Convolutional Networks

Table of Contents

[Convolution Layers 2](#_Toc138881744)

[Stacking Convolutions 6](#_Toc138881745)

[Spatial Dimensions 6](#_Toc138881746)

[Receptive Fields 7](#_Toc138881747)

[Strided Convolutions 9](#_Toc138881748)

[1x1 Convolutions 10](#_Toc138881749)

[1D and 3D convolutions 11](#_Toc138881750)

[Pooling Layers 12](#_Toc138881751)

[LeNet-5 13](#_Toc138881752)

[Normalization 14](#_Toc138881753)

[Scaling and Shifting 15](#_Toc138881754)

[Inference 16](#_Toc138881755)

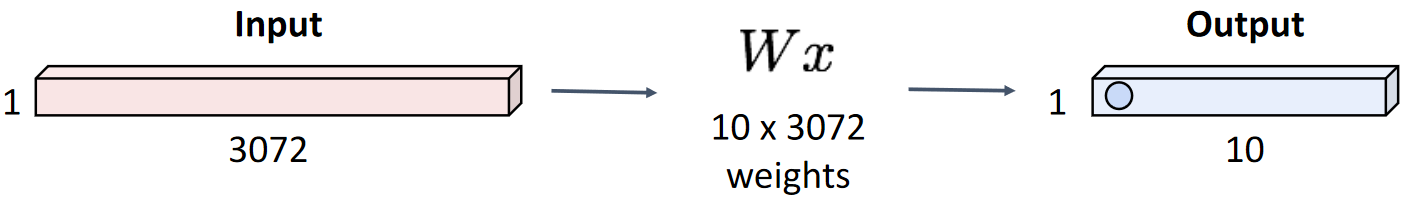
[ConvNets 16](#_Toc138881756)

So far, we have only looked at linear networks and neural networks. When dealing with images, these networks require that we **flatten** the images into a single column to be able to pass them as input. Unfortunately, doing this **destroys the spatial information** of the images. **Convolutional networks** allow us to avoid doing this and keep the 2D form of the images.

Convolutional networks have three major parts, convolution layers, pooling layers and normalization. We will be studying each of these parts in turn.

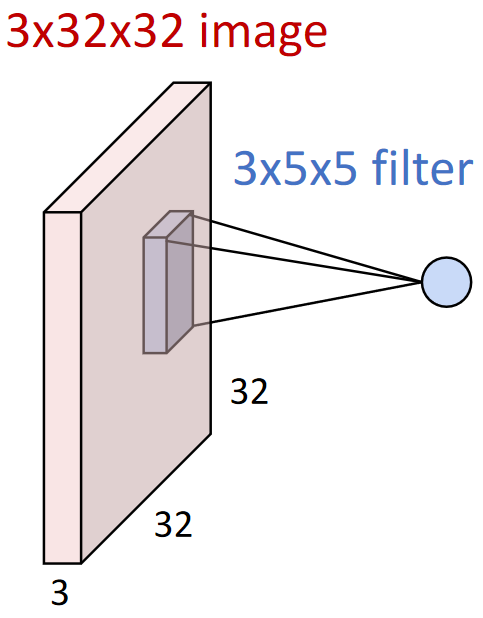
## Convolution Layers

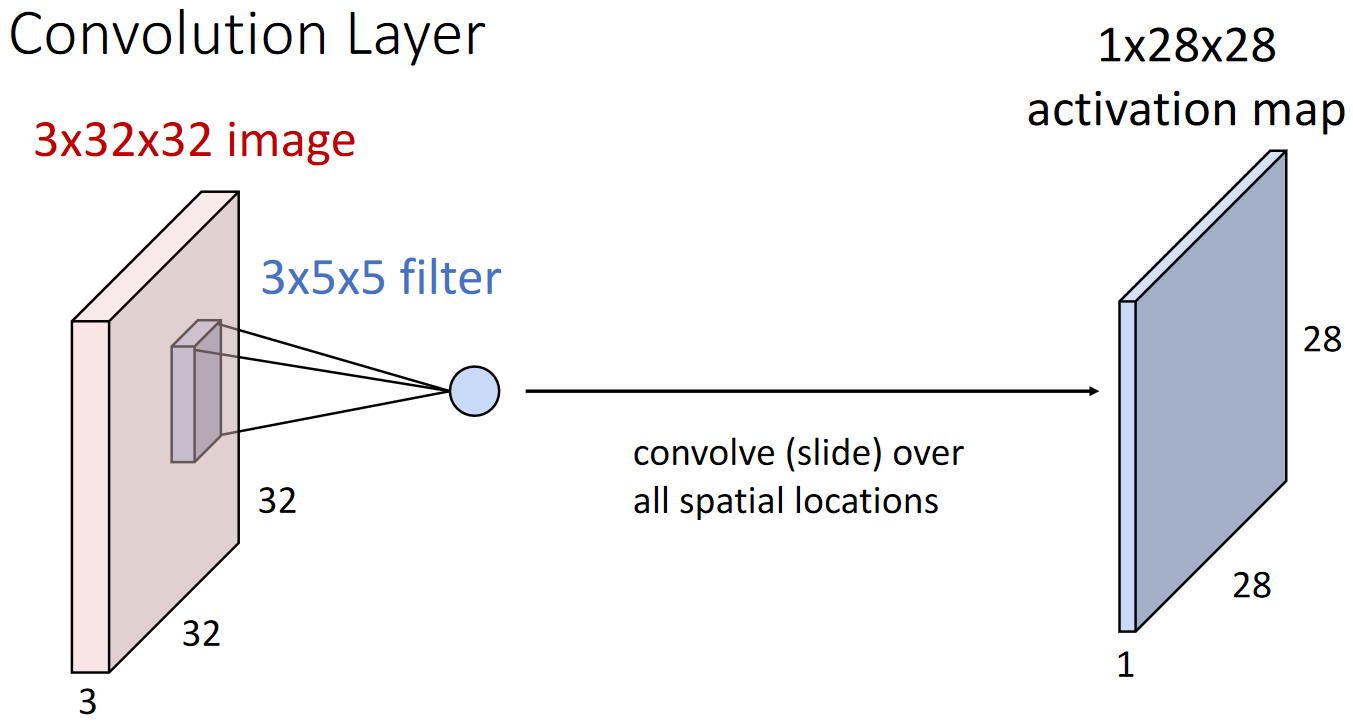
Suppose we have a 32x32x3 RGB image. Fully connected layers require that this image be flattened to 3072x1.



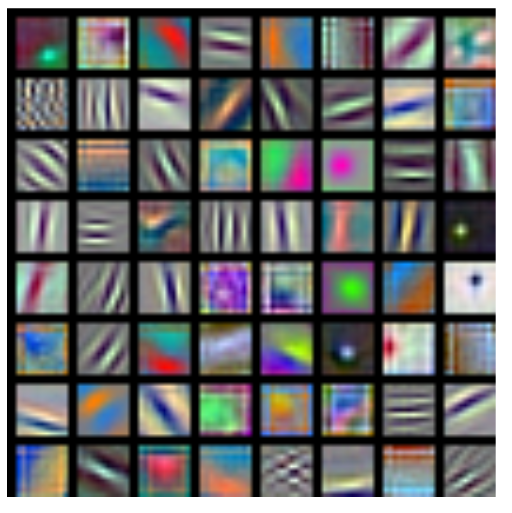
A single row of weights is multiplied by the input vector to give a single output unit.

A **convolution layer** on the other hand, preserves the original shape of the image. Instead of weights, it uses a **filter** that has the **same depth** as the image, but a **different width and height**. The filter is **convolved** (slid) over the image covering all the spatial locations. At each position, we get a single output. Thus, a single filter can be compared to a single layer of neurons in a neural network. The size of the filter is comparable to the number of neurons in the layer.





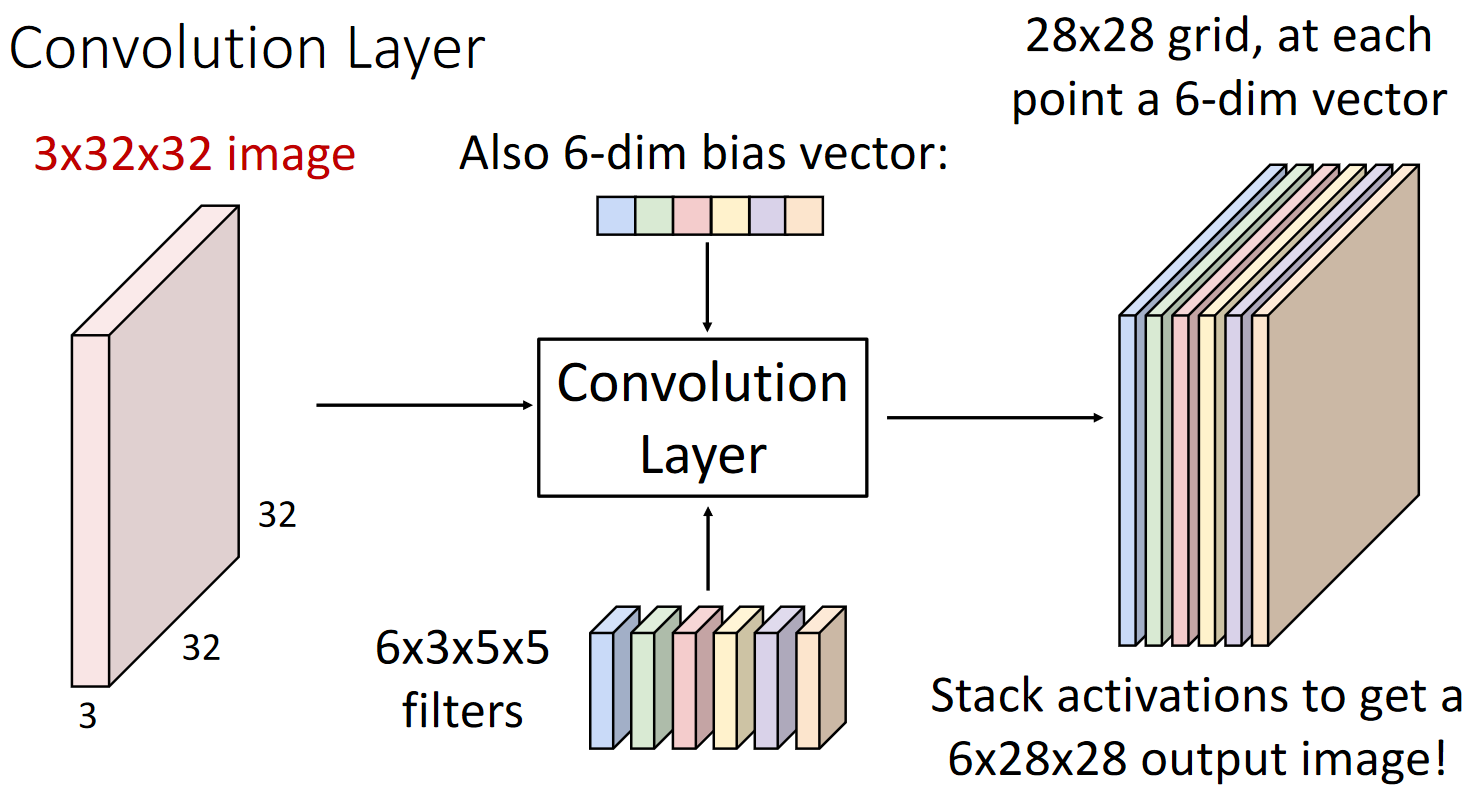
The output of a single filter is called an **activation map**. Typically, convolution layers have multiple filters, which results in multiple activation maps. Each filter has different weights and works to capture different information from the input. For example, one filter could be capturing horizontal edges while another captures vertical edges. Thus, the filters are capturing **local image templates** as opposed to global ones like linear layers and neural networks. The more layers we use, the more variations of information we can capture.



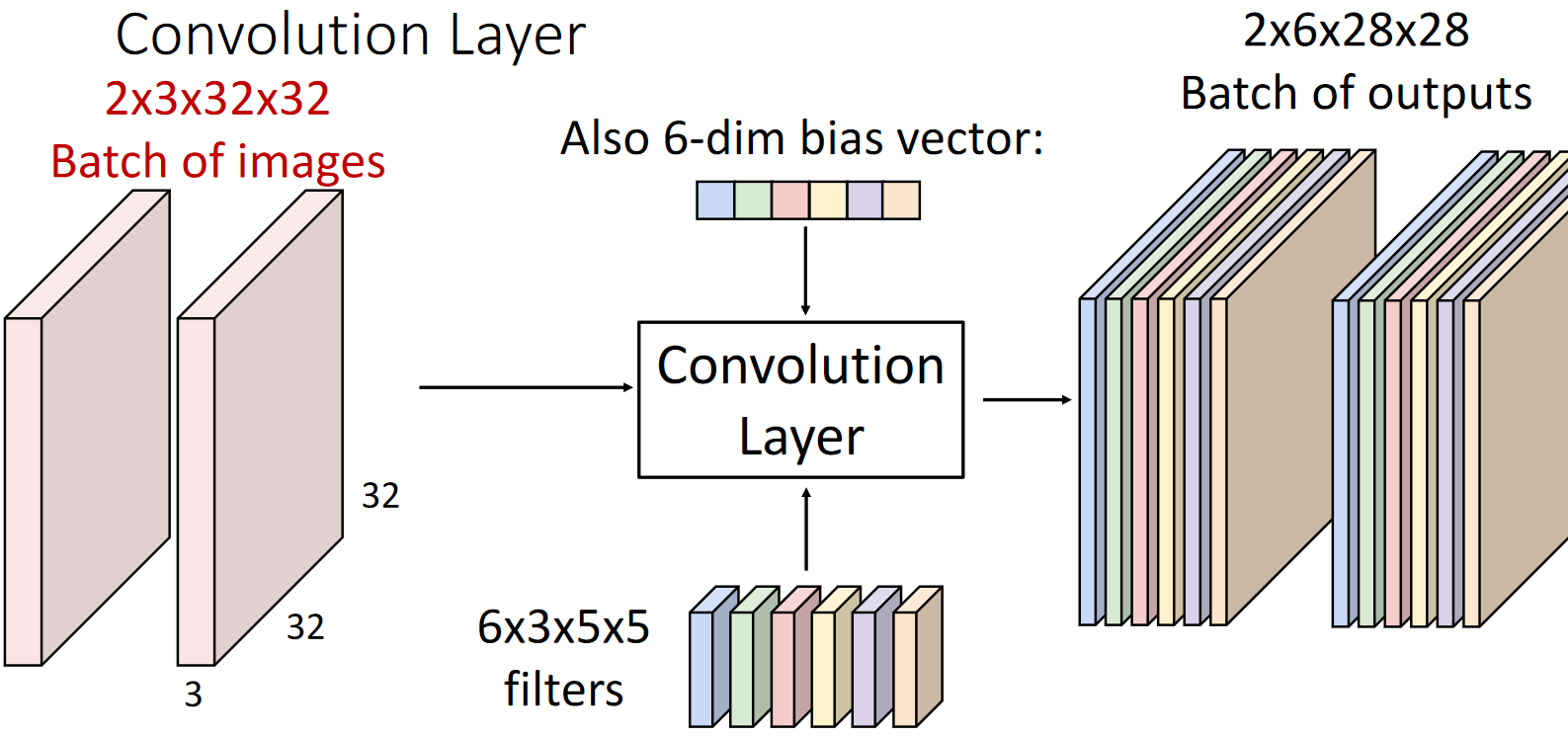
The combination of all the filters being used at once is called a **convolution layer**. The activation maps of all the filters are finally stacked to create an output ‘image’.



In additional to this, each filter has a corresponding **bias vector** added to it.



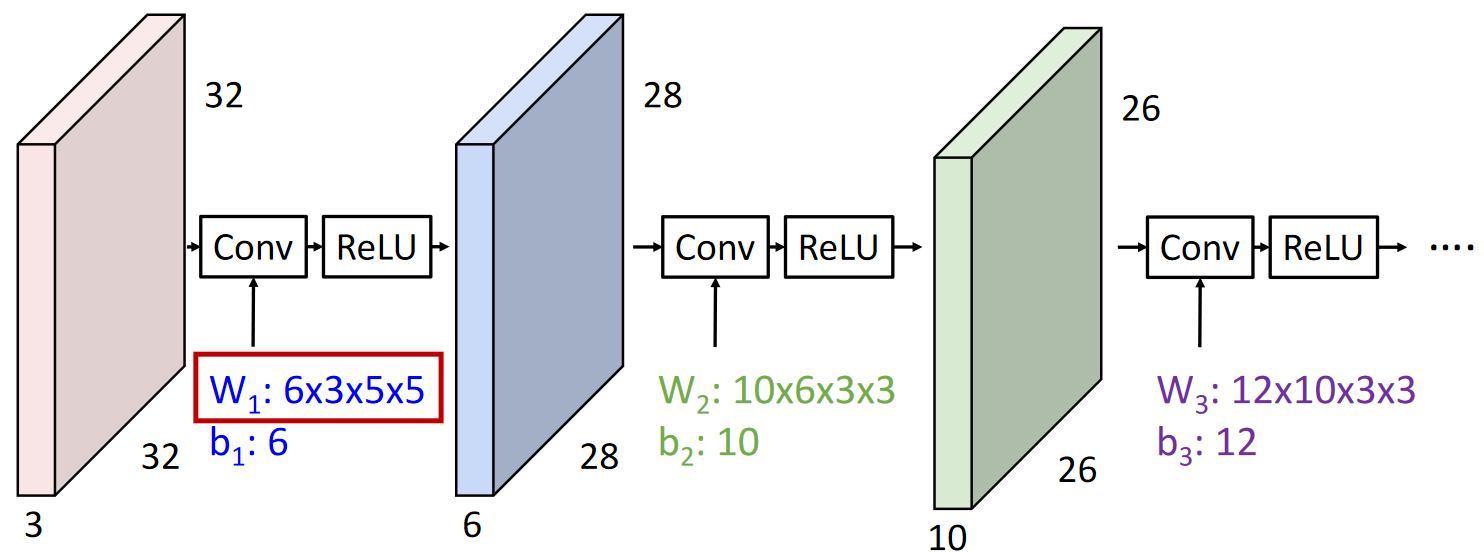
We can also have a batch of inputs instead of a single input. This will result in a batch of outputs.



It is important to note the **dimensions** of each of the components. The depth of each filter is equal to the depth of the input. The width and height can be different but are usually 3x3, 5x5 or 7x7. The number of filters in the convolution layer is equal to the depth of the output. The number of batches in the input and the output are equal. Formally, for an input and a convolution layer, we get an .

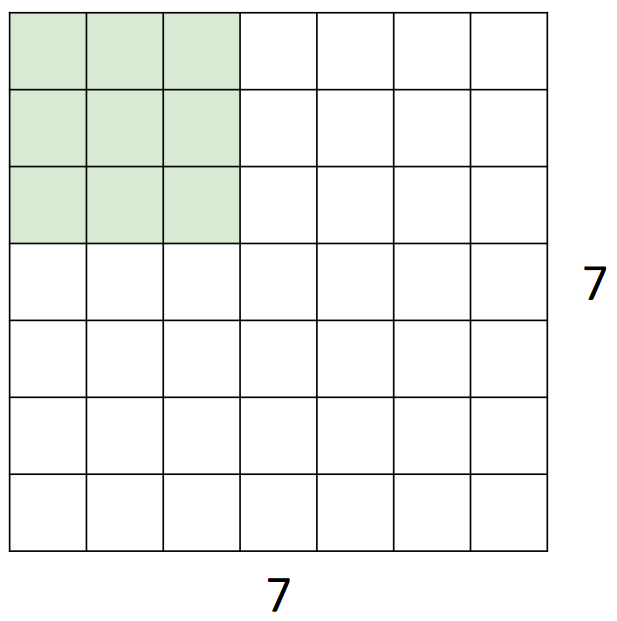
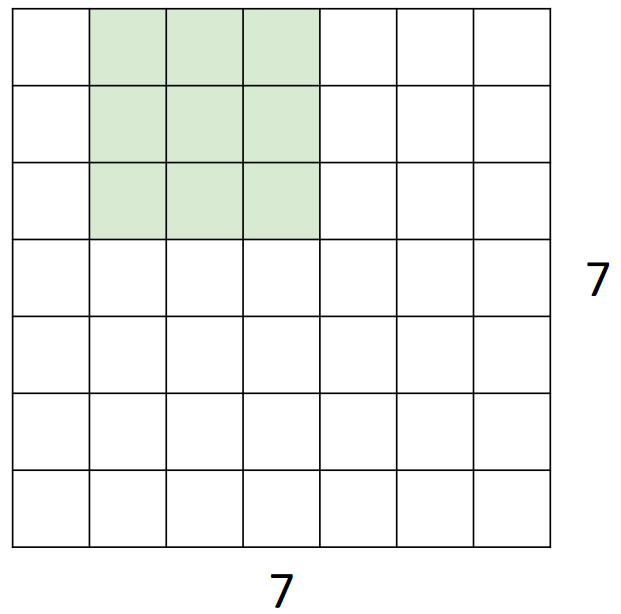
### Stacking Convolutions

If we stack one convolution layer after another, the result is the same as a single convolution layer. This is similar to the issue of stacking multiple linear layers together in a neural network. Doing this will prevent the model from learning complex decision patterns. To resolve this, as before, we use **activation layers**.



### Spatial Dimensions

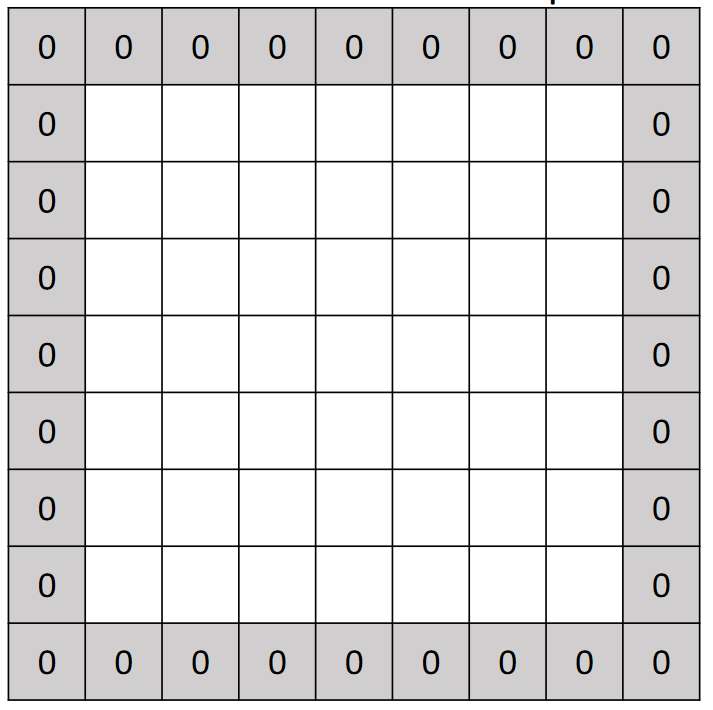
Suppose we have a 7x7 image and a 3x3 filter. The convolution operation for this case looks like this:

Since each convolution operation gives a single output unit, the entire process will result in a 5x5 image. The problem is, the image has **shrunk**. If we keep doing this over multiple layers, we will have nothing left to work with.

In general, for a sized input and a sized filter, we get a sized output.

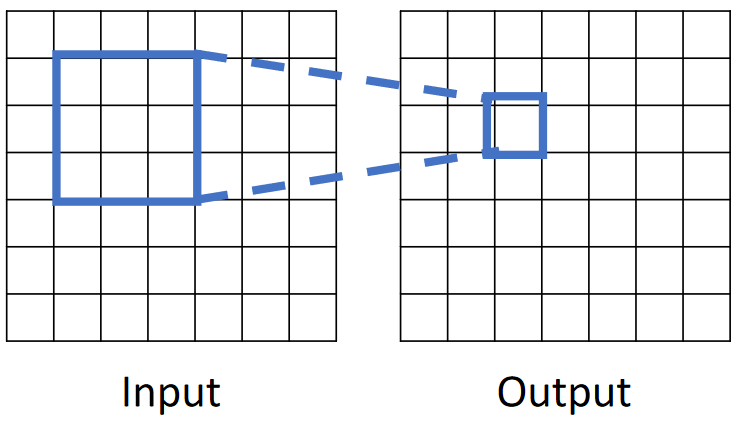
The solution to avoiding the shrinkage is to **pad the image**. To make the output the same size as the input, we need to add a padding of .



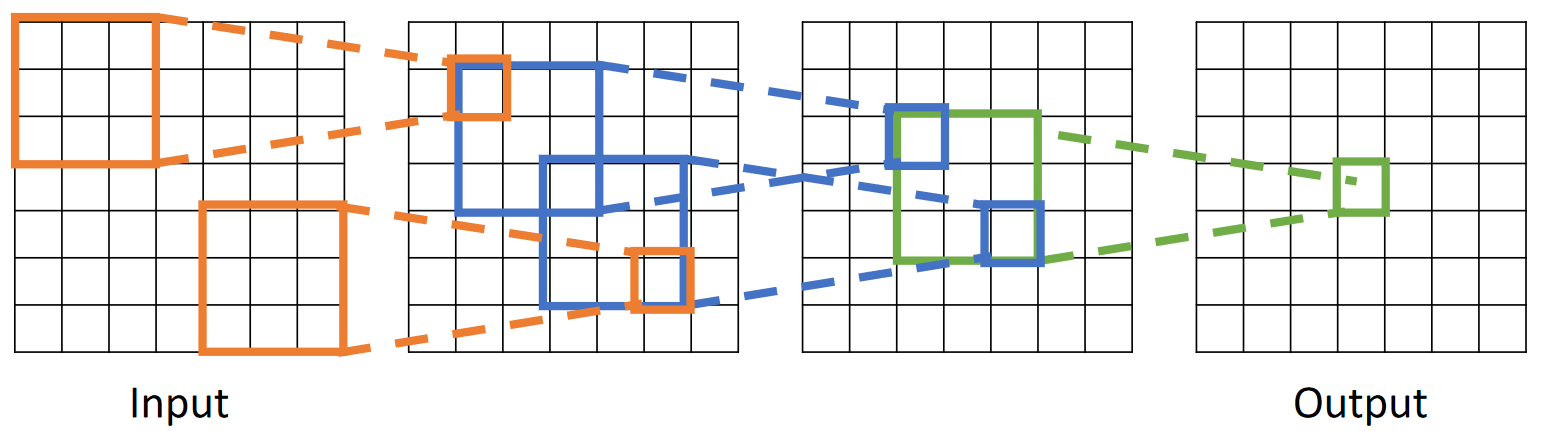
With a padding of , the output size becomes .

### Receptive Fields

For a filter size of , a single output unit depends on a units of the input. This size is called the **receptive field**.



If we have multiple convolution layers, each layer adds to the receptive field size. With layers, the total receptive field size becomes .

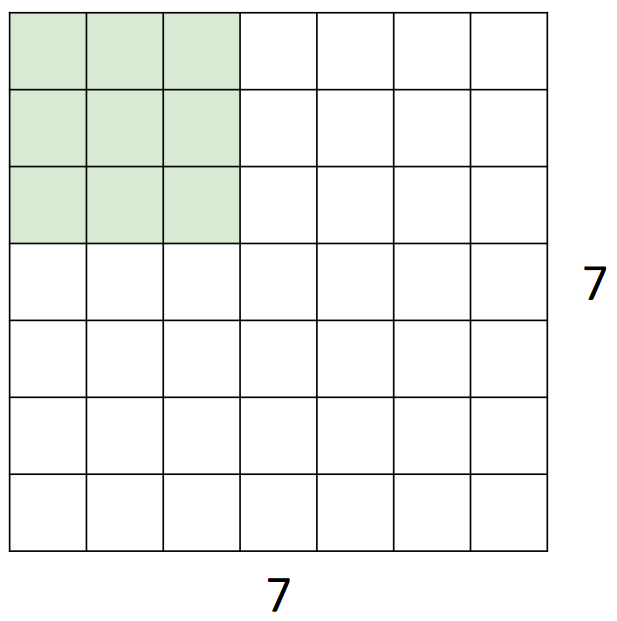
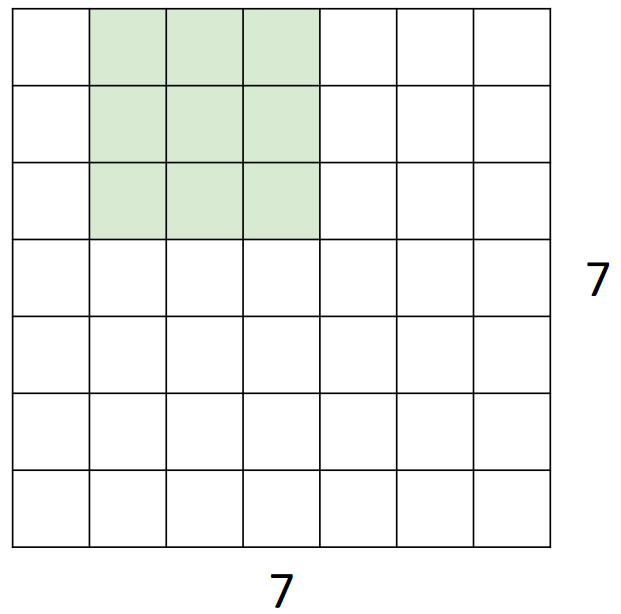


Clearly, the receptive field determines how much of the input information is available at the final output layer. The problem is, for a single unit of the input to be able to ‘see’ the entire input, we need to have a large number of convolution layers in between. We can increase the receptive field size in each layer by using a larger filter. Using a 9x9 filter will create an equivalent effect to using two 3x3 filters for example. However, this is computationally expensive and results in aggressive downsampling.

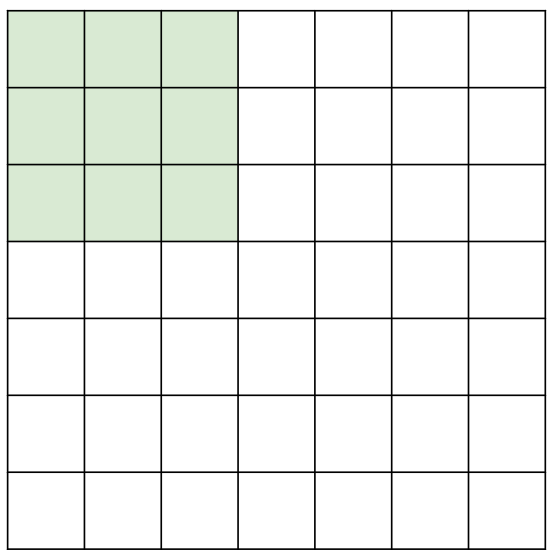
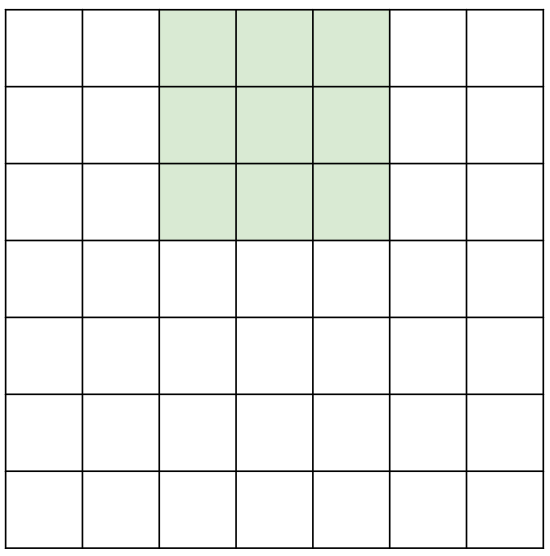
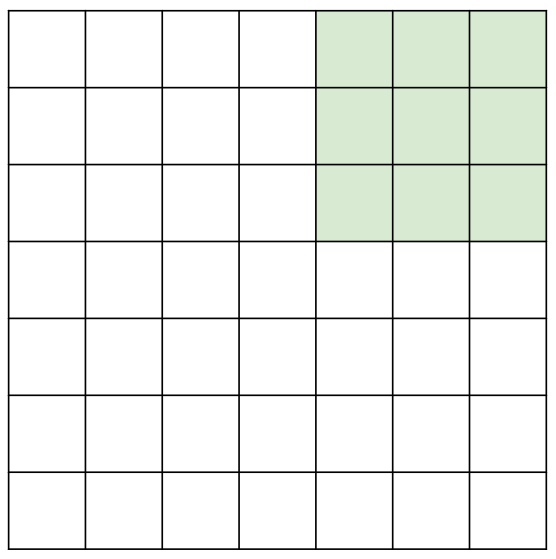
To avoid issues like this and still ensure that we can ‘see’ the entire image without having an absurd number of convolution layers, we can use **strided convolution**.

### Strided Convolutions

Normally, the convolution operation shifts the filter by a single column or a single row.

Strided convolution allows us to vary the number of rows and columns to shift. This is called the **stride**. For a stride of 2, the convolution operation looks like this:

The output size for strided convolution is .

Example

Suppose we have a 3x32x32 input and 10 5x5 filters with a stride of 1 and padding of 2. Thus, the output is of size 10x32x32.

The total number of learnable parameters here is .

There are a total of output units. Each unit is the result of a multiplication, which gives 75 multiplications. Thus, there are a total of multiplications.

### 1x1 Convolutions

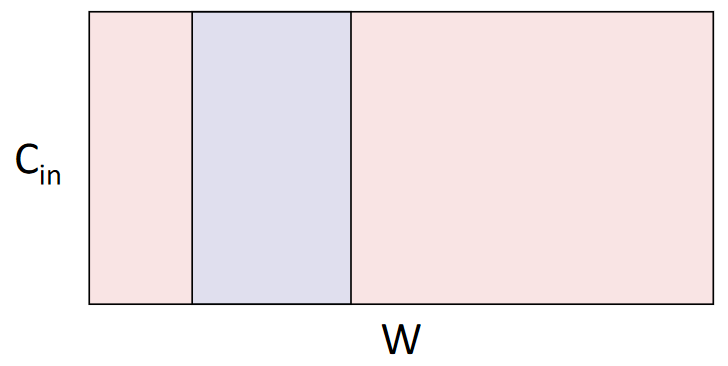
An interesting use case of convolution operations is the **1x1 convolution**. Here, the filter size is 1x1, so the height and width of the input do not change. However, depending on the number of filters we use, we can change the number of channels in the output.



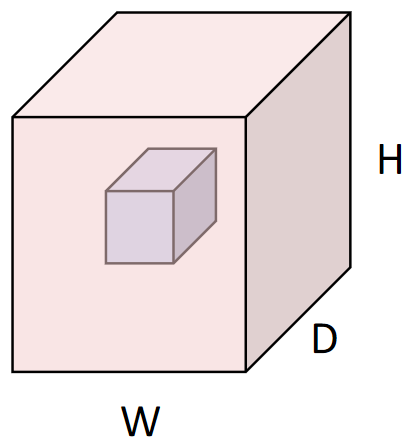
### 1D and 3D convolutions

Convolution operations can also be 1D or 3D.

For a 1D convolution, if the input size is , the filter size can be .

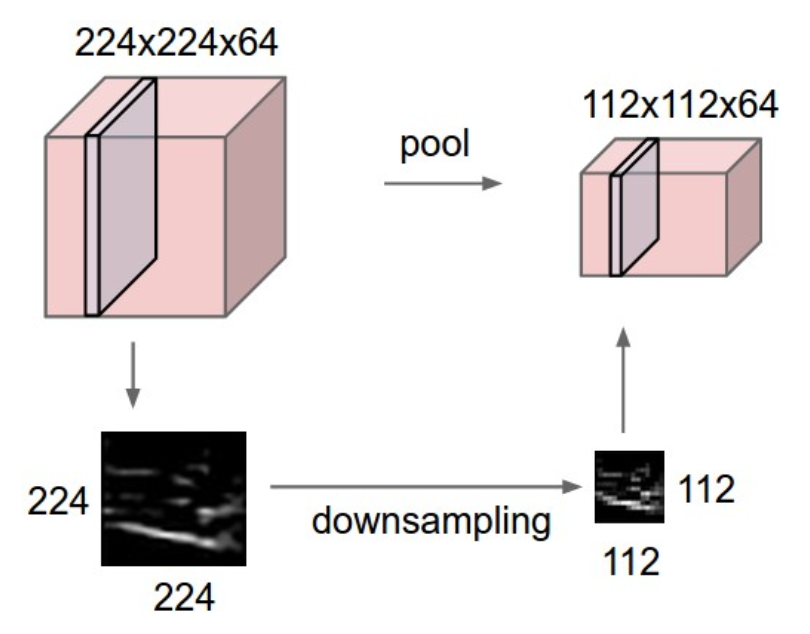


For a 3D convolution, if the input size is , the filter size can be .

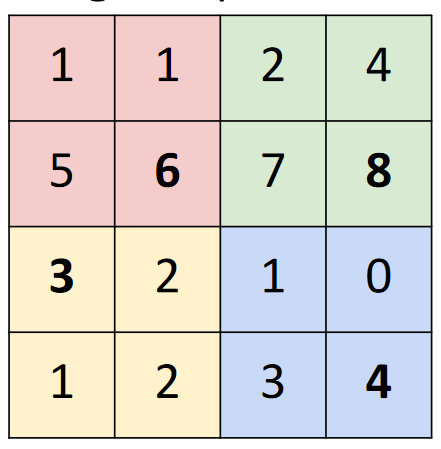


## Pooling Layers

An alternative way to downsample images is to use a **pooling layer**. This can be a max-pooling layer, a min-pooling layer or an average-pooling layer.



A pooling layer can provide some benefits as opposed to striding sometimes. For example, striding requires adding a filter, which has weights that must be learnt. A pooling layer does not, which means less computational expense. Pooling layers are also **invariant to small spatial variances**. For example, the bottom-right region in the image below has a maximum value of 4. Even if that pixel shifts one pixel to the top or to the left, the maximum value for the region remains unchanged.

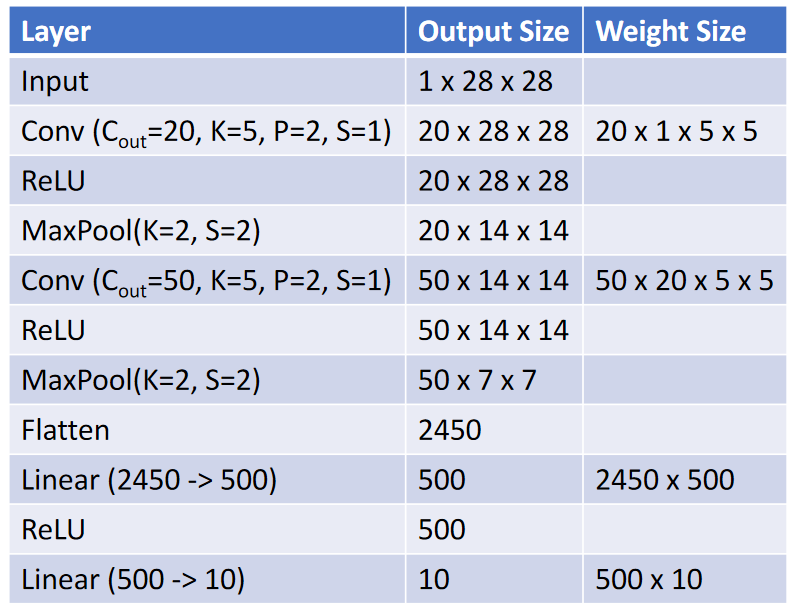


Pooling layers also introduce **non-linearity**, which has been used in the past to justify the lack of activation functions in some architectures.

## LeNet-5

A classical architecture based on CNN is the **LeNet-5** architecture.



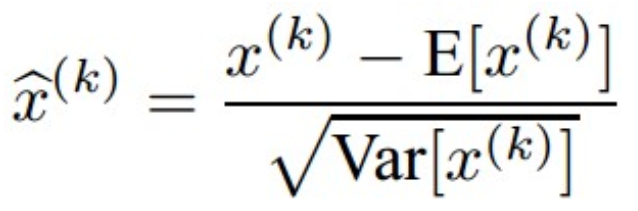


The pattern seen here, where the **spatial dimensions decrease** and the **depth dimension increases** as we go deeper into the network, will be seen repeatedly in later architectures as well. These changes preserve the **total volume**.

## Normalization

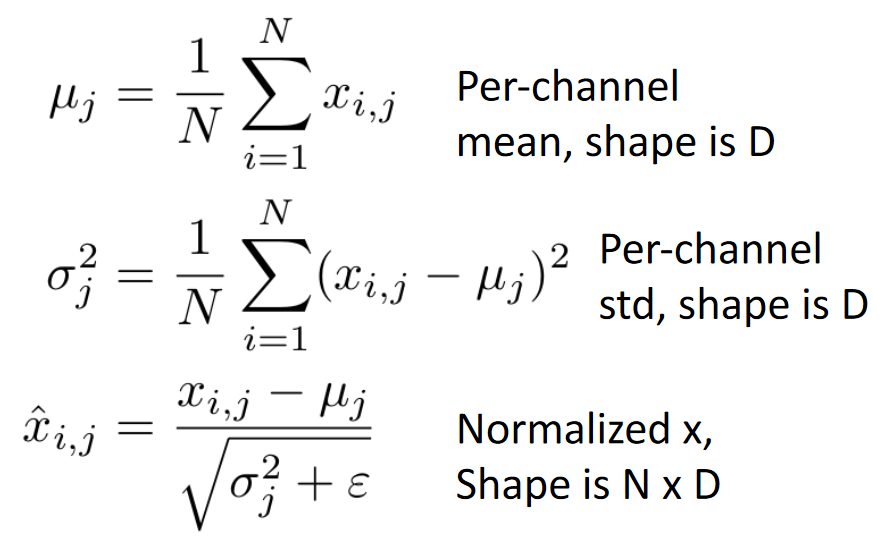
One of the issues that we might run into when working with deep networks is that the outputs of each layer may be on very different scales. Even if we normalize the inputs, each layer changes the distribution which may reintroduce or even worse the issue. Having an un-normalized output makes it harder for the model to learn the behaviour of the data.

The process of normalizing the output from a hidden layer is called **batch normalization**. Normalizing the outputs causes them to have **zero mean** and **unit variance**. This **reduces the internal covariate shift**, which in turn **improves optimization**.



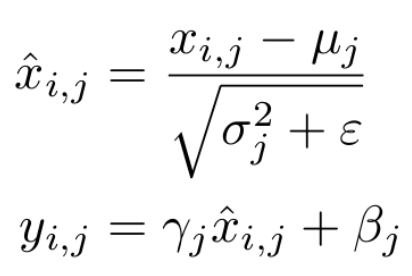
The batch normalization function is **differentiable**, which means we can just add batch normalization as a layer to our network.

The normalization process occurs **per channel** across each dimension of the input. If we consider a 2D input, similar to an image, then each pixel is being normalized based on the values of the pixels at the same position across all the inputs in the batch.



### Scaling and Shifting

One issue with this setup is that a zero mean and unit variance requirement may be too strict. It may be the case that it is better for the model to have a slightly un-normalized input for a specific layer. To allow for this, we can introduce two additional learnable parameters, and , called the **scale** and **shift** parameters respectively.



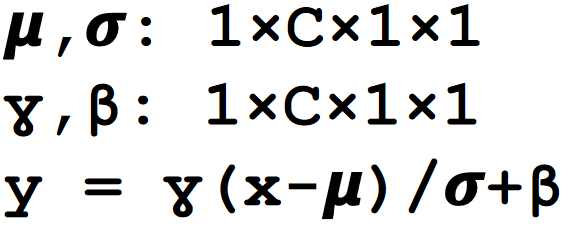
The values of and can be used by the model to offset the effect of batch normalization. Notice that if and , the original input is returned.

### Inference

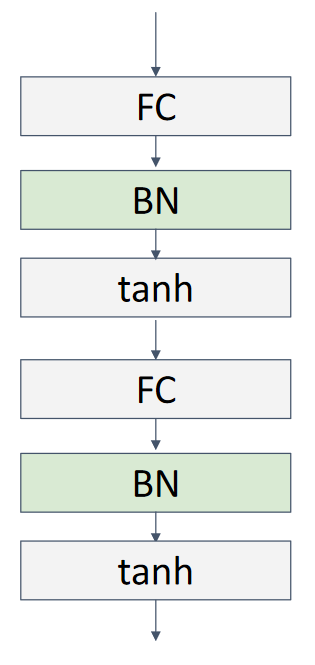
One issue with batch normalization is that the values of and depend on which images are present in the batch. Although this is not an issue during training, since the values are finetuned based on all of the data eventually, it becomes an issue during **inference**. The values should not change during inference. As such, the values of , , and are all fixed to the learned values during inference.

### ConvNets

We discussed that for a dimensional input, with images in the batch, the and values are dimensional, meaning the values are being normalized along the batches. ConvNets have a dimensional input. In this case, the and values are dimensional, meaning the values are being normalized along the batches, heights and widths.



Batch normalization is usually done after **fully connected** and **convolution layers** and before **activation layers**.



The use of batch normalization allows for faster convergence, makes the network more robust to variances in the initial weights, and acts as a regulariser since it introduces noise from other images in the batch. The graph below shows how batch normalization can be used to make the model converge far faster than it normally does.

