**Introduction**

Deep Learning is widely spreading and prevailing technology of young Data Science or Artificial Intelligence or Machine Learning enthusiasts. We all get fascinated by the daily research in these sectors. We get motivated when we hear some news in medical field like recognising a particular disease with the help of AI models from a patient report which was very difficult to recognise even by the big specialists of that particular disease. Even there are machines which are capable of performing various complicated Surgical Operations. So, there is a very vast field related to the above contexts. But we will only look at very basic chapter of this entire field. We will learn about recognition of various labels in a particular Image Dataset.

Our topic “Image Dataset Classification” will have concepts of Linear Algebra like Matrix Multiplication, Transpose of a Matrix, etc. We will build a Neural Network with the help of Linear Regression and Logistic Regression which will be capable of predicting the label corresponding a particular image or we can say that model will recognise a particular required object among a set of different objects.

Prerequisites for this Assignment

We are going to write some codes which will include some basic Python and some advanced frameworks of Python like PyTorch and Fast AI and we will also use some advanced modules of Python like Torch and NumPy but we will only see basic concepts of those above-mentioned advanced Modules and Frameworks.

So, overall, what we need to know before starting this Assignment?

The answer is –

1. Basic Programming with Python (data types, loops, functions, etc.)
2. Some Intermediate Mathematics (vectors, matrices, derivatives, probability, etc.)
3. No prior knowledge of data science or deep learning is required.

We will also need to learn some GPU enabled platforms to run our code fast if our pc or laptop is not GPU-enabled. Those platforms include Google Colab, Paperspace Gradient, Kaggle and many more.

Tensors

Let’s explore about Tensors as it is going to be our foundation while we move forward to make our assignment successful. We will also learn various NumPy functions side by side.

PyTorch is a library for processing tensors. A tensor is a number, vector, matrix, or any n-dimensional array.

import torch

import numpy as np

x = torch.tensor(4.)

print(x)

c = np.array([[[2,4,5],[7,4,1]], [[1,3,5], [2,5,4]]])

print(c)

# Now, we will convert given above 3-dimensional array (c) into tensors with help of following code 🡪

d = torch.from\_numpy(c)

print(d)

# Now, we will see the data type of the tensor 🡪

print(x.dtype)

Output 🡪 tensor(4.)

------------------------------

[[[2 4 5]

[7 4 1]]

[[1 3 5]

[2 5 4]]]

------------------------------

tensor([[[2, 4, 5],

[7, 4, 1]],

[[1, 3, 5],

[2, 5, 4]]], dtype=torch.int32)

------------------------------------------------------

torch.float32 # Tensor is of Float data type

Now, we are going to see some other basics of Tensors with help of some Python codes.

Tensors can have any number of dimensions and different lengths along each dimension. We can inspect the length along each dimension using the .shape property of a tensor.

print(x.shape)

print(d.shape)

Output 🡪 torch.Size([])

torch.Size([2, 2, 3])

Tensor Operations and Gradients 🡪

Pytorch has a unique feature known as autograd (automatic gradients). This feature allows us to automatically compute deratives of those tensors whose requires\_grad is set to True with respect to other tensors.

To compute the derivatives, we can invoke the .backward method on our result. Let’s understand autograd function with help of a simple exemplar code.

# We’ll create new Tensors now...

X = torch.tensor(3.)

w = torch.tensor(4. , requires\_grad = True)

b = torch.tensor(5. , requires\_grad = True)

# We are going to perform arithmetic operations

y = w \* x + b

print(y)

# We will now invoke .backward() method to compute derivatives

y.backward()

# Now, derivatives have already been calculated and now we’ll display them on the next page.…

print (‘dy/dx: ’, x.grad)

print (‘dy/dw: ’, w.grad)

print (‘dy/db: ’, b.grad)

# “grad” used above is short form of gradient which means derivative only but is

# generally used when we are dealing with vectors and matrices.

Output 🡪 tensor(17. , grad\_fn=<AddBackward0>)

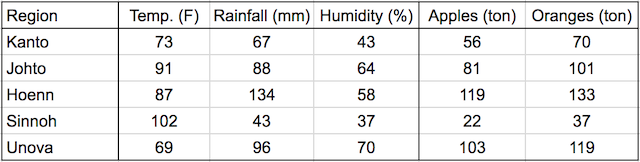
dy/dx: None

dy/dx: tensor(3.)

dy/dx: tensor(1.)

Now, we have learned the basics of Tensors and now we can proceed with Linear and Logistic regression and after that we’ll learn to classify an Image Dataset using built-in Neural Networks in Py-Torch.

We are going to create a model using Linear regression from Scratch that predicts crop yields for apples and oranges (target variables) by looking at the average temperature, rainfall, and humidity (input variables or features) in a region. Here's the training data (a data given to our model so that our model predicts the required outputs(preds) with the help of Training Data).



Now, we are going to assume a linear relationship between the variables to produce some outputs even if those outputs(preds) are wrong, because then we’ll train our model to predict the correct outputs(preds).

So, our assumed relation will be 🡪

yield\_apple = w11 \* temp + w12 \* rainfall + w13 \* humidity + b1

yield\_orange = w21 \* temp + w22 \* rainfall + w23 \* humidity + b2

Here, w11, w12, w13, w21, w22 and w23 are weights and b1 and b2 are biases or in lay-man’s language, these are just random numbers which we will be producing later and then after training our model, we will reach to the approx. weights which will be satisfying the above equations.

Visually, it means that the yield of apples is a linear or planar function of temperature, rainfall and humidity.

Now, we’ll try above learned things by coding the parts in Python.

# Our Training Data will contain inputs and targets # where inputs will be the variables like

# temperature, humidity, etc. while targets

# will include the actual weights and biases values # which will be satisfying the above linear

# relationship among variables.

inputs = np.array([[73, 67, 43],

                   [91, 88, 64],

                   [87, 134, 58],

                   [102, 43, 37],

                   [69, 96, 70]], dtype='float32')

targets = np.array([[56, 70],

                    [81, 101],

                    [119, 133],

                    [22, 37],

                    [103, 119]], dtype='float32')

# Convert inputs and targets to tensors

inputs = torch.from\_numpy(inputs)

targets = torch.from\_numpy(targets)

# print(inputs)

# print(targets)

# Now, we will create a set of weights and biases which will be completely random values with mean 0 and standard deviation equal to 1.

w = torch.randn(2, 3, requires\_grad=True) # 2x3 matrix

b = torch.randn(2, requires\_grad=True)

print(w)

print(b)

Output 🡪 tensor([[-0.4250, 1.3571, 1.4387],

[-0.5925, 0.4800, 0.9822]], requires\_grad=True)

tensor([ 0.2465, -0.3316], requires\_grad=True)

Now, we are actually going to implement a model function which will take inputs as it’s arguments and then will produce outputs which will simply be a matrix multiplication (represented by “@” character in Pytorch) of inputs matrix and transpose (represented by “.t” method in Pytorch) of weights matrix and then respective biases will be added to the resultant matrix.

Loss Function 🡪

Obviously, we are going to get a huge amount of difference between outputs and targets. So, we need to improve our model and we’ll do it with the help of Loss Function.

For that, firstly we will calculate Mean Squared Error (MSE).

Steps for calculating MSE are 🡪

1. Calculate the difference between the two matrices (preds and targets).
2. Square all elements of the difference matrix to remove negative values.
3. Calculate the average of the elements in the resulting matrix.

The result which we will get will be a single number, known as the **Mean Squared Error** (MSE).

We will also be using torch.sum and torch.numel method.

torch.sum returns the sum of all the elements in a tensor. The .numel method of a tensor returns the number of elements in a tensor. Let's compute the mean squared error for the current predictions of our model.

def model(x):

    return x @ w.t() + b

def mse(t1, t2):

    diff = t1 - t2

    return torch.sum(diff \* diff) / diff.numel()

# Generate predictions

preds = model(inputs)

# print(preds)

# Calculating loss

loss = mse(preds, targets)

# print(loss)

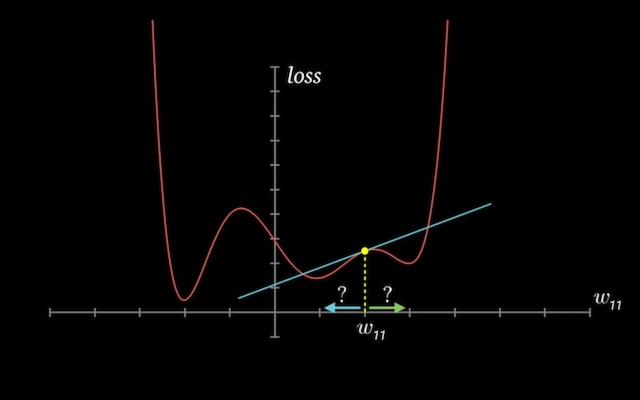
Now, we are going to use “autograd” function which we have already discussed earlier.

Now, after calculating Loss, we’ll have to modify our weights and biases based on some conditions discussed below 🡪

An important insight from calculus is that the gradient indicates the rate of change of the loss, i.e., the loss function's [slope](https://jovian.ai/outlink?url=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FSlope) w.r.t. the weights and biases.

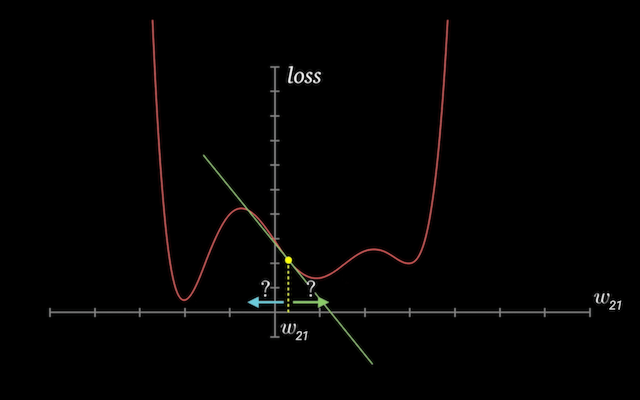
So, if a gradient element is positive:

* **increasing** the weight element's value slightly will **increase** the loss
* **decreasing** the weight element's value slightly will **decrease** the loss



And if a gradient element is negative :

* **increasing** the weight element's value slightly will **decrease** the loss
* **decreasing** the weight element's value slightly will **increase** the loss



After this all, we are going to use torch.no\_grad () method to indicate PyTorch that now it should stop modifying gradients while updating weights and biases.

After using torch.no\_grad () method, we should invoke torch.zero\_grad() method which will reset the gradients values to 0 otherwise the next time when we will invoke torch.no\_grad () method, PyTorch will add new gradients to previously existing gradients.

The main step while training our model will be to multiply the gradients with a very small number (10^-5 in this case) to ensure that we don't modify the weights by a very large amount. By this way, we are trying to reduce variations of the difference between outputs and targets.

# Calculating gradients with respect to loss

loss.backward()

# print(w.grad)

loss = mse(preds, targets)

print(loss)

with torch.no\_grad():

    w -= w.grad \* 1e-5

    b -= b.grad \* 1e-5

# Let's verify that the loss is actually lower

w.grad.zero\_()

b.grad.zero\_()

# print(w.grad)

# print(b.grad)

preds = model(inputs)

# print(preds)

loss = mse(preds, targets)

print(loss)

Output 🡪 tensor(71970.9531, grad\_fn=<DivBackward0>)

tensor(48580.0547, grad\_fn=<DivBackward0>)

# Here we can see the change in value of “loss” before and after torch.no\_grad () function.

Now, we are going to train our model using Gradient Descent whose steps include 🡪

1. Generate predictions
2. Calculate the loss
3. Compute gradients w.r.t the weights and biases
4. Adjust the weights by subtracting a small quantity proportional to the gradient
5. Reset the gradients to zero

We are going to perform all the above steps atleast 500 times(epochs) so that we can reduce our “loss” value and then we will compare between actual targets and outputs(preds).

Now, we can clearly see from above output shown that our model really got trained well and it has improved it’s predictions to a much greater extent and the predictions made by our model is nearly accurate.

for i in range(500):

    preds = model(inputs)

    loss = mse(preds, targets)

    loss.backward()

    with torch.no\_grad():

        w -= w.grad \* 1e-5

        b -= b.grad \* 1e-5

        w.grad.zero\_()

        b.grad.zero\_()

preds = model(inputs)

loss = mse(preds, targets)

print(loss)

print(preds)

print(targets)

Output 🡪 tensor(94.1454, grad\_fn=<DivBackward0>) #Loss

# Predicted output (preds)

tensor([[ 57.1486, 71.6660],

[ 85.3821, 91.4913],

[111.5003, 151.6556],

[ 20.7300, 42.9207],

[107.7661, 99.9191]], grad\_fn=<AddBackward0>)

# Actual Targets (targets)

tensor([[ 56., 70.],

[ 81., 101.],

[119., 133.],

[ 22., 37.],

[103., 119.]])

Let us congratulate ourselves that we have successfully learned

how to implement a Linear Regression model from Scratch which

now we will do with the help of built-in neural networks.

Now, we are going to learn Linear Regression with the help of PyTorch built-ins containing neural networks. We will import the package containing that neural network as nn with the help of Python statement “import torch.nn as nn” .

torch.nn package has a nn.linear class which itself has parameters for weights and biases which will automatically create weights and biases set when we will call that class in our Python program.

We will also be taking help of torch.utils.data class which will help us to have access over TensorDataset and DataLoader where former (TensorDataset) will help us to import a dataset containing inputs and targets (as discussed earlier) made up of tensors and later (DataLoader) will help us to split our dataset into batches of pre-defined size and will also enable us with utilities through which we will be able to shuffle our dataset.

import torch.nn as nn

import torch

from torch.utils.data import TensorDataset

from torch.utils.data import DataLoader

# Input (temp, rainfall, humidity)

inputs = np.array([[73, 67, 43],

                   [91, 88, 64],

                   [87, 134, 58],

                   [102, 43, 37],

                   [69, 96, 70],

                   [74, 66, 43],

                   [91, 87, 65],

                   [88, 134, 59],

                   [101, 44, 37],

                   [68, 96, 71],

                   [73, 66, 44],

                   [92, 87, 64],

                   [87, 135, 57],

                   [103, 43, 36],

                   [68, 97, 70]],

                  dtype='float32')

# Targets (apples, oranges)

targets = np.array([[56, 70],

                    [81, 101],

                    [119, 133],

                    [22, 37],

                    [103, 119],

                    [57, 69],

                    [80, 102],

                    [118, 132],

                    [21, 38],

                    [104, 118],

                    [57, 69],

                    [82, 100],

                    [118, 134],

                    [20, 38],

                    [102, 120]],

                   dtype='float32'

inputs = torch.from\_numpy(inputs)

targets = torch.from\_numpy(targets)

# Define dataset

train\_ds = TensorDataset(inputs, targets)

print(train\_ds[0:3])

# Define data loader

batch\_size = 5

train\_dl = DataLoader(train\_ds, batch\_size, shuffle=True)

for xb, yb in train\_dl:

    print(xb)

    print(yb)

    break

Output 🡪 (tensor([[ 73., 67., 43.],

[ 91., 88., 64.],

[ 87., 134., 58.]]), tensor([[ 56., 70.],

[ 81., 101.],

[119., 133.]]))

tensor([[ 88., 134., 59.],

[ 73., 67., 43.],

[ 91., 88., 64.],

[ 91., 87., 65.],

[ 92., 87., 64.]])

tensor([[118., 132.],

[ 56., 70.],

[ 81., 101.],

[ 80., 102.],

[ 82., 100.]])

Now, we are going to see the use of nn.linear class which automatically creates set of required random weights and biases.

# Define model

model = nn.Linear(3, 2)

print(model.weight)

print(model.bias)

# Parameters

# print(list(model.parameters())) will contain

# weights and biases only……

# Generate predictions

preds = model(inputs)

print(preds)

Output 🡪 Parameter containing: # weights

tensor([[ 0.0642, 0.5140, -0.5604],

[-0.4532, 0.0394, 0.2940]], requires\_grad=True)

Parameter containing: # biases

tensor([0.5216, 0.3917], requires\_grad=True)

# Preds

tensor([[ 15.5493, -17.4125],

[ 15.7308, -18.5694],

[ 42.4801, -16.7092],

[ 8.4374, -33.2646],

[ 15.0682, -6.5199],

[ 15.0995, -17.9051],

[ 14.6565, -18.3148],

[ 41.9839, -16.8684],

[ 8.8872, -32.7720],

[ 14.4436, -5.7727],

[ 14.4750, -17.1579],

[ 15.2810, -19.0620],

[ 43.5544, -16.9638],

[ 9.0620, -34.0118],

[ 15.5180, -6.0273]], grad\_fn=<AddmmBackward>)

Now, we are going to compute Loss and then we will be optimizing our model to put our prediction model on a right track.

Instead of computing gradients and then following steps of Gradient Descent, we will be using SGD (Stochastic Gradient Descent) and to use SGD, we will take help from optim.SGD class.

After having done with optimizing our model, we will repeat same steps of SGD approx. 100 times to train our model and make it efficient and then will compare the deviation between targets and optimised preds.

# Define loss function

loss\_fn = F.mse\_loss

loss = loss\_fn(model(inputs), targets)

# print(loss)

# Define optimizer

opt = torch.optim.SGD(model.parameters(), lr=1e-5)

#Utility function to train the model

def fit(num\_epochs, model, loss\_fn, opt, train\_dl):

    # Repeat for given number of epochs

    for epoch in range(num\_epochs):

        # Train with batches of data

        for xb,yb in train\_dl:

            # 1. Generate predictions

            pred = model(xb)

            # 2. Calculate loss

            loss = loss\_fn(pred, yb)

            # 3. Compute gradients

            loss.backward()

            # 4. Update parameters using gradients

            opt.step()

            # 5. Reset the gradients to zero

            opt.zero\_grad()

# Print the progress

         if (epoch+1) % 10 == 0:

print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1, num\_epochs, loss.item()))

fit(100, model, loss\_fn, opt, train\_dl)

# Generate predictions

preds = model(inputs)

print(preds)

print(targets)

Output 🡪 Epoch [10/100], Loss: 31.3437

Epoch [20/100], Loss: 277.8079

Epoch [30/100], Loss: 242.5903

Epoch [40/100], Loss: 112.5651

Epoch [50/100], Loss: 126.7586

Epoch [60/100], Loss: 88.8866

Epoch [70/100], Loss: 71.2840

Epoch [80/100], Loss: 20.6516

Epoch [90/100], Loss: 28.3818

Epoch [100/100], Loss: 23.4561

tensor([[ 58.6058, 71.2922],

[ 80.0696, 98.9888],

[120.5707, 135.4089],

[ 29.3285, 42.0688],

[ 93.5077, 113.3017],

[ 57.5018, 70.2631],

[ 79.4630, 98.7430],

[120.6626, 135.8467],

[ 30.4325, 43.0978],

[ 94.0050, 114.0849],

[ 57.9992, 71.0463],

[ 78.9656, 97.9598],

[121.1773, 135.6547],

[ 28.8312, 41.2856],

[ 94.6117, 114.3307]], grad\_fn=<AddmmBackward>)

tensor([[ 56., 70.],

[ 81., 101.],

[119., 133.],

[ 22., 37.],

[103., 119.],

[ 57., 69.],

[ 80., 102.],

[118., 132.],

[ 21., 38.],

[104., 118.],

[ 57., 69.],

[ 82., 100.],

[118., 134.],

[ 20., 38.],

[102., 120.]])

Now, we can clearly see from the above output of Python code that after 100 epochs, our model is giving approx. same predictions as compared to our targets.

Now, we should again congratulate ourselves that we have successfully learned how to implement a Linear Regression model with the help of PyTorch built-ins neural network and now we will do all these with the help of Logistic regression model.

Now, our next and final task is to build a model using Logistic Regression and we’ll learn Logistic Regression with the help of a real-world example where we will load a built-in Dataset and the work of the model will be to recognise an integer to be among 0-9 Natural numbers among the dataset which we will be importing in our Python program.

Here in this assignment, we are going to work on MNIST Dataset. We will first download the Dataset if it’s not downloaded in our system.

Actually, this dataset is already downloaded in my system so I’ll not show the output showing its downloading percentage. But I’ll give the Python code to download MNIST Dataset.

We are also going to import torchvision library because this library contains various datasets and it also has a class known as transforms which will directly convert each image into tensor format.

We will also be taking help from Matplotlib library of Python to plot our image dataset while running our Python program.

import torch

import torchvision

from torchvision.datasets import MNIST

# from torchvision import transforms as transforms

import matplotlib.pyplot as plt

import torchvision.transforms as transforms

dataset = MNIST(root='data/', download=True)

print(len(dataset))

# print(dataset[5])

test\_dataset = MNIST(root='data/', train=False)

print(test\_dataset[4])

print(len(test\_dataset))

image, label = dataset[10]

plt.imshow(image, cmap='gray')

print('Label:', label)

# MNIST dataset (images and labels)

# We have to convert images into tensors so to apply torch functions.dataset = MNIST(root='data/',

                train=True,

                transform=transforms.ToTensor())

img\_tensor, label = dataset[0]

print(img\_tensor.shape, label)

## Important ---------------------------------

# We can also print a specific area of a image by -->

print(img\_tensor[0,10:15,10:15])

# print(torch.max(img\_tensor), torch.min(img\_tensor))

# Plot the image by passing in the 28x28 matrix

plt.imshow(img\_tensor[0,10:15,10:15], cmap='gray')

from torch.utils.data import random\_split

train\_ds, val\_ds = random\_split(dataset, [50000, 10000])

print(len(train\_ds))

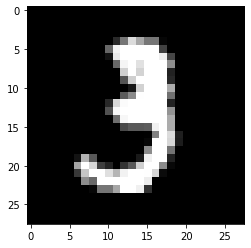
print(len(val\_ds))

-----------------------------------------------

Output 🡪 60000

(<PIL.Image.Image image mode=L size=28x28 at 0x2B3146F9AC0>, 4)

10000



Label: 3

torch.Size([1, 28, 28]) 5

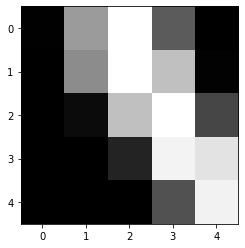
tensor([[0.0039, 0.6039, 0.9922, 0.3529, 0.0000],

[0.0000, 0.5451, 0.9922, 0.7451, 0.0078],

[0.0000, 0.0431, 0.7451, 0.9922, 0.2745],

[0.0000, 0.0000, 0.1373, 0.9451, 0.8824],

[0.0000, 0.0000, 0.0000, 0.3176, 0.9412]])



50000

10000

We can see from above Python code that we have split our data into Training Set and Validation Set. By the way, we will also be having Test Dataset also. Let’s see their importance:

1. **Training set** - used to train the model, i.e., compute the loss and adjust the model's weights using gradient descent.
2. **Validation set** - used to evaluate the model during training, adjust hyperparameters (learning rate, etc.), and pick the best version of the model.
3. **Test set** - used to compare different models or approaches and report the model's final accuracy.

In the MNIST dataset, there are 60,000 training images and 10,000 test images. The test set is standardized so that different researchers can report their models' results against the same collection of images.

Since there's no predefined validation set, we must manually split the 60,000 images into training and validation datasets. Let's set aside 10,000 randomly chosen images for validation. We have done this using the random\_spilt method from PyTorch.

It's essential to choose a random sample for creating a validation set. Training data is often sorted by the target labels, i.e., images of 0s, followed by 1s, followed by 2s, etc. If we create a validation set using the last 20% of images, it would only consist of 8s and 9s. In contrast, the training set would contain no 8s or 9s. Such a training-validation would make it impossible to train a useful model.

We can now create data loaders to help us load the data in batches. We'll use a batch size of 128.

We shuffle the datasets in train\_loader so thet we don't give samesets to loader otherwise

model will learn those sets and

efficiency of model will decrease and also because maybe data is

in order.

And, we don't need this case in val\_loader because that's just used for validation.

from torch.utils.data import DataLoader

batch\_size = 128

train\_loader = DataLoader(train\_ds, batch\_size, shuffle=True)

val\_loader = DataLoader(val\_ds, batch\_size)

Now, we will initialise the model with the help of built-in neural network. As we have already seen when we firstly created neural network with the help of nn.linear class that nn.linear expects each training example to be a vector, therefore, we will flatten each 1x28x28 image tensor into a vector of size 784 (28\*28) before being passed into the model with the help of .reshape method of a tensor, which will allow us to efficiently 'view' each image as a flat vector without really creating a copy of the underlying data. To include this additional functionality within our model, we need to define a custom model by extending the nn.Module class from PyTorch.

The output for each image is a vector of size 10, with each element signifying the probability of a particular target label (i.e., 0 to 9). The predicted label for an image is simply the one with the highest probability.

Now, we will also discuss some code explanation which we are going to write in MnistModel class and passing our neural network (nn.Module) as argument in the class.

And here goes the explanation:

Inside the \_\_init\_\_ constructor method, we instantiate the weights and biases using nn.Linear. And inside the forward method, which is invoked when we pass a batch of inputs to the model, we flatten the input tensor and pass it into self.linear.

So, we need to flatten the shape of the image. We will reshape the

image shape by reshape method by giving two arguments - -1 and 784.

-1 is given because torch is itself capable of calculating

the first argument. Second argument is the final reshaped shape of

image. We will also inherit properties of nn.Module and create

 a new model with some changes and name it as MnistModel.

import torch.nn as nn

input\_size = 28\*28

num\_classes = 10

# Logistic regression model

model = nn.Linear(input\_size, num\_classes)

print(model.weight.shape)

# print(model.weight)

print(model.bias.shape)

# print(model.bias)

class MnistModel(nn.Module):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.linear = nn.Linear(input\_size, num\_classes)

    def forward(self, xb):

        xb = xb.reshape(-1, 784)

        out = self.linear(xb)

        return out

model = MnistModel()

print(model.linear)

Output 🡪 torch.Size([10, 784])

torch.Size([10])

Linear(in\_features=784, out\_features=10, bias=True)

Now we should test our new Custom model….

for images, labels in train\_loader:

    print(images.shape)

    outputs = model(images)

    break

print('outputs.shape : ', outputs.shape)

print('Sample outputs :\n', outputs[:2].data)

Output 🡪 torch.Size([128, 1, 28, 28])

outputs.shape : torch.Size([128, 10])

Sample outputs :

tensor([[-0.3273, 0.3076, -0.1238, 0.1671, -0.0572, -0.2381, -0.0514, 0.1688,0.2077, 0.1218],

[-0.2530, 0.2181, -0.3318, 0.2400, 0.1459, 0.2250, -0.5609, 0.1265,0.2693, 0.1216]])

Now, the outputs we are getting by running the above code are just probabilities.

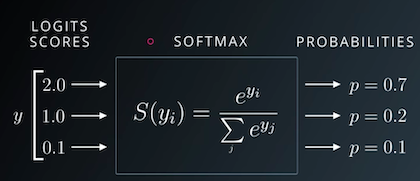
We get the probabilities on basis of random weights and biases so -ve prob also is shown. Now, we will use inbuilt Softmax function of

torch.nn.functional library which will at first convert all numbers

to positive numbers and raise each class (out of 10) to power of e and

will calculate probability of each e^i with summation of all e^i's. Purposeof softmax function is to raise the most probable to it's max.

We have attached a picture just below this para which will help us to understand more clearly about Softmax Function.



As we know that we are talking about probabilities of a given label out of (0-9), therefore we should check once whether the sum of all probabilities of all labels is 1 or approx. 1 for a given image. sSo, we will also try that out...

import torch.nn.functional as F

# Apply softmax for each output row

probs = F.softmax(outputs, dim=1)

# Look at sample probabilities

print("Sample probabilities:\n", probs[:2].data)

# Add up the probabilities of an output row

print("Sum: ", torch.sum(probs[0]).item())

Output 🡪 Sample probabilities:

tensor([[0.0965, 0.0791, 0.1004, 0.0968, 0.1097, 0.0864, 0.1021, 0.1447, 0.0988,

0.0854],

[0.0863, 0.0895, 0.1193, 0.1166, 0.0878, 0.1015, 0.0986, 0.1202, 0.0729,

0.1074]])

Sum: 1.0

Now, we are going to check working of our model.

Mam, after checking the working of our model, we got output as probabilities (most probable label) corresponding to each image and our output is for a batch of size 128. So, our output was very big, that’s why we are not going to show our output corresponding to following code.

The code for above stated output is 🡪

max\_probs, preds = torch.max(probs, dim=1)

print(preds)

print(max\_probs)

print(labels)

We will not get correct predictions because of random weights and

biases but now we will optimize our code...

But before optimizing our code, we’ll first calculate accuracy of our model to keep a track on efficiency of our model.

We can not take loss as our accuracy because :

1. Correct probabilities are not there.

2. torch.sum and == are not mathematical functions and they are neither continuous and differentiable.

PyTorch has in-built Cross-Entropy function which we will use to calculate accuracy.

The cross-entropy is the negative logarithm of the predicted probability of the correct label averaged over all training samples. Therefore, one way to interpret the resulting number e.g. 2.23 is look at e^-2.23 which is around 0.1 as the predicted probability of the correct label, on average. The lower the loss, The better the model. 😎

We will calculate cross\_entropy which will convert tensors in known to me format and multiply tensor to labels(correct) and then we will

calculate sum of it and then calculate accuracy.

loss\_fn = F.cross\_entropy

# Loss for current batch of data

loss = loss\_fn(outputs, labels)

print(loss)

Output 🡪 tensor(2.3302, grad\_fn=<NllLossBackward>)

Now, we will train our model and make it efficient.

So, while training our model has to pass through two phases – Training Phase and Validation Phase, where all previous methods which we had applied during Linear Regression using Neural Networks, are to be applied.

So, let’s start training our model…

def fit(epochs, lr, model, train\_loader, val\_loader,

opt\_func=torch.optim.SGD):

    optimizer = opt\_func(model.parameters(), lr)

    history = [] # for recording epoch-wise results

    for epoch in range(epochs):

        # Training Phase

        for batch in train\_loader:

            loss = model.training\_step(batch)

            loss.backward()

            optimizer.step()

            optimizer.zero\_grad()

        # Validation phase

        result = evaluate(model, val\_loader)

        model.epoch\_end(epoch, result)

        history.append(result)

    return history

def evaluate(model, val\_loader):

    outputs = [model.validation\_step(batch) for batch in val\_loader]

    return model.validation\_epoch\_end(outputs)

Our main function (fit) which will actually train our model is the one which we have just shown above.

Now, we’ll have to modify our model which will work compatibly with the above main (fit) function.

# Modifying our model -----

class MnistModel(nn.Module):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.linear = nn.Linear(input\_size, num\_classes)

    def forward(self, xb):

        xb = xb.reshape(-1, 784)

        out = self.linear(xb)

        return out

    def training\_step(self, batch):

        images, labels = batch

        out = self(images)      # Generate predictions

        loss = F.cross\_entropy(out, labels) # Calculate loss

        return loss

    def validation\_step(self, batch):

        images, labels = batch

        out = self(images)      # Generate predictions

        loss = F.cross\_entropy(out, labels)# Calculate loss

        acc = accuracy(out, labels)# Calculate accuracy

        return {'val\_loss': loss, 'val\_acc': acc}

def validation\_epoch\_end(self, outputs):

        batch\_losses = [x['val\_loss'] for x in outputs]

        epoch\_loss = torch.stack(batch\_losses).mean()

# Combine losses

        batch\_accs = [x['val\_acc'] for x in outputs]

        epoch\_acc = torch.stack(batch\_accs).mean()

# Combine accuracies

        return {'val\_loss': epoch\_loss.item(), 'val\_acc': epoch\_acc.item()}

    def epoch\_end(self, epoch, result):

        print("Epoch [{}], val\_loss: {:.4f}, val\_acc: {:.4f}".format(epoch, result['val\_loss'], result['val\_acc']))

In the output of code shown below, we are not going to show the print statement output of validation\_epoch\_end function because the output is very large.

model = MnistModel()

result0 = evaluate(model, val\_loader)

print(result0)

history1 = fit(5, 0.001, model, train\_loader, val\_loader)

print(history1)

history2 = fit(5, 0.001, model, train\_loader, val\_loader)

print(history2)

history3 = fit(5, 0.001, model, train\_loader, val\_loader)

print(history3)

history4 = fit(5, 0.001, model, train\_loader, val\_loader)

print(history4)

Output 🡪 {'val\_loss': 2.3245911598205566, 'val\_acc': 0.08742088824510574}

Epoch [0], val\_loss: 1.9594, val\_acc: 0.5935

Epoch [1], val\_loss: 1.6898, val\_acc: 0.7129

Epoch [2], val\_loss: 1.4878, val\_acc: 0.7589

Epoch [3], val\_loss: 1.3350, val\_acc: 0.7838

Epoch [4], val\_loss: 1.2172, val\_acc: 0.7956

# Other outputs are on next page…

Epoch [0], val\_loss: 1.1246, val\_acc: 0.8024

Epoch [1], val\_loss: 1.0502, val\_acc: 0.8108

Epoch [2], val\_loss: 0.9894, val\_acc: 0.8170

Epoch [3], val\_loss: 0.9387, val\_acc: 0.8210

Epoch [4], val\_loss: 0.8959, val\_acc: 0.8259

Epoch [0], val\_loss: 0.8592, val\_acc: 0.8298

Epoch [1], val\_loss: 0.8275, val\_acc: 0.8330

Epoch [2], val\_loss: 0.7997, val\_acc: 0.8354

Epoch [3], val\_loss: 0.7752, val\_acc: 0.8392

Epoch [4], val\_loss: 0.7533, val\_acc: 0.8417

Epoch [0], val\_loss: 0.7338, val\_acc: 0.8447

Epoch [1], val\_loss: 0.7161, val\_acc: 0.8468

Epoch [2], val\_loss: 0.7001, val\_acc: 0.8485

Epoch [3], val\_loss: 0.6855, val\_acc: 0.8509

Epoch [4], val\_loss: 0.6721, val\_acc: 0.8529

**Obtained accuracy of our model is approx. 85% which is a nice accuracy.**

We must verify our predictions of our model with at least 1 image of dataset. So, let’s verify that…

# Define test dataset

test\_dataset = MNIST(root='data/',

                     train=False,

                     transform=transforms.ToTensor())

def predict\_image(img, model):

    xb = img.unsqueeze(0)

    yb = model(xb)

    \_, preds = torch.max(yb, dim=1)

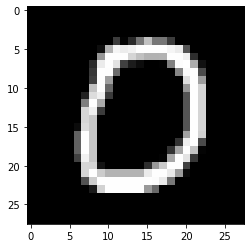
    return preds[0].item()

img, label = test\_dataset[0]

plt.imshow(img[0], cmap='gray')

print('Label:', label, ', Predicted:', predict\_image(img, model))

Output 🡪



Label: 0 , Predicted: 0

We have verified from above output that our model is working fine and now, we must end this assignment because it has already become too large.

We must feel happy now that after learning such a huge model of Logistic Regression, we have successfully completed our task of working with Linear and Logistic Regression from Scratch and even with the help of Neural Networks.

**Our special thanks to Aakash N S (Founder at Jovian), because of his tutorials and documentation, we have completed our learning and assignment.**

**We have taken help from his tutorials but it’s not that we have copied and pasted his documentation. We have carefully read his documentation and then completed our assignment. We all were learning these concepts before announcement of this assignment so it was easy for us to complete this very big assignment in this short period of time.**

**Obviously, this assignment has become very large but our main aim was to learn few basic concepts of Deep Learning and we all are happy that we have successfully gone through each and every concept of Linear and Logistic Regression and learned all of them through good exemplar codes and their outputs and we have verified our models also.**