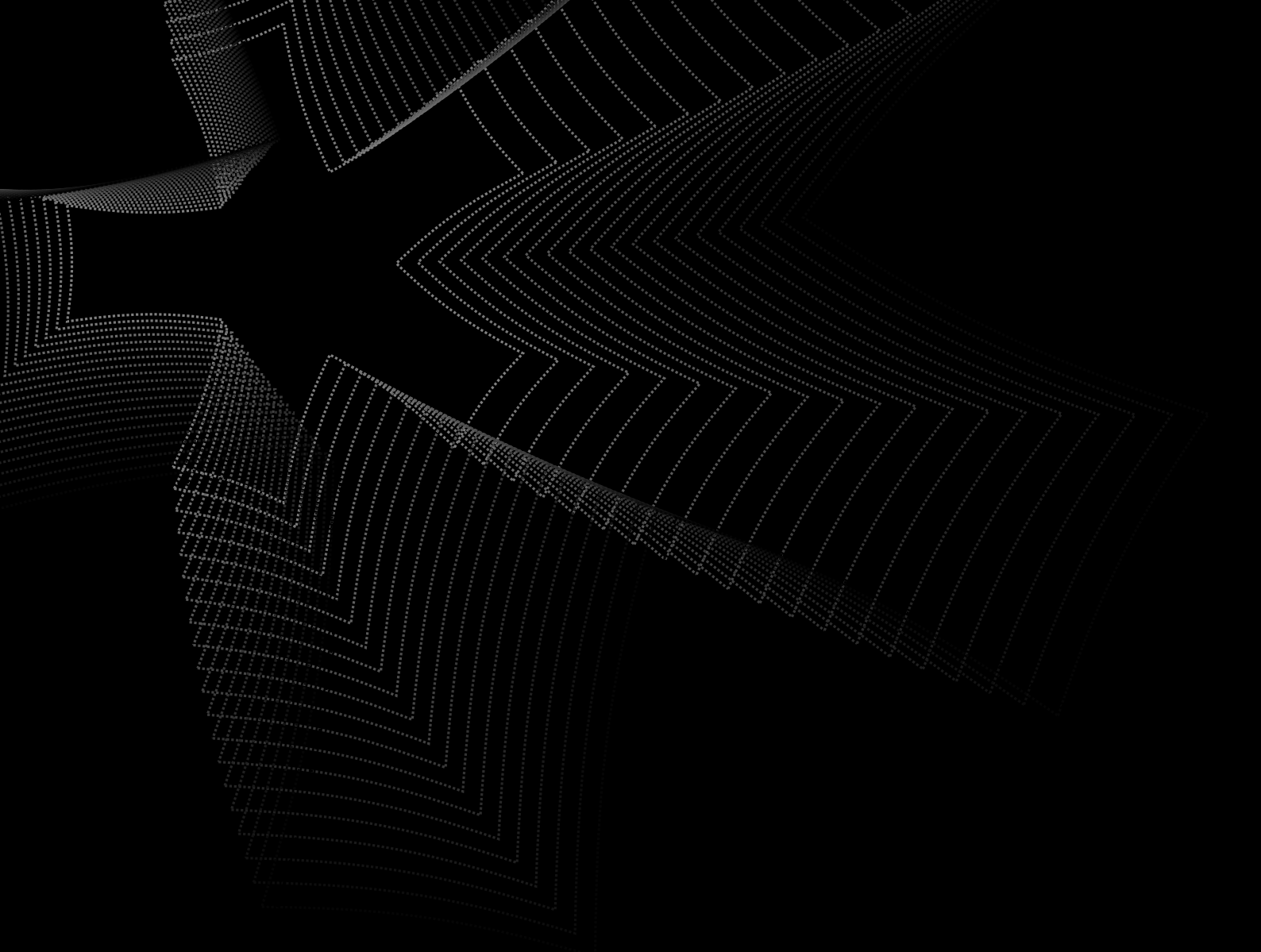


Deep Learning

University of Trento

Federico Brancasi
Jacopo Manenti

INTRODUCTION

Greetings, inquiring minds!

Welcome to this journey through the fascinating world of deep learning. We are Federico and Jacopo, two students like you who decided to put on paper (or rather, pixels on screen) our knowledge acquired during the Deep Learning course taught by Professor Elisa Ricci at the University of Trento.

First of all, a clarification: we are not professionals in the field, but simply two enthusiasts who decided to share their learning journey. These handouts are born out of a desire to help other students like us to tackle the study of Deep Learning.

The content of this book is based primarily on Professor Ricci's course, but we have also drawn from various other sources. For example, we have been inspired by 3Blue1Brown videos (which saved our lives in understanding backpropagation), scientific papers, and specialized websites. Whenever we have been inspired by an outside source, you will find the appropriate link in the text.

A special feature of this book are the illustrations: they are all done using TikZ library. We know that graphics can make all the difference in understanding complex concepts, so we paid special attention to this aspect.

Now, let's talk about you! This project was created for students and is open to students. If you would like to contribute, improve the LaTeX code or report any errors, you can do so on our **GitHub repository** (click on it to open the link). Your pull requests are more than welcome, whether they are corrections to the text or improvements to the illustrations.

We know that Deep Learning can seem like a tough subject at first. That is precisely why we have tried to go into detail on each topic, breaking down complex concepts into more digestible parts. Our goal is that, at the end of this book, you will be able to say, "*Hey, that wasn't so hard!*"

It's worth mentioning that this book was created using a modified version of the LatHex Forta template. We adapted it to fit our specific needs. If you're interested in the original template, you can find it here: **LatHex**.

That said, all that remains is for us to wish you good study! Remember: learning is a process, not a destination.

Have a safe journey!

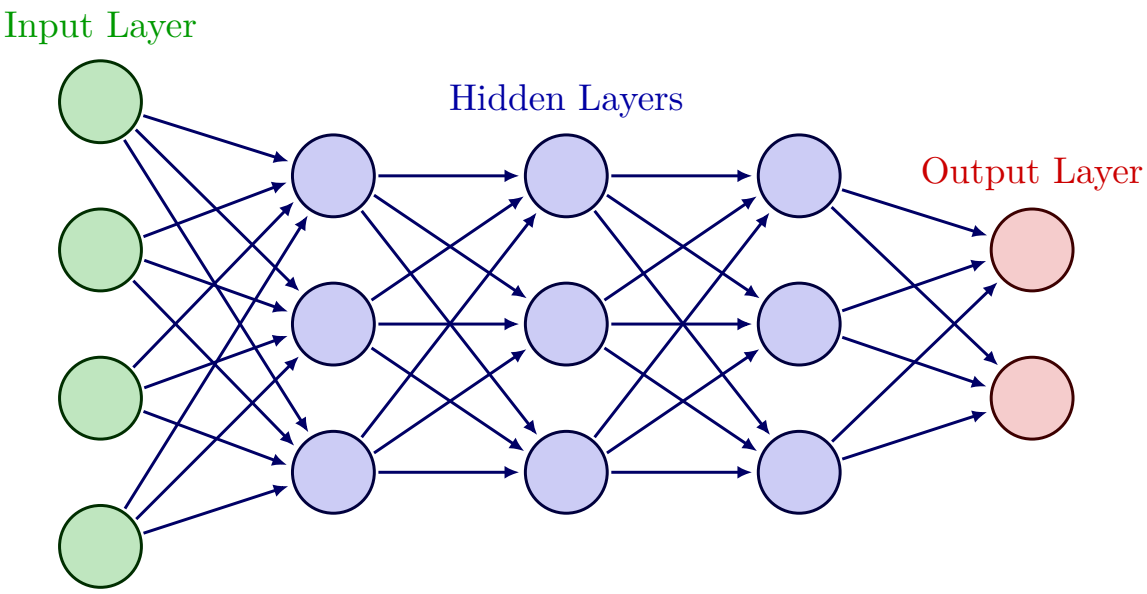
Federico and Jacopo

1. FEEDFORWARD NEURAL NETWORKS

1.1. FEEDFORWARD NEURAL NETWORKS

1.1.1. Understanding Feedforward Neural Networks

Feedforward neural networks represent one of the fundamental pillars of the field of deep learning. To fully understand how they work, it is essential to have a clear understanding of their components and their role in the broader context of data modeling.



Example of a Feedforward Neural Network

A feedforward neural network can be thought of as a sequence of features, each of which represents a layer of the network. This structure is based on the composition of these functions to form an overall representation of the input data. Formally, we can represent a feedforward network as a composition of several functions:

$$f(\mathbf{x}) = f^{(i)}(f^{(i-1)}(\dots f^{(2)}(f^{(1)}(\mathbf{x}))\dots))$$

Where $f^{(1)}$ is the first layer (or input layer), $f^{(i)}$ represents the output of the i-th layer, and so on. The depth of the network is defined by the number of layers, that is, the number of functions $f^{(i)}$ involved.

In evaluating the functions of a feedforward network, which is directed acyclic graph (DAG), the flow of information proceeds in a single direction, from input to output. This flow passes through intermediate computations in the various layers of the network.

In the initial layer, also called **Input Layer**, each node corresponds to an input feature or variable, indeed there are no calculations or transformations applied to the data here.

Instead, neurons in the **Hidden Layers** perform nonlinear computations on the input data through the application of weights and activation functions. The number of hidden layers and the number of neurons in each layer may vary depending on the architecture of

the network and the complexity of the problem. The main function of the hidden layers is to capture and represent complex and abstract features of the input data.

Finally in the **Output Layer** we obtain the desired output. The number of nodes in this layer depends on the nature of the problem, for example, in a binary classification problem there will be only one node, while in a multiclass classification problem there will be one node for each class. Also the type of activation function used in the output layer depends on the type of problem the network is addressing! For example, for binary classification problems, the sigmoid activation function can be used, while for multiclass we use the softmax.

1.1.2. Overcoming the Limitations of Linear Models

Linear models have long been used for their simplicity and convex optimization, but they have significant limitations in their ability to capture complex interactions between input variables. This limitation is particularly evident when dealing with complex problems such as nonlinear classification.

In the context of linear models, such as support vector machines, the choice of function ϕ (**Nonlinear mapping of input data in feature space**) is crucial to satisfactory performance. However, there are limitations in engineering this function, which can make it difficult to capture complex information in the data. Below, we review three common options for addressing these challenges:

1. **Use of generic ϕ**: A first option is to use a generic ϕ function. For example, in kernel machines with RBF (Radial Basis Function) kernels, ϕ has implicit infinite dimensionality. This provides sufficient capacity to fit the training data. However, this approach may suffer from poor generalization for highly variable objective functions.
2. **Engineer ϕ**: Here, efforts are devoted to manually engineering a function that can capture the relevant information in the input data. However, this approach may be limited by the complexity of the data and the difficulty in correctly identifying key relationships.
3. **Learning ϕ from the data**: Finally, an option is to learn the ϕ function directly from the data. This approach renounces convexity but offers significant advantages by combining the strengths of the first two options. ϕ can be modeled very generically, and engineering can focus on designing neural network architectures capable of learning complex and informative representations from input data.

Thanks to Feedforward Neural Networks we can overcome the limitations of traditional linear models because they are able to learn complex, nonlinear functions from data, thus overcoming the inherent limitations of linear models in capturing complex relationships between input variables.

The training of a neural network shares many similarities with that of other machine learning models. However, it has unique features that deserve to be explored. While it is essential to provide only the output of the final layers during training, the hidden layers do not require specific analysis during this phase, which contributes to the complexity and overall effectiveness of the network.

The main difference from linear models lies in the loss functions. While in linear models the loss is convex and guarantee convergence, in neural networks it become nonconvex, leading to greater complexity and not guaranteeing convergence to a global optimum.

Train a neural network means optimize the parameters' values $\mathbf{W}$ (wheights) in order to achieve $f(X, \mathbf{W})$ as close as the target unknown function $f(X)$. So, it is essential to apply gradient descent, an iterative approach to minimizing the cost function, and it is crucial to specify an appropriate **cost function** that accurately represents the error between the predicted output and the desired output. In addition, modeling choices may vary, including the selection of **output representation**, **activation functions**, and **network architecture** (e.g., number of layers), significantly affecting network performance.

Beyond that, there are other aspects of modeling that can be adapted to improve network performance, such as the choice of **optimizer** and the use of **regularization** techniques to avoid overfitting and improve generalization.

To summarize, in a Neural Network we need to specify the following elements: cost function, output representation, activation function, architecture, optimizer, regularization.

1.2. COST FUNCTIONS

Cost (or Loss) functions play a key role in the training of deep learning models. They measure the **discrepancy between model predictions and actual target values** during the training process. An accurate choice of loss function is essential for optimal model performance.

For **Classification** problems, two commonly used loss functions are Categorical Cross-Entropy and Binary Cross-Entropy. These functions are designed to work with class probabilities and are particularly well suited for models that produce probability distributions.

$$\textbf{Categorical Cross-Entropy} = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

y_i: true probability of class i

$\hat{y}_i$: predicted probability of class i

$$\textbf{Binary Cross-Entropy} = -\left(y \log(\hat{y}) + (1-y)\log(1-\hat{y})\right)$$

y: true label (0 or 1)

$\hat{y}$: predicted probability of class 1

In the case of **Regression**, different loss functions are used, including:

$$\textbf{Mean Squared Error} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

y_i: actual (or true) target value for the example i

$\hat{y}_i$: model prediction for the example i

Mean Squared Error (MSE) calculates the mean squares of the differences between model predictions and actual values. It is commonly used and strongly penalizes significant errors. However, it is sensitive to outliers.

$$\textbf{Mean Absolute Error} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

$\hat{y}_i$: model prediction for the example i

y_i: actual (or true) target value for the example i

Mean Absolute Error (MAE), or only Absolute Error (AE), calculates the mean of the absolute differences between predictions and actual values. It is less sensitive to outliers than MSE, but its derivative is undefined in zero.

$$\textbf{Huber Loss} = \frac{1}{N}\sum_{i=1}^{N}\begin{cases}\frac{1}{2}(y_i - \hat{y}_i)^2 & \text{if } |y_i - \hat{y}_i| \leq \delta \\ \delta(|y_i - \hat{y}_i| - \frac{1}{2}\delta) & \text{otherwise}\end{cases}$$

δ: hyperparameter controlling the transition between the quadratic and linear regions of the loss function

Huber Loss combines the best features of MSE and MAE. For small errors, it behaves similar to MSE, while for larger errors, it behaves like AE. This avoids problems such as overfitting and steep slope of the MSE curve.

Please Note: There are numerous other loss functions besides those mentioned above, which can be used depending on the specific problem to be addressed and the characteristics of the data. The choice of loss function depends on the goal of model training, the nature of the data, and the needs of the application. For example, in special contexts such as outlier detection or handling noisy data, specialized loss functions might be preferred. Moreover, as deep learning research advances, new loss functions are constantly being developed and existing ones adapted to meet emerging challenges in different application scenarios.

Also: The choice of the cost function is not independent of the choice of the output unit, because choosing one specific output unit over another can be convenient for the cost function we want to use (e.g., sigmoid and binary cross-entropy).

1.3. UNIT TYPES

1.3.1. Linear Units

Linear Units play a key role in the intermediate and output layers of neural networks. These units apply a **linear transformation to the inputs** by combining the weights associated with the inputs with the input values themselves. The output produced by Linear Units is a linear combination of the inputs.

The formula for calculating the output of Linear Units is:

$$\hat{y} = W^T h + b$$

Where $\hat{y}$ represents the predicted output, W are the weights associated with the inputs, h represents the features from the previous layer, and b is the bias term.

1.3.2. Softmax Units

Softmax Units are commonly used in the output layers of neural networks, especially in cases of Multiclass Classification (with Cross-Entropy Loss). These units transform the unnormalized scores from the last linear layer into a **normalized probability distribution**, allowing the model to assign a probability to each membership class for a given input.

The softmax formula used to calculate the probabilities is as follows:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Where z_i represents the non-normalized score associated with the class i, and $\sum_j e^{z_j}$ is the sum of all the exponents of the non-normalized scores.

Log Softmax is a variant of the Softmax function that is used to **improve numerical stability** when computing probabilities. The formula for calculating the Log Softmax is as follows:

$$\log \text{Softmax}(z_i) = z_i - \log\left(\sum_j e^{z_j}\right)$$

Here are some advantages:

- **The term z_i never saturates**: This means that the scale of values z_i is not compressed or restricted in any way. This is beneficial because it allows the model to continue to learn from more relevant information without limitations imposed by value saturation.
- **Maximizing the log-likelihood encourages z_i to be increased, while encouraging all other z to be decreased**: When we maximize the log-likelihood, we are trying to make the model's prediction as close to reality as possible. Thus, we encourage z_i to increase, since it represents the probability associated with the correct class. At the same time, we encourage the other z to decrease to reduce the probability associated with the wrong classes.
- **The log-likelihood cost function heavily penalizes the most active incorrect prediction**: The log-likelihood cost function significantly penalizes the most active incorrect prediction. This means that if the model is very sure of an incorrect prediction (for example, it has a very high probability for an incorrect class), the cost function will increase significantly, encouraging the model to correct that error.

This analysis prompts us to consider a fundamental question: *How can we select the activation function h most suitable for our neural network?*

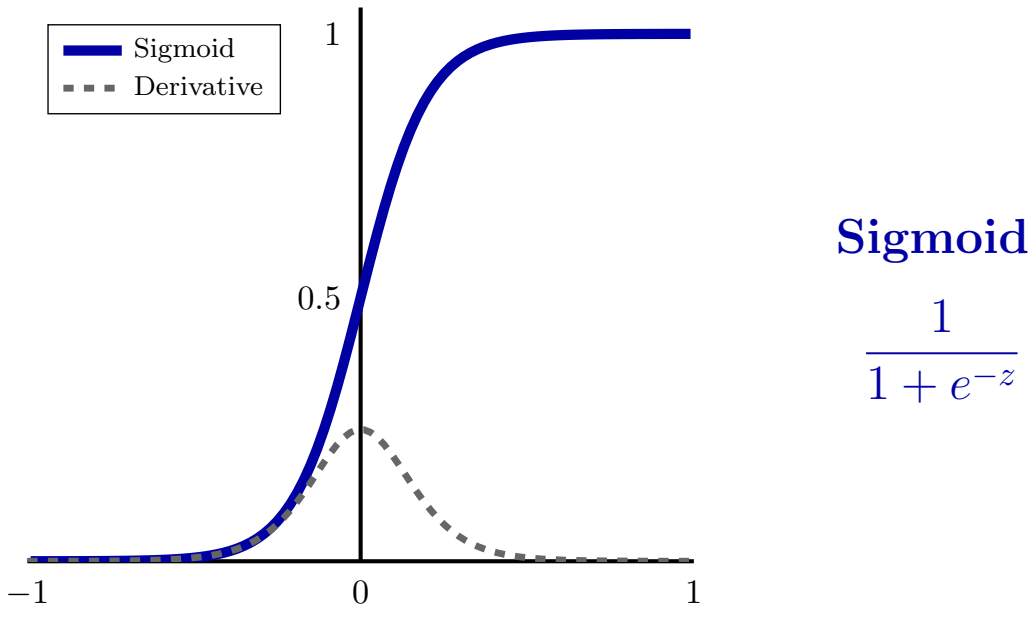
In the next section, we try to answer this crucial question! We have decided to be quite descriptive, as we believe that more details about these functions can facilitate a deeper understanding.

1.4. ACTIVATION FUNCTIONS

1.4.1. Sigmoid

Pros	Cons
The sigmoid function offers a valuable advantage because it acts as a type of **squashing non-linearity** that "squeezes" outputs within the range of [0, 1]. This feature is particularly advantageous in scenarios such as binary classification tasks, where the model must generate outputs that can be interpreted as probabilities. Moreover, the sigmoid function has a property of **differentiability and smoothness** over its entire domain. This property facilitates seamless integration with gradient-based optimization techniques, ensuring stable convergence during the training process.	However, it **tends to saturate** in most of its domain. This saturation phenomenon causes the gradient to tend toward zero, posing challenges during the learning phase. Consequently, this saturation can obstruct the effective propagation of gradients through the network during backpropagation. In addition, the sigmoid function shows strong sensitivity mainly when the input is near zero. In contrast, its **sensitivity decreases** for both large and small input values. This characteristic negatively affects the model's ability to learn from datasets that have a wide range of input values.

Benefits and Limitations of Sigmoid Activation

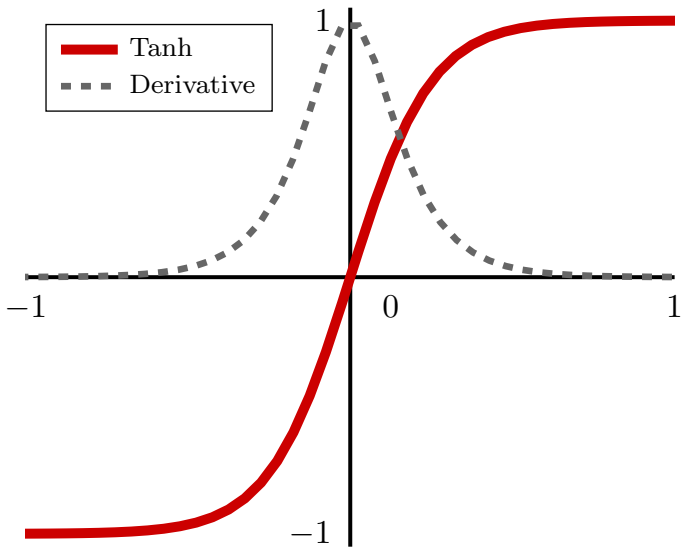


Sigmoid Activation Function Plot

1.4.2. Hyperbolic Tangent

Pros	Cons
First, it is **differentiable and continuous**, ensuring compatibility with gradient-based optimization techniques. In addition, the hyperbolic tangent function is **similar to sigmoid**, but provides a more favorable output range. By squashing the outputs in the interval $[-1, 1]$, it ensures that the outputs are centered in zero. This feature can improve the model's ability to learn and adapt to data distributions, especially in scenarios where centered data is advantageous.	However, the hyperbolic tangent function also has limitations. Like the sigmoid, it tends to **saturate** throughout much of its domain. This saturation phenomenon can hinder effective gradient propagation during back-propagation, impeding the learning process and potentially leading to slower convergence.

Benefits and Limitations of Hyperbolic Tangent Activation



Hyperbolic Tangent

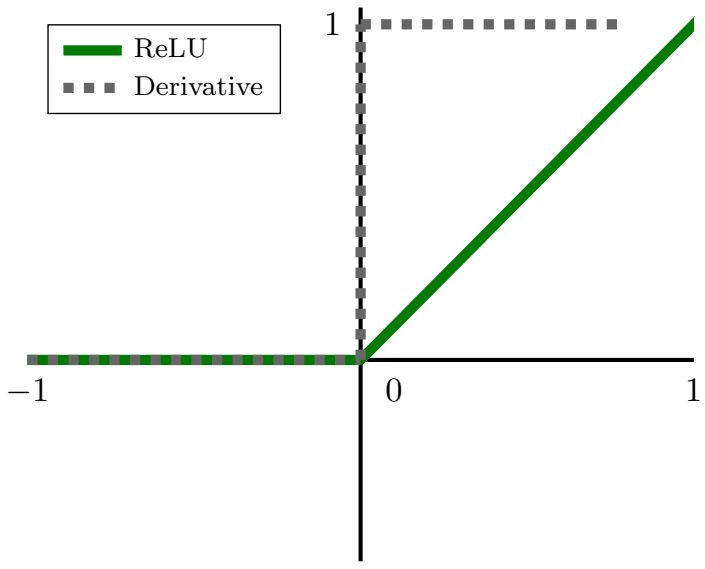
$$\tanh z = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Hyperbolic Tangent Activation Function Plot

1.4.3. Rectified Linear Unit (ReLU)

Pros	Cons
The ReLU function has several advantages. First, it provides **wide and consistent gradients** when it is active. This means that the ReLU function does not saturate when it is active, which greatly accelerates the convergence of gradient-based optimization. In addition, the **non-positive limitation** accelerates the convergence of gradient descent, allowing for faster learning. The ReLU function may have a limitation in its **differentiability**. However, in practice, this is not necessarily a problem since the one-sided derivative is returned when $z = 0$. Moreover, gradient-based optimization is subject to numerical errors anyway, so this differentiability limitation is often overlooked. A good practice is to **initialize bias** with small positive values. This ensures that units are initially active for most inputs and that derivatives can propagate through.	However, the ReLU function also has some disadvantages. First of all, its outputs are not **centered on zero**, which could cause problems in some machine learning contexts. In addition, ReLU neurons may suffer from a phenomenon called **"dying ReLU"**. This occurs when most of the inputs to ReLU neurons are in the negative range. When this occurs, the ReLU neurons return an output of 0 and the gradients do not flow during back-propagation, preventing the weights from updating. However, this problem does not always occur and can be mitigated by properly adjusting the learning rate or using Stochastic Gradient Descent (SDG). The ReLU function can also be sensitive to the **large learning rate**. If the learning rate is too high, the new weights are likely to end up in the highly negative range, negatively affecting the convergence of the model.

Benefits and Limitations of ReLU Activation



ReLU

$\max(0, z)$

ReLU Activation Function Plot

1.4.4. Generalized Rectified Linear Units

The **generalized rectified linear units** were introduced to address some limitations of the ReLU function. The main objective is to obtain a non-zero slope when $z_i < 0$. The generalized form of the ReLU is defined as follows:

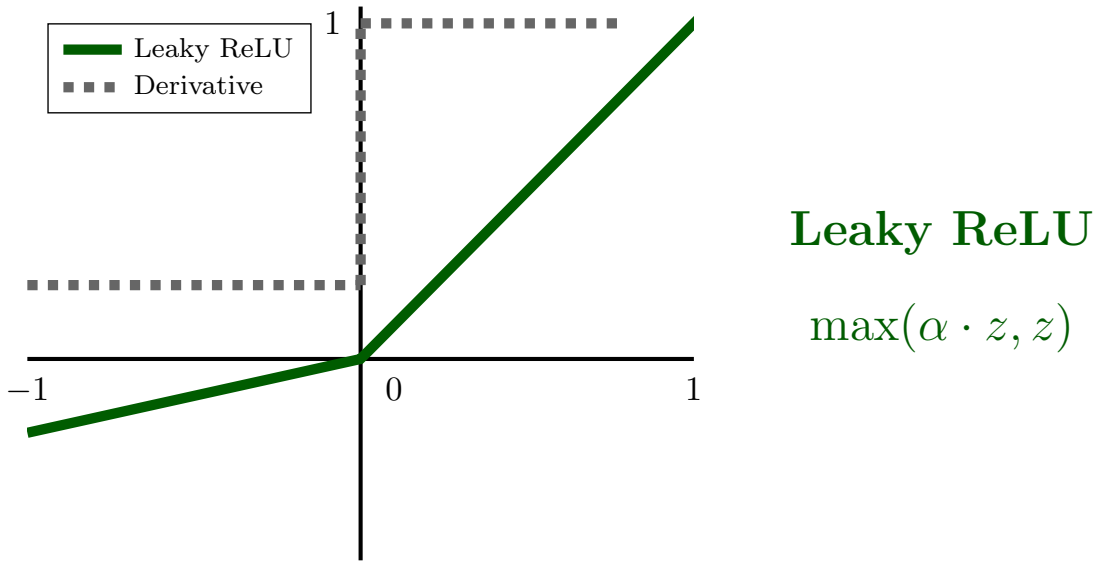
$$h(z, \alpha_i) = \max(0, z_i) + \alpha_i \min(0, z_i)$$

However, generalized rectified linear units are not universally better than ReLU and may lead to other problems, such as additional computational complexity. Some of the improvements that have been made include:

- **Leaky ReLU**: This variation solves the problem of neuron death by setting a_i to a small value, such as 0.01, to maintain a gradient flow even when the input is negative.
- **Parametric ReLU**: This variant allows the parameter a_i to be learned, allowing the network to adapt the gradient of the activation function based on the data.
- **ReLU Randomized**: This variant samples the parameter a_i from a fixed interval during training and keeps it constant during testing, introducing randomness into the model.

In addition, other activation functions have been proposed as alternatives to ReLU, including:

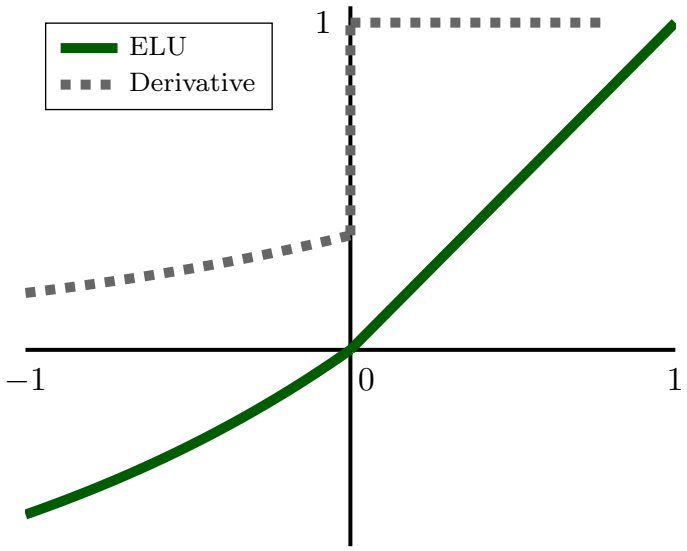
- **Exponential Linear Units (ELU)**: This activation function retains all the advantages of ReLU but avoids the problem of neuron death. However, it requires exponentiation during computation, slightly increasing the computational complexity.
- **Swish**: This function is a smooth, non-monotonic function that has been shown to perform better than ReLU on deeper models, improving performance on datasets such as ImageNet.
- **Linear Units with Gaussian Error (GELU)**: This activation function has been used in state-of-the-art algorithms such as GPT-3 and BERT. It combines the non-monotonic aspect (that helps for the gradient of the function), regularization techniques such as dropout and zone-out, and ReLU benefits to achieve a smoother version of ReLU.



Leaky ReLU Activation Function Plot

ELU

$$\begin{cases} \alpha \cdot \exp(z) - 1 & \\ z & \text{if } z \geq 0 \end{cases}$$



ELU Activation Function Plot

1.5. ARCHITECTURE DESIGN

How do we decide depth and width? Choosing the depth and width of a neural network depends on several factors, including the complexity of the problem to be solved, the availability of training data, and the computational resources available. In principle, how many hidden layer units are sufficient? Theory offers us some answers.

> Cybenko's Theorem (1989), also known as the universal approximation theorem, states that **a two-layer neural network with linear output can approximate any continuous function** over a compact domain with arbitrary precision, provided there are enough hidden units. This theorem has important implications for the design of neural architectures.

Implications: Regardless of the function we are trying to learn, we know that a large multilayer neural network can represent this function. However, there is no guarantee that our training algorithm will be able to learn that function, due to optimization or data overlap issues. Also, the theorem gives no indication of how large the network will be.

So is depth or width more important? There are trade-offs between depth and width of neural networks. For example, deeper networks generally generalize better: imagine a neural network as a set of tools to solve a problem. The more tools you have, the more complex problems you can solve. Then, by adding more "layers" to the network (making it deeper), we could give the network the ability to tackle more difficult problems or capture finer details in the data. In addition, architecture design has a significant impact on performance, for example between convolutional and feedforward neural networks.

2. BACKPROPAGATION

2.1. THE GRADIENT DESCENT ALGORITHM

To train neural networks effectively, we employ the (Stochastic) **Gradient Descent algorithm** in combination with **Backpropagation**. Backpropagation, a method for computing gradients, is crucial for updating the parameters of the neural network during training. We will delve into it shortly, but first, let's explore Gradient Descent and its significance.

2.1.1. Understanding the Gradient

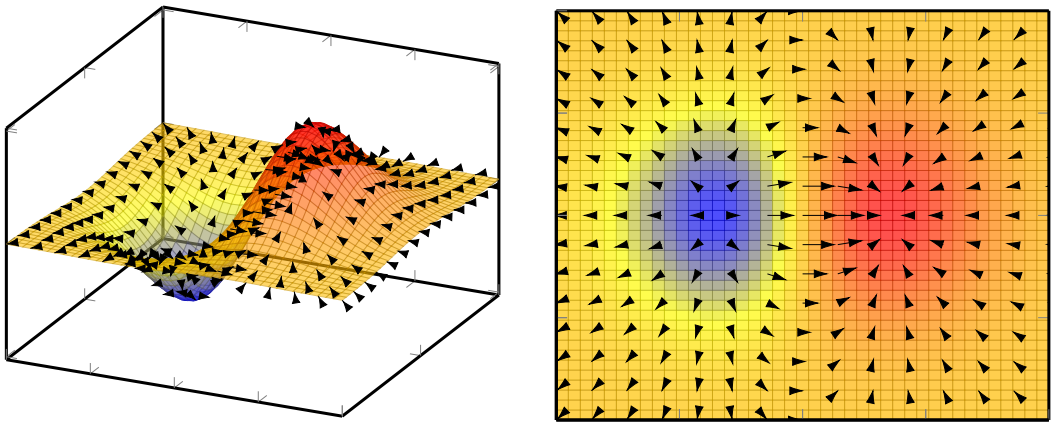
Gradient Descent is a fundamental optimization algorithm used to minimize the loss function of a neural network by iteratively adjusting its parameters. The goal is to find the optimal set of parameters that result in the lowest possible loss. The algorithm works by iteratively moving in the **direction of the steepest descent of the loss function** with respect to the parameters. This direction is determined by the **negative gradient** of the loss function, which is why we will utilize the negative sign in the algorithm.

Mathematically, the update rule for the parameters w in Gradient Descent can be expressed as:

$$\Delta_w L = \left[\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_N} \right]$$

where $\Delta_w L$ represents the change in the loss function L with respect to the parameters w, and $\frac{\partial L}{\partial w_i}$ denotes the partial derivative of the loss function with respect to the i-th parameter w_i.

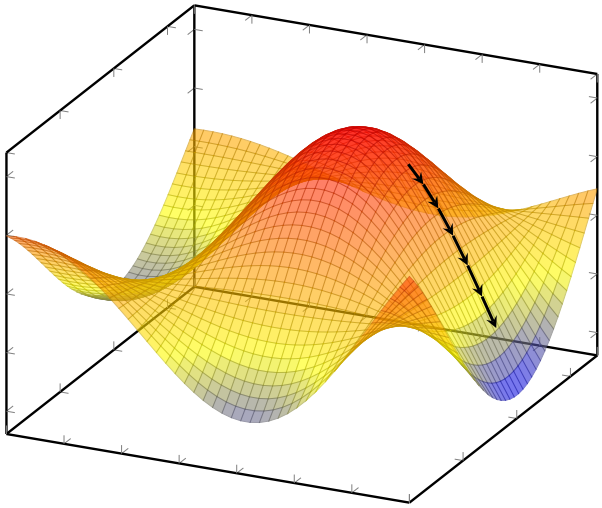
Below we show a figure illustrating the partial derivatives of a function. The figure is represented in two ways: in a 3D graph and in the two-dimensional projection of the same graph (top view). The surface of the graph represents the function, where the lowest point is colored blue and the highest point is colored red. The black arrows extending from the surface indicate the direction and intensity of the gradient change of the function at each point in space. This visualization is crucial for understanding how Gradient Descent finds the fastest direction to reduce loss and update model weights during neural network training.



Visualization of Partial Derivatives in Two Distinct Ways

Each partial derivative measures how fast the loss changes in one direction. When the gradient is zero, the loss is not changing in any direction.

Having understood partial derivatives and the role of the gradient in the optimization process, let's now look at a visual representation of how the Gradient Descent algorithm works. In the image below, the black arrows extending from the surface indicate the **path** that Gradient Descent follows **to reach the minimum**. Each arrow represents the direction and intensity of the change in the loss function at a given point in space.

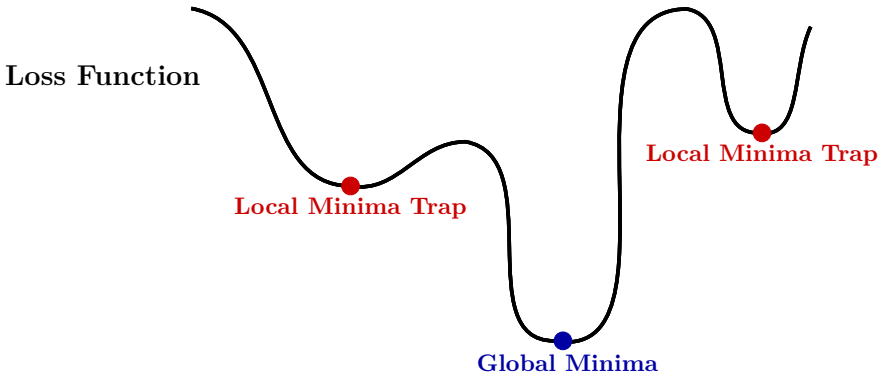


Visual Representation of the Gradient Descent Algorithm

The algorithm proceeds by iteratively updating the parameters according to the gradient direction **until convergence is reached or a predefined number of iterations is completed**. By following this process, Gradient Descent effectively navigates the parameter space to find the optimal configuration that minimizes the loss function.

2.1.2. The Problem of Local Minima

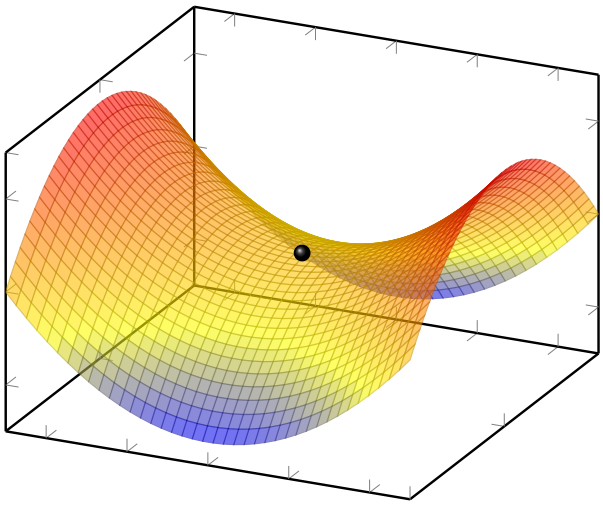
In neural networks, the optimization problem is non-convex, leading to the existence of **local minima**. However, practitioners have found that these local minima are often still effective solutions. Thus, local minima are not typically a significant concern in neural network training. Below, we show an example illustrating the local minima problem for a better insight.



Local and Global Minima

2.1.3. The Problem of Saddle Points

Another optimization challenge arises from saddle points, where some directions curve upwards and others downwards. At a saddle point, the **gradient is zero**, even if the point is not a minimum. saddle points are common in high-dimensional spaces. However, they only become problematic if the optimization algorithm becomes stuck exactly at the saddle point.



Saddle Point Illustration

In the figure, there is a saddle point depicted within the neural network optimization scenario, represented by the black dot.

2.2. BACKPROPAGATION ALGORITHM

2.2.1. Backpropagation Fundamentals

How do we learn the weights of a neural network? Let us examine some ideas that have guided the development of learning algorithms in the context of neural networks.

The initial idea was to **randomly perturb one weight at a time**, evaluating whether that change improved model performance and saving the change. This approach, although similar to an evolutionary process, proved to be very inefficient, requiring numerous passes over the training data for each change in the weights and presenting difficulties in the last stage of learning.

Another idea involved **perturbing all the weights simultaneously** and correlating performance improvement with changes in the weights. However, this method proved equally inefficient and extremely difficult to implement.

Subsequently, a more efficient idea was proposed: to **perturb only the activations**, which are fewer in number than the weights. Although this approach is an improvement over the previous ones, it is still inefficient.

Therefore, how can we achieve more efficient learning? This is where backpropagation comes in.

> Backpropagation is a widely used learning algorithm in neural networks, which consists of three main steps:
> 1. **Forward Propagation**: summation of inputs, production of activations and propagation of output through the network.
> 2. **Error Estimation**: comparison of labels with predictions obtained from the network.
> 3. **Backpropagation** of the error signal and using this signal to update the weights by computing gradients.

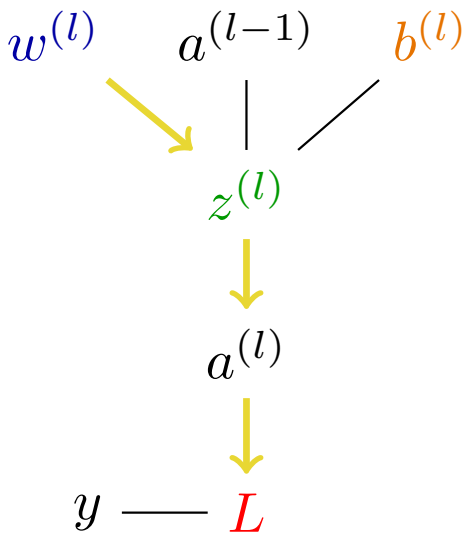
Starting from the training data, we do not directly know the optimal behavior of the hidden units within the neural network. However, using backpropagation, we can calculate how quickly the overall error of the network changes when we change the activity of these hidden units.

Using the derivatives of the error with respect to the hidden activities, we are able to understand **how each hidden unit affects the overall error of the network**. This allows us to gain a clear view of the separate effects that each hidden unit has on the error and how these effects are combined to determine the optimal direction to update the network weights.

Once we have calculated the derivatives of the error with respect to the hidden activities, we gain valuable information to update the weights associated with each connection in the network. This allows us to adjust the weights in a way that minimizes the overall error in the network, moving us closer and closer to the optimal solution for the learning problem.

2.3. EXPLORING BACKPROPAGATION: STEP BY STEP

In this section, we will explore backpropagation of the error through a neural network in detail, starting with the output layer and proceeding to the input layer. We begin with a neural network composed of **one neuron per layer**.



Conceptualisation of Calculations

After performing the forward step, we will get an output from the last activation $a^{(l)}$, which will allow us to calculate the loss L using the desired output y. The latter activation $a^{(l)}$ is calculated using three elements: a weight $w^{(l)}$, a bias $b^{(l)}$, and the activation of the preceding neuron $a^{(l-1)}$. The resulting equations are as follows:

$$z^{(l)} = w^{(l)} a^{(l-1)} + b^{(l)}$$
$$a^{(l)} = \sigma(z^{(l)})$$

Where $z^{(l)}$ is the input of a non-linear function σ, such as a sigmoid or a ReLU.

A way you might conceptualize this is that the weight $w^{(l)}$, the prior activation $a^{(l-1)}$, and the bias $b^{(l)}$ together **allow us to calculate** $z^{(l)}$, which in turn **allow us to calculate** $a^{(l)}$, which together with the desired output y **allows us to calculate the loss** L. This concept is illustrated in the diagram on the left.

Our initial goal during the backward step is to understand how sensitive the loss L is to small changes in the weight $w^{(l)}$, that is, calculate the derivative $\frac{\partial L}{\partial w^{(l)}}$. Conceptually, when you see this kind of formula, we need to think of it as saying to us "**how much does L (numerator) change if we make a small change at $w^{(l)}$ (denominator)?**"

When we calculate this derivative, we note that a small change in weight $w^{(l)}$ **causes a change in** $z^{(l)}$, which in turn **causes a change in** $a^{(l)}$, **directly affecting** the loss L. And so this is where the chain of derivatives rule comes into play!

As can be seen below we can now "chunk" the previous derivative:

$$\frac{\partial L}{\partial w^{(l)}} = \frac{\partial z^{(l)}}{\partial w^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial L}{\partial a^{(l)}}$$

How much does a little variation to $a^{(l)}$ changes L?

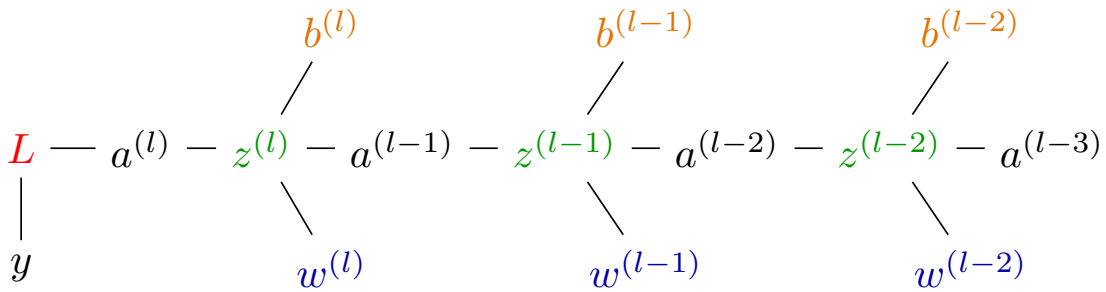
How much does a little variation to $z^{(l)}$ changes $a^{(l)}$?

How much does a little variation to $w^{(l)}$ changes $z^{(l)}$?

What about the bias term? Easy peasy! We use the same method as for the weight term:

$$\frac{\partial L}{\partial b^{(l)}} = \frac{\partial z^{(l)}}{\partial b^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial L}{\partial a^{(l)}}$$

Now what? For the other layers? Let's go back to our scheme for a moment and extend it:



Conceptualisation of Calculations (Extended)

All other weights and biases are in the earlier layers of the network, which means that their influence on cost is less direct. The way we handle them is to first consider how sensitive the cost is to the value of that activation in the penultimate layer, $a^{(l-1)}$, and then consider how sensitive that value is to all the previous weights and biases.

The derivative of cost with respect to that activation looks a lot like what we have already seen:

$$\frac{\partial L}{\partial a^{(l-1)}} = \frac{\partial z^{(l)}}{\partial a^{(l-1)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial L}{\partial a^{(l)}}$$

The trick here is to remember that the activation in the previous layer is determined by its set of weights and biases. For example, there is a long chain of dependencies between the weight $w^{(l-1)}$ and the cost L. The way this presents itself mathematically is that the partial derivative of the cost with respect to that weight appears as a long chain of partial derivatives for each intermediate step, see the image below to visualize the concept graphically.

Here is how you can decompose the derivative:

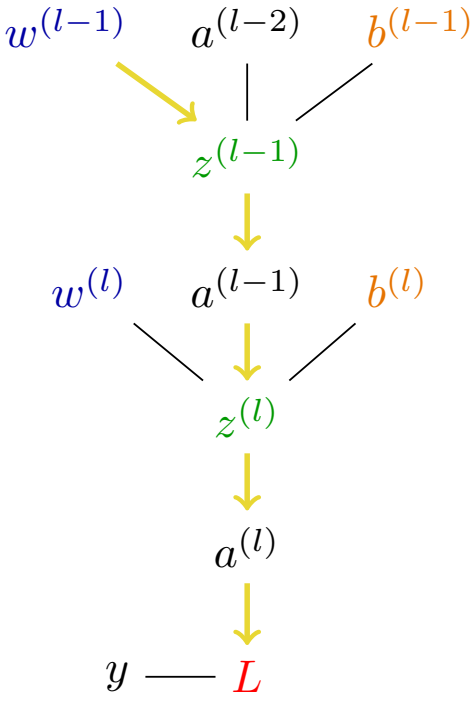
$$\frac{\partial L}{\partial w^{(l-1)}} = \frac{\partial z^{(l-1)}}{\partial w^{(l-1)}} \frac{\partial a^{(l-1)}}{\partial z^{(l-1)}} \underbrace{\frac{\partial z^{(l)}}{\partial a^{(l-1)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial L}{\partial a^{(l)}}}_{\frac{\partial L}{\partial a^{(l-1)}}}$$

By following the dependencies through our tree and multiplying together a long series of partial derivatives, we now **can calculate the derivative of the cost with respect to any weight or bias of the entire network**. *We are simply applying the same idea of the chain rule that we have always used!*

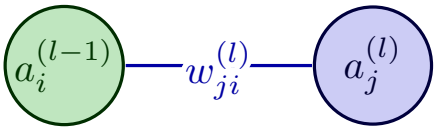
And since we can get any derivative, we can calculate the entire gradient vector:

$$\Delta_w L = \left[\frac{\partial L}{\partial w^{(1)}}, \frac{\partial L}{\partial b^{(1)}}, \ldots, \frac{\partial L}{\partial w^{(l)}}, \frac{\partial L}{\partial b^{(l)}} \right]$$

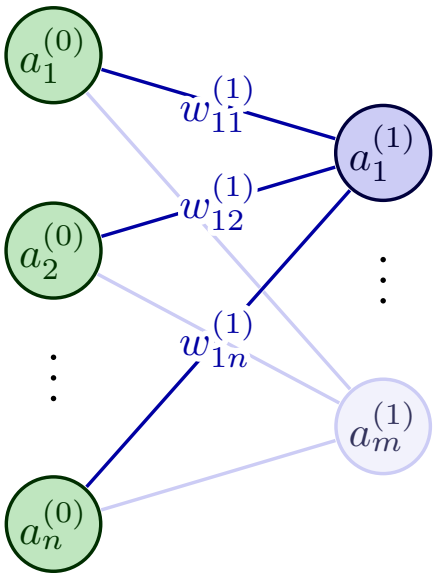
The job is done! At least for this network, now we must add more neurons for each layer. *I know, you are crying inside, but actually, not much changes when we give the layers more neurons: it's just a few more indexes to keep track of* 😄.



Calculations with 3 Layers



Neural Network Indexing



Example of Indexing

When dealing with neural networks with multiple neurons per layer, we have to change the notations a bit.

The activation of each neuron will be denoted with a subscript indicating its position within the layer. Thus, **the superscript of each neuron indicates which layer it is in, while the subscript indicates the specific neuron**.

The weights also need a refresh: additional indices are required to specify their location. In addition to the superscript representing the layer, **two subscripts indicate the edge weight connecting the neuron in layer i to the neuron in layer j**.

I know, these "ji" indices, being backwards, might seem strange or unconventional at first, but they align with the way the weight matrix is usually indexed.

On the left are two images: one formally represents how the nodes and edges of the network are represented, while the other provides an example with two dummy layers, named 0 and 1, with n and m nodes, respectively.

The weighted sum $z_j^{(l)}$ then takes the following form:

$$z_j^{(l)} = \sum_i w_{ji}^{(l)} a_i^{(l-1)} + b_j^{(l)}$$

The new equations turn out to be esentially the same with respect to the case of only one neuron per layer:

$$\frac{\partial L}{\partial w_{ji}^{(l)}} = \frac{\partial z_j^{(l)}}{\partial w_{ji}^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial L}{\partial a_j^{(l)}}$$

Indeed, the expression of the derivative of the chain rule describing the sensitivity of cost with respect to a particular weight is the same as in the previous case. The only difference is that we now have multiple indices, i and j, indicating which weight we are considering.

What changes, however, is the derivative of the cost with respect to one of the activations in the preceding layers, because **the preceding neurons influence the cost function through multiple pathways**.

To understand the sensitivity of the cost function with respect to a certain neuron, it is necessary to sum the influences along each of these different pathways. So, we **sum multiple expressions of different chain rules corresponding to each pathway of influence**.

$$\frac{\partial L}{\partial a_i^{(l-1)}} = \underbrace{\sum_j \frac{\partial z_j^{(l)}}{\partial a_i^{(l-1)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial L}{\partial a_j^{(l)}}}_{\text{Sum over layer } l}$$

This reasoning is carried out using traditional numerical values, but it is evident that in order to efficiently handle the weights and biases of the neural network, it is necessary to use the **matrix approach**, since it allows simultaneous operations to be performed on all the parameters of the network, greatly optimizing the computational process.

3. OPTIMIZATION

In this chapter, we will immerse ourselves in optimization, exploring a wide range of gradient descent algorithms. We will start from the classic batch gradient descent to more advanced ones such as ADAM. But optimization does not stop there: we will also introduce a crucial element, data normalization, which contributes significantly to the effectiveness of our models. *Ready to explore the nitty-gritty of these techniques? Let's dive in!*

3.1. TYPES OF GRADIENT DESCENT

3.1.1. Batch Gradient Descent (BGD)

The Batch Gradient Descent algorithm is one of the most widely used methods for parameter optimization in a neural network. Below we provide the algorithm.

Algorithm **Batch Gradient Descent**

Require: Learning Rate η
Require: Initial Parameters $\mathbf{w}$
while Stopping Criteria not met **do**
  Compute gradient estimate on **N training examples** $\{(\mathbf{x}^{(1)}, y^{(1)}), ..., (\mathbf{x}^{(N)}, y^{(N)})\}$
  $\widehat{\mathbf{g}} \leftarrow \frac{1}{N} \nabla_{\mathbf{w}} \sum_{i=1}^{N} L(f(\mathbf{x}^{(i)}, \mathbf{w}), y^{(i)})$
  Update parameters: $\mathbf{w} \leftarrow \mathbf{w} - \eta \widehat{\mathbf{g}}$
end while

The distinguishing feature of Batch Gradient Descent is that it uses the **entire training dataset to calculate the gradient** of the loss function. This means it requires **more computational resources and computation time** than other methods, but it can also lead to **more stable and accurate convergence**. Therefore, Batch Gradient Descent remains one of the most widely used algorithms for training neural networks due to its conceptual simplicity and its effectiveness in learning model parameters.

N.B. η represents the learning rate, an important parameter that determines the step size that is taken when updating model parameters.

3.1.2. Stochastic Gradient Descent (SGD)

The Stochastic Gradient Descent algorithm, on the other hand, computes and applies parameter updates for each individual training example rather than using the entire training dataset. Below is the algorithm:

Algorithm **Stochastic Gradient Descent**

Require: Learning Rate η
Require: Initial Parameters $\mathbf{w}$
while Stopping Criteria not met **do**
  Compute gradient estimate on **a random training example** $(\mathbf{x}^{(i)}, y^{(i)})$
  $\widehat{\mathbf{g}} \leftarrow \nabla_{\mathbf{w}} L(f(\mathbf{x}^{(i)}, \mathbf{w}), y^{(i)})$
  Update parameters: $\mathbf{w} \leftarrow \mathbf{w} - \eta \widehat{\mathbf{g}}$
end while

As we have mentioned, the distinguishing feature of Stochastic Gradient Descent is the use of **a single training example at a time** to calculate the gradient and update the parameters. This method requires **less computational resources** than Batch Gradient Descent, but can be **more susceptible to fluctuations** during optimisation due to the variability introduced by individual training examples. However, SGD is often preferred when dealing with large data sets or when one wishes to frequently update parameters during training, as it can lead to faster convergence.

3.1.3. Mini-Batch Gradient Descent

To mitigate the problem of noise or fluctuations associated with SGD, we can adopt the **Mini-Batch Gradient Descent**. This variant of the algorithm applies parameter updates using a **fixed-sized subset of the training dataset** at each iteration, known as a mini-batch, rather than using the entire dataset or a single training example. In this way, Mini-Batch Gradient Descent represents a compromise between the stability of Batch Gradient Descent and the computational efficiency of SGD.

The advantages of this approach are many. First of all, the computation time does not depend on the total size of the N dataset, which makes it **suitable even for large datasets**. Furthermore, the method allows **parallel processing**, making it suitable for implementations on parallel hardware such as GPUs. However, this method has still a problem related to the optimization surface: along flat direction the gradient step is small so the algorithm achieves very slow progress, whereas if the surface is steep the may be jittery movements. To solve this problem, *momentum* have been developed.

3.1.4. Stochastic Gradient Descent with Momentum

When the gradient points in a direction where the surface is almost flat, SGD moves very slowly because the updates are small and in regions where the gradient changes rapidly (steep gradients) SGD can fluctuate a lot. To solve these problems, we introduce Momentum!

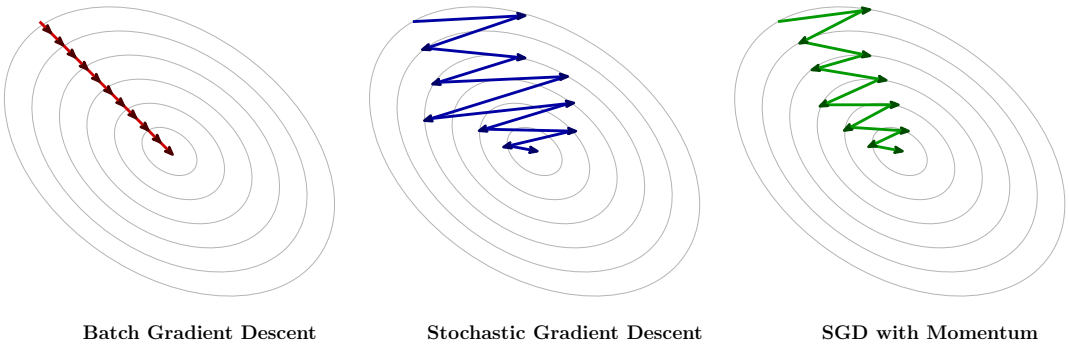
Stochastic Gradient Descent with Momentum is a variant of the gradient descent algorithm that introduces a new variable, called *velocity* (v), which represents an **exponentially decaying moving average** of negative gradients (in simple words, it tracks the progress history of the algorithm). The velocity accumulates the gradient over iterations, and it acts as a kind of "inertia" in the movement of the model parameters during learning, adding an acceleration component to the optimisation process.

Algorithm **Stochastic Gradient Descent with Momentum**

Require: Learning Rate η
Require: Momentum Parameter α
Require: Initial Parameters $\mathbf{w}$
Require: Initial Velocity $\mathbf{v}$
while Stopping Criteria not met **do**
  Compute gradient estimate on **a random training example** $(\mathbf{x}^{(i)}, y^{(i)})$
  $\widehat{\mathbf{g}} \leftarrow \nabla_{\mathbf{w}} L(f(\mathbf{x}^{(i)}, \mathbf{w}), y^{(i)})$
  Update the velocity: $\mathbf{v} \leftarrow \alpha\mathbf{v} - \eta\widehat{\mathbf{g}}$
  Update the parameters: $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{v}$
end while

The parameter α controls the **contribution of the previous velocity** in the current parameter update. If α is greater than η, the current update is more influenced by the previous gradients, thus increasing **convergence stability** and helping the network to **avoid local minima**.

Stochastic Gradient Descent with Momentum, while accelerating the process of reaching the minimum, can sometimes "overshoot" the desired target due to the inertia effect introduced by speed. However, it manages to reach the minimum much **faster than simple SGD**, making it an effective choice for optimising model parameters when training neural networks.



BGD vs SGD vs SGD with Momentum

3.1.5. Stochastic Gradient Descent with Nesterov Momentum

Stochastic Gradient Descent with Nesterov Momentum is a variant of the previous algorithm that adds a correction to the previous momentum before calculating the gradient. It works by firstly taking a step in the direction of the accumulated gradient (point 2 of the algorithm), and secondly calculating the gradient (point 4) and making the correction accordingly (point 5). (in simple momentum, we first compute the gradient and then make the correction)

Algorithm **Stochastic Gradient Descent with Nesterov Momentum**

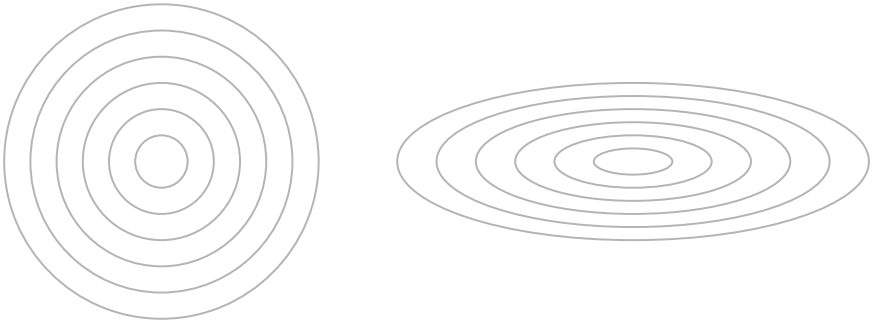
Require: Learning Rate η
Require: Momentum Parameter α
Require: Initial Parameters $\mathbf{w}$
Require: Initial Velocity $\mathbf{v}$
1: **while** Stopping Criteria not met **do**
2: Update parameters temporarily: $\widetilde{\mathbf{w}} \leftarrow \mathbf{w} + \alpha\mathbf{v}$
3: Compute gradient estimate on **a random training example** $(\mathbf{x}^{(i)}, y^{(i)})$
4: $\widehat{\mathbf{g}} \leftarrow \nabla_{\widetilde{\mathbf{w}}} L(f(\mathbf{x}^{(i)}, \widetilde{\mathbf{w}}), y^{(i)})$
5: Update velocity: $\mathbf{v} \leftarrow \alpha\mathbf{v} - \eta\widehat{\mathbf{g}}$
6: Update parameters: $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{v}$
7: **end while**

The main difference between the Nesterov momentum and the standard momentum lies in **where the gradient is evaluated**. In the Nesterov momentum the gradient term is not calculated from the current position in parameter space, but rather from an intermediate position. This approach offers a significant advantage: while the gradient term always points in the optimal direction, **the momentum term may not align consistently**. Consequently, if the momentum goes in the wrong direction or overshoots the target, the gradient term can still "go back" and correct it **in the same update step**.

3.1.6. Adagrad (Adaptive Gradient Algorithm)

The previous techniques that we have seen are based on the intuition of adjusting the learning rate while considering the history of past progress.

However, in many machine learning situations, we are faced with two distinct scenarios: a simpler one, in which all features are equally important, and a more difficult one, in which features have different levels of importance. However, up to this point, we have assigned the same learning rate to all features. *Is this a valid idea?*



All Features are Relevant for the Network

Features possess Distinct Levels of Significance

Adaptive Learning Methods Intuition

Adagrad is an optimisation algorithm that **scales the model parameters by the square root of the sum of the squares of all historical gradient values**. This approach makes it possible to automatically adapt the learning rate for each parameter according to its update rate (in simple words, we take into account how much progress we have already done along each dimension and adjust the learning rate accordingly).

Algorithm **Adagrad**

Require: Learning Rate η
Require: Initial Parameters $\mathbf{w}$
Require: Small Constant δ to avoid division by zero
1: Initialize sum of squared gradients vector: $\mathbf{r} \leftarrow 0$
2: **while** Stopping Criteria not met **do**
3: Compute gradient estimate on **a random training example** $(\mathbf{x}^{(i)}, y^{(i)})$
4: $\widehat{\mathbf{g}} \leftarrow \nabla_{\mathbf{w}} L(f(\mathbf{x}^{(i)}, \mathbf{w}), y^{(i)})$
5: Accumulate: $\mathbf{r} \leftarrow \mathbf{r} + \widehat{\mathbf{g}} \odot \widehat{\mathbf{g}}$
6: Update parameters: $\mathbf{w} \leftarrow \mathbf{w} - \frac{\eta}{\delta + \sqrt{\mathbf{r}}} \odot \widehat{\mathbf{g}}$
7: **end while**

Where $\odot$ denotes element-wise multiplication.

Parameters that have large partial derivatives with respect to loss will see their learning rates reduced quickly, while those with smaller gradients will have higher learning rates, allowing for a **more stable convergence** of the model as not all direction of the optimization surface will have the same importance.

3.1.7. RMSProp (Root Mean Square Propagation)

In many optimisation situations, Adagrad can excessively decrease the learning rate, making convergence difficult, especially in non-convex contexts. This occurs because Adagrad tends to aggressively adapt the learning rate according to the frequency of parameter updates. This means that if a parameter has a large variation in gradients during training (i.e. steep surface along that dimension), Adagrad will drastically reduce the learning rate for that parameter.

To address this problem, RMSProp has been proposed as a variant of Adagrad that **maintains an exponentially decreasing average of past gradients** (similar to momentum $\alpha v - \epsilon \hat{g}$), also called Running Average.

Algorithm RMSProp

Require: Learning Rate η
Require: Decay Rate ρ
Require: Small constant δ to avoid division by zero
Require: Initial Parameters $\mathbf{w}$
1: Initialize running average: $\mathbf{r} \leftarrow 0$
2: **while** Stopping Criteria not met **do**
3: Compute gradient estimate on **a random training example** $(\mathbf{x}^{(i)}, y^{(i)})$
4: $\widehat{\mathbf{g}} \leftarrow \nabla_{\mathbf{w}} L(f(\mathbf{x}^{(i)}, \mathbf{w}), y^{(i)})$
5: Accumulate: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1-\rho)\widehat{\mathbf{g}} \odot \widehat{\mathbf{g}}$
6: Update the parameters: $\mathbf{w} \leftarrow \mathbf{w} - \frac{\eta}{\delta + \sqrt{\mathbf{r}}} \odot \widehat{\mathbf{g}}$
7: **end while**

Thanks to this strategy, RMSProp **avoids an excessive decrease in the learning rate**, as past gradients r (that have been accumulated) counts less, and allows it to adapt better in non-convex contexts and to maintain a more stable convergence during optimisation. It is important to note that there is also a version of RMSProp that includes the Nesterov term, known as **RMSProp with Nesterov Momentum**, which can further improve the performance of the algorithm in certain situations.

3.1.8. ADAM (ADAptive Moments)

Adam is an optimisation algorithm that combines concepts derived from RMSProp and Momentum. However, Adam introduces a fundamental innovation: **bias correction terms** for first and second moments. These terms are crucial since the first and second moments are initialised at zero and require time to "warm up" (i.e. to adapt to the data).

Adam automatically adapts the learning rates for each parameter based on momentum and past momentum estimation, helping to improve the effectiveness of optimisation during the learning process. In Adam, we combine several concepts seen in this section: the **use of the stochastic** in the SGD, the **idea of using momentum**, the **adaptation of the learning rate based on the second moment** and the **use of a bias correction** for both moments.

Algorithm **Adam**

Require: Learning Rate η
Require: Decay Rates for First and Second Moments ρ_1, ρ_2
Require: Small constant δ to avoid division by zero
Require: Initial Parameters $\mathbf{w}$
Initialize first and second moments: $\mathbf{s} \leftarrow 0$, $\mathbf{r} \leftarrow 0$
Initialize time step $t \leftarrow 0$
while Stopping Criteria not met **do**
  Compute gradient estimate on **a random training example** $(\mathbf{x}^{(i)}, y^{(i)})$
  Compute the gradient: $\widehat{\mathbf{g}} \leftarrow \nabla_{\mathbf{w}} L(f(\mathbf{x}^{(i)}, \mathbf{w}), y^{(i)})$
  Update time step: $t \leftarrow t + 1$
  Update **biased** first moment estimate: $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1)\widehat{\mathbf{g}}$ {**Momentum Idea**}
  Update **biased** second moment estimate: $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2)\widehat{\mathbf{g}} \odot \widehat{\mathbf{g}}$ {**RMSProp Idea**}
  Correct bias in first moment: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1-\rho_1^t}$
  Correct bias in second moment: $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1-\rho_2^t}$
  Update parameters: $\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{\hat{\mathbf{s}}}{\delta + \sqrt{\hat{\mathbf{r}}}}$
end while

3.2. NORMALIZATION

In the world of neural networks, optimization is crucial to model success. However, simply adopting SGD may not be enough. Deep neural networks are known to be difficult to train, for example, one of the main problems we face during optimization is the so-called "Covariate Shift."

3.2.1. The Problem of Covariate Shift

When training a neural network, it is important to keep the distribution of input data stable throughout the training process. If the distribution changes significantly during training, the model may have difficulty generalizing well to new data, as patterns learned during training may no longer be relevant or representative.

The main problem arising from the covariate shift is that it can lead to a situation where **most of the input data falls into non-linear regions** of the neural network's activation function. This can significantly **slow down the learning process** as the network may require multiple iterations to adapt its weights effectively to the new input distributions.

To address the Covariate Shift problem and improve the training of neural networks, a fundamental concept was introduced: **Batch Normalization**.

3.2.2. Batch Normalization

It is a method for reconfiguring the parameters of a deep network, which **can be applied to the input layer as well as to any hidden layer**.

The key concept behind Batch Normalization is the standardization of the inputs of a network layer. This is done by subtracting the mean of the inputs and dividing by the standard deviation. In mathematical terms, if $\mathbf{H}$ represents the outputs of a layer, μ and σ represent the mean and standard deviation (both vectors) calculated on the columns

(features) of $\mathbf{H}$, then the transformation is given by:

$$\mathbf{H}' = \frac{\mathbf{H} - \mu}{\sigma}$$

where

$$\mu = \frac{1}{m}\sum^{j} \mathbf{H}_{:,j}$$

$$\sigma = \sqrt{\delta + \frac{1}{m}\sum_{j}(\mathbf{H} - \mu)_j^2}$$

The term δ in the formula represents the usual small constant added to avoid numerical problems when the calculated standard deviation is very close to zero. Standardizing the output of a unit could limit the expressive power of the neural network because, without introducing additional parameters, normalization could bring all features in a given stratum to a common scale, making it more difficult for the neural network to capture complexity and variation in the data. This is because standardization unifies features, bringing them to **a mean of zero and a standard deviation of one**, potentially "squeezing" them into a narrower range.

To mitigate this problem and allow the neural network to maintain its expressive capability, two additional parameters are introduced in Batch Normalization: the **scaling factor** (γ) and the **bias** (β). These parameters allow the network to learn a linear transformation of the normalized output, thus allowing greater flexibility in data representation.

$$\text{Output} = \gamma\mathbf{H}' + \beta$$

Essentially, the scaling factor and bias allow the network to adapt to a wide range of data while maintaining its ability to express and learn complex patterns as they are learned during the back-propagation process.

During training, back-propagation is performed through normalized activations and these two parameters (γ and β) are also learned. During testing, on the other hand, running averages of μ and σ collected during training are used to evaluate new entries.

3.2.3. Batch, Layer or Instance Normalization?

There are two famous variants of Batch Normalization: **Layer Normalization** and **Instance Normalization**. Briefly summarised, the difference is that Batch Normalization normalises across the entire batch, Layer Normalization normalises across the entire layer and Instance Normalization normalises each instance individually. *I know, it's not very intuitive, so let's do a more detailed analysis!*

To explain the various concepts, we will use a tuple **(N,C,H,W)**, which is commonly used to represent multidimensional data such as images (*which will be the focus of our example to give us a better understanding*). This representation is nothing more than a tensor and is common in frameworks such as TensorFlow and PyTorch to handle input data in neural networks. Here is what each dimension represents:

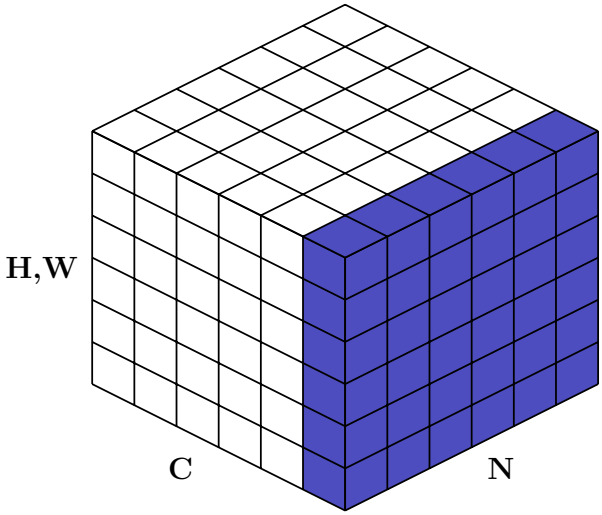
- **N**: Represents the **batch size**, i.e. the number of instances (or samples) within the batch. For example, if N=32, this means we are working with a batch of 32 images.
- **C**: Indicates the **number of channels** (feature maps) in the data. In a color image, we will typically have 3 channels for the colors red, green and blue (RGB). In general, in the input data, the channel can represent not only the color, but various aspects or features of the input, such as depth, features extracted from convolutional layers, etc.

- **H**: Represents the **height of the data**. In an image, it corresponds to the number of rows of pixels present.
- **W**: Represents the **width of the data**. In an image, it corresponds to the number of columns of pixels present.

H and W will be merged together (*Sorry, but my drawing skills stop at 3 dimensions*). We will also use a small example: let us consider a batch with 10 images, each of which has 3 features (RGB) and the image size is 100x100. Let's start then:

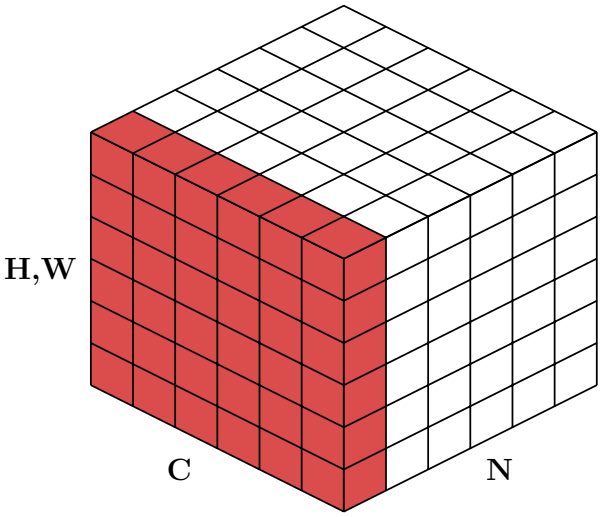
Batch Normalization (BN):

Consider the entire batch of size N. Each example in the batch has C channels, each of which has an image of size HxW. With Batch Normalization, we normalise the data on each channel across the entire batch. Then, for each channel, we calculate the mean and standard deviation over all examples in the batch and normalise the data on that channel accordingly.



Batch Norm Multidimensional Data

Taking our example, we would have a tensor of size (10, 3, 100), where: 10 represents the batch size (N), i.e. the number of images in the batch, 3 represents the number of channels (C), such as the three color channels: red, green and blue, 100 represents the height (H) of the image and 100 represents the width (W) of the image. For each color channel in each image, the mean and standard deviation on all pixels corresponding to that color channel in all images in the batch would be calculated. The pixel values for each color channel **in each image** would then be normalised against these calculated mean and standard deviations.



Layer Norm Multidimensional Data

Layer Normalization (LN):

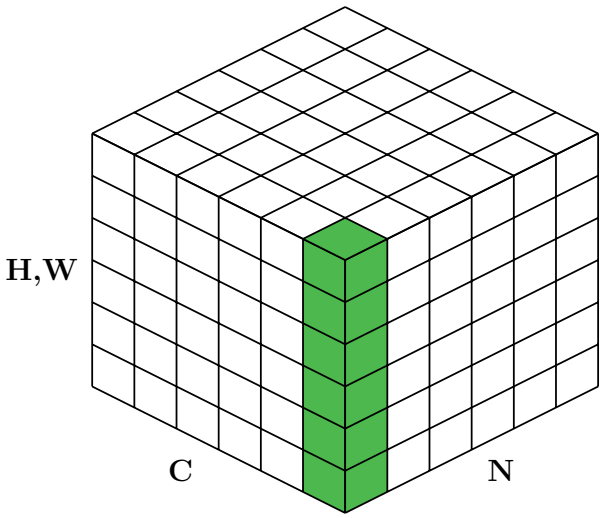
Here, we consider each example in the batch separately. Each example has C channels (*features!*), each of which has an image of size HxW. With Layer Normalization, we normalise the data of each channel for each example. Then, for each channel, we calculate the mean and standard deviation over the entire input of that example (i.e. over all positions of HxW) and normalise the data of that channel accordingly.

Once again, we would have a tensor of size (10, 3, 100). However, each image in the batch

is considered separately. For each image, the pixel values in each color channel would be normalised with respect to the distribution of pixel values within the same image. Thus, the pixel values in each image would be normalised to the distribution of pixel values within the image itself, **completely ignoring the other images in the batch**.

Instance Normalization (IN):

We consider each spatial position (HxW) separately for each example in the batch. Each position has C channels. With Instance Normalization, we normalise the data of each channel for each position in the image. Then, for each channel, we calculate the mean and standard deviation over all positions in the image for each example in the batch and normalise the data for that channel accordingly.



Instance Norm Multidimensional Data

Here again, we have the same size tensor (10, 3, 100). Now, each pixel in each color channel of each spatial position in the image is considered separately for each image in the batch. The pixel values at each spatial position (HxW) of **each color channel in each image** would be normalised to the corresponding pixel values at the same positions in the other images in the batch.

N.B. A more modern technique is **Group Normalisation (GN)**, which can be considered as a mix between Layer Normalisation and Instance Normalisation. With GN, the data channels are considered separately, as in Layer Normalisation, but instead of operating on an entire layer, the **channels are divided into groups** and the normalisation statistics within each group are calculated independently. This approach is reminiscent of the idea of instance-based normalisation, where each instance is considered separately, but instead of normalising individual instances, GN normalises groups of channels. This makes it useful in scenarios where the dimensionality of the data is high and normalisation on entire layers may be excessive.

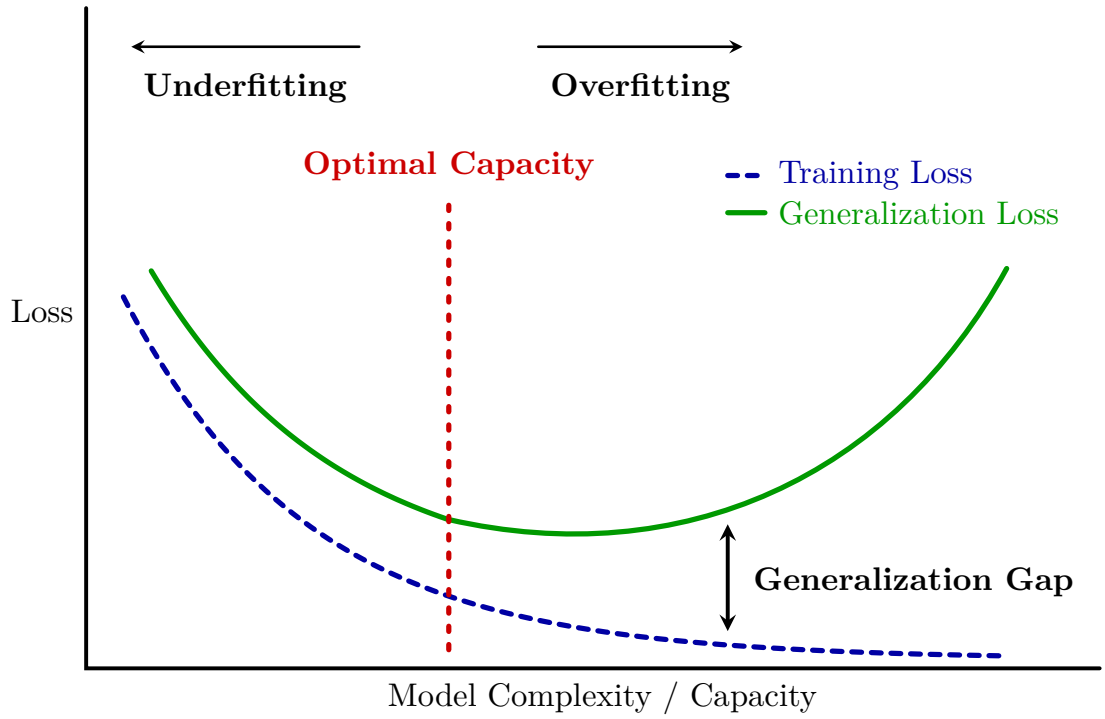
4. REGULARIZATION

4.1. MODEL CAPACITY AND OVERFITTING

The capability of a model refers to its **ability to fit a wide range of functions**. Indeed, it represents the intrinsic complexity of the model and its ability to capture relations in the data. *In other words, it describes how complex is the function it represents.*

Models with low capacity may have difficulty adapting to the training set, while those with high capacity may suffer from overfitting, that is, overfitting to the training data leading to poor generalization to new data.

Model capacity is closely related to the hypothesis space, which represents the set of all features that the model can potentially learn. A model with a higher capacity will have access to a larger hypothesis space, allowing it to learn more complicated relationships in the data.



Complexity of a Model with respect to the Losses

In the figure above, it is evident that as the complexity of the model increases, the loss on the training set tends to decrease, while the loss on generalization (e.g., on the test set) may increase, causing a gap between the losses known as the **generalization gap**.

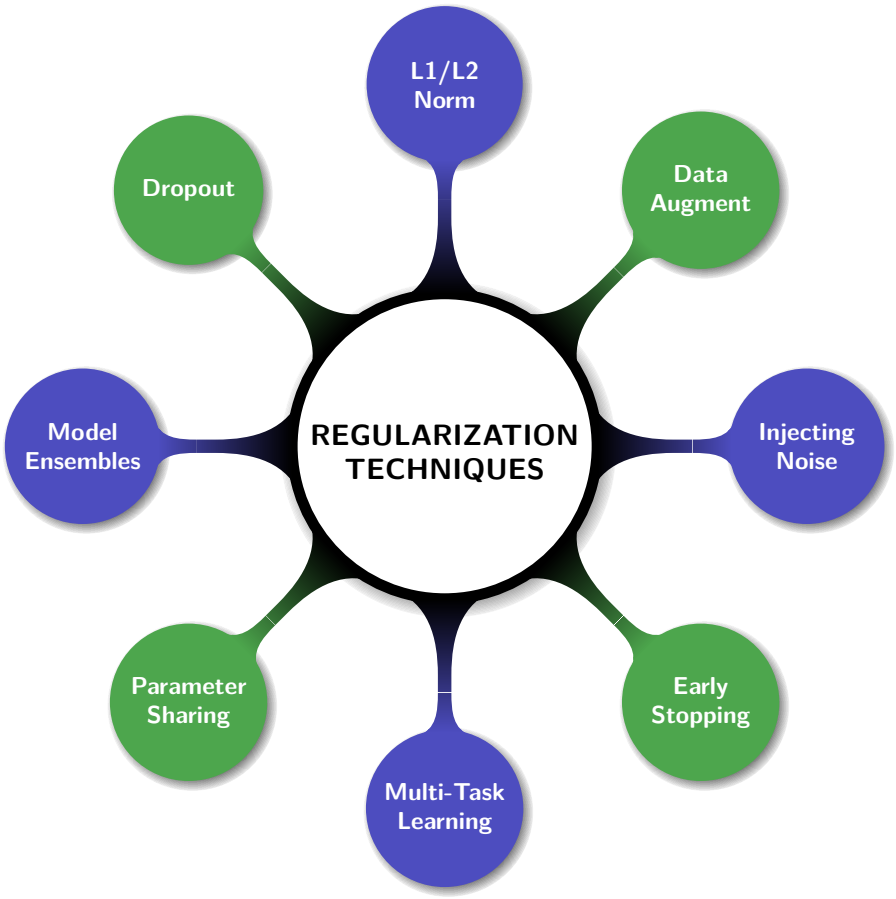
To deal with overfitting, it is important to choose a model with the right capacity. This means setting the capacity so that it is high enough to capture the main regularities in the training data, but not so high that it also captures spurious regularities or noise in the data. *Basically, it is a matter of finding a balance between the complexity of the model and its capacity for generalization.*

In addition to controlling model capacity, there are other strategies to reduce overfitting.

For example, **collecting more training data** can help the model better capture the variety of relationships in the data. Alternatively, **ensembling techniques**, such as averaging many different models (e.g. bagging), can be used. These strategies help avoid overfitting to training data, allowing the model to generalize better to new data.

Are there alternatives to cope with overfitting? Yes, regularization techniques! These provide effective tools for optimizing model performance. In the next sections, we will dive into the details of these powerful assets!

To give you a taste, here is an overview of the various regularization strategies we will see:



Different Regularization Strategies

4.2. PARAMETER NORM PENALTIES

We start with this first type of normalization in which we add a penalty to the loss function during training, which then **affects the size of the weights (parameters)** of the model. Here is the general formula for the error function:

$$\tilde{L}(\mathbf{W}) = L(\mathbf{W}) + \lambda\Omega(\mathbf{W})$$

The **Regularization L2**, also known as **Ridge**, adds a term to the loss function proportional to the **square norm of model weights**. Mathematically, the formula for L2 regularization is given by:

$$\Omega(\mathbf{W}) = \frac{1}{2} \sum_{l} ||\mathbf{w}_l||^2$$

This is inserted into the error function, obtaining the regularized loss function $\tilde{L}(\mathbf{w})$:

$$\tilde{L}(\mathbf{W}) = \sum_{i=1}^{n} L(f(x_i, \mathbf{W}), y_i) + \frac{\lambda}{2} \sum_{l} ||\mathbf{w}_l||^2$$

Where:

- $\sum_{i=1}^{n} L(f(x_i, \mathbf{W}), y_i)$ represents the sum of losses on all training data (x_i, y_i).
- $\frac{\lambda}{2} \sum_{l} ||\mathbf{w}_l||^2$ represents the L2 regularization term, where $|\mathbf{w}_l||^2$ is the squared norm of the model weights and λ is the regularization parameter.
- The sum over l represents the sum over all the weights of the model.

λ is **the hyperparameter that controls the importance of regularization**, i.e., it determines how much weight to give to the penalty of the model weights with respect to performance on the training data.

The use of L2 regularization leads the model weights to **tend to smaller values**, thus avoiding extremely high values that could lead to overfitting. In addition, this technique promotes the creation of simpler models.

On the other hand, the **Regularization L1**, also known as the **Lasso**, adds a term to the loss function proportional to the **absolute norm of model weights**. The L1 regularization formula is expressed as:

$$\Omega(\mathbf{W}) = \sum_{l} |\mathbf{w}_l|$$

Again, we insert in the loss function:

$$\tilde{L}(\mathbf{W}) = \sum_{i=1}^{n} L(f(x_i, \mathbf{W}), y_i) + \lambda \sum_{l} |\mathbf{w}_l|$$

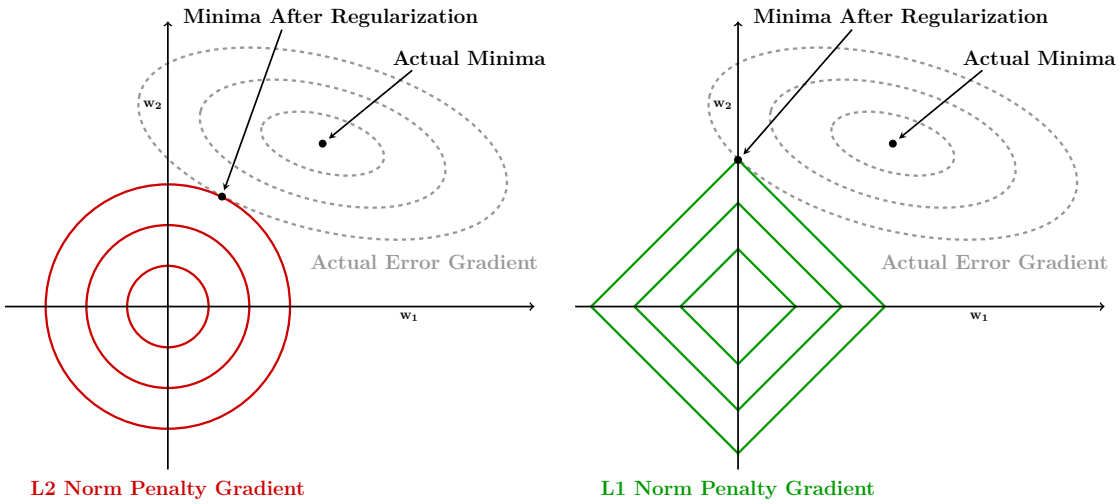
Where:

- $\sum_{i=1}^{n} L(f(x_i, \mathbf{W}), y_i)$ represents the sum of losses on all training data (x_i, y_i).
- $\lambda \sum_{l} |\mathbf{w}_l|$ represents the L1 regularization term, where $|\mathbf{w}_l|$ is the absolute norm of the model weights and λ is the regularization parameter.
- The sum over l represents the sum over all the weights of the model.

Unlike L2 regularization, L1 regularization favors **sparsity of weights**, that is, many weights tend to become exactly zero. This behavior is also useful for feature selection, since it makes the features most relevant to the model more obvious.

To summarize, here is a figure that visually shows how L1 and L2 regularization affect model weights differently:

The plot represents the space of model weights/parameters.



L2 Regularization and L1 Regularization Plots

In the center of the gradient error is the point where the model error is lowest. At this point, the model shows the best performance on the training data.

Next, you will notice an overlap between the gradient error and the gradient of the L1 or L2 norm penalty. This is an optimization point where the overall model error is minimized, also taking into account the regularization penalty. Here the **right balance between error reduction and model complexity** is achieved, thus avoiding both underfitting and overfitting.

For **L1 regularization**, it will be observed that many of the weights in the model will touch the L1 Norm function in the zero abscissa, representing null weights. This indicates that L1 regularization **favors sparsity of the weights**.

For **L2 regularization**, you will notice a gradual reduction of the weights, but none will be exactly zero. This represents that L2 regularization acts more uniformly, **reducing the magnitude of the weights without completely zeroing them**.

4.3. DATA AUGMENTATION

One of the main challenges in training models is the availability of sufficient and diverse training data. Data augmentation is a strategy that aims to improve model generalization by **increasing the amount and variety of training data**. The idea behind it is that the more training data we have, the better the model can learn to generalize to new data not seen during training.

Data augmentation is particularly effective in classification tasks, where it is relatively easy to generate new samples by transforming existing ones. This approach is based on the fact that the **main task of the classifier is to be invariant to a wide range of transformations of the input data**. For example, in the case of images, we can generate new samples by applying transformations such as rotation, translation, color change, scaling, and so on. The goal is to create a diversity of samples that cover as much of the input data space as possible, thus helping the model learn a more robust and general representation of discriminative features.

Some problems may require special precautions during data augmentation. For example,

in the context of character recognition using datasets such as MNIST, it is important not to apply transformations that would change the class, such as transformations that might turn a '6' into a '9'.

Data augmentation can also be applied to different types of data, such as **acoustic and textual data**. For example, for acoustic data, transformations such as pitch shift, stretching time and dynamic range compression can be applied. For textual data, one can use techniques such as substitution, deletion, and letter insertion, or use synonyms and modify adjectives.

4.4. INJECTING NOISE

Gaussian noise as input can be used as a regularizer in neural models. Specifically, each component of the input x_i can be modified by adding Gaussian noise $\mathcal{N}(0, \sigma_i^2)$, so the equation becomes:

$$x_i = x_i + \mathcal{N}(0, \sigma_i^2)$$

As for the output y_j of a layer, it can be expressed as the sum of the original output and the Gaussian noise **amplified by the weight squared**, so the equation becomes:

$$y_j = y_j + \mathcal{N}(0, w_i^2 \sigma_i^2)$$

where $N(0, \sigma_i^2)$ represents Gaussian noise with zero mean and variance σ_i^2, w_i represents the weight associated with the input x_i.

This contributes additively to the quadratic error, causing the model to minimize the overall error, which then **tends to minimize the square weights as well**. This effect is similar to L2 regularization through noisy inputs.

High levels of noise generate a smoother function. This is useful in applications where a smooth function is preferred.

In **autoencoder models for denoising**, noise injection is a common practice. The goal is to teach the neural network to reconstruct the original input from the noisy input. This technique is often used in unsupervised learning algorithms.

But can noise only be added to inputs? Absolutely not! It can also be added to weights and labels in neural models! Adding noise to the inputs makes the learned function smoother, while **adding noise to the weights encourages the network to be more robust to random variations in the training data** (we get a kind of "flexibility" in the model's weights). This can be especially useful when the model is in regions of the weight space where variations in individual weights do not significantly affect the model output. **Adding noise to labels can discourage overconfident model behavior**, such as during cross-entropy loss learning, thus preventing overestimation of class probabilities (cross-entropy assigns very large values for the correct class and very small values for the wrong classes). A common practice in this case is the application of "**label smoothing**", which is the random modification of labels.

4.5. EARLY STOPPING

Early Stopping is a technique used in model training to prevent the model from becoming too specialized on the training data. This technique is based on the idea of monitoring the performance of the model on a validation dataset during training and **stopping training when the performance on the validation set starts to deteriorate, even if the performance on the training set continues to improve**.

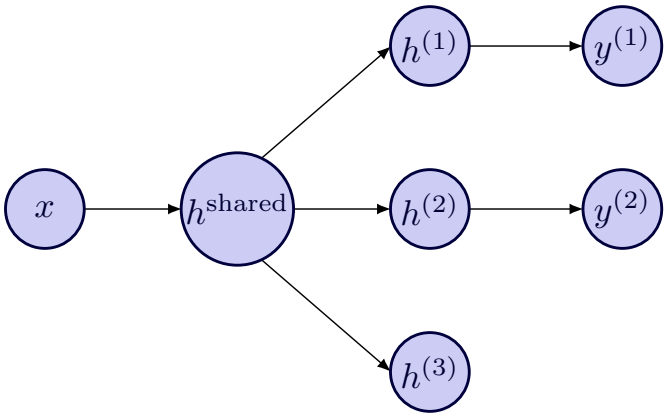
The number of training steps is seen as another **hyperparameter that needs to be optimized**. The Early Stopping procedure can be summarized in the following steps: It starts with small initial weights. Each time the error on the validation set improves, you keep a copy of the model parameters. When training ends, you return the parameters **for which the validation error is smallest**.

However, it is **important to select the stopping point correctly**, since an early stopping point could prevent the model from learning important information, while stopping too late could lead to overfitting. Consequently, the number of epochs or iterations before stopping must be carefully selected during the training phase.

4.6. MULTI-TASK LEARNING

Multi-Task Learning is a learning technique where the model is trained to do **more than one task at the same time**. For example, we might want to train a model to recognize objects in an image and simultaneously identify their location. This means that the model learns to perform more than one "task" during the same training process.

In Multi-Task Learning, several tasks **share the same set of input data and some intermediate layers of the neural network**, but may have **separate final layers for each task**. In this case, the parameters can be of two types: **task-specific** or **generic** (shared among all tasks). The former are designed to perform a specific task and learn only from data related to that task (typically found in the upper layers of the network), while the latter are shared across all tasks and can benefit from the combined data of all tasks (typically found in the lower layer of the network).



Visualization of Multi-Task Learning

Multi-Task Learning often compares with self-supervised learning. While in Multi-Task Learning multiple tasks are taught at the same time, **in self-supervised learning the model learns to represent data in such a way that some information is "hidden"**

and must be predicted. For example, it might train a model to predict missing parts of a picture.

In Multi-Task Learning, it is possible to apply **soft constraints on model parameters**. This means that we can enforce some parameters to be similar to others (this is called **"Paramaters Tying"**). For example, if we have multiple models performing the same type of classification but with slightly different data, we can force the parameters to be similar to ensure consistency across models.

4.7. PARAMETERS SHARING

Parameter Sharing is a technique used in neural networks to enforce that **certain parameters within the network are exactly the same**. This is done for several reasons, including reducing model complexity and sharing relevant information between different parts of the network.

Sharing parameters within the network offers an advantage over "Paramaters Tying", since only a subset of the parameters need to be stored in memory.

In convolutional neural networks (CNNs), used primarily in computer vision, parameter sharing is a key feature. This is based on two main properties of CNNs:

- **Local Connectivity**: Neurons in one layer are **connected to only a small region of the previous layer**. This feature allows CNNs to capture local image features.
- **Weight Sharing between Spatial Positions**: This feature allows CNNs to learn displacement-invariant filter kernels, reducing the number of parameters needed and ensuring that the model is **able to recognize the same features in different parts of the image**. For example, a picture of a cat remains a picture of a cat even if it is shifted by some pixel.

4.8. MODEL ENSEMBLES

In this technique, instead of relying on a single model, we train several models separately and then have all the models vote on the output for the test examples. The idea behind it is that **different models will usually make different errors on the test set**. Therefore, by combining the predictions of multiple models, a better overall result can be obtained. *This technique is similar to consulting multiple experts to get more reliable advice.*

One of the most common methods of implementing Model Ensembles is **Bagging** (Bootstrap Aggregating). In Bagging, several instances of the same model type are trained on **random subsets of the training set (with replacement)** and then the **predictions of all models are combined**.

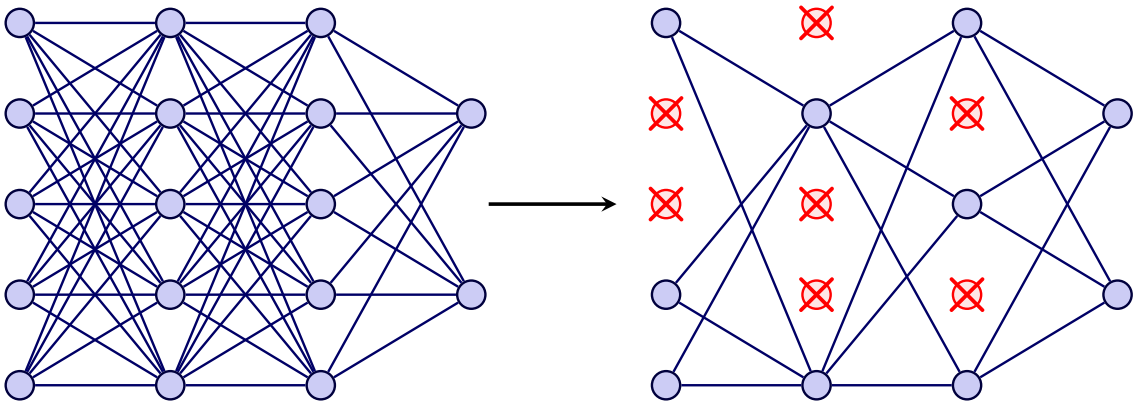
However, this method requires more computational resources and can be more complex to implement and manage than a single model.

4.9. DROPOUT

The basic idea of Dropout is to **randomly turn off some neurons** during the forward pass of the network. This process is **stochastic**, since the choice of neurons to be eliminated is random. By forcing the network to work with redundant representations, Dropout prevents hidden neurons from fitting too closely to each other and **forces them to focus on extracting more useful features** for the task.

Dropout can be viewed as training a large set of models, each of which shares network parameters. Each binary Dropout mask leads to a different model, and for each example, the network is trained using a different Dropout mask. *Each value in the mask represents whether the corresponding neuron is active (1) or deactivated (0) during the forward pass of the network.*

Dropout is similar to Bagging, but is more practical. Whereas in Bagging one defines k different models and builds k different datasets by sampling from the dataset with replacement, **Dropout aims to approximate this process with exponentially large numbers of neural networks**.



Visualization of the Dropout Technique

During training, at each step, a binary Dropout mask with a certain inclusion probability is randomly extracted for each neuron. This mask is then used to deactivate some neurons during the forward transition of the network. During testing, ideally we would like to integrate all probability contributions for all masks, but this is **computationally expensive**. Therefore, we use a **Monte Carlo approximation**, performing many forward steps with different Dropout masks and averaging all predictions. It has been shown that this procedure is equivalent to taking the average of all possible neural networks.

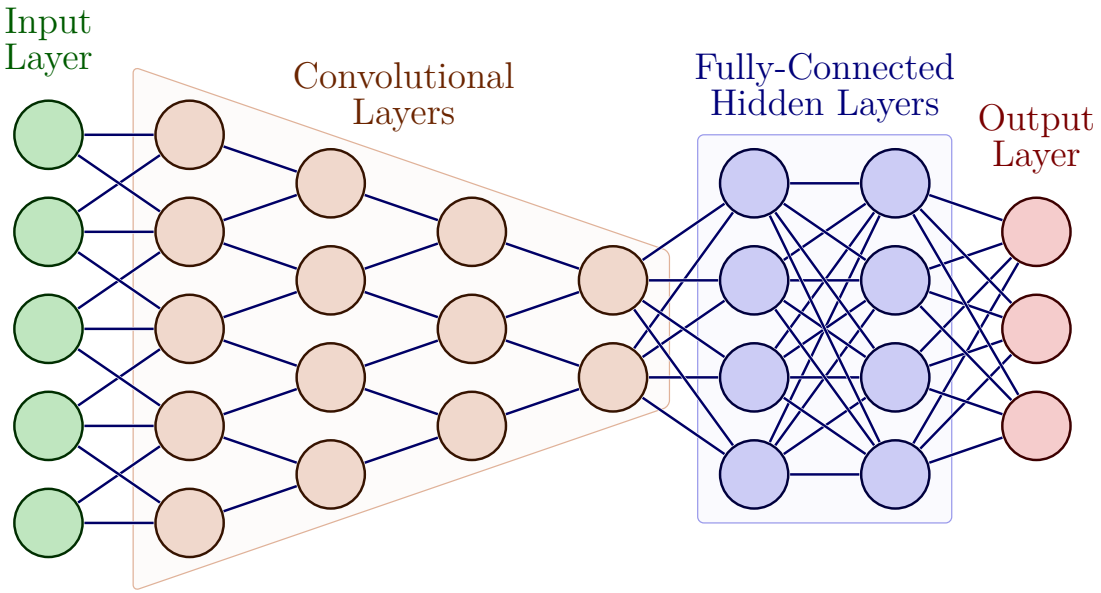
5. CONVOLUTIONAL NEURAL NETWORKS

Welcome to the second part of the course, in which we delve into the intricate world of **different neural network architectures**. *If things seemed complicated before, they will get even more challenging here.* 🥲

Convolutional Neural Networks (CNN) handles **structured data**, such as images. Various architectures have been developed, each attempting different methods to learn effective representations of structured data. In the case of images, these representations capture **spatially local dependencies** between pixels. The low-level layers of a CNN extract local features, while the high-level layers extract learn global patterns.

To understand Convolutional Neural Networks, think of our brain. When we look at an image, the brain processes it in different parts. At first, it recognises simple things like lines and angles. Then, it moves on to more complicated things, until it understands the complete object.

The cells in the brain are like small parts that only look at one part of the image. This helps to focus only on the important things. There are two main types of these cells: some look at small and close things, others look at larger things and care less about the exact position.



Example of a Convolutional Neural Network

Convolutional Neural Networks represent a **specialised version of feedforward neural networks**. If we were to apply a feedforward neural network to process an image, we would lose important information on the spatial relationships between pixels and would have to deal with a large number of parameters due to the need to treat the image as a long input vector. This situation therefore necessitates a complete redesign of the architecture.

CNNs are designed with some specific features, discussed in the previous chapter, that define their recurrent module. These features include **local connectivity** and **weight sharing**, which allow these architectures to effectively capture the spatial characteristics of images without having to manage an excessive number of parameters.

5.1. CONVOLUTIONS

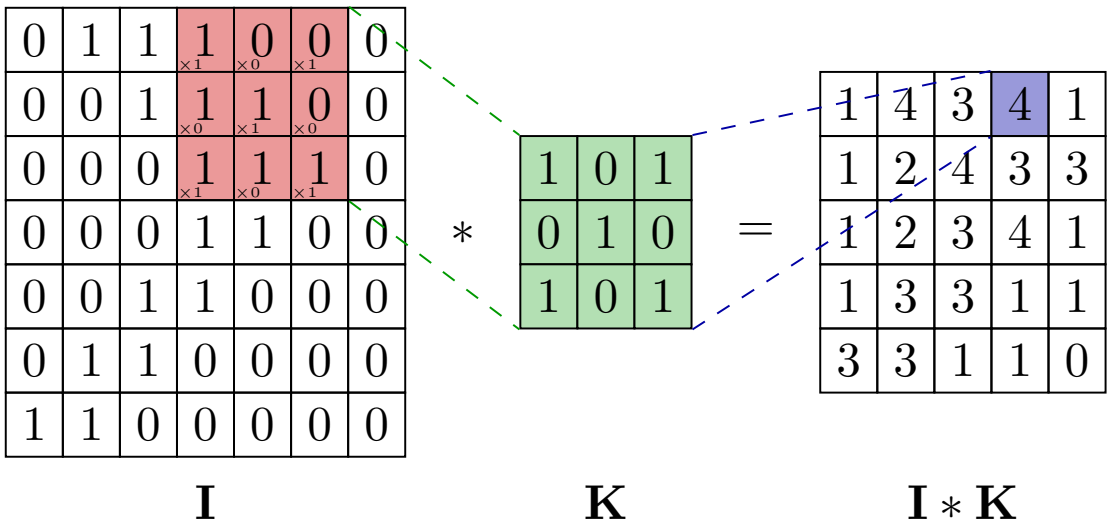
Convolutional Neural Networks basically **replace general matrix multiplication with convolution in at least one of their layers**. A convolution is an operation that takes a matrix (a portion of an image corresponding to the size of the so-called kernel) as input and produces a scalar by calculating the scalar product between the pixel values of the area and the kernel parameters. By progressively moving the kernel, the final output of the convolutional layer will be a matrix.

Formally, a Convolutional Layer consists of a set of filters, each covering a small spatial portion of the input, known as the **receptive field**. Each filter is convoluted along the dimensions of the input data, producing a multidimensional feature map representing the filter's responses to various fragments of the input data. **During training, the network learns filters that are activated in response to specific features at specific spatial locations**.

This concept can be formalised as follows:

$$S(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

Convolutional layers help to extract local features from an image using filters, which are learnt during training as they store weights. However, in a CNN, **filters** are not learnt randomly; rather, they **are optimised to best fit the architecture, loss function and data**.



Example of a 2D Convolution

An example of convolution is shown in the figure above, in which we have this kernel and calculations:

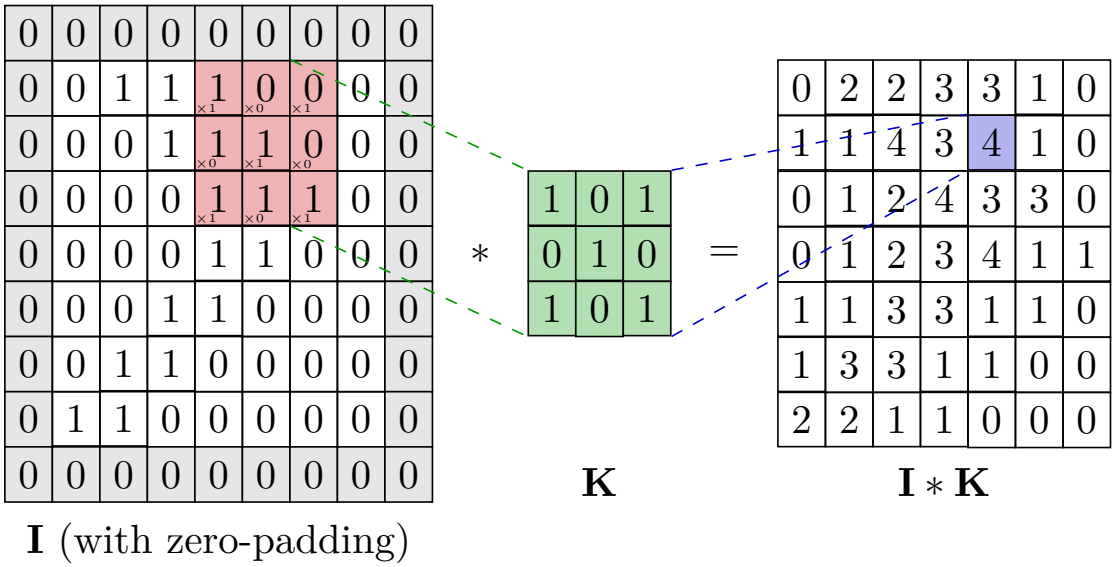
$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

$$\begin{aligned} &1 \times 1 + 0 \times 0 + 0 \times 1 \\ &+1 \times 0 + 1 \times 1 + 0 \times 0 \\ &+1 \times 1 + 0 \times 0 + 1 \times 1 \quad = 4 \end{aligned}$$

For the input considered, the convolution is calculated through the scalar product between the kernel values and the corresponding input values. The result of this calculation is 4, which represents the output of this specific convolution.

It is important to know two fundamental parameters used to control the size of the convolution output and influence the size and shape of the features extracted by the convolutional neural network, namely:

- **Padding**: Padding refers to the addition of values (commonly zeros) around the edges of the input image before applying the convolution. This is useful for controlling the size of the convolution output. Padding may be "**Valid**" (no padding) or "**Same**" (padding so that the output is the same size as the input, also called **zero-padding**).
- **Stride**: The stride indicates the number of pixels in which the filter (kernel) is moved along the input image during the convolution operation. **A stride value greater than one** results in a larger displacement, reducing the size of the output. **A stride value of one** is the most common and produces an output with the same size as the input.



Example of a 2D Convolution with Zero-Padding

The output of the convolutional layer depends on the size of the kernel K, the stride S, the padding P, and can be calculated using the formula $\frac{(\mathbf{N}-\mathbf{K})+\mathbf{2P}}{\mathbf{S}} + \mathbf{1}$, where N is the size of the input. In the case of our example, the input is a square matrix of size 7x7, the kernel is 3x3, the padding is 1 and the stride is 1, so the output will be a matrix of size 5x5.

N.B. It is important to note that when we speaks of "two-dimensional convolutions", we are actually working with three-dimensional tensors (volumes). Thus, the example of the two-dimensional convolution given above does not fully reflect the complexity of convolutions in CNNs. In fact, when exploring various architectures, such as AlexNet or VGGNet, it is evident that the drawings of the various layers show the use of three-dimensional tensors. If you want to try and explore this further, here is a useful link that shows what these convolutions actually look like visually: **Animated AI Convolution**.

5.2. INSIDE A CNN

Here's an overview of what happens inside a Convolutional Neural Network:

> **Input image**: This layer represents the raw image entering the network. Generally, images are represented as multidimensional arrays of pixels, where each pixel has a value indicating its light intensity or color.

Convolution layer: This is where the convolution operation takes place, which is fundamental for extracting features from the input image. Here, filters (or kernels) are applied to the image to detect specific patterns. **Each filter**, as it flows over the entire image, **produces a feature map**, highlighting the presence of edges, shapes and textures. *It is as if each small region of the image participates in a "discussion" process to identify its unique features, and the different filters carry on this discussion, each highlighting different aspects.* The size of a filter is not constrained, allowing the network to adapt and learn different aspects of the images.

Non-linearity: After convolution, a non-linear activation function such as ReLU is applied. This layer adds non-linearity to the model, allowing the network to learn and represent complex relationships between image features, without affecting the receptive fields.

Spatial Pooling: This layer reduces the size of feature maps obtained by convolution, reducing the number of parameters and controlling overfitting. Commonly used are **max pooling** or **average pooling** layers, which reduce the size of the feature map by **preserving only the most significant values**. This also helps to create a **spatially invariant representation**, as relevant features are preserved even if slightly shifted in the image. *So, the network is able to classify a photo of a dog even if it is translated in the image.*

Normalization Layer: This layer is optional and can be used to normalise feature maps before passing them on to the next layer of the network. As we have seen, Normalization can be useful to ensure that **data remain in an appropriate range** and to facilitate the network learning process.

Feature Maps: The feature maps resulting from the previous steps represent the information extracted from the input image. Each feature map corresponds to a specific feature detected by the network and can be interpreted as a **simplified representation of the image in terms of relevant patterns and structures**. These feature maps are then used as input to subsequent layers of the network for further processing and analysis.

Nope, the colors are not related with the colors in the pictures above, sorry.

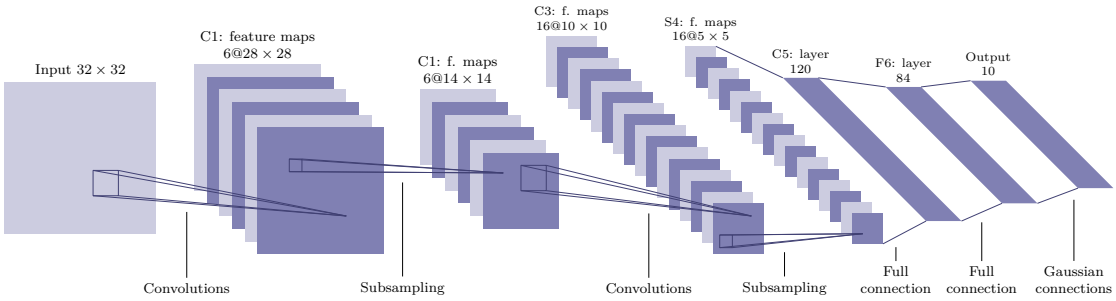
5.3. CNN ARCHITECTURES (FOR CLASSIFICATION)

Let's now discover the main architectures of Convolutional Neural Networks, from the pioneer LeNet to the modern ConvNeXt, which have revolutionised image recognition!

5.3.1. LeNet (1998)

LeNet was the **milestone that initiated the field of Convolutional Neural Networks**, developed at AT&T Labs. Its impact has been instrumental in promoting the use of Convolutional Neural Networks in character recognition and other computer vision applications. Its main features include:

- Uses **convolutional and subsampling** (pooling) followed by fully connected layers.
- **Requires complete retraining when image resolution changes**.
- Article: **"Gradient-Based Learning Applied to Document Recognition" (LeCun et al.)**

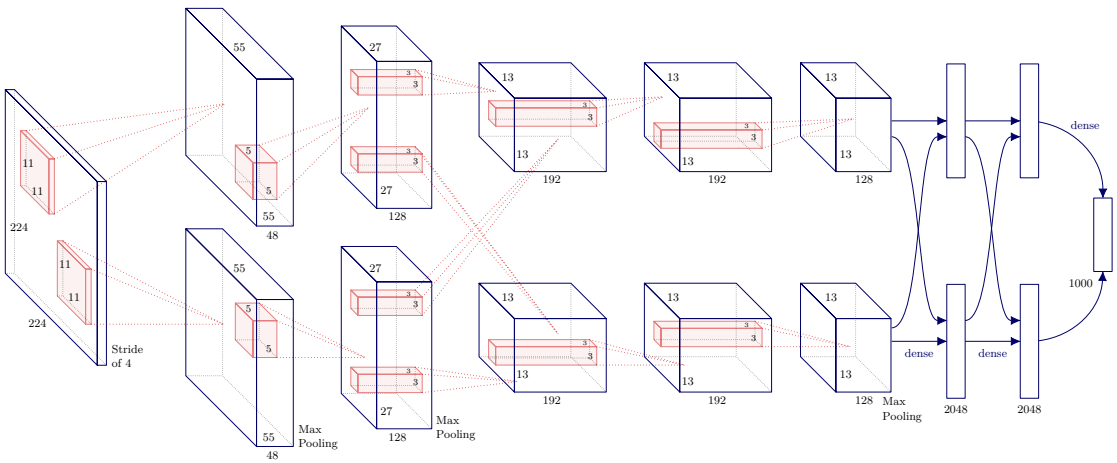


LeNet Architecture

5.3.2. AlexNet (2012)

AlexNet marked the **beginning of the deep learning revolution** and was the first modern CNN architecture. Its main features include:

- It has a similar structure to LeNet, but with a **larger model** (60M parameters) and **more data** (thanks to ImageNet).
- Because the resolution of the input is higher than LeNet (224x224), more hidden layers were needed (7 in this case).
- It was developed to participate in the **ImageNet Challenge competition** and outperformed all the previous models in the challenge.
- It was the **first GPU implementation** of a neural network (2 in parallel, that's why each CN layer is splitted into two halfs).
- Also, AlexNet used for the first time ReLu activation function instead of Tanh (because obtaininng the same performance using a Relu requires less time than using tanh).
- Article: **"ImageNet Classification with Deep Convolutional Neural Networks" (Krizhevsky et al.)**



AlexNet Architecture

Universal Feature Extractor: AlexNet showed that features learnt from convolutional layers (features before the output layer, 4096 dimensional features for each image) can be generalised and reused effectively for a wide range of computer vision tasks, making it a universal feature extractor (as they become quite discriminative). Also, a study showed

that features used in computer vision before AlexNet were way less discriminative even than features extracted after the first convolutional layer of the network. Finally, the features extracted from AlexNet are so powerful and discriminative that, when applied to another dataset (such as SUN-397) or to a different task from the original one, they produce **equally effective results without needing to be redefined or re-trained**.

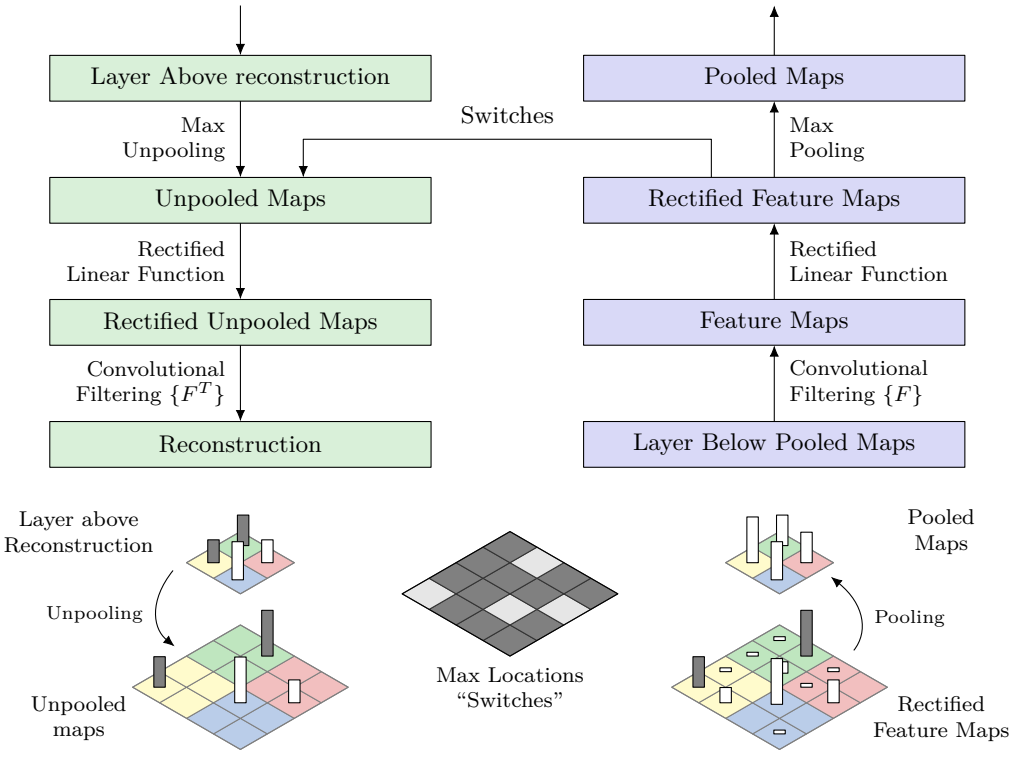
5.3.3. ILSVRC - Zeiler and Fergus Net (2013)

ILSVRC 2013 introduced similar architecture to AlexNet (5 convolutional layers and 2 fully connected layer at the end each with 4096 units), but with some significant innovations. Its main features include:

- Use of **smaller kernels in the first convolutional layers** (from 11x11 to 7x7).
- Use different number of channels. Why? Because their goal was to see visually what's happen inside a CNN.
- Introduction of the **explicability of CNNs through deconvolutional networks**.
- Article: **"Visualising and Understanding Convolutional Networks" (Zeiler and Fergus)**

In order to better understand what happens within a convolutional neural network and to identify the areas of interest on which it focuses to classify an image, we use **Deconvolutional Networks**. The work of Zeiler and Fergus is fundamental in this field and has made it possible to visualise activity in the intermediate layers of a CNN.

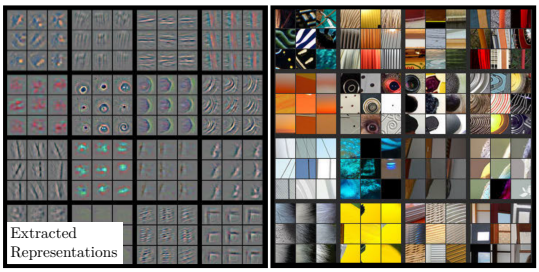
A Deconvolutional Network aims to interpret the activations in the various layers of a CNN by mapping them in pixel space. It works by reversing the convolution process: it is run in reverse and used as a probe without the need for learning. To do this, you **connect a Deconv Net to a layer of the CNN** you wish to probe, pass the input image through the CNN to obtain the activations, and then map these activations to pixel space.



Functioning of a Deconvolutional Network with Unpooling

The deconvolution process includes several steps: unpooling, rectification and filtering. Unpooling is performed using **switch variables to keep track of the positions of the maxima in the original pooling**, as max pooling is not invertible. Rectification uses the ReLU function to propagate only non-negative values, while filtering applies transposed versions of the learned filters.

The result is a visualisation that shows which features of the original image were relevant for classification, allowing us to understand which patterns the algorithm found discriminative for a given class and better understand the inner workings of the CNN to improve its performance and interpretability.



Patterns Captured in a Layer of a CNN

Another method to visualise whether the CNN is learning correctly is to **occlude an image at various positions** and observe the feature activations and classifier confidence.

Zeiler and Fergus' principle is based on developing a scheme to interpret the internal behaviour of neural networks, and occlusion experiments are a useful tool in this context.

There are also other ways of visualising the inner workings of a CNN, such as **Class Model Visualisation**, introduced in **"Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps" (Simonyan et al.)**, which is a more rigorous approach than the alternative. Essentially, a model can be viewed as a function that generates a vector containing the predictions for each class, so its size depends on the number of classes. One could consider reversing the optimisation process within a CNN: instead of adjusting the weights of the neural network through back-propagation, the goal is to find an **image that maximises the score for a specific label**.

However, two main challenges arise: different optimisers may produce different solutions and different optimisers may introduce distinct numerical optimisation problems. This final point is related to the fact that this approach requires the implementation of regularisation techniques.

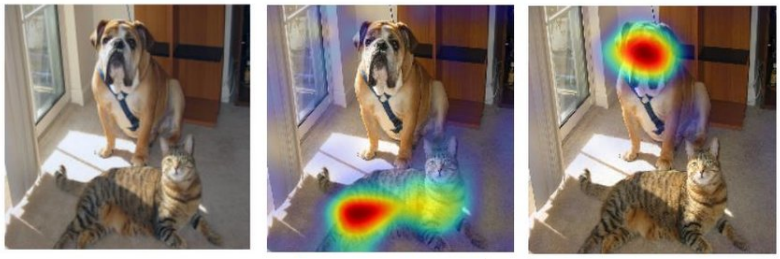
An important technique introduced later to better understand how CNNs work is **Grad-CAM** (Gradient-weighted Class Activation Mapping), which **evidences relevant** regions of an image for classification. It was introduced in the article **"Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization" (Selvaraju et al.)**.

The intuition is that gradients of the classification score with respect to the final convolutional feature map are used to identify the parts of an input image that most influence the classification score. The places where this gradient is large are those where the final score is most dependent on the data.

To calculate the Grad-CAM, we proceed as follows:

- The gradient of the score for the class c before softmax with respect to the feature maps of a convolutional level is calculated.

- A global pooling is performed on the obtained gradients.



Original Image | Grad-CAM "Cat" | Grad-CAM "Dog"

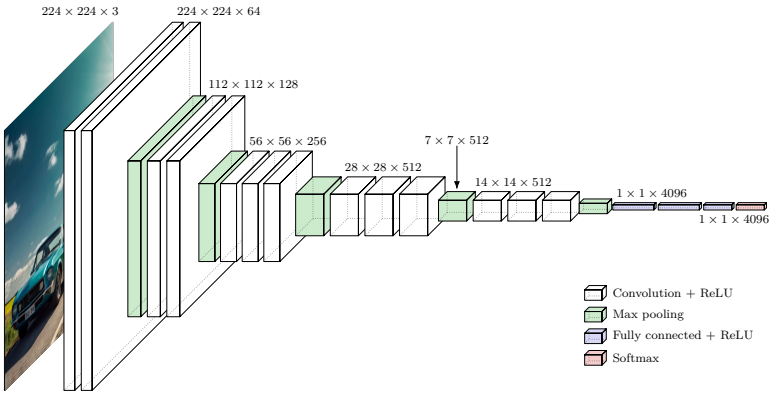
Grad-CAM applied to an Example Image

5.3.4. VGGNet (2014)

VGGNet represents an extension of the ideas introduced by AlexNet, with increased depth. It was the last NN developed with the AlexNet-like structure: alternation of convolutional layers and pooling layer until a small dimension is reached, and then attach a fully connected layer.

Its main features include:

- Very deep architecture with **more than twice the parameters of AlexNet**.
- Keeps the sequence of convolutions, subsamples and layers fully connected.
- One of the first architectures to demonstrate that **network depth is a critical factor for performance**.
- Article: **"Very Deep Convolutional Networks for Large-Scale Image Recognition" (Simonyan and Zisserman)**



VGGNet Architecture

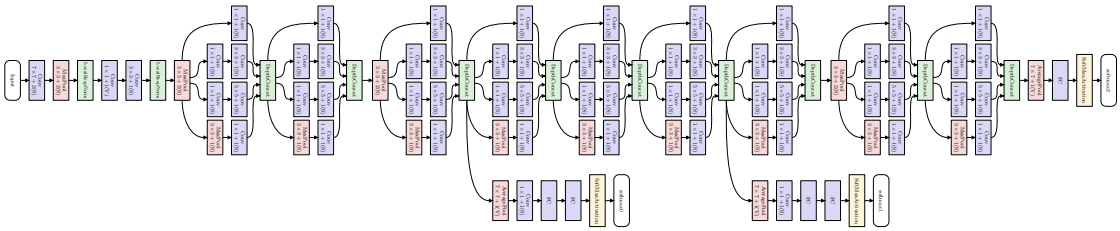
5.3.5. GoogleNet (2014)

The primary issue with VGG was the exponential increase in the number of parameters. Subsequently, neural network architectures underwent a dramatic shift towards significantly reducing the number of parameters. This reduction was prompted by the recognition that the alternating architecture employed by AlexNet had become impractical. Instead of expanding the parameter count, the focus shifted towards increasing the number of layers

(depth). A prominent example of this "new NN way" is GoogleNet, A 22-layer CNN but with a reduction in the number of parameters: from 60 million (AlexNet) to 4 million.

GoogleNet introduced an innovative structure, reducing the number of parameters compared to AlexNet. Its main features include:

- Does not use fully connected layers, **reducing the complexity of the model** (inception module) and the risk of overfitting.
- Usage of supervision both at the end and intermidiate layers in order to fight with vanishing gradient (a deeper network is more prone to).
- Usage of **inception modules**, allowing the network to capture details at different scales. An inception module consists of multiple convolutional layers with different kernel sizes, which are concatenated to capture features of different scales simultaneously. This parallelism helps GoogleNet capture fine-grained details as well as high-level features, making it exceptionally effective at image recognition tasks.
- Use of **1x1 convolutions to reduce the number of channels**, improving computational efficiency and reducing the computational workload.
- Introduction of **batch normalization**, which helps stabilise and speed up the training process. This is because if the inception block is trained with batch normalization, performance improves, even achieving the same performance as if not using batch normalization, but in fewer steps.
- Article: **"Going Deeper with Convolutions" (Szegedy et al.)**



GoogleNet Architecture

Furthermore, the same authors of this paper suggested some guidelines for designing deeper architectures:

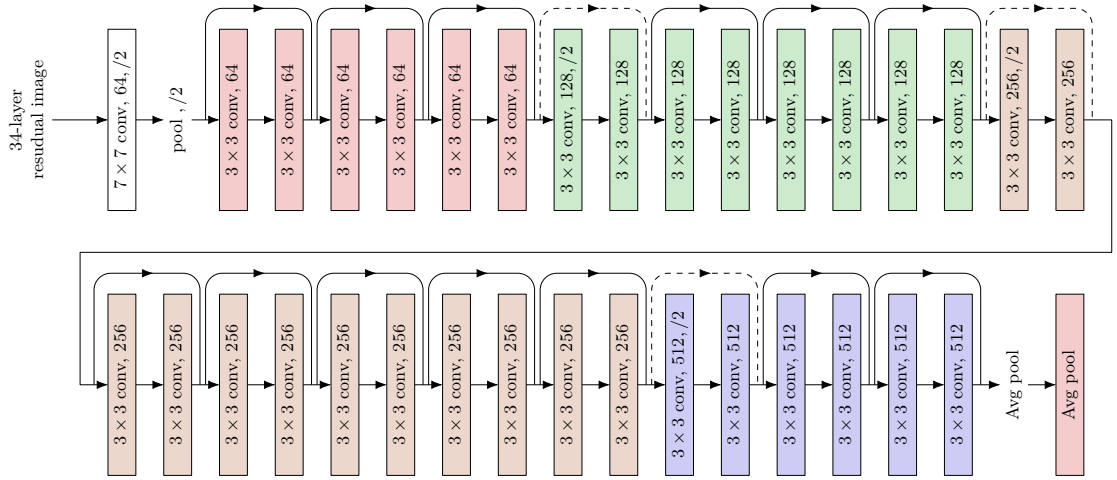
1. **Avoid extreme compression**: When designing a deeper CNN, it is important not to over-compress the information from the previous layer. It is preferable to gradually reduce the size of the representation rather than doing so drastically.
2. **Higher-dimensional representations are easier to process locally**: It means that within a network, it is easier to process representations that maintain a larger spatial dimension rather than compressing them too much.
3. **Reduce the input size before spatial aggregation**: Using 1x1 filters to reduce the input size before combining spatial features can be done without significantly losing the representation.
4. **Balancing the depth and width of Inception modules**: It is not yet completely clear how to best distribute the computation between height, width, and depth within an Inception module. However, it is advisable to look for an optimal balance to achieve good results.

5.3.6. ResNet (2015)

Generally, if we have a NN with 18 layers and another with 34 layer, we would think that the last should perform better. Unfortunately, it is not the case and performance

degradation is seen. ResNet addressed the problem of **performance degradation** of deep neural networks by introducing the idea of identity layers, also known as skip connection or identity bypass. Its main features include:

- **Identity Layers**: allow the model to learn to ignore certain layers when they are not needed.
- Using **many small kernels**, reducing the overall number of parameters and improving computational efficiency.
- ResNet can be seen as the **implicit ensemble of more superficial neural networks**, thanks to skip connections. In fact, if we unrolled a connection encompassing a residual blocks, we would have multiple paths through the network.
- Skip connections make the **loss landscape much more uniform** and easier to optimise during the training process.
- Article: **"Deep Residual Learning for Image Recognition" (He et al.)**



ResNet Architecture

After ResNet, CNN designs did not used anymore final fully connected layers if the model is deeper enough. Also, batch normalization became the only method used as regularization method for CNNs, allowing removal/decrease of dropout and L2 regularization.

There are two remarkable architectures inspired by ResNet:

DenseNet is an evolution of ResNet where each layer is directly connected to all other layers in the block. This promotes the reuse of features extracted from previous layers, improving gradient propagation and training stability. Advantages:

- **Feature reuse**: Each layer receives input from all previous layers in the dense block, enabling efficient reuse of information.
- **Parameter reduction**: As each layer uses existing features, the need to learn new features from scratch is reduced, improving the overall efficiency of the network.

SENet (Squeeze-and-Excitation Network) improves ResNet by introducing Squeeze-and-Excitation blocks that adaptively recalibrate features based on channel importance. This is done by calculating a weight for each channel using a sigmoidal activation function. Advantages:

- **Feature recalibration**: SE blocks compute weights for each channel, focusing on the most relevant ones, thus improving the network's ability to adapt to crucial details and patterns.

- **Performance improvement**: SENet has been shown to improve the accuracy of neural networks while reducing the number of parameters needed.

5.3.7. ConvNeXt (2022)

ConvNeXt represents a CNN architecture that has proven to be **competitive with modern transformers**. Its main features include:

- Designed to compete with modern transformer models.
- Use of **connection structures between layers** that enable more effective learning of spatial and semantic relationships.
- Article: **"A ConvNet for the 2020s" (Liu et al.)**

5.4. REGULARISATION TECHNIQUES FOR CNNS

Convolutional Neural Networks have demonstrated excellent learning capability on a wide range of computer vision tasks. However, like many other machine learning techniques, CNNs are **susceptible to the phenomenon of overfitting**, in which the model overfits the training data, reducing its ability to generalise to new data.

To mitigate overfitting and improve the performance of CNNs, several specific regularisation techniques have been developed. In this section, we will examine some of these popular techniques: Cutout, Dropblock, Cutmix.

Cutout is a regularisation technique that consists of **cutting a random rectangular area of the input image** (16x16) during the training process. This cut region is subsequently filled with the average pixel value of the entire dataset. *In practice, we force the model to consider more parts of the image during training, thus reducing the risk of overfitting.*

Dropblock is an extension of the dropout technique, in which instead of randomly deactivating certain neurons during training, **entire regions of neurons within a convolutional layer are deactivated**. This prevents the network from focusing too much on specific input features, promoting a more robust and general feature representation. Other variations are Dropout, drops individual values, and SpatialDropout, drops a whole channel.

Cutmix: During training, **a random rectangular area of an image is replaced with a corresponding area from another image**, while class labels are mixed in proportion to the replaced area. This technique encourages the model to learn useful features from multiple regions of the image, while providing a form of regularisation.

5.5. OBJECT DETECTION

Image classification is only one of the tasks that Convolutional Neural Networks can tackle. In addition to this, there are more complex tasks such as object detection, in which it is necessary not only to classify the objects in the image, but also to **identify their exact**

location through bounding boxes. In this section, we will explore the strategies used to deal with this challenge using *our beloved* CNNs.

Object detection is a complex task as it requires **both the classification of objects and the prediction of their positions through bounding boxes**. Thus, it implies doing simultaneously a classification and a regression task (for predicting continuous values, i.e. the bounding box). The input we provide is still an image, but the outputs are bounding boxes with labels for each object in the image.

This approach makes the training of networks more difficult than simple classification, as there are problems such as data sparsity and difficulty in accurately locating objects.

Before the advent of models such as AlexNet, the situation of object detection was at a standstill, with competition results stable on a plateau. However, the introduction of innovative techniques revolutionised this field.

5.5.1. Selective Search for Object Recognition (2013)

Selective Search is an image segmentation algorithm used to generate **proposed regions of interest** (RoI) for object recognition. The idea is to produce many "patches" in the image, and for each the algorithm assigns a likelihood score of that region containing the object of interest: regions with low likelihood are very unlikely to cointain it (and viceversa).

Its main features include:

- **Groups similar pixels together to form regions** and combines them into larger segments.
- **Preliminary Step**: identifies possible areas of the image that might contain objects before proceeding with classification.
- Using a hierarchical approach, the algorithm groups pixels according to specific criteria to form a "tree", which is used to obtain bounding boxes.
- Article: **"Selective Search for Object Recognition" (J.R.R. Uijlings et al.)**

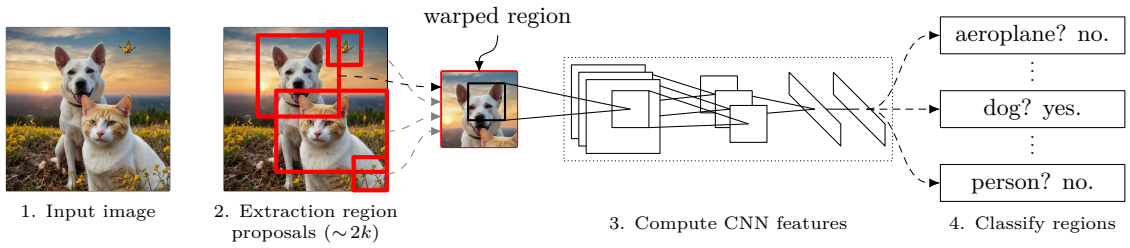
5.5.2. R-CNN - Region-based Convolutional Neural Network (2014)

R-CNN is the **first convolutional neural network-based approaches for object recognition**. Its main features include:

- *It's nothing but Selective Search + AlexNet.*
- Use **proposed regions** to identify objects in the image.
- Uses a convolutional neural network to **extract features and classify objects within these regions**: We begin by taking an image and employing selective search to extract a set of region proposals (∼2k/image). For each proposal, we fine-tune AlexNet individually, extracting features and training a classifier to determine whether the patch contains the target object.
- **Efficiency problem**: R-CNN uses a multi-stage model and Selective Search to explore the entire space of bounding boxes. Also, Selective Search is a "fixed" algorithm that does not dynamically adapt to the image, making it necessary to resize the input before passing it through the neural network, resulting in a **naive pipeline** approach. Lastly, both training and testing are very slow; for example, testing requires to rerun selective

search, about 2k forward pass per images. Despite this, R-CNN performed significantly better than shallow models.

- Article: **"Rich feature hierarchies for accurate object detection and semantic segmentation" (Ross Girshick et al.)**

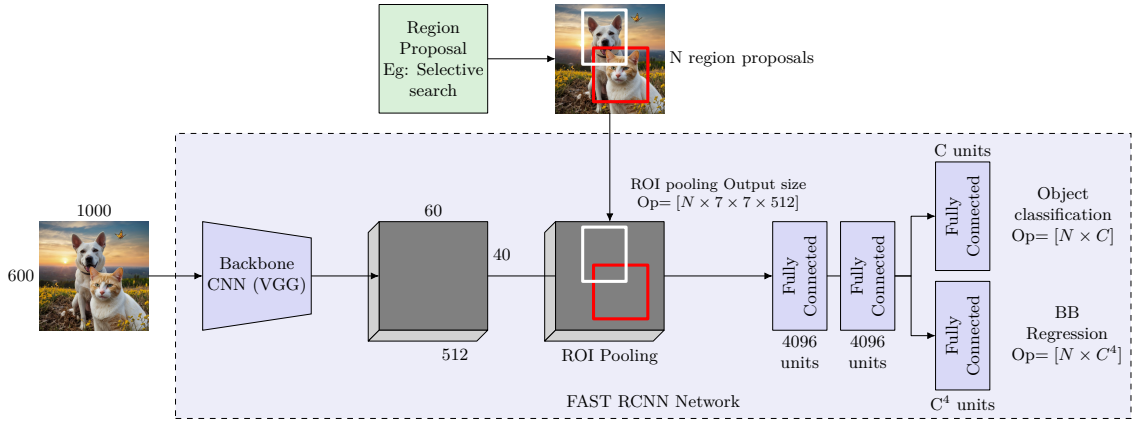


R-CNN Pipeline

5.5.3. Fast R-CNN (2015)

Fast R-CNN is an improved version of R-CNN that **addresses the efficiency issues** of its predecessor. Its main features include:

- **Region of Interest (RoI) pooling**: Introduces a RoI pooling layer that allows the best use of the information extracted from the convolutional network for each proposed region. This level makes it possible to **reduce the size of the proposed regions as input** to the final classifier without loss of significant information.
- **Convolution of the forward pass of a CNN**: Fast R-CNN shares the forward pass of a CNN on the image across its selected sub-regions. This means that **convolution is performed only once** for the entire image, reducing the computational load and improving the overall efficiency of the model.
- Article: **"Fast R-CNN" (Ross Girshick)**

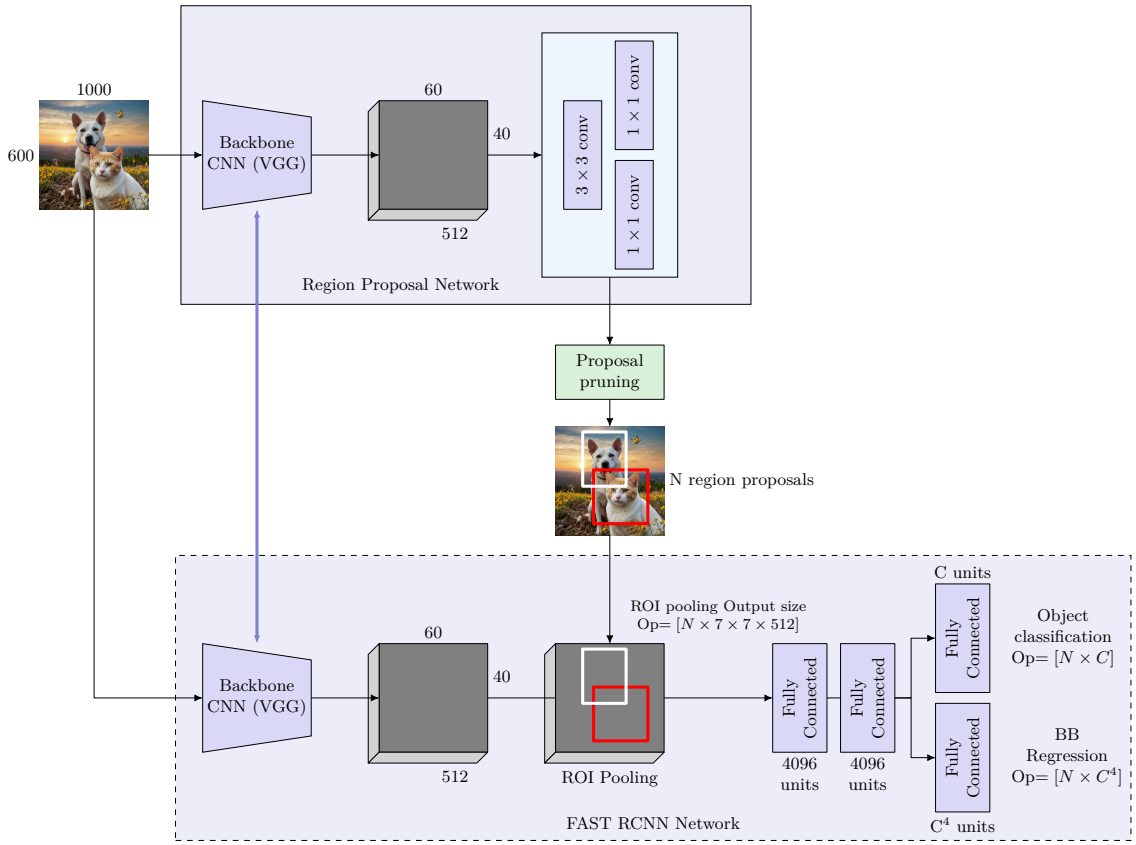


Fast R-CNN Architecture

5.5.4. Faster R-CNN (2015)

Faster R-CNN is another significant development in the field of object detection. Differently to R-CNN and Fast R-CNN which use selective search to find out the region proposals, Faster R-CNN eliminates it as it is a slow and time-consuming process. Instead, this model lets another the network to learn the region proposals. Its main features include:

- **Similarity with Fast R-CNN**: Like Fast R-CNN, the image is provided as input to a CNN that provides a convolutional feature map.
- **Use of a separate network for proposal generation**: Instead of using Selective Search, a separate network called RPN (Region Proposal Network) is used in Faster R-CNN to predict region proposals. The predicted region proposals are then reshaped using a RoI pooling layer which is then used to classify the image within the proposed region and predict the offset values for the bounding boxes.
- **End-to-end architecture**: With the integration of the RPN module, Faster R-CNN allows the entire end-to-end model to be trained in a single step, improving efficiency and ease of training.
- Article: **"Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks" (Shaoqing Ren et al.)**



Faster R-CNN Architecture

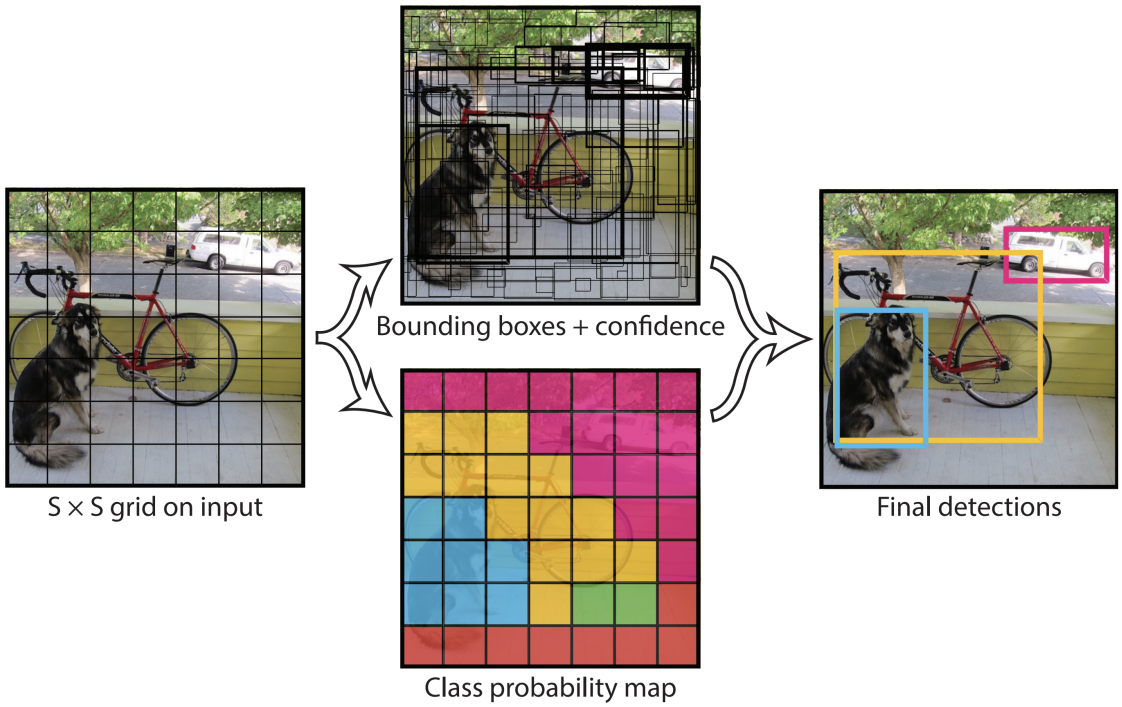
5.5.5. YOLO - You Only Look Once (2016)

YOLO is an innovative approach to object detection that **stands out for its speed and efficiency**. Its main features include:

- YOLO treats the problem of object detection as a **direct regression problem**, in which the entire image is divided into a grid and for each grid cell the bounding box and class probability of the objects within that cell are directly predicted.
- **Use of a single convolutional network**: Unlike previous methods that use regions to locate objects in the image, YOLO uses a single convolutional network to predict bounding boxes and class probabilities for these boxes. This makes YOLO's approach simpler and more efficient.
- **Split image into grid**: YOLO splits the image into an SxS grid and for each grid

cell predicts a set of bounding boxes. This approach allows YOLO to consider several regions of the image simultaneously.

- **Bounding box selection**: For each predicted bounding box, the YOLO grid produces a class probability and offset values for the bounding box. Bounding boxes with a class probability above a threshold are selected and used to locate the object in the image.
- Article: **"You Only Look Once: Unified, Real-Time Object Detection" (Joseph Redmon et al.)**



YOLO Architecture

5.6. INSTANCE SEGMENTATION

In addition to object classification and bounding box detection, there is another challenge in computer vision: instance segmentation. It consists of **assigning a label to each pixel of an image**, distinguishing individual objects within the scene. This is an **even more complex challenge** than simple object detection, as it requires a detailed understanding of the structure and shape of the objects themselves.

One of the most advanced techniques to address this challenge is known as Mask R-CNN, which extends the Faster R-CNN framework for instance segmentation.

5.6.1. Mask R-CNN (2017)

Mask R-CNN is an extension of Faster R-CNN that adds an **additional branch to the neural network for instance segmentation**. Its main features include:

- Mask R-CNN uses **RoI Align**, an improved version of RoI Pooling, to ensure more precise alignment of feature maps during the instance segmentation phase. This significantly improves the quality of segmentation masks.

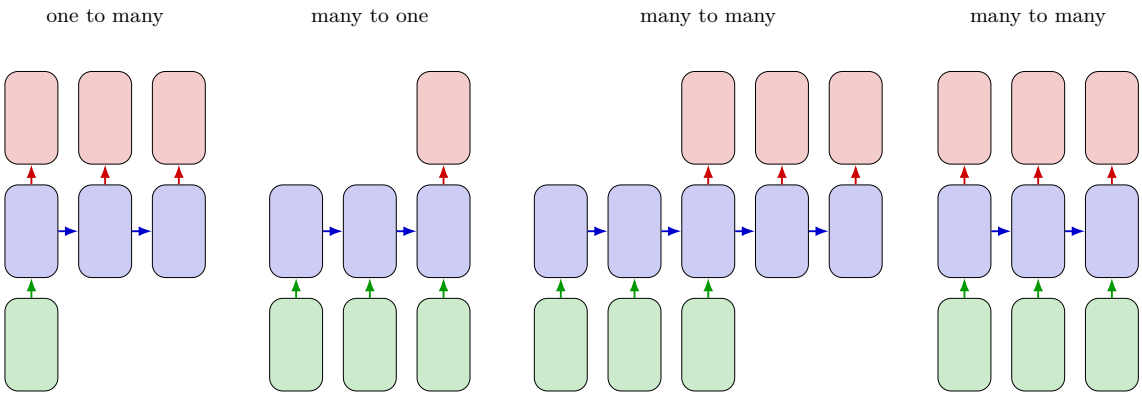
- **Pixel-level segmentation**: Mask R-CNN extends Faster R-CNN for pixel-level segmentation, allowing a label to be assigned to each pixel in the image based on the object instance to which it belongs.
- **FCN added**: Mask R-CNN adds a Fully Convolutional Network on top of selected features to produce accurate and detailed segmentation masks.
- Article: **"Mask R-CNN" (Kaiming He et al.)**

6. RECURRENT NEURAL NETWORKS

6.1. HANDLING VARIABLE-LENGTH SEQUENCES

So far we have mainly focused on models that handle **input and output of fixed size**. For example, in a convolutional neural network (CNN), we transform an input in the form of a matrix (image) into an output in the form of a vector (labels). However, the community has developed models that can handle variable-length structures, more specifically sequences, such as text, which is a sequence of strings.

Recurrent Neural Networks (RNNs) are flexible tools that can perform many different tasks, which can be classified as follows:



Diagrams of RNN Models

6.1.1. Multiple2Single: Sentiment Analysis

In the sentiment analysis, the model must classify a product, restaurant, or any other item as positive, neutral, or negative based on the reviews received. In this scenario, we provide the neural network with inputs of varying sizes. Each word is transformed into a **hidden representation that reflects the context and propagates through subsequent words**, as these words are not independent of each other. This propagation occurs unidirectionally, similar to a feedforward neural network. However, the main difference is that instead of mapping a single input to an output of fixed size, the model takes multiple inputs and produces a single label. Predictions are typically made about the last hidden state, which should include the context of previous words.

6.1.2. Single2Multiple: Image Captioning

In image captioning, the network is fed a fixed input (an image), but the output will have a variable length, since we want to produce a sentence describing the content of the image. In this case, the latent representation of the image obtained from a CNN is **combined with the latent representations of the words to model the dependencies between them**. This approach allows a single input to be mapped to multiple outputs, generating several words that describe the image.

6.1.3. Multiple2Multiple: Machine Translation

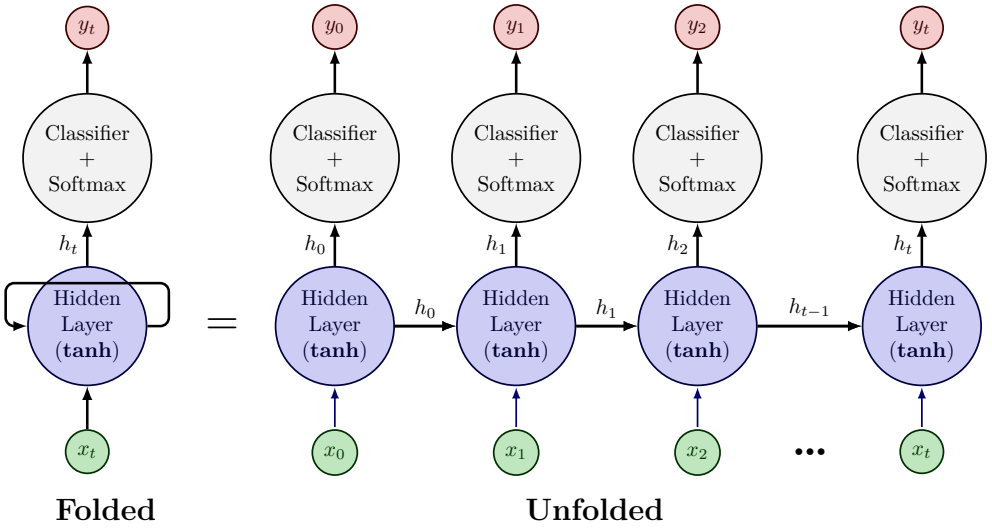
In machine translation, the task is to translate a sentence from one language to another. This problem involves both multiple inputs and multiple outputs. Each word in the input sentence is mapped to a hidden representation that takes into account the context of the preceding words. The neural network uses these representations to generate the translated sentence, word by word, **maintaining sequential dependencies between words**. This allows the model to produce an accurate translation that reflects the meaning of the original sentence.

6.2. VANILLA RNN

Recurrent neural networks are, in essence, **neural networks with cycles**.

The Vanilla RNN, introduced in **"Finding structure in time" (Jeffrey L. Elman)**, has a **recurrent design** in that it performs the same parametric function for each input (x_t, each marked with a timestep), while the output (h_t, the hidden representation marked with the corresponding timestep) depends on the previous computation (h_{t-1}). After producing the output, this is **used to generate the hidden representation of the next RNN cell** (h_{t+1}). To make a decision (y_t), a linear layer can be applied to the end of a cell, considering both the current input and the output learned from the previous input.

There are two types of representations for an RNN: the **Folded** representation, which is synthetic, and the **Unfolded** representation, which provides a more explicit description of the computations. Below is the architecture of the network in both representations:



RNN Architecture

The structure of an RNN cell, as shown in the figure above, has two inputs (x_t and h_{t-1}) and can be represented mathematically as follows:

$$h_t = f_W(x_t, h_{t-1}) = \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} = \tanh(W_x x_t + W_h h_{t-1})$$

In this equation, h_t represents the hidden state at time t, x_t is the input at time t, h_{t-1} is the previous hidden state, W_x and W_h are the weights associated with the current input and previous hidden state respectively. The function tanh provides nonlinearity to the network (*Using hyperbolic tangent is understandable given that the paper is from the 1990s, a time when the use of it was common*). During training, W weights are learned through the optimization process. For the prediction:

$$y_t = \text{softmax}(W_y h_t + b_y)$$

A peculiar aspect of RNNs is the **sharing of weights among different timesteps**: the weights W_x, W_h and W_y are shared between different timestep iterations, allowing the model to capture **sequential dependencies within the data**.

So, the main differences between an RNN and an FNN include the sharing of weights between timesteps, the use of backpropagation variation for training (in which we need to sum the contributions along the entire sequence) and the presence of three different sets of weights: W_x, W_h and W_y. In addition, **the problem of gradient vanishing is more severe** in RNNs than in feedforward networks, mainly because of the hyperbolic tangent function used as the activation function. Also, backpropagation through time can be computationally expensive for a large number of timesteps.

6.2.1. Truncated BPTT (Backpropagation Through Time)

To solve the problems described above, we use the Truncated BPTT algorithm, which is a more efficient technique when dealing with long sequences. It was introduced as a Ph.D. thesis in **"Training Recurrent Neural Networks" (Ilya Sutskever)**.

The idea of Truncated BPTT is to simplify this process. Instead of running the entire sequence through the network, we divide it into **smaller blocks of fixed length**. Each block is treated as a separate training unit. In this way, the network does not have to work with the whole sequence at once, but only with small chunks at a time, making training more efficient.

After presenting a block to the network, **backpropagation is performed only for a limited number of backward time steps**, rather than traversing the entire length of the sequence. This means that the network does not have to keep track of information too far back in time, reducing the computational cost of training.

To control how far back in time to go during backpropagation, we use two parameters, k_1 and k_2. The first, k_1, determines the **length of the blocks** into which we divide the sequence, while k_2 defines the **number of time steps** over which to perform backpropagation within each block. The Truncated BPTT algorithm can be schematized as follows:

Algorithm

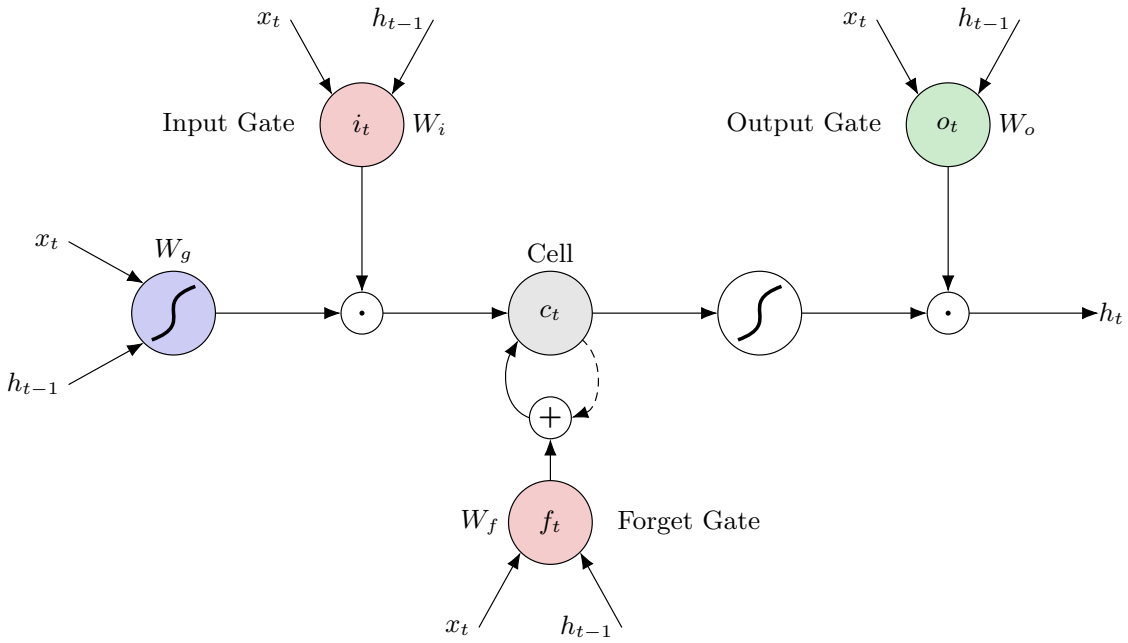
1: Present a sequence of k_1 timesteps of input and output pairs to the network.
2: Unroll the network then calculate and accumulate errors across k_2 timesteps.
3: Roll-up the network and update weights.
4: Repeat the process.

Adjusting k_1 and k_2 allows us to balance the training speed and the network's ability to capture long-range time dependencies.

6.3. LONG SHORT-TERM MEMORY (LSTM)

It has been recognized that the classical RNN suffers from an inefficient design in its individual cell, as efforts to improve learning have not produced significant improvements in the vanishing gradient problem due to inherent limitations in the cell design. To solve this problem, LSTM was introduced. The architecture was first presented in this article: **"Long Short-term Memory" (Sepp Hochreiter)**.

The LSTM cell not only produces the hidden state h_t, but adds another component called **Memory Cell** (c_t), which is responsible for maintaining and deleting information, based on the input context. This means that **some of the previous information must be remembered, some must be forgotten, and some new information must be added to memory**. Here is the design of the cell:



LSTM Cell Structure

As you can see, within an LSTM cell, there are three different parametric components, called **Gate**. Let's explore them in detail.

Input Gate: can allow the input signal (x_t and h_{t-1}) to alter the state of the memory cell or block it.
Its output i_t is computed by the sigmoid function applied to the input x_t and the preceding dependency h_{t-1}, and is associated with its own parameters W_i:

$$i_t = \sigma(W_i \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_i)$$

i_t is used as a trade-off between the information passing through the current input (x_t and h_{t-1}) and the information encoded by the previous cell (c_{t-1}).

Forget Gate: can modulate the self-recurrent connection of the memory cell, allowing the cell to remember or forget its previous state as needed.
Its output f_t is computed by the sigmoid function applied to the input x_t and the previous dependency h_{t-1}, and is associated with its own parameters W_f:

$$f_t = \sigma(W_f \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_f)$$

Memory Cell Calculation is a combination of the previous two gates:

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

We can interpret the equation as follows:

- $f_t \odot c_{t-1}$: The forget gate decides how much of the cell's previous state c_{t-1} should be forgotten or retained for the next state c_t.
- $i_t \odot g_t$: This determines how much of the new information g_t, i.e., the value of the input modulation, should be added to the state of the cell c_t.

In short, **the forget gate modulates how much of the previous state to retain**, while **the input gate modulates how much of the new information is to be added to the cell state**.

Output Gate: can allow the state of the memory cell (c_t) to have an effect on other neurons, thus affecting the output of the LSTM unit (h_t).
First, a sigmoid layer decides which parts of the cell state (c_t) the model is going to produce as output (associated with its parameters W_o):

$$o_t = \sigma(W_o \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_o)$$

To **Calculate the Output** of the cell, a tanh layer is used on the state of the memory cell to shrink values between -1 and 1, which are then multiplied by the output of the output gate:

$$h_t = o_t \odot \tanh(c_t)$$

It is important to note that although there are many nonlinearity functions, the calculations to update the value of the cell (c_t) are **linear**, so the gradient flow from c_t to c_{t-1} involves only **backpropagation through addition and element-by-element multiplication**, not multiplication of matrices or nonlinear functions.

We can also formulate a (*beautiful*) compact version of everything we have seen, summarizing the formulas in:

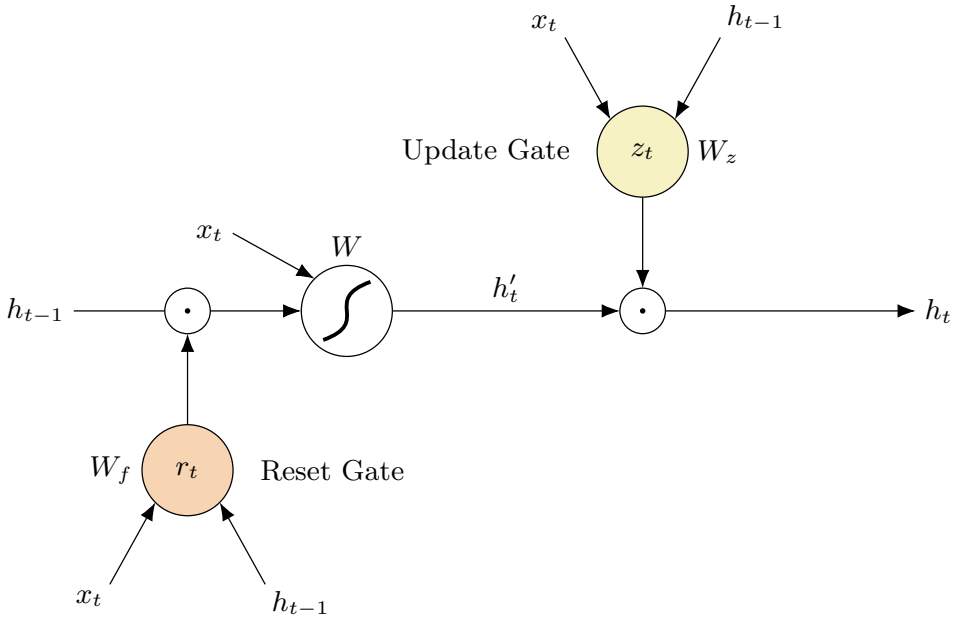
$$\begin{pmatrix} g_t \\ i_t \\ f_t \\ o_t \end{pmatrix} = \begin{pmatrix} \tanh \\ \sigma \\ \sigma \\ \sigma \end{pmatrix} \begin{pmatrix} W_g \\ W_i \\ W_f \\ W_o \end{pmatrix} \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

$$h_t = o_t \odot \tanh(c_t)$$

6.4. GATED RECURRENT UNIT (GRU)

As the community began to apply LSTM models to various tasks such as captioning, it became apparent that the network was overly complex. Thus, thanks to **"Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation" (Cho et al.)**, the GRU network was introduced to address this complexity by simplifying the LSTM cell by **removing one of its gates**. Despite having only two gates, the model is sufficiently resilient to the vanishing gradient problem. Moreover, the information flow is stored only in the hidden state h_t. Here is the architecture of a cell:



GRU Cell Structure

As before, let's analyze together the gates used in the GRU model.

Reset Gate: used to decide how much of the passed information to forget.
The output r_t is computed by the sigmoid function applied to the input x_t and the previous hidden state h_{t-1}, and is associated with its own parameters W_r:

$$r_t = \sigma(W_r \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_t)$$

An important change in the design of the LSTM cell is that **the value of the input modulation is no longer based solely on the complete information from the previous hidden state** h_{t-1}, but instead on a "restricted stream" determined by r_t:

$$h'_t = \text{tahn } W(\begin{pmatrix} x_t \\ r_t \odot h_{t-1} \end{pmatrix})$$

Update Gate: helps the model determine how much of the past information (from previous time steps) needs to be transmitted to the future.
The output z_t is computed by the sigmoid function applied to the input x_t and the previous hidden state h_{t-1}, and is associated with its own parameters W_z:

$$z_t = \sigma(W_z \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_z)$$

The **Final Representation of the Hidden State** is calculated by an additional point product:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot h'_t$$

7. SEQUENCE MODELS

As the scientific community began to address problems involving sequentially structured data, RNNs returned to the forefront, leading to the development of new designs such as GRUs. However, it became apparent that RNNs had difficulty handling very long sequences. To address this problem, the Attention model was introduced as an auxiliary module, significantly improving performance in complex tasks such as image captioning and neural machine translation.

The concept of Attention model initially found application in models that combined CNN and RNN, and then was incorporated into Transformers, the model underlying tools such as ChatGPT. This led to the emergence of an inevitable question, "*Do we really need RNN and Attention models together for sequencing tasks, or can one Attention model suffice on its own?*" This question prompted the exploration of sequence models without the use of RNNs, *but we'll see this in the next chapter.*

In this chapter, instead, we will explore how this change came about through innovations implemented in the **Image Captioning** and **Machine Translation** tasks.

7.1. CAPTIONING MODELS

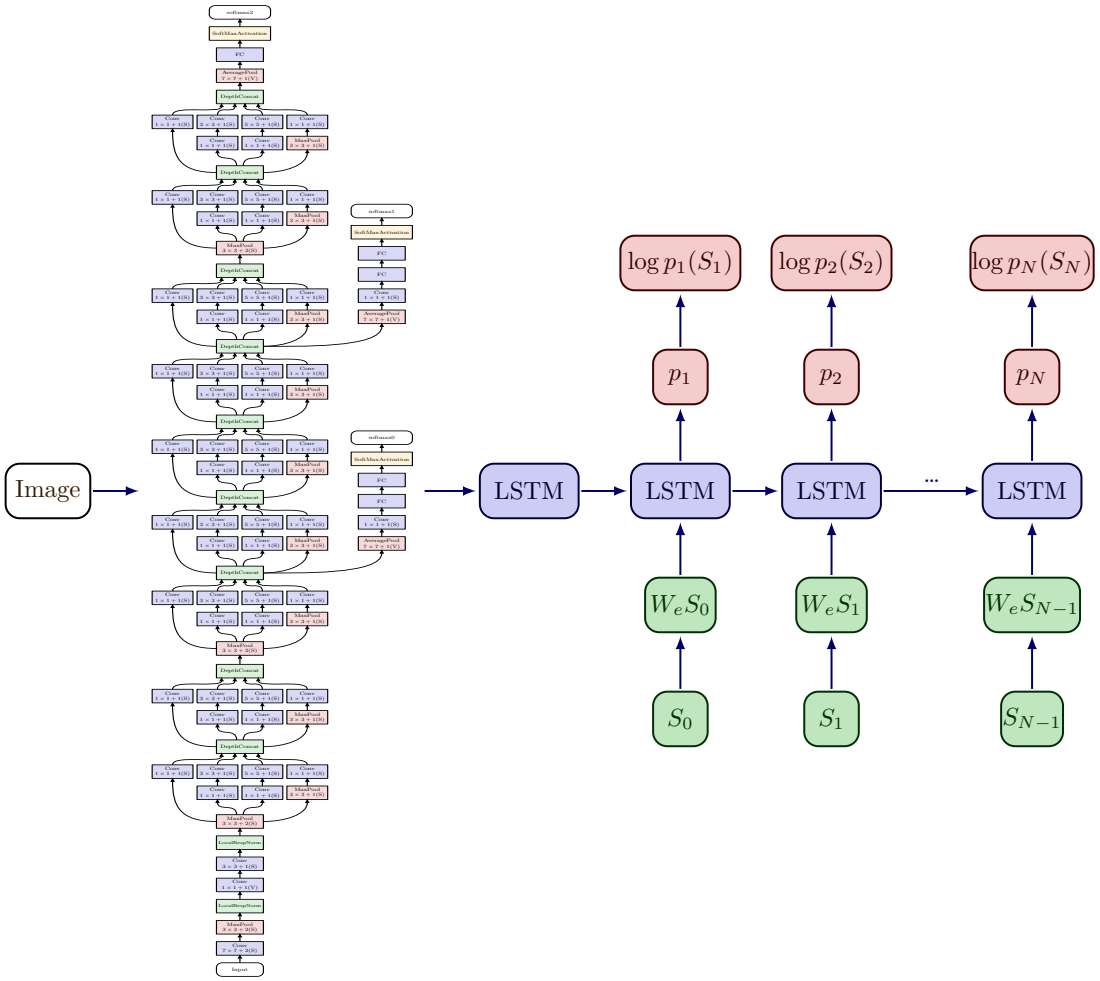
Image captioning is the process of generating a textual description for given images. It can be seen as a Sequence to Sequence problem from beginning to end, where images, considered as sequences of pixels, are transformed into sequences of words. This task requires processing both textual and image statements. For the textual part, we use recurrent neural networks (RNNs) and for the visual part, we use convolutional neural networks (CNNs) to obtain feature vectors.

7.1.1. Show and Tell

Specifically, the architecture developed by Google in 2014 in the paper **"Show and Tell: A Neural Image Caption Generator" (Bengio et al.)** uses both GoogleNet and LSTM.

The procedure begins with feature extraction of the image using GoogleNet, which produces a feature vector, denoted as I. This vector is then inserted into the LSTM sequence at time $t = -1$, only once. Next, starting at $t = 0$, a sequence of word vectors is inserted. Each word is represented as a one-hot vector S_t with dimensions equal to the size of the vocabulary.

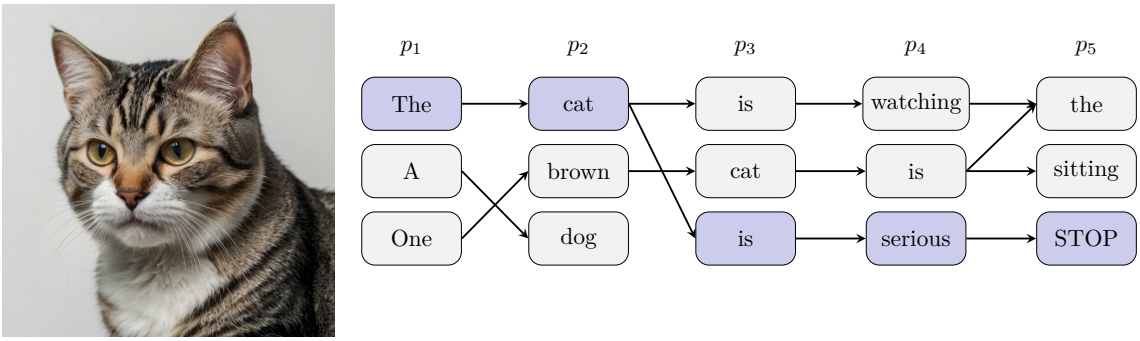
At each time step, the word vector S_t is transformed through an embedding layer $W_e S_t$ and provided to the LSTM. Then, the input is processed along with the previous hidden state to produce a new hidden state and calculate the probability p_t of the next word in the sequence. The word with the highest probability, denoted as $\log p_t(S_{t+1})$, is chosen as the output for that time step. This process is repeated until a special end-of-sentence word (S_N) or a maximum length of the sequence is reached. Check the image below for a visual representation of the process:



Show and Tell Caption Model

The quality of the generated caption depends strongly on the performance of the CNN, making fine-tuning crucial. So, the architecture combines CNN and LSTM in a way that leverages the visual feature extraction capabilities of CNN and the sequential modeling capabilities of LSTM to generate accurate and consistent captions for images.

For inference time, the model uses one of the following methods: **Sampling** and **Beam Search**.

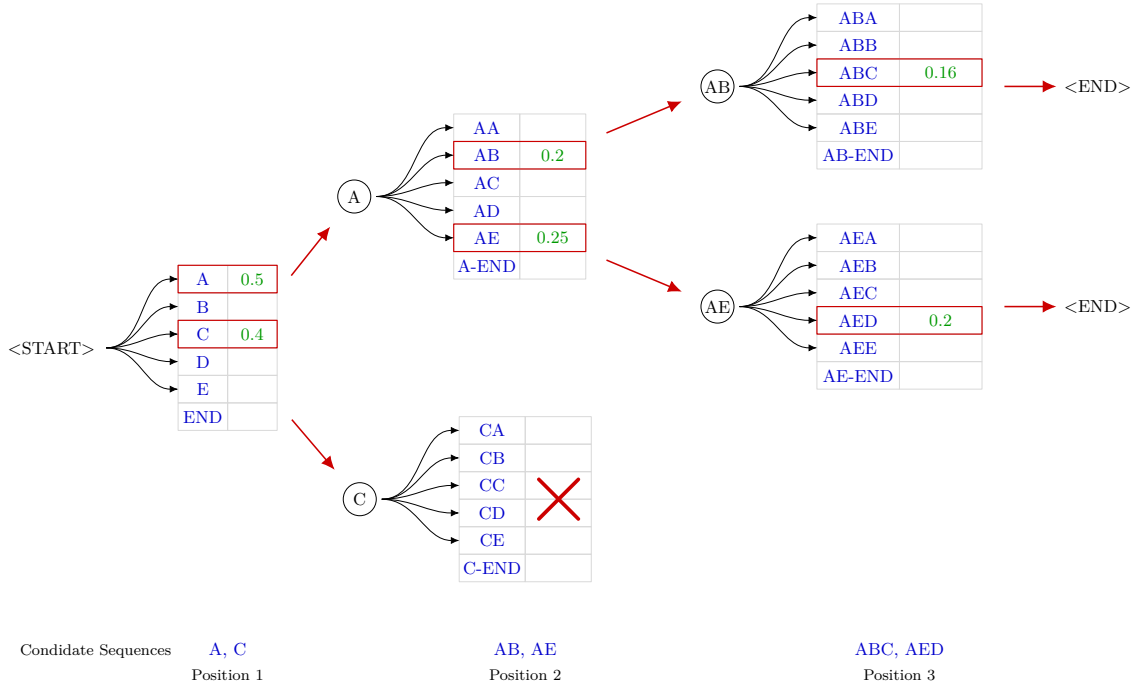


Sampling Visual Example

In the sampling method, the first word is extracted according to a probability p_1, then the corresponding embedding is provided as input and the next word is extracted with probability p_2 (which depends on the probability that a certain word has been predicted

previously). This process continues until the special end-of-sentence token is extracted or a maximum length is reached.

With Beam Search, on the other hand, we iteratively consider the k **best sentences up to time** t as candidates to generate sentences of length $t + 1$, keeping only the k best of them. This approach allows more word combinations to be explored, improving the quality of the generated sentences compared to simple sampling.



Beam Search Visual Example

In the image above you can see an example of Beam Search with a width of 2, using characters for simplicity. Here's the description of what is happening:

- **First Position**: The model starts with the token "<START>" and calculates the probabilities for each possible word. It selects the two characters with the highest probability, for example "A" and "C".
- **Second position**: The model generates the probabilities for the second position twice, holding the first position ("A" or "C") constant. It then chooses the two best pairs of characters based on the combined probability, such as "AB" and "AE."
- **Third position**: This process is repeated. The model calculates the probabilities for the third position, limiting the first two positions to "AB" or "AE," and selects the best three combinations of characters based on the combined probability of the first three characters. This continues until a token "<END>" is selected, marking the end of that branch of the sequence. Simple, isn't it? 😄

In order to test the performance of the model, we face the challenge of comparing the output of the model with the **captions of human annotators**. Of course, the annotations of different individuals may vary, presenting inherent discrepancies. To address this problem, the Bilingual Evaluation Understudy (BLEU) score was used.

7.1.2. BLEU Score

The BLEU score is a metric used to evaluate the quality of texts generated in natural language processing (NLP) models. The text generated by the model is called **Candidate**, while the possible correct texts are called **References**. The BLEU score compares the candidate with the references to measure their similarity.

The BLEU score ranges from 0 to 1 and is based on accuracy, i.e., the **proportion of matching n-grams with references** (sequences of n words) found in the candidate compared **to the total number of n-grams in the candidate**. Usually, n-grams of size 1 to 4 are considered.

To prevent the candidate from scoring well by repeating the same n-gram many times, we **limit the number of occurrences** of each n-gram in the references. For example, if a reference contains the word "the" twice, the candidate should also not have more than two occurrences of "the".

Finally, the BLEU score is multiplied by a **brevity penalty** to prevent translations that are too short from getting high scores.

Example:

The Reference is: "*Transformers: Transformers make everything quick and efficient.*"

The Candidate is: "*Transformers Transformers Transformers Transformers.*" 

The BLEU score is low because the candidate repeats "Transformers" without providing full context, while the reference has a meaningful sentence.

7.2. CAPTIONING MODELS (WITH ATTENTION)

7.2.1. Show, Attend and Tell

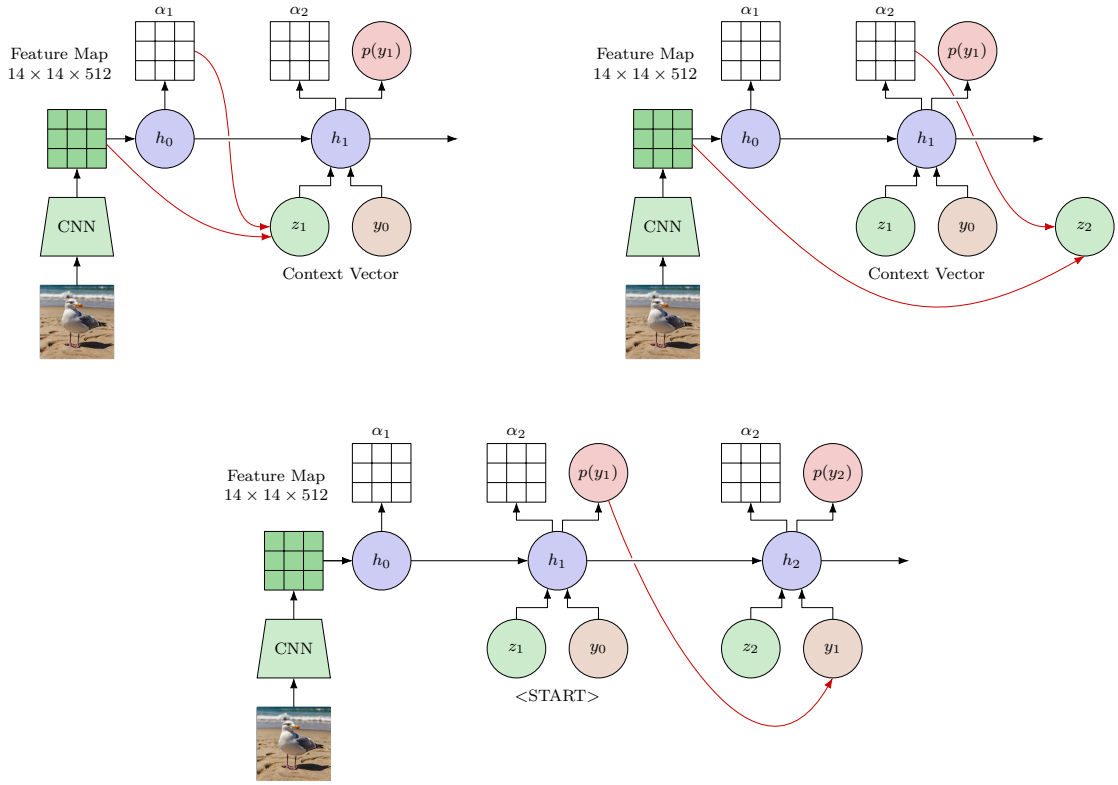
Captioning with attention represents a significant innovation in the field, especially for image and text processing. This approach, which was introduced in the paper **"Show, Attend and Tell: Neural Image Caption Generation with Visual Attention" (Xu et al.)** in 2015, has overcome some limitations of previous models, paving the way for more advanced results in image processing and caption generation.

The key idea of this approach is the integration of two key components: a convolutional neural network (CNN) for image feature extraction and a recurrent neural network (RNN) with an attention mechanism to generate captions. This combination allows the model to **focus attention on specific parts of the image while generating the corresponding description**.

The main innovations introduced by this approach are as follows:

- **Feature Extractor**: A CNN is used to extract a set of feature vectors, each of which represents a part of the image. So, here we maintain the spatial information.
- **RNN with Attention**: The extracted feature maps are not only used for initializing the RNN, but are passed to each cell of the RNN during caption generation. This approach allows the network to focus attention on specific parts of the image as it generates each

caption word. **Each RNN cell produces both the probability of the predicted word ($p(y_t)$) and an attention map** (α_t) indicating the relevant parts of the image.



Flow of Show, Attend and Tell

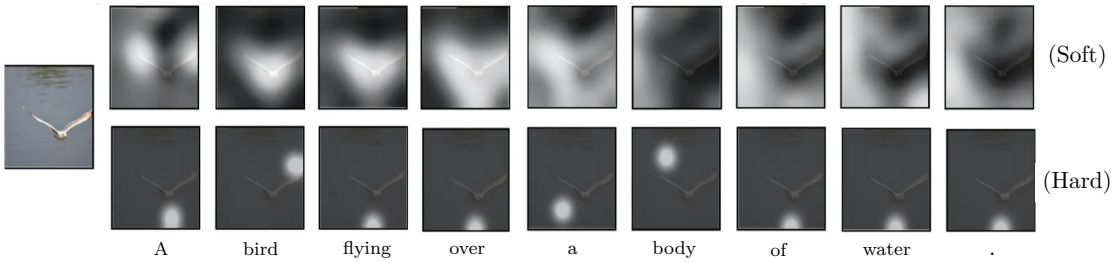
In the image above, each LSTM cell receives **three main inputs**: the previous latent space h_{t-1}, the predicted word in the current sequence y_{t-1} and a context vector z_t (obtained from the dot product between the feature map produced by the CNN and the previous attention map α_{t-1}).

7.2.2. Types of Attention

There are two main types of attention mechanisms: **soft** and **hard**.

In soft attention, the context vector z_t is computed as a weighted combination of image features, where the weights $\alpha_{i,t}$ represent the weighting coefficients: $z_t = \sum_i \alpha_{i,t} a_i$. During the process, different words become associated with different regions of the image, each represented by a different feature map (see the figure below). This mechanism allows the neural network to **dynamically focus on specific parts of the image as it generates a sequence of words**, using a weighted combination that emphasizes the regions most relevant to the current context.

On the other hand, in hard attention, the attention map gives score 1 to the highest weight among all and zero to the remaining. This indicates a specific location in the image. The context vector z_t is then obtained by selecting the corresponding cell in the image feature map. Although hard attention can theoretically produce better performance than soft attention, the training is considerably more complex. Moreover, the difference in performance from soft attention is often not significant. As a result, **hard attention is often neglected**, while soft attention can be trained using optimization algorithms such as stochastic gradient descent (SGD).



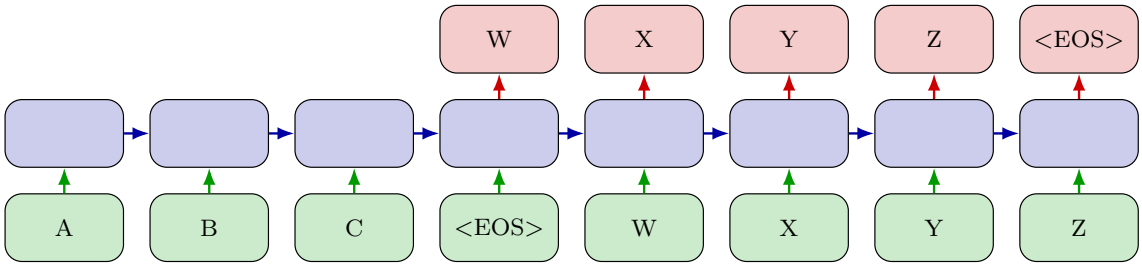
Soft and Hard Attention

7.3. SEQ2SEQ

While significant breakthroughs initially emerged in image captioning, the most notable advancements took place in machine translation. Specifically, in 2016, Google introduced the first Neural Machine Translation model to enhance Google Translate. Google achieved this not solely due to their resources but also thanks to influential academic works that preceded their efforts. One such influential work is **"Sequence to Sequence Learning with Neural Networks" (Sutskever et al.)**, which shows the promise of Neural Machine Translation (NMT) over Statistical Machine Translation (SMT).

The distinguishing factor of this paper was the implementation of a machine translator in an end-to-end way. **End-to-end learning** means that the machine translator handles the entire translation process, from inputting the original text to outputting the translated text, without the need for separate components or intermediate steps. In order to do this, the model introduces for the first time the **encoder-decoder** approach to handle input sequences of variable lengths.

1. **Encoder**: LSTM that is responsible for mapping the variable length input phrase into a fixed-dimensional vector (**context vector**). The latter serves as a connection between the encoder and the decoder.
2. **Decoder**: LSTM that maps the fixed vector into a variable length output sequence

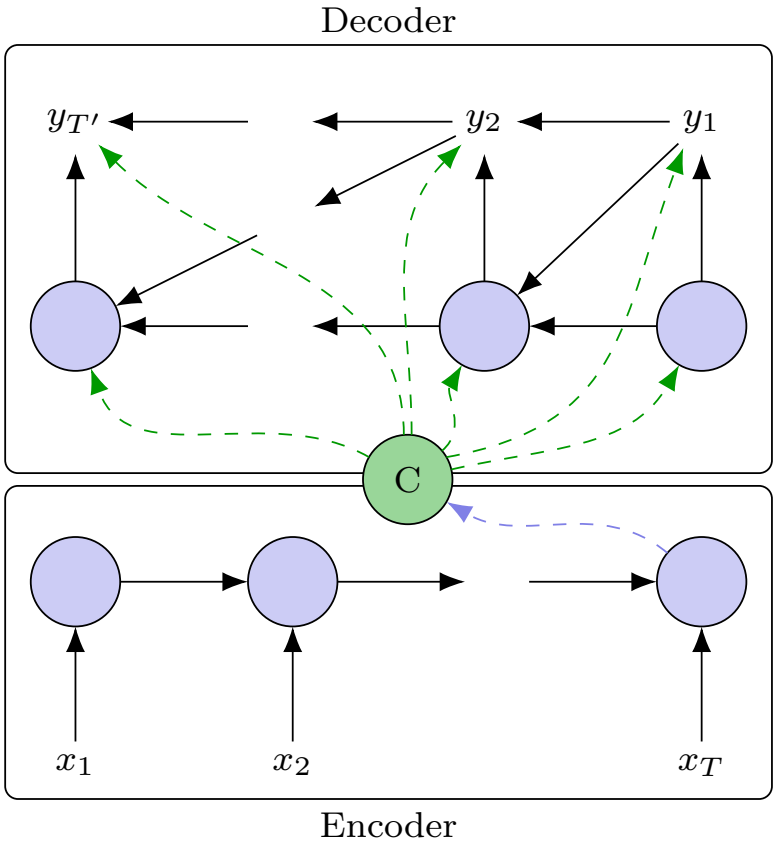


Sequence to Sequence Model

The image shows a very high prospective of the Seq2Seq model. The "c" block captures the entire input sequence. This limitation already suggest the need of an attention model, as the latent representation cannot contain all information about the entire input sequence.

7.4. RNN ENC-DEC

RNN Enc-Dec was introduced in the work **"Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation" (Cho et al.)**, in which a context vector is used for word decoding.



RNN Enc-Dec Architecture

The RNN Encoder-Decoder model is fundamentally identical to the Seq2Seq model, in that both follow the same principle of transforming a variable-length input sequence into a variable-length output sequence via a fixed-length vector representation, known as **context vector**.

This approach requires the **encoder and decoder to be trained jointly**, which allows the model to learn a latent representation that depends on the context vector. A significant improvement of this model is the introduction of GRUs (Gated Recurrent Units), which offer advantages in computational efficiency and learning capability over traditional RNNs.

However, as before, a critical problem with this approach is the need to compress all relevant information in a source sentence into a fixed-length vector. This can be particularly difficult when **dealing with long sentences**, as the model may not be able to capture all the necessary information in a compact representation. As a result, the quality of the output sequence generated by the decoder may decrease as the length of the input sequence increases.

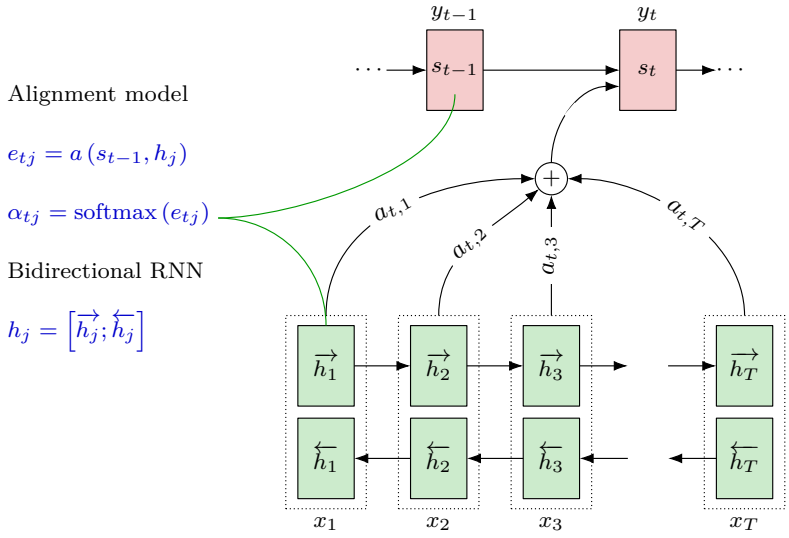
7.5. RNN WITH ATTENTION

Because of the above problem, the community realized that RNNs alone were not sufficient and that the integration of an attention module was needed. The same working group published the article **"Neural Machine Translation by Jointly Learning to Align and Translate" (Bahdanau et al.)**, in which the attention mechanism was introduced.

In an RNN architecture with attention, the encoder is a **bidirectional LSTM**, which

enhances latent representations for each input word. Indeed, every word is represented by a vector that is the concatenation of the hidden states of the LSTMs that read the sequence both left-to-right and right-to-left.

The decoder, on the other hand, generates the output sequence one step at a time, using the latent representations of the encoder and the dynamic context provided by the attention mechanism. **This context is not fixed**, but changes with each decoding step, allowing the decoder to "pay attention" to different parts of the input sequence depending on which part of the output it is generating. The context vector is modulated by attention variables α_{ti}.



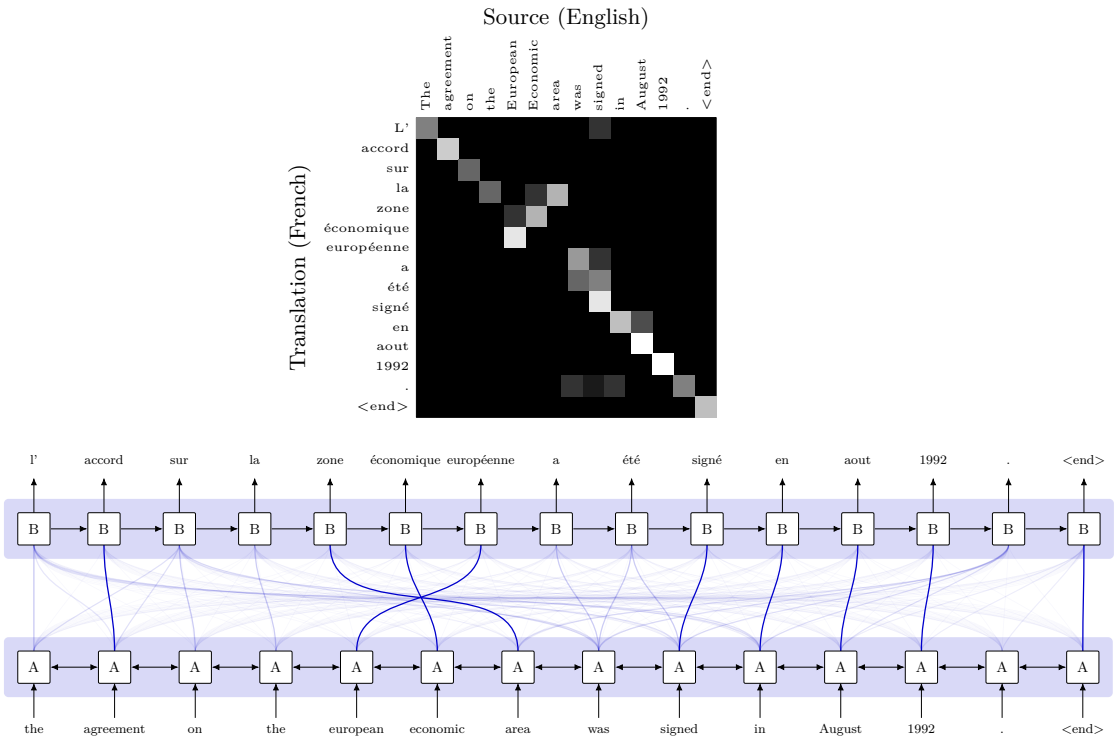
RNN with Attention Architecture

The attention mechanism works as follows:

1. **Calculation of Latent Representations**: For each word in the input sequence, a latent representation h_j is computed by concatenating the outputs of the bidirectional LSTM units:
$$h_j = \left[\overrightarrow{h_j} : \overleftarrow{h_j}\right]$$
2. **Model Alignment**: When generating the output word at time t, an **alignment score** e_{tj} is computed between the previous state of the decoder s_{t-1} and each latent representation h_j of the input sequence:
$$e_{tj} = a(s_{t-1}, h_j)$$
The alignment model a can be implemented with a simple neural network.
3. **Weights of Attention**: A softmax function is applied to the alignment scores to obtain the attention weights α_{tj}:
$$\alpha_{tj} = \text{softmax}(e_{tj})$$
These weights represent the relative importance of each input word to the current output word.
4. **Calculation of Vector Context**: The vector context c_t for time t is calculated as a weighted sum of the latent representations, using the attention weights:
$$c_t = \sum_{j=1}^{T} \alpha_{tj} h_j$$

The embeddings of all inputs are multiplied by their respective attention scores (in practice, most scores are zero, while one or two might have a value such as 0.8 and 0.2, for example). Thus, the context vector is the **combination of all relevant inputs for the current output**.

5. **Generation of Output**: The context vector c_t together with the previous state of the decoder s_{t-1} is used to generate the next state of the decoder s_t and the output word y_t.



The mechanism of attention in the RNN Encoder-Decoder model

In the image above, the attention matrix on the left shows the alignment between the words in the input sequence (English) and the words in the output sequence (French) for an example sentence. The lines in the diagram on the right illustrate how the attention mechanism allows the decoder to directly access representations of the encoder states by dynamically focusing on the relevant parts of the input sequence.

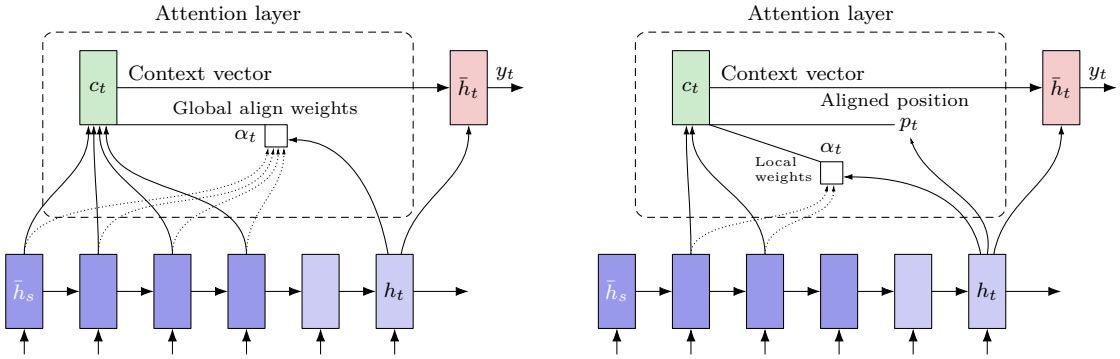
7.6. LOCAL ATTENTION

The attention used in the previous model is called **global attention** because the module attends all of the inputs. Logically, this is a waste of resources and it can also be misleading. Research showed lately that best results where obtained with **local attention**, firstly introduced in the paper **"Effective Approaches to Attention-based Neural Machine Translation" (Luong et al.)**.

In local attention mechanisms, **the model focuses only on a subset of the input sequence**, rather than the entire sequence, to compute the context vector c_t. This subset is determined by an aligned position indication p_t, which the model learns during training based on the previous decoder state s_{t-1}.

Instead of using a global view of all input tokens, local attention utilizes a **position-based approach**. The aligned position p_t determines a specific range or window around which the attention is concentrated. This approach allows the model to adapt dynamically to different

parts of the input sequence, which can vary in relevance during the output generation process.



Global Attention vs. Local Attention

The image above shows a comparison between a global attention module (right) and a local attention module (left), illustrating the difference in focus areas for the input sequence during output generation.

The computation of local attention involves the following steps:

- **Aligned Position Calculation**: To make backpropagation feasible, the aligned position p_t is determined using a gating mechanism and is dependent on the previous decoder state s_{t-1}:

  $$p_t = L_s \cdot \sigma(v_p^T \tanh(W_p s_{t-1}))$$

  Here, L_s represents the length of the input sequence, σ denotes the sigmoid function, and v_p and W_p are learnable parameters.

- **Attention Weights Calculation**: The attention weights α_{tj} are computed as a combination of soft attention and a proximity-based term, similar to a radial kernel:

  $$\alpha_{tj} = \text{softmax}(h_j^T W_a s_{t-1}) \cdot \exp\left(-\frac{(j - p_t)^2}{2\sigma^2}\right)$$

  This equation emphasizes words closer to the aligned position p_t, thereby creating a localized attention effect.

- **Context Vector Calculation**: The context vector c_t is computed as a weighted sum of the input representations h_j, using the attention weights α_{tj}:

  $$c_t = \sum_j \alpha_{tj} h_j$$

- **Decoder State Update**: Using c_t and the previous decoder state s_{t-1}, the current state $\widetilde{s_t}$ of the RNN cell is updated:

  $$\widetilde{s_t} = \tanh(W_c[c_t; s_t])$$

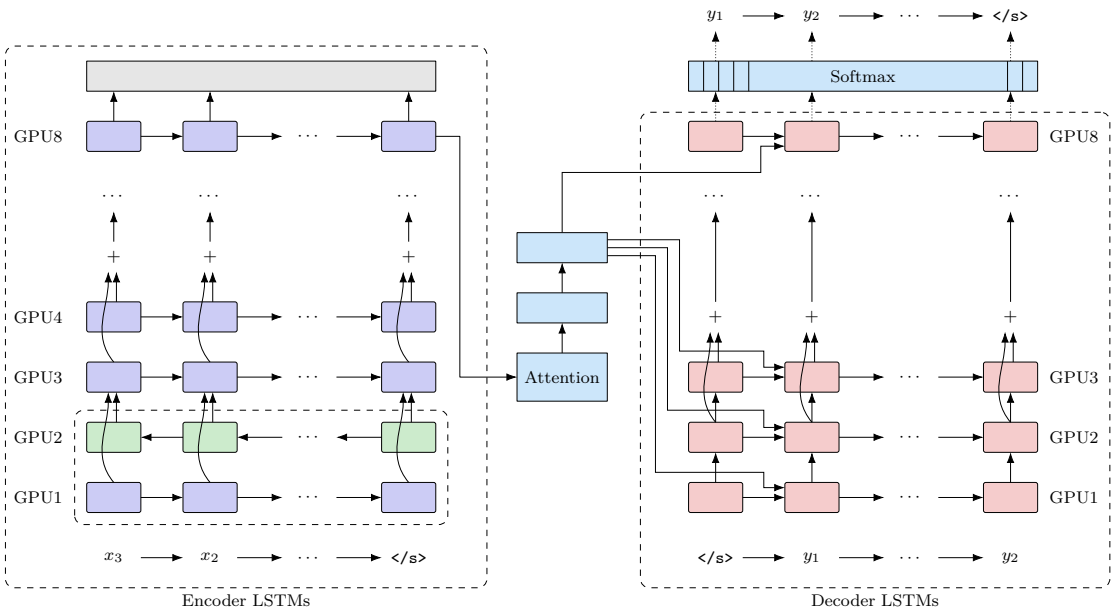
- **Output Probability Calculation**: Finally, the probability distribution over the output vocabulary is computed using $\widetilde{s_t}$:

  $$p(y_t \mid y_{<t}, x) = \text{softmax}(W_s \widetilde{s_t})$$

  ($y_{<t}$ are all the output words generated up to the time $t-1$)

7.7. GNMT (GOOGLE NEURAL MACHINE TRANSLATION)

Having discussed the significant milestones in the field of Machine Translation, we are now prepared to delve into the Google Neural Machine Translation (GNMT) system, as introduced in paper **"Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation" (Schuster et al.)**. This remarkable architecture integrates all the previously mentioned components in an innovative manner. At a high level, GNMT consists of an encoder-decoder system with attention mechanisms. The attention model leverages latent representations learned by the encoder LSTM and those associated with the target words in the decoder. The decoder's output is then processed through a softmax layer.



GNMT Architecture

Encoder Architecture: The encoder is composed of several layers of LSTMs. It starts with bidirectional lower layers of LSTMs, followed by stacked unidirectional LSTMs. Jump connections are incorporated during stacking to facilitate training. **The model is distributed over 8 GPUs, with the model divided into 8 parts, each on a different GPU**. The bidirectional lower layers of the encoder compute in parallel first. Once both are finished, the unidirectional layers of the encoder can begin computation, each on a separate GPU. This approach uses stacked LSTMs with residual connections.

Decoder Architecture: As far as the decoder is concerned, **only the output of the lower layer is used to derive the attention context**, which is then directly fed to all the upper layers of the decoder. Training is optimized using a loss function that improves the BLEU score, thus ensuring better quality of machine translation.

7.8. APPENDIX - WORD EMBEDDINGS

Because we are thoughtful authors, we also add this tidbit: word embeddings!

Computers are not able to comprehand raw words (ma dai!). For example, the word "drug"

has no meaning for my beutifull laoptop. [1] Therefore, **any task aimed at processing language** must begin with the essential representation of words.

A preliminary method is the **"bag of words"** model, which encodes words using a "one-hot" pattern. If our dataset contains sentences such as "I like the new movie!" and "I love the weather." we can represent the words as vectors:

"I"	$[1,0,0,0,0,0,0]$
"like"	$[0,1,0,0,0,0,0]$
"the"	$[0,0,1,0,0,0,0]$
"new"	$[0,0,0,1,0,0,0]$
"movie"	$[0,0,0,0,1,0,0]$
"love"	$[0,0,0,0,0,1,0]$
"time"	$[0,0,0,0,0,0,1]$

The sentences will then be represented as $[1,1,1,1,1,0,0]$ and $[1,0,1,0,0,1,1]$.

However, as is evident, this representation is not effective in showing semantic relationships between words, as each is encoded as an individual entity in a vector space and there is no way to establish that the words "love" and "like" have similar connotations. This is where word embeddings come in.

Word embeddings are representations in which contexts and similarities are captured by encoding in vector space. Similar words will have similar representations. **Word2Vec techniques** create a representation of each word in our vocabulary in a vector. Words used in similar contexts or that have semantic relationships are effectively captured through their **vicinity in vector space**. Embeddings create semantic and syntactic relationships between words. These relationships, or weights, are determined by the model and not pre-assigned (see example in the figure below).

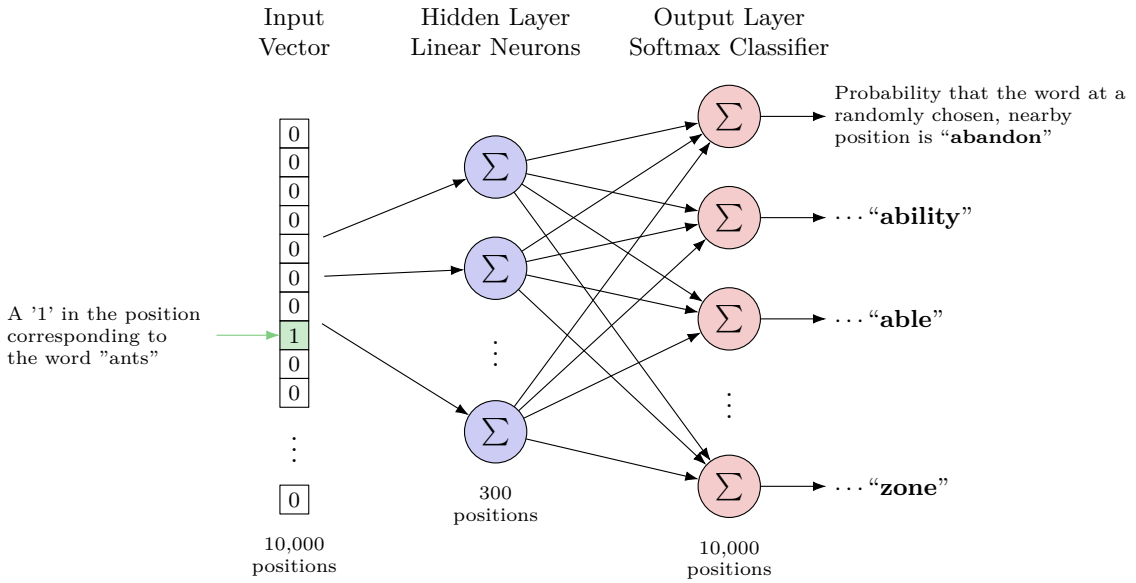
	KING	QUEEN	MAN	GIRL	PRINCE
Royalty	0.96	0.98	0.05	0.56	0.95
Age	0.92	0.01	0.90	0.09	0.85
Masculinity	0.08	0.93	0.10	0.91	0.15
Femininity	0.61	0.71	0.56	0.11	0.42

Word Embeddings Example

The image shows hypothetical features (weights) in word embeddings learned by the neural network. Consider a classic example: "king," "queen," "man," "women," "prince." In a hypothetical world, vectors could then define the weight of each criterion (e.g., nobility, masculinity, femininity, age, etc.) for each of the words provided in our vocabulary. For example, "king," "queen," and "prince" have similar scores for "nobility," while "women" and "queen" have similar scores for "femininity".

The architecture of Woxrd2Vec is essentially a **two-layer surface neural network** trained to represent words in a document or text context. During training, the input consists of all the documents or texts in the training set, each of which is represented using a **one-hot encoding of the words** present. This means that each word is represented by a vector in which a single dimension is set to 1 and all others to 0, indicating the presence of the word in the document.

[1] I'm reviewing the section and I'm not sure if Jacopo was writing or having a seizure. 🤨 - Fede



Understanding the neural network training of Word2Vec model

The hidden layer of the neural network has a **number of neurons equal to the desired size for word embedding**. For example, if you want each word to be represented by a vector of length 300, the hidden layer will contain 300 neurons. During the training process, the weights associated with the hidden layers of the network are treated as the word embedding. These weights represent network parameters that capture the semantic and syntactic features of words, allowing them to be placed in a vector space in which similar words are close to each other. In other words, each word can be viewed as having a set of weights (300 in the case of the example) that "weigh" different semantic and syntactic features, such as the context in which they appear and their relationships to other words in the text.

Eventually, we can train a neural network to assign weights to each word embedding. The new weights on connections from inputs to activation features are the **Word Embeddings**.

8. TRANSFORMERS

8.1. TRANSITION FROM RNN TO TRANSFORMERS

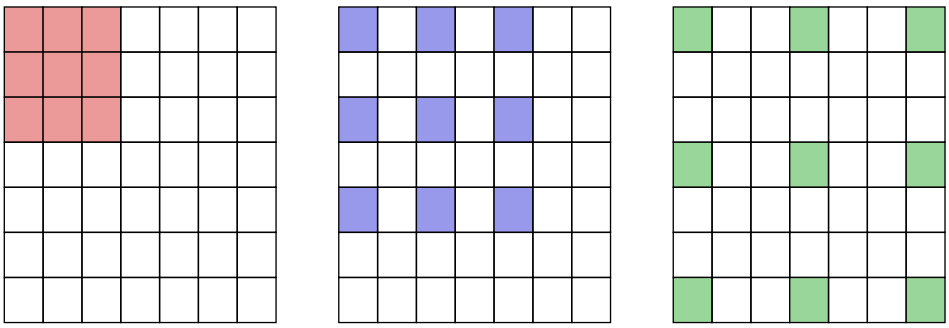
In this chapter, we will explore the significant transition from Recurrent Neural Networks (RNNs) to Transformers in the field of Natural Language Processing (NLP). A pivotal milestone in this journey, as discussed in Chapter 7, was the introduction of the attention mechanism in sequential models. Following this development, the NLP community began questioning whether RNNs were indeed the optimal choice for modeling sequences or if alternative approaches could yield better results.

The initial indication of this paradigm shift emerged with the development of CNN-like architectures for sequence processing. Traditionally, CNNs and RNNs have been distinguished by their input characteristics, with CNNs typically handling fixed-length inputs and RNNs accommodating variable-length inputs. However, since an image can be viewed as a two-dimensional input patches sequence, it becomes feasible to apply CNNs to sequence modeling tasks.

CNNs deal with sequence modeling differently from RNNs. Instead of preserving the order of the sequence, CNNs adopt a bottom-up approach, extracting features **hierarchically** from local regions of the input. While this departure from sequential processing might seem unconventional, CNNs offer several advantages. Firstly, the processing of each element in the sequence occurs **uniformly across the entire input**, facilitated by the shared parameters within each layer. This uniform processing contributes to the homogeneity of the model's operations. Additionally, CNNs exhibit enhanced scalability, particularly when accelerated by **GPU computing**, due to their inherent parallelism.

By leveraging CNNs for sequence modeling, practitioners can benefit from their efficiency in capturing **local patterns** and their ability to exploit **parallel computation**, making them a compelling alternative to traditional RNN architectures. This shift laid the groundwork for subsequent advancements in sequence modeling, ultimately leading to the emergence of Transformer architectures (refining and extend these benefits in a distinct design).

To understand how a Convolutional Neural Network extracts context from an input sequence to produce a translation, we first need to delve into a crucial concept: **Dilation**, which refers to the spacing between the elements (or receptive fields) of a filter as it moves across the input sequence.



Example of Three Different Dilation Rate

In Chapter 5, we introduced CNN architecture, focusing on the traditional convolutions, which have a dilation rate of 1, meaning that the kernel considers adjacent pixels.

Unlike traditional convolutions, dilated convolutions introduce gaps between the elements of the filter. This allows the network to capture information over larger spans of the input sequence. For instance, with a dilation rate of 2 (skipping 1 pixel) or 3 (skipping 2 pixels), as shown in the figure above, we achieve the same number of features in the next layer without changing the kernel size. This approach extends the receptive field of the convolutional filter, enabling it to incorporate more context from the input data while maintaining the computational efficiency of the original kernel size.

8.2. TEMPORAL CONVOLUTIONAL NETWORK

Convolutional Sequence Model, also called **Temporal Convolutional Network** (TCN), was the first architecture to apply a CNN to sequence processing. It was introduced in the work **"An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling" (Bai et al.)** and marked the shift from RNN to CNN for sequence input problems.

TCNs are distinguished by two main features:

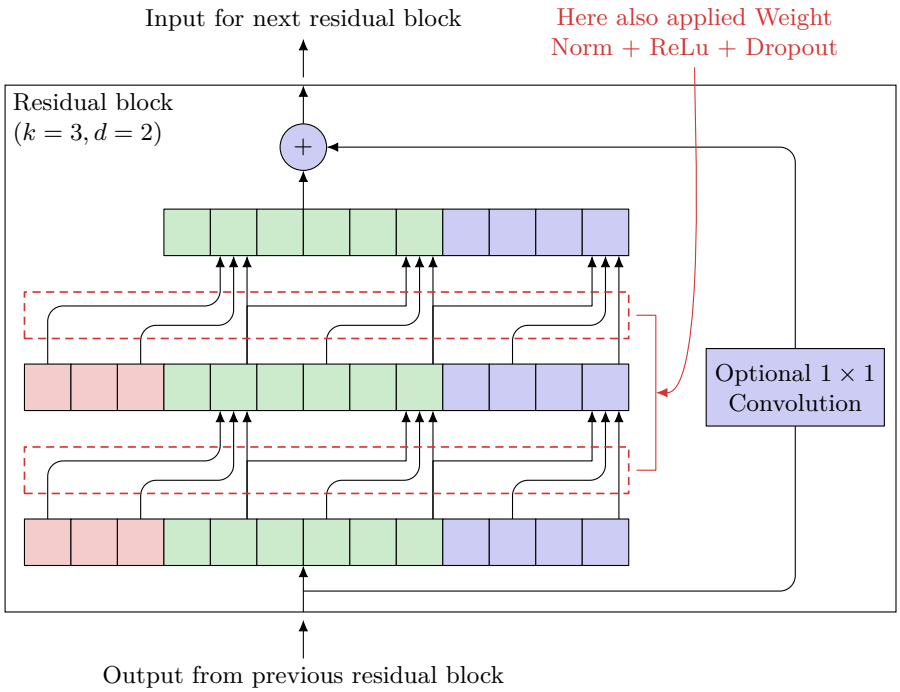
- **Sequence Length Preservation**: The architecture of TCNs can take a sequence of any length and map the input to an output sequence of the same length.
- **Causality in Convolutions**: The convolutions used in the architecture are causal, ensuring that there is no loss of information from the future to the past.

To achieve sequence length preservation, TCNs use a 1D fully convolutional network (FCN) architecture, where each hidden layer has the same length as the input layer. Length padding (kernel size $-$ 1) is added to preserve the length of subsequent layers, ensuring that they match the length of the previous layers.

To maintain causality, TCNs employ **dilated causal convolution**, which is a 1D convolution operation in which the output at time t is computed using specific elements of the input positions of the previous level, determined by the dilation rate. This method ensures that each output depends only on specific past inputs, respecting the causality constraint. This convolution operation is crucial for capturing temporal dependencies in sequential data, as it ensures that predictions are based only on past information, avoiding any "loss" of future information in past data. **Residual connections** are also used to enhance the learning process.

To put it simply, a TCN can be viewed as a combination of a 1D FCN and dilated causal convolution layers. In addition, the model uses the ReLU activation function and dropout as regularization techniques. An additional trick used is the **weight normalization block**, which normalizes the vector of weights, accelerating convergence without introducing dependencies between examples in a minibatch, which is the reason for using it.

For a better understanding see this explanatory diagram:



A global overview of Dilated Causal Convolution.

The architecture differs significantly from standard CNNs because the output of each hidden layer is obtained using dilation (1, 2, 4) and kernels of size 3. The most notable change is the causal effect: the CNN has been adapted to predict tokens sequentially. For example, the output y_t is predicted after y_{t-1} using only a "shifted" number of features from the previous level, i.e. x_t, x_{t-4}, x_{t-8}.

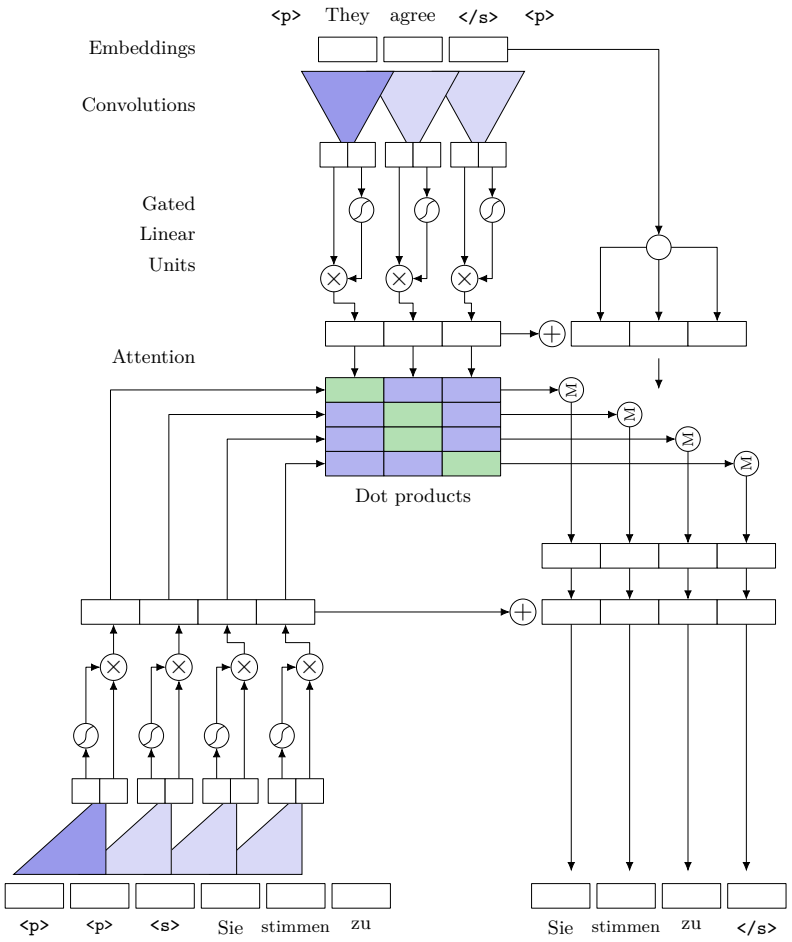
8.3. CONVOLUTIONAL SEQ2SEQ LEARNING

In **"Convolutional Sequence to Sequence Learning" (Auli et al.)** Google's Seq2Seq learning model is adopted and CNN-based approach is introduced. In this work, the task is to translate a sentence from German to English.

In the original architecture (RNN with Attention), the encoder consists of a bidirectional LSTM. The output of this bidirectional LSTM is concatenated to form the input for subsequent layers. This concatenated output is then used to construct a soft attention mechanism, which captures important relationships within the input sequence. Next, the decoder provides English translation using the latent representations associated with the concatenated output and the attention mechanism.

In this work, researchers replaced the bidirectional LSTM in the encoder with a Temporal Convolutional Neural Network (TCN), which produces latent representations used for attention computation. In addition, the decoder was also transformed to a CNN-based architecture. In both cases, encoder and decoder, Gated Linear Units and residual connections are used.

The CNN-based decoder improves training efficiency by exploiting parallel computing capabilities. Both the encoder and decoder consist of 15 levels each.



Convolutional Seq2Seq Architecture

8.4. ATTENTION IS ALL YOU NEED

Transformers are a novel neural network architecture introduced the paper **"Attention Is All You Need" (Vaswani et al.)**. They are based on a **self-attention mechanism**, forming the backbone of neural machine translation architectures that use fully connected (FC) layers and attention.

Compared to recurrent or convolutional architectures, Transformers are more efficient and easier to parallelize. This characteristic makes them faster to train and capable of handling larger datasets effectively.

The classical Transformer architecture is composed of two main components: the encoder and the decoder.

- **Encoder**: The encoder processes the entire input sequence at once and outputs an encoded sequence of the same length. Each input word is transformed into a high-dimensional vector representation.
- **Decoder**: The decoder generates the output sequence one word at a time. It is conditioned on the encoded sequence from the encoder and the previously predicted words. This ensures that the decoder takes into account both the context provided by the encoder and the sequence of words generated so far.

8.4.1. But what is Self-Attention?

Self-Attention is a mechanism that captures the context of other words relevant to the one we are currently processing. This makes it possible to create long-range weighted dependencies between word tokens in language models. In practice, it allows the model to focus on words that are important for understanding the meaning of a word at a given position in the sentence.

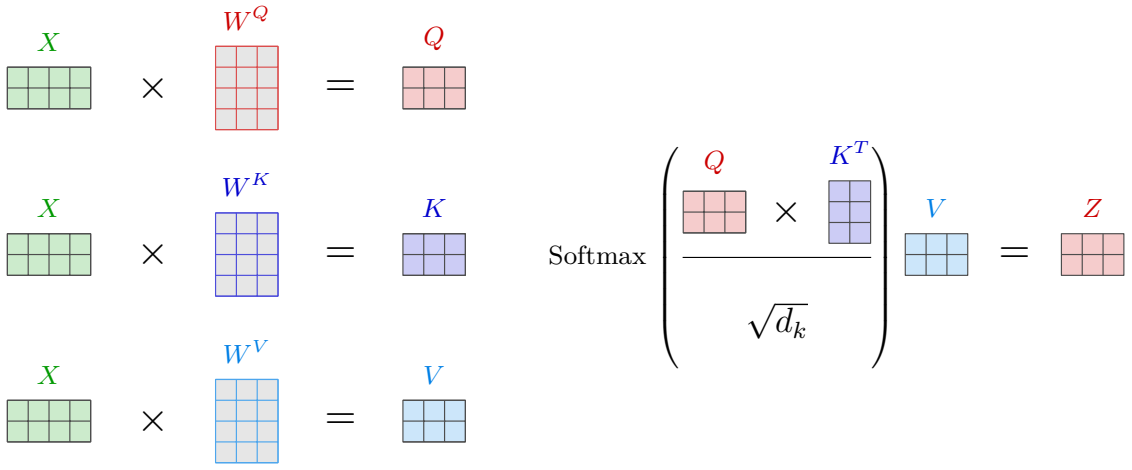
To calculate self-attention, we start by creating three vectors from the embeddings: **query (Q)**, **key (K)** and **value (V)**. These vectors are obtained by **multiplying the word embeddings** (X) **by three different weight matrices** (W^Q, W^K and W^V) learned during the training process. Here's the meaning of these vectors:

- Q: the query represents the current word for which we are searching for relevant words.
- K: the key represents the potentially relevant words.
- V: the value represents the content of the relevant words.

The self-attention formula is as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

The scalar product between query and key (QK^T) is called **score** and measures how similar two words are. In other words, it indicates **how much we need to focus on other words in the input sentence** while processing a specific word. To obtain stable scores, we divide the scalar product between the query and the keys by the square root of the size of the keys (d_k) and apply softmax. The word in that position will have the highest softmax score, but **it may also pay attention to another relevant word**. We multiply each value (V) by the softmax score, keeping the values of important words and reducing those of irrelevant words. We then sum these weighted values to get the self-attention output for that position (Z). Graphically:



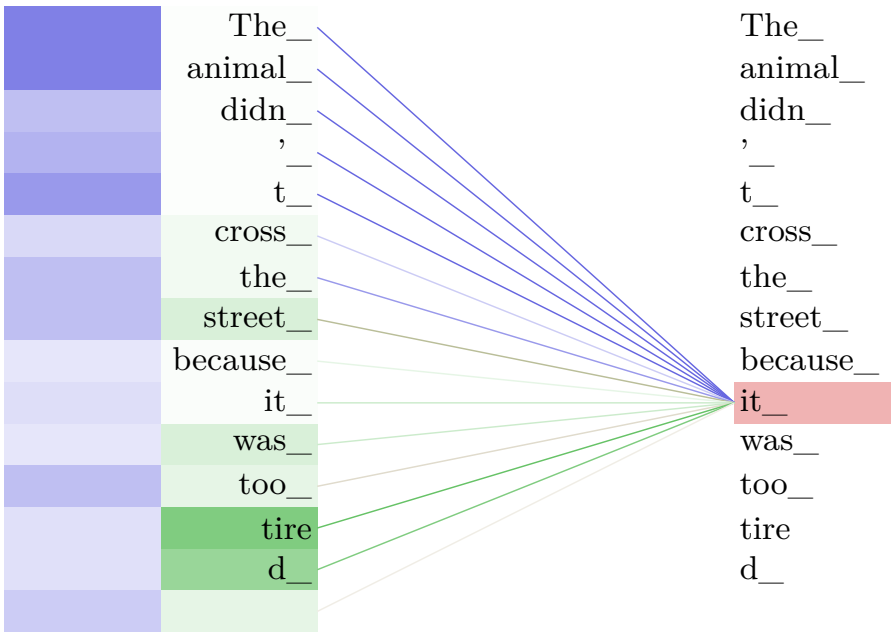
Self-Attention Calculation

One thing that is important is that soft attention is differentiable, making it easy to train the model by backpropagation.

8.4.2. Multi-Head Attention

In transformers, we use **multi-head attention** to improve the model's ability to focus on different positions in the sentence and provide the attention level with more "subspaces of representation". For example, some heads may focus on words nearby, while others may focus on words far away in the text. This allows the model to capture a wider range of information.

But how does it work? In multi-head attention, vectors of input embeddings are divided into **several separate "heads"**. Usually in the classical Transformer, this number is 8. Each head of multi-head attention operates on the linear projections we have already seen: query, key and value. After calculating the attention for each head, the results are **concatenated and multiplied by an additional weight matrix** to produce the final attention level output.



Multi Head Self-Attention

The image provided shows a visualization of the outputs using two heads. We can see that **if the query word is "it"**, **the first head focuses more on the words "the animal"**, while **the second head focuses on the word "tired"**. As a result, the final context representation takes into account all the words "the", "animal", and "tired", providing a richer and more accurate representation than traditional methods.

8.4.3. Positional Encoding

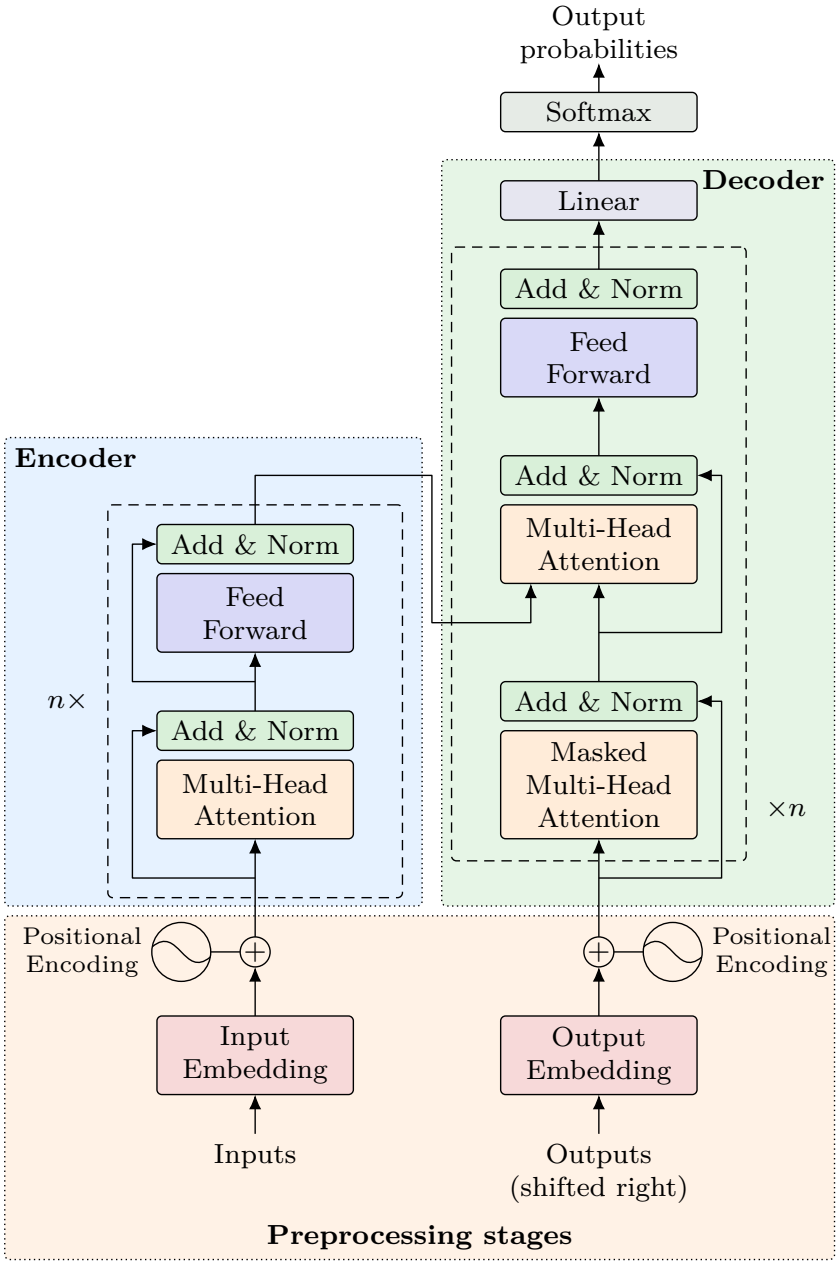
Before passing the embeddings as input to the Encoder and Decoder, it is critical to add information regarding the position of each word. Unlike other models, such as recurrent neural networks, the Transformer has no intrinsic sense of the order or position of words within a sentence. To compensate for this lack, we use **positional encoding**.

When we combine word embeddings with positional encoding in the context of Transformer models, we obtain "**time-signal embeddings**". This means that each embedding vector not only represents the meaning of the word, but also includes a component that indicates its relative position in the input sequence.

One of the common techniques for positional encoding is the use of **sinusoidal functions**, which allow different positions in the embedding vector to be represented. This method helps to maintain consistency of positional information despite arbitrary word order, which is essential for the proper functioning of the Transformer.

8.4.4. The Architecture

The architecture includes all the elements explained so far. Before moving to the encoder and decoder, embeddings are computed with positional encoding added.



Transformer Architecture

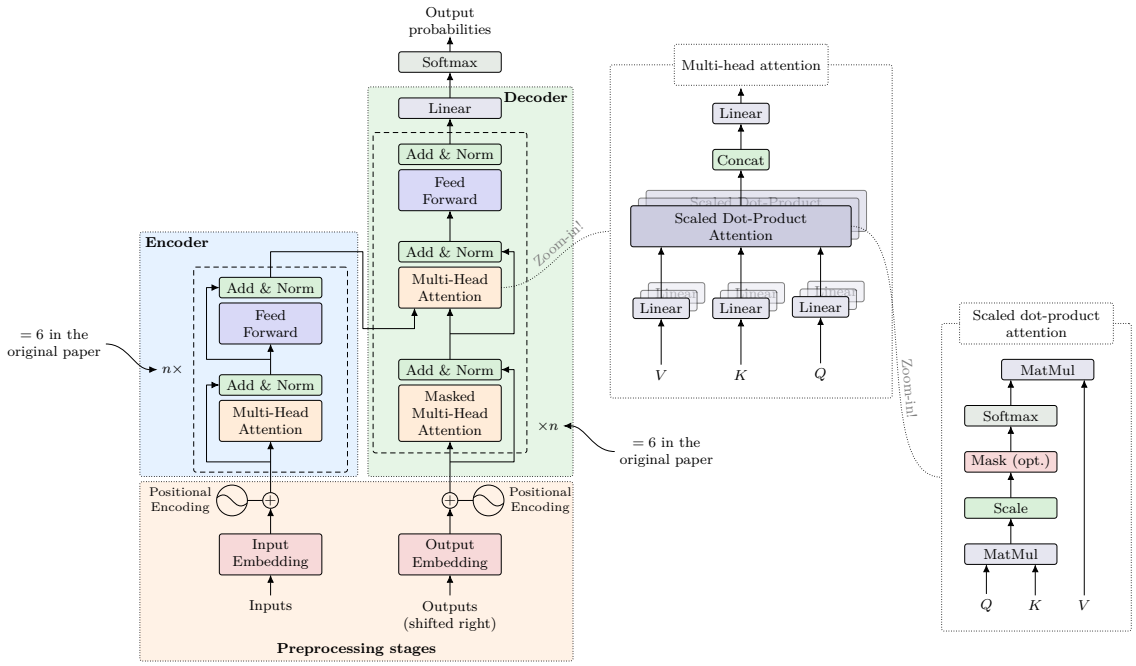
The Encoder consists mainly of two modules: the **Multi-Head Attention** and the **Feed-Forward Network**, which processes the output vectors of the multi-head attention. This layer is independent along different paths, enabling parallel execution.

The Decoder, in contrast, consists of three separate modules: the **Multi-Head Attention**,

the **Masked Multi-Head Attention**, and the **Feed-Forward Network**. The Masked Multi-Head Attention is used to ensure that **each position in the sequence can only attend to previous positions**, and not future ones, thereby allowing the Decoder to generate the sequence step-by-step without peeking ahead. The additional module of Attention Multi-Head is used to operate on the outputs of the Encoder stack, allowing the Decoder to focus on different parts of the Encoder input, improving the model's ability to capture information relevant to output generation.

In both the Decoder and the Encoder, **Residual Connections** and **Layer Normalizations** are used to provide greater stability during training and to facilitate the flow of gradients through the various layers of the architecture.

In the visual representation below, it is emphasized with a kind of magnifying glass how a multi-head attention module is structured, which in turn includes numerous self-attention modules for each of its heads.



Transformer Architecture Extended

In conclusion, transformers offer significant advantages, such as **efficient parallel computations** and improved handling of long sequences compared to previous architectures like RNNs and CNNs. Moreover, the paradigm of **pre-training language models on large-scale datasets**, followed by fine-tuning for specific tasks, has transformed natural language processing.

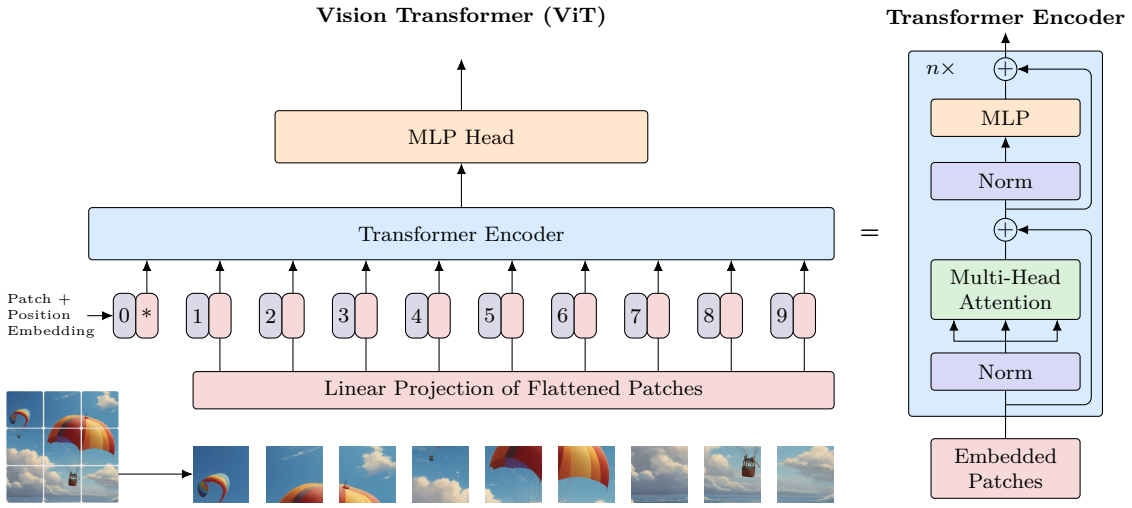
After the advent of transformers, the community began exploring models that utilize **either the encoder or decoder independently**. The first category leverages output hidden states, which can serve as features in broader models designed for diverse tasks (e.g., BERT). The second category, specifically designed for text generation, proves highly effective for applications such as machine translation or summarization (e.g., GPT series).

8.5. VISION TRANSFORMER (VIT)

In the context of convolutional neural networks, a significant limitation has been the concept of **receptive field**, which is related to the size of the convolutional kernel. This limitation has affected the ability of CNNs to capture complex, long-range relationships within images.

Visual Transformer, introduced in **"An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale" (Dosovitskiy et al.)**, marked a fundamental step in the evolution of neural networks for computer vision. ViT extended the capability of convolutions by applying the concept of self-attention to images as well, thus transferring the Transformer model from the field of natural language processing (NLP) to the field of computer vision, dealing with three-dimensional rather than two-dimensional data.

Self-attention is a mechanism for constructing **learnable long-range semantic relations**. In this case, the concept is transformed into **learning the relationships between pixels in fixed areas of the image**. The main difference of self-attention from NLP is that, in computer vision, it is used to model self-similarity within images. For example, if there are two cars in the background of an image, the areas that include these objects will be detected as similar. Therefore, to pass an input image to the encoder, it is divided into "**patches**", deciding in advance the number of divisions.



ViT architecture

The input (as a patch) is embedded in a smaller size to reduce complexity. In addition, ViT adds an additional learnable embedding called **class token**, colored gray in the figure above, which is a randomly initialized embedding vector learned during the training process. This is used to collect global information from the image through the Transformer's self-attention mechanism. At the end of the process, the MLP (Multilayer Perceptron) head **only looks at the data from the class token of the last layer** and no other information.

When trained on medium-sized datasets such as ImageNet, such models show modest accuracies (a few points below ResNets of comparable size). Transformers lack some of the inductive biases present in CNNs, such as translation equivalence and locality, and thus **do not generalize well when trained on insufficient amounts of data**. However, the situation changes if the models are trained on larger datasets (14M-300M images).

8.6. SWIN TRANSFORMER

Unlike ViT, which is not optimized for dense recognition tasks, the Swin Transformer can serve as a general skeleton for these tasks as well. Presented in **"Swin Transformer: Hierarchical Vision Transformer using Shifted Windows" (Hu et al.)**, this model builds hierarchical feature maps by joining image patches in the deepest layers. By using dynamic partitioning, it improves the accuracy of localized representations.

The Swin calculates self-attention in sub-areas of the image. At each level, the Swin Transformer partitions the image into regular windows and calculates the self-attention within each window. When moving to the next level, the window layout is updated, creating **new windows that may partially or completely overlap the previous windows**. This approach allows the model to manage and integrate information from different parts of the image effectively, improving its ability to capture complex, long-range relationships.

9. GENERATIVE MODELS

So far we have explored **Supervised Learning**, the goal of which is, given a set of data $(\mathbf{X}, y)$, to learn a function that maps to $\mathbf{X} \longrightarrow y$. For example, regression or image captioning belong to this category of machine learning tasks.

Another fundamental class of tasks is **Unsupervised Learning**, which, given only the data $\mathbf{X}$, the algorithm learns some underlying hidden structure. For example, tasks such as clustering and density estimation belong to this class.

In general, models that belong to unsupervised learning can be divided into two categories: **non-probabilistic** and **probabilistic generative**. The latter will be the subject of this chapter. The goal of this class of models is to solve the so-called **density estimation** task.

Given a set of training data with a certain `distribution of training data` $\sim p_{\mathbf{data}}(x)$, the model aims to approximate this underlying distribution in order to generate new samples. To do this, the model learns a `distribution of generated samples` $\sim p_{\mathbf{model}}(x)$, such that $p_{\mathbf{model}}(x)$ is as close as possible to $p_{\mathbf{data}}(x)$.

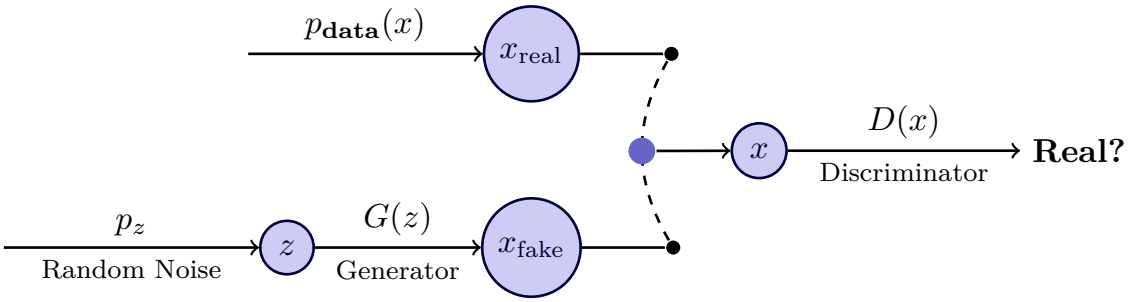
We can further categorize the generative models according to whether the density of the samples has been explicitly specified or not, namely **Explicit Density Estimation Models** (e.g., $p_{\mathbf{model}}(x)$ should be a normal distribution whose parameters are unknown) and **Implicit Density Estimation Models**. An example of Explicit Density Estimation are VAEs, which explicitly define and solve for $p_{\mathbf{model}}(x)$. An example of Implicit Density Estimation are GANs, which learn a model that can sample from $p_{\mathbf{model}}(x)$ without explicitly defining it.

9.1. GENERATIVE ADVERSARIAL NETWORKS (GANS)

The Generative Adversarial Networks, first introduced in the article **"Generative Adversarial Nets" (Goodfellow et al.)**, are probabilistic generative models that rely on implicit density functions. Instead, GANs exploit an **adversarial approach with two neural networks**: the **Generator** and the **Discriminator**.

The central role is that of the Generator: it learns to transform random noise into **complex samples that resemble those from a target distribution** (e.g., real images). This transformation process is supervised by the Discriminator, which simultaneously **learns to distinguish between real data samples and those generated by the Generator** (i.e., "fake" samples).

The adversarial aspect of this setup is crucial, where the Generator aims to produce samples indistinguishable from the real data in order to fool the Discriminator, while the latter tries to accurately differentiate between real and fake samples. This adversarial interaction drives both networks to improve iteratively, as each performance improvement in one network requires a corresponding adjustment in the other.



GAN Architecture

As you can see, the Generator takes random noise z from a simple distribution p_z and transforms it into fake samples x_{fake} that resemble real data. The Discriminator receives both real samples x_{real} from the true data distribution $p_{\textbf{data}}(x)$ and the fake samples from the Generator. It then classifies them as real or fake, $D(x)$. The adversarial training process drives the Generator to produce increasingly realistic samples to fool the Discriminator, which in turn becomes better at distinguishing real from fake.

9.1.1. Objective Function of GANs

The training procedure involves two "players", the Generator and the Discriminator, who are jointly trained through a **min-max optimization problem**:

$$min_{\theta_g} max_{\theta_d} \left[E_{x \sim p_{data}} \log(D_{\theta_d}(x)) + E_{z \sim p_z} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Discriminator Output for Real Data

Discriminator Output for Generated Fake Data $G(z)$

The Discriminator outputs a probability in the range (0,1) for real data. Each network has its parameters: θ_d for the Discriminator and θ_g for the Generator. Each module is described by a function, $\mathbf{D}_{\theta_\mathbf{d}}(\mathbf{x})$ and $\mathbf{G}_{\theta_\mathbf{g}}(\mathbf{z})$.

The Discriminator (θ_d) aims to maximize the objective such that $D(x)$ is close to 1 (real) and $D(G(z))$ is close to 0 (fake). Conversely, the Generator (θ_g) aims to minimize the objective such that $D(G(z))$ is close to 1 (tricking the Discriminator into believing $G(z)$ is real).

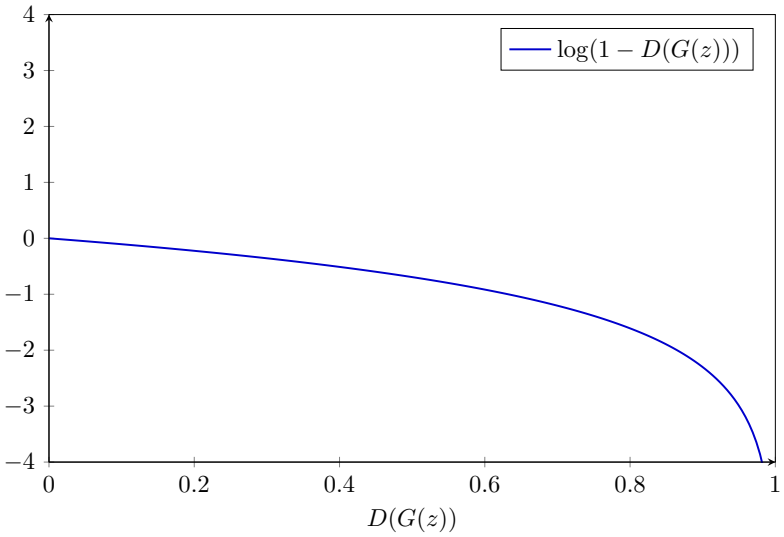
9.1.2. Training GANs

During training, we use Stochastic Gradient Descent alternating between **gradient ascent** on the discriminator and **gradient descent** on the generator, respectively:

$$\max_{\theta_d} \left[E_{x \sim p_{data}} \log(D_{\theta_d}(x)) + E_{z \sim p_z} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$
$$\min_{\theta_g} \left[E_{z \sim p_z} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

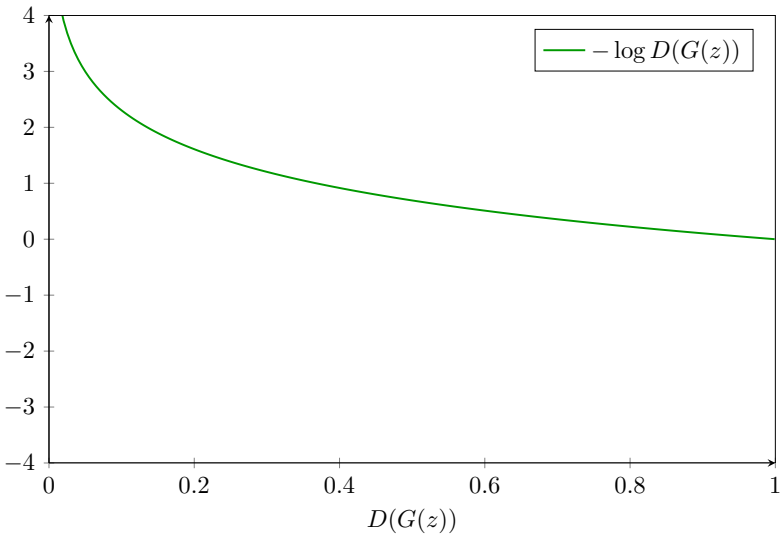
The function to be optimized is complex, so the training is **unstable**. In addition, it is

very difficult to monitor the GAN process because the objective function is not related to the quality of the generated samples.



SGD Plot for Generator during Training

In the figure, the line represents the gradient for the generator and the left part is when the generated sample is probably false. In this area, the gradient is quite **flat** and corresponds to the region where the model should learn more to improve its generating ability. In contrast, in the right part, the gradient signal dominates, which means that the sample is already good and not useful for learning purposes. Conceptually, **we waste a lot of resources in our area of interest, making learning more difficult**.



SGD Plot (Inverted) for Generator during Training

Therefore, one way to improve training is to modify this part of the objective function and turn it into a **maximization problem**. This means that instead of minimizing the probability that the discriminator is correct, the model now **maximizes the probability that the discriminator is wrong**. As shown by the curve above, now the flat area has been inverted in the region outside our interest. Thus, the new training procedure requires alternating between two **ascending gradient** procedures:

$$\max_{\theta_d} \left[E_{x \sim p_{data}} \log(D_{\theta_d}(x)) + E_{z \sim p_z} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

$$\max_{\theta_g} \left[E_{z \sim p_z} \log(D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Although the training efficiency has been improved, there are still some stability problems.

9.1.3. Inference using GANs

In probabilistic generative models, the inference phase is called the **sampling phase**: after training, the Discriminator is discarded and the Generator is used to produce samples similar to the training data from random noise.

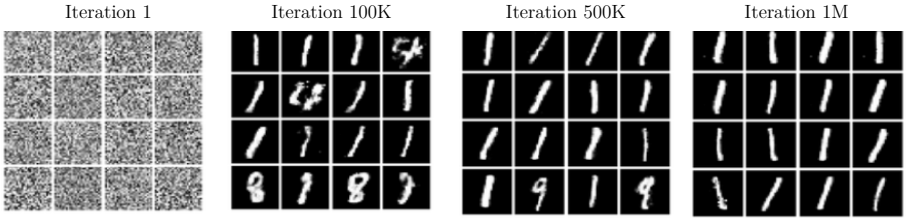
Conceptually, the **Discriminator is just an auxiliary object** to match the distribution learned from the Generator with the distribution of our data.

The results showed that if this version of GAN is trained on an **unconstrained dataset**, i.e., one with a very wide distribution of labels (MNIST is a simple dataset since the digits have a similar distribution, while CIFAR-10 is a more complex one), it **performs poorly** regardless of the models used for both the Generator and the Discriminator (such as FC or CNN networks). However, the advantages of using the Generator network are that it can be used as a backbone for CNN or in NLP.

In the following sections, we will analyze the main problems of GANs and see some articles that have tried to mitigate them. These problems can be listed as follows:

- **Overfitting and Control**: We do not have effective tools to determine whether the model is overfitting. The lack of clear indicators makes it difficult to implement techniques to control and prevent overfitting.
- **Evaluation of GANs**: There are no standardized and reliable metrics to evaluate the performance of a GAN. The quality of the samples generated can be subjective and difficult to quantify.
- **Training stability**: The training of GANs is notoriously unstable.
- **Parameter oscillations and divergence**: Model parameters may oscillate or diverge during training.

Another serious problem is the **Mode collapse**. This phenomenon occurs when the Generator, instead of capturing all modes of the target distribution, ends up capturing only one sub-mode, i.e., it learns only a certain area of the target distribution.

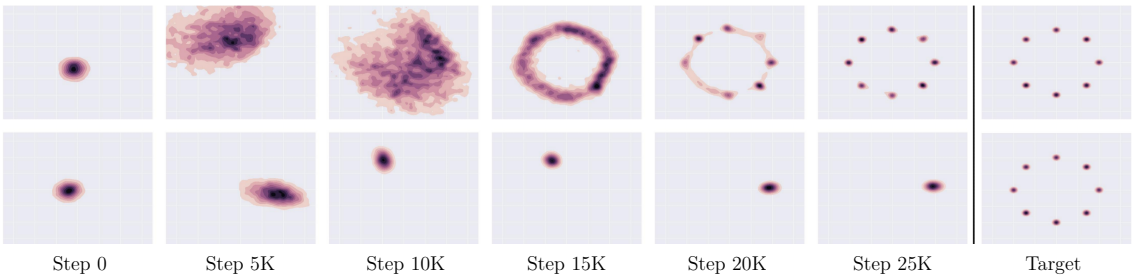


Mode Collapse Example

The figure shows an example of mode collapse with the MNIST dataset: instead of learning to generate all the numbers, the Generator focuses only on one area of the overall distribution. As a result, after many iterations, it generates only number one.

Again, the figure below shows two rows of heatmaps representing the generator's distribution at different stages of training, from the initial phase (Step 0) up to 25K steps, along with the target distribution.

Both rows display the same sequence, but the crucial difference lies in how the model behaves in these two scenarios:



Mode Collapse HeatMap

The top row illustrates the ideal behavior of the model, where the generator's distribution gradually converges towards the target distribution. In this case, we observe how the model successfully captures all data modes in a balanced manner.

The bottom row demonstrates the phenomenon of mode collapse. Here, despite the training steps being the same, we see that the model focuses on only one data mode at a time. At each step, the generator assigns a significant probability mass to a single mode, ignoring the others. This behavior persists throughout all training steps, and the model never achieves a distribution similar to the target.

This visual representation effectively highlights the problem of mode collapse: while the desired behavior is to model all data modes in a balanced way (as shown in the top row), in the case of mode collapse (bottom row), the model fixates on a single mode at a time, failing to capture the diversity of the target distribution.

9.2. DCGANS

Through the work of **"Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks" (Radford et al.)**, the DCGAN model was introduced. This represents the first GAN architecture based on deep convolutional networks and was the first model to generate high-resolution images using GAN. In fact, multiple convolutional layers are used to produce high-quality images (64x64).

The architecture of the discriminator differs slightly from a classical CNN in that it **does not use pooling**, but only convolutions with stride. In principle, pooling is effective in models such as AlexNet because it helps avoid overfitting, but in GANs it negatively affects the discriminator's ability to judge whether an image is real or fake, making discrimination too easy.

In addition, Leaky ReLU is used as the activation function, and only one fully connected layer is present before the softmax output. For normalization, the model employs batch normalization after most levels.

Batch normalization is beneficial because it speeds up training, however, applied to DCGAN could lead to reduced performance because it **introduces correlation between samples**

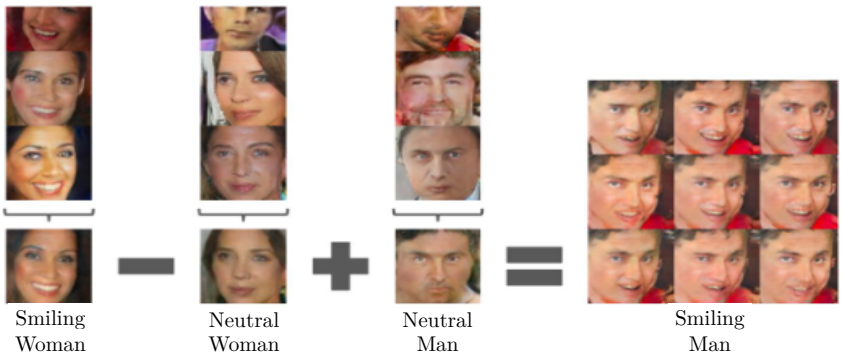
within the minibatch. The effect of correlation leads to a **loss of the diversity property** (samples not different enough) during the generation of samples by the Generator. As a result, the model will not be able to generalize.

9.2.1. Arithmetic of GAN

In the same paper, the authors explored the latent space of GANs trained on a dataset of celebrity faces.

They found that, through training, **the generator learns to map points in the latent space** (which by itself has no intrinsic meaning) **to specific images**, and this mapping will be different each time the model is trained.

Also, the article showed that, to compute the output image, the generator performs **vector arithmetic with faces**. For example, the figure below shows that a smiling woman's face minus a neutral woman's face plus a neutral man's face results in a smiling man's face.



Example of Vector Arithmetic in GANs.

In this case, arithmetic is performed on the points in latent space corresponding to each resulting face. The results showed that the Generator has some interpolation capabilities (interpolation means that the vectors in latent space can be combined to obtain a sample whose features are a combination of the two).

9.3. EVALUATION OF GANS

Evaluating a GAN model is complex because of the lack of clear methods for assessing the visual quality of the generated images. The first attempt was the use of human feedback, which naturally involves problems of subjectivity and monetary costs.

Moreover, **we are not only interested in the quality of the generated images, but also in the overall distribution of the GAN-generated samples**, which should be similar to our dataset. However, there is no direct way to calculate the probability of high-dimensional samples (real or generated) or to compare their distributions.

Thanks to **"A note on the evaluation of generative models" (Theis et al.)**, we will explore two widely used metrics: **Inception Score** and **Fréchet Inception Distance (FID)**.

To simplify the evaluation problem, we can divide into two main properties: the ability of the GAN to generate **recognizable objects** and the **variety of objects** generated.

- If the GAN produces concentrated distributions when it generates images belonging to the same class, it means that each of these images is associated with a high probability (**recognizable objects**) of belonging to that certain class.
- Globally, the dataset generated by GAN should be able to produce a **variety of objects**.

Both properties are associated with probability (likelihood). This is computed using a classifier, such as Inception-v3 (state of the art at the time). Then, we use its output (probability distribution) to calculate the inception score.

> The inception score is calculated based on the output of an image classification model and is maximized if the entropy of the distribution of labels for the generated images is minimized (**recognizable**) and the predictions of the classification model are uniformly distributed over all possible labels (**variety**).

However, this metric has an overfitting problem: with this method, a GAN that simply stores training data could still score well. If this statement does not make sense to you, try thinking, "*Why would a GAN that generates samples that are very similar to my training set be useful?*" Obviously, such a model is not useful since it simply copies the images seen.

Therefore, a better approach is to use the Fréchet Inception Distance (FID).

> Using FID, both the real and the generated samples are passed through a classification network and for a chosen layer the activations are calculated. The metric then takes the embeddings of that layer and treats them as **multivariate normal distributions**. The mean and covariance are calculated for both the generated samples and the actual data and compared using Multivariate Normal Fréchet Distance. However, **the overfitting problem is not completely avoided**.

9.4. DIFFERENT GAN LOSSES

The loss function plays a key role in GANs: **different losses correspond to different ways of comparing the two distributions**.

Let us now analyze two different loss functions compared with the standard GAN: **LSGAN** and **Wasserstein GAN**.

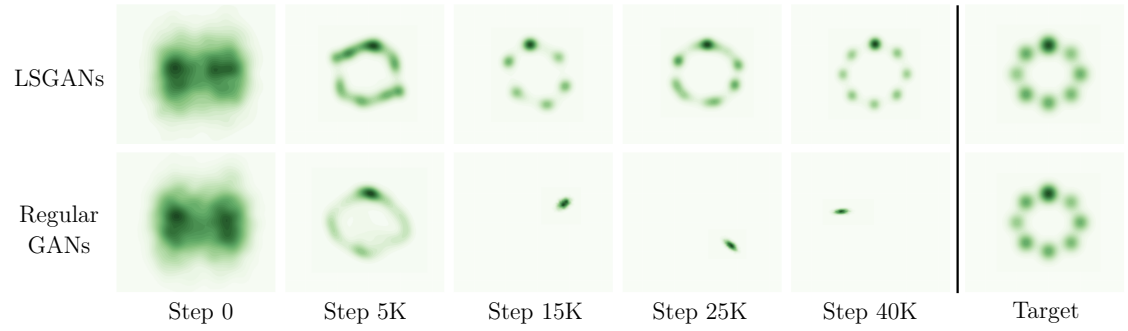
LSGAN was introduced in **"Least Squares Generative Adversarial Networks" (Mao et al.)** with the goal of stabilizing the training of a GAN.

Cross entropy is replaced by **Least Square Loss**:

$$\min_{\theta_d} \left[\frac{1}{2} E_{x \sim p_{data}} \left(D_{\theta_d}(x) - 1 \right)^2 + \frac{1}{2} E_{z \sim p_z} \left(D_{\theta_d}(G_{\theta_g}(z)) \right)^2 \right]$$

$$\min_{\theta_g} \left[\frac{1}{2} E_{z \sim p_z} \left(D_{\theta_d}(G_{\theta_g}(z)) - 1 \right)^2 \right]$$

This leads to better quality generated images (better IS and FID) and more robust generators to mode collapse:



Results Comparison between LSGAN and Regular GAN

On the other hand, the work **"Wasserstein GAN" (Arjovsky et al.)** introduced the Wasserstein GAN with the idea of modifying the loss using a new method to better align the distributions of the dataset and the generated samples.

First, the **Wasserstein distance** is a metric that aligns two distributions with the concept of **optimal transport**. It measures the minimum cost of mass transport to convert the data distribution q into the data distribution p.

Wasserstein GAN implements the idea of aligning the two distributions with this distance, providing **better gradients and more stable training**.

The most important result was that by implementing the Wasserstein GAN with the DCGAN architecture, it is possible to **correlate sample quality with the value of the Wasserstein distance**. This represents a huge improvement over other GANs. For example, not even DCGAN with the original divergence was able to achieve this property.

However, the more complex the distance, the more difficult the training becomes.

9.5. GANS TRAINING TRICKS

In this section, we explore the tricks suggested in this paper: **"Improved Techniques for Training GANs" (Goodfellow et al.)**.

Feature Matching aims to stabilize GAN training by modifying the generator's objective. Instead of trying to directly deceive the discriminator, the generator focuses on **matching the statistics of features** from an intermediate layer of the discriminator. By minimizing the difference between these feature statistics for real and generated data, the generator avoids overfitting to the discriminator's current decision boundary, resulting in more stable training and higher-quality samples.

Mini-batch Discrimination helps the generator produce more diverse outputs by allowing the discriminator to **examine relationships between samples in a mini-batch**. An additional layer in the discriminator computes differences between pairs of samples, providing information on how each sample compares to others in the batch. This enables the discriminator to identify if the generator is creating similar or identical samples, thus encouraging the generator to generate a wider variety of outputs and avoiding the problem of mode collapse.

The **Virtual Batch Normalization (VBN)** is a normalization method that extends traditional batch normalization. Regular batch normalization makes the output of a neural network for an input example highly dependent on other inputs in the same minibatch. To avoid this problem, in VBN each example is **normalized based on statistics collected on a reference batch of examples** chosen once and fixed at the beginning of training, and on itself. The reference batch is normalized using only its own statistics.

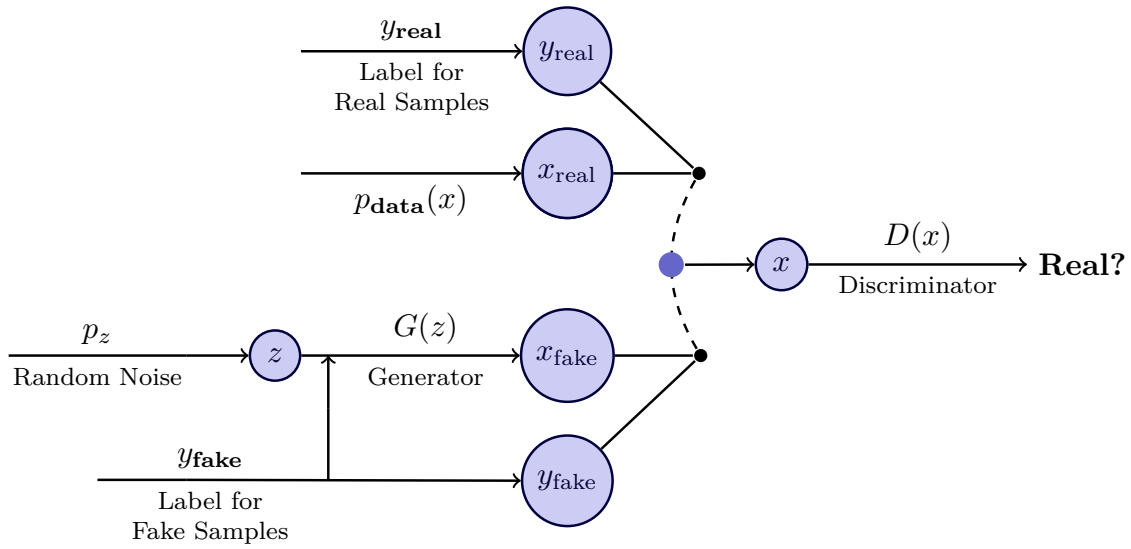
9.6. GANS ZOO

Since their advent in 2014, GANs have gone through numerous iterations and improvements. In this section, we will embark on a journey through the "GAN Zoo", exploring how an initially simple concept has evolved into a multitude of innovative variations.

9.6.1. Conditional GAN

Introduced in **"Conditional Generative Adversarial Nets" (Mirza et al.)**,a Conditional GAN (cGAN) **links an input with its corresponding label**, enabling the use of supervised information. For example, a photo of a lion paired with the label "*lion*" can be used to guide the generation process. **The optimization problem remains unchanged**, so the techniques and knowledge applied to standard GANs are still relevant.

In the cGAN framework, the generator incorporates the **label as an extension of the latent space**. This can be achieved by appending a one-hot encoded label vector to the latent spacez. The discriminator also utilizes the label information of real data, enhancing its ability to differentiate between real and generated samples. With this additional information, the discriminator can focus on a specific class, improving its performance.



Conditional GAN Architecture

9.6.2. Image-to-Image Translation

The paper **"Image-to-Image Translation with Conditional Adversarial Networks" (Zhu et al.)** introduces a conditional GAN architecture for image-to-image translation, known as **PIX2PIX Network**. The generative network transforms an input image into an output image, while the discriminative network evaluates how realistic and consistent the generated image looks with the input image. This approach is applicable to various tasks, such as translating photos into sketches, coloring black-and-white images, and improving image resolution. It is important to note that **training this model requires coupled/paired data**.

9.6.3. Text-to-Image Synthesis

Another interesting article is **"Generative Adversarial Text to Image Synthesis" (Reed et al.)**, which explores the use of GANs for image synthesis from textual descriptions. The generative network receives as input a textual description and a noise vector, and produces an **image that attempts to respect the content described in the text**. The discriminative network, on the other hand, distinguishes between real images with matching descriptions and generated images. This approach finds application, for example, in generating verbally described scenes or creating artwork from descriptions.

9.6.4. Cycle GAN

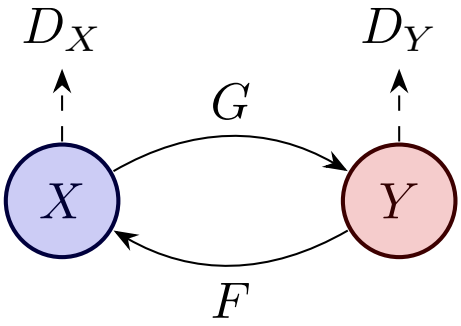
In the study **"Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks" (Zhu et al.)** an innovative methodology is introduced for translating images between two different domains, **overcoming the need for matching pairs of images**. For example, it is possible to transform photographic images into artworks in the style of Monet using CycleGAN, even when the two collections are not paired.

The CycleGAN model is based on **three main components**: the Discriminator, Generator and Translator networks, as follows:

- **Translator**: this network is responsible for "converting" an image from one domain (e.g., photographic images) to another domain (such as Monet paintings).
- **Discriminator**: trained to determine whether an image belongs to the target domain. It also verifies that the translated image, when converted back to the original domain, actually looks similar to the source image.
- **Generator**: this network takes care of transforming the translated image back to the original domain, trying to preserve fidelity to the initial version.

The CycleGAN architecture modifies the classical GAN training process by adding a **Cycle-Consistency Loss (similar to L1 loss)** to ensure that the image returned after translation and reconstruction is as close as possible to the original image.

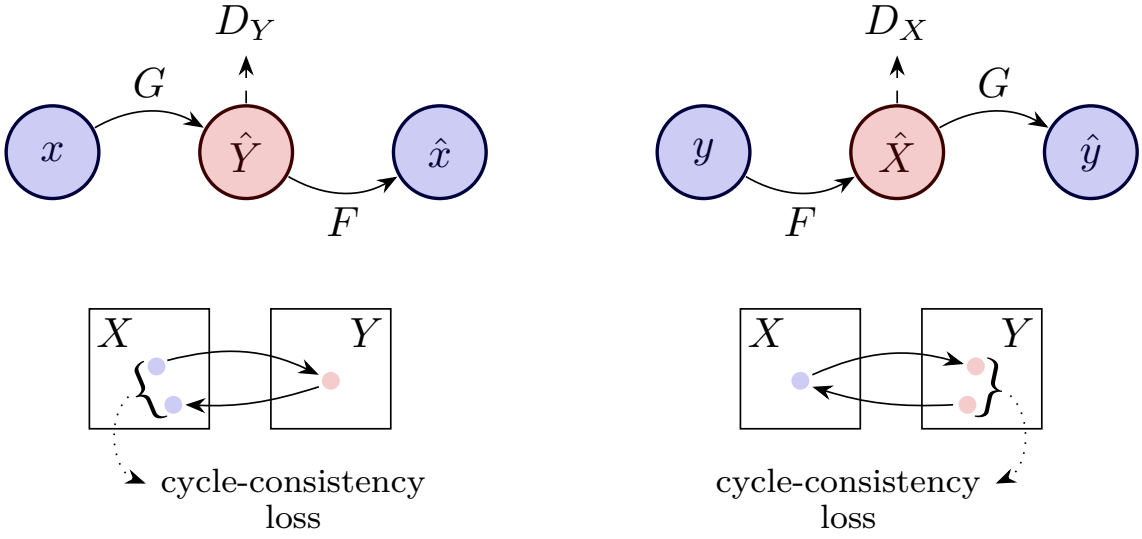
The model operates in both directions: it translates an image from one domain to another and then brings it back. This operation is replicated by reversing the roles of the domains and sharing parameters between the two directions.



Cycle GAN Rolled Architecture

The figure represents the architecture of CycleGAN. On the top, the generative networks G and F handle the translation of images between the X and Y domains. The discriminators D_X and D_Y evaluate the generated images to confirm whether they belong to the respective domains.

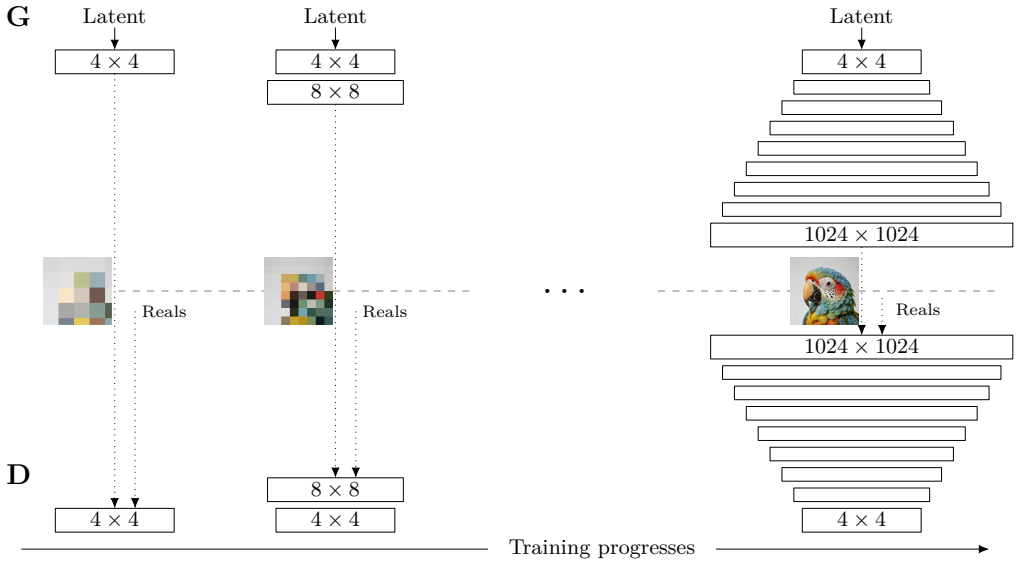
Below we have the "unrolled" versions of the model and the concept of Cycle-Consistency Loss is illustrated, which ensures that an image translated between the two domains and then reconverted retains a similarity to the original image.



Cycle GAN Unrolled Architecture

9.6.5. Progressive GAN

"Progressive Growing of GANs for Improved Quality, Stability, and Variation" (Laine et al.) introduces the Progressive GAN model, which introduces a novel approach for generating high-quality images by a process of **progressive growth of the Generator and Discriminator networks**. This method involves the gradual expansion of the networks during training, thus enabling high-quality images.



Progressive GAN architecture

The principle behind this approach is that, at each stage of training, both networks benefit from the parameters learned in the previous stages. Initially, the networks work with smaller image sizes. As the training continues and stabilizes, new blocks of layers capable of handling larger images are added. This gradual growth process allows the model to **first learn the main structures and then refine the details**, thus improving the final quality of the generated images.

9.6.6. Style GAN

The model created by NVIDIA and described in the paper **"A Style-Based Generator Architecture for Generative Adversarial Networks" (Laine et al.)** is named StyleGAN and brings significant innovations to the architecture of GANs through the use of a Generator that is able to capture and manipulate in detail the **style attributes of images**, such as pose, hair type, or smile shape. It also allows these attributes to be modified stochastically, using random noise to vary the styles.

In the StyleGAN model, fundamental changes were made to the Generator architecture, such as the introduction of a **style vector**. This vector, derived from a latent representation via 8 fully connected layers (FC), captures the stylistic features of the image.

StyleGAN uses **Instance Normalization** to manage the stylistic attributes of images. This process calculates, for each image, the mean (μ) and standard deviation (σ) of the pixel values. Normalizing the image with these statistics **removes the variations in brightness and color that are specific to each image, without altering the structure of the content.** The statistics μ and σ then capture stylistic aspects such as texture and color. Instance Normalization is applied at different layers of the network to

extract and control various aspects of style. These stylistic attributes are then reintegrated at the final generation stage, allowing precise control over the stylistic details of the final image.

9.6.7. Drag GAN (2023)

GANs are back with an exciting new feature! 😋 DragGAN represents a recent advancement in GAN models, allowing interactive editing of images via a simple drag-and-drop method, as described in **"Drag Your GAN: Interactive Point-based Manipulation on the Generative Image Manifold" (Pan et al.)**.

In the DragGAN model, the user defines pairs of **control points** and **target points**, which indicate the areas of the image to be modified. The goal is to drag portions of the image from the position of the control points to the target points.

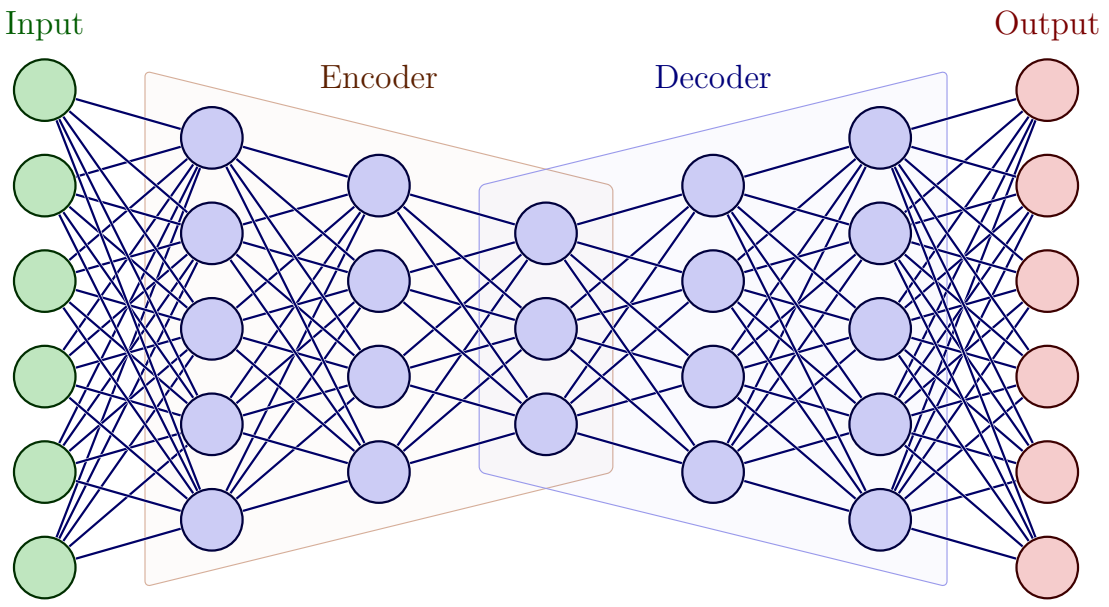
The optimization process is divided into two main steps:

1. **Motion Supervision**: A module that guides the control points to the desired target points.
2. **Point Tracking**: A module that continuously updates the position of control points to reflect changes.

DragGAN demonstrates that, despite the emergence of more advanced models, GANs continue to innovate, offering practical and interactive solutions for image generation.

9.7. AUTOENCODERS

Autoencoders are unsupervised models that learn to represent input data in a space of reduced size and then reconstruct it. This process occurs in two main steps: the encoder and the decoder.



Autoencoder Architecture

The input is passed through the **Encoder**, a neural network that compiles data into a latent z representation of lower dimension. This representation contains only the **most relevant features** extracted from the input. Next, the **Decoder**, which is a symmetric network with respect to the encoder, decodes the latent representation z back to the size of the input to generate a reconstructed sample x'. The generated sample is then compared with the original input x via a **reconstruction loss function (Norm L2)**.

The goal of training is to optimize the encoder and decoder weights to minimize this loss function. In particular, the encoder must refine its ability to extract essential features from the data, while the decoder must improve the quality of reconstruction of compressed features. After training, **the decoder part is generally discarded and the encoder can be used to initialize a supervised model** for predicting labels from the z feature space.

Keeping the bottleneck (z) small forces the autoencoder to learn an intelligent representation of the data. Another approach to force the autoencoder to improve its representation is to add random noise to the inputs and have the model retrieve the original noise-free data, as described in the paper **"Extracting and Composing Robust Features with Denoising Autoencoders" (Bengio et al.)**, which introduces the **Denoising Autoencoders**.

In many modern autoencoder architectures, the encoder is implemented as a CNN, to compress the data into a more compact representation. The decoder, which has a similar but opposite structure to the encoder, uses **Transposed Convolution Layers** to reconstruct the original image from this compact representation. Transposed convolution layers work in the opposite way to the convolution layers used in the encoder: **while the encoder reduces the size of the image, the decoder increases it**. This is useful for recreating the image from the reduced representation.

9.8. VARIATIONAL AUTOENCODERS (VAES)

Traditional autoencoders are designed to **replicate input data from the same dataset**. Variational autoencoders enhance this capability by learning distributions for latent features, allowing them to generate **unique images** with **similar features** to those in the training dataset. VAEs create a **continuous latent space** that facilitates sampling and interpolation, generating new samples that are similar to the inputs but different from those in the training set.

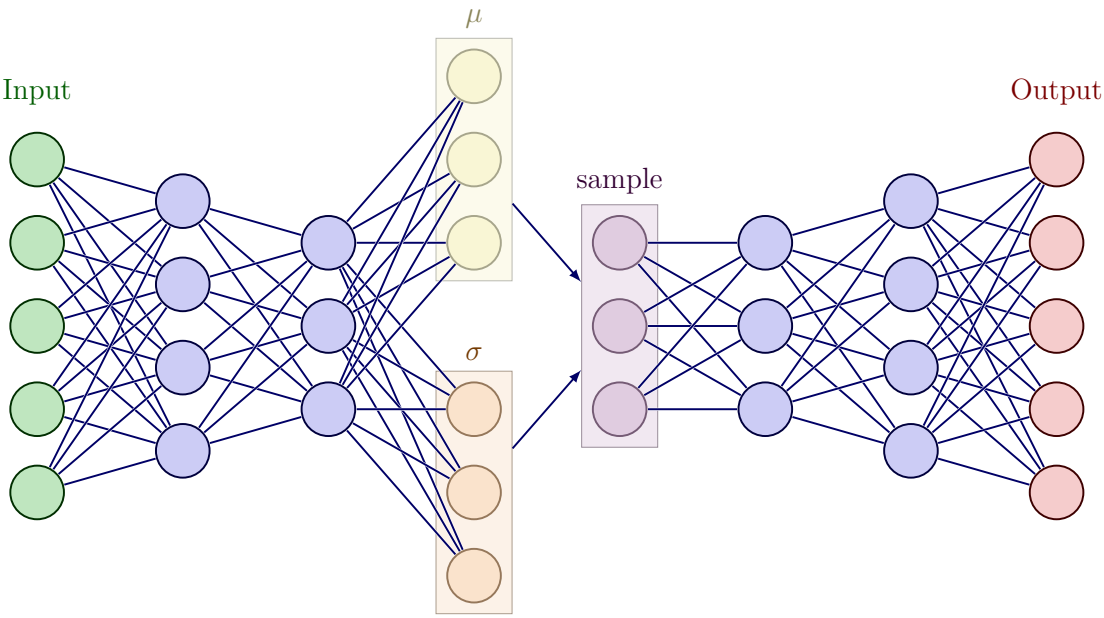
VAEs belong to the family of **explicit density estimation** models. Unlike models that indirectly measure the distribution of the data, VAEs use a direct approach to estimate the likelihood function of the data. The likelihood function for a sample of the dataset x can be expressed as:

$$p_\theta(x) = \int p_\theta(z) p_\theta(x|z) dz$$

where z corresponds to the latent representation vector of our training data.

The formula represents a likelihood function, thus we would like to learn the parameters θ that maximize it, i.e. such that generated data fit training data.

This formula represents a likelihood function, and the goal is to **learn the θ parameters that maximize it**, that is, those that best fit the generated data to the training data.



Variational Autoencoders Architecture

However, we cannot compute this likelihood directly because it is mediated by z, which is a **latent representation learned from the training samples and not from the data itself**. Consequently, we do not have direct access to the data on which we are integrating.

To explain this problem, let us consider the formula in detail. It involves two probability distributions: $p_\theta(z)$ (**prior distribution**) represents the latent space, while $p_\theta(x|z)$ represents the Decoder, which samples z from the prior distribution $p_\theta(z)$ and transforms it into a sample x that must be similar to the training data.

Since the Decoder samples from the prior distribution, the latent space must be a simple probability distribution to facilitate this process. Therefore, the latent space is generally a **Gaussian Multivariate**. However, we cannot directly produce $p_\theta(z)$ since it is derived from the training data. **This is the integration problem**.

Here is a possible solution! We could obtain $p_\theta(z)$ using Bayes' theorem. However, another problem would arise when we try to calculate the a posteriori probability distribution $p_\theta(z|x)$, since the denominator is intractable (🥲):

$$p_\theta(z|x) = \frac{p_\theta(x|z)p_\theta(z)}{p_\theta(x)}$$

In this context, $p_\theta(z|x)$ represents the **latent space distribution extracted from the given input**.

As usual in machine learning, the solution is VERY simple: we throw in a neural network. In this case, the network is the **Encoder**, which is trained to map the input image to the latent space, $q_\phi(z|x)$. In this way, the encoder approximates the a posteriori distribution $p_\theta(z|x)$. **Beware that the encoder does not provide a global solution to the optimization problem, only an approximation: it provides a lower bound on the likelihood of the data!**

Since we are modeling probabilistic data generation, both the encoder and the decoder are probabilistic models and are described by their own parameters, ϕ and θ, respectively. The

Encoder learns the mean and variance associated with the Gaussian distribution of the latent space z. The Decoder learns the mean and variance of the generating distribution, which should be as similar as possible to the dataset data. **Then, the Variational Autoencoder can be seen as an Autoencoder with a probabilistic component**.

The objective function can be derived as follows by applying log-likelihood and several transformations:

$$
\begin{aligned}
\log(p_\theta(x)) &= E_{z\sim q_\phi(z|x)}\left[\log p_\theta(x)\right] \\
&= E_z\left[\log \frac{p_\theta(x|z)p_\theta(z)}{p_\theta(z|x)}\right] \text{ (Bayes Rule)} \\
&= E_z\left[\log \frac{p_\theta(x|z)p_\theta(z)}{p_\theta(z|x)}\frac{q_\phi(z|x)}{q_\phi(z|x)}\right] \text{ (Multiply by Constant)} \\
&= E_z\left[\log p_\theta(x|z)\right] - E_z\left[\log \frac{q_\phi(z|x)}{p_\theta(z)}\right] + E_z\left[\log \frac{q_\phi(z|x)}{p_\theta(z|x)}\right] \text{ (Logarithms)} \\
&= E_z\left[\log p_\theta(x|z)\right] - D_{KL}\left[q_\phi(z|x)||p_\theta(z)\right] + D_{KL}\left[q_\phi(z|x)||p_\theta(z|x)\right]
\end{aligned}
$$

From the equation above, **the third term is excluded** since we cannot compute it, since we do not have access to $p_\theta(z|x)$ (remember, the encoder **approximates** the a posteriori distribution). However, this term is always a positive value since it is calculated via the KL divergence, which allows us to approximate the likelihood as a lower bound. If you do not know, **KL divergence is a measure of how similar two probability distributions are**; if they are identical, the divergence is zero, while a positive number indicates that the distributions are different.

Two terms remain: the first is the Reconstruction Loss or **Reconstruction Term**. This term is applied to the decoder to train it to **generate meaningful results** when it samples from the latent representation relative to the input data.

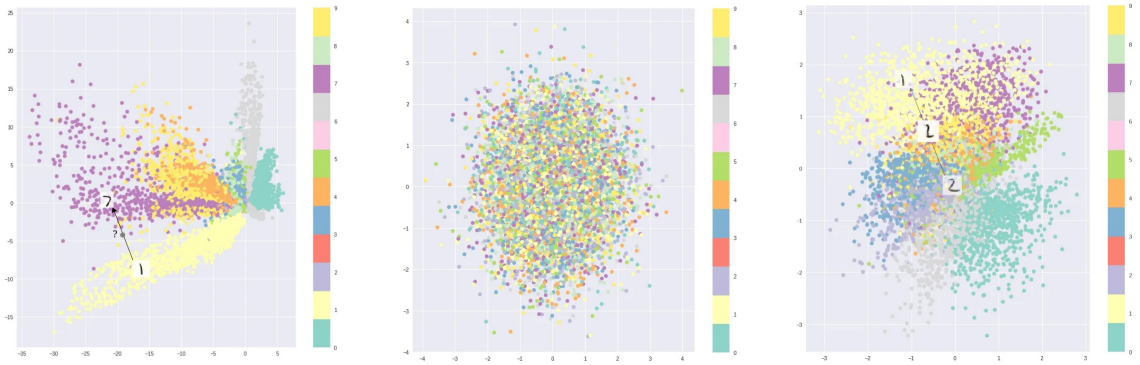
The second term can be computed in closed form because it involves the KL divergence between two Gaussian distributions and is used for the encoder (since it tries to approximate the prior distribution, which is a Gaussian). This is often called **Regularization Term** because it **forces the latent space to fit a Gaussian distribution**.

The combination of these two factors allows the latent space to be continuous and well-structured, ensuring smooth interpolation between latent representations without gaps. This forces the encoder to produce a Gaussian distribution that approaches the prior.

The figure compares three different latent spaces obtained by various autoencoder training strategies.

The latent space on the left is the result of **exclusive use of reconstruction loss**, as in traditional autoencoders. In this case, no constraint is applied on the latent space distribution. As can be seen, the latent space has no well-defined structure and the latent points are not distributed according to a centered Gaussian. In other words, the latent data appear scattered and disorganized, with no clear structure. *Indeed, autoencoders are not generative models by nature, since there are no constraints on the latent space to guarantee its structured behavior. If we train an autoencoder, take the latent space and add noise, the decoder will produce meaningless results.*

The second latent space, located in the center of the figure, is obtained by using **only**



Only Reconstruction Term | Only Regularization Term | Both

Latent Spaces Comparison

the regularization term. In this scenario, the model learns a narrower distribution for each sample type, trying to fit a Gaussian shape. However, as highlighted in the figure, although the distribution assumes a Gaussian shape, the latent points tend to overlap and mix. This makes it difficult to distinguish the different data sets, as the latent data are not well separated.

Finally, the latent space on the right represents the result of **joint optimization of both terms: the reconstruction loss and the regularization term**. This combination allows the latent space to fit the data well, being continuous and similar to a Gaussian distribution. As a result, the latent space presents a uniform and continuous structure, with the data distinct in well-defined clusters, exactly what we want to achieve! 😎

Now, **to train VAEs we want to use backpropagation, but in practice this is impossible** (😔) because sampling from the latent distribution $z \sim q_\phi(z|x)$ is not differentiable. *However, the* ***reparametrization trick*** *comes to our rescue!* This trick is to sample from a standard normal distribution $\epsilon \sim \mathcal{N}(0, I)$ and transform it with $z = \mu(x) + \sigma(x) \cdot \epsilon$. So, **the sampling process becomes differentiable with respect to the model parameters**, facilitating the training of latent representations. 😎 (again)

9.9. VQ-VAES

Typically, in a classic VAE, the prior ($p_\theta(z)$) and posterior ($q_\phi(z|x)$) are assumed to be normally distributed with diagonal variance. In the proposed work on VQ-VAEs **"Neural Discrete Representation Learning" (van den Oord et al.)**, however, the authors use **discrete latent variables** (instead of a continuous normal distribution). The posterior and prior distributions are categorical, and **samples drawn from these distributions index a embedding table**. Then, the encoders model a **categorical distribution**, from which we sample to obtain integral values. These integral values are used to index an embedding dictionary, and the indexed values are then passed to the decoder.

For example, in images we may have categories such as "Dog" or "Bicycle," and it would not make sense to mix these categories. Discrete representations are also easier to handle, since each category has a unique value. In contrast, in a continuous latent space, it would be necessary to normalize the density function and understand the relationships between the various variables, which can be very complicated.

9.10. GAN AND VAE COMPARISON

	GANs	VAEs
Efficient sampling	✓	✓
High quality	✓	×
Coverage	×	?
Well-behaved latent space	✓	✓
Interpretable latent space	?	?
Efficient likelihood	n/a	×

GAN and VAE Comparison

Comparison of GAN and VAE using the Ideal Properties of Generative Models

High-Quality Sampling: GANs are capable of generating high-quality samples, meaning the samples they produce are often indistinguishable from real data.

Coverage: GANs often struggle with coverage, meaning they might generate samples that represent only a subset of the training data, rather than covering the entire distribution. VAEs generally aim to cover the full distribution, though their success in this area can vary.

Well-Behaved Latent Space: Both models have well-behaved latent spaces. In other words, each latent variable z corresponds to a realistic data example x, and smooth changes in z lead to smooth changes in x.

Interpretable Latent Space: Neither GANs nor VAEs excel at providing an interpretable latent space. Ideally, altering each dimension of z should affect a specific, understandable feature of the data, but neither model achieves this effectively.

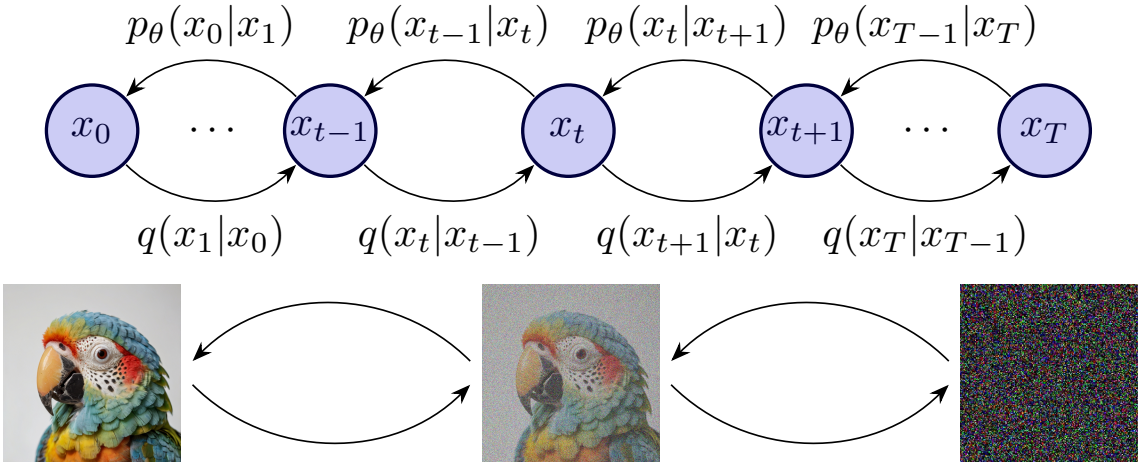
Efficient Likelihood Computation: VAEs, despite being probabilistic models, struggle with efficiently computing likelihoods. GANs do not use likelihoods, so this characteristic does not apply to them.

10. DIFFUSION MODELS

Diffusion Models, introduced in 2015 in the paper **"Deep Unsupervised Learning using Nonequilibrium Thermodynamics" (Sohl-Dickstein et al.)**, are generative models invented at the same time as GANs, but they gained popularity only in 2020. This delay is due to several factors, including computational complexity and the need for more powerful hardware, which has only recently become more accessible and widespread. We now delve into how these models work.

The training of diffusion models can be divided into two main steps:

1. **Forward Diffusion Process ($\mathbf{q(x_t|x_{t-1})}$)**: the contents of an input image are "destroyed" by adding noise over time t until the data distribution becomes a Gaussian.
2. **Reverse Diffusion Process ($\mathbf{p_\theta(x_{t-1}|x_t)}$)**: remove noise from the image in order to reconstruct the original image from a Gaussian distribution of noise.



Diffusion Model Idea

10.1. FORWARD PASS

To approximate a Gaussian distribution, the added noise is increased slightly at each step (t). To obtain a better approximation, the direct diffusion procedure requires **many iterations** (T). However, this involves a trade-off: more steps lead to a better approximation but also increase the computational time required. Therefore, the selection of the number of steps is crucial, since it affects both the quality of the approximation and the computational resources required.

The direct diffusion process gradually adds Gaussian noise to the input image x_0 step by step, and there will be T steps in total. The process will produce a sequence of noisy image samples $x_1, \ldots, x_T$. **All intermediate steps are Gaussianly distributed**:

$$q(x_t|x_{t-1}) = \mathcal{N}(\sqrt{1-\beta_t}x_{t-1}, \beta_t I)$$

β_t represents the **(small) increase in the noise variance at each time interval** t. The formula shows that, at each time, the new Gaussian obtained is centered on the previous

sample and rescaled by the variance of the added noise β_t. **This makes it possible to approach a zero mean**. If β increases, less data are retained when the final Gaussian is obtained ($\sqrt{1-\beta_t}$), and more and more noise is added ($\beta_t I$).

Defining $\alpha_t = (1-\beta_t)$, we can rewrite the formula as:

$$\mathbf{q(x_t|x_{t-1})} = \mathcal{N}(\sqrt{\alpha_t}x_{t-1}, (1-\alpha_t)I)$$

When we sample all steps sequentially, we need to calculate the product of all intermediate random variables:

$$q(x_{1:T}|x_0) = \prod_{t=1}^{T} q(x_t|x_{t-1})$$

Instead of designing an algorithm to compute this iteratively, we can use a **closed-form formula for a Gaussian process** to directly sample a noisy image (an intermediate step) at a specific time interval t using the **reparameterization trick**, which we have also seen in VAE:

$$x_t = \sqrt{\alpha_t}x_{t-1} + \sqrt{(1-\alpha_t)}\epsilon_{t-1}$$

Where $\epsilon \sim \mathcal{N}(0,1)$, that is, a standard Gaussian random variable.

Then we can expand it recursively by substituting x_{t-1} until we get x_0, to get the formula in closed form (*many mathematical derivations have been omitted for your sanity*):

$$x_t = \sqrt{\overline{\alpha}_t}x_0 + \sqrt{1-\overline{\alpha}_t}\epsilon_0^*$$

Where $\overline{\alpha}_t = \prod_{t=1}^{T}\alpha_t$ and $x_t \sim \mathcal{N}(\sqrt{\overline{\alpha}}_t x_0, (1-\overline{\alpha}_t)I)$.

As we can see, the final formula depends only on the input image x_0. This has a significant implication: **it allows us to directly sample x_t at any time interval**, making the direct diffusion process much faster since we do not have to compute the entire product of all time intervals. The significance is that we can sample any intermediate x_t simply by multiplying all α_t along the way. **As t gets closer to T, i.e., after many steps, we eventually get an (almost certainly) Gaussian distribution.**

10.2. REVERSE PASS

So far, we have seen how to destroy all image information using efficient transformations, but this procedure **does not involve any training**. What we lack is an understanding of how the model can generate a sample from this final Gaussian distribution. In particular, *how does the model learn to reverse the process and transform the noise back into the original data distribution?*

Unlike the forward process, we cannot use $p(x_{t-1}|x_t)$ to reverse the noise because the **denominator** $p(x_t)$ and the **marginal distribution of an intermediate step** $p(x_{t-1})$ are **intractable**:

$$p(x_{t-1}|x_t) = \frac{p(x_t|x_{t-1})p(x_{t-1})}{p(x_t)}$$

Therefore, we need to train a neural network $p_\theta(x_{t-1}|x_t)$ to approximate the inverse process $p(x_{t-1}|x_t)$, i.e., to approximate the posterior distribution (*...sounds familiar?*). The approximation $p_\theta(x_{t-1}|x_t)$ follows a normal distribution (**remember that the inverse of a Gaussian process is still a Gaussian distribution**), and the **neural network learns/models only the parameter μ_θ of the inverse Gaussian process**, since the **variance is fixed**, i.e., it is shared among the intermediate inverse steps:

$$\mathbf{p_\theta(x_{t-1}|x_t)} = \mathcal{N}(\mu_\theta(x_t, t), \Sigma_t)$$

> As you may have already realized, diffusion models can be seen as a **special case of (hierarchical) VAE**, but:
> - The encoder is fixed (it is the forward process).
> - The latent variables have the same dimensionality as the data.
> - The same model is applied to different time intervals.

With this approach, instead of directly optimizing the intractable loss function, we can **optimize the Variational Lower Bound**. Eventually, the neural network can be trained with **ELBO (Evidence Lower Bound)**:

$$\mathcal{L} = \mathbb{E}_{q(x_1|x_0)}\left[\log p_\theta(x_0|x_1)\right] - \sum_{t=2}^{T} \mathbb{E}_{q(x_t|x_0)}\left[D_{KL}\left[q(x_{t-1}|x_t, x_0) \| p_\theta(x_{t-1}|x_t)\right]\right]$$

The first term represents a **boundary condition** and the second term is a **KL divergence at every other step**.

Using this formula, we perform some mathematical steps and derivations: calculate the closed form for the second term, simplify the closed form by focusing directly on the divergence in the added noise, rewrite the first term by defining it through the added noise, define the lower bound of the log-verosimilarity of a sample, and use the Monte Carlo approximation to not compute all the time steps of all the samples. *Fortunately for you, we will not see all these steps in detail but only the final closed formula that is used for the inverse process.*

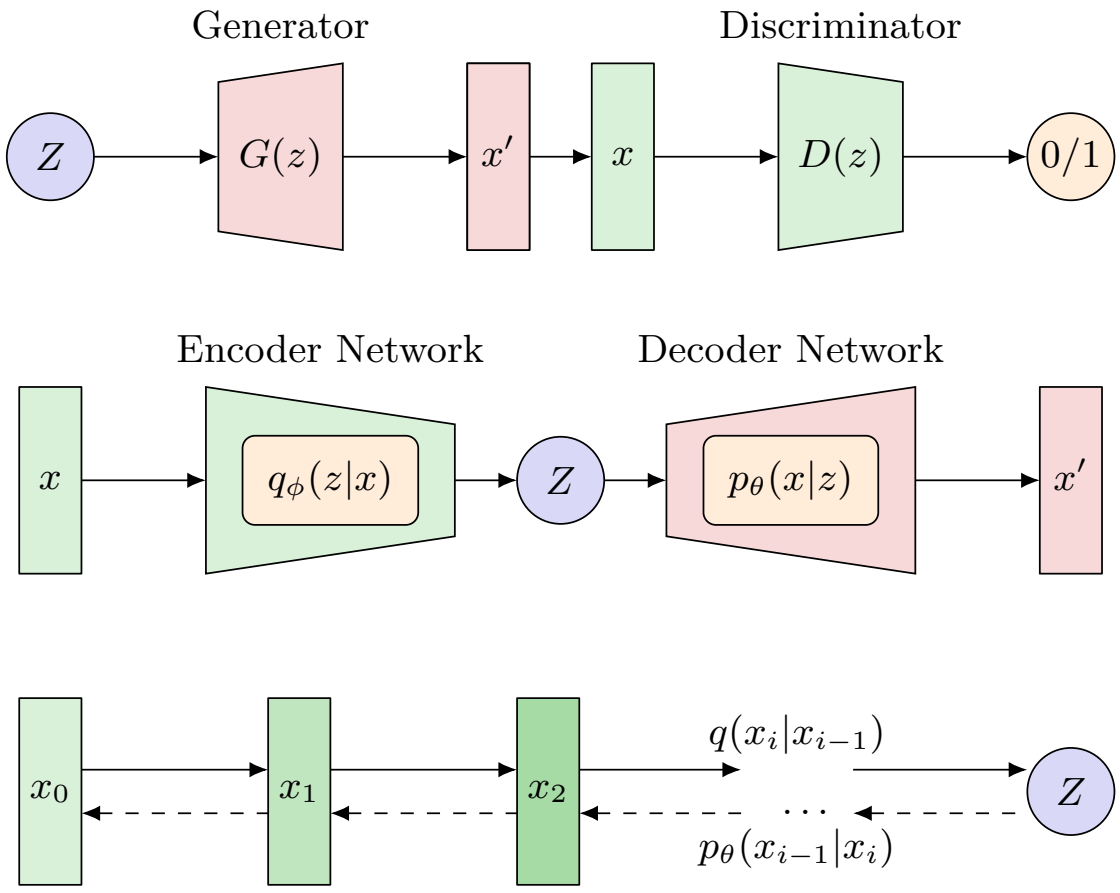
The inverse process, which removes the added noise, is described by the following closed formula (*oh, of course we are also using the reparameterization trick*):

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\overline{\alpha}_t}}\epsilon_\theta(x_t, t)\right) + \sigma_t z$$

We have finally obtained an efficient formula that achieves the training goal: **an iterative process that allows the model to transform noisy data into clean data by progressively removing estimated noise**.

10.3. GENERATIVE TRILEMMA

So far, we have examined three types of generative models: the GANs, VAEs, and diffusion models. Below is an image that briefly summarizes their different architectures:

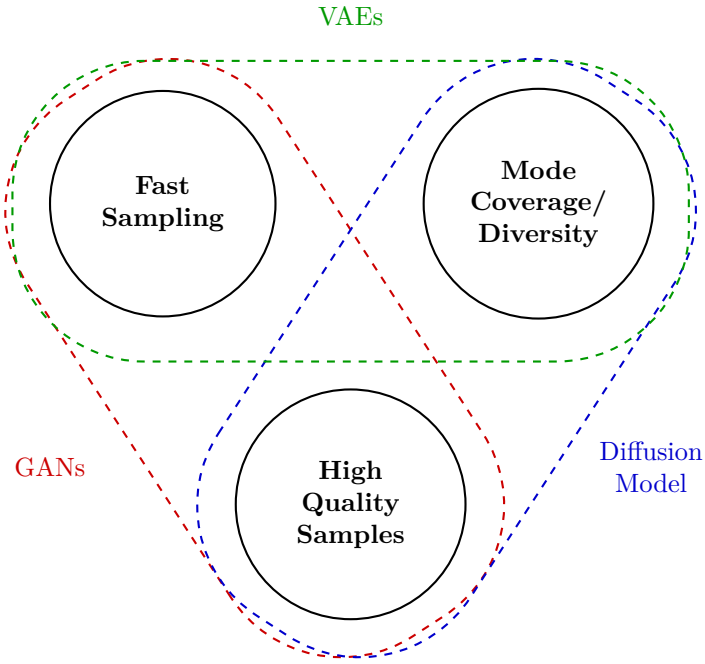


Generative Models Comparison

In NVIDIA's paper **"Tackling the Generative Learning Trilemma with Denoising Diffusion GANs" (Vahdat et al.)**, the authors state that, in the context of generative models, there is a trilemma based on three properties: **Fast Sampling**, **Diversity** (Mode Coverage) and **High Sample Quality**. Interestingly, our three types of generative models excel in different combinations of these properties:

- **VAEs**: Fast Sampling and Diversity.
- **GANs**: Fast Sampling and high Quality.
- **Diffusion Models**: High Quality and Diversity.

Recently, diffusion models have made significant progress in fast sampling, trying to balance all three properties.



Generative Trilemma

10.4. STOCHASTIC EQUATION AND DENOISING SCORE MATCHING

In this very short section we will give an high-level overview two fundamental aspects of training diffusion models.

In the forward diffusion process, noise is added to the original image at each step. This process can be described by a **stochastic equation that separates the effect of noise into two components**: one representing **average change** and one representing **random variation**. For the reverse process, which aims to reconstruct the original image, these two components are used to guide the model in removing the noise.

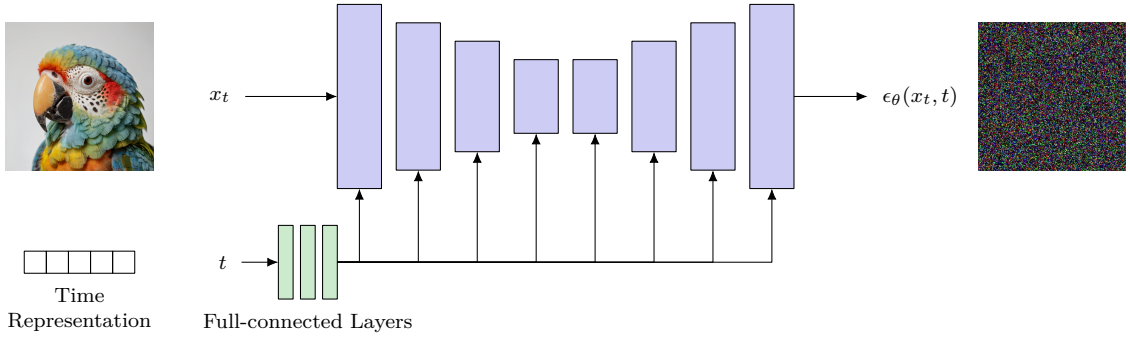
Training a diffusion model is based on **Denoising Score Matching**, which aims to minimize the difference between the noise added and the noise estimated by the neural network. This approach allows the model to refine its ability to interpolate between latent spaces, resulting in continuous and semantically meaningful changes in the data.

10.5. NETWORK ARCHITECTURE AND TEXT CONDITIONING

The network architecture used in diffusion models can vary, but is often inspired by established models such as PIX2PIX. In this configuration, **temporal information is embedded via Fourier features**, which can be represented by sinusoidal (sine and cosine) functions, similar to positional embedding. These features help to encode time more expressively.

For images, the **U-Net architecture, with the addition of attention mechanisms, is commonly employed to preserve and reconstruct complex spatial details**. In addition, Vision Transformers (ViTs) can be used to exploit self-attention mechanisms, enhancing the model's ability to capture complex patterns and relationships in images.

The choice of how to incorporate temporal features and the underlying network is relatively free, with no stringent theoretical constraints.



Diffusion Model Architecture Example

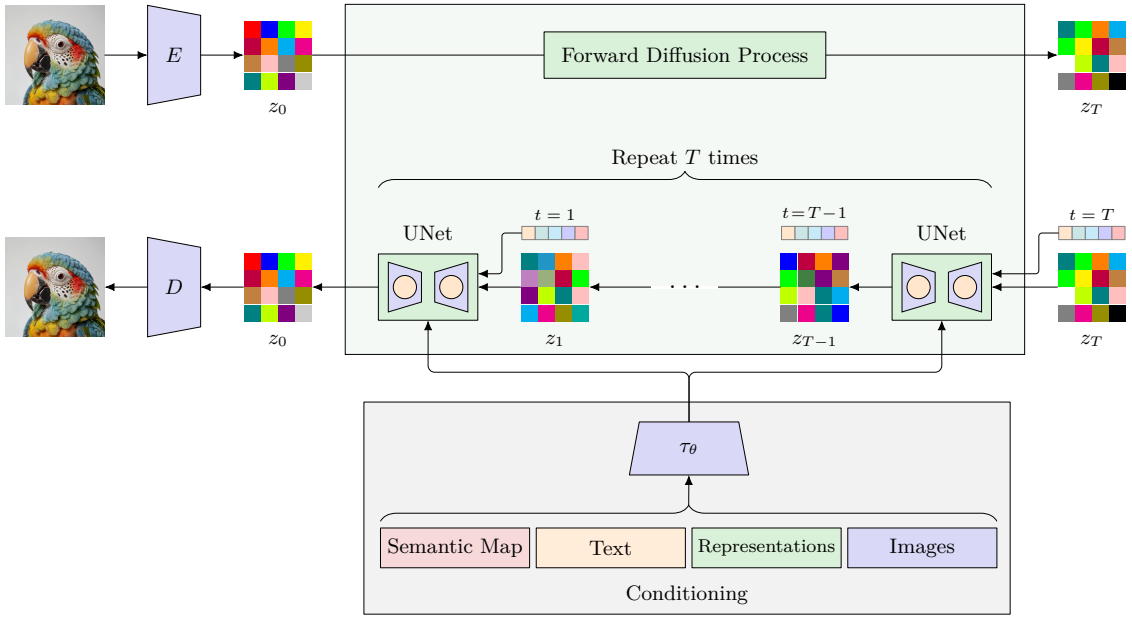
But how can we guide diffusion models to generate text-based images? The **Text Conditioning** in diffusion models uses textual information to influence image generation. Here is a (highly synthesized) summary of the process:

- **Textual Embeddings:** Textual descriptions are converted into numerical vectors using embedding models such as CLIP, which capture the meaning of the text. Textual embeddings are integrated into the diffusion model, guiding the generation of the image to reflect the provided description.
- **Training:** The model is trained on text and image pairs, learning to generate images that match the text descriptions. The goal is to minimize the difference between the generated image and the target image based on the text.

10.6. STABLE DIFFUSION

At the end of this chapter on diffusion models, it is important to mention a significant advance in the field: **Stable Diffusion**, described in the article **"High-Resolution Image Synthesis with Latent Diffusion Models" (Rombach et al.)**. This approach represents an evolution from traditional diffusion models, introducing some key innovations. Here are the main ones:

- **Latent Space:** Instead of applying the diffusion process directly on high-resolution images, Stable Diffusion operates in a compressed latent space. This latent space is represented by a **low-dimensional encoding of the images**, obtained through an encoder. Diffusion takes place on these latent representations, significantly **reducing computational costs and making the process more efficient**. Next, a decoder reconstructs the original images from the latent space.
- **Denoising U-Net:** In the reverse process, Stable Diffusion uses a U-Net for denoising. This network is designed to remove noise from latent representations and restore the original images with high quality. The U-Net model is **equipped with attention mechanisms** that enhance the model's ability to capture and reconstruct complex image details.
- **Conditioning Space:** Stable Diffusion can also include additional conditions, such as text or other semantic information, to **guide image generation**. This conditioning space allows images to be tailored to specific textual descriptions or other constraints, increasing the model's versatility in synthesizing visual content.



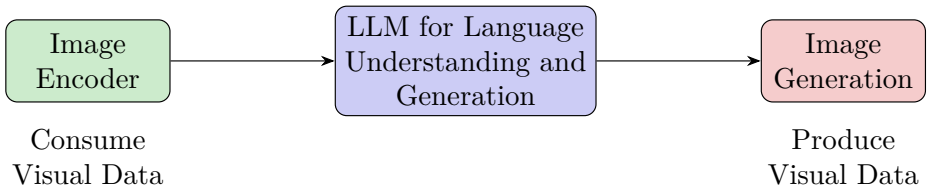
Stable Diffusion Architecture

This approach enables practical challenges in high-resolution image synthesis, making visual content generation more affordable and scalable.

11. LARGE MULTI-MODAL MODELS

In this chapter we will look at **Large Multimodal Models** (LMMs). No, *you heard me right*, we will not be looking at **Large Language Models** (LLMs), in which the data modality is primarily textual, but at Large Multimodal Models, which can handle multiple types of input data, including text, images, audio, video, and sometimes other types of data such as sensory data.

We will look at **multimodal architectures**, which are computational models designed to process and integrate data from different modalities or input types. To get to LMMs, we will look at **Large Vision-Language Models**, focusing on the *Image Encoder* in a multimodal setup. Using this setup, we aim to create networks capable of consuming different types of input to learn more robust visual representations.



Example of Input Data: (Image + Text)

Large Vision-Language Models Idea

As you can see, the goal is to create an encoder that learns a representation of our data (pairs of images and text), and then use an LLM as a frontend for generation.

Currently, all methods for training an encoder share the same goal: to **build a robust model capable of learning robust representations of data**. Let us now look at the four main methods of training an encoder:

- **Supervised Learning**: This is the first approach we used to train encoders, in which models are trained using large amounts of manually annotated data. Although this method can produce excellent results, it has a significant bottleneck due to the high cost and time required for human annotation.
- **Image-Only (Non-) Contrastive Learning**: This approach exploits methods such as the DINO (DIstillation with NO labels) model, which allows encoders to be trained using images only, without the need for labels. Through contrastive learning techniques, the model learns to distinguish between similar and different representations.
- **Masked Image Modelling**: Inspired by the invention of the BERT model for language, this method applies a similar concept to images. Some parts of the image are "masked" and the model's task is to predict the missing parts. This process forces the encoder to understand the visual context and develop a robust internal representation.
- **Contrastive Language-Image Pre-Training**: Using the CLIP (Contrastive Language-Image Pre-Training) model developed by OpenAI and its variants, this approach involves training an encoder that can understand both language and images. The model is trained

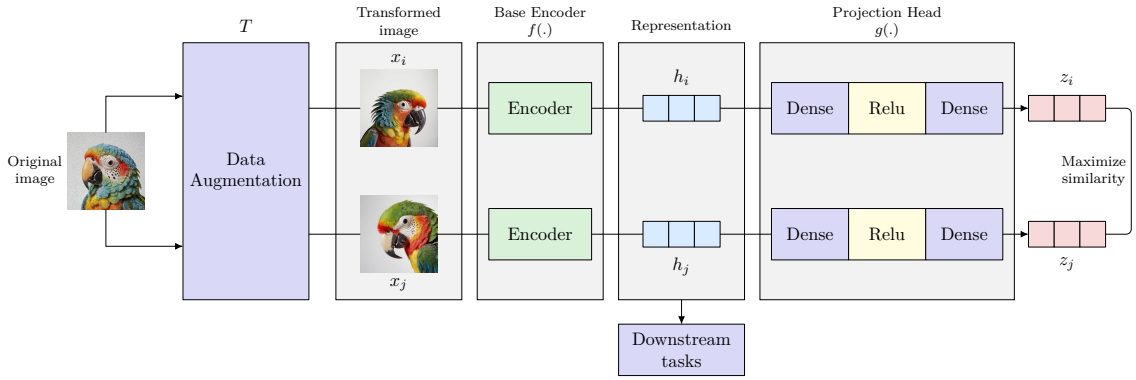
to correlate images and corresponding text descriptions, allowing it to create visual representations that are semantically aligned with the text. This method has shown outstanding results in a variety of multimodal tasks and has opened up new possibilities for the integration of different data modalities.

In the next sections, we will examine these approaches in detail.

11.1. IMAGE-ONLY (NON-) CONTRASTIVE LEARNING

11.1.1. SimCLR

The first alternative to the previous class of models that emerged was **SimCLR**, introduced in the article **"A Simple Framework for Contrastive Learning of Visual Representations" (Chen et al.)**.



SimCLR architecture

Here's how it works: given an image, **two different data augmentations** are applied. Each augmented image is then processed by an encoder, producing corresponding fixed latent representation vectors. These vectors are subsequently passed through a **projection head**, which maps them to another latent space, resulting in the vectors z_i and z_j.

Each module of the projection head consists of a sequence of fully connected layers, ReLU activations and a final dense layer. The vectors z_i and z_j are used in the **contrast loss function**, specifically **InfoNCE**, to perform training. The goal is to maximize the agreement between augmentations by determining whether they come from the same image or not.

This approach is **contrastive** because we impose two constraints on the model parameter space: **positive examples will have similar representations while negative examples will have very dissimilar representations**.

In this case, **projection head is discarded for downstream tasks**, since the best representations are h_i and h_j instead of z_i and z_j. However, this model requires a large batch size to ensure a sufficient number of negative samples for each class.

Recent research has addressed this problem using the principle that the representations of a sample and its transformation should be similar. Alternatives include **DINO**, which uses a method based on **clustering**.

11.1.2. DINO

The article **"Emerging Properties in Self-Supervised Vision Transformers" (Caron et al.)** introduces DINO (Distillation with No Labels), which is a Vision Transformer model that employs self-supervised loss. It is widely recognized for its ability to compute segmentation maps that emerge from the transformer's self-attention modules.

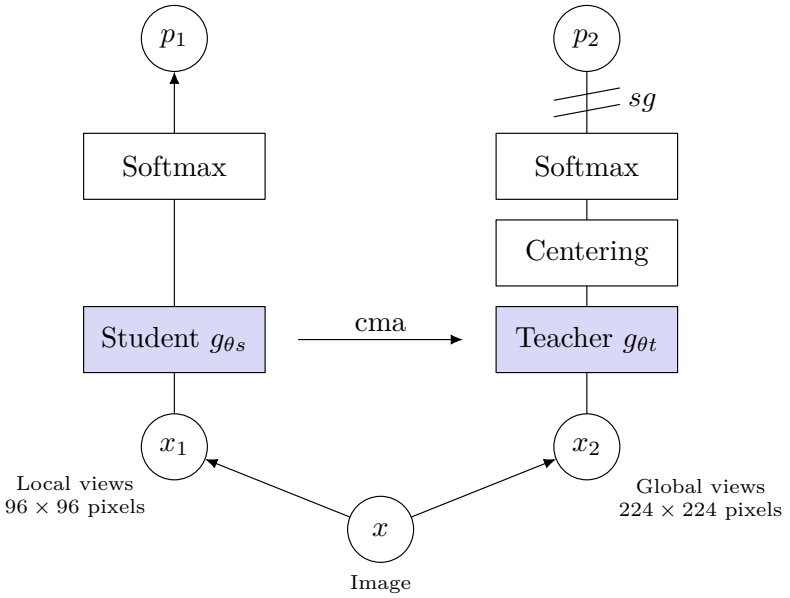
The architecture uses a **teacher-student** framework with a **cross-entropy loss**. The student network extracts the parameters of interest, and since DINO is part of the clustering approach, the output of the student network goes through a softmax function to determine whether an image belongs to a certain cluster.

This produces a probability distribution, allowing training with a **self-supervised cross-entropy loss**. The teacher network is updated using the **exponential moving average (EMA)** of the parameters learned from the student network.

To ensure that the teacher makes accurate predictions for a given class, centering and softmax layers are added. The centering layer subtracts the average feature and the output is refined using a low temperature softmax.

The gradient is propagated only through the student network, allowing centering to be used on the teacher model.

One important thing is that the student and the teacher have the same architecture (ViT or RN50), but a different set of weights. Also, the gradients only propagate loss through student (sg = stop gradient).



DINO Architecture

Thus, clustering tends to solve the batch size problem of SimCLR **learning a powerful visual encoder through self-supervised loss**.

11.2. MASKED IMAGE MODELLING

In this learning class, models are trained to predict a missing part of the input (masked) given the other (unmasked) information.

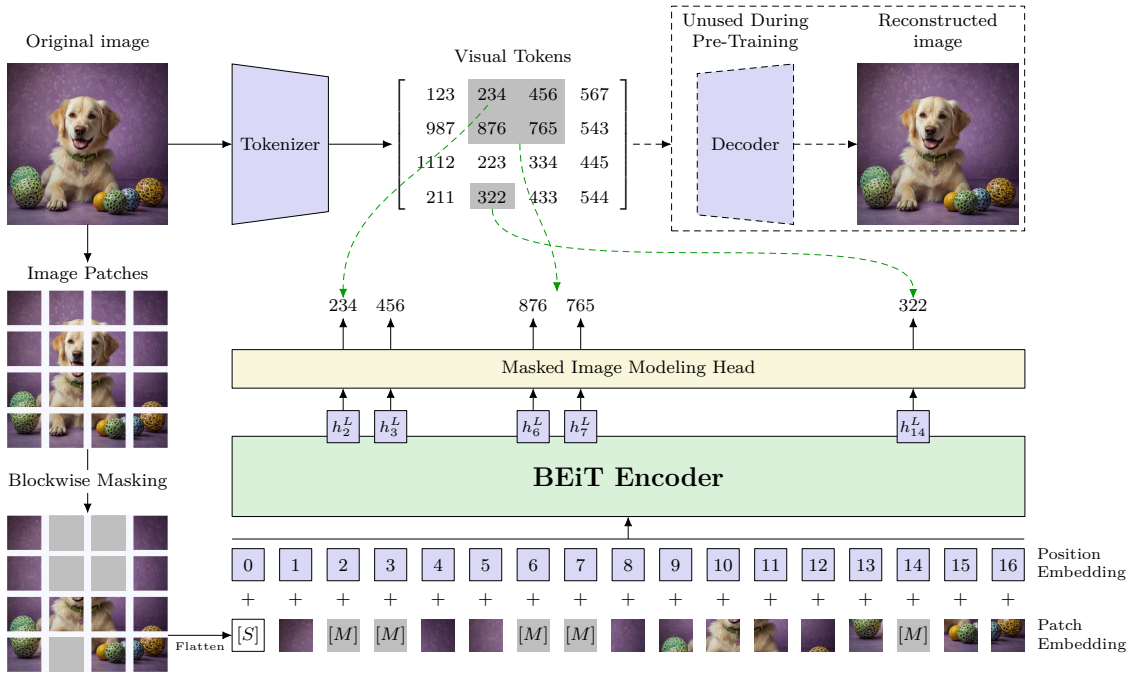
11.2.1. BEiT

Following the BERT model developed in the area of natural language processing, Microsoft proposed a masked image modelling task to pre-train Vision Transformers in the article **"BEiT: BERT Pre-Training of Image Transformers" (Dong et al.)**.

The model works by taking an image and dividing it into patches. Next, **some of these patches are masked and all patches are flattened into a vector**. This vector becomes the input (plus positional embedding) of the **encoder BEiT**, which is pre-trained similarly to the BERT model in natural language processing. Only embeddings related to the masked tokens are then retrieved. The chosen embeddings are then passed as input to an additional encoder, the **Masked Image Modelling Head**. This represents what we ultimately want to learn. To do this, we employ supervised help.

To facilitate this process, an additional network is used. Before pre-training, an "image tokenizer" is learned through VQ-VAEs or GANs, which **tokenizes an image into discrete visual tokens, which act as supervisors for the encoder**. The Masked Image Modelling Head can thus be seen as a form of **knowledge distillation** (Process in which a smaller model, the "student", learns to replicate the behavior of a larger, more complex model, the "teacher") between the image tokenizer and the BEiT encoder, with the key difference being that the BEiT encoder sees only part of the image.

In the end, we use a pre-trained decoder to predict the masked visual tokens.



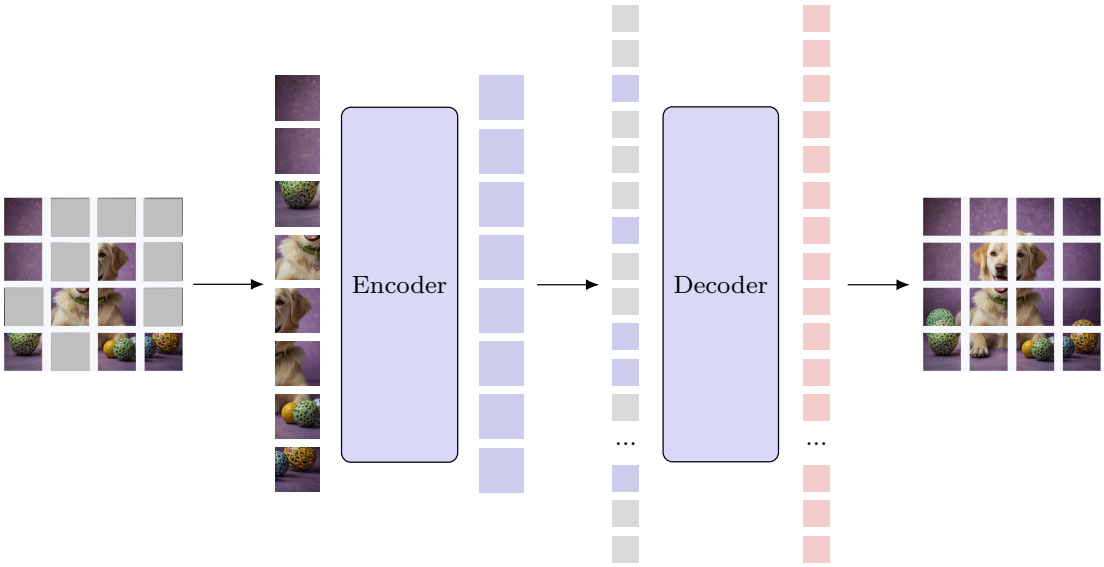
BEiT Architecture

The model works rather well with fine-tuning rather than pre-training. Of course, one reason related to this fact is the complexity of the model. As a result, variants have been

developed.

11.2.2. MAE

MAE is a variant of BEiT introduced by Meta in the article **"Masked Autoencoders Are Scalable Vision Learners" (He et al.)**. It uses **pixel values as targets** instead of visual tokens. The model can be very aggressive in masking patches, since a large random subset of the images (for example 75%) can be masked. Thus, the encoder is trained to **produce representations of only the visible patches**. Before the decoder, masked patches are remapped with latent representations, and then the decoder reconstructs the original image.



MAE Architecture

11.3. CONTRASTIVE LANGUAGE-IMAGE PRE-TRAINING

We have come to the most fascinating and juicy part of this chapter. These types of models represent the current state of the art.

11.3.1. CLIP

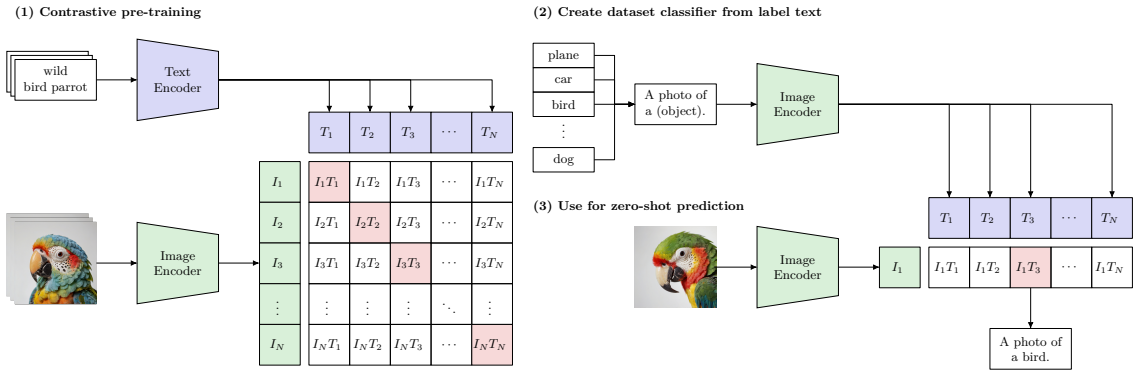
CLIP combines two main neural networks:

- **The Image Encoder**: It uses a Vision Transformer based network or deep convolutional network to convert images into embedding vectors.
- **The Text Encoder**: Employs a Transformer-type neural network, similar to that used in language models such as GPT, to convert texts into embedding vectors.

The main goal of CLIP is to create a **shared embedding space in which images and text descriptions are semantically aligned**. During training, the model uses a **contrastive learning** technique to optimize the similarity between image embeddings and corresponding text descriptions, while minimizing the similarity between images and

unrelated text. This approach allows CLIP to make inferences and classifications based on text descriptions, without the need to train task-specific models.

CLIP was trained on a large and diverse dataset containing approximately **400 million image-text pairs taken from the Internet**. Although this dataset is sizable, even larger datasets, such as LAION, containing billions of images, are now also being used.



CLIP Architecture

In essence, CLIP aims to maximize the similarity between pairs of images and texts, corresponding to **the diagonal of the similarity matrix** you see in the figure. At the same time, it tries to minimize the similarity (increase dissimilarity) between all other pairs, that is, between unrelated images and texts.

CLIP's capabilities extend to numerous applications, called **downstream tasks**, such as image classification and search: it can be used to search for images based on text descriptions and vice versa. A notable aspect of CLIP is its **zero-shot learning** capability, which allows it to generalize to new categories without the need for additional training specific to those categories.

However, CLIP has some limitations:

1. **Limited Capability**: CLIP is limited to connecting image and text embedding spaces.
2. **Use Restricted to Specific Tasks**: Although it excels at classification and searching, CLIP is less suitable for more complex tasks such as text generation or visual question answering.
3. **Language Generation Capability Absence**: CLIP lacks the ability to generate language, making it less suitable for open-ended tasks such as caption generation or visual question answering.

So, how can we improve the model? We can make improvements in three main areas: **Data Scalability**, **Model Design** (for both image and text encoder) and **Objective Functions** (contrastive learning).

In the next subsections we will explore in detail the various improvements proposed for CLIP and see some models that implement these ideas.

11.3.2. Data Scaling Improvement

"Reproducible scaling laws for contrastive language-image learning" (Jitsev et al.) studied the use of pre-trained CLIP with the LAION-2B dataset at different scales, including models and data. The main message is that **scale of dataset is crucial**.

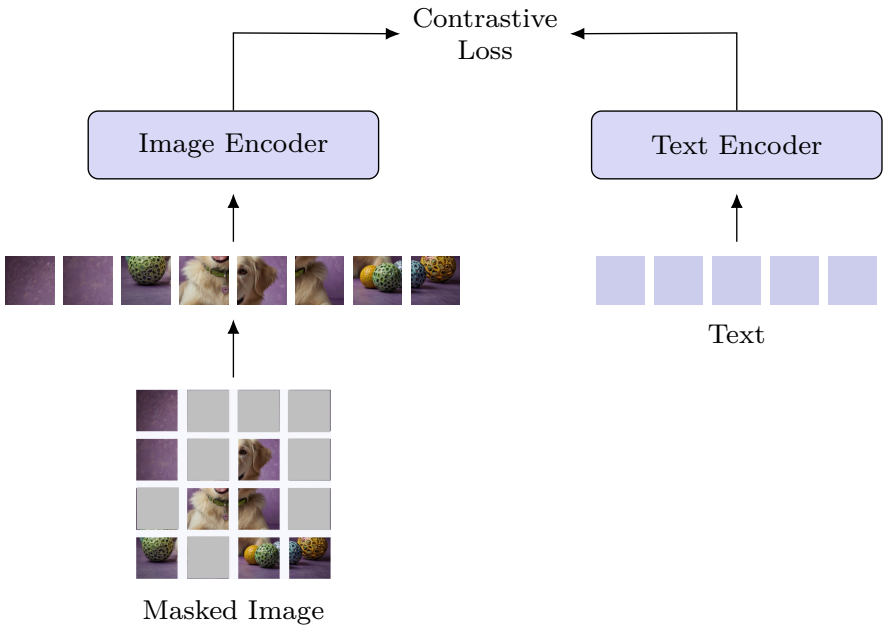
But how can we scale further? One possible solution is to search for next-generation image-text datasets. However, in **"DataComp: In search of the next generation of multimodal datasets" (Ilharco et al.)**, researchers found that **filtering the data** provided to the model leads to better results even with smaller datasets.

11.3.3. Image Model Design Improvement: FLIP

A Model Design Improvement for the image encoder is **FLIP**, introduced in , which tried to scale CLIP training (more efficient) via randomly masking out image patches with a high masking ratio. This allows the image encoder to process just the non-masked patches. Thus, during training we still use CLIP loss, but no reconstruction of masked patches is made.

A significant improvement in model design for the image encoder is FLIP, introduced in **"Scaling Language-Image Pre-training via Masking" (Li et al.)**. This approach aims to make CLIP training more efficient by **randomly masking image patches with a high masking ratio**. This allows the image encoder to process only the unmasked patches. During training, CLIP loss is still used, but no reconstruction of masked patches is performed.

Results have shown that this approach not only does not compromise performance, but **also improves training efficiency**.



FLIP Architecture

11.3.4. Language Model Design Improvement: K-LITE

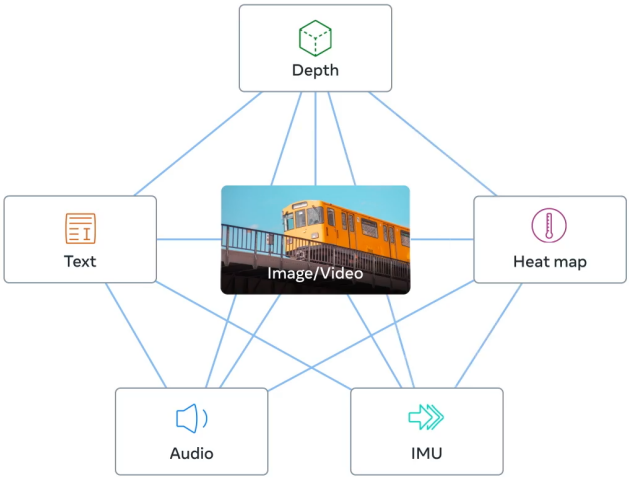
Because classes are very specific, the work **"K-LITE: Learning Transferable Visual Models with External Knowledge" (Shen et al.)** introduces a mechanism to extend text knowledge via **external knowledge** in order to provide more detailed information. For example, WordNet or Wikipedia can be used. This is easily accomplished since the text encoder can be fed with input of arbitrary length (original text + external knowledge).

The mechanism of knowledge expansion takes place in two stages: during training, the **model acquires the ability to read and understand a specific knowledge source**. During evaluation, the knowledge provides an **additional source of information to improve model inference**.

Results showed that f**or datasets containing specific images**, such as the Flowers dataset, **knowledge expansion led to improvements in performance**, as the additional knowledge helped to better discriminate between classes. However, for datasets such as Eurosat, the approach performed poorly because the additional knowledge retrieved was irrelevant to the image.

11.3.5. Model Design Improvement: Multi Modalities

In this case, improvement is achieved by adding more modes than just the traditional two (such as audio, video, etc.). For example, in the study **"ImageBind: One Embedding Space To Bind Them All" (Girdhar et al.)**, Meta introduced ImageBind, a model that uses one of the seven available modes as an anchor (images) and aligns the other modes to this key mode. In this way, all modes are **linked to a common space**. A pre-trained CLIP is used and kept frozen, i.e., only the encoders for the other modes are learned to align the CLIP embedding space.



ImageBind Idea of Shared Embedding Space

The model employs pairs of modalities (I,M), where I signifies images and M is another modality, to learn a **single joint embedding**. Each modality's embedding is aligned with image embeddings, for instance, text is aligned with images using web data. The embeddings and encoders are optimized using an **InfoNCE loss**.

The model uses mode pairs (I, M), where I represents images and M is another mode, to learn a **unique joint embedding**, indeed every embedding is aligned with image embeddings. The embeddings and encoders are optimized using a **InfoNCE (Information Noise-contrastive Estimation) loss**.

ImageBind showed an **emergent behavior**: the model is able to align two different modes (M_1, M_2) even though it was trained only with pairs (I, M_1) and (I, M_2). For example, it obtains state-of-the-art results in zero-shot text-audio classification without having been exposed to paired audio-text samples. Hence, given an audio of the sound of a fire, the

model can generate an image or video of the fire.

11.3.6. Model Design Improvement: STAIR

One of the limitations of CLIP is the lack of interpretability in the image embedding space. However, thanks to **"STAIR: Learning Sparse Text and Image Representation in Grounded Tokens" (Zhang et al.)**, it is possible to achieve better performance than CLIP with a significant improvement in interpretability. Instead of using dense, uninterpretable representations, STAIR creates a **sparse, semantic representation of images and texts**.

STAIR is **based on a large vocabulary of words and phrases**, known as a "dictionary of tokens". Each word in the dictionary is considered a unique token, representing a specific concept or term.

For each image or text, STAIR constructs an embedding space in which **each dimension of the vector corresponds to a vocabulary word**. In other words, images and texts are represented as vectors scattered throughout this space, reflecting which and how many tokens are present and relevant.

Each token in the dictionary is **associated with a non-negative scalar value**, indicating the token's importance to the image or text in question. Finally, STAIR uses a unit called the **Token Projection Head** to project representations into its sparse embedding space.

Thanks to this method, we can observe the relevance of each word in the embedding space. In the figure below, we consider the first 20 tokens predicted by STAIR for an image and represent them graphically with a font size indicating the prediction weight.

STAIR Results

11.3.7. Objective Functions Improvement: FILIP

As we have stated, another way of improving clip is to improve the objective function and FILIP, introduced in **"FILIP: Fine-grained Interactive Language-Image Pre-Training" (Yao et al.)**, belongs to this category.

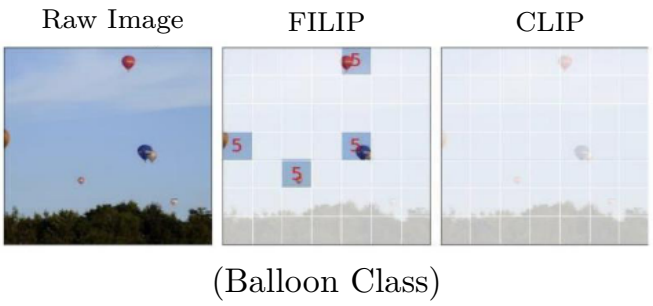
FILIP uses two main encoders:

- **Visual Encoder**: Based on Vision Transformer (ViT), this encoder takes as input images divided into patches. A special token `[CLS]`, representing the image as a whole, is added to these patches. The patches and the `[CLS]` token are linearly projected and then processed by the model.
- **Text Encoder**: After embedding the words, the text is processed by a Transformer-only decoder. This model handles the token sequences generated by the words in the text.

The objective function of FILIP is fine-grained alignment:

- **Token-wise similarity**: FILIP calculates the **similarity between each token in the text and each patch in the image**. This process allows you to see how each word in the text aligns with different parts of the image.
- **Pooling and Aggregation**: After calculating the similarity matrix between tokens and patches, FILIP uses max pooling to **aggregate these similarities**, making the representations more useful for later tasks.

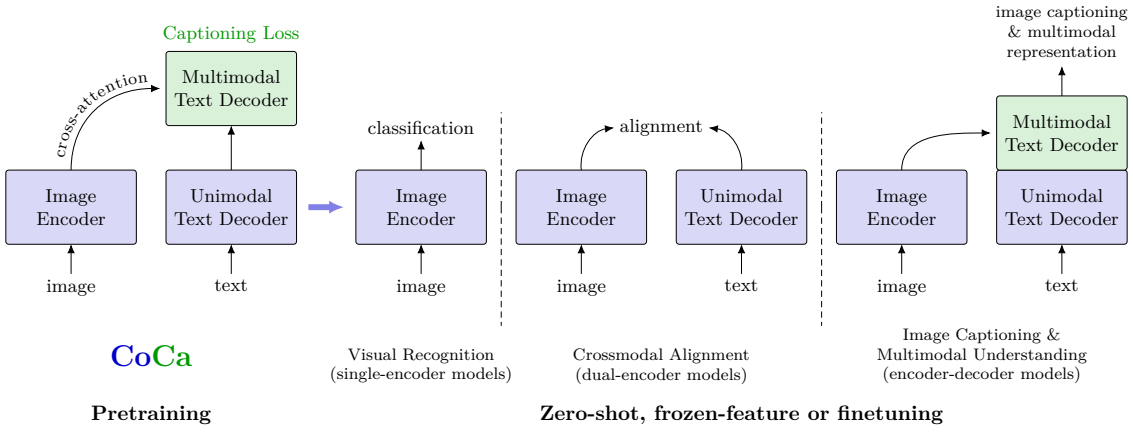
FILIP is designed to learn detailed alignment between words and image patches, improving visual and textual understanding. The image below shows an example with the class label "balloon" (in this case class 5), which is entered into the prompt "`[BOS]` a photo of a balloon. `[EOS]`" and tokenized into a sequence of 8 tokens. **The model learns the corresponding tokens that are activated in the patches of the image where the class is visible.**



An Example of Word-patch Visualization with FILIP

11.3.8. Objective Functions Improvement: CoCa

The CoCa model, introduced in the paper by Google **"CoCa: Contrastive Captioners are Image-Text Foundation Models" (Yu et al.)**, represents a significant advance in the integration of contrastive learning with language generation. CoCa enhances CLIP by adding a **generative branch** (Captioner), thus allowing the model to perform an additional task: captioning images. This approach led to the creation of a **basic encoder-decoder image-text pretrained model**, jointly trained using a contrastive loss and a captioning loss. As a result, the model is not only limited to visual recognition tasks, but is also capable of performing downstream tasks, as you can see from the figure below.



CoCa Architecture

CoCa unifies three paradigms into one model: single-encoder, dual-encoder and encoder-decoder. The model has several key features. Cross-attention is omitted in the

unimodal decoder layer, thus allowing only textual representations to be encoded. The multimodal decoder, on the other hand, uses cross-attention on the image encoder outputs to learn multimodal representations. While the dual-encoder approach encodes text as a whole, the generative approach aims for detailed granularity. This means that the model **not only creates a global representation of the text, but also predicts word by word the text associated with the image**.

This is done **autoregressively**, using this formula: $L_{\text{Cap}} = -\sum_{t=1}^{T} \log P_\theta(y_t|y_{<t}, x)$

This loss reflects one of the typical ways to generate a text y using the probability of generating a token y_t given all previous predictions $y_{<t}$ and visual information x from the image encoder. The training approach is termed **autoregressive learning**, and the loss maximizes the conditional probability (P_θ) of paired text images under forward autoregressive factorization.

11.3.9. Objective Functions Improvement: SimVLM

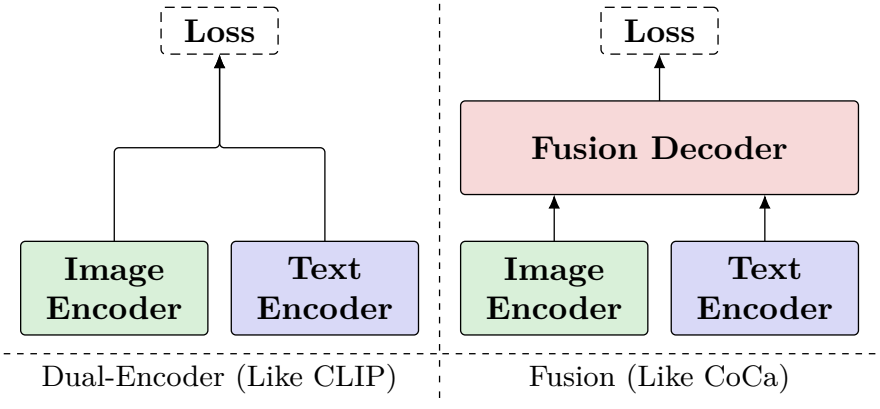
In **"SimVLM: Simple Visual Language Model Pretraining with Weak Supervision" (Wang et al.)**, SimVLM is introduced as a **precursor to CoCa** by the same authors. Although SimVLM was an innovative attempt, it wasn't as competitive as CLIP.

The architecture of SimVLM employs a transformer encoder-decoder model. The idea is to have the model generating text sequentially, using the preceding tokens and visual context to predict each subsequent token.

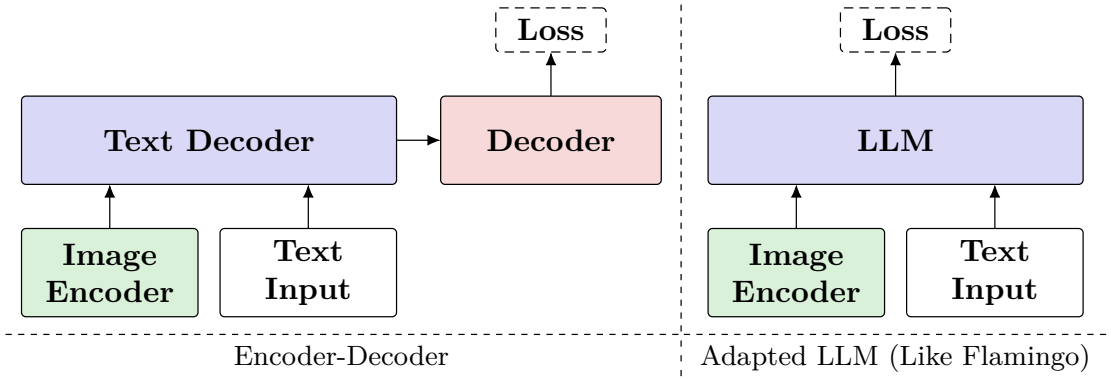
Ultimately, what we have seen so far are three different design approaches for tackling Vision-Language models:

- **Dual Encoders** (CLIP): Text cannot be generated, but it excels in matching images and texts.
- **Encoder-Decoder** (SimVLM): Text can be generated, but it doesn't compete as effectively with CLIP in terms of performance.
- **Fusion Decoder** (CoCa): Combines the strengths of both, enabling both text generation and robust performance.

Here's a recap of all the variations of CLIP that we have explored:



CLIP Architectures Recap (1)



CLIP Architectures Recap (2)

Flamingo, which we will discuss in the next section, blends the dual encoder approach with an **additional LLM** to further enhance performance.

11.4. LLMS AS UNIVERSAL INTERFACE

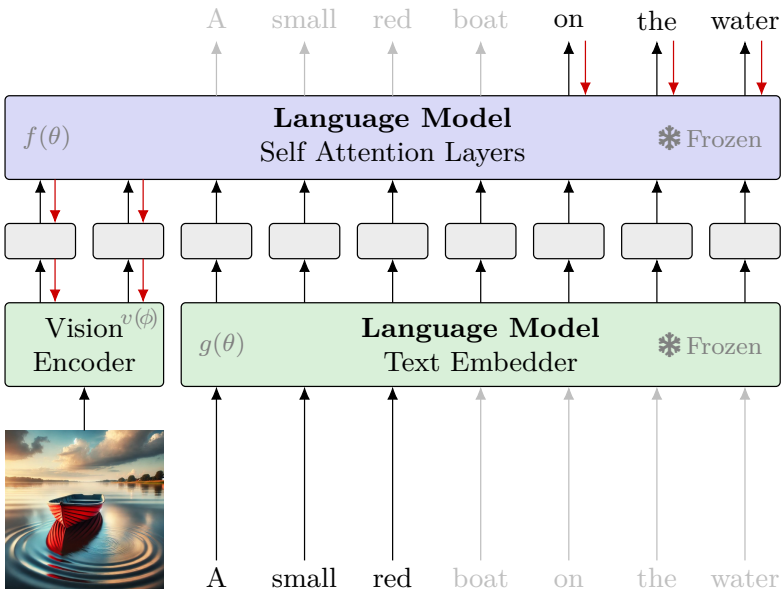
Multimodal systems offer a more flexible way to interact with models, allowing the use of different types of input. For example, users can provide input by typing (text), speaking (audio), or using the camera (images or video).

How can we harness the potential of LLMs to create a general-purpose assistant? These models need to be able to handle a **multimodal prompt**, including images and/or video mixed with text. The idea is to equip LLMs with the ability to "see" the world by training them on vision-conditioned language generation tasks. In this way, LLMs can serve as a general interface for other modalities, leveraging the knowledge gained during training on language-only.

11.4.1. Frozen LM Prefix

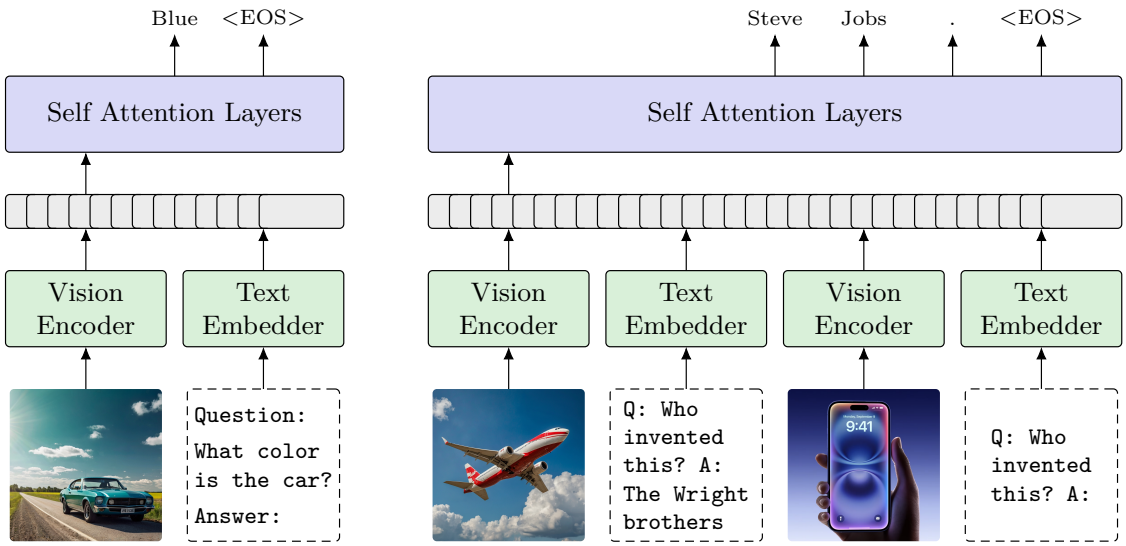
Now, to better understand Flamingo, we must first introduce the concept of Frozen LM Prefix, described in **"Multimodal Few-Shot Learning with Frozen Language Models" (Menick et al.)**.

This model consists of three main components: a **visual encoder** (NF-ResNet50), a **language embedder**, and a **language model for text generation**. The last two components, the language embedder and the language model, are "frozen", meaning that **their weights are not changed during training or inference**. The resulting textual and visual representations are concatenated and then sent to the language model decoder (LLM), which generates the textual output in an autoregressive manner.



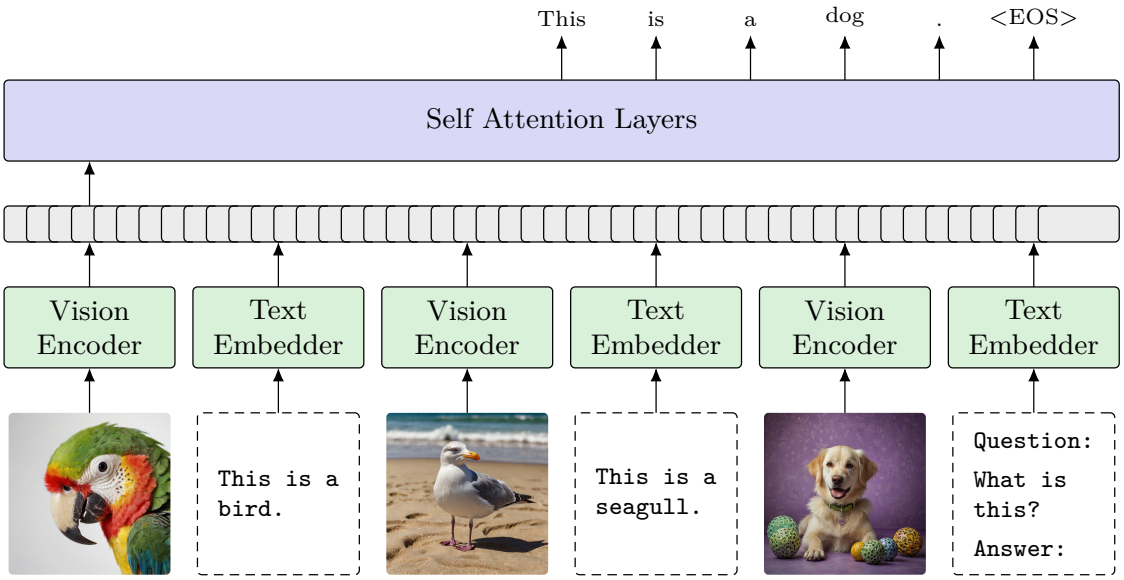
Frozen Architecture

Next, **fine-tuning of the image encoder is performed** to improve the quality of its representations. However, its outputs are directly used as **soft prompts** for the LLM, which means they are **used to guide the behavior of the language model without directly changing its weights**. It is like giving a hint or guidance to the model on what it should focus on. Here are some examples:



(a) **0-shot VQA** (b) **1-shot outside-knowledge VQA**

An example of Finetuning Frozen LM (1)

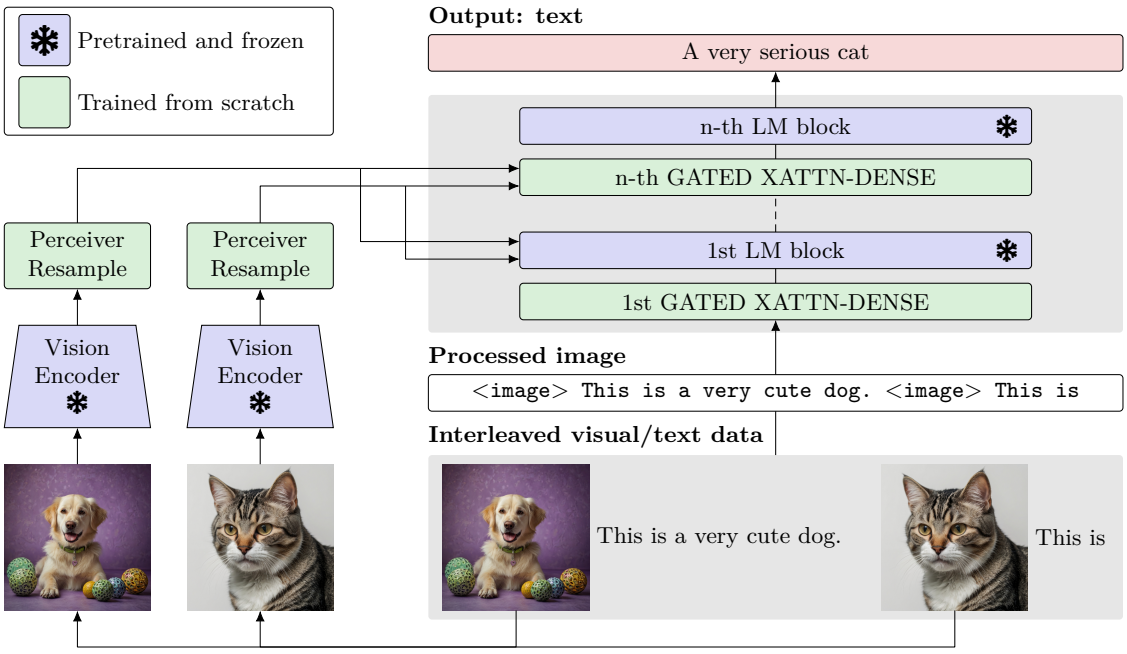


(c) **few-shot image classification**

An example of Finetuning Frozen LM (2)

11.4.2. Flamingo

Flamingo, developed by Google DeepMind and introduced in **"Flamingo: a Visual Language Model for Few-Shot Learning" (Donahue et al.)**, represents an advance over previous frozen language models. It links powerful pre-trained visual encoders and pre-trained language models using innovative architectural components.



Flamingo Architecture

The model can process images, video, and text, enabling it to perform a variety of tasks such as classification, captioning, question answering, and text completion.

Flamingo's visual encoders, based on a **Normalizer-Free ResNet architecture**, are

trained similarly to CLIP to "perceive" visual scenes. For the textual encoder, Flamingo initially uses **BERT** instead of GPT-2; however, it is **discarded after the visual encoder is trained** because it is not used in the final model. Large language models (LLMs) are used to perform basic reasoning tasks.

Flamingo's main innovations include the **Perceiver Resampler** and **Gated XATTN-DENSE** layers, which are trained from scratch.

The model is trained with sequences of arbitrarily interleaved visual and textual data (**heterogeneous data**). One challenge of this approach is to **handle visual inputs of varying sizes**. The Perceiver Resampler addresses this challenge by providing meaningful representations of fixed size. This flexibility allows Flamingo to be trained on **large-scale multimodal data**.

Flamingo uses **Chinchilla** as its language model. To generate conditional text on both textual and visual inputs, Flamingo leverages the Perceiver Resampler and Gated XATTN-DENSE layers.

The Perceiver Resampler maps features of variable size to a **fixed number of output tokens**, regardless of the resolution of the input image or the number of frames in the video. These output tokens, known as latent queries, are a small set learned during training. The output representations from this module are **later used in the XATT module**.

Gated XATTN-DENSE layers (where XATTN stands for cross-attention) are inserted between the frozen language model layers to **allow more efficient attention to visual tokens during the generation of textual tokens**. The output of the Gated XATTN-DENSE layers serves as the key, value, and query for the language model layers.

Finally, text is predicted in an autoregressive manner by the language model.

And here we are, at the end of this journey into the world of Deep Learning. If you have made it this far, congratulations! You have just completed no small feat.

Remember when we started, talking about neurons and activation functions? It seems like an eternity ago, doesn't it? Since then, we have been exploring increasingly complex neural networks, taming gradients, training models, and maybe even dreaming tensors at night.

We hope this book has not only taught you the fundamentals but also ignited a passion for Deep Learning. The challenges ahead are complex, but so are the potential rewards. You have the power to shape the future of technology and positively impact society.

Thank you for joining us on this journey. Good luck and happy innovating!

Federico and Jacopo