MLP-MNIST Dataset

Pytorch has a very convenient way to load the MNIST data using datasets. MNIST instead of data structures such as NumPy arrays and lists. Deep learning models use a very similar DS called a Tensor. When compared to arrays tensors are more computationally efficient and can run on GPUs too. I will convert our MNIST images into tensors when loading them. There are lots of other transformations that you can do using torchvision.transforms() like Reshaping, normalizing, etc. on your images but we won't need that since MNIST is a very primitive dataset.

Visualization of a Batch of Training Data

The train data has 60,000 images and the test has 10,000. Each image is made up of 28X28 pixels. The 1 in torch.size() stands for the number of channels, since it's a grayscale image there's only one channel.

Multilayer Perceptron

- A multilayer perceptron has several Dense layers of neurons in it, hence the name multi-layer.

- These artificial neurons/perceptrons are the fundamental unit in a neural network, quite analogous to the biological neurons in the human brain. The computation happening in a single neuron can be denoted by the equation. **N = Wx + b**, where x denotes the input to that neuron and W,b stands for weight and bias respectively. These two values are set at random initially and then keep on updating as the network learns.
- Each neuron in a layer is connected to every other neuron in its next layer. In MLPs, data only flows forwards hence they are also sometimes called Feed-Forward Networks.

There are 3 basic components:

1. **Input Layer**- The input layer would take in the input signal to be processed. In our case, it's a tensor of image pixels.

2. **Output Layer**- The output layer does the required task of classification/regression. In our case, it outputs one of the 10 classes for digits 0-9 for a given input image.

3. **Hidden Layers** - There is an arbitrary number of hidden layers in between the input and output layer that do all the computations in a Multilayer Perceptron. The number of hidden layers and the number of neurons can be decided to keep in mind the fact that one layer's output is the next layer's input.

process into different steps:

1. The pixels in the 28X28 handwritten digit image are flattened to form an array of 784-pixel values. Nothing heavy going on here, just decompressing a 2D array into one dimension

2. The function of the input layer is just to pass-on the input (array of 784 pixels) into the first hidden layer

3. flattened to form an array of 784-pixel values. Nothing heavy going on here, just decompressing a 2D array into one dimension.

4. The function of the input layer is just to pass-on the input (array of 784 pixels) into the first hidden layer.

5. The first hidden layer is where the computations start. It has 120 neurons that are each fed the input array. After calculating the result from the formula stated above, each neuron generates an output that is fed into each neuron of the next layer. Except, there is a little twist here. Instead of just simply passing on the result of **Wx+b**, an activation is calculated on this result.

The activation function is used to clip the output in a definite range like 0-1 or -1 to 1, these ranges can be achieved by *Sigmoid* and *Tanh* respectively. The activation function I have used here is ReLu. The main advantage of using the ReLu function is that it does not activate all the neurons at the same time thus making it more computationally efficient than Tanh or Sigmoid.

ReLu clips all the negative values and keeps the positive values just the same.

4. The same thing happens in the second hidden layer. It has 84 neurons and takes 120 inputs from the previous layer. The output of this layer is fed into the last layer which is the Output Layer.

5. The Output Layer has only 10 neurons for the 10 classes that I have(digits between 0-9). There isn't any activation function in the output layer because I'll apply another function later.

6. The Softmax takes the output of the last layer(called logits) which could be any 10 real values and converts it into another 10 real values that sum to 1. Softmax transforms the values between 0 and 1, such that they can be interpreted as probabilities. The maximum value pertains to the class predicted by the classifier. In our case, the value is 0.17 and the class is 5.

Defining Model:

- The code is straightforward. In Pytorch there isn't any implementation for the input layer, the input is passed directly into the first hidden layer. However, you'll find the InputLayer in the Keras implementation.

- The number of neurons in the hidden layers and the number of hidden layers is a parameter that can be played with, to get a better result.

the Loss Function and the Optimizer

There are a lot of loss functions out there like Binary Cross Entropy, Mean Squared Error, Hinged loss etc. The choice of the loss function depends on the problem at hand and the number of classes. Since we are dealing with a Multi-class classification problem, Pytorch's CrossEntropyLoss is our go-to loss function.

During Backpropagation, we update the weights according to the loss throughout the iterations. We basically try to minimize loss as we move ahead through our training. This process is called optimization. Optimizers are algorithms that try to find the optimal way to minimize the loss by navigating the surface of our loss function. We use Adam because it's the best optimizer out there, as proven by different experiments in the scientific community.

Training/Testing

- **Epoch** - An epoch is a single pass through our full training data(60,000 images). An epoch consists of training steps, which is nothing, but the number of batches passed to the model until all the training data is covered.

It could be expressed as number of training steps = number of training records/batch size, which is 600(60000/100) in our case. We'll train the model for 10 epochs- the model will see the full training data exactly 10 times.

- **Flattening the image** - Instead of sending the image as a 2D tensor, we flatten it in one-dimension.

Test the Trained Neural Network

The training loss keeps on decreasing throughout the epochs and we can conclude that our model is definitely learning. But to gauge the performance of our model we'll have to see how well it does on unseen(test) data.

```
print(f'Test accuracy: {test_correct[-1].item()*100/10000:.3f}%') # test accuracy for the last epoch
```

```
Test accuracy: 97.800%
```

Predictions are made on our test data after training completes in every epoch. Since our model continually keeps getting better, the test accuracy of the last epoch is the best.