

# SCRIBE NOTES

Date: 10th Oct, '23 ; Tuesday

S no.	Name 1	Roll no.
<u>1</u>	Srihitha Mallepally	2021101043
<u>2</u>	Smruti Biswal	2020112011
<u>3</u>	Anush Anand	2021101086
<u>4</u>	Shubhankar Kamthankar	2020114004
<u>5</u>	Shreyas Palley	2021101016
<u>6</u>	Srujana Vanka	2020102005
<u>7</u>	Kabir Shamlani	2021101124
<u>8</u>	Nipun Goyal	2021102029

## Classification

Consider the task of image classification. In machine learning, we would have a labeled dataset of images for classification. This dataset should have images of the different classes that we want to classify. To train a classifier on this task, we follow these steps in general:

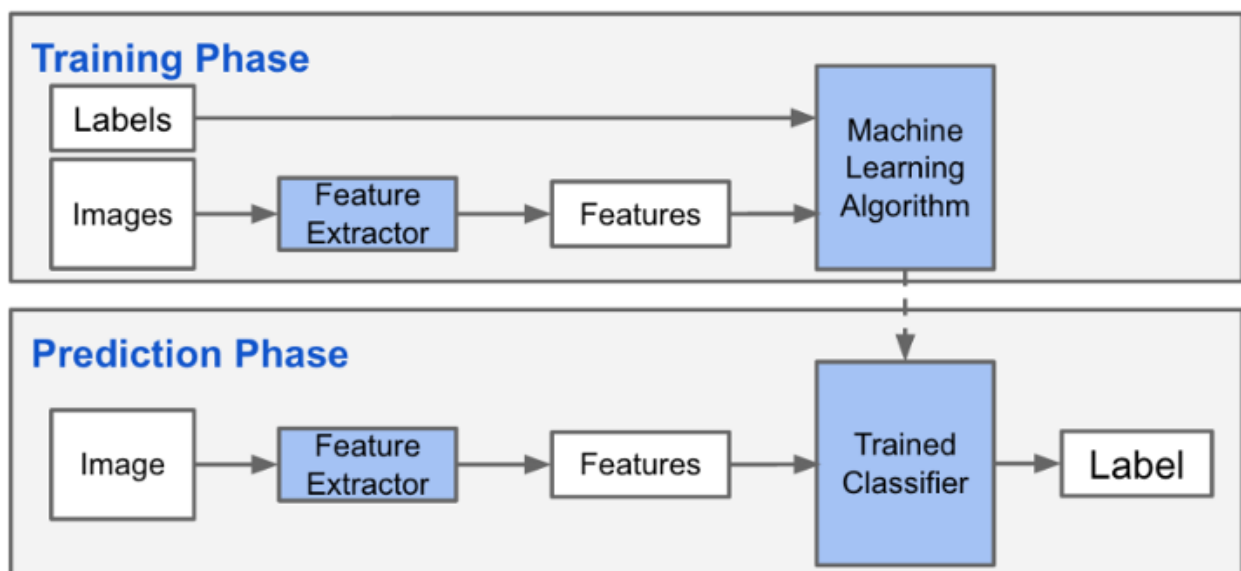
1. **Training Phase:** The primary goal of this phase is to teach the model to recognize patterns and make accurate predictions. This is achieved by exposing the model to labeled data and adjusting the parameters and hyper-parameters to minimize the loss function chosen. Simply put, this phase is learning of the matrix weights across the parameters of the model so that it fits classifying labels. **Key activities:**
  - a. Data Collection

- b. Preprocessing
- c. Feature Extraction
- d. Defining the architecture of the model
- e. Loss function selection
- f. Optimization
- g. Model evaluation

2. **Prediction Phase:** In this phase, the trained model is used to make predictions or classifications on new, unseen data. This phase focuses on applying the knowledge learnt during training to make inferences on the data. During this phase, there is no further adjustment of the model's parameters (which are frozen) **Key activities:**

- a. Data preprocessing of the new input data
- b. Feeding the data through the trained model
- c. Extracting predictions or classifications (Label)

Find below a simplified block diagram of the same:



## Feature Vectors:

Feature vectors are numerical representations of data, often used in ML and Data Analysis. These are crucial for the following reason:

1. Numerical Representation: To try to understand intuitively why they are needed, think of this example: "You are an English speaker, and you want to convey to another person, who is a non-English speaker about the appearance of a dog. Then, you would describe it as an animal with 4 feet, a tail, droopy ears, coat color, etc." All the individual features that you chose become essential to describe the animal because these features are representative of that being. It is also useful when the other person (to whom we want to convey the idea/message) is being told that in a format that they can understand. Similarly, in case of ML models, who understand only numbers (and not speech or text or images or videos), all other formats have to be converted to their respective numerical representations, making them suitable for modelling.
2. Standardization: Feature vectors provide a standardized way to describe the data. For example, in wav2vec2.0, all variable length audio is processed into discretized speech vectors with a dimensionality of 768 each. I.e. Each dimension can, in theory, capture a different part of information corresponding to that audio.
3. Pattern recognition: By extracting relevant features from data, the feature vectors (also sometimes called as embeddings) highlight patterns and characteristics that are important for the model to understand. This is the factor that helps the model differentiate between different classes.

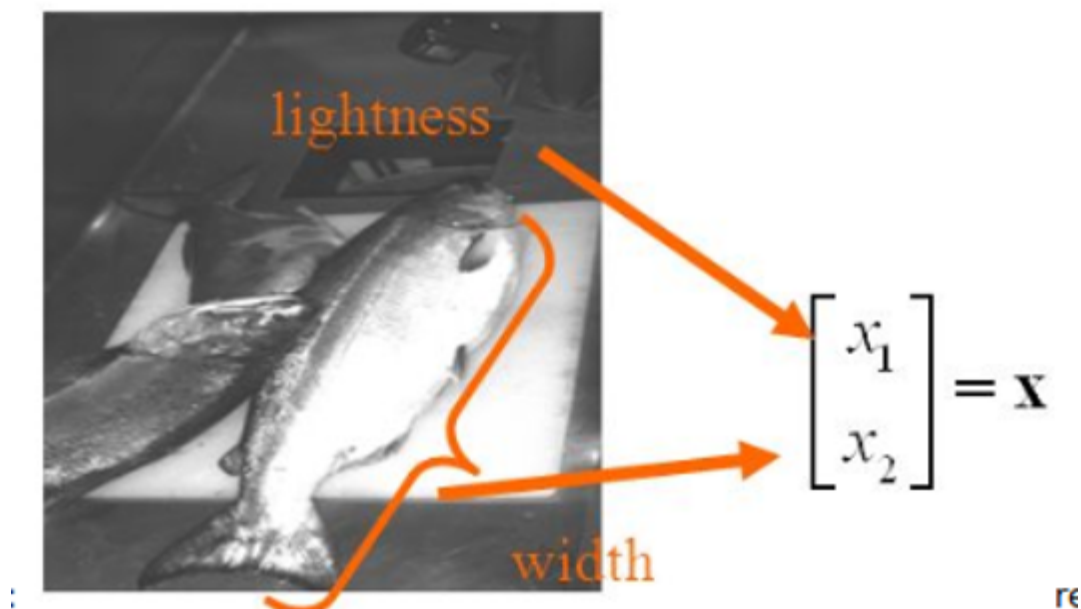
Let's take a look at why feature vectors play a crucial role in Image classification, for example:

In image classification tasks, a feature vector represents an image's content, making it possible for the ML model to recognize objects or patterns within images (more precisely, patterns of grayscale quantities of neighboring pixels). Here's how the feature vectors are used:

1. Feature Extraction: Initially, raw images are high-dimensional quantities and contain pixel values, which renders them useless to work with directly. Feature extraction, using techniques such as CNNs (Convolutional Neural Networks), gets us informative features from images. These features are representative of the information contained in the image, for eg: edges, textures, shape, color, etc.

2. Vectorization: The extracted features are transformed into a feature vector. Each element of the vector represents a particular feature. For example, if a CNN extracts 10 features, then for every value of any such feature would bear correspondence to a feature of the image, I.e you would have different values of the same feature for different images.
3. Classification: Once the images are represented as vectors, ML algorithms can be used for image classification. These models will learn to recognize patterns in the feature vectors that correspond to specific object classes.
4. Generalization: By using feature vectors, the model can generalise its learning to new, unseen images. When a new image is encountered, its features are extracted and represented as a feature vector, and the model draws conclusions about the class of this image using its previously learnt knowledge.

Example:



## MNIST DATASET

### Introduction:

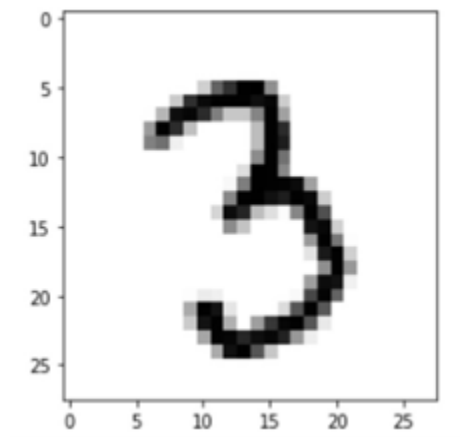
MNIST stands for **Modified National Institute of Standards and Technology**. It is a dataset of handwritten digits widely used for training and testing machine learning and

computer vision algorithms, particularly for tasks like digit recognition and image classification.



#### Description:

1. **Data:** The MNIST dataset contains 70,000 grayscale images of handwritten digits. These images are divided into two main subsets:
2. **Image Size:** Each image in the MNIST dataset is 28x28 pixels, resulting in 784 pixels per image.



3. **Labeling:** Each image is associated with a corresponding label, which is the digit it represents (0-9).
4. **Variability:** The dataset includes a variety of handwriting styles and qualities, making it a challenging but manageable dataset for digit recognition tasks.

### **Common Downstream Tasks using MNIST:**

1. Human-Machine Interaction
2. Autoencoders
3. Anomaly Detection
4. One-shot Learning.
5. Data Augmentation.
6. Transfer Learning
7. Benchmarking Algorithms.
8. Feature Engineering.
9. Image Classification
10. Digit Classification

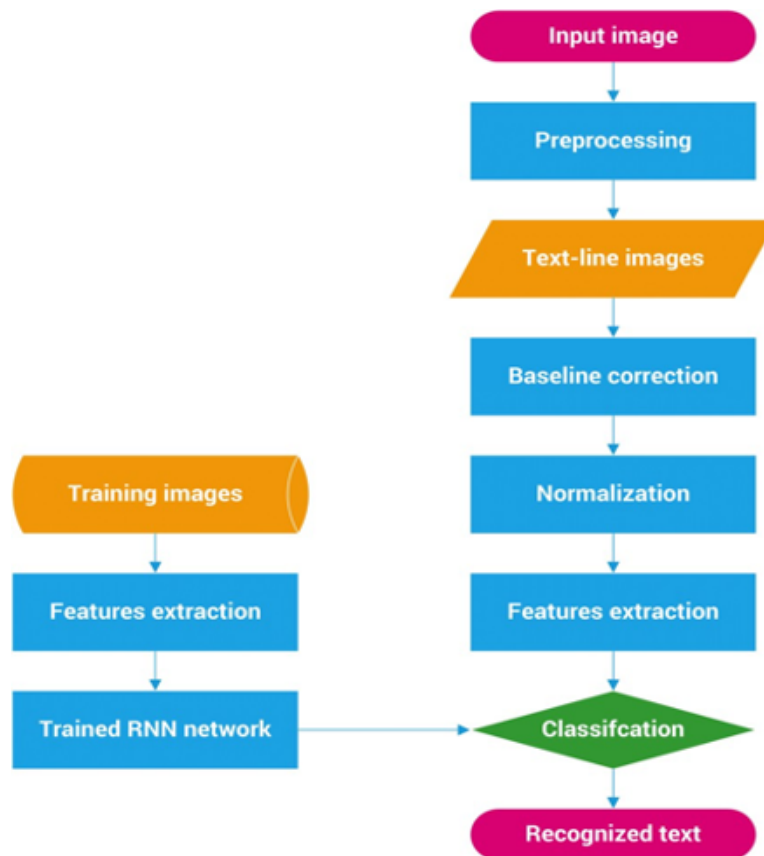
## **Digit OCR Task:**

### **Introduction:**

In the context of neural networks (NN), a "Digit OCR task" refers to the process of performing Optical Character Recognition (OCR) on handwritten or printed digits using neural networks. OCR is a technology that converts different types of documents, such as scanned paper documents, PDF files, or images captured by a digital camera, into machine-readable text. In the case of a "Digit OCR task," the goal is to recognize and convert handwritten or printed digits (0-9) into a digital format that can be processed by a computer.

### **Procedure:**

Optical Character Recognition (OCR) is a complex process that involves several steps to convert images or scanned documents into machine-readable text. Here are the key steps of an OCR algorithm in detail:



### Image Acquisition:

**Image Input:** The first step in OCR is to acquire the source image. This image can be a scanned page, a photograph, or any other image containing text that needs to be recognized.

**Preprocessing:** Images may undergo preprocessing steps like noise reduction, contrast enhancement, and image rotation to improve their quality and consistency.

### Text Localization:

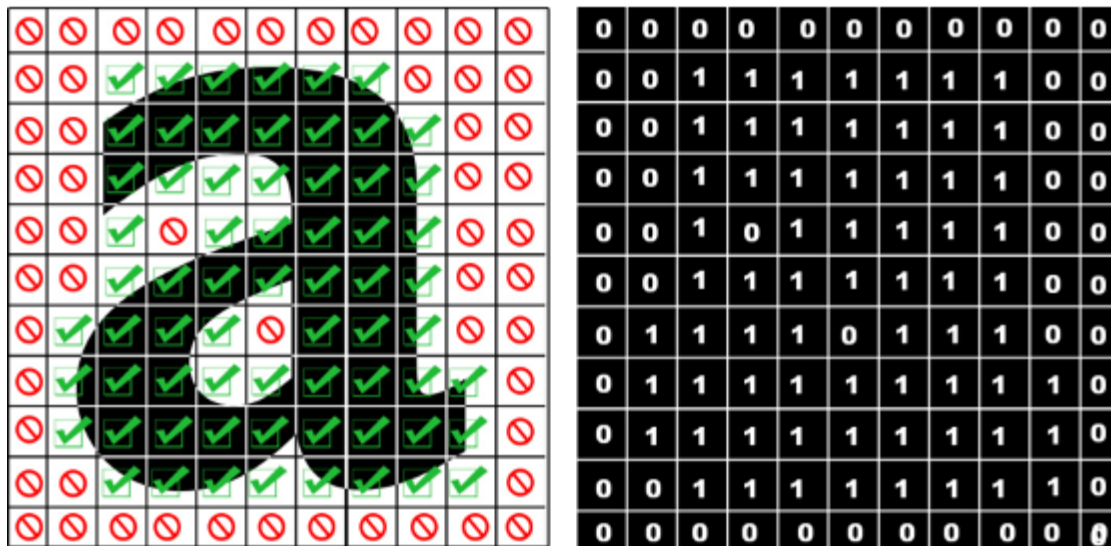
**Bounding Box Detection:** OCR algorithms often use techniques like edge detection and connected component analysis to locate text regions within the image.

**Bounding Box Verification:** These detected regions are often verified to ensure that they indeed contain text, and that extraneous objects are filtered out.

## Turning language into numbers with OCR



The matrix on the right mirrors the coordinates where letter content was or was not found, and can be calculated into a checksum. Later scans that correspond to that checksum are likely to be of the same letter. An OCR algorithm will establish an average derived space between letters in order to group characters into words, but may also have to cope with ligatures, unclear serif fonts, poor image quality, and many other retarding factors.



### Text Segmentation:

**Line Segmentation:** In cases where multiple lines of text are present, the image is segmented into individual lines to process them separately.

**Word Segmentation:** Lines are further divided into individual words, and sometimes, words into individual characters. This is especially important for cursive or overlapping handwriting.

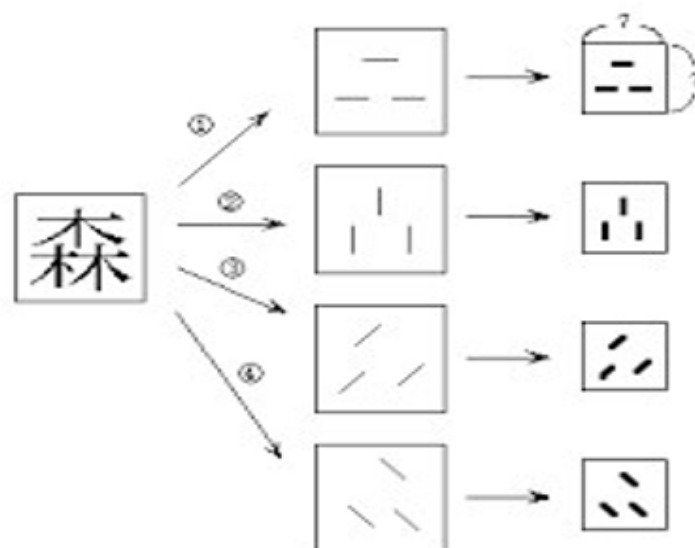




## Feature Extraction:

**Feature Detection:** Features such as edges, corners, and texture are extracted from each character or word. For handwriting recognition, these features can include stroke direction, curvature, and pen pressure.

**Feature Vectors:** The extracted features are often represented as feature vectors, which are numerical representations of the characteristics of the text elements.



## Character Recognition:

**Classifier Model:** A classifier model, such as a neural network (in particular, Convolutional Neural Networks for image-based OCR), is trained to recognize individual characters or words. The model takes the feature vectors as input and produces text predictions.

A neural network, often a Convolutional Neural Network (CNN), is used to recognize characters or words.

It extracts features from text elements and is trained on labeled data.

The model assigns probabilities to each class and selects the one with the highest probability as the recognized character or word during inference.

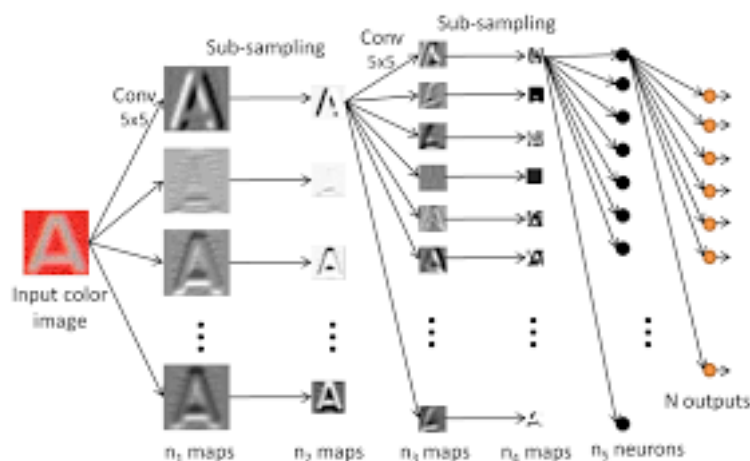
**Dictionary or Language Model:** In some cases, a language model or dictionary may be used to improve recognition accuracy by validating the recognized text against a list of valid words or characters.

Used to validate and correct OCR results.

Compares the recognized text to a dictionary of valid words or characters.

Helps in error detection and correction by suggesting alternatives based on context and known patterns.

Enhances accuracy, particularly for structured text with a predefined vocabulary.



## Post-processing:

**Error Correction:** The OCR output may contain errors due to variations in handwriting or printing. Post-processing techniques like spell-checking or language modeling can be applied to correct these errors.

**Word Context:** Word-level or sentence-level context is considered to correct ambiguous or incorrect character recognition results.

### **Output Generation:**

**Text Output:** The recognized characters or words are converted into machine-readable text and presented as the OCR output.

**Formatting:** OCR output can include information on font, size, and style, depending on the application.

### **Quality Assessment:**

**Confidence Scores:** OCR systems may assign confidence scores to their recognition results, indicating the system's confidence in each character or word.

**Error Estimation:** Errors and uncertainties are assessed and quantified, allowing for a measure of the overall accuracy of the OCR process.

### **Validation and Verification:**

**Validation:** The OCR results are often compared to a ground truth or manually verified to assess accuracy.

**Feedback Loop:** In some applications, human validation or correction may be part of the process, enabling continuous improvement of the OCR system.

### **Integration:**

**Application Integration:** The OCR output is integrated into the intended application, which could be document archiving, data extraction, or any other task that requires digitized text.

OCR algorithms can vary significantly depending on the specific use case and the complexity of the documents being processed. Advanced OCR systems may incorporate machine learning and deep learning techniques to improve recognition accuracy, especially for handwritten or challenging text. Additionally, pre-processing and post-processing steps are crucial to enhance the overall performance of the OCR system.

## Case of Colour Images:

Coloured images, as compared to grayscale images, present a more complex challenge for machine learning models due to the additional dimension representing the Red, Green, and Blue (RGB) components. These images are represented as 3D matrices, and when considering the size of a typical 200 x 200 image, we are faced with a dataset of 200 x 200 x 3, totaling 120,000 units. To build a neural network model for such data without proper techniques would require an immense number of parameters, leading to overfitting issues and an insatiable demand for training data.

### The Problem of Full Connectivity:

The concept of full connectivity implies that every unit in one layer of a neural network is connected to every unit in the next layer. In the context of image processing, this would mean connecting each pixel to every neuron in the first hidden layer. For a 200 x 200 x 3 image, we would have approximately 14.4 billion parameters, which is not only computationally expensive but also prone to overfitting. Overfitting occurs when a model becomes too complex and starts to memorise the training data instead of learning useful patterns, leading to poor generalisation on unseen data.

### Why We Don't Need Full Connectivity:

Full connectivity is impractical for images due to two main reasons:

1. **Spatial Locality:** Images have a property called spatial locality, meaning that nearby pixels are more likely to be related in terms of colour and features. When using full connectivity, we disregard this important property, treating all pixels as equally important, resulting in inefficient modelling. Example : the head of a sparrow or its body; each pixel in them is not unrelated and can be correlated with one another.
2. **Parameter Overhead:** As mentioned earlier, a densely connected network with billions of parameters is computationally expensive and requires a massive amount of data for training. It often leads to overfitting because the model captures too much noise from the training data.

### Building a Smaller Model with CNNs:

Convolutional Neural Networks provide a solution to these challenges by utilising convolutional layers. CNNs have several key components:

1. **Convolutional Layers:** These layers consist of small filters that slide across the image, capturing local patterns and reducing the number of parameters. The filters

are shared, which enforces weight sharing and preserves the spatial locality of the data.

2. Pooling Layers: Pooling layers reduce the spatial dimensions of the data, further decreasing the number of parameters. Common pooling operations include max pooling and average pooling (more about these below) .
3. Hierarchical Features: CNNs use a hierarchy of layers to capture increasingly complex features, from basic edges to more abstract representations.
4. Fully Connected Layers: While CNNs do incorporate fully connected layers at the end, they operate on the compact feature maps generated by the previous layers, significantly reducing the number of parameters.

### **The Power of CNNs:**

CNNs are designed to exploit the inherent structure in image data. By focusing on local patterns and using shared weights, they can efficiently model images with a smaller number of parameters. This design also enhances generalisation by preventing overfitting.

## **Filter And Abstraction**

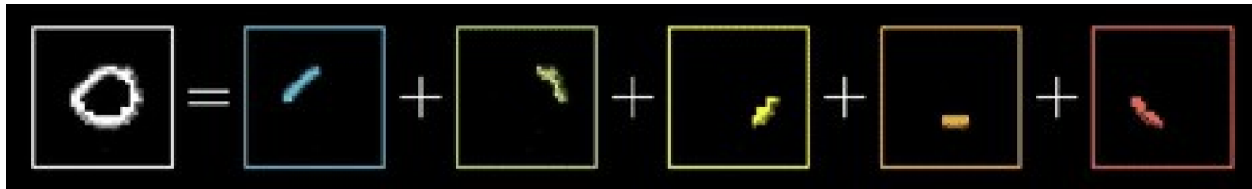
### **Problem of working with images:**

$$\begin{aligned}\text{Given: RGB Image } I & (200, 200, 3) \\ \text{Hidden Layers}_I & \rightarrow 200 * 200 * 3 = 120000 \\ \text{Parameters}_I & \rightarrow (200 * 200 * 3)^2 = 14,400,000,000\end{aligned}$$

These values for Hidden layers and parameters are too large for our Neural Network so we need to find a way to reduce them.

### **Alternative Representation:**

Given this Image  $I$  being a 2D array with dimensions of  $200 * 200 * 3$  pixels. It can be conceptualized as a hierarchical combination of various patterns. This is seen in the figure below:



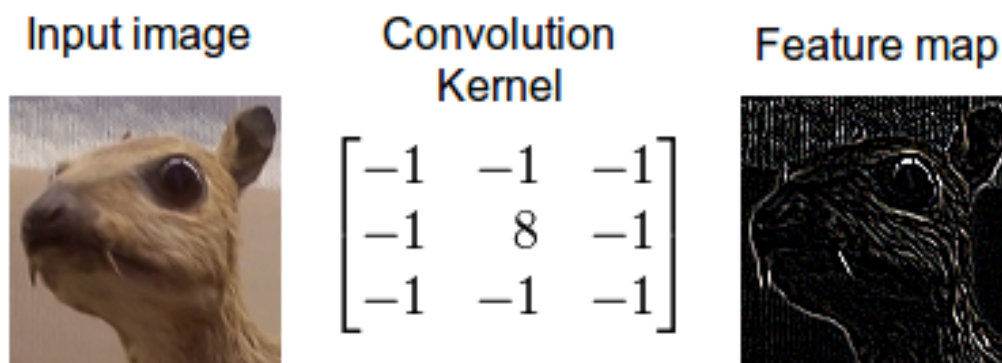
This perspective is advantageous because individual 2D patterns are much smaller compared to the overall image size, allowing us to reduce the size of the problem.

In the pursuit of pattern detection, we can utilize specialized **filters** designed for pattern identification. These filters can exist in either the spatial domain (i.e. Kernel, Spatial Mask, Window, or Template) or the transform domain (utilizing techniques like Fourier Transform). In this context, we will focus on Kernels.

## What is a Kernel?

A Kernel is a small-sized matrix, typically square but not always, which is used to perform operations like blurring, sharpening, edge detection, etc. They are used to scan through an image, identifying patterns and features at different scales. These features can range from simple edges to complex textures or shapes.

Kernels operate on local regions of an image known as receptive fields. This allows the network to focus on small, meaningful parts of the image, capturing local patterns without being influenced by distant information. Each cell in the kernel contains a weight value. These values determine how much influence the corresponding pixel in the original image has on the pixel in the output image.



The above figure shows the input and resulting feature map after a kernel convolution

# Intuitive understanding of matrix convolutions

## Mathematical Representation of Convolutions

Given the Functions:  $f, g$

A convolution of  $f$  with  $g$  can be represented by the formula:

$$\begin{aligned}(f * g)(t) &= \int_{-\infty}^{\infty} f(\tau)g(t - \tau) d\tau \\ &= \int_{-\infty}^{\infty} f(t - \tau)g(\tau) d\tau\end{aligned}$$

As the Convolution operation is commutative

For discrete signals (like images or discrete data), the convolution formula is slightly modified:

$$(f * g)(n) = \sum_{m=-\infty}^{\infty} f(m)g(n - m)$$

## Mathematical Intuition

But what does this formula represent?

At each point  $n$ , the operation multiplies the values of  $f$  and  $g$  at different points after adjusting for their relative positions where it then sums up these products.

The term  $g[n-m]$  shifts the function  $g$  along the  $n$  axis to make it align with the point  $n$  of  $f$ . This alignment is crucial for combining the information appropriately.

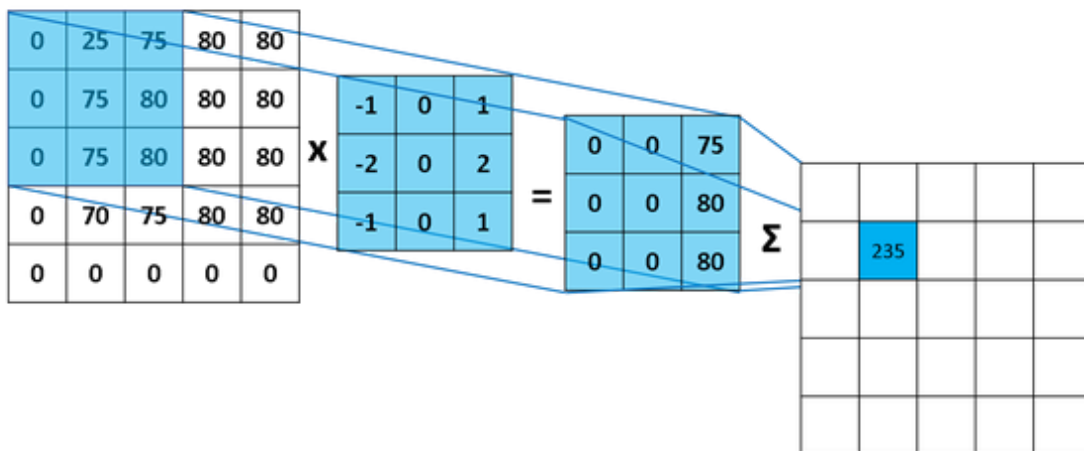
The operation considers all possible shifts of  $g$  across  $n$  and integrates the results. At each  $n$ , the convolution operation calculates a weighted sum of  $f$  and  $g$ , with the weights determined by the shapes of  $f$  and  $g$  and their relative positions.

The result  $(f*g)[n]$  represents the merged information at point  $n$  after considering all possible alignments and weighting them appropriately.

## Image Convolution Process

In Spatial Filtering, we find the patterns of an image by convolving the image with a kernel. We use Convolution to apply a kernel to an image. It involves the following steps:

1. Overlaying the kernel on top of the image, and aligning its center with the pixel we're currently processing. The center of the kernel corresponds to the pixel being processed in the original image.
2. Multiply each element of the kernel with the corresponding element in the image region covered by the kernel. Add up all the products obtained to get a single value which will be the new value of the pixel in the output image.





Refer to the figure above where the 3x3 Kernel  $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$  is multiplied with a pixel and its neighbors in a 3x3 area  $\begin{bmatrix} 0 & 25 & 75 \\ 0 & 75 & 80 \\ 0 & 75 & 80 \end{bmatrix}$  to sum to a single value 235 in the output image. This process is shown by the following equations:

$$\begin{bmatrix} 0 & 25 & 75 \\ 0 & 75 & 80 \\ 0 & 75 & 80 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 75 \\ 0 & 0 & 80 \\ 0 & 0 & 80 \end{bmatrix}$$

$$\sum_i \sum_j \begin{bmatrix} 0 & 0 & 75 \\ 0 & 0 & 80 \\ 0 & 0 & 80 \end{bmatrix} = 235$$

3. Repeat for Every Pixel as if we were to slide the kernel over the entire image, performing the element-wise multiplication and summation for each position. Hence we *Convolve* the kernel across the image.

For pixels near the edges of the image where the kernel may extend beyond the image boundaries, there are different techniques to handle it. One common approach is padding, where extra pixels are added around the image (usually either zeros or constants).

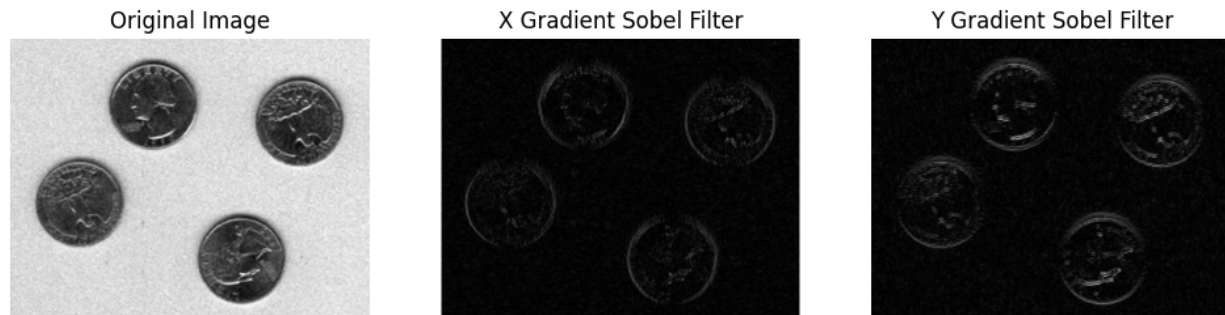
## Examples of Convolution

Kernels are powerful as they can help find the gradients within an image. Take 2 such kernels representing the X and Y sobel filters:

$$X_{\text{sobel}} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

$$Y_{\text{sobel}} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Before and After Convolution:



## Introduction to Convolution Layer

Convolutional layers are responsible for feature extraction. They consist of filters (also known as kernels) that are convolved with the input images to capture relevant patterns and features. These layers learn to detect edges, textures, shapes, and other important visual elements. Convolution preserves the relationship between pixels by learning image features using small blocks of input data.

It is a mathematical operation that takes two inputs such as image matrix and a filter or kernel. The output of a convolution operation is called a feature map. When the input matrix is  $n \times n$  and the filter size is  $f \times f$ , the resulting feature map size is  $(n - f + 1) \times (n - f + 1)$ .

In real-world applications, multiple filters are applied to extract image information, leading to the generation of numerous feature maps. The number of feature maps in a layer corresponds to the number of filters applied at that layer.

Convolution of an image when applying different filters can perform operations such as edge detection, blur, and sharpening. Fig. 1 shows various convolution images after applying different types of filters






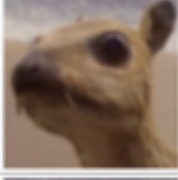
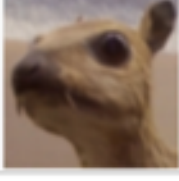
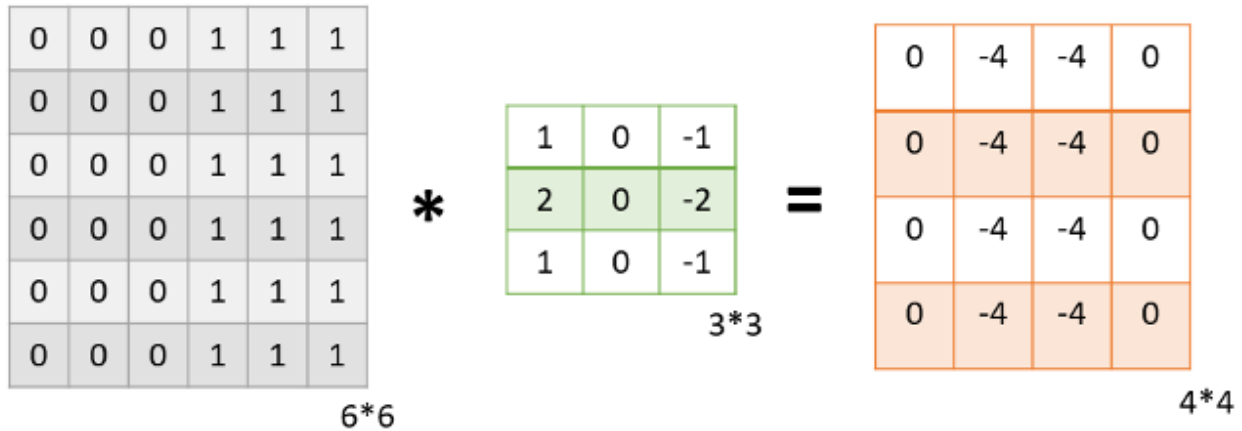
Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Fig. 1

## Understanding the Convolution Process

Let us understand the convolution process with the help of an example. Here we have taken a 2D input image with normalized pixels.

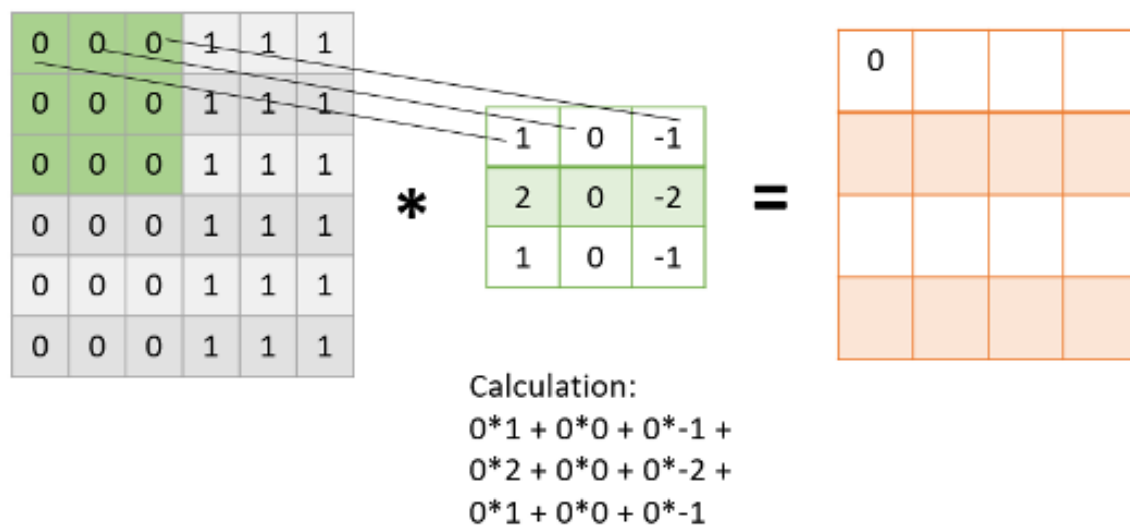


**Fig. 2**

In the above figure, we have an input image of size 66 and apply a filter of 33 on it to detect some features. The result of applying the filter to the image is that we get a feature map of 4\*4 ( $6 - 3 + 1 = 4$ ) which has some information about the input image. (In this example, we have applied only one filter but in practice, many such filters are applied to extract information from the image. All the feature maps generated will be sent to the next layer)

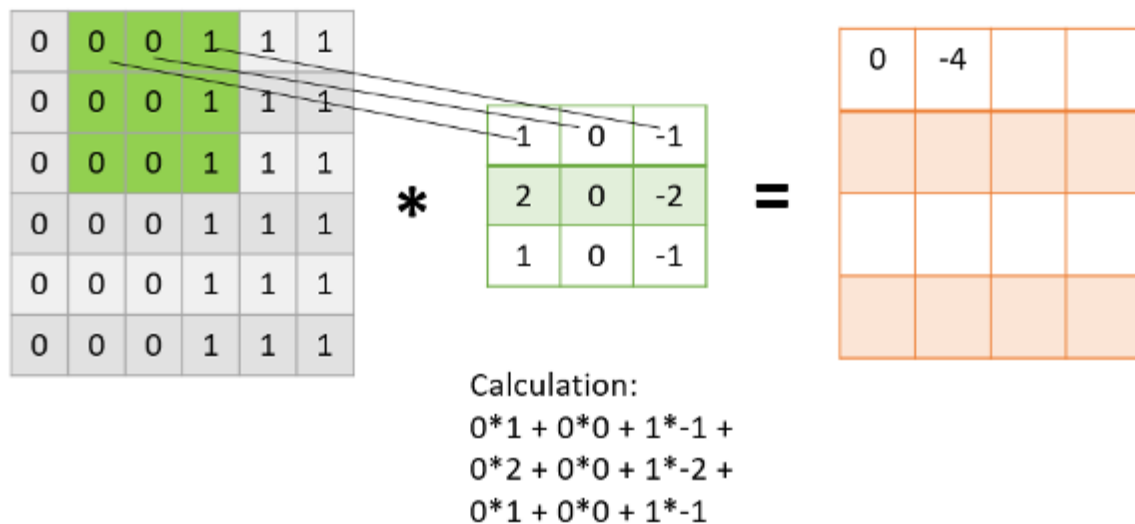
## Math Behind the Feature Map

Let us understand the math behind getting the feature map in Fig. 2.



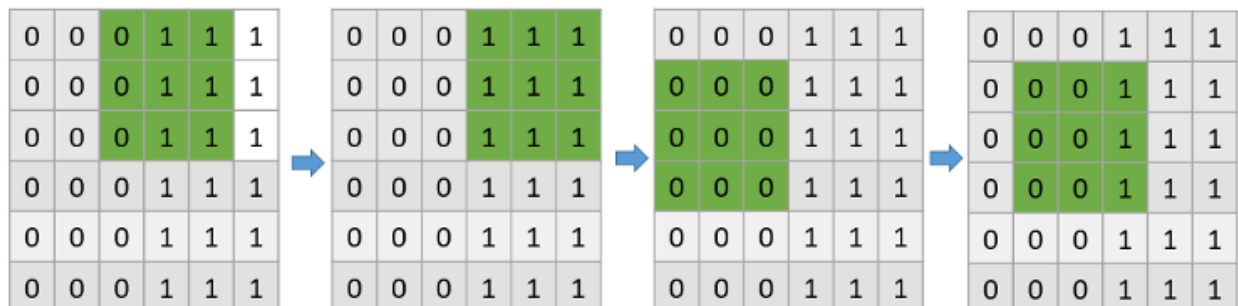
**Fig. 3**

- As presented in the above Fig. 3, in the **first step** the filter is applied to the green highlighted part of the image, and the pixel values of the image are multiplied with the values of the filter (as shown in the figure using lines) and then summed up to get the final value.
- In the next step, the filter is shifted by one column as shown in Fig. 4. This jump to the next column or row is known as **stride** and in this example, we are taking a stride of 1 which means we are shifting by one column.



**Fig. 4**

- Similarly, the filter passes over the entire image (as in Fig. 5), and we get our final **Feature Map**. Once we get the feature map, an activation function is applied to it for introducing non-linearity.



**Fig. 5**

***This is how a filter passes through the entire image with the stride of 1***

A point to note here is that the feature map we get is smaller than the size of our image. As we increase the value of stride the size of the feature map decreases.

## **Design Choices in Convolution Layers**

A smaller filter size captures fine-grained details and features, while a larger filter size captures broader, more global features.

The number of filters in a convolutional layer defines the depth of the layer and determines how many different features the layer can learn.

The low-level features like edges and textures are captured with a small number of filters and the deeper layers gradually increase the number of filters to capture high-level, more abstract features and patterns. This is a form of feature hierarchy.

## **Introduction to Pooling Layer**

Pooling layers reduce the spatial dimensions of the feature maps produced by the convolutional layers. They perform down-sampling operations (e.g., max pooling) to retain the most salient information while discarding unnecessary details. This helps in achieving translation invariance and reducing computational complexity.

The emphasis lies in identifying features' presence, not their exact location.

Impact on the Network:

- Dimensionality Reduction
- Prevents Overfitting: Pooling helps in generalizing and capturing essential features.

The hyperparameters for a pooling layer are:

1. Filter size
2. Stride
3. Pooling type

If the input of the pooling layer is  $H \times W$  then the output will be

$\frac{H}{S} \times \frac{W}{S}$ , where  $S$  = filter-size and  $S$  = stride-length

# Types of Pooling Layers in CNNs

Convolutional neural networks (CNNs) use several pooling layers.

## Max Pooling

It's the most popular pooling layer because it uses the input feature map's pooling regions to get the values that are the highest overall. It discards the noisy activations. With the help of max pooling, we may minimize the amount of input without losing the most crucial details.

## Global Pooling

The maximum or average value over the full spatial dimension of the input feature map is calculated using global pooling. Global pooling is often used to prepare the data from a convolutional layer to be utilized in a fully connected layer.

## Average Pooling

The average value from each pooling area in the input feature map is used for this operation. When input characteristics are noisy, average pooling may assist in smoothing them out.

## Stochastic Pooling

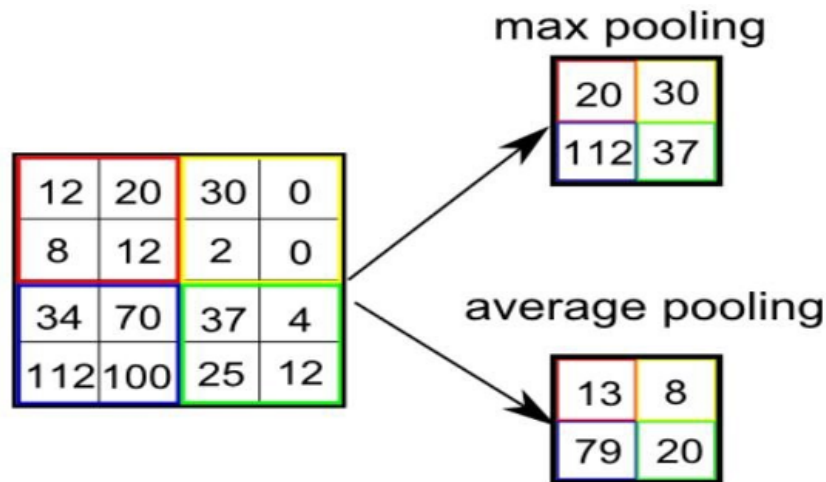
This method selects a single value from the input feature map's pooling regions at random. Little translations in the input may be made more forgiving by stochastic pooling.

## Lp Pooling

The Lp norm of each pooling area in the input feature map is used for Lp pooling. In max pooling, the Lp norm is often employed since it generalizes the Euclidean norm. Lp pooling might provide additional wiggle room when down sampling the input feature map.

The application and network topology must be considered while deciding on a pooling layer. While Max pooling is the most used pooling layer, other pooling layers in CNN may be more suited to certain tasks.

**Example:**



**Fig. 6**

In Fig. 6, we are Pooling with a filter with stride 2

- Max pooling: For every consecutive block, we take the maximum number.
- Average Pooling: For every consecutive block, we take the average of the numbers in the block.

## Fully-connected layers and resulting model capabilities

Fully-connected layers, often referred to as dense layers, are a fundamental component in many artificial neural network architectures, including feedforward neural networks and multi-layer perceptrons (MLPs). These layers play a crucial role in connecting the neurons (nodes) in one layer to the neurons in the next layer.

In an FC layer, each neuron or node from the previous layer is connected to each neuron of the current layer.

FC layers are also known as: Dense layers, Linear layers.

In an FC layer, each neuron applies a linear transformation to the input vector through a weights matrix. The input is multiplied by a weight matrix and then a bias vector is added.

Neural networks are a set of dependent non-linear functions. Each individual function consists of a neuron (or a perceptron). In fully connected layers, the neuron applies a

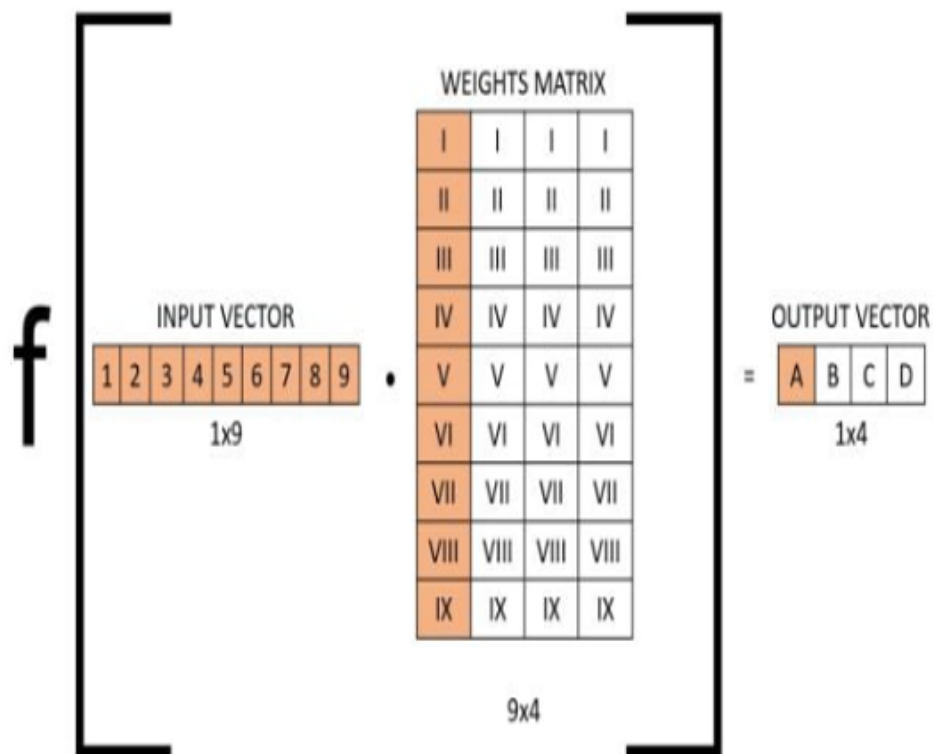


linear transformation to the input vector through a weights matrix. A non-linear transformation is then applied to the product through a non-linear activation function  $f$ .

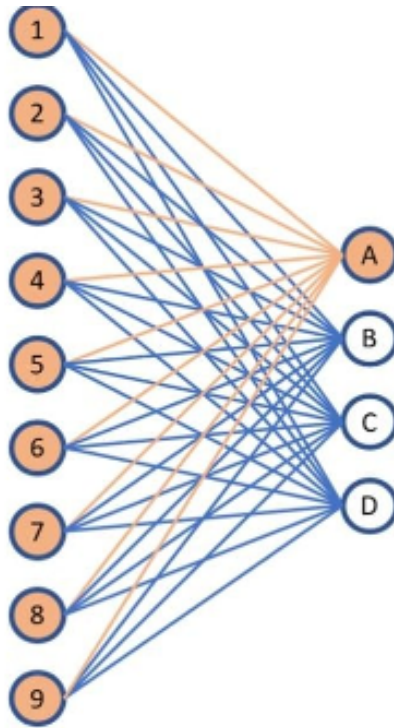
$$y_{jk}(x) = f \left( \sum_{i=1}^{n_H} w_{jk} x_i + w_{j0} \right)$$

Where, weights matrix  $W$  and the input vector  $x$ . The bias term ( $w_0$ ) can be added inside the non-linear function. I will ignore it for the rest of the article as it doesn't affect the output sizes or decision-making and is just another weight.

If we take a layer in a fully connected neural network with an input size of nine and an output size of four, the operation can be visualized as follows:



You can also visualize this layer the following way:



### How to Work With Fully Connected Layers and Convolutional Neural Networks

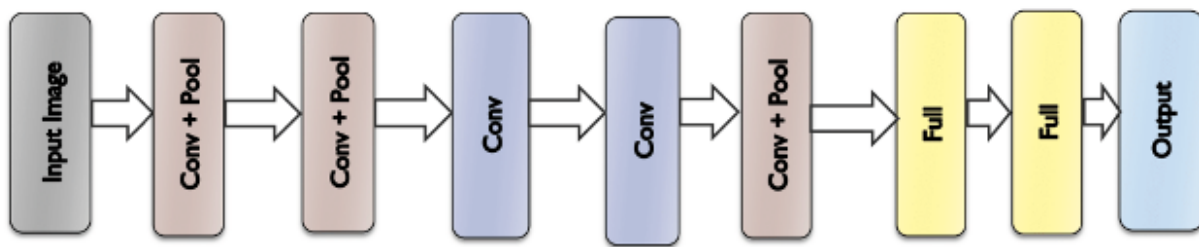
In FC layers, the output size of the layer can be specified very simply by choosing the number of columns in the weights matrix. The same cannot be said for convolutional layers. Convolutions have a lot of parameters that can be changed to adapt the output size of the operation.

Capabilities:

1. **Function Approximation:** Fully-connected layers are capable of approximating a wide range of functions, making them suitable for regression and classification tasks.
2. **Feature Extraction:** They can automatically learn and extract relevant features from raw data, reducing the need for manual feature engineering.
3. **Complex Patterns:** Deep neural networks with multiple fully-connected layers can model intricate and nonlinear patterns in the data.
4. **Hierarchical Abstractions:** The hierarchical structure of deep networks allows them to learn hierarchical abstractions of features in data, which is particularly useful for tasks like image and speech recognition.

5. **Scalability:** The number of neurons and layers in fully-connected networks can be adjusted to match the complexity of the problem, making them suitable for a wide range of tasks.
6. **Feedforward Computation:** During the forward pass of training or inference, the input from the previous layer (or the input data) is multiplied by weights and summed up for each neuron in the current layer. This result is then passed through an activation function to introduce non-linearity into the model.
7. **Activation Function:** Fully-connected layers are typically followed by activation functions like ReLU (Rectified Linear Unit), sigmoid, or hyperbolic tangent (tanh). These activation functions introduce non-linearity, which is essential for modeling complex relationships in the data.
8. **Parameter Learning:** The weights and biases in fully-connected layers are learned during the training process using optimization techniques like gradient descent. This enables the neural network to adapt and fit the training data, making it capable of learning complex functions.

## Alexnet: Case Study:



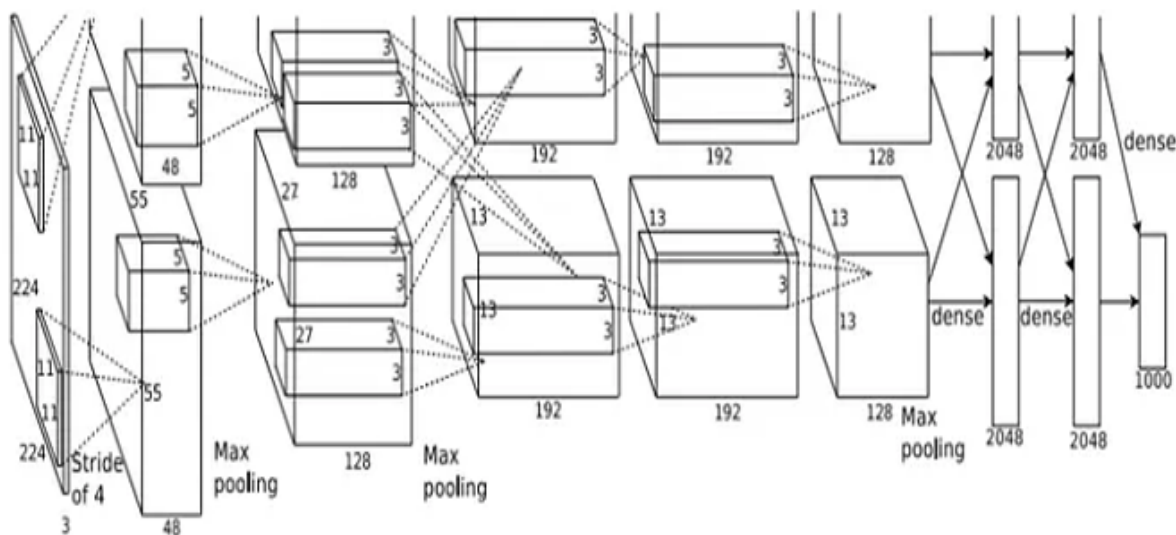
AlexNet is a convolutional neural network (CNN) architecture that won the LSVRC competition in 2012. It was designed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. AlexNet was the first CNN to use a GPU to improve performance.

LSVRC (Large Scale Visual Recognition Challenge) is a competition where research teams evaluate their algorithms on a huge dataset of labelled images (ImageNet) and compete to achieve higher accuracy on several visual recognition tasks.

AlexNet was trained on over a million (1.2M) images from the ImageNet database using Stochastic Gradient Descent (SGD) with regularization. The pretrained network can classify images into 1,000 object categories, such as: Keyboards, Mice, Pencils, Many animals. AlexNet revolutionized the field of machine learning. It was better than Lenet because it had more filters per layer and stacked convolutional layers.

AlexNet is made up of eight layers:

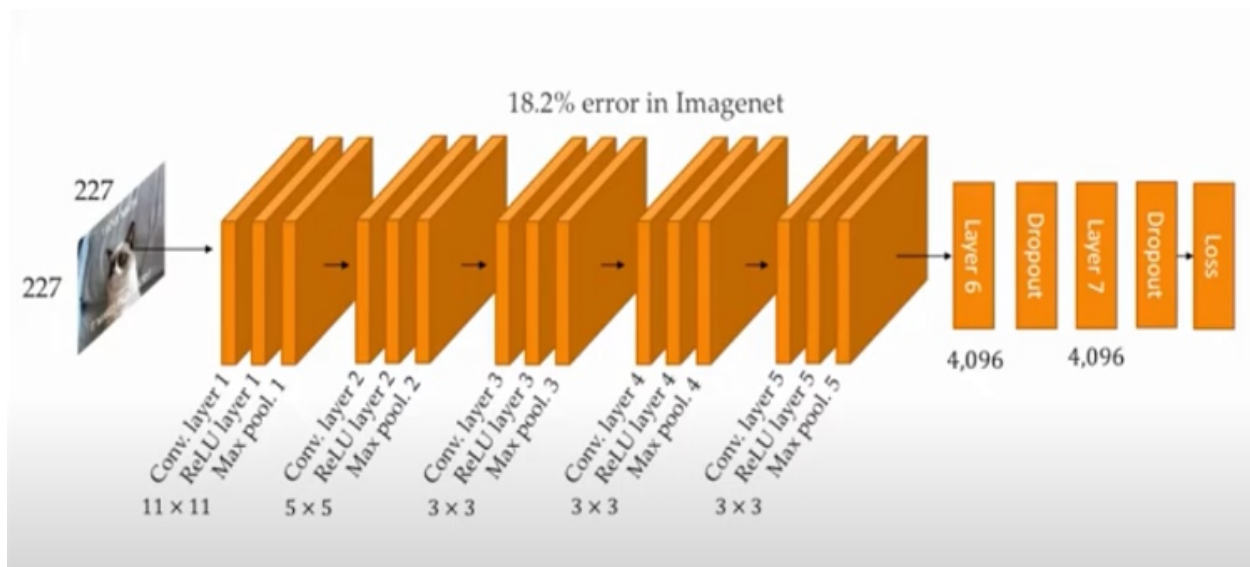
1. 5 convolutional layers
2. 3 max-pooling layers
3. 2 normalization layers
4. 2 fully connected layers
5. 1 softmax layer



AlexNet uses grouped convolutions to fit the model across two GPUs. It also uses pooling windows that are 3×3 in size with a stride of 2 between adjacent windows.

At the end of each layer, ReLu activation is performed except for the last one, which outputs with a softmax with a distribution over the 1000 class labels. Dropout is applied

in the first two fully connected layers. As the figure above shows also applies Max-pooling after the first, second, and fifth convolutional layers. The kernels of the second, fourth, and fifth convolutional layers are connected only to those kernel maps in the previous layer, which reside on the same GPU. The kernels of the third convolutional layer are connected to all kernel maps in the second layer. The neurons in the fully connected layers are connected to all neurons in the previous layer.



### Key Features:

1. **Deep Architecture:** AlexNet is a deep neural network consisting of eight layers, five of which are convolutional layers, and three are fully connected layers. It was one of the first deep CNNs to gain widespread attention. It uses around 60 million parameters.
2. **Rectified Linear Unit (ReLU) Activation:** AlexNet used the rectified linear unit as the activation function in its hidden layers. ReLU is computationally efficient and helps mitigate the vanishing gradient problem, allowing for faster training.
3. **Local Response Normalization (LRN):** AlexNet employed local response normalization as a form of regularization in the first two convolutional layers. This technique helps neurons to respond to a wider range of inputs and improves generalization.

4. Max-Pooling: Max-pooling layers are used to downsample the feature maps obtained from the convolutional layers, reducing the dimensionality of the data.
5. Overlapping Pooling: In contrast to traditional pooling, which is non-overlapping, AlexNet used overlapping pooling, which means the pooling regions overlap. This further contributed to the network's discriminative power.
6. Dropout: AlexNet used dropout in the fully connected layers to prevent overfitting. Dropout randomly deactivates a fraction of neurons during training, forcing the network to learn more robust features.
7. Large Dataset: To train AlexNet, the researchers used the ImageNet Large Scale Visual Recognition Challenge dataset, which contains millions of labeled images in thousands of categories.

#### SGD in AlexNet:

Stochastic Gradient Descent (SGD) is a widely used optimization algorithm in deep learning, including in the training of the AlexNet model. SGD is used to update the weights of the neural network during the training process in order to minimize the loss function and improve the model's performance. Here's how SGD is used in AlexNet training:

1. Initialization: Initially, the model's weights are randomly initialized. The goal is to learn the optimal weights through training.
2. Mini-Batch Training: Instead of computing gradients and updating weights for the entire dataset in each training step, AlexNet, like many deep learning models, uses mini-batches. In each training iteration, a small random subset of the dataset (a mini-batch) is used to compute gradients and update the weights. This introduces stochasticity into the optimization process, which can help the model escape local minima and converge faster.
3. Compute Gradients: For each mini-batch, the model computes the gradients of the loss function with respect to the model's parameters (weights and biases) using backpropagation. This involves propagating the error gradients from the output layer to the input layer.
4. Update Weights: The SGD algorithm then updates the model's weights based on the computed gradients. The update rule typically follows this formula for each

weight parameter  $\theta$ :

$$\theta_{\text{new}} = \theta - \text{learning\_rate} * \text{gradient}$$

1. Learning Rate: This is a hyperparameter that determines the step size for weight updates. It controls how quickly or slowly the model learns.
2. Regularization: In the case of AlexNet, additional techniques like dropout and local response normalization (LRN) may be used to regularize the model during training to prevent overfitting.
3. Repeat: Steps 3 to 5 are repeated for a fixed number of iterations (epochs) or until a convergence criterion is met. The process continues until the model's performance on the validation set stops improving or plateaus.

ReLU Feature:

An important feature of the AlexNet is the use of ReLU(Rectified Linear Unit) Nonlinearity.

Tanh or sigmoid activation functions used to be the usual way to train a neural network model.

AlexNet showed that using ReLU nonlinearity, deep CNNs could be trained much faster than using the saturating activation functions like tanh or sigmoid.

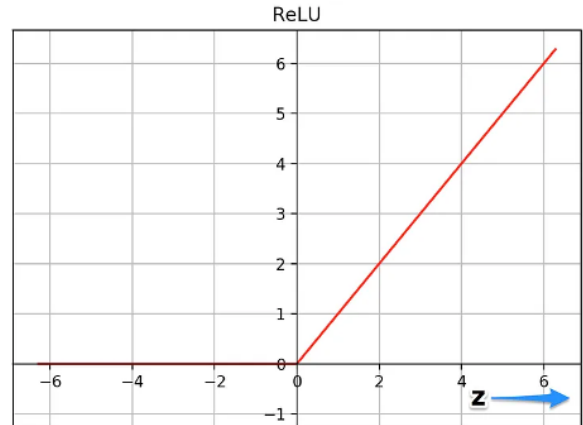
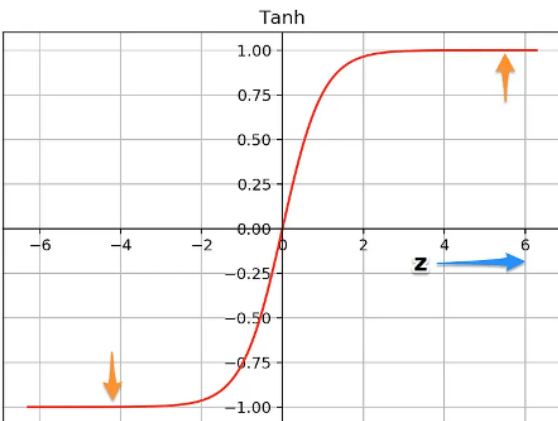
Tested on the CIFAR-10 dataset.

Let's see why it trains faster with the ReLUs. The ReLU function is given by

$$f(x) = \max(0, x)$$

plots of the two functions —

1. tanh
2. ReLU.



The tanh function saturates at very high or very low values of  $z$ . In these regions, the slope of the function goes very close to zero. This can slow down gradient descent. The ReLU function's slope is not close to zero for higher positive values of  $z$ . This helps the optimization to converge faster.

### The Overfitting Problem:

AlexNet had 60 million parameters, a major issue in terms of overfitting.

Two methods to reduce overfitting:

- Data Augmentation:

The authors generated image translations and horizontal reflections, which increased the training set by 2048. They also performed Principle Component Analysis (PCA) on the RGB pixel values to change RGB channels' intensities, which reduced the top-1 error rate by more than 1%.

- Dropout:

It consists of setting to zero the output of each hidden neuron with a probability of 0.5. The neurons which are “dropped out” in this way do not contribute to the forward pass and do not participate in backpropagation. So every time an input is presented, the neural network samples a different architecture. This technique consists of turning off neurons with a predetermined probability. This means that every iteration, the neurons “turned off” do not contribute to the forward pass and do not participate in backpropagation

Advantages:



1. AlexNet is considered as the milestone CNN for image classification. Many methods, such as the conv+pooling design, dropout, GPU, parallel computing, ReLU, are still the industrial standard for computer vision. The unique advantage of AlexNet is the direct image input to the classification model.
2. The convolution layers can automatically extract the edges of the images and fully connected layers learning these features
3. Theoretically the complexity of visual patterns can be effectively extracted by adding more conv layer

#### Disadvantages:

1. AlexNet is NOT deep enough compared to the later model, such as VGGNet, GoogLeNet, and ResNet.
2. The use of large convolution filters (5\*5) is not encouraged shortly after that.
3. Use normal distribution to initiate the weights in the neural networks, can not effectively solve the problem of gradient vanishing, replaced by the Xavier method later. The performance is surpassed by more complex models such as GoogLeNet (6.7%), and ResNet (3.6%)

#### **cuda-convnet**

"cuda-convnet" refers to an optimized GPU implementation of convolutional neural networks (CNNs) for deep learning. It was developed by Alex Krizhevsky, the same researcher who was one of the authors of the AlexNet architecture. The "cuda-convnet" library was created to take advantage of the computational power of GPUs for training and running deep neural networks efficiently.

#### Key Points:

1. GPU Acceleration: The primary purpose of "cuda-convnet" was to leverage the parallel processing capabilities of graphics processing units (GPUs) to significantly accelerate the training and inference of convolutional neural networks. Training deep networks like AlexNet can be computationally intensive, and GPUs excel at handling the large-scale matrix multiplications and convolutions involved.
2. Optimized CUDA Code: "cuda-convnet" is implemented using CUDA, a parallel computing platform and application programming interface created by NVIDIA. It

provides low-level access to the GPU, allowing developers to write highly optimized code for deep learning operations.

3. **Efficient Convolution Operations:** The library is specifically tailored for convolutional neural networks, making it highly efficient for convolution operations, which are a fundamental part of CNNs. Convolution operations involve sliding a filter or kernel over an input feature map and performing element-wise multiplications and summations. These operations can be massively parallelized on GPUs.
4. **Flexible and Customizable:** While "cuda-convnet" was originally developed to support the training of specific neural network architectures, it is highly customizable. Researchers and developers can adapt it to their specific network architectures and tasks.

## Autoencoders

Autoencoders are very useful in the field of unsupervised machine learning. One can use them to compress the data and reduce its dimensionality, making it easier to work with. Autoencoders are a specific type of feedforward neural networks where the input is the same as the output. They compress the input into a lower-dimensional code and then reconstruct the output from this representation. The code is a compact "summary" or "compression" of the input, also called the latent-space representation.

### Key Characteristics of Autoencoders

The main difference between Autoencoders and Principle Component Analysis (PCA) is that while PCA finds the directions along which you can project the data with maximum variance, Autoencoders reconstruct our original input given just a compressed version of it. That is, PCA finds the best angles to show off the important parts of your data, like a photographer trying to capture the most interesting angles. Autoencoders, on the other hand, are like master puzzle solvers. They take a small, mysterious piece of your data and cleverly put together the whole picture.

Autoencoders are mainly a dimensionality reduction (or compression) algorithm with a couple of important properties:

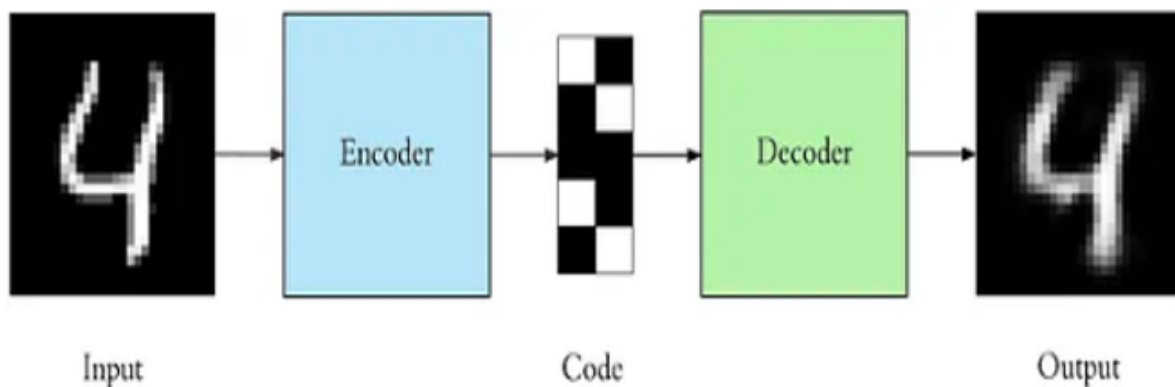
- **Tailored for Specific Data:** Autoencoders work best when they've seen similar data before. They learn to crunch the numbers in a way that's unique to their training

data. So, an autoencoder trained on numbers won't be much help with pictures of landscapes.

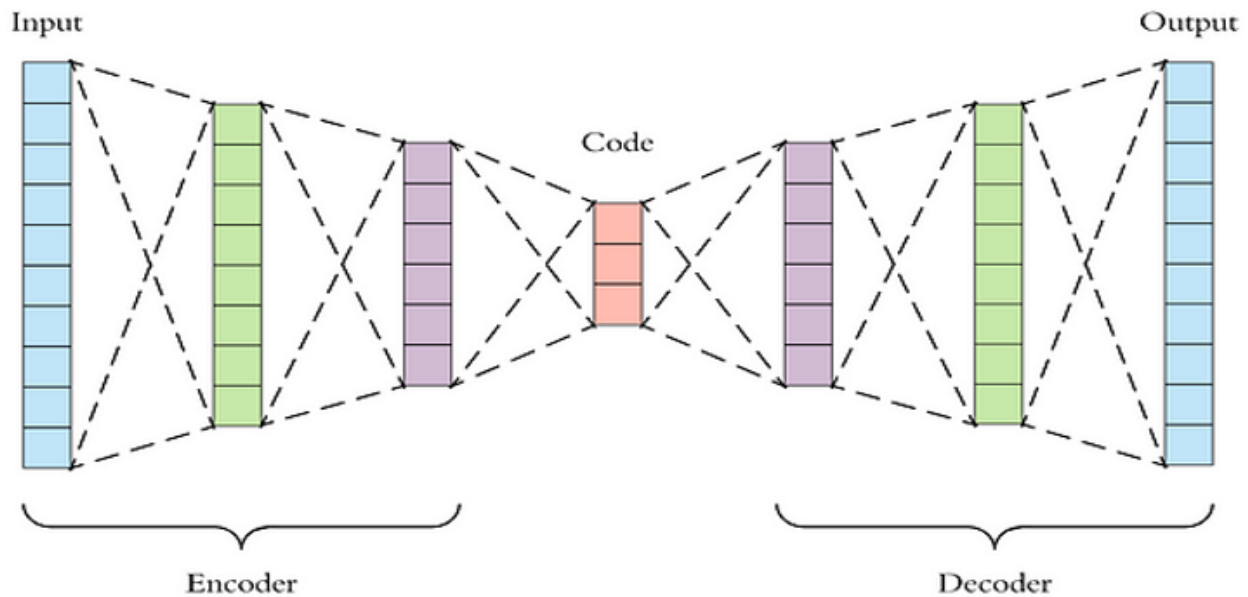
- **Lossy:** The output of the autoencoder will not be exactly the same as the input, it will be a close but degraded representation. If you want lossless compression they are not the way to go.
- **Unsupervised:** Autoencoders are easy to train. You just feed them raw data. They're called unsupervised learners because they don't rely on fancy labels. They're more like self-learners, creating their own labels from the data they get."

## Architecture

An autoencoder consists of 3 components: encoder, code and decoder. The encoder compresses the input and produces the code, the decoder then reconstructs the input only using this code.



To build an autoencoder we need 3 things: an encoding method, decoding method, and a loss function to compare the output with the target. Both the encoder and decoder are fully-connected feedforward neural networks. Code is a single layer of an ANN with the dimensionality of our choice. The number of nodes in the code layer (code size) is a hyperparameter that we set before training the autoencoder.



Let's take a closer look at autoencoders. They have two main parts: an encoder and a decoder. The encoder is like a neural network that takes your input and transforms it into a secret code. The decoder, with a similar structure, then works its magic to recreate the original input. The ultimate goal is to make the output as identical as possible to the input.

You might notice that the decoder looks like a mirror image of the encoder, although this isn't a strict rule – it's just a common setup. The critical thing is that the input and output dimensions need to match, but you have plenty of flexibility in between to experiment and customize.

There are 4 hyperparameters that we need to set before training an autoencoder:

- Code size: number of nodes in the middle layer. Smaller size results in more compression.
- Number of layers: the autoencoder can be as deep as we like. In the figure above we have 2 layers in both the encoder and decoder, without considering the input and output.
- Number of nodes per layer: the autoencoder architecture we're working on is called a stacked autoencoder since the layers are stacked one after another. Usually stacked autoencoders look like a "sandwich". The number of nodes per layer decreases with each subsequent layer of the encoder, and increases back in the

decoder. Also the decoder is symmetric to the encoder in terms of layer structure. As noted above this is not necessary and we have total control over these parameters.

- **Loss function:** we either use mean squared error (mse) or binary crossentropy. If the input values are in the range  $[0, 1]$  then we typically use crossentropy, otherwise we use the mean squared error.

Autoencoders are trained the same way as ANNs via backpropagation.

## Use Cases

Autoencoders have various use-cases like:

- **Anomaly detection:** autoencoders can identify data anomalies using a loss function that penalizes model complexity. It can be helpful for anomaly detection in financial markets, where you can use it to identify unusual activity and predict market trends.
- **Data denoising image and audio:** autoencoders can help clean up noisy pictures or audio files. You can also use them to remove noise from images or audio recordings.
- **Image inpainting:** autoencoders have been used to fill in gaps in images by learning how to reconstruct missing pixels based on surrounding pixels. For example, if you're trying to restore an old photograph that's missing part of its right side, the autoencoder could learn how to fill in the missing details based on what it knows about the rest of the photo.
- **Information retrieval:** autoencoders can be used as content-based image retrieval systems that allow users to search for images based on their content.

## Disadvantages

- **Image Quality:** As the complexity of images increases, autoencoders can struggle to maintain image quality. This can result in blurry or distorted reconstructions, especially with highly detailed or intricate images.
- **Limited Generalization:** Autoencoders may not generalize well beyond the training data. They can misclassify input errors or fail to adapt to changes in underlying relationships that may be noticeable to a human observer.

- **Hyperparameter Sensitivity:** Autoencoders often require careful tuning of hyperparameters to achieve optimal results, making them less user-friendly and more time-consuming to work with.
- **Training Complexity:** Training autoencoders can be computationally intensive and time-consuming, particularly with deep and complex architectures. This can be a barrier for practical applications that require real-time processing.
- **Limited Applicability:** Autoencoders are not suited for all types of data. Their effectiveness varies depending on the specific problem and the characteristics of the data. In some cases, alternative approaches may be more suitable.

#### References:

<https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/>

<https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148>