Introduction to Large Language Models (LLMs)

Prof. Tanmoy Chakraborty, Prof. Soumen Chakraborti

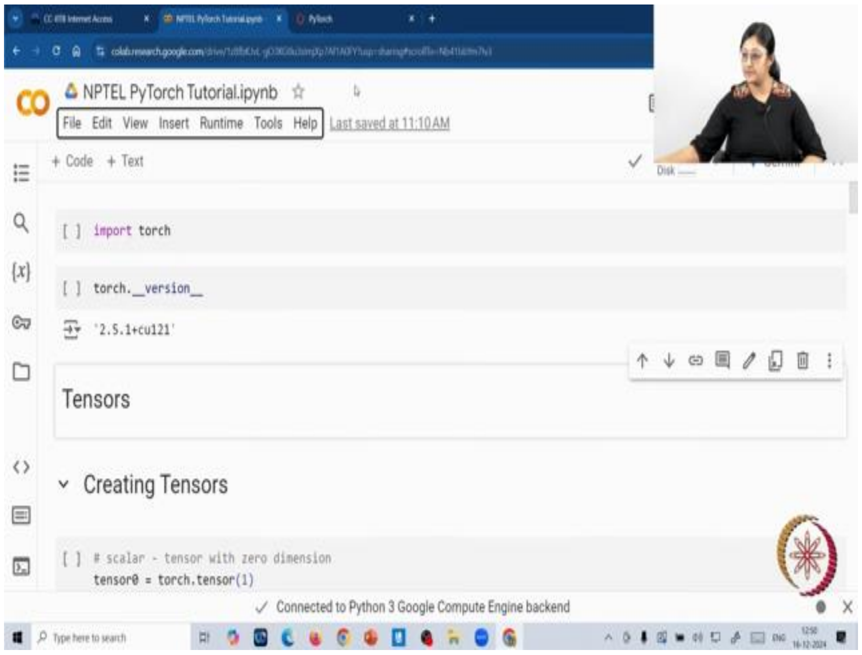
Department of Computer Science & Engineering

Indian Institute of Technology, Delhi

Lecture 6

Introduction to PyTorch

Hello everyone, welcome to the tutorial session. Today we will be covering the fundamental concepts of PyTorch. So, PyTorch is an open-source deep learning framework that allows us to build and train deep learning models. So today, we will first start with understanding the key components of PyTorch which are tensors, autograd and once we are done with that then we will discuss what is the typical PyTorch pipeline that we should follow in order to build a deep learning framework and to train it. And we'll do so by implementing a very basic neural network.


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help
Last saved at 11:10 AM
+ Code + Text
Connected to Python 3 Google Compute Engine backend

```
import torch
```

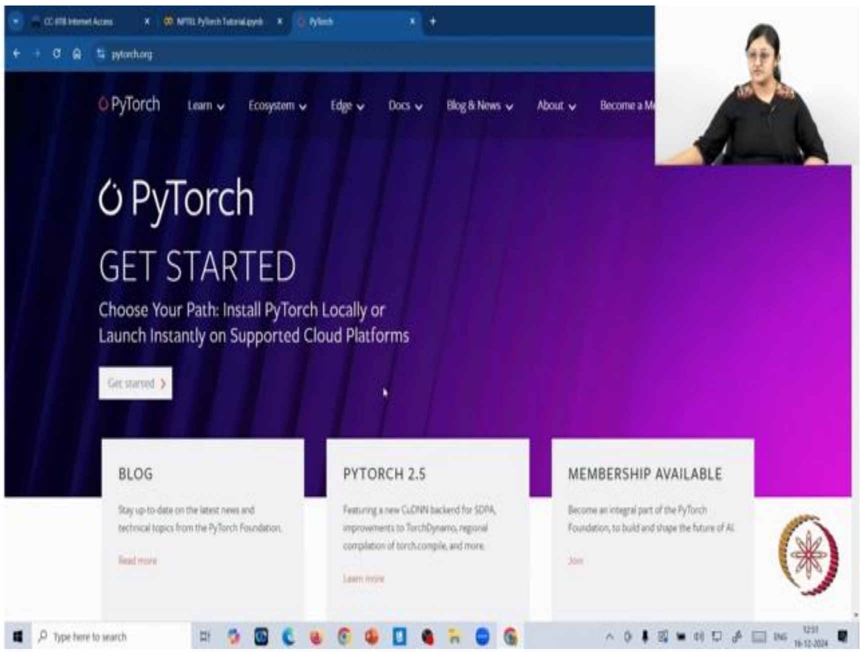
```
torch.__version__
```

```
'2.5.1+cu121'
```

Tensors

Creating Tensors

```
# scalar - tensor with zero dimension
tensor0 = torch.tensor(1)
```



PyTorch

GET STARTED

Choose Your Path: Install PyTorch Locally or Launch Instantly on Supported Cloud Platforms

BLOG

Stay up-to-date on the latest news and technical topics from the PyTorch Foundation.

PYTORCH 2.5

Featuring a new CuDNN backend for SDPA, improvements to TorchDynamo, regional compilation of torch.compile, and more.

MEMBERSHIP AVAILABLE

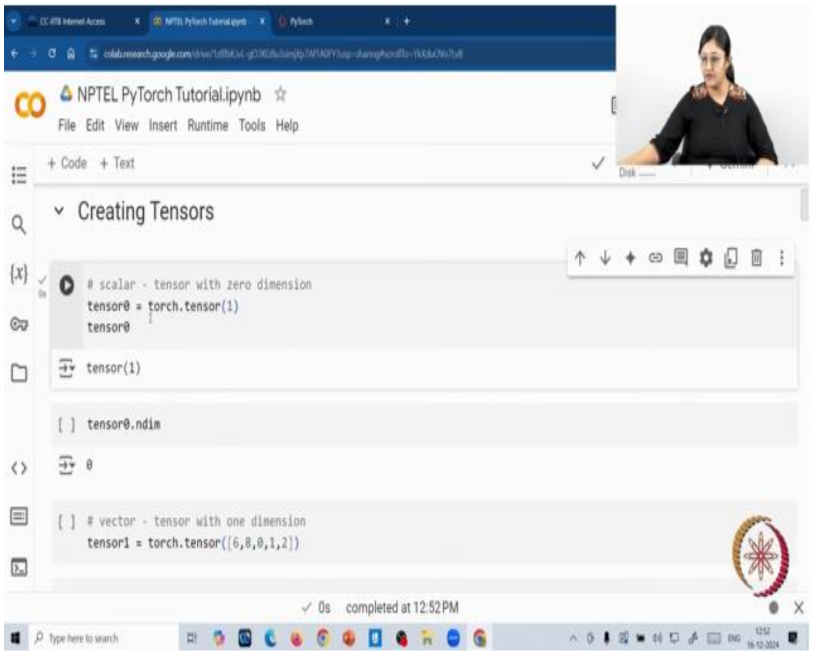
Become an integral part of the PyTorch Foundation, to build and shape the future of AI.

One more general note that I would like to state at the beginning is it's always a good practice to visit the PyTorch documentation.

So PyTorch is always evolving. It's an evolving open source deep learning library. So it's a good practice to keep yourself updated with whatever is happening. And with that, having said that, let's get started with the tutorial.

So one good thing about Colab is the PyTorch library is already pre-installed in Colab. So we can simply start by importing Torch. Once we have done that, let's quickly check the version of PyTorch. So currently when I'm recording this session, the PyTorch version is 2.5.1. But as I mentioned, PyTorch keeps on evolving. So maybe when you are watching this tutorial session, there might be a different, more updated version of PyTorch. And maybe you can encounter some errors. There could be some changes in the functionalities and utilities that PyTorch is following in that version.

So again, kindly refer to the PyTorch documentation anytime you feel stuck with anything. Now, tensors are the backbone of PyTorch. It allows us to do anything and everything, almost anything and everything in PyTorch. So we'll first get started by looking at how to create tensors.


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help
+ Code + Text
Creating Tensors
✓ 0s completed at 12:52 PM

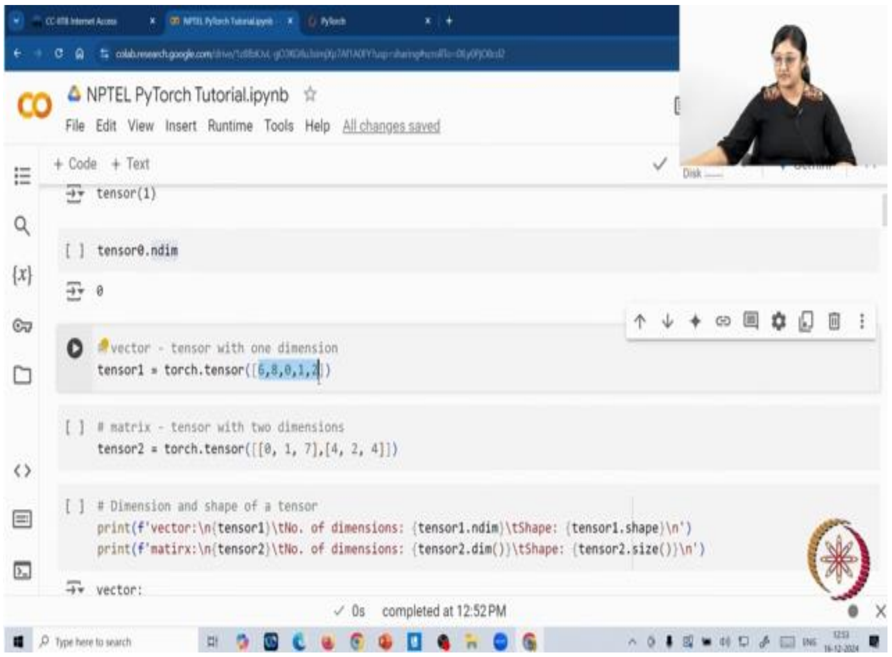
```
# scalar - tensor with zero dimension
tensor0 = torch.tensor(1)
tensor0
```

```
tensor(1)
```

```
tensor0.ndim
```

```
0
```

```
# vector - tensor with one dimension
tensor1 = torch.tensor([6,8,0,1,2])
```


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
✓ 0s completed at 12:52 PM

```
tensor(1)
```

```
tensor0.ndim
```

```
0
```

```
#vector - tensor with one dimension
tensor1 = torch.tensor([6,8,0,1,2])
```

```
# matrix - tensor with two dimensions
tensor2 = torch.tensor([[0, 1, 7],[4, 2, 4]])
```

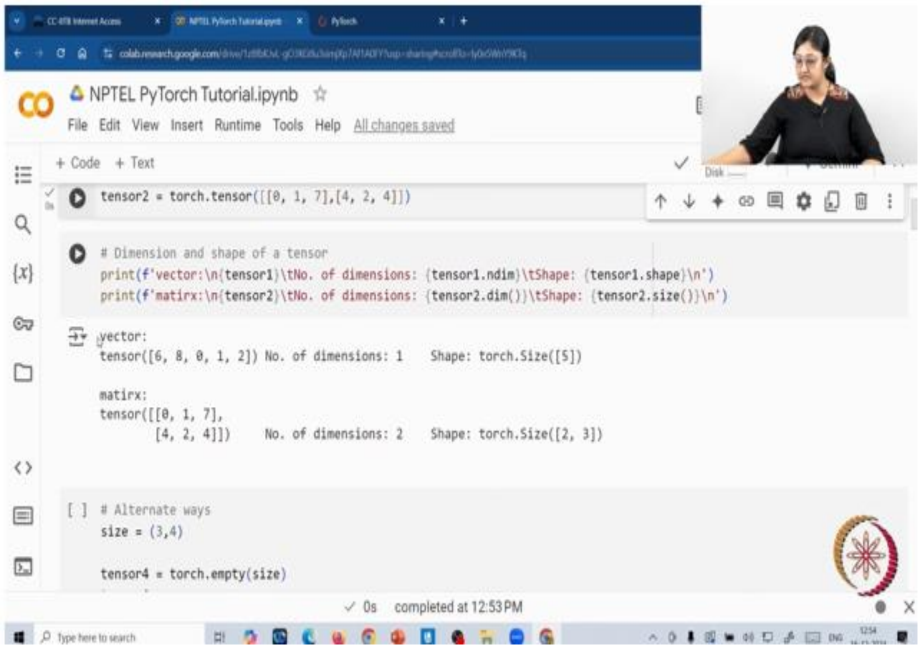
```
# Dimension and shape of a tensor
print(f'vector:\n{tensor1}\tNo. of dimensions: {tensor1.ndim}\tShape: {tensor1.shape}\n')
print(f'matirx:\n{tensor2}\tNo. of dimensions: {tensor2.dim()}\tShape: {tensor2.size()}\n')
```

```
vector:
```

So first, let us create a tensor.

So if we create a tensor by passing any list or a numeric value through the method torch.tensor. Here we pass the single numeric number 1 and when we pass it through torch.tensor, that gives us a tensor, this tensor, tensor one, which is a scalar, and it has zero dimension, as we can see here. So in order to find the dimension of a tensor, we can either use this ndim attribute, or we can also use the dim method.

We'll see that in a while. Next is what we can do is we can stack up a collection of numbers in a list and when we pass that to a tensor to torch dot tensor method that gives us another tensor but this time the tensor would be a one-dimensional tensor and popularly this is known as a vector. So next what we can do is we can stack up vectors which are one dimensional tensors and that would give us 2D tensors which are also known as matrices which we can see over here. So let's run these and till now we have created a scalar, we have created a vector and we have created a matrix.

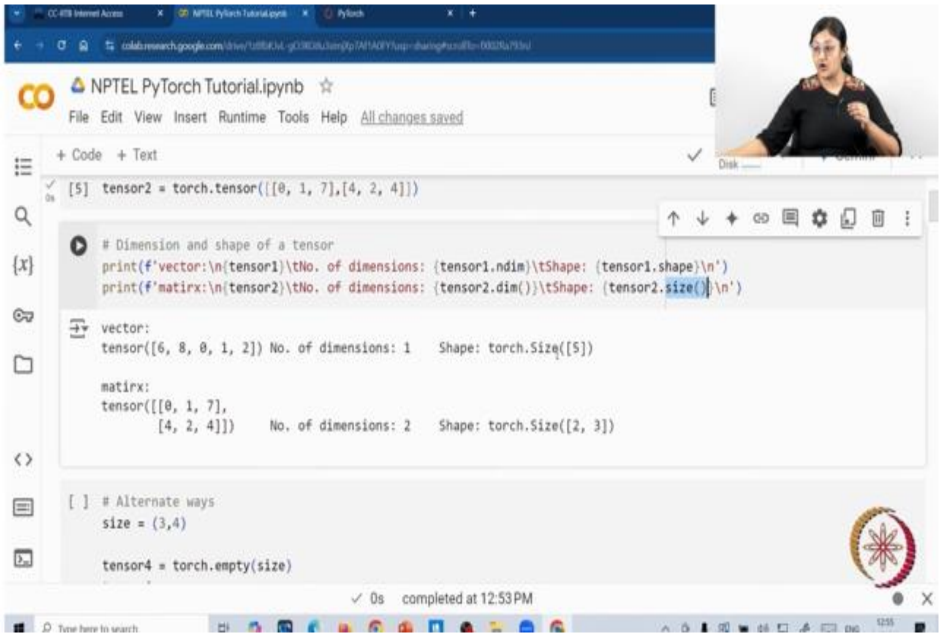

NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
completed at 12:53 PM

```
tensor2 = torch.tensor([[0, 1, 7],[4, 2, 4]])
```

```
# Dimension and shape of a tensor
print(f'vector:\n{tensor1}\tNo. of dimensions: {tensor1.ndim}\tShape: {tensor1.shape}\n')
print(f'matirx:\n{tensor2}\tNo. of dimensions: {tensor2.dim()}\tShape: {tensor2.size()}\n')
```

```
vector:
tensor([6, 8, 0, 1, 2]) No. of dimensions: 1    Shape: torch.Size([5])

matirx:
tensor([[0, 1, 7],
        [4, 2, 4]])     No. of dimensions: 2    Shape: torch.Size([2, 3])
```

```
# Alternate ways
size = (3,4)

tensor4 = torch.empty(size)
```


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text

```
[5] tensor2 = torch.tensor([[0, 1, 7],[4, 2, 4]])
```

```
# Dimension and shape of a tensor
print(f'vector:\n{tensor1}\tNo. of dimensions: {tensor1.ndim}\tShape: {tensor1.shape}\n')
print(f'matirx:\n{tensor2}\tNo. of dimensions: {tensor2.dim()}\tShape: {tensor2.size()}\n')
```

```
vector:
tensor([6, 8, 0, 1, 2]) No. of dimensions: 1    Shape: torch.Size([5])

matirx:
tensor([[0, 1, 7],
        [4, 2, 4]])     No. of dimensions: 2    Shape: torch.Size([2, 3])
```

```
[ ] # Alternate ways
    size = (3,4)

    tensor4 = torch.empty(size)
```

So when we are dealing with tensors a very important concept which we should be thorough with is the dimension and shape of the tensor.

So, let us take a while to understand these concepts. So, first let us if we look at these. So, let us look at the vector and the matrix. So, first if we start by looking at this tensor. So, one way to understand what is the number of dimensions of a tensor is to look at the number of square brackets which is present at one end of the tensor.

So, here we see there is one square bracket. So, the dimension of the tensor is also one, whereas if we look at this matrix. So here we have two square brackets and that gives us the dimension of two. So in pytorch to find the dimension of a tensor we can either use the endrim attribute or we can use the dim method. Also another concept is that of shape.

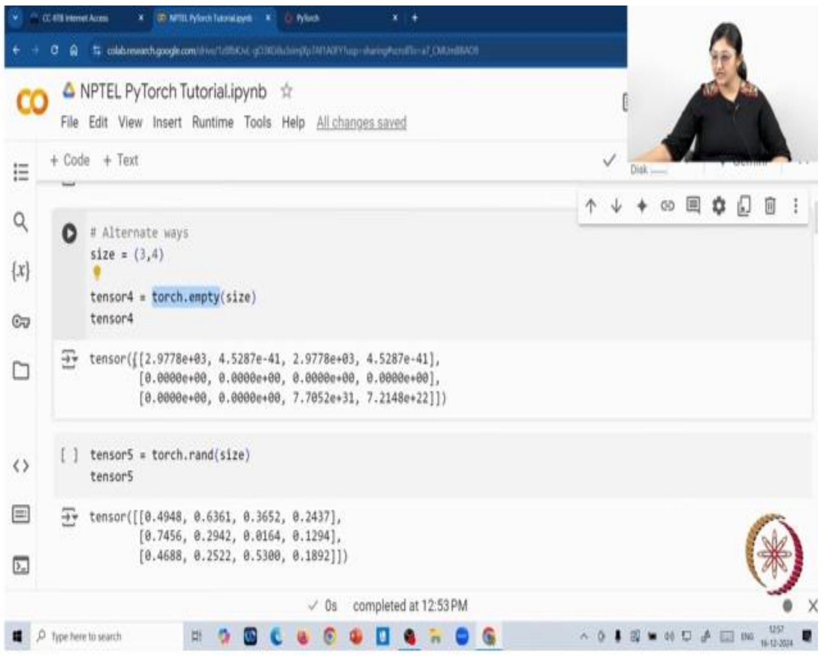
So when we can find the shape of a tensor using the again the shape attribute either by using the shape attribute or by using the dot size method. Here, one thing that I would like to highlight is that the shape or size of a tensor tells us how the elements are arranged across different dimensions in the tensor. And also, the number of elements in the torch.size object that we obtain is the same as the number of dimension the tensor has. So here, this tensor has dimension 2.

And there are also two elements in the torch.size object. each saying what are the number of elements in each of the dimension. In PyTorch, indexing starts from 0. So, since there are two dimensions, this tensor would have dimensions 0 and 1.

So, this 2 is nothing but the number of elements in the 0th dimension and this 3 is the number of elements in the first dimension. So how can we understand that? So this outermost square bracket that we can see in the tensor, this is the 0D dimension. Within that, we can see there is this one, there is this vector, this 1D tensor and another one. So that gives us 2. So there are two elements in 0D.

So, correspondingly, we have this number 2 over here. Now inside each of these vectors or these 1D tensors that I just highlighted, there are three more elements. So here's 0, 1, 7, here's 4, 2, 4. So that means that there are three elements in the first dimension. So, having discussed this, till now we were only seeing how to like we are manually supplying data in torch dot tensor and we are creating the tensor.

There are alternate ways to create tensors as well. So, let us look at that.


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text

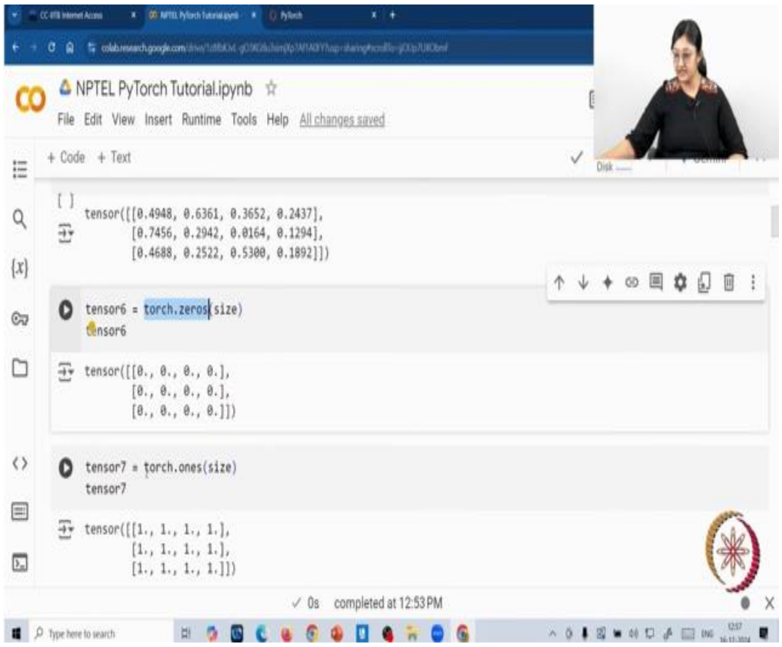
```
# Alternate ways
size = (3,4)

tensor4 = torch.empty(size)
tensor4
```

```
tensor([[2.9778e+03, 4.5287e-41, 2.9778e+03, 4.5287e-41],
        [0.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00],
        [0.0000e+00, 0.0000e+00, 7.7052e+31, 7.2148e+22]])
```

```
tensor5 = torch.rand(size)
tensor5
```

```
tensor([[0.4948, 0.6361, 0.3652, 0.2437],
        [0.7456, 0.2942, 0.0164, 0.1294],
        [0.4688, 0.2522, 0.5300, 0.1892]])
```

✓ 0s completed at 12:53 PM


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text

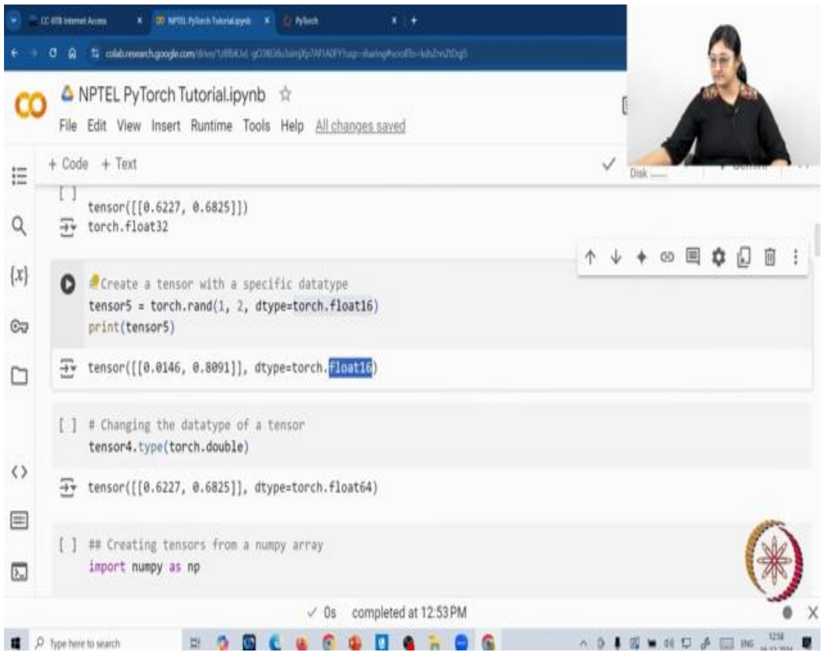
```
tensor([[0.4948, 0.6361, 0.3652, 0.2437],
        [0.7456, 0.2942, 0.0164, 0.1294],
        [0.4688, 0.2522, 0.5300, 0.1892]])
```

```
tensor6 = torch.zeros(size)
tensor6
```

```
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
```

```
tensor7 = torch.ones(size)
tensor7
```

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


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text

```
tensor([[0.6227, 0.6825]])
torch.float32
```

```
#Create a tensor with a specific datatype
tensor5 = torch.rand(1, 2, dtype=torch.float16)
print(tensor5)
```

```
tensor([[0.0146, 0.8091]], dtype=torch.float16)
```

```
# Changing the datatype of a tensor
tensor4.type(torch.double)
```

```
tensor([[0.6227, 0.6825]], dtype=torch.float64)
```

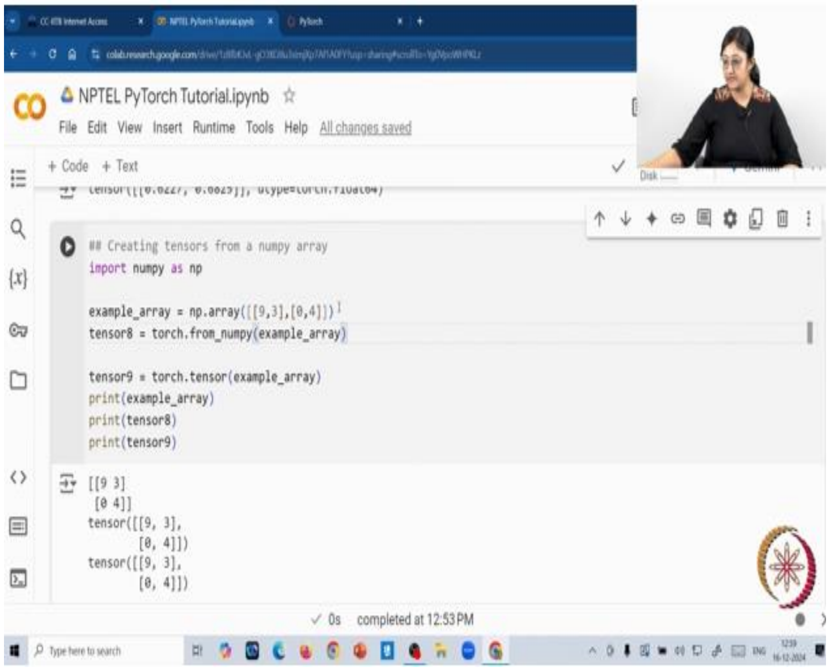
```
## Creating tensors from a numpy array
import numpy as np
```

So, in all these cases, all we need to do is we need to supply the size of the tensor that we want. So, here we are initializing the size to be 3,4 just for an example. First, we can create a torch dot empty tensor.

So that would give us a tensor of the size that we just mentioned and all the values would be uninitialized in this tensor. Another way is to create a random tensor where all the values in the tensor would be a random number between zero and one according to the uniform distribution. We can also create constant tensors using torch.zeros and torch.ones using these methods and that would respectively give us a tensor full of zeros and a tensor full of ones.

Next, we can check the data type of a tensor using the .dtype attribute. So, here we are again creating a random tensor of a desired shape and when we check the data type of the tensor, we see that the data type is float.32. So in PyTorch, the default data type for tensors with float values is float.32. Now we can also specify, when we are creating this tensor, we can specify what data type we want the tensor to be. So here, Here we are creating a random tensor using the torch.ran method. We are specifying the size and then we are specifying the data type over here. So we want now the tensor to be of data type float16.

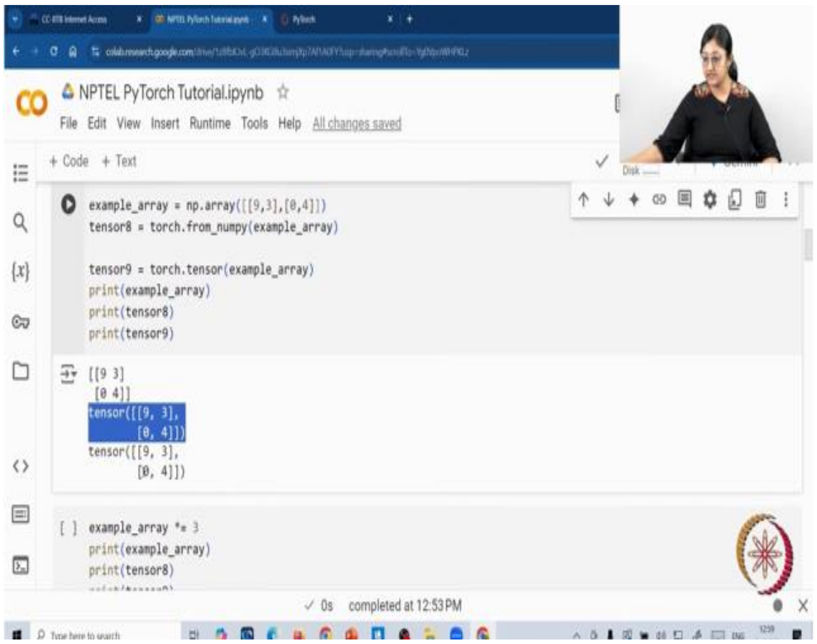
So when we print the tensor, we see over here. So this tensor now has data type float16. Now having created a tensor, we can also change this data type. So torch, sorry, tensor4, additionally had data type float32, we just saw. Now I can use the type method, and now I can pass torch.double. This would change the data type of the tensor from float 32 to float 64. So torch.double and torch.64 are the same data types.


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text

```
## Creating tensors from a numpy array
import numpy as np

example_array = np.array([[9,3],[0,4]])
tensor8 = torch.from_numpy(example_array)

tensor9 = torch.tensor(example_array)
print(example_array)
print(tensor8)
print(tensor9)
```

```
[[9 3]
 [0 4]]
tensor([[9, 3],
        [0, 4]])
tensor([[9, 3],
        [0, 4]])
```

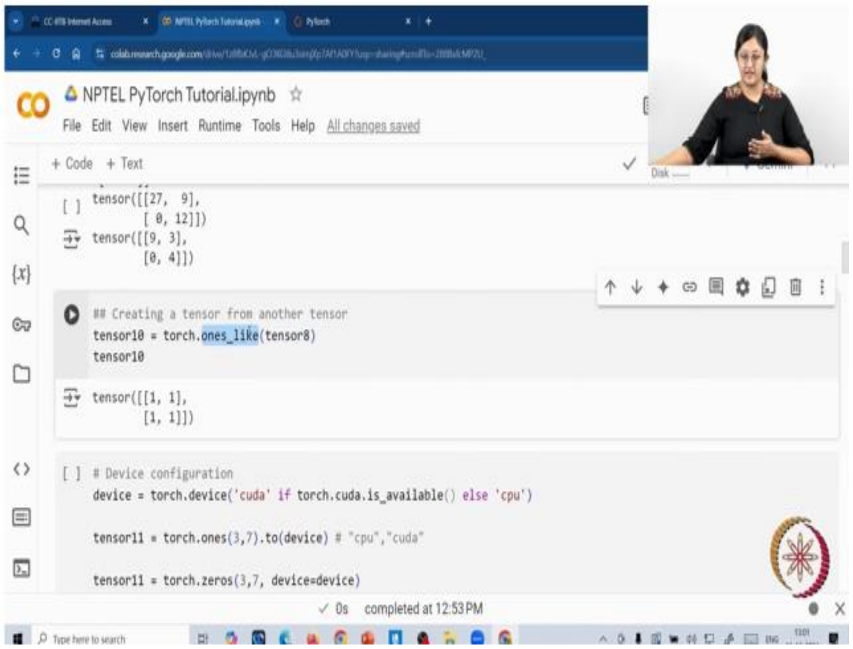
✓ 0s completed at 12:53 PM



```
example_array = np.array([[9,3],[0,4]])
tensor8 = torch.from_numpy(example_array)

tensor9 = torch.tensor(example_array)
print(example_array)
print(tensor8)
print(tensor9)
```

```
[[9 3]
 [0 4]]
tensor([[9, 3],
        [0, 4]])
tensor([[9, 3],
        [0, 4]])
```

```
example_array *= 3
print(example_array)
print(tensor8)
```



```
tensor([[27,  9],
        [ 0, 12]])
tensor([[9, 3],
        [0, 4]])
```

```
## Creating a tensor from another tensor
tensor10 = torch.ones_like(tensor8)
tensor10
```

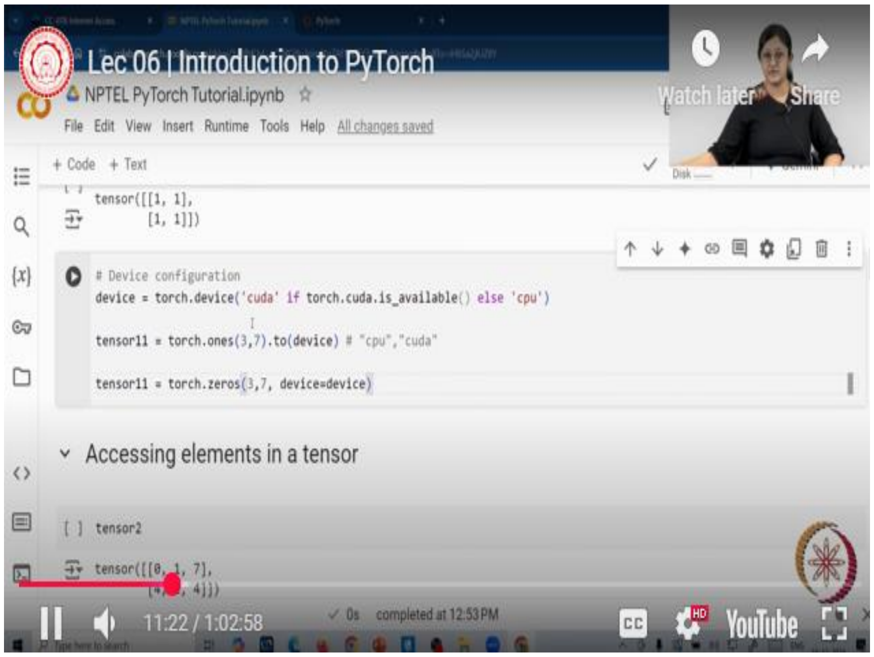
```
tensor([[1, 1],
        [1, 1]])
```

```
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

tensor11 = torch.ones(3,7).to(device) # "cpu","cuda"

tensor11 = torch.zeros(3,7, device=device)
```

Next we can also create a tensor from numpy array.

So first let's create a numpy array. So here we are simply creating a numpy array. Next, there are two ways in which we can create the tensor. So one is we can use torch. fromNumPy method. Here we can simply pass the NumPy array and we will get a tensor. Another way is, as we saw, the method that we were using all these while, which is torch.tensor, we can again pass the array over here and that would give us a tensor corresponding to the NumPy array. So as we can see, this was the numpy array, this is the tensor created using the torch.fromNumpy method, and this is the tensor created using torch.tensor method. So both gives us tensors. However, there's a small distinction that I would like to highlight over here between both of these methods. So let's first look at this. So we take the array, the example array, numpy array, we multiply each of its elements by three. Now when we print the tensors, we see that the tensor which got created using the torch. fromNumpy method, that also gets updated. However, the tensor which got created using the torch.tensor method remains the same. So this implies that whenever you're using this method torch.fromNumpy method then the numpy array and the tensor created they share the same memory. So whatever operation you perform in one tensor will get reflected in the other and however that's not the case when you're doing torch.tensor. So what we can do is we can also create a tensor from another tensor. So here we usually use methods like once underscore like or rand underscore like so it would have this underscore like, the methods would have this trailing underscore like phrase present in it. So what it does is it takes this tensor 8, it retains the data type and retains the shape of this tensor and creates another tensor, which is tensor 10 in this case, but however all its values would be 1. If you had used rand underscore length, then all those values would be random numbers between 0 and 1. So all these are different ways in which you can create a tensor.

Either you can manually supply data or you can create empty or random tensors, constant tensors. You can also create tensors from NumPy arrays and also from other tensors. So all these methods that we just discussed, these are in gist, this is the functionality.


Lec 06 | Introduction to PyTorch
NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Watch later
Share
11:22 / 1:02:58
0s completed at 12:53 PM
YouTube

```
tensor([[1, 1],
        [1, 1]])
```

```
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

tensor11 = torch.ones(3,7).to(device) # "cpu","cuda"

tensor11 = torch.zeros(3,7, device=device)
```

Accessing elements in a tensor

```
tensor2
```

```
tensor([[0, 1, 7],
        [4, , 4]])
```

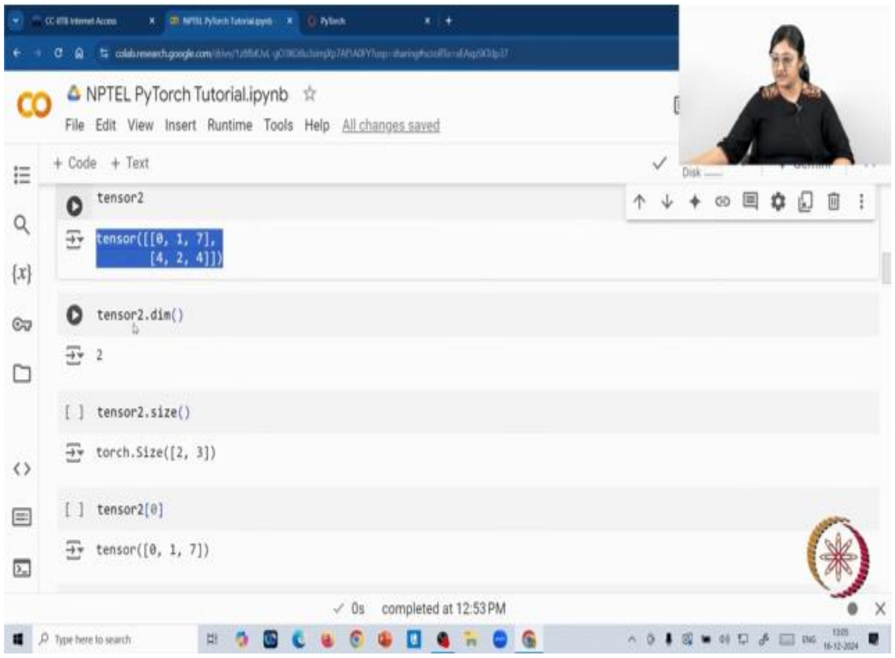
One key concept is that of device. So device is an attribute of each tensor which specifies where the tensor is located.

Is it located on the CPU or is it located on the GPU, which is CUDA? So here, one thing that I would like to mention is, so we have also come across libraries like NumPy, which also more or less give us similar functionalities if you're observing in PyTorch. However, one key advantage of PyTorch is that it gives us GPU support along with the facility to perform tensor operations. So here we see that. So one thing is we can define the device by using this statement.

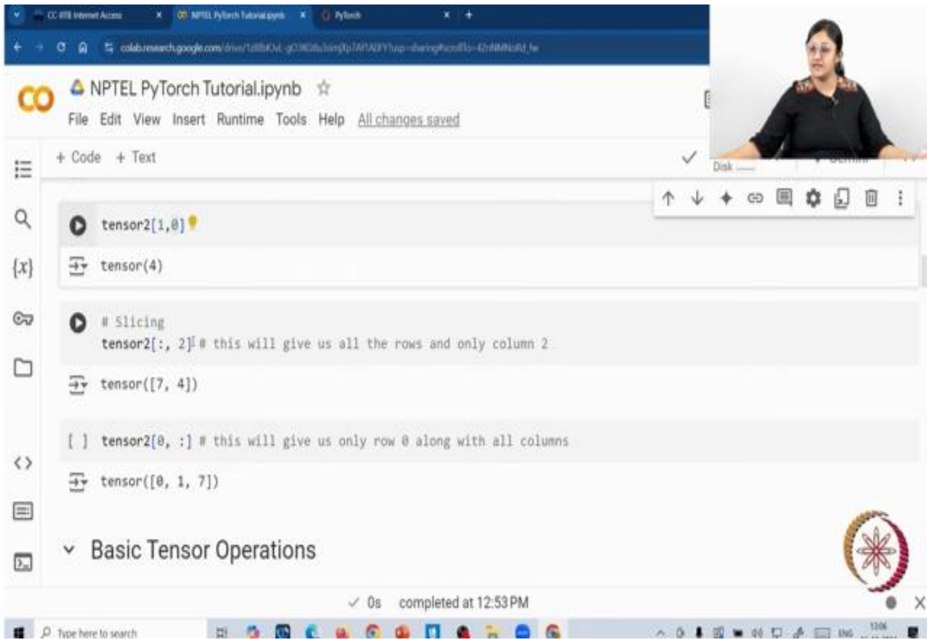
So first we check if CUDA or GPU is available. If it is available, then we set it to CUDA. Otherwise, we let it be CPU. So here we see, so when we are creating this tensor, we first, so this is, which we already saw, then we use this dot two device. So what it would do is, in this case, the first, this tensor, torch dot ones, it would get created in the CPU, and then it would be moved to the GPU, assuming that GPU facility is available on the system. Also, you can explicitly mention where you want to move the tensor to.

Instead of passing device, you can explicitly give CPU, then it would be moved to CPU, or you can explicitly give CUDA, in which case the tensor would be explicitly moved to the GPU. Also, you can specify the device at the time of creating the tensor. So here, we just pass the device to the device argument, and that would directly create the tensor on GPU in this case. So unlike in this first statement, here the tensor was first getting created on the CPU and then it was being moved to GPU. However, when we directly specify the device, when we are creating the tensor, it gets created at the device which we specify over here.

So we can also access elements in a tensor.

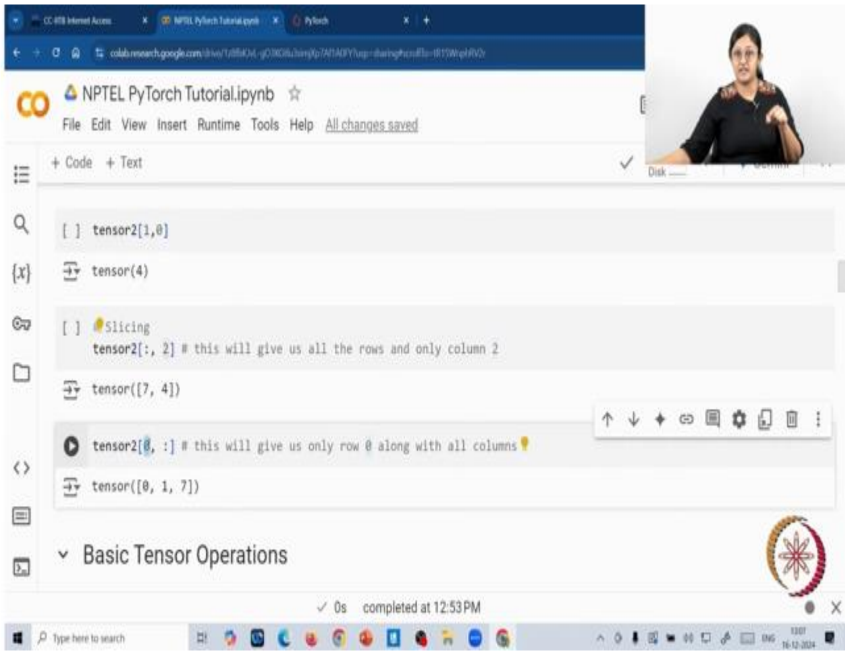


```
tensor2
tensor([[0, 1, 7],
        [4, 2, 4]])

tensor2.dim()
2

tensor2.size()
torch.Size([2, 3])

tensor2[0]
tensor([0, 1, 7])
```



```
tensor2[1,0]
tensor(4)

# Slicing
tensor2[:, 2] # this will give us all the rows and only column 2
tensor([7, 4])

tensor2[0, :] # this will give us only row 0 along with all columns
tensor([0, 1, 7])
```

Basic Tensor Operations


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text

```
tensor2[1,0]
```

```
tensor(4)
```

```
#Slicing
tensor2[:, 2] # this will give us all the rows and only column 2
```

```
tensor([7, 4])
```

```
tensor2[0, :] # this will give us only row 0 along with all columns
```

```
tensor([0, 1, 7])
```

Basic Tensor Operations
✓ 0s completed at 12:53 PM

So let's look at a few examples. So here we have this tensor 2. We know its dimension is 2.

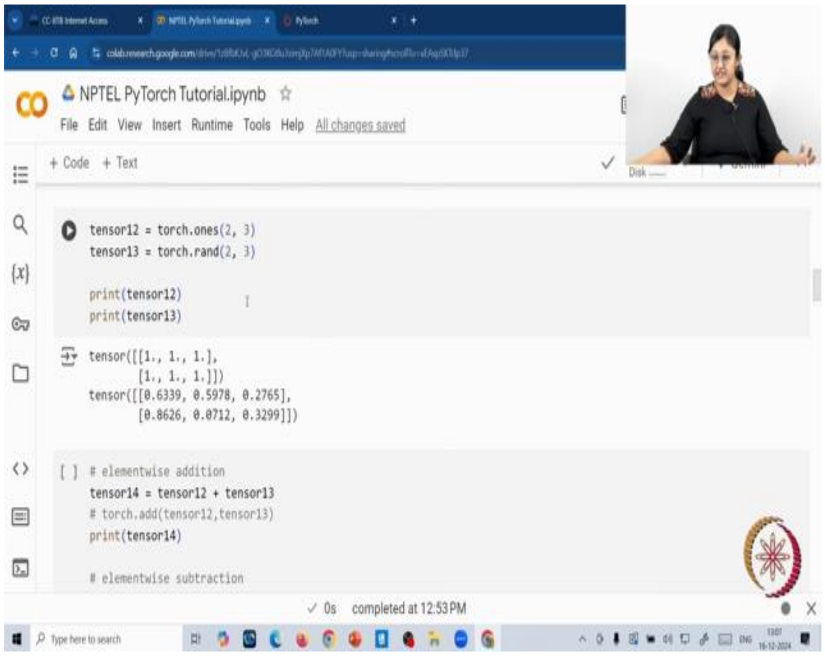
We have two square brackets here. Its size is 2, 3. Let's look at this once just to reiterate. So there are three scalar values. Three zero dimensional tensors which are being packed to create this one dimensional tensor. And there are two of these one dimensional tensors.

So that gives us the size of two comma three. Now if you want to look at tensor zero, so this index specifies what is the element I'm looking in the zeroth dimension. So in the zeroth dimension, these are the two elements that I have. And in that, this is the 0th element and this is the first element. So tensor 2, 0 would give us this vector.

So as we see over here. Now if we do tensor 1,0, in that case what would happen is, so this is the index corresponding to the 0th dimension and this is the index corresponding to the first dimension. So first what we do is we go to the 0th dimension and see what is the first element. This is the first element and then further we check that here what is the 0th element.

So that is none other than 4. So we have 4 over here. We can also do slicing operations over here. Slicing is similar to we have also come across slicing when we have deal with numpy. It's quite similar to that. So, here, when we do colon comma 2, so that means all the elements in the 0th dimension, but only 2, only element 2 in the first dimension, that is nothing but only column 2.

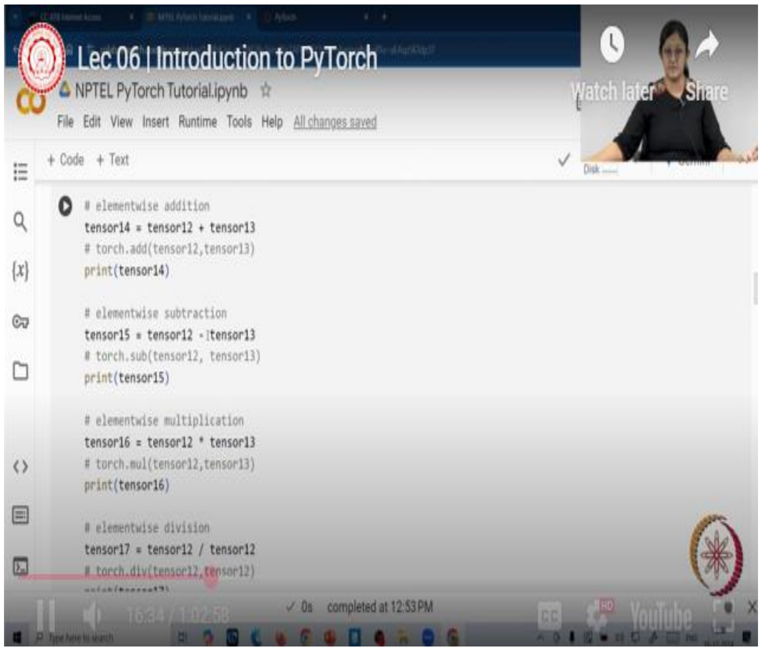
Let us look at it. So, this 7 comma 4. So that is what we obtain over here. Whereas next if we look at tensor 0 colon, so colon means all the elements, 0 is we are specifying we want this specific element in this dimension. So what would be the result in this case? Nothing but the first, this 0, 1, 7. So I think this is quite intuitive, I hope this was clear. So next we can discuss about a few very, so this is very basic, so some basic tensor operations.


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
✓ 0s completed at 12:53 PM

```
tensor12 = torch.ones(2, 3)
tensor13 = torch.rand(2, 3)

print(tensor12)
print(tensor13)
```

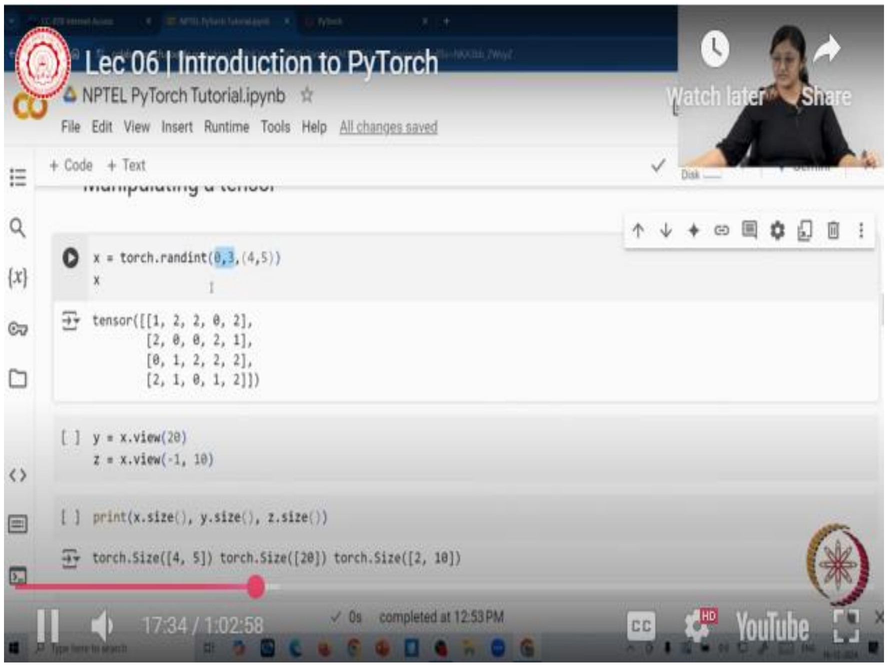
```
tensor([[1., 1., 1.],
        [1., 1., 1.]])
tensor([[0.6339, 0.5978, 0.2765],
        [0.8626, 0.0712, 0.3299]])
```

```
# elementwise addition
tensor14 = tensor12 + tensor13
# torch.add(tensor12,tensor13)
print(tensor14)

# elementwise subtraction
```



Lec 06 | Introduction to PyTorch
NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Watch later
Share
16:34 / 1:02:58
✓ 0s completed at 12:53 PM

```
# elementwise addition
tensor14 = tensor12 + tensor13
# torch.add(tensor12,tensor13)
print(tensor14)

# elementwise subtraction
tensor15 = tensor12 - tensor13
# torch.sub(tensor12, tensor13)
print(tensor15)

# elementwise multiplication
tensor16 = tensor12 * tensor13
# torch.mul(tensor12,tensor13)
print(tensor16)

# elementwise division
tensor17 = tensor12 / tensor12
# torch.div(tensor12,tensor12)
print(tensor17)
```



Lec 06 | Introduction to PyTorch
NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Watch later
Share
17:34 / 1:02:58
✓ 0s completed at 12:53 PM

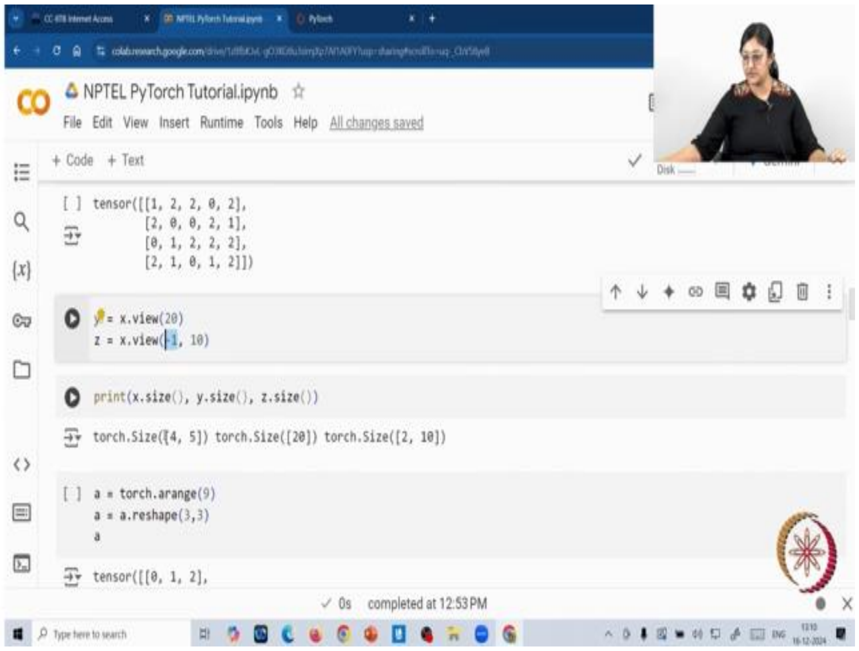
```
x = torch.randint(0,3,(4,5))
x
```

```
tensor([[1, 2, 2, 0, 2],
        [2, 0, 0, 2, 1],
        [0, 1, 2, 2, 2],
        [2, 1, 0, 1, 2]])
```

```
y = x.view(20)
z = x.view(-1, 10)
```

```
print(x.size(), y.size(), z.size())
```

```
torch.Size([4, 5]) torch.Size([20]) torch.Size([2, 10])
```


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
✓ 0s completed at 12:53 PM

```
tensor([[1, 2, 2, 0, 2],
        [2, 0, 0, 2, 1],
        [0, 1, 2, 2, 2],
        [2, 1, 0, 1, 2]])
```

```
y = x.view(20)
z = x.view(-1, 10)
```

```
print(x.size(), y.size(), z.size())
```

```
torch.Size([4, 5]) torch.Size([20]) torch.Size([2, 10])
```

```
a = torch.arange(9)
a = a.reshape(3,3)
a
```

```
tensor([[0, 1, 2],
```

I'll just quickly go through this because these are quite easy. So we're just creating two tensors just to illustrate how these operations work. So we randomly, we created two tensors. Now there are four operations, element wise addition, subtraction, multiplication and division. So, As you can see, so in order to perform addition, we can either use this plus symbol or we can use the method torch.add and supply the corresponding tensors as the function argument. So this allows us to add two tensors. Similarly for subtraction, we can use the symbol minus or we can use torch.sub method and similarly for multiplication and division. So here we can see, so these were the tensors that were created.

Now if we add them, we get this. If we subtract, we get this. And this is again multiplication and division. These are the results. Next we'll look at a few operations which allows us to manipulate tensors. So, here we come across a new function.

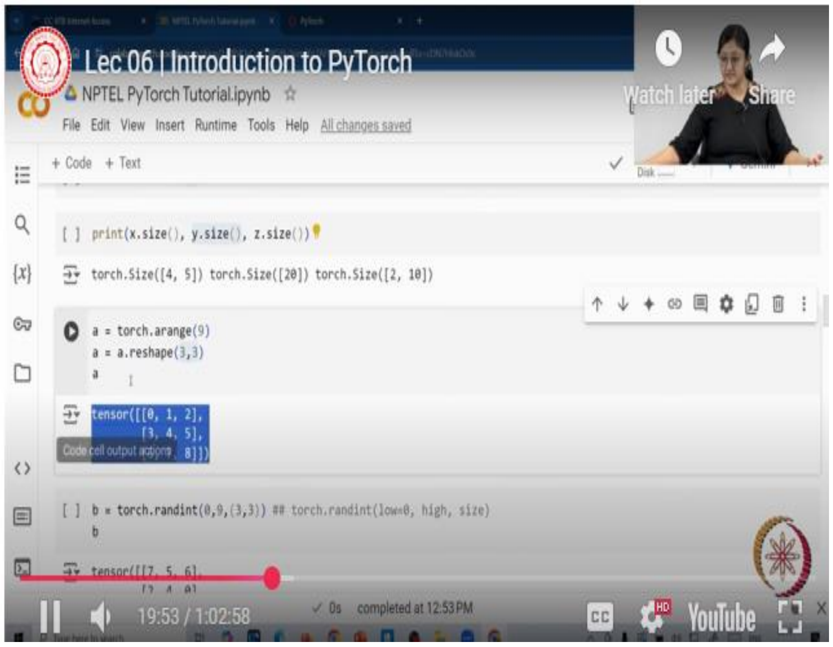
So, this is nothing but like this. So, here we specify the range of integers and within this range a tensor will get created of size 4 comma 5 with numbers that lie within this range. So, that is what ran dot int does. So, here we have created this tensor. So one important idea which we encounter quite often in pipe torches is that of view.

So this allows us to change the shape of the tensor. So, here we create another tensor y where we specify the shape to be 20 and another tensor z where we specify minus 1 comma

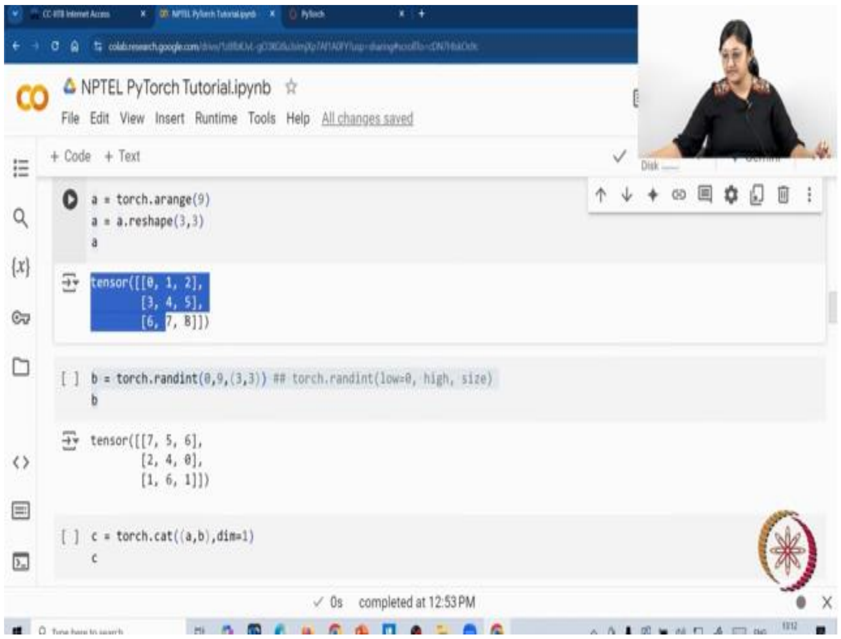
10. So, minus 1 comma 10 means in the second. So, this is the 0th and this is the first. So, in the first dimension we want 10 elements, whereas minus 1 is a flexible argument. So PyTorch will automatically figure out what the number of elements should be in this dimension.

So as we can see over here, we're printing the size, originally x was 4, 5. Now when I use the view method to create y, now the size is 20 and when I created Z so now I had only specified 10 over here and torch automatically figured out that okay there should be two so there were two elements in dimension zero so that the overall shape and all the elements in the tensor gets retained. So one thing is when you use the view method to create another tensor, all the tensors share the common memory space. So if you do any change to any one of the tensors, it would get reflected in others.

So please be careful about it when you're using this method. There is also torch.arrange. Using this, since I specified 9, a 1D tensor will get created with numbers from 0 to 8. So this is similar to the range function that we have in PyTorch.


Lec 06 | Introduction to PyTorch
NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Watch later
Share
Code cell output actions
19:53 / 1:02:58
0s completed at 12:53 PM
YouTube

```
print(x.size(), y.size(), z.size())
```
```
torch.Size([4, 5]) torch.Size([20]) torch.Size([2, 10])
```
```
a = torch.arange(9)
a = a.reshape(3,3)
a
```
```
tensor([[0, 1, 2],
        [3, 4, 5],
        8]])
```
```
b = torch.randint(0,9,(3,3)) ## torch.randint(low=0, high, size)
b
```
```
tensor([[7, 5, 6],
```


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
✓ 0s completed at 12:53 PM

```
a = torch.arange(9)
a = a.reshape(3,3)
a
```

```
tensor([[0, 1, 2],
        [3, 4, 5],
        [6, 7, 8]])
```

```
b = torch.randint(0,9,(3,3)) ## torch.randint(low=0, high, size)
b
```

```
tensor([[7, 5, 6],
        [2, 4, 0],
        [1, 6, 1]])
```

```
c = torch.cat((a,b),dim=1)
c
```

Now we can also do reshape. Reshape also allows us to change the shape of the tensor. And here we specify 3, 3. So that gives us a tensor with the desired shape and the elements from this tensor. So, there is a distinction between view and reshape. So, sometimes reshape creates a view, sometimes it creates a copy of the original tensor.

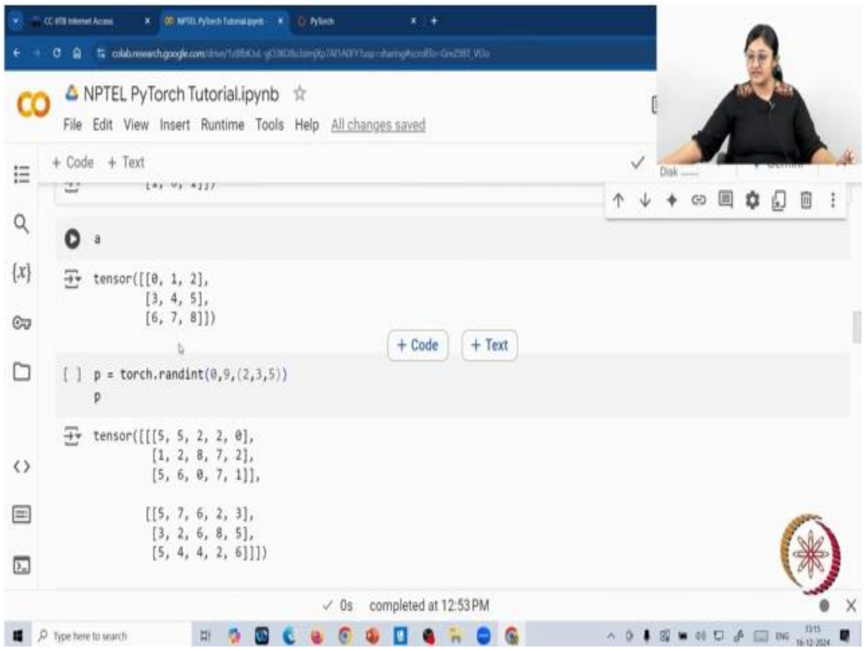
So, there is also, there is a bit of ambiguity over here. So, also please be careful if you are using the reshape method. Next is, as I mentioned, we were discussing about dimension in the beginning. Dimension is quite important when we are dealing with various tensor operations. Here we'll look at the first for such instance, for such a method where we are specifying the dimension and along the dimension the operation is happening.

So here we have tensor A, here we have tensor B. Now how do we want to create tensor C? So for tensor C, I want to concatenate both of these tensors A and B along dimension 1. So what is dimension 1? This is dimension 1. So the 0, 1, 2, 3, 4, 5, these are dimension 1. So if we look at the first element, so this is the first element in the 0 dimension.

Now if I further look into it, in one dimension the elements are 0, 1, and 2. Similarly, in one dimension the elements in tensor B are 7, 5, 6. So when we concatenate along dimension 1, the resulting tensor would stack up all these elements 0, 1, 2 and 7, 5, 6 in one vector. So as we can see over here. So this was tensor A, this was tensor B when I concatenated along dimension 1. 0, 1, 2, 7, 5, 6 came along in one vector and corresponding similarly for the remaining values.

You can apply the same logic to the remaining values to get tensor C. So, one thing is if you look at the shape of tensor A, it was 3, 3. If you look at the shape of tensor B, it was again 3, 3. Now, when I concatenate along dimension 1, the shape of the resulting tensor is 3, 6. So the number of elements in dimension 1 got concatenated and so as a result the size in dimension 1 increased.

So that is 3 plus 3, now 6. You can also concatenate along dimension 0, then simply all these things would get stacked one below the other. So as we can see over here. So, again here the resulting shape would be 6, 3. So, when we were, we specified dimension 0 to be the dimension across which we want to perform the concatenation operation, so that got changed, so that got increased.


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
+ Code + Text

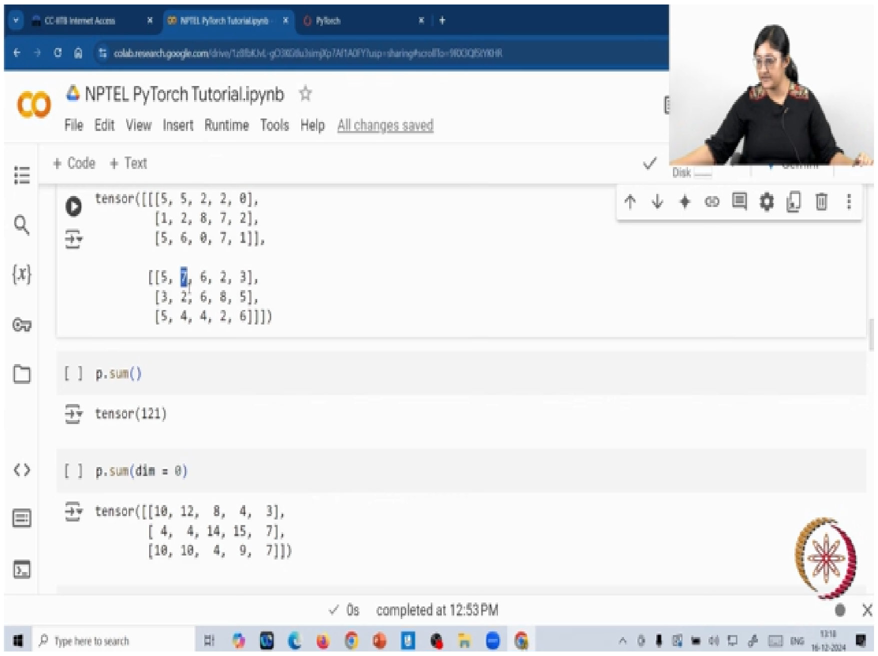
```
a
```

```
tensor([[0, 1, 2],
        [3, 4, 5],
        [6, 7, 8]])
```

```
p = torch.randint(0,9,(2,3,5))
p
```

```
tensor([[[5, 5, 2, 2, 0],
         [1, 2, 8, 7, 2],
         [5, 6, 0, 7, 1]],

        [[5, 7, 6, 2, 3],
         [3, 2, 6, 8, 5],
         [5, 4, 4, 2, 6]]])
```

✓ 0s completed at 12:53 PM



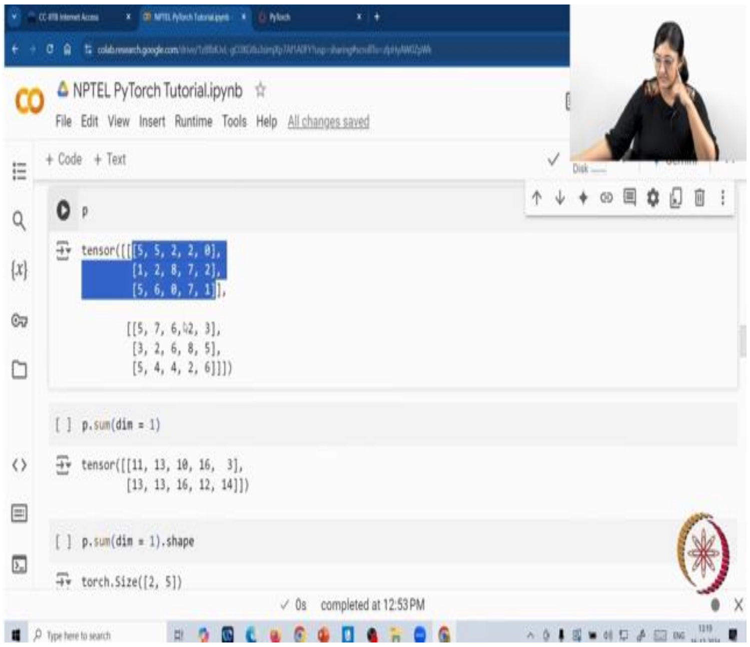
```
tensor([[[5, 5, 2, 2, 0],
         [1, 2, 8, 7, 2],
         [5, 6, 0, 7, 1]],

        [[5, 7, 6, 2, 3],
         [3, 2, 6, 8, 5],
         [5, 4, 4, 2, 6]]])
```

```
p.sum()
```

```
tensor(121)
```

```
p.sum(dim = 0)
```

```
tensor([[10, 12,  8,  4,  3],
        [ 4,  4, 14, 15,  7],
        [10, 10,  4,  9,  7]])
```



```
p
```

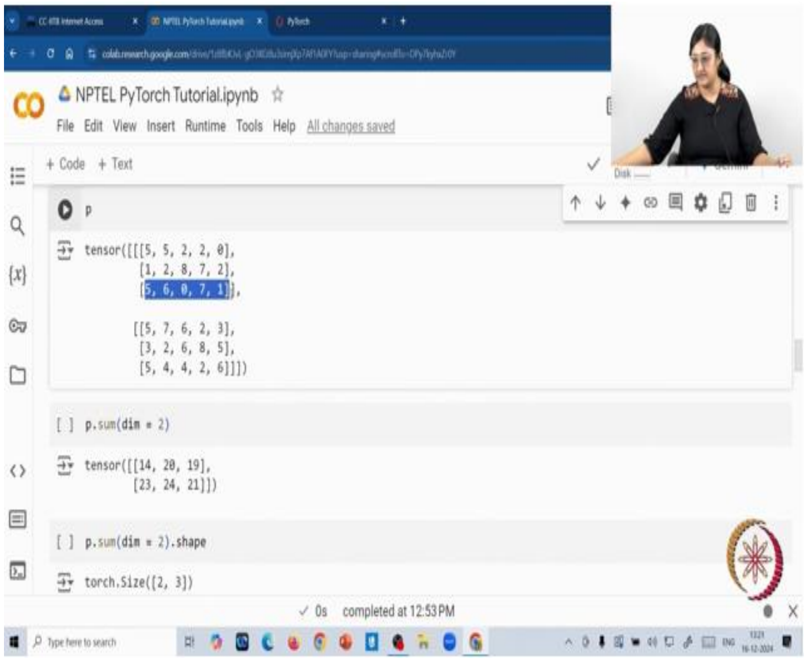
```
tensor([[[5, 5, 2, 2, 0],
         [1, 2, 8, 7, 2],
         [5, 6, 0, 7, 1]],

        [[5, 7, 6, 2, 3],
         [3, 2, 6, 8, 5],
         [5, 4, 4, 2, 6]]])
```

```
p.sum(dim = 1)
```

```
tensor([[11, 13, 10, 16,  3],
        [13, 13, 16, 12, 14]])
```

```
p.sum(dim = 1).shape
```

```
torch.Size([2, 5])
```


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
✓ 0s completed at 12:53 PM

```
p
```

```
tensor([[[5, 5, 2, 2, 0],
         [1, 2, 8, 7, 2],
         [5, 6, 0, 7, 1]],

        [[5, 7, 6, 2, 3],
         [3, 2, 6, 8, 5],
         [5, 4, 4, 2, 6]]])
```

```
p.sum(dim = 2)
```

```
tensor([[14, 20, 19],
        [23, 24, 21]])
```

```
p.sum(dim = 2).shape
```

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

Next, we can look at a tensor reduction operation. So in tensor reduction operation, what happens is one of the dimensions gets reduced. So it is called a reduction operation. There are many such reduction operations. So here we'll be looking at some.

There is mean, there is argmax, all these are reduction operations. So I will be going through one of them over here. I would encourage the students to look at the remaining whenever they are going through a tutorial. So here I am creating this tensor. So this is a three-dimensional tensor of shape 2, 3, 5 and it has, so this is a random tensor, it would have values between 0 to 9. So, just to again reiterate the concepts of dimension and shape, so there are 3 square brackets, its dimension would be 3 and the shape is, so if you look, so this is the outermost square bracket, it has 2 elements. So that is why it has a shape of so it has it has the number two so two elements in zero dimension if we look within that we'll have three three such 1d vectors. So that gives us the number three, so three elements in the first dimension and then in each of these we further have five elements.

So one, two, three, four, five, five elements. So two, three, five. Again, like if I go in the opposite direction, if I start from here, so I have five zero-dimensional scalars. So I have five zero-dimensional scalars stacked up to create this one-dimensional tensor. And there are three such one-dimensional tensors, okay, so that corresponds to number three, which is coming together to form the, so which is coming together, and there are two of them, and that gives us this number two. So we start from 0D, we move to 1D, we stack up 1D, that gives us 2D and then we stack up 2D, that gives us 3D.

So by stacking up these tensors, we are getting a higher dimensional tensor. So now when I do p.sum, all the values get summed up and I get 121. Then if I want to sum up along dimension 0, what does that mean? So, again I mentioned that dimension 0 is this, the outermost dimension, the outermost square bracket. Within that, these are the two elements, this one and this one. So, when we are summing these, what we do is we take each of these elements and along dimension 0. So, and this remember, so when we specify dimension 0, so ultimately since this is a reduction operation, after the operation dimension 0 would, so that would get reduced.

So, what we do is we simply add all these elements. So, there are two elements we simply add them up and that would give us the result when we perform the sum operation in dimension 0. So, 5 plus 5 over here that gives us 10, 5 plus 7 gives us 12. So, similarly for the other values.

So, this is along dimension 0. And one thing is now the shape is 3, 5 of the resulting operation. So, when we performed across dimension 0, this got reduced, but only remaining with 3, 5 as we can see here. Again, if I look at tensor P, now I want to perform the sum operation across dimension 1. So, which is the dimension 1? So, this entire vectors would be dimension 1. Each of these vectors would be dimension 1. So, what happens is when we perform an operation across dimension 1, we just have to individually add up all of these.

So, this plus this plus this would give us this. So we can see 5 plus 5 plus 1 gives us the first element over here. 5 plus 6 plus 2 gives us this. So you add all these things, you get the first element over here.

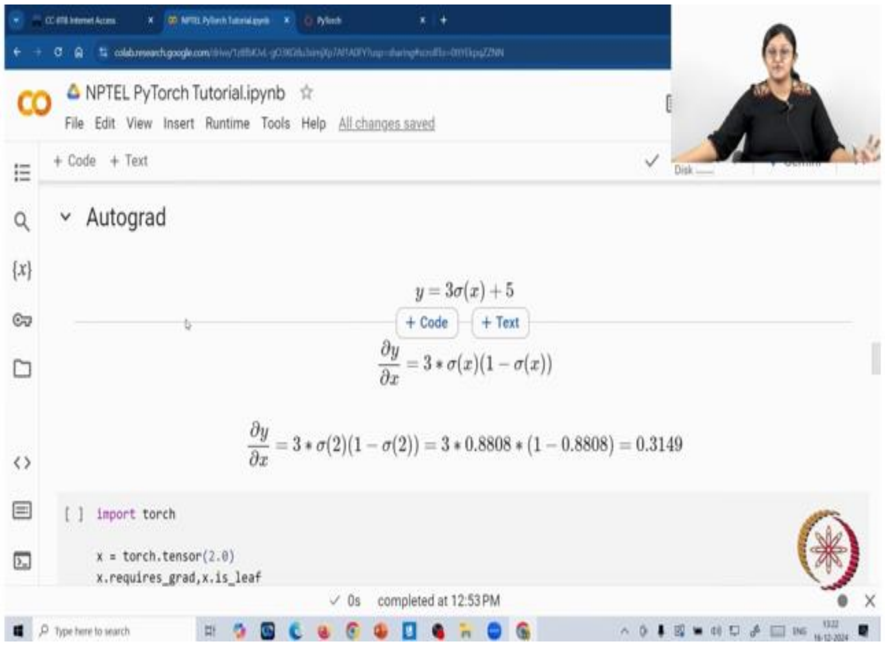
You add all these things, you get the second element here. So you need to observe. So each of these are the one-dimensional elements. So each of these are the elements in dimension one. So we are simply, when we're doing sum, so we are simply adding those up. And ultimately, now the resultant tensor has shape two comma five.

So now we can also perform the sum operation across dimension 2. So what is dimension 2? Dimension 2 is the innermost dimension. So each of these individual values, these individual 0D tensors that I have, that is the scalars, these are the elements in dimension 0.

So we simply sum them up. We sum these. So this is the elements in dimension 0. We sum these. We get 14. We sum these. we get 20. We sum these, we get 19. So now we are performing the sum operation across dimension 0.

Previously we were summing across dimension 1 which is this dimension and that is what we observed a while back. Now we are summing across dimension 2. So now as I mentioned so we simply add them up and we get this 14. Now the shape would be 2, 3. So the dimension across which I am performing the operation that gets reduced.

So now we have more or less covered the fundamental concepts of tensor. However, like PyTorch is a huge library, there are a lot more things that are present. So again, I would encourage the students to go and visit the PyTorch documentation and learn more if they feel interested.


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text

Autograd

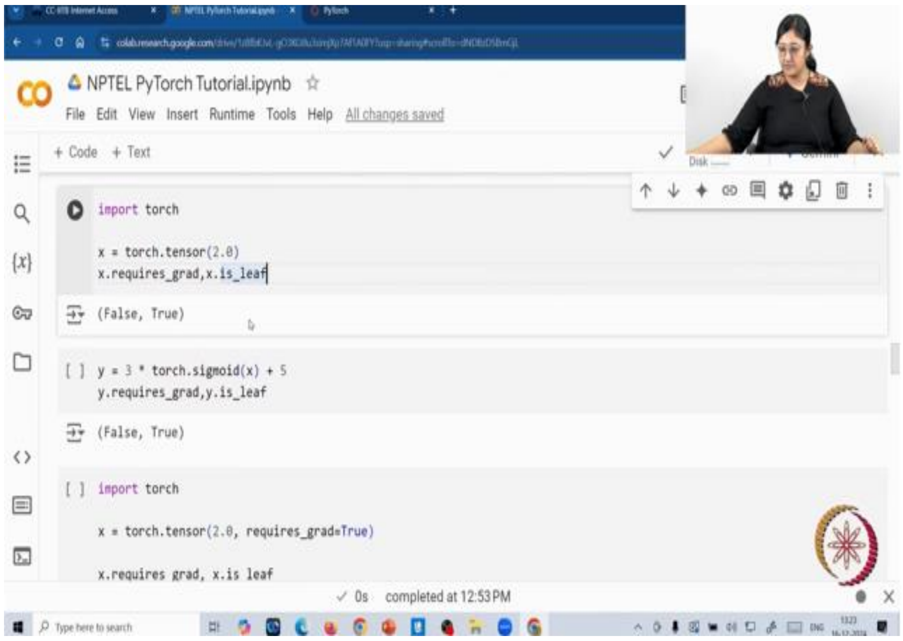
$$y = 3\sigma(x) + 5$$

$$\frac{\partial y}{\partial x} = 3 * \sigma(x)(1 - \sigma(x))$$

$$\frac{\partial y}{\partial x} = 3 * \sigma(2)(1 - \sigma(2)) = 3 * 0.8808 * (1 - 0.8808) = 0.3149$$

```
import torch

x = torch.tensor(2.0)
x.requires_grad,x.is_leaf
```

✓ 0s completed at 12:53 PM


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
✓ 0s completed at 12:53 PM

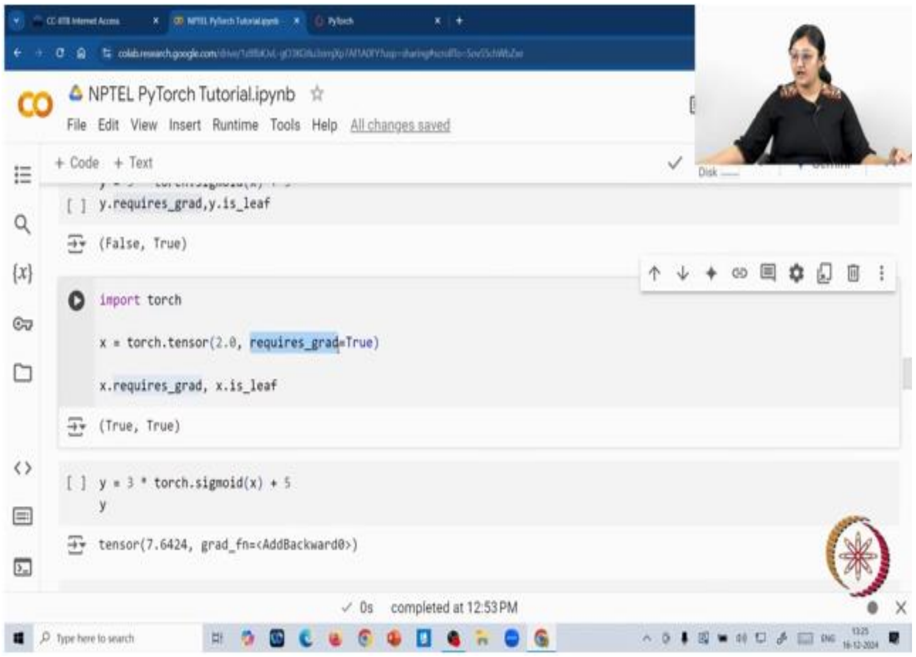
```
import torch

x = torch.tensor(2.0)
x.requires_grad,x.is_leaf
```

```
(False, True)
```

```
y = 3 * torch.sigmoid(x) + 5
y.requires_grad,y.is_leaf
```

```
(False, True)
```

```
import torch

x = torch.tensor(2.0, requires_grad=True)

x.requires_grad, x.is_leaf
```


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
✓ 0s completed at 12:53 PM

```
y.requires_grad,y.is_leaf
```

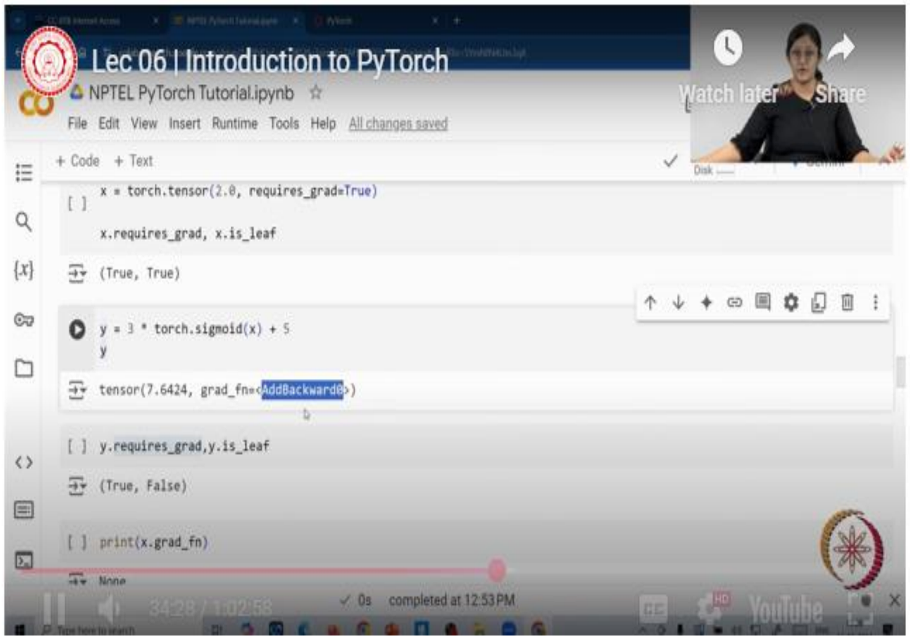
```
(False, True)
```

```
import torch

x = torch.tensor(2.0, requires_grad=True)

x.requires_grad, x.is_leaf
```

```
(True, True)
```

```
y = 3 * torch.sigmoid(x) + 5
y
```

```
tensor(7.6424, grad_fn=<AddBackward0>)
```


Lec 06 | Introduction to PyTorch
NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Watch later
Share
34:28 / 1:02:58
completed at 12:53 PM

```
x = torch.tensor(2.0, requires_grad=True)

x.requires_grad, x.is_leaf
```

```
(True, True)
```

```
y = 3 * torch.sigmoid(x) + 5
y
```

```
tensor(7.6424, grad_fn=<AddBackward0>)
```

```
y.requires_grad,y.is_leaf
```

```
(True, False)
```

```
print(x.grad_fn)
```

```
None
```

Next is we'll understand autograd. So autograd allows us to compute gradients in PyTorch.

So first we'll start with a toy example and we'll see what is happening. So let's take this example. So say we have this equation y is equals to 3 times sigma of x plus 5. Now mathematically if I want to calculate the gradient of y with respect to x we just simply take the derivative. The derivative would be 3 times sigma of x 1 minus sigma of x. Now say I want the derivative at a particular value say at x is equals to 2.

So I pass in those values and ultimately dy dx is we get this 0.3149.

$$y = 3\sigma(x) + 5$$

$$\frac{\partial y}{\partial x} = 3 \times \sigma(x)\big(1 - \sigma(x)\big)$$

$$\frac{\partial y}{\partial x} = 3 \times \sigma(2)\big(1 - \sigma(2)\big)$$

$$= 3 \times 0.8808 \times (1 - 0.8808)$$

$$0.3149$$

Now if I want to do a similar thing in PyTorch, how can I do it? So this is just for elastration. This is a very simple example. This is not the reason why we use PyTorch.

This is just so that I can make you all understand what is happening. So here, I would like to mention one thing. That is the concept of leaf and non-leaf tensors. So what happens in pie torches when we are using autograd? Internally what is happening is that there's this computational graph which is getting created which keeps track of all the operations that are being performed on a tensor. We'll understand this as we go ahead.

I'm just giving a heads up. So in this computational graph, there are leaf nodes and non-leaf nodes. So what are those? So here I create this tensor. and i look at the attributes required grad and is_Leaf. So by default the required grad attribute is false and is_Leaf is true. So, in general, all tensors which have required grad to be false are leaf tensors.

So I use the tensor x that I just created and I create a tensor y which is a function of x, it is 3 times sigma of x plus 5. So, now what are the attributes requiredGrad and isLeaf for y? RequiresGrad is still false, and isLeaf is true. So, now both x and y are leaf tensors. However, let us look at a different scenario. Here remember we did not explicitly set the requiresGrad attribute to true.

So, in this case both of them were leaf nodes, both x and y were leaf tensors. However, now Let's explicitly set when we are creating tensor X, I'm explicitly setting requires grad to be true. So this is true and we see that X is leaf is still true. So here one thing is, so when a user is creating a tensor and explicitly setting its required grad to be true, it still remains a leaf tensor.

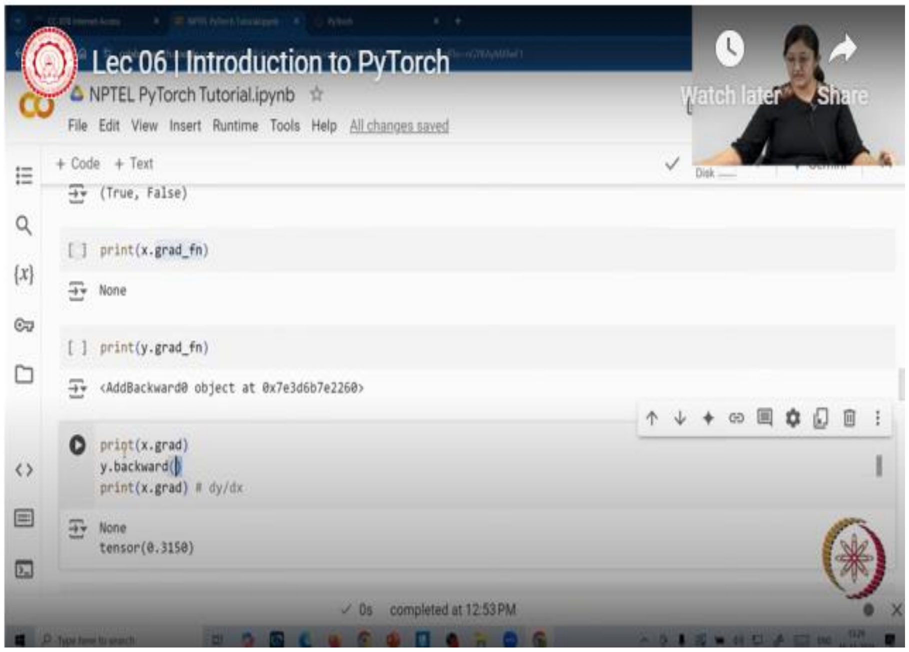
However, let's see what happens next. Now I'm again creating Y. Y is the same function of X that we just saw. And now if I look at the requires grad and isLeaf of Y, we see that Y is no longer a leaf node. So this is because Y, the tensor Y is the resultant tensor of, because we performed an operation on X, so it no longer remains a leaf tensor. It's now a non-leaf tensor. And another thing is it requires grad is also true for y, even though we did not set anything. So why is that? So what happens in the computational graph is, so say I said requires grad to be true for x and then we performed an operation using x and we generated y.

So all these things are being tracked or traced in the computational graph. So, and after a while we see that we will use this backward method to compute the gradient. So what happens is when we call the backward method, the gradient flows through all the nodes in the computational graph. So as a result, even though I did not specify why to have requires grad to be true, however, since it was created from this tensor, it also has required grad to be true. So because the gradient needs to flow through all these, even through this node, so the gradient needs to be calculated.

So that is why it has requires grad to be true. Another thing is grad function, so grad function is what is called when, so when I call the backward method to compute the gradient, so what happens is this grad function contains a pointer to this function add backward and this is called in order to compute the So, the computation happens via this. So, I will tell this very briefly. If you want to know further, you can refer to the PyTorch documentation.

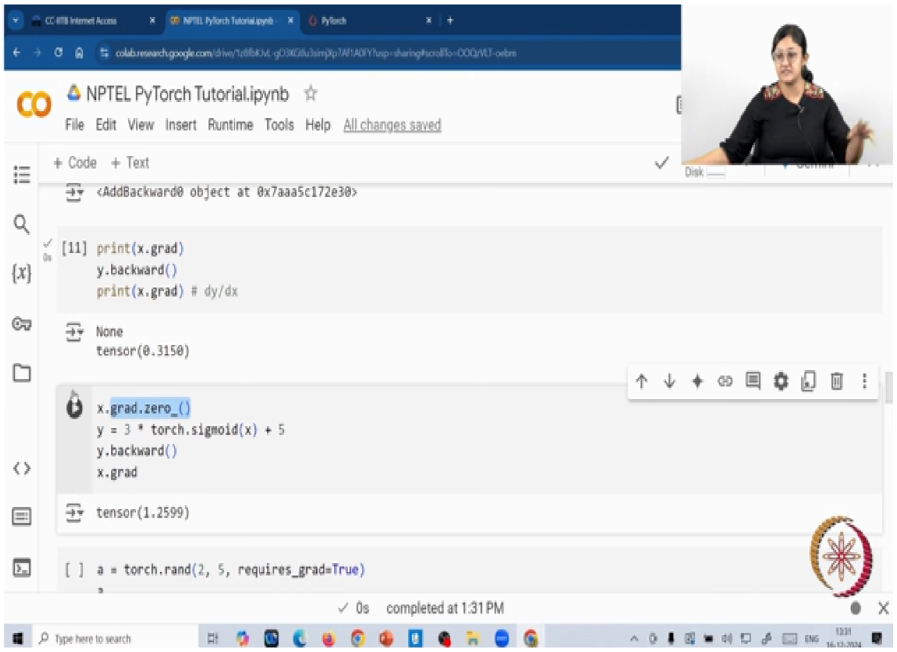
So, here you see that there is this addition operation. So, that is why we get add backward. Further, if you look further, you will also find since there are sigmoid operation, there would be a sigmoid backward involved. There is a multiplication operation. So, there will be a mull backward operation involved. So, for each of the operations, we simply add this word backward and that actually points to the function which will be called when this operation is encountered, when we are tracing back the computational graph during the backward pass or during the computation of gradient. So for y which was formed as a result of an operation on x, it has a grad function, however x which was the input vector does not have a grad function.

So yeah, as we saw over here.


Lec 06 | Introduction to PyTorch
Watch later
Share
NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text

```
(True, False)

[ ] print(x.grad_fn)

None

[ ] print(y.grad_fn)

<AddBackward0 object at 0x7e3d6b7e2260>

print(x.grad)
y.backward()
print(x.grad) # dy/dx

None
tensor(0.3150)
```

✓ 0s completed at 12:53 PM


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text

```
<AddBackward0 object at 0x7aaa5c172e30>

[11] print(x.grad)
     y.backward()
     print(x.grad) # dy/dx

None
tensor(0.3150)

x.grad.zero_()
y = 3 * torch.sigmoid(x) + 5
y.backward()
x.grad

tensor(1.2599)

[ ] a = torch.rand(2, 5, requires_grad=True)
```

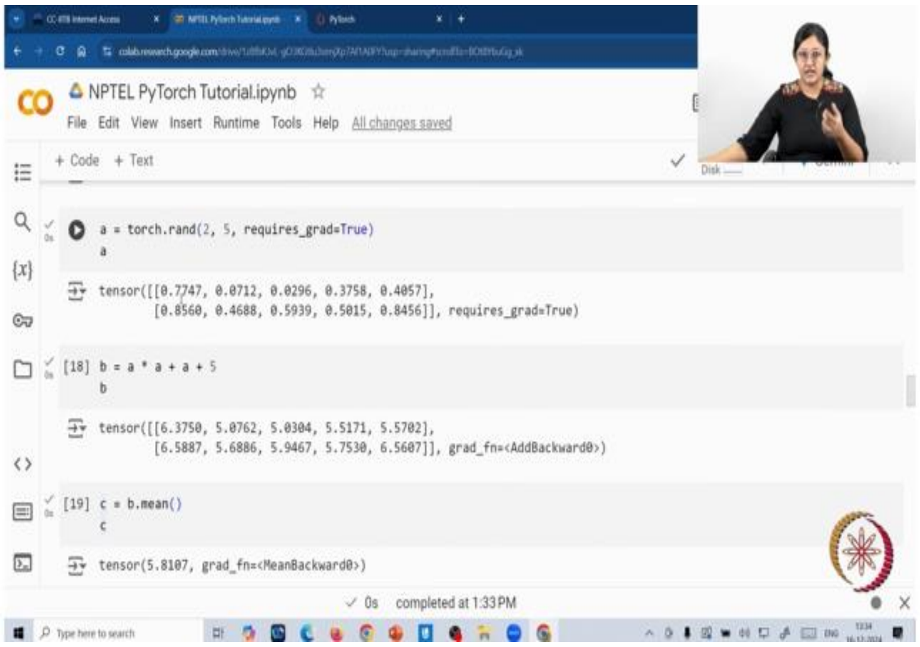
✓ 0s completed at 1:31 PM

Now let's compute the gradient of y with respect to x, which is dy dx. That we can do by simply calling the backward method on y. So we see previously, before calling y.backward, x.grad was none. Now I call backward method, so the gradient computation happened and it got stored in the x.grad attribute, so which we can find over here. So here we got the value 0.3150. When we had done it mathematically, we had got 0.3149. So they are very close. So, yes. Okay. So, another thing that I would mention over here is, so there's this notion that the gradient gets accumulated each time.

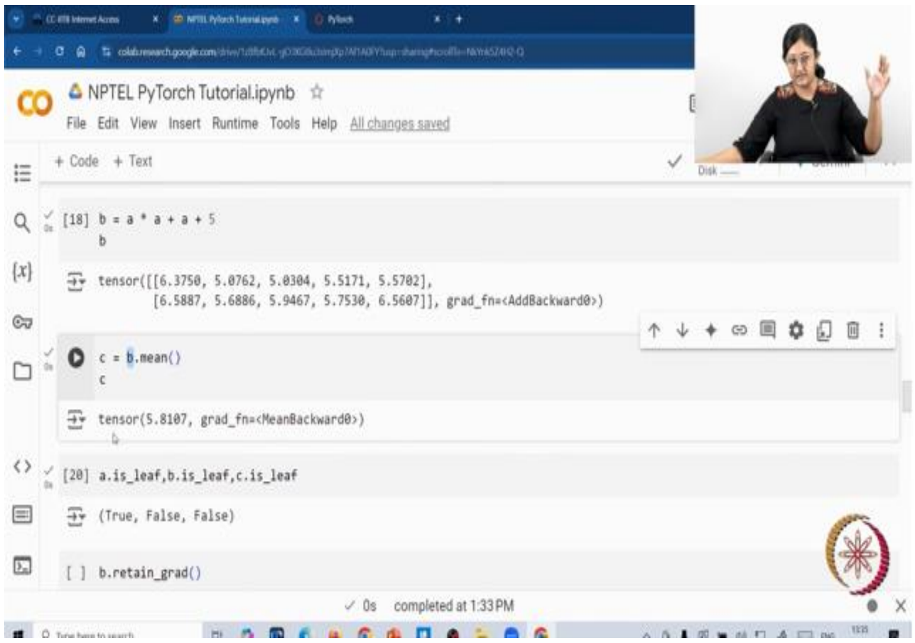
So, we perform this operation. Now, x dot grad holds this value 0.30. Now, again I repeat it. If I repeat it again, again the gradient will get accumulated. So now we see when I look at x.grad again, it is actually this value times 2. So that is what is getting stored. So the gradient gets accumulated at each step. So now I'm doing the same operation again, right? Again, I am calculating the derivative of y with respect to x using this backward method. So, I run it. What happens? As I said, the gradient will get accumulated and now I will get this value, which is 2 times this.

If I run it again, it would again get accumulated. Now, it is 3 times of this and it keeps on going. So, the gradient is getting accumulated at each step. However, this is not desired and if you do not want it, what you can do is, can simply set x.grad, so we can use this, x.grad.0, and what this would do is at each time when you do this, the gradient would get, again, it would be set to 0. So let's check. So now since I ran this line of code, now no longer the gradient is getting accumulated.

So no matter how many times I run this, I'll get 0.3150.

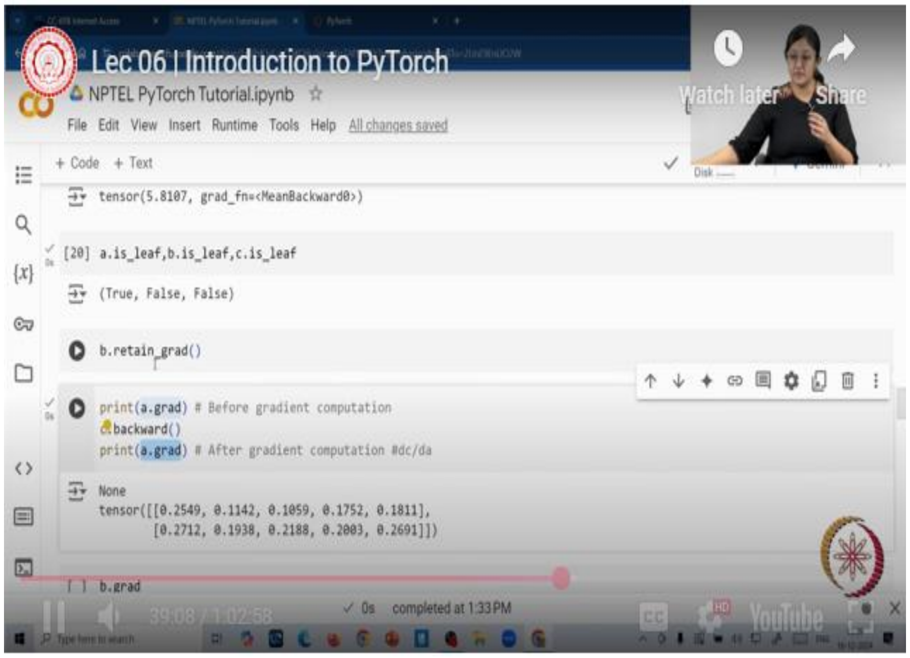

NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
✓ 0s completed at 1:33 PM

```
a = torch.rand(2, 5, requires_grad=True)
a

tensor([[0.7747, 0.0712, 0.0296, 0.3758, 0.4057],
        [0.8560, 0.4688, 0.5939, 0.5015, 0.8456]], requires_grad=True)

[18] b = a * a + a + 5
     b

tensor([[6.3750, 5.0762, 5.0304, 5.5171, 5.5702],
        [6.5887, 5.6886, 5.9467, 5.7530, 6.5607]], grad_fn=<AddBackward0>)

[19] c = b.mean()
     c

tensor(5.8107, grad_fn=<MeanBackward0>)
```


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text

```
[18] b = a * a + a + 5
     b

tensor([[6.3750, 5.0762, 5.0304, 5.5171, 5.5702],
        [6.5887, 5.6886, 5.9467, 5.7530, 6.5607]], grad_fn=<AddBackward0>)

c = b.mean()
c

tensor(5.8107, grad_fn=<MeanBackward0>)

[20] a.is_leaf,b.is_leaf,c.is_leaf

(True, False, False)

[ ] b.retain_grad()
```

✓ 0s completed at 1:33 PM


Lec 06 | Introduction to PyTorch
Watch later
Share
NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
39:08 / 1:02:58
✓ 0s completed at 1:33 PM

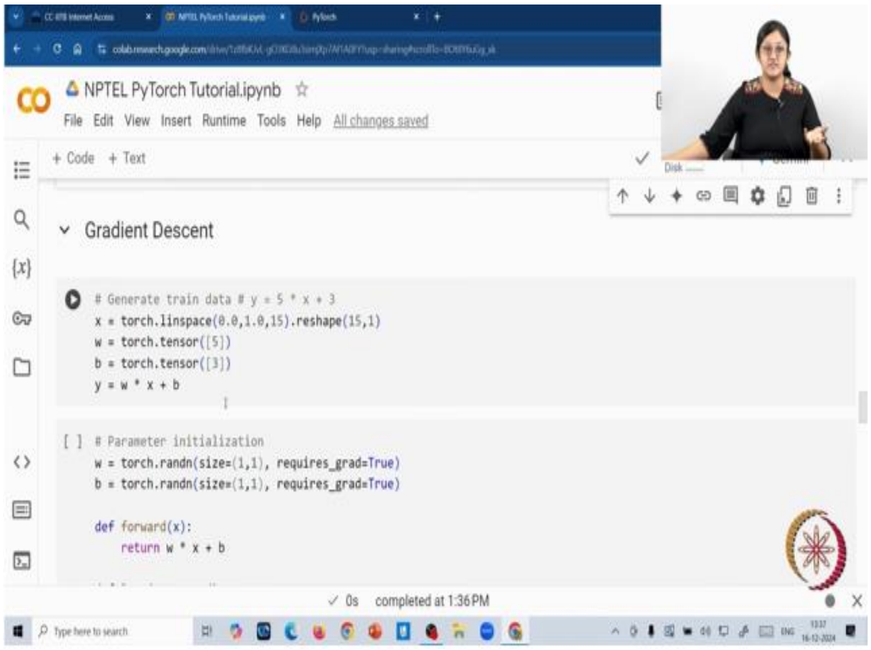
```
tensor(5.8107, grad_fn=<MeanBackward0>)

[20] a.is_leaf,b.is_leaf,c.is_leaf

(True, False, False)

b.retain_grad()

print(a.grad) # Before gradient computation
c.backward()
print(a.grad) # After gradient computation #dc/da

None
tensor([[0.2549, 0.1142, 0.1059, 0.1752, 0.1811],
        [0.2712, 0.1938, 0.2188, 0.2003, 0.2691]])

[ ] b.grad
```

Now let us look at a slightly elaborate example where we first obtain A. So we first create the tensor A. We set its required grab to be true. Then we create B where B is a tensor which is being formed as a result of operation on A. So A would be a leaf tensor but B will be a non-leaf tensor.

Similarly, we have C, where C is nothing but the resultant tensor when I perform the mean operation on B tensor. So, here we have A, B, C. So, that is how the tensors are present in the computational graph. And if I look at the, so as I mentioned, A is the leaf attribute, B is the non-leaf and C is also non-leaf. Now if i simply do c dot backward what would happen is I would now get the derivative of c okay, so which is this and it would get stored in a dot grad. Now, what if so b is a intermediate node over here. So, what can I do? What if I want to compute the gradient of b? So, what to do? So, if I simply do c dot backward and simply do b dot grad in that case we will get this warning that we are calling for the dot grad attribute of a tensor that is a non-leave tensor.

And if you want to access it, what we need to do is we need to set the return grad attribute. So if I want the gradient of B, here what can I do is before running C dot backward, I need to set this.

I need to set B dot retain grad. In that case, I will get the gradient stored in B. So let's see that. So now if I do b.grad I can get the derivative with respect to b as well. So now we can discuss.

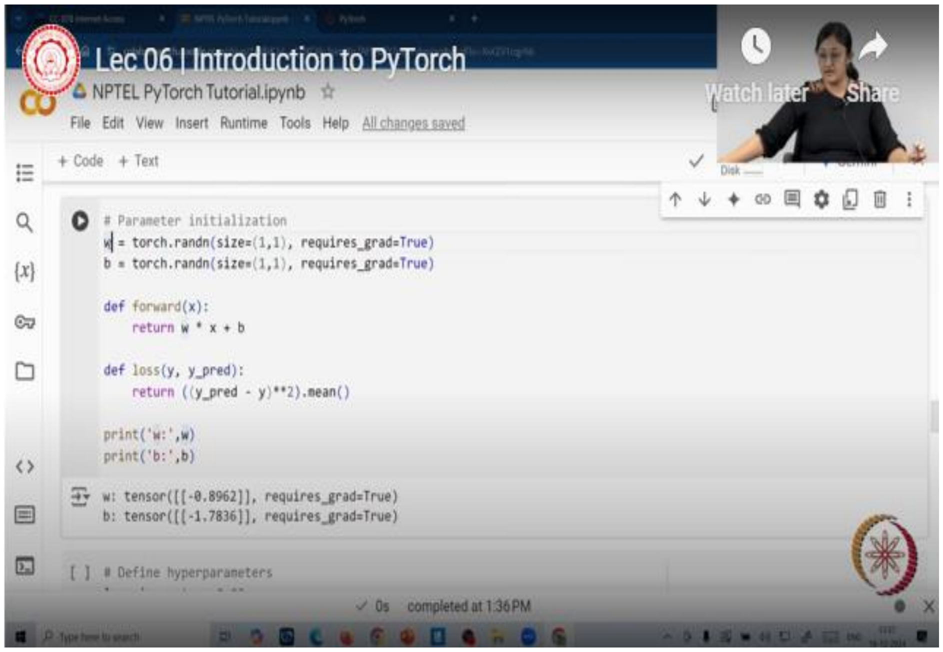

NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text

Gradient Descent

```
# Generate train data # y = 5 * x + 3
x = torch.linspace(0.0,1.0,15).reshape(15,1)
w = torch.tensor([5])
b = torch.tensor([3])
y = w * x + b
```

```
# Parameter initialization
w = torch.randn(size=(1,1), requires_grad=True)
b = torch.randn(size=(1,1), requires_grad=True)

def forward(x):
    return w * x + b
```

✓ 0s completed at 1:36 PM



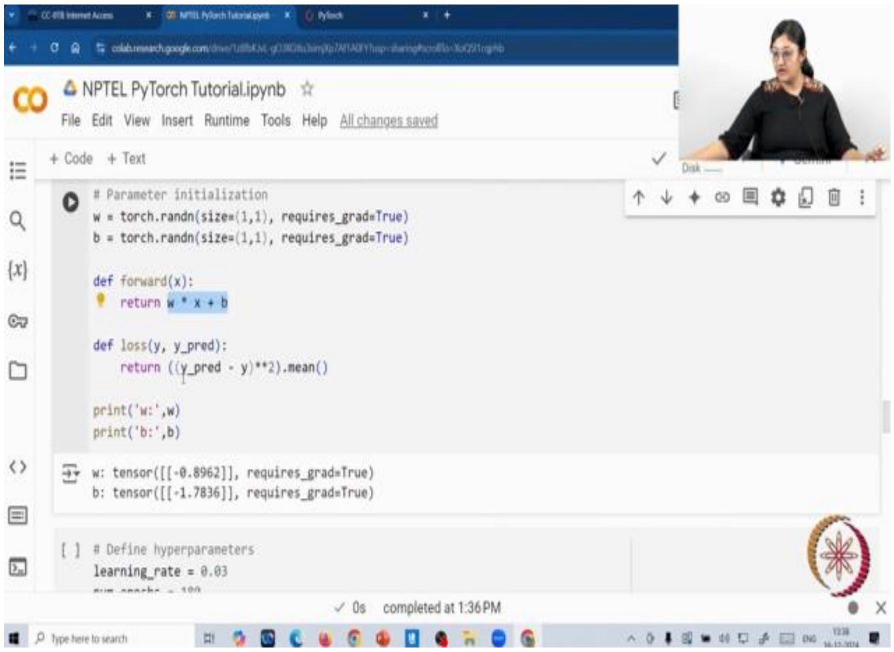
Lec 06 | Introduction to PyTorch

NPTEL PyTorch Tutorial.ipynb

```
# Parameter initialization
w = torch.randn(size=(1,1), requires_grad=True)
b = torch.randn(size=(1,1), requires_grad=True)

def forward(x):
    return w * x + b

def loss(y, y_pred):
    return ((y_pred - y)**2).mean()

print('w:',w)
print('b:',b)
```

```
w: tensor([[-0.8962]], requires_grad=True)
b: tensor([[-1.7836]], requires_grad=True)
```

```
# Define hyperparameters
```

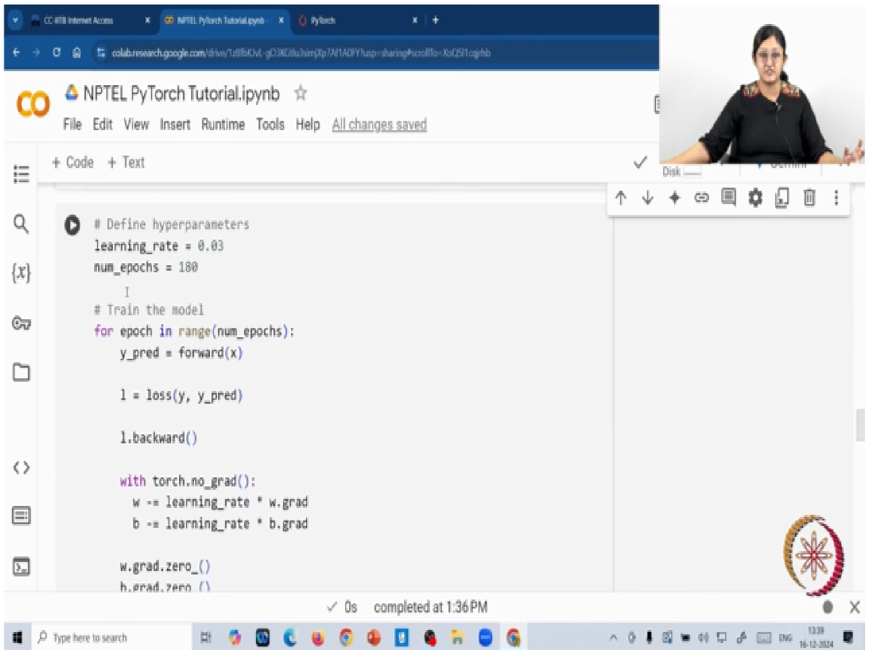


NPTEL PyTorch Tutorial.ipynb

```
# Parameter initialization
w = torch.randn(size=(1,1), requires_grad=True)
b = torch.randn(size=(1,1), requires_grad=True)

def forward(x):
    return w * x + b

def loss(y, y_pred):
    return ((y_pred - y)**2).mean()

print('w:',w)
print('b:',b)
```

```
w: tensor([[-0.8962]], requires_grad=True)
b: tensor([[-1.7836]], requires_grad=True)
```

```
# Define hyperparameters
learning_rate = 0.03
```


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
completed at 1:36PM

```
# Define hyperparameters
learning_rate = 0.03
num_epochs = 180

# Train the model
for epoch in range(num_epochs):
    y_pred = forward(x)

    l = loss(y, y_pred)

    l.backward()

    with torch.no_grad():
      w -= learning_rate * w.grad
      b -= learning_rate * b.grad

    w.grad.zero_()
    b.grad.zero_()
```

So now we understood the basics of autograd. So where in the context of neural network we actually use autograd. So for that this is a toy example again and just for illustration purposes we'll take the example of linear equation, so say we have this equation y is equals to 5x plus 3, we generate our training data following this equation.

So, we first get like the values of x, we define w and b and then we get y. So, this is the input and output of the training data that I just created. Now, using this, I will be training the neural network with the intention that I want the neural network to converge, finally converge at these values of W and B by observing the patterns in this data that we just created. So first we need to initialize the parameters. So we have the parameters w and b since we are dealing with linear regression. So we initialize them and we set requires grad to be true because down the line we would be computing gradients with respect to w and b.

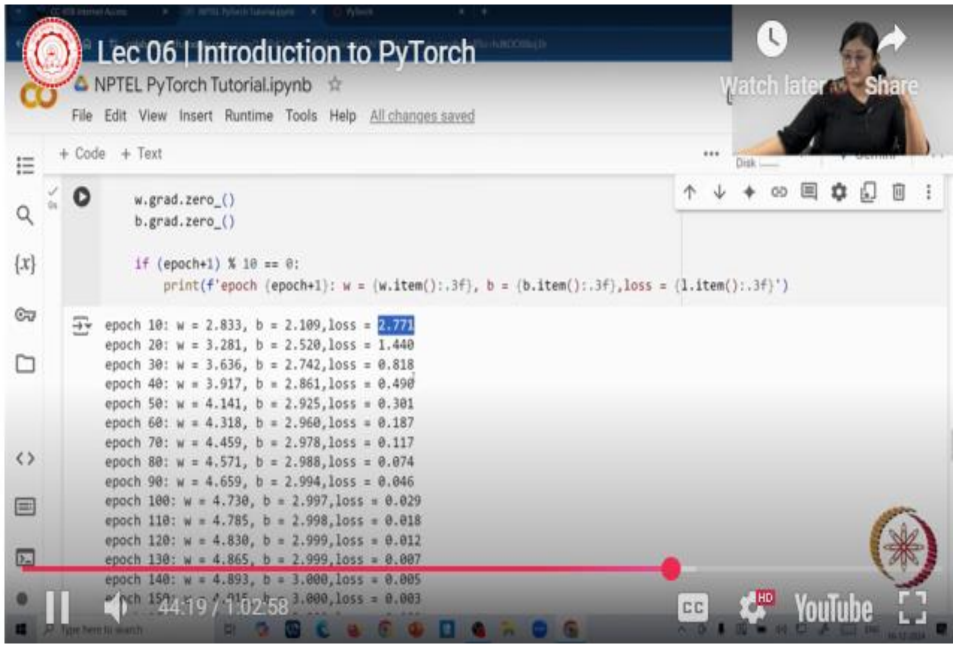
So next there is the forward method. In forward method we define the forward pass. The forward method takes in the input, runs the forward pass gives us the output. So in our example the output would be w times x plus b Then we have the loss method. So the forward method gives us the output or the predicted values. So loss method takes the predicted values, takes the original ground truth values and it computes the loss.

So since here we are dealing with linear regression, the loss is a mean square error loss and this is the formula. Now if I print wnv, we see that wnv has been randomly initialized and the required grad attribute has also been set to true. Now in order to train the neural network, we first set a few parameters which are very common, the learning rate and the number of epochs. Next what we can do is we run the training loop.

So during each iteration, during each epoch, I first do the forward pass. I pass the input to the forward method and I get the predicted y. I use the predicted y along with the ground truth value to compute the loss and then now that I have the loss I use the backward method to compute the gradient of this loss with respect to the parameters of the model that I have that is w and v and that would get stored these gradients that I just computed. So the derivative of the loss with respect to w will get stored in w dot grad. And the derivative of the loss with respect to b will get stored in b.grad. So this is nothing but the gradient descent step where we update the parameters, w and v, using the gradient descent formulation. So here we are doing stochastic gradient descent. So for each of the samples in the training data, parameter update step is happening, the optimization step is happening.

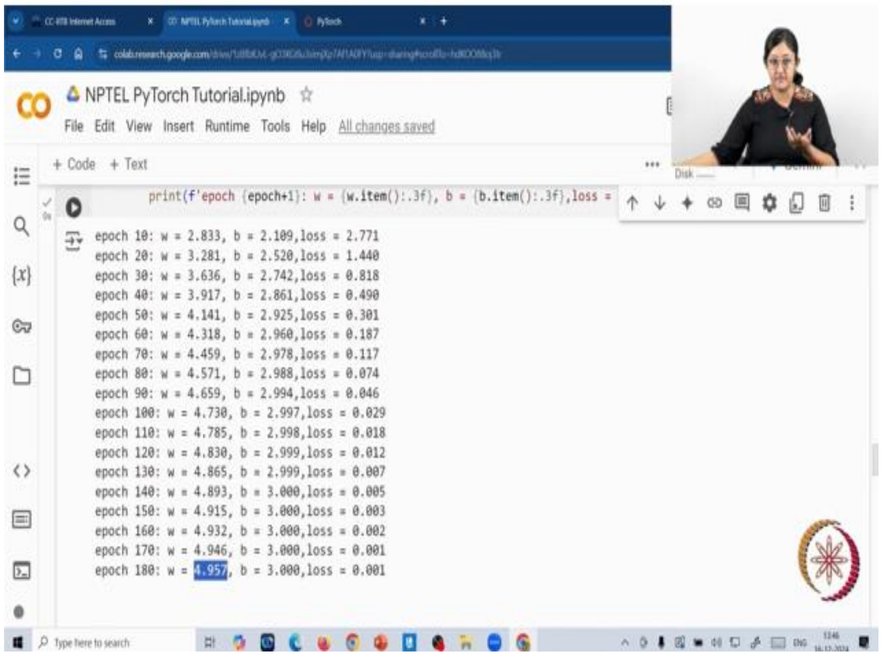
And here we see that we have specified this inside this, so we have given withTorch.noGraph. So this is a context manager. What happens is as I was mentioning, so whenever we have a leaf node like W and B, so these are leaf tensors, so whenever we perform any operation using the leaf tensor it gets tracked in the computational graph. But if we don't want that, we use this torch.nograt and that would disable that for some while.

As long as we are within the scope of this context manager, that would get disabled. So here we simply update the parameters and then we are back again to normal. Here, so we do not want the gradients to get accumulated, so we do w.grad.0 and this would again initialize the gradients to be 0. And this is a common step, so just to observe the progress.


Lec 06 | Introduction to PyTorch
NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Watch later
Share
44:19 / 1:02:58

```
    w.grad.zero_()
    b.grad.zero_()

    if (epoch+1) % 10 == 0:
        print(f'epoch {epoch+1}: w = {w.item():.3f}, b = {b.item():.3f},loss = {l.item():.3f}')
```

```
epoch 10: w = 2.833, b = 2.109,loss = 2.771
epoch 20: w = 3.281, b = 2.520,loss = 1.440
epoch 30: w = 3.636, b = 2.742,loss = 0.818
epoch 40: w = 3.917, b = 2.861,loss = 0.490
epoch 50: w = 4.141, b = 2.925,loss = 0.301
epoch 60: w = 4.318, b = 2.960,loss = 0.187
epoch 70: w = 4.459, b = 2.978,loss = 0.117
epoch 80: w = 4.571, b = 2.988,loss = 0.074
epoch 90: w = 4.659, b = 2.994,loss = 0.046
epoch 100: w = 4.730, b = 2.997,loss = 0.029
epoch 110: w = 4.785, b = 2.998,loss = 0.018
epoch 120: w = 4.830, b = 2.999,loss = 0.012
epoch 130: w = 4.865, b = 2.999,loss = 0.007
epoch 140: w = 4.893, b = 3.000,loss = 0.005
epoch 150: w = 4.915, b = 3.000,loss = 0.003
```

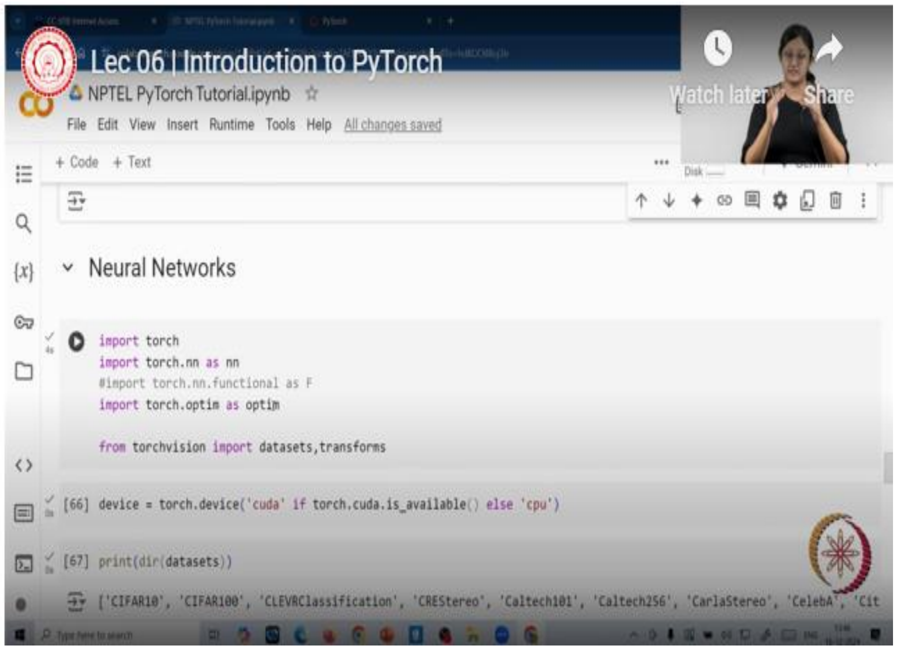

NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text

```
        print(f'epoch {epoch+1}: w = {w.item():.3f}, b = {b.item():.3f},loss =
```

```
epoch 10: w = 2.833, b = 2.109,loss = 2.771
epoch 20: w = 3.281, b = 2.520,loss = 1.440
epoch 30: w = 3.636, b = 2.742,loss = 0.818
epoch 40: w = 3.917, b = 2.861,loss = 0.490
epoch 50: w = 4.141, b = 2.925,loss = 0.301
epoch 60: w = 4.318, b = 2.960,loss = 0.187
epoch 70: w = 4.459, b = 2.978,loss = 0.117
epoch 80: w = 4.571, b = 2.988,loss = 0.074
epoch 90: w = 4.659, b = 2.994,loss = 0.046
epoch 100: w = 4.730, b = 2.997,loss = 0.029
epoch 110: w = 4.785, b = 2.998,loss = 0.018
epoch 120: w = 4.830, b = 2.999,loss = 0.012
epoch 130: w = 4.865, b = 2.999,loss = 0.007
epoch 140: w = 4.893, b = 3.000,loss = 0.005
epoch 150: w = 4.915, b = 3.000,loss = 0.003
epoch 160: w = 4.932, b = 3.000,loss = 0.002
epoch 170: w = 4.946, b = 3.000,loss = 0.001
epoch 180: w = 4.957, b = 3.000,loss = 0.001
```

So as we are training the model, initially we begin with this loss, but as the model is getting trained, the loss is decreasing and the values of w and b is approaching the the actual values.

So W was actually 5, so as a result of training the model it is now 4.957, very close to 5 and B is now 3. So we see that W and B have more or less like converged to the actual values.

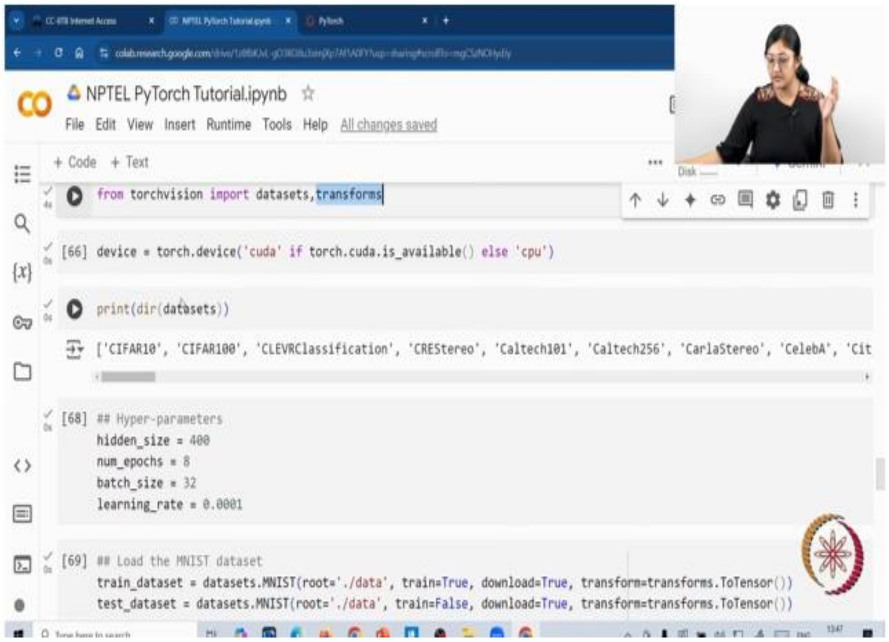

Lec 06 | Introduction to PyTorch
NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Watch later
Share
Neural Networks

```
import torch
import torch.nn as nn
#import torch.nn.functional as F
import torch.optim as optim

from torchvision import datasets,transforms
```

```
[66] device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
[67] print(dir(datasets))
```

```
['CIFAR10', 'CIFAR100', 'CLEVRClassification', 'CREStereo', 'Caltech101', 'Caltech256', 'CarlaStereo', 'CelebA', 'Cit
```


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text

```
from torchvision import datasets,transforms
```

```
[66] device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
print(dir(datasets))
```

```
['CIFAR10', 'CIFAR100', 'CLEVRClassification', 'CREStereo', 'Caltech101', 'Caltech256', 'CarlaStereo', 'CelebA', 'Cit
```

```
[68] ## Hyper-parameters
     hidden_size = 400
     num_epochs = 8
     batch_size = 32
     learning_rate = 0.0001
```

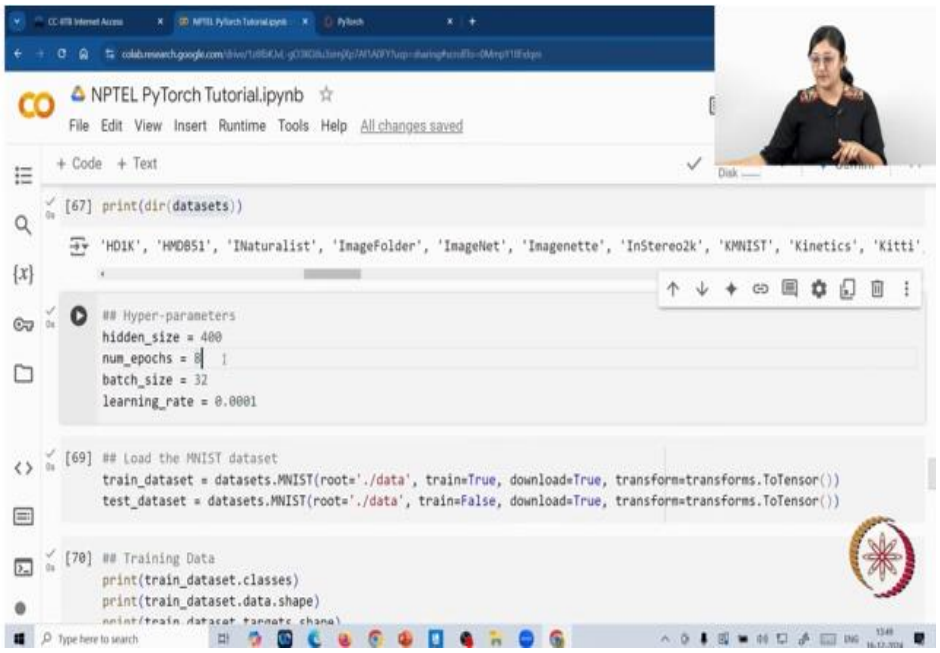
```
[69] ## Load the MNIST dataset
     train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transforms.ToTensor())
     test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transforms.ToTensor())
```

Now we will look at like neural networks. So we'll build a very basic neural network and understand the typical PyTorch pipeline that is followed in order to build and train the model.

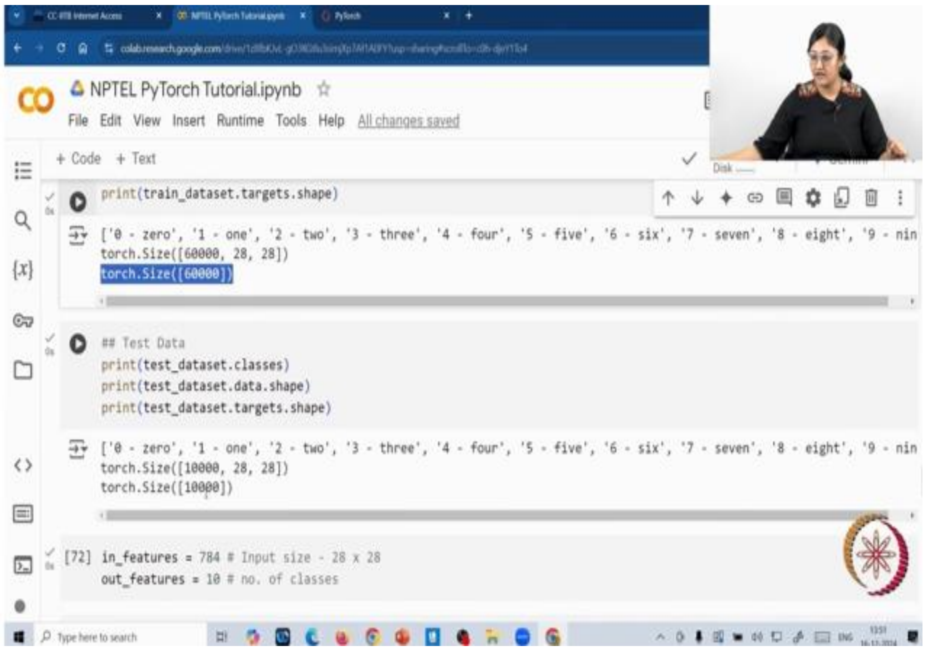
So many of the things that I just explained would again be, again you'll see the same things here as well. So first is we encounter this module which is torch.nn. So whenever we are dealing with neural networks, so however simple or however complex a neural network you want to create, we need the torch.nn module. So everything we need for creating a neural network is present in torch.nn. And so you'll also find different layers if you have come across convolutional layers, linear layers, all these things are present in this module. Also like optimizers, so we can either use gradient descent, we can use Adam, all the various optimizers that are present in Bitorch can be accessed through the Optum module. Now for this example, we'll be using the MNIST dataset, which I think all of you must have heard of. So that is already available in the Torch Vision module. So for that, we are also using this module in the code. So from that, we have imported datasets, to get access to the MNIST dataset and we'll be performing some transformations on the dataset and that is why we need the transforms module as well.

Again, this we already observed, so we are setting the device. Now, if we look at the properties or the elements that are present in the datasets model, we see all these elements. You also have other datasets such as CIFAR-10, CIFAR-100, you will find many more. If you go through this, there are others, many, many datasets available.

One exercise you all can perform is, right now I'm just showing an example using MNIST. You can try a different data set. It would just require some minor modifications in the code and you can also try training a neural network on a different data set.


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text

```
[67] print(dir(datasets))

'HD1K', 'HMDB51', 'INaturalist', 'ImageFolder', 'ImageNet', 'Imagenette', 'InStereo2k', 'KMNIST', 'Kinetics', 'Kitti'

## Hyper-parameters
hidden_size = 400
num_epochs = 8
batch_size = 32
learning_rate = 0.0001

[69] ## Load the MNIST dataset
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transforms.ToTensor())
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transforms.ToTensor())

[70] ## Training Data
print(train_dataset.classes)
print(train_dataset.data.shape)
```



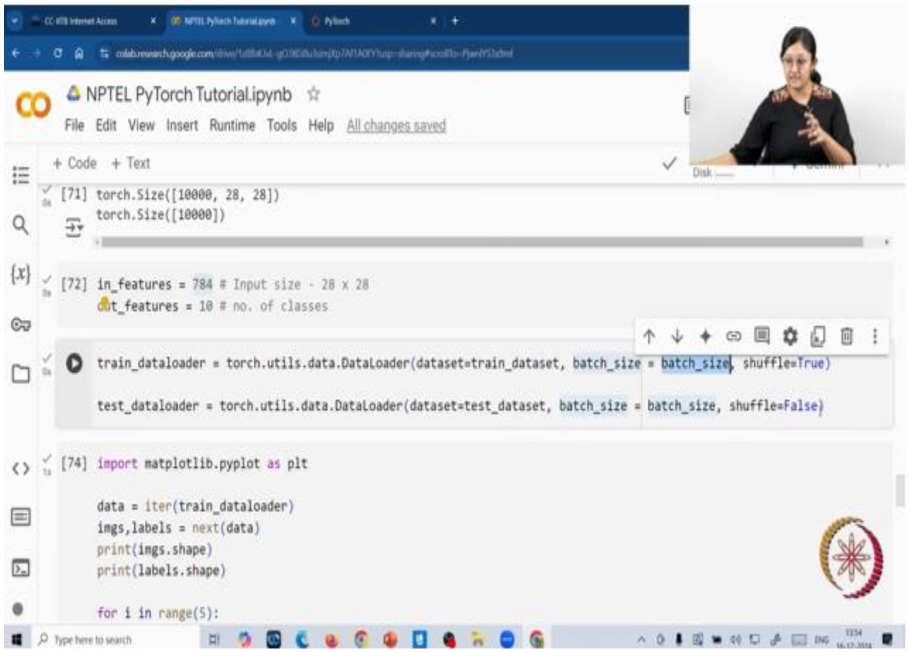
```
print(train_dataset.targets.shape)
```

```
['0 - zero', '1 - one', '2 - two', '3 - three', '4 - four', '5 - five', '6 - six', '7 - seven', '8 - eight', '9 - nin
torch.Size([60000, 28, 28])
torch.Size([60000])
```

```
## Test Data
print(test_dataset.classes)
print(test_dataset.data.shape)
print(test_dataset.targets.shape)
```

```
['0 - zero', '1 - one', '2 - two', '3 - three', '4 - four', '5 - five', '6 - six', '7 - seven', '8 - eight', '9 - nin
torch.Size([10000, 28, 28])
torch.Size([10000])
```

```
[72] in_features = 784 # Input size - 28 x 28
     out_features = 10 # no. of classes
```



```
[71] torch.Size([10000, 28, 28])
     torch.Size([10000])
```

```
[72] in_features = 784 # Input size - 28 x 28
     out_features = 10 # no. of classes
```

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

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

```
[74] import matplotlib.pyplot as plt

     data = iter(train_dataloader)
     imgs,labels = next(data)
     print(imgs.shape)
     print(labels.shape)

     for i in range(5):
```

So first you can set the hyperparameters. So here we'll be creating a neural network, very simple neural network with only one hidden layer.

So here we set the hidden size. So hidden size is the number of neurons that is present in the hidden layer of the neural network. So there would be the input, the output, and in between there would be one hidden layer with 400 neurons.

We'd be running the training for eight epochs. Batch size is 32. We'll look at the batch size in a while. And learning rate is this. We can play around with these values and see what changes is happening in the training is increasing or decreasing the learning rate leading to faster or slower convergence? Or is it diverging? Or what does changing the batch size do? So you can vary all these things and you can explore further. So here we load the MNIST dataset.

So datasets.mnist, we specify the root. Then for training dataset, we set train to be true, whereas for test, we set train to be false, quite intuitively. We download it. Now the dataset needs to be like, needs to be tensors. So that is where we perform the transformation.

Transforms are two tensors. As a result, the data set that we have now would have tensor values. So we have loaded the data set, so that is done. Next, what we need to do, let's just observe what is present in the data set. So we can look at the classes present in the training data set. So this is MNIST data set, so we know that this deals with images of numbers from zero to nine, where given an image, the neural network needs to identify what is the number, what is the number that is present in the image.

So here we see the classes are from 0 to 9. So there are 10 classes. The shape of the trained data set is 60,000, 28, 28. 60,000 is the number of samples, the number of images in the training data set. And 28, 28 is the size of each image. Okay, so this is the shape of the data and this is corresponding to the 60,000 samples, there are 60,000 labels as well.

Similarly for test data, everything is same, just the number of test samples is 10,000. So, from here we can understand something. So, when we want to construct a neural network, we need to specify what is the input, what is the dimension of the input that we are passing and what is the dimension of the output we expect from the neural network. So, since there are 10 classes, the output features would be 10 in number, whereas the input features would be 784. 784 is nothing but 28 cross 28. So, instead of directly passing a tensor of size 28 cross 28, we flatten it and then we pass it as a 784 size tensor to the neural network.

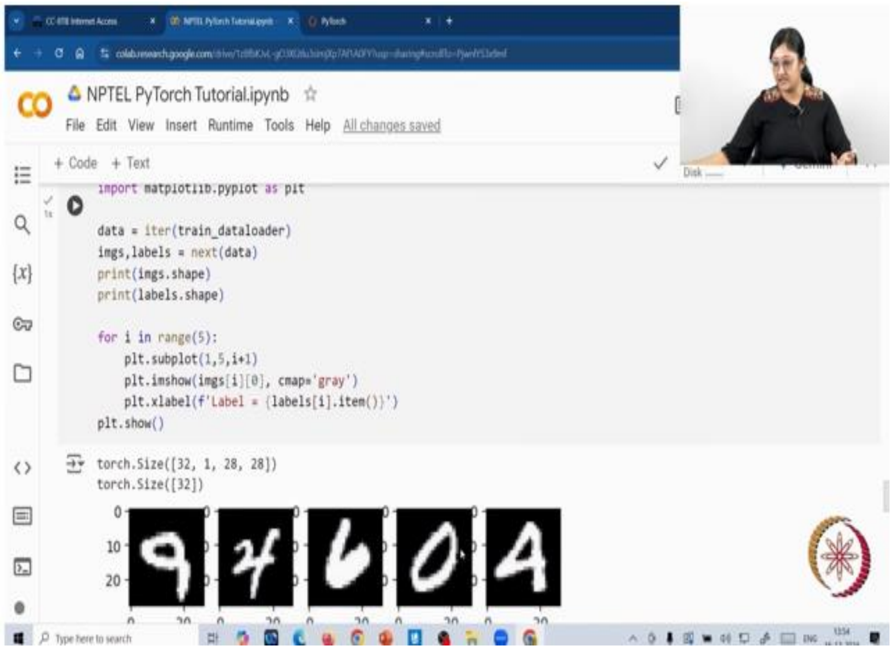
We'll understand this in case anybody has a doubt in this as we go through the designing of the model and I think these things will be clear. Now that we have observed the data, so the number of elements, the number of classes, size of the image. Now we can look at data loaders. So as we saw that the training data set has 60,000 samples, the test data set has 10,000 samples.

So we cannot load all of them into memory at once when we are performing the training or when we are doing the evaluation. So here the idea of data loaders comes. So what data loaders do is they give us a batch of the training. So here as we remember, we had set the batch size to 32.

So for each call of the data loader, I'll get samples as many as the batch size. So I'll get 32 randomly selected samples of the training data when I call train.dataloader. And so here we can see, I need to pass the data set. These are the arguments to the data loader.

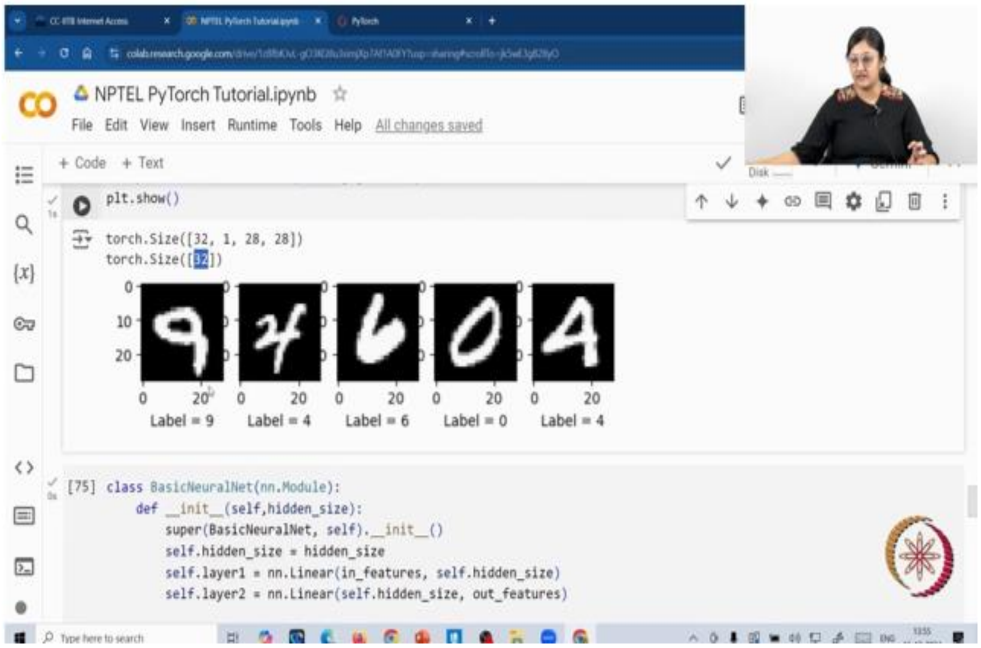
The data set I pass the train underscore data set that I have. I specify the batch size and for train data loader, I want the elements in the batch to be random elements. So I set shuffle to be true. So I want 32 random elements to be selected from the training dataset and supplied to me for each batch.

However that is not required for test. So for test I don't need to shuffle. So this is the concept of data loader.


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text

```
import matplotlib.pyplot as plt

data = iter(train_dataloader)
imgs,labels = next(data)
print(imgs.shape)
print(labels.shape)

for i in range(5):
    plt.subplot(1,5,i+1)
    plt.imshow(imgs[i][0], cmap='gray')
    plt.xlabel(f'Label = {labels[i].item()}')
plt.show()
```

```
torch.Size([32, 1, 28, 28])
torch.Size([32])
```


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text

```
plt.show()
```

```
torch.Size([32, 1, 28, 28])
torch.Size([32])
```

Label = 9
Label = 4
Label = 6
Label = 0
Label = 4

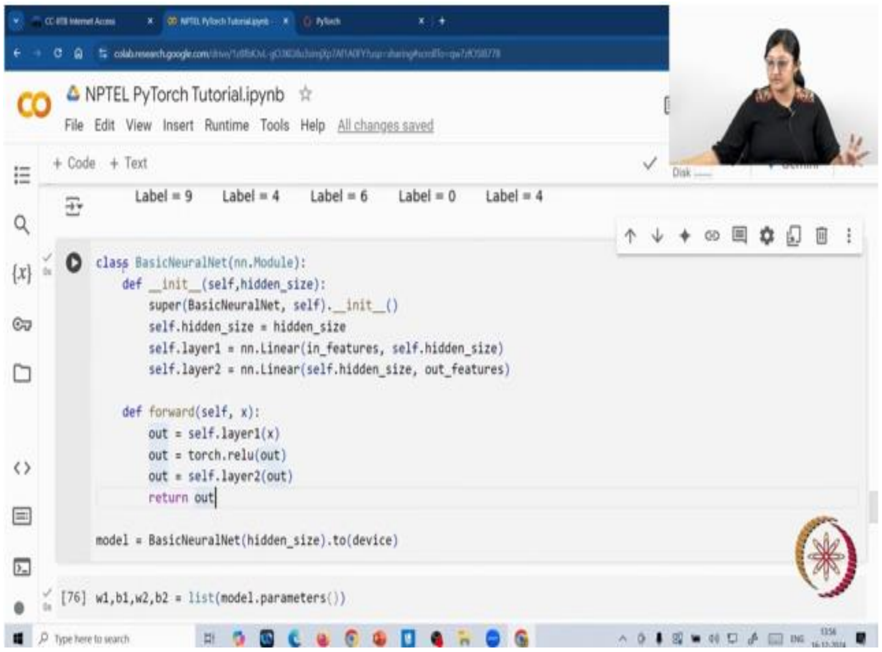
```
[75] class BasicNeuralNet(nn.Module):
         def __init__(self,hidden_size):
             super(BasicNeuralNet, self).__init__()
             self.hidden_size = hidden_size
             self.layer1 = nn.Linear(in_features, self.hidden_size)
             self.layer2 = nn.Linear(self.hidden_size, out_features)
```

Next is I can look at a few images of what it looks like, what the data set looks like. So here I just first take the data loader, create an iterator from it, I do next data.

And that would give me again like 32 samples from the training data set. I unpack them into images and labels. I can check the shape over here. This is the shape. As you can see, we have batch size number of images and corresponding to 32 images at the 32 labels. Now I can look at a few, like five of them. As I can see over here, this is the image that I have and the corresponding label. This looks 9 and the label is 9 and 4 and 4.

This could also be a basic sanity check where you can look at a few random samples of the data and you can see whether the image and the label align with each other or not. So we have looked at the data. We have figured out what is the input and output dimension to the neural network.

Now all we need to do is design the neural network, design the model.

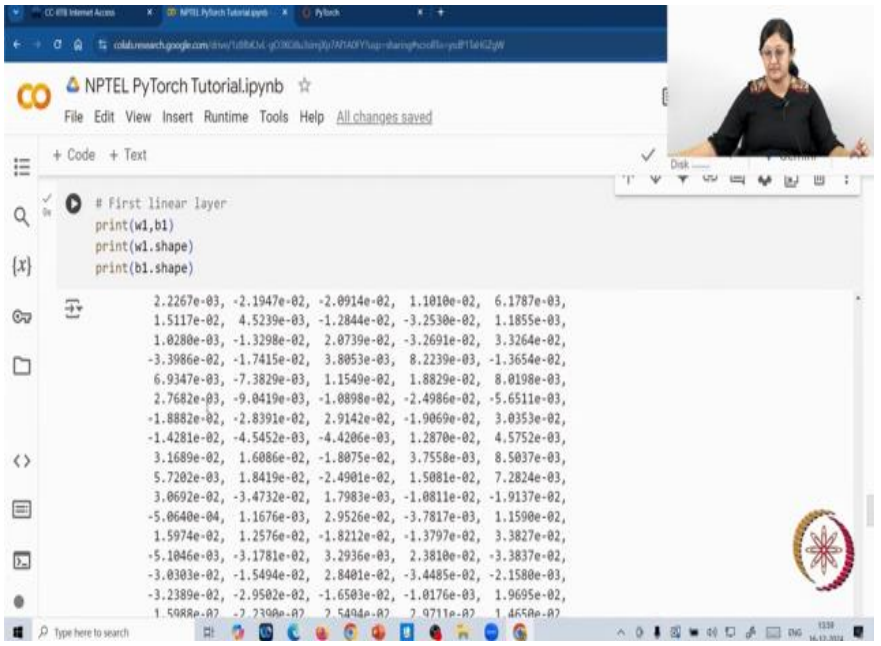


NPTEL PyTorch Tutorial.ipynb

```
Label = 9    Label = 4    Label = 6    Label = 0    Label = 4
```

```python
class BasicNeuralNet(nn.Module):
    def __init__(self,hidden_size):
        super(BasicNeuralNet, self).__init__()
        self.hidden_size = hidden_size
        self.layer1 = nn.Linear(in_features, self.hidden_size)
        self.layer2 = nn.Linear(self.hidden_size, out_features)

    def forward(self, x):
        out = self.layer1(x)
        out = torch.relu(out)
        out = self.layer2(out)
        return out

model = BasicNeuralNet(hidden_size).to(device)
```

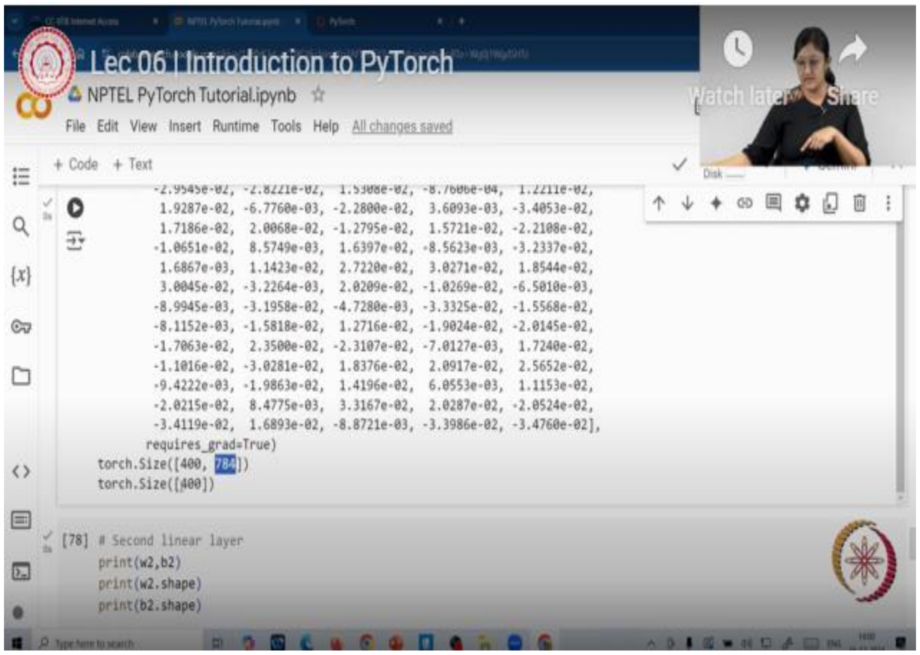
```python
[76] w1,b1,w2,b2 = list(model.parameters())
```



NPTEL PyTorch Tutorial.ipynb

```python
# First linear layer
print(w1,b1)
print(w1.shape)
print(b1.shape)
```

```
         2.2267e-03, -2.1947e-02, -2.0914e-02,  1.1010e-02,  6.1787e-03,
         1.5117e-02,  4.5239e-03, -1.2844e-02, -3.2530e-02,  1.1855e-03,
         1.0280e-03, -1.3298e-02,  2.0739e-02, -3.2691e-02,  3.3264e-02,
        -3.3986e-02, -1.7415e-02,  3.8053e-03,  8.2239e-03, -1.3654e-02,
         6.9347e-03, -7.3829e-03,  1.1549e-02,  1.8829e-02,  8.0198e-03,
         2.7682e-03, -9.0419e-03, -1.0898e-02, -2.4986e-02, -5.6511e-03,
        -1.8882e-02, -2.8391e-02,  2.9142e-02, -1.9069e-02,  3.0353e-02,
        -1.4281e-02, -4.5452e-03, -4.4206e-03,  1.2870e-02,  4.5752e-03,
         3.1689e-02,  1.6086e-02, -1.8075e-02,  3.7558e-03,  8.5037e-03,
         5.7202e-03,  1.8419e-02, -2.4901e-02,  1.5081e-02,  7.2824e-03,
         3.0692e-02, -3.4732e-02,  1.7983e-03, -1.0811e-02, -1.9137e-02,
        -5.0640e-04,  1.1676e-03,  2.9526e-02, -3.7817e-03,  1.1590e-02,
         1.5974e-02,  1.2576e-02, -1.8212e-02, -1.3797e-02,  3.3827e-02,
        -5.1046e-03, -3.1781e-02,  3.2936e-03,  2.3810e-02, -3.3837e-02,
        -3.0303e-02, -1.5494e-02,  2.8401e-02, -3.4485e-02, -2.1580e-03,
        -3.2389e-02, -2.9502e-02, -1.6503e-02, -1.0176e-03,  1.9695e-02,
         1.5988e-02, -2.2390e-02,  2.5494e-02,  2.9711e-02,  1.4650e-02
```


Lec 06 | Introduction to PyTorch
NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Watch later
Share

```
        -2.9545e-02, -2.8221e-02,  1.5308e-02, -8.7606e-04,  1.2211e-02,
         1.9287e-02, -6.7760e-03, -2.2800e-02,  3.6093e-03, -3.4053e-02,
         1.7186e-02,  2.0068e-02, -1.2795e-02,  1.5721e-02, -2.2108e-02,
        -1.0651e-02,  8.5749e-03,  1.6397e-02, -8.5623e-03, -3.2337e-02,
         1.6867e-03,  1.1423e-02,  2.7220e-02,  3.0271e-02,  1.8544e-02,
         3.0045e-02, -3.2264e-03,  2.0209e-02, -1.0269e-02, -6.5010e-03,
        -8.9945e-03, -3.1958e-02, -4.7280e-03, -3.3325e-02, -1.5568e-02,
        -8.1152e-03, -1.5818e-02,  1.2716e-02, -1.9024e-02, -2.0145e-02,
        -1.7063e-02,  2.3500e-02, -2.3107e-02, -7.0127e-03,  1.7240e-02,
        -1.1016e-02, -3.0281e-02,  1.8376e-02,  2.0917e-02,  2.5652e-02,
        -9.4222e-03, -1.9863e-02,  1.4196e-02,  6.0553e-03,  1.1153e-02,
        -2.0215e-02,  8.4775e-03,  3.3167e-02,  2.0287e-02, -2.0524e-02,
        -3.4119e-02,  1.6893e-02, -8.8721e-03, -3.3986e-02, -3.4760e-02],
       requires_grad=True)
torch.Size([400, 784])
torch.Size([400])
```

```
[78] # Second linear layer
     print(w2,b2)
     print(w2.shape)
     print(b2.shape)
```

So here we need to define a class for that.

So in the class, I named the class. You can name it anything you want. I've named basic neural net. And this needs to inherit the nn.module class. So this would be a subclass of this class. So, nn.module has all the basic functionalities that you need to design a neural network. So, all we need to simply do now is just define the constructor and the constructor will be just defining the skeleton of the neural network and in the forward method I would be like defining the forward pass, rest everything else would be handled internally by nn.module. So here I just simply call the constructor of the base class, then pass the hidden size. So one thing is when I'm defining the function, so here self is mandatory, you need to mandatorily pass self, rest, other arguments can be anything, you can pass hidden size, you can pass input size, output size, anything, this you can arbitrarily set.

For forward you need to pass self and x, which is the input. So then if I talk about the layers that the model has, so I have two linear layers, one mapping the input to the hidden layer, another from the hidden layer to the output.

So this would be, how much was the hidden size that I had set? Let me check. It's 400. Input size was 784, right? Which was 28 cross 28. So this would take, the input features would be 784. Then from 784, this linear layer would give me the hidden size, so 400. Then again from 400, I get output features, which is 10. So that is how it is happening. Now in the forward pass, it takes the input and gives me the output of the neural network.

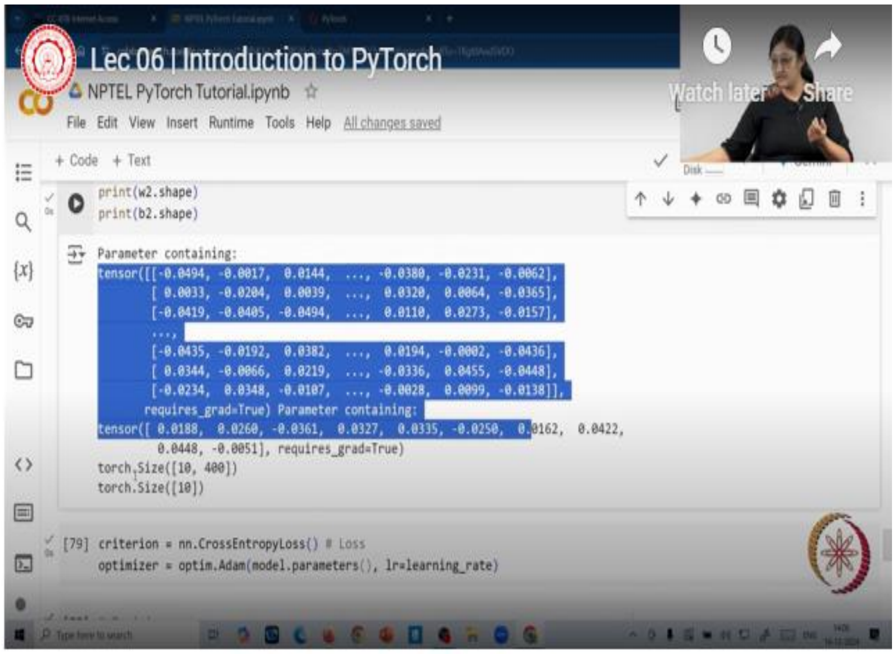
What would happen if I pass any input through the neural network? What would happen? So first I take the input, I pass it through my first layer and that gives me output. So that I pass through a non-linearity.

In this case, I'm passing it through ReLU. Again, the output of this is fed to the output of the second linear ray. And this would be my final output of the forward pass. So one thing is... One thing is, since this is a multi-class classification, we'll be using cross-entropy loss during training the neural network. So in cross-entropy loss, we can simply pass the raw value. We are not doing any further operation out here, further passing it through another activation layer, we're not doing anything of that sort, we're not passing it through another softmax. We're directly taking the raw values and we'll supply it to cross entropy loss and it would do everything, the function would handle everything.

Then I create the neural network, I simply pass the hidden size, and then I move the model to the device, so to the CA, to the GPO, if it is available. Now, as you saw, we created two linear layers. So linear layers has a weight and a bias, so we can look at that. So we can, once we have generated the model, I can look at model parameters, and that would give me what are the weights and biases. So here we can look at it. So for the first linear layer, we can look at the weight and bias. So this is the weight and bias, as you can see. This is the weight and bias. And you can see that, just a second, and you can see that, see, you can see the requires grad is automatically set to true.

We do not have to explicitly mention it anywhere. We just mentioned the size of the linear layer. And these things got handled internally. So both of them, the requires grad attribute is true as these are parameters of the model and we need to compute the gradient of loss with respect to these parameters.

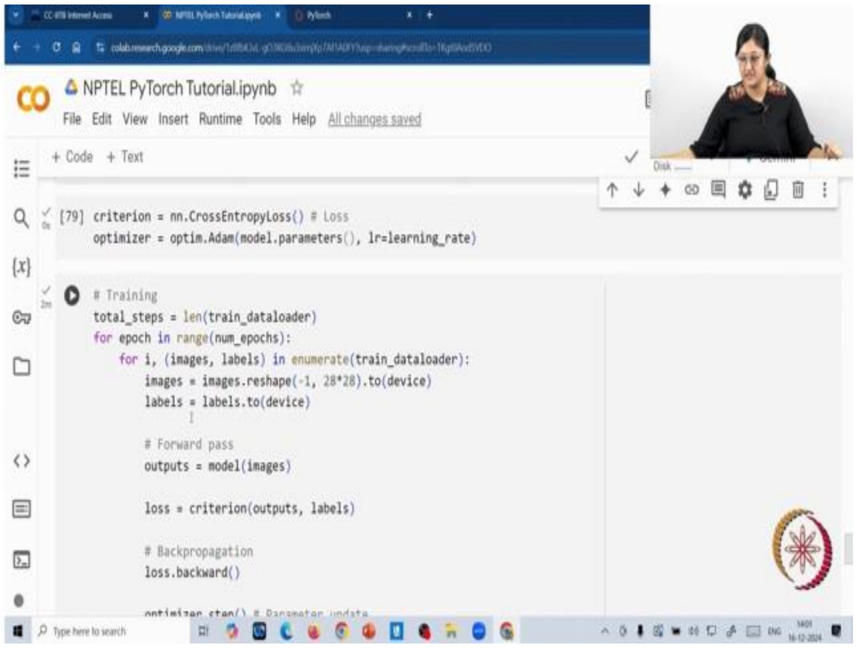
And also you can look at the shape. So this is hidden size and this is the input size. So this is the size of the weight parameter and this is the size of the bias parameter. So this is bias tensor.


Lec 06 | Introduction to PyTorch
NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Watch later
Share

```
print(w2.shape)
print(b2.shape)
```

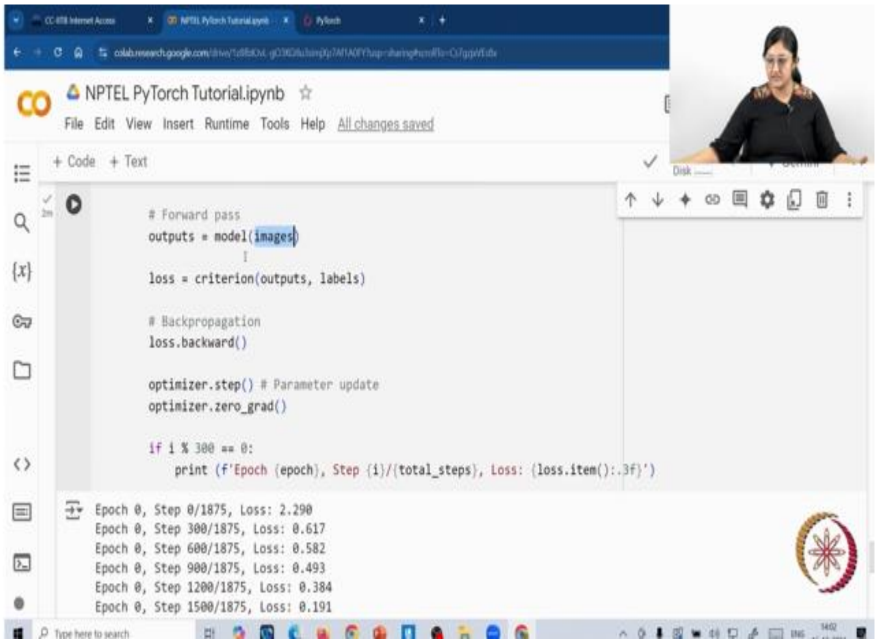
```
Parameter containing:
tensor([[-0.0494, -0.0017,  0.0144,  ..., -0.0380, -0.0231, -0.0062],
        [ 0.0033, -0.0204,  0.0039,  ...,  0.0320,  0.0064, -0.0365],
        [-0.0419, -0.0405, -0.0494,  ...,  0.0110,  0.0273, -0.0157],
        ...,
        [-0.0435, -0.0192,  0.0382,  ...,  0.0194, -0.0002, -0.0436],
        [ 0.0344, -0.0066,  0.0219,  ..., -0.0336,  0.0455, -0.0448],
        [-0.0234,  0.0348, -0.0107,  ..., -0.0028,  0.0099, -0.0138]],
       requires_grad=True) Parameter containing:
tensor([ 0.0188,  0.0260, -0.0361,  0.0327,  0.0335, -0.0250,  0.0162,  0.0422,
         0.0448, -0.0051], requires_grad=True)
torch.Size([10, 400])
torch.Size([10])
```

```
[79] criterion = nn.CrossEntropyLoss() # Loss
     optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```


NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text

```
[79] criterion = nn.CrossEntropyLoss() # Loss
     optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

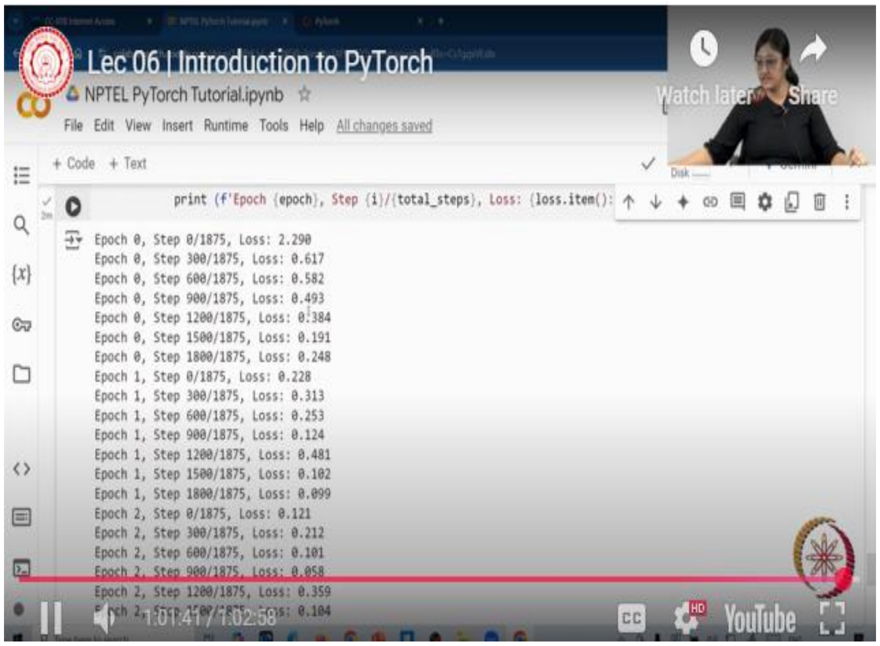
```
# Training
total_steps = len(train_dataloader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_dataloader):
        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)

        loss = criterion(outputs, labels)

        # Backpropagation
        loss.backward()

        optimizer.step() # Parameter update
```



NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text

```
        # Forward pass
        outputs = model(images)

        loss = criterion(outputs, labels)

        # Backpropagation
        loss.backward()

        optimizer.step() # Parameter update
        optimizer.zero_grad()

        if i % 300 == 0:
            print (f'Epoch {epoch}, Step {i}/{total_steps}, Loss: {loss.item():.3f}')
```

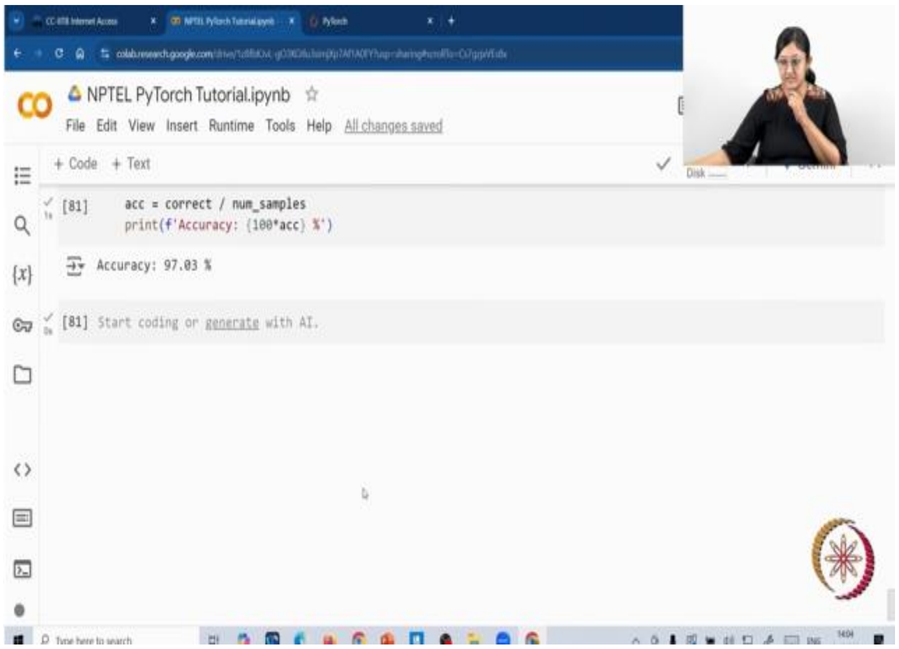
```
Epoch 0, Step 0/1875, Loss: 2.290
Epoch 0, Step 300/1875, Loss: 0.617
Epoch 0, Step 600/1875, Loss: 0.582
Epoch 0, Step 900/1875, Loss: 0.493
Epoch 0, Step 1200/1875, Loss: 0.384
Epoch 0, Step 1500/1875, Loss: 0.191
```



Lec 06 | Introduction to PyTorch
NPTEL PyTorch Tutorial.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Watch later
Share
1:01:41 / 1:02:58

```
            print (f'Epoch {epoch}, Step {i}/{total_steps}, Loss: {loss.item():
```

```
Epoch 0, Step 0/1875, Loss: 2.290
Epoch 0, Step 300/1875, Loss: 0.617
Epoch 0, Step 600/1875, Loss: 0.582
Epoch 0, Step 900/1875, Loss: 0.493
Epoch 0, Step 1200/1875, Loss: 0.384
Epoch 0, Step 1500/1875, Loss: 0.191
Epoch 0, Step 1800/1875, Loss: 0.248
Epoch 1, Step 0/1875, Loss: 0.228
Epoch 1, Step 300/1875, Loss: 0.313
Epoch 1, Step 600/1875, Loss: 0.253
Epoch 1, Step 900/1875, Loss: 0.124
Epoch 1, Step 1200/1875, Loss: 0.481
Epoch 1, Step 1500/1875, Loss: 0.102
Epoch 1, Step 1800/1875, Loss: 0.099
Epoch 2, Step 0/1875, Loss: 0.121
Epoch 2, Step 300/1875, Loss: 0.212
Epoch 2, Step 600/1875, Loss: 0.101
Epoch 2, Step 900/1875, Loss: 0.058
Epoch 2, Step 1200/1875, Loss: 0.359
Epoch 2, Step 1500/1875, Loss: 0.104
```



```
acc = correct / num_samples
print(f'Accuracy: {100*acc} %')
```

```
Accuracy: 97.03 %
```

Next we can look at the second linear layer as well. So here we have, so this is again similarly, it's very similar. So now it's 10 which is the number of output features and this is the hidden layer and this is the shape of the bias.

So right now they are all like garbled up values. Now we need to train the neural network and we want the neural network to, given an image, predict what is the number associated with it. So as I mentioned, we're using cross entropy loss as the loss function over here.

For optimizer, we're using Adam. There are various others available. Please check the Optum module for that. And so here we are passing nothing but the model parameters and the learning rate, which we had defined a while back. Here I'm doing batched stochastic gradients. So there would be two for loops, one for loop to keep track of the epoch, one for loop to keep track of the epoch, and another for loop is where I am calling the data loader and it's giving me samples of the batch size.

And then I again unpack them. So this is images and labels. I reshape the image as my input linear layer had, it was of shape 784 and 400. So I need to flatten the tensor. So that is what I'm doing. I'm flattening the tensor. This is the input to the model.

And these are the ground truth labels. Since I moved the model to the device, these also need to be moved to the GPU. Next, I do the forward pass. The input in this case are the images which have been flattened now. I'm passing it through the model. When I pass it through the model, internally the forward method would be called.

The forward pass would be performed and I would get the output. So this is the predicted value. I pass the predicted value and the ground truth labels to the loss function. I get the loss. And now it's the time for the backward pass. I need to compute the gradient of the loss with respect to the parameters of the network, which is rate and bias. So, I do loss dot backward. So, now once the gradients have been computed, I need to update the weight parameters.

Now, here I do not need to explicitly mention the gradient descent step, neither do I have to explicitly like write I do not have to explicitly write the context manager torch.nograd. Then I don't have to explicitly update the weights and parameters, weights and biases. Internally, it is done by this method.

So whenever I do optimizer.step, the parameter update happens. And this takes in the current states of the parameters, it takes the current gradients and using that it updates the weights and parameters appropriately. And then again we do not want the gradients to get accumulated, so at the end of each iteration we set it to 0 so that from next time it starts afresh.

So again, this is the basic. We are just looking at the progress. So we see that the loss is decreasing. It's also fluctuating in between.

Ultimately, this is after training for like seven, sorry, eight epochs, I think. After training for 8 epochs, this is what we have. Now we can simply check the accuracy that we have. So accuracy is we get 97.03. So that is good enough accuracy. You can further play around with the hyperparameters and you can see whether you are able to get a better accuracy. So this was more or less the basics of PyTorch, the things you need to get started if you want to walk on any deep learning model or anything.

So I hope now the basic foundational concepts related to PyTorch are clear. So students have the foundational concepts and now they can design any deep learning model. They can easily get started with anything. So thank you for listening and happy learning.