श्रद्धावान् लभते ज्ञानम्

AMRITA
VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY

Implementation in PyTorch

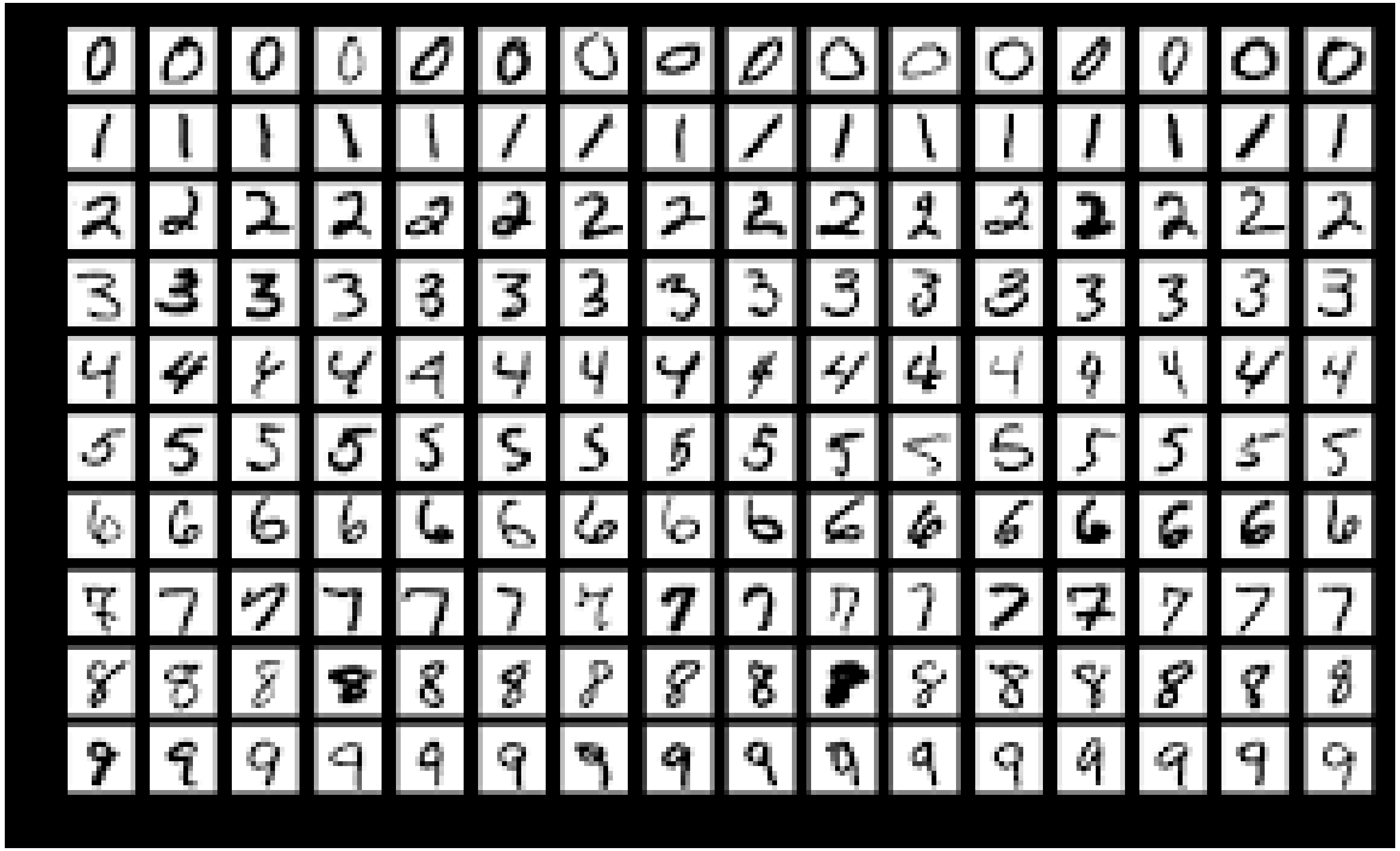
Building a Feed Forward Neural Network Model

- Model A: 1 Hidden Layer Feedforward Neural Network (Sigmoid Activation)
- Model B: 1 Hidden Layer Feedforward Neural Network (Tanh Activation)
- Model C: 3 Hidden Layer Feedforward Neural Network (ReLU Activation)
- Model D : 3-layer FNN with ReLU Activation on GPU
- Model E : Test Loop example

Refer Sharepoint for the following Practice Codes

- Model A: 1 Hidden Layer Feedforward Neural Network (Sigmoid Activation)
  - ✓ FeedForward-ModelA.ipynb
- Model B :1-layer FNN with Tanh Activation
  - ✓FeedForward Model B.ipynb
- Model C : 3-layer FNN with ReLU Activation
  - ✓FeedForward-ModelC.ipynb
- Model D : 3-layer FNN with ReLU Activation on GPU
  - ✓FeedForward-ModelD.ipynb
- Model E: TestLoop
  - ✓FeedForward-with TestLoop.ipynb

Dataset: MNIST dataset- handwritten digits



The MNIST database of handwritten digits, It is a dataset of 60,000 small square 28×28 pixel (784) grayscale images of handwritten single digits between 0 and 9.

Training set size: 60,000 images
Test set : 10,000 images
No: of classes: 10
Class labels- (0-9)
Size of each image : 28x28
Input Size: 784(28*28)

http://yann.lecun.com/exdb/mnist/

Building a Feedforward Neural Network with PyTorch

- Steps
  - Step 1: Load Dataset
  - Step 2: Make Dataset Iterable
  - Step 3: Create Model Class
  - Step 4: Instantiate Model Class
  - Step 5: Instantiate Loss Class
  - Step 6: Instantiate Optimizer Class
  - Step 7: Train Model

Import libraries

```
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets
```

PyTorch has two primitives to work with data: torch.utils.data.DataLoader and torch.utils.data.Dataset

PyTorch offers domain-specific libraries such as TorchText, TorchVision, and TorchAudio, all of which include datasets. For this tutorial, we will be using a TorchVision dataset.

The torchvision.datasets module contains Dataset objects for many real-world vision data like CIFAR, COCO. In this tutorial, we use the MNIST dataset. Every TorchVision Dataset includes two arguments: transform and target_transform to modify the samples and labels respectively

Model A: 1 Hidden Layer Feedforward Neural Network (Sigmoid Activation)

Step1 : Loading MNIST Train Dataset

```
train_dataset = dsets.MNIST(root='./data',
                            train=True,
                            transform=transforms.ToTensor(),
                            download=True)

test_dataset = dsets.MNIST(root='./data',
                           train=False,
                           transform=transforms.ToTensor())
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz
 9913344/? [00:00<00:00, 56442143.75it/s]
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw
```

Step 2: Make Dataset Iterable

```
[3]  batch_size = 100
     n_iters = 3000
     num_epochs = n_iters / (len(train_dataset) / batch_size)
     num_epochs = int(num_epochs)

     train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                                batch_size=batch_size,
                                                shuffle=True)

     test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                               batch_size=batch_size,
                                               shuffle=False)
```

Batch sizes and iterations - MiniBatch
Because we have 60000 training samples (images), we need to split them up to small groups (batches) and pass these batches of samples to our feedforward neural network subsequently.

If we have 60,000 images and we want a batch size of 100,
60000/100=600 iterations
An epoch means that you have successfully passed the whole training set, 60,000 images, to the model.
1 epoch has 600 iterations
If we want to go through the whole dataset 5 times (5 epochs) for the model to learn, then we need 3000 iterations (600 x 5=3000).
3000/60,000/100=5

Step 3: Create Model Class

```
class FeedforwardNeuralNetModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(FeedforwardNeuralNetModel, self).__init__()
        # Linear function
        self.fc1 = nn.Linear(input_dim, hidden_dim)

        # Non-linearity
        self.sigmoid = nn.Sigmoid()

        # Linear function (readout)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        # Linear function  # LINEAR
        out = self.fc1(x)

        # Non-linearity  # NON-LINEAR
        out = self.sigmoid(out)

        # Linear function (readout)  # LINEAR
        out = self.fc2(out)
        return out
```

CLASS **torch.nn.Module** is the Base class for all neural network modules.

- super().**init**() creates a class that tracks the architecture and provides a lot of useful methods and attributes.
- self.fc1 = nn.Linear(input_dim, hidden_dim): This line creates a module for a linear transformation, input_dim inputs and hidden_dim outputs for first hidden layer and assigns it to self.fc1. The module automatically creates the weight and bias tensors which we'll use in the forward method.
- Then a sigmoid activation function Layer is there.
- The following line indicates the output layer creating another linear transformation with hidden_dim inputs and output_dim output which is 10 output classes.

Alternative

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

Alternative: nn.sequential (refer ModelE Testloop)

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork()
```

Step 4: Instantiate Model Class

```
[5] input_dim = 28*28
    hidden_dim = 100
    output_dim = 10

    model = FeedforwardNeuralNetModel(input_dim, hidden_dim, output_dim)
```

Input dimension: 784 Size of image $28 \times 28 = 784$

Output dimension: 10 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Hidden dimension: 100 Can be any number

Our input size is determined by the size of the image (numbers ranging from 0 to 9) which has a width of 28 pixels and a height of 28 pixels. Hence the size of our input is 784 (28 x 28).

Our output size is what we are trying to predict. When we pass an image to our model, it will try to predict if it's 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. That is a total of 10 classes, hence we have an output size of 10.

Step 5: Instantiate Loss Class

```
[6]  criterion = nn.CrossEntropyLoss()
```

Step 6: Instantiate Optimizer Class

```
[11] learning_rate = 0.1

     optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

- Simplified equation
  - $\theta = \theta - \eta \cdot \nabla_\theta$
    - θ: parameters (our tensors with gradient accumulation capabilities)
    - η: learning rate (how fast we want to learn)
    - ∇_θ: parameters' gradients
- Even simplier equation
  - `parameters = parameters - learning_rate * parameters_gradients`
  - **At every iteration, we update our model's parameters**

Parameters In-Depth- Print Parameters

```
[12] print(model.parameters())
     print(len(list(model.parameters())))
     # FC 1 Parameters
     print(list(model.parameters())[0].size())
     # FC 1 Bias Parameters
     print(list(model.parameters())[1].size())
     # FC 2 Parameters
     print(list(model.parameters())[2].size())
     # FC 2 Bias Parameters
     print(list(model.parameters())[3].size())
```

```
<generator object Module.parameters at 0x7f7108c17d50>
4
torch.Size([100, 784])
torch.Size([100])
torch.Size([10, 100])
torch.Size([10])
```

Step 7: Train Model

- Steps
  - Convert inputs to tensors with gradient accumulation capabilities
  - Clear gradient buffers
  - Get output given inputs
  - Get loss
  - Get gradients w.r.t. parameters
  - Update parameters using gradients
  - parameters = parameters - learning_rate * parameters_gradients
  - REPEAT

Step 7: Train Model

```
[13] iter = 0
     for epoch in range(num_epochs):
         for i, (images, labels) in enumerate(train_loader):
             # Load images with gradient accumulation capabilities
             images = images.view(-1, 28*28).requires_grad_()

             # Clear gradients w.r.t. parameters
             optimizer.zero_grad()

             # Forward pass to get output/logits
             outputs = model(images)

             # Calculate Loss: softmax --> cross entropy loss
             loss = criterion(outputs, labels)

             # Getting gradients w.r.t. parameters
             loss.backward()

             # Updating parameters
             optimizer.step()

             iter += 1
```

```
if iter % 500 == 0:
    # Calculate Accuracy
    correct = 0
    total = 0
    # Iterate through test dataset
    for images, labels in test_loader:
        # Load images with gradient accumulation capabilities
        images = images.view(-1, 28*28).requires_grad_()

        # Forward pass only to get logits/output
        outputs = model(images)

        # Get predictions from the maximum value
        _, predicted = torch.max(outputs.data, 1)

        # Total number of labels
        total += labels.size(0)

        # Total correct predictions
        correct += (predicted == labels).sum()

    accuracy = 100 * correct / total

    # Print Loss
    print('Iteration: {}. Loss: {}. Accuracy: {}'.format(iter, loss.item(), accuracy))
```

Output

```
Iteration: 500. Loss: 0.6632527709007263. Accuracy: 85.3499984741211
Iteration: 1000. Loss: 0.33584755659103394. Accuracy: 89.47000122070312
Iteration: 1500. Loss: 0.3813075125217438. Accuracy: 90.43000030517578
Iteration: 2000. Loss: 0.23587213456630707. Accuracy: 91.30000305175781
Iteration: 2500. Loss: 0.29119277000427246. Accuracy: 91.76000213623047
Iteration: 3000. Loss: 0.2163892388343811. Accuracy: 91.95999908447266
```

Homework

Visualize the Train Results

```
*   Draw the Train Loss graph
*   Draw the Train Accuracy/Error Graphs
```

Perform Validation and Testing Use Softmax function at the output layer and get output probabilities

Visualize the Validation/Test Results

```
    *   Draw the Validation Accuracy/Error Graphs
    *   calculate and draw Confusion Matrix
    *   Compute Precision Recall, F1 Score
```

Use Tensorboard in Pytorch to analyse the results of the above problem

- Refer the link https://pytorch.org/tutorials/recipes/recipes/tensorboard_with_pytorch.html

Namah Shivaya