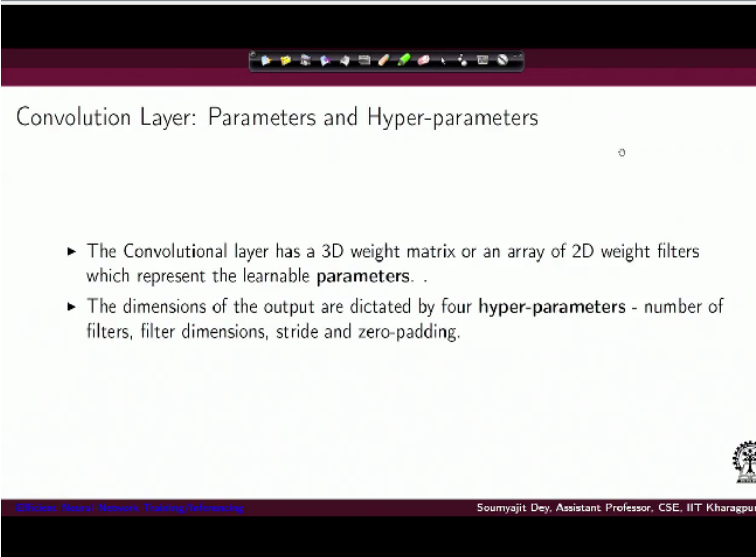


GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Science Education and Research-Kharagpur

Lecture-58
Efficient Neural Network Training/Inferencing (Contd.)


Hi, welcome back to the lecture series on GPU architectures and programming. So, in the previous lecture we have been discussing about convolutional neural networks. And we are due to start with how to define parameters of the neural network.

(Refer Slide Time: 00:34)



Convolution Layer: Parameters and Hyper-parameters

- ▶ The Convolutional layer has a 3D weight matrix or an array of 2D weight filters which represent the learnable **parameters**.
- ▶ The dimensions of the output are dictated by four **hyper-parameters** - number of filters, filter dimensions, stride and zero-padding.



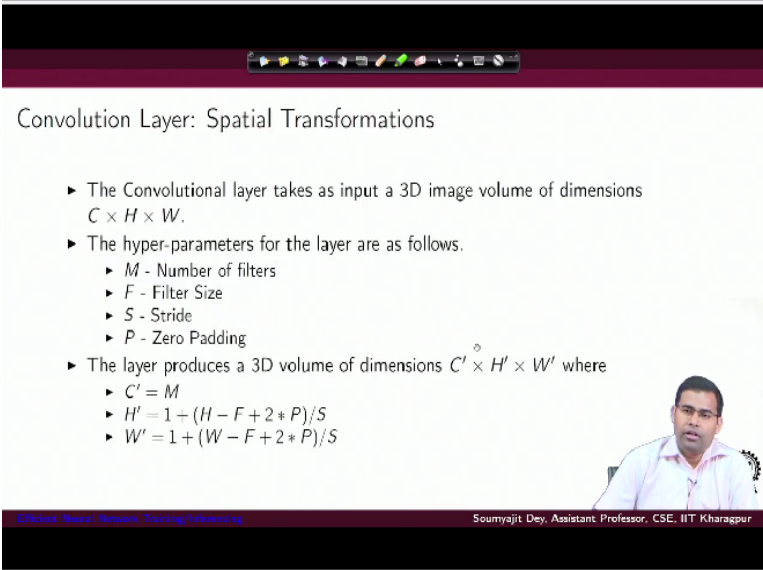
Efficient Neural Network Training/Inferencing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, essentially, in a convolution layer, we have a 3D weight matrix or an array of 2D weight filters and they represent the learnable parameters like what would be the suitable values of these masks right. And also the dimension of the output is dictated by these 4 hyper parameters which is what is the number of filters and the filter dimension stride and zero padding, right. So essentially the architecture of the CNN.

So again, we will just remember what we mean. So essentially, we are talking about how many filters to consider that it is like how many weight matrices to consider what will be the dimension of those filtered filters or the weight matrices. And then when we are performing the convolution, convolution also has an important parameter which is the stride of the convolution. And the other thing is the amount of zero padding that we want to do right.

Because we will soon see that how that affects the convolution operation. Now, where does the hyper parameters because essentially, your neural network architecture, how it really functions. It depends on this initial unknown quantity, right. Because it defines the architecture of the neural network.

(Refer Slide Time: 01:52)



Convolution Layer: Spatial Transformations

- ▶ The Convolutional layer takes as input a 3D image volume of dimensions $C \times H \times W$.
- ▶ The hyper-parameters for the layer are as follows.
 - ▶ M - Number of filters
 - ▶ F - Filter Size
 - ▶ S - Stride
 - ▶ P - Zero Padding
- ▶ The layer produces a 3D volume of dimensions $C' \times H' \times W'$ where
 - ▶ $C' = M$
 - ▶ $H' = 1 + (H - F + 2 * P) / S$
 - ▶ $W' = 1 + (W - F + 2 * P) / S$

Eduncam: Neural Network Training/Interviewing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

That I mean how many filters to take, what are the structures of the filters and finally, once you reset the hyper parameter set then you are trying to discover what will be the value of arguments inside those filters. And there is a weight matrices, so that you get them loss function minimized. So, coming to the convolution layer. So, formally speaking, the layer takes as input a 3D image volume and the dimension is the number of channel cross height cross weight.

And the hyper parameters are let M represent the number of masks of filters, F represent the size of the filter, right, I mean, it is an F cross F matrix like that, S is the stride, and P represents the amount of zero padding, we will soon see what that means. Now, what is this layer going to produce is going to produce a 3D volume of dimension, C prime cross H prime cross W prime. So as you can see there is a I mean, in terms of dimension I have the channels for the input, they are now changing to C prime why because the input image gets multiple different filters applied right. For example, if you see here.

(Refer Slide Time: 03:17)

Convolution Layer

The diagram illustrates a 2D convolution layer. On the left, there is a 3x3x3 input volume with three layers: red (top), blue (middle), and green (bottom). In the center, there are three 3x3x3 kernels: orange (top), blue (middle), and brown (bottom). On the right, there is a 3x3x3 output volume with three layers: purple (top), blue (middle), and green (bottom). The operations are indicated by asterisks and an equals sign.

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

(Refer Slide Time: 03:18)

3D Convolution Example

The diagram illustrates a 3D convolution example. On the left, there is a 3x3x3 input volume with three layers: red (top), blue (middle), and green (bottom). In the center, there are three 3x3x3 kernels: orange (top), blue (middle), and green (bottom). On the right, there is a 3x3x3 output volume with one layer: green (top). The operations are indicated by asterisks and equals signs. A dashed line labeled "Element-wise addition" points from the three intermediate 3x3x3 volumes to the final output volume.

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

I have a 3D image, I applied a 3D mask, I got one output. Because I essentially they are summing up to one mask right.

(Refer Slide Time: 03:26)

3D Convolution Example

Given a 3-D input image I and a 3-D weight filter(mask) W , the convolution operation slides the filter over the image, computes a neighborhood operation (weighted sum) over the elements of I and produces a 2-D output image.

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, the number of outputs that I really get that depends on the number of masks I applied. For example, here if I apply 3 masks, each of them operating on 3D image, the 3D masks, each of them are going to give me 1 2 cross 2 output. So, for using the 3 masks essentially I get a 3D output like this right. So, essentially C prime is M there is a number of filters, but what is H prime. H prime is essentially the height the modified the height of the output. And W prime is the height and width of the output.

(Refer Slide Time: 04:02)

Padding $P = 1$ ✓

$F = 3$ $H = W = 5$
 $H' = W' = 5$

$$H' = 1 + (H - F + 2P)$$

$$= 1 + \frac{5 - 3 + 2 \times 1}{2}$$

$$= 1 + \frac{5 - 1}{2}$$

$$= 1 + 2$$

$$= 3$$

$S = 1$

How to determine that. So, let us consider some examples to determine that and also there is some important parameter that is the zero padding, that how much of extra 0s should I surround the image with, so that I get to decide how many or what should be the output image dimension,

we will see what that means. So consider P equal to 1 that is the padding is 1, that means we have this as the original image.

So this is my image right. So if I just write down for this image, height is equal to 1 2 3 4 5, width is 5. So these are all 5, right. And then I decided to put one layer of 0s, right, so the padding P is 1, right like we already have here. So that increases, right the idea on which the mask can float around. Once I put padding of one layer, then I am essentially saying that will the mask need not start overlaying the image from this point.

But rather it can start from this point and it can come up to this point, this point and go up to this point, right. So that increases the number of shifts that the mask can make, right. Why is that important, because that decides the output images height and width, right. So, as you can see, if I consider P equal to 0, then this mask of size 3 cross 3 can make how many stripes, it can make I mean can make how many movements on the x axis, it can have one location, then the next location, then the third location, right.

So that makes output images width to be 3. But now with the zero padding, what are the maximum number of locations on which the mask can traverse in the x direction. So as you can see, I can really start here, then I can move the mask here. And then here like that, right. So 1 2 3 4 and 5, because one of the most comes here and done right. So 5 possibilities, that is that makes W_{prime} equal to 4.

And since H is also equal to W, so the mask and also take how many locations in this direction 1 2 3 4 5 right, is the last possibility, right. So H_{prime} equal to W_{prime} equal to 5. Now, let us now go and walk back on the formula. So these are formula right. $1 + H - F + \text{twice of } P$ divided by S. So let us see how it comes. Plus twice have P divided by S. Now, in this case since I am moving the mask by one position at a time, so the stride S equal to 1 right.

So if you just check it out, so effectively I have A as 1, the height is 5. Now of course, the mass has a width F, which is equal to 3. That means after this point it really cannot move. So the entire height - 3 +, you get 2 extra positions to move because of the padding right., Since padding is 1,

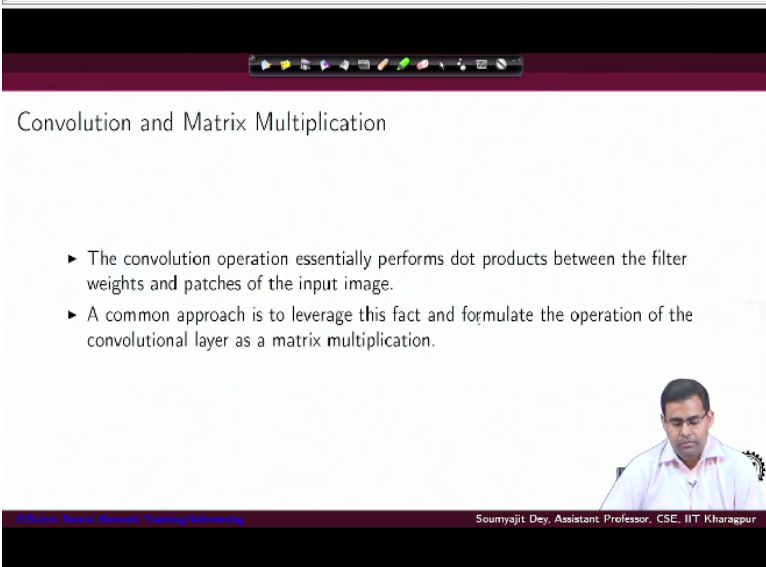
one position this way, one position this way, so 2 into 1, right, and it should work out to $5 - 3 + 2$.

So that is $5 - 1$ by 1. So that is $4 + 1$, that is 5, right. It is coming of properly. And similarly, the calculation can be done for W prime. So for W prime, I will really have $1 + W - F + 2P$ by S , right. But what about modifying the S . If you modify the S and make it 2 the naturally the strides get bigger, so, you get one mask position here, then the next mask position will be here, right. Because this is then $S = 2$, right.

And then the last mask position is here. Because, again you make another stride right, so, 3 positions, right, so, I expect if I put $S = 2$, it should really give me the result as 3 and that is going to happen if you put $S = 2$. So, this is instead of 1, you have $1 + 4$ by 2, that is 2, so you will get a 3 here, right. So, that is how it works. **(Video Starts: 10:06)** So, with $P = 1$, this is how you make progress and compute the convolved output you are moving in strides of 1.

And you get all the values computed. If we just show a small simulation of $S = 2$, just like we discussed you will move in half of 2 and similarly, like just like we discussed the output will be 3cross 3, right. **(Video Ends: 10:31)**

(Refer Slide Time: 10:32)



Convolution and Matrix Multiplication

- ▶ The convolution operation essentially performs dot products between the filter weights and patches of the input image.
- ▶ A common approach is to leverage this fact and formulate the operation of the convolutional layer as a matrix multiplication.

Efficient Neural Network Training/Inference

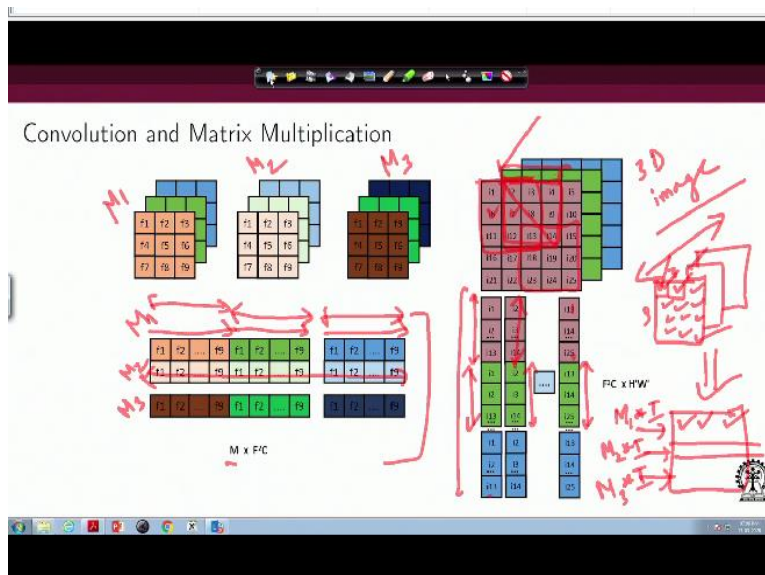
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, just to relate what is the difference now, instead of doing the matrix multiplication, you are now doing convolution and the operation essentially performs dot products between the filter weights and the patches of the input image on which it is working locally, right. By patch I mean the region of the input image on which we are overlaying the mask and doing the operation. Now, a common approach like if you want to do the convolution operation of the CNN efficiently is to leverage this fact.

And formulate the operation of the convolution layer as a matrix multiplication. So let us try and understand what is the I mean advantage. So, just like we discussed, convolution is just individual dot products, right, all the filter weights and the input image, we already have efficient routines for performing matrix multiplication and different optimizations.

So, if we can just adjust the masks weight values and the images, pixel values in such a way that the operation reduces to the matrix multiplication operation that can really help us to get the job done using available or optimized matrix multiplication routines, right. So there is a trick we are going to play for implementing convolution efficiently.

(Refer Slide Time: 12:03)



So let us see how it works. So, this is a very simple and nice example. So, consider on the left hand side, you have these masks, right you have 3D masks, and you what you really want to do is you have mask M_1 mask M_2 and you have mask M_3 at some region of the computation.

What you really want to do is you have this 3D image and you want to transform this 3D image and get the transformed outputs for all these masks M_1 , M_2 and M_3 .

And so is just like you have the input image and you want these transformed outputs to flow to the next layer. And they can really be done in parallel right because they are quite independent operations. So, as an optimization, what we can do is we can arrange the masks and the image data values in such a way that finally, the operation of computing the transformed outputs will boil down to the matrix multiplication operation.

Because as we have seen that individually the mask weights will be component wise multiplied will get component wise multiplied with the image pixel values, right. So, what do we really do is very simple. Let us take mask 1, for mask 1, you arrange all the values in a single row, first layer, and then again from the next layer. Then again from the next layer right, this is all for M_1 . Do the same thing for M_2 , do the same thing for M_3 like that.

And prepare one large matrix. What would be the size of course the number of masks times the size of each row. The size of each row is of course a square right. So this is a square, $F = 3$ here, right F cross F , I mean, I am speaking of capital F , which is the width or height of the filters, right. So, a square times the number of channels, 2 channels F square cross 3, that is the row size multiplied by the number of masks.

So that is one large matrix that you are preparing by using all these masks. Why you do this. Well, that becomes clear when you look at what transformation we make for the image. So this is the image on the right hand side, 3 channels for each channel and sorry, here we are talking about different masks, right. Each of them are 3D masks, and each mask finally gets to 1 row, right. And each of them and the second row is for mask 2, third row is for mask 3, right like that.

Here I have the image. And for the image I have 3 channels for each channel I have a 5 cross 5 matrix right. Now, if you think what is really going to happen, when you do a normal computation, let us say I am trying to convolve the image with mask M_1 , what am I really

supposed to do. So I will take the first layer, let us say this brown layer. And then I am supposed to overlay it with this image, right at this position.

Then the next position assuming a stride of 1, and so on and so forth right. So that gives me how much for the for this position, I am going to do a transformation. For the next position again I am going to do another transformation, right. And for each of the computations, I am going to calculate and store the value, right. So how do I do it in the case that where I have modified this masks into that matrix that I have shown here.

How do I really do it. Because it is expecting the input data now in a different format, right. So what I am really going to do is I am going to duplicate this data into multiple positions. What I do is let us say this initial position of the mask, initial position would have been here. And I am supposed to do F_1 multiplied by i_1 , F_2 multiplied by i_2 , F_3 multiplied by i_3 , F_4 multiplied by i_6 , F_5 multiplied by i_7 like that, right.

Since I have i_1 F_1 to F_9 in a row, I just put this position of the data which is here. In the first overlay position of the mask, I just arrange it in the form for column. So now the column contains i_1 i_2 i_3 i_6 i_7 i_8 i_{11} i_{12} i_{13} right, as you can see if I multiply this row, this row segment, let us say, if I just multiply this row segment with this column up to this, I get the value I was looking for, which I am supposed to put in an output.

So if I just write what I am supposed to do after all this right, so for M_1 , I am going to do these multiplications and I am supposed to get 3 cross 3 matrix right 3cross 3 and a stack of them considering the other components this, this, this, right, they are going to give me a stack of these kind of 3 cross 3 matrices, right. So now, what I am doing is I am supposed to compute each of these locations and they come out when I multiply this with this.

This row segment gets multiplied with this column segment to give this value, right. Again, this row segment, when it gets multiplied with a next column segment is going to give me the next value right. So what is this column segment. Now I am going to shift the mask here, right. So

now I am going to take values i_2 i_3 i_4 i_7 i_8 i_9 . So, i_2 up to i_{14} , right. So, essentially this much, as you can see from i_2 to i_{14} .

And at the last I am expecting i_3 to i_{15} . So i_2 to i_{15} right, sorry, I mean, sorry, and last means this location. So in this location, I am expecting these values, right. So that is i_3 to i_{25} , as you can see i_3 to i_{25} , right. So all I am saying is I want this individual values to be computed, right. But what I do is I arrange them like these columns, I arranged the input mask like this row segment.

And I multiply this row segment with all these columns to get the first layer of the 2D output done right. Now, you can understand the same thing is going to work for the next set of channels right, because now, if I just continue with the operation, then whatever mask values are there, in the next layer of M_1 , they get multiplied with similar things here, right. And how does that help because I am not only going to compute this layer wise weighted values weighted sums.

But the next thing is I am supposed to add them across layers. Since I am supposed to add them across layers. Here in this case, it is getting reduced to an entire matrices row multiplied by column computation, right, because I have already computed M_1 up to this position, then you compute all the locations from F_1 to F_9 for the intermediate layer in the mask and you are already having the data for this second layer arranged right here right.

So, all you get is you are while you are computing this locations, you are also simultaneously computing the backward location for the same mask. That means, you are computing this as well as this value, but these are not the individual values that you are finally interested in what you are finally interested in, you are essentially interested in this entire column, entire row, pair wise multiplications and then the final addition.

That is going to give you the final 2D output matrix considering 1 mask M_1 right, I hope this is clear. So, once you do this you get for M_1 what is going to be the value here, the next value, the next value and so on so forth right. So, for M_1 I am finally expecting what 9 values right and

then they get multiplied here and finally, what do I really have essentially, and when you add them up you are going to have 1 value here, next value like that so on so forth.

But, overall you have successfully computed the entire thing for the 2D image, right. And then what are the total number of positions that you have covered. If you look at it minutely, it is exactly same as the size of the output, right. Considering 1 mask, but finally, you are going to have so many of them right. So, the same thing will now repeat for mask 2, because you now have the entire mask 2 capture in a single row.

So, you just keep on applying this mask 2 on all the columns individually, right. So, in that way, you will have all the values that you are going to compute for mask 2 available for these columns, get them working with the rows and you get the final results for mask 2 computed in the next row. First row is computing all the values that you would have required for mask 1, right. So, this is essentially mask 1 transformation with image i .

Next row is mask 2 transformation with image i . Next row is mask 3 transformation with image i , so on so forth. I hope this is clear. Now, how are the entries holding up here. For mask 1, how many entries are going to be there. Well, it is really the number of overlay positions I can have for mask 1, which is essentially 3 cross 3, so I essentially get 9 entries. And that is what is going to happen because I have actually got 9, all the 9 possibilities for mask already captured here, right.

In terms of each of the columns, so I am really having 9 columns here. So here in 1 row with 9 entries, I am going to get the entire final output for mask 1, then for mask 2 in the next row, then for mask 3 in the next row, and that is how it continues right. So as you can see, the whole point of doing this transformation and the multiplication is I am successfully able to convert this convolution of many weight matrices into matrix multiplication problem.

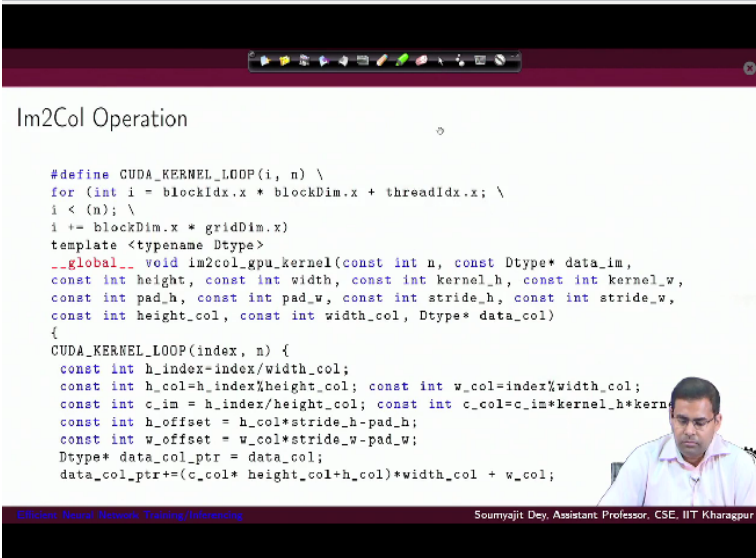
Where all these convolutions outputs are computed in parallel right. What is the overhead. Of course, you have some data overhead, because essentially you are creating a data layout where entries are getting duplicated. Why is that happening. Well, here you do not have any duplication

you are just considering each of the masks and taking all of that together into a large matrix wherever arranged it in a smart way.

So that the number of rows is same as the number of masks, the number of columns is same as mask size times the number of layers, which is obvious here, right. But so there is no duplication. The duplication is the way that you are arranging the image, right. Because here you are come for each let us this is the I mean, one layer of the image, for that you have got 25 possible values and you are duplicating them.

Because you are trying to keep each of the possible overlay instances of the mask separate, since you are going to have 3 cross 3 overlay instances, 9 overlay instances, so you are having to i 2 hear i 2 here you are having i 3 here i 3 will also be here. So, there will be many duplications like this, right. But as we are considering parallel programming language, so essentially we will have multiple threads, who will be creating this data structure in parallel. And then engaging into the matrix multiplication task.

(Refer Slide Time: 25:41)



```
#define CUDA_KERNEL_LOOP(i, n) \
for (int i = blockIdx.x * blockDim.x + threadIdx.x; \
i < (n); \
i += blockDim.x * gridDim.x)
template <typename Dtype>
__global__ void im2col_gpu_kernel(const int n, const Dtype* data_im,
const int height, const int width, const int kernel_h, const int kernel_w,
const int pad_h, const int pad_w, const int stride_h, const int stride_w,
const int height_col, const int width_col, Dtype* data_col)
{
CUDA_KERNEL_LOOP(index, n) {
const int h_index=index/width_col;
const int h_col=h_index/height_col; const int w_col=index%width_col;
const int c_im = h_index/height_col; const int c_col=c_im*kernel_h*kernel_w;
const int h_offset = h_col*stride_h-pad_h;
const int w_offset = w_col*stride_w-pad_w;
Dtype* data_col_ptr = data_col;
data_col_ptr+=(c_col* height_col+h_col)*width_col + w_col;
```

So, the creation of this image to column thus what it is popularly called image 2 col col column operation is available in most of these free software's like the deep learning libraries like caffe. And this is one of those examples which you have just given, you can have a look at it with the

slides that we provide, I think it is not required that we keep on talking about what is happening here in each of the lines.

(Refer Slide Time: 26:09)

Im2Col Operation

```
const Dtype* data_im_ptr = data_im;
data_im_ptr += (c_im * height + h_offset) * width + w_offset;
for (int i=0; i<kernel_h; ++i) {
  for (int j = 0; j < kernel_w; ++j) {
    int h_im = h_offset + i; int w_im = w_offset + j;
    *data_col_ptr = (h_im >= 0 && w_im >= 0 && h_im < height && w_im < width)
    ?
      data_im_ptr[i * width + j] : 0;
    data_col_ptr += height_col * width_col;
  }
}
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, just to repeat.

(Refer Slide Time: 26:13)

Convolution and Matrix Multiplication

M x FC

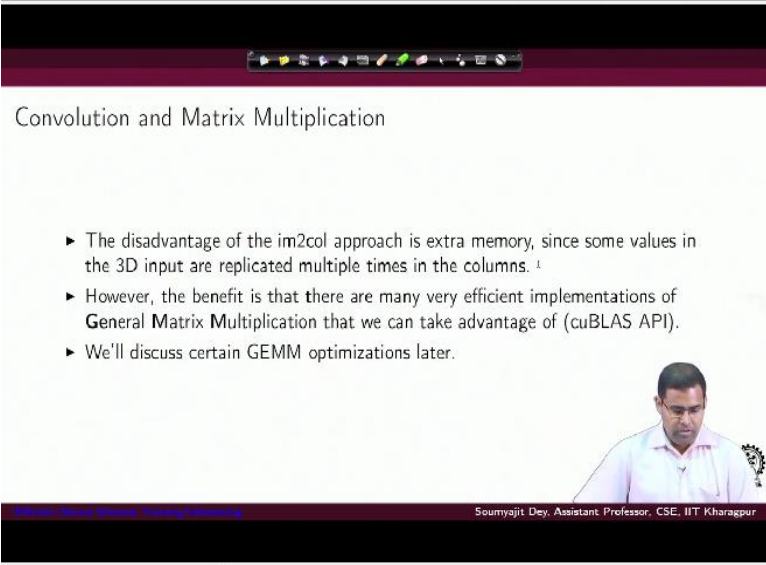
FC x H'W'

All we are doing is we are launching sufficient number of threads, so that each thread is responsible for copying one possible overlay of the image, right, I mean, the mask position. So, for this, I will have one thread will do the copy here. Another thread will do the copy here. Another thread will do the copy here, another thread for this, so on so forth, right. So, in that way the threads will be doing and doing the copy operations in parallel.

And well, will there be some coalescing happening here. I believe so, because if you see, let us say one thread is doing copy of all these data, the next thread is doing copy of these data points. So naturally, I mean, when this guy is copying this one, the next one is going to copy this, the next one will be copying this. So, you have scope of memory coalescing, and successful use of shared memory and all that, which you can really explore using the optimizations we have discussed earlier right.

But if you just look at a vanilla code that we have provided in the next slide, we are not really making use of shared memory or other things, but we are engaging multiple threads to do the copies parallel from the global memory, and you can check as a task. How much is the scope for memory optimization here. How much is the coalescent here and all that right.

(Refer Slide Time: 27:43)



Convolution and Matrix Multiplication

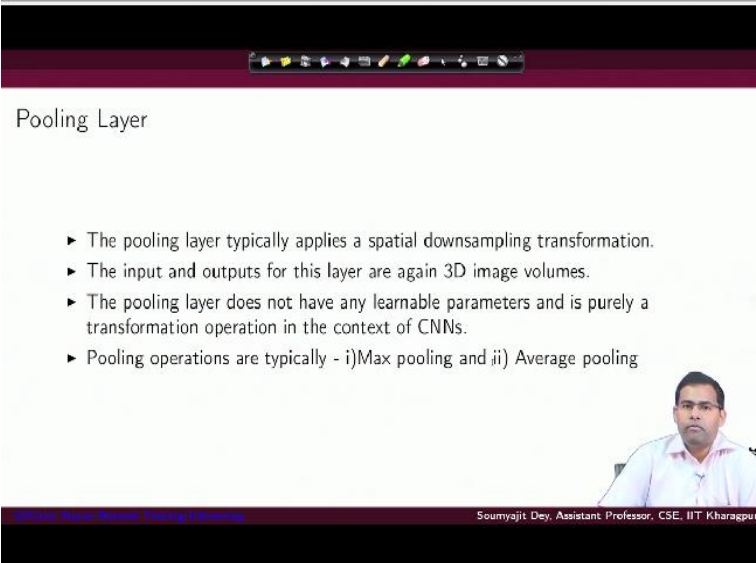
- ▶ The disadvantage of the im2col approach is extra memory, since some values in the 3D input are replicated multiple times in the columns. ¹
- ▶ However, the benefit is that there are many very efficient implementations of General Matrix Multiplication that we can take advantage of (cuBLAS API).
- ▶ We'll discuss certain GEMM optimizations later.

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, that is kind of a disadvantage, we can say that when I am doing the image 2 column approach, I require extra memory, since some of the values are replicated multiple times in the columns. But the good thing as we have discussed is that there are really many, many efficient implementations of the matrix multiplication or the way people popularly call it is general matrix multiplication or GEMM.

And people can take advantage of the GEMM operations. Once this image 2 column transformation is ridden. For example, there is this very popular CuBLAS library, which contains efficient implementations of GEMM.

(Refer Slide Time: 28:24)



Pooling Layer

- ▶ The pooling layer typically applies a spatial downsampling transformation.
- ▶ The input and outputs for this layer are again 3D image volumes.
- ▶ The pooling layer does not have any learnable parameters and is purely a transformation operation in the context of CNNs.
- ▶ Pooling operations are typically - i) Max pooling and ii) Average pooling

©2020, All rights reserved. Teaching AI Engineering Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Some of those GEMM optimizations are also things that we will be discussing later on. **(Video Starts: 28:29)** Well, that would be all about our convolution layer. The next thing that comes into space is the pooling layer. Because if you remember our convolution architecture, so ((28:38)) we will just go back once. So, here, we talked about this multiple layers. So input layer is there, then convolution then pooling and then the RELU layers, right.

So essentially, the pooling is nothing but you are doing a down sampling of the image. **(Video Ends: 29:01)** So summarize is typically a special down sampling transformation, the inputs and outputs both will be 3D image volumes, right and but there are not really any learnable parameters, unlike the CNNs and the convolution layers, but what do you do is you are trying to reduce the input image size.

Why are you interested in the pooling. Because remember that while in the front, the input layer you may have these heavy images with multiple channels and all that but at the end if you are doing a classification kind of problem, you are only interested in last this class course right. So, you want a linear array with different scores field it up right. So, you there has to be intermediate

layers interspersing the convolution layers where you will try to reduce the output image sizes right.

Now, there are several possible pooling operations typically one is max pooling and the another is average pooling. **(Video Starts: 30:01)** So what does that really mean. So here by let us say these are bigger dimension image, and I am trying to reduce it to a smaller dimension. So I can do a max pooling, that means I choose a mask of this size. And for all these values, I just put the maximum here, I just move it over a stride of the same size.

And again, I put the maximum value here. This is just a transformation, which is kind of down sampling the because I am moving from a more information representation to a smaller information. And compact representation and losing information here. So that is the down sampling right. So I could have done max operation or I could have also done a I mean average operation. So both are possible in pooling layers, right. **(Video Ends: 30:51)**

(Refer Slide Time: 30:52)

Pooling Layer: Spatial Transformations

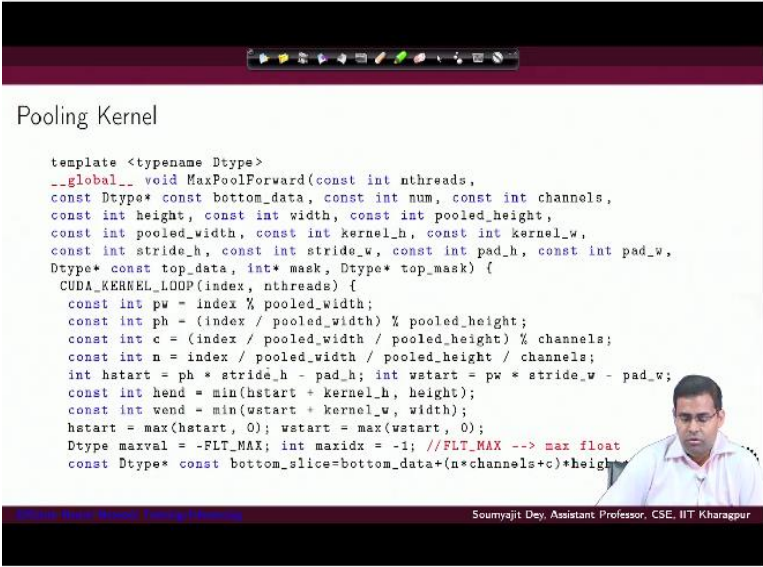
- ▶ The Pooling layer takes as input a 3D image volume of dimensions $C \times H \times W$.
- ▶ The hyper-parameters for the layer are as follows.
 - ▶ F - Filter Size
 - ▶ S - Stride
- ▶ The layer produces a 3D volume of dimensions $C' \times H' \times W'$ where
 - ▶ $C' = C$
 - ▶ $H' = 1 + (H - F) / S$
 - ▶ $W' = 1 + (W - F) / S$

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So just to formulize I am considering 3D images here for pooling, number of channels, times height, width and then the hyper parameters of the layer are well again have to decide that what is the filter size that I am going to talk about, for example, in this picture, there is a 3 cross 3 filter, and it is moving in strides of 3, if I increase the stride size, I am going to have more amount of down sampling as we can see, right.

So once I choose the filter size and the stride size, what the pooling, I mean, out of the pooling, I get a modified transformed output of dimension C prime H prime W prime, where naturally the C prime remains C same because nothing changes here, the number of channels is same, but of course, the height and the width values get modified, I mean following similar formulas that we like we have elaborated earlier. So it is simply 1 plus H minus the filter size divided by the stride.

(Refer Slide Time: 31:52)



```
template <typename Dtype>
__global__ void MaxPoolForward(const int nthreads,
const Dtype* const bottom_data, const int num, const int channels,
const int height, const int width, const int pooled_height,
const int pooled_width, const int kernel_h, const int kernel_w,
const int stride_h, const int stride_w, const int pad_h, const int pad_w,
Dtype* const top_data, int* mask, Dtype* top_mask) {
  CUDA_KERNEL_LOOP(index, nthreads) {
    const int pw = index % pooled_width;
    const int ph = (index / pooled_width) % pooled_height;
    const int c = (index / pooled_width / pooled_height) % channels;
    const int n = index / pooled_width / pooled_height / channels;
    int hstart = ph * stride_h - pad_h; int wstart = pw * stride_w - pad_w;
    const int hend = min(hstart + kernel_h, height);
    const int wend = min(wstart + kernel_w, width);
    hstart = max(hstart, 0); wstart = max(wstart, 0);
    Dtype maxval = -FLT_MAX; int maxidx = -1; //FLT_MAX --> max float
    const Dtype* const bottom_slice=bottom_data+(n*channels+c)*height
```

So that gives you a smaller and simpler a transformed image to work with right, you can just do a simple implementation of pooling kernel, so, this is also taken from available repositories. Again, we are not really getting into discussing this operation is quite simple.


(Refer Slide Time: 32:08)

Pooling Kernel

```

for (int h = hstart; h < hend; ++h) {
    for (int w = wstart; w < wend; ++w) {
        if (bottom_slice[h * width + w] > maxval) {
            maxidx = h * width + w;
            maxval = bottom_slice[maxidx];
        }
    }
    top_data[index] = maxval;
    if (mask) {
        mask[index] = maxidx;
    } else {
        top_mask[index] = maxidx;
    }
}
}

```



©2016 Deep Neural Network Training | Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And the idea is that by doing the pooling, you are conveniently decreasing the transformed image sizes. So, that at the end of the pipeline, you really get the class course.

(Refer Slide Time: 32:24)


RELU Activation Function

```

__global__ void ReLUForward(const int n, const Dtype* in, Dtype* out,
Dtype negative_slope) {
    CUDA_KERNEL_LOOP(index, n) {
        out[index] = in[index] > 0 ? in[index] : in[index] * negative_slope;
    }
}

```

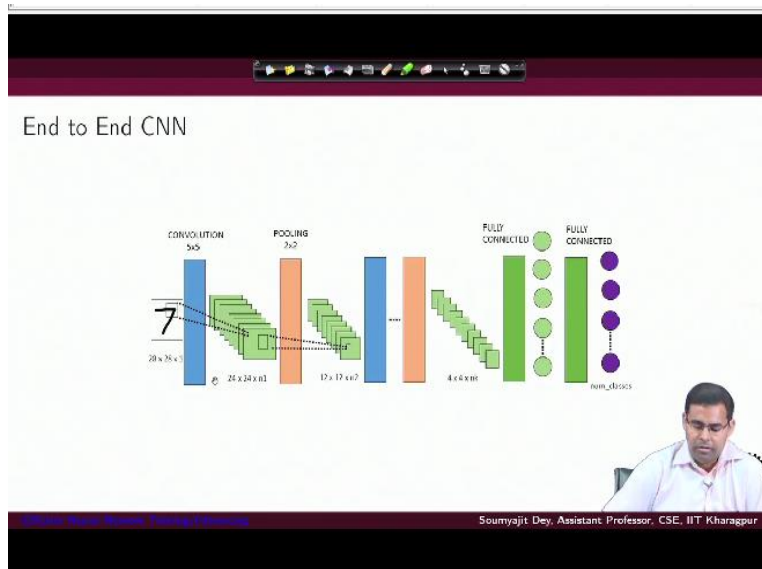
Rectified Linear Unit is the standard activation function used in CNNs.



©2016 Deep Neural Network Training | Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, the other layer that you have is the activation layer, which is very similar to the ones that we have been talking about earlier. And for the activation function in CNN, the standard the one that we use is the rectified linear unit, which we have discussed earlier during our discussion on current neural networks, right.

(Refer Slide Time: 32:43)



So, if I look at the structure of an end to end CNN, you have an image here, and then you will have convolution layers, followed by pooling layers, again convolution layers, followed by pooling layers like that. And, of course, after every convolution, you are also I mean, you are going to have RELU layers also, right. So again, if I just go back into that slide where we define the CNS. So that is what you have input, then convolution, pooling and then the RELU right.\

And I mean this will repeat many times you have convolution again, the pooling was down sampling and then RELU and all that. And the RELU is, like we have discussed earlier, it is giving you the required non linearity. And so in that way, you keep on transforming the image and also down sampling your size. And at the end, you have a fully connected layer, just like our normal neural network architecture.

And the fully connected layer will be reducing this, the transformed image to the final class course for the output layer where you have the different number of nodes as the number of classes that were talking right.

(Refer Slide Time: 34:01)

Recap: Backward Propagation

- ▶ After the forward pass is completed, the first error term is calculated as $Y - O$ where Y is a column vector of true labels, O is a column vector of predicted labels.
- ▶ During the backward pass, for the layer with weight matrix W' , two items are computed:
 - ▶ The error term δ^1 is calculated as an elementwise product: $\delta^1 = (Y - O) \cdot f'(Z)$ where $f'(Z)$ is a column vector where each value represents the gradient of the activation function in the given layer.
 - ▶ The gradient of the layer i.e. $\frac{\partial J}{\partial W'}$ which is $A^T \delta^1$ where A was the input matrix for that layer.



So that is how the CNN pipelines are arranged. The next thing that we will be looking into is how the data really flows across this pipeline, how the backpropagation and of course, the feed forward propagation works specific to CNN pipelines, what are the computations involved and how are they carried out okay. So with this will end the current lecture, thank you for your attention.