OR 610 – Assignment 03

(Solution to Question 2-5)

Abhishek Shambhu

Code with commented outputs:

```
# -*- coding: utf-8 -*-
"""
Created on Sat Oct  5 19:31:12 2019

@author: abhishek shambhu
"""

'''
Question:

2. Organic CNN Replace the fully connected layer from the organic DL the convolutional layer.

3. Initialization Implement four different initialization techniques and apply them to your CNN
model
   • Zeros initialization
   • Random initialization. This initializes the weights to some random values.
   • Xavier initialization, which scales the variance of the inputs to each layer are scaled to
     have variance of sqrt(1./layers_dims[l-1]).
   • He initialization. This initializes the weights to random values scaled according to a paper by
He et al., 2015.
     Similar to Xavier you need to scale the inputs so they have the variance
sqrt(2./layers_dims[l-1])

4. Optimization Implement ADAM modification to the SGD and apply it to MNIST classification.

5. Dropout Implement forward and backward propagation with dropout. Apply to MNIST
classification problem.
'''

'''
Solution:
  Questions 2,3,4 and 5 are implemented together since it is the same MNIST dataset
'''
import struct
import numpy as np
import warnings
warnings.filterwarnings("ignore")

def read_idx(filename):
  with open(filename, 'rb') as f:
     zero, data_type, dims = struct.unpack('>HBB', f.read(4))
     shape = tuple(struct.unpack('>I', f.read(4))[0] for d in range(dims))
```

```
        return np.frombuffer(f.read(), dtype=np.uint8).reshape(shape)

x_train = read_idx(r'D:\sem3\or610\HW2\train-images.idx3-ubyte')
y_train = read_idx(r'D:\sem3\or610\HW2\train-labels.idx1-ubyte')
x_test = read_idx(r'D:\sem3\or610\HW2\t10k-images.idx3-ubyte')
y_test = read_idx(r'D:\sem3\or610\HW2\t10k-labels.idx1-ubyte')

x_train = x_train.astype(np.float)/255.
x_test = x_test.astype(np.float)/255.

class CNN:
    """ Defining a Convolutional Neural Network Model with l2 regularization and applying it to
MNIST Dataset for classification
    Parameters:
        l2: l2 regularization rate
        n: number of iterations
        l_rate: learning rate
        act_func: activation function used on the hidden layer
        encode: One hot encode target values to binary form if not
        size: size of each batch (number of samples in each batch)
        kernel_depth: depth of kernel in convolutional layer
        kernelfilter_size: size of kernel filter
    """
    def __init__(self, n=1000, l_rate=0.05, act_func='sigmoid',encode=True, size=100,
kernel_depth=8, kernelfilter_size=3,weight_method='random', dropout_prob=0.0):
        self.n = n
        self.l_rate = l_rate
        self.encode=encode
        self.size = size
        self.act_func = act_func
        self.kernel_depth = kernel_depth
        self.kernelfilter_size = kernelfilter_size
        self.weight_method = weight_method
        self.dropout_prob = dropout_prob

    def hidden_layer(self, z):
        if self.act_func=='tanh':
            return (np.exp(z) - np.exp(-z))/(np.exp(z) + np.exp(-z))
        else:
            return 1/(1+ np.exp(-z))

    def softmax(self, z):
        z = z- np.max(z)
        return np.exp(z)/np.sum(np.exp(z), axis=1).reshape(-1,1)

    def onehot(self, data):
        n_classes = np.unique(data)
        onehot_array = np.zeros((data.shape[0], np.max(data)+1))
        onehot_array[np.arange(data.shape[0]), data] = 1
        if n_classes.shape[0] > 2:
```

```
        return onehot_array[:,n_classes]
    else:
        return onehot_array[:,n_classes[0]]

def iterate_kernels(self, X,R,C, batch, kernelfilter_size, step_size):
    for i in np.arange(0,R-(kernelfilter_size-1),step_size):
        for j in np.arange(0, C-(kernelfilter_size-1), step_size):
            yield X[batch,i:i+kernelfilter_size,j:j+kernelfilter_size], i, j

def forward_prop(self, X, R, C, kernelfilter_size, step_size, kernel_depth, output_size, size):
    Zconv = np.zeros((size, output_size, output_size, self.kernel_depth))
    for batch in range(size):
        for X_region, i, j in self.iterate_kernels(X,R,C,batch, kernelfilter_size, step_size):
            Zconv[batch, i,j] = np.sum(X_region * self.Wconv) + self.Bconv
    Ah = self.hidden_layer(Zconv)
    Ah = self.dropout(Ah, self.dropout_prob)
    flat_Ah = Ah.reshape(size,self.flat_weight)
    Zout = np.dot(flat_Ah, self.Wout) + self.Bout
    Aout = Zout
    Aout = self.softmax(Aout)
    return Zconv, Ah, Zout, Aout

def backward_prop(self, Zconv, Ah, Zout, Aout, x_train, y_train):
    output_error = Aout - y_train
    DJ_DWout = np.dot(Ah.reshape(self.size, self.flat_weight).T, output_error)
    if self.act_func == 'tanh':
        hidden_deriv = 1 - Ah**2
    else:
        hidden_deriv = Ah * (1-Ah)
    DJ_DWhid = np.dot(output_error,
self.Wout.T).reshape(self.size,self.output_size,self.output_size,self.kernel_depth) * hidden_deriv
    filters_deriv = np.zeros(self.Wconv.shape)
    for batch in range(self.size):
        for X_region, i, j in self.iterate_kernels(x_train, x_train.shape[1],x_train.shape[2], batch,
self.kernelfilter_size,1):
            for f in range(self.kernel_depth):
                filters_deriv[f] += np.sum(DJ_DWhid[batch,i,j,f] * X_region)
    grad_Wconv = filters_deriv
    grad_Bconv = np.sum(DJ_DWhid, axis=(0,1,2))
    grad_Wout = DJ_DWout
    grad_Bout = np.sum(output_error, axis=0)
    return grad_Wconv, grad_Bconv, grad_Wout, grad_Bout

def weight_init_(self, rows, cols, method, three_D=False, third_dim=None):
    if three_D:
        # Initializing method to zero
        if method == 'zero':
            x = np.zeros((third_dim, rows, cols))
        # Initializing method to Xavier
        elif method == 'Xavier':
```

```
            x = np.zeros((third_dim, rows, cols))
            for i in range(third_dim):
                x[i] = np.random.randn(rows, cols)*np.sqrt(1/cols)
        # Initializing method to He
        elif method == 'He':
            x = np.zeros((third_dim, rows, cols))
            for i in range(third_dim):
                x[i] = np.random.randn(rows, cols)*np.sqrt(2/cols)
        # Initializing method to Random
        else:
            x = np.random.randn((third_dim, rows, cols))
    else:
        # Initializing method to zero
        if method == 'zero':
            x = np.zeros((rows, cols))
        # Initializing method to Xavier
        elif method == 'Xavier':
            x = np.random.randn(rows, cols)*np.sqrt(1/cols)
        # Initializing method to He
        elif method == 'He':
            x = np.random.randn(rows, cols)*np.sqrt(2/cols)
        # Initializing method to Random
        else:
            x = np.random.randn(rows, cols)
    return x

def adam(self, gradient, t, w,m,v,learning_rate=0.05, bl_rate_1=0.9, bl_rate_2=0.999,
epsilon=1e-8):
        m = bl_rate_1 * m + (1-bl_rate_1)*gradient
        v = bl_rate_2 * v + (1-bl_rate_2)*(gradient**2)
        m_hat = m/(1-bl_rate_1**t)
        v_hat = v/(1-bl_rate_2**t)
        w = w - learning_rate*(m_hat/(np.sqrt(v_hat) + epsilon))
        return w, m, v

def dropout(self, data, dropout_prob):
    percentage_kept = 1-dropout_prob
    self.mask = np.random.binomial(1,percentage_kept,size=data.shape)*(percentage_kept)
    return self.mask * data

def fit(self, x_train, y_train, X_valid, y_valid):
    if self.encode:
        y_train_oh = self.onehot(y_train)
    padding = 0
    step_size=1
    R, C = x_train.shape[1:3]
    num_classes = 10
    self.output_size = int(((R - self.kernelfilter_size + 2*padding)/step_size) + 1)
    self.flat_weight = self.output_size*self.output_size*self.kernel_depth
```

```
        self.Wconv = self.weight_init_(self.kernelfilter_size,self.kernelfilter_size,self.weight_method,
three_D=True, third_dim=self.kernel_depth)
        self.Bconv = np.zeros(self.kernel_depth)
        self.Wout = self.weight_init_(self.flat_weight, num_classes,self.weight_method)
        self.Bout = np.zeros(num_classes)
        self.Wconv_m, self.Wconv_v = 0,0
        self.Bconv_m, self.Bconv_v = 0,0
        self.Wout_m, self.Wout_v = 0,0
        self.Bout_m, self.Bout_v = 0,0
        m = y_train.shape[0]

        time = 0
        self.metrics = {'cost': [],'train_accuracy': [], 'valid_accuracy':[]}
        for epoch in range(self.n):
          print("Epoch: {}".format(epoch+1))
          shuffled_values = np.random.permutation(m)
          X_shuffled = x_train[shuffled_values]
          y_shuffled = y_train_oh[shuffled_values]
          for batch in range(0,m,self.size):
            time += 1
            print("Batch {:.0f}/{:.0f}".format((batch/self.size)+1,m/self.size))
            x_batch = X_shuffled[batch:batch+self.size]
            y_batch = y_shuffled[batch:batch+self.size]

            # Forward Propagation
            Zconv, Ah, Zout, Aout = self.forward_prop(x_batch,R,C,self.kernelfilter_size,step_size,
self.kernel_depth,self.output_size, self.size)

            # Backward Propagation
            grad_Wconv, grad_Bconv, grad_Wout, grad_Bout = self.backward_prop(Zconv, Ah, Zout,
Aout, x_batch, y_batch)

            ## Adam Optimization
            self.Wconv, self.Wconv_m, self.Wconv_v = self.adam(grad_Wconv,t = time,w=self.Wconv,
m=self.Wconv_m, v=self.Wconv_v, learning_rate=self.l_rate)
            self.Bconv, self.Bconv_m, self.Bconv_v = self.adam(grad_Bconv, t= time, w=self.Bconv,
m=self.Bconv_m, v=self.Bconv_v, learning_rate=self.l_rate)

            self.Wout, self.Wout_m, self.Wout_v = self.adam(grad_Wout, t=time, w=self.Wout,
m=self.Wout_m, v=self.Wout_v, learning_rate=self.l_rate)
            self.Bout, self.Bout_m, self.Bout_v = self.adam(grad_Bout, t=time, w=self.Bout,
m=self.Bout_m, v=self.Bout_v, learning_rate=self.l_rate)

          # Training set Evaluation
          Zconv, Ah, Zout, Aout = self.forward_prop(x_train,R,C,self.kernelfilter_size, step_size,
self.kernel_depth, self.output_size, size=m)
          cost = self.cost_function(Aout, y_train_oh)
          train_pred = self.predict(Aout)

          # Validation set Evaluation
```

```
            Zconv, Ah, Zout, Aout = self.forward_prop(X_valid,R,C,self.kernelfilter_size, step_size,
self.kernel_depth, self.output_size, size=y_valid.shape[0])
            valid_pred = self.predict(Aout)

            train_accuracy = (np.sum(train_pred ==
y_train).astype(np.float)/train_pred.shape[0])*100
            valid_accuracy = (np.sum(valid_pred ==
y_valid).astype(np.float)/valid_pred.shape[0])*100
            print("Epoch: {}\t Train Acc: {:.3f}\t Validation Acc: {:.3f}".format(epoch+1,
train_accuracy, valid_accuracy))
            self.metrics['cost'].append(cost)
            self.metrics['train_accuracy'].append(train_accuracy)
            self.metrics['valid_accuracy'].append(valid_accuracy)

    def cost_function(self, Aout, y):
        self.Aout = Aout
        self.y_oh = y
        return np.average(np.sum((-y*np.log(Aout)),axis=1))

    def predict(self, output):
        return np.argmax(output, axis=1)

x_train_sample = x_train[:2000]
y_train_sample = y_train[:2000]

x_train_valid = x_train[40000:40500]
y_train_valid = y_train[40000:40500]

print("-----Model 1-----")
model1 = CNN(n=5, l_rate=0.05, act_func = 'tanh', encode=True, size=400, kernel_depth=15,
kernelfilter_size=5, weight_method='Xavier', dropout_prob=0.0)
model1.fit(x_train_sample, y_train_sample, x_train_valid, y_train_valid)

'''
-----Model 1-----
Epoch: 1
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 1	Train Acc: 70.850	Validation Acc: 69.600
Epoch: 2
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 2	Train Acc: 79.700	Validation Acc: 77.200
Epoch: 3
```

```
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 3      Train Acc: 84.650      Validation Acc: 81.400
Epoch: 4
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 4      Train Acc: 85.300      Validation Acc: 81.000
Epoch: 5
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 5      Train Acc: 90.000      Validation Acc: 85.800
'''
print("-----Model 2-----")
model2 = CNN(n=5, l_rate=0.05, act_func = 'tanh', encode=True, size=400, kernel_depth=15,
kernelfilter_size=5, weight_method='He', dropout_prob=0.0)
model2.fit(x_train_sample, y_train_sample, x_train_valid, y_train_valid)

'''
-----Model 2-----
Epoch: 1
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 1      Train Acc: 75.250      Validation Acc: 73.000
Epoch: 2
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 2      Train Acc: 82.600      Validation Acc: 83.000
Epoch: 3
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 3      Train Acc: 85.150      Validation Acc: 83.000
Epoch: 4
```

```
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 4	Train Acc: 87.750	Validation Acc: 82.800
Epoch: 5
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 5	Train Acc: 89.700	Validation Acc: 83.000
'''

print("-----Model 3-----")
model3 = CNN(n=5, l_rate=0.05, act_func = 'tanh', encode=True, size=400, kernel_depth=15, kernelfilter_size=5, weight_method='zero', dropout_prob=0.0)
model3.fit(x_train_sample, y_train_sample, x_train_valid, y_train_valid)

'''
-----Model 3-----
Epoch: 1
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 1	Train Acc: 10.700	Validation Acc: 9.600
Epoch: 2
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 2	Train Acc: 11.000	Validation Acc: 11.000
Epoch: 3
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 3	Train Acc: 11.000	Validation Acc: 11.000
Epoch: 4
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 4	Train Acc: 11.200	Validation Acc: 10.800
```

```
Epoch: 5
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 5	Train Acc: 11.200	Validation Acc: 10.800
'''

print("-----Model 4-----")
model4 = CNN(n=5, l_rate=0.05, act_func = 'tanh', encode=True, size=400, kernel_depth=15,
kernelfilter_size=5, weight_method='Xavier', dropout_prob=0.25)
model4.fit(x_train_sample, y_train_sample, x_train_valid, y_train_valid)

'''
-----Model 4-----
Epoch: 1
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 1	Train Acc: 25.300	Validation Acc: 25.200
Epoch: 2
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 2	Train Acc: 68.300	Validation Acc: 62.000
Epoch: 3
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 3	Train Acc: 67.200	Validation Acc: 56.000
Epoch: 4
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 4	Train Acc: 78.100	Validation Acc: 69.200
Epoch: 5
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
```

Epoch: 5	Train Acc: 78.600	Validation Acc: 72.200
'''

print("-----Model 5-----")
model5 = CNN(n=5, l_rate=0.05, act_func = 'tanh', encode=True, size=400, kernel_depth=15, kernelfilter_size=5, weight_method='He', dropout_prob=0.25)
model5.fit(x_train_sample, y_train_sample, x_train_valid, y_train_valid)

'''
-----Model 5-----
Epoch: 1
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 1	Train Acc: 76.100	Validation Acc: 69.600
Epoch: 2
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 2	Train Acc: 82.800	Validation Acc: 77.600
Epoch: 3
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 3	Train Acc: 86.150	Validation Acc: 78.200
Epoch: 4
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 4	Train Acc: 90.450	Validation Acc: 79.200
Epoch: 5
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 5	Train Acc: 92.550	Validation Acc: 79.800
'''

print("-----Model 6-----")
model6 = CNN(n=5, l_rate=0.05, act_func = 'tanh', encode=True, size=400, kernel_depth=15, kernelfilter_size=5, weight_method='zero', dropout_prob=0.25)

```
model6.fit(x_train_sample, y_train_sample, x_train_valid, y_train_valid)

'''
-----Model 6-----
Epoch: 1
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 1	Train Acc: 10.700	Validation Acc: 9.600
Epoch: 2
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 2	Train Acc: 11.200	Validation Acc: 10.800
Epoch: 3
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 3	Train Acc: 11.200	Validation Acc: 10.800
Epoch: 4
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 4	Train Acc: 11.200	Validation Acc: 10.800
Epoch: 5
Batch 1/5
Batch 2/5
Batch 3/5
Batch 4/5
Batch 5/5
Epoch: 5	Train Acc: 11.200	Validation Acc: 10.800
'''
```

Summary of results:

Building models using various initialization techniques and drop out values.

Model	Parameters	Training Accuracy	Validation Accuracy
Model1	n=5, l_rate=0.05, act_func = 'tanh', encode=True, size=400, kernel_depth=15, kernelfilter_size=5, weight_method='Xavier', dropout_prob=0.0	0.90	0.858
Model2	n=5, l_rate=0.05, act_func = 'tanh', encode=True, size=400, kernel_depth=15, kernelfilter_size=5, weight_method='He', dropout_prob=0.0	0.897	0.83
Model3	n=5, l_rate=0.05, act_func = 'tanh', encode=True, size=400, kernel_depth=15, kernelfilter_size=5, weight_method='zero', dropout_prob=0.0	0.112	0.108
Model4	n=5, l_rate=0.05, act_func = 'tanh', encode=True, size=400, kernel_depth=15, kernelfilter_size=5, weight_method='Xavier', dropout_prob=0.25	0.786	0.722
Model5	n=5, l_rate=0.05, act_func = 'tanh', encode=True, size=400, kernel_depth=15, kernelfilter_size=5, weight_method='He', dropout_prob=0.25	0.925	0.798
Model6	n=5, l_rate=0.05, act_func = 'tanh', encode=True, size=400, kernel_depth=15, kernelfilter_size=5, weight_method='zero', dropout_prob=0.25	0.112	0.108

Findings:

1. When there is no dropout layer, Xavier and He performs equally well on the train set but Xavier activation performs slightly better on the test set.
2. Both train and validation accuracy seems to be less when there is no activation in presence or absence of dropout.
3. Dropout of 0.25 significantly reduces accuracy in terms of Xavier activation function. However, not much effect when using He activation.