

Module 3: Convolutional Networks

Module 3: Convolution neural networks: The Convolution Operation, Motivation, Pooling, Convolution and Pooling as an Infinitely Strong Prior, Variants of the Basic Convolution Function, Structured Outputs, Data Types, Efficient Convolution Algorithms, Convolutional Networks and the History of Deep Learning.

Text book 1 : Chapter 9 (9.1 to 9.8, 9.11)

Convolutional Neural Networks:

Convolutional networks (LeCun, 1989), also known as convolutional neural networks or CNNs, are a specialized kind of neural network for processing data that has a known, grid-like topology. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, and image data, which can be thought of as a 2D grid of pixels. Convolutional networks have been tremendously successful in practical applications. The name “convolutional neural network” indicates that the network employs a mathematical operation called convolution. Convolution is a specialized kind of linear operation. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

○ The Convolutional Operation:

- Convolution Operation: As convolution is a mathematical operation on two functions that produces a third function that expresses how the shape of one function is modified by another.
- Similarly, CNN is more like a convolution operation where we take different measurements and rely more on the near ones for better results.
- So here we take revised measurements which is a weighted average of the measurements taken such that the near ones are assigned more weight than the measurements taken earlier.
- Let us assume we are tracking the location of a spaceship with a laser sensor. Our laser sensor provides a single output $x(t)$, the position of the spaceship at time t . Both x and t are real-valued, i.e., we can get a different reading from the laser sensor at any instant in time.
- Now suppose that our laser sensor is somewhat noisy. To obtain a less noisy estimate of the spaceship's position, we would like to average together several measurements.
- Of course, more recent measurements are more relevant, so we will want this to be a weighted average that gives more weight to recent measurements.

We can do this with a weighting function $w(a)$, where a is the age of a measurement. If we apply such a weighted average operation at every moment, we obtain a new function s providing a smoothed estimate of the position of the spaceship:

$$s(t) = \int x(a)w(t-a)da$$

- This operation is called **convolution**. The convolution operation is typically denoted with an asterisk:

$$s(t) = (x * w)(t)$$

- In the above equation the x represents the input, $*$ represents the convolution operation and w denotes the filter that is applied.
- In the above example, w needs to be a valid probability density function, or the output is not a weighted average. Also, w needs to be 0 for all negative arguments, or it will look into the future, which is presumably beyond our capabilities. These limitations are particular to the example discussed above though.
- In general, convolution is defined for any functions for which the above integral is defined, and may be used for other purposes besides taking weighted averages.
- In convolutional network terminology, the first argument (x) to the convolution is often referred to as the input and the second argument (the function w) as the kernel. The output is sometimes referred to as the feature map.
- Similarly, for 2D input, re-estimation of each pixel is done by taking a weighted average of all its neighbours.

○ Convolutional Neural Network Architecture:

- A CNN typically has three layers: a convolutional layer, a pooling layer, and a fully connected layer.
- **Convolution Layer:**
- The convolution layer is the core building block of the CNN. It carries the main portion of the network's computational load. The CNN architecture is as shown in Figure 1.
- This layer performs a dot product between two matrices, where one matrix is the set of learnable parameters otherwise known as a kernel, and the other matrix is the restricted portion of the receptive field.
- The kernel is spatially smaller than an image but is more in-depth.

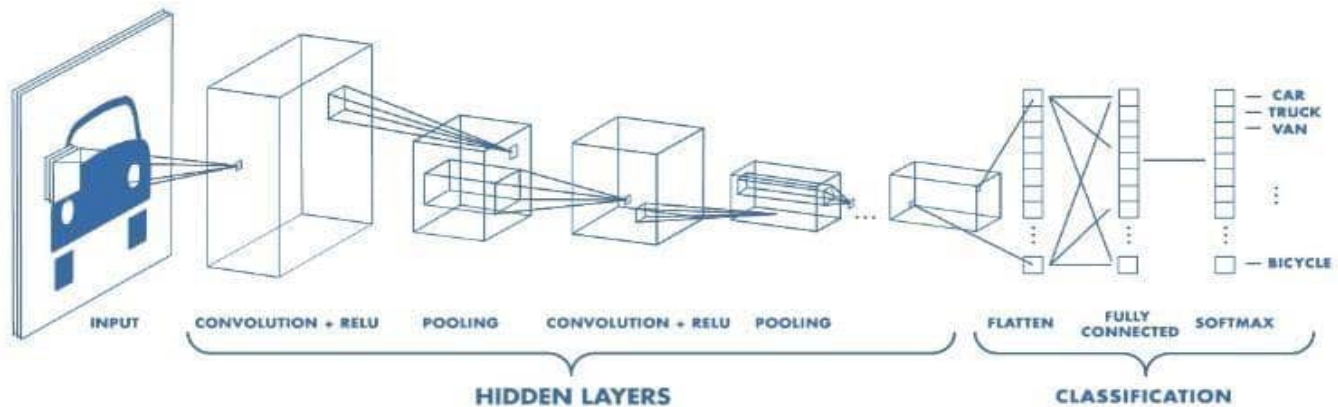


Fig 1. The CNN Architecture

- During the forward pass, the kernel slides across the height and width of the image-producing the image representation of that receptive region.
- This produces a two-dimensional representation of the image 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.
- If we have an input of size $W \times W \times D$ and D_{out} number of kernels with a spatial size of F with stride S and amount of padding P , then the size of output volume can be determined by the following formula:

$$W_{out} = \frac{W - F + 2P}{S} + 1$$

- This will yield an output volume of size $W_{out} \times W_{out} \times D_{out}$.
- **Input Volume Dimensions:** $W \times W \times D$: The input volume has a width and height of W , and a depth of D .
- In CNNs, depth D typically represents the number of channels (for example, $D=3$ for RGB images with three color channels).
- **Kernel Parameters F:** This is the spatial size of the filter (kernel). If $F=3$, it means the kernel is 3×3 .
- **S:** The stride is the number of pixels by which the filter slides over the input each time. If $S=1$, the filter moves one pixel at a time. If $S=2$, it moves two pixels, and so on.
- **P:** Padding refers to the number of pixels added around the input's border. Padding can help preserve spatial dimensions after convolution. For example, if padding $P=1$, one pixel layer is added to all four sides of the input.

- Number of Filters: Dout This represents the number of filters (kernels) used in the convolution layer, determining the depth of the output volume.
- Each filter produces one output channel, so the output will have Dout channels. The figure 2 depicts the process of how the activation map is obtained.

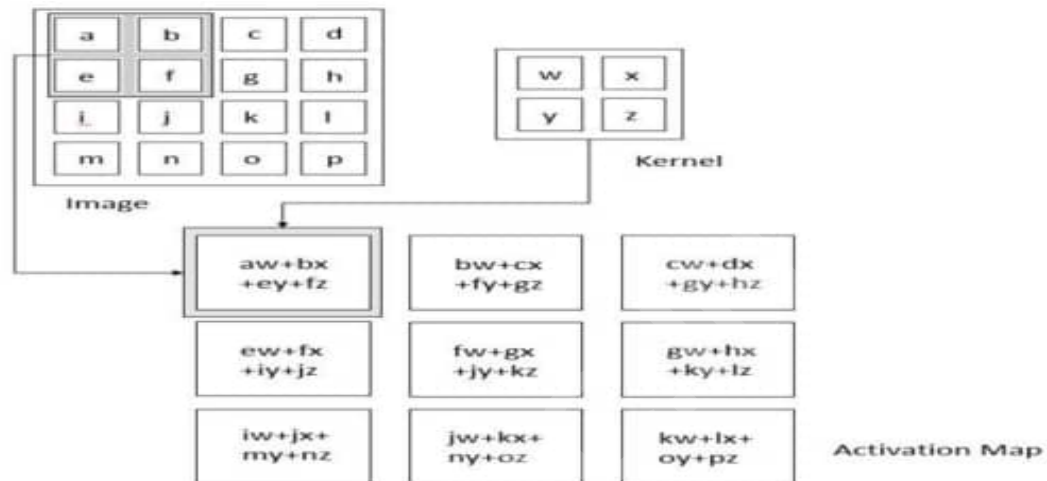


Fig 2. Representation of how the activation map/ feature map is obtained.

- Before we go on to the next layer let us try and understand Cross-Correlation and its Role in CNNs.
- In convolutional networks, the operation commonly referred to as "convolution" is, in fact, **crosscorrelation**.
- Cross-correlation computes the similarity between the input signal and the kernel as the kernel slides over the input. Mathematically, this can be written as:

$$(f \star g)[n] = \sum_m f[m]g[n+m]$$

- Here, f represents the input, g the kernel, and n the spatial or temporal shift. Cross-correlation captures localized patterns in the data, which is essential for feature extraction in CNNs.
- **Toeplitz Matrix in Convolution:** The convolution operation can be expressed in matrix form using a Toeplitz matrix, where each diagonal contains the same elements.
- For a 1D convolution, the input vector can be expanded into a Toeplitz matrix, enabling the convolution to be represented as:

$$\text{Output} = \text{Toeplitz}(\text{Input}) \times \text{Kernel}$$

- This matrix-based representation helps in understanding the linear transformations performed by convolutional layers.
- **Block-Circulant Matrices for Efficiency:** In advanced implementations, the Toeplitz matrix is often extended into a block-circulant matrix to optimize memory usage and computation. Blockcirculant structures allow for the decomposition of the convolution operation into efficient computations using FFT, reducing the computational cost significantly.
- **Circulant Matrices and FFT:**
- A circulant matrix is a specific type of Toeplitz matrix where each row is a circular shift of the previous row. Circulant matrices are fundamental to the FFT-based acceleration of convolutions, making them integral to efficient deep learning frameworks.
- By leveraging these mathematical structures, modern CNNs achieve high efficiency in both training and inference.

○ Motivation behind Convolution:

- Convolution leverages three important ideas that motivated computer vision researchers: sparse interaction, parameter sharing, and equivariant representation. Let's describe each one of them in detail.
- Trivial neural network layers use matrix multiplication by a matrix of parameters describing the interaction between the input and output unit. This means that every output unit interacts with every input unit. However, convolution neural networks have **sparse interaction**.
- This is achieved by making kernel smaller than the input e.g., an image can have millions or thousands of pixels, but while processing it using kernel, we can detect meaningful information that is of tens or hundreds of pixels.
- This means that we need to store fewer parameters that not only reduces the memory requirement of the model but also improves the statistical efficiency of the model.
- If computing one feature at a spatial point (x_1, y_1) is useful then it should also be useful at some other spatial point say (x_2, y_2) . It means that for a single two-dimensional slice i.e., for creating one activation map, neurons are constrained to use the same set of weights.
- In a traditional neural network, each element of the weight matrix is used once and then never revisited, while convolution network has **shared parameters** i.e., for getting output, weights applied to one input are the same as the weight applied elsewhere.

- Due to parameter sharing, the layers of convolution neural network will have a property of **equivariance to translation**. It says that if we changed the input in a way, the output will also get changed in the same way.
- Convolution leverages three important ideas that can help improve a machine learning system: sparse interactions, parameter sharing and equivariant representations.
- Traditional neural network layers use matrix multiplication by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. This means every output unit interacts with every input unit.
- In a regular neural network, every output unit (like a neuron) connects to every input unit, creating a dense web of connections.
- In CNNs, it's different. Instead of connecting everything, each output unit only interacts with a small, specific part of the input data. This is called sparse interactions or sparse connectivity.

How Does It Work?

- **Small Kernel:** CNNs use a small "kernel" (a sliding window or filter) that's much smaller than the input. For example:
 - An image might have thousands or millions of pixels (the input).
 - The kernel might only cover a tiny area, like 3x3 or 5x5 pixels, to detect small features like edges.
 - This means the network focuses on local patterns (e.g., an edge or a curve) rather than the whole image at once.
- when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that computing the output requires fewer operations.
- If there are m inputs and n outputs, then matrix multiplication requires $m \times n$ parameters and the algorithms used in practice have $O(m \times n)$ runtime (per example). If we limit the number of

connections each output may have to k , then the sparsely connected approach requires only $k \times n$ parameters and $O(k \times n)$ runtime.

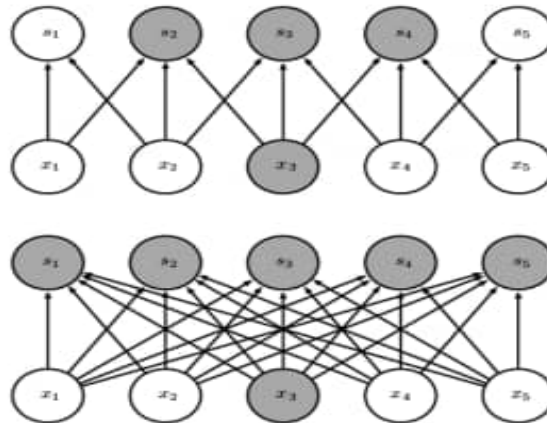


Figure 9.2: *Sparse connectivity, viewed from below:* We highlight one input unit, x_3 , and also highlight the output units in s that are affected by this unit. (*Top*) When s is formed by convolution with a kernel of width 3, only three outputs are affected by x . (*Bottom*) When s is formed by matrix multiplication, connectivity is no longer sparse, so all of the outputs are affected by x_3 .

Top Part: Sparse Connectivity with Convolution

- **What's Shown:**
 - x_3 only connects to s_2 , s_3 , and s_4 (three outputs).
 - Each output (s) connects to only a few nearby inputs. For example:
 - s_1 connects to x_1 and x_2 (and maybe x_3 , but the focus is on x_3 's limited reach).
 - The connections form a zigzag pattern, showing limited, local links.
- **Key Idea:** This uses a "kernel" of width 3 (a small sliding window). The kernel scans the inputs, so each output only "sees" 3 nearby inputs.

Why Sparse? Only a few arrows per connection—it's not a full web.

Benefits: Fewer connections mean less computation, less memory, and better efficiency for big data like images.

Bottom Part: Dense Connectivity with Matrix Multiplication

- **What's Shown:**
 - x_3 connects to **all** outputs: s_1 , s_2 , s_3 , s_4 , s_5 .
 - The pattern is a full mesh—every input links to every output, creating a dense tangle of arrows.

- **Key Idea:** This is like a traditional neural network layer where everything is multiplied together in a big matrix. No limits; it's fully connected.
- **Why Not Sparse?** All arrows are active—connectivity is "no longer sparse."

Drawbacks: More connections mean more parameters to learn, higher memory use, and slower processing.

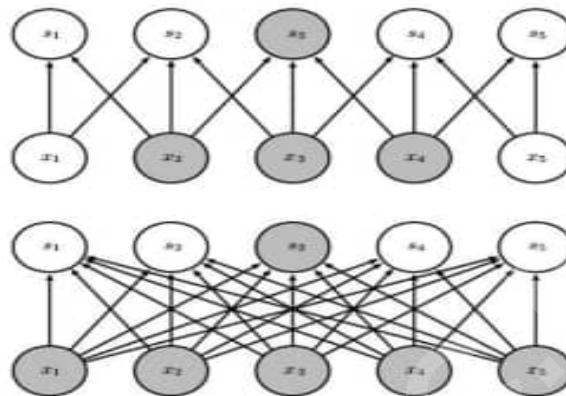


Figure 9.3: *Sparse connectivity, viewed from above:* We highlight one output unit, s_3 , and also highlight the input units in $\mathbf{x}$ that affect this unit. These units are known as the *receptive field* of s_3 . (*Top*) When $\mathbf{s}$ is formed by convolution with a kernel of width 3, only three inputs affect s_3 . (*Bottom*) When $\mathbf{s}$ is formed by matrix multiplication, connectivity is no longer sparse, so all of the inputs affect s_3 .

- Parameter sharing is a smart trick in Convolutional Neural Networks (CNNs) where the same set of "rules" (called parameters or weights) is reused multiple times in the model, instead of creating new ones for every part of the data.

How It Works in Traditional Neural Networks (For Comparison)

- In a basic neural network, each connection between input and output has its own unique weight.
- When calculating the output, each weight is multiplied by one specific input element and then thrown away—it's never reused.
- This is like having a huge list of custom instructions: one for each spot in the data. If your input has millions of pixels (like an image), you'd need millions of unique weights, which takes a ton of memory and computing power.

How Parameter Sharing Works in CNNs

- CNNs use a small "kernel" (a tiny matrix of weights, like a 3x3 grid) as the shared parameters.

- This kernel slides over the entire input (e.g., an image), applying the same weights to every position.
- Result: Instead of learning separate weights for every location, you learn just one set for the whole thing

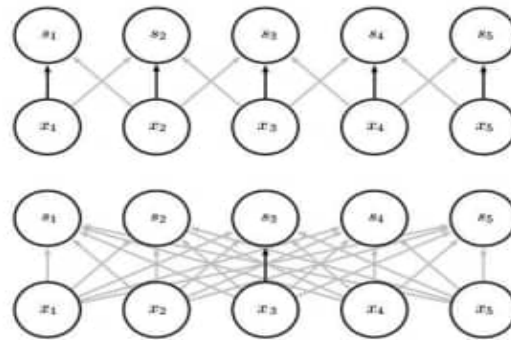


Figure 9.5: Parameter sharing: Black arrows indicate the connections that use a particular parameter in two different models. (Top) The black arrows indicate uses of the central element of a 3-element kernel in a convolutional model. Due to parameter sharing, this single parameter is used at all input locations. (Bottom) The single black arrow indicates the use of the central element of the weight matrix in a fully connected model. This model has no parameter sharing so the parameter is used only once.

Pooling Layer:

- A typical layer of a convolutional network consists of three stages. In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations. 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.
- 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 neighbourhood.
- Other popular pooling functions include the average of a rectangular neighbourhood, the L2 norm of a rectangular neighbourhood, or a weighted average based on the distance from the central pixel.
- In all cases, pooling helps to make the representation become approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change.
- Translation Invariance: 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. For example, if we want to find a corner defined by two edges meeting at a specific orientation, we need to preserve the location of the edges well enough to test whether they meet.

○ **Fully Connected Layer:**

- Neurons in this layer have full connectivity with all neurons in the preceding and succeeding layer as seen in regular FCNN.
- This is why it can be computed as usual by a matrix multiplication followed by a bias effect.
- The FC layer helps to map the representation between the input and the output.
- The non linearity is introduced in this fully connected layer through the activation functions.

Convolution and Pooling as an Infinitely Strong Prior:

- A prior is something we already assume to be true about a problem.
- Convolution and pooling assume that:
- Patterns matter more than exact positions (e.g., a "tail" is a tail no matter where it is in the photo).
- Small details add up to the big picture (e.g., spotting "fur" and "paws" helps identify a tiger).
- These assumptions (or priors) are so strong that they work incredibly well for many tasks, especially image recognition.
- They save time, computation, and generalize well to real-world images without needing extra data.
- An infinitely strong prior means having an assumption about how something works that is so powerful and rigid that it dominates how we process or interpret data no matter what the actual data says.
- IN CNN the assumptions made by the convolution and pooling are that patterns matter more than their exact location.
- A cat's ears will look the same whether they are in the top-left corner or the bottom-right corner of the image. This is called translation invariance.
- Local relationships are key: Convolution assumes the meaningful information in an image (e.g., an eye, a whisker) can be found by looking at small patches of the image at a time.

- These assumptions are so strong that the model focuses entirely on patterns and ignores other possibilities.
- For example: If an image's pattern looks slightly like an ear, the convolutional model might still say, "This must be part of a cat!" even if it's just a random shape.
- In other words, convolution and pooling force the model to think in terms of patterns and local information no matter what the data might suggest otherwise.

Variants of the Basic Convolution Function:

- Convolutional layers in deep learning have different variants depending on how the convolution operation is applied:
- **Full Convolution:** Expands the input by padding it with zeros, ensuring the output is larger than the input. This helps in capturing more context around the edges.
- **Valid Convolution:** No padding is added, so the output is smaller than the input because only fully overlapping parts are computed.
- **Same Convolution:** Pads the input to maintain the same output size as the input.
- **Unshared Convolution:** Uses different filters for each location in the input, rather than sharing the same filter across all positions. This increases flexibility but requires more parameters.
- **Tiled Convolution:** Reuses filters at intervals instead of applying them at every position, balancing parameter sharing and diversity.
- These variations provide trade-offs between computational cost, memory, and the ability to capture features. So, let's discuss each variant in detail.
- **Full Convolution:** To understand better the full convolution let's elaborate these terms:
- Kernel K with element $K_{i,j,k,l}$ giving the connection strength between a unit in channel i of output and a unit in channel j of the input, with an offset of k rows and l columns between the output unit and the input unit.
- Input: $V_{i,j,k}$ with channel i , row j and column k
- Output Z same format as V
- Use 1 as first entry
- **0 Padding 1 stride, the equation is:**

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

- **0 Padding s stride, the equation is:**

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

- Convolution with a stride greater than 1 pixel is equivalent to conv with 1 stride followed by downsampling.
- In the below figure 3 depicts the convolution with stride. In the top figure stride value is 2 and that is applied in a single operation.
- In the bottom figure the stride value is greater than 1 pixel which is convolution with unit stride along with downsampling.
- The 2-step approach involving downsampling is computationally wasteful because it computes many values that later on are discarded.

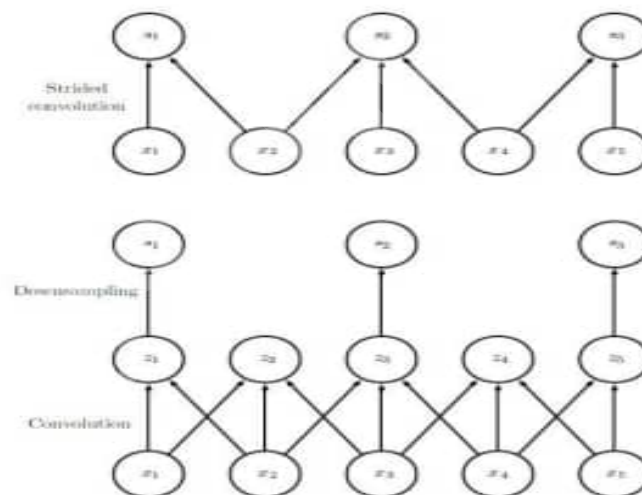


Figure 9.12: Convolution with a stride. In this example, we use a stride of two. (Top) Convolution with a stride length of two implemented in a single operation. (Bottom) Convolution with a stride greater than one pixel is mathematically equivalent to convolution with unit stride followed by downsampling. Obviously, the two-step approach involving downsampling is computationally wasteful, because it computes many values that are then discarded.

Fig 3. Convolution with stride.

Some 0 Paddings and 1 stride

- Without 0 paddings, the width of representation shrinks by one pixel less than the kernel width at each layer. We are forced to choose between shrinking the spatial extent of the network rapidly and using small kernel. 0 padding allows us to control the kernel width and the size of the output independently.
- Special case of 0 padding:
- Valid: no 0 padding is used. Limited number of layers.

- Same: keep the size of the output to the size of input. Unlimited number of layers. Pixels near the border influence fewer output pixels than the input pixels near the Center.
- Full: Enough zeros are added for every pixel to be visited k (kernel width) times in each direction, resulting width $m + k - 1$. Difficult to learn a single kernel that performs well at all positions in the convolutional feature map.
- Usually, the optimal amount of 0 padding lies somewhere between 'Valid' or 'Same'.

Unshared Convolution: Unlike standard convolution, where the same kernel is shared across all spatial regions, unshared convolution uses a unique kernel for each spatial location.

- Properties:
- Each input region has its own filter.
- Captures highly localized, diverse spatial features.
- Requires significantly more parameters and computational resources.

Working:

- The input is divided into small overlapping or non-overlapping patches.
- Each patch is processed by a separate kernel, which allows fine-tuned learning specific to localized features.
- The outputs of these convolutions are combined to form the final feature map.

Use Case:

- Tasks like image style transfer, where local patterns in different spatial regions vary significantly.
- Applications in medical imaging to analyse highly localized features in scans or images.

Drawbacks:

- High computational cost due to the large number of parameters.
- Increased memory usage, as unique filters must be stored for each spatial location.
- In some case when we do not want to use convolution but want to use locally connected layer. We use Unshared convolution. Indices into weight W
- i : the output channel
- j : the output row;
- k : the output column
- l : the input channel
- m : row offset within input
- n : column offset within input

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

- Useful when we know that each feature should be a function of a small part of space, but no reason to think that the same feature should occur across all the space. eg: look for mouth only in the bottom half of the image.
- 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.

Advantages:

- Reduce memory consumption.
- Increase statistical efficiency.
- Reduce computation for both forward and backward prop.
- **Tiled Convolution:**
- **Definition:** A hybrid approach between standard convolution and unshared convolution, where kernels are shared across tiles or groups of spatial regions instead of globally.
- **Properties:**
 - ✦ Provides a middle ground by allowing partial sharing of parameters within specific spatial regions (tiles).
 - ✦ Retains some flexibility of unshared convolution while reducing the computational burden.
- **Working:**
 - ✦ Divide the input into tiles, which are typically rectangular or square regions of the input.
 - ✦ Assign a unique kernel to each tile. Kernels are shared within a tile but not across tiles.
 - ✦ Perform convolution independently on each tile.
 - ✦ Concatenate the results from all tiles to produce the final output feature map.
- **Use Case:**
 - ✦ Tasks involving repetitive but spatially localized patterns, such as texture synthesis or segmentation.
 - ✦ Useful in scenarios where there is a trade-off between the diversity of spatial features and computational efficiency.
- **Design Parameters:**

- ✦ Tile Size: Determines the spatial granularity of kernel sharing. Larger tiles lead to more sharing, reducing parameter count.
- ✦ Number of Kernels: Affects model capacity and complexity. Too few kernels can underfit, while too many may overfit or become computationally expensive.
- **Advantages:**
 - ✦ Offers flexibility to capture diverse spatial patterns without the full parameter cost of unshared convolution.
 - ✦ Scales well for moderately complex tasks.
 - ✦ Challenges:
 - ✦ Requires careful selection of tile size and kernel sharing strategy to balance performance and efficiency.
 - ✦ May introduce boundary artifacts between tiles if not handled properly.

Structured Outputs:

- As studied in the previous chapters, a convolutional network works with multidimensional tensors. This can be used, not only to solve a classification or regression task, but to output a structured object in form of a tensor.
- For example, with a convolutional layer, we can output a tensor, where each pixel has a vector containing probabilities for belonging to a certain class. This could be used to label every pixel in an image and use it for segmentation.
- A problem which has to be considered is the shrinking size of the input. Such reductions of the size are mainly a result of pooling.
- To avoid such a shrinkage, we can first of all avoid pooling at all or using pooling layers with a stride of one.
- One also may work with a lower resolution when it is possible. Another possibility is to upscale the image again with multiple convolutions. This can be done by producing an initial guess about the missing pixels.
- From there, we can use this to create a recurrent neuronal network with the same kernels in each step. In each step our initial guess gets refined.
- Strategy for size reduction issue:
 - ✦ avoid pooling altogether

- ✦ emit a lower-resolution grid of labels
- ✦ pooling operator with unit stride
- One strategy for pixel-wise labelling of images is to produce an initial guess of the image label.
- Produce an initial guess of the image labels. Refine this initial guess using the interactions between neighbouring pixels. Repeat this refinement step several times corresponds to using the same convolution at each stage, sharing weights between last layers of the deep net. The figure 4, Recurrent Convolutional Network:

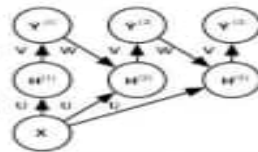


Figure 9.17: An example of a recurrent convolutional network for pixel labelling. The input is an image tensor $\mathbf{X}$, with axes corresponding to image rows, image columns, and channels (red, green, blue). The goal is to output a tensor of labels $\mathbf{Y}$, with a probability distribution over labels for each pixel. This tensor has axes corresponding to image rows, image columns, and the different classes. Rather than outputting $\mathbf{Y}$ in a single shot, the recurrent network iteratively refines its estimate $\mathbf{Y}$ by using a previous estimate of $\mathbf{Y}$ as input for creating a new estimate. The same parameters are used for each updated estimate, and the estimate can be refined as many times as we wish. The tensor of convolution kernels $\mathbf{U}$ is used on each step to compute the hidden representation given the input image. The kernel tensor $\mathbf{V}$ is used to produce an estimate of the labels given the hidden values. On all but the first step, the kernels $\mathbf{W}$ are convolved over $\mathbf{Y}$ to provide input to the hidden layer. On the first time step, this term is replaced by zero. Because the same parameters are used on each step, this is an example of a recurrent network, as described in chapter 10.

Fig 4. Recurrent Convolutional network for pixel labelling.

○ Data Type:

- Convolutional networks are capable of processing a wide variety of data, often with multiple channels. Typically, datasets contain examples that share the same spatial dimensions, a requirement also common for traditional multilayer perceptron.
- However, CNNs offer the flexibility to handle datasets with examples of varying sizes. This is achieved by applying one or more kernels across the data, with the number of kernel applications depending on the data's dimensions, which can lead to variable output sizes.
- While this variability can sometimes pose challenges, maintaining consistent output sizes can be addressed using pooling layers.
- These layers adapt their pooling regions based on the input size, enabling the network to process inputs of arbitrary dimensions while producing outputs with uniform size.

○ Examples of Data Formats for Convolutional Networks:

• 1-Dimensional Data

Single Channel:

- ✦ Example: Audio waveform.
- ✦ The convolution is performed over the time axis.
- ✦ Time is discretized into steps, and the amplitude of the waveform is measured at each step.

Multi-Channel:

- ✦ Example: Skeleton animation data.
- ✦ Represents animations of 3D characters. Each character's pose at a given time is described by the angles of its skeletal joints.
- ✦ Each channel corresponds to the angle for one joint's axis over time.

○ 2-Dimensional Data**Single Channel:**

- ✦ Example: Pre-processed audio data (Fourier transform).
- ✦ Audio is converted into a 2D tensor:
- ✦ Rows: Represent different frequencies.
- ✦ Columns: Represent different time points.
- ✦ Convolution along the time axis makes the model invariant to time shifts.
- ✦ Convolution along the frequency axis makes the model invariant to pitch/octave shifts, ensuring melodies at different octaves are represented consistently but at different output heights.

Multi-Channel:

- ✦ Example: Color image data.
- ✦ Each channel represents one of the three primary colors:
 - ✦ Red, Green, and Blue.
- ✦ The convolution kernel operates across both the horizontal and vertical axes of the image, ensuring translation equivariance in both directions.

○ 3-Dimensional Data**Single Channel:**

- ✦ Example: Volumetric data (e.g., CT scans).
- ✦ Represents data in three spatial dimensions, such as depth, height, and width.

Multi-Channel:

- ✦ Example: Color video data.
- ✦ Dimensions include:
- ✦ Time: The sequence of frames.

- ✦ Height: Vertical resolution of each frame.
- ✦ Width: Horizontal resolution of each frame.

○ Efficient Convolutional Algorithms:

- Efficient convolutional algorithms are designed to reduce computational overhead while maintaining the accuracy of convolutional operations. These methods optimize performance in CNNs, especially for large datasets and complex architectures.
- As already discussed, variants of the basic convolution function. One option to decrease the computational effort is to use the strided convolution. But even with strides the convolution is quite expensive to calculate, since we do a pointwise multiplication for each element.
- To overcome this problem, we may regard the input as a signal. Using the Fourier transform, the convolution becomes a simple multiplication.

Algorithm:

- Take the Fourier Transform (e.g., Fast Fourier Transform/FFT) of both the input and kernel.
- Multiply the transformed signals pointwise.
- Apply an inverse Fourier Transform to get the output.

Fast Fourier Transform (FFT)-Based Convolutions:

- Uses the Fast Fourier Transform (FFT) to convert spatial domain convolutions into element-wise multiplications in the frequency domain.
- **Significant/efficient:** computational savings for large kernel sizes.
- **Drawback: Overhead** of transforming between **spatial** and **frequency** domains.
- **Another approach** can be used, when the filters are separable. A filter of dimension d is separable if it can be expressed as a product of d vectors.
- When a filter is separable, the convolution of the d -dimensional filter is equal to d one-dimensional convolutions.
- In this case, the runtime and memory consumption decreases from $O(dn)$ to $O(d*n)$.

1D kernel?

- A **1D kernel** is just a vector — a single row or column of weights.
- For example:

outer product mean?

- The **outer product** of two vectors produces a matrix by multiplying every element of one vector by every element of the other.

Example:

$$k_x = [1, 2, 1], \quad k_y = [1, 0, -1]$$

Then their **outer product** is:

$$K = k_y^T \otimes k_x = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \times [1, 2, 1] = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

See that?

We just built the 2D kernel K by multiplying two 1D kernels together — one for rows (y) and one for columns (x).

“separable” mean?

- If you can represent a 2D (or 3D, or higher) kernel as a **product of smaller 1D kernels**, it is **separable**.
- In other words:

$$K(x, y) = k_x(x) \cdot k_y(y)$$

or in general for d dimensions:

$$K(x_1, x_2, \dots, x_d) = k_1(x_1) \cdot k_2(x_2) \cdot \dots \cdot k_d(x_d)$$

Why does this matter?

- Because this factorization lets us replace one **expensive d-dimensional convolution** with **multiple cheaper 1D convolutions**.

Example:

- A 2D 5×5 kernel normally requires 25 multiplications per pixel.
- If it's separable (say 5×1 and 1×5), we can:
 - Convolve once horizontally (5×1)
 - Then convolve vertically (1×5)
 - Only **10 multiplications per pixel** instead of 25!
- This makes convolution much **faster and memory-efficient**.

In Higher Dimensions (d-Dimensional)

- For a **d-dimensional** kernel (say 3D for video or volumetric data):
- If you can express the kernel as:

$$K(x, y, z) = k_x(x) \cdot k_y(y) \cdot k_z(z)$$

- Then it's **separable** — it means you can perform **three 1D convolutions** instead of one full 3D convolution.

Separable Convolutions

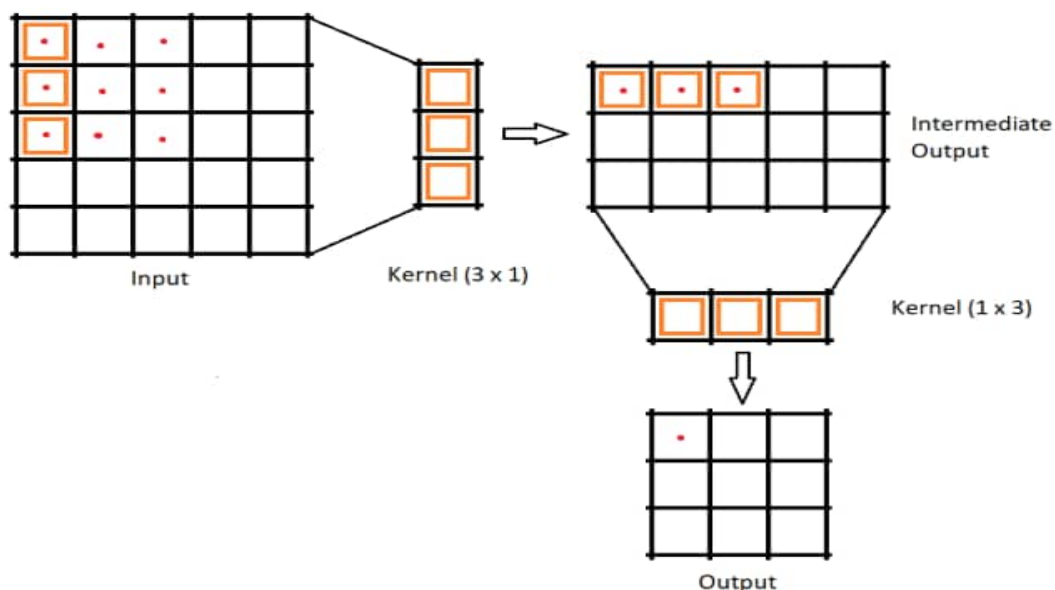
- **Definition:** If a d-dimensional kernel can be factored as the outer product of d 1D kernels (one per dimension), it is separable.
- Why is it efficient?
 - Naive d-dimensional convolution: Requires $O(w^d)$ computations (with kernel width w).
 - Separable convolution: Only $O(d \times w)$ computations, because it decomposes the convolution into successive 1D convolutions for each dimension.
 - Parameter reduction: Instead of storing wd weights, we just store dw weights.

Example: **2D Gaussian blur is separable:** you can blur horizontally first, then vertically, much faster than convolving the full 2D kernel.

- **Mainly there are two types of Separable Convolutions**

- Spatially Separable Convolutions.
- Depth-wise Separable Convolutions.

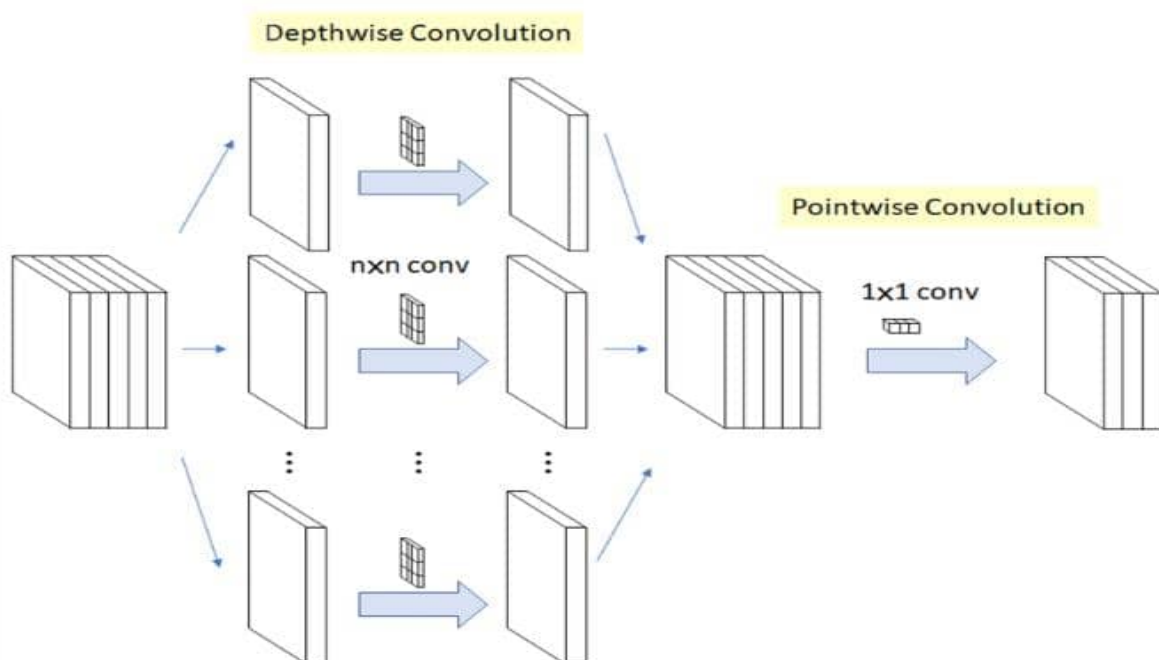
Convolutions with spatially separable kernels



- In the conventional convolutions, if the kernel is 3×3 then the number of parameters would be 9.
- In spatially separable convolution we divide the kernel into two kernels of shapes 3×1 and 1×3 .
- The input is first convolved with 3×1 kernel and then with 1×3 , then the number of parameters would be $3 + 3 = 6$. So less matrix multiplication is required.
- An important thing to note here is that not every kernel can be separated. Because of this drawback, this method is used lesser compared to Depthwise separable convolutions.

Depthwise Separable Convolutions

- Principle: Used in modern networks like MobileNet and Xception, this operation divides regular convolution into two steps:
 - Depthwise convolution: Applies a single filter per input channel, processing spatial information independently (kernel shape, e.g., $3 \times 3 \times 1$ per channel).
 - Pointwise convolution: Uses 1×1 convolutions to mix information across channels.
- Why is this faster? Greatly reduces the number of parameters and multiplications compared to standard convolution.
 - Example calculation: For a 3×3 filter over 32 input channels with 64 outputs, standard convolution needs 18,432 parameters, while depthwise separable needs only 2,336—nearly an 8x reduction.
- Applications: Ubiquitous in mobile-optimized architectures (MobileNet, Xception)



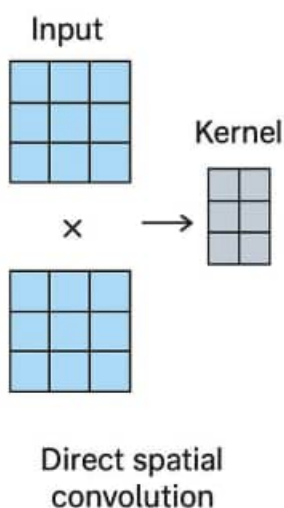
Depthwise convolution

- Let us assume we have an image input of shape $7 \times 7 \times 3$. We make sure after the depthwise convolution the intermediate image has the same depth. This is done by convolving with 3 kernels with shape $3 \times 3 \times 1$.
- Each kernel iterates on only one channel of the image-producing an intermediate output of shape $5 \times 5 \times 1$ which are stacked together to create an output of shape $5 \times 5 \times 3$.

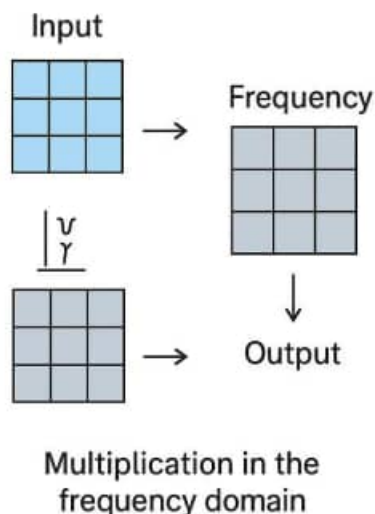
Pointwise convolution

- After the depthwise convolution, we have an intermediate output of shape $5 \times 5 \times 3$. Now we need to increase the depth of the output, which is done by convolving with a kernel with shape $1 \times 1 \times \text{depth}$ which is called a pointwise convolution.
- Let us assume the depth is 32. Then after pointwise convolution, the output would have the shape $5 \times 5 \times 32$. Which is equivalent to convolution with 32 filters of shape 5×5 .
- In this example, the regular convolution would have to do 32 $3 \times 3 \times 3$ kernels that move 5×5 times that is 21600 multiplication but in this method, we would have to do $3 \times 3 \times 3 \times 5 \times 5 + 32 \times 1 \times 1 \times 3 \times 5 \times 5 = 3075$ multiplication.
- So we can see that this method can eliminate a large chunk of multiplication.

NAIVE CONVOLUTION



FFT-BASED CONVOLUTION



SEPARABLE CONVOLUTION

