Apprentissage Statistique Automatique II

Lucas Reding, Thibault Defourneau

Lecture note

Contents

1 Introduction

Deep learning is a set of learning methods attempting to model data with complex architectures combining different non-linear transformations. The elementary bricks of deep learning are the neural networks, that are combined to form the deep neural networks.

These techniques have enabled significant progress in the fields of sound and image processing, including facial recognition, speech recognition, computer vision, automated language processing, text classification (for example spam recognition). Potential applications are very numerous. A spectacularly example is the AlphaGo program, which learned to play the go game by the deep learning method, and beated the world champion in 2016.

There exist several types of architectures for neural networks :

- The multilayer perceptrons, that are the oldest and simplest ones
- The recurrent neural networks, used for sequential data such as text or times series.
- The Convolutional Neural Networks (CNN), particularly adapted for image processing
- The LLMs, which now widely used in various applications

They are based on deep cascade of layers. They need clever stochastic optimization algorithms, and initialization, and also a clever choice of the structure. They lead to very impressive results, although very few theoretical fondations are available till now.

2 Tensorflow

2.1 Basics

Tensorflow is a framework developped by Google Brain which has been designed for Machine learning. It is the core tool behind some well-known services like Google Cloud Speech, Google Photos and Google Search. In the following lines of code, we present the main component of this framework:

```
import tensorflow as tf

# High level of Deep learning API
tf.keras

# Low level of Deep learning API
tf.nn

# Non-exhaustive list of numpy like packages
tf.math
tf.linalg
tf.signal
tf.random
tf.bitwise

# Data processing
tf.data
tf.audio
tf.images
tf.io
tf.queue

# Optimization and Deployment
tf.distribute
tf.saved_model
tf.autograph
tf.graph_util
tf.lite
tf.quantization
tf.tpu

# Special data structure
tf.lookup
tf.nest
tf.ragged
tf.sets
tf.sparse
tf.strings
```

Remark 1. *In the lowest level, each Tensorflow operation is implemented in C++. Moreover, may operations have several implementations, called kernels. Each kernel is dedicated to a specific type of processor like CPU, GPU or*

even TPU (Tensor Processing Unit).

Using Tensorflow is a similar experience than using the package Numpy. However the naming of basic operations may differ. We are now comparing some similar methods between Tensorflow and Numpy:

```
import numpy as np
import tensorflow as tf

# Matrix definition
tensor = tf.constant([[1, 2, 3], [4, 5, 6]]) # in Tensorflow
arr = np.array([[1, 2, 3], [4, 5, 6]]) # in Numpy
arr_from_tensor = tensor.numpy() # It is possible to retrieve the numpy version of a tensor

# Matrix multiplication
tensor @ tf.transpose(tensor) # in Tensorflow
arr.dot(arr.T) # in Numpy

# Mean computation
tf.reduce_mean(tensor) # in Tensorflow
np.sum(arr) # in Numpy
```

Remark 2. *In the above examples, the variable tensor is a $tf.Tensor$ object which is immutable. However, we need an object which can be modified in some context, as during backpropagation step. This is why the object $tf.Variable$ exists.*

```
import numpy as np
import tensorflow as tf

var = tf.Variable([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
arr = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])

var.assign(2 * v) # In Tensor, it replaces var with 2 * var using assign method
arr = 2 * arr # In numpy, we keep the affectation '='
```

Exercises 1. *Write the Huber loss using Tensorflow using only the basic operations. The Huber loss is defined as follows:*

$$Huber(x) = \begin{cases} \frac{1}{2}\,x^2 & , if\,|x| \leq 1 \\ |x| - \frac{1}{2} & , if\,|x| > 1 \end{cases}$$

2.2 Loss wrapper

Defining a custom loss is possible by using the base class $tf.keras.losses.Loss$, with two methods to customize:

- The method *call* gets the targets and the predictions for calculating all the losses during the training step;
- The method get_config returns a dictionary which associates each hyperparameter name to its value.

Let's define the Huber loss with this approach:

```
import tensorflow as tf

class HuberLoss(tf.keras.losses.Loss):
    def __init__(self, threshold = 1.0, **kwargs):
        self.threshold = threshold
        super().__init__(**kwargs)

    def call(self, y_true, y_pred):
        # The logic already defined in the first exercise
        return computed_loss

    def get_config(self):
        base_config = super().get_config()
        return {**bas_config, "threshold": self.threshold}
```

This way of defining a custom loss is very practical, because all necessary abstractions have already been taken into account in the base class $tf.keras.losses.Loss$. In particular, it is easy to save the hyperparameters of the custom loss function when saving the trained model:

```python
import tensorflow as tf

# Let `model` be any Tensorflow model. When compiling it with the loss function, it is enought to write:
model.compile(loss=HuberLoss(2,), optimizer="nadam")

# When loading the trained model, it is important to specify the custom loss as well:
model = tf.keras.models.load_model("my_model_with_the_custom_huber_loss", custom_objects={"HuberLoss": HuberLoss})
```

2.3 Regularizer wrapper

Similarly to the custom loss, it is possible to define a custom regularizer by using the base class $tf.keras.regularizers.Regularizer$. Let's take an example by defining the l_1 regularization:

```python
import tensorflow as tf

class HuberLoss(tf.keras.regularizers.Regularizer):
    def __init__(self, factor):
        self.factor = factor

    def __call__(self, weights):
        return tf.reduce_sum(tf.abs(self.factor * weights))

    def get_config(self):
        return {"factor": self.factor}
```

Remark 3. *As we can observe, there are some differences between a custom loss and a custom regularizer:*

- *We don't need to invoke the parent builder, as well as the parent configuration.*
- *We need to define the __call__ method in the class method instead of call method.*

2.4 Metric wrapper

```python
import tensorflow as tf

class HuberMetric(tf.keras.metrics.Metric):
    def __init__(self, threshold=1.0, **kwargs):
        super().__init__(**kwargs)
        self.threshold = threshold
        self.huber_fn = create_huber_(threshold)
        self.total = self.add_weigth("total", initializer="zeros")
        self.count = self.add_weigth("count", initializer="zeros")

    def update_state(self, y_true, y_pred, sample_weight=None):
        sample_metrics = self.huber_fun(y_true, y_pred)
        self.total.assign_add(tf.reduce_sum(sample_weights))
        self.count.assign_add(tf.cast(tf.size(y_true), tf.float32))

    def result(self):
        return self.total / self.count

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}
```

Let's explain the above code:

- The builder $tf.keras.metrics.Metric$ uses the method add_weight in order to generate variables for computing the metric over several batches.
- The method *update_state* updates the variables from targets and predictions from a single batch.
- The method *result* provides the final output, which is here the mean of Huber metric over all batches.
- The builder also provides the method *reset_states* which initializes again all the variables to 0.0. It can be customized if necessary.

2.5 Custom layer with Keras

```python
import tensorflow as tf

class CustomDenseLayer(tf.keras.layers.Layer):
    def __init__(self, units=None, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.activation = tf.keras.activations.get(activation)

    def build(self, batch_input_shape):
        self.kernel = self.add_weight(name="kernel", shape=[batch_input_shape[-1], self.units],
        ↪  initializer="glorot_normal")
        self.biais = self.add_weight(name="biais", shape=[self.units], initializer="zeros")

    def call(self):
        return self.activation(X @ self.kernel + self.biais)

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "units": self.units, "activation": tf.keras.activations.serialize(self.activation)}
```

Exercises 2. *Implement a custom layer performing a normalization:*

- *The method build should have two trainable weights of type* $tf.float32$ *called* α *and* β *respectively initialized to* 1 *and* 0.
- *The method call must compute the mean* μ *and the standard deviation* σ *for each instance, by using the method* $tf.nn.moments(inputs, axes = -1, keepdims = True)$. *The final output should be* $\alpha \otimes (X - \mu)/(\sigma + \epsilon) + \beta$, *where* $\otimes$ *is the pointwise multiplication and* ϵ *equals to* 0.001.
- *Check if your custom implementation produces the same result as* $tf.keras.layers.LayerNormalization$.

3 Regularization

Training an neural network is an optimization problem. Suppose that, for a given set of parameters W embedded into an euclidean space, we have a neural network (hence a specific non-linear smooth map) $f(-;W) : \mathbb{R}^n \to \mathbb{R}^p$. For determining the parameters W living in some euclidean space, we need:

1. A set of training points $\{(x_i, y_i)\}_{1 \leq i \leq N}$;
2. A loss function $L : \mathbb{R}^n \times \mathbb{R}^p \to \mathbb{R}$ which smooth at least at each x_i;

Hence the expected parameters are those minimizing the following empirical expectation:

$$W^* = argmin_W \Big(\frac{1}{N} \sum_{i=1}^{N} L\big(f(x_i; W),\, y_i\big) \Big) \tag{1}$$

Remark 4. *It is important to note that this is not a convex optimization problem in general, and therefore there is no unique solution.*

Exercises 3. *Let $W \in \mathbb{R}^{p \times n}$ be a matrix, and $f(-;W) : \mathbb{R}^n \to \mathbb{R}^p$ the linear map defined by $f(x;W) = Wx$. Imagine that we found W^* such that it minimizes the following empirical expectation of the hinge loss:*

$$\frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} max\big(0, (f(x_i;W))_j - (f(x_i;W))_{y_i} + 1\big)$$

Is W^* unique ?

Remark 5. *The main drawbacks of this training strategy is that:*

- *It may lead to an overfitting problem, i.e. the trained neural network fits too well with the training data points, so that it won't be able to predict correctly on new data points.*
- *It eventually lead to an underfitting problem, i.e. the trained model network doesn't manage to capture enough patterns from training data points.*
- *It introduces more uncertainty, i.e. variance because of wide amplitude of some weights.*

Overfitting and underfitting problems are quite common when training a deep neural networks. A way to check when such model tend to "overfit" or "underfit" is to:

1. Split the overall data into training data, validation data and test data.
2. Track the the evolution of losses on both training data and validation data.
3. Evaluate the trained model on test data.

The below figure shows you how to identify quickly these problems:

Remark 6. *During the training step, the best model can be obtained by stopping the training of the model when the loss or metric on validation data points start to increase. This is called the early stopping strategy.*

A strategy to solve the overfitting issue is to implement a regularization strategy. This can be done following three main approaches:

1. Adding a regularization term to the optimization problem when minimizing the empirical expected loss.
2. Reducing the number of parameters in the neural network, for instance by using *Dropout*.
3. Transforming the data before or during the training process.

3.1 Regularization term

Adding a regularization term is the most common strategy for preventing from overfitting. Concretely, instead of solving (1), we try to solve the following optimization problem:

$$\frac{1}{N} \sum_{i=1}^{N} L\big(f(x_i; W),\, y_i\big) \;+\; \lambda\, R(W)\;,$$

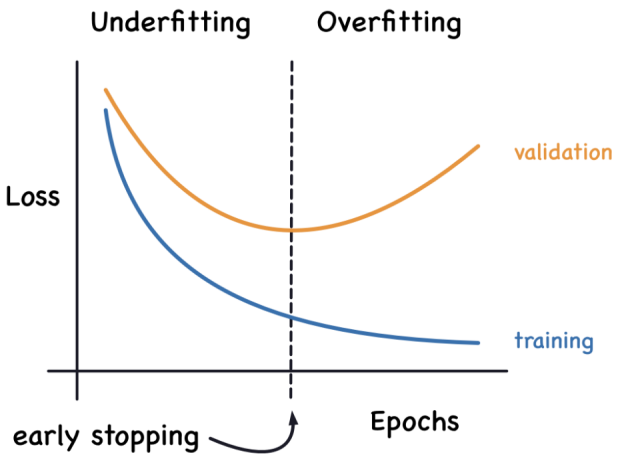

Figure 1: Evolution of either losses or metrics on training data points (in blue) and validation data point (in orange) during a training step.

where $\lambda \in \mathbb{R}$ is a parameter to optimize, and $R(W)$ is the regularization term. Here are the classical regularization terms:

- l_1-regularizer: $R(W) = ||W||_1 - C$
- l_2-regularizer: $R(W) : ||W||_2 - C$

for a fixed positive $C \in \mathbb{R}^*$.

Remark 7. *Including this regularization term ensures that the weights are bounded by C.*

Let's see a 2D visualization of this optimization problem for the loss function $L : \mathbb{R}^2 \to \mathbb{R}$ defined by:

$$\forall\, w_1, w_2 \in \mathbb{R}\,, \quad L(w_1, w_2) = (w_1 - a)^2 + (w_2 - b)^2$$

where $a = 1.5$ and $b = 1$. It is straightforward that the minimum is reached when $(w_1, w_2) = (a, b)$. We can observe that the norm of the optimal point is greater than 1. In the figure 2, we show the minimum when adding a l_1 and l_2 regularization with $C = 1$.

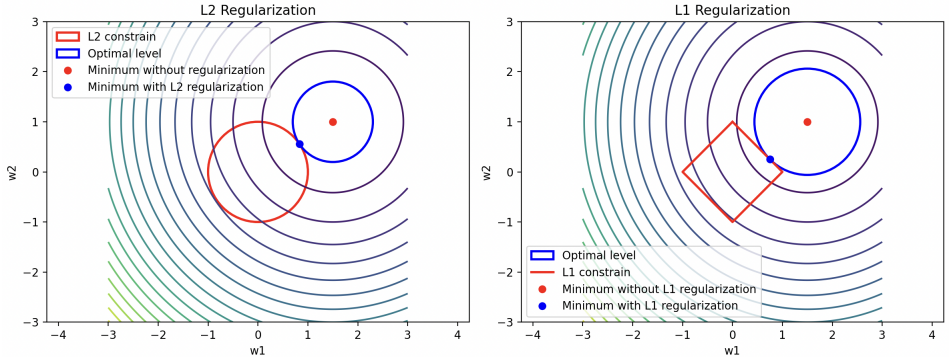

Figure 2: The contours illustrate the values of the loss function across different parameter values. The red point corresponds to the global minimum, whereas the blue point represents the minimum achieved when regularization is applied.

3.2 Dropout

The dropout is quite simple: it randomly deactivates fixed proportion of parameters from selected layers of a neural network during a training step.

Remark 8. *It is important to note that the deactivated neurons during a training step might be activated in the next one.*

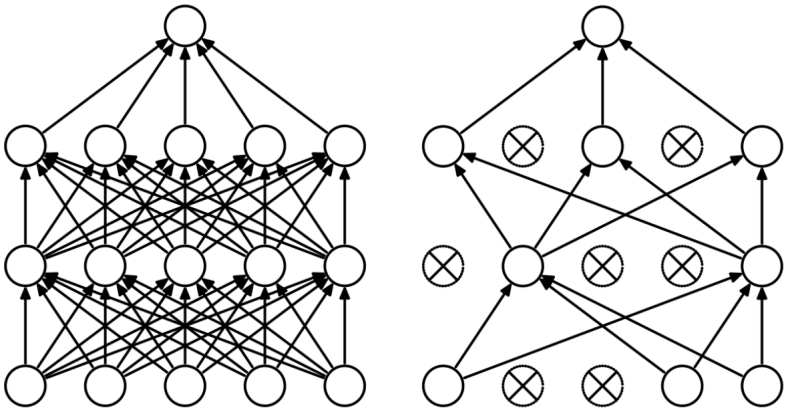

Figure 3: Example of dropout layer applied to all layers, except the last one.

The idea behind the dropout strategy is to prevent from relying too much on some neurons, which may focus on specific patterns from the inputs.

Let's illustrate the effect of a dropout on a simple model which classifies handwritten digits from the dataset MNIST provided by Tensorflow:

```
import os

import cv2

import numpy as np

import matplotlib.pyplot as plt

import tensorflow as tf
import tensorflow_datasets as tfds

# Indicate a path to store your models and data

MAIN_PATH = "./dropout"

# Creation of necessary subfolders

subfolders = ["data", "models", "figures"]
for subfolder in subfolders:
    if not os.path.exists(f"{MAIN_PATH}/{subfolder}"):
        os.makedirs(f"{MAIN_PATH}/{subfolder}")


# Some utils


def normalize_img(image, label):
    """Normalizes images: `uint8` -> `float32`."""
    return tf.squeeze(tf.cast(image, tf.float32), axis=-1) / 255.0, label


def plot_losses(nb_epochs, loss, val_loss, path_to_save):
    epochs = range(1, nb_epochs + 1)

    plt.figure()

    plt.plot(epochs, loss, "bo", label="Training loss")
    plt.plot(epochs, val_loss, "b", label="Validation loss")
    plt.title("Training and validation loss")
    plt.legend()

    plt.savefig(path_to_save)
```

```
def save_test_images(ds_test, path_to_save):
    for images, labels in ds_test:
        _images = images.numpy()
        _labels = labels.numpy()

        np.save(f"{path_to_save}/labels.npy", _labels)

        for i in range(_images.shape[0]):
            _image = _images[i, :, :].reshape(28, 28)
            cv2.imwrite(f"{path_to_save}/image_{i}.png", _image)

        break


# Load the dataset

(ds_train, ds_test), ds_info = tfds.load(
    "mnist",
    split=["train", "test"],
    shuffle_files=True,
    as_supervised=True,
    with_info=True,
)

ds_train = ds_train.map(normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
ds_train = ds_train.batch(32).prefetch(tf.data.AUTOTUNE)

save_test_images(
    ds_test=ds_test.batch(32).prefetch(tf.data.AUTOTUNE),
    path_to_save=f"./{MAIN_PATH}/data",
)

ds_test = ds_test.map(normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
ds_test = ds_test.batch(32).prefetch(tf.data.AUTOTUNE)

# Build the models respectively without and with the dropout

model2_without_dropout = tf.keras.Sequential(
    [
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(100, activation="relu"),
        tf.keras.layers.Dense(100, activation="relu"),
        tf.keras.layers.Dense(10, activation="softmax"),
    ]
)

model2_with_dropout = tf.keras.Sequential(
    [
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dropout(rate=0.2),
        tf.keras.layers.Dense(100, activation="relu"),
        tf.keras.layers.Dropout(rate=0.2),
        tf.keras.layers.Dense(100, activation="relu"),
        tf.keras.layers.Dropout(rate=0.2),
        tf.keras.layers.Dense(10, activation="softmax"),
    ]
)

# Indicate the training parameters

EPOCHS = 25
LR = 0.001


# Compile the models

model2_without_dropout.compile(
    optimizer=tf.keras.optimizers.Adam(LR),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy()],
)
```

```
model2_with_dropout.compile(
    optimizer=tf.keras.optimizers.Adam(LR),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy()],
)

# Train the models

history_without_dropout = model2_without_dropout.fit(
    ds_train, validation_data=ds_test, epochs=EPOCHS
)
history_with_dropout = model2_with_dropout.fit(
    ds_train, validation_data=ds_test, epochs=EPOCHS
)

# Save the models

model2_without_dropout.save(f"./{MAIN_PATH}/models/model_without_dropout.keras")
model2_with_dropout.save(f"./{MAIN_PATH}/models/model_with_dropout.keras")

# Retrieve the losses computed during the training step

losses_training_without_dropout = history_without_dropout.history["loss"]
losses_validation_without_dropout = history_without_dropout.history["val_loss"]
losses_training_with_dropout = history_with_dropout.history["loss"]
losses_validation_with_dropout = history_with_dropout.history["val_loss"]

# Save the evolution of losses

plot_losses(
    nb_epochs=EPOCHS,
    loss=losses_training_without_dropout,
    val_loss=losses_validation_without_dropout,
    path_to_save=f"{MAIN_PATH}/figures/loss_without_dropout.png",
)

plot_losses(
    nb_epochs=EPOCHS,
    loss=losses_training_with_dropout,
    val_loss=losses_validation_with_dropout,
    path_to_save=f"{MAIN_PATH}/figures/loss_with_dropout.png",
)
```

Exercises 4. *After running the above code, answer the following questions:*

- *What is the main difference between the generated plots loss_without_dropout.png and loss_with_dropout.png ?*
- *What do you conclude ?*

3.3 Data transformation

Another common strategy for inducing implicit regularization is to transform the input data. Two common approaches are:

- **Data augmentation:** This technique artificially increases the size of the training set by creating modified versions of existing data points. In Computer Vision, for example, transformations such as rotations, flips, or color shifts help neural networks learn more robust and invariant representations.
- **Normalization layers:** These layers standardize activations by subtracting the mean and dividing by the standard deviation, typically on a per-batch basis. Normalization improves numerical stability and often accelerates training by keeping the distribution of activations more consistent across layers.

We will explore data augmentation in more detail during the exercise sessions, where you will implement and visualize several augmentation strategies and study their impact on model performance.

Normalization layers, on the other hand, primarily address the problem of *internal covariate shift*, which is the

phenomenon where the distribution of inputs to a layer changes during training as the parameters of the previous layers are updated. By keeping activations centered and scaled, normalization ensures that gradients propagate more smoothly through the network.

In the case of Batch Normalization, for a mini-batch of activations $\{x_1, \ldots, x_m\}$, the normalized output is computed as:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}},$$

where μ_B and σ_B^2 are the mean and variance computed over the mini-batch, and ε is a small constant to ensure numerical stability. The normalized values are then scaled and shifted using learnable parameters γ and β:

$$y_i = \gamma \hat{x}_i + \beta.$$

This adaptive re-scaling allows the network to preserve representational capacity while benefiting from the normalization effect.

4 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) were originally designed to automatically extract spatial features from images. An image can be represented as a matrix or tensor, where each element corresponds to a pixel. In grayscale images, each pixel has an intensity value between 0 and 255, with 255 representing white. For color images, each pixel is typically represented as a triplet (r, g, b), where r, g, and b denote the intensity of the red, green, and blue channels, respectively, each ranging from 0 to 255.

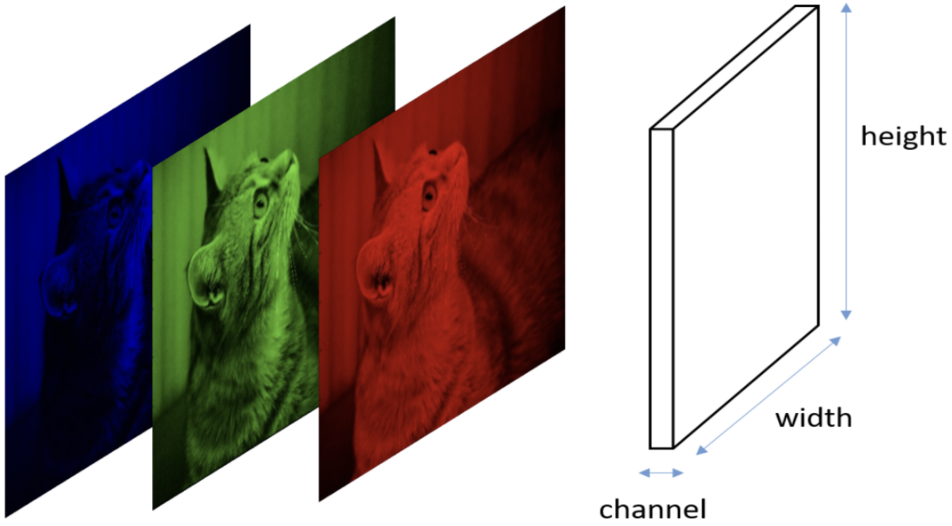

Figure 4: A color image can be represented as the superposition of three channels (red, green and blue) each encoding the intensity of its respective color component.

Training a classical Multi-Layer Perceptron (MLP) directly on raw image data is inefficient due to the extremely high number of parameters required. For instance, an image of size 100×100 contains 10,000 pixels; if the first hidden layer of an MLP has 1,000 neurons, this results in 10 million connections—an impractically large number that can easily lead to overfitting and high computational cost.

CNNs address this issue by introducing local connectivity and weight sharing. Instead of connecting every neuron to every pixel, convolutional layers apply a set of small filters (also called kernels) that slide across the image. Each filter learns to detect specific local patterns such as edges or textures, enabling the network to capture hierarchical spatial features efficiently.

4.1 Convolution product

They are particularly effective for extracting features from data with a grid-like topology, such as images or time series. Before delving deeper into the architecture of these networks, let us first recall the definition of the convolution operation.

Let $f, g : \mathbb{R} \to \mathbb{R}$ be two L^1-integrable functions. The convolution of f and g, denoted by $f * g$, is defined for all $t \in \mathbb{R}$ as:

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(s)\, g(t - s)\, ds$$

Intuitively, the convolution can be interpreted as a **weighted average** of the function f around each point t, where the weights are provided by the function g. In this context, g is referred to as the **kernel** or **filter** of the convolution.

Remark 9. *The convolution product $*$ satisfies the following properties:*

- ***Commutativity.*** $f * g = g * f$
- ***Associativity.*** $f * (g * h) = (f * g) * h$
- ***Bilinearity.*** $\lambda(f * g) = (\lambda f) * g = f * (\lambda g)$ *and* $(f + g) * h = f * h + g * h$

where f, g and h are L_1-integrable maps in $\mathbb{R}$, and λ is a real.

In practice, we usually deal with discrete signals, as is the case for grayscale images, which can be represented as two-dimensional arrays. If we now consider two discrete functions $f, g : \mathbb{Z} \to \mathbb{R}$, their discrete convolution is defined as:

$$(f * g)[n] = \sum_{k=-\infty}^{+\infty} f[k]\, g[n-k]$$

Let I be a grayscale image of dimensions $w \times h$, and let K be a kernel (or filter) of dimensions $k_w \times k_h$. These can be viewed respectively as mappings

$$I : [\![1, w]\!] \times [\![1, h]\!] \to \mathbb{R} \quad \text{and} \quad K : [\![1, k_w]\!] \times [\![1, k_h]\!] \to \mathbb{R}.$$

The two-dimensional discrete convolution between I and K is then defined by:

$$(I * K)[m, n] = \sum_{i=-p}^{p} \sum_{j=-q}^{q} I[m-i, n-j]\, K[i, j],$$

where $p = \frac{k_w - 1}{2}$ and $q = \frac{k_h - 1}{2}$.

Remark 10. *In image processing, the result of such an operation is often called a filtered image, and the matrix K is referred to as a filter or kernel. Filters are widely used for tasks such as denoising, edge detection, and texture extraction.*

Remark 11. *In image processing, the output of such operation is called a filter. We may also identify kernel and filter. Filters are useful for denoising an image, detecting outlines, and so on.*

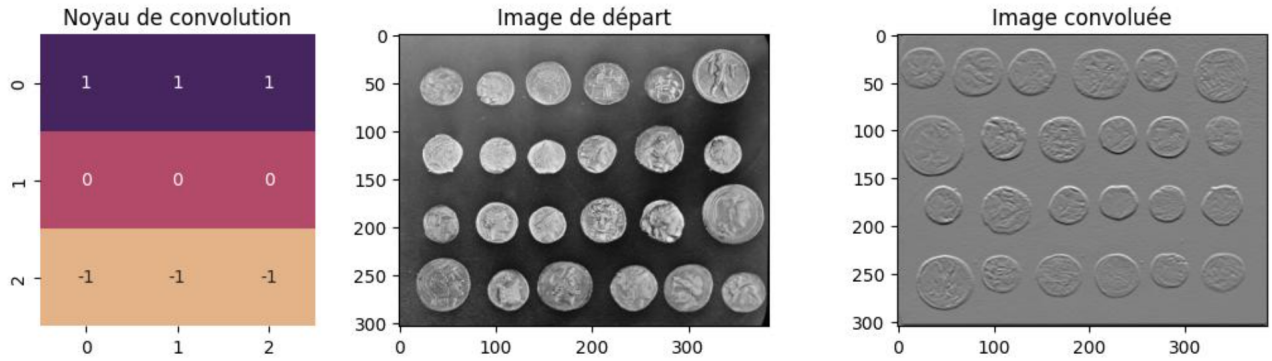

Figure 5: Example of convolution product between an image and a Prewitt filter

Remark 12. *In practice, most deep learning libraries implement the cross-correlation operation instead of the mathematical convolution, as it is simpler to compute and yields equivalent results in this context. The cross-correlation between I and K is the matrix $I \star K$ defined by:*

$$(I \star K)[m, n] = \sum_{i=-p}^{p} \sum_{j=-q}^{q} I[m+i, n+j]\, K[i, j],$$

where $p = \frac{k_w - 1}{2}$ and $q = \frac{k_h - 1}{2}$.

Unlike convolution, cross-correlation does not involve flipping the kernel, and thus the operation is not commutative. However, this distinction has no practical consequence for convolutional neural networks, since the kernel weights are learned during training. As a result, the terms convolution and cross-correlation are often used interchangeably in the deep learning literature.

4.2 Convolution layer

Roughly speaking, a *convolutional layer* performs a convolution operation between an input tensor and a set of learnable filters (also called kernels). A tensor is a generalization of a matrix to higher dimensions and can be viewed as a multi-dimensional array. For example, a color image can be represented as a tensor of shape (H, W, C), where H denotes the height, W the width, and C the number of channels. In most cases, $C = 3$ since each pixel in a color image is described by three primary color components: red, green, and blue. For grayscale images, we have only one channel, i.e., $C = 1$.

There are two common parameters that affect how a convolutional layer operates:

- **Padding:** the process of adding zeros around the border of the input image (or feature map) before applying the convolution. Padding helps preserve spatial dimensions, ensuring that the output feature map has the same height and width as the input.
- **Stride:** the number of pixels by which the filter shifts (or "strides") over the input at each step. A larger stride reduces the output's spatial dimensions and increases the downsampling effect.

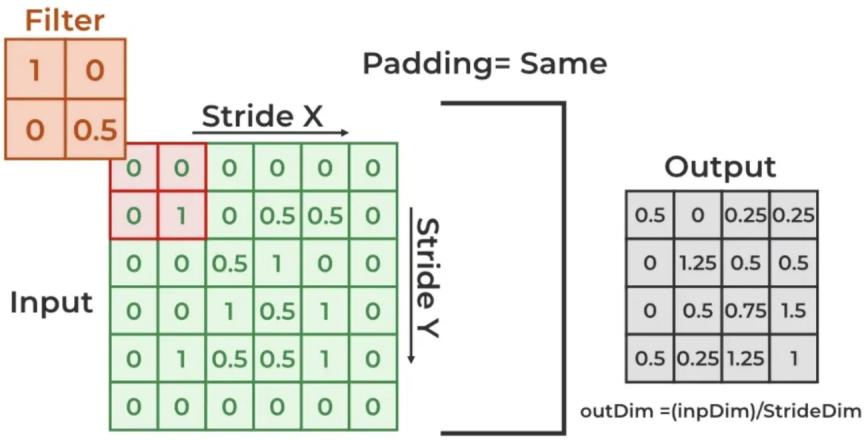

Figure 6: Illustration of Padding and Stride on an example

Remark 13. *In Figure 6, we note that the filter matrix or tensor corresponds to the weights we have to learn using backpropagation for solving an optimization problem.*

Now we are ready to define formally the mathematical operation of a CNN layer:

Definition 1. *Let X be a tensor of shape (H, W, C), and let W be a tensor filter of shape (k_h, k_w, k_c). Then a convolutional layer with horizontal stride s_h and vertical stride s_w maps the tensor X to a tensor Z of shape $(, , k_c)$ given by:*

$$z_{i,j,k} = b_k + \sum_{u=-p}^{p} \sum_{v=-q}^{q} \sum_{w=1}^{C} x_{i\times s_h+u,\, j\times s_w+v,\, w}\, w_{u,v,w,k} \quad ,$$

where $p = \frac{k_h-1}{2}$ and $q = \frac{k_w-1}{2}$.

Remark 14. *Following the previous definition, we note that:*

- *$w_{u,v,w,k}$ corresponds to the weight representing the connection between the feature map k with the component of row u and column v, relative to the feature map w from the filter.*
- *b_k is the biais of the feature map k from the filter, which can be seen as a global brightness adjustment of this feature map.*

4.3 Pooling and Unpooling layers

Pooling layers are operations which aim at under-sampling, i.e reducing dimension of feature maps. This is important for reducing the computation load, for memory usage and reducing the number of training parameters. This is

possible by taking one of the two filters, namely the max or average layer.

Unpooling layers are operations whose objective is to over-sampling, generally for reconstructing the original input signal from a signal in lower dimension.

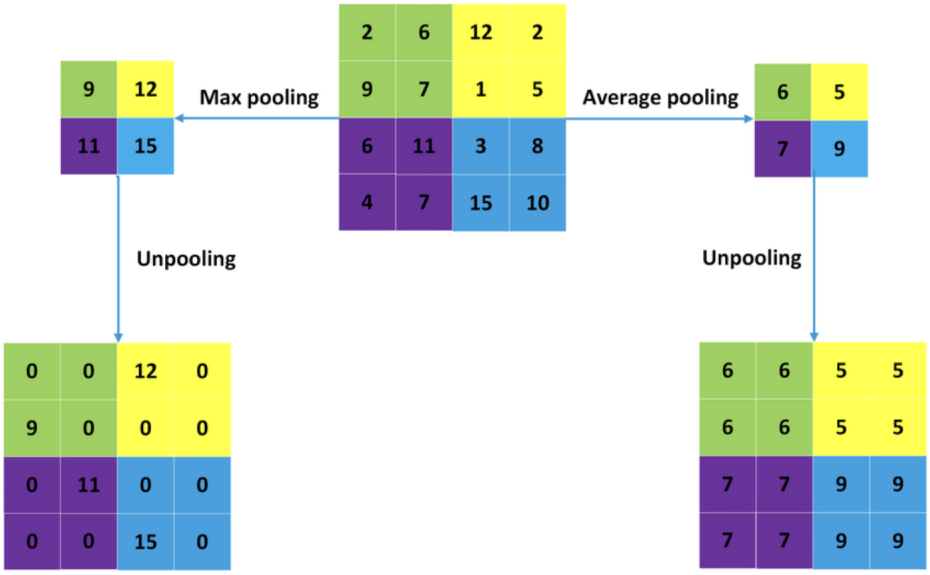

Figure 7: Example of max and average pooling layers with dimension 2×2 and stride 2 on a matrix of dimension 4×4, and of the corresponding max and average unpooling layers applied on the reduced matrices

4.4 Taxonomy of convolutional networks

CNNs are a kind of feedforward neural networks that are able to extract features from data with convolution structures. A CNN is composed by three main components:

- **Feature extraction**, which is generally a sequence of convolutional layers and pooling layers
- **Fully connected layers**, generally a sequence of multi-perceptron layers
- **Output layer**, used for returning a prediction, generally a probability distribution. This layer strongly depends on the objective of the CNN.

Remark 15. *It is worth noting that an activation function is applied pointwise to the feature map returned by a convolutional layer. The most commonly used activation functions in this context are the hyperbolic tangent, the sigmoid function, and the ReLU function.*

There is no a unique way to build CNNs. They are designed following what matters the most. Here are examples of points to take into account:

- The accuracy
- The size of the CNN, which might be important if it is deployed on a mobile phone
- The inference time on a CPU or a GPU

In the following sections, we present a non-exhaustive list of the most important CNNs in chronological order.

4.4.1 LeNet-5

The authors proposed LeNet-5 in 1998, which is an efficient convolutional neural network trained with the backpropagation algorithm for handwritten character recognition.

In layer $F6$ returns a vector $x \in \mathbb{R}^{10}$ where:

$$\forall i \in [\![1, 10]\!], \quad x_i = f(a_i) ,$$

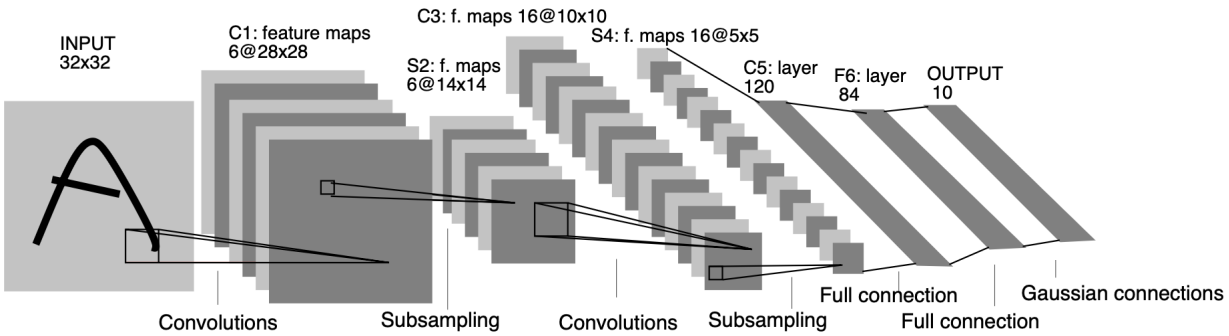

Figure 8: Architecture of LetNet-5

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully connected MLP	-	10	-	-	RBF
F6	Fully connected MLP	-	84	-	-	*tanh*
C5	Convolution	120	1 x 1	5 x 5	1	*tanh*
S4	Average pooling	16	5 x 5	2 x 2	2	*tanh*
C3	Convolution	16	10 x 10	5 x 5	1	*tanh*
S2	Average pooling	6	14 x 14	2 x 2	2	*tanh*
C1	Convolution	6	28 x 28	5 x 5	1	*tanh*
In	Input	1	32 x 32	-	-	-

Table 1: Details on the LetNet-5 architecture.

with a_i is the dot product between the input vector of $F6$ and the weight vector, and f is a scaled and dilated hyperbolic function *tanh* defined by:

$$\forall a \in \mathbb{R}\,, \quad f(a) = A\, tanh(S\, a)$$

where $A = 1.7152$ and S is the slope of *tanh* at the origin in the original paper. The output of the final layer is a vector $y \in \mathbb{R}^{10}$ given by:

$$\forall i \in [\![1, 10]\!]\,, \quad y_i = \sum_{j=1}^{84} (x_j - w_{ij})^2$$

In this original paper, the output layer is composed of Euclidean Radial Basis Function units (RBF) one for each class, with 84 output each. In other words, each output RBF unit computes the Euclidian distance between the input vector and its parameter vector. Hence the original loss function was the Mean Square Error (MSE).

Remark 16. *However, in practice, it is more appropriate to take the Softmax layer in $F6$, and the cross entropy function as the loss function.*

4.4.2 AlexNet

AlexNet is a convolutional neural network (CNN) that won the prestigious ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012. It was developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Unlike LeNet-5, which was designed to classify handwritten digits, AlexNet was built to classify full-color images.

Its architecture shares some similarities with LeNet-5 but introduces several important innovations:

- **ReLU activation:** A ReLU (Rectified Linear Unit) activation function is applied after each convolution operation, replacing the hyperbolic tangent function used in earlier networks.
- **Deeper convolutional layers:** AlexNet includes more consecutive convolutional layers, enabling it to learn more complex features.

- **Regularization techniques:**
  - **Dropout:** A dropout rate of 50% is applied to the outputs of layers F9 and F10 to prevent overfitting.
  - **Data augmentation:** The training data is augmented by randomly shifting images and altering lighting conditions, which improves generalization.
- **Local Response Normalization (LRN):** This technique is applied after the ReLU activations in the outputs of layers C1 and C3, helping to improve generalization and model performance.

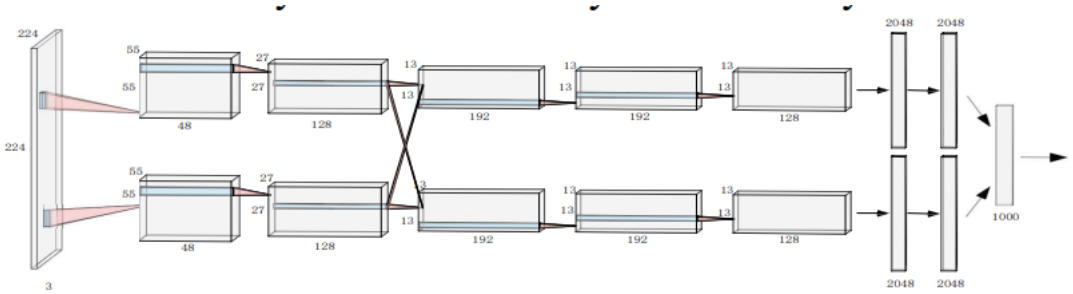

Figure 9: Architecture of AlexNet

Layer	Type	Maps	Size	Kernel size	Stride	Filling	Activation
Out	Fully connected MLP	-	1000	-	-	-	Softmax
F10	Fully connected MLP	-	4096	-	-	-	ReLU
F9	Fully connected MLP	-	4096	-	-	-	ReLU
S8	Max pooling	256	14 x 14	2 x 2	2	valid	-
C7	Convolution	256	1 x 1	5 x 5	1	same	ReLU
C6	Convolution	384	1 x 1	5 x 5	1	same	ReLU
C5	Convolution	384	1 x 1	5 x 5	1	same	ReLU
S4	Max pooling	256	5 x 5	2 x 2	2	valid	-
C3	Convolution	256	10 x 10	5 x 5	1	same	ReLU
S2	Max pooling	96	14 x 14	2 x 2	2	valid	-
C1	Convolution	96	55 x 55	11 x 11	4	same	ReLU
In	Input	3 (RGB)	227 x 227	-	-	-	-

Table 2: Details on the AlexNet architecture.

Definition 2. *Let define the LRN function applied in the outputs of layers C1 and C3:*

$$b^i_{u,v} = a^i_{u,v} \left(k + \alpha \sum_{j=j_{down}}^{j_{up}} (a^j_{u,v})^2\right)^{-\beta},$$

with $j_{down} = min(i + \frac{r}{2}, K - 1)$ *and* $j_{up} = max(0, i - \frac{r}{2})$. *Here we have:*

- $a^i_{u,v}$ *is the activity neuron computed by applying the kernel* i *at position* (u, v) *;*
- $b^i_{u,v}$ *is the normalized output ;*
- K *is the total number of kernels in the layer ;*

Remark 17. *This sort of response normalization implements a form of lateral inhibition inspired by the type found in real neurons, creating competition for big activities among the neuron outputs computed using different kernels.*

In the original paper, the hyperparameters taken for LRN are $r = 5$, $\alpha = 0.00001$, $\beta = 0.75$ *and* $k = 2$.

Training conditions:

- *Loss function:* Cross-entropy function

- *Optimizer:* Stochastic gradient descent with a batch size of 128 examples, momentum of 0.9, and weight decay of 0.0005.

Remark 18. *More precisely the update rule for the weight w is:*

$$v_{i+1} = 0.9\, v_i - 0.0005\, w_i - \epsilon\, Average_{D_i}\Big(\frac{\partial L}{\partial w}_{|w_i}\Big)$$
$$w_{i+1} = w_i + v_{i+1}$$

where i is the iteration index, v is the momentum variable, ϵ is the learning rate, and $Average_{D_i}\Big(\frac{\partial L}{\partial w}_{|w_i}\Big)$ is the average of the i-th batch D_i, of the derivative of the loss function with respect to w, evaluated at w_i.

4.4.3 VGGNets

VGGNets are a series of convolutional neural network algorithms proposed by the Visual Geometry Group (VGG) of Oxford University, including VGG-11, VGG-11-LRN, VGG-13, VGG-16, and VGG-19. VGGNets secured the first place in the localization track of ImageNet Challenge 2014. The authors of VGGNets prove that increasing the depth of neural networks can improve the final performance of the network to some extent.

Layer	Type	Maps	Size	Kernel size	Stride	Filling	Activation
Out	Fully connected MLP	-	1000	-	-	-	Softmax
F15	Fully connected MLP	-	4096	-	-	-	ReLU
F14	Fully connected MLP	-	4096	-	-	-	ReLU
S13	Max pooling	512	-	2 x 2	2	-	-
C12	Convolution	512	-	3 x 3	1	-	ReLU
C11	Convolution	512	-	3 x 3	1	-	ReLU
S10	Max pooling	512	-	2 x 2	2	-	-
C9	Convolution	512	-	3 x 3	1	-	ReLU
C8	Convolution	512	-	3 x 3	1	-	ReLU
S7	Max pooling	256	-	2 x 2	2	-	-
C6	Convolution	256	-	3 x 3	1	-	ReLU
C5	Convolution	256	-	3 x 3	1	-	ReLU
S4	Max pooling	128	-	2 x 2	2	-	-
C3	Convolution	128	-	3 x 3	1	-	ReLU
S2	Max pooling	64	-	2 x 2	2	-	-
C1	Convolution	64	-	3 x 3	1	-	ReLU
In	Input	3 (RGB)	224 x 224	-	-	-	-

Table 3: Details on the VGG-11 architecture.

Table 3 presents the architecture of the smallest VGG network, with the fundamental differences from the AlexNet architecture highlighted in bold. As we can see, the main differences lies on the additional convolution and max pooling layers.

Remark 19. *Using kernel with smaller dimensions (here 3×3) with stride, decreases dramatically the number of parameters.*

4.4.4 ResNet

Before the introduction of ResNet, research primarily focused on increasing the depth of convolutional neural networks (CNNs) to improve performance on benchmark tasks. However, this approach encountered several significant challenges:

- **Vanishing gradients:** Gradients diminish rapidly towards zero, preventing the model from learning effectively during training.
- **Exploding gradients:** Gradients can grow uncontrollably large, often due to error accumulation, leading to unstable training.

- **Degradation problem:** As the network depth increases, accuracy initially saturates and then begins to degrade, even though overfitting is not the cause.

Remark 20. *While the issues of vanishing and exploding gradients have largely been mitigated through techniques such as careful weight initialization and the use of normalization layers (e.g., Batch Normalization), enabling the training of moderately deep networks with stochastic gradient descent (SGD) and backpropagation, the degradation problem remains a major obstacle, especially in very deep architectures.*

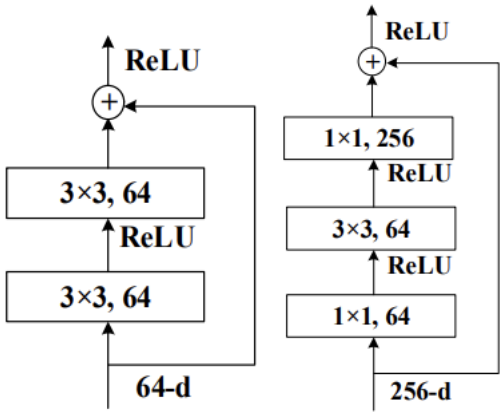

Figure 10: Structure of ResNet blocks. On the left, the structure of two-layer residual block. On the right, the structure of three-layer residual block

ResNet (Residual Network), introduced by He *et al.* in 2015, won the ILSVRC 2015 competition for image classification and detection. Its key innovation is the introduction of **residual connections**, designed to overcome the *degradation problem* observed in very deep networks such as VGG.

Formally, the core building block of a residual network is defined as:

$$y = \mathscr{F}(x, \{W_i\}) + x \ ,$$

where $\mathscr{F}(x, \{W_i\})$ represents the **residual mapping** to be learned, typically a stack of two or three convolutional layers. The shortcut (or skip) connection adds the input x directly to the output of $\mathscr{F}$, enabling efficient gradient propagation and stable training of very deep networks.

Figure 10 illustrates this residual connection structure used in the original ResNet architectures.

5 Transformers based models

Many AI systems today rely heavily on transformer-based models as foundational predictive tools for handling a wide variety of data and serving numerous purposes. This trend gained momentum following the release of ChatGPT, which demonstrated the transformative potential of this technology in how we interact with information and digital systems. Reaching one million users within five days and one hundred million in just two months, this new wave of AI began as human-like chatbots but quickly evolved into a fundamental shift in how we perform everyday tasks such as translation, text generation, summarization, and more.

Transformer-based models were initially applied to tasks involving textual data, particularly within the field of Natural Language Processing (NLP). NLP is an area of study focused on analyzing and interpreting textual information using statistical methods and computer science. Early challenges in this field included the difficulty of encoding textual data effectively and building models capable of capturing long-range dependencies with reasonable computational efficiency. These challenges were addressed through the introduction of the attention mechanism, a core innovation of transformer models, combined with the use of dense vector representations for tokens (units of text drawn from a defined vocabulary).

5.1 Dense vector embeddings for text representations

A technique widely used for representing text is the bag-of-word, introduced in 1950, which became popular in 2000s.

Bag-of-words works as follows: let's assume that we have two sentences for which we want to create numerical representations. The first step of the bag-of-word model is *tokenization*.

Definition 3. *Tokenization is the process of splitting up the sentences into individual wors or subwords, called tokens.*

The *vocabulary* is built by taking the unique tokens collected after tokenization. Using the vocabulary, we simply count how often a word in each sentence appears, which creates a sparse vector representations of text, as illustrated in Figure 11.

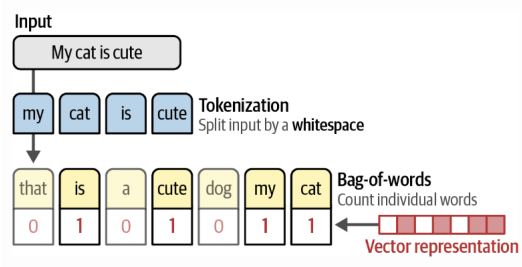

Figure 11: Sparse vector representation of the sentence "My cat is cute" using the vocabulary $\{that, is, a, cute, dog, my, cat\}$

Remark 21. *Bag-of-words is a simple and interpretable method for representing text. However, it has significant limitations when applied to advanced tasks involving deep neural networks:*

- *The vocabulary size can grow rapidly when processing large collections of documents.*
- *Redundant or unnecessary tokens may appear, such as the same word in singular and plural forms. This requires effective pre-processing strategies, such as normalization or weighting more informative tokens.*
- *The resulting text vectors are sparse, which can lead to computational instability.*

Instead, dense vector representations—also known as embeddings—address the limitations mentioned above. These embeddings not only reduce sparsity but also capture semantic meaning, with each component encoding, to some extent, a concept or category. One of the most well-known embedding methods is *word2vec*, which uses a feedforward neural network to learn relationships between words. This semantic information is then encoded into the resulting word embeddings.

Remark 22. *Although dense vector representations are more flexible and capture richer information than the bag-of-words method, they are generally less interpretable. As a result, clustering algorithms are often applied to gain insight into how the neural network has encoded the information.*

5.2 Attention mechanism

The attention mechanism is the core concept behind Transformers, introduced in the paper *"Attention Is All You Need"* published in 2017. Prior to this, the primary models used to capture relationships between tokens were Recurrent Neural Networks (RNNs) and their variants. These models faced several challenges:

- Difficulty in maintaining long-term dependencies between tokens
- High computational cost, as their architecture is inherently sequential

The attention mechanism computes *attention scores*, which serve as weights applied to the input dense embeddings to modify their representations. Compared to recurrent neural networks, attention scores can be computed efficiently and in parallel for all tokens in an input sequence. The calculation of these scores is illustrated in Figure 12.

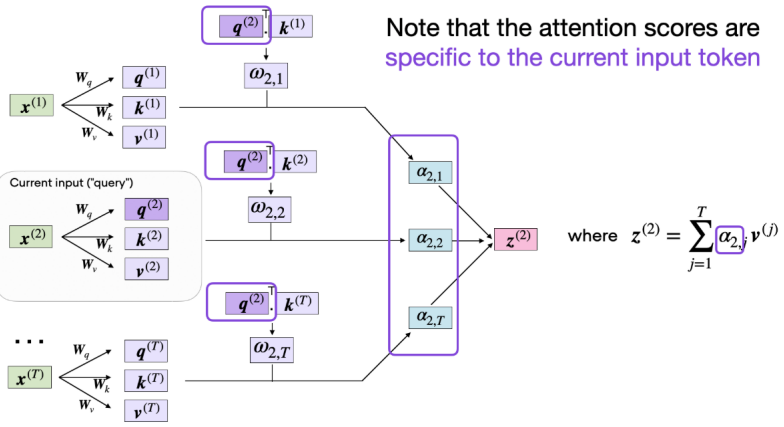

Figure 12: Attention scores calculated for a token represented by the dense vector $x^{(2)}$. The $\alpha_{2,j}$ are the attention scores of $x^{(2)}$, and $z^{(2)}$ is a dense vector which can be seen as a new vector representation of $x^{(2)}$. The integer T is called the *context window*.

Using the notation of Figure 12, we hare:

$$z^{(i)} = \sum_{j=1}^{T} Softmax\Big(\frac{(q^{(i)})^T k^{(j)}}{\sqrt{d_k}}\Big) v^{(j)} = \sum_{j=1}^{T} Softmax\Big(\frac{(x^{(i)})^T W_q^T W_k x^{(j)}}{\sqrt{d_k}}\Big) v^{(j)} \ ,$$

where d_k is the rank of the matrices W_q and W_k. The matrices W_q, W_k and W_v are respectively called the *query matrix*, the *key matrix* and the *value matrix*.

5.3 Positional encoding layers

Positional encoding layers have been a key component since the release of the paper *"Attention Is All You Need"*. They enable the model to keep track of the order of tokens / words in a sequence / sentence, which is an indispensable source of information in language.

The idea here is to encode positions of tokens using their dense vector representations. The easiest way to make possible is to create a "position" vector for each token. In the original article *"Attention Is All You Need"*, they use the so-called *absolute positional encoding* which consists in adding a sinusoidal positional encoding. It works as follows:

- For each position in the sequence, we generate a vector of numbers.
- Each number in this vector is calculated using either a sine or cosine function.

- We use different frequencies for different dimensions of the vector.

More precisely, for a token settled in position i, the j-th coordinate of the position encoding $PE \in \mathbb{R}^d$, where d is the dimension of the dense vector representing, is determined by:

$$PE_j^{(i)} = \begin{cases} sin\left(\frac{i}{10000^{j/d}}\right) & \text{, if } j \text{ is even} \\ cos\left(\frac{i}{10000^{(j-1)/d}}\right) & \text{, if } j \text{ is odd} \end{cases}$$

Here is the Python code for illustrating the values of the components of absolute positional encoding for token at different position:

```
import numpy as np
import matplotlib.pyplot as plt

def sinusoidal_positional_encoding(max_position, d_model):
    position = np.arange(max_position)[:, np.newaxis]
    # The original formula pos / 10000^(2i/d_model) is equivalent to pos * (1 / 10000^(2i/d_model)).
    # I use the below version for numerical stability
    div_term = np.exp(np.arange(0, d_model, 2) * -(np.log(10000.0) / d_model))

    pe = np.zeros((max_position, d_model))
    pe[:, 0::2] = np.sin(position * div_term)
    pe[:, 1::2] = np.cos(position * div_term)

    return pe

max_position = 100  # Maximum sequence length
d_model = 128        # Embedding dimension

pe = sinusoidal_positional_encoding(max_position, d_model)

plt.figure(figsize=(12, 8))
plt.imshow(pe, cmap='coolwarm', aspect='auto', vmin=-1, vmax=1)
plt.colorbar()
plt.title('Sinusoidal Positional Encoding')
plt.xlabel('Dimension')
plt.ylabel('Position')
plt.tight_layout()
plt.show()

plt.figure(figsize=(12, 6))
dimensions = [0, 21]
for d in dimensions:
    plt.plot(pe[:, d], label=f'Dim {d}')
plt.legend()
plt.title('Sinusoidal Positional Encoding for Specific Dimensions')
plt.xlabel('Position')
plt.ylabel('Value')
plt.tight_layout()
plt.show()
```

PRESENT HERE THE ROTARY POSITIONAL ENCODING (RoPE)

5.4 Transformers architecture

The power of the Transformer model, introduced by Google, was brought to prominence thanks to OpenAI's contribution in the paper *"Improving Language Understanding by Generative Pre-Training"*, released in 2018. The authors demonstrated that such a model can be trained on a massive amount of textual data by predicting the next token given a sequence of tokens, and then fine-tuned to solve specific downstream tasks. This strategy led to the development of the first *Generative Pre-trained Transformer (GPT)* model.

The full architecture of the Transformer block is divided into two main components:

- The *encoder part* aims to represent textual information as dense vectors that encode complex semantics.
- The *decoder part* is responsible for generating tokens based on encoded representations.

In Figure 13, we identify four important layers present in both the encoder and decoder parts:

- **Positional encoding layer**: Since Transformers do not have recurrence, this layer injects information about the position of tokens in the sequence.
- **Multi-Head Attention layer**: This mechanism allows the model to focus on different parts of the input simultaneously, capturing various types of relationships between tokens.
- **Feed Forward layer**, i.e., a perceptron model: Applies non-linear transformations independently to each position to increase the model's expressiveness.
- **Residual connections**: These connections help with gradient flow and stabilize training by allowing the model to reuse features from earlier layers.

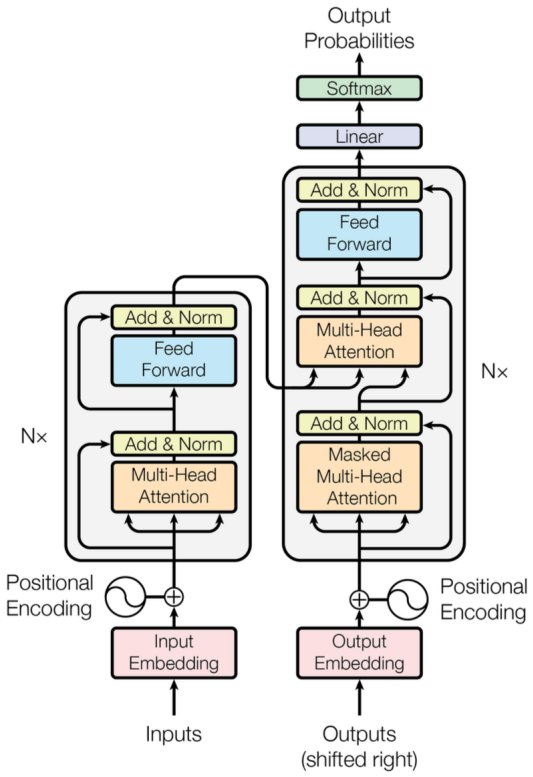

Figure 13: The Transformer - model architecture

It is worth noting that the Transformer model can be configured in three distinct ways:

- **Encoder-only Transformer**: This configuration uses only the encoder block and is typically employed for tasks that require understanding or classification of text, such as text classification, named entity recognition, or paragraph segmentation. A well-known example is *Bidirectional Encoder Representations from Transformers (BERT)*, introduced in 2018, which laid the foundation for many similar models. BERT and its variants (e.g., RoBERTa, ALBERT, DeBERTa, XLM, XLNet, UNILM) are often used in transfer learning, where the model is first pretrained on a language modeling task and then fine-tuned for a specific downstream application.
- **Decoder-only Transformer**, also known as a *causal Transformer*: This configuration uses only the decoder block and is best suited for generative tasks where output is produced in a sequential manner, such as in chatbots. Just as BERT is foundational for encoder-only models, *GPT-1*, introduced in 2018, serves as the foundation for this architecture and other autoregressive language models.
- **Encoder-decoder Transformer**, also called a *sequence-to-sequence Transformer*: This configuration uses both encoder and decoder blocks. It is especially useful for tasks that involve transforming the structure of the input text, such as text summarization or machine translation, where the output sequence differs from the input. Representative encoder-decoder PLMs are T5, mT5, MASS, and BART.

5.5 Large Language Models (LLMs)

The Transformers architecture have become quickly the baseline model for building language models. After releasing GPT-1 and GPT-2, the researchers from OpenAI wondered what happened if they stack multiple decoder-only Transformer blocks. They find that scaling that way often leads to an improved model capacity on downstream tasks.

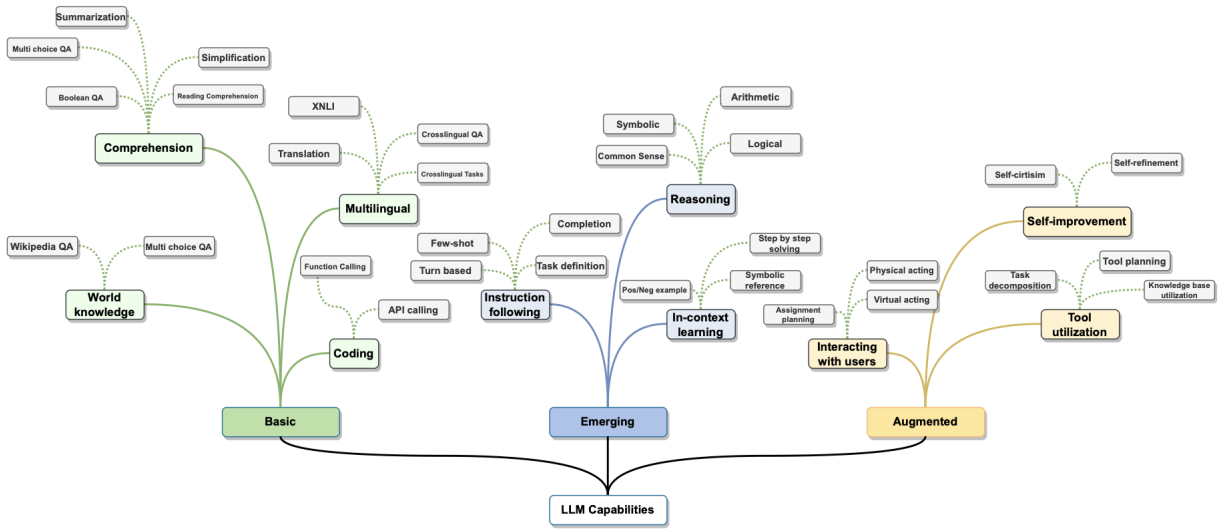

Figure 14: LLM capabilities

As training a model containing a large amount of parameters is quite expense, the OpenAI teams relied on statistics for analyzing the effect of staking Transformers blocks in term of performance. They built the so-called *scaling laws*, by modeling the power-law relationship of model performance based on three factors, namely the model size N, the dataset size D, and the amount of training compute C. Given a compute budget c, they empirically presented three basic formulas for the scaling laws:

$$L(N) = \left(\frac{N_c}{N}\right)^{\alpha_N} \quad , \alpha_N \sim 0.076 \, , N_c \sim 8.8 \times 10^{13}$$
$$L(D) = \left(\frac{D_c}{D}\right)^{\alpha_D} \quad , \alpha_D \sim 0.095 \, , D_c \sim 5.4 \times 10^{13}$$
$$L(C) = \left(\frac{C_c}{C}\right)^{\alpha_C} \quad , \alpha_C \sim 0.050 \, , C_c \sim 3.1 \times 10^{8}$$

where L denotes the cross entropy loss in nats. By scaling Transformer based model, emergent abilities appear, i.e. he abilities that are not initially present in small models but arise in large models. Thanks to theses A non-exhaustive list of LLM capabilities is given in Figure 14. These emergent abilities can be grouped into three main categories:

- **In-context learning.** The in-context learning (ICL) ability is formally introduced by GPT-3. Assuming that the language model has been provided with a natural language instruction and/or several task demonstrations, it can generate the expected output for the test instances by completing the word sequence of input text, without requiring additional training or gradient update.
- **Instruction following.** By fine-tuning with a mixture of multi-task datasets formatted via natural language descriptions (called instruction tuning), LLMs are shown to perform well on unseen tasks that are also described in the form. With instruction tuning, LLMs are enabled to follow the task instructions for new tasks without using explicit examples, thus having an improved generalization ability of instructions.
- **Step-step reasoning.** For small language models, it is usually difficult to solve complex tasks that involve multiple reasoning steps, e.g., mathematical word problems. In contrast, with the chain-of-thought (CoT) prompting, LLMs can solve such tasks by utilizing the prompting mechanism that involves intermediate reasoning steps for deriving the final answer. This ability is speculated to be potentially obtained by training on code strategy.

However, the counterparts of these emergent abilities are that LLMs suffer from majors issues:

- **Hallucinations.** LLMs are prone to generate untruthful information that either conflicts with the existing source or cannot be verified by the available source. This issue can be partially alleviated by special approaches such as alignment tuning, or tool utilization.
- **Knowledge recency.** The parametric knowledge of LLMs is hard to be updated in a timely manner. Augmenting LLMs with external knowledge sources is a practical approach to tackling the issue. However, how to effectively update knowledge within LLMs remains an open research problem.
- **Reasoning inconsistency.** LLMs may generate the correct answer following an invalid reasoning path, or produce a wrong answer after a correct reasoning process, leading to inconsistency between the derived answer and the reasoning process. The issue can be alleviated by fine-tuning LLMs with process-level feedback, using an ensemble of diverse reasoning paths, and refining the reasoning process with self-reflection or external feedback.
- **Toxic answers.** Since LLMs are trained to capture the data characteristics of pre-training corpora (including both high-quality and low-quality data), they are likely to generate toxic, biased, or even harmful content for humans.

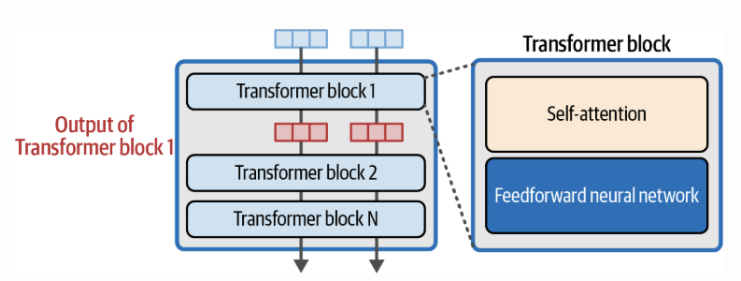

Figure 15: The Transformer block

These problems led the researchers to find some strategies to ensure that the LLMs are aligned with human preference, and provide correct answers. Briefly, LLMs should be *helpful*, *honest* and *harmless*.

5.6 Multimodal Large Language Models

At first, LLMs were only language models. However they quickly were quickly useful for handling different other types (modalities) of data, such as images, audio, video or sensors, as illustrated in Figure 16.

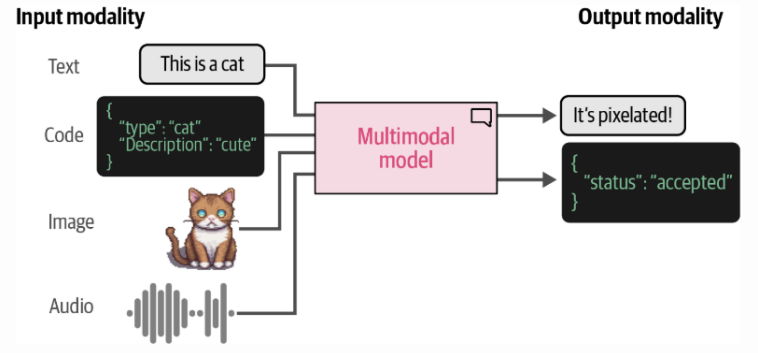

Figure 16: Illustration of a multimodal model

5.7 Transformers for Vision

Let focus how Transformers are used for handling images. Such models are called Vision Transformers (ViT), which have been shown to do tremendously well on image recognition tasks compared to the previously default CNNs. Like the original Transformer, ViT is used to transform unstructured data, an image, into numerical representations through its encoder part.

Just like text is converted into tokens of text, an image is converted into patches of images. The flattened input of image patches can be thought of as the tokens in a piece of text.

Remark 23. *However, unlike tokens, we cannot just assign each patch an identifier since these patches will rarely be found in other images, unlike vocabulary of a text.*

Following the above remark, the patches are linearly embedded to create dense vectors, namely embeddings. Then these can be used as the input of a Transformer model. That way, the patches of images are treated the same way as tokens, as it is shown in Figure 17.

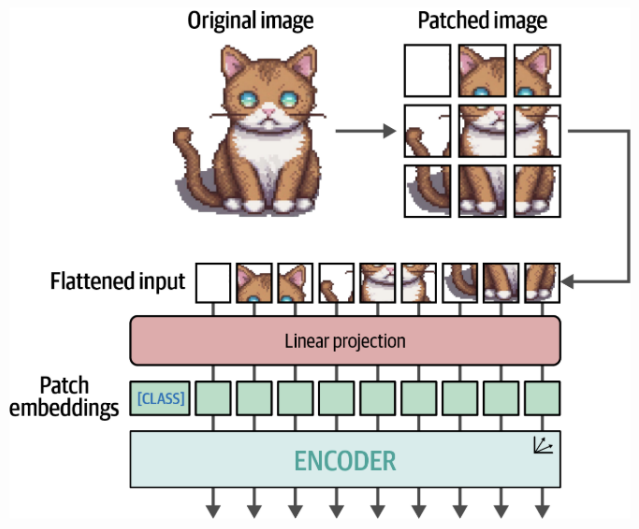

Figure 17: Illustration of a multimodal model

Now we focus on two baseline multimodal models, namely *CLIP* for representing both texts and images with embeddings and *BLIP-2* for generating images from texts.

5.7.1 CLIP

CLIP is an embedding model that can compute embeddings of both images and texts. The resulting embeddings lie in the same vector space, which means that the embeddings of images can be compared with the embeddings of text. This comparison capability makes CLIP, and similar models, usable for tasks such as:

- *Zero-shot classification.* We can compare the embedding of an image with that of the description of its possible classes to find which class is the most similar.
- *Clustering.* Cluster both images and a collection of keywords to find which keywords belong to which sets of images.
- *Search.* Across billions of texts or images, we can quickly find what relates to an input text or image.
- *Generation.* Use multimodal embeddings to drive the generation of images.

CLIP has been trained on millions of images alongside captions using the *contrastive learning*. This training strategy aims at maximizing the cosine similarity between the embeddings corresponding to image / caption pairs, and at minimizing them for dissimilar image / caption pairs, as illustrated in Figure 18.

Here is a summary of the learning process:

1. **Embed input**. Images and texts are converted into embeddings, namely dense vectors.
2. **Compare embeddings**. The similarity between the sentence and the image embedding is calculated using cosine similarity.
3. **Update model**. This updates the embeddings such that they are closer in vector space if the inputs are similar.

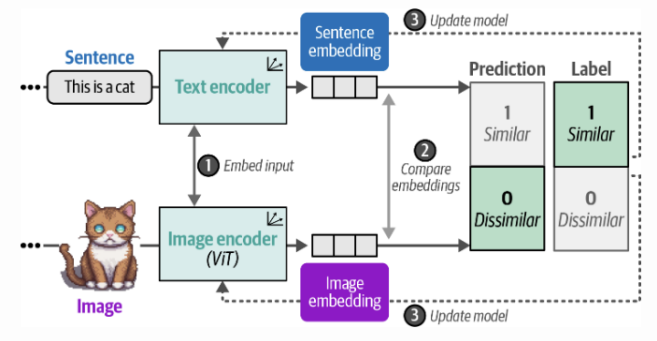

Figure 18: Contrastive learning of CLIP model

The training step consists in minimizing the *symetric cross entropy loss*, also called the *multi-class N pair loss*, which is here defined as follows:

$$-\frac{1}{N}\sum_i ln\Big(\frac{e^{v_i^T w_i/T}}{\sum_j e^{v_i^T w_j/T}}\Big) - \frac{1}{N}\sum_j ln\Big(\frac{e^{v_j^T w_j/T}}{\sum_i e^{v_i^T w_j/T}}\Big) \quad ,$$

where the v_i's and w_i's are respectively the embeddings of texts and images through the text and image encoder.

5.7.2 BLIP-2

BLIP-2 is a generative model that supports images as additional context for generating an answer. Instead of building an architecture from scratch, BLIP-2 bridges the vision-language gap by building a bridge, named the *Querying Transformer* also called *Q-Former*, that connects a pretrained image encoder and a pretrained LLM, as illustrated in Figure 19.

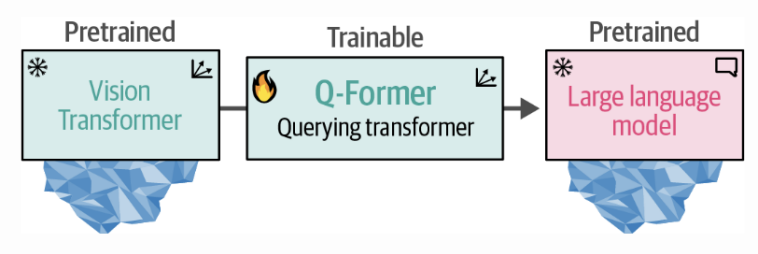

Figure 19: The Q-Former is the bridge between ViT and LLM that is the only trainable component of the pipeline

To connect the two pre-trained models, the Q-Former mimics their architectures. It has two modules that share their attention layers:

- An Image Transformer to interact with the frozen Vision Transformer for feature extraction
- A Text Transformer that can interact with the LLM

BLIP-2 is trained in two stages:

1. **Step 1:** Image-document pairs are used to train the Q-Former to represent both images and text, as illustrated in Figure 20. These pairs are generally captions of images, as we seen before when training CLIP in the previous section.
2. **Step 2:** The learnable embeddings derived from Step 1 now contain visual information in the same dimensional space as the corresponding textual information within the Q-Former. The learnable embeddings are then passed to the LLM. A fully connected linear layer in put in between them to make sure that the learnable embeddings have the same shape as the LLM expects, as illustrated in Figure 21.

It is now worth to deep dive into the training strategy of Step 1, which aims at optimizing three objectives:

- *Image-text contrastive learning.* This task attempts to align pairs of image and text embeddings such that they maximize their mutual information.

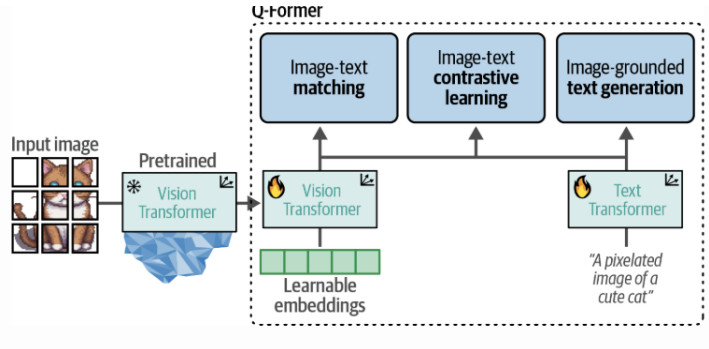

Figure 20: In step 1, the output of the frozen ViT is used together with its caption and trained on three contrastive-like tasks to learn visual-text representations.

- *Image-text matching.* A classification task to predict wether an image and text pair is positive (matched) or negative (unmatched).
- *Image-grounded text generation.* It Trains the model to generate text based on information extracted from the input image.

Remark 24. *These three objectives are jointly optimized to improve the visual representations that are extracted from the frozen ViT. In a way, the first stage of training tries to inject textual information into the embeddings of the frozen ViT so that we can use them in the LLM.*

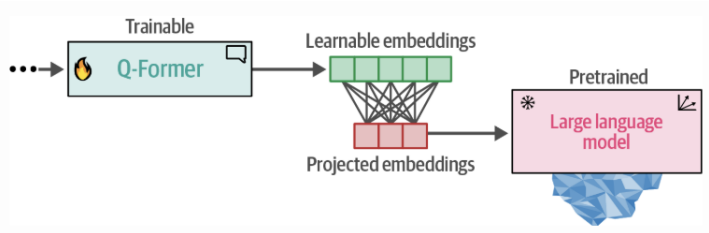

Figure 21: In step 2, the learned embeddings from the Q-Former are passed to the LLM through a projection layer.

Since BLP-2, many other visual LLMs have been released that have similar processes, like LLaVA, a framework for making textual LLMs multi-modal or Idefics 2, an efficient visual LLM based on Mistral 7B LLM.