Evan Tan 27401995
etan0008@student.monash.edu

1.

Would it be possible to train a perceptron using a variant of the perceptron training algorithm in which the bias weight is left unchanged, and only the other weights are modified?

The answer depends on several factors, including the separability of the data and the value of the bias.

For example, if the data is linearly separable about the origin and thus can be classified using a perceptron algorithm without any bias, keeping the bias constant would offset the decision boundary in which the data is classified as 0 or 1, modifying the accuracy of the perceptron. However, the degree of error is dependent on the location of the boundary, which relies on the magnitude and sign of the bias, that determines the direction and how much the boundary is shifted.

Imagine in 2D space, where the decision boundary is a line. If the model achieves X% accuracy for a perceptron without bias, and the perceptron with bias that is not updated is used, the decision boundary is shifted using the new perceptron. The reduction or improvement in accuracy is also dependent on the margin which separates the data, as a larger margin would mean there is a larger threshold for the perceptron to incorrectly classify data, so adding a bias that shifts the decision boundary would not affect the perceptron too much.

2. CODING UP FUNCTIONS

Before marking, **PLEASE NOTE** that I have changed the shape of theta. Originally, the shape of theta is (2,1):

```
theta = np.random.randn(X_train.shape[1], 1) # shape (2,1)
```

and I have changed it to (1,2):

```
theta = np.random.randn(1, X_train.shape[1])  # shape (1,2)
```

To make results reproducible, I have used a global NumPy random seed of **0**.

2.1 Sigmoid Function

```
def sigmoid(x: np.ndarray):
    """Computes the sigmoid of x"""
    return 1 / (1 + np.exp(-x))
```

2.2 Gradient and loss (cost) Function

```
def compute_grad_loss(X, y, theta):
    """Compute gradient and loss"""
    # alternative
    # epsilon = 7.0 / 3 - 4.0 / 3 - 1
    sig_wx = sigmoid(X @ theta.T)
    loss = -np.mean(
        y * np.log(sig_wx + np.finfo(float).eps)
        + (1 - y) * np.log(1 - sig_wx + np.finfo(float).eps)
    )
    # numpy still screams log(0) errors using this
    # sig_wx[sig_wx < np.finfo(float).eps] = np.finfo(float).eps
    # loss = -np.mean(y * np.log(sig_wx) + (1 - y) * np.log(1 - sig_wx))

    # calculate mean across all datapoints
    grad_vec = (sig_wx - y).T @ X / X.shape[0]
    # alternatively
    # grad_vec = np.mean((sig_wx - y) * X, axis=0, keepdims=True)

    return loss, grad_vec
```

2.4 Predict Function

```
def predict(X, theta):
    """Predict labels based on learned model parameters
    :param X: (M,N) samples, where M is the number of samples
    :type X: np.ndarray
    :param theta: (1,N) model parameters, N = number of features
    :type theta: np.ndarray
    :return: predictions of labels
    :rtype: np.ndarray.astype(int)
    """
    # jank boolean conversion
    return (sigmoid(X @ theta.T) >= 0.5).astype(int)
```

Evan Tan 27401995
etan0008@student.monash.edu

2.3 Gradient Descent Algorithm

```
def do_gradient_descent(LR=5e-2, epochs=1e3):
    """Perform gradient descent"""
    np.random.seed(0)
    # randomly initilize theta, the parameters of the logistic model
    # theta = np.random.randn(X_train.shape[1], 1)  # shape (2,1)
    theta = np.random.randn(1, X_train.shape[1])  # shape (1,2)

    # this is the learning rate of the GD algorithm, you need to tune this
    # and study its effects in your report
    lr = LR

    # this is the maximum number of iterations of the GD algorithm.
    # Since we use the GD, each iteration of the algorithm is equivalent
    # to one epoch, hence the name
    max_epoch = int(epochs)

    # keep track of the loss/accuracy values for plotting
    loss = np.zeros(max_epoch)
    accuracy = np.zeros(max_epoch)
    log_interval = max_epoch // 10
    for epoch in range(max_epoch):
        # call the compute_grad_loss that you have implemented above to
        # measure the loss and the gradient
        loss[epoch], grad_vec = compute_grad_loss(X_train, y_train, theta)
        # update the theta parameter according to the GD here
        theta -= lr * grad_vec

        # storage for plotting
        y_test_hat = predict(X_test, theta)
        accuracy[epoch] = float(sum(y_test_hat == y_test)) / float(len(y_test))

        if (epoch + 1) % log_interval == 0:
            print(
                f"Epoch:{epoch+1}/{max_epoch} \
                Loss: {loss[epoch]:.6f} \
                Acc: {accuracy[epoch]:.6f}"
            )
    return theta, loss, accuracy
```

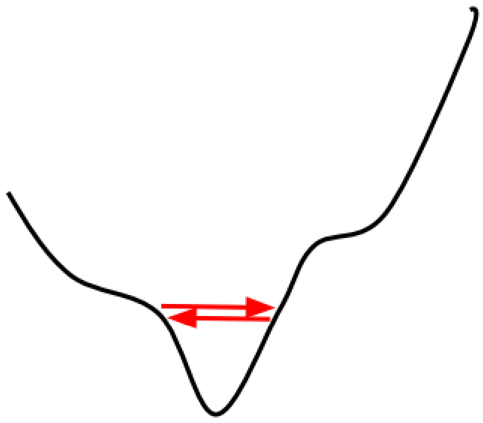
3. toy_data.npz Dataset

3.1 Effect of Learning Rate

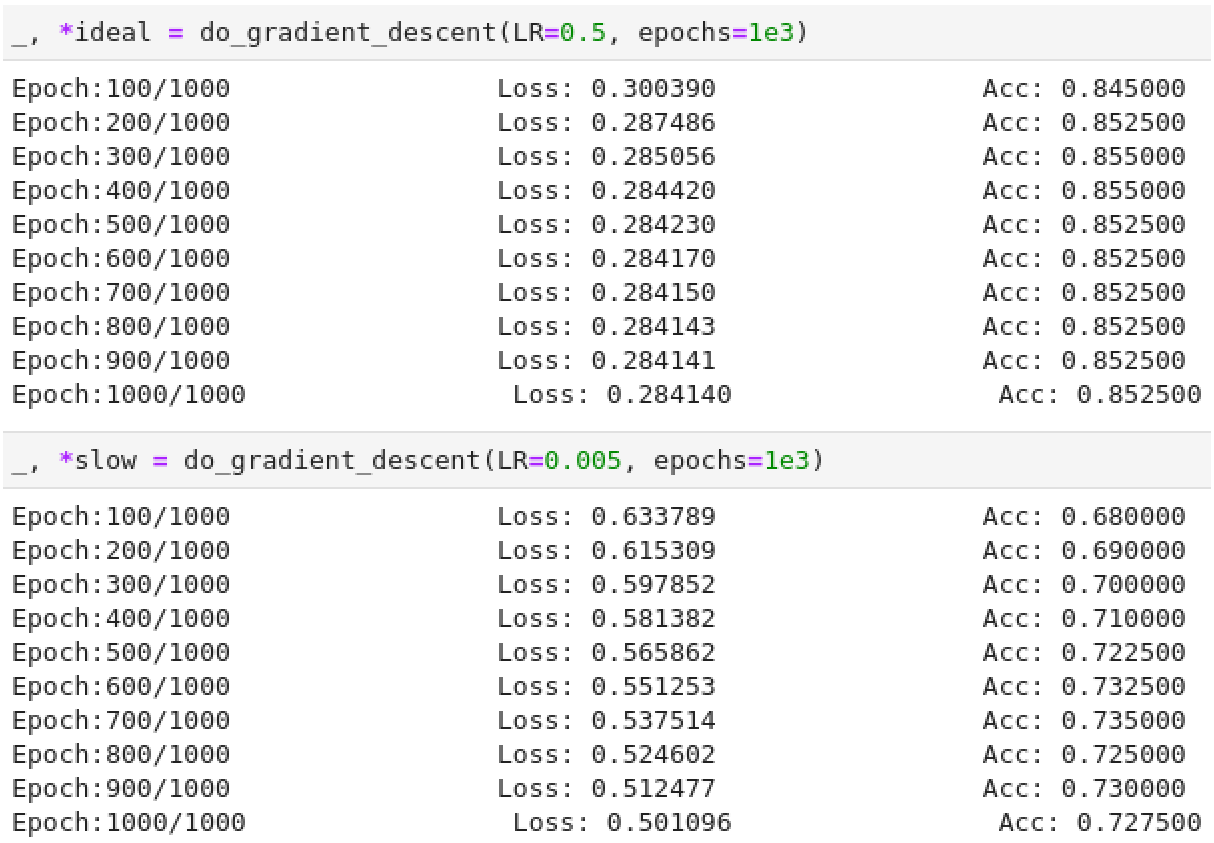
The learning rate, η in our experiment is fixed, and affects our gradient descent algorithm by determining (in conjunction with gradient tensor) how much to update our model parameters at every epoch, and coupled with the gradient vector gives us both direction and how much to update the model parameters by in order to minimize loss.

Evan Tan 27401995
etan0008@student.monash.edu

If the learning rate is ideal, the loss converges to a value quickly and nicely. If the learning rate is low, the more training iterations or epochs are required as the model parameters are updated with smaller steps. If the learning rate is large, it is possible for gradient descent to either diverge or oscillate in local minima as seen in the diagram below from Mehrtash's lecture slides.



The convergence rate will be much slower when the η is low as the model parameters are updated in smaller steps, as compared to η at a higher value, as seen in Fig. 1. where decreasing the learning rate significantly from 0.5 to 0.005, and keeping the number of epochs the same, the model with lower learning rate is unable to converge to the same loss, thus lower learning rates require more training iterations or epochs. However if the learning rate is extremely low, the model might not converge to the optimal solution at all.



```
_, *ideal = do_gradient_descent(LR=0.5, epochs=1e3)
```

```
Epoch:100/1000                Loss: 0.300390                Acc: 0.845000
Epoch:200/1000                Loss: 0.287486                Acc: 0.852500
Epoch:300/1000                Loss: 0.285056                Acc: 0.855000
Epoch:400/1000                Loss: 0.284420                Acc: 0.855000
Epoch:500/1000                Loss: 0.284230                Acc: 0.852500
Epoch:600/1000                Loss: 0.284170                Acc: 0.852500
Epoch:700/1000                Loss: 0.284150                Acc: 0.852500
Epoch:800/1000                Loss: 0.284143                Acc: 0.852500
Epoch:900/1000                Loss: 0.284141                Acc: 0.852500
Epoch:1000/1000                Loss: 0.284140                Acc: 0.852500
```

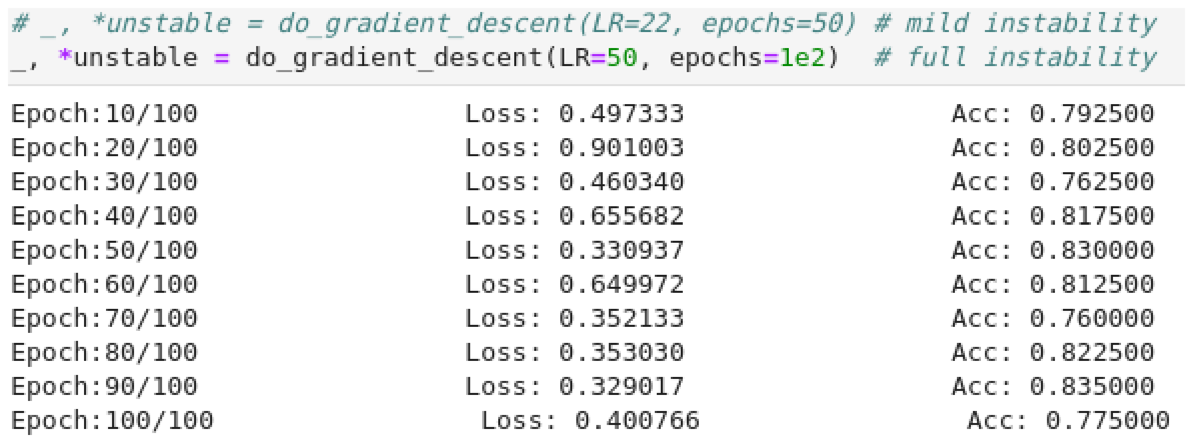
```
_, *slow = do_gradient_descent(LR=0.005, epochs=1e3)
```

```
Epoch:100/1000                Loss: 0.633789                Acc: 0.680000
Epoch:200/1000                Loss: 0.615309                Acc: 0.690000
Epoch:300/1000                Loss: 0.597852                Acc: 0.700000
Epoch:400/1000                Loss: 0.581382                Acc: 0.710000
Epoch:500/1000                Loss: 0.565862                Acc: 0.722500
Epoch:600/1000                Loss: 0.551253                Acc: 0.732500
Epoch:700/1000                Loss: 0.537514                Acc: 0.735000
Epoch:800/1000                Loss: 0.524602                Acc: 0.725000
Epoch:900/1000                Loss: 0.512477                Acc: 0.730000
Epoch:1000/1000                Loss: 0.501096                Acc: 0.727500
```

Figure. 1: Effect of learning rate on convergence, using 0.5 and 0.005 learning rate respectively, for 1000 epochs.

Evan Tan 27401995
etan0008@student.monash.edu

If the learning rate is too high, the steps taken in updating the model parameters during training will be too high, the model could diverge or oscillate. This is seen in the loss curve as it begins to oscillate between increasing and decreasing and does not converge to a value as seen in Fig. 2, where the loss values continuously increase and decrease, and does not converge to a single value as seen in Fig. 1's η = 0.5.



```
# _, *unstable = do_gradient_descent(LR=22, epochs=50) # mild instability
_, *unstable = do_gradient_descent(LR=50, epochs=1e2)  # full instability
```

```
Epoch:10/100                 Loss: 0.497333                Acc: 0.792500
Epoch:20/100                 Loss: 0.901003                Acc: 0.802500
Epoch:30/100                 Loss: 0.460340                Acc: 0.762500
Epoch:40/100                 Loss: 0.655682                Acc: 0.817500
Epoch:50/100                 Loss: 0.330937                Acc: 0.830000
Epoch:60/100                 Loss: 0.649972                Acc: 0.812500
Epoch:70/100                 Loss: 0.352133                Acc: 0.760000
Epoch:80/100                 Loss: 0.353030                Acc: 0.822500
Epoch:90/100                 Loss: 0.329017                Acc: 0.835000
Epoch:100/100                 Loss: 0.400766                Acc: 0.775000
```

Figure 2: Model instability using a learning rate of 50 and 1000 epochs.

3.1.1 Model Instability

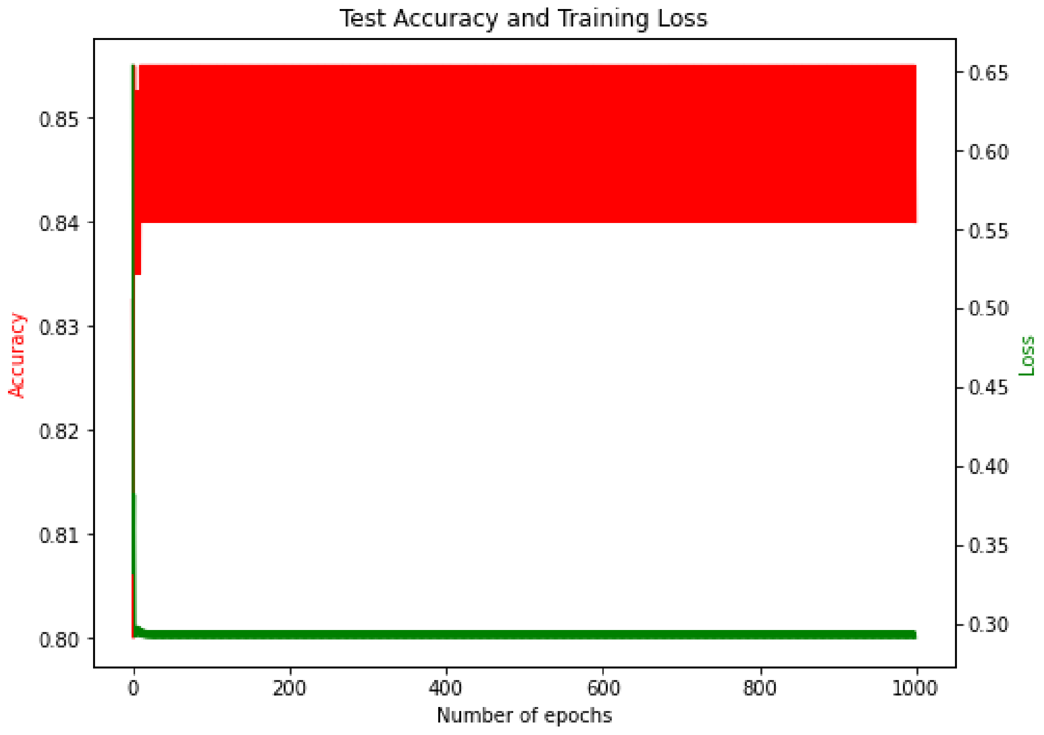


Figure 3: Model instability using learning rate of 22 and 1000 epochs.

Using a learning rate of 22, the model becomes unstable as there is some strange behaviour occurring as seen by the lack of a distinct smooth curve in Fig. 3. Upon closer inspection in Fig. 4, using only 100 epochs and the same learning rate of 22, the oscillatory behaviour is observed as expected.

Evan Tan 27401995
etan0008@student.monash.edu

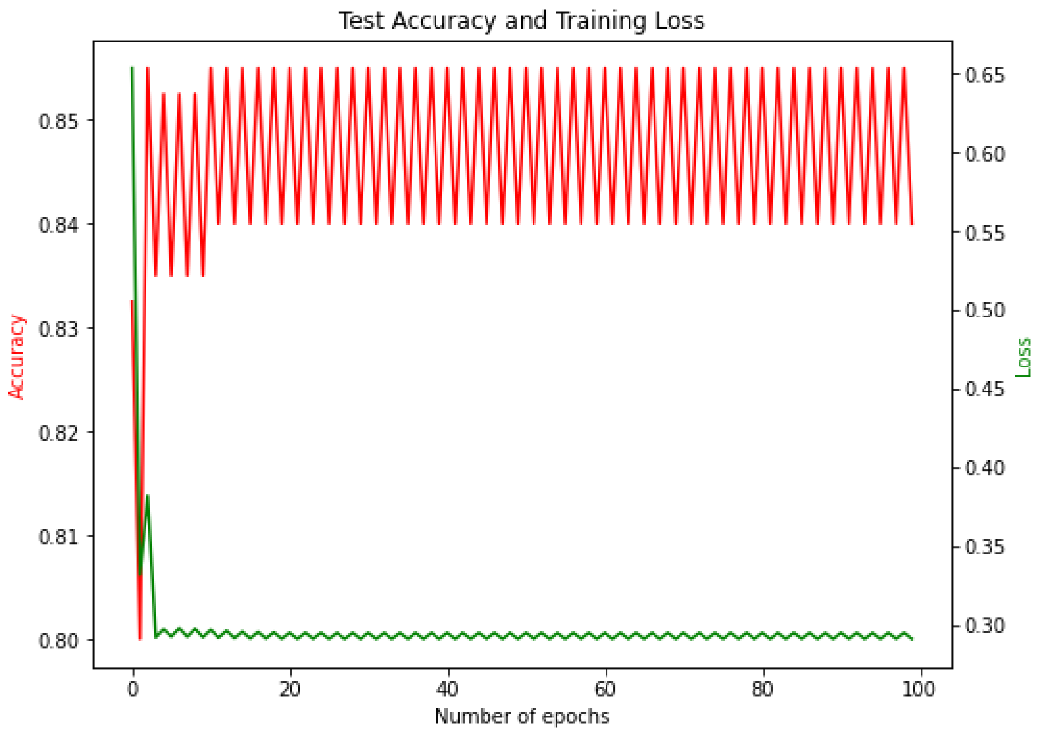


Figure 4: Model instability and oscillatory behaviour using learning rate of 22 and 100 epochs.

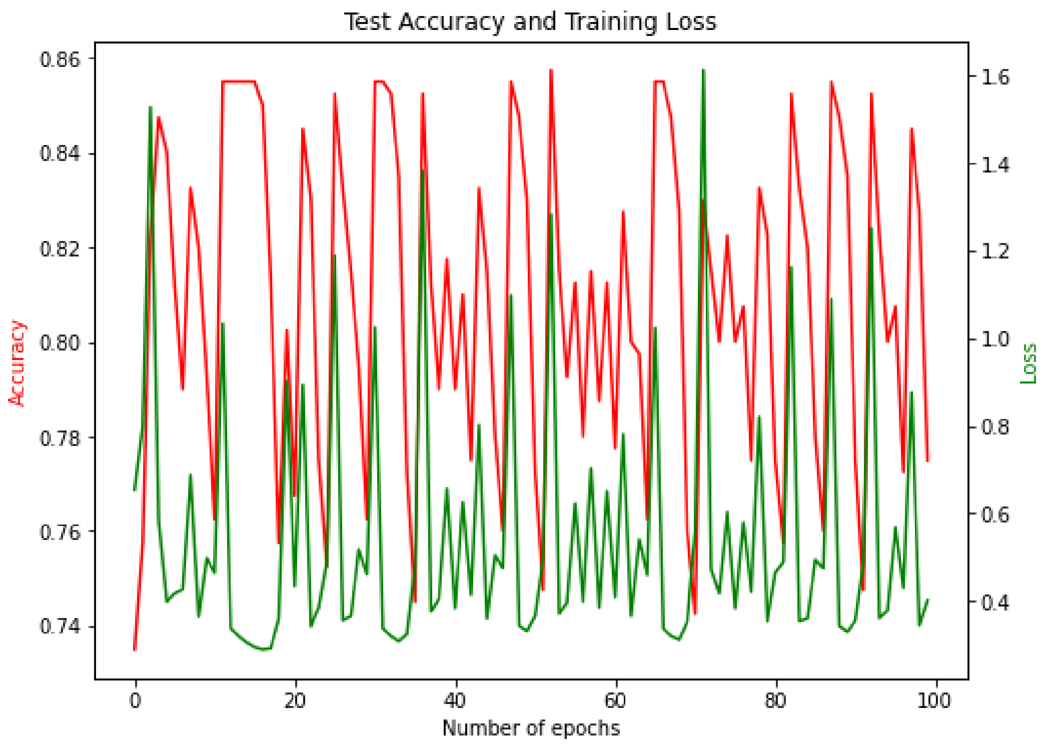


Figure 5: Model instability for learning rate of 50 and 100 epochs, corresponding to model trained in Figure 2.

From Fig. 5, if the learning rate is increased further from 22, which demonstrates mild instability in Fig. 4, the effect is more obvious, where the model is fully unstable with larger oscillations and noisier loss and accuracy curves and is unable to converge at all.

Evan Tan 27401995
etan0008@student.monash.edu

3.2.2 Slow Model Convergence

By decreasing the learning rate to 0.005, the model converges extremely slowly as seen in Fig. 6.

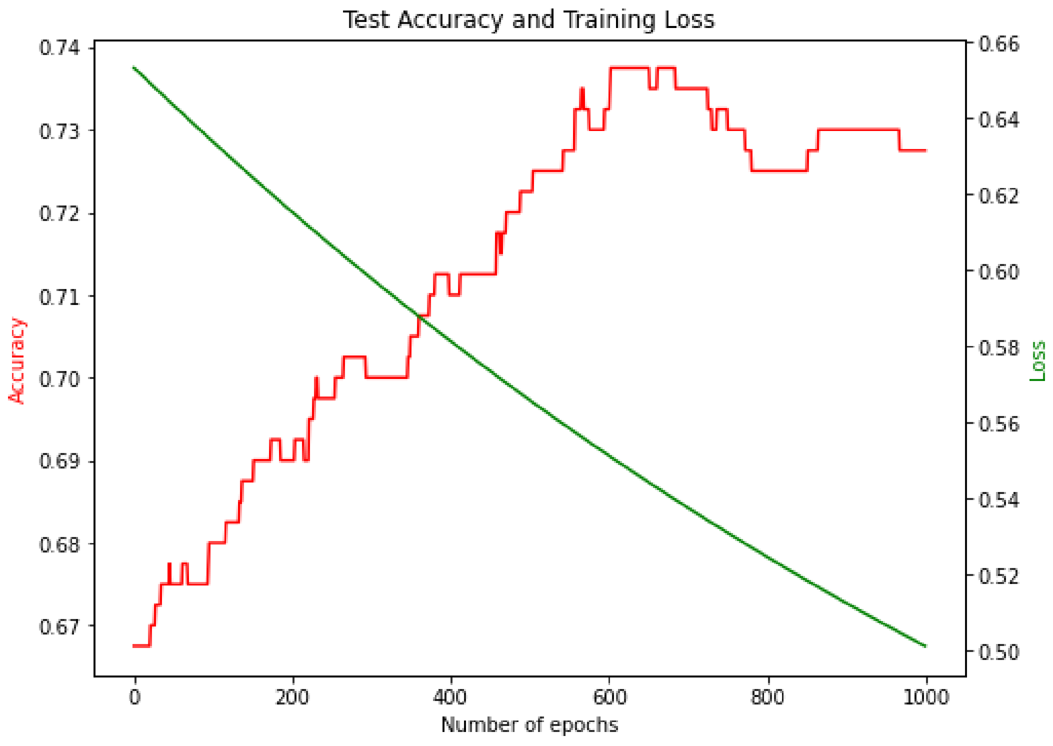


Figure 6: Slow model convergence at learning rate of 0.005 and 1000 epochs.

3.1.3 Ideal Model Convergence

Using a learning rate of 0.5 yields a nice result, requiring approximately 300 epochs to converge and converges much more ideally as seen in Fig 7 and faster as compared to 0.005 in Fig. 6.

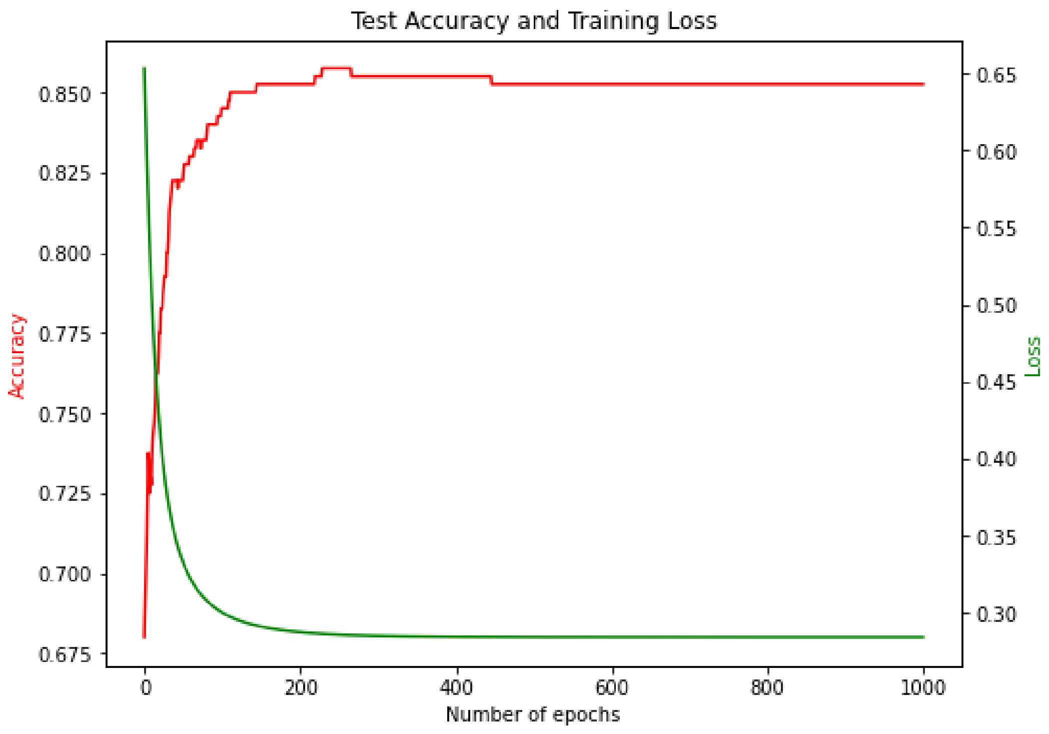


Figure 7: Model converging nicely using learning rate of 0.5, and 1000 epochs.

Acceptable Range

From the previously discussed values, the following model convergence condition for this particular problem is as follows.

$$\text{convergence} = \begin{cases} \eta \leq 0.05 & \text{slow} \\ 0.05 < \eta \leqslant 20 & \text{ideal} \\ \eta > 20 & \text{unstable} \end{cases}$$

Figure 8: Model instability, slow and ideal convergence conditions based on learning rate.

The acceptable range for learning rate is between 0.05 and 20 to ensure that the model converges somewhere within 300 epochs or less, as the accuracy at η = 0.05 is 84.25%, with a loss of 0.300394, and the accuracy at η = 20 is 85.25% with a loss of 0.284140. The difference in accuracy is 1% lower in absolute terms, and the difference in loss is -0.01625, which is -5.72% relatively lower and somewhat acceptable.

The use of such high learning rates is unusual as learning rates typically range from 0-1 for gradient descent in neural networks, but for our specific data and problem the ideal range seems to work well. Furthermore, manual tuning was used with a fixed random seed, perhaps the ideal range would be different when removing the random seed and using automated hyperparameter tuning libraries such as Ray Tune.

The learning rate of 1 to 22 is considered acceptable as the loss curves were stable, even though the accuracy curves had similar shape to that in Fig. 9, where the curve peaks at the start before converging to a stable value.

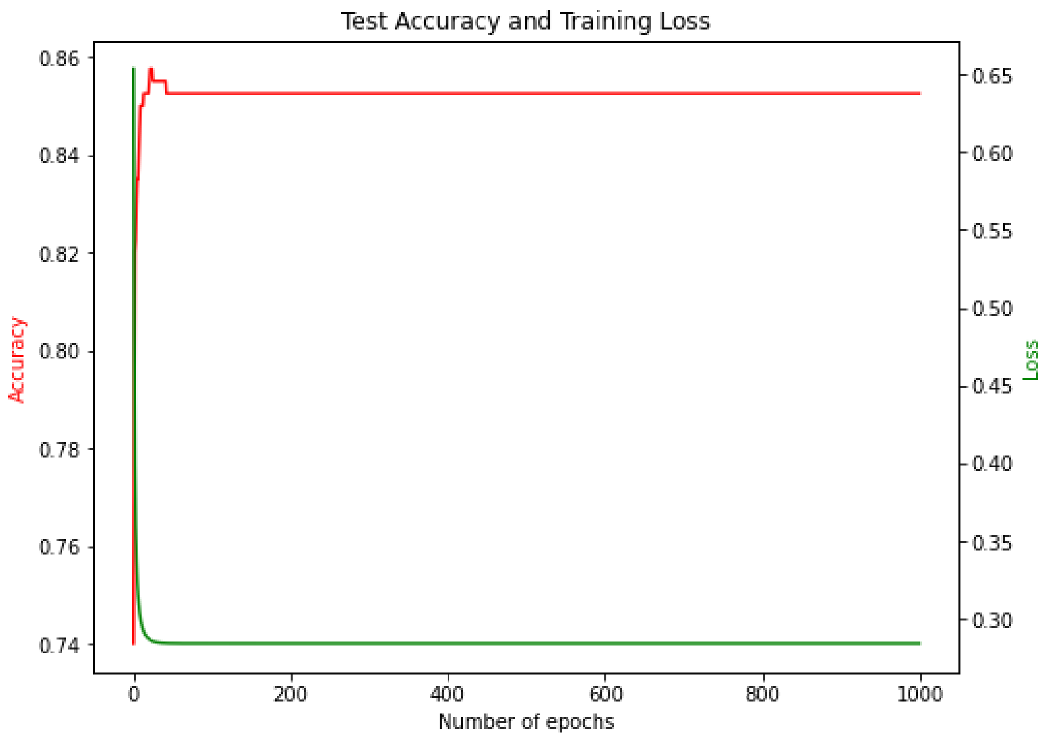


Figure 9: Model using learning rate 5 for 1000 epochs.

3.2 Model Accuracy

As seen in Fig. 1, using a learning rate of 0.5 and 1000 epochs, the accuracy is 0.8525, or **85.25%** accurate with a loss of 0.284140 on the test dataset, which is expected from the decision boundary plot in Fig. 10, where the data is not linearly separable but a simple linear model is used in an attempt to classify the data.

3.3 Model Accuracy with Bias

Using a learning rate of 0.5 and 1000 epochs, without bias, our loss converges to 0.284140 and accuracy is 85.25%. However, **with bias**, our loss converges to 0.275031 and accuracy is **84.75%** on the test dataset.

There is an improvement of 2.26% in having relatively lower loss values, however the model accuracy is 0.5% lower, which is counterintuitive. This could be explained with more '0's being classified incorrectly, since the data is asymmetric as there are 212 '0's and 188 '1's boundary, resulting in lower accuracy. This could mean that 2D features were barely enough to classify our data as seen in the decision boundary plot.

3.4 Decision Boundary Plot

Using blue to represent label '0' and red to represent label '1', the following decision boundary plot is created.

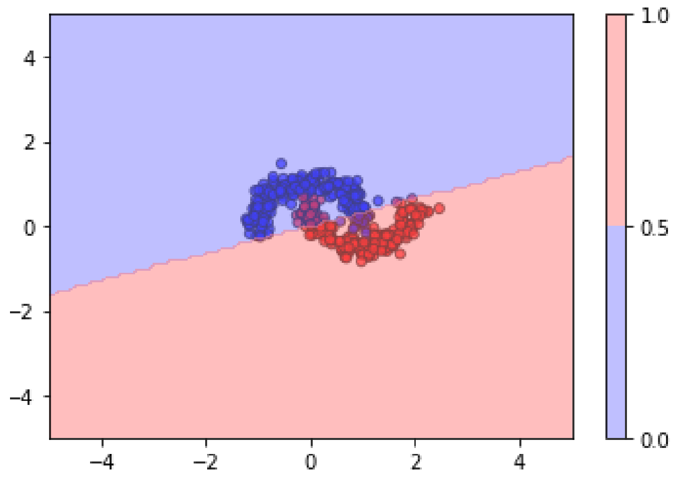


Figure 10: Decision boundary for model without bias, learning rate 0.5, 1000 epochs.

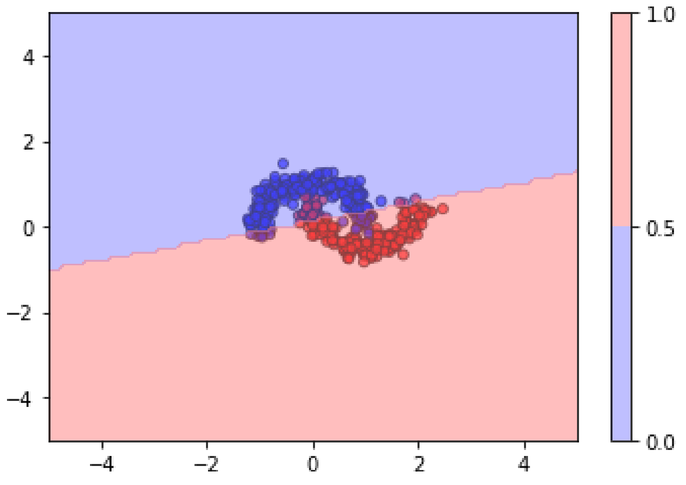


Figure 11: Decision boundary for model with bias, learning rate 0.5, 1000 epochs.

Comparing Fig. 10 and 11, the bias has shifted the decision boundary to incorrectly classify more ‘0’s, and correctly classify more ‘1’s, as seen by the points crossing the decision boundary, resulting in a lower accuracy as seen in the reduction in accuracy from 85.25% to 84.75%. In particular, the model with bias has a decision boundary that is rotated about the origin and translated slightly.

4. toy_data_two_circles.npz

4.1 Model Accuracy

Using a learning rate of 0.5, and 1000 epochs, the same logistic model achieves **47.5% accuracy** with a loss of 0.691811 on the new test dataset.

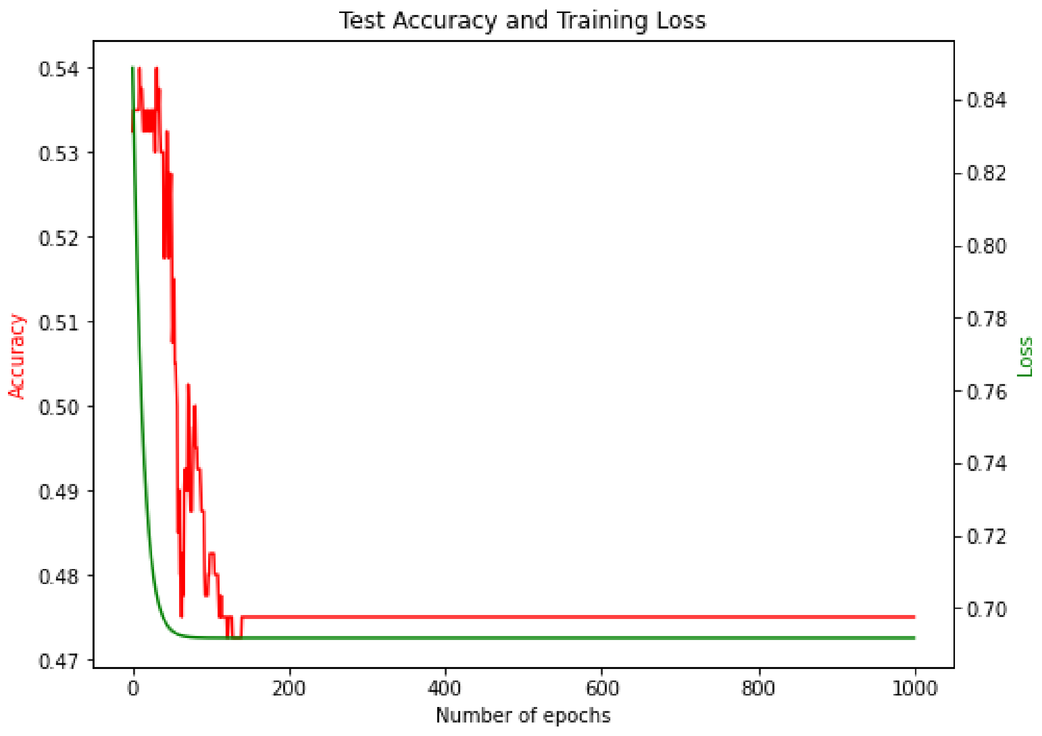


Figure 12: Model loss and accuracy for toy_data_two_circles.npz dataset, using learning rate of 0.5 and 1000 epochs

4.2 Nonlinear Features

After mapping the original 2D features to 5D nonlinear features, without bias, the loss converges to 0.553337 and test accuracy decreases by 0.5% to **47%** on the test dataset for 1000 epochs on the test dataset.

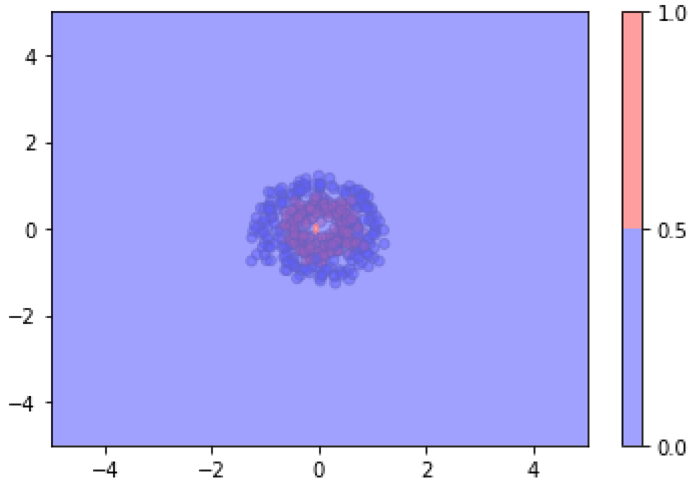


Figure 13: Plot of decision boundary, where decision boundary is represented by the colorbar on the right hand side, and test data points are translucent. Blue represents label '0' and red represents label '1'.

With further data exploration, the decision boundary in Fig. 13 is expected as the model has incorrectly classified all data points with label '1', as there are 188 data points labelled '0' and 212 data points labelled '1', the model is only able to classify the '0's correctly to achieve 188/400 = 47% accuracy.

However, if a bias is added on top of nonlinear features to create 6D model parameters, the model loss converges to 0.163754 and achieves an accuracy of 93.5% on the test dataset, a significant increase where the model parameters have correctly classified almost all of the points. By specifying the bias on top of the 5D model parameters, 6D model parameters has created an ellipsoid with radius of approximately 1 unit, intuitively in equation (2). Thus the decision boundary is shifted outwards from the origin to achieve high accuracy.

$$(1)\ \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2 + \theta_5 x_1 x_2 = 0$$
$$(2)\ \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2 + \theta_5 x_1 x_2 + \theta_6 = 0$$

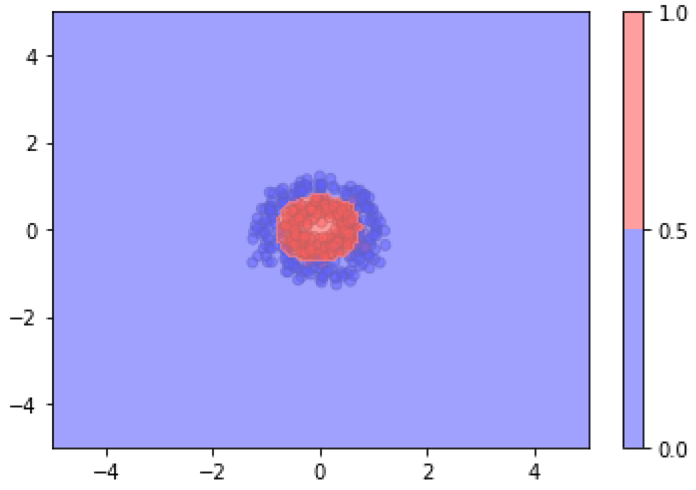


Figure 14: Plot of decision boundary for model with 6D features.

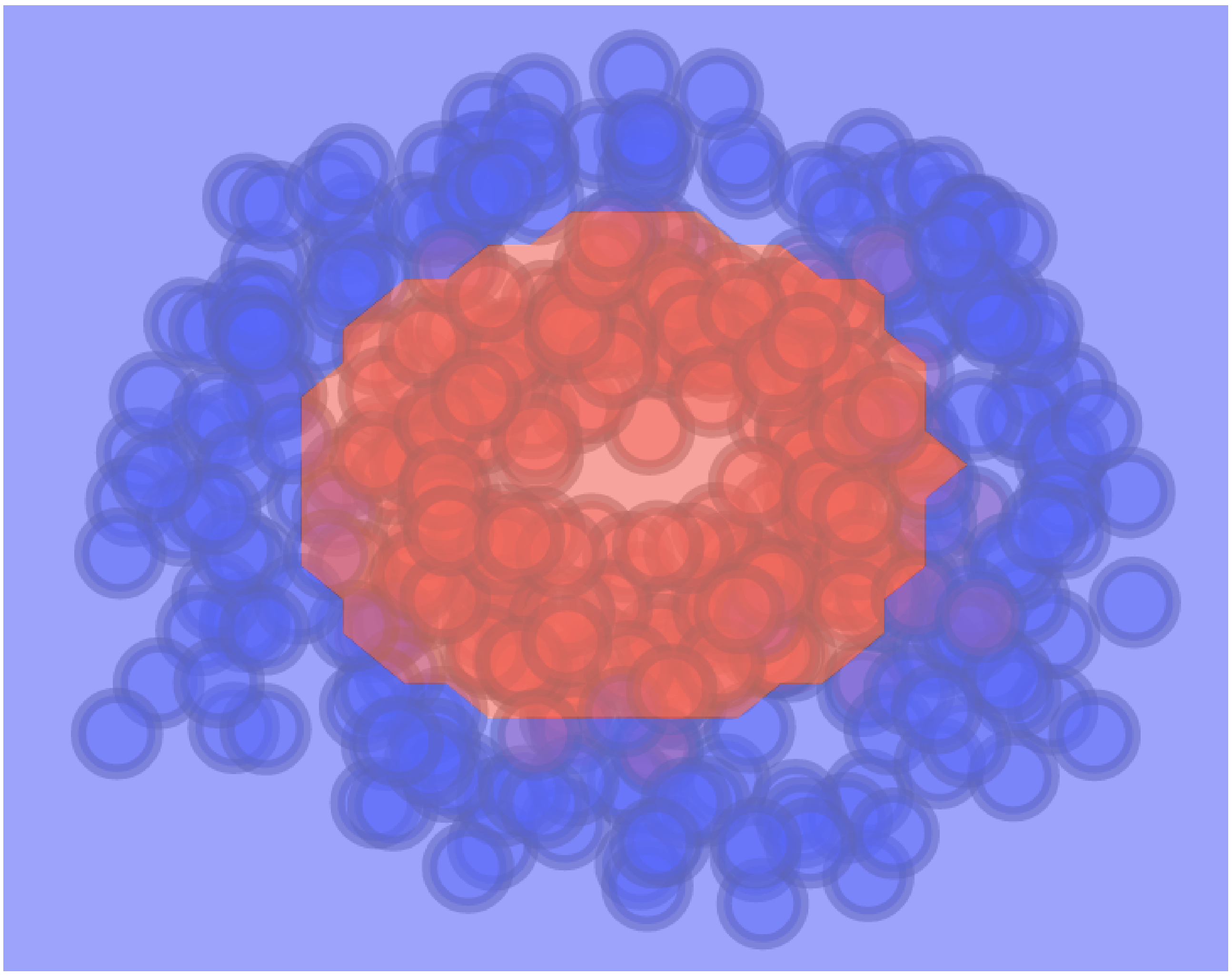


Figure 15: Figure 14 zoomed in, with increased opacity of datapoints.