Modul 1

Artificial Intelligence

Introduction to Deep Learning (Part 1)

Job Connector Program

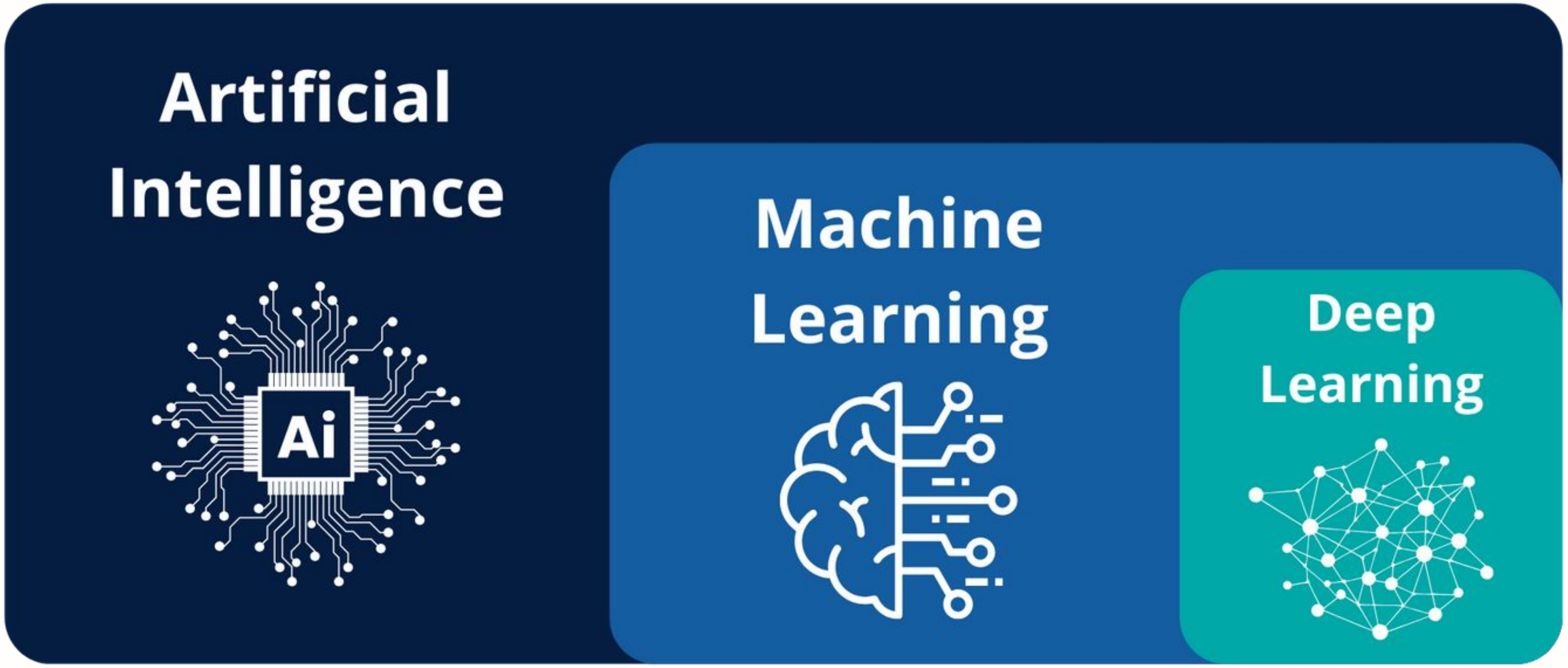
Outline

- Getting Started to Deep Learning
- Convolutional Neural Network (CNN)
- Popular CNN Architecture
- Recurrent Neural Network (RNN)
- Long Short Term Memory (LSTM)
- Transfer Learning

Deep Learning

Getting Started to Deep Learning

Area of AI, ML, and DL



Why We Need Deep Learning?

Rule-Based

UserId	Amount of Transaction	Last Transaction (days)
1	1000000	2
2	1500000	10
3	2350000	14
4	1250000	7

Churn Classification:

```
If Last Transaction >7 days:
   Churn
else:
   Not-churn
```

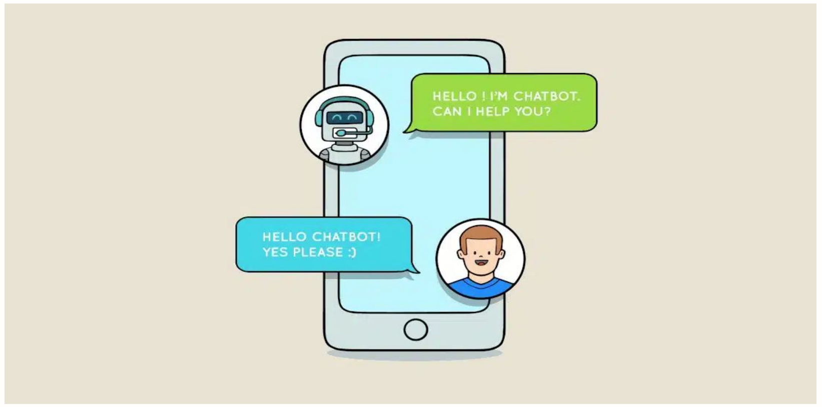
Why We Need Deep Learning? (cont'd)

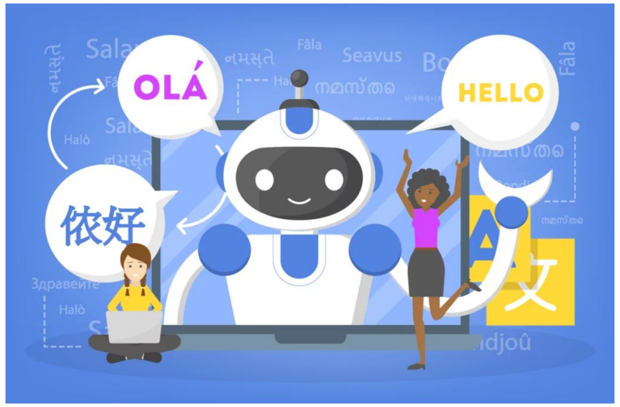
Machine Learning

UserId	Amount of Transaction	Last Transaction (days)	Days from Registration	City Code	is promo	Discount (%)
1	1000000	2	10	1	1	0
2	1500000	10	30	3	0	25
3	2350000	14	150	2	0	10
4	1250000	7	180	3	0	0

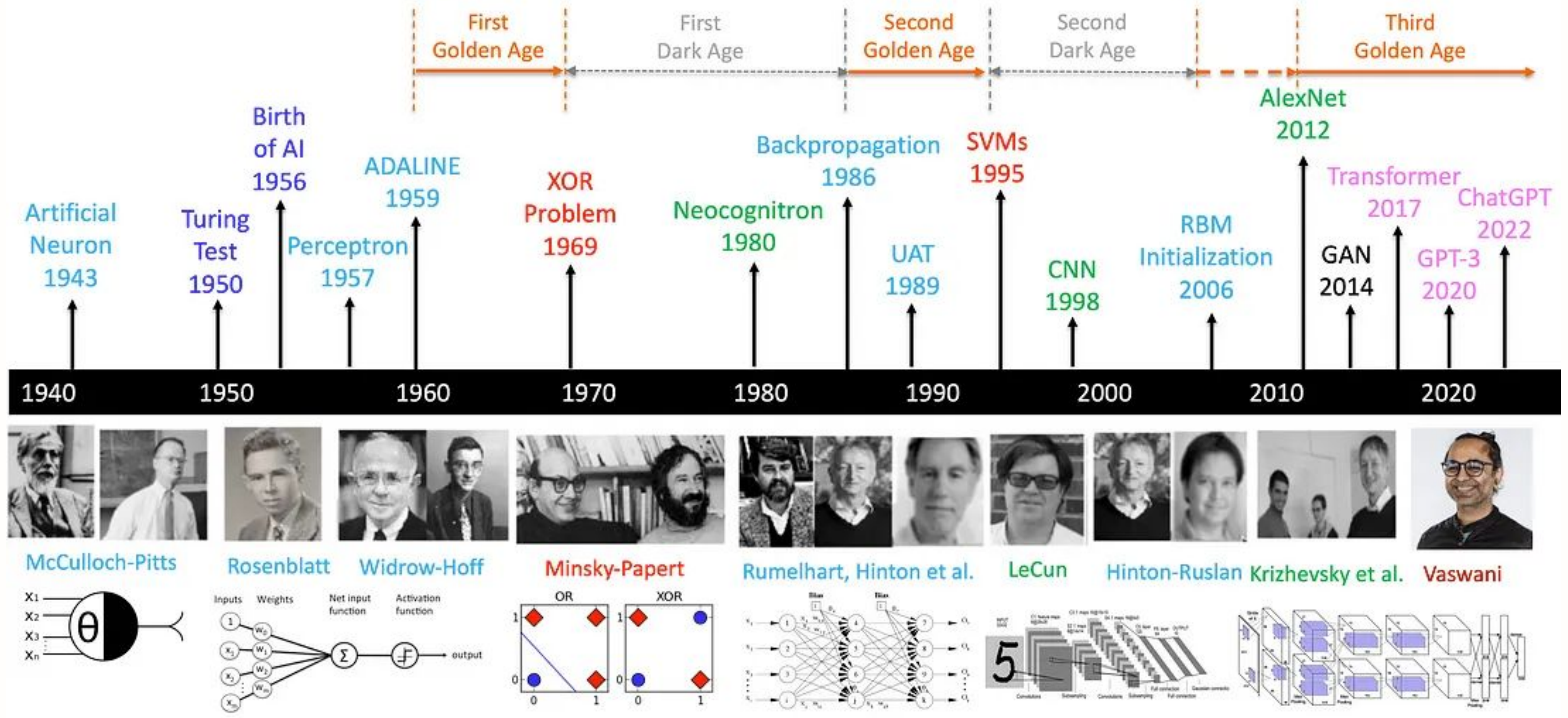
Why We Need Deep Learning? (cont'd)

Deep Learning

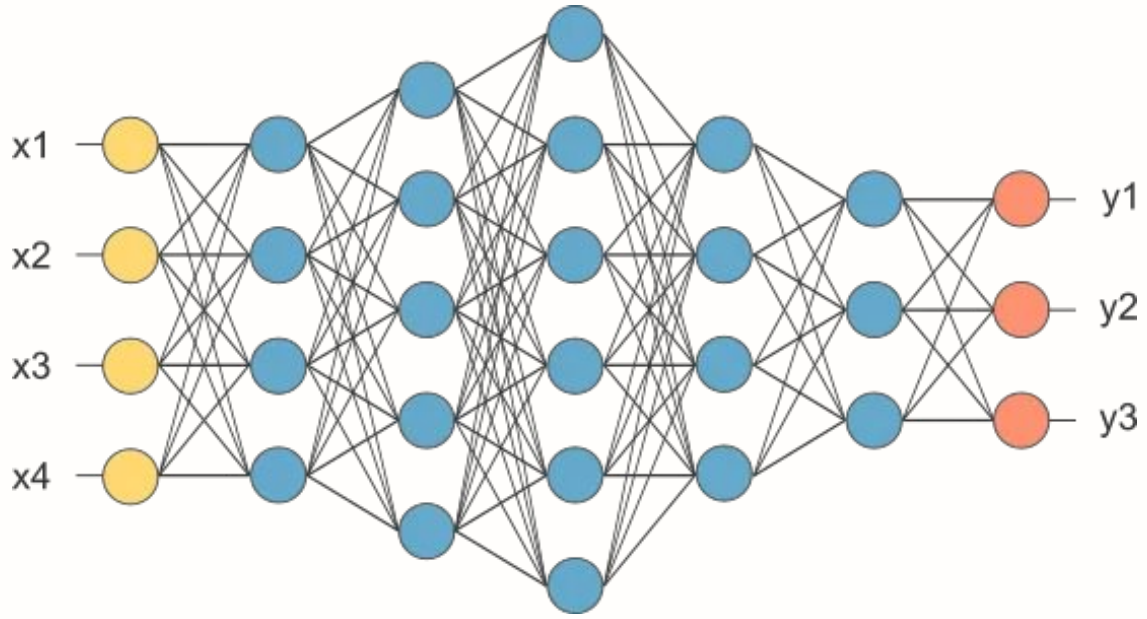

HELLO ! I'M CHATBOT. CAN I HELP YOU?
HELLO CHATBOT! YES PLEASE :)



History of Deep Learning



About Deep Learning

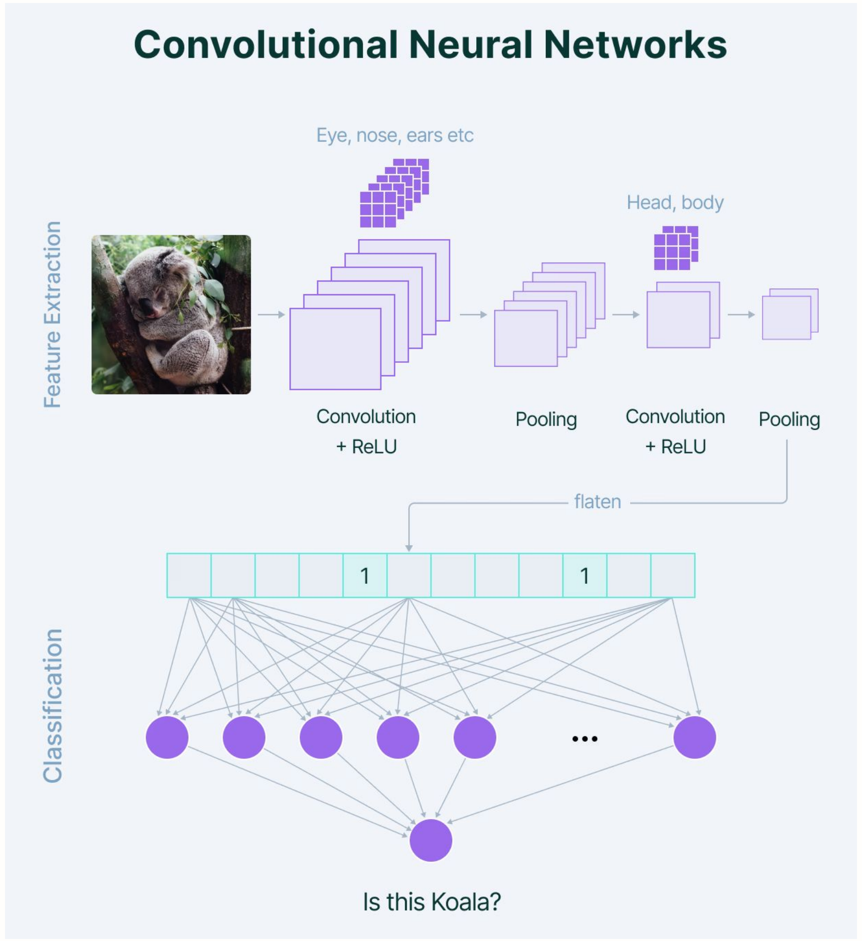


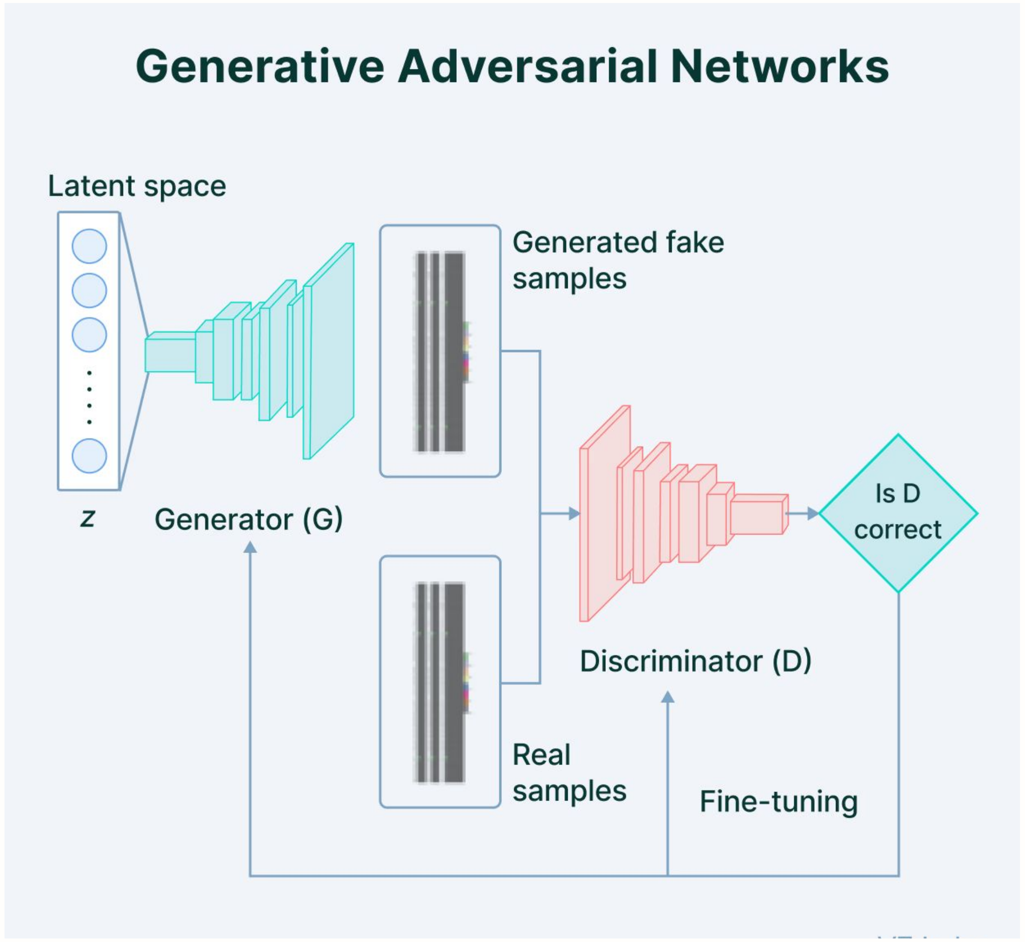
Deep learning is a branch of machine learning that uses artificial neural networks with many layers (hence "deep") to automatically learn patterns from large amounts of data. Inspired by how the human brain processes information, deep learning models can recognize complex structures and features in data without needing to be explicitly programmed for each task.

Deep Learning Process

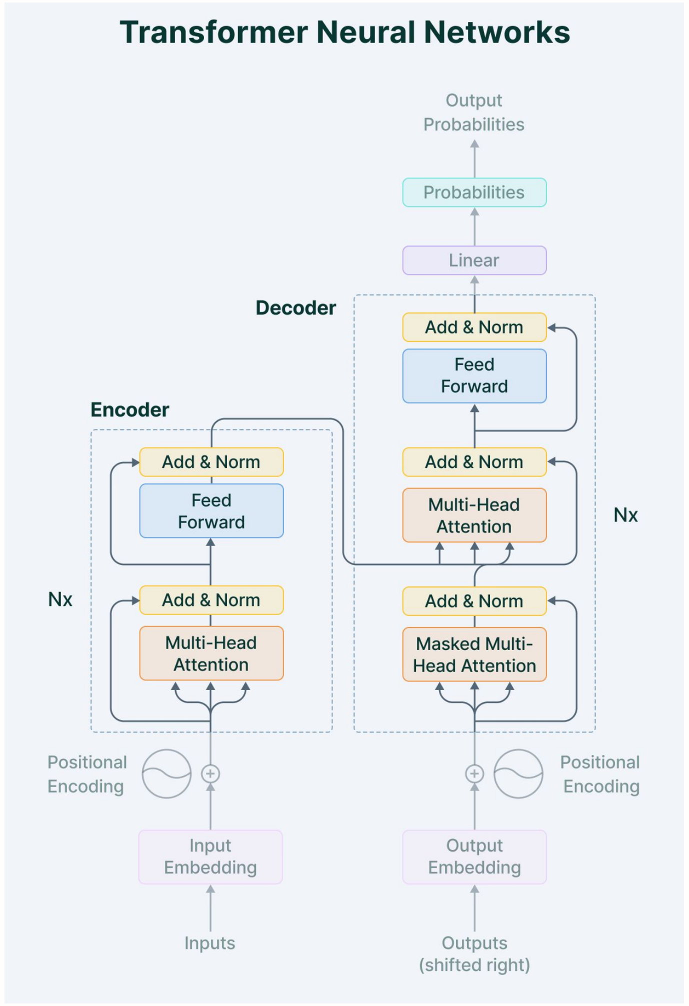
- **Input Data**: Feed raw data like images, text, or sound into the system.
- **Preprocessing**: Clean and prepare the data for training.
- **Layers of the Neural Network**:
  - Input Layer: Receives the data.
  - Hidden Layers: Learn patterns step by step.
  - Output Layer: Gives the final result (like a label or number).
- **Forward Pass**: Data flows through the layers to make a prediction.
- **Loss Calculation**: Compare the prediction to the correct answer to see the error.
- **Backpropagation**: Send the error backward to adjust the model’s settings.
- **Optimization**: Improve the model by updating it to reduce errors.
- **Training**: Repeat the steps many times to help the model learn.
- **Testing**: Check how well the model works on new, unseen data.

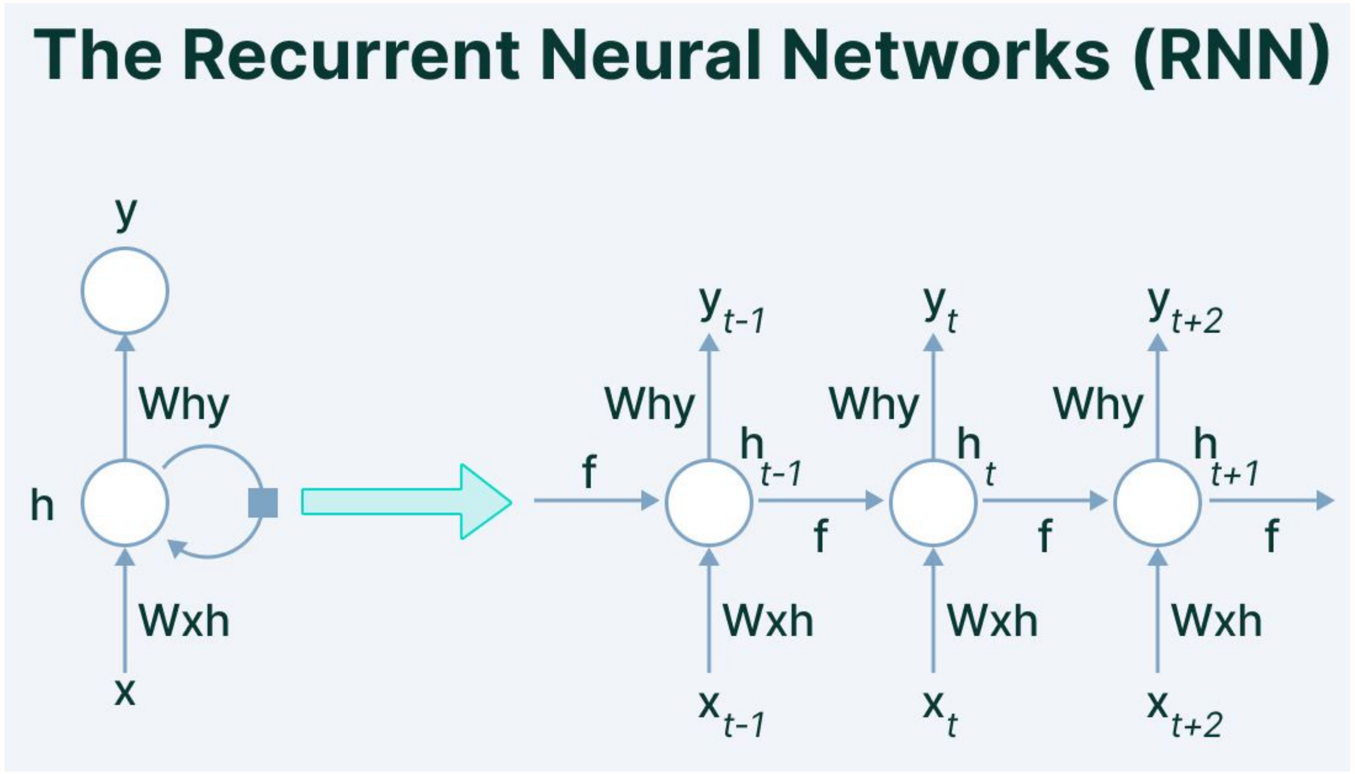
Several Examples of Deep Learning Architecture





Several Examples of Deep Learning Architecture (cont'd)





Deep Learning

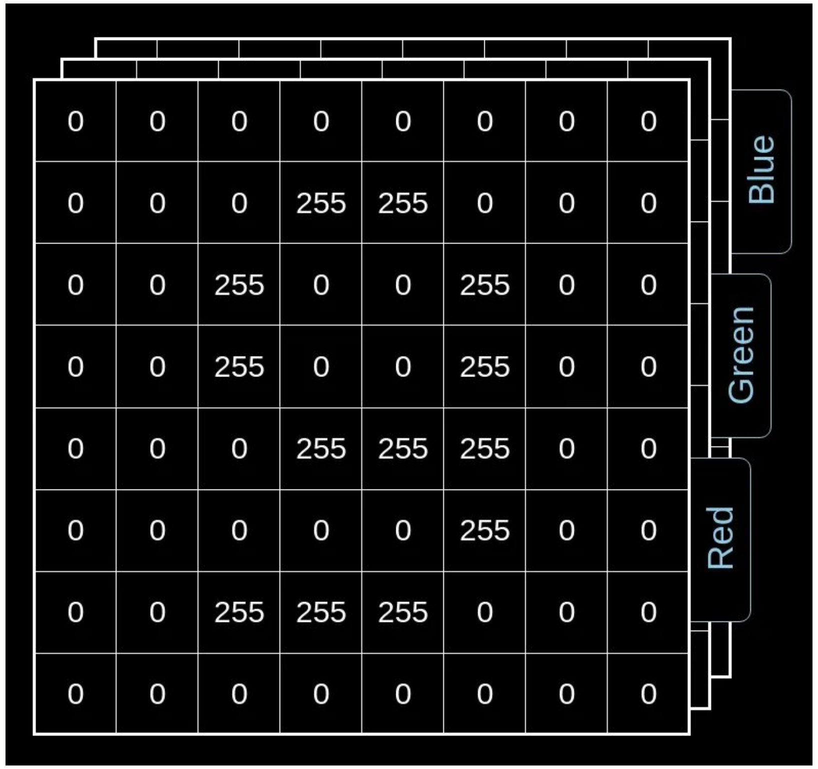
Convolutional Neural Network (CNN)

How do Computer See Images?

Human Vision

VS

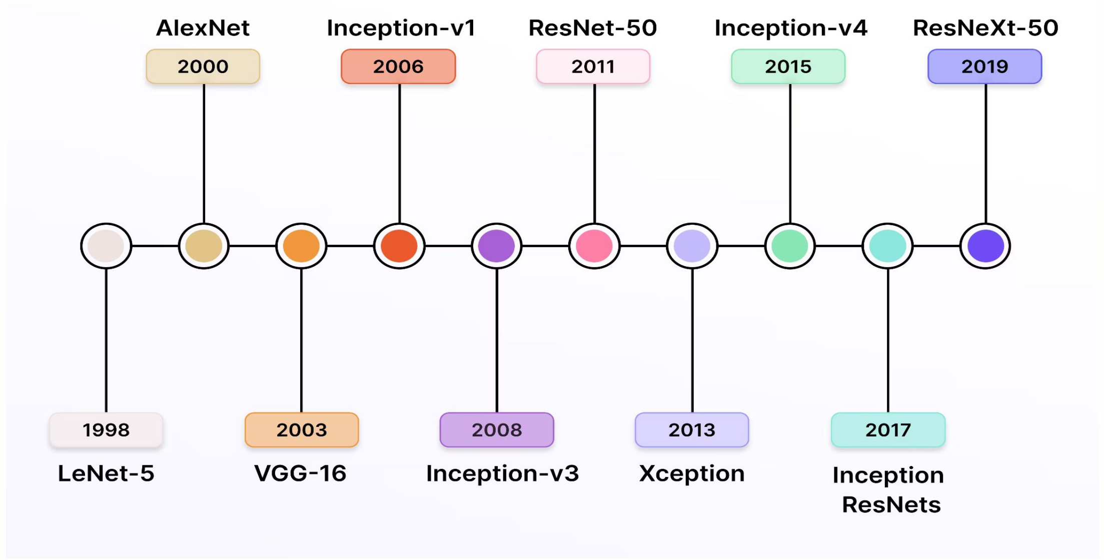
Computer Vision



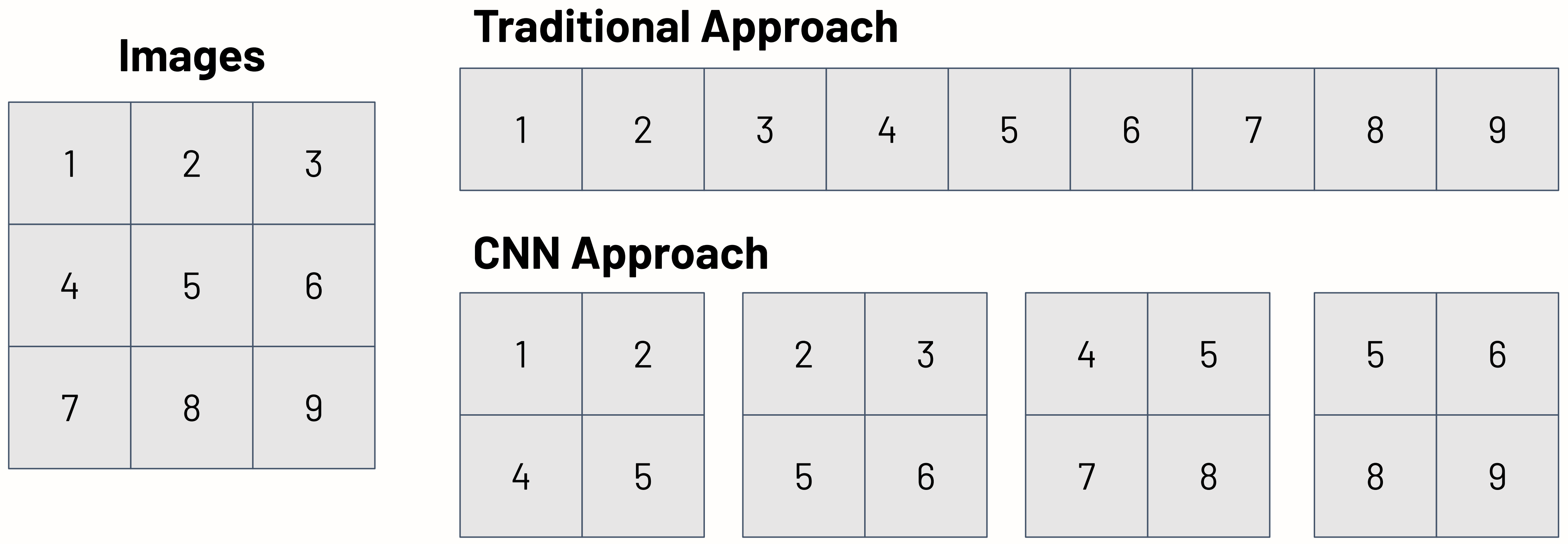
0	0	0	0	0	0	0	0
0	0	0	255	255	0	0	0
0	0	255	0	0	255	0	0
0	0	255	0	0	255	0	0
0	0	0	255	255	255	0	0
0	0	0	0	0	255	0	0
0	0	255	255	255	0	0	0
0	0	0	0	0	0	0	0

What is Convolutional Neural Network?

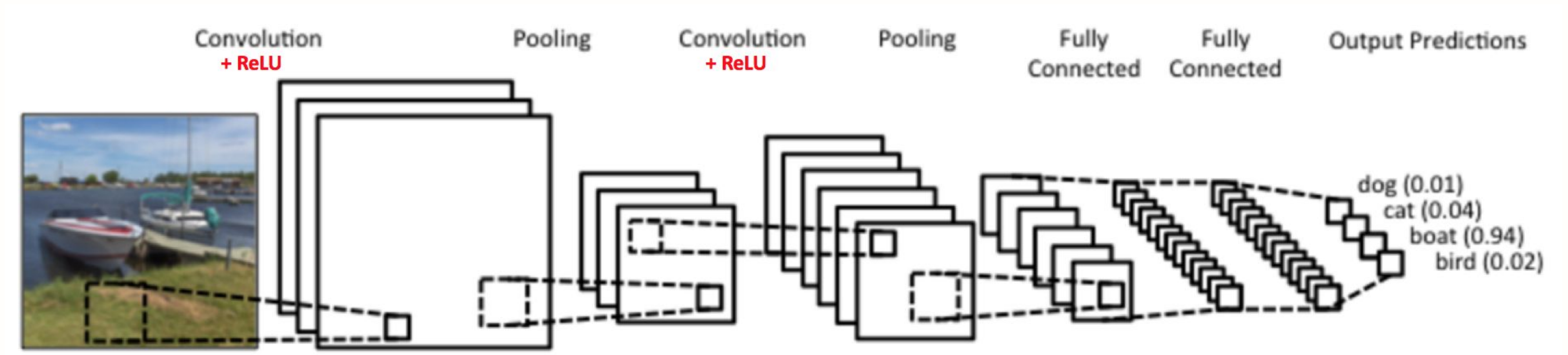
A CNN (Convolutional Neural Network) is a special type of deep learning model that is very good at processing images. Instead of looking at every pixel individually, CNNs use filters (also called kernels) to scan over the image and detect important features like edges, shapes, or patterns. These features help the model understand what's in the image.



Why CNN Needed?



Simple CNN Architecture



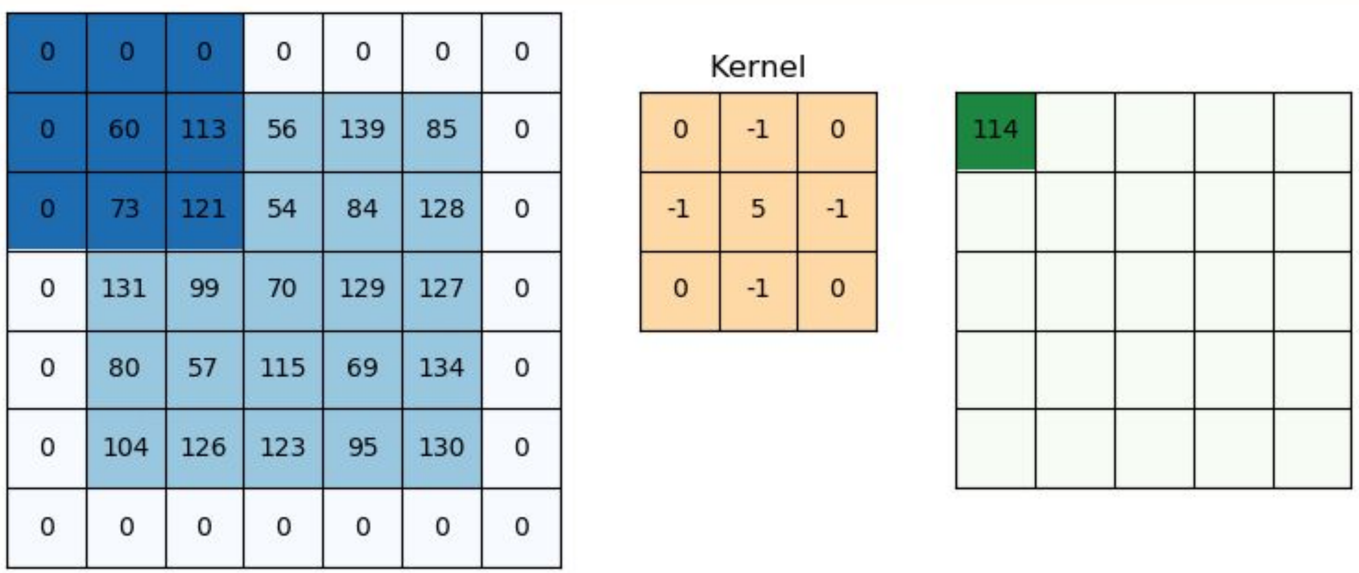
There are four main operations in the ConvNet shown in Figure above: **Convolution**, **Non Linearity (ReLU)**, **Pooling or Sub Sampling**, **Classification (Fully Connected Layer)**

What is Convolution?

Kernel convolution is not only used in CNNs, but is also a key element of many other Computer Vision algorithms.

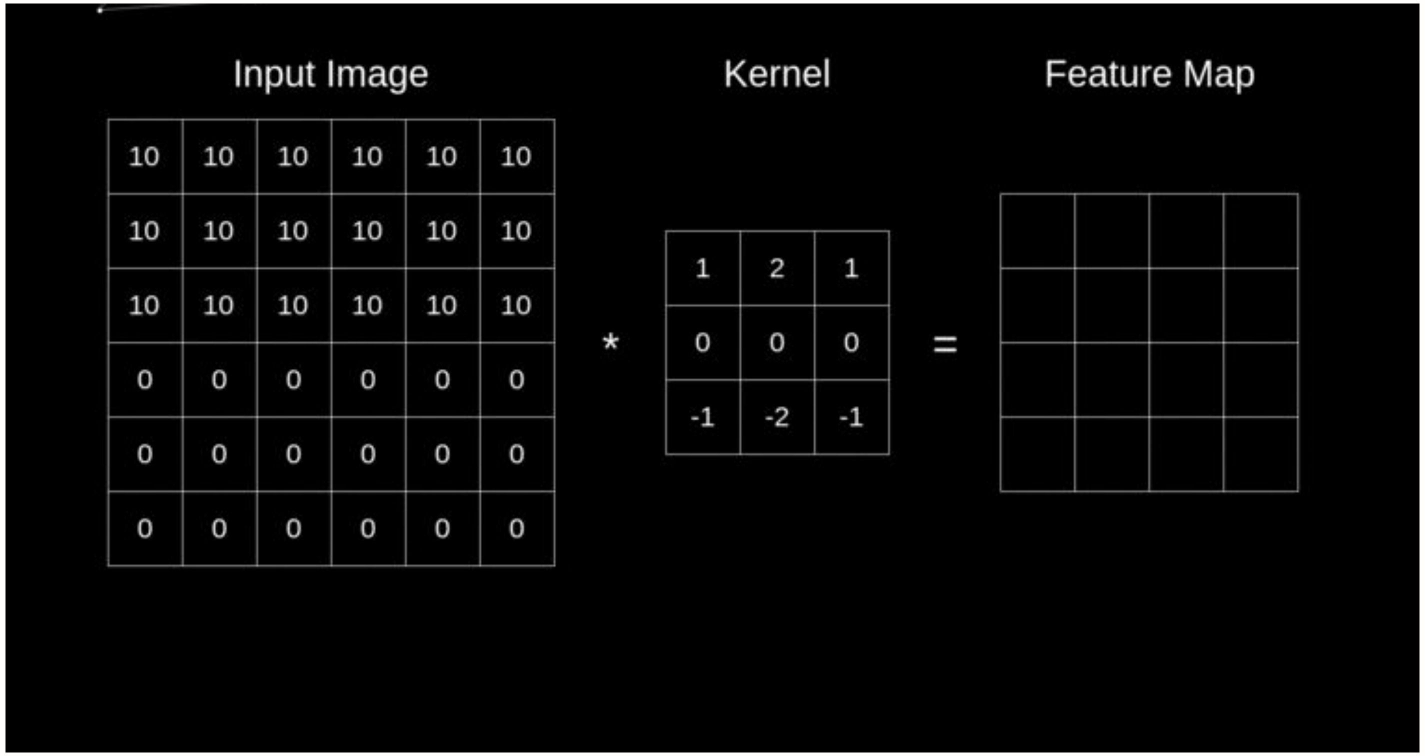
It is a process where we take a small matrix of numbers (called **kernel** or **filter**), we pass it over our image and transform it based on the values from filter.

This layer aims to extract local features such as texture, pattern, contour, etc.



$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 60 & 113 & 56 & 139 & 85 & 0 \\ 0 & 73 & 121 & 54 & 84 & 128 & 0 \\ 0 & 131 & 99 & 70 & 129 & 127 & 0 \\ 0 & 80 & 57 & 115 & 69 & 134 & 0 \\ 0 & 104 & 126 & 123 & 95 & 130 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad \begin{bmatrix} 114 & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \end{bmatrix}$$

Convolution Formula



10	10	10	10	10	10
10	10	10	10	10	10
10	10	10	10	10	10
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

*

1	2	1
0	0	0
-1	-2	-1

=

$$G[m,n] = (f * h)[m,n] = \sum_{j} \sum_{k} h[j,k] f[m-j, n-k]$$

Pooling Layer

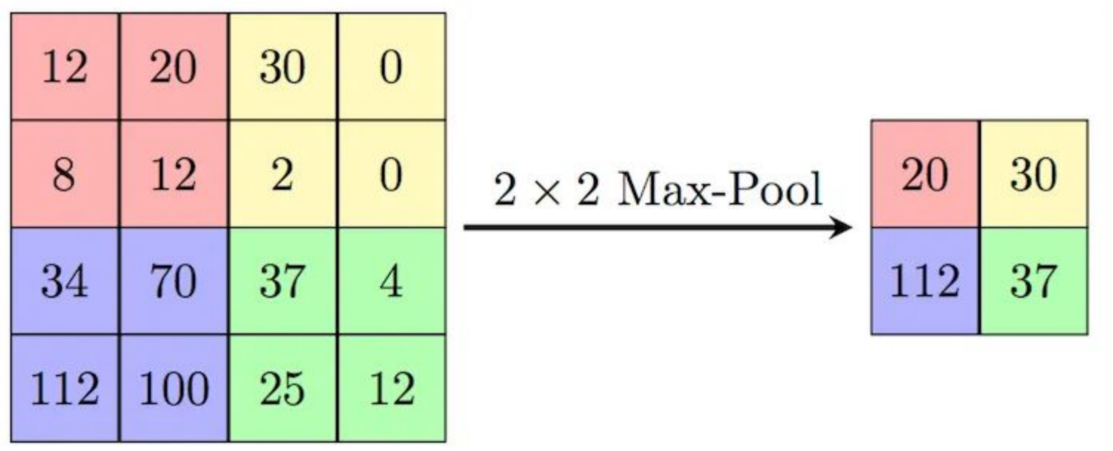
In Convolutional Neural Networks (CNNs), pooling layers are used to reduce the size of the data (called tensors) and make computations faster, while still keeping the most important information. Pooling works by dividing the image into small regions and applying a simple operation to each one.

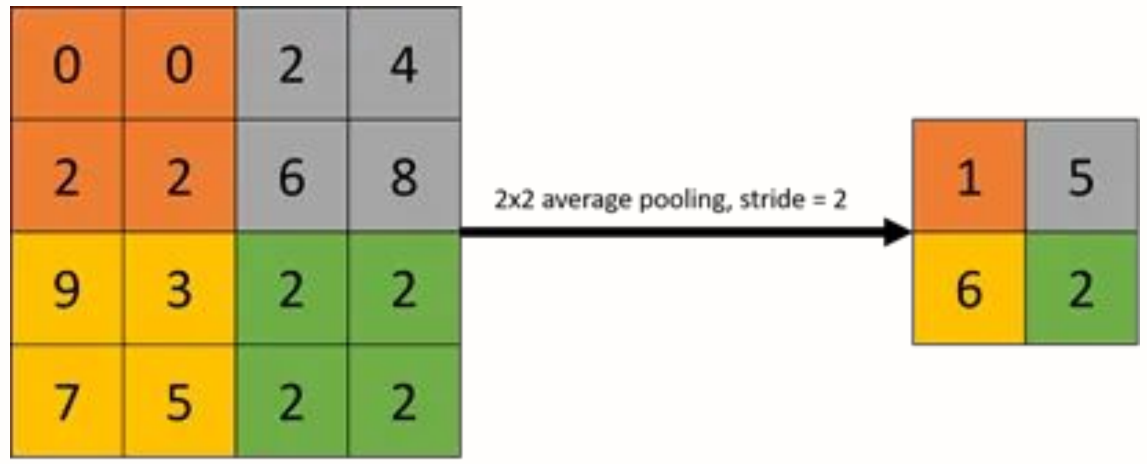
There are two common types of pooling:

- **Max Pooling**: Takes the largest value from each region. This helps capture strong features like edges.
- **Average Pooling**: Takes the average of all values in each region, which gives a more general, smoothed-out view.

This layer aims to reduce the data dimension while maintaining information. In addition, this layer also maintains object information even though its position shifts slightly (different from its original position).

Max Pooling and Average Pooling





Max Pooling

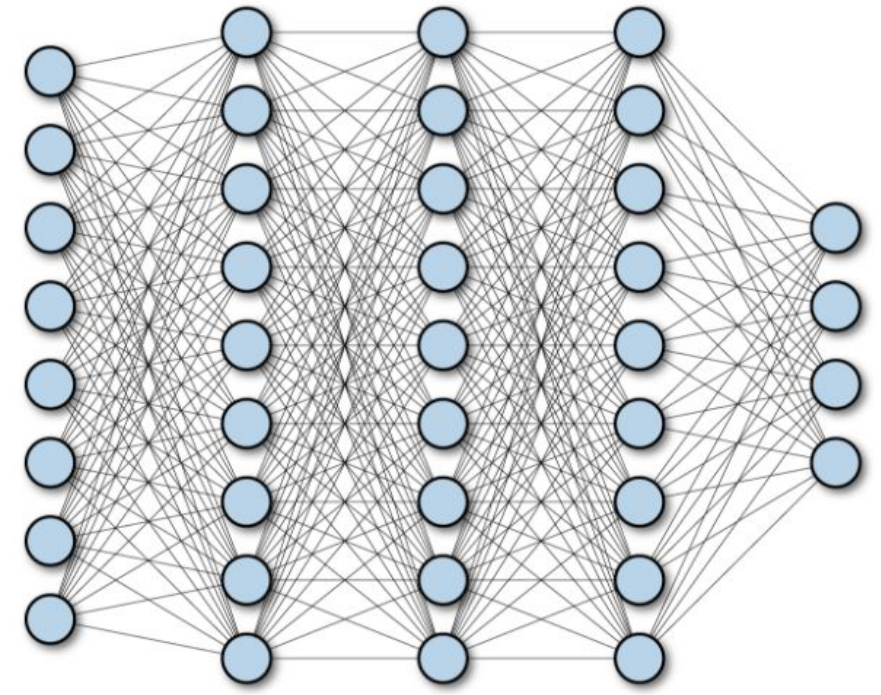
Take the large value every region

Average Pooling

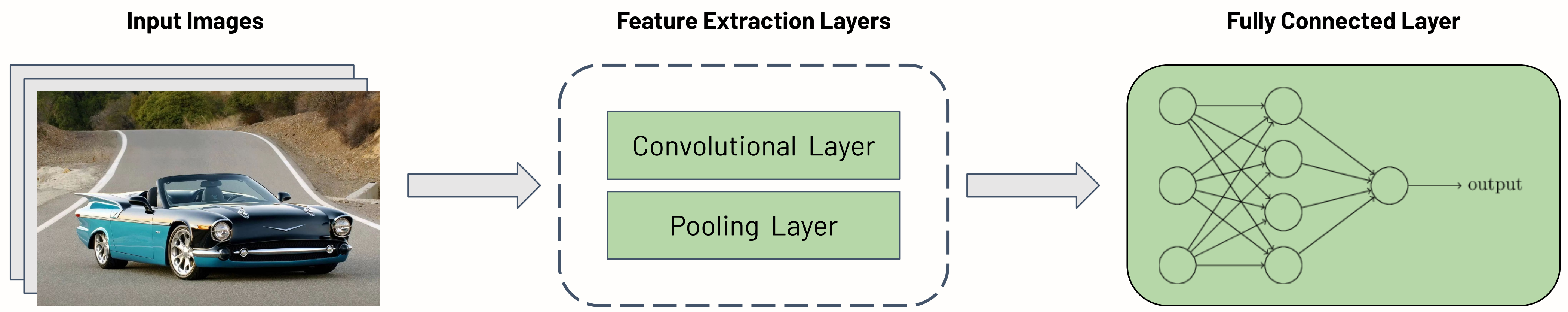
Calculate average value every region

Fully Connected Layer

Neurons in this layer have full connectivity with all neurons in the preceding and succeeding layer as seen in regular FCNN. This is why it can be computed as usual by a matrix multiplication followed by a bias effect. The FC layer helps to map the representation between the input and the output.



CNN Flow


Input Images
Feature Extraction Layers
Convolutional Layer
Pooling Layer
Fully Connected Layer
output

Building CNN Architecture using Keras

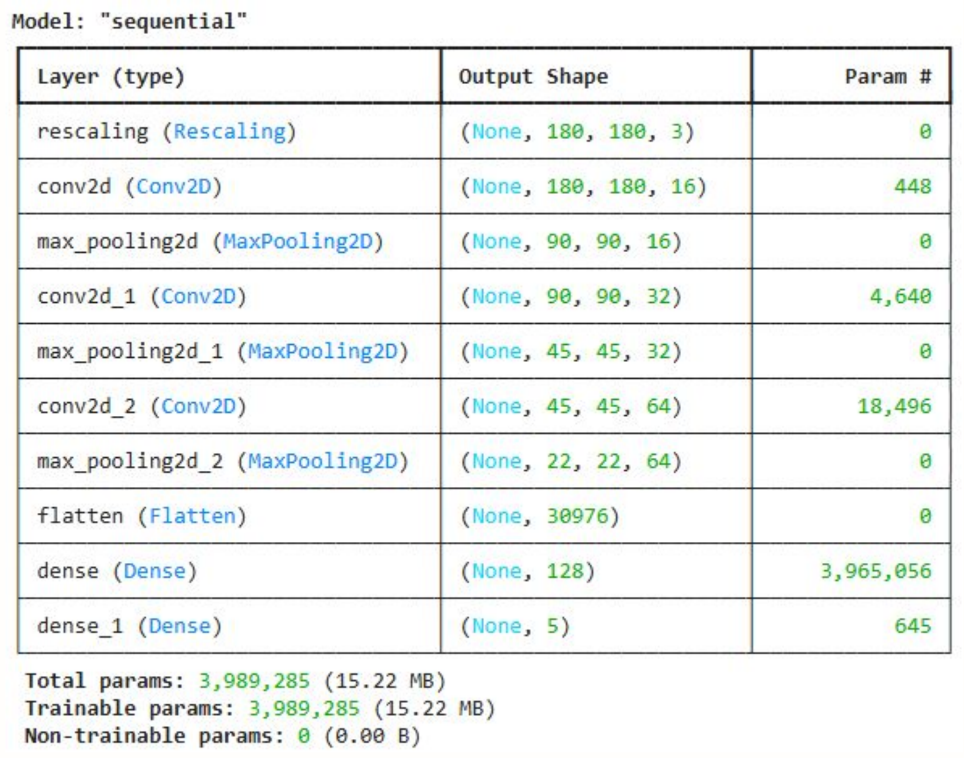
Purwadhika
Digital Technology School

```python
from tensorflow import keras
from tensorflow.keras import layers

num_classes = 5 # There are 5 classes of flowers

model = tf.keras.Sequential([
  layers.Input((img_height, img_width, 3)),
  layers.Rescaling(1./255),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])

model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```



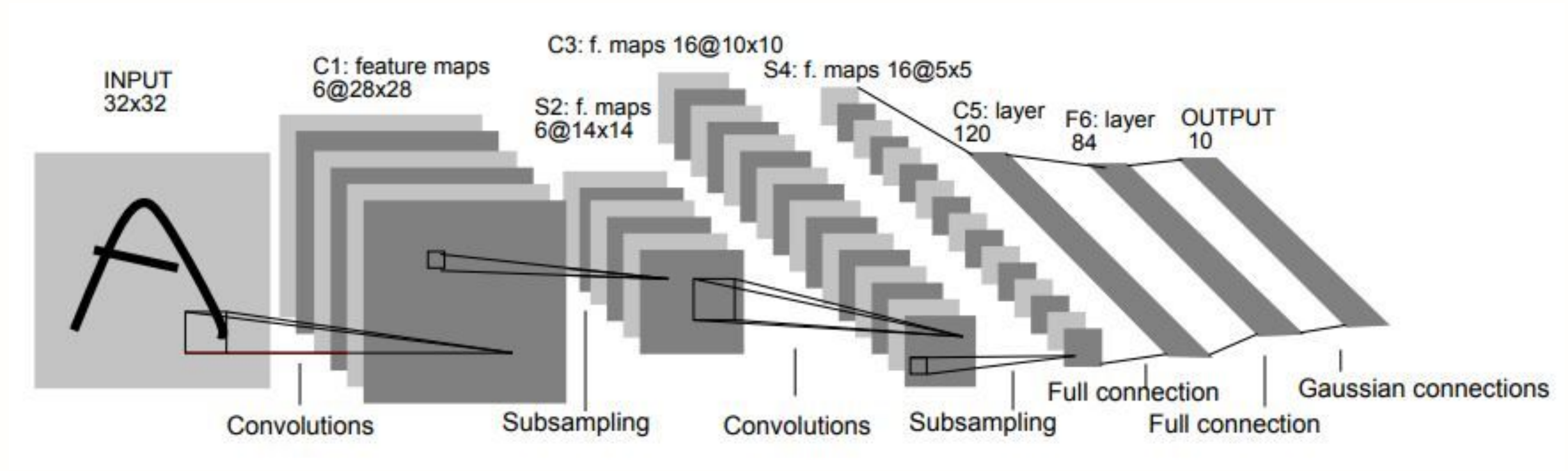
```
Model: "sequential"
┌──────────────────────────────────┬─────────────────────────┬──────────────┐
│ Layer (type)                     │ Output Shape            │      Param # │
├──────────────────────────────────┼─────────────────────────┼──────────────┤
│ rescaling (Rescaling)            │ (None, 180, 180, 3)     │            0 │
├──────────────────────────────────┼─────────────────────────┼──────────────┤
│ conv2d (Conv2D)                  │ (None, 180, 180, 16)    │          448 │
├──────────────────────────────────┼─────────────────────────┼──────────────┤
│ max_pooling2d (MaxPooling2D)     │ (None, 90, 90, 16)      │            0 │
├──────────────────────────────────┼─────────────────────────┼──────────────┤
│ conv2d_1 (Conv2D)                │ (None, 90, 90, 32)      │        4,640 │
├──────────────────────────────────┼─────────────────────────┼──────────────┤
│ max_pooling2d_1 (MaxPooling2D)   │ (None, 45, 45, 32)      │            0 │
├──────────────────────────────────┼─────────────────────────┼──────────────┤
│ conv2d_2 (Conv2D)                │ (None, 45, 45, 64)      │       18,496 │
├──────────────────────────────────┼─────────────────────────┼──────────────┤
│ max_pooling2d_2 (MaxPooling2D)   │ (None, 22, 22, 64)      │            0 │
├──────────────────────────────────┼─────────────────────────┼──────────────┤
│ flatten (Flatten)                │ (None, 30976)           │            0 │
├──────────────────────────────────┼─────────────────────────┼──────────────┤
│ dense (Dense)                    │ (None, 128)             │    3,965,056 │
├──────────────────────────────────┼─────────────────────────┼──────────────┤
│ dense_1 (Dense)                  │ (None, 5)               │          645 │
└──────────────────────────────────┴─────────────────────────┴──────────────┘
 Total params: 3,989,285 (15.22 MB)
 Trainable params: 3,989,285 (15.22 MB)
 Non-trainable params: 0 (0.00 B)
```

Deep Learning

Popular CNN Architecture

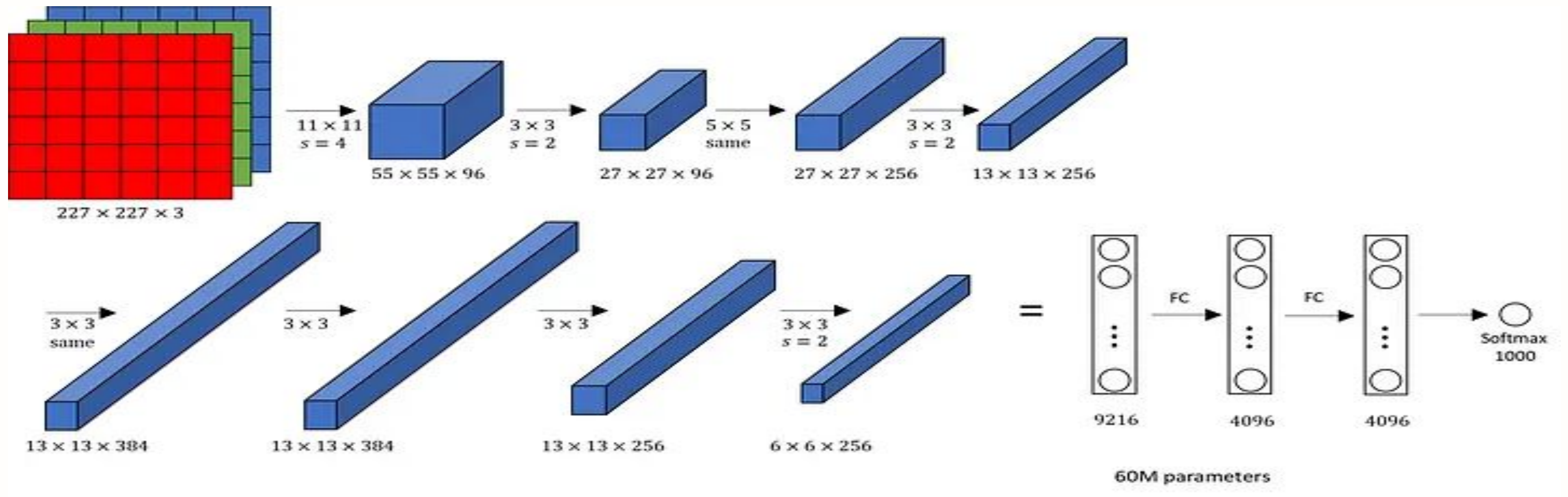
LeNet

LeNet-5 (1998): Developed by Yann LeCun and his team, LeNet-5 is one of the earliest CNN architectures. It was primarily designed for handwritten digit recognition tasks and consisted of convolutional layers, pooling layers, and fully connected layers.



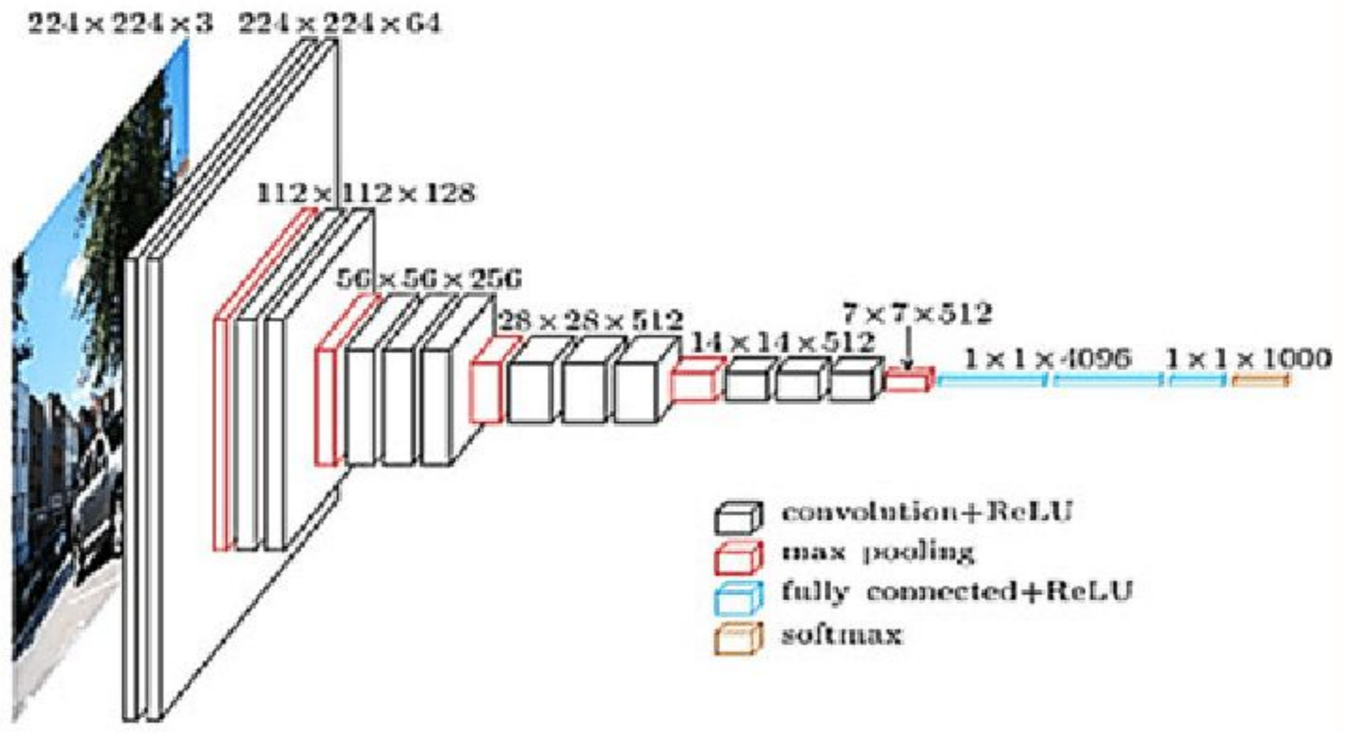
AlexNet

AlexNet (2012) - by Alex Krizhevsky - was a deep learning model that made a big impact in image recognition. It used several convolutional layers, ReLU activation, and techniques like dropout and data augmentation to improve learning. AlexNet won a major competition called ImageNet in 2012, showing how powerful deep CNNs can be for recognizing images.



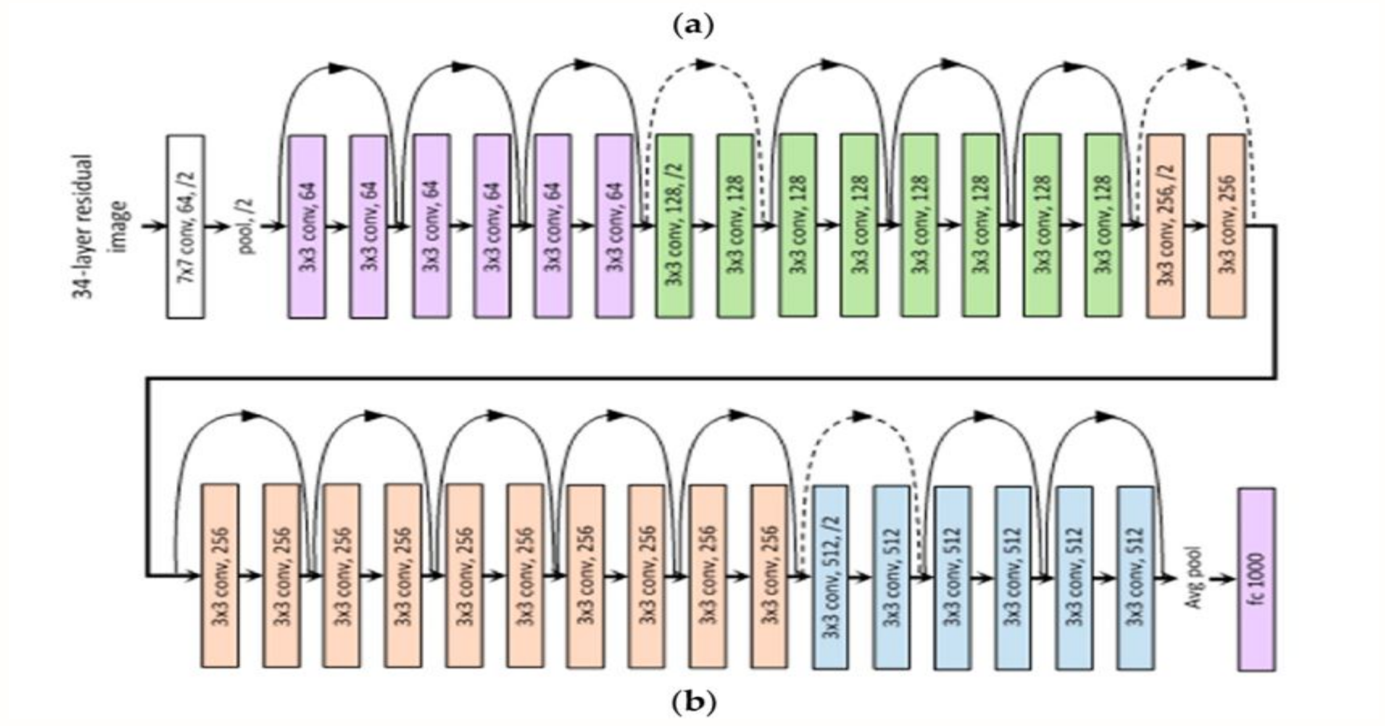
VGGNet

VGGNet (2014): VGGNet, developed by the Visual Geometry Group (VGG) at the University of Oxford, featured a simple and uniform architecture with very deep networks (up to 19 layers). It utilized small 3x3 convolutional filters and achieved impressive performance on image classification tasks.



ResNet

ResNet (2015): Residual Networks, or ResNets, were introduced by Kaiming He et al. ResNets addressed the problem of vanishing gradients in very deep networks by introducing skip connections, which enabled the gradient to flow directly through the network. This architecture allowed the training of extremely deep networks (up to 152 layers) effectively.



Try in Hands On

Deep Learning

Recurrent Neural Network (RNN)

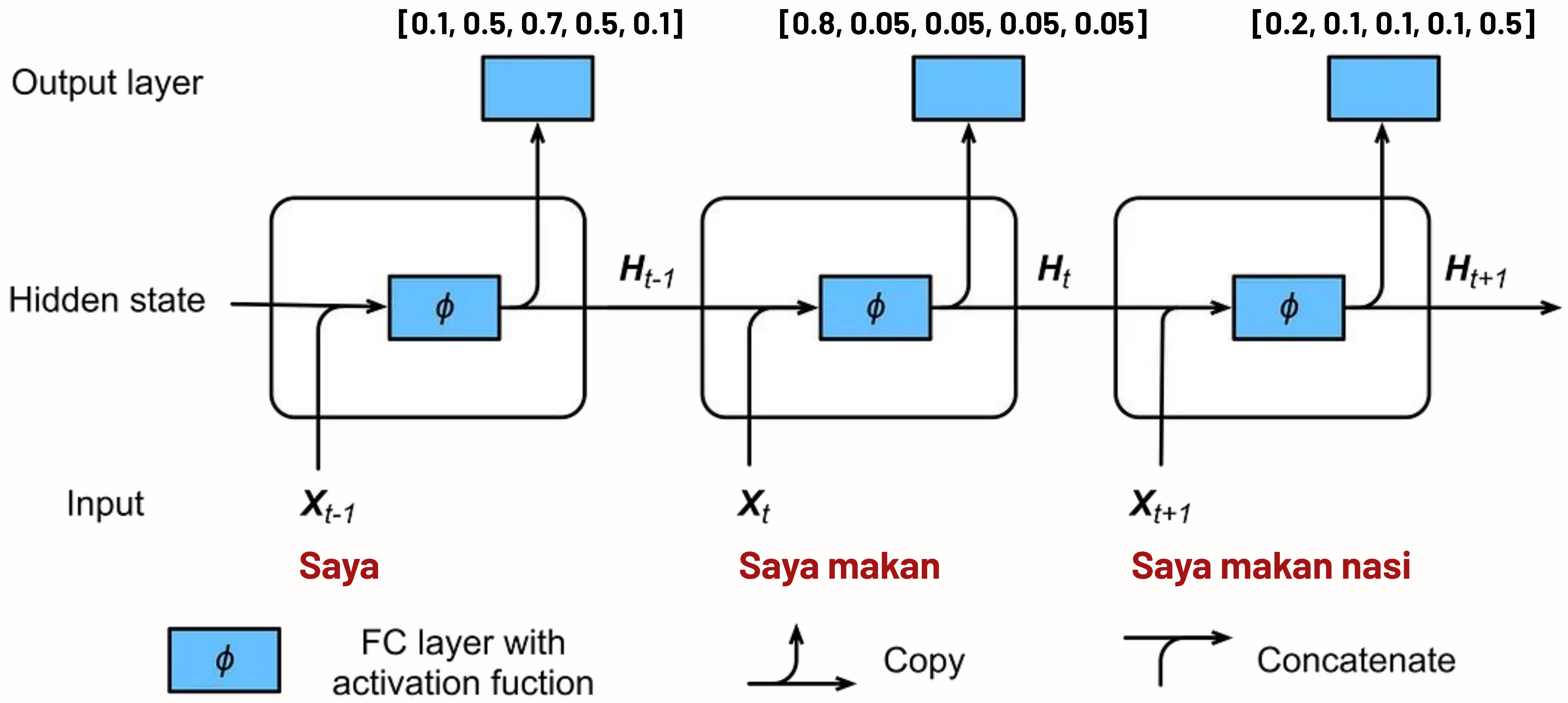
What is Recurrent Neural Network (RNN) ?

Recurrent neural networks are a type of neural network used for sequential data. But before we dive deep into RNN’s let's understand what are neural networks.

A Neural Network consists of different layers connected to each other, working on the structure and function of a human brain. It learns from huge volumes of data and uses complex algorithms to train a neural net.

RNN works on the principle of saving the output of a particular layer and feeding this back to the input in order to predict the output of the layer.

Simple RNN Architecture

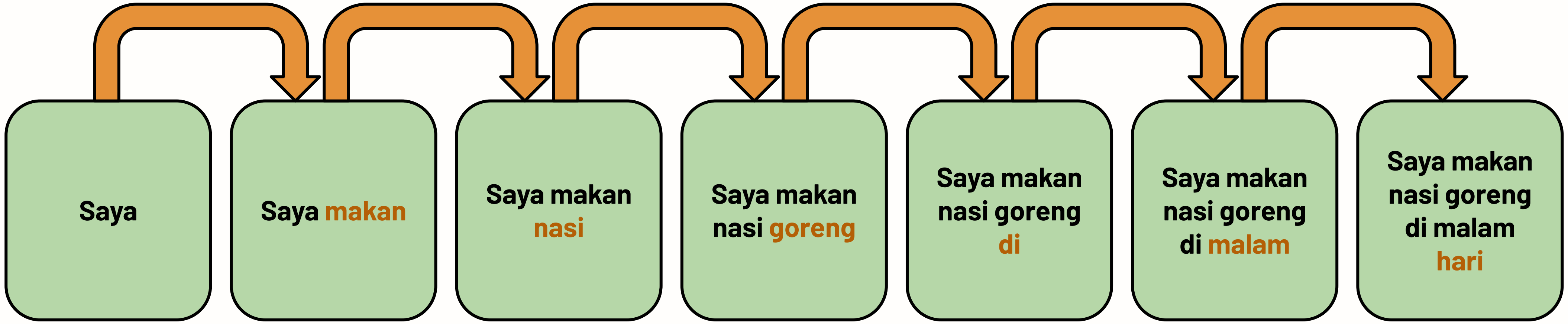


How RNN Works?

- The input layer ‘X’ takes in the input to the neural network and processes it and passes it onto the middle layer.
- The Hidden state ‘H’ can consist of multiple hidden layers, each with its own activation functions and weights and biases.
- The Recurrent Neural Network will standardize the different activation functions and weights and biases so that each hidden layer has the same parameters. Then, instead of creating multiple hidden layers, it will create one and loop over it as many times as required.
- Because of its internal memory, RNN’s produces output, copies that output and loops it back into the network.

Example

Sentence: **Saya makan nasi goreng di malam hari**


Saya
Saya makan
Saya makan nasi
Saya makan nasi goreng
Saya makan nasi goreng di
Saya makan nasi goreng di malam
Saya makan nasi goreng di malam hari

RNN’s Problems

🔥 **Exploding Gradients**

- Happens during training of Recurrent Neural Networks (RNNs) (and other deep networks
- The gradients (used to update the model’s weights) become very large
- This causes the weights to change too drastically, making the model unstable or unable to learn properly
- You might see the loss value suddenly jump to NaN (not a number) or extremely high numbers
- It's like the model is panicking and overreacting to mistakes.

Example: Imagine you're trying to balance on a tightrope, and someone gives you a push — but instead of a small correction, you jump 5 meters away. That’s exploding gradients.

RNN's Problems (cont'd)

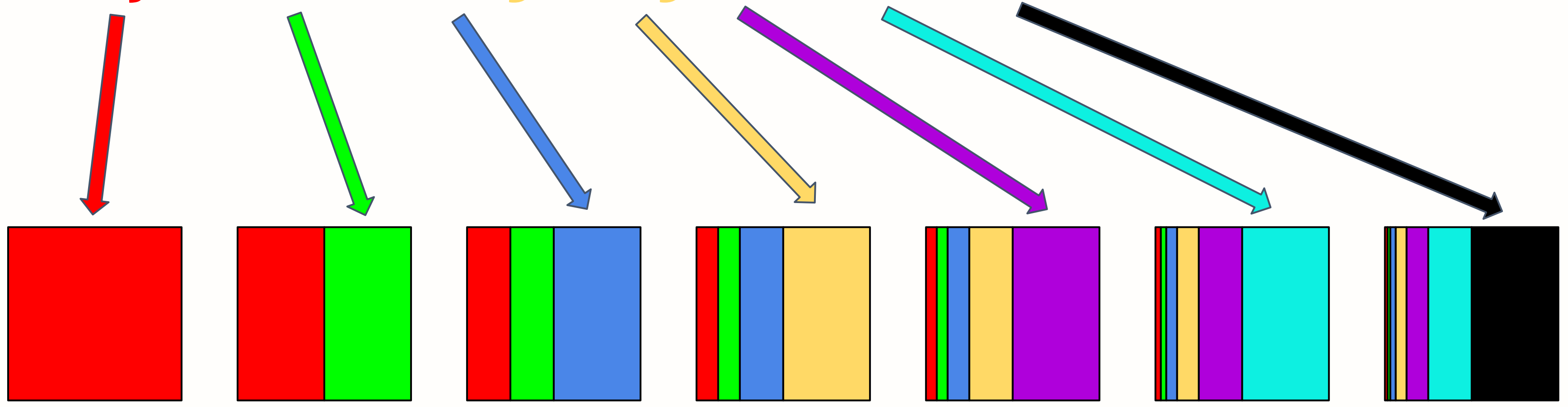
🧊 **Vanishing Gradients**

- Also happens during training, especially in deep or long RNNs
- The gradients become very small as they are passed backward through layers
- The model learns very slowly or stops learning completely, especially for early layers or earlier time steps in sequences
- This makes it hard for RNNs to learn long-term dependencies (i.e., remembering things from earlier in a sequence).

Example: It's like whispering a message down a long line of people, and by the time it gets to the end, it's so quiet it's lost.

Illustration

Sentence: **Saya makan nasi goreng di malam hari**

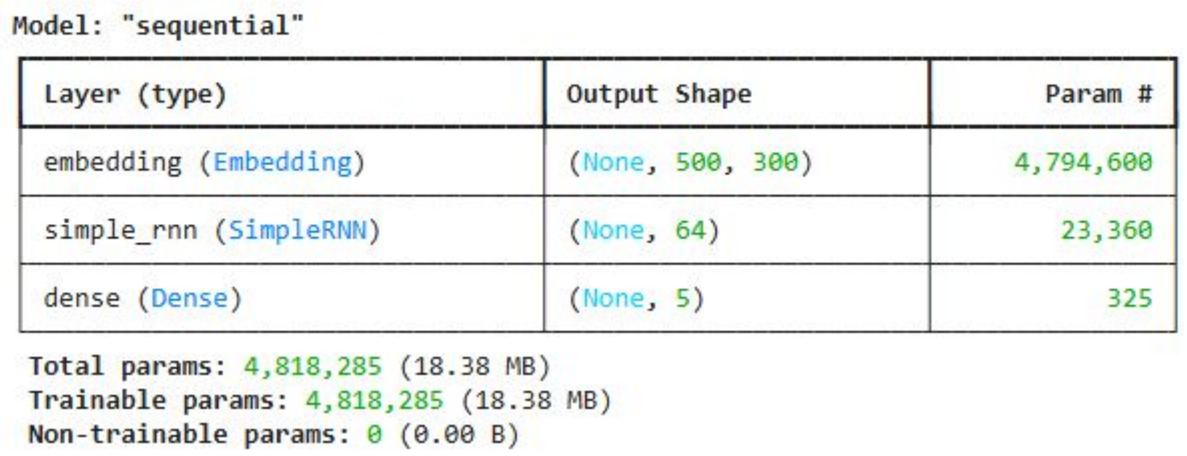


Informations that contained in vector representation

Build RNN Architecture using Keras

```python
# Build the RNN model
vocab_size = len(tokenizer.word_index) + 1
embedding_dim = 300 # Adjust as needed
model = Sequential()
model.add(Input(shape=(max_sequence_length,)))
model.add(Embedding(vocab_size, embedding_dim))
model.add(SimpleRNN(64)) # Adjust units as needed
model.add(Dense(len(le.classes_), activation='softmax')) # Output layer with softmax

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.summary()
```



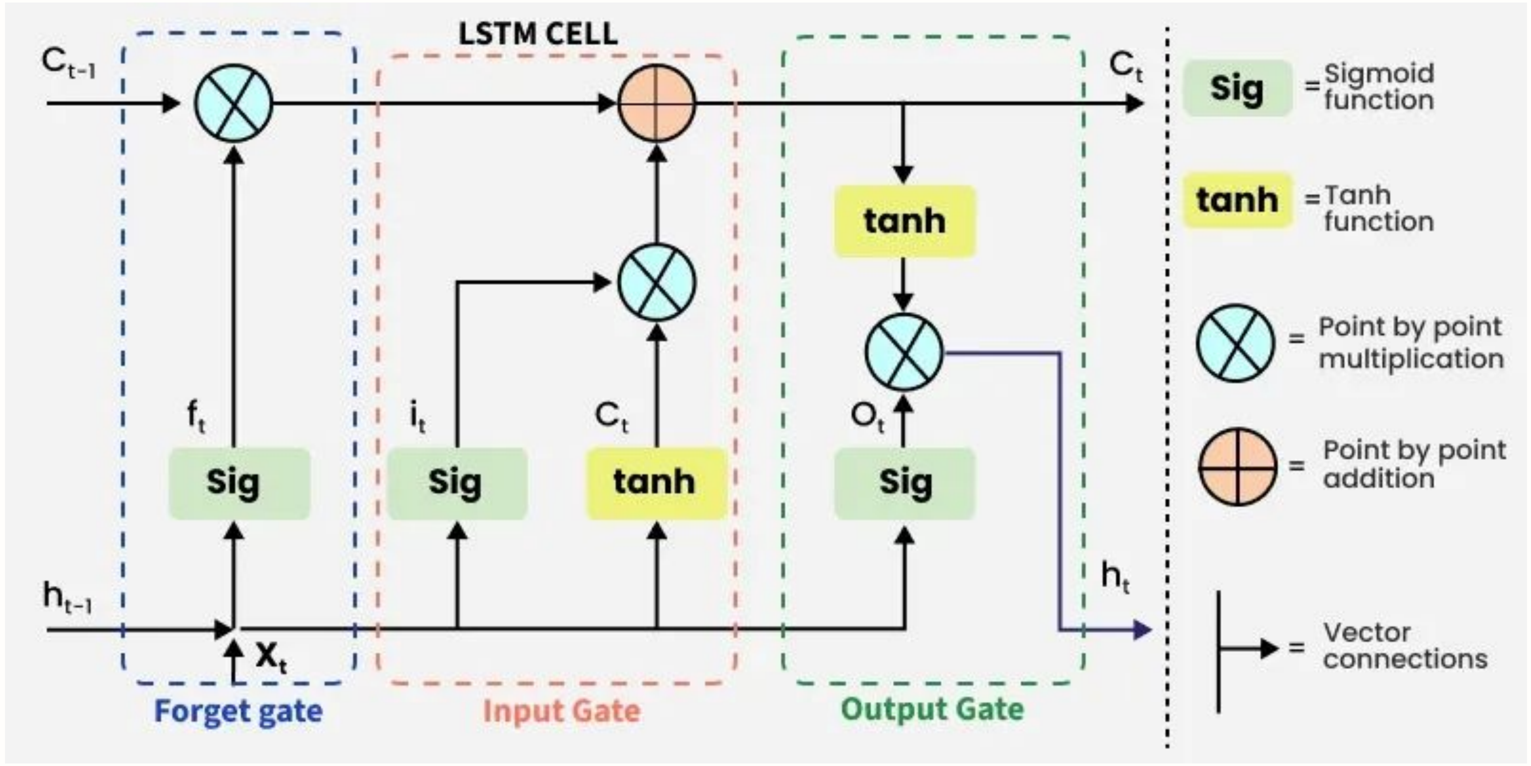
```
Model: "sequential"
┌──────────────────────────┬──────────────────┬────────────┐
│ Layer (type)             │ Output Shape     │    Param # │
├──────────────────────────┼──────────────────┼────────────┤
│ embedding (Embedding)    │ (None, 500, 300) │  4,794,600 │
├──────────────────────────┼──────────────────┼────────────┤
│ simple_rnn (SimpleRNN)   │ (None, 64)       │     23,360 │
├──────────────────────────┼──────────────────┼────────────┤
│ dense (Dense)            │ (None, 5)        │        325 │
└──────────────────────────┴──────────────────┴────────────┘
 Total params: 4,818,285 (18.38 MB)
 Trainable params: 4,818,285 (18.38 MB)
 Non-trainable params: 0 (0.00 B)
```

Deep Learning

Long Short Term Memory (LSTM)

What is Long Short Term Memory?

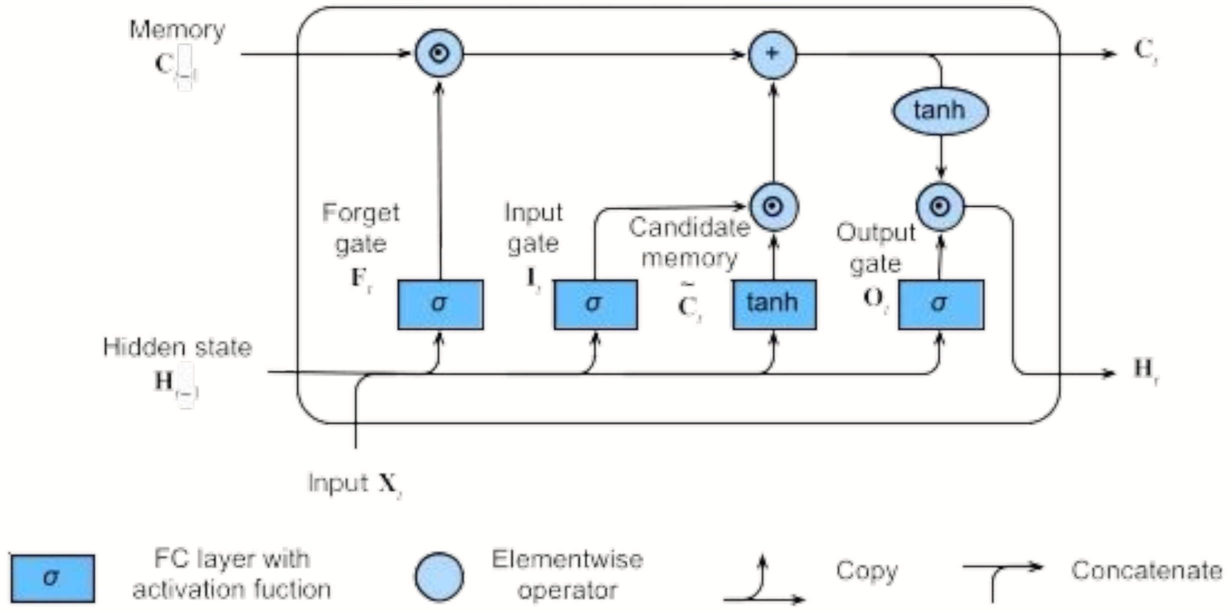
Long Short-Term Memory (LSTM) networks are an improved type of recurrent neural network (RNN) designed to better retain information over time. They solve the vanishing gradient problem, making them effective for time series tasks with varying time lags.



How LSTM Works?

An LSTM cell manages information through three key gates:

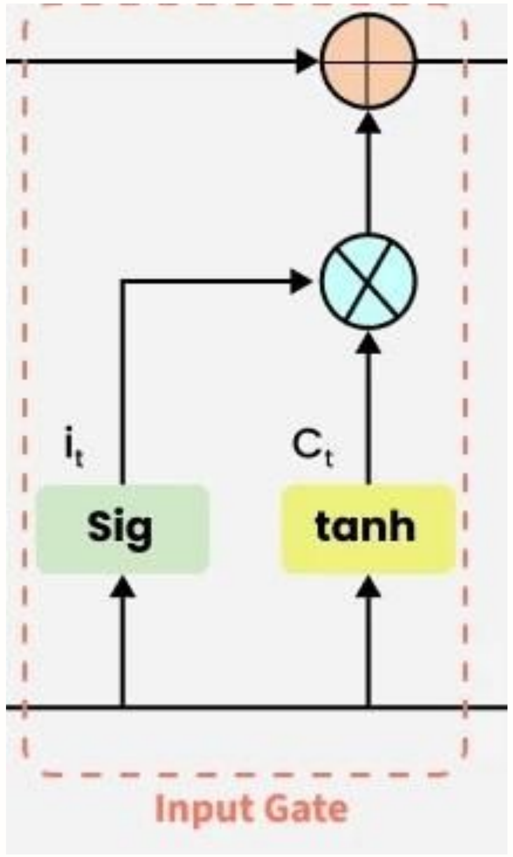
- Forget Gate: Decides what information from the previous cell state should be discarded.
- Input Gate: Determines which new information from the current input should be added to the cell state.
- Output Gate: Controls what information from the cell state is passed to the next hidden state.



Each gate uses a sigmoid activation function to regulate the flow of information. The cell state is updated through these gates, enabling the network to retain long-term dependencies in sequential data.

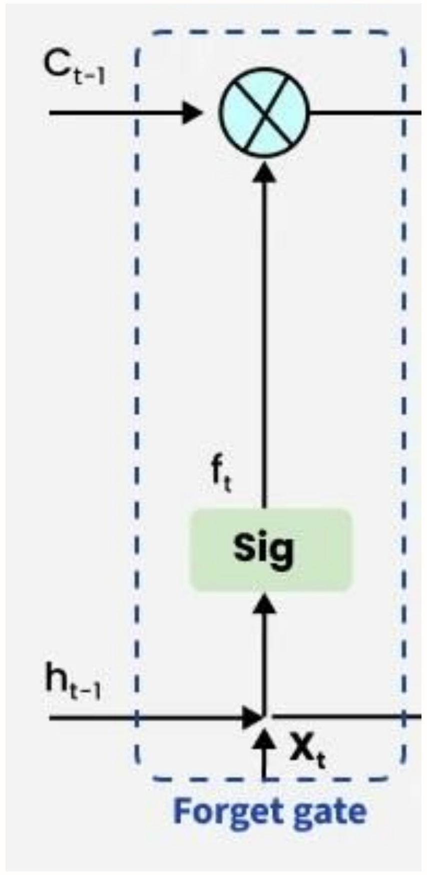
Input Gate

- Controls what new information will be added to the cell state.
- A sigmoid function selects which values to update, and a tanh function creates candidate values. These are combined to update the state.



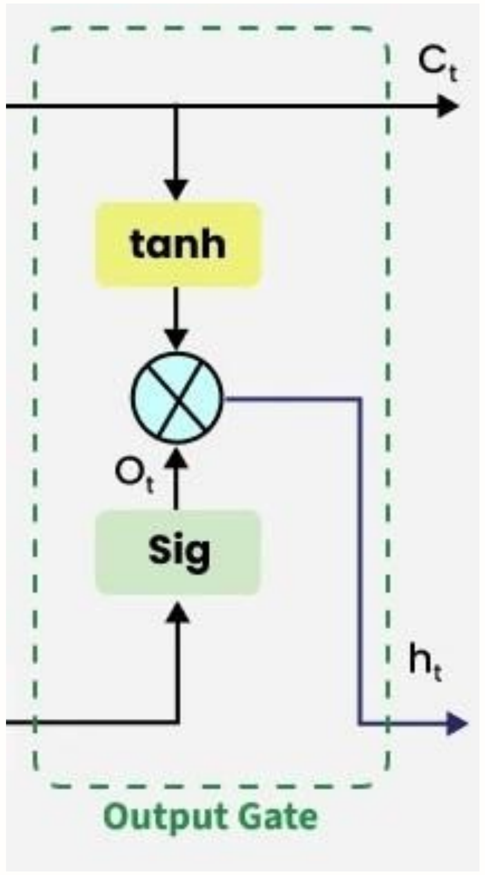
Forget Gate

- Determines which information from the previous cell state should be discarded.
- It uses a sigmoid function on the previous hidden state and current input, producing values between 0 (forget) and 1 (retain).



Output Gate

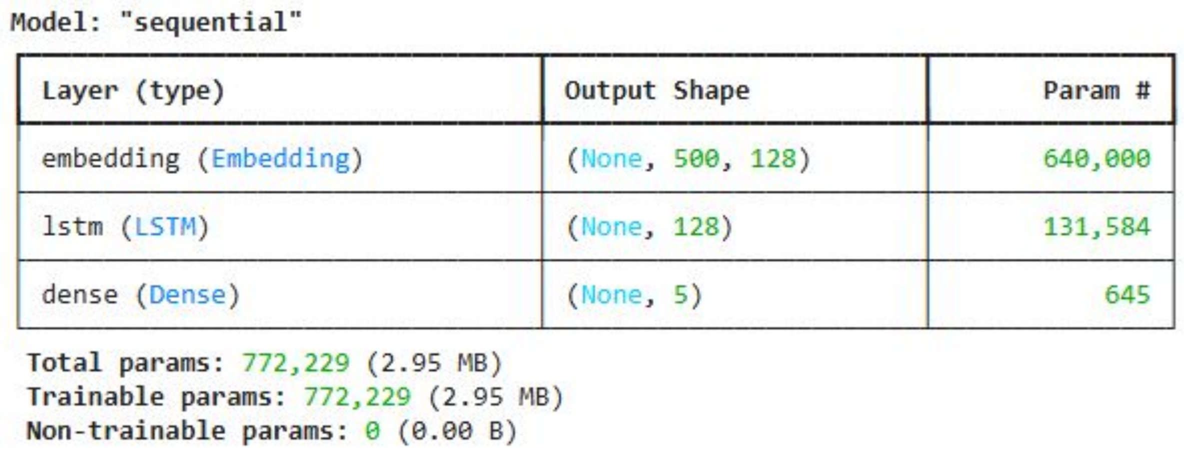
- Decides the next hidden state. A sigmoid function evaluates the importance of the current input and previous hidden state.
- The updated cell state is passed through a tanh function and multiplied with the sigmoid output to produce the new hidden state.



Build LSTM Architecture using Keras

```
# Define the LSTM model
model = Sequential()
model.add(Input(shape=(max_sequence_length,)))
model.add(Embedding(input_dim=5000, output_dim=128))
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(len(le.classes_), activation='softmax'))  # Output layer with softmax for multi-class

# Compile the model
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
```



```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 500, 128)	640,000
lstm (LSTM)	(None, 128)	131,584
dense (Dense)	(None, 5)	645

```
Total params: 772,229 (2.95 MB)
Trainable params: 772,229 (2.95 MB)
Non-trainable params: 0 (0.00 B)
```

RNN vs LSTM

Aspect	RNN (Recurrent Neural Network)	LSTM (Long Short-Term Memory)
Basic Structure	Simple recurrent cell with one hidden state	More complex cell with **input, forget, and output gates**
Long-Term Memory	Poor – struggles with long-term dependencies	Strong – designed to retain long-term information
Vanishing Gradient Issue	Common, especially with long sequences	Mitigated due to gating mechanisms
Computational Complexity	Lower – simpler and faster to train	Higher – more parameters and longer training time
Accuracy on Long Sequences	Lower – tends to forget earlier context	Higher – capable of learning long dependencies
When to Use	Suitable for short and simple sequences	Suitable for complex and long sequences

Deep Learning

Transfer Learning

What is Transfer Learning?

Transfer Learning is a technique where a pre-trained model—trained on a large dataset—is reused to solve a different but related task.

It is especially useful in deep learning tasks like computer vision when:

- You have limited data
- You lack high computational power

Instead of training a complex neural network from scratch (which may require weeks and massive data), you fine-tune a model that has already learned general features from millions of images. This reduces training time, minimizes overfitting, and improves performance on small datasets.

Popular Pre-trained Models for Transfer Learning

- **VGG-16**
  - ~138 million parameters
  - Known for its simplicity and uniform architecture
  - Easy to understand and modify
- **ResNet (Residual Network)**
  - Introduces skip connections to prevent vanishing gradients
  - Enables training of very deep networks without performance degradation
- **MobileNet**
  - Lightweight and efficient
  - Optimized for mobile and embedded vision applications
  - Low computational cost at inference
- **EfficientNet**
  - Balances model depth, width, and resolution
  - Scalable for different computational resources
  - High accuracy with fewer parameters

Conclusion

- Convolutional layers and pooling layers aim to extract information from image input.
- Fully connected layers aim to carry out the main function as a classifier or regressor that predicts output values.
- RNN can be used for sequential data.
- LSTM provides a solution for better performance with long word inputs.
- Existing architectures can be utilized in various ways, adopting architectures for other cases, maintaining the backbone (feature extraction layer) and only replacing the last layer, or even using its weights as initial weights when training new data.

Hands On Transfer Learning with Python

- Choose a Pre-trained Model

  Load the ResNet50 model trained on ImageNet and exclude its top layers:

```
from keras.applications.resnet50 import ResNet50

base_model = ResNet50(weights='imagenet', include_top=False)
```

- Add Custom Classification Layers

  Append new layers tailored to your task (e.g., binary classification):

```
from keras.layers import GlobalAveragePooling2D, Dense

from keras.models import Model

x = base_model.output

x = GlobalAveragePooling2D()(x)

x = Dense(1024, activation='relu')(x)

predictions = Dense(1, activation='sigmoid')(x)

model = Model(inputs=base_model.input, outputs=predictions)
```

Hands On Transfer Learning with Python (cont'd)

- Freeze the Base Model Layers

  Prevent ResNet50's pre-trained weights from updating during initial training:

```
for layer in base_model.layers:
    layer.trainable = False
```

- Prepare Your Dataset

  Use ***ImageDataGenerator*** or similar tools to resize and normalize your input images consistently.

- Train the New Layers (Feature Extraction)

  Train only the added fully connected layers with the base model frozen.

- Optional: Fine-tune the Model

  Unfreeze some top layers of ResNet50 and continue training with a low learning rate:

  - Useful when your task differs significantly from the original ImageNet task.
  - Helps refine higher-level feature representations.

THANK YOU