Practical deep learning Assignment 1

Names: Dan Yarden 316611854 and Ofer Moses 311452171

**[A black and white image of a person's face

Description automatically generated with low confidence](https://github.com/ofer73/pdl)**

GitHub project:

Part 1: the classifier and optimizer

1. We implemented the soft max regression function as type of layer in the neural network, called “SoftMaxLayer”,which extends the basic Layer class:

class Layer:  
 def \_\_init\_\_(self, in\_dimensions, out\_dimensions, activation=ReLU):  
 *"""* ***:param*** *in\_dimensions: dimensions of input* ***:param*** *out\_dimensions: dimensions of output* ***:param*** *activation: activation function used by the layer  
 """* self.X = None  
 np.random.seed(0)  
 self.W = np.random.uniform(-1, 1, size=(out\_dimensions, in\_dimensions))  
 self.b = np.zeros((out\_dimensions, 1))  
 self.activation = activation.activate  
 self.activation\_derivative = activation.deriviative  
 self.dX = None # derivative with respect to input  
 self.dW = None # derivative with respect to Weight  
 self.db = None # derivative with respect to bias  
 self.train = True

class SoftMaxLayer(Layer):  
 def \_\_init\_\_(self, in\_dimensions, num\_of\_classes):  
 super(SoftMaxLayer, self).\_\_init\_\_(in\_dimensions, num\_of\_classes)  
 self.activation = lambda X: X  
  
 def forward(self, X):  
 self.X = X.copy()  
 out = self.activation(self.W @ X + self.b)  
  
 return out  
  
 def backward(self, V=None):  
 pass  
  
 def soft\_max(self, net\_out, Y):  
 *"""  
 description..* ***:param*** *net\_out: a matrix of size nxm, output of forward layer* ***:param*** *Y: a matrix of size lxm,   
 where Y[i,:] is c\_i (indicator vector for label i)* ***:return****: loss score, and probabilities matrix for each class,x  
 """* W = self.W.T  
 n = self.X.shape[0]  
 if len(self.X.shape) > 1:  
 m = self.X.shape[1]  
 else:  
 m = 1  
 l = self.W.shape[0]  
 self.dW = np.zeros((l, n))  
 self.db = np.zeros((l, 1))  
  
 ettas\_vector = get\_ettas(self.X, W, m, self.b)  
 scores = exp(net\_out - ettas\_vector)  
 right\_sum = np.sum(scores, axis=0)  
 probabilities = scores / right\_sum  
 loss = np.sum(Y \* np.log(probabilities))  
  
 # if during training, calculate gradients of loss and save  
 if self.train:  
 self.dW = (1 / m) \* (self.X @ (probabilities - Y).T).T  
 self.db = (1 / m) \* np.sum((probabilities - Y), axis=1).reshape(-1, 1)  
 self.dX = (1 / m) \* (W @ (probabilities - Y))  
  
 return -loss / m, probabilities

Gradient Tests for loss:

Weights:

def grad\_test\_soft\_max\_weights(X: np.array, Y: np.array):  
 iter\_num = 20  
 diff = np.zeros(iter\_num)  
 diff\_grad = np.zeros(iter\_num)  
 epsilons = [0.5 \*\* i for i in range(iter\_num)]  
 n = X.shape[0]  
 l = Y.shape[0]  
 if len(X.shape) > 1:  
 m = X.shape[1]  
 else:  
 m = 1  
 soft\_max\_layer = SoftMaxLayer(n, l)  
 d = normalize(np.random.rand(\*soft\_max\_layer.W.shape))  
 W\_orig = soft\_max\_layer.W.copy()  
 out = soft\_max\_layer.forward(X)  
 fw, \_ = soft\_max\_layer.soft\_max(out, Y)  
 grad\_w = soft\_max\_layer.dW  
 for i, epsilon in enumerate(epsilons):  
 W\_diff = W\_orig.copy()  
 W\_diff += d \* epsilon  
 soft\_max\_layer.W = W\_diff  
 out\_epsilon = soft\_max\_layer.forward(X)  
 fw\_epsilon, \_ = soft\_max\_layer.soft\_max(out\_epsilon, Y)  
 diff[i] = abs(fw\_epsilon - fw)  
 d\_flat = d.reshape(-1, 1)  
 grads\_flat = grad\_w.reshape(-1, 1)  
 diff\_grad[i] = abs(fw\_epsilon - fw - epsilon \* d\_flat.T @ grads\_flat)  
  
 plt.semilogy(np.arange(1, iter\_num + 1, 1), diff)  
 plt.semilogy(np.arange(1, iter\_num + 1, 1), diff\_grad)  
 plt.xlabel('epsilons')  
 plt.ylabel('difference')  
 plt.title('weights Grad Test Results')  
 plt.legend(("diff without grad", "diff with grad"))  
 plt.show()

Chart, line chart

Description automatically generated

Input(X):

def grad\_test\_soft\_max\_X(X: np.array, Y: np.array):  
 iter\_num = 20  
 X\_orig = X.copy()  
 diff = np.zeros(iter\_num)  
 diff\_grad = np.zeros(iter\_num)  
 epsilons = [0.5 \*\* i for i in range(iter\_num)]  
 n = X.shape[0]  
 l = Y.shape[0]  
 soft\_max\_layer = SoftMaxLayer(n, l)  
 d = normalize(np.random.rand(\*X.shape))  
 out = soft\_max\_layer.forward(X)  
 fx, \_ = soft\_max\_layer.soft\_max(out, Y)  
 grad\_x = soft\_max\_layer.dX.copy()  
 for i, epsilon in enumerate(epsilons):  
 X\_diff = X\_orig.copy()  
 X\_diff += d \* epsilon  
 out\_epsilon = soft\_max\_layer.forward(X\_diff)  
 fx\_epsilon, \_ = soft\_max\_layer.soft\_max(out\_epsilon, Y)  
 diff[i] = abs(fx\_epsilon - fx)  
 d\_flat = d.reshape(-1, 1)  
 grads\_flat = grad\_x.reshape(-1, 1)  
 diff\_grad[i] = abs(fx\_epsilon - fx - epsilon \* d\_flat.T @ grads\_flat)  
  
 plt.semilogy(np.arange(1, iter\_num + 1, 1), diff)  
 plt.semilogy(np.arange(1, iter\_num + 1, 1), diff\_grad)  
 plt.xlabel('epsilons')  
 plt.ylabel('difference')  
 plt.title('X Grad Test Results')  
 plt.legend(("diff without grad", "diff with grad"))  
 plt.show()

Chart, line chart

Description automatically generated

def grad\_test\_soft\_max\_bias(X: np.array, Y: np.array):  
 iter\_num = 20  
 diff = np.zeros(iter\_num)  
 diff\_grad = np.zeros(iter\_num)  
 epsilons = [0.5 \*\* i for i in range(iter\_num)]  
 n, m = X.shape  
 l = Y.shape[0]  
 soft\_max\_layer = SoftMaxLayer(n, l)  
 b\_original = soft\_max\_layer.b.copy()  
 d = normalize(np.random.rand(\*soft\_max\_layer.b.shape))  
 out = soft\_max\_layer.forward(X)  
 fb, \_ = soft\_max\_layer.soft\_max(out, Y)  
 grad\_b = soft\_max\_layer.db  
 for i, epsilon in enumerate(epsilons):  
 b\_diff = b\_original.copy()  
 b\_diff += d \* epsilon  
 soft\_max\_layer.b = b\_diff  
 out\_epsilon = soft\_max\_layer.forward(X)  
 fb\_epsilon, \_ = soft\_max\_layer.soft\_max(out\_epsilon, Y)  
 diff[i] = abs(fb\_epsilon - fb)  
 diff\_grad[i] = abs(fb\_epsilon - fb - epsilon \* d.T @ grad\_b)  
  
 plt.semilogy(np.arange(1, iter\_num + 1, 1), diff)  
 plt.semilogy(np.arange(1, iter\_num + 1, 1), diff\_grad)  
 plt.xlabel('epsilons')  
 plt.ylabel('difference')  
 plt.title('bias Grad Test Results')  
 plt.legend(("diff without grad", "diff with grad"))  
 plt.show()

Chart, line chart

Description automatically generated

2.SGD Optimizer:

We implemented the SGD optimizer as a class which keeps a pointer to the network, and uses it to update the parameters during the step() call using the gradients.

from network import NeuralNetwork  
  
  
class SGD:  
 def \_\_init\_\_(self, net: NeuralNetwork, lr=0.001):  
 self.net = net  
 self.lr = lr  
  
 def step(self):  
 for layer in self.net.layers:  
 layer.W = layer.W - self.lr \* layer.dW  
 layer.b = layer.b - self.lr \* layer.db

For the train code itself, we used the generic code we wrote for training a neural network, but created the network with the policy = “loss” parameter indicating it will only have 1 layer, the soft max regression:

def train\_network(data\_path: str, num\_layers=1, batch\_size: int = 32, lr: int = 0.001, epochs: int = 60):  
 data = sio.loadmat(data\_path)  
 X\_train = data["Yt"]  
 Y\_train = data["Ct"]  
 X\_test = data["Yv"]  
 Y\_test = data["Cv"]  
 input\_size = X\_train.shape[0]  
 m = X\_train.shape[1]  
 num\_of\_classes = Y\_train.shape[0]  
 net = NeuralNetwork(input\_size, num\_layers, num\_of\_classes, policy='loss')  
 optimizer = SGD(net, lr)  
 losses = np.zeros(epochs)  
 validation\_accuracy = np.zeros(epochs)  
 training\_accuracy = np.zeros(epochs)  
  
 for epoch in range(epochs):  
 net.train\_mode()  
 perm\_indices = np.random.permutation(m)  
 for j in range(0, m, batch\_size):  
 X\_batch = X\_train[:, perm\_indices[j:j + batch\_size]]  
 Y\_batch = Y\_train[:, perm\_indices[j:j + batch\_size]]  
  
 out = net.forward\_pass(X\_batch)  
 loss, probabilities = net.soft\_max\_layer.soft\_max(out, Y\_batch)  
 net.backward\_pass()  
 optimizer.step()  
 losses[epoch] += loss  
 training\_accuracy[epoch] += get\_acc(probabilities, Y\_batch)  
  
 losses[epoch] /= (m // batch\_size)  
 training\_accuracy[epoch] /= (m // batch\_size)  
 validation\_accuracy[epoch] = validate(net, X\_test, Y\_test)  
  
 print(f"epochs = {epoch}, loss = {losses[epoch]}, validation\_accuracy = {validation\_accuracy[epoch]}"  
 f" train\_accuracy = {training\_accuracy[epoch]}")  
  
 plt.plot(np.arange(0, epochs, 1), validation\_accuracy)  
 plt.xlabel("epochs")  
 plt.ylabel("score")  
 plt.legend("accuracy")  
 plt.title(f"accuracy : batchsize = {batch\_size} lr = {lr}")  
 plt.show()  
  
 plt.plot(np.arange(0, epochs, 1), losses)  
 plt.xlabel("epochs")  
 plt.ylabel("score")  
 plt.legend("loss")  
 plt.title(f"loss: batchsize = {batch\_size} lr = {lr}")  
 plt.show()  
  
  
def validate(net: NeuralNetwork, X\_test, Y\_test):  
 net.eval\_mode()  
 out = net.forward\_pass(X\_test)  
 \_, probabilities = net.soft\_max\_layer.soft\_max(out, Y\_test)  
 acc = get\_acc(probabilities, Y\_test)  
 return acc

we tried any combination of the parameters : batch sizes = [32,64,128],   
learning rates = [0.0001, 0.001, 0.05].

the results:

|  |  |  |  |
| --- | --- | --- | --- |
| Learning rate/ batch size | Loss | Train accuracy | Validate accuracy |
| 0.0001,32 | 1.344 | 0.42 | 0.42 |
| 0.0001,64 | 1.48 | 0.27 | 0.28 |
| 0.0001,128 | 1.65 | 0.18 | 0.18 |
| 0.001,32 | 1.22 | 0.57 | 0.57 |
| 0.001,64 | 1.23 | 0.56 | 0.56 |
| 0.001,128 | 1.26 | 0.53 | 0.53 |
| 0.05,32 | 1.22 | 0.57 | 0.56 |
| 0.05,64 | 1.22 | 0.57 | 0.57 |
| 0.05,128 | 1.22 | 0.57 | 0.57 |

Chart

Description automatically generatedWe chose lr = 0.001, batch size =32, as it provided us with the best results:

Chart, line chart

Description automatically generated

class NeuralNetwork:  
 def \_\_init\_\_(self, input\_size, num\_of\_layers, num\_of\_classes,

policy="constant"):  
  
 if policy == "constant" or num\_of\_layers == 1

# creating a constant sized network with num\_of\_layers layers  
 self.layers = [Layer(input\_size, input\_size) for i in

range(num\_of\_layers - 1)] +

[SoftMaxLayer(input\_size, num\_of\_classes)]

elif policy == "loss":  
 self.layers = [SoftMaxLayer(input\_size, num\_of\_classes)]  
  
 else:  
 # creating a list of layers, where we increase in dimensions each time

until the middle layer  
 # then we start decreasing again until the final loss layer with

output size num\_of\_classes  
 self.layers = [Layer(input\_size, 6)] +

[Layer(2 \* (i + 2), 2 \* (i + 3))

for i in range(1, (num\_of\_layers) // 2)] \  
 + [Layer(2 \* (i + 3), 2 \* (i + 2))

for i in range((num\_of\_layers) // 2 - 1, 0, -1)] \  
 + [SoftMaxLayer(6, num\_of\_classes)]  
  
 self.soft\_max\_layer = self.layers[-1]  
  
 def forward\_pass(self, X):  
 out = X  
 for layer in self.layers:  
 out = layer.forward(out)  
 return out  
  
 def backward\_pass(self):  
 prev\_dx = None  
 for layer in self.layers[::-1]:  
 layer.backward(prev\_dx)  
 prev\_dx = layer.dX.copy()

def train\_mode(self):  
 for layer in self.layers:  
 layer.train\_mode()  
  
 def eval\_mode(self):  
 for layer in self.layers:  
 layer.eval\_mode()

class Layer:  
 def \_\_init\_\_(self, in\_dimensions, out\_dimensions, activation=ReLU):  
 *"""* ***:param*** *in\_dimensions: dimensions of input* ***:param*** *out\_dimensions: dimensions of output* ***:param*** *activation: activation function used by the layer  
 """* self.X = None  
 np.random.seed(0)  
 self.W = np.random.uniform(-1, 1,

size=(out\_dimensions, in\_dimensions))  
 self.b = np.zeros((out\_dimensions, 1))  
 self.activation = activation.activate  
 self.activation\_derivative = activation.deriviative  
 self.dX = None # derivative with respect to input  
 self.dW = None # derivative with respect to Weight  
 self.db = None # derivative with respect to bias  
 self.train = True  
  
 def forward(self, X):  
 self.X = X.copy()  
 out = self.activation(self.W @ X + self.b)  
 return out  
  
 def backward(self, V):  
 temp = self.activation\_derivative(self.W @ self.X + self.b) \* V  
 self.dX = self.W.T @ temp  
 self.dW = temp @ self.X.T  
 self.db = np.sum(temp, axis=1).reshape(-1, 1)  
  
 def train\_mode(self):  
 self.train = True  
  
 def eval\_mode(self):  
 self.train = False

class SoftMaxLayer(Layer):  
***:param*** *Y: a matrix of size lxm,  
 where Y[i,:] is c\_i (indicator vector for label i)* ***:return****: loss score, and probabilities matrix for each class,x  
 """*(Included Above In Part 1)

Activation functions:

class ReLU:  
 @staticmethod  
 def activate(x):  
 return np.maximum(0, x)  
  
 @staticmethod  
 def deriviative(x):  
 f = lambda t: 1 if t >= 0 else 0  
 vfunc = np.vectorize(f)  
  
 return vfunc(x)  
  
class tanh:  
 @staticmethod  
 def activate(x):  
 return np.tanh(x)@staticmethod  
 def deriviative(x):  
 f = lambda X: 1 - np.tanh(X) \*\* 2  
 return f(x)

Jacobian Tests (conducted with tanh)

we used the shortcut version, defining a function

and performing a grad test for it, as

.

Input(X):

def jacobian\_test\_layer\_X(X):  
 layer = Layer(2, 3)  
 n, m = X.shape  
 out\_dimensions = layer.b.shape[0]  
 U = normalize(np.random.rand(out\_dimensions, m))  
  
 iter\_num = 20  
 diff = np.zeros(iter\_num)  
 diff\_grad = np.zeros(iter\_num)  
 epsilons = [0.5 \*\* i for i in range(iter\_num)]  
 d = normalize(np.random.rand(\*X.shape))  
  
 fx = np.dot(layer.forward(X).T, U).item()  
 layer.backward(U)  
 JacTu\_X = layer.dX  
  
 for i, epsilon in enumerate(epsilons):  
 X\_diff = X.copy()  
 X\_diff += d \* epsilon  
 fx\_epsilon = np.dot(layer.forward(X\_diff).T, U).item()  
 d\_flat = d.reshape(-1, 1)  
 JacTu\_X\_flat = JacTu\_X.reshape(-1, 1)  
  
 diff[i] = abs(fx\_epsilon - fx)  
 diff\_grad[i] = abs(fx\_epsilon - fx - epsilon \* d\_flat.T @ JacTu\_X\_flat)  
 plt.semilogy(np.arange(1, iter\_num + 1, 1), diff)  
 plt.semilogy(np.arange(1, iter\_num + 1, 1), diff\_grad)  
 plt.xlabel('epsilons')  
 plt.ylabel('difference')  
 plt.title('X Jacobian Test Results')  
 plt.legend(("diff without grad", "diff with grad"))

Chart, line chart

Description automatically generated

Weights:

def jacobian\_test\_layer\_W(X):  
 layer = Layer(2, 3)  
 n, m = X.shape  
 out\_dimensions = layer.b.shape[0]  
 U = normalize(np.random.rand(out\_dimensions, m))  
 original\_W = layer.W.copy()  
  
 iter\_num = 20  
 diff = np.zeros(iter\_num)  
 diff\_grad = np.zeros(iter\_num)  
 epsilons = [0.5 \*\* i for i in range(iter\_num)]  
 d = normalize(np.random.rand(\*layer.W.shape))  
 fw = np.dot(layer.forward(X).T, U).item()  
 layer.backward(U)  
 JacTu\_W = layer.dW  
  
 for i, epsilon in enumerate(epsilons):  
 W\_diff = original\_W.copy()  
 W\_diff += d \* epsilon  
 layer.W = W\_diff  
 fw\_epsilon = np.dot(layer.forward(X).T, U).item()  
 diff[i] = abs(fw\_epsilon - fw)  
 d\_flat = d.reshape(-1, 1)  
 JacTu\_W\_flat = JacTu\_W.reshape(-1, 1)  
 diff\_grad[i] = abs(fw\_epsilon - fw - epsilon \* d\_flat.T @ JacTu\_W\_flat)  
  
 plt.semilogy(np.arange(1, iter\_num + 1, 1), diff)  
 plt.semilogy(np.arange(1, iter\_num + 1, 1), diff\_grad)  
 plt.xlabel('epsilons')  
 plt.ylabel('difference')  
 plt.title('weights Jacobian Test Results')  
 plt.legend(("diff without grad", "diff with grad"))  
 plt.show()

Chart, line chart

Description automatically generated

Bias:

def jacobian\_test\_layer\_b(X):  
 layer = Layer(2, 3)  
 n, m = X.shape  
 out\_dimensions = layer.b.shape[0]  
 U = normalize(np.random.rand(out\_dimensions, m))  
 original\_b = layer.b.copy()  
  
 iter\_num = 20  
 diff = np.zeros(iter\_num)  
 diff\_grad = np.zeros(iter\_num)  
 epsilons = [0.5 \*\* i for i in range(iter\_num)]  
 d = normalize(np.random.rand(\*layer.b.shape))  
 fb = np.dot(layer.forward(X).T, U).item()  
 layer.backward(U)  
 JacTu\_b = layer.db  
 for i, epsilon in enumerate(epsilons):  
 b\_diff = original\_b.copy()  
 b\_diff += d \* epsilon  
 layer.b = b\_diff  
 fb\_epsilon = np.dot(layer.forward(X).T, U).item()  
 diff[i] = abs(fb\_epsilon - fb)  
 diff\_grad[i] = abs(fb\_epsilon - fb - epsilon \* d.T @ JacTu\_b)  
  
 plt.semilogy(np.arange(1, iter\_num + 1, 1), diff)  
 plt.semilogy(np.arange(1, iter\_num + 1, 1), diff\_grad)  
 plt.xlabel('epsilons')  
 plt.ylabel('difference')  
 plt.title('bias Jacobian Test Results')  
 plt.legend(("diff without grad", "diff with grad"))  
 plt.show()

Chart, line chart

Description automatically generated

2.2.3 check if should add more output

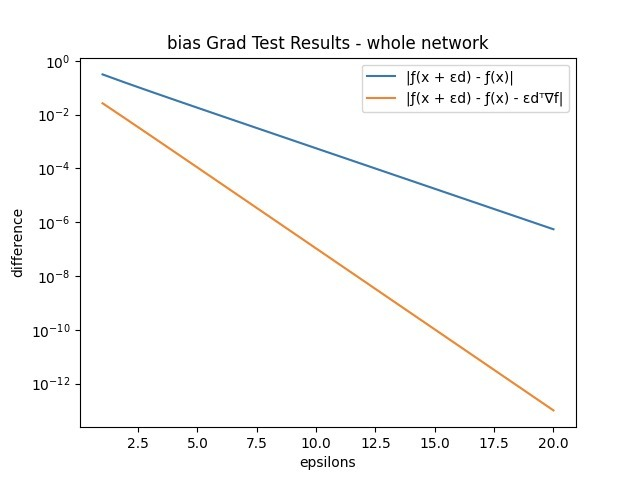
Grad Tests for the whole network:

Weights:

Chart, line chart

Description automatically generated

Bias:



3.Expiriments:

We experimented with a few layer sizes ranging between 1-16, where in the first l/2 layers the dimensions increase by 2 in each forward pass, then in the last l/2 layers we decrease back by 2 each pass.

Chart

Description automatically generated

Peaks Data

Chart, line chart

Description automatically generated

Chart

Description automatically generated

Chart, line chart

Description automatically generated

GMM Data

Chart

Description automatically generated

Chart

Description automatically generated

Conclusions:

As we can see from the graphs above, in both the data sets of GMM and Peaks, the best results were achieved using a network with 12 layers (

In GMM: best epoch achieved 95.8% train accuracy, 95.9% validation accuracy, and 0.2 loss score.

In Peaks: best epoch achieved 92.6% train accuracy, 92.9% validation accuracy, and 0.26 loss score.

Between 1 – 12 layers we can the results (both accuracy and loss) improve as the layer number increase, and in the 16 layers version we witness a decrease in performance.

Also, as we increase in the layers number (especially with 12 and 16) we can see that the validation accuracy becomes less stable, with substantial changes between epochs.