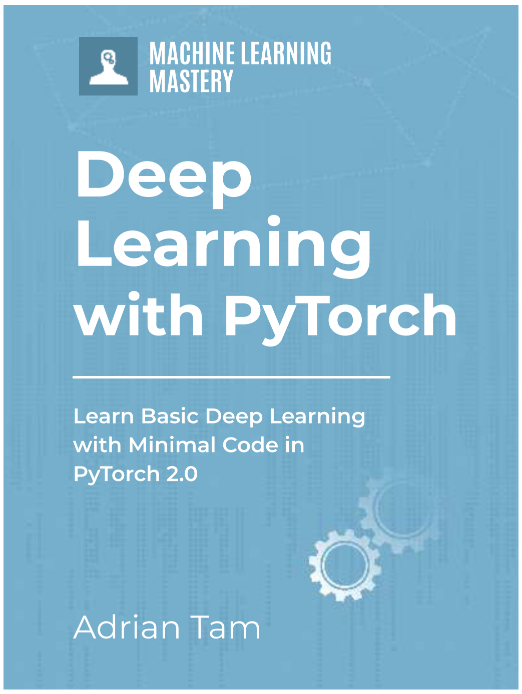
MACHINE LEARNING MASTERY

Deep Learning with PyTorch

Learn Basic Deep Learning with Minimal Code in PyTorch 2.0

Adrian Tam


MACHINE LEARNING MASTERY
Deep Learning with PyTorch
Learn Basic Deep Learning with Minimal Code in PyTorch 2.0
Adrian Tam

Credits

Founder: Jason Brownlee
Authors: Adrian Tam
Technical Reviewers: Darci Heikkinen, Amy Lam, and Devansh Sethi

Contents

Preface

No explanation is needed for why deep learning is amazing. But where to start?

Nowadays, probably the ecosystem for deep learning is based on either of the two libraries: TensorFlow and PyTorch. To learn about deep learning, it is inevitable to study the program in either frameworks. Neither is perfect and either has some edge over the other.

This book is to assemble some quick tips to get started on PyTorch. You will need some basic knowledge on what deep learning is about and some high level idea on the steps to build and use a deep learning model. Then, you can learn how to implement all these idea in PyTorch.

This is the guide you may find useful to bring you up to speed quickly in developing a deep learning model in PyTorch as well as understanding other people's code.

Adrian Tam
New York
2023

Introduction

Welcome to Deep Learning with PyTorch. This book is your guide to deep learning. You will discover the PyTorch library for deep learning and how to use it to develop and evaluate deep learning models. In this book you will discover the techniques, recipes and skills in deep learning that you can then bring to your own machine learning projects.

Deep learning does have a lot of fascinating math under the covers, but you do not need to know it to be able to pick it up as a tool and wield it on important projects and deliver real value. From the applied perspective, deep learning is quite a shallow field and a motivated developer can quickly pick it up and start making very real and impactful contributions. This is our goal for you and this book is your ticket to that outcome.

Deep Learning the Wrong Way

If you ask a deep learning practitioner how to get started with neural networks and deep learning, what do they say? They say things like

- ▷ You must have a strong foundation in linear algebra.
- ▷ You must have a deep knowledge of traditional neural network techniques.
- ▷ You really must know about probability and statistics.
- ▷ You should really have a deep knowledge of machine learning.
- ▷ You probably need to be a PhD in computer science.
- ▷ You probably need 10 years of experience as a machine learning developer.

You can see that the *common sense* advice means that it is not until after you have completed years of study and experience that you are ready to actually start developing and evaluating machine learning model for your machine learning projects.

This advice is dead wrong.

Deep Learning with Python

The approach taken with this book and with all of Machine Learning Mastery is to flip the traditional approach. If you are interested in deep learning, start by developing and evaluating deep learning models. Then if you discover you really like it or have a knack for it, later you can step deeper and deeper into the background and theory, as you need it in order to serve you in developing better and more valuable results. This book is your ticket to jumping in and making a ruckus with deep learning.

Unlike R, Python is a fully featured programming language allowing you to use the same libraries and code for model development as you can use in production. Unlike Java, Python has the SciPy stack for scientific computing and scikit-learn which is a professional grade machine learning library.

Machine Learning Mastery has published a similar book on TensorFlow and Keras, but it is impossible to ignore the existence of the PyTorch library from Facebook/Meta. These libraries are developed for use in Python, with concise API, and allows you to focus on the deep learning model that you want to develop rather than the detail on how numbers and matrices change values. This book is solely in PyTorch.

You will develop your own and perhaps your first neural network and deep learning models while working through this book. You will have the skills to bring this amazing new technology to your own projects. It is going to be a fun journey and we can't wait to start.

Book Organization

There are three kinds of chapters in this book.

▷ **Lessons**, where you learn about specific features of neural network models and how to use specific aspects of PyTorch.

▷ **Projects**, where you will pull together multiple lessons into an end-to-end project and deliver a result, providing a template for your own projects.

▷ **Recipes**, where you can copy and paste the standalone code into your own project, including all of the code presented in this book.

Lessons and Projects

Lessons are discrete and are focused on one topic, designed for you to complete in one sitting. You can take as long as you need, from 20 minutes if you are racing through, to hours if you want to experiment with the code or ideas and improve upon the presented results. Your lessons are divided into five parts:

▷ Background

▷ Multilayer perceptron models

▷ Techniques for better deep learning models

▷ Advanced models

Part 1: Background

In this part you will learn about the PyTorch library that lay the foundation for your deep learning journey. This part of the book includes the following lessons:

- ▷ Overview of Some Deep Learning Libraries
- ▷ Introduction to PyTorch
- ▷ Manipulating Tensors in PyTorch
- ▷ Using Autograd in PyTorch to Solve a Regression Problem

The lessons will introduce you to the library that you need to install and use on your workstation. You will also learn about how PyTorch can help deep learning. At the end of this part you will be ready to start developing models in PyTorch on your workstation.

Part 2: Multilayer Perceptron Models

In this part you will learn about feedforward neural networks that may be deep or not and how to expertly develop your own networks and evaluate them efficiently using PyTorch. This part of the book includes the following lessons:

- ▷ A Crash Course to Deep Learning
- ▷ Multilayer Perceptron Building Blocks in PyTorch
- ▷ Your First Neural Network in PyTorch, Step by Step
- ▷ Creating a Training Loop for Your Models
- ▷ Evaluating PyTorch Models

These important lessons are tied together with three foundation projects. These projects demonstrate how you can quickly and efficiently develop neural network models for tabular data and provide project templates that you can use on your own regression and classification machine learning problems. These projects include:

- ▷ Project: Building a Multiclass Classification Model in PyTorch
- ▷ Project: Building a Binary Classification Model in PyTorch
- ▷ Project: Building a Regression Model in PyTorch

Not only the classification and regression are major application of deep learning models, these projects are also good opportunities to learn about data preprocessing techniques for a more effective model. At the end of this part you will be ready to discover the finer points of deep learning.

Part 3: Techniques for Better Deep Learning Models

In this part you will learn about some finer points of the PyTorch library and API for practical machine learning projects and some of the more important developments in applied neural networks that you need to know in order to deliver a world-class results. This part of the book includes the following lessons:

- ▷ Save and Load Your PyTorch Models

- ▷ Using Activation Functions in Deep Learning Models
- ▷ Loss Functions in PyTorch Models
- ▷ Using Dropout Regularization in PyTorch Models
- ▷ Using Learning Rate Schedule in PyTorch Models
- ▷ Training a PyTorch Model with DataLoader and Dataset
- ▷ Use PyTorch Deep Learning Models with scikit-learn
- ▷ Optimize Hyperparameters with Grid Search
- ▷ Managing Training Process with Checkpoints and Early Stopping
- ▷ Visualizing a PyTorch Model
- ▷ Understanding Model Behavior During Training by Visualizing Metrics

At the end of this part you will know how to confidently wield PyTorch on your machine learning projects with a focus on the finer points of investigating model performance, persisting models for later use and gaining lifts in performance over baseline models.

Part 4: Advanced Models

Neural networks started their history with fully-connected multilayer perceptron models. However, the recent advance in neural networks proposed various different designs. In this part you will receive a crash course in the dominant model for computer vision machine learning problems and some natural language problems and learn how you can best exploit the capabilities of the PyTorch for your own projects. This part of the book includes the following lessons and projects:

- ▷ From MLP to CNN and RNN
- ▷ Building a Convolutional Neural Network in PyTorch
- ▷ Handwritten Digit Recognition with LeNet5 Model in PyTorch
- ▷ LSTM for Time Series Prediction in PyTorch
- ▷ Text Generation with LSTM in PyTorch

The best way to learn about this impressive type of neural network model is to apply it. The last four chapters each has a project to let you practice what you learned earlier while learning new types of network models.

After completing the lessons and projects in this part, you will have the skills and the confidence to use the templates and recipes to tackle your own deep learning projects using convolutional neural networks and recurrent neural networks.

Conclusions

The book concludes with some resources that you can use to learn more information about a specific topic or find help if you need it, as you start to develop and evaluate your own deep learning models.

Recipes

Building up a catalog of code recipes is an important part of your deep learning journey. Each time you learn about a new technique or new problem type, you should write up a short code recipe that demonstrates it. This will give you a starting point to use on your next deep learning or machine learning project.

As part of this book you will receive a catalog of deep learning recipes. This includes recipes for all of the lessons presented in this book, as well as complete code for all of the projects. You are strongly encouraged to add to and build upon this catalog of recipes as you expand your use and knowledge of deep learning in Python.

Requirements for This Book

Python, PyTorch, and NumPy

You do not need to be a Python expert, but it would be helpful if you know how to install and setup Python and NumPy. The lessons and projects assume that you have a Python environment available. This may be on your workstation or laptop, it may be in a VM or a Docker instance that you run, or it may be a server instance that you can configure in the cloud. You will be guided as to how to install the PyTorch library in Part I of the book. If you have trouble, you can follow the step-by-step tutorial in Appendix B.

Machine Learning

You do not need to be a machine learning expert, but it would be helpful if you knew how to navigate a small machine learning problem using scikit-learn. Basic concepts like cross-validation and one-hot encoding used in lessons and projects are described, but only briefly. There are resources to go into these topics in more detail at the end of the book, but some knowledge of these areas might make things easier for you.

Deep Learning

You do not need to know the math and theory of deep learning algorithms, but it would be helpful to have some basic ideas of the field. You will get a crash course in neural network terminology and models, but I will not go into much detail. Again, there will be resources for more information at the end of the book, but it might be helpful if you can start with some ideas about neural networks.

Note: All tutorials can be completed on standard workstation hardware with a CPU. GPU is not required. Some tutorials later in the book can be speed up significantly by running on the GPU and a suggestion is provided to consider using GPU hardware at the beginning of those sections. You can access GPU hardware easily and cheaply in the cloud and a step-by-step procedure is taught on how to do this in Appendix C.

Your Outcomes from Reading This Book

This book will lead you from being a developer who is interested in deep learning with Python to a developer who has the resources and capabilities to work through a new dataset end-to-end using Python and develop accurate deep learning models. Specifically, you will know:

▷ How to develop and evaluate neural network models end-to-end.

▷ How to use more advanced techniques required for developing state-of-the-art deep learning models.

▷ How to build larger models for image.

▷ How to get help with deep learning in Python.

From here you can start to dive into the specifics of the functions, techniques and algorithms used with the goal of learning how to use them better in order to deliver more accurate predictive models, more reliably in less time. There are a few ways you can read this book. You can dip into the lessons and projects as your need or interests motivate you. Alternatively, you can work through the book end-to-end and take advantage of how the lessons and projects built toward complexity and range. I recommend the latter approach.

To get the very most from this book, I recommend taking each lesson and project and build upon them. Attempt to improve the results, apply the method to a similar but different problem, and so on. Write up what you tried or learned and share it on your blog, social media or send me an email at `jason@MachineLearningMastery.com`. This book is really what you make of it and by putting in a little extra, you can quickly become a true force in applied deep learning.

What This Book Is Not

This book solves a specific problem of getting you, a developer, up to speed applying deep learning to your own machine learning projects in Python. As such, this book was not intended to be everything to everyone and it is very important to calibrate your expectations. Specifically:

▷ **This is not a deep learning textbook**. You will not be getting into the basic theory of artificial neural networks or deep learning algorithms. You are also expected to have some familiarity with machine learning basics, or be able to pick them up yourself.

▷ **This is not an algorithm book**. You will not be working through the details of how specific deep learning algorithms work. You are expected to have some basic knowledge of deep learning algorithms or be able to pick up this knowledge yourself.

▷ **This is not a Python programming book**. You will not be spending a lot of time on Python syntax and programming (e.g., basic programming tasks in Python). You are expected to already be familiar with Python or be a developer who can pick up a new C-like language relatively quickly.

You can still get a lot out of this book if you are weak in one or two of these areas, but you may struggle picking up the language or require some more explanation of the techniques.

If this is the case, see the Getting More Help chapter at the end of the book and seek out a good companion reference text.

Summary

It is a special time right now. The tools for applied deep learning have never been so good. The pace of change with neural networks and deep learning feels like it has never been so fast, spurred by the amazing results that the methods are showing in such a broad range of fields. This is the start of your journey into deep learning and I am excited for you. Take your time, have fun and I am so excited to see where you can take this amazing new technology to.

Next

Let's dive in. Next up is Part I where you will take a whirlwind tour of the foundation libraries for deep learning in Python, namely the PyTorch library that you will be using throughout this book.

8 Creating a Training Loop for Your Models

PyTorch provides a lot of building blocks for a deep learning model, but a training loop is not part of them. It is a flexibility that allows you to do whatever you want during training, but some basic structure is universal across most use cases.

In this chapter, you will see how to make a training loop that provides essential information for your model training, with the option to allow any information to be displayed. After completing this chapter, you will know:

- ▷ The basic building block of a training loop
- ▷ How to use tqdm to display training progress

Let's get started.

Overview

This chapter is in three parts; they are:

- ▷ Elements of Training a Deep Learning Model
- ▷ Collecting Statistics During Training
- ▷ Using tqdm to Report the Training Progress

8.1 Elements of Training a Deep Learning Model

As with all machine learning models, the model design specifies the algorithm to manipulate an input and produce an output. But in the model, there are parameters that you need to fine-tune to achieve that. These model parameters are also called the weights, biases, kernels, or other names depending on the particular model and layers. Training is to feed in the sample data to the model so that an optimizer can fine-tune these parameters.

When you train a model, you usually start with a dataset. Each dataset is a fairly large number of data samples. When you get a dataset, it is recommended to split it into two portions: the training set and the test set. The training set is further split into batches and used in the training loop to drive the gradient descent algorithms. The test set, however, is used as a benchmark to tell how good your model is. Usually, you do not use the training set as a metric but take the test set, which is not seen by the gradient descent algorithm, so you can tell if your model fits well to the unseen data.

Overfitting is when the model fits too well to the training set (i.e., at very high accuracy) but performs significantly worse in the test set. Underfitting is when the model cannot even fit well to the training set. Naturally, you don't want to see either on a good model.

Training of a neural network model is in epochs. Usually, one epoch means you run through the entire training set once, although you only feed one batch at a time. It is also customary to do some housekeeping tasks at the end of each epoch, such as benchmarking the partially trained model with the test set, checkpointing the model, deciding if you want to stop the training early, and collecting training statistics, and so on.

In each epoch, you feed data samples into the model in batches and run a gradient descent algorithm. This is one step in the training loop because you run the model in one forward pass (i.e., providing input and capturing output), and one backward pass (evaluating the loss metric from the output and deriving the gradient of each parameter all the way back to the input layer). The backward pass computes the gradient using automatic differentiation. Then, this gradient is used by the gradient descent algorithm to adjust the model parameters. There are multiple steps in one epoch.

Reusing the examples in Chapter 7, you can download the dataset[1] and split the dataset into two as follows:

```python
import numpy as np
import torch

# load the dataset
dataset = np.loadtxt('pima-indians-diabetes.csv', delimiter=',')
X = dataset[:,0:8]
y = dataset[:,8]
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)

# split the dataset into training and test sets
Xtrain = X[:700]
ytrain = y[:700]
Xtest = X[700:]
ytest = y[700:]
```

Listing 8.1: Loading data from a CSV file

This dataset is small — only 768 samples. Here, it takes the first 700 as the training set and the rest as the test set.

It is not the focus of this chapter, but you can reuse the model, the loss function, and the optimizer from a previous chapter:

```python
import torch.nn as nn
import torch.optim as optim

```

[1]https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv

```
model = nn.Sequential(
    nn.Linear(8, 12),
    nn.ReLU(),
    nn.Linear(12, 8),
    nn.ReLU(),
    nn.Linear(8, 1),
    nn.Sigmoid()
)
print(model)

# loss function and optimizer
loss_fn = nn.BCELoss()  # binary cross entropy
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Listing 8.2: Setting up a model, the loss function, and the optimizer

With the data and the model, this is the minimal training loop, with the forward and backward pass in each step:

```
n_epochs = 50    # number of epochs to run
batch_size = 10  # size of each batch
batches_per_epoch = len(Xtrain) // batch_size

for epoch in range(n_epochs):
    for i in range(batches_per_epoch):
        start = i * batch_size
        # take a batch
        Xbatch = Xtrain[start:start+batch_size]
        ybatch = ytrain[start:start+batch_size]
        # forward pass
        y_pred = model(Xbatch)
        loss = loss_fn(y_pred, ybatch)
        # backward pass
        optimizer.zero_grad()
        loss.backward()
        # update weights
        optimizer.step()
```

Listing 8.3: Training the model in a loop

In the inner for-loop, you take each batch in the dataset and evaluate the loss. The `loss` is a PyTorch tensor that remembers how it comes up with its value. Then you zero out all gradients that the optimizer manages and call `loss.backward()` to run the backpropagation algorithm. The result sets up the gradients of all the tensors that the tensor `loss` depends on directly and indirectly. Afterward, upon calling `step()`, the optimizer will check each parameter that it manages and update them.

After everything is done, you can run the model with the test set to evaluate its performance. The evaluation can be based on a different function than the loss function. For example, this classification problem uses accuracy:

```
...

# evaluate trained model with test set
with torch.no_grad():
    y_pred = model(X)
accuracy = (y_pred.round() == y).float().mean()
print("Accuracy {:.2f}".format(accuracy * 100))
```

Listing 8.4: Evaluating the trained model

Putting everything together, this is the complete code:

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim

# load the dataset
dataset = np.loadtxt('pima-indians-diabetes.csv', delimiter=',')
X = dataset[:,0:8]
y = dataset[:,8]
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)

# split the dataset into training and test sets
Xtrain = X[:700]
ytrain = y[:700]
Xtest = X[700:]
ytest = y[700:]

model = nn.Sequential(
    nn.Linear(8, 12),
    nn.ReLU(),
    nn.Linear(12, 8),
    nn.ReLU(),
    nn.Linear(8, 1),
    nn.Sigmoid()
)
print(model)

# loss function and optimizer
loss_fn = nn.BCELoss()  # binary cross entropy
optimizer = optim.Adam(model.parameters(), lr=0.001)

n_epochs = 50    # number of epochs to run
batch_size = 10  # size of each batch
batches_per_epoch = len(Xtrain) // batch_size

for epoch in range(n_epochs):
    for i in range(batches_per_epoch):
        start = i * batch_size
        # take a batch
        Xbatch = Xtrain[start:start+batch_size]
        ybatch = ytrain[start:start+batch_size]
        # forward pass
        y_pred = model(Xbatch)
        loss = loss_fn(y_pred, ybatch)
        # backward pass
        optimizer.zero_grad()
        loss.backward()
        # update weights
        optimizer.step()

# evaluate trained model with test set
with torch.no_grad():
    y_pred = model(X)
accuracy = (y_pred.round() == y).float().mean()
print("Accuracy {:.2f}".format(accuracy * 100))
```

Listing 8.5: Creating, training, and evaluating a model

8.2 Collecting Statistics During Training

The training loop above should work well with small models that can finish training in a few seconds. But for a larger model or a larger dataset, you will find that it takes significantly longer to train. While you're waiting for the training to complete, you may want to see how it's going as you may want to interrupt the training if any mistake is made.

Usually, during training, you would like to see the following:

- ▷ In each step, you would like to know the loss metrics, and you are expecting the loss to go down
- ▷ In each step, you would like to know other metrics, such as accuracy on the training set, that are of interest but not involved in the gradient descent
- ▷ At the end of each epoch, you would like to evaluate the partially-trained model with the test set and report the evaluation metric
- ▷ At the end of the training, you would like to be above to visualize the above metrics

These all are possible, but you need to add more code into the training loop, as follows:

```
n_epochs = 50    # number of epochs to run
batch_size = 10  # size of each batch
batches_per_epoch = len(Xtrain) // batch_size

# collect statistics
train_loss = []
train_acc = []
test_acc = []

for epoch in range(n_epochs):
    for i in range(batches_per_epoch):
        start = i * batch_size
        # take a batch
        Xbatch = Xtrain[start:start+batch_size]
        ybatch = ytrain[start:start+batch_size]
        # forward pass
        y_pred = model(Xbatch)
        loss = loss_fn(y_pred, ybatch)
        acc = (y_pred.round() == ybatch).float().mean()
        # store metrics
        train_loss.append(float(loss))
        train_acc.append(float(acc))
        # backward pass
        optimizer.zero_grad()
        loss.backward()
        # update weights
        optimizer.step()
        # print progress
        print(f"epoch {epoch} step {i} loss {loss} accuracy {acc}")
    # evaluate model at end of epoch
    y_pred = model(Xtest)
    acc = (y_pred.round() == ytest).float().mean()
    test_acc.append(float(acc))
    print(f"End of {epoch}, accuracy {acc}")
```

Listing 8.6: The training loop expanded to collect more statistics

As you collect the loss and accuracy in the list, you can plot them using matplotlib. But be careful that you collected training set statistics at each step, whereas the test set accuracy only at the end of the epoch. Thus you would like to show the *average accuracy* from the training loop in each epoch, so they are comparable to each other.

```
import matplotlib.pyplot as plt

# Plot the loss metrics, set the y-axis to start from 0
plt.plot(train_loss)
plt.xlabel("steps")
plt.ylabel("loss")
plt.ylim(0)
plt.show()

# plot the accuracy metrics
avg_train_acc = []
for i in range(n_epochs):
    start = i * batch_size
    average = sum(train_acc[start:start+batches_per_epoch]) / batches_per_epoch
    avg_train_acc.append(average)

plt.plot(avg_train_acc, label="train")
plt.plot(test_acc, label="test")
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0)
plt.show()
```

Listing 8.7: Visualizing accuracy metric during the training of a model

Putting everything together, below is the complete code:

```
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim

# load the dataset
dataset = np.loadtxt('pima-indians-diabetes.csv', delimiter=',')
# split into input (X) and output (y) variables
X = dataset[:,0:8]
y = dataset[:,8]
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)

# split the dataset into training and test sets
Xtrain = X[:700]
ytrain = y[:700]
Xtest = X[700:]
ytest = y[700:]

model = nn.Sequential(
    nn.Linear(8, 12),
    nn.ReLU(),
    nn.Linear(12, 8),
    nn.ReLU(),
    nn.Linear(8, 1),
    nn.Sigmoid()
)
print(model)

# loss function and optimizer
loss_fn = nn.BCELoss()  # binary cross entropy
optimizer = optim.Adam(model.parameters(), lr=0.0001)

n_epochs = 50    # number of epochs to run
batch_size = 10  # size of each batch
batches_per_epoch = len(Xtrain) // batch_size

# collect statistics
train_loss = []
train_acc = []
test_acc = []

for epoch in range(n_epochs):
    for i in range(batches_per_epoch):
        # take a batch
        start = i * batch_size
        Xbatch = Xtrain[start:start+batch_size]
        ybatch = ytrain[start:start+batch_size]
        # forward pass
        y_pred = model(Xbatch)
        loss = loss_fn(y_pred, ybatch)
        acc = (y_pred.round() == ybatch).float().mean()
        # store metrics
        train_loss.append(float(loss))
        train_acc.append(float(acc))
        # backward pass
        optimizer.zero_grad()
        loss.backward()
        # update weights
        optimizer.step()
        # print progress
        print(f"epoch {epoch} step {i} loss {loss} accuracy {acc}")
    # evaluate model at end of epoch
    y_pred = model(Xtest)
    acc = (y_pred.round() == ytest).float().mean()
    test_acc.append(float(acc))
    print(f"End of {epoch}, accuracy {acc}")

# Plot the loss metrics
plt.plot(train_loss)
plt.xlabel("steps")
plt.ylabel("loss")
plt.ylim(0)
plt.show()

# plot the accuracy metrics
avg_train_acc = []
for i in range(n_epochs):
    start = i * batch_size
    average = sum(train_acc[start:start+batches_per_epoch]) / batches_per_epoch
    avg_train_acc.append(average)

plt.plot(avg_train_acc, label="train")
plt.plot(test_acc, label="test")
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0)
plt.show()
```

Listing 8.8: Complete code to build and train a model with visualized accuracy metric

The story does not end here. Indeed, you can add more code to the training loop, especially in dealing with a more complex model. One example is checkpointing. You may want to save your model (e.g., using pickle) so that, if for any reason, your program stops, you can restart the training loop from the middle. Another example is early stopping, which lets you monitor the accuracy you obtained with the test set at the end of each epoch and interrupt the training if you don't see the model improving for a while. This is because you probably can't go further, given the design of the model, and you do not want to overfit. Chapter 21 will cover more on this topic.

8.3 Using `tqdm` to Report the Training Progress

If you run the above code, you will find that there are a lot of lines printed on the screen while the training loop is running. Your screen may be cluttered. And you may also want to see an animated progress bar to better tell you how far you are in the training progress. The library `tqdm` is the popular tool for creating the progress bar. Converting the above code to use `tqdm` cannot be easier:

```python
for epoch in range(n_epochs):
    with tqdm.trange(batches_per_epoch, unit="batch", mininterval=0) as bar:
        bar.set_description(f"Epoch {epoch}")
        for i in bar:
            # take a batch
            start = i * batch_size
            Xbatch = Xtrain[start:start+batch_size]
            ybatch = ytrain[start:start+batch_size]
            # forward pass
            y_pred = model(Xbatch)
            loss = loss_fn(y_pred, ybatch)
            acc = (y_pred.round() == ybatch).float().mean()
            # store metrics
            train_loss.append(float(loss))
            train_acc.append(float(acc))
            # backward pass
            optimizer.zero_grad()
            loss.backward()
            # update weights
            optimizer.step()
            # print progress
            bar.set_postfix(
                loss=float(loss),
                acc=f"{float(acc)*100:.2f}%"
            )
    # evaluate model at end of epoch
    y_pred = model(Xtest)
    acc = (y_pred.round() == ytest).float().mean()
    test_acc.append(float(acc))
    print(f"End of {epoch}, accuracy {acc}")
```

Listing 8.9: A training loop with `tqdm` progress bar

The usage of `tqdm` creates an iterator using `trange()` just like Python's `range()` function, and you can read the number in a loop. You can access the progress bar by updating its description or "postfix" data, but you have to do that before it exhausts its content. The `set_postfix()` function is powerful as it can show you anything.

In fact, there is a `tqdm()` function besides `trange()` that iterates over an existing list. You may find it easier to use, and you can rewrite the above loop as follows:

```python
starts = [i*batch_size for i in range(batches_per_epoch)]

for epoch in range(n_epochs):
    with tqdm.tqdm(starts, unit="batch", mininterval=0) as bar:
        bar.set_description(f"Epoch {epoch}")
        for start in bar:
            # take a batch
            Xbatch = Xtrain[start:start+batch_size]
            ybatch = ytrain[start:start+batch_size]
            # forward pass
            y_pred = model(Xbatch)
            loss = loss_fn(y_pred, ybatch)
            acc = (y_pred.round() == ybatch).float().mean()
            # store metrics
            train_loss.append(float(loss))
            train_acc.append(float(acc))
            # backward pass
            optimizer.zero_grad()
            loss.backward()
            # update weights
            optimizer.step()
            # print progress
            bar.set_postfix(
                loss=float(loss),
                acc=f"{float(acc)*100:.2f}%"
            )
    # evaluate model at end of epoch
    y_pred = model(Xtest)
    acc = (y_pred.round() == ytest).float().mean()
    test_acc.append(float(acc))
    print(f"End of {epoch}, accuracy {acc}")
```

Listing 8.10: Using tqdm to iterate over a list

The following is the complete code (without the matplotlib plotting):

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import tqdm

# load the dataset
dataset = np.loadtxt('pima-indians-diabetes.csv', delimiter=',')
# split into input (X) and output (y) variables
X = dataset[:,0:8]
y = dataset[:,8]
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)

# split the dataset into training and test sets
Xtrain = X[:700]
ytrain = y[:700]
Xtest = X[700:]
ytest = y[700:]

model = nn.Sequential(
    nn.Linear(8, 12),
    nn.ReLU(),
    nn.Linear(12, 8),
    nn.ReLU(),
    nn.Linear(8, 1),
    nn.Sigmoid()
)
print(model)

# loss function and optimizer
loss_fn = nn.BCELoss()  # binary cross entropy
optimizer = optim.Adam(model.parameters(), lr=0.0001)

n_epochs = 50    # number of epochs to run
batch_size = 10  # size of each batch
batches_per_epoch = len(Xtrain) // batch_size

# collect statistics
train_loss = []
train_acc = []
test_acc = []

for epoch in range(n_epochs):
    with tqdm.trange(batches_per_epoch, unit="batch", mininterval=0) as bar:
        bar.set_description(f"Epoch {epoch}")
        for i in bar:
            # take a batch
            start = i * batch_size
            Xbatch = Xtrain[start:start+batch_size]
            ybatch = ytrain[start:start+batch_size]
            # forward pass
            y_pred = model(Xbatch)
            loss = loss_fn(y_pred, ybatch)
            acc = (y_pred.round() == ybatch).float().mean()
            # store metrics
            train_loss.append(float(loss))
            train_acc.append(float(acc))
            # backward pass
            optimizer.zero_grad()
            loss.backward()
            # update weights
            optimizer.step()
            # print progress
            bar.set_postfix(
                loss=float(loss),
                acc=f"{float(acc)*100:.2f}%"
            )
    # evaluate model at end of epoch
    y_pred = model(Xtest)
    acc = (y_pred.round() == ytest).float().mean()
    test_acc.append(float(acc))
    print(f"End of {epoch}, accuracy {acc}")
```

Listing 8.11: Complete code to train a model with progress printed using `tqdm`

8.4 Further Readings

This section provides more resources on the topic if you are looking to go deeper.

Software

tqdm.
https://github.com/tqdm/tqdm

Articles

torch.optim. PyTorch API.
https://pytorch.org/docs/stable/optim.html
Optimizing Model Parameters. PyTorch tutorials.
https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html

8.5 Summary

In this chapter, you looked in detail at how to properly set up a training loop for a PyTorch model. In particular, you saw:

- ▷ What are the elements needed to implement in a training loop
- ▷ How a training loop connects the training data to the gradient descent optimizer
- ▷ How to collect information in the training loop and display them

In the next chapter, you will learn how to evaluate and compare deep learning models.

12 Project: Building a Regression Model in PyTorch

Regression problems are usually easier to solve compared to classification problems. Many models can handle regression problems well and deep learning is one of them. In this chapter, you will discover how to use PyTorch to develop and evaluate neural network models for regression problems.

After completing this chapter, you will know:

- ▷ How to load data from scikit-learn and adapt it for PyTorch models
- ▷ How to create a neural network for regression problem using PyTorch
- ▷ How to improve model performance with data preparation techniques

Let's get started.

12.1 Description of the Dataset

The dataset you will use in this chapter is the California housing dataset.

This is a dataset that describes the median house value for California districts. Each data sample is a census block group. The target variable is the median house value in multiples of USD 100,000 in 1990 and there are 8 input features, each describing something about the house. They are, namely,

- ▷ MedInc: median income in block group
- ▷ HouseAge: median house age in block group
- ▷ AveRooms: average number of rooms per household
- ▷ AveBedrms: average number of bedrooms per household
- ▷ Population: block group population
- ▷ AveOccup: average number of household members
- ▷ Latitude: block group centroid latitude
- ▷ Longitude: block group centroid longitude

This data is special because the input data is in vastly different scale. For example, the number of rooms per house is usually small but the population per block group is usually large. Moreover, most features should be positive but the longitude must be negative (because that's about California). Handling such diversity of data is a challenge to some machine learning models.

You can get the dataset from scikit-learn, which in turn, is downloaded from the Internet at realtime:

```
from sklearn.datasets import fetch_california_housing

data = fetch_california_housing()
print(data.feature_names)

X, y = data.data, data.target
```

Listing 12.1: Loading the dataset using scikit-learn

12.2 Building a Model and Train

This is a regression problem. Unlike classification problems, the output variable is a continuous value. In case of neural networks, you usually use linear activation at the output layer (i.e., no activation) such that the output range theoretically can be anything from negative infinty to positive infinity.

Also for regression problems, you should never expect the model to predict the values perfectly. Therefore, you should care about how close the prediction is to the actual value. The loss metric that you can use for this is the mean square error (MSE) or mean absolute error (MAE). But you may also interested in the root mean squared error (RMSE) because that's a metric in the same unit as your output variable.

Let's try the traditional design of a neural network, namely, the pyramid structure. A pyramid structure is to have the number of neurons in each layer decreasing as the network progresses to the output. The number of input features is fixed, but you set a large number of neurons on the first hidden layer and gradually reduce the number in the subsequent layers. Because you have only one target in this dataset, the final layer should output only one value.

One design is as follows:

```
import torch.nn as nn

# Define the model
model = nn.Sequential(
    nn.Linear(8, 24),
    nn.ReLU(),
    nn.Linear(24, 12),
    nn.ReLU(),
    nn.Linear(12, 6),
    nn.ReLU(),
    nn.Linear(6, 1)
)
```

Listing 12.2: A model built in PyTorch

To train this network, you need to define a loss function. MSE is a reasonable choice. You also need an optimizer, such as Adam.

```python
import torch.nn as nn
import torch.optim as optim

# loss function and optimizer
loss_fn = nn.MSELoss()  # mean square error
optimizer = optim.Adam(model.parameters(), lr=0.0001)
```

Listing 12.3: Creating the loss function and optimizer

To train this model, you can use your usual training loop. In order to obtain an evaluation score so you are confident that the model works, you need to split the data into training and test sets. You may also want to avoid overfitting by keeping track on the test set MSE. The following is the training loop with the train-test split:

```python
import copy
import numpy as np
import torch
import tqdm
from sklearn.model_selection import train_test_split

# train-test split of the dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, shuffle=True)
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32).reshape(-1, 1)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.float32).reshape(-1, 1)

# training parameters
n_epochs = 100   # number of epochs to run
batch_size = 10  # size of each batch
batch_start = torch.arange(0, len(X_train), batch_size)

# Hold the best model
best_mse = np.inf   # init to infinity
best_weights = None
history = []

# training loop
for epoch in range(n_epochs):
    model.train()
    with tqdm.tqdm(batch_start, unit="batch", mininterval=0, disable=True) as bar:
        bar.set_description(f"Epoch {epoch}")
        for start in bar:
            # take a batch
            X_batch = X_train[start:start+batch_size]
            y_batch = y_train[start:start+batch_size]
            # forward pass
            y_pred = model(X_batch)
            loss = loss_fn(y_pred, y_batch)
            # backward pass
            optimizer.zero_grad()
            loss.backward()
            # update weights
            optimizer.step()
            # print progress
            bar.set_postfix(mse=float(loss))
    # evaluate accuracy at end of each epoch
    model.eval()
    y_pred = model(X_test)
    mse = loss_fn(y_pred, y_test)
    mse = float(mse)
    history.append(mse)
    if mse < best_mse:
        best_mse = mse
        best_weights = copy.deepcopy(model.state_dict())

# restore model and return best accuracy
model.load_state_dict(best_weights)
```

Listing 12.4: Training loop for the regression model

In the training loop, `tqdm` is used to set up a progress bar and in each iteration step, MSE is calculated and reported. You can see how the MSE changed by setting the `tqdm` parameter `disable` above to `False`.

Note that in the training loop, each epoch is to run the forward and backward steps with the training set a few times to optimize the model weights, and at the end of the epoch, the model is evaluated using the test set. It is the MSE from the test set that is remembered in the list `history`. It is also the metric to evaluate a model, which the best one is stored in the variable `best_weights`.

After you run this, you will have the best model restored and the best MSE stored in the variable `best_mse`. Note that the mean square error is the average of the square of the difference between the predicted value and the actual value. The square root of it, RMSE, can be regarded as the average difference and it is numerically more useful.

In below, you can show the MSE and RMSE, and plot the history of MSE. It should be decreasing with the epochs.

```
print("MSE: %.2f" % best_mse)
print("RMSE: %.2f" % np.sqrt(best_mse))
plt.plot(history)
plt.show()
```

Listing 12.5: Plotting the MSE history

This model produced:

```
MSE: 0.47
RMSE: 0.68
```

Output 12.1: The best MSE and RMSE achieved

The MSE graph would like the following.

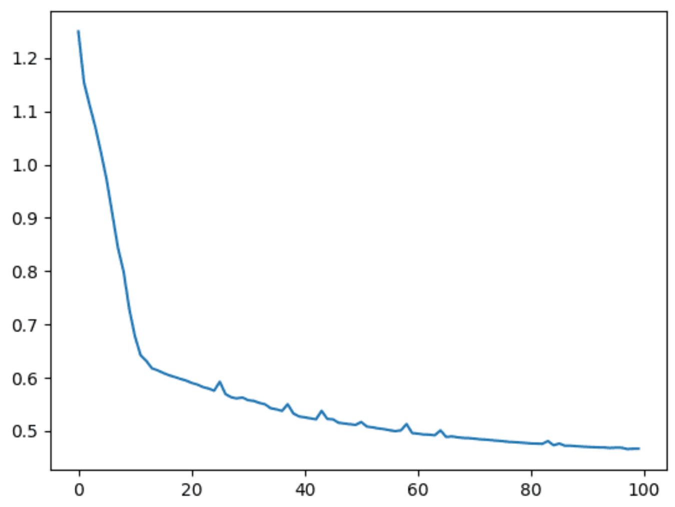


Figure 12.1: The plot of the MSE history

Putting everything together, the following is the complete code.

```
import copy

import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import tqdm
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_california_housing

# Read data
data = fetch_california_housing()
X, y = data.data, data.target

# train-test split for model evaluation
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, shuffle=True)

# Convert to 2D PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32).reshape(-1, 1)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.float32).reshape(-1, 1)

# Define the model
model = nn.Sequential(
    nn.Linear(8, 24),
    nn.ReLU(),
    nn.Linear(24, 12),
    nn.ReLU(),
    nn.Linear(12, 6),
    nn.ReLU(),
    nn.Linear(6, 1)
)

# loss function and optimizer
loss_fn = nn.MSELoss()  # mean square error
optimizer = optim.Adam(model.parameters(), lr=0.0001)

n_epochs = 100   # number of epochs to run
batch_size = 10  # size of each batch
batch_start = torch.arange(0, len(X_train), batch_size)

# Hold the best model
best_mse = np.inf   # init to infinity
best_weights = None
history = []

for epoch in range(n_epochs):
    model.train()
    with tqdm.tqdm(batch_start, unit="batch", mininterval=0, disable=True) as bar:
        bar.set_description(f"Epoch {epoch}")
        for start in bar:
            # take a batch
            X_batch = X_train[start:start+batch_size]
            y_batch = y_train[start:start+batch_size]
            # forward pass
            y_pred = model(X_batch)
            loss = loss_fn(y_pred, y_batch)
            # backward pass
            optimizer.zero_grad()
            loss.backward()
            # update weights
            optimizer.step()
            # print progress
            bar.set_postfix(mse=float(loss))
    # evaluate accuracy at end of each epoch
    model.eval()
    y_pred = model(X_test)
    mse = loss_fn(y_pred, y_test)
    mse = float(mse)
    history.append(mse)
    if mse < best_mse:
        best_mse = mse
        best_weights = copy.deepcopy(model.state_dict())

# restore model and return best accuracy
model.load_state_dict(best_weights)
print("MSE: %.2f" % best_mse)
print("RMSE: %.2f" % np.sqrt(best_mse))
plt.plot(history)
plt.show()
```

Listing 12.6: Solving the regression problem with a PyTorch model

12.3 Improving the Model with Preprocessing

In the above, you see the RMSE is 0.68. Indeed, it is easy to improve the RMSE by polishing the data before training. The problem of this dataset is the diversity of the features: Some are with a narrow range and some are wide. And some are small but positive while some are very negative. This indeed is not very nice to most of the machine learning model.

One way to improve this is to apply a *standard scaler*. It is to convert each feature into their standard score. In other words, for each feature x, you replace it with

$$z = \frac{x - \bar{x}}{\sigma_x}$$

Where $\bar{x}$ is the mean of x and σ_x is the standard deviation. This way, every transformed feature is centered around 0 and in a narrow range that around 70% of the samples are between -1 to $+1$. This can help the machine learning model to converge.

You can apply the standard scaler from scikit-learn. The following is how you should modify the data preparation part of the above code:

```python
import torch
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import StandardScaler

# Read data
data = fetch_california_housing()
X, y = data.data, data.target

# train-test split for model evaluation
X_train_raw, X_test_raw, y_train, y_test = train_test_split(X, y, train_size=0.7,
                                                             shuffle=True)

# Standardizing data
scaler = StandardScaler()
scaler.fit(X_train_raw)
X_train = scaler.transform(X_train_raw)
X_test = scaler.transform(X_test_raw)

# Convert to 2D PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32).reshape(-1, 1)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.float32).reshape(-1, 1)
```

Listing 12.7: Scaling the dataset as a preprocessing step

Note that standard scaler is applied after train-test split. The `StandardScaler` above is fitted on the training set but applied on both the training and test set. You must not fit the standard scaler to all data because nothing from the test set should be hinted to the model. Otherwise you are introducing *data leakage*.

Other than that, virtually nothing shall be changed: You still have 8 features (only they are not the same in value). You still use the same training loop. If you train the model with the scaled data, you should see the RMSE improved, e.g.,

```
MSE: 0.29
RMSE: 0.54
```

Output 12.2: The MSE improved by preprocessing data

While the MSE history is in a similar falling shape, the y-axis shows it is indeed better after scaling:

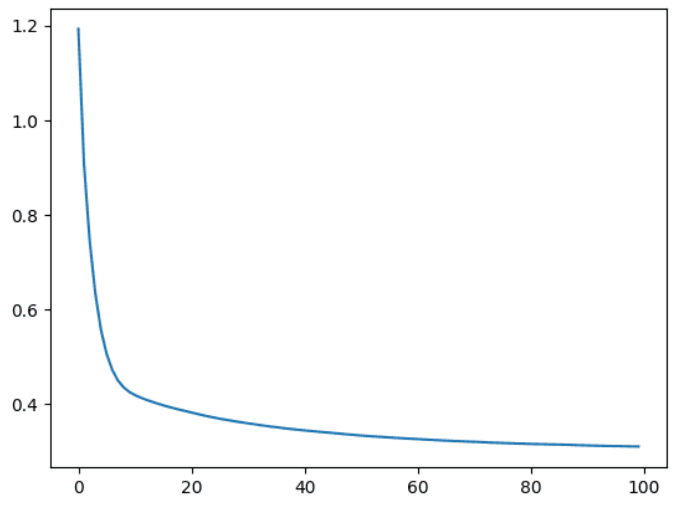


Figure 12.2: The MSE history

However, you need to be careful at the end: When you use the trained model and apply to new data, you should apply the scaler to the input data before feed into the model. That is, inference should be done as follows:

```python
model.eval()
with torch.no_grad():
    # Test out inference with 5 samples from the original test set
    for i in range(5):
        X_sample = X_test_raw[i: i+1]
        X_sample = scaler.transform(X_sample)
        X_sample = torch.tensor(X_sample, dtype=torch.float32)
        y_pred = model(X_sample)
        print(f"{X_test_raw[i]} -> {y_pred[0].numpy()} (expected {y_test[i].numpy()})")
```

Listing 12.8: Using a model that expects scaled dataset

The following is the complete code:

```python
import copy

import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import tqdm
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import StandardScaler

# Read data
data = fetch_california_housing()
X, y = data.data, data.target

# train-test split for model evaluation
X_train_raw, X_test_raw, y_train, y_test = train_test_split(X, y, train_size=0.7,
                                                            shuffle=True)

# Standardizing data
scaler = StandardScaler()
scaler.fit(X_train_raw)
X_train = scaler.transform(X_train_raw)
X_test = scaler.transform(X_test_raw)

# Convert to 2D PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32).reshape(-1, 1)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.float32).reshape(-1, 1)

# Define the model
model = nn.Sequential(
    nn.Linear(8, 24),
    nn.ReLU(),
    nn.Linear(24, 12),
    nn.ReLU(),
    nn.Linear(12, 6),
    nn.ReLU(),
    nn.Linear(6, 1)
)

# loss function and optimizer
loss_fn = nn.MSELoss()  # mean square error
optimizer = optim.Adam(model.parameters(), lr=0.0001)

n_epochs = 100   # number of epochs to run
batch_size = 10  # size of each batch
batch_start = torch.arange(0, len(X_train), batch_size)

# Hold the best model
best_mse = np.inf   # init to infinity
best_weights = None
```

```python
history = []

for epoch in range(n_epochs):
    model.train()
    with tqdm.tqdm(batch_start, unit="batch", mininterval=0, disable=True) as bar:
        bar.set_description(f"Epoch {epoch}")
        for start in bar:
            # take a batch
            X_batch = X_train[start:start+batch_size]
            y_batch = y_train[start:start+batch_size]
            # forward pass
            y_pred = model(X_batch)
            loss = loss_fn(y_pred, y_batch)
            # backward pass
            optimizer.zero_grad()
            loss.backward()
            # update weights
            optimizer.step()
            # print progress
            bar.set_postfix(mse=float(loss))
    # evaluate accuracy at end of each epoch
    model.eval()
    y_pred = model(X_test)
    mse = loss_fn(y_pred, y_test)
    mse = float(mse)
    history.append(mse)
    if mse < best_mse:
        best_mse = mse
        best_weights = copy.deepcopy(model.state_dict())

# restore model and return best accuracy
model.load_state_dict(best_weights)
print("MSE: %.2f" % best_mse)
print("RMSE: %.2f" % np.sqrt(best_mse))
plt.plot(history)
plt.show()

model.eval()
with torch.no_grad():
    # Test out inference with 5 samples
    for i in range(5):
        X_sample = X_test_raw[i: i+1]
        X_sample = scaler.transform(X_sample)
        X_sample = torch.tensor(X_sample, dtype=torch.float32)
        y_pred = model(X_sample)
        print(f"{X_test_raw[i]} -> {y_pred[0].numpy()} (expected {y_test[i].numpy()})")
```

Listing 12.9: A regression model with the input data scaled

Of course, there is still room to improve the model. One way is to present the target in log scale or, equivalently, use mean absolute percentage error (MAPE) as the loss function. This is because the target variable is the value of houses and it is in a wide range. For the same error magnitude, it is more significant in percentage for low-valued houses. It is your exercise to modify the above code to produce a better prediction.

12.4 Further Readings

This section provides more resources on the topic if you are looking to go deeper.

Articles

California housing dataset. scikit-learn documentation.
https://scikit-learn.org/stable/datasets/real_world.html#california-housing-dataset

Train test split. scikit-learn documentation.
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

Standard scaler. scikit-learn documentation.
https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

MSELoss. PyTorch API.
https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html

12.5 Summary

In this chapter, you discovered the use of PyTorch to build a regression model.

You learned how you can work through a regression problem step-by-step with PyTorch, specifically:

- ▷ How to load and prepare data for use in PyTorch
- ▷ How to create neural network models and choose a loss function for regression
- ▷ How to improve model accuracy by applying standard scaler

Starting from next chapter, you will learn various techniques to fine tune a deep learning model.

18 Training a PyTorch Model with DataLoader and Dataset

When you build and train a PyTorch deep learning model, you can provide the training data in several different ways. Ultimately, a PyTorch model works like a function that takes a PyTorch tensor and returns you another tensor. You have a lot of freedom in how to get the input tensors. Probably the easiest is to prepare a large tensor of the entire dataset and extract a small batch from it in each training step. But you will see that using the `DataLoader` can save you a few lines of code in dealing with data.

In this chapter, you will see how you can use the Data and DataLoader in PyTorch. After finishing this chapter, you will learn:

- ▷ How to create and use DataLoader to train your PyTorch model
- ▷ How to use Data class to generate data on the fly

Let's get started.

Overview

This chapter is divided into three parts; they are:

- ▷ What is `DataLoader`?
- ▷ Using `DataLoader` in a Training Loop

18.1 What is DataLoader?

To train a deep learning model, you need data. Usually data is available as a dataset. In a dataset, there are a lot of data sample or instances. You can ask the model to take one sample at a time but usually you would let the model to process one batch of several samples. You may create a batch by extracting a slice from the dataset, using the slicing syntax on the tensor. For a better quality of training, you may also want to shuffle the entire dataset on each epoch so no two batch would be the same in the entire training loop. Sometimes, you may introduce *data augmentation* to manually introduce more variance to the data. This is common for image-related tasks, which you can randomly tilt or zoom the image a bit to generate a lot of data sample from a few images.

You can imagine there can be a lot of code to write to do all these. But it is much easier with the `DataLoader`.

The following is an example of how to create a `DataLoader` and take a batch from it. In this example, the sonar dataset[1] is used and ultimately, it is converted into PyTorch tensors and passed on to `DataLoader`:

```
import pandas as pd
import torch
from torch.utils.data import DataLoader
from sklearn.preprocessing import LabelEncoder

# Read data, convert to NumPy arrays
data = pd.read_csv("sonar.csv", header=None)
X = data.iloc[:, 0:60].values
y = data.iloc[:, 60].values

# encode class values as integers
encoder = LabelEncoder()
encoder.fit(y)
y = encoder.transform(y)

# convert into PyTorch tensors
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)

# create DataLoader, then take one batch
loader = DataLoader(list(zip(X,y)), shuffle=True, batch_size=16)
for X_batch, y_batch in loader:
    print(X_batch, y_batch)
    break
```

Listing 18.1: Loading the sonar dataset and convert to PyTorch tensors

You can see from the output of above that `X_batch` and `y_batch` are PyTorch tensors. The `loader` is an instance of `DataLoader` class which can work like an iterable. Each time you read from it, you get a batch of features and targets from the original dataset.

When you create a `DataLoader` instance, you need to provide a list of sample pairs. Each sample pair is one data sample of feature and the corresponding target. A list is required because `DataLoader` expect to use `len()` to find the total size of the dataset and using array index to retrieve a particular sample. The batch size is a parameter to `DataLoader` so it knows how to create a batch from the entire dataset. You should almost always use `shuffle=True` so every time you load the data, the samples are shuffled. It is useful for training because in each epoch, you are going to read every batch once. When you proceed from one epoch to another, as `DataLoader` knows you depleted all the batches, it will re-shuffle so you get a new combination of samples.

[1] http://archive.ics.uci.edu/ml/datasets/connectionist+bench+(sonar,+mines+vs.+rocks)

18.2 Using DataLoader in a Training Loop

The following is an example to make use of `DataLoader` in a training loop:

```python
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split

# train-test split for evaluation of the model
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, shuffle=True)

# setup DataLoader for training set
loader = DataLoader(list(zip(X_train, y_train)), shuffle=True, batch_size=16)

# create model
model = nn.Sequential(
    nn.Linear(60, 60),
    nn.ReLU(),
    nn.Linear(60, 30),
    nn.ReLU(),
    nn.Linear(30, 1),
    nn.Sigmoid()
)

# Train the model
n_epochs = 200
loss_fn = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)
model.train()
for epoch in range(n_epochs):
    for X_batch, y_batch in loader:
        y_pred = model(X_batch)
        loss = loss_fn(y_pred, y_batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

# evaluate accuracy after training
model.eval()
y_pred = model(X_test)
acc = (y_pred.round() == y_test).float().mean()
acc = float(acc)
print("Model accuracy: %.2f%%" % (acc*100))
```

Listing 18.2: Training a model with `DataLoader`

You can see that once you created the `DataLoader` instance, the training loop can only be easier. In the above, only the training set is packaged with a `DataLoader` because you need to loop through it in batches. You can also create a `DataLoader` for the test set and use it for model evaluation, but since the accuracy is computed over the entire test set rather than in a batch, the benefit of `DataLoader` is not significant.

Putting everything together, below is the complete code.

```python
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader
from sklearn.preprocessing import LabelEncoder

# Read data, convert to NumPy arrays
data = pd.read_csv("sonar.csv", header=None)
X = data.iloc[:, 0:60].values
y = data.iloc[:, 60].values

# encode class values as integers
encoder = LabelEncoder()
encoder.fit(y)
y = encoder.transform(y)

# convert into PyTorch tensors
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)

# train-test split for evaluation of the model
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, shuffle=True)

# set up DataLoader for training set
loader = DataLoader(list(zip(X_train, y_train)), shuffle=True, batch_size=16)

# create model
model = nn.Sequential(
    nn.Linear(60, 60),
    nn.ReLU(),
    nn.Linear(60, 30),
    nn.ReLU(),
    nn.Linear(30, 1),
    nn.Sigmoid()
)

# Train the model
n_epochs = 200
loss_fn = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)
model.train()
for epoch in range(n_epochs):
    for X_batch, y_batch in loader:
        y_pred = model(X_batch)
        loss = loss_fn(y_pred, y_batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

# evaluate accuracy after training
model.eval()
y_pred = model(X_test)
```

```
acc = (y_pred.round() == y_test).float().mean()
acc = float(acc)
print("Model accuracy: %.2f%%" % (acc*100))
```

Listing 18.3: Reading data from CSV and train a model with DataLoader

18.3 Create Data Iterator using Dataset Class

In PyTorch, there is a `Dataset` class that can be tightly coupled with the `DataLoader` class. Recall that `DataLoader` expects its first argument can work with `len()` and with array index. The `Dataset` class is a base class for this. The reason you may want to use `Dataset` class is there are some special handling before you can get the data sample. For example, data should be read from database or disk and you only want to keep a few samples in memory rather than prefetch everything. Another example is to perform real-time preprocessing of data, such as random augmentation that is common in image tasks.

To use `Dataset` class, you just subclass from it and implement two member functions. Below is an example:

```
from torch.utils.data import Dataset

class SonarDataset(Dataset):
    def __init__(self, X, y):
        # convert into PyTorch tensors and remember them
        self.X = torch.tensor(X, dtype=torch.float32)
        self.y = torch.tensor(y, dtype=torch.float32)

    def __len__(self):
        # this should return the size of the dataset
        return len(self.X)

    def __getitem__(self, idx):
        # this should return one sample from the dataset
        features = self.X[idx]
        target = self.y[idx]
        return features, target
```

Listing 18.4: Creating a `Dataset`

This is not the most powerful way to use `Dataset` but simple enough to demonstrate how it works. With this, you can create a `DataLoader` and use it for model training. Modifying from the previous example, you have the following:

```
...

# setup DataLoader for training set
dataset = SonarDataset(X_train, y_train)
loader = DataLoader(dataset, shuffle=True, batch_size=16)

# create model
model = nn.Sequential(
    nn.Linear(60, 60),
    nn.ReLU(),
    nn.Linear(60, 30),
    nn.ReLU(),
    nn.Linear(30, 1),
    nn.Sigmoid()
)

# Train the model
n_epochs = 200
loss_fn = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)
model.train()
for epoch in range(n_epochs):
    for X_batch, y_batch in loader:
        y_pred = model(X_batch)
        loss = loss_fn(y_pred, y_batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

# evaluate accuracy after training
model.eval()
y_pred = model(torch.tensor(X_test, dtype=torch.float32))
y_test = torch.tensor(y_test, dtype=torch.float32)
acc = (y_pred.round() == y_test).float().mean()
acc = float(acc)
print("Model accuracy: %.2f%%" % (acc*100))
```

Listing 18.5: Using `DataLoader` with `Dataset`

You setup `dataset` as an instance of `SonarDataset` which you implemented the `__len__()` and `__getitem__()` functions. This is used in place of the list in the previous example to setup the `DataLoader` instance. Afterward, everything is the same in the training loop. Note that you still use PyTorch tensors directly for the test set in the example.

In the `__getitem__()` function, you take an integer that works like an array index and returns a pair, the features and the target. You can implement anything in this function: Run some code to generate a synthetic data sample, read data on the fly from the internet, or add random variations to the data. You will also find it useful in the situation that you cannot keep the entire dataset in memory, so you can load only the data samples that you need it.

In fact, since you created a PyTorch dataset, you don't need to use scikit-learn to split data into training set and test set. In `torch.utils.data` submodule, you have a function `random_split()` that works with `Dataset` class for the same purpose. A full example is below:

```
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, random_split, default_collate
from sklearn.preprocessing import LabelEncoder
```

```python
# Read data, convert to NumPy arrays
data = pd.read_csv("sonar.csv", header=None)
X = data.iloc[:, 0:60].values
y = data.iloc[:, 60].values

# encode class values as integers
encoder = LabelEncoder()
encoder.fit(y)
y = encoder.transform(y).reshape(-1, 1)

class SonarDataset(Dataset):
    def __init__(self, X, y):
        # convert into PyTorch tensors and remember them
        self.X = torch.tensor(X, dtype=torch.float32)
        self.y = torch.tensor(y, dtype=torch.float32)

    def __len__(self):
        # this should return the size of the dataset
        return len(self.X)

    def __getitem__(self, idx):
        # this should return one sample from the dataset
        features = self.X[idx]
        target = self.y[idx]
        return features, target

# set up DataLoader for data set
dataset = SonarDataset(X, y)
trainset, testset = random_split(dataset, [0.7, 0.3])
loader = DataLoader(trainset, shuffle=True, batch_size=16)

# create model
model = nn.Sequential(
    nn.Linear(60, 60),
    nn.ReLU(),
    nn.Linear(60, 30),
    nn.ReLU(),
    nn.Linear(30, 1),
    nn.Sigmoid()
)

# Train the model
n_epochs = 200
loss_fn = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)
model.train()
for epoch in range(n_epochs):
    for X_batch, y_batch in loader:
        y_pred = model(X_batch)
        loss = loss_fn(y_pred, y_batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

```
# create one test tensor from the testset
X_test, y_test = default_collate(testset)
model.eval()
y_pred = model(X_test)
acc = (y_pred.round() == y_test).float().mean()
acc = float(acc)
print("Model accuracy: %.2f%%" % (acc*100))
```

Listing 18.6: Complete example of training a model with `DataLoader` and `Dataset`

It is very similar to the example you have before. Beware that the PyTorch model still needs a tensor as input, not a `Dataset`. Hence in the above, you need to use the `default_collate()` function to collect samples from a dataset into tensors.

18.4 Further Readings

This section provides more resources on the topic if you are looking to go deeper.

Articles

torch.utils.data. PyTorch documentation.
https://pytorch.org/docs/stable/data.html

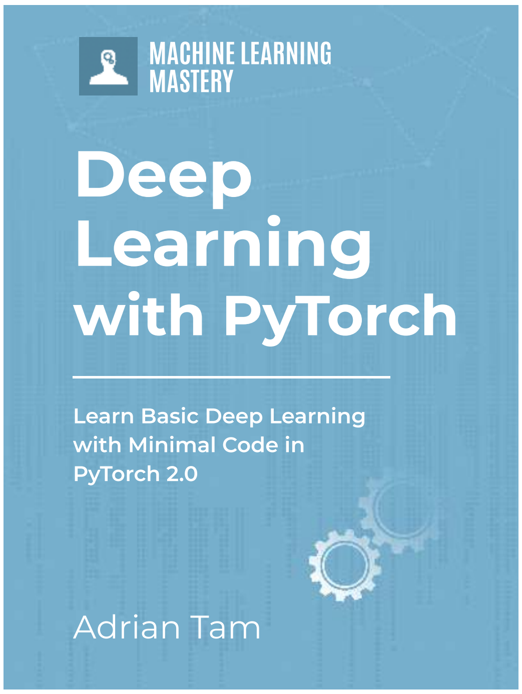
Datasets & DataLoaders. PyTorch tutorial.
https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

18.5 Summary

In this chapter, you learned how to use `DataLoader` to create shuffled batches of data and how to use `Dataset` to provide data samples. Specifically you learned:

- `DataLoader` as a convenient way of providing batches of data to the training loop
- How to use `Dataset` to produce data samples
- How combine `Dataset` and `DataLoader` to generate batches of data on the fly for model training

As you have seen in many examples so far, you make use of scikit-learn for various tasks such as train-test split. In the next chapter, you will learn how to wrap your PyTorch model into a scikit-learn model so you can make use of its model selection routines.


MACHINE LEARNING MASTERY
Deep Learning with PyTorch
Learn Basic Deep Learning with Minimal Code in PyTorch 2.0
Adrian Tam