ECE-GY 6953- Deep Learning

Assignment-1

Abirami Sivakumar- as16288

---

1) *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.*

Solution:

The l1-norm loss function for linear regression can be written using matrix/vector notation as:

$L(\boldsymbol{w})=||\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w}||1$

where:

- X is a *n x p* matrix of n training data points with p features
- w is a *p x 1* vector of model parameters
- y is a *n x 1* vector of the corresponding labels

The size of the matrices/vectors are as follows:

- X: *n x p*
- w: *p x 1*
- y: *n x 1*

Note that ||.||1 denotes the l1-norm, which is the sum of absolute values of the elements in the vector.

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

Solution:

No, we cannot write down the optimal linear model in closed form for the l1-norm loss function as we did for standard linear regression. The reason for this is that the l1-norm loss function is not differentiable at zero. This makes it difficult to find a closed-form solution for the optimal coefficients.

Instead, we can use optimization algorithms such as **gradient descent, proximal gradient descent, or coordinate descent** to find the optimal coefficients that minimize the l1-norm loss function. These algorithms iteratively update the coefficients until they converge to a minimum of the loss function.

One popular algorithm for solving l1-regularized linear regression (also known as **LASSO**) is the coordinate descent algorithm. This algorithm updates one coefficient at a time while holding all others fixed. The update rule involves solving a simple subproblem that can be done efficiently. The coordinate descent algorithm can converge to the optimal solution even if the number of features is much larger than the number of training examples.

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.*

Solution:

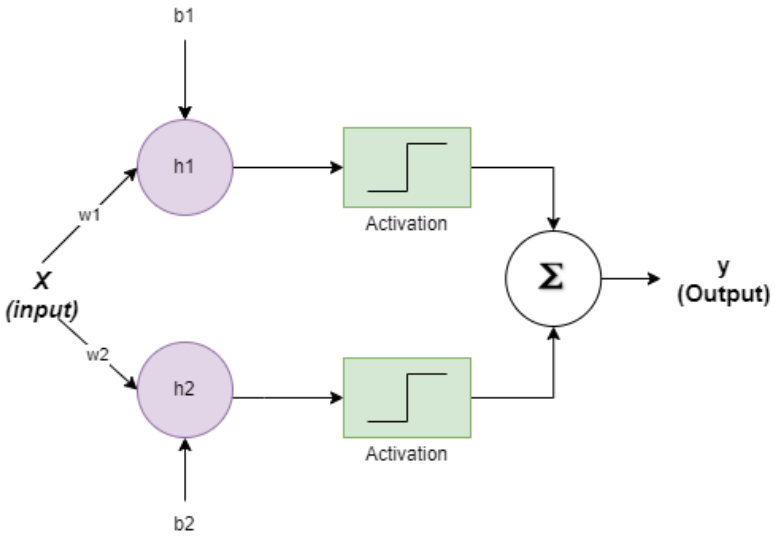
In this question, I used the knowledge of different loss functions and how to express them in matrix/vector notation to derive a loss function based on the 1-norm. I did not explicitly compute the optimal linear model in closed form, but the same approach can be applied to obtain an analytical solution. This problem can be related to the first step of the three-step recipe for ML, which involves formulating the problem and selecting an appropriate loss function that captures the desired behaviour of the model.

---

2) *Expressivity of neural networks. Recall that the functional form for a single neuron is given by y = σ(hw, xi+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 h and width δ is the function f(x) = h for 0 < x < δ and 0 otherwise. Show that a simple neural network with 2 hidden neurons with step activations 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 nonlinearity.)*

Solution:



For the box function with height **h** and width **δ**, we want the network to output **h** for inputs **x** such that **0 < x < δ**, and **0** otherwise. The step function has a value of **1** for positive inputs and **0** for non-positive inputs. Therefore, we need to find weights and biases that will make the hidden neuron output **1** for **0 < x < δ** and **0** for **negative** values of x.

The step function is given by :

$\sigma(z) = 1$ if $z > 0$

0 else

Let's split the interval **$0 < x < \delta$ to**

$0 < x < \delta/2$ and $\delta/2 < x < \delta$

- In Hidden Neuron 1**(h1)** :
  $Oh1 = \sigma(w_1x + b_1)$, **$0 < x < \delta/2$**
  $h/2 = w_1x + b_1$
  Subs boundary conditions

  $h/2 = \delta/2 \cdot w_1 + b_1$
  Let $w1 = -h/2\delta$
  $h/2 = \delta/2 \cdot (-h/2\delta) + b_1$
  $b_1 = 3h/4$

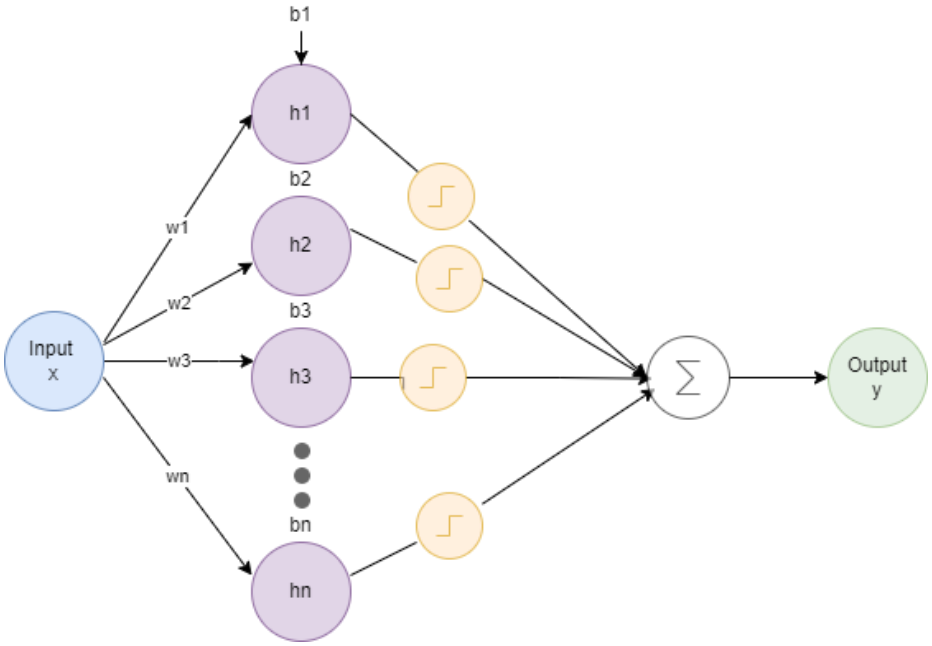
- In Hidden Neuron 2 **(h2)**:
  $Oh2 = \sigma(w_2x+b_2)$, **$\delta/2 < x < \delta$**
  $h/2 = w_2x + b_2$
  Subs boundary conditions
  $h/2 = w_2\,\delta + b_2$
  Let $w_2 = h/\delta$
  $h/2 = h/\delta.\,\delta + b_2$
  $b_2 = -h/2$

- Output Layer:
  $Y = \sum(Oh1, Oh2)$
  $= h/2 + h/2 =$ **h**

$W1 = -h/2\delta$
$W2 = h/\delta$
$b1 = 3h/4$
$b2 = -h/2$

b) *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.*

Solution:



The Universal Approximation Theorem states that a neural network with a single hidden layer, with a sufficient number of neurons, can approximate any continuous function on a compact subset of R^n, to any desired degree of accuracy. This means that we can use the network architecture we derived for the box function, with some modifications, to approximate any smooth and bounded function over the interval [-B, B].

In this architecture, the input x is connected to the first hidden neuron h1 via weight w1, and the nth hidden neuron hn has weight wn. Each hidden neuron has a bias of b_i where $i \in (1,n)$. All hidden neurons apply the step activation function. Their ouputs are then summed over to obtain the function approximation.

To approximate the function f, we can choose the weights and biases such that the output of each hidden neuron is 1 when the input is in the corresponding subinterval of width 2B, and 0 otherwise.

c) *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.*

Solution*:*

Yes, the argument can be extended to the case of d-dimensional inputs. One can simply use the same structure of the network and replace the scalar inputs with d-dimensional vectors. In fact, this is the basic idea behind many popular neural network architectures used for image and text processing, where the inputs are often high-dimensional vectors.

However, defining neural networks with a large number of neurons and layers for high-dimensional inputs can lead to some practical issues such as the **curse of dimensionality**. As the dimensionality of the input space increases, the number of neurons required to approximate a function can grow exponentially, which can make training and inference in such networks computationally infeasible. Additionally, high-dimensional inputs may also require large amounts of data to train such networks effectively, which can be another challenge. They can also be susceptible to overfitting. Various techniques such as dimensionality reduction, regularization, and efficient optimization methods need to be used to address some of these issues in practice.

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.*

***Solution**:*

When solving this question, I first identified the function we wanted to approximate (a box function) and then used my understanding of activation functions, weights, and biases to create a functional network that produces the desired output This process of identifying a problem and then designing a solution based on existing knowledge is a key aspect of the first step of the three-step recipe for machine learning, which is **problem formulation**. In this case, we already had a well-defined problem. We don't need to prepare or train data in this particular problem.

---

3) *Calculating gradients. Suppose that z is a vector with n elements. We would like to compute the gradient of y = softmax(z). Show that the Jacobian of y with respect to z, J, is given by the* $J_{ij} = \partial y_i \, \partial z_j = y_i(\delta_{ij} - y_j)$ *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,* $\partial \log y_i \, \partial z_j$.

***Solution**:*

The Softmax function is a *vector function*, which takes a vector as input and produces a vector as output:

$$\text{softmax: } \mathbb{R}^n \to \mathbb{R}^n$$

y=softmaz(z)

The Jacobian Matrix is given by :

$$\begin{pmatrix} \partial y_1/\partial z_1 & \partial y_1/\partial z_2 & ..\partial y_1/\partial z_n \\ \partial y_2/\partial z_1 & \partial y_2/\partial z_2 & ..\partial y_3/\partial z_n \\ \quad . & & \\ \quad . & & \\ \partial y_n/\partial z_1 & \partial y_n/\partial z_2 & ..\partial y_n/\partial z_n \end{pmatrix}$$

where

Each output of the softmax function depends on all the input values. For this reason, the off-diagonal elements of the Jacobian are not zero.

Since the outputs of the softmax function are strictly positive values, instead of taking the partial derivative of the output, we take the partial derivative of the log of the output like so :

$$\frac{\partial log(y_i)}{\partial z_j} = \frac{1}{y_i}.\frac{\partial y_i}{\partial z_j} log(y_i) \qquad (1)$$

where the expression on the right-hand side follows directly from the chain rule. Next, we rearrange and obtain:

$$\frac{\partial y_i}{\partial z_j} = y_i.\frac{\partial}{\partial z_j} log(y_i)$$

Taking the logarithm of y:

$$log(y_i) = log(\frac{e^{z_i}}{\sum_{l=1}^{n} e^{zl}}) = z_i - log(\sum_{l=1}^{n} e^{zl})$$

The partial derivative of the resulting expression is:

$$\frac{\partial}{\partial z_j} log(y_i) = \frac{\partial z_i}{\partial z_j} - \frac{\partial}{\partial z_j} log(\sum_{l=1}^{n} e^{zl})$$

Let's have a look at the first term on the right-hand side:

$$\frac{\partial z_i}{\partial z_j} = \begin{cases} 1, & if\ i = j \\ 0, & otherwise \end{cases}$$

which can be concisely written using the indicator function $1\{\cdot\}$. The indicator function takes on a value of 1 if its argument is true, and 0 otherwise.

The second term on the right-hand side can be evaluated by applying the chain rule:

$$\frac{\partial}{\partial z_j} log(y_i) = \{i = j\} - \frac{1}{\sum_{l=1}^{n} e^{zl}} \cdot \frac{\partial}{\partial z_j} log(\sum_{l=1}^{n} e^{zl})$$

In the step above we used the derivative of the natural logarithm:

$$\frac{d}{dx} log(x) = \frac{1}{x}$$

Obtaining the partial derivative of the sum :

$$\frac{\partial}{\partial z_j} \sum_{l=1}^{n} e^{z_l} = \frac{\partial}{\partial z_j} [e^{z_1} + e^{z_2} + \cdots + e^{z_j} + \cdots + e^{z_n}] = \frac{\partial}{\partial z_j} [e^{z_j}] = e^{z_j}$$

Plugging the result into the formula gives:

$$\frac{\partial}{\partial z_j} log(y_i) = \{i = j\} - \frac{1}{\sum_{l=1}^{n} e^{zl}} \qquad = 1\{i = j\} - y_j$$

Finally, we have to multiply the upper expression with *s* (as shown in eqn 1):

$$\frac{\partial y_i}{\partial z_j} = y_i \cdot \frac{\partial}{\partial z_j} log(y_i) = y_i \cdot (1\{i = j\} - y_j)$$

$$= y_i \cdot (\ \delta_{ij} - y_j)$$

Improving the FashionMNIST Classifier

Abirami Sivakumar

Net ID: as16288

```
# We start by importing the libraries

import numpy as np
import torch
import torchvision
import torch.backends.cudnn as cudnn
from tqdm import tqdm
```

Step 1: Data Preparation

We will train a dense neural network on a popular image dataset called *Fashion-MNIST*.

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

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-i
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-i
100%                                      26421880/26421880 [00:01<00:00, 25100463.68it/s]
Extracting ./FashionMNIST/FashionMNIST/raw/train-images-idx3-ubyte.gz to ./FashionMNI

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-i
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-i
100%                                      29515/29515 [00:00<00:00, 266730.91it/s]
Extracting ./FashionMNIST/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ./FashionMNI

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-id
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-id
100%                                      4422102/4422102 [00:00<00:00, 7972287.51it/s]
Extracting ./FashionMNIST/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ./FashionMNIS

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-id
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-id
100%                                      5148/5148 [00:00<00:00, 194103.59it/s]
Extracting ./FashionMNIST/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ./FashionMNIS
```

Let's check that everything has been downloaded.

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

```
60000
10000
```

Let's investigate to see what's inside the dataset.

```
image, label = trainingdata[0]
print(image.shape, label)
```

```
torch.Size([1, 28, 28]) 9
```

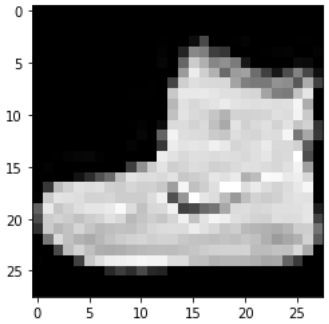
We cannot directly plot the image object given that its first dimension has a size of 1. So we will use the `squeeze` function to get rid of the first dimension.

```
print(image.squeeze().shape)
```

```
torch.Size([28, 28])
```

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.imshow(image.squeeze(), cmap=plt.cm.gray)
```

```
<matplotlib.image.AxesImage at 0x7f3e6434cb80>
```



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

Let's also check the length of the train and test dataloader

```
print(len(trainDataLoader))
print(len(testDataLoader))
```

```
938
157
```

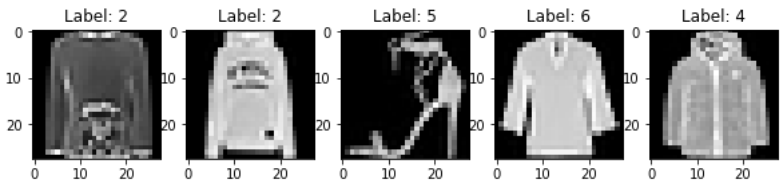
The length here depends upon the batch size defined above. Multiplying the length of our dataloader by the batch size should give us back the number of samples in each set.

```
print(len(trainDataLoader) * 64) # batch_size from above
print(len(testDataLoader) * 64)
```

```
60032
10048
```

Now let's use it to look at a few images.

```
images, labels = next(iter(trainDataLoader))

plt.figure(figsize=(10,4))
for index in np.arange(0,5):
  plt.subplot(1,5,index+1)
  plt.title(f'Label: {labels[index].item()}')
  plt.imshow(images[index].squeeze(),cmap=plt.cm.gray)
```



Step 2 : Defining and Training the Network

Let us setup our model. We need to use a (dense) neural network with three (3) hidden layers with 256, 128, and 64 neurons respectively, all with ReLU activations.

```
from torch import nn, optim
import torch.nn.functional as F


#TODO: Defining the Neural Network

class Classifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.i1 = nn.Linear(784, 256)
        self.h1 = nn.Linear(256, 128)
        self.h2 = nn.Linear(128, 64)
        self.h3 = nn.Linear(64, 10)

    def forward(self, x):
```

```
        # make sure input tensor is flattened
        x = x.view(x.shape[0], -1)

        x = F.relu(self.i1(x))
        x = F.relu(self.h1(x))
        x = F.relu(self.h2(x))

        return x
```

We choose an appropriate loss function and optimizer. Since we are dealing with a multi-class classification problem, we will use cross entropy loss. We use the Adam optimizer as it can minimize the loss function efficiently and has hyperparametrs that require little to no explicit tuning.

```
#Creating the model
model=Classifier()
loss_function=nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.003)
```

```
#Getting Model Summary
model.cuda()
from torchsummary import summary
summary(model,(1,28,28))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                 [-1, 256]         200,960
            Linear-2                 [-1, 128]          32,896
            Linear-3                  [-1, 64]           8,256
================================================================
Total params: 242,112
Trainable params: 242,112
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.92
Estimated Total Size (MB): 0.93
----------------------------------------------------------------
```

```
print(model)
device = 'cuda' if torch.cuda.is_available() else 'cpu'
model = model.to(device)
if device == 'cuda':
    model = torch.nn.DataParallel(model)
    cudnn.benchmark = True
```

```
Classifier(
  (i1): Linear(in_features=784, out_features=256, bias=True)
  (h1): Linear(in_features=256, out_features=128, bias=True)
  (h2): Linear(in_features=128, out_features=64, bias=True)
  (h3): Linear(in_features=64, out_features=10, bias=True)
)
```

```
epochs = 50
train_loss_history = []
test_loss_history = []
train_acc_history = []
test_accuracy = []
```

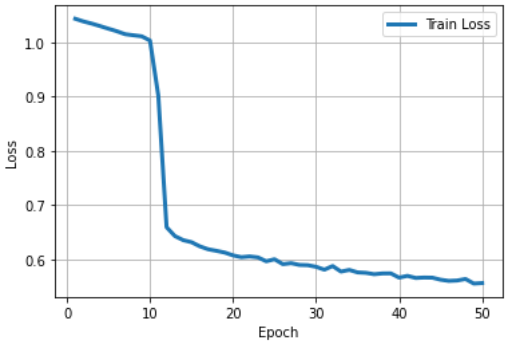
```
def train(epoch, train_loss_history, train_acc_history):
  model.train()
  train_loss = 0
  correct = 0
  total = 0
  accuracy = 0
  with tqdm(trainDataLoader, unit="batch") as tepoch:
    for batch_idx, (inputs, targets) in enumerate(tepoch):
      inputs, targets = inputs.to(device), targets.to(device)
      optimizer.zero_grad()
      outputs = model(inputs)
      loss = loss_function(outputs, targets)
      loss.backward()
      optimizer.step()

      train_loss += loss.item()
      _, predicted = outputs.max(1)
      total += targets.size(0)
      correct += predicted.eq(targets).sum().item()
      tepoch.set_postfix(Epoch = epoch, Loss = loss.item(), Accuracy = 100.*correct/total)
```

```
  train_loss = train_loss / len(trainDataLoader)
  train_loss_history += [train_loss]
  accuracy = 100.*correct/total
  train_acc_history += [accuracy]

def test(test_loss_history, test_accuracy):
  model.eval()
  test_loss = 0
  correct = 0
  total = 0
  accuracy = 0
  for batch_idx, (inputs, targets) in enumerate(testDataLoader):
    with torch.no_grad():
      inputs, targets = inputs.to(device), targets.to(device)
      outputs = model(inputs)
      loss = loss_function(outputs, targets)

      test_loss += loss.item()
      _, predicted = outputs.max(1)
      total += targets.size(0)
      correct += predicted.eq(targets).sum().item()
      test_loss_history += [test_loss / (batch_idx + 1)]
  accuracy = 100.*correct/total
  test_accuracy += [accuracy]



for epoch in range(1, epochs + 1):
  train(epoch, train_loss_history, train_acc_history)
test(test_loss_history, test_accuracy)
```

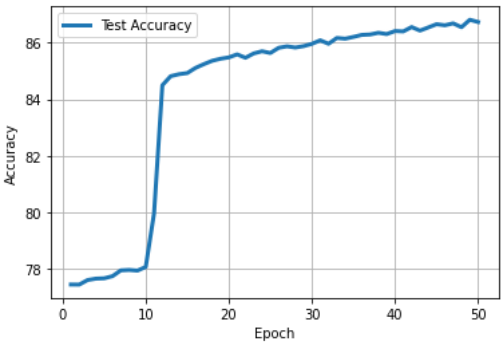
```
    100%|██████████| 938/938 [00:12<00:00, 75.83batch/s, Accuracy=77.5, Epoch=1, Loss=0.937]
    100%|██████████| 938/938 [00:11<00:00, 82.05batch/s, Accuracy=77.5, Epoch=2, Loss=0.645]
    100%|██████████| 938/938 [00:11<00:00, 82.25batch/s, Accuracy=77.6, Epoch=3, Loss=0.969]
    100%|██████████| 938/938 [00:11<00:00, 81.51batch/s, Accuracy=77.7, Epoch=4, Loss=0.649]
    100%|██████████| 938/938 [00:11<00:00, 82.25batch/s, Accuracy=77.7, Epoch=5, Loss=0.547]
    100%|██████████| 938/938 [00:12<00:00, 76.21batch/s, Accuracy=77.8, Epoch=6, Loss=1.68]
    100%|██████████| 938/938 [00:11<00:00, 81.63batch/s, Accuracy=78, Epoch=7, Loss=0.834]
    100%|██████████| 938/938 [00:11<00:00, 82.53batch/s, Accuracy=78, Epoch=8, Loss=1.01]
    100%|██████████| 938/938 [00:11<00:00, 83.59batch/s, Accuracy=78, Epoch=9, Loss=1.17]
    100%|██████████| 938/938 [00:11<00:00, 85.08batch/s, Accuracy=78.1, Epoch=10, Loss=0.919]
    100%|██████████| 938/938 [00:11<00:00, 78.77batch/s, Accuracy=80, Epoch=11, Loss=0.942]
    100%|██████████| 938/938 [00:11<00:00, 78.83batch/s, Accuracy=84.5, Epoch=12, Loss=0.655]
    100%|██████████| 938/938 [00:11<00:00, 82.65batch/s, Accuracy=84.8, Epoch=13, Loss=0.446]
    100%|██████████| 938/938 [00:11<00:00, 82.75batch/s, Accuracy=84.9, Epoch=14, Loss=0.671]
    100%|██████████| 938/938 [00:11<00:00, 82.20batch/s, Accuracy=84.9, Epoch=15, Loss=0.719]
    100%|██████████| 938/938 [00:11<00:00, 81.60batch/s, Accuracy=85.1, Epoch=16, Loss=0.359]
    100%|██████████| 938/938 [00:12<00:00, 76.11batch/s, Accuracy=85.2, Epoch=17, Loss=0.401]
    100%|██████████| 938/938 [00:11<00:00, 81.67batch/s, Accuracy=85.4, Epoch=18, Loss=0.723]
    100%|██████████| 938/938 [00:11<00:00, 80.84batch/s, Accuracy=85.4, Epoch=19, Loss=0.588]
    100%|██████████| 938/938 [00:11<00:00, 80.85batch/s, Accuracy=85.5, Epoch=20, Loss=0.449]
    100%|██████████| 938/938 [00:11<00:00, 81.42batch/s, Accuracy=85.6, Epoch=21, Loss=1.21]
    100%|██████████| 938/938 [00:12<00:00, 74.62batch/s, Accuracy=85.5, Epoch=22, Loss=0.556]
    100%|██████████| 938/938 [00:11<00:00, 83.36batch/s, Accuracy=85.6, Epoch=23, Loss=1.21]
    100%|██████████| 938/938 [00:11<00:00, 83.21batch/s, Accuracy=85.7, Epoch=24, Loss=0.61]
    100%|██████████| 938/938 [00:11<00:00, 81.16batch/s, Accuracy=85.6, Epoch=25, Loss=0.44]
    100%|██████████| 938/938 [00:11<00:00, 82.41batch/s, Accuracy=85.8, Epoch=26, Loss=0.687]
    100%|██████████| 938/938 [00:12<00:00, 75.46batch/s, Accuracy=85.9, Epoch=27, Loss=0.688]
    100%|██████████| 938/938 [00:11<00:00, 81.63batch/s, Accuracy=85.8, Epoch=28, Loss=0.275]
    100%|██████████| 938/938 [00:11<00:00, 82.47batch/s, Accuracy=85.9, Epoch=29, Loss=1.11]
    100%|██████████| 938/938 [00:11<00:00, 81.15batch/s, Accuracy=86, Epoch=30, Loss=1.47]
    100%|██████████| 938/938 [00:11<00:00, 82.69batch/s, Accuracy=86.1, Epoch=31, Loss=0.65]
    100%|██████████| 938/938 [00:12<00:00, 76.72batch/s, Accuracy=86, Epoch=32, Loss=1.29]
    100%|██████████| 938/938 [00:11<00:00, 82.13batch/s, Accuracy=86.2, Epoch=33, Loss=0.503]
    100%|██████████| 938/938 [00:11<00:00, 82.35batch/s, Accuracy=86.1, Epoch=34, Loss=0.486]
    100%|██████████| 938/938 [00:11<00:00, 84.16batch/s, Accuracy=86.2, Epoch=35, Loss=0.302]
    100%|██████████| 938/938 [00:11<00:00, 83.99batch/s, Accuracy=86.3, Epoch=36, Loss=1.06]
    100%|██████████| 938/938 [00:11<00:00, 78.65batch/s, Accuracy=86.3, Epoch=37, Loss=0.718]
    100%|██████████| 938/938 [00:11<00:00, 79.39batch/s, Accuracy=86.3, Epoch=38, Loss=0.14]
    100%|██████████| 938/938 [00:11<00:00, 80.30batch/s, Accuracy=86.3, Epoch=39, Loss=1.29]
    100%|██████████| 938/938 [00:11<00:00, 81.41batch/s, Accuracy=86.4, Epoch=40, Loss=0.285]
    100%|██████████| 938/938 [00:11<00:00, 81.28batch/s, Accuracy=86.4, Epoch=41, Loss=0.599]
    100%|██████████| 938/938 [00:12<00:00, 77.76batch/s, Accuracy=86.5, Epoch=42, Loss=0.555]
    100%|██████████| 938/938 [00:12<00:00, 77.79batch/s, Accuracy=86.4, Epoch=43, Loss=0.624]
    100%|██████████| 938/938 [00:11<00:00, 82.26batch/s, Accuracy=86.5, Epoch=44, Loss=0.644]
    100%|██████████| 938/938 [00:11<00:00, 81.38batch/s, Accuracy=86.6, Epoch=45, Loss=1.49]
    100%|██████████| 938/938 [00:11<00:00, 80.39batch/s, Accuracy=86.6, Epoch=46, Loss=0.585]
    100%|██████████| 938/938 [00:11<00:00, 79.93batch/s, Accuracy=86.7, Epoch=47, Loss=0.148]
    100%|██████████| 938/938 [00:12<00:00, 77.35batch/s, Accuracy=86.5, Epoch=48, Loss=0.752]
    100%|██████████| 938/938 [00:11<00:00, 83.26batch/s, Accuracy=86.8, Epoch=49, Loss=0.765]
    100%|██████████| 938/938 [00:11<00:00, 81.29batch/s, Accuracy=86.7, Epoch=50, Loss=0.593]
```

Step 3 : Model Evaluation

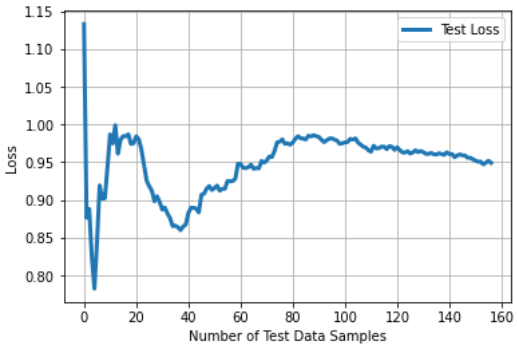
```
plt.plot(range(1, epochs + 1),train_loss_history,'-',linewidth=3,label='Train Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.grid(True)
plt.legend()
plt.show()
```




```
plt.plot(range(1, epochs + 1),train_acc_history,'-',linewidth=3,label='Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.grid(True)
plt.legend()
plt.show()
```




```
plt.plot(range(len(testDataLoader)),test_loss_history,'-',linewidth=3,label='Test Loss')
plt.xlabel('Number of Test Data Samples')
plt.ylabel('Loss')
plt.grid(True)
plt.legend()
plt.show()
```



The Test Accuracy is :

```
print("Test Accuracy:", test_accuracy[0])
```

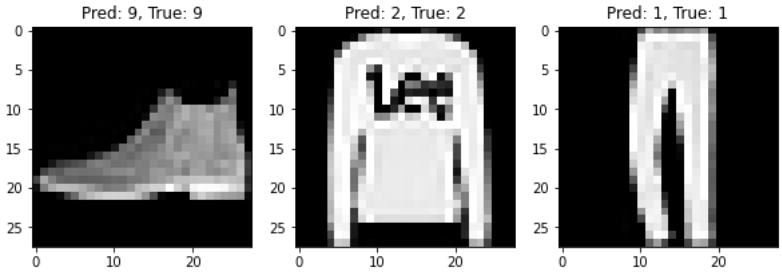
```
Test Accuracy: 81.99
```

Now for the labels and predicted labels.

Displaying 3 images with predicted and True labels

```
images, labels = next(iter(testDataLoader))
```

```
plt.figure(figsize=(10,4))
for index in np.arange(0,3):
  plt.subplot(1,3,index+1)
  plt.title(f'Pred: {torch.max(model(images[index]), 1)[1].item()}, True: {labels[index].item()}')
  plt.imshow(images[index].squeeze(),cmap=plt.cm.gray)
```



We obtain the test accuracy to be 81.99% i.e the model is able to correctly predict the class labels for 81.99% of the test set. The accuracy obtained is reasonable and does not entail further training as the test loss begins to flatten out. Training further in hopes of improving accuracy could lead to overfitting. As seen in the plots above, the neural network is able to predict correct labels for an input image.

Implementing Back-propagation in Python from Scratch

Abirami Sivakumar

Net ID: as16288

Collaborators : Nagharjun Mathi Mariappan (Net ID: nm4074)

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/mni
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
```

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
import numpy as np
# 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.
# ...


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))
]
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).

```
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
  """

  a=np.reshape(sample,(1,28*28))

  #Forward Pass
  u1=np.dot(a,weights[0])+biases[0]
  z1=sigmoid(u1)
  u2=np.dot(z1,weights[1])+biases[1]
  z2=sigmoid(u2)
  u3=np.dot(z1,weights[2]+biases[2])
  yhat=softmax(u3)


#Loss calculation
  pred_class=np.argmax(yhat)
  one_hot_guess=integer_to_one_hot(pred_class,10)
  yout=integer_to_one_hot(y,10)
  loss=cross_entropy_loss(yout,yhat)

  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: 2.47
    Accuracy (# of correct guesses): 1174.0 / 10000 ( 11.74 %)
```

The code defines two functions: feed_forward_training_data() and feed_forward_test_data() that call the feed_forward_dataset() function to feed forward the training and test datasets through the neural network. After calling feed_forward_dataset(), it prints the average loss and accuracy of the predictions.

```
x_test.shape[0]
```

```
    10000
```

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

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


  weight_gradients = []
  bias_gradients = []

  # Forward pass
  u1=np.dot(a,weights[0])+biases[0]
  z1=sigmoid(u1)
  u2=np.dot(z1,weights[1])+biases[1]
  z2=sigmoid(u2)
  u3=np.dot(z1,weights[2]+biases[2])
  yhat=softmax(u3)


#Loss calculation
  pred_class=np.argmax(yhat)
  one_hot_guess=integer_to_one_hot(pred_class,10)
  yout=integer_to_one_hot(y,10)
  loss=cross_entropy_loss(yout,yhat)


  # Backward pass
  du3 = yhat - yout
  dw3 = z1.T.dot(du3)
  db3 = du3

  #dz2 = du3.dot(weights[2].T)
  du2 = np.dot(du3,weights[2].T)*dsigmoid(u2)
```

```
  dw2 = z1.T.dot(du2)
  db2 = du2

  du1 = np.dot(du2,weights[1].T)*dsigmoid(u1)
  dw1 = np.dot(a.T,du1)
  db1 = du1

  weight_gradients.append(dw1)
  weight_gradients.append(dw2)
  weight_gradients.append(dw3)
  bias_gradients.append(db1)
  bias_gradients.append(db2)
  bias_gradients.append(db3)

  # update weights and biases based on 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]

  return loss
```

The input sample and corresponding ground truth label y are used to calculate the forward pass through the network. The function then calculates the loss using cross-entropy loss function and performs backpropagation to compute the gradients of the weights and biases. The updated weights and biases are then returned to the caller of this function.

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: 10.79
Accuracy (# of correct guesses): 851.0 / 10000 ( 8.51 %)

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

Feeding forward all test data...

Average loss: 0.92
Accuracy (# of correct guesses): 6835.0 / 10000 ( 68.35 %)

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

Feeding forward all test data...

Average loss: 0.86
Accuracy (# of correct guesses): 6855.0 / 10000 ( 68.55 %)

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

Feeding forward all test data...

Average loss: 0.97
Accuracy (# of correct guesses): 6673.0 / 10000 ( 66.73 %)
```

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.