EECE6036: Intelligent Systems
Homework # 3

Zuguang Liu (M10291593)

November 15, 2020

1 Neural Network Classifier with Hidden Layer

1.1 Problem Statement

A classifier shall be implemented by a two-layer neural network to identify hand-written digits from MNIST dataset. The given MNIST dataset includes 5,000 labeled images, each of which writes digits '0' to '9' in different fashions, and is thus labeled into 10 classes. The images are represented by 28 $\times$ 28 pixels of normalized greyscale color values.

1.2 System Description

Since neural network is a supervised learning model that requires training, there shall be an exclusive split of the dataset for training and testing. Consequently, apart from randomly shuffled, the data is also partitioned into 4,000 data points for training, and 1,000 for testing. The partitioning shall be stratified over the 10 classes, i.e., having the same number of data points for each class, in both the training set and testing set. In addition, the labels of the data are one-hot encoded to expand the output feature space to 10 dimensions.

The two-layer neural network consists of a hidden layer and an output layer. **The hidden layer incorporates 128 neurons** that transfer all 784 pixel values of an image into a 128-dimensional feature space. Then the output layer uses this feature space to classify the image with an array of 10 elements, each of which represents the probability that the image can belong to a class. Hence the input and output of the network matches the images and labels in dimensions respectively. **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 in the interval (a, 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 repeated for multiple epochs. The training data is exclusively partitioned into 3,000 points for back propagation, and 1,000 points for validation. Within one epoch, all 3,000 points are shuffled, then used to adjust weights and biases. **The model is tested on the validation set every 10 epochs, resulting in a series of on-line training errors**. The error 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. To further improve the training efficiency, several mechanisms are used in the training algorithm, including:

a. The gradient decent has an additional term to implement momentum, demonstrated in (3), **where $\eta = 0.05$, $\alpha = 0.8$, and J is loss function implemented by J_2 loss**.

$$\Delta w_{ij}(t) = -\eta \frac{\partial J}{\partial w_{ij}} + \alpha \Delta w_{ij}(t - 1) \tag{3}$$

b. **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.

c. Though the total repetition is 500 epochs, **an early stopping policy is used so that training stops when the on-line training loss does not improve 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 tested on the test dataset for analysis.

1.3 Results

Fig. 1 (a) and (b) are confusion matrix that demonstrates the performance of the classifier on the training and testing dataset, respectively. Fig. 2 is show the change of the on-line training error over epochs.

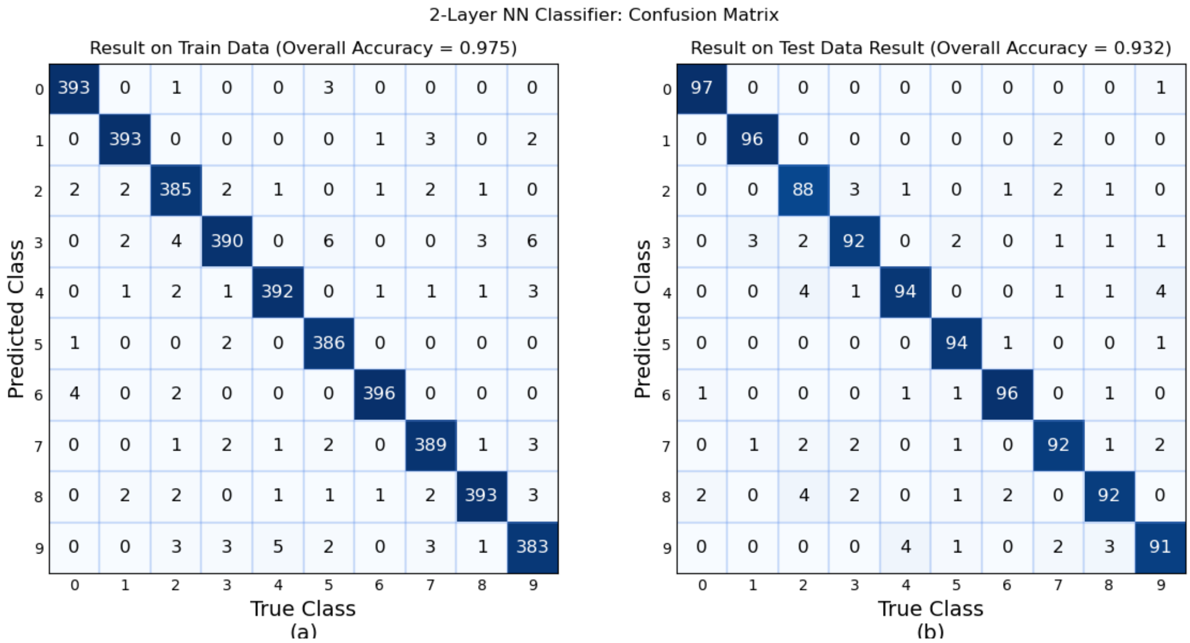

Fig. 1: Confusion matrix of the classifier on the train data (a) and test data (b).

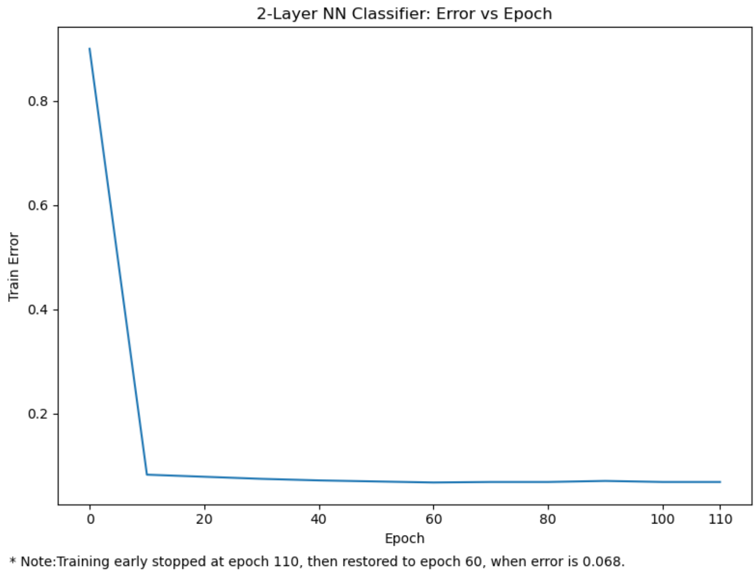

Fig. 2: The time series of the error (1-balanced accuracy) vs epochs.

1.4 Analysis of Results

The overall hit-rate on the train and test data is 98.2% and 92.5%, making the model a success. The 6.1% difference could be caused by a slight level of overfitting, but both accuracy is good in general for a classifier.

In the confusion matrices (Fig. 1), columns mark the actual class by the label, and rows mark the prediction from the model. In (b), since the test dataset are made so that there are 100 images for each 'true class', the numbers in each column sums up to 100, so the numbers in the diagonal is the percent hit rate for each class. Images of classes '0', '1', '3', '5', '6', '7' have over 90% hit rate for example, while other classes are also predicted well with the lowest hit rate at 88%. Looking back into (a), we can find there are at minimum 391 correctly classified images for every class, which is still better than the worst result on the test data. Overfitting may have affected the result, but it could also be due to the way the digits are written has too many degrees of freedom and the classifier was not able to pick up all of them.

To further improve the performance of the classifier, changes in several directions could be considered.

a. Improvements on the quantity and quality of the data could be used. Quantitatively, the model could have larger number of data points for training and testing. Qualitatively, the images can be more defined in the resolution.

b. The hyper-parameters of the model could be fully tested using grid search.

c. A more advanced optimization algorithm could be used during training. For example, instead of a single term to represent the first derivative of Δw (3), the *ADAM* algorithm includes the second derivative also to further improve the efficiency of gradient descent.

2 Neural Network Autoencoder with Hidden Layer

2.1 Problem Statement

An auto-encoder network with one hidden layer shall be constructed for the same dataset, with the objective of reconstructing the image exactly. Therefore, the two-layer network has 784 inputs, and 784 outputs, representing a pixel in a 28 × 28 image. The number of perceptrons in the hidden layer should be the same as that of the one in the classifier.

2.2 System Description

Since there are 128 neurons in the hidden layer of the classifier, the **autoencoder also has 128 hidden neurons**. The architecture of the autoencoder is similar to that of the classifier, except the output layer has 784 neurons to offer 784 values to regenerate the image. **Sigmoid (1) and Xavier initialization policy (2) are still used for both layers.** The training of the model also uses similar algorithm with minor modifications, including:

a. The input image and the ground truth used in the training are the same per step in the back propagation.

b. The back propagation uses **momentum-based gradient descent (3) with $\eta = 0.01, \alpha = 0.8$**.

c. **The on-line training error is calculated by the average J_2 loss function over all data points in the validation set.** This is presented by (4), 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. However, the same early-stop policy is used as well, so **the training will stop after no improvement on the error for 50 epochs**.

$$\bar{J}_2 = \frac{1}{N}\sum_{n=1}^{N} J_2 = \frac{1}{1000}\sum_{n=1}^{1000}\left(\frac{1}{2}\sum_{i=1}^{784}(y_i^n - \hat{y}_i^n)^2\right) \tag{4}$$

After training, the autoencoder is again tested on the test dataset for analysis.

2.3 Results

Fig. 3 demonstrates the performance of the autoencoder overall and at each class, where the error is calculated by (4). The on-line training error vs epochs is shown in Fig. 4.

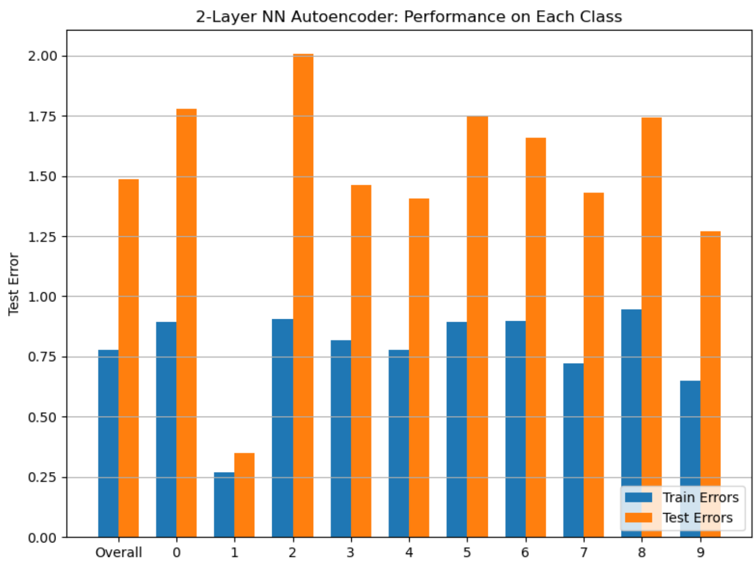

Fig. 3: Performance of the autoencoder on the training and test set.

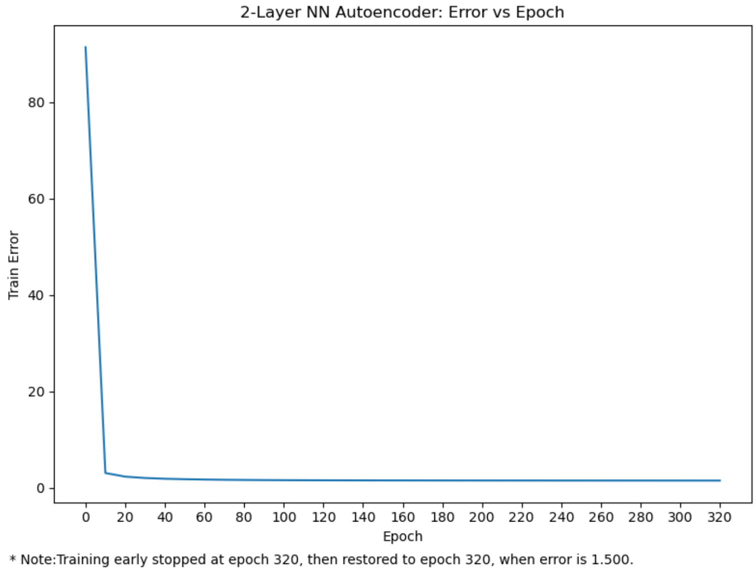

Fig. 4: On-line training error vs training epochs over time

2.4 Features

Weights of 20 neurons in the hidden layer of the classifier (Fig. 5) and the autoencoder (Fig. 6) are normalized and illustrated by 28 × 28 images, 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 between the classifier and the autoencoder.

The original expectation was that the hidden features in the autoencoder could demonstrate more distinct shapes or patterns than the classifier, but the result suggests otherwise. It was surprising to see larger clusters of approximate pixel values in the classier features than in the autoencoder features. This is discusses further in the analysis.

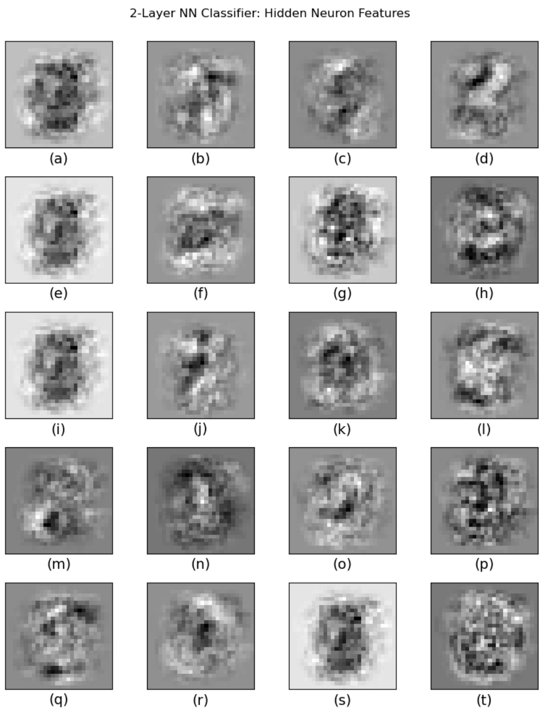

Fig. 5: Feature map of 20 neurons in the hidden layer of the classifier network.

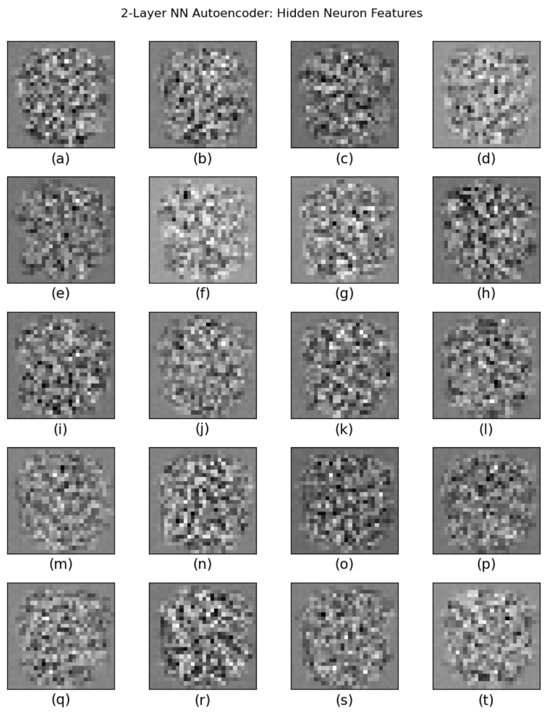

Fig. 6: Feature map of 20 neurons in the hidden layer of the classifier network.

2.5 Sample Outputs

Fig. 7 demonstrates 8 reconstructed images (i)-(p) compared against the original images (a)-(h).

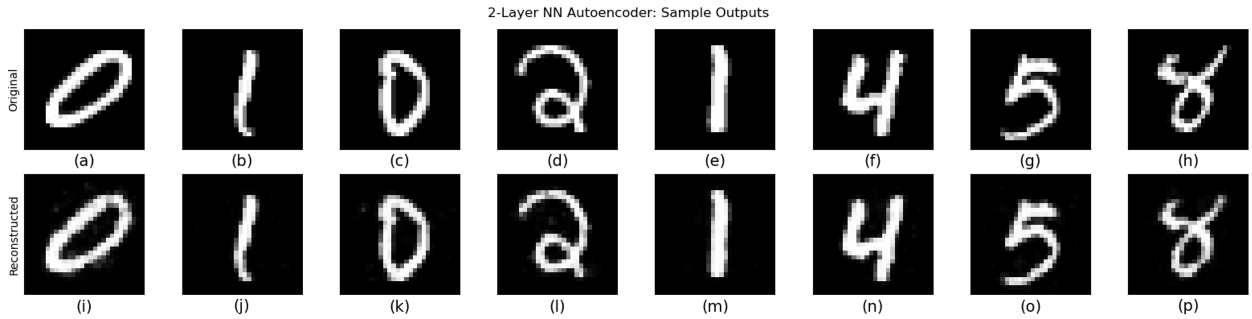

Fig. 7: 8 Sample output of the autocencoder compared to original images.

2.6 Analysis of Results

The resulting autoencoder is successfully trained with good level of accuracy. Mathematically, when the model is applied on the testing dataset, the overall J_2 error between the output and the original pixels is around 1.6 per image (Fig. 3), similar to the end on-line training error (Fig. 4). Visually, the reconstructed images look very much alike the original images (Fig. 7). From a detailed observation, there is minor noise in (p) that writes '2', (i) that writes '5' and (l) that writes '8'. This aligns with the J_2 loss in Fig. 3, as class 2, 5, and 8 have the highest errors among all classes.

Though both classifier and autoencoder network do well in their own problems, the difference in the patterns of the hidden layer feature was quite astonishing at first. It is theorized that the checkerboard pattern in the autoencoder hidden features (Fig. 6) is due to the need for the hidden layer to encode spatial correlations between pixels while reducing the feature space dimension. The model meets this goal by a set of weights that "accepts" some pixels ($w_{ij} > \bar{\boldsymbol{w}}$) and "denys" some pixels ($w_{ij} < \bar{\boldsymbol{w}}$) periodically at a certain "sampling rate", thus the checkerboard pattern. In the classifier (Fig. 5), however, there is no need to encode local correlations, so pixels that are spatially close are allowed to be weighted similarly, forming more visual clusters. Due to limited time and knowledge constraints, this hypothesis is not verified yet but may offer a reasonable explanation.

Lastly, it is important that the training efficiently converges to such result, thanks to a well-defined model, detailed training policy (momentum-based gradient descent, early stop, etc.) and a finely-tuned set of hyper-parameters. The success of both models will provide an amount of experience in the future machine learning problems.

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
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Oct 24 16:13:36 2020

@author: liu
"""

import numpy as np
import csv
import settings
import json
import pandas as pd

def prepare_img(img_a):
    img_a = np.array(img_a).flatten()
    img_a = 1 - 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):
    length = int(length)
    return np.random.choice(length,length,replace=False)

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 find_ES(train_errors, max_epochs):
    epochs = 10*np.arange(len(train_errors))
    epochs = np.flip(epochs)
    train_errors = np.flip(train_errors)
    if len(train_errors) < max_epochs:
        return "Training early stopped at epoch {}, then restored to epoch {}, when error is {:.3f}.".format(epochs[0], epochs
    [np.argmin(train_errors)], np.min(train_errors))
    else:
        return "Last training error is {:.3f}".format(train_errors[0])


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

Appendix B Python Code: nn.py

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Oct 25 14:15:46 2020

@author: liu
"""
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.
```

```python

        """
        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

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

            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)
        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.

        """
        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.

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

    def layers(self, n):
        """
        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.

        """
        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
```

Appendix C Python Code: p1_train.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Oct 25 21:27:27 2020

@author: liu
"""

from preprocess import get_train, get_test
from settings import SIZES, H3P1_NN, HIDDEN_NEURONS, MAX_EPOCHS, VALI_R
from nn import NeuralNetwork, DenseLayer

train_db = get_train()
test_db = get_test()
network = NeuralNetwork()
network.add_layer(DenseLayer(n_input=SIZES['x'][1], n_neurons=HIDDEN_NEURONS,
                             activation='sigmoid'))
network.add_layer(DenseLayer(n_input=HIDDEN_NEURONS, n_neurons=SIZES['classes'],
                             activation='sigmoid'))
network.train(train_db['x'],train_db['y'],max_epochs=MAX_EPOCHS,
              classify=True, threshold=0.25,
              validation_ratio=VALI_R, earlystop=(0,5),
              learning_rate = 0.04,
              momentum=0.8,
               weight_decay=1E-4,
              )
network.save(str(H3P1_NN))
```

Appendix D Python Code: p1_test.py

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Oct 27 18:07:34 2020

@author: liu
"""
from preprocess import get_train, get_test, find_ES
from settings import CLASSES, H3P1_NN, H3P1_CM_PLOT, H3P1_TRAIN_PLOT, MAX_EPOCHS
from nn import NeuralNetwork
import numpy as np
import matplotlib.pyplot as plt

# Load the network
train_db = get_train()
test_db = get_test()
network = NeuralNetwork()
network.load(str(H3P1_NN))

# Plot training error vs epoch
train_errors = network.train_errors
epochs = 10*np.arange(len(train_errors))
fig1, ax1 = plt.subplots(figsize=(8,6))
ax1.plot(epochs, train_errors)
ax1.set_xticks(np.append(np.arange(0,epochs[-1],20),epochs[-1]))
ax1.set_title("2-Layer NN Classifier: Error vs Epoch")
ax1.set_xlabel("Epoch")
ax1.set_ylabel("Train Error")
notetxt = find_ES(train_errors, MAX_EPOCHS)
fig1.text(0.02, 0.01, "* Note:"+notetxt, ha='left')
fig1.tight_layout(rect=[0, 0.02, 1, 1])
fig1.savefig(H3P1_TRAIN_PLOT)

# Plot confusion metrix
cm = []
cm.append(network.get_cm(train_db['x'], train_db['y']))
cm.append(network.get_cm(test_db['x'], test_db['y']))
errors =[]
errors.append(network.classify_test(train_db['x'], train_db['y']))
errors.append(network.classify_test(test_db['x'], test_db['y']))
fig2, ax2 = plt.subplots(1,2, figsize=(12,6))
for n in range(2):
    ax2[n].imshow(cm[n], cmap='Blues')
    ax2[n].set_xticks(CLASSES)
    ax2[n].set_yticks(CLASSES)
    ax2[n].set_xticklabels(CLASSES)
    ax2[n].set_yticklabels(CLASSES)
    ax2[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[n][i,j]>=50 else 'k'
            text = ax2[n].text(j, i, int(cm[n][i, j]), ha="center", va="center", color=c, fontsize=12)
    ax2[n].set_xlabel("True Class\n({})".format(chr(ord('a')+n)), fontsize=14)
    ax2[n].set_ylabel("Predicted Class", fontsize=14)
    for num in CLASSES:
        ax2[n].axvline(num-0.5, c='cornflowerblue', lw=1.5, alpha=0.3)
        ax2[n].axhline(num-0.5, c='cornflowerblue', lw=1.5, alpha=0.3)
ax2[0].set_title("Result on Train Data (Overall Accuracy = {:.3f})".format(1-errors[0]))
ax2[1].set_title("Result on Test Data Result (Overall Accuracy = {:.3f})".format(1-errors[1]))
fig2.suptitle("2-Layer NN Classifier: Confusion Matrix")
fig2.tight_layout(rect=[0, 0, 1, 0.92])

fig2.savefig(H3P1_CM_PLOT)

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

Appendix E Python Code: p2_train.py

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from preprocess import get_train, get_test
from settings import SIZES, H3P2_NN, HIDDEN_NEURONS, MAX_EPOCHS, VALI_R
from nn import NeuralNetwork, DenseLayer


train_db = get_train()
test_db = get_test()
network = NeuralNetwork()
network.add_layer(DenseLayer(n_input=SIZES['x'][1], n_neurons=HIDDEN_NEURONS,
                             activation='sigmoid'))
network.add_layer(DenseLayer(n_input=HIDDEN_NEURONS, n_neurons=784,
                             activation='sigmoid'))
network.train(train_db['x'], train_db['x'],learning_rate = 0.01,
              max_epochs=MAX_EPOCHS, classify=False, momentum=0.8,
              threshold=0, validation_ratio=VALI_R, earlystop=(1E-3,5),
              # weight_decay=1E-5,
              )
network.save(str(H3P2_NN))
```

Appendix F Python Code: p2_test.py

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Oct 27 18:36:30 2020

@author: liu
"""

from preprocess import get_train, get_test, get_rand_list, prepare_img, find_ES
from settings import CLASSES, SIZES, MAX_EPOCHS
from settings import H3P2_NN, H3P2_TRAIN_PLOT, H3P2_TEST_PLOT, H3P2_FEATURE_MAP, H3P2_OUTPUT_MAP, H3P1_NN, H3P1_FEATURE_MAP,
    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 = NeuralNetwork()
autoenc.load(str(H3P2_NN))

# Plot training error vs epoch
train_errors = autoenc.train_errors
epochs = 10*np.arange(len(train_errors))
fig1, ax1 = plt.subplots(figsize=(8,6))
ax1.plot(epochs, train_errors)
ax1.set_xticks(np.append(np.arange(0,epochs[-1],20),epochs[-1]))
ax1.set_title("2-Layer NN Autoencoder: Error vs Epoch")
ax1.set_xlabel("Epoch")
ax1.set_ylabel("Train Error")
notetxt = find_ES(train_errors, MAX_EPOCHS)
fig1.text(0.02, 0.01, "* Note:"+notetxt, ha='left')
fig1.tight_layout(rect=[0, 0.02, 1, 1])
fig1.savefig(H3P2_TRAIN_PLOT)

# Plot testing errors
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])
    train_errors[c].append(autoenc.raw_test([x],[x]))
for i,x in enumerate(test_db['x']):
    c = np.argmax(test_db['y'][i])
    test_errors[c].append(autoenc.raw_test([x],[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))
fig2, ax2 = plt.subplots(figsize=(8,6))
width = 0.35
ticks = [str(c) for c in CLASSES]
ticks.insert(0,'Overall')
ax2.bar(np.arange(len(ticks)) - width/2, train_errors, width, label='Train Errors')
ax2.bar(np.arange(len(ticks)) + width/2, test_errors, width, label='Test Errors')
ax2.set_xticks(np.arange(len(ticks)))
ax2.set_xticklabels(ticks)
ax2.set_ylabel('Test Error')
ax2.set_title("2-Layer NN Autoencoder: Performance on Each Class")
ax2.legend(loc='lower right')
ax2.grid(axis='y')
fig2.tight_layout()
fig2.savefig(H3P2_TEST_PLOT)

# Plot feature maps
fig3, ax3 = plt.subplots(5,4, figsize=(8,10))
neuron_i = get_rand_list(HIDDEN_NEURONS)[:20]
for i,ni in enumerate(neuron_i):
    features = autoenc.layers(0).weights[:,ni][1:]
    features = features / np.linalg.norm(features)
    ax3[i//4][i%4].imshow(prepare_img(features), cmap='binary')
    ax3[i//4][i%4].set_xticks([])
    ax3[i//4][i%4].set_yticks([])
    ax3[i//4][i%4].set_xlabel('({})'.format(chr(ord('a')+i)), fontsize=14)
fig3.suptitle("2-Layer NN Autoencoder: Hidden Neuron Features")
fig3.tight_layout(rect=[0, 0, 1, 0.95])
fig3.savefig(H3P2_FEATURE_MAP)

classifier = NeuralNetwork()
classifier.load(str(H3P1_NN))
fig4, ax4 = plt.subplots(5,4, figsize=(8,10))
for i,ni in enumerate(neuron_i):
    features = classifier.layers(0).weights[:,ni][1:]
    ax4[i//4][i%4].imshow(prepare_img(features), cmap='binary')
    ax4[i//4][i%4].set_xticks([])
    ax4[i//4][i%4].set_yticks([])
    ax4[i//4][i%4].set_xlabel('({})'.format(chr(ord('a')+i)), fontsize=14)
fig4.suptitle("2-Layer NN Classifier: Hidden Neuron Features")
fig4.tight_layout(rect=[0, 0, 1, 0.95])
fig4.savefig(H3P1_FEATURE_MAP)

# Plot sample output
img_i = get_rand_list(SIZES['test'])[:8]
fig5, ax5 = plt.subplots(2,8, figsize=(16,4))
for i, ii in enumerate(img_i):
    original = test_db['x'][ii]
    ax5[0][i].imshow(prepare_img(original), cmap='binary')
    reconstructed = autoenc.predict([test_db['x'][ii]])
    ax5[1][i].imshow(prepare_img(reconstructed), cmap='binary')
    ax5[0][i].set_xticks([])
    ax5[0][i].set_yticks([])
    ax5[0][i].set_xlabel('({})'.format(chr(ord('a')+i)), fontsize=14)
    ax5[1][i].set_xticks([])
    ax5[1][i].set_yticks([])
```

```
    ax5[1][i].set_xlabel('({})'.format(chr(ord('i')+i)), fontsize=14)
ax5[0][0].set_ylabel("Original")
ax5[1][0].set_ylabel("Reconstructed")
fig5.suptitle("2-Layer NN Autoencoder: Sample Outputs")
fig5.tight_layout(rect=[0, 0, 1, 0.95])
fig5.savefig(H3P2_OUTPUT_MAP)


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