EECE6036: Intelligent Systems
Homework # 4

Zuguang Liu (M10291593)

April 23, 2021

1 Denoising Autoencoder

1.1 Problem Statement

An autoencoder shall be implemented by a two-layer neural network to remove noise applied to the MNIST dataset. Then the performance and the model itself is compared with the autoencoder constructed in Homework #3, whose purpose was simply to reconstruct the images.

To tell apart the two models, from now on the report uses **denoising autoencoder** to address this new network, and call the previous one **reconstructing autoencoder**.

1.2 System Description

Using the same stratified sampling policy the previous Homework used, the preprocessing section partitions the total 5,000 data points into 4,000 for training and 1,000 for testing, where there are 400 and 100 images for each model respectively. Over the training, each image is tempered with noise by a certain policy as input, and the original image is learned as target to fulfill the denoising purpose.

The noise policy generates "salt and pepper" noise (salt pixels = 1, pepper pixels = 0) whose densities are approximately 5.56% and 44.44%. This is due to the fact that the dark (< 0.5) vs bright pixels (> 0.5) in an image is at a ratio around 8:1 on average, and the goal of total noise density = 50%.

The autoencoder is implemented by a 2-layer neural network with a hidden layer and an output layer. **The hidden layer incorporates 128 neurons** that transfer the noisy image into a 128-dimensional feature space. Then the output layer brings the reduced feature space back into the original 784 pixel values but with reduced noise. **Both layers use the *sigmoid* activation function** (1) **and a weight initialization scheme known as "Xavier initialization"** showed in (2), where $\mathcal{U}[a, b]$ is the uniform distribution within the interval between a and b, n_{in} is the input dimension of a layer, and n_{out} is the output dimension of a layer [1].

$$f(x) = \frac{1}{1+e^{-x}},\ f'(x) = f(1-f) \tag{1}$$

$$W \sim \mathcal{U}\left[-\sqrt{\frac{6}{n_{in}+n_{out}}},\ \sqrt{\frac{6}{n_{in}+n_{out}}}\ \right] \tag{2}$$

The training of the model is done by back propagation per data point, repeated for multiple epochs. 1,000 data points are separated for validation, presented to the model every 10 epochs, resulting in a series of on-line training errors. Within one epoch, the remaining 3,000 points are shuffled, then used to adjust weights and biases. The error is calculated with the average J_2 loss over all data points in the validation set, shown in (3), where N is the number of data points, y_i^n and $\hat{y}_i^n$ are the original and predicted i-th pixel value on the n-th image, respectively.

$$\bar{J_2} = \frac{1}{N}\sum_{n=1}^{N} J_2 = \frac{1}{1000}\sum_{n=1}^{1000}\left(\frac{1}{2}\sum_{i=1}^{784}(y_i^n - \hat{y}_i^n)^2\right) \tag{3}$$

To further improve the training, several mechanisms are used in the training algorithm, including:

a. Using **Weight decay** for regularization by adding a weight penalty term to the loss function, as in (4) where $\boldsymbol{\lambda = 10^{-4}}$.

$$J = \bar{J_2} + \lambda \sum_{Layers L}\ \sum_{j \in Layer L}\ \sum_{i \in Layer L+1} w_{ij}^2 \tag{4}$$

b. The gradient decent has an additional term to implement **momentum**, demonstrated in (5), where $\boldsymbol{\eta = 0.05,\ \alpha = 0.8}$.

$$\Delta w_{ij}(t) = -\eta\frac{\partial J}{\partial w_{ij}} + \alpha \Delta w_{ij}(t-1) \tag{5}$$

c. Though the total repetition is 500 epochs, **an early stopping policy** is used so that it stops when the on-line training loss **does not improve more than $\mathbf{10^{-3}}$ for 50 epochs** compared to the minimum loss over the whole training session. As the validation set used for on-line testing is separate from the data used in back propagation, this ensures the model does not overfit the data.

After training, the resulting model is analyzed by feeding in the test dataset. A reconstructing autoencoder with the same hyper-parameters and policies except zero noise is also trained and tested for comparison.

1.3 Results

For the sake of discussion, both autoencoders are included in plots. Though the prediction is compared against the original image for both models, the reconstructing autoencoder has the exact same image as input, but the denoising autoencoder has noisy images as input.

1.4 Overall Performance

Fig. 1 demonstrates the performance of the autoencoders overall and on each class, where the error is calculated by (3). The on-line training error vs epochs is shown in Fig. 2.

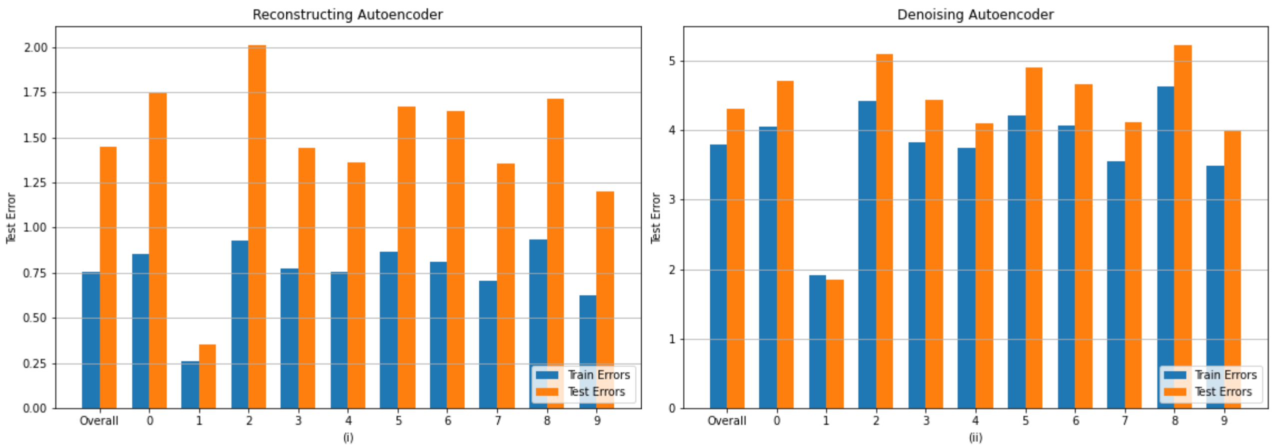

Fig. 1: Performance of the autoencoders on the training and test set

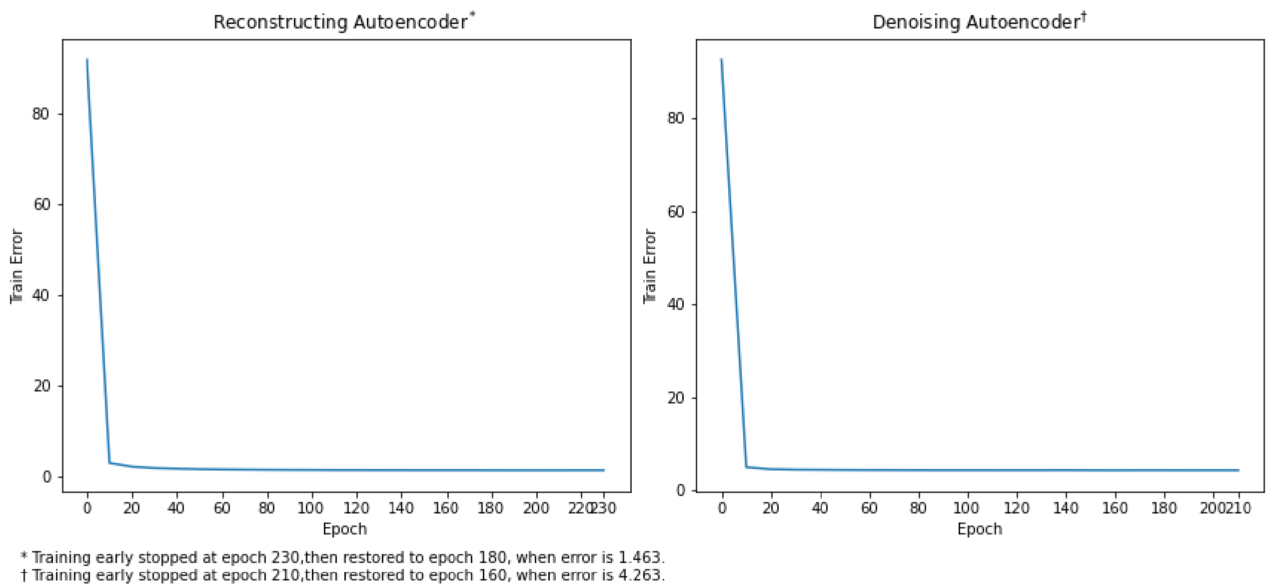

Fig. 2: On-line training error vs training epochs over time

1.5 Features

Weights of 20 neurons in the hidden layer of the two networks are illustrated by 28 $\times$ 28 images in Fig. 3, as the feature space is 784-dimensional, in accordance to 784 pixels of the original images. Though neurons are chosen randomly, the selection of neurons in both model uses the same indexes, so that neurons in the same position are compared.

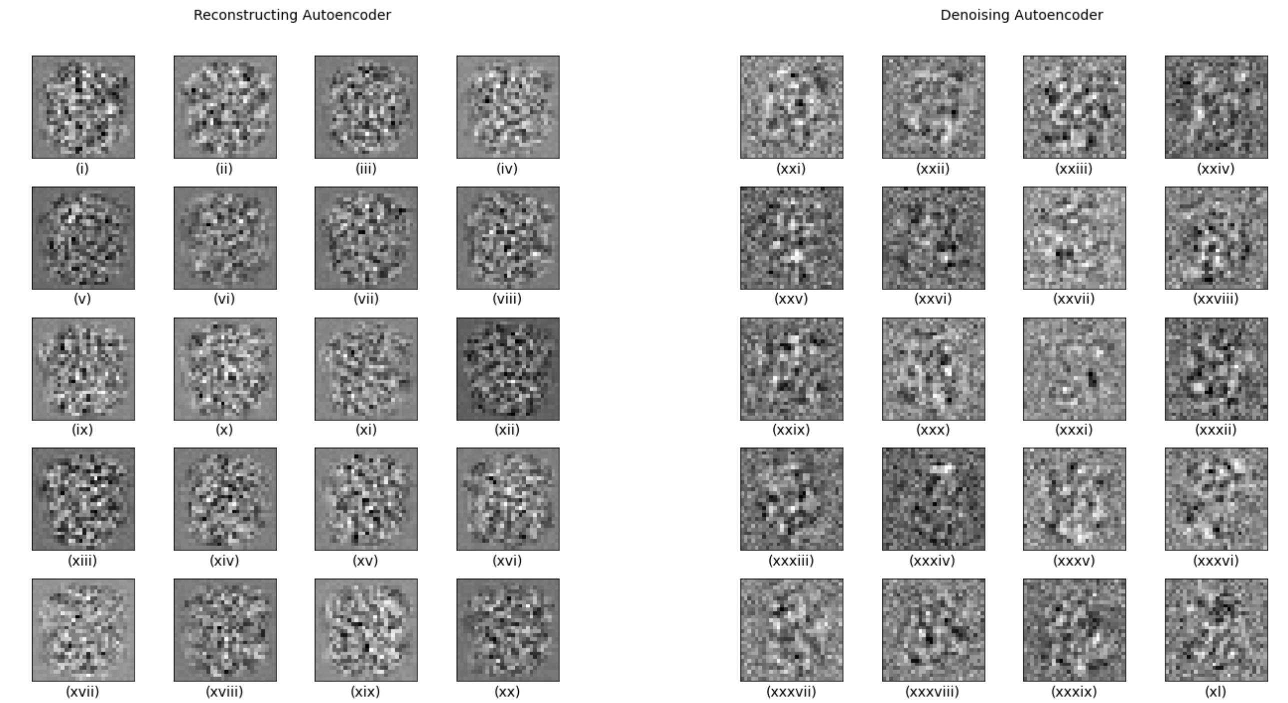

Fig. 3: Feature map of 20 neurons in the hidden layer of both autoencoders.

1.6 Sample Outputs

Fig. 4 uses 8 sets of images to visualize the two model's performance in reconstruction and noise removal.

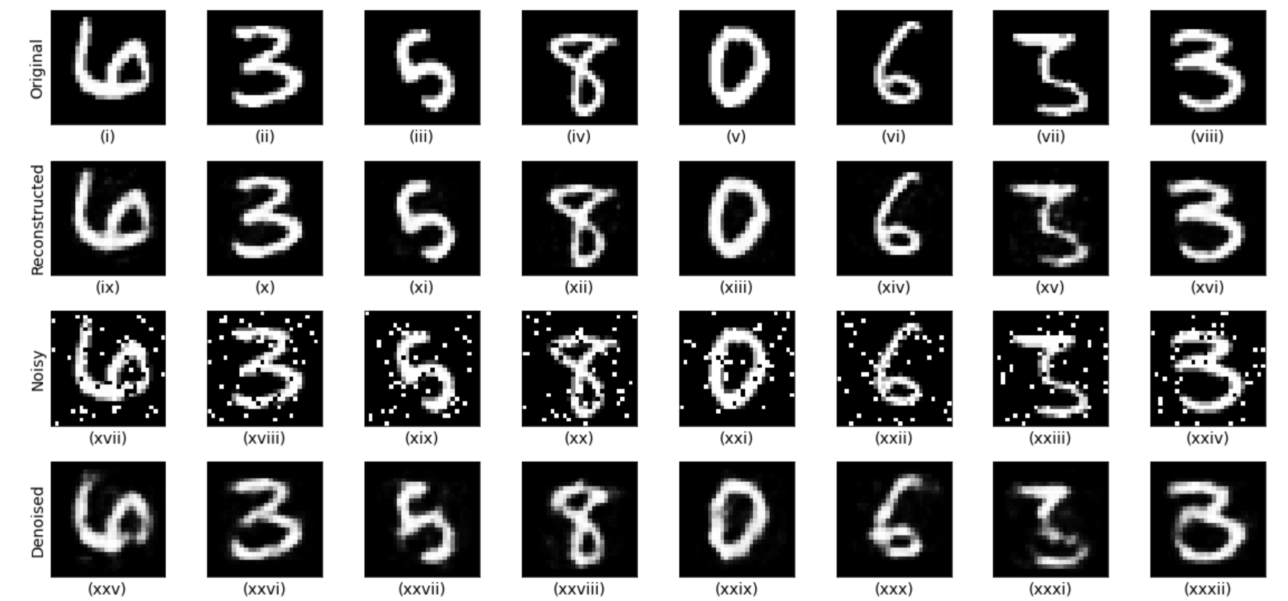

Fig. 4: Sample output of the models, (ix) - (xvi) for the reconstructing autoencoder, and (ix) - (xvi) for the denoising autoencoder

1.7 Analysis of Results

Observed from Fig. 4, both autoencoders serve their respective purposes well. Comparing the denoised images in with the original images, the "salt and pepper" noise is significantly reduced with minor errors. It is found that images with noise more scattered such as (xxiii) are denoised better than those with clustered noise such as (xxvii), due to the fact that clustered "salt" or "pepper" could be recognized as original pattern to remain.

Comparing the feature maps in their hidden layers, first difference to notice is that there are granular difference in each pixels throughout in the denoising autoencoder, which is absent in the reconstructing autoencoder. Moreover, larger clouds of dark or light pixels are in denoising autoencoder compared to the reconstructing one. It is theorized that the two differences could imply the denoising model learned a "bigger picture" in the image compared to reconstructing model, despite having larger training and testing error shown in Fig. 1 and 2.

2 Classifiers Based on Autoencoders

2.1 Problem Statement

Using the first layer from the autoencoders, two classifiers are constructed with an output layer that predicts the label of images instead. After training, the performance of the two networks shall be compared and contrasted together with the classifier from Homework #3 whose hidden layer is initialized randomly.

To tell apart the models, the report uses **denoising classifier** to address the model based on denoising autoencoder, and calls the reconstruction-based network **reconstructing classifier**. The other one from Homework #3 whose hidden layer is initialized randomly shall be called **BP classifier** as the gradient back-propagates to the first layer to adjust its weights, while only the output layer applies the weight change for the other two models.

2.2 System Description

To control the training settings, **all three classifiers** (reconstructing, denoising and BP) **are trained with exactly the same algorithm and hyper-parameters.**

As the problem states, **the two classifiers use a hidden layer with 128 neurons** from the denoising and reconstructing autoencoder, and an output layer that classifies images with an array of 10 elements, each of which represents the probability that the image can belong to a class. **This output layer is again implemented by sigmoid (1) and Xavier initialization policy (2).**

The training of the models proceeds similarly to that of the autoencoders, except the prediction is compared against the labels of the images rather than the image itself. Original version of the images is used as input for both models, since it could be unfair for the reconstructing and BP classifer. There are other minor modifications in the training session, including:

1. The back propagation uses momentum-based gradient descent (5) with $\boldsymbol{\eta = 0.01, \alpha = 0.8}$ and weight decay uses $\boldsymbol{\lambda = 10^{-5}}$.

2. Only weights of the output layer change over training. The hidden layer is treated as "read-only".

3. The loss is calculated by (1 - balanced accuracy), where the balanced accuracy is the hit rate when comparing the true class and the predicted class using "winner-take-all" strategy over the output array.

4. **Operating thresholds of 0.25 and 0.75 are used** so that output $\in [0, 0.25)$ is considered 0 when the corresponding truth is 0, and output $\in (0.75, 1]$ is considered 1 when the corresponding truth is 1.

5. Early stop strategy is again used for regularization. **The training stops when no improvement is observed for 50 epochs.**

After training, test dataset is presented to all models including reconstructing classier, denoising classifier and BP classifier.

2.3 Results

Time series of on-line training loss on both models is presented in Fig. 5. Fig. 6 shows the confusion matrices of both classifiers on train data and test data, along with the ones of BP classifier.

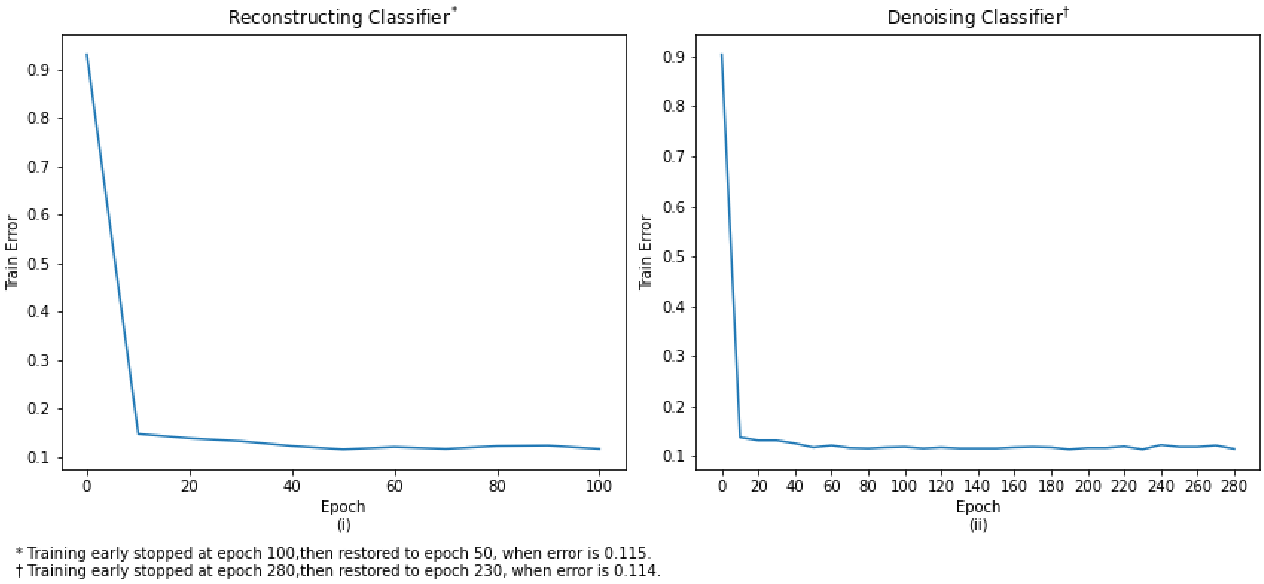

Fig. 5: Error vs epochs during training of reconstructing classifier, denoising classifier and BP classifier

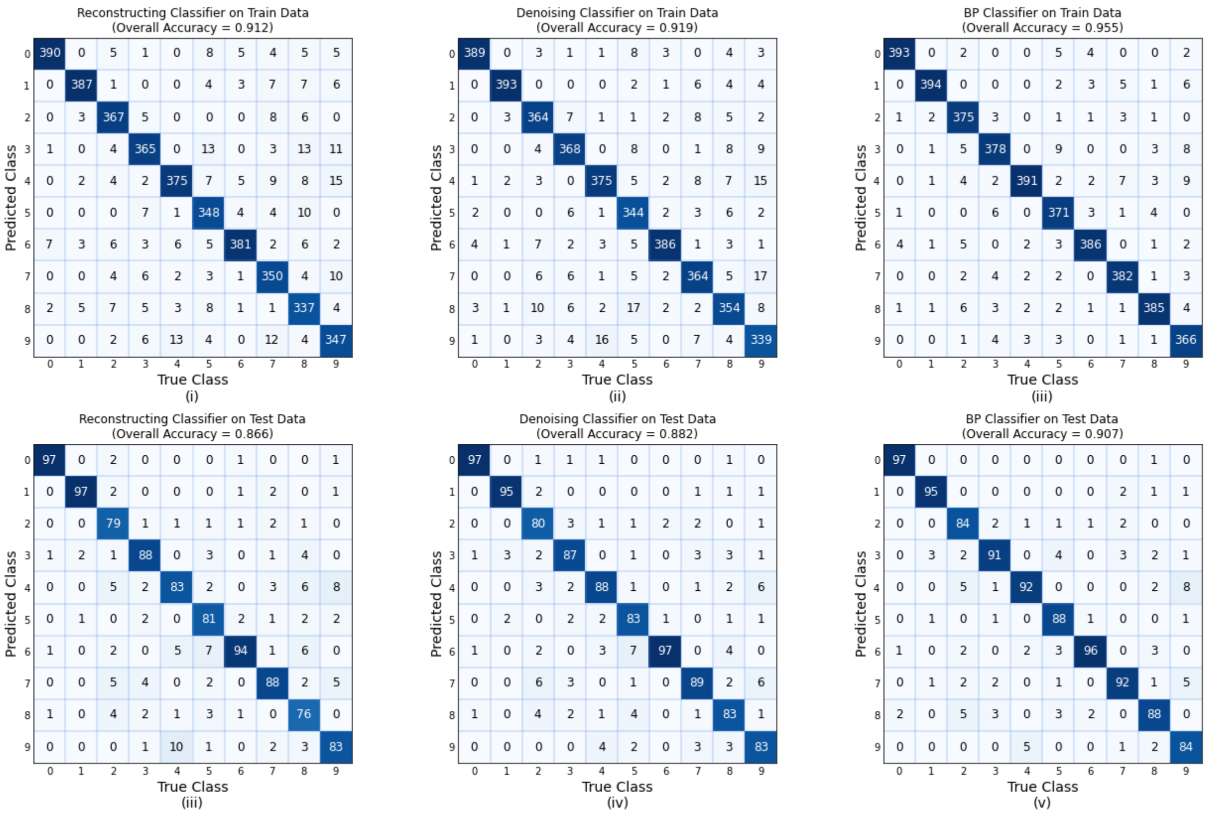

Fig. 6: Performance of reconstructing classifier, denoising classifier and BP classifier

2.4 Analysis of Results

Unfortunately, neither the reconstructing nor the denoising classifier beat the BP classifier in performance (Fig. 6). If to rank their accuracy in classifying, BP is better than denoising, and the reconstructing one is the worst. To break down the analysis, I will discuss the difference between each other separately.

The fact that reconstructing classifier performs worse than the denoising classifier on all classes is not surprising. As the hypothesis in Section 1.4 states, the denoising autoencoder generalizes the image more than the reconstructing autoencoder, and thus, each perceptron in the hidden layer is less likely to be restricted to a small amount of pixels. Comparatively, perceptrons in the reconstructing autoencoder are more likely to miss the global information, which could be important in classification problems.

The denoising classifier has the potential to perform better than the BP classifier, since the train error experienced an amount of up-and-down's in Fig. 5 (ii). Yet it did not manage to win in the end. This could be caused by the fact that it only has 10 neurons to train, while the BP classifier has 138 (128 in layer 1, 10 in layer 2). The pre-trained well-performing autoencoder layer may have established a good basis for the feature reduction, but the inability to fine tune the hidden 128 neurons makes it hard to achieve high-resolution training no matter how long the training lasts. If the pre-trained autoencoder weights are used as a weight initialization policy rather than a "read-only" layer, it is believe that the model could have even better performance and training efficiency than the BP classifier.

References

[1] X. Glorot and Y. Bengio, "Understanding the difculty of training deep feedforward neural networks," p. 8.

Appendix A Python Code: preprocess.py

```python
import numpy as np
import csv
import settings
import json
import pandas as pd
import matplotlib.pyplot as plt

def prepare_img(img_a):
    img_a = 1 - np.array(img_a).flatten()
    img_a = img_a / np.linalg.norm(img_a)
    return np.reshape(img_a, (-1, int(len(img_a)**0.5)), order='F')

def get_rand_list(length):
    return np.random.choice(length,length,replace=False).astype(int)

def prepare_data():
    x_db = []
    with open(str(settings.X_FILE)) as csv_file:
        csv_reader = csv.reader(csv_file, delimiter='\t')
        for row in csv_reader:
            x_db.append([float(x) for x in row])


    y_db = []
    with open(str(settings.Y_FILE)) as csv_file:
        csv_reader = csv.reader(csv_file, delimiter='\t')
        for row in csv_reader:
            y_db.append(int(row[0]))

    print('Distribution of original dataset:',np.bincount(y_db))

    train_i, test_i = stratify_split(y_db,
                                     settings.SIZES['train']/settings.SIZES['y'])

    train_db = {'x':[x_db[i] for i in train_i],
                'y':[y_db[i] for i in train_i]}

    test_db = {'x':[x_db[i] for i in test_i],
            'y':[y_db[i] for i in test_i]}


    print('Distribution of train dataset:',np.bincount(train_db['y']))
    print('Distribution of test dataset:',np.bincount(test_db['y']))

    test_db['y'] = np.eye(settings.SIZES['classes'])[test_db['y']].tolist()
    train_db['y'] = np.eye(settings.SIZES['classes'])[train_db['y']].tolist()

    with open(str(settings.TRAIN_FILE),'w') as f:
        json.dump(train_db, f)

    print("Saved train data in", settings.TRAIN_FILE)

    with open(str(settings.TEST_FILE),'w') as f:
        json.dump(test_db, f)

    print("Saved test data in", settings.TEST_FILE)

def stratify_split(y, ratio):
    if len(np.array(y).shape) > 1: # collapse for one hot
        y = np.argmax(y,axis=1)
    df = pd.DataFrame(y).groupby(0) # Sort data by class
    indxs = [] # buffer for indexes
    for _,g in df:
        indxs.append(g.index.to_numpy()) # indexes of each class take a row
    indxs = np.array(indxs)
    p1_indx = indxs[:, :int(indxs.shape[1]*ratio)].flatten() # partition 1
    np.random.shuffle(p1_indx) # mix index
    p2_indx = indxs[:, int(indxs.shape[1]*ratio):].flatten() # partition 2
    np.random.shuffle(p2_indx) # mix index
    return p1_indx, p2_indx


def get_test():
    with open(str(settings.TEST_FILE),'r') as f:
        return json.load(f)

def get_train():
    with open(str(settings.TRAIN_FILE),'r') as f:
        return json.load(f)

def add_noise(img, noise_type):
    img = np.array(img)
    noise_type = noise_type.lower()
    if noise_type == "gaussian":
        mu = 0.5
        sigma = np.sqrt(0.001)
        return img + np.random.normal(mu,sigma, len(img))
    elif noise_type == "s&p":
        density = 0.5
        svp = 1/8
        rand_i = get_rand_list(len(img))[:int(len(img)*density)]
        salt_i = rand_i[:int(len(rand_i)*svp/(svp+1.0))]
        pepper_i = rand_i[int(len(rand_i)/(svp+1.0)):]
        img[salt_i] = 0.0
        img[pepper_i] = 1.0
        return img
    elif noise_type == "poisson":
        vals = len(np.unique(img))
        vals = 2 ** np.ceil(np.log2(vals))
        return np.random.poisson(img * vals) / float(vals)
    elif noise_type == "speckle":
        return img + img*np.random.uniform(0,1)

def int_to_roman(num):
```

```python
    result = ''
    mapping = {1000:'M', 900:'CM', 500:'D', 400:'CD', 100:'C', 90:'XC', 50:'L', 40:'XL', 10:'X', 9:'IX', 5:'V', 4:'IV', 1:'I'}

    while num != 0:
        for k, v in mapping.items():
            if num >= k:
                dividend = int(num/k)
                num %= k
                result += dividend*v
    return result.lower()

if __name__ == '__main__':
    # prepare_data()

    x = get_train()['x']
    i = int(get_rand_list(len(x))[0])
    fig, ax = plt.subplots(1,2,figsize=(8,6))
    ax[0].imshow(prepare_img(x[i]),cmap='binary')
    ax[1].imshow(prepare_img(add_noise(x[i],"s&p")), cmap='binary')

    # print(int_to_roman(15))
    pass
```

Appendix B Python Code: nn.py

```
import numpy as np
from preprocess import get_rand_list, stratify_split
import json
from tqdm import trange


class DenseLayer(object): # fully connected layer
    def __init__(self, n_input, n_neurons, activation=None, trainable=True,
                 weights=None):
        """
        Initialize a fully-connected layer

        Parameters
        ----------
        n_input : uint
            number of input nodes.
        n_neurons : uint
            number of neurons / output nodes.
        activation : str, optional
            activation function name. The default is None.
        weights : np array, optional
            matrix for weights. The default is None.

        Returns
        -------
        None.

        """

        if weights is None:
            a = np.sqrt(6/(n_input+n_neurons))
            self.weights = np.random.uniform(low=-a, high=+a, size=(n_input+1, n_neurons)) #Xavier initialization
        else:
            weights = np.array(weights)
            if weights.shape == (n_input+1, n_neurons):
                self.weights = weights
            else:
                raise ValueError("Given weights does not match given dimensions")
        self.trainable = trainable

        self.last_dweights = np.zeros((n_input+1, n_neurons))

        self.activation = activation
        self.last_activation = None
        self.error = None
        self.delta = None

    def set_trainable(self, trainable):
        """
        Configure if the layer is trainable

        Parameters
        ----------
        trainable : bool
            whether the layer is trainable.

        Returns
        -------
        None.

        """
        self.trainable = trainable

    def call(self,x):
        """
        Calculate the output given input

        Parameters
        ----------
        x : np array or list
            array or list of input to the layer.

        Returns
        -------
        np array
            array of output from the layer.

        """
        x = np.append([1],x)
        s = x @ self.weights
        self.last_activation = self._apply_activation(s)
        return self.last_activation


    def _apply_activation(self, s):
        """
        calcualte activated output

        Parameters
        ----------
        s : np array
            array of the inner product between input and weights.

        Returns
        -------
        np array
            activated output.

        """
        if self.activation == 'relu':
            return np.maximum(s,0)
        elif self.activation == 'tanh':
            return np.tanh(s)
        elif self.activation == 'sigmoid':
```

```
            return 1.0 / (1.0 + np.exp(-s))
        else:
            return s # if no or unkown activation, f = s

    def apply_activation_derivative(self, s):
        """
        calculate the derivative of activation function

        Parameters
        ----------
        s : np array
            array of the inner product between input and weights.

        Returns
        -------
        np array
            calcualted output after activation dirivative.

        """
        if self.activation == 'relu':
            grad = np.copy(s)
            grad[s>0] = 1.0
            grad[s<=0] = 0.0
            return grad
        elif self.activation == 'tanh':
            return 1 - s ** 2
        elif self.activation == 'sigmoid':
            return s * (1-s)
        else:
            return np.ones_like(s) # if no or unkown activation, f' = 1

    def update_weights(self, dweights):
        """
        Used for gradient descent to change weight values

        Parameters
        ----------
        dweights : TYPE
            DESCRIPTION.

        Returns
        -------
        None.

        """
        if self.trainable:
            self.weights += dweights
            self.last_dweights = dweights

    def io(self):
        """
        Get the input and output number of the layer

        Returns
        -------
        int
            input dimension.
        TYPE
            output dimension.

        """
        return (self.weights.shape[0]-1, self.weights.shape[1])










class NeuralNetwork(object): # neural network model
    def __init__(self):
        """
        Initialize model with a buffer for layers

        Returns
        -------
        None.

        """
        self._layers = []
        self.learning_rate = None
        self.momentum = None
        self.train_errors = []

    def add_layer(self,layer):
        """
        Append a layer at the end

        Parameters
        ----------
        layer : XXLayer type
            a nn layer.

        Returns
        -------
        None.

        """
        self._layers.append(layer)


    def _feed_forward(self,x):
        """
        Calculate output from an input
```

```

        Parameters
        ----------
        x : np array
            input vector into the network.

        Returns
        -------
        x : np array
            output vector out of the network.

        """
        for layer in self._layers:
            x = layer.call(x)
        return x

    def _back_prop(self, x, y, learning_rate, momentum=0.0, threshold=0.0, weight_decay=0.0):
        """
        Implement back propagation with momentum gradient descent and
        thresholded output

        Parameters
        ----------
        x : np array
            input vector to the network.
        y : np array
            target output vector for the network.
        learning_rate : float
            learning rate.
        momentum : float, optional
            alpha value to control the momentum gradient descent. The default is 0.
        threshold : float, optional
            threshold window to be considered 0 or 1, detail see self.train(). The default is 0.

        Returns
        -------
        None.

        """
        output = self._feed_forward(x)
        # Calculate gradients
        for i in reversed(range(len(self._layers))): # start from the last layer
            layer = self._layers[i]
            if i == len(self._layers) -1: # for output layer
                raw_error = y - output
                raw_error = [0 if np.abs(e)<threshold else e for e in raw_error]# implement thresholding
                layer.error = raw_error
                layer.delta = layer.apply_activation_derivative(output) * layer.error
            else: # for hidden layers
                next_layer = self._layers[i+1]
                layer.error = next_layer.weights[1:,:] @ next_layer.delta
                layer.delta =  layer.apply_activation_derivative(layer.last_activation) * layer.error
        # Update weights
        for i,layer in enumerate(self._layers):
            pre_synaptic = (x if i == 0 else self._layers[i-1].last_activation)
            pre_synaptic = np.append([1], pre_synaptic)
            pre_synaptic = np.atleast_2d(pre_synaptic)
            delta_weights = pre_synaptic.T @ np.atleast_2d(layer.delta) * learning_rate # basic gradient descent
            delta_weights -= 2*weight_decay*learning_rate*layer.weights # implement weight decay
            delta_weights += momentum * layer.last_dweights # implement momentum
            layer.update_weights(delta_weights)


    def train(self, X_train, Y_train, learning_rate, max_epochs, classify=False,
              momentum=0, threshold=0, validation_ratio=0.0, weight_decay=0.0,
              earlystop=None):
        """
        Train the network with given input, output, and hyper-parameters

        Parameters
        ----------
        X_train : list or np array
            a batch of input vector to the network.
        Y_train : list or np array
            a batch of target output for the network.
        learning_rate : float
            specifies the learning rate of gradient descent.
        max_epochs : int
            specifies the max amount of epochs to train.
        momentum : float, optional
            specifies the alpha value for gradient descent. The default is 0.
        threshold : float, optional
            specified the threshold windows for the output to consider 0 or 1.
            Output is 0 if 0<= output < threshold; output is 1 if
            1-threshold < output <=1.
            The default is 0.
        stochastic_ratio : float, optional
            specifies how much of the input batch is selected.
            The default is 1.0.
        earlystop : set of 2 elements, optional
            specifies earlystop. [0] represents the max value for the output
            to be the 'same'. [1] represents the patience. The default is None.

        Returns
        -------
        errors : np array
            errors every 10 epochs of training.

        """
        if earlystop is None:
            earlystop = (0, max_epochs//10)


        X_train = np.array(X_train)
        Y_train = np.array(Y_train)

        if X_train.shape[1] != self._layers[0].weights.shape[0]-1:
```

```
            raise ValueError("Input data does not match layer dimension")
        if Y_train.shape[1] != self._layers[-1].weights.shape[1]:
            raise ValueError("Output data does not match layer dimension")

        self.learning_rate = learning_rate
        self.momentum = momentum
        self.weight_decay = weight_decay

        if classify:
            validation_i, realtrain_i = stratify_split(Y_train, validation_ratio)
        else:
            shuffle_i =  get_rand_list(len(X_train))
            realtrain_i = shuffle_i[int(len(X_train)*validation_ratio):]
            validation_i = shuffle_i[:int(len(X_train)*validation_ratio)]

        X_vali = [X_train[i] for i in validation_i]
        Y_vali = [Y_train[i] for i in validation_i]
        good_layers = self._layers
        errors = []
        earlystop_counter = 0

        self.info()

        for epoch in trange(max_epochs+1, ncols=75, unit='epoch'):

            if epoch % 10 == 0:

                if classify:
                    error = self.classify_test(X_vali, Y_vali)
                else:
                    error = self.raw_test(X_vali, Y_vali)
                errors.append(error)


                if epoch == 0:
                    print("\nLoss = {} at epoch {}".format(errors[-1], epoch))
                    continue

                if (np.min(errors[:-1]) - errors[-1]) < earlystop[0]:
                    earlystop_counter += 1
                    if earlystop_counter == earlystop[1]:
                        print("\nEarly stop triggered at epoch {}, restored to epoch {}"
                              .format(epoch, epoch-earlystop[1]*10))
                        self._layers = good_layers
                        self.train_errors = np.array(errors)
                        return self.train_errors
                else:
                    good_layers = self._layers
                    earlystop_counter = 0

                print("\nLoss = {} at epoch {}, training stops in {} epochs".format(errors[-1], epoch, (earlystop[1]-
    earlystop_counter)*10))

            np.random.shuffle(realtrain_i)
            for i in realtrain_i:
                self._back_prop(X_train[i], Y_train[i], learning_rate, momentum, threshold, weight_decay)


        self.train_errors = np.array(errors)
        return np.array(errors)

    def raw_test(self, X_test, Y_test):
        """
        Test the model with given data, calculate J2 loss

        Parameters
        ----------
        X_test : 2D list or np array
            input data to the network.
        Y_test : 2D list or np array
            true output data.

        Returns
        -------
        float
            average J2 loss.

        """
        X_test = np.array(X_test)
        Y_test = np.array(Y_test)
        error = 0
        for i in range(len(X_test)):
            pred = self._feed_forward(X_test[i])
            error += np.sum((pred-Y_test[i])**2)
        return 0.5*error/len(X_test)


    def classify_test(self, X_test, Y_test):
        """
        Test the network with given input, output and accuracy

        Parameters
        ----------
        X_test : list or np array
            input vector to the network.
        Y_test : list or np array
            ground truth for the testing.

        Returns
        -------
        float
            test accuracy

        """
        errors = []
        X_test = np.array(X_test)
        Y_test = np.array(Y_test)
```

```python
        for i in range(len(X_test)):
            pred = self._feed_forward(X_test[i])
            pred = np.argmax(pred)
            truth = np.argmax(Y_test[i])
            errors.append(pred==truth)
        return 1-np.sum(errors)/len(errors)

    def get_cm(self, X_test, Y_test):
        """
        Give the confusion matrix for classification problems

        Parameters
        ----------
        X_test : list or np array
            input vector to the network.
        Y_test : list or np array
            ground truth for the testing.

        Returns
        -------
        cm : np array
            confusion matrix.

        """
        X_test = np.array(X_test)
        Y_test = np.array(Y_test)
        n_classes = Y_test.shape[1]
        cm = np.zeros((n_classes, n_classes))
        for i in range(len(X_test)):
            pred = self._feed_forward(X_test[i])
            max_pred = np.max(pred)
            pred_bin = np.atleast_2d([p==max_pred for p in pred]).T
            truth = np.atleast_2d(Y_test[i])
            cm += pred_bin @ truth
        return cm

    def save(self, file_name):
        """
        Save the model in a json file
        First line of json is meta data
        Following line includes layer info

        Parameters
        ----------
        file_name : str
            string to save data into.

        Returns
        -------
        None.

        """
        if type(file_name) is not str:
            file_name = str(file_name)

        with open(file_name,'w') as f:
            meta_dict = {}
            meta_dict['learning_rate'] = self.learning_rate
            meta_dict['momentum'] = self.momentum
            meta_dict['weight_decay'] = self.weight_decay
            meta_dict['train_errors'] = self.train_errors.tolist()
            json.dump(meta_dict, f)
            f.write("\n")
            for layer in self._layers:
                layer_dict = {}
                layer_dict['n_input'], layer_dict['n_neurons'] = layer.io()
                layer_dict['activation'] = layer.activation
                layer_dict['trainable'] = layer.trainable
                layer_dict['weights'] = layer.weights.tolist()
                json.dump(layer_dict, f)
                f.write("\n")

    def load(self, file_name):
        """
        Loads the json file for a model

        Parameters
        ----------
        file_name : TYPE
            DESCRIPTION.

        Returns
        -------
        None.

        """
        if type(file_name) is not str:
            file_name = str(file_name)

        with open(file_name,'r') as f:
            for i, line in enumerate(f):
                if i == 0:
                    meta = json.loads(line)
                    self.learning_rate = meta['learning_rate']
                    self.momentum = meta['momentum']
                    self.weight_decay = meta['weight_decay']
                    self.train_errors = np.array(meta['train_errors'])
                else:
                    layer = json.loads(line)
                    self.add_layer(DenseLayer(n_input=layer['n_input'],
                                              n_neurons=layer['n_neurons'],
                                              activation=layer['activation'],
                                              weights=layer['weights'],
                                              trainable=layer['trainable']))
    def info(self):
        """
        Print the model information
```

```

        Returns
        -------
        None.

        """
        print("{} layer neural network".format(len(self._layers)))
        print('Learning rate: {}\nMomentum: {}\nWeight Decay: {}'.format(self.learning_rate, self.momentum, self.weight_decay))
        for i,layer in enumerate(self._layers):
            print("Layer {} = input: {}, output: {}, activation: {}, trainable: {}".format(i, layer.io()[0], layer.io()[1], layer.activation, layer.trainable))

    def layers(self, n=None):
        """
        Get layers of a model

        Parameters
        ----------
        n : int
            index of the layer starting from 0.

        Returns
        -------
        Layer object
            the n-th layer of the model.

        """
        if n is None:
            return self._layers
        else:
            return self._layers[n]

    def predict(self, X):
        """
        Make prediction from the given input

        Parameters
        ----------
        X : 2D list or np array
            Input data to the network.

        Returns
        -------
        pred : list
            predicted output.

        """
        pred = []
        for x in X:
            for layer in self._layers:
                x = layer.call(x)
            pred.append(x)
        return pred

    def pop_layer(self):
        self._layers.pop()
```

Appendix C Python Code: p1_train.py

```python
from preprocess import get_train, get_test, add_noise
from settings import SIZES, H4P1_NN, HIDDEN_NEURONS, MAX_EPOCHS, VALI_R, NOISE, PATIENCE
from nn import NeuralNetwork, DenseLayer

train_db = get_train()
test_db = get_test()
autoenc = NeuralNetwork()
autoenc.add_layer(DenseLayer(n_input=SIZES['x'][1], n_neurons=HIDDEN_NEURONS,
                             activation='sigmoid'))
autoenc.add_layer(DenseLayer(n_input=HIDDEN_NEURONS, n_neurons=SIZES['x'][1],
                             activation='sigmoid'))
noisy_x = [add_noise(x,NOISE) for x in train_db['x']]
autoenc.train(noisy_x, train_db['x'], max_epochs=MAX_EPOCHS,
              classify=False,
              validation_ratio=VALI_R, earlystop=(1E-3,PATIENCE),
              learning_rate = 0.01,
              momentum=0.8,
               weight_decay=1E-4,
              )
autoenc.save(H4P1_NN)
```

Appendix D Python Code: p1_test.py

```python
from preprocess import get_train, get_test, get_rand_list, prepare_img, add_noise, int_to_roman
from settings import CLASSES, SIZES, PATIENCE, NOISE
from settings import H4P1_NN, H4P1_TRAIN_PLOT, H4P1_TEST_PLOT, H4P1_FEATURE_MAP, H4P1_OUTPUT_MAP, H3P2_NN, HIDDEN_NEURONS
from nn import NeuralNetwork
import numpy as np
import matplotlib.pyplot as plt

# Load the network
train_db = get_train()
test_db = get_test()
autoenc_noise = NeuralNetwork()
autoenc_noise.load(H4P1_NN)
autoenc_clean = NeuralNetwork()
autoenc_clean.load(H3P2_NN)

# Plot training error vs epoch
train_errors = autoenc_clean.train_errors, autoenc_noise.train_errors
epochs = 10*np.arange(len(train_errors[0])), 10*np.arange(len(train_errors[1]))
fig1, ax1 = plt.subplots(1,2,figsize=(12,6))
for i in range(2):
    ax1[i].plot(epochs[i], train_errors[i])
    ax1[i].set_xticks(np.append(np.arange(0,epochs[i][-1],20),epochs[i][-1]))
    ax1[i].set_xlabel("Epoch")
    ax1[i].set_ylabel("Train Error")
ax1[0].set_title("Reconstructing Autoencoder$^*$")
ax1[1].set_title("Denoising Autoencoder$^  $ ")
fig1.text(0.02, 0.01, '* Training early stopped at epoch {},'
          'then restored to epoch {}, when error is {:.3f}.\n'
          '    Training early stopped at epoch {},'
          'then restored to epoch {}, when error is {:.3f}.\n'
          .format(epochs[0][-1], epochs[0][-1-PATIENCE], train_errors[0][-1-PATIENCE],
                  epochs[1][-1], epochs[1][-1-PATIENCE], train_errors[1][-1-PATIENCE],
                  ), ha='left')
fig1.tight_layout(rect=[0, 0.08, 1, 0.95])
fig1.savefig(H4P1_TRAIN_PLOT)

# Plot training errors
fig2, ax2 = plt.subplots(1,2, figsize=(16,6))

for n in range(2):
    if n == 0:
        autoenc = autoenc_clean
        ax2[n].set_title("Reconstructing Autoencoder")
    else:
        autoenc = autoenc_noise
        ax2[n].set_title("Denoising Autoencoder")
    test_errors = [[] for _ in CLASSES]
    train_errors = [[] for _ in CLASSES]
    for i,x in enumerate(train_db['x']):
        c = np.argmax(train_db['y'][i])
        if n == 0:
            train_errors[c].append(autoenc.raw_test([x],[x]))
        else:
            train_errors[c].append(autoenc.raw_test([add_noise(x,NOISE)],[x]))
    for i,x in enumerate(test_db['x']):
        c = np.argmax(test_db['y'][i])
        if n==0:
            test_errors[c].append(autoenc.raw_test([x],[x]))
        else:
            test_errors[c].append(autoenc.raw_test([add_noise(x,NOISE)],[x]))
    test_errors = np.mean(test_errors,axis=1)
    test_errors = np.insert(test_errors, 0, np.mean(test_errors))
    train_errors = np.mean(train_errors,axis=1)
    train_errors = np.insert(train_errors, 0, np.mean(train_errors))
    width = 0.35
    ticks = [str(c) for c in CLASSES]
    ticks.insert(0,'Overall')
    ax2[n].bar(np.arange(len(ticks)) - width/2, train_errors, width, label='Train Errors')
    ax2[n].bar(np.arange(len(ticks)) + width/2, test_errors, width, label='Test Errors')
    ax2[n].set_xticks(np.arange(len(ticks)))
    ax2[n].set_xticklabels(ticks)
    ax2[n].set_ylabel('Test Error')
    ax2[n].set_xlabel('('+int_to_roman(n+1)+')')
    ax2[n].legend(loc='lower right')
    ax2[n].grid(axis='y')

fig2.tight_layout(rect=[0, 0, 1, 0.95])
fig2.savefig(H4P1_TEST_PLOT)

# Plot feature maps
fig3, ax3 = plt.subplots(5,9, figsize=(18,10))
neuron_i = get_rand_list(HIDDEN_NEURONS)[:20]
features = [0]*2
for i,ni in enumerate(neuron_i):
    features[0] = autoenc_clean.layers(0).weights[:,ni][1:]
    features[1] = autoenc_noise.layers(0).weights[:,ni][1:]
    ax3[i//4][4].axis('off')
    for j in range(2):
        ax3[i//4][i%4+5*j].imshow(prepare_img(features[j]), cmap='binary')
        ax3[i//4][i%4+5*j].set_xticks([])
        ax3[i//4][i%4+5*j].set_yticks([])
        ax3[i//4][i%4+5*j].set_xlabel('('+int_to_roman(i+20*j+1)+')', fontsize=14)

fig3.suptitle('Reconstructing Autoencoder{}Denoising Autoencoder'.format(' '*123), fontsize=14)
fig3.tight_layout(rect=[0, 0, 1, 0.93])
fig3.savefig(H4P1_FEATURE_MAP)

# Plot sample output

img_i = get_rand_list(SIZES['test'])[:8]
fig4, ax4 = plt.subplots(4,8, figsize=(16,8))
for i, ii in enumerate(img_i):
    clean = test_db['x'][ii]
    ax4[0][i].imshow(prepare_img(clean), cmap='binary')
```

```
    ax4[0][i].set_xticks([])
    ax4[0][i].set_yticks([])
    ax4[0][i].set_xlabel('('+int_to_roman(i+0*8+1)+')', fontsize=14)

    reconstructed = autoenc_clean.predict([clean])
    ax4[1][i].imshow(prepare_img(reconstructed), cmap='binary')
    ax4[1][i].set_xticks([])
    ax4[1][i].set_yticks([])
    ax4[1][i].set_xlabel('('+int_to_roman(i+1*8+1)+')', fontsize=14)

    noisy = add_noise(clean, NOISE)
    ax4[2][i].imshow(prepare_img(noisy), cmap='binary')
    ax4[2][i].set_xticks([])
    ax4[2][i].set_yticks([])
    ax4[2][i].set_xlabel('('+int_to_roman(i+2*8+1)+')', fontsize=14)

    denoised = autoenc_noise.predict([noisy])
    ax4[3][i].imshow(prepare_img(denoised), cmap='binary')
    ax4[3][i].set_xticks([])
    ax4[3][i].set_yticks([])
    ax4[3][i].set_xlabel('('+int_to_roman(i+3*8+1)+')', fontsize=14)

ax4[0][0].set_ylabel("Original", fontsize=14)
ax4[1][0].set_ylabel("Reconstructed", fontsize=14)
ax4[2][0].set_ylabel("Noisy", fontsize=14)
ax4[3][0].set_ylabel("Denoised", fontsize=14)
fig4.tight_layout(rect=[0, 0, 1, 0.95])
fig4.savefig(H4P1_OUTPUT_MAP)

plt.show()
plt.close('all')
```

Appendix E Python Code: p2_train.py

```python
from preprocess import get_train
from settings import SIZES, H3P2_NN, HIDDEN_NEURONS, MAX_EPOCHS, VALI_R, H4P1_NN, H4P2C1_NN, H4P2C2_NN, PATIENCE
from nn import NeuralNetwork, DenseLayer


train_db = get_train()

nn_clean = NeuralNetwork()
nn_clean.load(H3P2_NN)
nn_clean.pop_layer()
for layer in nn_clean.layers():
    layer.set_trainable(False)
nn_clean.add_layer(DenseLayer(n_input=HIDDEN_NEURONS, n_neurons=SIZES['classes'],
                              activation='sigmoid'))

nn_noise = NeuralNetwork()
nn_noise.load(H4P1_NN)
nn_noise.pop_layer()
for layer in nn_noise.layers():
    layer.set_trainable(False)
nn_noise.add_layer(DenseLayer(n_input=HIDDEN_NEURONS, n_neurons=SIZES['classes'],
                             activation='sigmoid'))


nn_clean.train(train_db['x'], train_db['y'], max_epochs=MAX_EPOCHS,
                classify=True, threshold=0.25,
                validation_ratio=VALI_R, earlystop=(0,PATIENCE),
                learning_rate = 0.002,
                momentum=0.8,
                weight_decay=1E-4,
                )
nn_clean.save(H4P2C1_NN)


nn_noise.train(train_db['x'], train_db['y'], max_epochs=MAX_EPOCHS,
               classify=True, threshold=0.25,
               validation_ratio=VALI_R, earlystop=(0,PATIENCE),
               learning_rate = 0.002,
               momentum=0.8,
                weight_decay=1E-4,
               )
nn_noise.save(H4P2C2_NN)
```

Appendix F Python Code: p2_test.py

```python
from preprocess import get_train, get_test, int_to_roman
from settings import CLASSES, H4P2C1_NN, H4P2C2_NN, H4P2_CM_PLOT, H4P2_TRAIN_PLOT, PATIENCE, H3P1_NN
from nn import NeuralNetwork
import numpy as np
import matplotlib.pyplot as plt

# Load the autoenc_clean
train_db = get_train()
test_db = get_test()
autoenc_clean = NeuralNetwork()
autoenc_clean.load(H4P2C1_NN)

autoenc_noise = NeuralNetwork()
autoenc_noise.load(H4P2C2_NN)

# Plot training error vs epoch
train_errors = autoenc_clean.train_errors, autoenc_noise.train_errors
epochs = 10*np.arange(len(train_errors[0])), 10*np.arange(len(train_errors[1]))
fig1, ax1 = plt.subplots(1,2,figsize=(12,6))
for i in range(2):
    ax1[i].plot(epochs[i], train_errors[i])
    ax1[i].set_xticks(np.append(np.arange(0,epochs[i][-1],20),epochs[i][-1]))
    ax1[i].set_xlabel("Epoch\n({})".format(int_to_roman(i+1)))
    ax1[i].set_ylabel("Train Error")
ax1[0].set_title("Reconstructing Classifier$^*$")
ax1[1].set_title("Denoising Classifier$^  $  ")
fig1.text(0.02, 0.01, '* Training early stopped at epoch {},'
          'then restored to epoch {}, when error is {:.3f}.\n'
          '    Training early stopped at epoch {},'
          'then restored to epoch {}, when error is {:.3f}.\n'
          .format(epochs[0][-1], epochs[0][-1-PATIENCE], train_errors[0][-1-PATIENCE],
                  epochs[1][-1], epochs[1][-1-PATIENCE], train_errors[1][-1-PATIENCE],
                  ), ha='left')
fig1.tight_layout(rect=[0, 0.08, 1, 0.95])
fig1.savefig(H4P2_TRAIN_PLOT)

# Plot confusion metrix
classifier = NeuralNetwork()
classifier.load(H3P1_NN)

cm = [[0 for _ in range(3)] for _ in range(2)]
cm[0][0] = autoenc_clean.get_cm(train_db['x'], train_db['y'])
cm[1][0] = autoenc_clean.get_cm(test_db['x'], test_db['y'])
cm[0][1] = autoenc_noise.get_cm(train_db['x'], train_db['y'])
cm[1][1] = autoenc_noise.get_cm(test_db['x'], test_db['y'])
cm[0][2] = classifier.get_cm(train_db['x'], train_db['y'])
cm[1][2] = classifier.get_cm(test_db['x'], test_db['y'])

errors = [[0 for _ in range(3)] for _ in range(2)]
errors[0][0] = autoenc_clean.classify_test(train_db['x'], train_db['y'])
errors[1][0] = autoenc_clean.classify_test(test_db['x'], test_db['y'])
errors[0][1] = autoenc_noise.classify_test(train_db['x'], train_db['y'])
errors[1][1] = autoenc_noise.classify_test(test_db['x'], test_db['y'])
errors[0][2] = classifier.classify_test(train_db['x'], train_db['y'])
errors[1][2] = classifier.classify_test(test_db['x'], test_db['y'])

fig2, ax2 = plt.subplots(2,3, figsize=(18,12))
for m in range(2):
    for n in range(3):
        ax2[m,n].imshow(cm[m][n], cmap='Blues')
        ax2[m,n].set_xticks(CLASSES)
        ax2[m,n].set_yticks(CLASSES)
        ax2[m,n].set_xticklabels(CLASSES)
        ax2[m,n].set_yticklabels(CLASSES)
        ax2[m,n].tick_params(axis=u'both', which=u'both',length=0)
        for i in range(len(CLASSES)):
            for j in range(len(CLASSES)):
                c = 'w' if cm[m][n][i,j]>=50 else 'k'
                text = ax2[m,n].text(j, i, int(cm[m][n][i, j]), ha="center", va="center", color=c, fontsize=12)
        ax2[m,n].set_xlabel("True Class\n({})".format(int_to_roman(n+m*2+1)), fontsize=14)
        ax2[m,n].set_ylabel("Predicted Class", fontsize=14)
        for num in CLASSES:
            ax2[m,n].axvline(num-0.5, c='cornflowerblue', lw=1.5, alpha=0.3)
            ax2[m,n].axhline(num-0.5, c='cornflowerblue', lw=1.5, alpha=0.3)

ax2[0,0].set_title("Reconstructing Classifier on Train Data\n(Overall Accuracy = {:.3f})".format(1-errors[0][0]))
ax2[1,0].set_title("Reconstructing Classifier on Test Data\n(Overall Accuracy = {:.3f})".format(1-errors[1][0]))
ax2[0,1].set_title("Denoising Classifier on Train Data\n(Overall Accuracy = {:.3f})".format(1-errors[0][1]))
ax2[1,1].set_title("Denoising Classifier on Test Data\n(Overall Accuracy = {:.3f})".format(1-errors[1][1]))
ax2[0,2].set_title("BP Classifier on Train Data\n(Overall Accuracy = {:.3f})".format(1-errors[0][2]))
ax2[1,2].set_title("BP Classifier on Test Data\n(Overall Accuracy = {:.3f})".format(1-errors[1][2]))
fig2.tight_layout(rect=[0, 0, 1, 0.96])

fig2.savefig(H4P2_CM_PLOT)

plt.show()
plt.close('all')
```