MNIST Digit Classifier

LIBRARIES USED AND THEIR FUNCTIONS:-

- Torch.nn: Has all the modules required for building an unordered graph. In this case a Neural Network. A list of all given modules:
  - Containers
  - Convolution Layers
  - Pooling layers
  - Padding Layers
  - Non-linear Activations (weighted sum, nonlinearity)
  - Non-linear Activations (other)
  - Normalization Layers
  - Recurrent Layers
  - Transformer Layers
  - Linear Layers
  - Dropout Layers
  - Sparse Layers
  - Distance Functions
  - Loss Functions
  - Vision Layers
  - Shuffle Layers
  - DataParallel Layers (multi-GPU, distributed)
  - Utilities
  - Quantized Functions
  - Lazy Modules Initialization
    - Aliases
- Torch.nn.functional: Has all the functions required to build the network. A list of all the functions is given here:-https://pytorch.org/docs/stable/nn.functional.html
- Torch.Optim: has all the optimisation functions for the network. For this project, we will be using the adam optimiser. More about it later. https://pytorch.org/docs/stable/optim.html
- Torchvision: Dataset library. You will find all default learning datasets here. We will be using the MNIST digit dataset which has 60000 digits.
- Numpy & Matplotlib for data visualization.

DATA MANIPULATION:-

Division of our entire dataset into a 6:1 train:test datasets. The first part of the dataset will train the model and the second part will evaluate the performance of the model based on our training result. Lastly, we will use the test dataset to sample from our network. The basic difference between train and test data is that the training dataset has something called labels, which is like the answer key for our network's reference.

This method of learning is called supervised learning. Thankfully, MNIST was kind enough to provide us with labels for this one.

```
from torchvision import datasets
from torchvision.transforms import ToTensor
from torch.utils.data import DataLoader
transform = ToTensor()
train_data = datasets.MNIST(root='data',train=True,transform=transform,
download = True)
test_data = datasets.MNIST(root='data',train=False,transform=transform,
download = True)
loaders =
{'train':DataLoader(train_data,batch_size=100,shuffle=True,num_workers=1),

'test':DataLoader(test_data,batch_size=100,shuffle=True,num_workers=1)}
```

After importing all the important libraries, we name a variable called transform=ToTensor(). The function ToTensor converts the image from the MNIST database to a tensor. You can think of it as luminosity values of different layers of the images converted to values between 0-255. Train_data and Test_data just divide the dataset into 2 parts used for training and testing. Transform will make it into a tensor from an image.
Now, make a dictionary out of the dataset named loaders. This dictionary will act as a link between the labels and the data tensor. Divide it into batch sizes of 100 so we don't get a runtime error :(and shuffle them during every epoch so that their order does not interfere with the training of the network. Num_workers is basically telling how many channels the data is being fed from. As this is a small dataset, one channel should suffice.

MODEL ARCHITECTURE:-

```
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
class CNN(nn.Module):
  def __init__(self):
    super(CNN,self,).__init__()

    self.conv1 = nn.Conv2d(1,10,kernel_size=5)
    self.conv2 = nn.Conv2d(10,20,kernel_size=5)
    self.drop = nn.Dropout2d()
    self.fc1 = nn.Linear(320,50)
    self.fc2 = nn.Linear(50,10)
```

```
  def forward(self,x):
    x = F.relu(F.max_pool2d(self.conv1(x),2))
    x = F.relu(F.max_pool2d(self.drop(self.conv2(x),),2))
    x = x.view(-1,320)
    x=F.relu(self.fc1(x))
    x=F.dropout(x,training=self.training)
    x = self.fc2(x)
    return x
model = CNN().to(device)
optimizer = optim.Adam(model.parameters(),lr=0.001)
loss_fn = nn.CrossEntropyLoss()
```

Specifying the Device: Okay so here is the deal, Neural networks need to learn and for learning you can either choose to train them on your CPU or if you have a dedicated graphics processor, that's always better. Pytorch actually lets you control which device you wanna train your model on. The function torch.device() lets you specify the device your network will run on. The function is_available checks whether the system you are using has cuda architecture or not. To know more and understand CUDA architecture you can refer here
https://www.geeksforgeeks.org/introduction-to-cuda-programming/

Now time to define our model!!!
So we will make a 5 layer model
Layer1 and layer2 will be convolution layers
Layer3 is a dropout layer
Layer 4 and layer5 are your linear layers

To understand what convolution layers are, we need to know convolution.
Now convolution is nothing but a mathematical operator.

$$(f*g)(t) \triangleq \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\tau'$$

This comes from probabilities and statistics, but to quickly run it down. It's just combining two functions(matrix in this case with another matrix) This combination can be done with different matrices sampling from different values of distribution. For example the Gaussian Blur is actually just a convolution of a matrix sampling from the gaussian distribution and an image. When 2 square waves are combined, that is also a convolution.
Why do we need this in our network????
Because convolution forms the basics of image processing and convolution. It gives our Neurons the power to manipulate the image in way more ways than a linear layer can. A linear layer does Y=Ax+B. But a convolution image does matrix multiplication. Another way of understanding convolution is if you plot the area enclosed between 2 approaching square waves wrt time.

Convolution Kernel is the matrix you are convolving with, if the kernel is smaller it can figure out more intricate details about the model.

Dropout Layer: The Dropout layer is a mask that nullifies the contribution of some neurons towards the next layer and leaves unmodified all others. We can apply a Dropout layer to the input vector, in which case it nullifies some of its features; but we can also apply it to a hidden layer, in which case it nullifies some hidden neurons.
Dropout layers are important in training CNNs because they prevent overfitting on the training data. If they aren't present, the first batch of training samples influences the learning in a disproportionately high manner. This, in turn, would prevent the learning of features that appear only in later samples or batches. The outputs are scaled by a factor of 1/p-1 where p is the probability of an element to be Zeroed. We are using default p=0.5 that means the chance of each neuron to break off is ½.

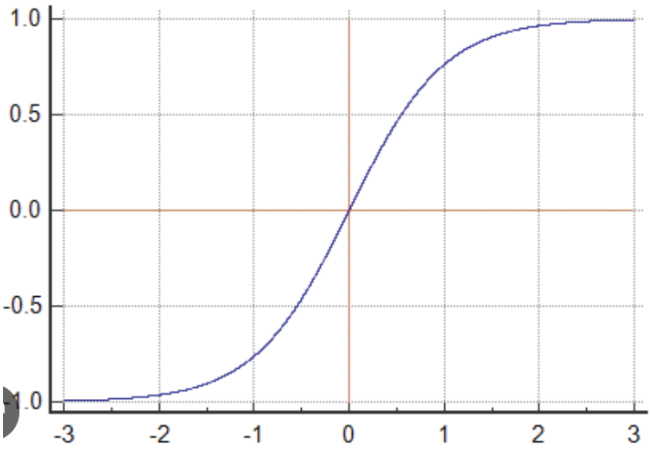
Linear/Dense Layer: It's a fully connected layer which means every neuron is connected to every other neuron. This is where classification starts. It is a simple linear transformation from one domain to another. What domains you may ask?----A transformation from a 28X28 matrix to a domain of a list of numbers from 0-9.

How does it do that???

Simple linear transformation Y=aX+b. a is weight and b is a bias. Y is the output domain of the network(0-9) and x is the input domain(28X28 tensor). In this case though, a and b are also tensors of we don't know and we don't care dimensions, although what we care about is that they will manipulate the final aX+b result to get a list of numbers from 0-9 and a probability assigned with each number. Finally the network chooses the highest probability. But how does it do that? Trial and error, if you give 60000 chances to a toddler for guessing a number between 1-10 and tell them after each chance if they are right or wrong, they will probably master that job. That's because our brain uses the same linear transformations to make conclusions, yes the weights and biases are messed up but we can figure that out slowly.

OKAY LET'S BEGIN THE FORWARD LOOP NOW:

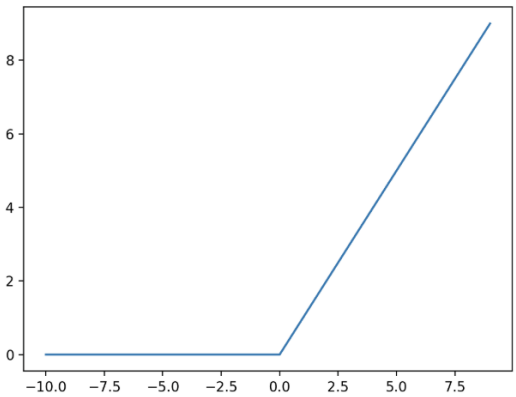
First order of business→ Activation function. Basically a random function attached to the end of every neuron. Why do we need one? Lets answer that question slowly, What does tanh look like?



Okay, so it takes inputs in the domain of R and gives an output in the range of (-1,1). The closer the function is to 0, the probability of it being the answer also decreases, but the further it goes away from 0, The probability increases if we assume there is a mod over there to account for the negative values. So what makes this function so special? First of all it introduces non-linearity. Until now all the equations we solved, every single calculation done by our model were linear. A simple Y=Ax+b. But a problem with linear function is everything is so normalised and ratioed. We must decrease the p value of answers that we dont need and increase the p values if answers that we need. That is where non-linearity plays a role.

Gradients: activation functions brings differentiability into our functions. Lets say you have a constant function and you multiply it with a tanh, suddenly now the function is differentiable. It might seem very obvious but i want you to take a moment and appreciate how useful this is. Because in the end our learning algorithm will be gradient descent. So we cant just get a 0 everywhere(derivative of a constant is 0). We need functions which are highly differentiable multiple times. Anyways, now that Im done yapping, lets just get into RELU function which is our activation function for this network.

RELU(rectified linear Unit):



After all the yapping about gradients and non linearity, we finally choose a function which is half linear and the rest half is basically 0 😐 lmao. Okay but let me tell you, this is very useful. First of all, every negative number GONE, so atleast we wont get negative probabilities. After we have that solved. It overcomes this vanishing gradient problem. The vanishing gradient problem is a challenge that occurs when training neural networks using backpropagation. It happens when the gradients used to update the network become very small, or "vanish". This can cause the network to learn very slowly or not at all.

So RELU avoids gradient saturation in-turn fixing the issue. And it also brings out the required probabilities we need.

LOSS CALCULATOR: Basically calculates the error between the required probabilities and the actual probabilities. Example for this is basically just standard deviation.

Cross Entropy Loss:-

In machine learning for classification tasks, the model predicts the probability of a sample belonging to a particular class. Since each sample can belong to only a particular class, the true probability value would be 1 for that particular class and 0 for the other class(es). Cross entropy measures the difference between the predicted probability and the true probability.

The Cross-Entropy Loss is derived from the principles of maximum likelihood estimation when applied to the task of classification. Maximizing the likelihood is equivalent to minimizing the negative log-likelihood. In classification, the likelihood function can be expressed as the product of the probabilities of the correct classes:

Binary Cross-Entropy Loss and Multiclass Cross-Entropy Loss are two variants of cross-entropy loss, each tailored to different types of classification tasks. Let us see them in detail.

$$-\frac{1}{N}\Sigma_{i=1}^{N}\Sigma_{j=1}^{C}(y_{i,j}.log(p_{i,j}))$$

- where
  - C is the number of classes.
  - $y_{i,j}$ are the true labels for class j for instance i
  - $p_{i,j}$ is the predicted probability for class j for instance i

This is also known as the negative log likelihood of the network.

Optimizer function:- ADAM(Adaptive moment estimation algorithm)

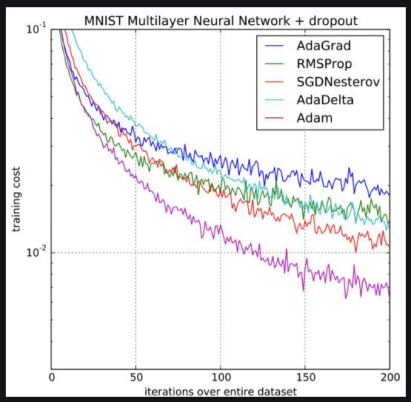
$$w_{t+1} = w_t - \alpha m_t$$

where,

$$m_t = \beta m_{t-1} + (1-\beta)\left[\frac{\delta L}{\delta w_t}\right]$$

m_t = aggregate of gradients at time t [current] (initially, $m_t = 0$)
m_{t-1} = aggregate of gradients at time t-1 [previous]
W_t = weights at time t
W_{t+1} = weights at time t+1
α_t = learning rate at time t
∂L = derivative of Loss Function
∂W_t = derivative of weights at time t
β = Moving average parameter (const, 0.9)

Building upon the strengths of previous models, Adam optimizer gives much higher performance than the previously used and outperforms them by a big margin into giving an optimized gradient descent. The plot is shown below clearly depicts how Adam Optimizer outperforms the rest of the optimizer by a considerable margin in terms of training cost (low) and performance (high).



$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\left[\frac{\delta L}{\delta w_t}\right]$$

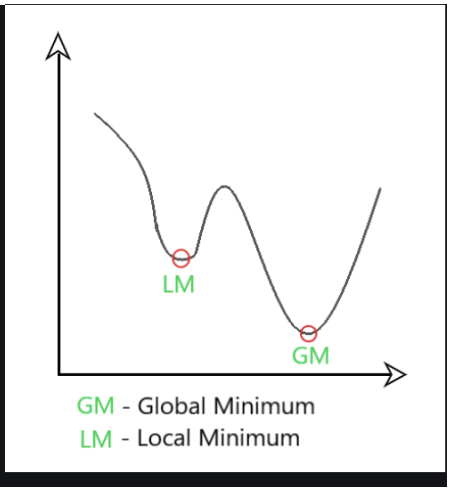
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\left[\frac{\delta L}{\delta w_t}\right]^2$$

Parameters Used :

1. ϵ = a small +ve constant to avoid 'division by 0' error when (v_t -> 0). (10^{-8})
2. β_1 & β_2 = decay rates of average of gradients in the above two methods. (β_1 = 0.9 & β_2 = 0.999)
3. α – Step size parameter / learning rate (0.001)

Here, we control the rate of gradient descent in such a way that there is minimum oscillation when it reaches the global minimum while taking big enough steps (step-size) so as to pass the local minima hurdles along the way. Hence, combining the features of the above methods to reach the global minimum efficiently.

This algorithm is used to accelerate the gradient descent algorithm by taking into consideration the 'exponentially weighted average' of the gradients. Using averages makes the algorithm converge towards the minima in a faster pace.



TRAINING AND TESTING LOOP:-

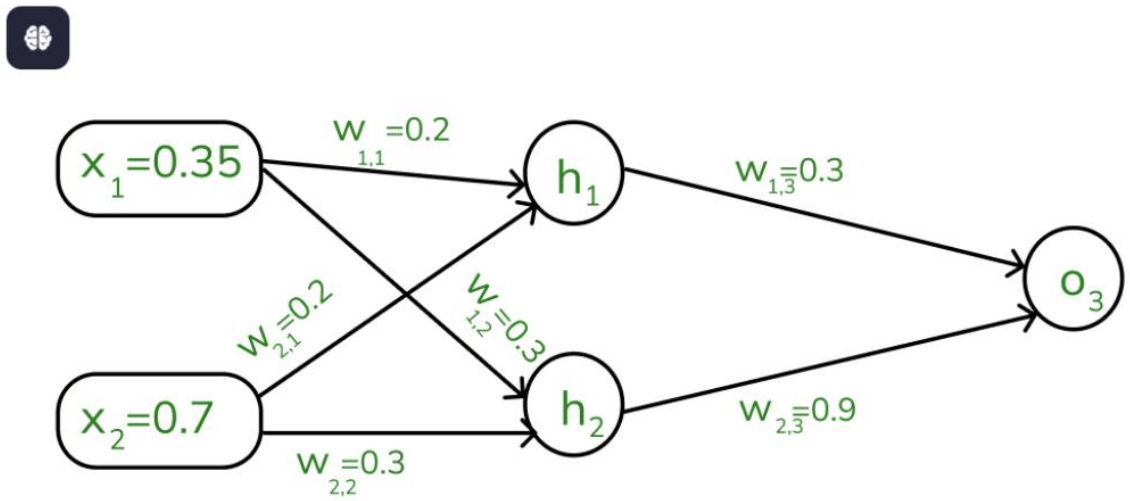
```
def train(epoch):

  model.train()

  for batch_idx,(data,target) in enumerate(loaders['train']):

    data,target = data.to(device),target.to(device)
```

```
    optimizer.zero_grad()

    output = model(data)

    loss = loss_fn(output,target)

    loss.backward()

    optimizer.step()

 if batch_idx%50==0:

      print(f'Train Epoch:
{epoch}[{batch_idx*len(data)}/{len(loaders["train"].dataset)}({10000. *
batch_idx / len(loaders["train"]):.0f}% )]\t{loss.item():.6f}')

def test():

  model.eval()

  test_loss = 0

  correct = 0

  with torch.no_grad():

    for data, target in loaders['test']:

      data,target = data.to(device),target.to(device)

      output = model(data)

      test_loss += loss_fn(output,target).item()

      pred = output.argmax(dim=1,keepdim=True)

      correct += pred.eq(target.view_as(pred)).sum().item()

  test_loss /= len(loaders['test'].dataset)

  print(f'\nTest set: Average loss: {test_loss:.4f},Accuracy:
{correct}/{len(loaders["test"])}')

for epoch in range(1,16):

  train(epoch)

   test()
```

.Zero_Grad():Resets all the gradients back to 0. To begin training.

Loss.Backward(): Does backprop.

Okay, so what is backprop and how to do backprop(honestly I can take an entire 3hr class on this 🤦). But for now



Backpropagation is a powerful algorithm in deep learning, primarily used to train artificial neural networks, particularly feed-forward networks. It works iteratively, minimizing the cost function by adjusting weights and biases.

In each epoch, the model adapts these parameters, reducing loss by following the error gradient. Backpropagation often utilizes optimization algorithms like gradient descent or stochastic gradient descent. The algorithm computes the gradient using the chain rule from calculus, allowing it to effectively navigate complex layers in the neural network to minimize the cost function.

Why is Backpropagation Important?

Backpropagation plays a critical role in how neural networks improve over time. Here's why:

Efficient Weight Update: It computes the gradient of the loss function with respect to each weight using the chain rule, making it possible to update weights efficiently.

Scalability: The backpropagation algorithm scales well to networks with multiple layers and complex architectures, making deep learning feasible.

Automated Learning: With backpropagation, the learning process becomes automated, and the model can adjust itself to optimize its performance.

The Backpropagation algorithm involves two main steps: the Forward Pass and the Backward Pass.

In the forward pass, the input data is fed into the input layer. These inputs, combined with their respective weights, are passed to hidden layers.

For example, in a network with two hidden layers (h1 and h2 as shown in Fig. (a)), the output from h1 serves as the input to h2. Before applying an activation function, a bias is added to the weighted inputs.

Each hidden layer applies an activation function like ReLU (Rectified Linear Unit), which returns the input if it's positive and zero otherwise. This adds non-linearity, allowing the model to learn complex relationships in the data. Finally, the outputs from the last hidden layer are passed to the output layer, where an activation function, such as softmax, converts the weighted outputs into probabilities for classification.

Once the error is calculated, the network adjusts weights using gradients, which are computed with the chain rule. These gradients indicate how much each weight and bias should be adjusted to minimize the error in the next iteration. The backward pass continues layer by layer, ensuring that the network learns and improves its performance. The activation function, through its derivative, plays a crucial role in computing these gradients during backpropagation.

SAMPLING FROM THE MODEL:-

```
import matplotlib.pyplot as plt

%matplotlib inline

model.eval()

data,target = test_data[80]

data = data.unsqueeze(0).to(device)

output = model(data)

prediction = output.argmax(dim=1,keepdim=True).item()

print(f'Prediction:{prediction}')

image = data.squeeze(0).squeeze(0).cpu().numpy()

plt.imshow(image,cmap='gray')

plt.show
```

ENJOY SAMPLING FROM YOUR FIRST AI MODEL