



Applied Accelerated Artificial Intelligence
Dr. Satyajit Das
Department of Computer Science and Engineering
Indian Institute of Technology, Palakkad


Lecture - 19
Introduction to PyTorch Part - 2

(Refer Slide Time: 00:14)





Training procedure

- Define the neural network
- Iterate over a dataset of inputs
- Process input through the network
- Compute the loss
- Propagate gradients back into the network's parameters
- Update the weights of the network



So, first we will see how to define one neural network in PyTorch right.


(Refer Slide Time: 00:18)



Build Neural Networks using PyTorch

Neural networks can be constructed using the torch.nn package.

- Forward
 - An nn.Module contains layers, and
 - A method forward(input) that returns the output
 - You can use any of the Tensor operations in the forward function
- Backward
 - nn depends on autograd
 - You just have to define the forward function



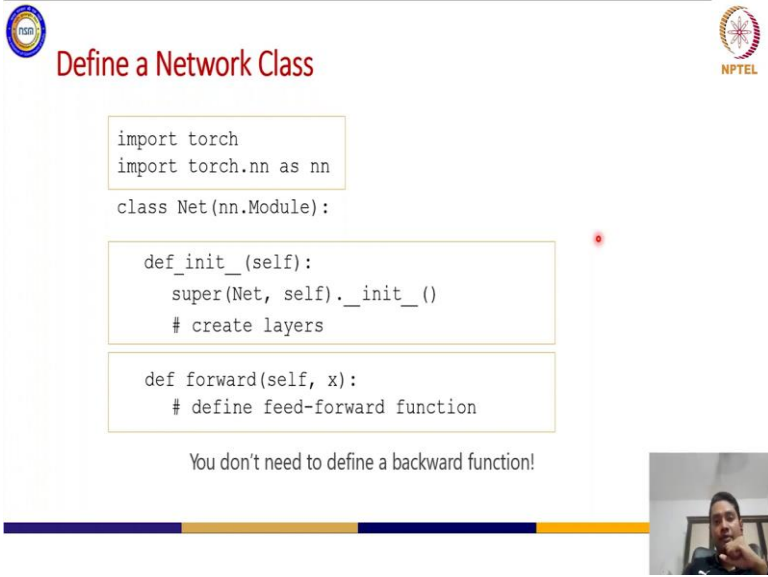
So, neural networks can be constructed with torch.nn package. So, basically nn.Module this class contains all the layers that you can have. So, as I was mentioning that using PyTorch block wise or module wise you can define your networks, your modules DNN modules very efficiently

So, nn dot module has all the layers predefined functions as the wrapper, function and you can use them you can inherit this class, you can define in your own class and you can extend this functions to define your own customized layers for your nn modules.

So, what are the steps you remember? So, this is where you will pass your input with a forward pass. So, forward pass basically from the input to your output of the neural network and backward pass is essentially when you want to compute the gradient and you want to update the parameters ok.

So, forward pass is very very simple because these are just a sequence of layers and backward pass is essentially computing the gradients with a back propagation algorithm and that is the most complex operation. But it will be you will see that how to define one backward function is in the easiest thing you can do in PyTorch.

(Refer Slide Time: 01:51)



Define a Network Class

```
import torch
import torch.nn as nn

class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # create layers

    def forward(self, x):
        # define feed-forward function
```

You don't need to define a backward function!

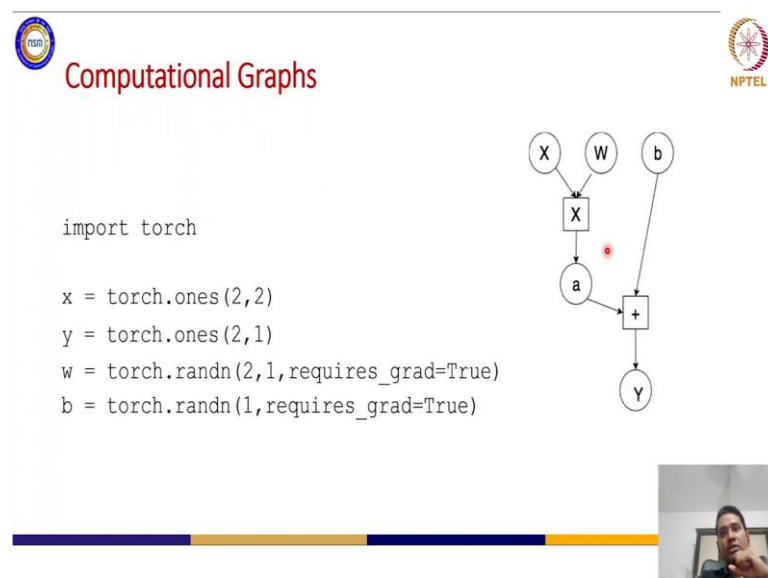
So, how to define one network class. So, here first we need to input the packages. So, basically nn dot nn package we have imported as nn and then we need to define the class for our network module.

Now, class net here is the nn module that we want to define. You can rename it any module let us say whatever modelling you are targeting, you can rename it from net to anything. So, nn dot module, these are the basic PyTorch structure for inheriting one class. So, nn dot module is the super class of this net class that we are defining here and then we want to initialize or we want to initialize the class itself. And for that you need to first initialize or first initialize the super class and then in the initializer of this class we will create the layers for our network module.

So, what are the layers we need for defining our module that will be initialized or that will be defined is this initializer of this class. And as you can see these are just the regular expressions and regular definition for initializer of a class in PyTorch. We will not go into details of description of these statements and then another function you need to define the feed forward function, which is the forward pass that we are talking about.

So, how the input will flow from input to output. Remember here, one thing we talked about the computational graph. So, computational graph we have the input tensors as let us say we have seen x and output let us say as y and you have your weight parameters as w and bias as b and intermediate tensor outputs ok which were by a.

(Refer Slide Time: 04:13)



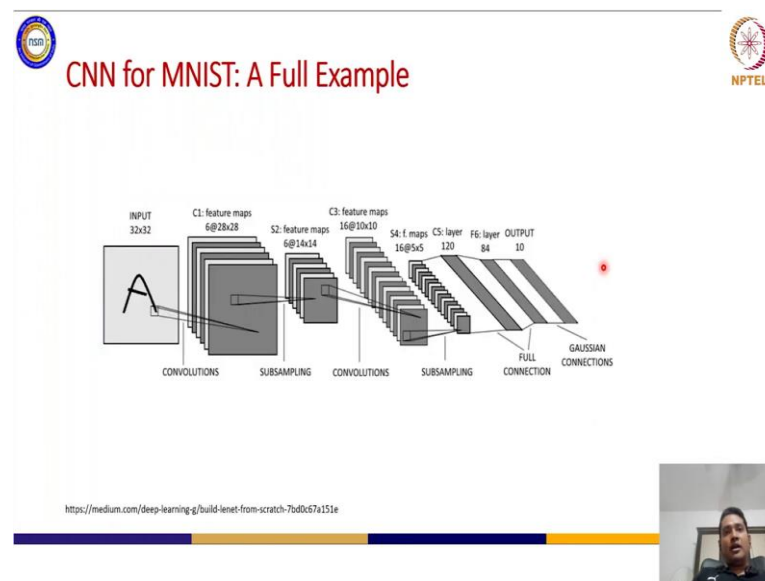
So, you just see the computational graph here. So, x, w, b, a, y. So, all these are essentially values that you need to store inside your GPU. So, if you are working with GPU, so you need to be aware of the sizes of these tensors that you are working, as well

as when you do the back propagation you will compute the gradients of these parameters and these gradients also you need to store inside the GPU.

So, when you have very large computational graph, then you can imagine how much data that you need to store inside your GPU and if you are out of the storage that is available for your GPU. Then you will get one error and you need to change certain parameters to be compatible for the training of this level.

So, that we will discuss our structure, but let us get on with defining this forward pass that we are talking about. And you do not need to define one backward function important because as I was mentioning that, it is just one layer of code which will define the backward function.

(Refer Slide Time: 05:26)

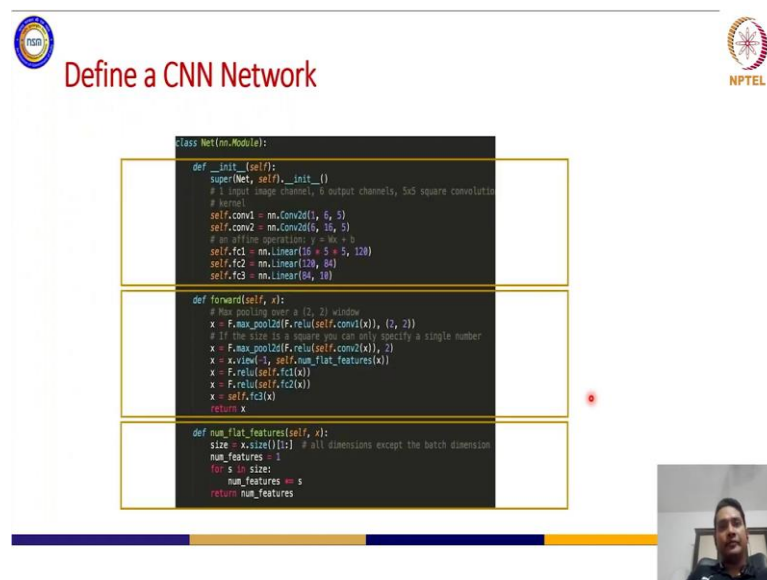


So, as use case example we will define this network model which is a LeNet architecture, which takes one input and it goes through several layers. So, all these layers as you can see convolutions layers first, then sub sampling layer which is essentially max pooling layer. Then we have convolutions again sub sampling again and two fully connected layer and in the end we have one fully connected, final layer with this Gaussian connections.

So, this is the entire network architecture or network model that we want to define as the forward pass. So, basically input will go from this left side to your right side at the output and that is one forward pass, that we want to define as the feed forward neural.

Now, while defining these networks, we need to be aware of the dimensions of each layer as you can see here right. So, these dimensions are essentially very very important to be able to define the tensor sizes particularly. And if you are not compatible with the sequence of dimensions that you want to define, then you will get one error.

(Refer Slide Time: 06:49)




Define a CNN Network

```
class Net(nn.Module):  
    def __init__(self):  
        super(Net, self).__init__() # init_  
        # 1 input image channel, 6 output channels, 5x5 square convolution  
        # kernel  
        self.conv1 = nn.Conv2d(1, 6, 5)  
        self.conv2 = nn.Conv2d(6, 16, 5)  
        # max pooling over a (2,2) window  
        self.fc1 = nn.Linear(16 * 5 * 5, 120)  
        self.fc2 = nn.Linear(120, 84)  
        self.fc3 = nn.Linear(84, 10)  
  
    def forward(self, x):  
        # Max pooling over a (2,2) window  
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))  
        # If the size is a square you can only specify a single number  
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)  
        x = x.view(-1, self.num_flat_features(x))  
        x = F.relu(self.fc1(x))  
        x = F.relu(self.fc2(x))  
        x = self.fc3(x)  
        return x  
  
    def num_flat_features(self, x):  
        size = x.size()[1:] # all dimensions except the batch dimension  
        num_features = 1  
        for s in size:  
            num_features *= s  
        return num_features
```


So, we will talk about this; let us get on with the first definition. So, as you can see here first we will define the initializer with all the layers definition inside this initializer then in the function of feed forward function which will be defined. So, all the things we are defining in this class net, which is inherited from nn dot module as you can see.

So, feed forward function for what this definition of this function will give you the feed forward function that we will discuss and you can have as many number of network functions inside your network as you can. Here we have one network function as num_flat_features, which used to flatten the input tensor to your 1-dimensional vector. So, that we will; so these are the essential functions that you need to develop. So, different what you all are know.


(Refer Slide Time: 07:47)



Define a CNN Network



```
def __init__(self):  
    super(Net, self).__init__()   
    # 1 input image channel, 6 output channels,   
    5x5 square convolution kernel  
  
    self.conv1 = nn.Conv2d(1, 6, 5)  
    self.conv2 = nn.Conv2d(6, 16, 5)  
  
    # an affine operation: y = Wx + b  
    self.fc1 = nn.Linear(16 * 5 * 5, 120)  
    self.fc2 = nn.Linear(120, 84)  
    self.fc3 = nn.Linear(84, 10)
```



So, the first function that we want to define one is the initializer. So, in the initializer we have, first we have defined the convolutional layers. So, you can remember from the architecture that one that we saw that we have two convolutional layers.

So, first we are defining the first convl. So, here we are not actually passing the input from from input to output. So, we are just defining the layers and once we connect all these layers together and then only we can actually do the feed forward pass.

So, while defining convolutional layer. So, let say I am defining convolution 1, so nn dot Conv2d. So, this function is already there here we are just using the function to overwrite these values with. So, the dimension here first is the batch size. So, first is the depth of the filter ok, that we want to convolve with and then the number of filters.

So, basically, we have only one depth. So, if you see this structure. So, this is one input that we are we want to onboard with some filters here. So, how many filters we are trying to convolve with? 6 filters in this convolution layer. So, that is why the second dimension was 6, which is the number of output channels or maybe output feature size that we want to work with.

So, total 6 filters we are using here and since we are using grayscale image. So, you can represent your data in any format and here we are talking about image format as the

input data, where we are using grayscale image as the input. So, it has only one channel so; that means, depth is 1.

So, you remember from your definition of convolutional layers or convolutional neural networks, there are several layers as you can see convolution layer fully connected layer. So, convolution layer is the transformation from volume to volume, where you have you have depth you have the number of filters and the spatial features.

So, here I am defining this convolution 2D function, we are defining that this convolution layer will have one depth, because that is equal to the number of channels that is available inside your input image, which is grayscale image 1, number of filters and number of feature kernel size.

So, here we are using 5 x 5 square convolution kernel. So, that is why 5 you can if you have, if you do not have square kernels or square convolution filters, you can define with comma and parenthesis with the dimension. So, if you are using 5 x 4 convolutional kernel then you can use 5 comma 4 in a parenthesis to define your irregular convolutional kernel right, which we do not use often.

Now, second convolution layer has 6 as the input depth and number of output depth will be 16. So, 16 filters we are using here and again the spatial features is 5 x 5. So, as you can see here, this convolution layer output depth and this convolution layer input depth is same. So, if you do not have this matching parameters, then you will actually cannot define these layers or cannot define this feed forward neural network as one class.

Then we will define the linear layers. So, just remember the figure here so two convolutional layers, that we have defined. Next, we will define the fully connected layers. So, fully connected layers is flattened. So, as you can see here we have flattened the entire tensor into 1-dimensional tensor ok.

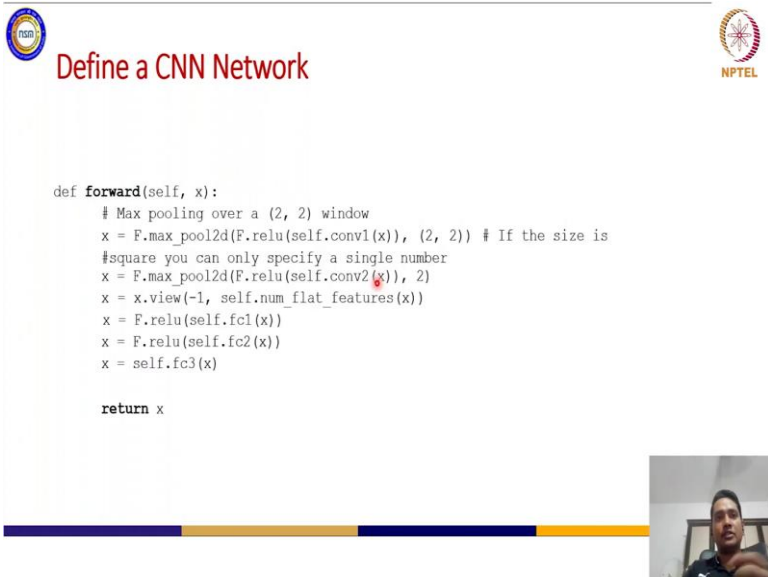
So, that is where we are actually using the this linear function. So, linear function in this nn package will actually be used to define your flat fully connected layer of linear layer ok.

So, linear layer we have seen before also. So, $w * x + b$ which is the linear definition and again another feed forward neural network which has 120 x 84 and another feed forward neural network sorry, fully connected layer which is 84, 10.

So, $16 * 5 * 5$, how we are getting this dimension? This dimension is essentially after flattening these layers output we are getting this dimension. So, if you see the structure. So, $16 * 5 * 5$ which is the total number of parameters that we are having in this layer and then we are flattening that into one dimension so that is why we have this.

So, this dimension as input. So, all these connections you need to be aware of and defining these layers.

(Refer Slide Time: 13:52)



Define a CNN Network

```
def forward(self, x):
    # Max pooling over a (2, 2) window
    x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2)) # If the size is
    #square you can only specify a single number
    x = F.max_pool2d(F.relu(self.conv2(x)), 2)
    x = x.view(-1, self.num_flat_features(x))
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)

    return x
```

So, once we have defined the layers. So, this layers definition is done, then we can actually defined the feed forward pass to using these functions ok.

So, first we will define the first layer. So, first layer is essentially, pass the convolution layer. So, feed forward is taking the input tensor as x , which is essentially the batch ok; batch of images, batch of text, batch of audio signal whatever you are actually transferring as the data. So, mostly we do batch processing as I was mentioning several times before. So, x is batch input and let say you have 16 samples inside your batch, here 16 images. So, let say you have 16 into 1 into 32 by 32.

So, 16 is the number of samples you have, one is the number of channels inside the image which is one for grayscale image and if you are using color image it will be 3 and let say size of these images are 32 by 32. So, x has x input tensor has this dimension of 16 into 1 into 3 into 3 sorry 32 into 32. So, that is the input tensor size and that we are passing to convolution 1.

So, convolution 1 that definition we have done in the previous slide ok, convolution 1. So, this function we are trying to pass it through this convolution function, then we are applying one relu function which is defined in this function. So, this F is nothing but the functional api of PyTorch.

So, while doing the demonstration we will see the usage of this, but this is how you will define it. So, F dot relu is giving you the access to this non-linear function that will be applied on this output tensor and then we are applying max pool2d which is the sub sampling with 2 by 2 parameter ok.

So, since all these are essentially non parametric layers. So, relu max pool2d, we are not defining explicitly inside the initializer. So, if you want to define it, you can also define it there and use it here.

So, we are directly using those layers from the nn from the functional api and so x is the output of the first layer. Now, next we want to make available this x into the next layers input. So, next layers convolution 2 is taking this x as input and so see here, to practice is use the same variable name all over, because this is always a sequential definition.

So, if you are trying to use different names for different inputs and outputs, you will for sure make a mess of the entire network while you are defining a large network. So, x input output is x, again x input and output is x that is why you can define it very efficient.

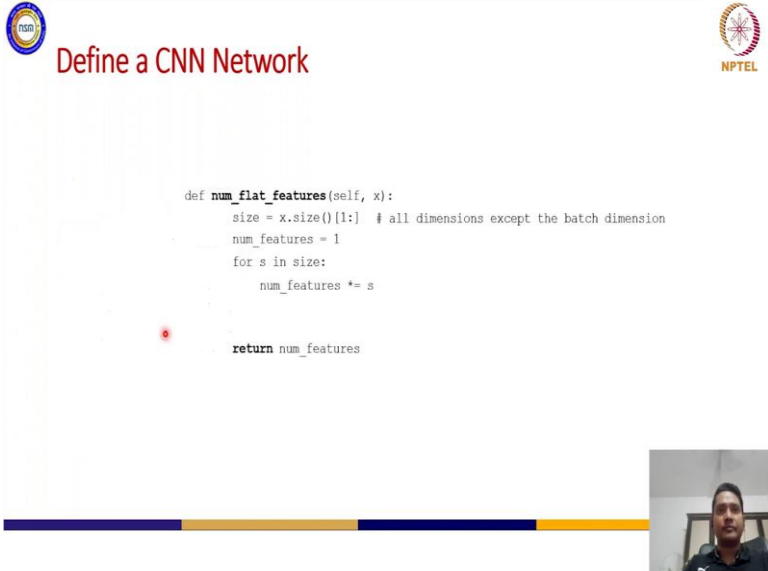
So, we are defining this convolution two and then applying the relu function on it and then max pool2d, another sub sampling and x is the output. And again we are flattening this x using this view function. So, view function you remember that is used to resize this.

Now, we do not know what is the max size that we are applying to this tensor input tensor. So, depending on the number of max sizes you want to flatten right; so for 16 what is the total number of? So, for 16 images what is the total number of flatten features?

So, convolution is transformation from volume to volume, fully connected layer is feature to feature. So, here we need to flatten the features that are output from your convolution layers. So, that is why we have defined this num flat features as one helper function, which will actually return the total size of this input tensor x, which is the output of the previous convolution layer.

And we will resize it, then we will give this x to your next fully connected layer. Then again relu function, then again fully connected layer and relu function, again fully connected layer which is the last and then we will return to this x. So, this x I want to compute in the forward pass. So, from the input tensor you are getting this output tensor or output of this feed forward neural value and that is how you will define this entire set of layers, you can see this is very modular and easy to define any part.

(Refer Slide Time: 18:44)



The slide is titled "Define a CNN Network" in red text. It features a Python code block defining a function `num_flat_features`. The code is as follows:

```
def num_flat_features(self, x):
    size = x.size()[1:] # all dimensions except the batch dimension
    num_features = 1
    for s in size:
        num_features *= s
    return num_features
```

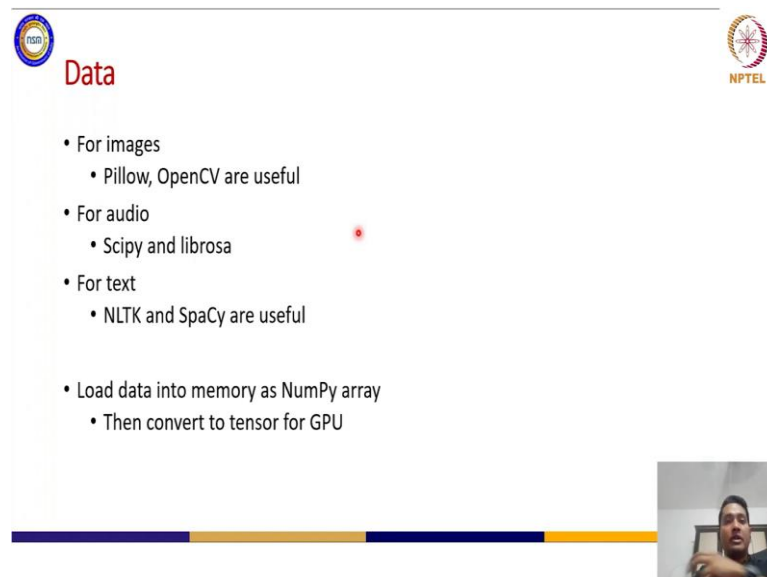
The slide also includes a small video inset in the bottom right corner showing a person speaking. The slide is decorated with a blue and yellow bar at the bottom and logos for "TUM" and "NPTEL" in the top corners.

This is the help of function that I was talking about, which is the num flat features, it is taking the input self comma x which is let say 16 into 5 into 5 or 16 let say max x right. So, here we are applying size in equation. So, x dot size will give you the size entire size, entire list of sizes and we are ignoring the first dimension which is the batch size. And

taking all the other dimension as the list in the size and iterating through the size and multiplying and so basically if you have 16 into 16; 16 4 batches, then 16 to 5 into 5.

So, ignoring 16 you are taking 16 5, 5 into size and then iterating to 16 into 5 into 5 and returning this number space. So, this is use of slices which is very handy in defining or flattening in the features. We can use dot flatten function instead of view that I have used in this to flatten the input tensor, into 1-dimensional tensor, but since you can see that based on the batch sizes this view is very very flexible more than the active function. Once you have defined the neural network then what is the step? Iterate over the dataset of inputs.

(Refer Slide Time: 20:14)



The slide is titled "Data" in red text. It features a bulleted list of data processing steps. The first three items are "For images", "For audio", and "For text", each with sub-bullets listing useful libraries. The fourth item is "Load data into memory as NumPy array", with a sub-bullet "Then convert to tensor for GPU". A small video inset in the bottom right corner shows a person speaking. The slide is framed by a blue and yellow border.

- For images
 - Pillow, OpenCV are useful
- For audio
 - Scipy and librosa
- For text
 - NLTK and SpaCy are useful
- Load data into memory as NumPy array
 - Then convert to tensor for GPU

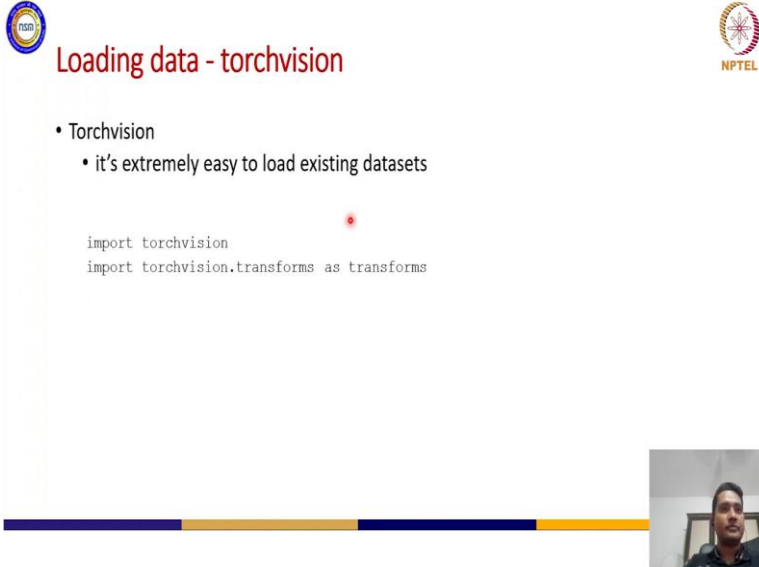
Now, platform you will get error. If you are working with images Pillow, OpenCV are very very useful to give you some operations which will be very very handy to of transformations which will be very very antique to apply on images

If you are working with audio you can pre-process or gate some audio tools SciPy and librosa, if you are working with text then you can take help of NLTK and SpaCy these two libraries very very useful or text processing.

So, first is to load in data into your memory, which memory if you are working with CPU then CPU memory and if you are working with GPU, GPU memory. And we need to take convert the number is which will be actually play as the data. So, basically if you

have data already inside your disk and you are reading it, you are reading it as an array and you need to convert it into a tensor and then you can transform it to GPU to be able to access into your module, which will be done on inside your GPU.

(Refer Slide Time: 21:19)




The slide is titled "Loading data - torchvision" in red text. It features a bullet point stating "Torchvision" and "it's extremely easy to load existing datasets". Below this, there is a code block with the following Python code:

```
import torchvision
import torchvision.transforms as transforms
```


The slide also includes logos for IIT Bombay and NPTEL in the top corners. A small video feed of the presenter is visible in the bottom right corner.

So, in this lecture we will use throughout mostly we will work with the images. So, torchvision will be very very handy if you are working with images to download already existing data set and you can work with them. So, work with torchvision, torchvision importing and then importing transforms as transforms will be very very important. Because transforms from this package will give you very much flexibility to apply many many transformations onto images which should be very useful into your neural networks or CNN modules that will define.

(Refer Slide Time: 22:01)




Loading data - torchvision



```
import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])])

trainset = torchvision.datasets.CIFAR10(root='./data',
                                         train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset,
                                           batch_size=4, shuffle=True, num_workers=2)
```



So, first how we want to load data? So, first we will load data with defining the transformations that we want to apply on the images. So, believe me if you are working with data models and if you are working with any kind of data, be with images be text data, audio data whatever kind of data. It is except of tuning and tuning the parameters and getting your desired accuracy and performance you will be applying most of the time or you can say 90 percentage of time transforming or pre-processing or preparing your data for your neural networks or your data models.

So, transforms will give you some pre-defined transforms that you can directly apply on the input tensors as the images. So, basically transforms dot ToTensor we are actually transforming these images into tensors and applying some normalization. So, these are the different set of transformations and we are defining with this compose function.

So, when we will apply this transform, all these transformations will be applied one by one. You can apply any number of transformations and define any number of transformations that you can have. And normalize you will take mean and standard deviation. So, first argument is mean, the set of means are essentially for the channels input channels. So, if we are working with. So, here we are working with CIFAR10 data set which is essentially color images.

So, three channels it has and to normalize each channel for your input data we are defining these three sets of mean and three sets of standard deviation. Then we are

defining the training set. So, essentially when you are doing the forward pass, you are essentially feeding the data from your training data set inside the forward pass. So, training data set to define in PyTorch, torchvision dot datasets dot CIFAR10 which we are using as the predefined data set root is in the path.

So, here is essentially the path where you want to store the data set and training is equal to true so; that means, this is a training data set downloading is equal to true, if that data is not available in this path then you can download automatically and apply this transform. So, which is the transform, the transform that we have defined. And then we need to define the trainloader and this is the data loader definition which is very very important inside to define in PyTorch to be able to effectively use the data inside your train.

Because mostly if you are using GPU, you are actually running your feed forward pass and backward pass inside your GPU. So; that means, you want to load first data with CPU maybe and your GPUs will run very very fast and to be able to feed the data with the synchronous fashion, because if you are keeping the GPUs waiting for data then nothing is of use.

So, basically you want to define your dataloader efficiently so that you can supply your data to your GPUs very effectively. So, trainloader torch dot utils dot data dot DataLoader we are defining this data loader here, training set is the data that we are actually keeping or taking the data from batch underscore size is defining the batch size.

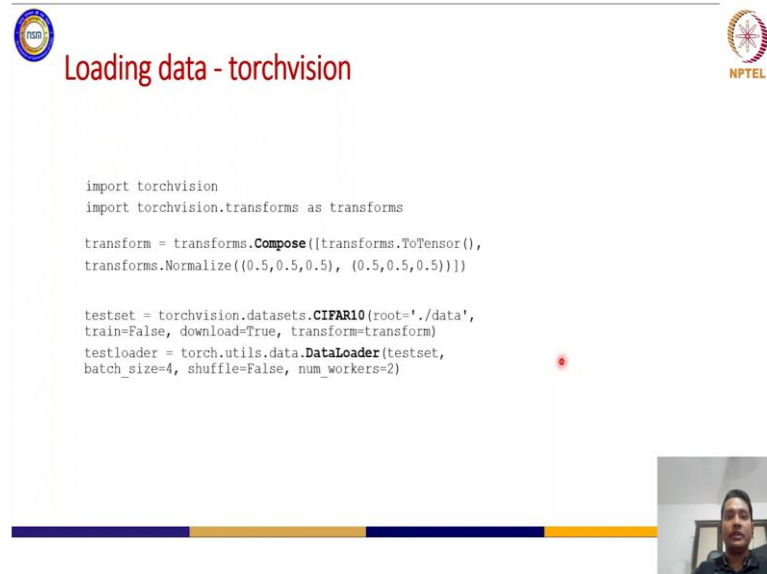
Now, here if you want to increase the batch size you can increase, but again to give how much your GPU is taking and how much you are supplying to the GPUs that is very important. So, this parameter you want to tune with different numbers for your train.

Shuffle equal to true for each iteration of training you want to you want to feed several sets of these batches, because let say 4 batches, 4 images you want to capture or take data from the train set and you want to give it to the training model. And if you want to if you do not shuffle; that means, every iteration will you are giving same set of image data or whatever data you are giving.

So, shuffle equal to true for defining training loader is very very important, number of workers. Now, how many number of cores, you are having that many number of workers

definition is the thumb rule, but you can define as many number of workers as you can to effectively process this much of a batch size to give the data to your view.

(Refer Slide Time: 27:03)



The slide is titled "Loading data - torchvision" in red text. It features a code editor with the following Python code:

```
import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])])

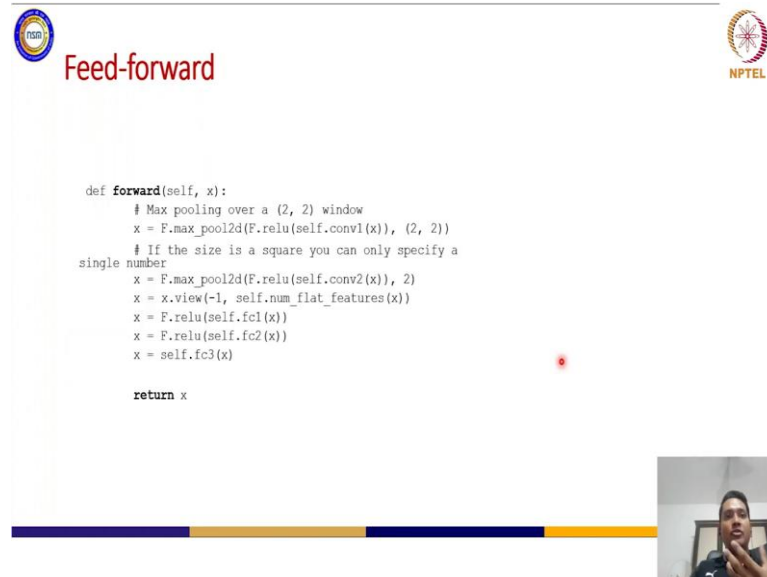
testset = torchvision.datasets.CIFAR10(root='./data',
                                       train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset,
                                          batch_size=4, shuffle=False, num_workers=2)
```

The slide also includes the NPTEL logo in the top right corner and a small video feed of a presenter in the bottom right corner.

Then loading the data because once we have the data set ready then you can actually get. So, this was for training the data or if you want to take some data so you want to load the test data as well. So, same again the transformation and let say test set we are defining this test set and this data set is having download = true, but training go on to false this is not for training and transformation is same transformations we and test loader is the data loader for test.

So, basically, we are using the data which the data set which we have to find with the specific transformation. And then the batch size shuffle equal to false because for the test you do not need to shuffle your images. If you want to you can and also you can define the number of workers depending on the configuration you have for your system and then you have to process the input to the network.

(Refer Slide Time: 28:10)



Feed-forward

```
def forward(self, x):
    # Max pooling over a (2, 2) window
    x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
    # If the size is a square you can only specify a
    single number
    x = F.max_pool2d(F.relu(self.conv2(x)), 2)
    x = x.view(-1, self.num_flat_features(x))
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)

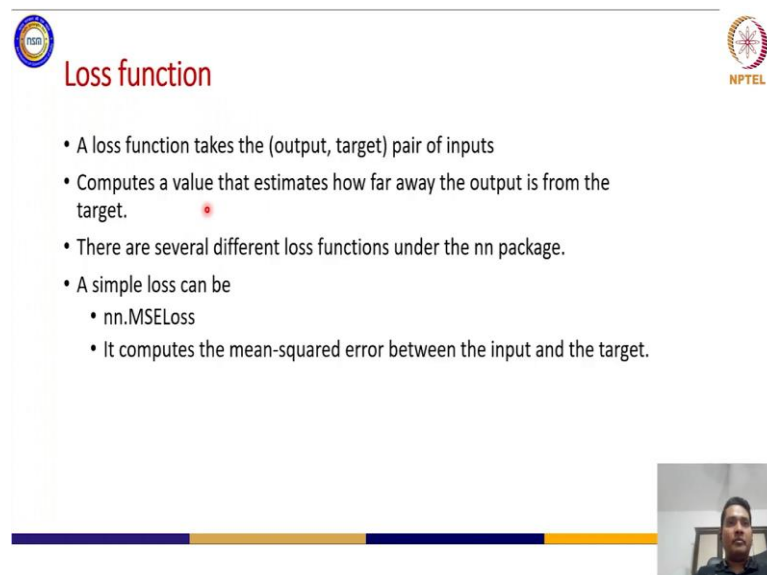
    return x
```

And how we want to process the network? Through this feed forward function.

This feed forward function we have defined so x equal to first we are applying this convolution layer relu and this max pooling, again convolution relu max pooling then to fully connected layer. And finally, fully connected layer to keep output. So, return x is essentially, feed forward output which we want to get from.

Again remember, so x is essentially the batch data we are giving in; that means, we want to compute the loss for the batches that we are giving ok.

(Refer Slide Time: 28:48)

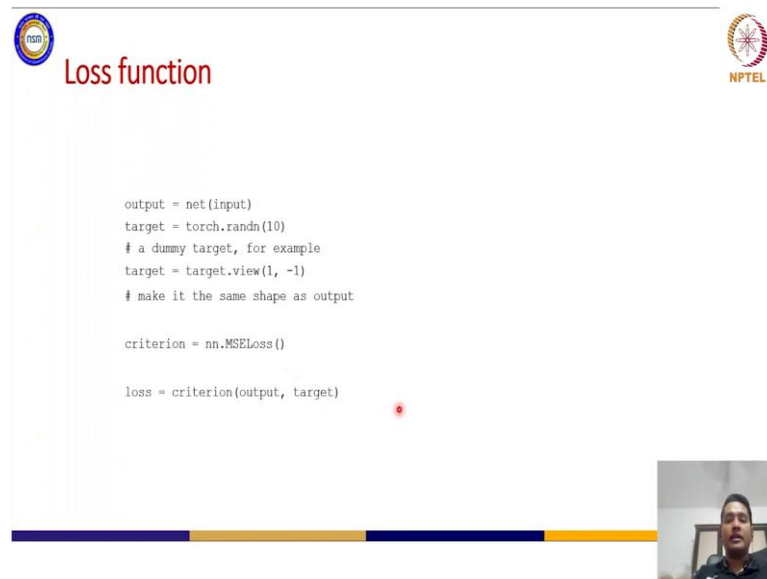


Loss function

- A loss function takes the (output, target) pair of inputs
- Computes a value that estimates how far away the output is from the target.
- There are several different loss functions under the nn package.
- A simple loss can be
 - nn.MSELoss
 - It computes the mean-squared error between the input and the target.

So, computing the loss function is the next improving step.

(Refer Slide Time: 28:52)



Loss function

```
output = net(input)
target = torch.randn(10)
# a dummy target, for example
target = target.view(1, -1)
# make it the same shape as output

criterion = nn.MSELoss()

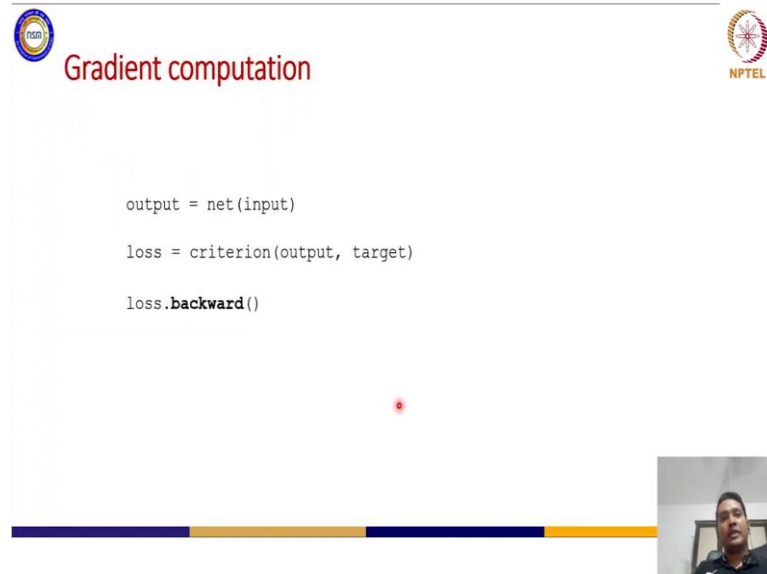
loss = criterion(output, target)
```

And to compute the loss function as we have seen that you need to compute the loss depending on a set of targets that are available. And here as you can see we have let say 10 classes and let say we are initializing one random 10 string with 10 values and so this is the target or ground truth.

And once you have the output from your network, which is the output from the network. So, this output and the target which is the down put that these two parameters will be used for computing your loss. And as you can see nn dot lot of loss function you can see MSELoss cross entropy loss. So, this is mean square error loss.

So, many loss functions you can use directly with this nn package and you can define your loss as criterions. And then you can define the loss with this output and target and then you can actually go to what is the next section which is to propagate the gradients back to your networks parameter, which is just the backward function.

(Refer Slide Time: 30:00)



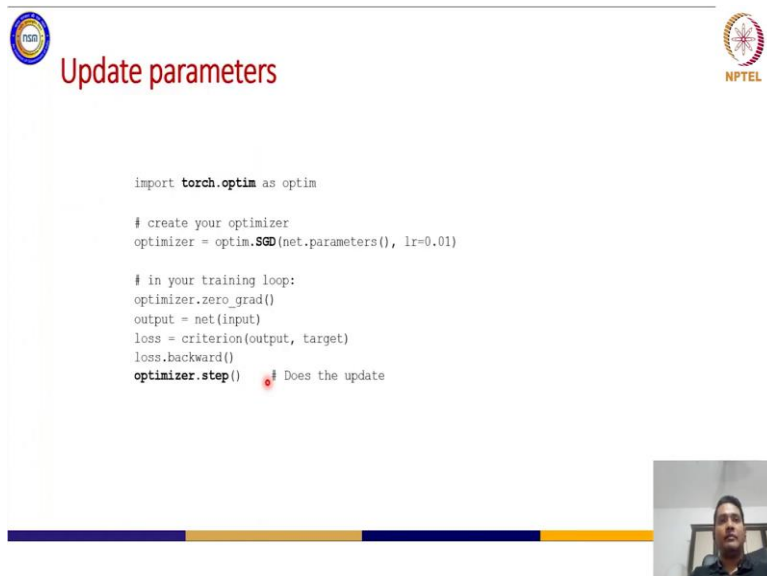
The slide is titled "Gradient computation" in red text. It features a code editor with the following Python code:

```
output = net(input)
loss = criterion(output, target)
loss.backward()
```

The slide includes logos for IIT Bombay and NPTEL in the top corners. A small video feed of a presenter is visible in the bottom right corner.

So, once you have the loss calculated, then you can use this backward function which will take care of computing the gradient and then, the next or last stage of the inter training pipeline is the object of weights, which is again very simple.

(Refer Slide Time: 30:18)



The slide is titled "Update parameters" in red text. It features a code editor with the following Python code:

```
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step() # Does the update
```



The slide includes logos for IIT Bombay and NPTEL in the top corners. A small video feed of a presenter is visible in the bottom right corner.

So, to update the weights what kind of optimizer we are using optimizer we can use, this optim library and SGD is the stochastic gradient descent we are using you can use a term (Refer Time: 30:35) grad. So, there are many optimizers you can use with different parameters and then you can actually define with zero grad.



Let say to initialize all the gradients to 0, you can actually initialize with dot zero grad function and then you can compute the output, then loss function and then backward function which will compute the gradient and then the output. So, then the update which is the last step.

So, update is again just one line function, that you can see dot step functionally do the update for all the parameters that we have defined as the parameter.

(Refer Slide Time: 31:18)



- Total training samples = 80000 Batch-size = 50
- Each iteration takes 30 seconds
- How many hours for 3 epochs of training?




So, one question here remains. So, basically one epoch is essentially to scroll through all the train set for all the batches.


So, let say you have 8000 samples and batch size is 50. So, basically how many total number of iterations you have to do to end the entire set of batch or entire set of samples that is one epoch. And let us imagine that to have this one iteration takes 30 seconds and how many iterations that you can have to compute 1 input.

And then you can compute let say I want to done 3 epochs of training. So, how many number of how many number of hours or how many number of seconds you can spend to train this network? So, you can do this as one exercise.

(Refer Slide Time: 32:15)



Full training




```
net = Net()

trainloader = torch.utils.data.DataLoader
                (trainset, batch_size=4,
                 shuffle=True, num_workers=2)

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(net.parameters(),
                       lr=0.001, momentum=0.9)
```



And we will discuss this full training as want to know in our coming class.

Thank you.