Convolutional Neural Networks with Swift for Tensorflow

Image Recognition and Dataset Categorization

Brett Koonce

Apress®

Convolutional Neural Networks with Swift for Tensorflow

Image Recognition and Dataset Categorization

Brett Koonce

Apress®

Convolutional Neural Networks with Swift for Tensorflow: Image Recognition and Dataset Categorization

Brett Koonce
Jefferson, MO, USA

ISBN-13 (pbk): 978-1-4842-6167-5 ISBN-13 (electronic): 978-1-4842-6168-2
https://doi.org/10.1007/978-1-4842-6168-2

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Aaron Black
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 NY Plaza, New York, NY 10014. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-6167-5. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Brett Koonce is the CTO of QuarkWorks, a mobile consulting agency. He's a developer with 5 years of experience creating apps for iOS and Android. His team has worked on dozens of apps that are used by millions of people around the world. Brett knows the pitfalls of development and can help you avoid them. Whether you want to build something from scratch, port your app from iOS to Android (or vice versa), or accelerate your velocity, Brett can help.

About the Technical Reviewer

Vishwesh Ravi Shrimali graduated from BITS Pilani in 2018, where he studied mechanical engineering. Since then, he has worked with Big Vision LLC on deep learning and computer vision and was involved in creating official OpenCV AI courses. Currently, he is working at Mercedes-Benz Research and Development India Pvt. Ltd. He has a keen interest in programming and AI and has applied that interest in mechanical engineering projects. He has also written multiple blogs on OpenCV and deep learning on LearnOpenCV, a leading blog on computer vision. He has also coauthored *Machine Learning for OpenCV 4* (Second Edition) by Packt. When he is not writing blogs or working on projects, he likes to go on long walks or play his acoustic guitar.

Introduction

In this book, we are going to learn convolutional neural networks by focusing on the specific problem of image recognition, using Swift for Tensorflow and a command-line Unix approach. If you are new to this field, then I would suggest you read the first few chapters and get a working system bootstrapped and then spend your time going through the basics with MNIST and CIFAR repeatedly, in particular familiarizing yourself with how neural networks work. If you feel comfortable with the core concepts already, then feel free to skip ahead to the middle where we explore some more powerful convolutional neural networks.

Why Swift

The short version is that I believe swift is a modern, open source, beginner-friendly language that has proven itself by solving real problems for iOS developers daily. By integrating automatic differentiation into the programming language, a number of interesting compiler techniques to address the limitations of current machine learning software and hardware become possible in the long term. This is in my opinion where the world is headed, one way or another.

Why image recognition

Image recognition is one of the oldest, most well-understood uses of neural networks. As a result, we can introduce the basics and then build up to advanced state-of-the-art approaches in a logically consistent manner. With this foundation, you will be able to branch out to tackle

other image-related tasks (e.g., object detection and segmentation) easily. The deep learning techniques needed to build large-scale convolutional neural networks translate easily to reinforcement learning and generative adversarial networks (GANs), two important areas of modern research. In addition, I believe this foundation will make it easy to make the transition to time sequence models such as recurrent neural networks (RNNs) and long short-term memory (LSTM) once you have mastered CNNs.

Why CLI

Broadly speaking, this book is going to focus on a command-line interface (CLI)–based approach using both a local machine on your home network and virtual machines in the remote, Google Cloud. This is in my opinion the best approach because

- We can control costs very effectively. In the worst-case scenario, you can perform the majority of your work using a local machine built for under a thousand dollars, and your only remaining cost will be electricity and time.
- We can scale easily from anywhere in the world. Using cloud instances full time can quickly become expensive, and so many people avoid learning cloud workflows. But using on-demand cloud-based resources periodically to augment your local workflow means you can learn the cloud in a very practical and efficient way. Eventually, you will be able to prototype and build solutions on your primary machine, then quickly scale them up in the cloud to parallelize computation and access more powerful hardware when needed or available.

- We can get the best of both worlds. While minimizing costs is certainly important, I have found that focusing on how much money you are spending tends to produce a mindset where you are afraid to try new things and experiment in general. Building your own machine puts you into the mindset of putting in more cycles to reduce your costs, which is in my opinion the key to success.

So, toward this end, we will utilize a command-line workflow with the following goals:

- We will use a local terminal interface to log in to all of our machines, so that there is literally no difference between our approaches on the desktop and in the cloud.
- We will utilize the same operating system and software locally and in the cloud so that we do not have to learn about differences between platforms. Then, by definition, any workflow you can do on your computer, you will be able to do in the cloud, and vice versa.

Ultimately, by blurring the line between your personal computer and the cloud, my goal is for you to understand that there is fundamentally no difference between doing things locally or remotely. The real limiting factor then is your imagination, not resources.

Doing things this way will be more work at first, I will admit. But once you have mastered this workflow, it will be much easier for you to scale in the future. If you are willing to put in the time now, this approach will make your skills much more flexible and powerful in the future. What you do with them is up to you.

How this book is organized

This book is organized as follows.

Basics

We will explore the basic building blocks of neural networks and how to combine them with convolutions to perform simple image recognition tasks.

- Neural networks (1D MLP/multilayer perceptron) and MNIST
- Convolutional neural networks (2D CNN) and MNIST
- Color, CNN stacks, and CIFAR

Advanced

We will build upon the above to produce actual state-of-the-art approaches in this field.

- VGG16
- ResNet 34
- ResNet 50

Mobile

We will look at some different approaches for mobile devices, which require us to utilize our computing resources carefully.

- SqueezeNet
- MobileNet v1
- MobileNet v2

State of the art

We will look at the work that leads up to EfficientNet, the current state of the art for image recognition. Then we will look at how people are working on finding ways to produce similar results by combining many different papers together.

- EfficientNet
- MobileNetV3
- Bag of tricks/reading papers

Future

We will zoom out a bit and look at why I am excited about swift for tensorflow as a whole and give you my vision of what the future of machine learning looks like.

- MNIST revisited
- You are here

Appendices

Here's some information that didn't quite fit in with the above but I still feel is important:

- A: Cloud Setup
- B: Hardware Prerequisites, Software Installation Guidelines, and Unix Quickstart
- C: Additional Resources

CHAPTER 1

MNIST: 1D Neural Network

In this chapter, we will look at a simple image recognition dataset called MNIST and build a basic one-dimensional neural network, often called a multilayer perceptron, to classify our digits and categorize black and white images.

Dataset overview

MNIST (Modified National Institute of Standards and Technology) is a dataset put together in 1999 that is an extremely important testbed for computer vision problems. You will see it everywhere in academic papers in this field, and it is considered the computer vision equivalent of hello world. It is a collection of preprocessed grayscale images of hand-drawn digits of the numbers 0–9. Each image is 28 by 28 pixels wide, for a total of 784 pixels. For each pixel, there is a corresponding 8-bit grayscale value, a number from 0 (white) to 255 (completely black).

At first, we're not even going to treat this as actual image data. We're going to unroll it – we're going to take the top row and pull off each row at a time, until we have a really long string of numbers. We can imagine expanding this concept across the 28 by 28 pixels to produce a long row of input values, a vector that's 784 pixels long and 1 pixel wide, each with a corresponding value from 0 to 255.

B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2_1

The dataset has been cleaned so that there's not a lot of non-digit noise (e.g., off-white backgrounds). This will make our job simpler. If you download the actual dataset, you will usually get it in the form of a comma-separated file, with each row corresponding to an entry. We can convert this into an image by literally assigning the values one a time in reverse. The actual dataset is 60000 hand-drawn **training** digits with corresponding **labels** (the actual number), and 10000 **test** digits with corresponding **labels**. The dataset proper is usually distributed as a python pickle (a simple way of storing a dictionary) file (you don't need to know this, just in case you run across this online).

So, our goal is to learn how to correctly guess what number we are looking at in the **test** dataset, based on our **model** that we have learned from the **training** dataset. This is called a **supervised learning** task since our goal is to emulate what another human (or model) has done. We will simply take individual rows and try to guess the corresponding digit using a simple version of a neural network called a **multilayer perceptron**. This is often shortened to **MLP**.

Dataset handler

We can use the dataset loader from "swift-models," part of the Swift for Tensorflow project, to make dealing with the preceding sample simpler. In order for the following code to work, you will need to use the following swift package manager import to automatically add the datasets to your code.

BASIC: If you are new to swift programming and just want to get started, simply use the swift-models checkout you got working in the chapter where we set up Swift for Tensorflow and place the following code (MLP demo) into the "main.swift" file in the LeNet-MNIST example and run "swift run LeNet-MNIST".

ADVANCED: If you are a swift programmer already, here is the base swift-models import file we will be using:

```
/// swift-tools-version:5.3
// The swift-tools-version declares the minimum version of
Swift required to build this package.

import PackageDescription

let package = Package(
  name: "ConvolutionalNeuralNetworksWithSwiftForTensorFlow",
  platforms: [
    .macOS(.v10_13),
  ],
  dependencies: [
    .package(
      name: "swift-models", url: "https://github.com/
      tensorflow/swift-models.git", .branch("master")
    ),
  ],
  targets: [
    .target(
      name: "MNIST-1D", dependencies: [.product(name:
      "Datasets", package: "swift-models")],
      path: "MNIST-1D"),
  ]
)
```

Hopefully, the preceding code is not too confusing. Importing this code library will make our lives much easier. Now, let's build our first neural network!

Code: Multilayer perceptron + MNIST

Let's look at a very simple demo. Put this code into a "main.swift" file with the proper imports, and we'll run it:

```
```

```
/// 1
import Datasets
import TensorFlow

// 2
struct MLP: Layer {
  var flatten = Flatten<Float>()
  var inputLayer = Dense<Float>(inputSize: 784, outputSize:
  512, activation: relu)
  var hiddenLayer = Den se<Float>(inputSize: 512, outputSize:
  512, activation: relu)
  var outputLayer = Dense<Float>(inputSize: 512, outputSize: 10)

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    return input.sequenced(through: flatten, inputLayer,
    hiddenLayer, outputLayer)
  }
}

// 3
let batchSize = 128
let epochCount = 12
var model = MLP()
```

```
let optimizer = SGD(for: model, learningRate: 0.1)
let dataset = MNIST(batchSize: batchSize)

print("Starting training...")

for (epoch, epochBatches) in
dataset.training.prefix(epochCount).enumerated() {
  // 4
  Context.local.learningPhase = .training
  for batch in epochBatches {
    let (images, labels) = (batch.data, batch.label)
    let (_, gradients) = valueWithGradient(at: model) { model
    -> Tensor<Float> in
      let logits = model(images)
      return softmaxCrossEntropy(logits: logits, labels: labels)
    }
    optimizer.update(&model, along: gradients)
  }

  // 5
  Context.local.learningPhase = .inference
  var testLossSum: Float = 0
  var testBatchCount = 0
  var correctGuessCount = 0
  var totalGuessCount = 0
  for batch in dataset.validation {
    let (images, labels) = (batch.data, batch.label)
    let logits = model(images)
    testLossSum += softmaxCrossEntropy(logits: logits, labels:
    labels).scalarized()
    testBatchCount += 1
```

```
    let correctPredictions = logits.argmax(squeezingAxis: 1) .
    == labels
    correctGuessCount += Int(Tensor<Int32>(correctPredictions).
    sum().scalarized())
    totalGuessCount = totalGuessCount + batch.data.shape[0]
  }

  let accuracy = Float(correctGuessCount) / Float(totalGuessCount)
  print(
    """
    [Epoch \(epoch + 1)] \
    Accuracy: \(correctGuessCount)/\(totalGuessCount)
    (\(accuracy)) \
    Loss: \(testLossSum / Float(testBatchCount))
    """
  )
}
```

Results

When you run the preceding code, you should get an output that looks like this:

```
Loading resource: train-images-idx3-ubyte Loading resource:
train-labels-idx1-ubyte Loading resource: t10k-images-idx3-
ubyte Loading resource: t10k-labels-idx1-ubyte
Starting training…
[Epoch 1] Accuracy: 9364/10000 (0.9364) Loss: 0.21411717
[Epoch 2] Accuracy: 9547/10000 (0.9547) Loss: 0.15427242
```

```
[Epoch 3] Accuracy: 9630/10000 (0.963) Loss: 0.12323072
[Epoch 4] Accuracy: 9645/10000 (0.9645) Loss: 0.11413358
[Epoch 5] Accuracy: 9700/10000 (0.97) Loss: 0.094898805
[Epoch 6] Accuracy: 9747/10000 (0.9747) Loss: 0.0849531
[Epoch 7] Accuracy: 9757/10000 (0.9757) Loss: 0.076825164
[Epoch 8] Accuracy: 9735/10000 (0.9735) Loss: 0.082270846
[Epoch 9] Accuracy: 9782/10000 (0.97) Loss: 0.07173009
[Epoch 10] Accuracy: 9782/10000 (0.97) Loss: 0.06860765
[Epoch 11] Accuracy: 9779/10000 (0.9779) Loss: 0.06677916
[Epoch 12] Accuracy: 9794/10000 (0.9794) Loss: 0.063436724
```

Congratulations, you've done machine learning! This demo is only a few lines long, but a lot is actually happening under the hood. Let's break down what's going on.

Demo breakdown (high level)

We will look at all of the preceding code, going through section by section using the number in the comments (e.g., //1, //2, etc.). We will first do a pass to try and explain what is going on at a high level and then do a second pass where we explain the nitty-gritty details.

Imports (1)

Our first few lines are pretty simple; we're importing the swift-models MNIST dataset handler and then the TensorFlow library.

Model breakdown (2)

Next, we build our actual neural network, an MLP model:

```
```
/// 2
struct MLP: Layer {
  var flatten = Flatten<Float>()
  var inputLayer = Dense<Float>(inputSize: 784, outputSize:
  512, activation: relu)
  var hiddenLayer = Dense<Float>(inputSize: 512, outputSize:
  512, activation: relu)
  var outputLayer = Dense<Float>(inputSize: 512, outputSize: 10)

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    return input.sequenced(through: flatten, inputLayer,
    hiddenLayer, outputLayer)
  }
}
```
```

What's in this data structure? Our first line just defines a new struct called MLP, which subclasses **Layer**, a type in swift for tensorflow. To define this class, S4tf enforces a **protocol** definition that we implement the function **forward** (formerly **callAsFunction**), which takes an **input** and maps it to an **output**. Our middle lines then actually define the layers of our perceptron:

```
```
    var flatten = Flatten<Float>()
    var inputLayer = Dense<Float>(inputSize: 784, outputSize:
    512, activation: relu)
```
```

```
    var hiddenLayer = Dense<Float>(inputSize: 512, outputSize:
    512, activation: relu)
    var outputLayer = Dense<Float>(inputSize: 512,
    outputSize: 10)
```

We have four internal layers:

1) A flatten operation: This just takes the input and reduces it to a single row of input numbers (a vector).

   Our dataset is internally giving us a picture of 28x28 pixels, and this just converts it into a row of numbers, 784 pixels long.

   Next, we have three **dense** layers, which are a special type of neural network called **fully connected** layers. The first goes from our initial input (e.g., the flattened 784x1 vector) to 512 nodes, like so.

2) A dense layer: 784 (the preceding input) to 512 nodes.

3) Another dense layer: 512 nodes to 512 nodes again.

4) An output layer: 512 nodes to 10 nodes (the number of digits, 0–9).

   And then, finally, a forward function, which is where our neural network logic magic happens. We literally take the input, run it through the flatten, dense1, dense2, and output layers to produce our result.

And so our

```
return input.sequenced(through: flatten, inputLayer,
hiddenLayer, outputLayer)
```

is then the call that actually takes the input and maps it through these four layers. We will look at the actual training loop next to understand how all of that actually happens, but a very large part of the magic of swift for tensorflow is on these few lines. We'll talk a little bit more about what is happening here in a second, but conceptually this function is nothing more than applying the preceding four layers in a sequence.

Global variables (3)

These lines are just setting up some different tools we're going to use:

```
```
let batchSize = 128
let epochCount = 12
var model = MLP()
let optimizer = SGD(for: model, learningRate: 0.1)
let dataset = MNIST(batchSize: batchSize)
```
```

The first two lines set a couple of global variables: our batchSize (how many MNIST examples we are going to look at each pass) and epochCount (number of passes over the dataset we're going to do).

The next line initializes our model, which we talked about earlier.

The fourth line initializes our optimizer, which we're going to talk about more in a second.

The last line sets up our dataset handler.

The next line starts our actual training process by looping over our data:

```
for (epoch, epochBatches) in dataset.training.
prefix(epochCount).enumerated()  {
```

Now we can get into the actual training loop!

Training loop: Updates (4)

Here's what the actual core of our training loop looks like. Conceptually, we're going to be taking a set of pictures or **batch** and showing each individual picture to the first input set of dense nodes, which will **fire** and go to the next hidden set of dense nodes, which will **fire** and go to the final output set of dense nodes. Then, we will take all of the outputs of the final layer of our network, select the largest one, and look at it. If this node is the same number as the original input we gave it, then we will give the network a **reward** and tell it to increase its confidence in the results. If this answer is the wrong one, then we will give the network a **negative reward** and tell it to decrease its confidence in its results. By repeating this process using thousands of samples, our network can learn to accurately predict inputs it has never seen before.

```
  Context.local.learningPhase = .training
  for batch in epochBatches {
    let (images, labels) = (batch.data, batch.label)
    let (_, gradients) = valueWithGradient(at: model) { model
    -> Tensor<Float> in
      let logits = model(images)
      return softmaxCrossEntropy(logits: logits, labels: labels)
    }
    optimizer.update(&model, along: gradients)
  }
```

How does this work under the hood? A little bit of calculus mixed together with all of our data. For each training example, we get the raw pixel values (image data) and then the corresponding label (actual number for the picture). Then, we determine the **gradient** for the **model** by calculating the values that the model will predict for X and then see how our prediction compares with the actual value y using a function called softmaxCrossEntropy. Conceptually, softmax just takes a collection of inputs and then normalizes their results across the set as a percentage. This can be a bit complex mathematically, so converting the numbers to use the natural log e and then dividing by the sum of the exponents has the useful dual properties of being consistent across arbitrary inputs and easy to evaluate on a computer. Then, we update our **model** in the direction of that it differs from where it should be slightly (more in the right direction if it's correct, away if it's not). Our learning rate determines how far we should go each pass (e.g., since our rate is .1, we're only going to go 10% of the direction the network thinks is the right one each time). In the for loop that calls all of this, we will repeat this process across all of our data (one pass) for multiple rounds, or **epochs**.

Training loop: Accuracy (5)

Next, we run our model on our test data and calculate how often it was correct on images it hasn't seen yet (but that we know the right answers to). So then, what does accuracy mean, and how do we calculate it? Our code looks like this:

```
  Context.local.learningPhase = .inference
  var testLossSum: Float = 0
  var testBatchCount = 0
  var correctGuessCount = 0
  var totalGuessCount = 0
```

```
  for batch in dataset.validation {
    let (images, labels) = (batch.data, batch.label)
    let logits = model(images)
    testLossSum += softmaxCrossEntropy(logits: logits, labels:
    labels).scalarized()
    testBatchCount += 1

    let correctPredictions = logits.argmax(squeezingAxis: 1) .
    == labels
    correctGuessCount += Int(Tensor<Int32>(correctPredictions).
    sum().scalarized())
    totalGuessCount = totalGuessCount + batch.data.shape[0]
  }

  let accuracy = Float(correctGuessCount) / Float(totalGuessCount)
  print(
    """
    [Epoch \(epoch + 1)] \
    Accuracy: \(correctGuessCount)/\(totalGuessCount)
    (\(accuracy)) \
    Loss: \(testLossSum / Float(testBatchCount))
    """
  )
```

In a similar process to our training dataset, we simply take our test input images, run them through our model, and then compare our results to what we know the right answer to be. Then we literally calculate the number of correct answers divided by the total number of images to produce our accuracy percentage. Our final few lines just print out various numbers each pass through the dataset, or **epoch**, so we can see if our loss is decreasing (e.g., the network is getting more accurate with each pass).

Demo breakdown (low level)

Okay, we've walked through our MNIST example at a high level. Now let's go through some of these functions we're calling and explore our simple training loop more deeply.

Fully connected neural network layers

Fully connected layers form the backbone of our network, so it's worth taking some time to understand them. At a high level, each set of nodes from the input dataset is mapped to the output dataset. Then each edge of the network has a weight that is updated by our training function. The math then for each node is literally [weight] * [input] + [bias], with the value of the output node being the result of this math function. **Weight** is how much value we're going to place on the input to this node, and then **bias** is a constant amount of value assigned to the node regardless of what happens. The values for both of these will be learned by our training. We use matrix math to represent our variables, so that is why each value is in [brackets].

For a single node, the preceding math is simple enough to understand, but the real magic of neural networks comes from many of these nodes firing together. Loosely each neuron learns one part or **feature** of the input, and then by working with the other neurons, they collectively learn the set of weights needed to produce the result we are looking for. The second element of how all this works is that we are combining multiple layers together. The nodes are not learning their values independently, they are learning from other nodes which are updating as well. What this means is that by combining with the idea of working together to figure out when to fire, the neurons are working together to find the most efficient way of representing the input data.

Please note that we are using the word learn very loosely here. The preceding math all works correctly, but people often attribute far more intelligence to this process than actually exists. I believe the best way to think about it is simply to think of your input data as a collection of semi-related samples (e.g., a distribution), and then the neural network is a way of reducing that distribution into an extremely small representation. We will keep on exploring different ways of understanding this key concept.

ReLU is a simple enough function to explain mathematically: relu(x) = max(0, x). All this means is that we return the original value, and then for all values below zero, we just return a zero. There are other choices here (which we will discuss in a future chapter), notably sigmoid functions, but since ReLU produces good results and is so easy to evaluate (and by extension fast), it has become the de facto standard activation function you will find in practice.

How the optimizer works

To continue with the preceding ideas, our goal then is to try to find a set of neurons that will fire together to represent our data. So at a high level, we will show our network our data, then calculate how far our model is from our theoretical result, and then try to move our network slightly closer to being more correct the next time around. This process then is what our optimizer does. If our network guesses correct and is moving in the right direction, then we tell it to keep on going. If our network guesses wrong and is moving in the wrong direction, then we tell it to keep on going in the opposite direction.

The easiest way of representing this is to consider trying to find the minimum of a curve like y = x^2. We can literally take any random point on the curve and calculate the result at another point nearby (a step away, so to speak). Then either one of two possibilities will happen: either we are getting further away from the base (e.g., moving in the wrong direction) or

we are getting closer. Then for our next step, we can either keep on going in the same direction or reverse our course. Either way, we will eventually end up near the bottom of the curve.

To continue the preceding ideas, there are a few problems with our approach. The first is when our step size is too large. Further away from the bottom, this will converge faster, but as we get near the bottom, we will eventually end up in a state where we are jumping from side to side. The flip side of this is choosing too small of a step size and taking a long time to get to the minima, but that isn't too much of a problem normally (if it gets there, it gets there). The next trick then is to add what is called momentum (or second-order gradients). The basic idea is that we don't completely change velocity each step but rather keep our previous motion (e.g., we only add say 10% of the step's change in direction each step).

Optimizers + neural networks

The preceding idea is what is called convex optimization. When dealing with neural networks, though, things are a little more tricky. The first is that by definition we are updating an optimization function for **every** neuron, and so the problem explodes to dealing with many different functions in hyperdimensional space. To the computer, this is nothing more than a very large math problem, but to humans there's no longer a good way to visualize what is going on. This is a large open area of math called nonconvex optimization.

The second problem is simpler: for our math problem, it's easy for us to calculate whether or not we're moving in the right direction because we know what the right answer is. A very large problem in neural networks (especially for more advanced areas) is finding the right goal function for our problem. For what we'll be doing in this book, we'll mostly be using softmax cross-entropy loss. For the problem of image recognition, this is easily represented by comparing our answers with the known results

(e.g., we're just grading things right or not). But constructing custom loss functions is an interesting problem in more advanced uses of neural networks you should be aware of.

Swift for Tensorflow

The preceding text covers the neural network piece. Now, let's look at where swift for tensorflow comes in. The mentioned approach is hopefully reasonably simple enough to understand from a mathematical perspective. The problem with applying it to our neural network problem in a way that scales to larger problem is more complicated. The largest problem is that for real-world networks keeping track of all of our gradients in memory makes updating them much more simple and significantly faster. The second is that when building these models by hand, it is easy to introduce subtle bugs that will create problems down the road. Swift for tensorflow uses Swift's type system to require the layer protocol, as we saw earlier. The basic idea then is simply that we make sure each model enforces this protocol. Then we can add new pieces to the model, and as long as they extend this protocol than in theory, any arbitrary combination of said pieces will work as well. Enforcing this layer protocol forces us, the programmer, to keep our chain of functions correct and by extension allows the compiler to model our gradients in whatever manner it so desires. By extension, then, the compiler can output code for whatever hardware device we have on hand. This is why we are using swift for tensorflow: to get compile-time checking of our networks as well as the ability to run our models on many different hardware back ends using platform-specific optimizations.

Side quests

Here are a couple of simple tweaks you can make to your code in order to understand what is happening:

- Try making the dense layers smaller or larger (e.g., change the 512 in the inputLayer, hiddenLayer, and outputLayer to 128 or 1024), and run things again to see how that affects results.
- Try increasing the number of epochs to 30 and reducing the learning rate to .001 to see how smaller step sizes will still converge to the same result.

Recap

We've looked at how to interact with a simple dataset called MNIST, which is composed of grayscale hand-drawn digits from 0 to 9, ten categories in total. We've built a simple, one-dimensional neural network (called a **multilayer perceptron**) to classify MNIST digits using swift for tensorflow. We've looked at how we can use a statistical technique called **stochastic gradient descent** to update our neural network each time it sees a new image to produce better and better results. We've built a basic but functional training loop that goes through the dataset multiple times, or **epochs**, to train our neural network from an initial random state (where it was essentially guessing) to eventually be able to recognize more than 90% of the digits it is shown.

This is the hardest chapter of the book conceptually. Literally, everything we are going to be doing forward is simply taking this same basic approach and improving it more and more. Spend some time getting everything mentioned down before moving forward. Next, we'll add some convolutions to the neural network we built to produce our first convolutional neural network.

CHAPTER 2

MNIST: 2D Neural Network

In this chapter, we will modify our one-dimensional neural network by adding convolutions to produce our first actual convolutional (2D) neural network and use it to categorize black and white (e.g., MNIST) images again.

Convolutions

Convolutions are a deep area of computer vision theory. At a high level we might think of taking an input image and producing another output image:

```
[cat] --> [magic black box] --> [dog]
```

Broadly, for any input image there's a way to convert it to the target image. At the simplest level we might destroy the source image (e.g., multiply by zero) and then insert our target image (e.g., add its pixels in):

```
[cat] --> 0 * [cat] + [dog] --> [dog]
```

Then, we can model our middle step using simple math:

```
```a[X] + b```
```

This piece of math is called a kernel. This is a convolution, albeit not a terribly useful one.

B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2_2
```

Broadly speaking, for every image in the universe, we can come up with a kernel to convert it into anything else we desire. By extension, there's a kernel for **anything** that you can imagine.

This is a very, very deep area of research in computer vision in general, and there are many different things that can be done here.

3x3 additive blur example

Next, let's look at a slightly more complicated example, a 3x3 additive blur. The actual kernel looks like this:

```
[ 1, 1, 1 ]
[ 1, 1, 1 ]
[ 1, 1, 1 ]
```

What this convolution will do is produce a simple blur to an input image. It does so by literally creating an output pixel for each block of 3x3 pixels in the input image that is the sum of the 9 pixels we are looking at. By then stepping over the row of the input image using a 1 step stride, we end up with a final image that blurred because each output pixel has information from not only the original corresponding pixel but also its neighbors. All of our outputs are larger numbers than we started with. We apply a final simple step to **normalize** the result by dividing all the values by 9 to produce values similar to the original image.

3x3 Gaussian blur example

This next bit you don't need to understand 100%, we're just trying to build upon the concepts.

We can change the 3x3 data and keep the same operation to produce something more complicated. Here's a slightly different multiplicative kernel we can use:

```
[1/16, 1/8, 1/16]
[1/8, 1/4, 1/8]
[1/16, 1/8, 1/16]
```

And we can then produce different results by using our same basic method as earlier. Here, we're taking advantage of matrix multiplication to keep more of our center pixel and less from the ones further away. At a 3x3 size, it's a bit difficult to see the difference between this and our first example, but if you can imagine building larger versions of the above matrix, this is the math that produces larger Gaussian blurs in image editing programs such as Photoshop.

Combined 3x3 convolutions – Sobel filter example

For an even more advanced example of what can be done with convolutions, let's look at combining two of these kernel operations together to produce what is called the Sobel filter. Once again, you don't need to understand this 100%.

Our first kernel looks like this:

```
[1, 0, -1]
[2, 0, -2]
[1, 0, -1]
```

And our second kernel looks like this:

```
[1, 2, 1]
[0, 0, 0]
[-1, -2, -1]
```

And then we combine them together with our input image like so, one after the other:

```
[A] x [B] = [C]
```

The result is interesting; what happens is that pixels that are similar get multiplied to zero (e.g., black), but sets of pixels that have significant differences get multiplied to infinity (e.g., white). So with a couple of basic convolutional kernels we have produced an edge detector! Let's avoid going deeper down the rabbit hole of convolutions for now. Just know that this is a deep, deep field and many things are possible.

3x3 striding

Very broadly, we're going to not actually be building our own convolutions. Instead, we're going to have the neural network learn them! For this, we really only need to focus on one key concept, which is the process of going over our image in these 3x3 blocks. This is called striding, and it's an extremely important concept to understand. Basically, the neural network will learn to make its own convolutions on the fly and then will be using them to better understand our input data, and then each step will be updating them slightly to improve its results. Don't worry, it's a bit mind bendy at first. Let's have the network learn some, and then we can look at how they work on a real-world example.

Padding

"Same" padding and "valid" padding are the two forms of padding you will encounter with convolutions. We will be using the "same" padding for our first few chapters, but "valid" is the default of the 2D convolution operator in swift for tensorflow, and so you will need to understand both.

Valid is perhaps easier to understand. Each stride advances until the far edge of the convolution hits the edge of the input image and then stops. This means that this convolutional type will by definition produce a smaller output than the input image (except for the special case of 1x1 filters). "Same" padding extends the edge of the input data to continue working on the input image until the leading edge of the stride hits the limits of the input image.

This means that "same" padding (when using a stride size of 1) will produce an output image that is the same size as the input image. We're going to use this same padding to jump to some more complicated models in the next few chapters, so focus on understanding that for now.

Maxpool

The other key concept you need to understand is maxpooling. All we're going to do is take each group of 4 input pixels, stepping across our image in strides of two, and convert it to a single output by selecting the largest value. For region, we're simply going to find the largest pixel and make that be our output.

2D MNIST model

If we take these two concepts together and revisit the MNIST problem, we can actually significantly improve our quality just by changing how we're modeling our data. We're going to take our same 784, but we'll treat it as an actual image, so it'll be 28x28 pixels now. We'll run it through two layers of 3x3 convolutions, a maxpool operation, and then we'll keep our same densely connected layers and output of ten categories.

Code

Here's what the actual swift code for this looks like. I've taken the example from before and added a stack of convolutions on top. Then, we take our input, run it through our convolutional layer, and then send it to our same output and densely connected layers as before. This will run a bit, and eventually we'll get up to about 98% accuracy on the MNIST dataset. So by simply changing how we modeled the input data to use convolutions instead, we're able to cut our error rate in half on this toy problem. In addition, convolutions are much easier to evaluate than our dense layers, so as our datasets start getting larger, we'll still be able to continue using this approach.

```
import Datasets
import TensorFlow

struct CNN: Layer {
  var conv1a = Conv2D<Float>(filterShape: (3, 3, 1, 32),
  padding: .same, activation: relu)
  var conv1b = Conv2D<Float>(filterShape: (3, 3, 32, 32),
  padding: .same, activation: relu)
  var pool1 = MaxPool2D<Float>(poolSize: (2, 2), strides: (2, 2))

  var flatten = Flatten<Float>()
  var inputLayer = Dense<Float>(inputSize: 14 * 14 * 32,
  outputSize: 512, activation: relu)
  var hiddenLayer = Dense<Float>(inputSize: 512, outputSize:
  512, activation: relu)
  var outputLayer = Dense<Float>(inputSize: 512, outputSize: 10)

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let convolutionLayer = input.sequenced(through: conv1a,
    conv1b, pool1)
```

```
    return convolutionLayer.sequenced(through: flatten,
    inputLayer, hiddenLayer, outputLayer)
  }
}

let batchSize = 128
let epochCount = 12
var model = CNN()
let optimizer = SGD(for: model, learningRate: 0.1)
let dataset = MNIST(batchSize: batchSize)

print("Starting training...")

for (epoch, epochBatches) in dataset.training.prefix(epochCount).
enumerated() {
  Context.local.learningPhase = .training
  for batch in epochBatches {
    let (images, labels) = (batch.data, batch.label)
    let (_, gradients) = valueWithGradient(at: model) { model
    -> Tensor<Float> in
      let logits = model(images)
      return softmaxCrossEntropy(logits: logits, labels: labels)
    }
    optimizer.update(&model, along: gradients)
  }

  Context.local.learningPhase = .inference
  var testLossSum: Float = 0
  var testBatchCount = 0
  var correctGuessCount = 0
  var totalGuessCount = 0
  for batch in dataset.validation {
    let (images, labels) = (batch.data, batch.label)
```

```
    let logits = model(images)
    testLossSum += softmaxCrossEntropy(logits: logits, labels:
    labels).scalarized()
    testBatchCount += 1

    let correctPredictions = logits.argmax(squeezingAxis: 1) .
    == labels
    correctGuessCount += Int(Tensor<Int32>(correctPredictions).
    sum().scalarized())
    totalGuessCount = totalGuessCount + batch.data.shape[0]
  }

  let accuracy = Float(correctGuessCount) / Float(totalGuessCount)
  print(
    """
    [Epoch \(epoch + 1)] \
    Accuracy: \(correctGuessCount)/\(totalGuessCount) (\(accuracy)) \
    Loss: \(testLossSum / Float(testBatchCount))
    """
  )
}
```

You should have an output that looks something like this:

```
Loading resource: train-images-idx3-ubyte Loading resource:
train-labels-idx1-ubyte Loading resource: t10k-images-idx3-ubyte
Loading resource: t10k-labels-idx1-ubyte Starting training...
[Epoch 1] Accuracy: 9657/10000 (0.9657) Loss: 0.11145979
[Epoch 2] Accuracy: 9787/10000 (0.9787) Loss: 0.06319246
[Epoch 3] Accuracy: 9834/10000 (0.9834) Loss: 0.05008082
[Epoch 4] Accuracy: 9860/10000 (0.986) Loss: 0.041191828
[Epoch 5] Accuracy: 9847/10000 (0.9847) Loss: 0.04551203
```

```
[Epoch 6] Accuracy: 9856/10000 (0.9856) Loss: 0.04516899
[Epoch 7] Accuracy: 9890/10000 (0.989) Loss: 0.036287367
[Epoch 8] Accuracy: 9860/10000 (0.986) Loss: 0.043286547
[Epoch 9] Accuracy: 9878/10000 (0.9878) Loss: 0.037299085
[Epoch 10] Accuracy: 9877/10000 (0.9877) Loss: 0.042443674
[Epoch 11] Accuracy: 9884/10000 (0.9884) Loss: 0.043763407
[Epoch 12] Accuracy: 9890/10000 (0.989) Loss: 0.038426008
```

Side quest

LeNet is the classic approach to solving the MNIST problem, from 1998. We're using a slightly different architecture to simplify making the jump to more advanced models later, but you should take a look at this paper.

> Gradient-based learning applied to document recognition

```
>  http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf
```

Recap

We've looked at how convolutions work with some different examples of what is possible. We've looked at how **striding** and *padding** work to step across an input image. Then, we've looked at **maxpool**, a simple operation to reduce the amount of data we have. Then, we've used two pairs of 3x3 convolutions and a maxpool operation to build our first convolutional neural network for image recognition on top of our multilayer perceptron from the last chapter. Running the same training loop as before, we're able to reduce the amount of error in our simple network, increasing our accuracy simply by changing how we're modeling our input data. Next, let's look at how we can expand our same basic approach to tackle color images and real-world data.

CHAPTER 3

CIFAR: 2D Neural Network with Blocks

In this chapter, we will look at how we can stack layers of convolutions to scale up our network to tackle a slightly more real-world problem of distinguishing between color pictures of animals and vehicles, called CIFAR.

CIFAR dataset

Where do we go from here? Let's take on a slightly larger, more complicated problem. This is a dataset called CIFAR. It's a collection of color pictures. So we have pictures of cats, dogs, animals, as well as human vehicles – cars and trucks. We have ten categories. Now we're going to be working with color data, so we have an RGB component.

Color

Color, from a neural network perspective, turns out to not be as complicated a problem as you might think. Conceptually, we simply take our first 3x3 convolution from our MNIST network, like so:

```
var conv1a = Conv2D<Float>(filterShape: (3, 3, 1, 32), padding:
.same, activation: relu)
```


B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2_3

And we simply increase our number of input layers to 3, like so:

```
var conv1a = Conv2D<Float>(filterShape: (3, 3, 3, 32), padding:
.same, activation: relu)
```

What's going on here? Literally, we were dealing with color as grayscale values (e.g., Int/255.0) in our MNIST example, so now we're moving to having three grayscale channels for each color component (e.g., red, green, blue). To our convolutional operation, this is simply adding more data for us to work with, but we're simply using the same process as before.

Breakdown

For CIFAR, then, we can take our same basic approach that we used before and scale it up to tackle this problem. So we'll simply take our color input data – three channels by 32x32 pixels. We'll run it through two sets of convolutions, a maxpool, two more sets of convolutions, a maxpool, and our same two densely connected layers, and then we'll have ten categories for our outputs.

Code

Here's what this model looks like. We've done nothing more really than add another stack of convolutions, but now we're working with color and real-world photos.

```
```

```
import Datasets
import TensorFlow

struct CIFARModel: Layer {
  var conv1a = Conv2D<Float>(filterShape: (3, 3, 3, 32),
  padding: .same, activation: relu)
```

```
  var conv1b = Conv2D<Float>(filterShape: (3, 3, 32, 32),
  padding: .same, activation: relu)
  var pool1 = MaxPool2D<Float>(poolSize: (2, 2), strides: (2, 2))

  var conv2a = Conv2D<Float>(filterShape: (3, 3, 32, 64),
  padding: .same, activation: relu)
  var conv2b = Conv2D<Float>(filterShape: (3, 3, 64, 64),
  padding: .same, activation: relu)
  var pool2 = MaxPool2D<Float>(poolSize: (2, 2), strides: (2, 2))

  var flatten = Flatten<Float>()
  var inputLayer = Dense<Float>(inputSize: 8 * 8 * 64,
  outputSize: 512, activation: relu)
  var hiddenLayer = Dense<Float>(inputSize: 512, outputSize:
  512, activation: relu)
  var outputLayer = Dense<Float>(inputSize: 512, outputSize: 10)

  @differentiable
  func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let conv1 = input.sequenced(through: conv1a, conv1b, pool1)
    let conv2 = conv1.sequenced(through: conv2a, conv2b, pool2)
    return conv2.sequenced(through: flatten, inputLayer,
    hiddenLayer, outputLayer)
  }
}

let batchSize = 128
let epochCount = 12
var model = CIFARModel()
let optimizer = SGD(for: model, learningRate: 0.1)
let dataset = CIFAR10(batchSize: batchSize)

print("Starting training...")
```

```
for (epoch, epochBatches) in dataset.training.
prefix(epochCount).enumerated() {
  Context.local.learningPhase = .training
  for batch in epochBatches {
    let (images, labels) = (batch.data, batch.label)
    let (_, gradients) = valueWithGradient(at: model) { model
    -> Tensor<Float> in
      let logits = model(images)
      return softmaxCrossEntropy(logits: logits, labels: labels)
    }
    optimizer.update(&model, along: gradients)
  }

  Context.local.learningPhase = .inference
  var testLossSum: Float = 0
  var testBatchCount = 0
  var correctGuessCount = 0
  var totalGuessCount = 0
  for batch in dataset.validation {
    let (images, labels) = (batch.data, batch.label)
    let logits = model(images)
    testLossSum += softmaxCrossEntropy(logits: logits, labels:
    labels).scalarized()
    testBatchCount += 1

    let correctPredictions = logits.argmax(squeezingAxis: 1) .
    == labels
    correctGuessCount += Int(Tensor<Int32>(correctPredictions).
    sum().scalarized())
    totalGuessCount = totalGuessCount + batch.data.shape[0]
  }

  let accuracy = Float(correctGuessCount) / Float(totalGuessCount)
```

```
  print(
    """
    [Epoch \(epoch + 1)] \
    Accuracy: \(correctGuessCount)/\(totalGuessCount) (\(accuracy)) \
    Loss: \(testLossSum / Float(testBatchCount))
    """
  )
}
```

Results

Our simple model can achieve over 70% accuracy using this simple convolutional stack. This isn't going to win awards any time soon, but this basic approach works. You should see results that look like this:

```
...
[Epoch 1] Accuracy: 4938/10000 (0.4938) Loss: 1.403413
[Epoch 2] Accuracy: 5828/10000 (0.5828) Loss: 1.1972797
[Epoch 3] Accuracy: 6394/10000 (0.6394) Loss: 1.0232711
[Epoch 4] Accuracy: 6857/10000 (0.6857) Loss: 0.92201495
[Epoch 5] Accuracy: 6951/10000 (0.6951) Loss: 0.9035831
[Epoch 6] Accuracy: 6778/10000 (0.6778) Loss: 1.0228367
[Epoch 7] Accuracy: 7082/10000 (0.7082) Loss: 0.95399994
[Epoch 8] Accuracy: 7088/10000 (0.7088) Loss: 1.0445035
[Epoch 9] Accuracy: 7117/10000 (0.7117) Loss: 1.1742744
[Epoch 10] Accuracy: 7183/10000 (0.7183) Loss: 1.347533
[Epoch 11] Accuracy: 7045/10000 (0.7045) Loss: 1.4588598
[Epoch 12] Accuracy: 7132/10000 (0.7132) Loss: 1.5338957
```

Side quest

Studying how color works in the real world and how we perceive light is an interesting area. You should check out CYMK (e.g., print color theory) and then how to compress video (e.g., YUV) color space. Sources of light, be they man-made (e.g., light bulbs, LEDs), monitors (TV/computer), or natural (e.g., fire, stars), lead to various interesting differences (hydrogen spectra, Hubble constant).

Recap

We've made the jump from grayscale to color and switched to a slightly larger, more complicated dataset called **CIFAR**, but other than that, our same approach from the last chapter is roughly the same. In order to better categorize our images, we have added another **block** of convolutions. Then, we've used these multiple convolutional blocks from the last chapter and our same fully connected network from the first chapter to categorize color images of real-world objects (albeit slightly hard to look at because they are small). Next, we'll build a larger version of our same network to tackle larger images and more data.

CHAPTER 4

VGG Network

In this chapter, we will build VGG, a state-of-the-art network from 2014, by making an even larger version of our CIFAR network.

Background: ImageNet

MNIST and CIFAR are popular, commonly cited datasets in academic research as a testbed for new ideas, but in the past few years people have increasingly reached the practical limits of building networks on top of them. Our next dataset is ImageNet, a popular real-world dataset for building and training image recognition and object detection networks. ImageNet has a thousand categories, so the networks we will be working with for the rest of the book will be able to support much larger categorization problems. The dataset proper is about ~1.3 million images scraped from the Internet. In data terms the training dataset is ~147GB and then there is another 7GB of test and validation files. If you go to the ImageNet website (e.g., `http://www.image-net.org`) you can browse some of the categories, which have names like "n01440764." If you compare these numbers to the synnet files, you can figure out what each category corresponds to.

B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2_4

Getting ImageNet

This used to be a slightly complicated affair, but recently the swift-models repository added a nice dataloader for the ImageNet dataset you can work with on your system. But, be warned that you will need a few hundred gigabytes of disk space free to deal with the files (extracting, converting, etc.). Having said that, ImageNet is a bit large for our purposes and so we will be working with a subset so as to not reach the limits of our computers and swift for tensorflow.

Imagenette dataset

Imagenette is a subset of ImageNet from Jeremy Howard of fast.ai, designed to make testing computer vision networks easier. It is specifically the following ten categories: tench, English springer, cassette player, chain saw, church, French horn, garbage truck, gas pump, golf ball, parachute.

There is a second, harder version of Imagenette, another subset of ten categories, called Imagewoof, that is specifically the following ten categories of dog breeds: Australian terrier, Border terrier, Samoyed, Beagle, Shih-Tzu, English foxhound, Rhodesian ridgeback, Dingo, Golden retriever, Old English sheepdog.

We can load both of these datasets from the swift-models repository and swap them in your training scripts. What is nice about using the swift-models loader is that it automates the process of downloading, extracting, and batch resizing the actual ImageNet images (which have semi-random sizes) into a predictable input size (e.g., 224 x 224 pixels).

Data augmentation

A very important topic in image recognition/deep neural networks in general is **data augmentation**, which we are basically going to skip in this book because I would like to avoid making things complicated for people new to this field. But, let us discuss it here briefly before heading onward.

We can imagine increasing the size of our neural network until we have a "perfect" neural network, in that for every image we show it, it gives us the correct result. The key concept is that the optimization function we are using is trying to minimize loss across the dataset we are giving it. So, our optimization function for this "perfect" network has gone to zero (it never makes a mistake), just as we desire. Sounds great, let's write a paper and collect our prizes!

But wait! Before we do so, we might try, say, flipping our cat picture horizontally, and then give this new picture to our neural network. What happens? Basically, we're showing our neural network a picture it's never seen before, and so the results are going to be semi-random at best. It may turn out the "closest" input image (in the neural network's search space) of our flipped cat is a dog picture instead, and so our network will say "dog" when it sees this new picture.

The basic idea of data augmentation (and training deep neural networks in general) then is to make sure that we're not **overfitting** (e.g., getting too close to our training dataset) that we can't **generalize** (e.g., correctly making new predictions on data we've never seen before). There are a few basic approaches:

1) Gather more data! You won't see this as much in academic competitions/purposes, but a lot of real-world machine learning involves getting or building even larger datasets in order to make sure our network isn't really guessing at new conditions but rather has "seen" a reasonably similar example already. In the same vein, another common problem is only giving our network pictures of gray cats, and then when it sees an orange one, it doesn't know what to do. Having a lot of examples doesn't help us much if they're all of the same cat! Another common version of this problem is getting training

examples that are different than what we would like to eventually classify, for example, training off photos from the Internet and then trying to apply them to real-world camera input. Whenever possible, use the same data you are eventually going to be testing with. Likewise, whenever you can, gather more data!

2) Data augmentation: We can use the computer to perform various modifications to our data in order to increase the number of samples we have in general. Some common examples:

   - We can flip our images left to right.
   - Change our brightness (gamma).
   - Rotate our images.
   - Random crops (cutting the edges of pictures off).
   - Random zooms (making our picture larger and then cutting off the now larger edges).

   Often these methods will be combined as well in order to make sure the neural network is getting as many possible variants of our training data as well. As an important note, oftentimes, these approaches become domain specific. Or rather, we can flip cat/dog pictures, but not letters of the alphabet!

3) Add noise to our network: Another extremely important area involves adding noise to our operations in order to make sure our network isn't getting too tied to specific inputs/images. This is an incredibly valuable technique to improve

> real-world performance by making our network robust to noise. There is an important related area of research called adversarial inputs that tries to fool networks by introducing subtle noise to fool classifiers.

Here are some interesting papers on this subject you might look at:

> Dropout: A Simple Way to Prevent Neural Networks from Overfitting

```
> www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf
```

This is an important paper for you to know. By randomly pruning (removing) dense nodes when training our model, the resulting network generalizes much better. The other interesting effect is that this speeds up the network as well.

> mixup: Beyond Empirical Risk Minimization

```
> https://arxiv.org/abs/1710.09412
```

Loosely, we're training our networks to recognize images and giving them a reward when they get the right answer. This paper randomly combines two input images (e.g., 50% dog and 50% cat --> new picture) and rewards the network for guessing a corresponding answer (e.g., 50% dog and 50% cat labels). This simple tweak significantly improves the network's generalization ability.

> Improved Regularization of Convolutional Neural Networks with Cutout

```
> https://arxiv.org/abs/1708.04552
```

This idea is similar to mixup, but we are creating our target image by cutting and pasting images together, with similarly improved results. And then broadly, sometimes it's important to tackle this issue at a higher level, making sure we're not trying to make our network so deep that it always

ends up trying to chase "perfect" solutions but is instead learning just enough to be able to do well in a nontest environment. This is a subtle area where people often get stuck chasing the "perfect" set of parameters, but then their networks don't do as well when working with new data. This is an area where we can spend a bunch of time. We will revisit this later in the book.

VGG

Now, let's get into our first real state-of-the-art convolutional neural network for image recognition. VGG stands for the Visual Geometry Group, a group of computer vision/math–related researchers at the University of Oxford in England.

> "Very Deep Convolutional Networks for Large-Scale Image Recognition"

```
> https://arxiv.org/abs/1409.1556
```

They produced a set of networks (named after their group) that in 2014 were second place on the leaderboard for the ILSVRC competition that year, behind GoogLeNet.

However, don't let this scare you, because their approach isn't anything more technically complicated than what we've looked at so far with our MNIST and CIFAR networks. They built their large neural network by literally stacking the same sets of convolutions we've been looking at in the past few chapters. Their network starts literally the same as what we built before: two sets of 3x3 convolutions, a maxpool, two more sets of 3x3 convolutions, and a maxpool. Next, they keep on stacking layers and adding three more sets of 3x3 convolutions, a maxpool, three more sets of 3x3 convolutions, a maxpool, three more sets of 3x3 convolutions, and a maxpool. Finally, for the output layers, they use two large layers of 4096 fully connected nodes (making their network able to learn much more, so

to speak) and finally have an output layer of a thousand nodes to map to each ImageNet category.

This network is called VGG16 because it has (input) [2 + 2 + 3 + 3 + 3] + 2 (fully connected neural network) + 1 (output) layers. For our purposes, we will only be using ten output nodes at the end (e.g., why our classCount init parameter is 10), to work with our smaller Imagenette dataset, but otherwise everything else is the same.

Code

First, let's look at our training loop, which should look very familiar to our CIFAR and MNIST training loops. The only real difference is that now we are working with a larger dataset. Our next network is a little bit more finicky about training, so we are using SGD with smaller learning rate (update step values) to make sure it trains correctly by not "bouncing" around so much.

```
import Datasets
import TensorFlow

struct VGG16: Layer {
  var conv1a = Conv2D<Float>(filterShape: (3, 3, 3, 64),
  padding: .same, activation: relu)
  var conv1b = Conv2D<Float>(filterShape: (3, 3, 64, 64),
  padding: .same, activation: relu)
  var pool1 = MaxPool2D<Float>(poolSize: (2, 2), strides: (2, 2))

  var conv2a = Conv2D<Float>(filterShape: (3, 3, 64, 128),
  padding: .same, activation: relu)
  var conv2b = Conv2D<Float>(filterShape: (3, 3, 128, 128),
  padding: .same, activation: relu)
  var pool2 = MaxPool2D<Float>(poolSize: (2, 2), strides: (2, 2))
```

```
  var conv3a = Conv2D<Float>(filterShape: (3, 3, 128, 256),
  padding: .same, activation: relu)
  var conv3b = Conv2D<Float>(filterShape: (3, 3, 256, 256),
  padding: .same, activation: relu)
  var conv3c = Conv2D<Float>(filterShape: (3, 3, 256, 256),
  padding: .same, activation: relu)
  var pool3 = MaxPool2D<Float>(poolSize: (2, 2), strides: (2, 2))

  var conv4a = Conv2D<Float>(filterShape: (3, 3, 256, 512),
  padding: .same, activation: relu)
  var conv4b = Conv2D<Float>(filterShape: (3, 3, 512, 512),
  padding: .same, activation: relu)
  var conv4c = Conv2D<Float>(filterShape: (3, 3, 512, 512),
  padding: .same, activation: relu)
  var pool4 = MaxPool2D<Float>(poolSize: (2, 2), strides: (2, 2))

  var conv5a = Conv2D<Float>(filterShape: (3, 3, 512, 512),
  padding: .same, activation: relu)
  var conv5b = Conv2D<Float>(filterShape: (3, 3, 512, 512),
  padding: .same, activation: relu)
  var conv5c = Conv2D<Float>(filterShape: (3, 3, 512, 512),
  padding: .same, activation: relu)
  var pool5 = MaxPool2D<Float>(poolSize: (2, 2), strides: (2, 2))

  var flatten = Flatten<Float>()
  var inputLayer = Dense<Float>(inputSize: 512 * 7 * 7,
  outputSize: 4096, activation: relu)
  var hiddenLayer = Dense<Float>(inputSize: 4096, outputSize:
  4096, activation: relu)
  var outputLayer = Dense<Float>(inputSize: 4096, outputSize: 10)

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let conv1 = input.sequenced(through: conv1a, conv1b, pool1)
```

```
    let conv2 = conv1.sequenced(through: conv2a, conv2b, pool2)
    let conv3 = conv2.sequenced(through: conv3a, conv3b,
    conv3c, pool3)
    let conv4 = conv3.sequenced(through: conv4a, conv4b,
    conv4c, pool4)
    let conv5 = conv4.sequenced(through: conv5a, conv5b,
    conv5c, pool5)
    return conv5.sequenced(through: flatten, inputLayer,
    hiddenLayer, outputLayer)
  }
}

let batchSize = 32
let epochCount = 10

let dataset = Imagenette(batchSize: batchSize, inputSize:
.resized320, outputSize: 224)
var model = VGG16()
let optimizer = SGD(for: model, learningRate: 0.002, momentum: 0.9)

print("Starting training...")

for (epoch, epochBatches) in dataset.training.
prefix(epochCount).enumerated() {
  Context.local.learningPhase = .training
  for batch in epochBatches {
    let (images, labels) = (batch.data, batch.label)
    let (_, gradients) = valueWithGradient(at: model) { model
    -> Tensor<Float> in
      let logits = model(images)
      return softmaxCrossEntropy(logits: logits, labels: labels)
    }
    optimizer.update(&model, along: gradients)
  }
```

```
  Context.local.learningPhase = .inference
  var testLossSum: Float = 0
  var testBatchCount = 0
  var correctGuessCount = 0
  var totalGuessCount = 0
  for batch in dataset.validation {
    let (images, labels) = (batch.data, batch.label)
    let logits = model(images)
    testLossSum += softmaxCrossEntropy(logits: logits, labels:
    labels).scalarized()
    testBatchCount += 1

    let correctPredictions = logits.argmax(squeezingAxis: 1)
                             .== labels
    correctGuessCount += Int(Tensor<Int32>(correctPredictions).
    sum().scalarized())
    totalGuessCount = totalGuessCount + batch.label.shape[0]
  }

  let accuracy = Float(correctGuessCount) / Float(totalGuessCount)
  print(
    """
    [Epoch \(epoch+1)] \
    Accuracy: \(correctGuessCount)/\(totalGuessCount) (\(accuracy)) \
    Loss: \(testLossSum / Float(testBatchCount))
    """
  )
}
```

Results

Running this network on the Imagenette dataset should produce results that look like this:

```
[Epoch 1 ] Accuracy: 125/500 (0.25)  Loss: 2.290163
[Epoch 2 ] Accuracy: 170/500 (0.34)  Loss: 1.8886051
[Epoch 3 ] Accuracy: 205/500 (0.41)  Loss: 1.6971107
[Epoch 4 ] Accuracy: 243/500 (0.486) Loss: 1.5611153
[Epoch 5 ] Accuracy: 257/500 (0.514) Loss: 1.43015
[Epoch 6 ] Accuracy: 290/500 (0.58)  Loss: 1.2774785
[Epoch 7 ] Accuracy: 67/500  (0.534) Loss: 1.3170111
[Epoch 8 ] Accuracy: 309/500 (0.618) Loss: 1.1680012
[Epoch 9 ] Accuracy: 299/500 (0.598) Loss: 1.403522
[Epoch 10] Accuracy: 303/500 (0.606) Loss: 1.40440996
```

Memory usage

With VGG16, you may hit the memory limits of your system. Remember that you may need to change the batch size to 16 (or even less) to fit your dataset cleanly into memory for your GPU. A good thing to practice is starting a job, then opening a new shell session using tmux, and running ```nvidia-smi -l 5``` to watch how the device fills up the memory at the start of a job.

Before we go too much further, let's look at one other important issue you're going to run into at some point in time in general and definitely

with VGG, running out of memory with swift for tensorflow. Set your batch size to 128, run your code, and wait a little bit:

```
Fatal error: OOM when allocating tensor with
shape[128,64,224,224] and type float on /job:localhost/
replica:0/task:0/device:GPU:0 by allocator GPU_0_bfc: file /
home/skoonce/swift/swift-source/tensorflow-swift-apis/Sources/
TensorFlow/Bindings/EagerExecution.swift, line 300 Current stack trace:
0 libswiftCore.so 0x00007fcb746f6c40
swift_reportError + 5
0
1 libswiftCore.so 0x00007fcb74767590
_swift_stdlib_reportF atalErrorInFile + 115
2 libswiftCore.so 0x00007fcb7445c53e
<unavailable> + 14554
22
3 libswiftCore.so 0x00007fcb7445c147
<unavailable> + 14544

33 libswiftCore.so 0x00007fcb745fc310 valueWithPullback<A,
B>(at:in:) + 106
34 libswiftTensorFlow.so 0x00007fcb74bb9e20
valueWithGradient<A, B>(at:in:) + 1073
35 VGG-Imagewoof 0x000055a5370311ed
<unavailable> + 46453
57
36 libc.so.6 0x00007fcb5d6d6ab0
libc_start_main + 231
37 VGG-Imagewoof 0x000055a536bf90ba
<unavailable> + 2213$0
Illegal instruction (core dumped)
```

The Imagenette dataset we are working with here is using about 16GB of primary memory. If you have a GPU with 8GB of RAM, you may need to reduce your batch size during the next few chapters to avoid the dreaded OOM (see previous text). Cutting it in half should make it easy for you to work with the larger dataset as needed, but for some of the larger networks, you may need to use even smaller batch sizes.

I would encourage you to play with different batch sizes and run nvidia-smi for each one to get a feel for how these concepts are related. This is an important skill to pick up in general in my opinion because it will enable you to scale your workloads up and down for devices with more/less memory. Swift for tensorflow in particular is currently a bit "globby" in that it seems to grab things in multigigabyte increments, so learning this with s4tf won't be as easy as with other machine learning frameworks, but knowing how to tune your workload for your device is a valuable skill that you will need for some time yet in this field (and other software packages as well).

Model refactoring

At some point, we're going to hit the limits of what we can accomplish by simply copying and pasting more layers to produce larger and larger neural networks. Now is a good time to look at how we can scale our approach by using a slightly more sophisticated programmatic approach. First, let's do some refactoring and learn about how we can combine multiple layers together to reduce the amount of duplicate code.

VGG16 with subblocks

What's going on here? Basically, we're building some smaller blocks, so that then we reduce the amount of duplicate code in our main network.

Since all of our VGG network blocks look the same (N 3x3 layers + a maxpool), we can define them programmatically.

```
struct VGGBlock2: Layer {
  var conv1a: Conv2D<Float>
  var conv1b: Conv2D<Float>
  var pool1 = MaxPool2D<Float>(poolSize: (2, 2), strides: (2, 2))

  init(featureCounts: (Int, Int)) {
    conv1a = Conv2D(filterShape: (3, 3, featureCounts.0,
    featureCounts.1), padding: .same, activation: relu)
    conv1b = Conv2D(filterShape: (3, 3, featureCounts.1,
    featureCounts.1), padding: .same, activation: relu)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    return input.sequenced(through: conv1a, conv1b, pool1)
  }
}

struct VGGBlock3: Layer {
  var conva: Conv2D<Float>
  var convb: Conv2D<Float>
  var convc: Conv2D<Float>
  var pool = MaxPool2D<Float>(poolSize: (2, 2), strides: (2, 2))

  init(featureCounts: (Int, Int)) {
    conva = Conv2D(filterShape: (3, 3, featureCounts.0,
    featureCounts.1), padding: .same, activation: relu)
    convb = Conv2D(filterShape: (3, 3, featureCounts.1,
    featureCounts.1), padding: .same, activation: relu)
```

```
    convc = Conv2D(filterShape: (3, 3, featureCounts.1,
    featureCounts.1), padding: .same, activation: relu)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    return input.sequenced(through: conva, convb, convc, pool)
  }
}

struct VGG16: Layer {
  var layer1 = VGGBlock2(featureCounts: (3, 64))
  var layer2 = VGGBlock2(featureCounts: (64, 128))
  var layer3 = VGGBlock3(featureCounts: (128, 256))
  var layer4 = VGGBlock3(featureCounts: (256, 512))
  var layer5 = VGGBlock3(featureCounts: (512, 512))

  var flatten = Flatten<Float>()
  var inputLayer = Dense<Float>(inputSize: 512 * 7 * 7,
  outputSize: 4096, activation: relu)
  var hiddenLayer = Dense<Float>(inputSize: 4096, outputSize:
  4096, activation: relu)
  var outputLayer = Dense<Float>(inputSize: 4096, outputSize: 10)

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let backbone = input.sequenced(through: layer1, layer2,
    layer3, layer4, layer5)
    return backbone.sequenced(through: flatten, inputLayer,
    hiddenLayer, outputLayer)
  }
}
```

Side quests

AlexNet has a somewhat unorthodox structure by modern standards so I have deliberately skipped it in this book, but it is an important paper to read for historical reasons.

> ImageNet Classification with Deep Convolutional Neural Networks

```
> https://papers.nips.cc/paper/4824-imagenet-classification-
with-deep-convolutional-neural-networks.pdf
```

Inception v1 (what GoogLeNet is better known as now) has better performance than VGG, but is a much more complicated model. This paper is historically important, but I would suggest you master residual networks first.

> Going Deeper with Convolutions

```
> https://arxiv.org/abs/1409.4842
```

Recap

VGG isn't as popular today as some of the other networks we'll look at shortly, but it's definitely still in use despite being half a decade old. This network is still seen in image processing contexts such as style transfer and as a base of object detection networks. You will also often see retrained VGG networks in domain-specific image recognition problems like face recognition. Congratulations on making it here. You've successfully reproduced your first academic paper! Next, let's look at how we can modify our network slightly to produce even better results.

CHAPTER 5

ResNet 34

In this chapter, we will look at how we can modify the VGG network backbone to produce ResNet 34, a network from 2015. Looking back at our past few chapters, the difference between our 2D MNIST, CIFAR, and VGG networks is simply the number of blocks of 3x3 convolutions. Why stop at this point, though? Let's make even larger networks! Next, we're going to look at the ResNet family of networks, starting with ResNet 34.

Conceptually, we're going to start with a similar base to the VGG network we were just looking at. If the backbone of our VGG network can be thought of as [2, 2, 3, 3, 3] for VGG16, then the backbone of ResNet 34 is [6, 8, 12, 6], with each block being composed of pairs of 3x3 convolutions, exactly the same as the networks we have looked at before. However, we're going to add one more crucial concept, called skip connections.

Skip connections

The magic of residual networks, so to speak, is the addition of what are called residual layers or skip connections.

> Deep Residual Learning for Image Recognition

```
> https://arxiv.org/abs/1512.03385
```


B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2_5

The basic idea is that we add an extra set of paths that hop from each set of layers to the output nodes. This is accomplished at the network level quite simply by adding each set of input layers to the block to the output step.

This is often illustrated as a single set of layers going down the side of the network.

Noise

Conceptually, the problem with the VGG-style approach isn't that we can't build larger and larger networks. We can certainly copy/paste our blocks for some time yet if we have enough GPU memory! The key limitation of VGG-style networks is noise. Each convolution is a destructive operation. If each convolution only loses a tiny piece of information, say 0.1%, that effect over 16 or 19 layers starts to compound since the effect is reapplied in each layer.

So, the first real trick of ResNet is just that these skip connections add each set of layer's input to the eventual layer's output. This gives the network more data with which to find the right combination of convolutional layers for the final prediction step. The second big trick of ResNet is at the end. Since we are sending more data through the network, we can stop using our fully connected layers and instead use an average pooling step to produce the final output.

As compared to our nodes firing together before, the neural network here is effectively learning this output layer in the same manner as our other convolutions, which are much less expensive to compute than fully connected nodes. This means the evaluation of our network suddenly becomes much, much cheaper to do. So, even though we've added more layers to our network and the skip connections mean we're sending more data through the network, this whole network is actually much faster to evaluate than our VGG network.

First, our total number of parameters has dropped significantly (approximately one fourth as many parameters). Also, this add operation is actually very cheap to evaluate in comparison to our fully connected layers. So this means that the network is both smaller and faster.

The first layer of ResNet is a 7x7 convolution, but this is just to break up our input into something smaller for the network. There's been recent research to show that there are better ways of doing the input/head layer (which we'll look at in Chapter 12), so be aware this isn't probably the best approach in general. Having said that, with the hardware limitations of the day, it was a good cheap way to reduce the size of the input down so that the convolutional neural network can do its work.

Batch normalization

> Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

```
> https://arxiv.org/abs/1502.03167
```

Batch normalization is an important training technique for you to know. Conceptually, it works by normalizing the output of a layer against the standard deviation of the data it has seen most recently. When working with random minibatches (what our training loop is doing), this has the useful property of smoothing the gradient space in order to make our backpropagation run much more efficiently. As a result, the network converges much more smoothly and the update process is an order of magnitude faster. Technically speaking, this process also introduces some noise into the training process, and so it is also sometimes considered a regularization technique.

Code

This will be our first large network that makes use of multiple code blocks. The first (head) block is slightly different, so we have specific logic there to deal with the input, and then everything else goes through the middle layers, which are generated programmatically. This is a pattern we will see time and time again from here.

```
import Datasets
import TensorFlow

struct ConvBN: Layer {
  var conv: Conv2D<Float>
  var norm: BatchNorm<Float>

  init(
    filterShape: (Int, Int, Int, Int),
    strides: (Int, Int) = (1, 1),
    padding: Padding = .valid
  ) {
    self.conv = Conv2D(filterShape: filterShape, strides:
    strides, padding: padding)
    self.norm = BatchNorm(featureCount: filterShape.3)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    return input.sequenced(through: conv, norm)
  }
}
```

```
struct ResidualBasicBlock: Layer {
  var layer1: ConvBN
  var layer2: ConvBN

  init(
    featureCounts: (Int, Int, Int, Int),
    kernelSize: Int = 3,
    strides: (Int, Int) = (1, 1)
  ) {
    self.layer1 = ConvBN(
      filterShape: (kernelSize, kernelSize, featureCounts.0,
      featureCounts.1),
      strides: strides,
      padding: .same)
    self.layer2 = ConvBN(
      filterShape: (kernelSize, kernelSize, featureCounts.1,
      featureCounts.3),
      strides: strides,
      padding: .same)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    return layer2(relu(layer1(input)))
  }
}

struct ResidualBasicBlockShortcut: Layer {
  var layer1: ConvBN
  var layer2: ConvBN
  var shortcut: ConvBN
```

```
  init(featureCounts: (Int, Int, Int, Int), kernelSize: Int = 3) {
    self.layer1 = ConvBN(
      filterShape: (kernelSize, kernelSize, featureCounts.0,
      featureCounts.1),
      strides: (2, 2),
      padding: .same)
    self.layer2 = ConvBN(
      filterShape: (kernelSize, kernelSize, featureCounts.1,
      featureCounts.2),
      strides: (1, 1),
      padding: .same)
    self.shortcut = ConvBN(
      filterShape: (1, 1, featureCounts.0, featureCounts.3),
      strides: (2, 2),
      padding: .same)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    return layer2(relu(layer1(input))) + shortcut(input)
  }
}

struct ResNet34: Layer {
  var l1: ConvBN
  var maxPool: MaxPool2D<Float>

  var l2a = ResidualBasicBlock(featureCounts: (64, 64, 64, 64))
  var l2b = ResidualBasicBlock(featureCounts: (64, 64, 64, 64))
  var l2c = ResidualBasicBlock(featureCounts: (64, 64, 64, 64))
```

```
var l3a = ResidualBasicBlockShortcut(featureCounts: (64, 128,
128, 128))
var l3b = ResidualBasicBlock(featureCounts: (128, 128, 128, 128))
var l3c = ResidualBasicBlock(featureCounts: (128, 128, 128, 128))
var l3d = ResidualBasicBlock(featureCounts: (128, 128, 128, 128))

var l4a = ResidualBasicBlockShortcut(featureCounts: (128,
256, 256, 256))
var l4b = ResidualBasicBlock(featureCounts: (256, 256, 256, 256))
var l4c = ResidualBasicBlock(featureCounts: (256, 256, 256, 256))
var l4d = ResidualBasicBlock(featureCounts: (256, 256, 256, 256))
var l4e = ResidualBasicBlock(featureCounts: (256, 256, 256, 256))
var l4f = ResidualBasicBlock(featureCounts: (256, 256, 256, 256))

var l5a = ResidualBasicBlockShortcut(featureCounts: (256,
512, 512, 512))
var l5b = ResidualBasicBlock(featureCounts: (512, 512, 512, 512))
var l5c = ResidualBasicBlock(featureCounts: (512, 512, 512, 512))

var avgPool: AvgPool2D<Float>
var flatten = Flatten<Float>()
var classifier: Dense<Float>

init() {
  l1 = ConvBN(filterShape: (7, 7, 3, 64), strides: (2, 2),
  padding: .same)
  maxPool = MaxPool2D(poolSize: (3, 3), strides: (2, 2))
  avgPool = AvgPool2D(poolSize: (7, 7), strides: (7, 7))
  classifier = Dense(inputSize: 512, outputSize: 10)
}

@differentiable
public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
  let inputLayer = maxPool(relu(l1(input)))
```

```
    let level2 = inputLayer.sequenced(through: l2a, l2b, l2c)
    let level3 = level2.sequenced(through: l3a, l3b, l3c, l3d)
    let level4 = level3.sequenced(through: l4a, l4b, l4c, l4d,
    l4e, l4f)
    let level5 = level4.sequenced(through: l5a, l5b, l5c)
    return level5.sequenced(through: avgPool, flatten, classifier)
  }
}

let batchSize = 32
let epochCount = 30

let dataset = Imagenette(batchSize: batchSize, inputSize:
.resized320, outputSize: 224)
var model = ResNet34()
let optimizer = SGD(for: model, learningRate: 0.002, momentum: 0.9)

print("Starting training...")

for (epoch, epochBatches) in dataset.training.
prefix(epochCount).enumerated() {
  Context.local.learningPhase = .training
  for batch in epochBatches {
    let (images, labels) = (batch.data, batch.label)
    let (_, gradients) = valueWithGradient(at: model) { model
    -> Tensor<Float> in
      let logits = model(images)
      return softmaxCrossEntropy(logits: logits, labels: labels)
    }
    optimizer.update(&model, along: gradients)
  }
```

```
  Context.local.learningPhase = .inference
  var testLossSum: Float = 0
  var testBatchCount = 0
  var correctGuessCount = 0
  var totalGuessCount = 0
  for batch in dataset.validation {
    let (images, labels) = (batch.data, batch.label)
    let logits = model(images)
    testLossSum += softmaxCrossEntropy(logits: logits, labels:
    labels).scalarized()
    testBatchCount += 1

    let correctPredictions = logits.argmax(squeezingAxis: 1) .
    == labels
    correctGuessCount += Int(Tensor<Int32>(correctPredictions).
    sum().scalarized())
    totalGuessCount = totalGuessCount + batch.label.shape[0]
  }

  let accuracy = Float(correctGuessCount) / Float(totalGuessCount)
  print(
    """
    [Epoch \(epoch+1)] \
    Accuracy: \(correctGuessCount)/\(totalGuessCount) (\(accuracy)) \
    Loss: \(testLossSum / Float(testBatchCount))
    """
  )
}
```

Results

This network converges extremely well and is much faster to train/evaluate than our VGG network from before due to the use of a simpler convolutional output and residual blocks.

```
Starting training...
[Epoch  1] Accuracy: 217/500 (0.434) Loss: 2.118794
[Epoch  2] Accuracy: 194/500 (0.388) Loss: 2.0524213
[Epoch  3] Accuracy: 295/500 (0.59)  Loss: 1.4818325
[Epoch  4] Accuracy: 177/500 (0.354) Loss: 2.1035159
[Epoch  5] Accuracy: 327/500 (0.654) Loss: 1.0758021
[Epoch  6] Accuracy: 278/500 (0.556) Loss: 1.680953
[Epoch  7] Accuracy: 327/500 (0.654) Loss: 1.3363588
[Epoch  8] Accuracy: 348/500 (0.696) Loss: 1.107703
[Epoch  9] Accuracy: 284/500 (0.568) Loss: 1.9379689
[Epoch 10] Accuracy: 350/500 (0.7)   Loss: 1.2561296
[Epoch 11] Accuracy: 288/500 (0.576) Loss: 1.995267
[Epoch 12] Accuracy: 353/500 (0.706) Loss: 1.2237265
[Epoch 13] Accuracy: 342/500 (0.684) Loss: 1.4842949
[Epoch 14] Accuracy: 374/500 (0.748) Loss: 1.385373
[Epoch 15] Accuracy: 313/500 (0.626) Loss: 2.0999825
[Epoch 16] Accuracy: 368/500 (0.736) Loss: 1.1946388
[Epoch 17] Accuracy: 370/500 (0.74)  Loss: 1.2470249
[Epoch 18] Accuracy: 382/500 (0.764) Loss: 1.1730658
[Epoch 19] Accuracy: 390/500 (0.78)  Loss: 1.1377627
[Epoch 20] Accuracy: 392/500 (0.784) Loss: 1.0375359
[Epoch 21] Accuracy: 371/500 (0.742) Loss: 1.3912839
[Epoch 22] Accuracy: 379/500 (0.758) Loss: 1.2445369
[Epoch 23] Accuracy: 384/500 (0.768) Loss: 1.1650964
[Epoch 24] Accuracy: 365/500 (0.73)  Loss: 1.4282515
[Epoch 25] Accuracy: 361/500 (0.722) Loss: 1.4129665
```

```
[Epoch 26] Accuracy: 376/500 (0.752) Loss: 1.3693335
[Epoch 27] Accuracy: 364/500 (0.728) Loss: 1.4527073
[Epoch 28] Accuracy: 376/500 (0.752) Loss: 1.3168014
[Epoch 29] Accuracy: 363/500 (0.726) Loss: 1.6024143
[Epoch 30] Accuracy: 383/500 (0.766) Loss: 1.1949569
```

Side quest

This is beyond the scope of our book, but this approach has been proven to scale up to extremely large networks. Thousand-layer ResNet networks have been built and successfully trained on the CIFAR dataset. A slightly different variant of this approach is called highway networks and is also worth looking at. This skip connection approach lends itself naturally to combining different blocks together and is the basis of many modern neural network approaches that use residual networks to combine custom block types together to tackle larger and larger problems.

> Highway Networks

```
> https://arxiv.org/abs/1505.00387
```

Recap

We've looked at how we can stack groups of convolutions similar to our VGG network to build a larger convolutional network. Then, by adding residual skip connections between our layer groups, we can make this approach resistant to noise and as a result can achieve an even higher level of accuracy than before. Next, we'll look at how we can modify this approach slightly to produce even better results!

CHAPTER 6

ResNet 50

ResNet 50 is a crucial network for you to understand. It is the basis of much academic research in this field. Many different papers will compare their results to a ResNet 50 baseline, and it is valuable as a reference point. As well, we can easily download the weights for ResNet 50 networks that have been trained on the ImageNet dataset and modify the last layers (called **retraining** or **transfer learning**) to quickly produce models to tackle new problems. For most problems, this is the best approach to get started with, rather than trying to invent new networks or techniques. Building a custom dataset and scaling it up with data augmentation techniques will get you a lot further than trying to build a new architecture.

To continue our thread from the end of the last chapter, the real power of residual networks is that they allow us to build, evaluate, and train much larger networks cheaply. As a result, we no longer need to stick with our 3x3 convolutions but can start to introduce different cell types. So, let us build something even more powerful. We will look at how we can modify ResNet 34 to produce ResNet 50, a solid modern architecture you will encounter repeatedly in this field.

Bottleneck blocks

What we're going to introduce are called bottleneck blocks. Conceptually, we will go from two 3x3 convolutions to a stack that looks like so: 1x1, 3x3, 1x1. From a mathematical perspective, this is actually less powerful

B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2_6

than the 3x3 approach we've used so far. The second thing that bottleneck blocks allow us to do is run more of them, since the 1x1 layers are cheaper to implement. As a result, we can run four times as many filters using these bottleneck layers, which is why I would argue they are ultimately more powerful. Or, to phrase things differently, they're technically less powerful but are also cheaper computationally. This means we can use more of them without significantly increasing our computation budget, for example, the full bottleneck block is approximately 5% more expensive than the ResNet 34 3x3 stack of two blocks. As a result, this network is able to produce even more accurate results than our ResNet 34 network simply by replacing these cells. This is a concept we will explore more in upcoming chapters.

Code

The only real difference between this network and Resnet 34 is converting things to use bottleneck layers and then the larger parameter inputs to the middle stages.

```
import Datasets
import TensorFlow

struct ConvBN: Layer {
  var conv: Conv2D<Float>
  var norm: BatchNorm<Float>

  init(
    filterShape: (Int, Int, Int, Int),
    strides: (Int, Int) = (1, 1),
    padding: Padding = .valid
  ) {
```

```
    self.conv = Conv2D(filterShape: filterShape, strides:
    strides, padding: padding)
    self.norm = BatchNorm(featureCount: filterShape.3)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    return input.sequenced(through: conv, norm)
  }
}

struct ResidualConvBlock: Layer {
  var layer1: ConvBN
  var layer2: ConvBN
  var layer3: ConvBN
  var shortcut: ConvBN

  init(
    featureCounts: (Int, Int, Int, Int),
    kernelSize: Int = 3,
    strides: (Int, Int) = (2, 2)
  ) {
    self.layer1 = ConvBN(
      filterShape: (1, 1, featureCounts.0, featureCounts.1),
      strides: strides)
    self.layer2 = ConvBN(
      filterShape: (kernelSize, kernelSize, featureCounts.1,
      featureCounts.2),
      padding: .same)
    self.layer3 = ConvBN(filterShape: (1, 1, featureCounts.2,
    featureCounts.3))
    self.shortcut = ConvBN(
```

```
      filterShape: (1, 1, featureCounts.0, featureCounts.3),
      strides: strides,
      padding: .same)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let tmp = relu(layer2(relu(layer1(input))))
    return relu(layer3(tmp) + shortcut(input))
  }
}

struct ResidualIdentityBlock: Layer {
  var layer1: ConvBN
  var layer2: ConvBN
  var layer3: ConvBN

  init(featureCounts: (Int, Int, Int, Int), kernelSize: Int = 3) {
    self.layer1 = ConvBN(filterShape: (1, 1, featureCounts.0,
    featureCounts.1))
    self.layer2 = ConvBN(
      filterShape: (kernelSize, kernelSize, featureCounts.1,
      featureCounts.2),
      padding: .same)
    self.layer3 = ConvBN(filterShape: (1, 1, featureCounts.2,
    featureCounts.3))
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let tmp = relu(layer2(relu(layer1(input))))
    return relu(layer3(tmp) + input)
  }
}
```

```
struct ResNet50: Layer {
  var l1: ConvBN
  var maxPool: MaxPool2D<Float>

  var l2a = ResidualConvBlock(featureCounts: (64, 64, 64, 256),
  strides: (1, 1))
  var l2b = ResidualIdentityBlock(featureCounts: (256, 64, 64, 256))
  var l2c = ResidualIdentityBlock(featureCounts: (256, 64, 64, 256))

  var l3a = ResidualConvBlock(featureCounts: (256, 128, 128, 512))
  var l3b = ResidualIdentityBlock(featureCounts: (512, 128,
  128, 512))
  var l3c = ResidualIdentityBlock(featureCounts: (512, 128,
  128, 512))
  var l3d = ResidualIdentityBlock(featureCounts: (512, 128,
  128, 512))

  var l4a = ResidualConvBlock(featureCounts: (512, 256, 256, 1024))
  var l4b = ResidualIdentityBlock(featureCounts: (1024, 256,
  256, 1024))
  var l4c = ResidualIdentityBlock(featureCounts: (1024, 256,
  256, 1024))
  var l4d = ResidualIdentityBlock(featureCounts: (1024, 256,
  256, 1024))
  var l4e = ResidualIdentityBlock(featureCounts: (1024, 256,
  256, 1024))
  var l4f = ResidualIdentityBlock(featureCounts: (1024, 256,
  256, 1024))

  var l5a = ResidualConvBlock(featureCounts: (1024, 512,
  512, 2048))
  var l5b = ResidualIdentityBlock(featureCounts: (2048, 512,
  512, 2048))
  var l5c = ResidualIdentityBlock(featureCounts: (2048, 512,
  512, 2048))
```

```
  var avgPool: AvgPool2D<Float>
  var flatten = Flatten<Float>()
  var classifier: Dense<Float>

  init() {
    l1 = ConvBN(filterShape: (7, 7, 3, 64), strides: (2, 2),
    padding: .same)
    maxPool = MaxPool2D(poolSize: (3, 3), strides: (2, 2))
    avgPool = AvgPool2D(poolSize: (7, 7), strides: (7, 7))
    classifier = Dense(inputSize: 2048, outputSize: 10)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let inputLayer = maxPool(relu(l1(input)))
    let level2 = inputLayer.sequenced(through: l2a, l2b, l2c)
    let level3 = level2.sequenced(through: l3a, l3b, l3c, l3d)
    let level4 = level3.sequenced(through: l4a, l4b, l4c, l4d,
    l4e, l4f)
    let level5 = level4.sequenced(through: l5a, l5b, l5c)
    return level5.sequenced(through: avgPool, flatten, classifier)
  }
}

let batchSize = 32
let epochCount = 30

let dataset = Imagenette(batchSize: batchSize, inputSize:
.resized320, outputSize: 224)
var model = ResNet50()
let optimizer = SGD(for: model, learningRate: 0.002, momentum: 0.9)

print("Starting training...")
```

```
for (epoch, epochBatches) in dataset.training.
prefix(epochCount).enumerated() {
  Context.local.learningPhase = .training
  for batch in epochBatches {
    let (images, labels) = (batch.data, batch.label)
    let (_, gradients) = valueWithGradient(at: model) { model
    -> Tensor<Float> in
      let logits = model(images)
      return softmaxCrossEntropy(logits: logits, labels: labels)
    }
    optimizer.update(&model, along: gradients)
  }

  Context.local.learningPhase = .inference
  var testLossSum: Float = 0
  var testBatchCount = 0
  var correctGuessCount = 0
  var totalGuessCount = 0
  for batch in dataset.validation {
    let (images, labels) = (batch.data, batch.label)
    let logits = model(images)
    testLossSum += softmaxCrossEntropy(logits: logits, labels:
    labels).scalarized()
    testBatchCount += 1

    let correctPredictions = logits.argmax(squeezingAxis: 1) .
    == labels
    correctGuessCount += Int(Tensor<Int32>(correctPredictions).
    sum().scalarized())
    totalGuessCount = totalGuessCount + batch.label.shape[0]
  }
```

```
  let accuracy = Float(correctGuessCount) / Float(totalGuessCount)
  print(
    """
    [Epoch \(epoch+1)] \
   Accuracy: \(correctGuessCount)/\(totalGuessCount) (\(accuracy)) \
    Loss: \(testLossSum / Float(testBatchCount))
    """
  )
}
```

Results

With the above settings, you should be able to get above 75% accuracy on Imagenette, without any data augmentations:

```
[Epoch 20] Accuracy: 362/500 (0.724) Loss: 1.4309547
[Epoch 21] Accuracy: 315/500 (0.63)  Loss: 2.2550986
[Epoch 22] Accuracy: 372/500 (0.744) Loss: 1.4735502
[Epoch 23] Accuracy: 345/500 (0.69)  Loss: 1.9369599
[Epoch 24] Accuracy: 359/500 (0.718) Loss: 2.0183568
[Epoch 25] Accuracy: 337/500 (0.674) Loss: 2.2227683
[Epoch 26] Accuracy: 369/500 (0.738) Loss: 1.4570786
[Epoch 27] Accuracy: 380/500 (0.76)  Loss: 1.3399329
[Epoch 28] Accuracy: 377/500 (0.754) Loss: 1.4157851
[Epoch 29] Accuracy: 357/500 (0.714) Loss: 1.8361444
[Epoch 30] Accuracy: 377/500 (0.754) Loss: 1.3033926
```

Side Quest: ImageNet

Here is how we would train a ResNet50 network on the ImageNet dataset using Swift for TensorFlow, stochastic gradient descent, and the TrainingLoop API:

```
import Datasets
import ImageClassificationModels
import TensorFlow
import TrainingLoop

// XLA mode can't load ImageNet, need to use eager mode to
limit memory use
let device = Device.defaultTFEager
let dataset = ImageNet(batchSize: 32, outputSize: 224, on: device)
var model = ResNet(classCount: 1000, depth: .resNet50)

// 0.1 for 30, .01 for 30, .001 for 30
let optimizer = SGD(for: model, learningRate: 0.1, momentum: 0.9)
public func scheduleLearningRate<L: TrainingLoopProtocol>(
  _ loop: inout L, event: TrainingLoopEvent
) throws where L.Opt.Scalar == Float {
  if event == .epochStart {
    guard let epoch = loop.epochIndex else  { return }
    if epoch > 30 { loop.optimizer.learningRate = 0.01 }
    if epoch > 60 { loop.optimizer.learningRate = 0.001 }
    if epoch > 80 { loop.optimizer.learningRate = 0.0001 }
  }
}

var trainingLoop = TrainingLoop(
  training: dataset.training,
  validation: dataset.validation,
  optimizer: optimizer,
  lossFunction: softmaxCrossEntropy,
  metrics: [.accuracy],
  callbacks: [scheduleLearningRate])

try! trainingLoop.fit(&model, epochs: 90, on: device)
```

As a note, the swift-models import is pulling in ResNet v1.5, the more common variant of Resnet that you will find in practice. The key difference that the 2x2 stride is moved from the first ConvBN in each group to the second one. Another paper from He et al is "Identity Mappings in Deep Residual Networks" (https://arxiv.org/abs/1603.05027), which is sometimes referred to as ResNet v2 or Pre-Activated Resnet, with the key difference that the batch normalization/activation steps are done before the convolution operation and the final activation in each group is removed.

Recap

We've taken our ResNet 34 model from the last chapter and modified it slightly by adding bottleneck blocks. Our 3x3 + 3x3 convolutions have been replaced with a 1x1, 3x3, 1x1–style approach where the last 1x1 convolution has four times as many layers. This makes our network larger, which improves results. Crucially, though, this approach is also cheap to evaluate, and so we get our improved results at roughly the same cost in terms of computation.

This residual approach can be combined with many other approaches in this field. Different sets of convolutional approaches (called **cells**) can be combined together using residual stacks to tackle different problems. Many large-scale reinforcement learning techniques (AlphaZero being a notable example) use large stacks of convolutional layers combined with residual networks.

If you only learn one network from this book, I think this is the best one for you to know. We have literally spent the past six chapters building up to this approach. Next, we are going to look at some mobile-specific networks to try and provide roughly similar results to our ResNet 50 network, but at significantly reduced cost in terms of size and complexity. Next, we will try to reduce the size of our network significantly in order to build networks that will run in resource-constrained environments.

CHAPTER 7

SqueezeNet

For the next few chapters, we're going to look at convolutional neural networks designed specifically for running on mobile devices, primarily phones. A lot of research has gone into building more complicated models using larger and larger clusters of computers to try and increase accuracy on the ImageNet problem. Mobile phones/edge devices are an area of machine learning that has not been explored as deeply, but in my opinion is extremely important. There is the direct goal of getting devices working on real-world devices, but to me what is interesting in particular is the idea that in finding ways of reducing the complexity of high-end approaches to something simpler, we can discover techniques that will allow us to build even larger networks.

Building upon the idea of bottleneck layers from our last chapter, we will sacrifice some of the quality of our network's results to produce SqueezeNet, a tiny neural network that can be run on devices with limited compute power, like phones.

SqueezeNet

A few years back, Cornell University published a paper discussing SqueezeNet and AlexNet-level accuracy.

> SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size

```
> https://arxiv.org/abs/1602.07360
```


B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2_7

The purpose of this paper was to reduce the size of the network as much as possible.

Some of the techniques are not applicable to modern phones, but many of the ideas are of value for you to know. Conceptually, the key thing that SqueezeNet does is use a much more aggressive version of our bottleneck block from the last chapter, called a fire module.

Fire modules

Each fire module takes the input and squeezes it down (e.g., applies a 1x1 convolution at the start) and then expands it in two different ways (e.g., a 3x3 conv and a 1x1 conv in parallel), then concatenates the result of these two expansion layers together to produce the final result. Conceptually, data gets significantly reduced before the second part of the block can learn from it. This is a destructive operation, but on the flip side, it reduces the number of parameters in our network considerably.

> Densely Connected Convolutional Networks

```
> https://arxiv.org/abs/1608.06993
```

Concatenating sets of results together is an interesting way of passing information through the network. Densenet is a paper later the same year that built took the ResNet network approach and used concat operators in place of add operations to produce a new state-of-the-art network (albeit extremely expensive computationally). We will revisit this idea later.

Since we have reduced the amount of data going through our network in general, the other thing that SqueezeNet does is useless maxpool operations in general, so we are performing less of this destructive operation.

Deep compression

Next, the SqueezeNet authors applied the techniques from another paper to make the model as small as possible:

> Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding

```
> https://arxiv.org/abs/1510.00149
```

It is crucial for you to understand the concepts of pruning and quantization in general as model compression techniques. The specific optimizations the authors make on top are valuable to understand as well, but not essential.

Model pruning

Another thing we can do to make models run more quickly is called network pruning. Conceptually, neural networks follow a sort of variant of Zipf's law, whereas 20% of our network activations produce 80% of the results. Therefore, if we're willing to sacrifice accuracy, we can easily make a significantly smaller network by throwing away all but the most popular nodes, called sparsification or pruning.

The "Deep Compression" paper takes this idea, but then retrains the network after performing the sparse step. Interestingly enough, by performing this retraining step, we can get an end network that is as accurate as our input one. Then, by applying a CRC compression scheme (a specific approach to this paper), we can end up a network with an order of magnitude less parameters.

Model quantization

Next, we can convert our 32-bit floats into 8-bit integer weights in order to reduce their size by a factor of 4 again. This is an extremely common step when producing smaller models to run on devices that support quantized math as well as to produce significantly smaller models to ship over the Internet to mobile and embedded devices. In the simplest form of model quantization, floats are represented as a range of +– ~10^38, whereas integer 8 math has a range of –128 to 127, and we simply map the larger float numbers to their nearest normalized integer equivalent. The problem with this approach, though, is that the process of reducing the amount of space available to our network is usually destructive, and so the resulting network doesn't work very well afterward (e.g., things work faster, but with a significant reduction in accuracy).

However, if we have the foresight to incorporate the knowledge that our network is going to be eventually quantized, then we can modify our training process (the technical term is quantized-aware training) to take advantage of this fact. In a similar style to the model pruning step earlier, the "Deep Compression" paper quantizes the network and trains it again in order to minimize the results of the quantization process. In doing so, they were able to eliminate any drop in accuracy, but still end up with a significantly smaller model.

The final paper-specific step the SqueezeNet paper did is utilize what is called Huffman encoding, a compression scheme that is lossless. As a result, they were able to compress the quantized network even more.

Size metric

So, at a high level, this network produces networks that have the same accuracy on ImageNet as AlexNet, a state-of-the-art computer vision network in 2012. By applying their model compression techniques, they were able to reduce the size of AlexNet from 240MB to 6.9MB, with no loss

in accuracy. By using fire modules to produce SqueezeNet, they were able to achieve the same accuracy as AlexNet on the ImageNet dataset with a model that is 4.8MB, a 50x improvement. Then, they were able to apply their model compression techniques to this model to produce a quantized version that is .47MB (under half a megabyte) yet still had an accuracy equivalent to the original model and AlexNet. Conceptually, SqueezeNet is able to achieve the same quality of results as AlexNet, with 510 times less parameters to work with, an impressive accomplishment.

Difference between SqueezeNet 1.0 and 1.1

There are two versions of SqueezeNet in the literature, v1.0 and v1.1. The major difference between the two is in the first layer, which in the 1.0 model used a 7x7 stride and 96 filters compared to the 1.1 model, which uses 3x3 strides and 64 filters.

Code

The following is a demo from 1.1. In it, the v1.1 model moves its maxpool layers higher up the stack (e.g., on layers 1, 3, 5 instead of 1, 4, 8). This produces a network with the same accuracy at approximately 2.4x less operations (e.g., 1.72 Gflops/image for 1.0 vs. 0.72 Gflops/image for 1.1).

```
import TensorFlow

public struct Fire: Layer {
  public var squeeze: Conv2D<Float>
  public var expand1: Conv2D<Float>
  public var expand3: Conv2D<Float>
```

```
  public init(
    inputFilterCount: Int,
    squeezeFilterCount: Int,
    expand1FilterCount: Int,
    expand3FilterCount: Int
  ) {
    squeeze = Conv2D(
      filterShape: (1, 1, inputFilterCount, squeezeFilterCount),
      activation: relu)
    expand1 = Conv2D(
      filterShape: (1, 1, squeezeFilterCount, expand1FilterCount),
      activation: relu)
    expand3 = Conv2D(
      filterShape: (3, 3, squeezeFilterCount, expand3FilterCount),
      padding: .same,
      activation: relu)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let squeezed = squeeze(input)
    let expanded1 = expand1(squeezed)
    let expanded3 = expand3(squeezed)
    return expanded1.concatenated(with: expanded3, alongAxis: -1)
  }
}

public struct SqueezeNet: Layer {
  public var inputConv = Conv2D<Float>(
    filterShape: (3, 3, 3, 64),
    strides: (2, 2),
    padding: .same,
    activation: relu)
```

```
public var maxPool1 = MaxPool2D<Float>(poolSize: (3, 3),
strides: (2, 2))
public var fire2 = Fire(
  inputFilterCount: 64,
  squeezeFilterCount: 16,
  expand1FilterCount: 64,
  expand3FilterCount: 64)
public var fire3 = Fire(
  inputFilterCount: 128,
  squeezeFilterCount: 16,
  expand1FilterCount: 64,
  expand3FilterCount: 64)
public var maxPool3 = MaxPool2D<Float>(poolSize: (3, 3),
strides: (2, 2))
public var fire4 = Fire(
  inputFilterCount: 128,
  squeezeFilterCount: 32,
  expand1FilterCount: 128,
  expand3FilterCount: 128)
public var fire5 = Fire(
  inputFilterCount: 256,
  squeezeFilterCount: 32,
  expand1FilterCount: 128,
  expand3FilterCount: 128)
public var maxPool5 = MaxPool2D<Float>(poolSize: (3, 3),
strides: (2, 2))
public var fire6 = Fire(
  inputFilterCount: 256,
  squeezeFilterCount: 48,
  expand1FilterCount: 192,
  expand3FilterCount: 192)
```

```
  public var fire7 = Fire(
    inputFilterCount: 384,
    squeezeFilterCount: 48,
    expand1FilterCount: 192,
    expand3FilterCount: 192)
  public var fire8 = Fire(
    inputFilterCount: 384,
    squeezeFilterCount: 64,
    expand1FilterCount: 256,
    expand3FilterCount: 256)
  public var fire9 = Fire(
    inputFilterCount: 512,
    squeezeFilterCount: 64,
    expand1FilterCount: 256,
    expand3FilterCount: 256)
  public var outputConv: Conv2D<Float>
  public var avgPool = AvgPool2D<Float>(poolSize: (13, 13),
  strides: (1, 1))

  public init(classCount: Int = 10) {
    outputConv = Conv2D(filterShape: (1, 1, 512, classCount),
    strides: (1, 1), activation: relu)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let convolved1 = input.sequenced(through: inputConv,
    maxPool1)
    let fired1 = convolved1.sequenced(through: fire2, fire3,
    maxPool3, fire4, fire5)
    let fired2 = fired1.sequenced(through: maxPool5, fire6,
    fire7, fire8, fire9)
```

```
    let output = fired2.sequenced(through: outputConv, avgPool)
    return output.reshaped(to: [input.shape[0], outputConv.
    filter.shape[3]])
  }
}
```

Training loop

Put the preceding code into a file named SqueezeNet.swift, and then add a training loop named main.swift:

```
import Datasets
import TensorFlow

let batchSize = 128
let epochCount = 100

let dataset = Imagenette(batchSize: batchSize, inputSize:
.resized320, outputSize: 224)
var model = SqueezeNet()

let optimizer = SGD(for: model, learningRate: 0.0001, momentum:
0.9); print("sgd")
//let optimizer = RMSProp(for: model, learningRate: 0.0001);
print ("rmsprop")
//let optimizer = Adam(for: model, learningRate: 0.0001); print
("adam")

print("Starting training...")

for (epoch, epochBatches) in dataset.training.
prefix(epochCount).enumerated() {
  Context.local.learningPhase = .training
```

```
  for batch in epochBatches {
    let (images, labels) = (batch.data, batch.label)
    let (_, gradients) = valueWithGradient(at: model) { model
    -> Tensor<Float> in
      let logits = model(images)
      return softmaxCrossEntropy(logits: logits, labels: labels)
    }
    optimizer.update(&model, along: gradients)
  }

  Context.local.learningPhase = .inference
  var testLossSum: Float = 0
  var testBatchCount = 0
  var correctGuessCount = 0
  var totalGuessCount = 0
  for batch in dataset.validation {
    let (images, labels) = (batch.data, batch.label)
    let logits = model(images)
    testLossSum += softmaxCrossEntropy(logits: logits, labels:
    labels).scalarized()
    testBatchCount += 1

    let correctPredictions = logits.argmax(squeezingAxis: 1) .
    == labels
    correctGuessCount += Int(Tensor<Int32>(correctPredictions).
    sum().scalarized())
    totalGuessCount = totalGuessCount + batch.label.shape[0]
  }

  let accuracy = Float(correctGuessCount) / Float(totalGuessCount)
```

```
  print(
    """
    [Epoch \(epoch+1)] \
    Accuracy: \(correctGuessCount)/\(totalGuessCount) (\(accuracy)) \
    Loss: \(testLossSum / Float(testBatchCount))
    """
  )
}
```

Going forward, we're just going to swap out models and run things that way. If you need custom training parameters, I'll note them here.

Results

Run your model, and you should end up with results like this:

```
...
[Epoch 95 ] Accuracy: 79/500 (0.158) Loss: 2.3003228
[Epoch 96 ] Accuracy: 78/500 (0.156) Loss: 2.3002906
[Epoch 97 ] Accuracy: 78/500 (0.156) Loss: 2.300246
[Epoch 98 ] Accuracy: 79/500 (0.158) Loss: 2.3002024
[Epoch 99 ] Accuracy: 78/500 (0.156) Loss: 2.3001637
[Epoch 100] Accuracy: 80/500 (0.16)  Loss: 2.3001184
```

Why is our network performing poorly? The reason why we only have 16% accuracy is because SqueezeNet is an extremely difficult network to train. Basic SGD will usually work for building models, but to train this model accurately will require a slightly more sophisticated approach on the optimizer front.

```
> SGD + momentum + (optional) Nesterov smoothing
```

We've not actually been using vanilla SGD so far; we've been using SGD + momentum, which is what's called a second-order method. By keeping track of the current path the network is currently moving toward and then updating based on the inertia of the vector (physics), we have a higher probability of not getting distracted by random updates. Nesterov momentum (which can be enabled with flag "nesterov: true") improves upon this process by mathematically smoothing the function that combines these two.

> RMSProp

```
> www.cs.toronto.edu/~tijmen/csc321/slides/ lecture_slides_
lec6.pdf
```

This made its way into existence from the preceding lecture notes by Geoffrey Hinton and was later written up in a paper by Alex Graves (see https://arxiv.org/abs/1308.0850). Conceptually, we replace the SGD + momentum process by storing the gradient for each vector of the search space, and then the update process is an exponentially decreasing sum of these squared gradients. Since we track multiple vectors, this does well when working with sparse networks.

> Adam

```
> https://arxiv.org/abs/1412.6980
```

Adam and its variants then can be loosely described as combining this concept of momentum with tracking of the gradient space to try and get the best of both worlds. Loosely, there are situations in which either form will have difficulty converging. For momentum-based methods, this occurs when the gradient search subspace is extremely bumpy. Likewise, tracking gradients can hit the so-called vanishing gradient effect, whereby the search process starts going slower and slower and ends up not moving at all.

Anyway, pick one of the non-SGD methods to enable, run the network, and your accuracy should improve significantly:

```
[Epoch 96 ] Accuracy: 378/500 (0.756) Loss: 0.7979737
[Epoch 97 ] Accuracy: 369/500 (0.738) Loss: 0.8244314
[Epoch 98 ] Accuracy: 387/500 (0.774) Loss: 0.74936193
[Epoch 99 ] Accuracy: 377/500 (0.754) Loss: 0.7717642
[Epoch 100] Accuracy: 379/500 (0.758) Loss: 0.7441697
```

For the rest of the book, we'll keep on working with SGD and a small (e.g., 0.002) learning rate, but you should be aware of the preceding optimizers to try in situations in which basic stochastic methods are failing.

Side quest

If you're interested in optimizers, Sebastian Ruder has a nice blog post you should read:

`https://ruder.io/optimizing-gradient-descent/`

Recap

We've looked at SqueezeNet, a computer vision network from 2016 that delivered good results with significantly less cycles and number of parameters than the networks we've looked at so far. Then, we looked at some of the optimizer tweaks sometimes needed to train these smaller networks. Next, let's look at some architectures that are designed around the hardware available on mobile phones.

CHAPTER 8

MobileNet v1

There were some interesting attempts to get smaller models running on device post SqueezeNet. What was needed was a model designed specifically on mobile devices. What a group of researchers at Google produced was called MobileNet, which is an important family of networks for you to understand and where we will be spending a few chapters. At a high level, we will use depthwise separable convolutions to produce an even more accurate network than SqueezeNet that runs well on mobile phone hardware.

MobileNet (v1)

> MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications

```
> https://arxiv.org/abs/1704.04861
```

Model designed specifically to run on mobile hardware, much better use of parameter + data space.

Spatial separable convolutions

Let's look again at our Sobel filter from our chapter where we introduced convolutions. There, we looked at it as being two 3x3 matrix operations. But if we are clever with our math, we can reduce this to a [3x1] and [1x3] multiplication.

B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2_8

This gives us the same result, but has the additional property that it can be computed much more cheaply. Our [3x3] • [3x3] combination ends up requiring nine operations, whereas our [3x1] • [1x3] only requires six operations, a reduction of 33% percent! However, not all kernels can be broken up like this.

Depthwise convolutions

We can exploit one key property in our image data: color. We have three channels – red, green, blue – that we are running through the same sets of filter operations each time we evaluate our neural network.

We can create separate sets of convolutional filters for each area of the input image, combined together by color channel. In academic settings, channels are also referred to as depth, so these are called depthwise convolutions. A variant of this you need to know is increasing the number of filter outputs, which is called a channel multiplier.

Pointwise convolutions

This is only half the puzzle; we still need to combine our channel data back together. In our last chapter on SqueezeNet, we looked at how we can put a 1x1 convolution into our stack as a way to reduce data down significantly before applying our 3x3 convolution. Conceptually, this is called a pointwise convolution because all of the channel input data passes through it. By using these pointwise convolutions, we can map our reduced data space back to our desired final filter size. Then, we simply need to increase the number of pointwise operators to match our desired number of output filters.

Conceptually, we are taking our input image and running groups of depthwise convolutions and then using a stack of small pointwise convolutions to combine them back to our desired output shape. This combination of filters together is called a depthwise separable convolution

and is key to the performance of this network. We have gotten most of the benefits of SqueezeNet's compression approach, but with a less destructive approach than SqueezeNet. In addition, we are now using cheaper operations because depthwise separable convolutions can be accelerated in mobile hardware.

ReLU 6

We have used a ReLU activation function for our models so far, which looks like this:

```
relu(x) = max(features, 0)
```

When building models which we know we are going to quantize, it is valuable to instead limit the output of the ReLU layers and by extension force the network to work with smaller numbers from the start. So, we simply introduce a ceiling function for our ReLU activation like so:

```
relu6(x) = min(max(features, 0), 6)
```

Now, we can simplify our output logic to take advantage of this reduced space.

Example of the reduction in MACs with this approach

> Benchmark Analysis of Representative Deep Neural Network Architectures

```
> https://arxiv.org/abs/1810.00736
```

This paper has a nice graph on page 3 visualizing the differences between these networks. Conceptually, we have a slightly larger network than SqueezeNet, but we have a top 1 accuracy comparable to ResNet 18

(a smaller version of ResNet 34 from earlier). Look at VGG16 vs. MobileNet v2 if you want to know where we're going next.

Code

This network uses many more types of layers than our SqueezeNet approach, but produces significantly better results because they are cheaper computationally. This is something we will see repeatedly going forward.

```
```

```
import TensorFlow

public struct ConvBlock: Layer {
  public var zeroPad = ZeroPadding2D<Float>(padding:
  ((0, 1), (0, 1)))
  public var conv: Conv2D<Float>
  public var batchNorm: BatchNorm<Float>

  public init(filterCount: Int, strides: (Int, Int)) {
    conv = Conv2D<Float>(
      filterShape: (3, 3, 3, filterCount),
      strides: strides,
      padding: .valid)
    batchNorm = BatchNorm<Float>(featureCount: filterCount)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let convolved = input.sequenced(through: zeroPad, conv,
    batchNorm)
    return relu6(convolved)
  }
}
```

```
public struct DepthwiseConvBlock: Layer {
  @noDerivative let strides: (Int, Int)
  @noDerivative public let zeroPad =
  ZeroPadding2D<Float>(padding: ((0, 1), (0, 1)))

  public var dConv: DepthwiseConv2D<Float>
  public var batchNorm1: BatchNorm<Float>
  public var conv: Conv2D<Float>
  public var batchNorm2: BatchNorm<Float>

  public init(
    filterCount: Int, pointwiseFilterCount: Int,
    strides: (Int, Int)
  ) {
    self.strides = strides
    dConv = DepthwiseConv2D<Float>(
      filterShape: (3, 3, filterCount, 1),
      strides: strides,
      padding: strides == (1, 1) ? .same : .valid)
    batchNorm1 = BatchNorm<Float>(
      featureCount: filterCount)
    conv = Conv2D<Float>(
      filterShape: (
        1, 1, filterCount,
        pointwiseFilterCount
      ),
      strides: (1, 1),
      padding: .same)
    batchNorm2 = BatchNorm<Float>(featureCount:
    pointwiseFilterCount)
  }
```

```
  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    var convolved1: Tensor<Float>
    if self.strides == (1, 1) {
      convolved1 = input.sequenced(through: dConv, batchNorm1)
    } else {
      convolved1 = input.sequenced(through: zeroPad, dConv,
      batchNorm1)
    }
    let convolved2 = relu6(convolved1)
    let convolved3 = relu6(convolved2.sequenced(through: conv,
    batchNorm2))
    return convolved3
  }
}

public struct MobileNetV1: Layer {
  @noDerivative let classCount: Int
  @noDerivative let scaledFilterShape: Int

  public var convBlock1: ConvBlock
  public var dConvBlock1: DepthwiseConvBlock
  public var dConvBlock2: DepthwiseConvBlock
  public var dConvBlock3: DepthwiseConvBlock
  public var dConvBlock4: DepthwiseConvBlock
  public var dConvBlock5: DepthwiseConvBlock
  public var dConvBlock6: DepthwiseConvBlock
  public var dConvBlock7: DepthwiseConvBlock
  public var dConvBlock8: DepthwiseConvBlock
  public var dConvBlock9: DepthwiseConvBlock
  public var dConvBlock10: DepthwiseConvBlock
  public var dConvBlock11: DepthwiseConvBlock
  public var dConvBlock12: DepthwiseConvBlock
```

```
public var dConvBlock13: DepthwiseConvBlock
public var avgPool = GlobalAvgPool2D<Float>()
public var dropoutLayer: Dropout<Float>
public var outputConv: Conv2D<Float>

public init(
  classCount: Int = 10,
  dropout: Double = 0.001
) {
  self.classCount = classCount
  scaledFilterShape = Int(1024.0 * 1.0)

  convBlock1 = ConvBlock(filterCount: 32, strides: (2, 2))
  dConvBlock1 = DepthwiseConvBlock(
    filterCount: 32,
    pointwiseFilterCount: 64,
    strides: (1, 1))
  dConvBlock2 = DepthwiseConvBlock(
    filterCount: 64,
    pointwiseFilterCount: 128,
    strides: (2, 2))
  dConvBlock3 = DepthwiseConvBlock(
    filterCount: 128,
    pointwiseFilterCount: 128,
    strides: (1, 1))
  dConvBlock4 = DepthwiseConvBlock(
    filterCount: 128,
    pointwiseFilterCount: 256,
    strides: (2, 2))
  dConvBlock5 = DepthwiseConvBlock(
    filterCount: 256,
    pointwiseFilterCount: 256,
    strides: (1, 1))
```

```
dConvBlock6 = DepthwiseConvBlock(
  filterCount: 256,
  pointwiseFilterCount: 512,
  strides: (2, 2))
dConvBlock7 = DepthwiseConvBlock(
  filterCount: 512,
  pointwiseFilterCount: 512,
  strides: (1, 1))
dConvBlock8 = DepthwiseConvBlock(
  filterCount: 512,
  pointwiseFilterCount: 512,
  strides: (1, 1))
dConvBlock9 = DepthwiseConvBlock(
  filterCount: 512,
  pointwiseFilterCount: 512,
  strides: (1, 1))
dConvBlock10 = DepthwiseConvBlock(
  filterCount: 512,
  pointwiseFilterCount: 512,
  strides: (1, 1))
dConvBlock11 = DepthwiseConvBlock(
  filterCount: 512,
  pointwiseFilterCount: 512,
  strides: (1, 1))
dConvBlock12 = DepthwiseConvBlock(
  filterCount: 512,
  pointwiseFilterCount: 1024,
  strides: (2, 2))
dConvBlock13 = DepthwiseConvBlock(
  filterCount: 1024,
  pointwiseFilterCount: 1024,
  strides: (1, 1))
```

```
    dropoutLayer = Dropout<Float>(probability: dropout)
    outputConv = Conv2D<Float>(
      filterShape: (1, 1, scaledFilterShape, classCount),
      strides: (1, 1),
      padding: .same)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let convolved = input.sequenced(
      through: convBlock1, dConvBlock1,
      dConvBlock2, dConvBlock3, dConvBlock4)
    let convolved2 = convolved.sequenced(
      through: dConvBlock5, dConvBlock6,
      dConvBlock7, dConvBlock8, dConvBlock9)
    let convolved3 = convolved2.sequenced(
      through: dConvBlock10, dConvBlock11, dConvBlock12,
      dConvBlock13, avgPool
    ).reshaped(to: [
      input.shape[0], 1, 1, scaledFilterShape,
    ])
    let convolved4 = convolved3.sequenced(through:
    dropoutLayer, outputConv)
    return convolved4.reshaped(to: [input.shape[0], classCount])
  }
}
```

Results

Our results are on par with our Resnet 50 network from before, but this network is smaller in general and can be evaluated much, much faster at runtime and so is a solid improvement for mobile devices.

```
Starting training...
[Epoch 1 ] Accuracy: 50/500  (0.1)   Loss: 2.5804458
[Epoch 2 ] Accuracy: 262/500 (0.524) Loss: 1.5034955
[Epoch 3 ] Accuracy: 224/500 (0.448) Loss: 1.928577
[Epoch 4 ] Accuracy: 286/500 (0.572) Loss: 1.4074985
[Epoch 5 ] Accuracy: 306/500 (0.612) Loss: 1.3206513
[Epoch 6 ] Accuracy: 334/500 (0.668) Loss: 1.0112444
[Epoch 7 ] Accuracy: 362/500 (0.724) Loss: 0.8360394
[Epoch 8 ] Accuracy: 343/500 (0.686) Loss: 1.0489439
[Epoch 9 ] Accuracy: 317/500 (0.634) Loss: 1.6159635
[Epoch 10] Accuracy: 338/500 (0.676) Loss: 1.0420185
[Epoch 11] Accuracy: 354/500 (0.708) Loss: 1.0034739
[Epoch 12] Accuracy: 358/500 (0.716) Loss: 0.9746185
[Epoch 13] Accuracy: 344/500 (0.688) Loss: 1.152486
[Epoch 14] Accuracy: 365/500 (0.73)  Loss: 0.96197647
[Epoch 15] Accuracy: 353/500 (0.706) Loss: 1.2438473
[Epoch 16] Accuracy: 367/500 (0.734) Loss: 1.044013
[Epoch 17] Accuracy: 365/500 (0.73)  Loss: 1.1098087
[Epoch 18] Accuracy: 352/500 (0.704) Loss: 1.3609929
[Epoch 19] Accuracy: 376/500 (0.752) Loss: 1.2861694
[Epoch 20] Accuracy: 376/500 (0.752) Loss: 1.0280938
[Epoch 21] Accuracy: 369/500 (0.738) Loss: 1.1655327
[Epoch 22] Accuracy: 369/500 (0.738) Loss: 1.1702954
[Epoch 23] Accuracy: 363/500 (0.726) Loss: 1.151112
[Epoch 24] Accuracy: 378/500 (0.756) Loss: 0.94088197
[Epoch 25] Accuracy: 386/500 (0.772) Loss: 1.03443
```

```
[Epoch 26] Accuracy: 379/500 (0.758) Loss: 1.1582794
[Epoch 27] Accuracy: 384/500 (0.768) Loss: 1.1210178
[Epoch 28] Accuracy: 377/500 (0.754) Loss: 1.136668
[Epoch 29] Accuracy: 382/500 (0.764) Loss: 1.2300915
[Epoch 30] Accuracy: 381/500 (0.762) Loss: 1.0231776
```

Recap

We've looked at MobileNet, an important computer vision network from 2017 that makes heavy use of depthwise separable convolutions in order to produce results on par with ResNet 18 (a smaller version of our ResNet 34 network) at a significantly reduced size and computational budget. We can run this on a phone at near real time (e.g., ~50ms/prediction speed) with hardware of the day. Next, let's look at how we can tweak our MobileNet network slightly to produce even better results.

CHAPTER 9

MobileNet v2

In this chapter, we'll look at how we can modify our MobileNet v1 approach to produce MobileNet v2, which is slightly more accurate and computationally cheaper. This network came out in 2018 and delivered an improved version of the v1 architecture.

> MobileNetV2: Inverted Residuals and Linear Bottlenecks

```
> https://arxiv.org/abs/1801.04381
```

The key concepts the Google team introduced in this paper were inverted residual blocks and linear bottleneck layers, so let's look at how they work.

Inverted residual blocks

In our ResNet 50 bottleneck blocks from before, we pass our input layer through a 1x1 convolution in our initial layer of each group, which reduces the data at this point. After passing the data through an expensive 3x3 convolution, we then use a 1x1 convolution to expand the number of filters.

In an inverted residual block, which is what MobileNet v2 uses, we instead use an initial 1x1 convolution to increase our network depth, then apply our depthwise convolution from MobileNet v1, and then use a 1x1 convolution to squeeze our network back down at the end.

B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2_9

Inverted skip connections

In our ResNet networks, we applied our skip connection (e.g., the add operation) to pass data from our input to our output layer. MobileNet v2 does this in a subtly different way by only performing this operation on blocks where the number of inputs and outputs are the same (e.g., not the first block of each stack but between the remaining ones). What this means is that this network is not as strongly connected as the original ResNet and less data passes through, but on the flip side, it is significantly cheaper to evaluate.

Linear bottleneck layers

The next subtle tweak is tied to our inverted skip connections. In the original ResNet network, we apply a ReLU activation function to the combined output of our bottleneck layer and input. Interestingly enough, the MobileNet *v2* authors found that we can eliminate this activation function and improve the network's performance. This activation then simply becomes a linear function, so they call the result a linear bottleneck function.

Code

For this network, we'll use our block operator to generate the sublayers (e.g., InvertedBottleneckBlockStack). Conceptually, the major difference from our MobileNet v1 architecture is the addition of a depthwise conv to our residual blocks and our inverted method of calculating our gradients each pass.

```
```

```
import TensorFlow

public struct InitialInvertedBottleneckBlock: Layer {
  public var dConv: DepthwiseConv2D<Float>
  public var batchNormDConv: BatchNorm<Float>
  public var conv2: Conv2D<Float>
  public var batchNormConv: BatchNorm<Float>

  public init(filters: (Int, Int)) {
    dConv = DepthwiseConv2D<Float>(
      filterShape: (3, 3, filters.0, 1),
      strides: (1, 1),
      padding: .same)
    conv2 = Conv2D<Float>(
      filterShape: (1, 1, filters.0, filters.1),
      strides: (1, 1),
      padding: .same)
    batchNormDConv = BatchNorm(featureCount: filters.0)
    batchNormConv = BatchNorm(featureCount: filters.1)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let depthwise = relu6(batchNormDConv(dConv(input)))
    return batchNormConv(conv2(depthwise))
  }
}

public struct InvertedBottleneckBlock: Layer {
  @noDerivative public var addResLayer: Bool
  @noDerivative public var strides: (Int, Int)
  @noDerivative public let zeroPad =
  ZeroPadding2D<Float>(padding: ((0, 1), (0, 1)))
```

```
  public var conv1: Conv2D<Float>
  public var batchNormConv1: BatchNorm<Float>
  public var dConv: DepthwiseConv2D<Float>
  public var batchNormDConv: BatchNorm<Float>
  public var conv2: Conv2D<Float>
  public var batchNormConv2: BatchNorm<Float>

  public init(
    filters: (Int, Int),
    depthMultiplier: Int = 6,
    strides: (Int, Int) = (1, 1)
  ) {
    self.strides = strides
    self.addResLayer = filters.0 == filters.1 && strides == (1, 1)

    let hiddenDimension = filters.0 * depthMultiplier
    conv1 = Conv2D<Float>(
      filterShape: (1, 1, filters.0, hiddenDimension),
      strides: (1, 1),
      padding: .same)
    dConv = DepthwiseConv2D<Float>(
      filterShape: (3, 3, hiddenDimension, 1),
      strides: strides,
      padding: strides == (1, 1) ? .same : .valid)
    conv2 = Conv2D<Float>(
      filterShape: (1, 1, hiddenDimension, filters.1),
      strides: (1, 1),
      padding: .same)
    batchNormConv1 = BatchNorm(featureCount: hiddenDimension)
    batchNormDConv = BatchNorm(featureCount: hiddenDimension)
    batchNormConv2 = BatchNorm(featureCount: filters.1)
  }
```

```
  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let pointwise = relu6(batchNormConv1(conv1(input)))
    var depthwise: Tensor<Float>
    if self.strides == (1, 1) {
      depthwise = relu6(batchNormDConv(dConv(pointwise)))
    } else {
     depthwise = relu6(batchNormDConv(dConv(zeroPad(pointwise))))
    }
    let pointwiseLinear = batchNormConv2(conv2(depthwise))

    if self.addResLayer {
      return input + pointwiseLinear
    } else {
      return pointwiseLinear
    }
  }
}

public struct InvertedBottleneckBlockStack: Layer {
  var blocks: [InvertedBottleneckBlock] = []

  public init(
    filters: (Int, Int),
    blockCount: Int,
    initialStrides: (Int, Int) = (2, 2)
  ) {
    self.blocks = [
      InvertedBottleneckBlock(
        filters: (filters.0, filters.1),
        strides: initialStrides)
    ]
```

```
    for _ in 1..<blockCount {
      self.blocks.append(
        InvertedBottleneckBlock(
          filters: (filters.1, filters.1))
      )
    }
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    return blocks.differentiableReduce(input) { $1($0) }
  }
}

public struct MobileNetV2: Layer {
  @noDerivative public let zeroPad = ZeroPadding2D<Float>
  (padding: ((0, 1), (0, 1)))
  public var inputConv: Conv2D<Float>
  public var inputConvBatchNorm: BatchNorm<Float>
  public var initialInvertedBottleneck: InitialInverted
  BottleneckBlock

  public var residualBlockStack1: InvertedBottleneckBlockStack
  public var residualBlockStack2: InvertedBottleneckBlockStack
  public var residualBlockStack3: InvertedBottleneckBlockStack
  public var residualBlockStack4: InvertedBottleneckBlockStack
  public var residualBlockStack5: InvertedBottleneckBlockStack

  public var invertedBottleneckBlock16: InvertedBottleneckBlock

  public var outputConv: Conv2D<Float>
  public var outputConvBatchNorm: BatchNorm<Float>
  public var avgPool = GlobalAvgPool2D<Float>()
  public var outputClassifier: Dense<Float>
```

```
public init(classCount: Int = 10) {
  inputConv = Conv2D<Float>(
    filterShape: (3, 3, 3, 32),
    strides: (2, 2),
    padding: .valid)
  inputConvBatchNorm = BatchNorm(
    featureCount: 32)

  initialInvertedBottleneck = InitialInvertedBottleneckBlock(
    filters: (32, 16))

  residualBlockStack1 = InvertedBottleneckBlockStack(filters:
  (16, 24), blockCount: 2)
  residualBlockStack2 = InvertedBottleneckBlockStack(filters:
  (24, 32), blockCount: 3)
  residualBlockStack3 = InvertedBottleneckBlockStack(filters:
  (32, 64), blockCount: 4)
  residualBlockStack4 = InvertedBottleneckBlockStack(
    filters: (64, 96), blockCount: 3,
    initialStrides: (1, 1))
  residualBlockStack5 = InvertedBottleneckBlockStack(filters:
  (96, 160), blockCount: 3)

  invertedBottleneckBlock16 = InvertedBottleneckBlock(filters:
  (160, 320))

  outputConv = Conv2D<Float>(
    filterShape: (1, 1, 320, 1280),
    strides: (1, 1),
    padding: .same)
  outputConvBatchNorm = BatchNorm(featureCount: 1280)

  outputClassifier = Dense(inputSize: 1280, outputSize:
  classCount)
}
```

```
  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let convolved = relu6(input.sequenced(through: zeroPad,
    inputConv, inputConvBatchNorm))
    let initialConv = initialInvertedBottleneck(convolved)
    let backbone = initialConv.sequenced(
      through: residualBlockStack1, residualBlockStack2,
      residualBlockStack3,
      residualBlockStack4, residualBlockStack5)
    let output = relu6(outputConvBatchNorm(outputConv(inverted
    BottleneckBlock16(backbone))))
    return output.sequenced(through: avgPool, outputClassifier)
  }
}
```

Results

This network performs better than our MobileNet v1 architecture using the same training loop and basic setup.

```
Starting training...
[Epoch 1 ] Accuracy: 50/500  (0.1)   Loss: 3.0107288
[Epoch 2 ] Accuracy: 276/500 (0.552) Loss: 1.4318728
[Epoch 3 ] Accuracy: 324/500 (0.648) Loss: 1.2038971
[Epoch 4 ] Accuracy: 337/500 (0.674) Loss: 1.1165649
[Epoch 5 ] Accuracy: 347/500 (0.694) Loss: 0.9973701
[Epoch 6 ] Accuracy: 363/500 (0.726) Loss: 0.9118728
[Epoch 7 ] Accuracy: 310/500 (0.62)  Loss: 1.2533528
[Epoch 8 ] Accuracy: 372/500 (0.744) Loss: 0.797099
[Epoch 9 ] Accuracy: 368/500 (0.736) Loss: 0.8001915
[Epoch 10] Accuracy: 350/500 (0.7)   Loss: 1.1580966
```

```
[Epoch 11] Accuracy: 372/500 (0.744) Loss: 0.84680176
[Epoch 12] Accuracy: 358/500 (0.716) Loss: 1.1446275
[Epoch 13] Accuracy: 388/500 (0.776) Loss: 0.90346915
[Epoch 14] Accuracy: 394/500 (0.788) Loss: 0.82173353
[Epoch 15] Accuracy: 365/500 (0.73)  Loss: 0.9974839
[Epoch 16] Accuracy: 359/500 (0.718) Loss: 1.2463648
[Epoch 17] Accuracy: 333/500 (0.666) Loss: 1.5243211
[Epoch 18] Accuracy: 390/500 (0.78)  Loss: 0.8723967
[Epoch 19] Accuracy: 383/500 (0.766) Loss: 1.0088551
[Epoch 20] Accuracy: 372/500 (0.744) Loss: 1.1002765
[Epoch 21] Accuracy: 392/500 (0.784) Loss: 0.9233314
[Epoch 22] Accuracy: 395/500 (0.79)  Loss: 0.9421617
[Epoch 23] Accuracy: 367/500 (0.734) Loss: 1.1607682
[Epoch 24] Accuracy: 372/500 (0.744) Loss: 1.1685853
[Epoch 25] Accuracy: 375/500 (0.75)  Loss: 1.1443601
[Epoch 26] Accuracy: 389/500 (0.778) Loss: 1.0197723
[Epoch 27] Accuracy: 392/500 (0.784) Loss: 1.0215062
[Epoch 28] Accuracy: 387/500 (0.774) Loss: 1.1886547
[Epoch 29] Accuracy: 400/500 (0.8)   Loss: 0.9691738
[Epoch 30] Accuracy: 383/500 (0.766) Loss: 1.1193326
```

Recap

We've looked at MobileNet v2, a state-of-the-art network from 2018 for performing image recognition on a device with limited computational capacity (e.g., a phone). Next, let's look at how with some reinforcement learning we can get even better results!

CHAPTER 10

EfficientNet

EfficientNet is the current state of the art for image recognition. I doubt this will remain the case forever, but I do not believe it is going to be replaced easily. It is the product of many years' worth of research in this field and combines multiple different techniques together. What is interesting to me in particular about this network is that we are seeing techniques developed for mobile devices having applications in the larger computer vision community. Or rather, research on building models for resource-constrained devices is driving progress in the cloud, while historically the reverse has been the case.

At a high level, EfficientNet was created using the inverted residual blocks of MobileNetV2 as an architecture type combined with the MnasNet search strategy. These smaller blocks weren't around when MnasNet was created, and by using them the researchers were able to find a significantly improved set of networks. In addition, they were able to find a reliably scalable set of heuristics for constructing larger networks given an initial starting point, which was the key limitation of the evolutionary strategies we looked at earlier in the chapter.

In addition, the researchers added two important concepts from other papers: the swish activation function and SE (Squeeze and Excitation) blocks.

B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2_10

Swish

The ReLU function, which we introduced way back in Chapter 1, isn't the only activation function that's been tried. They're just extremely simple to implement and extremely performant, both at the mathematical and hardware levels, and so have stood the test of time, so to speak.

> Searching for Activation Functions

```
> https://arxiv.org/abs/1710.05941
```

This paper explores a variety of alternative activation functions and found that the swish function (discovered in this paper) produces even better results when used in networks.

Swish is defined mathematically as

```
```f(x)=x·sigmoid(βx)```
```sigmoid(y)=1/(1+e^(-y))```
```

Combining these two together has the interesting property of going slightly negative around zero, whereas most traditional activation functions are always >= zero. Conceptually, this produces a smoother gradient space and by extension makes it easier for the network to learn the underlying data distribution, which translates into improved accuracy. Swish has been shown to improve performance in other reinforcement learning problem scenarios, and so it is an important activation function for you to know in general.

There are some limitations to swish from an implementation standpoint, namely, that it uses more memory than a simple ReLU. We will come back to this in the next chapter.

SE (Squeeze + Excitation) block

This is an interesting paper from the Oxford Visual Geometry Group (e.g., the people who produced VGG) from 2017, which won the ImageNet competition that year.

> Squeeze-and-Excitation Networks

```
> https://arxiv.org/abs/1709.01507
```

Conceptually, we might think of what our neural networks are actually learning as a collection of features. Then, when the network sees a picture that matches a particular collection of features, we train it to fire a particular neuron. To take things to the next level and avoid random activations, ideally for each feature map, we could define a sort of master neuron that decides whether or not the feature should activate as a whole.

This is loosely the idea of Squeeze and Excitation blocks. By taking the feature input and reducing it dramatically down (to as small as a single pixel in some cases), we allow the network to sort of train each block to teach itself as to whether or not it should fire given a particular input, so to speak. This produces state-of-the-art results, but is also computationally expensive.

EfficientNet uses a simpler variant based around combining two convolutions to produce similar results at a much cheaper cost computationally.

Code

Pay attention to the squeeze and excite blocks and how they are used to boost the results in the convolutional blocks. With this addition, the rest of this backbone is extremely similar to MobileNet v2. Look also at the subtle differences in the parameters to the MBConvBlockStack generator, which we will see much more of in our next chapter.

```
import TensorFlow

struct InitialMBConvBlock: Layer {
  @noDerivative var hiddenDimension: Int
  var dConv: DepthwiseConv2D<Float>
  var batchNormDConv: BatchNorm<Float>
  var seAveragePool = GlobalAvgPool2D<Float>()
  var seReduceConv: Conv2D<Float>
  var seExpandConv: Conv2D<Float>
  var conv2: Conv2D<Float>
  var batchNormConv2: BatchNorm<Float>

  init(filters: (Int, Int), width: Float) {
    let filterMult = filters
    self.hiddenDimension = filterMult.0
    dConv = DepthwiseConv2D<Float>(
      filterShape: (3, 3, filterMult.0, 1),
      strides: (1, 1),
      padding: .same)
    seReduceConv = Conv2D<Float>(
      filterShape: (1, 1, filterMult.0, 8),
      strides: (1, 1),
      padding: .same)
    seExpandConv = Conv2D<Float>(
      filterShape: (1, 1, 8, filterMult.0),
      strides: (1, 1),
      padding: .same)
    conv2 = Conv2D<Float>(
      filterShape: (1, 1, filterMult.0, filterMult.1),
      strides: (1, 1),
      padding: .same)
```

```
    batchNormDConv = BatchNorm(featureCount: filterMult.0)
    batchNormConv2 = BatchNorm(featureCount: filterMult.1)
  }

  @differentiable
  func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let depthwise = swish(batchNormDConv(dConv(input)))
    let seAvgPoolReshaped = seAveragePool(depthwise).
    reshaped(to: [
      input.shape[0], 1, 1, self.hiddenDimension,
    ])
    let squeezeExcite =
      depthwise
      * sigmoid(seExpandConv(swish(seReduceConv(seAvgPool
      Reshaped))))
    return batchNormConv2(conv2(squeezeExcite))
  }
}

struct MBConvBlock: Layer {
  @noDerivative var addResLayer: Bool
  @noDerivative var strides: (Int, Int)
  @noDerivative let zeroPad = ZeroPadding2D<Float>(padding:
  ((0, 1), (0, 1)))
  @noDerivative var hiddenDimension: Int

  var conv1: Conv2D<Float>
  var batchNormConv1: BatchNorm<Float>
  var dConv: DepthwiseConv2D<Float>
  var batchNormDConv: BatchNorm<Float>
  var seAveragePool = GlobalAvgPool2D<Float>()
  var seReduceConv: Conv2D<Float>
  var seExpandConv: Conv2D<Float>
```

```
  var conv2: Conv2D<Float>
  var batchNormConv2: BatchNorm<Float>

  init(
    filters: (Int, Int),
    width: Float,
    depthMultiplier: Int = 6,
    strides: (Int, Int) = (1, 1),
    kernel: (Int, Int) = (3, 3)
  ) {
    self.strides = strides
    self.addResLayer = filters.0 == filters.1 && strides == (1, 1)

    let filterMult = filters
    self.hiddenDimension = filterMult.0 * depthMultiplier
    let reducedDimension = max(1, Int(filterMult.0 / 4))
    conv1 = Conv2D<Float>(
      filterShape: (1, 1, filterMult.0, hiddenDimension),
      strides: (1, 1),
      padding: .same)
    dConv = DepthwiseConv2D<Float>(
      filterShape: (kernel.0, kernel.1, hiddenDimension, 1),
      strides: strides,
      padding: strides == (1, 1) ? .same : .valid)
    seReduceConv = Conv2D<Float>(
      filterShape: (1, 1, hiddenDimension, reducedDimension),
      strides: (1, 1),
      padding: .same)
    seExpandConv = Conv2D<Float>(
      filterShape: (1, 1, reducedDimension, hiddenDimension),
      strides: (1, 1),
      padding: .same)
```

```
  conv2 = Conv2D<Float>(
    filterShape: (1, 1, hiddenDimension, filterMult.1),
    strides: (1, 1),
    padding: .same)
  batchNormConv1 = BatchNorm(featureCount: hiddenDimension)
  batchNormDConv = BatchNorm(featureCount: hiddenDimension)
  batchNormConv2 = BatchNorm(featureCount: filterMult.1)
}

@differentiable
func forward(_ input: Tensor<Float>) -> Tensor<Float> {
  let piecewise = swish(batchNormConv1(conv1(input)))
  var depthwise: Tensor<Float>
  if self.strides == (1, 1) {
    depthwise = swish(batchNormDConv(dConv(piecewise)))
  } else {
   depthwise = swish(batchNormDConv(dConv(zeroPad(piecewise))))
  }
  let seAvgPoolReshaped = seAveragePool(depthwise).
  reshaped(to: [
    input.shape[0], 1, 1, self.hiddenDimension,
  ])
  let squeezeExcite =
    depthwise
    * sigmoid(seExpandConv(swish(seReduceConv(seAvgPool
    Reshaped))))
  let piecewiseLinear = batchNormConv2(conv2(squeezeExcite))

  if self.addResLayer {
    return input + piecewiseLinear
  } else {
    return piecewiseLinear
```

```
    }
  }
}

struct MBConvBlockStack: Layer {
  var blocks: [MBConvBlock] = []

  init(
    filters: (Int, Int),
    width: Float,
    initialStrides: (Int, Int) = (2, 2),
    kernel: (Int, Int) = (3, 3),
    blockCount: Int,
    depth: Float
  ) {
    let blockMult = blockCount
    self.blocks = [
      MBConvBlock(
        filters: (filters.0, filters.1), width: width,
        strides: initialStrides, kernel: kernel)
    ]
    for _ in 1..<blockMult {
      self.blocks.append(
        MBConvBlock(
          filters: (filters.1, filters.1),
          width: width, kernel: kernel))
    }
  }

  @differentiable
  func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    return blocks.differentiableReduce(input) { $1($0) }
  }
}
```

```
public struct EfficientNet: Layer {
  @noDerivative let zeroPad = ZeroPadding2D<Float>(padding:
  ((0, 1), (0, 1)))
  var inputConv: Conv2D<Float>
  var inputConvBatchNorm: BatchNorm<Float>
  var initialMBConv: InitialMBConvBlock

  var residualBlockStack1: MBConvBlockStack
  var residualBlockStack2: MBConvBlockStack
  var residualBlockStack3: MBConvBlockStack
  var residualBlockStack4: MBConvBlockStack
  var residualBlockStack5: MBConvBlockStack
  var residualBlockStack6: MBConvBlockStack

  var outputConv: Conv2D<Float>
  var outputConvBatchNorm: BatchNorm<Float>
  var avgPool = GlobalAvgPool2D<Float>()
  var dropoutProb: Dropout<Float>
  var outputClassifier: Dense<Float>

  public init(
    classCount: Int = 1000,
    width: Float = 1.0,
    depth: Float = 1.0,
    resolution: Int = 224,
    dropout: Double = 0.2
  ) {
    inputConv = Conv2D<Float>(
      filterShape: (3, 3, 3, 32),
      strides: (2, 2),
      padding: .valid)
    inputConvBatchNorm = BatchNorm(featureCount: 32)
```

```
initialMBConv = InitialMBConvBlock(filters: (32, 16),
width: width)

residualBlockStack1 = MBConvBlockStack(
  filters: (16, 24), width: width,
  blockCount: 2, depth: depth)
residualBlockStack2 = MBConvBlockStack(
  filters: (24, 40), width: width,
  kernel: (5, 5), blockCount: 2, depth: depth)
residualBlockStack3 = MBConvBlockStack(
  filters: (40, 80), width: width,
  blockCount: 3, depth: depth)
residualBlockStack4 = MBConvBlockStack(
  filters: (80, 112), width: width,
  initialStrides: (1, 1), kernel: (5, 5), blockCount: 3,
  depth: depth)
residualBlockStack5 = MBConvBlockStack(
  filters: (112, 192), width: width,
  kernel: (5, 5), blockCount: 4, depth: depth)
residualBlockStack6 = MBConvBlockStack(
  filters: (192, 320), width: width,
  initialStrides: (1, 1), blockCount: 1, depth: depth)

outputConv = Conv2D<Float>(
  filterShape: (
    1, 1,
    320, 1280
  ),
  strides: (1, 1),
  padding: .same)
outputConvBatchNorm = BatchNorm(featureCount: 1280)
```

```
    dropoutProb = Dropout<Float>(probability: dropout)
    outputClassifier = Dense(inputSize: 1280, outputSize:
    classCount)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let convolved = swish(input.sequenced(through: zeroPad,
    inputConv, inputConvBatchNorm))
    let initialBlock = initialMBConv(convolved)
    let backbone = initialBlock.sequenced(
      through: residualBlockStack1, residualBlockStack2,
      residualBlockStack3, residualBlockStack4,
      residualBlockStack5, residualBlockStack6)
    let output = swish(backbone.sequenced(through: outputConv,
    outputConvBatchNorm))
    return output.sequenced(through: avgPool, dropoutProb,
    outputClassifier)
  }
}
```

Results

This network trains extremely well, achieving higher accuracy than any of the networks we have seen so far without the addition of any data augmentation techniques.

```
Starting training...
[Epoch 1 ] Accuracy: 50/500  (0.1)   Loss: 3.919964
[Epoch 2 ] Accuracy: 315/500 (0.63)  Loss: 1.1730766
[Epoch 3 ] Accuracy: 340/500 (0.68)  Loss: 1.042603
```

```
[Epoch 4 ] Accuracy: 382/500 (0.764) Loss: 0.7738381
[Epoch 5 ] Accuracy: 358/500 (0.716) Loss: 0.8867168
[Epoch 6 ] Accuracy: 397/500 (0.794) Loss: 0.7941174
[Epoch 7 ] Accuracy: 384/500 (0.768) Loss: 0.7910826
[Epoch 8 ] Accuracy: 375/500 (0.75)  Loss: 0.9265955
[Epoch 9 ] Accuracy: 395/500 (0.79)  Loss: 0.7806258
[Epoch 10] Accuracy: 389/500 (0.778) Loss: 0.8921993
[Epoch 11] Accuracy: 393/500 (0.786) Loss: 0.913636
[Epoch 12] Accuracy: 395/500 (0.79)  Loss: 0.8772738
[Epoch 13] Accuracy: 396/500 (0.792) Loss: 0.819137
[Epoch 14] Accuracy: 393/500 (0.786) Loss: 0.7435807
[Epoch 15] Accuracy: 418/500 (0.836) Loss: 0.6915679
[Epoch 16] Accuracy: 404/500 (0.808) Loss: 0.79288286
[Epoch 17] Accuracy: 405/500 (0.81)  Loss: 0.8690043
[Epoch 18] Accuracy: 404/500 (0.808) Loss: 0.89440507
[Epoch 19] Accuracy: 409/500 (0.818) Loss: 0.85941887
[Epoch 20] Accuracy: 408/500 (0.816) Loss: 0.8633226
[Epoch 21] Accuracy: 404/500 (0.808) Loss: 0.7646436
[Epoch 22] Accuracy: 411/500 (0.822) Loss: 0.8865621
[Epoch 23] Accuracy: 424/500 (0.848) Loss: 0.6812671
[Epoch 24] Accuracy: 402/500 (0.804) Loss: 0.8662841
[Epoch 25] Accuracy: 425/500 (0.85)  Loss: 0.7081538
[Epoch 26] Accuracy: 423/500 (0.846) Loss: 0.7106852
[Epoch 27] Accuracy: 411/500 (0.822) Loss: 0.88567644
[Epoch 28] Accuracy: 410/500 (0.82)  Loss: 0.8509838
[Epoch 29] Accuracy: 409/500 (0.818) Loss: 0.85791296
[Epoch 30] Accuracy: 416/500 (0.832) Loss: 0.76689
```

EfficientNet variants

Once we have this base, we can then use our improved image recognition network to solve other related problems in different fields.

EfficientNet [B1-8]

To play off our exploration of network architecture search functions in the last chapter, the problem with these sort of approaches is that trying to make them larger is difficult because there's not a clear system for scaling them up.

What the authors introduce in this paper is a set of scaling heuristics for their base (B0) network that enables smooth scaling to produce larger and larger networks. Loosely speaking, we might say that each step of a larger network requires a squared amount of compute. Then, we can build large networks consistently given an extremely large amount of computational time to run on. So, here are EfficientNet variants that can be produced by simply scaling up our prior network compared to the various networks we've looked at so far in this book.

RandAugment

> RandAugment: Practical automated data augmentation with a reduced search space

```
> https://arxiv.org/abs/1909.13719
```

We discussed data augmentation briefly in a prior chapter, and I mentioned that it is an area of active research. This paper combines various augmentation techniques (e.g., flipping, rotating, zooming, etc.) with a reinforcement learning algorithm in order to find the optimal (largest effect on accuracy with the smallest set) combination of data augmentation filters when applied to a dataset. Then, they run this learned

algorithm against the ImageNet dataset and then train the EfficientNet variants on top to produce a significantly (~4–5%!) improved set of networks using nothing more than computational time.

Noisy Student

> Self-training with Noisy Student improves ImageNet classification

> https://arxiv.org/abs/1911.04252

Next, **network distillation** is an interesting area of research for building smaller networks. Loosely, we take a large network as a teacher and then train a smaller student network to give similar responses to the larger one given the same inputs and feedback on each answer from the teacher. This has interesting applications in building networks for devices with limited resources once a larger approach has proven itself on a GPU cluster, for example. The large area where this is of interest in natural language processing, where large networks (e.g., BERT) have achieved a state-of-the-art performance but are too large to be used for day-to-day problem solving.

Network distillation has been used to make networks smaller, but can it be used to make them larger? Loosely speaking, this paper takes data augmentation techniques and uses them to make the student's inputs much more noisy, but keeps on requiring the student network to give answers that match the teacher's answers. By iteratively training a larger student on a teacher and then replacing the teacher with the trained student, they were able to build a much larger network that was able to produce even more accurate ImageNet results than even Facebook's 2019 billion-picture Instagram corpus (see https://arxiv.org/abs/1905.00546).

EfficientDet

> EfficientDet: Scalable and Efficient Object Detection

> https://arxiv.org/abs/1911.09070

We've not talked about object detection networks in this book, but the basic idea of many approaches is to use a known good existing image recognition network (called a **backbone**), and then we can add an object detection output layer at the end (called a **head**). This approach enables a nice sort of mix and match style technique where we can use the same head with multiple different backbones or data augmentation strategies to find the best solution for a particular problem.

So, we take EfficientNet, add a custom object detection head, apply our scaling techniques, and voilà, we have an object detection (and with some other tweaks, semantic segmentation) network with state-of-the-art performance.

Recap

We've looked at EfficientNet, the current state of the art for image recognition. We've looked at how we can use the EfficientNet base to build state-of-the-art approaches in related fields. Next, let's look at how we can take these ideas back to the realm of mobile devices.

CHAPTER 11

MobileNetV3

In this chapter, we will look at a MobileNetV3, which delivers an optimized version of EfficientNet on mobile hardware by reducing the complexity of the network. This model is heavily based on EfficientNet's search strategy with mobile-specific parameter space goals.

This is the current state of the art for mobile models, but at this point, we're deeply into the realm of arguing about what hardware is running things, making 1:1 model comparisons difficult. Manufacturers are increasingly shipping custom hardware, and each device is going to run things slightly differently. The flip side of this though is that the EfficientNet search algorithm can be given an arbitrary starting point (e.g., knowledge of whatever hardware it is going to be run on) and then produce an optimized network for that device. I believe this is increasingly where things are going in the future: as more and more new AI hardware becomes available, then networks will be customized to run on that particular device.

First, let us look at some mobile-specific variants of the swish and sigmoid activation functions we can use to speed up the evaluation of our network.

B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2_11

Hard swish and hard sigmoid

In the last chapter, we discussed how we can use swish and sigmoid as activation functions to make it possible for the network to learn even more accurate results. At runtime, though, these functions are much more expensive in terms of memory than our ReLU activation function. The MobileNet authors introduced a relu6 variant of our sigmoid function:

```
hardSigmoid(x) = relu6(x + 3)/6
hardSwish(x) = x * hardSigmoid(x)
```

in order to reduce the amount of memory required to run the network and simplify the runtime.

However, they found that they couldn't simply apply this to all of the nodes without sacrificing performance. We will come back to this in a second.

Remove the Squeeze and Excitation (SE) block logic for half the network

Likewise, the SEBlock logic from EfficientNet is powerful, but this is an expensive operation on mobile devices. However, they found that they could remove this for some of the layers without sacrificing performance. Once again, we will come back to this in a second.

Custom head

The authors implement a custom head logic for their output layers that I think is interesting. Essentially, they use a pair of convolutions to replace the dense output neural network layer used in EfficientNet. From a technical standpoint, this is less accurate than the dense approach, but is much simpler and faster to implement on a mobile device.

Hyperparameters

And finally, the authors made heavy use of the EfficientNet search strategy combined with the preceding pieces. Conceptually, they gave the search algorithm the mentioned building blocks to work with and a cluster of TPUs run on and let the reinforcement learning do its magic. From this, they produced two different networks, MobileNetV3-Large and MobileNetV3-Small, both of which are subtly different because of the preceding constraints. As an example, while both variants use SEBlocks in the later parts of the network, the small variant uses the SEBlock on its second layer, whereas the large variant does not. The number of filters at each layer is entirely learned to optimize performance. Both networks use ReLU for the first few layers but then switch to hardSwish halfway through.

Performance

Combining all of the above, this network has a higher accuracy on ImageNet but can be evaluated in under 10ms on a mobile device with hardware that supports it. The authors then also ran their search strategy with different starting requirements (e.g., only allowing 3x3 convolutions) to produce a minimal variant which should be reasonably future-proof depending on whatever new hardware comes to market.

Code

Let's build MobileNetV3. This will combine hardware-aware network architecture search (NAS) and the NetAdapt algorithm to take advantage of both approaches. This network is significantly more complicated than the ones we have looked at so far, but if you look carefully, I think you can see it is simply a combination of all of the techniques we have looked at so far. The key section to note is the large collection of MBConvBlockStack

parameters at the end, which generate subtly different blocks which combine together to produce a network that is both accurate and will run well on mobile devices.

```

```
import TensorFlow

public enum ActivationType {
  case hardSwish
  case relu
}

public struct SqueezeExcitationBlock: Layer {
  // https://arxiv.org/abs/1709.01507
  public var averagePool = GlobalAvgPool2D<Float>()
  public var reduceConv: Conv2D<Float>
  public var expandConv: Conv2D<Float>
  @noDerivative public var inputOutputSize: Int

  public init(inputOutputSize: Int, reducedSize: Int) {
    self.inputOutputSize = inputOutputSize
    reduceConv = Conv2D<Float>(
      filterShape: (1, 1, inputOutputSize, reducedSize),
      strides: (1, 1),
      padding: .same)
    expandConv = Conv2D<Float>(
      filterShape: (1, 1, reducedSize, inputOutputSize),
      strides: (1, 1),
      padding: .same)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
```
```

```
    let avgPoolReshaped = averagePool(input).reshaped(to: [
      input.shape[0], 1, 1, self.inputOutputSize,
    ])
    return input
      * hardSigmoid(expandConv(relu(reduceConv(avgPoolReshaped))))
  }
}

public struct InitialInvertedResidualBlock: Layer {
  @noDerivative public var addResLayer: Bool
  @noDerivative public var useSELayer: Bool = false
  @noDerivative public var activation: ActivationType = .relu

  public var dConv: DepthwiseConv2D<Float>
  public var batchNormDConv: BatchNorm<Float>
  public var seBlock: SqueezeExcitationBlock
  public var conv2: Conv2D<Float>
  public var batchNormConv2: BatchNorm<Float>

  public init(
    filters: (Int, Int),
    strides: (Int, Int) = (1, 1),
    kernel: (Int, Int) = (3, 3),
    seLayer: Bool = false,
    activation: ActivationType = .relu
  ) {
    self.useSELayer = seLayer
    self.activation = activation
    self.addResLayer = filters.0 == filters.1 && strides == (1, 1)

    let filterMult = filters
    let hiddenDimension = filterMult.0 * 1
    let reducedDimension = hiddenDimension / 4
```

```
    dConv = DepthwiseConv2D<Float>(
      filterShape: (3, 3, filterMult.0, 1),
      strides: (1, 1),
      padding: .same)
    seBlock = SqueezeExcitationBlock(
      inputOutputSize: hiddenDimension, reducedSize:
      reducedDimension)
    conv2 = Conv2D<Float>(
      filterShape: (1, 1, hiddenDimension, filterMult.1),
      strides: (1, 1),
      padding: .same)
    batchNormDConv = BatchNorm(featureCount: filterMult.0)
    batchNormConv2 = BatchNorm(featureCount: filterMult.1)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    var depthwise = batchNormDConv(dConv(input))
    switch self.activation {
    case .hardSwish: depthwise = hardSwish(depthwise)
    case .relu: depthwise = relu(depthwise)
    }

    var squeezeExcite: Tensor<Float>
    if self.useSELayer {
      squeezeExcite = seBlock(depthwise)
    } else {
      squeezeExcite = depthwise
    }

    let piecewiseLinear = batchNormConv2(conv2(squeezeExcite))
```

```
    if self.addResLayer {
      return input + piecewiseLinear
    } else {
      return piecewiseLinear
    }
  }
}

public struct InvertedResidualBlock: Layer {
  @noDerivative public var strides: (Int, Int)
  @noDerivative public let zeroPad =
  ZeroPadding2D<Float>(padding: ((0, 1), (0, 1)))
  @noDerivative public var addResLayer: Bool
  @noDerivative public var activation: ActivationType = .relu
  @noDerivative public var useSELayer: Bool

  public var conv1: Conv2D<Float>
  public var batchNormConv1: BatchNorm<Float>
  public var dConv: DepthwiseConv2D<Float>
  public var batchNormDConv: BatchNorm<Float>
  public var seBlock: SqueezeExcitationBlock
  public var conv2: Conv2D<Float>
  public var batchNormConv2: BatchNorm<Float>

  public init(
    filters: (Int, Int),
    expansionFactor: Float,
    strides: (Int, Int) = (1, 1),
    kernel: (Int, Int) = (3, 3),
    seLayer: Bool = false,
    activation: ActivationType = .relu
  ) {
    self.strides = strides
```

```
    self.addResLayer = filters.0 == filters.1 && strides == (1, 1)
    self.useSELayer = seLayer
    self.activation = activation

    let filterMult = filters
    let hiddenDimension = Int(Float(filterMult.0) *
    expansionFactor)
    let reducedDimension = hiddenDimension / 4

    conv1 = Conv2D<Float>(
      filterShape: (1, 1, filterMult.0, hiddenDimension),
      strides: (1, 1),
      padding: .same)
    dConv = DepthwiseConv2D<Float>(
      filterShape: (kernel.0, kernel.1, hiddenDimension, 1),
      strides: strides,
      padding: strides == (1, 1) ? .same : .valid)
    seBlock = SqueezeExcitationBlock(
      inputOutputSize: hiddenDimension, reducedSize:
      reducedDimension)
    conv2 = Conv2D<Float>(
      filterShape: (1, 1, hiddenDimension, filterMult.1),
      strides: (1, 1),
      padding: .same)
    batchNormConv1 = BatchNorm(featureCount: hiddenDimension)
    batchNormDConv = BatchNorm(featureCount: hiddenDimension)
    batchNormConv2 = BatchNorm(featureCount: filterMult.1)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    var piecewise = batchNormConv1(conv1(input))
    switch self.activation {
```

```
    case .hardSwish: piecewise = hardSwish(piecewise)
    case .relu: piecewise = relu(piecewise)
    }
    var depthwise: Tensor<Float>
    if self.strides == (1, 1) {
      depthwise = batchNormDConv(dConv(piecewise))
    } else {
      depthwise = batchNormDConv(dConv(zeroPad(piecewise)))
    }
    switch self.activation {
    case .hardSwish: depthwise = hardSwish(depthwise)
    case .relu: depthwise = relu(depthwise)
    }
    var squeezeExcite: Tensor<Float>
    if self.useSELayer {
      squeezeExcite = seBlock(depthwise)
    } else {
      squeezeExcite = depthwise
    }

    let piecewiseLinear = batchNormConv2(conv2(squeezeExcite))

    if self.addResLayer {
      return input + piecewiseLinear
    } else {
      return piecewiseLinear
    }
  }
}

public struct MobileNetV3Large: Layer {
  @noDerivative public let zeroPad = ZeroPadding2D<Float>
(padding: ((0, 1), (0, 1)))
```

```
  public var inputConv: Conv2D<Float>
  public var inputConvBatchNorm: BatchNorm<Float>

  public var invertedResidualBlock1: InitialInvertedResidualBlock
  public var invertedResidualBlock2: InvertedResidualBlock
  public var invertedResidualBlock3: InvertedResidualBlock
  public var invertedResidualBlock4: InvertedResidualBlock
  public var invertedResidualBlock5: InvertedResidualBlock
  public var invertedResidualBlock6: InvertedResidualBlock
  public var invertedResidualBlock7: InvertedResidualBlock
  public var invertedResidualBlock8: InvertedResidualBlock
  public var invertedResidualBlock9: InvertedResidualBlock
  public var invertedResidualBlock10: InvertedResidualBlock
  public var invertedResidualBlock11: InvertedResidualBlock
  public var invertedResidualBlock12: InvertedResidualBlock
  public var invertedResidualBlock13: InvertedResidualBlock
  public var invertedResidualBlock14: InvertedResidualBlock
  public var invertedResidualBlock15: InvertedResidualBlock

  public var outputConv: Conv2D<Float>
  public var outputConvBatchNorm: BatchNorm<Float>

  public var avgPool = GlobalAvgPool2D<Float>()
  public var finalConv: Conv2D<Float>
  public var dropoutLayer: Dropout<Float>
  public var classiferConv: Conv2D<Float>
  public var flatten = Flatten<Float>()

  @noDerivative public var lastConvChannel: Int

  public init(classCount: Int = 1000, dropout: Double = 0.2) {
    inputConv = Conv2D<Float>(
      filterShape: (3, 3, 3, 16),
```

```
  strides: (2, 2),
  padding: .same)
inputConvBatchNorm = BatchNorm(
  featureCount: 16)

invertedResidualBlock1 = InitialInvertedResidualBlock(
  filters: (16, 16))
invertedResidualBlock2 = InvertedResidualBlock(
  filters: (16, 24),
  expansionFactor: 4, strides: (2, 2))
invertedResidualBlock3 = InvertedResidualBlock(
  filters: (24, 24),
  expansionFactor: 3)
invertedResidualBlock4 = InvertedResidualBlock(
  filters: (24, 40),
  expansionFactor: 3, strides: (2, 2), kernel: (5, 5),
  seLayer: true)
invertedResidualBlock5 = InvertedResidualBlock(
  filters: (40, 40),
  expansionFactor: 3, kernel: (5, 5), seLayer: true)
invertedResidualBlock6 = InvertedResidualBlock(
  filters: (40, 40),
  expansionFactor: 3, kernel: (5, 5), seLayer: true)
invertedResidualBlock7 = InvertedResidualBlock(
  filters: (40, 80),
  expansionFactor: 6, strides: (2, 2), activation: .hardSwish)
invertedResidualBlock8 = InvertedResidualBlock(
  filters: (80, 80),
  expansionFactor: 2.5, activation: .hardSwish)
invertedResidualBlock9 = InvertedResidualBlock(
  filters: (80, 80),
  expansionFactor: 184 / 80.0, activation: .hardSwish)
```

```
    invertedResidualBlock10 = InvertedResidualBlock(
      filters: (80, 80),
      expansionFactor: 184 / 80.0, activation: .hardSwish)
    invertedResidualBlock11 = InvertedResidualBlock(
      filters: (80, 112),
      expansionFactor: 6, seLayer: true, activation: .hardSwish)
    invertedResidualBlock12 = InvertedResidualBlock(
      filters: (112, 112),
      expansionFactor: 6, seLayer: true, activation: .hardSwish)
    invertedResidualBlock13 = InvertedResidualBlock(
      filters: (112, 160),
      expansionFactor: 6, strides: (2, 2), kernel: (5, 5),
      seLayer: true,
      activation: .hardSwish)
    invertedResidualBlock14 = InvertedResidualBlock(
      filters: (160, 160),
      expansionFactor: 6, kernel: (5, 5), seLayer: true,
      activation: .hardSwish)
    invertedResidualBlock15 = InvertedResidualBlock(
      filters: (160, 160),
      expansionFactor: 6, kernel: (5, 5), seLayer: true,
      activation: .hardSwish)

    lastConvChannel = 960
    outputConv = Conv2D<Float>(
      filterShape: (
        1, 1, 160, lastConvChannel
      ),
      strides: (1, 1),
      padding: .same)
    outputConvBatchNorm = BatchNorm(featureCount: lastConvChannel)
```

```
  let lastPointChannel = 1280
  finalConv = Conv2D<Float>(
    filterShape: (1, 1, lastConvChannel, lastPointChannel),
    strides: (1, 1),
    padding: .same)
  dropoutLayer = Dropout<Float>(probability: dropout)
  classiferConv = Conv2D<Float>(
    filterShape: (1, 1, lastPointChannel, classCount),
    strides: (1, 1),
    padding: .same)
}

@differentiable
public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
  let initialConv = hardSwish(
    input.sequenced(through: zeroPad, inputConv,
    inputConvBatchNorm))
  let backbone1 = initialConv.sequenced(
    through: invertedResidualBlock1,
    invertedResidualBlock2, invertedResidualBlock3,
    invertedResidualBlock4, invertedResidualBlock5)
  let backbone2 = backbone1.sequenced(
    through: invertedResidualBlock6, invertedResidualBlock7,
    invertedResidualBlock8, invertedResidualBlock9,
    invertedResidualBlock10)
  let backbone3 = backbone2.sequenced(
    through: invertedResidualBlock11,
    invertedResidualBlock12, invertedResidualBlock13,
    invertedResidualBlock14, invertedResidualBlock15)
```

```
    let outputConvResult = hardSwish(outputConvBatchNorm(output
    Conv(backbone3)))
    let averagePool = avgPool(outputConvResult).reshaped(to: [
      input.shape[0], 1, 1, self.lastConvChannel,
    ])
    let finalConvResult = dropoutLayer(hardSwish(finalConv(
    averagePool)))
    return flatten(classiferConv(finalConvResult))
  }
}

public struct MobileNetV3Small: Layer {
  @noDerivative public let zeroPad =
  ZeroPadding2D<Float>(padding: ((0, 1), (0, 1)))
  public var inputConv: Conv2D<Float>
  public var inputConvBatchNorm: BatchNorm<Float>

  public var invertedResidualBlock1: InitialInvertedResidualBlock
  public var invertedResidualBlock2: InvertedResidualBlock
  public var invertedResidualBlock3: InvertedResidualBlock
  public var invertedResidualBlock4: InvertedResidualBlock
  public var invertedResidualBlock5: InvertedResidualBlock
  public var invertedResidualBlock6: InvertedResidualBlock
  public var invertedResidualBlock7: InvertedResidualBlock
  public var invertedResidualBlock8: InvertedResidualBlock
  public var invertedResidualBlock9: InvertedResidualBlock
  public var invertedResidualBlock10: InvertedResidualBlock
  public var invertedResidualBlock11: InvertedResidualBlock

  public var outputConv: Conv2D<Float>
  public var outputConvBatchNorm: BatchNorm<Float>
```

```
public var avgPool = GlobalAvgPool2D<Float>()
public var finalConv: Conv2D<Float>
public var dropoutLayer: Dropout<Float>
public var classiferConv: Conv2D<Float>
public var flatten = Flatten<Float>()

@noDerivative public var lastConvChannel: Int

public init(classCount: Int = 1000, dropout: Double = 0.2) {
  inputConv = Conv2D<Float>(
    filterShape: (3, 3, 3, 16),
    strides: (2, 2),
    padding: .same)
  inputConvBatchNorm = BatchNorm(
    featureCount: 16)

  invertedResidualBlock1 = InitialInvertedResidualBlock(
    filters: (16, 16),
    strides: (2, 2), seLayer: true)
  invertedResidualBlock2 = InvertedResidualBlock(
    filters: (16, 24),
    expansionFactor: 72.0 / 16.0, strides: (2, 2))
  invertedResidualBlock3 = InvertedResidualBlock(
    filters: (24, 24),
    expansionFactor: 88.0 / 24.0)
  invertedResidualBlock4 = InvertedResidualBlock(
    filters: (24, 40),
    expansionFactor: 4, strides: (2, 2), kernel: (5, 5),
    seLayer: true,
    activation: .hardSwish)
  invertedResidualBlock5 = InvertedResidualBlock(
    filters: (40, 40),
```

```
      expansionFactor: 6, kernel: (5, 5), seLayer: true,
      activation: .hardSwish)
    invertedResidualBlock6 = InvertedResidualBlock(
      filters: (40, 40),
      expansionFactor: 6, kernel: (5, 5), seLayer: true,
      activation: .hardSwish)
    invertedResidualBlock7 = InvertedResidualBlock(
      filters: (40, 48),
      expansionFactor: 3, kernel: (5, 5), seLayer: true,
      activation: .hardSwish)
    invertedResidualBlock8 = InvertedResidualBlock(
      filters: (48, 48),
      expansionFactor: 3, kernel: (5, 5), seLayer: true,
      activation: .hardSwish)
    invertedResidualBlock9 = InvertedResidualBlock(
      filters: (48, 96),
      expansionFactor: 6, strides: (2, 2), kernel: (5, 5),
      seLayer: true,
      activation: .hardSwish)
    invertedResidualBlock10 = InvertedResidualBlock(
      filters: (96, 96),
      expansionFactor: 6, kernel: (5, 5), seLayer: true,
      activation: .hardSwish)
    invertedResidualBlock11 = InvertedResidualBlock(
      filters: (96, 96),
      expansionFactor: 6, kernel: (5, 5), seLayer: true,
      activation: .hardSwish)

    lastConvChannel = 576
    outputConv = Conv2D<Float>(
      filterShape: (
        1, 1, 96, lastConvChannel
      ),
```

```
    strides: (1, 1),
    padding: .same)
  outputConvBatchNorm = BatchNorm(featureCount: lastConvChannel)

  let lastPointChannel = 1280
  finalConv = Conv2D<Float>(
    filterShape: (1, 1, lastConvChannel, lastPointChannel),
    strides: (1, 1),
    padding: .same)
  dropoutLayer = Dropout<Float>(probability: dropout)
  classiferConv = Conv2D<Float>(
    filterShape: (1, 1, lastPointChannel, classCount),
    strides: (1, 1),
    padding: .same)
}

@differentiable
public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
  let initialConv = hardSwish(
    input.sequenced(through: zeroPad, inputConv,
    inputConvBatchNorm))
  let backbone1 = initialConv.sequenced(
    through: invertedResidualBlock1,
    invertedResidualBlock2, invertedResidualBlock3,
    invertedResidualBlock4, invertedResidualBlock5)
  let backbone2 = backbone1.sequenced(
    through: invertedResidualBlock6, invertedResidualBlock7,
    invertedResidualBlock8, invertedResidualBlock9,
    invertedResidualBlock10, invertedResidualBlock11)
  let outputConvResult = hardSwish(outputConvBatchNorm(output
  Conv(backbone2)))
```

```
    let averagePool = avgPool(outputConvResult).reshaped(to: [
      input.shape[0], 1, 1, lastConvChannel,
    ])
    let finalConvResult = dropoutLayer(hardSwish(finalConv(
    averagePool)))
    return flatten(classiferConv(finalConvResult))
  }
}
```

Results

This network will train to be slightly less accurate than EfficientNet, but can be evaluated on a mobile device quickly. In addition, the resulting network is small and so can be sent easily over the network to edge devices.

```
Starting training...
[Epoch 1] Accuracy: 50/500 (0.1) Loss: 3.3504734
[Epoch 2] Accuracy: 253/500 (0.506) Loss: 1.4156498
[Epoch 3] Accuracy: 335/500 (0.67) Loss: 1.0543922
[Epoch 4] Accuracy: 326/500 (0.652) Loss: 1.1357045
[Epoch 5] Accuracy: 353/500 (0.706) Loss: 0.9812555
[Epoch 6] Accuracy: 350/500 (0.7) Loss: 0.9210515
[Epoch 7] Accuracy: 380/500 (0.76) Loss: 0.7407557
[Epoch 8] Accuracy: 347/500 (0.694) Loss: 1.038017
[Epoch 9] Accuracy: 343/500 (0.686) Loss: 1.0409927
[Epoch 10] Accuracy: 377/500 (0.754) Loss: 0.8882818
[Epoch 11] Accuracy: 381/500 (0.762) Loss: 0.9374979
[Epoch 12] Accuracy: 383/500 (0.766) Loss: 0.8867029
[Epoch 13] Accuracy: 365/500 (0.73) Loss: 1.3112245
```

```
[Epoch 14] Accuracy: 377/500 (0.754) Loss: 0.9881239
[Epoch 15] Accuracy: 386/500 (0.772) Loss: 0.99048007
[Epoch 16] Accuracy: 406/500 (0.812) Loss: 0.78758305
[Epoch 17] Accuracy: 402/500 (0.804) Loss: 0.8263649
[Epoch 18] Accuracy: 407/500 (0.814) Loss: 0.8147187
[Epoch 19] Accuracy: 401/500 (0.802) Loss: 0.8540674
[Epoch 20] Accuracy: 387/500 (0.774) Loss: 0.90144944
[Epoch 21] Accuracy: 404/500 (0.808) Loss: 1.0089223
[Epoch 22] Accuracy: 396/500 (0.792) Loss: 0.97762024
[Epoch 23] Accuracy: 399/500 (0.798) Loss: 0.9001269
[Epoch 24] Accuracy: 389/500 (0.778) Loss: 1.1596041
[Epoch 25] Accuracy: 384/500 (0.768) Loss: 1.235701
[Epoch 26] Accuracy: 396/500 (0.792) Loss: 1.0384445
[Epoch 27] Accuracy: 405/500 (0.81) Loss: 0.9806802
[Epoch 28] Accuracy: 405/500 (0.81) Loss: 0.9442753
[Epoch 29] Accuracy: 411/500 (0.822) Loss: 0.85053337
[Epoch 30] Accuracy: 422/500 (0.844) Loss: 0.8129424
```

EfficientNet-EdgeTPU

In the same way, we can use the EfficientNet search strategy to build networks for mobile devices; we can use it to build networks for even smaller devices. Google has produced a line (Coral is the brand name) of small ASIC devices, called EdgeTPU, which plug into your computer and allow us to run tensorflow lite models on our own hardware. Conceptually, these devices have extremely limited memory space and compute power, but they are AI hardware just the same as our video card. By giving the device constraints to the EfficientNet search algorithm, they were able to discover an optimal set of networks to run on a device with extremely limited compute capacity.

Recap

In the past few chapters, we've gone from small networks to large ones, and now we've come back to small ones again. These areas of research are all getting very close together and interrelated. Let's look now at how to apply this to your own work.

CHAPTER 12

Bag of Tricks

In this chapter, we will look at how we can modify our original ResNet 50 network to achieve nearly as accurate of results as EfficientNet by combining many different approaches.

So, you've made it this far. We've gone from the very basics of using neural networks to perform image recognition to the current state of the art in this field. Allow me now to offer some qualifications on my approach. First, I've sort of clear-cut a very direct path through this field with the goal of making the early stages as simple as possible for somebody new. In the process, I've skipped over a lot of history, important milestones, and large swaths of research. There are many different papers and approaches that I've not mentioned that contain interesting ideas that you should look at. The short version is that progress is never as linear as I have attempted to present it here. There are usually lots of random approaches, false starts that lead to dead-end alleys, and many different things tried, of which only a small fraction actually work. Progress is usually ugly and tedious.

Bag of tricks

Let's look at an example of what are sometimes called bag of tricks style approaches. As a general rule, somebody will come up with a novel idea that they publish as a paper. We've seen a dozen such examples now. Then, various other researchers and groups will attempt to combine it together with as many other different approaches as possible to try and find a

B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2_12

magical combination that produces a novel result. At a high level, this is perhaps the academic version of NASNet. What often happens is that it is discovered that there are other ways of getting the same results and that the original researchers ended up in a local maxima, so to speak.

> Compounding the Performance Improvements of Assembled Techniques in a Convolutional Neural Network

```
> https://arxiv.org/abs/2001.06268
```

Let's look at a recent paper, "Compounding the Performance Improvements of Assembled Techniques in a Convolutional Neural Network," as an example of this. Lee et al. took our same ResNet 50 approach from a few chapters back and found how to modify it to produce nearly as good of results as EfficientNet, at a much cheaper cost.

They add the following tweaks to the basic network we looked at before:

- Replaced our 7x7 head of the ResNet 50 with a 3x3 stride 2 + 3x3 + 3x3 convolution approach
- Removed the 2x2 stride from our initial 1x1 convolution in the ResNet 50 block and added it to the 3x3 convolution
- Added an Averagepool2d step as part of the skip connection convolutional layer
- Added a Channel Attention (CA) operator
- Selective Kernel (SK) block
- Big-little net block skip connections

To do so, they used bits of the following image recognition papers:

> Bag of Tricks for Image Classification with Convolutional Neural Networks

```
> https://arxiv.org/abs/1812.01187
```

> Selective Kernel Networks

> https://arxiv.org/abs/1903.06586

> Big-Little Net: An Efficient Multi-Scale Feature Representation for Visual and Speech Recognition

> https://arxiv.org/abs/1807.03848

> Making Convolutional Networks Shift-Invariant Again

> https://arxiv.org/abs/1904.11486

In addition, they use the following data augmentation/training/normalization techniques:

> Regularizing Neural Networks by Penalizing Confident Output Distributions

> https://arxiv.org/abs/1701.06548

> AutoAugment: Learning Augmentation Policies from Data

> https://arxiv.org/abs/1805.09501

> mixup: Beyond Empirical Risk Minimization

> https://arxiv.org/abs/1710.09412

> Distilling the Knowledge in a Neural Network

> https://arxiv.org/abs/1503.02531

> DropBlock: A regularization method for convolutional networks

> https://arxiv.org/abs/1810.12890

What to learn from this

To me, this is why I don't get worked up about research groups throwing more and more computational power at problems. Even if their approach can be summed up as brute force, in proving that the larger-scale approach works, they leave the door open for individual researchers to be able to replicate their results with simpler hardware.

My experience is that things usually go like this:

- A lot of researchers trying to find small novel approaches since they don't have large-scale machinery.
- Somebody finds something that produces consistent improvements (e.g., people can replicate their results).
- Large research groups rush in to throw large compute resources at the problem. A few months later, they publish the results of trying to scale things.

Scaling usually looks like this:

- Original researcher: Sigma 0.5 improvement.
- 10x cluster: Sigma 0.85 improvement.
- 100x cluster: Sigma 0.95 improvement.
- All the computational power in the world: Sigma 0.985 improvement.
- Six months later: Somebody else figures out how to replicate the large cluster's work with a limited amount of compute resources, and the cycle repeats.

- Meanwhile, many small unknown researchers are publishing novel findings that are being roundly ignored.
- Someone publishes a blog post that goes viral, and we go back to the beginning.

Reading papers

The crucial skill you need to succeed at this field then is not the most cutting-edge network theory or the fastest computer, both of which will most likely be obsolete in a year. A timeless skill instead is the ability to read papers on your own and keep up with progress. When you encounter things in papers you don't understand, you need the ability to look up that paper's references and figure out where they got their ideas from. If you go back far enough, the references have a tendency to converge on a few key concepts. Learn those and you will have a solid foundation for whatever you want to do.

Stay behind the curve

A surprising number of papers come out, make a big splash, and then disappear. I find trying to keep up with the latest developments to have a high probability of getting sidetracked. My advice instead is to stay behind the curve by a few months. Let other people read the latest and greatest work and then wait for them to actually prove that things work. Look for code demos on GitHub showing how the new thing works or Jupyter Notebooks that explain what is going on. This to me is why you should pick up other frameworks (e.g., pytorch) as well because then you can much more easily absorb knowledge from the broader community of machine learning researchers continuously testing new ideas.

Find a few researchers to follow on Twitter and then see what they are reading and talking about. Let them do the filtering for you. Perhaps from a game-theoretic standpoint, progress would grind to a halt if everybody did this, but unless you are a leading researcher in these fields, then the chance of getting sucked down a blind alley is high.

To use a different example, we might consider the work required to learn and understand and run a model on these various test datasets like so:

- MNIST: 1 minute
- CIFAR: 1 hour
- Imagenette: 1 day
- ImageNet: 1 month

I'm just making these numbers up, don't read too much into them. To me, then, you should spend an order of magnitude more on the smaller stuff than the bigger networks. Before you jump to CIFAR, you should do MNIST a dozen different times; before you jump to Imagenette, you should do CIFAR a dozen times; and so on. For the compute it takes to do a single ImageNet run you could do MNIST a thousand different ways on a basic computer, but it is exceedingly rare to find people who have done so, even though the resources required should be accessible to anyone.

To me, it is difficult to compete with the high-end research teams with large clusters of the latest and greatest hardware able to run massive experiments at scale. But where we can surpass them is quite simple, by going deeper on a particular problem than anybody else can. The success of the large research groups is also their weakness, in that they are constantly searching for new ways to produce publishable results. If that's not important to you, then you can spend a lot more time in the weeds than they can, so to speak. By extension, you can uncover the things they miss in their haste to get results out the door.

How I read papers

Usually, I read the abstract and hopefully get a high-level understanding of what the paper is about. I am happy to confess that often I read the abstract and first few paragraphs and feel like I have no clue what the heck is going on. Sometimes with papers they cover so much ground that they cannot be reduced to a few sentences (or maybe they're not being terribly clear), so I would argue this is as much on the authors as me. I usually just skip right to looking at graphs and tables, which hopefully have some sort of easy visual of what the heck the paper is attempting to do. If that fails, then I will read the conclusion. And if all of this fails, then I will sit down and attempt to skim through the paper and try to get a high-level understanding that way. My basic process is to try and get a high-level understanding and then do successive rereadings until I actually follow what is going on.

I like to print out papers and look at them that way if I feel it is an important one. Making notes in the margins is an approach I do as well. Being able to carry thousands around on your laptop in digital form is nice if you are constantly on the go, but I have slowly amassed a collection of work that I think is important to keep at hand.

Finally, take your time! Depth is far more valuable than breadth. I have found that finding a few papers that are genuinely interesting and taking the time to understand them thoroughly is a far better plan than trying to dabble in a bunch of random fields.

Recap

We've broken down a paper in this field trying to build an EfficientNet-level performance by combining half a dozen other techniques from academia. We've talked about how to get started reading papers on your own.

CHAPTER 13

MNIST Revisited

The twentieth century had a number of interesting inventions, but I believe computers are the most important one. Every year has seen more and more compute cycles being brought to market, and every year has seen appetite and demand for computing increase. We may have hit the limits of Dennard scaling, but there are many decades of interesting improvements to be made.

Next steps

Here's how I see the near future coming at us:

- More cores
- More RAM
- More bandwidth
- More customized hardware
- More generic hardware

Cores in general are simple. We've hit the limits of how fast we can make silicon go, but we can continue to build extra transistors into devices. The easiest trick then is to simply increase the number of individual processors on a chip. AMD's recent Ryzen chiplet approach for processors shows that this can go on for a long time.

B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2_13

RAM: You can provision cloud servers with terabytes of memory today if you so desire. There is a long way yet to go on this front. The real blocker on this front is not memory size, but rather our next trend.

Bandwidth: PCI 4 has made it to market and people are already working on PCI 5 and PCI 6. The real limitation of most modern systems is no longer cores but rather coordinating and synchronizing between them. We've hit the limits of raw clock speed, and so now the crucial trick is keeping the cores fed with instructions and data. If each core on a Threadripper is literally processing a bit of data a cycle, then suddenly we are processing faster than our memory can keep up.

Custom hardware: Apple's ARM processors, Nvidia's GPUs, and Google's TPUv1 and recently new companies such as Cerberas using TSMC's fabs are driving a lot of things in the industry right now. They are forcing massive economies of scale onto the market and making it possible for people to rent fab space cheaply which is in turn allowing it to be possible to build custom silicon much more cheaply than was ever possible before. You can literally prototype a chip in software, ship the designs off, and get the result back in the mail a short while later. This is allowing an entirely new generation of hardware to be able to make it to come to market, and I think we are now just only seeing the beginnings of what is possible.

Generic hardware: This to me is the super interesting flip side of being able to make your own chips. A lot of progress in the field of computing in general has been held up over the years because of patent issues and needs to cross-license intellectual property. There are open source chip designs (RISC-V is a good example) that you can use to build a modern 64-bit processor at no cost. Tools like LLVM mean that if you can build an export module for your architecture, then all of a sudden you can bring entire software ecosystems to your new device.

Pain points

Hopefully, none of the given ideas is controversial to you. Now, I believe that if we look at these ideas, we can see some clear trends shaping up in general.

Multicore programming is not a new concept, but actually using it is. It has been readily available on desktop computers for over two decades now. Having said that, very little software actually really uses all the power available on the CPU, and most programmers are still stuck in single-threaded programming models. Most modern parallelization is by running lots of jobs at once (e.g., hosting a dozen virtual machines on one server or running 10,000 jobs in a queue), not by actually breaking individual jobs up properly.

RAM is a significant limiting factor of deep learning in particular, but this is actually because of the next problem, bandwidth. The real power of GPUs for deep learning is not the GPU itself but rather the internal memory/communication bus. Higher and higher speed RAM is one of the most expensive components of the ecosystem currently, but every iteration allows even more data to be run through the GPU processor, and so this piece continues to evolve. I think eventually this tech will make its way back into the CPU's proper and enable them to make more of a contribution.

Bandwidth: GPU --> RAM memory is reasonably well solved with the above, but whenever we want to try and coordinate the work of more than one GPU, we're right back to the starting point of hitting the PCI bus bandwidth limits. Nvidia is well aware of this weakness and has gone to great lengths to implement a custom intra-GPU networking stack (NCCL) with their DGX series of computers. Habana Labs' Gaudi simply replaces all this custom silicon and complexity by essentially gluing a 100 gigabit Ethernet switch onto each ASIC in order to guarantee 1 terabit of communication bandwidth between each node. Nvidia's recent acquisition of Mellanox, makers of switch hardware, to me points toward

this future as well. The EGX A100 puts 200Gbps Infiniband onto each GPU so that the PCI bus is no longer a limiting factor and multiple cards can have their own dedicated backplane to talk together. Then various network topologies can be implemented at will without having to rely on custom communication protocols, which means that this approach will easily scale with 200 and 400GbE coming online. Doubling this again in the future with 800GbE and 1.6TbE should be doable as well.

Custom ops: Beyond basic MAC operations, which is what most of the current generation of AI hardware is targeting, there's still uncertainty about what particular set of math operations are most useful in practice. On one side, you have say the technical approaches in the form of INT1, INT4, INT8, and FP16 math as natural extensions of making existing operations smaller and increasing the number of data that can be processed in a single pass. On the other side, you have the pragmatic approach of BFloat16 in Google's TPU and Intel's upcoming accelerators, which simplifies porting FP32 workflows to new devices by reducing the complexity of dealing with buffer overruns. Nvidia's Ampere road map shows them supporting basically every operation possible by adding larger versions of the BFloat approach (e.g., supporting INT1, INT4, INT8, FP16, BFloat16, TFloat32, FP32, TFloat64, FP64) and putting the onus of actually implementing things on the coder. What is exciting about this platform is that by standardizing the operations available to the end user, there will no longer be any good excuses for not using custom precision hardware.

Generic hardware: To me, the most interesting quiet revolution going on is ARM chipsets in general and Amazon's recent embrace of this platform for their next generation of server hardware. By removing proprietary silicon from the loop, even greater efficiencies of scale can be achieved. This will take several years to fully play out, but this is where we will be in the near future. ARM and RISC-V will follow the bleeding-edge platforms and quietly absorb whatever new innovations they bring to market. Meanwhile, proprietary silicon approaches will have to fight radically cheaper commoditized innovation.

TPU case study

All of this tech is cool, but fundamentally in order to write optimized software for it, programmers must be planning their data and memory access ahead. Like I said before, we have hit the limits of single data–style programming and increasingly must learn how to embrace dataflow-specific methodologies. Let us look then at Google's TPU as an example of tackling the mentioned problem in practice:

1) Cores: The TPU uses fairly straightforward ASIC logic and puts multiple cores together into a single processing package. Then they connect many of these processors together into using a ring topology to produce a single TPU unit.

2) For RAM, Google simply throws a few hundred gigabytes of RAM onto each unit to simplify local memory access.

3) Bandwidth: This is actually one of the secret abilities of the TPU system. Each TPU is mounted on a custom network backplane that allows intra-pod communication at extremely fast speeds. Groups of TPUs are put together into pods where they share the same network backplanes to optimize communication.

4) Custom ops: BFloat16 simplifies porting logic to the TPU, but long term they are looking at adding more custom types. TPUv1 was actually INT8, as a historical aside.

5) This is also off the radar, but each TPU unit has an internal processor that handles much of the more complicated logic internally so that the TPU chips can focus on raw math. Finding ways to do preprocessing on the fly so that the chips themselves can be keep fed is an area of active research.

Tensorflow 1 + Pytorch

To me, many of the design decisions and limitations of the first generation of tensorflow make sense looking from the perspective of writing software for TPUs. For a custom ASIC device like the TPU, you have to have a predefined graph and cannot be executing arbitrary code on the fly. If you have access to literally thousands of TPU cores on demand, then the crucial trick is breaking your code up into units that can be run on each core, not simplification of the overall logic. I would suggest that CUDA support was sort of an afterthought, but the success of the framework was because that this was the one that people in the real world were most likely to have actual hardware they could use. Google has spent a lot of cycles optimizing TPU code only to discover that similar optimizations do not work on CUDA devices, and vice versa. They have tried to bridge the gap but have increasingly hit the limits of trying to make the various worlds work together. For their internal work, they can easily afford to pay people to write custom C++ kernels to optimize software for running on large clusters, but for people outside the Googleplex, this is decidedly impractical.

Pytorch has rapidly become popular in the past few years as an alternative to Tensorflow. A large part of this is because it allows people to work with in-memory (e.g., nonstatic) graphs, which makes debugging much simpler (e.g., we can attach a debugger and look at network variables in place, rather than having to add log statements and run things repeatedly). Tensorflow 2 embraces this paradigm fully with eager execution being the preferred method going forward. Likewise, the Keras

Python wrapper for Tensorflow has been promoted to a full-fledged part of Tensorflow ecosystem (e.g., it's part of the standard library now).

With respect to optimization, Pytorch just takes the much simpler route of going from high-level code to CUDA as quickly as possible. This is significantly more easy to optimize, and the Pytorch team has a much simpler job of optimization as a result. However, they are now tied heavily to CUDA and by extension are heavily tied to whatever hardware Nvidia can bring to market. They have been experimenting with adding compiler techniques between the Pytorch and CUDA layer, but while this is where the problem is, I do not believe it is the right place to solve it.

Enter functional programming

To me, then, forcing the programmer to use functional paradigms is where everybody is going to end up. In order for compilers to make good decisions about how to optimize code, they have to have access to as much information as possible about what is being done. Trying to generate one blob of intermediate code and then analyze that in order to optimize it can produce short-term speedups but in the long term is an exercise in futility. Decades of compiler theory have taught us that no matter how smart the meta-compiler is, so to speak, it cannot compete with the programmer for knowing what really needs to be done.

Or rather, to use a contrived example, there are thousands of ways for a compiler to try and optimize this loop:

```
var i = 0
for n in 1...100000
{
    i = i + n
}
    print (i)
```

A human, though, can see that we can simplify it to

```f(n) = n * (n + 1) / 2```

using math. To me, the reason we use functional programming is not that it is easier in and of itself, but rather that by forcing the programmer to code in a stricter style, we make it dramatically easier for the compiler to make decisions for us about how to actually execute things. We are sacrificing a bit of time now to make our life simpler down the road. I have coded lots of C in my day, as an example, but have spent as much time debugging memory issues as I have trying to add new features. Swift incurs a runtime penalty on this front, but on the flip side, I have twice as much time to implement new features instead. The next level of functional programming comes when you learn to trust the compiler to catch/prevent certain categories of errors, and so you can focus instead on the core logic of your problem rather than minutiae.

No matter how you code the core deep learning logic itself, everybody is going to have to figure out how to actually schedule their jobs. In order to do so, the best approach then is to force the end user to work with data primitives that match the actual data they are manipulating and take account for the hardware it will run on. Then, the compiler can figure out the best way to convert the given data into actual operations. Even if you implement things by hand, this is where writing custom code fails, in that every time our end hardware changes, we have to write new kernels.

Swift + TPU demo

Time will tell if Swift for Tensorflow is the way forward for the broader machine learning ecosystem. For Google proper, I am convinced it is increasingly how they are going to be doing things in the future.
Let us return all the way back to our very first machine learning demo, a convolutional neural network applied to the MNIST dataset, and do
```

it again using a TPU. In order to convert this demo to run on a TPU, historically, we would have needed to work with C++ either directly or through a high-level API (e.g., Keras) that is hiding the rough edges from us.

For this to work, you will need to set up a remote server using the instructions from the chapter on Google Cloud. You do not need a GPU or CUDA since you will be using a TPU. Afterward, you will need to create a TPU instance in the same zone as your server so that they can talk together. Start by figuring out where you're going to create the TPU (v3-8 is all you need) and then work backward to the zone for your host server. After you have your system up and running, set the following shell parameters for your cloud system:

```
export XLA_USE_XRT=1
export  XRT_TPU_CONFIG="tpu_worker;0;<TPU_DEVICE_IP>:8470"
export  XRT_WORKERS='localservice:0;grpc://
localhost:40934'
export  XRT_DEVICE_MAP="TPU:0;/job:localservice/replica:0/
task:0/device:TPU:0"
```

And now we can run our simple MNIST CNN demo that uses XLA to run our swift code on the TPU:

```
```

```
import Datasets
import TensorFlow

struct CNN: Layer {
  var conv1a = Conv2D<Float>(filterShape: (3, 3, 1, 32),
padding: .same, activation: relu)
  var conv1b = Conv2D<Float>(filterShape: (3, 3, 32, 32),
padding: .same, activation: relu)
  var pool1 = MaxPool2D<Float>(poolSize: (2, 2), strides: (2, 2))
```

```
  var flatten = Flatten<Float>()
  var inputLayer = Dense<Float>(inputSize: 14 * 14 * 32,
  outputSize: 512, activation: relu)
  var hiddenLayer = Dense<Float>(inputSize: 512, outputSize:
  512, activation: relu)
  var outputLayer = Dense<Float>(inputSize: 512, outputSize: 10)

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let convolutionLayer = input.sequenced(through: conv1a,
    conv1b, pool1)
    return convolutionLayer.sequenced(through: flatten,
    inputLayer, hiddenLayer, outputLayer)
  }
}

let batchSize = 128
let epochCount = 12
var model = CNN()
var optimizer = SGD(for: model, learningRate: 0.1)
let dataset = MNIST(batchSize: batchSize)

let device = Device.defaultXLA
model.move(to: device)
optimizer = SGD(copying: optimizer, to: device)

print("Starting training...")

for (epoch, epochBatches) in dataset.training.
prefix(epochCount).enumerated() {
  Context.local.learningPhase = .training
  for batch in epochBatches {
    let (images, labels) = (batch.data, batch.label)
    let deviceImages = Tensor(copying: images, to: device)
```

```
  let deviceLabels = Tensor(copying: labels, to: device)
  let (_, gradients) = valueWithGradient(at: model) { model
  -> Tensor<Float> in
    let logits = model(deviceImages)
    return softmaxCrossEntropy(logits: logits, labels:
    deviceLabels)
  }
  optimizer.update(&model, along: gradients)
  LazyTensorBarrier()
}

Context.local.learningPhase = .inference
var testLossSum: Float = 0
var testBatchCount = 0
var correctGuessCount = 0
var totalGuessCount = 0

for batch in dataset.validation {
  let (images, labels) = (batch.data, batch.label)
  let deviceImages = Tensor(copying: images, to: device)
  let deviceLabels = Tensor(copying: labels, to: device)
  let logits = model(deviceImages)
  testLossSum += softmaxCrossEntropy(logits: logits, labels:
  deviceLabels).scalarized()
  testBatchCount += 1

  let correctPredictions = logits.argmax(squeezingAxis: 1) .==
  deviceLabels
  correctGuessCount += Int(Tensor<Int32>(correctPredictions).
  sum().scalarized())
  totalGuessCount = totalGuessCount + batch.data.shape[0]
  LazyTensorBarrier()
}
```

```
  let accuracy = Float(correctGuessCount) / Float(totalGuessCount)
  print(
    """
    [Epoch \(epoch + 1)] \
   Accuracy: \(correctGuessCount)/\(totalGuessCount) (\(accuracy)) \
    Loss: \(testLossSum / Float(testBatchCount))
    """
  )
}
```

Results

You should see similar results to our second chapter:

```
Starting training...
[Epoch 1] Accuracy: 9645/10000 (0.9645) Loss: 0.11085216
[Epoch 2] Accuracy: 9745/10000 (0.9745) Loss: 0.078900985
[Epoch 3] Accuracy: 9795/10000 (0.9795) Loss: 0.057063542
[Epoch 4] Accuracy: 9826/10000 (0.9826) Loss: 0.05429901
[Epoch 5] Accuracy: 9857/10000 (0.9857) Loss: 0.042912092
[Epoch 6] Accuracy: 9861/10000 (0.9861) Loss: 0.043906994
[Epoch 7] Accuracy: 9871/10000 (0.9871) Loss: 0.041553106
[Epoch 8] Accuracy: 9840/10000 (0.984) Loss: 0.050182436
[Epoch 9] Accuracy: 9867/10000 (0.9867) Loss: 0.044656143
[Epoch 10] Accuracy: 9872/10000 (0.9872) Loss: 0.040160652
[Epoch 11] Accuracy: 9876/10000 (0.9876) Loss: 0.041967977
[Epoch 12] Accuracy: 9878/10000 (0.9878) Loss: 0.041590735
```

Recap

Using the power of your knowledge of Swift for Tensorflow, you've run a custom kernel on the TPU (or CPU or GPU as desired). Time will tell what other back ends will be supported, but to me this is the real power of embracing this approach, the ability to write code once and run it anywhere.

CHAPTER 14

You Are Here

Congratulations on making it this far! You now have a solid working knowledge of the current state of the art of convolutional neural networks for image recognition, using swift for tensorflow. Let's look toward the future by first looking at the past.

A (short and opinionated) history of computing

It is valuable to study the history of to understand its future. There are many trends that are obvious only in hindsight. So, let us go all the way back to the beginning. The birth of Silicon Valley was arguably an overflow of military computing funding in the aftermath of World War II. The military wanted to fund various things, but they could not build them themselves, and so they started buying hardware from various labs that were set up in the valley to construct transistors. This was the real genesis of Silicon Valley, the ability to build strange new things with the knowledge that there was a willing buyer for what were extremely beta technologies.

The Internet itself, then, was an outgrowth of the ARPANET project, an initiative by DARPA to network various previously unconnected servers. If we can connect computers together locally using a network, then extending the network a few miles down the road is a fairly logical next step. But to quote Metcalfe's law, as each new node was added, the value of the network grew exponentially. What is interesting then is that,

B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2_14

at a certain point, the value of adding new nodes to the network exceeded the cost. At which point, the process of adding new computers to the network became self-sustaining and then grew to what we see today. Or rather, I would argue that at a certain point, the commercial value of the invention itself exceeded the cost to bootstrap it, and after that point, it was impossible to halt the growth of what became the Internet. The genie was out of the bottle, so to speak.

In the 1970s, a different phenomenon occurred with supercomputing and AI in particular. The military funded many different strategies in the field, which started making more and more outlandish claims in order to get a bigger piece of the pie. Once it became clear many of these approaches weren't going to work came the AI winter, when DARPA pulled funding for many of these projects and the field was forced to try and fend for itself. Without a wealthy benefactor, or more precisely a clear commercial plan, both supercomputing and AI fell on hard times. The UK and Japan experienced similar phenomena a decade later.

And so the supercomputer race failed for the most part. But computers had proven their value in general and so continued to become cheaper and cheaper in general. Personal computing took off and a similar scenario happened, whereas the value of a computer to individual users exceeded the threshold of cost, and so as a result, the personal computer revolution became self-sustaining. As a result of this massive interest into home computers came the PC revolution of the 1980s and 1990s. What is interesting to me in particular is the third-generation supercomputing wave of the late 1990s, which was largely the result of taking off the shelf commodity processors (which had progressed far faster than the specialized supercomputing manufacturers could ever dream of) and wiring them together using advanced networks in order to tackle problems in a distributed fashion. Commoditized general hardware beat building specialized processors and methodology. Most current/fourth-generation

supercomputing follows this trend, using commodity computing hardware and focusing on custom networking to increase intraprocess communication.

History of GPUs

And so, to look at another wave, we can consider the story of video cards. Originally, monochrome color and basic text were all that computers could generate. Memory capacity then increased to where larger amounts of data could be stored, leading to color becoming possible and gradually increasing resolutions. At some point, rastering 3D graphics on the fly became possible, and the 3dfx brought the first real GPU to market. Using a graphics programming language, all of a sudden a whole new world of interactive experiences (aka games) became possible. And so, to mirror the Internet and personal computing waves of before, the commercial value of playing games created a self-sustaining revolution in chipsets, which is still going on today. The entire reason we are running models on graphics cards today is due to the popularity of video gaming decades ago.

GPUs are getting close to becoming consumed by commoditization as well. While the market for new experiences continues to grow at this point today, even budget cards support features such as 4k video, which would have been unthinkable a few years ago. Running nongame code (notably bitcoin and deep learning) on the GPU itself is an extremely recent innovation that has breathed new life into the market. The companies making these devices are quickly reaching the limits of raw processing to make all of this possible. They are trying to bring new hardware to market without straying too far from the gaming market which drives everything. This is a large part of the push for VR and AR experiences. As GPUs become more general, they are increasingly absorbing more and more of the compute stack previously only controlled by CPUs.

Cloud computing

Virtual machines have significantly changed how people interact with computing, even if they are not aware of it. At one point, setting up and configuring a server took days; now it can be done in seconds. This enables workflows where resources are spun up on demand and then promptly discarded. Software is increasingly run at higher and higher abstractions which has allowed entirely new approaches to become commonplace. This will have long-term ramifications that we cannot even fully comprehend today. The largest computing clusters in the world are not supercomputers but rather managed servers running thousands of virtual machines for the cloud providers.

Crossing the chasm

AI and ML are not new fields. Neural networks, in the form of the perceptron, were invented in 1958. Only recently with the mentioned advances in compute power and hardware have they become practical to implement. Moreover, I would argue that they have finally crossed the chasm from intellectual curiosity into something driving the bottom line at large companies. As such, they have made the necessary transition to become a self-sustaining technology like the given examples. Google could delete the tensorflow repository tomorrow. Nvidia could stop shipping video cards. But these techniques will continue to be refined and improved regardless because they have real-world practical use cases in the industry. As such, the genie is out of the bottle. There is no going back to the pre-AI world. One way or another, the gains that AI brings will be brought to every field.

Computer vision

Let us look at the big areas that I believe will be important for the next decade.

Direct applications

Many of the more advanced forms of computer vision are finally seeing the hardware and compute capacity needed to run them become mainstream. I am particularly interested in the field of real-time systems, be it cameras on self-driving cars, being able to analyze medical data in the field, or even simply finding new ways to use the cameras on mobile phones. This area is only just now beginning to be touched.

Indirect applications

A number of interesting problems that are not necessarily image related can be converted into images and then solved using CNN-style approaches. Historically, many of these techniques have been impractical from a resource standpoint, but as more and more AI-specific hardware becomes mainstream, a lot of approaches that were previously infeasible become doable. AlphaGo, as an example, is a large-scale reinforcement algorithm that converts the board game go's game state into an image representation and then applies an extremely large convolutional neural network to it. The basic approach, though, is a convolutional neural network built using residual layers and large-scale compute. When average researchers gain access to similar amounts of resources, I think many interesting new approaches will be found in fields that are just now starting to experiment with AI.

Natural language processing

By using big data approaches (e.g., data corpuses from Wikipedia, scanned books, and gathered from the Internet at large), simpler approaches suddenly become powerful by virtue of giving the machine a lot more

information to work with. This in turn has direct financial ramifications (e.g., improving search and recommendation engines), and so a lot of resources are being poured into this right now. It is going to become commonplace eventually.

Reinforcement learning and GANs

I am somewhat bearish on these fields in the short term, in that they still seem to require massive amounts of resources and there are still not a lot of clear commercial applications at this point in time. Having said that, I believe that in the long term, this is the field that is most going to drive progress in AI/ML in general. Most improvements in computer vision are now very small incremental tweaks, and any time an idea shows promise upstream in RL, then very quickly people will be trying to use it elsewhere. Using synthetic data to train neural networks is the area that seems most poised to become a commercial driver in the near future. Supersampling/resolution is making its way into silicon and is clearly here to stay.

Simulations in general

The other interesting area that I think is poised to be revolutionized by neural techniques is physical simulations in general. A very large amount of compute power is thrown regularly at performing complicated simulations of interactions based upon physics. I'm bearish on neural networks replacing physical simulations directly, because there will always be a place for raw math, but using networks to simulate real-world datasets opens up an interesting window of being able to simulate simulations, so to speak, and by extension being able to build approximately correct models much, much more quickly than traditional approaches. If the neural network–based simulation proves itself, then the traditional method can be run as the final phase, giving the best of both worlds

(e.g., fast experimentation and fundamental rigor when desired). There is a danger of the networks losing touch with reality (e.g., simulating the wrong things), but I believe that having domain experts will obviate this problem.

To infinity and beyond

My experience is that this field as a whole has no shortage of ideas right now. There are thousands of papers being published each year on arXiv, and the rate of submissions only continues to grow. Many other fields, in particular mathematics, seem finally convinced that deep learning techniques are here to stay and that they need to get on the bandwagon, and so many extremely smart people are out there doing these hello world exercises, the same as you. In the short term, this is creating a lot of churn. There are countless blog posts by people attempting to explain their new ideas and online debates over the best approaches. Every new major release of pytorch or tensorflow breaks existing projects in all sorts of exciting new ways. People throw up their hands at the complexity and decide they're going to create a new unified system for doing things, and voilà, there's yet another new framework. This is literally going on as we speak. The industry as a whole is lurching from shiny thing to shiny thing. The simple truth is that nobody really knows what the right path forward is. New techniques are being discovered daily, and deep learning approaches have brought together dozens of related fields. Neural network and big data approaches have proven themselves on disparate problems such as biology, astronomy, physics, and economics. Every field now has to learn computer science or they will get left behind by those who do.

And so let me tell you grizzled programmer story of the early days of iOS. With the second generation, Apple let people submit apps. There was a massive gold rush where people could (and did try to) ship almost everything under the sun. The next few years were interesting as more

and more of the approaches finally stabilized and became popular. After a while, libraries and frameworks became standardized. To me, all of this deep learning hullabaloo is very much the same experience of yore.

Why Swift

Swift has been an interesting revolution within the iOS ecosystem. Objective-C was showing its age, and swift brought iOS programmers a long way forward in a hurry. Garbage collection is a traditional approach in this field that works well on systems with large amounts of memory and spare cycles to run garbage collection. But in production systems with hard real-time requirements, be they servers providing 24/7 packet handling guarantees or mobile devices with quasi-random use patterns, this approach doesn't work as well as would be desired. Android has tried to cover up this gap by getting manufacturers to ship more and more RAM with their devices, but this makes devices cost more, which is often not viable in the real world.

LLVM initially snuck into iOS in the form of automatic reference counting, a feature added to Objective-C to count/track memory cycles and by extension be able to manually add malloc and free calls for the developer. Once this tech had proven itself, by eliminating memory management from the day-to-day workflow of programmers, Lattner et al. set their sights significantly higher.

Swift is designed to be a modern language that does not look out of place to existing Objective-C programmers, and I feel like at this point it succeeded extremely well. It brought functional programming ideas and concepts into the world of iOS by making it easy to bridge between the worlds. Have a particular section of code that needs C raw memory access? Just drop down to raw memory access directly, and the compiler can put boundary checking on that entire region of code. Have an existing C library that needs to be brought to swift? Simply write a simple API layer

that encapsulates your library. Then all the system-level communication for iOS (and Mac proper, eventually) was forced to go through a swift layer of indirection. In the short term, this was painful in that it forced coders to no longer be able to do direct system calls. But over time, this approach drastically modularized the codebase at the system level and isolated many different bugs in their own particular islands.

While Apple was eating their own dogfood, iOS developers were going through a similar transition. Many open source libraries sprung up in the early days, each with their own set of trade-offs and patterns. By moving to swift, this forced much of the ecosystem to either evolve or get stuck in the past. In turn, though, this transition allowed people to concentrate on higher-level problems and not get stuck on low-level details.

And so then Apple did the crucial final step of making the language open source and opening it up fully to outside developers to make contributions and shape its future. Anybody can contribute and thousands have now. It is extremely hard for new programming languages to come into being. Small niche languages usually toil in obscurity. Large companies push new languages on the world, but this top-down approach usually only works so long as the original company is driving progress.

So to me then the strengths of swift are manyfold. It is an easy-to-learn language for beginners. It has the support of a large benefactor (Apple) that is committed to its success, but not technically in charge. It has a diverse ecosystem of open source contributors and is solving real problems in the real world daily while utilizing decades of experience building C libraries. It brings functional programming concepts to the procedural world in a pragmatic way without forcing people to completely change how they have been doing things.

Why LLVM

Swift's real magic power, though, is that it is the original language of LLVM. Compilers have historically focused on generating really, really fast code. This is great for progress but also means that many implementations chase speed over doing things correctly, so to speak. What happened as a result is that we ended up with many different compilers generating slightly different code for dozens of slightly different computers, and then build systems became really large and complicated. Generating a new programming language became very difficult because people demanded performance out of the gate.

LLVM rebuilt the foundations of compiler theory and has spawned a renaissance in new languages by reunifying these worlds. At a high level, all you have to do is generate an IR, and then LLVM can figure out how to get it to run on your device. This means that many, many different languages are using LLVM now. As a direct result, by using LLVM, you get the collective improvements of many, many different ecosystems.

This is a little bit more work up front for the programmer in terms of complexity, but as a result fundamentally makes it possible for the compiler to do much, much more. We've seen amazing progress in the LLVM world; people have demonstrated running gigantic jobs on large clusters and other approaches.

Machine learning is still in its infancy in many ways. Single-GPU code is the largest paradigm. People write stuff for clusters, but it is still very much custom code most of the time. We've got a ton of experience with single instruction and single variable code (CPU-style programming), but historically single-instruction multidata code has been really hard to write. We end up with lots of hand-customized kernels for different things. This works fine in a general sense, in that programmers can make system calls and get optimized code, but it means that it is hard for programmers to easily take advantage of whatever hardware they have at hand.

Why MLIR

The end result is that there's been a tremendous amount of churn in the machine learning ecosystem in general the past few years. Each manufacturer ends up trying to build libraries to provide an optimal experience for their hardware. Researchers have tried to make tensorflow do many things it was never designed to do, and so trying to support every permutation has been difficult for Google. Pytorch effectively rebuilt a framework just to make generating CUDA code simpler. MLIR provides a convenient bridge between these worlds. Hardware manufacturers can simply focus on getting an IR together that generates code for their device. Coders can write in arguably whatever language they prefer, and then language wonks simply need to find a way to convert their LLVM AST to an MLIR syntax. Then we can dream of a future in which we take our swift (or any language that supports LLVM) code and can compile it for whatever back end we desire.

Why ML is the most important field

Machine learning has the ability to absorb all of the world's compute capacity for the next few decades. This quiet revolution will have ramifications in dozens of fields and domains. The more and more we make it easier to use these tools and make them able to flexibly scale up to work on larger and larger compute systems, the greater the long-term potential of humanity as a whole. Large-scale compute has the ability to fundamentally do things that have never been possible before.

Clusters are only increasing in size. But all this emphasis on scaling ignores the reality that more compute resources are available to the individual today than at any point in history. If you are willing to invest the time and energy now, then as these things continue to improve, you will be the first to be able to take advantage of this revolution.

Toward this end, you can take two paths. One is to pick a particular horse, be it hardware or framework, and put all your efforts behind it. The other is to focus on helping make it so no particular framework or technology gains control of the ecosystem. Getting all these sundry groups of people working together as a whole has the potential to fundamentally revolutionize this field.

The hardware is just now being figured out, but this is going to change dramatically in the next few years. The software is a bit rough around the edges right now, I will admit. But opportunity never comes wrapped up neatly in a package with a bow. More often than not, it looks like hard work. But a little bit of work today will leave you well positioned for whatever tomorrow brings.

Why now

Progress is the result of many, many people working together over the centuries, not isolated to any one place or time. By helping make machine learning more accessible, you are helping improve tools that will indirectly touch millions of other people's lives. This has the potential to allow progress on a scale never before seen in history.

Why you

You can wait for other people to bring you the future or help them build it. There's never been a better time to get started. The future is now! Come join us!

APPENDIX A

Cloud Setup

We will look at how to get bootstrapped running Swift for Tensorflow on Google Cloud.

Technically speaking, you can skip this Appendix if you wish, but I would highly suggest you spend some time to learn how to use the cloud sooner rather than later. The cloud in general is a deep subject which we could spend an entire book on. I would suggest you not try to do it all at once, but rather get the basics down at first, and then you can slowly add more cloud usage into your workflows over time.

Once you are comfortable working with the command-line workflow we are teaching, then scaling becomes a natural extension of technique. There are various workflows you can make parallel very easily. For example, once you have a particular virtual machine configured exactly how you like, you can run cloud commands to literally create a dozen copies of it to run a job on a dozen different instances at once very quickly.

Having said that, the flip side of spinning up lots of instances is that these approaches can get expensive in a hurry. So, to reiterate my primary point, you should have a workflow where you can play with things locally without worrying about resources while you're figuring out what you need, and then send jobs to the cloud to be scaled as needed.

For the purposes of this book, we are going to be working with Google Cloud. I think this is a good future-proof platform for you to learn if you are new to the field. I am going to try and provide a quickstart for this field

B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2

based on the theory that you can pick up more on your own once you know the basics. If you like a different cloud provider, that is fine; Ubuntu configuration works the same anywhere in the world.

Outline

The three key workflows you need to know, in order of complexity:

1) How to set up, log in, and delete a basic cloud instance (no GPU)
2) How to deploy a preconfigured cloud instance with a GPU for a specific workflow
3) How to configure your own cloud instance with a GPU from scratch to match your local machine's setup, so that you can share code between the two systems easily (see Appendix B)

Most cloud providers will have prebuilt system images with various packages preinstalled and configured for you to run, notably the Nvidia drivers and specific configurations of pytorch/tensorflow. This can be a good way to get a project going quickly, but the converse is that it is easy to try and add a few packages or make a few tweaks on top of somebody else's system only to get stuck dealing with obscure package management issues or (even more fun!) subtle differences between versions of Unix. Having spent a nontrivial portion of my life troubleshooting such things, my basic rule is to use the same operating system in the cloud that we are using on the desktop, namely, Ubuntu 18.04 LTS. We will spend more time at first installing extra packages to catch up to the preconfigured approaches, but on the flip side, we will be able to avoid having to learn the ins and outs of any specific cloud vendor's platform configurations.

Google Cloud with CPU instances

How to sign up for Google Cloud

https://cloud.google.com/

You'll need to sign up for an account. They provide $300 in free credits to get started; this can go a long way if you're careful. You'll need a credit card for the sign-up process, for what it's worth.

I am going to assume you know how to get to the authentication step:

```
sudo snap install google-cloud-sdk --classic
gcloud auth login
```

This will give you a URL in the console. Load this in the same browser you used to load Google Cloud. After a security prompt, you will be given a long alphanumeric key.

Copy-paste this key into the console prompt.

You will also need to set your gcloud project right now:

```
gcloud config set project cnn-s4tf-tutorial
```

Creating your first few instances

The console may say that the compute engine is still getting ready; just wait a few minutes.Walk through creating an f1-micro instance.

- Log in.
- Shut down your instance.
- Delete your instance.
- Make a large RAM instance.
- Run free –h.
- Delete it.
- Make a large CPU instance.

- Run htop.
- Delete it.

I would highly recommend you redo this process several times until you feel extremely comfortable creating and deleting instances on the fly. When using small instances on Google Cloud/other providers, the cost is usually on the order of a few pennies an hour, so mistakes are very cheap at this scale. You can literally spin up and destroy dozens in an hour or two without worrying about cost. Do it a bunch of times until you feel **extremely** comfortable with this step.

Google Cloud with preconfigured GPU instance

This used to be much more complicated. Now Google is providing prebuilt binaries to make our lives much easier. Get a list of images which can run Swift for TensorFlow:

```
gcloud compute images list \
  --project deeplearning-platform-release \
  --no-standard-images | \
  grep swift
```

Run this to test swift with MNIST (CPU-only mode):

```
gcloud compute instances create s4tf-ubuntu \
  --image-project=deeplearning-platform-release \
  --image-family=swift-latest-cpu-ubuntu-1804 \
  --maintenance-policy=TERMINATE \
  --machine-type=n1-standard-2 \
  --boot-disk-size=256GB

gcloud compute ssh s4tf-ubuntu \
  --zone ${ZONE}
```

Run swift:

```
swift
```

Run swift-models:

```
git clone https://github.com/tensorflow/swift-models.git
cd swift-models
swift run

...
error: multiple executable products available: pix2pix, WordSeg,
VariationalAutoencoder1D, VGG-Imagewoof, Shallow-Water-PDE,
ResNet-CIFAR10, Regression-BostonHousing, PersonLab, NeuMF-
MovieLens, MobileNetV2-Imagenette, MobileNetV1-Imagenette,
MiniGoDemo, LeNet-MNIST, Gym-PPO, Gym-FrozenLake, Gym-DQN, Gym-
CartPole, Gym-Blackjack, GrowingNeuralCellularAutomata, GPT2-
WikiText2, GPT2-Inference, GAN, Fractals, FastStyleTransferDemo,
DCGAN, CycleGAN, Custom-CIFAR10, Catch, Benchmarks, BERT-CoLA,
Autoencoder2D, Autoencoder1D

swift run Custom-CIFAR10
```

Now let's try the same, but with a GPU:

```
export GPU_TYPE="t4"
gcloud compute accelerator-types list | grep ${GPU_TYPE}
> not all zones will have gpus free
export ZONE="us-central1-b"

gcloud compute instances create s4tf-ubuntu-${GPU_TYPE} \
  --zone=${ZONE} \
  --image-project=deeplearning-platform-release \
  --image-family=swift-latest-gpu-ubuntu-1804 \
  --maintenance-policy=TERMINATE \
  --accelerator="type=nvidia-tesla-${GPU_TYPE},count=1" \
```

```
  --metadata="install-nvidia-driver=True" \
  --machine-type=n1-highmem-2 \
  --boot-disk-size=256GB
```

If you're using a brand new account, this will fail because you don't have access to GPUs by default. Jump through the hoops to make a quota request and get access to a GPU, then try again.

After it creates, the gcloud command will return, but it will still be a few minutes before the image is ready to respond. SSH in eventually:

```
gcloud compute ssh s4tf-ubuntu-${GPU_TYPE} \
  --zone ${ZONE}
```

```
gcloud compute ssh --zone "us-central1-b" "s4tf-ubuntu-t4"
--project "swift-cnn-gcp-tutorial"
```

Run swift-models:

```
git clone https://github.com/tensorflow/swift-models.git
cd swift-models
swift run
```

```
...
error: multiple executable products available: pix2pix, WordSeg,
VariationalAutoencoder1D, VGG-Imagewoof, Shallow-Water-PDE,
ResNet-CIFAR10, Regression-BostonHousing, PersonLab, NeuMF-
MovieLens, MobileNetV2-Imagenette, MobileNetV1-Imagenette,
MiniGoDemo, LeNet-MNIST, Gym-PPO, Gym-FrozenLake, Gym-DQN, Gym-
CartPole, Gym-Blackjack, GrowingNeuralCellularAutomata, GPT2-
WikiText2, GPT2-Inference, GAN, Fractals, FastStyleTransferDemo,
DCGAN, CycleGAN, Custom-CIFAR10, Catch, Benchmarks, BERT-CoLA,
Autoencoder2D, Autoencoder1D
```

```
swift run Custom-CIFAR10
```

Run this book:

```
git clone REPO_URL
cd cnn-s4tf
swift run
```

Resources:

Google Deep Learning Platform StackOverflow: https://stackoverflow.com/questions/tagged/google-dl-platform

Google Cloud Documentation: https://cloud.google.com/deep-learning-vm

Google Group: https://groups.google.com/forum/#!forum/google-dl-platform

Once you're done plvaying with the demos, shut things down:

```
sudo shutdown -h now
```

You'll be able to see the system shutting down in the Google Console. Delete the instance when you're done with it.

Google Cloud nits

Cattle, not pets

In general, I would advise you to not get too attached to your virtual machines. My broad philosophy is to create them on demand for a specific project and then delete them after. If it took a decent amount of work to get a particular configuration up and running, you can leave a machine off and not running for relatively cheaply (you will get charged for disk space is all). Another good workflow to know is saving a snapshot of a VM and then using it to create more systems.

This is beyond the scope of this book, but I often need to productionize things by creating scripts using Docker/Kubernetes to deploy things on the fly easily. Figuring out what steps are needed to replicate a particular configuration is a good way to get rid of a bunch of complexity in a hurry. I will usually spend a bit of time getting something working, then immediately attempt to replicate my work with a new server while I still remember what is going on. If the new one works properly, then going to Docker is simply a matter of copy-pasting my workflow for the second server over.

These deep learning VMs, preconfigured by Google, are a good trick to know if you would like to experiment with other people's code, which often only work with specific library versions. Basically, you can select a specific version of CUDA and tensorflow/pytorch and usually get things running quickly on a remote server. Then, in an ideal world, you can just download/git clone the demo you want and install its dependencies to go from there. This is an excellent approach to have in your toolbox for testing things. Let Google or somebody else worry about the security implications for you.

Basic Google Cloud nomenclature

Let's review some Google Cloud–specific terminology just to make sure you feel comfortable poking around. There are many, many different areas in Google Cloud that I do not want to explain, but I feel like you will bump into the following with some regularity.

* Machine types

Just use n1 instances, the other types don't automatically attach to GPU instances.

* Buckets

Google lets you put data into different buckets on the server. I would advise against you worrying about access control lists or other sophisticated approaches. They have a few interesting tricks, namely, a transfer appliance button. You can use it to easily copy data between different buckets or from other cloud providers (e.g., S3, Amazon's similar utility).

* Billing

It is always a good idea to keep an eye on your billing in general because cloud costs can add up in a hurry.

Cleaning up

Now that you're done, remember to shut down your instance!

```
sudo shutdown -h now
```

Make sure it's shut down fully in the Google Console. Don't delete this instance; you can just start it up again when needed.

Recap

We've looked at how to get started with a Google Cloud account and basic virtual instances. Then, we've looked at how we can use a prebuilt system configuration to quickly play with random code off the Internet.

APPENDIX B

Hardware Prerequisites, Software Installation Guidelines, and Unix Quickstart

In this Appendix, we will discuss what to buy to build a deep learning server; walk through how to install Ubuntu 18.04 LTS (long-term support), Swift for tensorflow, and s4tf from scratch; and wrap up with the client setup process and a Unix quickstart.

Hardware

We will look at what sort of hardware you will need to build your own deep learning machine.

B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2

Don't go alone!

If you are nontechnical, then get a friend to help you with this. Find somebody in your network who can explain the process and help you with cabling and getting the hardware working, as well as the next steps (installing an OS). At a high level, you will need a GPU of some sort (to run your code as fast as possible) and then a host CPU (computer of some sort) to send commands to it.

GPU

One of my long-term hopes is that swift for tensorflow (and more broadly, MLIR) will enable you to build neural networks for whatever GPU hardware you have. In the here and now, though, Nvidia hardware is your best choice. Here are the cards I think you should have on your radar. There are three main things to look out for.

Chip architecture itself: Ampere has finally made it to market and this is what I think you should buy. Basically every past architecture is implemented and so you can run any form of research code locally with as much RAM as you desire. Pytorch 1.7 has integrated NVidia's apex libraries and so you can use this and XLA interchangeably. Being able to run bfloat16 code locally will drastically simplify the process of testing TPU code. Swift for TensorFlow doesn't currently support this well, but I think that will change. Tim Dettner has a nice article on why the Ampere architecture is a good choice if you enjoy technical detail.

Raw performance: CUDA cores are a good rough proxy for raw performance. There are various gamer cards that are overclocked to run slightly faster and cost a little more, but I would advise against them (generic hardware is fine), because:

GPU memory is the most important metric for machine learning, because it has the largest effect on what your card is capable of. More memory allows bigger models, allowing you to run more modern code. More memory allows you to put more data on the device, allowing you to train networks more quickly with larger batch sizes. Swift for tensorflow in particular isn't terribly well optimized on this front, compared to other frameworks, so this is the main limitation you will run into. Assuming whatever machine learning you do will run on your card (don't get off the beaten path with other architectures), then the memory will be the largest factor in what you can do.

GPUs to buy

I think you should target 8GB of GPU memory at a minimum. The important cards to consider are as follows.

Used/old cards: 1070Ti, 1080, 1080Ti series cards will run Swift for TensorFlow fine. Every demo in this book will run on a 1070Ti. I have had good luck with older Kepler/Maxwell era workstation cards but be aware you will need to deal with firmware and cooling.

RTX: All of the code in this book was written on a 2080 Ti in FP32 mode. If you have this already, I wouldn't throw out fp16 support yet, but if you don't have anything just jump to Ampere.

T4: 15GB, fp16, ~2070 speed, cheap in cloud (slower but use much less power)

V100: 16GB, fp16, ~2080ti speed

Cheapest Ampere NVidia: Support for every single hardware format in existence. $400, 8GB ram

3090: Support for every single hardware format in existence. $1500, 24GB ram (buy this if you can)

```
V100, v2: 32GB
A100: Ampere, 40GB/80GB
```

```
DGX-1: 8 * V100 --> 128GB
DGX-2: 16 * V100 --> 256GB
DGX-2H: 16 * V100v2 --> 512GB
DGX-A100: 8 * A100 @ 80GB --> 640GB
HGX-A100: 16 * A100 @ 80GB --> 1280GB

TPU-v2-8: 64GB
TPU-v3-8: 128GB
TPU-v3-2048: 32TB
```

In general, for computer vision, you can get away with less RAM than other areas. If you are just starting out, my advice is to get an 8GB card, and then you can upgrade down the road. I used a 1070 Ti for a year, and it was a good investment. A lot of NLP and current research will use more RAM, but you can rent a T4 on Google Cloud to experiment with this stuff for the price difference at first.

Multiple GPUs

There is something to be said for having two GPUs on your local machine for the workflow where you start a job on one device and then experiment with code on another in the meantime. In general, trying to get multiple GPUs to work together on a problem usually adds more trouble than it is worth. For paired training, you need to match GPUs to work together (or rather, they will run in sync at the speed of the slowest one) or do hardware tricks (e.g., nvbridge offers a performance boost for cross-card communication, but then you need to build your computer in a certain way so that the cards can be wired together), and so in general I would advise you not to worry about this too much unless you have a preconfigured multi-GPU cloud machine (you're not there quite yet). If I am going to use multiple GPUs, then I might as well start to use multiple computers. It's generally much easier to log in to two single-GPU machines

to do different versions of a job (and scaling from there to a hundred machines is trivial with cloud instances) than it is to do the work to make a single machine use two GPUs on the same problem at once effectively.

CPU

If you are compiling swift for tensorflow from scratch, then more cores will be of use for sure, but in general most cloud machines are four to eight cores, and you are usually going to be fine with this on a workstation (but more won't hurt if you can afford them). Reinforcement learning in particular uses lots of cores, so if that's a long-term goal of yours, then consider a beefier machine. I do a lot of work using virtual machines + batch processing in general, so I have a few more cores than usual. If you have a computer already or an old PC lying around, then that should be fine, but be aware of how much power they use. Old workstations in particular will use more electricity than you might expect sometimes. You can get a cheap wattmeter device to figure out how much energy your device actually uses. Repurposing a gaming rig is fine, but you don't need overclocked hardware or anything fancy on this front. Something that is stable is far, far more important; most of your code is going to be GPU bound anyway.

Motherboard

Getting more than 2 PCI-4 full length slots is a good future-proof option. You might consider faster network interfaces as well as remote management capabilities.

PSU

Things to be aware of: The power supply unit (PSU) determines how much electricity your computer can use at once. The GPU is going to use 250 watts or sometimes more, so make sure your system has at least twice that to handle the combined load of it and rest of the machine. Outervision has

a nice online calculator that will give you an estimate of how much power your system is going to draw. If in the long term you think you are going to buy a second GPU, then make sure your PSU will be able to handle the combined load. 750-1000 watts is a good current spot to target. Rewiring in a new PSU is a bit of a pain but might be the simplest approach to allow you to upgrade an older computer with a fancier GPU. If you've never built a computer before, ask around and you should be able to find somebody to help you with the process; it's not that hard (and is sort of a rite of passage of the aspiring machine learning practitioner).

Be aware that >1600 watts will require custom wiring.

Cooling

In the same vein, you're going to be running your graphics card and CPU for hours on end at full load, so it's important to make sure your setup has good cooling. Noctua makes good fans; find a hardware guru to guide you through getting a good heatsink setup for your particular machine. Some people like to do water cooling, but that's overkill for somebody new to this field in my opinion.

RAM

A simple rule of thumb is to have 2x your GPU RAM for your machine. 16GB will work okay for getting started, but upgrade to 32GB if you can. I think this is a good sweet spot. More doesn't hurt, but unless you know your workflows actually require more RAM than that, you should be fine.

SSD

This is a really good cheap upgrade for older computers. Buy a cheap SSD drive or, even better, an M.2 drive if your motherboard supports it and work entirely off the flash drive (e.g., install Linux to that). 128GB, 256GB, and 512GB drives can be had very cheaply, and you can do a lot of work

with this amount of space if you are on a budget. 1TB and 2TB drives are a little more but certainly affordable, and then you can keep lots of data locally. I wouldn't worry about the flash speed/online arguments about what drive is the fastest too much and would rather just have more space in general.

Cheap two and four bay m2 pci adapters exist so you can keep old drives plugged in for a long time to come, so I would consider this a reasonably future-proof investment. If you're comfortable with mounting multiple drives under Linux, then buying two drives (e.g., a small one for your OS to make reinstalling easy and a larger one for data/storage) is a nice workflow.

Recommendations

My budget system recommendation:

```
* Repurpose old computer + cheap NVidia 8gb card + cheap ram/flash
```

Build your own:

```
* CPU of your choice, Ampere GPU of your choice, motherboard
with PCI-4 slots, 16-32GB of ram, 1-2TB M.2
```

My setup:
My v1 server, hermes:

```
* 2x M40 @ 24GB, intel i7, asus prime x370, 48GB of ram, 2TB
M.2, corsair 750w/200R case
```

My v2 server, athena:

```
* RTX 2080 ti, ryzen 3950x, asus prime x570, 128GB of ram, 2TB
M.2, corsair 850w/540 case
```

SAN:

```
* 100TB @ 10 Gigabit, 1TB @ 40/56 Gigabit backbone
```

Saving up for:

```
* 2 * 3080 Ti @ 20GB (40GB)
```

My v3 server, venus:

```
* intel i9 + x399 board (eg 5+ PCI slots)
* +2 GPU --> 4x 3080Ti (80GB)
```

My v4 server, ares:

```
* 4 x 80GB setups --> 320GB
```

To infinity and beyond:

```
* a bigger gpu, faster network, more cores, more ram, more space!
```

Long term

Finally, my advice is to plan your budget with the goal of future upgrades. Broadly speaking, spending 1K today and 1K on upgrading in six months to a year is a better usage of your resources than spending 2K today. Today's high-end card that everybody is drooling over will be considered obsolete in a year more often than not. Spending more than 3K on a system is overkill until you have a better feel for how things work together and what limits you are bumping into.

Some real-world usage examples

I have been messing with various NLP models the past year, and a lot of them are designed for machines with larger GPUs (e.g., they won't run at all on consumer cards without heavy tweaking), so my next big purchase will be a card with more RAM, ideally 20GB or more.

A surprising number of data preprocessing workflows are single threaded, which means you can spend hours just watching a single thread chug away at 100% while you wait. This is a great project for your local machine where cycles are cheap, and then you copy the data up to the cloud to run actual jobs in parallel there.

A lot of computer vision stuff fits nicely onto smaller GPUs, so then the bottleneck becomes the raw speed (CUDA cores/processor) of the device. Swift for tensorflow still needs a bit of work on the memory management front and batch processing in general. People are working on it!

Another workflow I have (large data processing job) needs about 80–90GB of RAM to run comfortably, so I have bumped my RAM a bit. Likewise, I have been doing lots of compiles and running virtual machines on my primary system, so the extra cores are helpful.

If you start trying to gather your own data for a real-world project, video in particular, you are going to start needing a lot of scratch space. You can use traditional disks for this, but there will definitely be a speed hit going between the worlds.

Hardware recap

I would suggest you stick with the basics for now and get a cheap computer working, then upgrade it down the road. Until you're playing around with a bunch of things, you won't know what you need to buy or where your pain points are. Likewise, don't worry about overclocking or timing techniques or having the most advanced hardware. Just focus on getting something reliable that you will use regularly. That is the key to success.

Installing Ubuntu

Ubuntu is a popular Linux distribution that has been out for many years now. The world moves quickly, and my goal is not to let this book get out

of date. Ubuntu 20.04 LTS is out and offers a significant improvement to the distribution. Having said that, swift for tensorflow still doesn't work perfectly with it, so I am pushing 18.04 for now. As the situation changes, I will keep an online guide for you at

`convolutionalneuralnetworkswithswift.com`

This will keep you up to date on how to upgrade. In my work with 20.04 so far, nothing major has changed, just a bunch of internal library cleanup and depreciation that has predictably made running things out of the box not work. Swift 5.4 and later will support 20.04 by default, and after they ship, then swift for tensorflow will pick up 20.04 support as well.

I have used Ubuntu for a long time and think it is the best variant for you to learn. My reasoning is as follows:

1) Hardware support in general in Ubuntu is very good. After many, many generations of software now, most hardware will be supported out of the box. If something will run Linux in general, then it will usually run Ubuntu without too much fuss.

2) Ubuntu upstream is generally fairly pragmatic in general. They have continually driven solid evolution of their product by working hard to bring people from extremely varied groups together to solve real-world problems.

3) Community: The Ubuntu community is very good in general about being friendly and welcoming to outsiders. One of the benefits of having a large community of fellow users is that many other people will hit the same pain points you discover. If you have a question or bump into a configuration issue, generally, you can search "ubuntu 18.04

KEYWORD" and find blog posts and answered questions on Stack Overflow or documentation by cloud providers on how to solve your issue. This makes solving many day-to-day issues generally a few clicks away.

Nvidia driver support: This is another big one; the Ubuntu drivers for Nvidia are pretty painless to get going at this point in time compared to the trickery that you used to have to do.

Finally, Ubuntu is a default option for all of the major cloud providers. So, whatever skills you learn on your local machine will eventually translate to the cloud without too much trouble. Toward this end, we are going to install the command-line (server) version of Ubuntu 18.04 LTS (long-term support).

I personally do not believe in installing a GUI on your machine learning box, because I want you to be thoroughly familiar with the terminal. Having said that, if you have a particular window manager you want to install and are familiar with desktop Linux already, go for it.

General prep

Broadly, you will need

1) A USB key flashed with Ubuntu 18.04 LTS (use another computer to do this)

2) A keyboard to navigate the setup process and (optional) a USB mouse – my computers, with Asus motherboards, have a BIOS screen that is much easier to navigate with a mouse. If you have a mouse lying around, then plug it in!

3) A monitor for the initial steps here. Most graphics cards have an HDMI port so you can plug your machine into a TV if needed for this step. You may need to tweak the scan size to fit all the output on the screen.

4) An Internet connection. You're going to need to download some stuff, so make sure you can pull a few gigabytes down as needed.

You will need to know how to boot the post screen (e.g., you will hold down a certain button on the keyboard at launch, usually one of the function keys, to access the boot settings). If you are not familiar with this process, then find a friend who has done all this before and have them run through these instructions with you. I have streamlined this as much as possible; you can do it in under an hour easily.

And finally, I would suggest you reinstall your primary operating system periodically. Once you have jumped through all the hoops a few times, it will become second nature.

OS install prep

Please just buy a USB flash drive for this. You can get a cheap one for under $10. All you need is for it to be 2GB large. Flash it once and leave it sitting by your computer for when you need to repeat all of this again. Do not mess with partitioning your primary drive or trying to use a random caddy to convert a different kind of flash memory into a drive; I've seen this go astray too many times.

Download Ubuntu + flash to USB key

We'll be using the Ubuntu Server installer. We will be using the Live installer, which is how Ubuntu is going to be doing these installs going forward (e.g., 20.04 and later).

```
Download here: http://releases.ubuntu.com/18.04/
Actual iso (disk image): http://releases.ubuntu.com/18.04/
ubuntu-18.04.4-live-server-amd64.iso
```

Add a .torrent to the end of the preceding URLs to get a distributed download if you prefer torrents.

You can follow these guides for how to copy your ISO to your flash drive:

> Mac guide: https://tutorials.ubuntu.com/tutorial/tutorial-create-a-usb-stick-on-macos#0
>
> Windows guide: https://tutorials.ubuntu.com/tutorial/tutorial-create-a-usb-stick-on-windows#0
>
> Ubuntu guide: https://tutorials.ubuntu.com/tutorial/tutorial-create-a-usb-stick-on-ubuntu#0

Each version will have you launch a disk utility (there's a different name for each platform) which you will then use to overwrite your USB flash drive (WHICH YOU BOUGHT SPECIFICALLY FOR THIS). It will run a bit and do a verification pass, and then you should be ready for the next step. Eject your flash drive from your computer.

OS install

```
* plug into network (hard-wired cord)
* put usb key in computer
* boot + hold down bios screen button (f2 on asus, search
"BIOS KEY +MOTHERBOARD MANUFACTURER NAME")
```

PROTIP: If your computer has VT virtualization ability, then enable it now so you can do cool stuff with virtual machines later. Now is a good time to run an auto-tuner if your BIOS has one. As I have said elsewhere, I do not recommend overclocking your machine because I place an extremely high premium on stability, but if you know what you're doing and what to watch out for, then go for it.

Select the USB key (you should be able to identify the brand name in the list) as the primary boot drive. Reboot the machine (hit save and exit in the BIOS). Depending on your machine, you may need to reselect your USB device after boot and just do so manually the first time.

```
select language (en)
```

My Live installer has this screen; you probably won't see it:

```
install ubuntu server
select language – en
select location - us
detect keyboard layout - no
english - us
ubuntu network card working screen
```

If you have experience with Linux, you can configure your card later, but if this isn't autocompleting for you, then go back to step 1 and make sure you've got the computer plugged in properly. You're going to be using this interface a lot so it's super important to have this set up correctly from the start.

In the screen, just hit done, and it will autofill in the default proxy server.

Then, just hit done again to keep on going.

```
WARNING: PARTITION DISK STAGE
```

PROTIP: If you know how to do partitioning and want to dual boot, then go for it. I am not going to help you out when you mess things up. If you are new to this, just use your whole drive with ext4 (if you know how to use volume groups, then use that, but not if you are a beginner!). This is the cheapest component of your computer for a reason. If you would like to do something fancier, then install your operating system to one hard drive and mount a different one to store data. Then, you can reinstall the

OS really easily without worrying about data loss. But if you're new to all of this, then just put everything on one drive and call it a day.

Pick your m.2 drive; the thing shows the size.

Hit done again, and it will pop up a warning.

Select continue; it will be red to show you that this is a nonreversible process.

Hit continue.

It will show the screen where you will set your user info.

PROTIP: Give your computer a name that will look cool on the network, but not something you will hate typing over and over again.

```
full user name
short username
password 1
verify password
```

Fill out whatever you like. I like a simple shortname since you will be typing it a lot.

OpenSSH is NOT OPTIONAL – we will be using this a lot.

Hit done.

Extra screen

Install Postgres (optional). I like this one-click install for Postgres here; it's a super quick way to get a database up and running on your system.

You can install all of this later using snap installs, so it's not essential to get things running here.

Hit done and it will start installing.

After this, just sit back. Things will run for a while and then the computer will prompt you to restart.

One more prompt at the end asks you to remove your USB flash key.

Reboot

After a restart, you should have the standard Unix login screen.

Type in your shortname and password.

Let's run one of the universal Unix utilities you should know when logging in to a new system:

```
uname -a
Linux mercury 4.15.0-99-generic #100-Ubuntu SMP Wed Apr 22
20:32:56 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux
```

The first line is the base Unix type, the second your name, the third your actual Linux kernel version, and the rest a bunch of flags the actual compiler used to make the kernel was given. You're now a Unix user (soon to be a wizard)!

Let's test our hardware a bit before we go too far. Try pinging a remote server to make sure your network card + connection is working properly:

```
ping google.com
````
PING google.com (172.217.1.206) 56(84) bytes of data.
64 bytes from iad23s26-in-f206.1e100.net (172.217.1.206):
icmp_seq=1 ttl=56 time=23.6 ms
64 bytes from iad23s26-in-f206.1e100.net (172.217.1.206):
icmp_seq=2 ttl=56 time=23.1 ms
64 bytes from iad23s26-in-f206.1e100.net (172.217.1.206):
icmp_seq=3 ttl=56 time=23.4 ms
64 bytes from iad23s26-in-f206.1e100.net (172.217.1.206):
icmp_seq=4 ttl=56 time=23.4 ms
^C
```
```

```
--- google.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3003ms
rtt min/avg/max/mdev = 23.130/23.406/23.651/0.240 ms
```

Hit Ctrl+C to stop ping from running forever. This is a universal Unix command to kill a program; you'll be using it a lot, so learn it now.

Doing a sanity check of your new server

Before we go too much further, it's a good idea to do a quick stress test of your system to make sure that all your hardware is working properly. Let's install stress:

```
sudo apt install stress
```

Run stress on all your cores for a few minutes to test the CPU. Make a new terminal and run htop to see what's going on:

```
stress -c NUMBEROFCORESONYOURMACHINE
```

It's not filling up the memory bar in htop. All you really need is to be allocating enough RAM that each stick of memory is being used slightly (e.g., if you only have two sticks, then if anything over 50% is working, that means both sticks are fine).

```
stress --vm 8 --vm-bytes 4000M
```

If you get warnings here, you're going to need to find some better RAM before going further. Otherwise, your system will do all sorts of weird things internally, and things will randomly fail.

Ubuntu recap

Congratulations, you've successfully installed Linux to your computer.

Installing swift for tensorflow

Now let's install CUDA drivers to our machine learning box, then get a prebuilt version of s4tf up and running with the swift-models repository.

Installing graphics card drivers and swift for tensorflow

Let's install some drivers for your graphics card, and then we can install and run swift for tensorflow using our graphics card. Swift for tensorflow now supports CUDA 11.0, so use that (or whatever the latest supported version is) if possible.

CUDA 10.2 install process

Click https://developer.nvidia.com/cuda-downloads and to go to legacy download page for CUDA 10.2 drivers (the current highest-supported version of CUDA for swift for tensorflow). We want this specific version and related libraries to make our lives simpler (Linux --> x86_64 --> Ubuntu --> 18.04 --> deb (network)).

Follow the install guide:

```
wget https://developer.download.nvidia.com/compute/cuda/
repos/ubuntu1804/x86_64/cuda-ubuntu1804.pin
sudo mv cuda-ubuntu1804.pin /etc/apt/preferences.d/cuda-
repository-pin-600
sudo apt-key adv --fetch-keys https:// developer.download.
nvidia.com/compute/cuda/repos/ubuntu1804/ x86_64/7fa2af80.pub
sudo add-apt-repository "deb http://
```

```
developer.download.nvidia.com/compute/cuda/repos/
ubuntu1804/ x86_64/ /"
sudo apt-get update
sudo apt-get -y install cuda
```

Installing cudnn

We also need to install libcudnn and libcudnn-dev as well, which isn't quite as easy as the previous one. They are system libraries that swift for tensorflow uses internally, and you will need to have them installed on your system in order for swift for tensorflow to use your graphics card.

You will need to go to the NVidia site and sign up for an account. Then, you will get an email. Then you can go and log in to see the main dev console. Basically, you will need to download these files from the Nvidia site and then upload them yourself to your server.

https://developer.nvidia.com/cudnn

You will need both the cudnn runtime library **AND** dev library (note: you can't access these files without being logged in!) Anyway, assuming you got the cudnn files to your server, let's install libcudnn:

```
sudo dpkg -i libcudnn7_7.6.5.32-1-i-cuda10.2_amd64.deb
sudo dpkg -i libcudnn7-dev_7.6.5.32-1-i-cuda10.2_amd64.deb
```

Now, let's reboot your computer to test things:

```
sudo shutdown -r now
```

Afterward, we should be able to call nvidia-smi:

```
> nvidia-smi
```
```
Sat Apr 25 19:09:04 2020
+-----------------------------------------------------------+
| NVIDIA-SMI 440.64.00 Driver Version: 440.64.00 CUDA
Version: 10.2 |
|-----------------+-----------------+-------------------+
| GPU Name    Persistence-M| Bus-Id    Disp.A |
Volatile Uncorr. ECC |
| Fan Temp Perf Pwr:Usage/Cap| Memory-Usage | GPU-Util
Compute M. |
|=============================+=====================+
 ======================|
| 0 Tesla T4 On | 00000000:00:04.0 Off |
0 |
| N/A 70C P8 12W / 70W | 0MiB / 15109MiB | 0%
Default |
|---------------------+-----------------+--------------+
|-----------------------------------------------------------+
| Processes:
GPU Memory |
| GPU PID  Type  Process name
Usage |
|
=============|
| No running processes found
|
+-----------------------------------------------------------+
```

Installing swift for tensorflow using prebuilt packages

Next, let's get a working swift install using an official release of swift for tensorflow. This should be pretty painless, and by the end, we'll be doing actual deep learning on your computer using all the pieces we've put together so far.

Download swift

Our base guide is here:

```
https://github.com/tensorflow/swift/blob/master/ Installation.md
```

Log in to your server, then run

```
mkdir swift
cd swift
```

First, install some required libraries:

```
sudo apt-get install clang libpython-dev
libblocksruntime-dev
```

Download the actual file (latest version + cuda10.2 build):

```
https://storage.googleapis.com/swift-tensorflow-artifacts/
releases/v0.12/rc2/swift-tensorflow-RELEASE-0.12-cuda10.2-
cudnn7-ubuntu18.04.tar.gz
```

Extract things locally:

```
tar -xvf swift-tensorflow-RELEASE-0.12-cuda10.2-cudnn7-
ubuntu18.04.tar.gz
```

Now, we should be able to point our shell at the swift for tensorflow binary files:

```
export PATH=~/swift/usr/bin:"${PATH}"
```

And then we can actually run swift:

```
$ swift --version
Swift version 5.3-dev (LLVM 5a342fdfac, Swift 5c3508b5d6)
Target: x86_64-unknown-linux-gnu
$ which swift
/home/skoonce/swift/usr/bin/swift
```

Congratulations, we have a working swift for tensorflow install! Let's play with it a little.

Python

Set up Python:

```
sudo apt install python python-dev python-pip
python --version
> Python 2.7.17
pip --version
> pip 9.0.1 from /usr/lib/python2.7/dist-packages (python
2.7)
git clone https://github.com/tensorflow/swift-models.git
cd swift-models
swift run
 Fetching https://github.com/apple/swift-protobuf.git
Fetching https://github.com/apple/swift-argument-parser
Cloning https://github.com/apple/swift-argument-parser
Resolving https://github.com/apple/swift-argument-parser
 at 0.0.2
 Cloning https://github.com/apple/swift-protobuf.git
```

```
Resolving https://github.com/apple/swift-protobuf.git at 1.7.0
error: multiple executable products available: pix2pix,
Catch, Gym-Blackjack, Gym-CartPole, Gym-FrozenLake,
Autoencoder1D, Autoencoder2D, Benchmarks, VGG-Imagewoof,
Regression-BostonHousing, Custom-CIFAR10, ResNet-CIFAR10,
LeNet-MNIST, MobileNetV1-Imagenette, MobileNetV2-
Imagenette, GAN, DCGAN, BERT-CoLA, FastStyleTransferDemo,
MiniGoDemo, GPT2-Inference, GPT2-WikiText2, CycleGAN
swift run LeNet-MNIST
[15/15] Linking LeNet-MNIST
Loading resource: train-images-idx3-ubyte
File does not exist locally at expected path: /home/
skoonce/
swift/swift-models/t
rain-images-idx3-ubyte and must be fetched
....
[Epoch 11] Training Loss: 741.2036, Training Accuracy:
52567/59904 (0.8775207),
Test Loss: 123.73525, Test Accuracy: 8734/9984 (0.87479967)
[Epoch 12] Training Loss: 740.11896, Training Accuracy:
52683/59904 (0.8794571), Test Loss: 123.62877, Test Accuracy:
8737/9984 (0.87510014)
```

Verify you're using a GPU

While your MNIST program is running, start a new shell in tmux and run nvidia-smi:

```
> nvidia-smi
```

```
Sat Apr 25 19:14:08 2020
+-----------------------------------------------------+
```

```
| NVIDIA-SMI 440.64.00   Driver Version: 440.64.00 CUDA
Version: 10.2   |
|-------------------+------------------+--------------+
| GPU Name Persistence-M| Bus-Id    Disp.A |
Volatile Uncorr. ECC |
| Fan Temp Perf Pwr:Usage/Cap| Memory-Usage | GPU-
Util Compute M. |
|
======================+==================+==============|
|   0 Tesla T4   On    | 00000000:00:04.0 Off |
0 |
| N/A 52C  P0   27W / 70W |   886MiB / 15109MiB | 0%
Default |
|-----------------+------------------+----------------+
+-----------------------------------------------------+
| Processes:
GPU Memory |
| GPU        PID        Type        Process name
Usage        |
|
=============|
| 0 4004 C .../_64-unknown-linux-gnu/debug/LeNet-
MNIST 875MiB |
+-----------------------------------------------------+
```

Autoencoder demo

Let's try a slightly different demo next:

```
```

```
swift run GAN
swift run GAN
[6/6] Linking GAN
Loading resource: train-images-idx3-ubyte
Loading local data at: /home/skoonce/swift/swift-models/
train-
images-idx3-ubyte
Succesfully loaded resource: train-images-idx3-ubyte
Loading resource: train-labels-idx1-ubyte
Loading local data at: /home/skoonce/swift/swift-models/
train-
labels-idx1-ubyte
Succesfully loaded resource: train-labels-idx1-ubyte
Loading resource: t10k-images-idx3-ubyte
Loading local data at: /home/skoonce/swift/swift-models/t10k-
images-idx3-ubyte
Succesfully loaded resource: t10k-images-idx3-ubyte
Loading resource: t10k-labels-idx1-ubyte
Loading local data at: /home/skoonce/swift/swift-models/t10k-
labels-idx1-ubyte
Succesfully loaded resource: t10k-labels-idx1-ubyte
Start training...
[Epoch:1]  Loss-G: 1.1234643
[Epoch:2]  Loss-G: 1.1434635
[Epoch:3]  Loss-G: 1.1433427
```

```
```

```
swift run ResNet-CIFAR10
Downloading CIFAR dataset...
Archive missing, downloading...
```

```
Archive downloaded, processing...
Unarchiving completed
Starting training...
```

Reinforcement learning demo

First, add the gym reinforcement learning environment using pip:

```
pip install gym
```

Now you can run bot that teaches itself how to play Blackjack using Q-learning (and other strategies):

```
swift run Gym-Blackjack
Solver: random, Total reward: -4002 / 10000 trials Solver:
markov, Total reward: -2323 / 10000 trials Solver:
qlearning, Total reward: -2035 / 10000 trials Solver:
normal, Total reward: -1022 / 10000 trials
```

This is a good stopping point for today. Go tell your friends you're experimenting with GANs and reinforcement learning in Swift for Tensorflow!

Swift for Tensorflow recap

We installed our Nvidia drivers, then got swift for tensorflow working using a prebuilt package. Then we made sure we had Python working as well and ran some prebuilt demos. Next, we'll repeat the preceding process in the cloud.

Installing s4tf from scratch

In this section, we will look at how to build Swift for Tensorflow from source, for those who want to understand how things work under the hood.

There be dragons here

Please skip to the next section if you're new to building software libraries from scratch. You don't need any of this to do the exercises in the rest of the book; this simply seemed the most logical place to put these instructions. I would only advise trying to do this if you're on a Linux box and have time to spare.

How to build swift for tensorflow from scratch

Here is a quick guide to building swift for tensorflow from scratch. This is overkill for most users, but it is a good way to keep up to date on what is going on with the project day to day. If you would eventually like to help contribute to the project, you will need to be able to make changes locally, compile them, and then eventually make a pull request upstream.

At a high level, we will

1) Set up our system to compile swift for tensorflow from source (only need to do this once)
2) Check out the actual codebase (will do this everytime you need to sync with upstream)
3) Build/compile swift for tensorflow from source

Prerequisites

I'm assuming you've followed the sections to get here. Therefore, you have

1) Ubuntu 18.04 LTS base installed
2) Python support
3) CUDA and cudnn installed
4) 100+ GB of free disk space

Installing cmake

If you don't have the right version of cmake, you will see weird errors like this halfway through the install process:

```
cmake errors w/ 3.10.2 (ubuntu 18.04 default)
```

```
/usr/bin/ar: creating t.a
-- Building with -fPIC
CMake Error at cmake/modules/SwiftHandleGybSources.cmake:4
(find_package):
By not providing "FindPython2.cmake" in CMAKE_MODULE_
PATH this project has asked CMake to find a package
configuration file provided by "Python2", but CMake did
not find one.

Could not find a package configuration file provided by
"Python2" with any of the following names:
  Python2Config.cmake
  python2-config.cmake
Add the installation prefix of "Python2" to CMAKE_PREFIX_
PATH or set
```

```
"Python2_DIR" to a directory containing one of the above
files. If "Python2" provides a separate development
package or SDK, be sure it has been installed.
Call Stack (most recent call first):
CMakeLists.txt:495 (include)
```

Currently, swift for tensorflow requires a newer version of cmake than what Ubuntu is providing. As a general rule with build tools, **only use the recommended versions that your package requires**. In the same vein, try to **use the same operating system and configuration as the developers**. I have spent many, many hours of my life chasing weird errors that arise when you are using slightly different build tools or libraries, and I do not want you to have to do the same. Kitware has a nice repository set up for Ubuntu/Debian people here: https://apt.kitware.com.

Add their signing key to your computer:

```
wget -O - https://apt.kitware.com/keys/kitware-archive-
latest.asc 2>/dev/null | sudo apt-key add -
[sudo] password for USERNAME:
> OK
```

Add their repositories to your package lists:

```
sudo apt-add-repository 'deb https://apt.kitware.com/
ubuntu/ bionic main'
sudo apt-get update
```

Now, you should be able to do a

```
sudo apt-get install cmake
```

```
Reading package lists... Done
```

```
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
cmake-data
Suggested packages:
cmake-doc ninja-build
The following NEW packages will be installed:
cmake cmake-data
0 upgraded, 2 newly installed, 0 to remove and 27 not
upgraded.
Need to get 9,109 kB of archives.
After this operation, 35.4 MB of additional disk space
will be used.
Do you want to continue? [Y/n]
```

And then afterward, you can check that you have a recent version:

```
```
cmake --version
cmake version 3.17.1
CMake suite maintained and supported by Kitware (kitware.
com/cmake)
```
```

Packages we need

Let's install some dependencies that swift for tensorflow will need using the built-in Ubuntu repositories:

```
>    sudo apt-get install git cmake ninja-build clang
python uuid-dev libicu-dev icu-devtools libedit-dev libxml2-dev
libsqlite3-dev swig libpython2-dev libncurses5-dev pkg-config
libcurl4-openssl-dev libblocksruntime-dev systemtap-sdt-dev
tzdata rsync
```

Bazel

Bazel is a build system for building large projects in general and tensorflow in particular. You'll need to get a specific version of Bazel for s4tf, which is usually slightly behind the official version by a few months. Check the docs for the currently supported version; it was 2.0.0 when I did this. If you're interested in where the actual file that specifies this is, look at the configure.py file in the tensorflow directory:

```
... _TF _MIN _BAZEL_VERSION = '2.0.0'
_MAX_BAZEL_VERSION = '2.0.0'
...
```

I then would use the max one if there's a difference.

Follow the guide here: https://docs.bazel.build/versions/master/install-ubuntu.html

```
    ```
    sudo apt install curl
   curl https://bazel.build/bazel-release.pub.gpg | sudo apt-key
    add -
    echo "deb [arch=amd64] https://storage.googleapis.com/
    bazel-apt
    stable jdk1.8" | sudo tee /etc/apt/sources.list.d/bazel.
    list
    sudo apt-get update
`   ``
    Install bazel 2.0.0:
    sudo apt install bazel=2.0.0
    $ bazel --version:
    bazel 2.0.0
```

Yay!
```

Fetch swift for tensorflow sources

Create a high-level directory:

```
cd ~/swift
mkdir swift-source
cd swift-source
```

Get the source:

```
git clone https://github.com/apple/swift.git -b tensorflow
./swift/utils/update-checkout --clone --scheme tensorflow
```

This might take a while!

What a checkout will look like (different hashes)

```
```
/home/ubuntu/swift/swift-source/llvm-project
+ git submodule update --recursive
update-checkout succeeded
PythonKit   :
d921e19555e50b39606d528f2b3a7990a9cd6ce0
cmake   : skip
cmark   :
1168665f6b36be747ffe6b7b90bc54cfc17f42b7
icu   :
fd123bf023882f07bfacf51c39111be2f946d8f8
indexstore-db   :
b99f773fa83640174c41e580e6ddda2abc617367
llbuild   :
435e72e01369ed1397fa01e9f564a299c7e2095a
llvm-project   :
8725f77b49d1e41f87f11e3a1f275e9a519a8380
```
```

```
ninja                                :
ed7f67040b370189d989adbd60ff8ea29957231f
sourcekit-lsp                        :
babf190c1886ea3c1b4fe6017285a181ed09b182
swift                                :
9c4212a49b483f455bcd1caf4de6f2ef12a5ad54
swift-corelibs-foundation            :
eac91abb1d74f14ecf7cc788b25eb566e7d550f2
swift-corelibs-libdispatch           :
80b177209ca6960f42da07121cf86abc59ce3980
swift-corelibs-xctest                :
8b0eefa96c02a4cb4d3eb74e6c289eba34a744fa
swift-format                         :
6207f97c8602d4c34b84b0fec79759a596e50aa1
swift-integration-tests              :
11f0f6e8b34ba9782b5841dbeaa207d0b4620152
swift-stress-tester                  :
225a973f42140890bb49648ca090d1a53f32ff8e
swift-syntax                         :
1e524b3edc47e8ff66890d914b8bcd024e061631
swift-tools-support-core             :
693aba4c4c9dcc4767cc853a0dd38bf90ad8c258
swift-xcode-playground-support       :
88043d7d320f92598efb39408c3f4b1903a4fff6
swiftpm                              :
afc7bd9efb0f0a0e53b060d30e47a876a888cc86
tensorflow                           :
3c1e8c03419266bb6ba379d303d3e03a380617a8
tensorflow-swift-apis                :
aad3149d56350736fca45f4c84614eb6c1ac4cdf
```

Python 2 install + packages needed

We'll need some Python libraries in order to compile swift for tensorflow. Get a working pip install if you haven't done so already:

```
sudo apt install python python2-dev python-pip
```

Now let's install some libraries:

```
pip install six numpy future
```

Not having these will produce various weird import errors halfway through compilation.

Build swift for tensorflow from source with GPU support

There are two modes for building and running swift for tensorflow, one based around eager execution (Tensorflow 2) and then XLA. We will be working with the XLA variant as I feel that is the future of the project.

XLA builds are the default, but you will need to set an environment variable in order to get swift for tensorflow to build for your CUDA device as well:

```
export TF_NEED_CUDA=1
```

After this, we only need one command (+ something else to do while you're waiting for this to complete):

```
swift/utils/build-toolchain-tensorflow
```

Sample console output steps:

```
-DLLVM_VERSION_MAJOR:STRING=10 -DLLVM_VERSION
MINOR:STRING=0
-DLLVM_ VERSION_ PATCH:STRING=0 -DCLANG_ VERSION
 MAJOR:STRING=10
```

```
-DCLANG_ VERSION MINOR:STRING=0 –DCLANG _VERSION _
PATCH:STRING=0
-DCMAKE MAKE PROGRAM=/home/skoonce/swift/swift-source/build/
buildbot_ linux/ninja-build/ninja
-DLLVM_ ENABLE ASSERTIONS:BOOL=TRUE '-
DLLVM TARGETS TO_BUILD=X86;ARM;AArch64;PowerPC;SystemZ;Mi
ps' '-
DCMAKE_ C_ FLAGS= -Wno-unknown-warning-option -Werror=
unguarded-
availability-new -fno-stack-protector' '-DCMAKE_CXX_FLAGS= -
Wno-unknown-warning-option -Werror=unguarded-availability-
new -
fno-stack-protector' '-DCMAKE_C_FLAGS_RELWITHDEBINFO=-O2 -
DNDEBUG' '-DCMAKE_CXX_ FLAGS_ RELWITHDEBINFO=-O2 -DNDEBUG'
-DCMAKE_ BUILD_ TYPE:STRING=Release
-DLLVM_TOOL_ SWIFT_BUILD:BOOL=NO -DLLVM_ INCLUDE_ DOCS:
BOOL=TRUE
DLLVMENABLE LTOSTRING DCOMPILER RTINTERCEPTLIBDISPATCHON
-DLLVM_TOOL_ COMPILER_ RT_ BUILD:BOOL=TRUE
-DLLVM_ BUILD_ EXTERNAL_ COMPILER_ RT:BOOL=TRUE '-DLLVM_
LIT ARGS=-v
--time-tests -j 32' -DCMAKE_INSTALL_PREFIX:PATH=/usr/
-DINTERNAL_ INSTALL_ PREFIX=local '-
DLLVM_ ENABLE_ PROJECTS=clang;compiler-rt;clang-tools-extra'
-DLLVM_TOOL_LLD_BUILD:BOOL=TRUE /home/skoonce/swift/swift-
source/llvm-project/llvm
/usr/bin/ar: creating t.a
-- Version: 0.0.0
-- Performing Test HAVE_THREAD_SAFETY_ATTRIBUTES -- failed to
compile
-- Performing Test HAVE_GNU_POSIX_REGEX -- failed to compile
```

```
-- Performing Test HAVE_POSIX_REGEX -- success
Pfi Tt HAVES
```
```
Total performance tests executed: 1
--- Finished tests for benchmarks ---
--- Creating installable package ---
-- Package file: /home/skoonce/swift/swift-source/swift/
swift-
tensorflow-LOCAL-2020-05-06-a-ubuntu18.04.tar.gz --
+ pushd /home/skoonce/swift/swift-source/swift/swift-nightly-
install/
"/swift/swift-source/swift/swift-nightly-install "/swift/
swift-
source/swift
+ tar -c -z -f /home/skoonce/swift/swift-source/swift/swift-
tensorflow-LOCAL-2020-05-06-a-ubuntu18.04.tar.gz --owner=0 --
group=0 usr/
+ popd
"/swift/swift-source/swift
real    32m9.126s
user    44m41.600s
sys     1m40.160s
```

If you get an error, wait a day and try a new checkout and build again. If that doesn't work, then see the following section on how to reset your build cache.

Running our swift binary

At this point, we should be able to use the following command so that our shell/interpreter can find our locally built install of swift:

```
export PATH="/swift/swift-source/swift/swift-nightly-
install/ usr/bin:"${PATH}"
```

The preceding code assumes that you're following my build directory layout. If you've moved it elsewhere, I would suggest you just hard-code things going forward, using something like the following (you'll need to put your Unix username into the path):

```
/home/USERNAME/swift/swift-source/swift/swift-nightly-
install/
```

Reset your build artifacts

Sometimes, a number of upstream packages or imports will get changed, and then tensorflow will no longer build from source because it's pulling in files you've built cleanly on your system already but are no longer what the source code expects. Before spending a lot of time trying to chase random build errors, you should try to do a clean build first. The simplest solution is to delete everything the build directory in your "~/ swift/swift-source/ build" folder with an rm -rf * command (please be careful).

If that is not enough, resetting the Bazel build cache is a good idea as well. This one is hidden in an invisible directory in your home folder:

```
cd ~/.cache/bazel
rm -rf *
```

Some of the various scripts will use the tools that have been built (e.g., compile parts of swift for tensorflow to build other parts of swift for

tensorflow, also called bootstrapping), so you may have to delete the entire "~/swift/swift-source/swift/ swift-nightly-install/usr/" as well.

Also, swift itself will produce build artifacts (e.g., there will be an invisible ".build" folder in the base of the folder where you are running "swift run" commands), so sometimes you will need to delete that as well.

If you've tried all of the given solutions, remember sometimes upstream actually is broken, so wait a day for a new update from the source and try again. If that still isn't working, then file a bug!

Installing s4tf from scratch recap

We've installed the various compiler tools needed to build swift for tensorflow ourselves, then downloaded the latest checkout of the source code, and built it from scratch. From here, making your own contribution to the project is simply a matter of tweaking one of the files on your computer, making the needed changes, building/testing, and making a pull request!

Client setup process + Unix quickstart

We will look at how to configure a client computer to connect to your deep learning box.

Setting up your client computer/crash course in Unix

At a high level, here is all my advice on how to configure the computer you are going to use to log in to your remote machine and then a crash course in various remote utilities you should learn.

I have a MacBook Air for this purpose, but any machine that can run a shell will do. You do not need a fancy machine for this, but rather something you don't mind carrying around. Find one with a keyboard you like and with hopefully a light power supply. Bring a mouse (see below) and a sheaf of blank paper + pen to take notes with. A cellphone Internet connection means you can do machine learning anywhere, anytime.

General config

I usually configure my terminal to be large with a green on black background. Not only does this look cool (which is obviously crucial), it is also much easier on your eyes after long sessions. You should also take care to make sure that you are using your keyboard in a manner that you will not be hurting your wrists. Repetitive stress injuries are a real thing!

In the same manner, I like trackpads, but I have found that in the long term you should make sure you are using a mouse whenever possible to reduce the amount of strain you are putting on your wrists. Taking periodic breaks is good practice as well! Short walks will not only exercise your body but often allow you crucial perspective on what you're working on. Try it. The computer will always be there.

Configuring your network for remote access

You should set up your network to allow inbound forwarding to your Linux box from other networks. Then, you can access your machine whenever/wherever you have Internet access. This means you will be able to check and start new machine learning jobs whenever needed. This is another good one to find a guru for.

Setting up a VPN (ideal but more complicated)

VPNs are a nice way to connect to your network remotely so your computer simply seems to be on the local network. To do so, though, you need something running VPN software on your local network. I would advise against doing so on your deep learning box because then you will have another thing to worry about if you ever need to replace your operating system. Most consumer routers have VPN utilities you can enable, but my experience is that they often don't have great performance, require the use of some weird security protocol variant, or have memory leaks that require periodic reboots. If you're familiar with VPNs already, then great, use your existing config. Otherwise, I would advise you not to mess with your router's utilities directly, but rather set up WireGuard on a Raspberry Pi to access your network. This can be combined with the next step as an additional layer of protection (e.g., you would forward packets from the Internet --> modem --> VPN server --> use that to access your server on the local network). With Ubuntu 20.04, WireGuard is going to officially be part of the Linux kernel, and my hope is that the VPN setup process will finally be painless.

Setting up port forwarding

For outside access, a simpler solution instead is to set up port forwarding on port 443 to your local computer. This is slightly nonstandard, but basically, most real-world firewalls (e.g., network blockers) will let you talk to computers over this port since it is the basis of the https protocol. This in turn means then that many guest networks you can access (e.g., random coffee shops, networks designed to block casual hackers) will not block your traffic, and you can connect to your box remotely without having to do anything too clever involving redirects. If a network is messing with your traffic at the packet level, then find a different one to use.

```
WARNING: You have a port open.
```

P.S. I am going to assume you know how to edit files!

Crash course in tmux

Tmux is a super useful skill you're going to need to know. At a high level, it's a virtual terminal, just like what you're using to log in right now. The key concept is that it runs on the server for you. This means that currently when you type a command, the computer is being controlled by your little terminal program. If you shut the window (which happens) or, even more fun, the network connection goes offline for a second (hello wifi), then that program disappears and, by extension, your program! So with tmux, we will be running a server, on our server, that we will then connect to and relay all of our messages through. The practical upshot of this is that we can start a job on the server inside a tmux session, close our laptop, go someplace else, start a new terminal, and reconnect to our tmux session to see how our job is coming along.

- Starting tmux
- Creating windows
- Switching between windows
- Detaching from a session
- Resuming a session
- Naming a session
- Scrolling through logs/deep into terminal (checkme: setting the scrollback option higher)

The second level of this is we can start running nested tmux sessions, but we will avoid this too much for now. But I usually run tmux on my server (which I connect to) and then make new tmux panes for each cloud server (Google Cloud) that I connect to, where I run tmux there as well. Then, I can jump between multiple machines very easily.

APPENDIX C

Additional Resources

Python --> swift transition guide

The following is a discussion of the differences of swift for existing Python machine learning practitioners.

Many machine learning practitioners come from the world of Python. The purpose of this chapter is to explain the major differences between Swift and Python for someone with a background in Python-based ML already.

Python 3

Use this in general and don't deal with Python 2; it is deprecated in general for Tensorflow as of 2.1, for Pytorch as of 1.4, and for s4tf as well. Please make the switch!

REPL

If you are familiar with the Python REPL, then the good news is Swift proper has a nice REPL with many similar features to what you are used to. If you download regular (non-s4tf) swift, you can fire it up and play with it!

The bad news is swift for tensorflow has disabled access to this in their branch, encouraging people to use Swift-Jupyter instead.

B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2

```
> swift
```

The swift for TensorFlow toolchain does not support the Swift REPL. Colab (https://github.com/tensorflow/swift/blob/master/Usage.md#colaboratory) and Swift-Jupyter (https://github.com/google/swift-jupyter) are supported alternatives.

Hopefully, this will eventually be reenabled. If you rely on a debugger, check out LLDB.

Python --> Swift bridge

This is nice for the ability to bring existing Python code and libraries into the world of swift, but trying to go back and forth (e.g., send values back to Python to run a library call there) is going to set yourself up for pain. If you need to bring something over, using imports is fine, but in general I would view it as a one-way street.

Python --> C bridge

This is less of a pain point if you are willing to go all in on Swift. Swift makes it easy to build library wrappers and add C code to projects in general, given its Objective-C heritage. If you're willing to spend some time battling cmake, then you should be able to write a swift wrapper for your code, and then you can use your library with type safety and a clean interface in general.

Python libraries

This is a big one for Python programmers in general. If you're used to being able to accomplish a large part of your workflow using Python libraries for various tasks (e.g., data cleanup, import/export), then this is going to be the hardest part of making the transition. Most of the ML ecosystem won't have swift equivalents for some time yet. There are various library proposals

upstream, but until s4tf gets a bit more traction in general, you should plan on having to do things yourself. The broader iOS ecosystem has a large collection of libraries that are worth playing with, and swift package manager has made bringing these codebases into your project much simpler.

Self-study guide

Here are some resources you can look at to help you on your machine learning journey.

Things to study

This book is designed that you will need nothing beyond access to compute resources and time to get through it. Having said that, here is some material that I have found useful.

Python

A lot of existing machine learning is in Python, and it is a valuable language to have in your toolbox. I am not a language zealot. I have written a lot of Python code to date, and swift for tensorflow still has a long way to go before it can be considered possible to do things completely in swift. Swift for tensorflow includes Python bindings that let you import most Python language tools to use existing libraries and techniques easily, so I feel like this will be a complementary skill for some time yet.

Swift

If you are interested in swift, there is an Apple book on the subject that is worth reading. If you can get a Mac with access to Xcode (a free download from the app store), then you can use the playgrounds and debugger there, which is a good way to learn. There is an iPad app you can play with as well.

Erica Sadun has good books and a blog on swift programming. Mattt Thompson is producing content at https://flight.school. Objc.io has a lot of material on swift and functional programming in general.

iOS/Android

I have done a lot of mobile development the past few years for companies big and small. If you have a working knowledge of swift and a Mac, I would highly recommend you play with Xcode and iOS to learn how to make apps that can run on a mobile device.

In particular, I highly recommend the Big Nerd Ranch books on iOS development. They take you through building a real-world application from scratch without relying too much on library magic. They have a similar book on Android as well that is a good way to get started on that platform.

Ray Wenderlich has built a little empire around keeping developers up to date on the latest tools and techniques. There is a ton of interesting content on their website available for free that is good to keep an eye on.

Tensorflow

Tensorflow is a large subject to tackle and is continually evolving, so it is difficult to have easy recommendations. Tensorflow 2 has currently forced a transitional period where existing workflows will have to be rebuilt for the new platform and will take some time to shake out.

Keras is the best way to get started with Tensorflow in general. It is very high level, which allows you to do many cool tricks easily. The flip side of this is that it sometimes hides the magic which can make understanding what is really going on difficult. There are lots of nice Keras demos on the Internet. You can get up to speed with running other people's demo code, and then you can start to build a knowledge base of techniques you have played with.

Google has Colab, which are free online shared GPU instances that can run many notebooks (collections of interactive code) in the cloud, so that is a good resource to be aware of. Swift is also available on Colab so you can run a lot of demos just using this platform.

Pytorch

I think having multiple competing programming paradigms is good for the ecosystem in general. Pytorch is a good framework to know, and understanding the basics of both it and Tensorflow will serve you well. I do not feel like I really understood how Keras worked under the hood until I did a bunch of work with pytorch and had to relearn how to explain to the computer what I wanted it to do. Pytorch in general is extremely flexible, and there is a large active community of people who are exploring the frontiers of machine learning research, using it or porting work from other frameworks to it. As a result, you can download and run pytorch demos for most interesting new research papers a few months after they come out. A good rosetta stone in my opinion is to find side-by-side implementations of something like MNIST in pytorch/tensorflow and go through them line by line to understand their similarities/differences. With Tensorflow 2, the two frameworks have become very close, and being able to jump between these frameworks in theory gives you the best of both worlds.

fast.ai

I am an extremely big fan of the fast.ai courses and have gone through their notebooks many times now and attended the in-person classes whenever possible. The hardware I would suggest you buy and set up for this course will work for doing the fast.ai courses on your own. It is taught in a Jupyter Notebook–style approach, which is extremely effective for certain kinds of problems. I would encourage you to be able to run all of their tutorials end to end at the very least and understand roughly what is going on. You will learn a lot of the unfun data cleanup and preprocessing bits that you

will have to understand when you make the transition to custom machine learning down the road. The 2019 sequence of the fast.ai deep learning from the basic course is a great way to understand the low-level bits of deep learning and how we can convert them into swift. They have a book on this subject you should check out as well.

Cloud computing

I would suggest you learn enough to understand the basics and how to use the platform to accomplish small tasks. After that, the best way to improve is just by doing a little bit of usage day by day. Running servers 24/7 can consume a lot of resources in a hurry, but creating a server, clicking around on it for an hour, and then deleting it can be done for literally pennies. Once you repeat this process many, many times, you will be extremely comfortable with cloud instances. Then, you can layer other techniques on top such as spot instances and server preconfiguration techniques such as Docker/Kubernetes and be able to jump between different clouds easily.

Ekaba Bisong has a nice book on machine learning and Google Cloud from Apress that you should check out.

TPU

TPU usage is beyond the scope of this book. I wanted to include it, but swift for tensorflow support isn't where I would feel comfortable recommending it to people new to the field. Having said that, if you are comfortable with cloud computing and Python in general, the Google team has an excellent set of TPU tutorials that you should run to learn how to use their platform. They walk you through a number of state-of-the-art approaches in different fields that you should be familiar with. Running these experiments from start to finish will consume a lot of resources + money, so I would not recommend that.

But, you can set up spot instance TPUs and start the code/training loops to get a feel for the process, then delete your machines and assume

that the code will run to completion. This can be done for a few hundred dollars in resources and is an extremely efficient way to learn a number of advanced cloud techniques. At the very least, do the Tensorflow 2 demos as a starting point to understanding how they work.

Unix

The best way to learn Unix and machine learning in general is by doing it, not reading about it. You will have to learn this by typing lots of cryptic commands into the console a bunch yourself. If you can find somebody who can walk you through the basics to get started, that will make your life much simpler. If you can find a Unix sysadmin who can teach you the dark arts, even better. However, you don't need a terribly advanced understanding, just a willingness to learn.

Git + Unix + etc

This video course covers a bunch of practical material that many people struggle with:

```
https://missing.csail.mit.edu
```

Other machine learning compiler–related projects

Here are some non-s4tf but interesting machine learning compiler (see last chapter) projects to know about.

Jax is an interesting Google project to simplify the process of generating XLA (what s4tf is moving to using under the hood) code from numpy. You can think of it as a much simpler route for converting existing ML workflows to code that will run at scale. It has an ecosystem of tools around it similar to the broader Tensorflow ecosystem.

```
google/jax google/trax haiku
flax
```

Pytorch is working on improving their JIT capabilities and being able to scale to new hardware more easily in the future by using the Glow compiler; it is worth looking at:

```
pytorch/pytorch glow
```

MXNet is slightly off the radar of many ML practitioners, but they have a project called TVM exploring the same space:

```
mxnet tvm
```

Julia is another high-level functional programming language (think Lisp for supercomputers) that is working on utilizing MLIR as an export module for their entire ecosystem and flux ML library in particular:

```
julia flux
```

System monitoring/utilities

You're going to need to keep track of what your computer's up to. Here are some useful Unix monitoring tricks.

Make sure your cores are working: htop you should see all of your cores here. I'm running a large job just to look cool:

```
```
1   [||||||80.6%]   4 [||||||80.6%]   7 [||||||80.6%]
         10 [||||||80.1%]
2   [||||||81.0%]   5 [||||||86.5%]   8 [||||||81.0%]
         11 [||||||80.8%]
3   [||||||81.6%]   6 [||||||81.5%]   9 [||||||81.4%]
         12 [||||||79.9%]
```
```

```
Mem[||||||||||||||||||32.1G/47.1G]          Tasks: 55, 282
                                            thr; 12 running
         Swp[|                59.2M/1.91G]  Load average:
                                            10.30 8.50
5.12
                                            Uptime: 3 days,
                                            22:26:34
```

Hit “q” to quit htop. You'll have plenty of time to get familiar with it; don't worry.

GPU resources:

> nvidia-smi should be on any system with nvidia drivers.
>
> I also like nvtop, a tool to visualize usage of nvidia devices.

Check standard system utilities

Here's some other quick system utilities I like to install: sudo apt-get install nload

```
nload -u M
```

This one gives you a terminal-based view of what your network interfaces are doing. The only thing that makes it a pain for new users is that you will need to hit left/right arrows on your keyboard till you find the actual interface that goes to the Internet. This is nice if you need to check if the software is actually working (e.g., the network is just slow) or if there

is no traffic at all (generally a sign of larger problems). The -u M flag just puts everything into megabyte (not megabit) units, which are easier to keep track of in my head.

```
free -h
```

This will print out all the memory usage on your system, in (-h) human-readable units:

```
skoonce@hermes:~$ free -h
              total        used        free      shared  buff/cache   available
Mem:           47G         29G        9.5G        6.6M        8.4G          17G
Swap:         1.9G         59M        1.9G
```

Here's a quick guide to mounting a second hard drive and keeping track of disks:

```
df -h
```

This will print out the current disk status.

If you install Kubernetes or a ton of snaps, they will add a bunch of virtual filesystems so df -h | grep dev is a good trick to keep track of your base system.

lsblk is my preferred utility to keep track of different filesystems. It prints out a tree of what drives your system has available.

iotop is another nice tool to run to keep track of disk accesses, but requires root access to install.

zfs: I have had good luck with the zfs libraries for ubuntu 18.04. zpool init, export, and status are useful commands to know.

Index

A, B

C

B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2

D

E

F

G

H

I

J, K

L

M

N, O

P, Q

R

S

T

U

V, W, X, Y, Z