CS6910
Fundamentals of Deep Learning
Code Assignment -1 February 2020
Team 1: Arnav M(CS17B110), Samuel J(CS17B026) and Vamsi KV(CS17B045)

Neural Network Class:

```
import numpy as np
import matplotlib.pyplot as plt
import pickle

def mk_confusion_matrix(harvest):
    fig, ax = plt.subplots()
    ax.imshow(harvest)

    # Loop over data dimensions and create text annotations.
    for i in range(harvest.shape[0]):
        for j in range(harvest.shape[1]):
            ax.text(j, i, harvest[i, j],
                    ha="center", va="center", color="w")

    ax.set_title("Confusion Matrix")
    fig.tight_layout()
    plt.show()

class Sequential:
    # Local state variables:
    # The first elt of all the lists are
    # initialized to 0 so that it will be 1 indexed.
    # This for all except s and num_per_layer, where s[0] gives the input
    # vector:
    L = 0; nL= []; strfL = [0]; fL = [0]; dL = [0]; t = 0
    W = [0]; dW = [0]
    b = [0]; db = [0]
    a = [0]; da = [0]
    s = []
    delta = [0]
    bkprop_fn = 0
    prev_loss = 0
    lossf = 0
    curr_loss = 0
    prev_loss = np.array([[0]])
    r = [0]; q = [0]
    r_b = [0]; q_b = [0]
    pm1 = 1; pm2 = 1
    lamb=0

    # Activation functions:
    def Tanh(self, l):
        return np.tanh(self.a[l])

    def dTanh(self, l):
        # print(l, self.a[l])
```

```
        return 1-self.s[l]**2

    def Softmax(self, l):
        return np.exp(self.a[l])/np.sum(np.exp(self.a[l]),axis = 0, keepdims = True)

    def Lin(self, l):
        return self.a[l]

    def dLin(self,l):
        return 1

    dict_f = {'tanh': Tanh, 'softmax': Softmax, 'linear': Lin}
    dict_df = {'tanh': dTanh,'linear':dLin}

    # Constructor for the class:
    def __init__(self, layer_info):
        self.L = len(layer_info)-1
        for i in range(self.L+1):
            self.nL.append(layer_info[i][0])
            self.s.append(np.zeros((self.nL[i], 1)))
            if i > 0:
                self.strfL.append(layer_info[i][1])
                self.fL.append(self.dict_f[layer_info[i][1]])
                self.W.append(np.random.rand(self.nL[i-1], self.nL[i])-0.5)
                self.dW.append(np.zeros((self.nL[i-1], self.nL[i])))
                self.b.append(np.random.rand(self.nL[i], 1)-0.5)
                self.db.append(np.zeros((self.nL[i], 1)))
                self.a.append(np.zeros((self.nL[i], 1)))
                self.da.append(np.zeros((self.nL[i], 1)))
                self.delta.append(np.zeros((self.nL[i], 1)))
                self.r.append(np.zeros((self.nL[i-1], self.nL[i])))
                self.q.append(np.zeros((self.nL[i-1], self.nL[i])))
                self.r_b.append(np.zeros((self.nL[i], 1)))
                self.q_b.append(np.zeros((self.nL[i], 1)))
            if i > 0 and i < self.L:
                self.dL.append(self.dict_df[layer_info[i][1]])

    def loadWb(self, strf):
        f = open(strf, "rb")
        self.W, self.b = pickle.load(f)
        f.close()


    def fwprop_fn(self, flag = True):
        for l in range(1, self.L+1):
            # print(np.shape(self.W[l]))
            # print(np.shape(self.s[l-1]))
            # print(np.shape(self.b[l]))
            # print(np.shape(np.transpose(self.W[l])@self.s[l-1]))
            self.a[l] = (np.transpose(self.W[l])@self.s[l-1])+self.b[l]
            self.s[l] = self.fL[l](self, l)
            if not l == self.L:
```

```
                self.da[l] = self.dL[l](self, l)
            if (not l == 1) and (not l == self.L) and flag:
                siz = self.nL[l]
                shuf = np.arange(siz)
                np.random.shuffle(shuf)
                # print(shuf)
                dropout = np.zeros(shuf.shape, dtype=bool)
                dropout[shuf[:int(0.4*siz)]] = True
                # print(dropout)
                dropout = np.reshape(dropout, self.s[l].shape)
                self.s[l][dropout] = 0
                # print(self.s[l])
                np.random.shuffle(shuf)

        self.curr_loss += self.lossf(self)

    def computeDelta(self):
        # compute the delta next:
        self.delta[self.L] = self.t - self.s[self.L]
        for l in range(self.L-1, 0, -1):
            self.delta[l] = np.multiply(self.da[l], self.W[l+1]@self.delta[l+1])

    def deltaOptimizer(self, learning_rate):
        self.computeDelta()
        for l in range(1, self.L+1):
            self.dW[l] = learning_rate*self.s[l-1]@np.transpose(self.delta[l])
            self.W[l] += self.dW[l]
            self.db[l] = learning_rate*self.delta[l]
            self.b[l] += self.db[l]

            '''self.dW[l] = learning_rate*( self.s[l-1]@np.transpose(self.delta[l]) + self.lamb*self.W[l] )
            self.W[l] += self.dW[l]
            self.db[l] = learning_rate*( self.delta[l] + self.lamb*self.b[l] )
            self.b[l] += self.db[l]'''

    def genDeltaOptimizer(self, learning_rate, momentum_factor=.01):
        self.computeDelta()
        for l in range(1, self.L+1):
            # print(np.shape(momentum_factor*self.dW[l]))
            # print(np.shape(learning_rate*self.s[l-1]@np.transpose(self.delta[l])))
            self.dW[l] = learning_rate*self.s[l-1]@np.transpose(self.delta[l])-
momentum_factor*self.dW[l]
            self.W[l] += self.dW[l]
            self.db[l] = learning_rate*self.delta[l]-momentum_factor*self.db[l]
            self.b[l] += self.db[l]

    def adamOptimizer(self, learning_rate, rho1=0.9, rho2=0.999, epsilon=1e-8):
        self.pm1 *= rho1
        self.pm2 *= rho2
        self.computeDelta()
        for l in range(1, self.L+1):
            self.q[l] = rho1*self.q[l]+(1-rho1)*self.s[l-1]@np.transpose(self.delta[l])
```

```
            self.r[l] = rho2*self.r[l]+(1-rho2)*((self.s[l-1]@np.transpose(self.delta[l]))**2)
            qhat = self.q[l]/(1-self.pm1)
            rhat = self.r[l]/(1-self.pm2)
            self.dW[l] = learning_rate*qhat/(epsilon+(rhat)**0.5)
            self.W[l] += self.dW[l]

            #biases
            self.q_b[l] = rho1*self.q_b[l]+(1-rho1)*self.delta[l]
            self.r_b[l] = rho2*self.r_b[l]+(1-rho2)*self.delta[l]**2
            qhat_b = self.q_b[l]/(1-self.pm1)
            rhat_b = self.r_b[l]/(1-self.pm2)
            self.db[l] = learning_rate*qhat_b/(epsilon+(rhat_b)**0.5)
            self.b[l] += self.db[l]


    dict_opt = {'delta': deltaOptimizer, 'generalized delta': genDeltaOptimizer, 'adam':
adamOptimizer}

    def crossEntropy(self):
        return -np.sum(np.multiply(self.t,np.log(self.s[self.L])), axis = 0, keepdims = True)

    def sumOfSquares(self):
        return 0.5*np.sum((self.t-self.s[self.L])**2, axis = 0, keepdims = True)

    dict_loss = {'cross entropy': crossEntropy, 'sum of squares': sumOfSquares}

    def compile(self, optimizer = 'delta', loss = 'cross entropy'):
        self.bkprop_fn = self.dict_opt[optimizer]
        self.lossf = self.dict_loss[loss]

    def fit_r(self, in_v, out_v, E):
        storage = open("epoch_loss_img.txt", "w")
        for epoch in range(E):
            error = 0
            print(f"Epoch #{epoch}:", end = ' ')
            for x, y in zip(in_v, out_v):
                self.t = y
                self.s[0] = np.copy(np.reshape(x, (np.shape(x)[0], 1)))
                self.fwprop_fn()
                self.bkprop_fn(self, 0.00003, 0.01)
                error += (self.t - self.s[self.L][0])**2
                if epoch == E-1 :
                    plt.scatter([self.t], [self.s[self.L][0]])
            storage.write(f"{epoch} {error[0]/in_v.shape[0]}\n")

            print(f"Avg Error = {error/in_v.shape[0]}")
        plt.axis('equal')
        plt.show()
        f = open("weights.pkl", "wb")
        pickle.dump((self.W, self.b), f)
        f.close()
```

```
def fit(self, in_v, out_v, valin, valo, E):
    storage = open("epoch_loss_img.txt", "w")
    for epoch in range(E):
        corr = 0; incorr = 0
        self.curr_loss = 0
        # print("W: ", self.W)
        print(f"Epoch #{epoch}: ", end=' ')
        for x, y in zip(in_v, out_v):
            self.t = np.zeros((self.nL[self.L], 1))
            self.t[int(y), 0] = 1.
            self.s[0] = np.copy(np.reshape(x, (np.shape(x)[0], 1)))
            self.fwprop_fn()
            self.bkprop_fn(self, 0.01)
            if np.argmax(self.s[self.L]) == y:
                corr += 1
            else:
                incorr += 1
        storage.write(f"{epoch} {self.curr_loss[0, 0]} {100*(corr/(incorr+corr))}\n")
        print(f'Current loss: {self.curr_loss}',end=' ' )

        acc = self.test(valin, valo)
        print(f"Accuracy: {100*(corr/(incorr+corr))}")
        if 100*(corr/(incorr+corr)) >=75 or acc >= 52:
            break
        # if abs(self.curr_loss[0, 0] - self.prev_loss[0, 0]) <= 0.1:
        #     print(self.curr_loss, self.prev_loss)
        #     break
        self.prev_loss = np.copy(self.curr_loss)
    storage.close()
    f = open("weights.pkl", "wb")
    pickle.dump((self.W, self.b), f)
    f.close()

# def test(self, in_v, out_v):
#     corr = 0; incorr = 0
#     for x, y in zip(in_v, out_v):
#         self.t = np.zeros((self.nL[self.L], 1))
#         self.t[int(y), 0] = 1.
#         self.s[0] = np.copy(np.reshape(x, (np.shape(x)[0], 1)))
#         self.fwprop_fn(flag=False)
#         if np.argmax(self.s[self.L]) == y:
#             corr += 1
#         else:
#             incorr += 1
#     print(f"Accuracy of test: {100*corr/(corr+incorr)}", end = ' ')
#     return 100*corr/(corr+incorr)


def test(self, in_v, out_v):
    corr = 0; incorr = 0
```

```
        confusion = np.zeros((int(out_v.max()+1),int(out_v.max()+1)))
        for x, y in zip(in_v, out_v):
            self.t = np.zeros((self.nL[self.L], 1))
            self.t[int(y), 0] = 1.
            self.s[0] = np.copy(np.reshape(x, (np.shape(x)[0], 1)))
            self.fwprop_fn()
            if np.argmax(self.s[self.L]) == y:
                corr += 1
            else:
                incorr += 1
            confusion[int(np.argmax(self.s[self.L]))][int(y)]+=1
        print(confusion)
        mk_confusion_matrix(confusion)
        print(f"Accuracy of test: {100*corr/(corr+incorr)}", end = ' ')


    def test_r(self, in_v, out_v):
        #corr = 0; incorr = 0
        op=[]
        for x, y in zip(in_v, out_v):
            self.t = y
            self.s[0] = np.copy(np.reshape(x, (np.shape(x)[0], 1)))
            self.fwprop_fn(flag=False)
            op.append(self.s[self.L][0])
        return np.array(op)
```

Function Approximation:

Training Driver:

```
import NeuralNetwork_1 as nn
import numpy as np
import matplotlib.pyplot as plt

f = open("func_app_in/train100.txt",'r')
l = f.readlines()[1:]
f.close()

x = np.array([(float(s.split(' ')[0]), float(s.split(' ')[1])) for s in l])
y = np.array([float(s.split(' ')[2]) for s in l])
y = np.reshape(y, (np.shape(y)[0], 1))

f = open("func_app_in/val.txt",'r')
l = f.readlines()[1:]
f.close()

x_val = np.array([(float(s.split(' ')[0]), float(s.split(' ')[1])) for s in l])
y_val = np.array([float(s.split(' ')[2]) for s in l])
y_val = np.reshape(y_val, (np.shape(y_val)[0], 1))

model = nn.Sequential([
    (2, 'Nothing'),
    (50, 'tanh'),
    (50, 'tanh'),
```

```
    (1, 'linear')
])

# # model.loadWb("weights_2.pkl")
model.compile(optimizer='generalized delta', loss='cross entropy')

model.fit_r(x, y, 500)

op = model.test_r(x_val,y_val)
print(op.shape,y_val.shape)
plt.scatter(y_val,op)
plt.axis('equal')
plt.show()
print(model.W[-1])
```

Testing Driver:

```
import NeuralNetwork_1 as nn
import numpy as np
import matplotlib.pyplot as plt
from mayavi import mlab
from matplotlib import cm

f = open("func_app_in/train100.txt",'r')
l = f.readlines()
f.close()

x = np.array([(float(s.split(' ')[0]), float(s.split(' ')[1])) for s in l])
y = np.array([float(s.split(' ')[2]) for s in l])
y = np.reshape(y, (np.shape(y)[0], 1))

wtfile = "reg_wts.pkl"

model = nn.Sequential([
    (2, 'Nothing'),
    (50, 'tanh'),
    (50, 'tanh'),
    (1, 'linear')
])

model.compile(optimizer='generalized delta', loss='cross entropy')
model.loadWb(wtfile)

op = model.test_r(x,y)

print(op)
```

2D – Non Linear Data:

Training Driver:

```
import NeuralNetwork_1 as nn
import numpy as np

f = open("traingroup1.csv",'r')
l = f.readlines()[1:]
f.close()
```

```
x = np.array([(float(s.split(',')[0]), float(s.split(',')[1])) for s in l])
# x /= np.sum(x, axis = 0, keepdims = True)
y = np.array([float(s.split(',')[2]) for s in l])
y = np.reshape(y, (np.shape(y)[0], 1))

print(x.shape)
print(y.shape)

model = nn.Sequential([
    (2, 'Nothing'),
    (6, 'tanh'),
    (6, 'tanh'),
    (3, 'softmax')
])

# model.loadWb("weights_2.pkl")
model.compile(optimizer='adam', loss='cross entropy')

model.fit(x[:250], y[:250], x[250:], y[250:], 10000)
```
Testing Driver:
```
import NeuralNetwork_1 as nn
import numpy as np

f = open("traingroup1.csv",'r')
l = f.readlines()[1:]
f.close()
wt_file="delta_2d_non_linear/good_wts_adam.pkl"

x = np.array([(float(s.split(',')[0]), float(s.split(',')[1])) for s in l])
# x /= np.sum(x, axis = 0, keepdims = True)
y = np.array([float(s.split(',')[2]) for s in l])
y = np.reshape(y, (np.shape(y)[0], 1))

print(x.shape)
print(y.shape)

model = nn.Sequential([
    (2, 'Nothing'),
    (6, 'tanh'),
    (6, 'tanh'),
    (3, 'softmax')
])

model.loadWb(wt_file)
model.compile(optimizer='generalized delta', loss='cross entropy')

model.test(x,y)
```
Training Driver:
```
import NeuralNetwork_1 as nn
import numpy as np
import glob
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

fl = glob.glob('img_in/*.npy')
data = None
x = None
y = []
cnt = 0
for f in fl:
    tmp = np.load(f)
    if x is None:
        x = tmp
    else:
        x = np.vstack((x, tmp))
    y.extend([cnt]*len(tmp))
    cnt += 1
y = np.array(y)

x = x.reshape((x.shape[0], x.shape[1]))
y = y.reshape((y.shape[0], 1))
shuf = np.arange(len(y))

np.random.shuffle(shuf)
x = x[shuf]
y = y[shuf]

x = StandardScaler().fit_transform(x)
print(x[0].sum(), x[0].max())

pca = PCA(n_components=256)
x = pca.fit_transform(x)

model = nn.Sequential([
    (80, 'Nothing'),
    (10, 'tanh'),
    (8, 'tanh'),
    (5, 'softmax')
])

model.compile(optimizer='adam', loss='cross entropy')

model.fit(x[:1000], y[:1000], x[1000:], y[1000:], 10000)

# model.test(x[1200:], y[1200:])
```

Testing Driver:

```
import NeuralNetwork_1 as nn
import numpy as np
import glob
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
```

```
wt_file = "gen_delta_image/weights_83_2.pkl"

fl = glob.glob('img_in/*.npy')
data = None
x = None
y = []
cnt = 0
for f in fl:
    tmp = np.load(f)
    if x is None:
        x = tmp
    else:
        x = np.vstack((x, tmp))
    y.extend([cnt]*len(tmp))
    cnt += 1
y = np.array(y)

x = x.reshape((x.shape[0], x.shape[1]))
y = y.reshape((y.shape[0], 1))
shuf = np.arange(len(y))

np.random.shuffle(shuf)
x = x[shuf]
y = y[shuf]

x = StandardScaler().fit_transform(x)
print(x[0].sum(), x[0].max())

#pca = PCA(n_components=25)
#x = pca.fit_transform(x)

model = nn.Sequential([
    (512, 'Nothing'),
    (5, 'tanh'),
    (5, 'tanh'),
    (5, 'softmax')
])

model.loadWb(wt_file)
model.compile(optimizer='generalized delta', loss='cross entropy')

model.test(x, y)
```