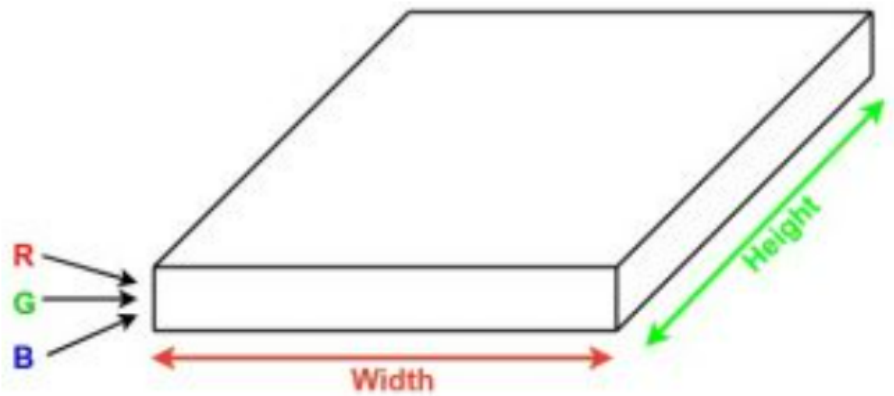
4.6 Relation between input, output and filter size:

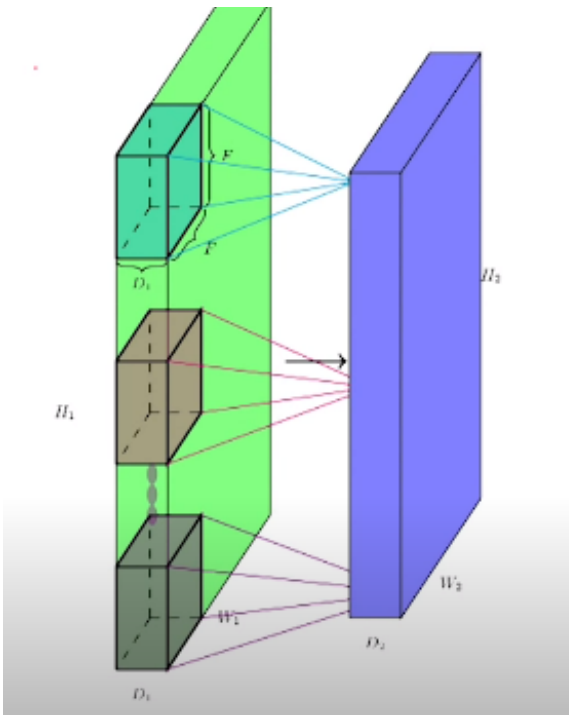
In Convolutional Neural Networks (CNNs), the filter size, input size, and output size are interrelated.

The input to a CNN is typically an image or a feature map.

The size of the input is represented by its height, width, and depth (number of channels). The output of a CNN is also an image or a feature map, with a height, width, and depth that depends on the architecture of the network and the configuration of its layers.

IN most cases, depth is 3(RGB) or 1(Grayscale) images.





Width (W_1), Height (H_1) and Depth (D_1) of the original input.

Stride S.

Number of filters: K.

The spatial extent (F) of each filter (the depth of each filter is same as the depth of each input)

The output is $W_2 \times H_2 \times D_2$ (we will soon see a formula for computing W_2, H_2 and D_2)

So what should our final formula look like,

$$W_2 = \frac{W_1 - F + 2P}{S} + 1$$

$$H_2 = \frac{H_1 - F + 2P}{S} + 1$$

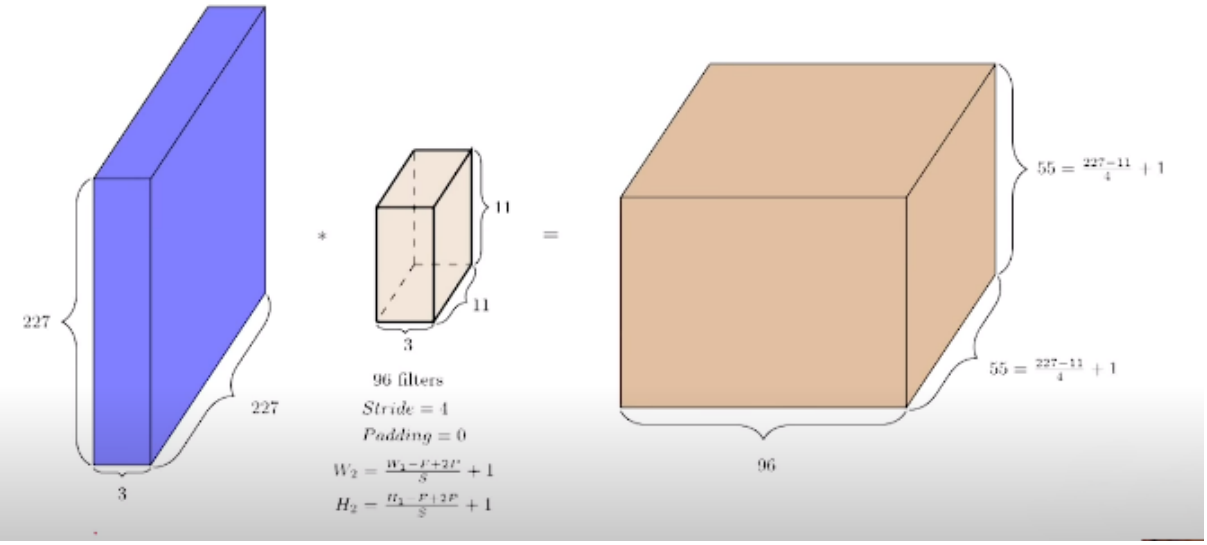
For every filter used, there is a 2D output, here the number filters used is K, hence the depth of output

$D_2 = K.$

In a multilayer convolutional network, the input to the second layer is the output of the first layer, which usually has the output of many different convolutions at each position.

These multi-channel operations are only commutative if each operation has the same number of output channels as input channels. To know how to calculate number of parameters and shape of output in convolution layer, consider the scenario as follows.

Example:



Input

- filter = 1
- kernel_size=(3,3)
- input_shape=(10,10,1)

1. **Number of parameters in the convolution layer:**
   Weights in one filter of size (3, 3) = 3*3=9
   Bias: 1 [One bias will be added to each filter. Since only one filter kernel is used, bias=1]
   In CNNs, biases are additional parameters that are learned alongside the weights. They allow the network to shift the activation function and therefore the output of the layer.
   Total parameters for one filter kernel of size (3, 3) = 9+1 = 10

2. **Calculating the output shape:**
   Output shape:

$$\frac{n+2p-f}{s}+1$$

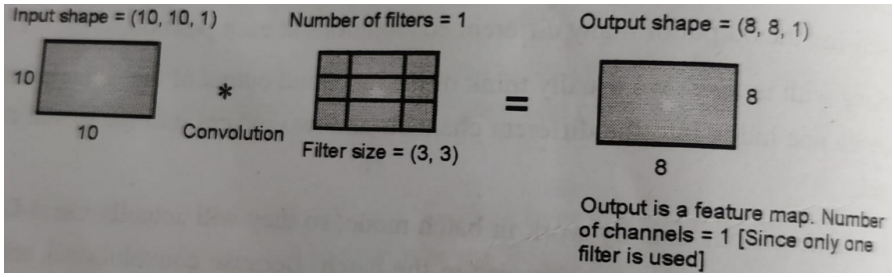
Where s→ stride, p→ padding, n→ input size, f→ filter size.

Stride by default = 1, padding is not mentioned (so, p = 0)

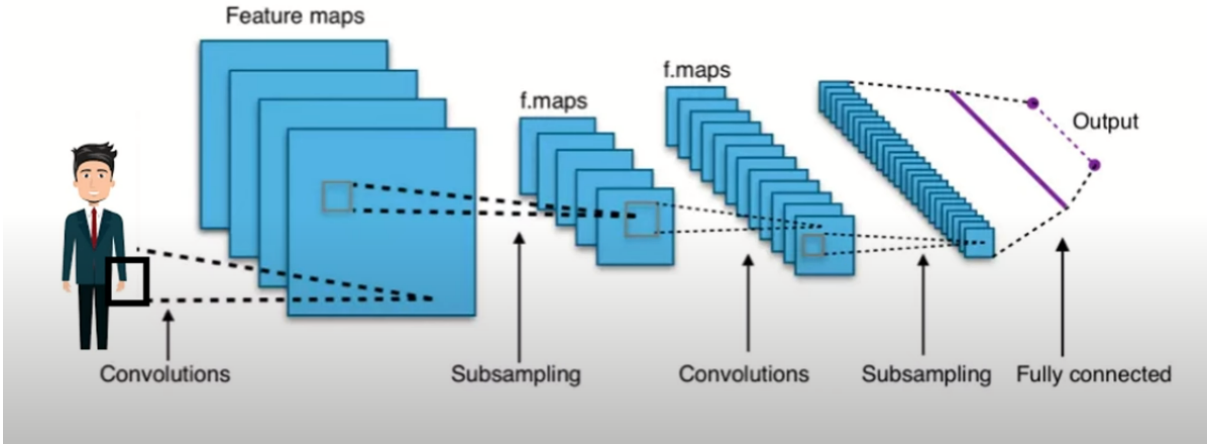
Hence, output shape=n-f+1=10-3+1 = 8

After applying convolution on the input image using a convolution filter, the output will be a feature map. **The number of channels in the feature map depends on the number of filters used**. Here, in this example, only one filter is used. So, the number of channels in the feature ma is 1.

So, Output shape of feature map = (8, 8, 1)



4.7 CNN architecture:



Convolution layer:
A filter passes over the input image, scanning the pixels and creating a feature map.

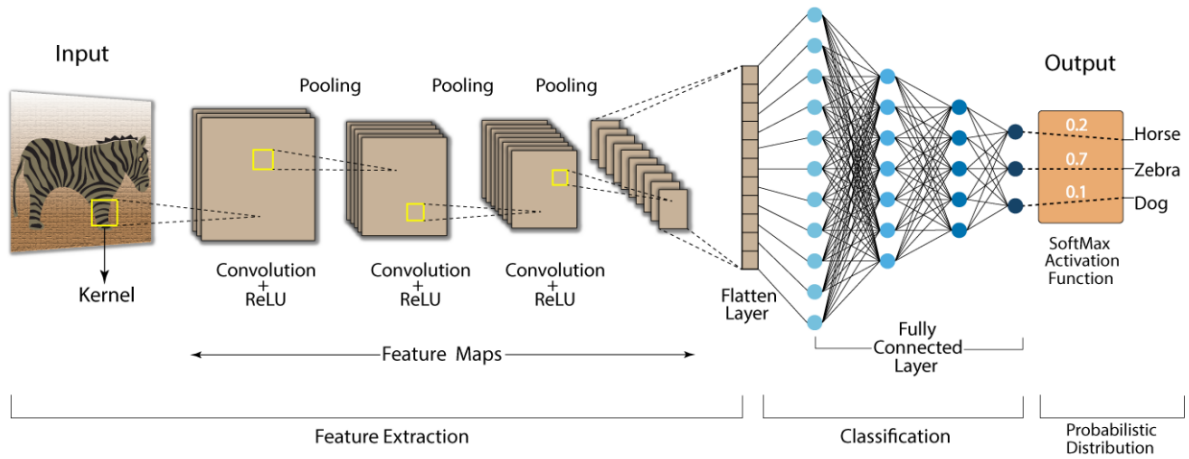
Pooling layer:
Down sampling or subsampling is a method of reducing number of pixels without losing the important information.

With max pooling, we could retain the strong pixels while ignoring the weaker pixels.

Flattening: Output of previous layer is flattened to a single vector and provided as an input to next layer.

The first fully connected layer — takes the inputs from the feature analysis and applies weights to predict the correct label.
Fully connected output layer — This gets us the final probabilities for each label.

Softmax activation is a type of activation function used in convolutional neural networks (CNNs). It is used to normalize the output of a neural network into a probability distribution, which can then be used to make predictions.



Convolution Layer

The convolution layer is a core building block of CNN. It plays an important role in handling the main portion of the network's computational load. At this layer a dot product is performed between two matrices, where one matrix is a set of learnable parameters also known as a kernel, and the other matrix represents the restricted portion of the receptive field.

During the forward pass, the kernel slides across the height and width of the image and generates a two-dimensional image representation of that receptive region. This is known as an activation map that gives the response of the kernel at each spatial position of the image. The sliding size of the kernel is called a stride.

Consider the input image of size WxWx(d). D_{out} is the number of kernels having a spatial size F. stride S and amount of padding is P. The size of output volume is given by the following formula:

W_{out} =

Output volume:

W_{out} x W_{out} x D_{out}

Pooling Layer

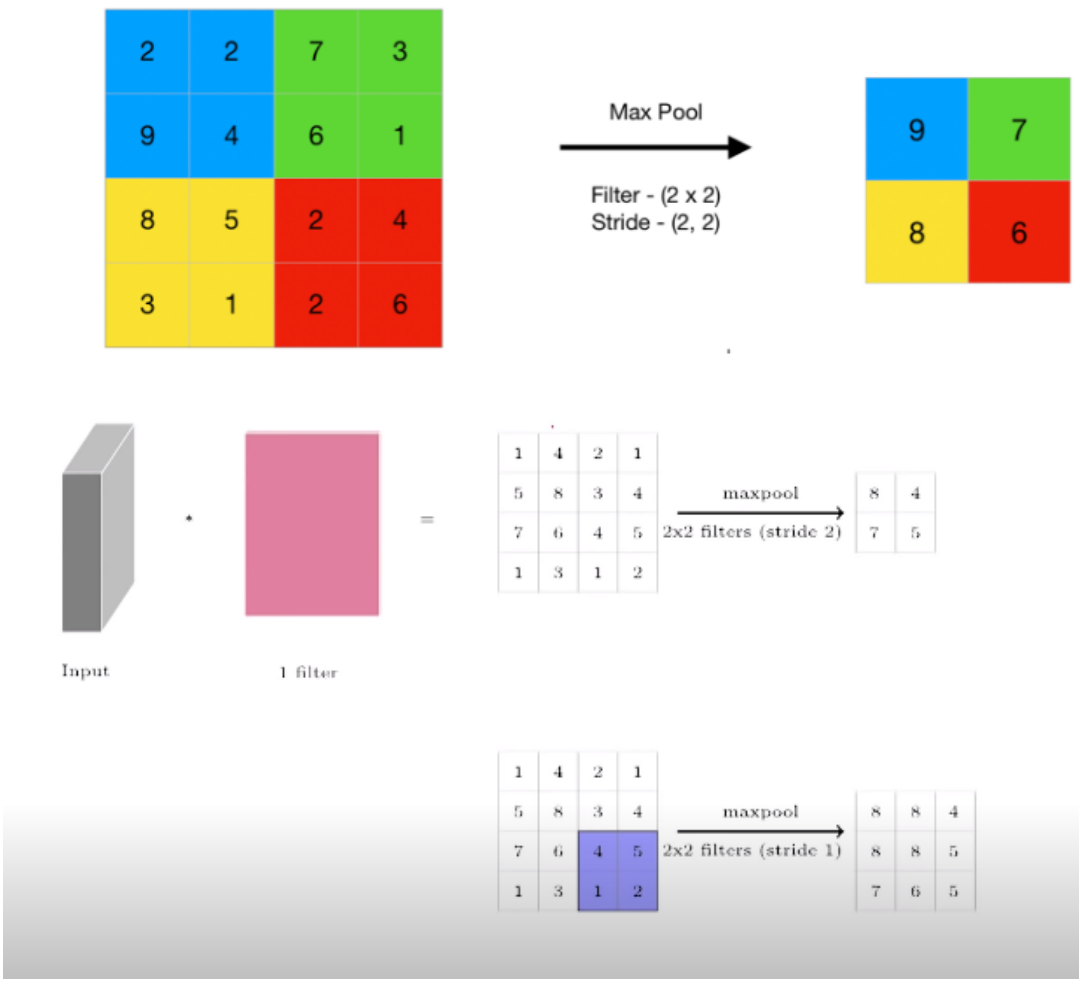
A typical layer of a convolutional network consists of three stages:

1. In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations.
2. In the second stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation function. This stage is sometimes called the detector stage.
3. In the third stage, we use a pooling function to modify the output of the layer further.

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the max pooling operation reports the maximum output within a rectangular neighborhood.

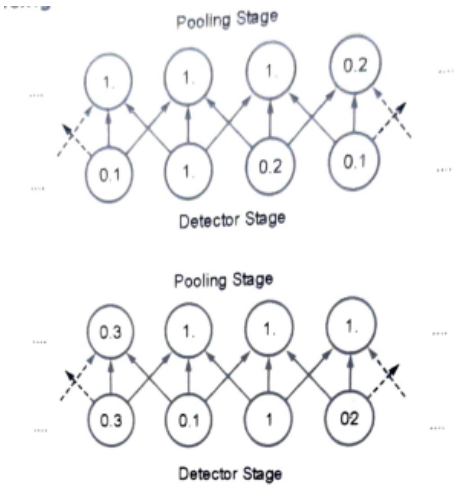
Max Pooling

1. Max pooling is a pooling operation that selects the maximum element from the region of the feature map covered by the filter. Thus, the output after max-pooling layer would be a feature map containing the most prominent features of the previous feature map.



Max pooling introducing invariance

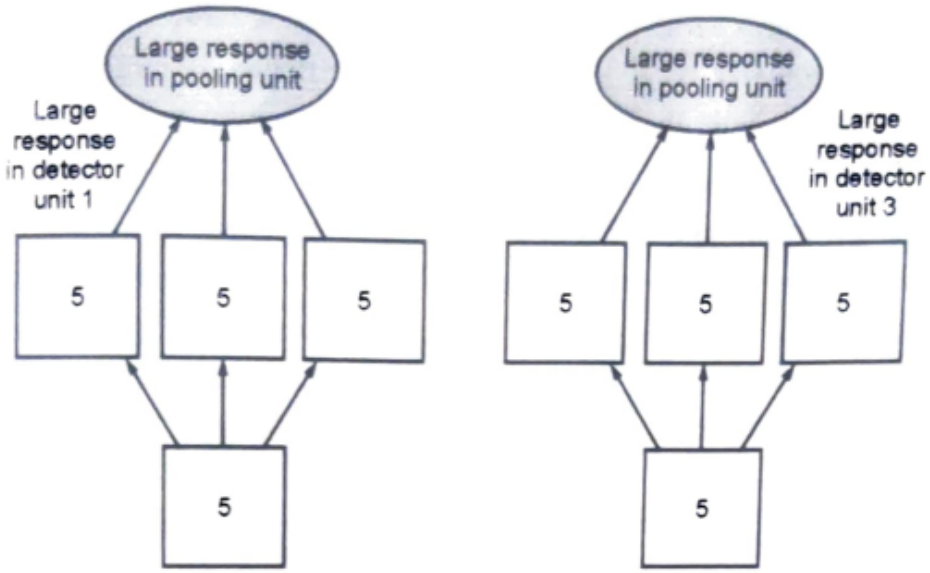
Invariance refers to the property of an object, system, or phenomenon that remains unchanged under certain transformations or operations. In other words, an object or system is said to be invariant under a particular transformation if it appears the same before and after the transformation.



- (Top)A view of the middle of the output of a convolutional layer. The bottom row shows outputs of the nonlinearity. The top row shows the outputs of max pooling, with a stride of one pixel between pooling regions and a pooling region width of three pixels.
- (Bottom)A view of the same network, after the input has been shifted to the right by one pixel. Every value in the bottom row has changed, but only half of the values in the top row have changed, because the max pooling units are only sensitive to the maximum value in the neighborhood, not its exact location.
- Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is. For example, when determining whether an image contains a face, we need not know the location of the eyes with pixel-perfect accuracy, we just need to know that there is an eye on the left side of the face and an eye on the right side of the face. In other contexts, it is more important to preserve the location of a feature.

Example of learned invariances:

Pooling over spatial regions produces invariance to translation, but if we pool over the outputs of separately parametrized convolutions, the features can learn which transformations to become invariant to.



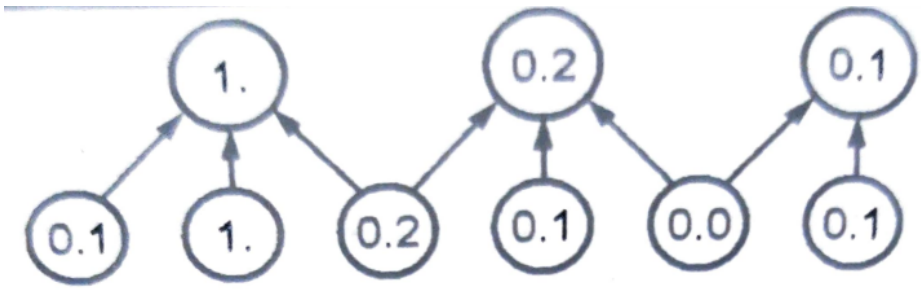
A pooling unit that pools over multiple features that are learned with separate parameters can learn to be invariant to transformations of the input. Here we show how a set of three learned filters and a max pooling unit can learn to become invariant to rotation. All three filters are intended to detect a hand-written 5.

Each filter attempts to match a slightly different orientation of the 5.

When a 5 appears in the input, the corresponding filter will match it and cause a large activation in a detector unit.

The max pooling unit then has a large activation regardless of which detector unit was activated. We show here how the network processes two different inputs, resulting in two different detector units being activated.

Pooling with downsampling:



Because pooling summarizes the responses over a whole neighborhood, it is possible to use fewer pooling units than detector units.

Here we use max-pooling with a pool width of three and a stride between pools of two. This reduces the representation size by a factor of two, which reduces the computational and statistical burden on the next layer.

Note that the rightmost pooling region has a smaller size, but must be included if we do not want to ignore some of the detector units.

This improves the computational efficiency of the network because the next layer has roughly k times fewer inputs to process. When the number of parameters in the next layer is a function of its input size (such as when the next layer is fully connected and based on matrix multiplication) this reduction in the input size can also result in improved statistical efficiency and reduced memory requirements for storing the parameters.

For many tasks, pooling is essential for handling inputs of varying size.

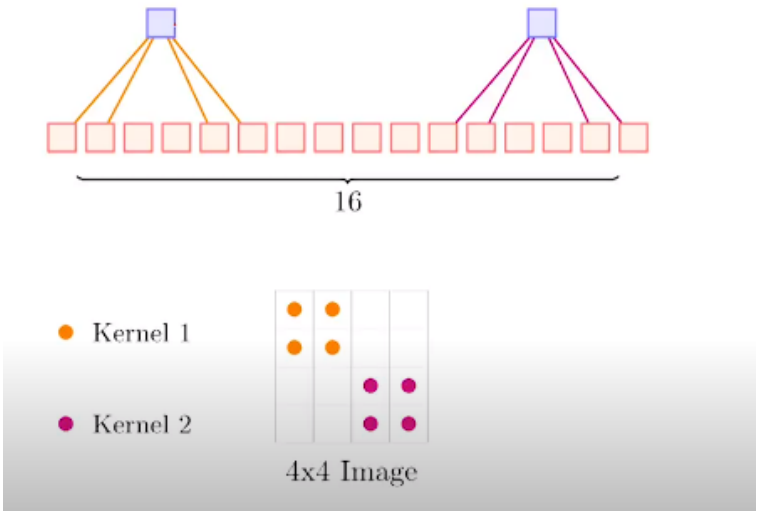
4.8 Weight sharing in CNN

Weight sharing refers to the practice of using the same filter at multiple locations in the image.

It allows the same set of weights to be used across multiple locations in an image.

A typical application of weight sharing is in convolutional neural networks: CNNs work by passing a filter over the image input.

Example, a 4x4 image and a 2x2 filter with a stride size of 2, this would mean that the filter (which has four weights, one per pixel) is applied four times, making for 16 weights total**. A typical application of weight sharing is to share the same weights across all four filters.**



In this context weight sharing has the following effects:

It reduces the number of weights that must be learned (from 16 to 4, in this case), which reduces model training time and cost.

It makes feature search insensitive to feature location in the image.

Weight sharing is for all intents and purposes a form of regularization. And as with other forms of regularization, it can actually increase the performance of the model.

4.9 CNN and fully connected neural network:

A fully connected neural network

- It consists of a series of fully connected layers that connect every neuron in one layer to every neuron in the other layer.
- The major advantage of fully connected networks is that they are "structure agnostic" ie. there are no special assumptions needed to be made about the input.
- While being structure agnostic makes fully connected networks very broadly applicable, such networks do tend to have weaker performance than special-purpose networks tuned to the structure of a problem space.

Convolutional Neural Network:

- A simple CNN is a sequence of layers, and every layer of a CNN transforms one volume of activations to another through a differentiable function.
- Three main types of layers are used to build CNN architecture: Convolutional Layer, Pooling Layer and Fully-Connected Layer.
- Convolutional networks are among the first working deep networks those are trained with the technique of back-propagation. Convolutional networks became successful as they were more computationally efficient than fully connected networks. Hence, it was easy to run multiple experiments with them and tune the hyper parameters.
- Larger networks also seem to be easier to train. With modern hardware, large fully connected networks appear to perform reasonably on many tasks, even when using datasets that were available and activation functions that were popular during the times when fully connected networks were believed not to work well. It may be that the primary barriers to the success of neural networks were psychological (practitioners did not expect neural networks to work, so they did not make a serious effort to use neural networks).
- Convolutional networks provide a way to specialize neural networks to work with data that has a clear grid-structured topology and to scale such models to very large size. This approach has been the most successful on a two-dimensional, image topology.

4.10 Variants of basic convolution function:

Definition of 4-D kernel tensor

- Assume we have a 4-D kernel tensor $\mathbf{K}$ with element $K_{i,j,k,l}$ giving the connection strength between
  - a unit in channel i of the output and
  - a unit in channel j of the input,
  - with an offset of k rows and l columns between output and input units
- Assume our input consists of observed data $\mathbf{V}$ with element $V_{i,j,k}$ giving the value of the input unit
  - within channel i at row j and column k.
- Assume our output consists of $\mathbf{Z}$ with the same format as $\mathbf{V}$.

A. Full convolution:

0 padding 1 stride.

- If $\mathbf{Z}$ is produced by convolving $\mathbf{K}$ across $\mathbf{V}$ without flipping $\mathbf{K}$, then

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n}$$

0 padding s stride

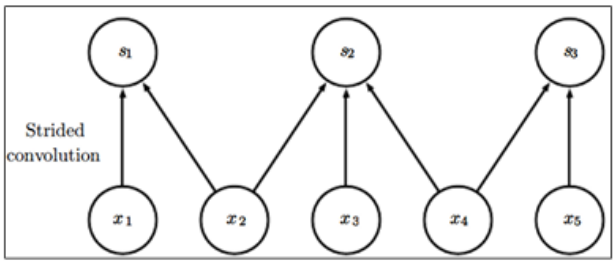
Convolution with a stride: Definition

- We may want to skip over some positions in the kernel to reduce computational cost
  - At the cost of not extracting fine features
- We can think of this as down-sampling the output of the full convolution function
- If we want to sample only every s pixels in each direction of output, then we can define a down-sampled convolution function c such that

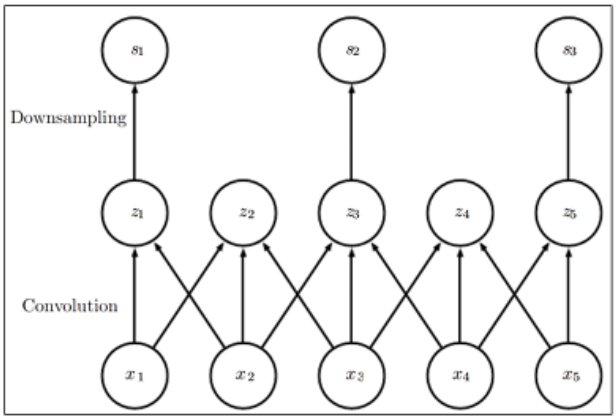
$$Z_{i,j,k} = c(\mathbf{K}, \mathbf{V}, s)_{i,j,k} = \sum_{l,m,n} \left[V_{l,(j-1)\times s+m,(k-1)\times s+n} K_{i,l,m,n} \right]$$

- We refer to s as the stride. It is possible to define a different stride for each direction

Here we use a stride of 2



Convolution with a stride of length two implemented in a single operation



Convolution with a stride greater than one pixel is mathematically equivalent to convolution with a unit stride followed by down-sampling.

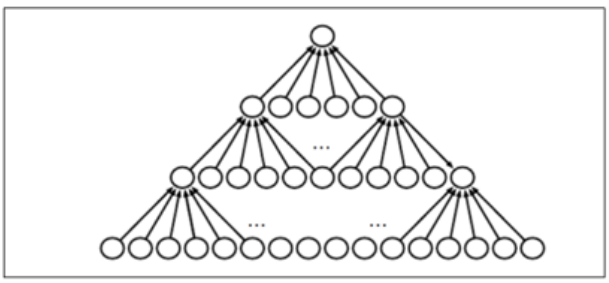
Two-step approach is computationally wasteful, because it discard many values that are discarded

Some 0 paddings and 1 stride

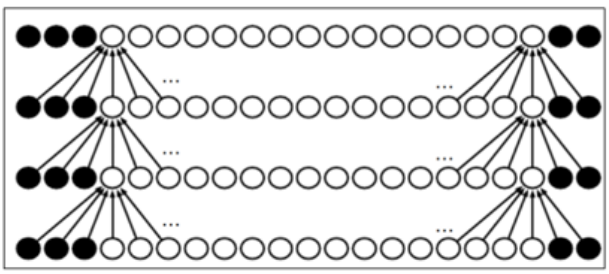
One essential feature of any convolutional network implementation is the ability to implicitly zero- pad the input V in order to make it wider. Without this feature, the width of the representation shrinks by one pixel less than the kernel width at each layer.

Zero padding the input allows us to control the kernel width and the size of the output independently. Without zero padding, we are forced to choose between shrinking the spatial extent of the network rapidly and using small kernels both scenarios that significantly limit the expressive power of the network.

Convolutional net with a kernel of width 6 at every layer
No pooling, so only convolution shrinks network size



We do not use any implicit zero padding
Causes representation to shrink by five pixels at each layer
Starting from an input of 16 pixels we are only able to have 3 convolutional layers and the last layer does not ever move the kernel, so only two layers are convolutional



By adding 5 implicit zeroes to Each layer, we prevent the Representation from shrinking with depth
This allows us to make an arbitrarily deep convolutional network

Three special cases of zero padding

Valid: no 0 padding is used.

One is the extreme case in which no zero-padding is used whatsoever, and the convolution kernel is only allowed to visit positions where the entire kernel is contained entirely within the image. This is also called as valid convolution. In this case, all pixels in the output are a function of the same number of pixels in the input, so the behaviour of an output pixel is somewhat more regular However, the size of the output shrinks at each layer.

Same: keep the size of the output to the size of input.

If the input image has width m and the kernel has width k, the output will be of width m-k+1. The rate of this shrinkage can be dramatic if the kernels used are large. Since the shrinkage is greater than 0, it limits the number of convolutional layers that can be included in the network. As layers are added, the spatial dimension of the network will eventually drop to 1 x 1, at which point additional layers cannot meaningfully be considered convolutional.

Full: Enough zeros are added for every pixels to be visited k (kernel width) times in each direction, resulting width m + k - 1.

Another special case of the zero-padding setting is when just enough zero-padding is added to keep the size of the output equal to the size of the input. This is called as same convolution. In this case, the network can contain as many convolutional layers as the available hardware can support, since the operation of convolution does not modify the architectural possibilities available to the next layer.

However, the input pixels near the border influence fewer output pixels than the input pixels near the center. This can make the border pixels somewhat underrepresented in the model.

Difficult to learn a single kernel that performs well at all positions in the convolutional feature map. Usually the optimal amount of zero padding lies between same and valid convolution.

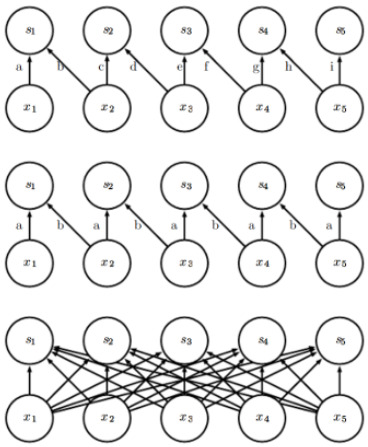
B. Unshared convolution

- In some cases, we do not actually want to use convolution, but rather locally connected layers
  - adjacency matrix in the graph of our MLP is the same, but every connection has its own weight, specified by a 6-D tensor $\mathbf{W}$.
  - The indices into $\mathbf{W}$ are respectively:
  - i, the output channel,
  - j, the output row,
  - k, the output column,
  - l, the input channel,
  - m, the row offset within the input, and
  - n, the column offset within the input.
- The linear part of a locally connected layer is then given by

$$Z_{i,j,k} = \sum_{l,m,n} [V_{l,j+m-1,k+n-1} w_{i,j,k,l,m,n}]$$

- Also called unshared convolution

Local connections, convolution, full connections

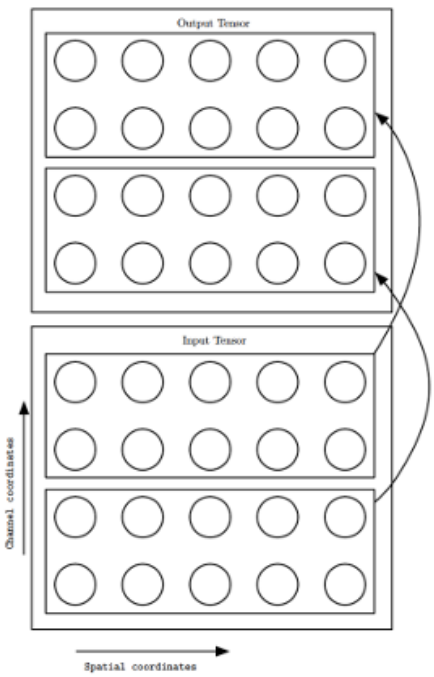


13

*(Top)*A locally connected layer with a patch size of two pixels. Each edge is labeled with a unique letter to show that each edge is associated with its own weight parameter. *(Center)*A convolutional layer with a kernel width of two pixels. This model has exactly the same connectivity as the locally connected layer. The difference lies not in which units interact with each other, but in how the parameters are shared. The locally connected layer has no parameter sharing. The convolutional layer uses the same two weights repeatedly across the entire input, as indicated by the repetition of the letters labeling each edge. *(Bottom)*A fully connected layer resembles a locally connected layer in the sense that each edge has its own parameter (there are too many to label explicitly with letters in this diagram). It does not, however, have the restricted connectivity of the locally connected layer.

It can be also useful to make versions of convolution or local connected layers in which the connectivity is further restricted, eg: constrain each output channel i to be a function of only a subset of the input channel.

Network with further restricted connectivity



A convolutional network with the first two output channels connected to only the first two input channels, and the second two output channels connected to only the second two input channels.

Advantages:

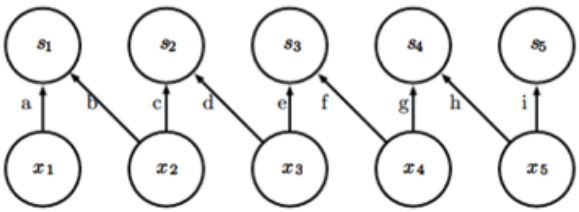
* reduce memory consumption

* increase statistical efficiency

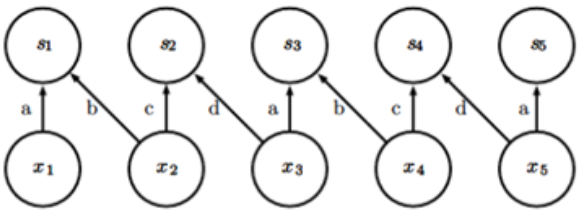
* reduce computation for both forward and backward propagation

C. Tiled Convolution

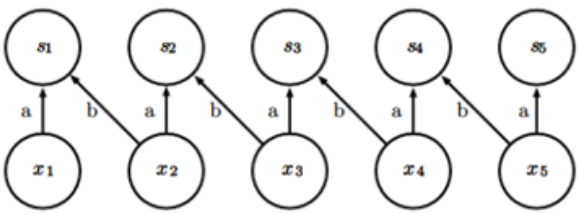
- Compromise between a convolutional layer and a locally connected layer.
  - Rather than learning a separate set of weights at *every* spatial location, we learn a set of kernels that we rotate through as we move through space.
- This means that immediately neighboring locations will have different filters, like in a locally connected layer,
  - but the memory requirements for storing the parameters will increase only by a factor of the size of this set of kernels
  - rather than the size of the entire output feature map.



A locally connected layer
Has no sharing at all
Each connection has its own weight



Tiled convolution
Has a set of different kernels
With $t=2$



Traditional convolution
Equivalent to tiled convolution
with $t=1$
There is only one kernel and it is
applied everywhere

Defining Tiled Convolution Algebraically

- Let k be a 6-D tensor, where two of the dimensions correspond to different locations in the output map.
- Rather than having a separate index for each location in the output map, output locations cycle through a set of t different choices of kernel stack in each direction.
- If t is equal to the output width, this is the same as a locally connected layer

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n,j\%t+1,k\%t+1}$$

- where $\%$ is the modulo operation, with $t\%t = 0$, $(t+1)\%t = 1$, etc

Back Propagation in Convolution layer:

Multiplication by the transpose of the matrix defined by convolution is one operation needed to back-propagate error derivatives through a convolutional layer. It is needed to train convolutional networks that have more than one hidden layer. This same operation is also needed to reconstruct the visible units from the hidden units.

Reconstructing the visible units is an operation commonly used in the models such as autoencoders, RBMs, and sparse coding. Transpose convolution is necessary to construct convolutional versions of those models

The size of the output that the transpose operation should return depends on the zero padding policy and stride of the forward propagation operation, as well as the size of the forward propagation's output map. In some cases, multiple sizes of input to forward propagation can result in the same size of output map, so the transpose operation must be explicitly told what the size of the original input was.

These three operations convolutions, backprop from output to weights, and backprop from output to inputs are sufficient to compute all of the gradients needed to train any depth of feedforward convolutional network, as well as to train convolutional networks with reconstruction functions based on the transpose of convolution.

The backpropagation algorithm is used to train convolutional neural networks (CNNs) by updating the weights of the network in order to minimize the loss function.

Suppose we want to minimize some loss function J (V, K). During forward propagation, we will need to use c itself to output Z, which is then propagated through the rest of the network and used to compute the cost function J. During back-propagation, we will receive a tensor G such

$$G_{i,j,k} = \frac{\partial}{\partial Z_{i,j,k}} J(V, K)$$

To train the network, we need to compute the derivatives with respect to the weights in the kernel. To do so, we can use a function

$$g(G,V,s)_{i,j,k,l} = \frac{\partial}{\partial K_{i,j,k,l}} J(V, K) = \sum_{m,n} G_{i,m,n} V_{j,(m-1)\times s+k,(n-1)\times s+l}$$

If this layer is not the bottom layer of the network, we will need to compute the gradient with respect to V in order to back-propagate the error farther down.

Adding Biases:

- For local conncted layers: give each unit its one bias
- For tiled conv layers: share the biases with the same tiling pattern as the kernels
- For conv layers: have one bias per channel of the output and share it across all locations within each convolution map. If the input is fixed size, it is also possible to learn a separate bias at each location of the output map.