# Introduction

giovedì 20 novembre 2025 17:29

These notes are the logbook of my journey in Machine Learning.
The goal is not to repeat formulas or rewrite manuals, but to build real working artificial intelligence models from scratch.
The approach is always the same: start from the problem, design the architecture, train the model and verify its behavior in the real world.
The theory is reduced to the essentials, just what is needed to understand what is happening under the hood, and is immediately flanked by real PyTorch code.
This is not a "closed" book. It is a collection of technical notes that evolves along with skills, projects and mistakes. Each chapter is born from concrete experiments, not from abstract exercises.
Project Repository: https://github.com/robworkhubb/from-neurons-to-llm

## Chapter 1:

1. Theoretical foundations
    a. Chapter 1: How a neural network is formed
    b. Activation functions
    c. Training cycle
2. Project 1: Thermometer Network (Regression)
    a. Project 1: Linear Regression for Temperature
    b. Learning curve display
    c. Model inference
3. Computer Vision theoretical foundations
    a. 1.1: Visual Recognition with CNN
    b. Avoiding Overfitting: The Dropout
    c. Techniques to improve learning: Data Augmentation
4. Project 2: MNIST Classifier
    a. Project 2: MNIST Classifier
5. Project 2.1: Dog and Cat Image Classifier
    a. Project 2.1: Visual classification
    b. 2.1: Image classifier inference
    c. 2.1: Model Persistence: Saving the "Weights"

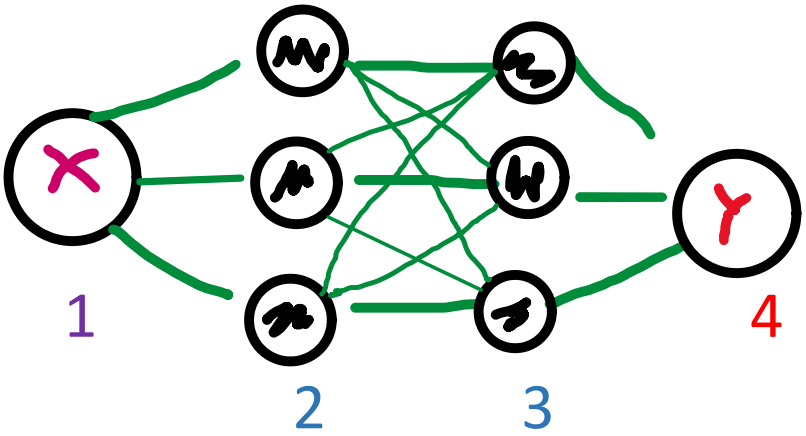
## Chapter 2 Natural Language Processing:

1. Theoretical foundations
    a. Chapter 2: NLP (Natural Language Processing)
    b. Tokenization
    c. Stemming & Lemmatization
    d. Embedding
    e. Transformers
    f. Positional Encoding
    g. Self-Attention
2. Project 3: Sentiment Analysis

# Conclusion and Updates:

# Chapter 1: How a neural network is formed

A neural network can be seen as a layered structure of artificial neurons that cooperate to turn a raw input into a meaningful output. Their organization vaguely resembles that of the human brain, but the functioning is purely mathematical: each neuron performs a small transformation and the entire network is born from the combination of thousands of these elementary transformations.



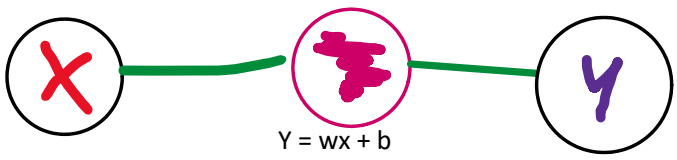

(1). Layers 2 and 3 are **hidden** layers, i.e. the non-visible layers, they are layers where the information is processed up to the output layer (y).

This structure is made up of neurons (layers 3 and 4), each neuron calculates linear combinations of inputs through weights and each neuron can specialize in different patterns, contributing to the generation of the output data.

Each neuron produces an output and the output of the first neuron will thus become the input
of the neuron of the next layer. In this type of layer, each neuron is connected to all those of the previous layer (**Fully Connected Layer**). The information then enters from (x) and is passed to several neurons in the hidden(1) layer.

Neurons use **parameters** to process information. These parameters decree how important and how important an input information is for the production of the output and these parameters are learned during **TRAINING**.

Example of a neural network with an input layer (x), a neuron, and an output layer (y).




A linear neuron can be seen as the simplest possible version of an artificial neuron. It receives an input value x, multiplies it by a weight w , and adds a term called bias **b**. The equation that describes this behavior is:

Weight controls how much the input affects the output: larger values amplify the effect of the input, while smaller values reduce it.
Bias, on the other hand, allows the neuron to move the result up or down regardless of the value of $x$. Together they determine how the neuron "responds" to the data it receives.
In real neural networks, an activation function is almost always applied after the linear combination. This step introduces nonlinearities and allows the network to learn more complex relationships that a simple linear transformation could not represent.

# Activation functions

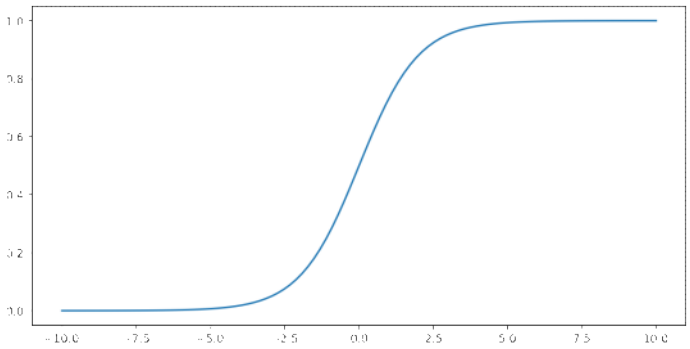
A firing function in a neural network is a mathematical operation applied to the output of a neuron. It determines whether or not a neuron should be activated, introducing nonlinearities into the pattern, which allows the network to learn complex patterns. Without these functions, a neural network would behave like a linear regression model.
The activation functions serve several purposes:

1. They allow neural networks to capture nonlinear data relationships, which are essential for solving complex tasks
2. They limit the output of neurons to a specific range (we will see later), preventing extreme values that can hinder the learning process
3. During backpropagation, the activation functions help in the calculation of gradients.

Three mathematical functions commonly used as activation functions are sigmoid, tanh, and ReLU.

The sigmoidal function (discussed above) performs the following transformation on the input, producing an output value between 0 and 1:

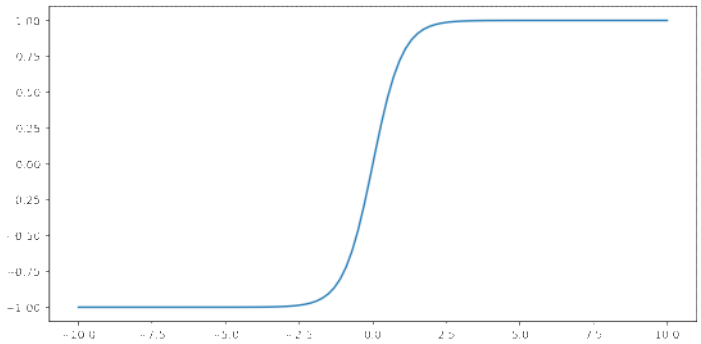
$$F(x) = \frac{1}{1 + e^{-x}}$$




The tanh function (short for "hyperbolic tangent") transforms the input to produce an output value between -1 and 1:

$$F(x) = \tanh(x)$$

Here's a chart of this feature:

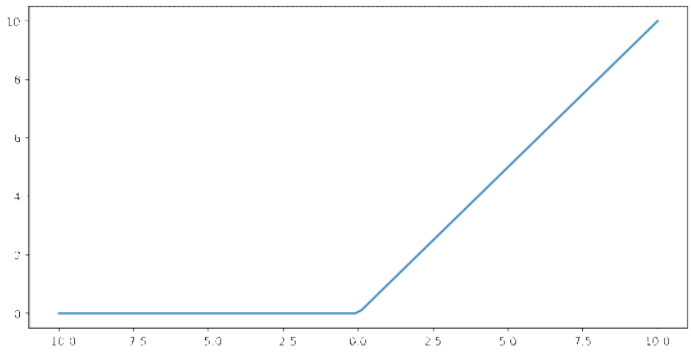


The **rectified linear unit** trigger function (or **ReLU**, for short) transforms the output using the following algorithm:

- If the input value is less than 0, it returns 0.
- If the input value is greater than or equal to 0, it returns the input value.

The ReLU function can be represented mathematically by the max() function:

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



The Softmax trigger function transforms a vector of numbers into a probability distribution, where each value represents the likelihood of a particular class. It is especially important for multi-class classification problems.

- Each output value ranges from 0 to 1.
- The sum of all output values is equal to 1.
  This property makes Softmax ideal for scenarios where each output neuron represents the probability of a distinct class.

**Softmax activation function**
**The Problem**
Imagine that your neural network is classifying an image and producing these numbers:
Cat: 4.2
Dog: 1.8
Bird: -0.5

**Question**: What do these numbers mean? These are not probabilities!
**The Solution: Softmax**

The Softmax function transforms any number into a **probability distribution**:

- All values become positive (between 0 and 1)
- The total sum is always 100% (= 1)

**How it works (step-by-step)**

**Input: Raw Scores**

Cat: 4.2
Dog: 1.8
Bird: -0.5

**Step 1: Exponentiation**

Let's raise everything to the power of $e$ (≈2.718) to make positive:

python

```
import math
gatto_exp = math.exp(4.2) # = 66,686
cane_exp = math.exp(1.8) # = 6.050
uccello_exp = math.exp(-0.5) # = 0.607
```

**Why exponential?**

1. It makes all numbers positive
2. Amplifies the differences (66 vs 6 is more noticeable than 4.2 vs 1.8)

**Step 2: Normalization**

Let's divide each value by the total sum:

python

```
total = 66.686 + 6.050 + 0.607 # = 73.343
prob_gatto = 66.686 / 73.343 # = 0.909 → 90.9%
prob_cane = 6.050 / 73.343 # = 0.083 → 8.3%
prob_uccello = 0.607 / 73.343 # = 0.008 → 0.8%
```

**Output: Probability**

Cat: 90.9% ✓
Dog: 8.3%
Bird: 0.8%

**Verify**: 90.9 + 8.3 + 0.8 = 100% ✓

**Key Properties**

**1. Sum = 1 (100%)**

```
print(probabilita.sum().item()) # 1.0
```

Each value is a "slice of the pie". All the slices make the whole cake.

**2. Values between 0 and 1**

python

```
print(probabilita.min().item()) # ≥ 0
print(probabilita.max().item()) # ≤ 1
```

It is impossible to have negative probabilities or greater than 100%.

**3. Amplify Differences**

```
# Input: Small Difference
```

```
input1 = torch.tensor([2.0, 2.1])
print(F.softmax(input1, dim=0))
# tensor([0.4750, 0.5250]) ← Almost 50/50

# Input: Big Difference
input2 = torch.tensor([2.0, 4.0])
print(F.softmax(input2, dim=0))
# tensor([0.1192, 0.8808]) ← Much sharper!
```

The more "secure" the network (distant scores), the more the probability focuses on a class.

**Mathematical Formula**

For a vector **z** = [$z_1$, $z_2$, ..., $z_n$], the Softmax is defined as:

$$\sigma(z_i) = \frac{e^{(z_i)}}{\sum_{j=1 \text{ to } N} e^{(z_j)}}$$

**Where**:

- **e^(zi)**: Exponential of the i-th value
- **Σ e^(zj)**: Sum of all exponentials (to normalize)

**In PyTorch**:

```
# PyTorch does all of this automatically:
probabilities = F.softmax(logits, dim=0)
```

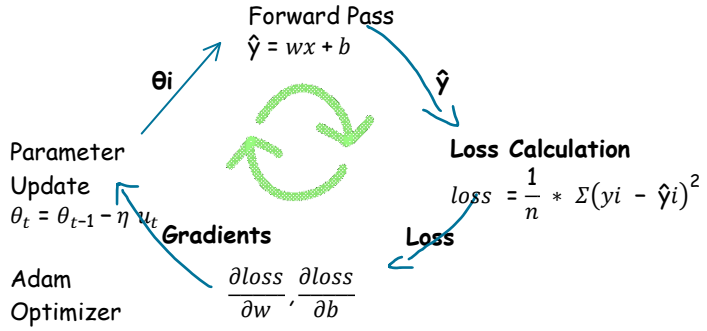
**When is it used?**

**Multi-class classification** (e.g.: recognizing cats, dogs, birds)

- Final Network Output
- It is used to interpret predictions

# Training cycle

As mentioned in the introduction, we have **weight** and **bias** parameters.
These parameters must be learned through the training cycle. To carry out these cycles we need many **training data**.
The model learns through this learning loop.




Note: These formulas are mostly used in solving linear regression problems.

The first step (**Forward Pass**) is used to calculate the expected output (y).
In the first iteration of the training, the model is not accurate and the output will be incorrect, because the training cycle has just begun.

The second step (**Loss Calculation**) takes our expected output generated and compares it with the real one (we know what the real output should be for each input and this type of training is called supervised training).
In the equation we are going to make a difference between the real value ($yi$ ) and the value predicted by the model (ŷ) and we square it, operation we do for all the expected outputs, comparing them with the real value and then making a mathematical average, in this way we calculated **the error**.
The error represents how wrong our model has been in calculating our values, the purpose of our cycle is **to reduce the error,** to reduce as much as possible the **difference** between the **real** outputs and those **predicted** by the **model**.

In the third step (**backpropagation**) we calculate what is the influence of each parameter in the production of the error (loss), we take the various parameters and obtain gradients.

In the fourth step (**Parameter Update**) we implement the **Adam Optimizer**, an **algorithm** that takes the parameters generally defined with theta (θ) which can mean (w) or (b) or any other parameter.
To obtain the updated parameter, it takes the previous one and subtracts this term (η), the term (u) takes into account the gradients, so the more a parameter has affected the error, the more it will have to be modified and the gradients (u) keep track of it.
The term Eta (η) takes into account the **learning rate**, the size of the step with which the parameters (step) are updated, the higher it is, the more drastically the parameters will be modified to achieve an improvement and the faster learning will be.
If we have a learning rate that is too high, we could run into instability.

At this point we have the updated parameters and the cycle starts again, we make a new Forward Pass formula with the new parameters (θi), we get a new expected output, we compare it with the real one and the error is assumed to be slightly less at the second pass. We get the gradients, update the parameters, and get the new ones. This cycle will be repeated thousands of times even showing the same training data over and over again until the model learns to produce more satisfactory results.

Imagine you are on a mountain in the dark. The gradient is the direction of the maximum slope. In our case:

- Mountain = error function
- Goal = go down to the valley (minimal error)
- Gradient = direction in which to descend

Mathematically: it is the derivative that indicates "how much the error changes if I change this parameter".

# Project 1: Linear Regression for Temperature

**First Project: Linear Regression for Temperature**

**Before You Begin: What Are Tensors?**

In the code that follows, you'll see torch.tensor(). But what exactly is a tensor?

**Tensor vs Python List**

```
# Normal Python List
temperature = [55, 60, 65, 70, 75]
print(temperature[0]) # 55
```

A Python list can contain anything (numbers, strings, objects), but it's **slow** for mathematical operations.

python

```
# Tensore PyTorch
temperature_tensor = torch.tensor([55, 60, 65, 70, 75])
print(temperature_tensor[0]) # tensor(55)
```

A tensor is a data structure optimized for:

- **Fast math calculations** (on GPU)
- **Automatic gradient calculation** (for training)
- **Parallel operations** (on thousands of numbers together)

**Practical Comparison**

```
import torch
import time

# Python List
lista_a = list(range(1000000))
lista_b = list(range(1000000))

start = time.time()
lista_c = [a + b for a, b in zip(lista_a, lista_b)]
print(f"Python lists: {time.time() - start:.4f} seconds")

# Tensori PyTorch
tensor_a = torch.arange(1000000)
tensor_b = torch.arange(1000000)

start = time.time()
tensor_c = tensor_a + tensor_b
print(f"PyTorch tensors: {time.time() - start:.4f} seconds")
```

**Typical Output:**

```
Python Lists: 0.1234 seconds
PyTorch tensors: 0.0012 seconds ← 100x faster!
```

**Tensor Dimensions**

Tensors can have multiple sizes:

```
#0D: Scalar (Single Number)
scalare = torch.tensor(42)
print(scalare.shape) # torch. Size([])

#1D: Vector (List of Numbers)
vettore = torch.tensor([1, 2, 3, 4])
print(vettore.shape) # torch. Size([4])
```

```
#2D: Matrix (table)
matrix = torch.tensor([[1, 2, 3],
[4, 5, 6]])
print(matrix.shape) # torch. Size([2, 3]) ← 2 rows, 3 columns

#3D: "Cube" (ex: RGB image)
cubo = torch.randn(3, 224, 224) # 3 canali, 224×224 pixel
print(cubo.shape) # torch. Size([3, 224, 224])
```

**Why unsqueeze(1)?**

In the code, you'll see:

```
x = x.unsqueeze(1)
```

This adds a dimension. Why is it needed?

```
# Before: 1D vector
x = torch.tensor([55, 60, 65, 70, 75])
print(x.shape) # torch. Size([5])

# After unsqueeze: 2D matrix with 1 column
x = x.unsqueeze(1)
print(x.shape) # torch. Size([5, 1])
print(x)
# tensor([[55],
# [60],
# [65],
# [70],
# [75]])
```

PyTorch models expect 2D input: [numero_esempi, features]

- In our case: 5 examples, 1 feature for example

**dtype: Data Type**

```
torch.set_default_dtype(torch.float32)
```

This tells PyTorch to use 32-bit decimal numbers (trade-off between accuracy and speed).

**Common types**:

- torch.float32 (default): Normal decimal numbers
- torch.float64: More precise but slower
- torch.int64: Integers
- torch.bool: True/False

**What does "Plot" mean?**

```
plt.scatter(x, y)
```

**"Plot"** = Draw a chart to visualize the data

It's like turning a table of numbers into an image:

```
# Data: Two columns
x = [55, 60, 65, 70, 75]
y = [13, 16, 19, 22, 25]

# Graph: points on a Cartesian plane
plt.scatter(x, y)
plt.xlabel("X (Temperature Inventate)")
plt.ylabel("Y (Temperature Reali)")
plt.show()
```

**Why is it useful?**

- Immediately see if there is a relationship between X and Y
- Discover outliers (weird spots)
- Understand if you need a linear or complex model

Now let's proceed with the code, we are going to create a very simple **neural network** . We'll create it via a **Jupyter Notebook** in **VSCode** using **pytorch**.

Let's imagine that we have this device that measures temperatures invented in Fahrenheit (list x), at each invented temperature our model will have to tell us what the corresponding temperature is, so we train the model to receive a temperature x as input and give the right

temperature y.

```
import torch
x = torch.tensor([55, 65, 75, 54, 67, 78, 53, 65, 76, 60, 70, 81, 51, 68],
dtype=torch.float32).unsqueeze(1)
y = torch.tensor([13, 18, 24, 12, 20, 26, 12, 18, 24, 14, 21, 28, 12, 20],
dtype=torch.float32).unsqueeze(1)
```

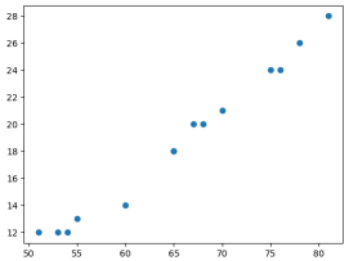
As you can see we used torch to create **tensors**, these are similar to the classic Python lists but with very specific functionality.
In the tensor x we have the invented temperatures (inputs), while in the tensor y we find the actual temperatures (output).
We define the standard datatype for tensors to achieve maximum compatibility and performance.
We use the unsqueeze function to give another dimension to our tensor, because our model that we will create later does not want **one-dimensional** tensors, with this function we are going to make the tensor two-dimensional with 1 column.

Now in the next statement we "plot" the training data using the **matplotlib** library.

```
from matplotlib import pyplot
pyplot.scatter(x, y)
```

"plotting" data means
**Graphically represent numerical data** to analyze and interpret them, highlighting trends, relationships and outliers.

Here is the output generated by pyplot. On the x-axis we have x (invented temperature), on the ordinate instead we have y (effective temperature).




The graph shows that the training data has
effectively a linear relationship between x and y.

# MODEL DEFINITION, LOSS FUNCTION AND OPTIMIZER

We define a linear model with 1 input, 1 output and a single neuron and visualize the outputs produced by the model.

```
tensor([[42.8796
],
[50.5250],
[58.1704],
[42.1151],
[52.0541],
[60.4640],
[41.3506],
[50.5250],
[58.9349],
[46.7023],
[54.3477],
[62.7576],
[39.8215],
[52.8186]],
```

Here is the answer of our model that as we can see
is not accurate, this output represents our expected output (ŷ).
The first input of the list was 55 and predicted 26.14, while our real value ($yi$) is 13.

```
grad_fn=<AddmmBa
ckward0>)
```

This happens because the model, when the parameters of the **equation y = wx + b weight and bias** respectively are first instantiated, are **randomly** generated. In fact, if we try to re-run the cell we will have different outputs because the parameters are regenerated leading to a different output every time the model is **instantiated**.
However, we can have equal outputs by fixing the random variability of these parameters in this way:

```
torch.manual_seed(42)
```

The **seed** is the seed from which the **randomness** of our model germinates, it is usually chosen randomly but usually the value 42 is used for philosophical reasons. Now every time the model is instantiated we have the same values in output

Now let's define the **loss function** and the **optimizer**.

```
loss_fn = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.1)
```

For the loss function we are going to define the algorithm we will use via torch, i.e. **MSELoss (Mean Squared Loss)**, defining it in this way torch will implement the formula defined in the second step of the training cycle.

Let's use the **Adam Optimizer**, also of this algorithm the formula is executed by pytorch. We pass 2 parameters to the optimizer, the parameters it needs to work on and our learning rate (η) initially set 0.01

**NOTE:** The **loss function** is a mathematical function to measure the difference between the **expected value** (ŷ) from a model and the **real** value ($yi$) or desired.

# TRAINING CYCLE

Now let's write the code of our training cycle, following the order of the functions of the training cycle Training cycle.

```
for epoch in range(3000): # for each epoch in the range 1000
 optimizer.zero_grad() # zero gradients
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
 loss.backward() # gradients
 optimizer.step() # Update weights
    if epoch % 100 == 0:
        print(f'Epoca {epoch}, Loss: {loss}')
```

We define the training cycle with a **for**, which means that for every 1000 **epochs** it performs the training cycle.
With the **optimizer.zero_grad()** function we reset the **gradients** of the previous cycle.
We then define a **variable y_pred** that represents the **expected output (Forward Pass)**.

An epoch is defined as a repetition in which the model learns all the data in our dataset

**After this instruction we go to calculate the loss using pytorch, then we initialize a loss variable that will contain the result of the formula generated by pytorch with the loss_fn command, as predicted by the formula we pass 2 parameters respectively the expected output (y_pred) and the actual output (y) or the tensor generated at the beginning (effective temperatures).**

Then we have to compare the loss with the various parameters to obtain the gradients to evaluate how much each parameter has affected the error.
Let's call the **backward** function on the **variable loss**.

Now let's move on to step 4 of our cycle where we update the parameters, then use our **optimizer** variable previously declared and call the **step** function, this function is used to make a parameter update step.

The if is used to print **the progress** every 100 epochs and display the current epoch and loss function.

Here we find the output generated after the training cycle.

```
Epoch 0, Loss: 1044.3514404296875
Epoch 100, Loss: 6.815361022949219
Epoch 200, Loss: 6.2898993492126465
Epoch 300, Loss: 5.671482086181641
Epoch 400, Loss: 5.007758617401123
Epoch 500, Loss: 4.340624809265137
Epoch 600, Loss: 3.7006661891937256
Epoch 700, Loss: 3.1093404293060303
Epoch 800, Loss: 2.580324411392212
Epoch 900, Loss: 2.1207163333892822
Epoch 1000, Loss: 1.7322407960891724
```

We can see that in the first epoch, the loss function is a very high value
So the difference between the expected output and the actual output is huge. As the cycle continues, this value decreases , so the difference will be smaller and the value will be much closer to the real one.

If we increase the range in for it will perform many more **repetitions**, so by doing more steps it will learn more **information**.
If we increase the **learning rate** it will decrease much more, but if we increase the learning rate we must increase the number of **repetitions** avoiding instability.
An example with 3000 **epochs** and a **learning rate** of 0.1:

```
Epoch 2800, Loss: 0.4438796937465668
Epoch 2900, Loss: 0.44381776452064514
```

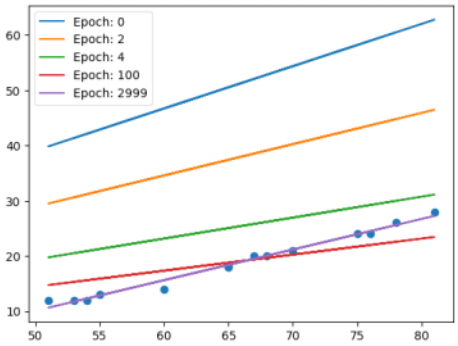
The model learns and reaches a low value but begins to "**stop**" on a value, beyond this value you can no longer go down if not **slightly**.

# Learning curve display

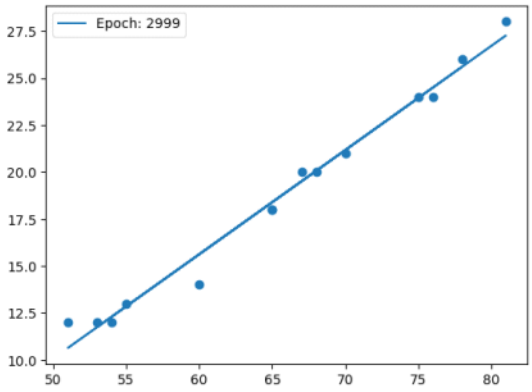
Now let's visualize the learning curve.

```
    if epoch in [0, 2, 4, 100, 2999]:
 Pyplot.plot(x, y_pred.detach(), label = f'Epoch: {epoch}')
pyplot.scatter(x,y)
Pyplot.legend()
```

We plot this data in **epochs 0,2,4,100,2999** and visualize the relationship between **x (input) and the expected y**, we use **detach to detach** the gradients in the representation. Let's print the chart with the **scatter** and activate the **legend**.




At epoch 0 we notice that the line is very **high** therefore completely different from the value of the x, in the second epoch we have a lowering of **the bias**. In the last thousand we notice that it changes the **weight that** regulates the slope of the line.
If we plot only the last epoch (2999) we see that the approximation is almost correct.




**We notice that the model can predict our x almost perfectly**

# Model inference

**Now let's carry out neural network inference, which is the process that the model uses to draw conclusions from new data.**

```
current_temperature = torch.tensor([70], dtype=torch.float32).unsqueeze(1)
model(current_temperature)
```

This cell goes to give **70** as input, saved on a variable and goes to call the model response. We receive an accurate answer consistent with the initial **y** .
So the model learned very well from our information.

```
tensor([[21.1672]], grad_fn=<AddmmBackward0>)
```

Now let's print the parameters of the model **respectively weight** and **bias**.

```
for param in model.parameters():
    print(param)
```

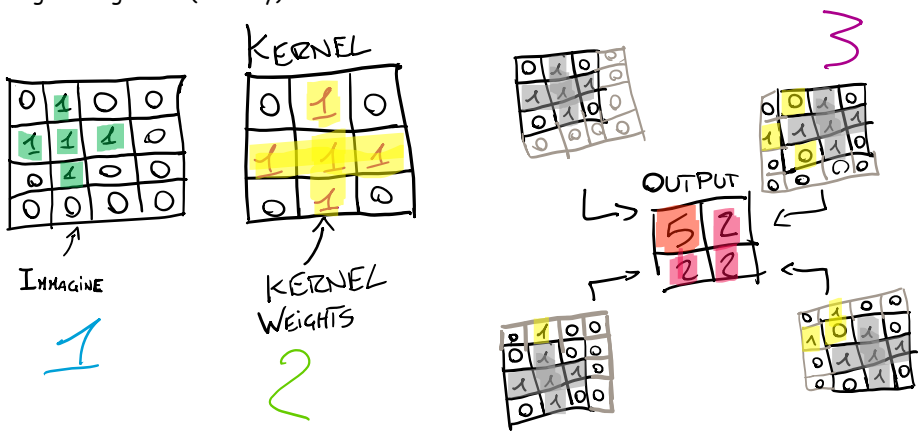
The output we receive is this:

```
Parameter containing:
tensor([[0.5537]], requires_grad=True)
Parameter containing:
tensor([-17.5949], requires_grad=True)
```

The first value (**0.55**) is the **weight, the** second (**-17.6**). Returning to the initial equation (**Page 1**). By realizing the equation we will find **y** or the **real temperature**.

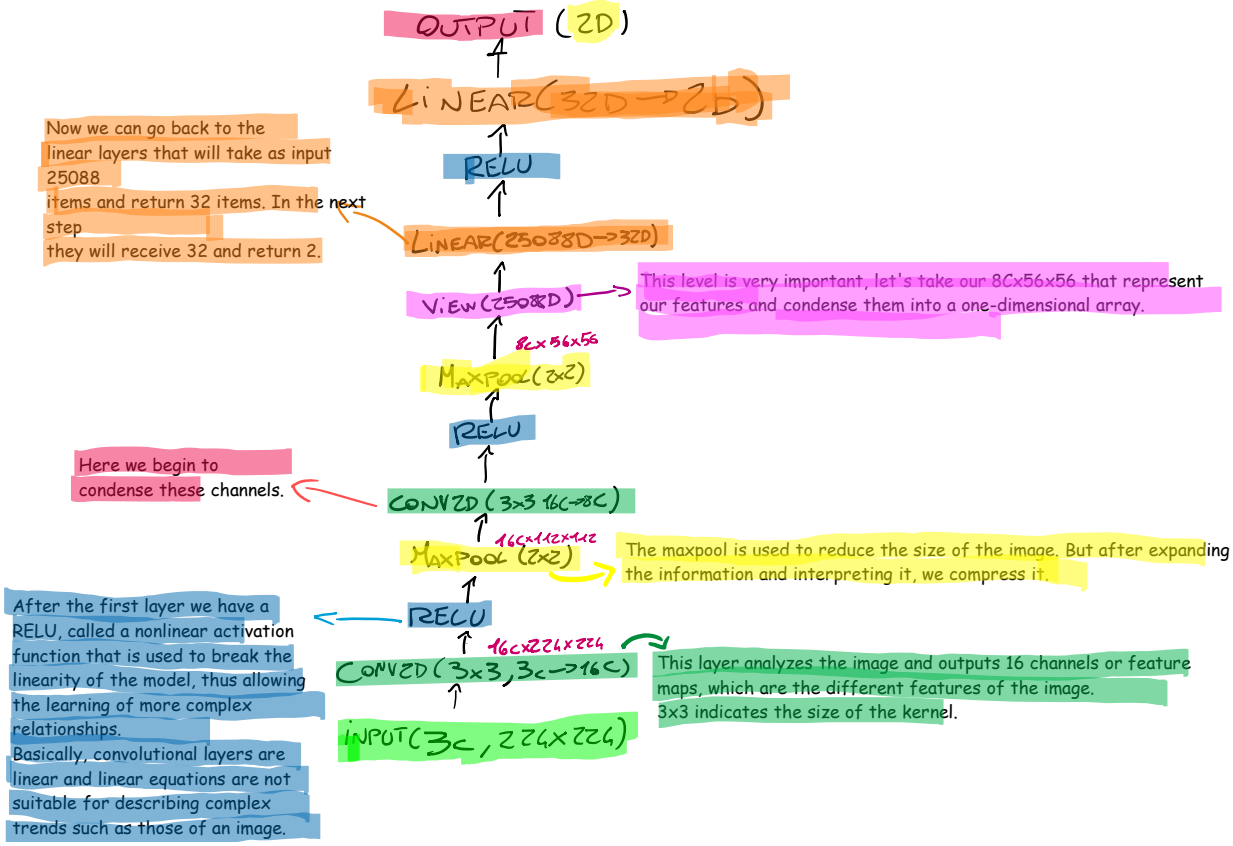
# 1.1: Visual Recognition with CNN

If in the last project we worked with numbers, in this one we will work with images, we will create an image classifier that returns a probability. To do this we need **convolutional layers**. They are layers made specifically for image management (not only).




The stride is the step at which the filter moves over the input during the convolution operation. A larger stride value results in a larger shift in the filter and therefore a reduction in the size of the output. Conversely, a smaller stride value results in less filter displacement and therefore larger output. The formula for calculating the size of the convolution output based on the stride value is:

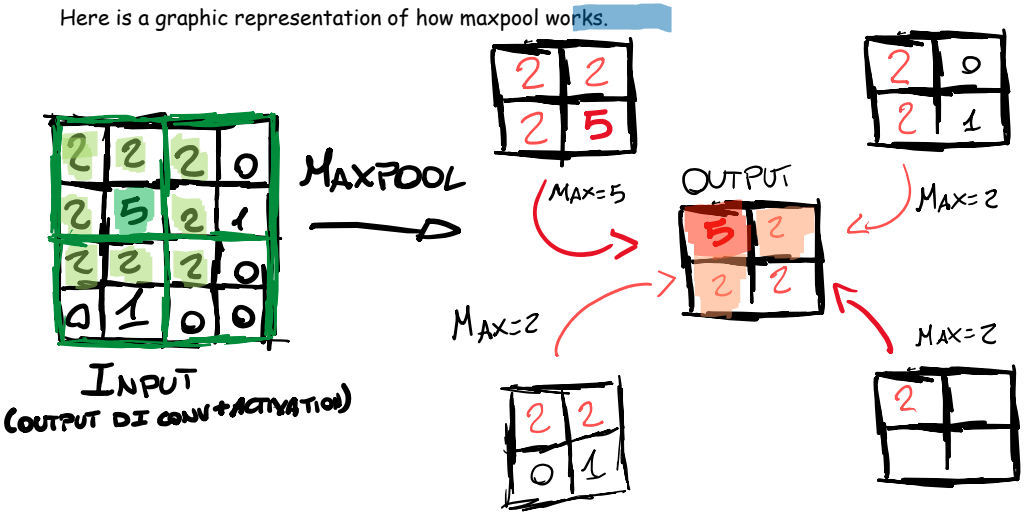
We use this layer because it can detect local features of our image such as edges, corners, textures, shapes, seeing just how the pixels are arranged. The output that produces this convolutional layer is called a channel or feature, it can also produce multiple channels. Each channel is capable of capture a different feature e.g. (edge).
To do this he uses a **filter called kernel**, usually the 3x3 matrix is used (Figure 2) in the kernel we find the kernel weights, the parameters that then have to be learned from the model, this will require an extra effort at the computational level.
This matrix scans (Figure 3) all the different parts of the image producing an output, mathematically it calculates a scalar product of the kernel itself with the scanned matrix of the image (Figure 1).
We start with a 4x4 image and in Figure 3 we see that the output is only 2x2, that output is called a feature.
Now let's look at the structure of the neural network.




Note: We use a one-dimensional array because we want our model to return a probability.

Basically, convolutional layers are linear and linear equations are not suitable for describing complex trends such as those of an image.

Here is a graphic representation of how maxpool works.

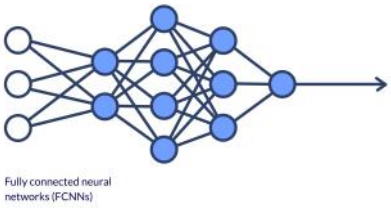



The maxpool takes the input, which in this case is a channel of feature maps output from a convolution layer, as we see in the output it takes only the main features of our channel and inserts them into a 2x2 matrix.

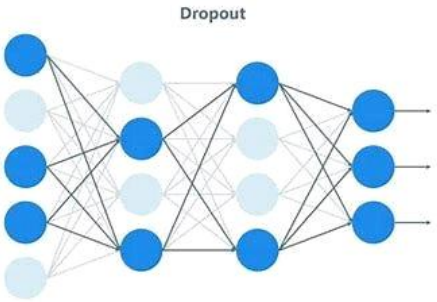
# Avoiding Overfitting: The Dropout

When a neural network is very large and trains for a long time, it risks going into **overfitting** (over-adaptation). Neurons begin to "trust" each other too much, creating complex dependencies that only work on training data but fail in the real world. It's like a group work where one student does everything and the others just copy.
**The Dropout** (literally "drop") solves this problem in a drastic but effective way.
**The concept:** During training, at each step, we randomly deactivate ("turn off") a percentage of neurons (usually 20% or 50%).
**The effect:** Neurons can no longer blindly rely on their neighbors, because they could be turned off at any time. Each neuron is forced to learn useful characteristics independently. This creates a much more robust and democratic network, capable of better generalizing on data never seen before.

*Note:* During inference (when we use the model), the Dropout is automatically deactivated and all neurons participate 100%.




We see a **Fully Connected** network. Each neuron in the previous layer is connected to all the neurons in the next layer. All nodes are active and participate in information processing. In this configuration, neurons can develop a "co-dependency," relying excessively on signals from neighbors to correct mistakes, which often leads to **overfitting**.




We see the same network to which the **Dropout** was applied. The gray nodes represent neurons that have been **temporarily deactivated** (turned off) for this specific training step.

- We notice that not only is the neuron turned off, but **also all connections (arrows)** in and out of it are removed.
- The network appears "leaner" and forces active neurons (the colored ones) to take on the extra work, learning more robust and independent characteristics.

**Please note:** The deactivation is random and changes with each individual update step. A neuron that is turned off now could be turned on in the next step.

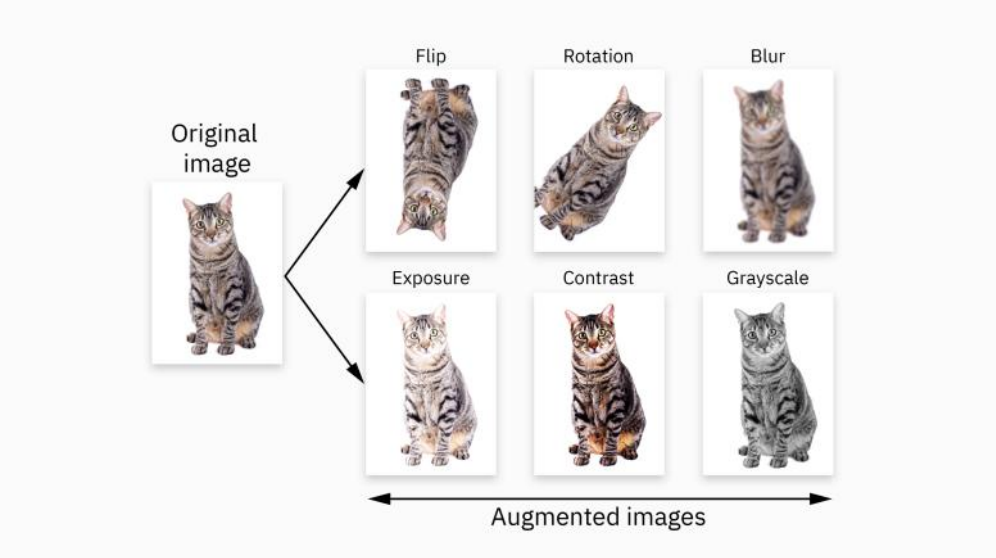
# Techniques to improve learning: Data Augmentation

One of the main problems in Deep Learning is having little data. If we show the same 100 photos of cats to the network over and over again, she will learn to recognize *those* specific cats, not the universal concept of "cat."

**Data Augmentation** is a technique to artificially augment our dataset without having to take new photos. The idea is to apply random transformations to the training images each time they are passed to the model.

**How does that work:** Every time the image is loaded, it undergoes a random change: it can be rotated a few degrees, flipped horizontally (like in a mirror), slightly zoomed in or changed in brightness.

**Why it's brilliant:** For the computer, a cat rotated 15° is a completely different matrix of pixels from the original. In this way, the model never sees the identical image twice and learns that the shape of the cat is independent of its orientation or light. It is as if our dataset becomes virtually infinite.



# Project 2: MNIST Classifier

Today we are going to use the knowledge learned earlier with image recognition and classification. We're going to build a model in which we pass an image of a number handwritten by a human taken from an online dataset and the model returns which number it corresponds to. Let's start by importing the necessary libraries.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
```

Let's do the initial part where we convert the images into tensors and prepare the data.

```
# Convert MNIST images to 4-dimensional tensors (n of images, height, width, and channels)
transform = transforms.ToTensor()
# Training data
train_data = datasets.MNIST(root = 'cnn_data', train = True, download=True,
transform=transform)
# Test Data
test_data = datasets.MNIST(root = 'cnn_data', train = False, download=True,
transform=transform)
# Dataloader
train_loader = DataLoader(train_data, batch_size=10, shuffle=True)
test_loader = DataLoader(test_data, batch_size=10, shuffle=False)
```

The dataset is taken from the library and installed in the project directory in the cnn_data

We split the data into training data and "test" data, we do it because if we feed an image that we used in training when we pass the image, it will already know what it is, while if we divide the data into training and set it will not only mechanically learn the training images but we give new images contained in the test data. So we will see if even with images that he has never seen he can give the accurate answer.

Why do we only mix train data and not test data? Because otherwise the model always learns in the same order and will not be able to contextualize, so for every 10 images (batch_size) we mix these images. The test data does not need to be mixed.

Now let's initialize the model and all its layers

```
# Model Class
class ConvolutionalNetwork(nn.Module):
    def __init__ (self):
 super().__init__()
        self.conv1 = nos.Conv2d(1, 6, 3, 1)
        self.conv2 = nos.Conv2d(6, 16, 3, 1)
        # Fully connected layer
        self.fc1 = nn.Linear(5*5*16, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
    def forward(self, x):
        x = F.relu(self.conv1(x))
 x = F.max_pool2d(x, 2, 2) #2 x 2 Kernel and Stride 2

        x = F.relu(self.conv2(x))
 x = F.max_pool2d(x, 2, 2)

 x = x.view(-1, 16*5*5) # Negative number so we can vary the batch size
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

        return x

# We create an instance of the model
torch.manual_seed(41)
model = ConvolutionalNetwork()
```

It means that the layer **receives 1 channel as input** , not a single piece of data, but an entire 2D map. Apply **6 distinct convolutional filters**. Each filter has a **3×3 kernel**. The stride is 1.
The result is **6 feature maps in output**.
The second layer receives **6 input channels**, i.e. the 6 feature maps produced before. Each filter is now **3×3×6**, because it has to watch all channels. Apply **16 different filters** and output **16 feature maps**.

After the convolutional layers we get 16 feature maps of 5×5 size.
Before moving to fully connected layers, these maps are flattened into a single vector of 5·5·16 = 400 values.
The first fully connected layer takes this vector and projects it into 120 neurons, building a more abstract representation.
The second layer further reduces dimensionality from 120 to 84.
The last layer maps the 84 neurons into 10 output values, one for each class.

In the forward method, we define how data traverses the network.
The x input first passes into the first convolutional layer, followed by a ReLU activation function, which introduces nonlinearities.
Immediately afterwards we apply a 2×2 max pooling with stride 2 to reduce the spatial dimension and make the features more robust.
The result then passes into the second convolutional layer, again followed by ReLU and another max pooling, further reducing the size of the feature maps.
At this point we flatten the output into a vector of dimension 16·5·5. We use -1 to let PyTorch calculate the batch size automatically.
The vector enters the fully connected layers: first from 400 to 120 neurons, then from 120 to 84, both with **ReLU**.
Finally, the last fully connected layer produces 10 output values, which represent the logits of the classes.

Now, after preparing the data, defining the network architecture and instantiating the model, we go on to create the training cycle of our model.

```
#Loss FN Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = 0.001)
# Training
# Variables
epochs = 5
train_losses = []
test_losses = []
train_correct = []
test_correct = []
# Loops
for i in range(epochs):
 trn_corr = 0
 tst_corr = 0
    # train
    for b,(x_train, y_train) in enumerate(train_loader):
 b += 1 # We initialize our batches to 1
 y_pred = model(x_train) # We take the leading y from our training set
 loss = criterion(y_pred, y_train) # Compare the prediction with the correct
```

```
answers in y_train

 predicted = torch.max(y_pred.data, 1)[1] # Add the number of correct
predictions. Indexed to the first point
 batch_corr = (predicted == y_train).sum() # When it was corrected by its batch.
True = 1, False = 0
 trn_corr += batch_corr # Tracks as we progress with the workout
        # Update Parameters
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Print
        if b%600 == 0:
            print(f'Epoch: {i} Batch: {b} Loss: {loss.item()}')
    train_losses.append(loss)
    train_correct.append(trn_corr)
```

After defining the loss function and the optimizer, we initialize the variables necessary to monitor the training.
We set the number of epochs, i.e. how many times the model sees the entire training dataset, and create lists to save loss and accuracy.

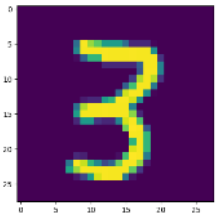
For each epoch, we iterate over the training dataloader batches.
For each batch, the model produces a prediction y_pred from the x_train inputs.
The loss is calculated by comparing these predictions with the real labels y_train using the CrossEntropyLoss.
Then we calculate how many predictions are correct by comparing the class with maximum probability with the real label. This allows us to track performance during training.
At this point we perform the classic optimization cycle: we reset the gradients, calculate the gradients via backpropagation and update the weights using the Adam optimizer.
We periodically print the loss to monitor the progress of the training.
At the end of each epoch we save the loss and the number of correct predictions.

Now after finishing the workout, let's get the image ready to be passed on to the model.

```
img_idx = 5214
# Let's take an image
test_data[img_idx] # tensor with an image and it will show the label
# Let's just take the data
test_data[img_idx][0]
# Let's resize
test_data[img_idx][0].Reshape(28, 28)
# Show the image
plt.imshow(test_data[img_idx][0].Reshape(28, 28)
)
```

This index is quite random, we can put any since the dataset is very large.

OUTPUT:



Once the image has been prepared and shown, let's pass it to the model and receive probabilities.

```
# Let's pass the image to the model
model.eval()
with torch.no_grad():
 new_pred = model(test_data[img_idx][0].view(1,1,28,28)) #batch_size of 1, 1 color channel, size 28*28
# Probability
new_pred
tensor([[-0.0991, -0.0449, 0.0486, 0.0004, 0.0598, 0.0327, -0.0668, -0.1152,
0.0228, -0.0117]])
```

If we execute this block of code we will get our probabilities for each class (number) in fact we have 10.
In the next block we are going to take the maximum probability with torch.

```
new_pred.argmax()
```

We get the tensor with the correct dataset label as the answer.
`Tensor(3).`

Now we have just finished developing our first MNIST image classifier.

# Project 2.1: Visual classification

After seeing the theory and structure of our neural network, we can now proceed with writing code and training. We have prepared 2 datasets, one contains 6000+ images of **cats** and the other 6000+ images of **dogs**, our model must return a probability, i.e. the probability that the image we feed to the model is a dog or a cat. Let's continue with importing the necessary libraries:

```
import torch
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision import datasets
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from PIL import Image
from sklearn.metrics import confusion_matrix, precision_score, recall_score,
accuracy_score
torch.manual_seed(42)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Here we simply imported the libraries and then declared a device variable that contains the device we are working on.
If the GPU is available (higher performance) we use that otherwise we use the CPU (much slower).

After this step we proceed with the creation of all the instance attributes, these will represent our **layers**.

```
class Net(nn.Module):
    # Neural Network Constructor Method/Schematic
    def __init__ (self):
 super().__init__()
        self.conv1 = nos.Conv2d(3, 16, kernel_size=3, padding=1)
        self.conv2 = nos.Conv2d(16, 8, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)

        self.fc1 = nn.Linear(8* 56 * 56, 32)
        self.fc2 = nn.Linear(32, 2)
```

Let's define a Net class by inheriting from the Module class made available by pytorch. Let's then define the constructor method (2 line). We then define the super class, and then create our instance attributes.

Here we are going to define the first level of the Conv2D structure, respectively it must have 4 parameters, namely the input channels (3), the output channels (16), the size of the kernel (3x3) and the padding (1) this is to avoid that after the transition from input to output you lose 1 pixel in the edge, so to avoid that after each layer the image is progressively reduced we add pixels to the edges of the channel.

Let's define the linear layers (view) we use fc1/2 (full connected layer) i.e. that all neurons are connected. Sending 25088 as input (as in the diagram) and 32 as output. In the second we have 32 inputs and 2 outputs (Last layer of the schema), these outputs will represent one-dimensional vectors and the first element shows the probability that an image shows a cat, while the second the probability that it shows a dog.

```
def forward(self, x):
    out = self.pool(F.relu(self.conv1(x)))
    out = self.pool(F.relu(self.conv2(out)))
    out = out.view(out.size(0), -1)
    out = F.relu(self.fc1(out))
    out = self.fc2(out)
    return out
```

The forward method represents the Forward Pass (Page 2). We pass the parameters self and x (image of the cat).

Now let's get into the last steps, then let's pass the first linear layer to the RELU, then let's move on to the last linear layer. Finally we come back out.

Now let's flatten our images into a single dimension by calling the view method to out, let's call the size method.

We need to apply the first layer to the input

we come back out.

We need to apply the first layer to the input and make it enter the RELU (Rows 2 and 3). Exited the first RELU must enter the pool (self.pool), now the image will no longer be 224x224 but will have been halved by the Max Pool.

In the next cell we are going to resize the image
to take the trading data from the folder in the root and then divide the dataset into two parts, training data and validation data.

We resize the image and turn them into **tensors**

```
transform = transforms.Compose([
    transforms.Resize((224, 224)),
 transforms.ToTensor(), # Let's turn the image into a tensor
])
# We draw training data
dataset = datasets.ImageFolder(root='dataset', transform=transform)
print(dataset.class_to_idx)
# We divide the dataset into training and validation
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = torch.utils.data.random_split(dataset, [train_size,
val_size])
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
```

We define the path to the dataset folder and print which classes it found in the

We randomly divide

Here we do a split and divide it into training data and validation data. We divide it so that we can interrogate the model and evaluate it, to see if it performs well.

1

Let's create the DataLoaders, i.e. pytorch tools that are used to load and pass data progressively to the training cycle, dividing them into batches so as not to pass them all (heavier model and would not be able to generalize since it would not see so much data changing). We use the shuffle property that mixes the data at each era.

Example: We give a student exercises (training data), but we keep exercises to ourselves to evaluate him, because by giving all the exercises he learns specifically and cannot generalize, not really understanding what happens so we take out a part of the data and question it.

2

Let's instantiate the model, using the Net class declared earlier, pass the model to the device (GPU or CPU) and set it to train mode. We also define the loss function and the optimizer

```
model = Net()
model = model.to(device)
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr = 0.001)
model.train()

for epoch in range(15):

    running_loss = 0.0# Represents the loss of the single batch
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
```

```
running_loss = 0.0# Represents the loss of the single batch
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
outputs = model(images) # First time we pass images over the network
        loss = loss_fn(outputs, labels) # Calcoliamo la loss
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    avg_loss = running_loss / len(train_loader) # Loss media su tutti i batch
    print(f"Epoch {epoch}, Loss: {avg_loss}")
```

Here we carry out the training cycle as shown in chapter 2, with slight differences. In this cycle we define a running loss variable, i.e. how much loss there is in the single batch (sample of data), while in the second cycle we go to say that for each image and each label (whether it is a cat or a dog) taking them from our train loader must carry out the training cycle (Chapter 2). The first step of the cycle passes the images and labels to the GPU/CPU, the output variable goes to analyze our images through our forward (net class forward method), so now our model will give us a predicted y, we go to calculate the loss function, we get the gradients with the backpropagation, we update the parameters and we add the loss item to running loss.
In the line below we are going to make an average of loss by printing it together with the epochs.

Now starting the model will start training and start learning image data.

```
Epoch 0, Loss: 0.6860524713993073
Epoch 1, Loss: 0.6737722635269165
Epoch 2, Loss: 0.647117656469345
Epoch 3, Loss: 0.6017773270606994
Epoch 4, Loss: 0.5721716940402984
Epoch 5, Loss: 0.5076314270496368
Epoch 6, Loss: 0.4637900412082672
Epoch 7, Loss: 0.41112600862979887
Epoch 8, Loss: 0.3859466940164566
Epoch 9, Loss: 0.2936780959367752
Epoch 10, Loss: 0.24542222470045089
Epoch 11, Loss: 0.20702120363712312
Epoch 12, Loss: 0.14328788593411446
Epoch 13, Loss: 0.13495455980300902
Epoch 14, Loss: 0.08506143242120742
```

Here we visualize the loss function in the various eras.

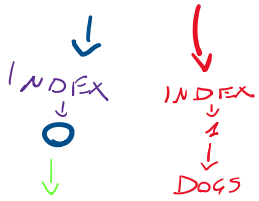
# Image classifier inference

After instantiating the model, trained on an image dataset, we now go to visualize the model's response by passing it images, which portray either a dog or a cat, the model will return a percentage.

```
image = Image.open('dataset\\cats\\cat (3).jpg').convert('RGB')
image_tensor = transform(image).unsqueeze(0).to(device)
model.eval()
with torch.no_grad():
    output = model(image_tensor)
    probs = torch.softmax(output, dim=1)
print(output)
print(f"{probs*100}%")
```

In this block of code we open the image, with the Image.Open(path) method, converting it to RGB, then a 3-channel input (as defined in the architecture). Then we call our transform method on the image, add the size with unsqueeze, and pass it to the GPU.

```
tensor([[ 0.9958, -4.2078]],
device='cuda:0')
tensor([[99.4533,  0.5467]],
device='cuda:0')%
```




Here, however, we are going to set the model in evaluation mode.
With the with torch.no_grad() method we are going to say that we don't want gradients because we are in evaluation mode, not training mode.
In the following lines we are going to switch the image tensor to the model.
Finally, we print the probabilities

The output we see is the probability that the image passed (cat (3).jpg) is a cat or a dog. Pytorch divides our dataset into classes and divides each one by indexes, in our case 0 is the index of the dataset containing cats and 1 is the index of the dataset containing dogs.

By passing the image of a cat we get on the index 1 99.3469% probability that it is a cat and 0.6531% that it is a dog. The model responds perfectly

# Model Persistence: Saving the "Weights"

A newly initialized neural network is "dumb": its weights (the numbers in the matrices) are random. All the intelligence acquired during the hours of training lies in the precise adjustment of these numbers.
However, RAM memory is volatile. If we close the Python program, we lose all the work done. To use the model in the future without retraining it, we need to save its state.
**State Dict:** In PyTorch, the "brain" of the model is stored in a dictionary called state_dict. This maps each layer of the network (e.g. conv1, fc1) to its parameters (weight and bias tensors).
**Serialization:** Saving the template means writing this dictionary on a physical file (usually with a .pth or .pt extension).
**Future inference:** When we want to use AI in a month, we will only need to instantiate an "empty" network (with random weights) and overwrite them by uploading the .pth file. In an instant, the network will recover all its "experience".
Here's how to save the weights of a model.

```
torch.save(model.state_dict(), "image_classifier.pth")
```

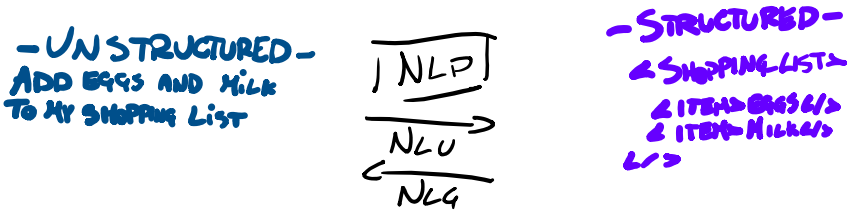
To load weights instead:

```
model.load_state_dict(torch.load("C:\\Users\\rdarc\\Desktop\\Reti Neurali
\\Codice\\image_classifier.pth", weights_only=True))
```

# Chapter 2: NLP (Natural Language Processing)

In chapter 1 we worked with images, i.e. static data organized in a 2D space. CNNs extracted spatial features where neighboring pixels are the most important.
Natural Language (text), on the contrary, is a sequential datum. Information is not static: word order is everything. For example, "The cat eats the mouse" is different from "The mouse eats the cat".

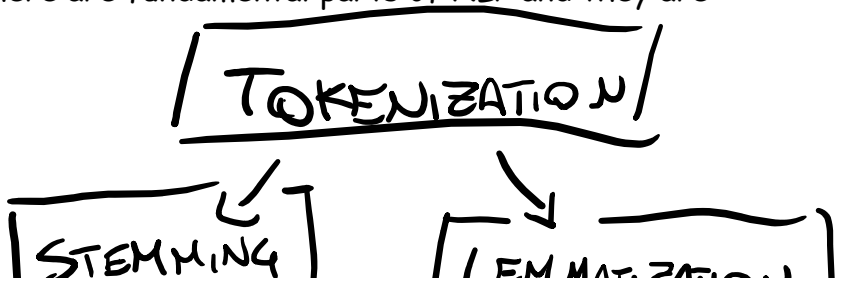
NLP, or Natural Language Processing, is a branch of artificial intelligence that allows computers to understand, interpret, and generate human language.
NLP uses a combination of computational linguistics, machine learning, and deep learning to analyze language. Through a series of steps, such as tokenization (splitting text into units), syntactic analysis, and semantic disambiguation, machines "digest" language to extract meaning from it.

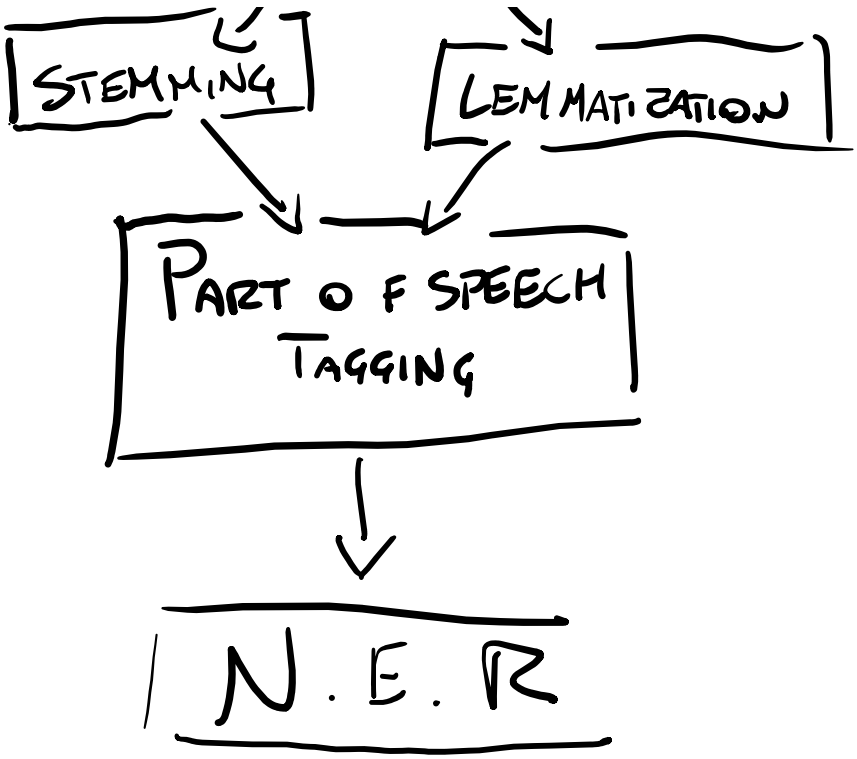



We have two types of data, structured and unstructured and our own NLP it resides right in the middle, translating from unstructured to structured data, when we go from structured to unstructured data it is called NLU (Natural Language Understanding), while vice versa it is called NLG (Natural Language Generation). There are several cases where NLP comes in handy:

1) Machine Translation, language translators
2) Virtual Assistant/ Chatbot, Siri o Alexa
3) Sentiment Analysis, as if we were evaluating whether a review is positive or not
4) Spam Detection, such as whether an email is potential spam or not

There are fundamental parts of NLP and they are




STEMMING
LEMMATIZATION
PART OF SPEECH TAGGING
N.E.R

# Tokenization

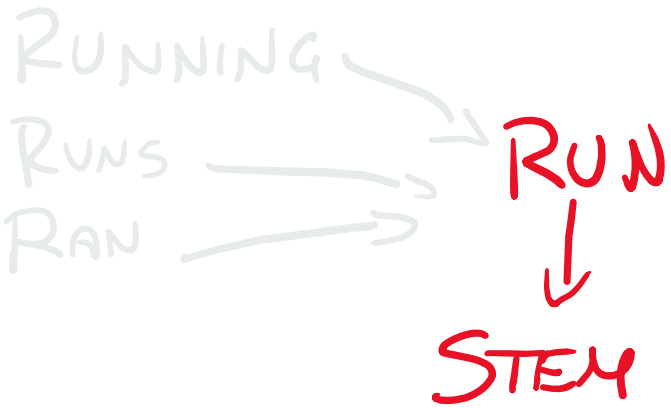
The first step of NLP is called tokenization, its operation is to take a string of unstructured data and break it into pieces, if we consider the phrase above "add eggs and milk to my shopping list", there are 8 words and they will be 8 tokens (Word Tokenization) or divides the text into single words, used by chatbots and virtual assistants.

"ADD EGGS AND MILK TO MY SHOPPING LIST"
["ADD", "EGGS", "AND", ...]

Character Tokenization: Segments text into individual characters. Useful for languages without clear word boundaries or for tasks that require granular analysis.
Subword Tokenization: Breaks text into units that are larger than characters but smaller than words. Useful for languages with complex morphology or for managing words out of vocabulary.

# Stemming & Lemmatization

Stemming and lemmatization are a step in text mining pipelines that convert raw text data into a structured format for automated processing. Both derivation and lemmatization eliminate affixes from inflected forms of words, leaving only the root.
Stemmers eliminate word suffixes by comparing incoming word tokens to a predefined list of common suffixes.

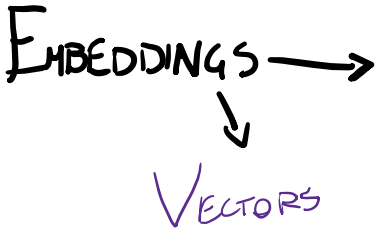



But stemming doesn't work well for every token, for example:


University and Universal, for example, do not derive from Universe. For these specific cases we have lemmatization.
Lemmatization takes the token and learns its meaning through dictionary definition, since lemmatization aims to produce basic forms of the dictionary, it requires a more robust morphological analysis than stemming.

# Embedding

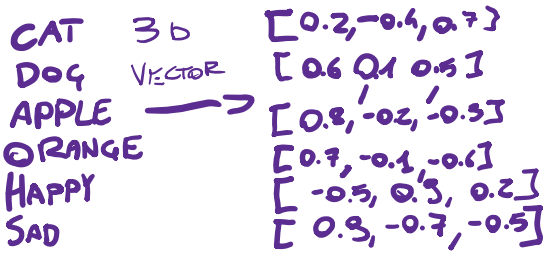
Word embeddings are embeddings of words, they are a way to represent words as vectors in a multidimensional space, where the distance and direction between vectors reflect the similarity and relationships between the corresponding words



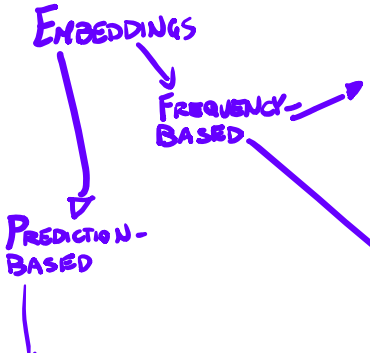

These embeddings are very present in NLP, Specifically in text classification and N.E.R, also for example in Spam Detection. They help with word similarity and word analogy tasks, another example being Q&A.

What is a Sliding Context Window? A method of processing continuous data streams by moving a fixed-size "window" over the data

Word embeddings are created by models trained on a large corpus of text, for example on the whole of Wikipedia.
The process starts from pre-processing the text such as Tokenization, Stemming & Lemmatization. A scrollable context window identifies words that help context by allowing the model to learn the relationships between words. The model is trained to predict based on context, semantically place similar words close to each other in multidimensional space, parameters are adjusted to minimize prediction errors. What does it look like?




Then each dimension has a numeric value and creates a 3D vector for each word. These values represent the positions of words in a continuous 3d vector space. These words have similar meaning or context, and each has a similar vector representation. For example, Apple and Orange represent a semantic relationship with each other, and the values of the vectors reflect this relationship. On the other hand, if we notice Happy and Sad, they have opposite directions in space due to their contrasting meaning. Of course, current word embeddings have hundreds of sizes not just three, this allows for more intricate relationships and nuances of meaning to be captured.
There are two basic approaches to how word embedding generates word representations.




Frequency-based embeddings are derived from the frequency of words in a body of text. They are based on the importance or meaning of a word, which can be deduced from how frequently a word appears in the text.

One of these frequency-based is called TF-IDF (Term Frequency Inverse Document Frequency). TF-IDF mentions words that are frequent within a specific document but are rare in the entire corpus

BASED

They capture the semantic and contextual relationships between words. For example in the sentence "the dog is barking loudly" and "the dog is wagging its tail", this embedding pattern associates the word "dog" with "bark", "tail". This allows these models to create a single still representation of a dog. These models excel at separating words with a neighboring meaning and can handle the various senses in which the word can be used.

One of these frequency-based is called TF-IDF (Term Frequency Inverse Document Frequency). TF-IDF mentions words that are frequent within a specific document but are rare in the entire corpus.
For example, if we have a document on coffee, it will highlight words such as espresso or cappuccino that often appear in that document but rarely in other topics. While words like "the", "and" that almost always appear in documents will have low scores in the TF-IDF.

There are various templates to generate word embeddings but the most famous is the "Word2vec" developed by Google in 2013, this template has two architectures.

WORD2VEC
- CBOW
- SKIP-GRAM

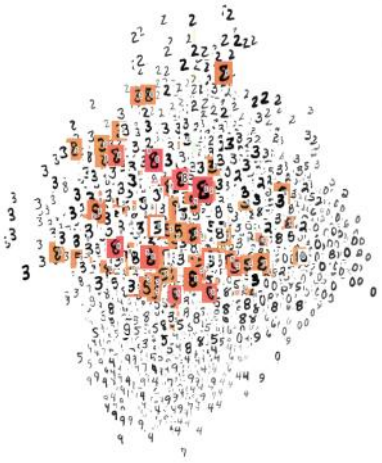
The CBOW predicts a target word based on the words of the surrounding context.

Skip-Gram does the opposite, that is, it predicts the context words, given by a target word

Another very popular model is the GLOVE, which stands for Global Vectors for Word Representation, created by Stanford University in 2014. Use statistical co-occurrence to create word vectors.

These models are still considered very valid in NLP. But while traditional word embeddings assign a fixed vector for each word, transformers models use another type of embedding called Contextual Based Embedding. This model changes the representation of the word based on the surrounding context. For example, the word "bank" has different representations in the sentence. This "context sensitivity" allows these patterns to capture many nuances of meaning and relationships between words. This model has given many improvements in various changes of NLP.

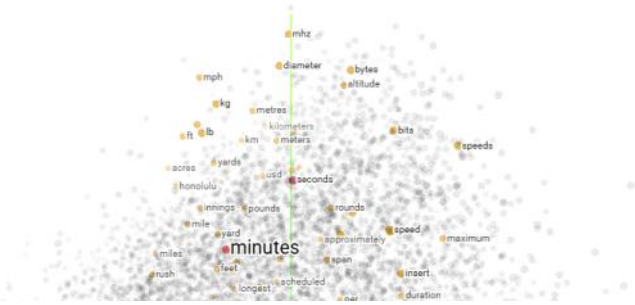
Co-occurrence: frequency with which two or more words appear together in a text or corpus, indicating a strong semantic or contextual relationship between them.



This figure represents a vector space applied to the image database of an MNIST Classifier, as we can see the orange images are the ones "close" to each other precisely because they have the same number.
We have the same thing with words that are placed, however, in a semantic map.

Here instead we can see it with words, I highlighted here the word minutes (I used the 10,000-point Word2vec), we can see that the words close to the word minutes have a semantics close to that of the word "minutes". For example:

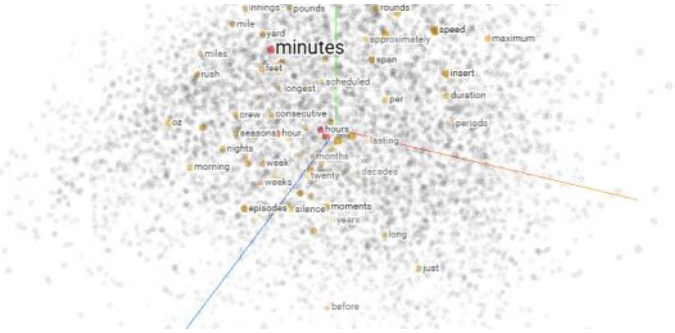
- Pounds (units of measurement)
- Mph, ft, lb, km, meters
- Long duration, years hours



example:

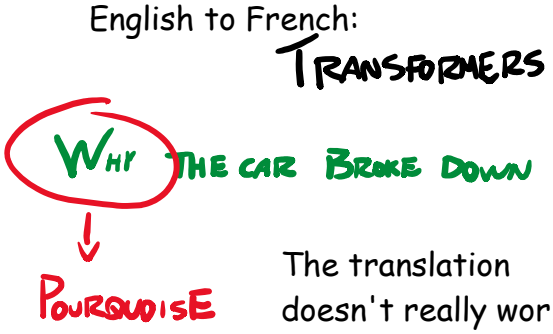
- Pounds (units of measurement)
- Mph, ft, lb, km, meters
- Long, duration, years hours

These are all words that have a meaning close to that of the word.

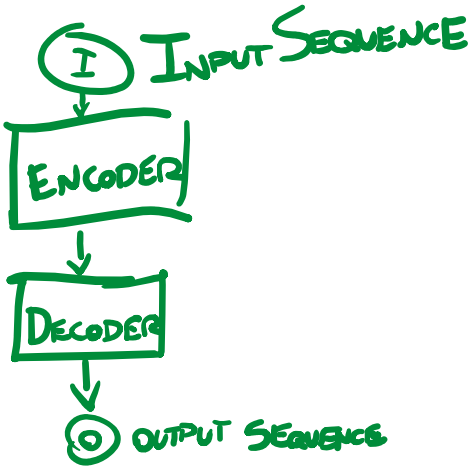


# Transformers

The task of the transformers is literally to transform one sequence into another sequence. For example, let's take a sentence to be translated from English to French:

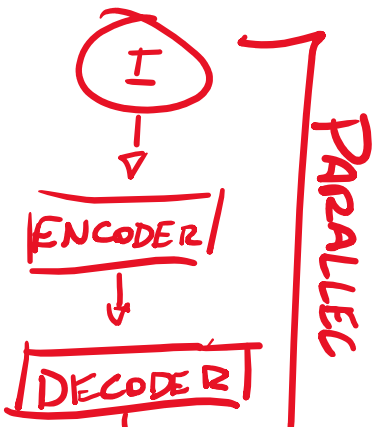


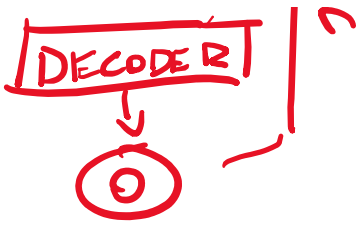

The translation doesn't really work like that but it's an example of transformers.




The work of transformers is done through sequence-by-sequence learning, where transformers take a sequence of tokens, in this case our sentence, and then predict the next word in the output sequence.
This happens by iterating through all encoder layers and passing these encodings to the next encoder layer. The decoder takes these encodings and uses their derived context to generate the output sequence.
Transformers have a semi-supervised type of training. By this term we mean that they have been pre-trained in an unsupervised manner then have passed a Fine-Tuning phase through supervised learning to improve their performance.

Transformers use an "Attention Mechanism" and this provides context around objects in the input sequence, with the example above the translation starts from "why" because it is the beginning of our sentence, but transformers try to identify the context that gives meaning to each word in the sentence. It is precisely this mechanism that gives transformers a step further than RNN Neural Networks that perform tasks in sequence: transformers perform various sequences in parallel.




DECODER
0

# Positional Encoding

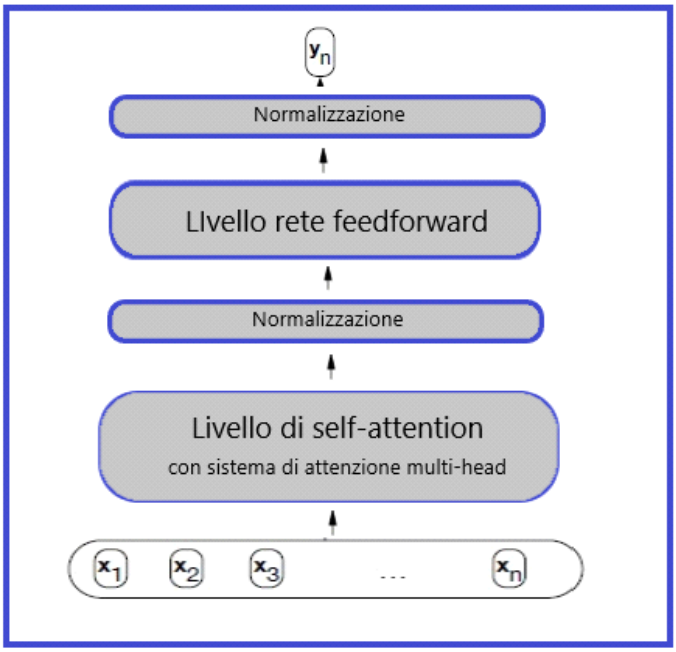
We've seen that Embeddings turn words into meaning-based vectors of numbers. "Cat" will be mathematically close to "Happiness".
However, there is a huge problem: the Transformer (the architecture we will use) does not care about the order.
If we pass the pure embedding vectors, for the network these two sentences are identical:

1. *"The dog bites the man"*
2. *"Man bites the dog"*

Why? Because unlike the old networks (RNNs) that read one word at a time in sequence, the Transformer reads everything together in parallel. It's like a teacher who corrects all the assignments at the same time instead of one at a time: he loses the sense of who turned in first.
The solution would be to add the Position we have to "stamp" each word with a number that indicates its position.
We don't add a new column, but add a position vector to the word vector.
$Input = Embedding(Parola) + Embedding(Posizione)$
(Imagine a graph where the vector of the word "Dog" is added to a vector representing "Position 2". The result is a slightly modified vector that contains both info).
Implementation in PyTorch.
In modern GPT models, instead of using complicated sinusoidal functions (as in the original 2017 paper), Learnable Embeddings are often used. The network learns on its own what is the best vector to represent "position 1", "position 2", etc.

# Self-Attention

The transformer is based on the self-attention technique that allows networks to analyze and weigh the importance of each part of a data sequence compared to all the other parts of the same sequence.
This is especially useful for understanding the context of a word in a sentence, as **a word can affect or be influenced by any other word,** regardless of distance in the sequence. Each word or symbol in the sequence is converted into a **numeric vector (embedding)** that represents its meaning in a multidimensional space.




Blocco del Transformer

**HumAI.it**

The self-attention process can be divided into:

1. Calculating attention scores
    a. For each word, we calculate the scalar product between its Query vector and the Key vectors of all the words in the sequence. This gives us a measure of how relevant each word in the sequence is to the current word.

    $$score(Qi, Kj) = \frac{Qi * Kj}{\sqrt{dk}}$$

    Where dk is the size of the Key vector, and the division for $\sqrt{dk}$ is used to stabilize gradients during training.
2. Application of Softmax:

a. We convert scores to probabilities using the softmax function.

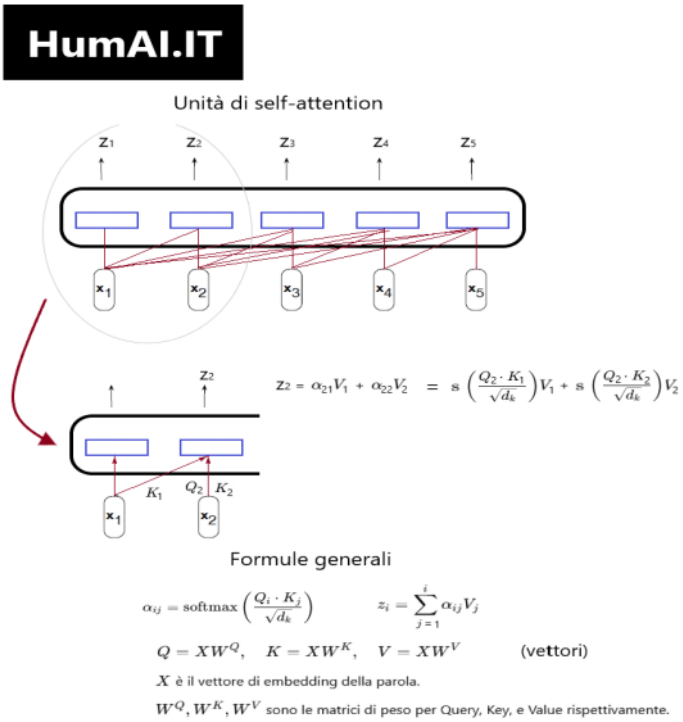
$$aij = softmax\left(\frac{Qi * Kj}{\sqrt{dk}}\right)$$

The aij represent the weight of attention that the word i gives to the word j.

3. Combination of values (V):
   a. We use these attention weights (aij) to make a weighted average of the Value(V) vectors. This gives us the new representative of the current word by considering the context of all words.

$$zi = \Sigma aijVj$$

The zi vector is a combination of the values Vj where the weights are determined by the relevance calculated using the Query and Key vectors.



# Project 3: Sentiment Analysis

After understanding how a network can understand text and natural language, now let's create a model capable of taking a sentence and understanding whether the purpose of this sentence is positive or negative.
Use case: "Today the concert was amazing!", the phrase has a positive purpose.
Use case 2: "I can't come to the concert today", the sentence has a negative purpose.
We will use the tiktoken library to carry out the tokenization.
The workflow is this:

1. Create the list of template parameters.
2. Prepare data on those work.
3. Go to tokenize.
4. Create the batches for training.
5. Train the model.
6. Instantiate the model with its layers.
7. Model inference.

These are the steps to follow to carry out this first project on the processing of the text.

```
import torch.nn as nn
import torch.optim as optim
import tiktoken
config = {
    "embed_dim": 128,
    "num_heads": 8,
    "max_len": 100,
    "num_classes": 1,
    "batch_size": 2,
    "num_epochs": 5,
    "LR": 0.001
}
encoder = tiktoken.get_encoding("cl100k_base")
```

With these lines we are going to define various parameters and initialize the encoder for the tokenizer.
Embed_dim, represents the size of the vector that describes each token after embedding.
Num_heads, the number of heads for attention,
Max_len, is the maximum number it can receive and output.
Num_classes, 1 class for the answer (label).
Batch_size, the batch number in the training.
Num_epochs, the number of eras.
Lr, is the learning rate.
Perfect, now let's move on to the preparation of the data and examples.

```
{"text": "I really liked the movie", "label": 1},
```

This is what a row of our dataset looks like, so let's create a series of examples and assign each of them a "label" this label will help us understand if our sentence is positive or negative (1 positive 0 negative).
So let's prepare the examples and send them to the encoder.

```
import json
import torch
with open("train_data.json", "r", encoding="utf-8") as f:
    data = json.load(f)
texts = []
labels = []
for item in data:
    texts.append(item["text"])
    labels.append(item["label"])
inputs = []
targets = []
for text, label in zip(texts, labels):
    tokens = encoder.encode(text)
    if len(tokens) < config["max_len"]:
```

```
        tokens += [0] * (config["max_len"] - len(tokens))
    else:
        tokens = tokens[:config["max_len"]]
    inputs.append(tokens)
    targets.append(labels)
inputs = torch.tensor(inputs, dtype=torch.long)
targets = torch.tensor(labels, dtype=torch.float)
```

In this code snippet we go through the following steps:

1. Let's open the json and load the content into a given variable.
2. Then we initialize two variables in which we will hang the two types of data, i.e. the sentence contained in texts and the label therefore positive or negative, all this for each sentence in date.
3. Then we initialize two other variables that have the same goal actually but must receive the final data in this case, inputs will receive the tokenized phrases while targets the labels that are already numbers.
4. Then in a for we say that for each text and each label, if the length of the sentence is less than the maximum length declared in the configuration it adds 0 to the end of the token, otherwise it limits the length.
5. We add tokens to the input list and labels to targets and transform them into tensors to pass to the model.

After this step we have batching and shuffling.
During training, data is **randomly scrambled before being fed** to the model.
Shuffling is used to prevent the model from learning dependencies related to the order of the data in the dataset. If positive or negative sentences were grouped, the model could fit that order instead of the content of the text.
By scrambling the data at each epoch, each batch contains different and more representative examples of the real distribution.
Then we create the batches that we will need in training.

```
batch_size = config["batch_size"]
dataset_size = inputs.size(0)
indices = torch.randperm(dataset_size)
shuffled_inputs = inputs[indices]
shuffled_targets = targets[indices]
for start_idx in range(0, dataset_size, batch_size):
    end_idx = start_idx + batch_size
    batch_inputs = shuffled_inputs[start_idx:end_idx]
    batch_targets = shuffled_targets[start_idx:end_idx]
```

The batch size is a gradient descent hyperparameter that controls the number of training samples to be processed on before the model's internal parameters are updated.

Let's take the two "measures" that we will need, that of the dataset and that of the batch, we mix the indexes with randperm and assign them to new mixed variables.

Let's continue with the definition of all the layers.

```
import torch.nn as nn
import torch
class Model(nn.Module):
    def __init__(self, vocab_size, embed_dim, num_heads, max_len = 100):
 super().__init__()

        self.embedding = nn.Embedding(encoder.n_vocab, embed_dim)

        self.pos_embedding = nn.Parameter(torch.randn(1, max_len, embed_dim))

        encoder_layer = nn.TransformerEncoderLayer(
            d_model = embed_dim,
            nhead = num_heads,
            batch_first = True
)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer, num_layers = 1)

        self.fc = nn.Linear(embed_dim, 1)

    def forward(self, x):
        x = self.embedding(x)

 seq_len = x.size(1)
```

```
        x = x + self.pos_embedding[:, :seq_len, :]

        x = self.transformer_encoder(x)

        x = x.mean(dim=1)

        x = self.fc(x)

        return x
model = Model(
encoder.n_vocab,
 config["embed_dim"],
    config["num_heads"],
    config["max_len"]
)
```

We create a class called a template where we use pytorch to initialize the layers and architecture of the template.

1. Embedding layer
2. Positional embedding
3. Transformer Layer
4. Output Layer

In the forward we pass our data in various layers, first we embedding it, we add the position information and then we pass everything to the transformer.
Now let's move on to training our model following the classic cycle. Training cycle

```
criterion = torch.nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = 0.001)
num_epochs = 20
for epoch in range(num_epochs):
    indices = torch.randperm(dataset_size)
    shuffled_inputs = inputs[indices]
    shuffled_targets = targets[indices]
    for start_idx in range(0, dataset_size, batch_size):
        end_idx = start_idx + batch_size
        batch_inputs = shuffled_inputs[start_idx:end_idx]
        batch_targets = shuffled_targets[start_idx:end_idx]
        optimizer.zero_grad()
        outputs = model(batch_inputs)
        loss = criterion(outputs.squeeze(), batch_targets)
        loss.backward()
        optimizer.step()

    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {loss.item():.4f}")
OUTPUT:
Epoch 1/20, Loss: 0.6868
Epoch 2/20, Loss: 0.6552
Epoch 3/20, Loss: 0.5603
Epoch 4/20, Loss: 0.2165
Epoch 5/20, Loss: 0.0920
Epoch 6/20, Loss: 0.2450
Epoch 7/20, Loss: 0.0115
Epoch 8/20, Loss: 0.0076
Epoch 9/20, Loss: 0.0062
Epoch 10/20, Loss: 0.0056
Epoch 11/20, Loss: 0.0047
Epoch 12/20, Loss: 0.0040
Epoch 13/20, Loss: 0.0043
Epoch 14/20, Loss: 0.0046
Epoch 15/20, Loss: 0.0034
Epoch 16/20, Loss: 0.0035
Epoch 17/20, Loss: 0.0029
Epoch 18/20, Loss: 0.0037
Epoch 19/20, Loss: 0.0027
Epoch 20/20, Loss: 0.0028
```

Now let's move on to inference:

```
new_sentence = "The concert was beautiful"
tokens = encoder.encode(new_sentence)
input_tensor = torch.tensor([tokens], dtype=torch.long)
model.eval()
with torch.no_grad():
    output = model(input_tensor)
    prob = torch.sigmoid(output)
```

```
if prob.item() > 0.5:
    print("Prediction: Positive, Probability:", prob.item())
else:
    print("Prediction: Negative, Probability:", prob.item())
OUTPUT:
Prediction: Positive, Probability: 0.5992962718009949
```

We have now completed our first text-processing model.
Here you will find the project files.
https://github.com/robworkhubb/from-neurons-to-llm

# Conclusion and Updates

venerdì 19 dicembre 2025 10:35

This book is not designed as a point of arrival, but as a solid base from which to start.
Machine learning and neural networks are constantly evolving fields, and stopping would mean betraying the very spirit of this discipline.
For this reason, the content will be **updated over time**:
New examples, clarifications, more advanced sections, and fixes will arise as my experience and technology landscape grow.
Updates, technical discussions and previews of new content will be shared on my **DevOrbit Discord**, which I will use as a reference point for those who want to continue learning, comparing and building real projects.
If you've come this far, you've not just read a book.
You have followed a path of understanding. And this is where the real work begins.
https://discord.gg/edxVahR2kU
Signature: