Universidad Intercontinental de la Empresa

Asignatura

Advanced Machine Learning

Profesor

Alejandro Esteban Martínez

Sesión 03

Unidad I

Deep Learning

Tema 3

Convolutional and Recurrent Neural Networks

Deep Learning

Session 02. Context

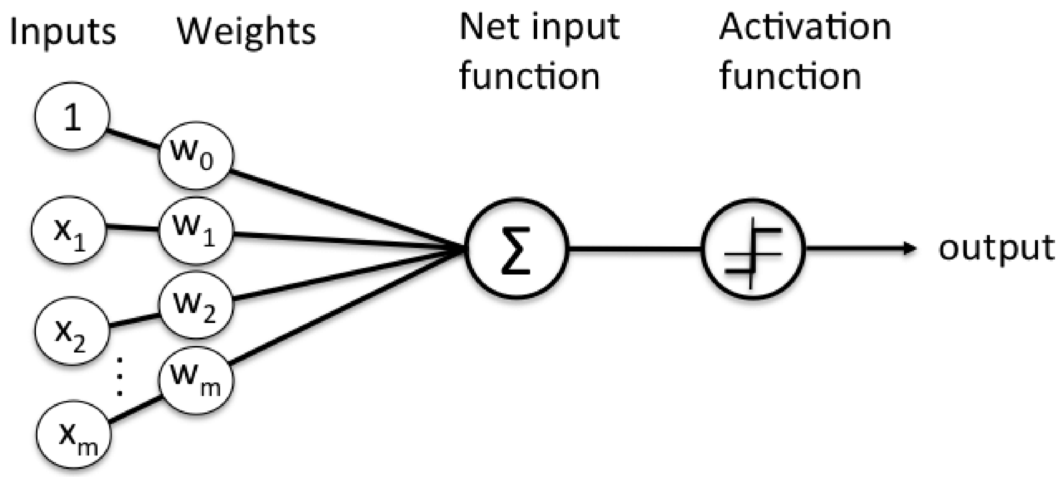
S04 **13/02/2025**	**Unit I Deep Learning** 1.3 Convolutional Neural Networks 1.4 Recurrent Neural Networks	LO01 LO02 LO06 LO07	Review contents of Session4.pdf	Active Exposition Session4.pdf Practical activity: Start assignment A05	Asignment 1 A05 (Delivery date S05 10:00)
	Form 01 for Quality Management		Fill out the form for quality management.		**AE06:** **Start Date 10/02/2025, Deadline 16/02/2025**

At last class we saw…

Deep Learning

What is a perceptron?

It is the building block of every deep learning model today (Basically what we call a neuron).
Every model has thousands of them and even models like GPT-3.5 are estimated to have around 60B neurons.



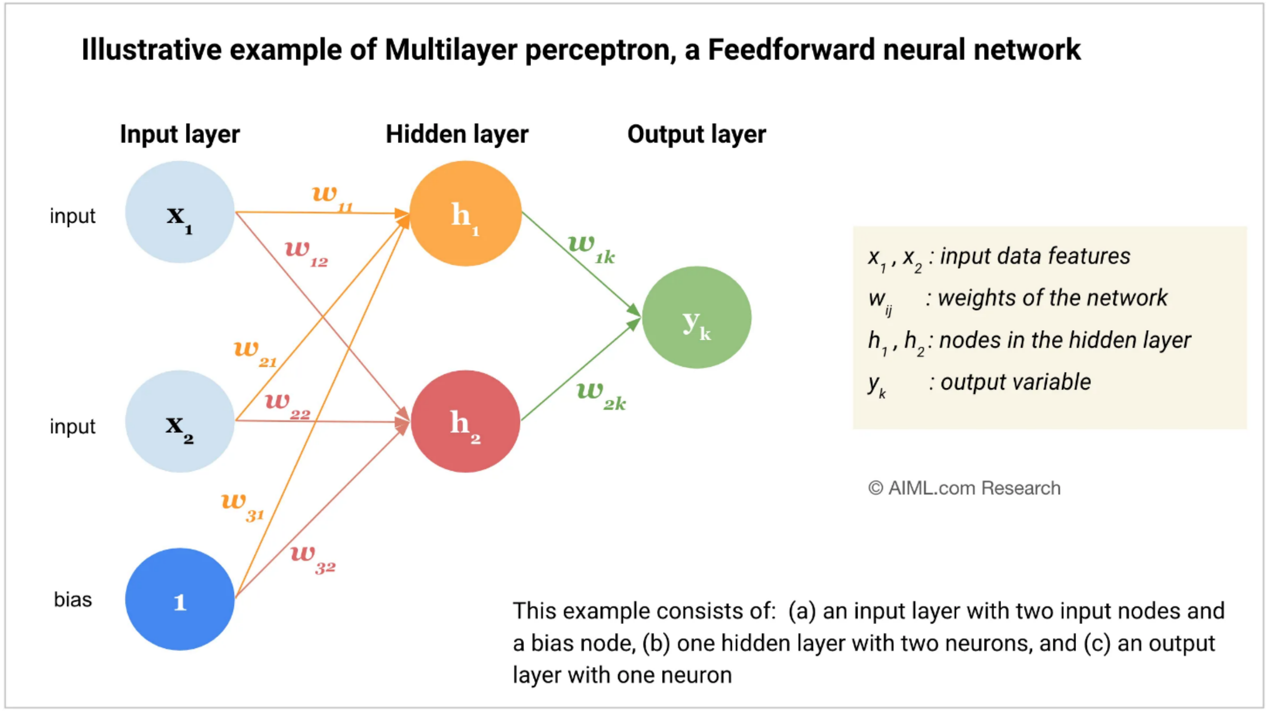
$$y = f(i = \sum_{i=1}^{n} w_i \cdot x_i + b)$$

Perceptron Function

Deep Learning

What is the Multilayer Perceptron?

As the name says, it is just a Perceptron that has multiple layers, but each layer actually works as the output of the previous and the input of the next one.
It is formed as multiple Perceptrons all densely connected with each other into multiple layers.



Illustrative example of Multilayer perceptron, a Feedforward neural network

x_1, x_2 : input data features
w_{ij} : weights of the network
h_1, h_2: nodes in the hidden layer
y_k : output variable

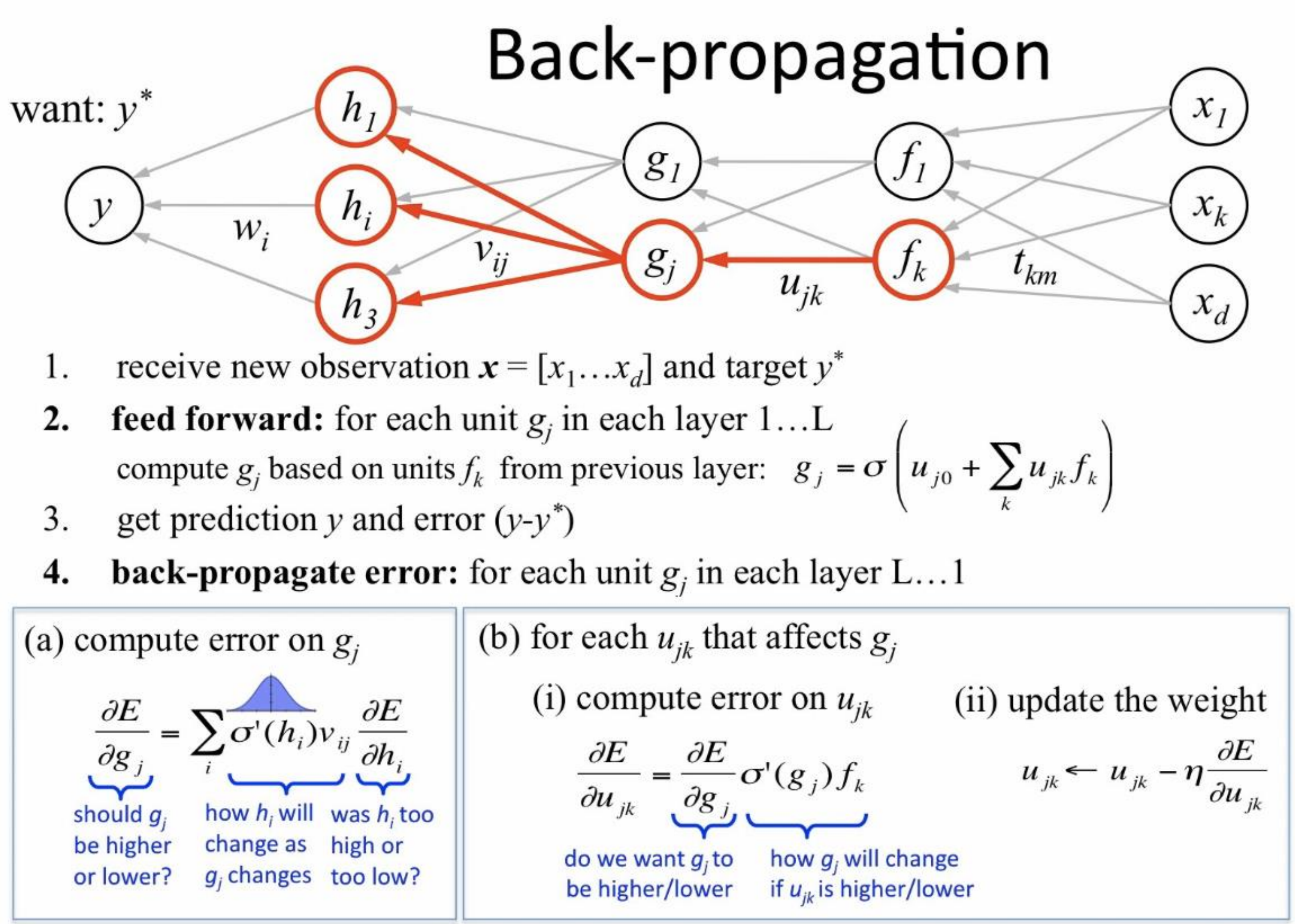
© AIML.com Research

This example consists of: (a) an input layer with two input nodes and a bias node, (b) one hidden layer with two neurons, and (c) an output layer with one neuron

Deep Learning

And how do we train it?

Ok, not like that because this is an applied degree, but I do want you to understand why we need it and how we use it



Back-propagation

1. receive new observation $\boldsymbol{x} = [x_1 \dots x_d]$ and target y^*
2. **feed forward:** for each unit g_j in each layer $1 \dots L$
   compute g_j based on units f_k from previous layer: $g_j = \sigma\left(u_{j0} + \sum_k u_{jk} f_k\right)$
3. get prediction y and error $(y\text{-}y^*)$
4. **back-propagate error:** for each unit g_j in each layer $L \dots 1$

(a) compute error on g_j

$$\underbrace{\frac{\partial E}{\partial g_j}}_{\text{should } g_j \text{ be higher or lower?}} = \sum_i \underbrace{\sigma'(h_i) v_{ij}}_{\text{how } h_i \text{ will change as } g_j \text{ changes}} \underbrace{\frac{\partial E}{\partial h_i}}_{\text{was } h_i \text{ too high or too low?}}$$

(b) for each u_{jk} that affects g_j

(i) compute error on u_{jk}

$$\frac{\partial E}{\partial u_{jk}} = \underbrace{\frac{\partial E}{\partial g_j}}_{\text{do we want } g_j \text{ to be higher/lower}} \underbrace{\sigma'(g_j) f_k}_{\text{how } g_j \text{ will change if } u_{jk} \text{ is higher/lower}}$$

(ii) update the weight

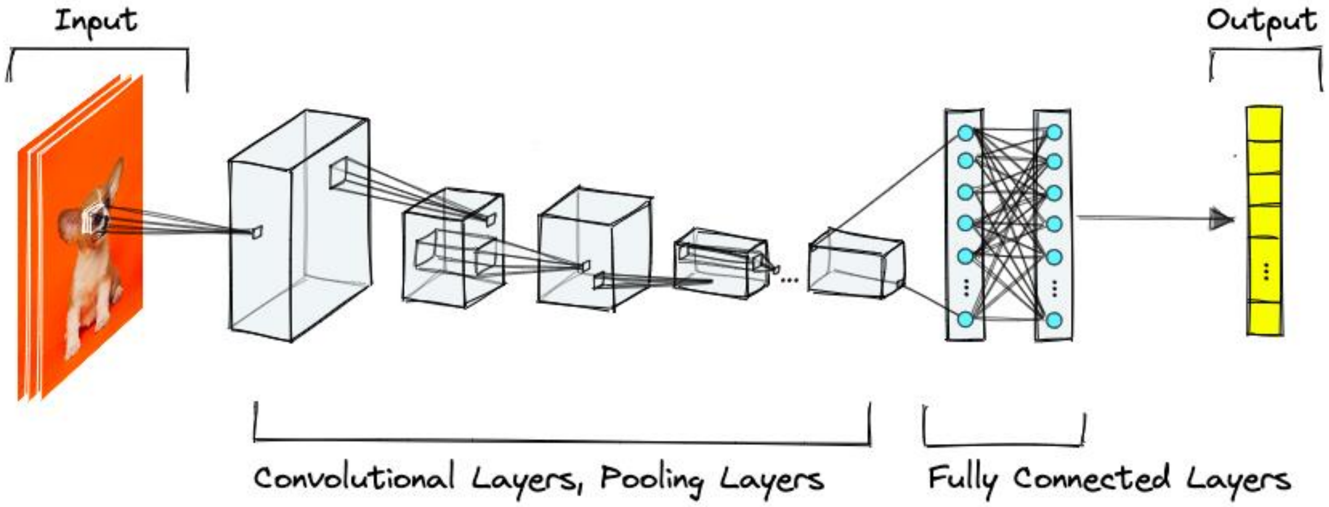
$$u_{jk} \leftarrow u_{jk} - \eta \frac{\partial E}{\partial u_{jk}}$$

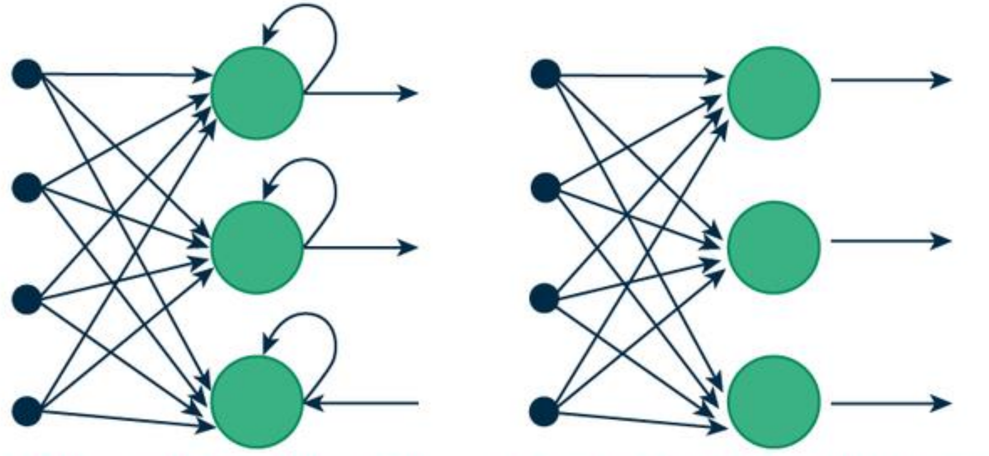
Today we will talk about…

Deep Learning

Convolutional Neural Networks & Recurrent Neural Networks

- Convolutional Neural Networks & Recurrent Neural Networks
  - What are they?
  - How do they work?
  - Implementation
  - Famous Architectures and Examples





(a) Recurrent Neural Network

(b) Feed-Forward Neural Network

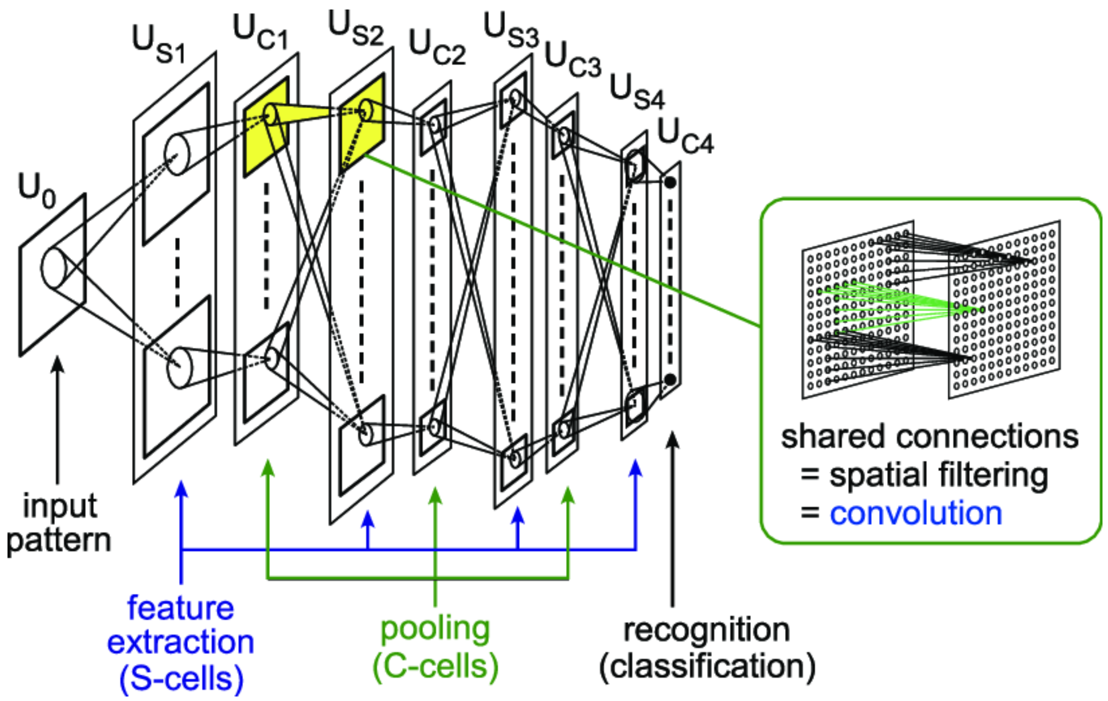
Deep Learning

Convolutional Neural Networks – First a bit of History

Neocognitron (1979)

It was used for Japanese handwritten character recognition and inspired CNNs.

It was inspired by the visual nervous system of vertebrates



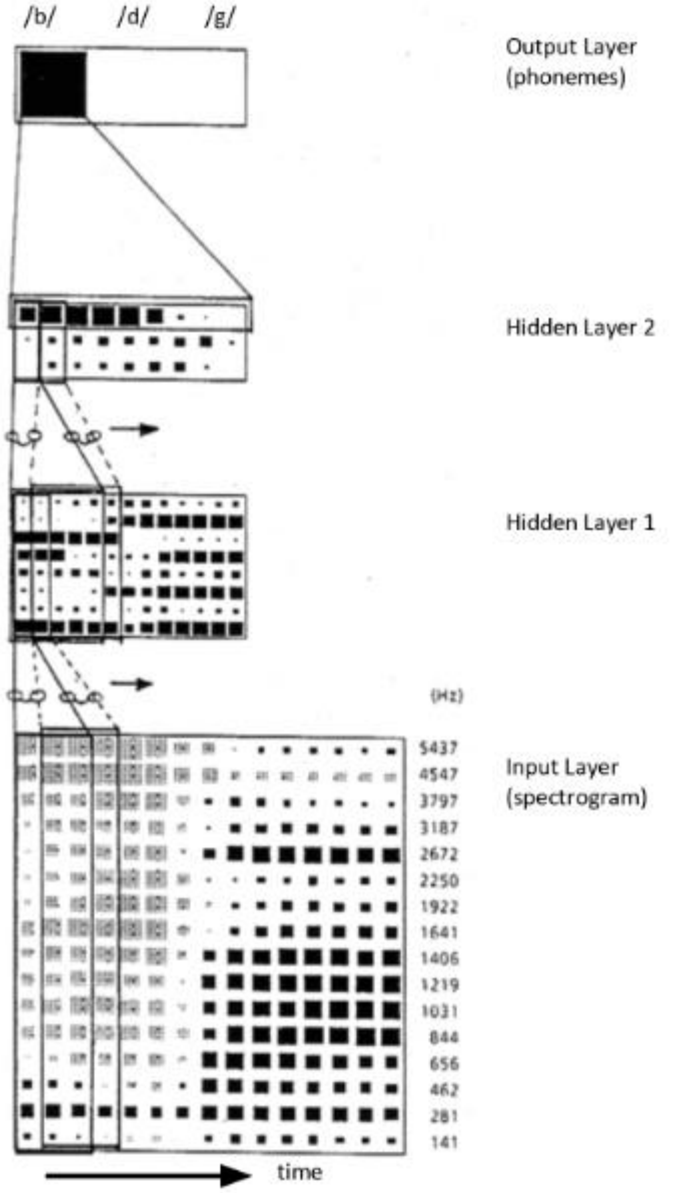
U_{S1}
U_{C1}
U_{S2}
U_{C2}
U_{S3}
U_{C3}
U_{S4}
U_{C4}
U_0
input pattern
feature extraction (S-cells)
pooling (C-cells)
recognition (classification)
shared connections = spatial filtering = convolution

Deep Learning

Convolutional Neural Networks – First a bit of History

Time Delay Neural Network (1987)

One of the first recognized examples of CNNs in use, it was used for speech recognition and the convolution was performed along the temporal dimension in a spectrogram.

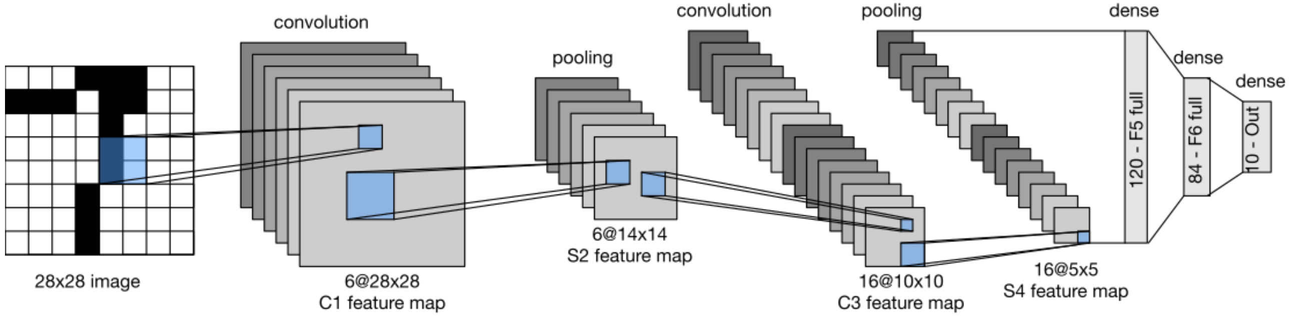


Deep Learning

Convolutional Neural Networks – First a bit of History

LeNet-5 (1989-1998)

The fifth (1995) and best-known version of the LeNet model. It was used for handwritten digit recognition and stablished the standard for CNNs architecture. It was used for reading millions of checks per day in the American banks since 1996



Deep Learning

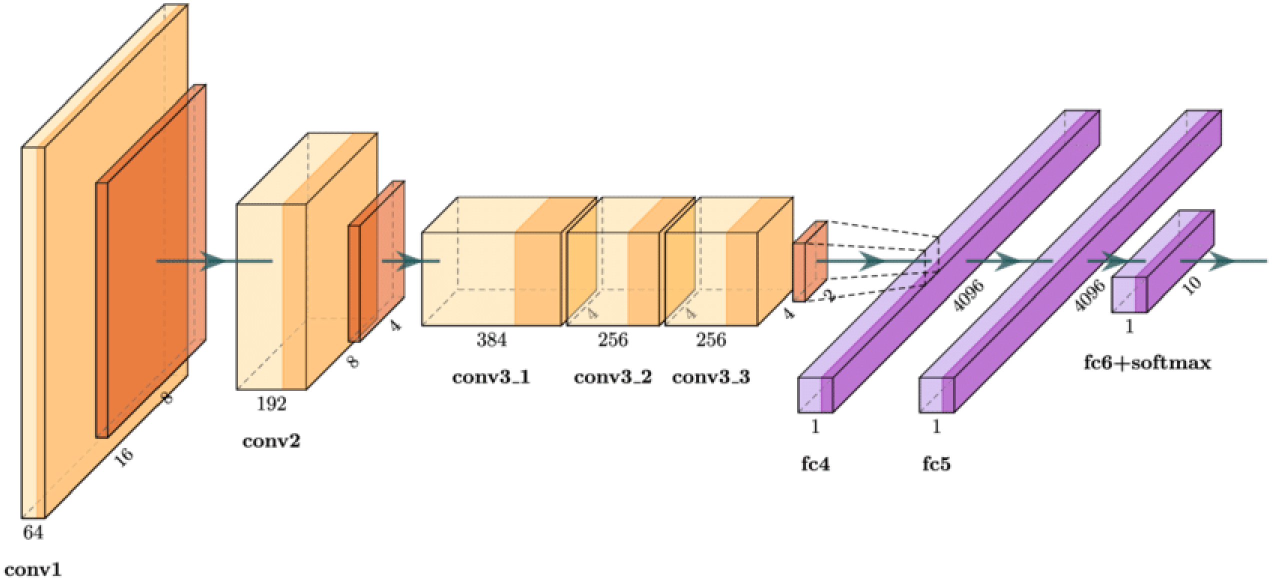
Convolutional Neural Networks – First a bit of History

AlexNet (2012)

It was the first CNN-based model to win the ImageNet competition in 2012 with more than 14M hand classified pictures and more than 20000 different classes.

Since then, this yearly competition has been won by CNN models.

It has 5 Convolutional Layers followed by 3 fully connected layers that classify the resulting extracted features and was trained on GPU.

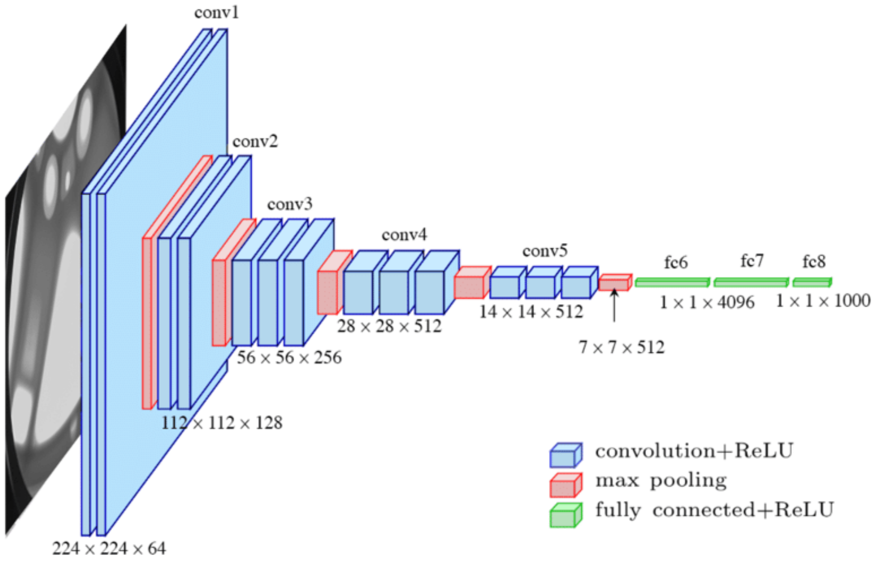


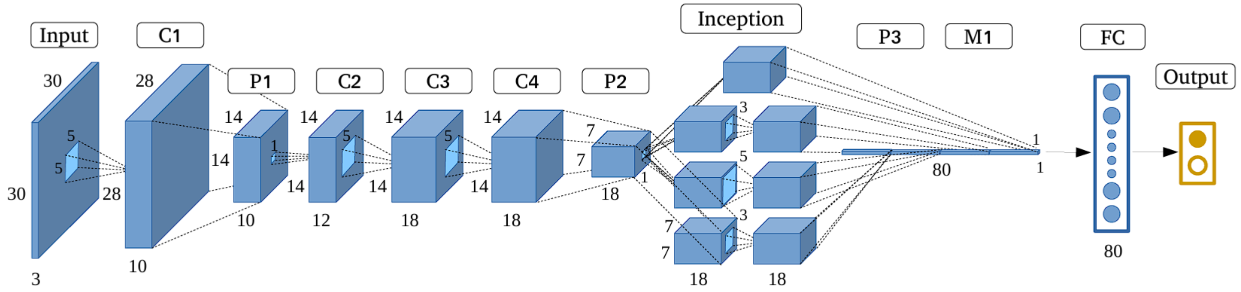
Deep Learning

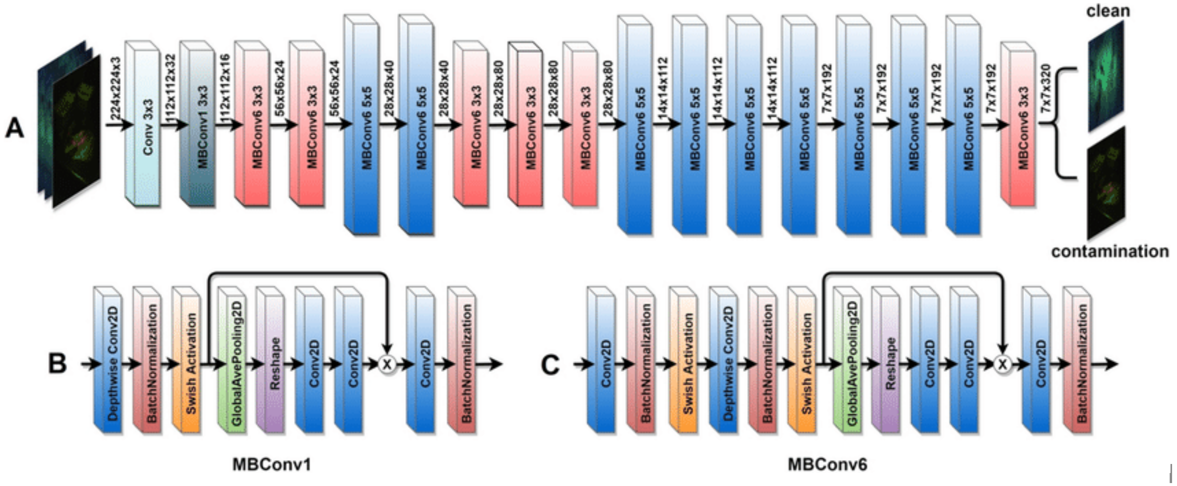
Convolutional Neural Networks – First a bit of History

VGGNet, GoogleNet, EfficientNet… (2013-2017)

After AlexNet, Computer Vision was dominated by CNN based models, increasing the number of layers and complexity while also focusing on efficient use of resources (EfficientNet and MobileNet are great examples of this)





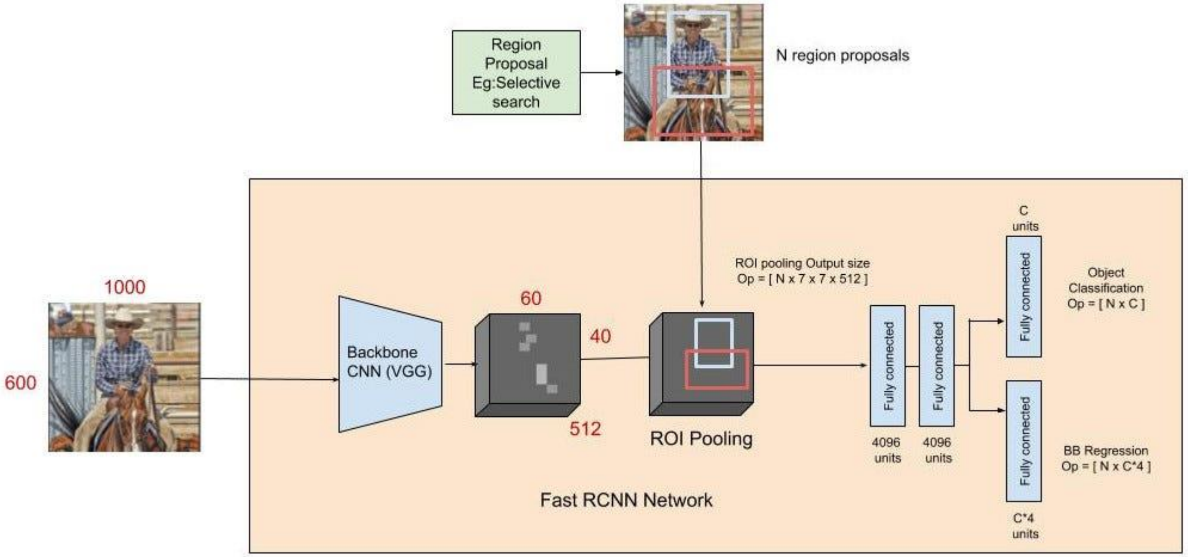


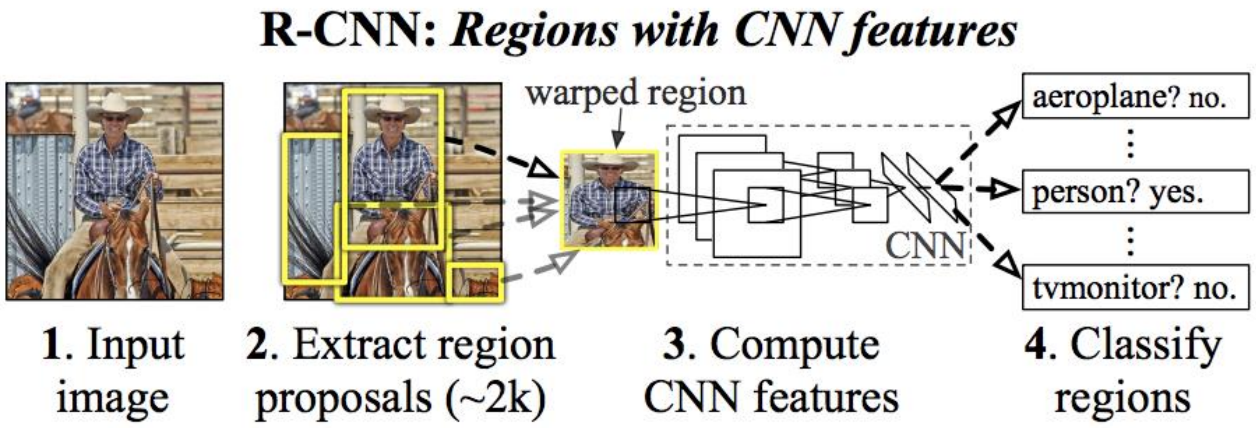
Deep Learning

Convolutional Neural Networks – First a bit of History

Region-based CNN (2013 - 2019)

CNN weren’t only used for image classification, nowadays we think about Computer Vision and we think about **object detection**, and most of it started with R-CNNs, later Fast R-CNN, Faster R-CNN, Mask R-CNN…





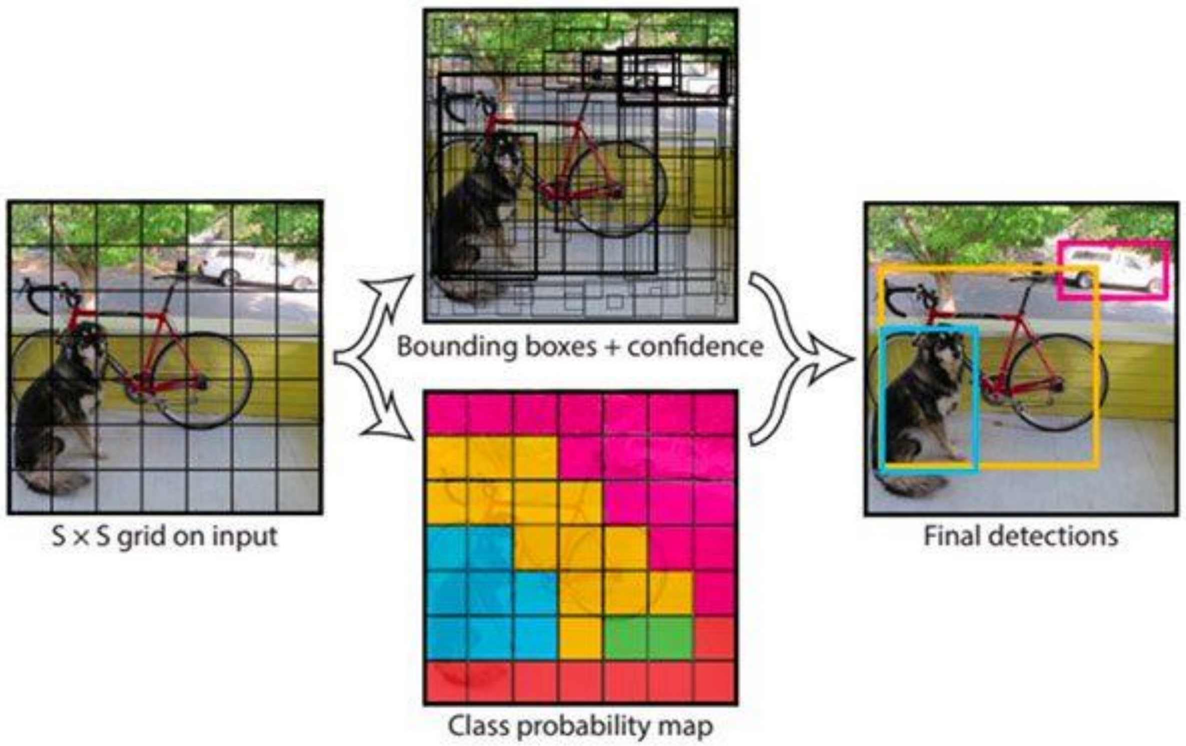
Deep Learning

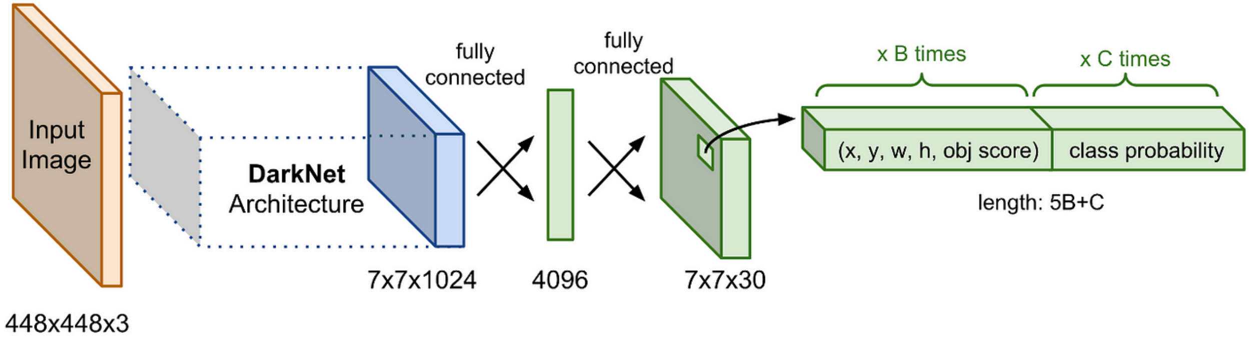
Convolutional Neural Networks – First a bit of History

You Only Look Once(2015 - Present)

Nowadays, the field of object detection is completely monopolized by YOLO algorithms, at least in real-time detection.
It processes an image in a single pass through a neural network, making it much faster than traditional methods like R-CNNs.
The last version available is YOLOv11 but there are new versions every couple of months that improve on the previous.

The trained weights are available online and it is possible to choose from different sizes, from a worst precision but better performance to really potent models that have impressive metrics but aren't suitable for real-time usage on most devices



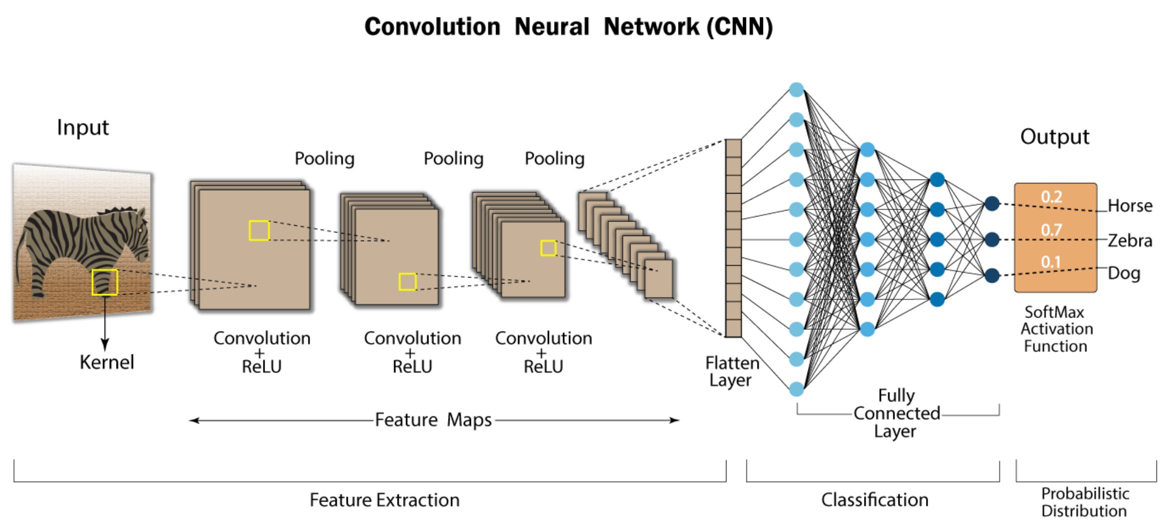


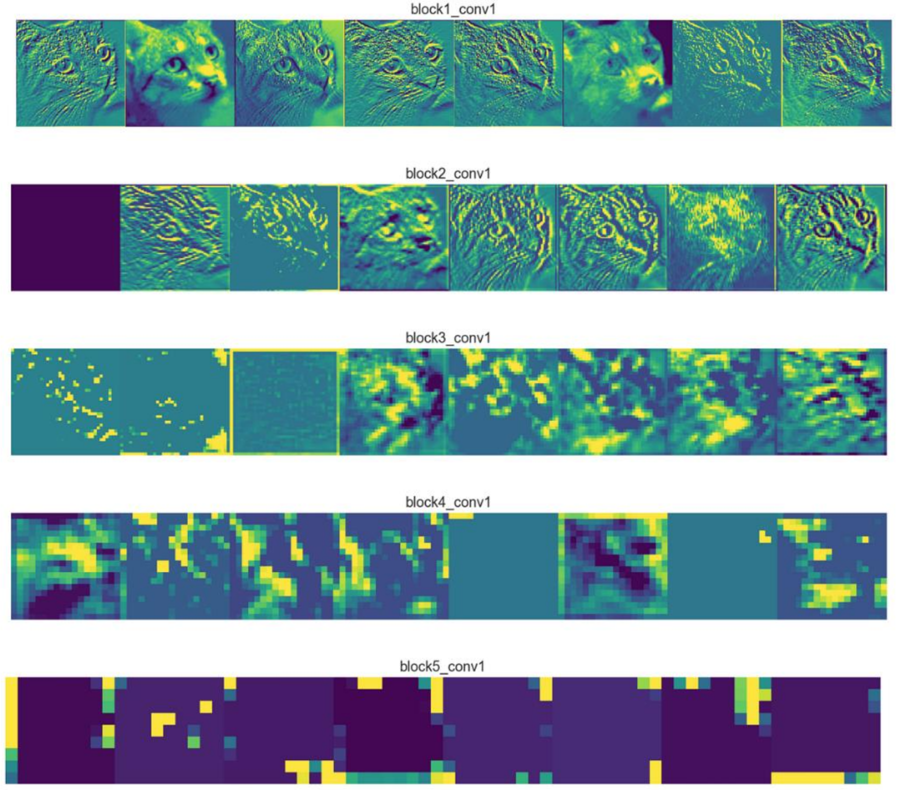
Deep Learning

Convolutional Neural Networks

A **Convolutional Neural Network (CNN)** is a specialized type of **deep learning model** designed to process **grid-like data** (e.g., images, videos, audio spectrograms) by automatically learning **spatial hierarchies of features**. Basically, it learns to "see" by breaking images into smaller features and assembling them hierarchically.

Importantly, it preserves spatial information in our data, so we can recognize patterns even if the image has been transformed (mirrored, transposed, rotated…) and it is also far more efficient at image tasks, as we will see that CNNs use less parameters than MLPs.



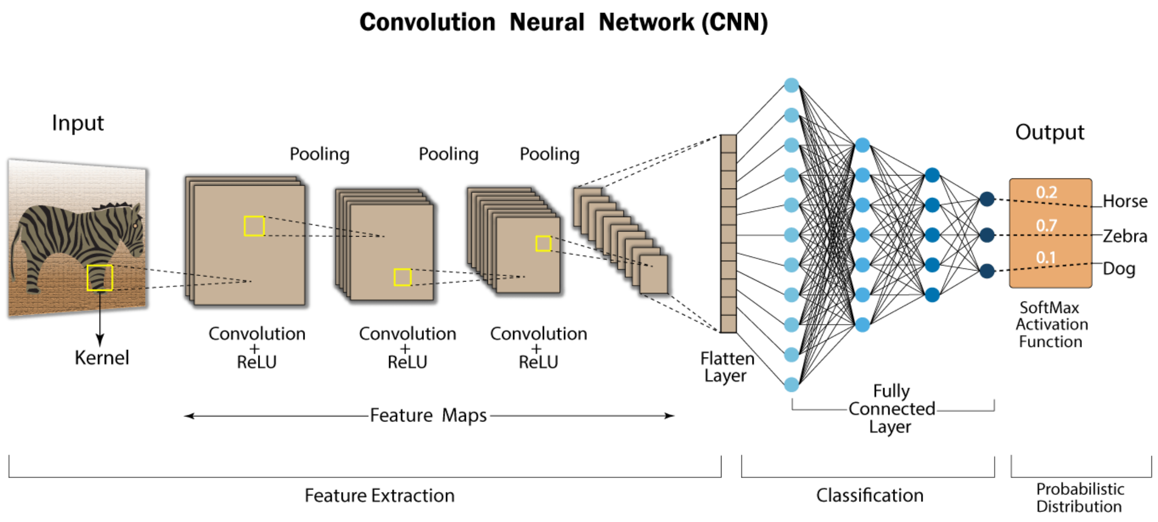


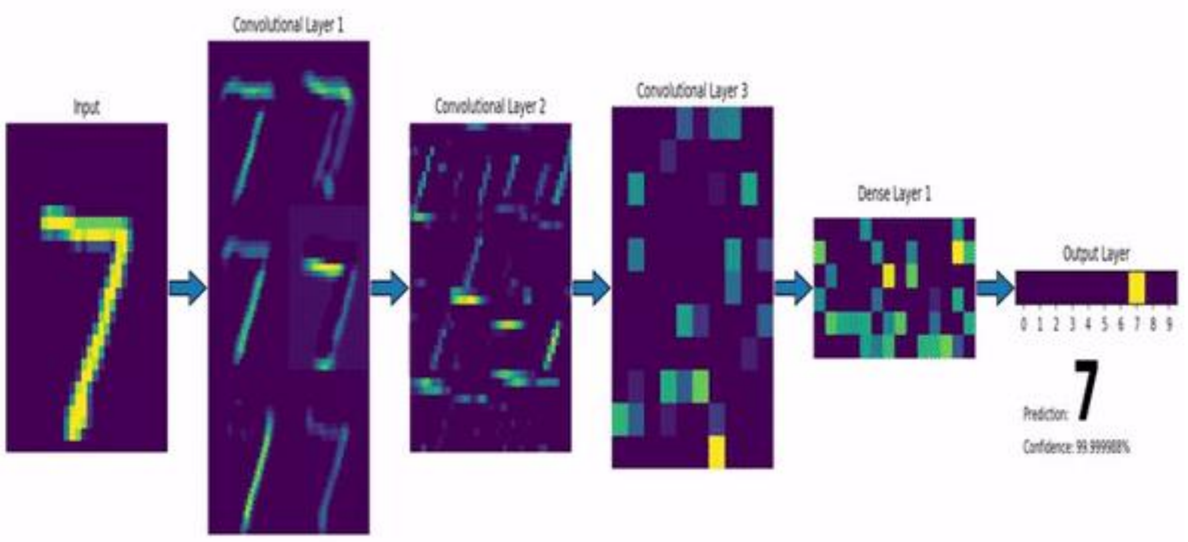
Deep Learning

Convolutional Neural Networks – How do they work?

A CNN is usually formed by 3 main components: Convolutional Layers, Pooling Layers and Fully Connected Layers.

- Convolutional Layer: It extracts the spatial features of the image applying filters hierarchically combined with non-linear activation functions (usually ReLU)
- Pooling Layer: It reduces spatial size (downsamples the extracted feature map) to prevent overfitting, reduce computation requirements and focus only on the most important features.
- Fully Connected Layer: An initial layer that flattens the resulting 2D feature maps into a 1D vector as input for an MLP that does the classification with the extracted features





Deep Learning

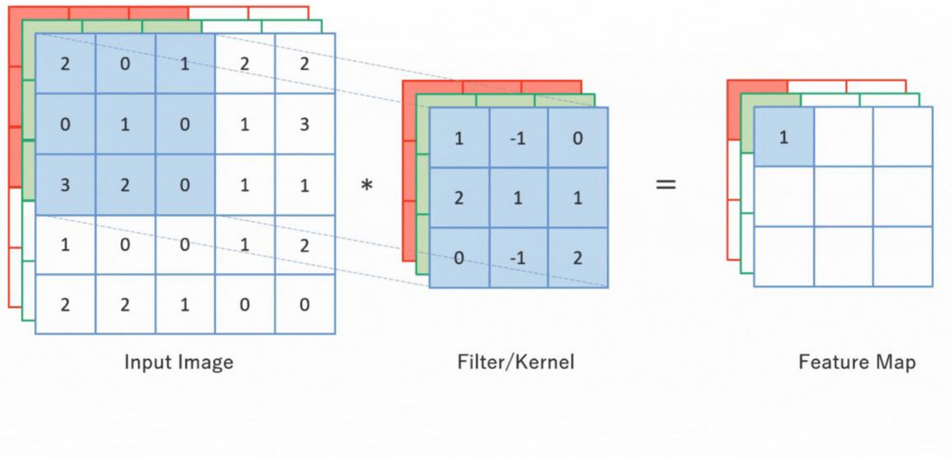
Convolutional Layer

This layer is composed by multiple trainable filters (also known as kernels). In the shown example we have a 3x3x3 filter, as it would be for a color image.

The filter is composed with a series of trainable weights, that will learn through back-propagation, how to better extract the features in each filter pass.

This filter is then slid (convolved) through the input image, extracting a feature map of the image by computer the dot product of the Filter weight matrix with the selected region of the image matrix.

From the filter we can change its dimensions, the stride (How many pixels it moves to the right each time the filter is applied, if more than one, down-sampling occurs) and zero-padding, to have an NxN image after applying the filter.



Input Image

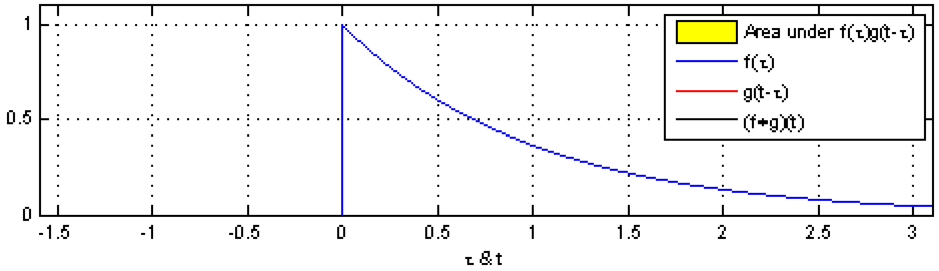
2	0	1	2	2
0	1	0	1	3
3	2	0	1	1
1	0	0	1	2
2	2	1	0	0

Filter/Kernel

1	-1	0
2	1	1
0	-1	2

Feature Map

1		



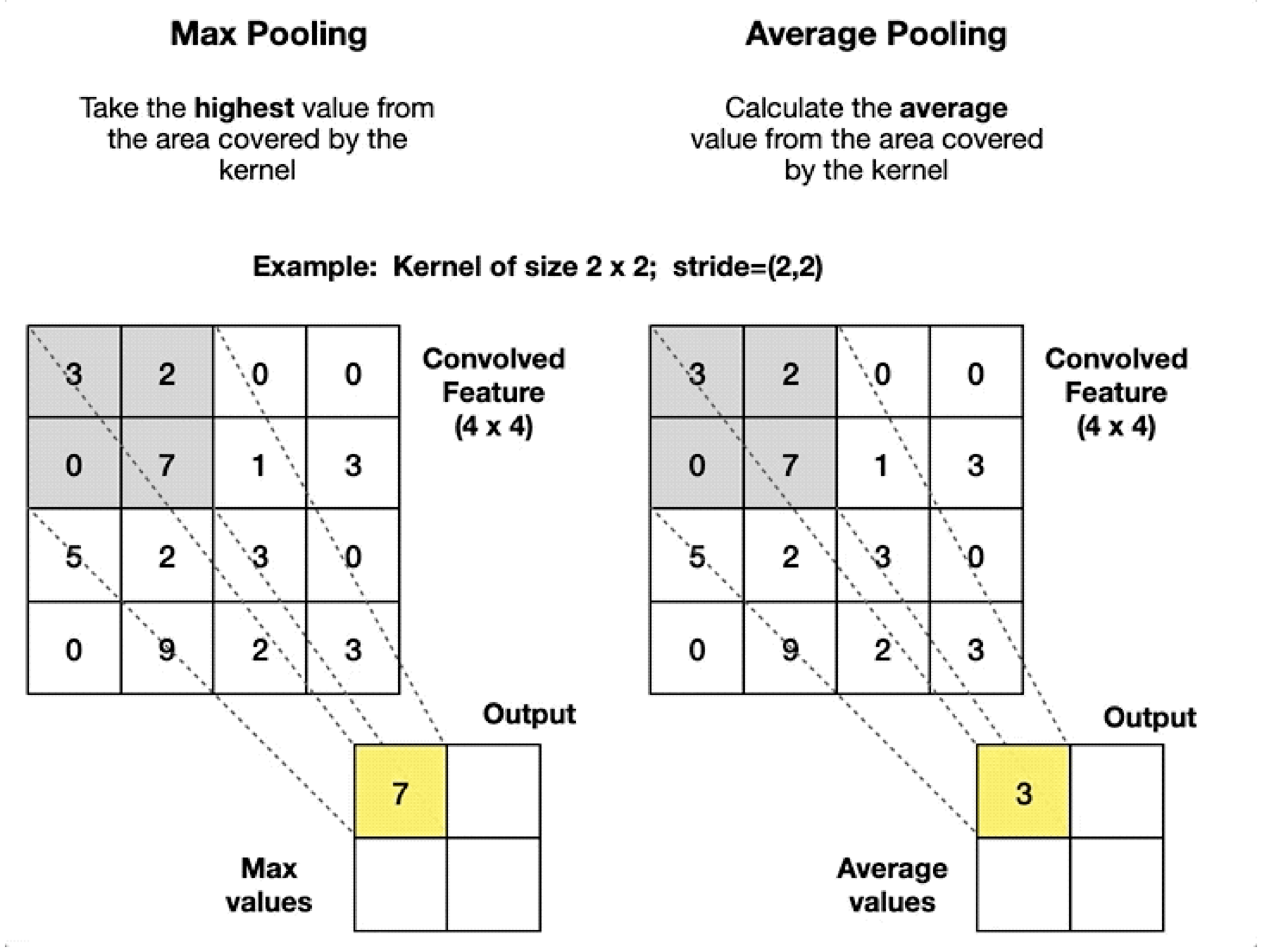
Deep Learning

Pooling Layer

In a CNN typical architecture, normally a Pooling Layer will be set in between each Convolutional Layer **to reduce the number of parameters in the network**, preventing overfitting and reducing the computational effort (Similar to the downsampling step in the jpg compression).
In this Layer you can change the dimensions and the stride, but the most typical are 3x3 with a stride of 2 (overlapping pooling) and 2x2 with a stride of 2. In any case with bigger dimensions, too much information could be removed.

The pooling operations applied are normally average pooling or **max pooling**, with the first one being replaced these recent years by max pooling.

Some researchers dislike pooling and believe that we should avoid too much complexity in our models and instead use some Conv Layers with bigger strides in our architecture to down-sample our feature maps.



Max Pooling

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

Average Pooling

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

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

Convolved Feature (4 x 4)

3	2	0	0
0	7	1	3
5	2	3	0
0	9	2	3

Output

7	

Max values

Convolved Feature (4 x 4)

3	2	0	0
0	7	1	3
5	2	3	0
0	9	2	3

Output

3	

Average values

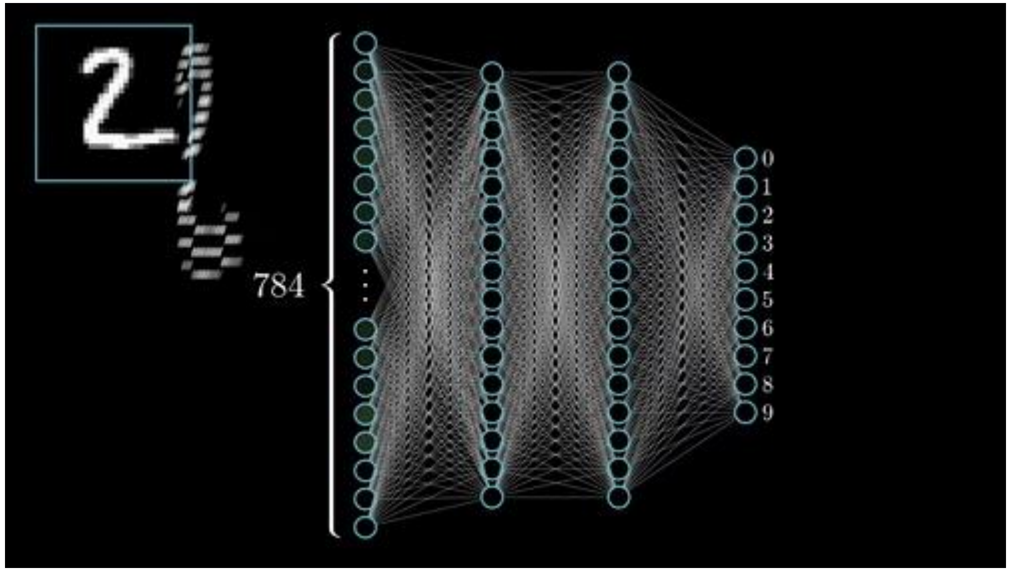
Deep Learning

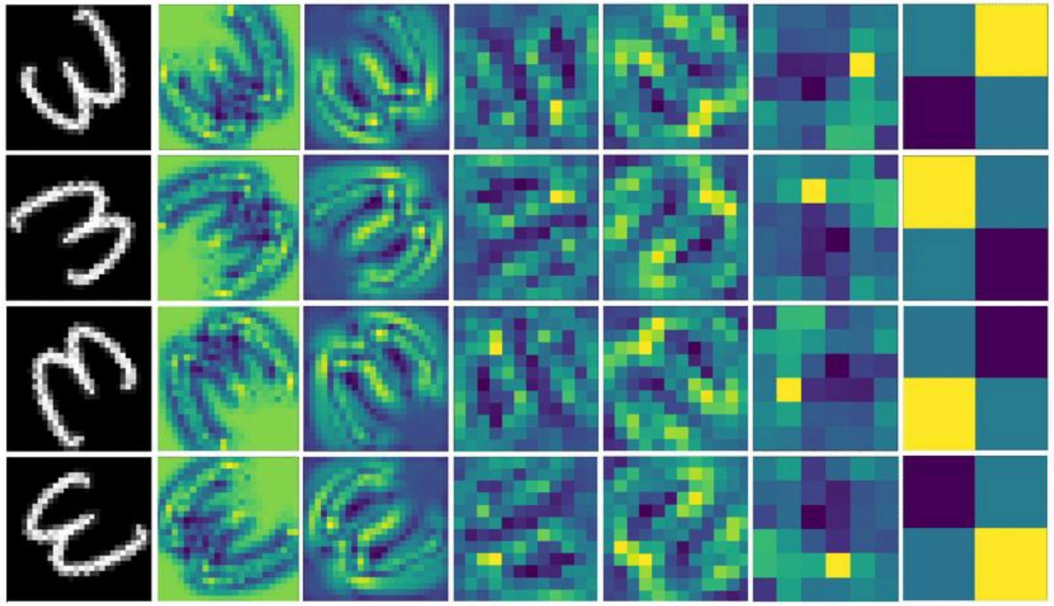
Fully Connected Layer

This last layer will first flatten all the information extracted through the layers of Convolutions and Poolings into a 1D Vector that will be the input of our Fully Connected Neural Network.

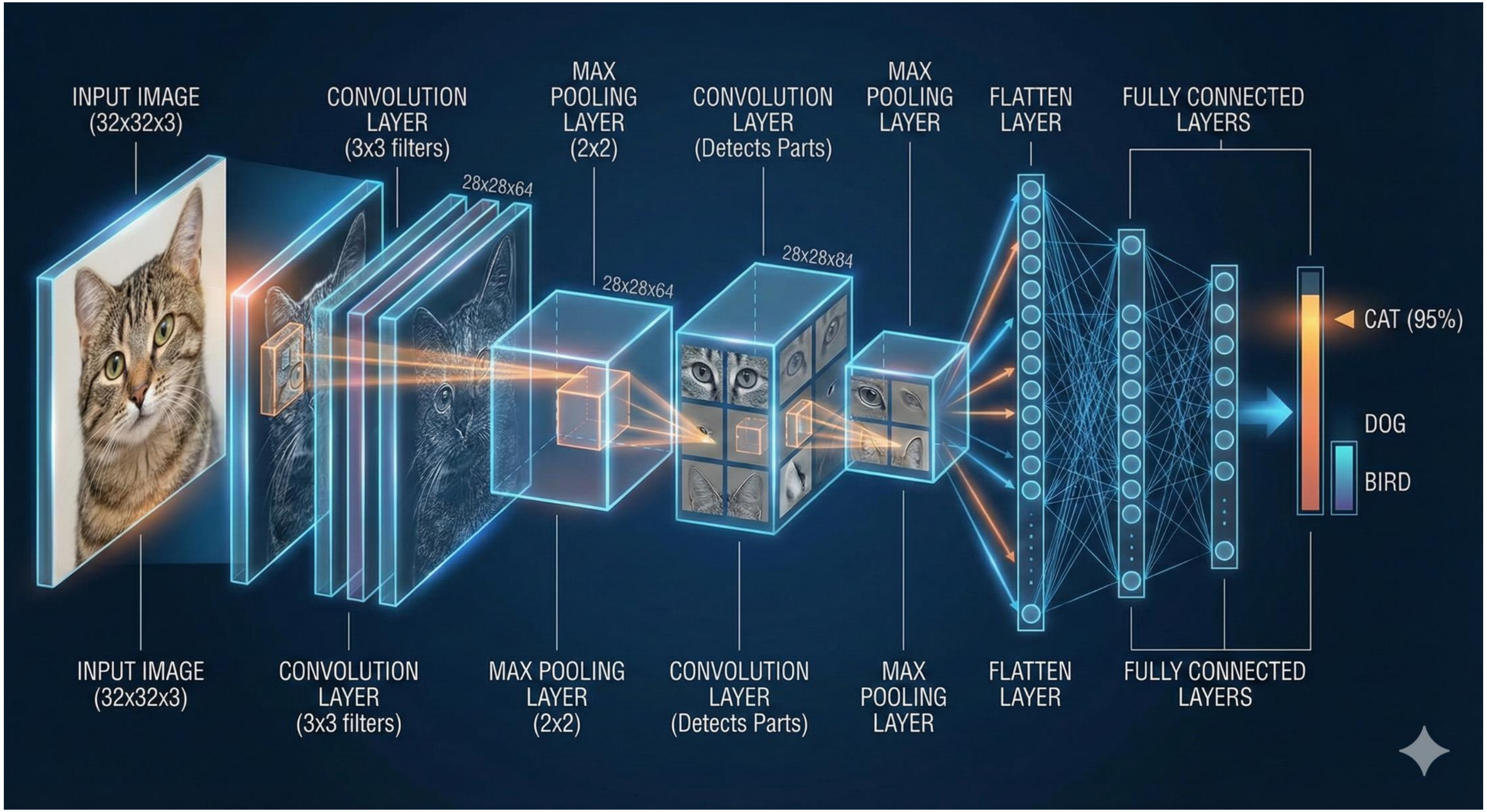
This is trained as a normal Multilayer Perceptron that learns how to classify the images not from the raw pixel data but from learned features in the previous layers.

Finally, a softmax activation function is normally applied for multiclass classification, as it converts our output into a Vector of probabilities that will give us the final probability of our image being classified into each of the classes.





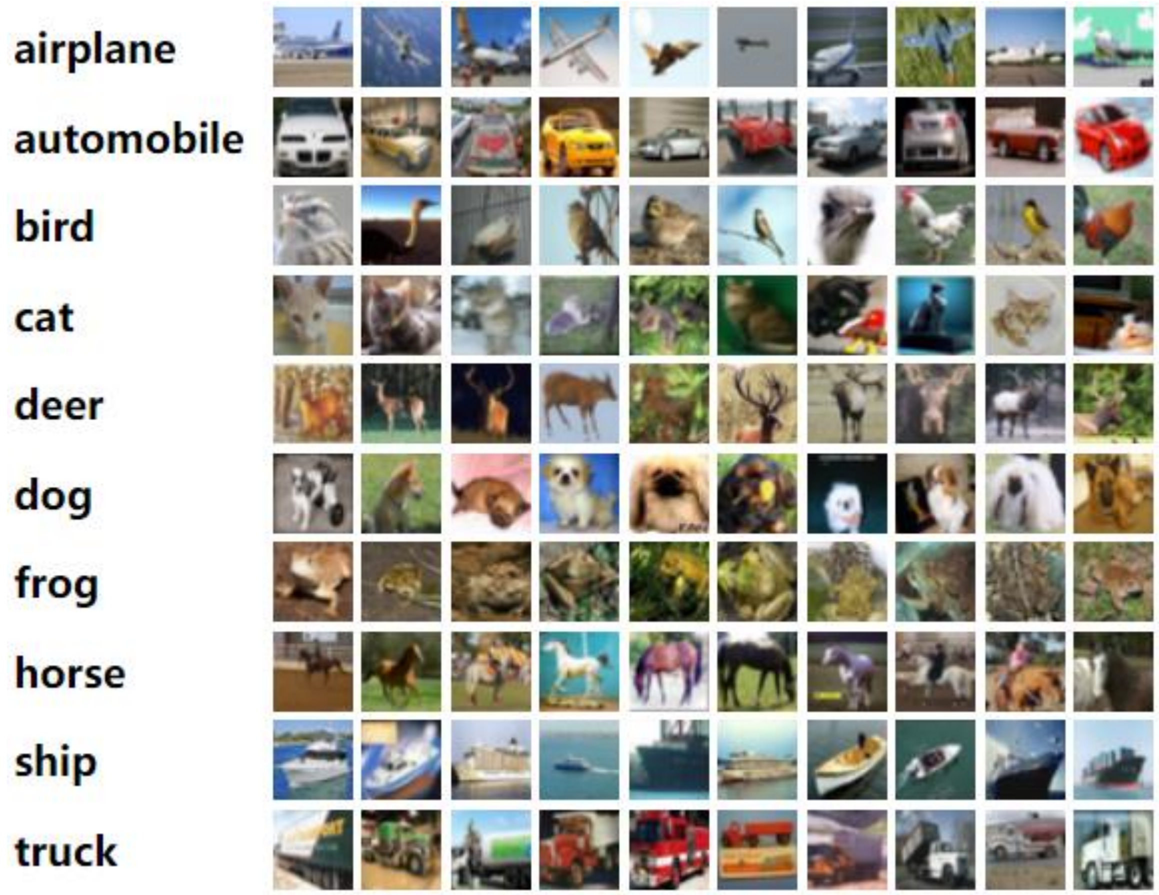
Deep Learning



Deep Learning

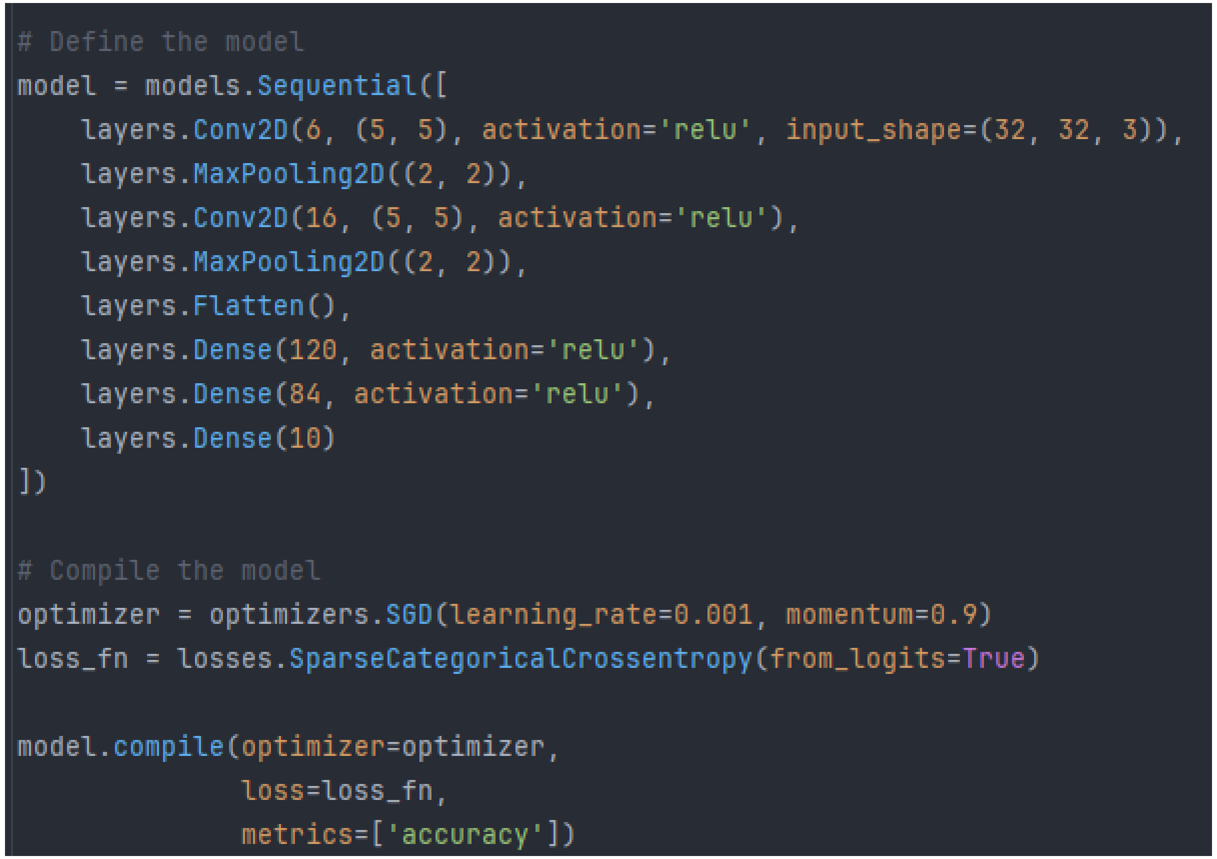
Convolutional Neural Networks – Let’s see the implementation!

For this we will be using CIFAR10, a classic image classification dataset with 60000 32x32x3 (colored) images over 10 different classes.

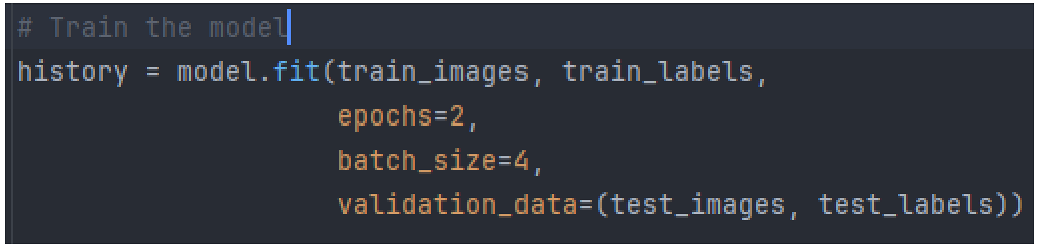


Deep Learning

Convolutional Neural Networks – Let’s see the implementation in Keras



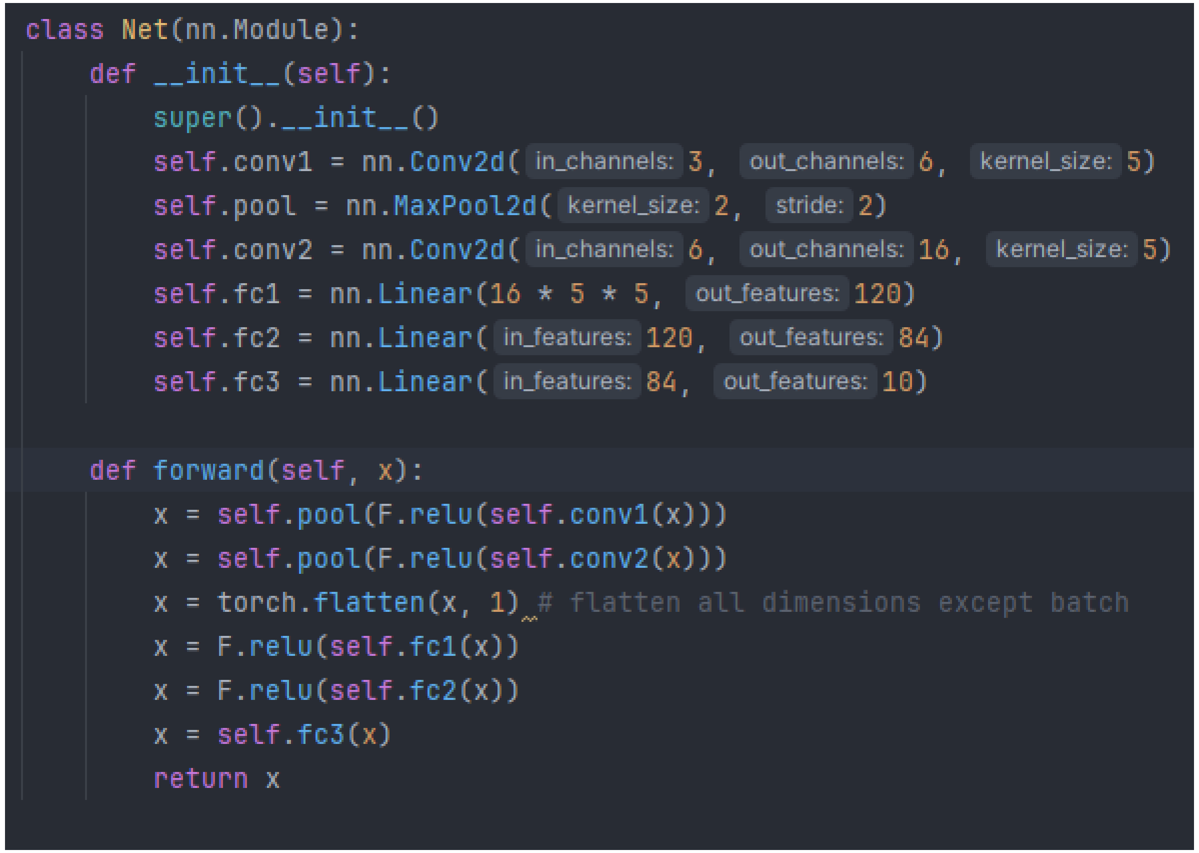
```python
# Define the model
model = models.Sequential([
    layers.Conv2D(6, (5, 5), activation='relu', input_shape=(32, 32, 3)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(16, (5, 5), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(120, activation='relu'),
    layers.Dense(84, activation='relu'),
    layers.Dense(10)
])

# Compile the model
optimizer = optimizers.SGD(learning_rate=0.001, momentum=0.9)
loss_fn = losses.SparseCategoricalCrossentropy(from_logits=True)

model.compile(optimizer=optimizer,
              loss=loss_fn,
              metrics=['accuracy'])
```



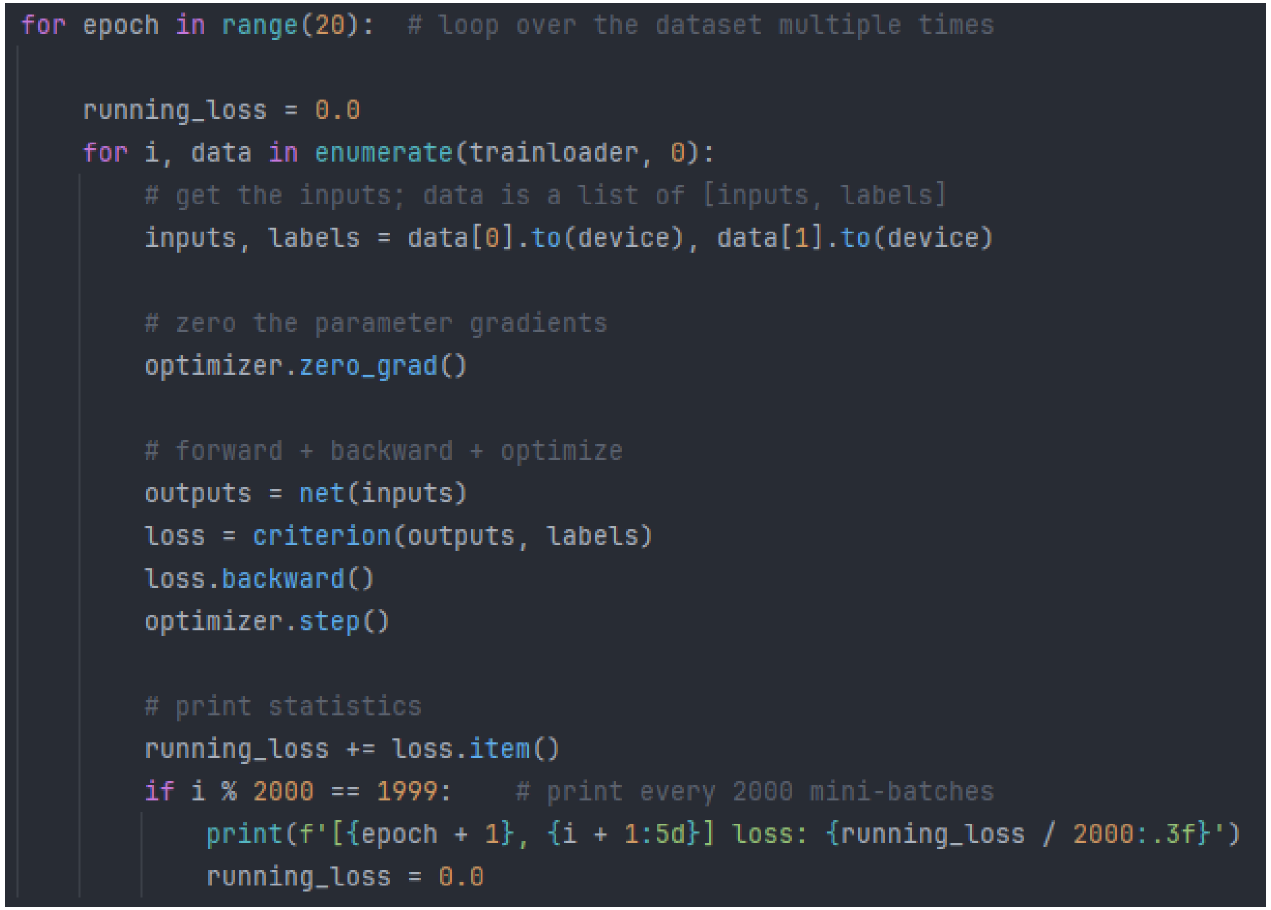
```python
# Train the model
history = model.fit(train_images, train_labels,
                    epochs=2,
                    batch_size=4,
                    validation_data=(test_images, test_labels))
```

Deep Learning

Convolutional Neural Networks – Let’s see the implementation in Pytorch



```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d( in_channels: 3,  out_channels: 6,  kernel_size: 5)
        self.pool = nn.MaxPool2d( kernel_size: 2,  stride: 2)
        self.conv2 = nn.Conv2d( in_channels: 6,  out_channels: 16,  kernel_size: 5)
        self.fc1 = nn.Linear(16 * 5 * 5,  out_features: 120)
        self.fc2 = nn.Linear( in_features: 120,  out_features: 84)
        self.fc3 = nn.Linear( in_features: 84,  out_features: 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```



```
for epoch in range(20):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data[0].to(device), data[1].to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')
            running_loss = 0.0
```

Everything is more complicated but also more customizable!

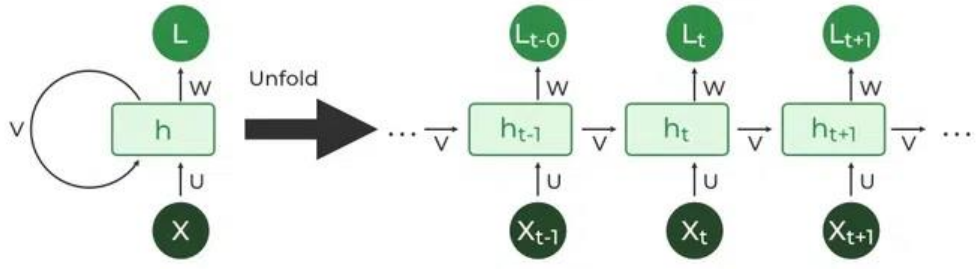
Deep Learning

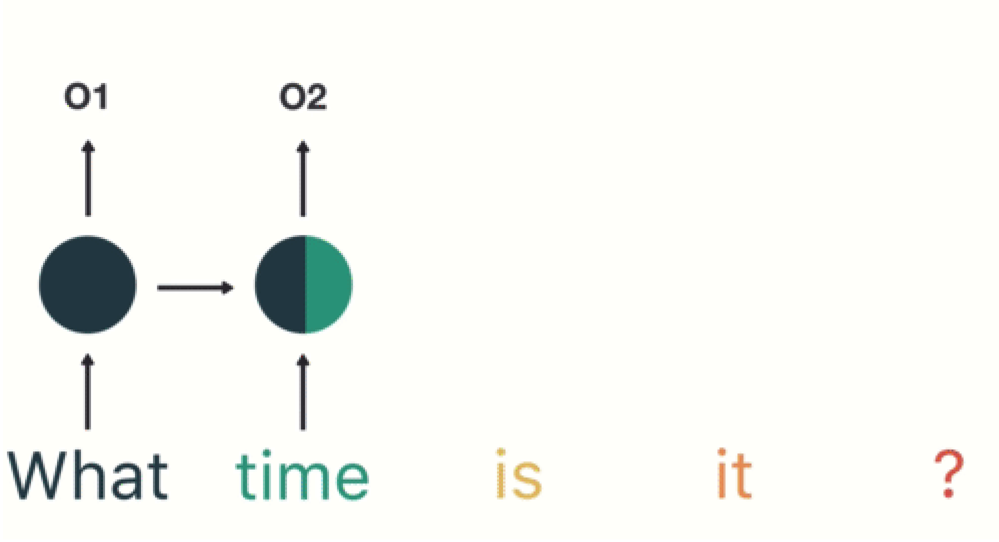
Recurrent Neural Networks

In CNNs we said that they were an improvement over FFNN for image classification as it preserved spatial information. In Recurrent Neural Networks, the main improvement is the preservation of time information an it is used for **sequential data** (Natural Language or Timeseries for example)

In this kind of Neural Network, previous outputs can be used as inputs for next computations, retaining information from the past in **order**.

This is important because it isn't the same saying "The food was good, not bad at all" as "The food was bad, not good at all" even though it contains the exact same words if we striped it of order.

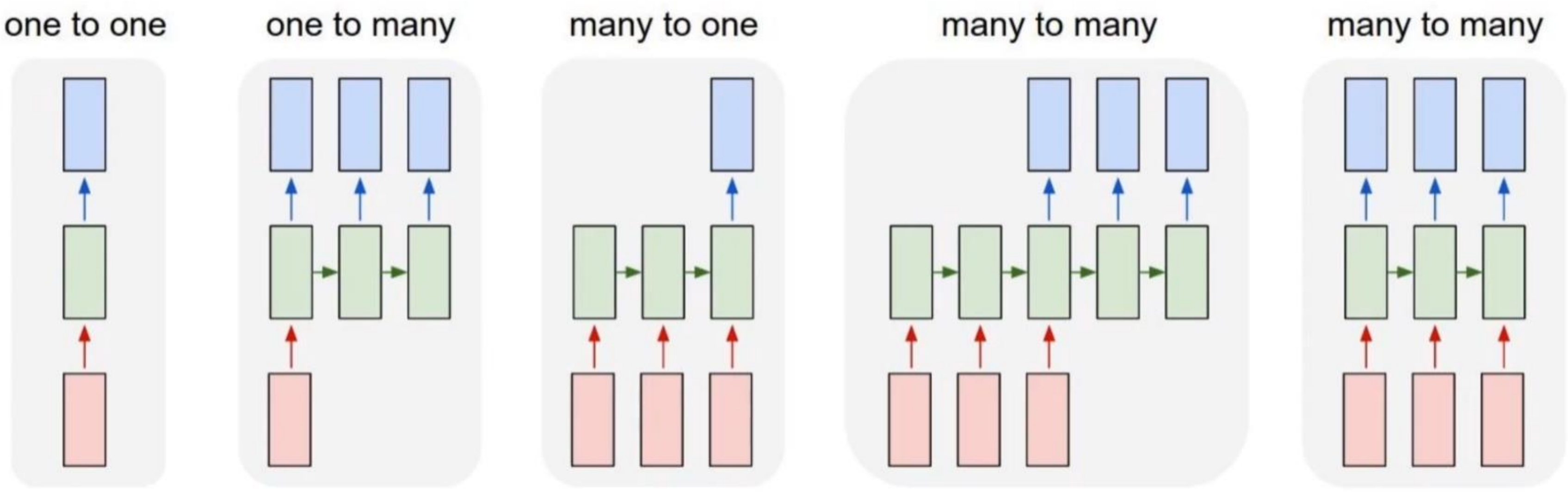




Deep Learning

Recurrent Neural Networks - Applications

Recurrent Neural Networks are used for tasks involving **sequential data**, with applications varying by input-output structure: **One-to-One** (not RNN use, this behaves as a FFNN), **One-to-Many** (text generation, where one input creates a sequence), **Many-to-One** (e.g., sentiment analysis or time-series forecasting, where a sequence leads to one output), and **Many-to-Many** (e.g., machine translation or speech recognition, where sequences map to sequences).



Deep Learning

Recurrent Neural Networks – How do they work?

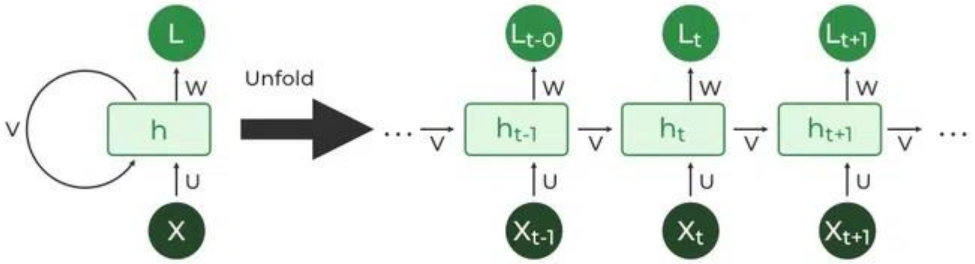
RNNs process sequences step-by-step, updating a "memory" (called the hidden state, h_t) at each step. At every time step t, the network receives two inputs:

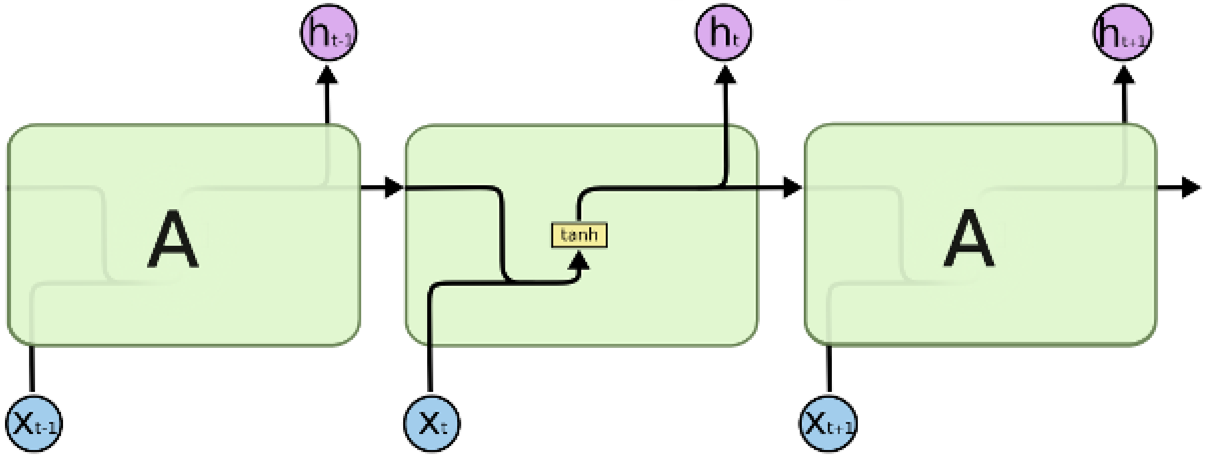
- Current Input (x_t): The data at position t in the sequence (e.g., a word in a sentence).
- Previous Hidden State (h_{t-1}): A compressed representation of all prior inputs, encoding the network's "memory" of the sequence up to step t - 1.

The hidden state is updated using a weighted combination of these inputs:

$$h_t = \sigma(W_x \cdot x_t + W_h \cdot h_{t-1} + b)$$

Here, W_x(input weights), W_h (recurrent weights), and b (bias) are shared parameters across all time steps.





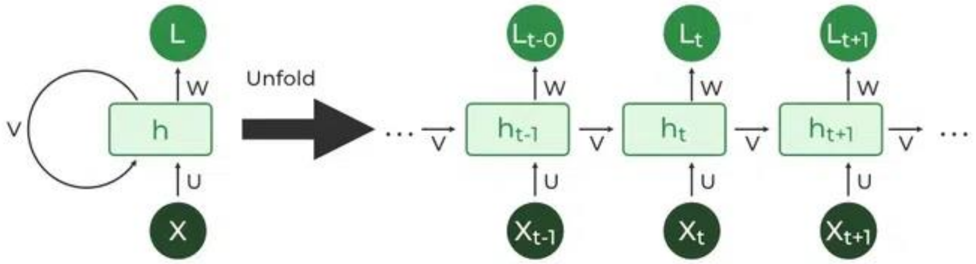
It is important that the input and recurrent weights are shared across time steps because this means that we will be able to process sequences as long as we want since we will have trained our weights for any situation.

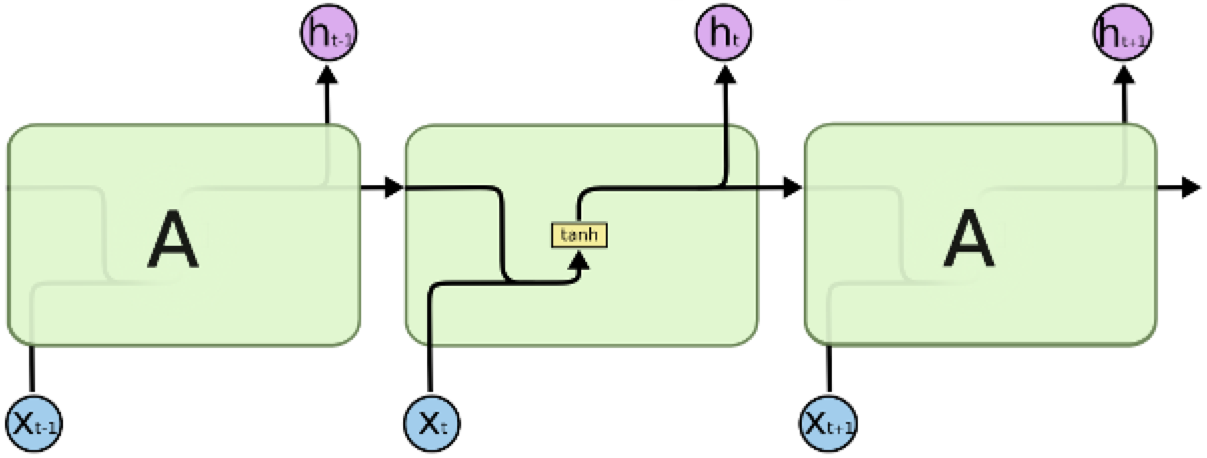
Deep Learning

Recurrent Neural Networks – How can we train it?

To train it we can unroll it and treat each step as if it was a layer in a FFNN and use back-propagation through time to propagate the error backwards and fix the weights for each step.

But with this, the problem of vanishing and exploding gradients arise. Because we are multiplying each step by shared weights, if the weights are lower than 1, the information from the beginning of the sequence will be lost at the time we reach the end of the sequence (Vanishing), and it will behave poorly during inference.

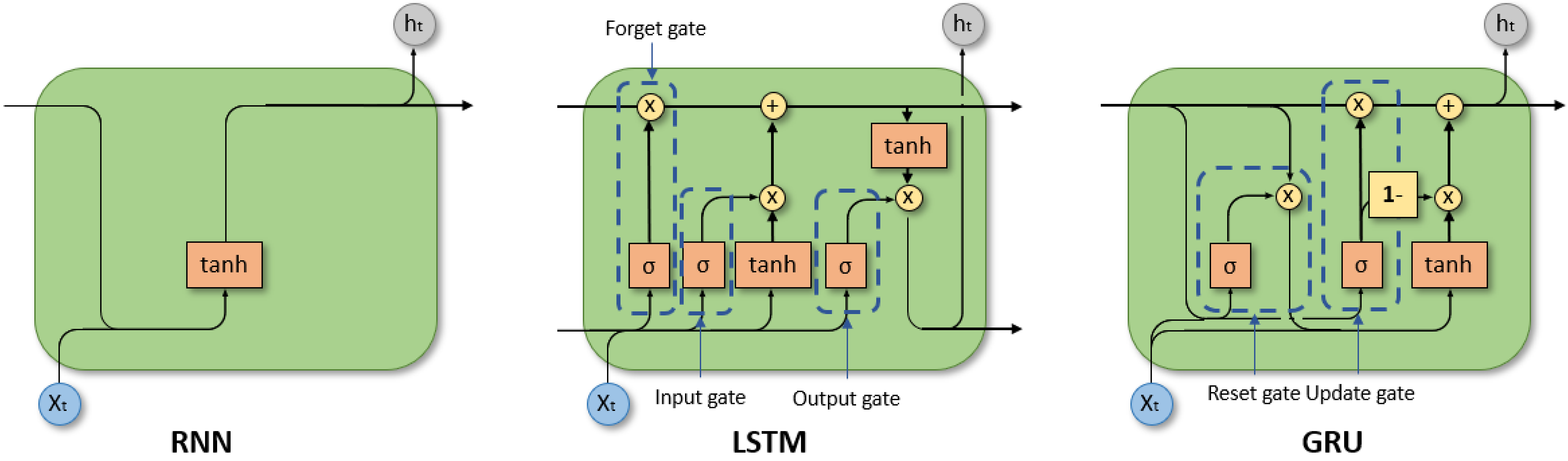




Deep Learning

Recurrent Neural Networks – Solving Gradient Vanish

To solve this problem, multiple implementations have been proposed that surpass the “Vanilla” RNN and are capable to retain information for longer. This are the Long-Short Term Memory and the Gated Recurrent Unit that have special gates that select which information to remember and what to forget.



RNN

LSTM

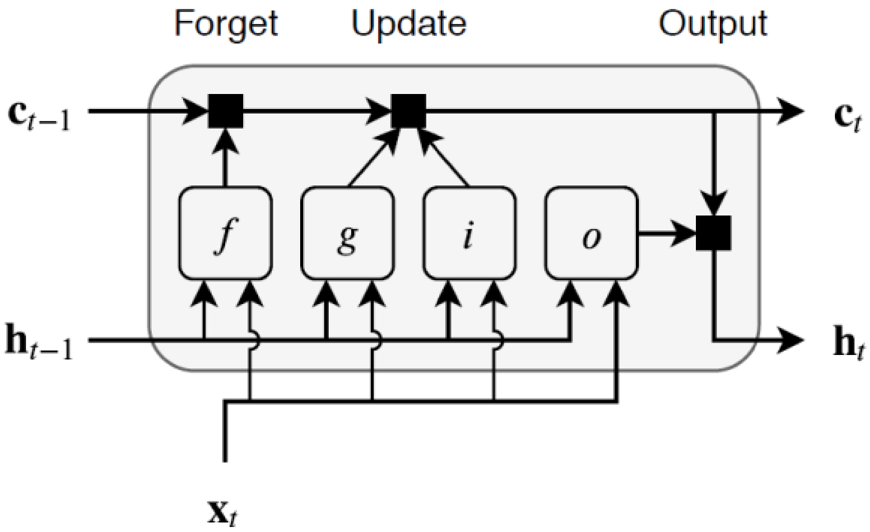
GRU

Deep Learning

Recurrent Neural Networks – LSTM

LSTMs use a cell state that acts as persistent memory, retaining information over extended sequences.
Three gating mechanisms regulate this memory:

- **Forget Gate**: discards irrelevant data (e.g., outdated details)
- **Input Gate**: adds new, relevant information to the cell state.
- **Output Gate**: determines what portion of the cell state to expose as the hidden state for predictions.



So, a well-trained LSTM would predict the next word like this:

“The cat, which chased the mouse for hours, finally caught it and felt very ______.”

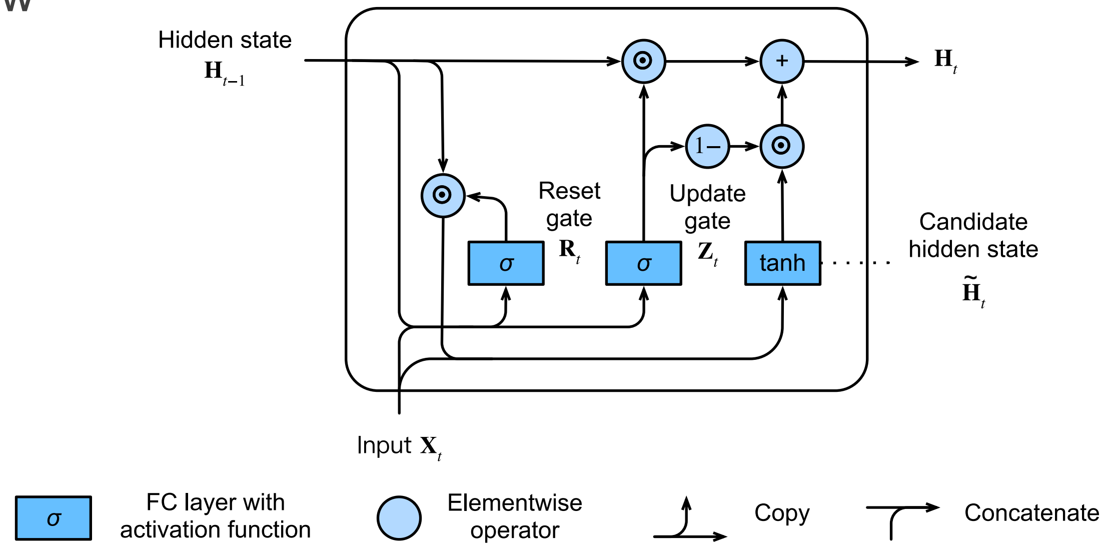
1. **Forget Gate**: Retains *“cat”* and discards *“mouse”* after it’s caught.
2. **Input Gate**: Stores *“chased for hours”* to imply exhaustion.
3. **Output Gate**: Uses *“exhausted”* from the cell state to predict *“tired”*.

Deep Learning

Recurrent Neural Networks – GRU

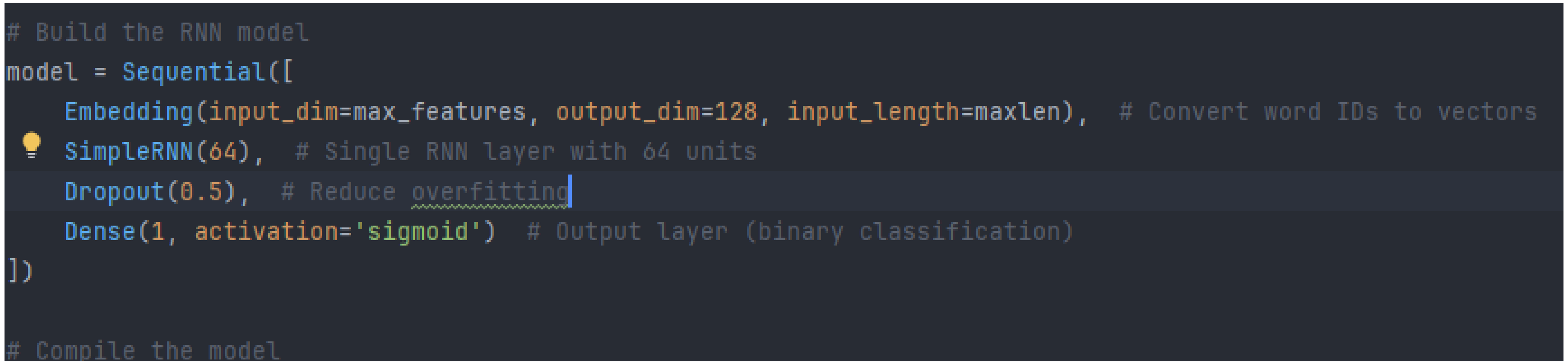
GRUs are basically a simplified version of the LSTM but with **two gates** instead of three: a **reset gate**, which determines how much past information to ignore when processing new input, and an **update gate**, which balances retaining old information with adding new data.

This makes them faster to train but worse at processing large sequences, so they are used for sentiment analysis of short texts (like tweets) or short-time forecasting.



Deep Learning

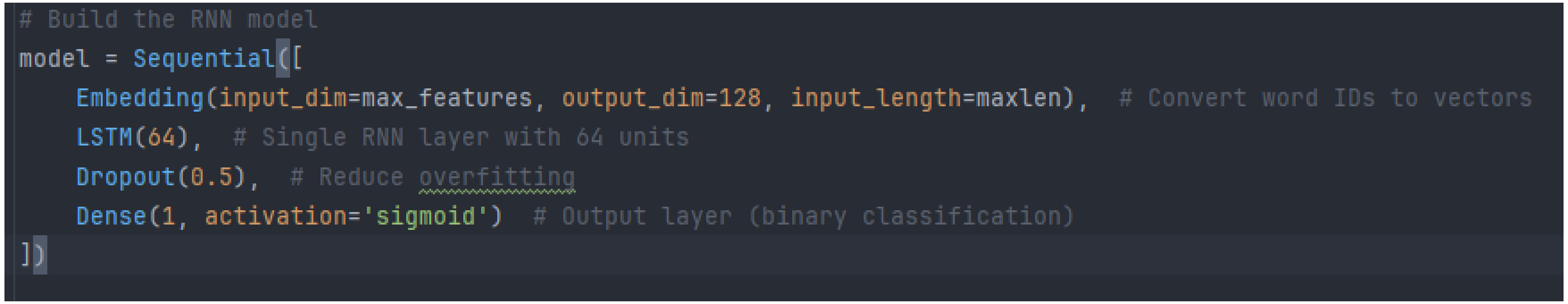
Recurrent Neural Networks – Code example



```
# Build the RNN model
model = Sequential([
    Embedding(input_dim=max_features, output_dim=128, input_length=maxlen),  # Convert word IDs to vectors
    SimpleRNN(64),  # Single RNN layer with 64 units
    Dropout(0.5),  # Reduce overfitting
    Dense(1, activation='sigmoid')  # Output layer (binary classification)
])

# Compile the model
```

RNN

LSTM



```
# Build the RNN model
model = Sequential([
    Embedding(input_dim=max_features, output_dim=128, input_length=maxlen),  # Convert word IDs to vectors
    LSTM(64),  # Single RNN layer with 64 units
    Dropout(0.5),  # Reduce overfitting
    Dense(1, activation='sigmoid')  # Output layer (binary classification)
])
```

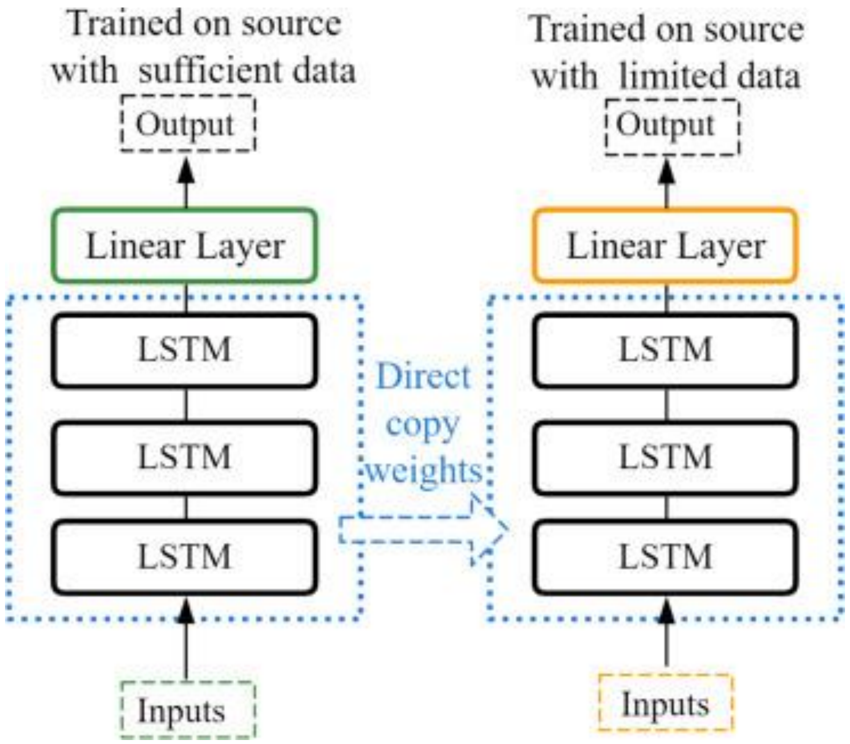
As simple as changing one word

Deep Learning

Transfer Learning

We have learned that both the filters that extract the features in spatial data and the parameters of a Recurrent Neural Network can be trained with large quantities of data. But what happens when we don’t have a big dataset from which to extract relations in our data for forecast or train feature extraction for object detection? We can use **Transfer Learning.**

We can train (or download pretained weights) our models with bigger sets of data that make sense for our problem, for example using a CIFAR100 trained ResNet, as the Convolutional Layers will have learned how to extract features from images, and we can freeze this layers and retrain the outer layers of our model to work for a new small set of data (For example changing the last FC for an untrained one and training it detect lung cancer in images).



Begin with Assignment 1