# Deep Dive into Convolution and CNN

Farid Afzali, Ph.D., P.Eng.

# Kernel Functions

**Kernel in Convolutional Neural Networks (CNNs)**:

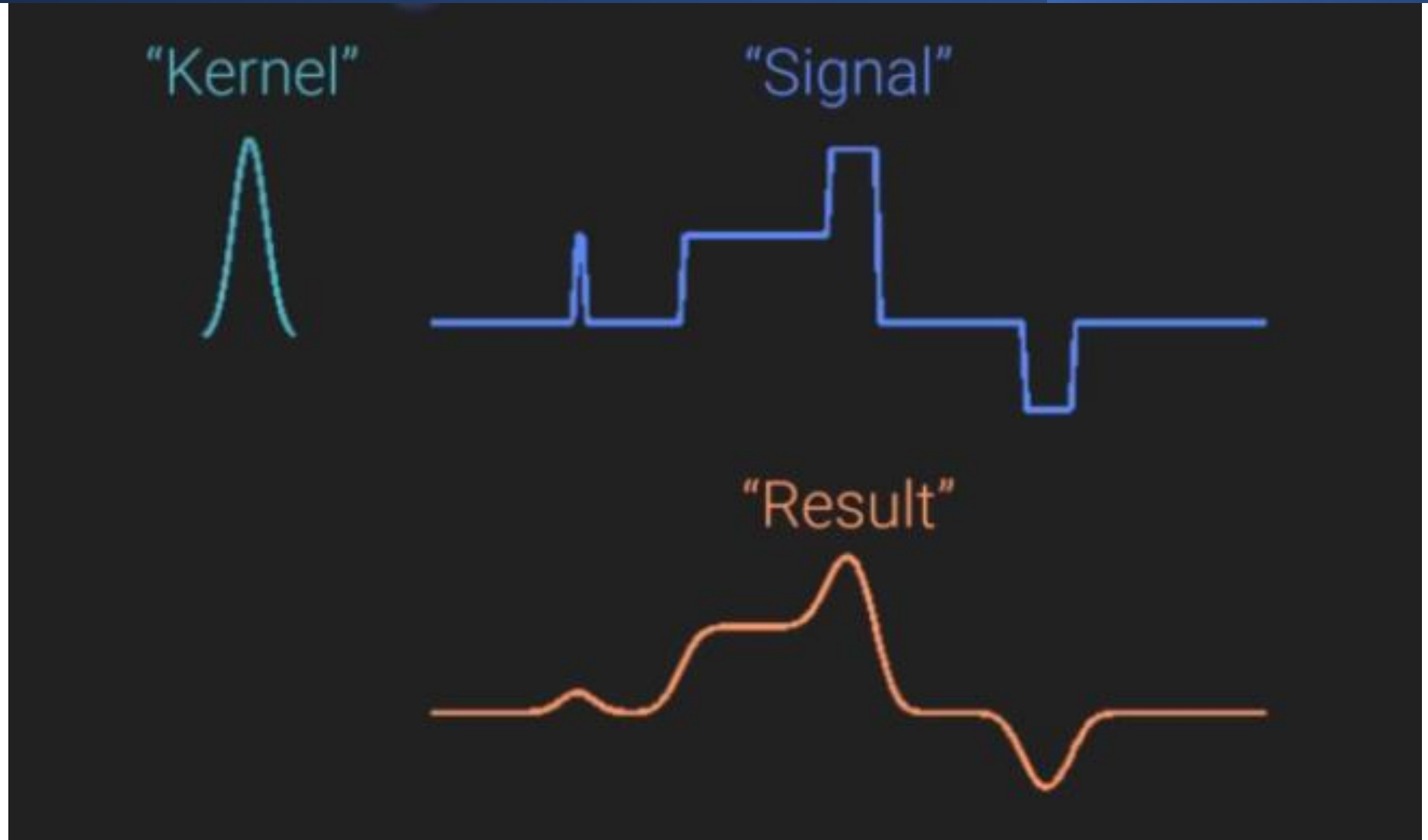
- In the context of CNNs, a kernel (also called a filter) is a small matrix used for feature extraction.
- Kernels in CNNs are applied to the input data using a convolution operation. This operation involves sliding the kernel over the input image or feature map to compute a weighted sum of neighboring pixels.
- CNN kernels are used to detect specific patterns or features in images, such as edges, corners, and textures.
- Convolutional layers in CNNs use multiple kernels to extract different features from the input data, creating feature maps that capture hierarchical information.

# Kernel Functions

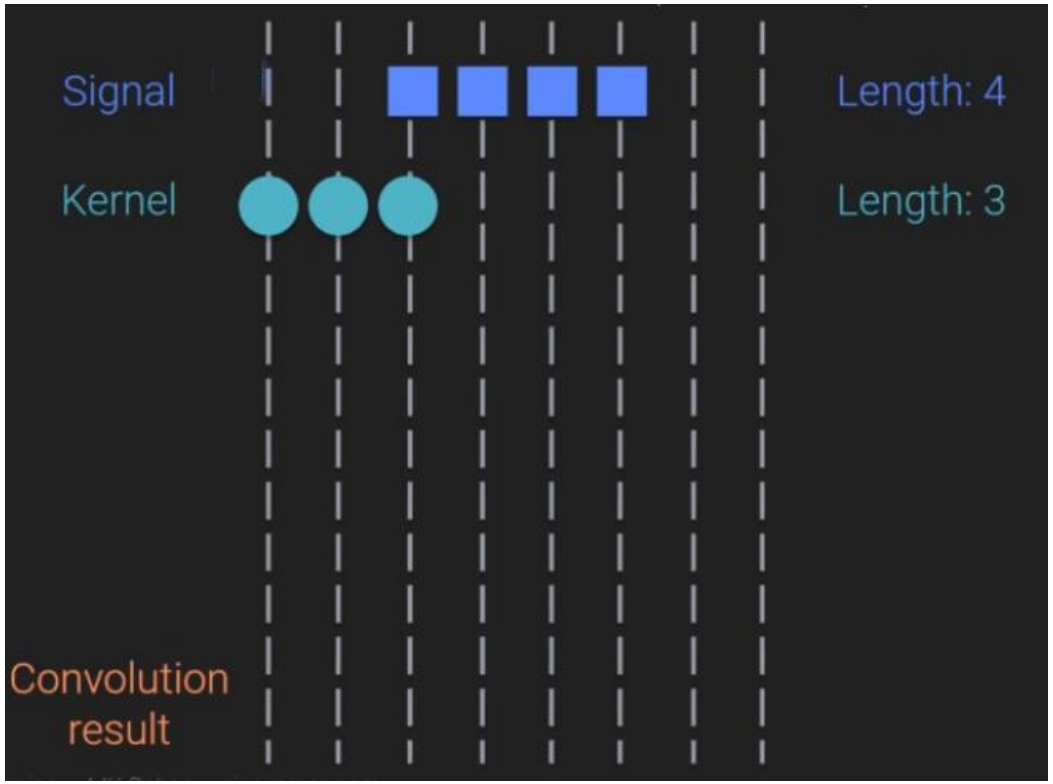
**Kernel in Machine Learning**:

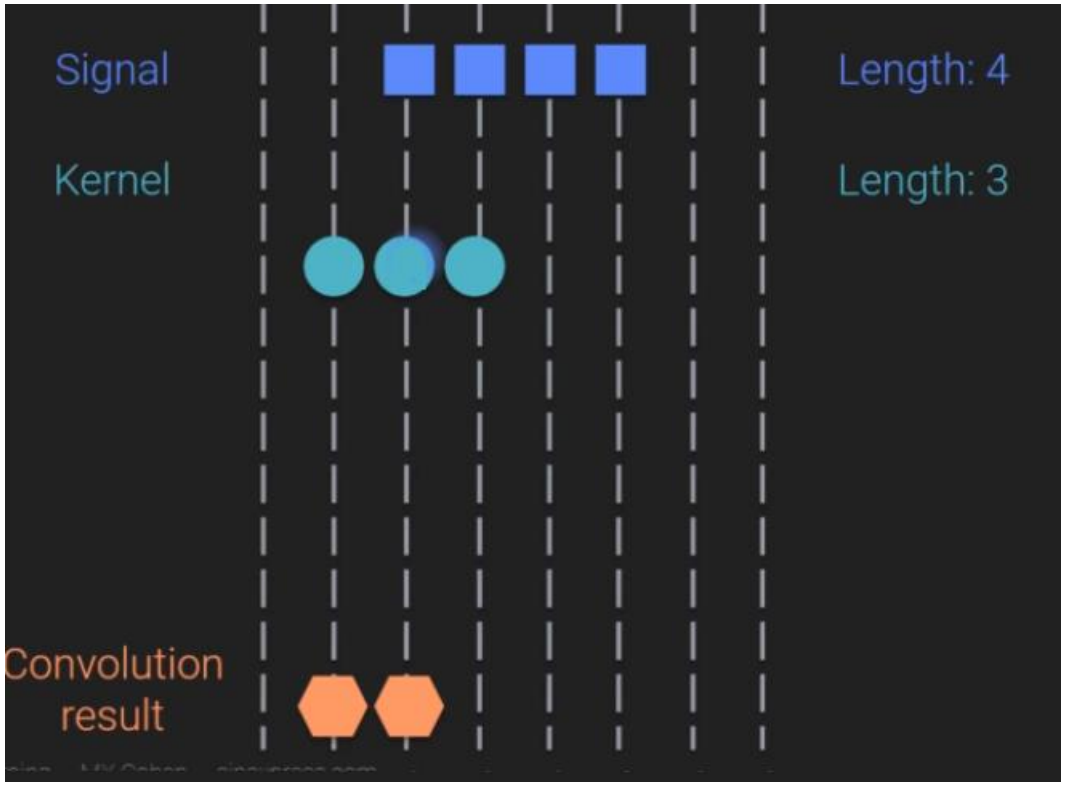
- In machine learning, a kernel refers to a mathematical function or transformation used primarily in support vector machines (SVMs) and kernel-based methods.
- Kernels in this context are used to map data points into a higher-dimensional space, where they become more separable. This transformation helps algorithms like SVMs find optimal decision boundaries.
- Common types of kernels in machine learning include linear, polynomial, radial basis function (RBF), and sigmoid kernels.
- Kernels in machine learning are used to capture complex patterns and relationships in data, especially in classification tasks.

# Convolution in 1D(Signal)

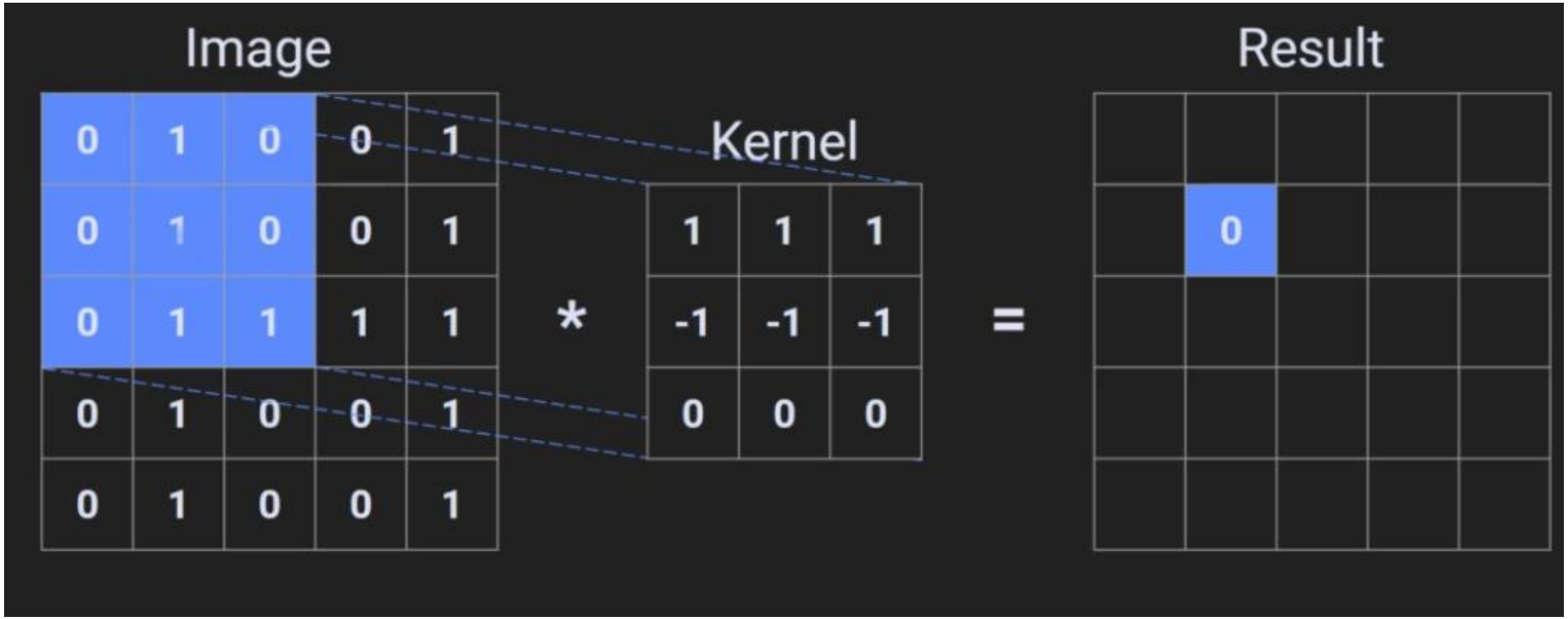


# Convolution in 1D(Signal)

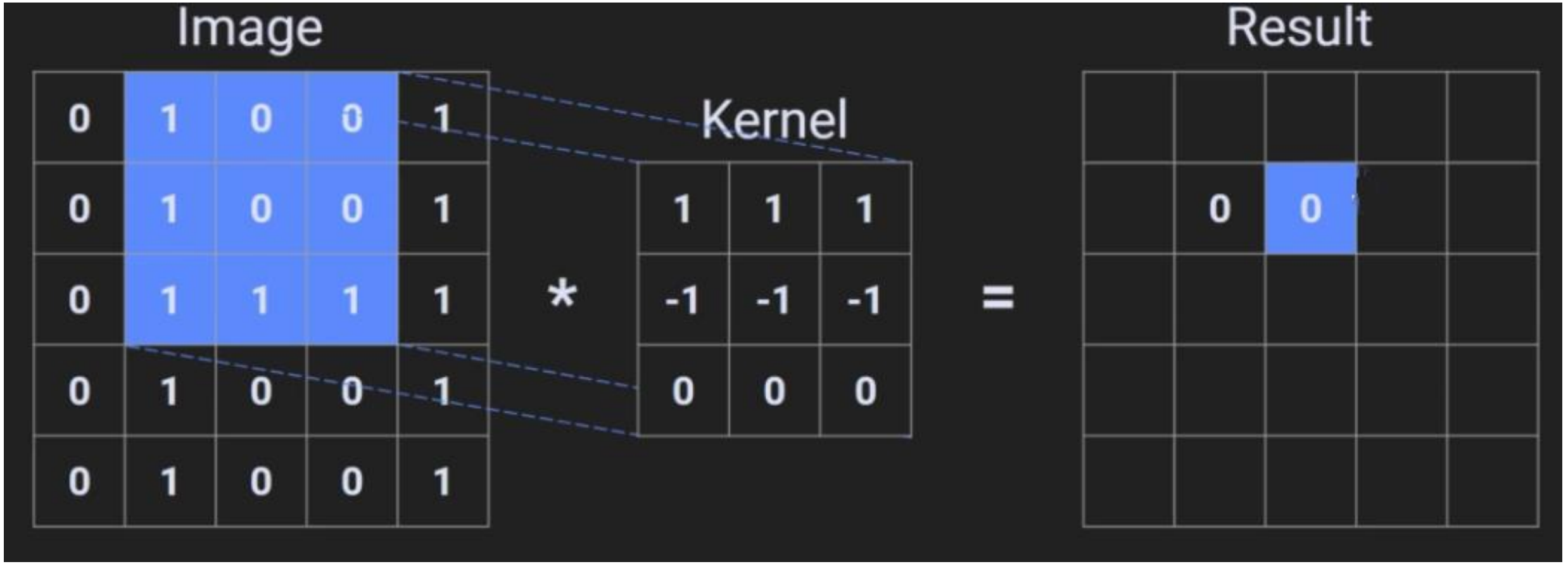


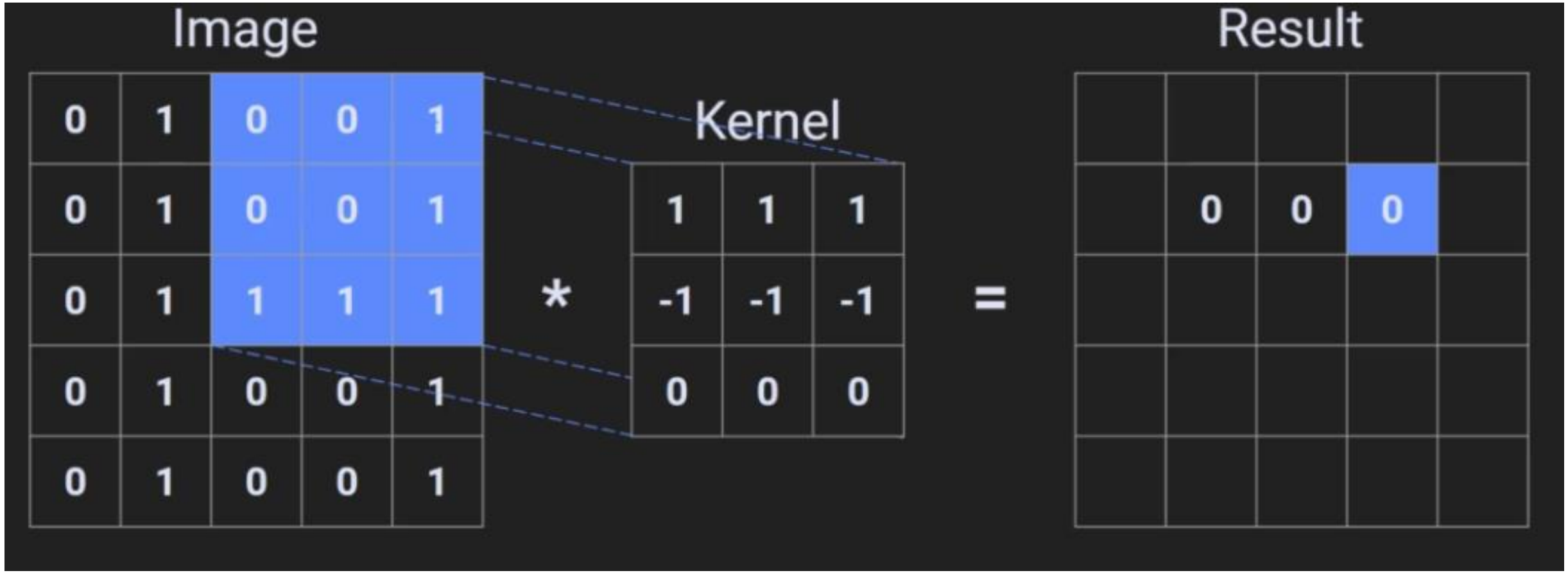
# Convolution in 2D(Image)



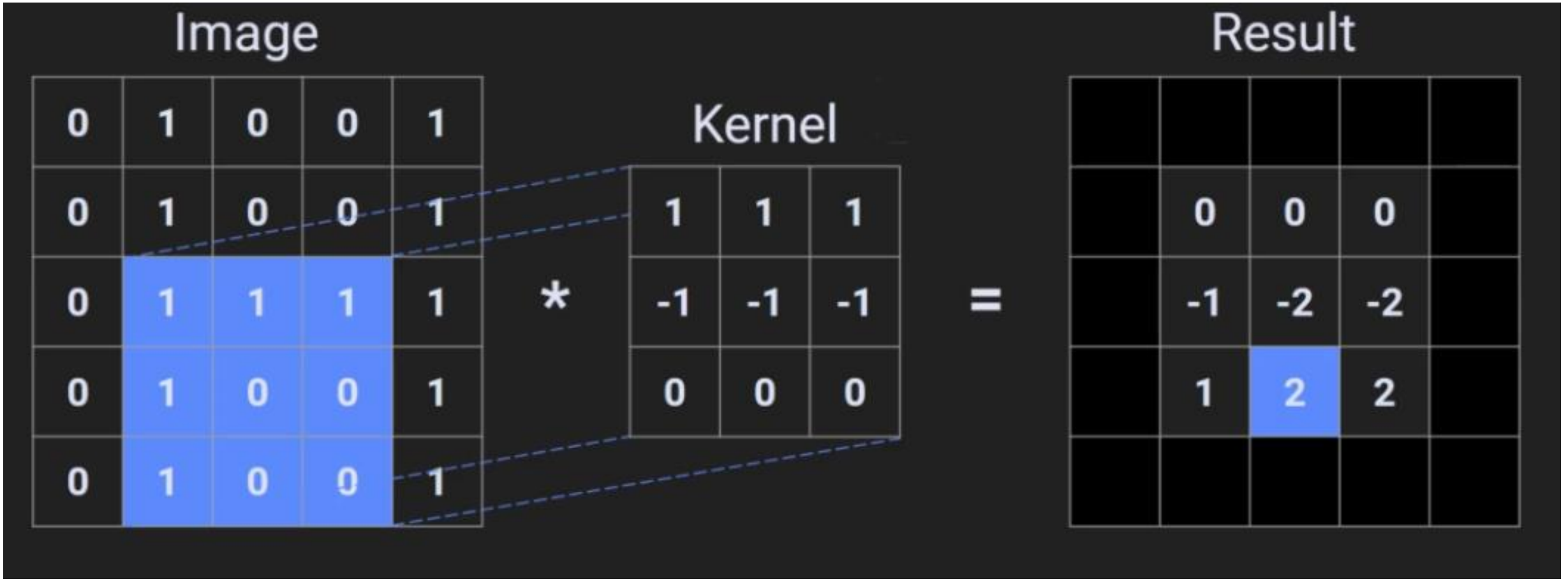
# Convolution in 2D(Image)



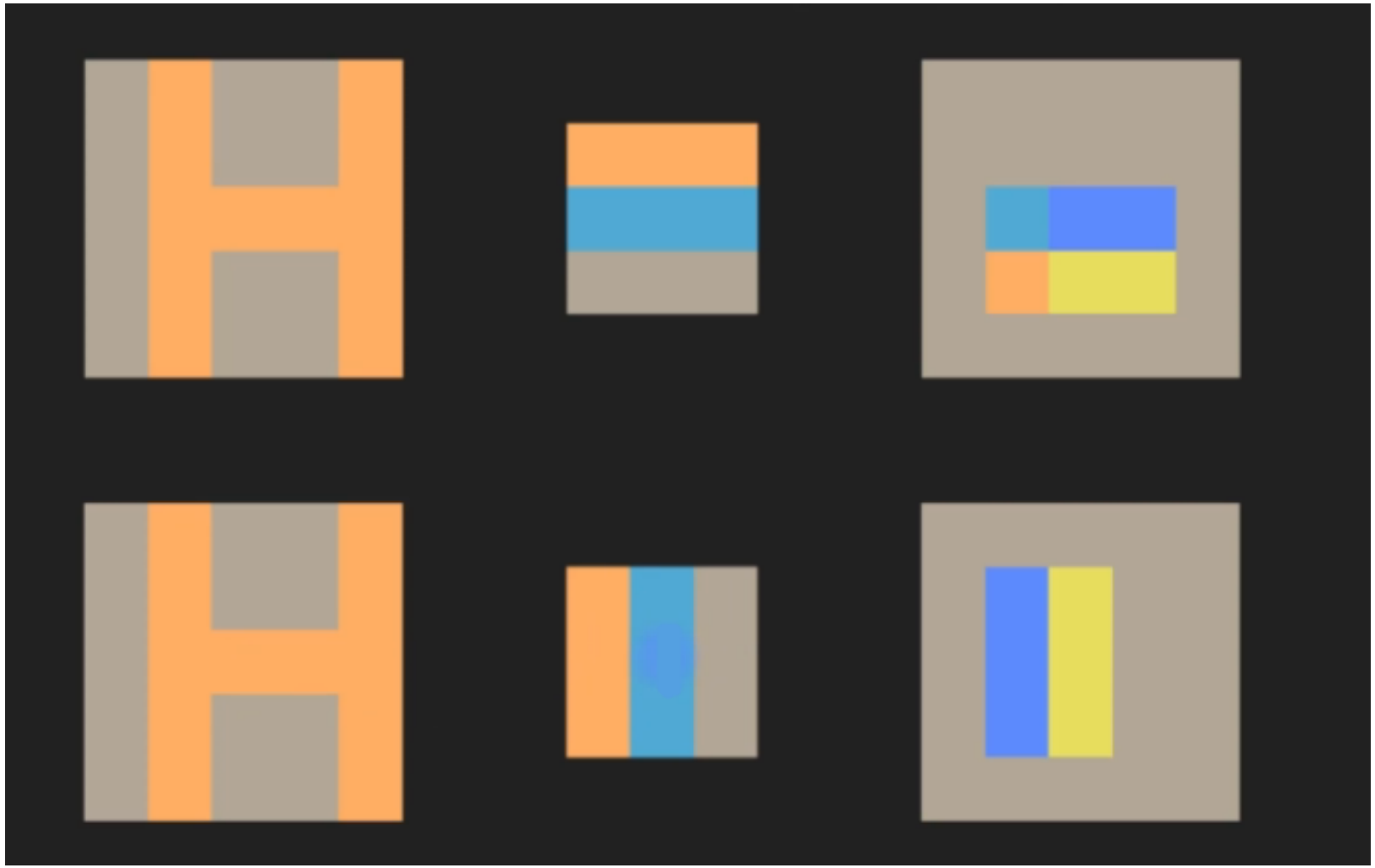
# Convolution in 2D(Image)



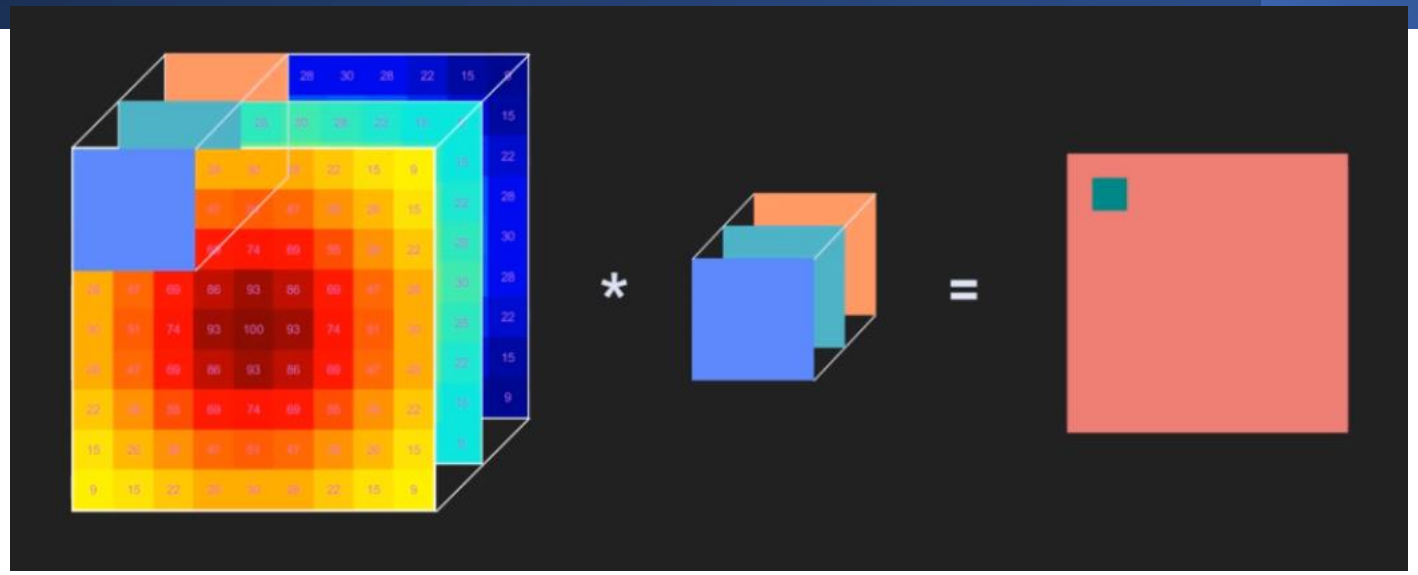
# Convolution in 2D(Image)

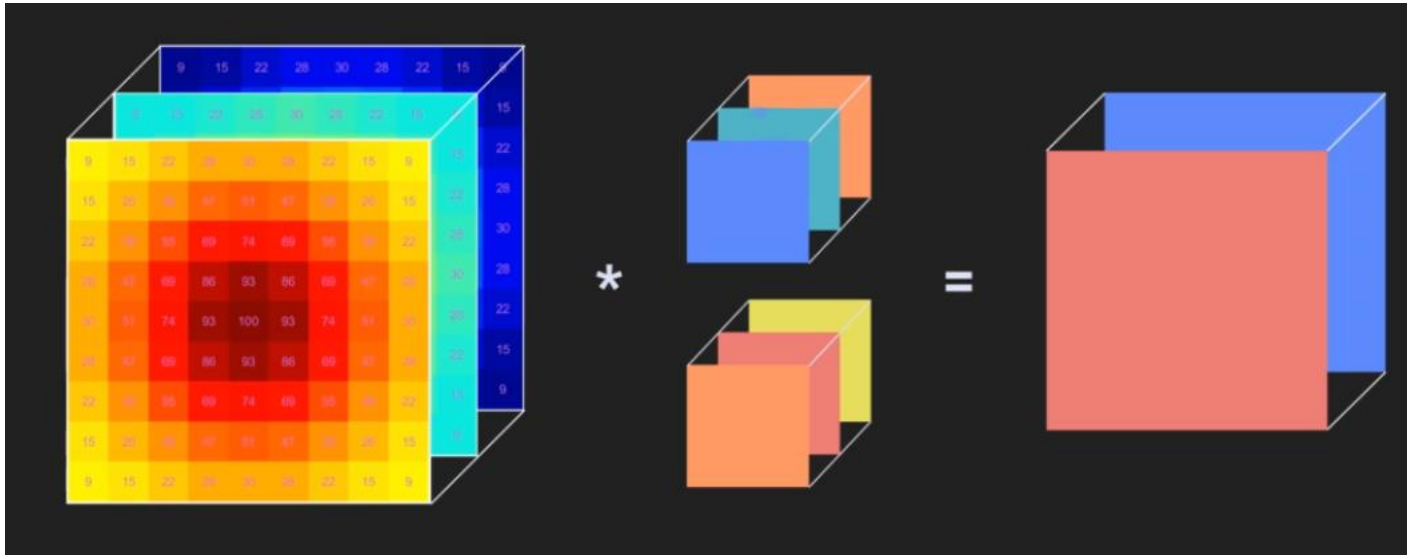


# Convolution in 2D(Image)



# Convolution in 2D(Image)

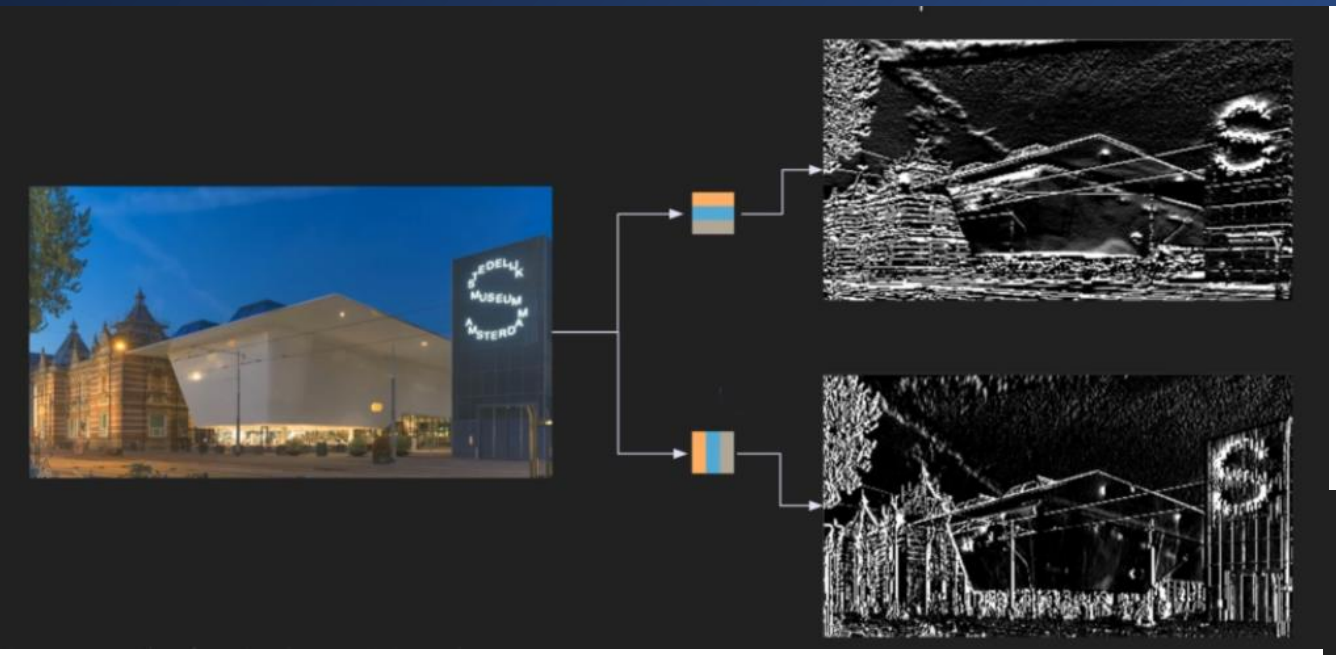


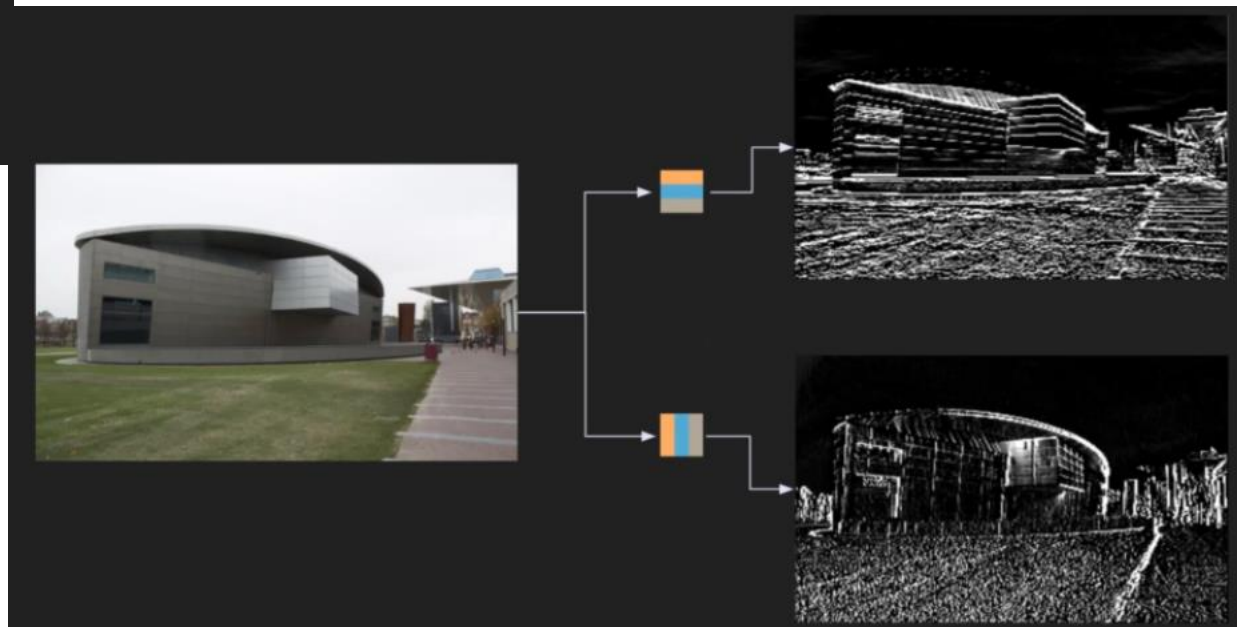


# Convolution in 2D(Image)

- Edges always cause difficulties. We deal with this through padding.
- Image convolution in DL also involves down sampling. This is done via stride and pooling.
- N kernels produces an N layer results. These layers are called channels, but these are features not RGB.
- Use kernel sizes (3, 5, 7 , etc.) to have an exact center.
- Formally, CNN implement cross-correlation, not convolution. But it doesn’t actually matter because the kernels are empirically learned.

# Kernel vs Feature maps

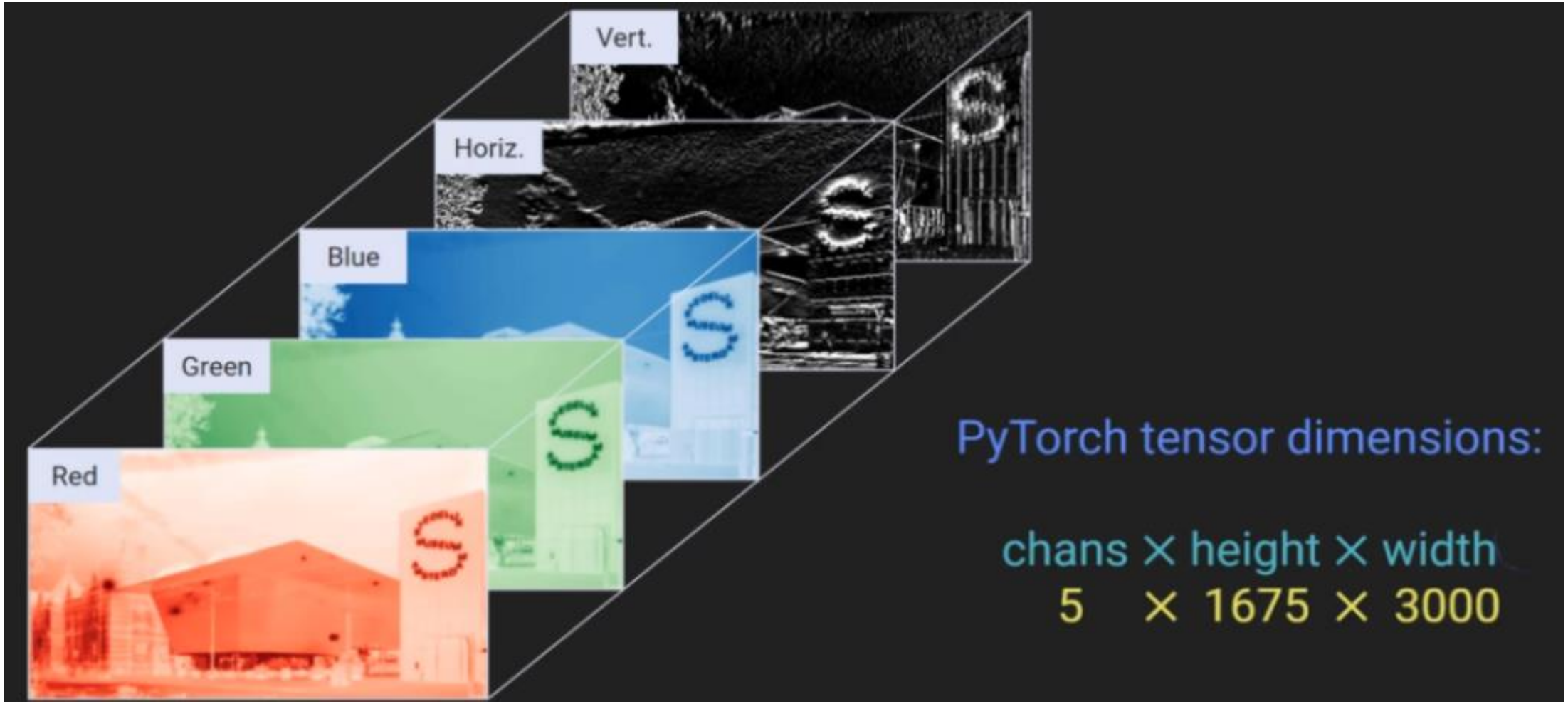




# Kernels concepts

- Kernels are filters that extract features from an image. The same kernel applied to different images will give different feature maps.
- In DL, kernels begin random and are learned through gradient descent. After learning, kernels are the same for all images. Using pre-trained kernels is called transfer learning.
- Kernels are not used to classify or make decisions; they are used to extract features . These features are used for classification.

# Feature maps vs Channels



# Convolution hyperparameters Padding, Stride

In convolutional neural networks (CNNs), stride and padding are two important hyperparameters that affect the behavior and output size of convolutional layers. Here's an explanation of stride and padding in convolution:

Stride:

- Stride is a hyperparameter that determines the step size at which the convolutional filter or kernel moves across the input data during the convolution operation.
- When a convolutional filter is applied to the input, it is initially placed at a specific position (e.g., the top-left corner). With a stride of 1, the filter moves one pixel at a time, performing a convolution operation at each position, until it has covered the entire input.
- If you increase the stride value to, for example, 2, the filter will move two pixels at a time, effectively reducing the spatial dimensions of the output feature map. A larger stride results in a smaller output size.
- Stride is used to control the spatial resolution of the output feature maps. Smaller strides preserve more spatial information, while larger strides downsample the feature maps more aggressively.

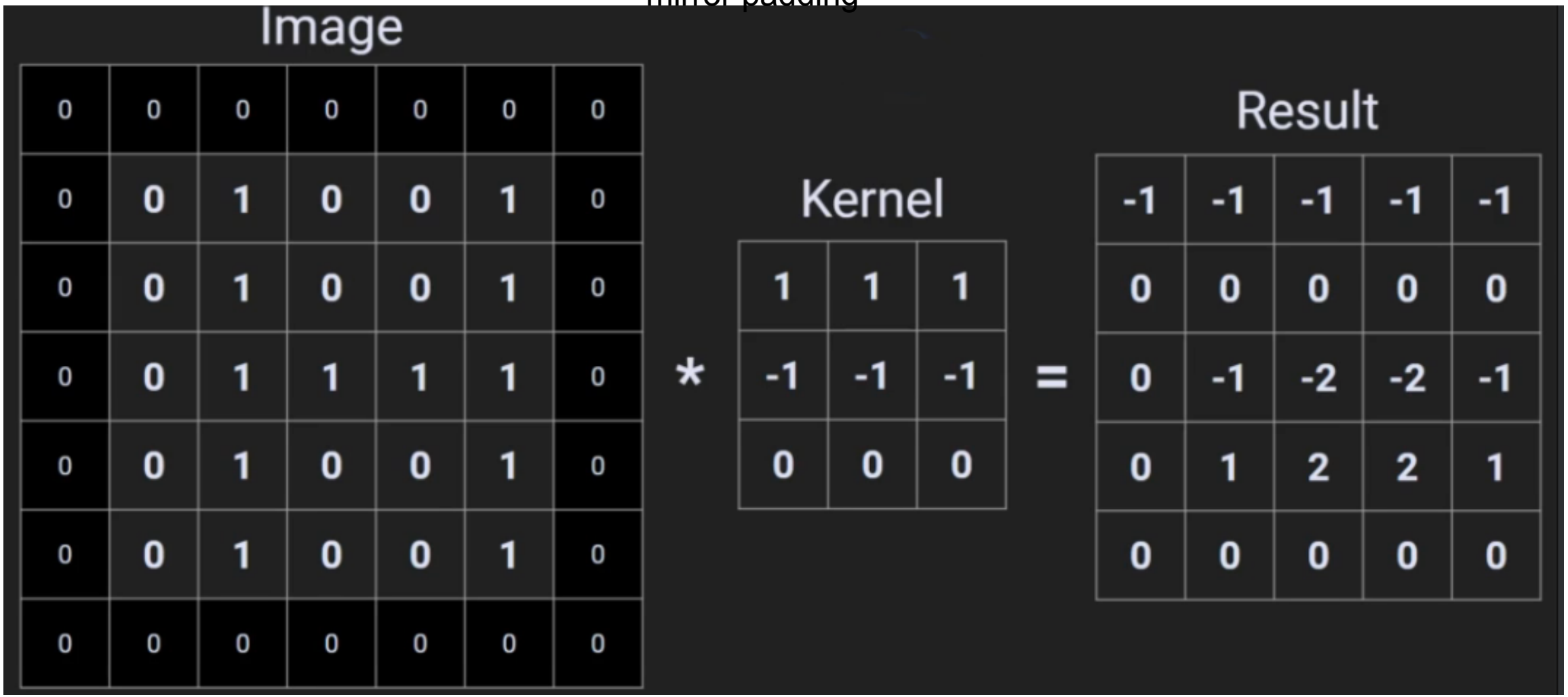
# Convolution hyperparameters Padding, Stride

Padding:

- Padding is another hyperparameter that determines whether and how the input data is padded with additional values (typically zeros) before applying convolutional filters.
- Padding is useful for controlling the spatial dimensions of the output feature maps. There are two common types of padding:
  - Valid (No Padding): When no padding is applied (often referred to as "valid" padding), the convolutional filter is only applied to positions where it completely fits within the input. This leads to a reduction in the spatial dimensions of the output feature map.
  - Same (Zero Padding): "Same" padding ensures that the output feature map has the same spatial dimensions as the input. To achieve this, zeros are added around the input data so that the filter can be applied to all positions, even at the edges of the input.
- Padding is useful for preserving spatial information, especially at the borders of the input image. It can help mitigate issues like the "border effect" where pixels at the edges of the image are less involved in the convolution operation.

# Convolution and Padding

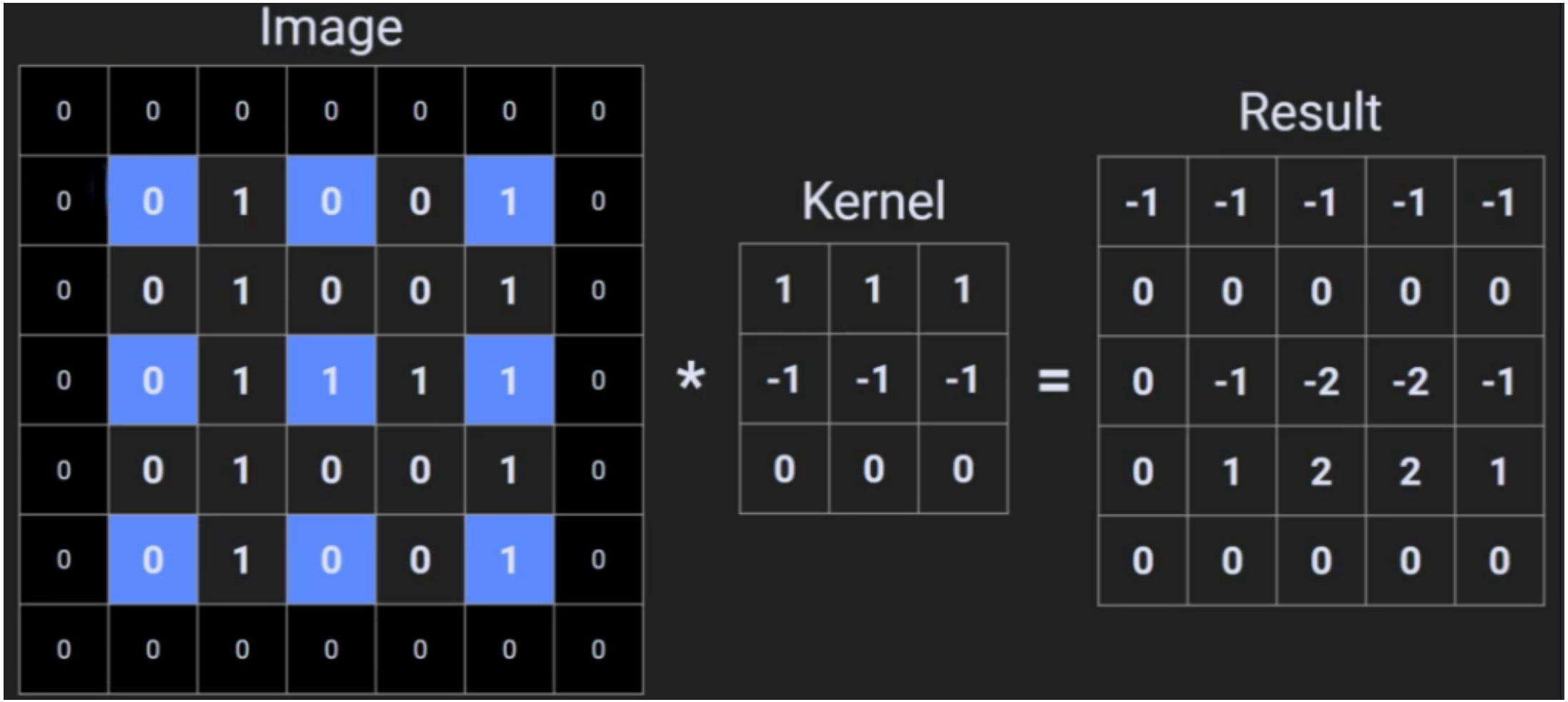
mirror padding



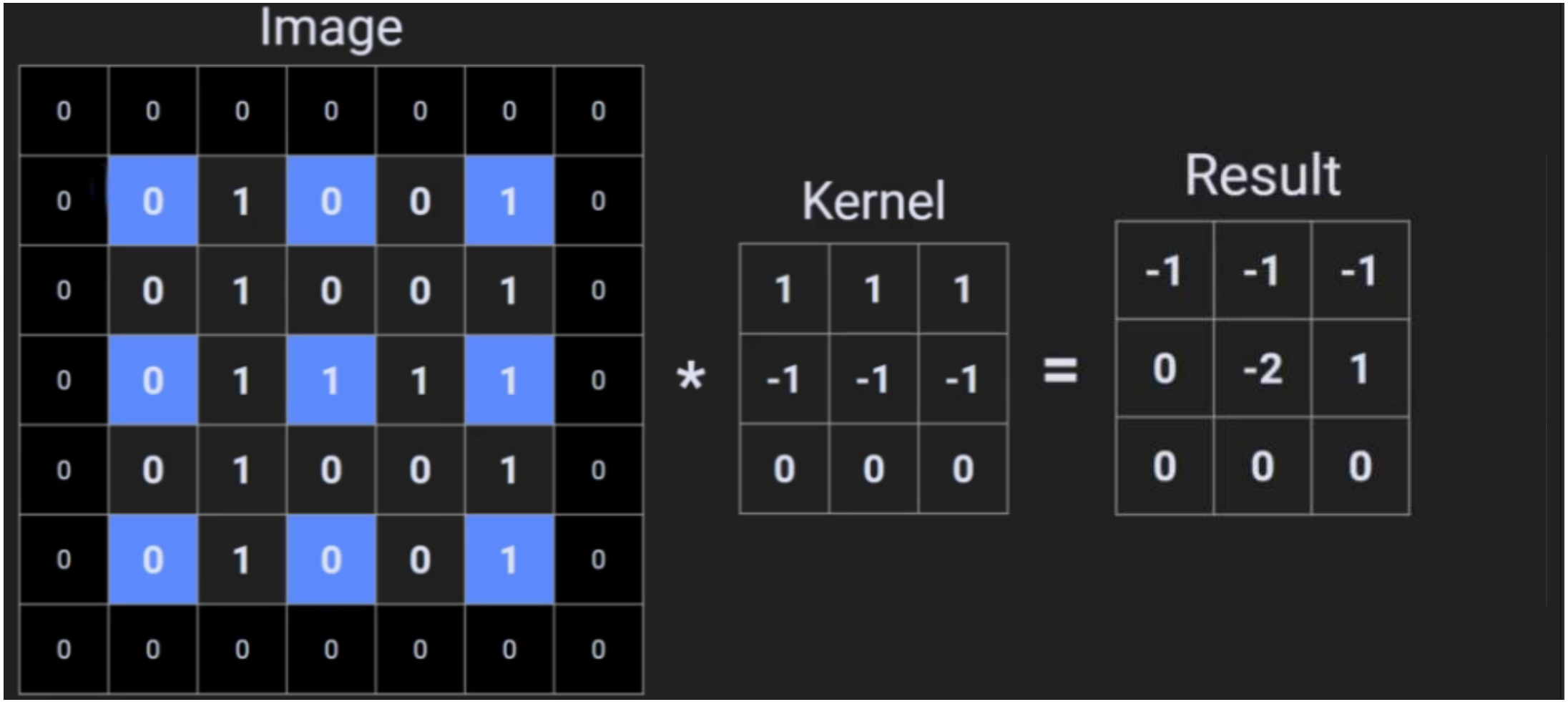
# Convolution hyperparameters Padding, Stride

- Padding involves inserting1+ rows and columns.
- Added rows/columns are symmetric.
- Padded numbers are usually zeros. It is also possible to wrap the image top-to-bottom (circular convolution).

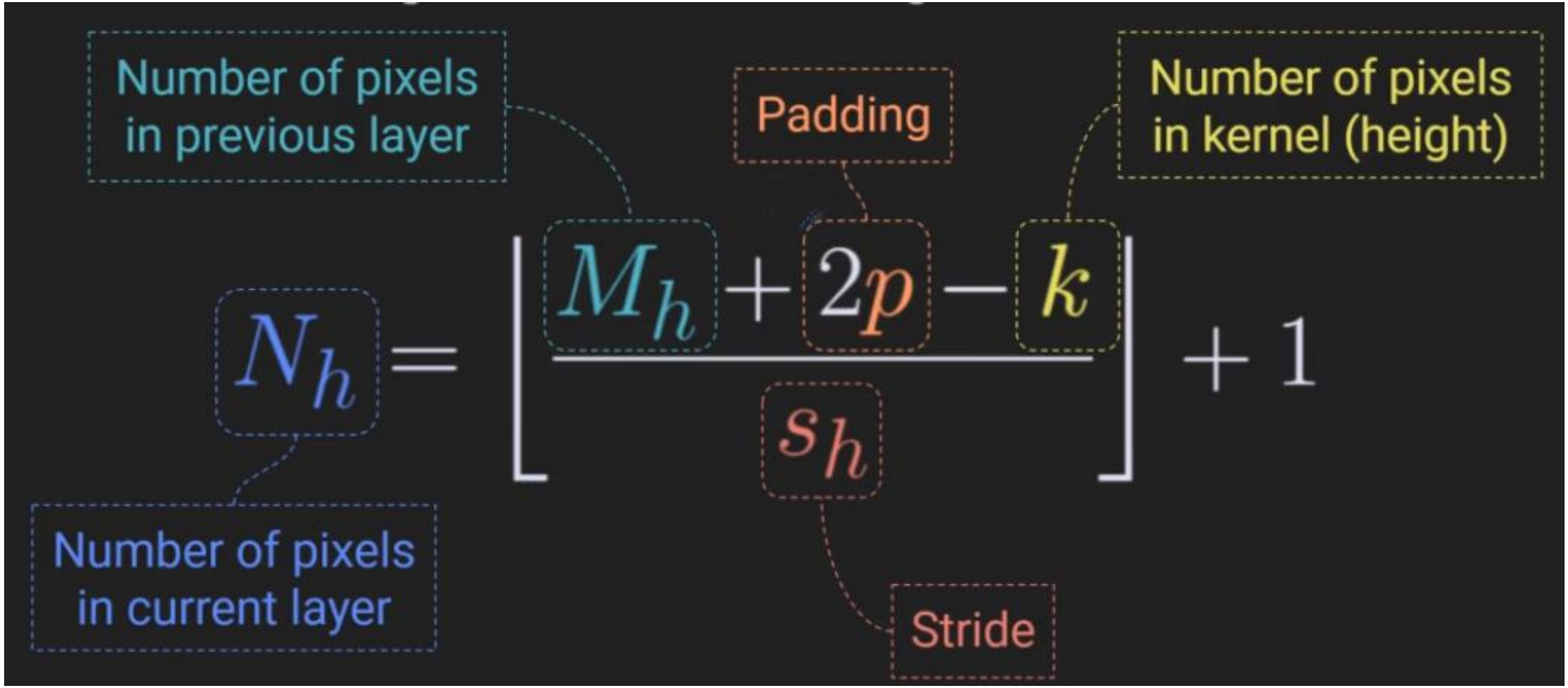
# Convolution and Stride



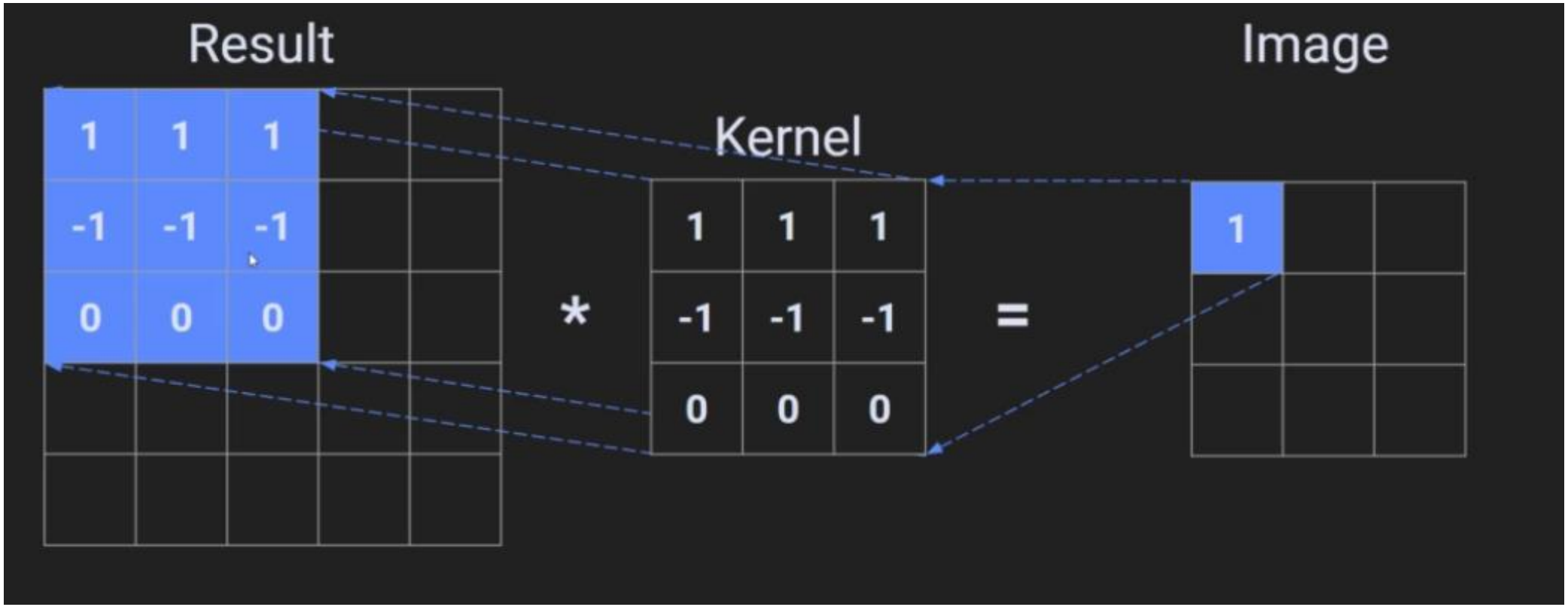
# Convolution and Stride



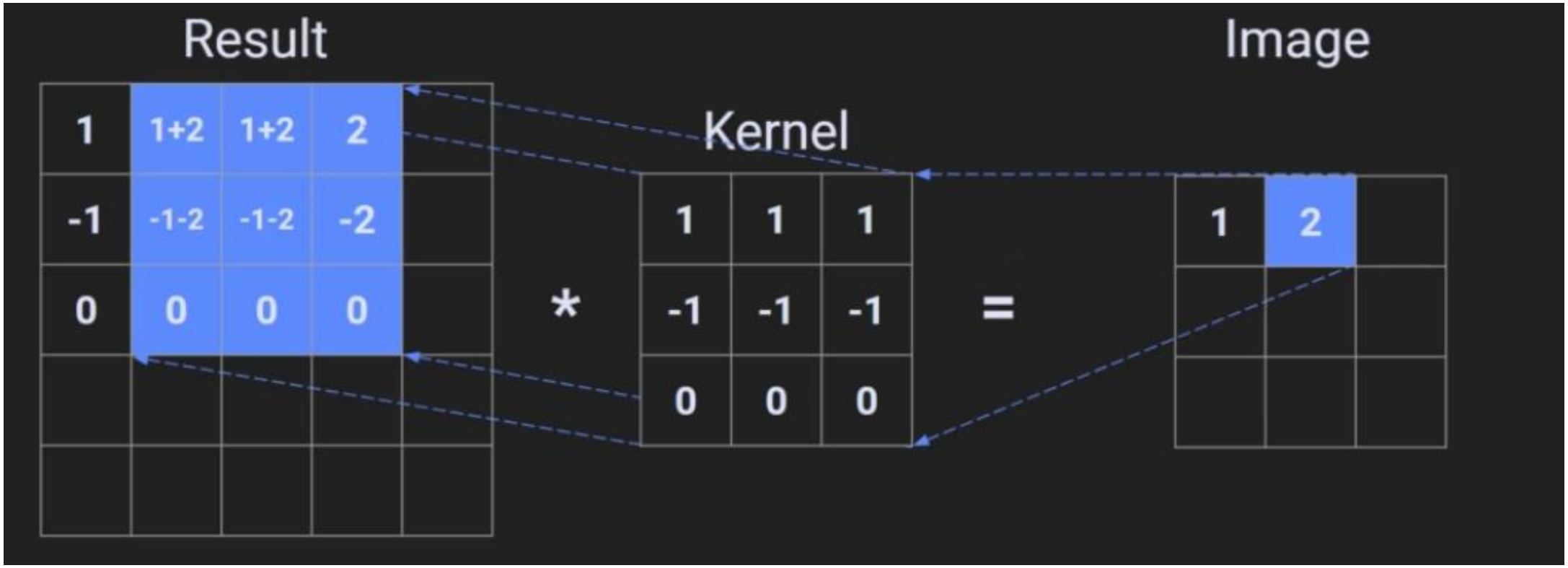
# Padding and Stride formulas



# Transpose Convolution



# Transpose Convolution



# Transpose Convolution, Convolution hyperparameters Padding, Stride

- Transpose convolution means to scalar-multiply a kernel by each pixel in an image.
- If the kernel is >1, the result will be higher resolution than the original image.
- Transpose convolution is used for autoencoders and super-resolution CNNs.
- Transpose convolution takes the same parameters as forward convolution: kernel size, padding, stride.

# Transpose Convolution

Number of pixels in input image.

Number of pixels in kernel (height)

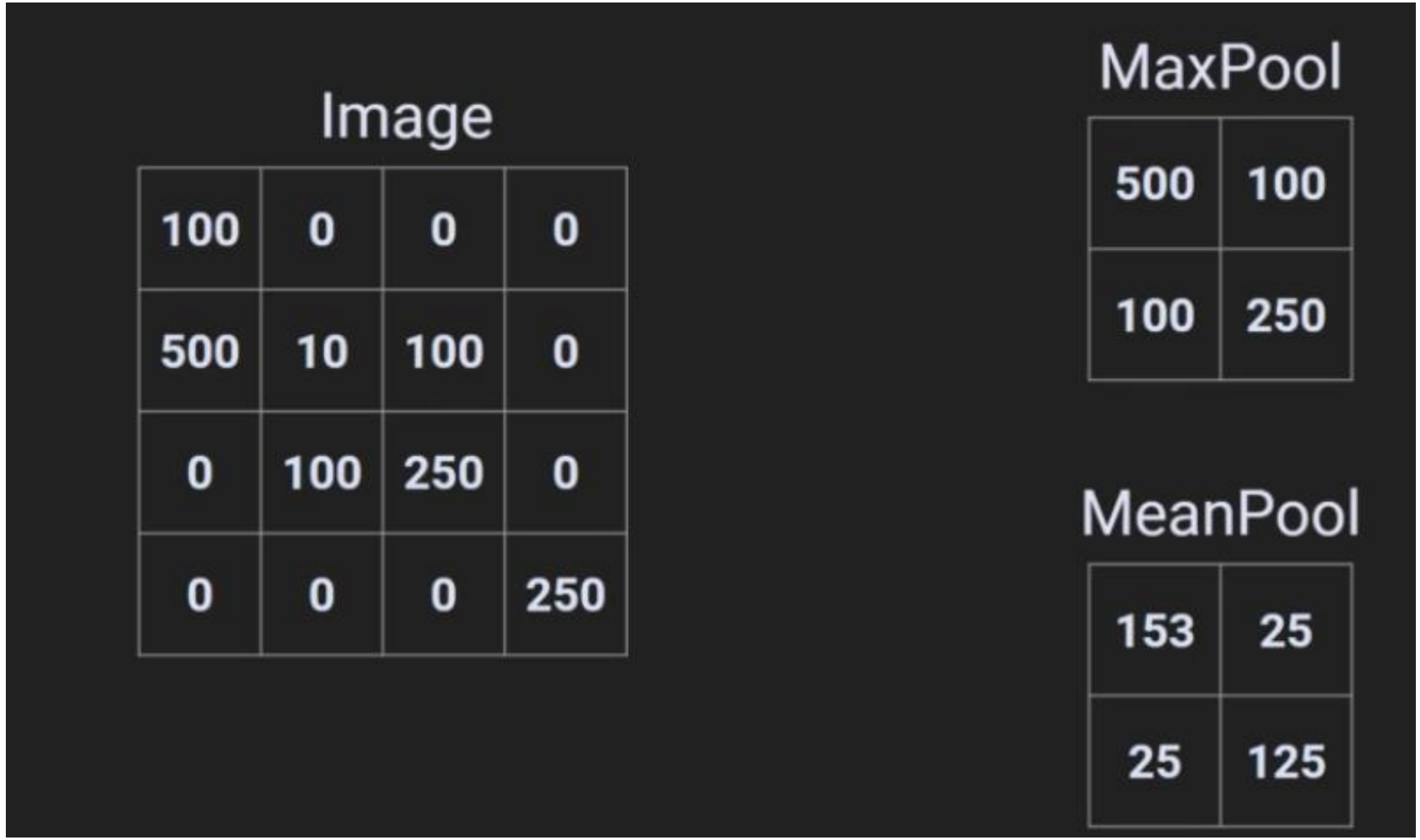
$$N_h = s_h(M_h - 1) + k - 2p$$

Number of pixels in output image.

Stride

Padding

# Max/Mean Pooling



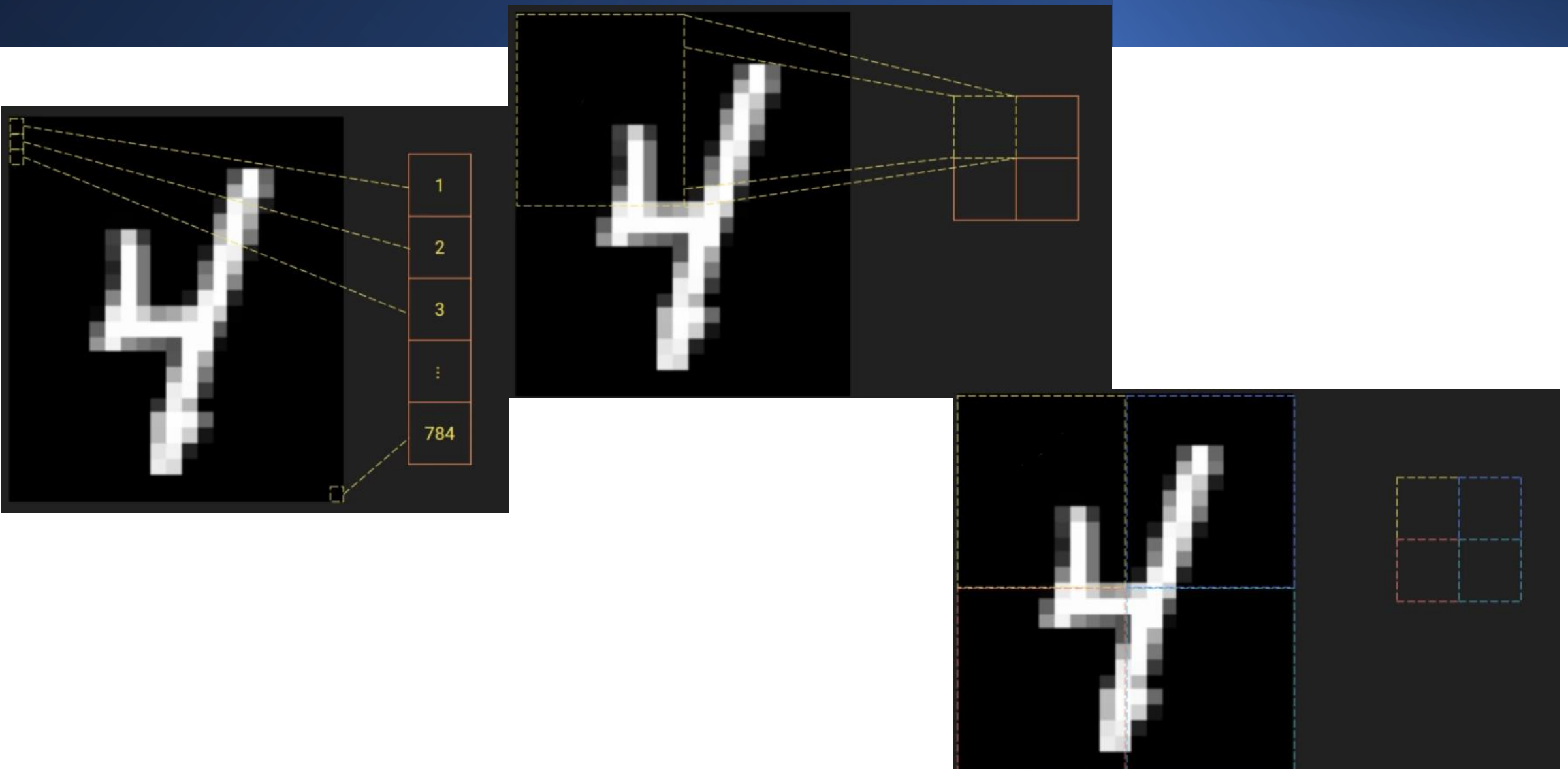
# Why use a pooling layer?

- Reduce dimensionality (fewer parameters)
- Selects for features over a broader spatial area
  - Increase receptive field size
- Deeper into the model, we want more channels with fewer pixels.
  - This make the representations increasingly abstract.

# Convolution hyperparameters Padding, Stride

- Max pooling: highlights sharp features.
  - Useful for sparse data and increasing contrast.
- Mean pooling: Smooths images (it is a low pass filter)
  - Useful for noisy data and reduces the impact of outliers on learning.

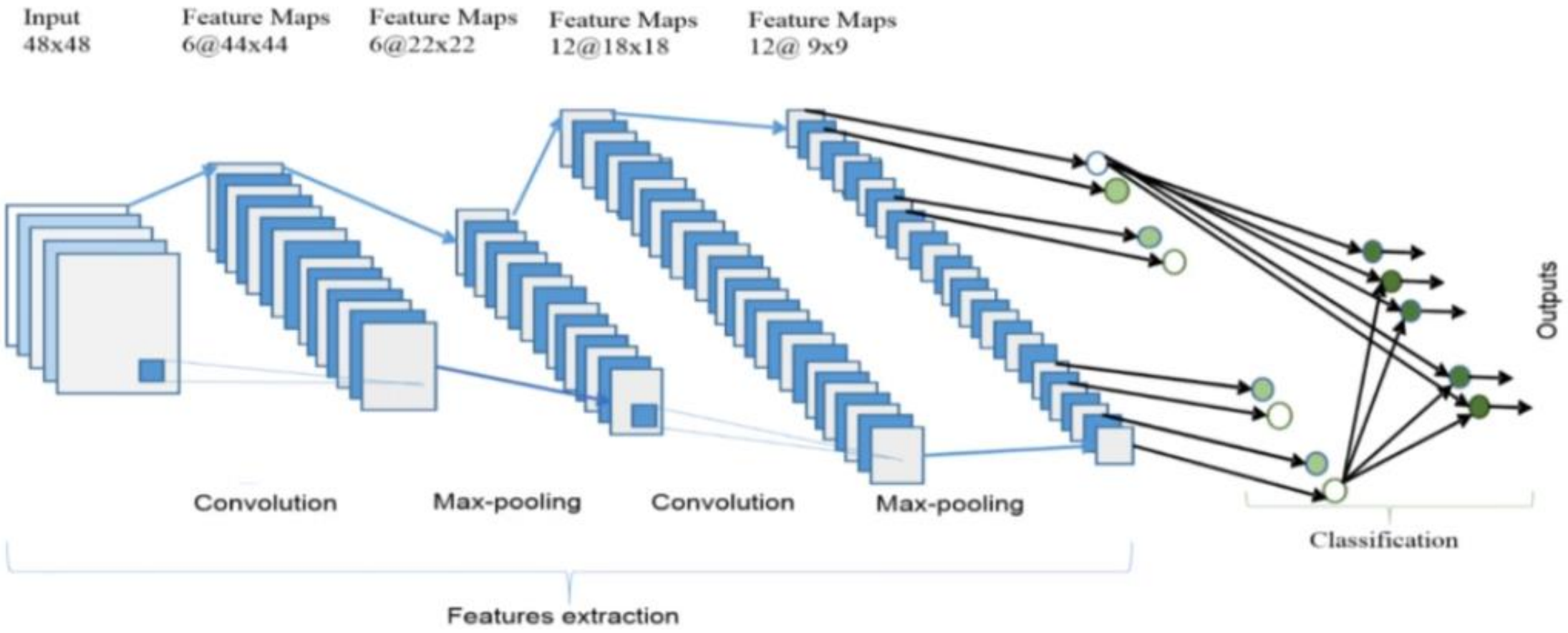
# Receptive Fields



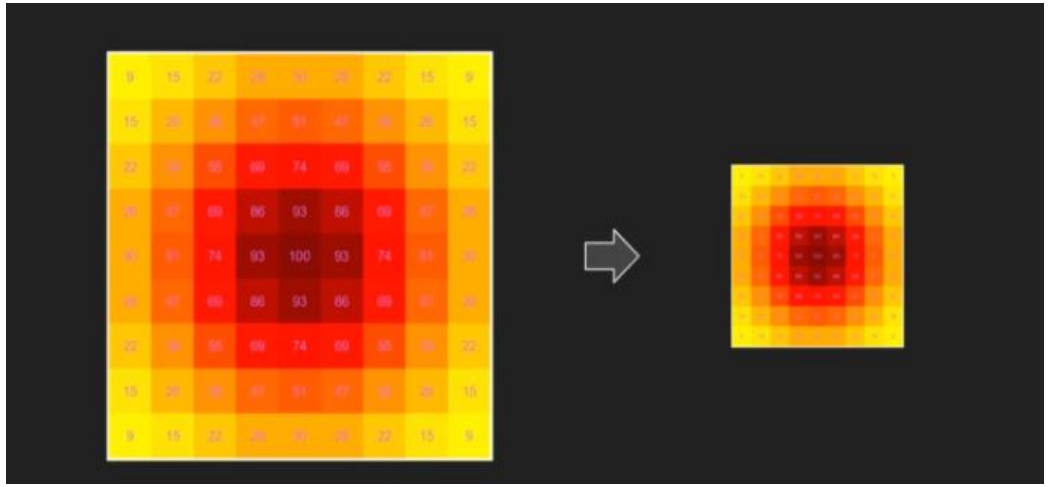
# Receptive Fields

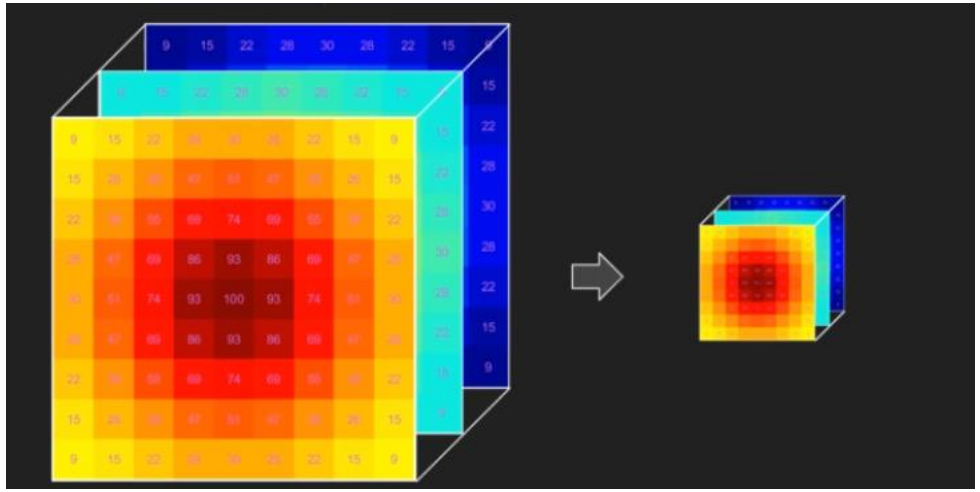
- ANN have one to one mapping between pixel and unit.
  - In other word, the receptive field of an ANN unit is one pixel.
- This makes the model non-robust to translations, resizing, rotations, etc.
- We want a model that can see the features anywhere in the image.

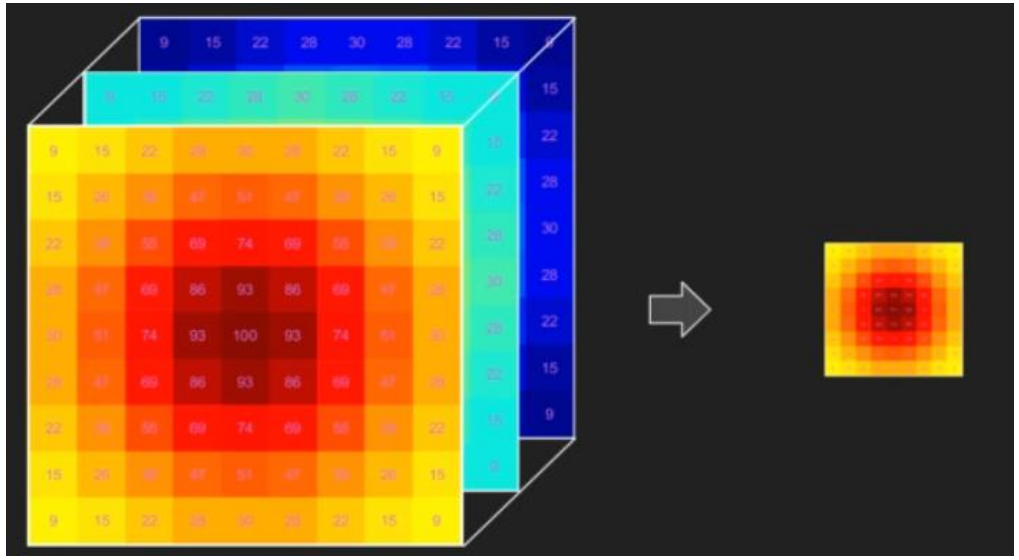
# Max/Mean Pooling



# Max/Mean Pooling



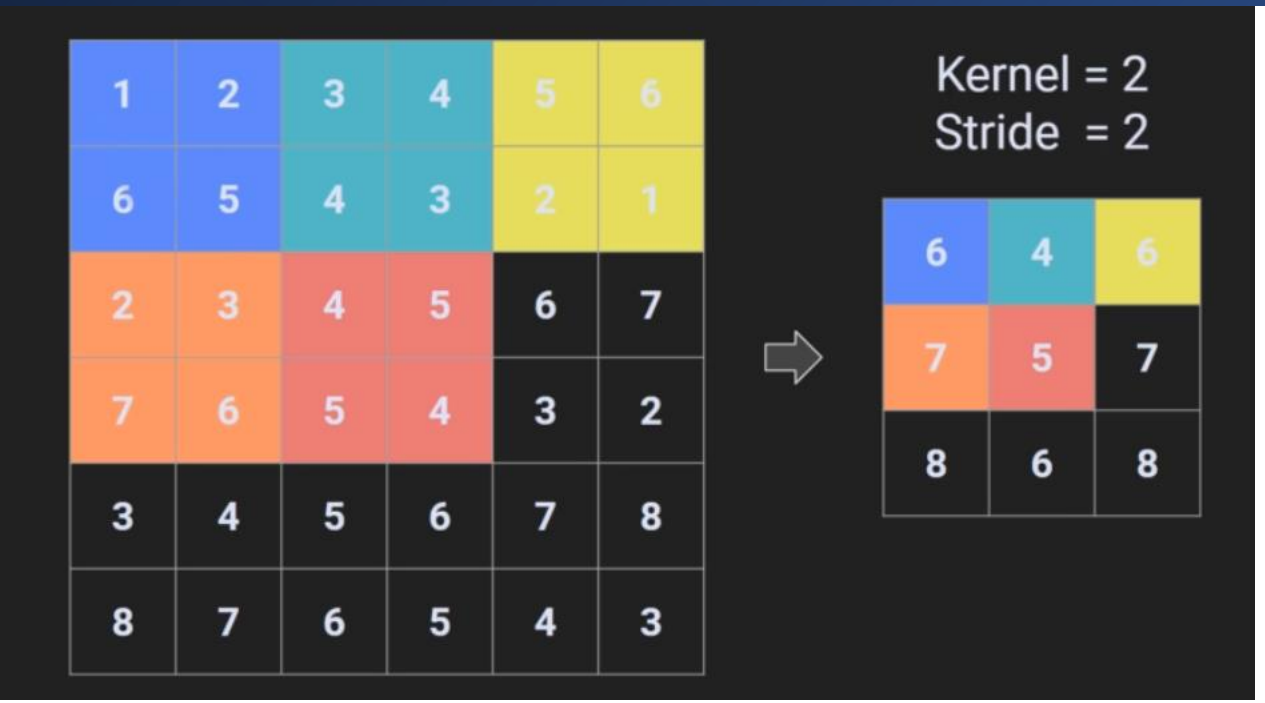


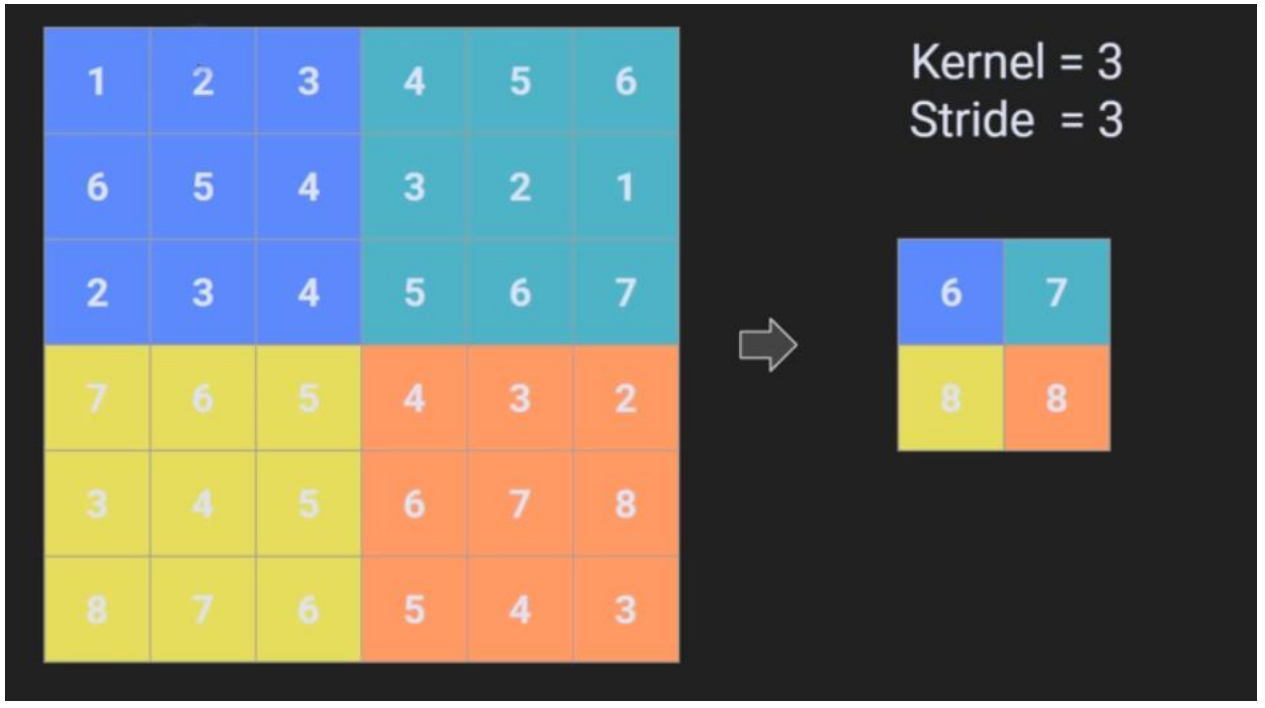


# Convolution hyperparameters Padding, Stride

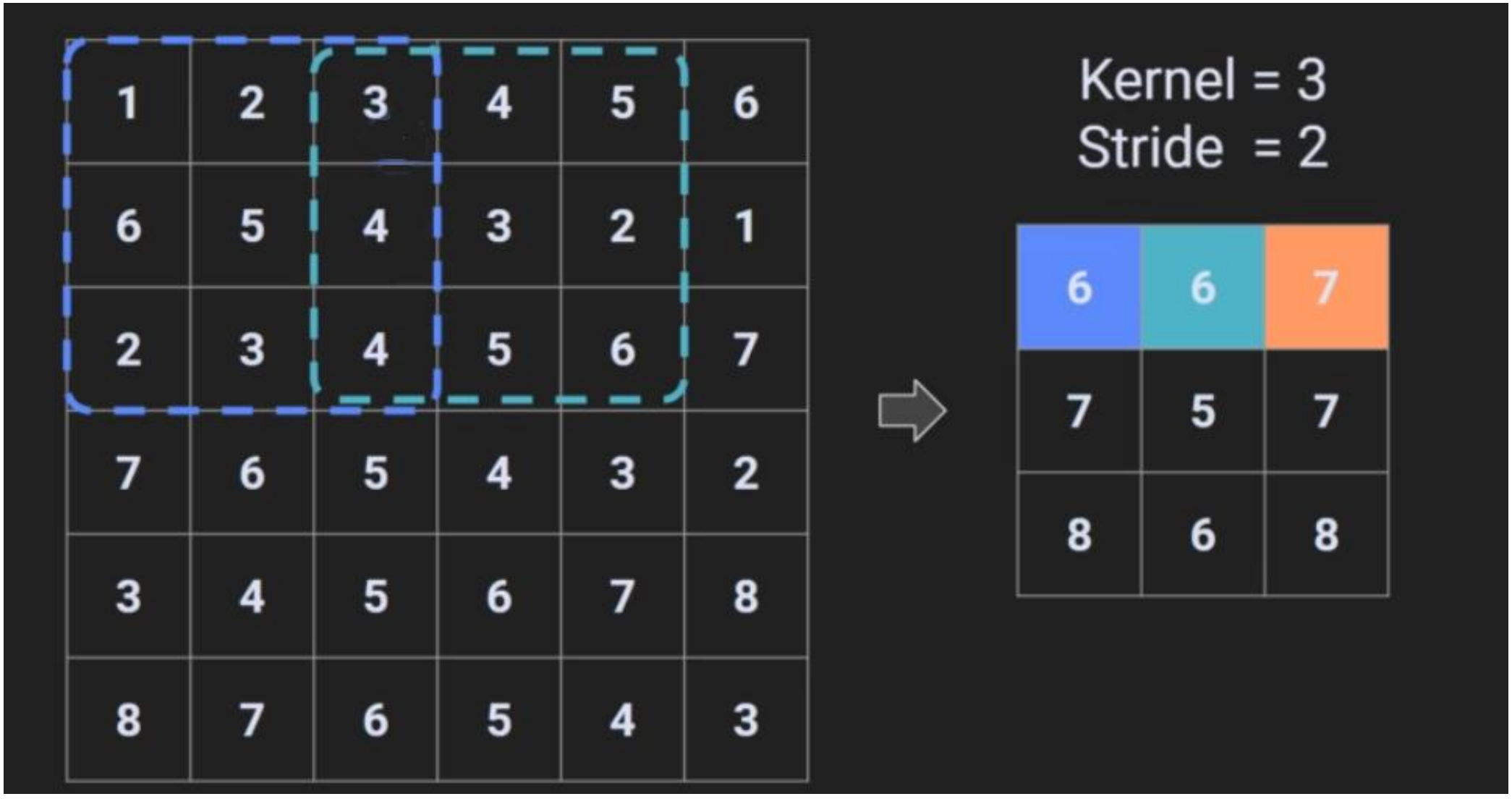
- Max pooling: highlights sharp features.
  - Useful for sparse data and increasing contrast.
- Mean pooling: Smooths images (it is a low pass filter)
  - Useful for noisy data and reduces the impact of outliers on learning.

# Parameters of Pooling

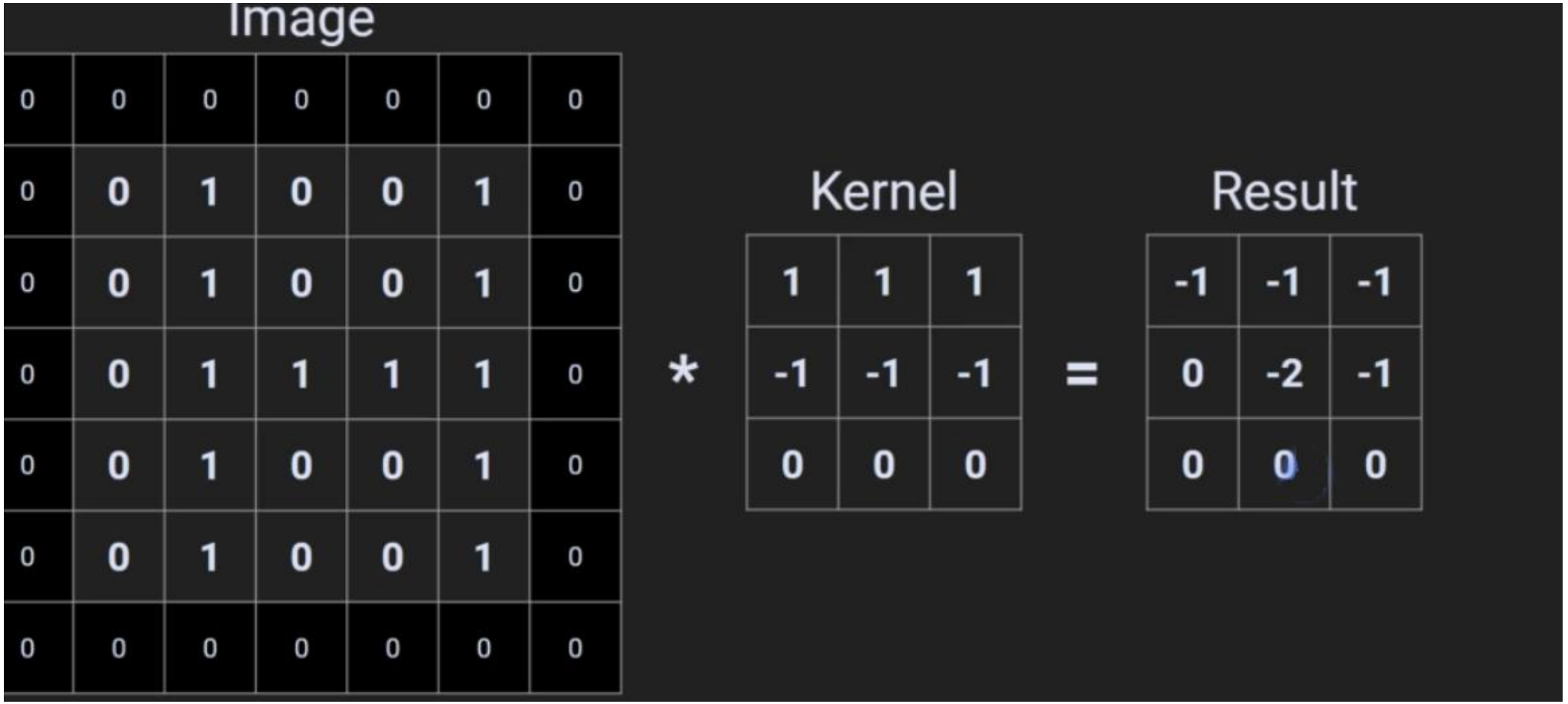


# Parameters of Pooling

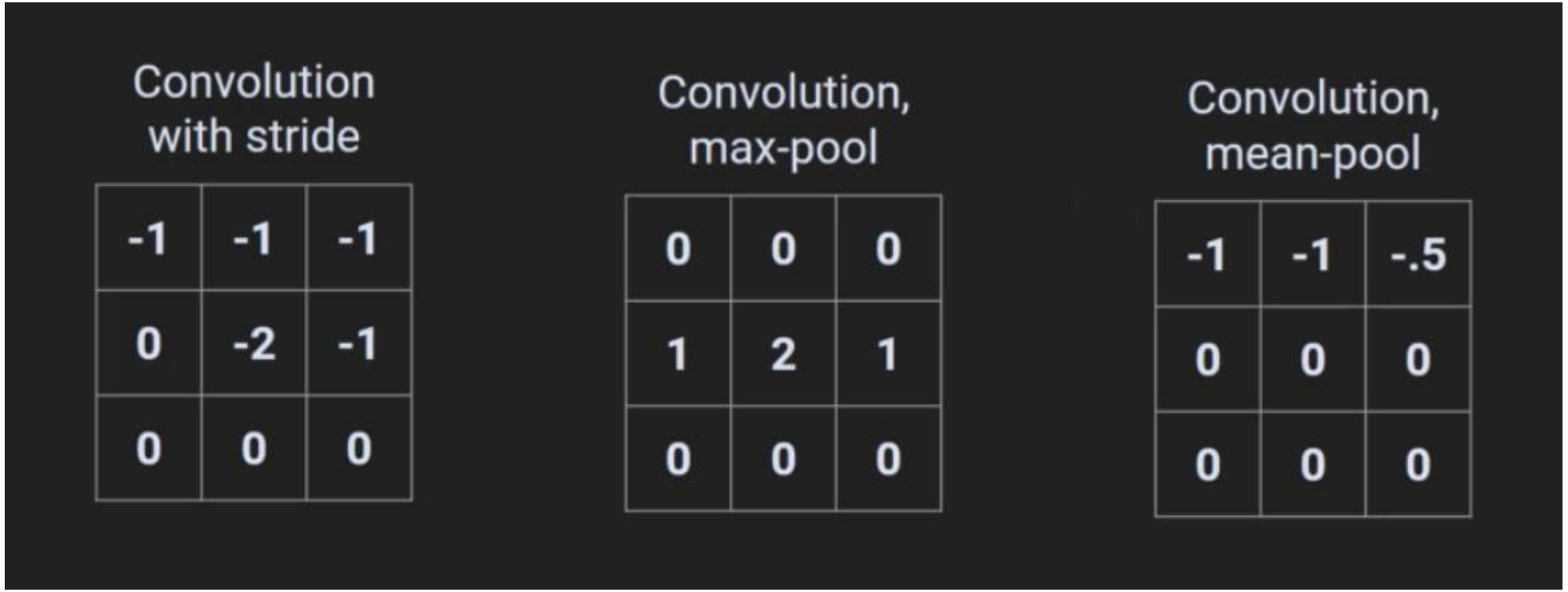


# Convolution with stride and no pooling



# Convolution and pooling, no stride

# Convolution and pooling, no stride



# Striding through the pool

- Pooling
  - Computationally Fast
  - No parameters
  - Kernel spans a smaller area (smaller receptive fields)
  - Highly Stable
- Stride
  - Somewhat slower
  - Learned parameters
  - Kernel spans a large area (large receptive fields)
  - Can be unstable in complex architecture

# Striding through the pool



## STRIVING FOR SIMPLICITY: THE ALL CONVOLUTIONAL NET

**Jost Tobias Springenberg*, Alexey Dosovitskiy*, Thomas Brox, Martin Riedmiller**
Department of Computer Science
University of Freiburg
Freiburg, 79110, Germany
{springj, dosovits, brox, riedmiller}@cs.uni-freiburg.de

### ABSTRACT

Most modern convolutional neural networks (CNNs) used for object recognition are built using the same principles: Alternating convolution and max-pooling layers followed by a small number of fully connected layers. We re-evaluate the state of the art for object recognition from small images with convolutional networks, questioning the necessity of different components in the pipeline. We find that max-pooling can simply be replaced by a convolutional layer with increased stride without loss in accuracy on several image recognition benchmarks. Following this finding – and building on other recent work for finding simple network structures – we propose a new architecture that consists solely of convolutional layers and yields competitive or state of the art performance on several object recognition datasets (CIFAR-10, CIFAR-100, ImageNet). To analyze the network we introduce a new variant of the "deconvolution approach" for visualizing features learned by CNNs, which can be applied to a broader range of network structures than existing approaches.

# Two reasons for image transforming

- #1
  - Pre-trained CNNs are coded for certain image sizes. You might need to resize your images to work, or convert to grayscale
- #2
  - Transforming images changes raw pixel values without changing the image information. Transforms are thus a way to increase the total amount of data.

# Two reasons for image transforming

- #1

**1.Pre-trained CNNs are coded for certain image sizes.**

- This statement refers to the requirement of many pre-trained CNN models that input images be of a specific size. CNNs often expect a fixed-size input because their internal architecture, which includes filters and layers, is designed to work with data of that exact dimensionality. Therefore, if the images you're working with do not match the required input size of the CNN, you might need to resize them so that each image fits the expected dimensions.

**2.You might need to resize your images to work, or convert to grayscale.**

- Resizing is the process of changing the dimensions of an image. This is commonly done to meet the input size requirements of a CNN.
- Converting to grayscale is a form of image transformation where a color image is converted into shades of gray, representing the intensity of the original colors. This is sometimes done to reduce the computational complexity, as grayscale images only contain intensity information, not color, reducing the amount of data the CNN needs to process.

- #2

**1.Transforming images changes raw pixel values without changing the image information.**

- When you transform an image (through resizing, rotating, scaling, etc.), you're altering the pixel values. For example, resizing changes the number of pixels and their arrangement, but the overall content or the information the image represents (like the objects in it) should ideally remain recognizable.

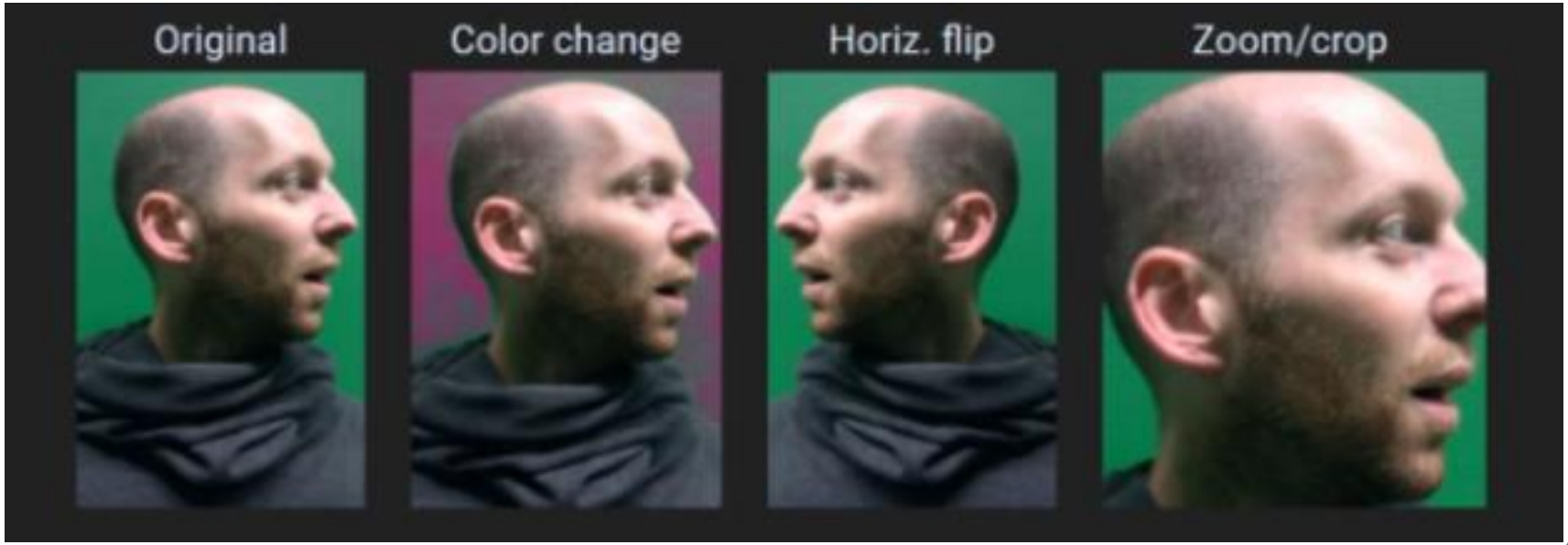
**2.Transforms are thus a way to increase the total amount of data.**

- Image transformations are often used for data augmentation, which is a technique to artificially expand the size of a training dataset by creating modified versions of images in the dataset. This can help improve the robustness of a model by providing it with more varied examples to learn from, without actually collecting new data. The "total amount of data" refers to the diversity of data points (images, in this context) that the model can learn from.

# Two reasons for image transforming

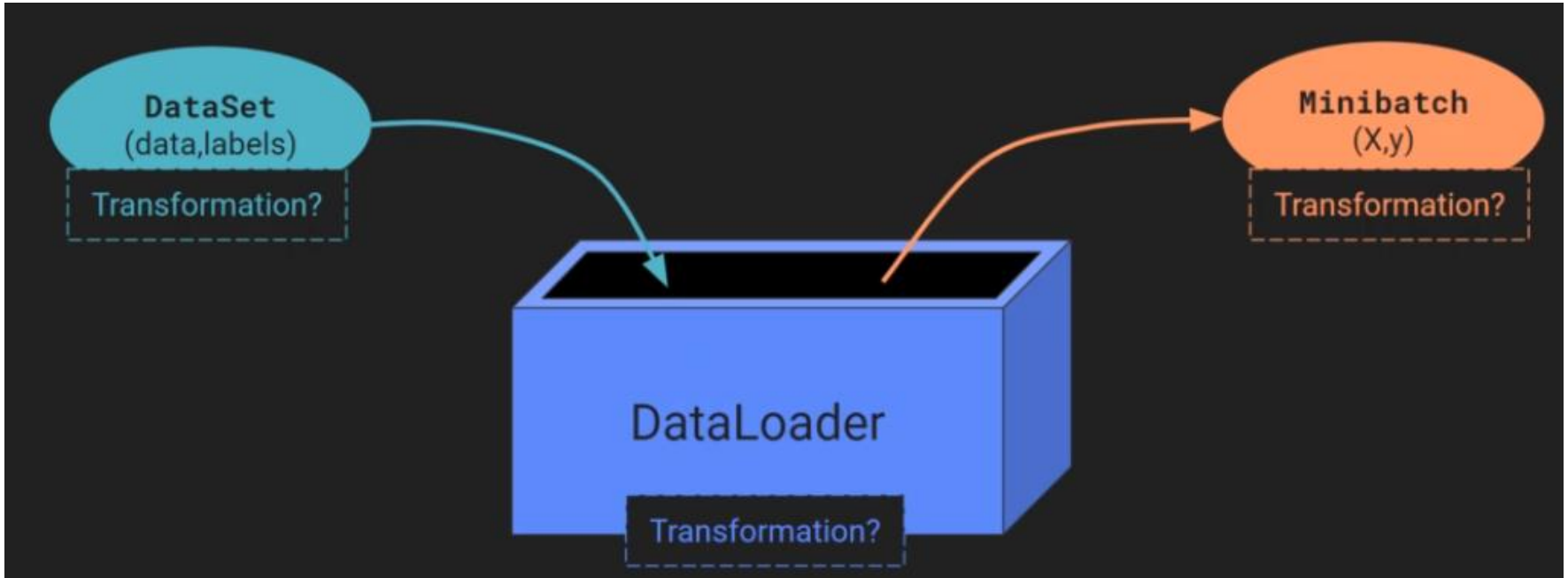
- Data Augmentation
  - Add More data as slightly modified version of the existing data (which is usually the case for the image processing)

- #Please work on the python practice

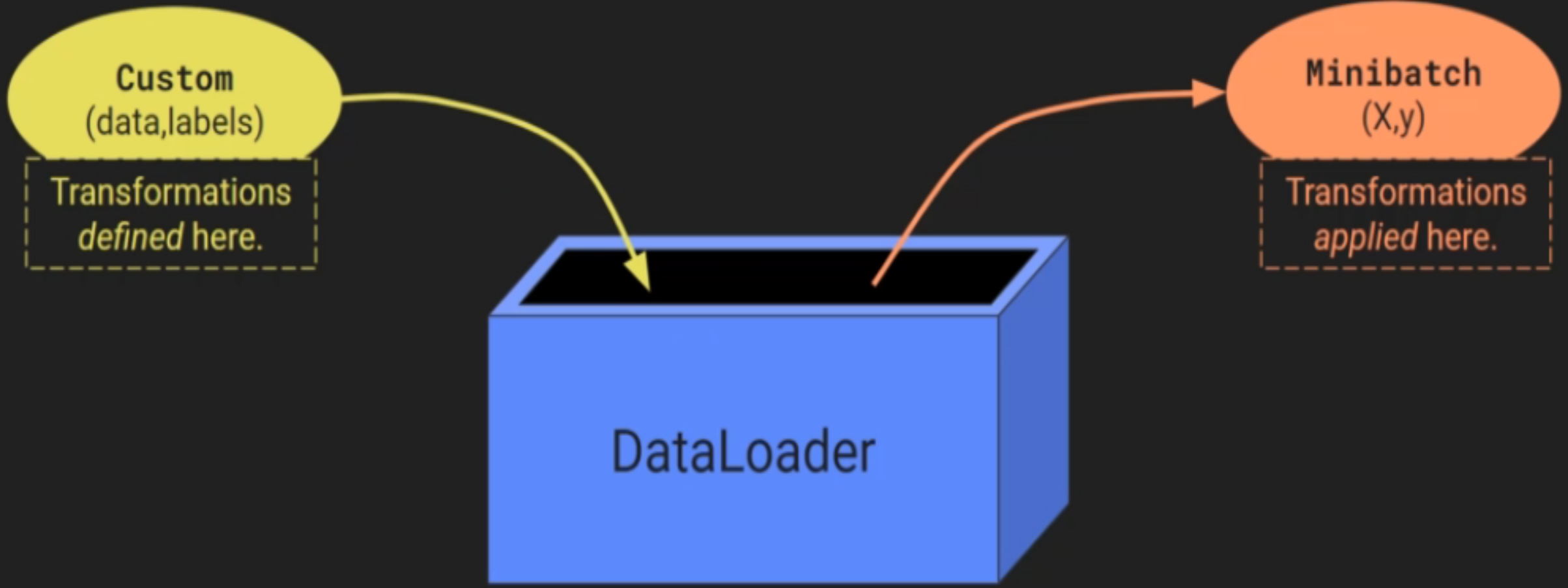


# Two reasons for image transforming

- Data Augmentation
  - Add More data as slightly modified version of the existing data (which is usually the case for the image processing)
- #Please go to the python practice



# Two reasons for image transforming



# Order of operation when transformation is applied

- ❑ Import data
- ❑ Create Custom dataset class
- ❑ Define the transformation
- ❑ Create your DataSet with data and transformation
- ❑ Create DataLoader (Same as usual)

# Three types of layers in a CNN

**1.Convolution Layer**

- **Purpose**: To learn filters (kernels) to create feature maps.
- This layer applies a number of filters to the input. Each filter slides over the input image (a process known as convolution) to produce a feature map, which highlights where certain features (like edges or textures) are detected in the image.

**2.Pooling Layer**

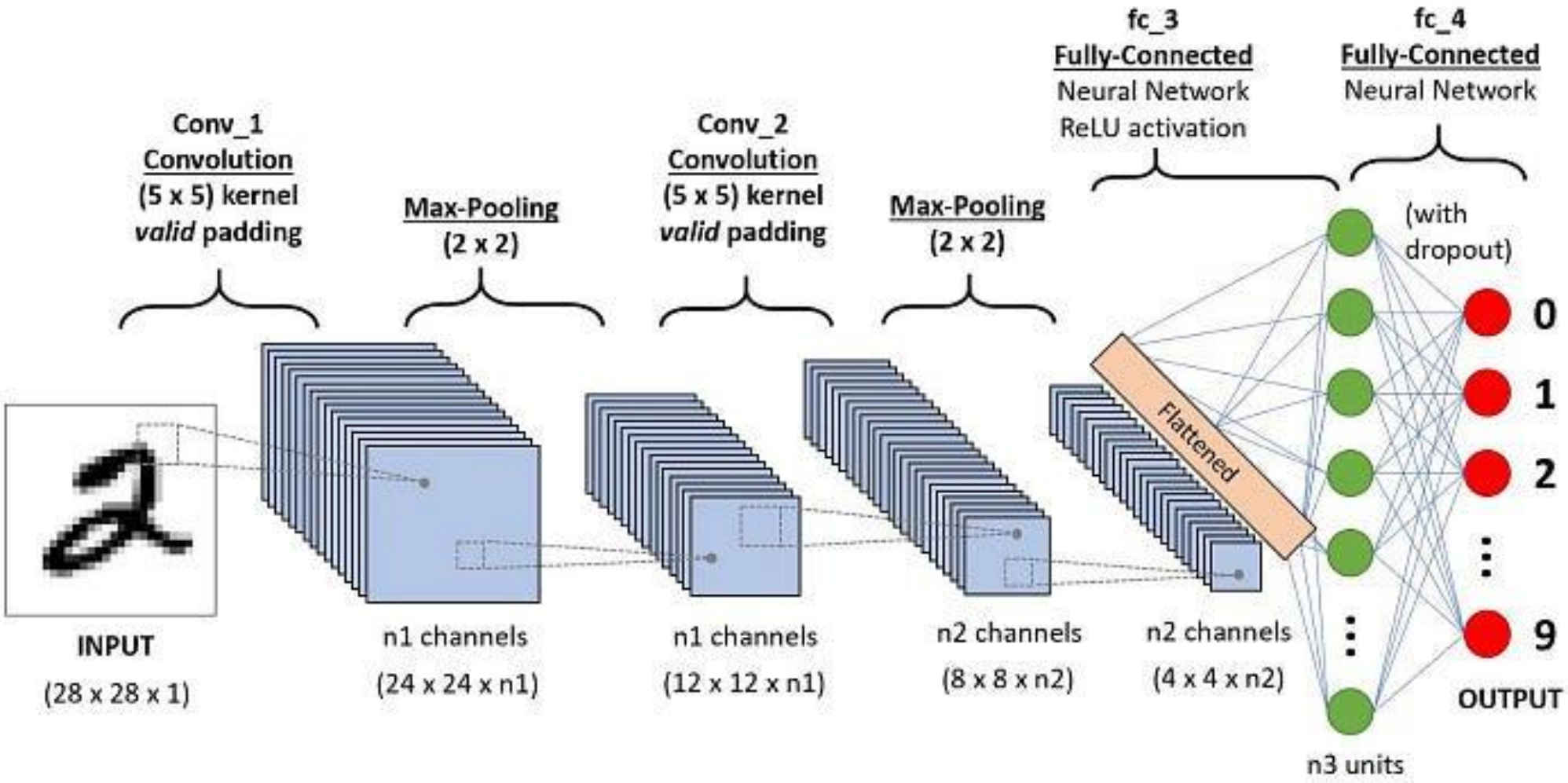
- **Purpose**: To reduce dimensionality and increase receptive field size.
- Pooling (often max pooling) reduces the spatial size of the feature maps, thus reducing the number of parameters and computation in the network. It also helps make the detection of features invariant to scale and orientation changes. The receptive field is the region in the input space that a particular CNN's feature is looking at (i.e., it can be thought of as the "window" of the input data that is being processed by the filter).

**3.Fully Connected Layer**

- **Purpose**: Prediction (categorical and/or continuous)
- In these layers, neurons have full connections to all activations in the previous layer, as seen in regular neural networks. Their role is to take the high-level features learned by the convolutional and pooling layers to perform the actual task such as classification or regression. In a classification task, the last fully connected layer would have the same number of neurons as the number of output classes and typically uses a softmax activation function to output probabilities of the classes.

# Three types of layers in a CNN

Link

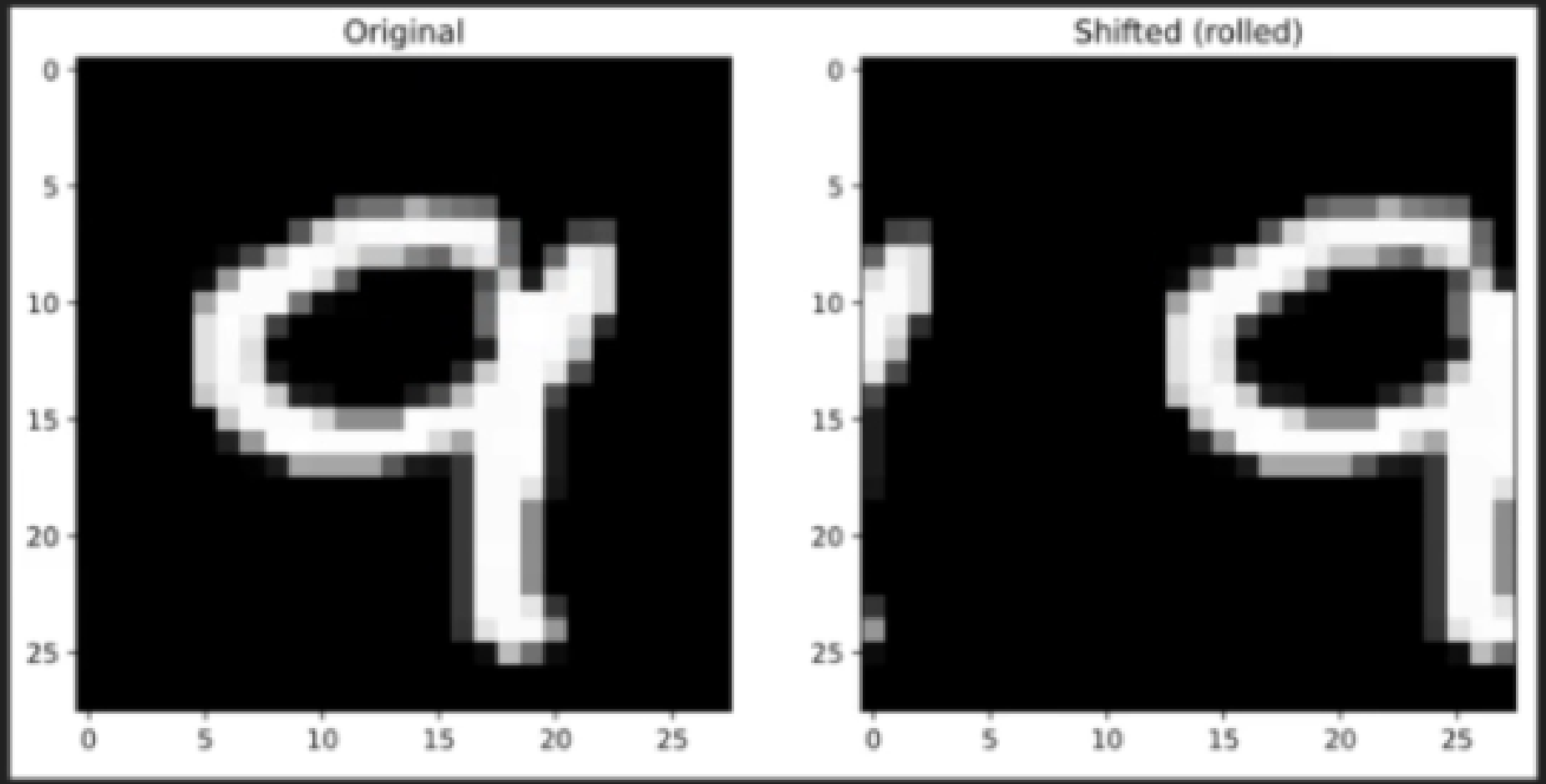


# Three types of layers in a CNN

**1.Image resolution (number of pixels) decreases**: This aspect of the diagram shows that as an image moves through layers of a CNN, it often goes through pooling layers, which reduce the spatial resolution of the image (i.e., the image is represented with fewer pixels). The purpose of this is to reduce the computational load and to create an abstracted version of the input that captures the essential features while discarding the less important details.

**2.Representation resolution (number of filters) increases**: As the network depth increases, the number of filters (also known as feature detectors) increases. Each layer of filters tends to extract more complex and abstract features from the image. Early layers might detect simple edges or textures, while deeper layers may detect parts of objects or even whole objects. This is often referred to as the depth or the complexity of the feature representation.

# Shifting an image/Python Practice



# GausNet (2D Gaussians)

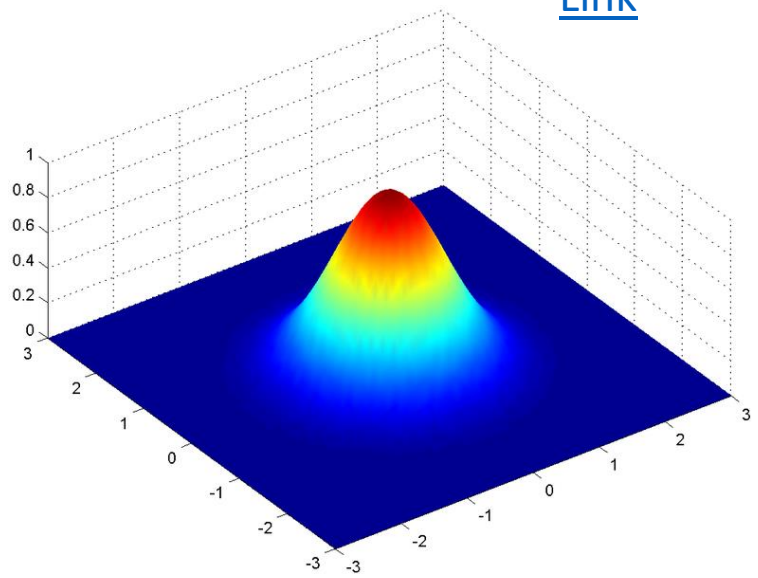
In 1D:

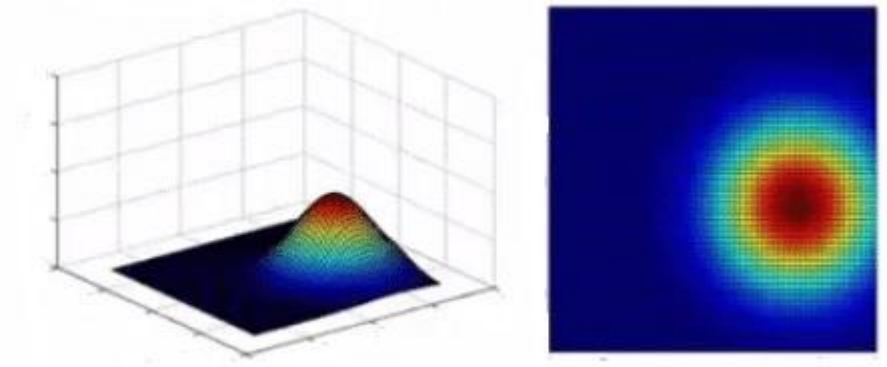
$$g_\sigma(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{x^2}{2\sigma^2}\right)$$

In 2D:

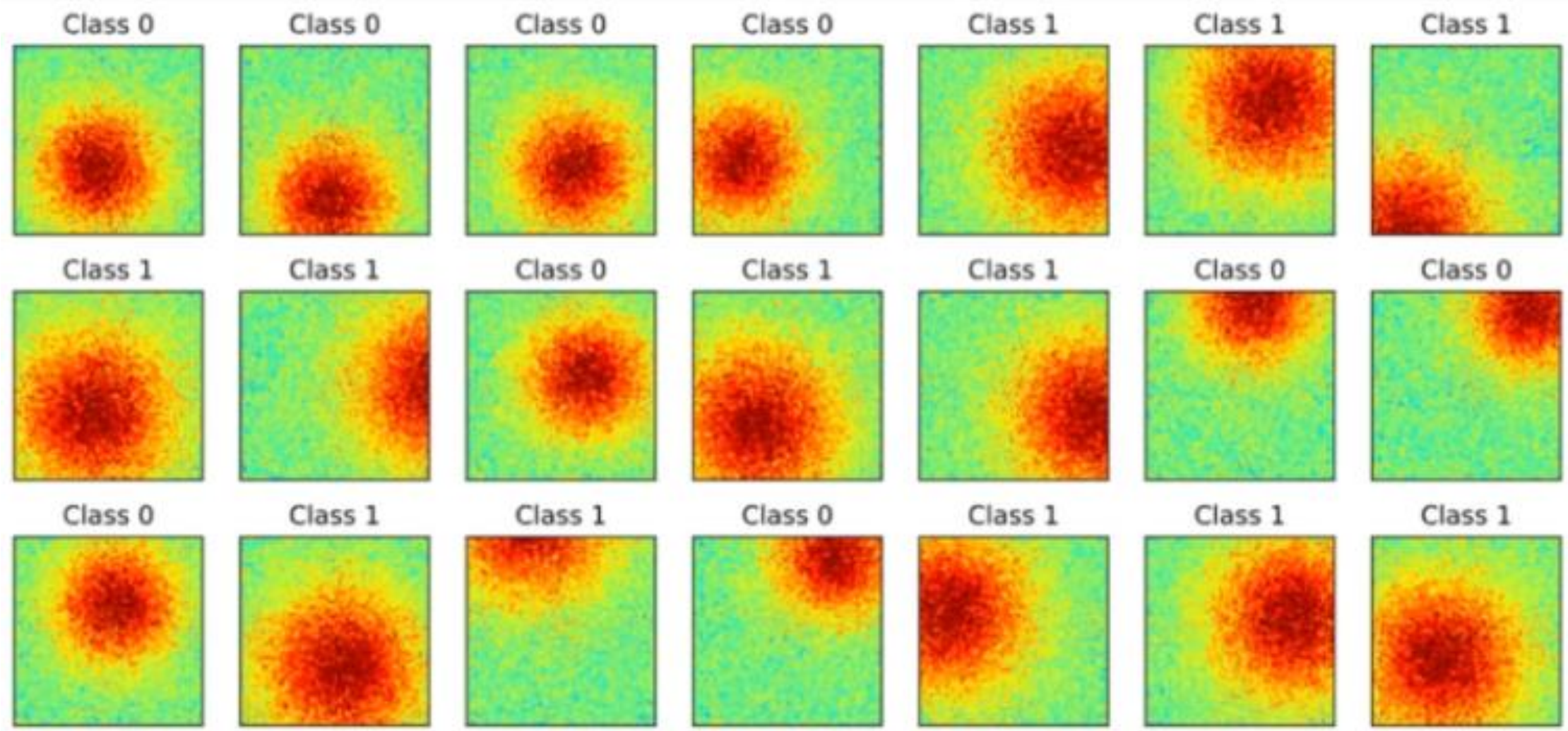
$$G_\sigma(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

Link

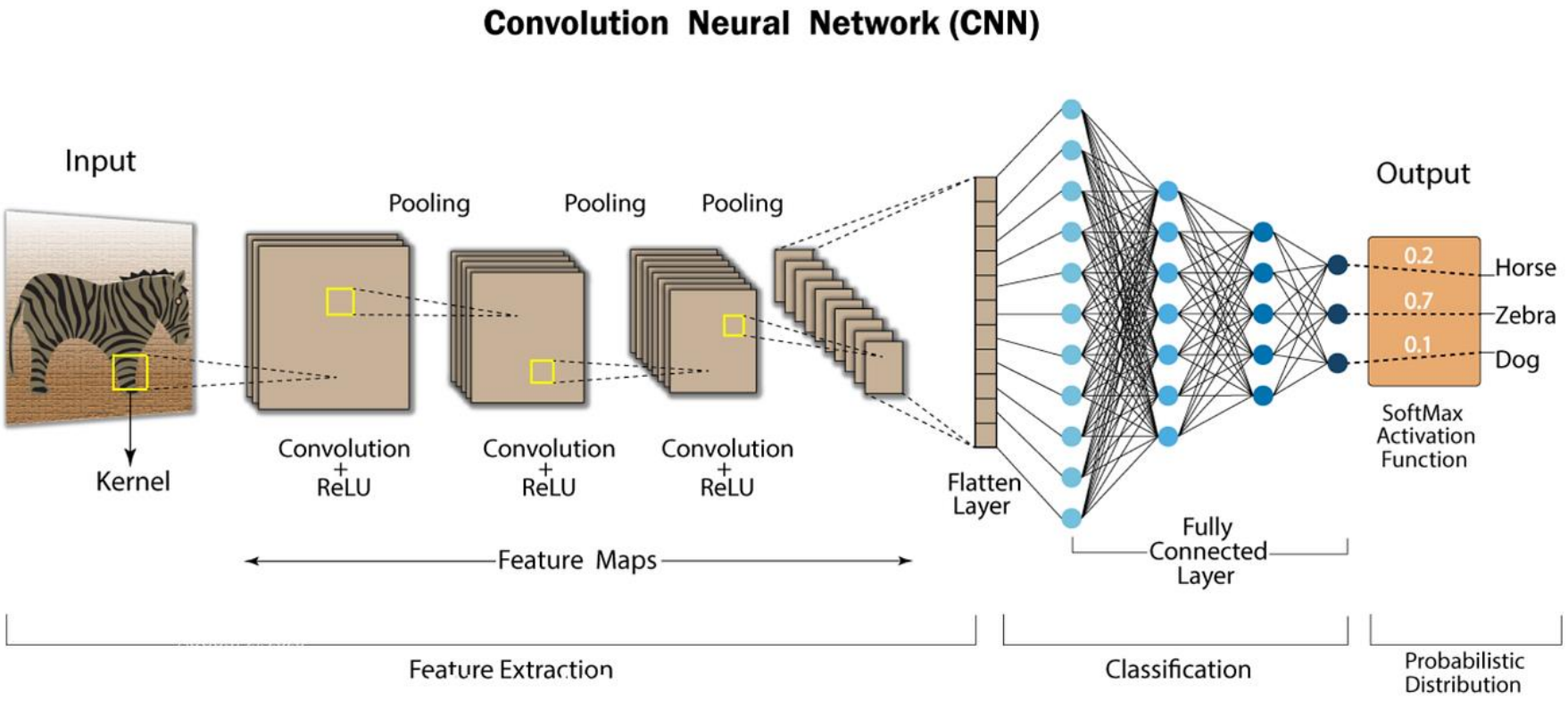


# GausNet (2D Gaussians)



# GausNet (2D Gaussians)




Link

# GausNet (2D Gaussians)

```python
class gausnet(nn.Module):
  def __init__(self):
    super().__init__()

    # all layers in one go using nn.Sequential
    self.enc = nn.Sequential(
        nn.Conv2d(1,6,3,padding=1),  # output size
        nn.ReLU(),                   # note that r
        nn.AvgPool2d(2,2),           # output size
        nn.Conv2d(6,4,3,padding=1),  # output size
        nn.ReLU(),                   #
        nn.AvgPool2d(2,2),           # output size
        nn.Flatten(),                # vectorize c
        nn.Linear(22*22*4,50),       # output size
        nn.Linear(50,1),             # output size
    )

  def forward(self,x):
    return self.enc(x)
```

| Class function | Analogies |
| --- | --- |
| def __init__(): | Objects, nouns, layers, characters in the story |
| def forward(): | Actions, verbs, operations, plot |

# Hardcoding and Softcoding

## Hardcoding

```
[0]

    # 100s of lines of code here...

    result = 4*5
```

## Softcoding

```
[0] param = 5

    # 100s of lines of code here...

    result = 4*param
```

# Hardcoding and Softcoding

**1.Hardcoding**: This term refers to the direct embedding of data into the source code rather than obtaining the data from external sources or generating it algorithmically. In the image, **result = 4*5** is a hardcoded operation because the numbers are directly written into the code. This means that if the values needed to be changed, the source code itself would need to be modified.

**2.Softcoding**: This is a more flexible approach where values are stored as parameters or variables that can be easily modified without changing the core source code. In the example, **param = 5** is set at the beginning, and then **result = 4*param** uses this parameter. This way, to change the operation, one would only need to change the value of **param** without touching the 'result' calculation logic.

# Hard Code Parameters

```
# conv1
self.conv1 = nn.Conv2d(1,6,3,padding=1)
# output size: (91+2*1-3)/1 + 1 = 91
# post-pooling: 91/2 = 45

# conv2
self.conv2 = nn.Conv2d(6,4,3,padding=1)
# output size: (45+2*1-3)/1 + 1 = 45
# post-pooling: 45/2 = 22

# fc1
self.fc1 = nn.Linear(22*22*4,50)
```

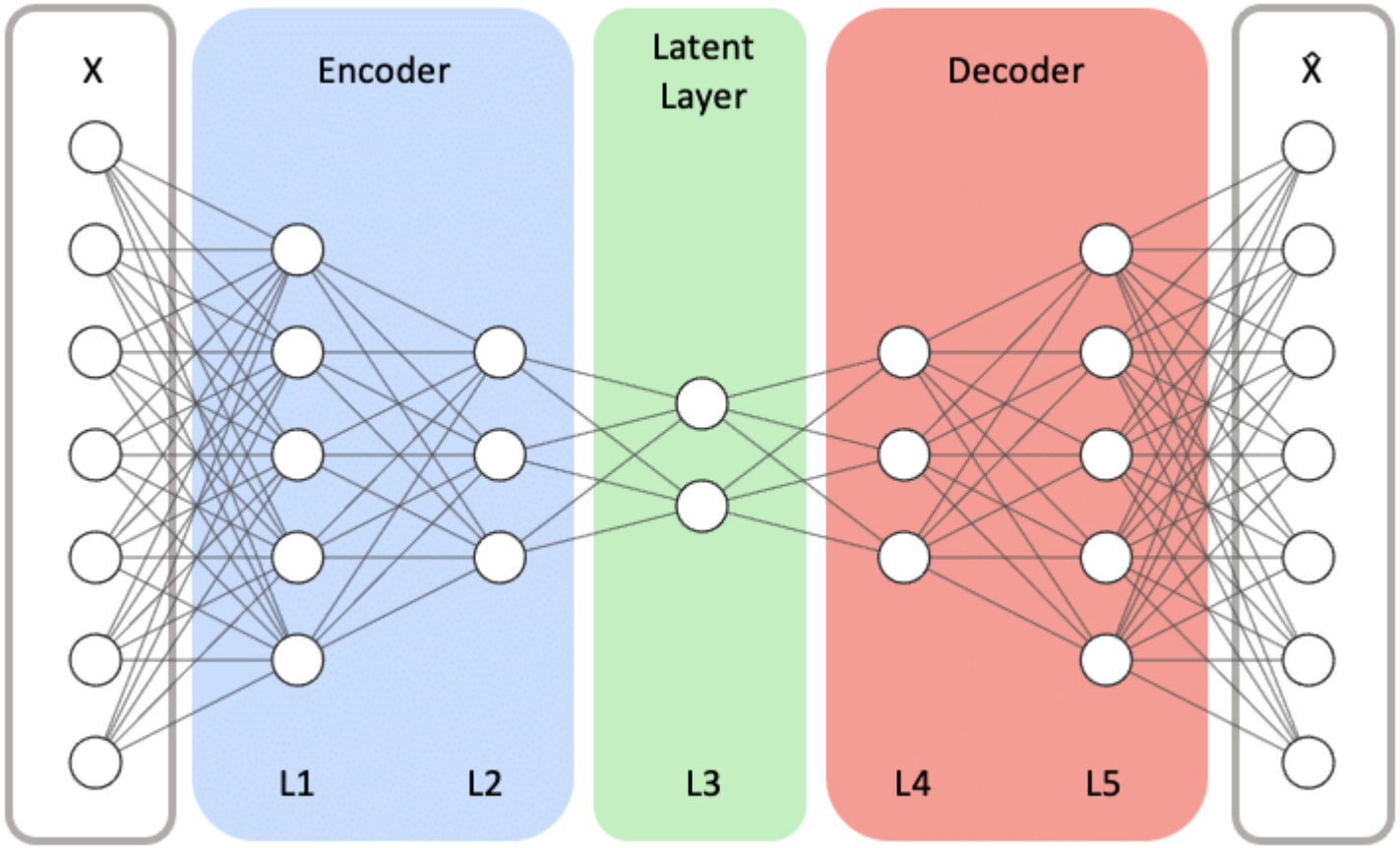
# Hard Code Parameters

**1.conv1**: The first convolutional layer is created using **nn.Conv2d**. The parameters **(1, 6, 3, padding=1)** define the layer with one input channel, six output channels (or filters), a kernel size of 3, and padding of 1. The comment calculates the output size of this layer, considering the input size (presumably 91 in one dimension), the kernel size, and the padding. Post-pooling, the size is halved, indicating the use of a 2x2 pooling layer that reduces the spatial dimensions by a factor of 2.

**2.conv2**: The second convolutional layer, also defined by **nn.Conv2d**, has parameters **(6, 4, 3, padding=1)**, which means it takes the six output channels from **conv1** as its input channels, has four output channels, a kernel size of 3, and padding of 1. The output size and post-pooling size are again calculated, with the post-pooling size being half of the output size due to pooling.

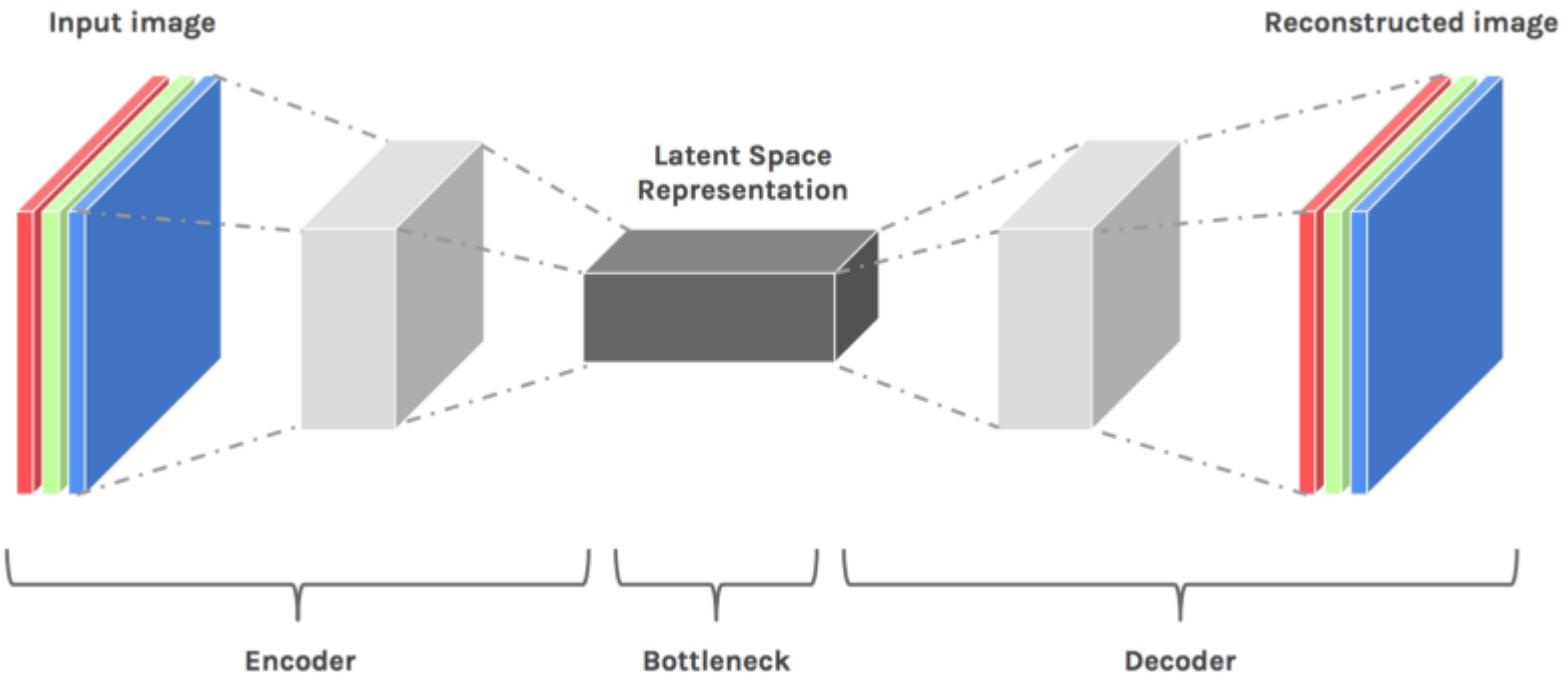
**3.fc1**: This is a fully connected layer defined by **nn.Linear**. The parameters **(22*22*4, 50)** indicate that the layer is expecting a flattened input of size 22 by 22 by 4 (the number of output channels from **conv2**), which is then connected to 50 output features.

# Autoencoder Architecture




Link

# Autoencoder Architecture




Link

Reference

# Autoencoder Architecture

1.Auto = self

- This part denotes that "auto" is a prefix coming from Greek, meaning "self" or "same". It implies an operation or process that is performed independently without external intervention.

2.Encode = convert into a different form (typically simpler or reduced-dimensional)

- Here, "encode" means to transform data into a different format or structure. The purpose of encoding in the context of autoencoders is typically to reduce the dimensionality of the data, creating a simpler representation that still captures the essential features.

3.Autoencoder = a system that teaches itself how to encode information.

- Combining the two components, an "autoencoder" is described as a system that autonomously learns to encode information. In the field of machine learning and neural networks, an autoencoder is a type of artificial neural network used for unsupervised learning of efficient codings, with the goal of reducing the feature space.
- An autoencoder learns to compress (encode) the input into a lower-dimensional representation and then reconstruct (decode) the input data from this representation as accurately as possible. This process allows the model to capture the most important features in the data.

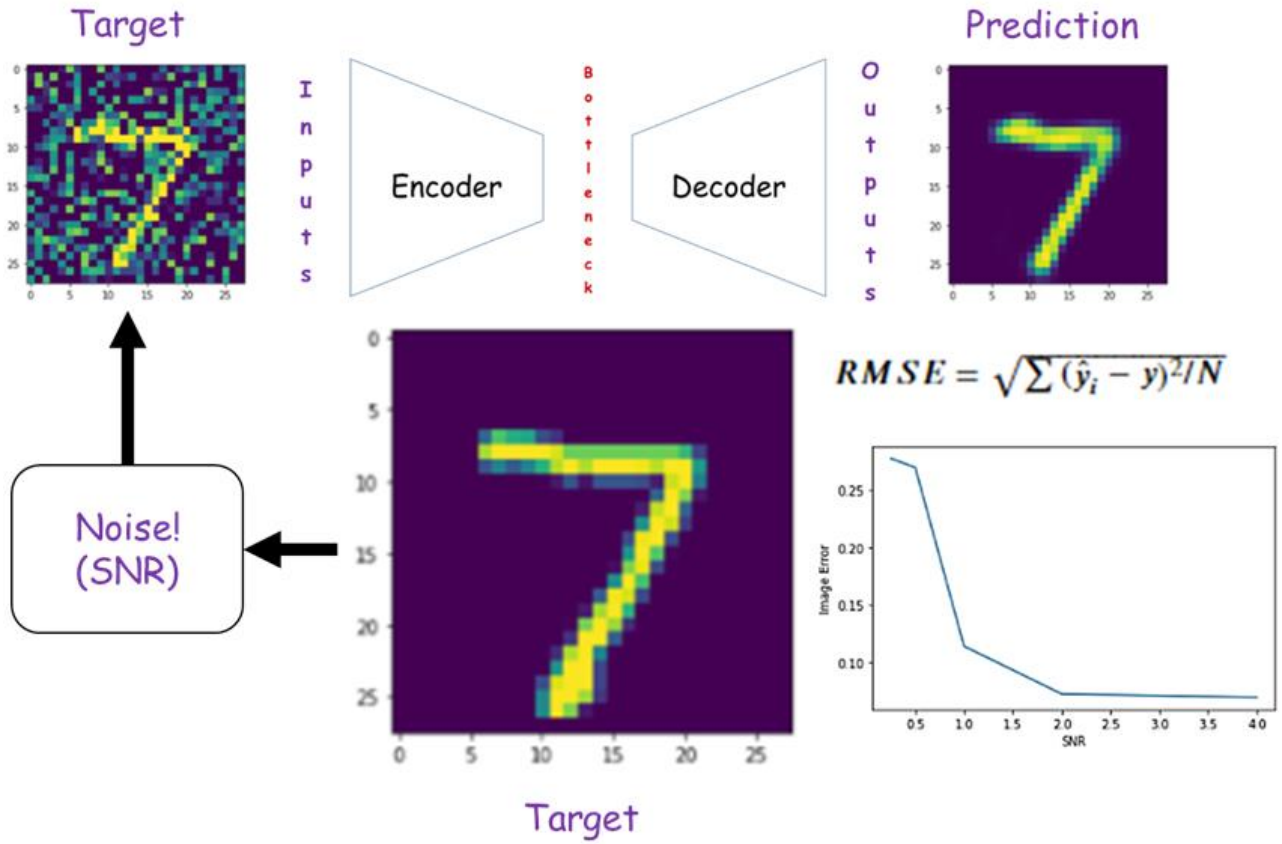
# Autoencoder Architecture

**1.Encoder**: The left portion of the diagram (typically depicted with layers contracting to a narrow point) represents the encoder. It takes the input data and processes it through one or more layers (with decreasing size) to generate a compressed representation of the input data. This is often called the "bottleneck" or latent space, where the data has been encoded to a lower-dimensional form.

**2.Decoder**: The right portion mirrors the encoder, with layers expanding from the narrow point back to the original input size. The decoder's job is to reconstruct the input data from its encoded form, ideally with minimal loss of information.

The central, narrowest layer (often depicted in blue in diagrams) is the code or latent space representation, which is a compact representation of the input data. Autoencoders are trained to minimize the difference between the input and the reconstructed output, often using a loss function that measures the difference, such as mean squared error.

# Autoencoder Architecture- Loss function




Link

# Autoencoder Architecture- Loss function

# Autoencoder Architecture

1.Data compression or dimension-reduction:

- Autoencoders can learn to represent input data in a compressed form, reducing the number of dimensions without significant loss of information, which is useful in reducing the computational cost for storage and processing.

2.Data cleaning (denoising, despeckling, occlusion):

- They can be used to remove noise from data. A denoising autoencoder, for example, is trained to use a corrupted version of data as input and output a clean, denoised version.

3.Feature extraction:

- Autoencoders can learn to identify and encode the most salient features of the data, which can then be used for further tasks like classification or clustering.

4.Anomaly/fraud detection:

- By learning a normal pattern of data, autoencoders can be used to detect outliers or anomalies, which do not conform to the learned representation. This is useful in scenarios like fraud detection, where anomalies need to be identified and investigated.

5.Pretraining deep or complex models:

- Autoencoders can be used to pretrain the weights of neural networks in an unsupervised manner, which can lead to improved performance and faster convergence when the network is later fine-tuned with supervised learning on a specific task.

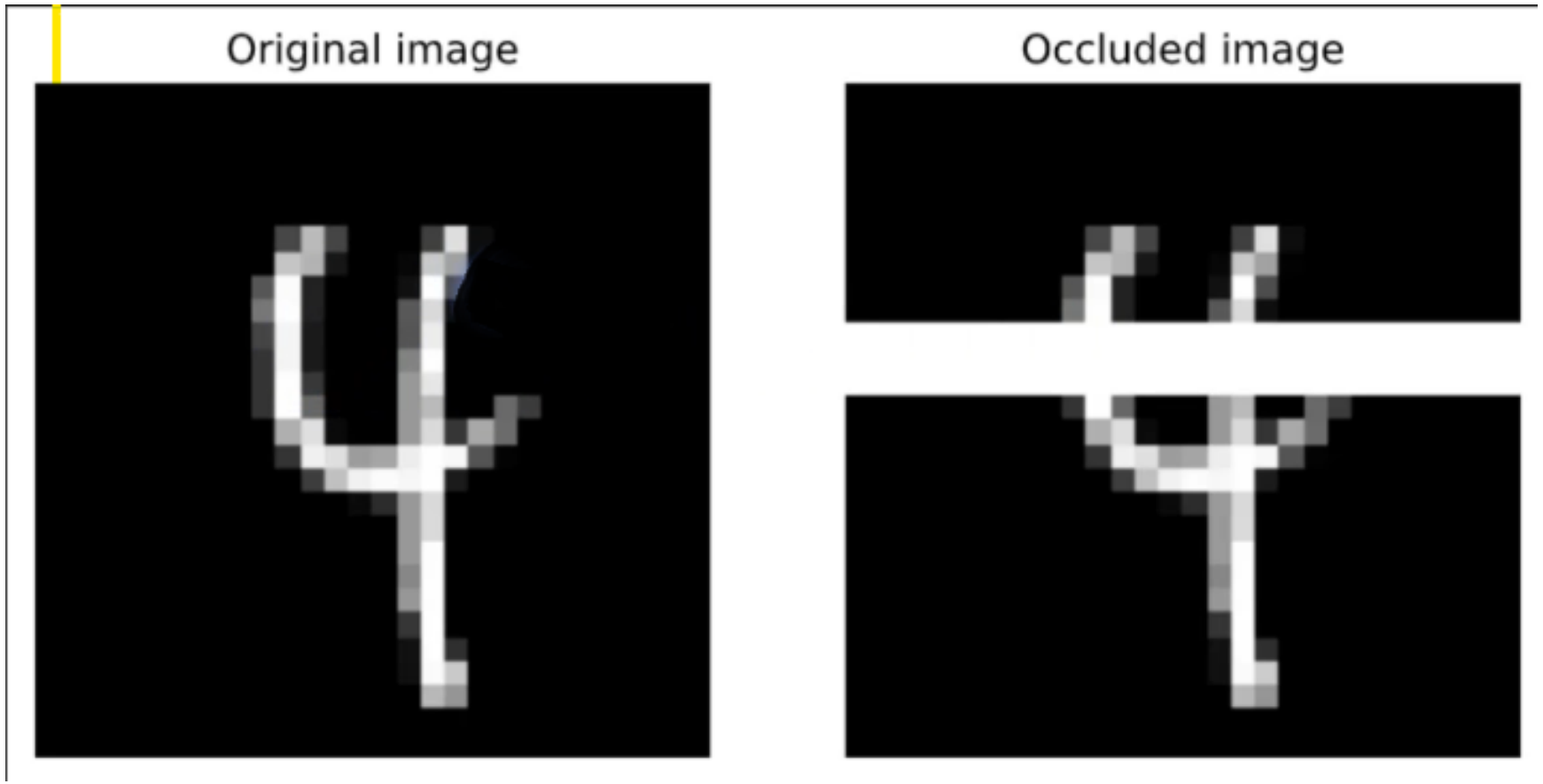
# Autoencoder Architecture- Occlusion


(a) Occluded Face 1

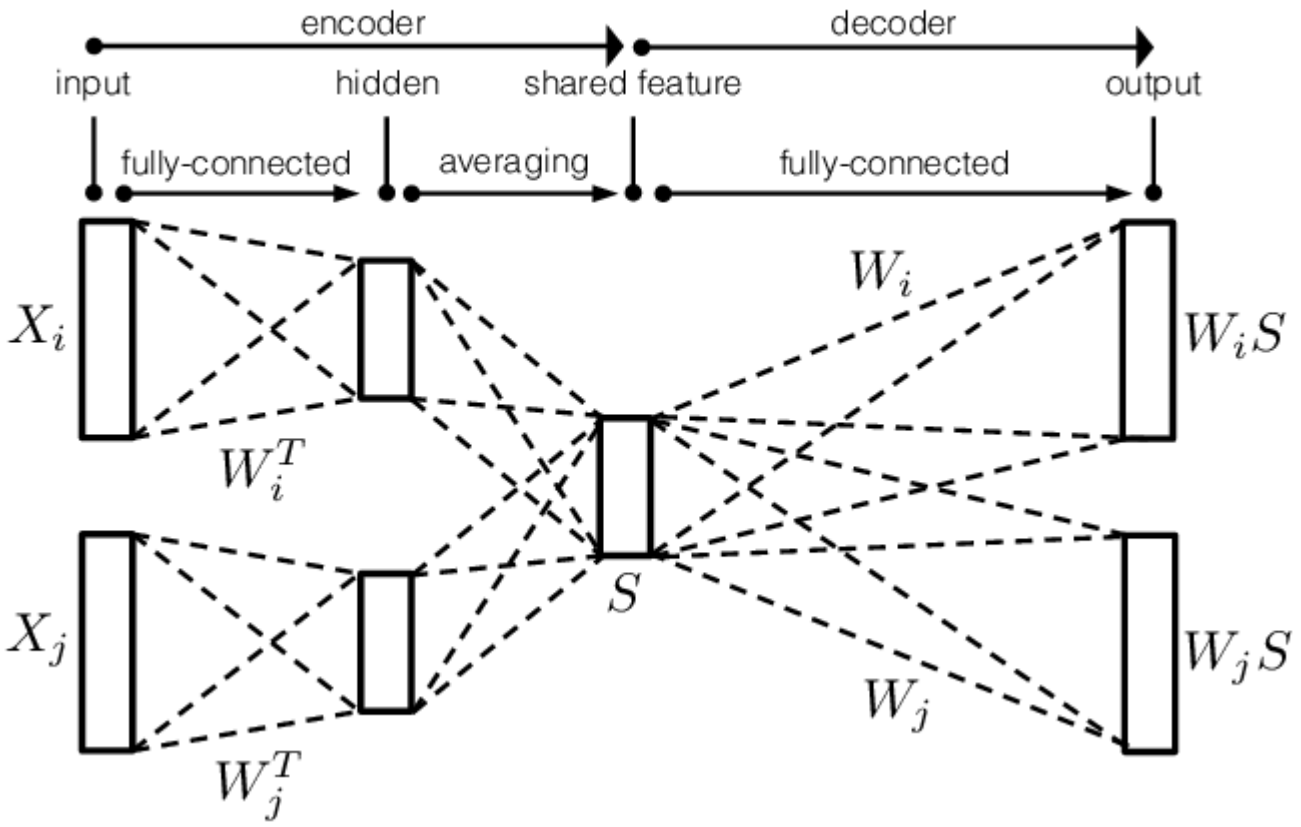

(c) Person

Link

# Autoencoder Architecture- FFN-AEs



# Autoencoder Architecture- AutoEncoder tied weights




Link

# Autoencoder Architecture

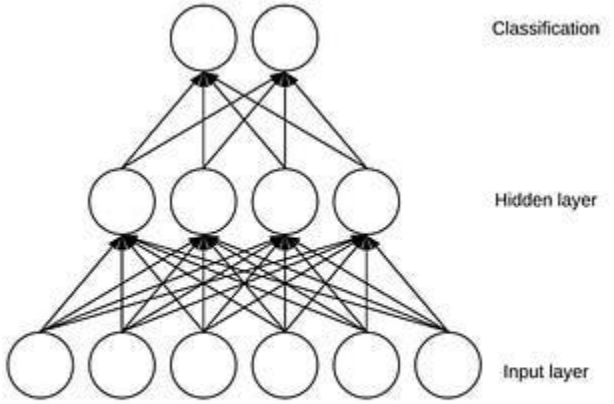
1.This is the L1 loss function, also known as Mean Absolute Error (MAE). It calculates the mean of the absolute differences between the predicted values ($y$^) and actual values ($y$). This loss function is often used when the distribution of the target variable is not Gaussian, as it is less sensitive to outliers than the L2 loss.

2.This function appears to be a variation of the L2 loss function (Mean Squared Error, MSE) with an additional term. The first part is the standard MSE, taking the mean of the squared differences between predicted and actual values. The second term, however, is unusual—it's the mean of the sum of all predicted values ($y$^), taken as an absolute value. This additional term does not typically appear in loss functions and could potentially penalize models based on the magnitude of their predictions regardless of their accuracy.

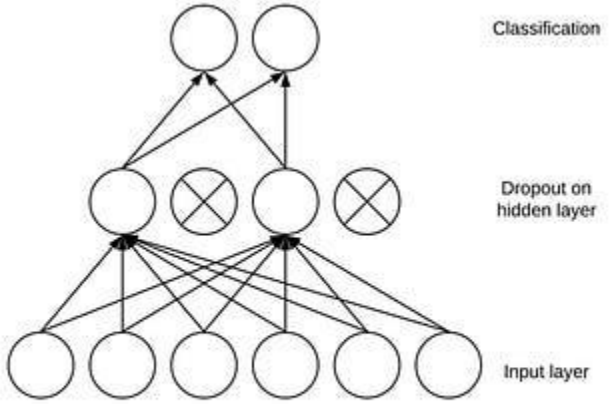
3.This loss function seems to be based on the correlation formula between the predicted values and the actual values. $\mu y$^ and $\mu y$ represent the mean of the predicted and actual values, respectively, while $\sigma y$^ and $\sigma y$ represent the standard deviation of the predicted and actual values, respectively. However, normally the correlation coefficient is a value between -1 and 1, and it's not typically used directly as a loss function without some transformation, because loss functions should generally be non-negative and should be minimized (whereas correlation can be positive, indicating a positive relationship).

# CNN-Dropout

Link




Without Dropout




With Dropout

All deep learning models == data preprocessing + fully-connected model

nvolution extract features? dot product between matrixes can show similarity: kernel can extract important information through convolution calculations