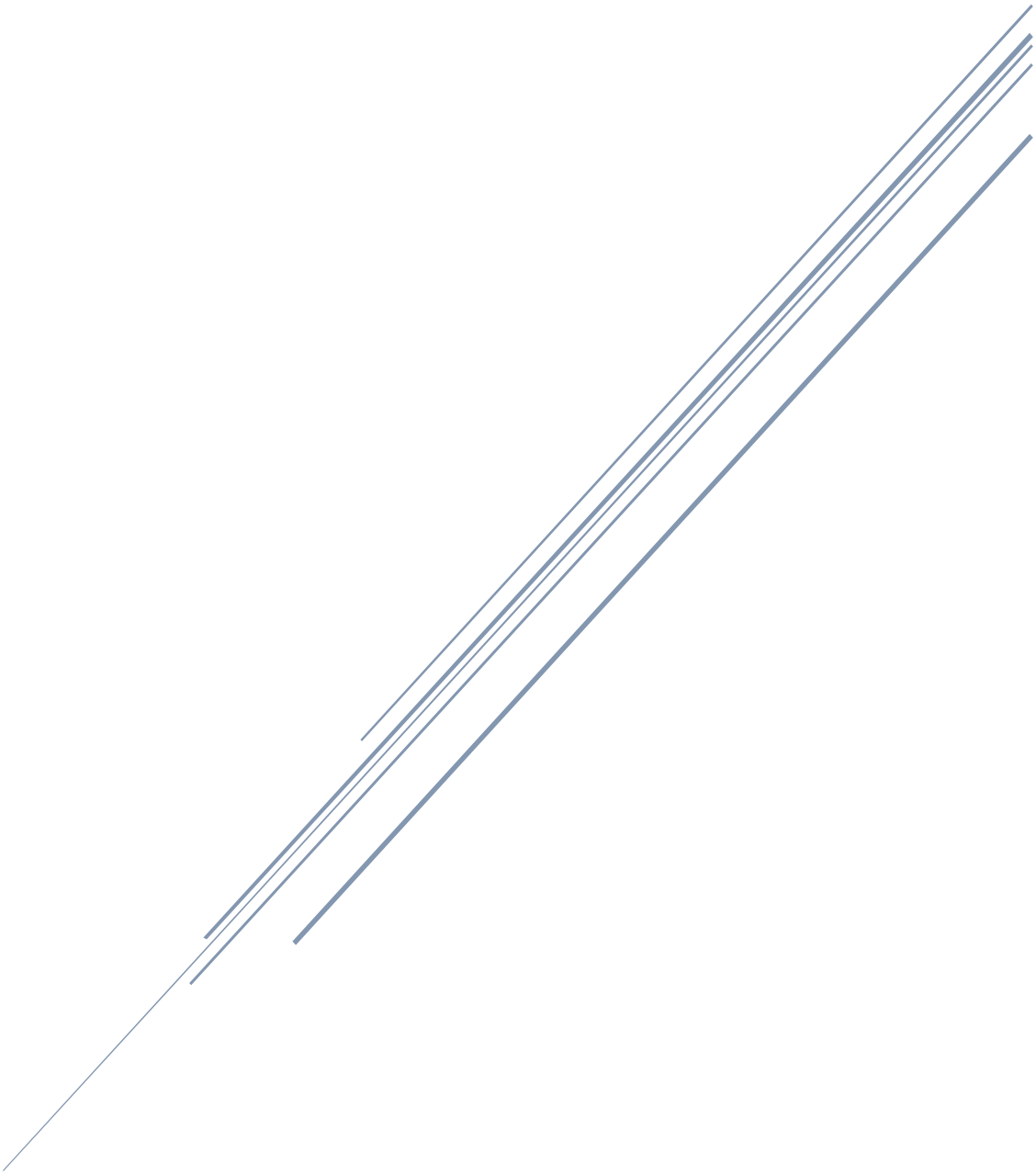
NETWORKS FOR COMPUTER VISION

Module 6



v-cardona

Deep Learning with Python and PyTorch

Contenido

Convolution

Convolution preserves the spatial relationship of the pixels, because in similar pictures we focused on the relationship of the pixels, not the position where they are.

Convolution is a linear operation similar to a linear equation, dot product, or matrix multiplication. Convolution has several advantages for analyzing images. As discussed in the video, convolution preserves the relationship between elements, and it requires fewer parameters than other methods.

You can see the relationship between the different methods that you learned:

$$\textit{linear equation}: y = wx + b$$

$$\textit{linear equation with multiple variables where } \mathbf{x} \textit{ is a vector } \mathbf{y} = \mathbf{wx} + b$$

$$\textit{matrix multiplication where } \mathbf{X} \textit{ in a matrix } \mathbf{y} = \mathbf{wX} + \mathbf{b}$$

$$\textit{convolution where } \mathbf{X} \textit{ and } \mathbf{Y} \textit{ is a tensor } \mathbf{Y} = \mathbf{w} * \mathbf{X} + \mathbf{b}$$

In convolution, the parameter **w** is called a kernel. You can perform convolution on images where you let the variable image denote the variable X and w denote the parameter.

Is like that we are going to cut the spaces of the image and rotate it.

What we're going to do is going to take the image and then we're going to take something called the kernel, we're going to perform convolution, and we're going to get an activation map. The parameter w is the kernel and the b is the bias.

$$z = wx + b$$

$$Z = W * X + b$$

```
conv = nn.Conv2d(in_channels=1, out_channels=1,kernel_size=3)
image=torch.zeros(1,1,5,5)
image[0,0,:,2]=1
image
z=conv(image)
```

- Torch.zeros(number of images in our tensor, number of channels(1 for gray), number of rows, number of columns)

Starting from top left, we put the kernel and multiply each element by the value of the kernel, and move the kernel repeating for each value to construct the Z matrix.

The number of steps that we have to do in one row are M-K+1 (where M are the number of columns of the image and K the number of columns of the activation map).

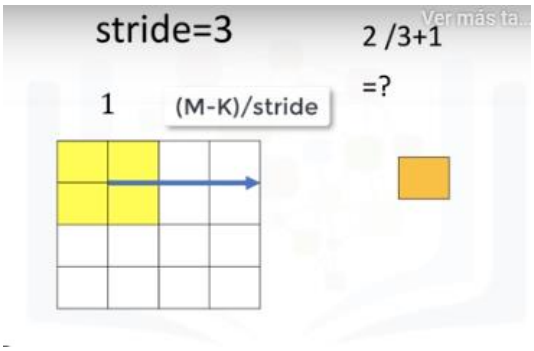
Stride

It is the jumps that we do when moving the activation map, 1 is for moving the map for all elements and 2 for jumps 1.

The parameter stride changes the number of shifts the kernel moves per iteration. As a result, the output size also changes and is given by the following formula. To determinate the size we calculate (M-K)/stride + 1.

Left picture is the matrix that we walk into and the right one is the activation map.

If the stride it too large that we do not have matrix to jump we have zero padding.



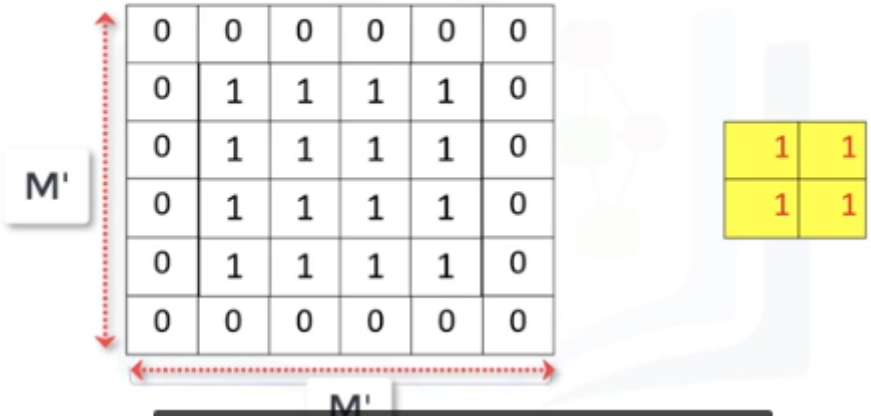
Zero padding

Increase the size of the matrix, adding row and column with zero elements.

```
nn.Conv2d(in_channels=1, out_channels=1,kernel_size=3, stride=2, padding=1)
```

- With a value of padding of 1, it will be added two additional rows of zeros and two additional columns of zeros.

$$M' = M + 2 \times padding = 4 + 2 \times 1 = 6$$



PyTorch

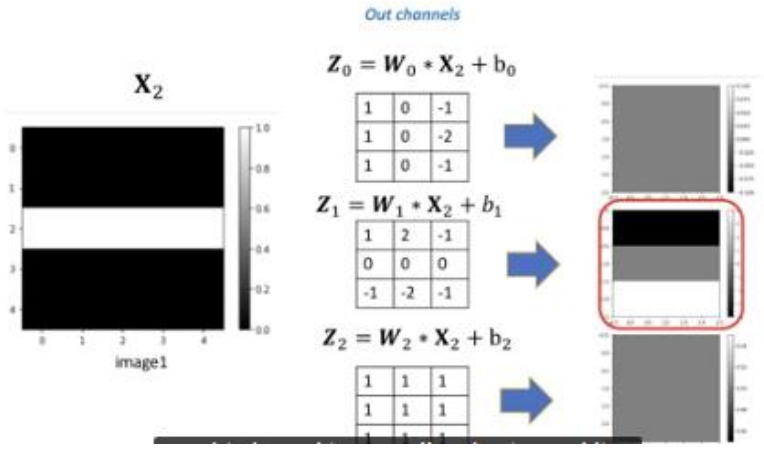
- Two dimendional convolution object:
  conv = nn.Conv2d(in_channels=1, out_channels=1,kernel_size=3)

Multiple channel convolution

```
conv1 = nn.Conv2d(in_channels=1, out_channels=3,kernel_size=3)
```

mini-batch x Channels

```
image1=torch.zeros(1,1,5,5)
image1[0,0,:,2]=1
```



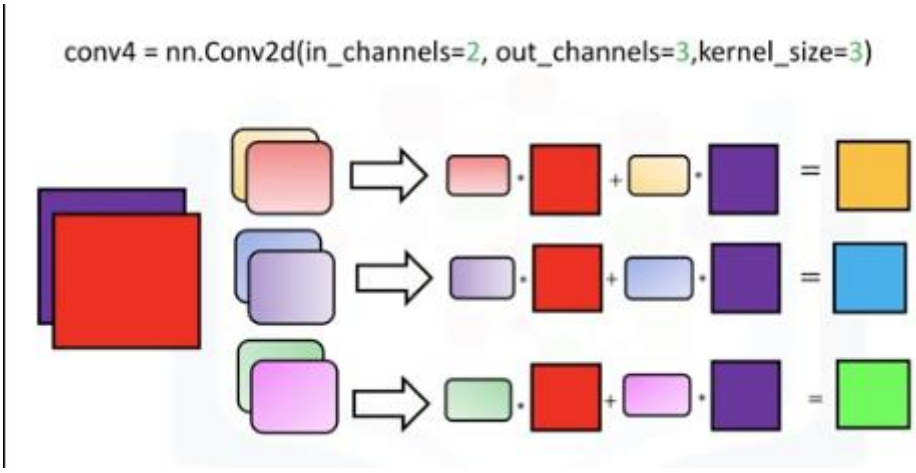
This image is for one input and multiple output channels.

Different kernels detect different features in the image. Each kernel has his own bias.

With multiple input channels, so, for multiple input channel convolution, we'll have one output, and for every input channel we'll have our own kernel, we'll perform convolution with that kernel with the corresponding input channel, we'll add the results together, and we'll get an output.

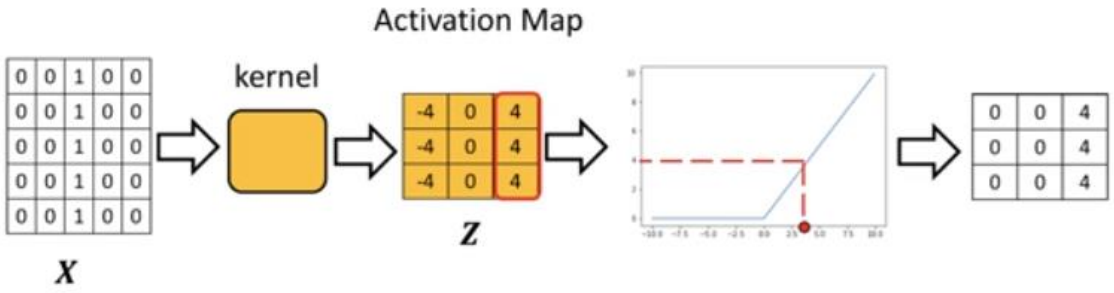
I like to use a matrix analogy, where our parameters are like a row vector and our input images are like a column vector, and we'll perform the dot product operation, but instead of performing multiplication we'll perform convolution.

With multiple input and outpus:



There is one kernel for each input and also multiply by the outputs, so in the image above, the quantity of kernels are 2*3=6.

Activation functions




```
import torch.nn.functional as F

image=torch.zeros(1,1,5,5)
image[0,0,:,2]=1

Z=conv(image)

A=F.relu(Z)
```

image

0	0	1	0	0
0	0	1	0	0
0	0	1	0	0
0	0	1	0	0
0	0	1	0	0

Z

-4	0	4
-4	0	4
-4	0	4

Max pooling

Instead of perform the convolution operation we choose the max value.

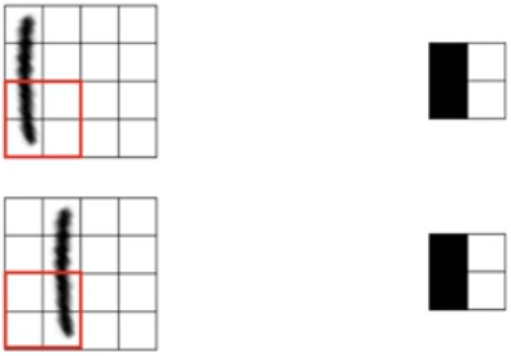
```
max=torch.nn.MaxPool2d(2,stride=1)
max(image)
```

- Where the first parameter is the size, and the second the stride. If the stride is None, it will be pass by areas.

1	2	3	-4
0	2	-3	0
0	2	3	1
0	0	0	0

2	3
2	3

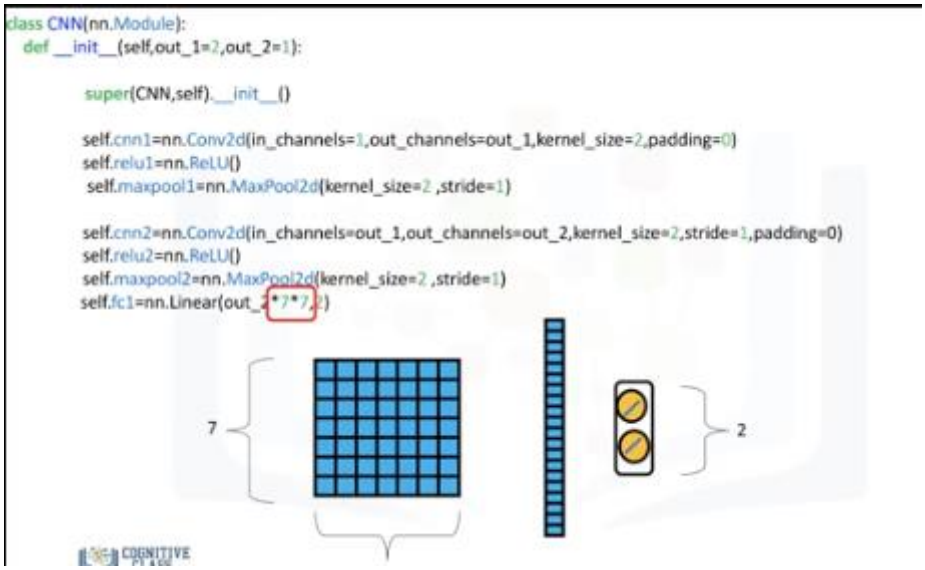
Example:



Convolution neural network

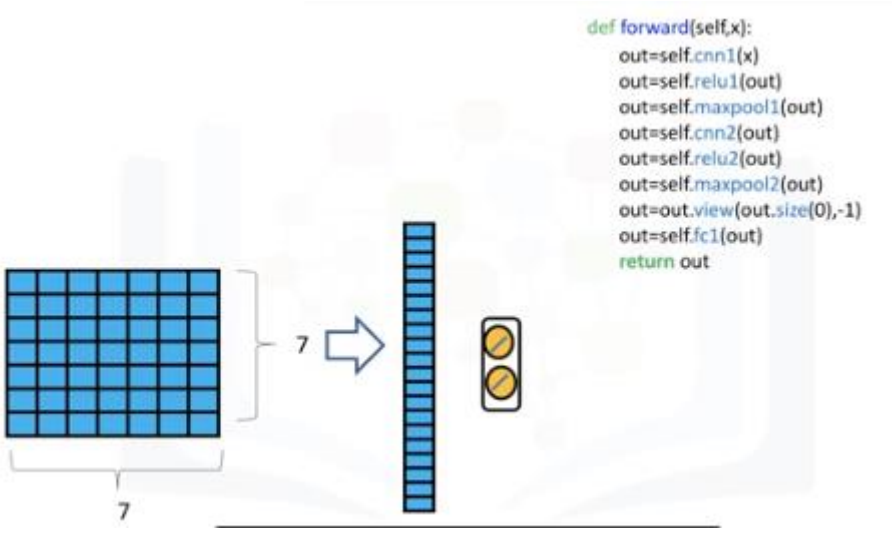
The output of the output channel in convolution will be the input to the neural network but flatter it.

- Constructor



Flatter; take into account the size of the image after all these layers.

- Forward:

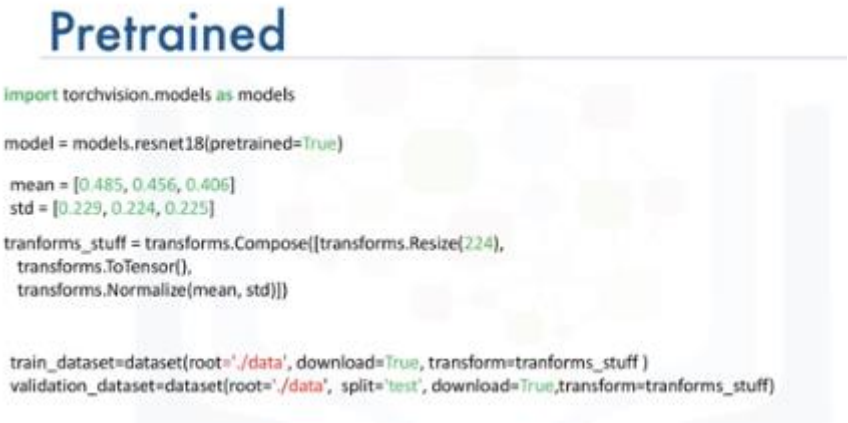


The only difference is, we are not going to have to reshape our tensor, because we are going to leave it as a rectangular image.

```
for epoch in range(n_epochs):
    for x, y in train_loader:
        optimizer.zero_grad()

        z=model(x)

        loss=criterion(z,y)

        loss.backward()

        optimizer.step()
correct=0

    for x_test, y_test in validation_loader:
        z =model(x)
        _,yhat=torch.max(z.data,1)
        correct+=yhat==y_test).sum().item()
accuracy=correct/N_test
accuracy_list.append(accuracy)
loss_list.append(loss.data)
```

Pre-trained models

We are going to re-train the final output layer of our neural network. Then all the layers preceding that layer will be a pre-trained model that has been pre-trained with lots of data; and what usually happens is you get better results on your dataset.



What we are going to do in a pre-trained model in the last hidden layer of a fully connected neural network is to set all the parameters to our model *required grad* equal false. That means that the model will not be differentiable. For our optimizer, the only thing we are going to adjust is to only optimize the parameters that *requires grad* equals to *true,* using any optimizer.

And we have to set the model to *train()* and *eval()*.