In [1]:
```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as np
import scipy.signal as sg
from utils import *
```

In [2]:

```
# superclass of modules
class Module:
    """
    Module is a super class. It could be a single layer, or a multilayer perce
ptron.
    """

    def __init__(self):
        self.train = True
        return

    def forward(self, _input):
        """
        h = f(z); z is the input, and h is the output.

        Inputs:
        _input: z

        Returns:
        output h
        """
        pass

    def backward(self, _input, _gradOutput):
        """
        Compute:
        gradient w.r.t. _input
        gradient w.r.t. trainable parameters

        Inputs:
        _input: z
        _gradOutput: dL/dh

        Returns:
        gradInput: dL/dz
        """
        pass

    def parameters(self):
        """
        Return the value of trainable parameters and its corresponding gradien
t (Used for grandient descent)

        Returns:
        params, gradParams
        """
```

```python
        pass

    def training(self):
        """
        Turn the module into training mode.(Only useful for Dropout layer)
        Ignore it if you are not using Dropout.
        """
        self.train = True

    def evaluate(self):
        """
        Turn the module into evaluate mode.(Only useful for Dropout layer)
        Ignore it if you are not using Dropout.
        """
        self.train = False
```

In [3]:
```
class Sequential(Module):
    """
    Sequential provides a way to plug layers together in a feed-forward manne
r.
    """
    def __init__(self):
        Module.__init__(self)
        self.layers = [] # layers contain all the layers in order

    def add(self, layer):
        self.layers.append(layer) # Add another layer at the end

    def size(self):
        return len(self.layers) # How many layers.

    def forward(self, _input):
        """
        Feed forward through all the layers, and return the output of the last
 layer
        """
        # self._inputs saves the input of each layer
        # self._inputs[i] is the input of i-th layer
        self._inputs = [_input]

        for i in range(self.size()):
            self._inputs.append(self.layers[i].forward(self._inputs[i]))

        # The last element of self._inputs is the output of last layer
        self._output = self._inputs[-1]
        return self._output

    def backward(self, _input, _gradOutput):
        """
        Backpropogate through all the layers using chain rule.
        """
        # self._gradInputs[i] is the gradient of loss w.r.t. the input of i-th
 layer
        self._gradInputs = [None] * (self.size() + 1)
        self._gradInputs[self.size()] = _gradOutput
```

```
        for i in range(self.size(), 0, -1):
            self._gradInputs[i-1] = self.layers[i-1].backward(self._inputs[i-
1], self._gradInputs[i])

        self._gradInput = self._gradInputs[0]
        return self._gradInput

    def parameters(self):
        """
        Return trainable parameters and its corresponding gradient in a nested
 list
        """
        params = []
        gradParams = []
        for m in self.layers:
            _p, _g = m.parameters()
            if _p is not None:
                params.append(_p)
                gradParams.append(_g)
        return params, gradParams

    def training(self):
        """
        Turn all the layers into training mode
        """
        Module.training(self)
        for m in self.layers:
            m.training()

    def evaluate(self):
        """
        Turn all the layers into evaluate mode
        """
        Module.evaluate(self)
        for m in self.layers:
            m.evaluate()
```

In [4]:

```
class Convolutional(Module):
    """
    Convolutional layer
    """
    def __init__(self, inputLength, inputDepth, filterLength, filterDepth):
        Module.__init__(self)
        # Initalization
        stdv = 1./np.sqrt(inputLength)

        self.weight = np.random.uniform(-stdv, stdv, (filterLength,
inputDepth, filterDepth))
        self.gradWeight = np.ndarray((filterLength, inputDepth, filterDepth))
        self.bias = np.random.uniform(-stdv, stdv, filterDepth)
        self.gradBias = np.ndarray(filterDepth)

    def forward(self, _input):
        """
        output = W * filterRegion + b convolution over input

        _input:
```

```
        N x inputLength x inputDepth matrix

        """
        inputDepth = self.weight.shape[1]
        filterDepth = self.weight.shape[2]

        _input = _input.reshape(_input.shape[0],-1,inputDepth)

        self._output = []
        for i in range(filterDepth):
            self._output.append(sg.correlate(_input, np.expand_dims(np.take(se
lf.weight,i,2),0), 'valid') + self.bias[i])
        self._output = np.array(self._output)
        self._output =
self._output.reshape(self._output.shape[1],self._output.shape[2],self._output.s
hape[0])

        return self._output

    def backward(self, _input, _gradOutput):
        """
        _input:
        N x inputLength x inputDepth matrix
        _gradOutputSize:
        N x outputLength x outputDepth matrix
        """
        filterLength = self.weight.shape[0]
        inputDepth = self.weight.shape[1]
        filterDepth = self.weight.shape[2]

        _input = _input.reshape(_input.shape[0],-1,inputDepth)
        _gradOutput = _gradOutput.reshape(_gradOutput.shape[0],-1,filterDepth)

        for i in range(filterDepth):
            self.gradWeight[:,:,i] = sg.correlate(_input, np.expand_dims(np.ta
ke(_gradOutput,i,2),2), 'valid')

        self.gradBias = np.sum(_gradOutput, axis=(0,1))

        self._gradInput = []
        for i in range(inputDepth):
            self._gradInput.append(sg.correlate(np.pad(_gradOutput, ((0,0),(fi
lterLength-1,filterLength-1),(0,0)), 'constant'),
np.expand_dims(np.take(self.weight,i,1),0), 'valid'))
        self._gradInput = np.array(self._gradInput)
        self._gradInput = self._gradInput.reshape(self._gradInput.shape[1],sel
f._gradInput.shape[2],self._gradInput.shape[0])

        return self._gradInput

    def parameters(self):
        """
        Return weight and bias and their g
        """
        return [self.weight, self.bias], [self.gradWeight, self.gradBias]
```

In [5]:
```
class FullyConnected(Module):
    """
    Fully connected layer
    """
    def __init__(self, inputSize, outputSize):
        Module.__init__(self)
        # Initalization
        stdv = 1./np.sqrt(inputSize)

        self.weight = np.random.uniform(-stdv, stdv, (inputSize, outputSize))
        self.gradWeight = np.ndarray((inputSize, outputSize))
        self.bias = np.random.uniform(-stdv, stdv, outputSize)
        self.gradBias = np.ndarray(outputSize)

    def forward(self, _input):
        """
        output = W * input + b

        _input:
        N x inputSize matrix

        """
        _input = _input.reshape(_input.shape[0],-1)

        self._output = np.dot(_input, self.weight) + self.bias
        return self._output

    def backward(self, _input, _gradOutput):
        """
        _input:
        N x inputSize matrix
        _gradOutputSize:
        N x outputSize matrix
        """
        _input = _input.reshape(_input.shape[0],-1)
        _gradOutput = _gradOutput.reshape(_gradOutput.shape[0],-1)

        self.gradWeight = np.dot(_input.T, _gradOutput)
        self.gradBias = np.sum(_gradOutput, axis=0)

        self._gradInput = np.dot(_gradOutput, self.weight.T)
        return self._gradInput

    def parameters(self):
        """
        Return weight and bias and their g
        """
        return [self.weight, self.bias], [self.gradWeight, self.gradBias]
```

In [6]:
```python
class ReLU(Module):
    """
    ReLU activation, not trainable.
    """
    def __init__(self):
        Module.__init__(self)
        return

    def forward(self, _input):
        """
        output = max(0, input)

        _input:
        N x d matrix
        """
        #_input = _input.reshape(_input.shape[0],-1)

        self._output = np.maximum(0, _input)
        return self._output

    def backward(self, _input, _gradOutput):
        """
        gradInput = gradOutput * mask
        mask = _input > 0

        _input:
        N x d matrix

        _gradOutput:
        N x d matrix
        """
        #_input = _input.reshape(_input.shape[0],-1)
        #_gradOutput = _gradOutput.reshape(_gradOutput.shape[0],-1)

        self._gradInput = _gradOutput * (_input > 0)
        return self._gradInput

    def parameters(self):
        """
        No trainable parametersm, return None
        """
        return None, None
```

In [7]:
```
class SoftMaxLoss(object):
    def __init__(self):
        return

    def forward(self, _input, _label):
        """
        Softmax and cross entropy loss layer. Should return a scalar, since i
t's a
        loss. (It's almost identical to what in hw2)

        _input: N x C
        _labels: N x C, one-hot

        Returns: loss (scalar)
        """
        self._output = -np.sum(_label * (_input -
np.log(np.sum(np.exp(_input), axis=1)).reshape(1, -1).T))
        return self._output

    def backward(self, _input, _label):
        self._gradInput = np.exp(_input)/np.sum(np.exp(_input), axis=1).reshap
e(1, -1).T - _label
        return self._gradInput
```

In [8]:
```
# Test softmaxLoss, the relative error should be small enough
def test_sm():
    crit = SoftMaxLoss()
    gt = np.zeros((3, 10))
    gt[np.arange(3), np.array([1,2,3])] = 1
    x = np.random.random((3,10))
    def test_f(x):
        return crit.forward(x, gt)

    crit.forward(x, gt)

    gradInput = crit.backward(x, gt)
    gradInput_num = numeric_gradient(test_f, x, 1, 1e-6)
    #print(gradInput)
    #print(gradInput_num)
    print(relative_error(gradInput, gradInput_num, 1e-8))

test_sm()
```

```
4.64713032996e-09
```

In [9]:
```python
# Test modules, all the relative errors should be small enough
def test_module(model):

    model.evaluate()

    crit = TestCriterion()
    gt = np.random.random((3,10))
    x = np.random.random((3,10))
    def test_f(x):
        return crit.forward(model.forward(x), gt)

    gradInput = model.backward(x, crit.backward(model.forward(x), gt))
    gradInput_num = numeric_gradient(test_f, x, 1, 1e-6)
    print(relative_error(gradInput, gradInput_num, 1e-8))

# Test fully connected
model = FullyConnected(10, 10)
test_module(model)

# Test ReLU
model = ReLU()
test_module(model)

# Test Sequential
model = Sequential()
model.add(FullyConnected(10, 10))
model.add(ReLU())
#model.add(Dropout())
test_module(model)
```

```
4.75853713145e-09
7.36022825478e-10
1.75509770151e-09
```

In [10]:

```python
# Test gradient descent, the loss should be lower and lower
trainX = np.random.random((10,25,15))

model = Sequential()
model.add(Convolutional(25,15, 6,20))
model.add(ReLU())
model.add(Convolutional(20,20, 11,5))
model.add(FullyConnected(50,1))

crit = TestCriterion()

it = 0
state = None
while True:
    output = model.forward(trainX)
    loss = crit.forward(output, None)
    if it % 100 == 0:
        print(loss)
    doutput = crit.backward(output, None)
    model.backward(trainX, doutput)
    params, gradParams = model.parameters()
    sgdmom(params, gradParams, 0.0005, 0.8)
    if it > 1000:
        break
    it += 1
```

```
0.411563871173
0.15327683183
0.117801346354
0.0804396823411
0.0675119314006
0.051551482172
0.0353256828849
0.0225411762008
0.0234306934355
0.0212756206
0.0122138643986
```

Now we start to work on real data.

In [11]:

```
import MNIST_utils
data_fn = "CLEAN_MNIST_SUBSETS.h5"

# We only consider large set this time
print("Load large trainset.")
Xlarge,Ylarge = MNIST_utils.load_data(data_fn, "large_train")
print(Xlarge.shape)
print(Ylarge.shape)

print("Load valset.")
Xval,Yval = MNIST_utils.load_data(data_fn, "val")
print(Xval.shape)
print(Yval.shape)
```

```
Load large trainset.
(7000L, 576L)
(7000L, 10L)
Load valset.
(2000L, 576L)
(2000L, 10L)
```

In [12]:
```
def predict(X, model):
    """
    Evaluate the soft predictions of the model.
    Input:
    X : N x d array (no unit terms)
    model : a multi-layer perceptron
    Output:
    yhat : N x C array
        yhat[n][:] contains the score over C classes for X[n][:]
    """
    return model.forward(X)

def error_rate(X, Y, model):
    """
    Compute error rate (between 0 and 1) for the model
    """
    model.evaluate()
    res = 1 - (model.forward(X).argmax(-1) == Y.argmax(-1)).mean()
    model.training()
    return res

from copy import deepcopy

def runTrainVal(X,Y,model,Xval,Yval,trainopt):
    """
    Run the train + evaluation on a given train/val partition
    trainopt: various (hyper)parameters of the training procedure
    During training, choose the model with the lowest validation error. (early
 stopping)
    """

    eta = trainopt['eta']

    N = X.shape[0] # number of data points in X
```

```
    # Save the model with lowest validation error
    minValError = np.inf
    saved_model = None

    shuffled_idx = np.random.permutation(N)
    start_idx = 0
    for iteration in range(trainopt['maxiter']):
        if iteration % int(trainopt['eta_frac'] * trainopt['maxiter']) == 0:
            eta *= trainopt['etadrop']
        # form the next mini-batch
        stop_idx = min(start_idx + trainopt['batch_size'], N)
        batch_idx = range(N)[int(start_idx):int(stop_idx)]
        bX = X[shuffled_idx[batch_idx],:]
        bY = Y[shuffled_idx[batch_idx],:]

        score = model.forward(bX)
        loss = crit.forward(score, bY)
        # print(loss)
        dscore = crit.backward(score, bY)
        model.backward(bX, dscore)

        # Update the data using
        params, gradParams = model.parameters()
        sgd(params, gradParams, eta, weight_decay = trainopt['lambda'])
        start_idx = stop_idx % N

        if (iteration % trainopt['display_iter']) == 0:
            #compute train and val error; multiply by 100 for readability (mak
e it percentage points)
            trainError = 100 * error_rate(X, Y, model)
            valError = 100 * error_rate(Xval, Yval, model)
            print('{:8} batch loss: {:.3f} train error: {:.3f} val error: {:.3
f}'.format(iteration, loss, trainError, valError))

            if valError < minValError:
                saved_model = deepcopy(model)
                minValError = valError

    return saved_model, minValError, trainError
```

In [13]:
```
def build_model(input_size, hidden_size, output_size, activation_func = 
'ReLU', dropout = 0):
    """
    Build the model:
    input_size: the dimension of input data
    hidden_size: the dimension of hidden vector
    output_size: the output size of final layer.
    activation_func: ReLU, Logistic, Tanh, etc. (Need to be implemented by you
rself)
    dropout: the dropout rate: if dropout == 0, this is equivalent to no dropo
ut
    """
    model = Sequential()
    model.add(Convolutional(input_size,1, hidden_size,1))
    model.add(FullyConnected((input_size-hidden_size+1), output_size))
    model.add(ReLU())
    model.add(FullyConnected(output_size, output_size))

    return model
```

In [14]:
```
# -- training options
trainopt = {
    'eta': .001,   # initial learning rate
    'maxiter': 2500,   # max number of iterations (updates) of SGD
    'display_iter': 500,  # display batch loss every display_iter updates
    'batch_size': 100,
    'etadrop': .5, # when dropping eta, multiply it by this number (e.g., .5 m
eans halve it)
    'eta_frac': .25  #
}

NFEATURES = Xlarge.shape[1]

# we will maintain a record of models trained for different values of lambda
# these will be indexed directly by lambda value itself
trained_models = dict()

# set the (initial?) set of lambda values to explore
lambdas = np.array([0, 0.001, 0.01, 0.1])
hidden_sizes = np.array([10])

for lambda_ in lambdas:
    for hidden_size_ in hidden_sizes:
        trainopt['lambda'] = lambda_
        model = build_model(NFEATURES, hidden_size_, 10, dropout = 0)
        crit = SoftMaxLoss()
        # -- model trained on large train set
        trained_model,valErr,trainErr = runTrainVal(Xlarge, Ylarge, model, Xva
l, Yval, trainopt)
        trained_models[(lambda_, hidden_size_)] = {'model': trained_model, "va
l_err": valErr, "train_err": trainErr }
        print('train set model: -> lambda= %.4f, train error: %.2f, val error:
 %.2f' % (lambda_, trainErr, valErr))

best_trained_lambda = 0.
best_trained_model = None
best_trained_val_err = 100.
for (lambda_, hidden_size_), results in trained_models.items():
    print('lambda= %.4f, hidden size: %5d, val error: %.2f' %(lambda_, hidden_
size_, results['val_err']))
    if results['val_err'] < best_trained_val_err:
        best_trained_val_err = results['val_err']
        best_trained_model = results['model']
        best_trained_lambda = lambda_

print("Best train model val err:", best_trained_val_err)
print("Best train model lambda:", best_trained_lambda)
```