Deep Learning HW1 - Spring 2023

Aniket Sinha - aas10120

Deadline: 17th February 2023

1. (3 points) Linear regression with non-standard losses. In class we derived an analytical expression for the optimal linear regression model using the least squares loss. If X is the matrix of n training data points (stacked row-wise) and y is the vector of their corresponding labels, then:

   (a) Using matrix/vector notation, write down a loss function that measures the training error in terms of the l1-norm. Write down the sizes of all matrices/vectors.

   (b) Can you simply write down the optimal linear model in closed form, as we did for standard linear regression? If not, why not?

   (c) In about 2-3 sentences, reflect on how you solved this question, and relate any aspects of this to the 3-step recipe for ML that we discussed in class.

ANSWER:

1(a) The predicted value of $\hat{Y}$ is given by the following equation:

$\hat{Y}_{n\times 1} = X_{n\times d} \times W_{d\times 1}$

Loss J in the form of L1 norm is given by :

$J(W, X, Y) = \mid Y - \hat{Y} \mid$

$\Rightarrow J(W, X, Y) = \mid Y_{n\times 1} - (X_{n\times d} \times W_{d\times 1}) \mid$

$\Rightarrow J(W, X, Y) = \Sigma_{i=1}^{n} \mid (y_i - (w_1 \times x_{i1} + \cdots + w_d \times x_{id})) \mid$

And the Dimensions are:

$$X = n \times d$$
$$w = d \times 1$$
$$y = n \times 1$$

1(b) No, we can not simply write down linear model in closed form as we did for standard linear regression.This is due to the fact that L1 norm is non-differentiable. The L1 norm is evaluated on the absolute value of the difference, and the absolute value of $\mid x \mid$ contains a kink at $x = 0$ and is therefore non-differentiable at that point. In addition, the process of computing the inverse of an $n \times n$ matrix can be very time-consuming, with a temporal complexity of O(n^3), and this can become even more pronounced as n grows larger.

1(c) This approach to solving the problem is connected to the three-step process for machine learning discussed in class. The first step involves choosing a model and specifying a loss function, in this instance the least squares loss. The second step requires the optimization of the model by reducing the loss function through setting the gradient of the loss function to zero. In this case, this resulted in an analytical expression for the model. Finally, the third step involves evaluating the model to ensure that it is performing well on the data, which may involve further testing and tuning.

2. (4 points) Expressivity of neural networks. Recall that the functional form for a single neuron is given by $y = \sigma(\langle w, x \rangle + b, 0)$, where x is the input and y is the output. In this exercise, assume that x and y are 1-dimensional (i.e., they are both just real-valued scalars) and σ is the unit step activation. We will use multiple layers of such neurons to approximate pretty much any function f. There is no learning/training required for this problem; you should be able to guess/derive the weights and biases of the networks by hand.

   (a) A box function with height hand width δ is the function $f(x) = h$ for $0 < x < \delta$ and 0 otherwise. Show that a simple neural network with 2 hidden neurons with step activation's can realize this function. Draw this network and identify all the weights and biases. (Assume that the output neuron only sums up inputs and does not have a non linearity.)

   (b) Do you think the argument in part b can be extended to the case of $d-$dimensional inputs? (i.e., where the input x is a vector – think of it as an image, or text query, etc). If yes, comment on potential practical issues involved in defining such networks. If not, explain why not.

   (c) Now suppose that f is any arbitrary, smooth, bounded function defined over an interval $[-B, B]$.(You can ignore what happens to the function outside this interval, or just assume it is zero). Use part a to show that this function can be closely approximated by a neural network with a hidden layer of neurons. You don't need a rigorous mathematical proof here; a handwavy argument or even a figure is okay here, as long as you convey the right intuition.

(d) In about 2-3 sentences, reflect on how you solved this question, and relate any aspects of this to the 3-step recipe for ML that we discussed in class.

ANSWER:

2(a) A simple neural network with 2 hidden neurons with step activation's can realize this function In order to show this, we will try to realise the function $f(x) = h$, for $0 < x < \delta$ and show that it can be implemented with 2 neurons:

let w be the weight and b be the bias

First Neuron

$$w = 1(\text{weight})\ ,\ b = 0(\text{bias})$$

$$z_1 = w \times x + b = x$$

$$\text{Activation } \sigma_1 = \begin{cases} 1\ ,\ \text{if } z \geq 0, \\ 0\ ,\ \text{otherwise} \end{cases} \tag{1}$$

$$\text{Output } u_1 = \begin{cases} 1\ ,\ \text{if } x \geq 0, \\ 0\ ,\ \text{otherwise} \end{cases}$$

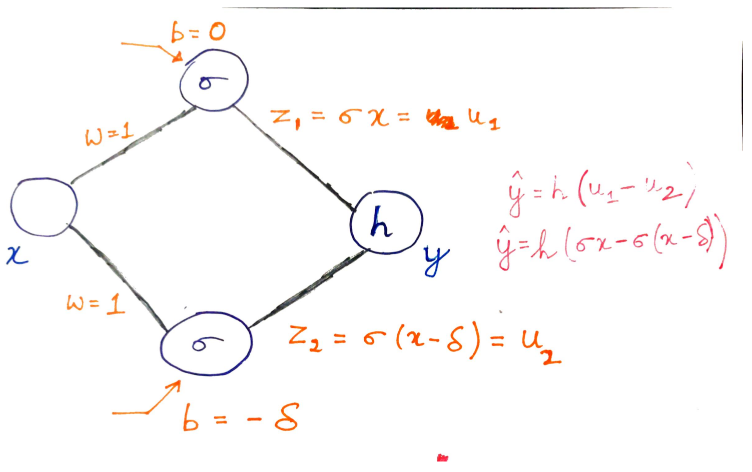
Second Neuron Similarly for the Second Neuron

$$w = 1\ ,\ b = -\delta$$

$$z_2 = w \times x + b = x - \delta$$

$$\text{Activation } \sigma_2 = \begin{cases} 1\ ,\ \text{if } z \geq 0, \\ 0\ ,\ \text{otherwise} \end{cases} \tag{2}$$

$$\text{Output } u_2 = \begin{cases} 1\ ,\ \text{if } x \geq \delta, \\ 0\ ,\ \text{otherwise} \end{cases}$$



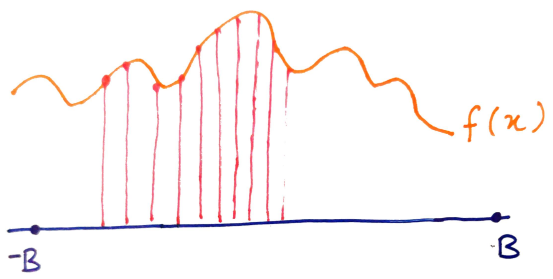
Final Output Layer: Outputs from Neuron 1 and Neuron 2 are added. Now, to realize the function, multiplier of h is used.

$$\hat{y} = h \times (u_1 - u_2) \tag{3}$$

- **Case I :** $x < 0$, then $u_1 = 0$, $u_2 = 0$, $\hat{y} = 0$
- **Case II :** $0 < x < \delta$, then $u_1 = 1$, $u_2 = 0$, $\hat{y} = h$
- **Case III :** $x > \delta$, then $u_1 = 1$, $u_2 = 1$, $\hat{y} = 0$

Based on the above solution, we can conclude that it is possible to achieve a box function using a single hidden layer consisting of two neurons.

2(b) Given that: $f(x) = y$,where x $\epsilon[-B, B]$ where f is a smooth and continuous function.



The function can be visualized as a curve that is composed from infinitely large number of infinitesimally small box function of height $f(x)$ and width $\triangle x$ where $\triangle x \rightarrow 0$ as shown above. We can represent each individual box or segment using a combination of two hidden neurons, and as a result, it becomes possible to approximate a continuous function by selecting a set of **N** data points for all the boxes/segments with a height equivalent to the value of $f(x)$. Hence, the function can be realized by maximum of **2N** neurons if **N** is sufficiently large.

2(c) The argument presented in part b can be applied to $d-$dimensional inputs by first flattening the input in X into $N \times d$ data points. However, this presents a significant challenge since the boxes would also be $d-$dimensional, and the number of required neurons would be $2 \times N \times d$. This is a concern because, N is already very large and as the complexity of the function increases, the number of required neurons grows rapidly, making the approach computationally infeasible.

2(d) This problem does relate to the 3-step recipe for ML that was taught in class as it involves selecting a Neural Network model and defining its architecture as the first step. The second step involves optimizing the model by setting its weights and biases, while the third step involves evaluating the model's performance, which was not necessary in this case as the weights and biases were derived to closely approximate the target functions. Ultimately, this problem emphasizes the expressive capabilities of neural networks in approximating any function.

3. (3 points) Calculating gradients. Suppose that z is a vector with n elements. We would like to compute the gradient of $y = \text{softmax}(z)$. Show that the Jacobian of y with respect to z, J, is given by the

$$J_{ij} = \frac{\partial y_i}{\partial z_j} = y_i(\partial_{ij} - y_j)$$

where δ_{ij} is the Dirac delta, i.e., 1 if $i = j$ and 0 else. Hint: Your algebra could be simplified if you try computing the log derivative, $\frac{\partial log y_i}{\partial z_j}$

Answer:

We are given a z vector which equals $[z_i]_{n\times 1}$ and $y = \text{softmax}(z)$

Which gives us softmax operation

$$y_i = \frac{e^{z_i}}{\Sigma_{j=1}^{n} e^{z_j}}$$

Taking log on both the LHS and the RHS of the above equation:

$$log(y_i) = z_i - log(\sum_{j=1}^{n} e^{z_j})$$

We want to show $J_{ij} = \frac{\partial y_i}{\partial z_j}$, Taking partial derivative w.r.t. z_j on the above equation and solving the equation so that it becomes $\Rightarrow$

$$\frac{1}{y_i} \times \frac{\partial y_i}{\partial z_j} = \frac{\partial z_i}{\partial z_j} + \frac{1}{\Sigma_{j=1}^{n} e^{z_j}} \times e^{z_j}$$

Now, the above resultant equation gets multiplied by y_i on both sides. Therefore, we get $\Rightarrow$

$$J_{ij} = \frac{\partial y_i}{\partial z_j} = \begin{cases} y_i \times (1 - y_j) \text{ , if } i = j, \\ -y_i \times y_j \text{ , otherwise} \end{cases}$$

Now, we have been given in the question that Dirac delta δ_{ij} is $\Rightarrow$

$$\delta_{ij} = \begin{cases} 1 \text{ , if } i = j, \\ 0 \text{ , otherwise} \end{cases}$$

Now, using the above two equations, we get $\Rightarrow$

$$J_{ij} = \frac{\partial y_i}{\partial z_j} = y_i(\delta_{ij} - y_j)$$

Question 4

SOURCE : https://github.com/kvgarimella/dl-demos/blob/main/demo01-basics.ipynb

The code is from the demo1 lab. Added few modifications to the demo1 code as per the requirements. Added comments to show the changes made.

```
import numpy as np
import torch
import torchvision
import tensorflow as tf
import matplotlib.pyplot as plt

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

Training and test data

We are using a popular image dataset called Fashion-MNIST.

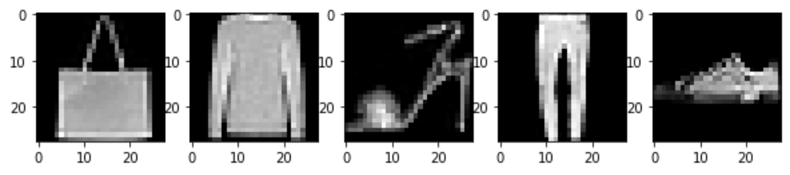
```
trainingdata = torchvision.datasets.FashionMNIST('./FashionMNIST/',train=True,download=True,transform=torchvision.transforms.ToTen
testdata = torchvision.datasets.FashionMNIST('./FashionMNIST/',train=False,download=True,transform=torchvision.transforms.ToTenso
```

```
print(len(trainingdata))
print(len(testdata))
```

```
    60000
    10000
```

```
trainDataLoader = torch.utils.data.DataLoader(trainingdata,batch_size=64,shuffle=True)
testDataLoader = torch.utils.data.DataLoader(testdata,batch_size=64,shuffle=False)
```

```
import matplotlib.pyplot as plt
%matplotlib inline
images, labels = next(iter(trainDataLoader))
plt.figure(figsize=(10,4))
for index in np.arange(0,5):
 plt.subplot(1,5,index+1)
 plt.imshow(images[index].squeeze().numpy(),cmap=plt.cm.gray)
```



Neural Network Model

Reference: https://pytorch.org/tutorials/

Using a dense neural network with three hidden layers with 256, 128, and 64 neurons respectively, all with ReLU activations, we do not add the RELU to the last layer

What is happening here?

The below code cell creates a neural network model named **LinearReg** with four fully-connected layers and a ReLU activation function. The input data is flattened before being passed through the network, which outputs a tensor of size 10. The instantiated model is then moved to the specified device to train using a cross-entropy loss function and an Adam optimizer.

```
class LinearReg(torch.nn.Module):
    def __init__(self):
        super(LinearReg, self).__init__()
        self.flatten = torch.nn.Flatten()
        # Define a feed-forward neural network with four linear layers
        # The first layer takes input of size 28*28 and outputs a tensor of size 256
        # The second layer takes input of size 256 and outputs a tensor of size 128
        # The third layer takes input of size 128 and outputs a tensor of size 64
        # The fourth layer takes input of size 64 and outputs a tensor of size 10
        self.linear_relu_stack = torch.nn.Sequential(
            torch.nn.Linear(28*28, 256),
            torch.nn.ReLU(),
            torch.nn.Linear(256, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, 64),
            torch.nn.ReLU(),
            torch.nn.Linear(64, 10),
            )

    def forward(self, x):

        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

net = LinearReg().to(device)
# Define the loss function
Loss = torch.nn.CrossEntropyLoss()
# Define the optimizer and set the learning rate to 0.001
optimizer = torch.optim.Adam(net.parameters(), lr=0.001)
```

WHat is happening in the below cell?

In the below cell, we are using a training loop for a deep learning model, where the model is being trained on a training dataset and evaluated on a separate test dataset. The loss and accuracy for both datasets are recorded.We have used PyTorch framework here. Finally, the model's parameters are updated with an optimizer, and the loss and accuracy are calculated for each epoch.

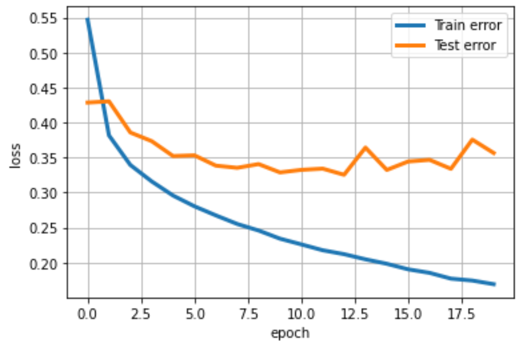
```
# create empty lists to store loss and accuracy history for training and testing
train_loss_history = []
test_loss_history = []
train_accuracy_history = []
test_accuracy_history = []

# loop over 20 epochs for training and testing
for epoch in range(20):
    # initialize loss and score variables for training and testing
    train_loss = 0.0
    test_loss = 0.0
    train_score = 0.0
    test_score = 0.0

    # loop over training data
    for i, data in enumerate(trainDataLoader):
        images, labels = data
        images = images.to(device)
        labels = labels.to(device)

        # zero the gradients, make forward pass, calculate loss and do backward pass
        optimizer.zero_grad()
        predicted_output = net(images)
        _, predictions = torch.max(predicted_output, 1)
        fit = Loss(predicted_output,labels)
        fit.backward()
        optimizer.step()
        # accumulate loss and score for training
        train_loss += fit.item()
        train_score += (predictions == labels).sum().item()
    # loop over testing data
    for i, data in enumerate(testDataLoader):
        with torch.no_grad():
            images, labels = data
            images = images.to(device)
            labels = labels.to(device)
            # make forward pass, calculate loss
            predicted_output = net(images)
            _, predictions = torch.max(predicted_output, 1)
            fit = Loss(predicted_output, labels)
            fit = Loss(predicted_output, labels)
            test_loss += fit.item()
            test_score += (predictions == labels).sum().item()

    # calculate average loss and accuracy for training and testing
    train_loss = train_loss/len(trainDataLoader)
    test_loss = test_loss/len(testDataLoader)
    train_accuracy = (train_score/len(trainDataLoader.dataset))*100
    test_accuracy = (test_score / len(testDataLoader.dataset))*100
    train_loss_history.append(train_loss)
    test_loss_history.append(test_loss)
    train_accuracy_history.append(train_accuracy)
    test_accuracy_history.append(test_accuracy)

    # print current epoch's loss and accuracy
    print('Epoch %s, Train loss = %s, Test loss = %s, Train Accuracy = %s, Test Accuracy = %s' % (epoch, train_loss, test_loss
```

```
Epoch 0, Train loss = 0.5468868662807733, Test loss = 0.4285319443720921, Train Accuracy = 80.30666666666667, Test Accuracy
Epoch 1, Train loss = 0.3815843195183826, Test loss = 0.4302642548539836, Train Accuracy = 86.04166666666667, Test Accuracy
Epoch 2, Train loss = 0.33932765883836413, Test loss = 0.38574347013880494, Train Accuracy = 87.44166666666666, Test Accurac
Epoch 3, Train loss = 0.3158076376850798, Test loss = 0.37336183837644615, Train Accuracy = 88.39333333333333, Test Accuracy
Epoch 4, Train loss = 0.295614047655101, Test loss = 0.3520544744600916, Train Accuracy = 89.03999999999999, Test Accuracy =
Epoch 5, Train loss = 0.2802640836217256, Test loss = 0.3530790074995369, Train Accuracy = 89.525, Test Accuracy = 87.72
Epoch 6, Train loss = 0.26728196495345663, Test loss = 0.3383779723173494, Train Accuracy = 89.99333333333334, Test Accuracy
Epoch 7, Train loss = 0.2550260498166593, Test loss = 0.33524139875629144, Train Accuracy = 90.42166666666667, Test Accuracy
Epoch 8, Train loss = 0.24567348114463058, Test loss = 0.3405934758721643, Train Accuracy = 90.80166666666668, Test Accuracy
Epoch 9, Train loss = 0.23403360683526567, Test loss = 0.328690578461073, Train Accuracy = 91.295, Test Accuracy = 88.48
Epoch 10, Train loss = 0.2258221715815794, Test loss = 0.3323224022699769, Train Accuracy = 91.43666666666667, Test Accuracy
Epoch 11, Train loss = 0.2174694082900278, Test loss = 0.334088995388359, Train Accuracy = 91.79333333333334, Test Accuracy
Epoch 12, Train loss = 0.2118684327317231, Test loss = 0.325310358291219, Train Accuracy = 91.97999999999999, Test Accuracy
Epoch 13, Train loss = 0.2045213447959184, Test loss = 0.3641901374527603, Train Accuracy = 92.16833333333334, Test Accuracy
Epoch 14, Train loss = 0.19812665546912628, Test loss = 0.3322423220534993, Train Accuracy = 92.45666666666666, Test Accurac
Epoch 15, Train loss = 0.19021243181055797, Test loss = 0.3443082585029162, Train Accuracy = 92.74, Test Accuracy = 88.81
Epoch 16, Train loss = 0.18515983671307373, Test loss = 0.34665453359247395, Train Accuracy = 92.94166666666666, Test Accura
Epoch 17, Train loss = 0.17689812694912527, Test loss = 0.3338402298985014, Train Accuracy = 93.31333333333333, Test Accurac
Epoch 18, Train loss = 0.17416429554404161, Test loss = 0.3755828378496656, Train Accuracy = 93.36, Test Accuracy = 88.96
Epoch 19, Train loss = 0.16899226787390867, Test loss = 0.35644717794504893, Train Accuracy = 93.61500000000001, Test Accura
```

Plotting the Training and Testing Error

```
plt.plot(range(20),train_loss_history,'-',linewidth=3,label='Train error')
plt.plot(range(20),test_loss_history,'-',linewidth=3,label='Test error')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.grid(True)
plt.legend()
```

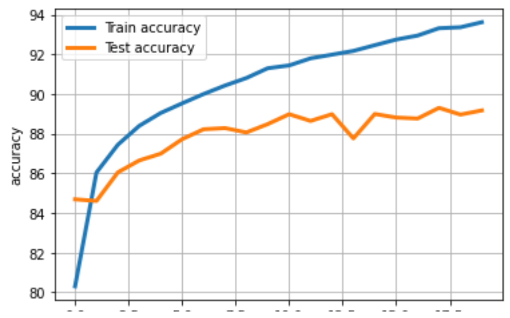
```
<matplotlib.legend.Legend at 0x7fc46ab494f0>
```



Plotting the Training and Testing Accuracy

```
plt.plot(range(20),train_accuracy_history,'-',linewidth=3,label='Train accuracy')
plt.plot(range(20),test_accuracy_history,'-',linewidth=3,label='Test accuracy')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.grid(True)
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fc46a1bfe20>
```



What is happening here?

Here we are selecting three images at random from the chosen dataset.

```
# Selecting 3 random images from the image dataset
l =np.random.randint(15, size=(3))
Image_1 = images[l[0]]
Image_2 = images[l[1]]
Image_3 = images[l[2]]
img_labels = np.arange(10)
```

What is happening in the below cell?

The "get_prediction" function takes a list of probabilities for each class label and a dictionary that maps class label indices to their names. It returns the name and index of the predicted class label by finding the index with the highest probability in the input list and looking up its name in the dictionary

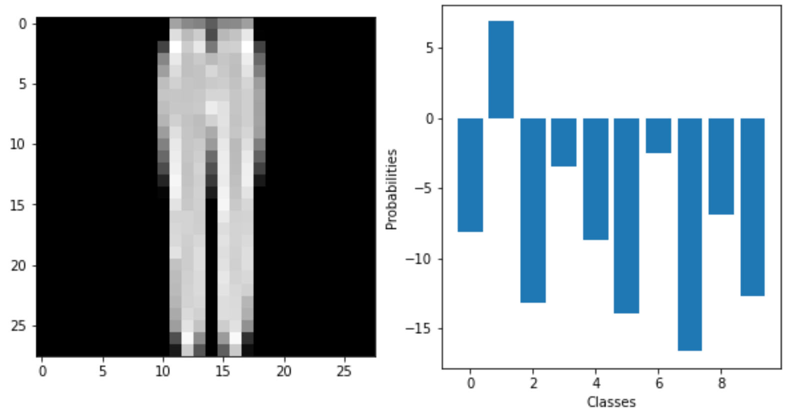
```
# Define a dictionary that maps class indices to their names
dict_classes = { 0: "T-shirt/Top",
                 1: "Trouser",
                 2: "Pullover",
                 3: "Dress",
                 4: "Coat",
                 5: "Sandal",
                 6: "Shirt",
                 7: "Sneaker",
                 8: "Bag",
                 9: "Ankle Boot"
                 }
# Define a function to get the predicted class label and index from the output probabilities
def get_prediction(output, dict_classes):
    # Get the maximum probability value from the output list
    get_max = max(output[0])
    # Convert the output tensor to a list and find the index of the maximum value
    li = output[0].tolist()
    idx = li.index(get_max)
    return dict_classes[idx] , idx
```

What are we doing here?

We are loading the random image selected earlier and passing it through the neural network to obtain the output prediction. Then, we are generating a plot with the input image and a bar graph of the predicted class probabilities. Finally, we print out the predicted class name and its index with the highest probability. The same is being done for all the three images.

First Image

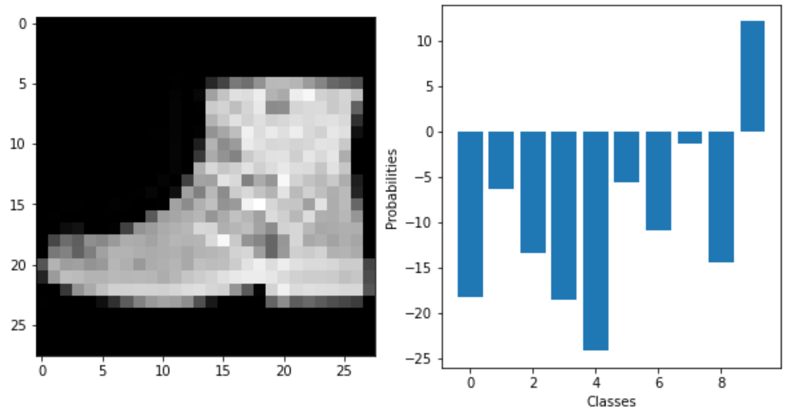
```
image = Image_1.to(device)
output = net(image)
output = output.to(device)
fig = plt.figure(figsize = (10,5))
plt.subplot(1,2,1)
plt.imshow(Image_1.cpu().squeeze().numpy(),cmap=plt.cm.gray)
plt.subplot(1,2,2)
plt.bar(img_labels, output[0].cpu().detach())
plt.xlabel("Classes")
plt.ylabel("Probabilities")
plt.show()
print('Image selected: '+ get_prediction(output, dict_classes)[0] )
print('Class which has the highest Probability: '+ str(get_prediction(output, dict_classes)[1]))
print('Name of the Class having the highest Probablity: '+ str(get_prediction(output, dict_classes)[0]))
```



```
Image selected: Trouser
Class which has the highest Probability: 1
Name of the Class having the highest Probablity: Trouser
```

Second Image

```
image = Image_2.to(device)
output = net(image)
output = output.to(device)
fig = plt.figure(figsize = (10, 5))
plt.subplot(1,2,1)
plt.imshow(Image_2.cpu().squeeze().numpy(),cmap=plt.cm.gray)
plt.subplot(1,2,2)
plt.bar(img_labels, output[0].cpu().detach())
plt.xlabel("Classes")
plt.ylabel("Probabilities")
plt.show()
print('Image selected: '+ get_prediction(output, dict_classes)[0] )
print('Class which has the highest Probability: '+ str(get_prediction(output, dict_classes)[1]))
print('Name of the Class having the highest Probablity: '+ str(get_prediction(output, dict_classes)[0]))
```

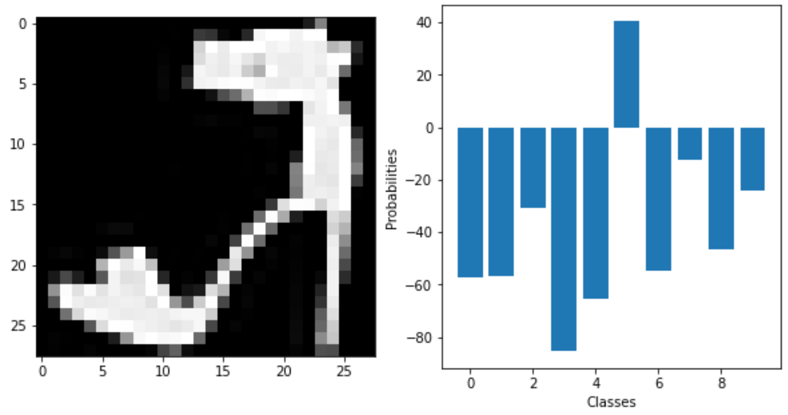


```
Image selected: Ankle Boot
Class which has the highest Probability: 9
Name of the Class having the highest Probablity: Ankle Boot
```

Third Image

```
image = Image_3.to(device)
output = net(image)
output = output.to(device)
```

```
fig = plt.figure(figsize = (10, 5))
plt.subplot(1,2,1)
plt.imshow(Image_3.cpu().squeeze().numpy(),cmap=plt.cm.gray)
plt.subplot(1,2,2)
plt.bar(img_labels, output[0].cpu().detach())
plt.xlabel("Classes")
plt.ylabel("Probabilities")
plt.show()
print('Image selected: '+ get_prediction(output, dict_classes)[0] )
print('Class which has the highest Probability: '+ str(get_prediction(output, dict_classes)[1]))
print('Name of the Class having the highest Probablity: '+ str(get_prediction(output, dict_classes)[0]))
```



```
Image selected: Sandal
Class which has the highest Probability: 5
Name of the Class having the highest Probablity: Sandal
```

Final Comments : About the observations above

From Observing the images and the classes above,We can see that the predictions made by our model is correct.

Image 1: looks like **Trousers**, and the model makes the prediction of **Trousers** and hence, **Trousers** class has the highest probablity

Image 2: looks like a **Ankle Boot**, and the model makes the prediction of **Ankle Boot** and hence, **Ankle Boot** class has the highest probablity

Image 3: looks like a **Sandal**, and the model makes the prediction of **Sandal** and hence, **Sandal** class has the highest probablity

Colab paid products - Cancel contracts here

✓ 0s completed at 14:58

Question 5

(3 points) Implementing back-propagation in Python from scratch. Open the (incomplete) Jupyter notebook provided as an attachment to this homework in Google Colab (or other Python IDE of your choice) and complete the missing items. In the second demo, we worked with autodiff. Autodiff enables us to implicitly store how to calculate the gradient when we call backward. We implemented some basic operations (addition, multiplication, power, and ReLU). In this homework problem, you will implement backprop for more complicated operations directly. Instead of using autodiff, you will manually compute the gradient of the loss function for each parameter.

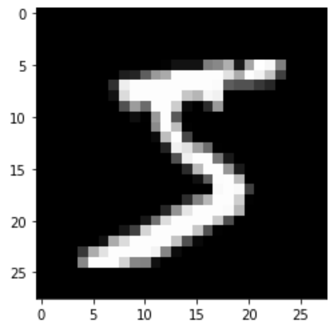
In this problem we will train a neural network from scratch using numpy. In practice, you will never need to do this (you'd just use TensorFlow or PyTorch). But hopefully this will give us a sense of what's happening under the hood.

For training/testing, we will use the standard MNIST benchmark consisting of images of handwritten images.

In the second demo, we worked with autodiff. Autodiff enables us to implicitly store how to calculate the gradient when we call backward. We implemented some basic operations (addition, multiplication, power, and ReLU). In this homework problem, you will implement backprop for more complicated operations directly. Instead of using autodiff, you will manually compute the gradient of the loss function for each parameter.

```
import tensorflow as tf
import matplotlib.pyplot as plt

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data(path="mnist.npz")

plt.imshow(x_train[0],cmap='gray');
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
```



Loading MNIST is the only place where we will use TensorFlow; the rest of the code will be pure numpy.

Let us now set up a few helper functions. We will use sigmoid activations for neurons, the softmax activation for the last layer, and the cross entropy loss.

```
import numpy as np

def sigmoid(x):
  # Numerically stable sigmoid function based on
  # http://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/

  x = np.clip(x, -500, 500) # We get an overflow warning without this

  return np.where(
    x >= 0,
    1 / (1 + np.exp(-x)),
    np.exp(x) / (1 + np.exp(x))
  )

def dsigmoid(x): # Derivative of sigmoid
  return sigmoid(x) * (1 - sigmoid(x))

def softmax(x):
  # Numerically stable softmax based on (same source as sigmoid)
  # http://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/
  b = x.max()
  y = np.exp(x - b)
  return y / y.sum()

def cross_entropy_loss(y, yHat):
  return -np.sum(y * np.log(yHat))

def integer_to_one_hot(x, max):
  # x: integer to convert to one hot encoding
  # max: the size of the one hot encoded array
  result = np.zeros(10)
  result[x] = 1
  return result
```

OK, we are now ready to build and train our model. The input is an image of size 28x28, and the output is one of 10 classes. So, first:

Q1. Initialize a 2-hidden layer neural network with 32 neurons in each hidden layer, i.e., your layer sizes should be:

784 -> 32 -> 32 -> 10

If the layer is $n_{in} \times n_{out}$ your layer weights should be initialized by sampling from a normal distribution with mean zero and variance 1/ $\max(n_{in}, n_{out})$.

```
import math

# Initialize weights of each layer with a normal distribution of mean 0 and
# standard deviation 1/sqrt(n), where n is the number of inputs.
# This means the weighted input will be a random variable itself with mean
# 0 and standard deviation close to 1 (if biases are initialized as 0, standard
# deviation will be exactly 1)

from numpy.random import default_rng

rng = default_rng(80085)

# Q1. Fill initialization code here.
# ...

# np.random.normal(mean, std_dev, size_tuple) as the params.
# We have 2 hidden layer so in total we have 3 weights and 3 biases to initialize.

#three numpy arrays, which represent the weights of the neural network model.
weights = [
 np.random.normal(0, 1/math.sqrt(784), (784, 32)),
 np.random.normal(0, 1/math.sqrt(32), (32, 32)),
 np.random.normal(0, 1/math.sqrt(32), (32, 10))
]
#Initializing bias arrays of zeros for the hidden layer, with the shape of the hidden layer used to determine the size of each ar
biases = [np.zeros((1,32)), np.zeros((1,32)), np.zeros((1,10))]
```

Next, we will set up the forward pass. We will implement this by looping over the layers and successively computing the activations of each layer.

Q2. Implement the forward pass for a single sample, and for the entire dataset.

Right now, your network weights should be random, so doing a forward pass with the data should not give you any meaningful information. Therefore, in the last line, when you calculate test accuracy, it should be somewhere around 1/10 (i.e., a random guess).

What is happening here

In the Forward Pass, we apply the neural network formula to compute the output of each layer.

We use the dot product of the input with weights, add bias, and apply an activation function to yield the output of each layer.

We use the sigmoid activation function for the hidden layers and the softmax activation function for the output layer.

The expected output is represented by y_v, and we use the cross-entropy loss function to evaluate the model's performance

```
def feed_forward_sample(sample, y):
  """ Forward pass through the neural network.
    Inputs:
      sample: 1D numpy array. The input sample (an MNIST digit).
      label: An integer from 0 to 9.

    Returns: the cross entropy loss, most likely class
  """
  # Q2. Fill code here.
  # ...
  #Reshapes the input sample into a 1D array with 784 elements.
  a = np.reshape(sample,(1,28*28))

  # Forward pass

  z1 = np.dot(a,weights[0]) + biases[0] # dot product
  a1 = sigmoid(z1) # applying sigmoid activation function to z1
  z2 = np.dot(a1,weights[1]) + biases[1]
  a2 =sigmoid(z2)
  z3= np.dot(a2,weights[2], biases[2])
  s = softmax(z3) # applying softmax to z3

  # Calculate loss
  pred_class = np.argmax(s)
  one_hot_guess = integer_to_one_hot(pred_class,10) #Converting the predicted class into a one-hot encoded array
  y_v = integer_to_one_hot(y,10)
  loss = cross_entropy_loss(y_v,s) #Calculating the cross-entropy loss between the true label and the predicted class probabiliti


  return loss, one_hot_guess


def feed_forward_dataset(x, y):
  losses = np.empty(x.shape[0])
  one_hot_guesses = np.empty((x.shape[0], 10))


  # ...
  # Q2. Fill code here to calculate losses, one_hot_guesses
  # ...
  for i in range(x.shape[0]):
    sample = np.reshape(x[i],(1,28*28))
    losses[i], one_hot_guesses[i] = feed_forward_sample(sample, y[i])

  y_one_hot = np.zeros((y.size, 10))
  y_one_hot[np.arange(y.size), y] = 1

  correct_guesses = np.sum(y_one_hot * one_hot_guesses)
  correct_guess_percent = format((correct_guesses / y.shape[0]) * 100, ".2f")

  print("\nAverage loss:", np.round(np.average(losses), decimals=2))
  print("Accuracy (# of correct guesses):", correct_guesses, "/", y.shape[0], "(", correct_guess_percent, "%)")

def feed_forward_training_data():
  print("Feeding forward all training data...")
  feed_forward_dataset(x_train, y_train)
  print("")

def feed_forward_test_data():
  print("Feeding forward all test data...")
  feed_forward_dataset(x_test, y_test)
  print("")

feed_forward_test_data()
```

```
Feeding forward all test data...

Average loss: 2.37
Accuracy (# of correct guesses): 1144.0 / 10000 ( 11.44 %)
```

OK, now we will implement the backward pass using backpropagation. We will keep it simple and just do training sample-by-sample (no minibatching, no randomness).

Q3: Compute the gradient of all the weights and biases by backpropagating derivatives all the way from the output to the first layer.

What is happening here?

We are training a neural network model using the backpropagation algorithm to classify images from the MNIST dataset.

It performs a forward pass to compute the output of the model for a given input and a backward pass to calculate the gradients of the weights and biases with respect to the loss.

These gradients are then used to update the weights and biases to minimize the loss.

The process is repeated for each sample in the dataset to improve the model's accuracy.

```
def train_one_sample(sample, y, learning_rate=0.003):
    a = np.reshape(sample,(1,28*28))

    # We will store each layer's activations to calculate gradient
    activations = []


    # Forward pass

    # Q3. This should be the same as what you did in feed_forward_sample above.
    # ...
    Z1 = np.dot(a,weights[0]) + biases[0]
    a1 = sigmoid(Z1)
    Z2 = np.dot(a1,weights[1]) + biases[1]
    a2 =sigmoid(Z2)
    Z3= np.dot(a2,weights[2], biases[2])
    s = softmax(Z3)
    y_v = integer_to_one_hot(y,10)

    # Backward pass

    # Q3. Implement backpropagation by backward-stepping gradients through each layer.
    # You may need to be careful to make sure your Jacobian matrices are the right shape.
    # At the end, you should get two vectors: weight_gradients and bias_gradients.
    # ...

    #calculating dW1, db1, dW2, db2, dW3, db3.
    #dZ, dW, db are the derivatives of the Cost function wrt Weighted sum, Weights, Bias of the layers
    #output layer
    dZ3 = s - y_v
    dW3 = np.dot(a2.T,dZ3)
    db3 = dZ3

    #second hidden layer
    dZ2 = np.multiply(dZ3.dot(weights[2].T),dsigmoid(Z2))
    dW2 = np.dot(a1.T,dZ2)
    db2 = dZ2

    #first hidden layer
    dZ1 = np.multiply(dZ2.dot(weights[1].T),dsigmoid(Z1))
    dW1 = np.dot(a.T,dZ1)
    db1 = dZ1

    # We will store the derivative weight for each layer. -> vector 1
    weight_gradients = []
    weight_gradients.append(dW1)
    weight_gradients.append(dW2)
    weight_gradients.append(dW3)

    # We will store the derivative bias for each layer. -> vector 2
    bias_gradients = []
    bias_gradients.append(db1)
    bias_gradients.append(db2)
    bias_gradients.append(db3)

    # Update weights & biases based on your calculated gradient
    num_layers = 3
    for i in range(num_layers):
        weights[i] = weights[i] - learning_rate * weight_gradients[i]
        biases[i] = biases[i] - learning_rate * bias_gradients[i]
```

Finally, train for 3 epochs by looping over the entire training dataset 3 times.

Q4. Train your model for 3 epochs.

```
def train_one_epoch(learning_rate=0.003):
  print("Training for one epoch over the training dataset...")

  # Q4. Write the training loop over the epoch here.
  # ...
  for i in range(x_train.shape[0]):
    train_one_sample(x_train[i], y_train[i], learning_rate)
  print("Finished training.\n")


feed_forward_test_data()

def test_and_train():
  train_one_epoch()
  feed_forward_test_data()

for i in range(3):
  test_and_train()
```

```
Feeding forward all test data...

Average loss: 2.37
Accuracy (# of correct guesses): 1144.0 / 10000 ( 11.44 %)

Training for one epoch over the training dataset...
Finished training.

Feeding forward all test data...

Average loss: 0.93
Accuracy (# of correct guesses): 7001.0 / 10000 ( 70.01 %)

Training for one epoch over the training dataset...
Finished training.

Feeding forward all test data...

Average loss: 0.96
Accuracy (# of correct guesses): 6539.0 / 10000 ( 65.39 %)

Training for one epoch over the training dataset...
Finished training.

Feeding forward all test data...

Average loss: 0.91
Accuracy (# of correct guesses): 6807.0 / 10000 ( 68.07 %)
```

That's it!

Your code is probably very time- and memory-inefficient; that's ok. There is a ton of optimization under the hood in professional deep learning frameworks which we won't get into.

If everything is working well, you should be able to raise the accuracy from ~10% to ~70% accuracy after 3 epochs.

Reference:

1. https://www.geeksforgeeks.org/deep-neural-net-with-forward-and-back-propagation-from-scratch-python/
2. https://github.com/kvgarimella/dl-demos

✓ 2m 11s completed at 15:29