Experiments with MNIST Digits - Multi-Class Classifier and Autoencoder

Shivchander Sudalairaj
Intelligent Systems (EECS 6036)
University of Cincinnati

November 2020

Abstract

In machine learning, multiclass or multinomial classification is the problem of classifying instances into one of three or more classes. For this experiment, we are considering the MNIST Handwritten digits dataset which consists of 10 classes (0-9 numbers). We first build a multilayer classifier which will classify the 10 digits. Then we build an autoencoder which will regenerate the image from the latent representation of the image.

1 Multiclass Classifier

1.1 System Description

Number of hidden layers: 1
Number of hidden neurons: 200
layer dimensions: [784, 200, 10]
Hidden layer activation: ReLU
Output layer activation: Sigmoid
Learning rate: 0.01
Output thresholds: {1: $y > 0.75$; 0: $y < 0.25$}
Weight initialization: Random
Bias initialization: Zeros
Epochs: 150
Stop training: Error (1 - balanced acc) $< 1\%$

1.2 Results

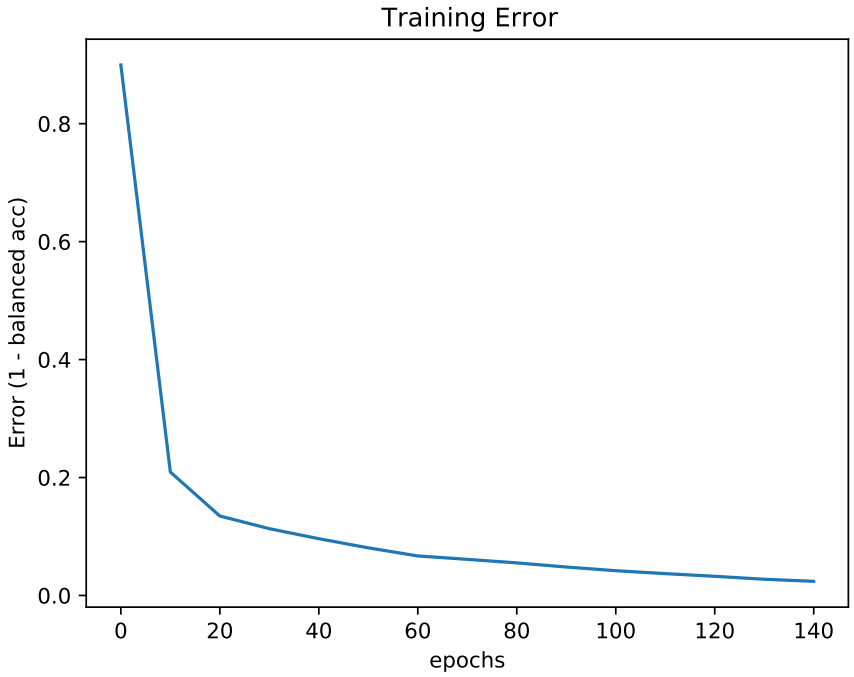


Figure 1.1: *Plot of Training error per epoch for the Dense Network with one hidden layer*

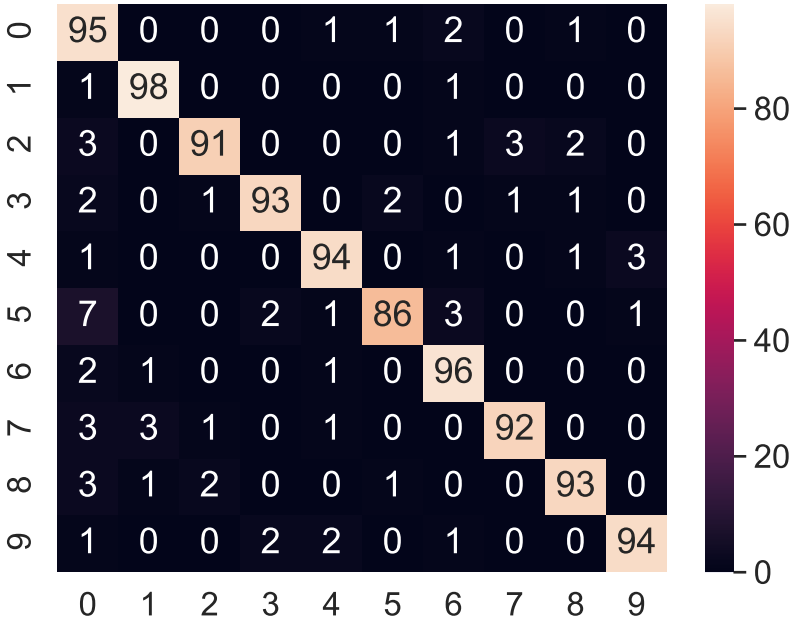


Figure 1.2: *Confusion Matrix for the digits in the test set*

1.3 Analysis of Results

The learning rate was determined after running a grid search on *lr = (0.1, 0.01, 0.001, 0.0001)* and based on the performance, *lr=0.01* was used. From Figure 1.1 it can be observed that Error rate keeps decreasing as the training progresses and eventually stays constant after 140^{th} epoch. The training was stopped after the error was less than 1%.

From Fig 1.2, the confusion matrix, we can observe that the classifier model performs the best in predicting digit 1s. The model correctly classified 1 98 out of 100. The reason why the model performs the best on this might be due to the shape of digit 1 which is just a line. While the classifier performs comparatively less better in terms of digits like 5 or other numbers which have curves and loops.

2 Autoencoder

2.1 System Description

Number of hidden layers: 1
Number of hidden neurons: 200
layer dimensions: [784, 200, 784]
Hidden layer activation: ReLU
Output layer activation: Sigmoid
Learning rate: 0.01
Output thresholds: {1: y $>$ 0.75; 0: y $<$ 0.25}
Weight initialization: Random
Bias initialization: Zeros
Epochs: 150

2.2 Results

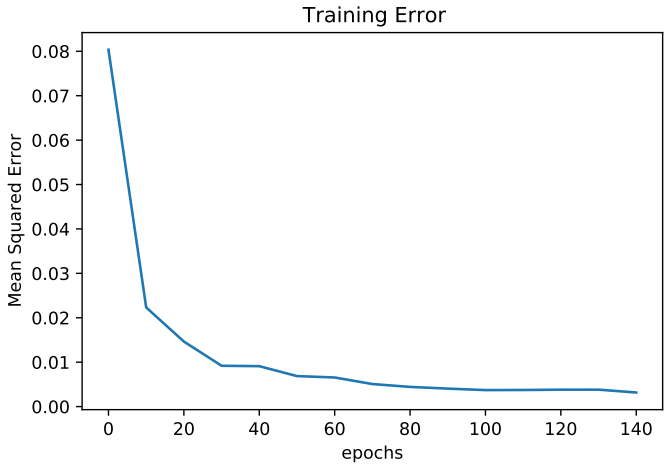


Figure 2.1: *Plot showing the Mean Squared Error on the training set per epoch*

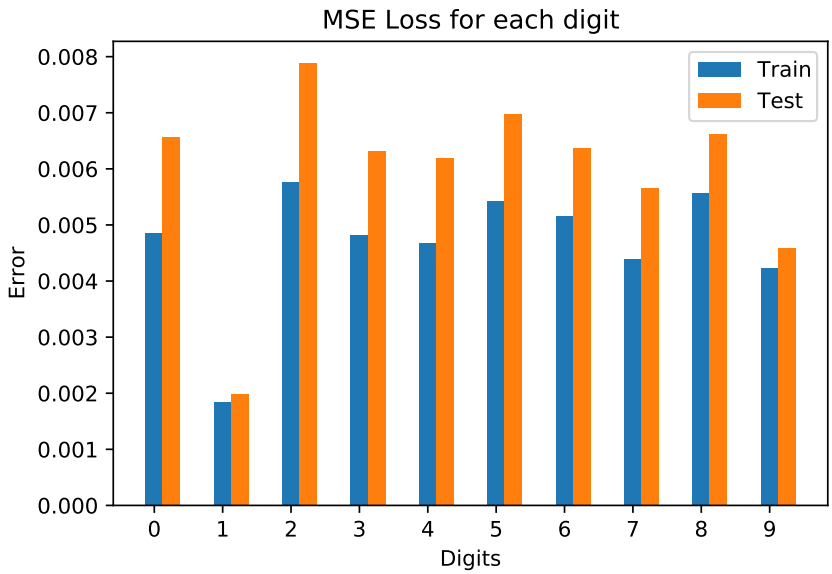
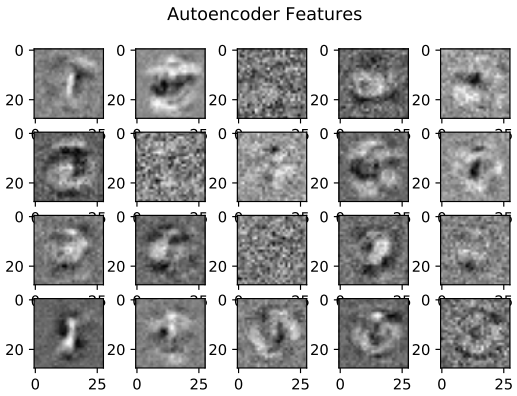


Figure 2.2: *Plot showing the Mean Squared Error on the training and test set per digit*

2.3 Features



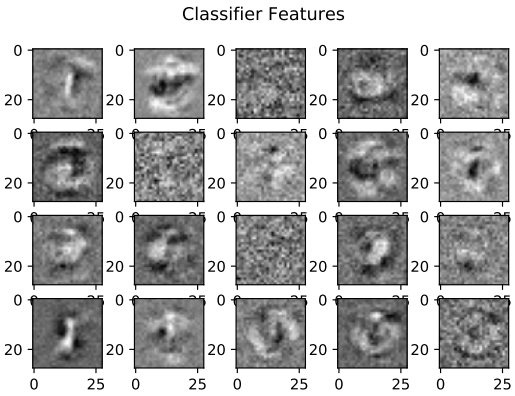


Figure 2.3: *Feature image captured by 20 random neurons from the hidder layer of the autoencoder*

Figure 2.4: *Feature image captured by 20 random neurons from the hidder layer of the classifier*

From figure 2.3 and 2.4, we can observe that both the network's hidden layers feature images are random pattern and can not be interpreted. This was interesting as one would expect the layers to pickup certain features from the digit image like loops or lines or edges. But this is not possible with a vanilla network or multilayer perceptron. Networks like CNN have latent spaces which can capture the loops and lines of the digits.

Since both the networks have the same number of hidden neurons(200) they capture the same latent space from the input neurons (784). The marginal difference can be atrributed to the random weight distribution.

2.4 Sample Outputs

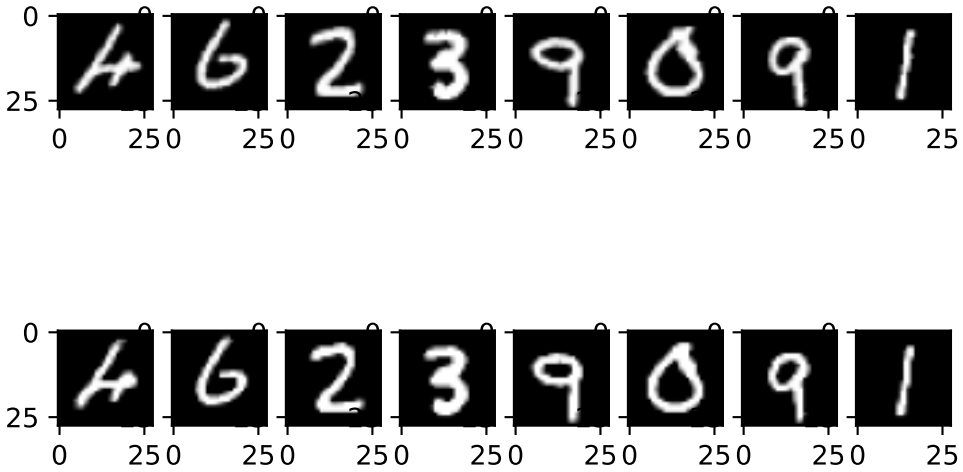


Figure 2.5: *Upper row is the true image and bottom is the decoded image from autoencoder*

2.5 Analysis of Results

The learning rate was determined after running a grid search on lr = (0.1, 0.01, 0.001, 0.0001)and based on the performance, lr=0.01 was used. From Figure 2.1 it can be observed that Error rate keeps decreasing as the training progresses and eventually stays constant after 140th epoch.

From Figure 2.2, we can observe that the model does really well on digit 1. This can be due to the fact that it is easier to detect the straight line from 1, rather than curves and loops from the other digits. The difference between the train and test errors for the other digits are close, except digit 2. This is because the digit 2 has the most variation in the handwriting amongst all other digits.

From Figure 2.3 and 2.4, we can observe that the feature images obtained from the weights of 20 randomly selected neurons from the classifier and the autoencoder, capture a close latent reprsentation. But this does not tell us anything as they are not interpretable. One would expect the neurons to capture lines or curves, but the vanilla multi layer perceptrons like these are not capable of capture those complex representations. Other networks like CNN are capable of capturing those.

From Figure 2.5 we can observe that the upper row of true images and the lower row of reconstructed images are close to the true image to the naked eye. This means that the latent representation of the image with just 200 neurons is good enough to reconstruct the original image.

3 Code Appendix

For futher reference and code implementation for this paper see : `https://github.com/shivchander/mnist-multilayer-feedforward`

3.1 main.py

```
#!/usr/bin/env python3
__author__ = "Shivchander Sudalairaj"
__license__ = "MIT"

'''
Multi Class Classification of MNIST Dataset using Multi Layer Feed Forward Neural Net implemente
'''

from utils import *
from model import DenseNN, plot_confusion_matrix
from autoencoder import AutoencoderNN


if __name__ == '__main__':
    data = parse_data('dataset/MNISTnumImages5000_balanced.txt', 'dataset/MNISTnumLabels5000_bal
    split_data(data)
    train = pd.read_csv('dataset/MNIST_Train.csv', sep=",")
    test = pd.read_csv('dataset/MNIST_Test.csv', sep=",")

    # Classification
    X_train, y_train, X_test, y_test = get_train_test('dataset/MNIST_Train.csv', 'dataset/MNIST_
    model = DenseNN()
    model.fit(X_train, y_train, 784, 200, 1, 10, learning_rate=0.01, batch_size=32,
              num_epochs=150, plot_error=True)
    y_preds = model.predict(X_test)
    print('Accuracy: ', balanced_accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_preds, ax
    plot_confusion_matrix(y_test, y_preds)

    # Autoencoder
    model2 = AutoencoderNN()
    model2.fit(X_train, X_train, 784, 200, 1, 784, learning_rate=0.01, batch_size=32, num_epochs
    random_outputs(model2, X_test)
    plot_train_test_error(model2, X_train, X_test)
    train_test_digit_error(model2, X_train, X_test)
    compare_features(model.parameters['W1'], model2.parameters['W1'])
```

3.2 model.py

```
#!/usr/bin/env python3
__author__ = "Shivchander Sudalairaj"
__license__ = "MIT"
```

```
'''
Model Definition: Multi Layer DenseNN
'''

import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt

np.random.seed(1)


class DenseNN(object):
    def __init__(self):
        self.n_x = 784
        self.n_h = 100
        self.n_l = 3
        self.n_y = 10
        self.layer_dims = []
        self.parameters = {}
        self.X = None
        self.y = None

    def initialize_parameters(self, n_x, n_h, n_l, n_y):
        self.n_x = n_x
        self.n_h = n_h
        self.n_l = n_l
        self.n_y = n_y
        self.layer_dims = [n_x] + [n_h] * n_l + [n_y]

        for l in range(1, len(self.layer_dims)):
            self.parameters['W' + str(l)] = np.random.randn(self.layer_dims[l], self.layer_dims[
            self.parameters['b' + str(l)] = np.zeros((self.layer_dims[l], 1))

            assert (self.parameters['W' + str(l)].shape == (self.layer_dims[l], self.layer_dims[
            assert (self.parameters['b' + str(l)].shape == (self.layer_dims[l], 1))

        return self.parameters

    def sigmoid(self, Z):
        A = 1 / (1 + np.exp(-Z))
        cache = Z

        return A, cache

    def relu(self, Z):
        A = np.maximum(0, Z)

        cache = Z
```

```
    return A, cache

def activation_forward(self, A_prev, W, b, activation):

    def linear_forward(A, W, b):
        Z = np.dot(W, A) + b
        cache = (A, W, b)
        return Z, cache

    if activation == "sigmoid":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = self.sigmoid(Z)

    elif activation == "relu":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = self.relu(Z)

    cache = (linear_cache, activation_cache)

    return A, cache

def forward_propagation(self, X, parameters):
    caches = []
    A = X
    L = len(parameters) // 2

    for l in range(1, L):
        A_prev = A
        A, cache = self.activation_forward(A_prev, parameters['W' + str(l)],
                                           parameters['b' + str(l)], activation='relu')
        caches.append(cache)

    AL, cache = self.activation_forward(A, parameters['W' + str(L)],
                                        parameters['b' + str(L)], activation='sigmoid')
    caches.append(cache)
    return AL, caches

def compute_cost(self, AL, Y):
    from sklearn.metrics import log_loss
    m = Y.shape[1]
    cost = 0
    for yt, yp in zip(Y.T, AL.T):
        cost += log_loss(yt, yp)
    return cost / m

def linear_backward(self, dZ, cache):

    A_prev, W, b = cache
    m = A_prev.shape[1]
```

```
    dW = np.dot(dZ, cache[0].T) / m
    db = np.squeeze(np.sum(dZ, axis=1, keepdims=True)) / m
    dA_prev = np.dot(cache[1].T, dZ)

    return dA_prev, dW, db

def relu_backward(self, dA, cache):
    Z = cache
    dZ = np.array(dA, copy=True)
    dZ[Z <= 0] = 0

    return dZ

def sigmoid_backward(self, dA, cache):
    Z = cache
    s = 1 / (1 + np.exp(-Z))
    dZ = dA * s * (1 - s)

    return dZ

def linear_activation_backward(self, dA, cache, activation):
    linear_cache, activation_cache = cache

    if activation == "relu":
        dZ = self.relu_backward(dA, activation_cache)
        dA_prev, dW, db = self.linear_backward(dZ, linear_cache)
        db = db.reshape(len(db), 1)

    elif activation == "sigmoid":
        dZ = self.sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = self.linear_backward(dZ, linear_cache)
        db = db.reshape(len(db), 1)

    return dA_prev, dW, db

def backward_propagation(self, AL, Y, caches):
    grads = {}
    L = len(caches)
    m = AL.shape[1]
    Y = Y.reshape(AL.shape)

    dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))
    current_cache = caches[L - 1]
    grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)] = self.linear_activatio



    for l in reversed(range(L - 1)):
```

```
            current_cache = caches[l]
            dA_prev_temp, dW_temp, db_temp = self.linear_activation_backward(grads["dA" + str(l
                                                                                "relu")
            grads["dA" + str(l + 1)] = dA_prev_temp
            grads["dW" + str(l + 1)] = dW_temp
            grads["db" + str(l + 1)] = db_temp

        return grads

    def initialize_velocity(self, parameters):
        L = len(parameters) // 2
        v = {}

        for l in range(L):
            v["dW" + str(l + 1)] = np.zeros_like(parameters["W" + str(l + 1)])
            v["db" + str(l + 1)] = np.zeros_like(parameters["b" + str(l + 1)])
        return v

    def update_parameters_with_momentum(self, parameters, grads, v, learning_rate):
        L = len(parameters) // 2
        beta = 0.9
        for l in range(L):
            # compute velocities
            v["dW" + str(l + 1)] = beta * v["dW" + str(l + 1)] + (1 - beta) * grads['dW' + str(l
            v["db" + str(l + 1)] = beta * v["db" + str(l + 1)] + (1 - beta) * grads['db' + str(l
            # update parameters
            parameters["W" + str(l + 1)] = parameters["W" + str(l + 1)] - learning_rate * v["dW"
            parameters["b" + str(l + 1)] = parameters["b" + str(l + 1)] - learning_rate * v["db"

        return parameters, v

    def random_mini_batches(self, X, Y, mini_batch_size=64, seed=0):

        m = X.shape[1]
        mini_batches = []

        permutation = list(np.random.permutation(m))
        shuffled_X = X[:, permutation]
        shuffled_Y = Y[:, permutation].reshape((10, m))

        num_complete_minibatches = math.floor(m / mini_batch_size)
        for k in range(0, num_complete_minibatches):
            mini_batch_X = shuffled_X[:, k * mini_batch_size:(k + 1) * mini_batch_size]
            mini_batch_Y = shuffled_Y[:, k * mini_batch_size:(k + 1) * mini_batch_size]
            mini_batch = (mini_batch_X, mini_batch_Y)
            mini_batches.append(mini_batch)

        if m % mini_batch_size != 0:
            end = m - mini_batch_size * math.floor(m / mini_batch_size)
```

```
            mini_batch_X = shuffled_X[:, num_complete_minibatches * mini_batch_size:]
            mini_batch_Y = shuffled_Y[:, num_complete_minibatches * mini_batch_size:]
            mini_batch = (mini_batch_X, mini_batch_Y)
            mini_batches.append(mini_batch)

        return mini_batches

    def fit(self, X, y, n_x, n_h, n_l, n_y, learning_rate=0.001, batch_size=64, num_epochs=1000,
        # parameters = L_layer_model(train_x, train_y, layers_dims, num_iterations=2500, print_c

        self.X = X.T
        self.y = y.T
        self.initialize_parameters(n_x, n_h, n_l, n_y)
        errors = []
        v = self.initialize_velocity(self.parameters)

        # Optimization loop
        for i in range(num_epochs):
            minibatches = self.random_mini_batches(self.X, self.y, batch_size)

            for minibatch in minibatches:
                (minibatch_X, minibatch_Y) = minibatch

                # Forward propagation
                al, caches = self.forward_propagation(minibatch_X, self.parameters)

                # Compute cost
                cost = self.compute_cost(al, minibatch_Y)

                # Backward propagation
                grads = self.backward_propagation(al, minibatch_Y, caches)

                # update parameters
                self.parameters, v = self.update_parameters_with_momentum(self.parameters, v, gr

            # Print the cost every 10 epoch
            if plot_error and i % 10 == 0:
                from sklearn.metrics import balanced_accuracy_score
                y_preds = self.predict(self.X.T)
                balanced_acc = balanced_accuracy_score(np.argmax(self.y.T, axis=1), np.argmax(y_
                error = 1 - balanced_acc
                print("Error after epoch %i: %f" % (i, error))
                errors.append(error)

            if error <= 0.01:
                print('Error is less than 1%. Stopping Training')
                break

        if plot_error:
```

```
            plt.plot(list(range(0, len(errors) * 10, 10)), errors)
            plt.ylabel('Error (1 - balanced acc)')
            plt.xlabel('epochs')
            plt.title('Training Error')
            plt.savefig('figs/error.pdf')
            plt.clf()

        return self.parameters

    def threshold_function(self, y_preds):
        rows, cols = y_preds.shape
        for row in range(rows):
            for col in range(cols):
                if y_preds[row, col] >= 0.75:
                    y_preds[row, col] = 1
                if y_preds[row, col] <= 0.25:
                    y_preds[row, col] = 0

        return y_preds

    def predict(self, X):
        X = X.T
        # Forward propagation
        a, caches = self.forward_propagation(X, self.parameters)

        return self.threshold_function(a.T)


def plot_confusion_matrix(y_true, y_pred):
    from sklearn.metrics import confusion_matrix
    cm = confusion_matrix(np.argmax(y_true, axis=1), np.argmax(y_pred, axis=1))
    import seaborn as sns
    df_cm = pd.DataFrame(cm, range(10), range(10))
    sns.set(font_scale=1.4)
    sns.heatmap(df_cm, annot=True)
    plt.savefig('figs/cm.pdf')
```

3.3 autoencoder.py

```
__author__ = "Shivchander Sudalairaj"
__license__ = "MIT"

'''
Autoencoder of MNIST Dataset using Multi Layer Feed Forward Neural Net implemented from scratch
'''

import numpy as np
import math
import matplotlib.pyplot as plt
```

```
np.random.seed(1)


class AutoencoderNN(object):
    def __init__(self):
        self.n_x = 784
        self.n_h = 100
        self.n_l = 1
        self.n_y = 784
        self.layer_dims = []
        self.parameters = {}
        self.X = None
        self.y = None

    def initialize_parameters(self, n_x, n_h, n_l, n_y):
        self.n_x = n_x
        self.n_h = n_h
        self.n_l = n_l
        self.n_y = n_y
        self.layer_dims = [n_x] + [n_h] * n_l + [n_y]

        for l in range(1, len(self.layer_dims)):
            self.parameters['W' + str(l)] = np.random.randn(self.layer_dims[l], self.layer_dims[
            self.parameters['b' + str(l)] = np.zeros((self.layer_dims[l], 1))

        return self.parameters

    def sigmoid(self, Z):
        A = 1 / (1 + np.exp(-Z))
        cache = Z

        return A, cache

    def relu(self, Z):
        A = np.maximum(0, Z)

        cache = Z
        return A, cache

    def activation_forward(self, A_prev, W, b, activation):

        def linear_forward(A, W, b):
            Z = np.dot(W, A) + b
            cache = (A, W, b)
            return Z, cache

        if activation == "sigmoid":
            Z, linear_cache = linear_forward(A_prev, W, b)
```

```
        A, activation_cache = self.sigmoid(Z)

    elif activation == "relu":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = self.relu(Z)

    cache = (linear_cache, activation_cache)

    return A, cache

def forward_propagation(self, X, parameters):
    caches = []
    A = X
    L = len(parameters) // 2

    for l in range(1, L):
        A_prev = A
        A, cache = self.activation_forward(A_prev, parameters['W' + str(l)],
                                           parameters['b' + str(l)], activation='relu')
        caches.append(cache)

    AL, cache = self.activation_forward(A, parameters['W' + str(L)],
                                        parameters['b' + str(L)], activation='sigmoid')
    caches.append(cache)
    return AL, caches

def compute_cost(self, AL, Y):
    m = Y.shape[1]
    cost = 0
    for yt, yp in zip(Y.T, AL.T):
        cost += np.square(yt-yp).mean()
    return cost / m

def linear_backward(self, dZ, cache):

    A_prev, W, b = cache
    m = A_prev.shape[1]

    dW = np.dot(dZ, cache[0].T) / m
    db = np.squeeze(np.sum(dZ, axis=1, keepdims=True)) / m
    dA_prev = np.dot(cache[1].T, dZ)

    return dA_prev, dW, db

def relu_backward(self, dA, cache):
    Z = cache
    dZ = np.array(dA, copy=True)
    dZ[Z <= 0] = 0
```

```
    return dZ

def sigmoid_backward(self, dA, cache):
    Z = cache
    s = 1 / (1 + np.exp(-Z))
    dZ = dA * s * (1 - s)

    return dZ

def linear_activation_backward(self, dA, cache, activation):
    linear_cache, activation_cache = cache

    if activation == "relu":
        dZ = self.relu_backward(dA, activation_cache)
        dA_prev, dW, db = self.linear_backward(dZ, linear_cache)
        db = db.reshape(len(db), 1)

    elif activation == "sigmoid":
        dZ = self.sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = self.linear_backward(dZ, linear_cache)
        db = db.reshape(len(db), 1)

    return dA_prev, dW, db

def backward_propagation(self, AL, Y, caches):
    grads = {}
    L = len(caches)
    m = AL.shape[1]
    Y = Y.reshape(AL.shape)

    # dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))
    dAL = 2*(AL - Y)
    current_cache = caches[L - 1]
    grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)] = self.linear_activatio



    for l in reversed(range(L - 1)):
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = self.linear_activation_backward(grads["dA" + str(l
                                                                                "relu")
        grads["dA" + str(l + 1)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads

def initialize_velocity(self, parameters):
    L = len(parameters) // 2
```

```
    v = {}

    for l in range(L):
        v["dW" + str(l + 1)] = np.zeros_like(parameters["W" + str(l + 1)])
        v["db" + str(l + 1)] = np.zeros_like(parameters["b" + str(l + 1)])
    return v

def update_parameters_with_momentum(self, parameters, grads, v, learning_rate):
    L = len(parameters) // 2
    beta = 0.9
    for l in range(L):
        # compute velocities
        v["dW" + str(l + 1)] = beta * v["dW" + str(l + 1)] + (1 - beta) * grads['dW' + str(l
        v["db" + str(l + 1)] = beta * v["db" + str(l + 1)] + (1 - beta) * grads['db' + str(l
        # update parameters
        parameters["W" + str(l + 1)] = parameters["W" + str(l + 1)] - learning_rate * v["dW"
        parameters["b" + str(l + 1)] = parameters["b" + str(l + 1)] - learning_rate * v["db"

    return parameters, v

def random_mini_batches(self, X, Y, mini_batch_size=64, seed=0):

    m = X.shape[1]
    mini_batches = []

    permutation = list(np.random.permutation(m))
    shuffled_X = X[:, permutation]
    shuffled_Y = Y[:, permutation]

    num_complete_minibatches = math.floor(m / mini_batch_size)
    for k in range(0, num_complete_minibatches):
        mini_batch_X = shuffled_X[:, k * mini_batch_size:(k + 1) * mini_batch_size]
        mini_batch_Y = shuffled_Y[:, k * mini_batch_size:(k + 1) * mini_batch_size]
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    if m % mini_batch_size != 0:
        end = m - mini_batch_size * math.floor(m / mini_batch_size)
        mini_batch_X = shuffled_X[:, num_complete_minibatches * mini_batch_size:]
        mini_batch_Y = shuffled_Y[:, num_complete_minibatches * mini_batch_size:]
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    return mini_batches

def fit(self, X, y, n_x, n_h, n_l, n_y, learning_rate=0.001, batch_size=64, num_epochs=1000,
    # parameters = L_layer_model(train_x, train_y, layers_dims, num_iterations=2500, print_c

    self.X = X.T
```

```
        self.y = y.T
        self.initialize_parameters(n_x, n_h, n_l, n_y)
        errors = []
        v = self.initialize_velocity(self.parameters)

        # Optimization loop
        for i in range(num_epochs):
            minibatches = self.random_mini_batches(self.X, self.y, batch_size)

            for minibatch in minibatches:
                (minibatch_X, minibatch_Y) = minibatch

                # Forward propagation
                al, caches = self.forward_propagation(minibatch_X, self.parameters)

                # Compute cost
                cost = self.compute_cost(al, minibatch_Y)

                # Backward propagation
                grads = self.backward_propagation(al, minibatch_Y, caches)

                # update parameters
                self.parameters, v = self.update_parameters_with_momentum(self.parameters, v, gr

            # Print the cost every 10 epoch
            if plot_error and i % 10 == 0:
                print("Error after epoch %i: %f" % (i, cost))
                errors.append(cost)

        if plot_error:
            plt.plot(list(range(0, len(errors) * 10, 10)), errors)
            plt.ylabel('Mean Squared Error')
            plt.xlabel('epochs')
            plt.title('Training Error')
            plt.savefig('figs/autoencoder_error.pdf')
            plt.clf()

        return self.parameters

    def threshold_function(self, y_preds):
        rows, cols = y_preds.shape
        for row in range(rows):
            for col in range(cols):
                if y_preds[row, col] >= 0.75:
                    y_preds[row, col] = 1
                if y_preds[row, col] <= 0.25:
                    y_preds[row, col] = 0

        return y_preds
```

```
    def predict(self, X):
        X = X.T
        # Forward propagation
        a, caches = self.forward_propagation(X, self.parameters)

        return self.threshold_function(a.T)
```

3.4 utils.py

```
__author__ = "Shivchander Sudalairaj"
__license__ = "MIT"

'''
Utility functions for the homework
'''

import pandas as pd
from sklearn.preprocessing import MultiLabelBinarizer
from sklearn.metrics import balanced_accuracy_score
import numpy as np
import matplotlib.pyplot as plt


def parse_data(feature_file, label_file):
    """
    :param feature_file: Tab delimited feature vector file
    :param label_file: class label
    :return: dataset as a pandas dataframe (features+label)
    """
    features = pd.read_csv(feature_file, sep="\t", header=None)
    labels = pd.read_csv(label_file, header=None)
    features['label'] = labels
    return features


def split_data(dataset):
    """
    Randomly choose 4,000 data points from the data files to form a training set, and use the re
    1,000 data points to form a test set. Make sure each digit has equal number of points in eac
    (i.e., the training set should have 400 0s, 400 1s, 400 2s, etc., and the test set should ha
    100 1s, 100 2s, etc.)
    :param dataset: pandas datafrome (features+label)
    :return: None. Saves Train and Test datasets as CSV
    """
    # init empty dfs
    train_df = pd.DataFrame()
    test_df = pd.DataFrame()
    for i in range(0, 10):
```

```
        df = dataset.loc[dataset['label'] == i]
        train_split = df.sample(frac=0.8, random_state=200)
        test_split = df.drop(train_split.index)
        train_df = pd.concat([train_df, train_split])
        test_df = pd.concat([test_df, test_split])

    train_df.to_csv('dataset/MNIST_Train.csv', sep=',', index=False)
    test_df.to_csv('dataset/MNIST_Test.csv', sep=',', index=False)


def get_train_test(train_file, test_file):
    train = pd.read_csv(train_file, sep=",")
    y_train = train['label'].values.reshape(4000, 1)
    mlb = MultiLabelBinarizer()
    y_train = mlb.fit_transform(y_train)
    X_train = train.iloc[:, :-1].values

    test = pd.read_csv(test_file, sep=",")
    y_test = test['label'].values.reshape(1000, 1)
    y_test = mlb.fit_transform(y_test)
    X_test = test.iloc[:, :-1].values

    return X_train, y_train, X_test, y_test


def train_test_digit_error(model, X_train, X_test):
    train_preds = model.predict(X_train)
    test_preds = model.predict(X_test)
    train_cost = {}
    test_cost = {}
    for i, e in enumerate(range(0, 4000, 400)):
        train_cost[i] = model.compute_cost(train_preds[e:e + 400, ].T, X_train[e:e + 400, ].T)

    for i, e in enumerate(range(0, 1000, 100)):
        test_cost[i] = model.compute_cost(test_preds[e:e + 100, ].T, X_test[e:e + 100, ].T)

    return train_cost, test_cost


def plot_train_test_error(model, X_train, X_test):
    train_errors, test_errors = train_test_digit_error(model, X_train, X_test)
    X = np.arange(10)
    plt.bar(X + 0.00, list(train_errors.values()), width = 0.25, label='Train')
    plt.bar(X + 0.25, list(test_errors.values()), width = 0.25, label='Test')
    plt.xticks(X)
    plt.title('MSE Loss for each digit')
    plt.xlabel('Digits')
    plt.ylabel('Error')
    plt.legend()
```

```
    plt.savefig('figs/digitwise_train_test_error.pdf')
    plt.clf()


def random_outputs(model, X_test):
    idx = np.random.randint(1000, size=8)
    sample_X = X_test[idx,:]
    sample_pred = model.predict(sample_X)

    fig, ax = plt.subplots(2, 8)
    for i in range(8):
        ax[0][i].imshow(sample_X[i].reshape(28,28).T, cmap = 'gray')
        ax[1][i].imshow(sample_pred[i].reshape(28,28).T, cmap = 'gray')
    plt.savefig('figs/random_output.pdf')


def compare_features(classifier_weights, autoencoder_weights):
    idx = np.random.randint(200, size=20)
    c_weights = classifier_weights[idx, :]
    a_weights = autoencoder_weights[idx, :]

    fig, ax = plt.subplots(4, 5)
    pos = 0
    for i in range(4):
        for j in range(5):
            x = (c_weights[pos] - np.min(c_weights[pos]))/np.ptp(c_weights[pos])
            ax[i][j].imshow(x.reshape(28,28).T, cmap='gray')
            pos+=1
    fig.suptitle('Classifier Features')
    fig.savefig('figs/classifier_features.pdf')
    fig.clf()

    fig, ax = plt.subplots(4, 5)
    pos = 0
    for i in range(4):
        for j in range(5):
            x = (a_weights[pos] - np.min(a_weights[pos]))/np.ptp(a_weights[pos])
            ax[i][j].imshow(x.reshape(28,28).T, cmap='gray')
            pos+=1
    fig.suptitle('Autoencoder Features')
    fig.savefig('figs/autoencoder_features.pdf')
    fig.clf()
```