EECE6036 - Homework 3

Wayne Stegner

November 1, 2020

1 Problem 1

1.1 Problem Summary

The goal of this problem is to create a multi-layer feed-forward neural network for classification of the MNIST dataset. The network was trained using gradient descent with back-propagation with momentum, and is trainable for arbitrary amounts and sizes of hidden layers. During training, the loss is calculated using J_2 loss with threshold values. For performance measurement, the loss is calculated as $1 - hit\_rate$ using winner-take-all on the output layer to define the predicted class.

1.2 Results

1.2.1 System Description

Table 1 shows the hyper-parameters used in training the classifier. These hyper-parameters were found empirically, considering both minimizing the final loss and training time. More work can to find more optimal hyper-parameters, but this set of hyper-parameters produces pretty good results.

Table 1: Classifier Training Hyper-Parameters

Parameter	Value	Description
$hidden\_layer\_size$	192	Neurons in hidden layer
η	0.05	Learning rate
α	0.8	Momentum
max_epochs	500	Maximum training epochs
L	0.25	Lower activation threshold
H	0.75	Upper activation threshold
$patience$	3	Patience before early stopping
es_delta	0.01	Delta value for early stopping

Weight initialization is done randomly on a uniform distribution between $(-a, +a)$ where $a = \sqrt{(\frac{6}{N_{source}+N_{target}})}$, and N_{source} and N_{target} are the numbers of neurons on the source and target layers respectively.

The network utilizes early stopping by monitoring a validation set, which consists of 1000 training points that are set aside before training. If the validation loss does not improve for *patience* steps (1 step = 10 epochs), training halts. The validation is considered improved if the validation is *es_delta* lower than the previous improved validation loss. The weights used by the final network are the weights from the epoch with the lowest validation loss. To improve the frequency of checking for early stopping, the network only uses 1000 training points per epoch.

1.2.2 Network Results

Throughout the duration of training, the loss of the training and validation sets was tracked every 10 epochs. Figure 1 shows a plot of the loss values while training the classifier. The vertical line designates the point where the validation error is minimized, which occurs at epoch 50. The weights at that epoch are the weights used in the final network, where the test loss is 0.070.

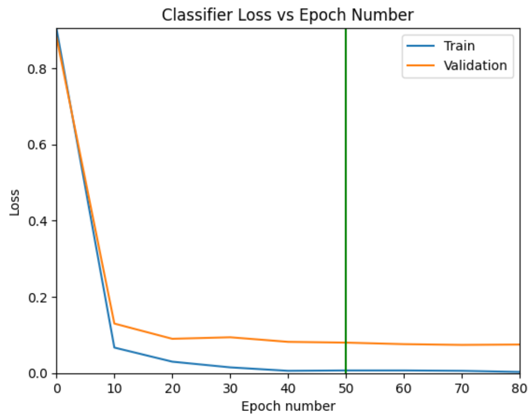
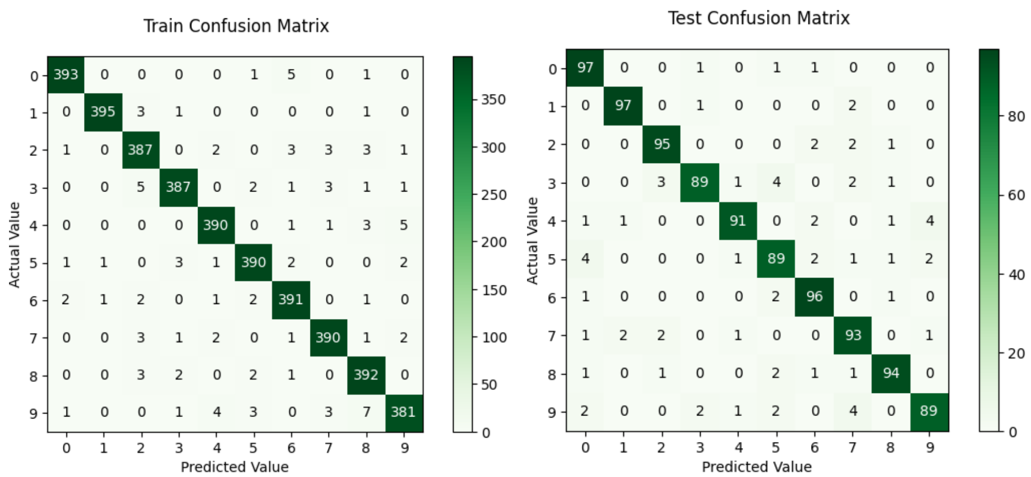


Figure 1: Training and validation loss of the classifier.

Figure 2 shows the confusion matrices for the train and test sets using the weights from the best epoch.



(a)

(b)

Figure 2: Confusion matrices of the train and test sets.

1.3 Discussion and Analysis of Results

Overall, the results look pretty good. The best weights happen pretty early at epoch 50 and get a fairly low loss of 0.070. While testing hyper-parameters, it was observed that when *es_delta* is set to 0 and *patience* is increased to around 5, the test loss would sometimes get as low as 0.05, but that generally occurred after several hundred epochs and is also dependent on weight initialization numbers and dataset shuffling during training. With the early stopping hyper-parameters used by this model, it sacrifices some performance, but it trains significantly faster, making these hyper-parameters ideal for implementing and troubleshooting the network from scratch.

The diagonal of the confusion matrices in Figure 2 is very dark, with hardly any coloring in the incorrect boxes. Many of the incorrect classifications are in two classes which look similar. For example, Figure 2a shows four 9s classified as 4s and five 4s classified as 9s. This mix-up makes a lot of sense, because the shapes of 4s and 9s tend to be similar, especially with some of the sloppy handwriting in MNIST.

1.4 Conclusion

This multi-layer feed-forward neural network classifier is able to effectively classify the MNIST dataset given in this problem. While the results are certainly not state-of-the-art, the trained network classifies over 90% of the test set digits correctly. Further testing in optimizing the hyper-parameters of the network can help to improve the accuracy even further.

2 Problem 2

2.1 Problem Summary

The goal of this problem is to create a multi-layer feed-forward neural network similar to Problem 1, except in this case it will be trained to be an autoencoder. This network has the same features as the network in Problem 1, except the performance measurement is calculated using J_2 loss instead of $1 - hit\_rate$. Additionally, winner-take-all is not used in the output layer, because it does not make sense in the context of an autoencoder.

2.2 Results

2.2.1 System Description

Table 2 shows the hyper-parameters used in training the autoencoder. As with the classifier, these hyper-parameters were found empirically with the same consideration for minimizing final loss and training time.

Table 2: Autoencoder Training Hyper-Parameters

Parameter	Value	Description
$hidden\_layer\_size$	192	Neurons in hidden layer
η	0.005	Learning rate
α	0.8	Momentum
max_epochs	500	Maximum training epochs
L	0	Lower activation threshold
H	1	Upper activation threshold
$patience$	3	Patience before early stopping
es_delta	0.1	Delta value for early stopping

The weight initialization process is the same as in Problem 1, as are the processes for allocating the validation set and applying early stopping.

2.2.2 Network Results

Throughout the duration of training, the loss of the training and validation sets was tracked every 10 epochs. Figure 3 shows a plot of the loss values while training the autoencoder. The vertical line designates the point where the validation error is minimized, which occurs at epoch 160. The weights at that epoch are the weights used in the final network, where the test loss is 2.087.

After training, the loss in each class was measured. Figure 4 shows the loss of each class for the train and test sets using the weights from the best epoch.

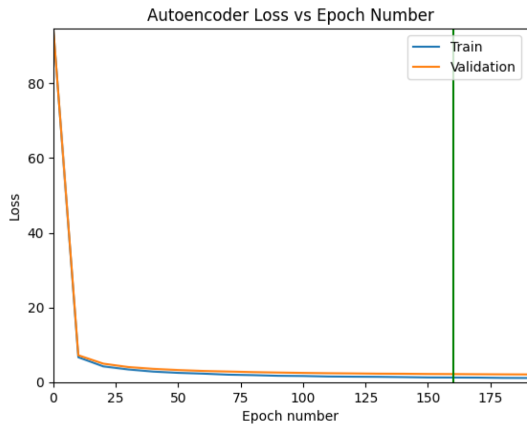


Figure 3: Training and validation loss of the autoencoder.

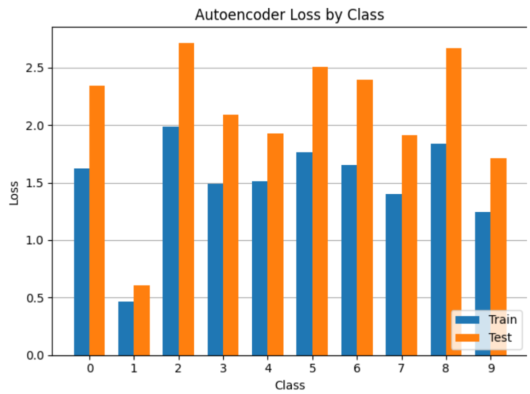


Figure 4: Loss of each class of the train and test sets.

2.2.3 Features

After training, the features learned by 20 arbitrary neurons in the hidden layer of both the classifier and autoencoder were observed by normalizing the weights from 0 to 1 and then displaying them like an image. The images are mapped so 1 is white and 0 is black. Figure 5 shows the features of the classifier, and Figure 6 shows the features of the autoencoder.

2.2.4 Sample Outputs

After training, the outputs of eight random data points from the test set were fed reconstructed the autoencoder, meaning the input data point was presented to the network and the output "prediction" was obtained. Figure 7 shows the original and reconstructed images.

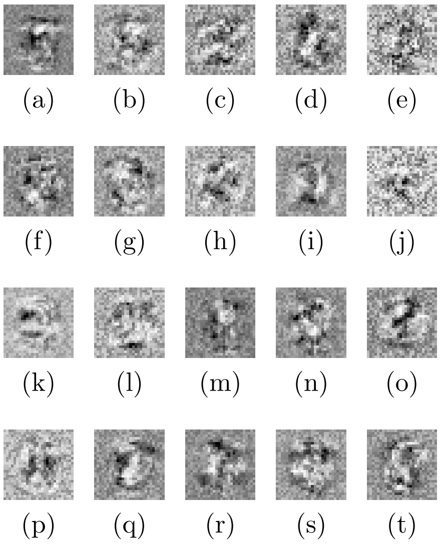


Figure 5: Classifier features.

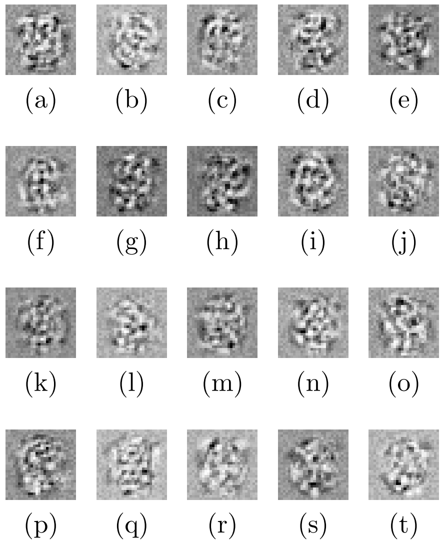


Figure 6: Autoencoder features.

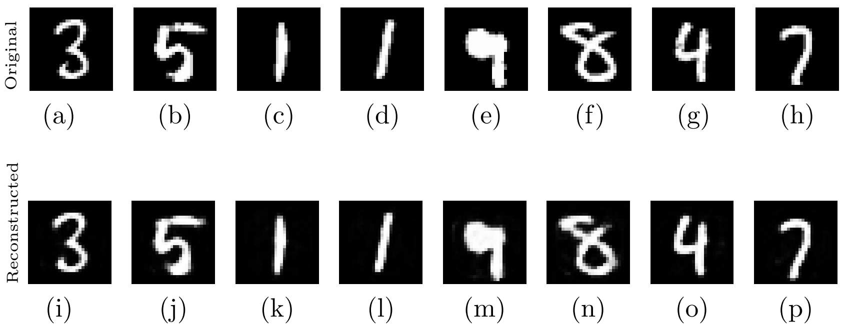


Figure 7: Original (top) and reconstructed (bottom) data points.

2.3 Discussion and Analysis of Results

Overall, the results look pretty good. The autoencoder takes longer to train, with its best weights occurring at epoch 160 with a loss of 2.087. In Figure 4, the loss for the 1s is particularly salient because it is drastically lower than the other classes loss values. This makes sense because a 1 is simply a straight line, so it should be quite simple to recognize and then reconstruct. While none of the loss values are as noticeably high as 1's loss is low, classes 2, 8, and 5 all have loss values above 2.5. I do not have any intuition as to why those numbers have the highest loss, but those digits are kind of similar in the sense that a 2 is roughly a backward 5, and stacking a 2 on top of a 5 roughly makes an 8. At first, I was surprised that 6 has a higher loss than

9, because 6 is just an upside-down 9, but it makes sense that 5 and 6 have similar features and ended up having similar loss values.

The features in Figure 5 and Figure 6 are somewhat difficult to interpret. Between the two feature sets, there is a trend where the center of the image appears to have some distinct patterns, while the edge of the image is either somewhat uniform gray or random TV static. This trend makes sense because the digits in the images tended to be mostly centered, so the edge pixels mostly contained 0. When the presented input is 0, the weight delta is 0 according to the back-propagation formula, meaning that the edge of the images did not get updated as much as the center of the image. In Figure 5, the shapes generally seem to be lines or curves, while in Figure 6 the features look like a bunch of dark dots with a few brighter spots.

2.4 Conclusion

This multi-layer feed-forward neural network autoencoder is able to effectively reconstruct digits from the MNIST dataset. Further testing in optimizing the hyper-parameters of the network can help to improve the reconstruction accuracy even further.

A Code

A.1 settings.py

```
################################################################################
# Imports
################################################################################
import pathlib
import numpy as np


################################################################################
# Constant Values
################################################################################
# Common constants
TRAIN_PORTION = 0.8
SEED = 69420
INPUTS = 784
HIDDEN_LAYER_SIZES = [192]
CLASSES = 10
# Classifier constants
CLASS_POINTS_PER_EPOCH = 1000
CLASS_VALID_POINTS = 1000
CLASS_MAX_EPOCHS = 500
CLASS_ETA = 0.05
CLASS_ALPHA = 0.8
CLASS_L = 0.25
CLASS_H = 0.75
CLASS_PATIENCE = 3
CLASS_ES_DELTA = 0.01
# Autoencoder constants
AUTO_POINTS_PER_EPOCH = 1000
AUTO_VALID_POINTS = 1000
AUTO_MAX_EPOCHS = 500
AUTO_ETA = 0.005
AUTO_ALPHA = 0.8
AUTO_L = 0
AUTO_H = 1
AUTO_PATIENCE = 3
AUTO_ES_DELTA = 0.1


################################################################################
# File locations
################################################################################
# Directory of current file
CODE_DIR = pathlib.Path(__file__).parent.absolute()
# Root directory of the project
ROOT_DIR = CODE_DIR.parent
# Dataset locations
DATA_FILE = CODE_DIR.joinpath('data.txt')
LABEL_FILE = CODE_DIR.joinpath('labels.txt')
DATASET_DIR = CODE_DIR.joinpath('dataset')
DATASET_DIR.mkdir(mode=0o755, exist_ok=True)
TRAIN_DATA_FILE = DATASET_DIR.joinpath('train_data.npy')
TRAIN_LABELS_FILE = DATASET_DIR.joinpath('train_labels.npy')
TEST_DATA_FILE = DATASET_DIR.joinpath('test_data.npy')
TEST_LABELS_FILE = DATASET_DIR.joinpath('test_labels.npy')
# Image directory
IMG_DIR = ROOT_DIR.joinpath('images')
IMG_DIR.mkdir(mode=0o775, exist_ok=True)
# Data directory
DATA_DIR = ROOT_DIR.joinpath('data')
DATA_DIR.mkdir(mode=0o775, exist_ok=True)
# Classifier locations
CLASS_MODEL_DIR = CODE_DIR.joinpath('class')
CLASS_MODEL_DIR.mkdir(mode=0o775, exist_ok=True)
CLASS_LOSS_PLOT = IMG_DIR.joinpath('class_loss.png')
CLASS_PARAM_CSV = DATA_DIR.joinpath('class_parameters.csv')
CLASS_BEST_EPOCH = DATA_DIR.joinpath('class_best_epoch.dat')
CLASS_TRAIN_CONF = IMG_DIR.joinpath('train_conf_mat.png')
CLASS_TEST_CONF = IMG_DIR.joinpath('test_conf_mat.png')
CLASS_TEST_LOSS = DATA_DIR.joinpath('class_test_loss.dat')
CLASS_FEAT_DIR = IMG_DIR.joinpath('class_feat')
CLASS_FEAT_DIR.mkdir(mode=0o775, exist_ok=True)
# Autoencoder files
AUTO_MODEL_DIR = CODE_DIR.joinpath('auto')
AUTO_MODEL_DIR.mkdir(mode=0o775, exist_ok=True)
AUTO_LOSS_PLOT = IMG_DIR.joinpath('auto_loss.png')
AUTO_PARAM_CSV = DATA_DIR.joinpath('auto_parameters.csv')
AUTO_BEST_EPOCH = DATA_DIR.joinpath('auto_best_epoch.dat')
```

```python
AUTO_BAR = IMG_DIR.joinpath('loss_bar_plot.png')
AUTO_SAMPLE_DIR = IMG_DIR.joinpath('auto_samples')
AUTO_SAMPLE_DIR.mkdir(mode=0o775, exist_ok=True)
AUTO_TEST_LOSS = DATA_DIR.joinpath('auto_test_loss.dat')
AUTO_FEAT_DIR = IMG_DIR.joinpath('auto_feat')
AUTO_FEAT_DIR.mkdir(mode=0o775, exist_ok=True)

################################################################################
# Load Dataset
################################################################################
if TRAIN_DATA_FILE.exists():
    train_data = np.load(str(TRAIN_DATA_FILE))
if TRAIN_LABELS_FILE.exists():
    train_labels = np.load(str(TRAIN_LABELS_FILE))
if TEST_DATA_FILE.exists():
    test_data = np.load(str(TEST_DATA_FILE))
if TEST_LABELS_FILE.exists():
    test_labels = np.load(str(TEST_LABELS_FILE))
```

A.2 dataset.py

```python
################################################################################
# Imports
################################################################################
from settings import *
import numpy as np


def shufflePair(data, labels):
    """Shuffle a pair of data and labels in place
    Parameters
    ----------
    data, labels : np.ndarray
        Data and labels to be shuffled
    """
    assert len(data) == len(labels), \
            'Size mismatch between data and labels'
    indeces = np.random.permutation(len(data))
    data[...] = data[indeces]
    labels[...] = labels[indeces]


if __name__ == '__main__':
    # Additional imports
    import pathlib
    # Seed for consistency
    np.random.seed(SEED)
    ################################################################################
    # Import dataset
    ################################################################################
    # Check that the files exist
    assert DATA_FILE.exists(), f'File not found: {str(data_file)}'
    assert LABEL_FILE.exists(), f'File not found: {str(label_file)}'
    # Read data and labels from txt file
    data = list()
    with open(DATA_FILE) as data_f:
        data = data_f.readlines()
    data = [d.split() for d in data]
    labels = list()
    with open(LABEL_FILE) as label_f:
        labels = label_f.readlines()
    labels = [int(l) for l in labels]
    # Sort into bins based on class
    data_points = dict()
    classes = list()
    for d, l in zip(data, labels):
        if l not in classes:
            classes.append(l)
            data_points[l] = list()
        data_points[l].append(d)
    classes.sort()
    # Turn data lists into numpy arrays
    for l in classes:
        data_points[l] = np.array(data_points[l], dtype=np.float64)

    ################################################################################
    # Partition dataset
    ################################################################################
    train_data = list()
    train_labels = list()
    test_data = list()
    test_labels = list()
```

```python
    # Shuffle the data points first
    for l in classes:
        np.random.shuffle(data_points[l])
    # Iterate through the data by class and partition
    points_per_class = len(data_points[classes[0]])
    train_per_class = int(TRAIN_PORTION * points_per_class)
    for l in classes:
        for i, d in enumerate(data_points[l]):
            if i < train_per_class:
                train_data.append(d)
                train_labels.append(l)
            else:
                test_data.append(d)
                test_labels.append(l)
    # Turn lists into numpy arrays
    train_data = np.array(train_data)
    train_labels = np.array(train_labels)
    test_data = np.array(test_data)
    test_labels = np.array(test_labels)
    # Turn labels into one-hot arrays
    train_labels = np.eye(len(classes))[train_labels]
    test_labels = np.eye(len(classes))[test_labels]
    # Shuffle arrays
    shufflePair(train_data, train_labels)
    shufflePair(test_data, test_labels)

    ###########################################################################
    # Save arrays
    ###########################################################################
    np.save(str(TRAIN_DATA_FILE), train_data)
    np.save(str(TRAIN_LABELS_FILE), train_labels)
    np.save(str(TEST_DATA_FILE), test_data)
    np.save(str(TEST_LABELS_FILE), test_labels)
```

A.3 layer.py

```python
###############################################################################
# Imports
###############################################################################
import numpy as np


class Layer:
    def __init__(
            self,
            weight_file = None,
            num_neurons = None,
            inputs = None,
        ):
        """Initialize Layer object either randomly or by a weight file
        Parameters
        ----------
        weight_file : str, optional
            File to load pre-existing weights from
        num_neurons, inputs : int, optional
            Number of hidden neurons and inputs to the layer
        """
        if weight_file:
            # Load saved weights
            self.loadWeights(weight_file)
            self.num_neurons = self.w.shape[0]
            self.inputs = self.w.shape[1] - 1
        else:
            # Randomly initialize weights
            self.num_neurons = num_neurons
            self.inputs = inputs
            self.initW()
        # States to save for back-prop
        self.x = None # Input for current pass
        self.s = None # Net inputs pre-activation function
        self.y = None # Final output of the layer
        self.delta = None # Delta of this layer
        self.w_change = np.zeros(self.w.shape) # Weight changes

    def initW(self):
        """Initialize weights for the perceptron using Xavier initialization
        """
        w_shape = (self.num_neurons, self.inputs + 1)
        a = np.sqrt(6 / (self.inputs + self.num_neurons))
        self.w = np.random.uniform(-a, a, size=w_shape)
        self.w_change = np.zeros(self.w.shape)
```

```python
    def forwardPass(self, x):
        """Pass the input forward through the layer
        Parameters
        ----------
        x : np.ndarray of np.float64
            Input to the layer
        Returns
        -------
        np.array of np.float64
            Output of the layer
        """
        # Add bias input
        x = np.concatenate(([1], x))
        # Save inputs for back-prop
        self.x = x
        # Dot product weights * inputs
        self.s = np.matmul(self.w, x)
        # Pass through sigmoid activation
        self.y = 1.0 / (1 + np.exp(-1 * self.s))
        return self.y

    def setDelta(self):
        """Calculate delta value, different for Output and Hidden layer
        Must be implemented per-class
        """
        raise NotImplementedError('Cannot call from Layer class')

    def getWChange(self, eta, alpha):
        """Calculate weight updates for the most recent forward pass
        Requires delta to be calculated (varies between hidden/output layers)
        Parameters
        ----------
        eta : float
            Learning rate for weight adjustments
        alpha : float
            Momentum scalar
        """
        # Set delta value
        self.setDelta()
        # Pre-scale weight change for momentum
        self.w_change *= alpha
        # Calculate new weight change
        d = self.delta.reshape(len(self.delta), 1)
        x = self.x.reshape(1, len(self.x))
        new_change = eta * np.matmul(d, x)
        self.w_change += new_change

    def changeW(self):
        """Apply w_change to the weights
        """
        self.w += self.w_change

    def saveWeights(self, weight_file):
        """Save weights to a file
        Parameters
        ----------
        weight_file : str
            File name to save weights in
        """
        np.save(str(weight_file), self.w)

    def loadWeights(self, weight_file):
        """Save weights to a file
        Parameters
        ----------
        weight_file : str
            File name to save weights in
        """
        self.w = np.load(str(weight_file))


if __name__ == '__main__':
    print('Warning: Tests for this file are deprecated')
```

A.4 output_layer.py

```python
################################################################################
# Imports
################################################################################
from layer import Layer
import numpy as np
```

```python
class OutputLayer(Layer):
    def setLabel(self, label):
        '''Set the labels for this batch
        Needed for calculating delta for this layer
        Parameters
        ----------
        label : np.ndarray
            One-hot encoded label
        '''
        self.label = label

    def setDelta(self):
        '''Calculate delta for the output layer
        '''
        e = self.label - self.y
        y_der = self.y * (1 - self.y)
        self.delta = e * y_der

    def thresholdOutputs(self, L, H):
        '''Threshold outputs based on the labels
        Parameters
        ----------
        L, H: float
            Low and high thresholds for outputs
        '''
        self.y[(self.y <= L) * (self.label == 0)] = 0
        self.y[(self.y >= H) * (self.label == 1)] = 1


if __name__ == '__main__':
    print('Warning: Tests for this file are deprecated')
```

A.5 hidden_layer.py

```python
################################################################################
# Imports
################################################################################
from layer import Layer
import numpy as np


class HiddenLayer(Layer):
    def setDownstreamSum(self, w, delta):
        """Sum the product of w and delta for the next layer
        Needed for calculating delta for this layer
        Parameters
        ----------
        w : np.ndarray
            Matrix of weight values for the next layer
        delta : np.ndarray
            Matrix of delta values for the next layer
        """
        self.downstream_sum = np.matmul(w[:,:-1].transpose(), delta)

    def setDelta(self):
        """Calculate delta for the hidden layer
        """
        # Derivative of sigmoid using last forward pass
        output_der = self.y * (1 - self.y)
        self.delta = output_der * self.downstream_sum


if __name__ == '__main__':
    print('Warning: Tests for this file are deprecated')
```

A.6 mlp.py

```python
################################################################################
# Imports
################################################################################
# Custom imports
from settings import *
from dataset import shufflePair
from hidden_layer import HiddenLayer
from output_layer import OutputLayer
# External imports
import numpy as np
import enlighten # Progress bar for training
```

```python
class MLP:
    def __init__(
            self,
            input_size=INPUTS,
        ):
        """Initialize Multi-Layer Perceptron object
        Parameters
        ----------
        input_size : int
            Number of inputs to the network
        """
        # Initialize parameters
        self.layers = list()
        self.input_size = input_size

    def addLayer(self, file_name=None, neurons=None, output=False):
        """Add a layer to the network
        """
        if len(self.layers) == 0:
            input_size = self.input_size
        else:
            input_size = self.layers[-1].num_neurons
        if file_name:
            if output:
                self.layers.append(OutputLayer(weight_file=file_name))
            else:
                self.layers.append(HiddenLayer(weight_file=file_name))
        else:
            if output:
                self.layers.append(OutputLayer(num_neurons=neurons,
                    inputs=input_size))
            else:
                self.layers.append(HiddenLayer(num_neurons=neurons,
                    inputs=input_size))

    def predict(self, data, one_hot):
        """Predict the output given an input
        Parameters
        ----------
        data : np.ndarray
            Data point to predict on
        one_hot : bool
            Whether the output should be one-hot or raw
        Returns
        -------
        np.ndarray
            Array of prediction
        """
        # Forward pass through all the layers
        layer_output = data
        for l in self.layers:
            layer_output = l.forwardPass(layer_output)
        if one_hot:
            max_idx = np.argmax(layer_output)
            pred = np.eye(10)[max_idx]
        else:
            pred = layer_output
        return pred

    def trainPoint(self, data, label, eta, alpha, L, H):
        """Update the weights for a single point
        Parameters
        ----------
        data : np.ndarray
            The data point
        labels : np.ndarray
            One-hot encoded label
        eta : float
            Learning rate
        alpha : float
            Momentum scalar
        L, H : float, optional
            Low and high thresholds for training
        """
        # Weight change for last layer
        self.predict(data, one_hot=False)
        self.layers[-1].setLabel(label)
        self.layers[-1].thresholdOutputs(L, H)
        self.layers[-1].getWChange(eta, alpha)
        # Back-prop error
        for i in range(len(self.layers)-2, -1, -1):
```

```
            down_w = self.layers[i+1].w
            down_delta = self.layers[i+1].delta
            self.layers[i].setDownstreamSum(down_w, down_delta)
            self.layers[i].getWChange(eta, alpha)
        # Apply weight changes
        for l in self.layers:
            l.changeW()

    def logError(self,
            train_data,
            train_labels,
            valid_data,
            valid_labels,
            epoch_num,
        ):
        """Log error for train/test data for a given epoch
        Parameters
        ----------
        train_data : np.ndarray
            Array of data points for the train set
        train_labels : np.ndarray
            Array of labels for the train set
        valid_data : np.ndarray
            Array of data points for the validation set
        valid_labels : np.ndarray
            Array of labels for the validation set
        epoch_num : int
            The current epoch number
        """
        # Evaluate and log train error
        train_err = self.eval(train_data, train_labels)
        self.train_err.append(train_err)
        # Evaluate and log test error
        valid_err = self.eval(valid_data, valid_labels)
        self.valid_err.append(valid_err)
        # Record epoch number
        self.epoch_num.append(epoch_num)
        # Print out metrics
        print(f'Epoch {epoch_num}')
        print(f'\tTrain Loss:\t\t{train_err:0.03f}')
        print(f'\tValidation Loss:\t{valid_err:0.03f}')

    def train(self,
            data,
            labels,
            points_per_epoch,
            valid_points,
            max_epochs,
            eta,
            alpha,
            L,
            H,
            patience,
            es_delta,
            save_dir,
        ):
        """Train the network up to the desired number of epochs
        Parameters:
        -----------
        data : np.ndarray
            Array of training data points
        labels : np.ndarray
            Array of training labels
        points_per_epoch : int
            Number of training points to use in each epoch
        valid_points : int
            Number of training points to set aside for validation
            These points are set aside before training
        max_epochs : int
            Maximum number of epochs to train
        eta, alpha : float
            Learning rate and Momentum scalar
        L, H : float
            Low and high thresholds for training
        patience : int
            Amount of epochs with no improvement before early stopping
        es_delta : float
            Required improvement for early stopping
        save_dir : pathlib.Path or str
            Directory to save best model parameters in
        """
        # Initialize progress bar
        pbar_manager = enlighten.get_manager()
```

```
        pbar = pbar_manager.counter(total=max_epochs, desc='Training',
                unit = 'epochs', leave=False)
        # Lists to track metrics
        self.train_err = list()
        self.valid_err = list()
        self.epoch_num = list()
        # Set aside validation partition (after shuffling)
        assert valid_points + points_per_epoch <= len(data), \
                'Not enough data for validation points and points per epoch'
        shufflePair(data, labels)
        valid_data = data[:valid_points]
        valid_labels = labels[:valid_points]
        train_data = data[valid_points:]
        train_labels = labels[valid_points:]
        # Log initial accuracy (using whole train partition)
        self.logError(
                train_data,
                train_labels,
                valid_data,
                valid_labels,
                0,
        )
        # Iterate through epochs or until early stopping
        impatience = 0
        self.best_weights_epoch = 0
        best_loss = np.inf
        for e in range(1, max_epochs+1):
            pbar.update()
            shufflePair(train_data, train_labels)
            epoch_train_data = train_data[:points_per_epoch]
            epoch_train_labels = train_labels[:points_per_epoch]
            for d, l in zip(epoch_train_data, epoch_train_labels):
                self.trainPoint(d, l, eta, alpha, L, H)
            # Log data every 10 epochs
            if (e % 10) == 0:
                self.logError(
                        epoch_train_data,
                        epoch_train_labels,
                        valid_data,
                        valid_labels,
                        e,
                )
                # Check for early stopping
                if ((best_loss - self.valid_err[-1]) >= es_delta):
                    impatience = 0
                    self.best_weights_epoch = e
                    best_loss = self.valid_err[-1]
                    for i, l in enumerate(self.layers):
                        layer_name = save_dir.joinpath(f'layer_{i:02d}')
                        l.saveWeights(layer_name)
                else:
                    impatience += 1
                    # We have become too impatient
                    if impatience >= patience:
                        print(f'* * * Early stopping hit * * *')
                        break
        pbar.close()


if __name__ == '__main__':
    print('Warning: This file does not do anything.')
    print('Run either classifier.py or autoencoder.py.')
```

A.7 classifier.py

```
################################################################################
# Imports
################################################################################
# Custom imports
from mlp import MLP
# External imports
import numpy as np
import matplotlib.pyplot as plt


class Classifier(MLP):
    def eval(self, data, labels):
        """Evaluate error on a data set
        Parameters
        ----------
        data : np.ndarray
            Array of data points
```

```
        labels : np.ndarray
            Array of labels for the data
        Returns
        -------
        float
            Error (1 - Hit Rate)
        """
        correct = 0
        for d, l in zip(data, labels):
            # Make prediction
            pred = self.predict(d, one_hot=True)
            # Check equality with label
            if (np.sum(np.abs(pred - l)) == 0):
                correct += 1
        hit_rate = correct / len(data)
        return (1 - hit_rate)


if __name__ == '__main__':
    # Additional imports
    from settings import *
    import pathlib
    import csv
    # Seed for consistency
    np.random.seed(SEED)
    # Delete old model
    for f in CLASS_MODEL_DIR.iterdir():
        f.unlink()
    # Test network
    classifier = Classifier(input_size=INPUTS)
    for h in HIDDEN_LAYER_SIZES:
        classifier.addLayer(neurons=h, output=False)
    classifier.addLayer(neurons=CLASSES, output=True)
    # Train the network
    print('* * * Begin training classifier * * *')
    classifier.train(
        train_data,
        train_labels,
        points_per_epoch=CLASS_POINTS_PER_EPOCH,
        valid_points=CLASS_VALID_POINTS,
        max_epochs=CLASS_MAX_EPOCHS,
        eta=CLASS_ETA,
        alpha=CLASS_ALPHA,
        L=CLASS_L,
        H=CLASS_H,
        patience=CLASS_PATIENCE,
        es_delta=CLASS_ES_DELTA,
        save_dir=CLASS_MODEL_DIR,
    )
    # Plot loss over epochs
    plt.figure()
    plt.plot(classifier.epoch_num, classifier.train_err)
    plt.plot(classifier.epoch_num, classifier.valid_err)
    plt.axvline(x=classifier.best_weights_epoch, c='g')
    plt.legend(['Train', 'Validation'], loc='upper right')
    plt.title(f'Classifier Loss vs Epoch Number')
    plt.xlabel('Epoch number')
    plt.xlim([0, classifier.epoch_num[-1]])
    plt.ylabel('Loss')
    plt.ylim([0, max(max(classifier.train_err), max(classifier.valid_err))])
    plt.savefig(str(CLASS_LOSS_PLOT), bbox_inches='tight', pad_inches=0)
    plt.close()
    # Save parameters to CSV
    csv_rows = list()
    csv_rows.append(['Parameter', 'Value', 'Description'])
    csv_rows.append(['$hidden\\_layer\\_size$', str(HIDDEN_LAYER_SIZES[0]),
        'Neurons in hidden layer'])
    csv_rows.append(['$\\eta$', str(CLASS_ETA), 'Learning rate'])
    csv_rows.append(['$\\alpha$', str(CLASS_ALPHA), 'Momentum'])
    csv_rows.append(['$max\\_epochs$', str(CLASS_MAX_EPOCHS),
        'Maximum training epochs'])
    csv_rows.append(['$L$', str(CLASS_L), 'Lower activation threshold'])
    csv_rows.append(['$H$', str(CLASS_H), 'Upper activation threshold'])
    csv_rows.append(['$patience$', str(CLASS_PATIENCE),
        'Patience before early stopping'])
    csv_rows.append(['$es\\_delta$', str(CLASS_ES_DELTA),
        'Delta value for early stopping'])
    with open(str(CLASS_PARAM_CSV), 'w') as csv_file:
        csv_writer = csv.writer(csv_file)
        csv_writer.writerows(csv_rows)
    # Write best epoch
    with open(str(CLASS_BEST_EPOCH), 'w') as best_epoch_file:
        best_epoch_file.write(str(classifier.best_weights_epoch))
```

A.8 autoencoder.py

```
################################################################################
# Imports
################################################################################
from mlp import MLP
import numpy as np
import matplotlib.pyplot as plt


class Autoencoder(MLP):
    def eval(self, data, labels):
        """Evaluate error on a data set
        Parameters
        ----------
        data : np.ndarray
            Array of data points
        labels : np.ndarray
            Array of labels for the data
        Returns
        -------
        float
            Error (1 - Hit Rate)
        """
        # Calculate loss
        total_loss = 0
        for d, l in zip(data, labels):
            pred = self.predict(d, one_hot=False)
            loss = np.sum(np.square(l - pred))
            total_loss += loss / 2
        total_loss /= len(data)
        return total_loss


if __name__ == '__main__':
    # Additional imports
    from settings import *
    import pathlib
    import csv
    # Seed for consistency
    np.random.seed(SEED)
    # Delete old model
    for f in AUTO_MODEL_DIR.iterdir():
        f.unlink()
    # Training constants
    # Test network
    autoencoder = Autoencoder(input_size=INPUTS)
    for h in HIDDEN_LAYER_SIZES:
        autoencoder.addLayer(neurons=h, output=False)
    autoencoder.addLayer(neurons=INPUTS, output=True)
    # Train the network
    print('* * * Begin training autoencoder * * *')
    autoencoder.train(
        train_data,
        train_data,
        points_per_epoch=AUTO_POINTS_PER_EPOCH,
        valid_points=AUTO_VALID_POINTS,
        max_epochs=AUTO_MAX_EPOCHS,
        eta=AUTO_ETA,
        alpha=AUTO_ALPHA,
        L=AUTO_L,
        H=AUTO_H,
        patience=AUTO_PATIENCE,
        es_delta=AUTO_ES_DELTA,
        save_dir=AUTO_MODEL_DIR,
    )
    plt.figure()
    plt.plot(autoencoder.epoch_num, autoencoder.train_err)
    plt.plot(autoencoder.epoch_num, autoencoder.valid_err)
    plt.axvline(x=autoencoder.best_weights_epoch, c='g')
    plt.legend(['Train', 'Validation'], loc='upper right')
    plt.title(f'Autoencoder Loss vs Epoch Number')
    plt.xlabel('Epoch number')
    plt.xlim([0, autoencoder.epoch_num[-1]])
    plt.ylabel('Loss')
    plt.ylim([0, max(max(autoencoder.train_err), max(autoencoder.valid_err))])
    plt.savefig(str(AUTO_LOSS_PLOT), bbox_inches='tight', pad_inches=0)
    plt.close()
    # Save parameters to CSV
    print('Saving training parameters')
    csv_rows = list()
    csv_rows.append(['Parameter', 'Value', 'Description'])
    csv_rows.append(['$hidden\\_layer\\_size$', str(HIDDEN_LAYER_SIZES[0]),
        'Neurons in hidden layer'])
```

```
    csv_rows.append(['$\\eta$', str(AUTO_ETA), 'Learning rate'])
    csv_rows.append(['$\\alpha$', str(AUTO_ALPHA), 'Momentum'])
    csv_rows.append(['$max\\_epochs$', str(AUTO_MAX_EPOCHS),
        'Maximum training epochs'])
    csv_rows.append(['$L$', str(AUTO_L), 'Lower activation threshold'])
    csv_rows.append(['$H$', str(AUTO_H), 'Upper activation threshold'])
    csv_rows.append(['$patience$', str(AUTO_PATIENCE),
        'Patience before early stopping'])
    csv_rows.append(['$es\\_delta$', str(AUTO_ES_DELTA),
        'Delta value for early stopping'])
    with open(str(AUTO_PARAM_CSV), 'w') as csv_file:
        csv_writer = csv.writer(csv_file)
        csv_writer.writerows(csv_rows)
    # Write best epoch
    with open(str(AUTO_BEST_EPOCH), 'w') as best_epoch_file:
        best_epoch_file.write(str(autoencoder.best_weights_epoch))
```

A.9 test_classifier.py

```
###############################################################################
# Imports
###############################################################################
# Custom imports
from settings import *
from classifier import Classifier
# External imports
import numpy as np
import pathlib
import matplotlib.pyplot as plt


def makeConfMat(classifier, data, labels, plot_name, title):
    '''Generate and save a confusion matrix
    Parameters
    ----------
    classifier : Classifier
        Classifier for use in classification
    data : np.ndarray
        Array of data values
    labels : np.ndarray
        Array of labels as one-hot-vectors
    plot_name : pathlib.Path or str
        File name to save the matrix as
    title : str
        Title of the confusion matrix
    '''
    assert len(data) == len(labels), \
            'Size mismatch between data and labels'
    conf_mat = np.zeros((CLASSES, CLASSES))
    for d, l in zip(data, labels):
        pred = classifier.predict(d, one_hot=True)
        conf_mat += l.reshape((CLASSES,1)) * pred.reshape((1,CLASSES))
    # Plot confusion matrix and save
    plt.figure()
    plt.suptitle(title)
    plt.imshow(conf_mat, cmap='Greens')
    for i in range(len(conf_mat)):
        for j in range(len(conf_mat[i])):
            color = 'k' if (conf_mat[i][j] <= 50) else 'w'
            plt.text(j, i, f'{int(conf_mat[i][j])}',
                    va='center', ha='center', color=color)
    plt.xlabel('Predicted Value')
    plt.xticks(range(CLASSES))
    plt.ylabel('Actual Value')
    plt.yticks(range(CLASSES))
    plt.colorbar()
    plt.tight_layout()
    plt.savefig(str(plot_name), bbox_inches='tight', pad_inches=0)
    plt.close()


# Seed for consistency
np.random.seed(SEED)
# Load best weights back up and make confusion matrices
classifier = Classifier(input_size=INPUTS)
weight_files = sorted(CLASS_MODEL_DIR.iterdir())
for weight_file in weight_files[:-1]:
    classifier.addLayer(file_name=weight_file, output=False)
classifier.addLayer(file_name=weight_files[-1], output=True)
train_conf_title = 'Train Confusion Matrix'
makeConfMat(classifier, train_data, train_labels, CLASS_TRAIN_CONF,
        title=train_conf_title)
```

```
test_conf_title = 'Test Confusion Matrix'
makeConfMat(classifier, test_data, test_labels, CLASS_TEST_CONF,
        title=test_conf_title)
test_err = classifier.eval(test_data, test_labels)
print(f'Test error: {test_err:0.3f}')
with open(str(CLASS_TEST_LOSS), 'w') as loss_f:
    loss_f.write(f'{test_err:0.3f}')
```

A.10 test_autoencoder.py

```
###############################################################################
# Imports
###############################################################################
# Custom imports
from autoencoder import Autoencoder
from settings import *
# External imports
import numpy as np
import pathlib
import matplotlib
import matplotlib.pyplot as plt


def splitClasses(data, labels):
    '''Split up dataset by class
    Parameters
    ----------
    data, labels : np.ndarray
        Arrays of data points and labels
    Returns
    -------
    list
        Data split up by class
    '''
    split_data = list()
    for i in range(CLASSES):
        split_data.append(list())
    for d, l in zip(data, labels):
        idx = np.argmax(l)
        split_data[idx].append(d)
    return split_data

def getLossByClass(autoencoder, data, labels):
    '''
    Parameters
    ----------
    autoencoder : Autoencoder
        Autoencoder for use in loss calculation
    data, labels : np.ndarray
        Arrays of data points and labels
    Returns
    -------
    float
        Loss values by class
    '''
    loss = list()
    split_data = splitClasses(data, labels)
    for i, d in enumerate(split_data):
        print(f'Evaluating class {i}')
        loss.append(autoencoder.eval(d, d))
    return loss

def getSamplePoints(data, n):
    '''Get sample points from the given data set
    Parameters
    ----------
    data : np.ndarray
        Array of data points
    n : int
        Number of sample points
    Returns
    -------
    np.ndarray
        Reduced list of data points
    '''
    indeces = np.random.choice(np.arange(len(data)), 8, replace=False)
    return data[indeces]

def drawSamples(autoencoder, data, num_samples, dir_name, title):
    '''Draw the output predictions and save them
    Parameters
    ----------
```

```
    autoencoder : Autoencoder
        Autoencoder for use in inference
    data : np.ndarray
        Array of data points
    num_samples : int
        Number of sample points
    dir_name : str
        Name of the directory to save the images to
    title : str
        Title of the plot
    '''
    sample_points = getSamplePoints(data, num_samples)
    for i, d in enumerate(sample_points):
        d_name = dir_name.joinpath(f'orig_{i}.png')
        matplotlib.image.imsave(str(d_name), d.reshape(28, 28, order='F'), cmap='Greys_r')
        pred = autoencoder.predict(d, one_hot=False)
        p_name = dir_name.joinpath(f'pred_{i}.png')
        matplotlib.image.imsave(str(p_name), pred.reshape(28, 28, order='F'), cmap='Greys_r')


# Seed for consistency
np.random.seed(SEED)
# Load best weights back up
autoencoder = Autoencoder(input_size=INPUTS)
weight_files = sorted(AUTO_MODEL_DIR.iterdir())
for weight_file in weight_files[:-1]:
    autoencoder.addLayer(file_name=weight_file, output=False)
autoencoder.addLayer(file_name=weight_files[-1], output=True)
# Test on all data and draw samples
test_err = autoencoder.eval(test_data, test_data)
print(f'Test loss: {test_err:0.3f}')
sample_title = 'Autoencoder Sample Outputs'
drawSamples(autoencoder, test_data, 8, AUTO_SAMPLE_DIR, sample_title)
# Graph loss by class
print('Testing train set')
train_loss = getLossByClass(autoencoder, train_data, train_labels)
print('Testing test set')
test_loss = getLossByClass(autoencoder, test_data, test_labels)
x = np.arange(len(train_loss))
plt.figure()
rect_width = 0.35
plt.bar(x-rect_width/2, train_loss, rect_width, label='Train')
plt.bar(x+rect_width/2, test_loss, rect_width, label='Test')
plt.title('Autoencoder Loss by Class')
plt.xlabel('Class')
plt.xticks(x)
plt.ylabel('Loss')
plt.grid(axis='y')
plt.gca().set_axisbelow(True)
plt.legend(loc='lower right')
plt.tight_layout()
plt.savefig(str(AUTO_BAR), bbox_inches='tight', pad_inches=0)
with open(str(AUTO_TEST_LOSS), 'w') as loss_f:
    loss_f.write(f'{test_err:0.3f}')
```

A.11 features.py

```
###############################################################################
# Imports
###############################################################################
# Custom imports
from settings import *
from classifier import Classifier
from autoencoder import Autoencoder
# External imports
import numpy as np
import pathlib
import matplotlib
import matplotlib.pyplot as plt


def drawFeatures(weights, dir_name):
    """Draw the output predictions and save them
    Parameters
    ----------
    weights : np.ndarray
        Array of weights for the neurons
    dir_name : str
        Name of the directory to save the images to
    """
    for i, w in enumerate(weights):
        # Remove bias and normalize on [0, 1]
```

```
        w = w[1:]
        w -= w.min()
        w /= (w.max() - w.min())
        w_name = dir_name.joinpath(f'feat_{i:02d}.png')
        matplotlib.image.imsave(str(w_name), w.reshape(28, 28, order='F'),
                cmap='Greys_r')


# Seed for consistency
np.random.seed(SEED)
# Load best weights back up for each model
autoencoder = Autoencoder(input_size=INPUTS)
for weight_file in sorted(AUTO_MODEL_DIR.iterdir()):
    autoencoder.addLayer(file_name=weight_file)
classifier = Classifier(input_size=INPUTS)
for weight_file in sorted(CLASS_MODEL_DIR.iterdir()):
    classifier.addLayer(file_name=weight_file)
# Neurons to check
neuron_count = classifier.layers[0].num_neurons
neurons = np.random.choice(np.arange(neuron_count), 20, replace=False)
class_weights = classifier.layers[0].w[neurons]
drawFeatures(class_weights, CLASS_FEAT_DIR)
auto_weights = autoencoder.layers[0].w[neurons]
drawFeatures(auto_weights, AUTO_FEAT_DIR)
```