**PYTORCH**

Pytorch is an open source machine learning library based on Torch library, used for application such as computer vision and NLP.

Facebook AI Research Lab.

Pytorch is

* Fast
* Flexible
* Seamless

It’s a Python based scientific computing package targeted at two set of audiences.

1. A replacement for Numpy to use the power of GPUs.
2. A deep learning research platform that provides maximum flexibility and speed.
3. **Tensors:**

Tensors are similar to Numpy’s ndarrays, with the addition being that Tensor can be used on a GPU to accelerate computing.

Can be used using – **import torch**

Example:

**Construct a 5by3 matrix, uninitialized:**

X =torch.empty(5,3)

Print(X)

Output:

tensor([[-4.7175e-25, 4.5740e-41, -4.7454e-25],

[ 4.5740e-41, -4.7596e-25, 4.5740e-41],

[-4.7596e-25, 4.5740e-41, 0.0000e+00],

[ 0.0000e+00, 0.0000e+00, 0.0000e+00],

[ 0.0000e+00, 0.0000e+00, 1.7403e-26]])

**Construct a randomly initialized matrix:**

X=torch.rand(5,3)

Print(X)

Output:

tensor([[0.8597, 0.4481, 0.7624],

[0.2173, 0.8450, 0.1942],

[0.0912, 0.8004, 0.0755],

[0.3035, 0.9528, 0.6915],

[0.3343, 0.2916, 0.7734]])

**Contruct a matrix filled zeros and of dtype long:**

X = torch.zeros(5,3,dtpe=torch.long)

Print(X)

Output:

tensor([[0, 0, 0],

[0, 0, 0],

[0, 0, 0],

[0, 0, 0],

[0, 0, 0]])

**Construct a tensor directly from data:**

X = torch.tensor([5.5,3])

Print(X)

Output:

tensor([5.5000, 3.0000])

**Create a tensor based on existing tensor:**

X = X.new\_ones(5,3,dtype=torch.double)

Print(X)

X= X.randn\_like(X,dtype=torch.float)

Print(X)

Output:

tensor([[1., 1., 1.],

[1., 1., 1.],

[1., 1., 1.],

[1., 1., 1.],

[1., 1., 1.]], dtype=torch.float64)

tensor([[-0.9048, -0.2639, -0.3928],

[-1.4317, -0.1446, -0.2875],

[ 0.3691, -0.5043, 0.0791],

[-2.1418, -0.3781, -0.5488],

[ 0.9542, 0.1281, -1.6653]])

**Get its size:**

Print(x.size())

Output:

torch.Size([5, 3]**)**

1. **Operations:**

X = torch.rand(5,3)

Y = torch.rand(5,3)

print(X+Y)

print(torch.add(X,Y))

Z = torch.empty(5,3)

torch.add(X,Y,out=Z)

print(Z)

Y.add\_(X)

print(Y)

Output:

tensor([[0.7082, 0.8615, 1.5948], [0.9886, 1.1455, 0.9631], [0.9322, 1.1506, 0.7727], [0.2706, 1.3203, 0.6880], [1.4459, 1.0704, 1.2248]])

tensor([[0.7082, 0.8615, 1.5948], [0.9886, 1.1455, 0.9631], [0.9322, 1.1506, 0.7727], [0.2706, 1.3203, 0.6880], [1.4459, 1.0704, 1.2248]])

tensor([[0.7082, 0.8615, 1.5948], [0.9886, 1.1455, 0.9631], [0.9322, 1.1506, 0.7727], [0.2706, 1.3203, 0.6880], [1.4459, 1.0704, 1.2248]])

tensor([[0.7082, 0.8615, 1.5948], [0.9886, 1.1455, 0.9631], [0.9322, 1.1506, 0.7727], [0.2706, 1.3203, 0.6880], [1.4459, 1.0704, 1.2248]])

NOTE

Any operation that mutates a tensor in-place is post-fixed with an \_. For example: x.copy\_(y), x.t\_(), will change x.

[We can also use standard Numpy-like indexing with all bells and whistles]

* For Resizing the tensors in Pytorch we have to use:

**torch.view**

X = torch.randn(4,4)

Y = X.view(16)

Z = X.view(-1,8) **# the size -1 is inferred from other dimesions**

print(X.size(),Y.size(),Z.size())

Output:

torch.Size([4, 4]) torch.Size([16]) torch.Size([2, 8])

* If you have one element tensor, use **.item()** to get value as a Python number

**x = torch.randn(1)**

**print(x)**

**print(x.item())**

Output:

tensor([-0.8269])

-0.8268657922744751

1. **NumPy Bridge**

Converting a Torch Tensor to a NumPy array and vice versa is a breeze.

The Torch Tensor and NumPy array will share their underlying memory locations (if the Torch Tensor is on CPU), and changing one will change the other.

* **Converting a Torch Tensor to a Numpy Array**

a = torch.ones(5)

print(a)

Output:

tensor([1., 1., 1., 1., 1.])

b = a.numpy()

print(b)

Output:

[1. 1. 1. 1. 1.]

a.add\_(1)

print(a)

print(b)

Output:

tensor([2., 2., 2., 2., 2.])

[2. 2. 2. 2. 2.]

* Converting NumPy array to Torch Tensor

import numpy as np

a = np.ones(5)

b = torch.from\_numpy(a)

np.add(a,1,out=a)

print(a)

print(b)

Output:

[2. 2. 2. 2. 2.]

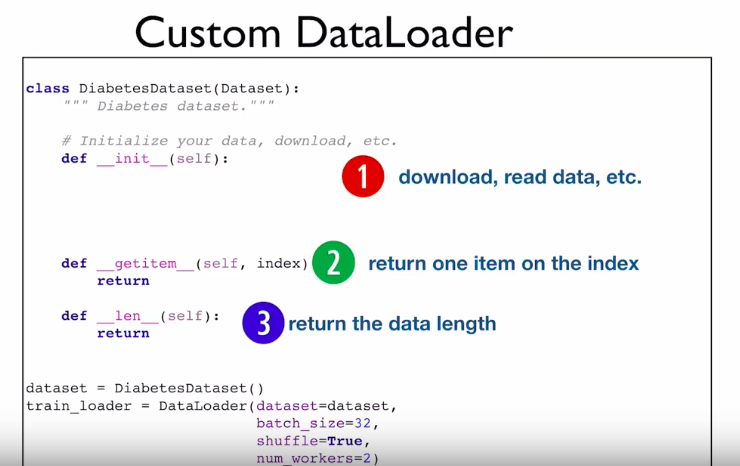
tensor([2., 2., 2., 2., 2.], dtype=torch.float64)

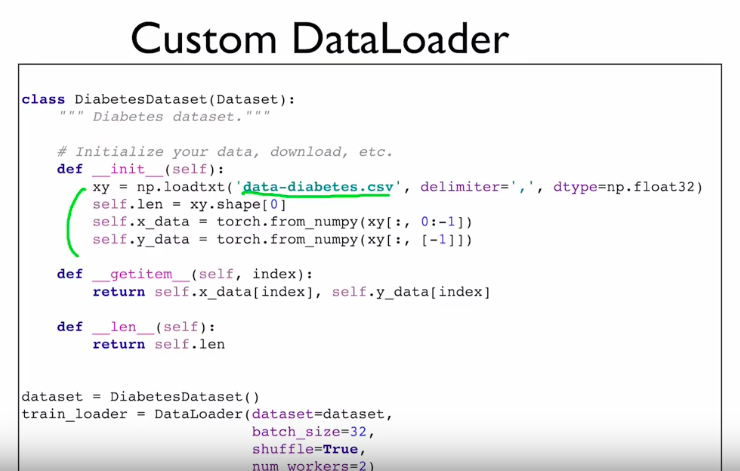
**Data Loading in PyTorch**

DataLoader function that comes in the torch.utils.data.DataLoader

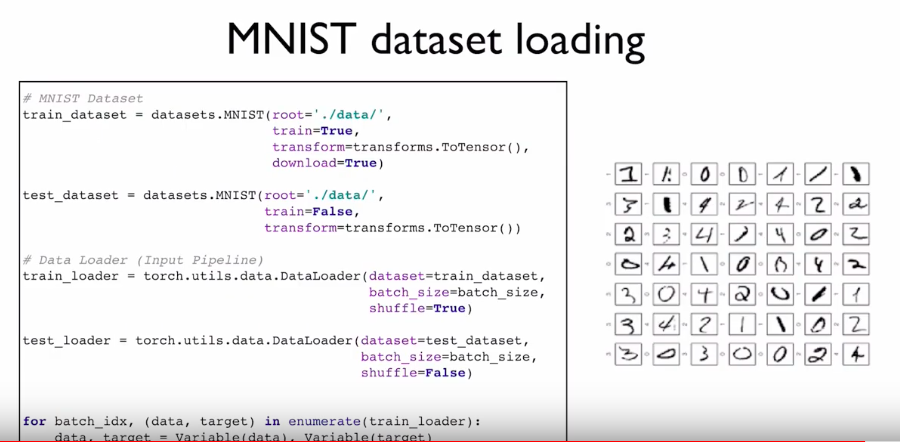
of PyTorch is used to load the data and process it by shuffling it running it using parallel processing and assigning a batch size.

* For loading data using the locally stored .csv files





* For loading MNIST data that comes with Pytorch library.



After loading check the data how it looks and is stored.

Check whether data is balanced

**Initializing Network**

For initializing n/w using PyTorch, there 2 most important imports that need to be made:

1. Import torch.nn as nn –

torch.nn gives us access to some helpful neural network things, such as neural network layer types (things like regular fully connected layers, convolutional layers(for imagery), recurrent layers…etc.)

1. Import torch.nn.functional as F

The torch.nn.functional gives us access to some handy function that we might not want to write ourselves.

Like relu activation function for our neurons.

Instead of writing all of the code for these things, we can just import them, since these are things everyone will be needing in their deep learning code.

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

import torch.nn as nn

import torch.nn.functional as F

## to make our model we are going to create a new class > this class will inherit nn.module class

## nn.module need to be inherited everytime we want to create a network

## It is a base class for all neural network modules

class Net(nn.Module):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        ## In Each nn.Linear layers > 1st param = input size | 2nd param = output size

        ##Input Layer

        self.fc1 = nn.Linear(28\*28,64) ## Input Layer expects an image of size 28 \* 28 = 784 - nn expects                                               ## flattened Tensor i.e. 1 \* 784

        self.fc2 = nn.Linear(64,64)    ## 64 is the number of neurons in the layer. it can be any number

        self.fc3 = nn.Linear(64,64)    ## But we have taken 64 in this case.

        ## Output Layer

        self.fc4 = nn.Linear(64,10) ## Output Layer represents 10 numbers which we have to predict. So, 10                                           ## neurons for 10 classes.

## activation is required to scale the output of the partiular neuron either between 0 or 1 / or betwen 0 or output itself

## Activation functions are mathematical equations that determine the output of a neural network.

##Function is attached to each neuron in the network, and determines whether it should be activated (“fired”) or not, based on whether each neuron’s input is relevantfor the model’s prediction.

    def forward(self, x):

        x = F.relu(self.fc1(x))

        x = F.relu(self.fc2(x))

        x = F.relu(self.fc3(x))

        x = self.fc4(x)

        ## As a output in multiclass classification we need probability distribution of each output

        ## For that we use softmax as it converts output into probability.

        return F.log\_softmax(x,dim=1) ## dim=1 is simlar to axes - as we want 1d array of output

net = Net()

print(net)

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

curl -H "Authorization: Bearer b93a5915ebfb46c18754c3fa446059ed” "https://api.api.ai/v1/query?v=20200127&e=WELCOMEWELCOME&timezone=Europe/Paris&lang=en&sessionId=1234567890"

curl -H "Authorization: Bearer b93a5915ebfb46c18754c3fa446059ed"

"https://api.api.ai/v1/query?v=20200127&e=WELCOM&timezone=Europe/Paris

&lang=en&sessionId=1234567890"

Hi.

I am Sci Of Relief's virtual agent. To get started, simply ask me a question. If you wish to start questionnaire. Just say "Start/Begin Questionnaire!!"

Hi! I'm Sci Of Relief's virtual agent. To get started, simply ask me a question. If you wish to start questionnaire. Just say "Start/Begin Questionnaire!!"

**Training Network**

1. **Loss Function –**

Types of loss

1. **L1Loss:**

Creates a criterion that measures the mean absolute error (MAE) between each element in the input x and target y.

The unreduced (i.e. with reduction set to 'none') loss can be described as: ℓ(*x*,*y*)=*L*={*l*1​,…,*lN*​}^⊤, *ln*​=∣*xn*​−*yn*​∣,

where *N* is the batch size. If reduction is not 'none' (default 'mean'), then: ℓ(*x*,*y*)={mean(*L*), if reduction=’mean’; sum(*L*),​ if reduction=’sum’.​

1. **MSELoss:**

Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input *x* and target *y* .

The unreduced (i.e. with reduction set to 'none') loss can be described as: ℓ(*x*,*y*)=*L*={*l*1​,…,*lN*​}^⊤, *ln*​=∣*xn*​−*yn*​∣^2,

where *N* is the batch size. If reduction is not 'none' (default 'mean'), then: ℓ(*x*,*y*)={mean(*L*), if reduction=’mean’; sum(*L*),​ if reduction=’sum’.​

1. **CrossEntropyLoss**
2. **CTCLoss**
3. **NLLLoss**
4. **PoissonNLLLoss**
5. **KLDivLoss**
6. **BCELoss**
7. **BCEWithLogitsLoss**
8. **MarginRankingLoss**
9. **HingeEmbeddingLoss**
10. **MultiLabelMarginLoss**
11. **SmoothL1Loss**
12. **SoftMarginLoss**
13. **MultiLabelSoftMarginLoss**
14. **CosineEmbeddingLoss**
15. **MultiMarginLoss**
16. **TripletMarginLoss**

loss\_function = nn.CrossEntropyLoss()

1. **Optimizer –**

During training we need an optimizer like gradient descent or stochastic gradient descent which helps us to get the changes in weights and biases so that the cost of our network decreases and accuracy increases.

In Pytorch, the optimizer is present in “torch.optim” class.

For importing it use.,

**import torch.optim as optim**

optimizer = optim.Adam(net.parameters(), lr=0.001)

## lr =  learning rate (alpha) for gradient to come at the local minima

## Parameters - A kind of Tensor that is to be considered a module parameter.

## Parameters are Tensor subclasses, that have a very special property when used with Module s - when they’re assigned as Module attributes they are automatically added to the list of its parameters, and will appear e.g. in **parameters()** iterator.