Project 5: Multi-layer Perceptron

Artificial Intelligence

Bibhash Mulmi

Goal: To improve the accuracy of a multi-layer neural network.

Procedures:

Experiment 1: Change the learning rate hyperparameter with the following hyperparameters as default,

minibatch size = 100

Number of hidden layer neurons = 100

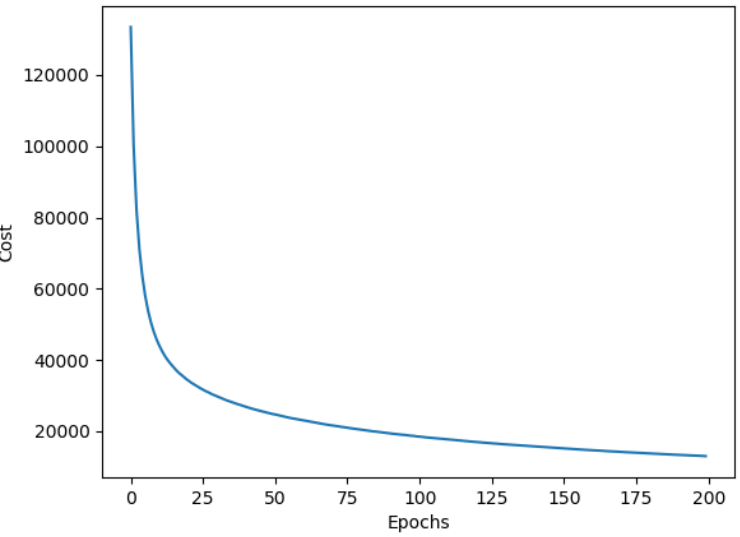
Function used = sigmoid

Topology used = two-layer neural network

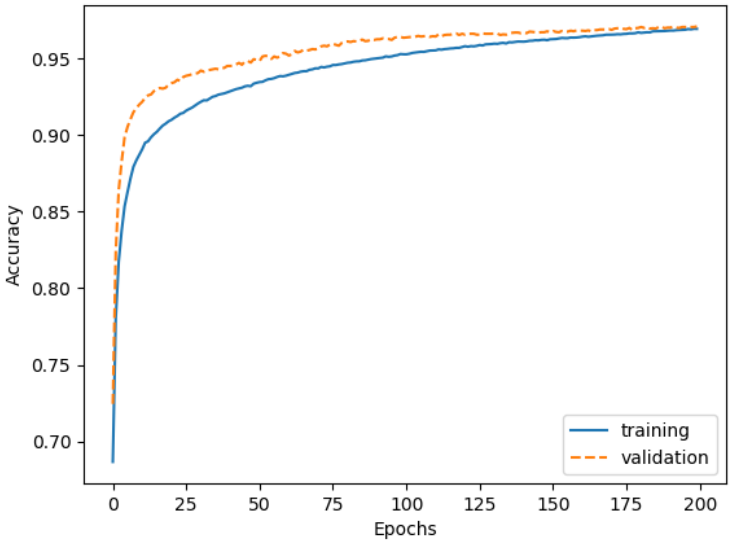
Observations:

The accuracy increased simultaneously with the value of learning rate, however the accuracy graph diverged. This showed that there was overfitting of data. By observing the graph, the value of learning rate was altered accordingly. Finally, the optimal learning rate corresponding to a converging graph was found to be 0.00011.

Learning rate constant	**Train accuracy (%)**	**Valid accuracy (%)**	**Test accuracy (%)**	**Graph (Converging/Diverging)**
0.0005	99.28	97.98	97.54	Diverging
0.002	99.76	98.22	97.89	Diverging
0.0035	99.63	98.18	97.92	Diverging
0.0002	98.13	97.58	96.92	Diverging
0.0001	99.67	98.10	97.86	Diverging
0.00015	97.57	97.38	96.61	Diverging
0.00013	97.31	97.22	96.38	Converging
0.00011	**96.92**	**97.08**	**96.24**	**Converging**



Graph representing cost per epoch



Graph representing accuracy per epoch

Conclusion: The accuracy of the network can be improved by changing the learning rate to 0.00011 which yields the optimal result.

Experiment 2: Changing the minibatch size with the following hyperparameters as default,

Learning rate = 0.00011

Number of hidden layer neurons = 100

Function used = sigmoid

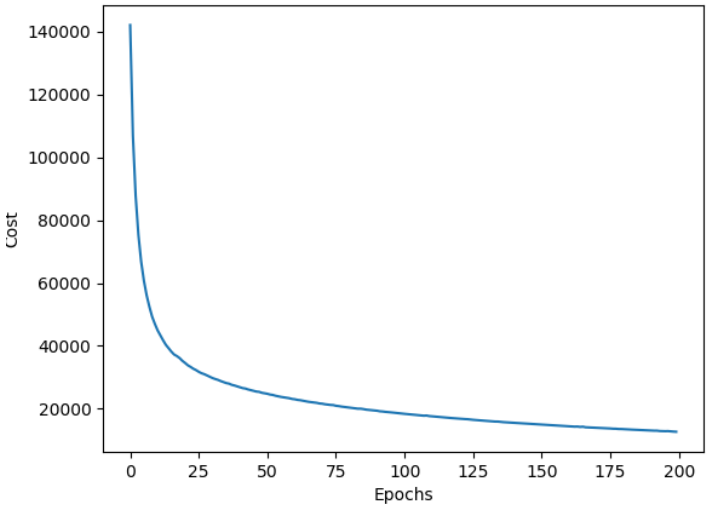
Topology used = two-layer neural network

Observations:

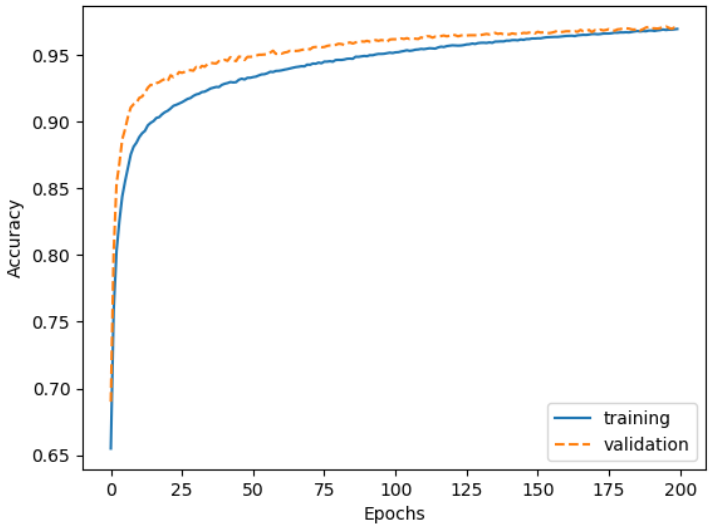
The accuracy increased simultaneously with the increase in minibatch sizes. However, the accuracy started decreasing after one point. This point was determined as the local maxima in the graph. The optimal value of the minibatch size was found to be 3000.

Minibatch size	**Train Accuracy (%)**	**Valid Accuracy (%)**	**Test Accuracy (%)**	**Graph (Converging/Diverging)**
100	96.92	97.08	96.24	Converging
50	96.79	97.04	96.18	Converging
125	96.95	97.10	96.25	Converging
150	96.95	97.08	96.26	Converging
165	96.96	97.08	96.25	Converging

700	97.02	97.06	96.33	Converging
1400	97.05	97.00	96.26	Converging
3000	**96.97**	**97.10**	**96.38**	**Converging**
4500	96.84	96.92	96.12	Converging



Graph representing cost per epoch



Graph representing accuracy per epoch

Conclusion: The accuracy of the network can be improved by increasing its value to 3000. Higher value than 3000 decreases the accuracy.

Experiment 3: Use tanh function with the following hyperparameters as default,

Number of hidden layer neurons = 100

Topology used = two-layer neural network

Observations:

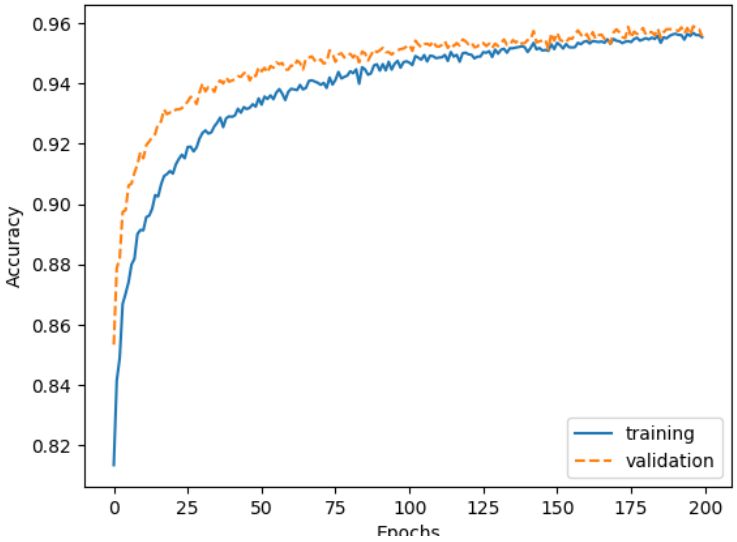
The values for the learning rate and minibatch sizes were altered accordingly to get a converging graph by observing the accuracy graph. The optimal values for the learning rate and minibatch sizes, while using tanh function, were observed to be 0.0003 and 150 respectively.

The accuracy was less for the learning rate (0.00011) and minibatch size (3000) values that were optimal when using sigmoid function.

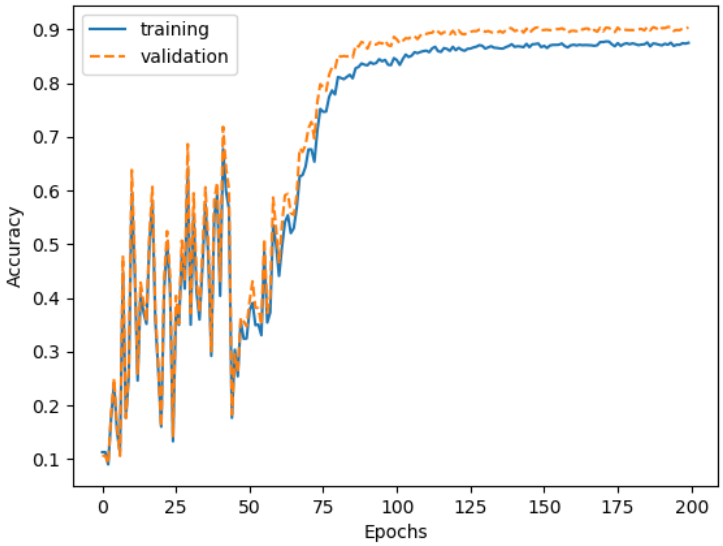
Additionally, the accuracy was observed to be highly fluctuating while using tanh function.

Learning Rate Constant	Minibatch Size	Train Accuracy (%)	Valid Accuracy (%)	Test Accuracy (%)	Graph observed
0.00011	150	94.25	95.60	93.75	Diverging
0.0002	100	95.51	96.24	94.90	Diverging

0.0004	100	96.18	96.36	95.34	Converging with high fluctuation in accuracy
0.0004	150	95.76	96.32	94.74	Diverging with fluctuation in accuracy
0.0003	**150**	**95.53**	**95.56**	**94.50**	**Converging with fluctuation in accuracy**
0.00009	500	93.53	94.58	93.50	Diverging
0.0003	500	92.54	93.62	92.16	Converging with fluctuation in accuracy
0.00011 (optimal value for sigmoid)	*3000 (optimal value for sigmoid)*	*87.49*	*90.36*	*88.17*	*Extremely high fluctuating diverging graph*
0.0001	3000	94.36	95.32	94.16	Diverging



Graph representing accuracy per epoch for the optimal values for learning rate and minibatch sizes using tanh function



Graph representing accuracy per epoch using the optimal learning rate and minibatch size values of sigmoid function while using tanh function

Conclusion: The accuracy does not improve when tanh function is used.

Experiment 4: Change the number of hidden layer neurons with the following hyperparameters as default,

Learning rate constant = 0.00011

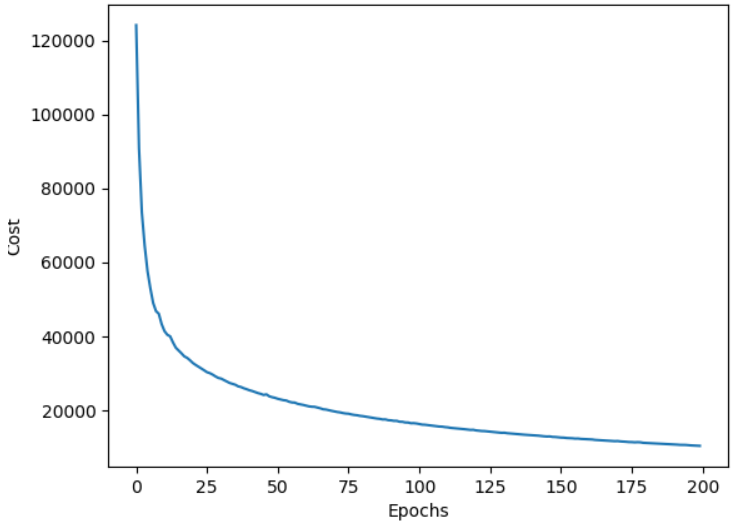
Topology used = two-layer neural network

Observations:

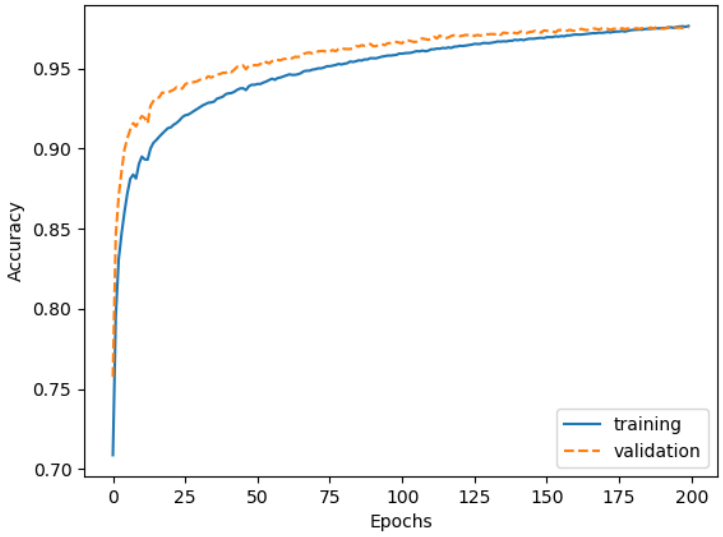
The accuracy increased with the increase of neurons in the hidden layer.

Number of neurons	Learning rate constant	Minibatch Size	Train accuracy (%)	Valid accuracy (%)	Test accuracy (%)	Graph observed
200	0.00011	500	97.67	97.66	98.87	Converging
150	0.00011	500	97.45	97.42	96.75	Converging
150	0.00011	500	97.45	97.42	96.75	Converging
150	0.00011	300	97.43	97.44	96.72	Converging
250	0.00011	3000	97.64	97.54	96.93	Converging
1000	**0.00011**	**500**	**99.96**	**99.90**	**98.17**	**Converging**

(**Note*: The time taken to train the machine increased dramatically with the increase in number of neurons. So, it was concluded that the accuracy increased with the increase in number of neurons. *)



Graph representing cost per epoch



Graph representing accuracy per epoch

Conclusion: The accuracy of the network can by improved by increasing the number of neurons in the hidden layer. The number of neurons in each layer is directly proportional to the accuracy.

Experiment 5: Change the topology from one to two hidden layers neural network with the following hyperparameters as default,

Learning rate constant = 0.00011

Minibatch size = 3000

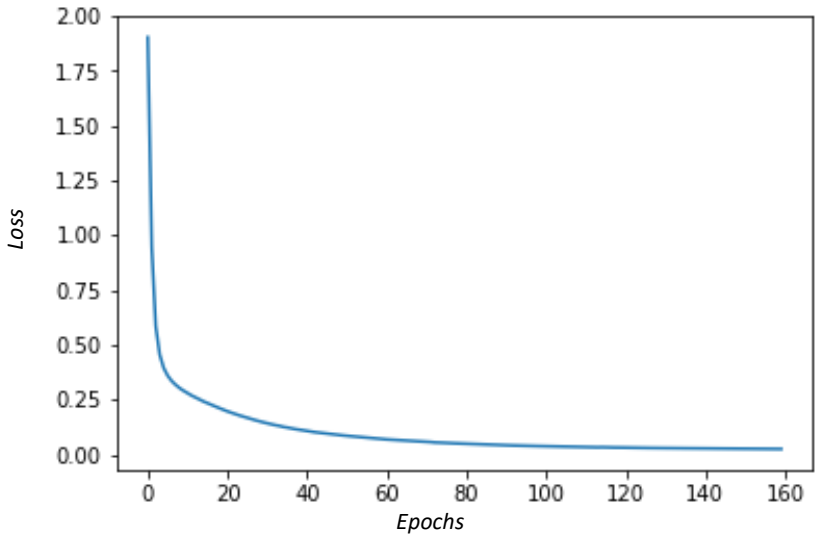
Function used = sigmoid

Python's Scikit Learn Library's MLP Classifier was used for this experiment.

Observations:

The accuracy increased with the addition of one hidden layer. Moreover, it also increased when the number of neurons in these layers were increased.

Number of Neurons in First Hidden Layer	Number of Neurons in Second Hidden Layer	Train Accuracy (%)	Valid Accuracy (%)	Test Accuracy (%)
100	100	99.80	98.14	97.89
250	100	99.97	98.42	98.14
250	500	99.98	98.54	98.05
550	500	99.98	98.56	98.13
1000	**500**	**98.20**	**98.58**	**98.20**



Graph representing loss per epoch

Conclusion: The accuracy improves when changing the topology from one to two hidden layers. The number of neurons in each layer is directly proportional to the accuracy. Thus, the best result is obtained when the number of neurons is maximum in each layer.

Source Code for Experiment 5:

```
import os
import struct
import numpy as np
def load_mnist(path, kind='train'):
    """Load MNIST data from `path`"""
    labels_path = os.path.join(path,
```

```
                               '%s-labels-idx1-ubyte' % kind)
    images_path = os.path.join(path,
                               '%s-images-idx3-ubyte' % kind)

    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',
                                 lbpath.read(8))
        labels = np.fromfile(lbpath,
                             dtype=np.uint8)

    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII",
                                               imgpath.read(16))
        images = np.fromfile(imgpath,
                             dtype=np.uint8).reshape(len(labels), 784)
        images = ((images / 255.) - .5) * 2

    return images, labels

from sklearn.neural_network import MLPClassifier
import matplotlib.pyplot as plt

import sys
import gzip
import shutil

if (sys.version_info > (3, 0)):
    writemode = 'wb'
else:
    writemode = 'w'

zipped_mnist = [f for f in os.listdir('./') if f.endswith('ubyte.gz')]
for z in zipped_mnist:
    with gzip.GzipFile(z, mode='rb') as decompressed, open(z[:-3], writemode)
as outfile:
        outfile.write(decompressed.read())


# Load training records
X_train, y_train = load_mnist('', kind='train')
print('Rows: %d, columns: %d' % (X_train.shape[0], X_train.shape[1]))


# Load testing records
X_test, y_test = load_mnist('', kind='t10k')
print('Rows: %d, columns: %d' % (X_test.shape[0], X_test.shape[1]))

np.savez_compressed('mnist_scaled.npz',
                    X_train=X_train,
```

```
                    y_train=y_train,
                    X_test=X_test,
                    y_test=y_test)


mnist = np.load('mnist_scaled.npz')
mnist.files

X_train, y_train, X_test, y_test = [mnist[f] for f in ['X_train', 'y_train',
                                    'X_test', 'y_test']]

del mnist

X_train.shape

X_valid=X_train[55000:]
y_valid=y_train[55000:]
X_train=X_train[:55000]
y_train=y_train[:55000]


mlp = MLPClassifier(hidden_layer_sizes=(250,500), #250 neurons, 500 neurons
                    activation='logistic', #logistic = sigmoid function
                    max_iter=200, #epochs
                    alpha=0.01, #l2 term
                    solver='adam', #solver for weight optimization
                    batch_size=100, #minibatch size
                    verbose=10, #to print no. of progress messages
                    tol=1e-4, #tolerance for optimization
                    shuffle=True,
                    random_state=None, #if None, the random number generator
is the RandomState instace used by np.random
                    learning_rate_init=.00011 #learning rate
                   )


mlp.fit(X_train, y_train)
print("Training set score: %f" % mlp.score(X_train, y_train))
print("Test set score: %f" % mlp.score(X_test, y_test))
print("Valid Training set score: %f" % mlp.score(X_valid, y_valid))

plt.plot(loss_curve_)
plt.show()
```

Source code:

```
import os
import struct
import numpy as np

def load_mnist(path, kind='train'):
    """Load MNIST data from `path`"""
    labels_path = os.path.join(path,
                               '%s-labels-idx1-ubyte' % kind)
    images_path = os.path.join(path,
                               '%s-images-idx3-ubyte' % kind)

    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',
                                 lbpath.read(8))
        labels = np.fromfile(lbpath,
                             dtype=np.uint8)

    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII",
                                               imgpath.read(16))
        images = np.fromfile(imgpath,
                             dtype=np.uint8).reshape(len(labels), 784)
        images = ((images / 255.) - .5) * 2

    return images, labels

# unzips mnist

import sys
import gzip
import shutil

if (sys.version_info > (3, 0)):
    writemode = 'wb'
else:
    writemode = 'w'

zipped_mnist = [f for f in os.listdir('./') if f.endswith('ubyte.gz')]
for z in zipped_mnist:
    with gzip.GzipFile(z, mode='rb') as decompressed, open(z[:-3], writemode)
as outfile:
        outfile.write(decompressed.read())

# Load training records
X_train, y_train = load_mnist('', kind='train')
print('Rows: %d, columns: %d' % (X_train.shape[0], X_train.shape[1]))
```

```
# Load testing records
X_test, y_test = load_mnist('', kind='t10k')
print('Rows: %d, columns: %d' % (X_test.shape[0], X_test.shape[1]))

X_train, y_train, X_test, y_test = [mnist[f] for f in ['X_train', 'y_train',
                                    'X_test', 'y_test']]

del mnist

X_train.shape

class NeuralNetMLP(object):
    """ Feedforward neural network / Multi-layer perceptron classifier.

    Parameters
    ------------
    n_hidden : int (default: 30)
        Number of hidden units.
    l2 : float (default: 0.)
        Lambda value for L2-regularization.
        No regularization if l2=0. (default)
    epochs : int (default: 100)
        Number of passes over the training set.
    eta : float (default: 0.001)
        Learning rate.
    shuffle : bool (default: True)
        Shuffles training data every epoch if True to prevent circles.
    minibatche_size : int (default: 1)
        Number of training samples per minibatch.
    seed : int (default: None)
        Random seed for initalizing weights and shuffling.

    Attributes
    -----------
    eval_ : dict
      Dictionary collecting the cost, training accuracy,
      and validation accuracy for each epoch during training.

    """
    def __init__(self, n_hidden=30,
                 l2=0., epochs=100, eta=0.001,
                 shuffle=True, minibatch_size=1, seed=None):

        self.random = np.random.RandomState(seed)
        self.n_hidden = n_hidden
        self.l2 = l2
        self.epochs = epochs
        self.eta = eta
```

```
        self.shuffle = shuffle
        self.minibatch_size = minibatch_size

    def _onehot(self, y, n_classes):
        """Encode labels into one-hot representation

        Parameters
        ------------
        y : array, shape = [n_samples]
            Target values.

        Returns
        -----------
        onehot : array, shape = (n_samples, n_labels)

        """
        onehot = np.zeros((n_classes, y.shape[0]))
        for idx, val in enumerate(y.astype(int)):
            onehot[val, idx] = 1.
        return onehot.T

    def _sigmoid(self, z):
        """Compute logistic function (sigmoid)"""
        return 1. / (1. + np.exp(-np.clip(z, -250, 250)))


    def _tanh(self, z):
        e_p = np.exp(z)
        e_m = np.exp(-z)
        return (e_p - e_m) / (e_p + e_m)

    def _forward(self, X):
        """Compute forward propagation step"""

        # step 1: net input of hidden layer
        # [n_samples, n_features] dot [n_features, n_hidden]
        # -> [n_samples, n_hidden]
        z_h = np.dot(X, self.w_h) + self.b_h

        # step 2: activation of hidden layer
        #a_h = self._sigmoid(z_h)
        a_h = self._tanh(z_h)

        # step 3: net input of output layer
        # [n_samples, n_hidden] dot [n_hidden, n_classlabels]
        # -> [n_samples, n_classlabels]

        z_out = np.dot(a_h, self.w_out) + self.b_out
```

```
        # step 4: activation output layer
        a_out = self._sigmoid(z_out)
        a_out = self._tanh(z_out)

        return z_h, a_h, z_out, a_out

    def _compute_cost(self, y_enc, output):
        """Compute cost function.

        Parameters
        ----------
        y_enc : array, shape = (n_samples, n_labels)
            one-hot encoded class labels.
        output : array, shape = [n_samples, n_output_units]
            Activation of the output layer (forward propagation)

        Returns
        ---------
        cost : float
            Regularized cost

        """
        L2_term = (self.l2 *
                   (np.sum(self.w_h ** 2.) +
                    np.sum(self.w_out ** 2.)))

        term1 = -y_enc * (np.log(output))
        term2 = (1. - y_enc) * np.log(1. - output)
        cost = np.sum(term1 - term2) + L2_term
        return cost

    def predict(self, X):
        """Predict class labels

        Parameters
        -----------
        X : array, shape = [n_samples, n_features]
            Input layer with original features.

        Returns:
        ----------
        y_pred : array, shape = [n_samples]
            Predicted class labels.

        """
        z_h, a_h, z_out, a_out = self._forward(X)
        y_pred = np.argmax(z_out, axis=1)
        return y_pred
```

```
def fit(self, X_train, y_train, X_valid, y_valid):
    """ Learn weights from training data.

    Parameters
    -----------
    X_train : array, shape = [n_samples, n_features]
        Input layer with original features.
    y_train : array, shape = [n_samples]
        Target class labels.
    X_valid : array, shape = [n_samples, n_features]
        Sample features for validation during training
    y_valid : array, shape = [n_samples]
        Sample labels for validation during training

    Returns:
    ----------
    self

    """
    n_output = np.unique(y_train).shape[0]  # number of class labels
    n_features = X_train.shape[1]

    ########################
    # Weight initialization
    ########################

    # weights for input -> hidden
    self.b_h = np.zeros(self.n_hidden)
    self.w_h = self.random.normal(loc=0.0, scale=0.1,
                                  size=(n_features, self.n_hidden))

    # weights for hidden -> output
    self.b_out = np.zeros(n_output)
    self.w_out = self.random.normal(loc=0.0, scale=0.1,
                                    size=(self.n_hidden, n_output))

    epoch_strlen = len(str(self.epochs))  # for progress formatting
    self.eval_ = {'cost': [], 'train_acc': [], 'valid_acc': []}

    y_train_enc = self._onehot(y_train, n_output)

    # iterate over training epochs
    for i in range(self.epochs):

        # iterate over minibatches
        indices = np.arange(X_train.shape[0])
```

```
            if self.shuffle:
                self.random.shuffle(indices)

            for start_idx in range(0, indices.shape[0] - self.minibatch_size +
                                   1, self.minibatch_size):
                batch_idx = indices[start_idx:start_idx + self.minibatch_size]

                # forward propagation
                z_h, a_h, z_out, a_out = self._forward(X_train[batch_idx])

                ##################
                # Backpropagation
                ##################

                # [n_samples, n_classlabels]
                sigma_out = a_out - y_train_enc[batch_idx]

                # [n_samples, n_hidden]
                #sigmoid_derivative_h = a_h * (1. - a_h)
                tan_derivative_h = ( 2 / (np.exp (a_h + np.exp(-a_h) ) ) ) **2

                # [n_samples, n_classlabels] dot [n_classlabels, n_hidden]
                # -> [n_samples, n_hidden]
                #sigma_h = (np.dot(sigma_out, self.w_out.T) *
                #           sigmoid_derivative_h)

                sigma_h = (np.dot(sigma_out, self.w_out.T) *
                           tan_derivative_h)

                # [n_features, n_samples] dot [n_samples, n_hidden]
                # -> [n_features, n_hidden]
                grad_w_h = np.dot(X_train[batch_idx].T, sigma_h)
                grad_b_h = np.sum(sigma_h, axis=0)

                # [n_hidden, n_samples] dot [n_samples, n_classlabels]
                # -> [n_hidden, n_classlabels]
                grad_w_out = np.dot(a_h.T, sigma_out)
                grad_b_out = np.sum(sigma_out, axis=0)

                # Regularization and weight updates
                delta_w_h = (grad_w_h + self.l2*self.w_h)
                delta_b_h = grad_b_h # bias is not regularized
                self.w_h -= self.eta * delta_w_h
                self.b_h -= self.eta * delta_b_h

                delta_w_out = (grad_w_out + self.l2*self.w_out)
                delta_b_out = grad_b_out  # bias is not regularized
                self.w_out -= self.eta * delta_w_out
```

```
                self.b_out -= self.eta * delta_b_out

            #############
            # Evaluation
            #############

            # Evaluation after each epoch during training
            z_h, a_h, z_out, a_out = self._forward(X_train)

            cost = self._compute_cost(y_enc=y_train_enc,
                                      output=a_out)

            y_train_pred = self.predict(X_train)
            y_valid_pred = self.predict(X_valid)

            train_acc = ((np.sum(y_train == y_train_pred)).astype(np.float) /
                         X_train.shape[0])
            valid_acc = ((np.sum(y_valid == y_valid_pred)).astype(np.float) /
                         X_valid.shape[0])

            sys.stderr.write('\r%0*d/%d | Cost: %.2f '
                             '| Train/Valid Acc.: %.2f%%/%.2f%% ' %
                             (epoch_strlen, i+1, self.epochs, cost,
                              train_acc*100, valid_acc*100))
            sys.stderr.flush()

            self.eval_['cost'].append(cost)
            self.eval_['train_acc'].append(train_acc)
            self.eval_['valid_acc'].append(valid_acc)

        return self

n_epochs = 200

nn = NeuralNetMLP(n_hidden=100,
                  l2=0.01,
                  epochs=n_epochs,
                  eta=0.00011,
                  minibatch_size=3000,
                  shuffle=True,
                  seed=1)

nn.fit(X_train=X_train[:55000],
       y_train=y_train[:55000],
       X_valid=X_train[55000:],
       y_valid=y_train[55000:])

import matplotlib.pyplot as plt
```

```
#cost per epoch graph
plt.plot(range(nn.epochs), nn.eval_['cost'])
plt.ylabel('Cost')
plt.xlabel('Epochs')
plt.show()

#accuracy graph
plt.plot(range(nn.epochs), nn.eval_['train_acc'],
         label='training')
plt.plot(range(nn.epochs), nn.eval_['valid_acc'],
         label='validation', linestyle='--')
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.legend()
plt.show()

y_test_pred = nn.predict(X_test)
acc = (np.sum(y_test == y_test_pred)
       .astype(np.float) / X_test.shape[0])

print('Test accuracy: %.2f%%' % (acc * 100))
```