track server. This can be done through the following statements:

```

np.save(f"{temp_dir_path}/shap_values",
        shap_values.values)
np.save(f"{temp_dir_path}/base_values",
        shap_values.base_values)
mlflow.log_artifact(
    f"{temp_dir_path}/shap_values.npy",
    artifact_root_path)
mlflow.log_artifact(
    f"{temp_dir_path}/base_values.npy",
    artifact_root_path)

```

This will first save a local copy of `shap_values.npy` and `base_values.npy` in the local temporary folder and then upload it to the MLflow tracking server's artifact store.

- If you followed the notebook up until here, you should be able to verify in the local MLflow server whether these artifacts are successfully stored. Go to the MLflow UI at the localhost – `http://localhost/` and then find the experiment `dl_explain_chapter10`. You should then be able to find the experiment you just ran. It should look something like *Figure 10.1*, where you can find three files in the `model_explanations_shap` folder: `base_values.npy`, `shap_bar_plot.png`, and `shap_values.npy`. *Figure 10.1* shows the bar plot of feature contribution of different tokens or words for the prediction result of the sentence – Not a good movie to spend time on. The URL for this experiment page is something like the following:

```
http://localhost/#/experiments/14/
runs/10f0655189f740aeb813a015f1f6e115
```

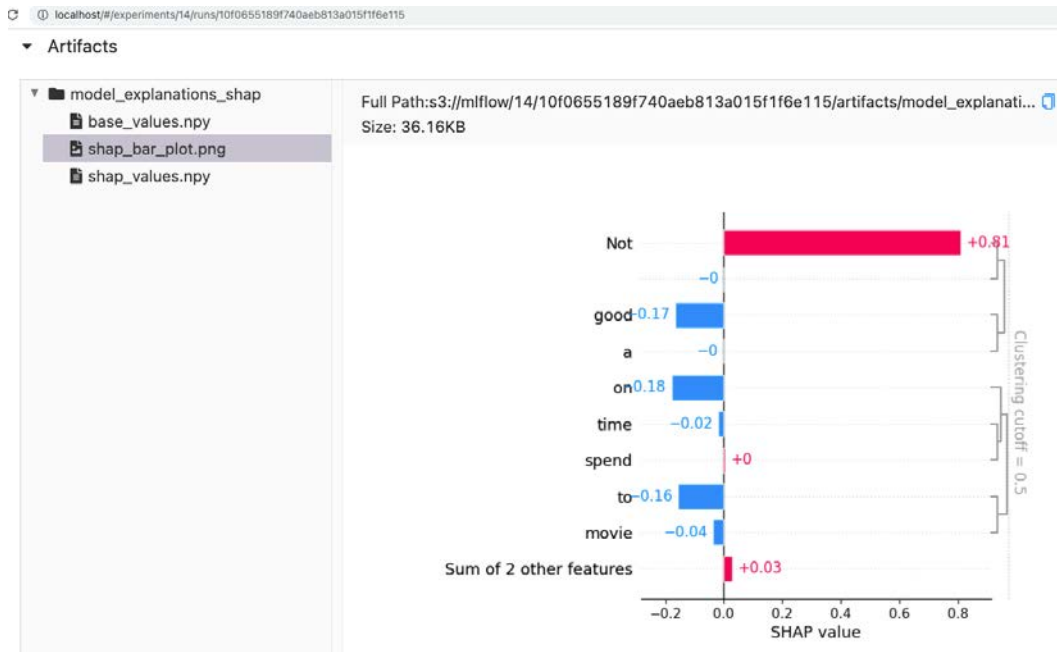


Figure 10.1 – MLflow `log_artifact` API saves the SHAP bar plot as an image in the MLflow tracking server

Alternatively, you can also use code to programmatically download these files stored in the MLflow tracking server and check them locally. We provide such code in the last cell of the notebook.

11. If you run the last cell block of the notebook code, which is to download the three files from the MLflow server we just saved and print them out, you should be able to see the following output, as displayed in *Figure 10.2*. The mechanism to download artifacts from the MLflow tracking server is to use the `MlflowClient().download_artifacts()` API, where you provide the MLflow run ID (in our example, it is `10f0655189f740aeb813a015f1f6e115`) and the artifact root path `model_explanations_shap` as the parameters to the API:

```
downloaded_local_path = MlflowClient().download_artifacts(run.info.run_id, artifact_root_path)
```

This will download all files in `model_explanations_shap` on the MLflow tracking server to a local path, which is the return variable `downloaded_local_path`:

```
# downloaded_local_path:
/var/folders/51/whxjy4r92dx18788yp11ycyr0000gp/T/tmpcuaqgktb/model_explanations_shap

# base_values:
[[0.71313685 0.          ]
 [0.69126123 0.          ]]

# shap_values:
[array([[ -2.60770321e-08,  2.23517418e-08],
        [ 8.09130404e-01, -7.33891942e-01],
        [-9.41950828e-04,  6.17101789e-03],
        [-1.65014420e-01,  2.59696975e-01],
        [-3.73203177e-02,  1.01590425e-01],
        [-1.55425506e-01,  1.55652583e-01],
        [ 4.62501030e-03, -4.64580208e-02],
        [-1.88832553e-02,  3.11851613e-02],
        [-1.78509811e-01,  2.20051158e-01],
        [ 2.90023014e-02,  6.00261986e-03],
        [ 2.98023224e-08,  0.00000000e+00]])]
[array([[ -1.11758709e-08,  0.00000000e+00],
        [-3.15686874e-02,  1.31188333e-03],
        [-7.42454268e-02,  1.01561680e-01],
        [-2.39816446e-01,  3.92191380e-01],
        [-2.83805508e-01,  4.31868218e-01],
        [-8.83626081e-02,  7.16432258e-02],
        [ 2.65375450e-02,  1.29778683e-03],
        [-8.94069672e-08,  0.00000000e+00]])]
```

Figure 10.2 – Download the SHAP `base_values` and `shap_values` array from the MLflow tracking server to a local path and display them

To display the two NumPy arrays, we need to call NumPy's `load` API to load them and then print them:

```
base_values = np.load(os.path.join(downloaded_local_path,
                                   "base_values.npy"), allow_pickle=True)
shap_values = np.load(os.path.join(downloaded_local_path,
                                   "shap_values.npy"), allow_pickle=True)
```

Note that we need to set the `allow_pickle` parameter to `True` when calling the `np.load` API so that NumPy can correctly load these files back into memory.

While you can run this notebook interactively in the VS Code environment, you can also run it in the command line as follows:

```
python shap_mlflow_log_artifact.py
```

This will produce all the output in the console and log all the artifacts into the MLflow server as we have seen in our interactive running of the notebook.

If you have run the code so far, congratulations on the successful completion of implementing logging SHAP explanations to the MLflow tracking server using MLflow's `mlflow.log_artifact` API!

Although the process of logging all the explanations seems a little bit long, this approach does have the advantage of having no dependency on what kind of explainer is used since the explainer is defined outside of the MLflow artifact logging API.

In the next section, we will see how to use the built-in `mlflow.pyfunc.PythonModel` API to log a SHAP explainer as an MLflow model and then deploy as an endpoint or use it in a batch mode as if it is a generic MLflow `pyfunc` model.

## Implementing a SHAP explainer using the MLflow pyfunc API

As we know from the previous section, a SHAP explainer can be used offline whenever needed by creating a new instance of an explainer using SHAP APIs. However, as the underlying DL models are often logged into the MLflow server, it is desirable to also log the corresponding explainer into the MLflow server, so that we not only keep track of the DL models, but also their explainers. In addition, we can use the generic MLflow `pyfunc` model logging and loading APIs for the explainer, thus unifying access to DL models and their explainers.

In this section, we will learn step-by-step how to implement a SHAP explainer as a generic MLflow pyfunc model and how to use it for offline and online explanation. We will break the process up into three subsections:

- Creating and logging an MLflow pyfunc explainer
- Deploying an MLflow pyfunc explainer for an EaaS
- Using an MLflow pyfunc explainer for batching explanation

Let's start with the first subsection on creating and logging a MLflow pyfunc explainer.

## Creating and logging an MLflow pyfunc explainer

In order to follow this section, please check out `nlp_sentiment_classifier_explainer.py` in the GitHub repository ([https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter10/pipeline/nlp\\_sentiment\\_classifier\\_explainer.py](https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter10/pipeline/nlp_sentiment_classifier_explainer.py)):

1. First, by subclassing `mlflow.pyfunc.PythonModel`, we can create a customized MLflow model that encapsulates a SHAP explainer. So, let's declare this class as follows:

```
class SentimentAnalysisExplainer(mlflow.pyfunc.  
PythonModel):
```

2. Next, we need to instantiate an explainer. Instead of creating an explainer in the `init` method of this class, we will use the `load_context` method to load a SHAP explainer for the Hugging Face NLP sentiment analysis classifier, as follows:

```
def load_context(self, context):  
    from transformers import pipeline  
    import shap  
    self.explainer = shap.Explainer(pipeline('sentiment-  
analysis', return_all_scores=True))
```

This will create a SHAP explainer whenever this `SentimentAnalysisExplainer` class is executed. Note that the sentiment classifier is a Hugging Face pipeline object, with the `return_all_scores` parameter set to `True`. This means that this will return the label and probability score for both positive and negative sentiment of each input text.

**Avoid Runtime Errors for SHAP explainers**

If we implement `self.explainer` in the `init` method in this class, we will encounter a runtime error related to the SHAP package's `_masked_model.py` file, which complains about **`TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'`**. Any code implemented in the `PythonModel` class's `init` method will be serialized by MLflow, so it is clear that this runtime error comes from MLflow's serialization. However, implementing `self.explainer` in the `load_context` function avoids MLflow's serialization, and works correctly when invoking this explainer at runtime.

3. We will then implement the `sentiment_classifier_explanation` method, which takes an input of a pandas DataFrame row and produces a pickled `shap_values` output as an explanation for a single row of text input:

```
def sentiment_classifier_explanation(self, row):
    shap_values = self.explainer([row['text']])
    return [pickle.dumps(shap_values)]
```

Note that we need to use a pair of square brackets to enclose the `row['text']` value so that it becomes a list not just a single value. This is because this SHAP explainer expects a list of texts, not just a single string. If we don't enclose the value within the square brackets, then the explainer will split the entire string character by character, treating each character as if it is a word, which is not what we want. Once we get the Shapley values as the output from the explainer as `shap_values`, we then need to serialize them using `pickle.dumps` before returning to the caller. MLflow pyfunc model input and output signature do not support complex object without serialization, so this pickling step makes sure that the model output signature is MLflow compliant. We will see the definition of this MLflow pyfunc explainer's input and output signature in *step 5* shortly.

4. Next, we need to implement the required `predict` method for this class. This will apply the `sentiment_classifier_explanation` method to the entire input pandas DataFrame, as follows:

```
def predict(self, context, model_input):
    model_input[['shap_values']] = model_input.apply(
        self.sentiment_classifier_explanation, axis=1,
        result_type='expand')
    model_input.drop(['text'], axis=1, inplace=True)
    return model_input
```



This will produce a new column named `shap_values` for each row of the input pandas DataFrame in the `text` column. We then drop the `text` column and return a single-column `shap_values` DataFrame as the final prediction result: in this case, the explanation results as a DataFrame.

- Now that we have the `SentimentAnalysisExplainer` class implementation, we can use the standard MLflow pyfunc model logging API to log this model into the MLflow tracking server. Before doing the MLflow logging, let's make sure we declare this explainer's model signature, as follows:

```
input = json.dumps([{'name': 'text', 'type': 'string'}])
output = json.dumps([{'name': 'shap_values', 'type':
    'string'}])
signature = ModelSignature.from_dict({'inputs': input,
    'outputs': output})
```

These statements declare that the input is a DataFrame with a single string type `text` column and the output is a DataFrame with a single string type `shap_values` column. Recall that this `shap_values` column is a pickled serialized bytes string, which contains the Shapley values object.

- Finally, we can implement the explainer logging step using the `mlflow.pyfunc.log_model` method in a task method, as follows:

```
with mlflow.start_run() as mrun:
    mlflow.pyfunc.log_model(
        artifact_path=MODEL_ARTIFACT_PATH,
        conda_env=CONDA_ENV,
        python_model=SentimentAnalysisExplainer(),
        signature=signature)
```

There are four parameters in the `log_model` method that we use. The `MODEL_ARTIFACT_PATH` is the name of the folder in the MLflow tracking server where the explainer will be stored. Here, the value is defined as `nlp_sentiment_classifier_explainer` in the Python file you checked out. `CONDA_ENV` is the `conda.yaml` file in this chapter's root folder. The `python_model` parameter is the `SentimentAnalysisExplainer` class we just implemented, and `signature` is the explainer input and output signature we defined.

- Now we are ready to run this whole file as follows in the command line:

```
python nlp_sentiment_classifier_explainer.py
```

Assuming you have the local MLflow tracking server and environment variables set up correctly by following the `README.md` file for this chapter in the GitHub repository, this will produce the following two lines in the console output:

```
2022-05-11 17:49:32,181 Found credentials in environment
variables.
2022-05-11 17:49:32,384 finished logging
nlp_sentiment_classifier_explainer run_id:
ad1edb09e5ea4d8ca0332b8bc2f5f6c9
```

This means we have successfully logged the explainer in our local MLflow tracking server.

- Go to the MLflow web UI at `http://localhost/` in the web browser and click the `dl_explain_chapter10` experiment folder. You should be able to find this run and the logged explainer in the `Artifacts` folder under `nlp_sentiment_classifier_explainer`, which should look as shown in *Figure 10.3*:

▼ Artifacts

▼ nlp\_sentiment\_classifier\_explainer

- MLmodel
- conda.yaml
- python\_model.pkl
- requirements.txt

Full Path: `s3://mlflow/14/ad1edb09e5ea4d8ca0332b8bc2f5f6c9/artifacts/nlp_sentiment_...`

Size: 503B

---

`artifact_path: nlp_sentiment_classifier_explainer`

`flavors:`

- `python_function:`
- `cloudpickle_version: 2.0.0`
- `env: conda.yaml`
- `loader_module: mlflow.pyfunc.model`
- `python_model: python_model.pkl`
- `python_version: 3.8.10`

`mlflow_version: 1.25.1`

`model_uuid: de5fe4836a924a1e95dd198456eed48`

`run_id: ad1edb09e5ea4d8ca0332b8bc2f5f6c9`

`signature:`

- `inputs: '[{"name": "text", "type": "string"}]'`
- `outputs: '[{"name": "shap_values", "type": "string"}]'`

`utc_time_created: '2022-05-12 00:49:32.053574'`

Figure 10.3 – A SHAP explainer is logged as an MLflow pyfunc model

Notice that the `MLmodel` metadata shown in *Figure 10.3* does not differ much from the normal DL inference pipeline that we logged before as an MLflow pyfunc model except for the `artifact_path` name and the `signature`. That's the advantage of using this approach because now we can use the generic MLflow pyfunc model methods to load this explainer or deploy it as a service.

**Problems with the `mlflow.shap.log_explainer` API**

As we mentioned earlier, MLflow has a `mlflow.shap.log_explainer` API that provides a method to log an explainer. However, this API does not support our NLP sentiment classifier explainer because our NLP pipeline is not a known model flavor that MLflow currently supports. Thus even though `log_explainer` can write this explainer object into the tracking server, when loading the explainer back into memory using the `mlflow.shap.load_explainer` API, it will fail with the following error message:

**TypeError: \_\_init\_\_() missing 1 required positional argument: 'pipeline'.**

Thus, we avoid using the `mlflow.shap.log_explainer` API in this book.

Now that we have a logged explainer, we can use it in two ways: deploy it into a web service so that we can create an endpoint to establish an EaaS, or load the explainer directly through MLflow `pyfunc.load_model` or `spark_udf` method using the MLflow `run_id`. Let's start with the web service deployment by setting up a local web service.

## Deploying an MLflow `pyfunc` explainer for an EaaS

We can set up a local EaaS in a standard MLflow way since now the SHAP explainer is just like a generic MLflow `pyfunc` model. Perform the following steps to see how this can be implemented locally:

1. Run the following MLflow command to set up a local web service for the explainer we just logged. The `run_id` in this example is `ad1edb09e5ea4d8ca0332b8bc2f5f6c9`:

```
mlflow models serve -m runs:/
ad1edb09e5ea4d8ca0332b8bc2f5f6c9/nlp_sentiment_
classifier_explainer
```

This will produce the following console output:

```
2022/05/11 19:03:52 INFO mlflow.models.cli: Selected backend for flavor 'python_function'
2022/05/11 19:03:52 INFO mlflow.utils.conda: Conda environment mlflow-d596c83405a908c55fc568031510fa216a5583ec already exists.
2022/05/11 19:03:52 INFO mlflow.pyfunc.backend: == Running command 'source /Users/yongliu/opt/miniconda3/bin/./etc/profile.d/conda.sh && conda activate mlflow-d596c83405a908c55fc568031510fa216a5583ec 1&2 && exec gunicorn --timeout=60 -b 127.0.0.1:5000 -w 1 ${GUNICORN_CMD_ARGS} -- mlflow.pyfunc.scoring_server.wsgi:app'
[2022-05-11 19:03:53 -0700] [58531] [INFO] Starting gunicorn 20.1.0
[2022-05-11 19:03:53 -0700] [58531] [INFO] Listening at: http://127.0.0.1:5000 (58531)
[2022-05-11 19:03:53 -0700] [58531] [INFO] Using worker: sync
[2022-05-11 19:03:53 -0700] [58538] [INFO] Booting worker with pid: 58538
No model was supplied, defaulted to distilbert-base-uncased-finetuned-sst-2-english (https://huggingface.co/distilbert-base-uncased-finetuned-sst-2-english)
```

Figure 10.4 – SHAP EaaS console output



## Using an MLflow pyfunc explainer for batch explanation

There are two ways to implement offline batch explanation using an MLflow pyfunc explainer:

- Load the pyfunc explainer as an MLflow pyfunc model to explain a given pandas DataFrame input.
- Load the pyfunc explainer as a PySpark UDF to explain a given PySpark DataFrame input.

Let's start with loading the explainer as an MLflow pyfunc model.

### Loading the MLflow pyfunc explainer as an MLflow pyfunc model

As we have already mentioned, another way to consume an MLflow logged explainer is to load the explainer in a local Python code using MLflow's pyfunc `load_model` method directly, instead of deploying it into a web service. This is very straightforward, and we will show you how it can be done. You can check out the code in the `shap_mlflow_pyfunc_explainer.py` file in the GitHub repository ([https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter10/notebooks/shap\\_mlflow\\_pyfunc\\_explainer.py](https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter10/notebooks/shap_mlflow_pyfunc_explainer.py)):

1. The first step is to load the logged explainer back into memory. The following code does this using `mlflow.pyfunc.load_model` and the explainer `run_id` URI:

```
run_id = "ad1edb09e5ea4d8ca0332b8bc2f5f6c9"
logged_explainer = f'runs://{run_id}/nlp_sentiment_classifier_explainer'
explainer = mlflow.pyfunc.load_model(logged_explainer)
```

This should load the explainer as if it is just a generic MLflow pyfunc model. We can print out the metadata of the explainer by running the following code:

```
explainer
```

This will show the following output:

```
mlflow.pyfunc.loaded_model: artifact_path: nlp_sentiment_classifier_explainer flavor: mlflow.pyfunc.model run_id: ad1edb09e5ea4d8ca0332b8bc2f5f6c9
```

This means this is a `mlflow.pyfunc.model` flavor, which is great news, since we can use the same MLflow pyfunc API to use this explainer.

- Next, we will get some example data to test the newly loaded explainer:

```
import datasets
dataset = datasets.load_dataset("imdb", split="test")
short_data = [v[:500] for v in dataset["text"][:20]]
df_test = pd.DataFrame(short_data, columns = ['text'])
```

This will load the IMDb test dataset, truncate each review text to 500 characters, and pick the first 20 rows to make a pandas DataFrame for explanation in the next step.

- Now, we can run the explainer as follows:

```
results = explainer.predict(df_test)
```

This will run the SHAP partition explainer for the input DataFrame `df_test`. It will show the following output for each row of the DataFrame when it is running:

```
Partition explainer: 2it [00:38, 38.67s/it]
```

The result will be a pandas DataFrame with a single column, `shap_values`. This may take a few minutes as it needs to tokenize each row, execute the explainer, and serialize the output.

- Once the explainer execution is done, we can check the results by deserializing the row content. Here is the code to check the first output:

```
results_deserialized = pickle.loads(results['shap_
values'][0])
print(results_deserialized)
```

This will print out the first row's `shap_values`. *Figure 10.6* shows a partial screenshot of the output of `shap_values`:

```
.values =
array([[[-0.01545583,  0.01545583],
        [-0.01545583,  0.01545583],
        [-0.01545583,  0.01545583],
        [-0.01545583,  0.01545583],
        [-0.0053423 ,  0.0053423 ],
        [-0.0053423 ,  0.0053423 ]],
```

Figure 10.6 – Partial output of the deserialized `shap_values` from the explanation

As we can see in *Figure 10.6*, the output of `shap_values` is no different from what we learned in *Chapter 9, Fundamentals of Deep Learning Explainability*, when we did not use MLflow to log and load the explainer. We can also generate Shapley text plots to highlight the contribution of the texts to the predicted sentiment.

5. Run the following statement in the notebook to see the Shapely text plot:

```
shap.plots.text(results_deserialized[:, :, "POSITIVE"])
```

This will generate a plot displayed in *Figure 10.7*:

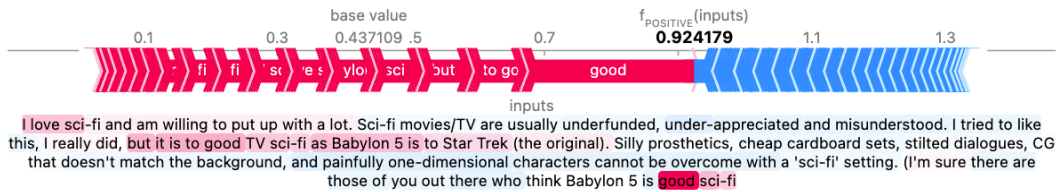


Figure 10.7 – Shapley text plot using deserialized `shap_values` from our MLflow logged explainer

As can be seen in *Figure 10.7*, this review has a positive sentiment and the keywords or phrases that contribute to the predicted sentiment are `good`, `love`, and some other phrases highlighted in red. When you see this Shapley text plot, you should give yourself a round of applause, as you have finished learning how to use an MLflow logged explainer to generate batch explanation.

As mentioned during the step-by-step implementation of this batch explanation, it is a little slow to do a large batch explanation using this pyfunc model approach. Luckily, we have another way to implement the batch explanation using the PySpark UDF function, which we will explain in the next subsection.

## Loading the pyfunc explainer as a PySpark UDF

For scalable batch explanation, we can use Spark's distributed computing capability, which is supported by loading the pyfunc explainer as a PySpark UDF. There is no extra work to use this capability, since this is provided by the MLflow pyfunc API already through the `mlflow.pyfunc.spark_udf` method. We will show you how to implement this at-scale explanation step by step:

1. First, make sure you have worked through the `README.md` file (<https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLflow/blob/main/chapter10/README.md>) to install Spark, create and activate the `chapter10-dl-pyspark-explain` virtual environment, and set up all the environment variables before you run the PySpark UDF code to do the explanation at scale.

- Then you can start running the VS Code notebook, `shap_mlflow_pyspark_explainer.py`, which you can check out in the GitHub repository: [https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter10/notebooks/shap\\_mlflow\\_pyspark\\_explainer.py](https://github.com/PacktPublishing/Practical-Deep-Learning-at-Scale-with-MLFlow/blob/main/chapter10/notebooks/shap_mlflow_pyspark_explainer.py). Run the following command at `chapter10/notebooks/`:

```
python shap_mlflow_pyspark_explainer.py
```

You will get the final output displayed in *Figure 10.8*, among quite a few lines of output preceding these final few lines:

```
-RECORD 0-----
text          | I love sci-fi and am willing to put up with a lot. Sci-fi movies/TV are usual...
shap_values   | b'\x80\x04\x95l4\x00\x00\x00\x00\x00\x00\x8c\x11shap._explanation\x94\x8c\x0b...
-RECORD 1-----
text          | Worth the entertainment value of a rental, especially if you like action movi...
shap_values   | b'\x80\x04\x95\x82-\x00\x00\x00\x00\x00\x00\x8c\x11shap._explanation\x94\x8c...
only showing top 2 rows
```

Figure 10.8 – PySpark UDF explainer's output of the first two rows of text's `shap_values` along with their input texts

As can be seen in *Figure 10.8*, the PySpark UDF explainer's output is a PySpark DataFrame that has two columns: `text` and `shap_values`. The `text` column is the original input text, while the `shap_values` column contains the pickled serialized Shapley values, just like we saw in the previous subsection when we used the `pyfunc` explainer for the pandas DataFrame.

Now let's see what's happening in the code. We will explain the key code blocks in the `shap_mlflow_pyspark_explainer.py` file. Since this is a VS Code notebook, you can run it either in the command line as we just did or interactively inside the VS Code IDE window.

- The first key code block is to load the explainer using the `mlflow.pyfunc.spark_udf` method, as follows:

```
spark = SparkSession.builder.appName("Batch explanation
with MLflow DL explainer").getOrCreate()
run_id = "ad1edb09e5ea4d8ca0332b8bc2f5f6c9"
logged_explainer = f'runs:/{run_id}/nlp_sentiment_
classifier_explainer'
explainer = mlflow.pyfunc.spark_udf(spark, model_
uri=logged_explainer, result_type=StringType())
```



The first statement is to initialize a `SparkSession` variable and then use `run_id` to load the logged explainer into memory. Run the explainer to get the metadata as follows:

```
explainer
```

We will get the following result:

```
<function mlflow.pyfunc.spark_udf.<locals>.udf(iterator:
Iterator[Tuple[Union[pandas.core.series.Series, pandas.
core.frame.DataFrame], ...]]) -> Iterator[pandas.core.
series.Series]>
```

This means we now have a SHAP explainer wrapped as a Spark UDF function. This allows us to directly apply the SHAP explainer for an input PySpark DataFrame in the next step.

4. We load the IMDb test dataset as before to get a list of `short_data`, and then create a PySpark DataFrame for the top 20 rows of the test dataset for explanation:

```
df_pandas = pd.DataFrame(short_data, columns = ['text'])
spark_df = spark.createDataFrame(df_pandas)
spark_df = spark_df.withColumn('shap_values',
explainer())
```

Note the last statement, which uses PySpark's `withColumn` function to add a new `shap_values` column to the input DataFrame, `spark_df`, which originally contained only one column, `text`. This is a natural way to use Spark's parallel and distributed computing capability. If you have run both the previous non-Spark approach using the MLflow `pyfunc load_model` method and the current PySpark UDF one, you will notice that the Spark approach runs much faster, even on a local computer. This allows us to do SHAP explanation at scale for many instances of input texts.

5. Finally, to verify the results, we show the `spark_df` DataFrame's top two rows, which was illustrated in *Figure 10.8*.

By now, with MLflow's `pyfunc` Spark UDF wrapped SHAP explainer, we can confidently do large-scale batch explanation. Congratulations!

Let's now summarize what we have learned in this chapter in the next section.

## Summary

In this chapter, we first reviewed the existing approaches in the MLflow APIs that could be used for implementing explainability. Two existing MLflow APIs, `mlflow.shap` and `mlflow.evaluate`, have limitations, thus cannot be used for the complex DL models and pipelines explainability scenarios we need. We then focused on two main approaches to implement SHAP explanations and explainers within the MLflow API framework: `mlflow.log_artifact` for logging explanations and `mlflow.pyfunc.PythonModel` for logging a SHAP explainer. Using the `log_artifact` API can allow us to log Shapley values and explanation plots into the MLflow tracking server. Using `mlflow.pyfunc.PythonModel` allows us to log a SHAP explainer as a MLflow `pyfunc` model, thus opening doors to deploy a SHAP explainer as a web service to create an EaaS endpoint. It also opens doors to use SHAP explainers through the MLflow `pyfunc.load_model` or `spark_udf` API for large-scale offline batch explanation. This enables us to confidently implement explainability at scale for DL models.

As the field of explainability continues to evolve, MLflow's integration with SHAP and other explainability toolboxes will also continue to improve. Interested readers are encouraged to continue their learning journey through the links provided in the further reading section. Happy continuous learning and growing!

## Further reading

- Shapley Values at Scale: <https://neowaylabs.github.io/data-science/shapley-values-at-scale/>
- Scaling SHAP Calculations With PySpark and Pandas UDF: <https://databricks.com/blog/2022/02/02/scaling-shap-calculations-with-pyspark-and-pandas-udf.html>
- Speeding up Shapley value computation using Ray, a distributed computing system: <https://www.telesens.co/2020/10/05/speeding-up-shapley-value-computation-using-ray-a-distributed-computing-system/>
- Interpreting an NLP model with LIME and SHAP: [https://medium.com/@kalia\\_65609/interpreting-an-nlp-model-with-lime-and-shap-834ccfa124e4](https://medium.com/@kalia_65609/interpreting-an-nlp-model-with-lime-and-shap-834ccfa124e4)
- Model Evaluation in MLflow: <https://databricks.com/blog/2022/04/19/model-evaluation-in-mlflow.html>



# Index

## A

- access tokens
  - for Databricks server access 110
- ante-hoc explainability 217
- Asynchronous Successive Halving
  - Algorithm (ASHA) 130
- AugLy
  - reference link 102
- auto-logging
  - limitations 61
- autologging, MLflow 31
- AWS SageMaker
  - about 193
  - deploying to 193-207

## B

- B2C2 v Quoine
  - reference link 215
- backends
  - MLflow Project, running on 122
- basic DL sentiment classifier
  - implementing 7-9
- batch explanation
  - MLflow pyfunc explainer, using 248

- batch inference
  - about 156
  - Deep Learning (DL) inference
    - pipeline, deploying for 185-187
- bert-base-multilingual-uncased (BERT) 175
- BERT-like pretrained model, Hugging
  - Face model repository
    - reference link 8
- binary 65
- bring your own model (BYOM) 193
- built-in model flavors, MLflow
  - reference link 43

## C

- caching postprocessing logic
  - implementing 171
- caching preprocessing logic
  - implementing 171
- Captum
  - about 226
  - reference link 226
- CircleCI 13
- classical ML 6

- click Python package
  - reference link 82
- cloud
  - DL pipelines, running remotely
    - with local code 109-118
  - DL pipelines, running remotely with
    - remote code in GitHub 118-122
- clusters
  - reference link 112
- cluster-scoped init scripts
  - reference link 104
- code, submitting to run in
  - Databricks server
    - prerequisites 109, 110
- code tracking, MLflow 43, 44
- Compact Language Detector v3
  - reference link 160
- Concourse 13
- configurable mechanisms,
  - MLproject framework
    - entry points 103
    - hardware dependencies 104
    - software and library dependencies 104
- Continuous Integration and Continuous Deployment (CI/CD) 12, 109
- Continuous Labeling 13
- Continuous Testing 13
- Continuous Training 13
- custom MLflow Python model
  - implementing 162-168

## D

- data
  - tracking, example with MLflow 93-95
- data and pipeline configurations
  - reference link 127

- Databand
  - URL 13
- Databricks
  - cluster specification 112
  - Delta table 92
  - GitHub Token, for Databricks to access
    - enterprise project repository 119
  - reference link 119
- Databricks CLI 110
- Databricks command-line tool
  - used, for generating
    - .databrickscfg file 110
- Databricks Community Edition
  - reference link 92
- Databricks feature
  - reference link 184
- Databricks File System (DBFS) 92, 116
- Databricks Runtime 9.1 LTS,
  - for machine learning
    - reference link 112
- data-centric AI competition
  - reference link 91
- Datadog 13
- DataFold
  - URL 13
- Dataiku 20
- data versioning
  - tracking, in Delta Lake 91, 92
- Deep Learning (DL)
  - about 3, 6
  - code challenges 18, 19
  - data challenges 15, 16
  - explainability challenges 19-22
  - life cycle development and stages 9-11
  - model challenges 17

- Deep Learning (DL) inference pipeline
    - caching postprocessing
      - and preprocessing logic, implementing 171
    - deploying, for batch inference 185-187
    - deploying, to web service 188, 189
    - deployment tools 184, 185
    - deployment tools, scenarios 183
    - language detection preprocessing
      - logic, implementing 169, 170
    - patterns 156-158
    - response composition postprocessing
      - logic, implementing 172-175
  - Defense Advanced Research Projects Agency (DARPA) 215
  - Delta Lake
    - data versioning, tracking 91, 92
    - reference link 92
    - URL 13
  - Delta table
    - in Databricks 92
  - Directed Acyclic Graph (DAG) 157
  - Disparate Impact 20
  - DL experiment
    - implementing, with MLflow
      - autologging 30-35
  - DL model
    - building 7-9
    - Ray Tune trainable, creating 137-142
  - DL pipelines
    - automatic HPO 127
    - running, locally with local code 104-107
    - running, locally with remote code in GitHub 107-109
    - running, remotely in cloud
      - with local code 109-118
    - running, remotely in cloud with remote code in GitHub 118-122
      - running, remotely in remote Databricks server 111-118
  - docker-compose tool
    - reference link 51
  - Docker Desktop
    - reference link 51
  - Docker error 197
  - DVC
    - URL 91
- ## E
- Elastic Container Registry (ECR) 199
  - Enterprise Databricks server
    - reference link 109
  - entry points 103
  - execution environments 101
  - execution scenarios
    - about 101
    - examples 102
  - experiments 35, 36
  - explainability
    - audience 215-221
    - categories 215-221
  - explainable artificial intelligence (XAI) 215
  - Explanation as a Service (EaaS)
    - about 233
    - MLflow pyfunc deploying for 246, 247
  - extraction/transformation/
    - loading (ETL) 18
- ## F
- fastai 61
  - FastText
    - URL 160
  - feature attribution 217

- Feature Importance Shift 22
- field-programmable gate arrays (FPGAs) 102
- foundation model 5
- full-fledged local MLflow tracking server
  - artifact store 51
  - backend store 51
  - setting up 51-53

## G

- GitHub
  - DL pipelines, running locally
    - with remote code 107-109
  - DL pipelines, running remotely in cloud with remote code 118-122
- Google Cloud Platform (GCP)
  - reference link 193
- Grafana 13
- Graphics Process Units (GPUs) 6

## H

- hadoop-yarn
  - reference link 122
- HPO-ready DL models
  - creating, with MLflow 134
  - creating, with Ray Tune 134
- HPO, with Ray Tune
  - running, HyperBand used 147-150
  - running, Optuna used 147-150
- Hugging Face 61
- Human-Computer Interface/
  - Interaction (HCI) 215
- HyperBand
  - used, for running HPO with Ray Tune 147-150

- hyperparameter optimization (HPO)
  - callback integration, with MLflow 131, 132
  - DL frameworks, support 134
  - GPU clusters, scalability and support 132
  - integrating, with cutting-edge HPO algorithms (CS and CE) 133
  - scalability, and support of GPU clusters 132
  - selecting 130
  - working 130
- Hyperparameter Optimization (HPO) 155
- hyperparameters 66
- hyperparameters types
  - data and pipeline configurations 128
  - DL model type and architecture 127
  - learning- and training-related parameters 128

## I

- ImageNet
  - URL 15
- inference pipeline
  - implementing, as new entry point in MLproject 175-178
- Internet Movie Database (IMDb)
  - building 8

## K

- Keras 61
- KernelExplainer 232
- Kubernetes (K8s) cluster 12, 104

**L**

Lakehouse  
  about 92  
  reference link 92  
language detection preprocessing logic  
  implementing 169, 170  
Language Interpretability Tool (LIT) 218  
local code  
  used, for running DL pipelines  
    locally 104-107  
  used, for running DL pipelines  
    remotely in cloud 109-118  
local GitHub code  
  versus remote GitHub code 44  
Local Interpretable Model-Agnostic  
  Explanations (LIME) 220  
long-term support (LTS) version 112

**M**

Machine Learning Operations (MLOps)  
  about 4  
  challenges 12  
  foundation layers 12  
  pillars 12  
micro-average method 65  
miniconda 7  
MinIO  
  URL 51  
MLflow  
  about 26  
  code tracking 43, 44  
  components 35  
  data, tracking example with 93-95  
  MLOps layers, building 12-14  
  model registry, versus model logging 43  
  models 39-43

  Ray Tune HPO experiment,  
    running with 145-147  
  reference link 26  
  setting up 26, 136, 137  
  setting up, locally with miniconda 27, 28  
  setting up, to interact with remote  
    MLflow server 29, 30  
  tracking server 39  
  usage patterns 35  
  used, for creating HPO-ready  
    DL models 134, 135  
MLflow, APIs for model provenance  
  logging API 55  
  registry API 55  
MLflow artifact logging API  
  used, for implementing SHAP  
    explanation 235-241  
MLflow autologging  
  about 31  
  DL experiment, implementing 30-35  
MLflow Custom Model  
  flavor, creating 168  
MLflow deployment plugins  
  deploying, with Ray Serve 190-192  
  references 184  
  using, for deployment 193  
mlflow.evaluate API  
  reference link 234  
MLflow experiment  
  exploring 37, 38  
MLflow Experiment Name/ID  
  for MLflow project run 106  
MLflow explainability integration  
  about 232-234  
  artifacts 232  
mlflow.log\_artifact API  
  reference link 233  
mlflow.log\_param API 66



- mlflow.log\_params API 66
  - MLflow Model Python Function
    - API 159-162
  - MLflow model tracking
    - implementing 55- 62
  - MLflow Project
    - running, on other backends 122
  - MLflow Project, running with
    - GitHub's Main Branch
      - hidden bug 108
  - MLflow pyfunc API
    - used, for implementing
      - SHAP explainer 241
  - MLflow pyfunc explainer
    - creating 242-246
    - deploying, for EaaS 246, 247
    - loading, as MLflow pyfunc
      - model 248-250
    - logging 242-246
    - using, for batch explanation 248
  - mlflow.pyfunc.load\_model
    - versus mlflow.pytorch.load\_model 59
  - MLflow pyfunc model
    - MLflow pyfunc explainer,
      - loading as 248-250
  - mlflow.pyfunc.PythonModel API
    - reference link 233
  - mlflow.run tool
    - reference link 82
  - MLflow run Python API
    - reference link 115
  - mlflow.shap API
    - reference link 233
  - mlflow.shap.log\_explainer API
    - issues 246
  - mlflow.tracking
    - reference link 85
  - MLflow version
    - reference link 233
  - MLmodel 159
  - MLproject
    - about 44
    - asynchronous mode 115
    - synchronous mode 115
    - entry point, executing, ways 82
    - inference pipeline, implementing as
      - new entry point in 175-178
  - Model as a Service (MaaS) 183
  - model hyperparameters 66
  - Model Logging
    - versus Model Registry 56
  - model metrics
    - tracking 63-65
  - model parameters
    - about 66
    - tracking 66-68
  - model provenance
    - tracking 53
  - Model Registry
    - versus Model Logging 56
  - Model Signature
    - reference link 161
- ## N
- Natural Language Processing (NLP) 104
- ## O
- online inference 157
  - Open Neural Network Exchange (ONNX)
    - about 184
    - reference link 17, 185

Open Provenance Model (OPM)  
  about 54  
  Vocabulary Specification,  
    reference link 54  
open provenance tracking  
  framework 54, 55  
Optuna  
  used, for running HPO with  
    Ray Tune 147-150  
Outreach  
  URL 15

## P

Parquet  
  about 92  
  reference link 92  
PartitionExplainer 232  
patterns  
  of DL inference pipelines 156-158  
  types 157  
pickle  
  reference link 17  
pipeline  
  tracking 79-87  
pipeline chaining 83  
Population-Based Training (PBT)  
  reference link 129  
post-hoc explainability 217  
pyfunc explainer  
  loading, as PySpark UDF 250-252  
pyfunc model  
  reference link 161  
PyPI  
  URL 89  
PySpark 92  
PySpark UDF  
  pyfunc explainer, loading as 250-252

Python libraries  
  tracking 88-90  
PyTorch lightning framework 61  
PyTorch Mobile  
  reference link 183  
PyTorch model 159  
PyTorch optimizers  
  reference link 128

## R

Ray Serve  
  about 190  
  advantages 190  
  reference link 184, 190  
  used, for deploying MLflow  
    deployment plugins 190-192  
Ray Tune  
  function-based APIs and  
    class-based APIs 135  
  scheduler 135  
  search space 135  
  setting up 136  
  suggest 135  
  trials 135  
  used, for creating HPO-ready  
    DL models 134, 135  
Ray Tune HPO experiment  
  running, with MLflow 145-147  
Ray Tune HPO run function  
  creating 142-144  
Ray Tune trainable  
  creating, for DL model 137-142  
remote code  
  used, for running DL pipelines locally  
    in GitHub cloud 118-122  
  used, for running DL pipelines  
    locally in GitHub 107-109

- remote GitHub code
  - versus local GitHub code 44
- response composition postprocessing logic
  - implementing 172-175

## S

- scikit-learn 159
- Seldon MLServer
  - reference link 184
- self-supervised learning 5
- SHAP Explainability toolbox
  - exploring 221-225
- SHAP explainer
  - implementing, with MLflow
    - pyfunc API 241
  - runtime errors, avoiding 243
- SHAP explanation
  - implementing, with MLflow
    - artifact logging API 235-241
- SHapley Additive exPlanations (SHAP) 217
- Software 2.0
  - reference link 12
- stochastic gradient descent (SGD) 128

## T

- target execution environment
  - (local and remote) 101
- TensorFlow 61
- TensorFlow Lite
  - reference link 183
- Tensor Processing Units (TPUs) 102
- Terraform 12
- Time Travel 92
- TinyML 183

- torchmetrics
  - reference link 63
- TorchScript
  - reference link 17, 185
- transfer learning 5
- Transformers Interpret toolbox
  - exploring 226-228
  - reference link 226
- TreeExplainer 232
- Tree-Structured Parzen Estimators (TPE) 133
- Triton
  - reference link 184
- Triton Inference Server
  - reference link 184
- typical core DL development paradigm
  - example 6

## U

- user defined function (UDF) 183
- User Experience (UX) 216

## V

- VGG-NETS
  - reference link 15
- VS Code 162

## W

- web service
  - Deep Learning (DL) inference
    - pipeline, deploying to 188, 189
- WordNet database
  - URL 15



Packt . com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

## Why subscribe?

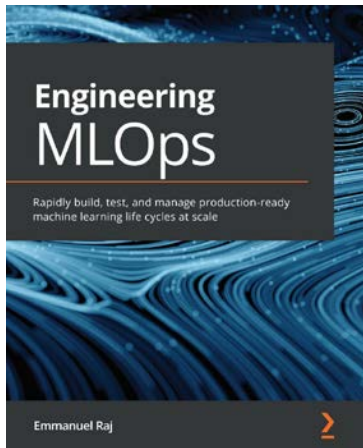
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [packt . com](http://packt.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [customercare@packtpub . com](mailto:customercare@packtpub.com) for more details.

At [www . packt . com](http://www.packt.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

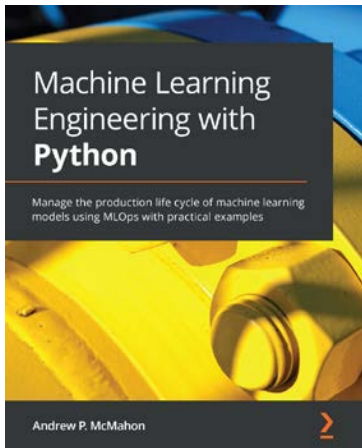


## **Engineering MLOps**

Emmanuel Raj

ISBN: 9781800562882

- Formulate data governance strategies and pipelines for ML training and deployment
- Get to grips with implementing ML pipelines, CI/CD pipelines, and ML monitoring pipelines
- Design a robust and scalable microservice and API for test and production environments
- Curate your custom CD processes for related use cases and organizations
- Monitor ML models, including monitoring data drift, model drift, and application performance
- Build and maintain automated ML systems



## **Machine Learning Engineering with Python**

Andrew McMahon

ISBN: 9781801079259

- Find out what an effective ML engineering process looks like
- Uncover options for automating training and deployment and learn how to use them
- Discover how to build your own wrapper libraries for encapsulating your data science and machine learning logic and solutions
- Understand what aspects of software engineering you can bring to machine learning
- Gain insights into adapting software engineering for machine learning using appropriate cloud technologies
- Perform hyperparameter tuning in a relatively automated way

## Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](https://authors.packtpub.com) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

## Share Your Thoughts

Now you've finished *Practical Deep Learning at Scale with MLflow*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

