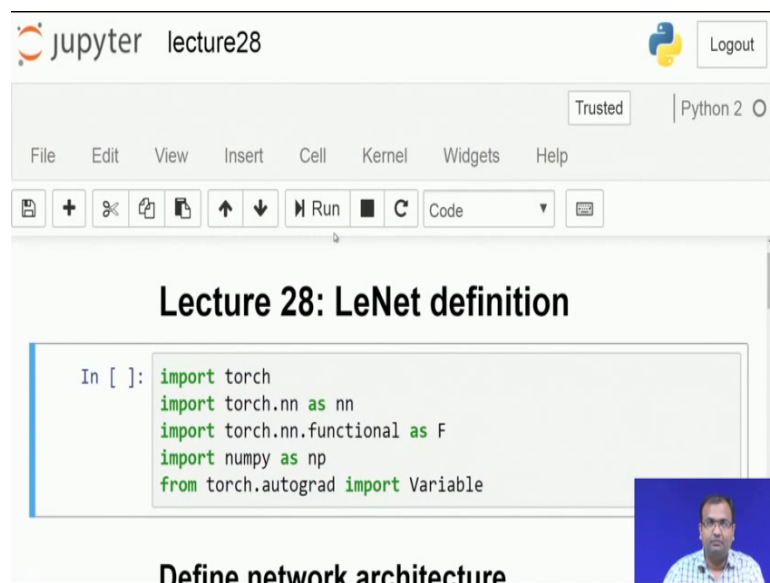


Deep Learning for Visual Computing.
Prof. Debdoot Sheet
Department of Electrical Engineering
Indian Institute of Technology, Kharagpur

Lecture – 28
LeNet Definition

Welcome, today we are going to do about defining a LeNet and as in the earlier lecture, you have already seen getting introduced to the functional blocks of how to get started with a convolutional neural network and subsequent to that I have introduced you to some of the very basic type of architectures and one of those predominant basic type of architecture was the LeNet which we are doing. Derive from the name of yann lecun who started with this one.

(Refer Slide Time: 00:42)



```
In [ ]: import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from torch.autograd import Variable
```

Define network architecture

What we are going to do today is essentially actually go through how the network definition happens. You might believe that it is quite easier to get it done because you have already learned a lot about defining networks which are fully connected in nature intent of. You have fully connected neural networks which we done based on your auto encoders and then autoencoders used for initializing down the rest of the networks.

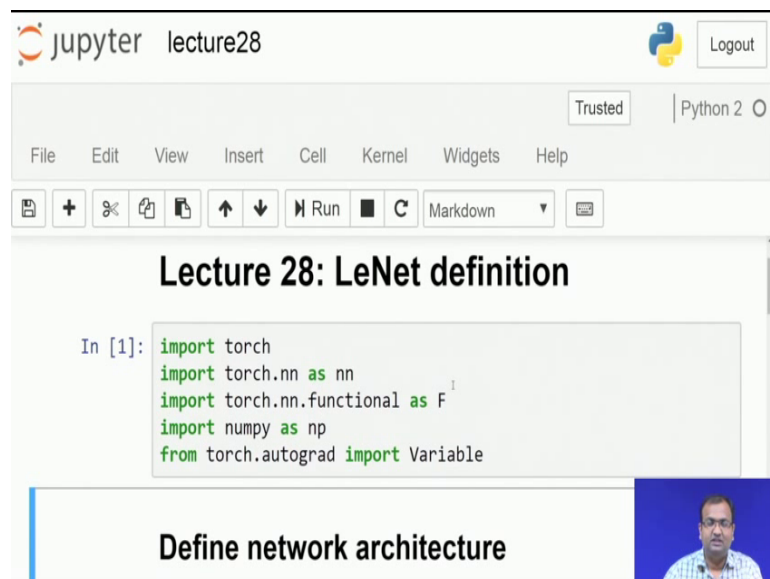
However, keep one thing in mind that when we are doing convolutions you do did get introduced on to this new kind of an operator which was the convolutional operator and as such the weights which are connected down and no more weights which connect

down each pixel to every other neuron over there, but these now operate more of in a convolutional sense.

There is no unique connection between a group of neurons, but they get connected by other convolutional operator over there and that leads to even different kind of data exchange. Because, while in a fully connected network your total number of input pixels was equal to the total number of neurons into the network and then at any intermediate layer. Whatever is the output is the size as the number of neurons whereas, over here what happens is that there is a change over there and in no way your number of neurons in the input layer is this somewhat which is dependent on the total number of pixels in the image.

Going by that let us get started with just trying to define the whole net purpose. The first part over here is that I would try to write down my header part.

(Refer Slide Time: 02:06)



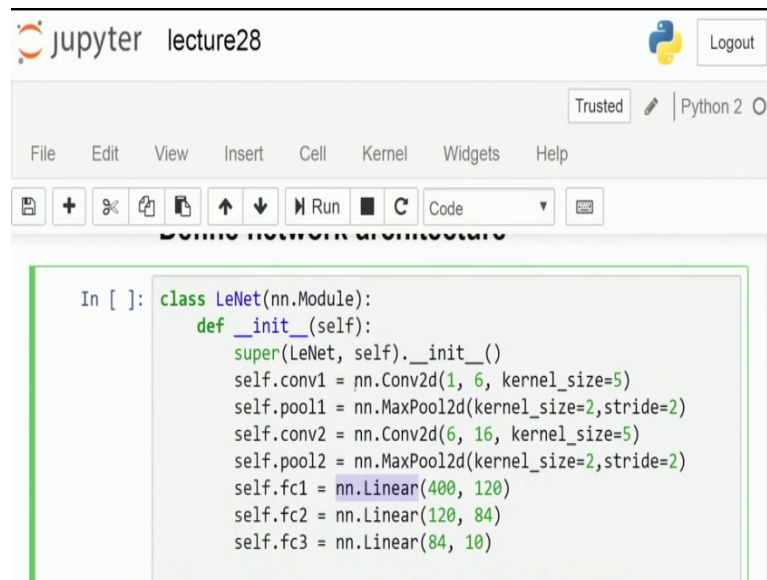
The screenshot shows a Jupyter Notebook window titled "lecture28". The interface includes a top bar with the Jupyter logo, the title "lecture28", a Python logo, and a "Logout" button. Below this is a "Trusted" status indicator and "Python 2" kernel information. A menu bar contains "File", "Edit", "View", "Insert", "Cell", "Kernel", "Widgets", and "Help". A toolbar below the menu bar contains icons for file operations, a plus sign, a refresh icon, a copy icon, a paste icon, up and down arrows, a "Run" button, a stop button, a refresh button, and a "Markdown" dropdown menu. The main content area displays the heading "Lecture 28: LeNet definition" in bold. Below the heading is a code cell with the following Python code:

```
In [1]: import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from torch.autograd import Variable
```

At the bottom of the notebook, there is a section titled "Define network architecture" and a small video feed of a person speaking.

Let us just import and this part of my headers and all the libraries which are to be called down are you know very different. Because, that is quite similar to what we have done in our fully connected architectures and that does not change anything that does not introduce anything new as such doing them.

(Refer Slide Time: 02:22)



The screenshot shows a Jupyter Notebook window titled "lecture28". The interface includes a menu bar with "File", "Edit", "View", "Insert", "Cell", "Kernel", "Widgets", and "Help". Below the menu bar is a toolbar with icons for file operations and execution. The main content area displays a Python code cell with the following code:

```
In [ ]: class LeNet(nn.Module):
        def __init__(self):
            super(LeNet, self).__init__()
            self.conv1 = nn.Conv2d(1, 6, kernel_size=5)
            self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
            self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
            self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
            self.fc1 = nn.Linear(400, 120)
            self.fc2 = nn.Linear(120, 84)
            self.fc3 = nn.Linear(84, 10)
```

Now, comes the next point which is trying to define the network architecture. Now, if you go by the LeNet definition and then clearly. Recall from your slides and you would see that what comes down over there is on the input side you have grayscale images which are handwritten digits in the range of 0 to 9 and these are encoded in 28 small patches of size 28 cross 28 now these patches of precise 20 across 28 are basically what correspond to each of these ones?

Now, in my LeNet what I was doing is technically that I take down a patch of this size 28 cross 28 and then I have a set of 6 convolutional operators each convolutional operator has a size of 5 cross 5. That is what I start by defining over here. You see 1 over here which technically corresponds to the number of input channels or how many number of gray channels are there on my input side over there now my input was just a 28 cross 28 patch and it wasn't a color image. 0 just one single channel on my input what I have. That corresponds to this one which comes down over here.

Now, the number of unique kernels which I am going to define and learn down in this linnet in the first layer is what is equal to 6 and that corresponds to here which can also be called as a number of channels in your output or the number of unique convolution colors which you are going to learn and then comes down my con convolution kernel size. My kernels over here are taken down as 5 cross 5. Defining isotropic sizes over

here now you can also choose by defining non-isotropic sizes and that is that is pretty doable which will be doing looking into in later on lectures as well.

Now, from this what I get down is that the connections are no more n, n dot linear. You had something like this in the earlier case which was a `nn.linear` which is to connect down all neurons in one layer to all the neurons in the subsequent layer. Here, it is not a one to one all the connections which go down, but 0 more of convolutional in nature that they are connected down and for that reason. This connection is also defined by a different kind of a function which is called as `nn.conv 2d`. This is a convolution which will happen in the 2-dimensional space.

Now, when you are doing that you also have an option of doing strided convolutions as well as you can do padding in terms over there. Over here in in standard LeNet since we do not have the concept of padding or stride coming down in any way. We are not going to define that. This is just the native straightforward definition of a LeNet as we had the next layer over there was basically a max pooling layer which you had and that was a max pooling of 2 cross 2 factor over there.

That essentially meant that you have a 2 cross 2 kernel which was max pooling it out and you were striding it by a factor of 2. That mean meant that every 2 cross 2 2 2 comma 2 or 4 for group of 4 pixels over there is what was represented in terms of it is maximum value and then you shift over to the next group of 4 pixels over there. That is what we use for defining over here as kernel size of 2 and stride of 2.

Now, once that is done the next part is to go down with the next amount of convolution and this convolution is again a 2-d convolution in which we are going to connect down 6 channels of output which comes over here now you can recall down from the last lecture that max pooling does not in any way do a pooling across the number of channels. The number of channels does not change in any way 0 it is it still remains the same.

The number of output channels over here is still going to remain 6 after this max pooling operation in the 2d. Now, once you have max pulled out then you have your second convolution layer or second convolutional layer definition which comes down. The number of input channels over there is still 6. Because, I do not change now that connects down to 6teen output layers or the total number of unique kernels which I defined over here. There are 16 such unique convolution converse which get defined

over here and each of the kernel has a spatial span of 5 cross 5 and that pretty much defines it.

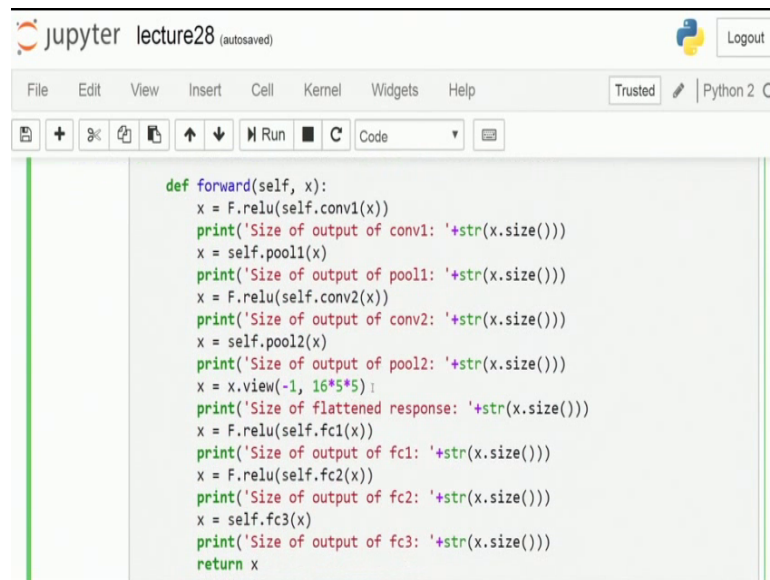
Following that you again have a max pooling. My max pooling over here is again 2 cross 2 max pulling with the stride of 2. That 4 pixels in a group of 2 cross 2 is what get represented in terms of the maximum value present over there. Till here what we have is from here till here is what is just a convolution part of the definition or within a network this these are just what operate down as a convolution operation they do not have any fully connected or they would not be doing all to all pick neurons connected to each other. The next part which comes down is my first fully connected layer and this is where it connects down 400 neurons in linearly 220 neurons.

Now, the fact is how did we get this 400 neurons. From the last class you do remember that we were doing a calculation of finding out exactly like if I have a 28 cross 28 sized image. Which is given on an input over here and then it does a full convolution then you get down what do you say max pooling on the 2d space and then you have another convolution and then you have another max pooling.

At the end of the day what comes down over here is basically of the size of 16 cross 5 cross 5. Your 16 channels which comes down over here and the spatial span of the resultant output over there is 5 into 5. That makes it 16 into 25 which is equal to 400. There are basically 400 such neurons present about that, but then this is a whole volume stack. You have the z axis where you have the different channel outputs over there and then you have a spatial span of 5 comma 5.

That all the neurons which is present in this whole compact volume of 400 neurons. Over there is what is connected down 100 and 20 neurons and this is a fully connected thing. All the 400 neurons go down 220 neurons subsequent to that is a very simple connection which is 120 such neurons which are linearly connected and then again subsequently linearly mapped on to 84 neurons and from 84 neurons. We go down to these 10 neurons and these 10 neurons are basically.

(Refer Slide Time: 08:11)



```
def forward(self, x):
    x = F.relu(self.conv1(x))
    print('Size of output of conv1: '+str(x.size()))
    x = self.pool1(x)
    print('Size of output of pool1: '+str(x.size()))
    x = F.relu(self.conv2(x))
    print('Size of output of conv2: '+str(x.size()))
    x = self.pool2(x)
    print('Size of output of pool2: '+str(x.size()))
    x = x.view(-1, 16*5*5)
    print('Size of flattened response: '+str(x.size()))
    x = F.relu(self.fc1(x))
    print('Size of output of fc1: '+str(x.size()))
    x = F.relu(self.fc2(x))
    print('Size of output of fc2: '+str(x.size()))
    x = self.fc3(x)
    print('Size of output of fc3: '+str(x.size()))
    return x
```

The final neurons in the classification which are present over there.

Now, following that the next part which we were defining is my forward pass of it now you can see that we have written down some extra lines over here as a print statement and the whole reason for writing this print statement was actually to pool down and probe out what is the size of the output coming down from each of them. You have your first layer of convolution. If I give down a 28 cross 28 then what comes down after my first layer then what goes into my second layer. After the max pooling and everything is not.

This was just to write it down; however, when you are writing an actual function over there you will you can pretty much do away with writing these prints over there it is just going to flood out your whole zone it does not have any other significance when you are doing the main training. In the next lecture when we are going to cover down the actual model and then we need down. You will see that these print statements are out over there.

Here to go down by simple what I have within my forward definition is that I have some input X which is taken over there and then the first pass is that I do a convolution with a as per what is defined in conv 1. Now, con one was what is defined over here and that is 2 d convolution of using kernels of size 5 cross 5 and there are 6 such corners which happen. What comes down over here is then passed on to my max pooling and my max

pooling is defined over here. Which is pool one and this max pooling is just 2 cross 2 max pooling present over there. The number of channels still remains the same.

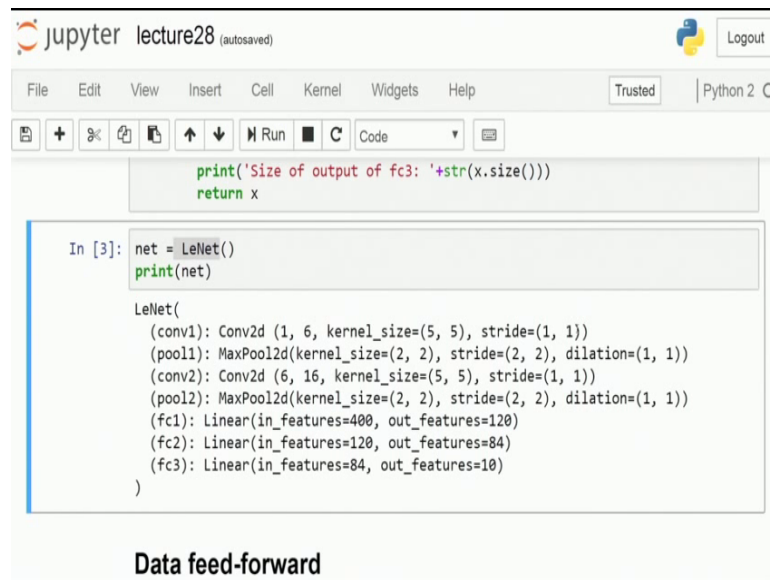
Now, that is what I print down over here now you need to keep something in mind. Over here the number of output channels is 6 over here. The output number of output channels is again going to be 6 and then you do a convolution and get down a number of output channels equal to 16 and then you do a max pooling and here also you end up getting at 16 and then finally, you would see that what is whatever is the size over here that would come down as 16 into 5 into 5 and that is equal to 400 neurons.

Now those 400 neurons are what are invoked over here and then the next part is basically 2. This this what I am doing technically over here is that in the earlier case you had a volume where you had 16 channels and you each you had a size 5 cross 5, but then I will have to arrange them into one single set of neurons. That I can have my fully connected layer being called out. So, the whole purpose of this command of view is basically to flatten it out or minimize it down to one single dimension over there.

Now, once that is minimized then you do your transfer function. Pretty much you can do your transfer function before you try to minimize or even after that that does not make much of a change onto it and then you have your fully connected layer going down now once your fully connected layer is connected. You have your from fc1 the total number of output is 100 and 20 which comes down over here.

Now, you connect down your fc 2 which will take 100 and 20 neurons over here and knock it down to 84 and then you go to fc 3 which comes over here which map strong 84 neurons onto just 10 class of neurons and finally, is the output to be returned. This is this is what becomes my function definition and my forward pass definition over the whole network. Once that is done. Let us try to actually invoke the network and try to print the whole network.

(Refer Slide Time: 11:30)



```
print('Size of output of fc3: '+str(x.size()))
return x

In [3]: net = LeNet()
print(net)

LeNet(
  (conv1): Conv2d (1, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
  (conv2): Conv2d (6, 16, kernel_size=(5, 5), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
  (fc1): Linear(in_features=400, out_features=120)
  (fc2): Linear(in_features=120, out_features=84)
  (fc3): Linear(in_features=84, out_features=10)
)
```

Data feed-forward

If you look into this this is what comes down as my network. Let us zoom out of it. That we can see.

Yeah. The whole network comes down in one line. You see that my model is what is called as LeNet it and because that is how I had defined it over here my class is called as LeNet. Now, within this network the first one is a convolution of 2 d that is what is corresponds to my structure of conv1 on the first level. 0 a 2-d convolution which maps down ah one channel to 6 channels with kernels of 5 cross 5 and a stride of one cross one.

Now, by default I had I had taught actually defined down what my stride over there was and as such it takes down a default side of one in case of convolution as well as I had just mentioned down my kernel size is equal to 5 if I had mentioned down say kernel size is equal to within brackets 5 comma 10. That would mean that the x size is equal to 5 and the y dimension is equal to ten. You can have rectangular kernels as well as we had discussed in the earlier class any generalized kind of a kernel and it becomes easier; however, since we are just sticking to writing down only LeNet. We are not making any change over there.

The next part is you have a max pooling of over a kernel size of 2 plus 2 and a stride of 2 comma 2 and then sort of what is called also called as a dilation of 1,1. Now, the whole purpose over here for this max pooling as you see is quite simple. The whole job is just

to get down a strided max pool coming down now in case you had meant down this stride as one comma one or just going up over here when you had defined this stride as one. Then you can have done something similar to just a maximum operation over a space or what is also called as a gray level dilation over there.

Now, this particular instruction over dilation of one comma one is something which is given down to actually instruct the code that in case there is a parity mismatch at the borders then how much of extra padding you can do. That your operator walks it over now that has a relationship with the stride itself. If the stride was larger than your dilation allowable dilation would also be going down higher now for all general purposes if you are sticking down to using a fixed size of the image and your image sizes are defined in such a way such that all of these are integral. Then you might not even have a residual dilation add it down at any point of time.

Now, comes down the second part of it which is the second convolution. My second convolution is what maps down from 6 to 16 kernels and then via kernel size of 5 cross 5 and stride of one comma one then I have my max pooling. Now, pretty much like whatever is the size of the output which comes from this after my convolution one and my pool one whatever is the size of my output I could have sort of achieved almost a similar sized output if I was giving down a stride of 2 comma 2 as we had discussed in the earlier class.

However, the problem is that there are significant differences in using just stride versus just using a max pooling over there and it has been found out that which strides you tend to lose down on higher frequency components. Which you can actually pressure down in max pool. Down the line as you go on. If you have very small objects which are very significant, but then they might be at a risk of getting eroded out. Because, of the resolution problem just by putting down these max poolings you are still going to reserve and then keep all of these objects and structures quite popularly preserved across the whole depth of the network as output comes down.

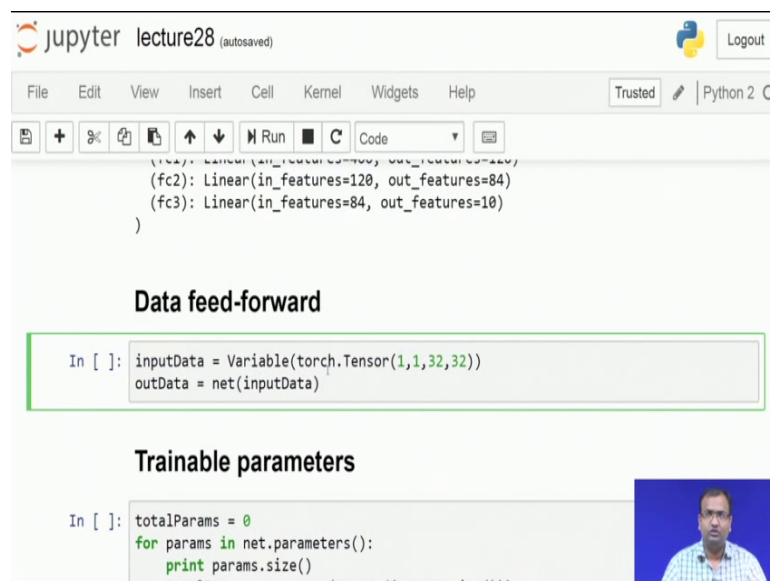
Now, till here it becomes and then the next part is where you have to connect down your linear neurons now this comes on from this definition over here. Which will assume that after this the output is somewhat made flattened and coming down to 400 and that is how it is connected. Now, since that is not part of a network as such but it is just part of some

data processing operator which, we have and for that reason this view is incorporated only within the forward processor. Similarly say relu or these kind of transfer functions which are not to do anything with the memory to be taken down within the network or within the data pipeline over there. They are just transfers over the data they will just be operating over the data and making a change to the nature of the data. You do not have them within the network itself, but they just come down within your forward path definition.

You do not see them in the network architecture. Just trying to free down a network might not all always and giving you exactly, what is the non-linearity? Or the transformations which come down over there and you have to be really clever. One way is like you can actually pooled down somebody by just giving them this network and then without having said about the rest of the stuff lot of people do practice it.

But please do not do that rather define everything in in the full possible way. That it becomes a reproducible kind of over now having said that the final part over here was just 10 neurons over there and what you get down on the final part is just a classification of some sort. We do not have any sort of a linear any sort of a non-linearity in terms of a tan hyperbolic or sigmoid which is given down over here, but it is just what comes down over there and that is what the network has to learn out.

(Refer Slide Time: 16:30)



The screenshot shows a Jupyter Notebook window titled "lecture28 (autosaved)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help), a toolbar with icons for file operations and execution, and a code editor. The code in the editor defines a neural network with three fully connected layers: (fc1) with 120 input and 84 output features, (fc2) with 120 input and 84 output features, and (fc3) with 84 input and 10 output features. Below the code, there are two sections: "Data feed-forward" and "Trainable parameters". The "Data feed-forward" section shows a code cell with the following code:

```
In [ ]: inputData = Variable(torch.Tensor(1,1,32,32))
outData = net(inputData)
```

 The "Trainable parameters" section shows a code cell with the following code:

```
In [ ]: totalParams = 0
for params in net.parameters():
    print params.size()
```

 A small video thumbnail of a person is visible in the bottom right corner of the notebook interface.

Now, let us do a feed forward operation over there now in this feed forward operation, but the whole objective is that I have my inputs of. Here what I am taking down is 32 cross 32 to be conformal with my LeNet. Now, because there they were not taking down 28 cross 28, but you had passed it out. That you do not decay while that by the time you reach over here. I just create a small dummy variable over there in terms of a tensor which has 32 cross 32 spatial span I have one gray channel present over here and this final one is what is called as a bad size.

In the last week lectures when we were doing advanced learning techniques with different kind of learning algorithms. Either it can be an epoch update or a batch update or poor sample wise update. This is what would define down over here now the good thing with convolutional neural networks is that they are really designed to optimize over here across the batch.

When you are passing down the batch it is not just one sample by sample which is going through it, but it is it is as if the whole tensor gets processed out over there and based on the whichever index of the tensor you are looking down it will be making a modification. That also brings us to another challenge is that if I just write it down as a torch dot tensor one comma 3 to cross 32. That would assume this first index to actually be the batch number and the next 2 indices to be what is related to the data.

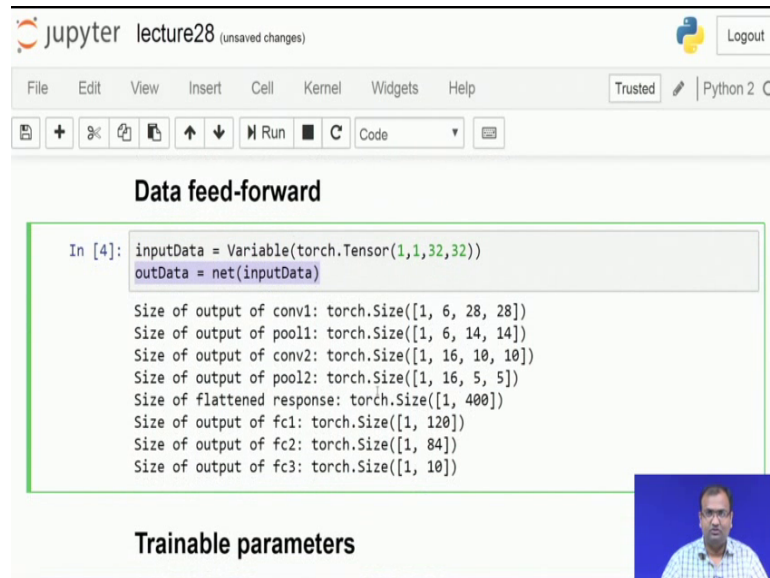
Now, the problem which comes down over here is that my input has to come down the form as the number of channels and the spatial span of the data over there. If my number of channels are not defined then it makes it hard because I have a 2-d convolution. In that case I will have to have my x span and my y span which is given down over here this has to be my number of channels and this has to be my batch size over there.

Even if you just have a grayscale image you still have you will still have to represent it not as a 2-d matrix, but as a 3-d matrix where the third index in in the standard case is equal to one and here it is it is the other way round of indexing. The lowest order index or say in in if you have a normal 2 d matrix given down in your python num pi your psi p then you have your x axis which is your first index then your y axis second index then z axis which is your third index.

Now, here we write it in the opposite way where my x axis is my last most index present over there then comes down my y axis then comes my z axis and then comes down my

batch number over there. If your doing 3 d convolutions you will just have an addition over there and then that is how it goes down. When we come down to them in later on lectures you will find out. How we define around that, but this is really critical I mean for your cns this is one of the main problem which a lot of people would otherwise end up doing over here.

(Refer Slide Time: 19:09)



```
lecture28 (unsaved changes) Python 2 O
File Edit View Insert Cell Kernel Widgets Help Trusted Python 2 O
Data feed-forward
In [4]: inputData = Variable(torch.Tensor(1,1,32,32))
        outData = net(inputData)
Size of output of conv1: torch.Size([1, 6, 28, 28])
Size of output of pool1: torch.Size([1, 6, 14, 14])
Size of output of conv2: torch.Size([1, 16, 10, 10])
Size of output of pool2: torch.Size([1, 16, 5, 5])
Size of flattened response: torch.Size([1, 400])
Size of output of fc1: torch.Size([1, 120])
Size of output of fc2: torch.Size([1, 84])
Size of output of fc3: torch.Size([1, 10])
Trainable parameters
```

Once we have defined this whole thing and then casted that as a variable my next job is actually to get whatever is the output over that. Now, when I do that you see a lot of these things which get printed over there and that is sort of the size of the output which comes over there, but the whole reason was that what I had done is we had these print statements given down over here and they were just printing out the size of each of these variables which comes on the output over there and that is what we are printing out over here as well.

For convolution one you see that it produces a torched tensor. Which has a size of one comma 6 comma 28 comma 28. You had a 32 cross 32 image of one channel you convolve it with a 5 cross 5 kernel and 6 of those unique kernels over there. Your total number of output is going to be 6 that is my batch size was one that is the batch size which I am getting down over here.

Now, comes to my spatial span over there. My spatial span is something like I had 32 minus 5 plus 1. That makes it 28 and divided by this stride which is equal to one itself. It

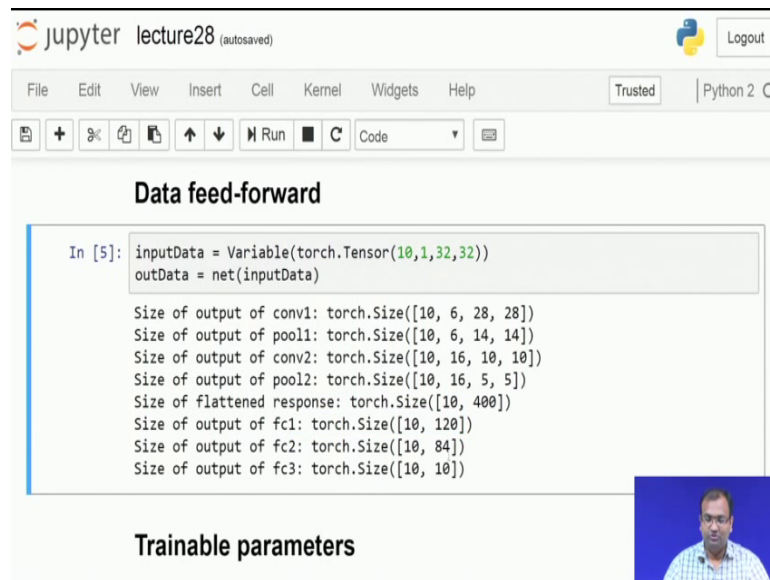
is still stride stays at 28 comma 28. Now, after that when I do my max pooling. That is a max pooling over a 2 cross 2 window and with the stride of 2. That is where my number of channels is still going to remain the same and my x span and y span or the x and y directions the size is going to reduce to half of that and that is what I exactly see over here.

Next what I do is I do a convolution with the second convolution operator and my second convolution operator if I go up here yeah. Over here, my second convolution operator is what maps down 6 channels on to 16 channels over there and with this kernel size of 5 comma 5 and a stride of one. What I am essentially doing down over here is that I have mapped on 6 channels on to 16 channels. The output over here will be having 16 channels now my batch size over there is still going to remain one. Because, I just have one batch of an input data given wrong just one single input data in a batch which has been given down.

Next comes down my result of the convolution. I had a 5 cross 5 cross over there and then my input to this convolution layer was of size 14 cross 14. That makes it 14 plus 14 minus 5 plus 1 divided by one. It is 14 minus 5 divided by 1 plus 1. That whole thing makes it down to 10 cross 10.

That is the output which comes over here then comes down my next pooling layer and that gives me one comma 16 comma 5 comma 5. You see that the total volume which comes down over here. You have one on one side you have your batch which is as of now size of one. In case it was batch of 10 and then you would be seeing down 10 coming down over there. I can actually change it down. In case say I.

(Refer Slide Time: 21:53)



```
lecture28 (autosaved) Python 2 O
File Edit View Insert Cell Kernel Widgets Help Trusted
Data feed-forward
In [5]: inputData = Variable(torch.Tensor(10,1,32,32))
        outData = net(inputData)

        Size of output of conv1: torch.Size([10, 6, 28, 28])
        Size of output of pool1: torch.Size([10, 6, 14, 14])
        Size of output of conv2: torch.Size([10, 16, 10, 10])
        Size of output of pool2: torch.Size([10, 16, 5, 5])
        Size of flattened response: torch.Size([10, 400])
        Size of output of fc1: torch.Size([10, 120])
        Size of output of fc2: torch.Size([10, 84])
        Size of output of fc3: torch.Size([10, 10])

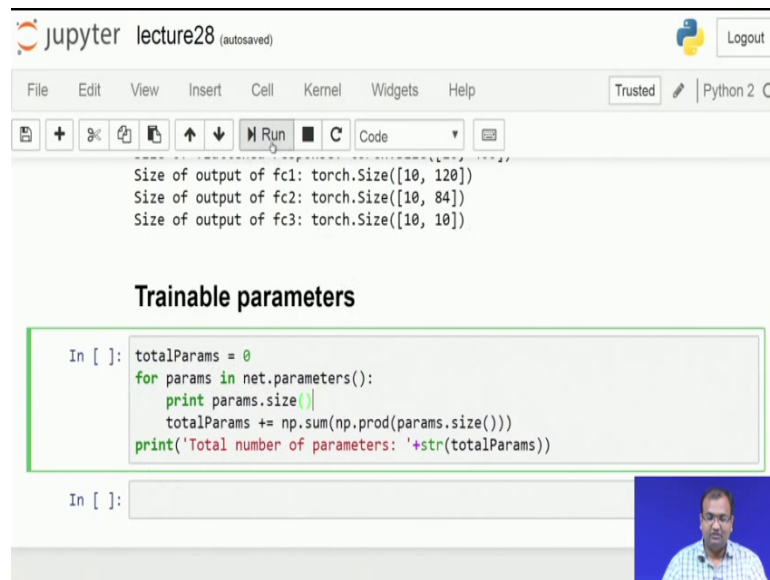
Trainable parameters
```

Just make a modification over here and then write it down you would see that this batch size of 10 comes down over here.

Now, what you need to remember is that the resultant over here is for each batch itself 0 a volume of size 16 comma 5 comma 5 that is 400. In the earlier case when I just had one sample in my batch or my batch size of one then it was just one comma 16 into 5 into 5 that was my whole volume which was being created over here. Now, that I have 10 over here. That is 10 such samples and then 16 into 5 into 5 is what is created.

That is equal to 400 and then what I do is I look into the flattened response my flattened response over there is the same as batch size and the total number of neurons in that volume which is 400. That goes down for my result over here when I'm just doing a flattening using the view function now after that I have my fully connected layers connected down which connects on 400 new neurons to 100 and 20 neurons. That is the output then those 100 and 20 to 84 neurons and then 84 neurons on to 10 neurons. This is what is to look known as how the data behaves as we go through over here.

(Refer Slide Time: 23:05)



The screenshot shows a Jupyter Notebook window titled "lecture28 (autosaved)". The top menu bar includes "File", "Edit", "View", "Insert", "Cell", "Kernel", "Widgets", and "Help". The toolbar contains icons for file operations, a "Run" button, and a "Code" dropdown. The output area displays the following text:

```
Size of output of fc1: torch.Size([10, 120])
Size of output of fc2: torch.Size([10, 84])
Size of output of fc3: torch.Size([10, 10])
```

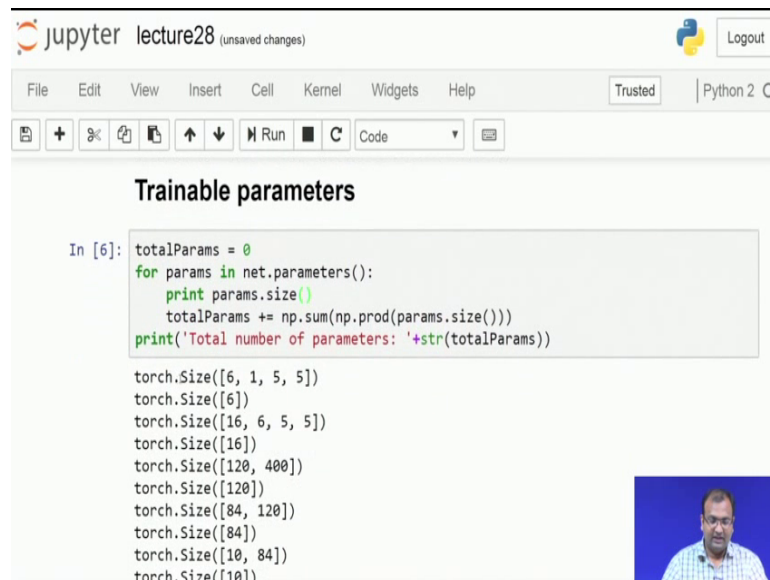
Below the output is a section titled "Trainable parameters". A code cell is highlighted with a green border, containing the following Python code:

```
In [ ]: totalParams = 0
        for params in net.parameters():
            print params.size()
            totalParams += np.sum(np.prod(params.size()))
        print('Total number of parameters: '+str(totalParams))
```

Below the code cell is an empty input field labeled "In []:". A small video thumbnail of a man is visible in the bottom right corner of the notebook interface.

The next part is to also try to investigate down what is the total number of parameters. Because, in the last class we had decided that what is the total number of weights which each of these kernels would be learning down over there. The total number of weights is dependent on what is the total number of input channels and the spatial span and these many times like this quantity time into the total number of such unique kernels. You are going to define in one particular layer that is the total number of parameters which concern. This is a small script which is written down to recursively actually traverse to through the whole network can actually find out the total number of parameters per layer and then find out to the next one.

(Refer Slide Time: 23:42)



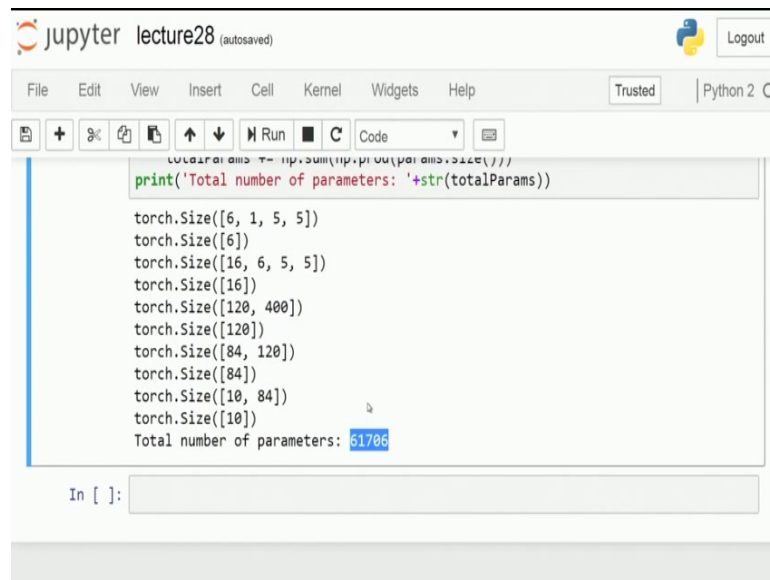
```
totalParams = 0
for params in net.parameters():
    print params.size()
    totalParams += np.sum(np.prod(params.size()))
print('Total number of parameters: '+str(totalParams))

torch.Size([6, 1, 5, 5])
torch.Size([6])
torch.Size([16, 6, 5, 5])
torch.Size([16])
torch.Size([120, 400])
torch.Size([120])
torch.Size([84, 120])
torch.Size([84])
torch.Size([10, 84])
torch.Size([10])
```

In the first one what I have is basically in the in the first layer which is basically a convolution layer. I have one such channel which is mapped out to 6 channels and via 5 cross 5. This is my weight space which create and gets created over there. There are 6 such kernels where each kernels size is one comma 5 into 5. That is a 5 cross 5 and then my z direction over here is just one. Because, the number of input channels. There are 25 neurons which come down over here and then you have this 6 of them being created.

Now, the next part is that corresponding to each of these convolution kernels which you have over. There you will also have a bias function because the resultant of convolution is basically you convolve or do a dot product with whatever is input over there and then you offset it with the bias function this is what comes down. Since there are 6 such unique kernels which are to be computed out over there. Each will be associated with it is own bias function those are s 6. What here my total number of bits which I learned down is 6 into 1 into 5 into 5 plus 6.

(Refer Slide Time: 24:54)



```
totalParams = sum([w.size()[0]*w.size()[1]*w.size()[2] for w in model.parameters()])
print('Total number of parameters: '+str(totalParams))

torch.Size([6, 1, 5, 5])
torch.Size([6])
torch.Size([16, 6, 5, 5])
torch.Size([16])
torch.Size([120, 400])
torch.Size([120])
torch.Size([84, 120])
torch.Size([84])
torch.Size([10, 84])
torch.Size([10])
Total number of parameters: 61706
```

Now, in the next case what I have is from 6 channels I map down to 6teen channels. Each kernel over here is going to be of the size of 6 into 5 into 5. That is about on the volume size it. While it has a spatial span of 5 cross 5 on the volume side it has a span of 6 such channels and there are 16 such unique channels which result out over here. Now, along with these 16 channels. There will be 16 such bias functions as well which will connect down to give an offset and that is why you get down 6teen biases coming down over there.

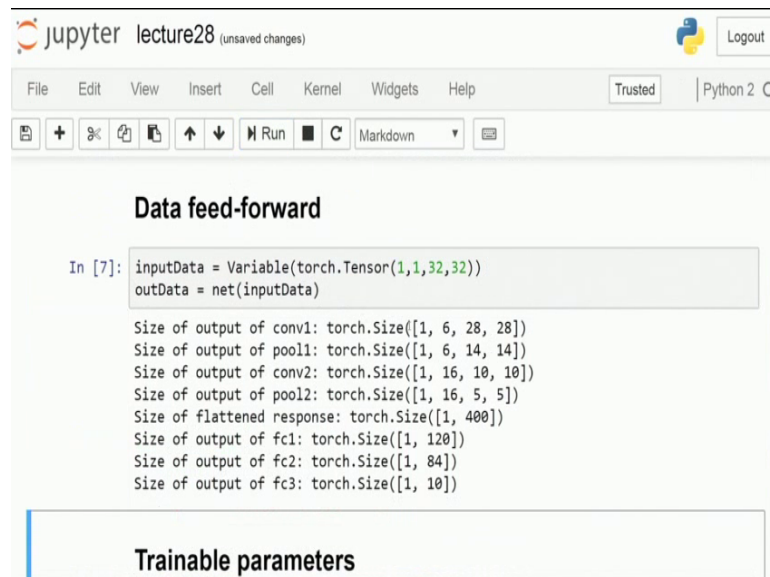
Now, once this part is done then what you have is 400 new neurons which are flattened out and they connect down to 100 and 20 neurons now 400 to 120 neurons is pretty simple it is 120 into 400. What you have done also remember that corresponding to each of these neurons in the first hidden layer which have a 100 and 20 neurons there is also a bias. There will be this 100 and 20 biases which also need to be added.

Going down by the same argument 100 and 20 outputs over there from the previous layer is what connects down to 84 neurons and for that reason you also have 84 biases which comes from then 84 neurons connect down to 10 neurons and then you have 10 biases which comes from and then if you do this multiplication and then the summation across each of them. Multiplication of number of w's in each layer is multiplied all the indices over here then sum done by multiplying it with the next layer. In the next layer 0 just 6. This whole thing plus 6 then the product of all of these numbers plus 16 then the product

of all of these numbers then plus c plus 100 and 20 then product of all of these numbers then sum it up with 804 and that is what you do and finally, you get down this is the total number of learnable parameters.

Do you have within a standard LeNet 60,000 700 and 6 such learnable parameters or the number of bits which will have to be tuned. This this is pretty straightforward now you might ask me a question that does not depend on the batch size no it does not depend on the batch size in any way. Let me do a very simple trick over here I will change it back to one and just run this part.

(Refer Slide Time: 26:59)



The screenshot shows a Jupyter Notebook window titled "lecture28 (unsaved changes)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help), a toolbar with icons for file operations and execution, and a code editor. The code in the cell is as follows:

```
In [7]: inputData = Variable(torch.Tensor(1,1,32,32))
        outData = net(inputData)
```

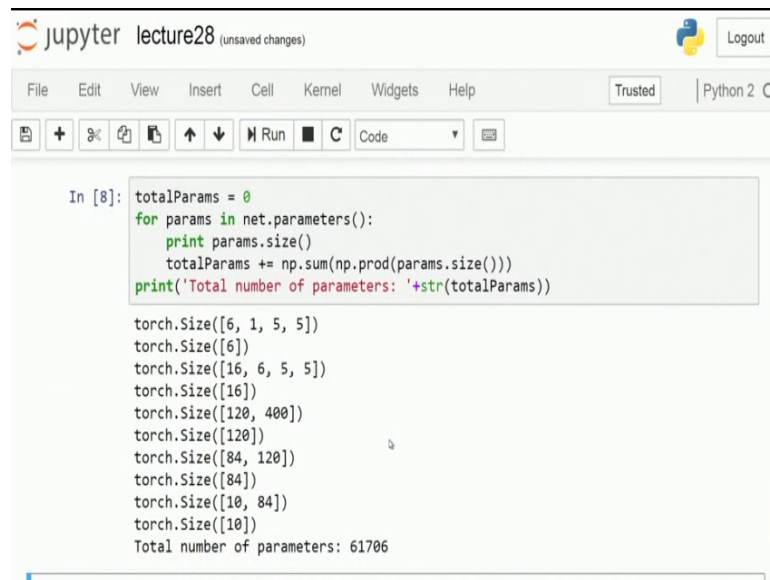
The output of the code is a list of tensor sizes for each layer in the network:

- Size of output of conv1: torch.Size([1, 6, 28, 28])
- Size of output of pool1: torch.Size([1, 6, 14, 14])
- Size of output of conv2: torch.Size([1, 16, 10, 10])
- Size of output of pool2: torch.Size([1, 16, 5, 5])
- Size of flattened response: torch.Size([1, 400])
- Size of output of fc1: torch.Size([1, 120])
- Size of output of fc2: torch.Size([1, 84])
- Size of output of fc3: torch.Size([1, 10])

Below the code and output, there is a section titled "Trainable parameters" which is currently empty.

now, you see the batch size has changed and now let us run.

(Refer Slide Time: 27:03)



```
In [8]: totalParams = 0
for params in net.parameters():
    print params.size()
    totalParams += np.sum(np.prod(params.size()))
print('Total number of parameters: '+str(totalParams))

torch.Size([6, 1, 5, 5])
torch.Size([6])
torch.Size([16, 6, 5, 5])
torch.Size([16])
torch.Size([120, 400])
torch.Size([120])
torch.Size([84, 120])
torch.Size([84])
torch.Size([10, 84])
torch.Size([10])
Total number of parameters: 61706
```

This part over here as well to look into the number of trainable parameters.

See the number of trainable parameters does not change because each weight is not dependent on the batch size of the total number of samples which goes into it. That is that is just for my processing part over there. How, to optimize and then speed it up beyond that it does not have a physical significance on my total number of parameters in any way. And so, do not get confused around with batch size in any of them. This was about getting you introduced to LeNet and then trying to get you into the nitty gritty of how it is defined and how the data passes through it and how you can look into it.

In the next lecture what I am going to do is to run down the whole trainer module over that train it over time and review of what was there within the architecture and then what was the kind of weights which had it had learned. We had done those beautiful weight visualizations for fully connected networks, but then within a cnn. Because, these are now convolutional weights. Then how do they look like is this something to be really amused and intriguing thing to look into it. Till then stay tune and we will get back.