```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
- - -
 #Define the non-linear functions used
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
- - -
import glob
import itertools

import sys
from PIL import Image
import collections
from sklearn import cross_validation, datasets, metrics
from sklearn.cross_validation import train_test_split
import matplotlib.pyplot as plt
import numpy as np


def logistic(z):
 #    prevent too small neither too high values
    z[z > 700] = 700
    z[z < -700] = -700
    return 1 / (1 + np.exp(-z))


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


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


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


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

    def __init__(self, n_in, n_out, rate=0.4):
"""        Initiate hidden layer parameters"""
        self.W = np.random.randn(n_in, n_out) * rate
        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=None, T=None):
"""        Return the gradient at the inputs of this layer."""
        return output_grad.dot(self.W.T)


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

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

```
    def get_input_grad(self, Y, output_grad=None, T=None):
"""        Return the gradient at the inputs of this layer."""
        return np.multiply(logistic_deriv(Y), output_grad)


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

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

    def get_input_grad(self, Y, output_grad=None, T=None):
"""        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."""
        return - np.multiply(T, np.log(Y)).sum() / Y.shape[0]


- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
- - -
 #forward
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  #
- - - -
 #Define the forward propagation step as a method.
def forward_step(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(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, T=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


- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
- - -
 #update_params
```

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
- - - -
 #Define a method to update the parameters
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.
"""
    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


- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
- - -
  #Loading inputs methods
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
- - -
def loadImagesByList(file_pattern):
    image_list = map(Image.open, glob.glob(file_pattern))
    imSizeVector = image_list[0].size[0] * image_list[0].size[1]
    images = np.zeros([len(image_list), imSizeVector])
    for idx, im in enumerate(image_list):
        images[idx, :] = np.array(im, np.uint8) \
.            reshape(imSizeVector, 1).T
    return images


def loadImages():
 #    Medical images
    OK_files_pattern = '*OK*axial.png'
    Cyst_files_pattern = '*Cyst*axial.png'
 #    ok images
    OK_images = loadImagesByList(OK_files_pattern)

 #    Cyst image
    Cyst_images = loadImagesByList(Cyst_files_pattern)
 #    concatenate the two types
    image_class = np.concatenate((np.zeros([OK_images.shape[0], 1]),
                              np.ones([Cyst_images.shape[0], 1])))

    all_images = np.concatenate((OK_images, Cyst_images))
    return all_images, image_class

 #CIFAR-10 data
def cifar(file):
    import cPickle
    fo = open(file, 'rb')
    dict = cPickle.load(fo)
    fo.close()
    return dict
```

```
 #sklearn images
def get_splitted_data(set):
 #    load raw data
    (all_images, image_class) = 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)

 #    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

    return X_train, y_train, X_val, y_val, X_test, y_test

 #    return X_train, y_train.T[0], X_val, y_val.T[0], X_test,
y_test.T[0]


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

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

 #    divide the data into train and 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
(

    return X_train, X_test, T_train, T_test, X_validation, T_validation



def load_cifar():
    image_list = []
    label_list = []
    d = cifar('cifar-10-batches-py\\data_batch_1')
    d_test = cifar('cifar-10-batches-py/test_batch')
    image_class = np.concatenate((np.zeros([d['data'].shape[0], 1]),
```

```
                                  np.ones([d_test['data'].shape[0], 1])))
 #    image_class = np.concatenate(d['data'])
    all_images = np.concatenate((d['data'], d_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)
    return X_train, y_train, X_val, y_val, X_test, y_test


class Run():
    def __init__(self, set, configuration):
        self.batch_size = 5
        self.hidden_neurons_1 = 20  # Number of neurons in the first
hidden-layer
        self.hidden_neurons_2 = 20  # Number of neurons in the second
hidden-layer
 #        Create the model
        self.layers = []  # Define a list of layers
        self.set_method(set)
        self.set_configuration(configuration)

    def set_running_method(self, set, configuration):
        pass

    def set_configuration(self, configuration):

        if configuration is 1:
            self.rate = 0.1
            self.batch_size *= 2
            self.learning_rate = 0.04
            loop = 1
        if configuration is 2:
            self.rate = 0.4
            self.batch_size *= 5
            self.learning_rate = 0.1
            loop = 4
        if configuration is 3:
            self.rate = 0.6
            self.batch_size *= 8
            self.learning_rate = 0.4
            loop = 10
 #            Add first hidden layer
        self.layers.append(LinearLayer(self.X_train.shape[1],
self.hidden_neurons_1, self.rate))
        self.layers.append(LogisticLayer())
        for i in range(0, loop):
 #            Add second hidden layer
            self.layers.append(LinearLayer(self.hidden_neurons_1,
self.hidden_neurons_2, self.rate))
```

```
            self.layers.append(LogisticLayer())
 #        Add output layer

        self.layers.append(LinearLayer(self.hidden_neurons_2,
self.T_train.shape[1], self.rate))
        self.layers.append(SoftmaxOutputLayer())

    def set_method(self, set):

        if set is 1:
            self.X_train, self.T_train, self.X_validation,
self.T_validation, self.X_test, self.T_test \
 =                get_splitted_data(set)
        if set is 2:
            self.batch_size = 50
            self.X_train, self.T_train, self.X_validation,
self.T_validation, self.X_test, self.T_test =  load_cifar()
        if set is 3:
            self.X_validation, self.X_train, self.T_validation,
self.T_train, self.X_test, self.T_test = scikit()

    def learn(self):

 #        create batches
 #        approx 25 samples per batch
        print "learning..."
 #        number of batches
        nb_of_batches = self.X_train.shape[0] / self.batch_size
        XT_batches = zip(
            np.array_split(self.X_train, nb_of_batches, axis=0),
            np.array_split(self.T_train, nb_of_batches, axis=0)
(

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
- - - - - - -
 #        train the network
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
- - - - - - -
 #        Perform backpropagation

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

        max_nb_of_iterations = 300  # Train for a maximum of 300
iterations

 #        Train for the maximum number of iterations
        for iteration in range(max_nb_of_iterations):
            for X, T in XT_batches:  # For each minibatch sub-iteration
                activations = forward_step(X, self.layers)  # Get the
activations
                minibatch_cost = self.layers[-1].get_cost(activations[-
1], T)  # Get cost
                minibatch_costs.append(minibatch_cost)
                param_grads = backward_step(activations, T, self.layers)
# Get the gradients
```

```
                update_params(self.layers, param_grads,
self.learning_rate)  # Update the parameters
 #            Get full training cost for future analysis (plots)
            activations = forward_step(self.X_train, self.layers)
            train_cost = self.layers[-1].get_cost(activations[-1],
self.T_train)
            training_costs.append(train_cost)
 #            Get full validation cost
            activations = forward_step(self.X_validation, self.layers)
            validation_cost = self.layers[-1].get_cost(activations[-1],
self.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

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
- - - - - - - -
 #        plot the network performance
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
- - - - - - - -
 #        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)

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
- - - - - - - -
 #        Get results of test data
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
- - - - - - - -

        y_true = np.argmax(self.T_test, axis=1)  # Get the target outputs
        activations = forward_step(self.X_test, self.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))
 #        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()
        plt.show()


- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
- - -
 #Define the network
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
- - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
- - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
- - -
  #MAIN
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
- - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
- - -
if __name__ == "__main__":
    set_num = 0
    configuration_num = 0
    set_str = """Choose the Set to Learn:
 - 1        Cyst and OK image
 - 2        CIFAR-10
 - 3        SK-LEARN
""" <
    conf_str = """Select Configuration:
 - 1        Slow Rate Learn
 - 2        Medium Rate Learn
 - 3        Fast Learning Mode
""" <
    set_map = {
" :1        Cyst and OK image",
" :2        CIFAR-10",
" :3        SK-LEARN"
{
    conf_map = {
" :1        Slow Rate Learn",
" :2        Medium Rate Learn",
" :3        Fast Learning Mode"
{
    if len(sys.argv) == 3:
        set_in = int(sys.argv[1])
        configuration_in = int(sys.argv[2])
    else:
        set_in = input(set_str)
        configuration_in = input(conf_str)
    try:
        set_num = int(set_in)
        configuration_num = int(configuration_in)
    except:
        print "Need Numeric Arguments"
        raise SystemExit
```

```
    if set_num <= 0 or configuration_num <= 0:
        print "wrong args"
        raise SystemExit

    print "Set to Learn {0} with {1}".format(set_map[set_num],
conf_map[configuration_num])
    run = Run(set_num, configuration_num)
    run.learn()
```