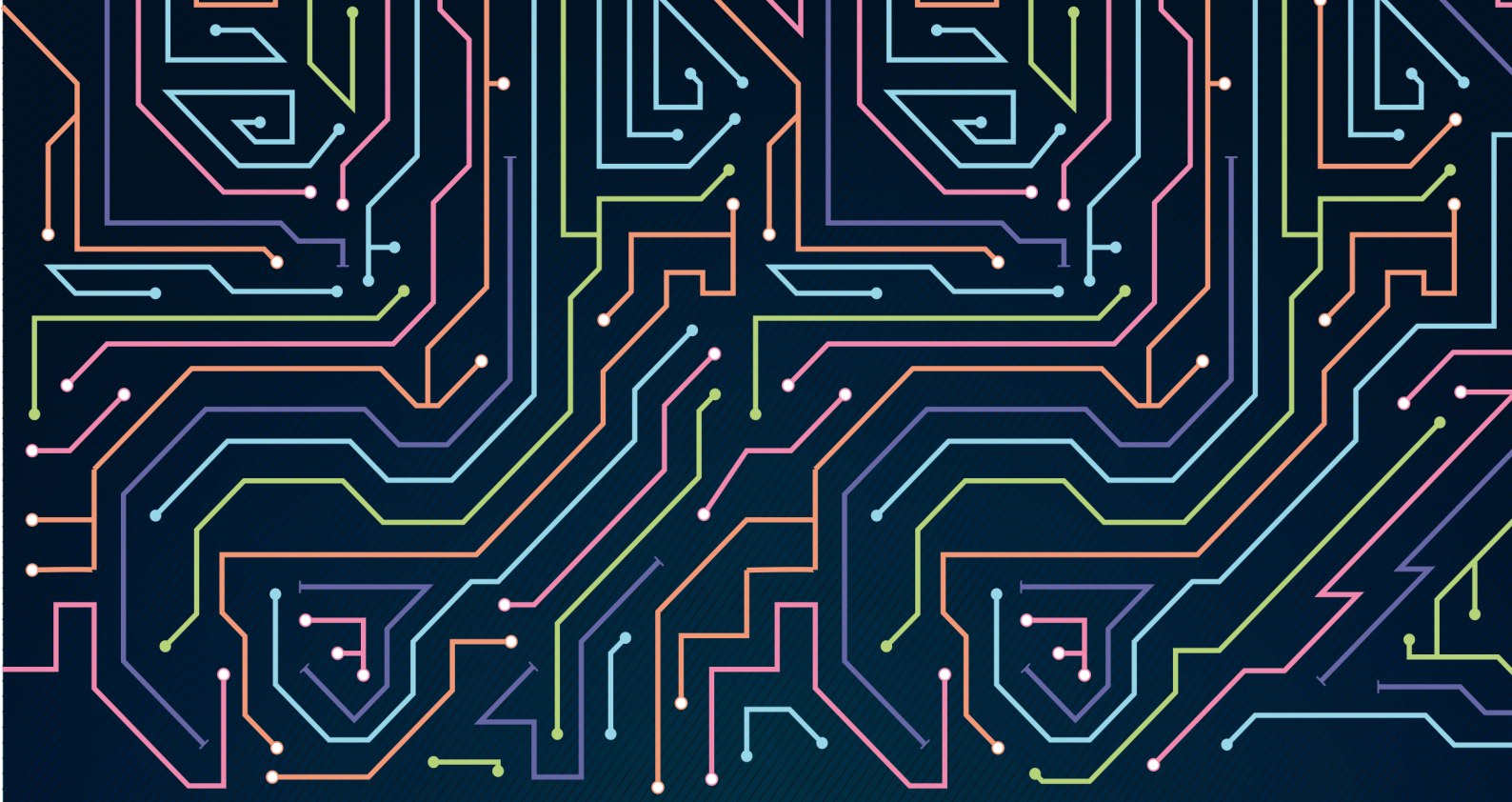

Computational Intelligence

EE40098 Laboratory

Dr. B. W. Metcalfe

Acknowledgements

Thanks to: **Tariq Rashid** for his excellent book ”Make Your Own Neural Network”, on which large aspects of this laboratory are based.

This version: September 30, 2019

Contents

Chapter 1

Introduction

1.1 Overall Objectives

In these laboratory sessions you will explore some of the computational intelligence techniques that have been covered in the lectures. These laboratories are self explanatory and should serve as an introduction to the tools that will be required in your coursework. In particular, this laboratory series will enable you to:

- Write, run, and debug Python using Visual Studio 2017
- Build a simple perception model of a neuron
- Assemble an artificial neural network using the perception model
- Implement back-propagation training for your artificial neural network
- Implement PCA and a *k*-nearest neighbour classification system

> **Important Message**
>
> This laboratory series builds on the knowledge developed within the lectures, you should make sure you are comfortable with the lecture content before attempting these laboratries. It does not assume any prior knowledge of Python and has a gentle introduction.

There are five scheduled laboratory slots and five corresponding activities within this script.

1.2 Assessment

The course will be assessed by exam and coursework. The coursework will comprise 25% of the overall unit mark and will be set and assessed in the second half of the course. Therefore laboratory attendance for these labs is **not** compulsory, however attendance will greatly improve your coursework performance, especially if you are new to Python. Each laboratory session is cumulative so it is important that you complete each exercise in order.

The coursework assessment is broken down into three different tasks:

1. Neural Networks MCQ (Formative, due in week 4)
2. Genetic Algorithms MCQ (Formative, due in Week 8)
3. **Coursework (25% of unit, due in Week 11)**

The coursework will be released to you during the semester, once the requisite material has been covered in the lectures.

Chapter 2

Lab One: Getting Started with Python

2.1 Objectives

In this laboratory session you will learn how to create, run, and debug a Python program from within Microsoft Visual Studio. Your objectives are:

1. Create a Python project using Visual Studio 2017
2. Run a basic Hello World example and become familiar with the editor
3. Write and test some simple programs that explore the key language concepts of Python
4. Learn how to use the debugger as a tool for software development

At the end of this laboratory session you should feel comfortable with creating, editing, debugging and running your own Python program.

2.2 Introduction

Home Computers

Python is free software, so if you want to work on your coursework at home then you can install Python on your own machines for free. Python is available for pretty much every operating system and platform, but there are two caveats to consider:

1. Python underwent a major version shift between Python 2.7 and Python 3.x, the differences were big enough that Python 3.x is backwards incompatible with Python 2.7. In this course we will be using **Python 3.6** and this is the version that you should install on your machine.
2. Python itself is just the core language, and throughout this course we will make use of many different libraries that extend the core functionality. You can install different libraries individually, but the easiest way is to install a packaged distribution of Python and the major libraries, I recommend using **Anaconda**.

Thus, the best way to install Python on your home computer is to grab Anaconda for Python 3.6 from:

`https://www.anaconda.com/download/`

You are free to use whatever IDE you wish, if you want to stick with Visual Studio then you also get it for free (the Community edition) from:

`https://www.visualstudio.com/`

2.3 Getting Started

The following assumes that you are working on a laboratory computer, i.e. with Visual Studio 2017 and Python 3.6.

Creating a Python Project in Visual Studio

Creating a Python project in Visual Studio is very similar to creating a C or C++ project. Firstly, launch Visual Studio 2017 from the Start Menu. Create a new project by selecting **File -> New -> Project**. In the list of Project Types select Python and then Python Application, see Figure 2.1 for clarification. Give your project a suitable name and save it within your H: drive. **Remember that good naming conventions are critically important - don't lose your coursework because you called the project temp2-catVideo.py**. Once you have set the name and location, click on OK.

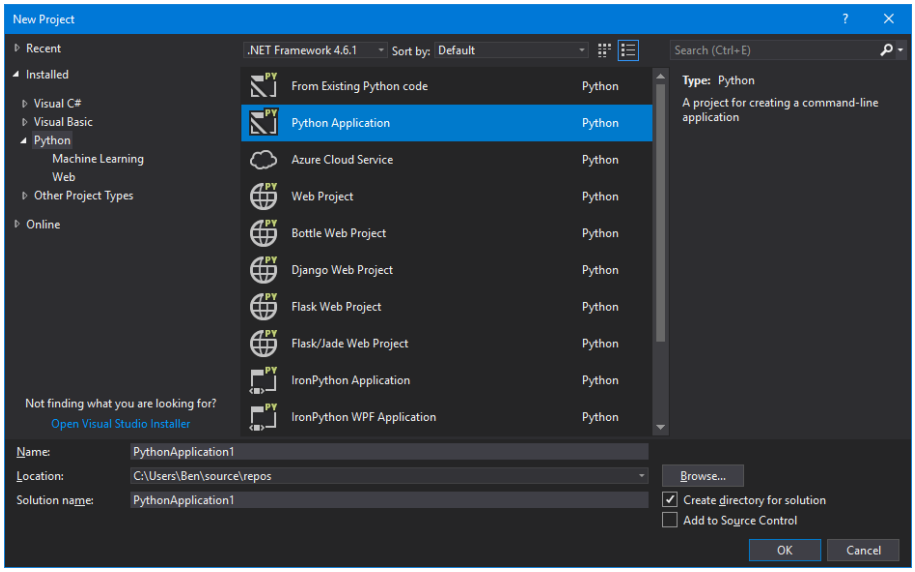

Figure 2.1: New project dialog with correct Python project selected

> **First Run**
>
> When you first run Visual Studio you may be asked to sign in, or to pick the colour scheme for your development environment. You do not need to sign in (click "Not now, maybe later" – obviously a lie), and you are free to choose whatever psychedelic colour scheme you desire.

Writing and Running Code

Once you have clicked on OK, Visual Studio will create the project for you, and is even kind enough to create and open a new source code file name **ProjectName.py**, note that the .py file extension is associated with Python source files, in much the same way a .c and .asm for C and Assembly respectively. It is now time to enter some code, and convention dictates that we announce our presence to the world by saying “Hello World”. Enter the following line of code into the editor:

Hello World

```
1 print("Hello World!")
```

Code 2.1: helloWorld.py

You will notice that this looks very much like the *printf* function from C, except that we didn't need to include an I/O library, or define a main function, or use a terminating semicolon – Python is designed to be *simple*.

Python Environment

In order to select the correct Python environment (Anaconda with Python 3.6), you will need to right click on "Python Environments" in the Solution Explorer on the right hand side of the development environment. Select "Add/Remove Python Environments" and associate Anaconda with the project by selecting the check box and clicking on "OK", as shown in Figure 2.2.

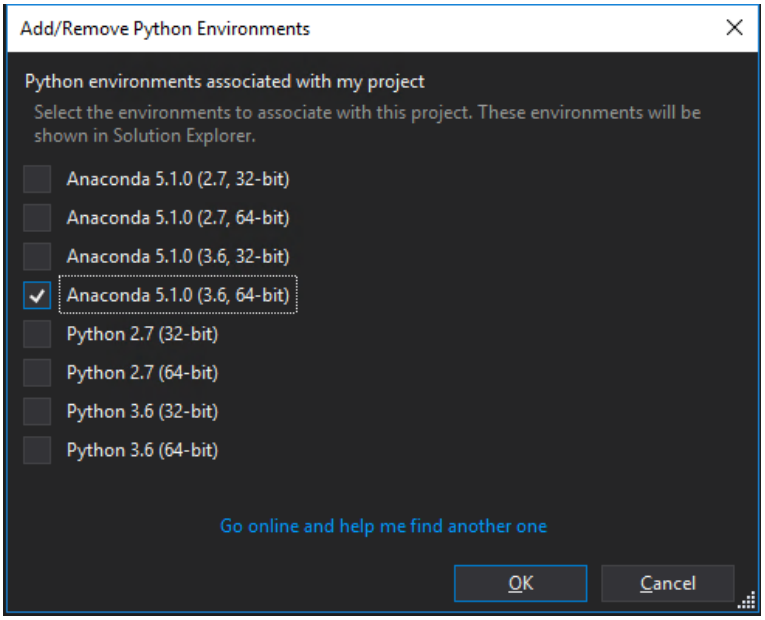

Figure 2.2: Setting Anaconda as the default Python environment for the session

Once you have written the code in the editor and setup the Python environment, save the file and hit F5 to run the program. (You can also select **Debug -> Start Debugging**, or click on the run arrow next to the word "Attach"). You should see a console window open with your message displayed, as in Figure 2.3. Note that unlike C, the Python process automatically waits for the user to terminate by pressing any key.

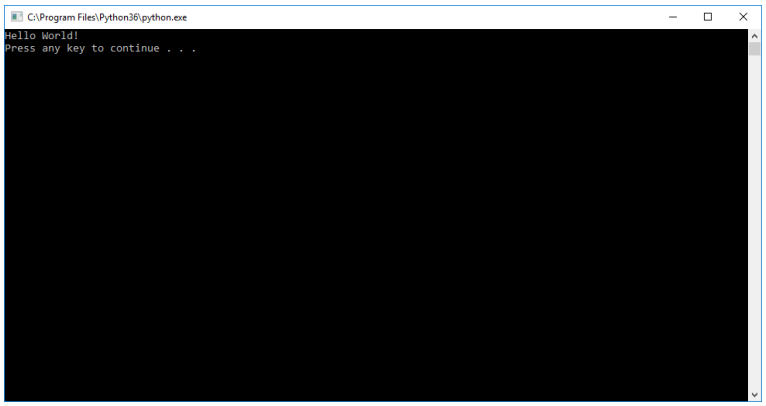

Figure 2.3: Simple Hello World example running in the console

Auto Insert

You will quickly notice that Visual Studio will auto-insert closing parenthesis or quotation marks for you, this can take a bit of getting used to but will make code writing much faster.

2.4 Some More Examples

In order to introduce you to some of the other Python concepts I have provided some short code examples that show you the core aspects of the languages. These are very simple, you do not need to try each one out, but you may find it useful to experiment with a few that you find more difficult. These examples can be used as a "cheat sheet" for more complicated programs. If you want to try them out you may either create new projects for each example or you could add them sequentially to your existing project and .py file. Remember that you can print a variable to the console by using the *print* command, if you want to see what the contents might be.

Most of these examples are taken from the very good Python tutorial, you can look there for more detailed notes:

https://docs.python.org/3/tutorial/

Variables and Mathematics

The first example to consider is how we store and manipulate variables (numbers, strings, etc). Python is very clever, and so unlike C it can work out what the type of your variable is from the contents – there is no need to explicitly type your variables. The code listing below shows some examples of creating and modifying some simple numerical variables.

Numerical

```
# This is a comment!

# Python knows that pi is a float
pi = 3.141

# and that freq is an integer
freq = 20000

# So nextFreq will also be an integer
nextFreq = freq + 1

# and radians/s will be a float, because we used a float in the calculation
rads = 2 * pi * nextFreq

# Classic division always returns a float
subRads = rads / 2.4

# Floor division returns the integer part (discards the fraction)
# Modulo division returns the fractional part (discards the integer part)
subRadsInt = rads // 2.4
remainder = rads % 2.4

# Powers can be calculated using **, here we square the frequency
f2 = freq ** 2
```

Code 2.2: numerical.py

Strings

```
# Strings can use either single or double quotes
firstName = "Bob"
lastName = 'Widlar'

# ..but be careful if you need to "contain" a quote mark, use \ to escape
# lastName = Mc'Gee
lastName = 'Mc\'Gee'

# String can be concatenated using +
fullName = firstName + " " + lastName

# Strings can be indexed, with the first character having index [0].
# There is no character type, a character is simply a string of length 1
firstLetter = fullName[0]

# You can also index from the right, so [-1] is the last character (-0 = 0)
lastLetter = fullName[-1]

# If you just want a slice, you can give a range
# Starting at 0 and ending at 2
fisrtName = fullName[0:3]

# You can't change modify a string, but you can over-write it or make a new one
# You can use len() to check the size..
nameLength = len(fullName)
```

Code 2.3: strings.py

Lists

```
# Lists are a way of storing lots of elements
squares = [1, 4, 9, 16, 25]

# Just like strings they can be indexed (from 0) and sliced
one = squares[0]
end = squares[-1]
# Note that the slice operation returns a new list containing the elements
squaresSlice = squares[0:3]

# Lists can be concatenated
squares = squares + [36, 49, 64, 81, 100]

# Unlike strings, lists are mutable. You can change the individual elements
squares[0] = 1 ** 2

# Len() also works on lists...
squaresLength = len(squares)
```

Code 2.4: lists.py

Loops and Flow Control

Loops and flow control are the next important area. You will have noticed by now that unlike C, Python does not require a semi-colon at the end of each line of code, or a set of curly brackets around code blocks. When want to group together bits of code, for example inside a loop, then we do so by indenting the code use a TAB space.

Loops

```
# Loops are a great way of repeating sections of code
# Can you work out what this does?
a = 0
b = 1
# Importantly, the contents of the loop is enclosed using indents (single tab)
# Not {} like you get in C, loops can be nested, you just keep indenting
while b < 10:
   print(b)
   a = b
   b = b + a

# We can also use for loops
# We need to generate a list of items to loop through
words = ['cat', 'window', 'defenestrate']
for word in words:
   print(word)

# If we just want to loop a set number of times, we can make a range of integers
# An easy way to this is using range()
# range(10) produces [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for i in range(10):
   print(i)

# We can perform a conditional check using the if statement
# the else and the elif (else if) are optional, note the indentation to group the code
if pi < 3:
   pi = 3.141
   print("Who ate the pie?")
elif pi > 4:
   print("Too much pie..")
else:
   pi = 0;
```

Code 2.5: loops.py

User Input

User Input

```
# We have already seen how to use print() to display data
# We can use input to get some data from the console
response = input("Is Python amazing? ")
print("You said: " + response)
# If we want to convert from string to int, or vice versa
limit = int(input("Enter a limit: "))
if limit > 3:
   # We could also have used "Your limit of", limit, "was too big"
   print("Your limit of " + str(limit) + " was too big")
```

Code 2.6: input.py

2.5 Debugging

One of the great features of Visual Studio is the excellent debugger, it will let us run our code one line at a time, and inspect the value of all of our variables. In order to run the code line by line we need to insert a *breakpoint*, this tells Python to run from the start of our code until this line and stop there. You can insert a breakpoint by **selecting a line of code -> right clicking -> breakpoint -> insert**, or by double clicking on the bar at the left hand side of the editor window. Once in place a red circle will indicate the breakpoint, and when you run your code it will now stop at this point.

If you now hover the cursor over a variable it will show you the current value of that variable, you can also use the watch window at the bottom of the screen (select Locals) to see the current value of all variables. Using the run options tool bar (as shown in Figure 2.4), you can either run to the next breakpoint, or single step the code one line at a time and watch as the variables change state.

Figure 2.4: The run options tool bar in debug mode

2.6 Exercises

Spend some time familiarising yourself with the Visual Studio environment and the debugger, it will prove a valuable asset in the rest of the course. If you want some more challenging exercises, see if you can produce software to perform the following functions:

1. Given a number entered by the user, compute and display the factorial
2. Store a secret password, ask the user to enter a password and compare the two (hint: strings are compared just like integers in C or MATLAB)

Chapter 3

Lab Two: The Perceptron

3.1 Objectives

In this laboratory session you will learn how to create a Perceptron model of the neuron, generate some test vectors, and evaluate the performance of the Perceptron as a classification tool. Your objectives are:

1. Learn about the use of functions and libraries in Python
2. Create a function that is based on the Perceptron model of the neuron
3. Evaluate this function as a binary (Boolean) classification tool
4. Learn how to perform basic plotting in Python to graphically visualise data

At the end of this laboratory session you should have the basic framework for a Perceptron model, you are free to modify this in any way you want for your future coursework.

3.2 Introduction

We will be the same tools as your previous laboratory (i.e Visual Studio & the Anaconda Python distribution). Follow the instructions from Laboratory 1 (Section 2) to create a new project, be sure to select Anaconda as the Python environment to run.

The Perceptron

The Perceptron was first described in 1957 by Frank Rosenblatt at Cornell. It is a simple type of linear classifier, that is to say the output depends on a linear summation of the inputs. When first introduced, a large number of Perceptions were connected together to form the first ever artificial neural network. In this laboratory we will start by considering a single Perceptron, whose output can be described by Equation 3.1:

$$f(x) = \begin{cases} 1 & \text{if } w.x + b > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

Where w is a vector of real-valued weights, $w.x$ is the dot product and b is a bias term that can be used to scale the effective threshold. We can begin a simple analysis by constraining the inputs to be Boolean (0 or 1), and we can set the weights and the bias by hand. A graphical description of the model is given in Figure 3.1 for N inputs, and a mathematical expansion of Equation 3.1 is given in Equation 3.2, where θ represents the threshold function.

$$f(x) = \theta\left(\sum_{i=1}^{N} w_i x_i + b\right) \tag{3.2}$$

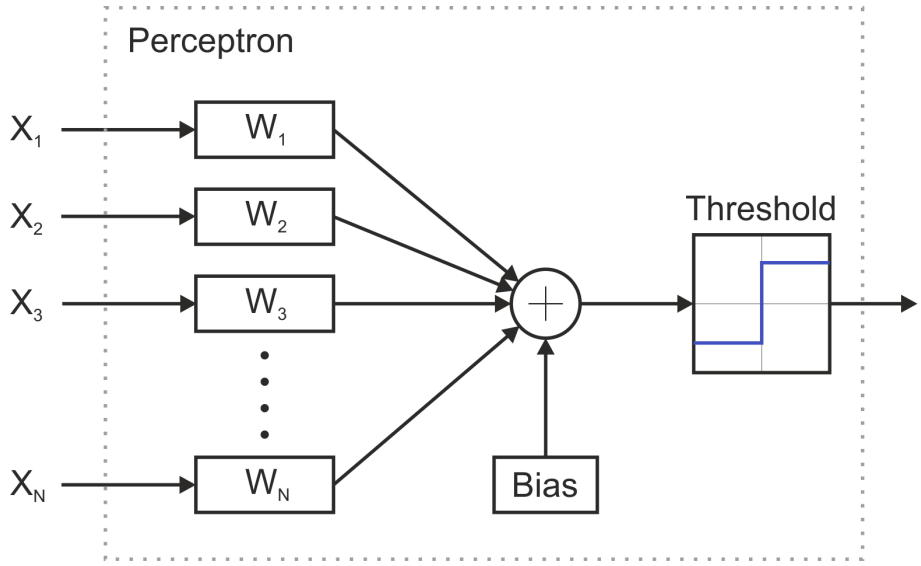

Figure 3.1: A single Perceptron with N inputs X_N, the inputs are multiplied by weights W_N and summed with the bias before a simple hard threshold.

Python Libraries

In order to extend the capabilities of Python we are going to use some of the libraries that are provided by the Anaconda distribution (these libraries are actually written by the open source community, Anaconda is just a package that installs the most popular libraries. There are two libraries that we will make use of in this laboratory, NumPy and matplotlib.

> **First Run**
>
> On the laboratory computers I have noticed that the first time you run a program that imports a library there can be quite a long delay before the program starts (around 6 seconds). This problem seems to be less noticeable on successive runs.

NumPy is a library that adds support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these matrices. In many ways, NumPy adds a matrix feature set that resembles the capabilities of MATLAB, for example the dot product in Equation 3.1 can be performed in a single line of code, rather than writing a long for loop. This makes working with arrays and matrices much easier.

matplotlib is a plotting library that provides a good interface for drawing graphs and charts. As the name suggests it has been influenced by the plotting engine in MATLAB, as so the syntax is broadly similar and should be familiar to most of you.

You do not need to understand the intricacies of these libraries, fully worked examples will be given, however for your coursework you may choose to explore some of the capabilities in more detail.

3.3 Getting Started

If you have not already done so, create a new Python project using the instructions given in Section 2.

Creating the Perceptron

Code listing 3.1 shows a simple program for simulating the Perceptron. Read through this code and type it into your own program. Library imports occur at the start of the program, followed by function definitions and then the main code. Note that in Python, functions work just like they do in MATLAB, here our Perceptron function takes three inputs and returns a single output. In the main code section, near the bottom, we create two lists that represent our inputs X_N and our weights W_N. Within the Perceptron function, we convert those lists into NumPy arrays. The reason for this is shown in line 14, using the NumPy array type we can compute the dot product in a single line, rather than using a for loop.

Perceptron

```
1  # Import the NumPy library for matrix math
2  import numpy
3
4
5  # A single perceptron function
6  def perceptron(inputs_list, weights_list, bias):
7      # Convert the inputs list into a numpy array
8      inputs = numpy.array(inputs_list)
9
10     # Convert the weights list into a numpy array
11     weights = numpy.array(weights_list)
12
13     # Calculate the dot product
14     summed = numpy.dot(inputs, weights)
15
16     # Add in the bias
17     summed = summed + bias
18
19     # Calculate output
20     # N.B this is a ternary operator, neat huh?
21     output = 1 if summed > 0 else 0
22
23     return output
24
25 # Our main code starts here
26
27 # Test the perceptron
28 inputs = [1.0, 0.0]
29 weights = [1.0, 1.0]
30 bias = -1
31
32 print("Inputs:  ", inputs)
33 print("Weights: ", weights)
34 print("Bias:    ", bias)
35 print("Result:  ", perceptron(inputs, weights, bias))
```

Code 3.1: perceptron.py

3.4 Exercises

Spend some time familiarising yourself with the basic Perceptron model. Once you have the model running, you should try the following exercises:

1. For a two-input perceptron with $W = [1.0, 1.0]$ and $b = -1$, simulate the output for all possible Boolean input combinations where $X \in \{0, 1\}$.
2. Interpreting this information as a truth table, what logical function is being performed?
3. What are the weight vectors and the bias for the logical functions: AND, OR, NAND, NOR and XOR?
4. Why was 3. a trick question?

Further Exercises

Once you are happy with the basic Perceptron model, you can extend it using the matplotlib library to give you a graphical interpretation of the data. Code listing 3.2 shows a basic program that plots a scatter diagram of the input vector state space (every possible value of every input) for our two input Boolean Perceptron. Here are a few features you could add:

1. Modify your Perceptron code so that it automatically tests all possible input vectors (i.e [0, 0], [0, 1],...) *Hint: store the input vectors in a list, and use a for loop to test each input vector.*
2. With the aid of the example plotting code in listing 3.2, modify your code so that it automatically generates a plot of the input vector state space - use colour coding to signify if the Perceptron fired (output = 1) for that input.
3. Calculate the linear separator for the Perceptron and add it to the plot, an example of the final result is shown in Figure 3.2.

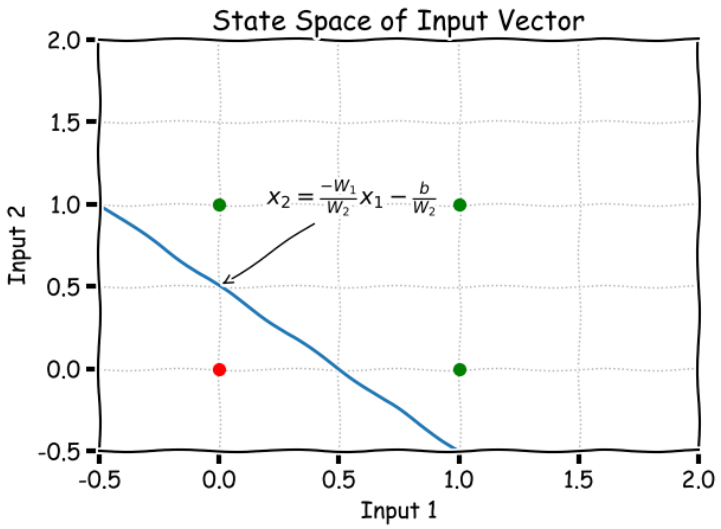

Figure 3.2: Example state space diagram with overlaid linear separator.

Plotting

```
# Import the matplotlib pyplot library
# It has a very long name, so import it as the name plt
import matplotlib.pyplot as plt

# Make a new plot (XKCD style)
fig = plt.xkcd()

# Add points as scatters - scatter(x, y, size, color)
# zorder determines the drawing order, set to 3 to make the
# grid lines appear behind the scatter points
plt.scatter(0, 0, s=50, color="red", zorder=3)
plt.scatter(0, 1, s=50, color="red", zorder=3)
plt.scatter(1, 0, s=50, color="red", zorder=3)
plt.scatter(1, 1, s=50, color="green", zorder=3)

# Set the axis limits
plt.xlim(-2, 2)
plt.ylim(-2, 2)

# Label the plot
plt.xlabel("Input 1")
plt.ylabel("Input 2")
plt.title("State Space of Input Vector")

# Turn on grid lines
plt.grid(True, linewidth=1, linestyle=':')

# Autosize (stops the labels getting cut off)
plt.tight_layout()

# Show the plot
plt.show()
```

Code 3.2: plotting.py

Chapter 4

Lab Three: The MLP

4.1 Objectives

In this laboratory session you will create a general purpose two-layer feed-forward artificial neural network. This neural network can solve many different classification problems, and it will extend the basic Perceptron from the previous laboratory. Your objectives are:

1. Learn about the use of classes and objects in Python
2. Create a neural network class that permits for an arbitrary number of neurons
3. Train the network using backpropagation
4. Explore the effect of the network size on training and performance

At the end of this laboratory session you should have a two-layer neural network, i.e a network with two layers of neurons, you are free to modify or extend this in any way you want for your future coursework.

4.2 Introduction

We will be using the same tools as your previous laboratory (i.e Visual Studio & the Anaconda Python distribution). Follow the instructions from Section 2 to create a new project, be sure to select Anaconda as the Python environment to run.

Classes and Objects

One of the more powerful aspects of modern languages, such as Python, is the concept of classes and objects (these are also found in C++, which extended the C programming language). The idea behind an object is to group together data and functions that pertain to one ”thing”. The easiest way to understand objects is to see them in action, consider the code listing 4.1. In this example we define a class (a class is the template that objects are based on). Our class defines a dog object, and that dog object has a number of functions. There is a special function called __init__, this is the function that is run when we make a new dog, it creates some variables that are specific to that individual dog object (using the self keyword). The class also defines some functions that the dog can have, these include a function called bark, which is arguably the most vital of all functions if you are a dog.

The real power behind classes is the ability to create objects, every time we make a new object (a Dog), we are creating a copy of all the functions and variables that are defined by the class. Imagine if the class contained a neural network rather than a dog, then we would be able to easily create multiple neural networks and run them all at the same time, even though we only wrote the neural network code once. Figure 4.1 shows an example of what happens when we create a new object based on the class, you can think of the objects as *instances* and the classes as *templates*.

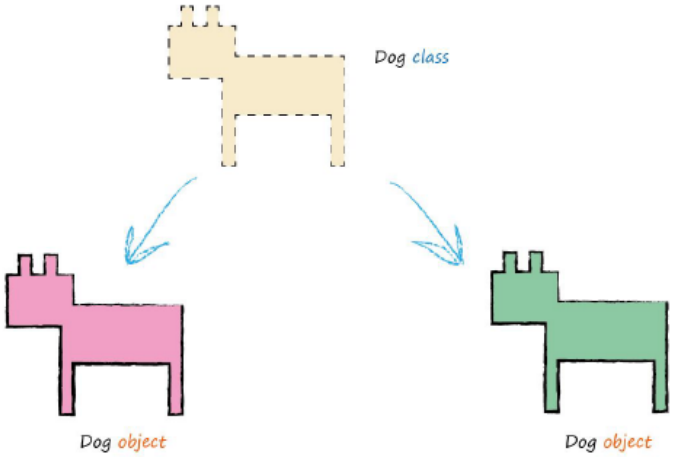

Figure 4.1: Example of two Dog objects being created based on the Dog class, which can be thought of as a template.

Dog Class

```
# Class template for a dog object
class Dog:
    # Initialisation method, gets run whenever we create a new Dog
    # The self just allows this function to reference variables relevant to this particular Dog
    def __init__(self, name, hungerLevel):
        self.name = name
        self.hungerLevel = hungerLevel

    # Query the status of the Dog
    def status(self):
        print("Dog is called ", self.name)
        print("Dog hunger level is ", self.hungerLevel)
        pass

    # Set the hunger level of the dog
    def setHungerLevel(self, hungerLevel):
        self.hungerLevel = hungerLevel
        pass

    # Dogs can bark
    def bark(self):
        print("Woof!")
        pass

# Create two dog objects
# Note that we don't need to include the self from the parameter list
lassie = Dog("Lassie", "Mild")
yoda = Dog("Yoda", "Ravenous")

# Check on Yoda & Lassie
yoda.status()
lassie.status()

# Get Lassie to bark
lassie.bark()

# Feed Yoda
yoda.setHungerLevel("Full")
yoda.status()
```

Code 4.1: dogClass.py

4.3 Getting Started

If you have not already done so, create a new Python project using the instructions given in Section 2. Type in the example class code that defines the Dog class, and make sure you can run the code. Read through the comments and make sure you understand the principles at work before you continue with the neural network example.

Creating the Neural Network Class

In code listing 4.2 I have provided you with a complete neural network class that implements a two-layer feed-forward perceptron neural network. The number of inputs, the number of neurons, and the number of outputs is all definable when you create an instance (an object) of the neural network. For example, a neural network object with two inputs, two hidden neurons and one output neuron would look like that of Figure 4.2. This is approximately the configuration we decided was required (as a minimum) to solve the XOR problem.

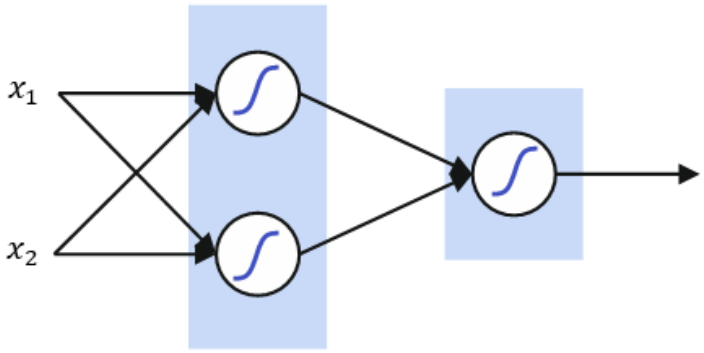

Figure 4.2: Example two-layer feed-forward neural network with sigmoid activation functions.

The neural network has three functions, which are documented in Table 4.1. The basic pattern of use should be to create an instance of the network, to train the network using an inputs list and a target list, and to then query the network with some test data to validate the training. The default activation function is the logistic sigmoid (where $\theta \in [0, 1]$) and there are no biases to the neurons.

Table 4.1: Functions of the NeuralNetwork Class

Function	Role
N = NeuralNetwork(self, input_nodes, hidden_nodes, output_nodes, learning_rate)	Creates a new neural network object with the number of inputs, hidden neurons and output neurons specified. An initial learning rate is also given and should be in the range [0, 1].
Train(self, inputs_list, targets_list)	Performs one iteration of backpropagation training given a list of inputs and a list of targets. Note this does not work on lists of lists.
Out = Query(inputs_list)	Run the network with the given inputs.

ANN

```
# Import scipy.special for the sigmoid function expit()
import scipy.special, numpy

# Neural network class definition
class NeuralNetwork:
   # Init the network, this gets run whenever we make a new instance of this class
   def __init__(self, input_nodes, hidden_nodes, output_nodes, learning_rate):
      # Set the number of nodes in each input, hidden and output layer
      self.i_nodes = input_nodes
      self.h_nodes = hidden_nodes
      self.o_nodes = output_nodes

      # Weight matrices, wih (input -> hidden) and who (hidden -> output)
      self.wih = numpy.random.normal(0.0, pow(self.h_nodes, -0.5), (self.h_nodes, self.i_nodes))
      self.who = numpy.random.normal(0.0, pow(self.o_nodes, -0.5), (self.o_nodes, self.h_nodes))

      # Set the learning rate
      self.lr = learning_rate

      # Set the activation function, the logistic sigmoid
      self.activation_function = lambda x: scipy.special.expit(x)

   # Train the network using back-propagation of errors
   def train(self, inputs_list, targets_list):
      # Convert inputs into 2D arrays
      inputs_array = numpy.array(inputs_list, ndmin=2).T
      targets_array = numpy.array(targets_list, ndmin=2).T

      # Calculate signals into hidden layer
      hidden_inputs = numpy.dot(self.wih, inputs_array)

      # Calculate the signals emerging from hidden layer
      hidden_outputs = self.activation_function(hidden_inputs)

      # Calculate signals into final output layer
      final_inputs = numpy.dot(self.who, hidden_outputs)

      # Calculate the signals emerging from final output layer
      final_outputs = self.activation_function(final_inputs)

      # Current error is (target - actual)
      output_errors = targets_array - final_outputs

      # Hidden layer errors are the output errors, split by the weights, recombined at hidden nodes
      hidden_errors = numpy.dot(self.who.T, output_errors)

      # Update the weights for the links between the hidden and output layers
      self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0 - final_outputs)),
      numpy.transpose(hidden_outputs))

      # Update the weights for the links between the input and hidden layers
      self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs * (1.0 - hidden_outputs)),
      numpy.transpose(inputs_array))

   # Query the network
   def query(self, inputs_list):
      # Convert the inputs list into a 2D array
      inputs_array = numpy.array(inputs_list, ndmin=2).T

      # Calculate signals into hidden layer
      hidden_inputs = numpy.dot(self.wih, inputs_array)

      # Calculate output from the hidden layer
      hidden_outputs = self.activation_function(hidden_inputs)

      # Calculate signals into final layer
      final_inputs = numpy.dot(self.who, hidden_outputs)

      # Calculate outputs from the final layer
      final_outputs = self.activation_function(final_inputs)

      return final_outputs
```

Code 4.2: ANN.py

4.4 Exercises

Spend some time familiarising yourself with the neural network model. You do not need to fully understand the internal workings, but you should understand the role of the three functions. Here are some exercises to try:

1. Create a neural network instance with two inputs, two hidden neurons and one output neuron. Create some test inputs and query the network, how do you explain the networks output?
2. Create a set of training vectors for all possible inputs, i.e [[0,0], [0,1], [1,0], [1,1]], and a target vector for one of the logical functions such as AND.
3. Train the network once for each target, then query the network for all possible inputs. Has your network successfully learned the AND function? If not, then why not?
4. Extend your training routine to repeat the training process, how many iterations does it take for the network to learn the AND function with two inputs?
5. Experiment with the number of neurons in the hidden layer, and the learning rate, what effect do these have on training the AND function?
6. If you have time, try other functions, such as XOR. What effect do the number of neurons in the hidden layer have and why?
7. How do these effects scale if you create, for example, a 4 input logical function?

Chapter 5

Lab Four: Handwriting Recognition

5.1 Objectives

In this laboratory session you will use the general purpose two-layer MLP from laboratory two to perform automatic recognition of human handwriting. Specifically, you will devise and train an MLP to recognise digits from the MNIST database. Your objectives are:

1. Learn about the MNIST database and the format of the images
2. Learn how to read files in Python and display the MNIST images
3. Implement a two-layer MLP to perform classification on the dataset
4. Train the network, and evaluate the performance against a test set
5. Tweak the network parameters to try and improve the performance

At the end of this laboratory session you should have a practical neural network that can solve quite a complex task, this is the starting point for practical pattern recognition. You are free to modify or extend this in any way you want for your future coursework.

5.2 Introduction

We will be using the same tools as your previous laboratory (i.e Visual Studio & the Anaconda Python distribution). Follow the instructions from Section 2 to create a new project, be sure to select Anaconda as the Python environment to run.

MNIST

The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of machine learning. Recognising human handwriting is an ideal test for neural networks, because the problem is well defined but relatively difficult. The MNIST database contains 60,000 training images and 10,000 testing images. There have been a number of scientific papers on attempts to achieve the lowest error rate; one paper, using a hierarchical system of convolutional neural networks, manages to get an error rate on the MNIST database of 0.23%.

> **MNIST Files**
>
> The MNIST database in CSV format is available to download from Moodle. There are four files, the full test and training sets, and the smaller subsets of 100 training and 10 test sets.

Reading Files

The images are stored in Comma Separated Value (CSV) file format, which is easily readable in Python. You do not need to worry too much about how the data is represented, essentially the first value is the label (that is the actual number represented) and the subsequent values are a 28 x 28 array of pixels (giving a total of 784). To begin with, you should work on the smaller subset of MNIST, which provides you with 100 training examples and 10 test examples. In code listing 5.1, I have given you an example of how to open the smaller training set, read the first image, and display it using matplotlib. Figure 5.1 shows the resulting plot, in this case the first sample in the training set is the number 5. You should run this example and take a look at the first few images from both of the smaller training and test sets.

DrawNumber

```
# Import numpy for arrays and matplotlib for drawing the numbers
import numpy
import matplotlib.pyplot as plt

# Open the 100 training samples in read mode
data_file = open("mnist_train_100.csv", 'r')
# Read all of the lines from the file into memory
data_list = data_file.readlines()
# Close the file (we are done with it)
data_file.close()

# Take the first line (data_list index 0, the first sample), and split it up based on the commas
# all_values now contains a list of [label, pixel 1, pixel 2, pixel 3, ... ,pixel 784]
all_values = data_list[0].split(',')

# Take the long list of pixels (but not the label), and reshape them to a 2D array of pixels
image_array = numpy.asfarray(all_values[1:]).reshape((28, 28))

# Plot this 2D array as an image, use the grey colour map and don't interpolate
plt.imshow(image_array, cmap='Greys', interpolation='None')
plt.show()
```

Code 5.1: drawNumber.py

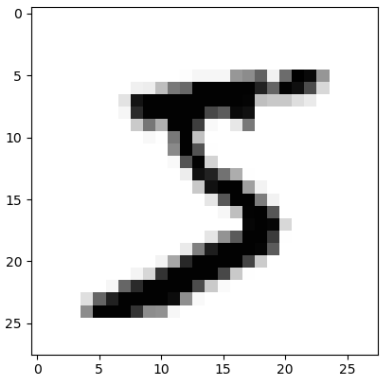

Figure 5.1: Greyscale plot of the first example from the training set of 100 numbers (in this case, the number 5).

Table 5.1: Suggested parameters for two-layer MNIST MLP.

Parameter	Suggested Value
Input Nodes	784
Hidden Nodes	100
Output Nodes	10
Learning Rate	0.3

5.3 Getting Started

If you have not already done so, create a new Python project using the instructions given in Section 2. Type in the example code that plots the handwritten numbers, and make sure you can run the code. Read through the comments and make sure you understand the principles at work before you continue with the neural network example.

Configuring the Network

Before we can jump in and start using our neural network to classify the images, we need to think about the topology of network that we want to use. We will also need to think about the format of the input and output data. We know that the input will be a vector of 784 pixels, so it is fairly easy to set the number of input nodes. The output needs to tell us what number the network thinks was in the image, the easiest way to do this is to have 10 output nodes, each one corresponding to the probability that the image contained that number. So for example a value of 0.99 at output node 0 would mean that the network was certain that the image contained the number 0. The number of hidden nodes and the learning rate are, at this stage, best guess. I have put some starting values in Table 5.1.

Training and Testing the Network

In Code Listing 5.2, is the code to train the network using the 100 sample training set (note that in the code the neural network is called n). There are two important aspects to note. Firstly, the pixels in each image are in the range 0..255, I have rescaled them to be in the range 0.01..1. It is often helpful to have non-zero inputs, as a zero input can cause the training process to become stuck (as changing to associated weight has no effect on the output). The second key aspect is that I create a target vector with 0.01 indicating "not this digit" and 0.99 indicating "this digit". In the previous labs you should have observed that the networks output never reaches 1 or 0 exactly, this is because the logistic sigmoid function is *asymptotic* - if we set targets of 1 or 0 then the training process becomes saturated at the asymptote. This is why you may have noticed that your MLP performed worse at the simple logic functions than you hand crafted Perceptron, which used the hard threshold function. In Code Listing 5.3, I have provided example code that tests the network using the 10 sample test set. A scorecard is created, with a 1 appended for each correct classification, and a 0 for an incorrect classification. Finally, the performance score is calculated as the percentage of correct classifications.

Training

```
# Load the MNIST 100 training samples CSV file into a list
training_data_file = open("mnist_train_100.csv", 'r')
training_data_list = training_data_file.readlines()
training_data_file.close()

# Train the neural network on each training sample
for record in training_data_list:
   # Split the record by the commas
   all_values = record.split(',')
   # Scale and shift the inputs from 0..255 to 0.01..1
   inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01
   # Create the target output values (all 0.01, except the desired label which is 0.99)
   targets = numpy.zeros(output_nodes) + 0.01
   # All_values[0] is the target label for this record
   targets[int(all_values[0])] = 0.99
   # Train the network
   n.train(inputs, targets)
pass
```

Code 5.2: mnistTrain.py

Testing

```
# Load the MNIST test samples CSV file into a list
test_data_file = open("mnist_test_10.csv", 'r')
test_data_list = test_data_file.readlines()
test_data_file.close()

# Scorecard list for how well the network performs, initially empty
scorecard = []

# Loop through all of the records in the test data set
for record in test_data_list:
   # Split the record by the commas
   all_values = record.split(',')
   # The correct label is the first value
   correct_label = int(all_values[0])
   print(correct_label, "Correct label")
   # Scale and shift the inputs
   inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01
   # Query the network
   outputs = n.query(inputs)
   # The index of the highest value output corresponds to the label
   label = numpy.argmax(outputs)
   print(label, "Network label")
   # Append either a 1 or a 0 to the scorecard list
   if (label == correct_label):
      scorecard.append(1)
   else:
      scorecard.append(0)
      pass
   pass

# Calculate the performance score, the fraction of correct answers
scorecard_array = numpy.asarray(scorecard)
print("Performance = ", (scorecard_array.sum() / scorecard_array.size)*100, '%')
```

Code 5.3: mnistTest.py

5.4 Exercises

Spend some time familiarising yourself with the code for training and testing the network with the MNIST dataset. You do not need to fully understand the internal workings, but you should understand the basic flow of the code. Here are some exercises to try:

1. Using the template network class from Section 4, create a neural network with the parameters given in Table 5.1 **Note: you will need to create variables for the network parameters such as output nodes**.
2. Train this network using the MNIST100 training set, and then test it using the MNIST10 testing set. What is your networks performance? Mine was 60%, but there will be small variances due to the random initial weights.
3. Display the images that were incorrectly classified, do you think that you (as a human), could have got those classifications correct?
4. Modify the network topology, the learning rate and any other parameters you desire – what is the highest network performance you can get?
5. Re-run the process using the full size MNIST training and testing sets. **This may take some time**. What is the performance now? Why is it different than before?

Chapter 6

Lab Five: PCA & KNN

6.1 Objectives

In this laboratory session you will use Principal Component Analysis (PCA) and the k-nearest neighbour (KNN) algorithms to perform automatic recognition of human handwriting. Specifically, you will use PCA to automatically extract a set of features, and KNN to classify digits from the MNIST dataset using the features. Your objectives are:

1. Load the MNIST images, and learn about the CI tools within the scikit-learn package
2. Perform PCA on the training set and evaluate the variance distribution
3. Use the principal components from objective 2 to create a KNN classifier
4. Extract the same principal components from the test data set, and classify them using the KNN classifier
5. Evaluate the overall performance and speed, and experiment with the different parameters

At the end of this laboratory session you should have a working handwriting recognition system that you can directly compare to the neural network system in the previous labs. You will also have a working demonstration of both PCA and KNN. You are free to modify or extend this in any way you want for your future coursework.

6.2 Introduction

We will be using the same tools as your previous laboratory (i.e Visual Studio & the Anaconda Python distribution). Follow the instructions from Laboratory 2 to create a new project, be sure to select Anaconda as the Python environment to run.

MNIST

We will be using the same MNIST database as Laboratory 4, as a reminder MNIST is a large database of handwritten digits that is commonly used for training various image processing systems. The MNIST database contains 60,000 training images and 10,000 testing images. There have been many scientific papers on attempts to achieve the lowest error rate; one paper, using a hierarchical system of convolutional neural networks, manages to get an error rate on the MNIST database of 0.23%.

> **MNIST Files**
>
> The MNIST database in CSV format is available to download from Moodle. There are four files, the full test and training sets, and the smaller subsets of 100 training and 10 test sets.

Scikit-learn

In the laboratories so far you have created the majority of the functional code yourselves, making use of libraries such as numpy and matplotlib to perform useful functions. This has been a worthwhile exercise because it enables a far greater level of understanding than simply using someone else's code. However, Computational Intelligence and machine learning, are very popular areas of research, and there are many Python libraries that provide you with working examples of structures such as neural networks. In this laboratory, we will be using the PCA algorithm and the KNN classifier. We could write the code for these from scratch, but it would become very long and quite tedious. Instead we will make use of a very popular Python library called scikit-learn. Scikit-learn is a machine learning library that includes various classification, regression and clustering algorithms including PCA and KNN. There is plenty of documentation available on the scikit-learn website if you wish to explore the algorithms further.

6.3 Getting Started

If you have not already done so, create a new Python project using the instructions given in Section 2.

Principal Component Analysis

Our first step is to look at the code that reads in the MNIST data and performs PCA. In code listing 6.1 the two CSV files are loaded, making use of a numpy file read command. Next the data and labels are identified and a PCA object is initiated with a parameter that sets the number of components to extract. The training data is then fed into the PCA object, once the principal components have been extracted it is possible to plot the variance that is represented by the components (if the components represented all the variance, then the sum of the variances would be 1).

PCA

```
# Numpy for useful maths
import numpy
# Sklearn contains some useful CI tools
# PCA
from sklearn.decomposition import PCA
from sklearn.preprocessing import MinMaxScaler
# k Nearest Neighbour
from sklearn.neighbors import KNeighborsClassifier
# Matplotlib for plotting
import matplotlib.pyplot as plt

# Load the train and test MNIST data
train = numpy.loadtxt('mnist_train_100.csv', delimiter=',')
test = numpy.loadtxt('mnist_test_10.csv', delimiter=',')

# Separate labels from training data
train_data = train[:, 1:]
train_labels = train[:, 0]
test_data = test[:, 1:]
```

```
test_labels = test[:, 0]

# Select number of components to extract
pca = PCA(n_components = 10)
# Fit to the training data
pca.fit(train_data)

# Determine amount of variance explained by components
print("Total Variance Explained: ", numpy.sum(pca.explained_variance_ratio_))

# Plot the explained variance
plt.plot(pca.explained_variance_ratio_)
plt.title('Variance Explained by Extracted Componenents')
plt.ylabel('Variance')
plt.xlabel('Principal Components')
plt.show()
```

Code 6.1: PCA.py

k-Nearest Neighbour

Once we have setup our PCA, we can move on the classification stage. Code listing 6.2 shows the next steps, the training and test data are both transformed using PCA into a list of principal component scores, these scores are then normalised. Next, we create a KNN classifier with the training principal component scores and labels as the training data, note that the KNN is initiated with $k = 5$ and the $p = 2$ (Euclidean) norm. Finally, we can now use the KNN object to predict the classification for the test data. As with the previous laboratories a scorecard is created, and the percentage success rate is calculated.

KNN

```
# Extract the principle components from the training data
train_ext = pca.fit_transform(train_data)
# Transform the test data using the same components
test_ext = pca.transform(test_data)

# Normalise the data sets
min_max_scaler = MinMaxScaler()
train_norm = min_max_scaler.fit_transform(train_ext)
test_norm = min_max_scaler.fit_transform(test_ext)

# Create a KNN classification system with k = 5
# Uses the p2 (Euclidean) norm
knn = KNeighborsClassifier(n_neighbors=5, p=2)
knn.fit(train_norm, train_labels)

# Feed the test data in the classifier to get the predictions
pred = knn.predict(test_norm)

# Check how many were correct
scorecard = []

for i, sample in enumerate(test_data):
   # Check if the KNN classification was correct
   if round(pred[i]) == test_labels[i]:
      scorecard.append(1)
   else:
      scorecard.append(0)
   pass

# Calculate the performance score, the fraction of correct answers
scorecard_array = numpy.asarray(scorecard)
print("Performance = ", (scorecard_array.sum() / scorecard_array.size) * 100, ' % ')
```

Code 6.2: KNN.py

6.4 Exercises

Spend some time familiarising yourself with the code for extracting the principal component scores and the KNN classifier. You do not need to fully understand the internal workings, but you should understand the basic flow of the code. Here are some exercises to try:

1. Try increasing the number of principal components, how many are needed to explain all the variance within the training data?
2. How many principal components do you think is enough to perform effective classification?
3. What is the best performance you can get on the MNIST100/10 data set?
4. What happens if you change the parameters k and p?
5. How does your best performance compare to the neural network from laboratory four?
6. What can you say about the time taken to run this type of classifier?
7. Using the code you have been provided, use PCA to reduce the number of features and then use a neural network to classify the MNIST characters. Does this give you the best result?
8. Repeat these tests on the full MNIST dataset (60,000 training and 10,000 test).