MultiLayer

February 17, 2025

1 Acknowledgements

This notebook lab has been taken from the TBMI26 course (Neural networks and learning systems).

Examiner: Magnus Borga

Initial code authors (MATLAB version): UNKNOWN

Conversion to Python and overall refinement: Martin Hultman & David Abramian

1.0.1 Quick introduction to jupyter notebooks

- Each cell in this notebook contains either code or text.
- You can run a cell by pressing Ctrl-Enter, or run and advance to the next cell with Shift-Enter.
- Code cells will print their output, including images, below the cell. Running it again deletes the previous output, so be careful if you want to save some results.
- You don't have to rerun all cells to test changes, just rerun the cell you have made changes to. Some exceptions might apply, for example if you overwrite variables from previous cells, but in general this will work.
- If all else fails, use the "Kernel" menu and select "Restart Kernel and Clear All Output". You can also use this menu to run all cells.
- A useful debug tool is the console. You can right-click anywhere in the notebook and select "New console for notebook". This opens a python console which shares the environment with the notebook, which let's you easily print variables or test commands.

1.0.2 Setup

[1]:
```
# Automatically reload modules when changed
%reload_ext autoreload
%autoreload 2
# Plot figures "inline" with other output
%matplotlib inline

# Import modules, classes, functions
from matplotlib import pyplot as plt
import numpy as np

from utils import loadDataset, splitData, plotProgress,␣
 ↪plotProgressNetworkMulti, \
```

```
    plotResultsDots, plotIsolines, plotConfusionMatrixOCR,␣
 ↪plotResultsDotsGradient
from evalFunctions import calcAccuracy, calcConfusionMatrix

# Configure nice figures
plt.rcParams['figure.facecolor']='white'
plt.rcParams['figure.figsize']=(8, 5)
```

1.0.3 *! IMPORTANT NOTE !*

Your implementation should only use the `numpy` (`np`) module. The `numpy` module provides all the functionality you need for this assignment and makes it easier debuging your code. No other modules, e.g. `scikit-learn` or `scipy` among others, are allowed. You can find everything you need about `numpy` in the official documentation.

1.0.4 1. Multi-layer neural network

The implementation of the multi-layer network is a bit beyond the scope of this course, so we will provide it for you. Your task is instead to optimize the training and interpret tthe results.

Similar to the single-layer network, the multi-layer keep track of weight matrices `W1` and `W2`, and bias vectors `B1` and `B2`. The structure of the code is still the same, with a `forward`, `backward`, and `update` function.

1.1 Implementing the forward pass This implementation will look similar to the single-layer code, but note that it also returns the intemediate variable `U`, which is the output of the hidden layer after passing throught the activation function. This will be used in the backward pass.

[2]:
```
def forward(X, W1, B1, W2, B2, useTanhOutput=False):
    """Forward pass of two layer network

    Args:
        X (array): Input samples.
        W1 (array): First layer neural network weights.
        B1 (array): First layer neural network biases.
        W2 (array): Second layer neural network weights.
        B2 (array): Second layer neural network biases.

    Returns:
        Y (array): Output for each sample and class.
        L (array): Resulting label of each sample.
        U (array): Output of hidden layer.
    """

    U = np.tanh(X @ W1 + B1)
    if useTanhOutput:
        Y = np.tanh(U @ W2 + B2)
    else:
```

```
        Y = U @ W2 + B2

    L = Y.argmax(axis=1)

    return Y, L, U
```

1.2 Implementing the backward pass This backward pass implementation uses the so called delta notation. This is very useful when implementing the general case of an N-layer network, since it can easily be implemnted in an iterative manner in a loop going over each layer. We will not do that in this assignment, but it is good to know that this is essentially how the popular frameworks for deep learning functions.

[3]:
```
def backward(W1, B1, W2, B2, X, U, Y, D, useTanhOutput=False):
    """Compute the gradients for network weights and biases

    Args:
        W1 (array): Current values of the layer 1 network weights.
        B1 (array): Current values of the layer 1 network biases.
        W2 (array): Current values of the layer 2 network weights.
        B2 (array): Current values of the layer 2 network biases.
        X (array): Training samples.
        U (array): Intermediate outputs of the hidden layer.
        Y (array): Predicted outputs.
        D (array): Target outputs.

        useTanhOutput (bool): (optional)
            True  - Network uses tanh activation on output layer
            False - Network uses linear (no) activation on output layer

    Returns:
        GradW1 (array): Gradients with respect to W1
        GradB1 (array): Gradients with respect to B1
        GradW2 (array): Gradients with respect to W2
        GradB2 (array): Gradients with respect to B2
    """

    N  = Y.shape[0]
    NC = Y.shape[1]
    if useTanhOutput:
        output_error = 2 * (Y - D) * (1 - Y**2)
    else:
        output_error = 2 * (Y - D)

    GradW2 = (U.T @ output_error) / N
    GradB2 = np.sum(output_error, axis=0) / N

    hidden_error = (output_error @ W2.T) * (1 - U**2)
```

```
    GradW1 = (X.T @ hidden_error) / N
    GradB1 = np.sum(hidden_error, axis=0) / N

    return GradW1, GradB1, GradW2, GradB2
```

1.3 Implementing the weight update For this update function we have implemented a more powerful algorithm called momentum gradient descent. This is part of an optional task at the end of the notebook, and you do not have to use it unless you want to.

[4]:
```
def update(W1, B1, W2, B2, GradW1, GradB1, GradW2, GradB2, params):
    """Update weights and biases using computed gradients.

    Args:
        W1 (array): Current values of the layer 1 network weights.
        B1 (array): Current values of the layer 1 network biases.
        W2 (array): Current values of the layer 2 network weights.
        B2 (array): Current values of the layer 2 network biases.
        GradW1 (array): Gradients with respect to W1.
        GradB1 (array): Gradients with respect to B1.
        GradW2 (array): Gradients with respect to W2.
        GradB2 (array): Gradients with respect to B2.
        params (dict): Dictionary containing learning rate and momentum.

    Returns:
        W1 (array): Updated layer 1 weights.
        B1 (array): Updated layer 1 biases.
        W2 (array): Updated layer 2 weights.
        B2 (array): Updated layer 2 biases.
    """

    LR = params["learningRate"]

    if "momentum" in params:
        M = params["momentum"]
        PrevGradW1 = params["PrevGradW1"]
        PrevGradB1 = params["PrevGradB1"]
        PrevGradW2 = params["PrevGradW2"]
        PrevGradB2 = params["PrevGradB2"]

        W1 = W1 + M * PrevGradW1 - LR * GradW1
        B1 = B1 + M * PrevGradB1 - LR * GradB1
        W2 = W2 + M * PrevGradW2 - LR * GradW2
        B2 = B2 + M * PrevGradB2 - LR * GradB2

        params["PrevGradW1"] = M * PrevGradW1 - LR * GradW1
        params["PrevGradB1"] = M * PrevGradB1 - LR * GradB1
```

```
        params["PrevGradW2"] = M * PrevGradW2 - LR * GradW2
        params["PrevGradB2"] = M * PrevGradB2 - LR * GradB2
    else:
        W1 = W1 - LR * GradW1
        B1 = B1 - LR * GradB1
        W2 = W2 - LR * GradW2
        B2 = B2 - LR * GradB2

    return W1, B1, W2, B2
```

1.4 The training function

[5]:
```
def trainMultiLayer(XTrain, DTrain, XTest, DTest, W1_0, B1_0, W2_0, B2_0,␣
↪params):
    """Trains a two-layer network

    Args:
        XTrain (array): Training samples.
        DTrain (array): Training network target values.
        XTest (array): Test samples.
        DTest (array): Test network target values.
        W1_0 (array): Initial values of the first layer network weights.
        B1_0 (array): Initial values of the first layer network biases.
        W2_0 (array): Initial values of the second layer network weights.
        B2_0 (array): Initial values of the second layer network biases.
        params (dict): Dictionary containing:
            epochs (int): Number of training steps.
            learningRate (float): Size of a training step.

    Returns:
        W1 (array): First layer weights after training.
        B1 (array): Fisrt layer biases after training.
        W2 (array): Second layer weights after training.
        B2 (array): Second layer biases after training.
        metrics (dict): Losses and accuracies for training and test data.
    """

    # Initialize variables
    metrics = {keys:np.zeros(params["epochs"]+1) for keys in ["lossTrain",␣
↪"lossTest", "accTrain", "accTest"]}

    if "useTanhOutput" not in params:
        params["useTanhOutput"] = False

    if "momentum" not in params:
        params["momentum"] = 0
```

```
    nTrain = XTrain.shape[0]
    nTest  = XTest.shape[0]
    nClasses = DTrain.shape[1]

    # Set initial weights
    W1 = W1_0
    B1 = B1_0
    W2 = W2_0
    B2 = B2_0

    # For optional task on momentum
    params["PrevGradW1"] = np.zeros_like(W1)
    params["PrevGradB1"] = np.zeros_like(B1)
    params["PrevGradW2"] = np.zeros_like(W2)
    params["PrevGradB2"] = np.zeros_like(B2)

    # Get class labels
    LTrain = np.argmax(DTrain, axis=1)
    LTest  = np.argmax(DTest , axis=1)

    # Calculate initial metrics
    YTrain, LTrainPred, UTrain = forward(XTrain, W1, B1, W2, B2,␣
↪params["useTanhOutput"])
    YTest , LTestPred , _      = forward(XTest , W1, B1, W2, B2,␣
↪params["useTanhOutput"])

    # Including the initial metrics makes the progress plots worse, set nan to␣
↪exclude
    metrics["lossTrain"][0] = np.nan # ((YTrain - DTrain)**2).mean()
    metrics["lossTest"][0]  = np.nan # ((YTest  - DTest )**2).mean()
    metrics["accTrain"][0]  = np.nan # (LTrainPred == LTrain).mean()
    metrics["accTest"][0]   = np.nan # (LTestPred  == LTest ).mean()

    # Create figure for plotting progress
    fig = plt.figure(figsize=(20,8), tight_layout=True)

    # Training loop
    for n in range(1, params["epochs"]+1):

        # ------------------------------------------
        # === This is the important part ==========
        # === where your code is applied ==========
        # ------------------------------------------

        # Compute gradients...
        GradW1, GradB1, GradW2, GradB2 = backward(W1, B1, W2, B2, XTrain,␣
↪UTrain, YTrain, DTrain, params["useTanhOutput"])
```

```
        # ... and update weights
        W1, B1, W2, B2 = update(W1, B1, W2, B2, GradW1, GradB1, GradW2, GradB2,␣
↪params)

        # ============================================

        # Evaluate errors
        YTrain, LTrainPred, UTrain = forward(XTrain, W1, B1, W2, B2,␣
↪params["useTanhOutput"])
        YTest , LTestPred , _      = forward(XTest , W1, B1, W2, B2,␣
↪params["useTanhOutput"])
        metrics["lossTrain"][n] = ((YTrain - DTrain)**2).mean()
        metrics["lossTest"][n]  = ((YTest  - DTest )**2).mean()
        metrics["accTrain"][n]  = (LTrainPred == LTrain).mean()
        metrics["accTest"][n]   = (LTestPred  == LTest ).mean()

        # Plot progress
        if (plotProgress and not n % 500) or n == params["epochs"]:
            if W1.shape[0] == 2 and W1.shape[1] <= 8:
                plotProgressNetworkMulti(fig, W1, B1, W2, B2, metrics, n=n)
            else:
                plotProgress(fig, metrics, n)

    return W1, B1, W2, B2, metrics
```

We also define the same function for normalizing the data, which is even more important now that we have more than one layer.

[6]:
```
def normalize(X):
    # Compute mean and std
    m = X.mean(axis=0)
    s = X.std(axis=0)
    # Prevent division by 0 is feature has no variance
    s[s == 0] = 1
    # Return normalized data
    return (X - m) / s
```

Question

1. Explain why large, non-normalized input features might be a problem when using two layers, but not when using a single layer.

Answer: For Large, non-normalized input features causes significant issues in a two layer neural network. This includes gradients becoming excessively larger, which reduces stability in learning and gradients becoming too small. In contrast, single layer network, it is less effected due to direct evaludation of gradients and since there are no hidden layers, so the gradient passes directly from input to outputwhere compounding effects can occur.

1.0.5 2 Optimizing each dataset

Like before, we define a function that performs the boilerplate code for training the networks.

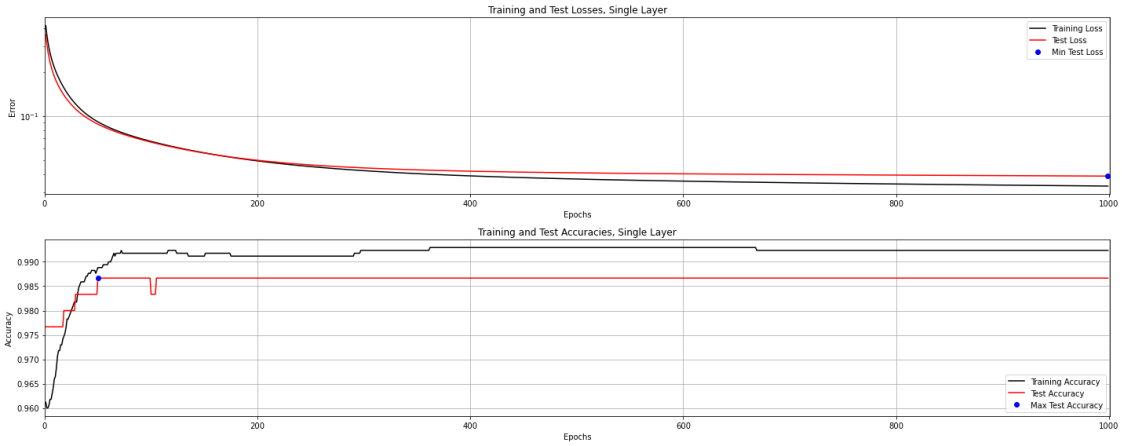
[7]:
```
def  trainMultiLayerOnDataset(datasetNr, testSplit, W1_0, B1_0, W2_0, B2_0,␣
↪params):
    """Train a two layer network on a specific dataset.

    Ags:
        datasetNr (int): ID of dataset to use
        testSplit (float): Fraction of data reserved for testing.
        W1_0 (array): Initial values of the first layer network weights.
        B1_0 (array): Initial values of the first layer network biases.
        W2_0 (array): Initial values of the second layer network weights.
        B2_0 (array): Initial values of the second layer network biases.
        params (dict): Dictionary containing:
            nIterations (int): Number of training steps.
            learningRate (float): Size of a training step.
    """

    # Load data and split into training and test sets
    X, D, L = loadDataset(datasetNr)
    XTrain, DTrain, LTrain, XTest, DTest, LTest = splitData(X, D, L, testSplit)

    if "normalize" in params and params["normalize"]:
        XTrainNorm = normalize(XTrain)
        XTestNorm  = normalize(XTest)
    else:
        XTrainNorm = XTrain
        XTestNorm  = XTest

    # Train network
    W1, B1, W2, B2, metrics = trainMultiLayer(XTrainNorm, DTrain, XTestNorm,␣
↪DTest, W1_0, B1_0, W2_0, B2_0, params)

    # Predict classes on test set
    LPredTrain = forward(XTrainNorm, W1, B1, W2, B2, params["useTanhOutput"])[1]
    LPredTest  = forward(XTestNorm , W1, B1, W2, B2, params["useTanhOutput"])[1]

    # Compute metrics
    accTrain = calcAccuracy(LPredTrain, LTrain)
    accTest  = calcAccuracy(LPredTest , LTest)
    confMatrix = calcConfusionMatrix(LPredTest, LTest)

    # Display results
    print(f'Train accuracy: {accTrain:.4f}')
```

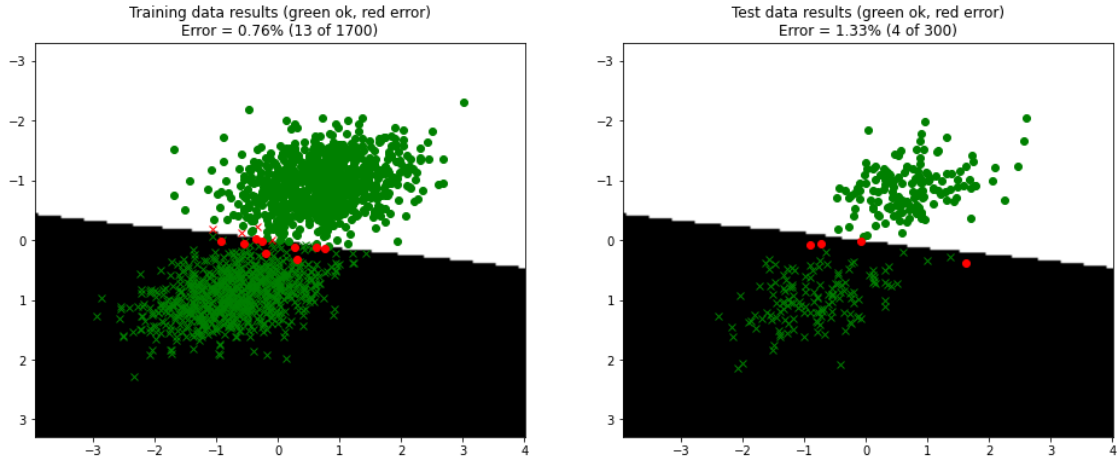
```
    print(f'Test accuracy: {accTest:.4f}')
    print("Test data confusion matrix:")
    print(confMatrix)

    if datasetNr < 4:
        # Switch between these two functions to see another way to visualize␣
↪the network output.
        plotResultsDots(XTrainNorm, LTrain, LPredTrain, XTestNorm, LTest,␣
↪LPredTest, lambda X: forward(X, W1, B1, W2, B2, params["useTanhOutput"])[1])
        #plotResultsDotsGradient(XTrainNorm, LTrain, LPredTrain, XTestNorm,␣
↪LTest, LPredTest, lambda X: forward(X, W1, B1, W2, B2,␣
↪params["useTanhOutput"])[0])
    else:
        plotConfusionMatrixOCR(XTest, LTest, LPredTest)
```

2.1 Optimizing dataset 1

[8]:
```
# ------------------------------------------
# === Your code here =======================
# ------------------------------------------
nInputs  = 2
nClasses = 2
nHidden  = 10

W1_0 = np.random.randn(nInputs, nHidden)
B1_0 = np.zeros((1, nHidden))
W2_0 = np.random.randn(nHidden, nClasses)
B2_0 = np.zeros((1, nClasses))

params = {"epochs": 1000, "learningRate": 0.05, "normalize": True,␣
 ↪"useTanhOutput": False}
# ==========================================

trainMultiLayerOnDataset(1, 0.15, W1_0, B1_0, W2_0, B2_0, params)
```

```
Train accuracy: 0.9924
Test accuracy: 0.9867
Test data confusion matrix:
[[139   4]
 [  0 157]]
```





Question:

2. Optimize the training until you reach at least 98% test accuracy. Briefly motivate your choice of hyperparameters.

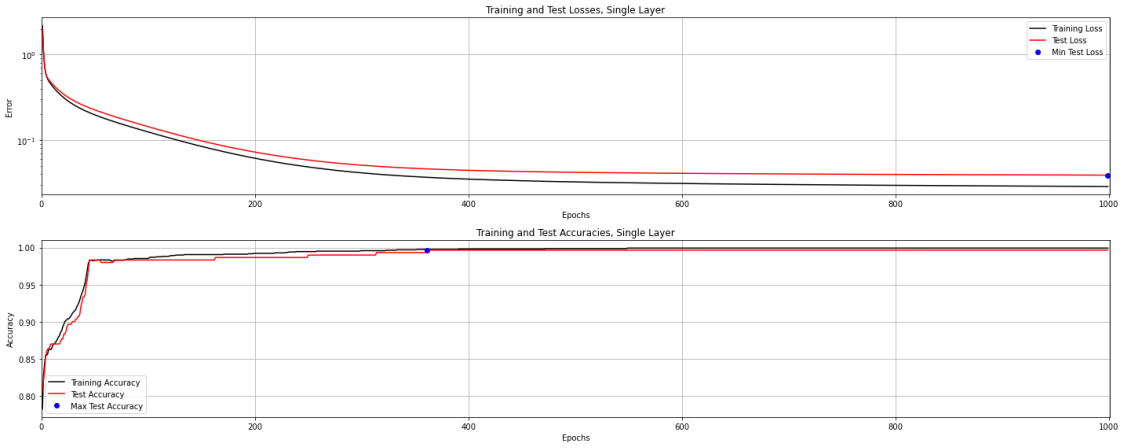
Answer: We kept the initial hyperparameters of `epochs = 1000`, `LR = 0.05`, `normalize = True`, `tanh = False` and as a rule of thumb chose 10 hidden neurons. We thought of this as the data was not complex and achieving a good accuracy was possible and we got accuracy of 99.67. Hence, the further optimization was not required.

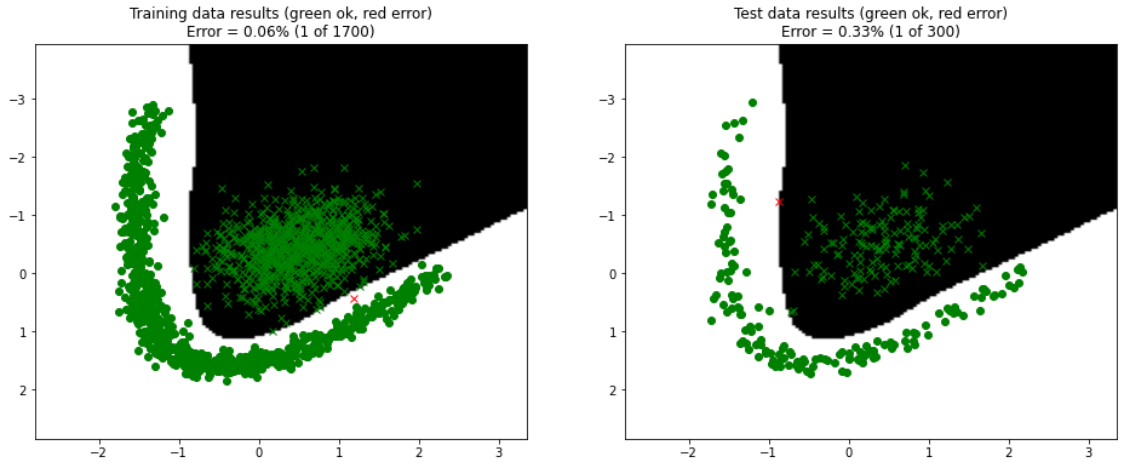
2.2 Optimizing dataset 2

[9]:
```
# ------------------------------------------
# === Your code here =======================
# ------------------------------------------
nInputs  = 2
```

```
nClasses = 2
nHidden  = 10

W1_0 = np.random.randn(nInputs, nHidden)
B1_0 = np.zeros((1, nHidden))
W2_0 = np.random.randn(nHidden, nClasses)
B2_0 = np.zeros((1, nClasses))

params = {"epochs": 1000, "learningRate": 0.05, "normalize": True,␣
  ↪"useTanhOutput": False}
# ============================================

trainMultiLayerOnDataset(2, 0.15, W1_0, B1_0, W2_0, B2_0, params)
```

```
Train accuracy: 0.9994
Test accuracy: 0.9967
Test data confusion matrix:
[[146   0]
 [  1 153]]
```





Question:

3. Optimize the training until you reach at least 99% test accuracy. Briefly motivate your choice of hyperparameters.

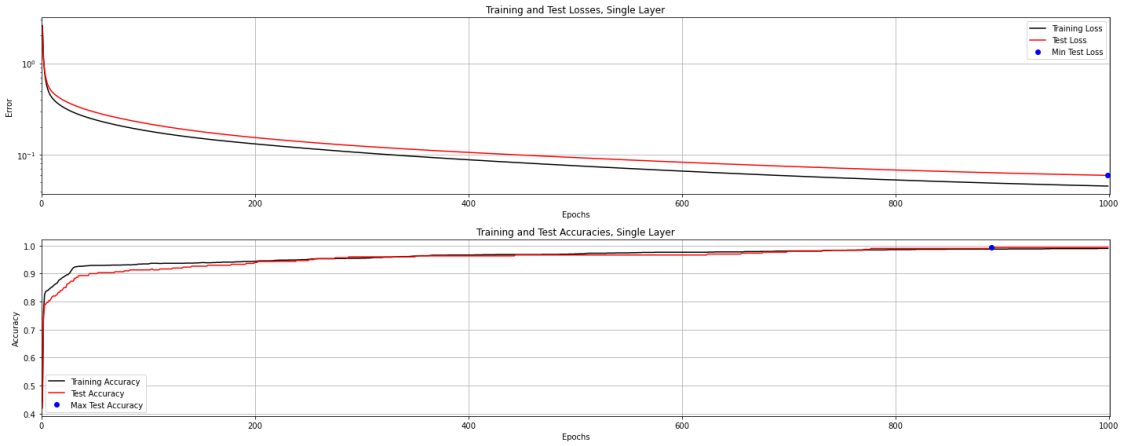
Answer: As stated in the previous dataset explanation, we set the initital hyperparameters and used 10 neurons as rule of thumb, and it was able to classify the points very easily getting 100% accuracy. But we think this is not ideal as these are signs of overfitting.

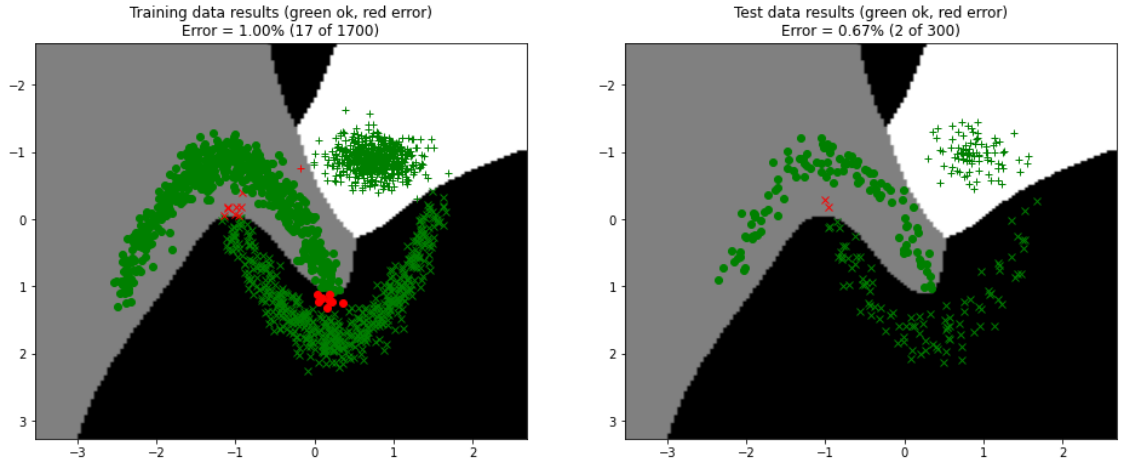
2.3 Optimizing dataset 3

[13]:
```
# ------------------------------------------
# === Your code here =======================
# ------------------------------------------
nInputs  = 2
nClasses = 3
nHidden  = 20

W1_0 = np.random.randn(nInputs, nHidden)
B1_0 = np.zeros((1, nHidden))
W2_0 = np.random.randn(nHidden, nClasses)
B2_0 = np.zeros((1, nClasses))

params = {"epochs": 1000, "learningRate": 0.03, "normalize": True,␣
 ↪"useTanhOutput": False}
# ==========================================

trainMultiLayerOnDataset(3, 0.15, W1_0, B1_0, W2_0, B2_0, params)
```

```
Train accuracy: 0.9900
Test accuracy: 0.9933
Test data confusion matrix:
```

```
[[ 98   0   0]
 [  2 111   0]
 [  0   0  89]]
```





Question:

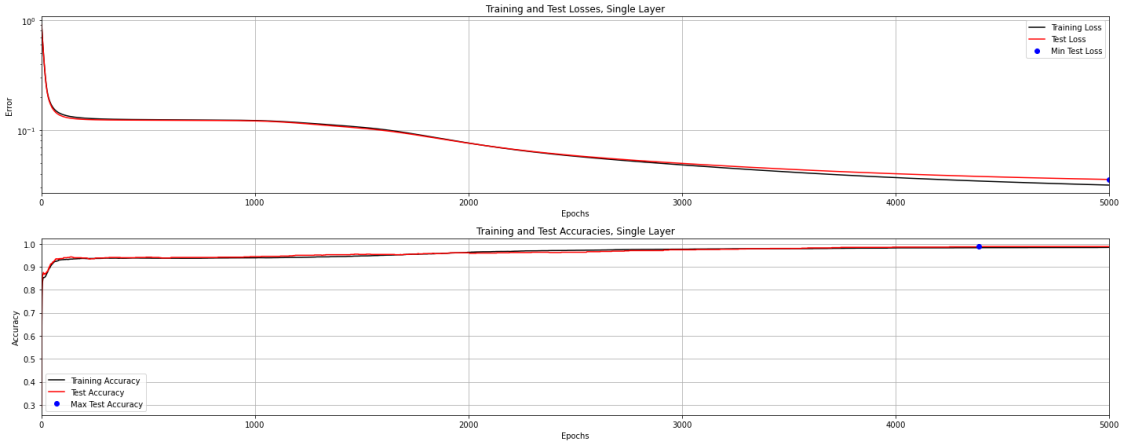
4. Optimize the training until you reach at least 99% test accuracy. Briefly motivate your choice of hyperparameters.

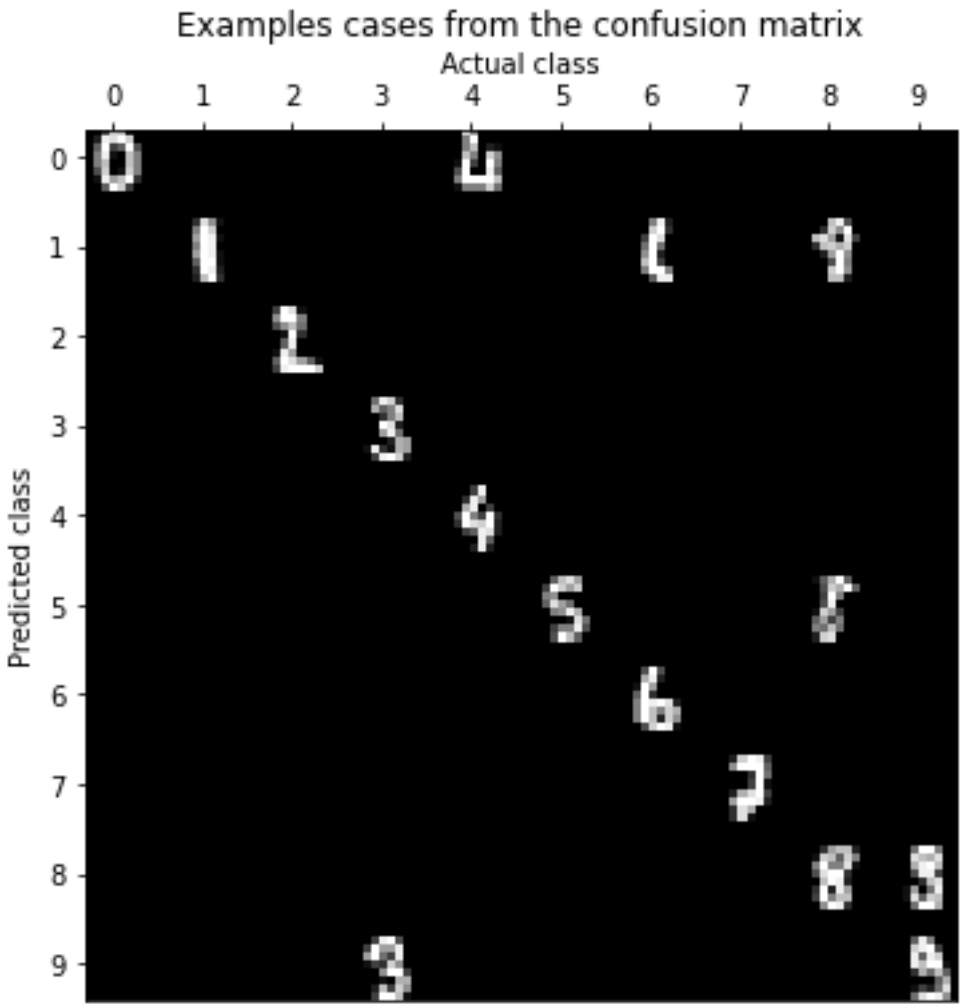
Answer: This data had 3 classes. We started with the same initial hyperparameters and data being separable with approximately 3 decision boundaries, we thought that the number of neurons as 10 should work for this classification. We achieved the accuracy of 99.3% using these hyperparameters, so we did not optimize it further.

2.4 Optimizing dataset 4

[11]:
```
# ----------------------------------------
# === Your code here =====================
# ----------------------------------------
nInputs  = 64
nClasses = 10
nHidden  = 500

W1_0 = np.random.randn(nInputs, nHidden) * 0.01
B1_0 = np.zeros((1, nHidden))
W2_0 = np.random.randn(nHidden, nClasses) * 0.01
B2_0 = np.zeros((1, nClasses))

params = {"epochs": 5000, "learningRate": 0.025, "normalize": True,␣
 ↪"useTanhOutput": False}
# ========================================

trainMultiLayerOnDataset(4, 0.15, W1_0, B1_0, W2_0, B2_0, params)
```

```
Train accuracy: 0.9843
Test accuracy: 0.9881
Test data confusion matrix:
[[92  0  0  0  1  0  0  0  0  0]
 [ 0 95  0  0  0  0  1  0  3  0]
 [ 0  0 68  0  0  0  0  0  0  0]
 [ 0  0  0 87  0  0  0  0  0  0]
 [ 0  0  0  0 80  0  0  0  0  0]
 [ 0  0  0  0  0 88  0  0  1  0]
 [ 0  0  0  0  0  0 82  0  0  0]
 [ 0  0  0  0  0  0  0 92  0  0]
 [ 0  0  0  0  0  0  0  0 80  1]
 [ 0  0  0  3  0  0  0  0  0 69]]
```





Question:

5. Optimize the training until you reach at least 96% test accuracy. Briefly motivate your choice of hyperparameters.

Answer: Scaled the initial weights by a small factor 0.01 to reduce to make sure data is not large. As this data had more input features than the previous datasets, we started with 64 neurons but were unable to achieve the desired accuracy. We further increased them to `500` and as the number of features were more to learn from, we set the `learning rate = 0.025` and ran it for more time by setting`epochs = 5000`. We also kept `normalize = True` so that the data is in the same scale achieving an accuracy of ~96%.

[ ]:

[ ]: