CNN:

Convolutional neural networks have gained significant popularity over the last few years as an especially promising form of deep learning. Rooted in image processing, convolutional layers have found their way into virtually all subfields of deep learning, and are very successful for the most part. The fundamental difference between fully connected and convolutional neural networks is the pattern of connections between consecutive layers. In the fully connected case, as the name might suggest, each unit is connected to all of the units in the previous layer. In a convolutional layer of a neural network, on the other hand, each unit is connected to a (typically small) number of nearby units in the previous layer. Furthermore, all units are connected to the previous layer in the same way, with the exact same weights and structure. This leads to an operation known as convolution, giving the architecture its name.

Some of the application areas of Convolution Neural Network are -

- Use mostly in image processing tasks - image classification, segmentation and detection. Ex: face detection.
- Self driving cars - recognizing images, signs and objects.
- Emotion recognition on bases of face expression.

BUILDING BLOCKS:

Dataset used: We have taken the MNIST dataset from torchvision package to develop the CNN model. This dataset has been discussed in chapter 3.

The PyTorch code for this is:

CODE SNIPPET FROM THE NOTEBOOK

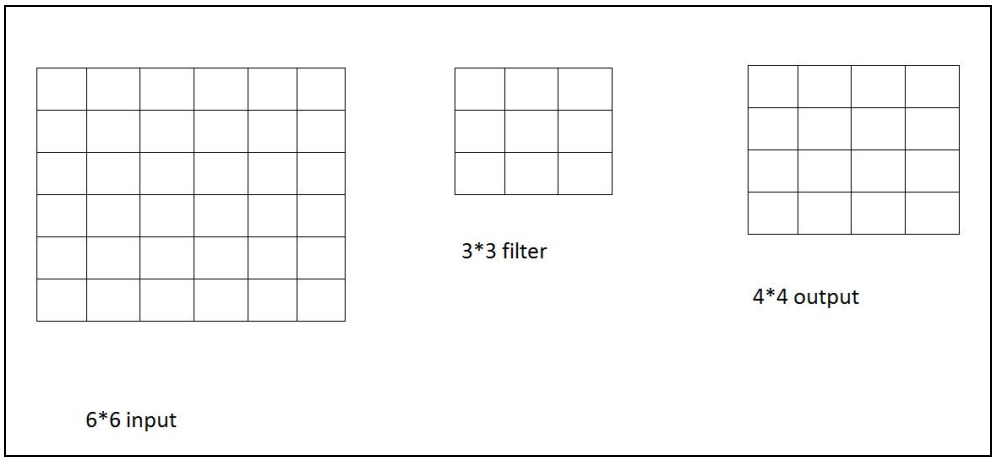
Convolution:
In layman terms it can put as a combined integration of two functions to produce a third to measure how one function modifies the shape of the other. It is represented by the following equations -

$$x_j^l = f\left(\sum_{i\,\varepsilon\,M_j} x_i^{l-1} * k_{ij}^l + b_j^l\right)$$

Where
i = input map,
j and k = output map
b = additive bias
M_j represents a selection of input maps, and the convolution is of the “valid” border handling type.

l = number of layers

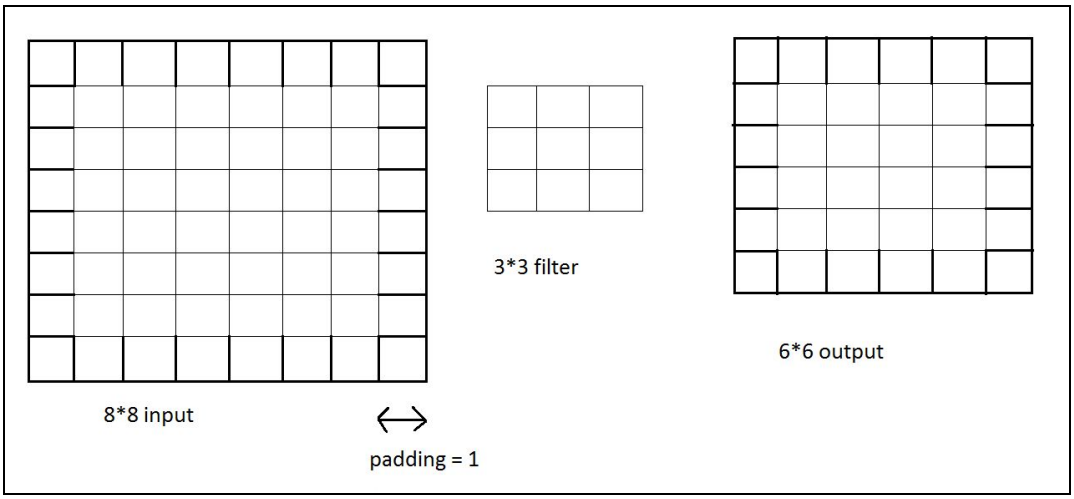
Say, you have an image 6*6 $(n * n)$ and use a filter of 3*3 $(f * f)$, this will result an image of 4*4. This can be calculated as $(n - f + 1) * (n - f + 1)$



Disadvantage:

- The image shrinks with every convolution, hence the edges and features on it also shrink.
- The pixels at the extreme corner of the picture is used only ones, whereas those in the centre overlap a lot of times. So a lot of information at the edge is wasted.

In order to fix this, padding is used. Padding is adding extra border at the side of the edges. Let's say, you have pad the image from 6*6 to 8*8, so going by the above formula (using a 3*3 filter), you get a 6*6 output image. Therefore, the original size of the image is preserved.



The PyTorch code for this is:

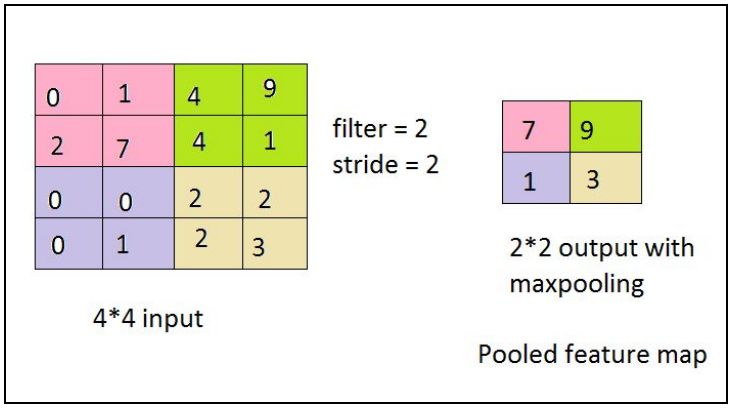
CODE SNIPPET FROM THE NOTEBOOK

Pooling:
Besides Convolution layers, Convolutional neural networks use pooling layers to reduce the size of the representation, speed up computation and the detected features more robust. Based on the size of the filter and stride, it can scan feature map and selects the maximum value. This is Max Pooling. Pooling is also known as downsampling.

Say, you have a 4*4 input and you want to apply Max pooling. The output of this max pooling would be 2*2. This is obtained by dividing the 4*4 input into different regions and the output will be the max value of the corresponding divided region.

This means that we are using a hyperparameters of filter size = 2 and stride = 2. Once these values are fixed, there is no change for gradient descent.

Max in max pooling preserves the maximum value of the features for better results. The general idea of this is to keep the feature but reduce noise.

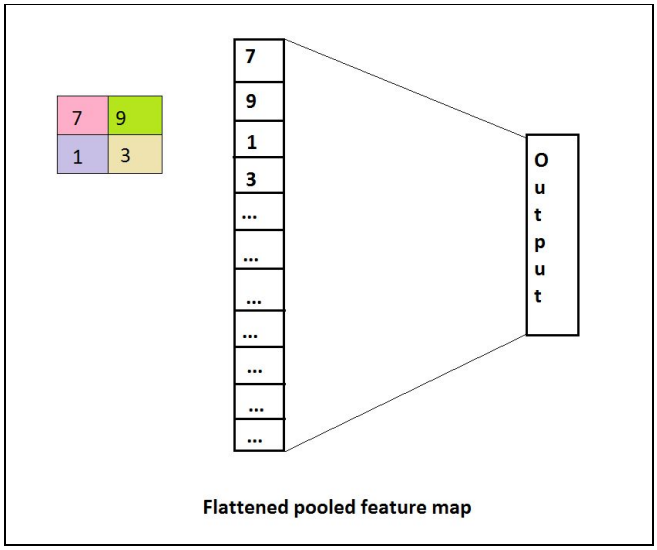


The PyTorch code for this is:

CODE SNIPPET FROM THE NOTEBOOK

Fully Connected Output Layer:
Next step is to flatten the pooled feature map and pass it through a fully connected neural network.

From the example above, we started off with a 6*6 input which is a kind of 2-D array. To connect it to the fully connected neural network, we need to flatten this to a single dimension. Based on the last output of the pooled feature map above, we have a 2*2 matrix, which will be flattened to connect it to the fully connected output layer.



Flattened pooled feature map

Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset. The sole difference between fully connected output layer and convolution layers, is that the neurons in the convolution layer are connected only to a local region in the input, and that many of the neurons in a convolution volume share parameters. However, the neurons in both layers still compute dot products, so their functional form is identical.

Converting between fully connected layer and convolution layer can be done:

- For any convolution layer there is an fully connected layer that implements the same forward function. The weight matrix would be a large matrix that is mostly zero except for at certain blocks because of local connectivity where the weights in many of the blocks are equal due to parameter sharing.
- Conversely, any fully connected layer can be converted to a convolution layer by setting up the filter size to be exactly the size of the input volume.

The PyTorch code for this is:

CODE SNIPPET FROM THE NOTEBOOK

COMPLETE CODE:

To predict handwritten images, using MNIST dataset present in the PyTorch dataloader.

CODE FROM COLAB NOTEBOOK