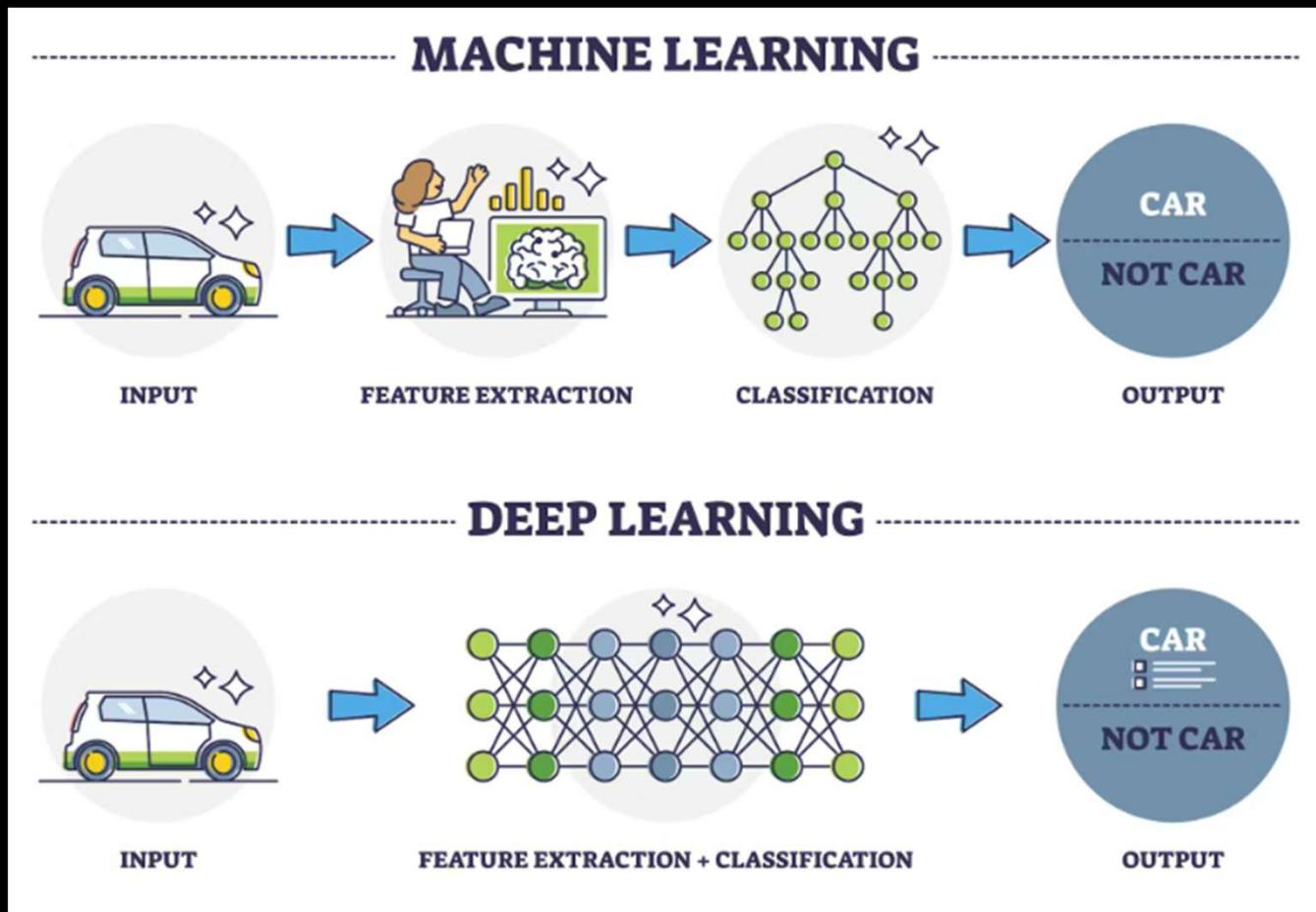




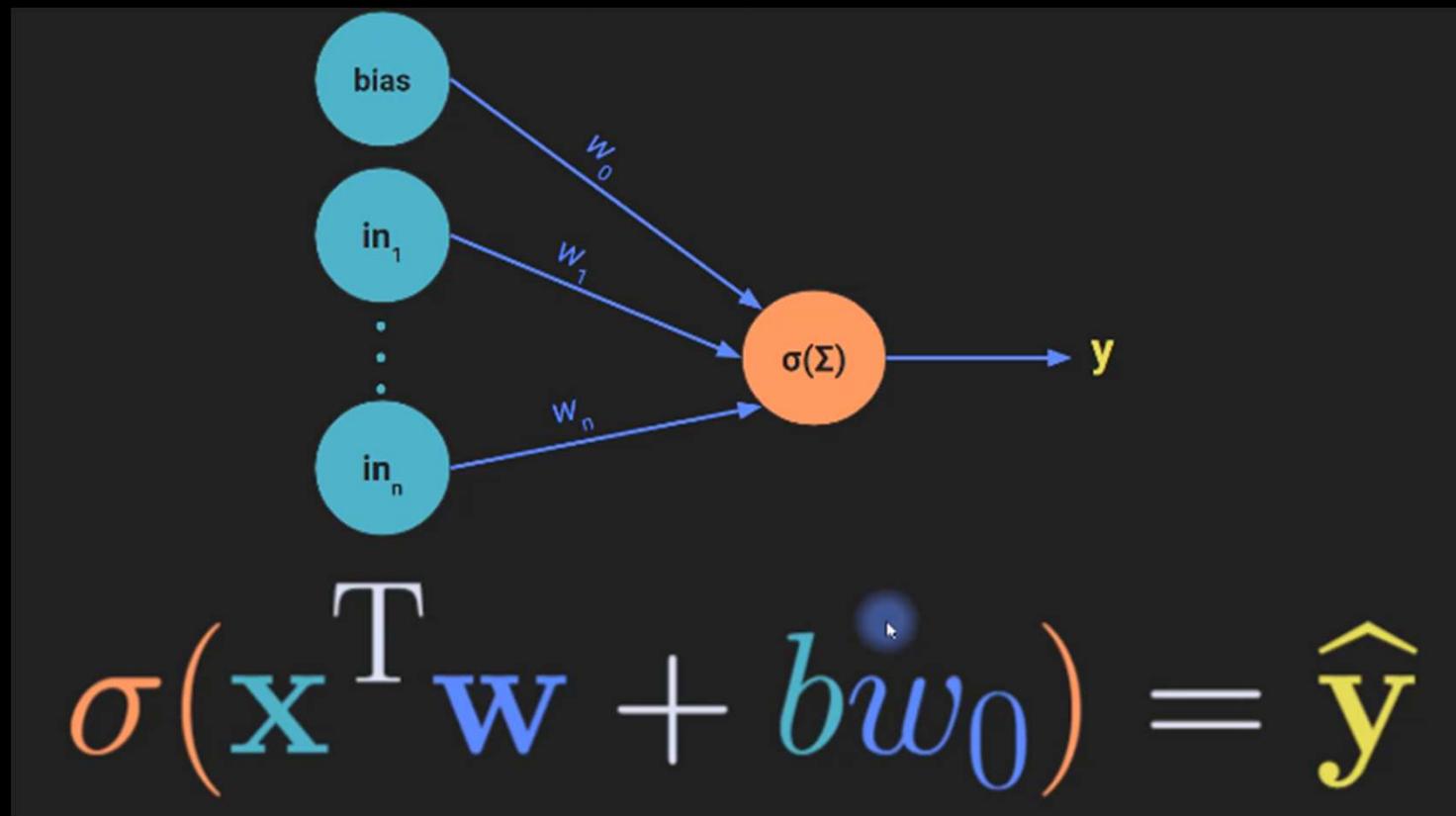
Generative AI Academy

# Deep Learning

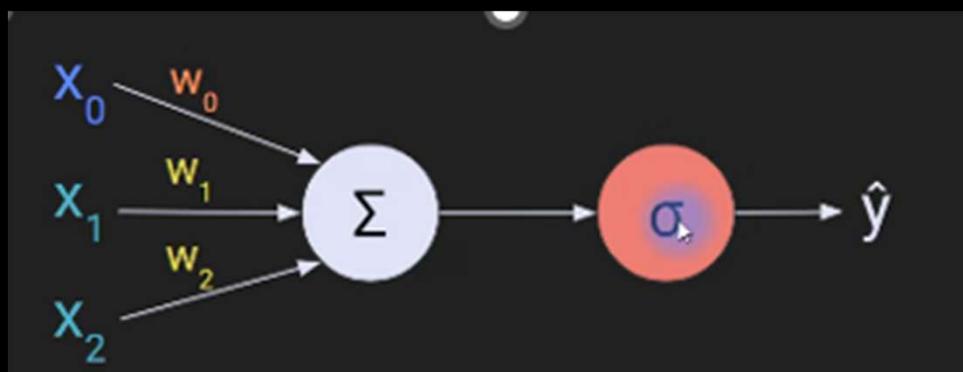
# Deep Learning



## Step 1: Linear transformation (Weighted sum of inputs + Bias)



## Step 2: Non linear function



$X_0$  = Bias

$X_1, X_2$ : Features/Inputs

$W_0, W_1, W_2$ : Weights

Sigma: Linear weighted sum

Theta: Non Linear Activation function

$Y$  = output

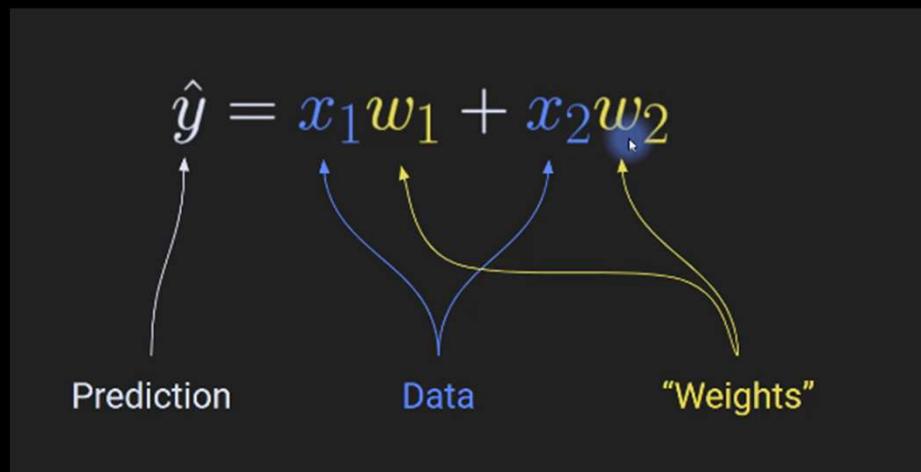
Deep Learning excels in non linear solutioning

# Transformation and activation functions

Step 1: Linear transformation

Output: Weighted sum of inputs

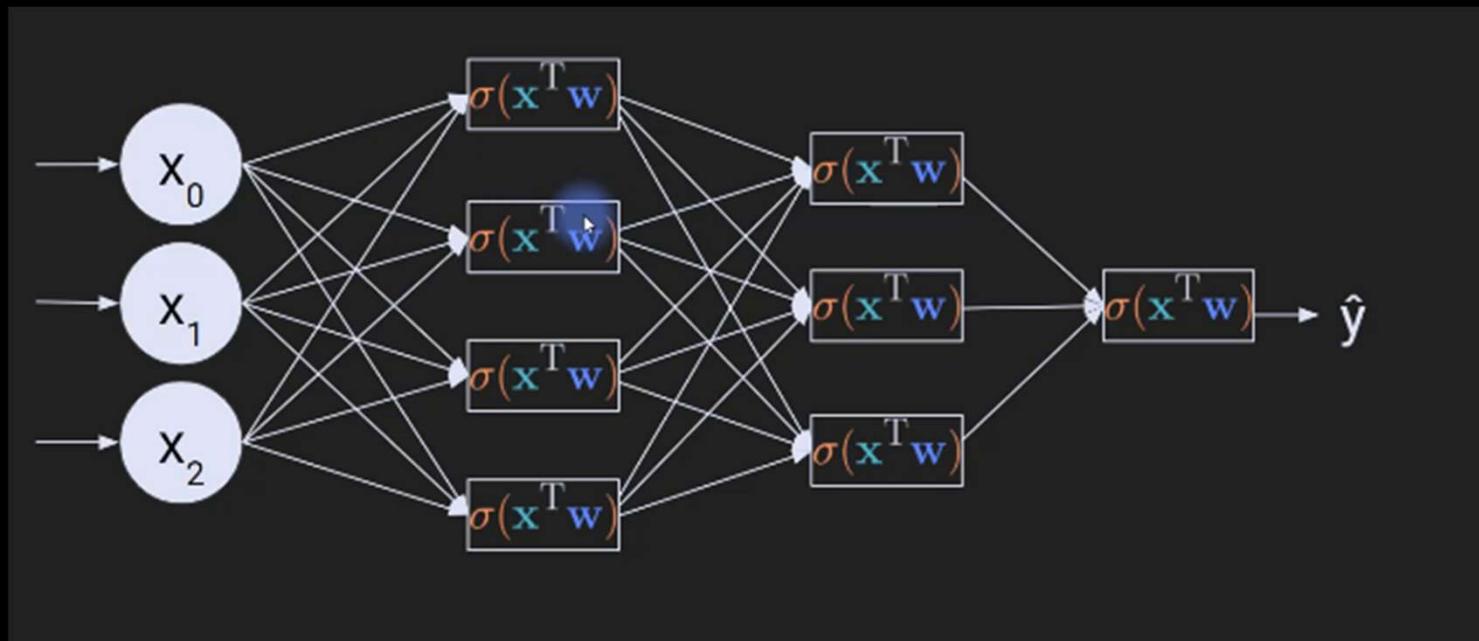
Step 2: Activation function (non linear),  
eg ReLU. Softmax or Sigmoid  
Output: activated value



$$\hat{y} = \sigma(x_1w_1 + x_2w_2)$$
$$\hat{y} = \log(x_1w_1 + x_2w_2)$$

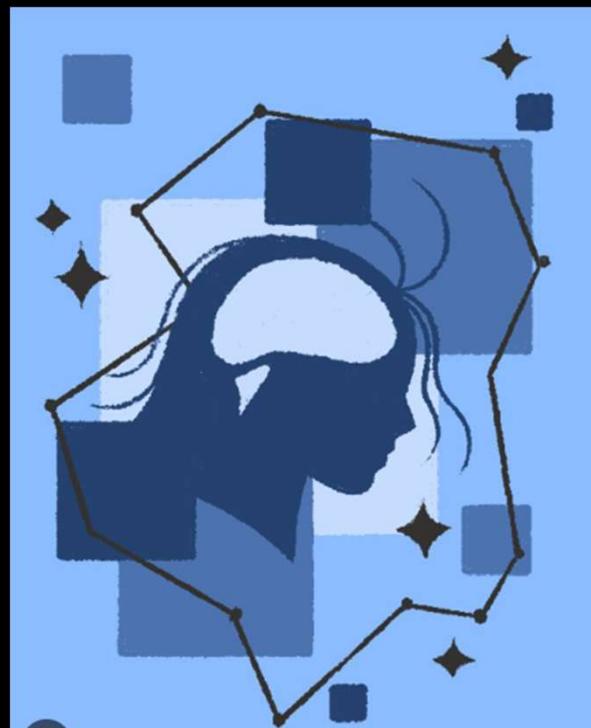
Model “Learns” the weights by a mechanism called Backpropagation

# A fully connected neural network



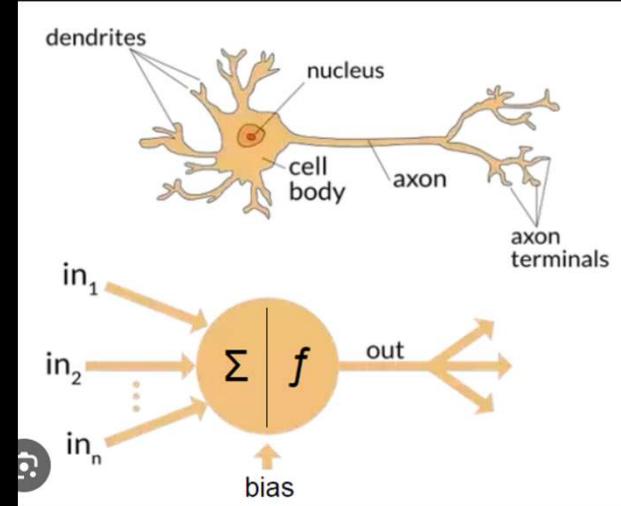
Nodes, Layers, Weights

# Neural Network

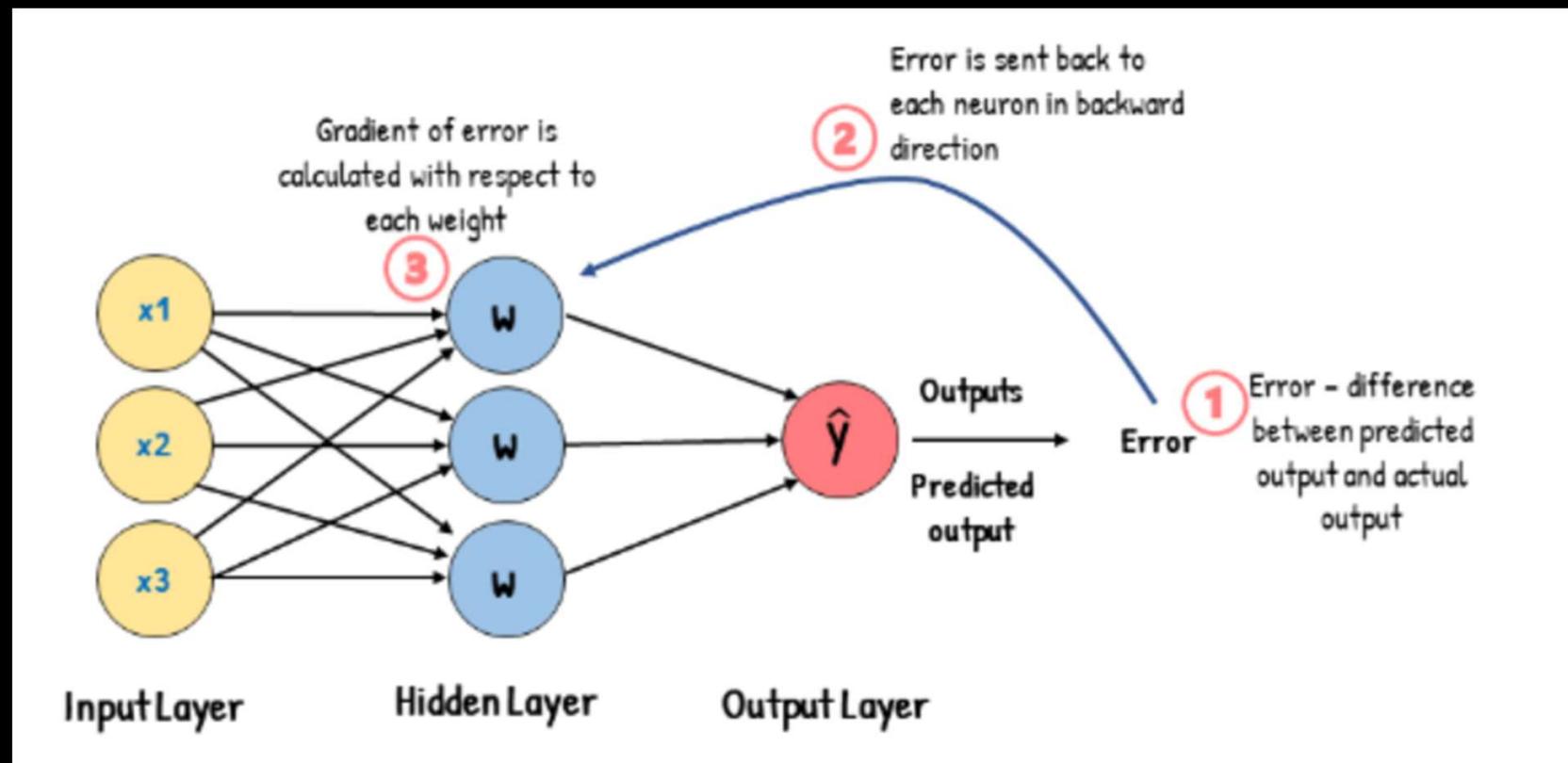


**Neural Network**  
[nur-əl 'net-wərk]

A series of algorithms that endeavors to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates.

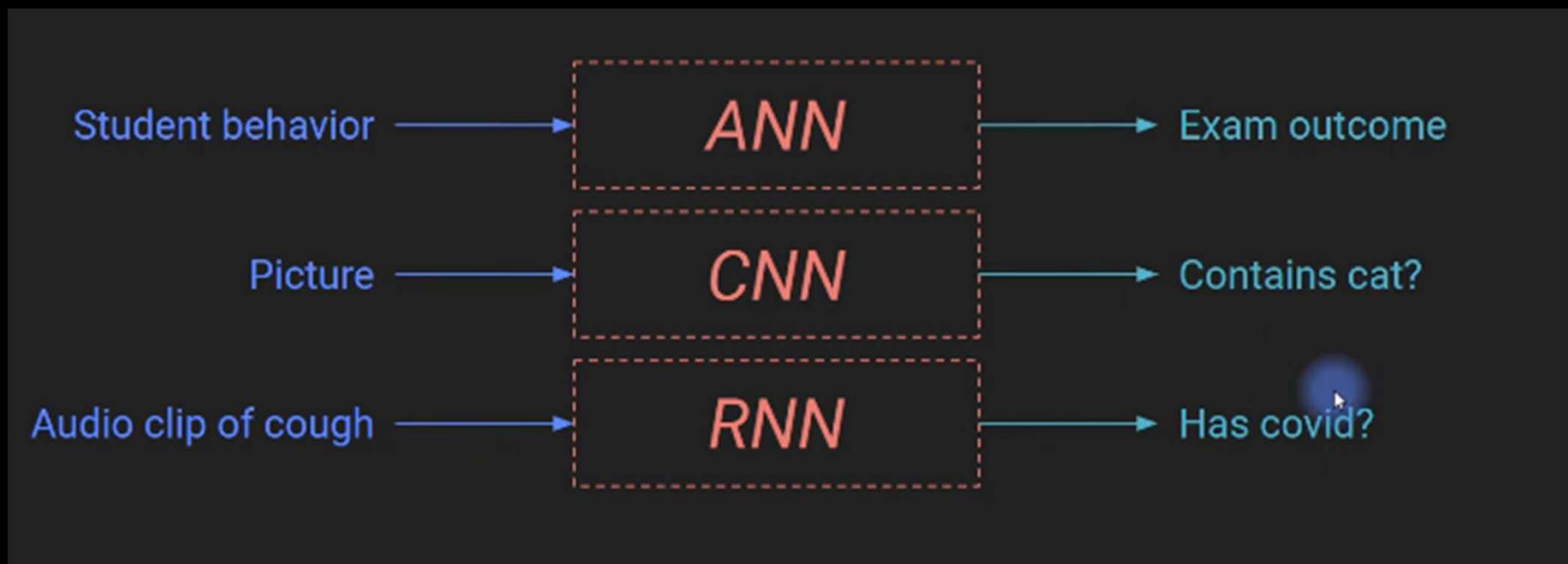


# Neural Network



Nodes, Activation Functions, Layers, Weights and bias, Loss function, Gradient descent, back propagation

# ANN, CNN, RNN

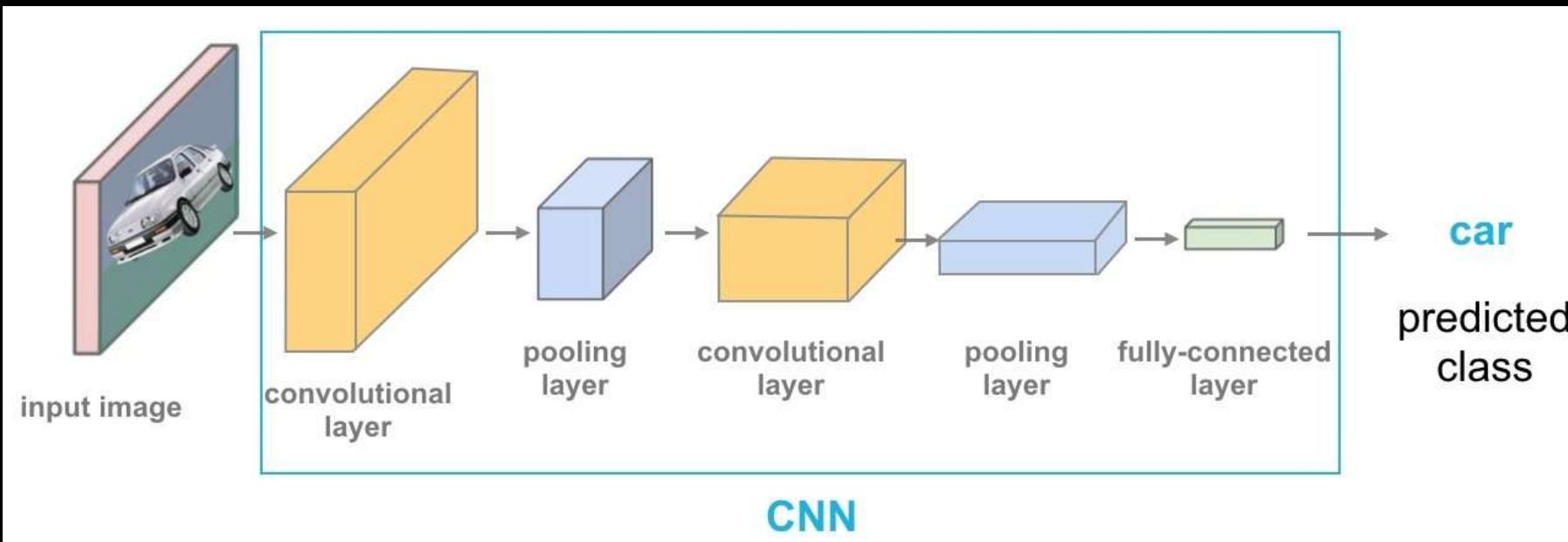


ANN - Artificial Neural Network

CNN – Convolutional Neural Network (typically used for Image input)

RNN – Recursive Neural Network (typically used for timeseries input)

# Convolutional Neural Network (CNN)



**Convolution layer:** Detects features such as edges, shapes resulting in a feature map

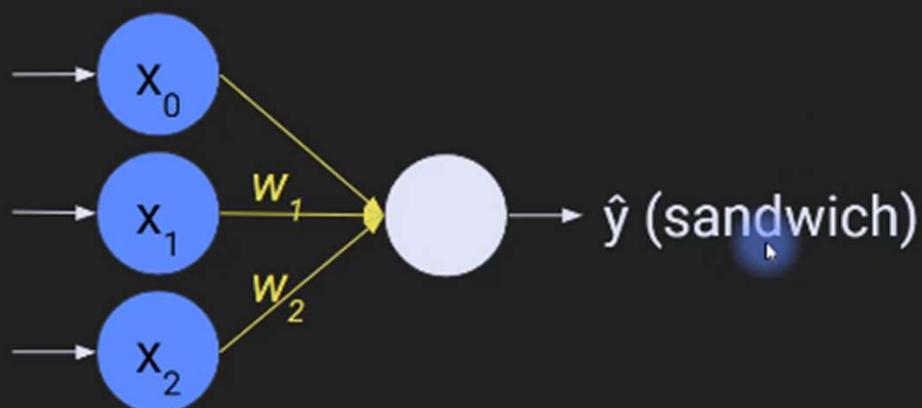
**Max Pooling Layer:** Reduces the size of the feature map, focusing only on prominent features

**Flattening:** Convert the 2D pooled feature map into a 1D vector

**Dense Layers:** Combines the features to make a prediction

**Softmax Output layer:** Outputs the possibilities of each digit class (0-9)

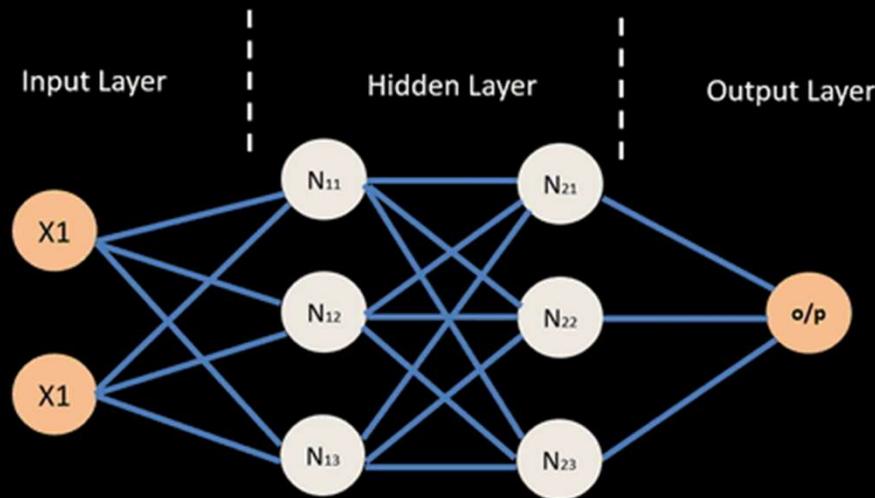
# Deep Learning



Backward propagation

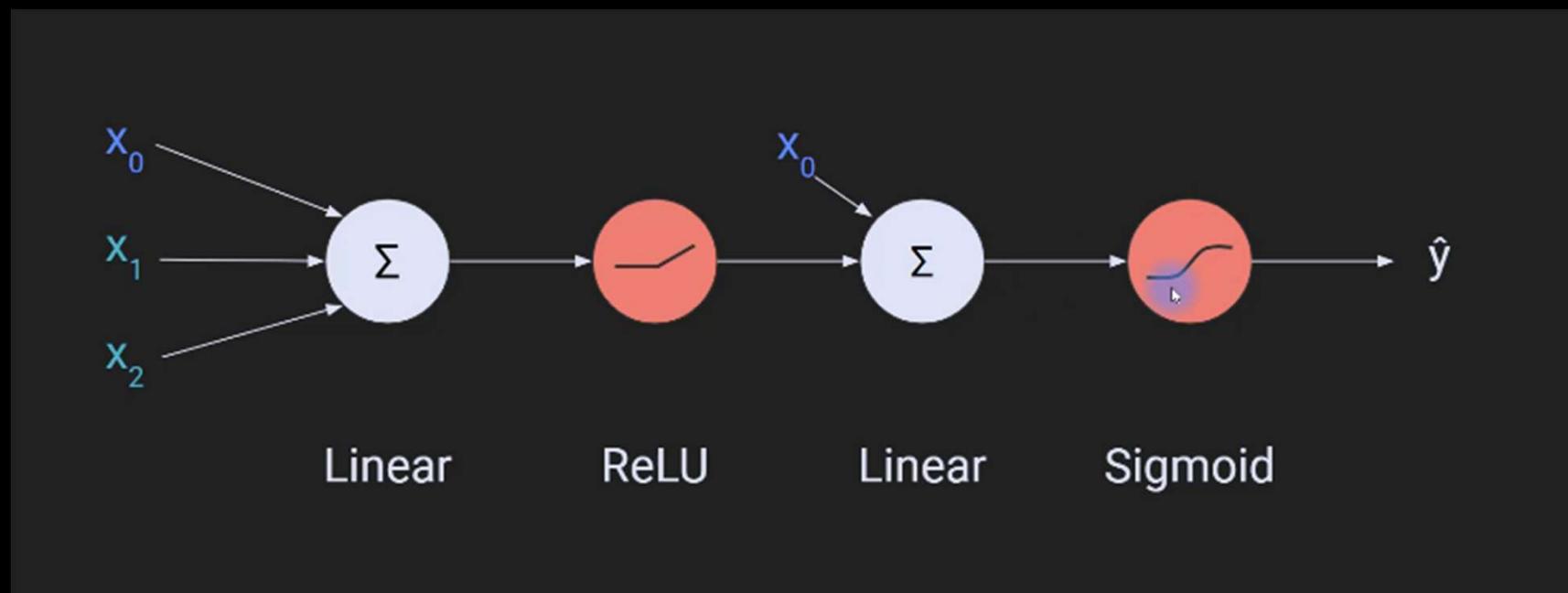
Adjust weights and biases based on error in output

# Backward propagation



Backward propagation  
Adjust weights and biases based on error in output

# Multi layer flow



Sigmoid: typically used as activation function for output layer for binary classification

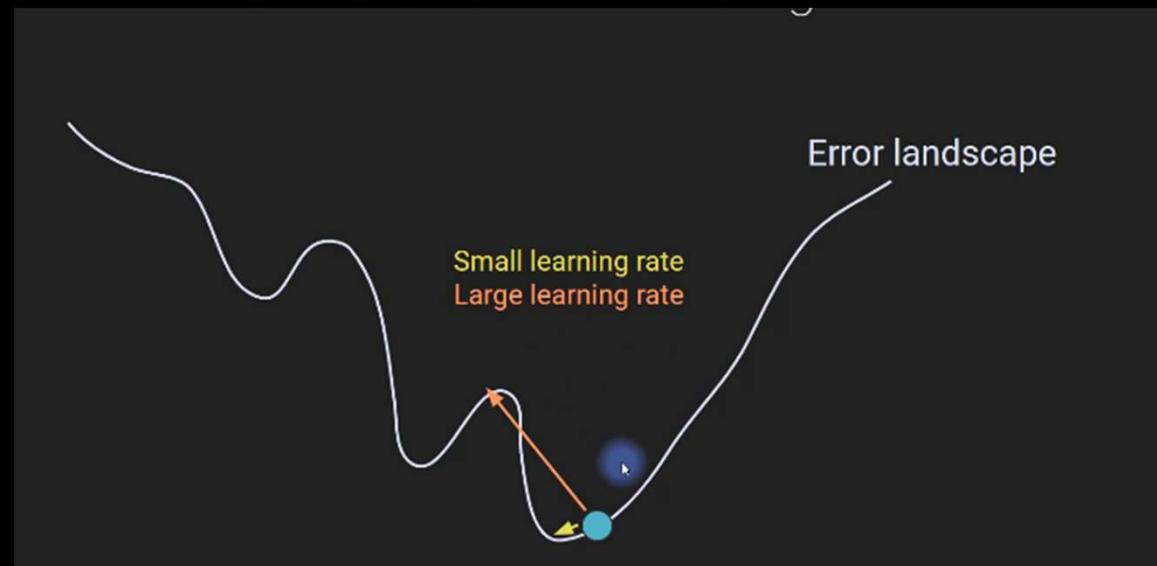
# Learning rate

The **learning rate** in a neural network is a hyperparameter that controls how much the model's weights are adjusted during each step of the training process.

It determines the **step size** at each iteration while moving toward the optimal weights that minimize the model's loss function.

Too high learning rate can lead to overshooting.

Too low learning rate can lead to slow convergence

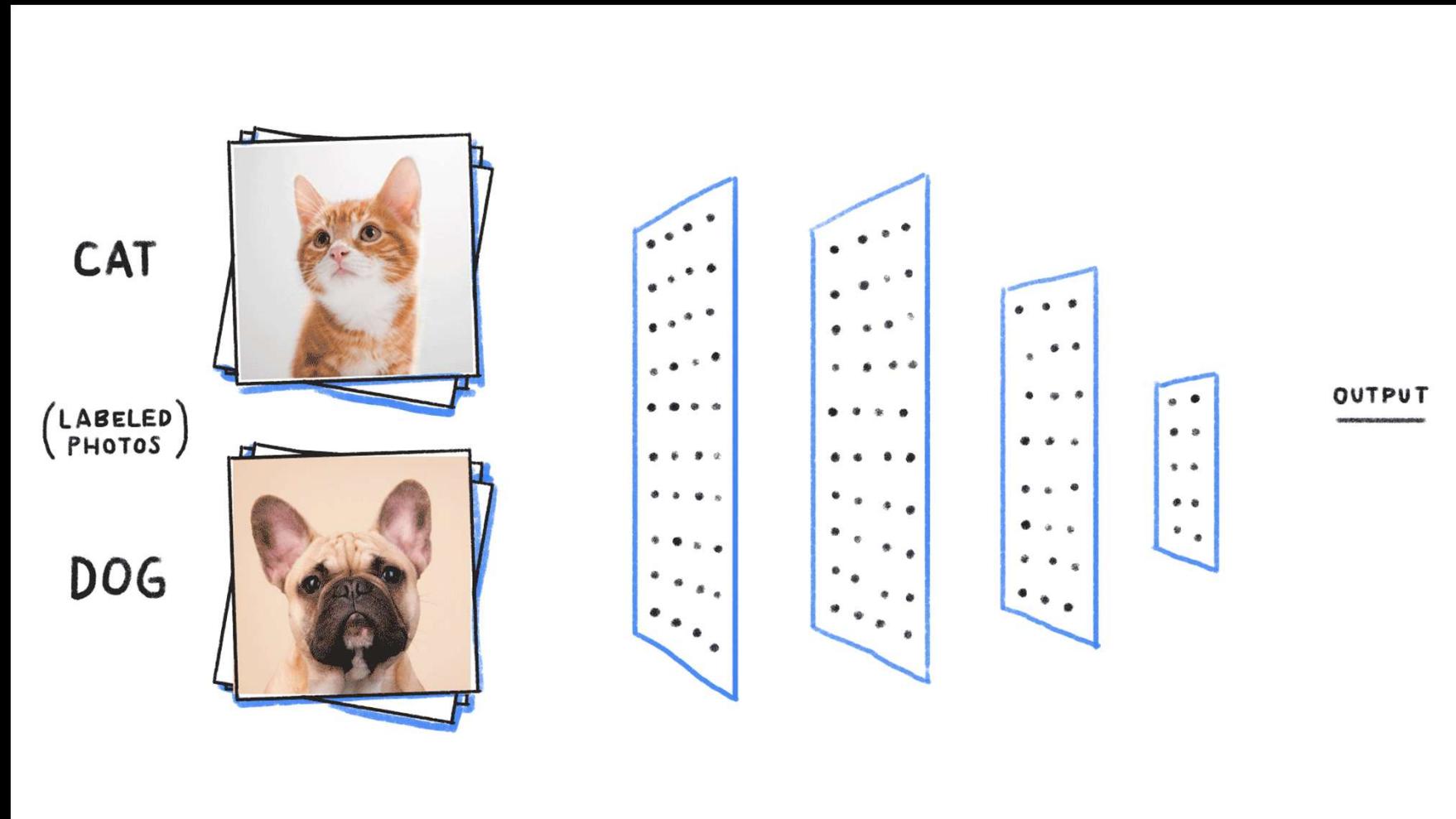


# Demo

[Machine Learning Playground \(ml-playground.com\)](https://ml-playground.com)

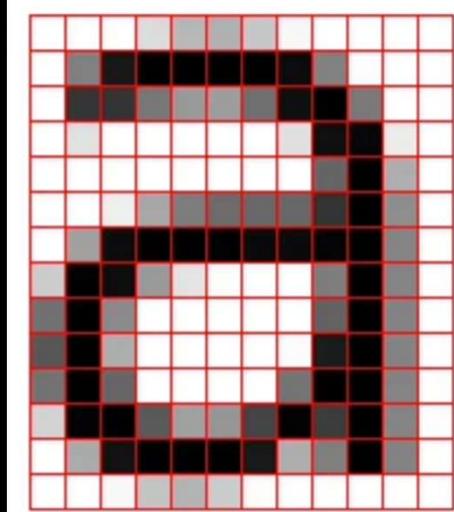
<https://playground.tensorflow.org/>

# Convolutional Neural Network (CNN)



# The first step – breaking down image, text into machine understandable format

Image is represented as matrix of numbers

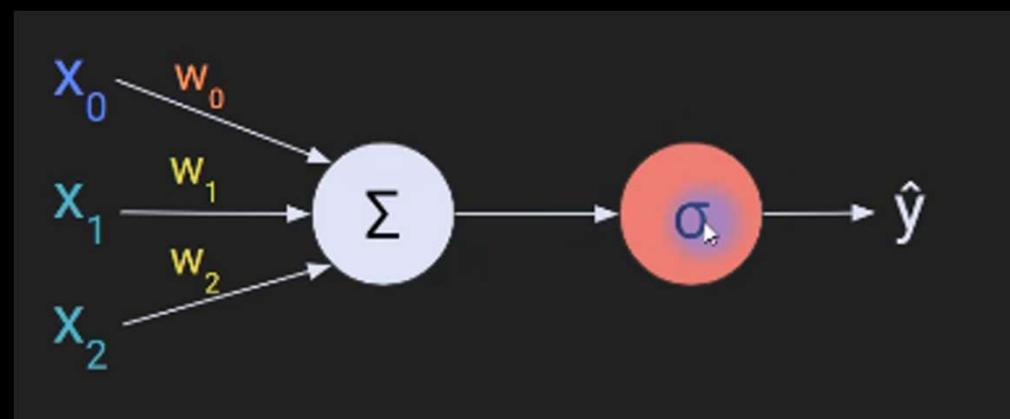
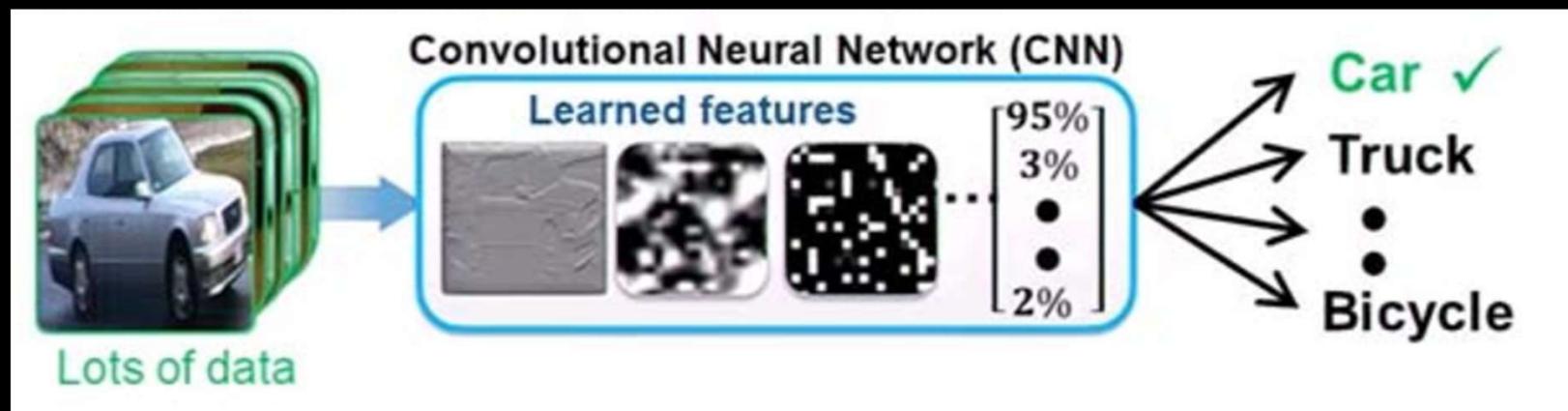


1.0	1.0	1.0	0.9	0.6	0.6	0.6	1.0	1.0	1.0	1.0
1.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.5	1.0	1.0
1.0	0.2	0.2	0.5	0.6	0.6	0.5	0.0	0.0	0.5	1.0
1.0	0.9	1.0	1.0	1.0	1.0	1.0	0.9	0.0	0.0	0.9
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.5	1.0	1.0
1.0	1.0	1.0	0.5	0.5	0.5	0.5	0.4	0.0	0.5	1.0
1.0	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	1.0
0.9	0.0	0.0	0.6	1.0	1.0	1.0	0.5	0.0	0.5	1.0
0.5	0.0	0.6	1.0	1.0	1.0	1.0	1.0	0.5	0.0	0.5
0.5	0.0	0.7	1.0	1.0	1.0	1.0	1.0	0.0	0.5	1.0
0.6	0.0	0.6	1.0	1.0	1.0	1.0	0.5	0.0	0.0	0.5
0.9	0.1	0.0	0.6	0.7	0.7	0.5	0.0	0.5	0.0	0.5
1.0	0.7	0.1	0.0	0.0	0.0	0.1	0.9	0.8	0.0	0.5
1.0	1.0	1.0	0.8	0.8	0.9	1.0	1.0	1.0	1.0	1.0

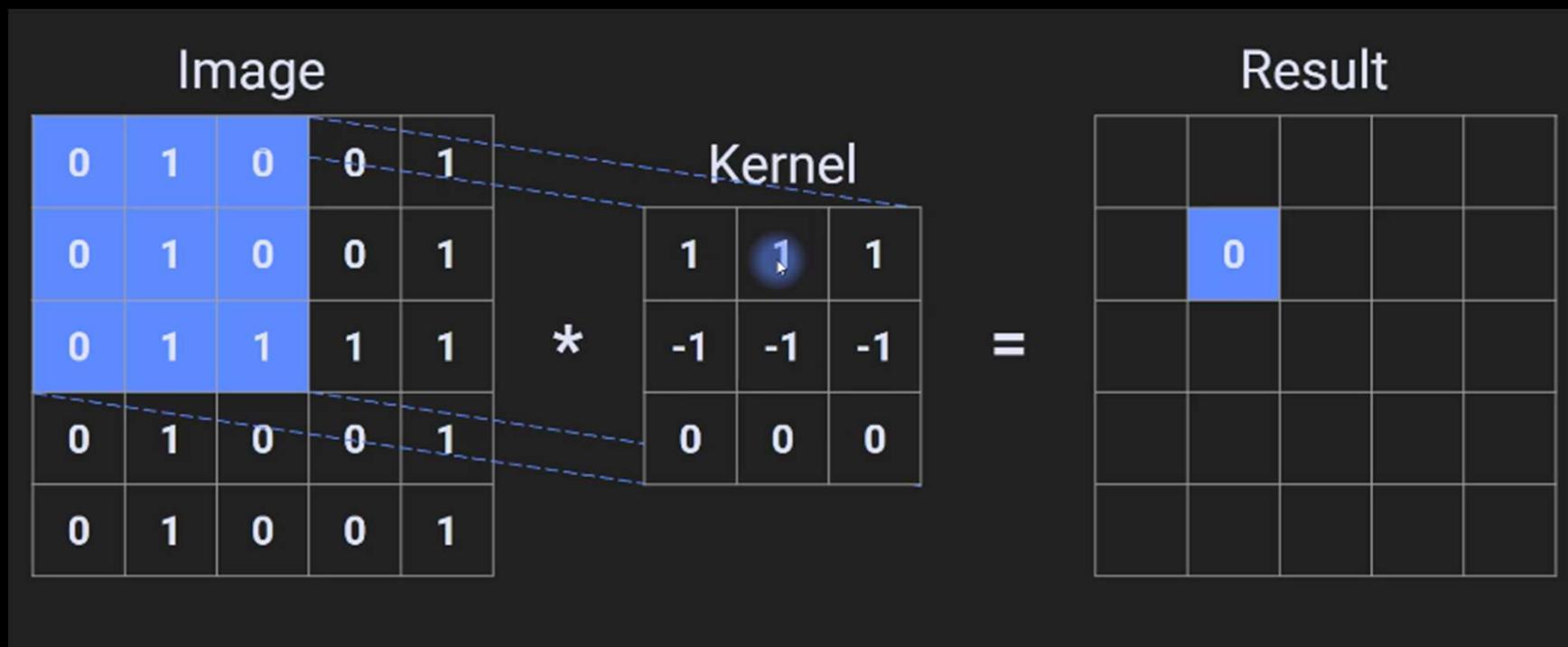
Text is represented as sequence of numbers, called vectors

word	vector
The	(0.12, 0.23, 0.56)
Cardinals	(0.24, 0.65, 0.72)
will	(0.38, 0.42, 0.12)
win	(0.57, 0.01, 0.02)
the	(0.53, 0.68, 0.91)
world	(0.11, 0.27, 0.45)
series	(0.01, 0.05, 0.62)

The second step – train the neural network with lots of data

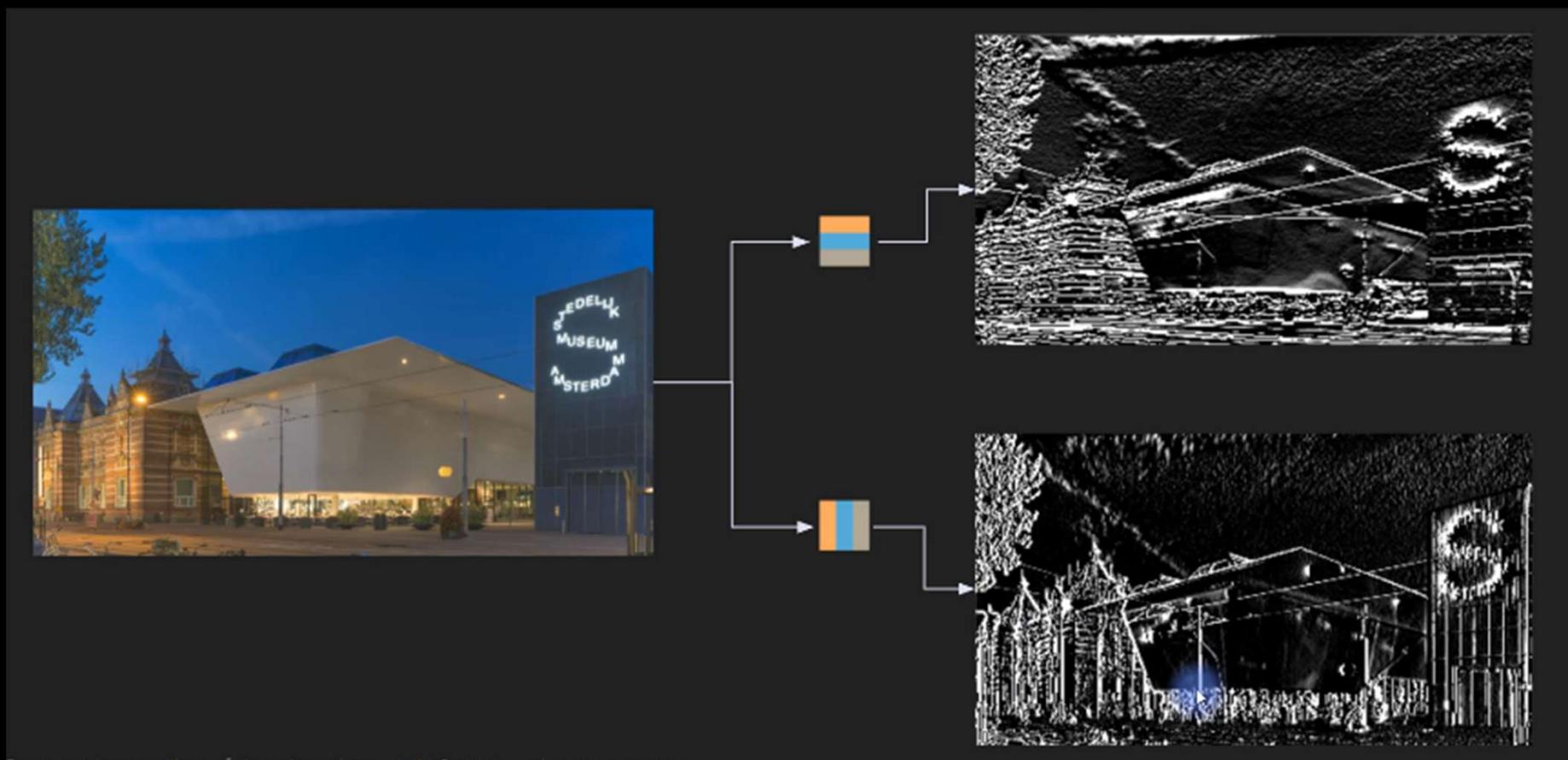


# Convolution



Identifies the features in the data, for example, in an image it identifies the edges, textures, patterns

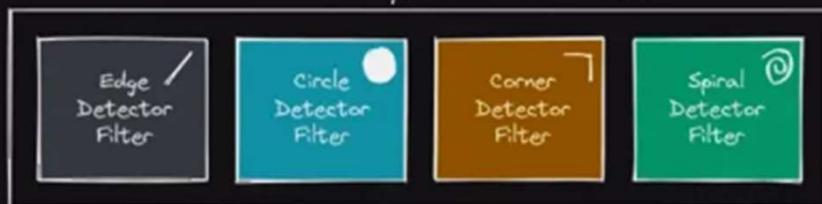
# Convolution



Horizontal and vertical kernels applied to an image – output is a feature map

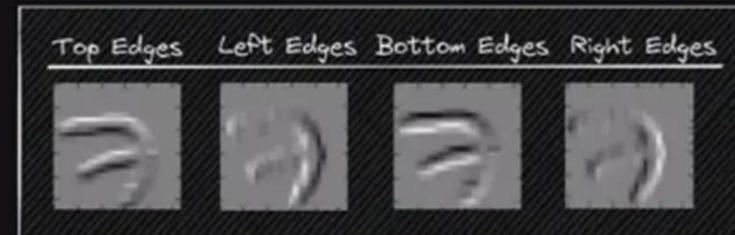
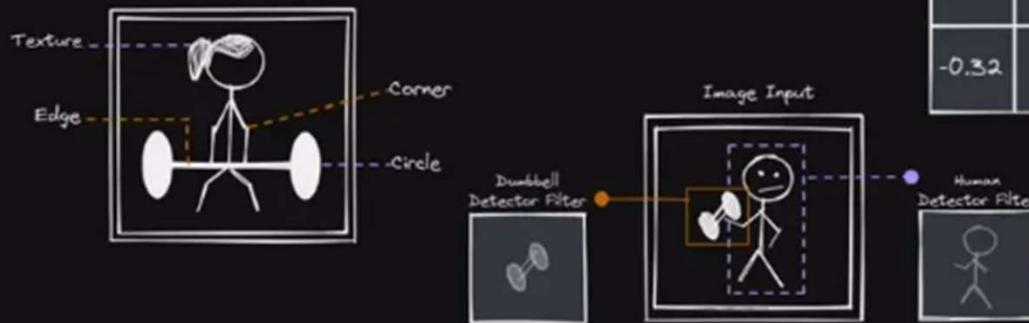
# CONVOLUTIONAL LAYERS & FILTERS

Convolutional Layer with Four Filters

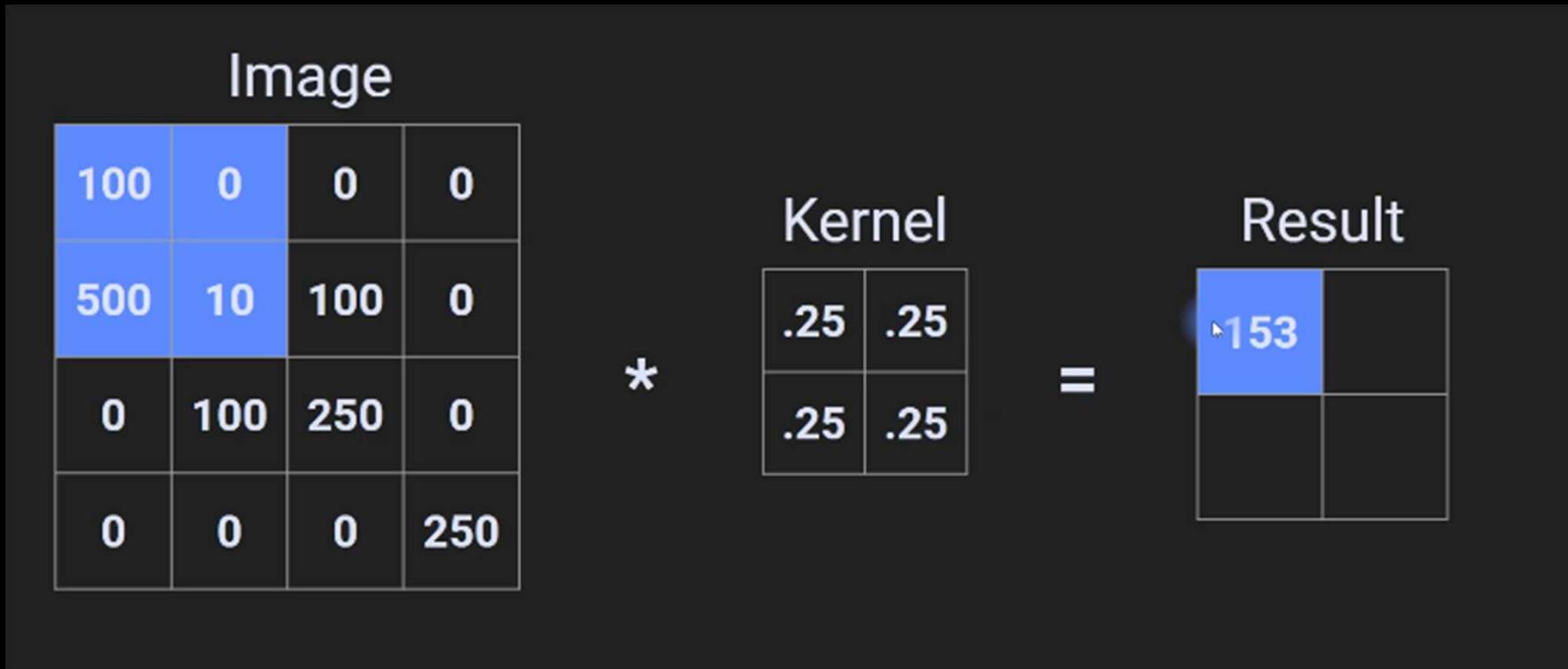


3 x 3 Convolutional Filter

0.1	-0.92	0.5
1.4	0.39	-1.1
-0.32	-1.2	0.84



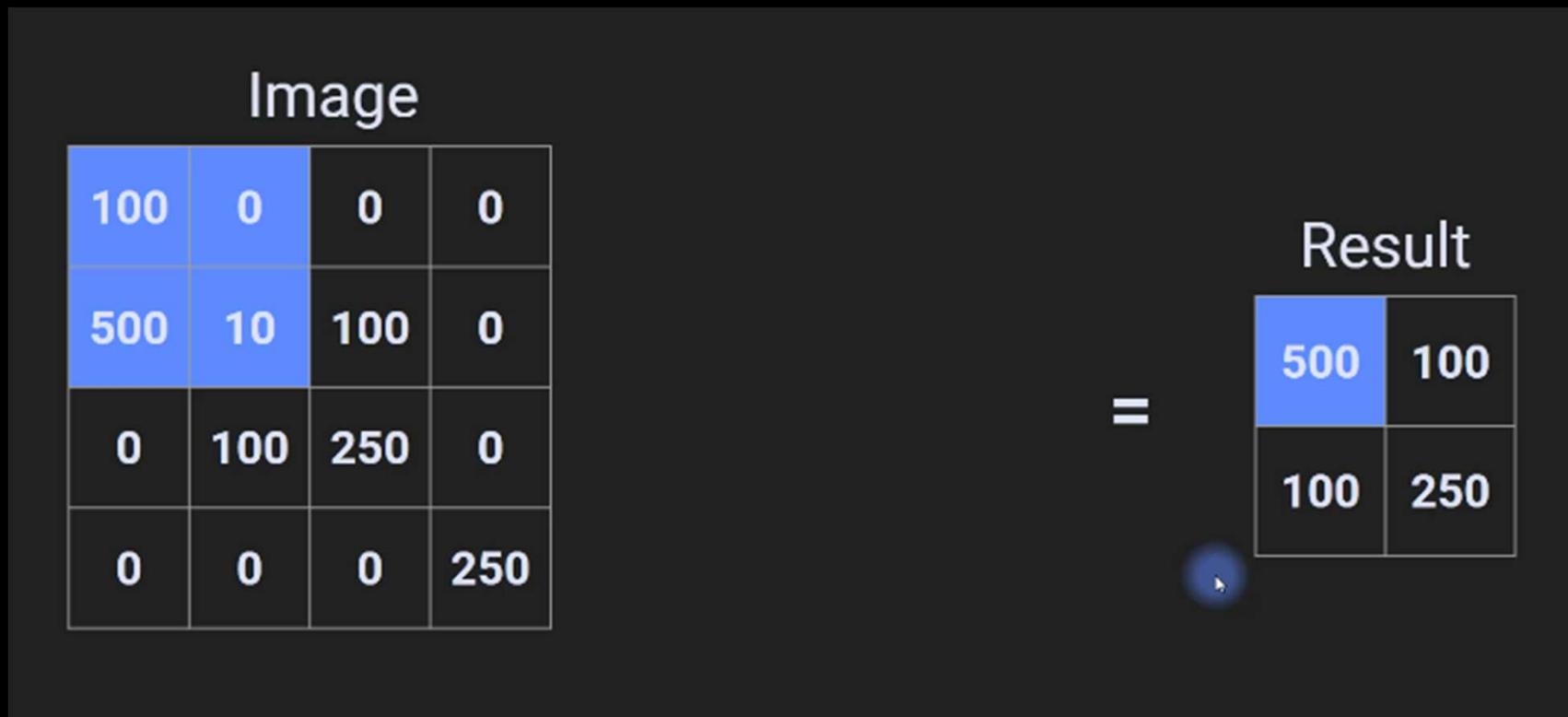
# Pooling



Purpose is to reduce dimensionality

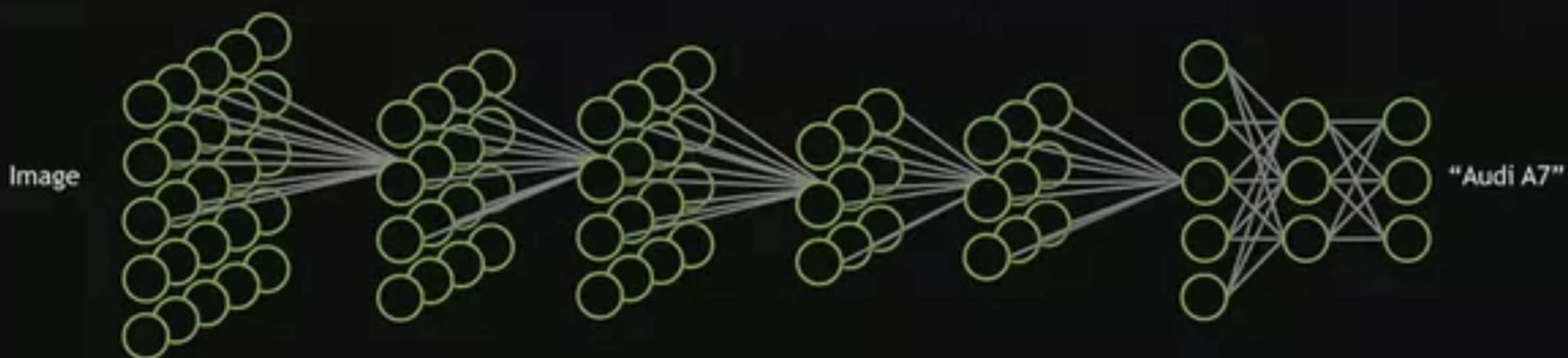
Above is doing pooling with a kernel 2x2 with a stride of 2

# Pooling

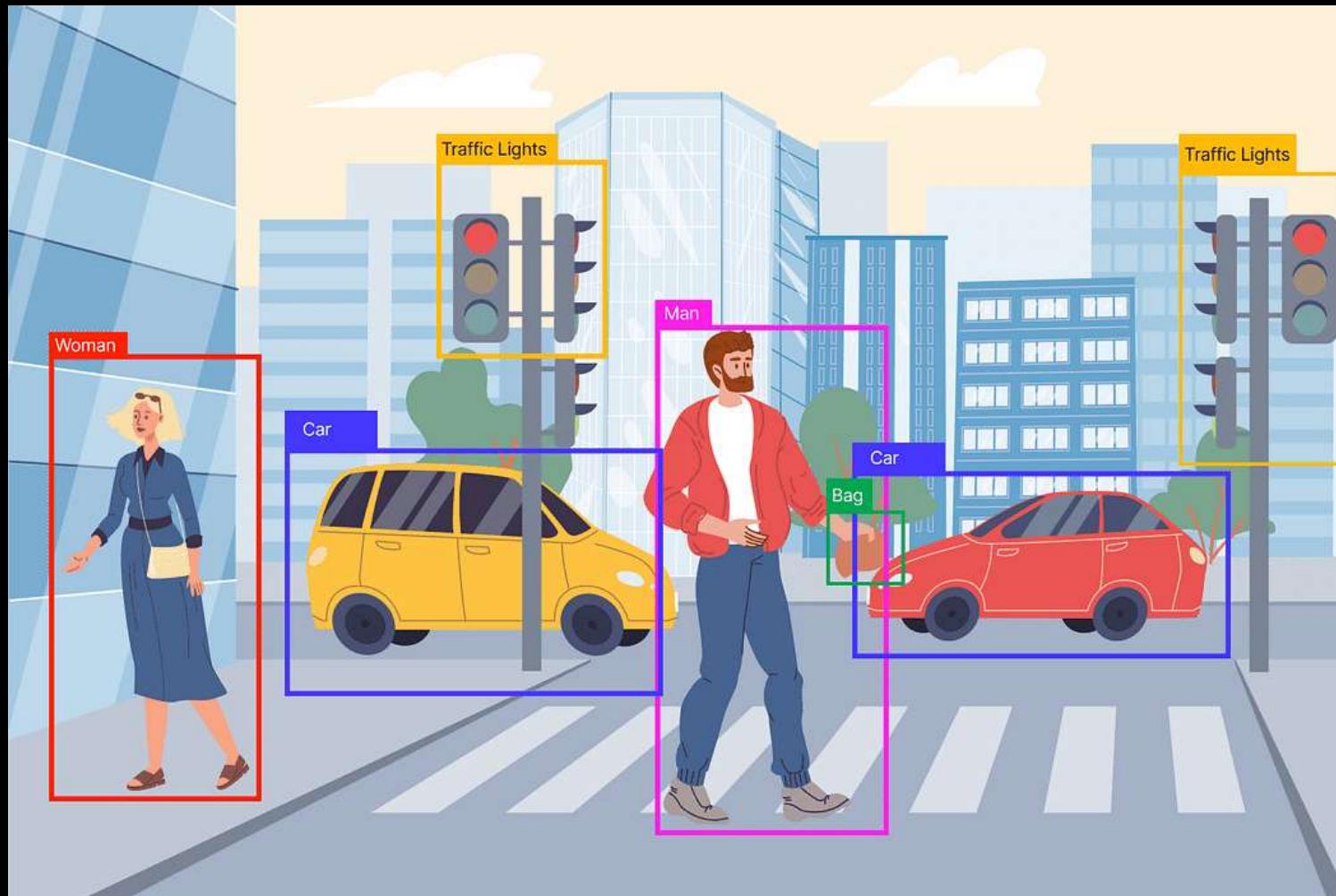


Max pooling

# Pooling



# Object Detection



# CNN advancements for object detection

Can CNN natively do it?

Advancements to CNN to make object detection possible"

## **Faster R-CNN (Region Proposal Network RPN)**

Two-Stage Detection: Faster R-CNN works in two stages:

1. A Region Proposal Network (RPN) suggests possible object locations.
2. The classification head refines these proposals and assigns class labels.

## **SSD (Single Shot MultiBox Detector)**

SSD predicts objects from multiple feature maps, each tuned to detect different object sizes. Early layers find small objects, and deeper layers detect larger ones

## **YOLO (You Look Only Once)**

YOLO processes the image in a single forward pass, dividing the image into a grid and predicting bounding boxes and class probabilities simultaneously.

# How does YOLO work?

## **Input Image:**

The input image is resized to a fixed size, usually 416x416 or 640x640.

## **Grid Division:**

YOLO divides the image into an  $S \times S$  grid (e.g., 13x13 for YOLOv3). Each grid cell is responsible for detecting objects whose center falls within the cell.

## **Bounding Box Prediction:**

Each grid cell predicts  $B$  bounding boxes (e.g., 3 per cell in YOLOv3). For each bounding box, YOLO predicts center coordinates relative to the grid cell ( $x, y$ ); Width and height relative to the image size ( $w, h$ ), confidence score. Additionally, it predicts class probabilities for each cell.

## **Output:**

The final output is a list of detected objects with their class labels, confidence scores, and bounding boxes.

# How does training data for YOLO look like?

## 1 Image

The image itself (e.g., `image1.jpg`).

## 2 YOLO Annotation File (`image1.txt`)

For each object in the image, the annotation file includes:

`php-template`

```
<class_id> <x_center> <y_center> <width> <height>
```

Where:

- `class_id`: Index of the object class (e.g., 0 for "car", 1 for "person").
- `x_center`: Horizontal center of the bounding box (normalized: 0 to 1).
- `y_center`: Vertical center of the bounding box (normalized: 0 to 1).
- `width`: Width of the bounding box (normalized by image width).
- `height`: Height of the bounding box (normalized by image height).

## Yolo versions

YOLOv1 (2016): Created by **Joseph Redmon**. First real-time object detection model.

YOLOv2 and YOLOv3 (2017–2018): Also developed by Redmon with improvements in accuracy and multi-scale detection.

YOLOv4 (2020): Released by Alexey Bochkovskiy after Redmon stopped working on computer vision due to ethical concerns.

YOLOv5 (2020): Released by Ultralytics, though not officially affiliated with the original creators.

YOLOv6, YOLOv7 (2022): Developed by different research groups, focusing on speed and efficiency.

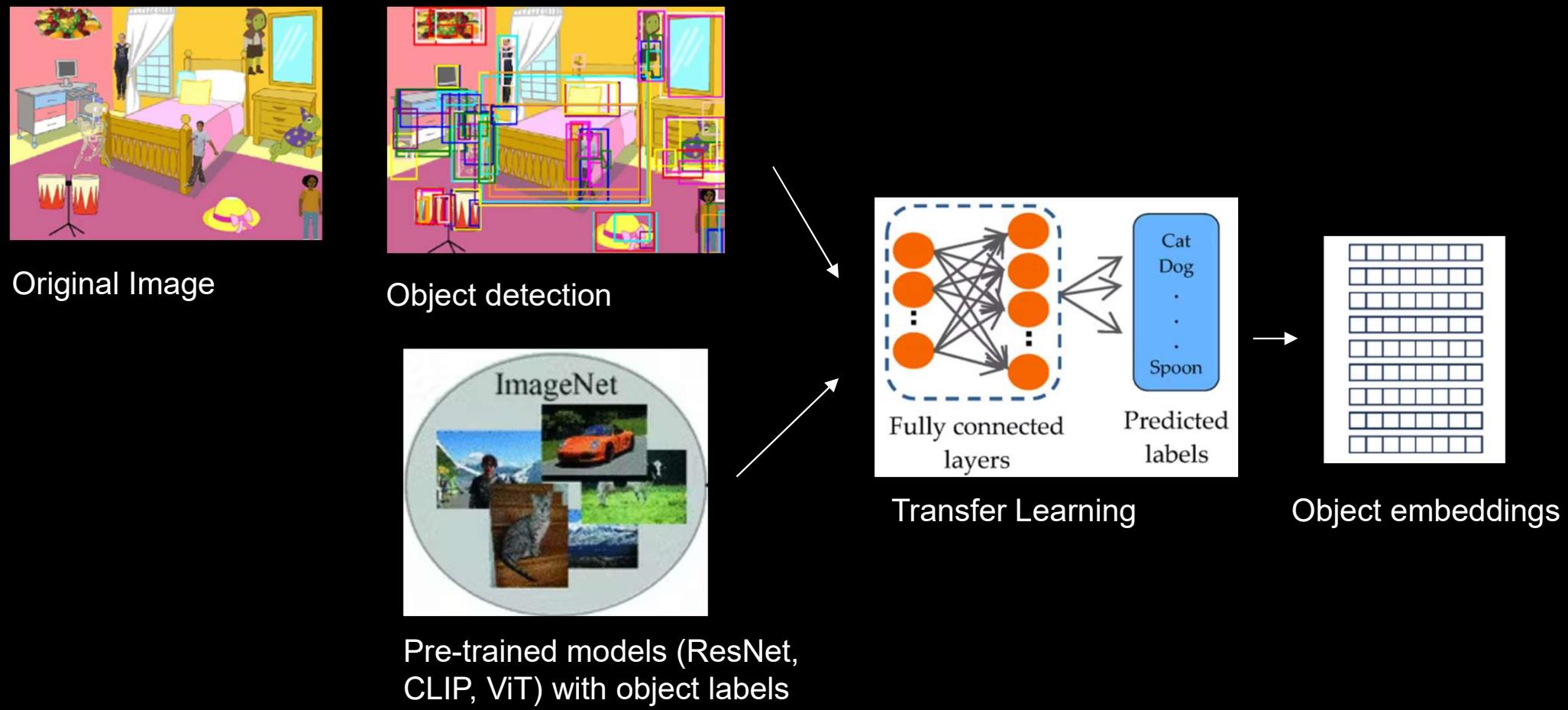
YOLOv8 (2023): Released by Ultralytics with further performance improvements.

# How to achieve fine grained detection?

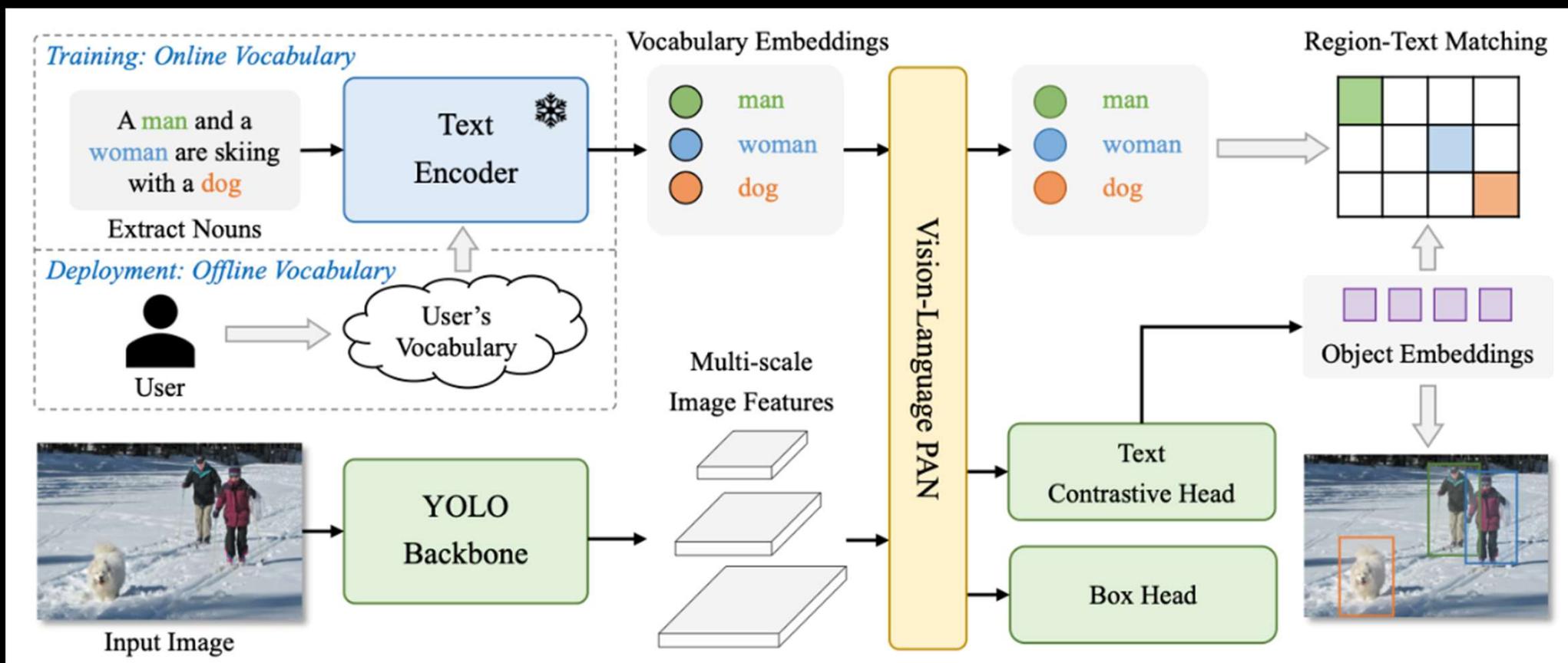
Detecting hair line fracture? Detecting tumour in x-ray/ct-scan/mri scan images

- Adapt or extend YOLO  
increase input resolution, use custom anchors, domain specific pre-processing (equalization, windowing, de-noising)  
targeted augmentations such as small rotations
- Use Faster R-CNN
- Hyper parameter tuning, transfer learning, domain specific pre-trained models
- Advanced models such as FPN (Feature Pyramid Network) or PAN (Path Aggregation Network) can help detect objects at multiple scales.

# Labeling the detected object – transfer learning



# Labeling the detected object – CLIP



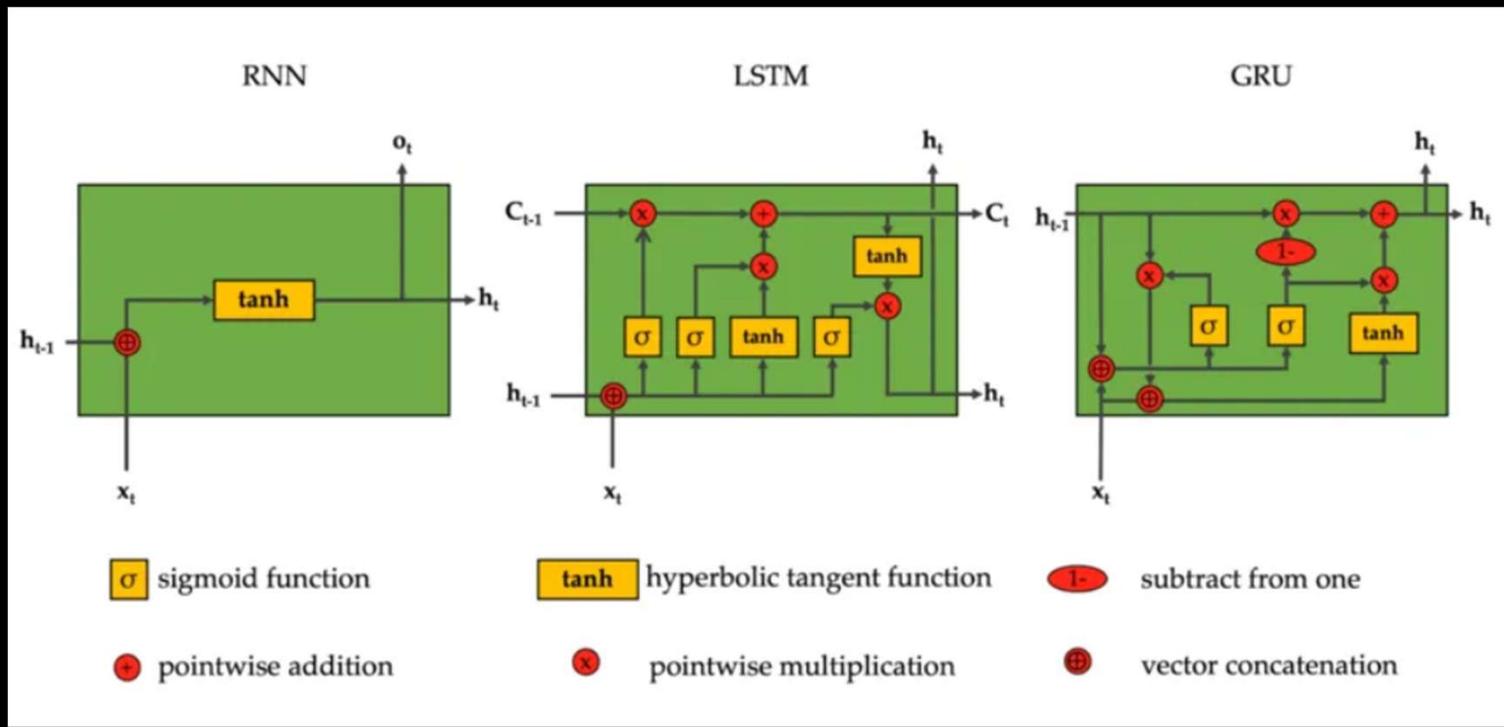
# Object detection demos

<https://www.kaggle.com/code/nirvanafl/comvisionproj-yolo8/notebook>

<https://huggingface.co/spaces/stevengrove/YOLO-World>

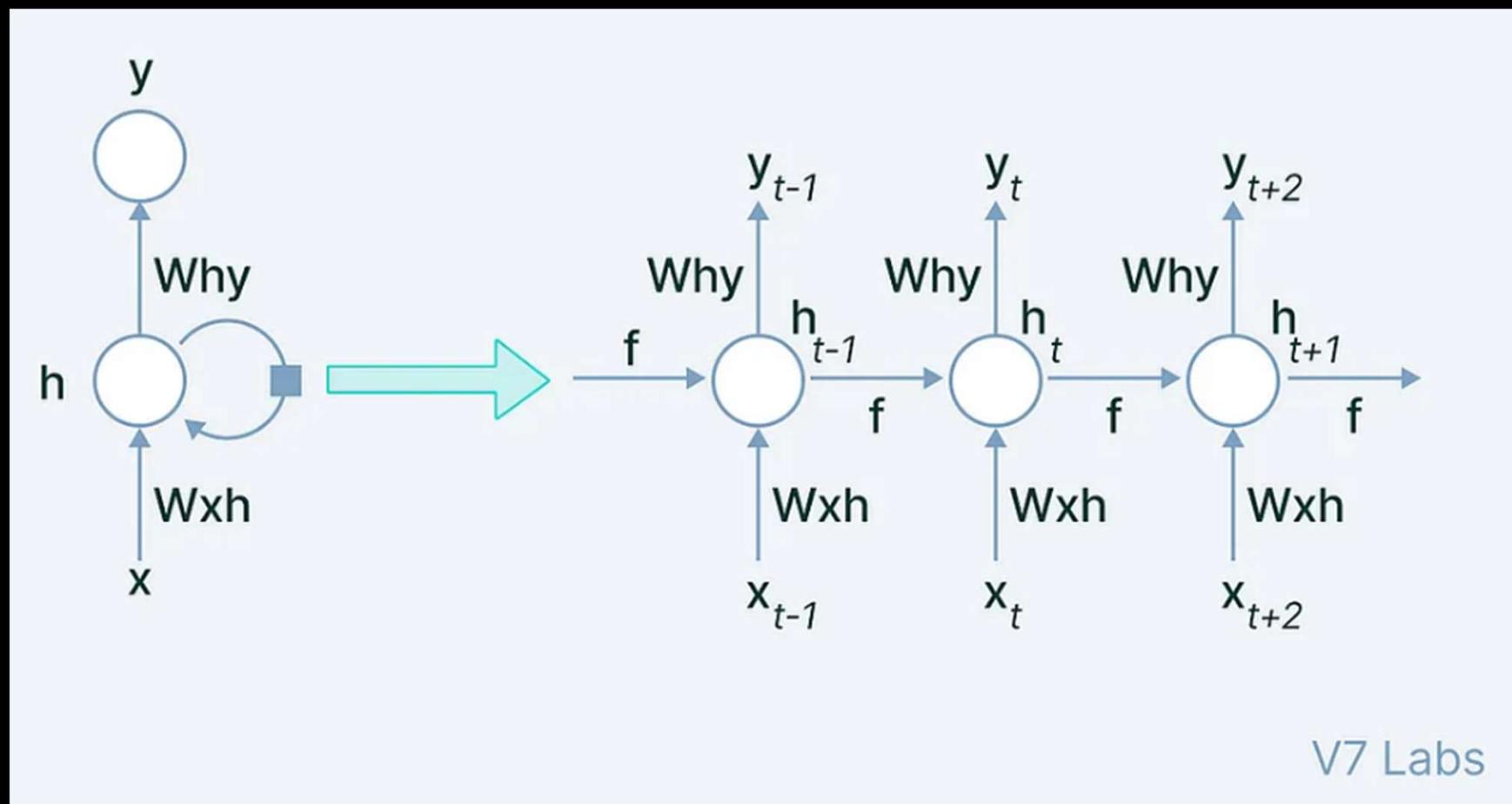
<https://huggingface.co/spaces/stevengrove/YOLO-World/blob/main/README.md>

## RNN, LSTM and GRU



- Concept of limited memory (state) enabling sequential data processing
- Examples: Timeseries analysis, text processing, Audio processing
- Enabled NLP (Natural Language Processing) use-cases

# Recurrent Neural Networks (RNN)



V7 Labs

## RNN- Recurrent connection

**Recurrent Connection:** This is a crucial feature of RNNs. It is a set of weights and connections that loop back to the hidden layer from itself at the previous time step. This loop allows the RNN to maintain and update its hidden state as it processes each element in the sequence. The recurrent connection enables the network to remember and utilize information from the past.

# RNN Applications

- Healthcare:  
Sequential analysis of disease progression, key metrics deterioration
- Stock Market  
Analyse stock data over time
- Text
- Audio

## Challenges with RNN – vanishing gradient

The vanishing gradient problem occurs when training deep Recurrent Neural Networks (RNNs)

Gradients **shrink exponentially** as they are propagated back through time.

This makes it difficult for the model to learn long-term dependencies because earlier layers receive negligible updates during backpropagation.

## Vanishing gradient example

Let's say we are building an RNN model to predict the likelihood of a diabetic patient developing complications (e.g., kidney failure) based on their historical medical records.

We have sequential health data for patients collected over 10 years, including:

- Blood sugar levels
- Blood pressure readings
- Medication adherence
- Lab test results
- Lifestyle habits (exercise, diet)

The goal is to predict whether the patient will develop kidney failure in the next 2 years.

## Vanishing gradient example contd.

Suppose an early symptom (e.g., consistently high blood sugar levels 5 years ago) is a key predictor of kidney failure.

The RNN needs to remember this early symptom while processing later medical records.

However, during training:

The error signal propagates back from the final prediction (kidney failure or not) to the earlier time steps.

If the model is deep, the gradients from the final output shrink exponentially before reaching the early time steps.

The model fails to learn the relationship between early high blood sugar levels and kidney failure.

As a result, the RNN relies only on recent symptoms, ignoring long-term trends, leading to poor accuracy.

# Exploding gradient

The exploding gradient problem happens when the gradients in a Recurrent Neural Network (RNN) grow exponentially during backpropagation. This results in unstable weight updates, making the model unable to learn effectively.

Why does this happen?

In an RNN, gradients are passed backward through time.  
If the weight updates keep multiplying by large values, they grow exponentially.  
This leads to very large updates, making the model's parameters explode to extremely high values.

When this happens, the model's loss function fluctuates wildly and fails to converge.

# Exploding gradient – real world example

Imagine we are building an RNN model to predict the likelihood of sepsis (a severe infection) in ICU patients based on their hourly vital signs:

Dataset:

Each patient has hourly health records for 48 hours, including:

- Heart Rate
- Blood Pressure
- Oxygen Levels
- Temperature
- White Blood Cell Count

Problem:

The RNN must analyze hourly data sequences and predict whether the patient will develop sepsis in the next 6 hours.

If we use ReLU as the activation function, the model starts learning but quickly encounters exploding gradients.

## Exploding gradient – real world example contd.

Exploding Gradient Happens in This Case:

Early time steps (hour 1-10): Model learns normal variations in patient vitals.

Middle time steps (hour 20-30): Slightly larger weight updates occur as the model detects risk patterns.

Later time steps (hour 40-48): The RNN keeps multiplying gradients backward in time.

Backpropagation amplifies the problem:

Since ReLU outputs unbounded values, each time step multiplies the previous gradients, making them exponentially large. Loss function diverges:

The model fails to learn meaningful trends and starts producing NaN (Not-a-Number) losses

# Choosing the right activation function

Activation	Where to Use	Pros	Cons
ReLU	RNNs (Shallow or Deep)	Faster training, helps mitigate vanishing gradients	Exploding gradients possible, no negative values
tanh	RNNs (Classic) & LSTMs	Keeps values balanced, better for sequential data	Slower training, vanishing gradients in deep networks
sigmoid	Rarely Used	Useful for probability-based tasks	Strong vanishing gradient issue

# Hidden size optimization

The number of hidden units (neurons in the hidden layer) is a hyperparameter that directly affects the model's memory capacity, performance, and computational cost.

While there is no theoretical limit, there are practical constraints based on:

- Computational resources (GPU/CPU/RAM)
- Model performance (overfitting vs underfitting)
- Training time

Model	Max Practical Hidden Units
Vanilla RNN	50-200 (Exploding gradients issue)
LSTM / GRU	128-512 (More stable)
Deep LSTM (Stacked)	256-1024 (With strong GPUs)

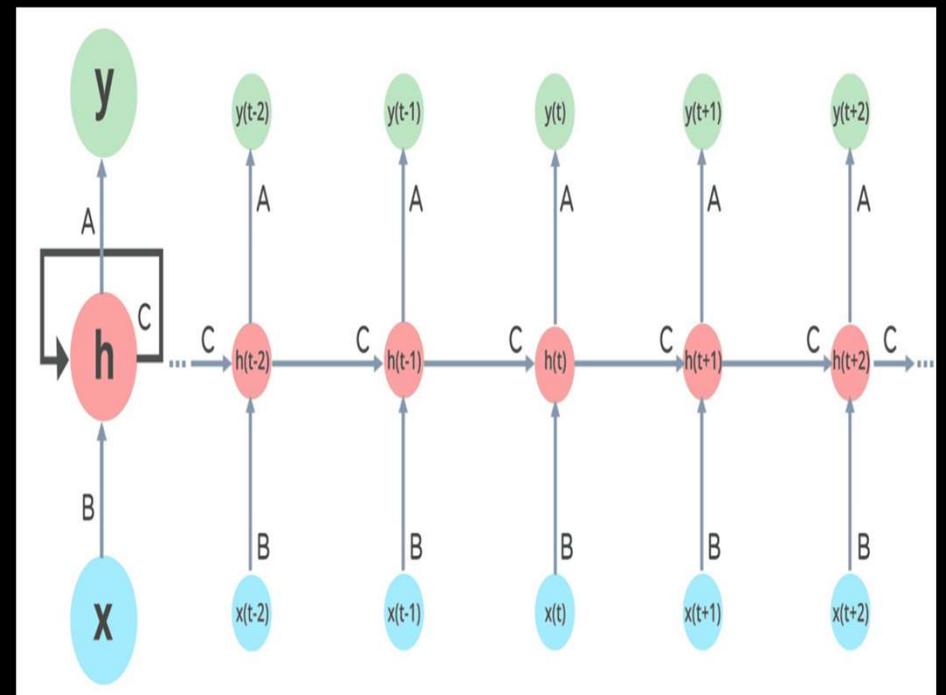
 **LSTMs and GRUs allow for higher hidden sizes than Vanilla RNNs because they prevent vanishing gradients.**

# Recurrent Neural Networks (RNN)

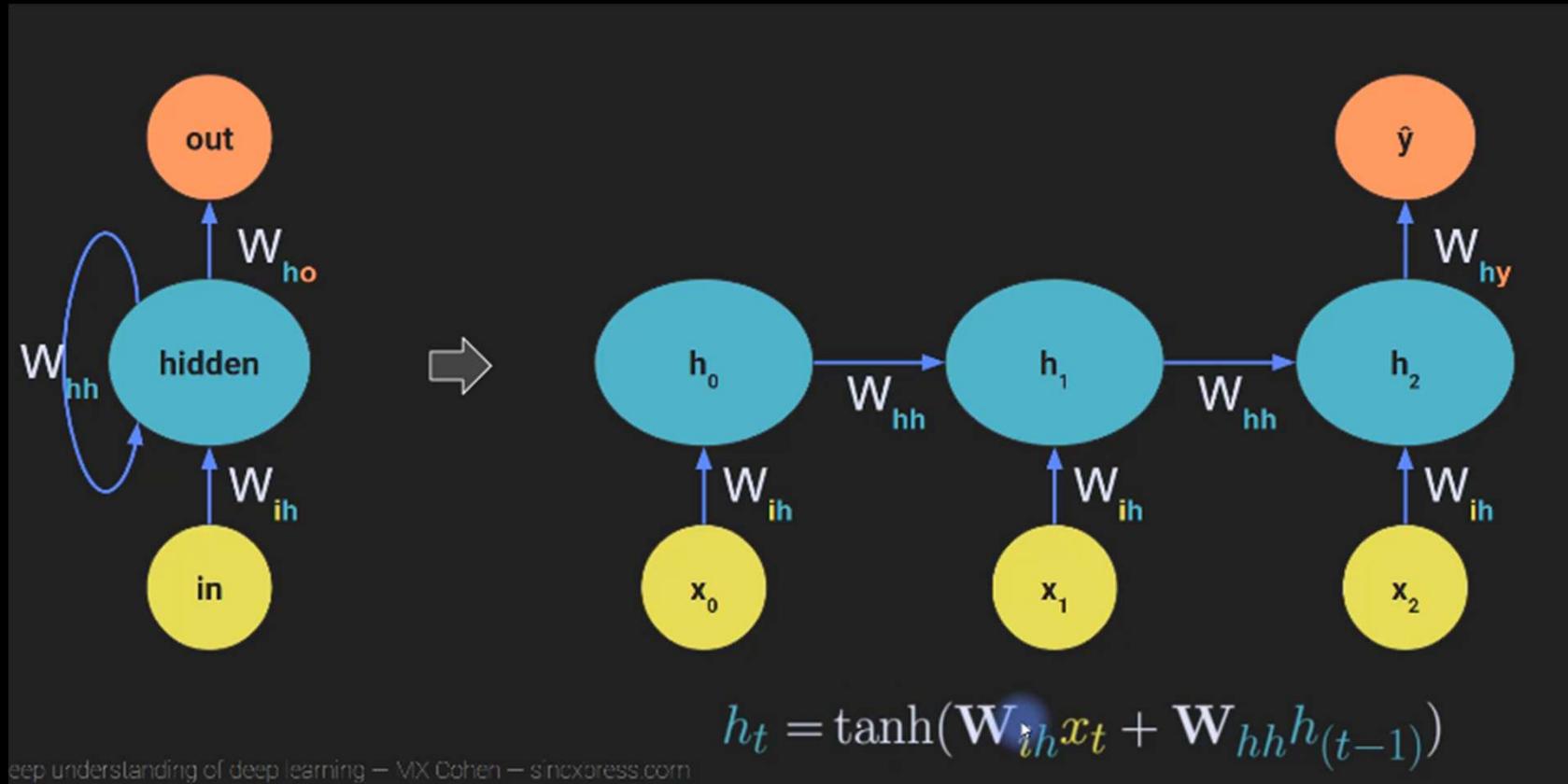
Recurrent neural networks (RNNs) are a type of neural network that is well-suited for processing sequential data.

This is because RNNs have an internal state that allows them to remember information from previous inputs.

This makes them ideal for tasks such as **natural language processing (NLP)**, where the meaning of a word or phrase can depend on its context



# Recurrent Neural Networks (RNN)

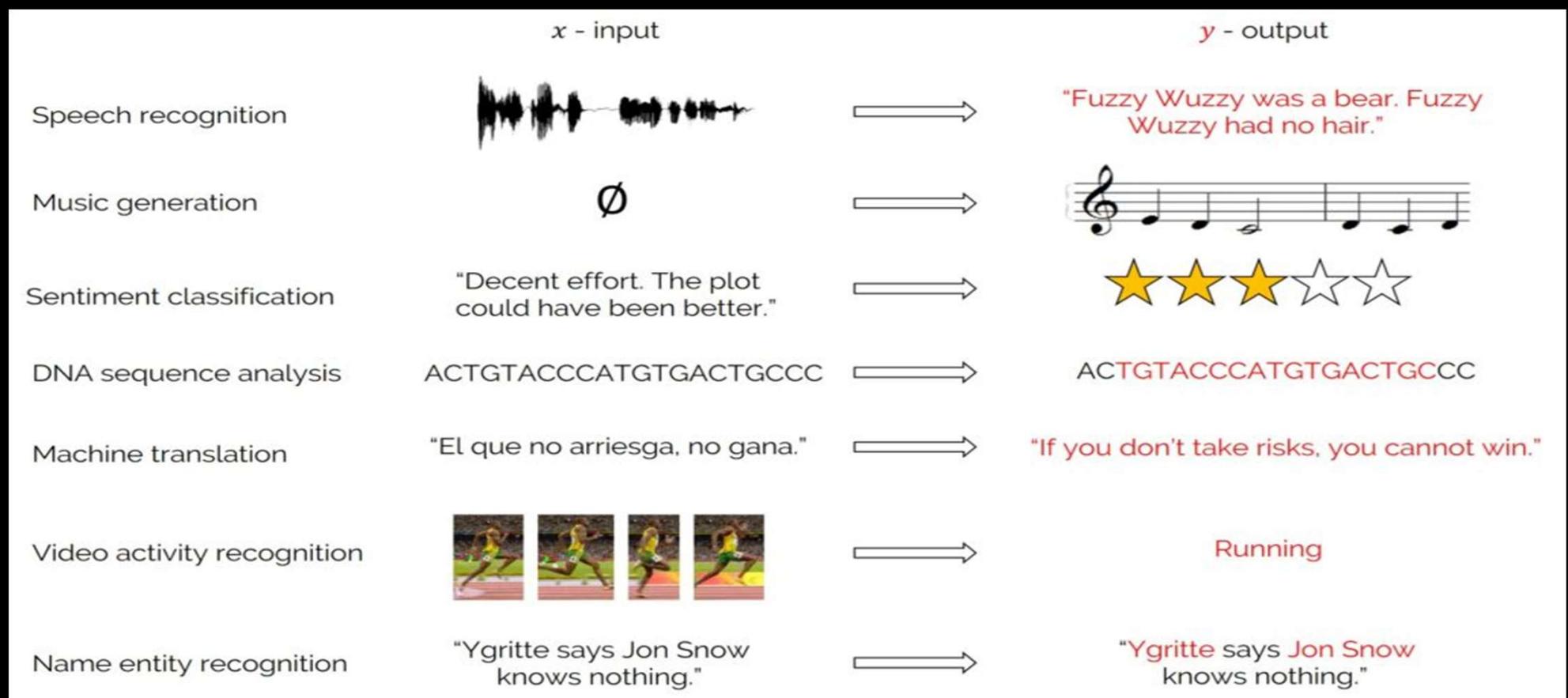


$h_0$  = hidden state at time t0

$h_1$  = hidden state at time t1

It is cumulative – combination of all previous hidden states is reflected in the next layer

# Applications of RNN

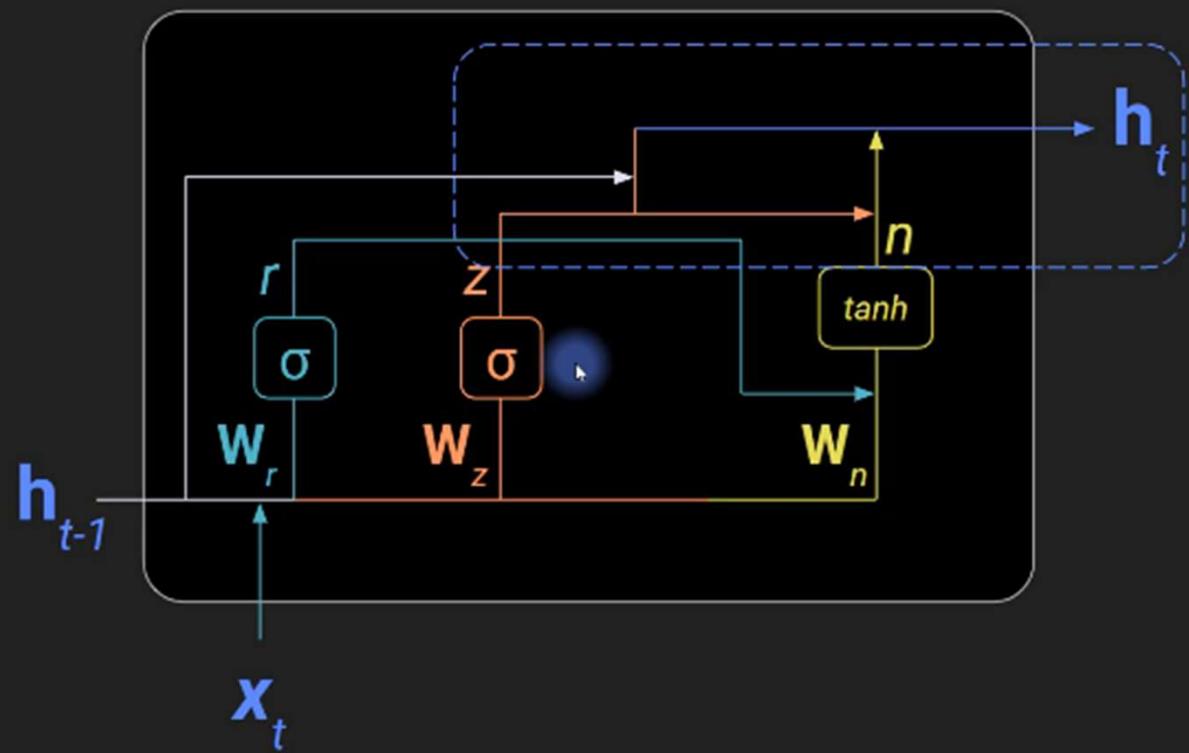


# Gated Recurrent Unit (GRU)

$$h_t = (1 - z_t)n_t + z_t h_{t-1}$$

**Output (hidden state)**

Weighted combination of  
to-remember new state ( $n$ )  
and to-forget old state ( $h_{t-1}$ ).



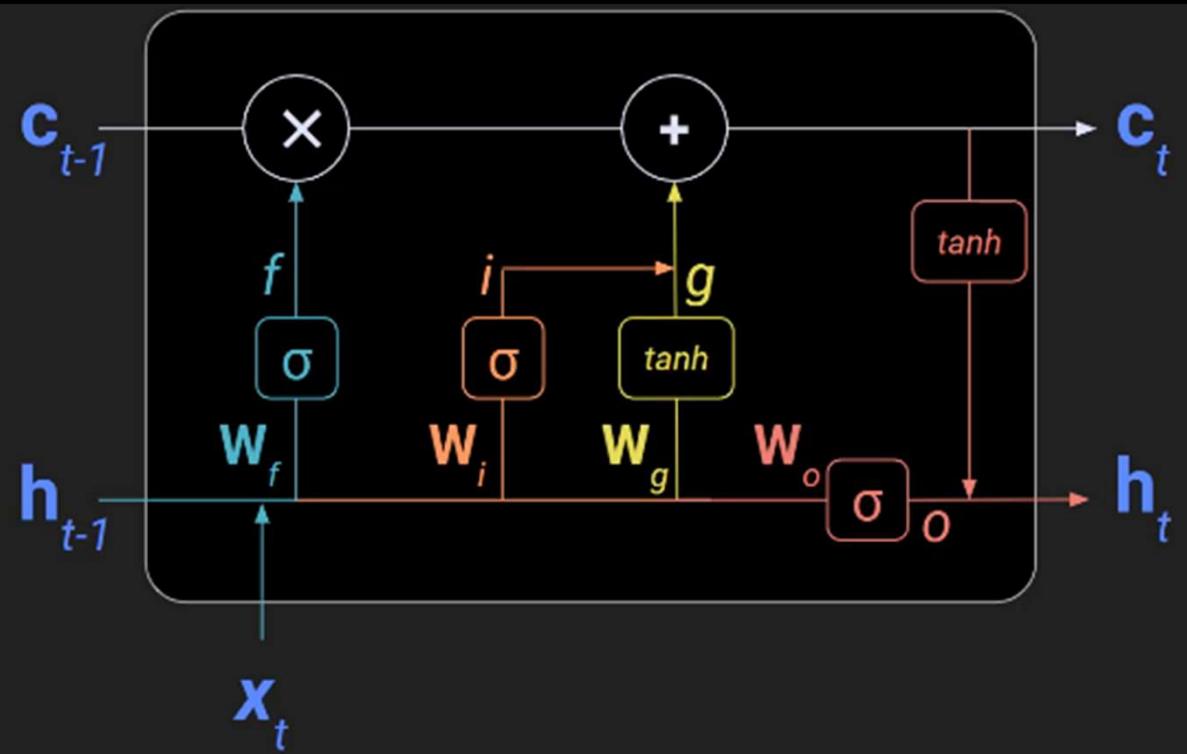
# Long Short Term Memory (LSTM)

Forget the past

Remember the present

Plan for the future

Recorded history



H = hidden state from previous time point

C = cell: preserves long term memory traits via c

## When to use what?

**Standard RNN:** Simple, fewest parameters. Good for sequences with short patterns.

**GRU:** More complex than standard RNNs, but simpler than LSTMs. Trains fast and highly scalable, but needs more data than RNNs.

**LSTM:** Most complex, requires more training data and computation time. Explicitly includes a memory trace. Good for longer sequences or complex patterns.

# Transfer Learning

## What is Transfer Learning?

Transfer learning is a machine learning technique where a **model developed for one task** is **reused** (either as-is or with some fine-tuning) for a **different** but related task

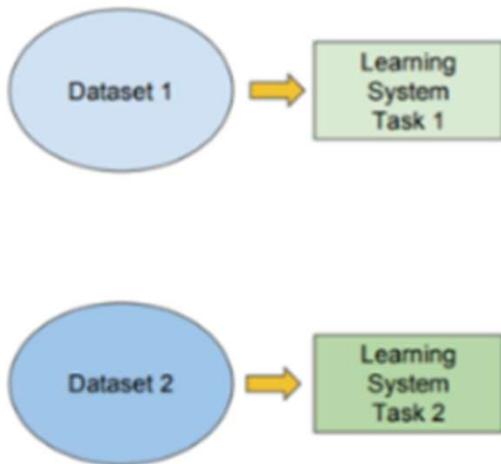
# Transfer Learning

## Traditional ML

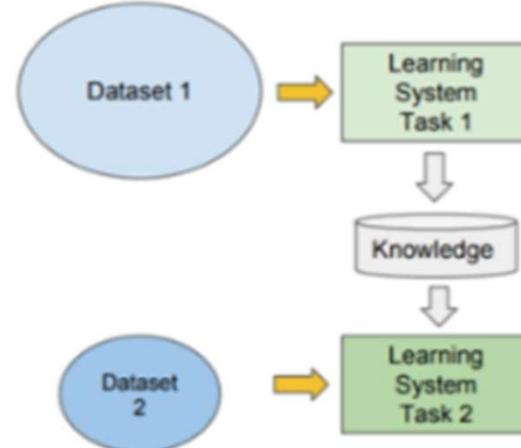
vs

## Transfer Learning

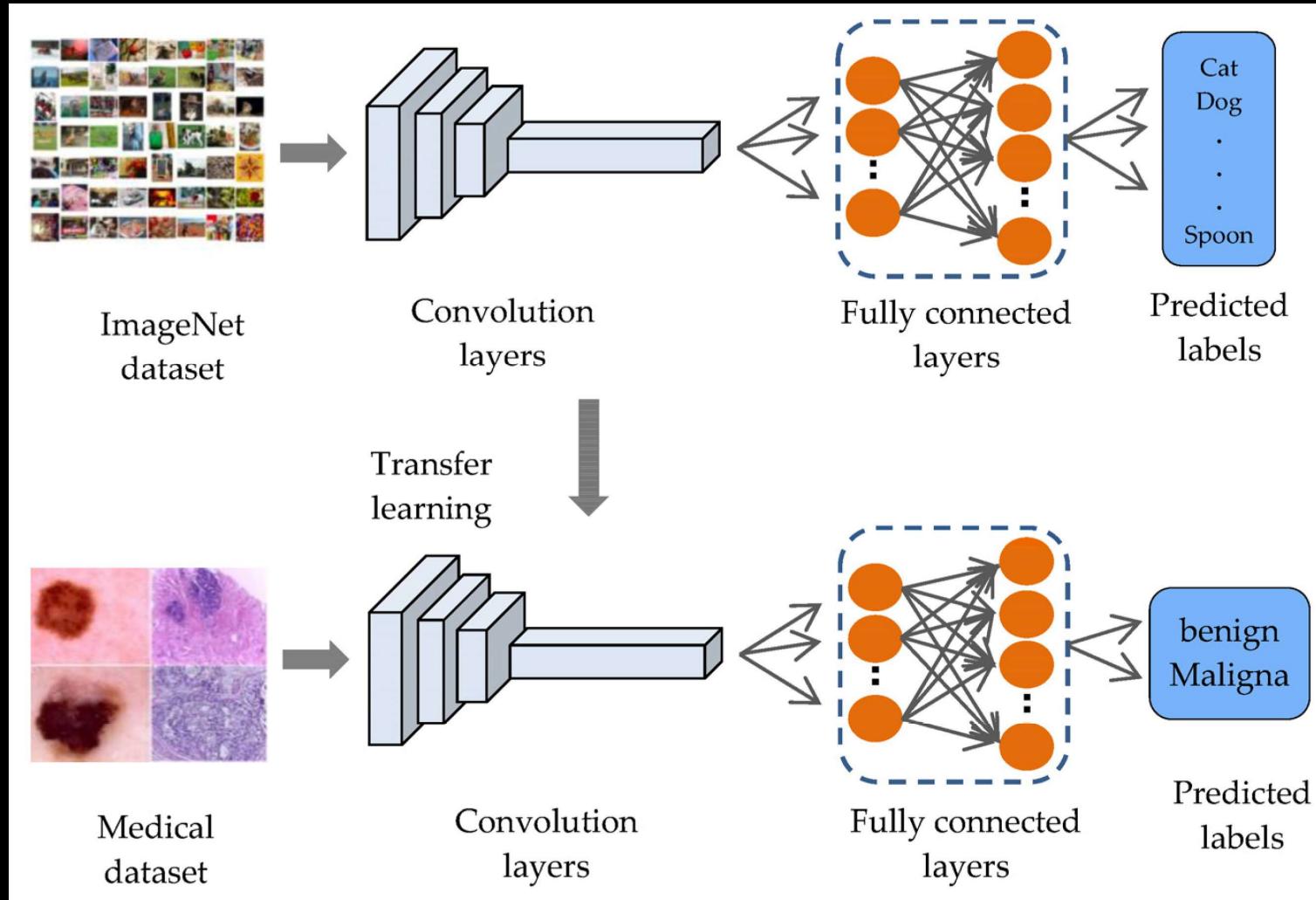
- Isolated, single task learning:
  - Knowledge is not retained or accumulated. Learning is performed w.o. considering past learned knowledge in other tasks



- Learning of a new tasks relies on the previous learned tasks:
  - Learning process can be faster, more accurate and/or need less training data

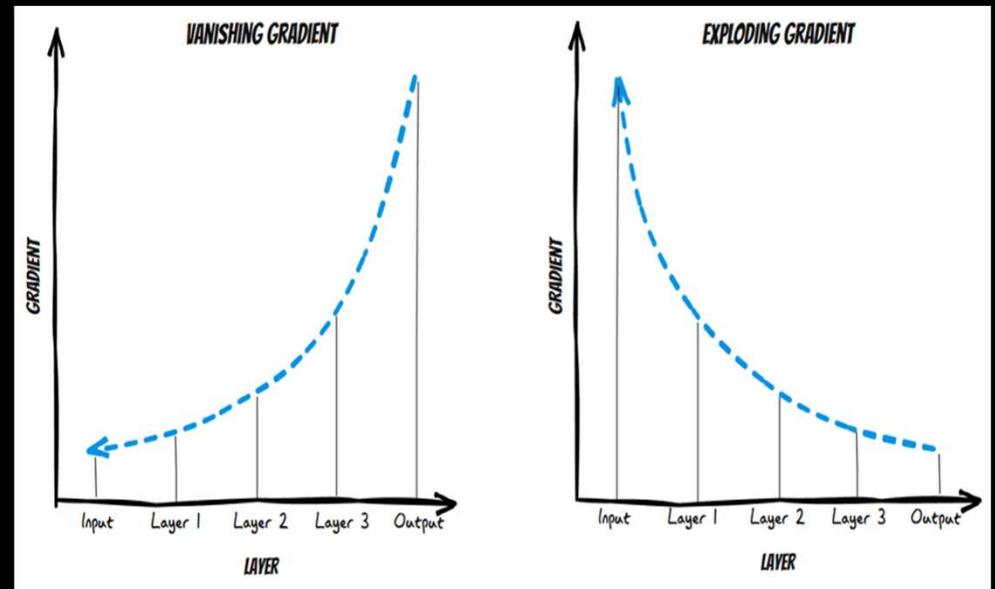


# Transfer Learning



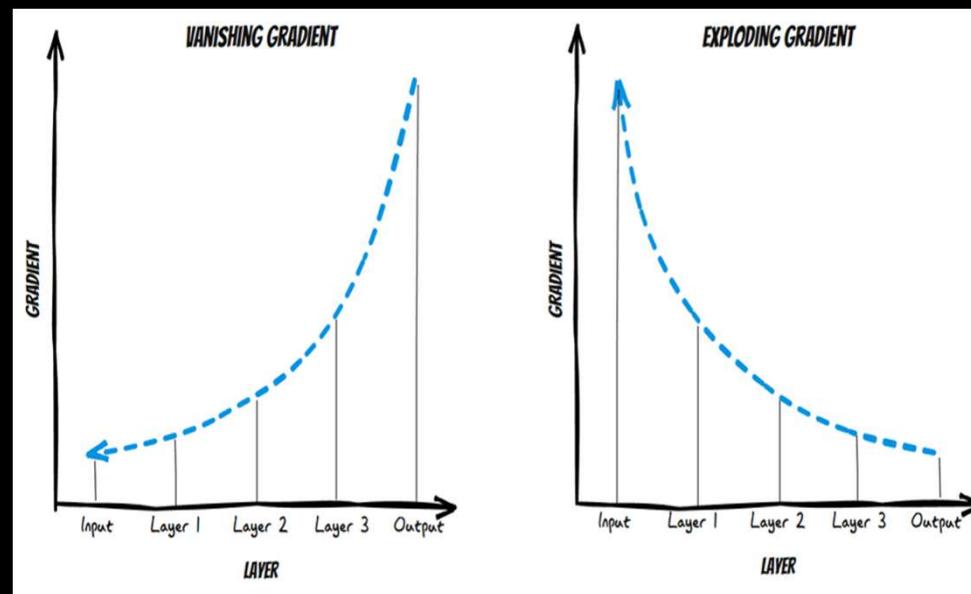
# Limitations of RNN, LSTM

- Limited ability to process context, limited memory  
*Text is sequential data, hence ability to process context is important*
- Suffered from problems such as vanishing gradient, exploding gradient
- Computationally intensive to train



# Limitations of RNN, LSTM

- Limited ability to process context, limited memory  
*Text is sequential data, hence ability to process context is important*
- Suffered from problems such as vanishing gradient, exploding gradient
- Takes lot of time to train and resources to train



# Limited Context

LSTMs struggle to distinguish between "bank" in **river bank** vs. **I went to the bank to deposit money**

- **Lack of Word Context Awareness**

LSTMs use embeddings (e.g., Word2Vec, GloVe).

That assign a single vector representation per word, meaning "bank" has the same representation in both sentences.

- **Limited Context Window**

While LSTMs have memory, they often struggle with long-distance dependencies. If the sentence is long, the context necessary to disambiguate "bank" might be lost.

- **No Explicit Word Sense Disambiguation**

Do not dynamically adjust word meanings based on context.

# Enter Transformers!!!

Transformers are a type of neural network that have a **unique ability to recognize long-range connections within sequences**. They are particularly useful for **tasks like generating text**, as the model needs to comprehend the preceding words in order to produce the next one.

**The introduction of transformers in 2018 was a groundbreaking moment for the field of natural language processing.**



Seminal paper published in 2017

# Transformers – key capabilities

Transformers  
are able to  
learn long-  
range  
dependencies  
in sequences.

Transformers  
are able to be  
trained on very  
large datasets.

Transformers  
are able to be  
parallelized