Home Assignment 1: Deep Learning from Scratch

Submited by
- Liel leman – 315929372
- Liran Nochumsohn – 204693410

2.1 Part I: the classifier and optimizer:

Section 2.1.1

In this block we implemented the softmax, the softmax loss and the softmax gradient.\

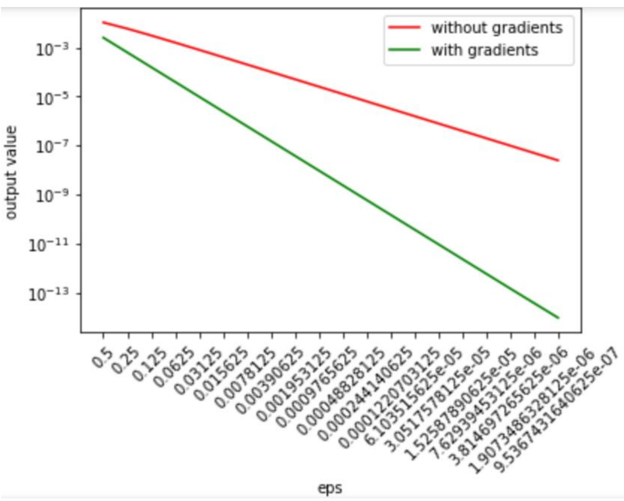
The parameters:

X - train data of shape (features, observations)
w - weights dataset of shape (features, labels)
C - the label data (observations, labels)
eta - softmax stablizier

```
import numpy as np
import scipy.io as sio
import matplotlib.pyplot as plt
import scipy
from tqdm.notebook import tqdm


def softmax(X, w, eta = True):
    product_Xw = X.T @ w
    if eta==True:
        exp = np.exp(product_Xw - np.max(product_Xw ))
    else:
        exp = np.exp(product_Xw )
    div = np.divide(exp, np.sum(exp, axis = 1).reshape(-1,1))
    return div

def softmax_loss(X, C, w, eta = True):
    sm = softmax(X, w, eta = eta)
    log  = np.log(sm)
    m = len(X[0])
    return -np.sum(C*log)/m

def softmax_gradient_W(X, C, w, eta = True):
    sm= softmax(X, w, eta = eta)
    m = len(X[0])
    gradient = (1/m)*X @ (sm - C)
    return gradient

def softmax_gradient_X(X, C, w, eta = True):
    sm= softmax(X, w, eta = eta)
    m = len(X[0])
    gradient = (1/m)*w @ (sm - C).T
    return gradient
```

Here we check the correctness of the code written above with a gradinet test technique.

```
def data_loader(file_name):
    mat = scipy.io.loadmat(file_name)
    Xtrain = mat.get('Yt')
    # for the bias
    Xtrain = np.vstack([Xtrain, np.ones(Xtrain.shape[1])])
    Ytrain = mat.get('Ct').T
    Xtest = mat.get('Yv')
    Xtest = np.vstack([Xtest, np.ones(Xtest.shape[1])])
    Ytest = mat.get('Cv').T
    return Xtrain, Ytrain, Xtest, Ytest

def gradient_test(X,C):
    # number of dimensions (features)
    n = X.shape[0]
    # number of labels
    l = C.shape[1]


    d = np.random.rand(n,l)
    d = d/np.linalg.norm(d,ord = 1, axis = 0)
    w = np.random.rand(n,l)

    loss = softmax_loss(X, C, w, eta = True)
    grad = softmax_gradient_W(X, C, w, eta = True)

    comp1 = []
    comp2 = []
    eps_list = []

    eps = 0.5
    test_range = list(range(20))
    for i in test_range:
        loss_d = softmax_loss(X, C, w + eps*d , eta = True)
        comp1.append(abs(loss_d  - loss ) )
        comp2.append(abs(loss_d  - loss - (d * eps).ravel() @ grad.ravel()))
        eps_list.append(eps)
        eps *= 0.5

    plt.plot(test_range,comp1,color = 'r')
    plt.plot(test_range,comp2, color = 'g')
    plt.legend(["without gradients ", "with gradients "])
    plt.xlabel('eps')
    plt.ylabel('output value')
    plt.yscale('log')
    plt.xticks( range(len(eps_list)),eps_list,rotation = 45)
    plt.show()

datasets = ["GMMData","PeaksData","SwissRollData"]
Xtrain, Ytrain, Xtest, Ytest = data_loader(datasets[0])
gradient_test(Xtrain ,Ytrain)
```



Section 2.1.2

In this section, we implement SGD of minibatches . We measure the value of the loss function(least squares error) and the accuracy at each epoch. This allows us to show that we have reached a minimum error, and maximum accuracy as expected, the loss and is being reduced by the optimization algorithm.

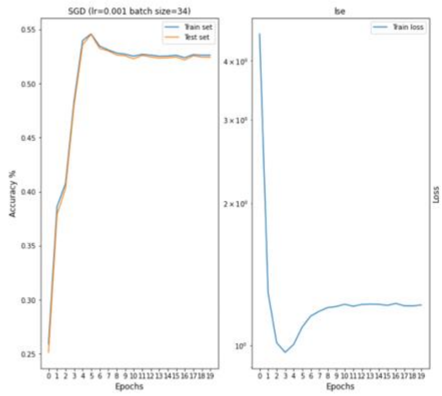
```
def least_squares_loss(x,y,w):
    pred_labels = np.argmax(softmax(x,w),axis=1)
    true_labels = y.argmax(axis=1)
    loss = np.linalg.norm(true_labels-pred_labels,2)**2 / x.shape[1]
    return loss

def grad_lse(x,y,w):
    xw = x.T @ w
    xw_sub_y = np.subtract(xw,y)
    grad = x @ xw_sub_y / x.shape[1]
    return grad

def accuracy(x, y, w):
    pred_labels = np.argmax(softmax(x,w),axis=1)
    true_labels = y.argmax(axis=1)
    return np.count_nonzero(true_labels == pred_labels) / x.shape[1]

def sgd(dataset , lr , batch_size, epochs=10,loss_type='lse', to_plot=False):
    if loss_type == 'lse':
        grad_loss_func = grad_lse
        loss_func = least_squares_loss
    else:
        grad_loss_func = softmax_gradient_W
        loss_func = softmax_loss

    Xtrain, Ytrain, Xtest, Ytest = dataset
    feature_num, examples_num = Xtrain.shape
    labels_num = Ytrain.shape[1]

    train_acc=[]
    test_acc=[]
    train_loss = []

    # Initialize weights
    w = np.random.randn(feature_num, labels_num)

    # Train loop
    for epoch in tqdm(range(epochs)):
        # Shuffle train data
        indices = np.arange(examples_num)
        np.random.shuffle(indices)
        Xtrain = Xtrain[:, indices]
        Ytrain = Ytrain[indices, :]
        i = 0
        while i * batch_size < examples_num:
            # Obtain minibatch
            batch_start = i * batch_size
            batch_end = min((i + 1) * batch_size, examples_num)
            minix = Xtrain[:, batch_start:batch_end]
            miniy = Ytrain[batch_start:batch_end, :]

            # Update weights
            grad = grad_loss_func(minix, miniy, w)
            w -= lr * grad
            i += 1

        # Calculate accuracies
        train_accuracy = accuracy(Xtrain, Ytrain, w)
        test_accuracy = accuracy(Xtest, Ytest, w)
        train_acc.append(train_accuracy)
        test_acc.append(test_accuracy)
        train_loss.append(loss_func(Xtrain,Ytrain,w))


    if to_plot:
        fig, (ax1, ax2) = plt.subplots(1, 2,constrained_layout=True,figsize=(9,8))

        xs = range(epochs)
        # Plot
        ax1.plot(xs, train_acc)
        ax1.plot(xs, test_acc)
        # Set y-axis parameters
        ax1.set_yticks(np.arange(round(min(train_acc + test_acc), 2), 0.6, 0.05))
        ax1.set_ylabel('Accuracy %', size=12)
        # Set x-axis parameters
        ax1.set_xlabel('Epochs', size=12)
        ax1.set_xticks(xs)
```

```
        # ax1.set_xticks( range(len(xs)),xs,rotation = 45)
        # Title and show
        ax1.set_title(f'SGD (lr={lr} batch size={batch_size})')
        ax1.legend(['Train set', 'Test set'], loc='best')

        ax2.semilogy(train_loss)
        ax2.legend(['Train loss'])
        ax2.set_title(f'{loss_type}')
        ax2.set_ylabel('Loss', size=12)
        ax2.yaxis.set_label_position("right")
        ax2.set_xlabel('Epochs',size=12)
        ax2.set_xticks(xs)
        plt.show()
```

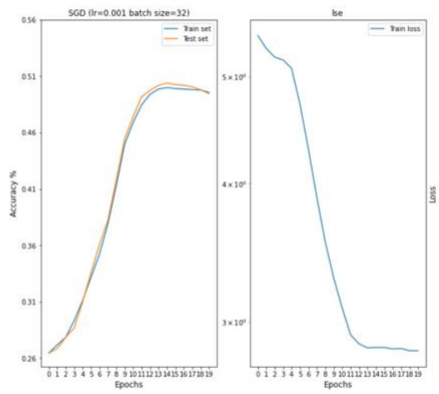
In our experiments we tested different parameters in for obtaining global minimum.
We got to conclusion that there needed to be a lot of fine tuning to batch size and the learning rate.
notice that since these datasets represent a non-linear problem, we expect our model be bad at this problems.

- for GMM we got around 50% accuracy.
- for Peaks we got around 60% accuracy.

```
sgd(data_loader('GMMData') , lr=0.001, batch_size=32, epochs=20,loss_type='lse' , to_plot=True)

sgd(data_loader('PeaksData') , lr=0.001, batch_size=32, epochs=20,loss_type='lse', to_plot=True)
```



GmmData

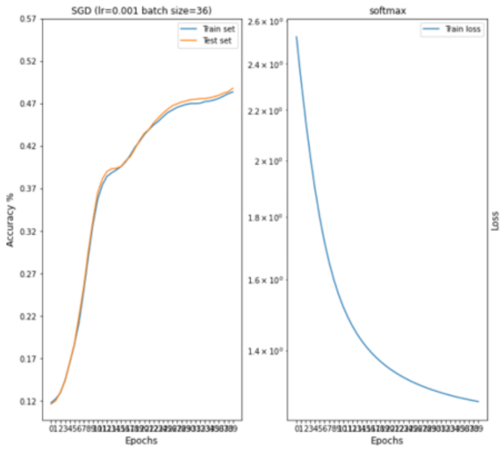


PeaksData

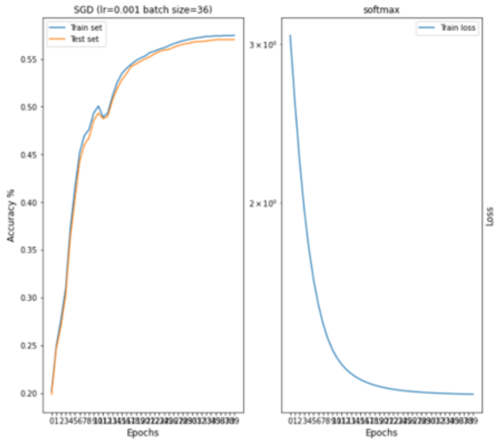
Section 3:

Same experiment as before only with Cross entropy as loss function:

```
sgd(data_loader('GMMData') , lr=0.001, batch_size=36, epochs=40,loss_type='softmax' , to_plot=True)
sgd(data_loader('PeaksData') , lr=0.001, batch_size=36, epochs=40,loss_type='softmax', to_plot=True)
```



GmmData



PeaksData

Part 2: The Neural Network

Section 1 and 2:

In this section we implement the NN with respect to a normal layer, resnet layer and output layer. The output layer ends with a softmax output.

```
# master layer class, HiddenLayer,OutputLayer and ResnetLayer inherit from this class.
# it contains the basic properties of a NN layer.
class Layer:
    def __init__(self,input_dim,output_dim):
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.W = 0.10 * np.random.randn(output_dim,input_dim)
        self.X = None
    def forward(self,input):
        pass
    def backward(self,output_grad,lr):
        pass
    def __repr__(self):
        pass

# class for a regular hidden layer
class HiddenLayer(Layer):
    def __init__(self,input_dim,output_dim,activation):
        super().__init__(input_dim,output_dim)
        self.b = np.random.randn(output_dim,1)
        self.activation = activation

    def forward(self,X,  training = True):
        # save x only if learning
        if training:
            self.X = X
        z = self.W @ X + self.b
        # send to activation function
        out = self.activation.forward(z)
        return out

    def backward(self,v,lr, test = None):
        # for gradient test only!
        if test:
            grad_W, grad_X, grad_b = self.activation.backward( v,self.X,self.W,self.b,test)
            return grad_W, grad_X, grad_b
```

```
        else:
            grad_W, grad_X, grad_b = self.activation.backward( v,self.X,self.W,self.b)

        self.W -= lr*grad_W
        self.b -= lr*grad_b
        v = grad_X
        return v
    def __repr__(self):
        rep = 'HiddenLayer' + str((self.input_dim,self.output_dim))
        return rep

#class for the output layer
class OutputLayer(Layer):
    def __init__(self,input_dim,output_dim,loss_f):
        super().__init__(input_dim,output_dim)
        self.loss_f = loss_f

    def forward(self,X, training = True):
        if training:
            self.X = X
        out = self.loss_f.forward(X,self.W)
        return out

    def backward(self,C,lr):
        grad_W, grad_X = self.loss_f.backward(self.X,self.W,C)
        self.W -= lr*grad_W.T
        v = grad_X
        return v

    def get_loss(self,X,C):
        return self.loss_f.calc_loss(X, C)
    def __repr__(self):
        rep = 'OutputLayer' + str((self.input_dim,self.output_dim))
        return rep

#class for the resnet layer
class ResNetLayer(HiddenLayer):
    def __init__(self,dim ,activation):
        super().__init__(dim,dim,activation)
        self.activation = activation
        self.W2 = 0.10 * np.random.randn(dim,dim)


    def forward(self,X,  training = True):
        z = super().forward(X,  training )
        out = X + self.W2 @ z
        return out

    def backward(self,v,lr):
        W2v = self.W2.T @ v
        grad_W,grad_X,grad_b = self.activation.backward( W2v,self.X,self.W,self.b)
        self.W -= lr*grad_W
        z = self.W @ self.X + self.b

        self.W2 -= lr*(v @ self.activation.forward(z).T )
        grad_X = v + grad_X
        self.b -= lr*grad_b
        v = grad_X
        return v

    def __repr__(self):
        rep = 'ResNetLayer' + str((self.input_dim,self.output_dim))
        return rep
# Master class for the activation functions, this class contains the forward (output)
# and the gradient of a given activation function.

class Activation:
    def __init__(self,activation,activation_prime):
        self.activation = activation
        self.activation_prime = activation_prime
        self.z = None

    def forward(self,z):
        self.z = z
        return self.activation(self.z)

    def backward(self, v,X,W,b, test = None):
        activation_p = self.activation_prime(self.z)
        grad = np.multiply(activation_p,v)

        if test=='test W':
            grad_X = None
            grad_W = np.multiply(activation_p ,(v @ X ))

        elif test=='test X':
            grad_X = np.multiply(activation_p ,(W @ v ))
            grad_W = None
```

```
        else:
            grad_X = W.T @ grad
            grad_W = grad @ X.T
        grad_b = np.sum(grad,axis = 1).reshape(-1,1)
        return grad_W,grad_X,grad_b

# Tanh activations function, inherties most of its properties from the master class.
class Tanh(Activation):
    def __init__(self):
        tanh = lambda x: np.tanh(x)
        tanh_prime = lambda x:1-np.tanh(x)**2
        super().__init__(tanh,tanh_prime)
# Softmax loss function class, the implementaion is similar to the implementation is part 1 of the
# home work assignment with slight adaptations.

class Softmax:
    def __init__(self):
        self.out = None

    def forward(self,X,W):
        product_Xw = X.T @ W.T
        exp = np.exp(product_Xw - np.max(product_Xw ))
        div = np.divide(exp, np.sum(exp, axis = 1).reshape(-1,1))
        self.out = div
        return self.out

    def calc_loss(self,X, C):
        #cross entrophy loss
        log  = np.log(self.out)
        m = len(X[0])
        return -np.sum(C*log)/m

    def backward(self,X,W, C):
        m = len(X[0])
        sub =  self.out - C
        grad_W  = (1/m)*(X @ sub)
        grad_X  = (1/m)*(W.T @ sub.T)
        return  grad_W, grad_X



# The general NN class
class Net:
    def __init__(self,layers,lr=0.01):
        self.lr = lr
        self.layers = layers
        self.loss = 0

    def forward(self,X,C = None, training = True, update_loss = True):
        '''
        forward pass of the network
        '''
        output = X
        for layer in self.layers:
            output = layer.forward(output, training)

        if update_loss:
            batch_loss = self.layers[-1].get_loss(output, C)
            self.loss += batch_loss
        return output


    def backward(self,C):
        '''
        backward pass through the network + updating params
        '''
        v = self.layers[-1].backward(C,self.lr)

        for layer in np.flip(self.layers[:-1]):
            v = layer.backward(v, self.lr)

    def predict(self,X):
        prob = self.forward(X,training = False, update_loss = False)
        output = (prob == prob.max(axis=1)[:,None]).astype(int)
        return output

def accuracy(pred, true):
    div = np.divide(pred + true,2).astype(int)
    output = np.sum(div)/len(div)
    return output

def data_loader(file_name):
    mat = scipy.io.loadmat(file_name)
    Xtrain = mat.get('Yt')
    # for the bias
    Ytrain = mat.get('Ct')
    Xtest = mat.get('Yv')
    Ytest = mat.get('Cv')
```

```
    return Xtrain, Ytrain, Xtest, Ytest

datasets = ["GMMData","PeaksData","SwissRollData"]

def jac_test(test, resnet = False, x = None,input_dim = None,output_dim = 5,layer = None, v=None):
    d = np.random.rand(3, 3)

    if x == None:
        input_dim = 3
        output_dim = 3
        x = np.random.rand(3, 3)
        if test == 'test b':
            d = np.random.rand(3, 1)
            x = np.random.rand(3, 1)
        layer = HiddenLayer(3, 3, Tanh())

    d = d / np.linalg.norm(d)
    x_input = x
    eps_num = 10
    eps_vals = [0.5**i for i in range(1,eps_num+1)]
    Oϵ, Oϵ2 = [], []

    fx = layer.forward(x)
    if test =='test W':
        w = layer.W
    elif test == 'test b':
        b= layer.b

    test_dict = {"test b": 0, "test W": 0, "test X":0}

    for eps in eps_vals:
        eps_d = eps * d

        if test =='test W':
            layer.W = np.add(w, eps_d)
        if test == 'test b':
            layer.b = np.add(b, eps_d)
        if test == 'test X':
            x_input = np.add(x, eps_d)

        fx_d = layer.forward(x_input)

        test_dict["test W"],test_dict["test X"],test_dict["test b"] = layer.backward(eps_d,0.001,test)


        first_ord = fx_d - fx
        second_ord = first_ord - test_dict[test]

        Oϵ.append(np.linalg.norm(first_ord))
        Oϵ2.append(np.linalg.norm(second_ord))
    return Oϵ,Oϵ2


def make_plot_test(Oϵ, Oϵ2, name):
    f,ax = plt.subplots(figsize=(3,4))
    ax.semilogy(range(len(Oϵ)), Oϵ, label='First Order')
    ax.semilogy(range(len(Oϵ2)), Oϵ2, label='Second Order')
    ax.legend()
    ax.set_title('jacMV '+ name)
    return ax
```

Here we demonstrate Jacobian tests for our linear layer network:

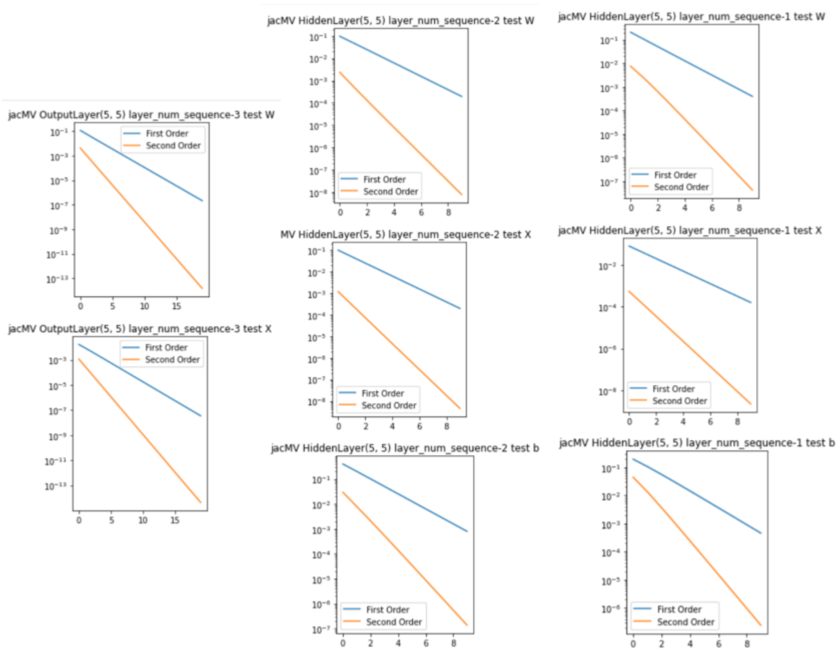
```
test = ['test X','test W','test b']
for test in test:
    Oϵ, Oϵ2 =  jac_test(test)
    make_plot_test(Oϵ, Oϵ2, name=test)
```

Section 3:

Now, after we’ve verified that all the individual parts are OK, we will examine that their combination together in the whole network also works—the forward pass and the backward pass of the network with L layers (L is a parameter). We would check that the gradient of the whole network (softmax + layers) passes the gradient test

```
def jac_test_layer(test, layer = None):
    if test == 'test b':
        d = np.random.rand(layer.input_dim, 1)
    else :
        d = np.random.rand(layer.input_dim, layer.output_dim)

    d = d / np.linalg.norm(d)
    x = layer.X
    x_input = x
    eps_num = 10
    eps_vals = [0.5**i for i in range(1,eps_num+1)]
    Oϵ, Oϵ2 = [], []

    fx = layer.forward(x,training=False)
    if test =='test W':
        w = layer.W
    elif test == 'test b':
        b= layer.b

    test_dict = {"test b": 0, "test W": 0, "test X":0}

    for eps in eps_vals:
        eps_d = eps * d

        if test =='test W':
            layer.W = np.add(w, eps_d)
        if test == 'test b':
            layer.b = np.add(b, eps_d)
        if test == 'test X':
            x_input = np.add(x, eps_d)

        fx_d = layer.forward(x_input,training=False)

        test_dict["test W"],test_dict["test X"],test_dict["test b"] = layer.backward(eps_d,0.001,test)


        first_ord = fx_d - fx
        second_ord = first_ord - test_dict[test]

        Oϵ.append(np.linalg.norm(first_ord))
        Oϵ2.append(np.linalg.norm(second_ord))
    return Oϵ,Oϵ2


def softmax_jac_test_layer(test,C,layer):
    X = layer.X
    # number of dimensions (features)
    n = X.shape[0]
    # number of labels
    l = C.shape[1]
    # batch size
    b = X.shape[1]

    if test=='test W':
        d = np.random.rand(l,n)
    if test=='test X':
        d = np.random.rand(n,b)
    d = d/np.linalg.norm(d,ord = 1, axis = 0)

    w = np.random.rand(l,n)
    layer.W = w
    layer.forward(X,training=False)

    grad_dict = {"test W": 0, "test X":0}
    loss = layer.loss_f.calc_loss(X, C)

    # keep the initial weights aside
    W = layer.W

    grad_dict["test W"],grad_dict["test X"] = layer.loss_f.backward(X,W, C)
    grad = grad_dict[test]
```

```python
    if test == 'test W':
        grad = grad.T

    Oϵ= []
    Oϵ2 = []
   # eps_list = []

    eps = 0.5
    test_range = list(range(20))
    for i in test_range:
        if test == 'test W':
            layer.W = W + eps*d
            layer.forward(X,training = False)

        if test == 'test X':
            layer.loss_f.forward(X+ eps*d,W )

        loss_d = layer.loss_f.calc_loss(X, C)

        Oϵ.append(abs(loss_d  - loss ) )
        Oϵ2.append(abs(loss_d  - loss - (d * eps).ravel() @ grad.ravel()))
      #  eps_list.append(eps)
        eps *= 0.5

    return Oϵ, Oϵ2


epocs = 4
lr = 0.1
batch_size = 1
input_dim = feature_num
output_dim = labels_num

nn = Net(layers = [HiddenLayer(input_dim,5,Tanh()),
                   HiddenLayer(5,5,Tanh()),
                   OutputLayer(5,output_dim,Softmax())],
                   lr = lr)

indices = np.arange(examples_num)
np.random.shuffle(indices)
Xtrain = Xtrain[:,indices]
Ytrain = Ytrain[:,indices]

batch_start = 0
batch_end =  batch_size

minix = Xtrain[:,batch_start:batch_end]
miniy = Ytrain[:,batch_start:batch_end].T

nn.forward(minix,miniy);
#%%
i = len(nn.layers)
for layer in reversed(nn.layers):
    if isinstance(layer,OutputLayer):
        tests = ['test W','test X']
        for test in tests:
            Oϵ,Oϵ2 = softmax_jac_test_layer(test,miniy,layer)
            make_plot_test(Oϵ,Oϵ2,str(layer)+f' layer_num_sequence-{i} '+test)
    else:
        tests = ['test W','test X','test b']
        for test in tests:
            Oϵ,Oϵ2 = jac_test_layer(test,layer)
            make_plot_test(Oϵ,Oϵ2,str(layer)+f' layer_num_sequence-{i} '+test)

    i-=1
```



We could observe that the gradient \ Jacobian tests are behave as we predicted.

<u>Section 4:</u>

First we will construct 5 different networks:

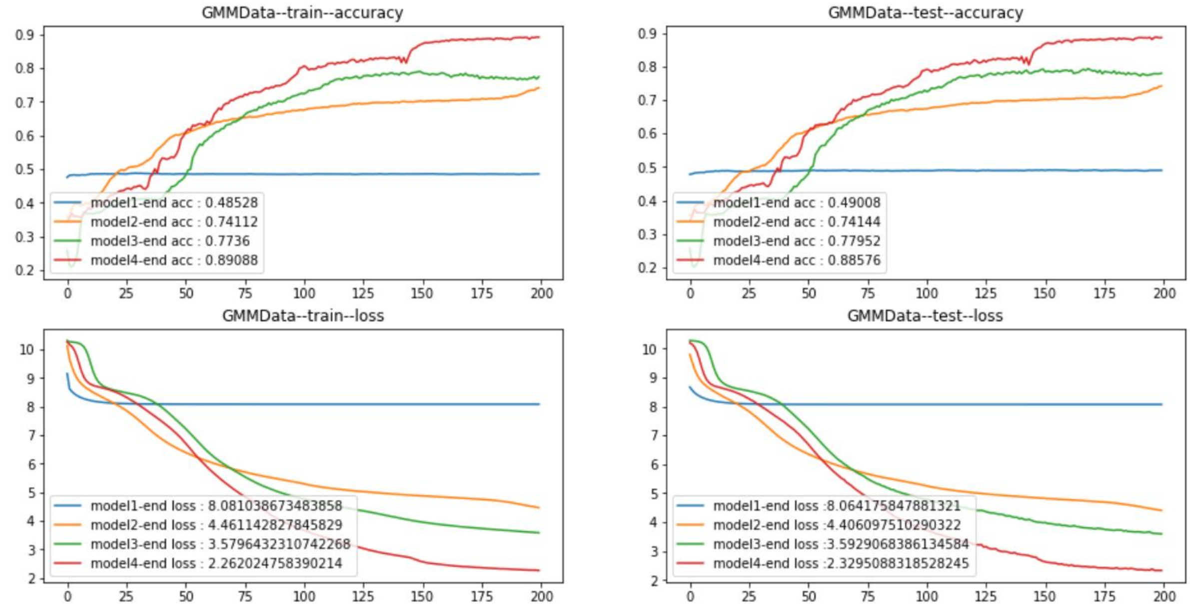
```
nn_linear = lambda feature_num,labels_num,lr : Net(layers = [
OutputLayer(feature_num,labels_num,Softmax())], lr = lr)
#%%
nn_lambda_2layer = lambda feature_num,labels_num,lr : Net(layers = [ HiddenLayer(feature_num,6,Tanh()),
                                   OutputLayer(6,labels_num,Softmax())], lr = lr)
#%%
nn_lambda_3layer = lambda feature_num,labels_num,lr : Net(layers = [ HiddenLayer(feature_num,6,Tanh()),
                                   HiddenLayer(6,6,Tanh()),
                                   OutputLayer(6,labels_num,Softmax())], lr = lr)
#%%
nn_lambda_4layer = lambda feature_num,labels_num,lr : Net(layers = [ HiddenLayer(feature_num,6,Tanh()),
                                   HiddenLayer(6,6,Tanh()),
                                   HiddenLayer(6,6,Tanh()),
                                   OutputLayer(6,labels_num,Softmax())] , lr = lr)

#%%
nn_resnet = lambda feature_num,labels_num,lr : Net(layers = [ HiddenLayer(feature_num,6,Tanh()),
                                   ResNetLayer(6,Tanh()),
                                   HiddenLayer(6,6,Tanh()),
                                   OutputLayer(6,labels_num,Softmax())] , lr = lr)
#%%
```

now we will implement train process:

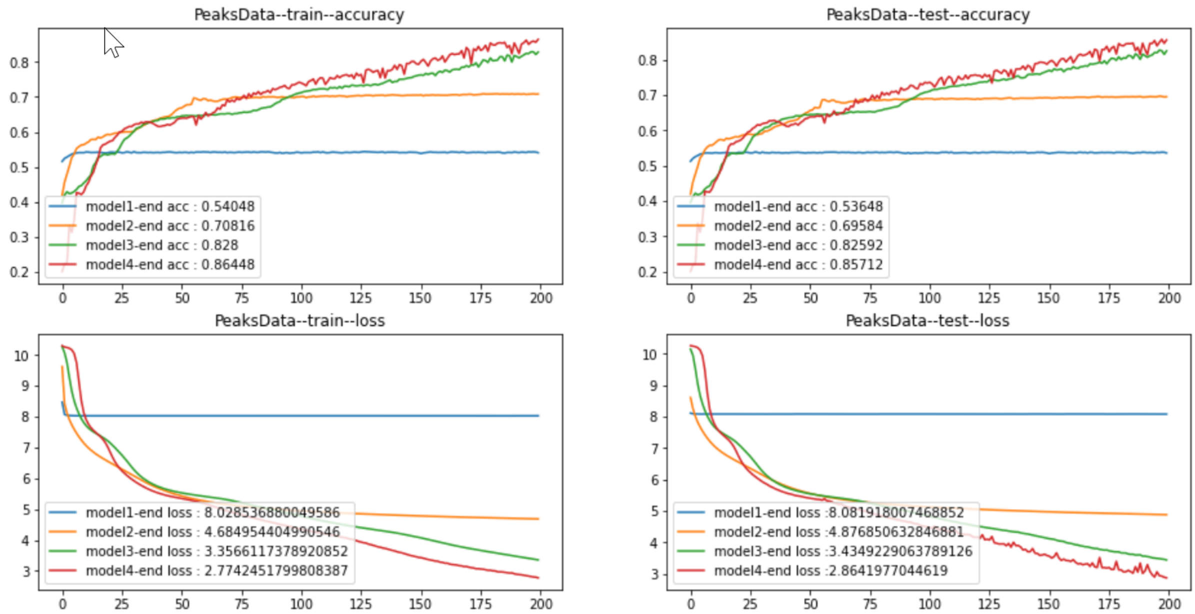
```
def train_net(net_lambda,data,epocs=200,lr=0.1,batch_size=64):
    Xtrain,Ytrain,Xtest,Ytest= data
    feature_num,examples_num = Xtrain.shape
    labels_num = Ytrain.shape[0]
    net = net_lambda(feature_num,labels_num,lr)
    print(f'netrwork layers:{net.layers}')
    # Train loop
    a_t = []
    a_v = []
    train_accuracy =[]
    val_accuracy =[]
    training_loss = []
    val_loss =[]
    for epoc in range(epocs):
        # Shuffle train data
        indices = np.arange(examples_num)
        np.random.shuffle(indices)
        Xtrain = Xtrain[:,indices]
        Ytrain = Ytrain[:,indices]
```

```python
        i = 0
        net.loss = 0
        while i * batch_size < examples_num:
            # Obtain minibatch
            batch_start = i * batch_size
            batch_end = min((i + 1) * batch_size, examples_num)

            minix = Xtrain[:,batch_start:batch_end]
            # transpoce the miniy to be inline with C
            miniy = Ytrain[:,batch_start:batch_end].T
            a_t.append(batch_end-batch_start)
            #forward pass
            net.forward(minix,miniy)

            #Backward+update nn params
            net.backward(miniy)

            #next batch...
            i += 1

        bias_fix = (i * batch_size - examples_num)/batch_size
        training_loss.append(net.loss/(i-1+bias_fix))
        train_accuracy.append(accuracy(net.predict(Xtrain),Ytrain.T))

        # validation

        examples_num = Xtest.shape[1]
        indices = np.arange(examples_num)
        np.random.shuffle(indices)
        Xtest = Xtest[:,indices]
        Ytest = Ytest[:,indices]
        j = 0
        net.loss = 0

        while j * batch_size < examples_num:
            # Obtain minibatch
            batch_start = j * batch_size
            batch_end = min((j + 1) * batch_size, examples_num)

            minix = Xtest[:,batch_start:batch_end]
            # transpoce the miniy to be inline with C
            miniy = Ytest[:,batch_start:batch_end].T
            a_t.append(batch_end-batch_start)
            #forward pass
            net.forward(minix,miniy, training=False )

            #next batch...
            j += 1
        bias_fix = (j * batch_size - examples_num)/batch_size
        val_loss.append(net.loss/(j-1+bias_fix))
        val_accuracy.append(accuracy(net.predict(Xtest),Ytest.T))
    return train_accuracy,training_loss,val_accuracy,val_loss

nets = [nn_linear,nn_lambda_2layer,nn_lambda_3layer,nn_lambda_4layer,nn_resnet]
for dataset in datasets:
    lst =[]
    for net in nets:
        acc = train_net(net,data_loader(dataset),epocs=200,lr=0.01,batch_size=32)
        lst.append(acc)

    lst = np.array(lst)
    lst_train_acc = lst[:,0,:]
    lst_train_loss = lst[:,1,:]
    lst_test_acc = lst[:,2,:]
    lst_test_loss = lst[:,3,:]

    fig, axs = plt.subplots(2, 2,figsize=(16,8))
    (ax1, ax2), (ax3, ax4) = axs
    for i,v in enumerate(lst_train_acc):
        ax1.set_title(f'{dataset}--train--accuracy')
        ax1.plot(v,label=f'model{i}layer-end acc : {v[-1]}')
        ax1.legend(loc=3)

    for i,v in enumerate(lst_test_acc):
        ax2.set_title(f'{dataset}--test--accuracy')
        ax2.plot(v,label=f'model{i}layer-end acc : {v[-1]}')
        ax2.legend(loc=3)

    for i,v in enumerate(lst_train_loss):
        ax3.set_title(f'{dataset}--train--loss')
        ax3.plot(v,label=f'model{i}layer-end loss : {v[-1]}')
        ax3.legend(loc=3)

    for i,v in enumerate(lst_test_loss):
        ax4.set_title(f'{dataset}--test--loss')
        ax4.plot(v,label=f'model{i}layer-end loss :{v[-1]}')
        ax4.legend(loc=3)
    plt.show()
```

```
netrwork layers:[OutputLayer(5, 5)]
netrwork layers:[HiddenLayer(5, 6), OutputLayer(6, 5)]
netrwork layers:[HiddenLayer(5, 6), HiddenLayer(6, 6), OutputLayer(6, 5)]
netrwork layers:[HiddenLayer(5, 6), ResNetLayer(6, 6), HiddenLayer(6, 6), OutputLayer(6, 5)]
```




```
netrwork layers:[OutputLayer(2, 5)]
netrwork layers:[HiddenLayer(2, 6), OutputLayer(6, 5)]
netrwork layers:[HiddenLayer(2, 6), HiddenLayer(6, 6), OutputLayer(6, 5)]
netrwork layers:[HiddenLayer(2, 6), ResNetLayer(6, 6), HiddenLayer(6, 6), OutputLayer(6, 5)]
```



We could observe that the worst model is model 1 : its because this model is just one linear layer with activation and our dataset problems need higher dimensions and separations , that’s why 2+

layer doing better in this problem.
The best model here is model 4 , composed out of 4 layer when one of them Resnet layer.

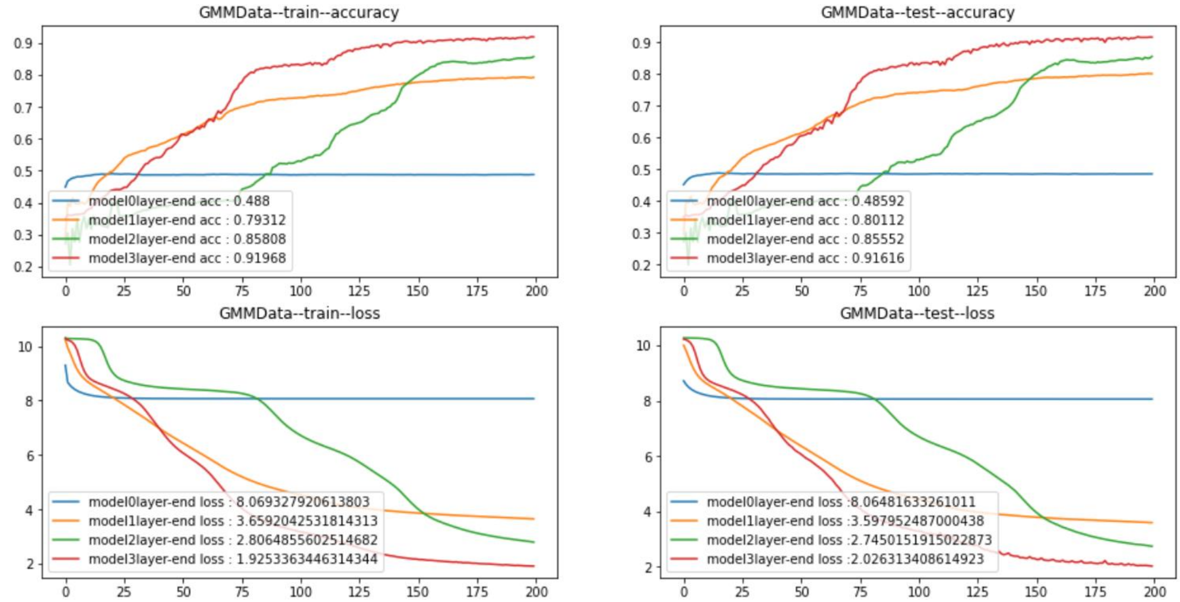
Section 5:

We repeat the previous section, only now use only 200 data points for training (sample them randomly).

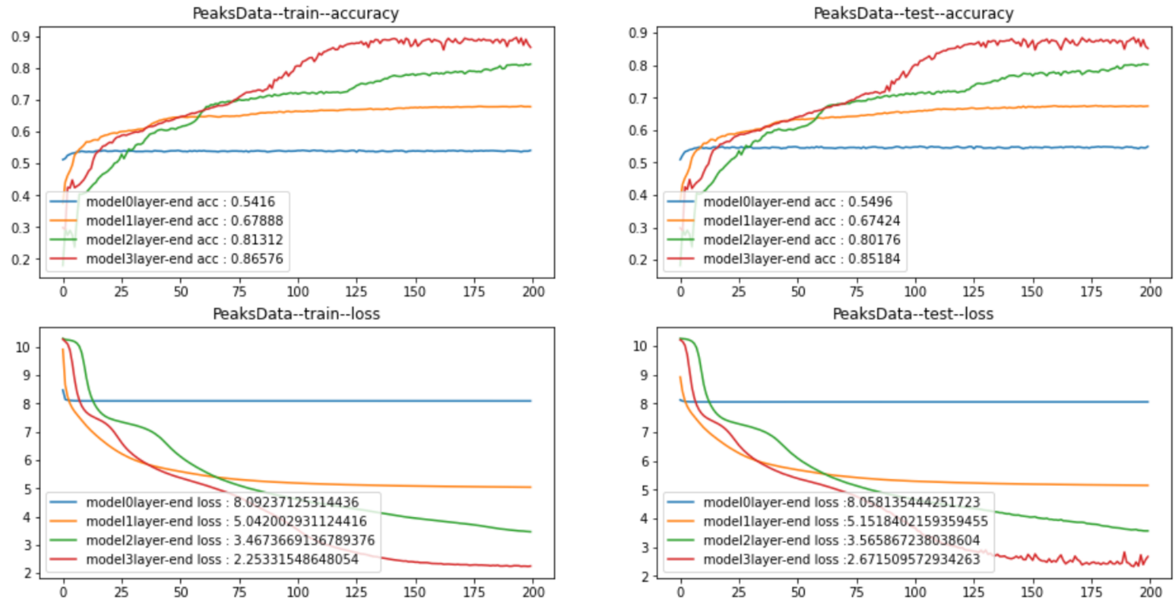
```
def train_net_200points(net_lambda,data,epocs=200,lr=0.1,batch_size=32):
    Xtrain,Ytrain,Xtest,Ytest= data
    feature_num,examples_num = Xtrain.shape
    labels_num = Ytrain.shape[0]
    net = net_lambda(feature_num,labels_num,lr)
    print(f'netrwork layers:{net.layers}')
    # Train loop
    a_t = []
    a_v = []
    train_accuracy =[]
    val_accuracy =[]
    training_loss = []
    val_loss =[]
    for epoc in range(epocs):
        # Shuffle train data
        indices = np.arange(examples_num)
        np.random.shuffle(indices)
        Xtrain = Xtrain[:,indices]
        Ytrain = Ytrain[:,indices]

        Xtrain = Xtrain[:200]
        Ytrain = Ytrain[:200]

        i = 0
        net.loss = 0
        while i * batch_size < 200:
            # Obtain minibatch
            batch_start = i * batch_size
            batch_end = min((i + 1) * batch_size, examples_num)

            minix = Xtrain[:,batch_start:batch_end]
            # transpoce the miniy to be inline with C
            miniy = Ytrain[:,batch_start:batch_end].T
            a_t.append(batch_end-batch_start)
            #forward pass
            net.forward(minix,miniy)

            #Backward+update nn params
            net.backward(miniy)

            #next batch...
            i += 1

        bias_fix = (i * batch_size - examples_num)/batch_size
        training_loss.append(net.loss/(i-1+bias_fix))
        train_accuracy.append(accuracy(net.predict(Xtrain),Ytrain.T))
        #training_loss.append(net.loss)

        # validation

        examples_num = Xtest.shape[1]
        indices = np.arange(examples_num)
        np.random.shuffle(indices)
        Xtest = Xtest[:,indices]
        Ytest = Ytest[:,indices]
        j = 0
        net.loss = 0

        while j * batch_size < examples_num:
            # Obtain minibatch
            batch_start = j * batch_size
            batch_end = min((j + 1) * batch_size, examples_num)

            minix = Xtest[:,batch_start:batch_end]
            # transpoce the miniy to be inline with C
            miniy = Ytest[:,batch_start:batch_end].T
            a_t.append(batch_end-batch_start)
```

```python
            #forward pass
            net.forward(minix,miniy, training=False )

            #next batch...
            j += 1

        bias_fix = (j * batch_size - examples_num)/batch_size
        val_loss.append(net.loss/(j-1+bias_fix))
        val_accuracy.append(accuracy(net.predict(Xtest),Ytest.T))
    return train_accuracy,training_loss,val_accuracy,val_loss
#%%
for dataset in datasets:
    lst =[]
    for net in nets:
        acc = train_net(net,data_loader(dataset),epocs=200,lr=0.01,batch_size=34)
        lst.append(acc)

    lst = np.array(lst)

    lst_train_acc = lst[:,0,:]
    lst_train_loss = lst[:,1,:]
    lst_test_acc = lst[:,2,:]
    lst_test_loss = lst[:,3,:]


    fig, axs = plt.subplots(2, 2,figsize=(16,8))
    (ax1, ax2), (ax3, ax4) = axs
    for i,v in enumerate(lst_train_acc):
        ax1.set_title(f'{dataset}--train--accuracy')
        ax1.plot(v,label=f'model{i+1}layer-end acc : {v[-1]}')
        ax1.legend(loc=3)

    for i,v in enumerate(lst_test_acc):
        ax2.set_title(f'{dataset}--test--accuracy')
        ax2.plot(v,label=f'model{i+1}layer-end acc : {v[-1]}')
        ax2.legend(loc=3)

    for i,v in enumerate(lst_train_loss):
        ax3.set_title(f'{dataset}--train--loss')
        ax3.plot(v,label=f'model{i+1}layer-end loss : {v[-1]}')
        ax3.legend(loc=3)

    for i,v in enumerate(lst_test_loss):
        ax4.set_title(f'{dataset}--test--loss')
        ax4.plot(v,label=f'model{i+1}layer-end loss :{v[-1]}')
        ax4.legend(loc=3)
    plt.show()
```

```
netrwork layers:[OutputLayer(5, 5)]
netrwork layers:[HiddenLayer(5, 6), OutputLayer(6, 5)]
netrwork layers:[HiddenLayer(5, 6), HiddenLayer(6, 6), OutputLayer(6, 5)]
netrwork layers:[HiddenLayer(5, 6), ResNetLayer(6, 6), HiddenLayer(6, 6), OutputLayer(6, 5)]
```




```
netrwork layers:[OutputLayer(2, 5)]
netrwork layers:[HiddenLayer(2, 6), OutputLayer(6, 5)]
netrwork layers:[HiddenLayer(2, 6), HiddenLayer(6, 6), OutputLayer(6, 5)]
netrwork layers:[HiddenLayer(2, 6), ResNetLayer(6, 6), HiddenLayer(6, 6), OutputLayer(6, 5)]
```



`when we compare the result we could see that in GMM we overfit the data what lead us to lower accuracy, but in Peaks its nearly the same.