Week 3

The MLP

1.1. 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.

1.2. Introduction

We will be the same tools as the previous laboratory (i.e Visual Studio Code & the Anaconda Python distribution). Follow the instructions from Laboratory 1 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 in 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 several 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*.

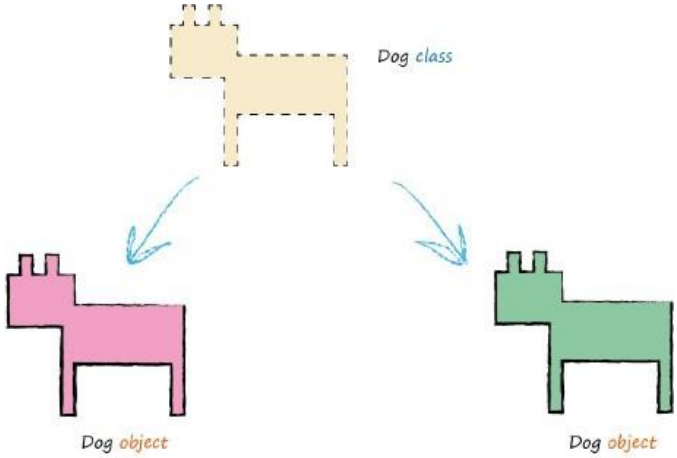

Figure 4.1: Example of two Dog objects being created based on the Dog class, which can be thought of as a template.

Dog Class

```
1   # Class template for a dog object
2   class Dog:
3       # Initialisation method, gets run whenever we create a new Dog
4       # The self just allows this function to reference variables relevant to this particular Dog
5       def__init__(self, name, hungerLevel):
6           self.name = name
7           self.hungerLevel = hungerLevel
8
9       # Query the status of the Dog
10      def status(self):
11          print("Dog is called ", self.name)
12          print("Dog hunger level is ", self.hungerLevel)
13          pass
14
15      # Set the hunger level of the dog
16      def setHungerLevel(self, hungerLevel):
17          self.hungerLevel = hungerLevel
18          pass
19
20      # Dogs can bark
21      def bark(self):
22          print("Woof!")
23          pass
24
25  # Create two dog objects
26  # Note that we don't need to include the self from the parameter list
27  lassie = Dog("Lassie", "Mild")
28  yoda = Dog("Yoda", "Ravenous")
29
30  # Check on Yoda & Lassie
31  yoda.status()
32  lassie.status()
33
34  # Get Lassie to bark
35  lassie.bark()
36
37  # Feed Yoda
38  yoda.setHungerLevel("Full")
39  yoda.status()
```

Code 4.1: dogClass.py

1.3. Getting Started

If you have not already done so, create a new Python project using the instructions given in the lab from week 1. 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 in- stance (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.

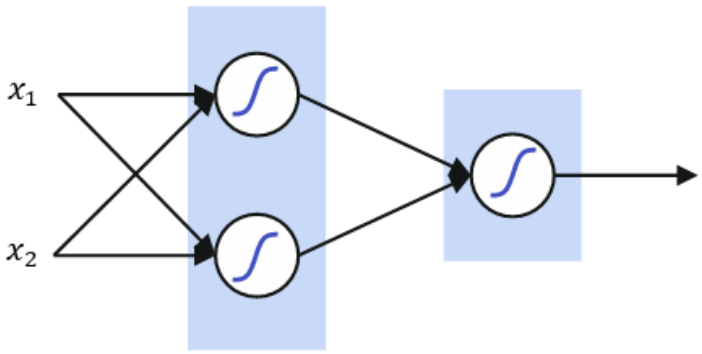

Figure 4.2: Example two-layer feed-forward neural network with sigmoid activation functions.

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

1.4. Exercises

Spend some time familiarising yourself with the neural network model. You do not need to fully understand the internal workings, but you should try and 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 (without any training), 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 more than once for each target (using a loop), 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?