Multi-Layer Feed Forward Networks

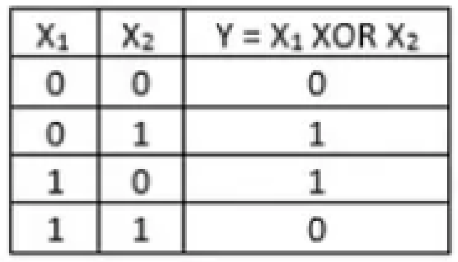
1. Develop a Feedforward neural for XOR Problem that takes two binary inputs, and simulate the logical operation of XOR
2. Develop a feedforward neural model for MNIST Digit classification

XOR Problem using FeedForward network

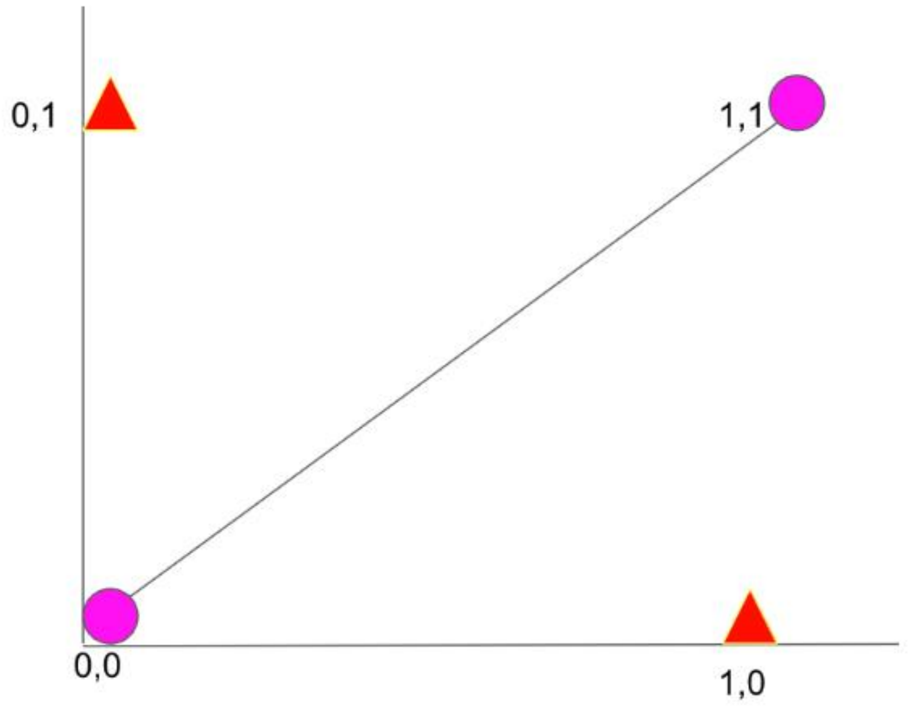
The linear separability of points

- Linear separability of points is the ability to classify the data points in the hyperplane by avoiding the overlapping of the classes in the planes.
- Each of the classes should fall above or below the separating line and then they are termed as linearly separable data points
- data points have to be linearly separable to eradicate the issues with wrong weight updation and wrong classifications

XOR Logic – truth table for XOR



X_1	X_2	$Y = X_1$ XOR X_2
0	0	0
0	1	1
1	0	1
1	1	0

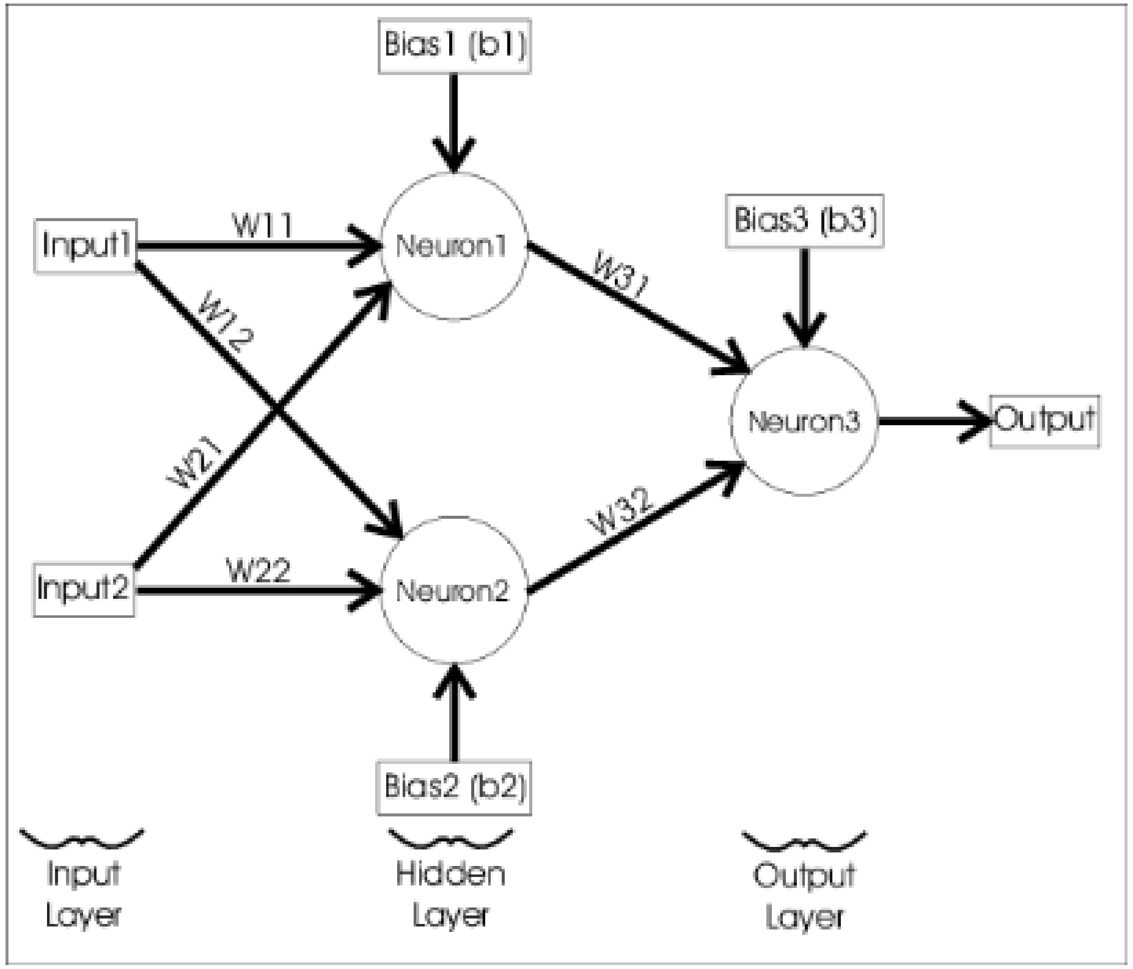


linear separability of data points is not possible using the XOR logic.
In the above figure, we can see that above the linear separable line the red triangle is overlapping with the pink dot

The Neural Network Model for XOR Logic

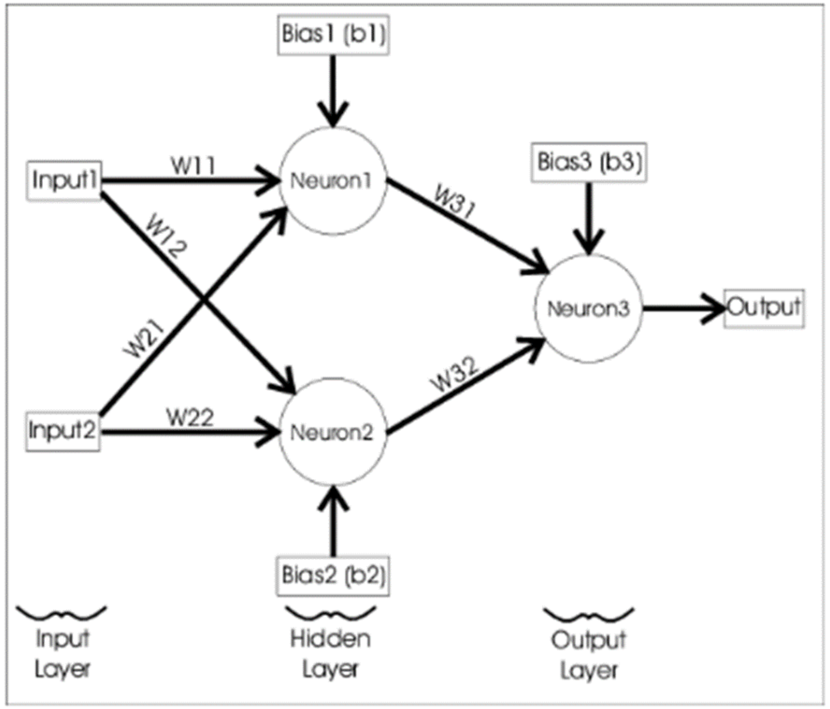
- The neural network needs to produce two different decision planes to linearly separate the input data based on the output patterns.
- This is achieved by using the concept of hidden layers
  - The neural network will consist of one input layer with two nodes (X1,X2);
  - one hidden layer with two nodes (since two decision planes are needed);
  - and one output layer with one node (Y).
- Hence, the XOR problem with neural networks can be solved using a neural network architecture with an input layer having two nodes, hidden layer with two nodes, and one output layer with one node.

The Neural Network Model for XOR Logic



The Neural Network Model for XOR Logic

- Define the neural network model
- Find the total number of learnable parameters



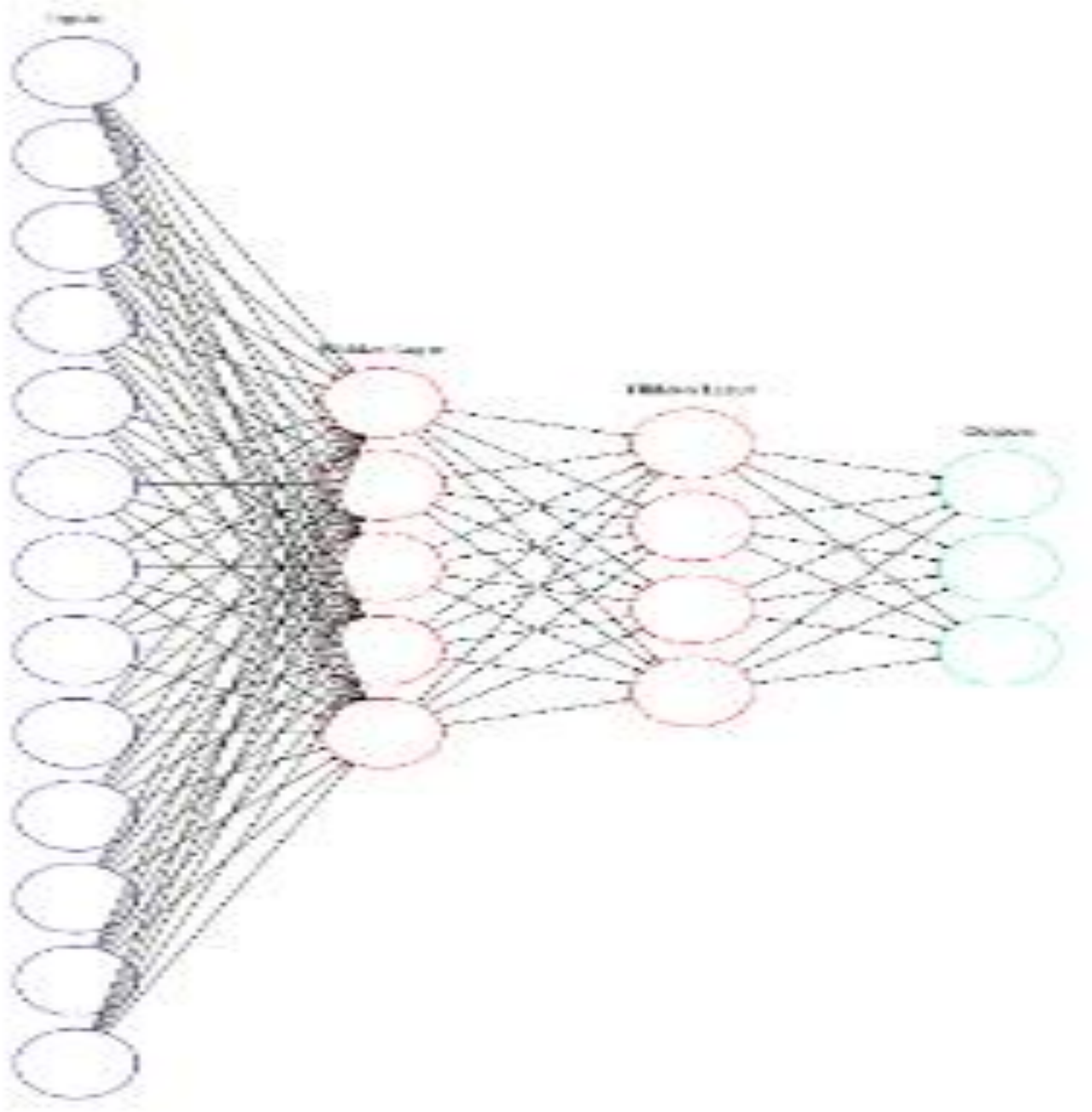
The Neural Network Model for XOR Logic

- Define the neural network model

- Find the total number of learnable parameters = 6 weights + 3 biases= 9 parameters

Problem - Find the total number of learnable parameters

- A neural network with 13 input neurons, two hidden layers of 5 and 4 neurons, and an output layer of 3 neurons.



Find the total number of learnable parameters

The total number of weights is (13 * 5) + (5 * 4) + (4 * 3) = 97.

The total number of biases is 5 + 4 + 3 = 12.

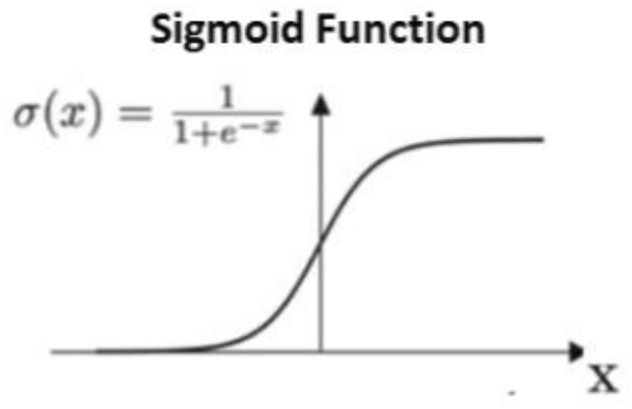
Total number of parameters 97 + 12 = 109.

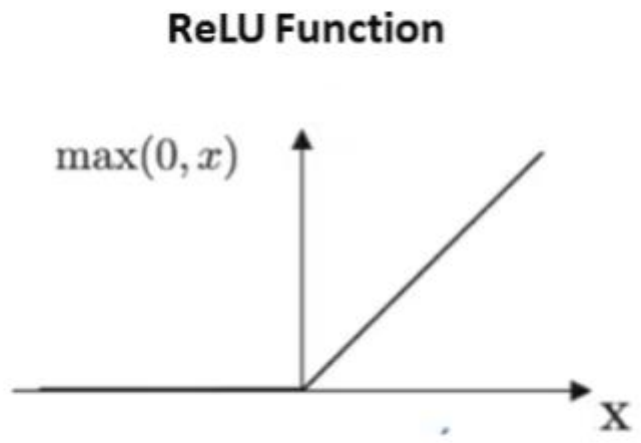
Activation functions

- These activation functions are non-linear functions (except for “identity”) which allow the neural network to learn non-linear relationships between the input and output data.
- Each of the activation functions transform values in different ways and will create different training processes, learned mappings, and output predictions in a neural network
- So, training to see what creates the best result by changing activation functions

Sigmoid and ReLU

- “Sigmoid” : uses the logistic sigmoid function, f(x) = 1 / (1 + exp(-x)) (the output will always range be greater than 0 and less than 1)
- “ReLU” : uses the rectified linear unit function, f(x) = max(x,0) (the output will always be greater than or equal to 0)





THE LEARNING ALGORITHM

- The information of a neural network is stored in the interconnections between the neurons i.e. the weights.
- A neural network learns by updating its weights according to a learning algorithm that helps it converge to the expected output.
- The learning algorithm is a principled way of changing the weights and biases based on the loss function.

1. Initialize the weights and biases randomly.

2. Iterate over the data

- i. Compute the predicted output using the sigmoid function
- ii. Compute the loss using the square error loss function
- iii. W(new) = W(old) — α ΔW
- iv. B(new) = B(old) — α ΔB

3. Repeat until the error is minimal

The algorithm divided into two parts: the forward pass and the backward pass also known as “backpropagation.”

Importing necessary Libraries

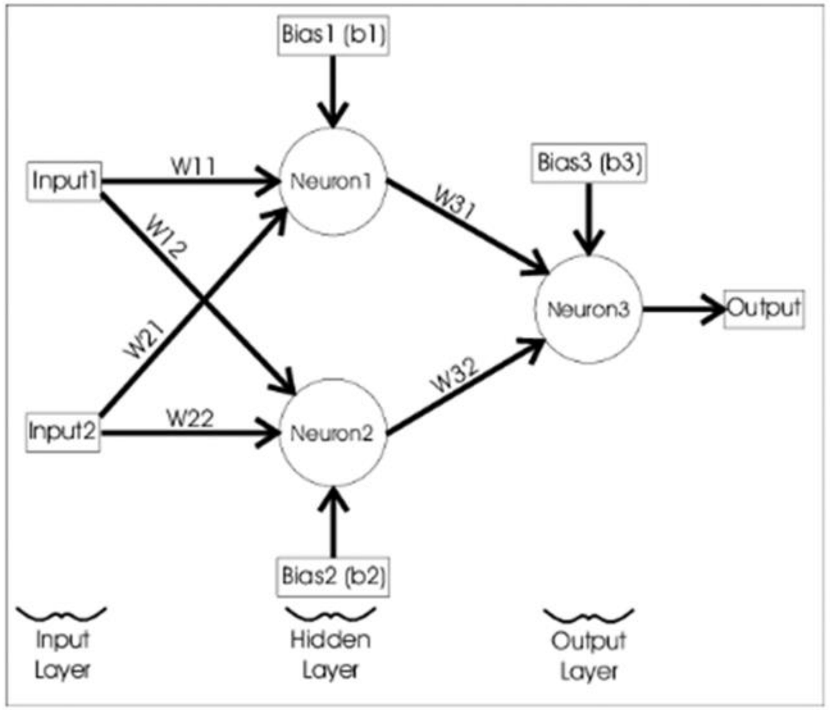
```
import torch
from matplotlib import pyplot as plt
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
import numpy as np

loss_list = []
torch.manual_seed(42)
```

Initialize inputs and expected outputs

- Step 1: Initialize inputs and expected outputs as per the truth table of XOR
- Create the tensors x1,x2 and y.
- They are the training examples in the dataset for the XOR
- X = torch.tensor([[0,0],[0,1],[1,0],[1,1]], dtype=torch.float32)
- Y = torch.tensor([0,1,1,0], dtype=torch.float32)

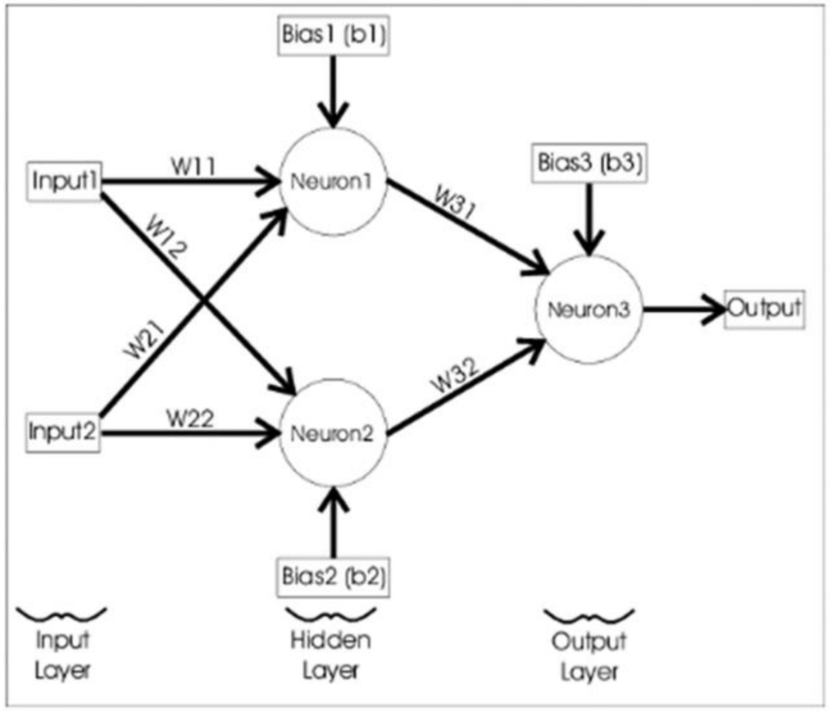
Define XORModel class

- Step 2: Define XORModel class - write constructor and forward function



Define XORModel class

- Step 2: Define XORModel as class



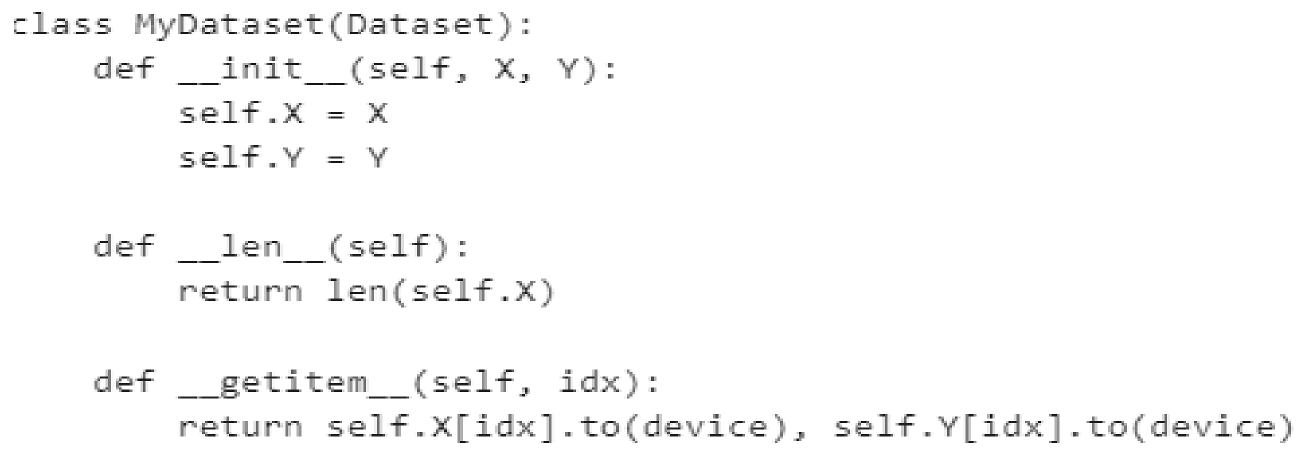
```
class XORModel(nn.Module):
    def __init__(self):
        super(XORModel, self).__init__()
        #self.w = torch.nn.Parameter(torch.rand([1]))
        #self.b = torch.nn.Parameter(torch.rand([1]))

        self.linear1 = nn.Linear(2,2,bias=True)
        self.activation1 = nn.Sigmoid()
        self.linear2 = nn.Linear(2,1,bias=True)
        #self.activation2 = nn.ReLU()
    def forward(self, x):
        x = self.linear1(x)
        x = self.activation1(x)
        x = self.linear2(x)
        #x = self.activation2(x)
        return x
```

Define XORModel class

- In PyTorch, neural networks are created by using Object Oriented Programming.
- The layers are defined in the init function
- Forward pass is defined in the forward function, which is invoked automatically when the class is called.
- These Functions are possible because of the class nn.Module from torch which was inherited

write Dataset class

- Step 3: Create DataLoader. Write Dataset class with necessary constructors and methods – len() and getitem()



```python
class MyDataset(Dataset):
    def __init__(self, X, Y):
        self.X = X
        self.Y = Y

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx].to(device), self.Y[idx].to(device)
```

- Create the dataset
- full_dataset = MyDataset(X, Y)

Dataset and DataLoader

- Dataset class in pytorch covers the data in a tuple
- Enables us to access the index of each data.
- This is necessary to create dataloader class which can be used to shuffle, apply Mini-Batch Gradient Descent etc
  - The init function is used to initialise the x and y of our dataset
  - getitem function is used to return a particular index in the dataset. It returns both the x and y value.
  - len function returns the size of the dataset
- DataLoader is used to perform mini batch or stochastic gradient descent by acting as an iterable

dataLoader and GPU loading

```
batch_size = 1
```

Create the dataloaders for reading data - # This provides a way to read the dataset in batches, also shuffle the data

```
train_data_loader = DataLoader(full_dataset, batch_size=batch_size, shuffle=True)
```

Find if CUDA is available to load the model and device on to the available device CPU/GPU

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

Load the model to GPU

```
model = XORModel().to(device)
print(model)
```

MSELoss Criterion and Optimizer

Add the criterion which is the MSELoss

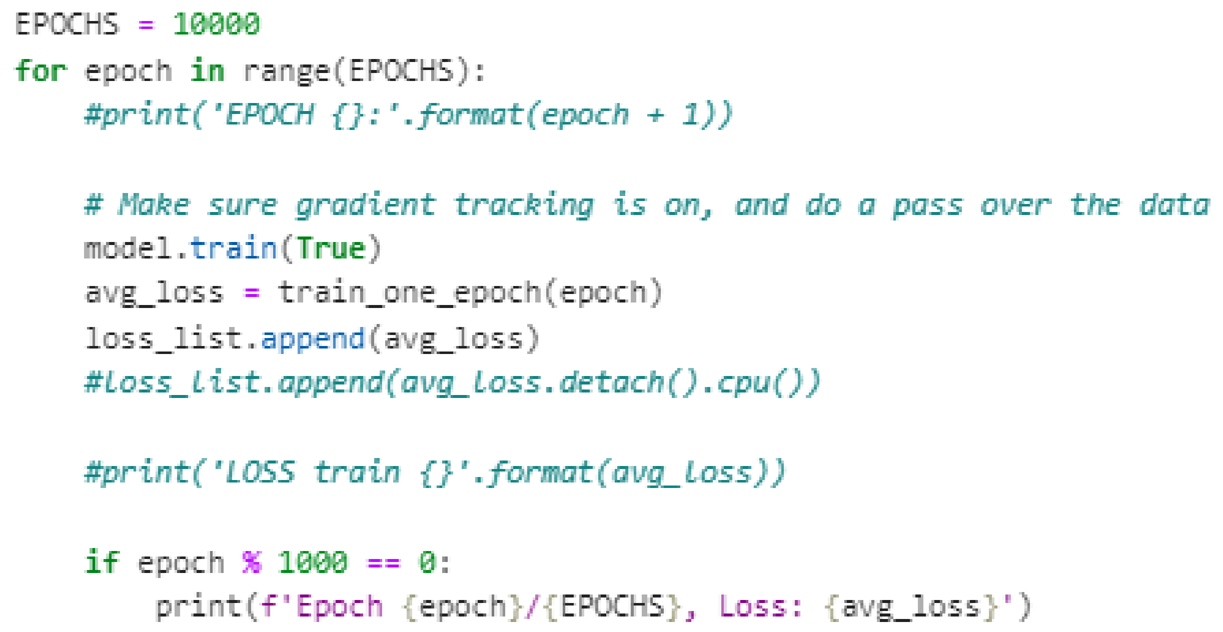
```
loss_fn = torch.nn.MSELoss()
```

Optimizers specified in the torch.optim package

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.03)
```

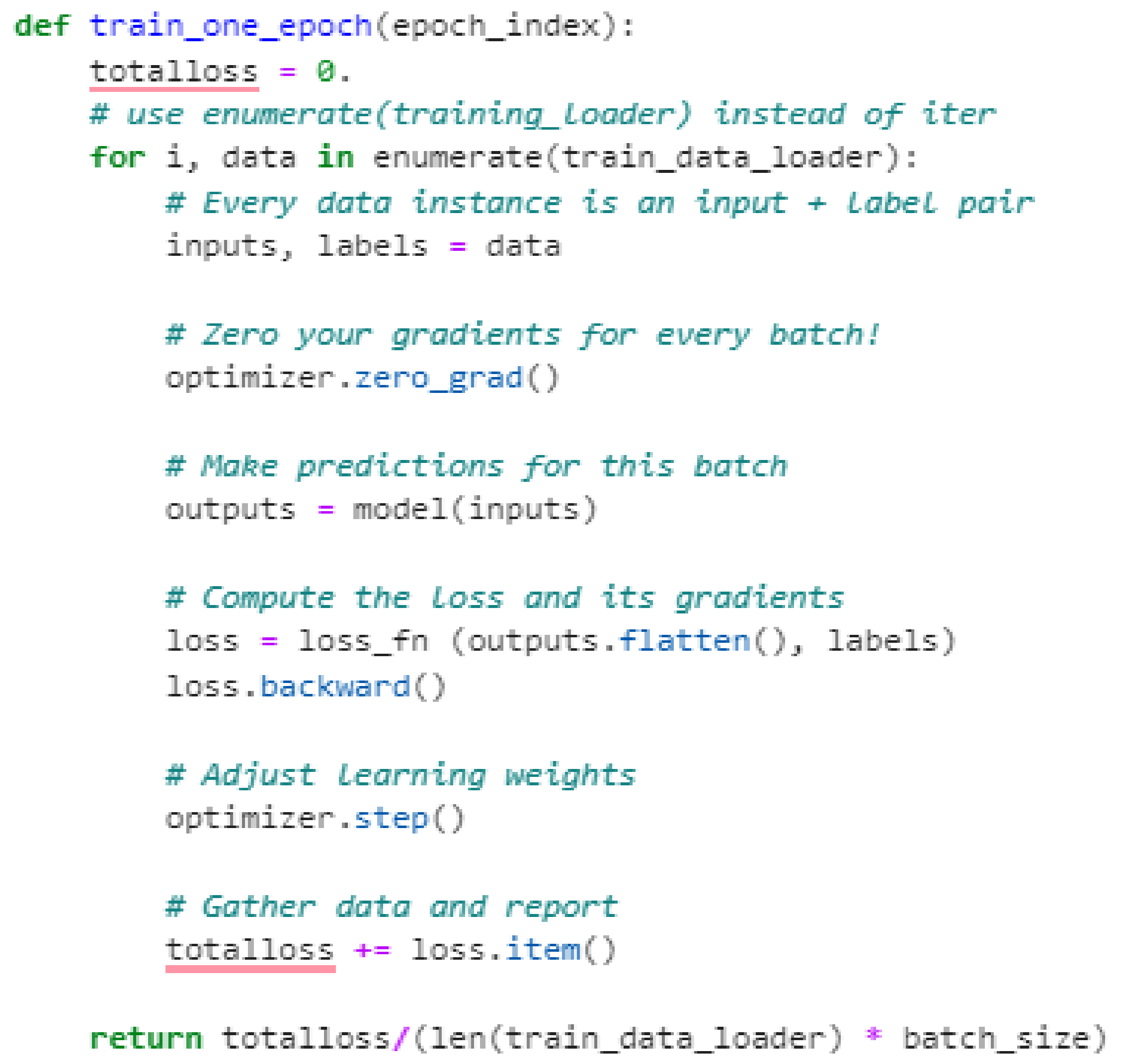
Training Loop

```
EPOCHS = 5000
for epoch in range(EPOCHS):
    print('EPOCH {}:'.format(epoch + 1))

    # Make sure gradient tracking is on, and do
a pass over the data
    model.train(True)
    avg_loss = train_one_epoch(epoch)
    loss_list.append(avg_loss.detach().cpu())

    print('LOSS train {}'.format(avg_loss))
```



```
EPOCHS = 10000
for epoch in range(EPOCHS):
    #print('EPOCH {}:'.format(epoch + 1))

    # Make sure gradient tracking is on, and do a pass over the data
    model.train(True)
    avg_loss = train_one_epoch(epoch)
    loss_list.append(avg_loss)
    #loss_list.append(avg_loss.detach().cpu())

    #print('LOSS train {}'.format(avg_loss))

    if epoch % 1000 == 0:
        print(f'Epoch {epoch}/{EPOCHS}, Loss: {avg_loss}')
```

Training an epoch



```python
def train_one_epoch(epoch_index):
    totalloss = 0.
    # use enumerate(training_loader) instead of iter
    for i, data in enumerate(train_data_loader):
        # Every data instance is an input + label pair
        inputs, labels = data

        # Zero your gradients for every batch!
        optimizer.zero_grad()

        # Make predictions for this batch
        outputs = model(inputs)

        # Compute the loss and its gradients
        loss = loss_fn (outputs.flatten(), labels)
        loss.backward()

        # Adjust learning weights
        optimizer.step()

        # Gather data and report
        totalloss += loss.item()

    return totalloss/(len(train_data_loader) * batch_size)
```

Model Inference step

```
for param in model.named_parameters():
    print(param)
```

Model inference – similar to prediction in ML

```
input = torch.tensor([0, 1], dtype=torch.float32).to(device)
model.eval()
print("The input is = {}".format(input))
print("Output y predicted ={}".format(model(input)))
#Display the plot
plt.plot(loss_list)
plt.show()
```

Output – Model parameters

```
Model parameters= ('linear1.weight', Parameter containing:
tensor([[ 0.5413,  0.5890],
        [-0.1679,  0.6455]], requires_grad=True))
Model parameters= ('linear1.bias', Parameter containing:
tensor([-0.1505,  0.1357], requires_grad=True))
Model parameters= ('linear2.weight', Parameter containing:
tensor([[-0.3832,  0.3738]], requires_grad=True))
Model parameters= ('linear2.bias', Parameter containing:
tensor([0.5562], requires_grad=True))
```

Output- XOR model

```
XOR Model= XORModel(
  (linear1): Linear(in_features=2, out_features=2, bias=True)
  (activation1): Sigmoid()
  (linear2): Linear(in_features=2, out_features=1, bias=True)
)
The input is = tensor([0., 1.])
Output y predicted =tensor([1.0000], grad_fn=<ViewBackward0>)
```

Outputs after applying round

```
#Model Inference
input = torch.tensor([0, 1], dtype=torch.float32).to(device)
model.eval()
print("The input is = {}".format(input))
print("Output y predicted ={}".format(model(input)))
inference_binary = torch.round(model(input))
print("inference-binary=", inference_binary)
test_inputs = torch.tensor([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float32)
inference_binary_inputs = torch.round(model(test_inputs))
print("\nInference for test inputs:", model(test_inputs))
print("\nInference Binary for test inputs:",inference_binary_inputs )
```

Outputs after applying round

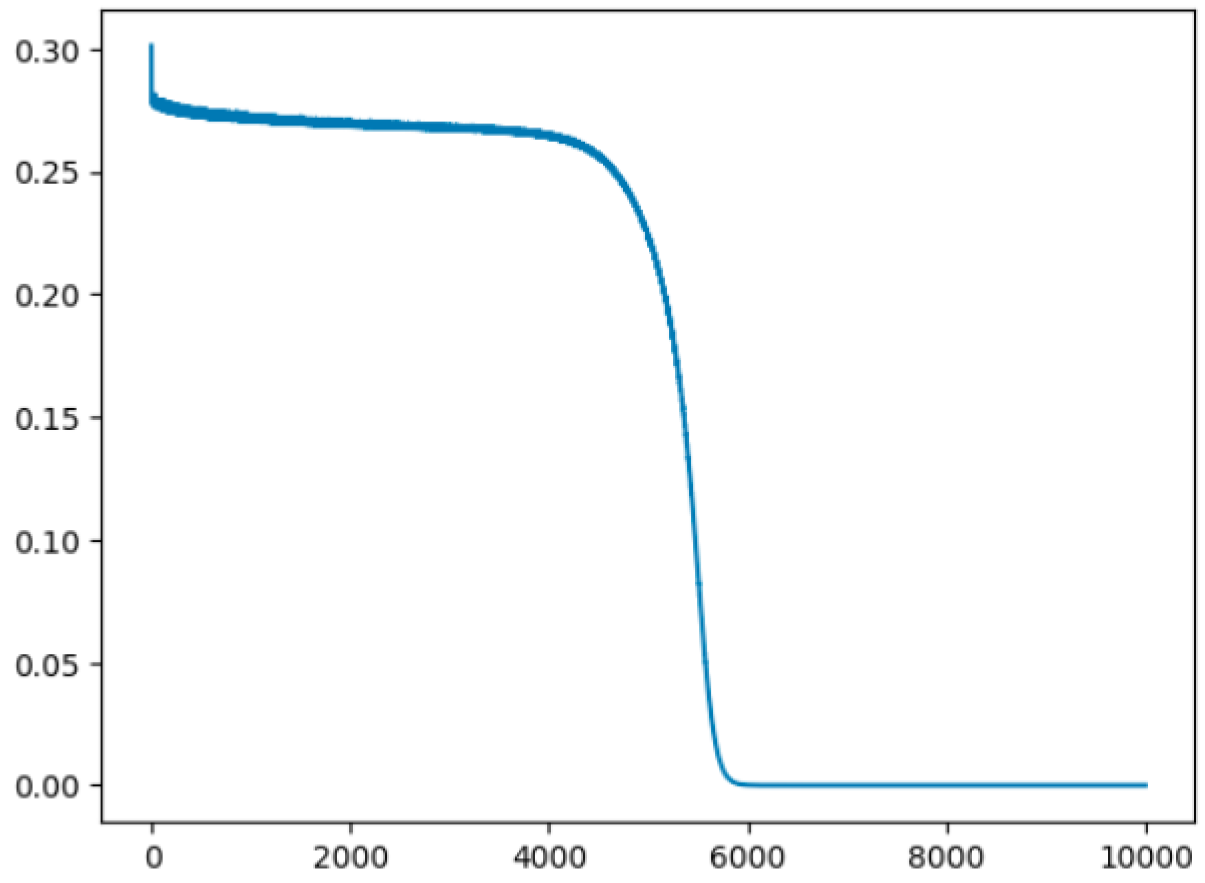
```
Inference for test inputs: tensor([[5.9605e-07],
        [1.0000e+00],
        [1.0000e+00],
        [1.9073e-06]], grad_fn=<AddmmBackward0>)

Inference Binary for test inputs: tensor([[0.],
        [1.],
        [1.],
        [0.]], grad_fn=<RoundBackward0>)
```

Final Result – Note the parameters for verification

```
XOR Model= XORModel(
  (linear1): Linear(in_features=2, out_features=2, bias=True)
  (activation1): Sigmoid()
  (linear2): Linear(in_features=2, out_features=1, bias=True)
)
Epoch 0/10000, Loss: 0.3011328335851431
Epoch 1000/10000, Loss: 0.27038102224469185
Epoch 2000/10000, Loss: 0.27022556215524673
Epoch 3000/10000, Loss: 0.26847539842128754
Epoch 4000/10000, Loss: 0.26516808941960335
Epoch 5000/10000, Loss: 0.22409666888415813
Epoch 6000/10000, Loss: 0.00014026711960468674
Epoch 7000/10000, Loss: 1.1688428003253648e-11
Epoch 8000/10000, Loss: 3.240074875066057e-12
Epoch 9000/10000, Loss: 3.0810909379397344e-12
```

```
Model parameters= ('linear1.weight',
Parameter containing:
tensor([[-1.9767,  2.0750],
        [-3.0809,  3.3435]],
requires_grad=True))
Model parameters= ('linear1.bias',
Parameter containing:
tensor([ 0.7012, -2.5002],
requires_grad=True))
Model parameters= ('linear2.weight',
Parameter containing:
tensor([[-2.6653,  2.7713]],
requires_grad=True))
Model parameters= ('linear2.bias',
Parameter containing:
tensor([1.5715], requires_grad=True))
The input is = tensor([0., 1.])
Output y predicted =tensor([1.0000],
grad_fn=<ViewBackward0>)
```

Final Result



Verification –Same parameter values taken from the output

```
import torch
weight1= torch.tensor([[-1.9767,  2.0750],[-3.0809,  3.3435]])
bias1= torch.tensor([ 0.7012, -2.5002])
weight2 = torch.tensor([[-2.6653,  2.7713]])
bias2=torch.tensor([1.5715])
X = torch.tensor([[0,0],[0,1],[1,0],[1,1]], dtype=torch.float32)
Y = torch.tensor([0,1,1,0], dtype=torch.float32)
result= torch.mm(X, weight1.t())+bias1
Sigm = torch.nn.Sigmoid()
sg_result = Sigm(result)
final_result= torch.mm(sg_result, weight2.t())+bias2
print(final_result)
```

Use directly model.linear1.weight, model.linear1.bias, model.linear2.weight, model.linear2. bias

Verification –Same parameter values taken from the output

```
import torch
weight1= torch.tensor([[-1.9767,  2.0750],[-3.0809,  3.3435]])
bias1= torch.tensor([ 0.7012, -2.5002])
weight2 = torch.tensor([[-2.6653,  2.7713]])
bias2=torch.tensor([1.5715])
X = torch.tensor([[0,0],[0,1],[1,0],[1,1]], dtype=torch.float32)
Y = torch.tensor([0,1,1,0], dtype=torch.float32)
result= torch.mm(X, weight1.t())+bias1
Sigm = torch.nn.Sigmoid()
sg_result = Sigm(result)
final_result= torch.mm(sg_result, weight2.t())+bias2
print(final_result)
```

```
tensor([[5.7220e-05],
        [1.0000e+00],
        [1.0000e+00],
        [1.7166e-05]])
```

Problem 2 - Feedforward neural model for MNIST Digit classification

Feedforward neural model for MNIST Digit classification

- A popular demonstration of the capability of deep learning techniques is object recognition in image data.
- The MNIST Handwritten Digit Recognition Problem
- The MNIST dataset was developed by Yann LeCun, Corinna Cortes, and Christopher Burges for evaluating machine learning models on the handwritten digit classification problem.
- The dataset was constructed from a number of scanned document datasets available from the National Institute of Standards and Technology (NIST).
- This is where the name for the dataset comes from, the Modified NIST or MNIST dataset.
- Images of digits were taken from a variety of scanned documents, normalized in size, and centered.
- This makes it an excellent dataset for evaluating models, to focus on minimal data cleaning or preparation

Feedforward neural model for MNIST Digit classification

- Baseline Model with feed-forward networks
- We can use a complex model like a convolutional neural network to get the best results with MNIST
- The MNIST dataset, also known as the Modified National Institute of Standards and Technology dataset, consists of a total of 70,000 images, the training set having 60,000 and the test set has 10,000.
- This means that there are 10 classes of digits, which includes the labels for the numbers 0 to 9.
- This dataset is mainly used for text classification using deep learning models.
- Each image is a 28×28-pixel square (flattening this into a one-dimensional array results in 784 pixels total) in grayscale.

Feedforward neural model for MNIST Digit classification

- The output targets are labels in the form of integers from 0 to 9.
- This is a multiclass classification problem.
- So use the cross entropy function to evaluate the model performance

Feedforward neural model for MNIST Digit classification

Step 1- Importing necessary libraries

- The torchvision library provides specialized functions for computer vision tasks.
- The dataset is downloaded the first time this function is called and stored locally, so no need to download again in the future

```
import torch
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
import matplotlib.pyplot as plt
import torchvision.datasets as datasets
from torchvision.transforms import ToTensor
```

Feedforward neural model for MNIST Digit classification

Step 2 - Loading the MNIST Dataset in PyTorch

```
#Create the dataset
mnist_trainset = datasets.MNIST(root='./data', train=True,
download=True, transform = ToTensor())
mnist_testset = datasets.MNIST(root='./data', train=False,
download=True, transform = ToTensor())
```

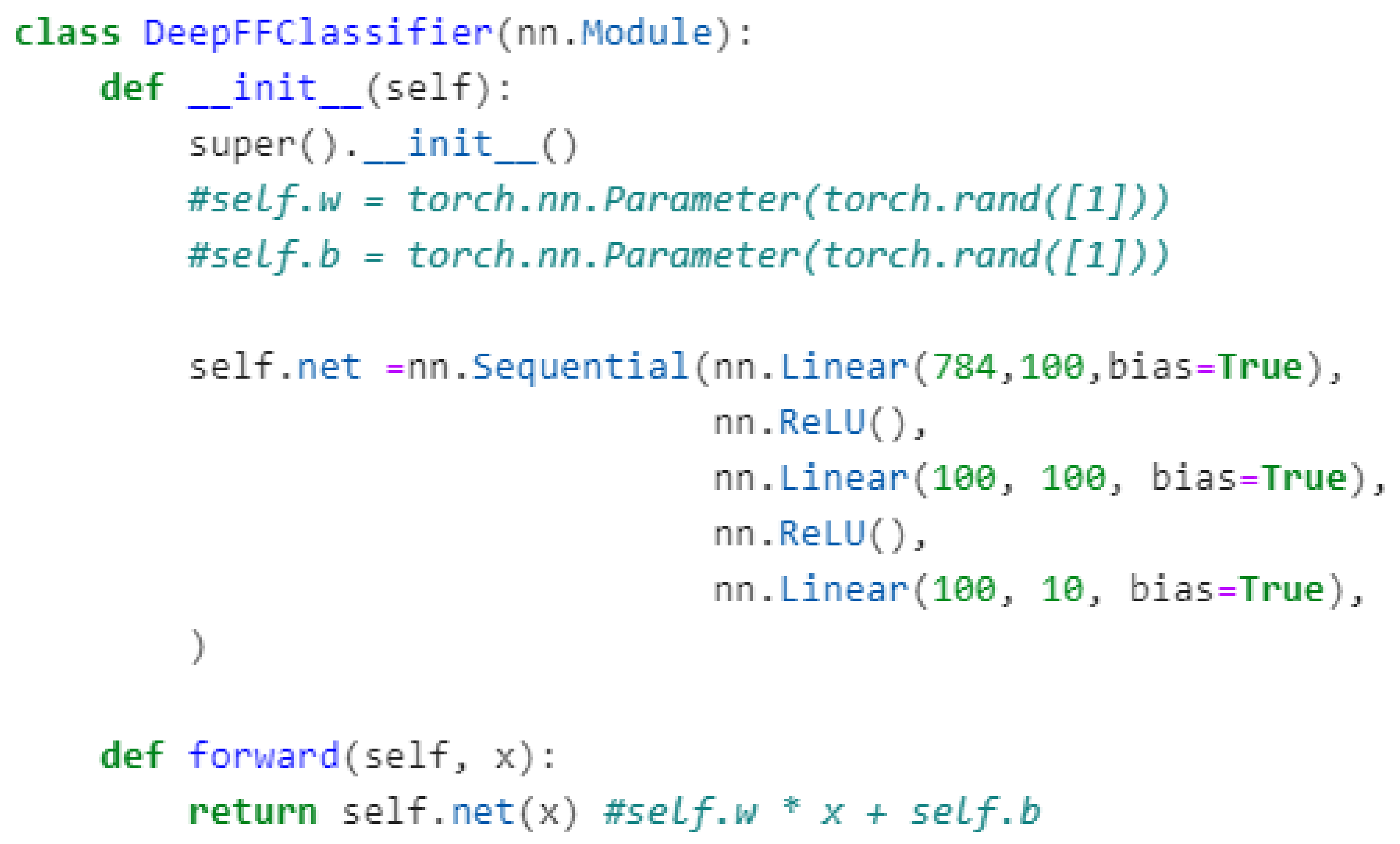
Note: Check the folder titled “data” under the current working directory for the downloaded dataset

Transform - ToTensor

- transforms.ToTensor()
- converts the image into numbers, that are understandable by the system.
- It separates the image into three color channels (separate images): red, green & blue (note: MNIST data set is grayscale image)
- Then it converts the pixels of each image to the brightness of their color between 0 and 255.
- These values are then scaled down to a range between 0 and 1
- The image is now a Torch Tensor

Feedforward neural model for MNIST Digit classification

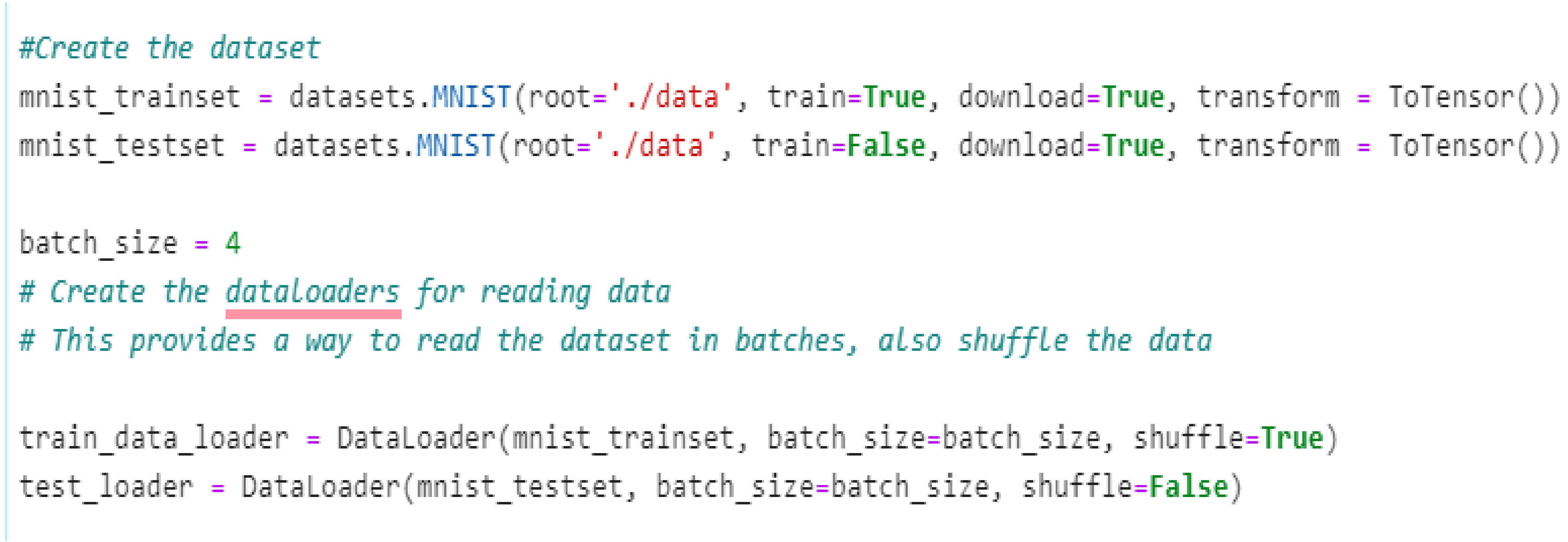
Step 3 - Define Network model



```python
class DeepFFClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        #self.w = torch.nn.Parameter(torch.rand([1]))
        #self.b = torch.nn.Parameter(torch.rand([1]))

        self.net =nn.Sequential(nn.Linear(784,100,bias=True),
                                nn.ReLU(),
                                nn.Linear(100, 100, bias=True),
                                nn.ReLU(),
                                nn.Linear(100, 10, bias=True),
        )

    def forward(self, x):
        return self.net(x) #self.w * x + self.b
```

Feedforward neural model for MNIST Digit classification

Step 4 - Create dataset and dataloader

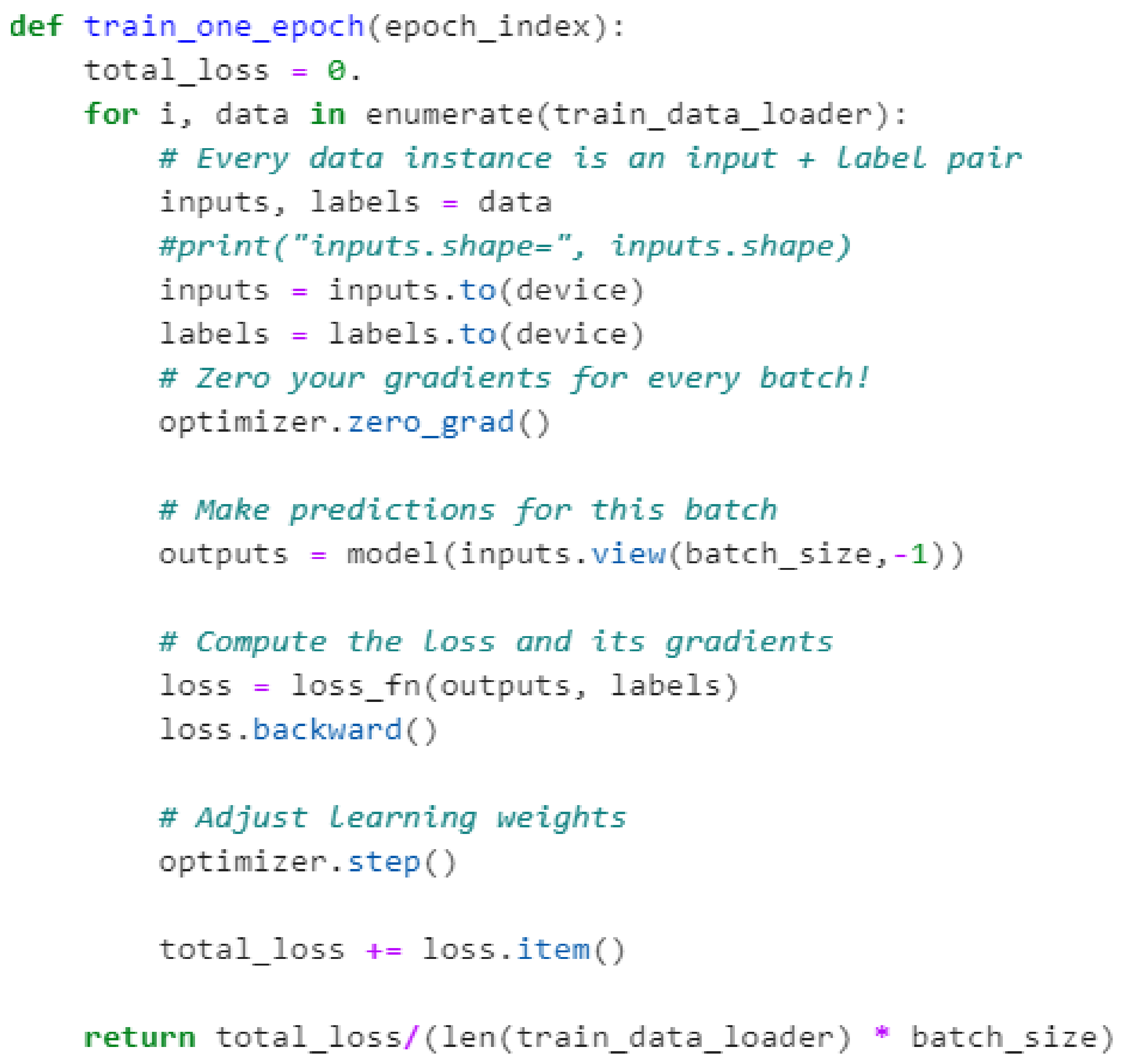


```
#Create the dataset
mnist_trainset = datasets.MNIST(root='./data', train=True, download=True, transform = ToTensor())
mnist_testset = datasets.MNIST(root='./data', train=False, download=True, transform = ToTensor())

batch_size = 4
# Create the dataloaders for reading data
# This provides a way to read the dataset in batches, also shuffle the data

train_data_loader = DataLoader(mnist_trainset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(mnist_testset, batch_size=batch_size, shuffle=False)
```

Feedforward neural model for MNIST Digit classification

- So, dataset is downloaded, shuffled and transformed to tensors
- Now, load them to DataLoader, which combines the data-set and a sampler and provides single or multi-process iterators over the data-set
- batch size is the number of images to read in one go
- Here batch_size = 4

Training within an epoch



```python
def train_one_epoch(epoch_index):
    total_loss = 0.
    for i, data in enumerate(train_data_loader):
        # Every data instance is an input + label pair
        inputs, labels = data
        #print("inputs.shape=", inputs.shape)
        inputs = inputs.to(device)
        labels = labels.to(device)
        # Zero your gradients for every batch!
        optimizer.zero_grad()

        # Make predictions for this batch
        outputs = model(inputs.view(batch_size,-1))

        # Compute the loss and its gradients
        loss = loss_fn(outputs, labels)
        loss.backward()

        # Adjust learning weights
        optimizer.step()

        total_loss += loss.item()

    return total_loss/(len(train_data_loader) * batch_size)
```

Step 5- Training within an epoch

len(train_data_loader)=60,000/4=15,000

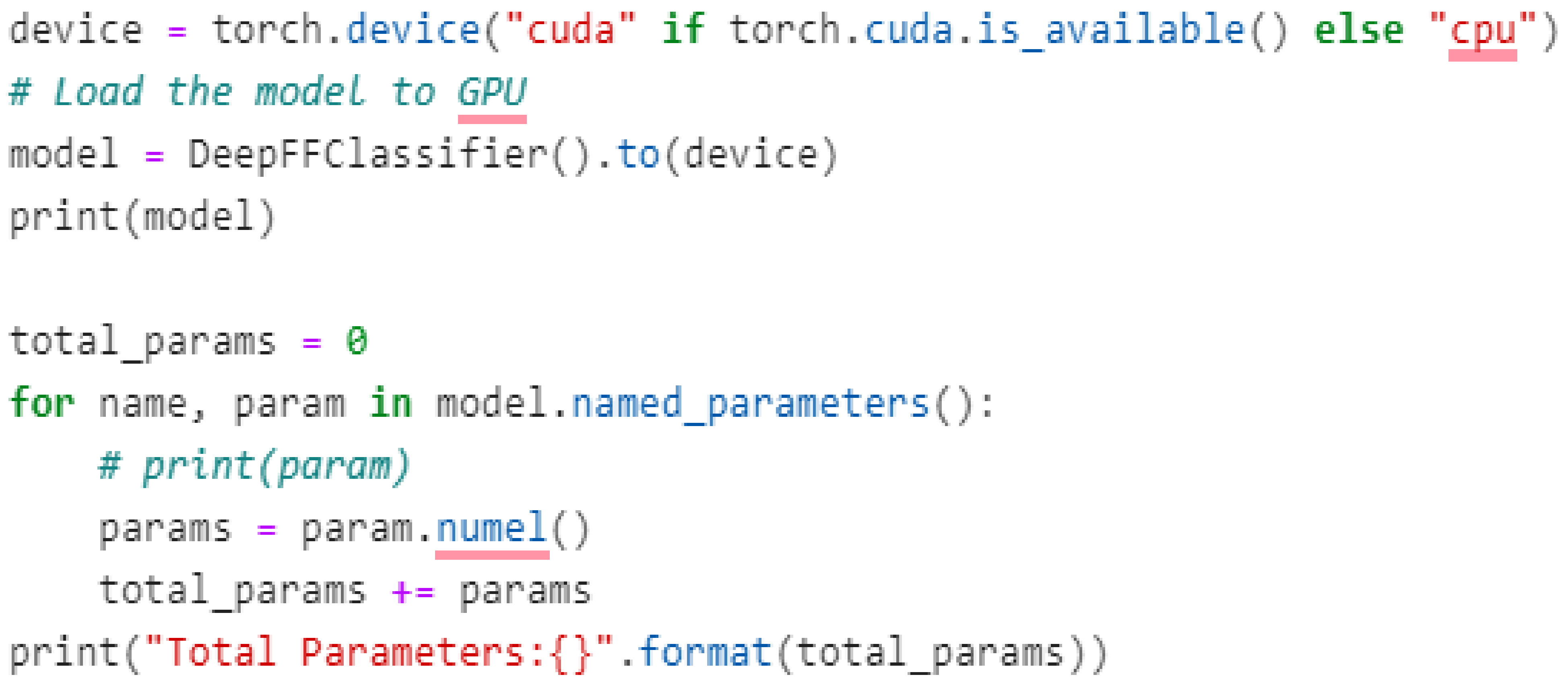
batch_size=4

len(train_data_loader) * batch_size =

15,000*4=60,000

Feedforward neural model for MNIST Digit classification

Step 6 - Find the total number of parameters



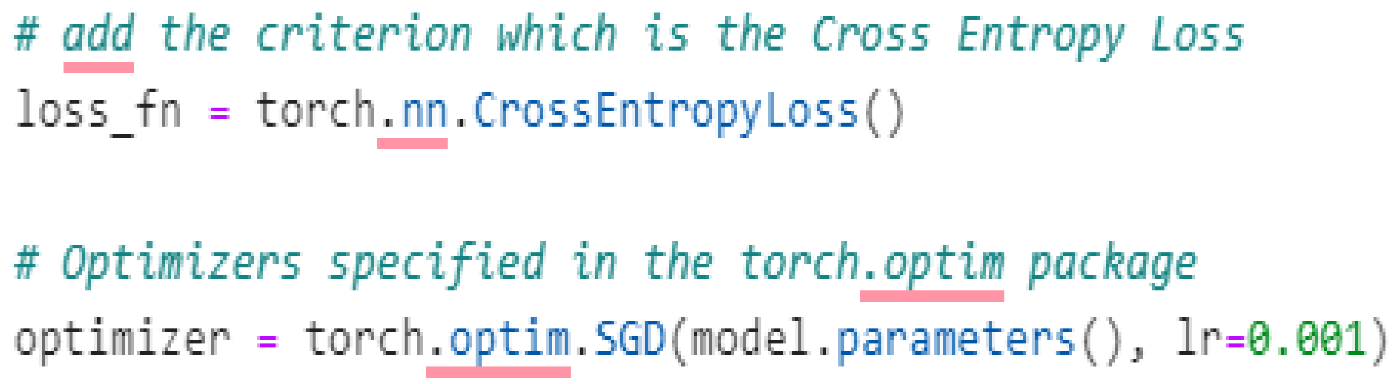
```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# Load the model to GPU
model = DeepFFClassifier().to(device)
print(model)

total_params = 0
for name, param in model.named_parameters():
    # print(param)
    params = param.numel()
    total_params += params
print("Total Parameters:{}".format(total_params))
```

Feedforward neural model for MNIST Digit classification - Output

```
DeepFFClassifier(
  (net): Sequential(
    (0): Linear(in_features=784, out_features=100, bias=True)
    (1): ReLU()
    (2): Linear(in_features=100, out_features=100, bias=True)
    (3): ReLU()
    (4): Linear(in_features=100, out_features=10, bias=True)
  )
)
Total Parameters:89610
```

Feedforward neural model for MNIST Digit classification

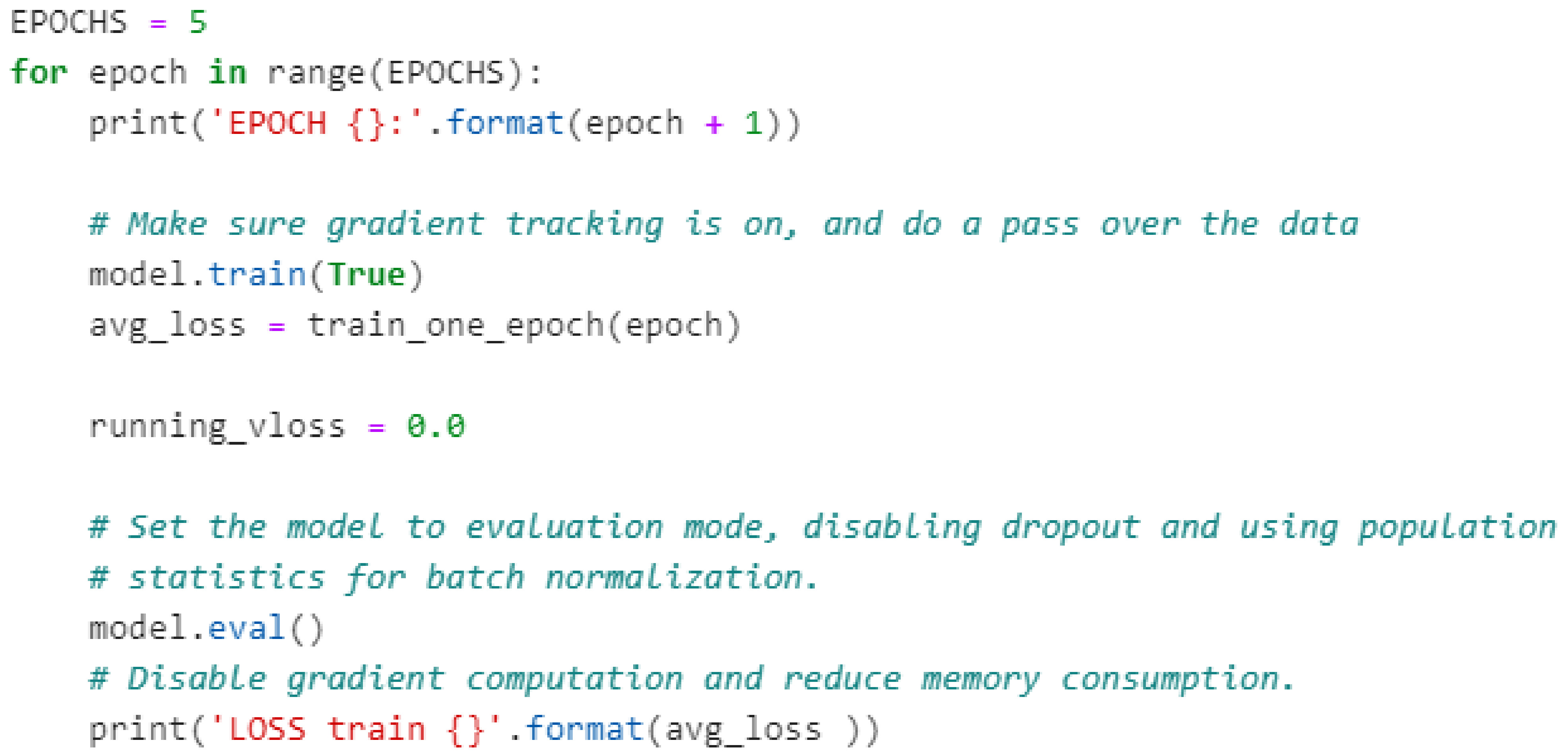
Step 7 – Cross entropy loss as criterion and SGD Optimizer



```
# add the criterion which is the Cross Entropy Loss
loss_fn = torch.nn.CrossEntropyLoss()

# Optimizers specified in the torch.optim package
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
```

Training Loop

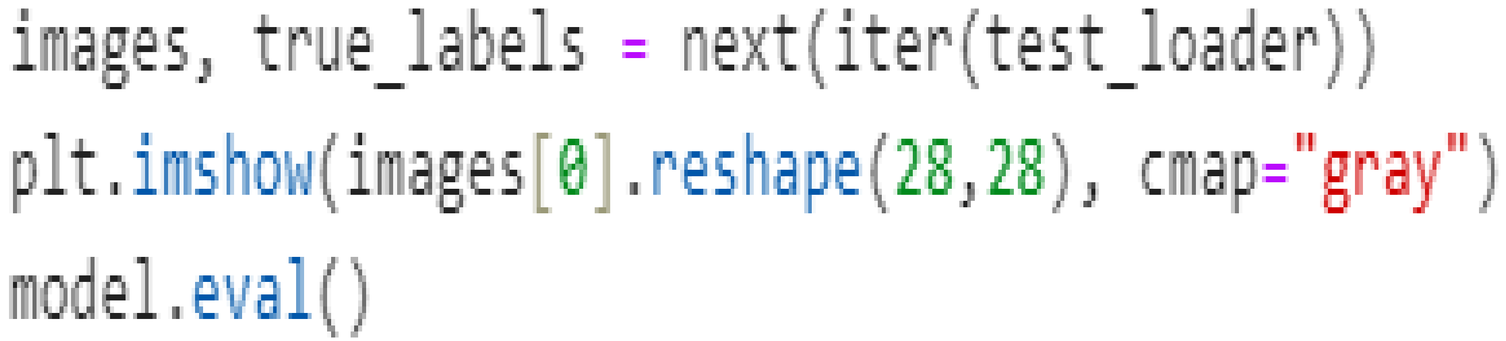
Step 8 – Training loop



```
EPOCHS = 5
for epoch in range(EPOCHS):
    print('EPOCH {}:'.format(epoch + 1))

    # Make sure gradient tracking is on, and do a pass over the data
    model.train(True)
    avg_loss = train_one_epoch(epoch)

    running_vloss = 0.0

    # Set the model to evaluation mode, disabling dropout and using population
    # statistics for batch normalization.
    model.eval()
    # Disable gradient computation and reduce memory consumption.
    print('LOSS train {}'.format(avg_loss ))
```

Display

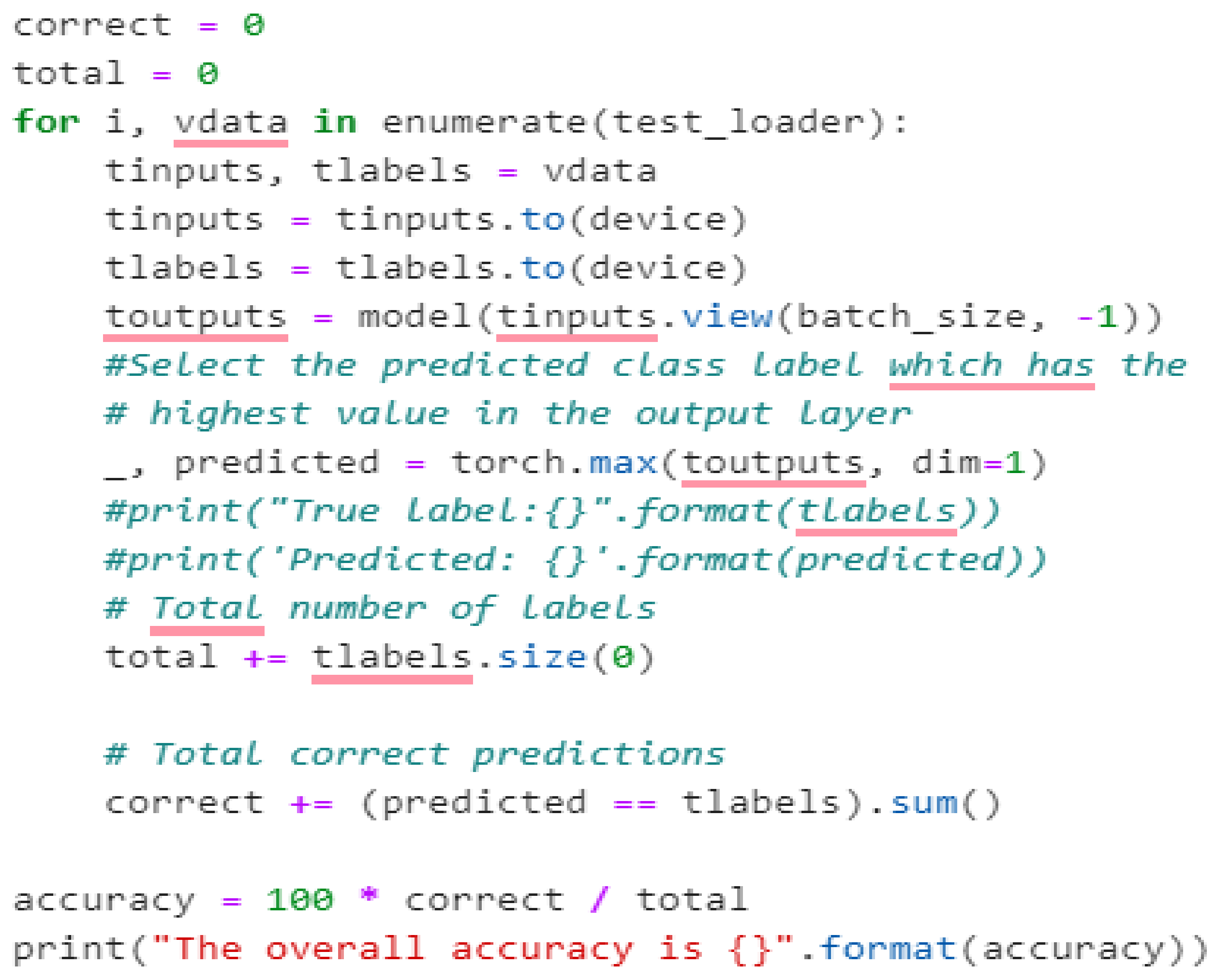
Step 9 – display an image



```
images, true_labels = next(iter(test_loader))
plt.imshow(images[0].reshape(28,28), cmap="gray")
model.eval()
```

Compute Accuracy

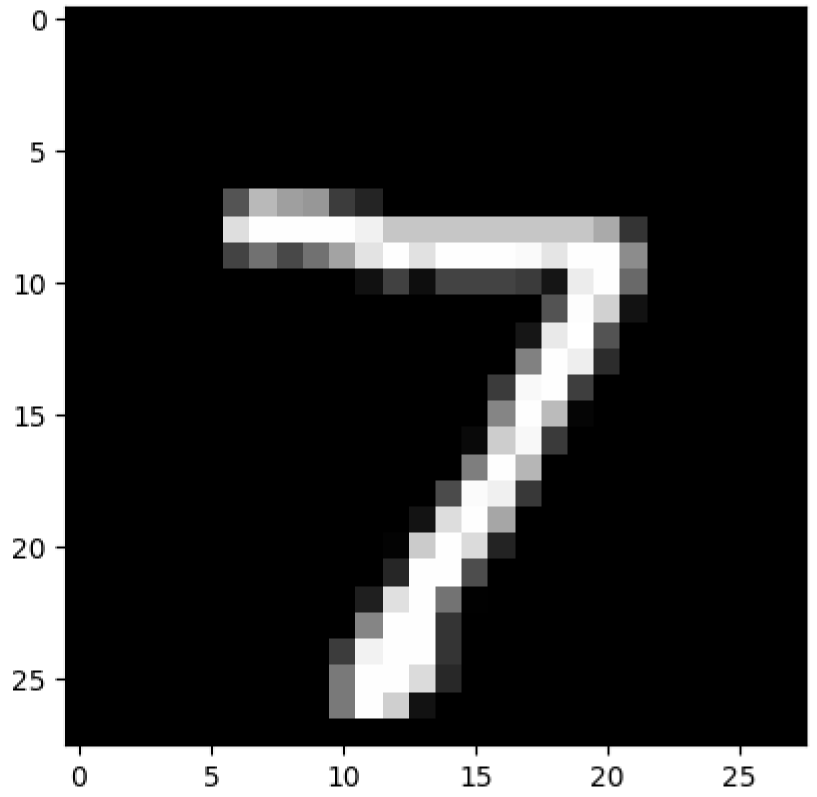
Step 10 – Accuracy



```python
correct = 0
total = 0
for i, vdata in enumerate(test_loader):
    tinputs, tlabels = vdata
    tinputs = tinputs.to(device)
    tlabels = tlabels.to(device)
    toutputs = model(tinputs.view(batch_size, -1))
    #Select the predicted class label which has the
    # highest value in the output layer
    _, predicted = torch.max(toutputs, dim=1)
    #print("True label:{}".format(tlabels))
    #print('Predicted: {}'.format(predicted))
    # Total number of labels
    total += tlabels.size(0)

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

accuracy = 100 * correct / total
print("The overall accuracy is {}".format(accuracy))
```

Output

```
EPOCH 1:
LOSS train 0.3290023316424961
EPOCH 2:
LOSS train 0.10278410078298184
EPOCH 3:
LOSS train 0.08167830266733071
EPOCH 4:
LOSS train 0.07194467245100143
EPOCH 5:
LOSS train 0.06452236264957756
The overall accuracy is 92.8499984741211
```


0
5
10
15
20
25
0
5
10
15
20
25