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

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.