JAX IN ACTION

Grigory Sapunov

MEAP

MANNING

MEAP Edition
Manning Early Access Program

JAX in Action

Version 3

welcome

Thank you for purchasing the MEAP for *JAX in Action*!

JAX is a Python mathematics library with a NumPy interface developed by Google. It is heavily used for machine learning research, and it seems that JAX has already become the #3 deep learning framework (after TensorFlow and PyTorch). It also became the main deep learning framework in companies such as DeepMind, and more and more of Google's own research use JAX.

JAX promotes a functional programming paradigm in deep learning. It has powerful function transformations such as taking gradients of a function, JIT-compilation with XLA, auto-vectorization, and parallelization. JAX supports GPU and TPU and provides great performance.
JAX gives you a strong foundation for building your neural networks, but the real power comes from its constantly growing ecosystem. There are many machine learning-related libraries, including high-level deep learning libraries Flax (by Google) and Haiku (by DeepMind), a gradient processing and optimization library called Optax, libraries for graph neural networks, reinforcement learning, evolutionary computations, federated learning, and so on. Together with its ecosystem, JAX provides an exciting alternative to the two current state-of-the-art deep learning frameworks — PyTorch and TensorFlow.

We are going to cover all these topics in the book.

But JAX is not limited to deep learning. There are many exciting applications and libraries on top of JAX for physics, including molecular dynamics, fluid dynamics, rigid body simulation, quantum computing, astrophysics, ocean modeling, and so on. There are libraries for distributed matrix factorization, streaming data processing, protein folding, and chemical modeling, with other new applications emerging constantly.

JAX is gaining momentum right now, and its ecosystem is constantly growing. It's the right time to start learning it!

At the moment, there are almost no comprehensive resources for those who want to start learning and using JAX. There is good documentation (and a really helpful community!) on the JAX's site, but it's still hard to get the whole picture, especially if you are going to use some libraries together with JAX. And you probably want to use them, as the ecosystem is the JAX's strength.

I write this book for such people. I strive to aggregate all the important things in a single place and to create a learning path that will help you sequentially understand the JAX concepts, build up your skills in using JAX and its ecosystem, and finally help you start applying JAX for your own projects and research.

I expect you have a general knowledge of deep learning and a reasonable Python proficiency. I am not going to teach you both JAX _and_ deep learning simultaneously, as there are many great books on deep learning and its particular aspects already. I will focus on JAX only. However, I'll try to provide short explanations of important deep learning concepts as we meet them along the way. It will be helpful for people who come from other than deep learning fields, say, physics.

I believe JAX is much more than a deep learning framework, and the growing amount of JAX modules not related to deep learning confirms it. JAX can potentially be a great tool for differentiable programming, large-scale physics simulations, and many other things, some of which are yet to come. So, I'd like to help these people as well. I'll be happy if my book helps you achieve your goals with JAX.

The book is structured in the following way. The first two chapters will give you a basic introduction to JAX and why you might want to use it. Then a large part of the book is dedicated to the core JAX and is not limited to deep learning. This basic knowledge and skills will be useful to anyone who wants to start using JAX. The second large part will be dedicated to the JAX ecosystem, mostly focused on deep learning modules.

I will be happy to receive your feedback along the way, and my goal will be achieved if you will understand JAX and start using it to implement your great ideas!

Please be sure to post any questions, comments, or suggestions you have about the book in the liveBook discussion forum. Your feedback is essential in developing the best book possible.

— Grigory Sapunov

brief contents

1 Intro to JAX

This chapter covers

- **What is JAX, and how does it compare to NumPy**
- **Why use JAX?**
- **Comparing JAX with TensorFlow/PyTorch**

JAX is gaining popularity as more and more researchers start using it for their research and large companies such as DeepMind contribute to its ecosystem.

In this chapter, we will introduce JAX and its powerful ecosystem. We will explain what JAX is and how it relates to NumPy, PyTorch, and TensorFlow. We will go through JAX's strengths to understand how they combine, giving you a very powerful tool for deep learning research and high-performance computing.

1.1 What is JAX?

JAX is a Python mathematics library with a NumPy interface developed by Google (the Google Brain team, to be specific). It is heavily used for machine learning research, but it is not limited to it, and many other things can be solved with JAX.

JAX creators describe it as Autograd and XLA. Do not be afraid if you are unfamiliar with these names; it’s normal, especially if you are just getting into the field.

Autograd (https://github.com/hips/autograd) is the library that efficiently computes derivatives of NumPy code, the predecessor of JAX. B the way, the Autograd library's main developers are now working on JAX. In a few words, Autograd means you can automatically calculate gradients for your computations, which is the essence of deep learning and many other fields, including numerical optimization, physics simulations, and, more generally, differentiable programming.

XLA is Google's domain-specific compiler for linear algebra called Accelerated Linear Algebra. It compiles your Python functions with linear algebra operations to high-performance code for running on GPU or TPU.

Let's start with the NumPy part.

1.1.1 JAX as NumPy

NumPy is a workhorse of Python numerical computing. It is so widely used in the industry and science that NumPy API became the de facto standard for working with multidimensional arrays in Python. JAX provides a NumPy-compatible API but offers many new features absent in NumPy, so some people call JAX the 'NumPy on Steroids'.

JAX provides a multidimensional array data structure called `DeviceArray` that implements many typical properties and methods of the `numpy.ndarray`. There is also the `jax.numpy` package that implements the NumPy API with many well-known functions like `abs()`, `conv()`, `exp()`, and so on.

JAX tries to follow the NumPy API as closely as possible, and in many cases, you can switch from `numpy` to `jax.numpy` without changing your program.

There are still some limitations, and not all the NumPy code can be used with JAX. JAX promotes a functional programming paradigm and requires pure functions without side effects. As a result, JAX arrays are immutable, yet NumPy programs frequently use in-place updates, like `arr[i] += 10`. JAX has a workaround by providing an alternative purely functional API that replaces in-place updates with a pure indexed update function. For this particular case, it will be `arr = arr.at[i].add(10)`. There are a few other differences, and we will address them in Chapter 3.

So, you can use almost all the power of NumPy and write your programs in the way you are accustomed to when using NumPy. But, you have new opportunities here.

1.1.2 Composable transformations

JAX is much more than NumPy. It provides a set of *composable function transformations* for Python+NumPy code. At its core, JAX is an extensible system for transforming numerical functions with four main transformations (but it doesn't mean that no more transformations are to come!):

1. **Taking the gradient** of your code or differentiating it. It is the essence of deep learning and many other fields, including numerical optimization, physics simulations, and, more generally, differentiable programming. JAX uses an approach called *automatic differentiation* (or *autodiff* for short). Automatic differentiation helps you focus on your code and not take derivatives by hand; the framework takes care of it. It is typically done by the `grad()` function, but other advanced options exist. We will give more context on automatic differentiation and dive deeper into the topic in Chapter 4.

2. **Compiling** your code with `jit()`, or Just-in-Time compilation. JIT uses Google's XLA to compile and produce efficient code for GPUs (typically NVIDIA ones through CUDA, though AMD ROCm platform support is in progress) and TPUs (Google's Tensor Processing Units). XLA is the backend that powers machine learning frameworks, originally TensorFlow, on various devices, including CPUs, GPUs, and TPUs. We will dedicate the whole of Chapter 5 to this topic.
3. **Auto-vectorization** your code with `vmap()`, which is the vectorizing map. If you are familiar with functional programming, you probably know what a map is. If not, do not worry; we will describe in detail what it means later. `vmap()` takes care of batch dimensions of your arrays and can easily convert your code from processing a single item of data to processing many items (called a batch) at once. You may call it auto-batching. By doing this, you vectorize the computation, which typically gives you a significant boost on the modern hardware that can efficiently parallelize matrix computations. We will discuss it in Chapter 6.
4. **Parallelizing** your code to run on multiple accelerators, say, GPUs or TPUs. It is done with `pmap()`, which helps write single-program multiple-data (SPMD) programs. `pmap()` compiles a function with XLA, then replicates it and executes each replica on its XLA device in parallel. This topic will also be discussed in Chapter 6.

Each transformation takes in a function and returns a function. You can mix different transformations however you like as long as you use functionally pure functions. We will talk about it later, but in short, a functionally pure function is a function whose behavior is determined only by input data. It has no internal state, and it should produce no side effects. For those who came from functional programming, it should be a natural thing to do. For others, it's not hard to adopt this way of writing programs, and we will help you with it.

If you respect these constraints, you can mix, chain, and nest transformations and create complicated pipelines if needed. JAX makes all those transformations arbitrarily composable. For example, you can prepare a function for processing an image with a neural network, then automatically generate another function that processes a batch of images with `vmap()`, then with `jit()` compile it into efficient code to run on GPU or TPU (or many of them in a parallel fashion with `pmap()`), and finally generate a function to calculate gradients with `grad()` to train our image-processing function with gradient descent. We will see some exciting examples along the way.

These are the transformations you do not want to implement yourself in pure NumPy. You don't have to calculate derivatives by hand anymore, even if you can. The powerful framework takes care of it, no matter how complicated your functions are—the same with auto-vectorization and parallelization.

Figure 1.1 visualizes that NumPy is an engine for working with multidimensional arrays with many useful mathematical functions. JAX has a NumPy-compatible API with its multidimensional arrays and many functions. But besides this, NumPy-like API JAX provides a set of powerful function transformations.

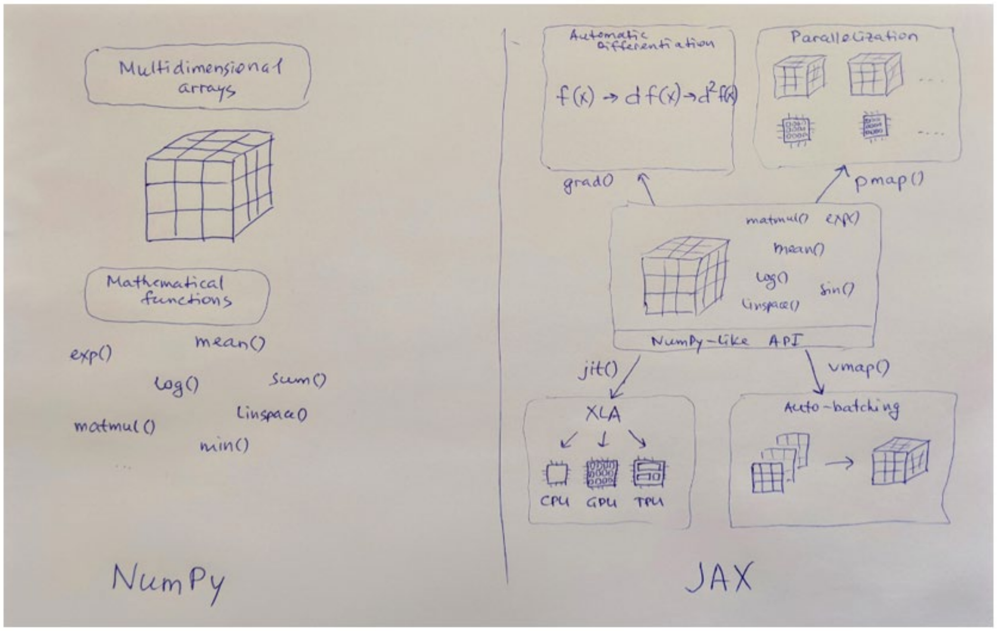


Figure 1.1 JAX vs. NumPy comparison

In some sense, JAX resembles Julia. Julia also has Just-in-Time compilation (JIT), good automatic differentiation capabilities and machine learning libraries, and rich hardware acceleration and parallel computing support. But with JAX, you can stay within your well-known Python world. Sometimes it matters.

1.2 Why use JAX?

JAX is gaining momentum now. The well-known State of AI 2021 report states JAX is a new framework challenger.

Deep learning researchers and practitioners love JAX. More and more new research is being done with JAX. Among the recent research papers, I can mention Vision Transformer (ViT) and MLP-Mixer by Google. Deepmind announced they are using JAX to accelerate their research, and JAX is easy to adopt as both Python and NumPy are widely used and familiar. Its composable function transformations help support machine learning research, and JAX has enabled rapid experimentation with novel algorithms and architectures, and it now underpins many of DeepMind's recent publications. Among them, I'd highlight a new approach to self-supervised Learning called BYOL ("Bootstrap your own latent"), a general transformer-based architecture for structured inputs and outputs called Perceiver IO, and research on large language models with 280-billion parameters Gopher and 70-billion parameters Chinchilla.

In the middle of 2021, Huggingface made JAX/Flax the 3rd officially supported framework in their well-known Transformers library. The Huggingface collection of pretrained models already has twice more JAX models (5,530) than TensorFlow models (2,221) as of April 2022. PyTorch is still ahead of both with 24,467 models, but porting models from PyTorch to JAX/Flax is an ongoing effort.

One of the open-source large GPT-like models called GPT-J-6B by EleutherAI, the 6 billion parameter transformer language model, was trained with JAX on Google Cloud. The authors state it was the right set of tools to develop large-scale models rapidly.

JAX might not be very suitable for production deployment right now, as it primarily focuses on the research side, but that was precisely the way PyTorch went. The gap between the research and production will most likely be closed soon. The Huggingface and GPT-J-6B cases are already moving in the right direction. Given Google's weight and the rapid expansion of the community, I'd expect a bright future for JAX.

JAX is not limited to deep learning. There are many exciting applications and libraries on top of JAX for physics, including molecular dynamics, fluid dynamics, rigid body simulation, quantum computing, astrophysics, ocean modeling, and so on. There are libraries for distributed matrix factorization, streaming data processing, protein folding, and chemical modeling, with other new applications emerging constantly.

Let's look deeper at JAX features you may want to use.

1.2.1 Computational performance

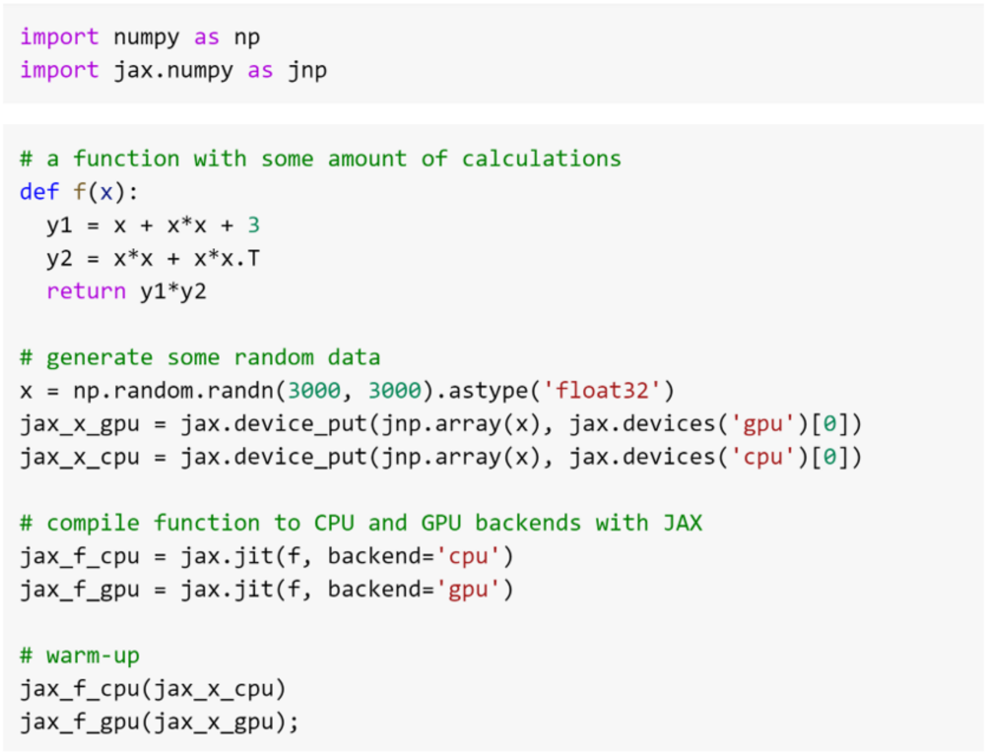
First of all, JAX provides good computational performance. Many things fit into this section, including the ability to use modern hardware such as TPU or GPU, JIT compilation with XLA, automatic vectorization, easy parallelization across the cluster, and the new experimental `xmap()` named-axis programming model being able to scale your program from your laptop CPU to the largest TPU Pod in the cloud. We will discuss all these topics in different chapters of the book.

You can use JAX as accelerated NumPy by replacing '`import numpy as np`' with '`import jax.numpy as np`' at the beginning of your program. This is, in some sense, an alternative to switching from NumPy to CuPy (for using GPUs with NVIDIA CUDA or AMD ROCm platforms), Numba (to have both JIT and GPU support), or even PyTorch if you want hardware acceleration for your linear algebra operations.

Hardware acceleration with GPU or TPU can speed up matrix multiplications and other operations that can benefit from running on this massively parallel hardware. For this type of acceleration, it's enough to perform computations on multidimensional arrays you have put in the accelerator memory. Chapter 3 will show how to manage data placement.

Acceleration can also stem from JIT-compilation with the XLA compiler that optimizes the computation graph and can fuse a sequence of operations into a single efficient computation or eliminate some redundant computations. It improves performance even on CPU, without any other hardware acceleration (yet, CPUs are different as well, and many modern CPUs provide special instructions suitable for deep learning applications).

In Figure 1.2, you can see a screenshot with a simple (and pretty useless) function with some amount of computations that we will evaluate with pure NumPy, with JAX using CPU and GPU (in my case Tesla-P100), and with JAX-compiled versions of the same function for CPU and GPU backends.

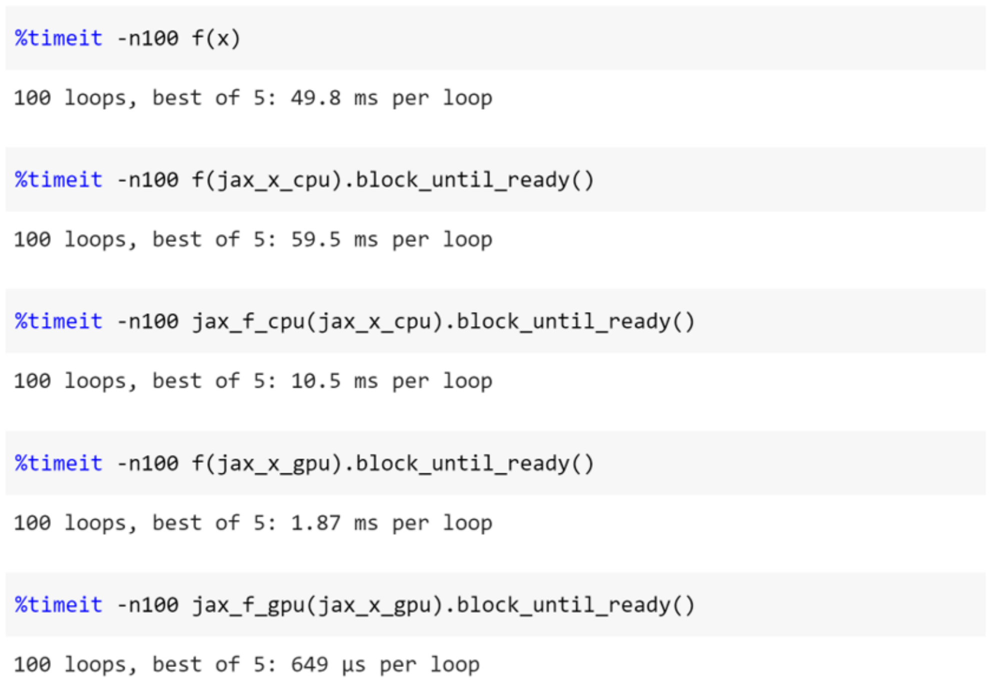


```
import numpy as np
import jax.numpy as jnp
```

```
# a function with some amount of calculations
def f(x):
  y1 = x + x*x + 3
  y2 = x*x + x*x.T
  return y1*y2

# generate some random data
x = np.random.randn(3000, 3000).astype('float32')
jax_x_gpu = jax.device_put(jnp.array(x), jax.devices('gpu')[0])
jax_x_cpu = jax.device_put(jnp.array(x), jax.devices('cpu')[0])

# compile function to CPU and GPU backends with JAX
jax_f_cpu = jax.jit(f, backend='cpu')
jax_f_gpu = jax.jit(f, backend='gpu')

# warm-up
jax_f_cpu(jax_x_cpu)
jax_f_gpu(jax_x_gpu);
```

Figure 1.2 The code for speed comparison between NumPy and different ways of using JAX. We will evaluate the time to calculate f(x).

In Figure 1.3, we compare the different ways of calculating our function. Please, ignore the specific things like `block_until_ready()` or `jax.device_put()` for now; it is needed to wait for the computation completion because JAX uses asynchronous dispatch and to stick an array to the specific device. We will talk about this in Chapter 3.



```
%timeit -n100 f(x)
100 loops, best of 5: 49.8 ms per loop

%timeit -n100 f(jax_x_cpu).block_until_ready()
100 loops, best of 5: 59.5 ms per loop

%timeit -n100 jax_f_cpu(jax_x_cpu).block_until_ready()
100 loops, best of 5: 10.5 ms per loop

%timeit -n100 f(jax_x_gpu).block_until_ready()
100 loops, best of 5: 1.87 ms per loop

%timeit -n100 jax_f_gpu(jax_x_gpu).block_until_ready()
100 loops, best of 5: 649 µs per loop
```

Figure 1.3 Time measurements for different ways of calculating our function f(x). The first line is a pure NumPy implementation, the second line uses JAX on CPU, the third line is a JAX JIT-compiled CPU version, the fourth line uses JAX on GPU, and the fifth line is a JAX JIT-compiled GPU version.

In this particular example, the CPU-compiled JAX version of the function is almost five times faster than the pure NumPy original, yet a non-compiled JAX CPU version is a bit slower. You do not need to compile your function to use GPUs, and we might also ignore all these direct transfers to the GPU device as, by default, all the arrays are created on the first GPU/TPU device if it is available. The non-compiled JAX function that still uses GPU is 5.6 times faster than the JAX CPU-compiled one. And the GPU-compiled version of the same function is another 2.9 times faster than the non-compiled one. The total speedup compared to the original NumPy function is close to 77 times. Decent speed improvements without changing the function code so much.

Auto-vectorization can provide another way to speed up your computations if your hardware resources and program logic allow you to perform computations for many items at once.

Finally, you can also parallelize your code across the cluster and perform large-scale computations in a distributed fashion which is impossible with pure NumPy but can be implemented with something like Dask, DistArray, Legate NumPy, and others.

And, of course, you can benefit from all the mentioned things simultaneously, which is the case for training large distributed neural networks, doing large physical simulations, performing distributed evolutionary computations, and so on.

GPT-J-6B by EleutherAI, the 6 billion parameter transformer language model, is an excellent example of model parallelism with `xmap()` on JAX. The authors state that the project *"required a substantially smaller amount of person-hours than other large-scale model developments did, which demonstrates that JAX + xmap + TPUs is the right set of tools for quick development of large-scale models."*

Some benchmarks also show that JAX is faster than TensorFlow. Others state that *"the performance of JAX is very competitive, both on GPU and CPU. It is consistently among the top implementations on both platforms."* And even the world's fastest transformer of 2020 was built with JAX.

1.2.2 Functional approach

In JAX, everything is in plain sight and explicit. Thanks to the functional approach, there are no hidden variables and side effects, the code is clear, you can change anything you want, and it is much easier to do something off the road. As we've mentioned, researchers love JAX, and much new research is being done with JAX.

Such an approach requires you to change some habits. In Pytorch and TensorFlow, the code is typically organized in classes. Your neural network is a class with all its parameters being an internal state. Your optimizer is another class with its internal state. If you work with reinforcement learning, your environment is usually another class with its state. Your deep learning program looks like a typical object-oriented program with a set of class instances and method calls.

In JAX, code is organized as functions instead. You need to pass any internal state as a function parameter, so all the model parameters (sets of weights in case of neural networks or even seeds for random numbers) are passed into functions directly. Gradients are calculated explicitly by calling a special function (obtained with the `grad()` transformation of your function of interest). Optimizer state and calculated gradients are also parameters of an optimizer function. And so on. No hidden state; everything is visible. Side effects are also bad because they do not work with JIT.

The functional approach also brings rich compositionality. As all the previously mentioned things (automatic differentiation, automatic batching, end-to-end compilation with XLA, parallelization) are implemented as function transformations, it is easy to combine them.

JAX's rich composability and expressivity lead to its powerful ecosystem.

1.2.3 JAX ecosystem

JAX gives you a strong foundation for building your neural networks, but the real power comes from its constantly growing ecosystem. There are many machine learning-related libraries, including high-level deep learning libraries Flax (by Google) and Haiku (by

DeepMind), a gradient processing and optimization library called Optax, libraries for graph neural networks, reinforcement learning, evolutionary computations, federated learning, and so on. Together with its ecosystem, JAX provides an exciting alternative to the two current state-of-the-art deep learning frameworks — PyTorch and Tensorflow.

With JAX, it is easy to compose a solution from different modules. Now you do not have to use an all-in-one framework like TensorFlow or PyTorch with everything included. These are powerful frameworks, but it is sometimes hard to replace one tightly integrated thing with a different one. In JAX, you can build a custom solution by combining the blocks you want.

Long before JAX, the deep learning field was a LEGO-style thing, where you had a set of blocks of different colors for creating your solutions. It is such a LEGO-style thing on different levels: from the lower levels of activation functions and types of layers to high-level engineering blocks of architecture primitives (like self-attention), optimizers, tokenizers, and so on.

With JAX, you have even more freedom to combine different blocks that suit your needs best. You can take a high-level neural network library you want (among several good ones, say Haiku or Flax), a separate module that implements optimizers you wish to use (Optax), customize the optimizer to have custom learning rates for different layers, use PyTorch data loaders you love, a reinforcement learning library (RLax), add Monte-Carlo Tree search (Mctx), and maybe use some other things for meta-learning optimizers from a separate library.

It heavily resembles the Unix way of simple and modular design, as Douglas McIlroy stated: *"Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams because that is a universal interface."*

Well, except for the text streams, this is relevant to JAX philosophy. However, text communication might also become the universal interface between different (large) neural networks one day in the future. Who knows.

And here emerges the ecosystem.

The ecosystem is already huge, and that's just the beginning. There are excellent modules for high-level neural network programming (Flax, Haiku, and others), a module with state-of-the-art optimizers (Optax), a graph neural network library (Jraph), a library for Molecular Dynamics (JAX, M.D.), and so on.

JAX ecosystem contains hundreds of modules already, and new libraries emerge constantly. Among recent ones I noticed are the EvoJAX and Evosax for evolutionary computations, FedJAX for federated learning, Mesh Transformer JAX for model parallel transformers used when training GPT-J-6B, and Scenic library for computer vision research. But it is far from the whole picture.

Deepmind already developed a set of libraries on top of JAX (some of them already mentioned above). There are also fresh libraries for Monte Carlo Tree Search, neural network verification, image processing, and many other things.

The number of JAX models in the Huggingface repository grows constantly, and we mentioned that JAX is the #2 framework according to these numbers.

We will dedicate the last half of the book to the JAX ecosystem.

So, JAX is gaining more and more momentum, and its ecosystem is constantly growing. It's an excellent time to jump in!

1.3 How is JAX different from TensorFlow/PyTorch?

We already discussed how does JAX compare to NumPy. Let's compare JAX with the two modern deep learning frameworks, PyTorch and TensorFlow.

We mentioned that JAX promotes the functional approach compared to the object-oriented approach common to PyTorch and TensorFlow. It is the first very tangible thing you face when you start programming with JAX. It changes how you structure your code and require some changing of habits. At the same time, it gives you powerful function transformations, forces you to write clean code, and brings rich compositionality.

JAX-like composable function transforms for PyTorch.

JAX composable function transformations heavily influenced PyTorch, so in March 2022, with the PyTorch 1.11 release, its developers announced a beta release of functorch (https://github.com/pytorch/functorch) library, JAX-like composable function transforms for PyTorch. The reason for doing this is that many use cases are tricky to do in PyTorch today, such as computing per-sample-gradients, running ensembles of models on a single machine, efficiently batching together tasks in the meta-learning inner-loops, efficiently computing Jacobians and Hessians and their batched versions. The list of supported transforms is still smaller than the JAX's ones.

Another tangible thing you soon notice is that JAX is pretty minimalistic. It does not implement everything. TensorFlow and PyTorch are the two most popular and well-developed deep learning frameworks with almost all possible batteries included. Compared to them, JAX is a very minimalistic framework, and it's even hard to name it a framework. It's rather a library.

For example, JAX does not provide any data loaders just because other libraries (e.g., PyTorch or Tensorflow) do this well. JAX authors do not want to reimplement everything; they want to focus on the core. And that's precisely the case where you can and should mix JAX and other deep learning frameworks. It is OK to take the data loading stuff from, say, PyTorch and use it. PyTorch has excellent data loaders, so let each library use its strengths.

Another noticeable thing is that JAX primitives are pretty low-level, and writing large neural networks in terms of matrix multiplications could be time-consuming. Hence, you need a higher-level language to specify such models. JAX does not provide such high-level APIs outside of the box (as did TensorFlow 1 before the high-level Keras API was added to the TensorFlow 2). There are no such batteries included, but it is not a problem as there are high-level libraries for the JAX ecosystem as well.

You don’t need to write your neural networks with the NumPy-like primitives. There are excellent neural network libraries like Flax by Google and Haiku by DeepMind. There is an Optax library with a collection of state-of-the-art optimizers. There are other libraries, and we will dedicate the book's second half to the powerful JAX ecosystem.

Figure 1.4 visualizes the difference between PyTorch/TensorFlow and JAX.

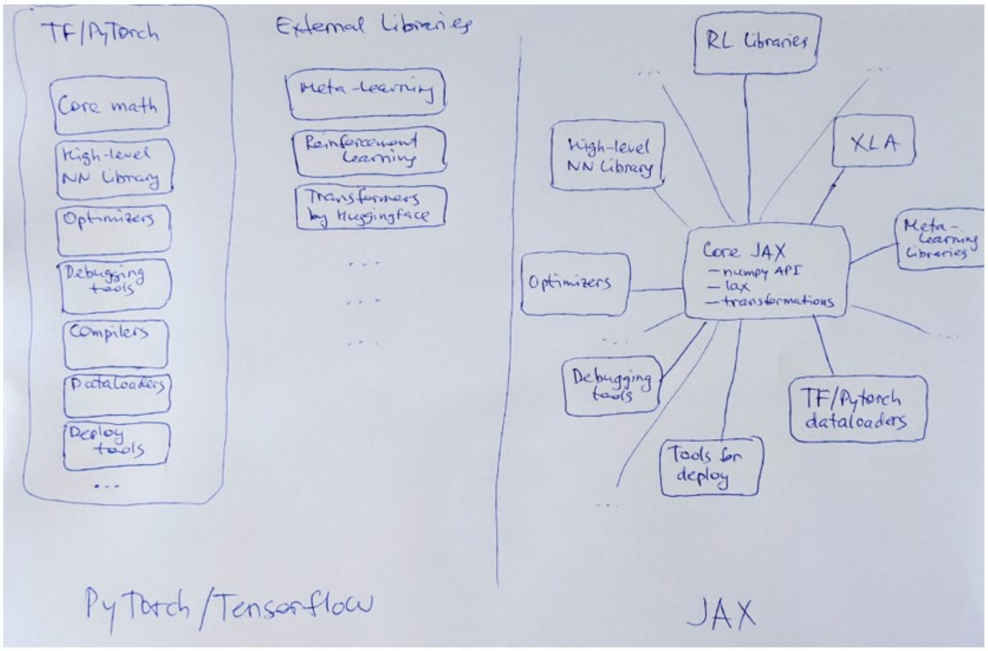


Figure 1.4 JAX vs. PyTorch/TensorFlow comparison

Because JAX is an extensible system for composable function transformations, it is easy to build separate modules for everything and mix them in any way you want.

1.4 Summary

JAX is a low-level Python library from Google that is heavily used for machine learning research (but can also be used in other fields like physics simulations and numerical optimization)

JAX provides a NumPy-compatible API for its multidimensional arrays and mathematical functions

JAX has a powerful set of function transformations, including automatic differentiation (autodiff), jit-compilation, auto-vectorization, and parallelization, which are arbitrarily composable

JAX provides good computational performance thanks to its ability to use modern hardware such as TPU or GPU, JIT compilation with XLA, automatic vectorization, and easy parallelization across the cluster

JAX uses a functional programming paradigm and requires pure functions without side effects

JAX has a constantly growing ecosystem of modules, and you have much freedom to combine different blocks that suit your needs in the best way

Compared to TensorFlow and PyTorch, JAX is a pretty minimalistic framework, but thanks to its constantly growing ecosystem, there are many good libraries to fit your particular needs

2 Your first program in JAX

This chapter covers

- **The MNIST handwritten digit classification problem**
- **Loading a dataset in JAX**
- **Creating a simple neural network in JAX**
- **Auto-vectorizing code with vmap() function**
- **Calculating gradients with grad() function**
- **Jist-in-Time compilation with jit() function**
- **Pure and impure functions**
- **The high-level structure of a JAX deep learning project**

In the previous chapter, we learned about JAX and its importance. We also described the JAX features that make it so powerful. This chapter will give you a practical understanding of JAX.

JAX is a library for composable transformations of Python and NumPy programs, and it is technically not limited to deep learning research. However, JAX is still considered a deep learning framework, sometimes the third after PyTorch and TensorFlow. Therefore many people start learning JAX for deep learning applications. So, a simple neural network application that shows the JAX approach to the problem is very valuable for many.

This chapter is a deep learning "hello world" style chapter. We introduce here a sample project of classifying images of handwritten digits. We will demonstrate three of the main JAX transformations: grad() for taking gradients, jit() for compilation and vmap() for auto-vectorization. With these three transformations, you can already build custom neural network solutions that do not need to be distributed on a cluster. You will develop a complete neural network classifier and understand how a JAX program is structured.

The chapter gives an overall picture highlighting JAX features and essential concepts. We will explain the details in further chapters.

2.1 A toy ML problem: classifying handwritten digits

Let's try to solve a simple but pretty useful image classification problem. The task of image classification is ubiquitous among computer vision tasks: you can classify foods by a photo in a grocery, determine the type of a galaxy in astronomy, determine animal specie or recognize a digit from a ZIP code.

Imagine you have a labeled set of images, each one being assigned a label, say "cat," "dog," or "human." Or a set of images with numbers from 0 to 9 with the corresponding labels. It is usually called *the training set*. Then you want a program that will take a new image without a label, and you want it to assign one of the predefined labels to the photo meaningfully.

In the modern deep learning era, you typically train a neural network to perform this task. You then integrate the trained neural network into a program that will feed the network with data and interpret its outputs.

Classification and regression tasks.

Classification is one of the classical machine learning tasks that, together with regression, falls under the umbrella of supervised learning. In both tasks, you have a dataset of examples (a training set) that provides the supervision signal (hence, the name "supervised learning") of what is correct for each example.

In classification, the supervision signal is a class label, so you must distinguish between a fixed number of classes. It could be classifying a dog breed by a photo, a tweet sentiment by its text, or marking a specific card transaction as a fraud by its characteristics and previous history.

It is called a binary classification for the case where you need to distinguish an object between only two classes. Classes can be mutually exclusive (say, animal species) or not (say, assigning predefined tags to a photo). For the former case, it is called multi-class classification. For the latter, it is a multi-label classification.

In regression, a supervision signal is usually a continuous number; you need to predict this number for new cases. It could be a prediction of room temperature at some point by other measurements and factors, a house price based on its characteristics and location, or the amount of food on your plate based on its photo.

We will take a well-known MNIST dataset (http://yann.lecun.com/exdb/mnist/) of handwritten digits. Figure 2.1 shows some example images from the dataset.

Let's first load this dataset and prepare it to use in the solution we will develop in the chapter.

2.2 Loading and preparing the dataset

As mentioned before, JAX does not include any data loaders, as JAX tends to concentrate on its core strengths. You can easily use TensorFlow or PyTorch data loaders; which one do you prefer or are more familiar with. JAX official documentation contains examples with both of them. We will use TensorFlow Datasets with its data loading API for this particular example.

TensorFlow Datasets contain a version of the MNIST dataset, not surprisingly under the name `mnist`. The total number of images is 70,000. The dataset provides a train/test split with 60,000 images in the train and 10,000 images in the test part. Images are grayscale with the size of 28x28 pixels.

Listing 2.1 Loading the dataset.

```
import tensorflow as tf    #A
import tensorflow_datasets as tfds    #A

data_dir = '/tmp/tfds'    #B

data, info = tfds.load(name="mnist",    #C
                       data_dir=data_dir,    #C
                       as_supervised=True,    #C
                       with_info=True)    #C

data_train = data['train']    #D
data_test  = data['test']    #D
```

#A Importing relevant modules from TensorFlow
#B A temporary directory for downloading the data
#C Loading the MNIST dataset using a function from TensorFlow Datasets. The as_supervised=True parameter returns the data as an (image, label) tuple instead of a dict
#D Extracting train and test splits from the data

After loading the data, we can look at the samples from the dataset with the following code:

Listing 2.2 Showing samples from the dataset.

```
import numpy as np
import matplotlib.pyplot as plt    #A
plt.rcParams['figure.figsize'] = [10, 5]    #A

ROWS = 3    #B
COLS = 10    #B

i = 0
fig, ax = plt.subplots(ROWS, COLS)
for image, label in data_train.take(ROWS*COLS):
    ax[int(i/COLS), i%COLS].axis('off')    #C
    ax[int(i/COLS), i%COLS].set_title(str(label.numpy()))    #C
    ax[int(i/COLS), i%COLS].imshow(np.reshape(image, (28,28)), cmap='gray')    #C
    i += 1

plt.show()
```

#A Importing Matplotlib drawing module and setting the canvas size
#B Parameters for image layout, We want to present images in a grid of 3 rows and 10 columns

#C Displaying each example at the corresponding position inside the grid, turning off the axis coordinates, providing a class label as an image title, and displaying each image with the grayscale palette.

The code above generates an image in Figure 2.1.

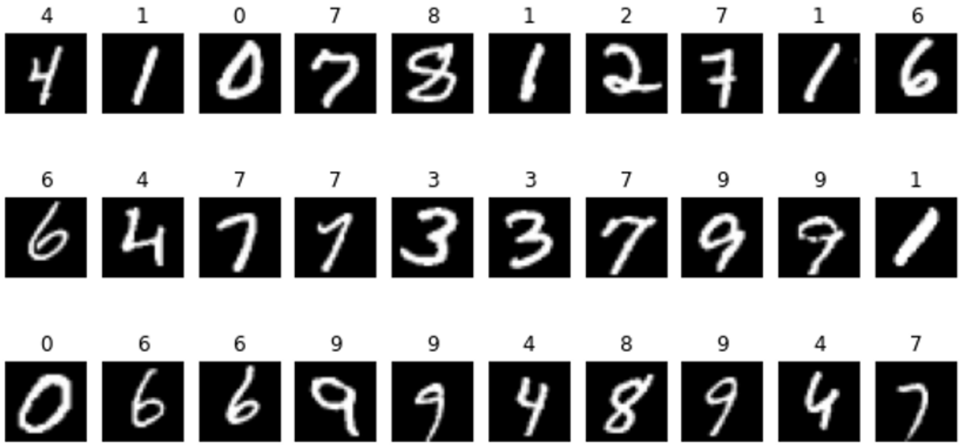


Figure 2.1 Examples from the MNIST dataset. Each handwritten image has a class label displayed on top of it.

As the images are the same size, we can work with them similarly and pack multiple images into a batch. The only preprocessing we might want is normalization. It converts the pixel byte values (`uint8`) from integer values in the range of [0, 255] into the floating type (`float32`) with the range of [0,1].

Listing 2.3 Preprocessing the dataset and splitting it into batches.

```
HEIGHT = 28    #A
WIDTH  = 28    #A
CHANNELS = 1    #A
NUM_PIXELS = HEIGHT * WIDTH * CHANNELS    #A
NUM_LABELS = info.features['label'].num_classes    #A

def preprocess(img, label):
  """Resize and preprocess images."""
  return (tf.cast(img, tf.float32)/255.0), label    #B

train_data = tfds.as_numpy(data_train.map(preprocess).batch(32).prefetch(1))    #C
test_data  = tfds.as_numpy(data_test.map(preprocess).batch(32).prefetch(1))    #C
```

#A Image and dataset parameters
#B The function converts an integer value into a float32 floating-point value and divides it by 255, the maximum integer value in the dataset, to obtain values in the [0, 1] range.
#C Applying our preprocessing function to the train and test splits of the dataset, generating a stream of batches with 32 images each and prefetching one batch.

We ask the data loader to apply the `preprocess` function to each example, pack all the images into a set of batches of the size of 32 items, and also prefetch a new batch without waiting for the previous batch to finish processing on the GPU.

It is enough now, and we can switch to developing our first neural network with JAX.

2.3 A simple neural network in JAX

We start from a simple feed-forward neural network known as a multilayer perceptron (MLP). It is a very simple (and pretty unpractical) network chosen for demonstrating important concepts without too much complexity.

Modern computer vision.

A typical real-world solution to the image classification problem will likely involve a convolutional neural network (CNN). CNN is the type of neural network well suited to work with images. It has remarkable properties like translation equivariance and the ability to learn local features. Later in the book, we will develop a better solution to our problem based on CNNs.

CNNs were long considered the best neural network type for working with images, and many improvements pushed the frontier further. Residual networks, or ResNets, were one of them. So, CNNs were long considered a natural choice for computer vision tasks. Yet, in the last few years situation has changed.

Transformers (https://arxiv.org/abs/1706.03762) emerged in 2017 and were first applied to NLP tasks such as machine translation, and then in 2018, after publishing a model called BERT (https://arxiv.org/abs/1810.04805) to almost any NLP task.

Later, in 2020 the Vision Transformer (ViT) (https://arxiv.org/abs/2010.11929) came and demonstrated exceptional performance on images. This work started a vast wave of Transformer applications for images and videos.

Then, in 2021 the model called MLP-Mixer (https://arxiv.org/abs/2105.01601) emerged along with several other similar works. These works finally found a way to use good old MLPs in computer vision tasks and showed comparable results to transformers.

CNN architectures didn't stand still all that time, and many other improvements appeared. ConvNeXt (https://arxiv.org/abs/2201.03545) and EfficientNetV2 (https://arxiv.org/abs/2104.00298) are among the top performers as of the beginning of 2022.

Interestingly enough, both ViT and MLP-Mixer were created with JAX.

Our solution will be a simple two-layer MLP, a typical "hello world" example for neural networks. Our MLP is much more straightforward than the advanced MLP-Mixer.

The neural network we will develop is shown in Figure 2.2.

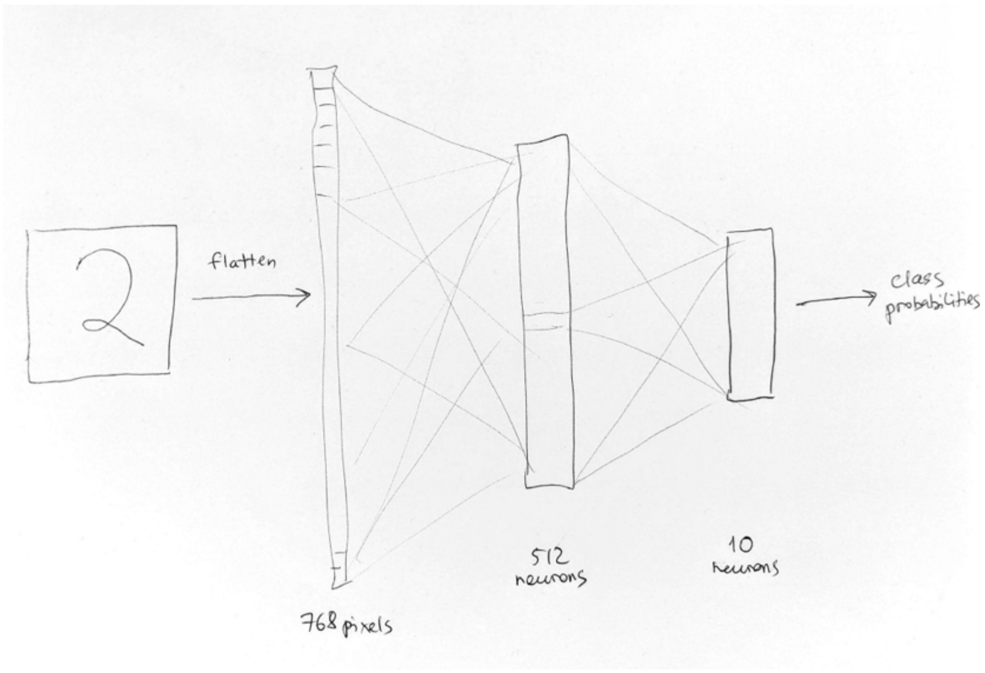


Figure 2.2 Structure of our neural network. The image of 28x28 pixels is flattened into 768 pixels without a 2D structure and passed to the network's input layer. Then a fully-connected hidden layer with 512 neurons follows, after which another fully-connected output layer with ten neurons produces target class activations.

The image is flattened into a 1D array of 768 values (because a 28x28 image contains 768 pixels), and this array is the input to the neural network. The *input layer* maps each of the 768 image pixels to a separate input unit. Then a fully-connected (or dense) layer with 512 neurons follows. It is called the *hidden layer* because it is located between the input and output layers. Each of 512 neurons "looks" at all the input elements simultaneously. Then another fully-connected layer follows. It is called the *output layer*. It has ten neurons, the same number as there are classes in the dataset. Each output neuron is responsible for its class. Say neuron #0 produces the probability of the class "0", neuron #1 for the class "1", and so on.

Each feed-forward layer implements a simple function y = f(x*w+b), consisting of weights **w,** that multiplies incoming data **x,** and biases **b**, which are added to the product. Activation function **f()** is a non-linear function applied to the result of the multiplication and addition.

First, we need to initialize the layer parameters.

2.3.1 Neural network initialization

Before training a neural network, we need to initialize all the **b** and **w** parameters with random numbers:

Listing 2.4 Initializing the neural network.

```
from jax import random

LAYER_SIZES = [28*28, 512, 10]    #A
PARAM_SCALE = 0.01    #B

def init_network_params(sizes, key=random.PRNGKey(0), scale=1e-2):
  """Initialize all layers for a fully-connected neural network with given sizes"""

  def random_layer_params(m, n, key, scale=1e-2):
    """A helper function to randomly initialize weights and biases of a dense layer"""
    w_key, b_key = random.split(key)    #C
    return scale * random.normal(w_key, (n, m)), scale * random.normal(b_key, (n,))    #D

  keys = random.split(key, len(sizes))
  return [random_layer_params(m, n, k, scale) for m, n, k in zip(sizes[:-1], sizes[1:],
       keys)]    #E

params = init_network_params(LAYER_SIZES, random.PRNGKey(0), scale=PARAM_SCALE)
```

#A The list with layer sizes
#B Parameter for scaling random values
#C Generating random keys (more on it in Chapter 7)
#D Generating random values for the layer parameters w and b
#E Running generation for all the layers

Working with random numbers in JAX differs from NumPy because JAX requires pure functions, and the NumPy random number generators (RNG) are not pure because they use a hidden internal state. TensorFlow and PyTorch RNGs usually also use an internal state. JAX implements random number generators that are purely functional, and we will talk about them deeper in Chapter 7. For now, it's enough to understand that you must provide each call of a randomized function with an RNG state which is called a key here, and you should use every key only once, so each time you need a new key, you split an old key into the required amount of new ones.

2.3.2 Neural network forward pass

Then you need a function that performs all the neural network computations, the forward pass. We already have initial values for the **b** and **w** parameters. The only missing part is the activation function. We will use the popular Swish activation function from the `jax.nn` library.

Activation functions.

Activation functions are essential components in the deep learning world. Activation functions provide non-linearity in neural network computations. Without non-linearity, a multi-layer feed-forward network will be equivalent to a single neuron. Because of simple mathematics: a linear combination of linear combinations of inputs is still a linear combination of inputs, which is what a single neuron does. We know that single neuron capabilities for solving complex classification problems are limited to linearly-separable tasks (you probably heard of the well-known XOR problem that is impossible to solve with a linear classifier). So activation functions ensure neural network expressivity and prevent collapse to a simpler model.

There are many different activation functions discovered.

The field started with simple and easy-to-understand functions like the sigmoid or the hyperbolic tangent. They are smooth and have properties mathematicians love, such as being differentiable at each point.

Sigmoid(x) = 1/(1+e^-x)
Tanh(x) = 2/(1+e^-2x) - 1

Then a new kind of function emerged, ReLU, or rectified linear unit. ReLU was not smooth, and its derivative at the point x=0 does not exist. Yet, practitioners found that neural networks learn faster with ReLU.

$$\mathrm{ReLU}(x) = \max(0, x)$$

Then many other activation functions were found, some of them experimentally, others by rational design. Among the popular designed functions are Gaussian Error Linear Units (GELUs, https://arxiv.org/abs/1606.08415) or Scaled Exponential Linear Unit (SELU, https://arxiv.org/abs/1706.02515)

\begin{split}\mathrm{Selu}(x) = \lambda \begin{cases}
 x, & x > 0\\
 \alpha e^x - \alpha, & x \le 0
\end{cases}\end{split}

Among the latest trends in deep learning is automatic discovery. It is typically called the Neural Architecture Search (NAS). The idea of the approach is to design a rich yet manageable search space that describes components of interest. It could be activation functions, layer types, optimizer update equations, etc. Then we run an automatic procedure to search through this space intelligently. Different approaches may also use reinforcement learning, evolutionary computations, or even gradient descent. The Swish (https://arxiv.org/abs/1710.05941) function was found this way.

$$\mathrm{Swish}(x) = x \cdot \mathrm{sigmoid}(\beta x)$$

NAS is an exciting story, and I believe that JAX's rich expressivity can significantly contribute to this field. Maybe some of our readers will make an exciting advancement in deep learning!

Here we develop a forward-pass function, often called a predict function. It takes an image to classify and performs all the forward-pass computations to produce activations on the output layer neurons. The neuron with the highest activation determines the class of the input image (so, if the highest activation is on neuron #5, then, according to the most straightforward approach, the neural network has detected that the input image contains the handwritten number 5).

Listing 2.5 The forward pass of a neural network.

```
import jax.numpy as jnp
from jax.nn import swish    #A

def predict(params, image):    #B
  """Function for per-example predictions."""
  activations = image    #C
  for w, b in params[:-1]:
    outputs = jnp.dot(w, activations) + b    #D
    activations = swish(outputs)    #D

  final_w, final_b = params[-1]    #E
  logits = jnp.dot(final_w, activations) + final_b    #E
  return logits
```

#A Importing the swish activation function
#B Notice we pass both the image and the network parameters to the function!
#C Initialize activations with an input image pixels
#D Successively update activations with the output of each layer
#E For the last layer, we do not apply the activation function

Notice how we pass the list of parameters here. It is different from a typical program in PyTorch or TensorFlow, where these parameters are usually hidden inside a class, and a function uses class variables to access them.

Keep an eye on how the neural network calculations are structured. In JAX, you typically have two functions for your neural networks: one for initializing parameters and another one for applying your neural network to some input data. The first function returns parameters as some data structure (here, a list of arrays, later, it will be a special data structure called PyTree). The second one takes in parameters and the data and returns the result of applying the neural network to the data. This pattern will appear many times in the future, even in high-level neural network frameworks.

That's it. We can use our new function for per-example predictions. Here in Listing 2.6, we generate a random image of the same size as our dataset and pass it to the `predict()` function. You can use a real image from the dataset as well. Do not expect any good results as our neural network is not trained yet. Here we are only interested in the output shape, and we see that our prediction returns a tuple of ten activations for the ten classes.

Listing 2.6 Making a prediction for a single random image.

```
>>>random_flattened_image = random.normal(random.PRNGKey(1), (28*28*1,))    #A
>>>preds = predict(params, random_flattened_image)    #B
>>>print(preds.shape)

(10,)    #C
```

#A Generating a random image of size 28x28 pixels with a single color channel
#B Passing an image to the prediction function
#C The prediction contains a tuple of activations for the ten classes

So, everything looks ok, but we want to work on batches of images, yet our function was designed to work only with a single image. Auto-batching can help here!

2.4 vmap: auto-vectorizing calculations to work with batches

Interestingly, our `predict()` function was designed to work on a single item, and it will not work if we pass in a batch of images. The only substantial change in Listing 2.7 is that we generate a random batch of 32 28x28 pixel images and pass it to the `predict()` function. We added some exception processing as well, but this is only to reduce the size of an error message and to highlight only the most crucial part:

Listing 2.7 Making a prediction for a batch of ten random images with the single-item function.

```
>>>random_flattened_images = random.normal(random.PRNGKey(1), (32, 28*28*1))    #A
>>>try:
>>>  preds = predict(params, random_flattened_images)    #B
>>>except TypeError as e:
>>>  print(e)

Incompatible shapes for dot: got (512, 784) and (32, 784).    #C
```

#A Generating a batch of 32 random images of size 28x28 pixels and a single color channel each
#B Passing a batch of images to the single-element prediction function
#C The error message shows that the single-element prediction function cannot process a batch of images

It is not surprising, as our `predict()` function was a straightforward implementation of matrix calculations, and these calculations assume specific array shapes. The error message says the function got a (512, 784) array with the weights of the first layer and a (32, 784) array with the incoming image, and the `jnp.dot()` function could not do anything with it. It expected 784 numbers for calculating a dot product with the weights array to obtain 512 activations. The new batch dimension (here, the size of 32) confuses it.

Tensors, matrices, vectors, and scalars.

In deep learning, multidimensional arrays are the primary data structures used to communicate between neural networks and their layers. They are also called *tensors*. In mathematics or physics, tensors have a more strict and complicated meaning, just do not be embarrassed if you find something difficult about tensors. Here in deep learning, they are just synonyms for multidimensional arrays. And if you worked with NumPy, you almost know everything you need.

There are particular forms of tensors or multidimensional arrays. A matrix is a tensor with two dimensions (or rank-2 tensor), a vector is a tensor with one dimension (rank-1 tensor), and a scalar (or just a number) is a tensor with zero dimensions (rank-0 tensor). So, tensors are generalizations of scalars, vectors, and matrices to an arbitrary number of dimensions (a rank).

For example, your loss function value is a scalar (just one number). An array of class probabilities at the output of a classification neural network for a single input is a vector of size k (the number of classes) and one dimension (do not confuse size and rank). An array of such predictions for a batch of data (several inputs at once) is a matrix of size k*m (where k is the number of classes and m is the batch size). An RGB image is a rank-3 tensor as it has three dimensions (width, height, color channels). A batch of RGB images is a rank-4 tensor (the new dimension being the batch dimension). A stream of video frames can also be considered a rank-4 tensor (with time being the new dimension). A batch of videos is a rank-5 tensor, and so on. In deep learning, you usually work with tensors with not more than 4 or 5 dimensions.

What are our options to fix the problem?

First, there is a naive solution. We can write a loop that decomposes the batch into individual images and process them sequentially. It will work, but it would be inefficient, as most hardware can potentially do much more calculations in a unit of time. In such a case, it will be significantly underutilized. If you have already worked with MATLAB, NumPy, or something similar, you are familiar with the benefits of vectorization. It would be an efficient solution to the problem.

So, the second option is to rewrite and manually vectorize the `predict()` function so it can accept batches of data at its input. It usually means our input tensors will have an additional batch dimension, and we need to rewrite the calculations to use it. It is straightforward for simple calculations, but it can be complicated for sophisticated functions.

And here comes the third option, automatic vectorization. JAX provides the `vmap()` transformation that transforms a function able to work with a single element into a function able to work on batches. It maps the function over argument axes.

Listing 2.8 Auto-vectorizing a function.

```
from jax import vmap    #A
batched_predict = vmap(predict, in_axes=(None, 0))    #B
```

#A Importing the vmap() transform

#B Creating a function that works with batches based on a function that works on a single item with the help of the vmap() transformation

It's just a one-liner. The `in_axes` parameter controls which input array axes to map over. Its length must equal the number of positional arguments of the function. `None` indicates we need not map any axis, and in our example, it corresponds to the first parameter of the `predict()` function, which is `params`. This parameter stays the same for any forward pass, so we do not need to batch over it (yet, if we used separate neural network weights for each call, we'd use this option). The second element of the `in_axes` tuple corresponds to the second parameter of the `predict()` function, `image`. The value of zero means we want to batch over the first (zeroth) dimension. In the hypothetical case when the batch dimension will be in another position in a tensor, we'd change this number to the proper index.

Now we can apply our modified function to a batch and produce correct outputs:

Listing 2.9 Making a prediction for a batch of ten random images with vmapped function.

```
>>>batched_preds = batched_predict(params, random_flattened_images)    #A
>>>print(batched_preds.shape)

(32, 10)    #B
```

#A Passing a batch of images to the batched_predict() function that we obtained with the vmap() transformation from the original predict() function
#B The output is now correct and contains ten classes activations for each of the 32 elements of the batch

Notice the vital thing. We did not change our original function. We created a new one.

Vmap is a beneficial transformation, as it frees you from doing manual vectorization. Vectorization might not be a very intuitive procedure as you have to think in terms of matrices or tensors and their dimensions. It is not easy for everyone, and it might be an error-prone process, so it is fantastic we have an automated vectorization in JAX. You write a function for a single example, then run it for a batch with the help of `vmap()`.

We are almost done. We created a neural network that can work with batches. The only missing part now is training. Here comes another exciting part. We need to train it.

2.5 Autodiff: how to calculate gradients without knowing about derivatives

We typically use a gradient descent procedure to train a neural network.

Gradient descent procedure.

Gradient descent is a simple iterative procedure for finding the local minima of a differentiable function. For a differentiable function, we can find a gradient, the direction of the most significant change of a function. If we go opposite the gradient, this is the direction of the steepest descent. We go to a local minimum of a function by taking repeated steps.

Neural networks are differentiable functions determined by their parameters (weights). We want to find a combination of weights that minimizes some loss function, which calculates the discrepancy between the model prediction and the ground truth values. If the difference is smaller, the predictions are better. We can apply gradient descent to solve this task.

We start from some random weights, a chosen loss function, and a training dataset. Then we repeatedly calculate the gradient of the loss function with respect to current weights on the training dataset (or some batch from the dataset). After calculating a gradient for each weight in a neural network, we can update each weight in the opposite direction to the gradient, subtracting some part of the gradient from the weight value. The procedure stops after some predefined number of iterations, when loss improvement stops, or by other criteria.

The process is visualized in the following figure:

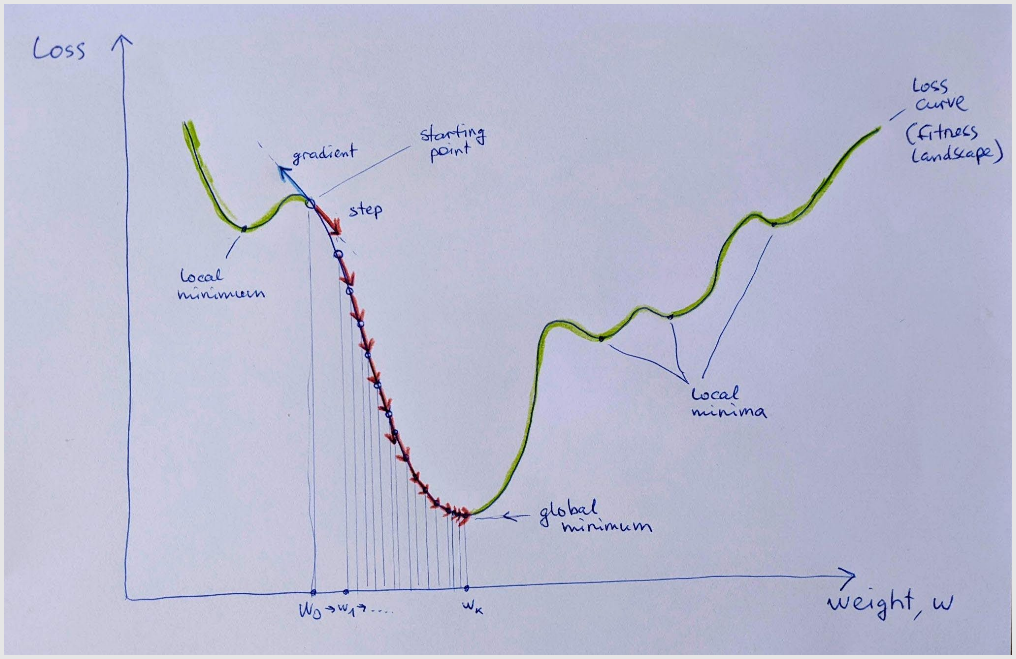


Figure 2.3 Visualizing gradient descent steps along the loss landscape

The loss function is the curve where for any weight value there is a corresponding loss value. This curve is also called a loss curve or a fitness landscape (this make sense in more complex cases with more than a single dimension).

Here, we started from some initial random weight (W_0), and after a series of steps came to the global minimum corresponding to some specific weight (W_k). The special parameter called a *learning rate* determines how big or a small amount of the gradient we take.

Repeating such steps, we follow the trajectory that leads to a local minimum of the loss function. And we hope this local minimum is the same as the global one, or at least not significantly worse. Strangely enough, this works for neural networks. Why it works so well is an interesting topic on its own.

In the image above for demonstration purposes I have chosen a “good” starting point for which it is easy to come to the global minimum. Starting in other places of the fitness landscape may lead to local minima, and there are few of them in the image.

This vanilla gradient descent procedure has many improvements, including gradient descent with momentum and adaptive gradient descent methods such as Adam, Adadelta, RMSProp, LAMB, etc. Many of them help in overcoming some local minima as well.

We will use a straightforward mini-batch gradient descent without momentum or adaptivity, similar to a vanilla stochastic gradient descent (SGD) optimizer in any deep learning framework. We will use only one improvement compared to the plain SGD, the exponential decaying learning rate. However, it is unnecessary in our case.

To implement such a procedure, we need to start with some random point in the parameter space (we already did when we initialized the neural network parameters in the previous section).

Then we need a loss function to evaluate our current set of parameters on the training dataset. The loss function calculates the discrepancy between the model prediction and the ground truth values from the training dataset labels. There are many different loss functions for specific machine learning tasks, and we will use a simple loss function suitable for multi-class classification, the categorical cross-entropy function.

Listing 2.10 Implementing the loss function.

```
from jax.nn import logsumexp    #A

def loss(params, images, targets):
  """Categorical cross entropy loss function."""
  logits = batched_predict(params, images)    #B
  log_preds = logits - logsumexp(logits)    #C
  return -jnp.mean(targets*log_preds)    #D
```

#A Importing logsumexp function.
#B Generate neural network activations (frequently called logits) for an input batch of images
#C Calculate log probabilities with the help of logsumexp() function
#D Calculate the categorical cross-entropy loss value

Here we use a `logsumexp()` function, a common trick in machine learning to normalize a vector of log probabilities without having under- or over-flow numerical issues. If you want to learn more about this topic, here is one reasonable explanation: https://gregorygundersen.com/blog/2020/02/09/log-sum-exp/.

The `loss()` function needs ground truth values, or targets, to calculate the discrepancy between the model prediction and the ground truth. Model predictions are already in the form of class activations, where each output neuron produces some score for the corresponding class. The targets are originally just a class number. For the class '0', it is number 0; for the class '1', it is number 1, and so on. We need to convert these numbers into activations, and in such cases, special one-hot encoding is used. The class '0' produces an array of activations with a number 1 being at position 0 and zeros at all the other positions. The class '1' produces a number 1 at position 1, and so on. This conversion will be done outside of the `loss()` function.

After the loss function is defined, we are ready to implement the gradient descent update.

The logic is simple. We have to calculate gradients of the loss function with respect to the model parameters based on the current batch of data. Here comes the `grad()` transformation. The transformation takes in a function (here, the loss function). It creates a function that evaluates the gradient of the loss function with respect to a specific parameter, by default, the first parameter of the function (here, the `params`).

It is an important distinction from other frameworks like TensorFlow and PyTorch. In those frameworks, you usually get gradients after performing the forward pass, and the framework tracks all the operations being done on the tensors of interest. JAX uses a different approach. It transforms your function and generates another function that calculates gradients. And then, you calculate the gradients by providing all the relevant parameters, the neural network weights, and the data into this function.

Here we calculate the gradients and then update all the parameters in the direction opposite to the gradient (hence, the minus sign in the weight update formulas). All the gradients are scaled with the learning rate parameter that depends on the number of epochs (one epoch is a complete pass through the training set). We made an exponentially decaying learning rate, so for later epochs, the learning rate will be lower than for the earlier ones:

Listing 2.11 Implementing gradient update step.

```
from jax import grad    #A

INIT_LR = 1.0    #B
DECAY_RATE = 0.95    #C
DECAY_STEPS = 5    #D

def update(params, x, y, epoch_number):
  grads = grad(loss)(params, x, y)    #E
  lr = INIT_LR * DECAY_RATE ** (epoch_number / DECAY_STEPS)    #F
  return [(w - lr * dw, b - lr * db)    #G
          for (w, b), (dw, db) in zip(params, grads)]    #G
```

#A Importing the grad() transform
#B Initial learning rate
#C The learning rate decay parameter

#D This parameter determines how many epochs have to pass before the learning rate decays once more
#E We generate a function for calculating gradients with the grad() transform and immediately apply it to the current parameters and data (params, x, y) to obtain gradient values
#F Calculating the learning rate for the current step
#G Returning updated parameters by taking a small step (determined by the learning rate parameter, lr) in the direction opposite to the gradient

In this example, you do not calculate your loss function directly. You calculate only gradients. In many cases, you also want to track the loss values, and JAX provides another function, `value_and_grad()`, that calculates both the value and the gradient of a function. We can change the `update()` function accordingly:

Listing 2.12 Implementing gradient update step with both loss value and gradient.

```
from jax import value_and_grad    #A

def update(params, x, y, epoch_number):
  loss_value, grads = value_and_grad(loss)(params, x, y)    #B
  lr = INIT_LR * DECAY_RATE ** (epoch_number / DECAY_STEPS)
  return [(w - lr * dw, b - lr * db)
          for (w, b), (dw, db) in zip(params, grads)], loss_value    #C
```

#A Importing value_and_grad function.
#B Calculating both the loss value and the gradient
#C Returning both updated parameters and the loss value

We did everything we needed; the last thing was to run a loop for a specified number of epochs. To do it, we need a few more utility functions to calculate accuracy and add some logging to track all the relevant information during training:

Listing 2.13 Implementing gradient descent.

```
from jax.nn import one_hot    #A

def batch_accuracy(params, images, targets):    #B
  images = jnp.reshape(images, (len(images), NUM_PIXELS))
  predicted_class = jnp.argmax(batched_predict(params, images), axis=1)
  return jnp.mean(predicted_class == targets)

def accuracy(params, data):    #C
  accs = []
  for images, targets in data:
    accs.append(batch_accuracy(params, images, targets))
  return jnp.mean(jnp.array(accs))

import time

for epoch in range(num_epochs):
  start_time = time.time()
  losses = []
  for x, y in train_data:
    x = jnp.reshape(x, (len(x), NUM_PIXELS))    #D
    y = jax.nn.one_hot(y, NUM_LABELS)    #E
    params, loss_value = update(params, x, y, epoch)    #F
    losses.append(loss_value)    #G
  epoch_time = time.time() - start_time    #H

  start_time = time.time()
  train_acc = accuracy(params, train_data)    #I
  test_acc = accuracy(params, test_data)    #I
  eval_time = time.time() - start_time
  print("Epoch {} in {:0.2f} sec".format(epoch, epoch_time))
  print("Eval in {:0.2f} sec".format(eval_time))
  print("Training set loss {}".format(jnp.mean(jnp.array(losses))))
  print("Training set accuracy {}".format(train_acc))
  print("Test set accuracy {}".format(test_acc))
```

#A One more helper function to generate one-hot encoding for the class label
#B Calculate accuracy (the percentage of correct answers) for a batch
#C Calculate accuracy for a whole dataset with many batches
#D Flattening an input image into a 1D array
#E Converting a class label into one-hot encoding
#F Updating parameters by a single step of the gradient descent procedure
#G Storing the loss value for training statistics
#H Logging time for training statistics
#I One per epoch calculate accuracy on the train and test data

We are ready to run our first training loop.

```
Epoch 0 in 36.39 sec
Eval in 8.06 sec
Training set loss 0.41040700674057007
Training set accuracy 0.9299499988555908
Test set accuracy 0.931010365486145
Epoch 1 in 32.82 sec
Eval in 6.47 sec
Training set loss 0.37730318307876587
Training set accuracy 0.9500166773796082
Test set accuracy 0.9497803449630737
Epoch 2 in 32.91 sec
Eval in 6.35 sec
Training set loss 0.3708733022212982
Training set accuracy 0.9603500366210938
Test set accuracy 0.9593650102615356
Epoch 3 in 32.88 sec
…
Epoch 23 in 32.63 sec
Eval in 6.32 sec
Training set loss 0.35422590374946594
Training set accuracy 0.9921666979789734
Test set accuracy 0.9811301827430725
Epoch 24 in 32.60 sec
Eval in 6.37 sec
Training set loss 0.354021817445755
Training set accuracy 0.9924833178520203
Test set accuracy 0.9812300205230713
```

We trained our first neural network with JAX. It seems everything works, and it solves the problem of handwritten digit classification on the MNIST dataset with an accuracy of 98.12%. Not bad.

Our solution takes more than 30 seconds per epoch and additional 6 seconds for the evaluation run each epoch. Is it fast on not?

Let's check what improvements can be made with the Just-in-Time compilation.

2.6 JIT: compiling your code to make it faster

We have already implemented an entire neural network for handwritten digit classification. It may even use GPU if you run it on a GPU machine, as all the tensors are placed by default on GPU in that case. However, we can make our solution even faster!

The previous solution did not use JIT compilation and acceleration provided by XLA. Let's do it.

Compiling your functions is easy. You can use either `jit()` function transformation or @jit annotation. We will use the latter.

Here we compile the two most resource-heavy functions, the `update()` and `batch_accuracy()` functions. All that you need is to add the `@jit` annotation before the function definitions:

Listing 2.14 Adding JIT compilation to the code.

```
from jax import jit    #A

@jit    #B
def update(params, x, y, epoch_number):
  loss_value, grads = value_and_grad(loss)(params, x, y)
  lr = INIT_LR * DECAY_RATE ** (epoch_number / DECAY_STEPS)
  return [(w - lr * dw, b - lr * db)
          for (w, b), (dw, db) in zip(params, grads)], loss_value

@jit
def batch_accuracy(params, images, targets):
  images = jnp.reshape(images, (len(images), NUM_PIXELS))
  predicted_class = jnp.argmax(batched_predict(params, images), axis=1)
  return jnp.mean(predicted_class == targets)
```

#A Importing the jit() transform
#B Using the jit() transform as a function annotation

After reinitializing neural network parameters and rerunning the training loop, we get the following results:

```
Epoch 0 in 2.15 sec
Eval in 2.52 sec
Training set loss 0.41040700674057007
Training set accuracy 0.9299499988555908
Test set accuracy 0.931010365486145
Epoch 1 in 1.68 sec
Eval in 2.06 sec
Training set loss 0.37730318307876587
Training set accuracy 0.9500166773796082
Test set accuracy 0.9497803449630737
Epoch 2 in 1.69 sec
Eval in 2.01 sec
Training set loss 0.3708733022212982
Training set accuracy 0.9603500366210938
Test set accuracy 0.9593650102615356
Epoch 3 in 1.67 sec
…
Epoch 23 in 1.69 sec
Eval in 2.07 sec
Training set loss 0.35422590374946594
Training set accuracy 0.9921666979789734
Test set accuracy 0.9811301827430725
Epoch 24 in 1.69 sec
Eval in 2.06 sec
Training set loss 0.3540217876434326
Training set accuracy 0.9924833178520203
Test set accuracy 0.9812300205230713
```

Quality is the same, but the speed improved significantly. Now an epoch takes nearly 1.7 seconds instead of 32.6, and an evaluation run takes close to 2 seconds instead of 6.3 seconds. That’s a pretty significant improvement!

You might also notice that the first iterations take longer than the subsequent ones. Because compilation happens during the first run of a function, the first run is slower. Subsequent runs use the compiled function and are faster. We will dive deeper into the JIT compilation in Chapter 5.

We are done with our machine learning problem. Now a bit more general thoughts.

2.7 Pure functions and composable transformations: why is it important?

We have created and trained our first neural network with JAX. We highlighted some significant differences between JAX and classical frameworks like PyTorch and TensorFlow along the way. These differences are based on the functional approach JAX follows.

As we have said several times, JAX functions must be pure. It means their behavior must be defined only by their inputs, and the same input must always produce the same output. No internal state that affects calculations is allowed. Also, side effects are not allowed as well.

There are many reasons why pure functions are good. Among them are easy parallelization, caching, and ability to make functional compositions like jit(vmap(grad(some_function))). Debugging is also becoming easier.

One crucial difference we noticed is related to random numbers. NumPy random number generators (RNG) are impure as they contain an internal state. JAX makes its RNG explicitly pure. Now a state is passed into a function that requires randomness. So, given the same state, you will always produce the same "random" numbers. So, be careful. We will talk about RNG in Chapter 7.

Another critical difference is that neural network parameters are not hidden inside some object but are always explicitly passed around. And many neural network computations are structured in the following way: first, you generate, or initialize, your parameters; then, you pass them into a function that uses them for calculations. We will see this pattern in high-level neural network libraries on top of JAX, like Flax or Haiku.

The neural network parameters become an entity on their own. And such structure gives you much freedom in what you do. You can implement custom updates, save and restore them easily, and create various function compositions.

Having no side effects is especially important when compiling your functions with JIT. You can get unexpected results when ignoring purity as `jit()` compiles and caches a function. If the function behavior is influenced by some state or produces some side effects, then a compiled version might save computations that happened during its first run and reproduce them on the subsequent calls, which might not be what you want. We will talk about it more in Chapter 5.

Let's finally summarize what a JAX deep learning project looks like.

2.8 An overview of a JAX deep learning project

A typical JAX deep learning project is depicted in Figure 2.4 and consists of several things:

1. A dataset for your particular task.
2. A data loader to read your dataset and transform it into a sequence of batches. As we have said, JAX does not contain its own data loaders, and you can use excellent tools for loading data from PyTorch or TensorFlow.
3. A model is defined as a set of model parameters and a function performing computations given the parameters (remember, JAX needs pure functions without state and side effects). The function can be defined using JAX primitives, or you can use a high-level neural network library like Flax or Haiku.
4. The model function can be auto-vectorized with `vmap()` and compiled with `jit()`. You can also distribute it to a cluster of computers with `pmap()`.
5. A loss function takes in model parameters and a batch of data. It calculates the loss value (usually some kind of error we want to minimize), but we really need the gradients of the loss function with respect to the model parameters. So, we obtain a gradient function with the help of the `grad()` JAX transformation.
6. The gradient function is evaluated on the model parameters and the input data and produces gradients for each model parameter.
7. The gradients are used to update the model parameters with some gradient descent procedure. You can update model parameters directly or use a special optimizer from a separate library (say, Optax).
8. After running the training loop for several epochs, you get a trained model (an updated set of the parameters) that can be used for predictions or any other task you designed the model for.
9. You can use your trained model to deploy it to production. There are several options available. For example, you can convert your model to TensorFlow or TFLite, or use the model with Amazon SageMaker.

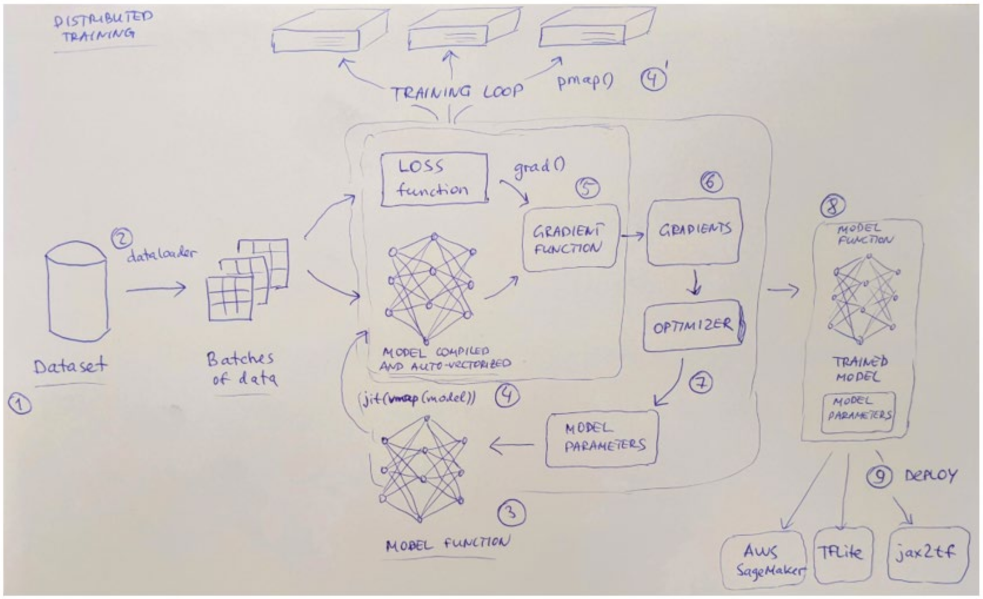


Figure 2.4 High-level structure of a JAX project, with data loading, training, and deployment to production

Different parts of the project can be replaced with modules from the ecosystem, even in this typical project. If you work with more advanced topics in machine learning like reinforcement learning, graph neural networks, meta-learning, or evolutionary computations, then you will add more special modules from the JAX ecosystem and/or change some parts of this general scheme.

That’s it! We are ready to dive into the core JAX.

2.9 Exercises

- Modify the neural network architecture and/or training process to improve classification quality.
- Implement a tweet sentiment classifier.

2.10 Summary

- JAX has no its own data loader, so you can use an external one from PyTorch or TensorFlow
- In JAX, parameters of a neural network are typically passed as an external parameter to the function performing all the calculations, not stored inside an object as it is usually done in TensorFlow/PyTorch
- The `vmap()` transformation converts a function for a single input into a function working on a batch.
- You can calculate a gradient of your function with the `grad()` function
- If you need both a value and a gradient of the function you can use the `value_and_grad()` function
- The `jit()` transformation compiles your function with XLA linear algebra compiler and produces optimized code able to run on GPU or TPU
- You need to use pure functions with no internal state and side effects for your transformations to work correctly

3 *Working with tensors*

This chapter covers

- **Working with NumPy arrays**
- **Working with JAX tensors on CPU/GPU/TPU**
- **Adapting code to differences between NumPy arrays and JAX DeviceArray**
- **Using high-level and low-level interfaces: jax.numpy and jax.lax**

In the previous two chapters we showed what JAX is and why to use it and developed a simple neural network on JAX. With this chapter, we start diving deeper into the JAX core, beginning with tensors.

Tensor or multi-dimensional array is the basic data structure in deep learning and scientific computing frameworks. Every program relies on some form of tensor, be it a 1D array, a 2D matrix, or a higher-dimensional array. Handwritten digit images from the previous chapter, intermediate activations, and the resulting network predictions — everything is a tensor.

NumPy arrays and their API became the de-facto industry standard that many other frameworks respect. This chapter will cover tensors and their respective operations in JAX. We will highlight the differences between NumPy and JAX APIs.

3.1 Image processing with NumPy arrays

Let's start with a real-life image processing task. Imagine you have a collection of photos you want to process. Some photos have an extra space to crop, others have noise artifacts you want to remove, and many are good, but you want to apply artistic effects to them. For simplicity, let's focus only on denoising images, as shown in Figure 3.1:

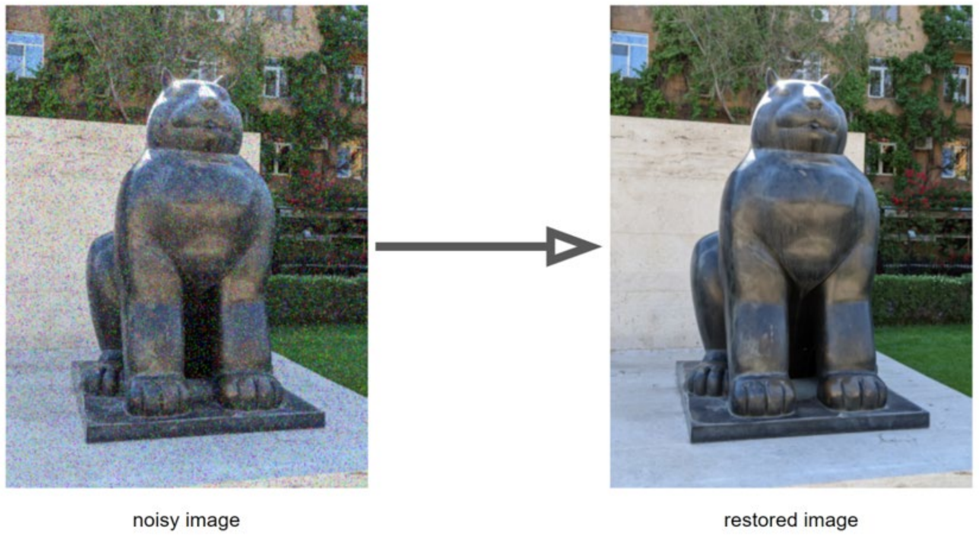


Figure 3.1 Example of image processing we want to implement

To implement such processing, you must load images into some data structure and write functions to process them.

Suppose you have a photo (I've chosen one of the cat statues by Fernando Botero at the Yerevan Cascade complex), and your camera produced some strange noise artifacts on it. You want to remove the noise, then maybe also apply some artistic filters to the image. Of course, there are many image processing tools, but we'd like to implement our own pipeline for demonstration purposes. You may also want to add neural network processing to the pipeline in the future, for example, implementing super-resolution or more artistic filters. With JAX, it's pretty straightforward.

3.1.1 Loading and storing images in NumPy arrays

First, we need to load images.

Images are good examples of multidimensional objects. They have two spatial dimensions (width and height) and usually have another dimension with color channels (typically red, green, blue, and sometimes alpha). Therefore, images are naturally represented with multi-dimensional arrays, and the NumPy array is a suitable structure to keep images in computer memory.

A corresponding image tensor also has three dimensions: width, height, and color channels. Black and white or grayscale images may lack the channel dimension. In NumPy indexing, the first dimension corresponds to rows, while the second one corresponds to columns. The channel dimension can be put before the spatial dimensions or after it. Technically it can also reside between the height and width dimension, but it makes no sense. In the `scikit-image`, the channels are aligned as RGB (red is the first, then green, then blue). Other libraries like OpenCV may use another layout, say, BGR, with the reverse order of the color channels.

Let's load the image and look at its properties. I put my image with the name `'The_Cat.jpg'` into the current folder. You can put any other image instead of it, but do not forget to change the filename in the code.

We will use the well-known `scikit-image` library for loading and saving images. In Google Colab, it is already preinstalled, but if you lack this library, install it according to the instruction at https://scikit-image.org/docs/stable/install.html.

The following code can be found in the book code repository on GitHub: https://github.com/che-shr-cat/JAX-in-Action.

The following code loads the image:

Listing 3.1 Loading an image into NumPy array.

```
>>>import numpy as np    #A
>>>from scipy.signal import convolve2d    #A

>>>from matplotlib import pyplot as plt    #A
>>>from skimage.io import imread, imsave    #A
>>>from skimage.util import img_as_float32, img_as_ubyte, random_noise    #A

>>>img = imread('The_Cat.jpg')    #B

>>>from matplotlib import pyplot as plt    #C
>>>plt.figure(figsize = (6, 10))    #C
>>>plt.imshow(img)    #C
```

#A Importing all we need here and later
#B Loading the image
#C Displaying the image

The image will be displayed as in Figure 3.2. Depending on your screen resolution, you may want to change the image size. Feel free to do it using the `figsize` parameter. It is a `(width, height)` tuple where each value is in inches.

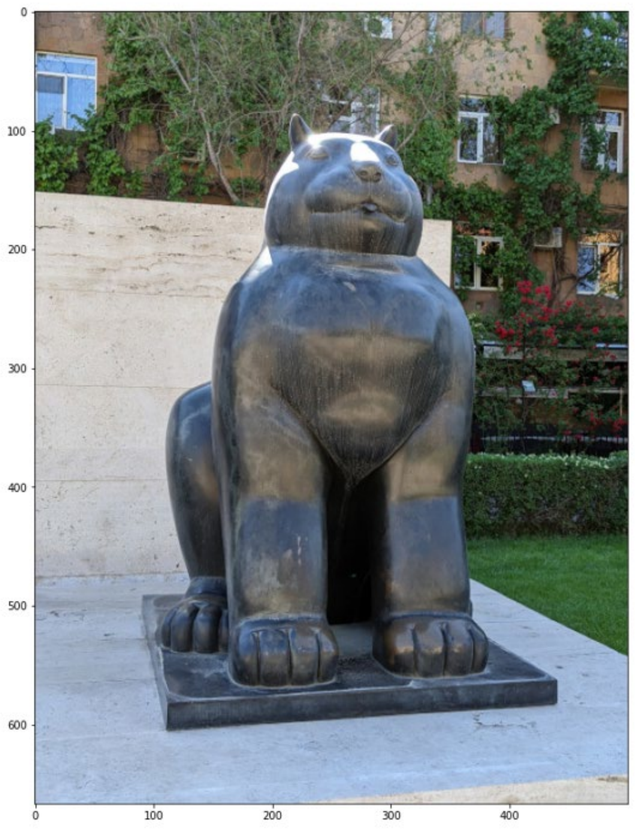


Figure 3.2 The color image that is represented as a 3-dimensional array with height, width, and color dimensions.

We can check the type of the image tensor:

```
>>>type(img)

numpy.ndarray    #A
```

#A This is a NumPy ndarray type

Tensors are usually described by their shapes, a tuple with the number of elements equal to the tensor rank (or the number of dimensions). Each tuple element represents a number of index positions along the dimension. The `shape` property references it. The number of dimensions may be known with the `ndim` property.

For example, a 1024*768 color image might be represented by a shape (768, 1024, 3) or (3, 768, 1024).

When you work with a batch of images, a new batch dimension is added, typically being the first one with index 0.

A batch of 32 1024*768 color images may be represented with a shape (32, 768, 1024, 3) or (32, 3, 768, 1024).

```
>>>img.ndim
3    #A

>>>img.shape
(667, 500, 3)    #B
```

#A The image tensor has 3 dimensions
#B Their size is 667 (height), 500 (width), and 3 (color channels)

As you see, the image is represented by a 3-dimensional array with the first dimension being height, the second being width, and the third being color channels.

Tensor Layouts In Memory: NCHW vs. NHWC.

Image tensors are usually represented in memory in two common formats: NCHW or NHWC. These uppercase letters encode tensor axis semantics, where N stands for batch dimension, C for channel dimension, H for height, and W for width. A tensor described this way contains a batch composed of N images of C color channels, each with height H and width W. It's problematic to have objects of different sizes in a batch, but special data structures, such as ragged tensors, exist. Another option is to pad different objects to the same size with some placeholder element.

Different frameworks and libraries prefer different formats. JAX (https://jax.readthedocs.io/en/latest/_autosummary/jax.lax.conv_general_dilated.html) and PyTorch (https://discuss.pytorch.org/t/why-does-pytorch-prefer-using-nchw/83637/4) use NCHW by default, TensorFlow (https://www.tensorflow.org/api_docs/python/tf/nn/conv2d) uses NHWC, and almost any library has some function to convert between these formats or a parameter specifying what type of an image tensor is being passed to a function.

From the mathematical point of view, these representations are equivalent, but from the practical point of view, there may be a difference. For example, convolutions implemented in NVIDIA Tensor Cores require NHWC layout and work faster when input tensors are laid out in the NHWC format (https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html#tensor-layout).

Tensor also has a data type associated with all the elements. It is referenced as `dtype`. Images are usually represented either with float values in the range of [0, 1] or unsigned integers in the range of [0, 255]. The values inside our tensor are unsigned 8-bit integers (uint8):

```
>>>img.dtype
dtype('uint8')    #A
```

#A The image is encoded with unsigned 8-bit integers

There are also two useful properties related to size. The `size` property returns the number of elements in the tensor. It is equal to the product of the array’s dimensions. The `nbytes` property returns the total bytes consumed by the elements of the array. It does not include memory consumed by non-element attributes of the array object. For a tensor comprising `uint8` values, these properties return the same number as each element requires only a single byte. For `float32` values, there are four times more bytes than the number of elements.

```
>>>img.size
1000500    #A
>>>img.nbytes
1000500    #B
```

#A There are 1000500 elements in the image tensor
#B The tensor takes 1000500 bytes (each element is a one-byte value)

The powerful feature of NumPy is slicing, and we can directly use it for cropping. With slicing, you can select specific elements of a tensor along each axis. For example, you can select a specific rectangular image subregion or take only chosen color channels.

Listing 3.2 Slicing NumPy arrays.

```
>>>cat_face = img[80:220, 190:330, 1]    #A
>>>cat_face.shape

(140, 140)    #B

>>>plt.figure(figsize = (3,4))
>>>plt.imshow(cat_face, cmap='gray')
```

#A We slice the pixels from line 80 to the line 220 (not including) by height, columns from 190 to 330, and the second color channel (with index of one, as indices start from zero)
#B The resulting tensor is 140 pixels height and 140 pixels width

This code selects pixels related to the cat’s face only from the image's green channel (the middle one with the index of 1). We displayed the image as grayscale as it contains only a single color channel information. You may choose another color palette if you want. The resulting image is shown in Figure 3.3.

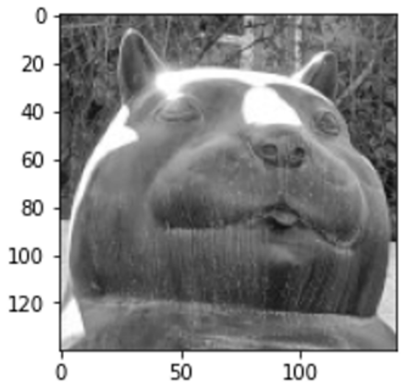


Figure 3.3 Cropped image with a single color channel

You can also easily implement basic things like image flipping and rotation with array slicing.

The code `img = img[:,::-1,:]` reverses the order of pixels along the horizontal dimension while preserving the vertical and channel axes. You can also use the `flip()` function from NumPy. Rotation can be performed with the `rot90()` function for a given number of rotations (parameter k=2) as in the code `img = np.rot90(img, k=2, axes=(0, 1))`.

We are now ready to implement some more advanced image processing.

3.1.2 Performing basic image processing with NumPy API

First, we will convert our image from the `uint8` data type to `float32`. It will be easier for us to work with floating-point values in the range of [0.0, 1.0]. We use the `img_as_float()` function from `scikit-image`.

Imagine we have a noised version of the image. To simulate the situation, we use the Gaussian noise, which frequently appears in digital cameras with low-light conditions and high ISO light sensitivities. We use a `random_noise` function from the same `scikit-image` library.

Listing 3.3 Generating a noisy version of the image.

```
>>>img = img_as_float32(img)    #A
>>>img.dtype

dtype('float32')    #B

>>>img_noised = random_noise(img, mode='gaussian')    #C

>>>plt.figure(figsize = (6, 10))
>>>plt.imshow(img_noised)
```

#A Converting an image from bytes (unsigned 8-bit integer) to 32-bit floating point values
#B The tensor data type is changed to the ‘float32’
#C We use a function from scikit-image to add random noise with specified type

The code generates a noisy version of the image displayed in Figure 3.4. You may also experiment with other interesting noise types, such as salt-and-pepper noise or impulse noise, occurring as a sparse minimum and maximum value pixels.

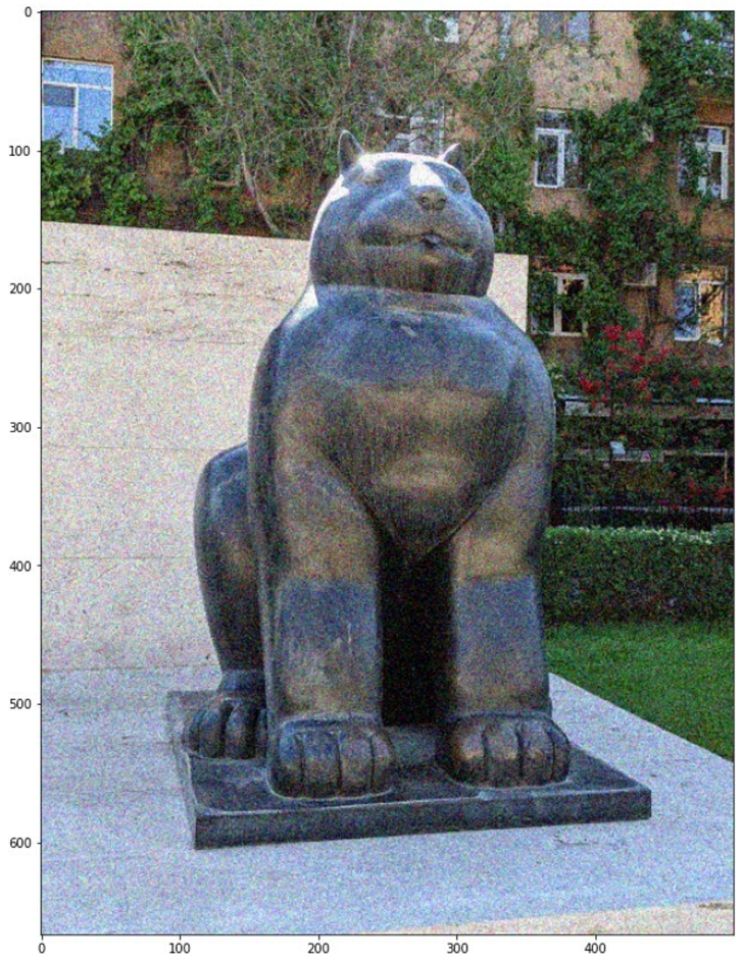


Figure 3.4 The noisy version of the image

Such type of noise is typically removed by Gaussian blur filters. Gaussian blur filter belongs to a large family of matrix filters also called finite impulse response, or FIR, filters in the

Digital Signal Processing (DSP) field. You may also have seen matrix filters in image processing applications like Photoshop or GIMP.

FIR filters and convolution.

An FIR filter is described by its matrix, also called the kernel. The matrix contains weights with which pixels of the image are taken when the filter slides along the image. During each step, all the pixels of a window through which the kernel “looks” at part of the image get multiplied by the corresponding weights of the kernel. Then the products are summed into a single number and this number is the resulting pixel intensity for a pixel in the output (filtered) image. This operation that takes in a signal and a kernel (another signal) and produces a filtered signal is called *convolution*. This process is visualized in Figure 3.4.

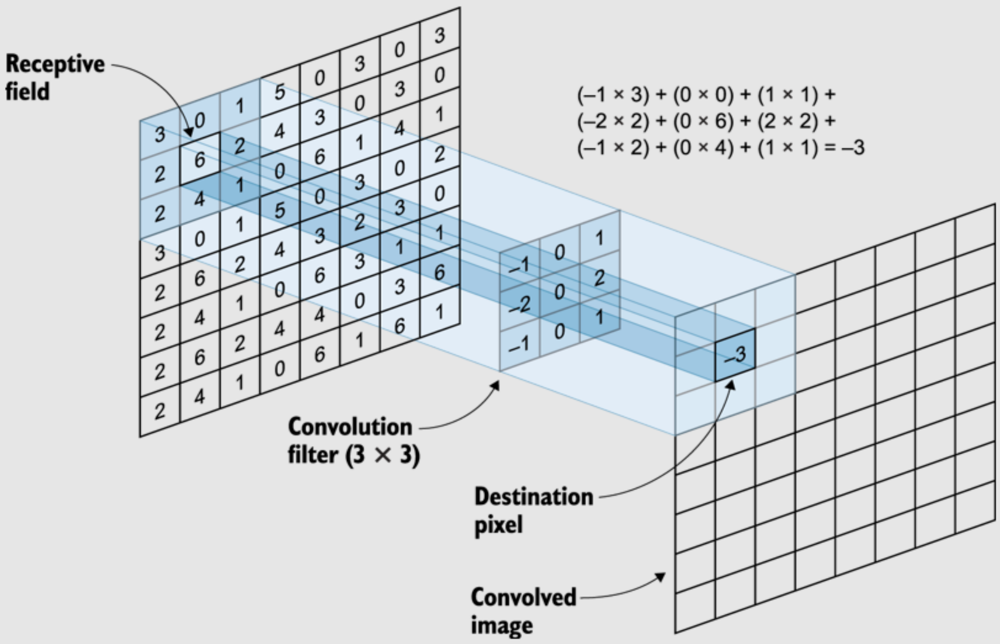


Figure 3.5 An FIR filter is implemented by convolution (image from the book “Deep Learning for Vision Systems” by Mohamed Elgendy)

The convolution can be any-dimensional. For 1D inputs, it is a 1D convolution; for 2D inputs (like images), it's a 2D convolution, and so on. This is the same operation heavily used in convolutional neural networks (CNNs). We will apply a 2D convolution to each image channel separately. In CNNs all the image channels are usually processed at once with a kernel of the same channel dimension as the image.

If you want to learn more about this digital filters and convolutions, I’d recommend the excellent book on Digital Signal Processing: http://dspguide.com/.

For example, a simple blur (not Gaussian blur) filter comprise of a matrix of equal values. It means that each pixel in the neighborhood of the target pixel is taken with an equal weight. It is equivalent to averaging all values inside the receptive field of the kernel. We will use a kernel of the size 5x5 pixels. You may have heard about such a filter as a simple moving average filter.

Listing 3.4 Matrix (or kernel) for a simple blur filter.

```
>>>kernel_blur = np.ones((5,5))    #A
>>>kernel_blur /= np.sum(kernel_blur)    #B
>>>kernel_blur

array([[0.04, 0.04, 0.04, 0.04, 0.04],    #C
       [0.04, 0.04, 0.04, 0.04, 0.04],    #C
       [0.04, 0.04, 0.04, 0.04, 0.04],    #C
       [0.04, 0.04, 0.04, 0.04, 0.04],    #C
       [0.04, 0.04, 0.04, 0.04, 0.04]])    #C
```

#A Generating a 5x5 matrix of ones
#B Dividing each element of the matrix by sum of elements
#C The resulting array contains the same numbers that together sum to one

Gaussian blur is a more involved version of the blur filter, its matrix contains different values with the tendency towards having higher values closer to the center. Gaussian kernel is produced by the well-known Gaussian function.

Listing 3.5 Gaussian blur kernel.

```
def gaussian_kernel(kernel_size, sigma=1.0, mu=0.0):
    """ A function to generate Gaussian 2D kernel """
    center = kernel_size // 2    #A
    x, y = np.mgrid[    #B
        -center : kernel_size - center,    #B
        -center : kernel_size - center]    #B
    d = np.sqrt(np.square(x) + np.square(y))
    koeff = 1 / (2 * np.pi * np.square(sigma))
    kernel = koeff * np.exp(-np.square(d-mu) / (2 * np.square(sigma)))    #C
    return kernel

>>>kernel_gauss = gaussian_kernel(5)
>>>kernel_gauss

array([[0.00291502, 0.01306423, 0.02153928, 0.01306423, 0.00291502],    #D
       [0.01306423, 0.05854983, 0.09653235, 0.05854983, 0.01306423],    #D
       [0.02153928, 0.09653235, 0.15915494, 0.09653235, 0.02153928],    #D
       [0.01306423, 0.05854983, 0.09653235, 0.05854983, 0.01306423],    #D
       [0.00291502, 0.01306423, 0.02153928, 0.01306423, 0.00291502]])    #D
```

#A Finding the center position of the kernel
#B Generating the X,Y grid values
#C Generating the kernel coefficients according to the formula
#D The resulting kernel

We now need to implement the function to apply the filter to an image. We will apply a filter to each color channel separately. Inside each color channel the function needs to apply a 2D convolution of the color channel and the kernel. We also clip the resulting values to restrict the range in [0.0, 1.0]. The function assumes the channel dimension is the last one.

Listing 3.6 Function to apply a filter to an image.

```
def color_convolution(image, kernel):
""" A function to generate Gaussian 2D kernel """
    channels = []
    for i in range(3):
        color_channel = image[:,:,i]    #A
        filtered_channel = convolve2d(color_channel, kernel, mode="same")    #B
        filtered_channel = np.clip(filtered_channel, 0.0, 1.0)    #C
        channels.append(filtered_channel)
    final_image = np.stack(channels, axis=2)    #D
    return final_image
```

#A Extract a channel with slicing
#B Applying the filter to the extracted channel
#C Clipping the values to the range of [0.0, 1.0]
#D Generating the final image by concatenating filtered channels

We are ready to apply our filter to the image:

Listing 3.7 Filtering the image from noise.

```
>>>img_blur = color_convolution(img_noised, kernel_gauss)    #A
>>>plt.figure(figsize = (12,10))
>>>plt.imshow(np.hstack((img_blur, img_noised)))
```

#A Apply the Gaussian blur filter to the noised image

The resulting denoised image is shown in the Figure 3.6. It is better to look at the images on computer, but you can see that the amount of noise is significantly reduced, but the image became blurry. While it is expected (we applied a blur filter) we would like to make image sharper if possible.

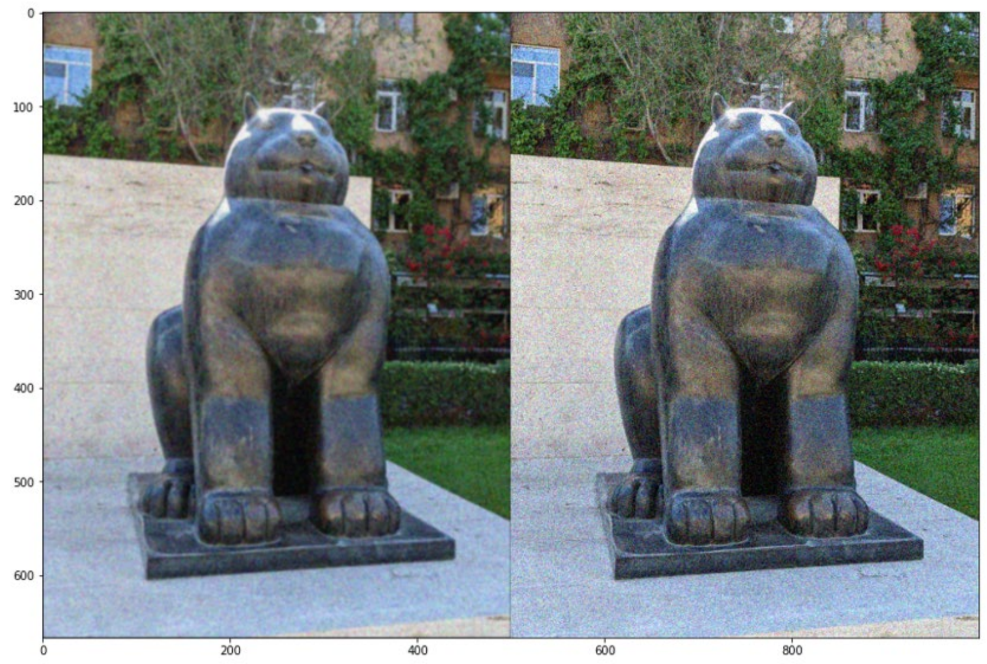


Figure 3.6 Denoised (left) and noised (right) versions of the image

Not surprisingly, there is another matrix filter for sharpening images. Its kernel contains a large positive number in the center and negative numbers in the neighborhood. The idea of this filter is to improve contrast between the central point and its neighborhood.

We normalize the kernel values by dividing each one by the sum of all values. It is performed to restrict the resulting values after filter application in the allowed range. We have a `clip()` function inside our filter application function, but that’s the last resort thing that cuts every value outside the range to its boundary. The finer normalizing approach would preserve more information in the signal.

Listing 3.8 Sharpening filter kernel.

```
>>>kernel_sharpen = np.array(    #A
>>>    [[-1, -1, -1, -1, -1],    #A
>>>     [-1, -1, -1, -1, -1],    #A
>>>     [-1, -1, 50, -1, -1],    #A
>>>     [-1, -1, -1, -1, -1],    #A
>>>     [-1, -1, -1, -1, -1]], dtype=np.float32    #A
>>>)    #A
>>>kernel_sharpen /= np.sum(kernel_sharpen)    #B
>>>kernel_sharpen

array([[-0.03846154, -0.03846154, -0.03846154, -0.03846154, -0.03846154],
       [-0.03846154, -0.03846154, -0.03846154, -0.03846154, -0.03846154],
       [-0.03846154, -0.03846154,  1.9230769 , -0.03846154, -0.03846154],
       [-0.03846154, -0.03846154, -0.03846154, -0.03846154, -0.03846154],
       [-0.03846154, -0.03846154, -0.03846154, -0.03846154, -0.03846154]],
      dtype=float32)
```

#A Creating a kernel with large positive value in the middle and small negative values around
#B Normalizing the kernel

Let's apply our sharpening filter to the blurred image:

Listing 3.9 Filtering the image from noise.

```
>>>img_restored = color_convolution(img_blur, kernel_sharpen)    #A
>>>plt.figure(figsize = (12,20))
>>>plt.imshow(np.vstack(    #B
>>>    (np.hstack((img, img_noised)),    #B
>>>    np.hstack((img_restored, img_blur)))    #B
>>>))    #B
```

#A Applying sharpening filter to the blurred image
#B Displaying four images together (clockwise): the original photo, noised version, blurred image, sharpened or restored image

Here in Figure 3.7 four different images are shown (in the clockwise direction): the original image at the top left, a noisy version in the top right, a denoised blurry version at bottom right, and finally a sharpened version at the bottom left. We restored some sharpness in the image while successfully removing some original noise!

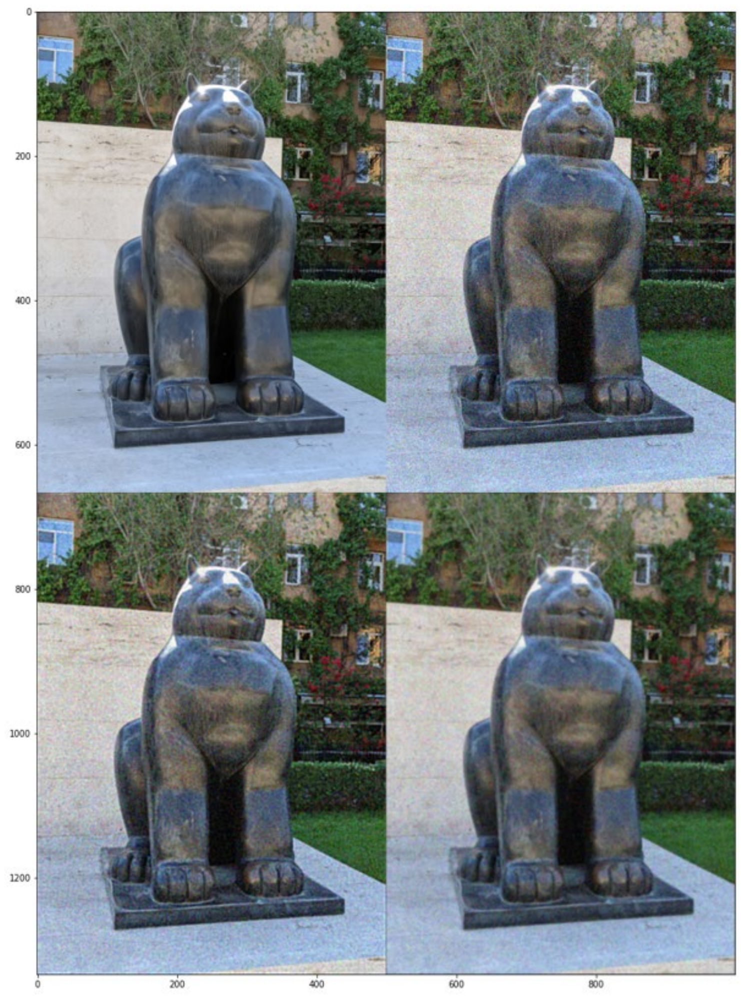


Figure 3.7 Original image (top left), noised (top right), denoised but blurred (bottom right), and sharpened (bottom left) versions of the image

The final step is to save the resulting image:

Listing 3.10 Saving an image from a NumPy array.

```
>>>image_modified = img_as_ubyte(img_restored)    #A
>>>imsave('The_Cat_modified.jpg', arr=image_modified)    #B
```

#A Converting float32 values back to 8-bit integers
#B Saving the array as a JPEG image

There are many other interesting filters you can try, for example emboss, edge detection or your custom filter. I put some of them into the corresponding Colab notebook. You can also combine several filter kernels into a single kernel. But we will stop here for now and review what we did.

We started with image loading and learned how to implement basic image processing operations like a crop of flip with slicing. Then we made a noisy version of the image and learned about matrix filters. With matrix filters, we did some noise filtering and image sharpening and might do a lot more.

We implemented everything with NumPy, and now it's time to look at what is changed with JAX.

3.2 Tensors in JAX

Now, we will rewrite our image processing program to work on top of JAX instead of NumPy.

I use Google Colab notebook with GPU runtime. It is the easiest way to start using JAX. Colab has JAX preinstalled already, and JAX can use hardware acceleration. Later in the chapter, we will also use Google TPU with JAX. So, I'd recommend to run this code in the Colab notebooks.

You may run the code on a different system, so you might need to install JAX manually. In order to do it check the installation guide in the official documentation (https://github.com/google/jax#installation) and choose the option that is most suitable for you.

3.2.1 Switching to JAX NumPy-like API

The beautiful thing is that you can replace a couple of import statements and all the rest of the code will work with JAX! Try it by yourself!

Listing 3.11 Replacing imports from NumPy to JAX.

```
## NumPy
#import numpy as np    #A
#from scipy.signal import convolve2d    #A

## JAX
import jax.numpy as np    #B
from jax.scipy.signal import convolve2d    #B
```

#A NumPy and SciPy imports
#B JAX imports

JAX has a NumPy-like API imported from the `jax.numpy` module. There are also some higher-level functions from SciPy reimplemented in JAX. This `jax.scipy` module is not as rich as the full SciPy library, yet the function we used (the `convolve2d()` function) is present there.

Sometimes you find there is no corresponding function in JAX. For example, we might use the `gaussian_filter()` function from `scipy.ndimage` for Gaussian filtering. There is no such function in `jax.scipy.ndimage`.

In such a case, you might still use the NumPy function with JAX and have two imports, one from NumPy and another for the JAX NumPy interface. It is usually done this way:

Listing 3.12 Using both NumPy and JAX.

```
## NumPy
import numpy as np    #A

## JAX
import jax.numpy as jnp    #A
```

#A We use different names for libraries to distinguish them and choose which one we use

Then, you use a function from NumPy with the prefix `np`, and a function from JAX with the prefix `jnp`. It might prevent you from using some JAX features on NumPy functions, either because they are implemented in C++, and Python only provides bindings, or because they are not functionally pure.

Another option is that you might implement new functions on your own, as we did with Gaussian filtering.

In our example above, all the code worked well when we replaced NumPy import with the JAX NumPy-compatible API. The only thing you might notice is that `numpy.ndarray` type will be replaced with the `jaxlib.xla_extension.DeviceArray` in places where we create arrays, as is the case with filter kernel creation:

Listing 3.13 Matrix (or kernel) for a simple blur filter when using JAX.

```
>>>kernel_blur = np.ones((5,5))
>>>kernel_blur /= np.sum(kernel_blur)
>>>kernel_blur

DeviceArray([[0.04, 0.04, 0.04, 0.04, 0.04],    #A
             [0.04, 0.04, 0.04, 0.04, 0.04],    #A
             [0.04, 0.04, 0.04, 0.04, 0.04],    #A
             [0.04, 0.04, 0.04, 0.04, 0.04],    #A
             [0.04, 0.04, 0.04, 0.04, 0.04]], dtype=float32)    #A

>>>type(kernel_blur)

jaxlib.xla_extension.DeviceArray    #B
```

#A NumPy array is replaced with JAX DeviceArray
#B The full name of the DeviceArray type

We also see that the data type is now float32. With NumPy, it will be float64 in most cases. Everything else works as usual.

We will discuss floating-point data types later in section 3.3.2, and for now, let’s dive into what DeviceArray is.

3.2.2 What is the DeviceArray?

DeviceArray is the type for representing arrays in JAX. It can use different backends — CPU, GPU, and TPU. It is equivalent to the `numpy.ndarray` backed by a memory buffer on a single device. In general, a device is something used by JAX to perform computations.

Types of Hardware: CPU, GPU, TPU

CPU is a Central Processing Unit, a typical processor from Intel, AMD, or Apple (which now uses their own ARM processors). It is a universal general-purpose computing device, yet many new processors have special instructions to improve performance on machine learning workloads.

GPU is a Graphics Processing Unit, a special highly-parallel processor originally created for computer graphics tasks. Modern GPUs contain a lot (thousands) of simple processors (cores) and are highly parallel, which makes them very effective in running some algorithms. Matrix multiplications, the core of deep learning right now, are among these. The most famous and best supported are NVIDIA GPUs, yet AMD and Intel have their own GPUs.

TPU is a Tensor Processing Unit from Google, the most well-known example of an ASIC (Application-specific Integrated Circuit). ASIC is an integrated circuit customized for a particular use rather than intended for general-purpose use like CPU. ASICs are more specialized than GPUs because a GPU is still a massively parallel processor with thousands of computational units capable of executing many different algorithms, while an ASIC is a processor designed to be capable of doing a very small set of computations (say, only matrix multiplications). But it does so extremely well. There are many other ASICs for deep learning and AI, but for now, their support is very limited among deep learning frameworks.

In many cases, you do not need to instantiate DeviceArray objects manually (and we didn’t). You rather will create them via `jax.numpy` functions like `array()`, `linspace()`, and so on.

A noticeable difference from NumPy here is that NumPy usually accepts Python lists or tuples as inputs to its API functions (not including the `array()` constructor). JAX deliberately chooses not to accept lists or tuples as inputs to its functions because that can lead to silent performance degradation, which is hard to detect. If you want to pass a Python list to a JAX function, you have to explicitly convert it into an array.

Listing 3.14 Using Python lists or tuples in functions.

```
>>>import numpy as np
>>>import jax.numpy as jnp

>>>np.array([1, 42, 31337])    #A

array([    1,    42, 31337])

>>>jnp.array([1, 42, 31337])    #B

DeviceArray([    1,    42, 31337], dtype=int32)

>>>np.sum([1, 42, 31337])    #C

31380

>>>try:
>>>  jnp.sum([1, 42, 31337])    #D
>>>except TypeError as e:
>>>  print(e)

sum requires ndarray or scalar arguments, got <class 'list'> at position 0.

>>>jnp.sum(jnp.array([1, 42, 31337]))    #E

DeviceArray(31380, dtype=int32)
```

#A NumPy nparray can be created from a Python list
#B JAX DeviceArray can also be created from a Python list
#C NumPy sum() function can work with Python lists
#D jax.numpy sum() function does not accept Python lists
#E You have to create a DeviceArray first in order to use jax.numpy sum() function

Notice, that scalars are also packed into DeviceArray.

DeviceArray has a similar list of properties to the Numpy array. The official documentation shows the full list of methods and properties (https://jax.readthedocs.io/en/latest/jax.numpy.html#jax-devicearray).

Listing 3.15 Using standard NumPy-like properties.

```
>>>arr = jnp.array([1, 42, 31337])    #A

>>>arr.ndim

1    #B

>>>arr.shape

(3,)    #C

>>>arr.dtype

dtype('int32')    #D

>>>arr.size

3    #E

>>>arr.nbytes

12    #F
```

#A Creating array of three integer elements
#B The resulting tensor has one dimension (as arrays do)
#C Here is the shape of the tensor
#D The elements are 32-bit integers
#E The tensor has 3 elements
#F These 3 elements require 12 bytes to store (as each element is 4-byte long)

DeviceArray objects are designed to work seamlessly with Python standard library tools where appropriate. For example, when `copy.copy()` or `copy.deepcopy()` from the built-in Python `copy` module encounters a DeviceArray, it is equivalent to calling the `copy()` method, which will create a copy of the buffer on the *same* device as the original array.

DeviceArrays can also be serialized, or translated into a format that can be stored in a file, via the built-in `pickle` module. In a similar manner to `numpy.ndarray` objects, DeviceArray will be serialized via a compact bit representation. When you do the reverse operation, deserializing or unpickling, the result will be a new DeviceArray object on the *default* device. This is because unpickling may take place in a different environment with a different set of devices.

3.2.3 Device-related operations

Not surprisingly, there is a bunch of methods dedicated to tensor device placement. There might be plenty of devices available.

As I use a system with GPU, I will have an additional device available. The following examples will use both CPU and GPU devices. If your system does not have any device other than the CPU, try to use Google Colab. It provides cloud GPUs even in the free tier.

LOCAL AND GLOBAL DEVICES

First of all, there is a host. A host is the CPU that manages several devices. A single host can manage a number of devices (usually up to 8), so in order to use more devices, a multi-host configuration is needed.

JAX distinguishes between local and global devices.

A **local device** for a process is a device that the process can directly address and launch computations on. It is a device attached directly to the host (or computer) where the JAX program is running, for example, a CPU, a local GPU, or 8 TPU cores directly attached to the host. The `jax.local_devices()` function shows a process's local devices. The `jax.local_device_count()` function returns the number of devices addressable by this process. Both functions receive a parameter for the XLA backend that could be `'cpu'`, `'gpu'`, or `'tpu'`. By default, this parameter is `None` which means the default backend (GPU or TPU if available).

A **global device** is a device across all processes. As long as each process launches the computation on its local devices, a computation can span devices across processes and use collective operations via direct communication links between the devices (usually, the high-speed interconnects between Cloud TPUs or GPUs). The `jax.devices()` function shows all available global devices, and `jax.device_count()` returns the total number of devices across all processes.

We will talk about multi-host and multi-process environments later in Chapter 6. For now, let's concentrate on single-host environments only; in our case, the global device list will be equal to the local one.

Listing 3.16 Getting info about devices.

```
>>>import jax
>>>jax.devices()    #A

[GpuDevice(id=0, process_index=0)]    #B

>>>jax.local_devices()    #C

[GpuDevice(id=0, process_index=0)]

>>>jax.devices('cpu')    #D

[CpuDevice(id=0)]

>>>jax.device_count('gpu')    #E

1
```

#A Asking for default backend, which is in my case GPU
#B There is one GPU device
#C Asking only for local devices
#D Asking directly about CPU backends
#E Asking for the number of GPU devices

For non-CPU devices (here for the GPU and for TPU later in the chapter) you can see a `process_index` attribute. Each JAX process can obtain its process index with the `jax.process_index()` function. In most cases, it will be equal to 0, but for multi-process configurations, this will vary. Here, in Listing 3.16, we see that there is a local device attached to the process with index 0.

COMMITTED AND UNCOMMITTED DATA

In JAX, the computation follows data placement. There are two different placement properties:

1. The device where the data resides.
2. Whether the data is committed to the device or not. When the data is committed, it is sometimes referred to as being sticky to the device.

You can know where the data is located with the help of the `device()` method.

By default, JAX DeviceArray objects are placed **uncommitted** on the default device. The default device is the first item of the list returned by `jax.devices()` function call (`jax.devices()[0]`). It is the first GPU or TPU if it is present, otherwise, it is the CPU.

```
>>>arr = jnp.array([1, 42, 31337])
>>>arr.device()

GpuDevice(id=0, process_index=0)    #A
```

#A The original tensor is on GPU, but it's not committed

You can use `jax.default_device()` context manager to temporarily override the default device for JAX operations if you want to. You can also use the `JAX_PLATFORMS` environment variable or the `--jax_platforms` command line flag. It is also possible to set priority order when providing a list of platforms in the `JAX_PLATFORMS` variable.

Computations involving uncommitted data are performed on the default device, and the results are also uncommitted on the default device.

You can explicitly put data on a specific device using the `jax.device_put()` function call with a `device` parameter. In this case, the data becomes **committed** to the device. If you pass `None` as the `device` parameter, then the operation will behave like the identity function if the operand is already on any device, otherwise, it will transfer the data to the default device uncommitted.

```
>>>arr_cpu = jax.device_put(arr, jax.devices('cpu')[0])    #A
>>>arr_cpu.device()

CpuDevice(id=0)    #B

>>>arr.device()

GpuDevice(id=0, process_index=0)    #C
```

#A We put a copy of the tensor on the first CPU device
#B Checking that the new tensor is on CPU

#C The original tensor location is still GPU

Remember the functional nature of JAX. The `jax.device_put()` function creates a copy of your data on the specified device and returns it. The original data is unchanged.

There is a reverse operation `jax.device_get()` to transfer data from a device to the Python process on your host. The returned data is a NumPy ndarray.

```
>>>arr_host = jax.device_get(arr)    #A
>>>type(arr_host)

numpy.ndarray

>>>arr_host

array([    1,    42, 31337], dtype=int32)
```

#A Taking the original tensor from GPU to the host in the form of NumPy ndarray

Computations involving committed data are performed on the committed device, and the results will also be committed to the same device. You will get an error when you invoke an operation on arguments committed to different devices (but no error if some arguments are uncommitted).

Listing 3.17 Placing data to a device.

```
>>>arr = jnp.array([1, 42, 31337])
>>>arr.device()

GpuDevice(id=0, process_index=0)    #A

>>>arr_cpu = jax.device_put(arr, jax.devices('cpu')[0])    #B
>>>arr_cpu.device()

CpuDevice(id=0)    #C

>>>arr + arr_cpu    #D

DeviceArray([    2,    84, 62674], dtype=int32)    #D

>>>arr_gpu = jax.device_put(arr, jax.devices('gpu')[0])    #E

>>>try:
>>>  arr_gpu + arr_cpu    #F
>>>except ValueError as e:
>>>  print(e)

primitive arguments must be colocated on the same device (C++ jax.jit). Arguments are on
       devices: TFRT_CPU_0 and gpu:0
```

#A The original tensor is on GPU, but it’s not committed
#B We put a copy of the tensor on the first CPU device
#C Checking that the new tensor is on CPU
#D Calling a binary operation for uncommitted and committed tensors on different devices (OK)
#E Committing a new tensor to a different device
#F Now calling a binary operation for two tensors committed to different devices (not OK)

There is some laziness in array creation, which holds for all the constant creation operations (`zeros`, `ones`, `eye`, etc). It means that if you make a call like `jax.device_put(jnp.ones(...), jax.devices()[1])`, then this call will create an array with zeros on the device corresponding to `jax.devices()[1]`, not creating it on the default device then copying to the `jax.devices()[1]`.

You can read more about JAX lazy sublanguage for DeviceArray operations in the relevant pull request (https://github.com/google/jax/pull/1668).

Now you have an easy way to have NumPy-like computations on GPU. Just remember that this is not the full story regarding increasing performance. Chapter 5 will talk about JIT which provides more performance improvements.

3.2.4 Asynchronous dispatch

Another important concept to know is asynchronous dispatch.

JAX uses asynchronous dispatch. It means, that when an operation is executed, JAX does not wait for operation completion and returns control to the Python program. JAX returns a `DeviceArray`, which is technically a 'future'. In the 'future' a value is not available immediately and, as the name suggests, will be produced in the future on an accelerator device. Yet it already contains the shape and type, and you can also pass it to the next JAX computation.

We did not notice it before, because when you inspect the result by printing or converting to a NumPy array, JAX automatically forces Python to wait for the computation to complete. If you want to explicitly wait for the result, you can use the `block_until_ready()` method of a `DeviceArray`.

Asynchronous dispatch is very useful as it allows Python to not wait for an accelerator and run ahead, helping the Python code not be on the critical path. If the Python code enqueues computations on a device faster than they can be executed, and if it does not need to inspect values in between, then the Python program can use an accelerator in the most efficient way without the accelerator having to wait.

Not knowing about asynchronous dispatch may mislead you when doing benchmarks, and you might obtain over-optimistic results. This is the reason we used `block_until_ready()` method in Chapter 1.

Listing 3.18 Working with asynchronous dispatch.

```
>>>a = jnp.array(range(1000000)).reshape((1000,1000))
>>>a.device()

GpuDevice(id=0, process_index=0)

>>>%time x = jnp.dot(a,a)    #A

CPU times: user 757 µs, sys: 0 ns, total: 757 µs
Wall time: 770 µs

>>>%time x = jnp.dot(a,a).block_until_ready()    #B

CPU times: user 1.34 ms, sys: 65 µs, total: 1.41 ms
Wall time: 4.33 ms

>>>a_cpu = jax.device_put(a, jax.devices('cpu')[0])
>>>a_cpu.device()

CpuDevice(id=0)

>>>%time x = jnp.dot(a_cpu,a_cpu).block_until_ready()    #C

CPU times: user 272 ms, sys: 0 ns, total: 272 ms
Wall time: 150 ms
```

#A Only measuring time to dispatch the work
#B Measuring the full computation on GPU
#C Measuring the full computation on CPU

In the example above we highlighted the difference in time when measuring with and without blocking. Without blocking, we measured only the time to dispatch the work, not counting the computation itself.

Additionally, we measured the computation time on GPU and CPU. We did it by committing the data tensors to the corresponding devices. As you can see, GPU computation is 30x times faster than CPU computation (4.33 milliseconds vs. 150 milliseconds).

When reading the documentation, you may notice some functions are explicitly described as asynchronous. For example, for the `device_put()`, it is said that "This function is always asynchronous, i.e. returns immediately." (https://jax.readthedocs.io/en/latest/_autosummary/jax.device_put.html#jax.device_put)

3.2.5 Moving image processing to TPU

We already changed our code to work with JAX, and our code already works on GPU. We did nothing special to transfer computations to GPU, but the default device became the GPU device when we ran this code on a system with GPU. Magic! Everything worked out of the box.

Let's do one more thing, run our code on TPU. In Google Colab, you have to change the runtime to TPU. This is done by choosing TPU in the 'Runtime' -> 'Change runtime type' ->

'Hardware accelerator' section. After changing the runtime type, you must restart the runtime.

If you have chosen the TPU runtime, the only other thing you need is to set up the Cloud TPU *before* importing JAX. There is a special code for it in the `jax.tools.colab_tpu` module. Behind the scenes, it changes the JAX XLA backend to TPU and links the JAX backend to the TPU host (https://github.com/google/jax/blob/main/jax/tools/colab_tpu.py).

Listing 3.19 Working with Cloud TPU.

```
>>>import jax.tools.colab_tpu    #A
>>>jax.tools.colab_tpu.setup_tpu()    #A

>>>from jax.lib import xla_bridge    #B
>>>print(xla_bridge.get_backend().platform)    #B

tpu

>>>import jax
>>>jax.local_devices()

[TpuDevice(id=0, process_index=0, coords=(0,0,0), core_on_chip=0),    #C
 TpuDevice(id=1, process_index=0, coords=(0,0,0), core_on_chip=1),    #C
 TpuDevice(id=2, process_index=0, coords=(1,0,0), core_on_chip=0),    #C
 TpuDevice(id=3, process_index=0, coords=(1,0,0), core_on_chip=1),    #C
 TpuDevice(id=4, process_index=0, coords=(0,1,0), core_on_chip=0),    #C
 TpuDevice(id=5, process_index=0, coords=(0,1,0), core_on_chip=1),    #C
 TpuDevice(id=6, process_index=0, coords=(1,1,0), core_on_chip=0),    #C
 TpuDevice(id=7, process_index=0, coords=(1,1,0), core_on_chip=1)]    #C

>>>import jax.numpy as jnp
>>>a = jnp.array(range(1000000)).reshape((1000,1000))
>>>a.device()

TpuDevice(id=0, process_index=0, coords=(0,0,0), core_on_chip=0)

>>>%time x = jnp.dot(a,a).block_until_ready()    #D

CPU times: user 1.32 ms, sys: 2.63 ms, total: 3.94 ms
Wall time: 4.19 ms
```

#A We need to call it before importing JAX
#B Checking what backend does JAX use
#C We see there are 8 TPU devices available
#D Measuring the time for a large operation on TPU

There are 8 TPU devices with IDs from 0 to 7, as each Cloud TPU is a TPU board with 4 TPU chips, each containing two cores. You can see that the device where our tensor resides is now a TPU. More interesting, it is one particular TPU core out of eight cores (the first device which was the default device). The dot product calculation took place on this particular core as well.

In order to get more information on TPU, namely its version and utilization, you can use the profiler service, which has been started for all the TPU workers at 8466 port. In our case, we use old TPU v2. As of the time I'm writing this chapter, there are TPU v3 in general availability, and TPU v4 available in preview.

More on TPUs here: https://blog.inten.to/hardware-for-deep-learning-part-4-asic-96a542fe6a81

Listing 3.20 Get more information about TPU.

```
>>>import os
>>>from tensorflow.python.profiler import profiler_client    #A
>>>tpu_profile_service_address = os.environ['COLAB_TPU_ADDR'].replace('8470', '8466')
>>>print(profiler_client.monitor(tpu_profile_service_address, 100, 2))    #B

  Timestamp: 08:53:30
  TPU type: TPU v2
  Utilization of TPU Matrix Units (higher is better): 0.000%
```

#A Importing profiler from TensorFlow
#B Getting info about TPU from the profiler

This way we can move our image processing to TPU.

3.3 Differences with NumPy

You may still want to use pure NumPy for cases where you do not need any benefits from JAX, especially when running small one-off calculations. If that's not the case, and you want to use the benefits JAX provides, you may switch from NumPy to JAX. However, you might also need to do some changes to the code.

However, JAX NumPy-like API tries to follow the original NumPy API as close as possible, there are several important distinctions.

An obvious distinction we already know about, is accelerator support. Tensors can reside on different backends (CPU, GPU, TPU) and you can manage tensor device placement in a precise way. Asynchronous dispatch also belongs to this category, as it was designed to efficiently use accelerated computations.

Another difference we already mentioned is the behavior with non-array inputs we talked about in Section 3.2.2. Remember, that many JAX functions do not accept lists or tuples as their inputs to prevent performance degradations.

Other differences include immutability and some more special topics related to supported data types and type promotion. Let's discuss these topics deeper.

3.3.1 Immutability

JAX arrays are immutable. We might not notice it before, but try to change any tensor and you will see an error. Why is it?

Listing 3.21 Updating an element of a tensor.

```
>>>a_jnp = jnp.array(range(10))
>>>a_np  = np.array(range(10))
>>>a_np[5], a_jnp[5]

(5, DeviceArray(5, dtype=int32))

>>>a_np[5] = 100    #A
>>>a_np[5]

100

>>>try:
>>>  a_jnp[5] = 100    #B
>>>except TypeError as e:
>>>  print(e)

'<class 'jax._src.device_array._DeviceArray'>' object does not support item assignment. JAX
       arrays are immutable. Instead of ``x[idx] = y``, use ``x = x.at[idx].set(y)`` or
       another .at[] method:
       https://jax.readthedocs.io/en/latest/_autosummary/jax.numpy.ndarray.at.html
```

#A In-place element assignment in NumPy (OK)
#B In-place element assignment in JAX (not allowed)

Remember, JAX is designed to follow the functional programming paradigm. This is why JAX transformations are so powerful. There are beautiful books on functional programming (<link to Manning "Grokking Functional Programming" and other books>) and we do not pretend to fully cover this topic in our book. But remember the basics of functional purity, the code must not have side-effects. The code, that modifies the original arguments, is not functionally pure. The only way to create a modified tensor is to create another tensor based on the original one. You may have seen such kind of behavior in other systems and languages with functional paradigm, for example, in Spark.

This contradicts some of the NumPy programming practices. A common operation in NumPy is index update, when you change a value inside a tensor by index. Say, changing the value of the fifth element of an array. That's perfectly fine in NumPy, yet raises an error in JAX:

Thanks to JAX, the error message is very informative and suggests a solution to the problem.

INDEX UPDATE FUNCTIONALITY

For all the typical in-place expressions used to update the value of an element of a tensor, there is corresponding functionally pure equivalent, you can use in JAX. In the table 3.1 you will find the list of JAX functional operations equivalent to NumPy-style in-place expressions.

Table 3.1 Index update functionality in JAX

NumPy-style in-place expression	Equivalent JAX syntax
`x[idx] = y`	`x = x.at[idx].set(y)`
`x[idx] += y`	`x = x.at[idx].add(y)`
`x[idx] *= y`	`x = x.at[idx].multiply(y)`
`x[idx] /= y`	`x = x.at[idx].divide(y)`
`x[idx] **= y`	`x = x.at[idx].power(y)`
`x[idx] = minimum(x[idx], y)`	`x = x.at[idx].min(y)`
`x[idx] = maximum(x[idx], y)`	`x = x.at[idx].max(y)`
`ufunc.at(x, idx)`	`x = x.at[idx].apply(ufunc)`
`x = x[idx]`	`x = x.at[idx].get()`

All of these `x.at` expressions return a modified copy of `x`, not changing the original. It could be less efficient than the original in-place modifying code, but, thanks for JIT compilation, at the low level expressions like x = x.at[idx].set(y) are guaranteed to be applied in-place, making the computation efficient. So, you don't have to worry about efficiency when using index update functionality.

Now we can change the code to fix the error:

Listing 3.22 Updating an element of a tensor in JAX.

```
>>>a_jnp = a_jnp.at[5].set(100)    #A
>>>a_jnp[5]

DeviceArray(100, dtype=int32)
```

#A Creating a copy of the original tensor with specific element changed

Be careful, there are older `jax.ops.index_*` functions that are now deprecated since JAX 0.2.22.

OUT-OF-BOUNDS INDEXING

A common type of bugs is to index arrays outside of their bounds. In NumPy it was pretty straightforward to rely on Python exceptions to handle such situations. However, when the code is running on an accelerator it may be difficult or even impossible. Therefore we need some non-error behavior for out-of-bounds indexing. For index update out-of-bound operations we'd like to skip such updates, and for index retrieval out-of-bound operations the index is clamped to the bound of the array as we need something to return. It resembles handling errors with floating-point calculations with special values like NaN.

By default, JAX assumes that all indices are in-bounds. There is experimental support for giving more precise semantics to out-of-bounds indexed accesses, via the `mode` parameter of the index update functions. The possible options are:

- `"promise_in_bounds"` (default): The user promises that all indexes are in-bounds, so no additional checking is performed. In practice, it means that all out-of-bound indices in `get()` are clipped, and in `set()`, `add()` and other modifying functions are dropped.
- `"clip"`: clamps out of bounds indices into valid range.
- `"drop"`: ignores out-of-bound indices.
- `"fill"`: is an alias for "drop", but for `get()`, it will return the value specified in the optional `fill_value` argument.

Here is an example of using different options:

Listing 3.23 Out-of-bounds indexing.

```
>>>a_jnp = jnp.array(range(10))
>>>a_jnp

DeviceArray([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=int32)

>>a_jnp[42]    #A

DeviceArray(9, dtype=int32)

>>>a_jnp.at[42].get(mode='drop')    #B

DeviceArray(-2147483648, dtype=int32)

>>>a_jnp.at[42].get(mode='fill', fill_value=-1)    #C

DeviceArray(-1, dtype=int32)

>>>a_jnp = a_jnp.at[42].set(100)    #D
>>>a_jnp

DeviceArray([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=int32)

>>>a_jnp = a_jnp.at[42].set(100, mode='clip')    #E
>>>a_jnp

DeviceArray([  0,   1,   2,   3,   4,   5,   6,   7,   8, 100], dtype=int32)
```

#A Default out-of-bounds behavior (clip for 'get' operation)
#B Using drop behavior
#C Using fill behavior with a specified fill value
#D Default out-of-bounds behavior (drop for 'set' operation)
#E Using clip behavior

As you see, out-of-bounds indexing does not produce errors, it always returns some value and you can control the behavior for such cases.

That’s it for immutability for now, let’s look at another big topic with many differences comparing to NumPy.

3.3.2 Types

There are several differences with NumPy regarding data types. This includes low and high precision floating-point format support, and type promotion semantic which governs what type will have an operation result if its operands are of specific (possibly different) types.

FLOAT64 SUPPORT

While NumPy aggressively promotes operands to double precision (or float64) type, JAX by default enforces single-precision (or float32) numbers. You may be surprised when you directly create a float64 array, but JAX silently makes it float32. For many machine learning (and, especially deep learning) workloads that’s perfectly fine, for some high-precision scientific calculations, it may be not desired.

Floating-point types: float64/float32/float16/bfloat16

There are many floating-point types used in scientific computing and deep learning. The IEEE Standard for Floating-Point Arithmetic (IEEE 754) defines several formats of different precision which are widely used.

The default floating-point data type for scientific computing is a double-precision float, or **float64**, as the size of this float is 64 bits. The IEEE 754 double-precision binary floating-point format has a 1-bit sign, 11-bit exponent, and 52-bit fraction part. It has a range of ~2.23e-308 to ~1.80e308 with full 15–17 decimal digits precision.

There are higher precision types for some cases, like long double, or extended precision float, which is typically an 80-bit float on x86 platforms (however, there are many caveats). NumPy supports the `np.longdouble` type for extended precision, while JAX has no support for this type.

Deep learning applications tend to be robust to lower precision, so single-precision float or **float32** became the default data type for such applications and is the default floating-point data type in JAX. The 32-bit IEEE 754 float has a 1-bit sign, 8-bit exponent, and 23-bit fraction. Its range is ~1.18e-38 to ~3.40e38 with 6–9 significant decimal digits precision.

For many deep learning cases, even 32-bit float is too much and during the last year,s lower precision training and inference have become popular. It is usually easier to do lower precision inference than training, and some tricky schemes of mixed float16/32 precision training are present.

Among the lower-precision floating-point formats, there are two 16-bit floats: float16 and bfloat16.

The IEEE 754 half-precision float or **float16** has a 1-bit sign, 5-bit exponent, and 10-bit fraction. Its range is ~5.96e−8 to 65504 with 4 significant decimal digits.

Another 16-bit format originally developed by Google is called “Brain Floating Point Format”, or **bfloat16** for short. The original IEEE float16 was not designed with deep learning applications in mind, its dynamic range is too narrow. The bfloat16 type solves this, providing a dynamic range identical to that of float32. It has a 1-bit sign, 8-bit exponent, and 7-bit fraction. Its range is ~1.18e-38 to ~3.40e38 with 3 significant decimal digits.

The bfloat16 format, being a truncated IEEE 754 float32, allows for fast conversion to and from an IEEE 754 float32. In conversion to the bfloat16 format, the exponent bits are preserved while the significand field can be reduced by truncation.

There are some other special formats, you can read more on them in my article here: https://moocaholic.medium.com/fp64-fp32-fp16-bfloat16-tf32-and-other-members-of-the-zoo-a1ca7897d407

In order to force float64 computations, you need to set the `jax_enable_x64` configuration variable at startup.

However, 64-bit data types are not supported on every backend. For example, TPU does not support it.

Listing 3.24 Forcing float64 computations.

```
# this only works on startup!
>>>from jax.config import config
>>>config.update("jax_enable_x64", True)    #A
>>>import jax.numpy as jnp

# this will not work on TPU backend. Try using CPU or GPU.
>>>x = jnp.array(range(10), dtype=jnp.float64)    #B
>>>x.dtype

dtype('float64')    #C
```

#A We need to enable float64
#B Creating an array with float64 type
#C Should be 'float64' if we enabled it on startup. Otherwise, it will be 'float32'

You may also want to go the opposite direction, to lower precision formats.

FLOAT16/BFLOAT16 SUPPORT

In deep learning there is a tendency of using lower precision formats, most frequently, half-precision or float16, or a more special bfloat16 which is not supported in vanilla NumPy.

With JAX you can easily switch into using these lower-precision 16-bit types:

Listing 3.25 Using 16-bit floating-point types.

```
>>>xb16 = jnp.array(range(10), dtype=jnp.bfloat16)
>>>xb16.dtype

dtype(bfloat16)    #A

>>>xb16.nbytes
20

>>>x16 = jnp.array(range(10), dtype=jnp.float16)
>>>x16.dtype

dtype('float16')    #B
```

#A Using bfloat16 type
#B Using float16 type

Again, there may be limitations on specific backends.

TYPE PROMOTION SEMANTIC

For binary operations, JAX's type promotion rules differ somewhat from those used by NumPy.

Obviously, it is different for the bfloat16 type, as this type is not supported by NumPy. Yet, it also differs in some other cases.

Interestingly, when you add two 16-bit floats, one being an ordinary float16 and another one being bfloat16, you get float32 type.

Listing 3.26 Type promotion semantic.

```
>>>xb16+x16    #A

DeviceArray([ 0.,  2.,  4.,  6.,  8., 10., 12., 14., 16., 18.], dtype=float32)

>>>xb16+xb16    #B

DeviceArray([0, 2, 4, 6, 8, 10, 12, 14, 16, 18], dtype=bfloat16)
```

#A Summing bfloat16 with float16 values
#B Summing two bfloat16 values

The table highlighting differences between NumPy and JAX NumPy type promotion semantic for binary operations is located here: https://jax.readthedocs.io/en/latest/type_promotion.html

There is also a more thorough comparison between NumPy/TensorFlow/PyTorch/JAX NumPy/JAX lax (see the next section about lax): https://jax.readthedocs.io/en/latest/design_notes/type_promotion.html#appendix-example-type-promotion-tables

SPECIAL TENSOR TYPES

We dealt with so-called *dense* tensors up to now. Dense tensors explicitly contain all the values. Yet there are many cases where you have *sparse* data, which means that there are many zero-valued elements and some (usually orders of magnitude less) non-zero ones.

Sparse tensors explicitly contain only non-zero values and may save a lot of memory, but in order to efficiently use them, you need special linear algebra routines with sparsity support. JAX has an experimental API related to sparse tensors and sparse matrix operations located in the `jax.experimental.sparse` module. Things will probably change when the book will be out, and we are not going to discuss this experimental module in the book.

Another use case is a collection of tensors with different shapes, say, speech recordings with different lengths. In order to make a batch with such tensors, there are special data structures in modern frameworks, for example, the *ragged tensor* in TensorFlow.

JAX has no special structure like ragged tensor, yet here you are not prohibited from working with such data. Auto batching with `vmap()` can help in many ways, and we will see examples in Chapter 6.

Another big difference between JAX and the original NumPy is that in JAX there is another lower-level API we are going to discuss.

3.4 High-level and low-level interfaces: jax.numpy and jax.lax

We got familiar with the `jax.numpy` API. It is designed to provide a familiar interface for those who know NumPy. For our image processing example, the NumPy API was enough, and we didn't need any other API.

However, you should know, there is another `jax.lax` lower-level API that underpins libraries such as `jax.numpy`. The `jax.numpy` API is a higher-level wrapper with all the operations expressed in terms of `jax.lax` primitives. Many of the `jax.lax` primitives themselves are thin wrappers around equivalent XLA operations (https://www.tensorflow.org/xla/operation_semantics).

In Figure 3.8, the diagram of the JAX API layers is shown.

The higher-level `jax.numpy` API is more stable and less likely to change than the `jax.lax` API. The JAX authors recommend using libraries such as `jax.numpy` instead of `jax.lax` directly, when possible.

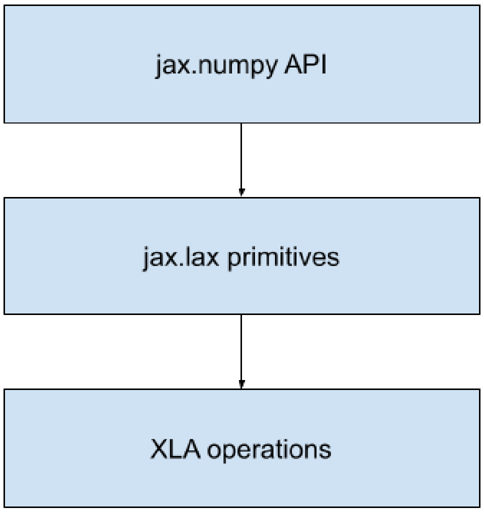


Figure 3.8 Diagram of the JAX API layers

The `jax.lax` provides a vast set of mathematical operators. Some of them are more general and provide features absent in `jax.numpy`. For example, when you do a 1D convolution in `jax.numpy`, you use the `jnp.convolve` function (https://jax.readthedocs.io/en/latest/_autosummary/jax.numpy.convolve.html#jax.numpy.convolve). This function uses a more general `conv_general_dilated` function from `jax.lax` (https://jax.readthedocs.io/en/latest/_autosummary/jax.lax.conv_general_dilated.html).

There are also other groups of operators: control flow operators, custom gradient operators, operators for parallelism support, and linear algebra operators. The full list of `jax.lax` operators is available at https://jax.readthedocs.io/en/latest/jax.lax.html#module-jax.lax.

We will talk about some gradient operators in Chapter 4, and about parallelism support in Chapter 6. Now we briefly describe control flow primitives.

CONTROL FLOW PRIMITIVES

The set of structured control flow primitives in jax.lax includes lax.cond to conditionally apply one of two functions (analog of an if statement), lax.switch for many branches, lax.while_loop and lax.fori_loop for repeatedly calling a function in a loop, lax.scan and lax.associative_scan for scanning a function over an array while carrying along with state, and a lax.map function to apply a function over leading array axes. These primitives are JIT'able and differentiable, helping to avoid un-rolling large loops.

In JAX you can still use Python control structures and in most cases it works. Yet, this solution might be suboptimal, either because there is a more performant solution, or because they produce less efficient differentiation code.

We will use many of these and other jax.lax primitives throughout the book and will describe them in later chapters during the first use. Right now we will give you one example with lax.switch.

Let's consider the case quite opposite to image filtering. In deep learning for computer vision, you often want to make your neural network robust to different image distortions. For example, you want your neural network to be robust to noise. In order to do it, you provide noisy versions of images in the dataset, and you need to create such noisy images. You may also want to use image augmentations to effectively increase your training dataset with variations of an original image: some rotations, left-right (and sometimes up-down) flip, color distortions, and so on.

Suppose, you have a set of functions for image augmentation:

```
augmentations = [
   add_noise_func,    #A
   horizontal_flip_func,    #A
   rotate_func,    #A
   adjust_colors_func    #A
]
```

#A A list of four stub functions for image processing (it is only an example, code for these functions is not provided)

We will not provide the code for these functions, they are just for demonstration purposes. You may implement them on your own as an exercise.

In the following example in Listing 3.27 we use lax.switch for choosing a random image augmentation among multiple options. The value of the first parameter, here the augmentation_index variable, determines which particular branch among several ones will be applied. The second parameter, here the augmentations list, provides a sequence of functions (or branches) to choose from. We use a list of image augmentation functions, each one performing its own image processing, be it noise addition, making horizontal flip, rotation, and so on. If the augmentation_index variable is 0, then the first element of the augmentations list is called, if augmentation_index variable is 1, then the second element is called, and so on. The rest of the parameters, here only one called the image, are passed to the chosen function, and the value returned by the chosen function, here an augmented image, is the resulting value of the lax.switch operator.

Listing 3.27 Control flow example with lax.switch.

```
from jax import random    #A

def random_augmentation(image, augmentations, rng_key):
   '''A function that applies a random transformation to an image'''
   augmentation_index = random.randint(    #B
       key=rng_key, minval=0, maxval=len(augmentations), shape=())    #B
   augmented_image = lax.switch(augmentation_index, augmentations, image)    #C
   return augmented_image

new_image = random_augmentation(image, augmentations, random.PRNGKey(42))
```

#A Importing module for random numbers in JAX (more on it in Chapter 7)
#B Generating a random integer value to index the `augmentations` list
#C Using lax.switch() function to choose between multiple options

In Chapter 5 we will understand the reasons why the code that use `jax.lax` control flow primitives may result in more efficient computations.

TYPE PROMOTION

The `jax.lax` API is stricter than `jax.numpy`. It does not implicitly promote arguments for operations with mixed data types. With `jax.lax`, you have to do type promotion in such cases manually.

Listing 3.28 Type promotion in jax.numpy and jax.lax.

```
>>>import jax.numpy as jnp
>>>jnp.add(42, 42.0)    #A

DeviceArray(84., dtype=float32, weak_type=True)    #B

>>>from jax import lax
>>>try:
>>>   lax.add(42, 42.0)    #C
>>>except TypeError as e:
>>>  print(e)

lax.add requires arguments to have the same dtypes, got int32, float32. (Tip: jnp.add is a
       similar function that does automatic type promotion on inputs).

>>>lax.add(jnp.float32(42), 42.0)    #D

DeviceArray(84., dtype=float32)
```

#A Adding integer and floating-point value in jax.numpy (OK)
#B The result is promoted to the float32 value
#C Adding integer and floating-point value in jax.lax (not OK)
#D Manually performing type conversion to have two float32 values and adding them with jax.lax (OK)

In the example above, you see a `weak_type` property in the `DeviceArray` value. This property means there was a value with no explicitly user-specified type, such as Python scalar literals. You can read more about weakly-typed values in JAX here: https://jax.readthedocs.io/en/latest/type_promotion.html#weak-types.

We will not dive deeply into `jax.lax` API here, as most of its benefits will be clear only after Chapters 4-6. Right now it's important to highlight that the `jax.numpy` API is not the only one in JAX.

3.5 Exercises

- Implement your own digital filter of choice.
- Implement image augmentation functions we proposed in Section 3.4.
- Implement other image processing operations: crop, resize, rotate, and anything else you need.

3.6 Summary

- JAX has a NumPy-like API imported from the `jax.numpy` module that tries to follow the original NumPy API as close as possible, but some differences exist.
- `DeviceArray` is the type for representing arrays in JAX.
- You can precisely manage device placement for `DeviceArray` data, committing it to the device of choice, be it a CPU, GPU, or TPU.
- JAX uses asynchronous dispatch for computations.
- JAX arrays are immutable, so you must use the functionally pure equivalent of NumPy-style in-place expressions.
- JAX provides different modes for controlling non-error behavior for out-of-bounds indexing.
- JAX default floating-point data type is float32. You can use float64/float16/bfloat16 if you want, but some limitations on specific backends might exist.
- The lower-level `jax.lax` API is stricter and often provides more powerful features than high-level `jax.numpy` API.

4 Autodiff

This chapter covers

- **Calculating derivatives in different ways**
- **Calculating gradients of your functions with the grad() transformation**
- **Using forward and reverse modes with jvp() and vjp() transformations**

Chapter 3 showed us how to work with tensors which is essential for almost any deep learning or scientific computing application. In Chapter 2, we also trained a simple neural network for handwritten digit classification. The crucial thing for training a neural network is the ability to calculate a derivative of a loss function with respect to neural network weights.

There are several ways of getting derivatives of your functions (or differentiating them), automatic differentiation (or autodiff for short) being the main one in modern deep learning frameworks. Autodiff is one of the JAX framework pillars. It enables you to write your Python and NumPy-like code, leaving the hard and tricky part of getting derivatives to the framework. In this chapter, we will cover autodiff essentials and dive into how autodiff works in JAX.

4.1 Different ways of getting derivatives

There are many tasks where you need to get derivatives. In neural network training, you need derivatives (actually, gradients) to train a network, which means minimizing a loss function by making consecutive steps in the directions opposite to the gradient. There are simpler cases familiar from the calculus classes when you need to find the minima (or maxima) of the function. There are many other cases in physics, engineering, biology, etc.

Let's start with an easily digestible illustrative example of finding a function minima. You have a simple mathematical function, say, $f(x) = x^4+12x + 1/x$. Let's implement it as code. The code for this part is available on GitHub in the following notebook: https://github.com/che-shr-cat/JAX-in-Action/blob/main/Chapter-4/JAX_in_Action_Chapter_4_Different_ways_of_getting_derivatives.ipynb

Listing 4.1 A simple function we will use as a model.

```
def f(x):
    return x**4 + 12*x + 1/x    #A
```

#A A simple mathematical function implemented as Python code

You are searching for the local minima of the function. To do it, you need to find the points where a derivative of this function is zero, then check each of the points found to be a minimum, not a maximum, or a saddle point.

Minimum, maximum, and saddle points.

There are three types of points at which a real-valued function has a zero derivative. Let's look at figure 4.1.

Maxima (a plural of **maximum**) are the points at which a function has a value larger or equal than any neighboring point. A maximum is a **global (or absolute) maximum** if it has the largest value on a domain (a set of accepted inputs) of the function. Otherwise, it is a **local (or relative) maximum**.

Minima (a plural of **minimum**) are the points at which a function has a value smaller or equal than any neighboring point. A minimum is a **global minimum** if it has the smallest value on a function domain. Otherwise, it is a **local (or relative) minimum**.

Both minima and maxima are called **extrema** (the plural of **extremum**).

A **saddle point (or minimax point)** is a point with zero derivative but which is not a local extremum of the function. Let's visualize these types of points in a figure:

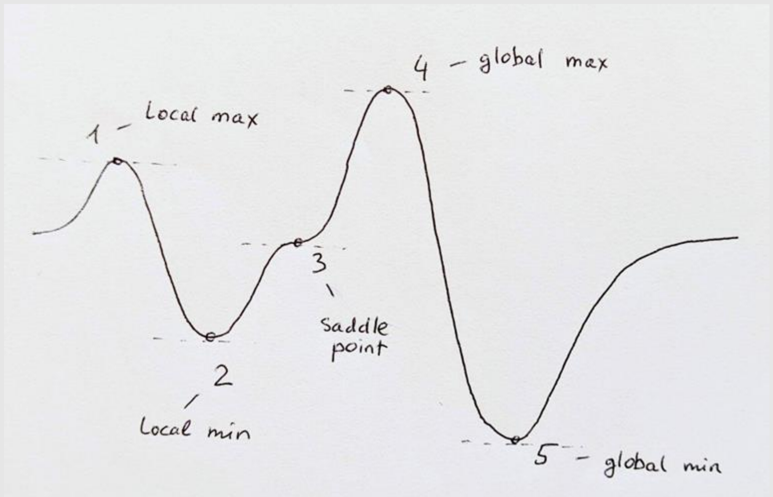


Figure 4.1. Visualizing local and global minima, maxima, and a saddle point of some function (not the same we use in the text). The dashed line displays a tangent line to the function at these specific points. Its slope is equal to the function's derivative at these points; at all these points, the derivative is zero.

Point #1 is a maximum, as it is higher than its neighboring points, but it is a local and not the global maximum of the function, as there is another maximum with a higher value, point #4. Points #2 and #5 are minima, as they are lower than their neighboring points, and point #5 is a global minimum (the lowest one) while #2 is a local one. Finally, point #3 is a saddle point, a point with a zero derivative of the function that is neither minimum nor maximum because there are both higher and lower points in its neighborhood.

So, you need to be able to calculate a function derivative. We will not go through the full procedure and only discuss the part on calculating derivatives.

There are different ways of calculating derivatives. We will discuss how they work, compare them, and see why automatic differentiation is so useful. It is essential to know about these different methods as several are sometimes negligently called automatic differentiation, while truly automatic differentiation is only one of them.

4.1.1 Manual differentiation

The good old method known by many from calculus classes in school is taking derivatives by hand.

For this particular function, the derivative will be $f'(x) = 4x^3+12 - 1/x^2$ and can be easily calculated by hand[1] if you remember differentiation rules and derivatives for some basic functions. If we need this result in code, we can implement it directly after we obtain a derivative by hand:

Listing 4.2 A closed-form expression for a derivative calculated manually.

```
def df(x):
    return 4*x**3 + 12 - 1/x**2    #A

x = 11.0

print(f(x))

>>> 14773.09090909091

print(df(x))

>>> 5335.99173553719
```

#A Manually calculated derivative of our model function

Here you have a so-called closed-form expression that can be evaluated at any point of interest. In the example above, we calculated the derivative at the point x = 11.0.

Things get harder for not-so-simple functions, for example, functions that are products of other functions or functions being a composition of other functions. Say, for a function $f(x) = (2x+7)(17x^2-3x)(x^2-1/x)(x^3+21x)/(17x-5/x^2)$, I do not even want to try doing this manually. I used WolframAlpha (https://www.wolframalpha.com/) to get the derivative of this function and got a huge expression of $(x^2 (-6615 + 47460 x + 17325 x^2 + 16620 x^3 - 122206 x^4 - 51282 x^5 - 33339 x^6 + 157352 x^7 + 58905 x^8 + 11526 x^9 + 4046 x^{10}))/(5 - 17 x^3)^2$. I'd use a couple of sheets of paper to perform all the differentiation steps, and I won't be sure about the final results. It's easy to make a mistake along the way.

For real neural networks, it's a pain to do everything with manual differentiation. It would be quite time-consuming and error-prone if you tried to implement a neural network without automatic differentiation (as you had to do if you worked on neural networks ten or more years ago). You must calculate all the layer derivatives by hand and implement them as a separate code for backpropagation calculations. With this process, you can not iterate your code quickly.

[1] Because a derivative of a sum is a sum of derivatives, and the derivative for x^4 is $4x^3$, for 12x is 12, and for 1/x is $-1/x^2$.

4.1.2 Symbolic differentiation

In the previous section, I used WolframAlpha to get the derivative of a function, and its engine does symbolic differentiation, performing all the steps you'd do by hand in an automatic way. You can program all the differentiation rules, and a computer can follow these rules much faster than a human. It also does it more reliably, without any chances of occasional errors (unless someone introduces bugs into the engine or you have buggy and unreliable hardware).

There are different options for using symbolic differentiation. In Python, you can use symbolic differentiation with a library called SymPy (https://www.sympy.org/en/index.html). It is installed by default in Google Colab, but you can install it on your particular system according to the instructions from the site (https://docs.sympy.org/latest/guides/getting_started/install.html#installation).

Listing 4.3 SymPy example of getting symbolic derivative

```
import sympy

x = 11.0

x_sym = sympy.symbols('x')    #A

f_sym = f(x_sym)    #B

df_sym = sympy.diff(f_sym)    #C

print(f_sym)    #D

>>> x**4 + 12*x + 1/x

print(df_sym)    #D

>>> 4*x**3 + 12 - 1/x**2

f = sympy.lambdify(x_sym, f_sym)    #E

print(f(x))    #F

>>> 14773.09090909091

df = sympy.lambdify(x_sym, df_sym)    #E

print(df(x))    #F

>>> 5335.99173553719
```

#A Define a variable x
#B Create a symbolic expression passing a variable into our function
#C Calculating symbolic derivative
#D Displaying our functions in symbolic form
#E Convert SymPy expressions into expressions that can be evaluated numerically
#F Numerically evaluate original function and its derivative

As with manual differentiation the result of symbolic differentiation is also a closed-form expression which is actually a separate function. You can apply this function to any point you are interested in. In the example above, we evaluate the derivative at the same point, x = 11.0.

It’s beautiful when you can do calculations in symbolic form. Symbolic differentiation is great, yet things still get hard in some cases.

First, you have to express all your calculations as a closed-form expression. It could be harder to have a closed-form expression if you implement your calculations as an algorithm, especially when using some control flow logic with if statements and for loops.

Second, symbolic expressions tend to grow bigger if you do subsequent differentiation, for example, when getting higher-order derivatives. It is called *expression swell*. Sometimes they get exponentially bigger.

4.1.3 Numerical differentiation

With manual or symbolic differentiation, you have a closed-form expression with a function of interest and obtain a closed-form expression for its derivative, which you can use to calculate the derivative at any given point. The problem is that expressing your original function in this form is not always easy. Moreover, there are many cases where you are interested in derivative values at one specific point, not at any possible point. It is also the case for deep learning, where you want gradients for the current set of weights.

A method called numerical differentiation was frequently used in science and engineering to estimate the derivative of a function.

Numerical differentiation is a way of getting an approximate value of a mathematical derivative. There are many techniques for numerical differentiation. One of the most well-known ones uses a method of finite differences based upon the limit definition of a derivative.

It estimates a derivative of the function f(x) by computing the slope of a nearby secant line through the points (x, f(x)) and (x+Δx, f(x+Δx)):

f’(x) ~= (f(x+Δx) - f(x))/Δx

This method uses two evaluations of the function at nearby points, denoted by x and (x+Δx), where Δx, or step size, is a small value, say 10^{-6}.

Listing 4.4 Finding derivative with numeric differentiation.

```
x = 11.0
dx = 1e-6    #A

df_x_numeric = (f(x+dx)-f(x))/dx    #B

print(df_x_numeric)

>>> 5335.992456821259
```

#A Choosing the step size
#B Doing numeric differentiation for our model function

The result is an approximated value of the gradient at specific point x (here, at the point x = 11.0). Because it is an approximation, the result is not exact, and we see that it is different from manual or symbolic differentiation results in the third decimal place.

Compared to manual and symbolic differentiation, which return a closed-form solution that can be used at any point, numeric differentiation does not know anything about it. It just calculates an approximated derivative at a specific point.

This method has several issues with speed, accuracy, and numerical stability.

Let’s start with speed. You need two function evaluations for a function with single scalar input, which might be costly for heavy functions. Moreover, you need two evaluations per scalar input for a function with many inputs, which leads to O(N) computational complexity. Almost all neural networks are functions of many variables (remember, they might have millions, billions, or even trillions of trainable weights that we optimize with gradient descent). So, this process scales poorly.

Accuracy is not perfect by definition, as we use approximation. Still, two types of errors lead to inaccuracy: *approximation error* and *round-off error*, both related to the step size.

You should choose the step size wisely. You want this step size to be small enough to better approximate the derivative. So, you start reducing the step size, and the *approximation error* (sometimes also called a *truncation error*, the error originating from the fact that the step size is not infinitesimal) reduces. But at some point, you become limited by the numerical precision of floating-point arithmetic, and the error tends to increase after some point if you continue reducing the step size. It is called a *round-off error*, originating from the fact that valuable low-order bits of the final answer compete for machine-word space with high-order bits of f(x+Δx) and f(x) values, as these values are stored just until they cancel each other in the subtraction at the end.

You also might get unstable results for noisy functions because small changes in the input data (weight values or input data) may result in large changes in the derivative.

In practice, you cannot use numeric differentiation as the main method of getting derivatives in gradient-based learning methods. It would be too slow. In the good old days, when you had to calculate all your layer derivatives manually, numeric differentiation was used as a sanity check by comparing with the manually calculated derivatives. If they were close enough to each other (another meta-parameter which you had to determine on your own, say, they should not differ more than 10^-8), then you probably were right with your manual differentiation and may use it further. If not, there is an error somewhere, and you have to redo your manual derivative calculation work.

4.1.4 Automatic differentiation

Finally, automatic differentiation (or autodiff for short, or even AD) came. The idea behind the autodiff is clever. A differentiable function is composed of primitives (additions, divisions, exponentiations, and so on) whose derivatives are known. During the computation of a function, autodiff tracks computations and propagates derivatives with the help of the chain rule (a rule for taking the derivative of a composite function) as the computation unfolds.

Automatic differentiation allows obtaining derivatives for very complex computer programs, including control structures, branching, loops, and recursion that might be hard to express as a closed-form expression. That’s a big deal!

Previously, if you had a Python function and wanted to calculate the derivative of this function, you had two options.

First option: you could express it as a closed-form expression and then take a manual or symbolic derivative. For many practical functions with nontrivial control flow inside, it could be time-consuming and hard or even impossible just to get a closed-form expression.

Second option: you could use a numerical derivative, but it is slow and scales poorly with the number of scalar inputs. Say, for a neural network for classifying MNIST images from Chapter 2, there are 28*28=784 scalar inputs. You need to perform two function evaluations for each one, so there are more than 1500 neural network forward propagations to estimate all the gradients. That's a lot.

Autodiff provides a new[2] alternative. It can be applied to regular code with minimal change, and you can focus on writing code for the computation you want while AD takes care of calculating derivatives.

All the major modern deep learning frameworks, including TensorFlow, PyTorch, and JAX, implement automatic differentiation. We have already used it in the Chapter 2 example, and you know that `grad()` transformation produces a function that calculates a derivative of the original function. In the next section, we will compare the JAX approach with TensorFlow and PyTorch approaches. The following example demonstrates using automatic differentiation in JAX:

Listing 4.5 Finding derivative with automatic differentiation in JAX.

```
df = jax.grad(f)    #A

print(df(x))    #B

>>> 5335.9917
```

#A Getting function derivative with automatic differentiation in JAX
#B Calculating derivative at the specific point

There are two modes for autodiff: the *forward mode*, and the *reverse mode*. We will dive deeper into it later in the chapter.

If you started your neural network journey ten or more years ago, you remember how things were done back then. You had some programming language with matrix support (and it was better to have Matlab than C++), and you implemented your neural network architecture as a sequence of basic matrix operations (similarly, you can do it right now with NumPy). Then you had to implement backpropagation (backprop for short). To do it, you had to calculate derivatives for your neural network layers by hand, implement them as another set of matrix operations, and then implement numerical differentiation only to check that your calculations were correct (you cannot use numerical differentiation instead of backprop as it is too slow, so you can use it only to check the results during development). It took a lot of time, and the process was highly error-prone. You iterated very slowly, and that limited your research speed significantly.

Then frameworks came. Some major frameworks (say, Caffe) still required manual differentiation to implement your custom layers with both forward and backward functions. Some other frameworks (say, Theano, and then TensorFlow, and PyTorch) started to use

[2] To be precise, AD is a pretty old topic starting at least in 1960s. Unfortunately, it was long unknown (and still is) for many practitioners in the deep learning field.

autodiff to take care of calculating derivatives and allow a user to focus on writing the forward computation logic only. That helped a lot. The speed of development iterations increased significantly, and (with the help of more powerful hardware) you could now test different modifications in minutes instead of hours or days. I believe these frameworks were the biggest democratizer for the deep learning field itself.

JAX continues this beautiful tradition of calculating derivatives of your functions, adding the possibility of calculating gradients even for custom Python code with control logic. It was not the first framework with such a capability (Autograd library emerged before Tensorflow, and Chainer and DyNet were among the first supporting dynamic computational graphs, then came PyTorch; TensorFlow 1 had some very limited options to do it with the Fold library, but generally, it can do it starting with version 2).

JAX gives you much more flexibility because it's not just calculating gradients for you, but it transforms functions into other functions that calculate gradients of the original function. And you may not stop here and generate a function for higher-order differentiation similarly.

We have described several approaches to differentiation and will dive deeper into how to use automatic differentiation in modern frameworks.

4.2 Calculating gradients with autodiff

Now, let's choose another practical example. We take a simple yet useful example of linear regression. Imagine we have a noisy dataset of temperature measurements you obtained from your brand new Raspberry Pi with a temperature sensor. You want to calculate a linear trend that describes your data (a more complicated function with both a trend and a periodic component would be a better fit, but I leave it to you as an exercise).

I used an algorithmic procedure to generate noisy data. You may use any data you want — any other temperature or humidity measurements, number of meteors, stock prices, or anything else.

The code for this part is available on GitHub in the following notebook: https://github.com/che-shr-cat/JAX-in-Action/blob/main/Chapter-4/JAX_in_Action_Chapter_4_Gradients_in_TensorFlow_PyTorch_JAX.ipynb.

I use the following procedure:

Listing 4.6 Generating data for our regression problem.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10*np.pi, num=1000)    #A
e = np.random.normal(scale=10.0, size=x.size)    #B
y = 65.0 + 1.8*x + 40*np.cos(x) + e    #C

plt.scatter(x, y)
```

#A Generating 1000 points in the range from 0 to 10π
#B Generating random Gaussian noise
#C Generating the data consisting of bias, linear trend, sinusoidal wave, and noise

This code generates and displays the data, you can see the result in figure 4.2:

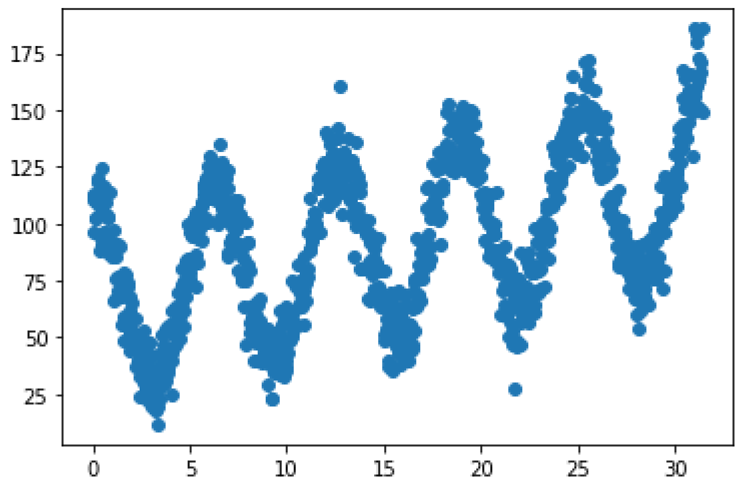


Figure 4.2. We visualized temperature measurements from our synthetic data.

The structure of the problem is very similar to the image classification problem from Chapter 2: we also have training data consisting of *x* and *y* values, a function *y=f(x)* to predict the *y* value from *x*, and a loss function to evaluate the prediction error, and a gradient procedure to adapt model weights in the direction opposite to gradient of the loss function with respect to the model weights.

Now *y* contains real-valued numbers instead of classes. The loss function that evaluates how far your predictions are from true data is now the mean squared error (or MSE) frequently used for regression problems. The prediction model is now a simple linear function[3] of the form $y = wx + b$, where *x* and *y* are the data, and *w* and *b* are the learned weights.

Finally, you have a training loop that iteratively performs gradient update steps. It is mostly the same loop we used for the image classification example from Chapter 2; this is the part we are most interested in right now.

Let's look at gradient computation differently and compare how things are done in TensorFlow/PyTorch and JAX. If you did not work with TensorFlow or PyTorch and are not interested in how they work with gradients, you can jump to the subsection on JAX.

4.2.1 Working with gradients in TensorFlow

In TensorFlow (as in many other frameworks), you need to let the framework know which tensors it needs to track computations to collect gradients. It is done by using `trainable=True` for variables in TensorFlow.

Then you perform calculations with the tensors, and the framework keeps tracking what computations are being done. In TensorFlow, you need to use *gradient tapes* to track that (you

[3] instead of a multi-layer neural network

will know more about gradient tapes and the end of the chapter when we dive into forward and reverse modes).

In the case of neural network training, you have some final value, a prediction error, or, more generally, a loss function. And finally, you want to calculate the derivatives of the loss function with respect to the parameters of the computation (your tensors of interest). In TensorFlow, you use a function `gradient()`, passing the loss function and parameters of interest. Autograd then calculates the gradients for each model parameter, returning them as a result of the `gradient()` function.

With these gradients, you can perform a gradient descent step if needed. In our examples, we implement only a single gradient step. You might want to expand the example to have a complete training loop, so treat it as an exercise.

Listing 4.7 Calculating gradients in TensorFlow.

```
import tensorflow as tf    # A

xt = tf.constant(x, dtype=tf.float32)    #B
yt = tf.constant(y, dtype=tf.float32)    #B

learning_rate = 1e-2

w = tf.Variable(1.0, trainable=True)    #C
b = tf.Variable(1.0, trainable=True)    #C

def model(x):    #D
    return w * x + b

def loss_fn(prediction, y):    #E
    return tf.reduce_mean(tf.square(prediction-y))

with tf.GradientTape() as tape:    #F
    prediction = model(x)    #F
    loss = loss_fn(prediction, y)    #F

dw, db = tape.gradient(loss, [w, b])    #G

w.assign_sub(learning_rate * dw)    #H
b.assign_sub(learning_rate * db)    #H
```

#A Importing TensorFlow
#B Converting the training data into TensorFlow tensors
#C Tensors with model weights marked with a flag to track gradients
#D The function implements a simple linear model
#E MSE loss function
#F Doing computations inside GradientTape context
#G Extracting gradients from the gradient tape
#H Making a single step of gradient update

As you see, things are not particularly hard, especially when you are already used to it. But it’s not intuitive when you first face it, and you have to remember a recipe: mark tensors with a special parameter, create a gradient tape, and call a special function to get the gradients. It’s very far from mathematical notation.

Now we look at PyTorch’s way of getting gradients.

4.2.2 Working with gradients in PyTorch

In PyTorch, you also need to mark the tensors you want to track gradients for. It is done by a special parameter called `requires_grad=True` for PyTorch tensors.

PyTorch builds a directed acyclic graph (DAG) to track operations during computations involving marked tensors.

After computations, you calculate gradients by calling a special function `backward()` on the loss tensor.

Autograd then calculates the gradients for each model parameter, storing them in a special attribute of a tensor called `grad`.

Listing 4.8 Calculating gradients in PyTorch.

```
import torch    #A

xt = torch.tensor(x)    #B
yt = torch.tensor(y)    #B

learning_rate = 1e-2

w = torch.tensor(1.0, requires_grad=True)    #C
b = torch.tensor(1.0, requires_grad=True)    #C

def model(x):    #D
    return w * x + b

def loss_fn(prediction, y):    #E
    return ((prediction-y)**2).mean()

prediction = model(xt)    #F
loss = loss_fn(prediction, yt)    #F

loss.backward()    #G

with torch.no_grad():    #H
    w -= w.grad * learning_rate    #I
    b -= b.grad * learning_rate    #I
    w.grad.zero_()    #J
    b.grad.zero_()    #J
```

#A Importing PyTorch
#B Converting the training data into PyTorch tensors
#C Tensors with model weights marked with a flag to track gradients
#D The function implements a simple linear model
#E MSE loss function
#F Doing model and loss computations
#G Calculating gradients
#H Using context manager to disable gradient calculations (we do not want parameter updates to affect gradients)
#I Making a single step of gradient update
#J Set the gradients to zero before the next gradient computation

The structure is basically the same as with TensorFlow. You still mark tensors with a special attribute, you need to know about a special scope to calculate (or not calculate) gradients (now with underlying DAG instead of a gradient tape), and you need to know special methods and attributes to get the gradients.

Because all the operations are tracked in a gradient tape or a DAG, as they are executed, Python control flow is naturally handled, and you may use control flow statements (like if or while) in your model.

4.2.3 Working with gradients in JAX

JAX does automatic differentiation in a *functional programming-friendly way*. A special transformation, `jax.grad()`, takes a numerical function written in Python and returns a new Python function that computes the gradient of the original function with respect to the first parameter of a function.

The important thing is that the `grad()` result is a function, not a gradient value. To calculate the gradient values, you have to pass the point at which you want the gradient.

Listing 4.9 Calculating gradients in JAX.

```
import jax    #A
import jax.numpy as jnp    #A

xt = jnp.array(x)    #B
yt = jnp.array(y)    #B

learning_rate = 1e-2

model_parameters = jnp.array([1., 1.])    #C

def model(theta, x):    #D
    w, b = theta
    return w * x + b

def loss_fn(model_parameters, x, y):    #E
    prediction = model(model_parameters, x)
    return jnp.mean((prediction-y)**2)

grads_fn = jax.grad(loss_fn)    #F
grads = grads_fn(model_parameters, xt, yt)    #G
model_parameters -= learning_rate * grads    #H
```

#A Importing PyTorch
#B Converting the training data to JAX DeviceArrays
#C Tensors with model weights without any special mark
#D The function implements a simple linear model
#E MSE loss function
#F Creating a function for calculating gradients
#G Calculating gradients
#H Making a single step of gradient update

This approach makes the JAX API quite different from other autodiff libraries like Tensorflow and PyTorch. In JAX, you work directly with functions, staying closer to the underlying math, and in some sense, it is more natural: your loss function is a function of model parameters and the data, so you find its gradient in the same way you would do in math.

You do not need to mark tensors in a special way, you do not need to know about underlying machinery regarding gradient tapes or DAGs, and you do not need to remember any particular way of getting gradients with special functions and attributes. You just use a function to create a function that calculates gradients. Then you directly use that second function.

As you see, in frameworks like PyTorch or Tensorflow large part of this magic is usually hidden behind high-level APIs and objects with internal states. It is also the case with optimizers we didn't use in our examples.

WHEN DIFFERENTIATING WITH RESPECT TO THE FIRST PARAMETER IS NOT ENOUGH

We mentioned that the `jax.grad()` transformation computes the gradient of the original function with respect to the first parameter of a function. What if you need to differentiate with respect to more than one parameter or not the first one? There are ways to do it!

Let's examine the case when the parameter of interest is not the first one[4].

We create a generalized function for calculating the distance between two points called the Minkowski distance. This function takes in an additional parameter that determines the order. We do not want to differentiate with respect to this parameter. We want to differentiate with respect to the parameter called *x*.

```
def dist(order, x, y):    #A
    return jnp.power(jnp.sum(jnp.abs(x-y)**order), 1.0/order)
```

#A A function with additional parameter at the first position. We want to differentiate with respect to the second parameter.

There are several ways of rewriting your original function to use the default behavior of the `grad()` function and differentiate it with respect to the first parameter. You can either rewrite your original function and move the `order` parameter to the end, or you can use an adapter function to change the order of parameters.

But suppose you do not want to do this for some reason. For such a case, there is an additional parameter in the `grad()` function called `argnums`. This parameter determines which positional argument to differentiate with respect to.

```
dist_d_x = jax.grad(dist, argnums=1)    #A

dist_d_x(1, jnp.array([1.0,1.0,1.0]), jnp.array([2.0,2.0,2.0]))

>>> DeviceArray([-1., -1., -1.], dtype=float32)
```

#A We want to differentiate with respect to the second parameter, x

The `argnums` parameter can do more.

DIFFERENTIATING WITH RESPECT TO MULTIPLE PARAMETERS

The `argnums` parameter allows differentiating with respect to more than one parameter. If we want to differentiate with respect to both *x* and *y* parameters, we can pass a tuple specifying their positions.

[4] **The code from this point up until the end of the chapter is located in the following notebook: https://github.com/che-shr-cat/JAX-in-Action/blob/main/Chapter%204/JAX_in_Action_Chapter_4_Differentiating_in_JAX.ipynb**

```
dist_d_xy = jax.grad(dist, argnums=(1,2))    #A

dist_d_xy(1, jnp.array([1.0,1.0,1.0]), jnp.array([2.0,2.0,2.0]))

>>> (DeviceArray([-1., -1., -1.], dtype=float32),
 DeviceArray([1., 1., 1.], dtype=float32))
```

#A We want to differentiate with respect to the x and y parameters

The `argnums` parameter can be an integer (if you specify a single parameter to differentiate with respect to) or a sequence of integers (if you specify multiple parameters).

When `argnums` is an integer, the gradient returned has the same shape and type as the positional argument indicated by this integer.

When `argnums` is a sequence (say, tuple), then the gradient is a tuple of values with the same shapes and types as the corresponding arguments.

In addition to manually specifying with respect to which parameters of the original function we want to differentiate and using the `argnums` parameter, there is also an option to pack multiple values into a single parameter of the function, and we have already silently used this possibility in Listing 4.9. We packed two values into a single array and passed this array to a function.

JAX allows the user to differentiate with respect to different data structures, not only arrays and tuples. You can, for example, differentiate with respect to dictionaries. There is a more general data structure called pytree, a tree-like structure built out of container-like Python objects. We will dive deeper into this topic in Chapter 8.

Here is an example of how to differentiate with respect to Python dicts. The following code is the core part of Listing 4.9 changed to work with dicts:

Listing 4.10 Differentiating with respect to dicts.

```
model_parameters = {    #A
    'w': jnp.array([1.]),
    'b': jnp.array([1.])
    }

def model(param_dict, x):
    w, b = param_dict['w'], param_dict['b']    #B
    return w * x + b

def loss_fn(model_parameters, x, y):
    prediction = model(model_parameters, x)
    return jnp.mean((prediction-y)**2)

grads_fn = jax.grad(loss_fn)
grads = grads_fn(model_parameters, xt, yt)

grads    #C

       >>> {'b': DeviceArray([-153.43951], dtype=float32),
 'w': DeviceArray([-2533.8818], dtype=float32)}
```

#A Now model parameters are a dict, not an array
#B The model function changed to work with dicts
#C Gradients are now a dict

You see that differentiating with respect to standard Python containers works perfectly.

RETURNING AUXILIARY DATA FROM A FUNCTION

The function you pass to `grad()` transformation should return a scalar, and it is only defined on scalar functions. Sometimes you want to return intermediate results, but in this case, your function returns a tuple, and `grad()` does not work.

Suppose, in our linear regression example from Listing 4.9, we wanted to return the prediction results for logging purposes. Changing the `loss_fn()` function to return some additional data will cause `grad()` transform to return an error. To fix it, we let the `grad()` transform know that the function returns some auxiliary data.

Listing 4.11 Returning auxiliary data from a function.

```
model_parameters = jnp.array([1., 1.])

def model(theta, x):
    w, b = theta
    return w * x + b

def loss_fn(model_parameters, x, y):
    prediction = model(model_parameters, x)
    return jnp.mean((prediction-y)**2), prediction    #A

grads_fn = jax.grad(loss_fn, has_aux=True)    #B
grads, preds  = grads_fn(model_parameters, xt, yt)    #C
model_parameters -= learning_rate * grads
```

#A The original loss function from Listing 4.9 also returns prediction results
#B has_aux that the function now returns a pair (out, aux)
#C Now the gradient function returns both gradients and auxiliary data, prediction in our case

The `has_aux` parameter informs the `grad()` transformation that the function returns a pair (out, auxiliary_data). It makes `grad()` ignore additional returning parameter (auxiliary_data), passing it through to the user, and differentiate the `loss_fn()` function as if only its first (out) parameter was returned. If `has_aux` is set to True, then a pair of (gradient, auxiliary_data) is returned.

For neural networks, this is useful if your model has some internal state you need to maintain, for example, running statistics in the case of BatchNorm.

OBTAINING BOTH GRADIENT AND VALUE OF THE FUNCTION

Another frequent situation you face is when you want loss function values to track learning progress. It is nice to have gradients for the gradient descent algorithm, but it is important to know how good is the current set of weights and how loss values behave.

For such a case, there is a `value_and_grad()` function, which we already used in Chapter 2.

Listing 4.12 Returning both gradients, values and auxiliary data.

```
model_parameters = jnp.array([1., 1.])

def model(theta, x):
    w, b = theta
    return w * x + b

def loss_fn(model_parameters, x, y):
    prediction = model(model_parameters, x)
    return jnp.mean((prediction-y)**2), prediction

grads_fn = jax.value_and_grad(loss_fn, has_aux=True)    #A
(loss, preds), grads  = grads_fn(model_parameters, xt, yt)    #B
model_parameters -= learning_rate * grads
```

#A Now we want both values and gradients
#B In addition to gradients and auxiliary data, we now return the loss values

In this example, we combined both values, gradients, and auxiliary data. In such a case, a tuple of ((value, auxiliary_data), gradient) is returned. It is a tuple of (value, gradient) without auxiliary data.

PER-SAMPLE GRADIENTS

In a typical machine learning setup, when you train your model with a gradient descent procedure, you usually do it with batches of data. You get a gradient for the whole batch and update the model parameters accordingly.

In most cases, that level of granularity is OK, but in some situations, you want sample-level gradients. One way of doing this is just setting the batch size to one. But it’s not computationally efficient, as modern hardware such as GPU or TPU prefers to perform massive computations to load all the underlying computing units fully.

One interesting use-case for per-sample gradients is when you want to estimate the importance of separate data samples and, say, choose those with higher gradient magnitude as more important ones to use in a special training algorithm for prioritized sampling, do hard sample mining, or to highlight data points for further analysis.

While inside the gradient computations, these numbers exist, many libraries directly accumulate gradients over a batch, so you have no easy way to get this information.

In JAX, it is pretty straightforward. Remember the Chapter 2 example of MNIST digits classification. There we wrote a function to make a single prediction, then created its batched version with `vmap()`, and finally, in the loss function, we aggregated individual losses. If we wanted these individual losses, we could easily get them.

So, the recipe for obtaining per-sample gradients is simple:

1. Create a function for making a per-sample prediction, say, `predict(x)`.
2. Create a function for calculating gradients for per-sample prediction with the `grad()` transformation, getting `grad(predict)(x)`.
3. Make a batched version of the gradient calculating function with the `vmap()` transformation, obtaining `vmap(grad(predict))(x_batch)`.

4. Optionally compile the resulting function with the `jit()` transformation to make it more efficient on the underlying hardware, finally obtaining `jit(vmap(grad(predict)))(x_batch)`.

STOPPING GRADIENTS

Sometimes, you do not want the gradients to flow through some subset of the computation graph. There might be different reasons for this. Either there are some auxiliary computations you want to perform but do not want to influence your gradient computations. For example, this might be some additional metrics or logging data (it's not functionally pure). Or you might want to update variables of your model with other losses, so while calculating one of your losses, you want to exclude variables you update with another loss[5]. You frequently need this functionality in reinforcement learning.

In the following simple example, the computations for a function `f(x,y)=x**2+y**2` form a computational graph (a directed graph resembling the order of computations) with two branches, calculating `x**2` in one branch and `y**2` in another, each one does not depend on another one.

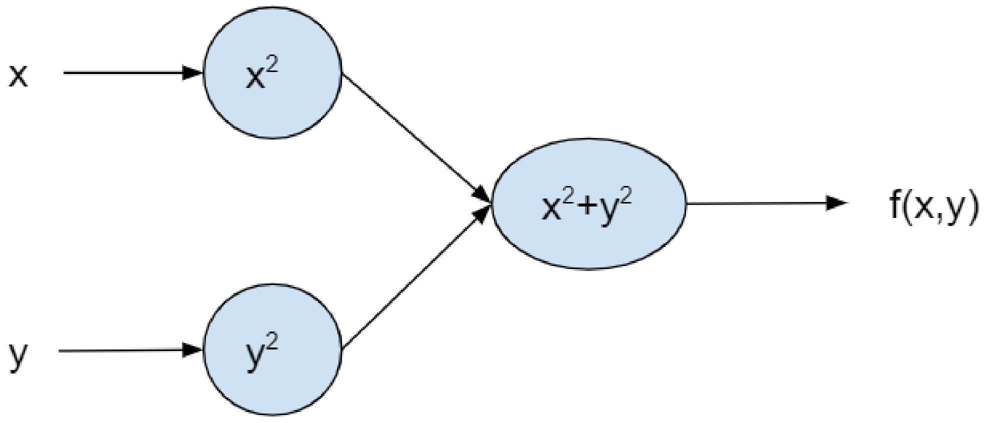


Figure 4.3. Computational graph of example f(x,y)=x^2+y^2. The branches for x^2 and y^2 do not depend on each other and in our example we want gradients to flow only through one of them.

Say you do not want the second branch to influence your gradients. There is a function for such cases. The `jax.lax.stop_gradient()` function applied for some input variable works as an identity transformation, returning the argument unchanged. At the same time, it prevents the flow of gradients during forward or reverse-mode AD. It is similar to PyTorch's `detach()` method.

[5] This might be also done from the optimizer side. There are ways to specify which parameters to update, and which to not.

Listing 4.13 Stopping gradient flow.

```
def f(x, y):
  return x**2 + jax.lax.stop_gradient(y**2)    #A

jax.grad(f, argnums=(0,1))(1.0, 1.0)    #B

>>> (DeviceArray(2., dtype=float32, weak_type=True),    #C
 DeviceArray(0., dtype=float32, weak_type=True))    #C
```

#A Using stop_gradient() to prevent gradient calculation for y
#B Calculating gradients at a particular point
#C The resulting gradients show that they were calculated only to the first variable

Here in the code you see we marked a part of the calculations related to the second function parameter with the `stop_gradient()` function. The subsequent gradient calculation gives us zero gradients with respect to the second function parameter. Without the `stop_gradient()` call, the result would be 2.0.

4.2.4 Higher-order derivatives

You can easily compute higher-order derivatives in JAX. Thanks to the functional nature of JAX, `grad()` transformation transforms a function into another function that calculates the derivative of the original function. As a `grad()` result is also a function, you can run this process several times to obtain higher-order derivatives.

Let's return to a simple function we differentiated at the beginning of the chapter. We used a function $f(x) = x^4+12x + 1/x$. We know its derivative is $f'(x) = 4x^3+12 - 1/x^2$. The second derivative is $f''(x) = 12x^2 + 2/x^3$. Its third derivative is $f'''(x) = 24x - 6/x^4$. And so on. Let's calculate these derivatives in JAX:

Listing 4.14 Finding higher-order derivatives.

```
def f(x):
    return x**4 + 12*x + 1/x

f_d1 = jax.grad(f)    #A
f_d2 = jax.grad(f_d1)    #B
f_d3 = jax.grad(f_d2)    #C

x = 11.0

print(f_d1(x))
print(f_d2(x))
print(f_d3(x))

>>> 5335.9917
>>> 1452.0015
>>> 263.9996
```

#A Taking the first derivative of the function
#B Second derivative
#C Third derivative

You can stack (or combine) these transformations together:

```
f_d3 = jax.grad(jax.grad(jax.grad(f)))
```

We calculated the derivatives at a given point (x=11.0), but remember, you have taking derivative in JAX returns you a function you can apply anywhere it is valid. In the following example, we take several consecutive derivatives of another function and draw them on a graph.

Listing 4.15 Drawing higher-order derivatives.

```
def f(x):
    return x**3 + 12*x + 7*x*jnp.sin(x)    #A

x = np.linspace(-10, 10, num=500)

fig, ax = plt.subplots(figsize=(10,10))
ax.plot(x, f(x), label = r"$y = x^3 + 12x + 7x*sin(x)$")

df = f
for d in range(3):
  df = jax.grad(df)    #B
  ax.plot(x, jax.vmap(df)(x), label=f"{['1st','2nd','3rd'][d]} derivative")    #C
  ax.legend()
```

#A Another function to differentiate
#B Consequently taking derivatives
#C Using vmap to apply a derivative function to vector of values

Here we used vmap to make derivative function be applicable to a vector of values. We can apply the original function to a vector of values thanks to NumPy broadcasting and vectorization. Functions obtained through the `grad()` transformation are only defined for scalar-output functions, so we use here `vmap()` to make this function vectorized along the array dimension. We will discuss `vmap()` more in Chapter 6.

The following figure shows the result of these calculations:

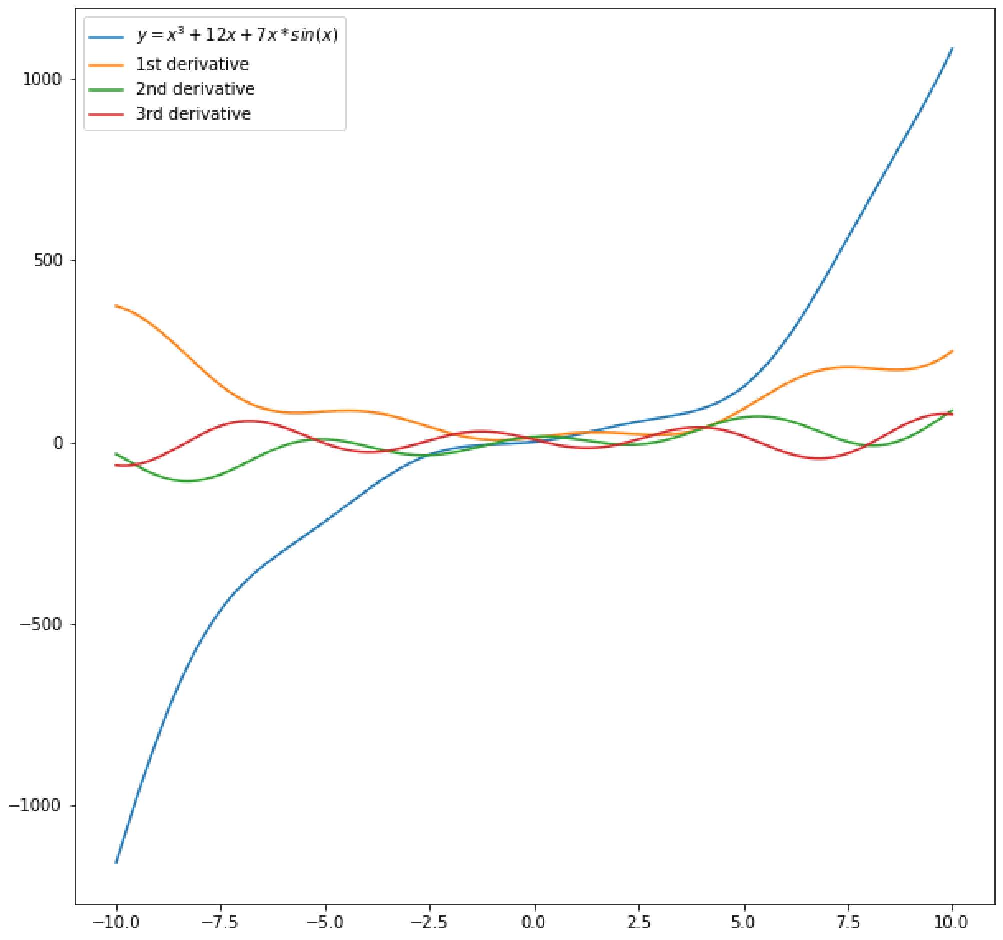


Figure 4.4. Calculating a function $f(x)=x^3 + 12x + 7x*\sin(x)$ and its three derivatives.

In JAX, you can also easily do higher-order optimization, for example, doing learned optimization or meta-learning, where you need to differentiate through gradient updates.

One interesting resource for learned optimization is Google's repository https://github.com/google/learned_optimization. Here you will find a set of interesting examples starting from an optimizer with learnable parameters.

Another interesting example is Model Agnostic Meta-Learning (MAML) (https://arxiv.org/abs/1703.03400) which tries to learn an initial set of model weights that can be quickly adapted to new tasks. One good tutorial on meta-learning can be found here https://blog.evjang.com/2019/02/maml-jax.html.

4.2.5 Multivariable case

We have started from a simple function for linear regression with a single scalar input (a single number) and a single scalar output. There was only one derivative, and we calculated it using a function obtained with the `grad()` transformation.

Then we saw a more general case of having multiple input parameters but still a single scalar output. We calculated partial derivatives with respect to multiple input variables with the help of the `argnums` parameter of the `grad()` transformation.

There is an even more general case when your function inputs and outputs are both vectors (multiple values).

JACOBIAN MATRIX

In order to calculate many partial derivatives with respect to many input variables, you might need a **Jacobian matrix**. A Jacobian matrix, or just Jacobian, is a matrix containing all partial derivatives for a vector-valued function of several variables:

$$J_{i,j} = \frac{\partial}{\partial x_j} f_i(x)$$

$$\mathbf{J} = \begin{bmatrix} \dfrac{\partial \mathbf{f}}{\partial x_1} & \cdots & \dfrac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \nabla^{\mathrm{T}} f_1 \\ \vdots \\ \nabla^{\mathrm{T}} f_m \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial f_m}{\partial x_1} & \cdots & \dfrac{\partial f_m}{\partial x_n} \end{bmatrix}$$

There are two functions to compute full Jacobian matrices, the `jacfwd()` and `jacrev()`. They compute the same values, but their implementations are different, based on forward and reverse mode AD respectively (we will talk about it later). The first one, `jacfwd()`, is more efficient for "tall" Jacobian matrices, where the number of outputs (rows in the Jacobian matrix) is significantly larger than the number of input variables (columns in the Jacobian matrix). The `jacrev()` is more efficient for "wide" Jacobian matrices where the number of input variables is significantly larger than the number of outputs. For near-square matrices `jacfwd()` is probably better (as we will see in the following example).

Let's create a simple vector-valued function, a function with two outputs and three input variables:

Listing 4.16 Calculating Jacobian of a vector-valued function.

```
def f(x):
  return [
      x[0]**2 + x[1]**2 - x[1]*x[2],    #A
      x[0]**2 - x[1]**2 + 3*x[0]*x[2]    #B
  ]

print(jax.jacrev(f)(jnp.array([3.0, 4.0, 5.0])))    #C

>>> [DeviceArray([ 6.,  3., -4.], dtype=float32), DeviceArray([21., -8.,  9.],
       dtype=float32)]

print(jax.jacfwd(f)(jnp.array([3.0, 4.0, 5.0])))    #D

>>> [DeviceArray([ 6.,  3., -4.], dtype=float32), DeviceArray([21., -8.,  9.],
       dtype=float32)]

%timeit -n 100 jax.jacrev(f)(jnp.array([3.0, 4.0, 5.0]))

>>> 29.9 ms ± 2.5 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)

%timeit -n 100 jax.jacfwd(f)(jnp.array([3.0, 4.0, 5.0]))

>>> 17.8 ms ± 336 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

#A First output of the function
#B Second output of the function
#C Using jacrev()
#D Using jacfwd()

As you see, both functions produce the same result, yet `jacfwd()` is slightly faster than `jacrev()` on this function with three inputs and two outputs. If the number of inputs were significantly larger than the number of outputs, `jacrev()` would be faster, but here the numbers are comparable, so that's the case where `jacfwd()` is faster.

By default, both functions for calculating Jacobian differentiate with respect to the first parameter of a function. In our case, we passed all the parameters in a single vector, but you might want to use a function with several separate parameters instead of this. In such a case, you can use the already familiar `argnums` parameter.

HESSIAN MATRIX

In the case of the second-order derivative of a function of multiple inputs, we obtain a **Hessian matrix** or just Hessian. Hessian is a square matrix that contains all the second derivatives:

$$H(f)(x)_{i,j} = \frac{\partial^2}{\partial x_i \, \partial x_j} f(x)$$

$$(\mathbf{H}_f)_{i,j} = \frac{\partial^2 f}{\partial x_i \, \partial x_j}$$

$$\mathbf{H}_f = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1 \, \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1 \, \partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2 \, \partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2 \, \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n \, \partial x_1} & \dfrac{\partial^2 f}{\partial x_n \, \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix},$$

You can use a `hessian()` function to compute dense Hessian matrices.

Listing 4.17 Calculating Hessian of a function.

```
def f(x):
  return x[0]**2 - x[1]**2 + 3*x[0]*x[2]    #A

jax.hessian(f)(jnp.array([3.0, 4.0, 5.0]))    #B
```

#A A function with three parameters
#B Calculating Hessian in a specific point

Hessian of a function f is a Jacobian of the gradient of f, so it can be implemented this way:

```
def hessian(f):
    return jacfwd(grad(f))
```

In JAX it is implemented the following way, which suggests that the grad() is implemented using reverse mode:

```
def hessian(f):
    return jacfwd(jacrev(f))
```

This function is a generalization of the usual definition of the Hessian that supports nested Python containers (and pytrees) as inputs and outputs.

In many cases, we do not need the full Hessian. Instead, a Hessian-vector product can be used for some calculations without materializing the whole Hessian matrix.

Now we have already met several things that require an understanding of the forward and reverse modes, so it’s time to dive deeper into the internals of autodiff and learn about it.

4.3 Forward and Reverse mode autodiff

This section is for those who want to understand the basics of machinery behind automatic differentiation. Here I will describe the forward and reverse modes of autodiff and two corresponding JAX transformations called `jvp()` and `vjp()`.

To understand it, you need some calculus knowledge. You need to know a derivative, partial derivative, and directional derivative. You need to know differentiating rules.

It is the hardest part of the book, I think. So to get it, you might have to read it several times and solve the examples independently. Do not be upset if it's hard. It's really hard. You can still use JAX without understanding autodiff internals in many cases. However, this knowledge will reward you with a better understanding of using JAX efficiently.

Reading on autodiff.

Several resources will help you along the way.

First a great short video tutorial called "What is Automatic Differentiation?" by Ari Seff (https://www.youtube.com/watch?v=wG_nF1awSSY). Ari explains these things visually and with simple words, which is very helpful.

Second, there is an in-depth article "Automatic differentiation in machine learning: a survey" by Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind (https://arxiv.org/abs/1502.05767). This article gives you much broader context and helps in understanding deeper details of autodiff.

In case you want to understand the internals of automatic differentiation even deeper, there are seminal books like "Evaluating Derivatives" by Andreas Griewank and Andrea Walther (https://epubs.siam.org/doi/book/10.1137/1.9780898717761) and "The Art of Differentiating Computer Programs" by Uwe Naumann (https://epubs.siam.org/doi/book/10.1137/1.9781611972078).

Let's return to the idea of automatic differentiation. The numerical computations we perform are ultimately compositions of a finite set of elementary operations like addition, multiplication, exponentiation, trigonometric functions, and so on. For these elementary operations, derivatives are known. For a computation consisting of such operations, we can combine derivatives of constituent operations through the chain rule for differentiation, so obtaining the derivative of the overall composition.

As Baydin et al. state in their paper, "automatic differentiation can be thought of as performing a non-standard interpretation of a computer program where this interpretation involves augmenting the standard computation with the calculation of various derivatives."

4.3.1 Evaluation trace

A computation can be represented as an **evaluation trace** of elementary operations, also called the **Wengert list**[6] (now also called a tape), with specific input values. The computation (or a function) is decomposed into a series of elementary functional steps, introducing *intermediary variables*. The Wengert list abstracts away from all control-flow considerations, which means autodiff is blind with respect to any operation, including control flow statements, which do not directly alter numeric values. The taken branch replaces all conditional statements, all the loops are unrolled, and all the function calls are inlined.

Let we have a specific real-valued function with two variables $f(x_1, x_2)=x_1*x_2 + \sin(x_1*x_2)$ to compute a derivative and an evaluation trace for that function:

Table 4.1. Evaluation trace (Wengert list) for the function $f(x_1, x_2)=x_1*x_2 + \sin(x_1*x_2)$.

Evaluation Trace
Input variables: $v_{-1} = x_1$ $v_0 = x_2$
Intermediate variables: $v_1 = v_{-1}*v_0$ $v_2 = \sin(v_1)$ $v_3 = v_1 + v_2$
Output variables: $y = v_3$

This computation can also be expressed as a computational graph:

[6] From R.E.Wengert article in 1964, https://dl.acm.org/doi/10.1145/355586.364791

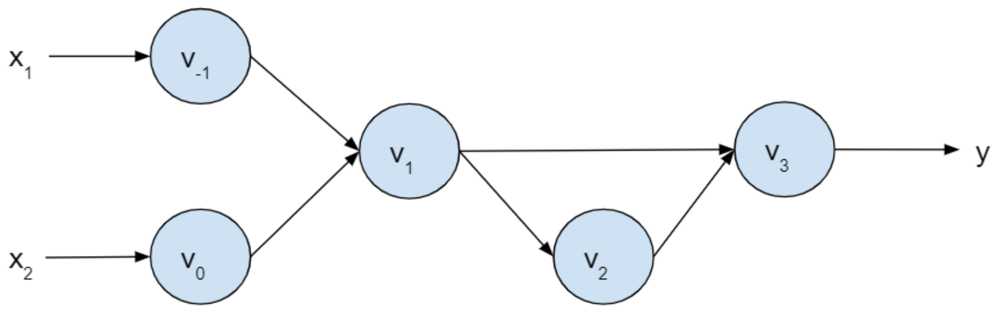


Figure 4.5. Computational graph of the example $f(x_1, x_2)=x_1*x_2 + \sin(x_1*x_2)$.

We will use this example to illustrate forward and reverse mode AD.

Suppose we want to compute a partial derivative of the function with respect to the first variable x_1 at some point, say $(x_1, x_2) = (7.0, 2.0)$.

Let's start with the forward mode autodiff.

4.3.2 Forward mode and jvp()

The forward accumulation mode autodiff (or tangent linear mode) is conceptually the simplest.

FORWARD MODE CALCULATIONS

To compute a derivative of the function with respect to the first variable x_1, we start to associate with each intermediate variable v_i its derivative dv_i/dx_1 (in the example below, denoted as just v'_i for short). Instead of a single value for each v_i, we have a tuple (v_i, v'_i). Original intermediate values are called **primals**, and derivatives are called **tangents**. It is called a dual numbers approach.

We apply the chain rule to each elementary operation in the forward tangent trace.

Table 4.2. Forward mode autodiff example with $f(x_1, x_2)=x_1*x_2 + \sin(x_1*x_2)$ evaluated at the point (7.0, 2.0) with respect to the first variable x_1.

Forward Primal Trace	Forward Tangent (Derivative) Trace
$v_{-1} = x_1 = 7.0$ (start) $v_0 = x_2 = 2.0$	$v'_{-1} = x'_1 = 1.0$ (start) $v'_0 = x'_2 = 0.0$
$v_1 = v_{-1}*v_0 = 14.0$ $v_2 = \sin(v_1) = 0.99$ $v_3 = v_1 + v_2 = 14.99$	$v'_1 = v'_{-1}*v_0 + v_{-1}*v'_0 = 1.0*2.0 + 7.0*0.0$ $v'_2 = v'_1*\cos(v_1) = 2.0*0.137$ $v'_3 = v'_1 + v'_2 = 2.0 + 0.274$
$y = v_3 = 14.99$ (end)	$y' = v'_3 = 2.274$ (end)

The single pass through the function now produces not only the function result (the original output, here in the example 14.99), but also its derivative with respect to the x_1 (here 2.274).

We can check our manual calculations using JAX:

Listing 4.18 Checking manual forward-mode calculations.

```
def f(x1,x2):    #A
    return x1*x2 + jnp.sin(x1*x2)

x = (7.0, 2.0)    #B

jax.grad(f)(*x)    #C

>>> DeviceArray(2.2734745, dtype=float32, weak_type=True)
```

#A Our function
#B Point at which we evaluate the derivative
#C Taking gradient with respect to the first parameter, x_1

The numbers almost match, manually we get a value of 2.274, JAX returns a more precise answer of 2.2734745. The manually obtained value is less precise, as I rounded some results during the calculation.

Autodiff systems use operator overloading or source code transformation approaches to implement such calculations. JAX uses operator overloading. The topic of AD systems implementation is out of the book's scope.

Now suppose the function has several outputs. We can compute partial derivatives for each output in a single forward pass. But you need to run a separate forward pass for each input variable. As you see, in the example above, we got only a derivative with respect to x_1. For x_2, we need to perform a separate forward pass.

For a general function $f: R^n \rightarrow R^m$, a forward pass for a single input variable produces one column of the corresponding function Jacobian (partial derivatives for each output with respect to the specific input variable). The full Jacobian can be computed in *n* evaluations. The previously mentioned `jacfwd()` function does exactly this.

So, it should be intuitive that the forward mode is preferred when the number of outputs is significantly larger than the number of input variables, or *m >> n*, or so-called “tall” Jacobians.

DIRECTIONAL DERIVATIVE AND JVP()

A *directional derivative* generalizes the notion of partial derivative. Partial derivatives calculate the slope in the positive direction of an axis represented by a specific variable. We used an input tangent vector (v_{-1}. v_0) = (1.0, 0.0) for the partial derivative with respect to the first variable. But we can calculate the slope in any direction. To do it, we must specify the direction. You specify direction with a vector (u_1,u_2) that points in the direction in which we want to compute the slope. The directional derivative is the same as the partial derivative when this vector points in the positive x_1 or x_2 direction and looks like (1.0, 0.0) or (0.0, 1.0).

Calculating directional derivatives is easy with AD. Just pass the direction vector as an initial value for your tangents. And that’s it. The result is a directional derivative value at a specific point.

Even more generally, we can compute a Jacobian-vector product (JVP for short) without computing the Jacobian itself in just a single forward pass. We set the input tangent vector (v_{-1}. v_0) to the vector of interest and proceed with the forward mode autodiff.

This operation is called `jvp()` for the Jacobian-vector product.

The `jvp()` takes in:

1. a function `fun` to be differentiated,
2. the `primals` values at which the Jacobian of the function `fun` should be evaluated,
3. and a vector of `tangents` for which the Jacobian-vector product should be evaluated.

The result is a `(primals_out, tangents_out)` pair, where `primals_out` is the function `fun` applied to `primals` (a value of the original function at the specific point), and `tangents_out` is the Jacobian-vector product of function evaluated at `primals` with given `tangents` (for example, it could be a directional derivative in a given direction or a partial derivative with respect to a particular variable).

We can say, that for the given function *f*, an input vector *x*, and a tangent vector *v*, `jvp()` produces both an output of the function *f(x)*, and a directional derivative *∂f(x)v*:

$(x,v) \to (f(x), \partial f(x)v)$

If you are familiar with Haskell-like type signatures[7], this function could be written as:

jvp :: (a -> b) -> a -> T a -> (b, T b)

Which means `jvp` is a function name. The function's first parameter is another function with a signature `(a -> b)` that transforms the value of type a into type b. The second parameter has type `a`; here, it is a vector of primals. The third parameter, denoted as `T a`, is a type of tangents for type a. The last type is the return value type `(b, T b)`. It consists of two items, the type for output primals `b` and the corresponding tangents type `T b`.

In the following example, we calculate JVP for the function we used in the Jacobian example in Listing 4.16.

[7] Some good basic introductions are https://en.wikibooks.org/wiki/Haskell/Type_basics#Functional_types, https://lhbg-book.link/03-html/02-type_signatures.html and http://learnyouahaskell.com/types-and-typeclasses

Listing 4.19 Calculating jvp().

```
def f2(x):    #A
  return [
      x[0]**2 + x[1]**2 - x[1]*x[2],
      x[0]**2 - x[1]**2 + 3*x[0]*x[2]
  ]

x = jnp.array([3.0, 4.0, 5.0])    #B
v = jnp.array([1.0, 1.0, 1.0])    #C

p,t = jax.jvp(f2, (x,), (v,))    #D

p    #E

>>> [DeviceArray(5., dtype=float32), DeviceArray(38., dtype=float32)]

t    #F

>>> [DeviceArray(3., dtype=float32), DeviceArray(-8., dtype=float32)]
```

#A The same function we used with Jacobian example
#B The primal vector, a value to pass into the function
#C The tangent vector, regarding which we calculate the directional derivative
#D Note how we transform our jnp.arrays into tuples with a single element
#E The value of the function at given point, f(x)
#F The directional derivative in the direction of vector v

Note that `jvp()` expects primals and tangents should be either a tuple or a list of arguments, it doesn't work with DeviceArray here, so we manually pack our jnp.arrays into a tuple here.

In this example, we calculated the value of the function f(x) and the directional derivative during the same pass. We can recover all the partial derivatives (the Jacobian) in three passes, by passing vectors [1.0, 0.0, 0.0], [0.0, 1.0, 0.0] and [0.0, 0.0, 1.0] respectively as tangents.

Listing 4.20 Recovering Jacobian columns with jvp().

```
p,t = jax.jvp(f2, (x,), (jnp.array([1.0, 0.0, 0.0]),))    #A
t

>>> [DeviceArray(6., dtype=float32), DeviceArray(21., dtype=float32)]

p,t = jax.jvp(f2, (x,), (jnp.array([0.0, 1.0, 0.0]),))    #A
t

>>> [DeviceArray(3., dtype=float32), DeviceArray(-8., dtype=float32)]

p,t = jax.jvp(f2, (x,), (jnp.array([0.0, 0.0, 1.0]),))    #A
t

>>> [DeviceArray(-4., dtype=float32), DeviceArray(9., dtype=float32)]
```

#A Passing unit vectors to recover separate Jacobian columns

Now, when we know what is JVP, we can also use the `jvp()` function to verify our manual forward mode computations:

Listing 4.21 Checking manual forward-mode calculations with JVP.

```
def f(x1,x2):    #A
  return x1*x2 + jnp.sin(x1*x2)

x = (7.0, 2.0)    #B

p,t = jax.jvp(f, x, (1.0, 0.0))    #C

p    #D

>>> DeviceArray(14.990607, dtype=float32, weak_type=True)

t    #E

>>> DeviceArray(2.2734745, dtype=float32, weak_type=True)
```

#A The same function we used in manual calculations
#B The same point where we want to evaluate derivative
#C Using JVP with the same tangent vector as in our manual calculations in Table 4.2
#D Primal values (function output)
#E Tangent values (derivative wrt x_1)

Now, when we are familiar with the JVP, our manual example directly translates to the code using the `jvp()` function. Here, our primal and tangent values are already tuples, so we didn't have to convert them and pass them directly into the `jvp()` function.

4.3.3 Reverse mode and vjp()

Forward mode is efficient when the number of inputs is much smaller than the number of outputs. But in machine learning, we typically have the opposite situation, the number of inputs is large, yet there are only a few outputs. Reverse mode autodiff solves this issue.

Reverse mode autodiff propagates derivatives back from the output and corresponds to a generalized backpropagation algorithm.

REVERSE MODE CALCULATIONS

In reverse mode autodiff, the process consists of two phases.

In the first phase, the original function is run forward. Intermediate variables (the values we get when building an evaluation trace) are populated during this process, and all the dependencies in the computation graph are recorded.

In the second phase, each intermediate variable v_i is complemented with an **adjoint** (or **cotangent**) $v'_i = dy_j/dv_i$. It's a derivative of the j-th output y_j with respect to v_i that represents the sensitivity of an output y_j with respect to changes in v_i. Derivatives are calculated backward by propagating adjoints v'_i in reverse, from outputs to inputs.

Table 4.3. Reverse mode autodiff example with $f(x_1, x_2)=x_1*x_2 + \sin(x_1*x_2)$ evaluated at the point (7.0, 2.0).

Forward Primal Trace	Reverse Adjoint (Derivative) Trace
$v_{-1} = x_1 = 7.0$ (start) $v_0 = x_2 = 2.0$	$x'_1 = v'_{-1} = 2.274$ (end) $x'_2 = v'_0 = 7.959$
$v_1 = v_{-1}*v_0 = 14.0$ $v_2 = \sin(v_1) = 0.99$ $v_3 = v_1 + v_2 = 14.99$	$v'_{-1} = v'_1\, dv_1/dv_{-1} = v'_1*v_0 = 1.137*2.0 = 2.274$ $v'_0 = v'_1\, dv_1/dv_0 = v'_1*v'_{-1} = 1.137*7.0 = 7.959$ $v'_1 = v'_1 + v'_2\, dv_2/dv_1 = v'_1 + v'_2*\cos(v_1) =$ $1.0+1.0*0.137 = 1.137$ $v'_1 = v'_3\, dv_3/dv_1 = v'_3*1 = 1.0$ $v'_2 = v'_3\, dv_3/dv_2 = v'_3*1 = 1.0$
$y = v_3 = 14.99$ (end)	$v'_3 = y' = 1.0$ (start)

In our example, after the forward pass (which is the same as in forward mode) we run the reverse pass, starting with $v'_3 = y' = 1.0$. If some variable affects the output in multiple ways, then we sum its contributions on these different paths, as in the example for v'_1. At the end of the procedure, we get all the derivatives $dy/dx_1=x'_1$ and $dy/dx_2=x'_2$ in a single backward pass.

Remember, we can calculate the derivatives with respect to both function variables using the `argnums` parameter to check out calculations:

Listing 4.22 Checking manual reverse-mode calculations.

```
def f(x1,x2):    #A
    return x1*x2 + jnp.sin(x1*x2)

x = (7.0, 2.0)    #B

jax.grad(f, argnums=(0,1))(*x)    #C

>>> (DeviceArray(2.2734745, dtype=float32, weak_type=True),
 DeviceArray(7.9571605, dtype=float32, weak_type=True))
```

#A The same function
#B The same point
#C Now we are taking gradients with respect to the both parameters, x_1 and x_2

Again, the numbers almost match, manually we get a value of 2.274 for the derivative with respect to x_1 and 7.595 with respect to x_2. JAX returns more precise values of 2.2734745 and 7.9571605.

As you see, we calculated derivatives for both input variables simultaneously in a single backward pass. So, the advantage of the reverse mode is that it is computationally cheaper than the forward mode for functions with many inputs, when n >> m. However, storage requirements are higher for reverse mode.

In an extreme case of a function with n input variables and a single output, reverse mode calculates all the derivatives in a single pass, while forward mode needs n passes.

It is typical in machine learning to have multiple parameters and a single scalar output, so the reverse mode (and backpropagation) is prevalent for such settings.

REVERSE MODE GENERALIZATION AND VJP()

In a similar way to the matrix-free computation of the Jacobian-vector product with the forward mode, the reverse mode can be used to compute the transposed Jacobian-vector product (or equivalently vector-Jacobian product, or VJP for short), initializing the reverse phase with a given adjoint (or cotangent). It can be used to build Jacobian matrices one row at a time and is efficient for so-called "wide" Jacobians.

This operation is called `vjp()` for the vector-Jacobian product.

The `vjp()` takes in:

1. a function `fun` to be differentiated,
2. and the `primals` values at which the Jacobian of the function `fun` should be evaluated.

The result is a `(primals_out, vjpfun)` pair, where `primals_out` is the function `fun` applied to primals (a value of the original function at the specific point), and `vjpfun` is a function from an adjoint (cotangent) vector with the same shape as `primals_out` to a tuple of adjoint (cotangent) vectors with the same shape as `primals`, representing the vector-Jacobian product of `fun` evaluated at `primals`.

This description is a bit hard to understand. Let's describe it another way and look at the code.

It means that for the given function *f* and an input vector *x*, the JAX `vjp()` transform produces both an output of the function *f(x)* and a function to evaluate VJP with a given adjoint (cotangent) for the backward phase.

With Haskell-like type signatures we can write `vjp()` type as:

vjp :: (a -> b) -> a -> (b, CT b -> CT a)

Which means that as with JVP you still pass the original function with a type `(a -> b)` and an input value x of the type `a`. You do not pass an adjoint value to this function. The output is also different. The first returning value is still a primal output, the result of calculating the f(x) with type `b`. The second value is a function for the backward pass that takes in an adjoint value for type `b` and returns an adjoint value for type `a`.

Now, look at the code using `vjp()` for the same function we used in Table 4.3 to verify our manual reverse mode computations.

Listing 4.23 Checking manual reverse-mode calculations with VJP.

```
def f(x1,x2):    #A
  return x1*x2 + jnp.sin(x1*x2)

x = (7.0, 2.0)    #B

p,vjp_func = jax.vjp(f, *x)    #C

p    #D

>>> DeviceArray(14.990607, dtype=float32, weak_type=True)

vjp_func(1.0)    #E

>>> (DeviceArray(2.2734745, dtype=float32, weak_type=True),
 DeviceArray(7.9571605, dtype=float32, weak_type=True))
```

#A The same function we used in manual calculations
#B The same point where we want to evaluate derivative
#C Note that here we pass original function parameters as separate parameters here
#D Primal values (function output)
#E Passing the same adjoint we used in the manual example to backward computation

We got derivatives with respect to both function parameters in a single reverse pass.

For a more complicated function with two outputs, let's use the same function we used in the Jacobian example and restore the full Jacobian.

Listing 4.24 Recovering Jacobian rows with vjp().

```
def f2(x):
  return [
      x[0]**2 + x[1]**2 - x[1]*x[2],
      x[0]**2 - x[1]**2 + 3*x[0]*x[2]
  ]

x = jnp.array([3.0, 4.0, 5.0])

p,vjp_func = jax.vjp(f2, x)

p

>>> [DeviceArray(5., dtype=float32), DeviceArray(38., dtype=float32)]

vjp_func([1.0, 0.0])    #A

>>> (DeviceArray([ 6.,  3., -4.], dtype=float32),)

vjp_func([0.0, 1.0])    #A

>>> (DeviceArray([21., -8.,  9.], dtype=float32),)
```

#A Passing unit vectors to recover separate Jacobian columns

Here we had to call the `vjp_func()` obtained from `vjp()` transformation twice, as there are two outputs of the function.

4.3.4 Going deeper

There is much more on automatic differentiation in JAX. And there are several things worth your attention if you want to go deeper.

First, there is a beautiful "Autodiff cookbook" (https://jax.readthedocs.io/en/latest/notebooks/autodiff_cookbook.html) in the original JAX documentation. If you want to know more about mathematical and implementation details on JVP and VJP, how to calculate Hessian-vector products, differentiate for complex numbers, and so on, start from this great source of information.

To define custom differentiation rules in JAX, read the "Custom derivative rules for JAX-transformable Python functions" (https://jax.readthedocs.io/en/latest/notebooks/Custom_derivative_rules_for_Python_code.html) document. Here you will know about using `jax.custom_jvp()` and `jax.custom_vjp()` to define custom differentiation rules for Python functions that are already JAX-transformable.

Another useful tutorial is "How JAX primitives work" (https://jax.readthedocs.io/en/latest/notebooks/How_JAX_primitives_work.html). This document explains the interface a JAX primitive must support to allow JAX to perform all its transformations. It can help if you want to define a new `core.Primitive` instances, along with all their transformation rules.

Last but not least, the "Autodidax: JAX core from scratch" (https://jax.readthedocs.io/en/latest/autodidax.html) tutorial explains the ideas in JAX's core system, including how autodiff works.

And do not forget to look into the Automatic differentiation section for the jax package (https://jax.readthedocs.io/en/latest/jax.html#automatic-differentiation) in the public API documentation.

4.4 Summary

- There are different ways of getting derivatives: manual differentiation, symbolic differentiation, numeric differentiation, and automatic differentiation.
- Automatic differentiation (autodiff, or simply AD) is a clever technique to compute gradients of the computations expressed as code.
- Autodiff can calculate gradients of even very complex computer programs, including control structures and branching, loops, and recursion that might be hard to express as a closed-form expression.
- In JAX, you compute gradients with the help of `grad()` transformation.
- By default, `grad()` computes derivative with respect to the function's first parameter, but you can control this behavior with `argnums` parameter.
- If you need to return additional auxiliary data from your function, you can use the `has_aux` parameter in the `grad()` and other related transformations.
- You can obtain both a gradient and a function value with the `value_and_grad()` transformation.
- To calculate higher-order derivatives, you may use subsequent `grad()` transformations.
- There are `jacfwd()` and `jacrev()` functions to calculate Jacobian matrices and the `hessian()` function to calculate Hessian matrices.

- Automatic differentiation has two modes: forward and reverse.
- Forward mode calculates gradients of all the function outputs with respect to a one function input in a single pass and is efficient when the number of outputs is significantly larger than the number of input variables.
- Reverse mode calculates the gradient of one function output with respect to all the function inputs in a single pass and is more efficient for functions with many inputs and few outputs.
- The `jvp()` function computes a Jacobian-vector product (JVP for short) without computing the Jacobian itself in a single forward pass.
- The `vjp()` function computes the transposed Jacobian-vector product (equivalently vector-Jacobian product, or VJP for short) in a single run of reverse mode autodiff.

5 Compiling your code

This chapter covers

- **Using Just-in-Time (JIT) compilation to produce performant code for CPU, GPU, or TPU**
- **Looking at JIT internals: jaxpr, the JAX intermediate language, and HLO, the High Level Operations Intermediate Representation of XLA, Google's Accelerated Linear Algebra compiler**
- **Dealing with JIT limitations**

In the previous chapter we learned about autodiff and the `grad()` transformation. In this chapter we will learn about compilation and another very useful transformation, `jit()`.

JAX uses Google's XLA compiler to compile and produce efficient code. XLA is the backend that powers machine learning frameworks, originally TensorFlow, on various devices, including CPUs, GPUs, and TPUs. JAX uses compilation under the hood for library calls, but you can also use it for just-in-time (JIT) compiling your Python functions with the `jit()` function transformation. JIT compilation optimizes the computation graph and can fuse a sequence of operations into a single efficient computation or eliminate some redundant computations. It improves performance even on the CPU.

In this chapter, we will cover the mechanics of JIT, learn to use it efficiently, and understand its limitations.

5.1 Using compilation

In Chapter 1, we compared the performance of a simple JAX function on a CPU and GPU, with and without JIT. In Chapter 2, we used JIT to compile two functions in a training loop for a simple neural network. So, you basically know what JIT does, it compiles your function for a target hardware platform and makes it faster.

Let’s start with a well-known activation function called *scaled exponential linear unit* or *SELU* (https://arxiv.org/abs/1706.02515), which we mentioned in Chapter 2:

$$\begin{split}\mathrm{Selu}(x) = scale * \begin{cases}
x, & x > 0\\
\alpha e^x - \alpha, & x \le 0
\end{cases}\end{split}$$

We implement this function with the following code, with alpha and scale constants having values from the paper:

Listing 5.1 SELU activation function.

```
def selu(x,
         alpha=1.6732632423543772848170429916717,
         scale=1.0507009873554804934193349852946):
  '''Scaled exponential linear unit activation function.'''
  return scale * jnp.where(x > 0, x, alpha * jnp.exp(x) - alpha)    #A
```

#A The core of SELU function

This function contains two branches. When the argument `x` is positive, it returns the value of `x` scaled by a factor we called `scale`. Otherwise, it is a more complicated formula of *scale*alpha*e*^{x}*-alpha*. This function is frequently used in neural networks as a nonlinear activation.

Now we will use this function to demonstrate JIT.

5.1.1 Using Just-in-Time (JIT) compilation

Suppose we have a million activations for which we have to apply this function. While in a single forward run of a typical neural network, you hardly get a million computations of the function, you still have many forward runs during training, so the example might be not too far from reality.

Let’s use JIT and compare the performance of the function with and without JIT.

USING JAX.JIT AS A TRANSFORMATION OR ANNOTATION

From chapters 1 and 2, you already know that there is a `jit()` transformation and a corresponding `@jit` annotation to compile a function of your choice.

JIT and AOT.

There are two types of compilation: **Just-in-Time (JIT)** and **Ahead-of-Time (AOT)** compilation.

JIT compilation performs compilation **during the** execution of the program at a run time. Code gets compiled when it's needed. In JAX, it happens when the code marked to be compiled is executed for the first time. It is traced using abstract values representing the input arrays, compiled, and then the actual arrays are passed to the compiled function for execution. So the first run is slower than the subsequent calls.

AOT compilation, or static compilation, converts a program in a high-level language into a lower-level language **before** the program is executed. All the code is compiled in advance, irrespective of whether it will be needed. It reduces the total amount of workload required to be done during execution and moves heavy compilation into the build stage.

JAX was originally known for its JIT compilation. However, JAX provides some options for AOT compilation now.

The `jax.jit()` transformation takes in a pure function and returns a wrapped version of the function set-up for just-in-time compilation, as shown in the following listing:

Listing 5.2 Comparing SELU performance with and without JIT.

```
x = jax.random.normal(jax.random.PRNGKey(42), (1_000_000,))    #A

selu_jit = jax.jit(selu)    #B

%timeit -n100 selu(x).block_until_ready()    #C

>>> 786 µs ± 31.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

%timeit -n100 selu_jit(x).block_until_ready()    #D

>>> The slowest run took 29.85 times longer than the fastest. This could mean that an
       intermediate result is being cached.
>>> 132 µs ± 260 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

#A Generating a million random numbers
#B Obtaining a JIT-transformed version of the function
#C Measuring speed of original (non-jitted function)
#D Measuring speed of jitted function

In the example above, you can see that the JIT-compiled version of the function is almost six times faster than the non-compiled one. At the same time, both still use GPU (in my case NVIDIA A100-SXM4-40GB).

Alternatively, you can use a `@jit` annotation before the function. In the following code we compile the same `selu()` function with the help of `@jit` annotation:`

Listing 5.3 Using JIT with an annotation.

```
@jax.jit    #A
def selu(x,
         alpha=1.6732632423543772848170429916717,
         scale=1.0507009873554804934193349852946):
  '''Scaled exponential linear unit activation function.'''
  return scale * jnp.where(x > 0, x, alpha * jnp.exp(x) - alpha)

%timeit -n100 selu(x).block_until_ready()

>>> The slowest run took 17.75 times longer than the fastest. This could mean that an
       intermediate result is being cached.
>>> 146 µs ± 250 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

#A Using annotation to JIT-compile the function

This way, you do not need a separate wrapped function, this magic is hidden from you, and you get your original function already wrapped. Its speed is in the same order as the compiled version of the function from the previous run.

The `jit()` transformation has a bunch of parameters, let's now discuss some of them.

COMPILING AND RUNNING ON SPECIFIC HARDWARE

First, there are two parameters related to the hardware you wish to use. Beware, both are experimental features and the API is likely to change.

There is the `backend` parameter which is a string representing the XLA backend: `'cpu'`, `'gpu'`, or `'tpu'`. If there is a GPU or TPU available in the system, JAX will use this backend instead of a CPU.

And there is the `device` parameter to specify the device the jitted function will run on. You may pass the specific device obtained from the `jax.devices()` call. Usually by default it is the first element of the resulting array. If you have more than one GPU or TPU in the system, this parameter allows you to precisely control where to run the computations.

In the following example, we deliberately compile the function for the CPU and GPU. Here we compile separate versions of our function for CPU and GPU, and we can see almost the tenfold difference in speed between `selu_jit_cpu()` and `selu_jit_gpu()` functions.

Listing 5.4 Controlling which backend to use.

```
def selu(x,    #A
         alpha=1.6732632423543772848170429916717,    #A
         scale=1.0507009873554804934193349852946):    #A
  '''Scaled exponential linear unit activation function.'''    #A
  return scale * jnp.where(x > 0, x, alpha * jnp.exp(x) - alpha)    #A

selu_jit_cpu = jax.jit(selu, backend='cpu')    #B
selu_jit_gpu = jax.jit(selu, backend='gpu')    #C

%timeit -n100 selu(x).block_until_ready()    #D

>>> 791 µs ± 66.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

%timeit -n100 selu_jit_cpu(x).block_until_ready()    #E

>>> 1.81 ms ± 178 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

%timeit -n100 selu_jit_gpu(x).block_until_ready()    #F

>>> The slowest run took 12.60 times longer than the fastest. This could mean that an
       intermediate result is being cached.
>>> 165 µs ± 247 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

#A Recreating the same function to not interfere with the @jit-annotated version
#B CPU-targeted version
#C GPU-targeted version
#D Calling the original, non-JIT-compiled version. This version still may use GPU/TPU if available
#E Calling the function on CPU
#F Calling the function on GPU

The `selu()` call measurement may be misleading, as JAX may still use GPU or TPU if available in the system. The only difference is the function will not be compiled with XLA. My system has a GPU available, and the tensor `x` with data resides on the GPU, so computations also happen there.

To have a correct benchmark, we have to put the data tensors on the corresponding devices with the `device_put()` method (as we did it in Chapter 3), then run the original function:

Listing 5.5 Controlling both backend and tensor device placement.

```
x_cpu = jax.device_put(x, jax.devices('cpu')[0])    #A
x_gpu = jax.device_put(x, jax.devices('gpu')[0])    #B

%timeit -n100 selu(x_cpu).block_until_ready()    #C

>>> 2.74 ms ± 95.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

%timeit -n100 selu(x_gpu).block_until_ready()    #D

>>> 872 µs ± 80.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

%timeit -n100 selu_jit_cpu(x_cpu).block_until_ready()    #E

>>> 437 µs ± 4.29 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

%timeit -n100 selu_jit_gpu(x_gpu).block_until_ready()    #F

>>> 27.1 µs ± 4.94 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

#A Placing data tensor on CPU
#B Placing data tensor on GPU
#C Measuring the non-compiled function on CPU
#D Measuring the non-compiled function on GPU
#E Measuring the jit-compiled function on CPU
#F Measuring the jit-compiled function on GPU

Here I have run these measurements on the same system I used before. You can see that the GPU-based data produces results at approximately the same time we had before for a simple non-compiled function call, while the CPU-based data produces more than three times slower results. Both jit-compiled versions are faster than the corresponding non compiled calls. For the CPU, it is 6x faster; for the GPU, it is 32x faster.

By the way, JAX (at least my current version, 0.3.17) does not prohibit you from using a GPU-compiled function with CPU-based data or vice versa.

WORKING WITH FUNCTION ARGUMENTS

You can mark some arguments as *static* or compile-time constants. It means that operations that depend only on static arguments will be constant-folded in Python during tracing (we will discuss tracing in section 5.2). Constant folding is an optimization technique that eliminates expressions that calculate a value that can already be determined before code execution. Calling the jitted function with different values for these constants will trigger recompilation. It can be useful if you need to compile a function with a condition on the value of some input but when the set of possible values is limited. We will see an example later in Section 5.2.1.

You can do it either with the `static_argnums` or `static_argnames` parameter. Both are optional. The `static_argnums` parameter is an integer or collection of integers that specify which positional arguments to treat as static. The `static_argnames` is a string or collection of strings specifying which named arguments to treat as static. No arguments are treated as static if neither `static_argnums` nor `static_argnames` are provided.

The arguments and return value of the function you want to compile should be arrays, scalars, or standard Python containers like tuple, list, or dict thereof, possibly nested. Positional arguments indicated by `static_argnums` can be anything, provided they are hashable and have an equality operation defined.

Listing 5.6 Marking an argument as static.

```
def dist(order, x, y):    #A
  print("Compiling")    #B
  return jnp.power(jnp.sum(jnp.abs(x-y)**order), 1.0/order)

dist_jit = jax.jit(dist, static_argnums=0)    #C

dist_jit(1, jnp.array([0.0, 0.0]), jnp.array([2.0, 2.0]))    #D

>>> Compiling
>>> DeviceArray(4., dtype=float32)

dist_jit(2, jnp.array([0.0, 0.0]), jnp.array([2.0, 2.0]))    #E

>>> Compiling
>>> DeviceArray(2.828427, dtype=float32)

dist_jit(1, jnp.array([10.0, 10.0]), jnp.array([2.0, 2.0]))    #F

>>> DeviceArray(16., dtype=float32)
```

#A A function of three parameters, the first one is supposed to have only a limited number of values
#B A side-effect that will be visible on each compilation
#C JIT-ing the function and declaring that the first parameter is static
#D Compile function for the given parameter value and run
#E Compile function for another parameter value and run
#F For this parameter value the function is already compiled

Here the function was compiled twice: for the value of 1 of the first parameter, and for the value of 2 of this parameter. We deliberately use a side-effect that will only be visible during the first run of the function. Why this is happening we will discuss later in section 5.2.

If you want to specify static arguments when using `jit` as a decorator, you can use Python's `functools.partial`:

Listing 5.7 Using functools.partial with jit as a decorator.

```
from functools import partial

@partial(jax.jit, static_argnums=0)    #A
def dist(order, x, y):
  return jnp.power(jnp.sum(jnp.abs(x-y)**order), 1.0/order)
```

#A Creating a partially applied jax.jit() function with an argument static_argnums=0

The `partial()` function creates a functional object, which when called will behave like the original function called with the given parameters. Here, it creates a partially applied `jax.jit()` function with the fixed `static_argnums=0` parameter. This partial function is applied to our `dist()` function as an annotation, compiling it with its first argument (the `order`) being static.

Another useful case for using static arguments is when an input to the function is not a valid JAX type, namely arrays, scalars, or standard Python containers (tuple/list/dict) thereof, possibly nested. For example, if an argument is a different function or some class, then JIT-compilation will give you an error. And because the real compilation happens during the first call of the jit-ted function, the error will appear only during the first call, not when you call `jit()`. To eliminate this error, you would need to mark this argument as static. The following code demonstrates it with a function for computing a fully-connected layer parameterised by an activation function:

Listing 5.8 Fixing 'not a valid JAX type' error with static arguments.

```
def dense_layer(x, w, b, activation_func):    #A
    return activation_func(x*w+b)

x = jnp.array([1.0, 2.0, 3.0])   #B
w = jnp.ones((3,3))   #B
b = jnp.ones(3)   #B

dense_layer_jit = jax.jit(dense_layer)    #C

dense_layer_jit(x, w, b, selu)    #D

>>> ...
>>> TypeError                                Traceback (most recent call last)
>>> <ipython-input-39-b7510037de9f> in <module>
>>> ----> 1 dense_layer_jit(x, w, b, selu)
>>>
>>> TypeError: Argument '<function selu at 0x7ff401c7f320>' of type <class 'function'> is
       not a valid JAX type.

dense_layer_jit = jax.jit(dense_layer, static_argnums=3)    #E

dense_layer_jit(x, w, b, selu)    #F

>>> DeviceArray([[2.101402, 3.152103, 4.202804],
>>>              [2.101402, 3.152103, 4.202804],
>>>              [2.101402, 3.152103, 4.202804]], dtype=float32)
```

#A A function parameterised by another function
#B Some test data
#C Creating the jit-compiled version of the function
#D The compilation fails because the fourth argument is a function, not a valid JAX type
#E Marking the fourth argument as static
#F JIT now succeeds

Here we passed an activation function as a parameter for our `dense_layer()` function. Because a function is not an allowed type for input and output values for a jit-ted function, we get an error. We get rid of this error by marking this specific parameter as a static one.

OPTIMIZATION-RELATED ARGUMENTS

There is another interesting `jit` function argument called `donate_argnums`. It marks specific positional arguments that can be "donated" to the computation. The rationale behind this is the following. Input arguments use some memory buffers to store their values. During the function execution, they might no longer be needed at some point after they were used in the computation. You can donate these memory buffers, and XLA may use them to reduce the amount of memory needed for the computation, for example, to store the result. Of course, you should not reuse the buffers you are donating. If you do it, JAX raises an error. By default, no argument buffers are donated. It does not work with named arguments. This feature is implemented only for TPU and GPU, and you can read more about it here: https://jax.readthedocs.io/en/latest/faq.html#buffer-donation.

The `keep_unused` argument (which is `False` by default) controls how JAX treats arguments it determines to be unused. By default, it drops such arguments from the resulting compiled XLA code. They will not be transferred to the device and will not be provided to the underlying executable. You can turn off this logic if you do not want to prune such parameters.

There is also an option to inline your function into enclosing jaxprs (an intermediate representation to which Python code is converted, more on jaxpr in section 5.2) removing overhead for function calls. You can use the `inline=True` parameter for doing this. By default, it is `False`, and the function call is represented as an application of the `xla_call` primitive with its own subjaxpr. With inlining there are no separate function calls, the function code is inserted in the place where it is called removing overhead from calling a function and returning values from it.

5.1.2 Pure functions

JIT compilation works with JAX-compatible functions. JAX is designed to work with functionally pure code, the code without global state and side-effects. You can still write and run impure functions, but JAX does not guarantee their correct work.

JAX transforms Python functions by first converting them into a simple intermediate language called jaxpr using a process called *tracing* (we will discuss jaxpr and tracing in section 5.2). Then the transformations work on the jaxpr representation. With compilation, the jaxpr representation is being compiled further with XLA (there are more steps at this stage, we will also discuss them in section 5.2). This high-level scheme is depicted in Figure 5.1.

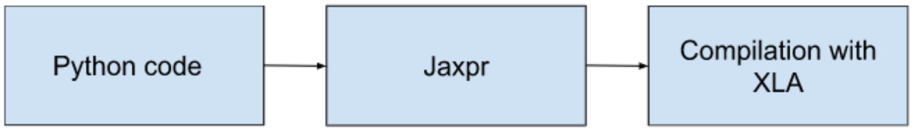


Figure 5.1. High-level scheme of how JAX compiles the code

Compilation happens during the first call of a function (so, the name *just-in-time* compilation). JAX traces the function by running it, creates a jaxpr representation, compiles it with XLA, and caches the compiled code.

There are several consequences of such a process.

First of all, the first run of a function is slower than subsequent runs. So, in order to correctly measure function performance you have to *warm-up* the function by running it a first time to make compilation really happen. Then you may measure the function performance with an already compiled code, not counting compilation overhead in your measurements.

Second, everything that happened during the first run of the function gets stored in the representation, and if the function behavior is different during other runs, you will not see it in the compiled version.

Also, side-effects are not logged in jaxpr, so you see their results only during the first run of the function. Here is an example that demonstrates what happens if a function is not pure and have a side-effect and uses a global state:

Listing 5.9 Compiling impure function.

```
global_state = 1    #A

def impure_function(x):
  print(f'Side-effect: printing x={x}')    #B
  y = x*global_state    #C
  return y

impure_function_jit = jax.jit(impure_function)    #D

impure_function_jit(10)

>>> Side-effect: printing x=Traced<ShapedArray(int32[],    #E
       weak_type=True)>with<DynamicJaxprTrace(level=0/1)>    #E
>>> DeviceArray(10, dtype=int32, weak_type=True)

impure_function_jit(10)

>>> DeviceArray(10, dtype=int32, weak_type=True)    #F

global_state = 2    #G

impure_function_jit(10)    #H

>>> DeviceArray(10, dtype=int32, weak_type=True)

impure_function(10)    #I

>>> Side-effect: printing x=10
>>> 20
```

#A Global state to be used in impure function
#B Side-effect of impure function
#C Using global state
#D Creating jit-compiled version of the function

#E Seeing side-effect during the first run
#F No side-effects during the second run
#G Changing the global state
#H Changed global state has no influence on the compiled function
#I Non-compiled function still demonstrates both side-effect and global state influence

In Listing 5.9 we have created an impure function with both a side-effect (a `print` statement) and global state (a variable `global_state`). Then we compile the function, but it actually creates a function wrapper and the real compilation will take place later, when we call the function for the first time. During the first call, JAX traces the function using abstract values representing the input arrays, and we see the side-effect. But this side-effect is not captured in the resulting jaxpr representation and the compiled code (we will look at the internals soon), and during the second call of the function we do not see the side-effect. Moreover, the global state is also not present in the compiled code, so after changing the global state the function behaves as if it still had an old value. The non-compiled original function behaves the way we expect, demonstrating both a side-effect and global state influence.

We used `print()` statements here to analyze what is happening and when, however it's just an implementation detail that the Python code is run at least once. You should not rely on it.

Take care of your code, and remember that the proper way to use JAX is to use it only on functionally pure Python functions.

To better understand why JIT works the way it works, it's worth diving into its internals. It is not strictly necessary if you want to just use JIT, but understanding internals will help you efficiently deal with JIT limitations (the topic of Section 5.3), and it may help you in debugging complicated cases, like slow compilation and inefficient code. And I believe it is valuable for those (like me) who love to look under the hood and understand how things work.

5.2 JIT internals

We have already mentioned jaxpr and the JAX compilation workflow several times. It's time to dive deeper into these details, and describe how different steps of this process work. We will first describe the "Python to jaxpr" conversion, then we will describe the "jaxpr to native code" conversion.

5.2.1 Jaxpr, an intermediate representation for JAX programs

First, we focus on the first stage of compilation, converting Python code to Jaxpr.

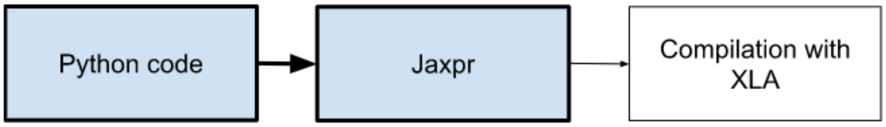


Figure 5.2. This section focuses on Python to Jaxpr conversion

JAX converts code to an intermediate representation (IR) of a computation before doing transformations and sending the code to XLA. This IR is called ***jaxpr***, short for JAX Expression. The transformations then work on the jaxpr representation.

JAXPR LANGUAGE

Jaxpr is essentially a simple functional language with very limited higher-order capabilities (a primitive is a higher-order if it is parameterized by a function). You can view the jaxpr using the `jax.make_jaxpr()` transformation:

Listing 5.10 Using jax.make_jaxpr().

```
def f1(x, y, z):    #A
  return jnp.sum(x + y * z)

x = jnp.array([1.0, 1.0, 1.0])    #B
y = jnp.ones((3,3))*2.0    #B
z = jnp.array([2.0, 1.0, 0.0]).T    #B

jax.make_jaxpr(f1)(x,y,z)    #C

>>> { lambda ; a:f32[3] b:f32[3,3] c:f32[3]. let
>>>     d:f32[1,3] = broadcast_in_dim[broadcast_dimensions=(1,) shape=(1, 3)] c
>>>     e:f32[3,3] = mul b d
>>>     f:f32[1,3] = broadcast_in_dim[broadcast_dimensions=(1,) shape=(1, 3)] a
>>>     g:f32[3,3] = add f e
>>>     h:f32[] = reduce_sum[axes=(0, 1)] g
>>>   in (h,) }
```

#A The same simple function with a few operations
#B Some test data
#C Generating jaxpr for the function

Here you see our function is converted into jaxpr. The jaxpr is printed using the following grammar:

```
jaxpr ::= { lambda Var* ; Var+.
            let Eqn*
            in  [Expr+] }
```

Jaxpr contains one or more input parameters, comprised of two lists appearing after the word `lambda` and separated by a semicolon: a list of constants (`Var*` in the grammar, which is empty in Listing 5.10) and a list of input variables to the Python function (`Var+` in the grammar, which is `a:f32[3] b:f32[3,3] c:f32[3]` in the listing). Any variables used by the function that are not input parameters will be treated as constant values and will be in the list of constants. There is a list of output atomic expressions (represented as `in [Expr+]` in the grammar, and `(h,)` in the listing, actually a tuple). And there is a list of equations (`let Eqn*`) defining intermediate variables referring to intermediate expressions. Each equation defines one or more variables as the result of applying a primitive on some atomic expressions (for example, `g:f32[3,3] = add f e` for addition). Each equation uses only input variables and intermediate variables defined by previous equations. Jaxpr is explicitly typed, and here you can see both types and shapes for each variable.

Most jaxpr primitives take just one or more `Expr` as arguments and are documented in the `jax.lax` module (https://jax.readthedocs.io/en/latest/jax.lax.html#module-jax.lax). These are the primitives like `add`, `sub`, `sin`, `mul`, `reduce_sum`. Jaxpr also includes several higher-order primitives that include sub-jaxprs. Among them are `switch` and `cond` conditionals, `while` and `fori` loops, `scan` (we mentioned them in Chapter 3), and a special `xla_call` primitive that encapsulates a sub-jaxpr along with parameters that specify the backend and the device on which the computation should run (we mentioned the `xla_call` primitive when discussed inlining in the "Optimization-related arguments" subsection of Section 5.1).

If you are interested in a thorough description of the jaxpr language, please follow the link https://jax.readthedocs.io/en/latest/jaxpr.html.

The `make_jaxpr()` transformation creates another function that takes in parameters of the original function, and returns jaxpr in the value of the `jax.core.ClosedJaxpr` type (https://jax.readthedocs.io/en/latest/_autosummary/jax.core.ClosedJaxpr.html). The `jax.core.ClosedJaxpr` type contains both jaxpr and constants. The jaxpr resides in the `jaxpr` attribute and has `jax.core.Jaxpr` type (https://jax.readthedocs.io/en/latest/_autosummary/jax.core.Jaxpr.html). Constants are in the `consts` attribute which is a list.

TRACING

JAX performs the conversion to jaxpr using tracing. During tracing, JAX wraps each argument by a special tracer object. It is an object of the `jax.core.Tracer` class, used as a substitute for a JAX `DeviceArray` in order to determine the sequence of operations performed by a Python function.

The tracer records all JAX operations performed on it during the function call. Then, JAX reconstructs the function using the tracer records. The output of that reconstruction is the jaxpr.

The Python side-effects still happen during the tracing, but the tracers do not record them and they do not appear in the jaxpr.

Listing 5.11 Looking at jaxpr for a function with side-effects.

```
x = jnp.array([1.0, 1.0, 1.0])
y = jnp.ones((3,3))*2.0
z = jnp.array([2.0, 1.0, 0.0]).T

def f2(x, y):
  print(f'x={x}, y={y}, z={z}')    #A
  return jnp.sum(x + y * z)    #B

f2_jaxpr = jax.make_jaxpr(f2)(x,y)    #C

>>> x=Traced<ShapedArray(float32[3])>with<DynamicJaxprTrace(level=1/0)>,
       y=Traced<ShapedArray(float32[3,3])>with<DynamicJaxprTrace(level=1/0)>, z=[2. 1. 0.]
       #D

f2_jaxpr.jaxpr    #E

>>> { lambda a:f32[3]; b:f32[3] c:f32[3,3]. let
>>>     d:f32[1,3] = broadcast_in_dim[broadcast_dimensions=(1,) shape=(1, 3)] a
>>>     e:f32[3,3] = mul c d
>>>     f:f32[1,3] = broadcast_in_dim[broadcast_dimensions=(1,) shape=(1, 3)] b
>>>     g:f32[3,3] = add f e
>>>     h:f32[] = reduce_sum[axes=(0, 1)] g
>>>   in (h,) }

f2_jaxpr.consts    #F

>>> [DeviceArray([2., 1., 0.], dtype=float32)]    #G
```

#A Side-effect
#B Using global variable z
#C Generating jaxpr for the function
#D Side-effect result
#E Checking the resulting jaxpr
#F Checking the list of constants
#G Our global variable is now a constant

Here we changed our function to contain a side-effect (the `print()` statement) and use a global variable (variable `z`). The side-effect emerges during tracing, so we can see its result during the call of the function obtained with the `make_jaxpr()` function transformation. The resulting jaxpr does not contain anything related to the print statement. The global variable is now saved in the list of constants accompanying the resulting jaxpr, so if you later change the global variable, the compiled function will still use the old value stored as a constant.

By default, `jax.jit` uses the `ShapedArray` tracer for tracing. It has a concrete shape but no concrete value. Because of this, the compiled function works on all possible inputs with the same shape, which is a typical case in machine learning.

The idea is that we want our compiled (or more generally transformed) code to work with different input values, so JAX traces on abstract values representing sets of possible inputs. There are different levels of abstraction (https://github.com/google/jax/blob/main/jax/_src/abstract_arrays.py), and different transformations use different levels. Higher levels of abstraction give a more general view of the Python code, reduce the number of recompilations, but impose more constraints on the Python code to be able to trace. Tracing with the highest level of abstraction using the `UnshapedArray` tracer is not currently a default for any transformation.

In the case of JIT, it means, tracing has no problems with control flow statements if they do not depend on any input parameter value (but it is allowed to use its shape):

Listing 5.12 Tracing with control structures.

```
def f3(x):
  y = x
  for i in range(5):    #A
    y += i
  return y

jax.make_jaxpr(f3)(0)

>>> { lambda ; a:i32[]. let
>>>     b:i32[] = add a 0    #B
>>>     c:i32[] = add b 1    #B
>>>     d:i32[] = add c 2    #B
>>>     e:i32[] = add d 3    #B
>>>     f:i32[] = add e 4    #B
>>>   in (f,) }

jax.jit(f3)(0)    #C

>>> DeviceArray(10, dtype=int32, weak_type=True)

def f4(x):
  y = 0
  for i in range(x.shape[0]):    #D
    y += x[i]
  return y

jax.make_jaxpr(f4)(jnp.array([1.0, 2.0, 3.0]))

>>> { lambda ; a:f32[3]. let    #E
>>>     b:f32[1] = slice[limit_indices=(1,) start_indices=(0,) strides=(1,)] a
>>>     c:f32[] = squeeze[dimensions=(0,)] b
>>>     d:f32[] = add 0.0 c
>>>     e:f32[1] = slice[limit_indices=(2,) start_indices=(1,) strides=(1,)] a
>>>     f:f32[] = squeeze[dimensions=(0,)] e
>>>     g:f32[] = add d f
>>>     h:f32[1] = slice[limit_indices=(3,) start_indices=(2,) strides=(1,)] a
>>>     i:f32[] = squeeze[dimensions=(0,)] h
>>>     j:f32[] = add g i
>>>   in (j,) }

jax.jit(f4)(jnp.array([1.0, 2.0, 3.0]))    #F

>>> DeviceArray(6., dtype=float32)
```

#A Loop does not depend on an input parameter
#B The loop is unrolled
#C JIT compilation succeeded
#D Loop depends on an input parameter shape

#E The loop in unrolled
#F JIT compilation succeeded

Here in the code we used a fixed value independent of any input parameter, and an input parameter shape to make a loop. JAX successfully traced both cases, unrolling the loops and JIT-compiling the functions.

However, if you try to use an input parameter value in a control structure, you will fail:

Listing 5.13 Tracing with a for loop depending on input parameter value.

```
def f5(x):
  y = 0
  for i in range(x):    #A
    y += i
  return y

f5(5)    #B

>>> 10

jax.make_jaxpr(f5)(5)    #C

>>> ...
>>> The above exception was the direct cause of the following exception:
>>>
>>> TracerIntegerConversionError              Traceback (most recent call last)
>>> <ipython-input-54-626705d2393b> in f5(x)
>>>       1 def f5(x):
>>>       2   y = 0
>>> ----> 3   for i in range(x):
>>>       4     y += i
>>>       5   return y
>>>
>>> TracerIntegerConversionError: The __index__() method was called on the JAX Tracer
       object Traced<ShapedArray(int32[],
       weak_type=True)>with<DynamicJaxprTrace(level=1/0)>
>>> See
       https://jax.readthedocs.io/en/latest/errors.html#jax.errors.TracerIntegerConversionE
       rror
jax.jit(f5)(5)   #D

>>> ...
>>> TracerIntegerConversionError: The __index__() method was called on the JAX Tracer
       object Traced<ShapedArray(int32[],
       weak_type=True)>with<DynamicJaxprTrace(level=0/1)>
>>> See
       https://jax.readthedocs.io/en/latest/errors.html#jax.errors.TracerIntegerConversionE
       rror
```

#A Loop depends on an input parameter
#B It is perfectly normal Python function that works
#C JAX has problems tracing the function
#D JIT fails because the tracing fails

Here we used an input parameter value as a parameter for the loop, and tracing has problems doing its work because of being unable to deal with the parameter value.

The same would happen if we used an if statement:

Listing 5.14 Tracing with an if statement depending on input parameter value.

```
def relu(x):
  if x > 0:    #A
    return x
  return 0.0

relu(10.0)    #B

>>> 10.0

jax.make_jaxpr(relu)(10.0)    #C

>>> The above exception was the direct cause of the following exception:
>>>
>>> ConcretizationTypeError                   Traceback (most recent call last)
>>> <ipython-input-58-5bd36ce502aa> in relu(x)
>>>       1 def relu(x):
>>> ----> 2   if x > 0:
>>>       3     return x
>>>       4   return 0.0
>>>
>>> ConcretizationTypeError: Abstract tracer value encountered where concrete value is
       expected: Traced<ShapedArray(bool[],
       weak_type=True)>with<DynamicJaxprTrace(level=1/0)>
>>> The problem arose with the `bool` function.
>>>
>>> See https://jax.readthedocs.io/en/latest/errors.html#jax.errors.ConcretizationTypeError
```

#A If statement depends on an input parameter
#B It is perfectly normal Python function that works
#C Tracing fails

Here we used an input parameter value in if statement and tracing failed as well, however, the function is a normal Python function that works.

You have control over the tracing process, and you can use the mechanism of static parameters you are already familiar with (see section 5.1.1). Specifying a particular parameter as static (both, `jax.jit()` and `jax.make_jaxpr()` functions support it) makes tracing use of concrete values and the compilation works:

Listing 5.15 Using static values to trace with concrete values.

```
def f5(x):
  y = 0
  for i in range(x):    #A
    y += i
  return y

def relu(x):
  if x > 0:    #A
    return x
  return 0.0

jax.make_jaxpr(f5, static_argnums=0)(5)    #B

>>> { lambda ; . let  in (10,) }    #C

jax.jit(f5, static_argnums=0)(5)    #B

>>> DeviceArray(10, dtype=int32, weak_type=True)    #D

jax.make_jaxpr(relu, static_argnums=0)(12.3)    #B

>>> { lambda ; . let  in (12.3,) }    #C

jax.jit(relu, static_argnums=0)(12.3)    #B

>>> DeviceArray(12.3, dtype=float32, weak_type=True)    #D
```

#A Dependency on an input parameter
#B Marking the first parameter as static
#C The resulting expression is effectively a pre-calculated constant
#D Compilation also succeeded

Here is the tradeoff. Your function is now being compiled on any call with a new input parameter value. This might be ok for a function with a small set of possible values (as might be for our `f5()` function), but it is definitely not ok for functions with a lot of possible values (which is definitely the case for the `relu()` activation function which may take in almost any value during neural network training).

Another way to solve this issue more efficiently is to use the structured control flow primitives we mentioned in section 3.4.

Let’s first replace our Python for-loop from Listing 5.13:

Listing 5.16 Replacing for loop with a structured control flow primitive.

```
def f5(x):
  return jax.lax.fori_loop(0, x, lambda i,v: v+i, 0)    #A

f5(5)

>>> DeviceArray(10, dtype=int32, weak_type=True)    #B

jax.make_jaxpr(f5)(5)    #C

>>> { lambda ; a:i32[]. let
>>>     _:i32[] _:i32[] b:i32[] = while[
>>>       body_jaxpr={ lambda ; c:i32[] d:i32[] e:i32[]. let
>>>           f:i32[] = add c 1
>>>           g:i32[] = add e c
>>>         in (f, d, g) }
>>>       body_nconsts=0
>>>       cond_jaxpr={ lambda ; h:i32[] i:i32[] j:i32[]. let
>>>           k:bool[] = lt h i
>>>         in (k,) }
>>>       cond_nconsts=0
>>>     ] 0 a 0
>>>   in (b,) }

jax.jit(f5)(5)    #D

>>> DeviceArray(10, dtype=int32, weak_type=True)
```

#A Using jax.lax.fori_loop to replace a for loop of Listing 5.12
#B Result is still the same
#C Jaxpr is now more complex
#D The function now compiles

Here we replaced a for-loop dependent on an input parameter from Listing 5.13 with a `jax.lax.fori_loop()` primitive (https://jax.readthedocs.io/en/latest/_autosummary/jax.lax.fori_loop.html). The `fori_loop(lower, upper, body_fun, init_val)` is equivalent to the following Python code:

```
def fori_loop(lower, upper, body_fun, init_val):
  val = init_val
  for i in range(lower, upper):
    val = body_fun(i, val)
  return val
```

Now let's replace the if-statement from the Listing 5.14:

Listing 5.17 Replacing if statement with a structured control flow primitive.

```
def relu(x):
  return jax.lax.cond(x>0, lambda x: x, lambda x: 0.0, x)    #A

relu(12.3)

>>> DeviceArray(12.3, dtype=float32, weak_type=True)    #B

jax.make_jaxpr(relu)(12.3)    #C

>>> { lambda ; a:f32[]. let
>>>     b:bool[] = gt a 0.0
>>>     c:i32[] = convert_element_type[new_dtype=int32 weak_type=False] b
>>>     d:f32[] = cond[
>>>       branches=(
>>>         { lambda ; e:f32[]. let  in (0.0,) }
>>>         { lambda ; f:f32[]. let  in (f,) }
>>>       )
>>>       linear=(False,)
>>>     ] c a
>>>   in (d,) }

jax.jit(relu)(12.3)    #D

>>> DeviceArray(12.3, dtype=float32, weak_type=True)
```

#A Using jax.lax.cond to replace an if-statement of Listing 5.14
#B Result is still the same
#C Jaxpr is now more complex
#D The function now compiles

Here we replaced Python if-statement with the `jax.lax.cond()` primitive (https://jax.readthedocs.io/en/latest/_autosummary/jax.lax.cond.html). It is equivalent to the following Python implementation:

```
def cond(pred, true_fun, false_fun, *operands):
  if pred:
    return true_fun(*operands)
  else:
    return false_fun(*operands)
```

Both `true_fun()` and `false_fun()` must be callable objects and return exactly the same types.

Actually, not every JAX transformation materializes a jaxpr, as we discussed before. Some transformations, say, taking gradients or batching, apply transformations incrementally during tracing, not taking a jaxpr first, then processing it to obtain a modified jaxpr.

We have finished with the first stage of compilation, converting Python code to an intermediate representation, jaxpr. Then the second stage happens, compiling Jaxpr to native code with XLA.

5.2.2 XLA

XLA stands for Accelerated Linear Algebra (https://www.tensorflow.org/xla), a domain-specific compiler for linear algebra, originally developed to accelerate TensorFlow models, potentially with no source code changes.

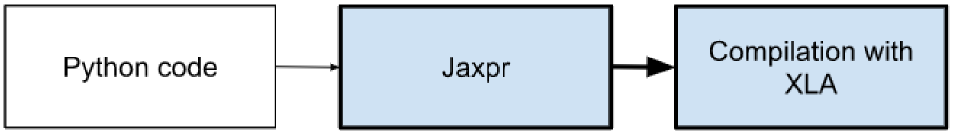


Figure 5.3. This section focuses on Jaxpr to native code conversion

XLA ORIGINS AND ARCHITECTURE

The reason to create XLA was that while each separate operation in a computation graph may be highly optimized, a user may compose more complex operations out of simpler ones or create a large composition that is not guaranteed to run in the most efficient way. So, Google developed XLA that used JIT compilation techniques to analyze the TensorFlow graph, specialize it for the actual runtime dimensions and types, and, most importantly, fuse multiple operations together. It can exploit model-specific information for the optimization and compiles the TensorFlow graph into a sequence of computation kernels generated specifically for the given model.

For example, let’s take an operation we had in Listing 5.10, that takes three tensors at its input and calculates output:

```
def f(x, y, z):
  return jnp.sum(x + y * z)    #A
```

#A Performing three operations: multiplication, addition, sum

Without XLA, such computation may result in three different operations implemented with different computation kernels, say, on GPU: one for multiplication, one for addition, and one for the final summation. XLA can optimize the computation to produce the result in a single kernel launch. It can fuse multiplication, addition and summation into a single GPU kernel. Moreover, the fused operation does not produce intermediate variables containing y*z and x+y*z in memory, and may feed them directly to subsequent computations keeping data in the same memory location or in GPU registers. Removing unnecessary memory transfers is a big deal, as memory bandwidth can be a bottleneck for your computation.

XLA emits efficient native machine code for such devices like CPUs, GPUs and custom accelerators such as Google’s TPU. The XLA subsystem that performs target device-specific optimization and code generation is called *backend*.

The system that sends data to XLA is called *frontend*. The original frontend for XLA was TensorFlow, but now XLA programs can also be generated by PyTorch, JAX, Julia and Nx (the numerical computing library for the Elixir programming language).

XLA uses a special input language called ***HLO IR*** (High Level Operations Intermediate Representation), or just ***HLO***. XLA takes in computations defined in HLO and compiles them into special machine instructions for target hardware architecture.

There are many basic operations in HLO. You can see the whole list here: https://www.tensorflow.org/xla/operation_semantics.

Figure 5.4 shows that there are two optimization steps: target-independent and target-dependent.

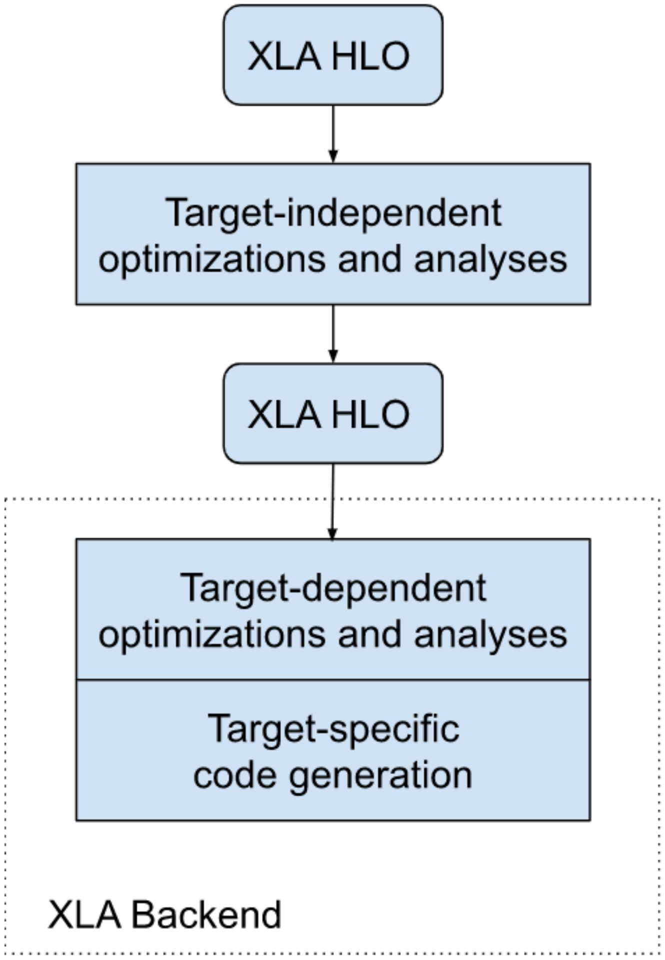

XLA HLO
Target-independent optimizations and analyses
XLA HLO
Target-dependent optimizations and analyses
Target-specific code generation
XLA Backend

Figure 5.4 The compilation process in XLA.

On the target-independent step, XLA makes optimizations independent of the hardware that will execute computations. This includes common subexpression elimination, target-independent operation fusion, and buffer analysis for allocating runtime memory for the computation.

On the target-dependent step, the XLA backend can perform further HLO-level optimizations. For example, it may determine how to better partition computation into GPU streams and perform other fusions specifically for the particular GPU. It may also perform special pattern matching to replace certain operation combinations to optimized library calls.

After the target-dependent optimization and analysis, the XLA backend generates target-specific code. The CPU and GPU backends use LLVM (https://llvm.org/) for low-level intermediate representation, optimization and code generation. These backends output LLVM IR that represents XLA HLO IR, and then invoke LLVM to generate native code from this LLVM IR.

The CPU backend supports x64 and ARM64, the GPU backend supports NVIDIA GPUs, and of course there is a TPU backend supporting Google Cloud TPUs. There is also an ongoing movement to support AMD GPUs in both JAX (https://github.com/google/jax/issues/2012) and XLA (https://github.com/tensorflow/tensorflow/pulls?utf8=%E2%9C%93&q=is%3Apr+ROCm+XLA).

There is also a way for developing your own XLA backends (https://www.tensorflow.org/xla/developing_new_backend) and this gives a real opportunity to have all recent deep learning hardware be supported soon.

XLA AND JAX

JAX uses XLA to generate efficient code for specific backends. But that is not the whole story.

Since January 2022, JAX uses the MHLO MLIR dialect as its primary target compiler IR by default, and the backend was switched from XLA/HLO to MLIR/MHLO. Now the workflow is the following (https://github.com/google/jax/issues/10715):

1. You write a Python function
2. JAX converts the Python function to Jaxpr
3. JAX converts the Jaxpr to MHLO for MLIR
4. MLIR converts MHLO to optimized MHLO
5. JAX converts optimized MHLO to HLO (?)
6. XLA converts HLO to optimized HLO
7. XLA converts optimized HLO to the native code for CPU/GPU/TPU (for CPU and GPU it uses LLVM)

MLIR.

There is an interesting project called MLIR (https://mlir.llvm.org/) or Multi-Level Intermediate Representation. It is a successor of LLVM also inspired by Chris Lattner (https://www.youtube.com/watch?v=qzljG6DKgic).

The MLIR's goal is to build reusable and extensible compiler infrastructure suitable for heterogeneous hardware world. In particular, it has great perspectives in building an optimizing compiler infrastructure for deep learning applications.

MLIR is intended to be a hybrid intermediate representation (IR) which can support multiple different requirements in a unified infrastructure, including but not limited to:

- Representing dataflow graphs such as in TensorFlow.
- Optimizations and transformations typically done on such graphs.
- Ability to host high-performance-computing-style loop optimizations across kernels (fusion, loop interchange, tiling, etc.), and to transform memory layouts of data.
- Code generation "lowering" transformations such as DMA insertion, explicit cache management, memory tiling, and vectorization for 1D and 2D register architectures.
- Ability to represent target-specific operations, e.g. accelerator-specific high-level operations.
- Quantization and other graph transformations are done on a Deep-Learning graph.
- Polyhedral primitives.
- Hardware Synthesis Tools / HLS.

MLIR is a powerful representation, but it also has non-goals. It does not try to support low level machine code generation algorithms (like register allocation and instruction scheduling), as there are better fits for lower level optimizers (such as LLVM). Also MLIR is not intended to be a source language that end-users would themselves write kernels in (analogous to CUDA C++).

MLIR supports different *dialects* (https://www.tensorflow.org/mlir/dialects) for its IR. JAX uses MLIR MHLO dialect ("meta"-HLO dialect, https://github.com/tensorflow/mlir-hlo#meta-hlo-dialect-mhlo). The list of its operations can be viewed here: https://www.tensorflow.org/mlir/hlo_ops.

There is a movement to build a standalone "HLO" MLIR-based compiler called MLIR-HLO (https://github.com/tensorflow/mlir-hlo) and an MLIR backend (https://www.tensorflow.org/mlir/xla_gpu_codegen).

In the following code we will see how the Python function is converted first to MHLO and then to lower-level HLO:

Listing 5.18 Compiling Python code to MHLO and HLO.

```
def f(x, y, z):    #A
  return jnp.sum(x + y * z)

x = jnp.array([1.0, 1.0, 1.0])    #B
y = jnp.ones((3,3))*2.0    #B
z = jnp.array([2.0, 1.0, 0.0]).T    #B

f_jitted = jax.jit(f)    #C

f_lowered = f_jitted.lower(x,y,z)    #D
print(f_lowered.as_text())

>>> module @jit_f.6 {
>>>  func.func public @main(%arg0: tensor<3xf32>, %arg1: tensor<3x3xf32>, %arg2:
       tensor<3xf32>) -> tensor<f32> {
>>>    %0 = "mhlo.broadcast_in_dim"(%arg2) {broadcast_dimensions = dense<1> :
       tensor<1xi64>} : (tensor<3xf32>) -> tensor<1x3xf32>
>>>    %1 = "mhlo.broadcast_in_dim"(%0) {broadcast_dimensions = dense<[0, 1]> :
       tensor<2xi64>} : (tensor<1x3xf32>) -> tensor<3x3xf32>
>>>    %2 = mhlo.multiply %arg1, %1 : tensor<3x3xf32>
>>>    %3 = "mhlo.broadcast_in_dim"(%arg0) {broadcast_dimensions = dense<1> :
       tensor<1xi64>} : (tensor<3xf32>) -> tensor<1x3xf32>
>>>    %4 = "mhlo.broadcast_in_dim"(%3) {broadcast_dimensions = dense<[0, 1]> :
       tensor<2xi64>} : (tensor<1x3xf32>) -> tensor<3x3xf32>
>>>    %5 = mhlo.add %4, %2 : tensor<3x3xf32>
>>>    %6 = mhlo.constant dense<0.000000e+00> : tensor<f32>
>>>    %7 = mhlo.reduce(%5 init: %6) across dimensions = [0, 1] : (tensor<3x3xf32>,
       tensor<f32>) -> tensor<f32>
>>>     reducer(%arg3: tensor<f32>, %arg4: tensor<f32>)  {
>>>      %8 = mhlo.add %arg3, %arg4 : tensor<f32>
>>>      "mhlo.return"(%8) : (tensor<f32>) -> ()
>>>    }
>>>    return %7 : tensor<f32>
>>>  }
>>> }

f_compiled = f_jitted.lower(x,y,z).compile()    #E
print(f_compiled.as_text())

>>> HloModule jit_f.7, entry_computation_layout={(f32[3]{0},f32[3,3]{1,0},f32[3]{0})-
       >f32[]}

>>> %region_0.15 (Arg_0.16: f32[], Arg_1.17: f32[]) -> f32[] {
>>>  %Arg_0.16 = f32[] parameter(0)
>>>  %Arg_1.17 = f32[] parameter(1)
>>>  ROOT %add.18 = f32[] add(f32[] %Arg_0.16, f32[] %Arg_1.17),
       metadata={op_name="jit(f)/jit(main)/reduce_sum[axes=(0, 1)]" source_file="<ipython-
       input-15-95d48614b981>" source_line=2}
>>> }

>>> %fused_computation (param_0.4: f32[3,3], param_1.4: f32[3], param_2.2: f32[3]) -> f32[]
       {
>>>   %param_2.2 = f32[3]{0} parameter(2)
>>>   %broadcast.3 = f32[3,3]{1,0} broadcast(f32[3]{0} %param_2.2), dimensions={1},
       metadata={op_name="jit(f)/jit(main)/add" source_file="<ipython-input-15-
       95d48614b981>" source_line=2}
>>>   %param_0.4 = f32[3,3]{1,0} parameter(0)
>>>   %param_1.4 = f32[3]{0} parameter(1)
>>>   %broadcast.2 = f32[3,3]{1,0} broadcast(f32[3]{0} %param_1.4), dimensions={1},
       metadata={op_name="jit(f)/jit(main)/mul" source_file="<ipython-input-15-
       95d48614b981>" source_line=2}
>>>   %multiply.0 = f32[3,3]{1,0} multiply(f32[3,3]{1,0} %param_0.4, f32[3,3]{1,0}
       %broadcast.2), metadata={op_name="jit(f)/jit(main)/mul" source_file="<ipython-input-
       15-95d48614b981>" source_line=2}
>>>   %add.0 = f32[3,3]{1,0} add(f32[3,3]{1,0} %broadcast.3, f32[3,3]{1,0} %multiply.0),
       metadata={op_name="jit(f)/jit(main)/add" source_file="<ipython-input-15-
       95d48614b981>" source_line=2}
>>>   %bitcast.1 = f32[9]{0} bitcast(f32[3,3]{1,0} %add.0)
>>>   %constant_0 = f32[] constant(0)
>>>   ROOT %reduce.1 = f32[] reduce(f32[9]{0} %bitcast.1, f32[] %constant_0),
       dimensions={0}, to_apply=%region_0.15,
       metadata={op_name="jit(f)/jit(main)/reduce_sum[axes=(0, 1)]" source_file="<ipython-
       input-15-95d48614b981>" source_line=2}
>>> }

>>> ENTRY %main.20 (Arg_0.1: f32[3], Arg_1.2: f32[3,3], Arg_2.3: f32[3]) -> f32[] {
>>>   %Arg_1.2 = f32[3,3]{1,0} parameter(1)
>>>   %Arg_2.3 = f32[3]{0} parameter(2)
>>>   %Arg_0.1 = f32[3]{0} parameter(0)
>>>   ROOT %fusion = f32[] fusion(f32[3,3]{1,0} %Arg_1.2, f32[3]{0} %Arg_2.3, f32[3]{0}
       %Arg_0.1), kind=kLoop, calls=%fused_computation,
       metadata={op_name="jit(f)/jit(main)/reduce_sum[axes=(0, 1)]" source_file="<ipython-
       input-15-95d48614b981>" source_line=2}
>>> }
```

#A A simple function with a few operations
#B Some test data
#C JIT-ting the function
#D Lowering the function (generating MHLO code)
#E Compiling the lowered function for the specific backend (generating HLO code)

Here we took the same function we used in sections on Jaxpr and XLA. It contains multiplication, addition, and summation. We provide some test data to calculate the function with. Then the interesting things begin. We JIT-compile our function. For the JIT-ted function, we can create a lowered function. *Lowering* is a process of converting a higher-level representation to a lower-level representation. Here we create MHLO IR code consisting of basic MHLO operations. The MHLO code is pretty straightforward, almost resembling the original calculations. Then we move further, compiling this MHLO code to HLO optimized for the target backend. This HLO code contains fused computation.

Analyzing MHLO and HLO code, and diving deeper into XLA and MLIR is out of scope for the book, yet you now have a basic understanding of what stages comprise the compilation process and what happens there.

Here for obtaining MHLO and HLO we used special functions for lowering and compiling code. These functions actually comprise the new Ahead-of-Time (AOT) compilation API.

5.2.3 Using Ahead-of-Time (AOT) compilation

Since September 2022 and JAX version 0.3.18 AOT compilation is introduced (https://github.com/google/jax/blob/main/docs/aot.md). AOT compilation might help if you want control over when different parts of the compilation process happen or if you want to compile before execution time fully.

Say you have some Python function `f(x)` where `x` is an array. You apply the `jit()` transformation to the function and obtain a wrapped version `f_jit = jit(f)`. Then at some point, you invoke the jitted function with arguments `f_jit(x)`. Here compilation happens.

The compilation has several stages:

1. **Stage out** the original function f to an internal representation. This specialized version of the function reflects restrictions to input types inferred from properties of arguments (here, only `x`).
2. **Lower** the specialized staged-out representation to the input language of MLIR and XLA, here MHLO.
3. **Compile** the lowered HLO program into device-specific optimized code for CPU, GPU, or TPU.
4. **Execute** the compiled code with given arguments (here, `x`).

AOT API gives you control over the last three stages. There is a special `jax.stages` API (https://jax.readthedocs.io/en/latest/jax.stages.html) to represent stages of the compiled execution process.

Listing 5.19 Compiling with JIT and AOT.

```
def selu(x,    #A
         alpha=1.6732632423543772848170429916717,
         scale=1.0507009873554804934193349852946):
  '''Scaled exponential linear unit activation function.'''
  print('Function run')    #B
  return scale * jnp.where(x > 0, x, alpha * jnp.exp(x) - alpha)

selu_jit = jax.jit(selu)    #C

selu_aot = jax.jit(selu).lower(1.0).compile()    #D

>>> Function run    #E

selu_jit(17.8)    #F

>>> Function run    #G
>>> DeviceArray(18.702477, dtype=float32, weak_type=True)

selu_aot(17.8)    #H

>>> DeviceArray(18.702477, dtype=float32, weak_type=True)
```

#A Some function to test compiling
#B A side-effect for debugging purposes
#C JIT-compiled function
#D AOT-compiled function
#E Side-effect works during the AOT-compilation
#F Calling JIT-compiled function
#G Side-effect works during the JIT-compilation

#H Calling AOT-compiled function, no side-effects

Here in the code we used the same `selu()` activation function from the beginning of the chapter, but we deliberately added a side-effect to understand where tracing and compilation happens.

We created a JIT-compiled version of it and also an AOT-compiled version. Both functions produce the same results. However the AOT-compiled one does not require a warmup, and at the moment of its invocation we have already compiled code. We see it by the time the side-effect works. For the AOT-compiled function it works during the direct compilation, for the JIT-compiled function it happens during the first function invocation, when the compilation really happens.

Both lowering and compilation happen for a fixed type signature and the AOT-compiled function can only be called with arguments of the fixed type signature. If you invoke the AOT-compiled function with arguments that are incompatible with its lowering (say, float32 instead of int32), you will get an error.

AOT-compiled functions cannot be transformed with transformations such as `jit`, `grad`, `jvp`, `vmap`. This is done because internally many transformations alter type signature of functions. However, `jit` does not modify its arguments' type signature, it is disallowed as well.

Listing 5.20 Differences in behavior between JIT and AOT.

```
selu_jit(17)    #A

>>> Function run
>>> DeviceArray(17.861917, dtype=float32, weak_type=True)

selu_aot(17)    #B

>>> ...
>>> TypeError: Computation compiled for input types:
>>>   ShapedArray(float32[], weak_type=True)
>>> called with:
>>>   ShapedArray(int32[], weak_type=True)

selu_jit_batched = jax.vmap(selu_jit)    #C
selu_aot_batched = jax.vmap(selu_aot)    #C

selu_jit_batched(jnp.array([42.0, 78.0, -12.3]))    #D

>>> Function run
>>> DeviceArray([44.129444 , 81.95468  , -1.7580913], dtype=float32)

selu_aot_batched(jnp.array([42.0, 78.0, -12.3]))    #E

>>> ...
>>> TypeError: Cannot apply JAX transformations to a function lowered and compiled for a
       particular signature. Detected argument of Tracer type <class
       'jax.interpreters.batching.BatchTracer'>.
```

#A Calling a JIT-compiled function on integer data runs re-compilation
#B Calling an AOT-compiled function on integer data cannot run re-compilation and returns an error

#C Creating batched versions of the function (on vmap see the next chapter)
#D JIT-compiled function can be transformed and compilation happens again
#E AOT-compiled function cannot be transformed and an error returned

Here we have applied both compiled functions from the previous example to a different data type, now for 32-bit integers, instead of 32-bit floats. For the JIT-compiled function this is not a problem, the function is recompiled. The AOT-compiled function cannot be recompiled, so an error is returned.

Then we apply a second transformation, `vmap()`, that creates a batched version of the function (more on this transformation in the next chapter, however we have already used it in Chapter 2). The JIT-compiled function combines with the new transformation well, but the AOT-compiled function cannot do it, and an error is returned.

The AOT stages also offer some additional features for debugging purposes. We used this functionality in Listing 5.18 to obtain MHLO and HLO representations. For compiled functions you can also get cost and memory analyses via `cost_analysis()` and `memory_analysis()` functions. Both are intended for visualization and debugging purposes, they provide some simple data structure that can easily be printed or serialized, but it may be inconsistent across versions of JAX and jaxlib, or even across invocations.

JAX and jaxlib.

JAX is published as two separate Python packages:

- `jax`, **a pure Python package**
- `jaxlib`, **a mostly-C++ package that contains libraries such as XLA, pieces of LLVM used by XLA, MLIR infrastructure with MHLO Python bindings, JAX-specific C++ libraries for fast JIT and PyTree manipulation.**

The JAX distribution is structured this way because most changes to JAX touch only Python code, and it makes it easy to work on the Python part of JAX without building C++ code. So, the Python pieces can be updated independently of the C++ pieces, which makes development velocity better.

Both `jax` **and** `jaxlib` **share the same version number, but are released separately. When installed, the** `jax` **package version must be greater than or equal to** `jaxlib`**'s version, and** `jaxlib`**'s version must be greater than or equal to the minimum** `jaxlib` **version specified by** `jax`**.**

You can read more about JAX and Jaxlib versioning here: https://jax.readthedocs.io/en/latest/jep/9419-jax-versioning.html#why-are-jax-and-jaxlib-separate-packages

We have covered a lot of ground on how to use JIT and AOT compilation, and how JAX works internally. It will help us in dealing with JIT limitations, and it is important to summarize what are the JIT limitations.

5.3 JIT limitations

JIT is a powerful mechanism. However, it has limitations and does not work everywhere. We have already discussed some of them, yet it is the right place to gather them to have everything related in one place. Now with understanding the JIT internals it will be much easier to understand the nature of these limitations and we have tools to overcome them.

PURE AND IMPURE FUNCTIONS

First of all, as we have already discussed in section 5.1.3, JIT correctly works only with pure functions. So JIT can change the behavior of your function if you rely on its impurity, using side-effects or global state.

EXACT NUMERICS

Second, JIT can also change the exact numerics of function outputs. This may happen because of optimizations along the way. For example, XLA may rearrange floating-point operations, or get rid of some redundant computations that should not have any effect from the mathematical point of view (say, first dividing a value by some number, then multiplying the result with the same number), but may have effect because of accumulating arithmetical errors due to floating-point arithmetic issues.

CONDITIONING ON INPUT PARAMETER VALUES

Third, there are limitations related to having control flow and conditioning on input parameter values. We have discussed it in section 5.2.1.

SLOW COMPILATION

Fourth, jit-ted functions could sometimes be very slow to compile, even tens or more seconds. The reason for this is usually when your code generates a very large internal representation, the jaxpr. Long jaxpr could arise from long loops that are unrolled during tracing and heavy use of control flow. This problem can be diagnosed by making jaxpr of your function. If the jaxpr contains hundreds or thousands of lines, you can expect slow compilation, as XLA compilation time scales as roughly the square of the number of operations sent to it.

You should try to remove such loops from your program. There are many ways of doing this, including vectorizing the calculations (which is a typical advice for any NumPy-like computations), and replacing Python loops with structured control flow primitives from `jax.lax`. Changing the algorithm may also be an option (but do not mix compilation time with execution time, the latter one usually matters much more). You might also avoid wrapping such loops with `jit`, however you may use `jit` for functions inside the loop.

Consider an illustrative example of calculating a cumulative sum (or running total). You have an array of elements and the goal is to produce another array where each element is a sum of all elements from the original array before it and including it. A naive Python implementation may look like this:

Listing 5.21 Cumulative sum naive implementation

```
def cumulative_sum(x):    #A
  acc = 0.0
  y = []
  for i in range(x.shape[0]):
    acc += x[i]
    y.append(acc)
  return y

j = jax.make_jaxpr(cumulative_sum)(jnp.ones(10000))

len(j.jaxpr.eqns)    #B

>>> 30000

%time jax.jit(cumulative_sum)(jnp.ones(10000))

>>> CPU times: user 2min 16s, sys: 1.47 s, total: 2min 18s    #C
>>> Wall time: 2min 23s    #C
```

#A The function for calculating cumulative sum
#B Resulting jaxpr contains thousands of lines
#C Compilation takes pretty long

Here, our simple one-level loop produced a lot of intermediate jaxpr equations as the loop was unrolled. There are 30000 equations in the jaxpr, and the compilation takes more than two minutes.

We can easily replace the solution with a long for-loop with the `lax.scan` primitive (https://jax.readthedocs.io/en/latest/_autosummary/jax.lax.scan.html):

Listing 5.22 Cumulative sum implementation with lax.scan

```
def cumulative_sum_fast(x):
  result, array = jax.lax.scan(lambda carry, elem: (carry+elem, carry+elem), 1.0, x)   #A
  return array

j = jax.make_jaxpr(cumulative_sum_fast)(jnp.ones(10000))

len(j.jaxpr.eqns)    #B

>>> 1

%time cs = jax.jit(cumulative_sum_fast)(jnp.ones(10000))

>>> CPU times: user 145 ms, sys: 6.02 ms, total: 151 ms    #C
>>> Wall time: 213 ms    #C
```

#A The function for calculating cumulative sum using lax.scan
#B Resulting jaxpr contains only one line (yet, pretty complex)
#C Compilation is very fast

Here, we significantly reduced the intermediate representation and the compilation time now significantly smaller.

The `jax.lax.scan()` primitive works the following way. It goes (scans) along the array, and calculates a given function for each element and an accumulated state (called carry). We use carry for storing accumulated sums. The function returns a new value for the accumulated state and a corresponding element of the output array.

It is equivalent to the following Python code:

```
def scan(f, init, xs, length=None):
  if xs is None:
    xs = [None] * length
  carry = init
  ys = []
  for x in xs:
    carry, y = f(carry, x)
    ys.append(y)
  return carry, np.stack(ys)
```

The `lax.scan` is a JAX primitive and lowered to a single XLA While HLO, so the resulting representation is tiny.

CLASS METHODS

Another tricky thing is that we previously used jit annotations only with standalone functions. However, you might want to use it for class methods. Simply annotating a class method with `@jit` annotation will produce an error:

Listing 5.23 Annotating a class method

```
class ScaleClass:
  def __init__(self, scale: jnp.array):
    self.scale = scale

  @jax.jit
  def apply(self, x: jnp.array):    #A
    return self.scale * x

scale_double = ScaleClass(2)    #B

scale_double.apply(10)    #C

>>> TypeError                                 Traceback (most recent call last)
>>> <ipython-input-30-7f0319829861> in <module>
>>> ----> 1 scale_double.apply(10)
>>>
>>> TypeError: Argument '<__main__.ScaleClass object at 0x7fe05d645e50>' of type <class
       '__main__.ScaleClass'> is not a valid JAX type.
```

#A A class method we want to JIT
#B Creating a class instance
#C Calling the jit-ted function and getting an error

The reason we get an exception here is that the class function has a first parameter being the instance of the class (here, `ScaleClass`). JAX does not know how to handle this type.

There are several ways to deal with it.

First, you may use a helper function:

Listing 5.24 Using a helper function for a class method.

```
from functools import partial

class ScaleClass:
  def __init__(self, scale: jnp.array):
    self.scale = scale

  def apply(self, x: jnp.array):
    return _apply_helper(self.scale, x)    #A

@partial(jax.jit, static_argnums=0)    #B
def _apply_helper(scale, x):    #C
  return scale*x

scale_double = ScaleClass(2)

scale_double.apply(10)    #D

>>> DeviceArray(20, dtype=int32, weak_type=True)
```

#A Now class method does not use JIT but uses a helper function
#B Using jit annotation with the scale parameter being static
#C The helper function outside of the class
#D Now everything works correctly

Here we removed `@jit` annotation from the class method, created a helper function outside of the class, and used this function in the class method. The helper function is annotated with `jit` annotation, but we used `functools.partial` to specify a static parameter. We marked the first parameter of the helper function as static because we assume it will not frequently change as it initializes a class instance, and the same value will be applied to many different arguments.

You can also make the `self` parameter static in the original code, but you need more changes in the code to avoid unexpected things. This method is described here: https://jax.readthedocs.io/en/latest/faq.html#strategy-2-marking-self-as-static.

Another, the most flexible approach, is to make our `ScaleClass` class a PyTree. We will talk about pytrees in Chapter 8, but if you are interested in this approach, it is described here: https://jax.readthedocs.io/en/latest/faq.html#strategy-3-making-customclass-a-pytree.

SIMPLE FUNCTIONS

There might be cases where a function is already small, and using JIT does not provide any significant boost. Moreover, it takes additional time to compile, and all the overhead of using MLIR/XLA and even time to copy data to a hardware accelerator may not be worth it.

Measure what you get from JIT and try to compile the largest possible chunk of the computation. This gives the compiler freedom to optimize better.

In this chapter we have covered speeding-up your code with compilation and the corresponding `jit()` transformation. In the following chapter we will know how to make your code even more performant with vectorization and parallelization, and the corresponding `vmap()` and `pmap()` transformations.

5.4 Summary

- JAX uses Just-in-Time (JIT) compilation to produce efficient code for CPU, GPU, and TPU with the `jit()` transformation or a corresponding `@jit` annotation.
- JIT compilation happens when a JIT-transformed function is executed for the first time.
- JAX is designed to work with functionally pure code, the code without global state and side-effects. You can still write and run impure functions, but JAX does not guarantee they will work correctly.
- JAX transformations work by first converting Python functions into a simple intermediate language called Jaxpr (short for JAX Expression) using tracing.
- By default, `jit()` uses the `ShapedArray` tracer for tracing, which has a concrete shape, but no concrete value. Because of this, the compiled function works on all possible inputs with the same shape.
- You can use Python control flow and loops inside functions if they do not depend on input parameter values (but may depend on their shape).
- You can mark specific arguments as static to make tracing use of concrete values. When a function is called with a different value, re-compilation happens.
- You can also use `jax.lax` structured control flow primitives when you need to condition on an input parameter value.
- JAX uses MLIR and XLA to perform target-independent and target-dependent optimizations and generate target-specific code for CPU, GPU, and TPU.
- XLA and MLIR use their special input languages, or intermediate representations, called HLO (High Level Operations) and MHLO (Meta HLO), respectively.
- In addition to Just-in-Time compilation, JAX provides Ahead-of-Time (AOT) compilation API to control when different parts of the compilation process happen.
- JIT can change the exact numerics of function outputs because of optimizations along the way.
- JIT-ted functions can sometimes be very slow to compile when your code generates a large internal representation, the jaxpr.
- Class methods require special approaches to work with JIT.