Sigurður Ágúst Jakobsson

Project: Assignment 5 - The Back-propagation Algorithm

Section 2.3

The output from one round of training and testing can be seen below. Since the data splitter is random if I recall correctly, and the weight initialization as well the results will be somewhat different for each run. The accuracy is generally in the 90th percentile on the training set after training. The test accuracy is generally a little lower, but not always. Weight initialization is quite important as can be seen by some of the test cases given in the project along with ones provided on Piazza. It seems like there are several local minima that can be reached in the error function, that do not perform the same.

Train Accuracy: 0.9333333333333333

Test Accuracy: 0.8620689655172413

Train Confusion Matrix (rows- actual, columns- guesses) :

[40 0 0]

[0 31 8]

[0 0 41]

Test Confusion Matrix (rows- actual, columns- guesses) :

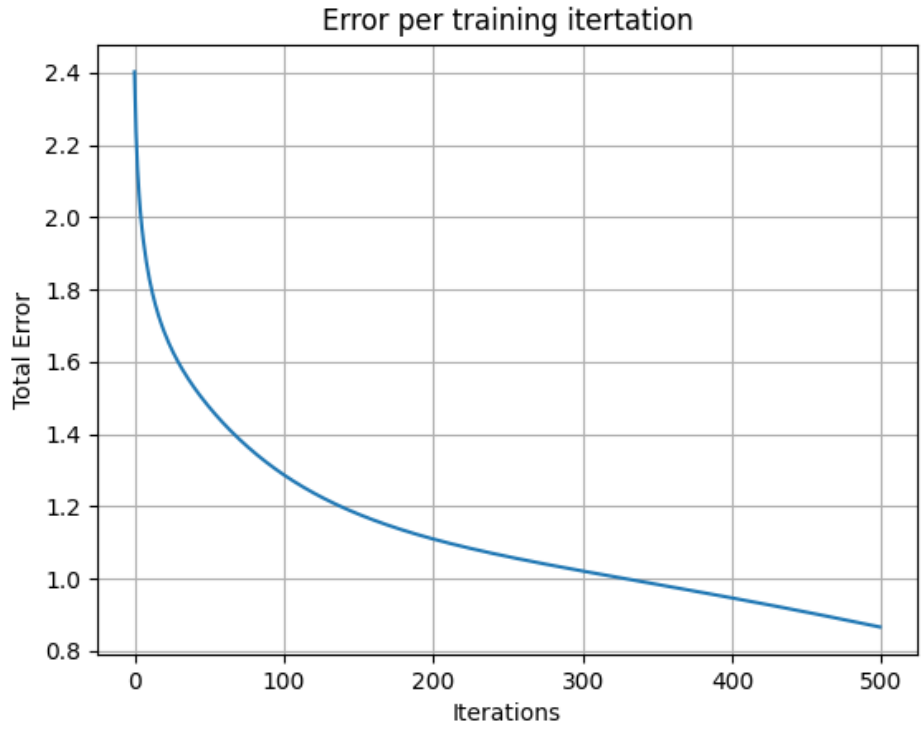
[9 0 0]

[0 7 4]

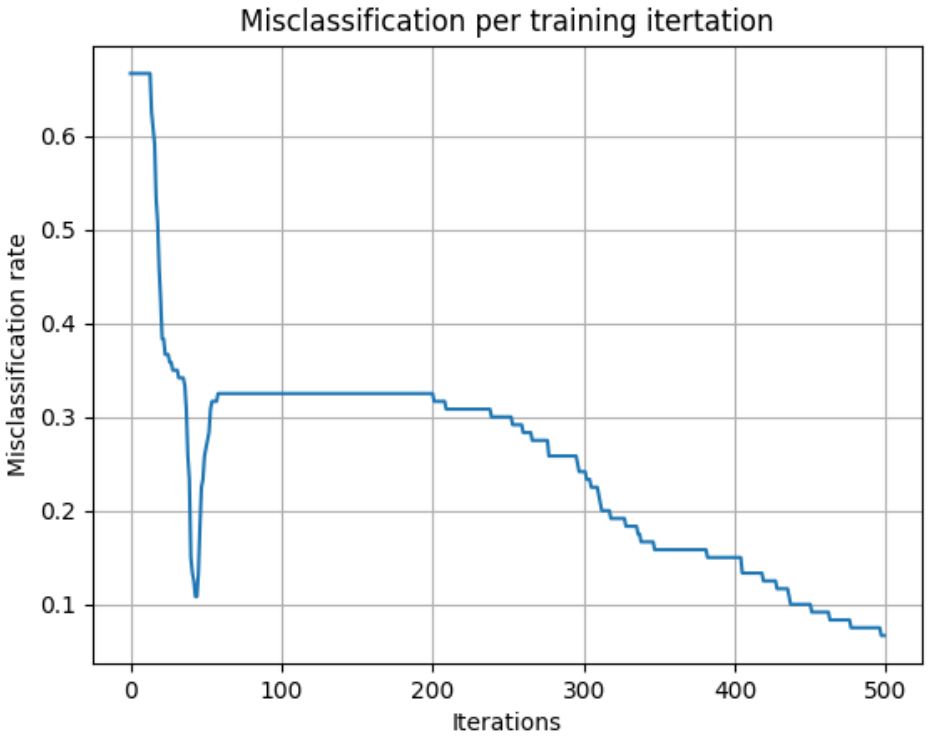
[0 0 9]

It takes some time to train the model (order of a few seconds) so I did not want to do to many runs of this and do a statistical analysis here. I did run another training with the same parameters in the independent section and the accuracies for that run were higher, again underlining the importance of weight initialization.

Plot of total error:



Plot of misclassification rate:



Independent Section

My first thoughts were to do tests with weight initialization. I expect that a Xavier initialization would be suitable for sigmoid activation functions, but I didn't find the literature I looked up to be quite clear on how to set the inputs to have zero mean and a standard deviation of one.

I then thought it would be interesting to use different activation functions and rerun the analysis from section 2.3. This might affect the gradients (vanishing or exploding). I repurposed the functions in the project to use different activation functions according to a passed *mode* variable and renamed them with a *_mode* suffix. The code is in the solution file and shown at the bottom of this report. I used the sigmoid function as a benchmark and then compared it to a rectified linear function, a unit step function and a hyperbolic tangent function. I ran training on the same datasets with similar initial weights. The results can be seen below:

Sigmoid

Train Accuracy sigmoid : 0.975

Test Accuracy sigmoid : 1.0

Train Confusion Matrix (rows- actual, columns- guesses): sigmoid

[41 0 0]

[0 40 2]

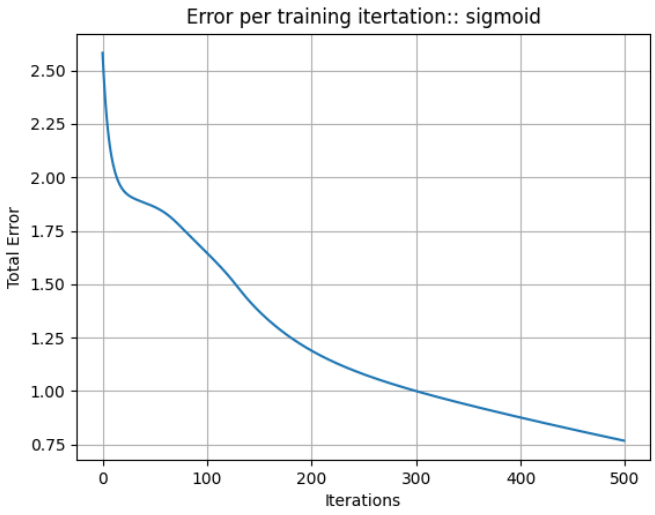
[0 1 36]

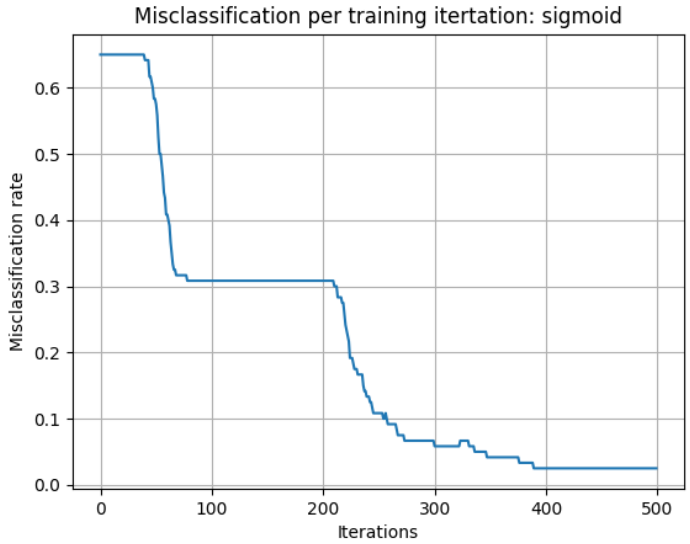
Test Confusion Matrix (rows- actual, columns- guesses): sigmoid

[9 0 0]

[0 8 0]

[0 0 12]





Rectified Linear function

Train Accuracy relu : 0.35

Test Accuracy relu : 0.27586206896551724

Train Confusion Matrix (rows- actual, columns- guesses): relu

[0 41 0]

[0 42 0]

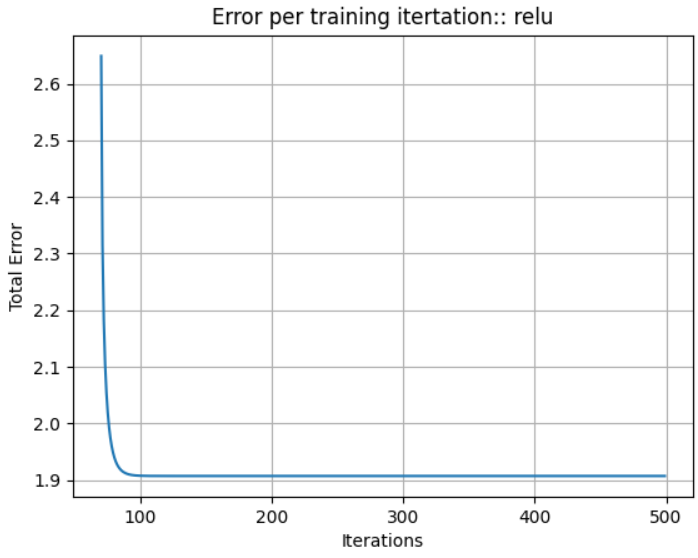
[0 37 0]

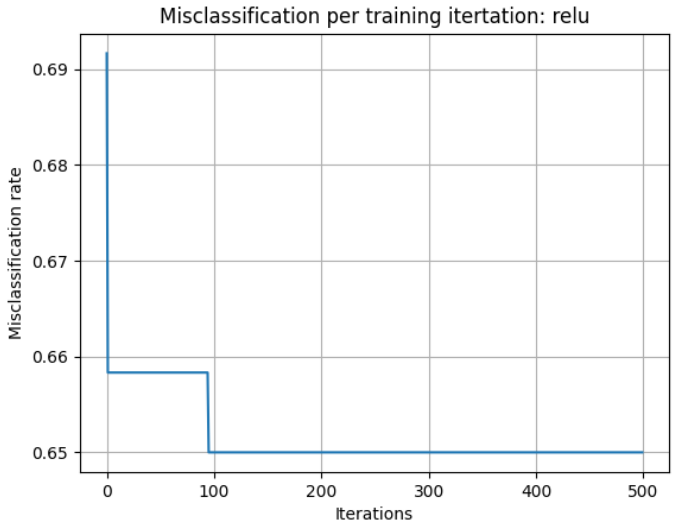
Test Confusion Matrix (rows- actual, columns- guesses): relu

[0 9 0]

[0 8 0]

[0 12 0]





Unit step function

Train Accuracy US : 0.35

Test Accuracy US : 0.3103448275862069

Train Confusion Matrix (rows- actual, columns- guesses): US

[0 41 0]

[0 42 0]

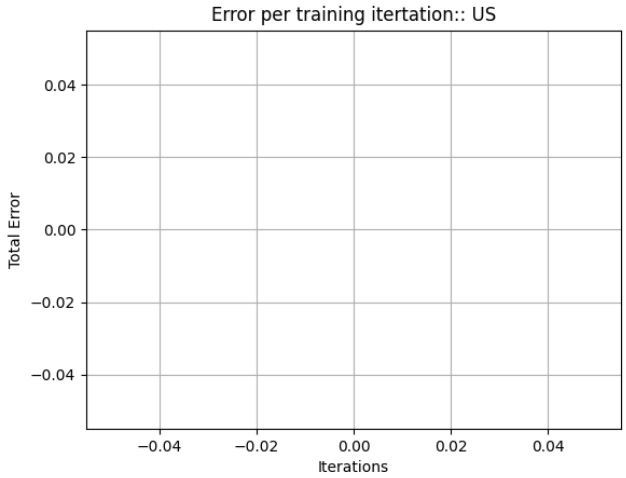
[0 37 0]

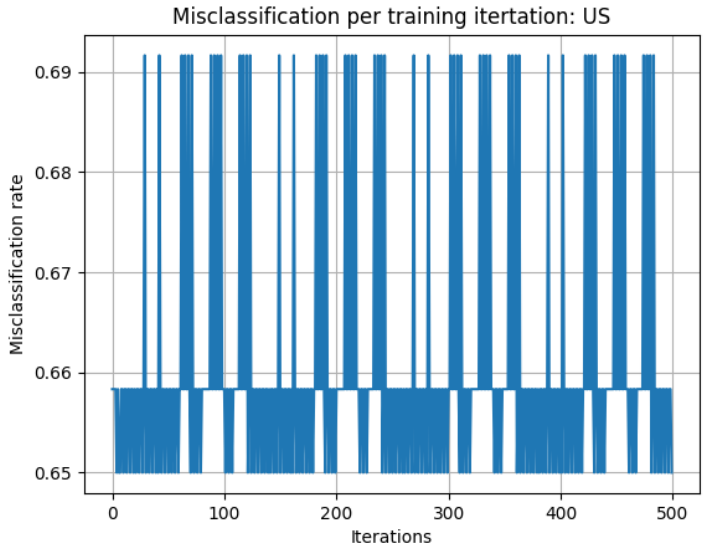
Test Confusion Matrix (rows- actual, columns- guesses): US

[9 0 0]

[8 0 0]

[12 0 0]





Hyperbolic tangent function

Train Accuracy tanh : 0.9416666666666667

Test Accuracy tanh : 0.9310344827586207

Train Confusion Matrix (rows- actual, columns- guesses): tanh

[41 0 0]

[0 35 7]

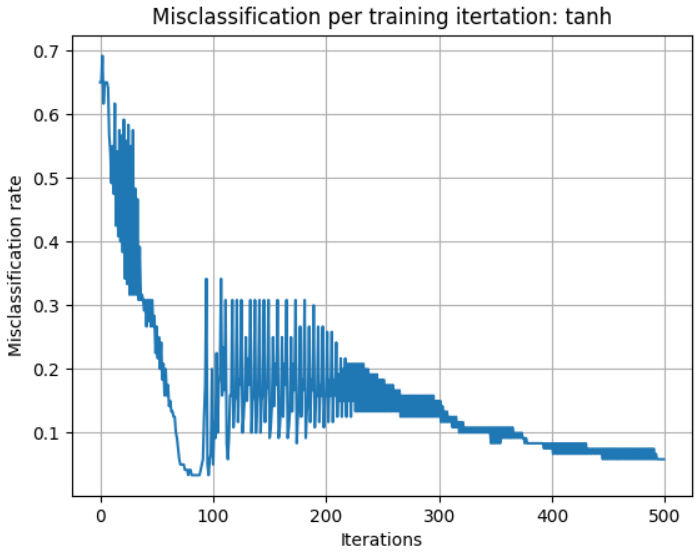
[0 0 37]

Test Confusion Matrix (rows- actual, columns- guesses): tanh

[9 0 0]

[0 8 0]

[0 2 10]



First analysis shows that the sigmoid functions and hyperbolic tangent functions returned relatively similar results. The hyperbolic tangent functions perform slightly worse but have a similar trend in misclassification after x iterations, even though the path I much more oscillatory.

The unit step and ReLU activations did not perform well however. At the end they predicted the same class for all inputs. This is clearly wrong and depending on the balance of the datasets this can be worse than a guess.

This leads to analysis on why the models behave as they do, and there are a few observations here.

In running the ReLU training I received the following runtime warnings:

C:\Users\Siggi\Desktop\Skólinn\Önn 9\Gagnanám og vitvélar\Programming Assignments\T809DATA_2022\05_backprop\solution05.py:466: RuntimeWarning: invalid value encountered in log

*error -= (target * np.log(y)).sum() + ((1 - target) * np.log(1 - y)).sum()*

C:\Users\Siggi\Desktop\Skólinn\Önn 9\Gagnanám og vitvélar\Programming Assignments\T809DATA_2022\05_backprop\solution05.py:466: RuntimeWarning: divide by zero encountered in log

*error -= (target * np.log(y)).sum() + ((1 - target) * np.log(1 - y)).sum()*

C:\Users\Siggi\Desktop\Skólinn\Önn 9\Gagnanám og vitvélar\Programming Assignments\T809DATA_2022\05_backprop\solution05.py:466: RuntimeWarning: invalid value encountered in multiply

*error -= (target * np.log(y)).sum() + ((1 - target) * np.log(1 - y)).sum()*

The error calculation relies on calculating the natural logarithm of the outputs. In the sigmoid function this is usually above 0 unless we encounter the edge case where x < -100. In ReLU, unit step and hyperbolic tangent methods we quite often get 0 outputs I assume when *a* is less than 0. We cant take the natural logarithm of <= 0 so this suggests that we need to calculate the error differently. This also suggests why the error plots on two of the runs print out empty.

This indicates that there might also be some other implications in the data or weight initialization that need to be taken into consideration before running this exact model with different activation functions. I tried to have different and positive weights in the ReLU and unit step models with the following code:

```
    W1 = 2 * np.random.rand(D + 1, M) - 1
    W2 = 2 * np.random.rand(M + 1, K) - 1

    for mode in modes:

        W1_use = W1
        W2_use = W2

        if mode in ['relu', 'US']:
            W1_use = (W1 + 1) / 2
            W2_use = (W2 + 1) / 2
```

From this introduction to neural networks I am not quite familiar with the assumptions needed to adjust the model properly or what data will work well in what context. There is also the possibility that there have been some trivial programming mistakes made.

While it would have been nice to show increase in performance by tweaking some network parameters, I have not managed to do this. I have made the network perform worse by changing the activation functions used, this does not necessarily indicate skill or insight. This might be a result in itself though. It suggests to me that what was proposed for this project fit the dataset well and may have been tailored to this specific problem. I also assume that these different functions may be beneficial at different layers of larger networks. At the very least this indicates the importance of using suitable activation functions for a given problem.

The complete updated code and original activation functions can be found below:

```
def sigmoid(x: float) -> float:
    '''
    Calculate the sigmoid of x
    '''
    if x < -100:
        sigmoid = 0.0
    else:
        sigmoid = 1 / (1 + np.exp(-x))

    return sigmoid


def d_sigmoid(x: float) -> float:
    '''
    Calculate the derivative of the sigmoid of x.
    '''
    return sigmoid(x) * (1-sigmoid(x))
def relu(x: float) -> float:
    '''
```

```
    Calculate the relu of x
    '''
    if x >= 0:
        relu = x
    else:
        relu = 0

    return relu


def d_relu(x: float) -> float:
    '''
    Calculate the derivative of the relu of x.
    '''
    if x >= 0:
        d_relu = 1
    else:
        d_relu = 0

    return d_relu

def unit_step(x: float) -> float:
    '''
    Calculate the unit step of x
    '''
    if x >= 0:
        us = 1
    else:
        us = 0

    return us


def d_unit_step(x: float) -> float:
    '''
    Calculate the derivative of the unit step of x.
    '''
    return 0

def tanh(x: float) -> float:
    '''
    Calculate the hyperbolic tangent of x
    '''
    #https://www.wolframalpha.com/input?i=tanh
    return (np.exp(2*x) - 1) / (np.exp(2*x) + 1)


def d_tanh(x: float) -> float:
    '''
```

```
    Calculate the derivative of the hyperbolic tangent of x.
    '''
    #https://www.wolframalpha.com/input?i=d%2Fdx+tanh(x)
    return 4 / (np.exp(-x) + np.exp(x))**2

def perceptron_mode(
    x: np.ndarray,
    w: np.ndarray,
    mode: str
) -> Union[float, float]:
    '''
    Return the weighted sum of x and w as well as
    the result of applying the sigmoid activation
    to the weighted sum
    '''
    a = w.T @ x

    if mode == 'sigmoid':
        z = sigmoid(a)
    elif mode == 'relu':
        z = relu(a)
    elif mode == 'US':
        z = unit_step(a)
    elif mode == 'tanh':
        z = tanh(a)
    else:
        #Default
        z = sigmoid(a)

    return a, z


def ffnn_mode(
    x: np.ndarray,
    M: int,
    K: int,
    W1: np.ndarray,
    W2: np.ndarray,
    mode: str
) -> Union[np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
    '''
    Computes the output and hidden layer variables for a
    single hidden layer feed-forward neural network.
    '''
    z0 = np.insert(x, 0, 1, axis=0)
    z1 = []
    a1 = []
    a2 = []
    y = []
```

```
    for neuron in range(M):
        a, z = perceptron_mode(z0, W1[:, neuron], mode)
        a1.append(a)
        z1.append(z)

    z1 = np.insert(z1, 0, 1, axis=0)

    for neuron in range(K):
        a, z = perceptron_mode(z1, W2[:, neuron], mode)
        a2.append(a)
        y.append(z)


    return np.array(y), np.array(z0), np.array(z1), np.array(a1),
np.array(a2)


def backprop_mode(
    x: np.ndarray,
    target_y: np.ndarray,
    M: int,
    K: int,
    W1: np.ndarray,
    W2: np.ndarray,
    mode: str
) -> Union[np.ndarray, np.ndarray, np.ndarray]:
    '''
    Perform the backpropagation on given weights W1 and W2
    for the given input pair x, target_y
    '''
    y, z0, z1, a1, a2 = ffnn_mode(x, M, K, W1, W2, mode)

    d_k = y - target_y
    d_j = np.zeros(a1.shape[0])

    #Try my best to follow the formula
    for j in range(len(d_j)):
        if mode == 'sigmoid':
            d_j[j] = d_sigmoid(a1[j]) * (W2[j+1, :] * d_k).sum()
        elif mode == 'relu':
            d_j[j] = d_relu(a1[j]) * (W2[j+1, :] * d_k).sum()
        elif mode == 'US':
            d_j[j] = d_unit_step(a1[j]) * (W2[j+1, :] * d_k).sum()
        elif mode == 'tanh':
            d_j[j] = d_tanh(a1[j]) * (W2[j+1, :] * d_k).sum()
        else:
            #Default
            d_j[j] = d_sigmoid(a1[j]) * (W2[j+1, :] * d_k).sum()
```

```
    dE1 = np.zeros((W1.shape[0], W1.shape[1]))
    dE2 = np.zeros((W2.shape[0], W2.shape[1]))

    #Relatively simple 2D loops to fill in matrices according to d_x*z_y

    #Start using K at output
    for k in range(len(d_k)):
        for j in range(len(z1)):
            dE2[j, k] = d_k[k] * z1[j]

    #Go to earlier layer
    for j in range(len(d_j)):
        for i in range(len(z0)):
            dE1[i, j] = d_j[j] * z0[i]

    return np.array(y), np.array(dE1), np.array(dE2)

def train_nn_mode(
    X_train: np.ndarray,
    t_train: np.ndarray,
    M: int,
    K: int,
    W1: np.ndarray,
    W2: np.ndarray,
    iterations: int,
    eta: float,
    mode: str
) -> Union[np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
    '''
    Train a network by:
    1. forward propagating an input feature through the network
    2. Calculate the error between the prediction the network
    made and the actual target
    3. Backpropagating the error through the network to adjust
    the weights.
    '''
    W1tr = np.zeros((W1.shape[0], W1.shape[1]))
    W2tr = np.zeros((W2.shape[0], W2.shape[1]))
    E_total = []
    misclassification_rate = []


    for it in range(iterations):
        dE1_total = np.zeros((W1.shape[0], W1.shape[1]))
        dE2_total = np.zeros((W2.shape[0], W2.shape[1]))
        guesses = []
        error = 0
```

```
        for point in range(X_train.shape[0]):
            #Put target in vector so it is handled right
            target = np.zeros(K)
            target[t_train[point]] = 1.0
            #Backpropogate
            y, dE1, dE2 = backprop_mode(X_train[point], target, M, K, W1, W2,
mode)
            guess_index = np.argmax(y)
            guesses.append(guess_index)
            dE1_total += dE1
            dE2_total += dE2
            error -= (target * np.log(y)).sum() + ((1 - target) * np.log(1 -
y)).sum()

        W1 = W1 - eta * dE1_total / X_train.shape[0]
        W2 = W2 - eta * dE2_total / X_train.shape[0]
        misclassification_rate.append((t_train != guesses).sum()
/  t_train.shape[0])
        E_total.append(error / X_train.shape[0])

    W1tr = W1
    W2tr = W2

    return np.array(W1tr), np.array(W2tr), np.array(E_total),
np.array(misclassification_rate), np.array(guesses)

def test_nn_mode(
    X: np.ndarray,
    M: int,
    K: int,
    W1: np.ndarray,
    W2: np.ndarray,
    mode: str
) -> np.ndarray:
    '''
    Return the predictions made by a network for all features
    in the test set X.
    '''
    guesses = np.zeros(X.shape[0],dtype=int)

    for feature in range(X.shape[0]):
       y, z0, z1, a1, a2 = ffnn_mode(X[feature], M, K, W1, W2, mode)
       guesses[feature] = np.argmax(y)

    return guesses

def train_test_plot_mode():
    modes = ['sigmoid', 'relu', 'US', 'tanh']
```

```
    features, targets, classes = tools.load_iris()
    (train_features, train_targets), (test_features, test_targets) =
tools.split_train_test(features, targets,train_ratio=0.8)

    K = len(classes)  # number of classes
    M = 6
    D = train_features.shape[1]

    W1 = 2 * np.random.rand(D + 1, M) - 1
    W2 = 2 * np.random.rand(M + 1, K) - 1

    for mode in modes:

        W1_use = W1
        W2_use = W2

        if mode in ['relu', 'US']:
            W1_use = (W1 + 1) / 2
            W2_use = (W2 + 1) / 2

        W1tr, W2tr, Etotal, misclassification_rate, last_guesses =
train_nn_mode(train_features, train_targets, M, K, W1_use, W2_use, 500, 0.1,
mode)
        guesses = test_nn_mode(test_features, M, K, W1tr, W2tr, mode)

        train_accuracy = accuracy(train_targets, last_guesses)
        test_accuracy = accuracy(test_targets, guesses)

        train_confusion = confusion_matrix(train_targets, last_guesses,
classes)
        test_confusion = confusion_matrix(test_targets, guesses, classes)

        print('Train Accuracy', mode, ':', train_accuracy)
        print('Test Accuracy', mode, ':', test_accuracy)
        print('Train Confusion Matrix:', mode, '\n', train_confusion)
        print('Test Confusion Matrix:', mode, '\n', test_confusion)

        fig = plt.figure()
        ax = fig.add_subplot()
        ax.plot(Etotal)
        plt.title('Error per training itertation:: ' + mode)
        plt.grid()
        ax.set_xlabel('Iterations')
        ax.set_ylabel('Total Error')
        plt.show()

        fig = plt.figure()
        ax = fig.add_subplot()
        ax.plot(misclassification_rate)
```

```
        plt.title('Misclassification per training itertation: ' + mode)
        plt.grid()
        ax.set_xlabel('Iterations')
        ax.set_ylabel('Misclassification rate')
        plt.show()
```