Course Materials for GEN-AI

Northeastern University

These materials have been prepared and sourced for the course **GEN-AI** at Northeastern University. Every effort has been made to provide proper citations and credit for all referenced works.

If you believe any material has been inadequately cited or requires correction, please contact me at:

Instructor: Ramin Mohammadi
r.mohammadi@northeastern.edu

Thank you for your understanding and collaboration.

Content sourced from multiple references - see Reference section for details

Convolutional Neural Networks (CNNs)

Introduction

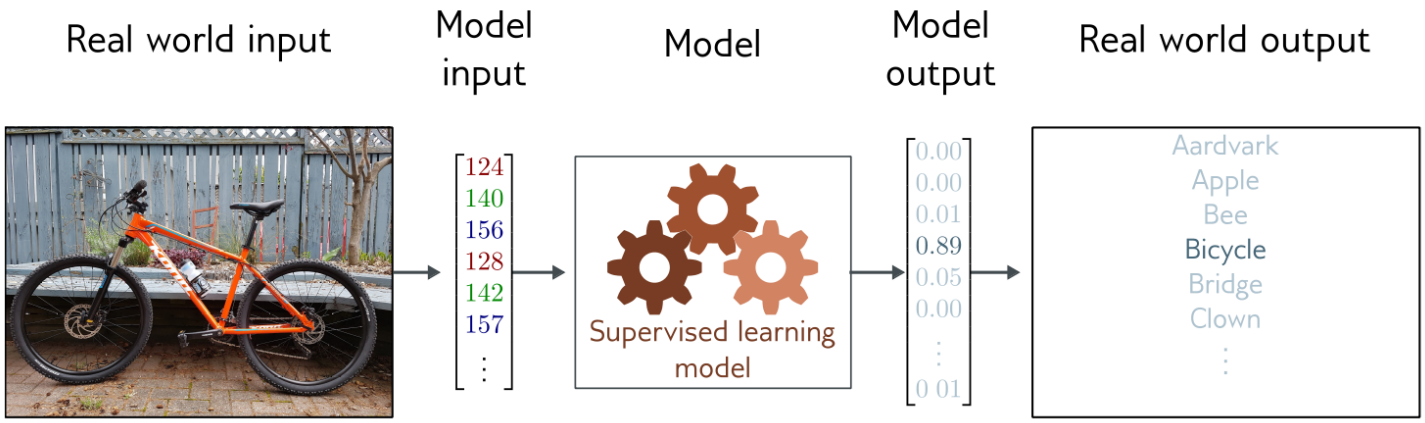


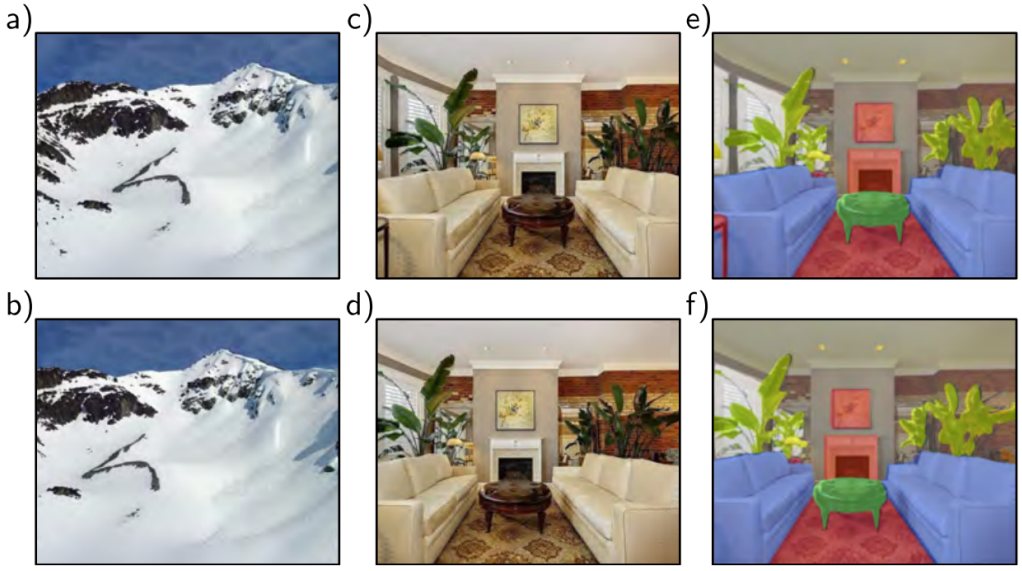
Image Classification

Images have three properties that suggest the need for specialized model architecture.

- First, they are high-dimensional. A typical image for a classification task contains 224×224 RGB values (i.e., 150,528 input dimensions). Hidden layers in fully connected networks are generally larger than the input size, so even for a shallow network, the number of weights would exceed 150, 5282 , or 22 billion. This poses obvious practical problems in terms of the required training data, memory, and computation.
- Second, nearby image pixels are statistically related. However, fully connected networks have no notion of "nearby" and treat the relationship between every input equally. If the pixels of the training and test images were randomly permuted in the same way, the network could still be trained with no practical difference.
- Third, the interpretation of an image is stable under geometric transformations. An image of a tree is still an image of a tree if we shift it leftwards by a few pixels. However, this shift changes every input to the network. Hence, a fully connected model must learn the patterns of pixels that signify a tree separately at every position, which is clearly inefficient.

Convolutional layers process each local image region independently, using parameters shared across the whole image. They use fewer parameters than fully connected layers, exploit the spatial relationships between nearby pixels, and don't have to re-learn the interpretation of the pixels at every position. A network predominantly consisting of convolutional layers is known as a convolutional neural network or CNN.

Invariance and Equivariance



Invariance and Equivariance for translation. a–b) In image classification, the goal is to categorize both images as "mountain" regardless of the horizontal shift that has occurred. In other words, we require the network prediction to be invariant to translation. c,e) The goal of semantic segmentation is to associate a label with each pixel. d,f) When the input image is translated, we want the output (colored overlay) to translate in the same way. In other words, we require the output to be equivariant with respect to translation. Panels c–f) adapted from Bousselham et al. (2021).

1- A function f[x] is **invariant** to a transformation t[] if:

$$f[t[X]] = f[X]$$

i.e., the function output is the same even after the transformation is applied. Classification networks should be invariant to transformation.

2- A function f is equivariant to transformation g if:

$$f(g(x)) = g(f(x))$$

i.e., the output is transformed in the same way as the input. Networks for per-pixel image segmentation should be equivariant to transformations.

$$\begin{array}{ccc} X & \xrightarrow{g} & X \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f} \\ Y & \xrightarrow[g]{} & Y \end{array}$$

Convolution

A Convolutional Neural Network (CNN) is a type of neural network that contains at least one convolutional layer. The convolution operation involves flipping a function w (often referred to as a filter or kernel) and sliding it over another function x (typically the input), instead of performing general matrix multiplication. This process is fundamental to many applications, including image processing. For a visual demonstration, refer to this Visual Demo for Convolution.
Convolution function: In the continuous form, the convolution of two functions $x(a)$ and $w(a)$ is defined as:

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

Where:

- $x(a)$ is the input function.
- $w(t-a)$ is the flipped and shifted function (kernel or filter).
- This operation is denoted by the convolution symbol $*$.

The continuous convolution can also be compactly written as:

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

In the discrete form, the convolution becomes a summation:

$$S(t) = \sum_{a=-\infty}^{\infty} x[a] \cdot w[t-a]$$

Convolution in Image Processing: When applying convolution to images, the operation is extended to two dimensions. The convolution of an image $I(x, y)$ with a kernel $K(i, j)$ is defined as:

$$(I * K)(x, y) = \sum_{i,j} I(x-i, y-j) \cdot K(i, j)$$

Here:

- $I(x, y)$ represents the input image.
- $K(i, j)$ is the kernel (or filter) applied to the image.
- The result is a new image obtained by sliding the kernel over the input image and computing the weighted sum of overlapping values.

Translation

In image processing, translation shifts an image $I(x, y)$ by a pixel value (t_x, t_y):

$$I'(x, y) = I(x - t_x, y - t_y)$$

where I' is the translated image.

- **Translation of Convolution** When applying convolution after translation:

  $$(I' * K)(x, y) = \sum_{i,j} I(x - t_x - i, y - t_y - j) \cdot K(i, j)$$

  Here, I' represents the translated image, and K is the kernel.

- **Convolution of Translation** When applying translation after convolution:

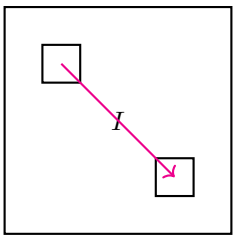
  $$I * K(x, y) = \sum_{i,j} I(x - t_x - i, y - t_y - j) \cdot K(i, j)$$

Conclusion This demonstrates that K is **equivariant to translation**, meaning that the convolution of a translated image is the same as translating the convolution result:

$$(I' * K)(x, y) = I * K(x, y)$$

Translation equivariance is a fundamental property of convolutional operations and is widely used in convolutional neural networks.

Note: This does not hold for all transformations, such as rotation.



In the above representation, if you shift this pixel to here, the convolution is the same as if you didn't translate it.

Kernel Flipping

In the definition of convolution, the kernel (or filter) is mirrored (flipped) around its center before performing element-wise multiplication and summation. This flipping ensures that the operation adheres to the mathematical definition of convolution and maintains commutativity:

$$f * g = g * f$$

How to Flip a Kernel

To mathematically perform convolution, a kernel K of size $m \times n$ is flipped by reversing its elements along both the horizontal and vertical axes. The flipped kernel can be expressed as:

$$K_{\text{flipped}}[i, j] = K[m - 1 - i, n - 1 - j]$$

Examples of Flipping:

- 1. Vertical Flip of a Kernel:

$$K = \begin{bmatrix} a & d \\ b & e \\ c & f \end{bmatrix} \xrightarrow{\text{Vertical Flip}} K_{\text{flipped}} = \begin{bmatrix} c & f \\ b & e \\ a & d \end{bmatrix}$$

- 2. Horizontal Flip of a Kernel:

$$K = [a, b, c] \xrightarrow{\text{Horizontal Flip}} K_{\text{flipped}} = [c, b, a]$$

- 3. Combining Horizontal and Vertical Flips: For a kernel:

$$K = \begin{bmatrix} k_{00} & k_{01} & k_{02} \\ k_{10} & k_{11} & k_{12} \\ k_{20} & k_{21} & k_{22} \end{bmatrix}$$

$$K \xrightarrow{\text{Horizontal Flip}} \begin{bmatrix} k_{02} & k_{01} & k_{00} \\ k_{12} & k_{11} & k_{10} \\ k_{22} & k_{21} & k_{20} \end{bmatrix} \xrightarrow{\text{Vertical Flip}} \begin{bmatrix} k_{22} & k_{21} & k_{20} \\ k_{12} & k_{11} & k_{10} \\ k_{02} & k_{01} & k_{00} \end{bmatrix}$$

After flipping, element $(2, 0)$ in the original kernel moves to position $(0, 2)$.

Convolution vs Cross-Correlation

- **Convolution** is commutative:

$$S[i, j] = (I * K)(i, j) = \sum_m \sum_n I[i, j] K[i - m, j - n]$$

Flipping a signal: In convolution, the kernel/filter is flipped before applying it to the input.

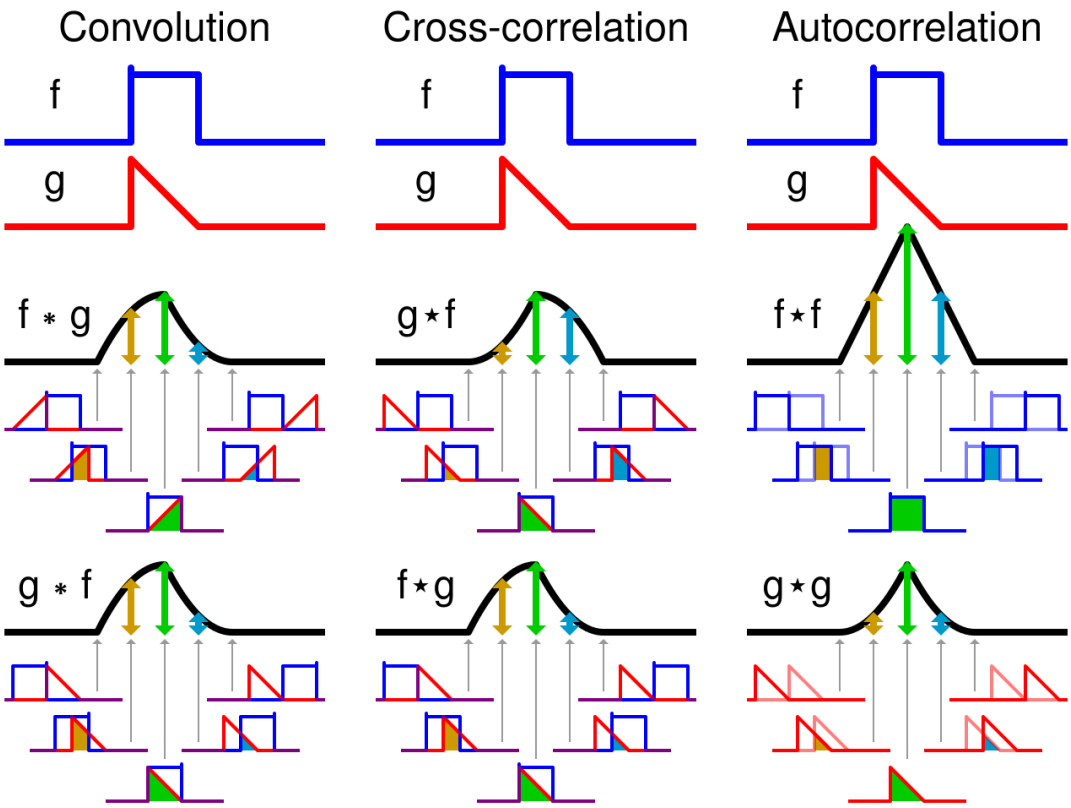
- **Cross-correlation**:

$$S[i,j] = (I * K)(i,j) = \sum_m \sum_n I[i,j]K[i+m, j+n]$$

No flipping: In cross-correlation, we do **not** flip the signal. This is the most commonly used operation, but we still refer to it as convolution. We just slide the kernel over the input signal.

Additional Explanation

- These operations can be thought of as a measure of similarity.
- When a filter (kernel) is learning an edge, the similarity between the edge in the image and the filter is highest when the filter perfectly aligns with the edge.
- For additional clarity, refer to this Visualization of Cross Correlation and Convolution.



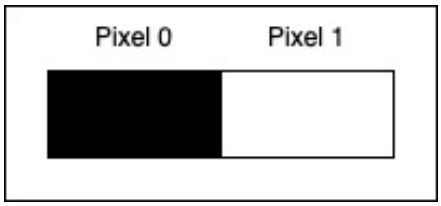
Visual comparison of convolution, cross−correlation and autocorrelation.

Flipping in Machine Learning Contexts

Although mathematical convolution includes flipping, most machine learning frameworks (e.g., TensorFlow, PyTorch) implement **cross-correlation** instead.
This approach avoids flipping for computational efficiency, and the difference is negligible because kernel weights are learned during training. When strict convolution is required, flipping the kernel manually ensures the operation aligns with the mathematical definition.

Edge Detection

- **1D Edge Detection**



A 1D edge detection example showing two pixels: Pixel 0 (black) and Pixel 1 (white), representing a sharp transition between intensities, which can be identified as an edge.

To detect an edge in a 1D signal, compare the neighbouring pixels.

- **Equation 1 (without intensity):**

$$\text{edge}(x) = \underbrace{-\text{pixels}[x-1]}_{neighbourpixel} - \underbrace{\text{pixels}[x]}_{targetpixel}$$

  where:

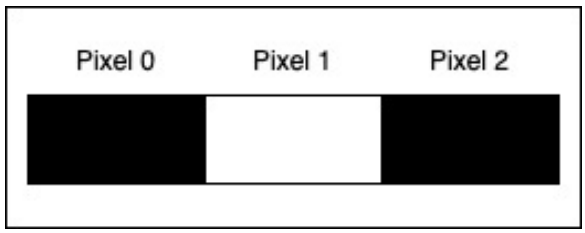
  * pixel[x-1] = pixel 0 and pixel[x] = pixel 1

  Consider a simple image where '1' is black and '0' is white. The possible scenarios for two pixels are: '00', '01', and '11' (no '10' since the first block is black and the next is white). If the edge value is outside of $\in [0, 1]$

Pixel Pattern	Explanation
00	$0 - 0 = 0$ (Not Edge)
01	$0 - 1 = -1$ (Edge)
11	$1 - 1 = 0$ (Not Edge)

- **Equation 2 (with intensity):** To enhance edge detection, consider the intensity of the centre pixel.

$$\text{edge}(x) = -\text{pixels}[x-1] + 2 \cdot \text{pixels}[x] - \text{pixels}[x+1]$$



A 1D edge detection example showing three pixels: Pixel 0 (black), Pixel 1 (white), and Pixel 2 (black), demonstrating an edge scenario where there is a sharp transition between Pixel 1 and its neighbouring black pixels, indicating an edge at both boundaries.

- **Clamping Values** If the result of the edge detection operation falls outside the range '[0, 1]', we "clamp" it to ensure valid values. For example:

$$\text{if edge}(x) = -2, \text{return} 0$$

  This ensures that pixel values remain within a valid range for intensity (usually grayscale or binary images).

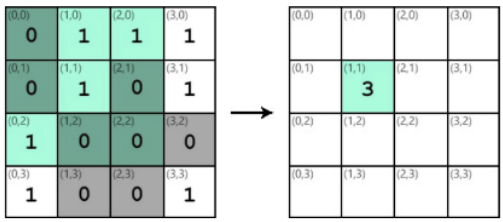
Pixel Pattern	Result (Before Clamping)	Clamped Value	Edge / Not Edge
000	$-0 + 2 \cdot 0 - 0 = 0$	0	Not Edge
001	$-0 + 2 \cdot 0 - 1 = -1$	0	Not Edge
010	$-0 + 2 \cdot 1 - 0 = 2$	1	Edge
011	$-0 + 2 \cdot 1 - 1 = 1$	1	Edge
111	$-1 + 2 \cdot 1 - 1 = 0$	0	Not Edge
110	$-1 + 2 \cdot 1 - 0 = 1$	1	Edge
100	$-1 + 2 \cdot 0 - 0 = -1$	0	Not Edge

A 2D edge detection example illustrating a continuous region of intensity, with black pixels surrounding a white central region, indicating the presence of edges at both the left and right boundaries of the white region.

- **2D Edge Detection** For detecting edges in a 2D image, consider the pixel at position '(x, y)' and its neighbours.

$$\text{edge}(x, y) = -\text{pixels}(x-1, y) - \text{pixels}(x, y-1) + 4 \cdot \text{pixels}(x, y) - \text{pixels}(x+1, y) - \text{pixels}(x, y+1)$$

- **Kernel for Edge Detection**
  - Rather than writing these expressions as a sequence of arithmetic operations, edge detection can be simplified using a kernel matrix.
  - The kernel is centered over the **target pixel**, with the middle element of the kernel aligned with the target.
    Kernel matrix:
$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$
  - **Example:** The kernel is placed on the image grid and aligned over the target pixel '(1,1)' and applied, producing the result '3' at that position after the convolution operation.



Kernel operation for edge detection with the target pixel (1,1) producing the result '3'.

Types of Kernels

- **Edge Detection Kernels:**These kernels (e.g., Sobel filters) highlight changes in intensity across an image.
  - **Horizontal Sobel Filter:**
$$\begin{bmatrix} -2 & 0 & 2 \\ -2 & 0 & 2 \\ -2 & 0 & 2 \end{bmatrix}$$
  - **Vertical Sobel Filter:**
$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$
- **Sharpening Kernels:**
  - These kernels emphasize the high-frequency components, making the fine details and image features more pronounced.

- Sharpening helps by enhancing finer details in the image.
  Example:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

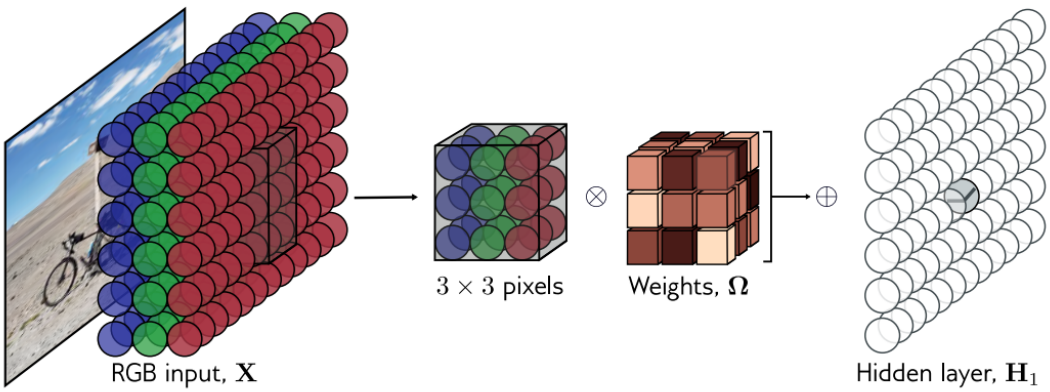
- **Blurring Kernels:**
  - Blurring kernels are used to smooth an image by reducing noise and details. They often use Gaussian filters.
  - This is useful when focusing on larger features while reducing noise.
    Example (3x3 Gaussian Filter with $\sigma = 1$):

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

- **Embossing Kernels:**
  - Used to create a 3D shadow effect, emphasizing edges in a specific direction.
  - This highlights contours and textures.
    Example:

$$\begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

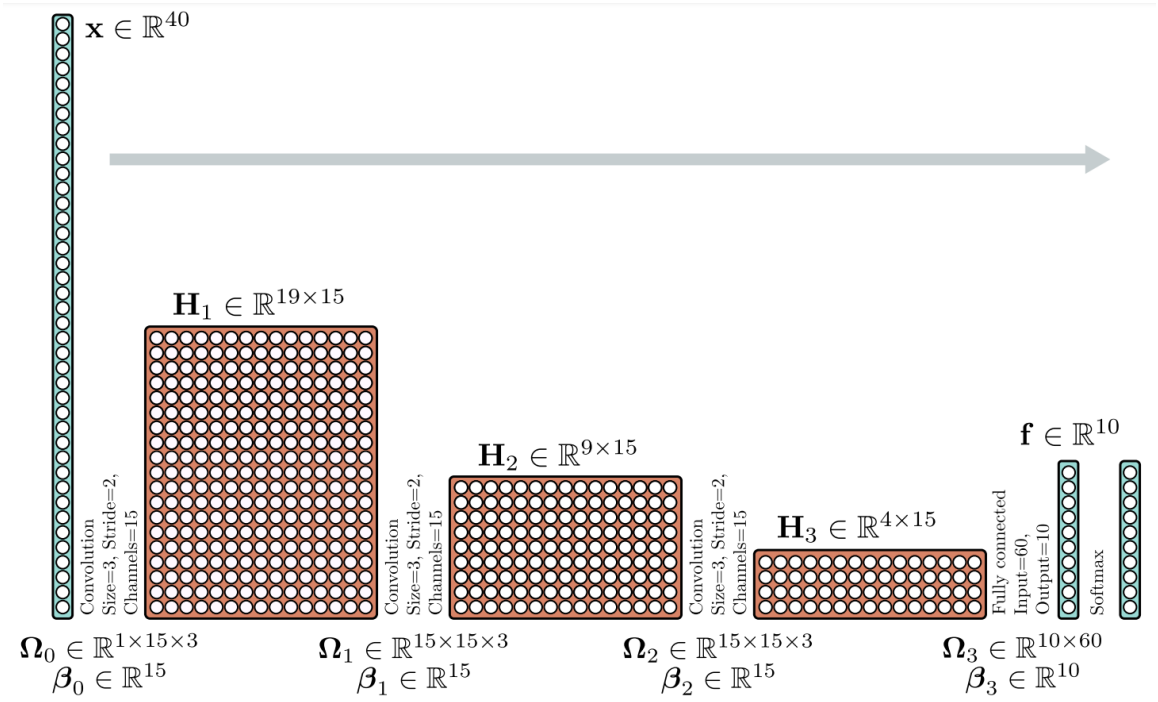
Parameter Sharing and Filters



Shared kernel weights across all inputs

- **Kernels / Filters:**
  - A kernel (or filter) is a collection of weights used to extract specific features from an input image. These features could represent edges, textures, or patterns in the image.
  - The kernel has a fixed spatial size $F \times F$ (width and height) and a depth d, which matches the number of channels in the input image (e.g., $d = 3$ for RGB images).
  - Each kernel is applied to the input image to detect a specific feature, generating an **activation map** that highlights the regions where the feature is present.
- **Parameter Sharing:**
  - The weights of a kernel are shared across all spatial regions of the input image. This means the same kernel is applied to every part of the image, allowing the network to efficiently detect the same feature regardless of its location.
  - Shared weights significantly reduce the number of parameters compared to fully connected layers, making convolutional layers computationally efficient and translationally equivariant.

Convolutional networks for 1D inputs

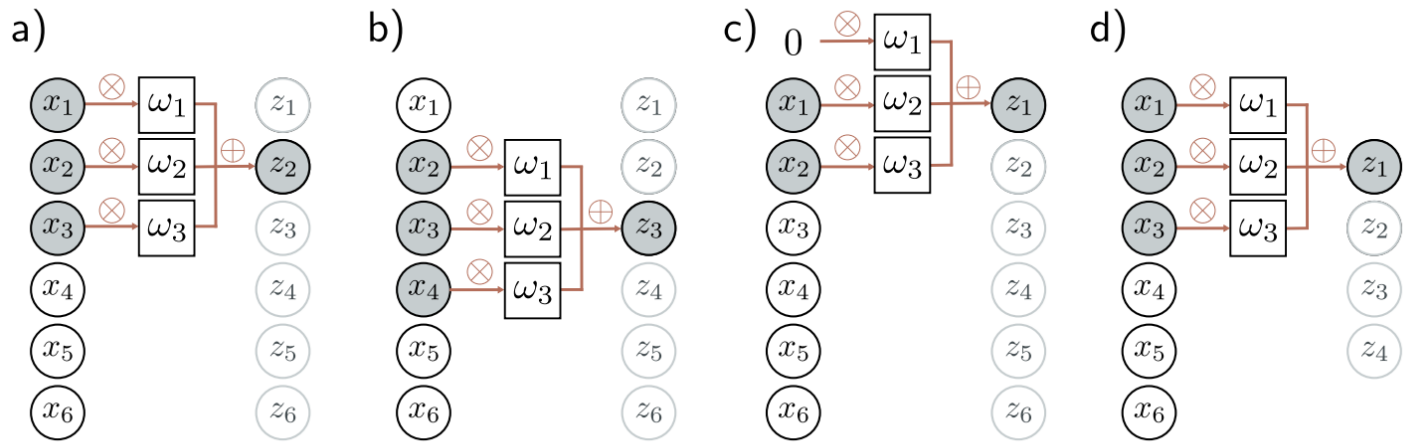


Convolutional network for classifying MNIST-1D data: The MNIST-1D input has dimension $D_i = 40$. The first convolutional layer has fifteen channels, kernel size three, stride two, and retains only "valid" positions, resulting in a hidden layer with nineteen positions and fifteen channels. The subsequent two convolutional layers use the same settings, gradually reducing the representation size at each layer. Finally, a fully connected layer takes all sixty hidden units from the third hidden layer and outputs ten activations, which are passed through a softmax layer to produce the ten class probabilities.

Convolutional layers are a type of neural network layer that leverage the convolution operation. In the 1D case, a convolution takes an input vector x and produces an output vector z, where each element z_i is computed as a weighted sum of neighboring inputs. These weights, shared across all positions, are collectively referred to as the *convolution kernel* or *filter*. The extent of the input region involved in the computation is defined by the *kernel size*.

- **Input vector $\boldsymbol{x}$:**
$$\boldsymbol{x} = [x_1, x_2, \ldots, x_I]$$
- **Output is weighted sum of neighbors:**
$$z_i = \omega_1 x_{i-1} + \omega_2 x_i + \omega_3 x_{i+1}$$
- **Convolutional kernel or filter:**
$$\boldsymbol{\omega} = [\omega_1, \omega_2, \omega_3]^T$$

  Kernel size = 3

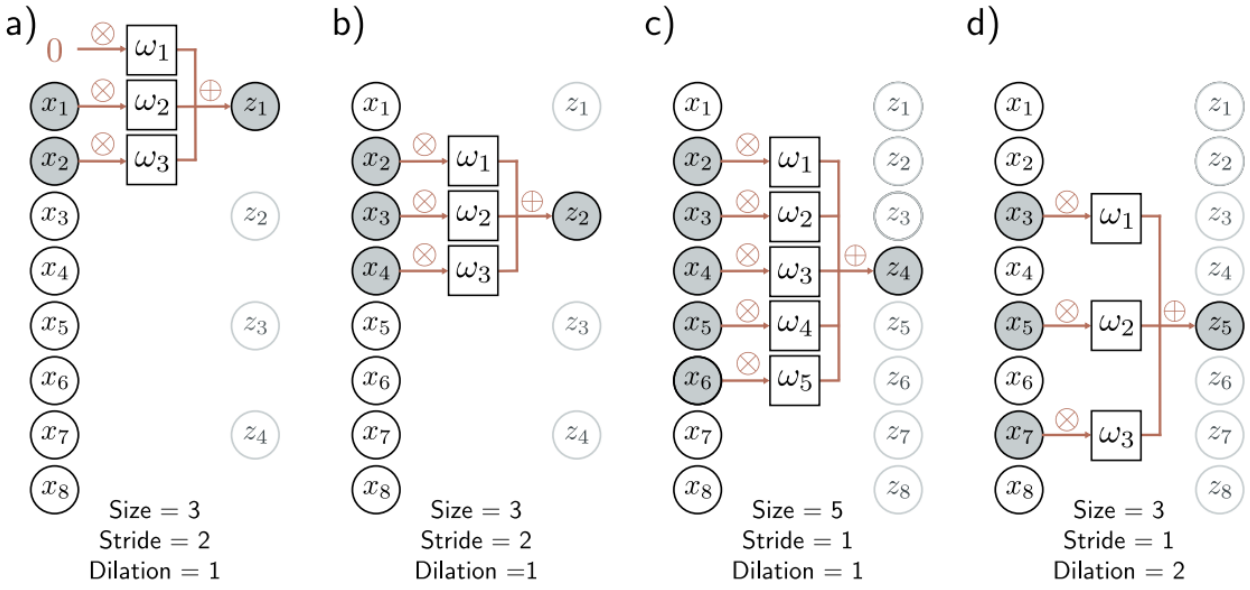


1D convolution with kernel size three. Each output z_i is a weighted sum of the nearest three inputs x_{i-1}, x_i, and x_{i+1}, where the weights are $\boldsymbol{w} = [w_1, w_2, w_3]$. a) Output z_2 is computed as $z_2 = w_1x_1 + w_2x_2 + w_3x_3$. b) Output z_3 is computed as $z_3 = w_1x_2 + w_2x_3 + w_3x_4$. c) At position z_1, the kernel extends beyond the first input x_1. This can be handled by zero-padding, in which we assume values outside the input are zero. The final output is treated similarly. d) Alternatively, we could only compute outputs where the kernel fits within the input range (valid convolution). In this case, the output will be smaller than the input.

Padding

As we saw earlier each output is computed by taking a weighted sum of the previous, current, and subsequent positions in the input. This begs the question of how to deal with the first output (where there is no previous input) and the final output (where there is no subsequent input).

There are two common approaches. The first is to pad the edges of the inputs with new values and proceed as usual. *Zero-padding* assumes the input is zero outside its valid range. Other possibilities include treating the input as circular or reflecting it at the boundaries. The second approach is to discard the output positions where the kernel exceeds the range of input positions. These *valid convolutions* have the advantage of introducing no extra information at the edges of the input. However, they have the disadvantage that the representation decreases in size.



a) With a stride of two, we evaluate the kernel at every other position, so the first output z_1 is computed from a weighted sum centered at x_1, and b) the second output z_2 is computed from a weighted sum centered at x_3, and so on. c) The kernel size can also be changed. With a kernel size of five, we take a weighted sum of the nearest five inputs. d) In *dilated or atrous convolution* (from the French "*à trous*" – with holes), we intersperse zeros in the weight vector to allow us to combine information over a large area using fewer weights.

Stride, Kernel Size, and Dilation

In the example above, each output was a sum of the nearest three inputs. However, this is just one of a larger family of convolution operations, the members of which are distinguished by their *stride*, *kernel size*, and *dilation rate*.

When we evaluate the output at every position, we term this a *stride of one*. However, it is also possible to shift the kernel by a stride greater than one. If we have a stride of two, we create roughly half the number of outputs.

The *kernel size* can be increased to integrate over a larger area. However, it typically remains an odd number so that it can be centered around the current position. Increasing the kernel size has the disadvantage of requiring more weights. This leads to the idea of *dilated or atrous* convolutions, in which the kernel values are interspersed with zeros.

For example, we can turn a kernel of size five into a dilated kernel of size three by setting the second and fourth elements to zero. We still integrate information from a larger input region but only require three weights to do this. The number of zeros we intersperse between the weights determines the *dilation rate*.

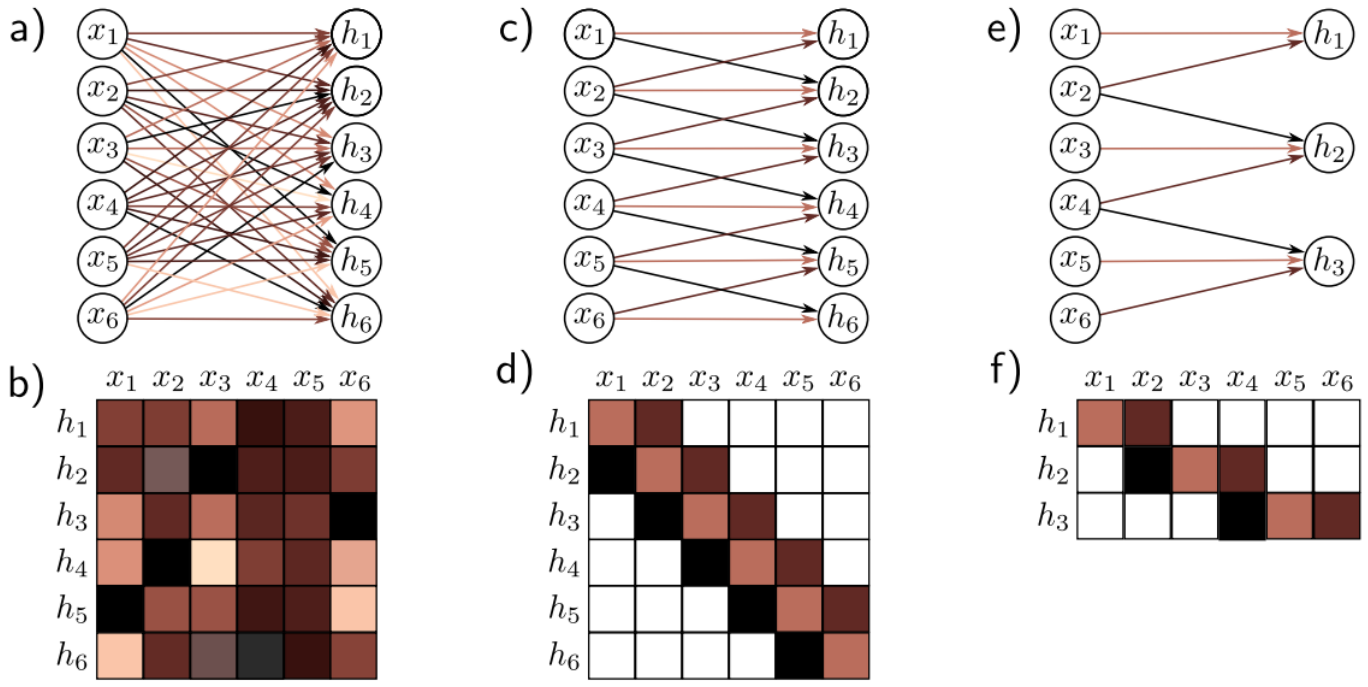
Convolutional Layers as Fully Connected Layers

Convolutional layers can be seen as a special case of fully connected layers. In a fully connected layer, the i-th hidden unit is computed as:

$$h_i = a\left(\beta_i + \sum_{j=1}^{D} \omega_{ij} x_j\right),$$

where D is the number of inputs, ω_{ij} are the weights, and β_i are the biases.

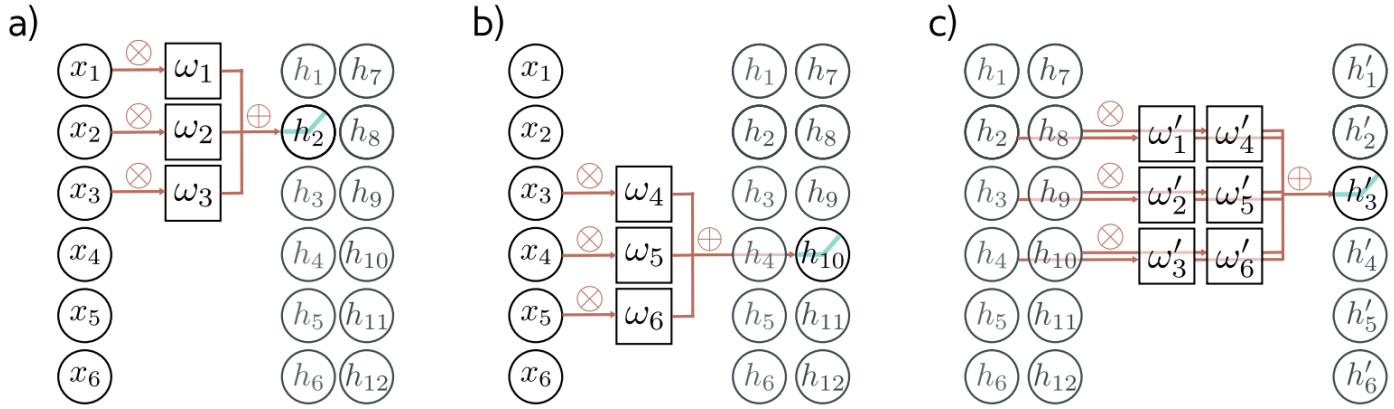
If there are D inputs and D hidden units, the fully connected layer requires D^2 weights and D biases. In contrast, a convolutional layer significantly reduces the number of parameters by using only a small number of weights (e.g., a kernel) and a single bias. A fully connected layer can replicate the behavior of a convolutional layer if most weights are set to zero and others are constrained to be identical.



a) A fully connected layer has a weight connecting each input x to each hidden unit h (colored arrows) and a bias for each hidden unit (not shown). b) Hence, the associated weight matrix $\mathbf{\Omega}$ contains 36 weights relating the six inputs to the six hidden units. c) A convolutional layer with kernel size three computes each hidden unit as the same weighted sum of the three neighboring inputs (arrows) plus a bias (not shown). d) The weight matrix is a special case of the fully connected matrix where many weights are zero and others are repeated (same colors indicate the same value, white indicates zero weight). e) A convolutional layer with kernel size three and stride two computes a weighted sum at every other position. f) This is also a special case of a fully connected network with a different sparse weight structure.

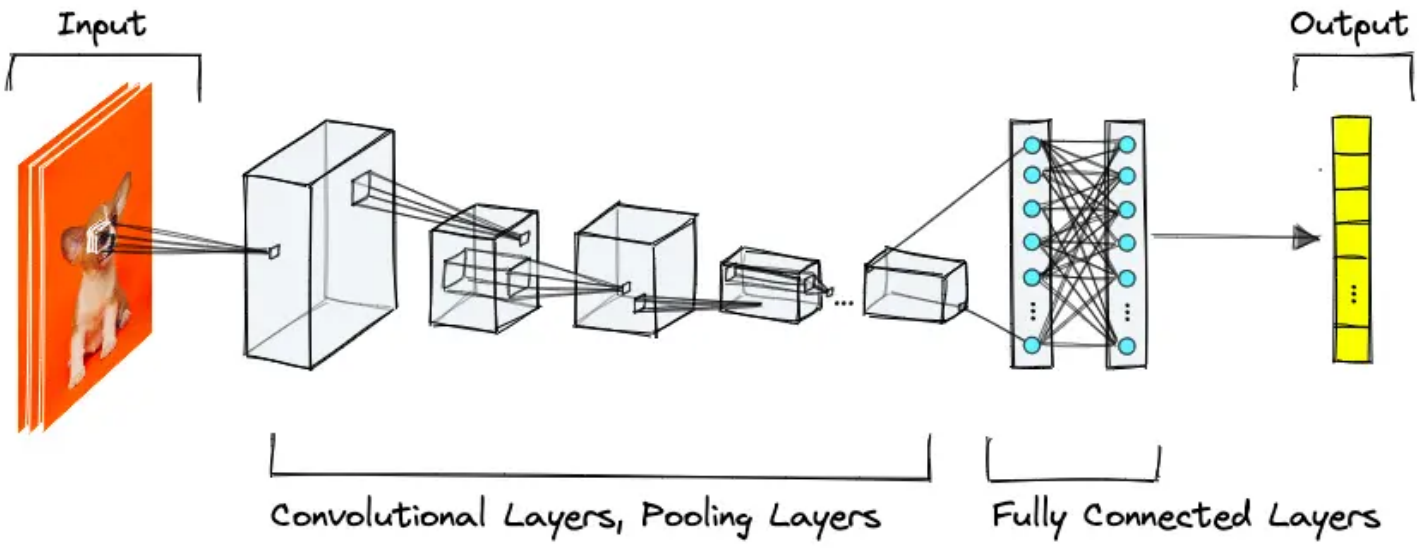
Channels

If we only apply a single convolution, information will likely be lost; we are averaging nearby inputs, and the ReLU activation function clips results that are less than zero. Hence, it is usual to compute several convolutions in parallel. Each convolution produces a new set of hidden variables, termed a feature map or channel.



Typically, multiple convolutions are applied to the input x and stored in channels. a) A convolution is applied to create hidden units h_1 to h_6, which form the first channel. b) A second convolution operation is applied to create hidden units h_7 to h_{12}, which form the second channel. The channels are stored in a 2D array H_1 that contains all the hidden units in the first hidden layer. c) If we add a further convolutional layer, there are now two channels at each input position. Here, the 1D convolution defines a weighted sum over both input channels at the three closest positions to create each new output channel.

Convolution in Multi-Dimensional Arrays



In practice, convolution is often applied to multi-dimensional arrays:

- **Input** is usually a multi-dimensional array. (e.g., a 3D image)
- **Kernel/Filter** is also typically a multi-dimensional array of parameters.

The equation for multi-dimensional convolution is:

$$S[i,j] = (I * K)(i,j) = \sum_m \sum_n I[m,n]K[i-m, j-n]$$

Where:

- I is the input array.
- K is the kernel/filter.
- $S[i,j]$ is the result of the convolution at position (i,j).
- The summation applies across dimensions m and n corresponding to the kernel and input array.

Architecture of Convolutional NNs

Convolution layers

Convolutional Neural Network (CNN) architectures typically begin with a convolutional layer, which serves as the foundation for feature extraction. This layer applies a set of learnable filters (or kernels) to the input image, performing convolution operations to detect local patterns such as edges, textures, or shapes. Each filter produces an activation map that highlights specific features, preserving spatial relationships in the data. Visualization of CNN layer

Output Spatial Size:

- For input of size $W_1 \times H_1 \times D_1$, we need 4 parameters:

  - Number of filters (K)
  - Filter size (square) F
  - Stride (S)
  - Amount of padding (P)
- The output will have size $W_2 \times H_2 \times D_2$, where:

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

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

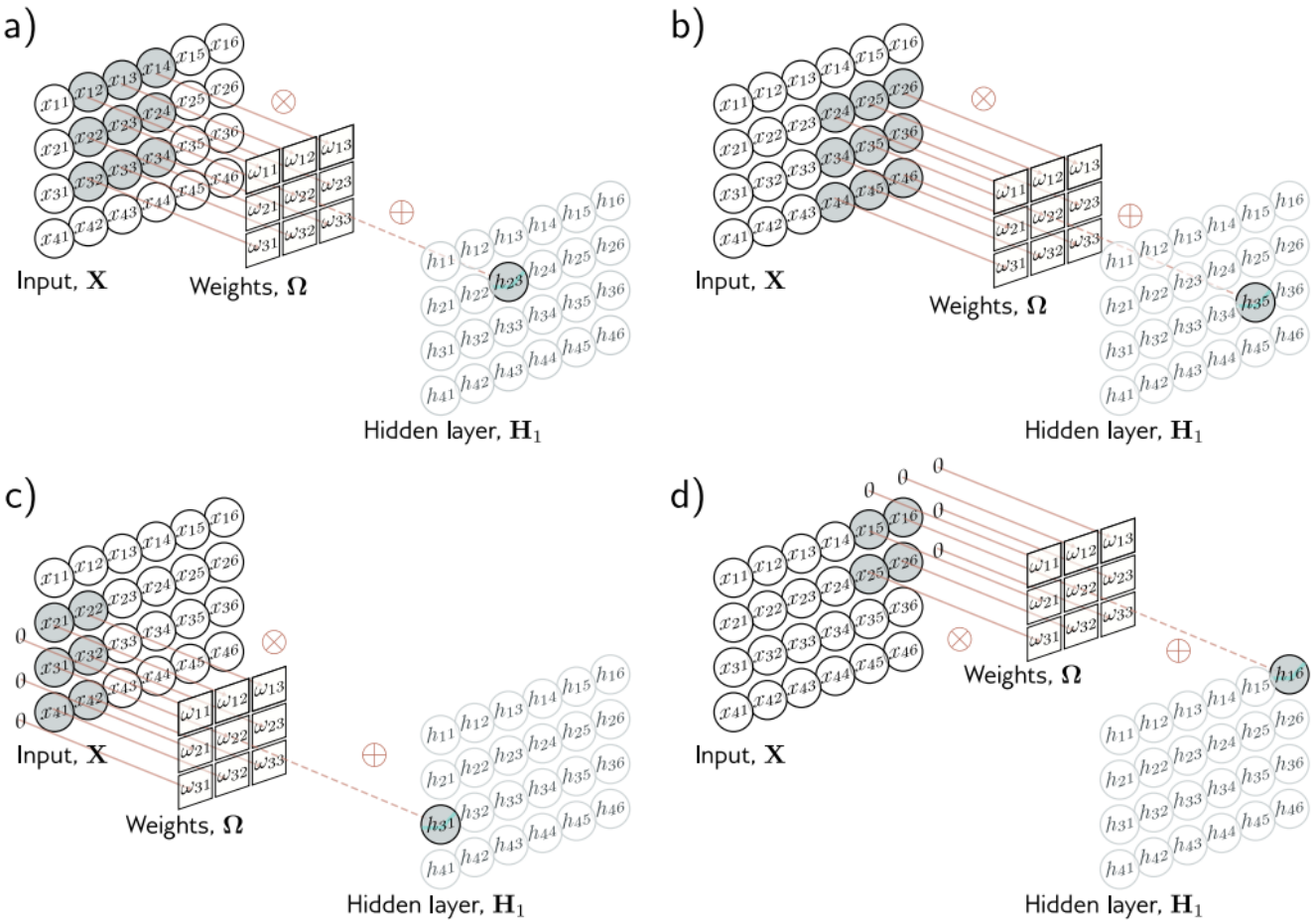
$$D_2 = K$$

- We will have $F \cdot F \cdot D_1$ parameters per filter, for a total of:

$$(F \cdot F \cdot D_1) \cdot K$$

  weights and K biases.

- **Explanation: Convolution vs. Cross-correlation**
  - The term "convolution" is technically cross-correlation because we slide the kernel over the image without inverting the kernel, which makes it cross-correlation.
  - This creates sparse connections (locality), and we are also sharing weights (kernel weights).



Each output h_{ij} computes a weighted sum of the 3 $\times$ 3 nearest inputs, adds a bias, and passes the result through an activation function. a) Here, the output h_{23} (shaded output) is a weighted sum of the nine positions from x_{12} to x_{34} (shaded inputs). b) Different outputs are computed by translating the kernel across the image grid in two dimensions. c–d) With zero-padding, positions beyond the image's edge are considered to be zero.

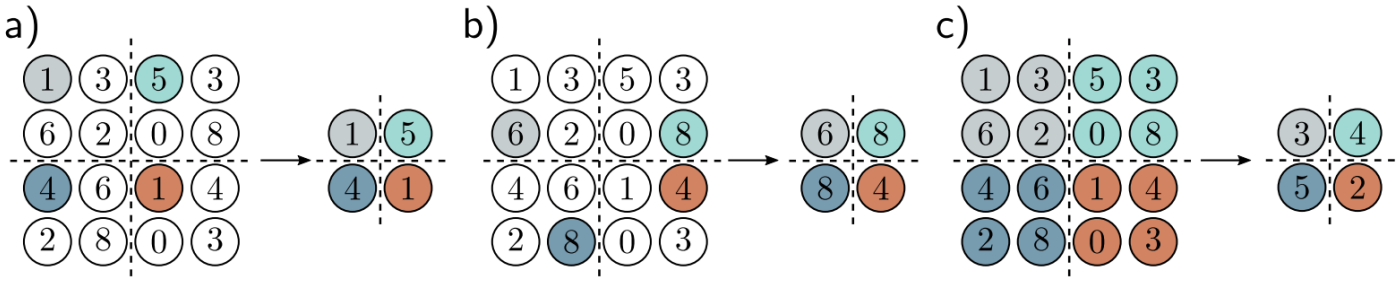
Non-Linearity

The convolutional layer is followed by a non-linear activation function, such as ReLU, to introduce non-linearity and enhance the model's representational power. This initial layer enables the network to learn low-level features, which are gradually combined into more complex patterns in deeper layers.

Downsampling and Upsampling

Downsampling

Downsampling in CNNs, often achieved through pooling (e.g., max pooling or average pooling) or strided convolutions, reduces the spatial dimensions of the feature maps. The primary reasons for downsampling are to reduce computational complexity, compress information, and make the network more efficient by focusing on the most prominent features. By decreasing the resolution, downsampling ensures the model captures larger-scale patterns and invariance to minor spatial shifts, which is particularly useful for tasks like object detection and image classification. This also helps prevent overfitting by reducing the number of parameters in subsequent layers.



a) **Sub-sampling**: The original 4 × 4 representation (left) is reduced to size 2 × 2 (right) by retaining every other input. Colors on the left indicate which inputs contribute to the outputs on the right. This is effectively what happens with a kernel of stride two, except that the intermediate values are never computed. b) **Max pooling**: Each output comprises the maximum value of the corresponding 2 × 2 block. c) **Mean pooling**: Each output is the mean of the values in the 2 × 2 block.

Pooling Layer (with Down Sampling)

- **Definition:** Pooling means returning statistics of the data rather than the data itself.
- **Common Types of Pooling:**
  - The **average** of a rectangular neighborhood.
  - The **L2 norm** of the data.
  - The **weighted average** based on distance from a central pixel.
  - **Max Pooling:** Takes the maximum value from each neighborhood of data points (visualized by max pooling of a figure in the image).
- **Purpose of Pooling Layers:**
  - **Purpose:** Pooling makes the model **invariant to small translations**.
  - Pooling layers are commonly inserted between successive CNN layers.
  - The goal is to reduce the spatial size of the representation of the data, which helps to reduce the number of parameters and avoid overfitting.
  - By applying pooling with down-sampling, you can convert different size activation maps to the same size post pooling.
- **Note:** In the pooling layer, **no parameters are learned**.

Spatial Dimensions of Output

- **Input Size:** $W_1 \times H_1 \times D_1$
- **Parameters:**
  - **Spatial Extent (F):** A square $F \times F$.
  - **Stride (S)**.
- **Output Volume Size:** $W_2 \times H_2 \times D_2$, where:

$$W_2 = \frac{(W_1 - F)}{S} + 1$$

$$H_2 = \frac{(H_1 - F)}{S} + 1$$

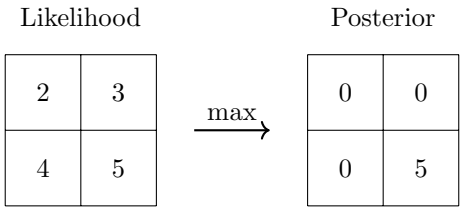
$$D_2 = D_1$$

- **Additional Information:**
  - Introduces **0 parameters**.
  - Common to use **0 padding**.

Prior Probability

(For insight, not mainly what happens in CNN)

- **Weak Prior:** A prior probability with high entropy. *Example:* Gaussian with large variance.
- **Strong Prior:** A prior probability with small entropy. *Example:* Gaussian with small variance.
- **Infinity Strong Prior:** A prior probability with 0 probability for some parameters.
- The pooling layer and convolution are considered as an infinite strong prior. It sets some values to 0 and returns a value for all of them.

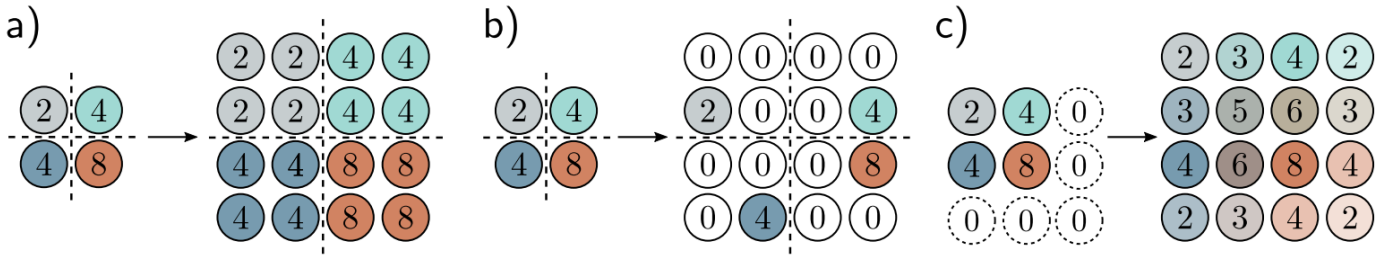


infinite strong prior

Illustration of likelihood and posterior with infinite strong prior.

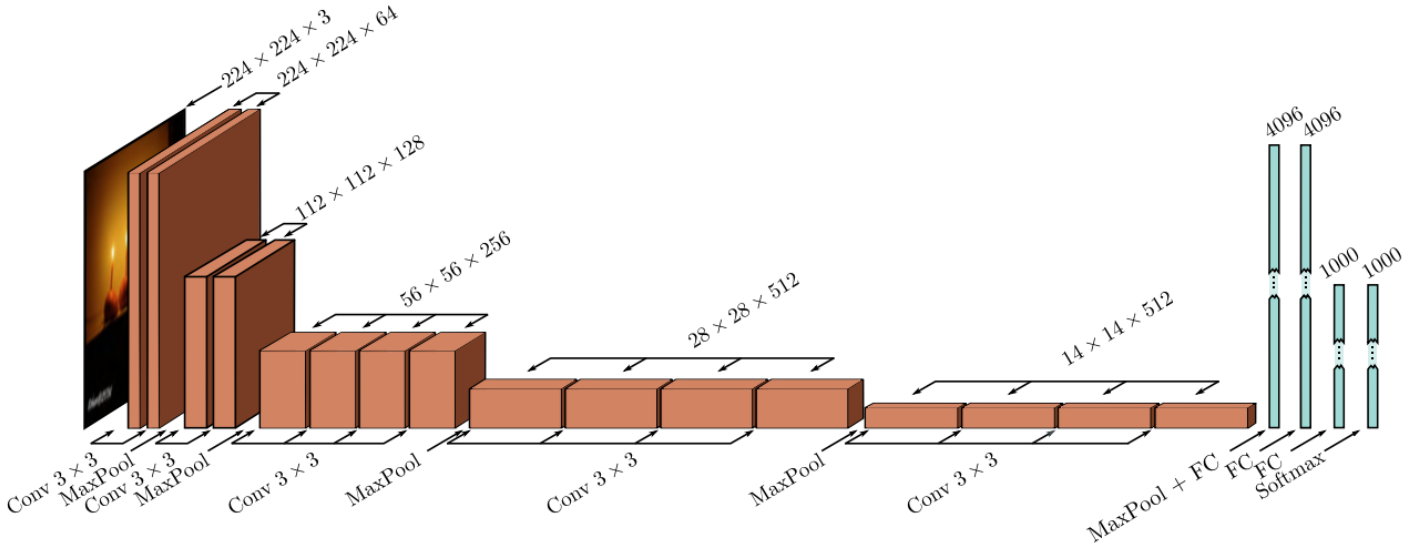
Upsampling

Upsampling in CNNs is the process of increasing the spatial dimensions of feature maps, typically used in tasks like image generation or semantic segmentation. It restores resolution lost during downsampling by methods such as duplicating values (nearest neighbor), using learned weights (transposed convolution), or interpolating between input values (e.g., bilinear or bicubic interpolation).



Fully Connected Layers

- First, we need to flatten the previous layer and then apply a fully connected feed-forward layer.
- For example, 5x5x2 layer will be a vector of size 50



The hidden layer after the last convolutional operation is resized to a 1D vector, and three fully connected layers follow. The network outputs 1000 activations corresponding to the class labels, which are passed through a softmax function to create class probabilities.

Final Layer

The final layer of a CNN is typically responsible for producing the network's predictions. In classification tasks, it is often a fully connected (dense) layer that outputs a vector where each element corresponds to the probability of a specific class. This is achieved by applying an activation function, such as softmax, to ensure the outputs represent probabilities that sum to one.

In regression tasks, the final layer might have a single unit with a linear activation to predict continuous values. The design of the last layer depends on the specific task, ensuring the outputs align with the problem's requirements, such as predicting labels, bounding boxes, or pixel-level values. Visualization of the final layer

Putting it All together

End-to-End Visualization of CNN

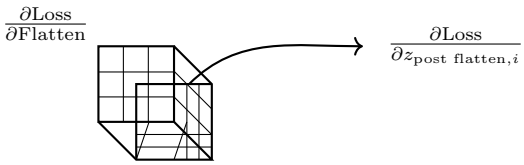
Back Propagation

- **Fully Connected Layers:** Same as feed-forward layers, find the backpropagation all the way to the input of the first fully connected layer.

$$\frac{\partial \text{Loss}}{\partial z_{\text{post fully}}} = \left| \begin{array}{c} \\ \\ \\ \end{array} \right| \; s_{\text{flatten}\times 1}$$

Gradient of Loss with respect to the fully connected layer output reshaped as a vector.

- **Flatten Layer:** Reconstruct/reshape the gradients from above into the tensor of gradients before flattening.



Backpropagation of the gradient through the flattening operation.

- **Pooling Layers:**
  - For max pooling:

$$\frac{\partial \text{Loss}}{\partial \text{Input}_{i,j}} = \begin{cases} \frac{\partial \text{Loss}}{\partial \text{Pooled Output}_{i,j}} & \text{if Input}_{i,j} \text{ was the max value} \\ 0 & \text{otherwise} \end{cases}$$

  - For average pooling:

$$\frac{\partial \text{Loss}}{\partial \text{Input}_{i,j}} = \frac{\partial \text{Loss}}{\partial \text{Pooled Output}_{i,j}} \times \frac{1}{\text{Pool Size}}$$

- **Example: Max Pooling Backpropagation**

  Let's assume our activation map is:

$$\text{Input Matrix:} \quad I = \begin{bmatrix} 1 & 2 & 3 & 0 \\ 4 & 5 & 6 & 1 \\ 7 & 8 & 9 & 2 \\ 0 & 1 & 2 & 3 \end{bmatrix}$$

  Applying max pooling with a stride $s = 2$:

$$\Rightarrow \begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix}$$

  Let's assume **gradient of loss** with respect to the output of the pooling layer is:

$$\text{Gradient Loss} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix}$$

  Backpropagating through max pooling, we get the gradient with respect to the input of the pooling layer:

$$\frac{\partial \text{Loss}}{\partial \text{Input of Pooling Layer}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0.1 & 0.2 & 0 \\ 0 & 0.3 & 0.4 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Only the max value from the original input has a gradient, which is because the max/min operation is linear.

- **Example: Max Pooling Backpropagation**

  Let's assume our activation map is:

$$\text{Input Matrix:} \quad I = \begin{bmatrix} 1 & 2 & 3 & 0 \\ 4 & 5 & 4 & 1 \\ 7 & 8 & 9 & 2 \\ 0 & 1 & 0 & 3 \end{bmatrix}$$

  Applying max pooling with a stride $s = 1$:

$$\Rightarrow \begin{bmatrix} 5 & 5 & 4 \\ 8 & 9 & 9 \\ 8 & 9 & 9 \end{bmatrix}$$

  Let's assume **gradient of loss** with respect to the output of the pooling layer is:

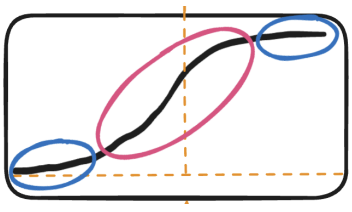
$$\text{Gradient Loss} = \begin{bmatrix} 0.1 & 0.2 & 0.3 \\ 0.4 & 0.5 & 0.6 \\ 0.7 & 0.8 & 0.9 \end{bmatrix}$$

  Backpropagating through max pooling, we get the gradient with respect to the input of the pooling layer:

$$\frac{\partial \text{Loss}}{\partial \text{Input of Pooling}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0.3 & 0.3 & 0 \\ 0 & 1.1 & 2 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

  Note: The gradient values 0.5 and 0.6 correspond to the positions of 9 in the input, and they sum up because the max value 9 was selected from overlapping regions during pooling.

Detector Stage / Activation Function



Sigmoid:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

The derivative of the sigmoid function is important for backpropagation in neural networks:

$$\frac{\partial}{\partial x}\text{sigmoid}(x) = \sigma(x)(1 - \sigma(x))$$

where $\sigma(x)$ is the sigmoid function itself. This derivative indicates how the output of the sigmoid function changes with respect to changes in its input x.

Convolutional Layers

Backward Pass:

- **Gradient w.r.t. Input:** To find the gradient with respect to the input of the convolution layer, you distribute the gradient of the output feature map back through the convolution operation:

$$\frac{\partial \text{Loss}}{\partial \text{Input}_{i,j}} = \sum_{m,n} \frac{\partial \text{Loss}}{\partial \text{Feature Map}_{i+m,j+n}} \cdot \text{Filter}_{m,n}$$

- **Gradient w.r.t. Filter:** To find the gradient with respect to the filter weights, you calculate:

$$\frac{\partial \text{Loss}}{\partial \text{Filter}_{m,n}} = \sum_{i,j} \left(\frac{\partial \text{Loss}}{\partial \text{Feature Map}_{i,j}} \cdot \text{Input}_{i+m,j+n} \right)$$

Example: Convolution Operation

Input Activations:

$$\text{Input channels } C = 1, \text{ number of images } N = 1, \text{ Image height } H = 5, \text{ width } W = 5$$

$$\text{Input Matrix } I = \begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{03} & x_{04} \\ x_{10} & x_{11} & x_{12} & x_{13} & x_{14} \\ x_{20} & x_{21} & x_{22} & x_{23} & x_{24} \\ x_{30} & x_{31} & x_{32} & x_{33} & x_{34} \\ x_{40} & x_{41} & x_{42} & x_{43} & x_{44} \end{bmatrix}$$

Filter (aka Kernel):

$$\text{Filter height } R = 3, \text{ width } S = 3, \quad \text{stride}_R = \text{stride}_S = 2$$

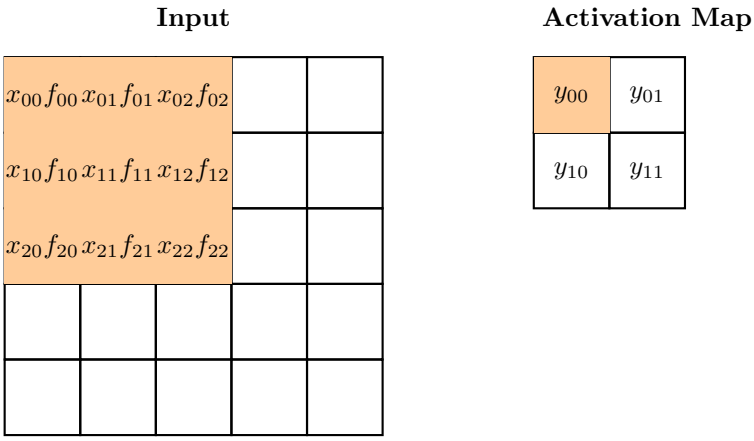
$$\text{Filter Matrix } F = \begin{bmatrix} f_{00} & f_{01} & f_{02} \\ f_{10} & f_{11} & f_{12} \\ f_{20} & f_{21} & f_{22} \end{bmatrix}$$

Output:

$$\text{Output channels } K = 1, \text{ number of outputs } N = 1, \text{ Output height } H = 2, \text{ width } W = 2$$

$$\text{Output Matrix } Y = \begin{bmatrix} y_{00} & y_{01} \\ y_{10} & y_{11} \end{bmatrix}$$

Convolutional Backpropagation



Input region contributing to y_{00} and the resulting activation map.

Calculation of y_{00}:

$$y_{00} = x_{00}f_{00} + x_{01}f_{01} + x_{02}f_{02} + x_{10}f_{10} + x_{11}f_{11} + x_{12}f_{12} + x_{20}f_{20} + x_{21}f_{21} + x_{22}f_{22}$$

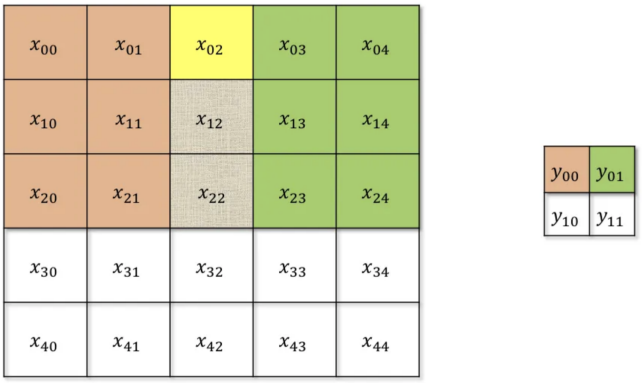
Gradient Calculation:
To compute the gradient of the loss with respect to an input element x_{00}, we use the chain rule:

$$\Rightarrow \frac{\partial \text{Loss}}{\partial x_{00}} = \frac{\partial \text{Loss}}{\partial y_{00}} \cdot \frac{\partial y_{00}}{\partial x_{00}} = \frac{\partial \text{Loss}}{\partial y_{00}} \cdot f_{00}$$

Explanation: This equation indicates that the gradient of the loss with respect to x_{00} is influenced by the gradient of the loss with respect to y_{00} and the filter value f_{00}. This relationship arises because y_{00} is computed using x_{00} and the filter value f_{00}.

Now, for Pixels that Overlap, we calculate the gradient for them as:

$$\frac{\partial \text{Loss}}{\partial x_{mn}} = \sum_{i,j} \frac{\partial \text{Loss}}{\partial y_{ij}} \cdot \frac{\partial y_{ij}}{\partial x_{mn}}$$



Gradient of x_{02}:
Next, consider x_{02}. It contributes to y_{00} and y_{01}.

$$y_{00} = x_{00}f_{00} + x_{01}f_{01} + x_{02}f_{02} + x_{10}f_{10} + x_{11}f_{11} + x_{12}f_{12} + x_{20}f_{20} + x_{21}f_{21} + x_{22}f_{22}$$

$$y_{01} = x_{02}f_{00} + x_{03}f_{01} + x_{04}f_{02} + x_{12}f_{10} + x_{13}f_{11} + x_{14}f_{12} + x_{22}f_{20} + x_{23}f_{21} + x_{24}f_{22}$$

Thus,

$$\frac{\partial \mathcal{L}}{\partial x_{02}} = \frac{\partial \mathcal{L}}{\partial y_{00}} f_{02} + \frac{\partial \mathcal{L}}{\partial y_{01}} f_{00}$$

Explanation: For overlapping regions in the input, such as x_{02}, which affects multiple outputs (e.g., y_{00} and y_{01}), the gradient accumulates contributions from each path. This is why the gradient is a sum of terms involving the partial derivatives of each affected output.

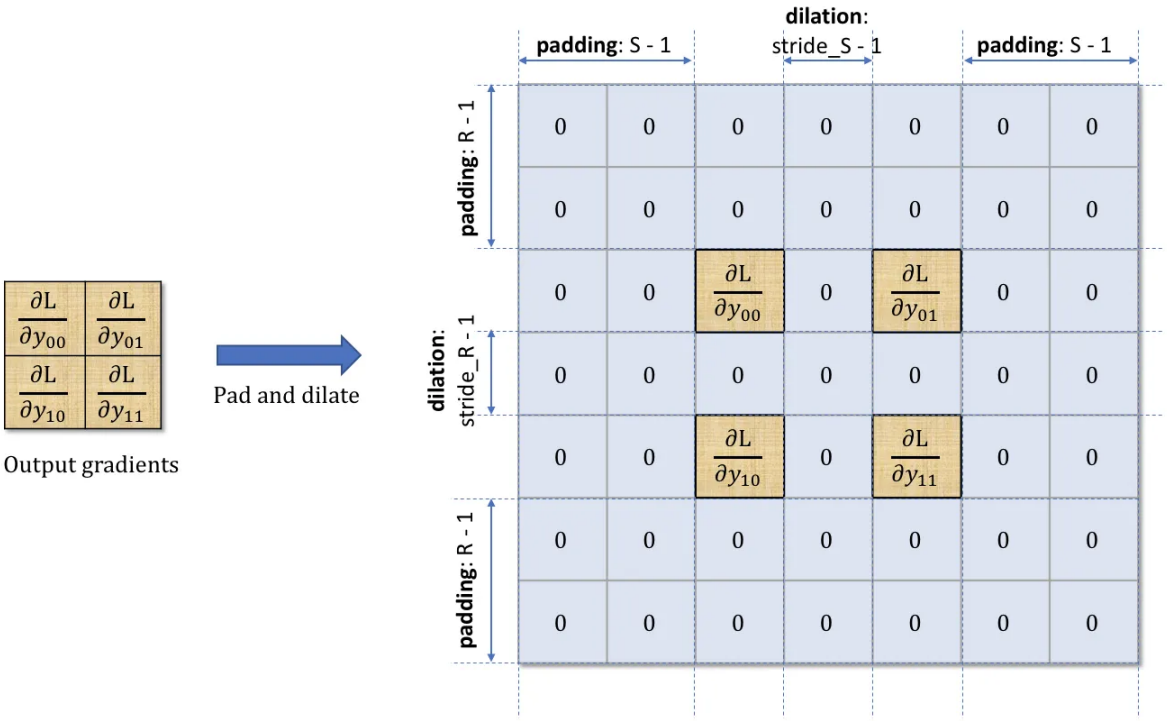
We can also calculate the gradient by applying padding and dilation to the output gradients, ensuring alignment with the convolution operation's structure.

Explanation: We pad and dilate the output gradients to match the input dimensions during backpropagation through a convolutional layer.

- **Padding:** Add $P = F - 1$ (half on all sides).

- **Dilation:** Introduce $D = S - 1$ zeros between elements of the gradient tensor.

These adjustments ensure the backpropagation aligns with the original convolution operation. If your stride S is 1, dilation isn't necessary ($D = 0$).



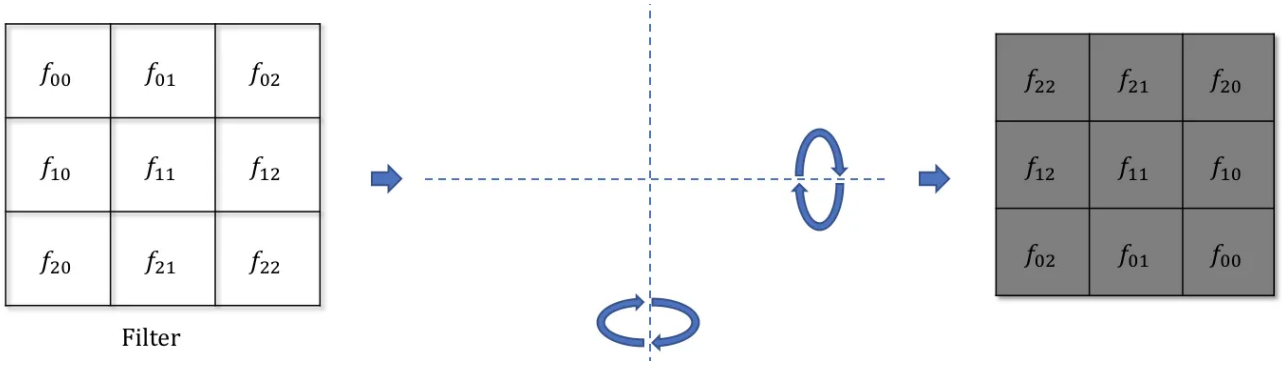
Padded and dilated output gradients for backpropagation.

Details:

- Padding: $R - 1$ rows and columns of zeros are added around the original gradients.
- Dilation: Stride of $S - 1$ is applied between the elements of the output gradient.

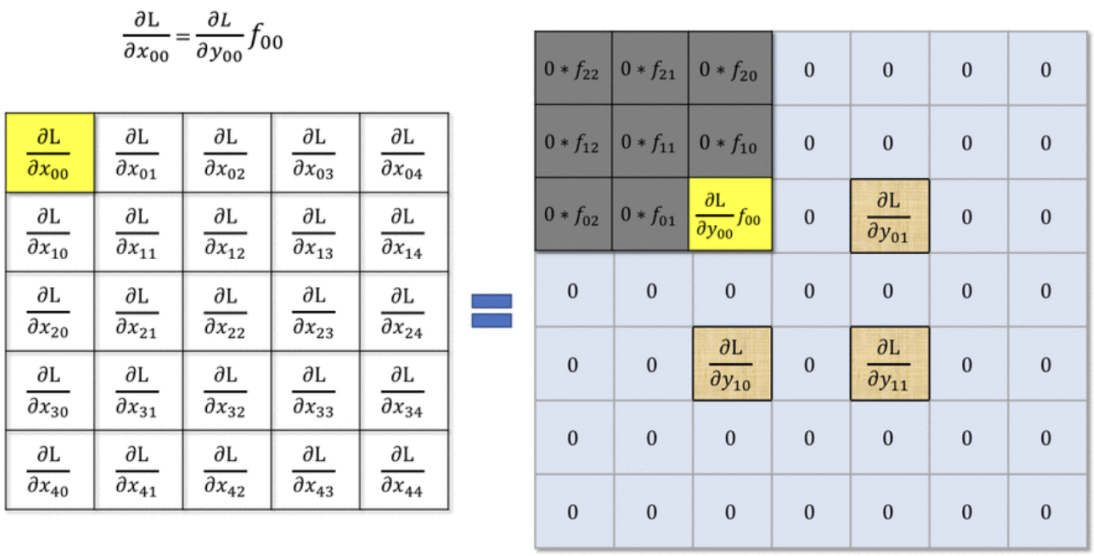
$$R = S = 3 \quad \text{(our original filter size above) and } Stride_S = 2$$

Here, we modified our gradient with the same stride.
Next, we will modify our kernel by flipping it horizontally & vertically:

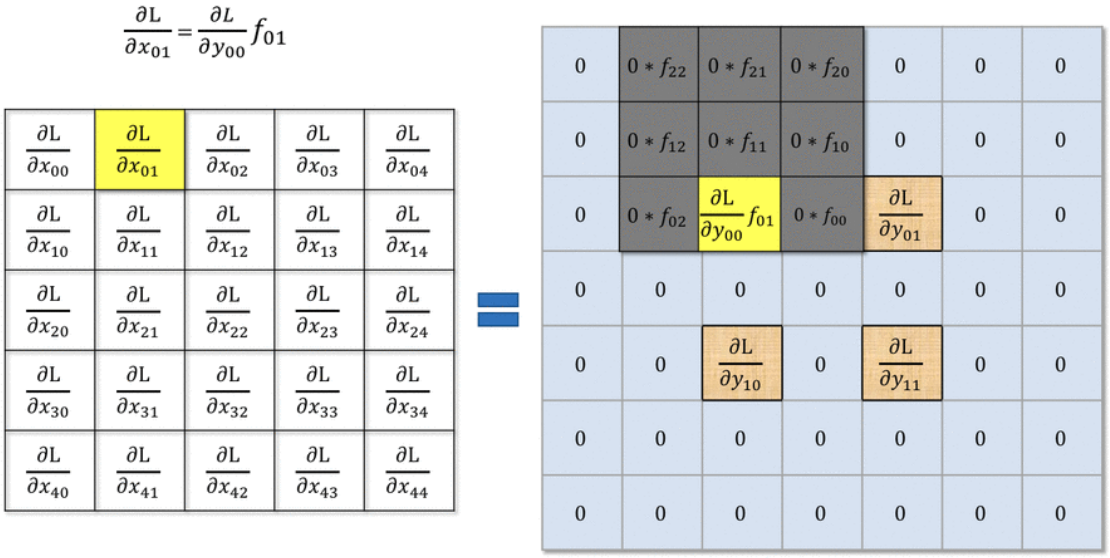


Flipping the filter horizontally and vertically.

The Gradient



Gradient calculation using a padded, dilated version of the output gradient tensor with a flipped filter.



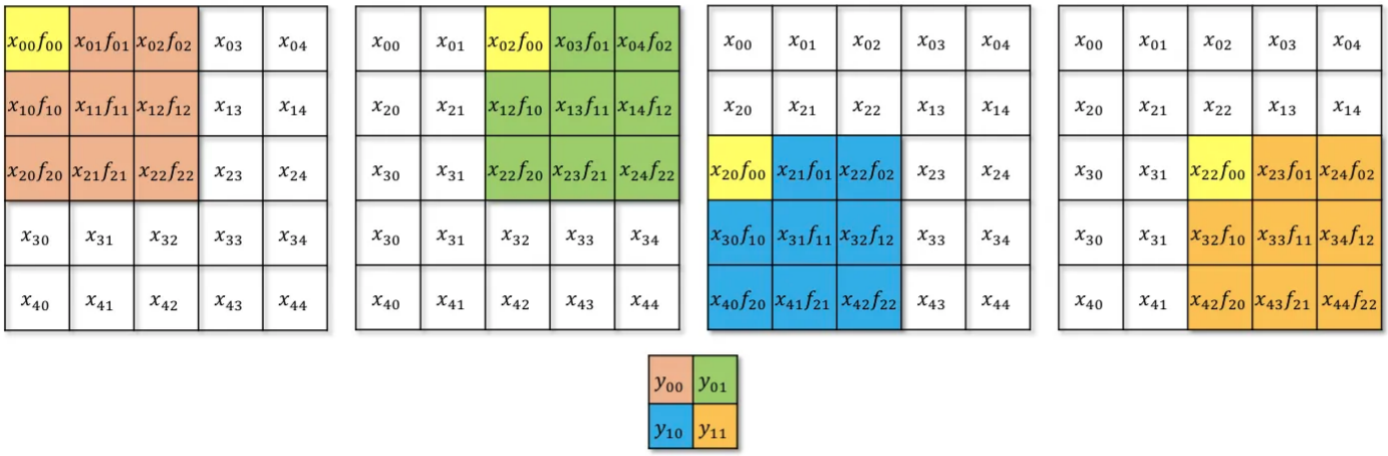
Gradient calculation using a padded, dilated version of the output gradient tensor with a flipped filter.

Explanation: It turns out that the backpropagation operation is identical to a stride = 1 convolution of a padded, dilated version of the output gradient tensor with a flipped version of the filter! Visualization of the gradient calculation

Kernel Weights

We need to calculate the gradient of the loss with respect to the filter pixels (aka the weight tensor).

$$\frac{\partial \mathcal{L}}{\partial f_{mn}} = \sum_{ij} \frac{\partial \mathcal{L}}{\partial y_{ij}} \cdot \frac{\partial y_{ij}}{\partial f_{mn}}$$



First, consider f_{00}. It contributes to all outputs: y_{00}, y_{01}, y_{10}, and y_{11}.

$$y_{00} = x_{00}f_{00} + x_{01}f_{01} + x_{02}f_{02} + x_{10}f_{10} + x_{11}f_{11} + x_{12}f_{12} + x_{20}f_{20} + x_{21}f_{21} + x_{22}f_{22}$$

$$y_{01} = x_{02}f_{00} + x_{03}f_{01} + x_{04}f_{02} + x_{12}f_{10} + x_{13}f_{11} + x_{14}f_{12} + x_{22}f_{20} + x_{23}f_{21} + x_{24}f_{22}$$

$$y_{10} = x_{20}f_{00} + x_{21}f_{01} + x_{22}f_{02} + x_{30}f_{10} + x_{31}f_{11} + x_{32}f_{12} + x_{40}f_{20} + x_{41}f_{21} + x_{42}f_{22}$$

$$y_{11} = x_{22}f_{00} + x_{23}f_{01} + x_{24}f_{02} + x_{32}f_{10} + x_{33}f_{11} + x_{34}f_{12} + x_{42}f_{20} + x_{43}f_{21} + x_{44}f_{22}$$

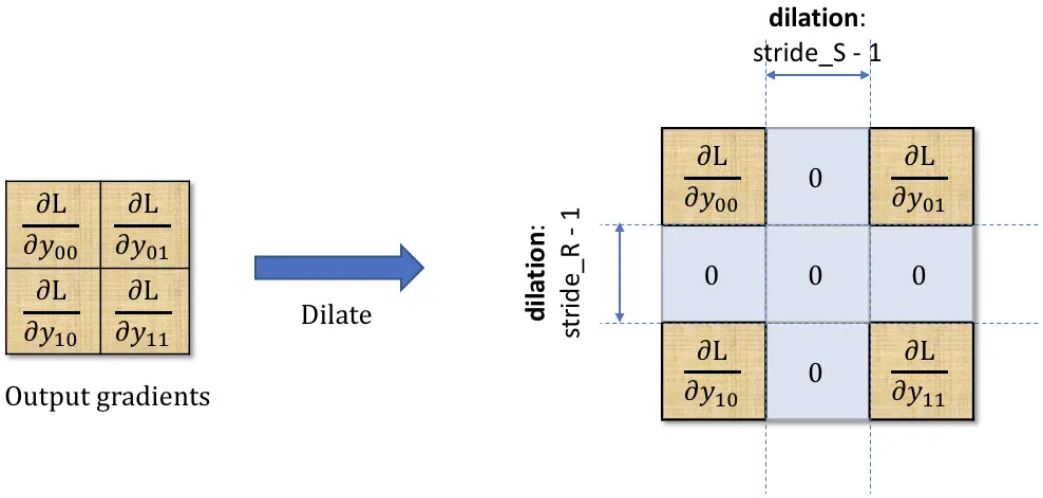
The gradient of the loss with respect to f_{mn} is:

$$\frac{\partial \mathcal{L}}{\partial f_{mn}} = \sum_{ij} \frac{\partial \mathcal{L}}{\partial y_{ij}} \frac{\partial y_{ij}}{\partial f_{mn}}$$

Thus, for f_{00}:

$$\frac{\partial \mathcal{L}}{\partial f_{00}} = \frac{\partial \mathcal{L}}{\partial y_{00}} x_{00} + \frac{\partial \mathcal{L}}{\partial y_{01}} x_{02} + \frac{\partial \mathcal{L}}{\partial y_{10}} x_{20} + \frac{\partial \mathcal{L}}{\partial y_{11}} x_{22}$$

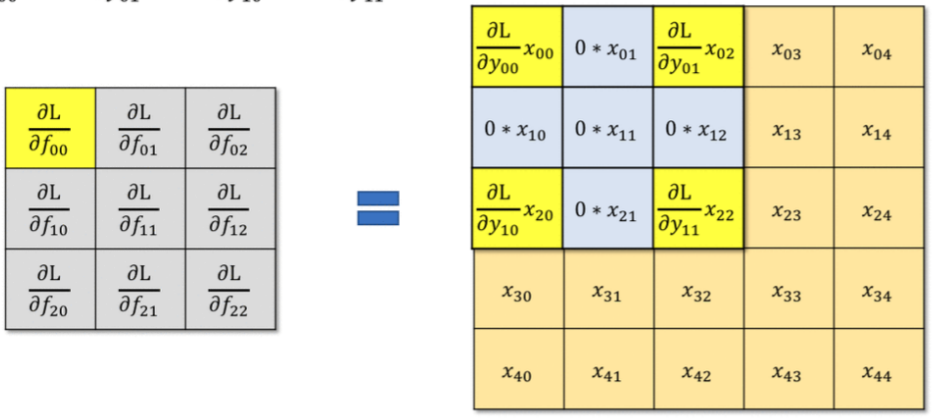
We can also calculate the gradient by pplying padding and dilation the output gradients to match the filter dimensions during backpropagation through a convolutional layer.



$$Stride_R = Stride_S = 2 \quad \text{(our original stride above)}$$

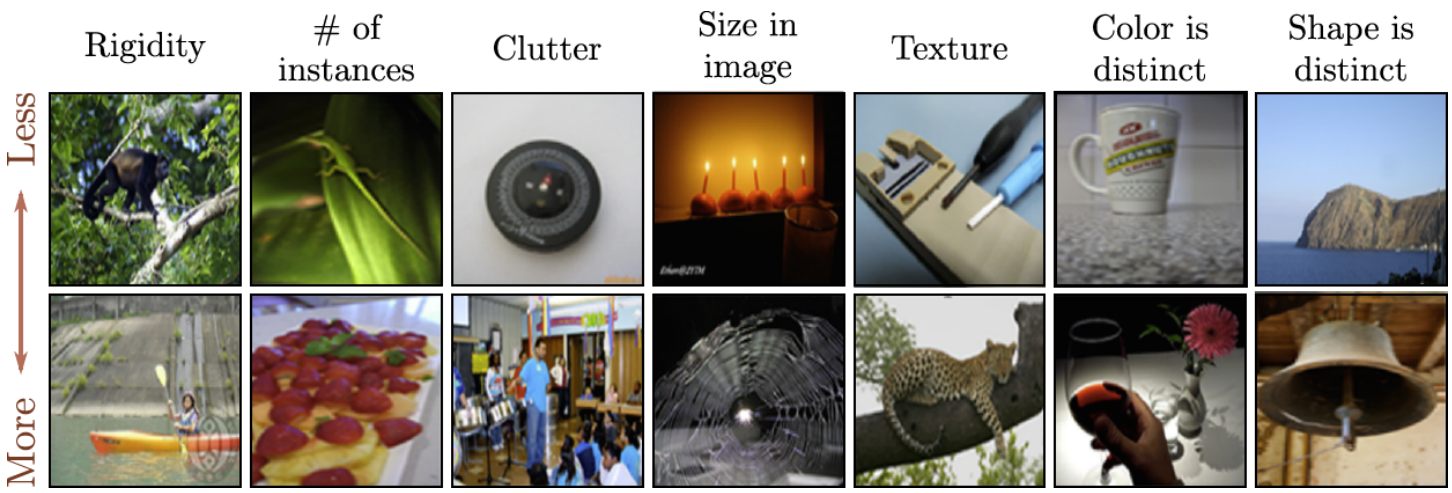
With this modification to the output gradient tensor, we'll see that the values for the filter gradients we calculated in Examples 1 to 4 above fit into a nice pattern:

$$\frac{\partial L}{\partial f_{00}} = \frac{\partial L}{\partial y_{00}} x_{00} + \frac{\partial L}{\partial y_{01}} x_{02} + \frac{\partial L}{\partial y_{10}} x_{20} + \frac{\partial L}{\partial y_{11}} x_{22}$$

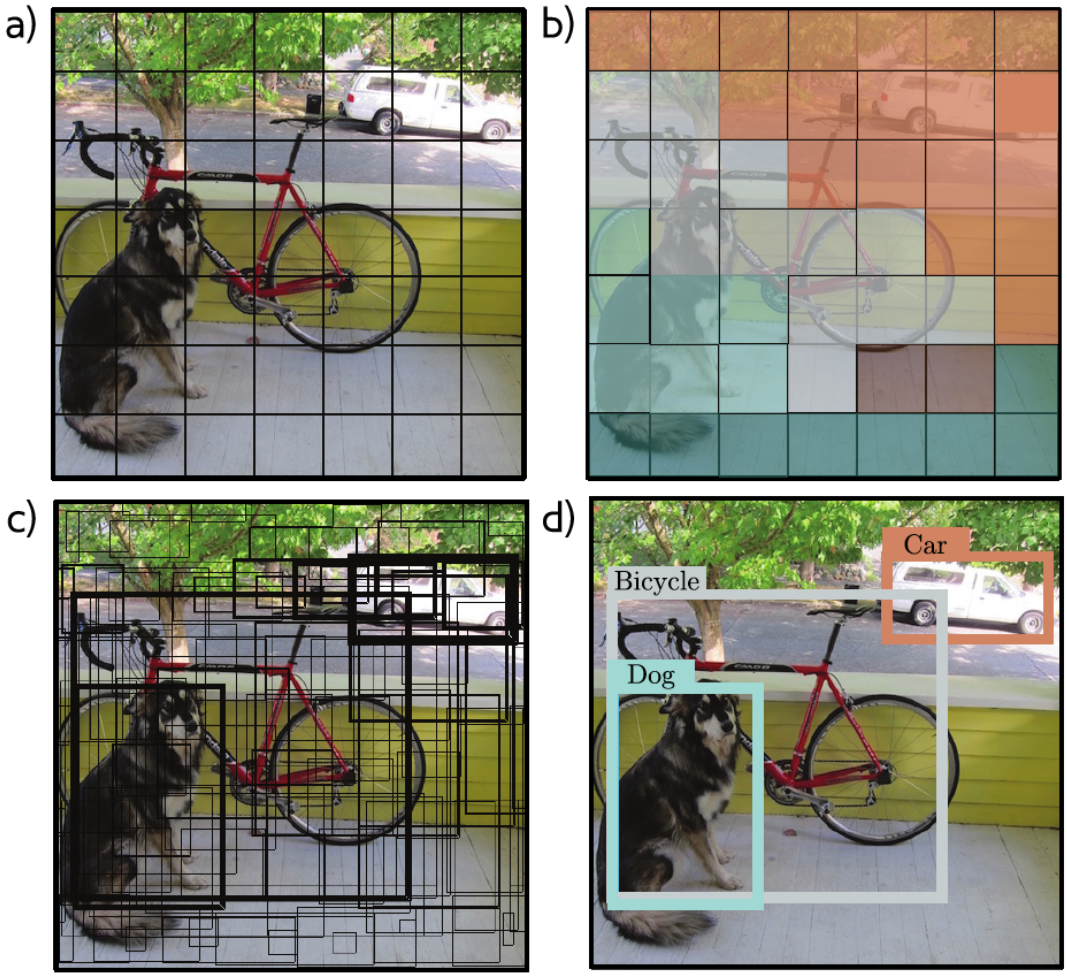


Applications of CNN

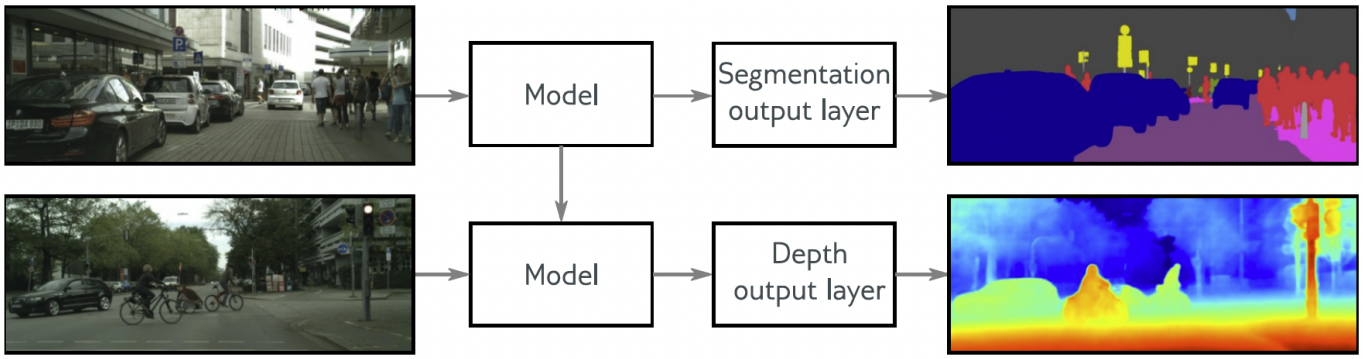
Image Classification



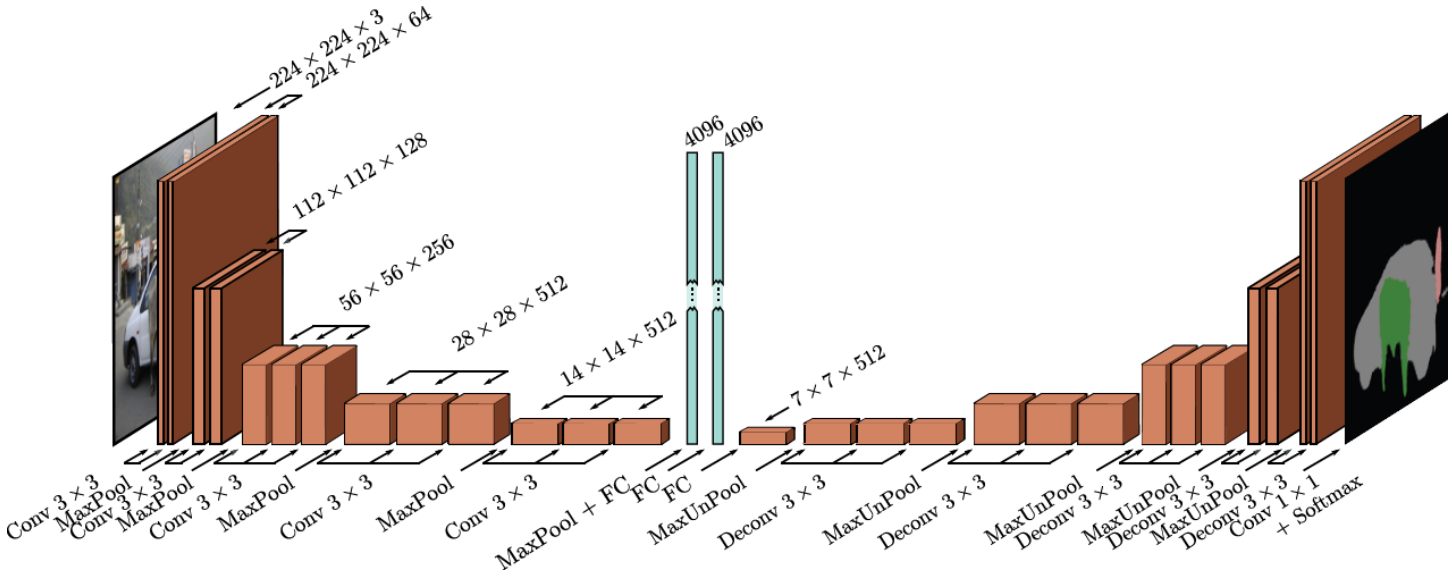
Object detection

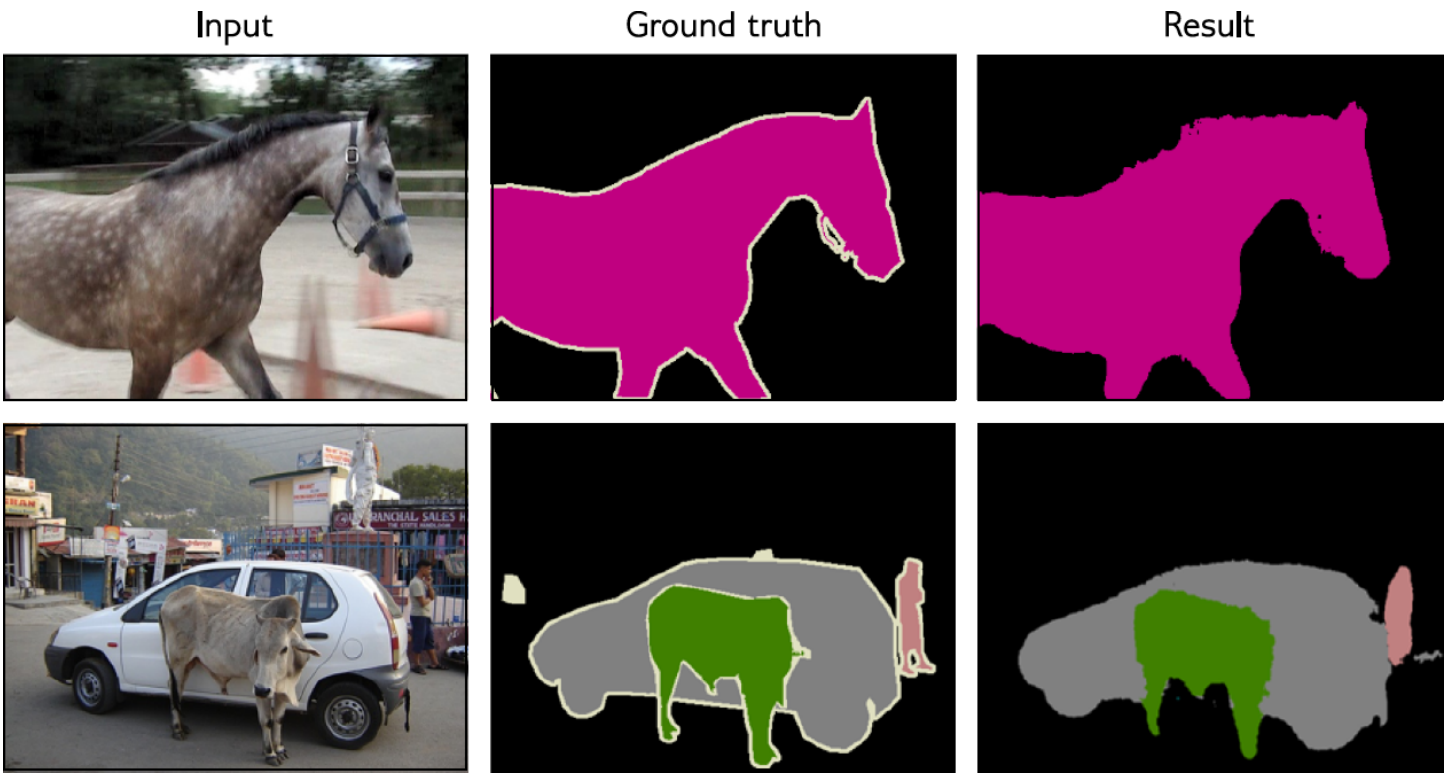


Transfer Learning



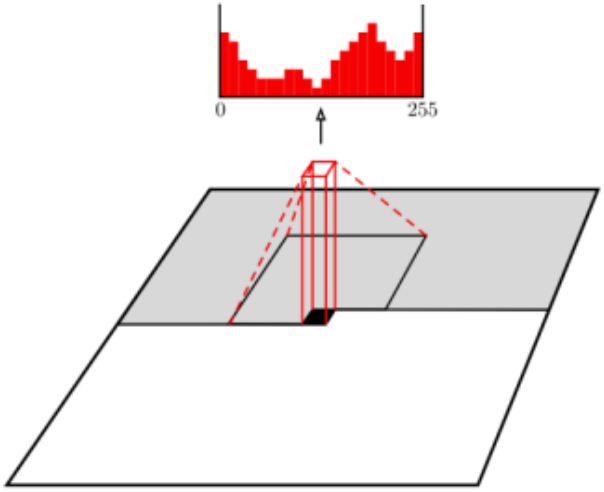
Semantic segmentation





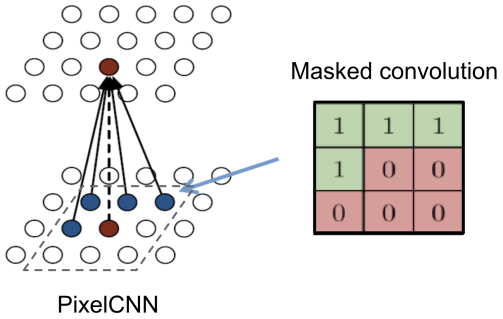
PixelCNN

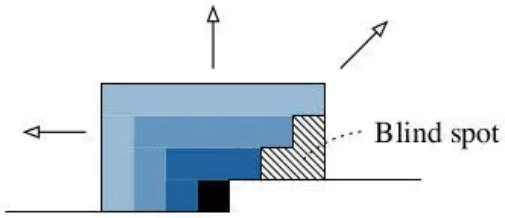
This model was developed in industry - Google

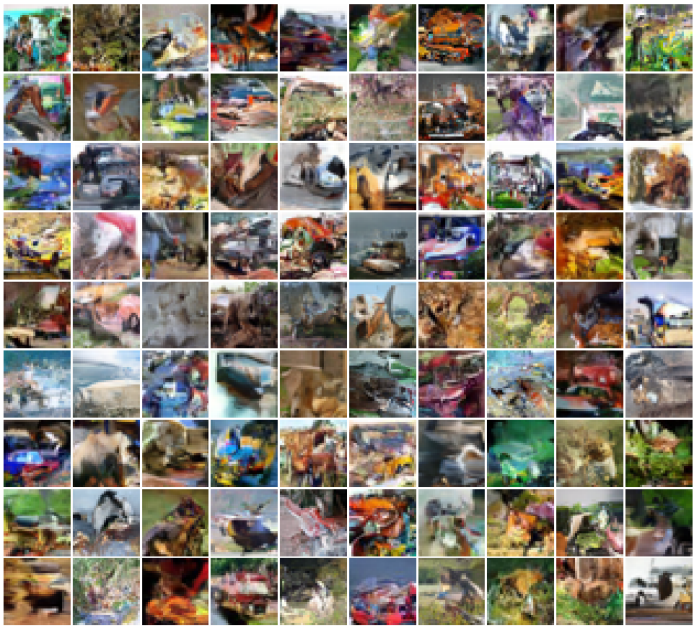


Idea and Challenge

- **Idea:** Use convolutional architecture to predict the next pixel given context (a neighborhood of pixels).
- You need to ensure that for predicting the value of a new pixel model is only looks at the shaded area (gray) and can not look at the future pixels.
- **Challenge:** The model has to be autoregressive. Masked convolutions preserve raster scan order, and additional masking is required for color order. This means to add 0 for weights corresponding to the future pixels.

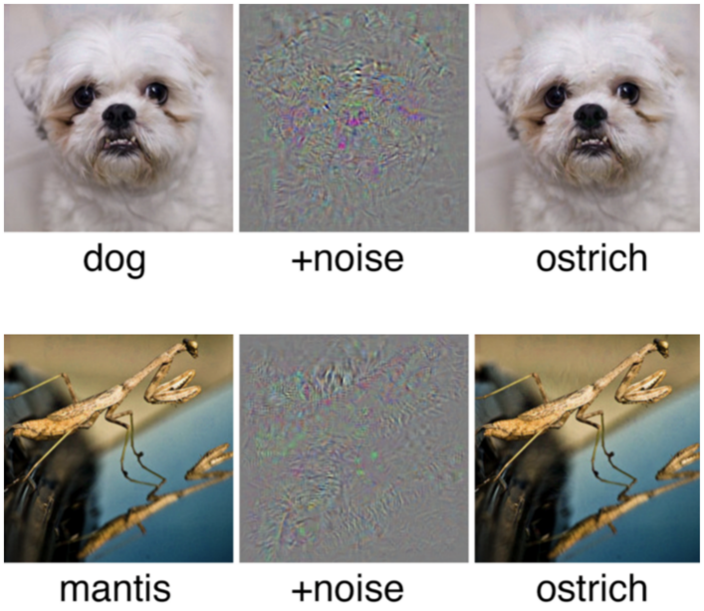






Samples from the model trained on Imagenet (32 × 32 pixels). Similar performance to PixelRNN, but much faster.

Adversarial Attacks and Anomaly detection



Machine learning methods are vulnerable to adversarial examples, Can we detect them?

- One approach was **Pixel Defend**

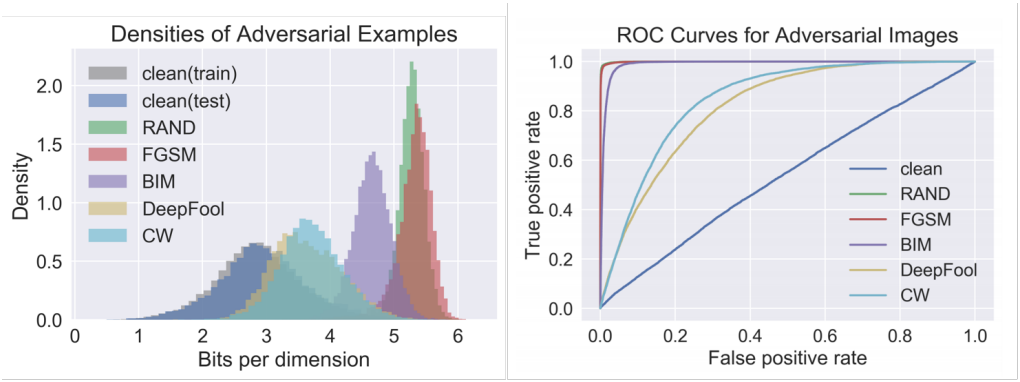


***Figure**: (Left) Likelihoods of different adversarial examples. (Right) ROC curves for detecting various attacks.*

- Train a generative model $p(x)$ on clean inputs (PixelCNN).
- Given a new input x, evaluate $p(x)$.
- Adversarial examples are significantly less likely under $p(x)$.

Residual Neural Networks

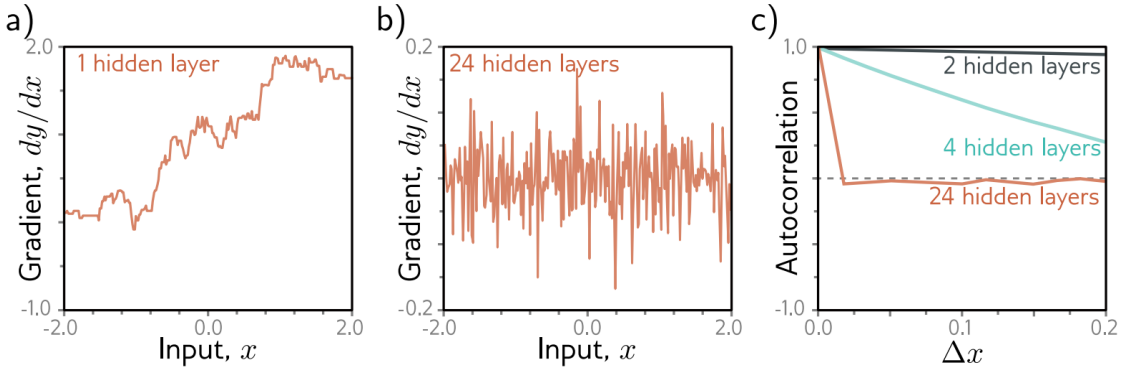
Origin and Motivation:

Residual Networks, or ResNets, were introduced by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun in their groundbreaking paper "Deep Residual Learning for Image Recognition," which was presented at the 2015 Conference on Computer Vision and Pattern Recognition (CVPR). ResNets were developed to address a key challenge in deep learning,

particularly with very deep neural networks: the **vanishing gradient problem** and the resulting degradation of model performance.

The Problem:

- **Performance Trends:**
  - Adding more layers to convolutional networks generally improves performance (e.g., VGG with 19 layers outperforms AlexNet with 8 layers).
  - However, classification performance decreases when too many layers are added, for both training and test sets, indicating a training issue rather than a generalization problem.
- **Gradient Challenges in Deep Networks:**
  - At initialization, loss gradients can change unpredictably when modifying parameters in early layers.
  - Proper weight initialization can mitigate exploding or vanishing gradients, but optimization still struggles due to finite step sizes.
  - The loss surface may resemble tiny mountains, leading to unpredictable gradient changes and slow optimization progress.
- **Shattered Gradients Phenomenon:**
  - Gradients behave differently in shallow versus deep networks:
    - In shallow networks, gradients change smoothly with input changes.
    - In deep networks, tiny input changes lead to completely different gradients.
  - This behavior is captured by the autocorrelation function of the gradient:
    - Gradients in shallow networks are correlated for nearby inputs.
    - In deep networks, gradient correlation drops quickly to zero.



a) Consider a shallow network with 200 hidden units and Glorot initialization (He initialization without the factor of two) for both the weights and biases. The gradient $\partial y/\partial x$ of the scalar network output y with respect to the scalar input x changes relatively slowly as we change the input x. b) For a deep network with 24 layers and 200 hidden units per layer, this gradient changes very quickly and unpredictably. c) The autocorrelation function of the gradient shows that nearby gradients become unrelated (have autocorrelation close to zero) for deep networks. This shattered gradients phenomenon may explain why it is hard to train deep networks. Gradient descent algorithms rely on the loss surface being relatively smooth, so the gradients should be related before and after each update step. Adapted from Balduzzi et al. (2017).

The Solution:

Residual Networks introduce a novel architecture that helps mitigate these issues.

How Residual Networks Work:

Example: Three-Layer Network

The previous layer's output is passed to the next, forming a sequential processing pipeline. For example, a three-layer network is defined by:

$$\begin{aligned}\mathbf{h}_1 &= f_1[x, \phi_1] \\ \mathbf{h}_2 &= f_2[\mathbf{h}_1, \phi_2] \\ \mathbf{h}_3 &= f_3[\mathbf{h}_2, \phi_3] \\ y &= f_4[\mathbf{h}_3, \phi_4],\end{aligned}$$

where $\mathbf{h}_1$, $\mathbf{h}_2$, and $\mathbf{h}_3$ denote the intermediate hidden layers, x is the network input, y is the output, and the functions $f_k[\cdot, \phi_k]$ perform the processing.

Since the processing is sequential, we can equivalently think of this network as a series of nested functions:

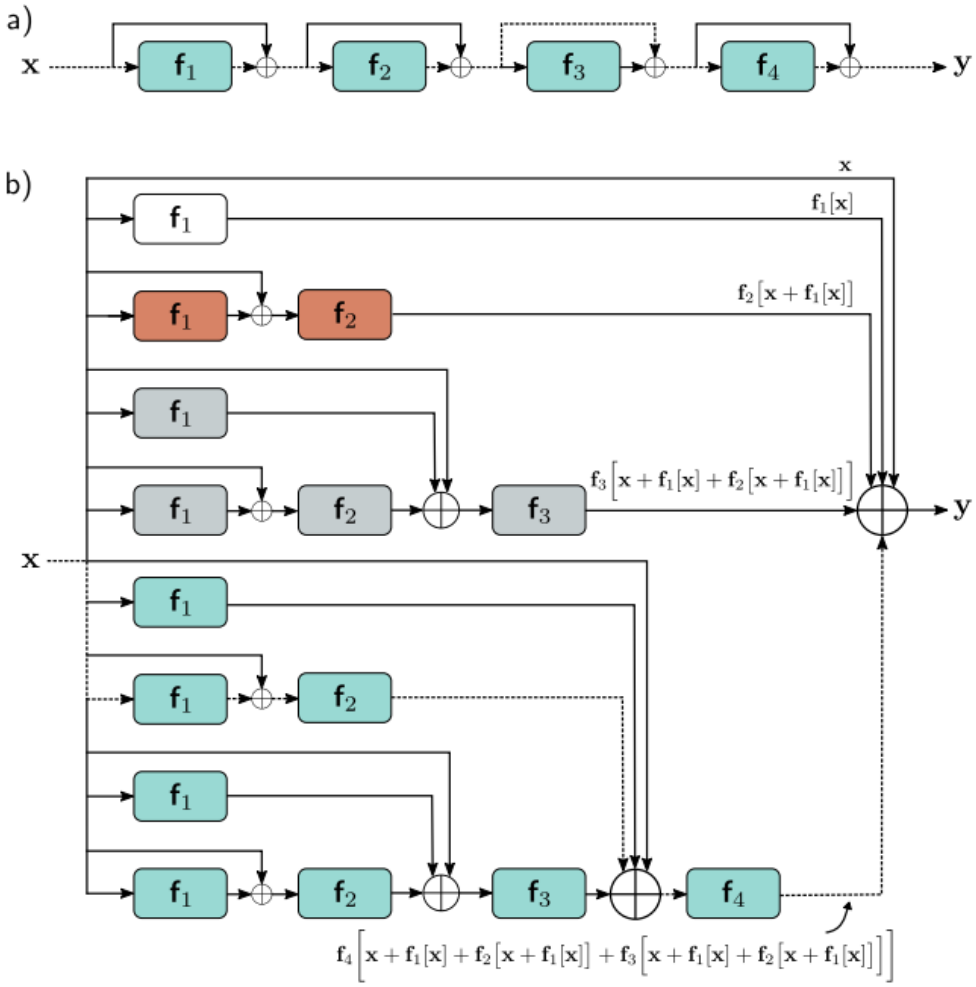
$$y = f_4\left[f_3\left[f_2\left[f_1[x, \phi_1], \phi_2\right], \phi_3\right], \phi_4\right].$$

Residual or Skip Connections

Residual or skip connections are branches in the computational path, whereby the input to each network layer $f[\cdot]$ is added back to the output. We can write the residual network as:

$$\begin{aligned}\mathbf{h}_1 &= x + f_1[x, \phi_1] \\ \mathbf{h}_2 &= \mathbf{h}_1 + f_2[\mathbf{h}_1, \phi_2] \\ \mathbf{h}_3 &= \mathbf{h}_2 + f_3[\mathbf{h}_2, \phi_3] \\ y &= \mathbf{h}_3 + f_4[\mathbf{h}_3, \phi_4]\end{aligned}$$

where the first term on the right-hand side of each line is the residual connection. Each function f_k learns an additive change to the current representation. It follows that their outputs must be the same size as their inputs. Each additive combination of the input and the processed output is known as a *residual block* or *residual layer*.

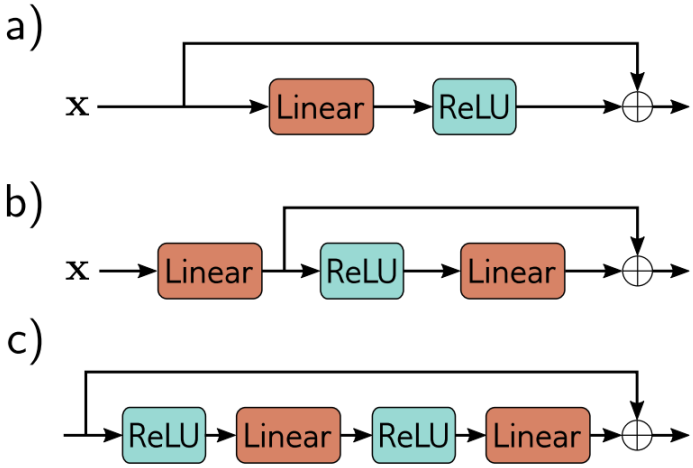


a) The output of each function $f_k[x, \phi_k]$ is added back to its input, which is passed via a parallel computational path called a residual or skip connection. Hence, the function computes an additive change to the representation. b) Upon expanding (unraveling) the network equations, we find that the output is the sum of the input plus four smaller networks (depicted in white, orange, gray, and cyan, respectively, and corresponding to terms in equation); we can think of this as an ensemble of networks. Moreover, the output from the cyan network is itself a transformation $f_4[\cdot, \phi_4]$ of another ensemble, and so on. Alternatively, we can consider the network as a combination of 16 different paths through the computational graph. One example is the dashed path from input x to output y, which is the same in panels (a) and (b).

Once more, we can write this as a single function by substituting in the expressions for the intermediate quantities $\mathbf{h}_k$:

$$\begin{aligned} y = & x + f_1[x] \\ & + f_2\left[x + f_1[x]\right] \\ & + f_3\left[x + f_1[x] + f_2\left[x + f_1[x]\right]\right] \\ & + f_4\left[x + f_1[x] + f_2\left[x + f_1[x]\right] + f_3\left[x + f_1[x] + f_2\left[x + f_1[x]\right]\right]\right]. \end{aligned} \tag{1}$$

Order of operations is important



a) The usual order of linear transformation or convolution followed by a ReLU nonlinearity means that each residual block can only add non-negative quantities. b) With the reverse order, both positive and negative quantities can be added. However, we must add a linear transformation at the start of the network in case the input is all negative. c) In practice, it's common for a residual block to contain several network layers.

Why Residual Blocks Work

1. **Easier Gradient Flow:** In deep networks, gradients can become extremely small, leading to slow or ineffective learning, a problem known as **vanishing gradients**. The identity mapping from the skip connection ensures that gradients flow more freely through the network, making backpropagation more effective.
2. **Identity Mapping:** If the transformation $F(x)$ produces an output very close to zero, the residual block essentially performs an **identity mapping**, i.e., $y = x$. This feature allows the network to avoid performance degradation, especially in deeper networks, as layers can simply learn to perform no transformation when none is needed.
3. **Mitigating the Degradation Problem:** As neural networks grow deeper, they sometimes suffer from a **degradation problem**, where adding more layers leads to worse training performance. This is different from overfitting—it's a failure to optimize. Residual blocks help mitigate this issue by ensuring that even if deeper layers are not needed, the network can simply learn identity mappings and perform as if those layers were not there, thus avoiding degradation.

References

[1] Simon J.D. Prince. *Understanding Deep Learning*. MIT Press, 2023.

[2] Deep Learning, by Bengio, Goodfellow, and Aaron Courville, 2015

[3] Zijie J. Wang, Robert Turko, Omar Shaikh, Haekyu Park, Nilaksh Das, Fred Hohman, Minsuk Kahng, and Duen Horng Chau. *CNN Explainer: Learning Convolutional Neural Networks with Interactive Visualization*. IEEE Transactions on Visualization and Computer Graphics (TVCG), 2020.

[4] Mayank. *Backpropagation for Convolution with Strides*. Medium, September 27, 2020. Available at: `https://medium.com/@mayank.utexas/backpropagation-for-convolution-with-strides-8137e4fc2710`.

[5] Reza Ghodsi. *Data Science Courses YouTube Channel*. Available at: `https://www.youtube.com/@DataScienceCourses`.

[6] Convolution vs Cross Correlation: Cmglee - Own work, CC BY-SA 3.0. Available at: `https://commons.wikimedia.org/w/index.php?curid=20206883`