```
#Alon Shmilovich 034616359
#JCE, Jerusalem College of Engineering

import numpy as np
import itertools
from PIL import Image
import collections
from sklearn.cross_validation import train_test_split
from sklearn import cross_validation
from sklearn import datasets
import matplotlib.pyplot as plt
import glob


# Global variable, will be used in logistic function to see if it's MNIST or
    not
dataset_type = ''

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    -
# Define the layers used in this model
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    -
class Layer(object):
    """Base class for the different layers.
    Defines base methods and documentation of methods."""

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    def get_params_iter(self):
        """Return an iterator over the parameters (if any).
        The iterator has the same order as get_params_grad.
        The elements returned by the iterator are editable in-place."""
        return []

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    def get_params_grad(self, X, output_grad):
        """Return a list of gradients over the parameters.
        The list has the same order as the get_params_iter iterator.
        X is the input.
        output_grad is the gradient at the output of this layer.
        """
        return []

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    def get_output(self, X):
        """Perform the forward step linear transformation.
        X is the input."""
        pass

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    def get_input_grad(self, Y, output_grad=None, T=None):
        """Return the gradient at the inputs of this layer.
        Y is the pre-computed output of this layer (not needed in this case).
        output_grad is the gradient at the output of this layer
         (gradient at input of next layer).
        Output layer uses targets T to compute the gradient based on the
```

```
         output error instead of output_grad"""
        pass

# -----------------------------------------------------------------------------
#  LogisticLayer
# -----------------------------------------------------------------------------
class LogisticLayer(Layer):
    """The logistic layer applies the logistic function to its"""

    def get_output(self, X):
        """Perform the forward step transformation."""
        return self.logistic(X)

    # -------------------------------------------------------------------------

    def get_input_grad(self, Y, output_grad):
        """Return the gradient at the inputs of this layer."""
        return np.multiply(self.logistic_deriv(Y), output_grad)

    # -------------------------------------------------------------------------

    def logistic(self, z):
        if dataset_type != 'MNIST':
             z[z > 700] = 700
             z[z < 700] = -700
        return 1 / (1 + np.exp(-z))

    # -------------------------------------------------------------------------

    def logistic_deriv(self, y):  # Derivative of logistic function
        return np.multiply(y, (1 - y))

# -----------------------------------------------------------------------------
#  LinearLayer
# -----------------------------------------------------------------------------
class LinearLayer(Layer):
    """The linear layer performs a linear transformation to its input."""

    def __init__(self, n_in, n_out, rate):
        """Initialize hidden layer parameters.
        n_in is the number of input variables.
        n_out is the number of output variables."""
        self.W = np.random.randn(n_in, n_out) * rate # initial with random
            values
        self.b = np.zeros(n_out)

    # -------------------------------------------------------------------------

    def get_params_iter(self):
        """Return an iterator over the parameters."""
        return itertools.chain(np.nditer(self.W, op_flags=['readwrite']),
                               np.nditer(self.b, op_flags=['readwrite']))

    # -------------------------------------------------------------------------

    def get_output(self, X):
```

```
        """Perform the forward step linear transformation."""
        return X.dot(self.W) + self.b

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    def get_params_grad(self, X, output_grad):
        """Return a list of gradients over the parameters."""
        JW = X.T.dot(output_grad)
        Jb = np.sum(output_grad, axis=0)
        return [g for g in itertools.chain(np.nditer(JW), np.nditer(Jb))]

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    def get_input_grad(self, Y, output_grad):
        """Return the gradient at the inputs of this layer."""
        return output_grad.dot(self.W.T)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    -
#  SoftmaxOutputLayer
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    -
class SoftmaxOutputLayer(Layer):
    """The softmax output layer computes the classification propabilities at
       the output."""

    def get_output(self, X):
        """Perform the forward step transformation."""
        return self.softmax(X)

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    def get_input_grad(self, Y, T):
        """Return the gradient at the inputs of this layer."""
        return (Y - T) / Y.shape[0]

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    def get_cost(self, Y, T):
        """Return the cost at the output of this output layer."""
        dim_ = 0
        return - np.multiply(T, np.log(Y)).sum() / Y.shape[dim_]

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    def softmax(self, z):
        return np.exp(z) / np.sum(np.exp(z), axis=1, keepdims=True)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    -
#  Neural Network Builder
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    -
class neural_network:
    """
    This class builds a neural network for each datasets (according to what
       the user choose)
      1-Cyst
      2-MNIST
      3-CIFAR
```

```
    According to what configuration the user chose, the output will be
        accuracy percent and cost/loss graph
    """

    def __init__(self):
        pass

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    def get_data_Cyst(self, conf):
        """
        Based on Cyst dataset. Define the network, train it and output
            accuracy percent and cost/loss graph.
        Args:
            conf:

        Returns:

        """
        # Load the raw data
        (all_images, image_class) = self.loadImages()

        # test / train split
        X_, X_test, y_, y_test = \
            train_test_split(all_images, image_class, test_size=0.20,
                random_state=42)

        X_train, X_val, y_train, y_val = \
            train_test_split(X_, y_, test_size=0.20, random_state=42)

        print "Total: ", len(all_images), "Train ", str(len(X_train)), ", Val:
            " \
            , len(X_val) , ", Test: ", len(X_test)

        # Normalize the data: subtract the mean image
        mean_image = np.mean(X_train, axis=0)
        X_train -= mean_image
        X_val -= mean_image
        X_test -= mean_image

        layers = []
        self.define_network(layers, X_train, y_train, conf)

        nb_of_iterations, nb_of_batches, minibatch_costs, training_costs,
            validation_costs = \
               self.train_network(layers, X_train, X_val, y_train, y_val, conf
                   )

        self.display_plot(nb_of_iterations, nb_of_batches, minibatch_costs,
            training_costs, validation_costs)

        self.get_data_results(layers, X_test, y_test)

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    def get_data_CIFAR(self, conf):
        """
        Based on CIFAR dataset. Define the network, train it and output
            accuracy percent and cost/loss graph.
        Args:
```

```
            conf:

        Returns:

        """
        image_list = []
        label_list = []
        data_train = self.unpickle('cifar-10-batches-py/data_batch_1')
        data_test = self.unpickle('cifar-10-batches-py/test_batch')
        image_class = np.concatenate((np.zeros([data_train['data'].shape[0], 1
            ]),
                                      np.ones([data_test['data'].shape[0], 1])
                                          ))
        # image_class = np.concatenate(d['data'])
        all_images = np.concatenate((data_train['data'], data_test['data']))
        # test_label_list = d['labels']

        # all_images = np.concatenate(image_list) / np.float32(255)
        # image_class = np.concatenate(label_list)

        X_train, X_test, y_train, y_test = cross_validation.train_test_split
            (all_images, image_class, test_size=0.20, random_state=42)

        X_val, X_test, y_val, y_test = cross_validation.train_test_split
            (X_test, y_test, test_size=0.20, random_state=42)

        layers = []
        self.define_network(layers, X_train, y_train, conf)

        nb_of_iterations, nb_of_batches, minibatch_costs, training_costs,
            validation_costs = \
               self.train_network(layers, X_train, X_val, y_train, y_val, conf
                   )

        self.display_plot(nb_of_iterations, nb_of_batches, minibatch_costs,
            training_costs, validation_costs)

        self.get_data_results(layers, X_test, y_test)

    # -----------------------------------

    def unpickle(self, file):
        """
        Used to load CIFAR dataset ("unpickle" the dataset).
        Args:
            file:

        Returns:

        """
        import cPickle
        fo = open(file, 'rb')
        dict = cPickle.load(fo)
        fo.close()
        return dict

    # -----------------------------------

    def get_data_MNIST(self, conf):
        """
```

```
        Based on MNIST dataset. Define the network, train it and output
            accuracy percent and cost/loss graph.
        Args:
            conf:

        Returns:

        """
        #Here the global dataset_type
        # If we're here, then we will remember that it's MNIST
        global dataset_type
        dataset_type = 'MNIST'

        # load the data from scikit-learn
        digits = datasets.load_digits()

        # Load the targets
        # Note that the targets are stored as digits, these need to be
        #   converted to one-hot-encoding for the output softmax layer.
        T = np.zeros((digits.target.shape[0],10))
        T[np.arange(len(T)), digits.target] += 1

        # Divide the data into a train and test set.
        X_train, X_test, T_train, T_test = cross_validation.train_test_split(
            digits.data, T, test_size=0.4)

        # Divide the test set into a validation set and final test set.
        X_validation, X_test, T_validation, T_test = cross_validation.
            train_test_split(
            X_test, T_test, test_size=0.5)

        layers = []
        self.define_network(layers, X_train, T_train, conf)

        nb_of_iterations, nb_of_batches, minibatch_costs, training_costs,
            validation_costs = \
            self.train_network(layers, X_train, X_validation, T_train,
                T_validation, conf)

        self.display_plot(nb_of_iterations, nb_of_batches, minibatch_costs,
            training_costs, validation_costs)

        self.get_data_results(layers, X_test, T_test)

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    def define_network(self, layers, X_train, T_train, conf):
        """
        Build the neural network according to the configuration
        Args:
            layers:
            X_train:
            T_train:
            conf:

        Returns:

        """
        # Number of neurons in the first hidden-layer
        hidden_neurons_1 = conf['hidden_neurons_1'] # Configuration 0
```

```
    # Number of neurons in the second hidden-layer
    hidden_neurons_2 = conf['hidden_neurons_2']

    # Create the model
    # layers = [] # Define a list of layers
    # Add first hidden layer
    layers.append(LinearLayer(X_train.shape[1], hidden_neurons_1, conf
        ['rate']))
    layers.append(LogisticLayer())
    # Add second hidden layer
    layers.append(LinearLayer(hidden_neurons_1, hidden_neurons_2, conf
        ['rate']))
    layers.append(LogisticLayer())
    # Add output layer
    layers.append(LinearLayer(hidden_neurons_2, T_train.shape[1], conf
        ['rate']))
    layers.append(SoftmaxOutputLayer())

    return layers

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def train_network(self, layers, X_train, X_test, T_train, T_validation,
    conf):
    """
    Train the network according to the dataset and configuration
    Args:
        layers:
        X_train:
        X_test:
        T_train:
        T_validation:
        conf:

    Returns:

    """
    # Create the minibatches

    # Approximately 25 samples per batch
    batch_size = conf['batch_size']

    nb_of_batches = X_train.shape[0] / batch_size # Number
    XT_batch = zip(
        np.array_split(X_train, nb_of_batches, axis=0), # X samples
        np.array_split(T_train, nb_of_batches, axis=0)) # Y targets

    # Perform backpropagation

    # initalize some lists to store the cost for future analysis
    minibatch_costs = []
    training_costs = []
    validation_costs = []
    activations = []

    # Train for a maximum of 300 iterations
    # max_nb_of_iterations = 300
    max_nb_of_iterations = conf['max_nb_of_iterations']
```

```
        # Gradient descent learning rate
        learning_rate = conf['learning_rate']

        # Train for the maximum number of iterations
        for iteration in range(max_nb_of_iterations):

            for X, T in XT_batch:  # For each minibatch sub-iteration
                print "Training iteration #", iteration, "..."
                activations = self.forward_step(X, layers)  # Get the
                    activations
                minibatch_cost = layers[-1].get_cost(activations[-1], T)  #
                    Get cost
                minibatch_costs.append(minibatch_cost)
                param_grads = self.backward_step(activations, T, layers)  #
                    Get the gradients
                self.update_params(layers, param_grads, learning_rate)  #
                    Update the parameters

            # Get full training cost for future analysis (plots)
            activations = self.forward_step(X_train, layers)
            train_cost = layers[-1].get_cost(activations[-1], T_train)
            training_costs.append(train_cost)
            # Get full validation cost
            activations = self.forward_step(X_test, layers)
            validation_cost = layers[-1].get_cost(activations[-1],
                T_validation)
            validation_costs.append(validation_cost)
            if len(validation_costs) > 3:
                # Stop training if the cost on the validation set doesn't
                    decrease
                #  for 3 iterations
                if validation_costs[-1] >= validation_costs[-2] >=
                    validation_costs[-3]:
                    break

        nb_of_iterations = iteration + 1  # The number of iterations that have
            been executed
        print "Number of iterations that have been executed: ",
            nb_of_iterations
        return nb_of_iterations, nb_of_batches, minibatch_costs,
            training_costs, validation_costs

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        - - -
    # forward
    #  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        - - -
    # Define the forward propagation step as a method.
    def forward_step(self, input_samples, layers):
        """
        Compute and return the forward activation of each layer in layers.
        Input:
            input_samples: A matrix of input samples (each row is an input
                vector)
            layers: A list of Layers
        Output:
            A list of activations where the activation at each index i+1
                corresponds to
            the activation of layer i in layers. activations[0] contains the
                input samples.
```

```
        """
        activations = [input_samples] # List of layer activations
        # Compute the forward activations for each layer starting from the
            first
        X = input_samples
        for layer in layers:
            Y = layer.get_output(X)  # Get the output of the current layer
            activations.append(Y)  # Store the output for future processing
            X = activations[-1]  # Set the current input as the activations of
                the previous layer
        return activations  # Return the activations of each layer

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        - - -
    # Backward step
    #  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        - - -
    # Define the backward propagation step as a method
    def backward_step(self, activations, targets, layers):
        """
        Perform the backpropagation step over all the layers and return the
            parameter gradients.
        Input:
            activations: A list of forward step activations where the
                activation at
                each index i+1 corresponds to the activation of layer i in
                    layers.
                activations[0] contains the input samples.
            targets: The output targets of the output layer.
            layers: A list of Layers corresponding that generated the outputs
                in activations.
        Output:
            A list of parameter gradients where the gradients at each index
                corresponds to
            the parameters gradients of the layer at the same index in layers.
        """
        param_grads = collections.deque()  # List of parameter gradients for
            each layer
        output_grad = None  # The error gradient at the output of the current
            layer
        # Propagate the error backwards through all the layers.
        #  Use reversed to iterate backwards over the list of layers.
        for layer in reversed(layers):
            Y = activations.pop()  # Get the activations of the last layer on
                the stack
            # Compute the error at the output layer.
            # The output layer error is calculated different then hidden layer
                error.

            if output_grad is None:
                input_grad = layer.get_input_grad(Y, targets)
            else:  # output_grad is not None (layer is not output layer)
                input_grad = layer.get_input_grad(Y, output_grad)
            # Get the input of this layer (activations of the previous layer)
            X = activations[-1]
            # Compute the layer parameter gradients used to update the
                parameters
            grads = layer.get_params_grad(X, output_grad)
            param_grads.appendleft(grads)
            # Compute gradient at output of previous layer (input of current
```

```
            layer):
        output_grad = input_grad
    return list(param_grads)  # Return the parameter gradients

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def update_params(layers, param_grads, learning_rate):
    """
    Function to update the parameters of the given layers with the given
        gradients
    by gradient descent with the given learning rate.
    Args:
        param_grads:
        learning_rate:

    Returns:

    """
    for layer, layer_backprop_grads in zip(layers, param_grads):
        for param, grad in itertools.izip(layer.get_params_iter(),
            layer_backprop_grads):
            # The parameter returned by the iterator point to the memory
                space of
            #  the original layer and can thus be modified inplace.
            param -= learning_rate * grad  # Update each parameter

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def display_plot(self, nb_of_iterations, nb_of_batches, minibatch_costs,
    training_costs, validation_costs):
    """
    Plot the cost/loss graph.
    Args:
        nb_of_iterations:
        nb_of_batches:
        minibatch_costs:
        training_costs:
        validation_costs:

    Returns:

    """
    # Plot the minibatch, full training set, and validation costs
    minibatch_x_inds = np.linspace(0, nb_of_iterations, num=
        nb_of_iterations*nb_of_batches)
    iteration_x_inds = np.linspace(1, nb_of_iterations, num=
        nb_of_iterations)
    # Plot the cost over the iterations
    plt.plot(minibatch_x_inds, minibatch_costs, 'k-', linewidth=0.5, label
        ='cost minibatches')
    plt.plot(iteration_x_inds, training_costs, 'r-', linewidth=2, label=
        'cost full training set')
    plt.plot(iteration_x_inds, validation_costs, 'b-', linewidth=3, label=
        'cost validation set')
    # Add labels to the plot
    plt.xlabel('iteration')
    plt.ylabel('$\\xi$', fontsize=15)
    plt.title('Decrease of cost over backprop iteration')
    plt.legend()
    x1,x2,y1,y2 = plt.axis()
```

```
    plt.axis((0,nb_of_iterations,0,2.5))
    plt.grid()
    print "See graph on the the opened window"
    plt.show()

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def get_data_results(self, layers, X_test, T_test):
    """
    Return the accuracy percent.
    Args:
        layers:
        X_test:
        T_test:

    Returns:

    """
    from sklearn import metrics

    y_true = np.argmax(T_test, axis=1)  # Get the target outputs
    activations = self.forward_step(X_test, layers)  # Get activation of
        test samples
    y_pred = np.argmax(activations[-1], axis=1)  # Get the predictions
        made by the network
    test_accuracy = metrics.accuracy_score(y_true, y_pred)  # Test set
        accuracy
    print('The accuracy on the test set is {:.2f}'.format(test_accuracy))

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def update_params(self, layers, param_grads, learning_rate):
    """
    Function to update the parameters of the given layers with the given
        gradients
    by gradient descent with the given learning rate.
    """
    for layer, layer_backprop_grads in zip(layers, param_grads):
        for param, grad in itertools.izip(layer.get_params_iter(),
            layer_backprop_grads):
            # The parameter returned by the iterator point to the memory
                space of
            #  the original layer and can thus be modified inplace.
            param -= learning_rate * grad  # Update each parameter

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def loadImagesByList(self, file_pattern):
    """
    Used by CIFAR.
    Loads the data batch files.
    Return list of images
    """
    import glob
    image_list = map(Image.open, glob.glob(file_pattern))
    imSizeAsVector = image_list[0].size[0] * image_list[0].size[1]
    images = np.zeros([len(image_list), imSizeAsVector])
    for idx, im in enumerate(image_list):
        images[idx,:] = np.array(im, np.uint8).reshape(imSizeAsVector,1).T
    return images
```

```
    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    def loadImages(self):
        """
        Used by CIFAR.
        Loads the data batch files.
        Return list of all images and image class.
        """
        OK_file_pattern = 'image_patches/*OK*axial.png'
        Cyst_file_pattern = 'image_patches/*Cyst*axial.png'

        # OK-images
        OK_image = self.loadImagesByList(OK_file_pattern)

        # Cyst-images
        Cyst_image = self.loadImagesByList(Cyst_file_pattern)

        # concatenate the two types
        image_class = np.concatenate( (np.zeros([OK_image.shape[0],1]) ,
                                       np.ones([Cyst_image.shape[0],1]) ) )
        all_images = np.concatenate((OK_image, Cyst_image))
        return (all_images, image_class)

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    def get_data_results(self, layers, X_test, T_test):
        from sklearn import metrics

        y_true = np.argmax(T_test, axis=1)  # Get the target outputs
        activations = self.forward_step(X_test, layers)  # Get activation of
            test samples
        y_pred = np.argmax(activations[-1], axis=1)  # Get the predictions
            made by the network
        test_accuracy = metrics.accuracy_score(y_true, y_pred)  # Test set
            accuracy
        print('The accuracy on the test set is {:.2f}'.format(test_accuracy))

if __name__ == "__main__":
    print "Enter a set's number: 1 for Cyst, 2 for CIFAR, 3 for MNIST"
    set_op = input()
    print "You have an option to choose different rates, hidden neurons, batch
        sizes and maximum number of iterations."
    print "See README for more details."
    print "Enter configuration number wanted, from 1 to 6: "
    conf_op = input()

    nn = neural_network()
    conf = {}

    configuration_no_1 = {'hidden_neurons_1': 20, 'hidden_neurons_2': 20,
        'batch_size': 25, 'max_nb_of_iterations': 300,
              'learning_rate': 0.1, 'rate': 0.1}
    configuration_no_2 = {'hidden_neurons_1': 50, 'hidden_neurons_2': 50,
        'batch_size': 25, 'max_nb_of_iterations': 300,
              'learning_rate': 0.1, 'rate': 0.1}
    configuration_no_3 = {'hidden_neurons_1': 20, 'hidden_neurons_2': 20,
        'batch_size': 50, 'max_nb_of_iterations': 200,
              'learning_rate': 0.3, 'rate': 0.1}
    configuration_no_4 = {'hidden_neurons_1': 20, 'hidden_neurons_2': 20,
```

```
    'batch_size': 25, 'max_nb_of_iterations': 3,
          'learning_rate': 0.3, 'rate': 0.1}
configuration_no_5 = {'hidden_neurons_1': 20, 'hidden_neurons_2': 20,
    'batch_size': 25, 'max_nb_of_iterations': 300,
          'learning_rate': 0.8, 'rate': 0.1}
configuration_no_6 = {'hidden_neurons_1': 20, 'hidden_neurons_2': 20,
    'batch_size': 25, 'max_nb_of_iterations': 300,
          'learning_rate': 0.8, 'rate': 0.5}



if conf_op == 0:
    configuration = configuration_no_1
elif conf_op == 1:
    configuration = configuration_no_2
elif conf_op == 2:
    configuration = configuration_no_3
elif conf_op == 3:
    configuration = configuration_no_4
elif conf_op == 4:
    configuration = configuration_no_5
elif conf_op == 5:
    configuration = configuration_no_6


if set_op == 1:
    nn.get_data_Cyst(configuration)
elif set_op == 2:
    nn.get_data_CIFAR(configuration)
elif set_op == 3:
    nn.get_data_MNIST(configuration)
```