# Python for Machine Learning

Learn Python Skills from Machine Learning Projects

Zhe Ming Chng  Daniel Chung  Stefania Cristina
Mehreen Saeed  Adrian Tam

*Authors*

Jason Brownlee

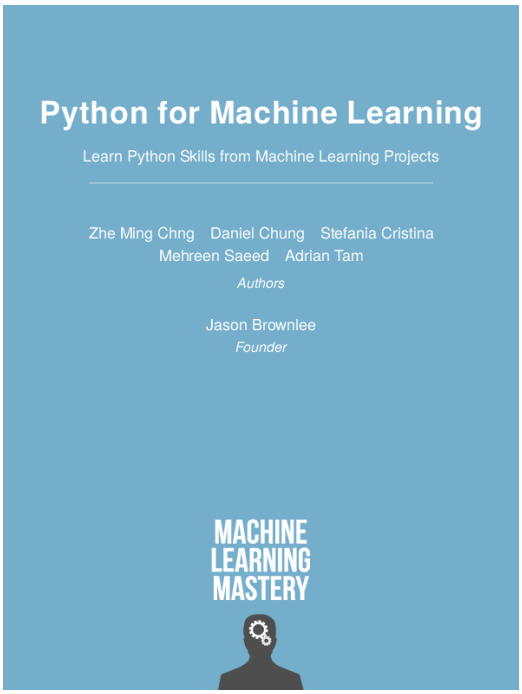
*Founder*

MACHINE
LEARNING
MASTERY

# This is Just a Sample

Thank-you for your interest in **Python for Machine Learning**.

This is just a sample of the full text. You can purchase the complete book online from: https://machinelearningmastery.com/python-for-machine-learning/



## Disclaimer



## Credits

Founder: Jason Brownlee
Authors: Zhe Ming Chng, Daniel Chung, Stefania Cristina, Mehreen Saeed, and Adrian Tam
Lead Editor: Adrian Tam
Technical Reviewers: Darci Heikkinen, Amy Lam, and Jerry Yiu

## Copyright



Edition: v1.00

# Contents

# Preface

Over the years, MachineLearningMastery.com has received a lot of email from people asking for help on their machine learning project. Some of them are quite specific, asking why a piece of code didn't work or why there is an error.

It is quite tedious to answer those emails but definitely not difficult. Questions like those are not related to machine learning at all but about how Python should be used as a language to keep our machine learning projects afloat. You can't work on a project without knowing your tools. Python is a tool for your machine learning project.

Python has been around for many years, and it is still evolving. At the time of writing, Python 3.10 is the latest version with the `match` statement introduced as the enhanced counterpart to `switch-case` statement in C. You should be able to find a book to learn Python from your local bookstore or library. But as a practitioner, you probably do not want to deep dive into the language but want to know just enough to get the job done. However, as Python's *ecosystem* has became very large, it is difficult to tell what you should know and what you might skip.

This book is not intended to be your first book on Python. But it can be your second book. I wish you learned about Python programming and can get something done. Perhaps you can use Python to answer some of the questions on Project Euler or Leetcode. Then this book tells you what's out there that can help your machine learning project. It can be a third-party library. It can be a way to make your Python program easier to use by your colleagues. It can also offer some bells and whistles to make your project more attractive.

The earlier chapters of this book give you some foundation. It helps if you came from a different programming language. Then we gradually introduce the tools such as logging, debugger, and testing frameworks to help you develop your Python projects. We even cover some third-party tools that might be useful for your machine learning projects, such as web scraping and visualization. While almost all the things covered in this book can be used outside of machine learning projects, we try to build the connection on how they can help using example code in machine learning. After reading this book, we hope you will find yourself a stronger machine learning engineer as you know your tool better.

# Introduction

Welcome to *Python for Machine Learning*.

Python is an amazing programming language. On one hand, it is simple and easy to read. You don't even need to learn about it but you can still understand what a Python code does most of the time. On the other hand, it is not a toy language. It can do a lot of things and do them well. Compared to other languages, it allows fast iteration. If you want to tweak your code a bit, you only need to change a line or two, and you can run the modified code right away. No need to update many places for small changes. No need to wait for minutes and hours to re-compile your code to run it.

That's the reason Python became the lingua franca for machine learning. In machine learning projects, we never know the right solution at the start. We need many experiments and iterations to finalize our approach. Having a language that allows us to iterate fast means we can improve our solution faster. As a result, a lot of people is using Python. And a lot of libraries are written for Python. This virtuous cycle made Python a mature language with a powerful ecosystem.

Python is never meant to run fast. In fact, if execution speed is the concern, using a different language such as C++ or Java might be a better idea. However, rather than *computer time*, the *human time* might be more valuable. Python is the language that allows you to trade off computer time for developer's time.

This book will make your human time more valuable by making you more productive. We tell you how you can get the most out of Python. What the techniques are to get more done in shorter time. Also, how we can save time in the long run by using tools such as unit testing, profiler, or connecting Python code to other tools.

## Who Is This Book for?

Despite the title, you don't need to know machine learning. You don't even need to work on machine learning in the future. Nothing in this book needs a prior knowledge in machine learning algorithms or libraries. However, you need to know Python. This is not a beginners' book. We are not going to cover what is a variable and what are loops. This book assumes:

- ▷ You know your way around Python IDEs and how to run a Python program.

- ▷ You know the basic Python language. You can tell how to use an `if` statement or how to use a `for` loop. You can tell why you want to use them.
- ▷ You can do simple tasks with Python. For example, you know how to write a function to do a binary search on a sorted array.

This book begins with some special language features that are unique to Python, expand to the many tools you can use from the Python ecosystem. Most of this guide was written in the top-down and results-first style that you're used to from MachineLearningMastery.com.

## What to Expect?

This book will teach you the bells and whistles of Python. If you came from a background of another programming language, you probably can learn the Python syntax in less than an hour and write some Python code immediately. But to make the most out of Python, this book tells you what to look at next. After reading and working through the book, you will know:

- ▷ The list comprehension syntax that allows you to write less code than a `for`-loop.
- ▷ The carefully selected set of built-in function in Python to save you time in daily tasks.
- ▷ Python dictionary is highly optimized, and we can achieve a lot with it. Hence you don't see other data structures such as trees or linked lists in Python library.
- ▷ The functional syntax in Python makes your imperative programming language on steroids.
- ▷ How to work faster in experiments by using tools and tricks such as breakpoints, debuggers, and profilers.
- ▷ How to create maintainable code by conforming to a coding standard, preparing unit tests, and adding input sanitation and guard rails in code.
- ▷ How to leverage the duck-typing nature of Python to write less code but achieve more.
- ▷ How to get data from the Internet for your machine learning project or other uses, including writing your own web scrapping code.
- ▷ How to visualize data in Python, either as a picture using matplotlib or as an interactive web page using Bokeh.
- ▷ How to use your Python program with other systems, such as a database or web browser.
- ▷ How to prepare for deployment so you can bring your Python program to other computers.

This book is not to replace your other Python tutorial book. In fact, you should read those first. In Appendix A, we list out some books for you to begin with before starting with this one.

# How to Read This Book?

This book was written to be read linearly, from start to finish. However, if you are already familiar with a topic, you should be able to skip a chapter without losing track. If you want to learn a particular topic, you can also flip straight to a particular section. The content of this book is created in a guidebook format. There is a substantial amount of example codes in this book. Therefore, you are expected to have this book opened on your workstation with an editor side-by-side so you can try out the examples while you read them. You can get the most from the content by extending and modifying the examples.

We cannot cover everything in Python. In fact, no book can do that. Instead, you will be provided with intuitions for the bits and pieces you need to know and how to get things done with Python. This book is divided into five parts:

- **Part I: Foundations**. The language features in Python that you probably won't find in another languages, as well as how to run a Python program.
- **Part II: Debugging, Profiling, and Linting**. As you are working on your Python program, there are tools to help you feel more confident you're doing the right thing. There are also tools to help you identify what you did wrong. This part gives you the concept of a call stack and debugger. We will also learn about the tools in this area.
- **Part III: Better Code, Better Software**. Once we get our code working correctly, we will likely want to add the bells and whistles to our program to make it better in some ways. It can be some logging function so we can trace a program's execution easier. It can be a decorator syntax that allows us to write code that is easier to maintain. It can also be a command line argument parser to pass in parameters to our Python script from the command line so we don't need to hard code the input in our program. We also cover the testing technique to check if our Python program can work as expected after some future modification.
- **Part IV: Furnish Your Library**. The large ecosystem in Python is part of the reason for its success. In this part, we will learn about getting and installing Python packages. Then we will learn about some useful packages to make your project more powerful. We will cover the use of databases, the visualization libraries, web scraping, multiprocessing, as well as let your Python program become a web application that can interact with a browser. In the final chapter, we will also learn about making our own code a new library. It will be helpful if we want to distribute our code to our friends and colleagues so they can run our code on their machine.
- **Part V: Platforms**. In the last part of this book, we will learn about some free resources on the Internet. We will cover the Google Colab and Kaggle Notebook. Both allow us to run our Python code on the cloud for free. It will be particularly useful for our machine learning projects as they both offer GPU computing.

These are not designed to tell you everything but just let you peek into the immense power of Python language and tools. Afterward, you should be able to use Python in a smarter way.

## How to Run the Examples?

All examples in this book are, of course, in Python. The examples in each chapter are complete and standalone. You should be able to run it successfully as-is without modification, given you have installed the required packages. No special IDE or notebooks are required. A command line execution environment is all it needs in most cases. A complete working example is always given at the end of the chapter. To avoid mistakes with copy-and-paste, all source codes are also provided with this book. Please use them whenever possible for a better learning experience.

All code examples were tested on a POSIX-compatible machine with Python 3.9. In case TensorFlow is used, we assume it is in TensorFlow 2.5 or above. Except some cases that we are demonstrating the new syntax, most code should work for Python 3.6 and above.

## About Further Readings

Each chapter includes a list of further reading resources. This may include:

- ▷ Books and book chapters
- ▷ API documentation
- ▷ Articles and web pages

Wherever possible, links to the relevant API documentation are provided in each chapter. Books referenced are provided with links to Amazon so you can learn more about them. If you find some good references, feel free to let us know so we can update this book.

# Python Debugging Tools

19

In all programming exercises, it is difficult to go far and deep without a handy debugger. The built-in debugger, `pdb`, in Python is a mature and capable one that can help us a lot if you know how to use it. In this chapter, we are going to see what the `pdb` can do for you as well as some of its alternatives.

In this chapter, you will learn:

- ▷ What a debugger can do
- ▷ How to control a debugger
- ▷ The limitation of Python's pdb and its alternatives

Let's get started.

## Overview

This chapter is in four parts; they are

- ▷ The concept of running a debugger
- ▷ Walk-through of using a debugger
- ▷ Debugger in Visual Studio Code
- ▷ Using GDB on a running Python program

## 19.1 The Concept of Running a Debugger

The purpose of a debugger is to provide you with a slow-motion button to control the flow of a program. It also allows you to freeze the program at a certain time and examine the state.

The simplest operation under a debugger is to *step through* the code. That is to run one line of code at a time and wait for your acknowledgment before proceeding to the next. The reason we want to run the program in a stop-and-go fashion is to allow us to check the logic and value or verify the algorithm.

For a larger program, we may not want to step through the code from the beginning as it may take a long time before we reach the line that we are interested in. Therefore, debuggers

also provide a *breakpoint* feature that will kick in when a specific line of code is reached. From that point onward, we can step through it line by line.

## 19.2 Walk-through of Using a Debugger

Let's see how we can make use of a debugger with an example. The following is the Python code for showing the particle swarm optimization in an animation:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

def f(x,y):
    "Objective function"
    return (x-3.14)**2 + (y-2.72)**2 + np.sin(3*x+1.41) + np.sin(4*y-1.73)

# Compute and plot the function in 3D within [0,5]x[0,5]
x, y = np.array(np.meshgrid(np.linspace(0,5,100), np.linspace(0,5,100)))
z = f(x, y)

# Find the global minimum
x_min = x.ravel()[z.argmin()]
y_min = y.ravel()[z.argmin()]

# Hyper-parameter of the algorithm
c1 = c2 = 0.1
w = 0.8

# Create particles
n_particles = 20
np.random.seed(100)
X = np.random.rand(2, n_particles) * 5
V = np.random.randn(2, n_particles) * 0.1

# Initialize data
pbest = X
pbest_obj = f(X[0], X[1])
gbest = pbest[:, pbest_obj.argmin()]
gbest_obj = pbest_obj.min()

def update():
    "Function to do one iteration of particle swarm optimization"
    global V, X, pbest, pbest_obj, gbest, gbest_obj
    # Update params
    r1, r2 = np.random.rand(2)
    V = w * V + c1*r1*(pbest - X) + c2*r2*(gbest.reshape(-1,1)-X)
    X = X + V
    obj = f(X[0], X[1])
    pbest[:, (pbest_obj >= obj)] = X[:, (pbest_obj >= obj)]
    pbest_obj = np.array([pbest_obj, obj]).min(axis=0)
    gbest = pbest[:, pbest_obj.argmin()]
    gbest_obj = pbest_obj.min()
```

```
# Set up base figure: The contour map
fig, ax = plt.subplots(figsize=(8,6))
fig.set_tight_layout(True)
img = ax.imshow(z, extent=[0, 5, 0, 5], origin='lower', cmap='viridis', alpha=0.5)
fig.colorbar(img, ax=ax)
ax.plot([x_min], [y_min], marker='x', markersize=5, color="white")
contours = ax.contour(x, y, z, 10, colors='black', alpha=0.4)
ax.clabel(contours, inline=True, fontsize=8, fmt="%.0f")
pbest_plot = ax.scatter(pbest[0], pbest[1], marker='o', color='black', alpha=0.5)
p_plot = ax.scatter(X[0], X[1], marker='o', color='blue', alpha=0.5)
p_arrow = ax.quiver(X[0], X[1], V[0], V[1], color='blue', width=0.005,
                    angles='xy', scale_units='xy', scale=1)
gbest_plot = plt.scatter([gbest[0]], [gbest[1]], marker='*', s=100, color='black',
                         alpha=0.4)
ax.set_xlim([0,5])
ax.set_ylim([0,5])

def animate(i):
    "Steps of PSO: algorithm update and show in plot"
    title = 'Iteration {:02d}'.format(i)
    # Update params
    update()
    # Set picture
    ax.set_title(title)
    pbest_plot.set_offsets(pbest.T)
    p_plot.set_offsets(X.T)
    p_arrow.set_offsets(X.T)
    p_arrow.set_UVC(V[0], V[1])
    gbest_plot.set_offsets(gbest.reshape(1,-1))
    return ax, pbest_plot, p_plot, p_arrow, gbest_plot

anim = FuncAnimation(fig, animate,
                     frames=list(range(1,50)), interval=500, blit=False, repeat=True)
anim.save("PSO.gif", dpi=120, writer="imagemagick")

print("PSO found best solution at f({})={}".format(gbest, gbest_obj))
print("Global optimal at f({})={}".format([x_min,y_min], f(x_min,y_min)))
```

*Listing 19.1: Particle swarm optimization code*

The particle swarm optimization is done by executing the `update()` function a number of times. Each time it runs, we are closer to the optimal solution to the objective function. We are using matplotlib's `FuncAnimation()` function instead of a loop to run `update()`, so we can capture the position of the particles at each iteration.

Assume this program is saved as `pso.py`. To run this program in the command line simply requires entering:

```
$ python pso.py
```

*Listing 19.2: Running a Python script*

The solution will be printed to the screen, and the animation will be saved as `PSO.gif`. But if we want to run it with the Python debugger, we enter the following in the command line:

```
$ python -m pdb pso.py
```

*Listing 19.3: Running a Python script under debugger*

The `-m pdb` part will load the `pdb` module and let the module execute the file `pso.py` for you. When you run this command, you will be welcomed with the `pdb` prompt as follows:

```
> /Users/mlm/pso.py(1)<module>()
-> import numpy as np
(Pdb)
```

*Output 19.1: Interactive session of the `pdb` debugger*

At the prompt, you can type in the debugger commands. To show the list of supported commands, we type "`h`" at the `pdb` prompt. And to show the details of the specific command (such as `list`), we can use "`h list`":

```
> /Users/mlm/pso.py(1)<module>()
-> import numpy as np
(Pdb) h

Documented commands (type help <topic>):
========================================
EOF    c          d        h         list      q        rv       undisplay
a      cl         debug    help      ll        quit     s        unt
alias  clear      disable  ignore    longlist  r        source   until
args   commands   display  interact  n         restart  step     up
b      condition  down     j         next      return   tbreak   w
break  cont       enable   jump      p         retval   u        whatis
bt     continue   exit     l         pp        run      unalias  where

Miscellaneous help topics:
==========================
exec  pdb

(Pdb)
```

*Output 19.2: Help message of `pdb` debugger*

At the beginning of a debugger session, we start with the first line of the program. Normally, a Python program would start with a few lines of `import`. We can use `n` to move to the next line or `s` to step into a function:

```
> /Users/mlm/pso.py(1)<module>()
-> import numpy as np
(Pdb) n
> /Users/mlm/pso.py(2)<module>()
-> import matplotlib.pyplot as plt
(Pdb) n
> /Users/mlm/pso.py(3)<module>()
```

```
-> from matplotlib.animation import FuncAnimation
(Pdb) n
> /Users/mlm/pso.py(5)<module>()
-> def f(x,y):
(Pdb) n
> /Users/mlm/pso.py(10)<module>()
-> x, y = np.array(np.meshgrid(np.linspace(0,5,100), np.linspace(0,5,100)))
(Pdb) n
> /Users/mlm/pso.py(11)<module>()
-> z = f(x, y)
(Pdb) s
--Call--
> /Users/mlm/pso.py(5)f()
-> def f(x,y):
(Pdb) s
> /Users/mlm/pso.py(7)f()
-> return (x-3.14)**2 + (y-2.72)**2 + np.sin(3*x+1.41) + np.sin(4*y-1.73)
(Pdb) s
--Return--
> /Users/mlm/pso.py(7)f()->array([[17.25... 7.46457344]])
-> return (x-3.14)**2 + (y-2.72)**2 + np.sin(3*x+1.41) + np.sin(4*y-1.73)
(Pdb) s
> /Users/mlm/pso.py(14)<module>()
-> x_min = x.ravel()[z.argmin()]
(Pdb)
```

*Output 19.3: Debugging session*

In `pdb`, the line of code will be printed before the prompt. Usually, the `n` command is what we would prefer as it executes that line of code and moves the flow at the *same level* without drilling down deeper. When we are at a line that calls a function (such as line 11 of the above program, that runs `z = f(x, y)`), we can use `s` to *step into* the function.

In the above example, we first step into the `f()` function, then another step to execute the computation, and finally, collect the return value from the function to give it back to the line that invoked the function. We see there are multiple `s` commands needed for a function as simple as one line because finding the function from the statement, calling the function, and returning it each takes one step. We can also see that in the body of the function, we called `np.sin()` like a function, but the debugger's `s` command does not go into it. It is because the `np.sin()` function is not implemented in Python but in C. The `pdb` does not support compiled code.

If the program is long, it is quite boring to use the `n` command many times to move to somewhere we are interested. We can use the `until` command with a line number to let the debugger run the program until that line is reached:

```
> /Users/mlm/pso.py(1)<module>()
-> import numpy as np
(Pdb) until 11
> /Users/mlm/pso.py(11)<module>()
-> z = f(x, y)
(Pdb) s
```

```
--Call--
> /Users/mlm/pso.py(5)f()
-> def f(x,y):
(Pdb) s
> /Users/mlm/pso.py(7)f()
-> return (x-3.14)**2 + (y-2.72)**2 + np.sin(3*x+1.41) + np.sin(4*y-1.73)
(Pdb) s
--Return--
> /Users/mlm/pso.py(7)f()->array([[17.25... 7.46457344]])
-> return (x-3.14)**2 + (y-2.72)**2 + np.sin(3*x+1.41) + np.sin(4*y-1.73)
(Pdb) s
> /Users/mlm/pso.py(14)<module>()
-> x_min = x.ravel()[z.argmin()]
(Pdb)
```

*Output 19.4: Debug using `until` command*

A command similar to `until` is `return`, which will execute the current function until the point that it is about to return. You can consider that as `until` with the line number equal to the last line of the current function. The `until` command is a one-off, meaning it will bring you to that line only. If you want to stop at a particular line *whenever* it is being run, we can make a *breakpoint* on it. For example, if we are interested in how each iteration of the optimization algorithm moves the solution, we can set a breakpoint right after the update is applied, by typing "`b`" and the line number at the prompt:

```
> /Users/mlm/pso.py(1)<module>()
-> import numpy as np
(Pdb) b 40
Breakpoint 1 at /Users/mlm/pso.py:40
(Pdb) c
> /Users/mlm/pso.py(40)update()
-> obj = f(X[0], X[1])
(Pdb) bt
  /usr/local/Cellar/python@3.9/3.9.9/Frameworks/Python.framework/Versions/3.9/lib/python3.
-> exec(cmd, globals, locals)
  <string>(1)<module>()
  /Users/mlm/pso.py(76)<module>()
-> anim.save("PSO.gif", dpi=120, writer="imagemagick")
  /usr/local/lib/python3.9/site-packages/matplotlib/animation.py(1078)save()
-> anim._init_draw()  # Clear the initial frame
  /usr/local/lib/python3.9/site-packages/matplotlib/animation.py(1698)_init_draw()
-> self._draw_frame(frame_data)
  /usr/local/lib/python3.9/site-packages/matplotlib/animation.py(1720)_draw_frame()
-> self._drawn_artists = self._func(framedata, *self._args)
  /Users/mlm/pso.py(65)animate()
-> update()
> /Users/mlm/pso.py(40)update()
-> obj = f(X[0], X[1])
(Pdb) p r1
0.8054505373292797
(Pdb) p r2
0.7543489945823536
(Pdb) p X
```

```
array([[2.77550474, 1.60073607, 2.14133019, 4.11466522, 0.2445649 ,
        0.65149396, 3.24520628, 4.08804798, 0.89696478, 2.82703884,
        4.42055413, 1.03681404, 0.95318658, 0.60737118, 1.17702652,
        4.67551174, 3.95781321, 0.95077669, 4.08220292, 1.33330594],
       [2.07985611, 4.53702225, 3.81359193, 1.83427181, 0.87867832,
        1.8423856 , 0.11392109, 1.2635162 , 3.84974582, 0.27397365,
        2.86219806, 3.05406841, 0.64253831, 1.85730719, 0.26090638,
        4.28053621, 4.71648133, 0.44101305, 4.14882396, 2.74620598]])
(Pdb) n
> /Users/mlm/pso.py(41)update()
-> pbest[:, (pbest_obj >= obj)] = X[:, (pbest_obj >= obj)]
(Pdb) n
> /Users/mlm/pso.py(42)update()
-> pbest_obj = np.array([pbest_obj, obj]).min(axis=0)
(Pdb) n
> /Users/mlm/pso.py(43)update()
-> gbest = pbest[:, pbest_obj.argmin()]
(Pdb) n
> /Users/mlm/pso.py(44)update()
-> gbest_obj = pbest_obj.min()
(Pdb)
```

*Output 19.5: Using breakpoing in debugger session*

After we set a breakpoint with the `b` command, we can let the debugger run our program until the breakpoint is hit. The `c` command means to *continue* until a trigger is met. At any point, we can use the `bt` command to show the traceback to check how we reached that point. We can also use the `p` command to print the variables (or an expression) to check what value they are holding.

Indeed, we can place a breakpoint with a condition so that it will stop only if the condition is met. The below will impose a condition that the first random number (`r1`) is greater than 0.5:

```
(Pdb) b 40, r1 > 0.5
Breakpoint 1 at /Users/mlm/pso.py:40
(Pdb) c
> /Users/mlm/pso.py(40)update()
-> obj = f(X[0], X[1])
(Pdb) p r1, r2
(0.8054505373292797, 0.7543489945823536)
(Pdb) c
> /Users/mlm/pso.py(40)update()
-> obj = f(X[0], X[1])
(Pdb) p r1, r2
(0.5404045753007164, 0.2967937508800147)
(Pdb)
```

*Output 19.6: Setting a breakpoint with a condition*

Indeed, we can also try to manipulate variables while we are debugging.

```
(Pdb) l
 35          global V, X, pbest, pbest_obj, gbest, gbest_obj
 36          # Update params
 37          r1, r2 = np.random.rand(2)
 38          V = w * V + c1*r1*(pbest - X) + c2*r2*(gbest.reshape(-1,1)-X)
 39          X = X + V
 40 B->        obj = f(X[0], X[1])
 41          pbest[:, (pbest_obj >= obj)] = X[:, (pbest_obj >= obj)]
 42          pbest_obj = np.array([pbest_obj, obj]).min(axis=0)
 43          gbest = pbest[:, pbest_obj.argmin()]
 44          gbest_obj = pbest_obj.min()
 45
(Pdb) p V
array([[ 0.03742722,  0.20930531,  0.06273426, -0.1710678 ,  0.33629384,
         0.19506555, -0.10238065, -0.12707257,  0.28042122, -0.03250191,
        -0.14004886,  0.13224399,  0.16083673,  0.21198813,  0.17530208,
        -0.27665503, -0.15344393,  0.20079061, -0.10057509,  0.09128536],
       [-0.05034548, -0.27986224, -0.30725954,  0.11214169,  0.0934514 ,
         0.00335978,  0.20517519,  0.06308483, -0.22007053,  0.26176423,
        -0.12617228, -0.05676629,  0.18296986, -0.01669114,  0.18934933,
        -0.27623121, -0.32482898,  0.213894  , -0.34427909, -0.12058168]])
(Pdb) p r1, r2
(0.5404045753007164, 0.2967937508800147)
(Pdb) r1 = 0.2
(Pdb) p r1, r2
(0.2, 0.2967937508800147)
(Pdb) j 38
> /Users/mlm/pso.py(38)update()
-> V = w * V + c1*r1*(pbest - X) + c2*r2*(gbest.reshape(-1,1)-X)
(Pdb) n
> /Users/mlm/pso.py(39)update()
-> X = X + V
(Pdb) p V
array([[ 0.02680837,  0.16594979,  0.06350735, -0.15577623,  0.30737655,
         0.19911613, -0.08242418, -0.12513798,  0.24939995, -0.02217463,
        -0.13474876,  0.14466204,  0.16661846,  0.21194543,  0.16952298,
        -0.24462505, -0.138997  ,  0.19377154, -0.10699911,  0.10631063],
       [-0.03606147, -0.25128615, -0.26362411,  0.08163408,  0.09842085,
         0.00765688,  0.19771385,  0.06597805, -0.20564599,  0.23113388,
        -0.0956787 , -0.07044121,  0.16637064, -0.00639259,  0.18245734,
        -0.25698717, -0.30336147,  0.19354112, -0.29904698, -0.08810355]])
(Pdb)
```

*Output 19.7: Manipulating variables while debugging*

In the above, we use the `l` command to list the code around the current statement (identified by the arrow `->`). In the listing, we can also see the breakpoint (marked with `B`) is set at line 40. As we can see the current value of `V` and `r1`, we can modify `r1` from 0.54 to 0.2 and run the statement on `V` again by using `j`(jump) to line 38. And as we see after we execute the statement with the `n` command, the value of `V` is changed.

If we use a breakpoint and find something unexpected, chances are that it was caused by issues in a different level of the call stack. Debuggers allow you to navigate to different levels:

```
(Pdb) bt
  /usr/local/Cellar/python@3.9/3.9.9/Frameworks/Python.framework/Versions/3.9/lib/python3.
-> exec(cmd, globals, locals)
  <string>(1)<module>()
  /Users/mlm/pso.py(76)<module>()
-> anim.save("PSO.gif", dpi=120, writer="imagemagick")
  /usr/local/lib/python3.9/site-packages/matplotlib/animation.py(1091)save()
-> anim._draw_next_frame(d, blit=False)
  /usr/local/lib/python3.9/site-packages/matplotlib/animation.py(1126)_draw_next_frame()
-> self._draw_frame(framedata)
  /usr/local/lib/python3.9/site-packages/matplotlib/animation.py(1720)_draw_frame()
-> self._drawn_artists = self._func(framedata, *self._args)
  /Users/mlm/pso.py(65)animate()
-> update()
> /Users/mlm/pso.py(39)update()
-> X = X + V
(Pdb) up
> /Users/mlm/pso.py(65)animate()
-> update()
(Pdb) bt
  /usr/local/Cellar/python@3.9/3.9.9/Frameworks/Python.framework/Versions/3.9/lib/python3.
-> exec(cmd, globals, locals)
  <string>(1)<module>()
  /Users/mlm/pso.py(76)<module>()
-> anim.save("PSO.gif", dpi=120, writer="imagemagick")
  /usr/local/lib/python3.9/site-packages/matplotlib/animation.py(1091)save()
-> anim._draw_next_frame(d, blit=False)
  /usr/local/lib/python3.9/site-packages/matplotlib/animation.py(1126)_draw_next_frame()
-> self._draw_frame(framedata)
  /usr/local/lib/python3.9/site-packages/matplotlib/animation.py(1720)_draw_frame()
-> self._drawn_artists = self._func(framedata, *self._args)
> /Users/mlm/pso.py(65)animate()
-> update()
  /Users/mlm/pso.py(39)update()
-> X = X + V
(Pdb) l
 60
 61     def animate(i):
 62         "Steps of PSO: algorithm update and show in plot"
 63         title = 'Iteration {:02d}'.format(i)
 64         # Update params
 65  ->     update()
 66         # Set picture
 67         ax.set_title(title)
 68         pbest_plot.set_offsets(pbest.T)
 69         p_plot.set_offsets(X.T)
 70         p_arrow.set_offsets(X.T)
(Pdb) p title
'Iteration 02'
(Pdb)
```

*Output 19.8: Navigate on the call stack while debugging*

In the above, the first `bt` command gives the call stack when we are at the bottom frame, i.e., the deepest of the call stack. We can see that we are about to execute the statement

`X = X + V`. Then, the `up` command moves our focus to one level up on the call stack, which is the line running the `update()` function (as we see at the line preceded with `>`). Since our focus is changed, the list command `l` will print a different fragment of code, and the `p` command can examine a variable in a different scope.

The above covers most of the useful commands in the debugger. If we want to terminate the debugger (which also terminates the program), we can use the `q` command to quit or hit Ctrl-D if your terminal supports it.

## 19.3 Debugger in Visual Studio Code

If you are not very comfortable running the debugger in command line, you can rely on the debugger from your IDE. Almost always, the IDE will provide you with some debugging facility. In Visual Studio Code, for example, you can launch the debugger in the "Run" menu.

The screen below shows Visual Studio Code during a debugging session. The buttons at the center top correspond to the `pdb` commands `continue`, `next`, `step`, `return`, `restart`, and `quit`, respectively. A breakpoint can be created by clicking on the line number, and a red dot will be appeared to identify that. The bonus of using an IDE is that the variables are shown immediately at each debugging step. We can also watch for an express and show the call stack. These are on the left side of the screen below.

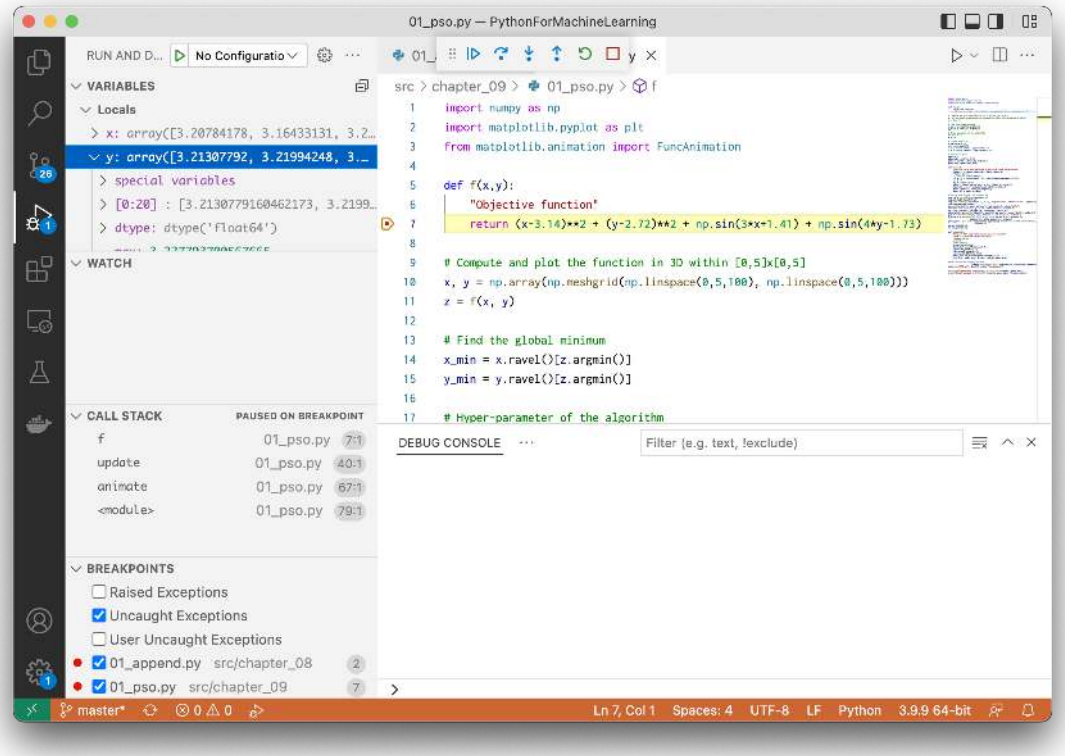



*Figure 19.1: Debugging session in Visual Studio Code*

## 19.4 Using GDB on a Running Python Program

The `pdb` from Python is suitable only for programs running from scratch. If we have a program already running but is stuck, we cannot use pdb to *hook into* it to check what's going on. The Python extension from GDB, however, can do this.

To demonstrate, let's consider a GUI application. It will wait until the user's action before the program can end. Hence it is a perfect example of how we can use `gdb` to hook into a running process. The code below is a "hello world" program using PyQt5 that just creates an empty window and waits for the user to close it:

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget, QMainWindow

class Frame(QMainWindow):
        def __init__(self):
                super().__init__()
                self.initUI()
        def initUI(self):
                self.setWindowTitle("Simple title")
                self.resize(800,600)

def main():
        app = QApplication(sys.argv)
        frame = Frame()
        frame.show()
        sys.exit(app.exec_())

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

*Listing 19.4: A simple program using PyQt5*

Let's save this program as `simpleqt.py` and run it using the following in Linux under an X window environment:

```
$ python simpleqt.py &
```

*Listing 19.5: Running a Python script in background*

The final `&` will make it run in the background. Now we can check for its process ID using the `ps` command:

```
$ ps a | grep python
```

*Listing 19.6: Running a Python script in background*

```
...
   3997 pts/1    Sl     0:00 python simpleqt.py
...
```

*Output 19.9: Result of Listing 19.6*

The `ps` command will tell you the process ID in the first column. If you have `gdb` installed with a Python extension, we can run:

```
$ gdb python 3997
```

*Listing 19.7: Running `gdb` and hook to a process ID*

This will bring you into the GDB's prompt:

```
GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
...
Type "apropos word" to search for commands related to "word"...
Reading symbols from python...
Reading symbols from /usr/lib/debug/.build-id/f9/02f8a561c3abdb9c8d8c859d4243bd8c3f928f.de
Attaching to program: /usr/local/bin/python, process 3997
[New LWP 3998]
[New LWP 3999]
[New LWP 4001]
[New LWP 4002]
[New LWP 4003]
[New LWP 4004]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
0x00007fb11b1c93ff in __GI___poll (fds=0x7fb110007220, nfds=3, timeout=-1) at ../sysdeps/u
29      ../sysdeps/unix/sysv/linux/poll.c: No such file or directory.
(gdb) py-bt
Traceback (most recent call first):
  <built-in method exec_ of QApplication object at remote 0x7fb115f64c10>
  File "/mnt/data/simpleqt.py", line 16, in main
    sys.exit(app.exec_())
  File "/mnt/data/simpleqt.py", line 19, in <module>
    main()
(gdb) py-list
  11
  12    def main():
  13            app = QApplication(sys.argv)
  14            frame = Frame()
  15            frame.show()
 >16            sys.exit(app.exec_())
  17
  18    if __name__ == '__main__':
  19            main()
(gdb)
```

*Output 19.10: Interactive session of GDB*

GDB is supposed to be a debugger for compiled programs (usually from C or C++). The Python extension allows you to check the code (written in Python) being run by the Python interpreter (written in C). It is less feature-rich than Python's `pdb` in terms of handling Python code but valuable when you need to hook it into a running process.

The commands supported under GDB are `py-list`, `py-bt`, `py-up`, `py-down`, and `py-print`. They are comparable to the same commands in `pdb` without the `py-` prefix.

GDB is useful if your Python code uses a library compiled from C (such as NumPy), and you want to investigate how it runs. It is also helpful to learn why your program is frozen by checking the call stack in run time. However, it may be rare that you need to use GDB to debug your machine learning project.

## 19.5 Further Readings

The Python `pdb` module's document is at

- *pdb module*. Python Standard Library.
  https://docs.python.org/3/library/pdb.html

But `pdb` is not the only debugger available. Some third-party tools are listed in:

- *Python Debugging Tools*. Python Wiki.
  https://wiki.python.org/moin/PythonDebuggingTools

For GDB with Python extension, it is best used in a Linux environment. Please see the following for more details on its usage:

- *Easier Python Debugging*. Fedora Wiki.
  https://fedoraproject.org/wiki/Features/EasierPythonDebugging
- *Debugging with GDB*. Python Wiki.
  https://wiki.python.org/moin/DebuggingWithGdb

The command interface of `pdb` is influenced by that of GDB. Hence we can learn the technique of debugging a program in general from the latter. A good primer on how to use a debugger would be:

- Norman Matloff. *The Art of Debugging with GDB, DDD, and Eclipse*. No Starch Press, 2008.
  https://www.amazon.com/dp/159327002X

## 19.6 Summary

In this chapter, you discovered the features of Python's `pdb`. Specifically, you learned:

- What can `pdb` do and how to use it
- The limitation and alternatives of `pdb`

In the next chapter, we will see that `pdb` is also a Python function that can be called inside a Python program.

# Web Frameworks for Your Python Projects

30

When we finish a Python project and roll it out for other people to use, the easiest way is to present our project as a command-line program. If you want to make it friendlier, you may want to develop a GUI for your program so people can interact with it with mouse clicks while it runs. Developing a GUI can be difficult as the model of human-computer interaction is complex. Therefore, a compromise is to create a web interface for your program. It requires some extra work compared to a pure command-line program, but not as heavy as writing an interface using, say, Qt5 library. In this chapter, we will show you the details of a web interface and how easy it is to give your program one.

After finishing this chapter, you will learn:

- ▷ The Flask framework from a simple example
- ▷ Using Dash to build an interactive web page entirely in Python
- ▷ How a web application operates

Let's get started!

## Overview

This chapter is divided into five parts; they are:

- ▷ Python and the web
- ▷ Flask for web API applications
- ▷ Dash for interactive widgets
- ▷ Polling in Dash
- ▷ Combining Flask and Dash

## 30.1 Python and the Web

The web is served using the hypertext transfer protocol (HTTP). Python's standard library comes with support for interacting with HTTP. If you simply want to run a web server with Python, nothing can be easier than going to a directory of files to serve and run the command.

```
python -m http.server
```

This will usually launch a web server at port 8000. If `index.html` exists in the directory, that would be the default page to serve if we open a browser on the same computer with the address `http://localhost:8000/`.

This built-in web server is great if we just need to quickly set up a web server (e.g., let another computer on the local network download a file). But it would not be sufficient if we want to do more, such as having some dynamic content.

Before we move on to the details, let's review what we would like to achieve when we speak of the web interface. Firstly, a web page in the modern day would be an interface for disseminating information to the user interactively. This means not only sending information from the server but also receiving input from the user. The browser is capable of rendering the information aesthetically.

Alternatively, we may use a web page without a browser. A case would be to download a file using web protocols. In Linux, we have the `wget` tool famous for doing this task. Another case is to query information or pass information to the server. For example, in AWS EC2 instances, you can check the machine instances' metadata[1] at the address `http://169.254.169.254/latest/meta-data/` (where the 169.254.169.254 is the special IP address available on EC2 machines). In Linux instances, we may use the `curl` tool to check. Its output will not be in HTML but in a plain-text machine-readable format. Sometimes, we call this the web API as we use it like a remotely executed function.

These are two different paradigms in web applications. The first one needs to write code for the interaction between user and server. The second one needs to set up various end-points on the URL so users can request different things using different addresses. In Python, there are third-party libraries to do both.

## 30.2 Flask for Web API Applications

The tools that allow us to write programs in Python to build a web-based application are called *web frameworks*. There are a lot. Django is probably the most famous one. However, the learning curve of different web frameworks can vary dramatically. Some web frameworks assume you use a model-view design, and you need to understand the rationale behind it to make sense of how you should use it.

As a machine learning practitioner, you probably want to do something quick, not too complex, and yet powerful enough to meet many use cases. Flask is probably a good choice in this class.

Flask is a lightweight web framework. You can run it as a command and use it as a Python module. Let's say we want to write a web server that reports the current time in any user-specified time zone. It can be done using Flask in a trivial way:

[1] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instancedata-data-retrieval.html

```
from datetime import datetime
import pytz
from flask import Flask

app = Flask("time now")

@app.route("/now/<path:timezone>")
def timenow(timezone):
    try:
        zone = pytz.timezone(timezone)
        now = datetime.now(zone)
        return now.strftime("%Y-%m-%d %H:%M:%S %z %Z\n")
    except pytz.exceptions.UnknownTimeZoneError:
        return f"Unknown time zone: {timezone}\n"

app.run()
```

*Listing 30.1: Create a time-reporting API using Flask*

Save the above into `server.py` or any filename you like, then run it on a terminal. You will see the following:

```
* Serving Flask app 'time now' (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000 (Press CTRL+C to quit)
```

*Output 30.1: Launch message of Listing 30.1*

This means your script is now running as a web server at `http://127.0.0.1:5000`. It will serve web requests forever until you interrupt it with Ctrl-C.

If you open up another terminal and query for the URL, e.g., using `curl` in Linux:

```
$ curl http://127.0.0.1:5000/now/Asia/Tokyo
2022-04-20 13:29:42 +0900 JST
```

*Listing 30.2: Querying our web API using `curl` command*

You will see the time printed on the screen in the time zone you requested (Asia/Tokyo in this case, you can see the list of all supported time zone on Wikipedia[2]). The string returned by the function in your code will be the content responded by the URL. If the time zone is not recognized, you will see the "Unknown time zone" message as returned by the `except` block in the code above.

If we want to extend this a little bit such that we will assume UTC if no time zone is provided, we just need to add another decorator to the function:

[2] https://en.wikipedia.org/wiki/List_of_tz_database_time_zones

```
from datetime import datetime
import pytz
from flask import Flask

app = Flask("time now")

@app.route('/now', defaults={'timezone': ''})
@app.route("/now/<path:timezone>")
def timenow(timezone):
    try:
        if not timezone:
            zone = pytz.utc
        else:
            zone = pytz.timezone(timezone)
        now = datetime.now(zone)
        return now.strftime("%Y-%m-%d %H:%M:%S %z %Z\n")
    except pytz.exceptions.UnknownTimeZoneError:
        return f"Unknown timezone: {timezone}\n"

app.run()
```

*Listing 30.3: Enhanced time-reporting API*

Restarting the server, we can see the result as follows:

```
$ curl http://127.0.0.1:5000/now/Asia/Tokyo
2022-04-20 13:37:27 +0900 JST
$ curl http://127.0.0.1:5000/now/Asia/Tok
Unknown timezone: Asia/Tok
$ curl http://127.0.0.1:5000/now
2022-04-20 04:37:29 +0000 UTC
```

*Listing 30.4: Querying our web API using `curl` command*

Nowadays, many such applications return a JSON string for more complex data, but technically anything can be delivered. If you wish to create more web APIs, simply define your functions to return the data and decorate it with `@app.route()` as in the above examples.

## 30.3 Dash for Interactive Widgets

The web end points, as provided by Flask, are powerful. A lot of web applications are done in this way. For example, we can write the web user interface using HTML and handle the user interaction with JavaScript. Once the user triggers an event, we can let JavaScript handle any UI change and create an AJAX call to the server by sending data to an end point and waiting for the reply. An AJAX call is asynchronous; hence when the web server's response is received (usually within a fraction of a section), JavaScript is triggered again to further update the UI to let the user know about it.

However, as the web interface gets more and more complex, writing JavaScript code can be tedious. Hence there are many *client-side* libraries to simplify this. Some are to simplify JavaScript programming, such as jQuery. Some are to change the way HTML and JavaScript should interact, such as ReactJS. But since we are developing machine learning

projects in Python, it would be great to develop an interactive web application in Python without resorting to JavaScript. Dash is a tool for this.

Let's consider an example in machine learning: We want to use the MNIST handwritten digits dataset to train a handwritten digit recognizer. The LeNet5 model is famous for this task. But we want to let the user fine-tune the LeNet5 model, retrain it, and then use it for recognition. Training a simple LeNet5 model can be done with only a few lines of code:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dense, AveragePooling2D, Flatten
from tensorflow.keras.utils import to_categorical

# Load MNIST digits
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Reshape data to (n_samples, height, width, n_channel)
X_train = np.expand_dims(X_train, axis=3).astype("float32")
X_test = np.expand_dims(X_test, axis=3).astype("float32")

# One-hot encode the output
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# LeNet5 model
model = Sequential([
    Conv2D(6, (5,5), activation="tanh",
           input_shape=(28,28,1), padding="same"),
    AveragePooling2D((2,2), strides=2),
    Conv2D(16, (5,5), activation="tanh"),
    AveragePooling2D((2,2), strides=2),
    Conv2D(120, (5,5), activation="tanh"),
    Flatten(),
    Dense(84, activation="tanh"),
    Dense(10, activation="softmax")
])

# Train the model
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=100, batch_size=32)
```

*Listing 30.5: The LeNet5 model for MNIST digit recognition*

There are several hyperparameters that we can change in this code, such as the activation function, the optimizer for training, the number of epochs, and the batch size. We can make an interface in Dash to let the user change these and retrain the model. This interface will be presented in HTML but coded in Python:

```
...
from flask import Flask
from dash import Dash, html, dcc

# default values
model_data = {
    "activation": "relu",
    "optimizer": "adam",
    "epochs": 100,
    "batchsize": 32,
}
...
server = Flask("mlm")
app = Dash(server=server)
app.layout = html.Div(
    id="parent",
    children=[
        html.H1(
            children="LeNet5 training",
            style={"textAlign": "center"}
        ),
        html.Div(
            className="flex-container",
            children=[
                html.Div(children=[
                    html.Div(id="activationdisplay", children="Activation:"),
                    dcc.Dropdown(
                        id="activation",
                        options=[
                            {"label": "Rectified linear unit", "value": "relu"},
                            {"label": "Hyperbolic tangent", "value": "tanh"},
                            {"label": "Sigmoidal", "value": "sigmoid"},
                        ],
                        value=model_data["activation"]
                    )
                ]),
                html.Div(children=[
                    html.Div(id="optimizerdisplay", children="Optimizer:"),
                    dcc.Dropdown(
                        id="optimizer",
                        options=[
                            {"label": "Adam", "value": "adam"},
                            {"label": "Adagrad", "value": "adagrad"},
                            {"label": "Nadam", "value": "nadam"},
                            {"label": "Adadelta", "value": "adadelta"},
                            {"label": "Adamax", "value": "adamax"},
                            {"label": "RMSprop", "value": "rmsprop"},
                            {"label": "SGD", "value": "sgd"},
                            {"label": "FTRL", "value": "ftrl"},
                        ],
                        value=model_data["optimizer"]
                    ),
                ]),
                html.Div(children=[
```

```
                    html.Div(id="epochdisplay", children="Epochs:"),
                    dcc.Slider(1, 200, 1, marks={1: "1", 100: "100", 200: "200"},
                               value=model_data["epochs"], id="epochs"),
                ]),
                html.Div(children=[
                    html.Div(id="batchdisplay", children="Batch size:"),
                    dcc.Slider(1, 128, 1, marks={1: "1", 128: "128"},
                               value=model_data["batchsize"], id="batchsize"),
                ]),
            ]
        ),
        html.Button(id="train", n_clicks=0, children="Train"),
    ]
)
```

*Listing 30.6: A web interface created in Dash*

Here we set up a Dash app built on top of a Flask server. The majority of the code above is to set up the *layout* of the Dash app that will be displayed on the web browser. The layout has a title on top, a button (with the label "Train") at the bottom, and a large box containing multiple option widgets in the middle. There is a dropdown box for an activation function, another for a training optimizer, and two sliders, one for the number of epochs and one for the batch size. The layout will be like the following:

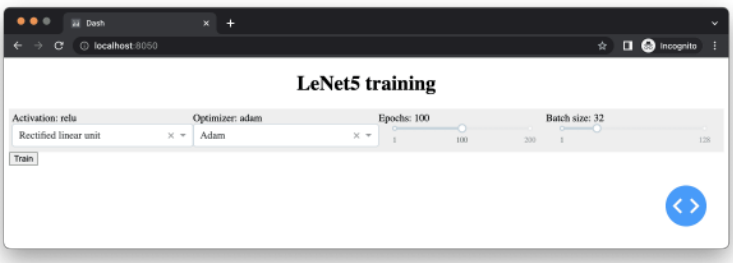



*Figure 30.1: The web page as created by Listing 30.6*

If you're familiar with HTML development, you probably noticed we used many `<div>` elements above. Moreover, we provided `style` arguments to some elements to change the way they are rendered on the browser. Indeed, we saved this Python code into file `server.py` and created a file `assets/main.css` with the following content:

```
.flex-container {
    display: flex;
    padding: 5px;
    flex-wrap: nowrap;
    background-color: #EEEEEE;
}

.flex-container > * {
    flex-grow: 1
}
```

*Listing 30.7: The CSS file for Listing 30.6*

This is how we can have the four different user options aligned horizontally when this code is run.

After we have the HTML frontend created, the key is to let the user change the hyperparameter by selecting from the dropdown list or moving the slider. Then, we kick start the model training after the user clicks on the "Train" button. Let's define the training function as follows:

```
...
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dense, AveragePooling2D, Flatten
from tensorflow.keras.callbacks import EarlyStopping

def train():
    activation = model_data["activation"]
    model = Sequential([
        Conv2D(6, (5, 5), activation=activation,
               input_shape=(28, 28, 1), padding="same"),
        AveragePooling2D((2, 2), strides=2),
        Conv2D(16, (5, 5), activation=activation),
        AveragePooling2D((2, 2), strides=2),
        Conv2D(120, (5, 5), activation=activation),
        Flatten(),
        Dense(84, activation=activation),
        Dense(10, activation="softmax")
    ])
    model.compile(loss="categorical_crossentropy",
                  optimizer=model_data["optimizer"],
                  metrics=["accuracy"])
    earlystop = EarlyStopping(monitor="val_loss", patience=3,
                              restore_best_weights=True)
    history = model.fit(
            X_train, y_train, validation_data=(X_test, y_test),
            epochs=model_data["epochs"],
            batch_size=model_data["batchsize"],
            verbose=0, callbacks=[earlystop])
    return model, history
```

*Listing 30.8: The training function for our LeNet5 model*

This function depends on an external dictionary `model_data` for the parameters and the dataset, such as `X_train` and `y_train`, defined outside of the function. It will just create a new model, train it, and return the model with the training history. We just need to run this function when the "Train" button on the browser is clicked. We set `verbose=0` in the `fit()` function to ask the training process not to print anything to the screen since it is supposed to run in the server while the user is looking at the browser. The user cannot see the terminal output at the server anyway. We can also take one step further to display the history of loss and evaluation metrics along the training epochs. This is what we need to do:

```
...
import pandas as pd
import plotly.express as px
```

```
from dash.dependencies import Input, Output, State

...
app.layout = html.Div(
    id="parent",
    children=[
        ...
        html.Button(id="train", n_clicks=0, children="Train"),
        dcc.Graph(id="historyplot"),
    ]
)

...
@app.callback(Output("historyplot", "figure"),
              Input("train", "n_clicks"),
              State("activation", "value"),
              State("optimizer", "value"),
              State("epochs", "value"),
              State("batchsize", "value"),
              prevent_initial_call=True)
def train_action(n_clicks, activation, optimizer, epoch, batchsize):
    model_data.update({
        "activation": activation,
        "optimizer": optimizer,
        "epoch": epoch,
        "batchsize": batchsize,
    })
    model, history = train()
    model_data["model"] = model  # keep the trained model
    history = pd.DataFrame(history.history)
    fig = px.line(history, title="Model training metrics")
    fig.update_layout(xaxis_title="epochs",
                      yaxis_title="metric value", legend_title="metrics")
    return fig
```

*Listing 30.9: Add a button to start training*

We first add a `Graph` component to the web page to display our training metrics. The `Graph` component is not a standard HTML element but a Dash component. There are a number of such components provided by Dash as its major feature. Dash is a sister project of Plotly, another visualization library similar to Bokeh that renders interactive charts into HTML. The `Graph` component is to display a Plotly chart.

Then we defined a function `train_action()` and decorated it with our Dash application's callback function. The function `train_action()` takes several inputs (model hyperparameters) and returns an output. In Dash, the output is usually a string, but we return a Plotly graph object here. The callback decorator requires us to specify the input and output. These are the web page components specified by their ID field and the property that served as the input or output. In this example, in addition to input and output, we also need some additional data called "states."

In Dash, input is what triggers an action. In this example, a button in Dash will remember the number of times it has been pressed in the component's property `n_clicks`. So we declared

the change in this property as the trigger for this function. Similarly, when this function is returned, the graph object will replace the `Graph` component. The state parameters are provided as non-trigger arguments to this function. The order of specifying the output, input, and states is essential as this is what the callback decorator expects, as well as the order of arguments to the function we defined.

We are not going to explain the Plotly syntax in detail. If you learned what a visualization library like Bokeh does, it should not be very difficult to adapt your knowledge to Plotly after consulting its documentation.

However, there is one thing we need to mention about Dash callbacks: When the web page is first loaded, all callbacks will be invoked once because the components are newly created. Since all components' properties changed from non-existence to some values, they are trigger events. If we do not want to have them invoked on the page load (e.g., in this case, we do not want our time-consuming training process to start until the user confirms the hyperparameters), we need to specify `prevent_initial_call=True` in the decorator.

We can go one step further by getting the hyperparameter selection interactive as well. This is polite because you give the user feedback on their action. As we already have a `<div>` element for the title of each selection component, we can make use of it for feedback by creating the following functions:

```
...

@app.callback(Output(component_id="epochdisplay", component_property="children"),
              Input(component_id="epochs", component_property="value"))
def update_epochs(value):
    return f"Epochs: {value}"


@app.callback(Output("batchdisplay", "children"),
              Input("batchsize", "value"))
def update_batchsize(value):
    return f"Batch size: {value}"


@app.callback(Output("activationdisplay", "children"),
              Input("activation", "value"))
def update_activation(value):
    return f"Activation: {value}"


@app.callback(Output("optimizerdisplay", "children"),
              Input("optimizer", "value"))
def update_optimizer(value):
    return f"Optimizer: {value}"
```

*Listing 30.10: Callback functions to display user selections*

These functions are trivial and return a string, which will become the "children" of the `<div>` elements. We also demonstrated the named arguments in the first function's decorator in case you prefer to be more explicit.

Putting everything together, the following is the complete code that can control a model training from a web interface:

```
import numpy as np
import pandas as pd
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dense, AveragePooling2D, Flatten
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping

import plotly.express as px
from dash import Dash, html, dcc
from dash.dependencies import Input, Output, State
from flask import Flask


server = Flask("mlm")
app = Dash(server=server)
# Load MNIST digits
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = np.expand_dims(X_train, axis=3).astype("float32")
X_test = np.expand_dims(X_test, axis=3).astype("float32")
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

model_data = {
    "activation": "relu",
    "optimizer": "adam",
    "epochs": 100,
    "batchsize": 32,
}


def train():
    activation = model_data["activation"]
    model = Sequential([
        Conv2D(6, (5, 5), activation=activation,
               input_shape=(28, 28, 1), padding="same"),
        AveragePooling2D((2, 2), strides=2),
        Conv2D(16, (5, 5), activation=activation),
        AveragePooling2D((2, 2), strides=2),
        Conv2D(120, (5, 5), activation=activation),
        Flatten(),
        Dense(84, activation=activation),
        Dense(10, activation="softmax")
    ])
    model.compile(loss="categorical_crossentropy",
                  optimizer=model_data["optimizer"],
                  metrics=["accuracy"])
    earlystop = EarlyStopping(monitor="val_loss", patience=3,
                              restore_best_weights=True)
    history = model.fit(
```

```
            X_train, y_train, validation_data=(X_test, y_test),
            epochs=model_data["epochs"],
            batch_size=model_data["batchsize"],
            verbose=0, callbacks=[earlystop])
    return model, history


app.layout = html.Div(
    id="parent",
    children=[
        html.H1(
            children="LeNet5 training",
            style={"textAlign": "center"}
        ),
        html.Div(
            className="flex-container",
            children=[
                html.Div(children=[
                    html.Div(id="activationdisplay"),
                    dcc.Dropdown(
                        id="activation",
                        options=[
                            {"label": "Rectified linear unit", "value": "relu"},
                            {"label": "Hyperbolic tangent", "value": "tanh"},
                            {"label": "Sigmoidal", "value": "sigmoid"},
                        ],
                        value=model_data["activation"]
                    )
                ]),
                html.Div(children=[
                    html.Div(id="optimizerdisplay"),
                    dcc.Dropdown(
                        id="optimizer",
                        options=[
                            {"label": "Adam", "value": "adam"},
                            {"label": "Adagrad", "value": "adagrad"},
                            {"label": "Nadam", "value": "nadam"},
                            {"label": "Adadelta", "value": "adadelta"},
                            {"label": "Adamax", "value": "adamax"},
                            {"label": "RMSprop", "value": "rmsprop"},
                            {"label": "SGD", "value": "sgd"},
                            {"label": "FTRL", "value": "ftrl"},
                        ],
                        value=model_data["optimizer"]
                    ),
                ]),
                html.Div(children=[
                    html.Div(id="epochdisplay"),
                    dcc.Slider(1, 200, 1, marks={1: "1", 100: "100", 200: "200"},
                               value=model_data["epochs"], id="epochs"),
                ]),
                html.Div(children=[
                    html.Div(id="batchdisplay"),
                    dcc.Slider(1, 128, 1, marks={1: "1", 128: "128"},
```

```
                                value=model_data["batchsize"], id="batchsize"),
                ]),
            ]
        ),
        html.Button(id="train", n_clicks=0, children="Train"),
        dcc.Graph(id="historyplot"),
    ]
)


@app.callback(Output(component_id="epochdisplay", component_property="children"),
              Input(component_id="epochs", component_property="value"))
def update_epochs(value):
    model_data["epochs"] = value
    return f"Epochs: {value}"


@app.callback(Output("batchdisplay", "children"),
              Input("batchsize", "value"))
def update_batchsize(value):
    model_data["batchsize"] = value
    return f"Batch size: {value}"


@app.callback(Output("activationdisplay", "children"),
              Input("activation", "value"))
def update_activation(value):
    model_data["activation"] = value
    return f"Activation: {value}"


@app.callback(Output("optimizerdisplay", "children"),
              Input("optimizer", "value"))
def update_optimizer(value):
    model_data["optimizer"] = value
    return f"Optimizer: {value}"


@app.callback(Output("historyplot", "figure"),
              Input("train", "n_clicks"),
              State("activation", "value"),
              State("optimizer", "value"),
              State("epochs", "value"),
              State("batchsize", "value"),
              prevent_initial_call=True)
def train_action(n_clicks, activation, optimizer, epoch, batchsize):
    model_data.update({
        "activation": activation,
        "optimizer": optimizer,
        "epcoh": epoch,
        "batchsize": batchsize,
    })
    model, history = train()
    model_data["model"] = model  # keep the trained model
```

```
    history = pd.DataFrame(history.history)
    fig = px.line(history, title="Model training metrics")
    fig.update_layout(xaxis_title="epochs",
                      yaxis_title="metric value", legend_title="metrics")
    return fig


# run server, with hot-reloading
app.run_server(debug=True, threaded=True)
```

*Listing 30.11: A web application implemented in Dash*

The final line of the above code is to run the Dash application, just like we run our Flask app in the previous section. The `debug=True` argument to the `run_server()` function is for "hot-reloading," which means to reload everything whenever Dash detects our script has been changed. It is convenient to see how it will work while editing our code on another window, as it doesn't require us to terminate our Dash server and run it again. The `threaded=True` is to ask the Dash server to run in multithreads when serving multiple requests. It is generally not recommended for Python programs to run in multithread due to the issue of global interpreter locks. However, it is acceptable in the web server environment as mostly the server is waiting for I/O. If not multithread, the option would be to run in multiprocesses. We cannot run a server in a single thread in a single process because even if we serve only one user, the browser will launch multiple HTTP queries at the same time (e.g., request for the CSS file we created above while loading the web page).

## 30.4 Polling in Dash

If we run the above Dash application with a moderate number of epochs, it would take noticeable time to complete. We want to see it running rather than just having the chart updated after it is finished. There is a way to ask Dash to *push* updates to our browser, but that would require a plugin (e.g., `dash_devices`[3] package can do this). But we can also ask the browser to *pull* for any updates. This design is called *polling*.

In the `train()` function we defined above, we set `verbose=0` to skip the terminal output. But we still need to know the progress of the training process. In Keras, this can be done with a custom callback. We can define one as follows:

```
...
from tensorflow.keras.callbacks import Callback


train_status = {
    "running": False,
    "epoch": 0,
    "batch": 0,
    "batch metric": None,
    "last epoch": None,
}
```

[3] https://pypi.org/project/dash-devices/

```
class ProgressCallback(Callback):
    def on_train_begin(self, logs=None):
        train_status["running"] = True
        train_status["epoch"] = 0
    def on_train_end(self, logs=None):
        train_status["running"] = False
    def on_epoch_begin(self, epoch, logs=None):
        train_status["epoch"] = epoch
        train_status["batch"] = 0
    def on_epoch_end(self, epoch, logs=None):
        train_status["last epoch"] = logs
    def on_train_batch_begin(self, batch, logs=None):
        train_status["batch"] = batch
    def on_train_batch_end(self, batch, logs=None):
        train_status["batch metric"] = logs


def train():
    ...
    history = model.fit(
            X_train, y_train, validation_data=(X_test, y_test),
            epochs=model_data["epochs"],
            batch_size=model_data["batchsize"],
            verbose=0, callbacks=[earlystop, ProgressCallback()])
    return model, history
```

*Listing 30.12: Callback class for Keras model to keep track of progress*

If we provide an instance of this class to the `fit()` function of a Keras model, the member function of this class will be invoked at the beginning or the end of the training cycle, or epoch, or a batch in one epoch. It is quite flexible on what we can do inside the function. At the end of an epoch or a batch, the `logs` arguments to the functions are a dictionary of the loss and validation metrics. Hence we defined a global dictionary object to remember the metrics.

Now given we can check the dictionary `train_status` any time to know the progress of our model training, we can modify our web page to display it:

```
...

app.layout = html.Div(
    id="parent",
    children=[
        ...
        html.Button(id="train", n_clicks=0, children="Train"),
        html.Pre(id="progressdisplay"),
        dcc.Interval(id="trainprogress", n_intervals=0, interval=1000),
        dcc.Graph(id="historyplot"),
    ]
)
```

```
import json


@app.callback(Output("progressdisplay", "children"),
              Input("trainprogress", "n_intervals"))
def update_progress(n):
    return json.dumps(train_status, indent=4)
```

*Listing 30.13: Display progress using the `Interval` component*

We create a non-visible component `dcc.Interval()` that changes its property `n_intervals` automatically once every 1000 milliseconds (= 1 second). Then we create a `<pre>` element below our "Train" button and name it `progressdisplay`. Whenever the `Interval` component fires, we convert the `train_status` dictionary into a JSON string and display it in that `<pre>` element. If you prefer, you can make a widget to display this information. Dash has a few provided.

With just these changes, your browser will look like the following when your model is trained:

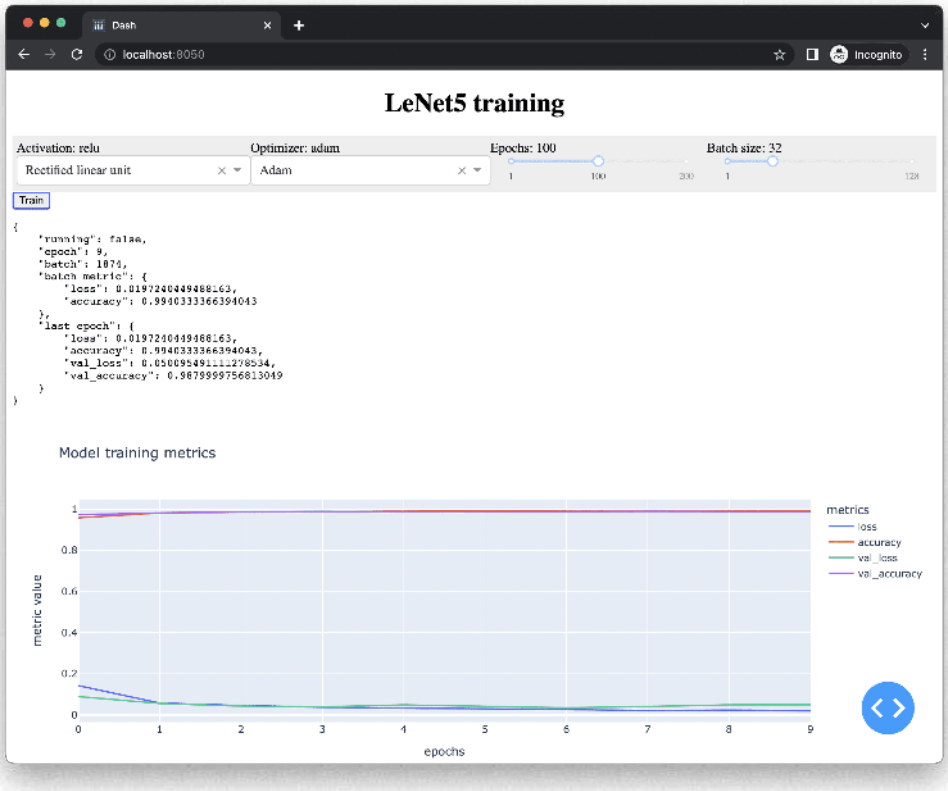



*Figure 30.2: The Dash web app after model training*

Below is the complete code. Don't forget you also need the `assets/main.css` file to properly render the web page:

```
import json

import numpy as np
import pandas as pd
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dense, AveragePooling2D, Flatten
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import Callback, EarlyStopping

import plotly.express as px
from dash import Dash, html, dcc
from dash.dependencies import Input, Output, State
from flask import Flask


server = Flask("mlm")
app = Dash(server=server)

# Load MNIST digits
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = np.expand_dims(X_train, axis=3).astype("float32")
X_test = np.expand_dims(X_test, axis=3).astype("float32")
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

model_data = {
    "activation": "relu",
    "optimizer": "adam",
    "epochs": 100,
    "batchsize": 32,
}

train_status = {
    "running": False,
    "epoch": 0,
    "batch": 0,
    "batch metric": None,
    "last epoch": None,
}


class ProgressCallback(Callback):
    def on_train_begin(self, logs=None):
        train_status["running"] = True
        train_status["epoch"] = 0
    def on_train_end(self, logs=None):
        train_status["running"] = False
    def on_epoch_begin(self, epoch, logs=None):
        train_status["epoch"] = epoch
        train_status["batch"] = 0
    def on_epoch_end(self, epoch, logs=None):
        train_status["last epoch"] = logs
    def on_train_batch_begin(self, batch, logs=None):
```

```
        train_status["batch"] = batch
    def on_train_batch_end(self, batch, logs=None):
        train_status["batch metric"] = logs


def train():
    activation = model_data["activation"]
    model = Sequential([
        Conv2D(6, (5, 5), activation=activation,
               input_shape=(28, 28, 1), padding="same"),
        AveragePooling2D((2, 2), strides=2),
        Conv2D(16, (5, 5), activation=activation),
        AveragePooling2D((2, 2), strides=2),
        Conv2D(120, (5, 5), activation=activation),
        Flatten(),
        Dense(84, activation=activation),
        Dense(10, activation="softmax")
    ])
    model.compile(loss="categorical_crossentropy",
                  optimizer=model_data["optimizer"],
                  metrics=["accuracy"])
    earlystop = EarlyStopping(monitor="val_loss", patience=3,
                              restore_best_weights=True)
    history = model.fit(
            X_train, y_train, validation_data=(X_test, y_test),
            epochs=model_data["epochs"],
            batch_size=model_data["batchsize"],
            verbose=0, callbacks=[earlystop, ProgressCallback()])
    return model, history


app.layout = html.Div(
    id="parent",
    children=[
        html.H1(
            children="LeNet5 training",
            style={"textAlign": "center"}
        ),
        html.Div(
            className="flex-container",
            children=[
                html.Div(children=[
                    html.Div(id="activationdisplay"),
                    dcc.Dropdown(
                        id="activation",
                        options=[
                            {"label": "Rectified linear unit", "value": "relu"},
                            {"label": "Hyperbolic tangent", "value": "tanh"},
                            {"label": "Sigmoidal", "value": "sigmoid"},
                        ],
                        value=model_data["activation"]
                    )
                ]),
                html.Div(children=[
```

```
                    html.Div(id="optimizerdisplay"),
                    dcc.Dropdown(
                        id="optimizer",
                        options=[
                            {"label": "Adam", "value": "adam"},
                            {"label": "Adagrad", "value": "adagrad"},
                            {"label": "Nadam", "value": "nadam"},
                            {"label": "Adadelta", "value": "adadelta"},
                            {"label": "Adamax", "value": "adamax"},
                            {"label": "RMSprop", "value": "rmsprop"},
                            {"label": "SGD", "value": "sgd"},
                            {"label": "FTRL", "value": "ftrl"},
                        ],
                        value=model_data["optimizer"]
                    ),
                ]),
                html.Div(children=[
                    html.Div(id="epochdisplay"),
                    dcc.Slider(1, 200, 1, marks={1: "1", 100: "100", 200: "200"},
                               value=model_data["epochs"], id="epochs"),
                ]),
                html.Div(children=[
                    html.Div(id="batchdisplay"),
                    dcc.Slider(1, 128, 1, marks={1: "1", 128: "128"},
                               value=model_data["batchsize"], id="batchsize"),
                ]),
            ]
        ),
        html.Button(id="train", n_clicks=0, children="Train"),
        html.Pre(id="progressdisplay"),
        dcc.Interval(id="trainprogress", n_intervals=0, interval=1000),
        dcc.Graph(id="historyplot"),
    ]
)


@app.callback(Output(component_id="epochdisplay", component_property="children"),
              Input(component_id="epochs", component_property="value"))
def update_epochs(value):
    return f"Epochs: {value}"


@app.callback(Output("batchdisplay", "children"),
              Input("batchsize", "value"))
def update_batchsize(value):
    return f"Batch size: {value}"


@app.callback(Output("activationdisplay", "children"),
              Input("activation", "value"))
def update_activation(value):
    return f"Activation: {value}"
```

```
@app.callback(Output("optimizerdisplay", "children"),
              Input("optimizer", "value"))
def update_optimizer(value):
    return f"Optimizer: {value}"


@app.callback(Output("historyplot", "figure"),
              Input("train", "n_clicks"),
              State("activation", "value"),
              State("optimizer", "value"),
              State("epochs", "value"),
              State("batchsize", "value"),
              prevent_initial_call=True)
def train_action(n_clicks, activation, optimizer, epoch, batchsize):
    model_data.update({
        "activation": activation,
        "optimizer": optimizer,
        "epoch": epoch,
        "batchsize": batchsize,
    })
    model, history = train()
    model_data["model"] = model  # keep the trained model
    history = pd.DataFrame(history.history)
    fig = px.line(history, title="Model training metrics")
    fig.update_layout(xaxis_title="epochs",
                      yaxis_title="metric value", legend_title="metrics")
    return fig


@app.callback(Output("progressdisplay", "children"),
              Input("trainprogress", "n_intervals"))
def update_progress(n):
    return json.dumps(train_status, indent=4)


# run server, with hot-reloading
app.run_server(debug=True, threaded=True)
```

*Listing 30.14: A web application in Dash with polling*

## 30.5 Combining Flask and Dash

Can you also provide a web interface to *use* the trained model? Certainly. It will be easier if the model takes a few numerical inputs because we can just provide an input box element on the page. In this case, since it is a handwritten digit recognition model, we need to have a way to provide an image on the browser and pass it on to the model at the server. Only then can we get the result and display it. There are two options we can do this: We can let the user upload an image of a digit for our model to recognize it, or we can let the user draw the image directly on the browser.

In HTML5, we have a `<canvas>` element that allows us to draw or display pixels in an area on the web page. We can make use of this to let the user draw on it, then convert it into

a numerical matrix of size $28 \times 28$, and send it to the server side for the model to predict and display the prediction result.

Doing this would not be Dash's job because we want to read the `<canvas>` element and convert it to a matrix of the correct format. We will do this in JavaScript. But after that, we would invoke the model in a web URL like what we described at the beginning of this chapter. A query is sent with the parameter, and the response from the server would be the digit that our model recognized.

Behind the scene, Dash uses Flask, and the root URL points to the Dash application. We can create a Flask endpoint that makes use of the model as follows:

```
...
@server.route("/recognize", methods=["POST"])
def recognize():
    if not model_data.get("model"):
        return "Please train your model."
    matrix = json.loads(request.form["matrix"])
    matrix = np.asarray(matrix).reshape(1, 28, 28)
    proba = model_data["model"].predict(matrix).reshape(-1)
    result = np.argmax(proba)
    return "Digit "+str(result)
```

*Listing 30.15: Creating a web API for using the trained model for recognition*

As we can recall, the variable `server` is the Flask server upon which we build our Dash application. We create an endpoint with its decorator. Since we are going to pass a $28 \times 28$ matrix as the parameter, we use the HTTP POST method, which is more suitable for a large block of data. The data provided by the POST method will not be part of the URL. Hence we do not set up a path parameter to the `@server.route()` decorator. Instead, we read the data with `request.form["matrix"]` in which `"matrix"` is the name of the parameter we passed in. Then we convert the string into a list of numbers by assuming it is in JSON format, and then further convert it into a NumPy array and give it to the model to predict the digit. We kept our trained model in `model_data["model"]`, but we can make the above code more robust by checking if this trained model exists and returning an error message if it does not.

To modify the web page, we just add a few more components:

```
app.layout = html.Div(
    id="parent",
    children=[
        ...
        dcc.Graph(id="historyplot"),
        html.Div(
            className="flex-container",
            id="predict",
            children=[
                html.Div(
                    children=html.Canvas(id="writing"),
                    style={"textAlign": "center"}
                ),
                html.Div(id="predictresult", children="?"),
```

```
                html.Pre(
                    id="lastinput",
                ),
            ]
        ),
        html.Div(id="dummy", style={"display": "none"}),
    ]
)
```

*Listing 30.16: New component to the web page for handwritten digit recognition*

The bottom one is a hidden `<div>` element that we will use later. The main block is another `<div>` element with three items in it, namely, a `<canvas>` element (with ID `"writing"`), a `<div>` element (with ID `"predictresult"`) to display the result, and a `<pre>` element (with ID `"lastinput"`) to display the matrix that we passed to the server.

Since these elements are not handled by Dash, we do not need to create any more functions in Python. But instead, we need to create a JavaScript file `assets/main.js` for the interaction with these components. A Dash application will automatically load everything under the directory `assets` and send it to the user when the web page is loaded. We can write this in plain JavaScript, but to make our code more concise, we will use jQuery. Hence we need to tell Dash that we will require jQuery in this web application:

```
...
app = Dash(server=server,
           external_scripts=[
               "https://code.jquery.com/jquery-3.6.0.min.js"
           ])
```

*Listing 30.17: Adding jQuery dependency to our web page*

The `external_scripts` argument is a list of URLs to be loaded as additional scripts *before* the web page is loaded. Hence we usually provide the library here but keep our own code away.

Our own Javascript code would be a single function because it is called after our web page is fully loaded:

```
function pageinit() {
    // Set up canvas object
    var canvas = document.getElementById("writing");
    canvas.width = parseInt($("#writing").css("width"));
    canvas.height = parseInt($("#writing").css("height"));
    var context = canvas.getContext("2d");  // to remember drawing
    context.strokeStyle = "#FF0000";        // draw in bright red
    context.lineWidth = canvas.width / 15;  // thickness adaptive to canvas size

    ...
};
```

*Listing 30.18: Javascript function to be invoked at the browser*

We first set up our `<canvas>` element in Javascript. These are specific to our requirements. Firstly, we added the following into our `assets/main.css`:

```
canvas#writing {
    width: 300px;
    height: 300px;
    margin: auto;
    padding: 10px;
    border: 3px solid #7f7f7f;
    background-color: #FFFFFF;
}
```

*Listing 30.19: CSS entry for the canvas component*

This fixed the width and height to 300 pixels to make our canvas square, along with other cosmetic fine tuning. Since ultimately, we would convert our handwriting into a $28 \times 28$ pixel image to fit what our model expects, every stroke we write on the canvas cannot be too thin. Therefore we set the stoke width relative to the canvas size.

Having this is not enough to make our canvas usable. Let's assume we never use it on mobile devices but only on a desktop browser; the drawing is done by mouse click and movements. We need to define what a mouse click does on the canvas. Hence we added the following functions to JavaScript code:

```
function pageinit() {
    ...

    // Canvas reset by timeout
    var timeout = null; // holding the timeout event
    var reset = function() {
        // clear the canvas
        context.clearRect(0, 0, canvas.width, canvas.height);
    }

    // Set up drawing with mouse
    var mouse = {x:0, y:0}; // to remember the coordinate w.r.t. canvas
    var onPaint = function() {
        clearTimeout(timeout);
        // event handler for mouse move in canvas
        context.lineTo(mouse.x, mouse.y);
        context.stroke();
    };

    // HTML5 Canvas mouse event - in case of desktop browser
    canvas.addEventListener("mousedown", function(e) {
        clearTimeout(timeout);
        // mouse down, begin path at current mouse position
        context.moveTo(mouse.x, mouse.y);
        context.beginPath();
        // all mouse move from now on should be painted
        canvas.addEventListener("mousemove", onPaint, false);
    }, false);
    canvas.addEventListener("mousemove", function(e) {
        // mouse move remember position w.r.t. canvas
        mouse.x = e.pageX - this.offsetLeft;
        mouse.y = e.pageY - this.offsetTop;
```

```
    }, false);
    canvas.addEventListener("mouseup", function(e) {
        clearTimeout(timeout);
        // all mouse move from now on should NOT be painted
        canvas.removeEventListener("mousemove", onPaint, false);
        // read drawing into image
        var img = new Image(); // on load, this will be the canvas in same WxH
        img.onload = function() {
            // Draw the 28x28 to top left corner of canvas
            context.drawImage(img, 0, 0, 28, 28);
            // Extract data: Each pixel becomes a RGBA value, hence 4 bytes each
            var data = context.getImageData(0, 0, 28, 28).data;
            var input = [];
            for (var i=0; i<data.length; i += 4) {
                // scan each pixel, extract first byte (R component)
                input.push(data[i]);
            };

            // TODO: use "input" for prediction
        };
        img.src = canvas.toDataURL("image/png");
        timeout = setTimeout(reset, 5000); // clear canvas after 5 sec
    }, false);
};
```

*Listing 30.20: JavaScript code to handle drawing on canvas*

This is a bit verbose, but essentially, we ask to listen on three mouse events on the canvas, namely, press down the mouse button, moving the mouse, and release the mouse button. These three events combined are how we draw one stroke on the canvas.

Firstly, the `mousemove` event handler we added to the `<canvas>` element is to simply remember the current mouse position in the JavaScript object `mouse`.

Then in the `mousedown` event handler, we start our drawing context at the latest mouse position. And since the drawing is started, all subsequent mouse moves should be painted on the canvas. We defined the `onPaint` function to extend a line segment on the canvas to the current mouse position. This function is now registered as an additional event handler to the `mousemove` event.

Finally, the `mouseup` event handler is to handle the case when the user finishes one stroke and releases the mouse button. All subsequent mouse movements should not be painted on the canvas, so we need to remove the event handler of the `onPaint` function. Then, as we finished one stroke, this *may be* a finished digit, so we want to extract it into a $28 \times 28$ pixel version. This can be done easily. We simply create a new `Image` object in JavaScript and load our entire canvas into it. When this is finished, JavaScript will automatically invoke the `onload` function associated with it. In which, we will transform this `Image` object into $28 \times 28$ pixels and draw it into the top left corner of our `context` object. Then we read it back pixel by pixel (each will be the RGB values of 0 to 255 per channel, but since we paint in red, we concern only the red channel) into the JavaScript array `input`. We just need to give this `input` array to our model, and the prediction can be carried out.

We do not want to create any additional buttons to clear our canvas or submit our digit for recognition. Hence we want to clear our canvas automatically if the user has not drawn anything new for 5 seconds. This is achieved with the JavaScript function `setTimeout()` and `clearTimeout()`. We make a `reset` function to clear the canvas, which will be fired at 5 seconds after the `mouseup` event. And this scheduled call to the `reset` function will be canceled whenever a drawing event happens before the timeout. Similarly, the recognition is automatic whenever a `mouseup` event happens.

Given we have the input data in $28 \times 28$ pixels transformed into a JavaScript array, we can just make use of the `recognize` end point we created with Flask. It would be helpful if we could see what we passed into `recognize` and what it returns. So we display the input in the `<pre>` element with ID `lastinput`, and display the result returned by the `recognize` end point in the `<div>` element with ID `predictresult`. This can be done easily by extending a bit on the `mouseup` event handler:

```
function pageinit() {
    canvas.addEventListener("mouseup", function(e) {
        ...
        img.onload = function() {
            ...
            var input = [];
            for (var i=0; i<data.length; i += 4) {
                // scan each pixel, extract first byte (R component)
                input.push(data[i]);
            };
            var matrix = [];
            for (var i=0; i<input.length; i+=28) {
                matrix.push(input.slice(i, i+28).toString());
            };
            $("#lastinput").html("[[" + matrix.join("],<br/>[") + "]]");
            // call predict function with the matrix
            predict(input);
        };
        img.src = canvas.toDataURL("image/png");
        setTimeout(reset, 5000); // clear canvas after 5 sec
    }, false);

    function predict(input) {
        $.ajax({
            type: "POST",
            url: "/recognize",
            data: {"matrix": JSON.stringify(input)},
            success: function(result) {
                $("#predictresult").html(result);
            }
        });
    };
};
```

*Listing 30.21: JavaScript code to send the handwritten digit to server side for recognition*

We defined a new Javascript function `predict()` that fires an AJAX call to the `recognize` end point that we set up with Flask. It is using a POST method with the data `matrix` assigned with a JSON version of the Javascript array. We cannot pass an array directly on an HTTP request because everything has to be serialized. When the AJAX call returns, we update our `<div>` element with the result.

This `predict()` function is invoked by the `mouseup` event handler when we finished transforming our $28 \times 28$ pixel image into a numerical array. At the same time, we write a version into the `<pre>` element solely for display purposes.

Up to here, our application is finished. But we still need to call the `pageinit()` function when our Dash application is loaded. Behind the scenes, the Dash application is using React for the web for delayed rendering. Therefore we should not hook our `pageinit()` function to the `document.onload` event handler, or we will find that the components we are looking for do not exist. The correct way to call a Javascript function only when the Dash application is fully loaded is to set up a *client callback*, which means it is a callback but handled by the browser-side Javascript rather than on the server-side Python. We add the following function call to our Python program, `server.py`:

```
...
app.clientside_callback(
    "pageinit",
    Output("dummy", "children"),
    Input("dummy", "children")
)
```

*Listing 30.22: A client-side callback to make the browser load our Javascript function at start*

The `clientside_callback()` function is not used as a decorator but as a complete function call. It takes a Javascript function as the first argument (the name of a function defined or the Javascript code to define one here), and the `Output` and `Input` object as the second and third arguments like the case of callback decorators. Because of this, we created a hidden dummy component in our web page layout just to help triggering the Javascript function at page load. All Dash callback would be invoked once unless `prevent_initial_call=True` is an argument to the callback.

Here we are all set. We can now run our `server.py` script to start our web server, and it will load the two files under the `assets/` directory. Opening a browser to visit the URL reported by our Dash application, we can change the hyperparameter and train the model, then use the model for prediction.

Tying everything together, the below is the complete code on our Javascript part, saved as `assets/main.js`:

```
function pageinit() {
    // Set up canvas object
    var canvas = document.getElementById("writing");
    canvas.width = parseInt($("#writing").css("width"));
    canvas.height = parseInt($("#writing").css("height"));
    var context = canvas.getContext("2d");  // to remember drawing
```

```
    context.strokeStyle = "#FF0000";        // draw in bright red
    context.lineWidth = canvas.width / 15;  // thickness adaptive to canvas size

    // Canvas reset by timeout
    var timeout = null; // holding the timeout event
    var reset = function() {
        // clear the canvas
        context.clearRect(0, 0, canvas.width, canvas.height);
    }

    // Set up drawing with mouse
    var mouse = {x:0, y:0}; // to remember the coordinate w.r.t. canvas
    var onPaint = function() {
        clearTimeout(timeout);
        // event handler for mouse move in canvas
        context.lineTo(mouse.x, mouse.y);
        context.stroke();
    };

    // HTML5 Canvas mouse event - in case of desktop browser
    canvas.addEventListener("mousedown", function(e) {
        clearTimeout(timeout);
        // mouse down, begin path at mouse position
        context.moveTo(mouse.x, mouse.y);
        context.beginPath();
        // all mouse move from now on should be painted
        canvas.addEventListener("mousemove", onPaint, false);
    }, false);
    canvas.addEventListener("mousemove", function(e) {
        // mouse move remember position w.r.t. canvas
        mouse.x = e.pageX - this.offsetLeft;
        mouse.y = e.pageY - this.offsetTop;
    }, false);
    canvas.addEventListener("mouseup", function(e) {
        // all mouse move from now on should NOT be painted
        canvas.removeEventListener("mousemove", onPaint, false);
        clearTimeout(timeout);
        // read drawing into image
        var img = new Image(); // on load, this will be the canvas in same WxH
        img.onload = function() {
            // Draw the 28x28 to top left corner of canvas
            context.drawImage(img, 0, 0, 28, 28);
            // Extract data: Each pixel becomes a RGBA value, hence 4 bytes each
            var data = context.getImageData(0, 0, 28, 28).data;
            var input = [];
            for (var i=0; i<data.length; i += 4) {
                // scan each pixel, extract first byte (R component)
                input.push(data[i]);
            };
            var matrix = [];
            for (var i=0; i<input.length; i+=28) {
                matrix.push(input.slice(i, i+28).toString());
            };
            $("#lastinput").html("[[" + matrix.join("],\n[") + "]]");
```

```
            // call predict function with the matrix
            predict(input);
        };
        img.src = canvas.toDataURL("image/png");
        timeout = setTimeout(reset, 5000); // clear canvas after 5 sec
    }, false);

    function predict(input) {
        $.ajax({
            type: "POST",
            url: "/recognize",
            data: {"matrix": JSON.stringify(input)},
            success: function(result) {
                $("#predictresult").html(result);
            }
        });
    };
};
```

*Listing 30.23: Javascript code for our web app of handwritten digit recognition*

And the following is the complete code for the CSS, `assets/main.css` (the `pre#lastinput` part is to use a smaller font to display our input matrix):

```
.flex-container {
    display: flex;
    padding: 5px;
    flex-wrap: nowrap;
    background-color: #EEEEEE;
}

.flex-container > * {
    flex-grow: 1
}

canvas#writing {
    width: 300px;
    height: 300px;
    margin: auto;
    padding: 10px;
    border: 3px solid #7f7f7f;
    background-color: #FFFFFF;
}

pre#lastinput {
    font-size: 50%;
}
```

*Listing 30.24: CSS for the web application*

The following is the main Python program, `server.py`:

```
import json

import numpy as np
import pandas as pd
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Conv2D, Dense, AveragePooling2D, Flatten
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import Callback, EarlyStopping

import plotly.express as px
from dash import Dash, html, dcc
from dash.dependencies import Input, Output, State
from flask import Flask, request


server = Flask("mlm")
app = Dash(server=server,
           external_scripts=[
               "https://code.jquery.com/jquery-3.6.0.min.js"
           ])

# Load MNIST digits
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = np.expand_dims(X_train, axis=3).astype("float32")
X_test = np.expand_dims(X_test, axis=3).astype("float32")
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

model_data = {
    "activation": "relu",
    "optimizer": "adam",
    "epochs": 100,
    "batchsize": 32,
    "model": None
}
train_status = {
    "running": False,
    "epoch": 0,
    "batch": 0,
    "batch metric": None,
    "last epoch": None,
}


class ProgressCallback(Callback):
    def on_train_begin(self, logs=None):
        train_status["running"] = True
        train_status["epoch"] = 0
    def on_train_end(self, logs=None):
        train_status["running"] = False
    def on_epoch_begin(self, epoch, logs=None):
        train_status["epoch"] = epoch
        train_status["batch"] = 0
```

```
    def on_epoch_end(self, epoch, logs=None):
        train_status["last epoch"] = logs
    def on_train_batch_begin(self, batch, logs=None):
        train_status["batch"] = batch
    def on_train_batch_end(self, batch, logs=None):
        train_status["batch metric"] = logs


def train():
    activation = model_data["activation"]
    model = Sequential([
        Conv2D(6, (5, 5), activation=activation,
               input_shape=(28, 28, 1), padding="same"),
        AveragePooling2D((2, 2), strides=2),
        Conv2D(16, (5, 5), activation=activation),
        AveragePooling2D((2, 2), strides=2),
        Conv2D(120, (5, 5), activation=activation),
        Flatten(),
        Dense(84, activation=activation),
        Dense(10, activation="softmax")
    ])
    model.compile(loss="categorical_crossentropy",
                  optimizer=model_data["optimizer"],
                  metrics=["accuracy"])
    earlystop = EarlyStopping(monitor="val_loss", patience=3,
                              restore_best_weights=True)
    history = model.fit(
            X_train, y_train, validation_data=(X_test, y_test),
            epochs=model_data["epochs"],
            batch_size=model_data["batchsize"],
            verbose=0, callbacks=[earlystop, ProgressCallback()])
    return model, history


app.layout = html.Div(
    id="parent",
    children=[
        html.H1(
            children="LeNet5 training",
            style={"textAlign": "center"}
        ),
        html.Div(
            className="flex-container",
            children=[
                html.Div(children=[
                    html.Div(id="activationdisplay"),
                    dcc.Dropdown(
                        id="activation",
                        options=[
                            {"label": "Rectified linear unit", "value": "relu"},
                            {"label": "Hyperbolic tangent", "value": "tanh"},
                            {"label": "Sigmoidal", "value": "sigmoid"},
                        ],
                        value=model_data["activation"]
```

```
                    )
                ]),
                html.Div(children=[
                    html.Div(id="optimizerdisplay"),
                    dcc.Dropdown(
                        id="optimizer",
                        options=[
                            {"label": "Adam", "value": "adam"},
                            {"label": "Adagrad", "value": "adagrad"},
                            {"label": "Nadam", "value": "nadam"},
                            {"label": "Adadelta", "value": "adadelta"},
                            {"label": "Adamax", "value": "adamax"},
                            {"label": "RMSprop", "value": "rmsprop"},
                            {"label": "SGD", "value": "sgd"},
                            {"label": "FTRL", "value": "ftrl"},
                        ],
                        value=model_data["optimizer"]
                    ),
                ]),
                html.Div(children=[
                    html.Div(id="epochdisplay"),
                    dcc.Slider(1, 200, 1, marks={1: "1", 100: "100", 200: "200"},
                               value=model_data["epochs"], id="epochs"),
                ]),
                html.Div(children=[
                    html.Div(id="batchdisplay"),
                    dcc.Slider(1, 128, 1, marks={1: "1", 128: "128"},
                               value=model_data["batchsize"], id="batchsize"),
                ]),
            ]
        ),
        html.Button(id="train", n_clicks=0, children="Train"),
        html.Pre(id="progressdisplay"),
        dcc.Interval(id="trainprogress", n_intervals=0, interval=1000),
        dcc.Graph(id="historyplot"),
        html.Div(
            className="flex-container",
            id="predict",
            children=[
                html.Div(
                    children=html.Canvas(id="writing"),
                    style={"textAlign": "center"}
                ),
                html.Div(id="predictresult", children="?"),
                html.Pre(
                    id="lastinput",
                ),
            ]
        ),
        html.Div(id="dummy", style={"display": "none"}),
    ]
)
```

```
@app.callback(Output(component_id="epochdisplay", component_property="children"),
              Input(component_id="epochs", component_property="value"))
def update_epochs(value):
    model_data["epochs"] = value
    return f"Epochs: {value}"


@app.callback(Output("batchdisplay", "children"),
              Input("batchsize", "value"))
def update_batchsize(value):
    model_data["batchsize"] = value
    return f"Batch size: {value}"


@app.callback(Output("activationdisplay", "children"),
              Input("activation", "value"))
def update_activation(value):
    model_data["activation"] = value
    return f"Activation: {value}"


@app.callback(Output("optimizerdisplay", "children"),
              Input("optimizer", "value"))
def update_optimizer(value):
    model_data["optimizer"] = value
    return f"Optimizer: {value}"


@app.callback(Output("historyplot", "figure"),
              Input("train", "n_clicks"),
              State("activation", "value"),
              State("optimizer", "value"),
              State("epochs", "value"),
              State("batchsize", "value"),
              prevent_initial_call=True)
def train_action(n_clicks, activation, optimizer, epoch, batchsize):
    model_data.update({
        "activation": activation,
        "optimizer": optimizer,
        "epoch": epoch,
        "batchsize": batchsize,
    })
    model, history = train()
    model_data["model"] = model  # keep the trained model
    history = pd.DataFrame(history.history)
    fig = px.line(history, title="Model training metrics")
    fig.update_layout(xaxis_title="epochs",
                      yaxis_title="metric value", legend_title="metrics")
    return fig


@app.callback(Output("progressdisplay", "children"),
              Input("trainprogress", "n_intervals"))
def update_progress(n):
```

```
    return json.dumps(train_status, indent=4)


app.clientside_callback(
    "function() { pageinit(); };",
    Output("dummy", "children"),
    Input("dummy", "children")
)


@server.route("/recognize", methods=["POST"])
def recognize():
    if not model_data.get("model"):
        return "Please train your model."
    matrix = json.loads(request.form["matrix"])
    matrix = np.asarray(matrix).reshape(1, 28, 28)
    proba = model_data["model"].predict(matrix).reshape(-1)
    result = np.argmax(proba)
    return "Digit "+str(result)


# run server, with hot-reloading
app.run_server(debug=True, threaded=True)
```

*Listing 30.25: Server side code for the Dash web app*

If we run all of these, we should see a screen like Figure 30.3.

## 30.6 Further Readings

There are a vast amount of web frameworks available, and Flask is just one of them. Another popular one is CherryPy. Below are resources on the topic if you are looking to go deeper.

### Books

Adam Schroeder, Christian Mayer, and Ann Marie Ward. *Python Dash: Build Stunning Data Analysis and Visualization Apps with Plotly*. No Starch Press, 2022.
https://www.amazon.com/dp/1718502222/

Elias Dabbas. *Interactive Dashboards and Data Apps with Plotly and Dash*. Packt Publishing, 2021.
https://www.amazon.com/dp/1800568916/

Miguel Grinberg. *Flask Web Development*. 2nd ed. O'Reilly, 2018.
https://www.amazon.com/dp/1491991739

Shalabh Aggarwal. *Flask Framework Cookbook*. 2nd ed.
https://www.amazon.com/dp/1789951291/

### Articles

*Web Frameworks*. Python.org wiki.
https://wiki.python.org/moin/WebFrameworks

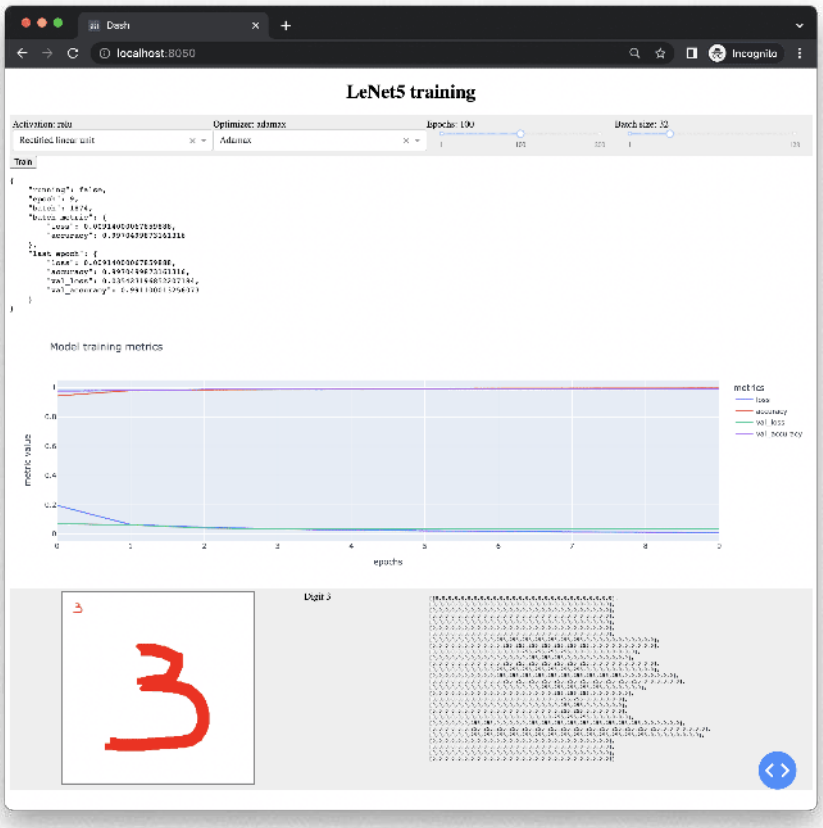



*Figure 30.3: Screenshot of the web app in action*

*Javascript*. MDN.
https://developer.mozilla.org/en-US/docs/Web/JavaScript

## Software and APIs

*CherryPy*.
https://cherrypydocrework.readthedocs.io/
*Django*.
https://www.djangoproject.com/
*Flask*.
https://flask.palletsprojects.com/en/2.1.x/
*Dash*.
https://dash.plotly.com/
*Plotly*.
https://plotly.com/
*Canvas API*. MDN.
https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API
*jQuery*.
https://jquery.com/

## 30.7 Summary

In this chapter, you learned how we can build a web app easily in Python with the Dash library. You also learned how we can create some web API using Flask. Specifically, you learned

- ▷ The mechanism of a web application
- ▷ How we can use Dash to build a simple web application triggered by web page components
- ▷ How can we use Flask to create a web API
- ▷ How a web application can be built in Javascript and run on a browser that uses the web API we built with Flask

In the next chapter, we will learn about some basic techniques to deploy our project to another computer.

# This is Just a Sample

Thank-you for your interest in **Python for Machine Learning**.

This is just a sample of the full text. You can purchase the complete book online from: https://machinelearningmastery.com/python-for-machine-learning/