Student: Rasul Mankaev (孟開達)
ID: 312540007

Deep Learning (Fall 2023)

Homework 1

Introduction

This report details the construction of a deep neural network model from scratch, emphasizing the understanding of backpropagation and stochastic gradient descent. The accompanying compressed file (hw1 312540007.zip) contains the source code and results, displaying the methodology and outcomes of this hands-on exercise.

There are 3 files (source code) attached to the archive. The first file (part1) refers to the first task - regression. The second (part2) and third (part2-batching) files relate to the second task (classification). The difference between them is that in file part2 only regularization is used, and in file part2-batching batching of the input dataset is also added.

1) Regression

1. Network Architecture

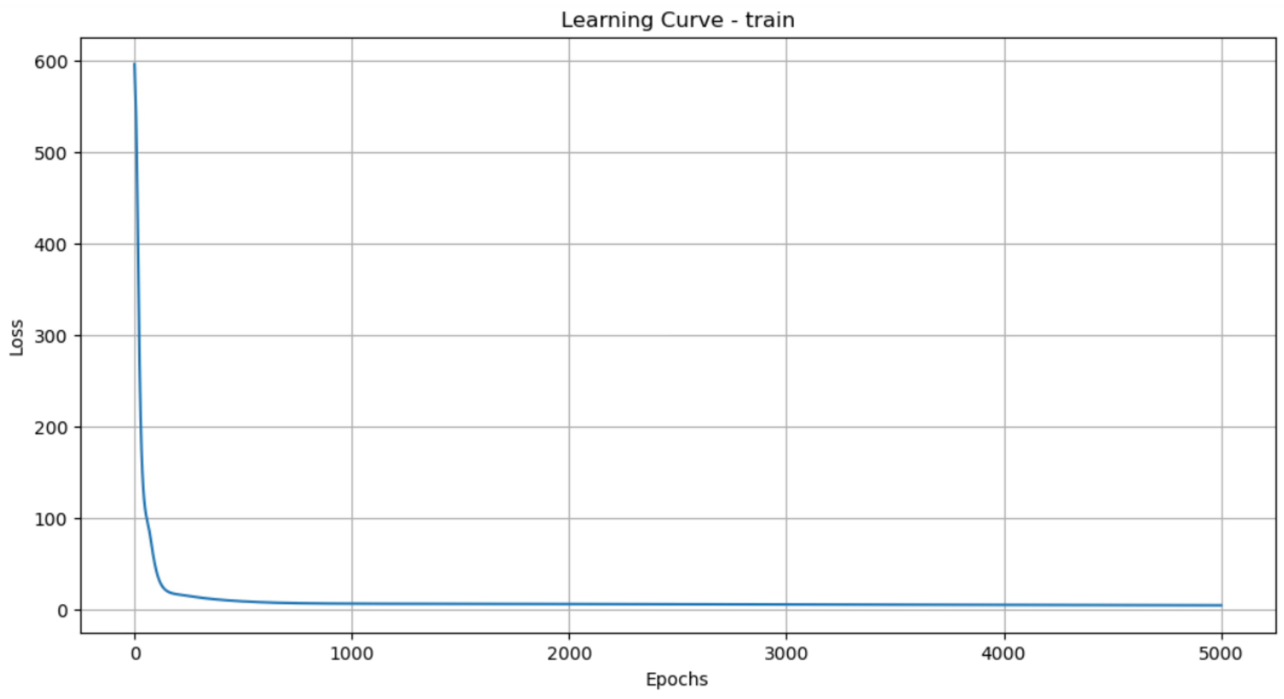
The neural network consists of:

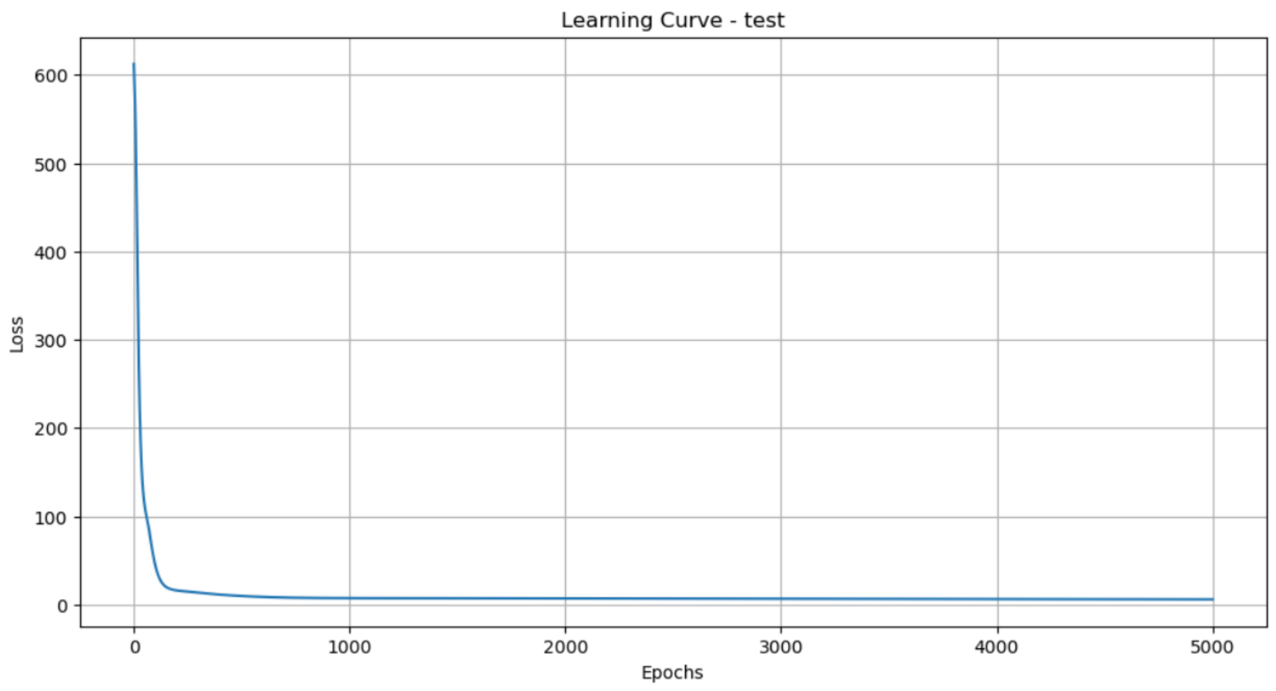
- First layer with 8 neurons. The activation function for this layer is **tanh** (hyperbolic tangent).

  * I also tried the **sigmoid** activation function. The results for the same input data turned out to be a little worse, so I decided to focus on the tangent function (details in Appendix 1)
- Output layer with a size equivalent to the target variable's dimension (determined by the shape of y_train).

2. Learning Curve

The learning curve displays how the model's loss evolves over the training epochs. The model started with a high loss of around 596.06, and after 5000 epochs (with a learning rate of 0.005), the loss was reduced to approximately 5.06. This indicates the model has successfully learned from the training data over time.





3. Training RMS Error

Mean Squared Error (MSE) for Training Data: 5.0604185316144275

Root Mean Squared Error (RMSE) for Training Data: 2.2495374039153977

4. Test RMS Error

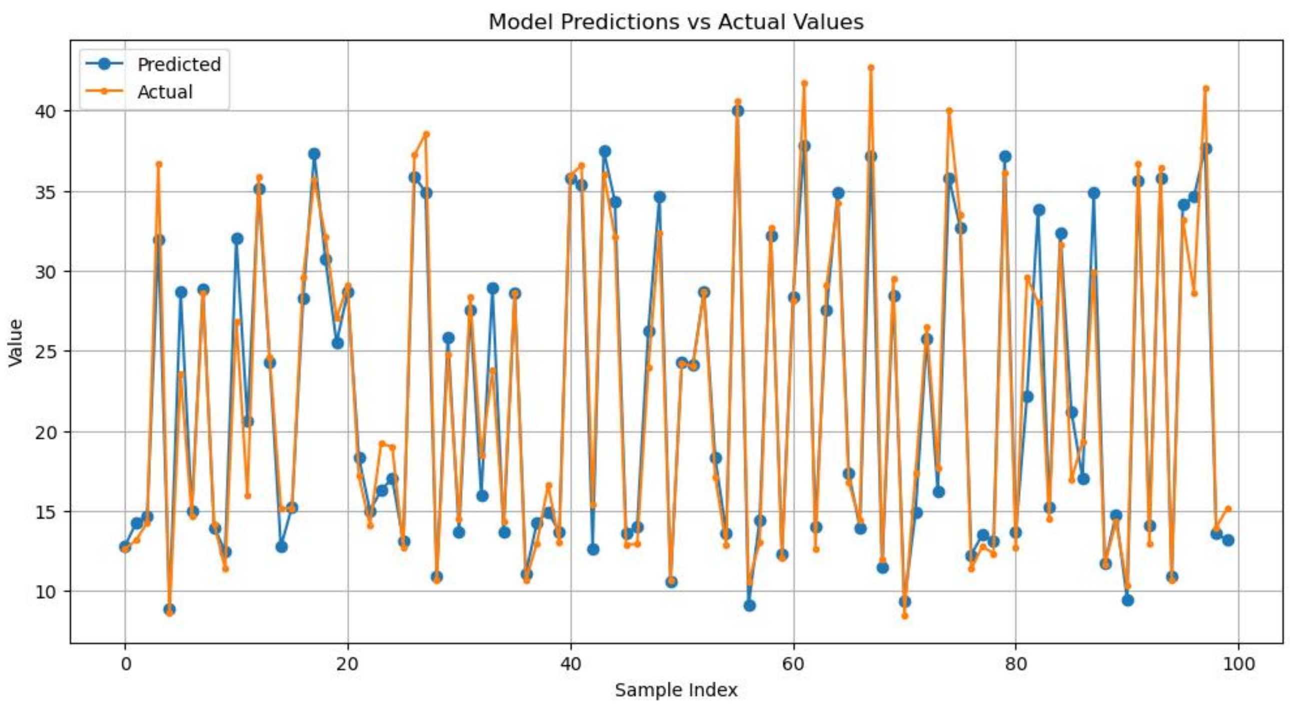
Mean Squared Error (MSE) for Test Data: 5.9060033645446035

Root Mean Squared Error (RMSE) for Test Data: 2.4302270191372255

5. Regression Result with Training Labels

The regression results for the training data are as follows for the first ten samples:

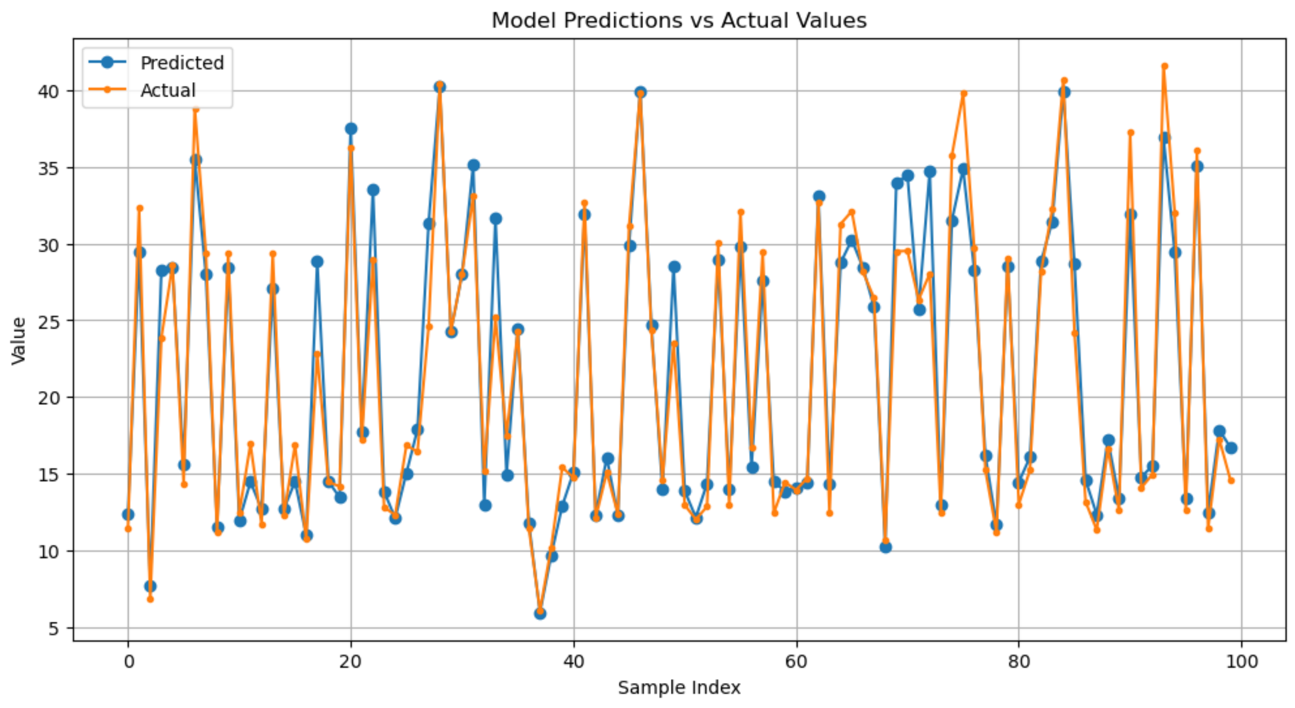
```
Predicted 1: 14.27, Actual: 13.17
Predicted 2: 14.71, Actual: 14.28
Predicted 3: 31.92, Actual: 36.7
Predicted 4: 8.90,  Actual: 8.6
Predicted 5: 28.66, Actual: 23.54
Predicted 6: 15.04, Actual: 14.71
Predicted 7: 28.82, Actual: 28.62
Predicted 8: 13.97, Actual: 14.18
Predicted 9: 12.48, Actual: 11.42
```



6. Regression Result with Test Labels

The regression results for the test data are as follows for the first ten samples:

```
Predicted 0: 12.35, Actual: 11.45
Predicted 1: 29.43, Actual: 32.31
Predicted 2: 7.74,  Actual: 6.85
Predicted 3: 28.29, Actual: 23.86
Predicted 4: 28.40, Actual: 28.6
Predicted 5: 15.61, Actual: 14.34
Predicted 6: 35.50, Actual: 38.84
Predicted 7: 28.00, Actual: 29.39
Predicted 8: 11.51, Actual: 11.2
Predicted 9: 28.43, Actual: 29.4
```



Which input features influence the energy load significantly?

To find out which input features significantly affect the energy load, I decided to do the following:

1. Fix random.seed (so that the random function does not affect our experiments)

2. Reduce the number of neurons to 1.

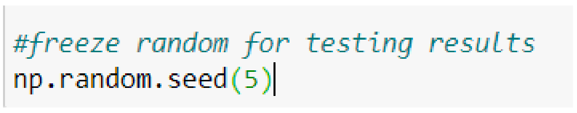
3. Output the value of the Weights connecting the input layer to the first hidden layer (W1) after training. This will give us the opportunity to find out which Weights has a largest magnitude (accordingly, a greater impact on the energy load)

4. Change the fixed value of random.seed and conduct several experiments

5. Remove from the input data by dropping the column of this value. Look at the final error value.

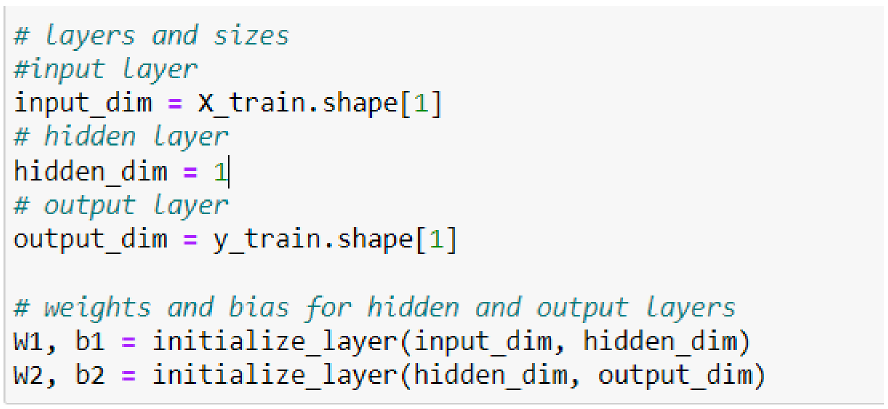
Experiments:

1. Let's assume that the value ***np.random.seed()*** will be 1, 2, 3, 4 and 5



```
#freeze random for testing results
np.random.seed(5)
```

2. Reducing numbers of neurons to 1:

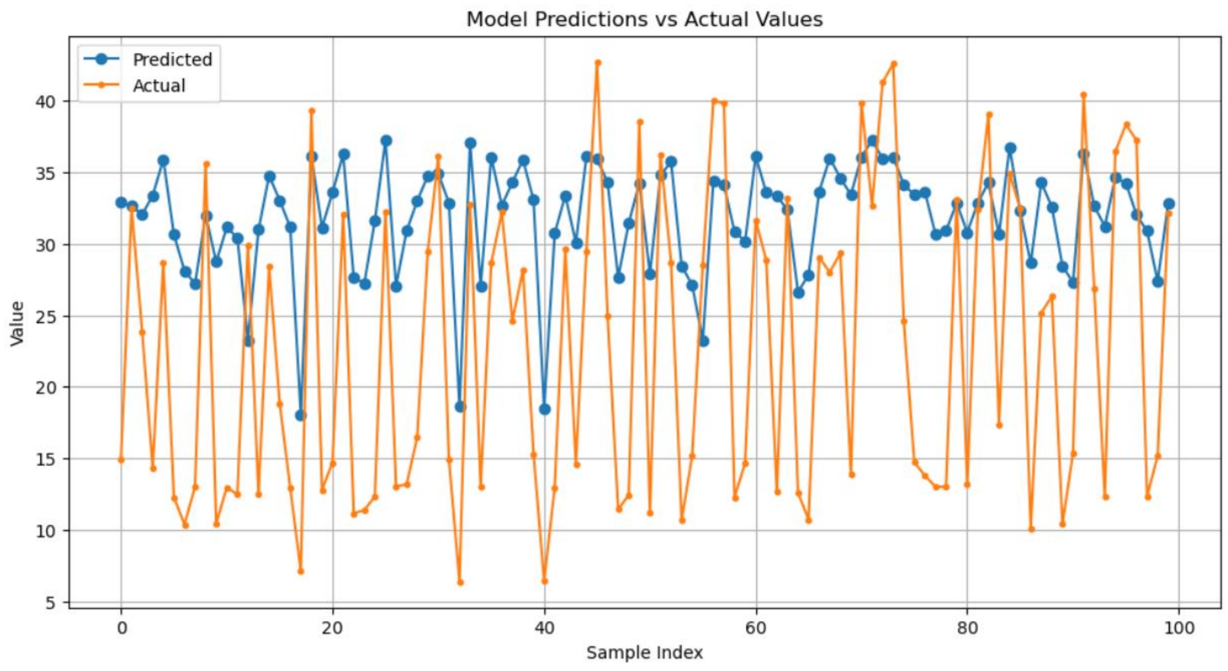


```
# layers and sizes
#input layer
input_dim = X_train.shape[1]
# hidden layer
hidden_dim = 1
# output layer
output_dim = y_train.shape[1]

# weights and bias for hidden and output layers
W1, b1 = initialize_layer(input_dim, hidden_dim)
W2, b2 = initialize_layer(hidden_dim, output_dim)
```

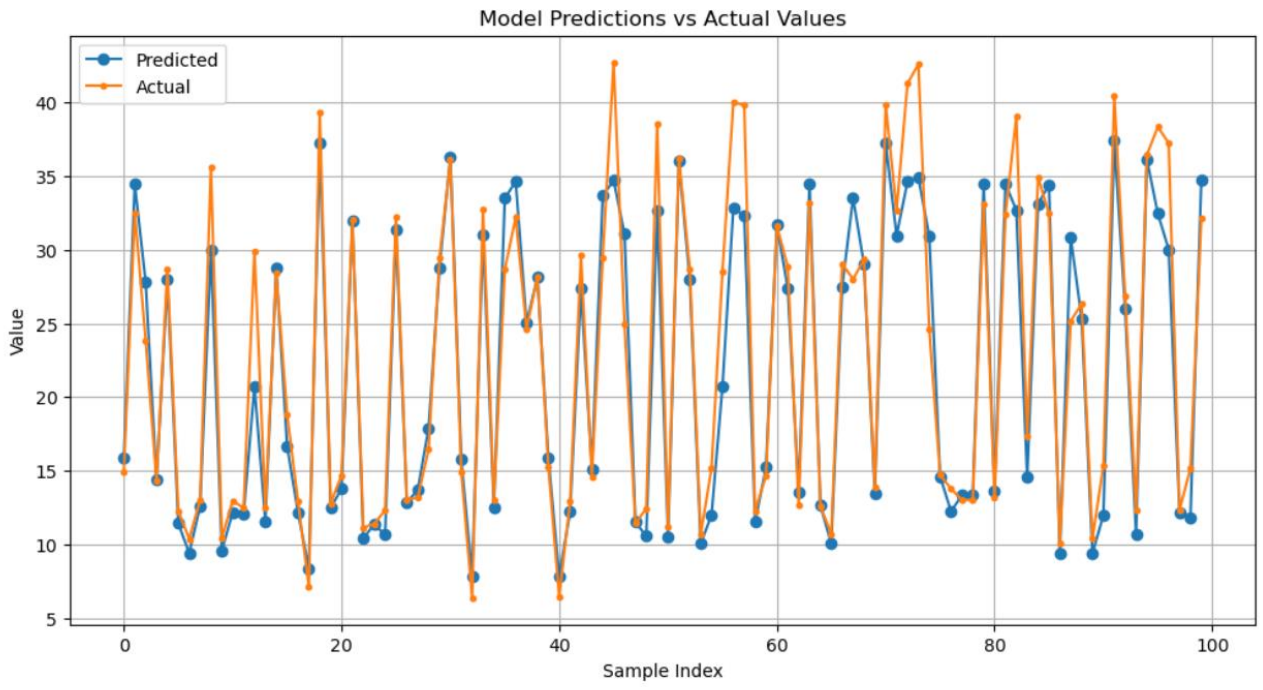
3. Outputs for different values of random.seed:

1	2	3	4	5
[[1.77111689] [0.70634518] [-0.22485226] **[1.03338085]** **[-0.9632152]** [-0.59571525] [-0.13775418] [-0.16002274] [-0.15937828] [-0.12911775] [0.27450817] [-0.12170737] [-0.10119396] [-0.10096223] [-0.10510062] [-0.09752464]]	**[[1.58043166]** [0.57341595] [-0.26481426] **[0.89491381]** **[-1.04805516]** [-0.60966817] [-0.03843698] [-0.04121063] [-0.01965023] [-0.00687438] [0.25130237] [-0.13980149] [-0.13369621] [-0.10529499] [-0.11849855] [-0.10447326]]	**[[-1.62369953]** [-0.70053339] [0.33071186] **[-0.91818697]** **[0.9775266]** [0.58740906] [0.01807542] [0.01150564] [0.00855965] [0.017809] [-0.17002586] [0.22862093] [0.2137707] [0.21013003] [0.21161092] [0.20102721]]	**[[1.57591796]** [0.28108017] [-0.04759237] **[1.09618405]** **[-1.04752671]** [-0.57056354] [-0.01340378] [-0.02296938] [-0.04042649] [-0.02322827] [0.28047305] [-0.09881163] [-0.07476266] [-0.06752638] [-0.0703584] [-0.08153291]]	**[[1.64469205]** [0.67640062] [-0.26175675] **[0.97608743]** **[-0.94629145]** [-0.56746672] [-0.18630723] [-0.18704999] [-0.17970612] [-0.17214303] [0.42905826] [-0.03458281] [-0.0185292] [-0.01621104] [-0.03614624] [0.00672187]]

So now, we will try to reset these Weights (***W1[[0,3,4],:] = 0***) and see what happens:



As we see, the result is getting worse. Now let's try to reset other Weights that deviate from zero very slightly (for example, 14, 15 and 16): ***W1[[13,14,15],:] = 0***



As we can see, the result has remained virtually unchanged. Thus, we can conclude that the greatest impact is made by the Weights (top 3) of columns numbered 1, 4 and 5:

- **Relative Compactness**
- **Roof Area**
- **Overall Height**

Student: Rasul Mankaev (孟開達)
ID: 312540007

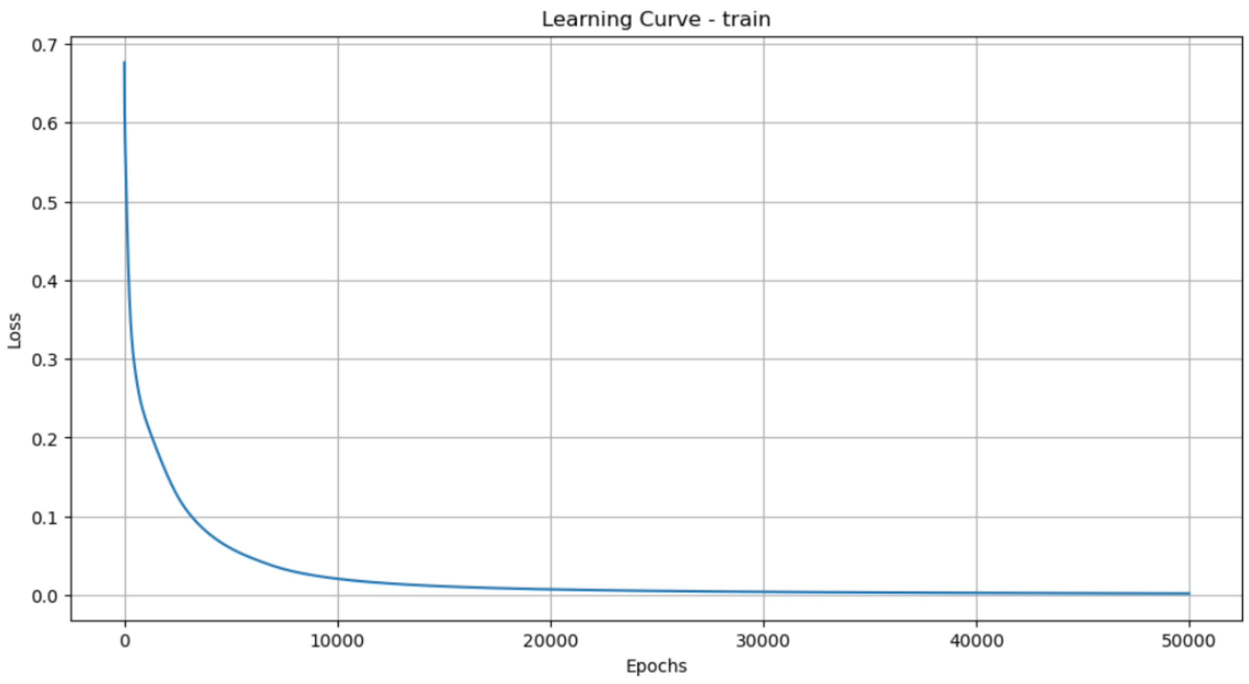
2) Classification

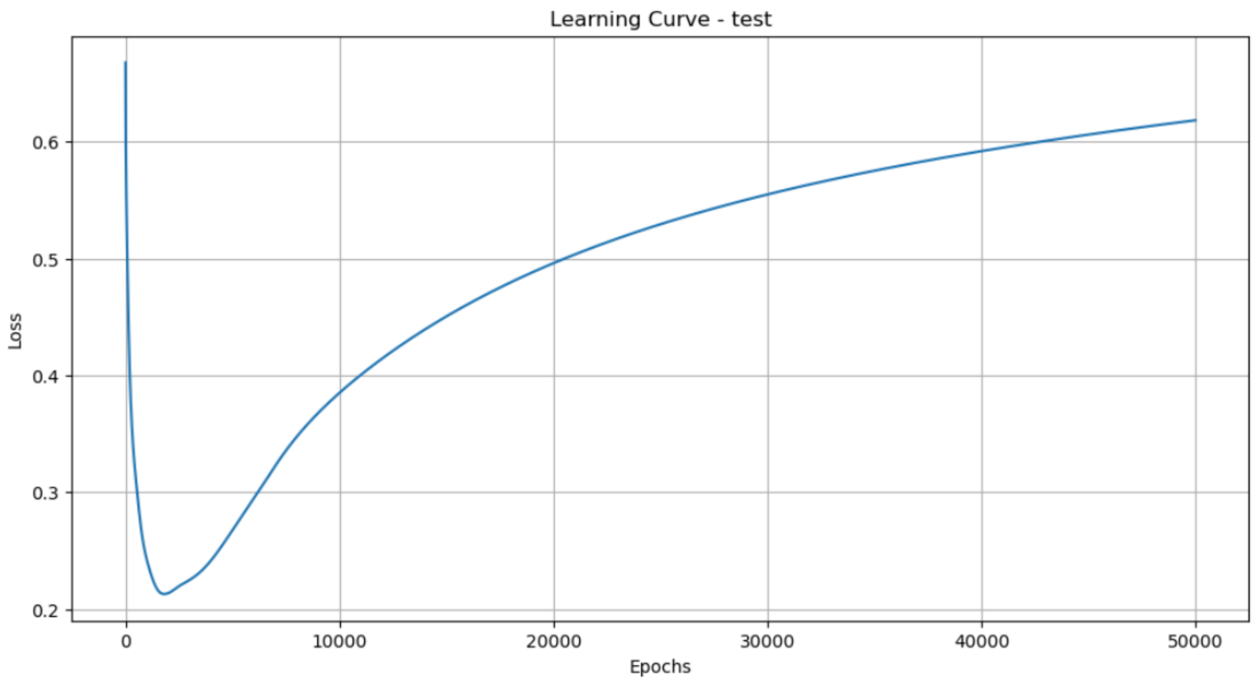
1. Network Architecture

The neural network consists of:

- First layer with 3 neurons. The activation function for the neurons in this layer is the hyperbolic tangent function (**tanh**). The tanh function outputs values in the range [-1, 1], which can help center the outputs around zero and possibly make convergence faster.
- Output Layer: Contains a single neuron. The activation function used here is the **sigmoid** function. Given the binary classification task, sigmoid activation is ideal for the output layer as it squeezes values between 0 and 1.

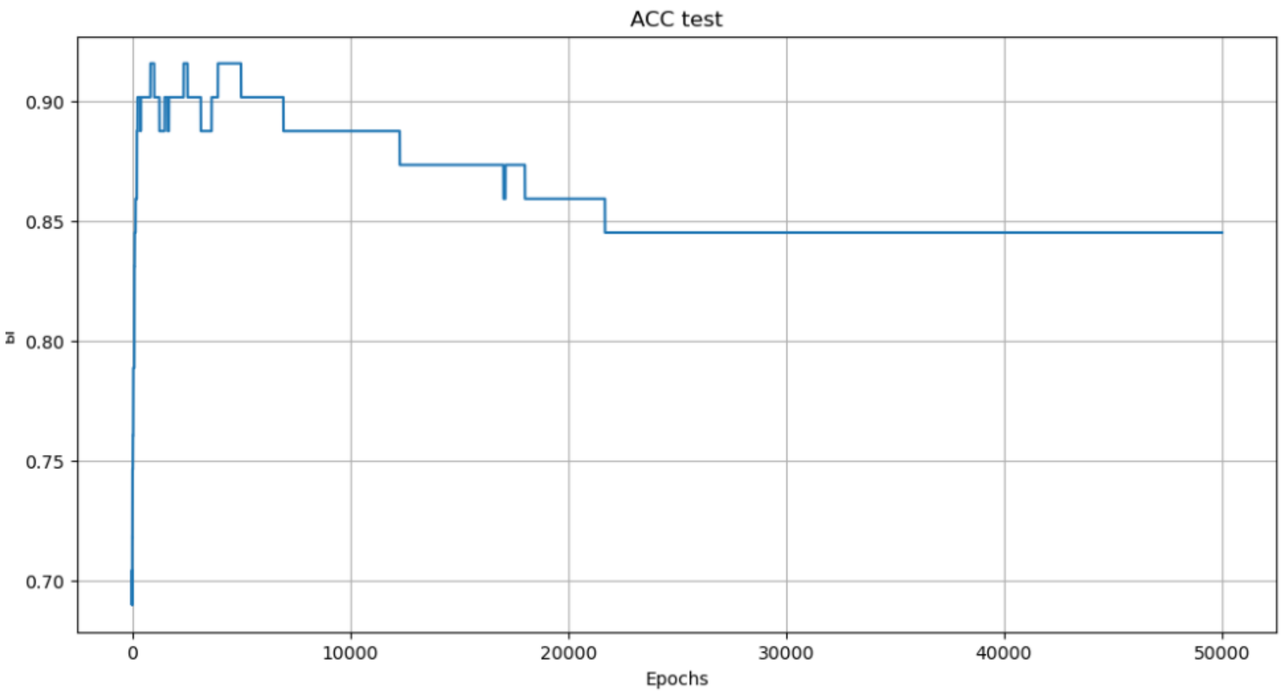
2. Learning Curve





The learning curve for both the training and testing data displays the loss value against the epochs. It provides insights into how well the model is learning and generalizing. As we can see in the test data, **overfitting** is happens (approximately after 3000 epoch with learning rate 0.05). I decided to add **weight decay** later as a possible solution to this problem.

If we look at the accuracy graph, this is also clearly visible:



3. Training Error Rate

Training BCE: **0.002121708085190773** Training Accuracy: **100.00%**

This indicates how well the model has fitted to the training data. A low BCE and high accuracy suggest good training performance.

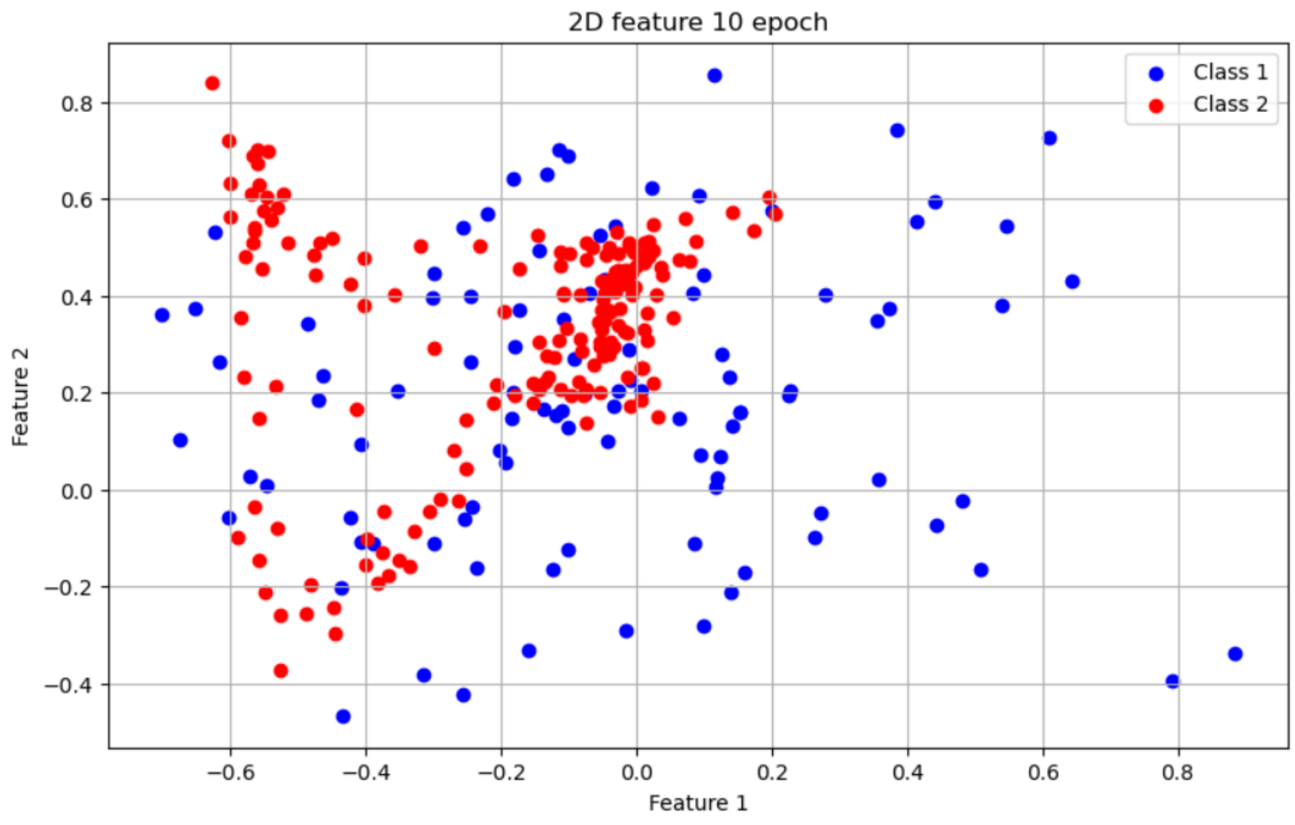
4. Test Error Rate

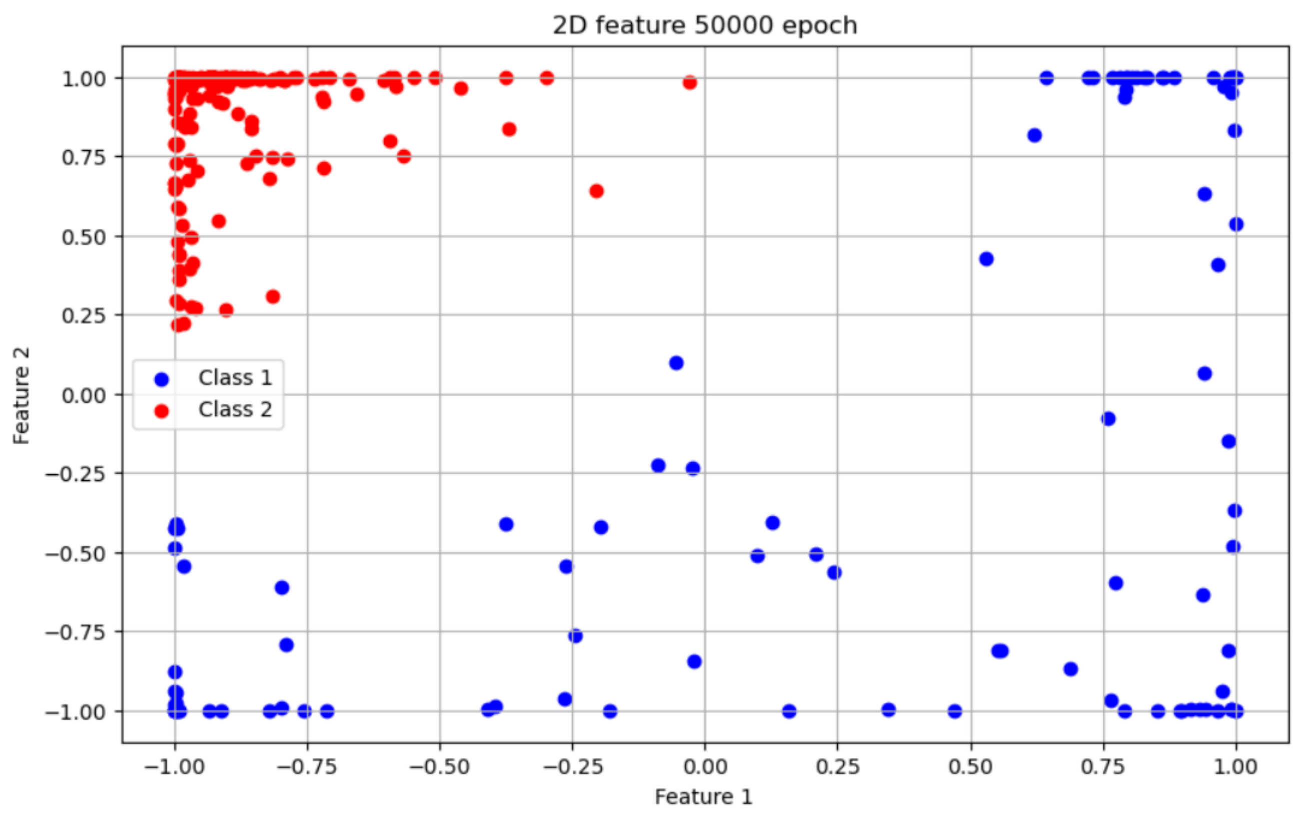
Test BCE: **0.6184380409636354** Test Accuracy: **84.51%**

This provides a measure of how well the model generalizes to new, unseen data. Nevertheless, the error is very big.

5. Feature Visualization

To understand the features learned by the network, their representations in 2D and 3D spaces are visualized at different epochs:



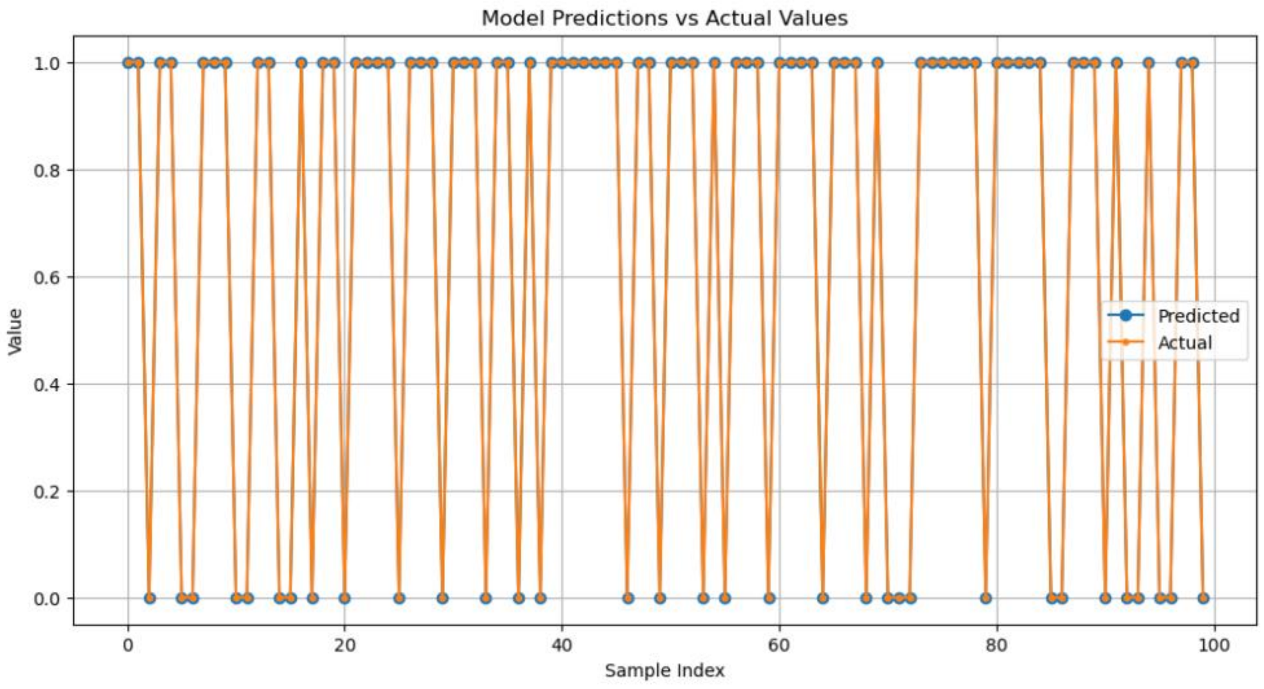


6. Model Predictions vs. Actual Values

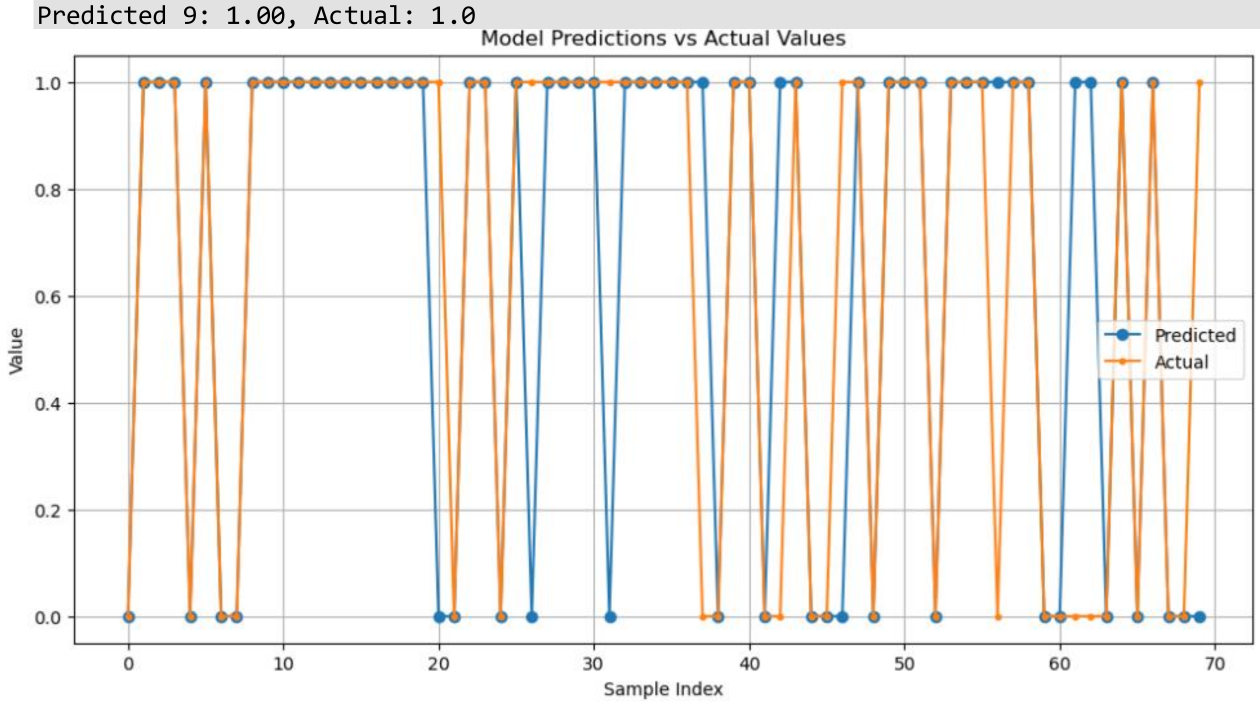
Visual representation of the predicted values against the actual values for both training and test datasets:

```
Training BCE: 0.002121708085190773
Training Accuracy: 100.00%
Predicted 0: 1.00, Actual: 1.0
Predicted 1: 1.00, Actual: 1.0
Predicted 2: 0.00, Actual: 0.0
Predicted 3: 1.00, Actual: 1.0
Predicted 4: 1.00, Actual: 1.0
Predicted 5: 0.00, Actual: 0.0
Predicted 6: 0.00, Actual: 0.0
Predicted 7: 1.00, Actual: 1.0
Predicted 8: 1.00, Actual: 1.0
Predicted 9: 1.00, Actual: 1.0
```

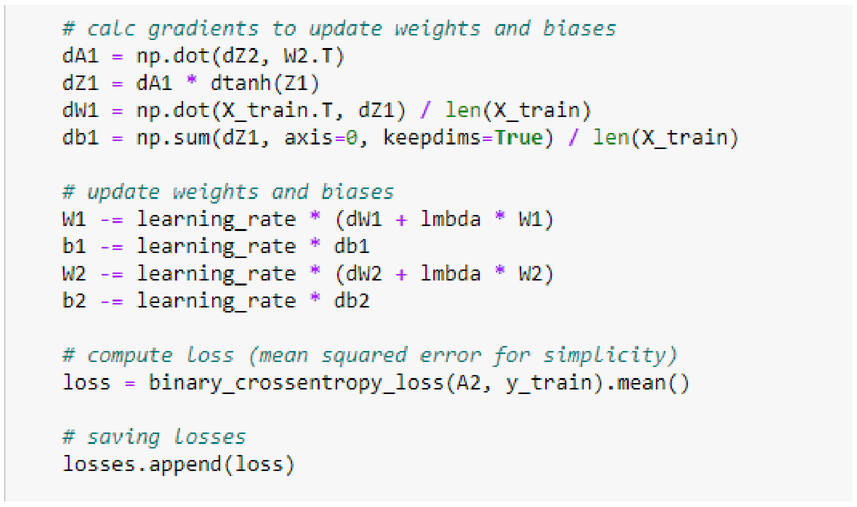
Student: Rasul Mankaev (孟開達)
ID: 312540007



```
Test BCE: 0.6184380409636354
Test Accuracy: 84.51%
Predicted 0: 0.00, Actual: 0.0
Predicted 1: 1.00, Actual: 1.0
Predicted 2: 1.00, Actual: 1.0
Predicted 3: 1.00, Actual: 1.0
Predicted 4: 0.00, Actual: 0.0
Predicted 5: 1.00, Actual: 1.0
Predicted 6: 0.00, Actual: 0.0
Predicted 7: 0.00, Actual: 0.0
Predicted 8: 1.00, Actual: 1.0
Predicted 9: 1.00, Actual: 1.0
```



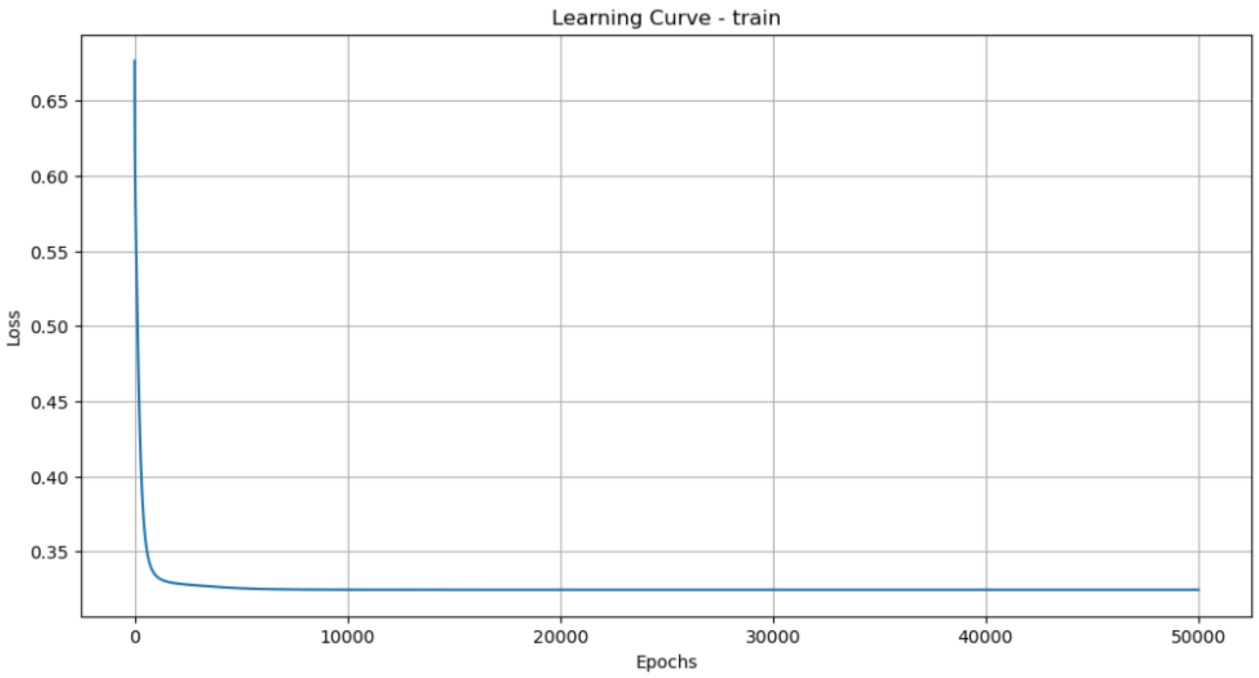
Adding weight decay

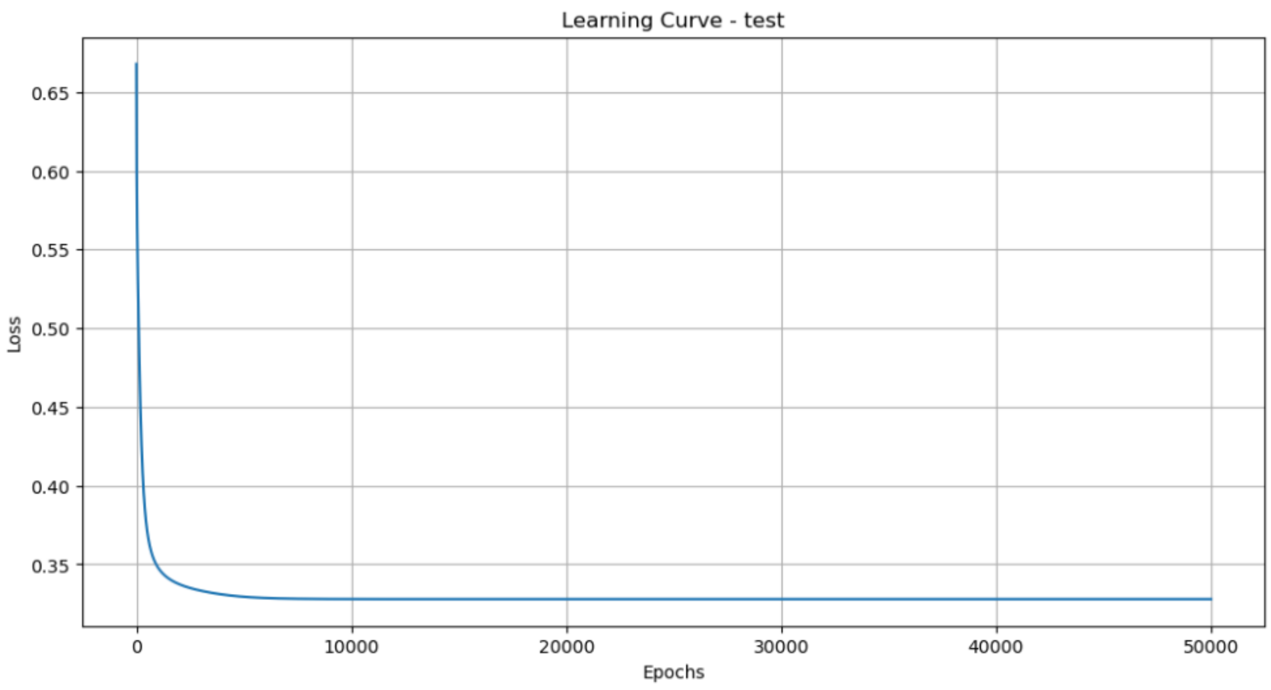


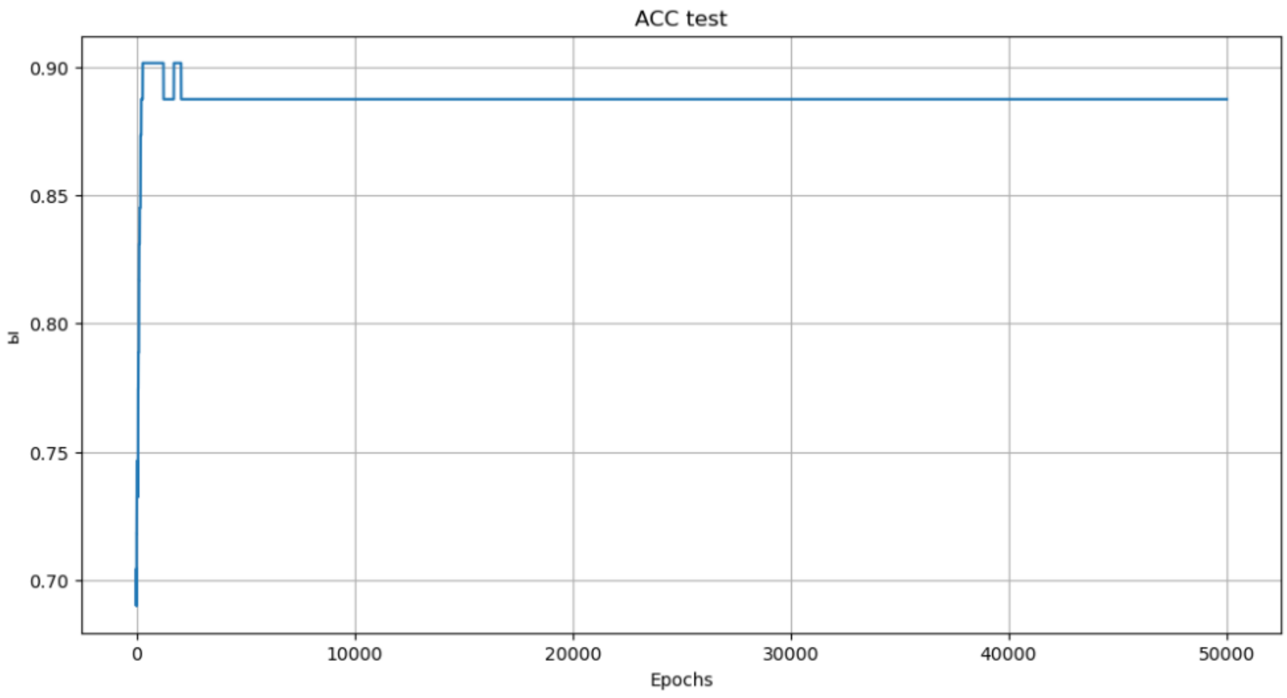
```
# calc gradients to update weights and biases
dA1 = np.dot(dZ2, W2.T)
dZ1 = dA1 * dtanh(Z1)
dW1 = np.dot(X_train.T, dZ1) / len(X_train)
db1 = np.sum(dZ1, axis=0, keepdims=True) / len(X_train)

# update weights and biases
W1 -= learning_rate * (dW1 + lmbda * W1)
b1 -= learning_rate * db1
W2 -= learning_rate * (dW2 + lmbda * W2)
b2 -= learning_rate * db2

# compute loss (mean squared error for simplicity)
loss = binary_crossentropy_loss(A2, y_train).mean()

# saving losses
losses.append(loss)
```

lmbda = 0.045

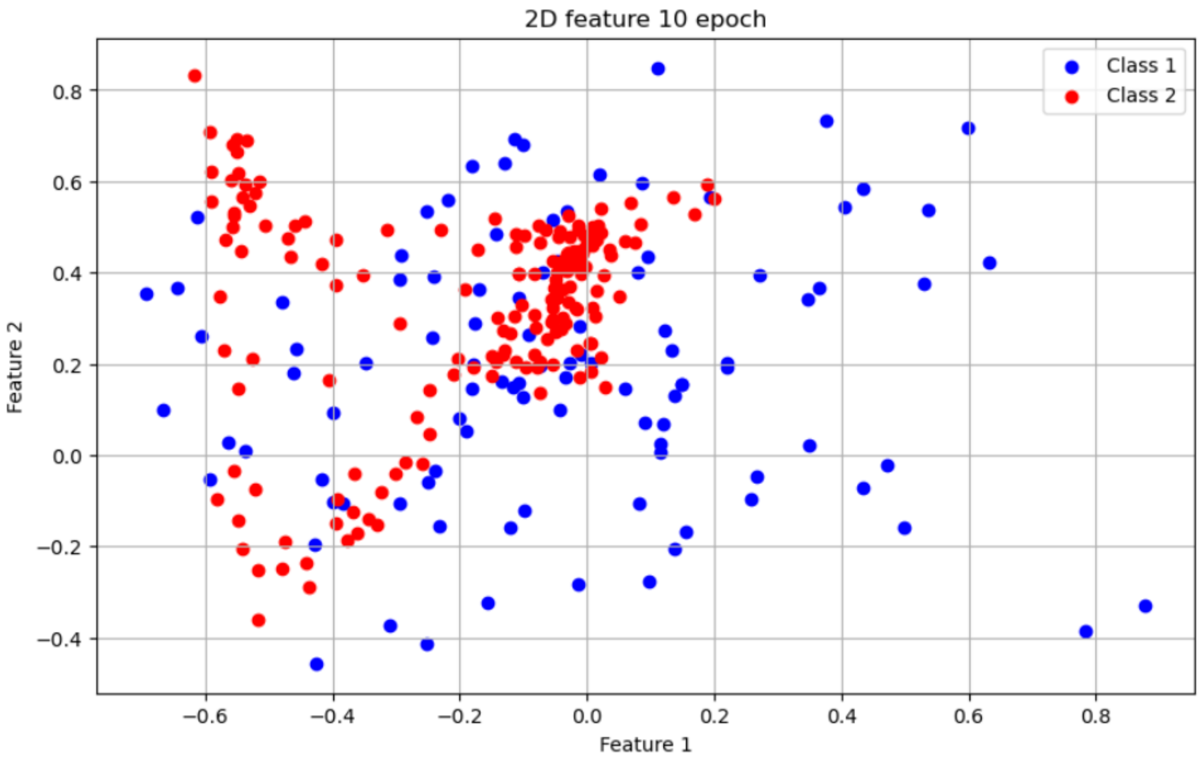
Let’s take a look at the loss, accuracy and distribution:

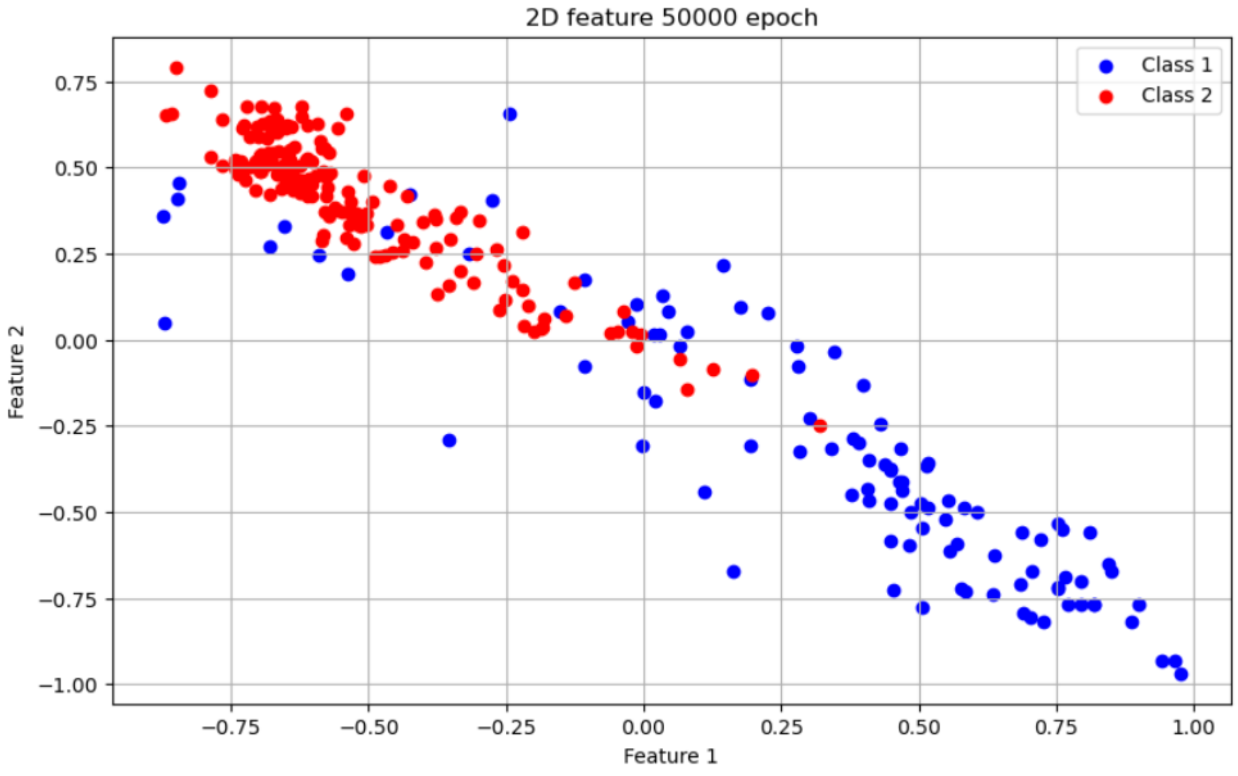



Learning Curve - test
Loss
Epochs
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0
10000
20000
30000
40000
50000

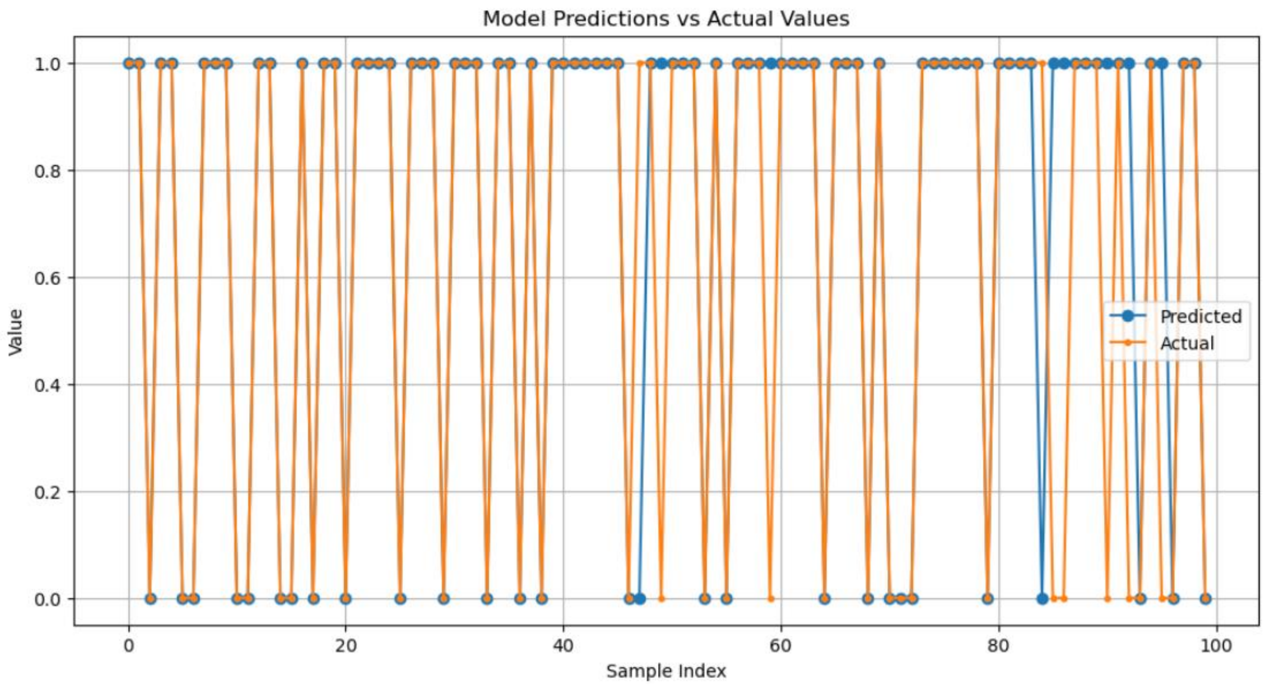

ACC test
ы
Epochs
0.90
0.85
0.80
0.75
0.70
0
10000
20000
30000
40000
50000

Student: Rasul Mankaev (孟開達)
ID: 312540007

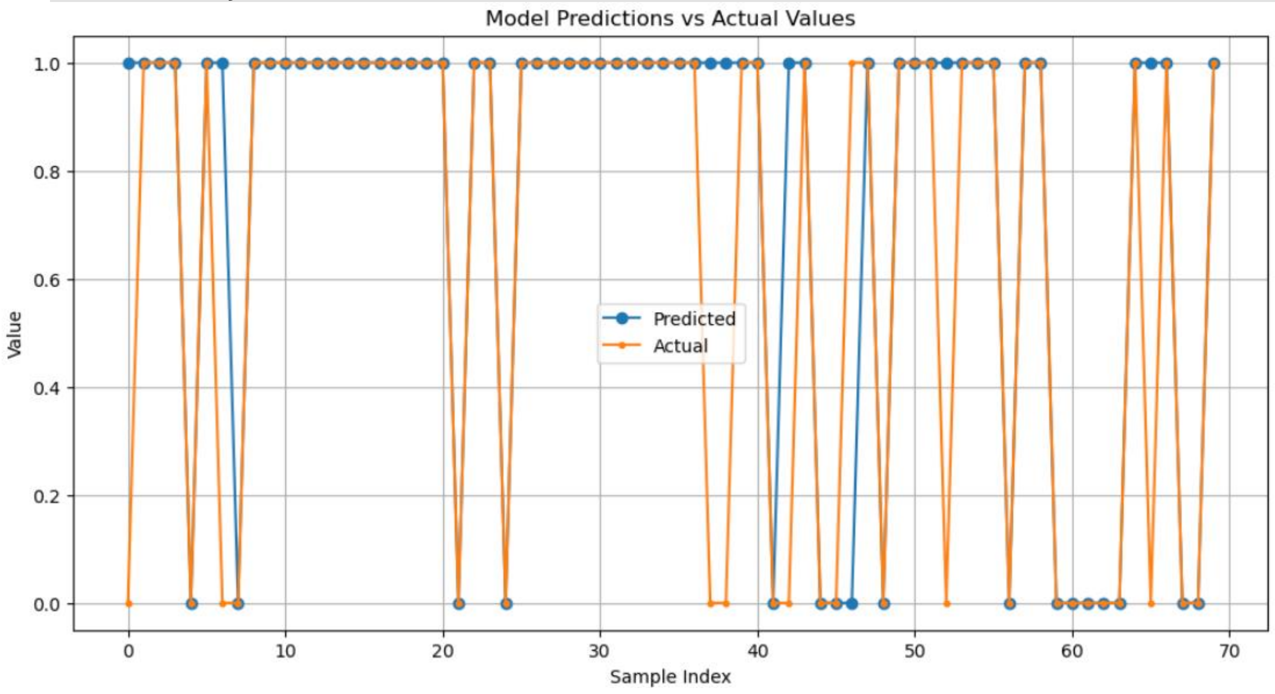




```
Training BCE: 0.32440495975195277
Training Accuracy: 89.64%
```



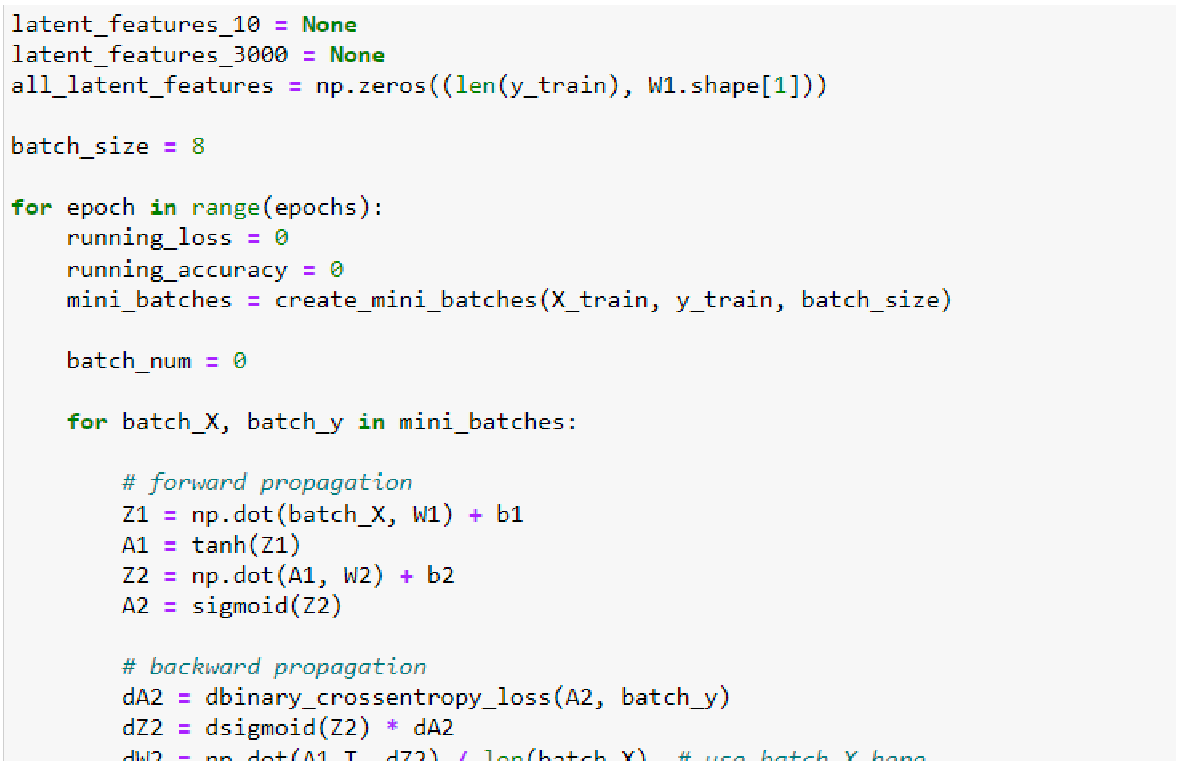
```
Test BCE: 0.32791937849383666
Test Accuracy: 88.73%
```



Looks much better, but still want to improve.

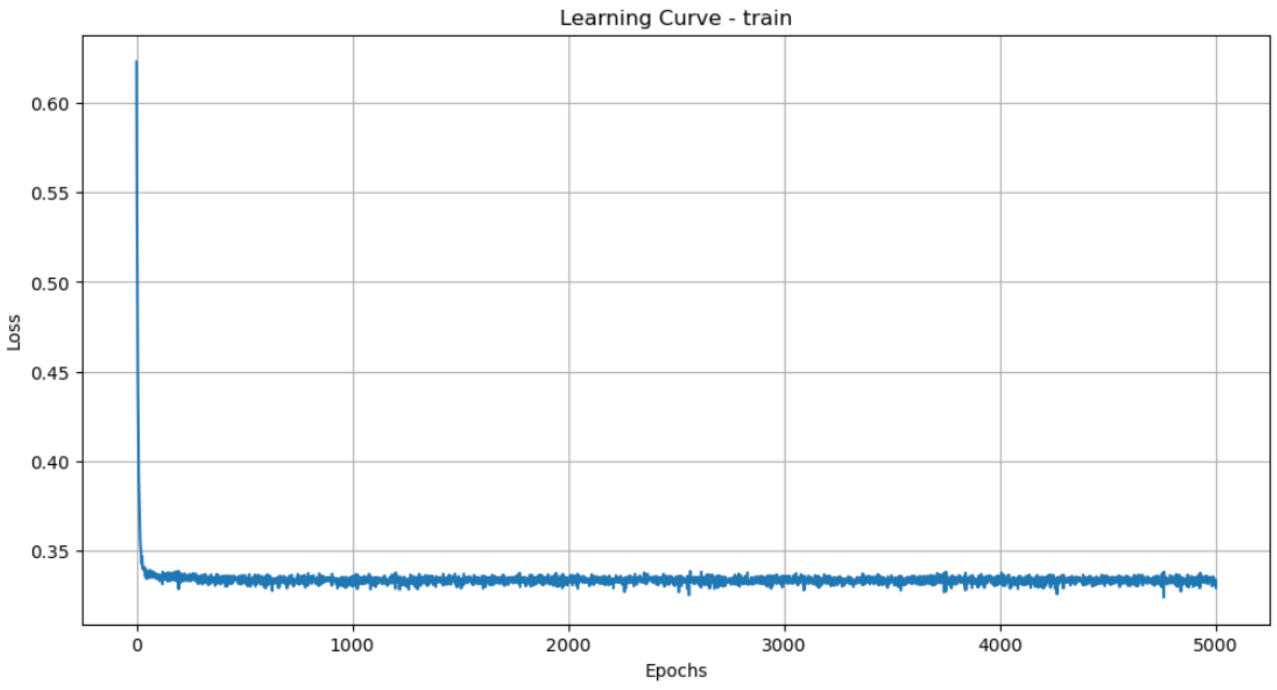
Then I decide to divide the input data and feed it in portions, which is called **batching.**

Adding batching

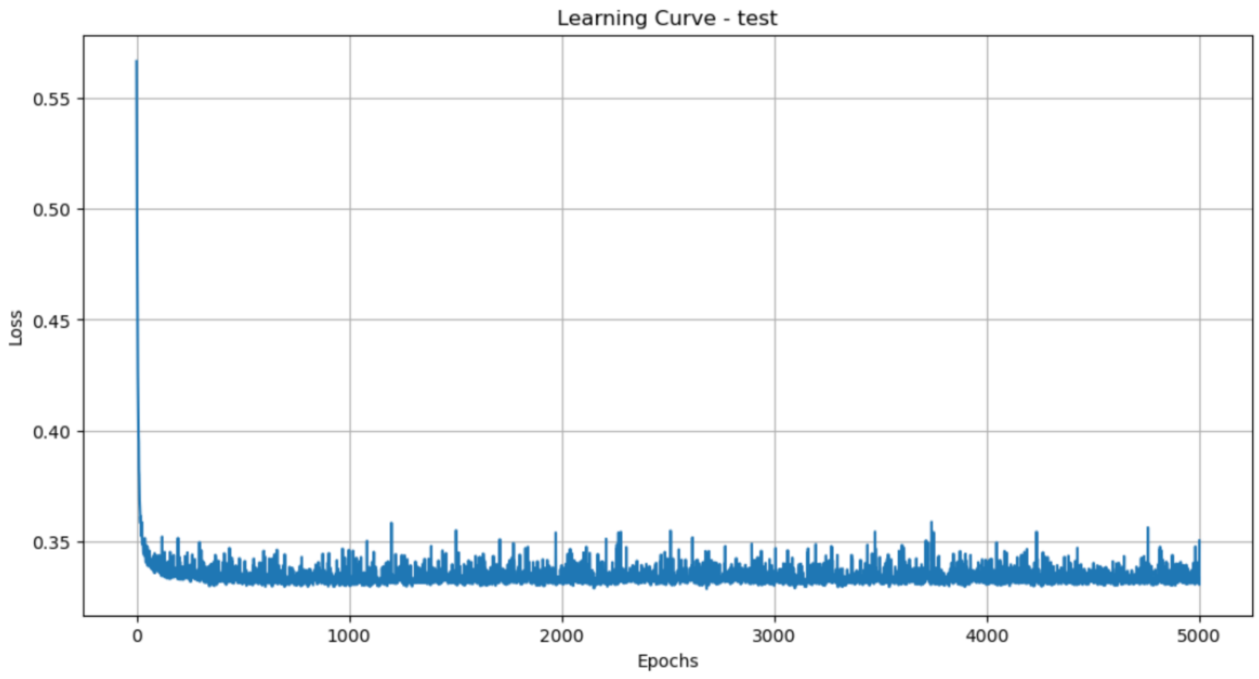


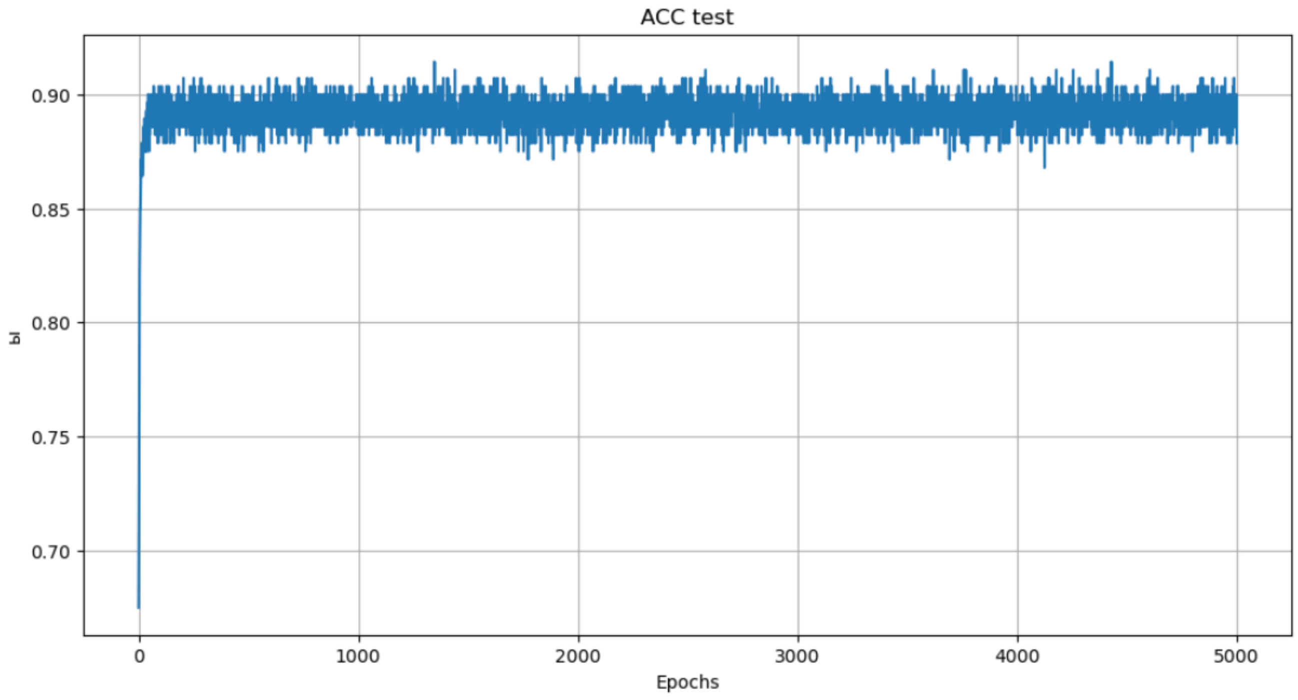
```
latent_features_10 = None
latent_features_3000 = None
all_latent_features = np.zeros((len(y_train), W1.shape[1]))

batch_size = 8

for epoch in range(epochs):
    running_loss = 0
    running_accuracy = 0
    mini_batches = create_mini_batches(X_train, y_train, batch_size)

    batch_num = 0

    for batch_X, batch_y in mini_batches:

        # forward propagation
        Z1 = np.dot(batch_X, W1) + b1
        A1 = tanh(Z1)
        Z2 = np.dot(A1, W2) + b2
        A2 = sigmoid(Z2)

        # backward propagation
        dA2 = dbinary_crossentropy_loss(A2, batch_y)
        dZ2 = dsigmoid(Z2) * dA2
        dW2 = np.dot(A1.T, dZ2) / len(batch_X)  # use batch_X here
```

In addition, I decide to lower count of epochs (to 5000), because we using batching now (350 rows, mini_batch = 8).

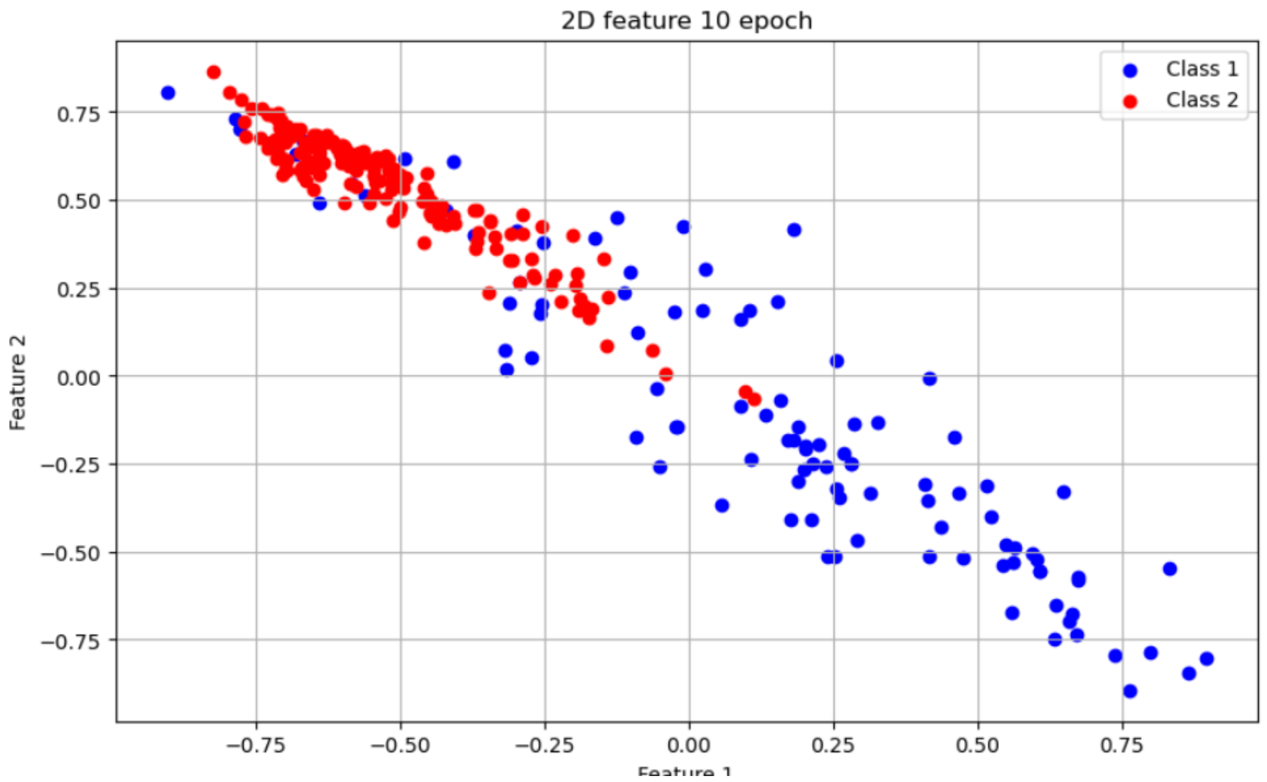


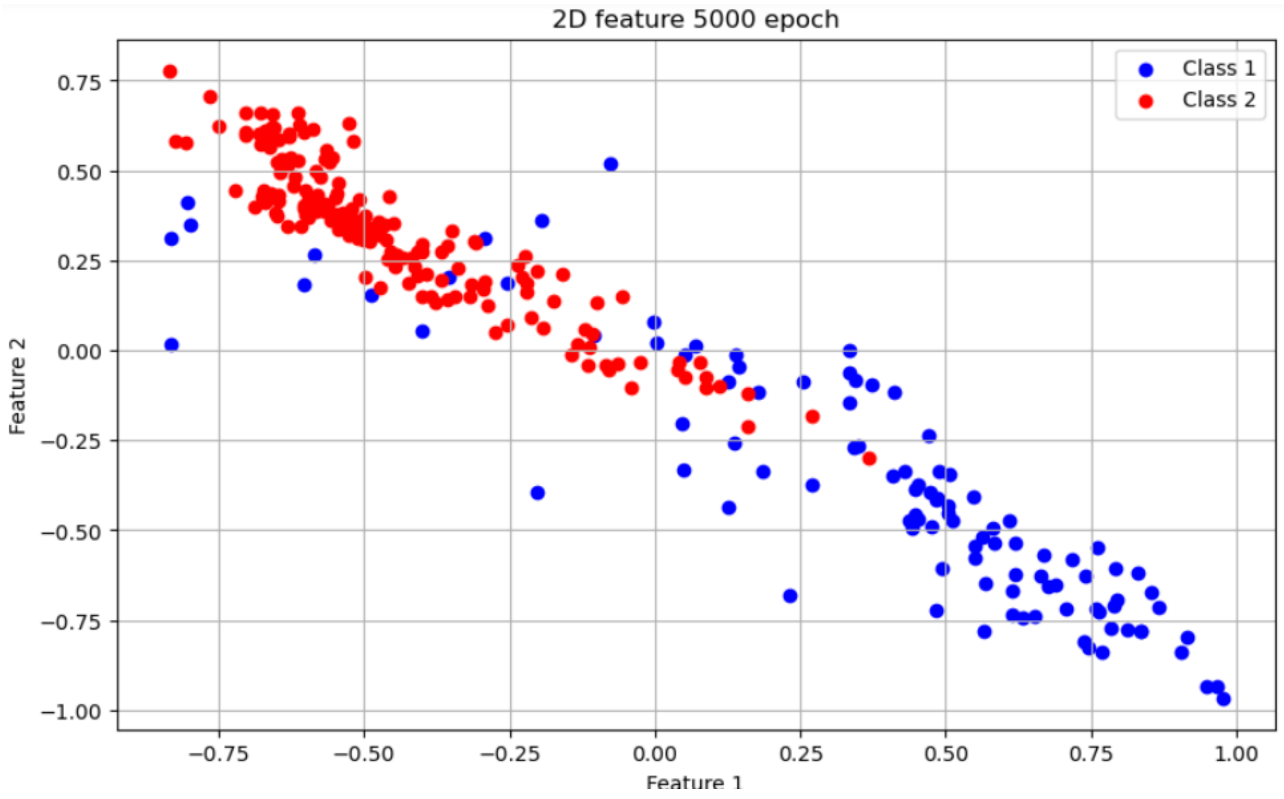
Student: Rasul Mankaev (孟開達)
ID: 312540007



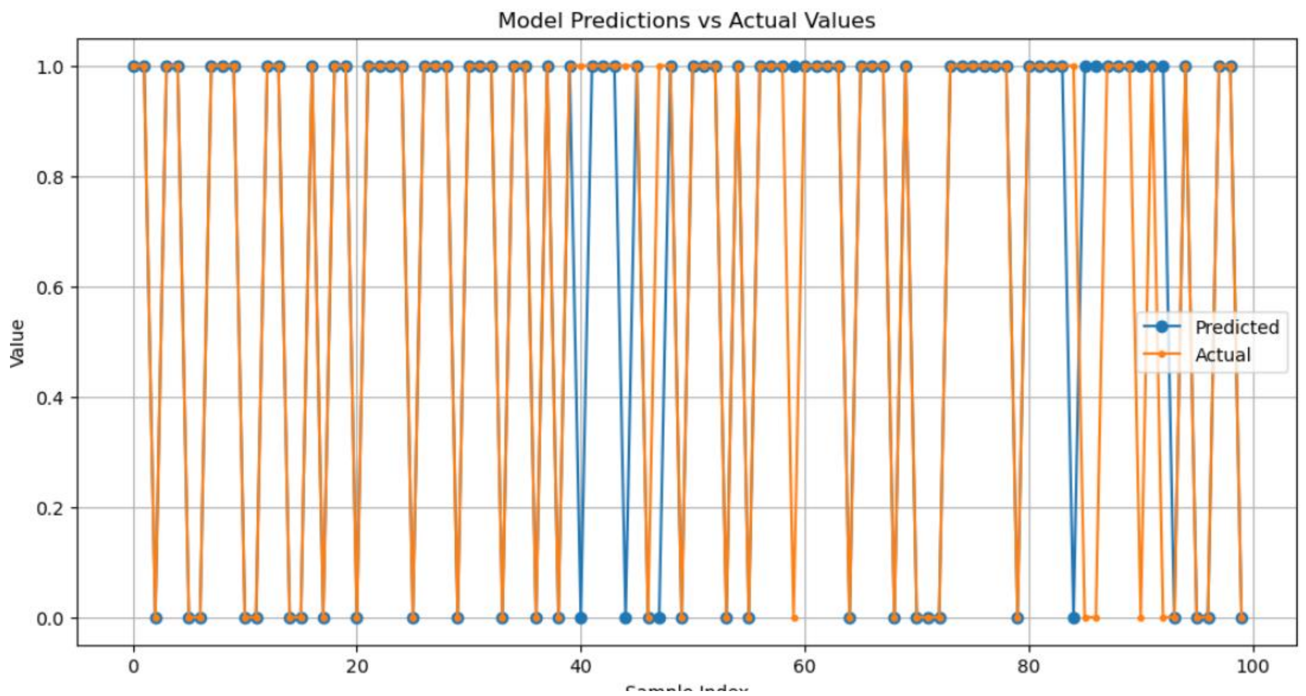


The torn graph can be explained by the fact that the data in each epoch is supplied portionwise, in batches. Therefore, it is normal for this case.

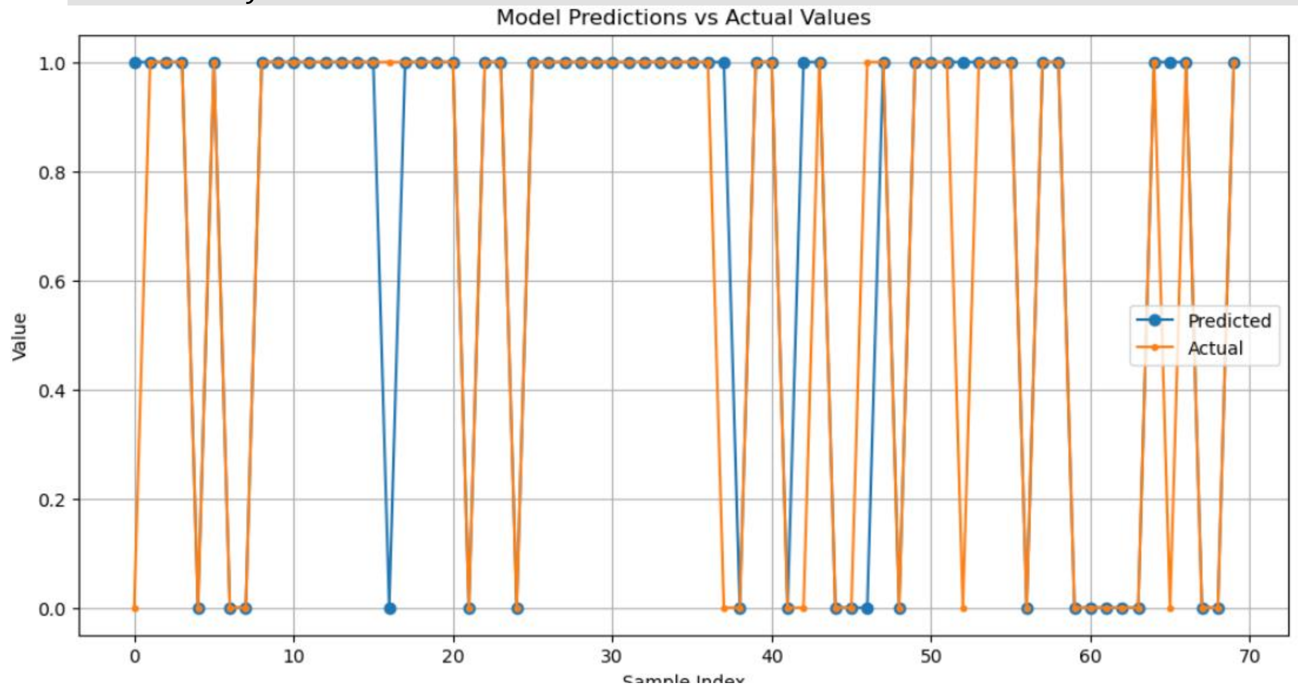




```
Training BCE: 0.3390311144060863
Training Accuracy: 90.00%
```



```
Test BCE: 0.35043703832617706
Test Accuracy: 90.14%
```



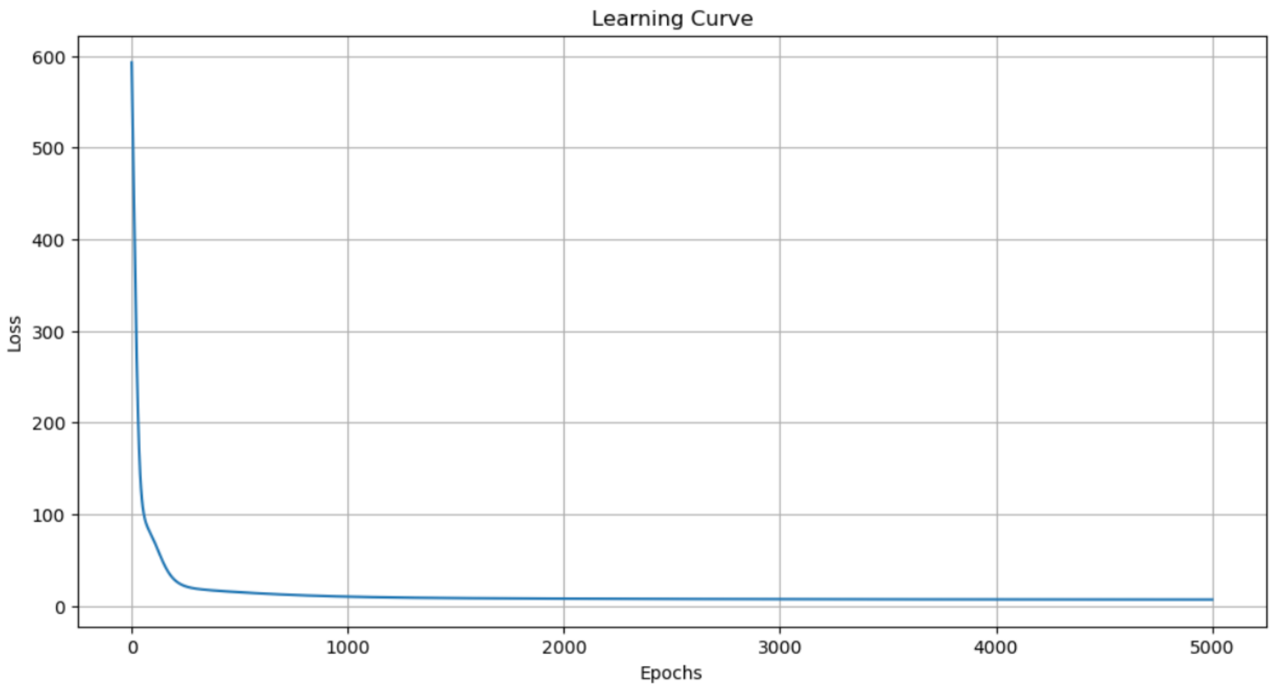
Thus, we managed to further improve our accuracy, which is good news.

You can also see the supplement for completing the classification task in Appendix 2

Appendix 1 – Results for sigmoid activation function

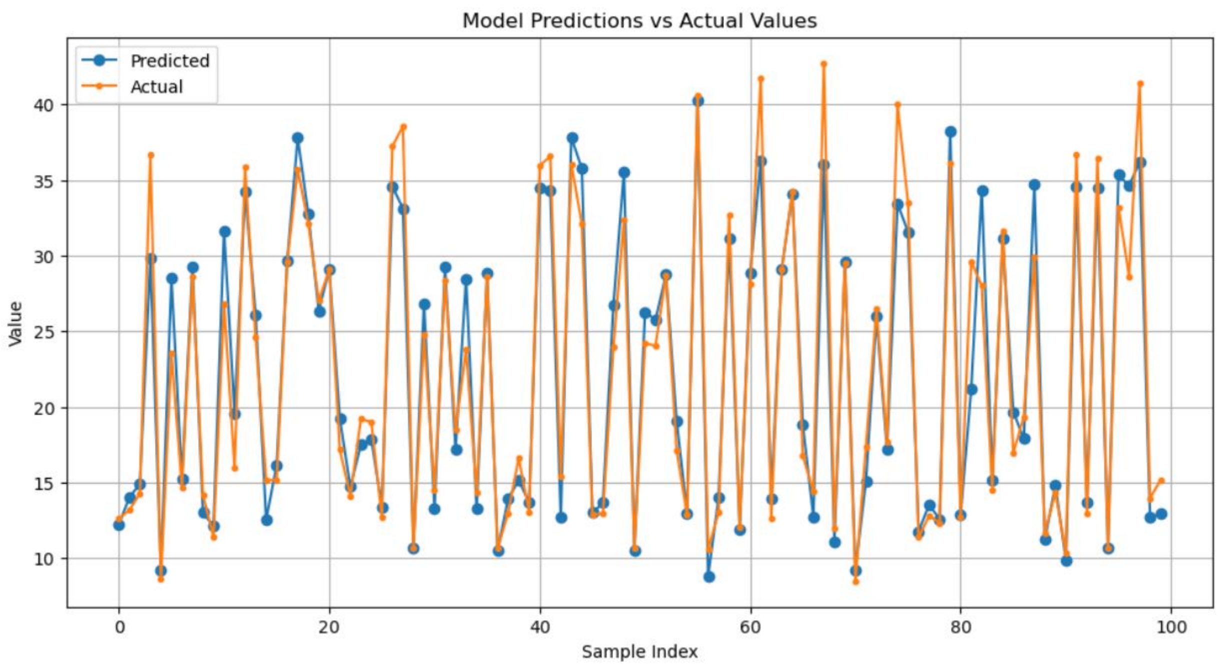
Learning curve:

```
Epoch 0/5001 - Loss: 593.0125647079395
Epoch 500/5001 - Loss: 14.870569966868333
Epoch 1000/5001 - Loss: 10.077888071244884
Epoch 1500/5001 - Loss: 8.480347099738266
Epoch 2000/5001 - Loss: 7.8738897356458075
Epoch 2500/5001 - Loss: 7.545270021281336
Epoch 3000/5001 - Loss: 7.323478922964707
Epoch 3500/5001 - Loss: 7.164838541220904
Epoch 4000/5001 - Loss: 7.050890267900147
Epoch 4500/5001 - Loss: 6.9694152581023845
Epoch 5000/5001 - Loss: 6.911196076767894
```



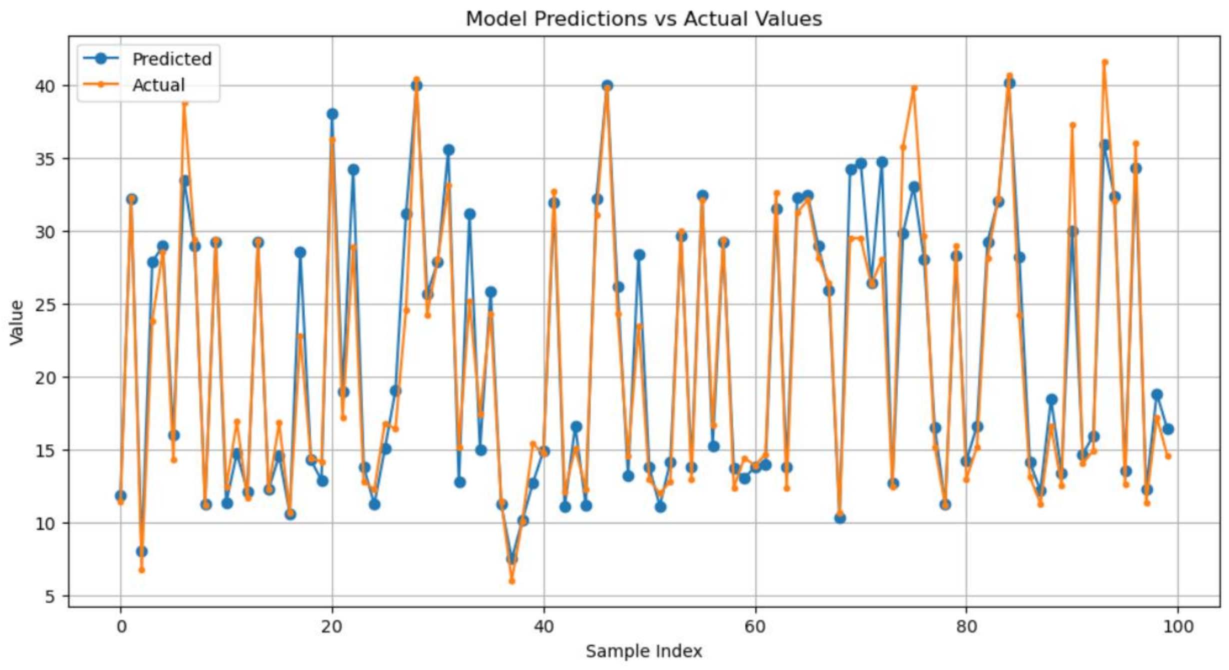
Training RMS Error

```
Training MSE: 6.911196076767894
Training RMSE: 2.628915380298098
Predicted 0: 12.21, Actual: 12.63
Predicted 1: 14.03, Actual: 13.17
Predicted 2: 14.90, Actual: 14.28
Predicted 3: 29.81, Actual: 36.7
Predicted 4: 9.19, Actual: 8.6
Predicted 5: 28.52, Actual: 23.54
Predicted 6: 15.22, Actual: 14.71
Predicted 7: 29.29, Actual: 28.62
Predicted 8: 13.01, Actual: 14.18
Predicted 9: 12.12, Actual: 11.42
```



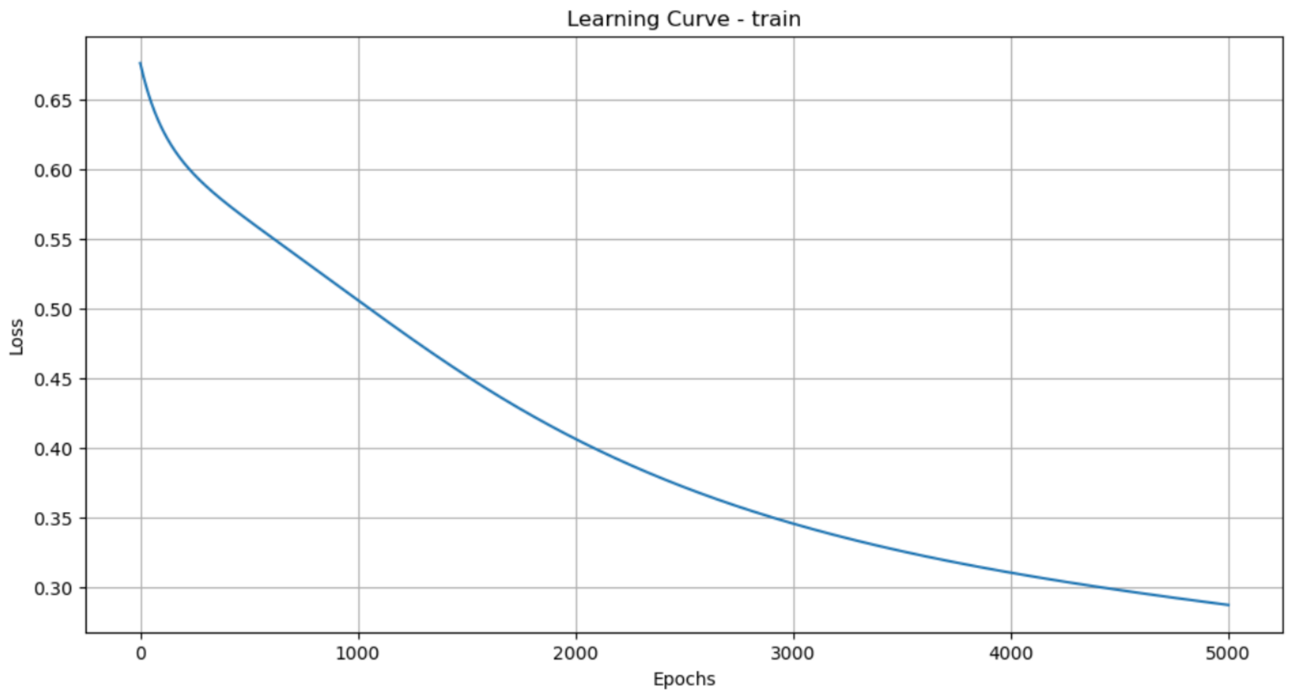
Test RMS Error

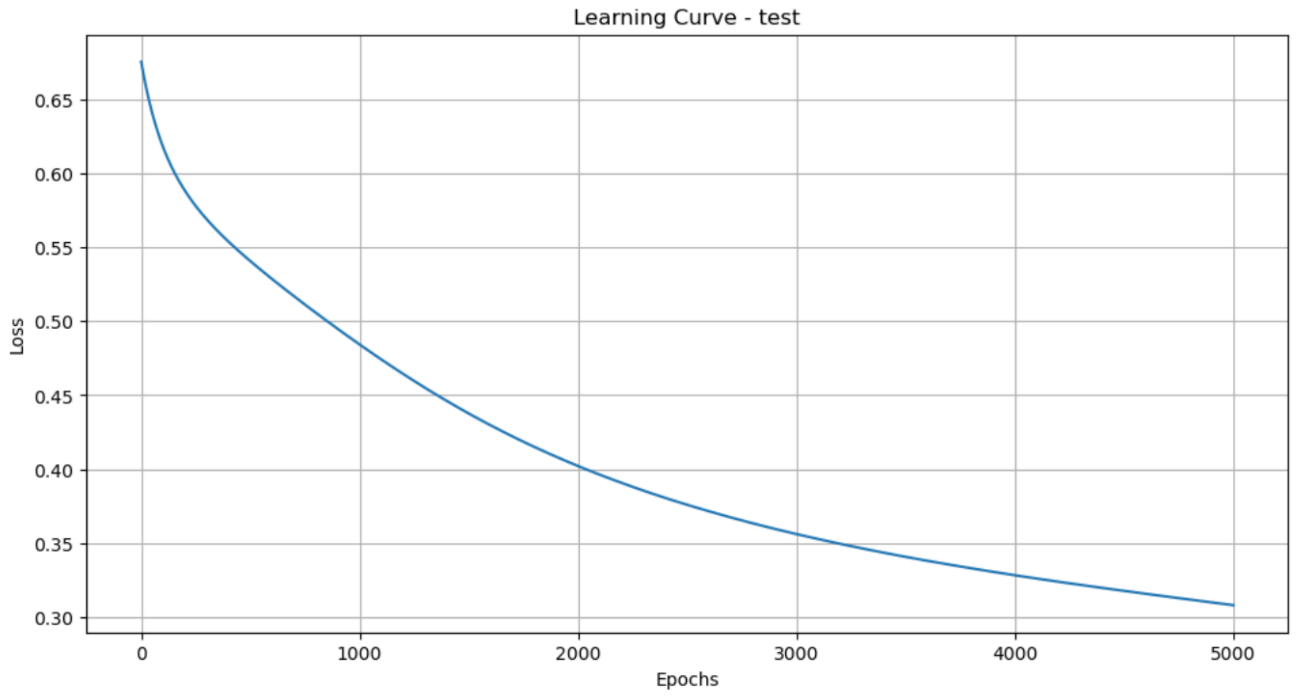
```
Test MSE: 7.187594189641267
Test RMSE: 2.6809688900920254
Predicted 0: 11.93, Actual: 11.45
Predicted 1: 32.17, Actual: 32.31
Predicted 2: 8.04, Actual: 6.85
Predicted 3: 27.88, Actual: 23.86
Predicted 4: 28.97, Actual: 28.6
Predicted 5: 16.00, Actual: 14.34
Predicted 6: 33.52, Actual: 38.84
Predicted 7: 28.98, Actual: 29.39
Predicted 8: 11.33, Actual: 11.2
Predicted 9: 29.25, Actual: 29.4
```

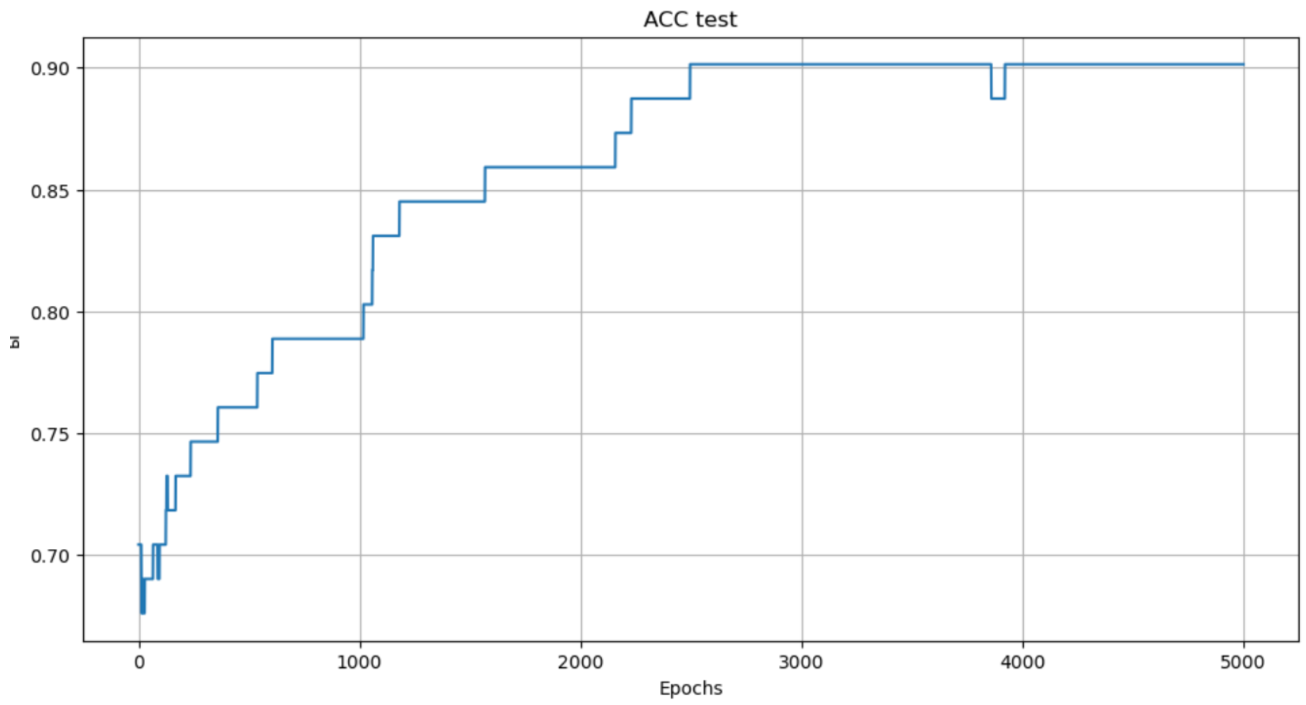


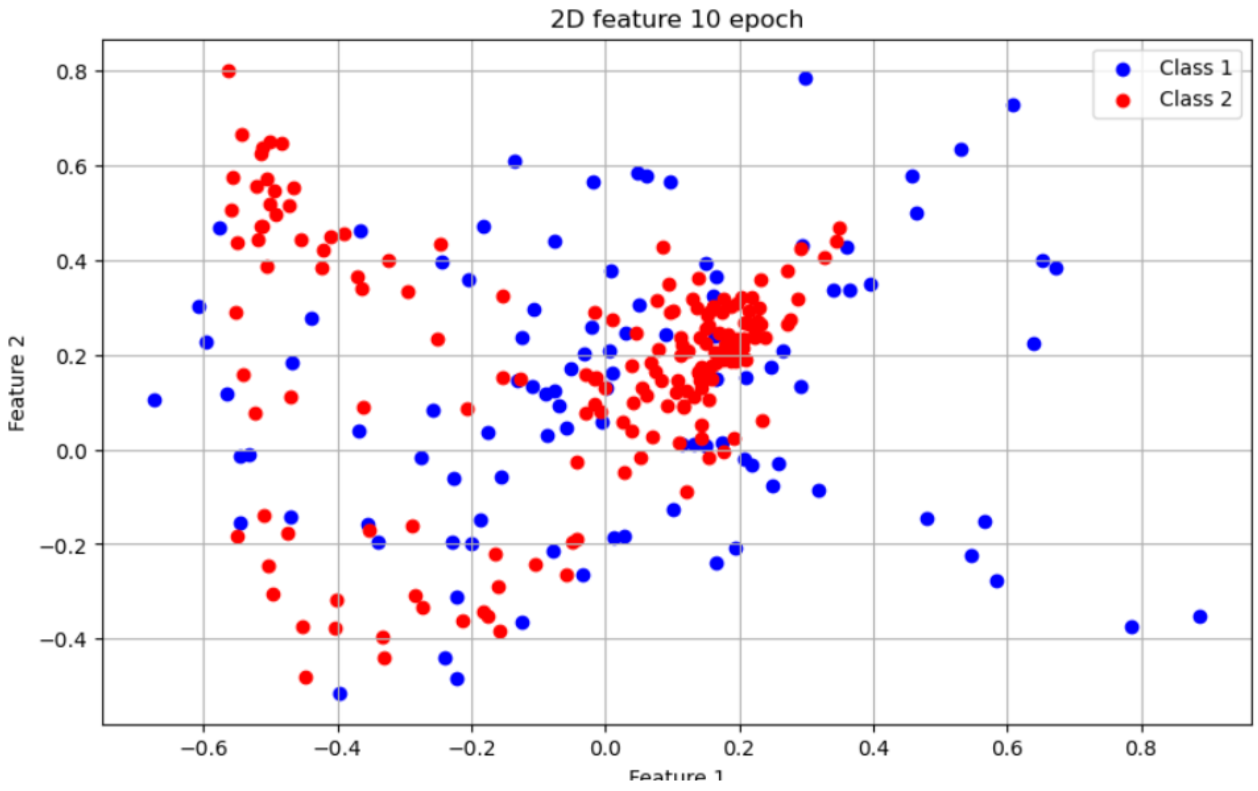
Appendix 2 – Easy way

Only at the end of doing my homework did I discover that if we use other parameters to train the model (***learning_rate* = *0.005*, *epochs* = *5001*** and ***hidden_dim* = *3***) without using Weight Decay, without Batching, then we get excellent results (shown below). Nevertheless, I felt sorry not to include in the report all the work I had done, into which I had invested a considerable amount of time. Therefore, I decided to add this part to Appendix 2 (:

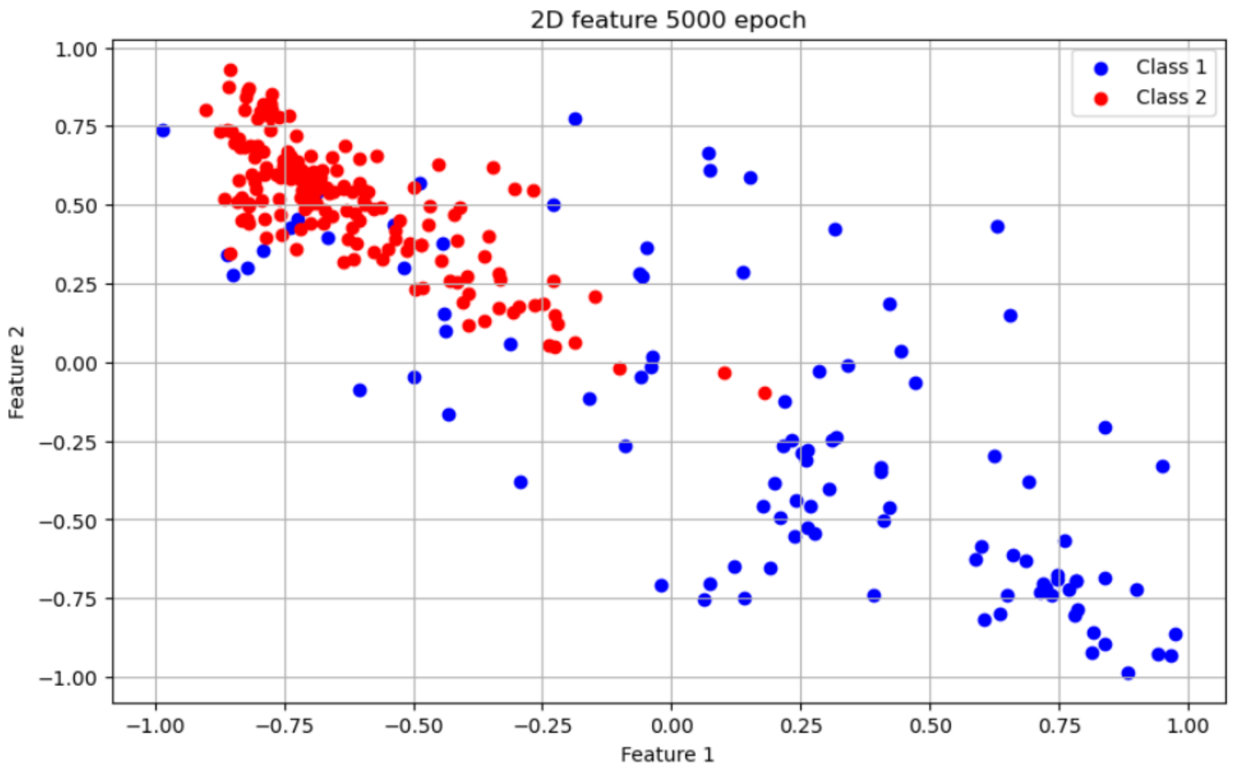




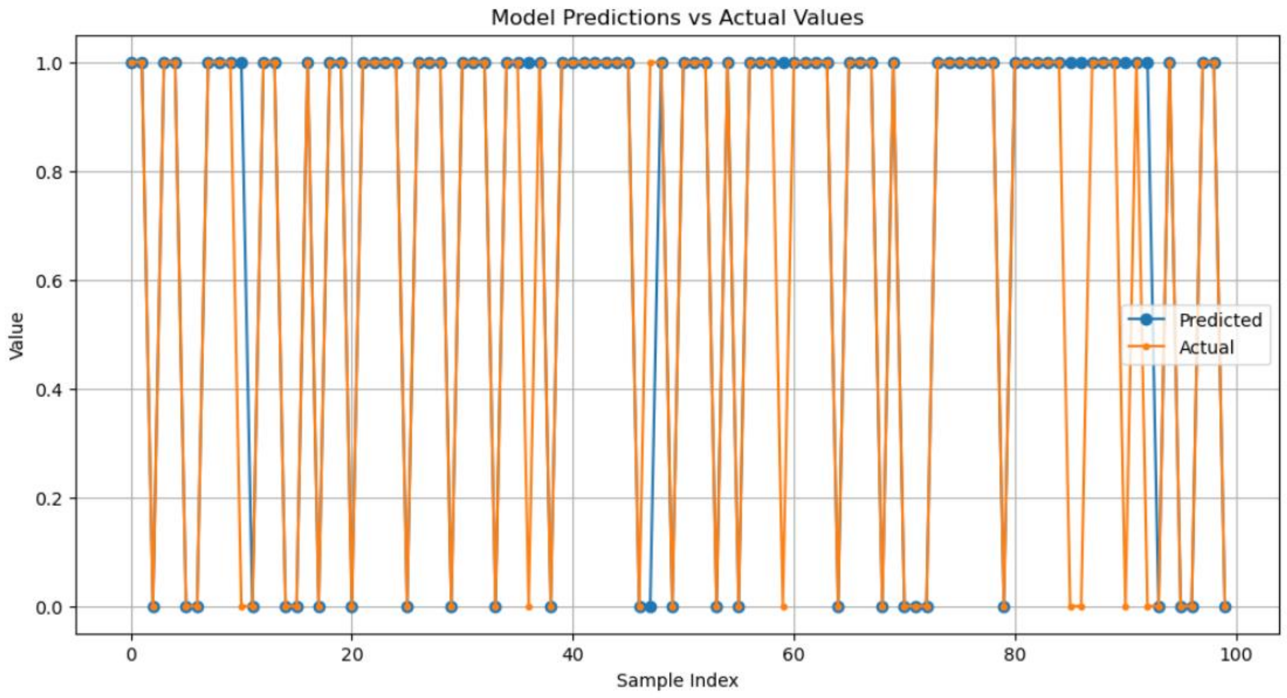

ACC test
Epochs
0.90
0.85
0.80
0.75
0.70
0
1000
2000
3000
4000
5000


2D feature 10 epoch
Feature 2
Feature 1
Class 1
Class 2
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
0.0
0.2
0.4
0.6
0.8

Student: Rasul Mankaev (孟開達)
ID: 312540007

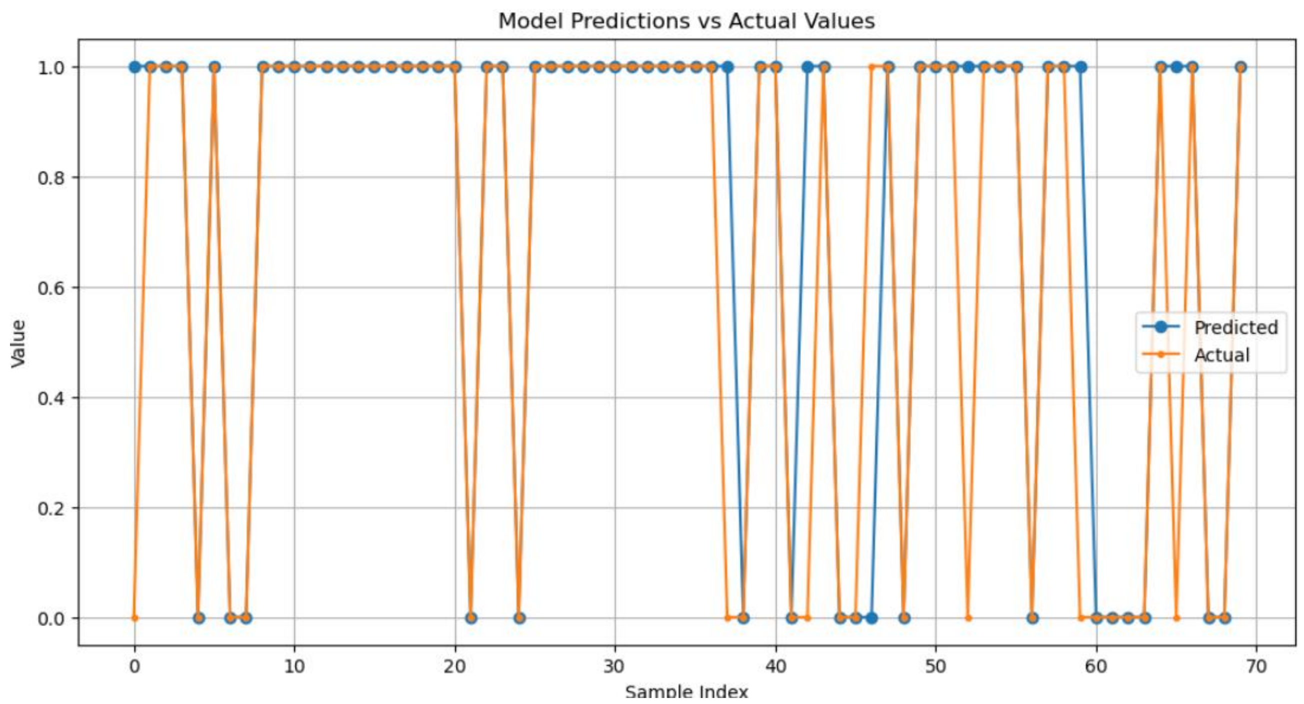


```
Training BCE: 0.28693609447664 77
Training Accuracy: 90.00%
```



```
Test BCE: 0.30812432836739995
Test Accuracy: 90.14%
```

Student: Rasul Mankaev (孟開達)
ID: 312540007


Model Predictions vs Actual Values
Value
Sample Index
Predicted
Actual