EECE6036: Intelligent Systems
Homework # 5

Zuguang Liu (M10291593)

December 1, 2020

1 Self-Organized Feature Map (SOFM)

1.1 Problem Statement

In this problem, a SOFM is implemented and trained on the MNIST dataset. The map consists of 12 $\times$ 12 neurons, each of which is a 784-long vector that forms the input layer.

1.2 System Description

Using the same stratified sampling policy the previous Homework used, the preprocessing section splits 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. This balanced partition controls the training by letting the model adjust to equal amount of data over all classes.

Weights are initialized based on the first six principal components of the training data from Singular Value Decomposition process. The six vectors are linearly combined in a random fashion to form the initial weights of 144 neurons.

The training of the model is a two-phase process, where each phase is also dynamic with gradually decreasing learning rate (1), and gradually centralized neighborhood function (2, σ is the standard deviation in the Gaussian function). Weights are adjusted per data point, repeated for multiple epochs (as in t). Specifically,

1. In self-organizing phase, $t = 1,000, \eta_0 = 0.1, \tau_L = 1,000, \sigma_0 = 8.5, \tau_N = 467$. These values are set so that the neighborhood function starts from covering most of the map, to only cover the winning unit at the end.

2. In convergence phase, the training slowly adjusts weights that only belong to the winning unit for 500 epochs. This is done by setting $t = 500, \eta_0 = 0.01, \tau_L \approx \infty, \sigma_0 \approx 0, \tau_N \approx \infty$.

$$\eta(t) = \eta_0 \exp(-t/\tau_L) \tag{1}$$

$$\sigma(t) = \sigma_0 \exp(-t/\tau_N) \tag{2}$$

After training, the resulting model is analyzed using the test dataset.

1.3 Results

Fig. 1 shows heat maps for each class, where in each plot, the winning rate of the 12 $\times$ 12 neurons are presented by a greyscale pixel (dark pixels means low winning rate), and the pixels are mapped in the same configuration as the neurons.

The weights of the neurons at the end of the training are demonstrated in Fig. 2, with the same mapping fashion as designed.

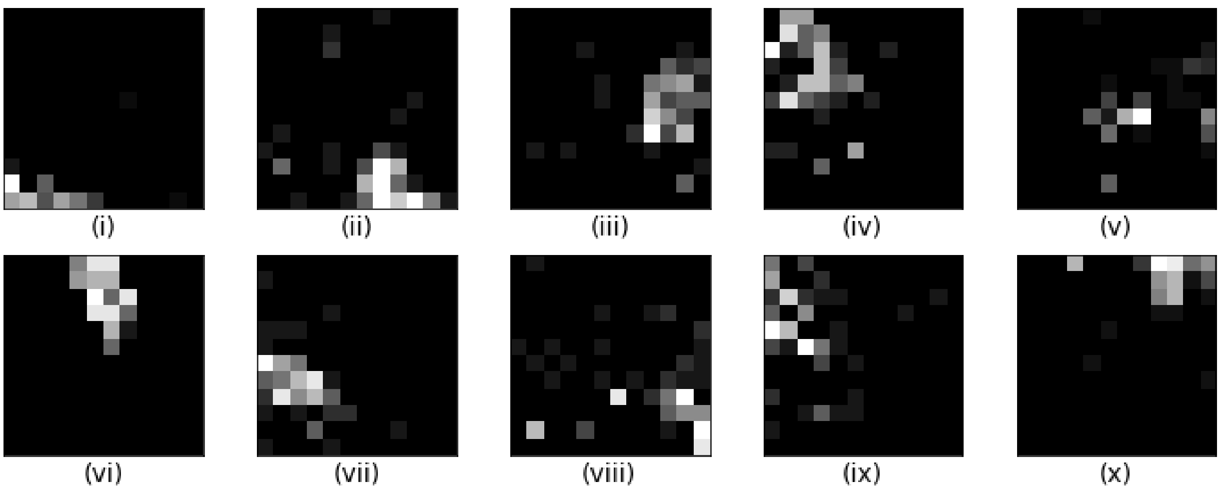


Fig. 1: Heat map of the neurons separated by classes. The plot label uses roman letters that also represent the class of the images, for example, (i) uses images of digit '1'. Digit '0' is shown in (x).

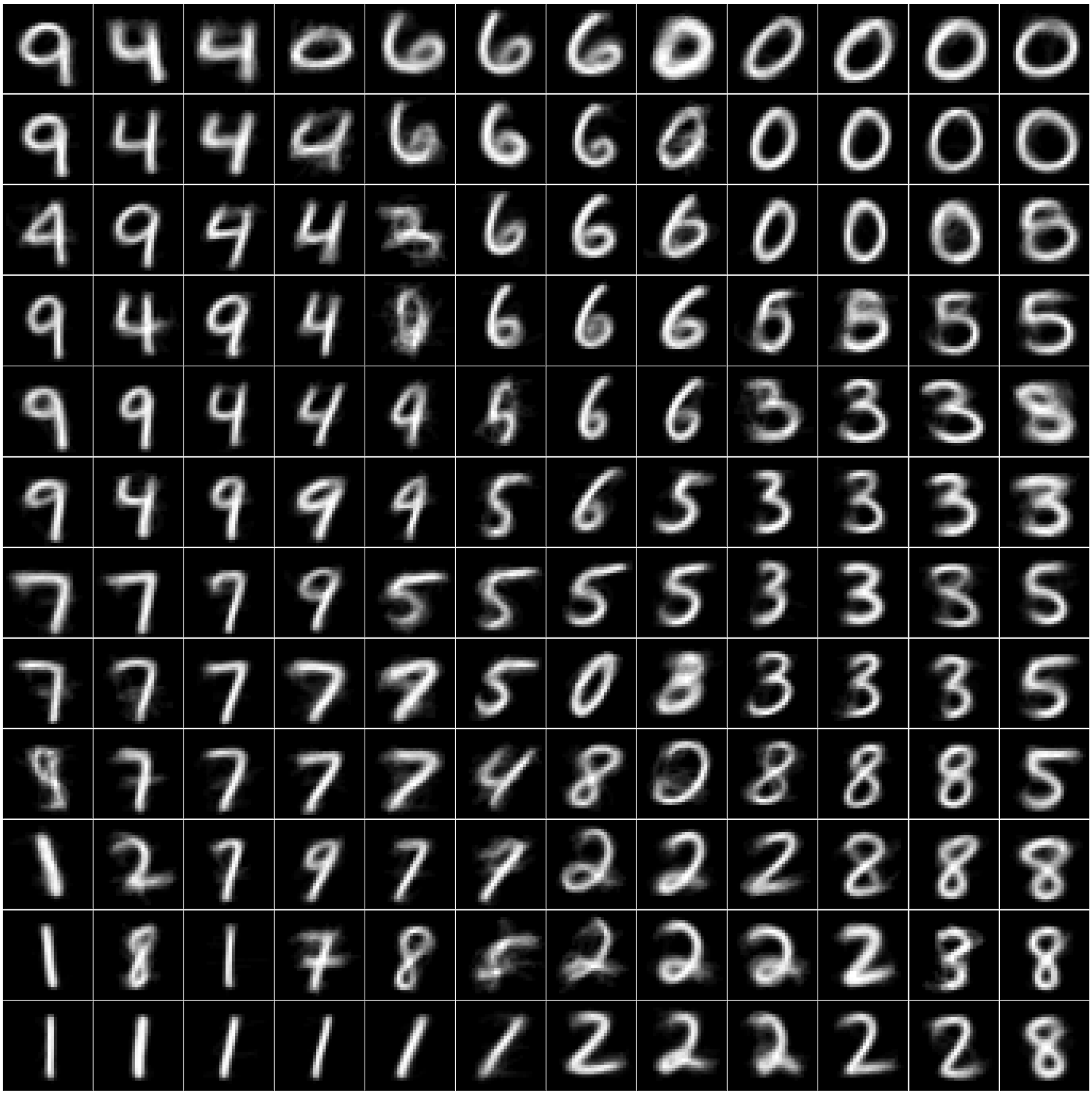


Fig. 2: Feature map of the neurons in a 12 × 12 configuration

1.4 Analysis of Results

The feature map in Fig. 2 shows great resemblance with the looks of the original data, making the resulting model successful and explainable. The map does not only demonstrates the evolution on how the model interprets the image data, but also groups similar image patterns together. For example, in the lower left corner of the map, there are a few drawings of digit '1' with small levels of variations. Then going from this corner to the lower right corner, images that look like '2' and '8' are shown, where the visual complexity of the image pattern increases gradually.

With the grouping pattern found in the feature map, it is not surprising to find that spatially close neurons fire together in choosing the winner in heat maps, and that winning neurons all have feature maps that look like corresponding digits as well. The upper right corner of Fig. 2 is populated with images that look like '0', so accordingly, neurons at the upper right corner mostly win when the model is presented with test data of class '0' in Fig. 1 (x).

2 Classifiers Based on SOFM

2.1 Problem Statement

Using the trained SOFM as a hidden layer, a classifier can be constructed with an additional output layer that infers the class of images. 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, and the two classifiers from Homework # 4 whose hidden layer comes from autoencoders.

To tell apart the models, the report uses **SOFM classifier** to address this new model, and calls the two autoencoder-based network **denoising classifier** and **reconstructing classifier**, as the original autoencoders are to reduce noise or reconstruct the image. The last 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 models.

2.2 System Description

To control the training settings, **all four classifiers are trained with exactly the same data, algorithm and hyper-parameters**. Since the previous Homework used 128 neurons instead of 144, the relevant models are adjusted to 144 neurons and trained again.

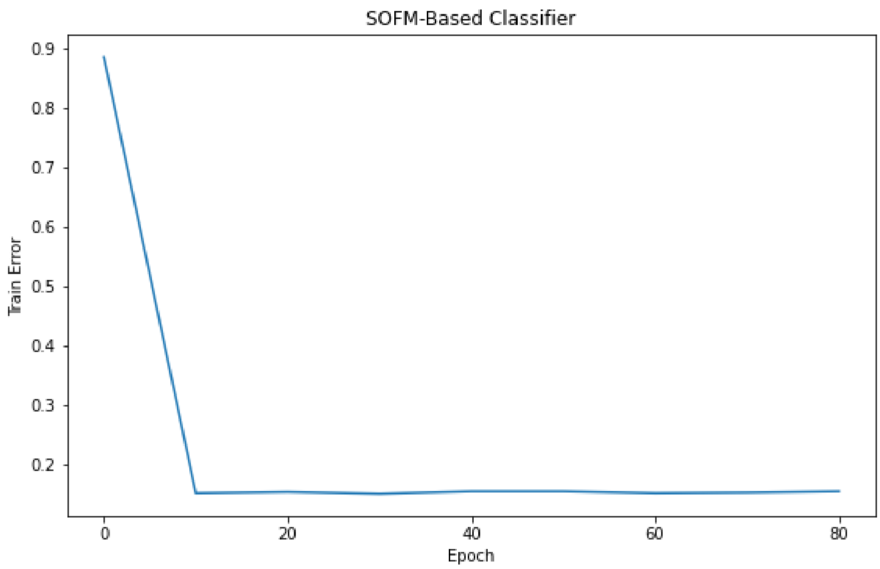
Similar to the previous Homework, the SOFM connects to an output layer of 10 neurons, each of which represents the possibility of the image belonging to a class. To implement forward feeding from the SOFM to the output layer, "winner-take-all" strategy is used, so only the winner outputs 1 and the rest is 0. Finally, to decide on the label, neurons with the highest output are selected.

The training proceeds similarly to the previous Homework. Back propagation with momentum ($\eta = 0.1, \alpha = 0.8$) is used, iterated over all data points and repeated for 500 epochs. Only the weights of the output layer is adjusted. Threshold of 0.25 and 0.75 are used for consistent hyperparameter set though not practically applicable. Regularization methods include weight decay ($\lambda = 10^{-5}$) and validation-based early stopping (50 epochs patience).

After training, the test dataset is presented to all mentioned classifying networks.

2.3 Results

Time series of on-line training loss on SOFM classifer is presented in Fig. 3. Fig. 4 shows the confusion matrices of the four classifiers on train data and test data.



* Training early stopped at epoch 80,then restored to epoch 30, when error is 0.152.

Fig. 3: Error vs epochs during training of SOFM classifier

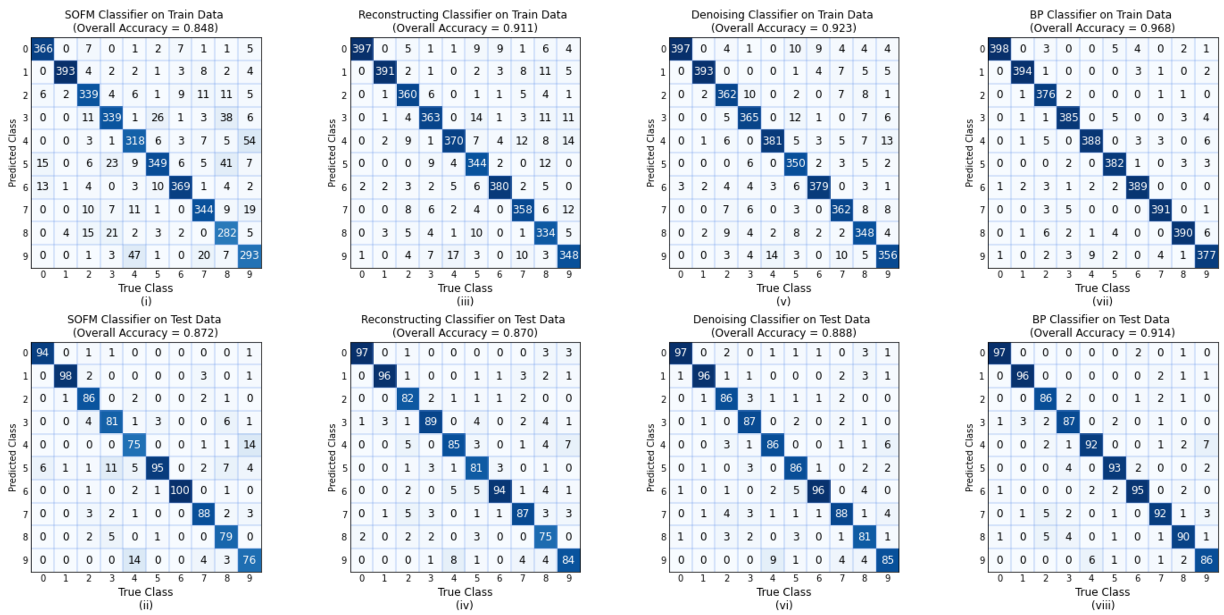


Fig. 4: Performance of four classifier implemented so far

2.4 Analysis of Results

The accuracy of the SOFM classifer is slightly higher than that of the reconstructing classifier, making it an acceptable candidate for the classification problem. It is also unexpected to see the accuracy on the test dataset is higher than that on the train dataset, which is not the case for any other models, suggesting that the SOFM classifier is more robust in terms of preventing overfitting. Moreover, since the features of the SOFM look visually close to the original images (Fig. 2), it is easy to identify the source of error and adjust accordingly. For example, in Fig. 4 (ii), there are 14 images of digit '4' classified as '9' and vice versa, which is explainable by the fact that feature maps that look like '4' and '9' are not distinctly separated in Fig. 2.

The four models, including SOFM classifier, reconstructing classifier, denoising classifier, and BP classifier, took 47 min, 15 min, 14 min, and 12 min collectively to train. More optimization can be done admittedly, but it is a fact that a simple two-layer fully-connected neural network has the best accuracy with the least computational effort. However, though having the longest training time and mediocre accuracy, SOFM classifer has undeniably high robustness and explainability, and therefore some potential to assist on highly complex deep networks.

Appendix A Python Code: preprocess.py

```python
import numpy as np
import csv
import settings
import json
import pandas as pd
import itertools


def prepare_img(img_a):
    img_a = 1- np.array(img_a).flatten()
    img_a = (img_a - np.min(img_a)) / (np.max(img_a) - np.min(img_a))
    # 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)
```

```python

def int_to_roman(num):
    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()

def subplt_size(pane_shape, plt_shape):
    return tuple(np.flip(np.multiply(pane_shape,plt_shape)))

def grid_combo(*args):
    return list(itertools.product(*args))

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

```python
import numpy as np
from preprocess import get_rand_list, stratify_split
import json
from tqdm import trange
import cupy as cp
from sofm import SOFM


class DenseLayer(object): # fully connected layer
    def __init__(self, n_input, n_neurons, activation=None, trainable=True,
                 weights=None, zero_bias=False):
        """
        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))
            weights = cp.random.uniform(low=-a, high=+a, size=(n_input+1, n_neurons)) #Xavier initialization
            if zero_bias:
                weights[0] = 0.
            self.weights = weights
        else:
            if type(weights) != cp.core.core.ndarray:
                weights = cp.asarray(weights)

            if zero_bias:
                weights = cp.concatenate((cp.asarray([0], weights)))

            if weights.shape != (n_input+1, n_neurons):
                raise ValueError('Given weights {} does not match given '
                                 'dimensions {}'.format(weights.shape, (n_input+1, n_neurons)))
            self.weights = weights

        self.trainable = trainable

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

        self.activation = activation
        self.last_activation = None
        self.error = cp.zeros(n_neurons)
        self.delta = cp.zeros(n_neurons)
        self.layer_type = "Dense"

    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.

        """
        if type(x) != cp.core.core.ndarray:
                x = cp.asarray(x)
        x = cp.concatenate((cp.asarray([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 cp.maximum(s,0)
        elif self.activation == 'tanh':
            return cp.tanh(s)
        elif self.activation == 'sigmoid':
            return 1.0 / (1.0 + cp.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 = cp.copy(s)
            grad[s>0] = 1.0
            grad[s<=0] = 0.0
            return grad
        elif self.activation == 'tanh':
            return 1 - cp.power(s,2)
        elif self.activation == 'sigmoid':
            return s * (1-s)
        else:
            return cp.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.

        """
        self.weights += dweights
        self.last_dweights = dweights

    def get_weights(self):
        return cp.asnumpy(self.weights.copy())

    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])

    def save_dict(self):
        layer_dict = {}
        layer_dict['n_input'], layer_dict['n_neurons'] = self.io()
        layer_dict['activation'] = self.activation
        layer_dict['trainable'] = self.trainable
        layer_dict['weights'] = self.weights.tolist()
        return layer_dict

    def info(self):
        return 'Dense, input: {}, output: {}, activation: {}, trainable: {}'.format(self.io()[0], self.io()[1], self.
      activation, self.trainable)







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.weight_decay = 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 layer.trainable:
                if i == len(self._layers) -1: # for output layer
                    raw_error = y - output
                    raw_error[abs(raw_error)<threshold] = 0 # 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):
            if layer.trainable:
                pre_synaptic = (x if i == 0 else self._layers[i-1].last_activation)
                pre_synaptic = cp.concatenate((cp.asarray([1]),pre_synaptic))
                delta_weights = cp.atleast_2d(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)

        if X_train != cp.core.core.ndarray:
            X_train = cp.asarray(X_train)
        if Y_train != cp.core.core.ndarray:
            Y_train = cp.asarray(Y_train)



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

        if classify:
            validation_i, realtrain_i = stratify_split(Y_train.get(), 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 = cp.asarray(errors)
                        return 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.

        """
        if X_test != cp.core.core.ndarray:
            X_test = cp.asarray(X_test)
```

```python
        if Y_test != cp.core.core.ndarray:
            Y_test = cp.asarray(Y_test)

        error = 0
        for i in range(len(X_test)):
            pred = self._feed_forward(X_test[i])
            error += cp.sum(cp.power(pred-Y_test[i],2))
        return cp.asnumpy(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 = []

        if X_test != cp.core.core.ndarray:
            X_test = cp.asarray(X_test)
        if Y_test != cp.core.core.ndarray:
            Y_test = cp.asarray(Y_test)
        for i in range(len(X_test)):
            pred = self._feed_forward(X_test[i])
            pred = cp.argmax(pred)
            truth = cp.argmax(Y_test[i])
            errors.append(pred==truth)
        return cp.asnumpy(1-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 = cp.asnumpy(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['layers'] = [layer.layer_type for layer in self._layers]
            meta_dict['train_errors'] = self.train_errors.tolist()
            json.dump(meta_dict, f)
            f.write("\n")
            for layer in self._layers:
                layer_dict = layer.save_dict()
                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'])
                    layers = meta['layers']
                else:
                    layer = json.loads(line)
                    if layers[i-1] == "Dense":
                        self.add_layer(DenseLayer(n_input=layer['n_input'],
                                                  n_neurons=layer['n_neurons'],
                                                  activation=layer['activation'],
                                                  weights=layer['weights'],
                                                  trainable=layer['trainable']))
                    elif layers[i-1] == "SOFM":
                        self.add_layer(SOFM(mapshape=layer['mapshape'],
                                            weights=layer['weights']))



    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 {}] {}'.format(i, layer.info()))

    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(cp.asnumpy(x))
        return np.array(pred)

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

Appendix C Python Code: sofm.py

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Nov 21 17:02:34 2020

@author: liu
"""

import numpy as np
import cupy as cp
from tqdm import trange
import json

class SOFM(object):
    def __init__(self, mapshape, weights=None):
        self.mapshape = mapshape

        if weights is not None:
            weights = cp.asarray(weights)
            if weights.shape[1] != np.prod(mapshape):
                raise Exception("Given weights does not match dimension")
            self.weights = weights
        else:
            self.weights = None

        indxmap = []
        for i in np.arange(np.prod(mapshape)):
            indx = np.unravel_index(i, mapshape, order='F')
            indxmap.append(indx)
        self.indxmap = cp.asarray(indxmap)

        self.sigma = 0
        self.eta = 0
        self.sigma_tau = 0

        self.trainable = False
        self.activation = "SOFM"
        self.zero_bias = True
        self.last_activation = cp.zeros(int(np.prod(mapshape)))
        self.error = cp.zeros(int(np.prod(mapshape)))
        self.delta = cp.zeros(int(np.prod(mapshape)))

        self.layer_type = "SOFM"




    def initialize(self, data, pca=6):
        if type(data) != cp.core.core.ndarray:
            data = cp.asarray(data)

        if type(pca) != int:
            pca = int(pca)

        dmin = cp.min(data)
        dmax = cp.max(data)

        if pca > 0:
            data = cp.asarray(data)
            _,_,pc = cp.linalg.svd(data)
            pc = pc[:pca].T
            combo = np.random.dirichlet(np.ones(pca),
                                        size=np.prod(self.mapshape)).T
            combo = cp.asarray(combo)

            w = pc @ combo
            self.weights = dmin + (w - cp.min(w)) * (dmax-dmin) / (cp.max(w) - cp.min(w))
        else:
            data = cp.asarray(data)
            self.weights = cp.random.uniform(dmin, dmax, size=
                                         (data.shape[-1],np.prod(self.mapshape)))



    def winner(self, vector, flat=True):
        if type(vector) != cp.core.core.ndarray:
            vector = cp.asarray(vector)

        if vector.shape != self.weights.shape:
            vector = cp.broadcast_to(vector, self.weights.shape[::-1]).T

        indx_win = cp.argmin(cp.linalg.norm(self.weights - vector, axis=0))

        if flat:
            return indx_win
        else:
            return cp.asanyarray(cp.unravel_index(indx_win, self.mapshape, order='F'))




    def cycle(self, vector, eta_t, epoch):
        if type(vector) != cp.core.core.ndarray:
            vector = cp.asarray(vector)
        if vector.shape != self.weights.shape:
            vector = cp.broadcast_to(vector, self.weights.shape[::-1]).T

        indx_win = self.winner(vector, flat=False)
        dists = cp.linalg.norm(self.indxmap - indx_win, axis=1)

        sigma_t = self.sigma * cp.exp( -1 * epoch / self.sigma_tau)
        delta = cp.exp(-1*cp.power(dists, 2) / 2 / cp.power(sigma_t,2))
        delta = cp.broadcast_to(delta, self.weights.shape)
        self.weights += eta_t * delta * (vector - self.weights)
```

```python

    def train(self, data, max_epochs, learning_rate, learning_rate_decay,
                 learning_width, learning_width_decay, online_test=True):
        self.sigma = learning_width
        self.sigma_tau = learning_width_decay

        if type(data) != cp.core.core.ndarray:
            data = cp.asarray(data)

        if self.weights is None:
            self.initialize(data)

        for epoch in trange(max_epochs+1, ncols=75, unit='epoch'):
            cp.random.shuffle(data)

            if epoch % 10 == 0 and online_test:
                print("\nLoss = {} at epoch {}".format(self.test(data), epoch))

            for vector in data:
                self.cycle(vector, learning_rate*cp.exp(-epoch/learning_rate_decay), epoch)


    def test(self, data):
        errors = []

        if type(data) != cp.core.core.ndarray:
            data = cp.asarray(data)

        for vector in data:
            vector = cp.broadcast_to(vector, self.weights.shape[::-1]).T
            dist_win = cp.min(cp.linalg.norm(self.weights - vector, axis=0))
            errors.append(cp.asnumpy(dist_win))

        return np.mean(errors)

    def save(self, file_name):
        if type(file_name) is not str:
            file_name = str(file_name)

        with open(file_name,'w') as f:
            json.dump(self.weights.tolist(), f)

    def save_dict(self):
        layer_dict = {}
        layer_dict['mapshape'] = self.mapshape
        layer_dict['weights'] = self.weights.tolist()
        return layer_dict


    def load(self, file_name):
        if type(file_name) is not str:
            file_name = str(file_name)

        with open(file_name, 'r') as f:
            weights = json.load(f)
            weights = cp.asarray(weights)
            if weights.shape[-1] != np.prod(self.mapshape):
                raise Exception("Loaded weights do not match dimension")
            self.weights = weights


    def get_weights(self):
        return cp.asnumpy(self.weights.copy())


    def call(self, x):
        if type(x) != cp.core.core.ndarray:
            x = cp.asarray(x)

        if x.shape != self.weights.shape:
            x = cp.broadcast_to(x, self.weights.shape[::-1]).T


        indx_win = self.winner(x, flat=True)
        output = cp.zeros(self.weights.shape[1])
        output[indx_win] = 1.0

        # output = cp.linalg.norm(self.weights - x, axis=0)
        # output = 1.0 - (output - cp.min(output)) / (cp.max(output) - cp.min(output))

        self.last_activation = output

        return output

    def apply_activation_derivative(self, s):
        return cp.zeros(self.weights.shape[1])

    def update_weights(self, dweights):
        pass

    def io(self):
        return (self.weights.shape[0], self.weights.shape[1])

    def info(self):
        return 'SOFM, mapshape: {}'.format(self.mapshape)
```

Appendix D Python Code: h5p1_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 E Python Code: h5p1_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).get_weights()[:,ni][1:]
    features[1] = autoenc_noise.layers(0).get_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 F Python Code: h5p2_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 G Python Code: h5p2_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')
```