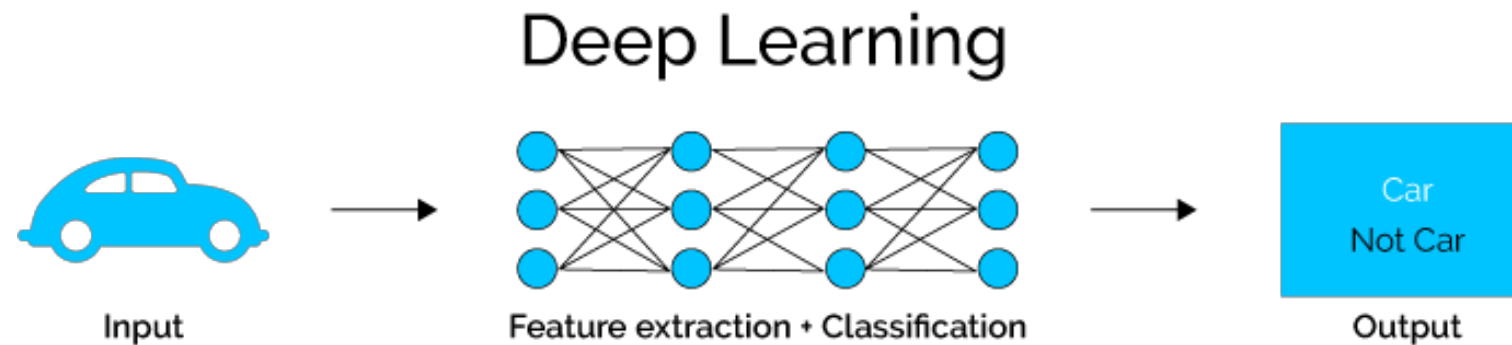
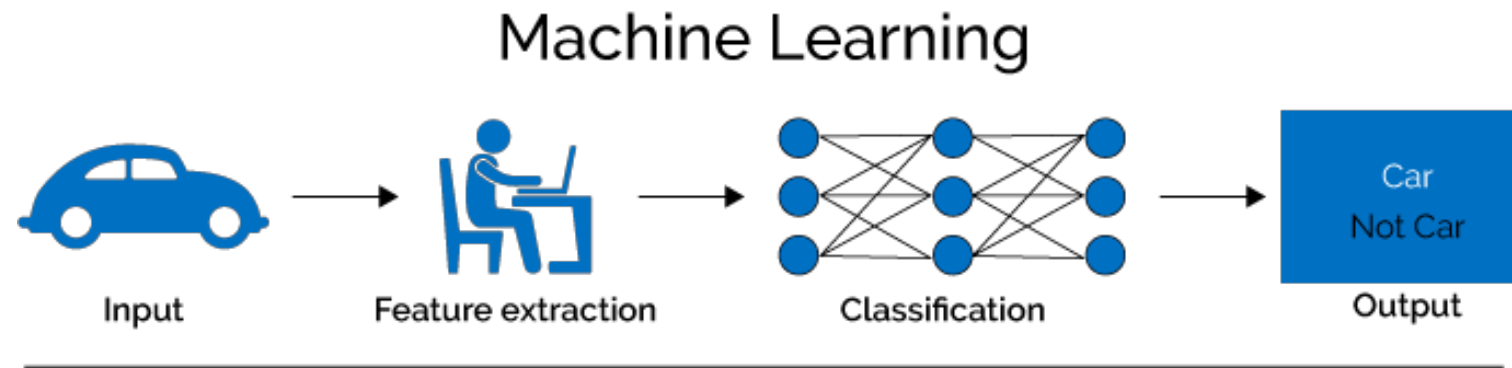


COMP30027 MACHINE LEARNING TUTORIAL

Workshop - 10

Neural Networks, Autoencoders, and GAN Overview



Machine learning

- A **broad field** of algorithms that learn patterns from data to make predictions or decisions.
- Requires **manual feature engineering**
- Easier to **interpret and debug**
- Can work well with **small to medium datasets**

Examples: Linear Regression, Decision Trees, Support Vector Machines, and K-Nearest Neighbours

Deep learning

- A **subset of ML** that uses **neural networks with many layers** (deep neural networks).
- **Automatic feature extraction**
- Harder to interpret
- Requires **large datasets** to work well

Examples: CNNs (for images), RNNs / LSTMs (for sequences) and Transformers (for text)

Neural Network

Neural networks draw inspiration from the human brain's structure and functionality. They excel in tasks such as classification, regression, and pattern recognition.

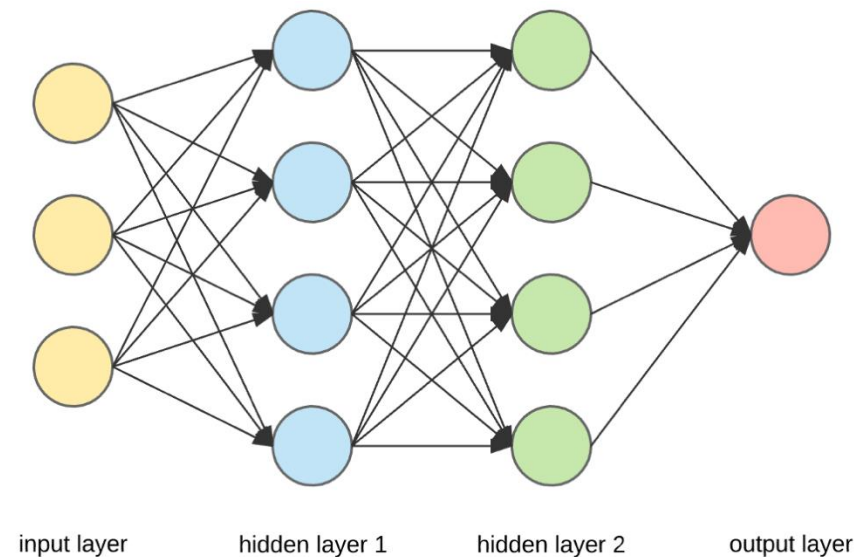
The architecture typically consists of interconnected layers: input, hidden, and output. Each layer transforms data in meaningful ways.

Input layer

The input layer of a neural network receives data. This data will have been processed from sources like images or tabular information and reduced into a structure that the network understands.

Hidden layer

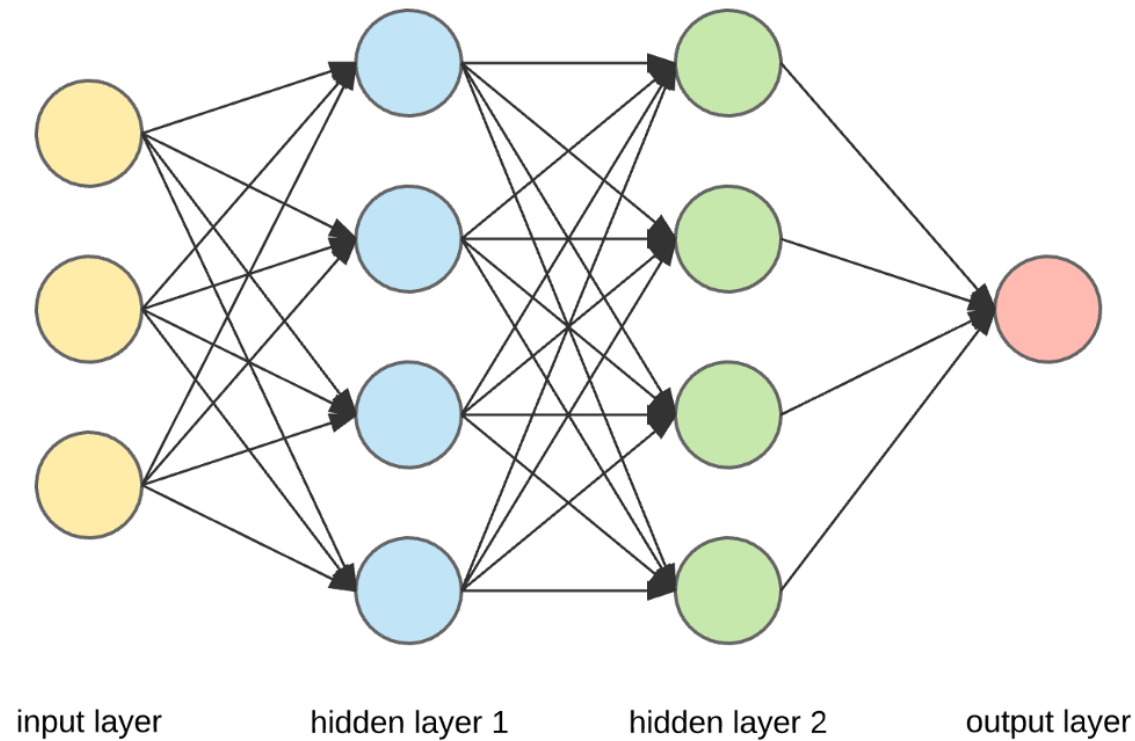
They are the intermediate layers that perform computations and extract features from data. There may be multiple interconnected hidden layers, each responsible for automatically identifying different features in the data. For instance, in image classification, early hidden layers detect high-level features such as edges or shapes, while later layers recognise complete objects like cars, buildings, or people.



Output layer

The output layer receives input from the preceding hidden layers and generates a final prediction based on the model's learned information.

Neural Network Workflow: Training, Testing, Evaluation



Training

Adjust weights via backpropagation and
and gradient descent

Testing

Evaluate on unseen data for generalization
generalization

Evaluation

Use metrics like accuracy, loss, F1 score, and
MSE

Trainable Parameters in Neural Networks

Weights and Biases

These values are modified during training to optimize performance.

They determine how input signals transform across the network.

Parameter Count

Equals the total connections between neurons plus bias terms.

A single dense layer with 100 inputs and 50 outputs has 5,050 parameters.

$$(100 + 1 \text{ (bias)}) \times 50 = 5050$$

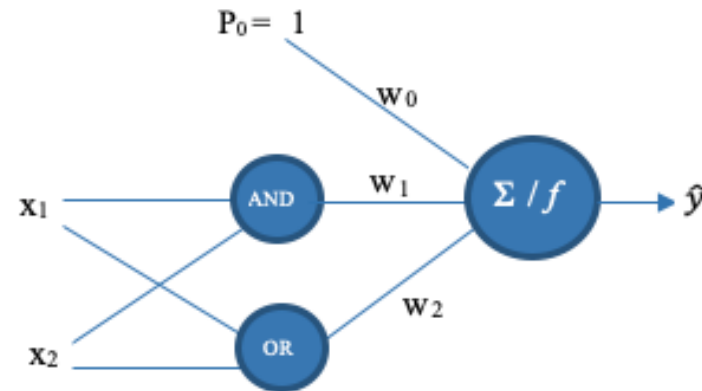
Complexity Scaling

Parameter count grows dramatically with network depth and width.

Modern networks can have millions or billions of parameters.

Q1

Consider the two layers deep network illustrated below. It is composed of three perceptrons. The two perceptrons of the first layer implement the AND OR functions, respectively.



Determine the weights w_1 , w_2 and bias w_0 for the layer 2 perceptron such that the network implements the XOR function.

Notes:

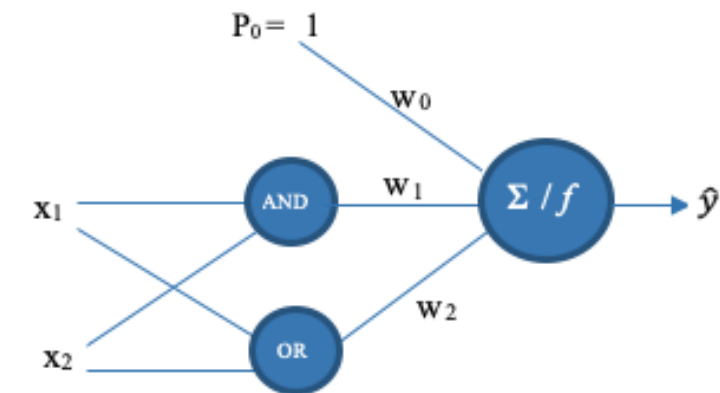
- The input function for the perceptron on layer 2 is the weighted sum (Σ) of its input.
- The activation function f for the perceptron on layer 2 is a step function which outputs 1 if the weighted sum is >0 and 0 otherwise.
- Assume the weights for the layer 1 perceptrons are given.

Step 1: XOR Truth Table and Layer 1 Outputs

x_1	x_2	$\text{AND}(x_1, x_2)$	$\text{OR}(x_1, x_2)$	$\text{XOR}(x_1, x_2)$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

[AND(x1, x2), OR(x1, x2)] → want to learn y

AND(x1, x2)	OR(x1, x2)	Target Output (XOR)
0	0	0
0	1	1
1	1	0
1	0	1



Step 2: Learn Weights for Layer 2 Perceptron

We want a perceptron with inputs AND(x1, x2), OR(x1, x2), and weights w1, w2, and bias w0 such that:

Let AND(x1, x2) and OR(x1, x2) be denoted by h1 and h2, respectively.

$y = \text{step}(w_1 * h_1 + w_2 * h_2 + w_0)$ — given activation function is step function

$\text{step}(z) = 1$ if $z > 0$, else 0

Let's plug in the values and build a system of inequalities based on the truth table:

From the inputs:

1. $h1 = 0, h2 = 0 \rightarrow$ output should be 0

$\rightarrow w1*0 + w2*0 + w0 \leq 0$

$\rightarrow w0 \leq 0$

2. $h1 = 0, h2 = 1 \rightarrow$ output should be 1

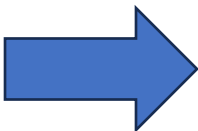
$\rightarrow w1*0 + w2*1 + w0 > 0$

$\rightarrow w2 + w0 > 0$

3. $h1 = 1, h2 = 1 \rightarrow$ output should be 0

$\rightarrow w1*1 + w2*1 + w0 \leq 0$

$\rightarrow w1 + w2 + w0 \leq 0$



Let's pick some values that satisfy all these:

Try:

• $w1 = -1$

• $w2 = 1$

• $w0 = 0$

Now check:

1. $0 + 0 + 0 = 0 \rightarrow \text{step}(0) = 0$

2. $0 + 1 + 0 = 1 \rightarrow \text{step}(1) = 1$

3. $-1 + 1 + 0 = 0 \rightarrow \text{step}(0) = 0$

You'll get different solutions depending on the initial values you choose.

h1	h2	Target Output (XOR)
0	0	0
0	1	1
0	1	1
1	1	0

Q3

Calculate the number of trainable parameters in a multilayer perceptron (neural network) with one hidden layer. Assume:

- the input size is 1000
- the hidden layer size is 100
- the output layer size is 20

Consider the number of trainable parameters on each layer:

• *The input to this model is 1000 inputs.*

• *The **hidden layer** contains 100 neurons. Each one will compute a weighted sum of the 1000 inputs, so each neuron will learn 1000 weights and 1 bias. The number of trainable parameters in the hidden layer is*

$$100 \times (1000 + 1) = 100100.$$

• *The **output layer** contains 20 neurons. Each one will compute a weighted sum of the values from the previous layer, which has 100 neurons. So each of the output layer neurons will learn 100 weights and 1 bias. The number of trainable parameters in the output layer is $20 \times (100 + 1) = 2020$.*

Sum up the layers to get the total number of trainable parameters in this network:

$$100100 + 2020 = 102120.$$

Neural Network Embeddings

What They Are

An embedding is a numerical representation of discrete input data (such as words, items, or categories) in a continuous vector space, learned automatically by the network during training. For example, it maps symbolic entities—like the word "apple"—to dense vectors (e.g., [0.21, -0.58, 1.13, ...]) that can be understood and processed by neural networks.

Benefits

- Capture **semantic relationships** between entities. Enable **similarity comparisons and relationship** discovery between words or images.

Let the learned embedding for the words “great”, “awesome”, and “terrible” be:

Embedding("great") \approx [0.12, -0.43, 0.88, ...], Embedding("awesome") \approx [0.14, -0.40, 0.91, ...], and Embedding("terrible") \approx [-0.77, 0.65, -0.34, ...]

"great" and "awesome" will be close in the embedding space. Meanwhile, "terrible" will be far from both. You can measure similarity with cosine similarity:

$\text{cos_sim}(\text{"great"}, \text{"awesome"}) \approx 0.98$

$\text{cos_sim}(\text{"great"}, \text{"terrible"}) \approx -0.80$

- Reduce dimensionality while preserving important information.

Q2

Neural networks are used for **representation learning**; the representations (or "embeddings") learned by neural networks trained on one task are often useful for a variety of other tasks.

- What are the features of neural networks that make them particularly suitable for representation learning?
- What is a "neural network embedding" and how is it useful in machine learning?

What Neural Networks Do

- Neural networks **automatically learn features** during training — this is called **automatic feature engineering**.
- **Each layer** in the network transforms its input into more useful representations.
- Layers form a **hierarchy**:
 - Early layers learn **simple patterns** (e.g., color, edges).
 - Deeper layers learn **complex patterns** (e.g., object parts like eyes, faces).

What Are Embeddings?

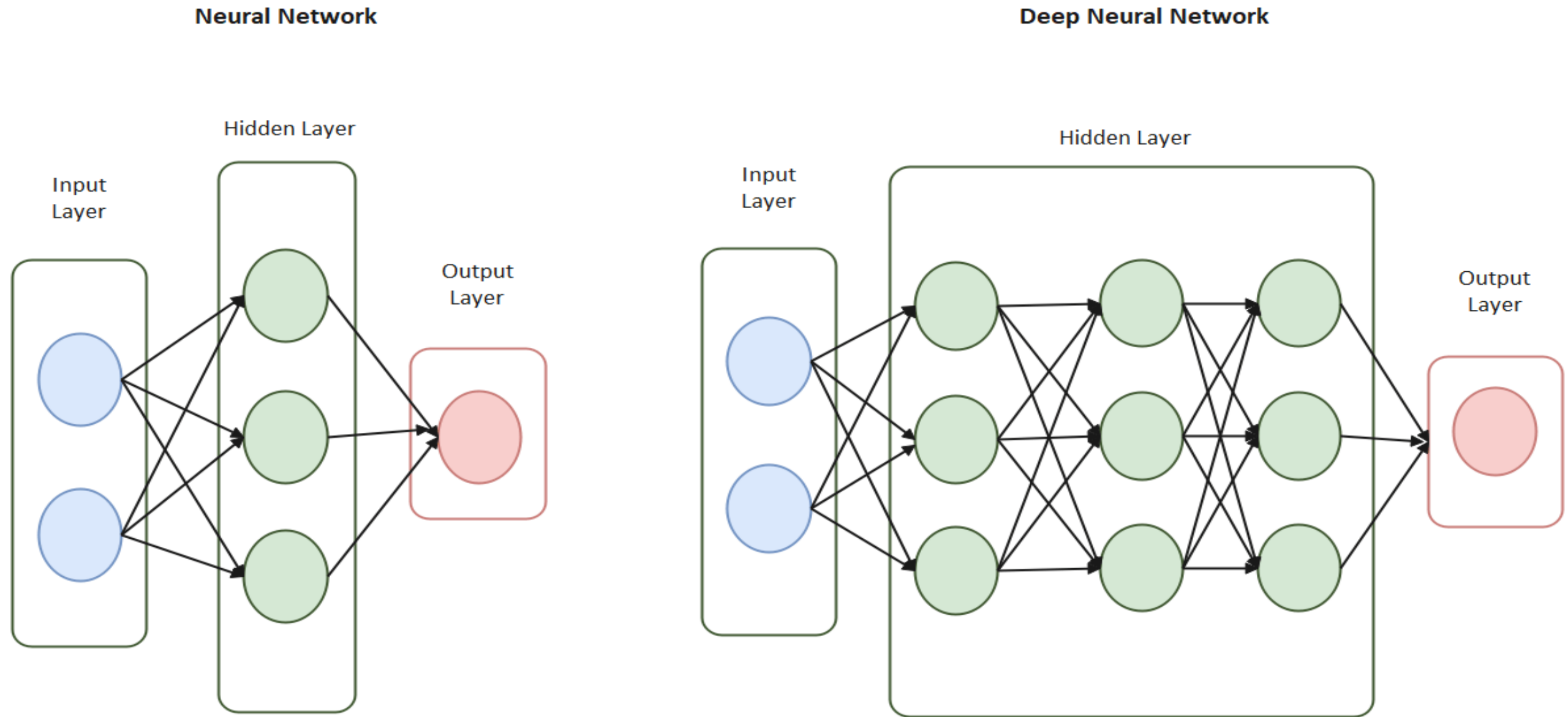
Embeddings are compact, dense vector representations of input data that captures the **important features** of the input data in a **compressed, meaningful way**.

- Usually, embeddings are taken from **late layers**, because:
- They capture **abstract, high-level features**.
- They are **lower-dimensional** (more compact).

Why Use Embeddings?

- Embeddings are **low-dimensional**, helping avoid the **curse of dimensionality**.
- They summarize useful information in a compact way.
- You can **re-use embeddings** from a trained network for new tasks (this is called **transfer learning**).
- Even **simple models** (like logistic regression or K-NN) can perform well using embeddings.

Deep Neural Networks (DNN): Architecture and Depth



Deep networks learn hierarchical representations through multiple layers. Each layer transforms data into increasingly abstract features.

Convolutional Neural Networks (CNNs): Introduction

Input

Raw image data enters the network

Convolution

Filters detect patterns across the image

Pooling

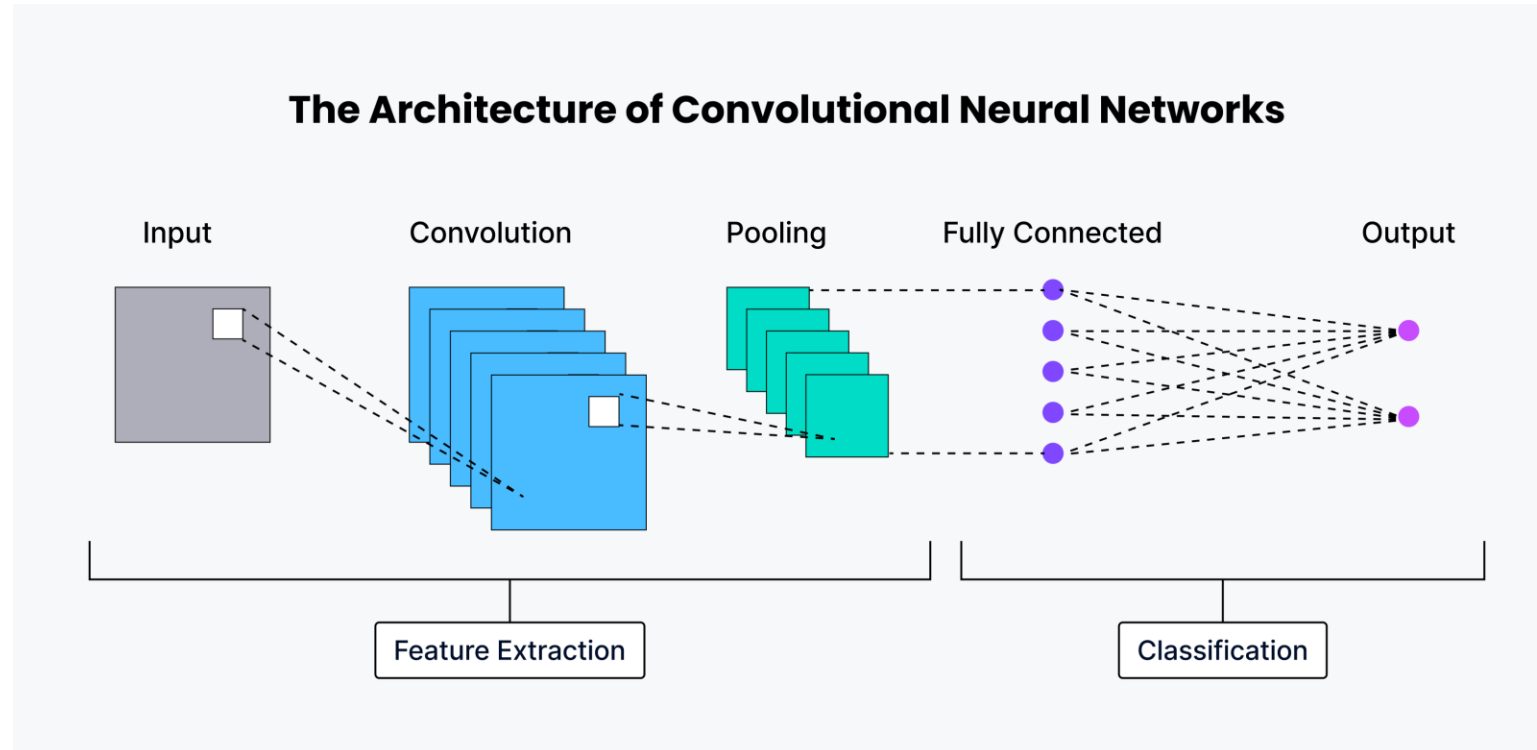
Downsampling reduces dimensions while preserving features

Fully Connected

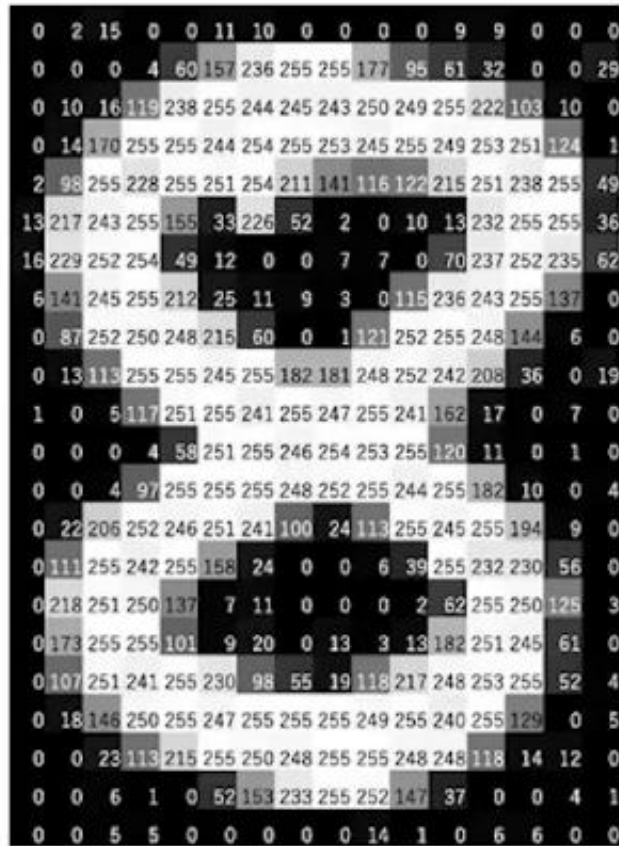
Final layers classify based on extracted features

Output

The final predicted class for the image



Input Image



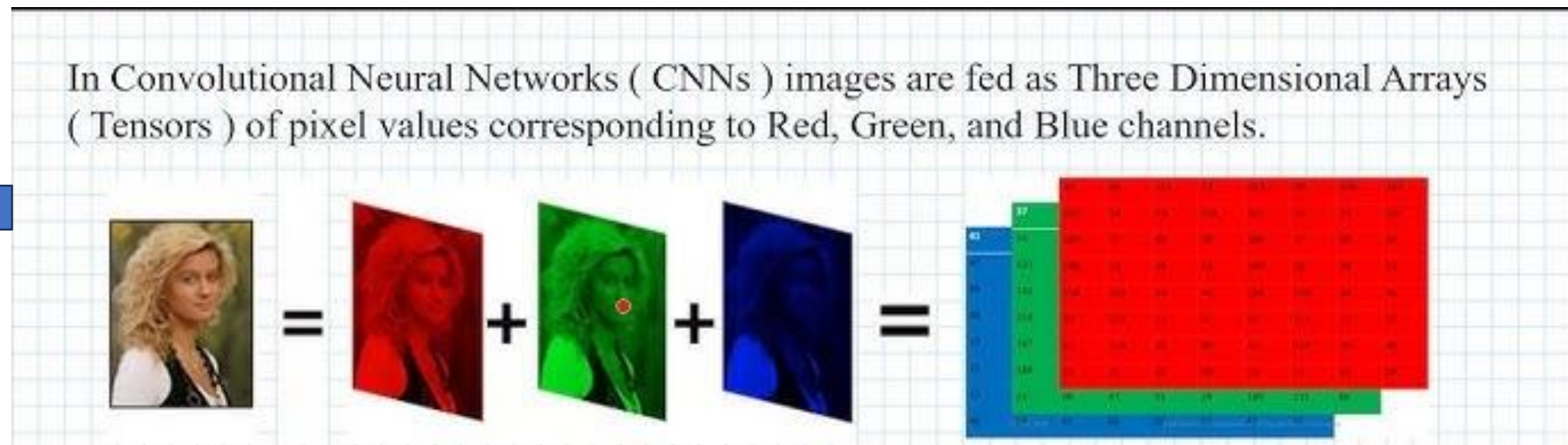
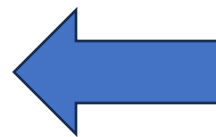
0	2	15	0	0	11	10	0	0	0	9	9	0	0	0
0	0	0	4	60	157	236	255	255	177	95	61	32	0	29
0	10	16	119	238	255	244	245	243	250	249	255	222	103	10
0	14	170	255	255	244	254	255	253	245	255	249	253	251	124
2	98	255	228	255	251	254	211	141	116	122	215	251	238	255
13	217	243	255	155	33	226	52	2	0	10	13	232	255	255
16	229	252	254	49	12	0	0	7	7	0	70	237	252	235
6	141	245	255	212	25	11	9	3	0	115	236	243	255	137
0	87	252	250	248	215	60	0	1	121	252	255	248	144	6
0	13	113	255	255	245	255	182	181	248	252	242	208	36	0
1	0	5	117	251	255	241	255	247	255	241	162	17	0	7
0	0	0	4	58	251	255	246	254	253	255	120	11	0	1
0	0	4	97	255	255	255	248	252	255	244	255	182	10	0
0	22	205	252	246	251	241	100	24	113	255	245	255	194	9
0	111	255	242	255	158	24	0	0	6	39	255	232	230	56
0	218	251	250	137	7	11	0	0	2	62	255	250	125	3
0	173	255	255	101	9	20	0	13	3	13	182	251	245	61
0	107	251	241	255	230	98	55	19	118	217	248	253	255	52
0	18	146	250	255	247	255	255	255	249	255	240	255	129	0
0	0	23	113	215	255	250	248	255	255	248	248	118	14	12
0	0	6	1	0	52	153	233	255	252	147	37	0	0	4
0	0	5	5	0	0	0	0	14	1	0	6	6	0	0



A black and white (grayscale) image is represented as a 2D array

Here, each pixel value is an **integer from 0 to 255**

A colour image is represented as a 3D array



How do humans recognise a koala in a given image?



Koala's **eye**? = Y



Koala's **nose**? = Y



Koala's **ears**? = Y



Koala's **hands**? = Y



Koala's **legs**? = Y



Koala's **head**? = Y

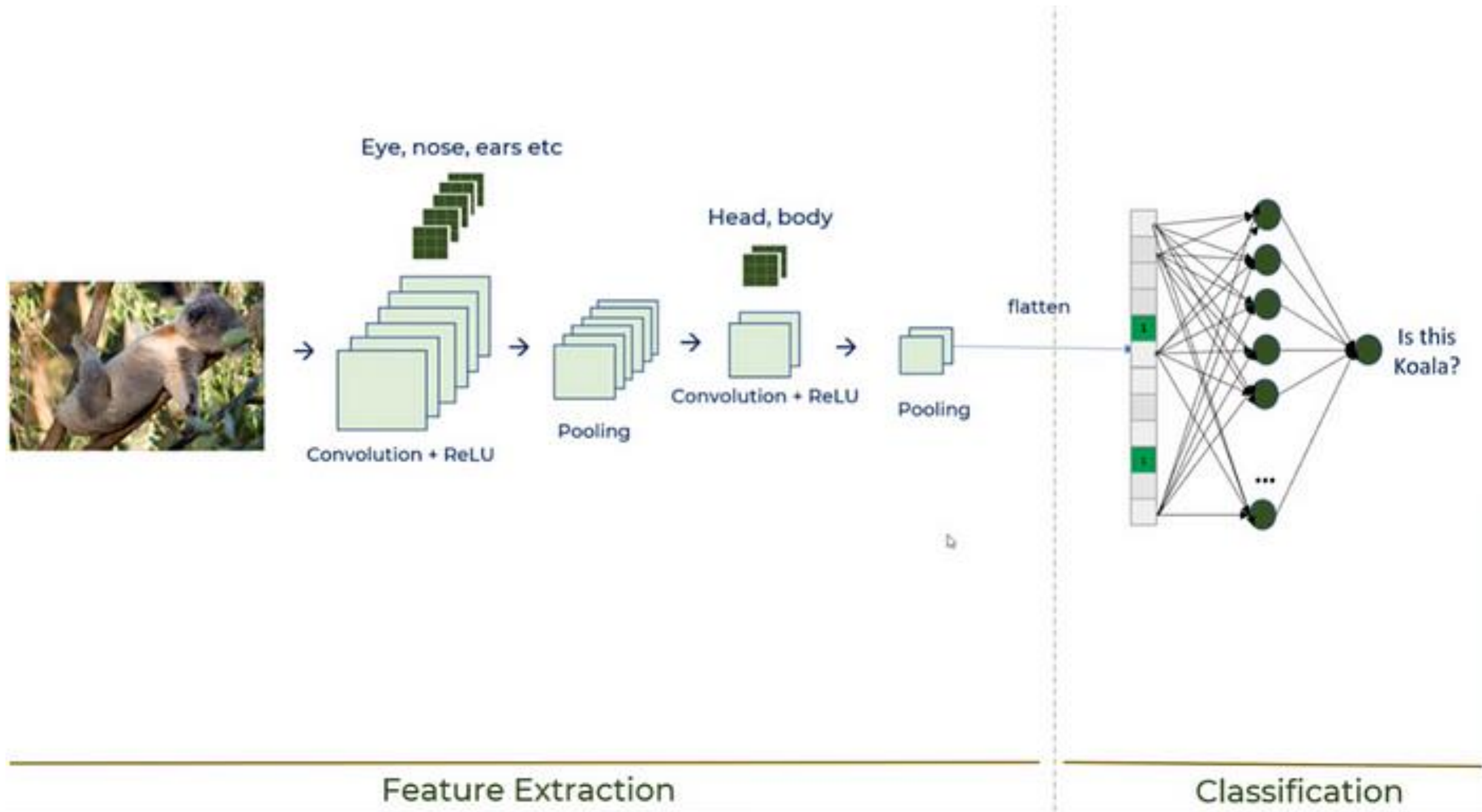


Koala's **body**? = Y



Is it **Koala**? = Y

CNN for Animal Classification

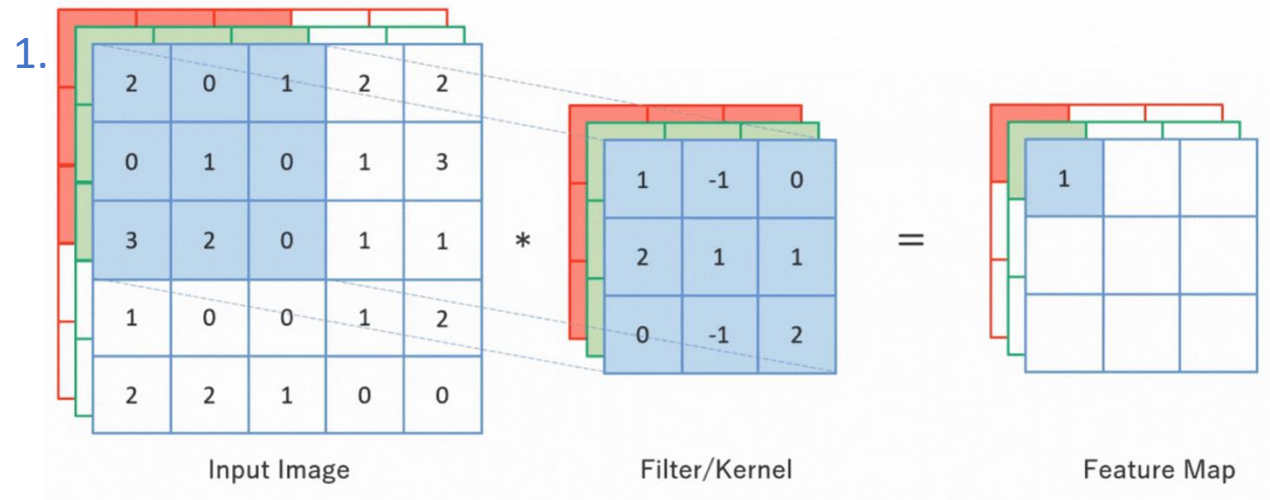


Convolution

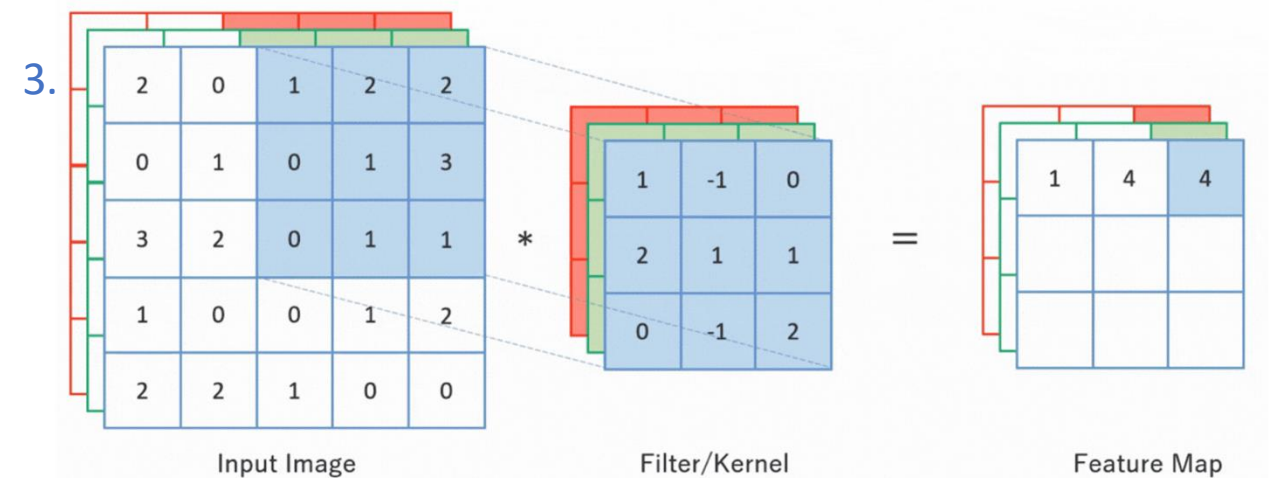
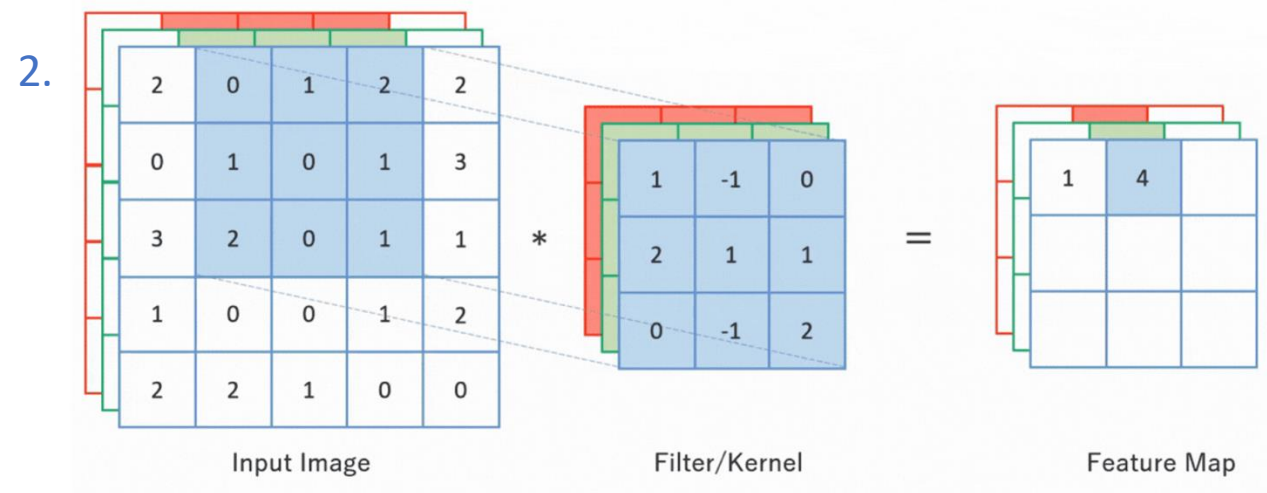
- Convolutions are defined by
 - A **kernel**, which is a matrix overlaid on the input and computes an element-wise product with the input
 - A **stride** which defines how many positions in the input to advance the kernel on each iteration (stride = 1 means the kernel will operate on every input)

Convolution is used in CNNs to automatically detect and learn useful patterns in data, especially images.

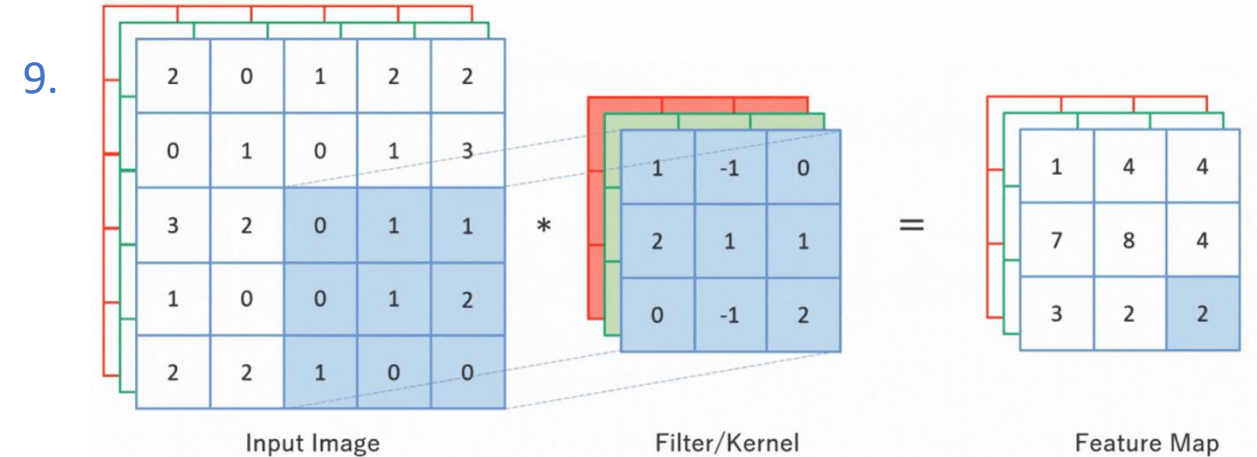
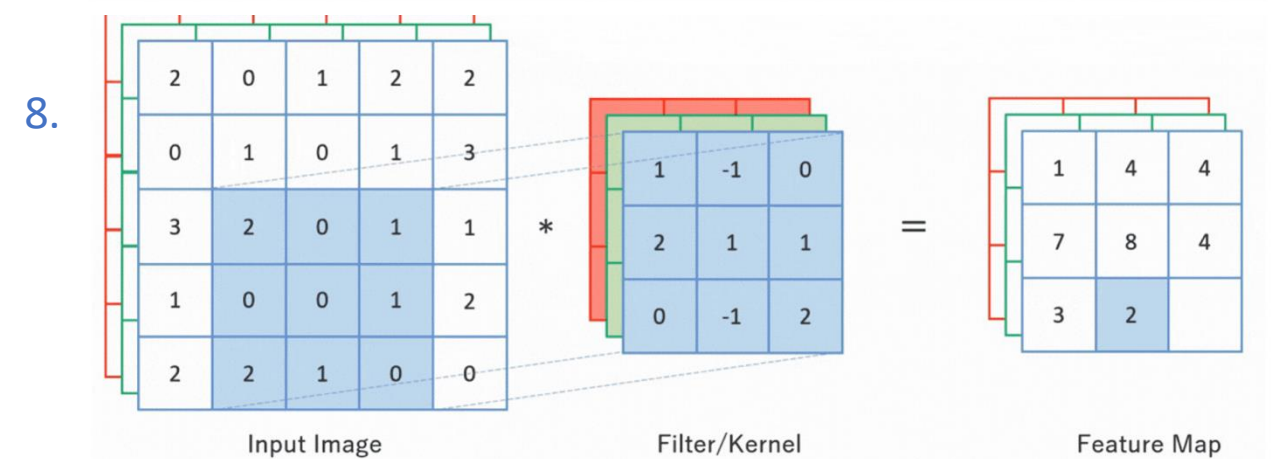
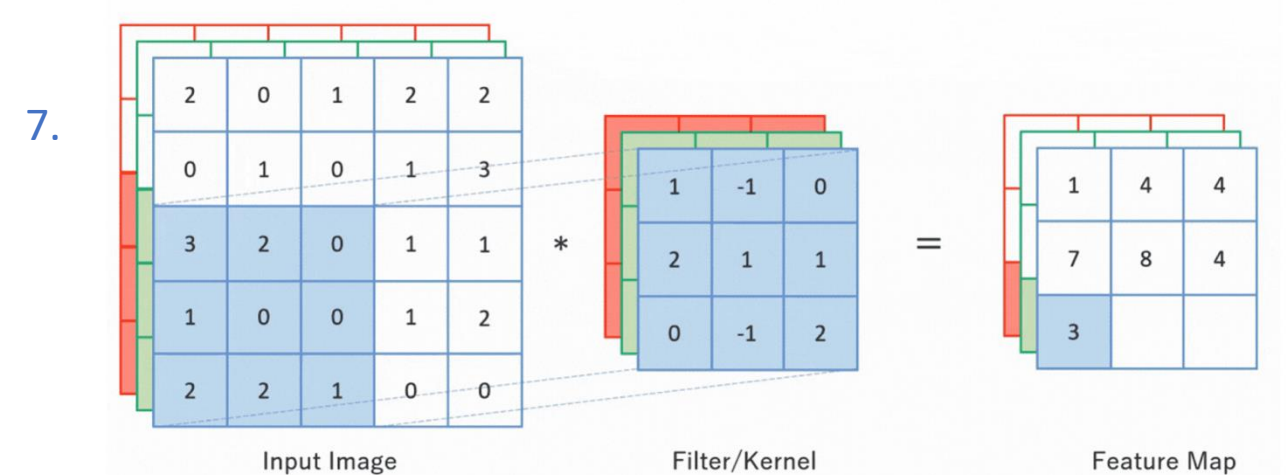
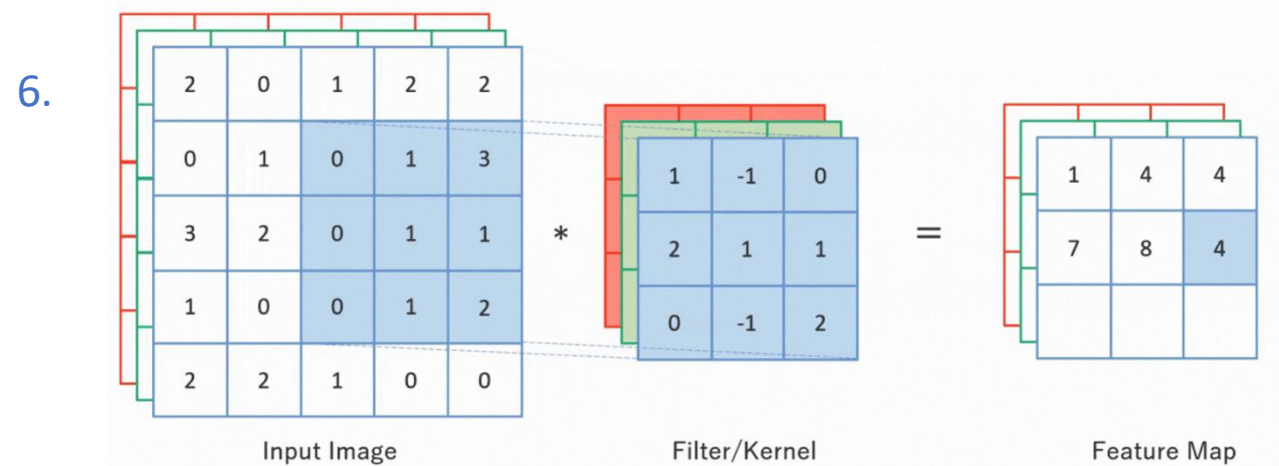
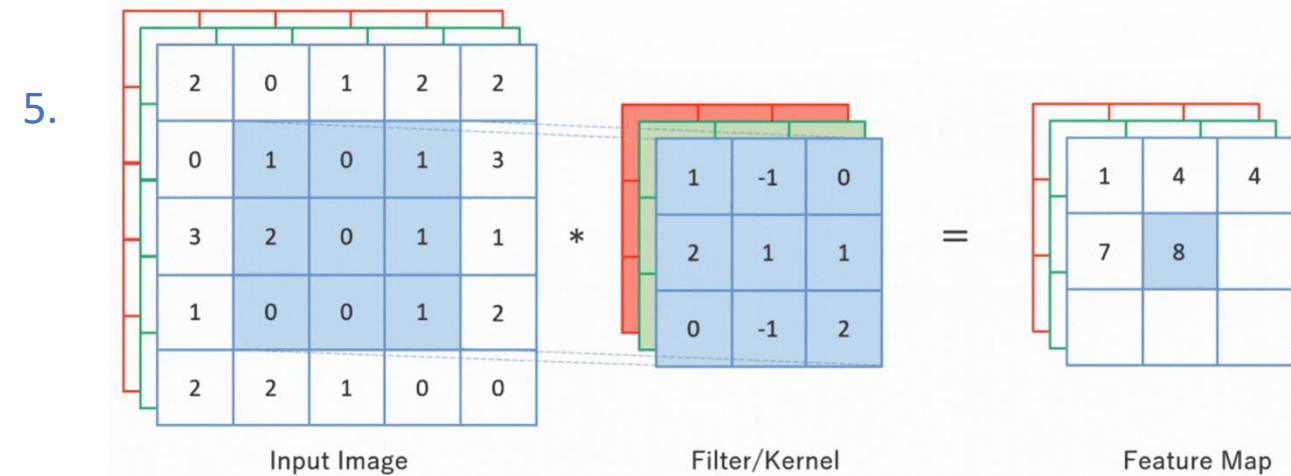
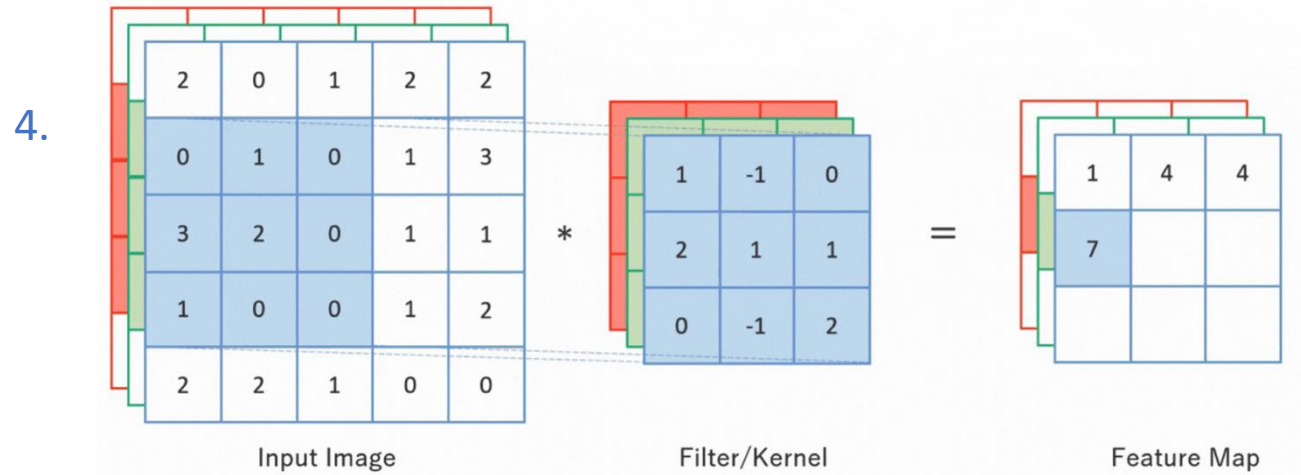
Convolution



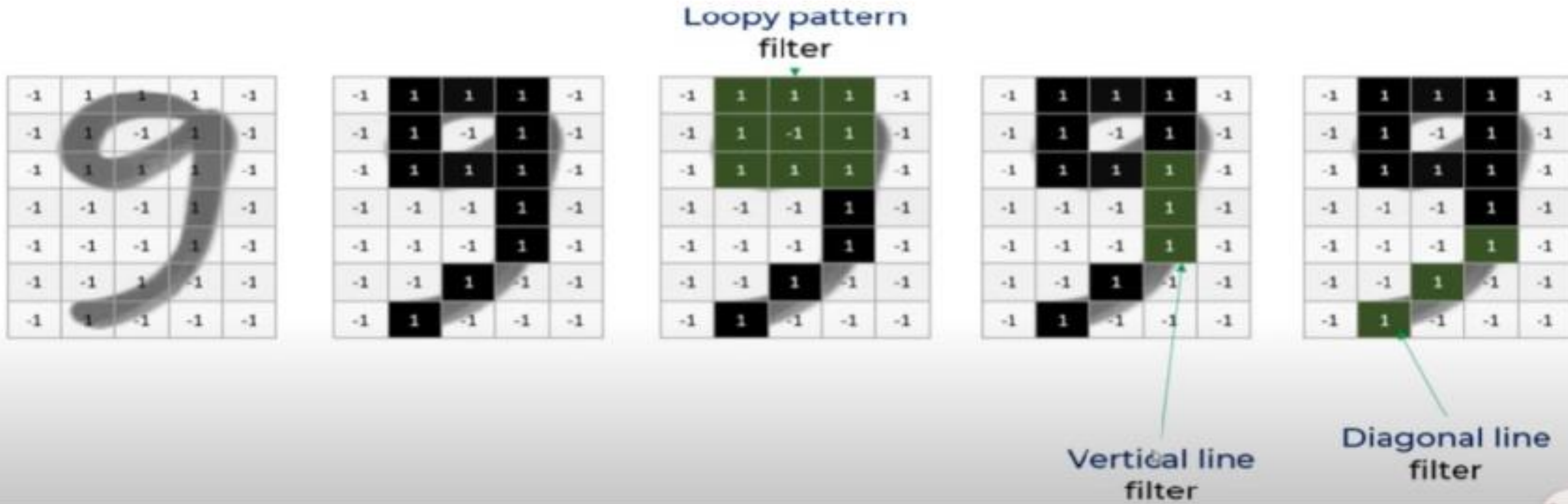
- Elementwise multiplication is performed here.
- Here stride = 1



Convolution



Convolution – Different Filters



Convolution helps the network find simple patterns like edges or shapes in small parts of the image.

Convolution - How feature space gets updated

Loopy pattern detector

9 *

1	1	1
1	-1	1
1	1	1

 =

	1	

Loopy pattern detector

6 *

1	1	1
1	-1	1
1	1	1

 =

	1	

Loopy pattern detector

8 *

1	1	1
1	-1	1
1	1	1

 =

	1	
	1	

Loopy pattern detector

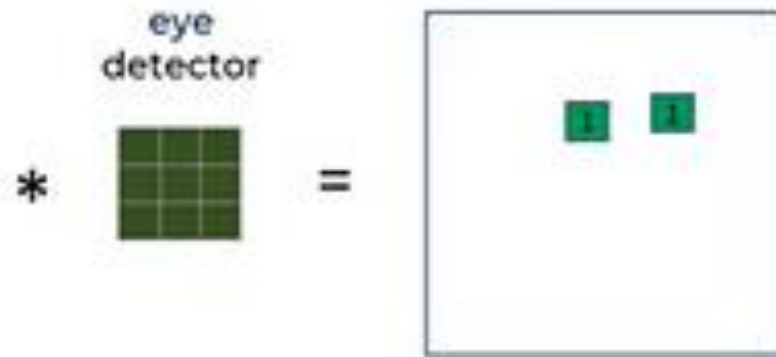
96 *

1	1	1
1	-1	1
1	1	1

 =

1			
			1

Convolution



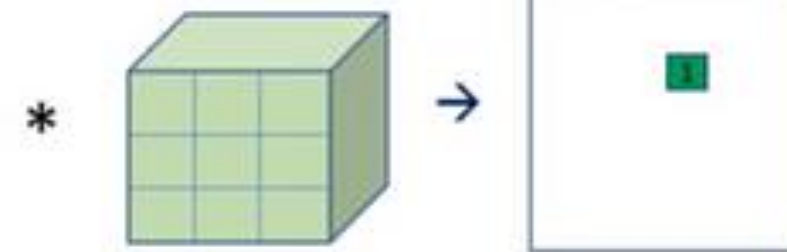
Location invariant: It can detect eyes in any location of the image



Convolution – Different filters for different features



Filter for head



CNNs learn filters that detect patterns like edges, shapes, and objects.

Early layers detect low-level features like edges and textures.

Deeper layers recognize complex objects and concepts.

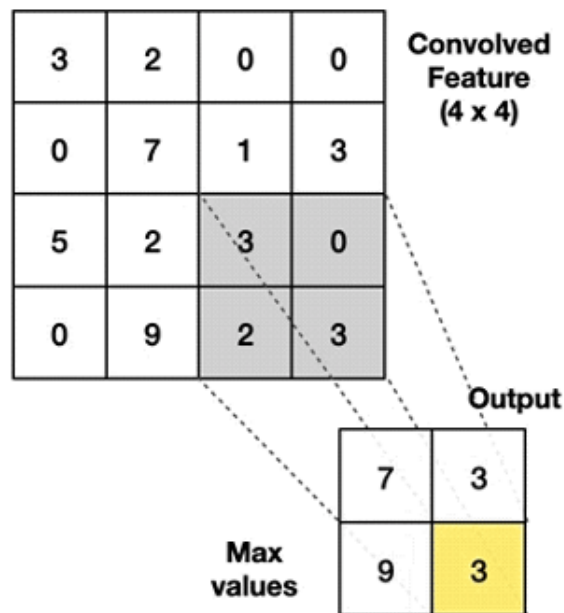
Max Pooling

Max pooling is a downsampling operation used in Convolutional Neural Networks (CNNs) to **reduce the spatial size** of feature maps while **keeping the most important information**.

Max Pooling

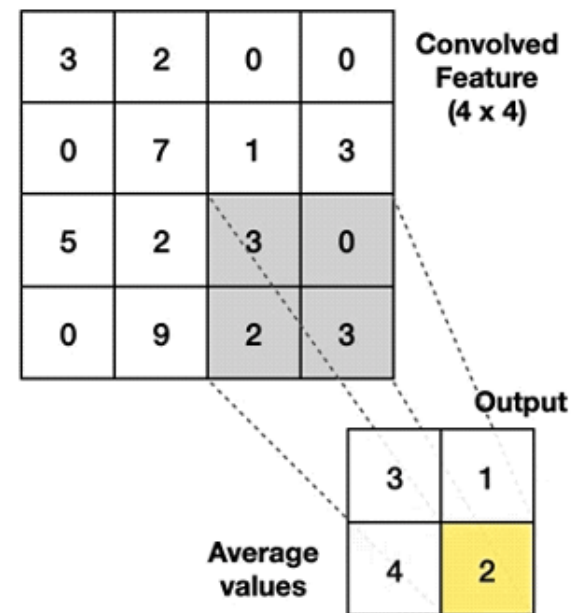
Take the **highest** value from the area covered by the kernel

Example: Kernel of size 2 x 2; stride=(2,2)

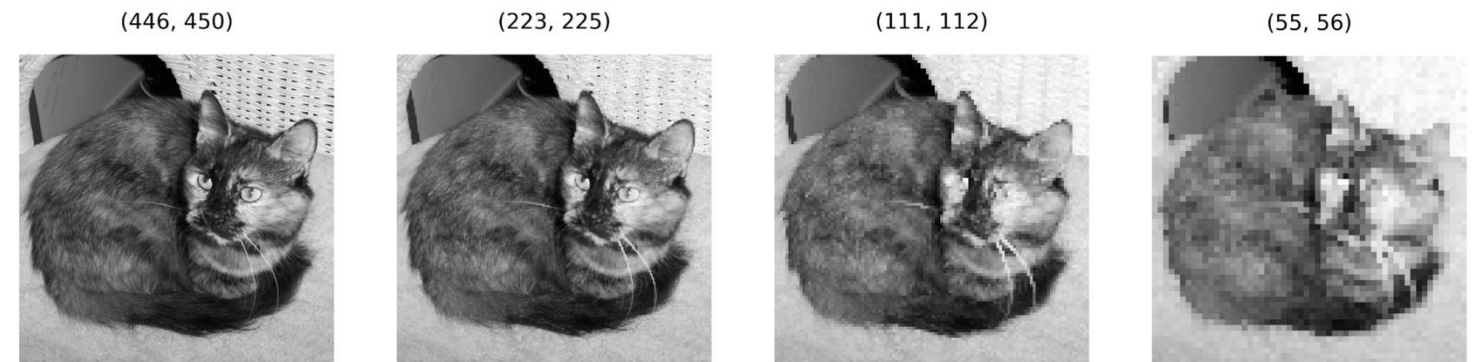


Average Pooling

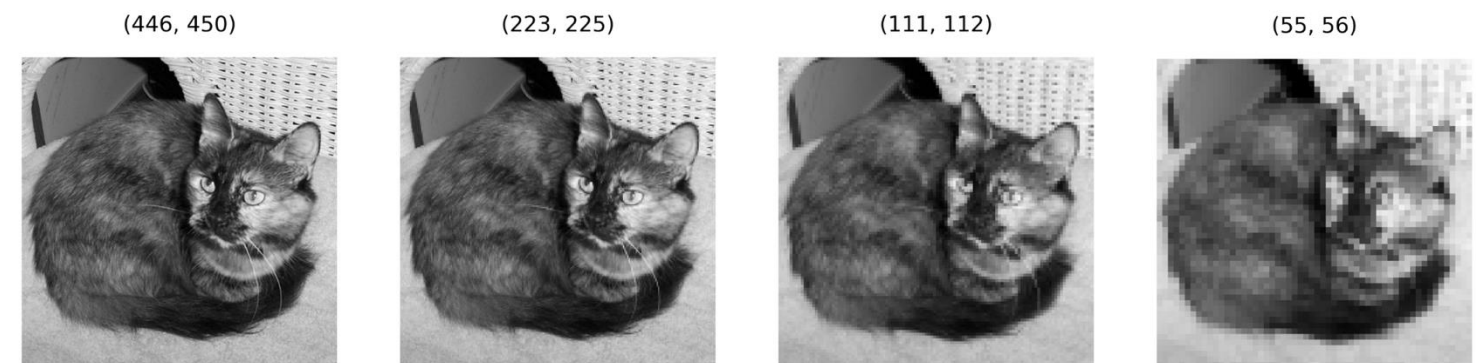
Calculate the **average** value from the area covered by the kernel



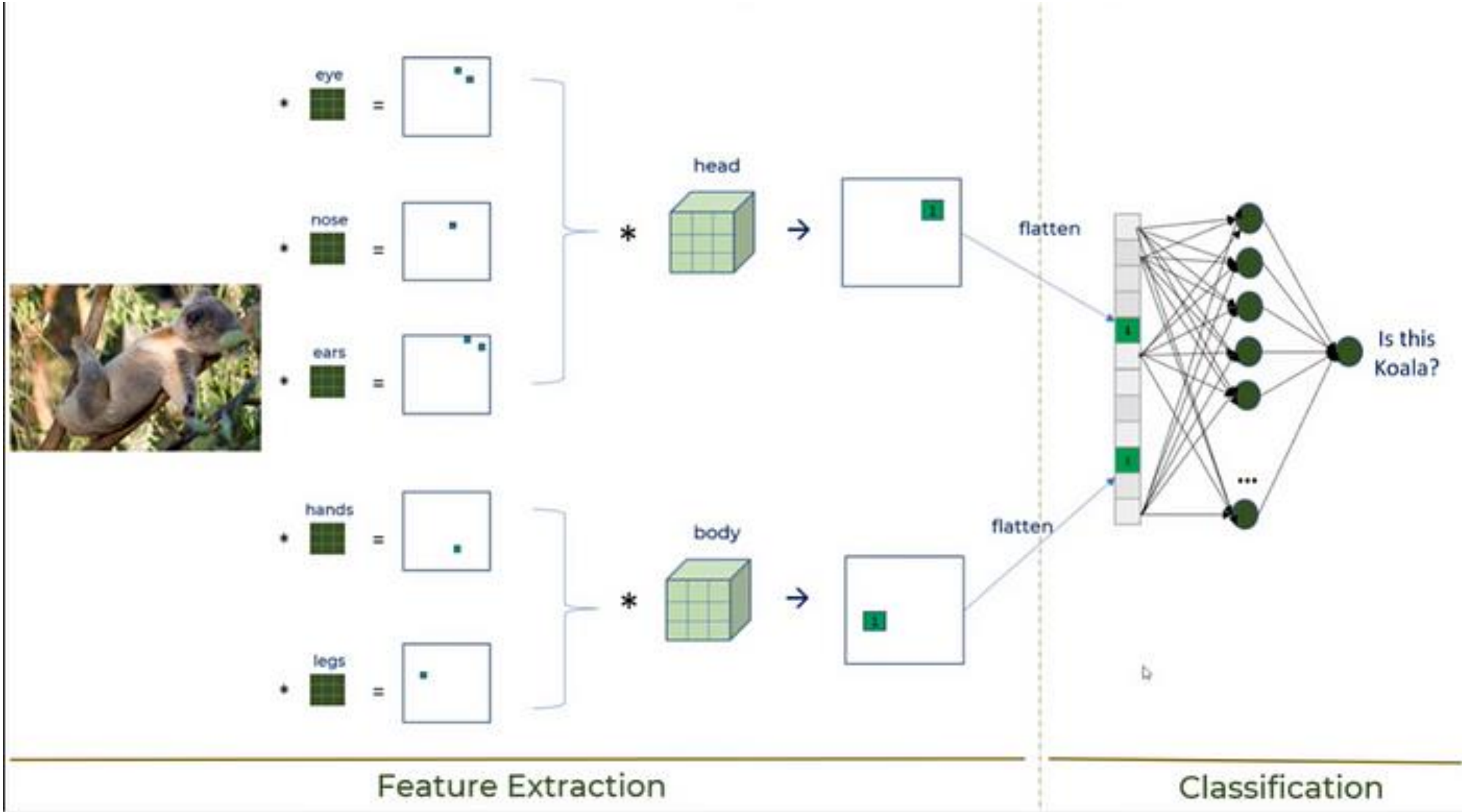
After 3 iterations:
max pooling



average pooling



Once the features have been extracted, it is flattened into a single vector and connected to a fully connected layer to classify koala or not.



Fully-connected vs. Convolutional

Fully-connected layer

- Each neuron is connected to every neuron in the last layer
- The neuron learns some combination of the last layer's responses
- The output to the next layer is the neuron's response

Convolutional layer

- Each neuron is connected to a small patch of all of the last layer's outputs
- The neuron learns a convolutional kernel
- The output to the next layer is the input convolved with the neuron's kernel

In MLPs, the model *learns features* by combining all inputs directly — but it can't know that “nearby pixels” are related unless it's taught that.

In CNNs, the model *learns features* from small parts of the image and builds up a full understanding layer by layer — preserving the image structure

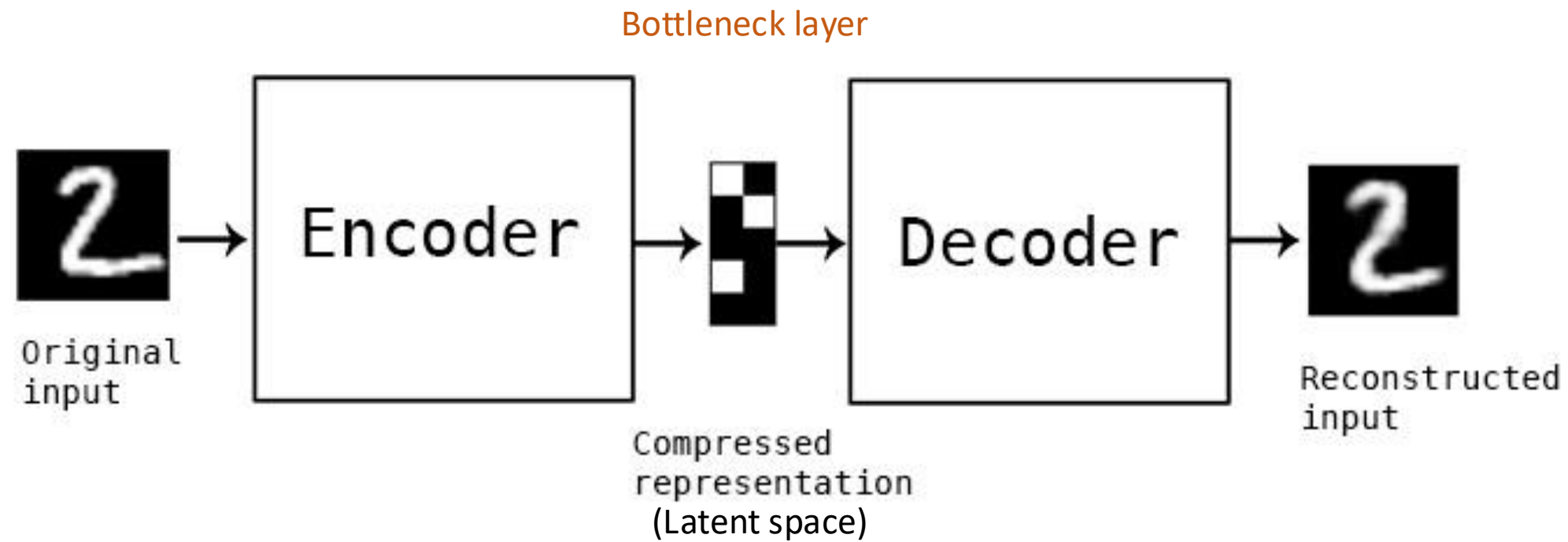
Feature	Fully Connected NN (MLP)	CNN
Learns features automatically	Yes	Yes
Handles spatial data (images)	Not ideal	Excellent
Preserves spatial structure	No	Yes
Number of parameters	High (dense)	Lower (shared filters)
Best suited for	Tabular data, flat vectors	Images, signals, videos

Generative model approaches

- Goal: model the probability density function (PDF) of what you want to generate (e.g., images)
- Often hard to model the density function directly
- Instead:
 - Map to a lower-dimensional “latent” space – **autoencoders**, variational autoencoders (VAE)
 - Learn a function to convert samples from a simple PDF (e.g., Gaussian) into the target PDF, to allow sampling from the PDF – **generative adversarial networks (GAN)**
 - Learn the gradient of the PDF and move along the gradient to generate more-probable samples – score-based models, **diffusion models**

Autoencoders

Neural networks designed to learn the latent representation of input in an unsupervised manner. The output of the network is whatever was passed to the network (e.g., an image)



Encoder

Compresses input data into a lower-dimensional representation called the latent latent space.

Bottleneck

The compressed representation forces the network to learn the most important features.

Decoder

Reconstructs the original input from the compressed representation.

Evaluation

Measures reconstruction quality using loss metrics like MSE.

Q4

What is an **autoencoder** and how is it trained?

What Is an Autoencoder?

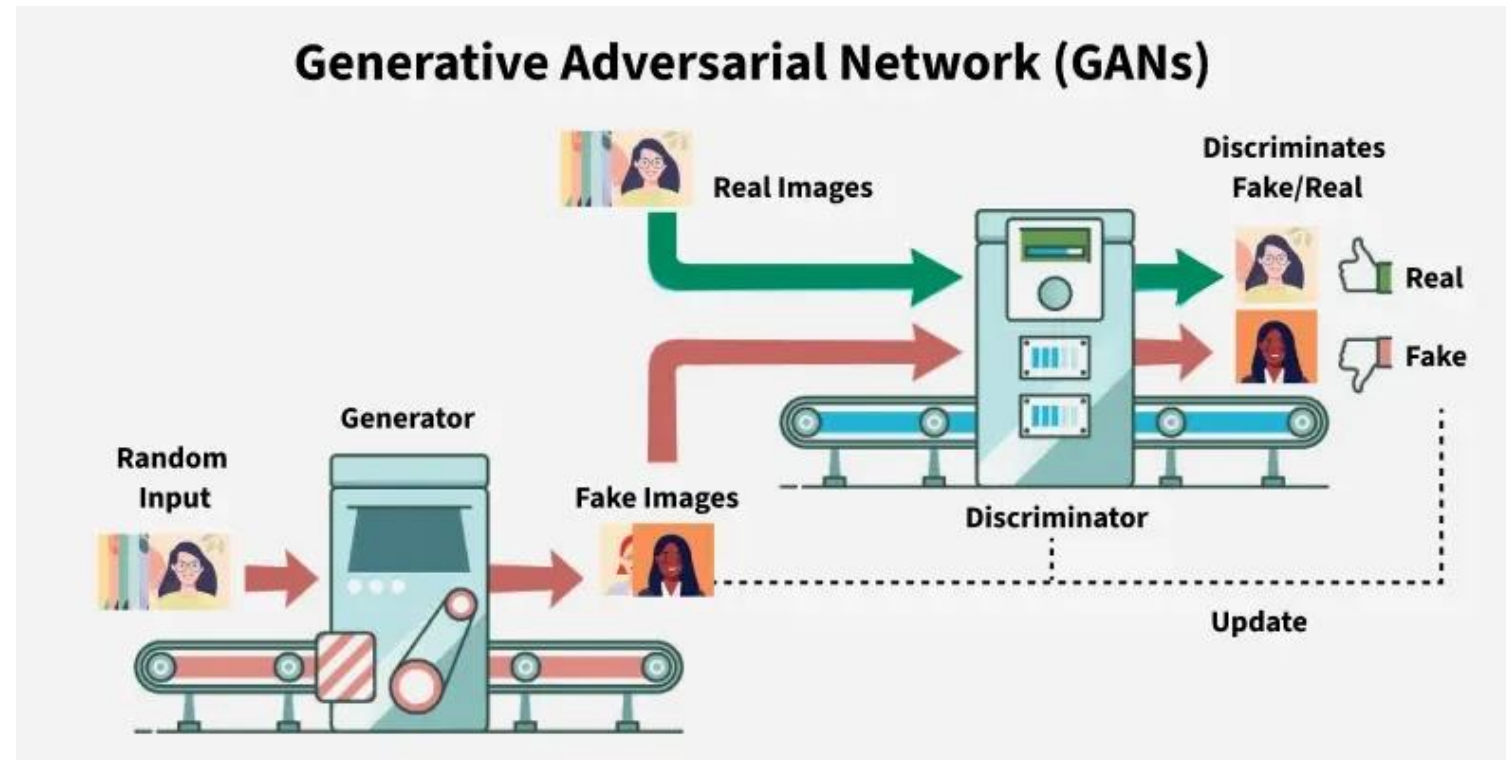
- An **autoencoder** is a type of neural network used for:
 - **Data compression**
 - **Dimensionality reduction**
 - **Learning useful representations (embeddings)**

How It's Trained

- The model is trained to **recreate the input** (i.e., $\text{output} \approx \text{input}$).
- The **loss function** measures the difference between:
 - The original input and
 - The reconstructed output.

Autoencoders are used to learn low-dimensional embeddings of complex data which can be used in other tasks. Some types of autoencoders can be used as generative models

GANs: Generative Adversarial Networks



Generative Adversarial Networks (GANs) are neural networks that learn to generate instances from a particular distribution (e.g., images of faces)

GAN's architecture consists of two neural networks:

1. Generator: The Generator takes random values as its input, and generates a data instance (for example, a generated image) as its output. The goal of the Generator is to create outputs that look like real data so that the discriminator cannot distinguish them from real data.

2. Discriminator: The Discriminator takes a data instance as input, which can be either a "real" instance from the training data or a "fake" instance created by the Generator. The Discriminator's output is a class label "real" or "fake." The goal of the Discriminator is to correctly classify real instances as real and fake instances as fake.

GAN Training

A **Generative Adversarial Network (GAN)** has two parts:

1. **Generator (G)**: Takes a random vector $z \sim p_z(z)$ and outputs a fake sample $G(z)$.
2. **Discriminator (D)**: Takes an input sample (real or fake) and outputs a probability $D(x) \in [0, 1]$, representing how likely the input is real.

Minimax Loss

The overall objective is:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}} [\log D(x)] + \mathbb{E}_{z \sim p_z} [\log(1 - D(G(z)))]$$

Probability of predicting fake image as fake

Probability of predicting real image as real

Meaning:

- $D(x)$: This is the probability the **Discriminator assigns to a real image** being real.
- $D(G(z))$: This is the probability the **Discriminator assigns to a fake image** (created by the Generator) being real.
- p_{data} : Real data distribution.
- p_z : Noise distribution (e.g., uniform or normal).

GAN Training

A **Generative Adversarial Network (GAN)** has two parts:

1. **Generator (G)**: Takes a random vector $z \sim p_z(z)$ and outputs a fake sample $G(z)$.
2. **Discriminator (D)**: Takes an input sample (real or fake) and outputs a probability $D(x) \in [0, 1]$, representing how likely the input is real.

Minimax Loss

Expectation is just the average over many samples.

The overall objective is:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}} [\log D(x)] + \mathbb{E}_{z \sim p_z} [\log(1 - D(G(z)))]$$

Take all fake image instances $G(z)$, compute $\log(1 - D(G(z)))$ for each, then average them

Meaning:

Take all real image instances, compute $\log D(x)$ for each, then average them.

- $D(x)$: This is the probability the **Discriminator assigns to a real image** being real.
- $D(G(z))$: This is the probability the **Discriminator assigns to a fake image** (created by the Generator) being real.
- p_{data} : Real data distribution.
- p_z : Noise distribution (e.g., uniform or normal).

GAN Training

A **Generative Adversarial Network (GAN)** has two parts:

1. **Generator (G)**: Takes a random vector $z \sim p_z(z)$ and outputs a fake sample $G(z)$.
2. **Discriminator (D)**: Takes an input sample (real or fake) and outputs a probability $D(x) \in [0, 1]$, representing how likely the input is real.

Minimax Loss

The overall objective is:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}} [\log D(x)] + \mathbb{E}_{z \sim p_z} [\log(1 - D(G(z)))]$$

Probability of predicting fake image as fake

Probability of predicting real image as real

Meaning:

- $D(x)$: This is the probability the **Discriminator assigns to a real image** being real.
- $D(G(z))$: This is the probability the **Discriminator assigns to a fake image** (created by the Generator) being real.
- p_{data} : Real data distribution.
- p_z : Noise distribution (e.g., uniform or normal).

Discriminator wants to maximise this. i.e., correctly classify real images as real and fake images as fake.

GAN Training

A **Generative Adversarial Network (GAN)** has two parts:

1. **Generator (G)**: Takes a random vector $z \sim p_z(z)$ and outputs a fake sample $G(z)$.
2. **Discriminator (D)**: Takes an input sample (real or fake) and outputs a probability $D(x) \in [0, 1]$, representing how likely the input is real.

Minimax Loss

The overall objective is:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}} [\log D(x)] + \boxed{\mathbb{E}_{z \sim p_z} [\log(1 - D(G(z)))]}$$

Probability of predicting fake image as fake
↓

Meaning:

- $D(x)$: This is the probability the **Discriminator assigns to a real image** being real.
- $D(G(z))$: This is the probability the **Discriminator assigns to a fake image** (created by the Generator) being real.
- p_{data} : Real data distribution.
- p_z : Noise distribution (e.g., uniform or normal).

Generator wants to

minimise $\mathbb{E}_z[\log(1 - D(G(z)))]$,

i.e., reduce the discriminator's probability of predicting fake image as fake or maximise $D(G(z))$, i.e., increase the discriminator's probability of predicting fake image as real.

- How to tell if a GAN has learned?
- Ideally:
 - Outputs should look like inputs (look “real” and not “fake”)
 - Outputs should not be identical to inputs (memorized training data)
 - Outputs should be as diverse as real data (avoid **mode collapse** = the generator only creates one or a few outputs)

Problem	Cause	What You See	Why It's Bad
Discriminator (D) too strong	D learns to classify real and fake correctly very quickly	D always wins	G gets no gradient from D, causing G to stop learning
Generator (G) too strong	G fools D every time. D gets confused.	D outputs 0.5 for all	D stops learning
Mode collapse	G finds one great fake image and keeps generating that same image over and over	G outputs one/few samples only	No diversity