Assignment 1 Part 2: Logistic Regression and Neural Networks

Student Name: Luca J. Ricagni

Student Number: 200894968

Module Name: Machine Learning

Module Code: ECS708U/ECS708P

Due Date: 09/11/2020

Contents

1. Logistic regression

This section will focus on logistic regression for use in classification tasks. When using logistic regression, the aim is to predict discrete classes, such as whether an email is spam or not spam, rather than continuous variables as with linear regression.

Task 1

The first task was to complete the *sigmoid.py* function to use the new hypothesis function in equation 1.

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}} \tag{1}$$

The code for this can be seen in Figure 1.

```
# if z is scalar then
if np.isscalar(z):
    output = 1 / (1 + np.exp(-z))
else:  # z is an array
    output = np.full_like(z, 1)

    for i in range(0, len(z)):
        output[i] = 1 / (1 + np.exp(-z[i]))
```

Figure 1 – Code to implement the hypothesis function, found in sigmoid.py

The code shown in Figure 1 checks if an input value z is a scalar or not. If it is then equation 1 is used to calculate the output. If z is an array, then equation 1 is used for each value of z and the output is an array rather than a scaler value.

After implementing this function it can now be used, *sigmoid.py* is called by *plot_sigmoid.py*, therefore *plot_sigmoid.py* must now be called from within m1_assgn1_ex1_.py. This call can be seen in Figure 2.

```
plot_sigmoid()
```

Figure 2 – Function call to plot_sigmoid.py

Figure 3 shows the graph output when this call is made. This function takes an array of evenly spaced points between -10 and 10 and applies function 1 to it using the code in Figure 1. As can be seen the output is an S shaped curve which is due to the use of the exponential function.

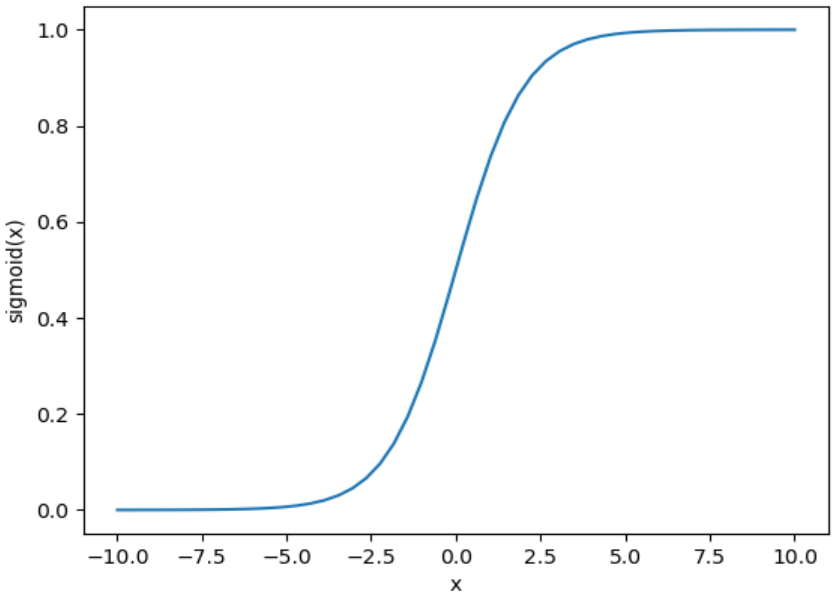

Figure 3 – Shows the sigmoid plot

Task 2

The second task was to plot both the data and the normalized data. This was accomplished by simply adding the code shown in Figure 4 to *plot_data.py* and then running it.

```
X_not_normalised = np.append(column_of_ones, X, axis=1)
fig, ax1 = plt.subplots()
ax1 = plot_data_function(X_not_normalised, y, ax1)
```

Figure 4 – Shows code used to generate graph for the non-normalized data

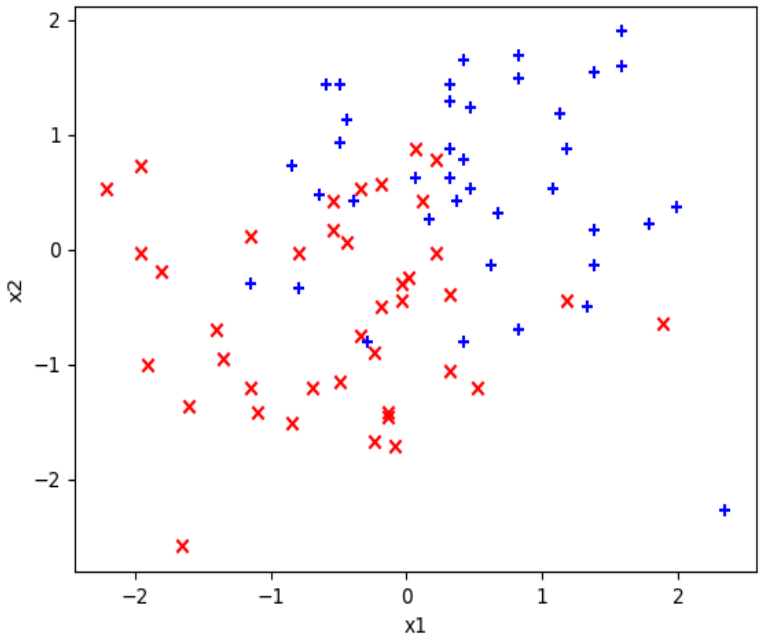

Figure 5 - Plot of normalized data

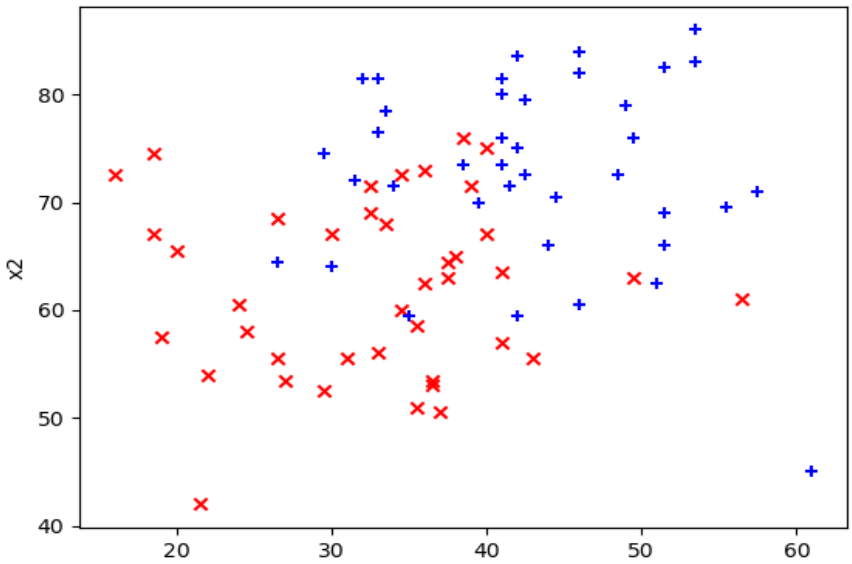

Figure 6 – Plot showing the non-normalized data

Figures 5 & 6 show the plots of the normalized and non-normalized data respectively. As can be seen they are visually identical, however the axis scales are different, this is due to the normalization process.

1.1 Cost function and gradient for logistic regression

Task 3

Now the hypothesis function needs to be completed, this was done using the code in Figure 7.

```
for j in range(0, len(theta)):
    hypothesis += theta[j] * X[i, j]
```

Figure 7 – Code to calculate the hypothesis value

The code shown in Figure 7 uses a for loop to calculate the sum of the product of $\theta_j * X_j$, where j is the length of theta. Therefore, this code will work for datasets of any size. This hypothesis value is then passed to *sigmoid.py* to calculate the final prediction.

Task 4

The cost function must be implemented in *compute_cost.py*. This is done using equation 2.

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}\log\left(h_\theta\left(x^{(i)}\right)\right) - (1-y^{(i)})\log(1-h_\theta\left(x^{(i)}\right))\right] \quad (2)$$

This was done by adding the line of code found in Figure 8 to *compute_cost.py*.

```
cost = (-output * np.log(hypothesis) - ((1 - output) * np.log(1 - hypothesis)))
```

Figure 8 – Code to calculate the cost for a given data point

After both the hypothesis and cost functions were properly implemented (and *gradient_descent.py* was updated with code from the previous report), the full code can now be run. When alpha (the learning rate) was given a value of 1 the program crashes due to a divide by 0 error within the cost function shown in Figure 8.

When alpha is given a more reasonable value of 0.001 a much better result is gained, shown in Figure 9. The minimum cost was 0.49782.

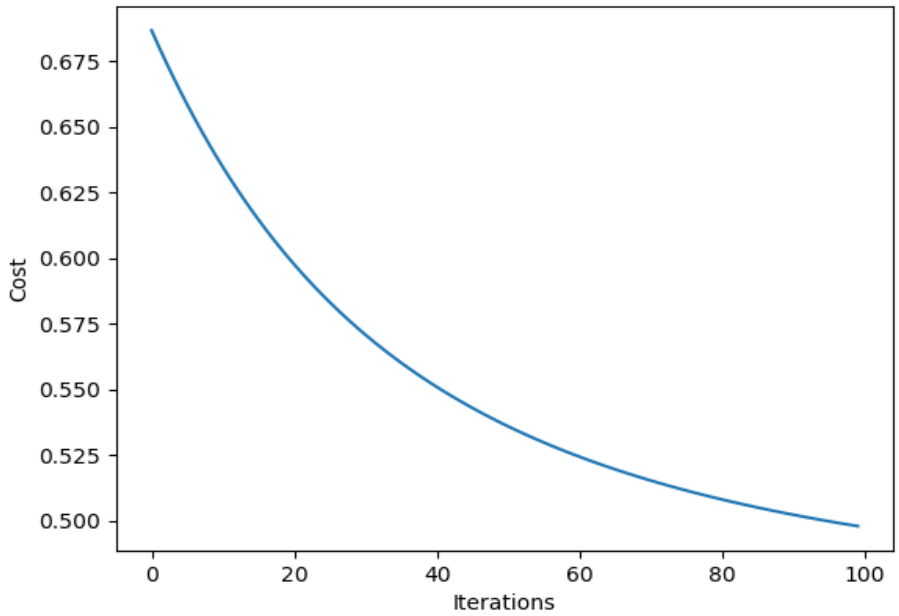

Figure 9 – Shows plot of cost function for alpha = 0.001 and 100 iterations

1.2 Draw the decision boundary

Task 5

The next task was to plot the decision boundary. This was done by equating the formula in equation 1 to 0, as shown in equation 3.

$$g(\theta^T x) = 0 \tag{3}$$

Then writing it out as component-wise multiplication as shown in equation 4.

$$\theta_0 * x_0 + \theta_1 * x_1 + \theta_2 * x_2 = 0 \tag{4}$$

Then by rearranging the formula to solve for x_2. This is shown in equation 5.

$$x_2 = \frac{\theta_0 * x_0 + \theta_1 * x_1}{\theta_2} \tag{5}$$

Now the minimum and maximum values for x_2 can be calculated using the minimum and maximum values of x_1. The code for this can be found in Figure 10 and in *plot_boundary.py*.

```
min_x1 = np.amin(X, axis=0)[1]  # Get min value for x1
max_x1 = np.amax(X, axis=0)[1]  # Get max value for x1

x2_on_min_x1 = -((theta[0] * 1) + (theta[1] * min_x1))/theta[2]  # Calculate min value of x2
x2_on_max_x1 = -((theta[0] * 1) + (theta[1] * max_x1))/theta[2]  # Calculate max value of x2
```

Figure 10 – Code to compute the maximum and minimum values of x_2

Figure 11 shows the normalized data along with the decision boundary. As can be seen this seems to divide the data into two distinct groups although there is some overlap.

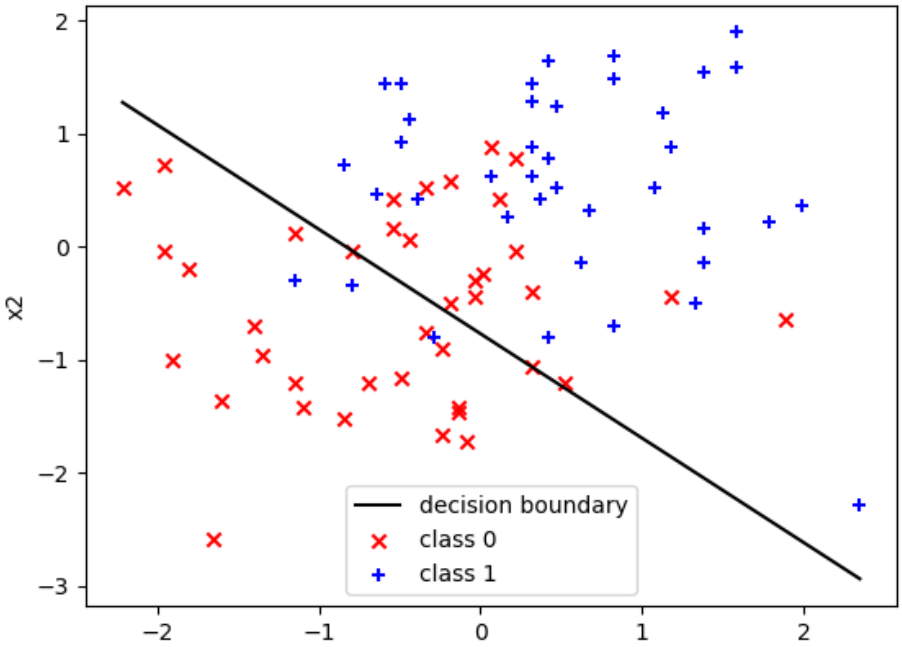

Figure 11 – Plot showing the normalized data and the decision boundary

1.3 Non-linear features and overfitting

Task 6

The first part of this task was to run *assgn1_ex2.py* several times to observe the costs changing as a result of the random shuffling of the training data (20 samples) vs the test data (60 samples). Table 1 shows some examples.

Run	Final Training cost	Minimum Training Cost	Iteration finished	Final Test Cost
1	0.30658	0.29259	11	0.72292
2	0.27046	0.26205	1	0.58364
3	0.43104	0.43104	52	0.40701
4	0.43724	0.43662	5	0.40472
5	0.42209	0.41824	4	0.44657
6	0.36366	0.36214	2	1.433307
7	0.38386	0.36838	2	0.60684
8	0.53065	0.5284	8	0.53742
9	0.57306	0.52941	5	0.41506
10	0.29601	0.28819	5	1.19152

Table 1 – Shows some example output values from running assgn1_ex2.py

Based off the output data from these experiments it is extremely difficult to ascertain a concrete general difference between the training and test costs. However, the average difference between the final training cost and the final test was found to be 0.2734357. The training set tends to generalise well when the majority of the points in a class are close together, and the two sets are generally in different places on the graph – allowing a clearer distinction to be made between them. If there is a lot of overlap between the two sets the classifier performs poorly.

Figure 12 shows a near perfect example of good generalisation, as all but one data point in each set was classified correctly.

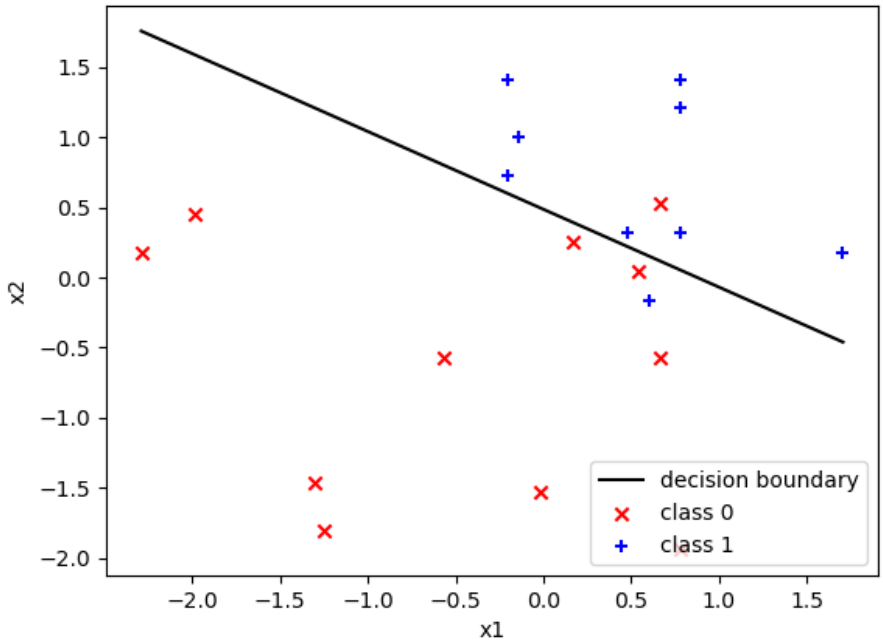

Figure 12 – Shows plot with good generalisation

Figure 12 shows an example of poor generalisation as three points from each set have been incorrectly classified – are on the wrong side of the decision boundary.

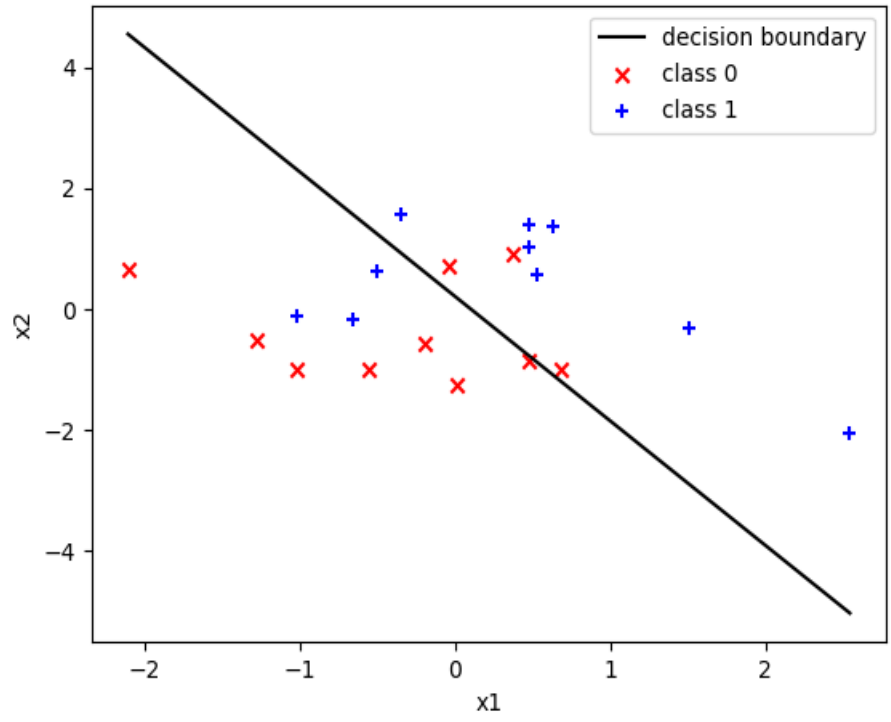

Figure 13 – Shows plot with bad generalisation

The second part of this task was to incorporate the non-linear features x_1x_2, x_1^2, x_2^2 into the feature vector in *assgn1_ex3.py*. This was accomplished using the code found in Figure 14.

```
featureProduct = X[:, 0] * X[:, 1]  # Calculate x1*x2

featureFirstSquared = X[:, 0] ** 2  # Calculate x1^2

featureSecondSquared = X[:, 1] ** 2  # Calculate x2^2

#  Append all features to X
X = np.append(X, featureProduct[:, None], axis=1)
X = np.append(X, featureFirstSquared[:, None], axis=1)
X = np.append(X, featureSecondSquared[:, None], axis=1)
```

Figure 14 – Code to implement non-linear features into the feature vector

The code works by calculating each of the non-linear features individually, then appending them to the end of X individually.

Task 7

Upon running *assgn1_ex3.py* there are two runtime errors which occur, a screenshot of these can be found in Figure 15.

```
C:\Users\ljric\PycharmProjects\Assignment1_Task4\compute_cost.py:24: RuntimeWarning: divide by zero encountered in log
  cost = (-output * np.log(hypothesis) - ((1 - output) * np.log(1 - hypothesis)))
C:\Users\ljric\PycharmProjects\Assignment1_Task4\compute_cost.py:24: RuntimeWarning: invalid value encountered in double_scalars
  cost = (-output * np.log(hypothesis) - ((1 - output) * np.log(1 - hypothesis)))
```

Figure 15 – Shows the runtime error messages when running assgn1_ex3.py

These errors are the same errors which occurred during Task 4. The first error occurs when the argument of $\log()$ is 0, as $\log 0$ gives a divide by 0 error. The second error is similar but subtly different. It occurs when the argument of $\log()$ is very close to 0 and the compiler just converts it to a 0, however this obviously causes an error when in a $\log()$ function for the aforementioned reasons.

Task 8

This task was about experimenting with changing the ratio of test samples to training samples. However, in order to do this some gaps in the code must be filled. Firstly, the non-linear features discussed at the end of Task 6 had to be implemented, this was done in *assgn1_ex4.py* using the same code as before shown in Figure 14. Then *gradient_descent_training.py* had to be updated to calculate the hypothesis, compute the values of sigma for all values of theta and update the values of theta based off the values of sigma. In addition, the function must be modified to store the current cost for the training set and the testing set. This was done using the code shown in Figure 16.

```
train_cost = compute_cost(X_train, y_train, theta)
test_cost = compute_cost(X_test, y_test, theta)

cost_vector_train = np.append(cost_vector_train, train_cost)
cost_vector_test = np.append(cost_vector_test, test_cost)
```

Figure 16 – Code to calculate and store the cost values for the training and test sets

Figure 17 shows the plot of the training cost and testing cost over the length of the program with an alpha value of 0.001 and 20 training samples. As can be seen both costs decrease as number of iterations increases, which is to be expected as the system closes in on the best solution.

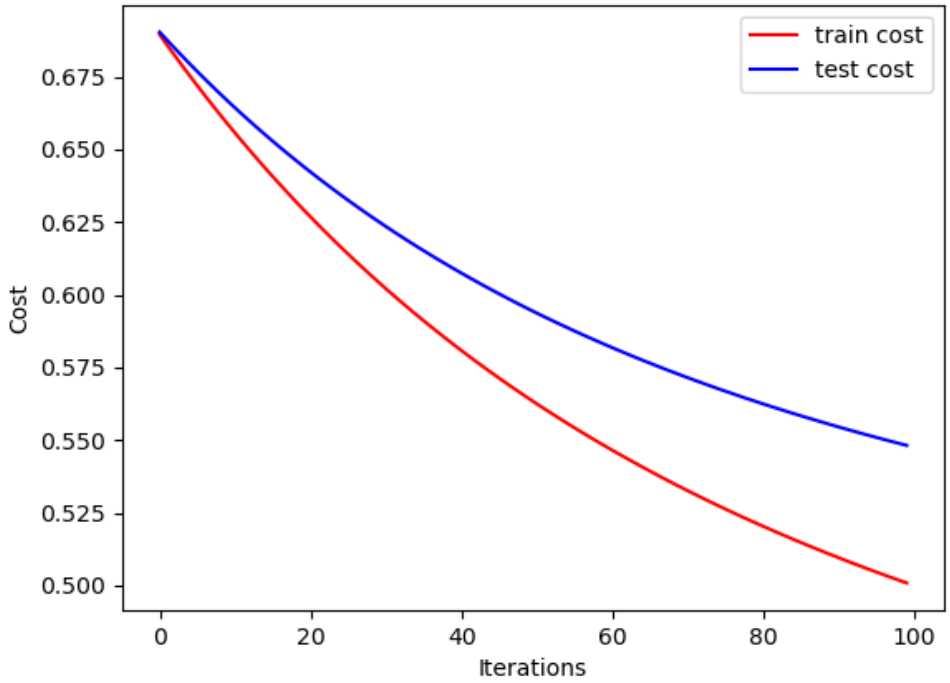

Figure 17 – Graph showing the training and test costs with alpha = 0.001 and 20 training samples

Figure 18 shows the plot of the training cost and testing cost over the length of the program with an alpha value of 0.001 and 50 training samples. When compared to Figure 17, the graph shown in Figure 18 has a lower overall cost, and there is less of a difference between the training and test costs, with the two lines diverging at around iteration 20.

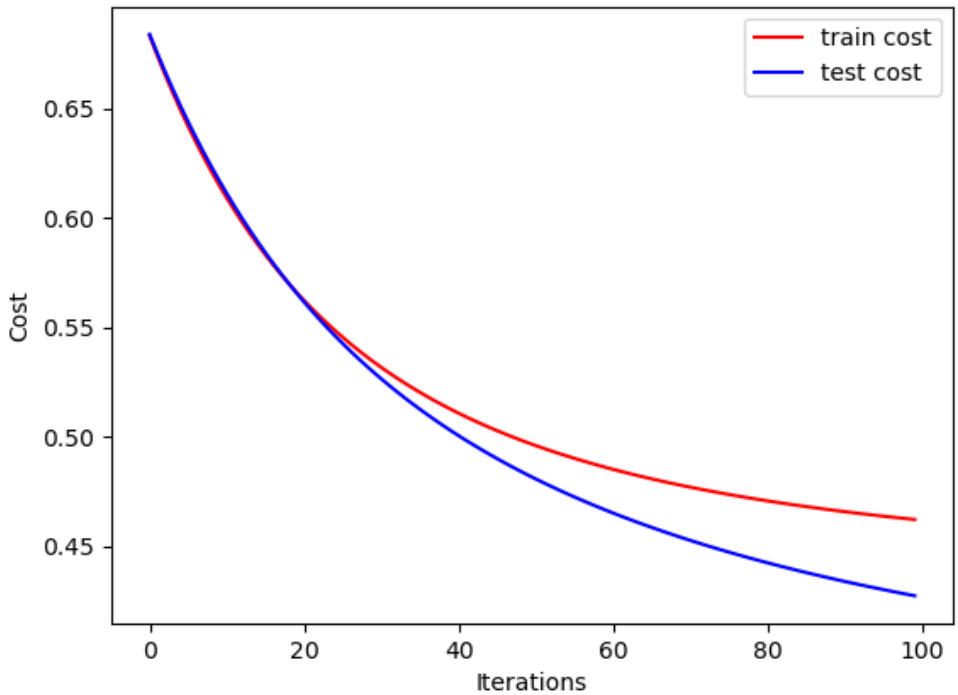

Figure 18 - Graph showing the training and test costs with alpha = 0.001 and 50 training samples

Figure 18 shows the plot of the training cost and testing cost over the length of the program with an alpha value of 0.001 and 70 training samples. There is a much larger difference between the final training and test costs than previously. Furtheremore the lines seem to start plateauing earlier.

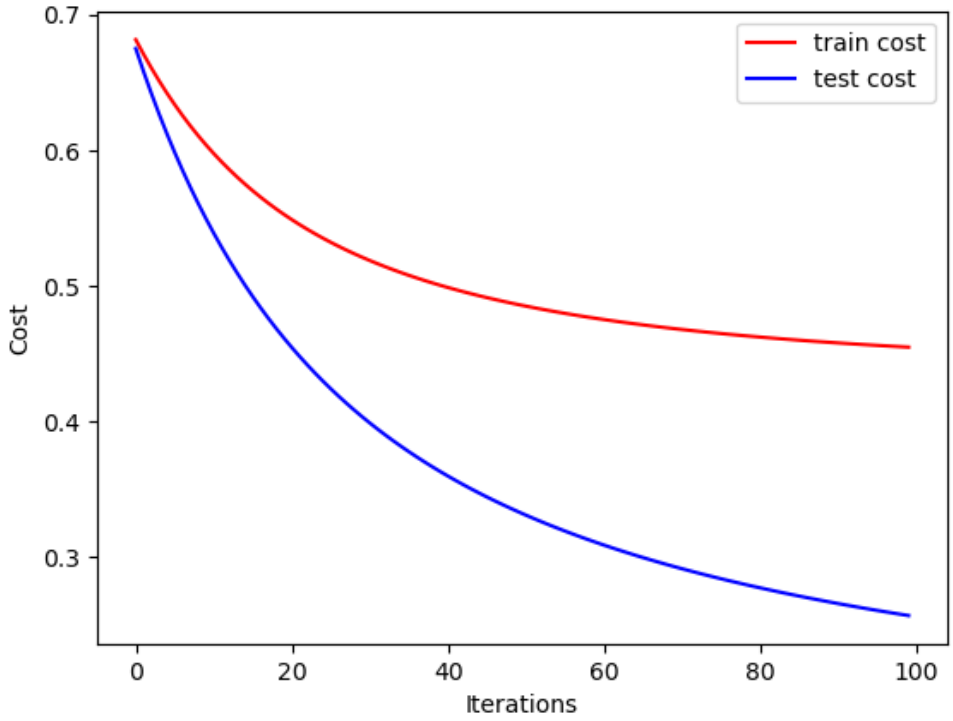

Figure 19 - Graph showing the training and test costs with alpha = 0.001 and 70 training samples

It is difficult to draw an overall conclusion due to the randomness within selecting the samples to be used for training, meaning there is a great deal of variation between different runs with the same parameters. However, it seems that generally speaking the more of the data set is given over to training the higher the training cost is, but the lower the test cost. This is thought to be because having more training samples means that the system will be able to better optimise itself, thus lowering the average cost for the testing samples, giving better predictions.

Next some extra features, a second order and third order polynomial, were added to *assgn1_ex5.py* as shown in Figure 20.

```
featureProduct = X[:, 0] * X[:, 1]  # Calculate x1*x2
featureFirstSquared = X[:, 0] ** 2  # Calculate x1^2
featureSecondSquared = X[:, 1] ** 2  # Calculate x2^2
featureFirstCubed = X[:, 0] ** 3  # Calculate x1^3
featureSecondCubed = X[:, 1] ** 3  # Calculate x2^3

#  Append all features to X
X = np.append(X, featureProduct[:, None], axis=1)
X = np.append(X, featureFirstSquared[:, None], axis=1)
X = np.append(X, featureSecondSquared[:, None], axis=1)
X = np.append(X, featureFirstCubed[:, None], axis=1)
X = np.append(X, featureSecondCubed[:, None], axis=1)
```

Figure 20 – Code to add the product, second and third order polynomials

The code was then run with 20, 50 and 70 training samples for comparison. The outputs are shown in Figures 21, 22 and 23, respectively.

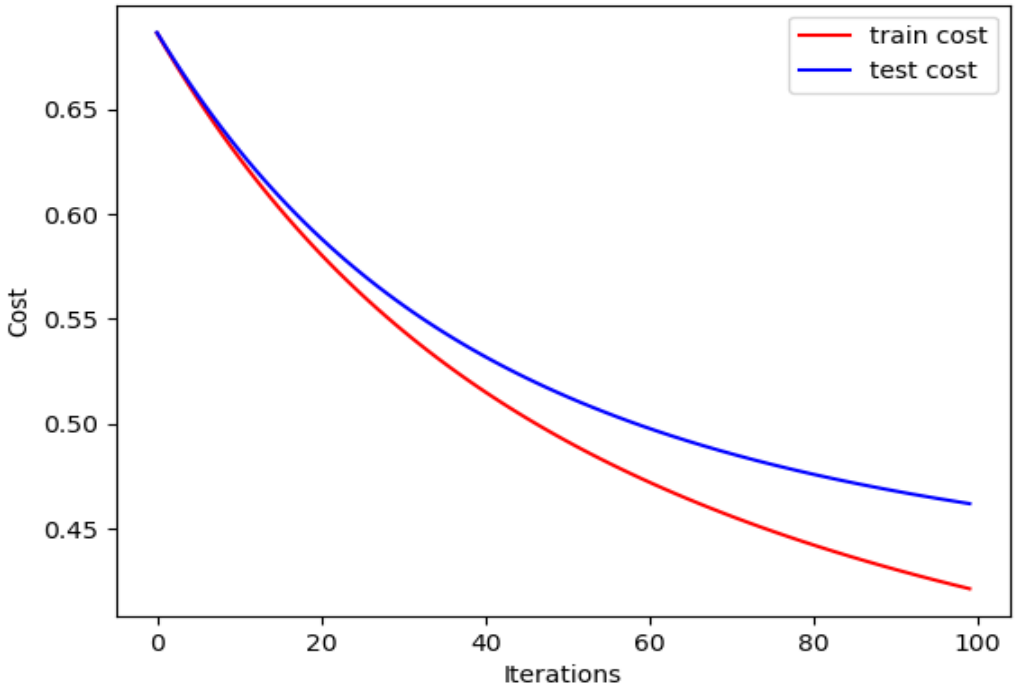

Figure 21 – Graph showing the training and test costs with alpha = 0.001 and 20 training samples

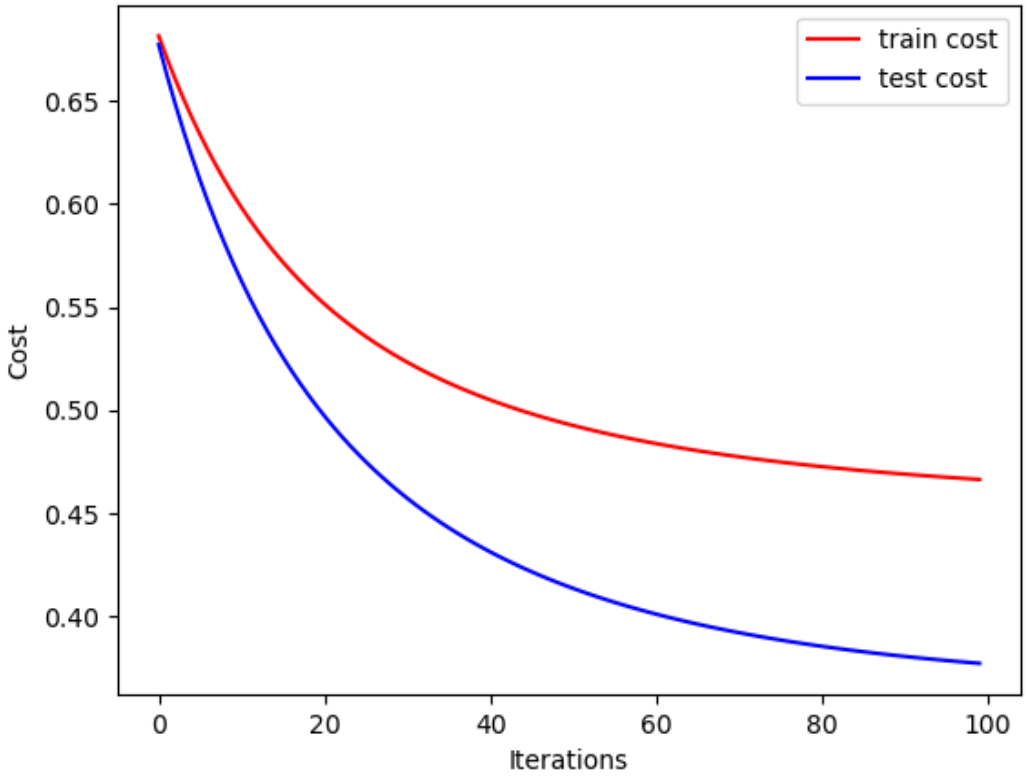

Figure 22 – Graph showing the training and test costs with alpha = 0.001 and 50 training samples

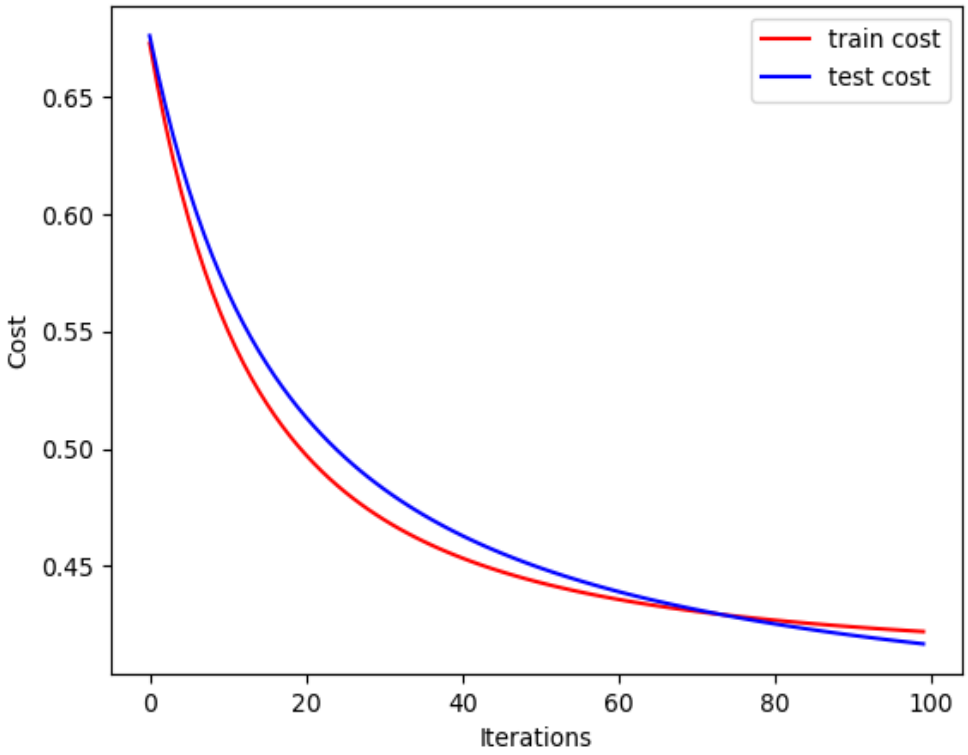

Figure 23 – Graph showing the training and test costs with alpha = 0.001 and 70 training samples

Once again it is difficult to draw a concrete conclusion because of the randomness within the selection of the training samples. But it would seem that overall adding the third order polynomial has lowered the average training cost across the board, the exception being the test cost with 70 training samples. This was significantly higher on average as shown by comparing Figures 19 & 23. To conclude there is still a strong correlation between having a larger training set and having lower average test costs.

Task 9

Logistic regression works by splitting the plane into two halves with a line, thus splitting the data into two sets. For a problem with an OR problem space this works very well, as can be seen in the left graph of Figure 24. However, as can be clearly seen in the right graph of Figure 24 there is no way to draw a straight line which will give satisfactory classification. This is because no matter where the line is drawn the classification coefficient will never be more than 50%.

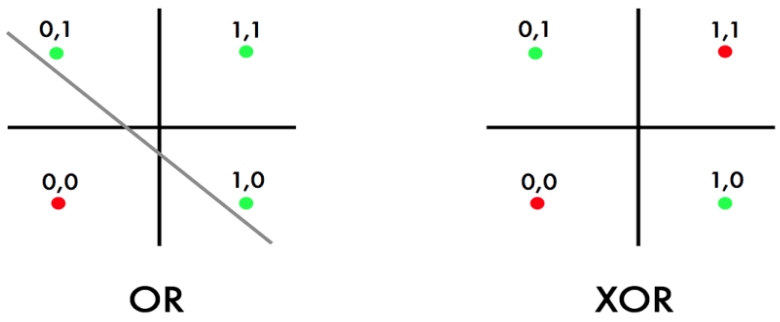

Figure 24 – Graphs showing the OR and XOR problem space

Image can be found here: https://res.cloudinary.com/practicaldev/image/fetch/s--7VACmpbO--/c_imagga_scale,f_auto,fl_progressive,h_500,q_auto,w_1000/https://dev-to-uploads.s3.amazonaws.com/i/lkli02223oqhlac1jetz.png

2. Neural Network

This section will look at using backpropagation on a feedforward neural network in order to solve the XOR problem.

The first step was to fill in the *sigmoid.py* function, this was once again done using the code shown in Figure 1.

Task 10

Next the backpropagation function within NeuralNetwork.py must be completed and should support outputs of any size. To accommodate outputs of any size all the steps of backpropagation were implemented using loops set to run for the length of arrays, meaning they will dynamically adjust for any size input/output. In addition, steps 1 & 2 contain *if* statements in order to deal with cases where the variable is not an array but rather a scaler value. The code was implemented in steps 1-4.

Step 1: Calculate the error delta

The first step was to calculate the error delta using the formula shown in equation 6, which was implemented using the code shown in Figure 5

$$\delta_k = (y_k - t_k) * g'(x_k) \tag{6}$$

```
for i in range(self.n_out):
    ########################################
    # Write your code here
    # compute output_deltas : delta_k = (y_k - t_k) * g'(x_k)

    if np.isscalar(targets):  # If targets is a scalar
        output_deltas[i] = (outputs - targets) * sigmoid_derivative(outputs[i])
    else:  # targets is an array
        output_deltas[i] = (outputs[i] - targets[i]) * sigmoid_derivative(outputs[i])
    ########################################/
```

Figure 25 – Code to calculate error delta

Step 2: Backpropagate error delta to hidden neurons

The second step was to backpropagate the error delta to the hidden neurons using the formula shown in equation 7, implemented using the code shown in Figure 26.

$$\delta_j = g'(x_j) \sum_{k}^{j} (w_{jk}\delta_j) \tag{7}$$

```
# Step 2. Hidden deltas are used to update the weights of the hidden layer
hidden_deltas = np.zeros((len(self.y_hidden)))

# Create a for loop, to iterate over the hidden neurons.
# Then, for each hidden neuron, create another for loop, to iterate over the output neurons
for i in range(len(hidden_deltas)):
    sum = 0.0

    if self.n_out == 1:
        sum += output_deltas * self.w_hidden[0, i - 1]
    else:
        for j in range(0, self.n_out):
            sum += output_deltas[j] * self.w_hidden[j, i - 1]

    hidden_deltas[i] = sigmoid_derivative(self.y_hidden[i]) * sum
```

Figure 26 – Code to backpropagate error delta to hidden neurons

Step 3: Update output weights

The third step was to update the output weights using the formula in equation 8, this was implemented using the code in Figure 27.

$$w_{jk} = w_{jk} - \eta\delta_k\alpha_j \tag{8}$$

```
# Step 3. update the weights of the output layer
for i in range(len(self.y_hidden)):
    for j in range(len(output_deltas)):
        ########################################
        # Write your code here
        # update the weights of the output layer
        self.w_out[i, j] -= learning_rate * output_deltas[j] * self.y_hidden[i]
        ########################################/
```

Figure 27 – Code to update output weights

Step 4: Update hidden weights

The fourth and final step was to update the hidden weights using the formula found in equation 9, this was then implemented using the code shown in Figure 28.

$$w_{jk} = w_{jk} - \eta\delta_k\alpha_j \tag{9}$$

```
# Step 4. update the weights of the hidden layer
# Create a for loop, to iterate over the inputs.
# Then, for each input, create another for loop, to iterate over the hidden deltas
for i in range(len(inputs)):
    for j in range(len(hidden_deltas)):
        ########################################
        # Write your code here
        # update the weights of the hidden layer
        self.w_hidden[i, j] -= learning_rate * hidden_deltas[j] * inputs[i]
        ########################################/
```

Figure 28 – Code to update weights of hidden neurons

Results

After a great deal of testing with different values for the learning rate the best value was determined to be 0.5. This gave the best balance between getting the cost down to ~ 0, the number of iterations taken to do this, and achieving this in the majority of cases. An example graph of the error function when the systems successfully minimise the cost can be found in Figure 29.

Due to the way the backpropagation algorithm works, it can sometimes get stuck in a local optima. This is normally caused by update differences occurring on the weights connected between the hidden layer and the output layer. An example of the algorithm getting stuck in a local optima can be seen in Figure 30. As can be seen instead of converging to 0, the cost function instead converges to ~ 0.4, showing that the algorithm has become stuck in a local optima.

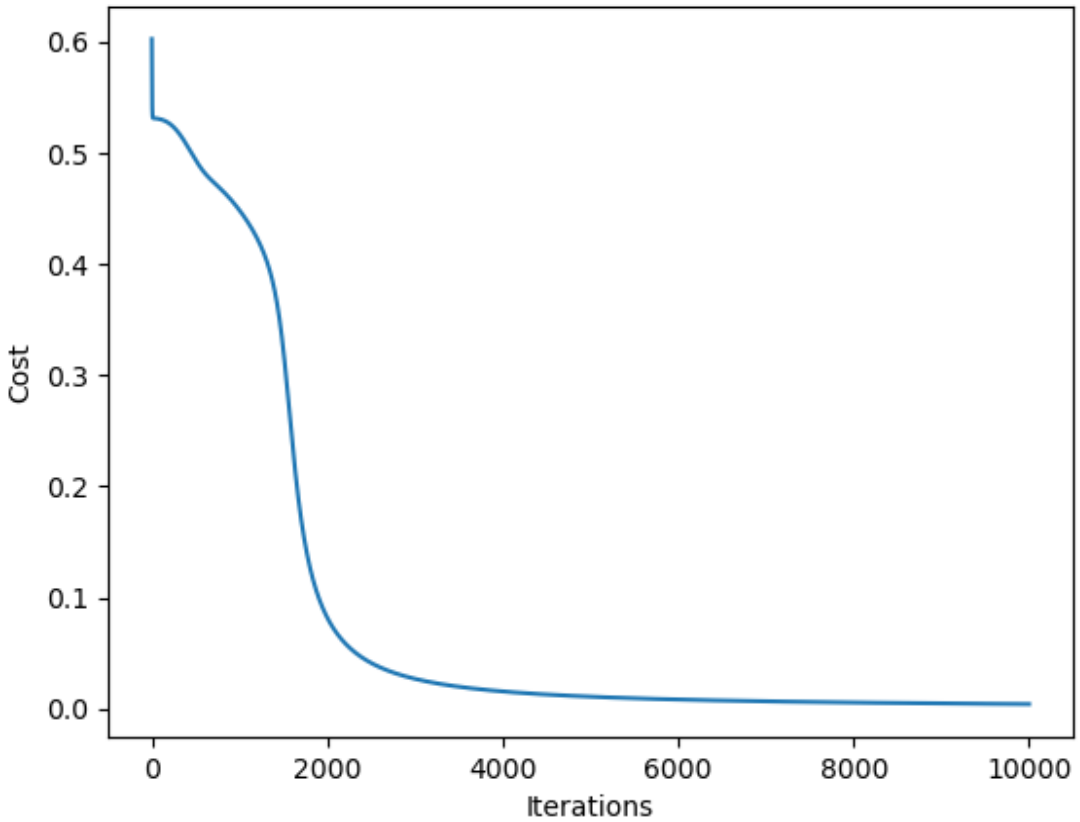

Figure 29 – Cost graph of a successful run, alpha = 0.5 and iterations = 10000

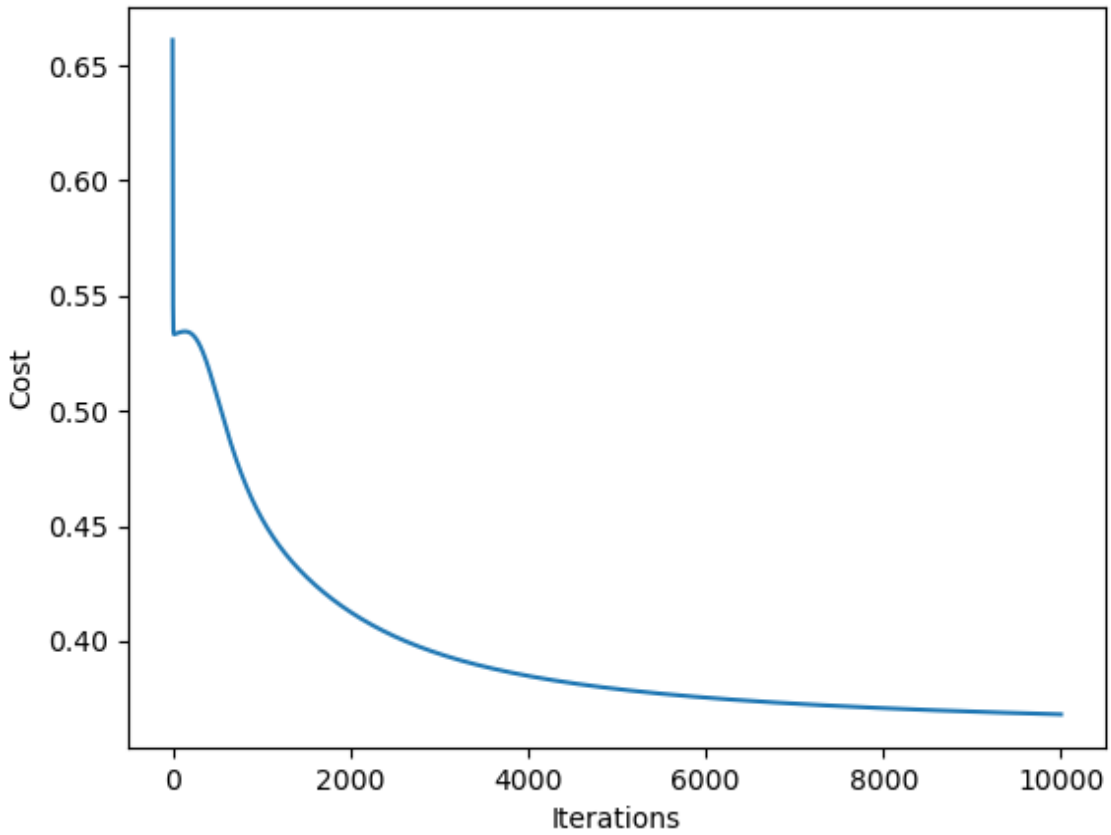

Figure 30 – Cost graph when backpropagation gets stuck in a local optima, alpha = 0.5 and iterations = 10000

2.1 Implement backpropagation on XOR

Task 11

The aim of this task was to implement a different logical function to model, such as NOR or AND, and plot the error function of a successful trial. In order to do this the desired output variable y, found in *xorExample.py*, must be changed. Figure 31 shows the code to change the desired output to a NOR function, and Figure 32 shows the code for an AND function.

```
y = np.array([0, 0, 0, 1])  # AND
```

Figure 31 – Code for an AND function

```
y = np.array([1, 0, 0, 0])  # NOR
```

Figure 32 – Code for a NOR function

Figure 33 shows an example of a successful trial where the output was an AND function. The trial was successful because the cost function converges to 0.

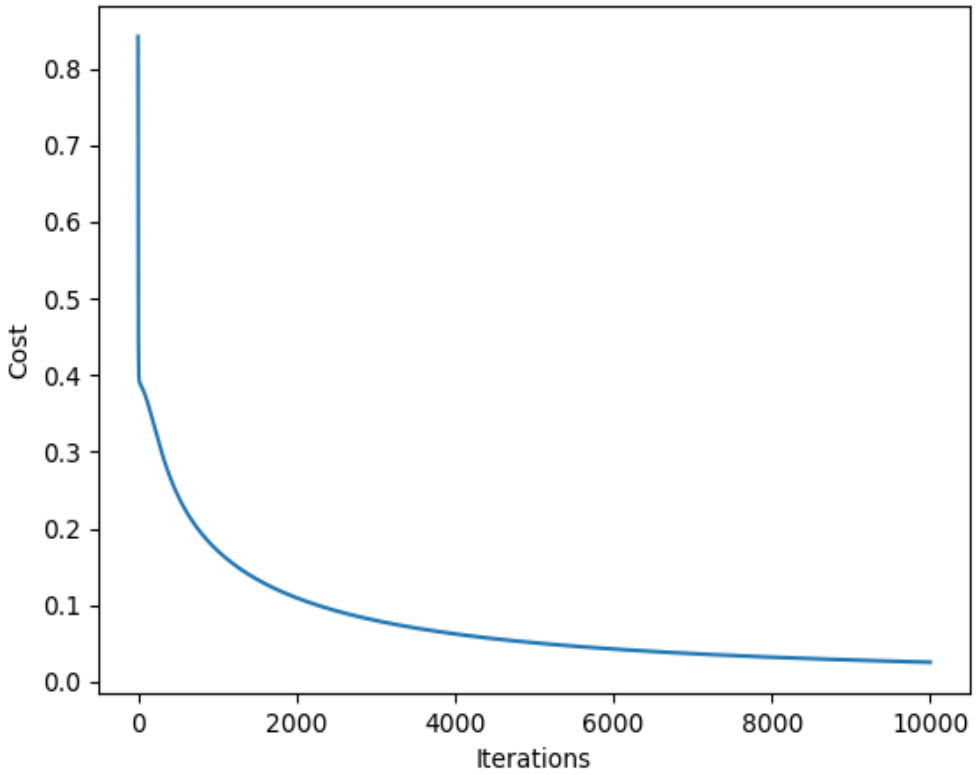

Figure 33 – Cost graph of successful trail to model an AND function, alpha = 0.5 and iterations = 10000

Figure 34 shows an example of a successful trial where the output was an NOR function. The trial was successful because the cost function converges to 0.

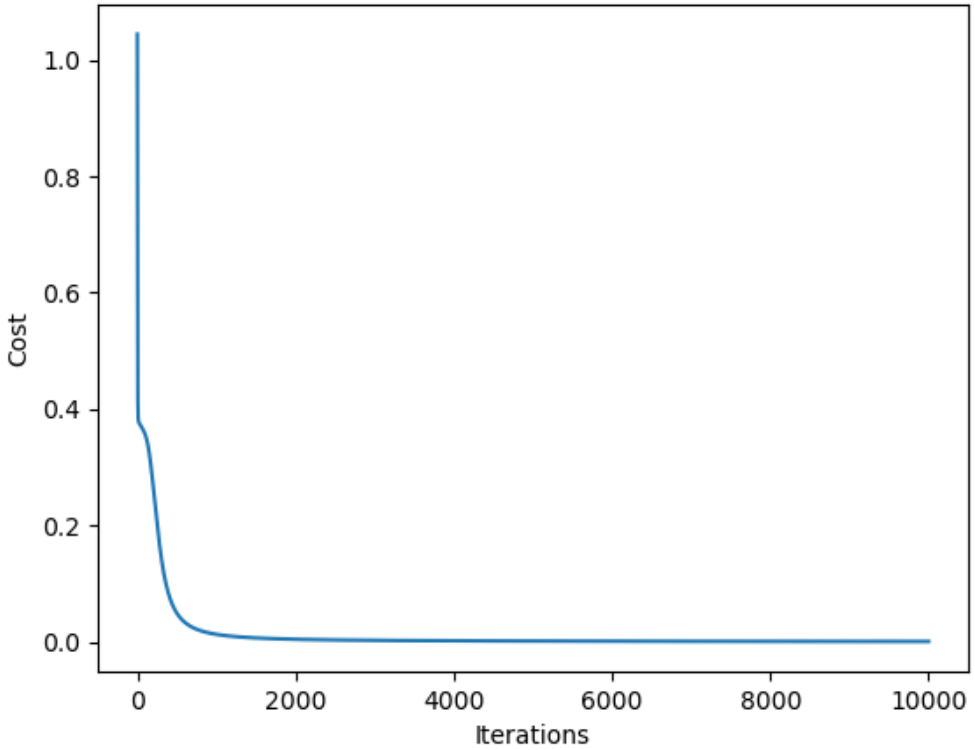

Figure 34 – Cost graph of successful trail to model a NOR function, alpha = 0.5 and iterations = 10000

2.2 Implement backpropagation on Iris

Task 12

In order to use logistic regression to discriminate between the three classes (types of iris) found in the iris dataset one would have to use multinomial regression, a modified version of logistic regression. Logistic regression works by dividing the state space in two, one for each class, and as such only works when classifying between two sets or variables.

Multinomial regression works by essentially performing logistic regression n times, where n is the number of desired output classes. Each model is developed separately, class A vs the rest, class B vs the rest etc. Then the probability of a given sample is calculated for each class, and the highest probability is where the sample is classified to be.

This classification problem could also be solved using a neural network. These have an input layer, n number of hidden layers, and an output layer. They can also have no hidden layers and this special case is the same as performing logistic regression. The main difference between logistic regression and neural networks lies in the way they calculate their predictions. Logistic regression uses a sigmoid function to calculate the probability, neural networks use forward and backwards propagation and then another function (such as a sigmoid) to calculate the final probability. As such neural networks tend to be able to model more complex data with more variables but tend to be more computationally expensive. Logistic regression is good for binary classification, but is not as good as a neural network for multi class classification.

Task 13

The aim of this task was to determine what the optimal number of hidden neurons was and why, in order to classify the iris dataset. This was done by running *irisExample.py* with different numbers of hidden neurons. In order to find a good average a simple for loop was added around the main body of code, which would run the code n number of times, then calculate the average minimum cost and

the iteration where the minimum cost was reached. This average minimum cost and associated average number of iterations will be used to determine how well differing numbers of hidden neurons perform, as the lower the average cost and number of iterations the better the performance. To prevent the code pausing until the output figures were closed, the code to do this was commented out.

For each number of hidden neurons, the code was run 100 times, and the average minimum cost and lowest iteration calculated for each. This process was repeated 3 times. The reader should note that the average lowest iteration value has been rounded to the nearest integer. The results of these trials can be found in Table 2.

Number of Hidden Neurons	Average Minimum Cost	Average Lowest Iteration
Test 1		
1	3.85665	76
2	0.68308	89
3	0.43509	85
5	0.45395	81
7	0.4823	79
10	0.48269	75
Test 2		
1	3.95157	68
2	0.75394	91
3	0.48484	88
5	0.4089	82
7	0.47493	77
10	0.4884	78
Test 3		
1	4.00401	73
2	0.85059	87
3	0.51408	84
5	0.51608	82
7	0.41098	81
10	0.43894	81

Table 2 – Shows 3 sets of the average minimum cost and lowest iteration across 100 trials for several numbers of hidden neurons

Table 2 shows the results of the 3 tests that were conducted. The results definitely show that 1 hidden neuron is the worst option, although there is some ambiguity otherwise. Based off the trials conducted in Tests 1 & 2 it would be obvious that either 3 or 5 hidden neurons was best, as they had the lowest average costs. However, the reader will notice that the results of Test 3 do not completely align with these results as 7 hidden neurons was the best in this Test.

It would seem that overall, the general trend is that the optimal number of hidden neurons is between 3 and 7, generally leaning towards the middle of that spectrum. In conclusion 5 hidden neurons is the best value to use as it provides the best compromise between the lowest cost, smallest number of iterations and reliability.