Feedforward and Backpropagation

Inclass Project 2 - MA4144

200222K - Himeka S.H.D

Outline of the project

The aim of the project is to build a Multi Layer perceptron (MLP) model from scratch for binary classification. That is given an input x output the associated class label 0 or 1.

In particular, we will classify images of handwritten digits $(0, 1, 2, \cdots, 9)$. For example, given a set of handwritten digit images that only contain two digits (Eg: 1 and 5) the model will classify the images based on the written digit.

For this we will use the MNIST dataset (collection of 28×28 images of handwritten digits) - you can find additional information about MNIST here.

Use the below cell to use any include any imports

```
In [1]: import numpy as np
        from keras.datasets import mnist
        import matplotlib.pyplot as plt
        from sklearn.metrics import accuracy_score
        from tqdm import tqdm
        from sklearn.metrics import accuracy_score, confusion_matrix
        import seaborn as sns
        import pandas as pd
```

Section 1: Preparing the data

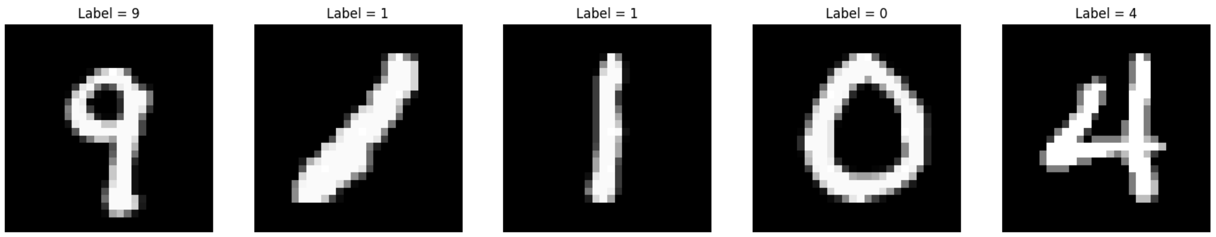
```
In [2]: #Load the dataset as training and testing, then print out the shapes of the data ma

        (train_X, train_y), (test_X, test_y) = mnist.load_data()
```

```
print(train_X.shape)
print(test_X.shape)
print(train_y.shape)
print(test_y.shape)
```

```
(60000, 28, 28)
(10000, 28, 28)
(60000,)
(10000,)
```

Q1. In the following cell write code to display 5 random images in train_X and it's corresponding label in train_y. Each time it is run, you should get a different set of images. The imshow function in the matplotlib library could be useful. Display them as grayscale images.

In [3]:
```
#TODO Code to display 5 random handritten images from train_X and corresponting lab
random_indices = np.random.choice(len(train_X), 5, replace=False)

plt.figure(figsize=(20,10))
for i, idx in enumerate(random_indices):
    plt.subplot(1, 5, i+1)
    plt.imshow(train_X[idx], cmap='gray')
    plt.title(f"Label = {train_y[idx]}")
    plt.axis('off')
plt.show()
```



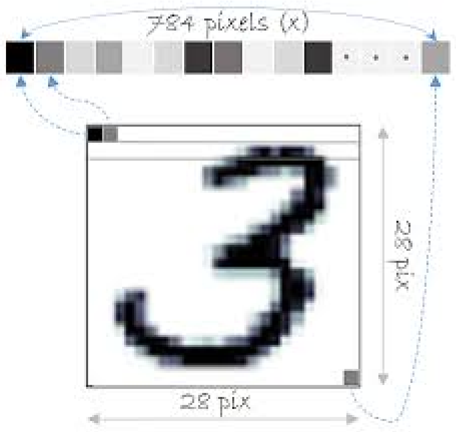
Q2. Given two digits d_1 and d_2, both between 0 and 9, in the following cell fill in the function body to extract all the samples corresponding to d_1 or d_2 only, from the dataset X and labels y. You can use the labels y to filter the dataset. Assume that the ith image $X[i]$ in X is given by $y[i]$. The function should return the extracted samples $X_{extracted}$ and corresponding labels $y_{extracted}$. Avoid using for loops as much as possible, infact you do not need any for loops. numpy.where function should be useful.

In [4]:
```
def extract_digits(X, y, d1, d2):

    assert d1 in range(0, 10), "d1 should be a number between 0 and 9 inclusive"
    assert d2 in range(0, 10), "d2 should be a number between 0 and 9 inclusive"

    #TODO
    indices = np.where((y == d1) | (y == d2))

    X_extracted = X[indices]
    y_extracted = y[indices]
    return (X_extracted, y_extracted)
```

Q3. Both the training dataset train_X and test_y is a 3 dimensional numpy array, each image occupies 2 dimensions. For convenience of processing data we usually comvert each 28×28 image matrix to a vector with 784 entries. We call this process **vectorize images**.

Once we vectorize the images, the vectorized data set would be structured as follows: ith row will correspond to a single image and jth column will correspond to the jth pixel value of each vectorized image. However going along with the convention we discussed in the lecture, the input to the MLP model will require that the columns correspond to individual images. Hence we also require a transpose of the vectorized results.

The pixel values in the images will range from 0 to 255. Normalize the pixel values between 0 and 1, by dividing each pixel value of each image by the maximum pixel value of that image. Simply divide each column of the resulting matrix above by the max of each column.



Given a dataset X of size $N \times 28 \times 28$, in the following cell fill in the function to do the following in order;

1. Vectorize the dataset resulting in dataset of size $N \times 784$.
2. Transpose the vectorized result.
3. Normalize the pixel values of each image.
4. Finally return the vectorized, transposed and normalized dataset $X_{transformed}$.

Again, avoid for loops, functions such as numpy.reshape, numpy.max etc should be useful.

In [5]:
```
def vectorize_images(X):

    #TODO
    #vectorize the image data
    X_vectorized = X.reshape(X.shape[0], X.shape[1]*X.shape[2])

    X_transposed = X_vectorized.T
    col_max = np.max(X_transposed, axis=0)
    X_transformed = X_transposed/col_max
    return(X_transformed)
```

Q4. In the following cell write code to;

1. Extract images of the digits $d_1 = 1$ and $d_2 = 5$ with their corresponding labels for both the training set (train_X, train_y) and testing set (test_X, test_y) separately.
2. Then vectorize the data, tranpose the result and normlize the images.
3. Store the results after the final transformations in numpy arrays train_X_1_5, train_y_1_5, test_X_1_5, test_y_1_5
4. Our MLP will output only class labels 0 and 1 (not 1 and 5), so create numpy arrays to store the class labels as follows: $d_1 = 1$ -> class label = 0 and $d_2 = 5$ -> class label = 1. Store them in the arrays named train_class_1_5, test_class_1_5.

Use the above functions you implemented above to complete this task. In addtion, numpy.where could be useful. Avoid for loops as much as possible.

In [6]:
```
def full_process_data(X_train, y_train, X_test, y_test, d1, d2):
    # Extract the digits d1 and d2
    X_train_extracted, y_train_extracted = extract_digits(X_train, y_train, d1, d2)
    X_test_extracted, y_test_extracted = extract_digits(X_test, y_test, d1, d2)

    # Vectorize the images
    X_train_transformed = vectorize_images(X_train_extracted)
    X_test_transformed = vectorize_images(X_test_extracted)

    return X_train_transformed, y_train_extracted, X_test_transformed, y_test_extra

# Call the function
train_X_1_5, train_y_1_5, test_X_1_5, test_y_1_5 = full_process_data(train_X, train

# Convert labels: d1 -> 0 and d2 -> 1
train_class_1_5 = np.where(train_y_1_5 == 1, 0, 1)
test_class_1_5 = np.where(test_y_1_5 == 1, 0, 1)
```

Section 2: Implementing MLP from scratch with training algorithms.

Now we will implement code to build a customizable MLP model. The hidden layers will have the **Relu activation function** and the final output layer will have **Sigmoid activation function**.

Q5. Recall the following about the activation functions:

1. Sigmoid activation: $y = \sigma(z) = \frac{1}{1+e^{-z}}$.
2. Derivative of Sigmoid: $y' = \sigma'(z) = \sigma(z)(1 - \sigma(z)) = y(1 - y)$
3. ReLu activation: $y = ReLu(z) = max(0, z)$
4. Derivative of ReLu: $y' = ReLu'(z) = \begin{cases} 0 \text{ if } z < 0 \\ 1 \text{ otherwise} \end{cases} = \begin{cases} 0 \text{ if } y = 0 \\ 1 \text{ otherwise} \end{cases}$

In the following cell implement the functions to compute activation functions Sigmoid and ReLu given z and derivatives of the Sigmoid and ReLu activation functions given y. Note that the input to the derivative functions is y not z.

In practice the input will not be just single numbers, but matrices. So functions or derivatives should be applied elementwise on matrices. Again avoid for loops, use the power of numpy arrays - search for numpy's capability of doing elementwise computations.

Important: When implementing the sigmoid function make sure you handle overflows due to e^{-z} being too large. To avoid you can choose to set the sigmoid value to 'the certain appropriate value' if z is less than a certain good enough negative threshold. If you do not handle overflows, the entire result will be useless fince the MLP will just output Nan (not a number) for evry input at the end.

In [7]:
```
def sigmoid(Z):


    threshold = 500

    Z_clipped = np.clip(Z, -threshold, threshold)

    sigma = 1 / (1 + np.exp(-Z_clipped))

    return(sigma)

def deriv_sigmoid(Y):


    sigma_prime = Y*(1-Y)
    return(sigma_prime)

def ReLu(Z):


    relu = np.where(Z < 0, 0 , Z)
    return(relu)

def deriv_ReLu(Y):


    relu_prime=np.where(Y==0,0,1)
    return(relu_prime)
```

Q6. The following piece of code defines a simple MLP architecture as a Python class and subsequent initialization of a MLP model. Certain lines of code contains commented line numbers. Write a short sentence for each such line explaining its purpose. Feel free to refer to the lecture notes or any resources to answers these question. In addition, explain what the Y, Z, W variables refer to and their purpose

In [8]:
```
class NNet:
    def __init__(self, input_size = 784, output_size = 1, batch_size = 1000, hidden
        self.Y = []
        self.Z = []
        self.W = []
        self.input_size = input_size
        self.output_size = output_size
        self.batch_size = batch_size
        self.hidden_layers = hidden_layers

        layers = [input_size] + hidden_layers + [output_size]
        L = len(hidden_layers) + 1

        for i in range(1, L + 1):
            self.Y.append(np.zeros((layers[i], batch_size)))
            self.Z.append(np.zeros((layers[i], batch_size)))
            self.W.append(2*(np.random.rand(layers[i], layers[i-1] + 1) - 0.5))
```

Answers

(i) **What does the Y, Z, W variables refer to and their purpose?**

- **Y**: Outputs of the activation function when the weighted sum plus the bias term is given as input. It represents the activated outputs of a layer in the network.
- **Z**: Linear combination of the weighted sum plus the bias term. It is the input to the activation function and represents the raw output from the neurons before applying the activation function.
- **W**: Weights of the network. These are the parameters that the network learns during training, representing the connections between neurons in different layers.

(ii) **Line1**: Initialize the `Y` values with zeros to suit the network dimensions. For each layer (iteration), we append an array of size `(number of neurons, batch size)` with zeros.

(iii) **Line 2**: Initialize the `Z` values with zeros to suit the network dimensions. For each layer (iteration), we append an array of size `(number of neurons, batch size)` with zeros.

(iv) **Line 3**: Initialize the weights with random numbers in the range ([-1, 1]) with dimensions `(number of neurons in current layer, number of neurons in previous layer)`. For each layer (iteration), we append an array with these dimensions to `W`.

Q7. Now we will implement the feedforward algorithm. Recall from the lectures that for each layer l there is input $Y^{(l-1)}$ from the previous layer if $l > 1$ and input data X if $l = 1$. Then we compute $Z^{(l)}$ using the weight matrix $W^{(l)}$ as follows from matrix multiplication:

$$Z^{(l)} = W^{(l)} Y^{(l-1)}$$

Make sure that during multiplication you add an additional row of one's to $Y^{(l-1)}$ to accommodate the bias term. However, the rows of ones should not permanently remain on $Y^{(l-1)}$. Explain what the bias term is and how adding a row of one's help with the bias terms. During definition above the weight matrices are initialised to afford this extra bias term, so no change to either $Z^{(l)}$ or $W^{(l)}$ is needed.

Next compute $Y^{(l)}$, the output of layer l by activation through sigmoid.

$Y^{(l)} = \sigma(Z^{(l)})$

The implemented feedforward algorithm should take in a NNet model and an input matrix X and output the modified MLP model - the Y's and Z's computed should be stored in the model for the backpropagation algorithm.

As usual avoid for loops as much as possible, use the power of numpy. However, you may use a for loop to iterate through the layers of the model.

```
In [ ]:
```

```
In [9]: def feedforward(model, X):

            #TODO
            #Implement the feedforward algorithm

            X=np.vstack((X, np.ones(X.shape[1])))
            model.Z[0] = np.dot(model.W[0], X)
            model.Y[0] = ReLu(model.Z[0])

            for i in range(1, len(model.Y)):
                model.Z[i] = np.dot(model.W[i], np.vstack((model.Y[i-1], np.ones(model.Y[i-
                if i == len(model.Y) - 1:
                    model.Y[i] = sigmoid(model.Z[i])
                else:
                    model.Y[i] = ReLu(model.Z[i])



            return(model)
```

Answer

Bias Term : Additional parameter which is added to the weighted sum, to make sure neurons adjusts independently of its inputs. Needed this for the model to fit the data better.

Row of ones helps matching the dimensions, since we need to convert W * X + bias into matrix form. W matrix contains bias, and Row of ones into the input matrix help matching the dimension of output matrix Z

Q8. Now we will implement the backpropagation algorithm. The cost function C at the end is given by the square loss.

$C = \frac{1}{2}||Y^{(L)} - Y||^2$, where $Y^{(L)}$ is the final output vector of the feedforward algorithm and Y is the actual label vector associated with the input X.

At each layer $l = 1, 2, \cdots, L$ we compute the following (note that the gradients are matrices with the same dimensions as the variable to which we derivating with respect to):

1. Gradient of C with respect to $Z^{(l)}$ as
   $\frac{\partial C}{\partial Z^{(l)}} = deriv(A^{(l)}(Z^{(l)})) \odot \frac{\partial C}{\partial Y^{(L)}}$,
   where $A^{(l)}$ is the activation function of the lth layer, and we use the derivative of that here. The $\odot$ refers to the elementwise multiplication.

2. Gradient of C with respect to $W^{(l)}$ as
   $\frac{\partial C}{\partial W^{(l)}} = \frac{\partial C}{\partial Z^{(l)}}(Y^{(l-1)})^T$
   this is entirely matrix multiplication.

3. Gradient of C with respect to $Y^{(l-1)}$ as
   $\frac{\partial C}{\partial Y^{(l-1)}} = (W^{(l)})^T \frac{\partial C}{\partial Z^{(l)}}$
   this is also entirely matrix multiplication.

4. Update weights by:
   $W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial C}{\partial W^{(l)}}$,
   where $\eta > 0$ is the learning rate.

The loss derivative (the gradient of C with respect to $Y^{(L)}$) at the last layer is given by:

$$\frac{\partial C}{\partial Y^{(L)}} = Y^{(L)} - Y$$

By convention we consider $Y^{(0)} = X$, the input data.

Based on the backpropagation algorithm implement the backpropagation method in the following cell. Remember to temporarily add a row of ones to $Y^{(l-1)}$ when computing $\frac{\partial C}{\partial W^{(l)}}$ as we discussed back in the feedforward algorithm. Make sure you avoid for loops as much as possible.

The function takes in a NNet model, input data X and the corresponding class labels Y. learning rate can be set as desired.

In [10]:
```python
def backpropagation(model, X, Y, eta = 0.01):
    # Gradient of loss for the last layer
    dC_dY_L = (model.Y[-1] - Y)


    dC_dW = [np.zeros_like(w) for w in model.W]

    num_layers = len(model.hidden_layers) + 1
    for i in range(num_layers-1 , -1, -1):
        # If last layer
        if i == num_layers - 1:
            dC_dZ_l = deriv_sigmoid(model.Y[i]) * dC_dY_L
        # Hidden Layers
        else:
            dC_dZ_l = deriv_ReLu(model.Y[i]) * dC_dZ_l
        dC_dW[i] = np.dot(dC_dZ_l, np.vstack((model.Y[i - 1], np.ones(model.Y[i - 1

        # Propagate loss to previous layer
        if i > 0:
            W_without_bias = model.W[i][:, :-1]  # Exclude the bias term
            dC_dZ_l = np.dot(W_without_bias.T, dC_dZ_l)


        # Update weights
        model.W[i] -= eta * dC_dW[i]

    return model
```

Q9. Now implement the training algorithm.

The training method takes in training data X, actual label Y, number of epochs, batch_size, learning rate $\eta > 0$. The training will happen in epochs. For each epoch, permute the data columns of both X and Y, then divide both X and Y into mini batches each with the given batch size. Then run the feedforward and backpropagation for each such batch iteratively.

At the end of each iteration, keep trach of the cost C and the l_2-norm of change in each weight matrix $W^{(l)}$.

At the end of the last epoch, plot the variation cost C and change in weight matrices. Then return the trained model.

```
In [11]: def train_NNet(X, Y, epochs=20, batch_size=1000, eta=0.01, hidden_layers=[500, 250,

    # Set the seed for reproducibility
    np.random.seed(42)

    # Initialize the model
    model = NNet(input_size=X.shape[0], output_size=1, batch_size=batch_size, hidde

    # Number of data points in dataset
    num_samples = X.shape[1]

    # Array to store loss history and weight change history
    loss_history = []
    avg_weight_change = []

    progress = tqdm(range(epochs), desc="Training", unit="epoch")
    for epoch in progress:
        epoch_cost = 0
        total_weight_change = 0

        # Shuffle data
        indices = np.random.permutation(num_samples)
        X_shuffled = X[:, indices]
        Y_shuffled = Y[indices]

        num_mini_batches = num_samples // batch_size
        batch_wise_loss = np.array([])
        # Process mini-batches
        for j in range(num_mini_batches):
            start = j*batch_size
            end = min(start + batch_size, num_samples)
            X_batch = X_shuffled[:, start:end]
            Y_batch = Y_shuffled[start:end]


            # Feedforward
            model = feedforward(model, X_batch)

            # Store old weights for weight change calculation
            old_weights = [np.copy(W) for W in model.W]

            # Backpropagation
            model = backpropagation(model, X_batch, Y_batch, eta)

            # Compute cost for the batch
            batch_cost = 0.5 * np.linalg.norm(model.Y[-1] - Y_batch) ** 2
            batch_wise_loss = np.append(batch_wise_loss, batch_cost)

            # Track change in weights using L2 norm
            weight_change_per_batch = sum(np.linalg.norm(old_weights[k] - model.W[k
            total_weight_change += weight_change_per_batch

        # Average epoch cost and store
        avg_cost = np.mean(batch_wise_loss) / batch_size
        loss_history.append(avg_cost)

        # Average weight change over the epoch
        avg_weight_change_epoch = total_weight_change / (num_mini_batches * len(mod
        avg_weight_change.append(avg_weight_change_epoch)

        # Update progress bar with epoch cost
        progress.set_postfix({"Epoch": epoch + 1, "Loss per epoch": avg_cost})

    if plot_true:
        # Plot loss and weight changes
        fig, ax = plt.subplots(1, 2, figsize=(25, 8))

        ax[0].plot(loss_history,label='Training Loss', color='r')
        ax[0].set_xlabel('Epochs')
        ax[0].set_ylabel('Loss')
        ax[0].set_title('Loss vs Epochs')
        ax[0].grid(True)

        ax[1].plot(avg_weight_change,label='Weight Differences', color='b')
        ax[1].set_xlabel('Epochs')
        ax[1].set_ylabel('Weight change (L2 norm)')
        ax[1].set_title('Weight change vs Epochs')
        ax[1].grid(True)

        plt.tight_layout()
        plt.show()
    return model
```

Section 3: Evaluation using test data

The following function will evaluate then return an accuracy score and the predicted labels for your model. Do not change anything here.

In [12]:
```
def test_model(test_data, test_labels, model, d1, d2):

    L = len(model.hidden_layers) + 1

    Y = test_data
    for i in range(L):
        Z = np.matmul(model.W[i], np.append(Y, np.array([np.ones(Y.shape[1])]), axi
        if i < L - 1:
            Y = ReLu(Z)
        else:
            Y = sigmoid(Z)

    Y = Y[0]
    Y = np.where(Y >= 0.5, 1, 0)
    Y_predicted = np.where(Y == 0, d1, d2)

    acc = accuracy_score(test_labels, Y_predicted)

    return(acc, Y_predicted)
```

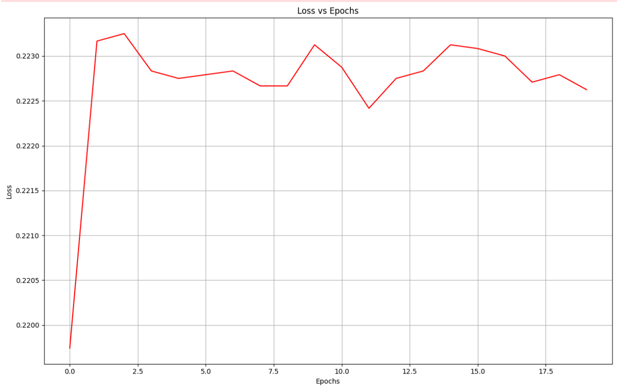
In [13]:
```
def plot_confusion_matrix(test_data, predicted_data, labels):
  cm = confusion_matrix(y_true=test_data, y_pred= predicted_data)

  cm_df = pd.DataFrame(cm,
                      index=labels,
                      columns=labels)

  # Plot the confusion matrix
  plt.figure(figsize=(8, 6))
  sns.heatmap(cm_df, annot=True, fmt='d', cmap='Blues', cbar=False,
              xticklabels=labels, yticklabels=labels)
  plt.title('Confusion Matrix')
  plt.xlabel('Predicted Label')
  plt.ylabel('True Label')
  plt.show()
```

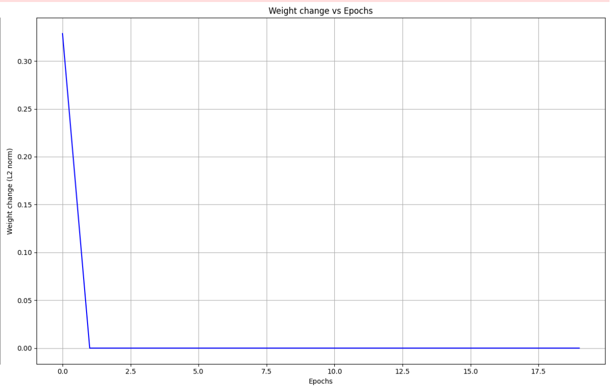
Q10. Use this test_model function to evaluate your model with the 1 and 5 digits. An accuracy $>= 99$ is achievable. Test with different batch sizes, η values and hidden layers. Find which of those hyperparameters gives the best test accuracy. Document the hyperparameter values that gives the best testing accuracy and that best accuracy. Plot a confusion matrix and comment on your observations with reasons. Also, look into the nature of the plots that result fom the training procedure, see how they vary with the hyperparameters and provide your ideas on the observations. Then do the same with a few other pairs of digits d_1, d_2. Especially, try $d_1 = 1, d_2 = 7$. Comment on your observations and explain possible reasons.

Training with default parameters

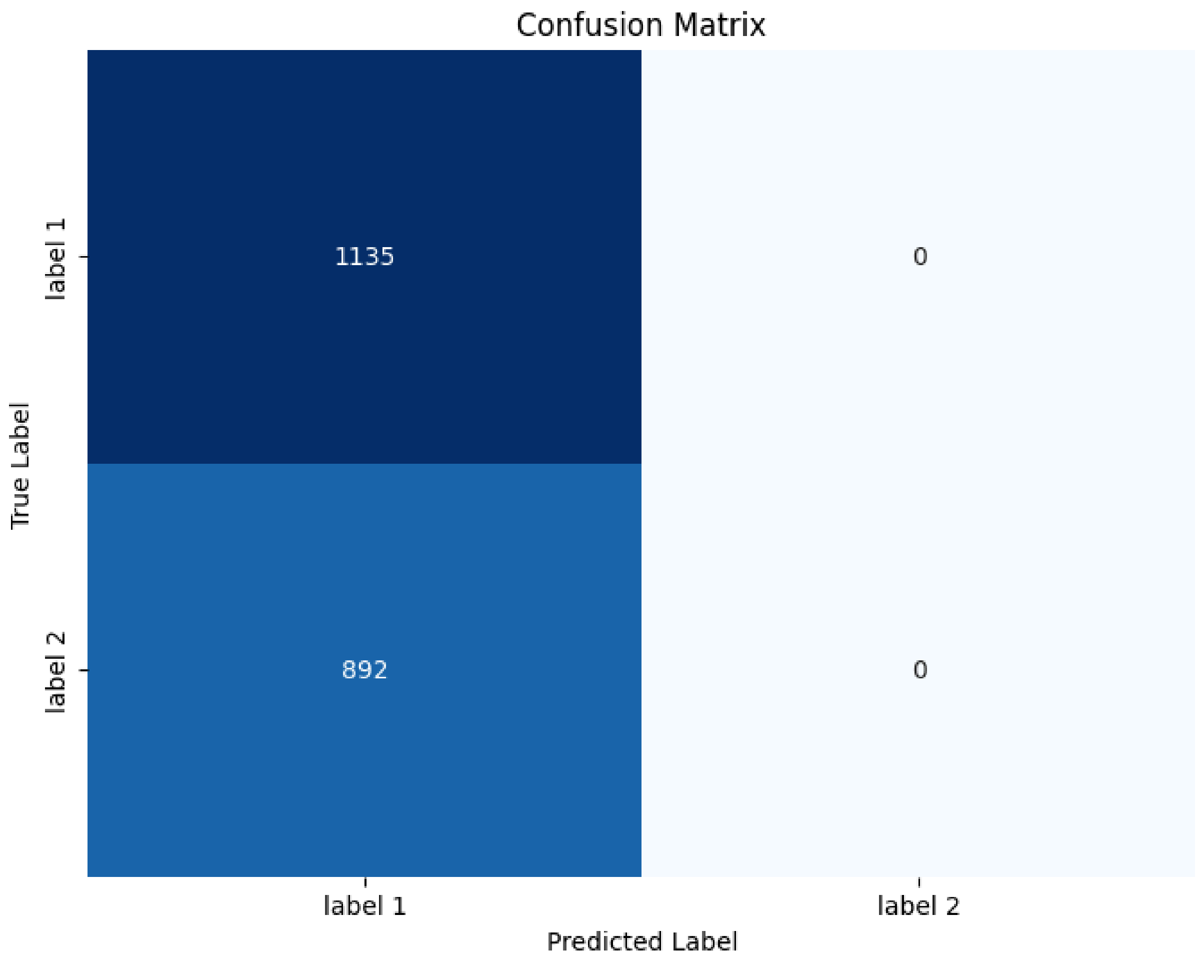
```
In [14]: default =  train_NNet(train_X_1_5, train_class_1_5,plot_true=True)
         #test
         accuracy, y_pred = test_model(test_X_1_5, test_y_1_5, default, 1, 5)
         print(f"Accuracy - {accuracy*100:.2f}% \n")
         plot_confusion_matrix(test_y_1_5, y_pred, ['label 1', 'label 2'])
```

```
Training:   0%|          | 0/20 [00:00<?, ?epoch/s]Training: 100%|██████████| 20/20
[00:41<00:00,  2.06s/epoch, Epoch=20, Loss per epoch=0.223]
```





```
Accuracy - 55.99%
```



We can see model has not learnt anything with default hyperparameter, just predicted every label as label1. The accuracy of the model is still considerably low. With proper hyperparameter tuning we can decrease loss and get better predictions

Getting Optimal hyperparameter set

In [15]:
```
import itertools

def hyperparameter_grid_search_optimized(train_X, train_y, test_X, test_y, grid, d1
    # Generate all combinations of hyperparameters using itertools.product
    param_combinations = list(itertools.product(grid['batch_size'], grid['hidden_la

    best_accuracy = 0
    best_model = None
    best_parameters = None

    # Loop through the combinations
    for batch_size, hidden_layers, eta, epochs in param_combinations:


        model = train_NNet(train_X, train_y, epochs=epochs, batch_size=batch_size,

        # Evaluate the model on the test set
        accuracy, _ = test_model(test_X, test_y, model, d1, d2)
        # Output the parameters and accuracy for tracking
        print(f" --- Hyperparameter configuration --- \n")
        print(f"Parameters: batch_size={batch_size}, hidden_layers={hidden_layers},
        print(f"Test Accuracy: {accuracy * 100:.2f}%\n")

        # Update best model if accuracy is higher
        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_model = model
            best_parameters = {
                'batch_size': batch_size,
                'hidden_layers': hidden_layers,
                'eta': eta,
                'epochs': epochs
            }

    # Output the best parameters and accuracy
    print("Best Model Summary:")
    print(f"Best Accuracy: {best_accuracy * 100:.2f}%")
    print(f"Best Hyperparameters: {best_parameters}")

    return best_model, best_parameters, best_accuracy
```

In [16]:
```
# Define the hyperparameter grid
grid = {
    'batch_size': np.array([1000, 1500,2000]),
    'hidden_layers': [[500, 250, 50],[500,250]],
    'eta_values': np.array([0.01, 0.001]),
    'epochs': np.array([50,100])
}

# Execute the optimized grid search on the 1 vs 5 digits task
best_model, best_params, best_acc = hyperparameter_grid_search_optimized(train_X_1_
```

```
Training:   0%|          | 0/50 [00:00<?, ?epoch/s]Training: 100%|██████████| 50/50
[01:41<00:00,  2.02s/epoch, Epoch=50, Loss per epoch=0.223]
 --- Hyperparameter configuration --- 

Parameters: batch_size=1000, hidden_layers=[500, 250, 50], eta=0.01, epochs=50
Test Accuracy: 55.99%

Training: 100%|██████████| 100/100 [02:52<00:00,  1.72s/epoch, Epoch=100, Loss per e
poch=0.223]
 --- Hyperparameter configuration --- 

Parameters: batch_size=1000, hidden_layers=[500, 250, 50], eta=0.01, epochs=100
Test Accuracy: 55.99%

Training: 100%|██████████| 50/50 [01:20<00:00,  1.61s/epoch, Epoch=50, Loss per epoc
h=0.00746]
 --- Hyperparameter configuration --- 

Parameters: batch_size=1000, hidden_layers=[500, 250, 50], eta=0.001, epochs=50
Test Accuracy: 98.57%
```

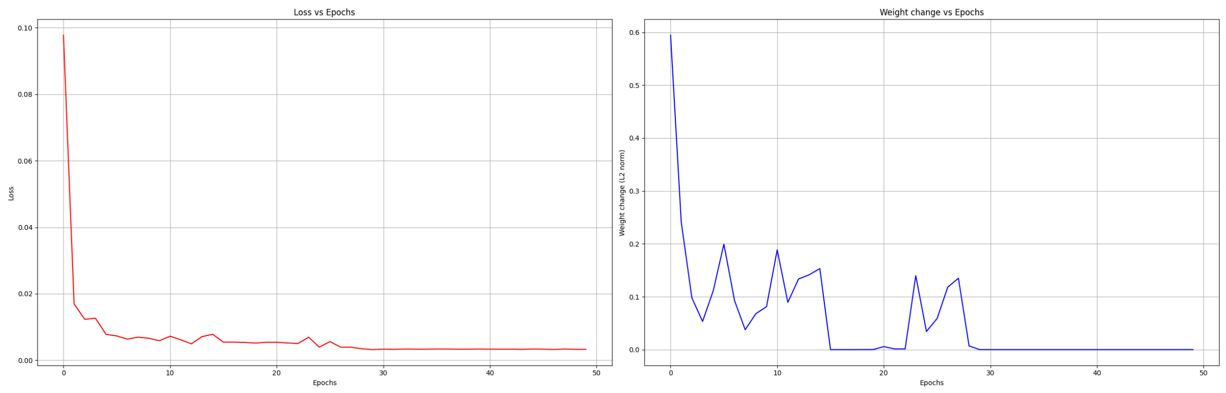
```
Training: 100%|██████████| 100/100 [02:51<00:00,  1.71s/epoch, Epoch=100, Loss per e
poch=0.00562]
 --- Hyperparameter configuration ---

Parameters: batch_size=1000, hidden_layers=[500, 250, 50], eta=0.001, epochs=100
Test Accuracy: 99.11%

Training: 100%|██████████| 50/50 [01:17<00:00,  1.55s/epoch, Epoch=50, Loss per epoc
h=0.00329]
 --- Hyperparameter configuration ---

Parameters: batch_size=1000, hidden_layers=[500, 250], eta=0.01, epochs=50
Test Accuracy: 99.65%

Training: 100%|██████████| 100/100 [02:31<00:00,  1.52s/epoch, Epoch=100, Loss per e
poch=0.00337]
 --- Hyperparameter configuration ---

Parameters: batch_size=1000, hidden_layers=[500, 250], eta=0.01, epochs=100
Test Accuracy: 99.65%

Training: 100%|██████████| 50/50 [01:40<00:00,  2.02s/epoch, Epoch=50, Loss per epoc
h=0.00442]
 --- Hyperparameter configuration ---

Parameters: batch_size=1000, hidden_layers=[500, 250], eta=0.001, epochs=50
Test Accuracy: 99.01%

Training: 100%|██████████| 100/100 [02:56<00:00,  1.77s/epoch, Epoch=100, Loss per e
poch=0.00446]
 --- Hyperparameter configuration ---

Parameters: batch_size=1000, hidden_layers=[500, 250], eta=0.001, epochs=100
Test Accuracy: 99.01%

Training: 100%|██████████| 50/50 [01:03<00:00,  1.26s/epoch, Epoch=50, Loss per epoc
h=0.223]
 --- Hyperparameter configuration ---

Parameters: batch_size=1500, hidden_layers=[500, 250, 50], eta=0.01, epochs=50
Test Accuracy: 55.99%

Training: 100%|██████████| 100/100 [02:03<00:00,  1.24s/epoch, Epoch=100, Loss per e
poch=0.223]
 --- Hyperparameter configuration ---

Parameters: batch_size=1500, hidden_layers=[500, 250, 50], eta=0.01, epochs=100
Test Accuracy: 55.99%

Training: 100%|██████████| 50/50 [01:02<00:00,  1.26s/epoch, Epoch=50, Loss per epoc
h=0.00437]
```

```
 --- Hyperparameter configuration ---

Parameters: batch_size=1500, hidden_layers=[500, 250, 50], eta=0.001, epochs=50
Test Accuracy: 99.11%

Training: 100%|██████████| 100/100 [02:08<00:00,  1.28s/epoch, Epoch=100, Loss per e
poch=0.0045]
 --- Hyperparameter configuration ---

Parameters: batch_size=1500, hidden_layers=[500, 250, 50], eta=0.001, epochs=100
Test Accuracy: 99.26%

Training: 100%|██████████| 50/50 [00:57<00:00,  1.16s/epoch, Epoch=50, Loss per epoc
h=0.00354]
 --- Hyperparameter configuration ---

Parameters: batch_size=1500, hidden_layers=[500, 250], eta=0.01, epochs=50
Test Accuracy: 99.41%

Training: 100%|██████████| 100/100 [01:57<00:00,  1.18s/epoch, Epoch=100, Loss per e
poch=0.00358]
 --- Hyperparameter configuration ---

Parameters: batch_size=1500, hidden_layers=[500, 250], eta=0.01, epochs=100
Test Accuracy: 99.41%

Training: 100%|██████████| 50/50 [01:14<00:00,  1.49s/epoch, Epoch=50, Loss per epoc
h=0.00458]
 --- Hyperparameter configuration ---

Parameters: batch_size=1500, hidden_layers=[500, 250], eta=0.001, epochs=50
Test Accuracy: 99.11%

Training: 100%|██████████| 100/100 [02:23<00:00,  1.43s/epoch, Epoch=100, Loss per e
poch=0.00429]
 --- Hyperparameter configuration ---

Parameters: batch_size=1500, hidden_layers=[500, 250], eta=0.001, epochs=100
Test Accuracy: 99.01%

Training: 100%|██████████| 50/50 [01:00<00:00,  1.20s/epoch, Epoch=50, Loss per epoc
h=0.223]
 --- Hyperparameter configuration ---

Parameters: batch_size=2000, hidden_layers=[500, 250, 50], eta=0.01, epochs=50
Test Accuracy: 55.99%

Training: 100%|██████████| 100/100 [02:01<00:00,  1.21s/epoch, Epoch=100, Loss per e
poch=0.223]
 --- Hyperparameter configuration ---

Parameters: batch_size=2000, hidden_layers=[500, 250, 50], eta=0.01, epochs=100
Test Accuracy: 55.99%
```

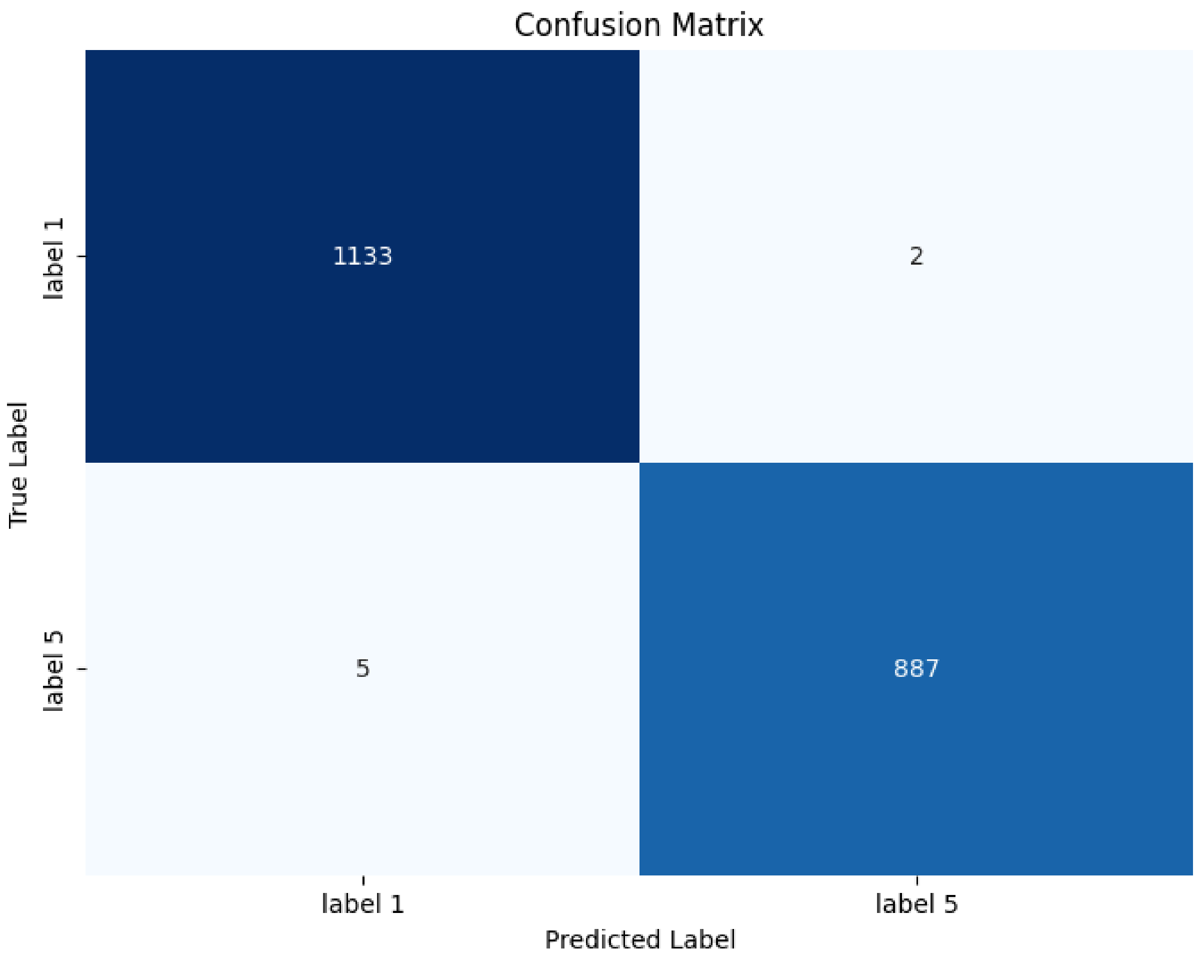
```
Training: 100%|██████████| 50/50 [01:11<00:00,  1.42s/epoch, Epoch=50, Loss per epoc
h=0.00425]
 --- Hyperparameter configuration ---

Parameters: batch_size=2000, hidden_layers=[500, 250, 50], eta=0.001, epochs=50
Test Accuracy: 99.21%

Training: 100%|██████████| 100/100 [02:08<00:00,  1.29s/epoch, Epoch=100, Loss per e
poch=0.00429]
 --- Hyperparameter configuration ---

Parameters: batch_size=2000, hidden_layers=[500, 250, 50], eta=0.001, epochs=100
Test Accuracy: 99.21%

Training: 100%|██████████| 50/50 [01:13<00:00,  1.47s/epoch, Epoch=50, Loss per epoc
h=0.0065]
 --- Hyperparameter configuration ---

Parameters: batch_size=2000, hidden_layers=[500, 250], eta=0.01, epochs=50
Test Accuracy: 99.21%

Training: 100%|██████████| 100/100 [02:25<00:00,  1.46s/epoch, Epoch=100, Loss per e
poch=0.00658]
 --- Hyperparameter configuration ---

Parameters: batch_size=2000, hidden_layers=[500, 250], eta=0.01, epochs=100
Test Accuracy: 99.21%

Training: 100%|██████████| 50/50 [01:24<00:00,  1.68s/epoch, Epoch=50, Loss per epoc
h=0.00451]
 --- Hyperparameter configuration ---

Parameters: batch_size=2000, hidden_layers=[500, 250], eta=0.001, epochs=50
Test Accuracy: 99.01%

Training: 100%|██████████| 100/100 [03:24<00:00,  2.04s/epoch, Epoch=100, Loss per e
poch=0.00354]
 --- Hyperparameter configuration ---

Parameters: batch_size=2000, hidden_layers=[500, 250], eta=0.001, epochs=100
Test Accuracy: 99.16%

Best Model Summary:
Best Accuracy: 99.65%
Best Hyperparameters: {'batch_size': 1000, 'hidden_layers': [500, 250], 'eta': 0.01,
'epochs': 50}
```

In [18]:
```python
best_model = train_NNet(train_X_1_5, train_class_1_5, epochs= 50, eta = 0.01, batch
acc,Y_pred = test_model(test_X_1_5, test_y_1_5, best_model, 1, 5)
print(f"The Best Model Accuracy - {acc*100:.2f}%")
plot_confusion_matrix(test_y_1_5,Y_pred,['label 1', 'label 5'])
```

```
Training: 100%|██████████| 50/50 [01:21<00:00,  1.63s/epoch, Epoch=50, Loss per epoc
h=0.00329]
```



```
The Best Model Accuracy - 99.65%
```



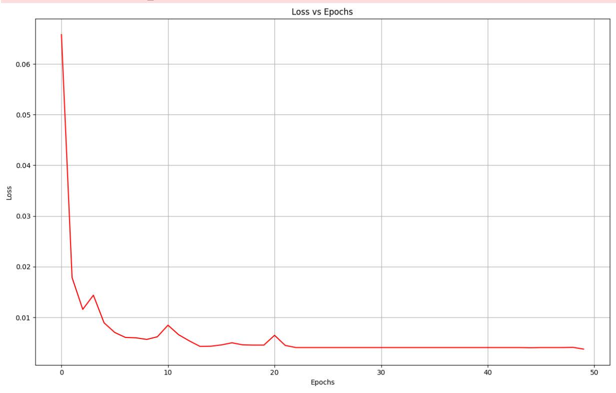
According to the above plots, its obvious that the set of best hyperparameters yield a better accuracy in predicting unseen data. It's achieving an accuracy of 99.65% since it has learnt the features well. In best hypeparameter configuration, number of hidden layers have been reduced to 2. Higher number of layers gives a lower accuracy since there's a chance of overfitting due to increasing model complexity. High no of epochs(50) help the model to learn well with time.

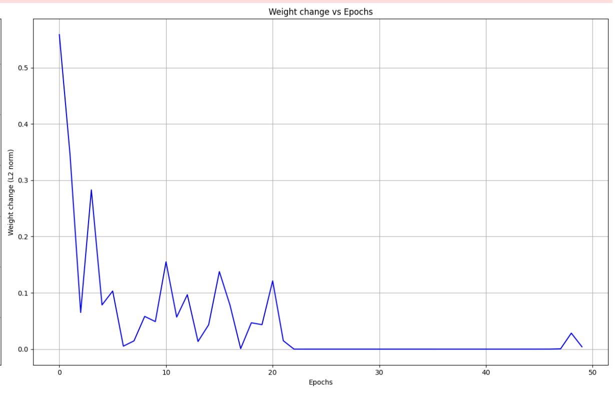
Training for other Digits

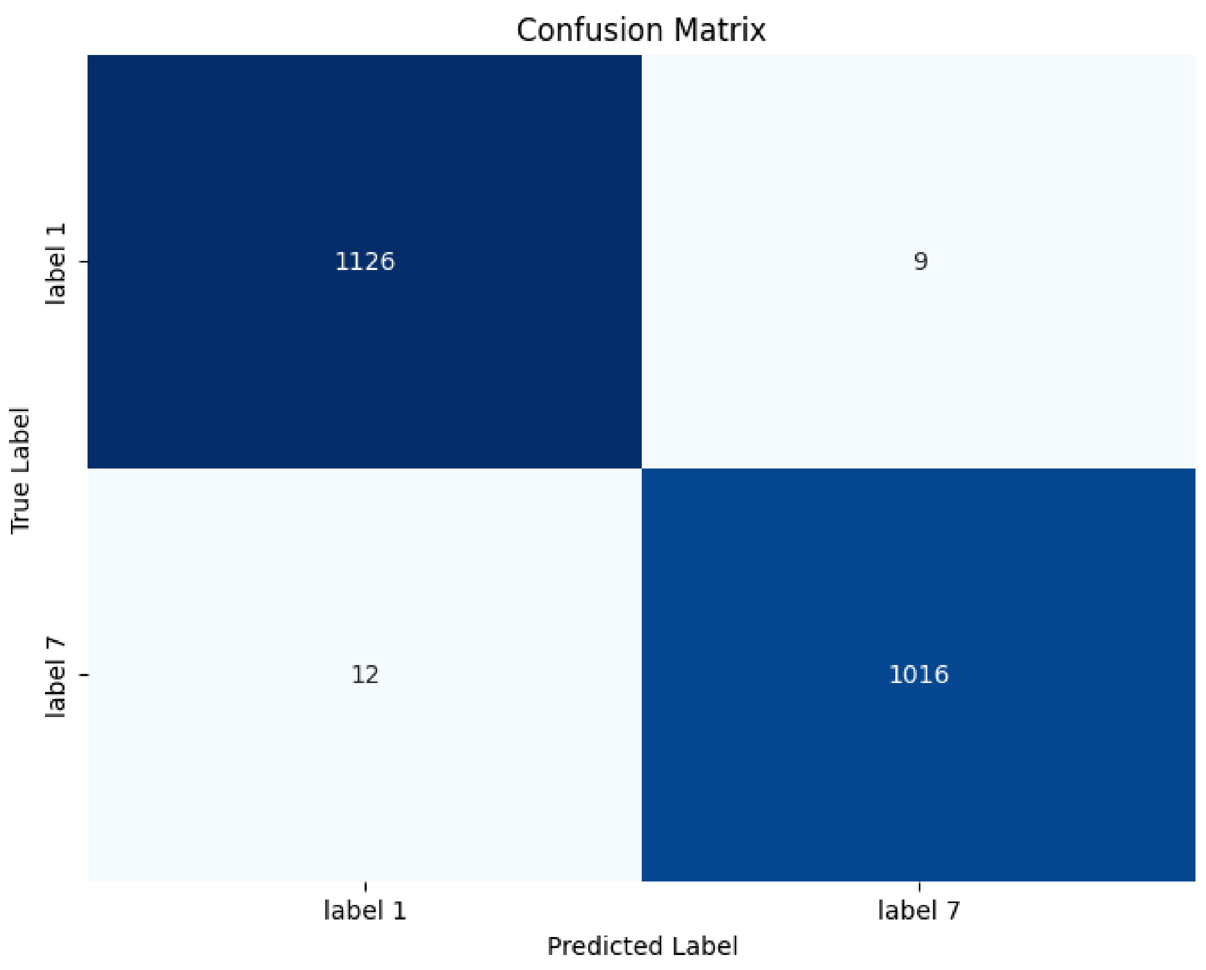
In [40]:
```
def run_experiments(digit_pairs, train_X, train_y, test_X, test_y, hyperparameters)
    for d1, d2 in digit_pairs:
        print(f'------------- Selected numbers - d1 = {d1}, d2 = {d2} -------------

        train_X_extracted, train_y_extracted =  extract_digits(train_X, train_y, d1
        test_X_extracted, test_y_extracted =  extract_digits(test_X, test_y, d1=d1,

        train_X_vectorized, train_y_labels = vectorize_images(train_X_extracted), t
        test_X_vectorized, test_y_labels = vectorize_images(test_X_extracted), test

        train_class = np.where(train_y_labels > d1,1,0)
        test_class = np.where(test_y_labels > d1,1,0)


        #train the model
        best_model1 = train_NNet(train_X_vectorized, train_class, epochs= hyperpara

        #test the model
        accuracy, y_pred = test_model(test_X_vectorized, test_y_labels, best_model1

        # Plot the confusion matrix
        plot_confusion_matrix(test_y_labels, y_pred, [f'label {d1}', f'label {d2}']


        # Print the model accuracy
        print(f"The Model Accuracy - {accuracy * 100:.2f}%")
        print('-' * 100)
        print('\n')
```

In [42]:
```
digit_pairs = [(1, 7), (4, 7), (2, 3),(4, 9)]
run_experiments(digit_pairs, train_X, train_y, test_X, test_y, best_params)
```

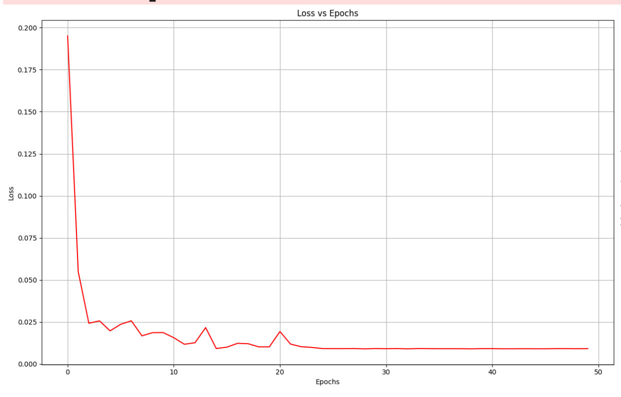
```
------------- Selected numbers - d1 = 1, d2 = 7 ----------------------
Training: 100%|██████████| 50/50 [01:28<00:00,  1.77s/epoch, Epoch=50, Loss per epoc
h=0.00373]
```

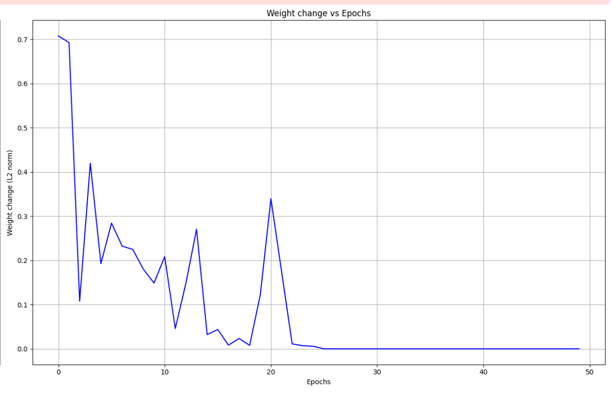


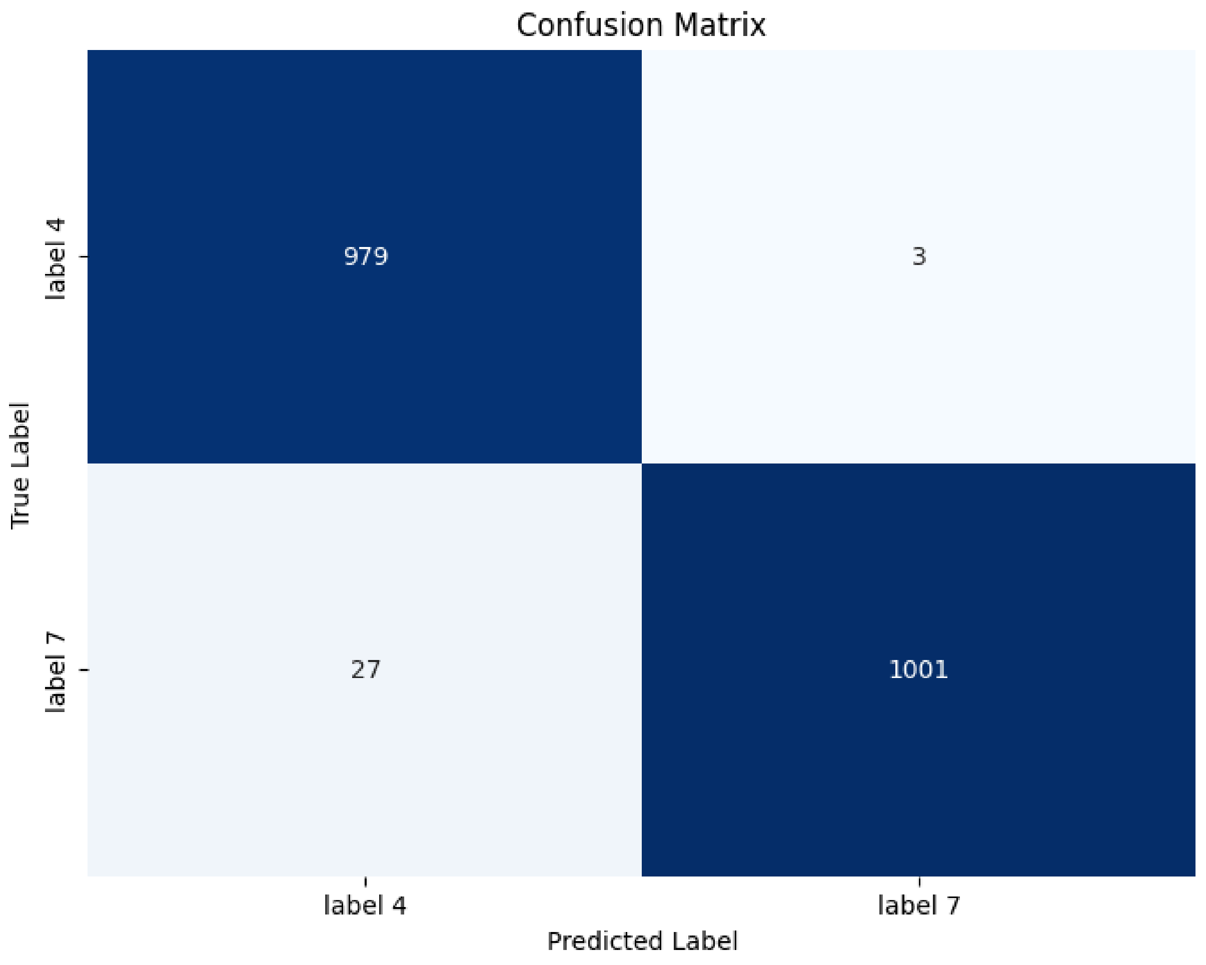




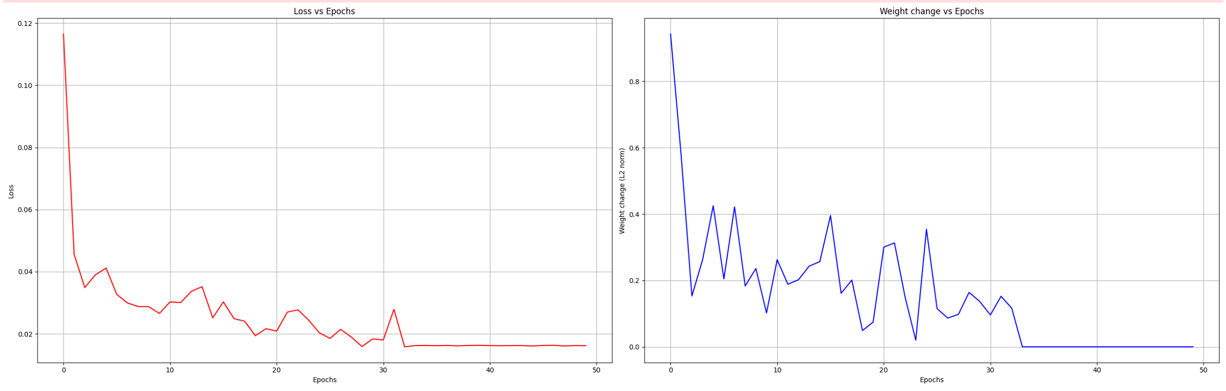
```
The Model Accuracy - 99.03%
----------------------------------------------------------------------------------------------


------------- Selected numbers - d1 = 4, d2 = 7 ----------------------
Training: 100%|██████████| 50/50 [01:26<00:00,  1.73s/epoch, Epoch=50, Loss per epoch=0.00917]
```

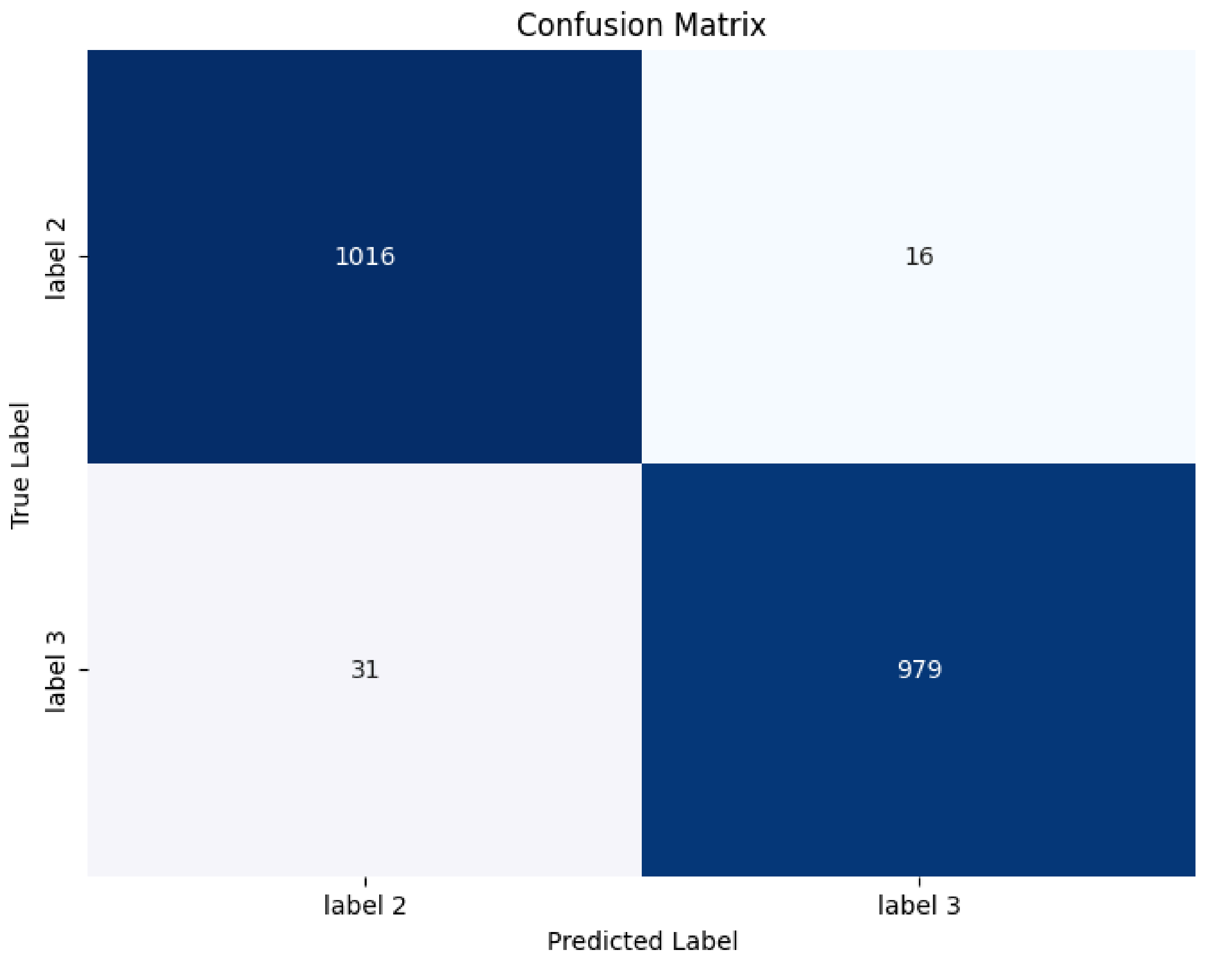






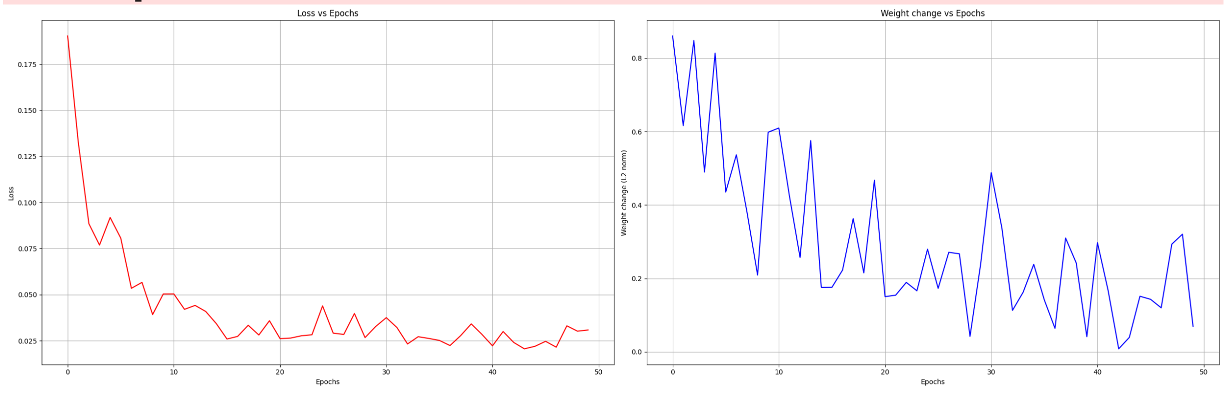
```
The Model Accuracy - 98.51%
--------------------------------------------------------------------------------------------------


------------- Selected numbers - d1 = 2, d2 = 3 ----------------------
Training: 100%|██████████| 50/50 [01:16<00:00,  1.53s/epoch, Epoch=50, Loss per epoch=0.0162]
```

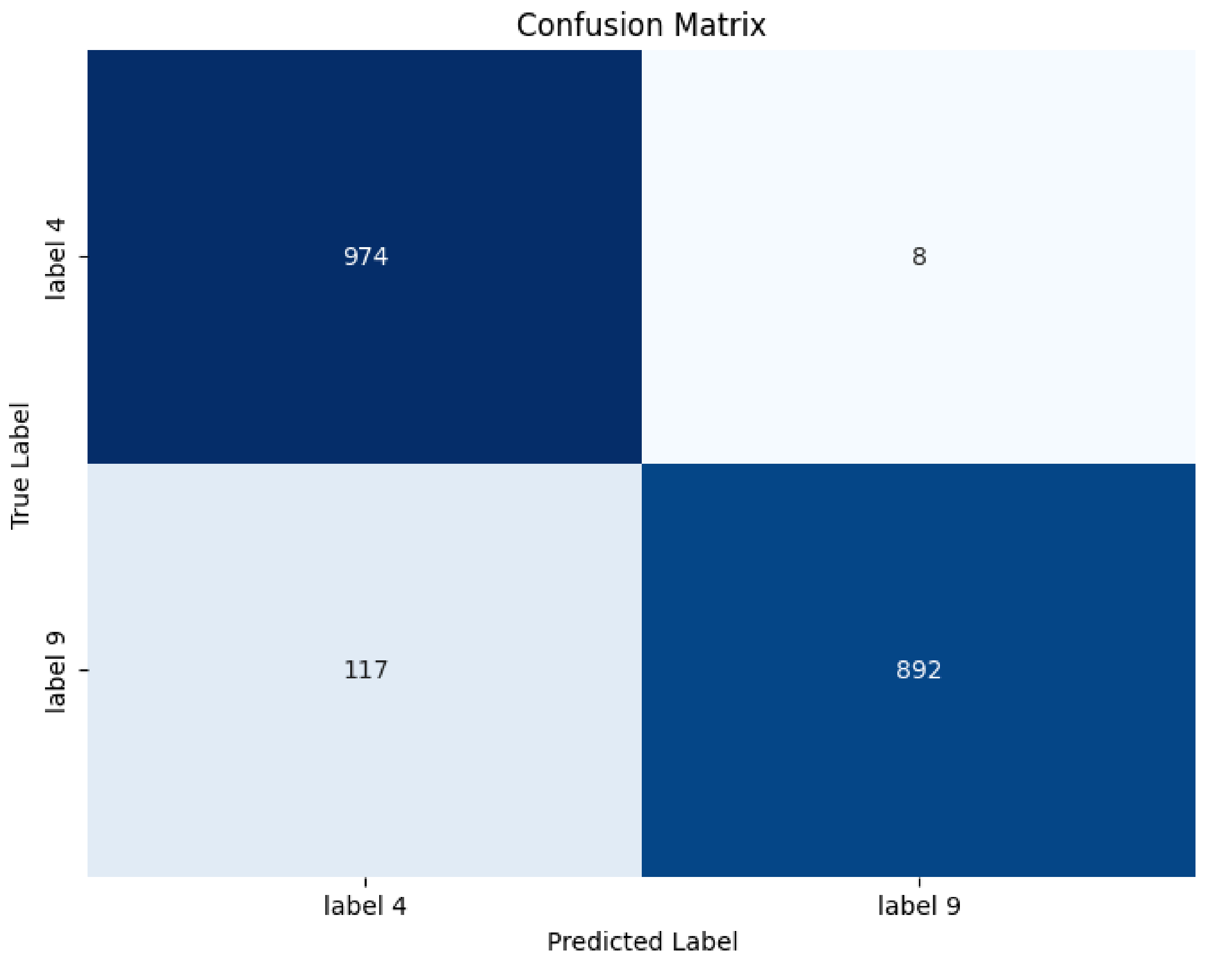




```
The Model Accuracy - 97.70%
--------------------------------------------------------------------------------
---------------
```

```
------------- Selected numbers - d1 = 4, d2 = 9 ----------------------
Training: 100%|██████████| 50/50 [01:16<00:00,  1.53s/epoch, Epoch=50, Loss per epoc
h=0.0308]
```





```
The Model Accuracy - 93.72%
------------------------------------------------------------------------------------
----------------
```

These digits receive good accuracies over 93%. Therefore we can conclude that the model is well trained and predicts test data well.