Queen Mary University of London

Department of Electrical Engineering & Computer Science

ECS708P Machine Learning

Assignment 1: Part 2 – Logistic Regression & Neural Networks

Hasan Emre Erdemoglu

9 November 2020

Table of Contents:

1 Logistic Regression

The sigmoid function formula is given in the lab assignment. The following code will realize this function. Note that $z = -\theta^T x$.

```python
import numpy as np

def sigmoid(z):
    return 1/(1+np.exp(-z))

```

*Figure 1. Implementation of **sigmoid.py***

The function called ***plot_sigmoid.py*** can be used to draw the graph for this sigmoid. This function is already prepared, and the output is as follows:

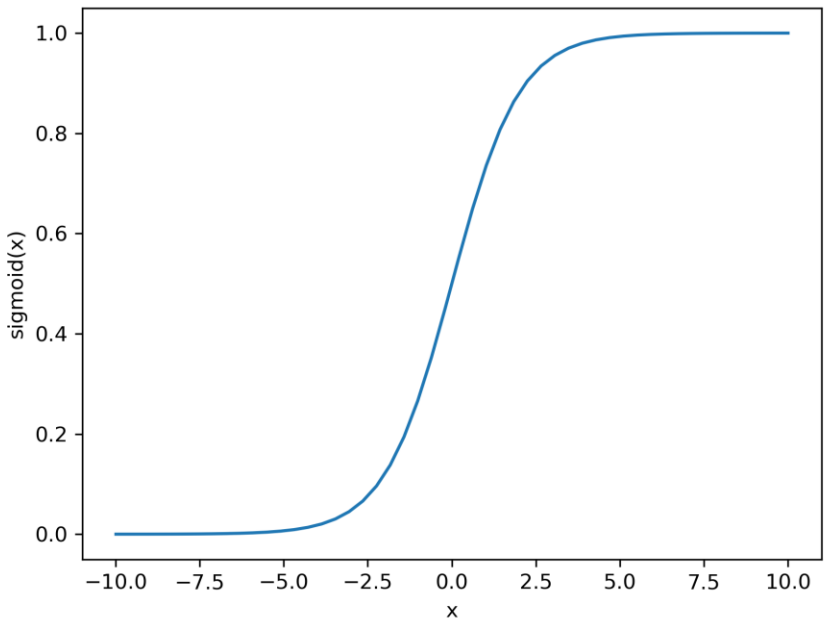


*Figure 2. Executing **plot_sigmoid.py***

Note that *plot_sigmoid()* function is within *plot_sigmoid.py*; to execute it the following code snippet is used:

```
from plot_sigmoid import *

import matplotlib.pyplot as plt

plot_sigmoid()

plt.show()
```

For **Task 2**; the non-normalized and normalized data can be seen below. For the graph without the normalization, in line 18 of *plot_data.py* the following changes are made:

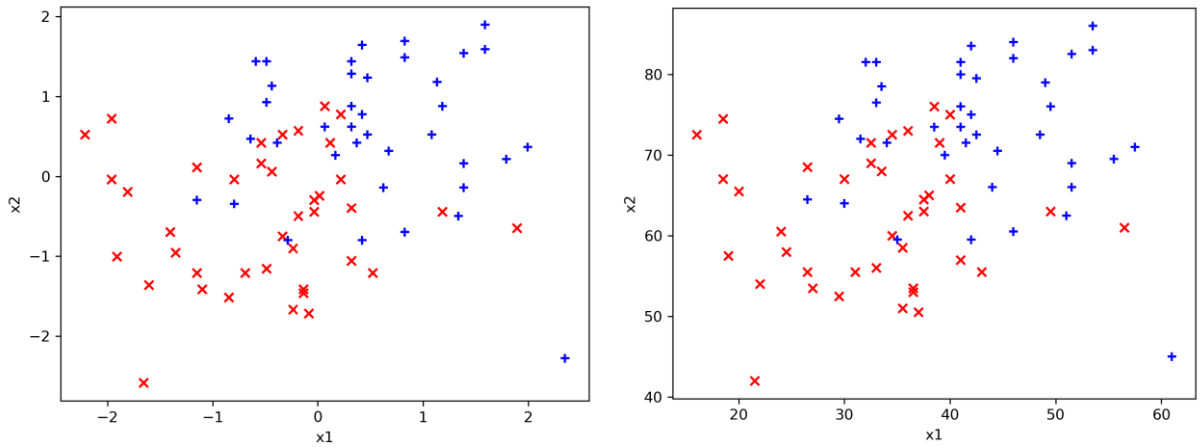


Figure 3. Data with normalization (left), Data without normalization (right)

```
### Normalized Version ###

(…) # imports

# this loads our data

X, y = load_data_ex1()

# this normalizes our data

X_normalized, mean_vec, std_vec = normalize_features(X)

# After normalizing, we append a column of ones to
X_normalized, as the bias term

column_of_ones = np.ones((X_normalized.shape[0], 1))

# append column to the dimension of columns (i.e., 1)

X_normalized = np.append(column_of_ones,
X_normalized, axis=1)

fig, ax1 = plt.subplots()

ax1 = plot_data_function(X_normalized, y, ax1)

# enter non-interactive mode of matplotlib, to keep figures
open

plt.ioff()

plt.show()
```

```
### Non-normalized Version ###

(…) # imports

# this loads our data

X, y = load_data_ex1()

# this normalizes our data

X_normalized, mean_vec, std_vec = normalize_features(X)

# After normalizing, we append a column of ones to
X_normalized, as the bias term

column_of_ones = np.ones((X.shape[0], 1))

# append column to the dimension of columns (i.e., 1)

X = np.append(column_of_ones, X, axis=1)

fig, ax1 = plt.subplots()

ax1 = plot_data_function(X, y, ax1)

# enter non-interactive mode of matplotlib, to keep figures
open

plt.ioff()

plt.show()
```

1.1 Cost Function & Gradient for Logistic Regression

The hypothesis is calculated by $h_\theta = \sigma(w^T X)$. This is achieved by writing the following code in *calculate_hypothesis.py:*

```
import numpy as np

from sigmoid import *

def calculate_hypothesis(X, theta, i):

        return sigmoid(np.dot(X[i], theta))
```

Due to its vectoral nature this function will be able to handle datasets of any size. The file *compute_cost.py* is also modified to incorporate the cost, the following lines are showing the implementation:

```
(…) # imports

def compute_cost(X, y, theta):

 (…) # pre-written code

    cost = 0.0

    cost = -y[i] * np.log(hypothesis) - (1-y[i])*(np.log(1-hypothesis))

    J += cost

  J = J/m

  return J
```

The task requires to run *ml_assgn1_ex1.py* however it seems that gradient_descent() method in *gradient_descent.py* is not implemented. The following section shows the implementation gradient_descent() method and the output of *ml_assgn1_ex1.py*.

```
(…) # imports

def gradient_descent(X, y, theta, alpha, iterations):

(…) # documentation comment and code

    sigma = np.zeros((len(theta)))

    for i in range(m):

        hypothesis = calculate_hypothesis(X, theta_temp, i)

        output = y[i]

        sigma +=  np.dot((hypothesis - output),X[i])

    theta_temp -= alpha/m * sigma

(…) # code

        return theta, cost_vector
```

For experimentation **alpha** in *ml_assgn1_ex1.py* is increased to 5. This led to reaching a minimum cost of 0.40545 in 56 iterations. In the original learning rate of 1; the same value is reached in 100 iterations where 100 iterations are the maximum value. A learning rate of 5 is stable and converges to the minimum faster.

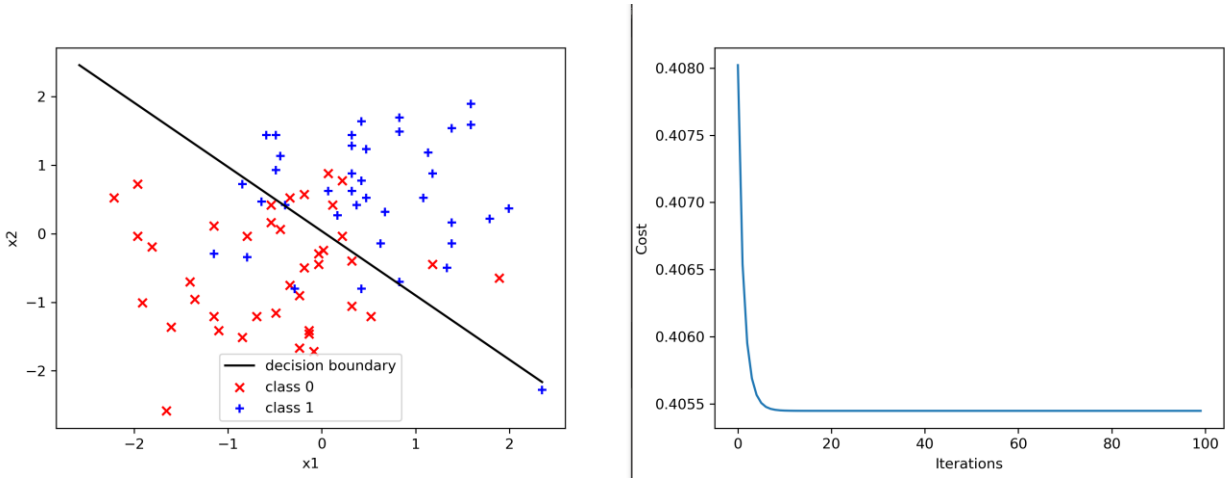


Figure 4. Outputs of ml_assgn1_ex1.py. Learning rate is 5, Min cost 0.40545 in 56 iterations.

1.2 Draw the Decision Boundary

Figure 4; the plot on the left also shows the decision boundary for the logistic regression. Implementation is done on *plot_boundary.py* which is called by *ml_assgn1_ex1.py*. The calculation steps, for this scenario, is listed below:

$$\theta^T X = 0$$

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$$

$$\boldsymbol{x_2 = \frac{-\theta_0 - \theta_1 x_1}{\theta_2}}$$

This equation is realized with the following code:

```
(…) # imports

def plot_boundary(X, theta, ax1):

 (…) # pre-written code

  ########################################

  # Re-arrange the terms in the equation of the hypothesis function, and solve with respect to x2, to find its values on given values of x1

  min_x1 = np.min(np.min(X, axis=1))

  max_x1 = np.max(np.max(X, axis=1))

  x2_on_min_x1 = (-theta[0]-theta[1] * min_x1) / theta[2]

  x2_on_max_x1 = (-theta[0]-theta[1] * max_x1) / theta[2]

  # np.dot(X[np.where(min_x1)], theta)

  ########################################/

(…) # pre-written code
```

1.3 Non-linear Features & Overfitting

For Task 6, *ml_assgn1_ex2.py* is run several times. For faster convergence the learning rate is increased to 5, following from previous sections. The following table shows the training and test costs respectively. The best generalization is shown with green and the worst generalization is shown in red. The respective boundary functions and the training cost graphs of these cases are also listed below.

Table 1. Different runs of ml_assgn1_ex2.py

TEST RUN	MIN TRAIN COST, ITERATION	FINAL TRAINING COST	TEST COST
#1	**0.02691, 100**	**0.02691**	**1.20835**
#2	0.028985,100	0.028985	0.58415
#3	0.36340, 72	0.36340	0.52471
#4	0.43644, 61	0.43644	0.49105
#5	**0.46912, 60**	**0.46912**	**0.47566**

From the table above it is seen that the test cost is always higher than the training costs. Since training dataset is significantly smaller than the test dataset; the training dataset is not able to generalize well to the entire data. This effect is further amplified as training samples are extracted from the dataset randomly. When the randomly sampled training samples have a clear-cut boundary between classes; it does not fully describe the noise between the classes. As the decision boundary is solely selected by the training samples; it does not generalize to the test set well. See the following set of figures for Test Run #1, which the model not generalize well to the test dataset:

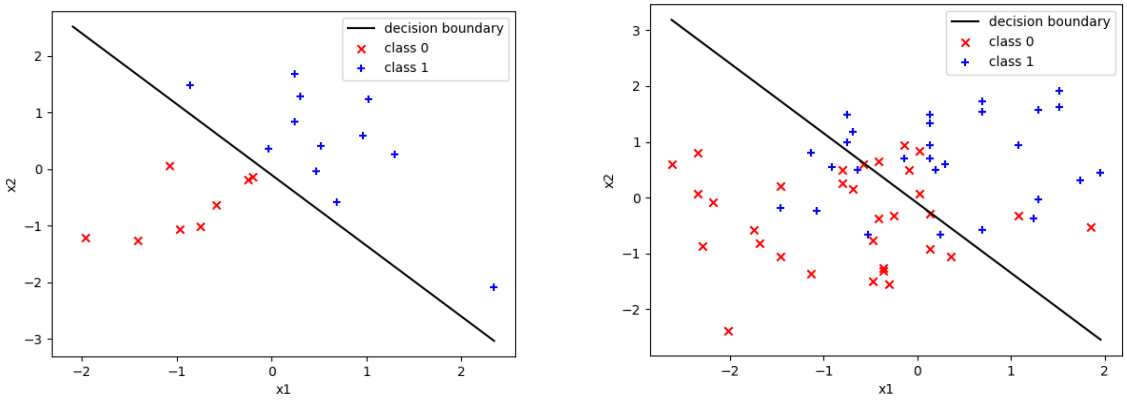


Figure 5. Decision Boundary on Training Data (Left) and Testing Data (Right) – Test Run #1

As this random sampled training data has a clear-cut boundary; the training error is very low. But note that this sample training dataset does not fully describe the actual distribution of the entire dataset; with the elements of noise; as seen by the test dataset on the right. The data which are closest to prospective decision boundaries have the most effect on generalizability. In this case, the training data caught a general boundary between classes; but the boundary is not fully

optimized where both classes are mixed to each other. If the training samples contain data points from the proximity of where these two classes get mixed, the model will be able to pick a best fit line minimizing the cost and providing a better boundary. Test Run #5 was able to achieve this as it picked 4-5 points which is somewhat close to each and near the decision boundary. It was able to get a training and test error of 0.46 and 0.47 respectively, where the best training error on the previous section was approximately 0.40. The decision boundary output can be seen below:

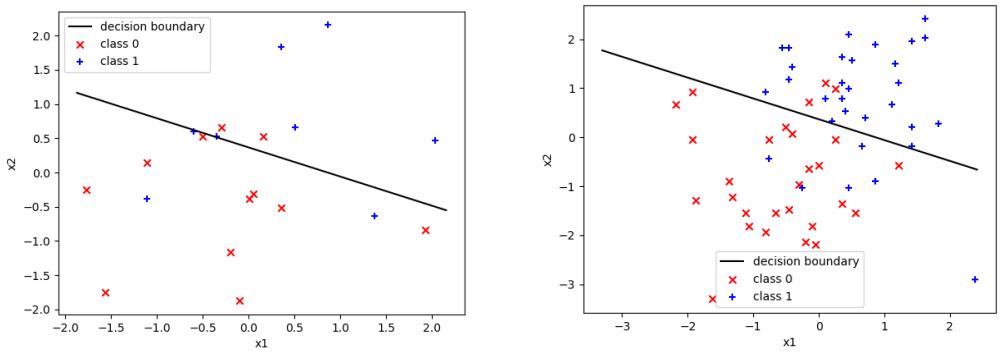


Figure 6. Decision Boundary on Training Data (Left) and Testing Data (Right) – Test Run #5

Many more runs may be made and there is a possibility that the random sampled training data could better explain the overall relations in the model. However, as a general case; 20 samples were not sufficient to describe the dataset. Increasing the number of training samples could help with the training process. But this time; low amount of test samples could output incorrect results; especially if the randomly selected datapoints are outliers. In this case as there a small number of samples are available; their effect on assessing the accuracy of the model will be larger.

Continuing with Task 6, some non-linear features are also implemented; to be tested on *ml_assgn1_ex3.py*. To be consistent with other experiments, learning rate is set to 5 and the following code segment is written inside of *ml_assgn1_ex3.py:*

```
(…) # imports and code:

# Create the features x1*x2, x1^2 and x2^2

new_feat1 = np.expand_dims(X[:,0] * X[:,1], axis=1)

new_feat2 = np.expand_dims(X[:,0] ** 2, axis=1)

new_feat3 = np.expand_dims(X[:,1] ** 2, axis=1)


# Not a neat way of doing it

X = np.append(X, new_feat1, axis=1)

X = np.append(X, new_feat2, axis=1)

X = np.append(X, new_feat3, axis=1)
```

```
(…) # pre-written code
theta = np.zeros((6))
# Set learning rate alpha and number of iterations
alpha = 5
iterations = 100
(…) # pre-written code
```

After implementing the non-linear features, Task 7, requires running this file. The following results are achieved:

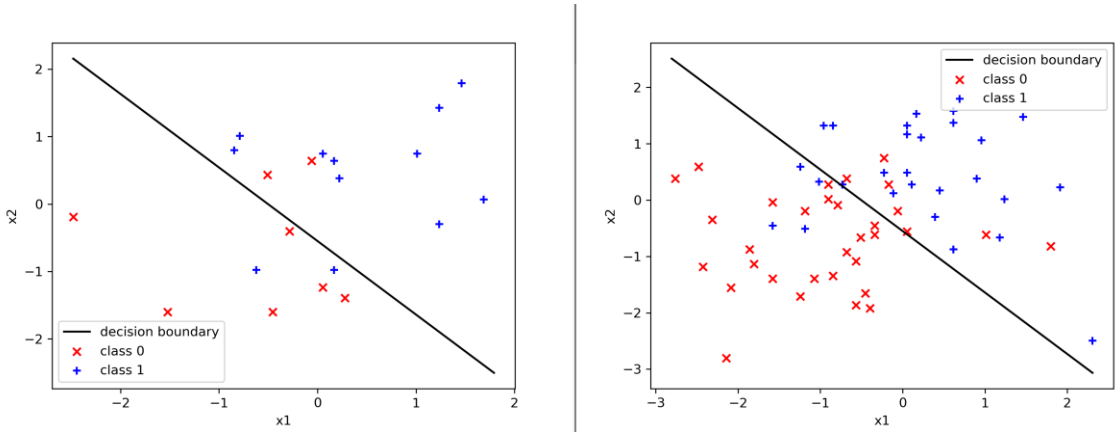


Figure 7. Decision Boundary on Training Data (Left) and Testing Data (Right)

Minimum (and final) training cost of 0.41428 is achieved on iteration 59, which is kept constant until 100th iteration. The final test cost is reported as 0.40902. The following is the progression of cost over iterations:

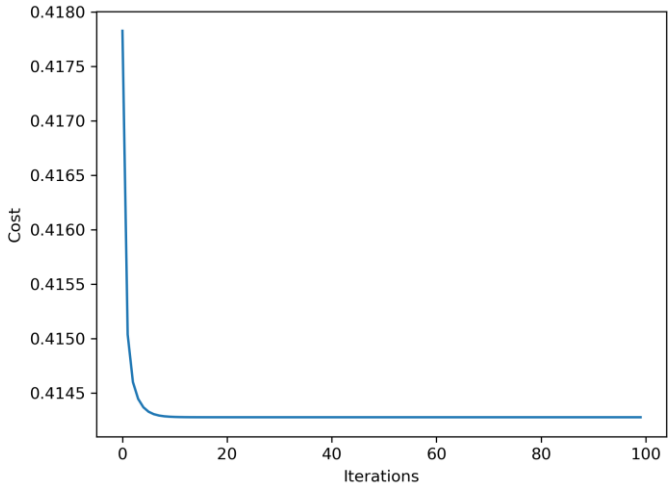


Figure 8. Progression of Cost on ml_assgn1_ex3.py

The error came to be very close to what was recorded in Task 4. Addition of new features using the existing features added more complexity to the hypothesis calculations. With the additional three dimensions, sigmoid can separate the classes better, hence the model was able to generalize better with a small dataset.

For Task 8, *ml_assgn1_ex4.py* was altered to accommodate the changes made in the features following from *ml_assgn1_ex3.py*. The following code shows the implementation:

```
(…) # code and comment

# Now add new features

new_feat1 = np.expand_dims(X[:,0] * X[:,1], axis=1)

new_feat2 = np.expand_dims(X[:,0] ** 2, axis=1)

new_feat3 = np.expand_dims(X[:,1] ** 2, axis=1)

# Not a neat way of doing it

X = np.append(X, new_feat1, axis=1)

X = np.append(X, new_feat2, axis=1)

X = np.append(X, new_feat3, axis=1)

(…) # prewritten code

# Initialise trainable parameters theta

theta = np.zeros((6))

# Set learning rate alpha and number of iterations

alpha = 5

iterations = 100

(…) # pre-written code
```

This part of the code uses *gradient_descent_training.py* to do the gradient descent operation. The only difference is that this function takes the test set into account, logging its cost through iterations as well, separately from the training set. Below is the implementation for this file:

```
(…) # imports

def gradient_descent_training(X_train, y_train, X_test, y_test, theta, alpha, iterations):

        (…) # documentation and pre-written code


    sigma = np.zeros((len(theta)))

    for i in range(m):

      ########################################

      # Write your code here
```

```
        # Calculate the hypothesis for the i-th sample of X, with a call to the "calculate_hypothesis" function
        hypothesis = calculate_hypothesis(X_train, theta_temp, i)
        ########################################/
        output = y_train[i]
        ########################################
        # Write your code here
        # Adapt the code, to compute the values of sigma for all the elements of theta
        sigma +=  np.dot((hypothesis - output),X_train[i])
        ########################################/
    # update theta_temp
    ########################################
    # Write your code here
    # Update theta_temp, using the values of sigma
    theta_temp -= alpha/m * sigma
    ########################################/
    ###############
    # copy theta_temp to theta
    theta = theta_temp.copy()
    # append current iteration's cost to cost vector
    ########################################
    # Write your code here
    # Store costs for both train and test set in their corresponding vectors
    iteration_cost = compute_cost(X_train, y_train, theta)
    cost_vector_train = np.append(cost_vector_train, iteration_cost)


    # same for testing as well
    iteration_cost = compute_cost(X_test, y_test, theta)
    cost_vector_test = np.append(cost_vector_test, iteration_cost)
    ########################################/
  print('Gradient descent finished.')
  return theta, cost_vector_train, cost_vector_test
```

For this Task, there will be two types of experiments. The first experiment will carry out progression of training and test errors with different training and test sizes (25, 50 and 75% splits). The second experiment will use the same structure; but will also add a third order polynomial

feature for the initial attributes. Note that since the training and testing data is randomly selected, these experimental results may change with each run.

The changes made in *ml_assgn1_ex5.py* is omitted as it directly as the implementation follows from the previous sections with addition of cube terms of the initial attributes as new attributes. The table hold 6-tuples which are final training cost, minimum training cost, minimum training iterations, final test cost, minimum test cost and minimum test iterations, respectively.

Table 2. Experimentation on different split percentages

Test	***20/60 (25%) Train-Test Split***	***40/40 (50%) Train-Test Split***	***60/20 (75%) Train-Test Split***
ml_assgn1_ex4.py	(0.25, 0,25, 100), (0.52, 0.44, 3)	(0.38, 0.38, 100), (0.44, 0.43, 5)	(0.45, 0.45, 100), (0.27, 0.27, 34)
ml_assgn1_ex5.py	(0.33, 0.33, 100), (0.53, 0.43, 1)	(0.37, 0.37, 100), (0.57, 0.41, 1)	(0.39, 0.39, 100), (0.46, 0.41, 5)

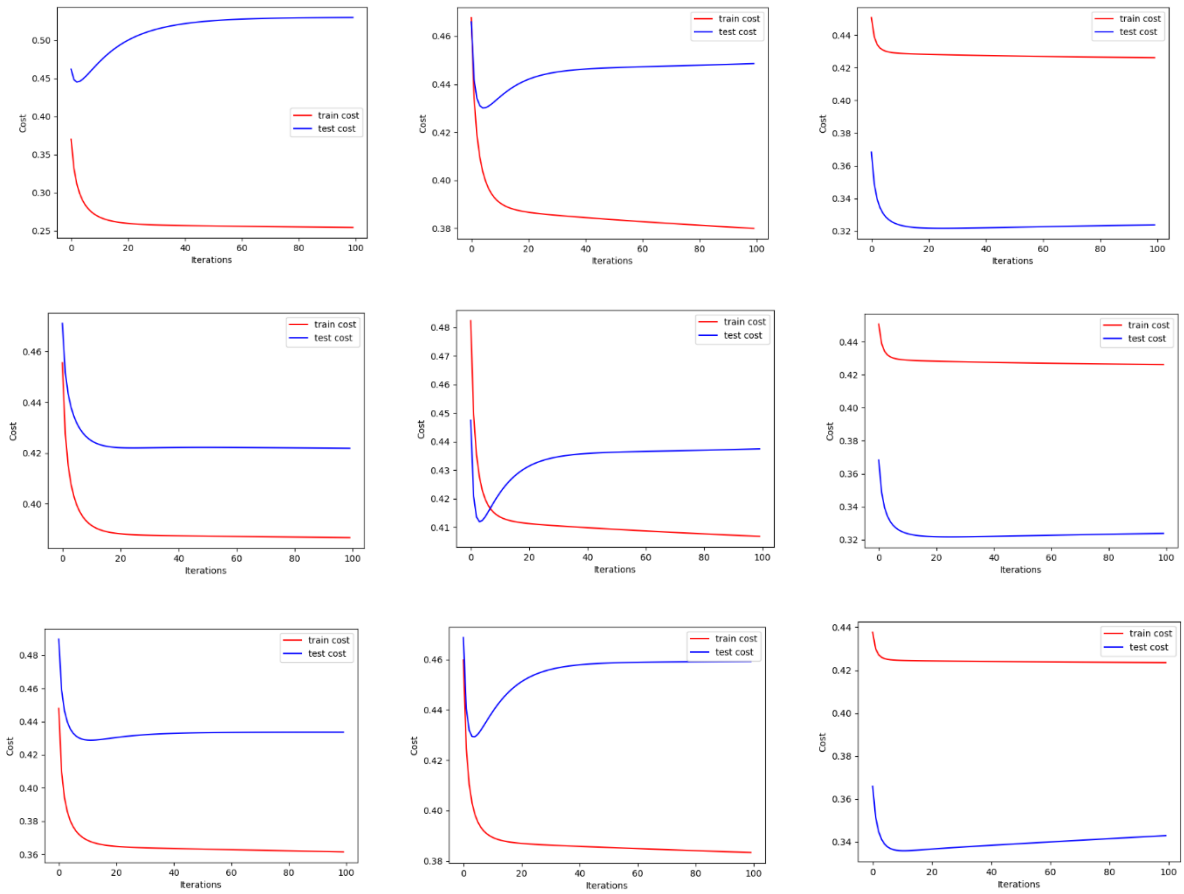


Figure 9. Training and Test Errors in 100 iterations for ml_assgn1_ex4.py. Splits are 25%, 50% and 75% from left to right for training dataset. Each row is a different trial, where table 2 shows results for the top row.

For the leftmost column, the training samples consists of 25% of the entire dataset. Since there are not many training samples; randomness of selecting point can affect the model performance. In this sense there is high variance in parameters chosen for the model. This effect is visible in the multiple runs in the column. The first test diverges; as the training samples might not be sampled good enough to be descriptive of the general model.

The middle column shows the experiments with a 50% split between training and testing data. In this setup; the model mostly overfits after some iterations. The testing cost increases while training cost decreases. This could be because of low number of training samples (40 may not be enough) or because of the features and how they are in the data space. The file *ml_assgn1_ex5.py* will investigate this issue as well.

The rightmost column has 75 % of the data reserved for training. In these experiments; most of the time; the testing cost came to be significantly lower than the training cost. In third example, there is a slight overfitting visible. However, this splitting proves to be the best among other splits. One should note that the 25 % test split corresponds to testing of 20 data points. This means each test data point correspond to 5 % in final accuracy calculations. If the number of test samples continue

to go down the reliability of the testing metrics would decrease as a single test sample contributes more to the overall testing results.

The following figure is the same with Figure 9 except that the experiments are made for *ml_assgn1_ex5.py*.

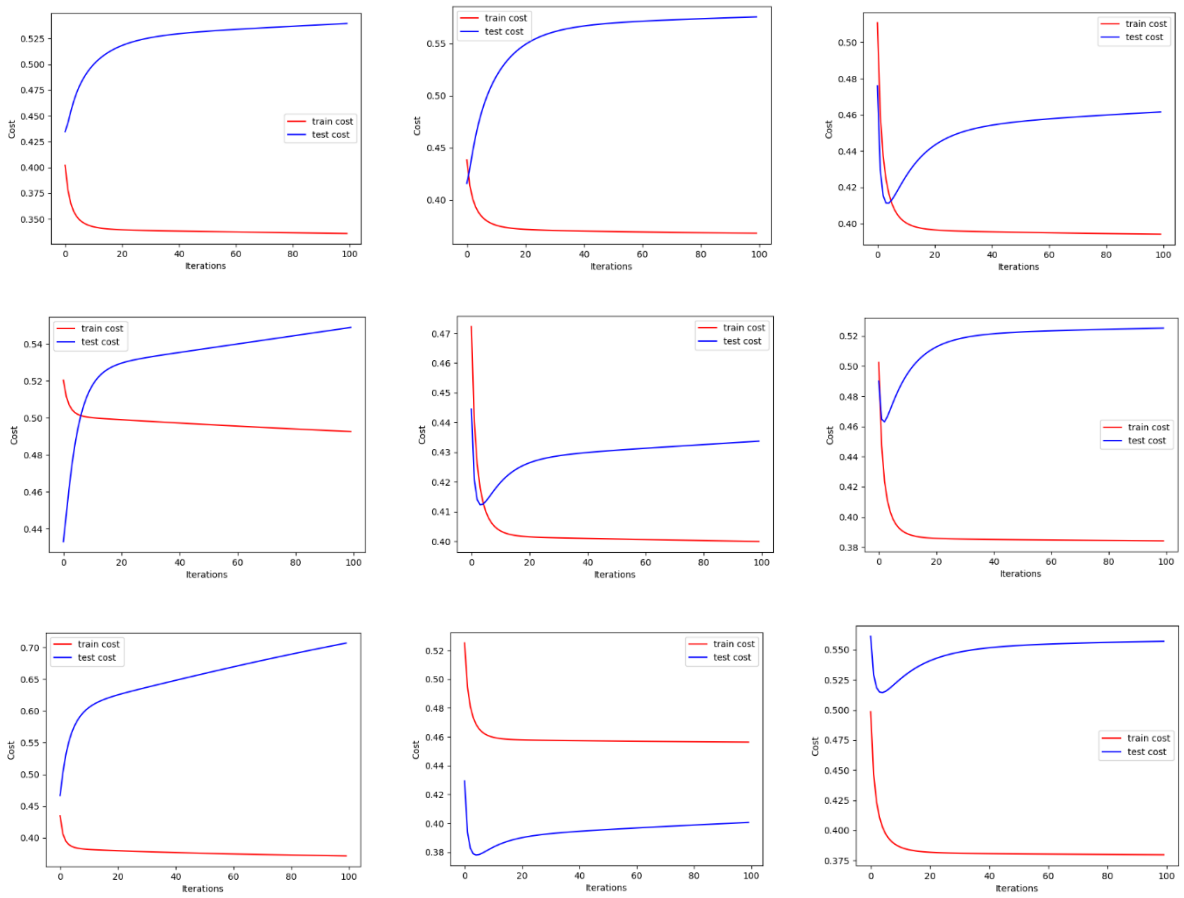
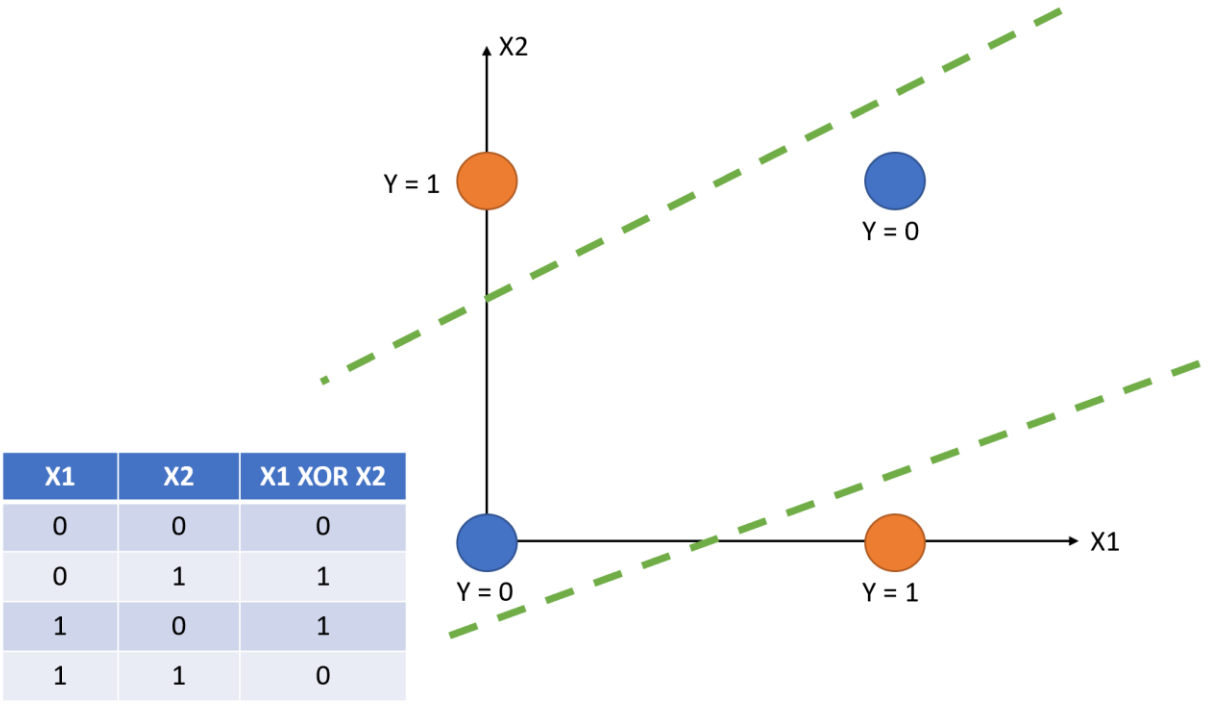


Figure 10. Training and Test Errors in 100 iterations for ml_assgn1_ex4.py. Splits are 25%, 50% and 75% from left to right for training dataset. Each row is a different trial, where table 2 shows results for the top row.

The feature space got more complex with additions made to *ml_assgn1_ex5.py*. It is clearly seen that 25% training sample split is not sufficient for training a good predictive model on the test dataset. For a 50% split; the performance of the model is still highly variant. Sometimes it is able to generalize well to the testing dataset; sometimes it fails to produce a good model. Again, this could be because of increasing complexity. As more features present in the sample columns; more examples the logistic regressor requires to define a good hyperplane separating the classes. As the number of training samples are increased to 75% of the entire dataset; each run of started to give more consistent results as seen on rightmost column of Figure 10. Here, the overall behavior is that the training cost function monotonically decreases however, test cost increases after some

point. Therefore, it can be inferred that the model starts to overfit to the training dataset after some iterations.

Task 9 is independent from the experiments done in the previous tasks. Verbally, a single logistic regression unit can only separate a space into two subspaces. The XOR solution space is organized in a way that it requires at least 3 linearly separated regions to fully describe it. Since a logistic unit cannot describe more than two subspaces; at least to logistic units are required to describe XOR classification problem. Please see the figure below for detailed explanation:



X1	X2	X1 XOR X2
0	0	0
0	1	1
1	0	1
1	1	0

Figure 11. XOR Truth Table & 2D Visualization

One green dashed line represents the decision boundary of a single logistic unit. These units linearly separate the feature space. Due to the positioning of the samples in the feature space; this feature space cannot be separated by using a single linear decision boundary. In fact; at least 2 decision boundaries; hence logistic units are required to realize this (XOR) problem.

Note that Figure 11, shows only a single way of drawing this. There are infinitely many solutions available if this space is divided into at least 3 regions.

2 Neural Networks

For this question *sigmoid.py* was modified again. The modification is the same with the previous section. The following part explains Task 10, implementation of backpropagation, with divided to sections followed by the assignment:

For reference the forward propagation formulas are given below:

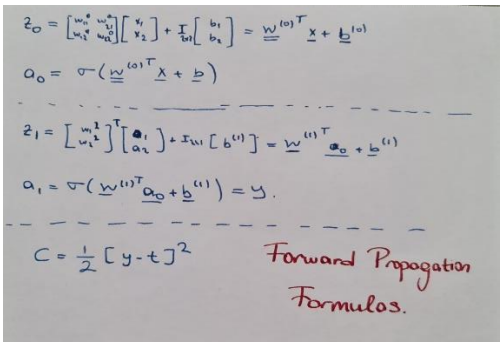


Figure 12. Forward Propagation Formulas

For the given network in the assignment the following computational graph can be drawn and using this graph the following backpropagation equations can be achieved. Note that layer (matrix) representation is used in these equations.

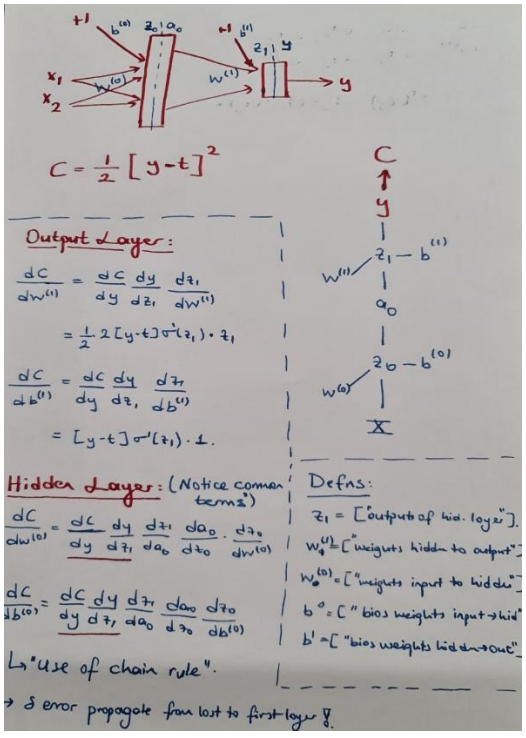


Figure 13. Computational Graph & Backpropagation Equations

Notice the common terms as the cost error propagates back to the input layer; therefore, a delta error can be defined, and this error can be kept in memory to have more efficient implementation of the algorithm. Additionally, this computational graph may be used to generalize this 1-hidden layer network to deeper architectures as well.

Below is the implementation of this network in Python; following the steps that the assignment provides:

Step 1:

As forward pass is finished; the updates should backpropagate from output to the inputs. In forward pass, each neuron takes a bias term and inputs from previous hidden layer or input layer. After dot product operation with weight vectors, they pass sigmoid activation to produce neuron output.

To backpropagate the error, the delta error at final layer is calculated by:

$$\delta_k = (y_k - t_k)g'(x_k)$$

Figure 14. Delta error for output layer

This can be realized by using the following vectoral implementation. This code will be located at location marked as "Step 1" in *backward_pass()* function. The function *sigmoid_derivative.py* is already implemented by the assignment.

```
output_deltas = (outputs-targets) * sigmoid_derivative(outputs)
```

Step 2:

At this stage, the error on the output neuron is backpropagated to the previous hidden layer. There could be one or more hidden layers available, in this case we only have one. Error on j^{th} neuron is given by:

$$\delta_j = g'(x_j)\sum_k (w_{jk}\delta_j)$$

Figure 15. Hidden Neuron Delta Error Function

This was realized by the following code segment in *backward_pass()* function:

```
# 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]
        ########################################/
    # we will remove the bias that was appended to the hidden neurons, as there is no
    # connection to it from the hidden layer
    # hence, we also have to keep only the corresponding deltas
    hidden_deltas = hidden_deltas[1:]
```

Steps 3 and 4:

After all errors are propagated back along the chain until the intended weight, the weight between j^{th} hidden and k^{th} output (or next hidden) neuron can be updated using following equation:

$$w_{jk} = w_{jk} - \eta \delta_k \alpha_j'$$

Figure 16.Weight updates using delta errors at each layer.

Note that updating weights between hidden neurons, hidden to output neurons or input weights to first initial neurons are not different. All of them use the same equation. The realization in the code is as follows:

```
self.w_out[i,j] -= learning_rate * output_deltas[j] * self.y_hidden[i]
```

2.1 Implement Backpropagation on XOR

Implementation of XOR was straight forward as the backpropagation algorithm was implemented in the previous part. For finding the best learning rate, number of iterations will be fixed to 10000, number of hidden neurons will be fixed to 2 and learning rate will be sampled from 9 equal points from 0.001 to 2. The error function with respect to iterations are shown in the grid below:

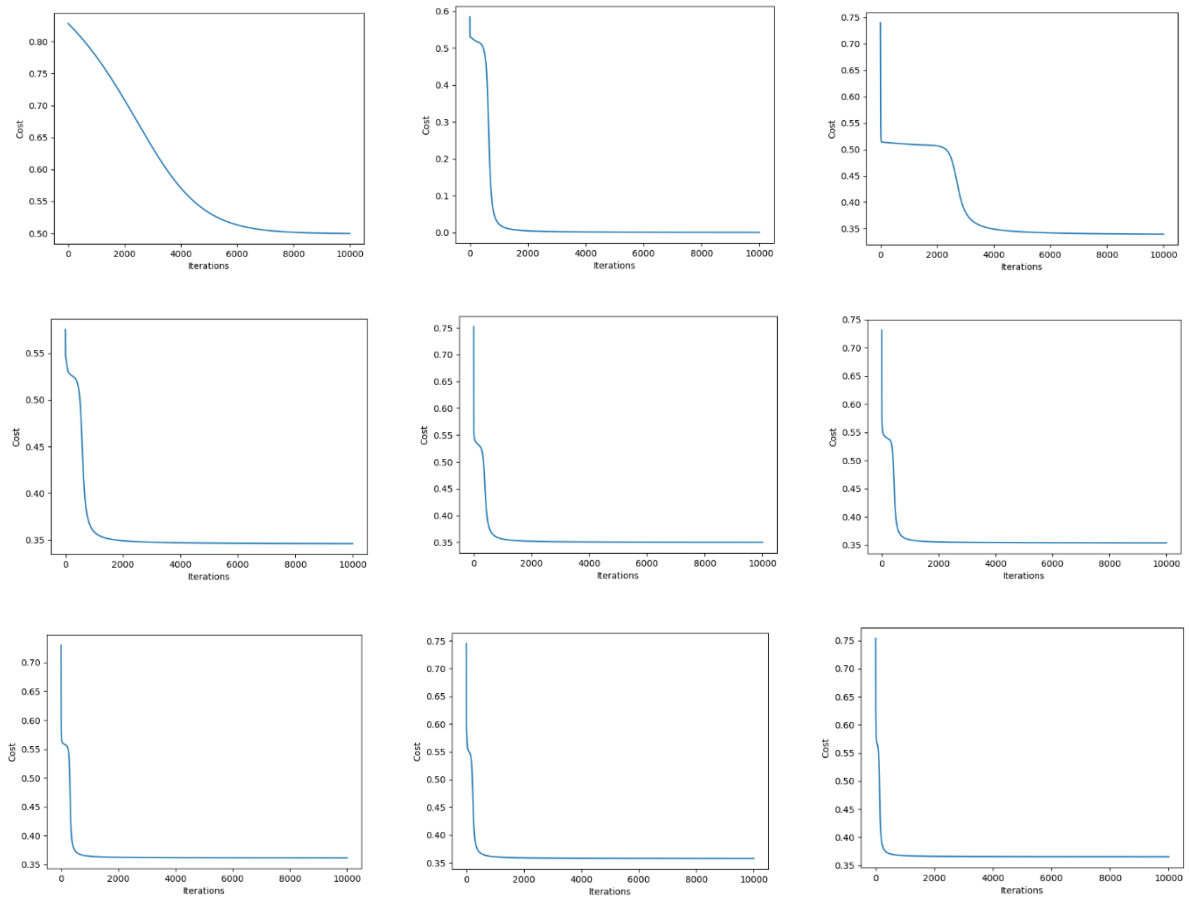


Figure 17. Graph for Error Functions (learning rate increases from 0.001 to 2 with equal spacing, shown from top left to bottom right in ascending order)

The learning rates are 0.001, 0.251, 0,500, 0.751, 1.000, 1.250, 1.500, 1.750 and 2.000 respectively. In first trial learning rate was so low that it seems that the model has converged to optimal solution however it is a local solution around a cost value of 0.50. All of the learning rates; lead to a model that stuck around a cost of 0.50 to 0.55. The models with higher learning rates had a second cost decrease which was stabilized for a longer duration.

However, note that except for the learning rate of 0.251; all these models are stuck at another local optima found with an error around 0.40 in 10000th iteration. For a learning rate of 0.251; the cost decreased under 0.1; which is significantly better than all other tests.

For Task 11, NOR and AND operations are conjugate operations. For sake of experimenting this report will focus on implementation of AND gate. AND gate is linearly separable by using a single logistic unit; therefore, using a 2-neuron system can produce a successful output.

Following from previous experience; a learning rate of 0.251 will be used for this part while also keeping the other parameters the same. For the implementation one could change the outputs to correct outputs of AND gate to use the same script:

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

After doing the following alteration to *xorExample.py* and setting the learning rate to 0.251; the following cost function is achieved:

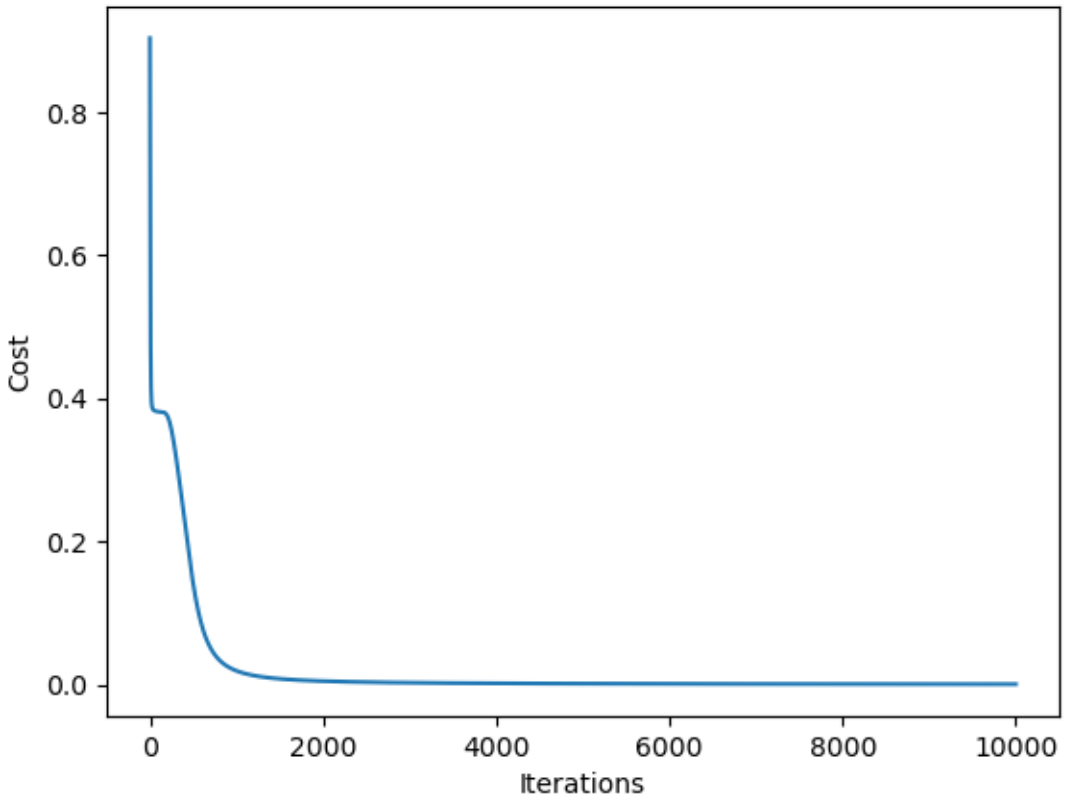


Figure 18. Error Function of the AND Gate

The minimum cost is achieved to be 0.00053 at the last iteration. As the logic space of the AND gate is linearly separable (Unlike XOR, see figure 11 for further information), a two-neuron neural network is more than sufficient to fully discriminate the 0 and 1 labeled regions from each other.

For reference entire *xorExample.py* script can be found below along with changes made on it:

```
(…) #imports
figures_folder = os.path.join(os.getcwd(), 'figures\\andExample')
if not os.path.exists(figures_folder):
   os.makedirs(figures_folder, exist_ok=True)
X = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]
              ])
y = np.array([0, 1, 1, 0])
n_hidden = 2
iterations = 10000
# learning_rate = [0.1,0.251]
# Modify to automatically do tests on XOR
learning_rate = np.linspace(0.001, 2, 9)
for i in range(len(learning_rate)):
   # Train the neural network on the XOR problem
   # For now, we will not use the 3rd and 4th outputs of the function, hence we use "_" on the returned outputs
   errors, nn, _, _ = train(X, y, n_hidden, iterations, learning_rate[i])
   # Test the neural network on the XOR problem
   test_xor(X, y, nn)
   # Plot the cost for all iterations
   fig, ax1 = plt.subplots()
   plot_cost(errors, ax1)
   plot_filename = os.path.join(os.getcwd(), 'figures\\andExample', 'XOR_cost_lr' + str(learning_rate[i]) + '.png')
   plt.savefig(plot_filename)
   min_cost = np.min(errors)
   argmin_cost = np.argmin(errors)
   print('Minimum cost: {:.5f}, on iteration #{}'.format(min_cost, argmin_cost+1))
   #plt.ioff()
   #plt.show()
```

2.2 Implement Backpropagation on Iris

This dataset has three classes to discriminate from. Each sample has 4 numerical attributes, and these attributes are mapped into three classes.

Neural networks make use of linear combinations of logistic units, layer after layer to create a non-linear complex model to divide the classes from one and the other. Ideal number of neurons required to fully solve this system would be the number of linear boundaries that are required to fully separate the class boundaries in this 4-dimensional space. (This could be proven using universal approximation theorem.)

If we were forced to use a single logistic unit; we would need to apply some kernel trick to the dataset; folding it around certain axes, adding more dimensionality which splits the classes into linearly separable clusters. The idea is to manipulate the dataset in such a way that the classes become linearly separable by using a single line (a single logistic unit).

For Task 13; *irisExample.py* will be run. A similar for loop to XOR Example will be written. Therefore, the script will consecutively try and save figures for 1, 2, 3, 5, 7, 10 neurons. The modified code is given below along with outputs and discussions regarding the results:

```
(…) # imports
figures_folder = os.path.join(os.getcwd(), 'figures\\irisExample')
if not os.path.exists(figures_folder):
    os.makedirs(figures_folder, exist_ok=True)
X, y = load_data_ex1()
# split the dataset into training and test set
X_train, y_train, X_test, y_test = return_test_set(X, y)
# Compute mean and std on train set
# Normalize both train and test set using these mean and std values
X_train_normalized, mean_vec, std_vec = normalize_features(X_train)
X_test_normalized = normalize_features(X_test, mean_vec, std_vec)
hidden_neurons = [1, 2, 3, 5, 7, 10]
learning_rate = 1.0
iterations = 100
is_iris = True
for i in range(len(hidden_neurons)):
    errors, nn, cost_train, cost_test = train(X_train_normalized, y_train, hidden_neurons[i], iterations, learning_rate, X_test_normalized, y_test, is_iris)
```

```
    # Plot the cost during training for all iterations
    fig, ax1 = plt.subplots()
    plot_cost(errors, ax1)
    plot_filename = os.path.join(os.getcwd(), 'figures\\irisExample', 'Iris_cost_neurons' + str(hidden_neurons[i]) + '.png')
    plt.savefig(plot_filename)
    min_cost = np.min(errors)
    argmin_cost = np.argmin(errors)
    print('Minimum cost: {:.5f}, on iteration #{}'.format(min_cost, argmin_cost+1))
    # Plot the train & test cost for all iterations
    fig, ax1 = plt.subplots()
    plot_cost_train_test(cost_train, cost_test, ax1)
    plot_filename = os.path.join(os.getcwd(), 'igures\\irisExample', 'Iris_train_test_cost_neurons' + str(hidden_neurons[i])
+'.png')
    plt.savefig(plot_filename)
    # enter non-interactive mode of matplotlib, to keep figures open
    plt.ioff()
    plt.show()
```

The training and testing error are as follows:

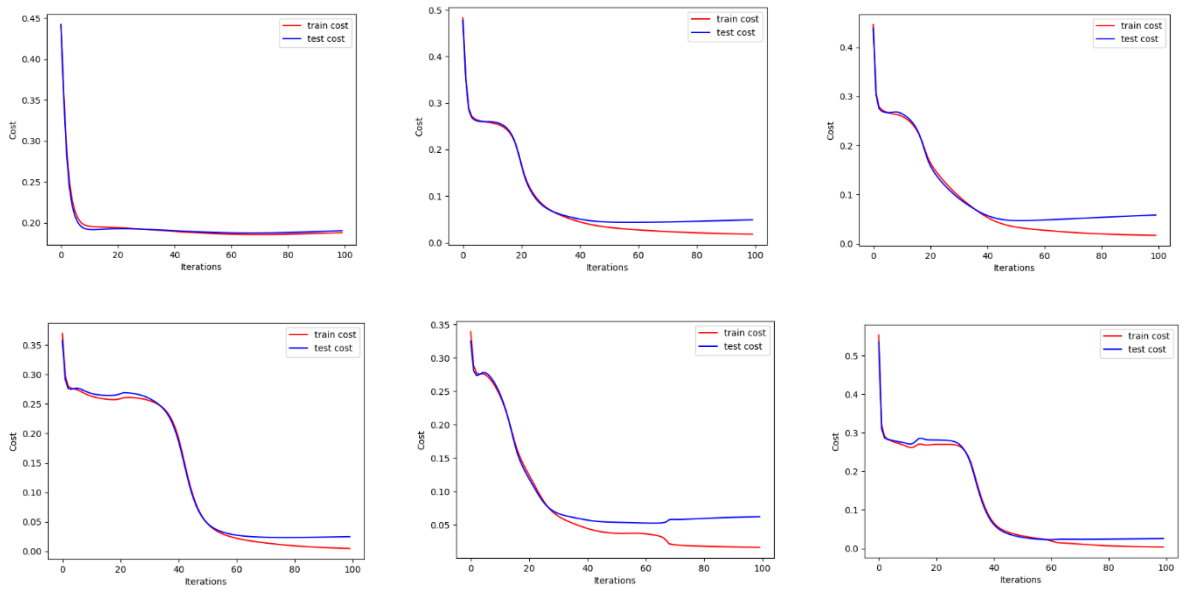


Figure 19. Train and Testing Cost Outputs

In terms of number of hidden units used: as the number of neurons in the hidden layer increase; the complexity of non-linear decision boundary, created by the linear combination of the logistic units inside hidden layer, increases. This means that the network can fit a more complex, non-linear curve between the classes. However, increasing this number caused the network to stay in a local optima for some time.

For a computational budget less than 20; it seems that using a single neuron gives the best solution with this run; as it was able to reach to a lower training and testing cost; compared to other networks. All the other networks are stuck around a local optimum around 0.3.

For larger computational budgets, and generally for this question; the best number of hidden units for the **network is 5**. In bottom leftmost graph its error outputs can be seen. Compared to other, both training and testing costs decay decent after staying at a local minimum for some time and they produce the best testing cost for the entire trials with a score of 0.05 compared to other scores which are around 0.1. The experiment with 7 hidden units also has similar results to the experiment with 5 hidden units; however, after iteration 70 it started to overfit.