Experiments with MNIST Digits - Denoising and Classifying Autoencoder

Shivchander Sudalairaj
Intelligent Systems (EECS 6036)
University of Cincinnati

November 2020

Abstract

Autoencoder refers to a set of architectures in neural networks which try to reconstruct the input given. By doing so, the network tries to learn an efficient encoding of the input features. Unlike other neural nets, the process is unsupervised. The network usually has two components, an encoder and a decoder. The first half of the network acts as the encoder which try to learn the latent representations of the input with fewest possible neurons. The second half of the network acts as the decoder and tries to reconstruct the original input from the latent representations learned from the encoder. The decoder tries to minimize the reconstruction error/noise and tries to generate the output to be as close to input as possible.

1 Denoising Autoencoder

1.1 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: 250
Stop training: MSE difference(for last 5 epochs) $< 1\%$
Input Noise: Masking Noise
f0: 0.4
f1: 0.05

1.2 Results

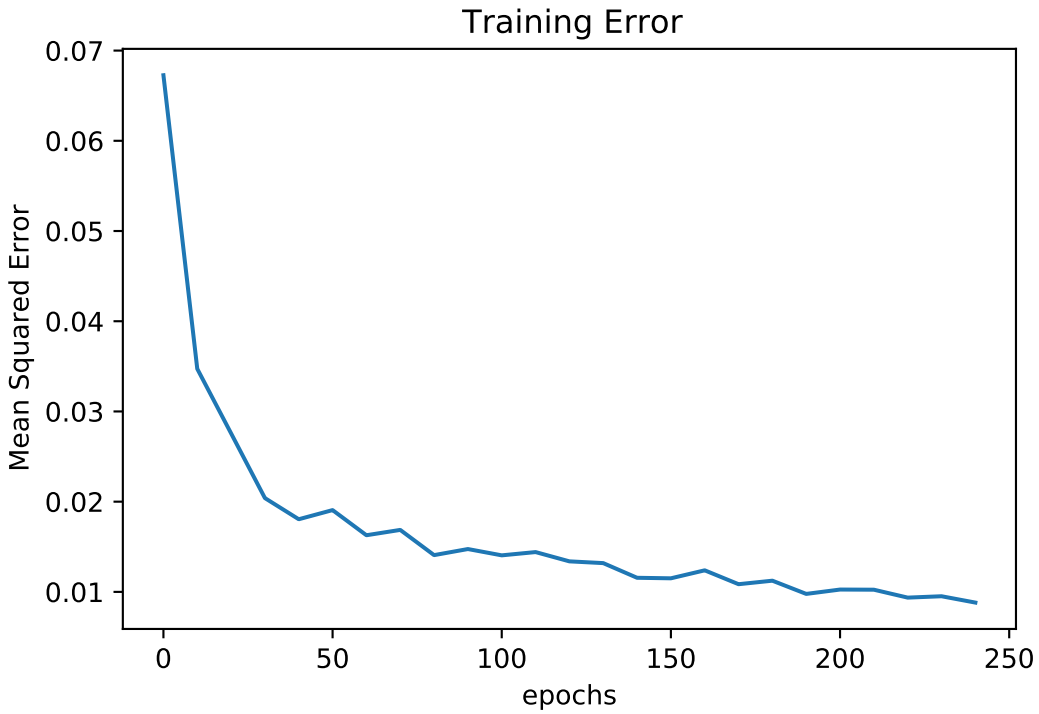
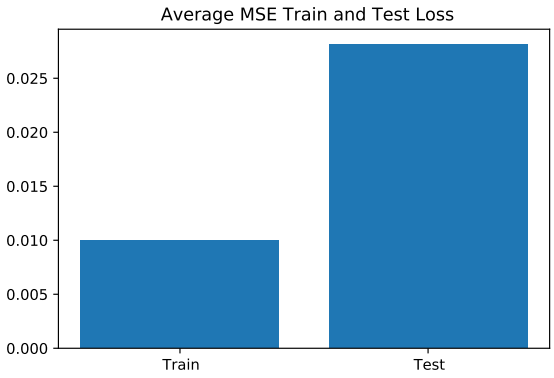


Figure 1.1: *Plot of Training error per epoch for the Denoising Autoencoder with one hidden layer*



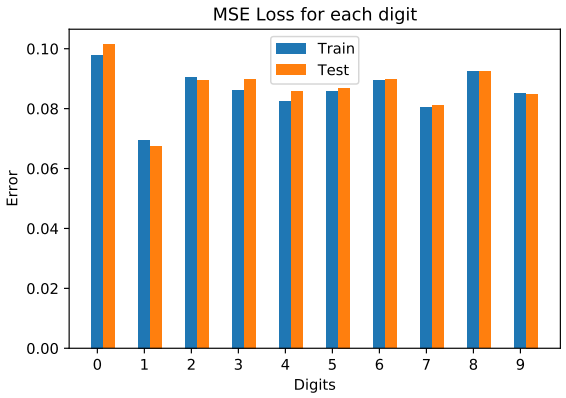


Figure 1.2: *Average Train Test Error across all digits*

Figure 1.3: *Train Test Error per digit*

1.3 Features

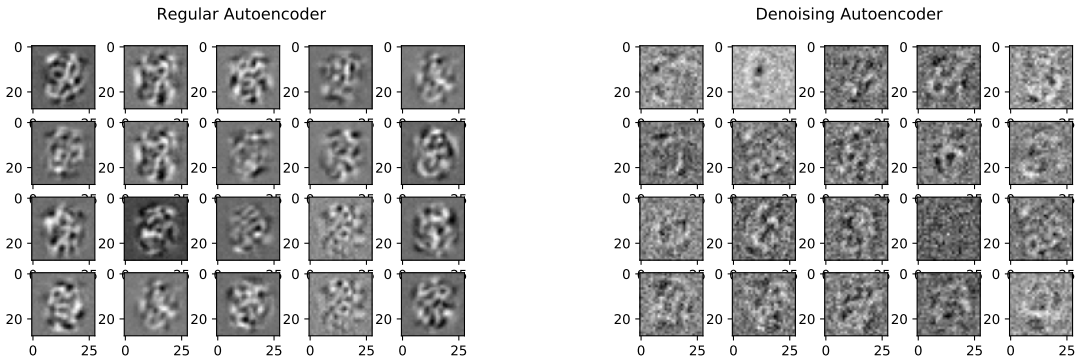


Figure 1.4: *Feature image captured by 20 random neurons from the hidder layer of the regular autoencoder*

Figure 1.5: *Feature image captured by 20 random neurons from the hidder layer of the denoising autoencoder*

From figure 1.4 and 1.5, 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.

But there is a difference between latent representations captured by the regular autoencoder when compared to the denoising autoencoder. The Weight maps of the Denoising Autoencoder has more noise than the regular autoencoder, this is due to the hard task performed by the Denoising Autoencoder

1.4 Sample Outputs

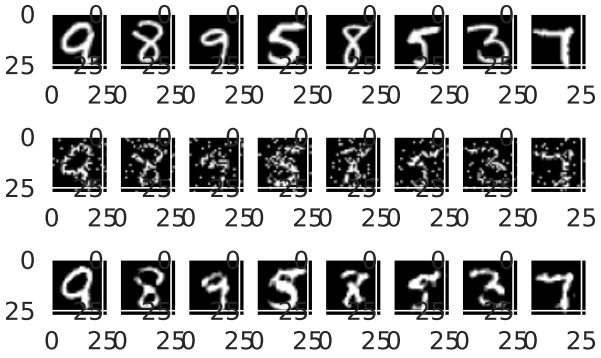


Figure 1.6: *Upper row is the true image, middle row is the mask noised image and bottom is the decoded image from autoencoder*

1.5 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 1.1 it can be observed that Error rate keeps decreasing as the training progresses and eventually stays constant after 220^{th} epoch. The training was stopped after the error was less than 1% for the last five epochs.

From Figure 1.3, 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 1.4 and 1.5, 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.

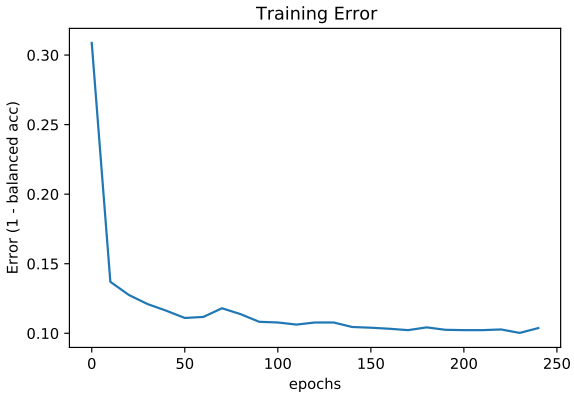
From Figure 1.6 we can observe that the upper row of true images and the lower row of reconstructed images are close to the true image to the naked eye. The middle row is the input image with Masking noise. The parameters for f0 and f1 was selected based on prior experience and upon instruction. This means that the latent representation of the image with just 200 neurons is good enough to reconstruct the original image from the noisy input image.

2 Autoencoder as Classifier

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: Regular Autoencoder (Case 1) Denoising Autoencoder (Case 2)
Bias initialization: Zeros
Epochs: 150 Stop Training: Error (1-balanced acc) $<$ 1%

2.2 Results



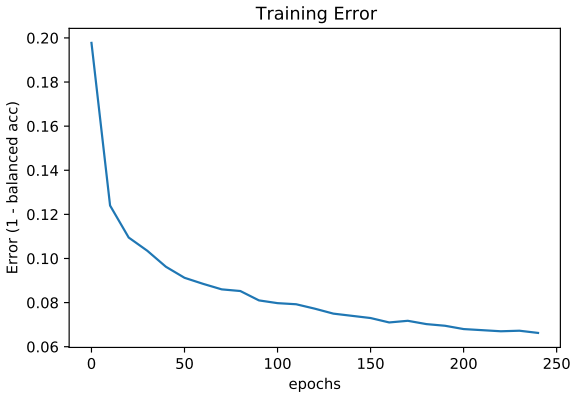
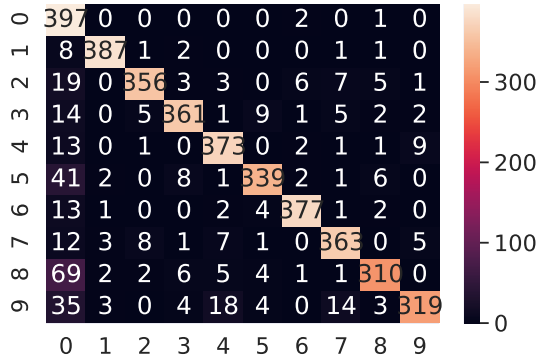


Figure 2.1: *Case 1: Training Error*

Figure 2.2: *Case 2: Training Error*



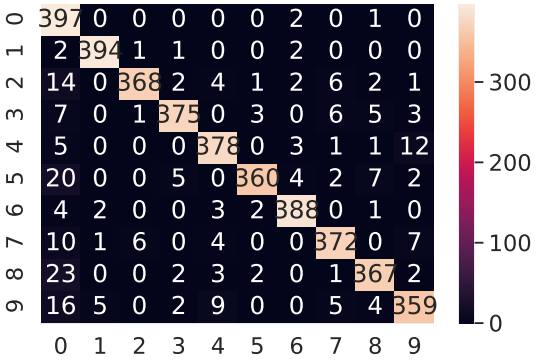


Figure 2.3: *Case 1: Confusion Matrix for digits in Train set*

Figure 2.4: *Case 2: Confusion Matrix for digits in Train Set*

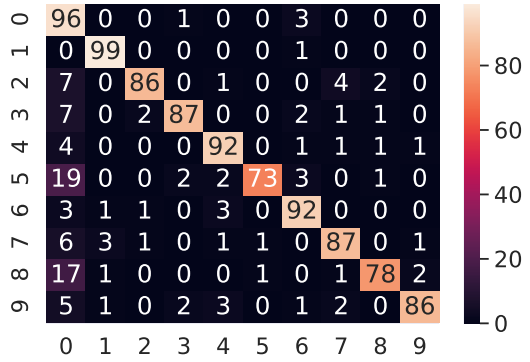


Figure 2.5: *Case 1: Confusion Matrix for digits in Test Set*

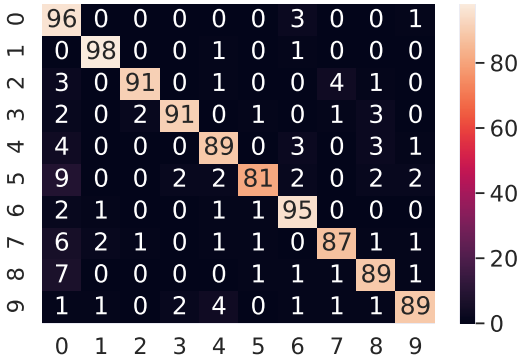


Figure 2.6: *Case 2: Confusion Matrix for digits in Test Set*

2.3 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 2.1 and 2.2 it can be observed that Error rate keeps decreasing as the training progresses and eventually stays constant after 240th epoch. It can also be observed that the initial training error for Case 2 is much lower than Case 1. This is because the initial weights of Case 2 is derived from the denoising autoencoder which has a better latent representation than the regular autoencoder.

From Figure 2.3 and 2.4 from the training set, 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. There are also a lot of misclassified instances where they are predicted as 0. This is due to the shape of 0 which is just a loop which is also present in 6, 8, 9

From Figure 2.5 and 2.6, we can observe that the classification works better in Case 2, when Denoising Autoencoder weights are used than Case 1 when regular autoencoder is used. This is due to the fact that Denoising autoencoder learns a harder task and thus has better latent feature representation.

3 Code Appendix

For futher reference and code implementation for this paper see : `https://github.com/shivchander/denoising-autoencoder-from-scratch`

3.1 main.py

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

'''
Denoising and classification of MNIST Dataset using Autoencoder implemented from scratch
'''

from utils import *
from autoencoder import AutoencoderNN
from classifier import ClassifierNN


if __name__ == '__main__':
    # data = parse_data('dataset/MNISTnumImages5000_balanced.txt', 'dataset/MNISTnumLabels5000_b
    # split_data(data)
    # train = pd.read_csv('dataset/MNIST_Train.csv', sep=",")
    # test = pd.read_csv('dataset/MNIST_Test.csv', sep=",")

    X_train, train_labels, X_test, test_labels = get_train_test('dataset/MNIST_Train.csv', 'data
    X_noisy_train, X_noisy_test = add_mask_noise(X_train, X_test, f0=0.4, f1=0.05)

    # q1 Denoising Autoencoder

    # model = AutoencoderNN()
    # _ = model.fit(X_noisy_train, X_train, 784, 200, 1, 784, learning_rate=0.01, batch_size=32,
    # random_outputs(model, X_noisy_test, X_test)
    # plot_train_test_error(model, X_train, X_test)
    # train_test_digit_error(model, X_train, X_test)
    # plot_features(model.parameters['W1'], title='Denoising Autoencoder')

    # q2 Classifier
    # Case 1: Pretrained weights from HW3 Problem 2
    # model1 = AutoencoderNN()
    # model1.fit(X_train, X_train, 784, 200, 1, 784, learning_rate=0.01, batch_size=32, num_epoc
    #
    # model2 = ClassifierNN()
    # model2.fit(X_train, X_train, 784, 200, 1, 784, pre_trained_weights=model1.parameters,
    #            learning_rate=0.01, batch_size=32, num_epochs=150, plot_error=True)
    #
    # model3 = AutoencoderNN()
    # _ = model3.fit(X_noisy_train, X_train, 784, 200, 1, 784, learning_rate=0.01, batch_size=32
```

```
    #                 plot_error=True)
    #
    # model4 = ClassifierNN()
    # _ = model4.fit(X_train, train_labels, 784, 200, 1, 10, pre_trained_weights=model3.paramete
    #                 learning_rate=0.01, batch_size=32, num_epochs=250, plot_error=True)
    #
    # y1_train_preds = model2.predict(X_train)
    # y1_test_preds = model2.predict(X_test)
    #
    # y2_train_preds = model4.predict(X_train)
    # y2_test_preds = model4.predict(X_test)
    #
    # plot_confusion_matrix(train_labels, y1_train_preds, 'case1train_cm')
    # plot_confusion_matrix(test_labels, y1_test_preds, 'case1test_cm')
    # plot_confusion_matrix(train_labels, y2_train_preds, 'case2train_cm')
    # plot_confusion_matrix(test_labels, y2_test_preds, 'case2test_cm')
```

3.2 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)
```

3.3 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 confusion_matrix
```

```
import seaborn as sns
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 add_mask_noise(X_train, X_test, f0, f1):
    def noisify(data):
        noisy_data = np.copy(data)
        for row in noisy_data:
            zero_indices = np.random.choice(np.arange(784), int(f0*784), replace=False)
            one_indices = np.random.choice(list(set(np.arange(784)) - set(zero_indices)), int(f1
            row[zero_indices] = 0
            row[one_indices] = 1
        return noisy_data

    return noisify(X_train), noisify(X_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, noisy_test, clean_test):
    idx = np.random.randint(1000, size=8)
```

```
    sample_noisy_X = noisy_test[idx, :]
    sample_clean_X = clean_test[idx, :]
    sample_pred = model.predict(sample_noisy_X)

    fig, ax = plt.subplots(2, 8)
    for i in range(8):
        ax[0][i].imshow(sample_clean_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 plot_features(model_weights, title):
    idx = np.random.randint(200, size=20)
    weights = model_weights[idx, :]

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


def plot_confusion_matrix(y_true, y_pred, file_name):

    cm = confusion_matrix(np.argmax(y_true, axis=1), np.argmax(y_pred, axis=1))
    df_cm = pd.DataFrame(cm, range(10), range(10))
    sns.heatmap(df_cm, annot=True, fmt='d')
    plt.show()
    plt.savefig(file_name+'.pdf')
    plt.clf()
```