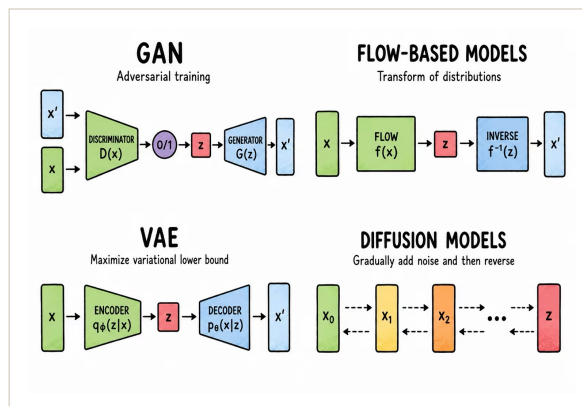


COURSE

Generative Networks

Walking through generative modeling: from simple probability to foundation models



Nahuel Costa

2026

About this book

This book collects the notes for the [Redes Generativas](#) course. The nine chapters follow the same structure as the web version of the course.

Contents

1	Introduction	1
1.1	Who is this course for?	1
1.2	Generative AI	2
1.3	Conditional generative models	6
1.4	Other applications	7
1.5	Timeline	9
2	Probabilistic Machine Learning	11
2.1	Probability	12
2.2	Basic rules	13
2.3	Bayesian inference	15
2.4	Probability distributions	17
3	GAN Networks	19
3.1	Architecture	21
3.2	Explanation	21
3.3	Applications	22
3.4	Loss function	23
3.5	Limitations	24
4	Variational Autoencoders	27
4.1	Autoencoders	27
4.2	VAE	29
4.3	VAE Variants	34
4.4	Extra: Compression	37
5	Flow-based Models	43
5.1	Normalizing flows	44
5.2	Applications	45
5.3	Basic linear algebra concepts	45

5.4	Application to normalizing flows	49
5.5	Models based on Normalizing flows	50
5.6	Advantages and disadvantages	50
5.7	Flow Matching	51
6	Diffusion models	57
6.1	Introduction	58
6.2	Encoder (forward diffusion)	59
6.3	Decoder (reverse diffusion)	61
6.4	Training	62
6.5	Example: Stable Diffusion	64
6.6	Relationship to Flow Matching	70
7	Transformers	73
7.1	Background	74
7.2	Transformers	83
7.3	Extra: Vision Transformers	93
8	Foundation Models	97
8.1	Self-supervised learning	98
8.2	Encoders and decoders	100
8.3	Pre-training tasks	101
8.4	Comparison	115
8.5	Downstream tasks	115
9	Efficiency and cost reduction	119
9.1	Knowledge distillation	120
9.2	Quantization	122
9.3	Low-Rank Adaptation (LoRA)	125
9.4	How to choose a technique	127
	References	129

CHAPTER 1

Introduction

1.1 Who is this course for?

This course is the material I would have liked to find when I started getting interested in generative models (back in 2017).

For people with an engineering background, exploring the Machine Learning literature can feel intimidating. Algebra, calculus, statistics, etc., are subjects taught in the first years of a bachelor's degree, and their details tend to be forgotten if you don't keep practicing them in later years. As a result, concepts like ELBO optimization, density transformation, or optimal transport can seem harder than they actually are.

This course is aimed at anyone who identifies with this situation, has at least a minimal technical background, and, above all, is interested in understanding at a lower level how these models work. My goal is to walk you through learning the underlying concepts while also enabling you use them from a practical standpoint. That's why, in every chapter, you'll find code to reinforce your understanding of the explanations and to let you experiment on your own.

Before we start, two things to clarify:

1. If you're looking for material that goes deep into statistics, calculus, or probability, this isn't the place. On the internet you can find plenty of high-quality content better than these notes.
2. I've tried to keep a visually appealing aesthetic so this is a resource you'll want to come back to, but it's important to note that understanding and consolidating knowledge takes time.

"Learning is not supposed to be fun. It doesn't have to be actively not fun either, but the primary feeling should be that of effort." — [Andrej Karpathy](#)

1.2 Generative AI

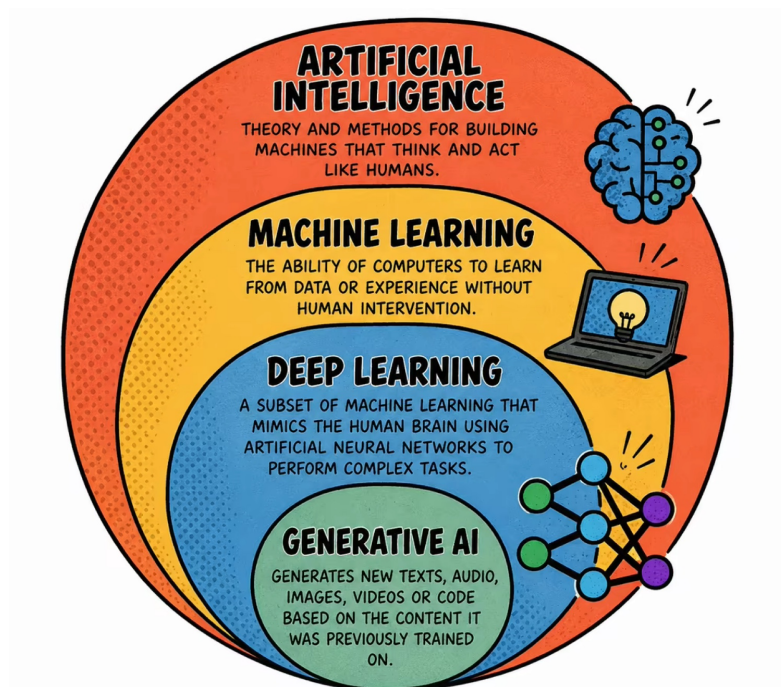


Figure 1.1. Overview of generative AI

As you can see in the figure, within the broader landscape of artificial intelligence, generative AI covers methods and techniques capable of generating new data samples.

From a probabilistic point of view, we fit a model to the training distribution $p_{\text{data}}(x)$ and then sample from the learned distribution $p_{\theta}(x)$.

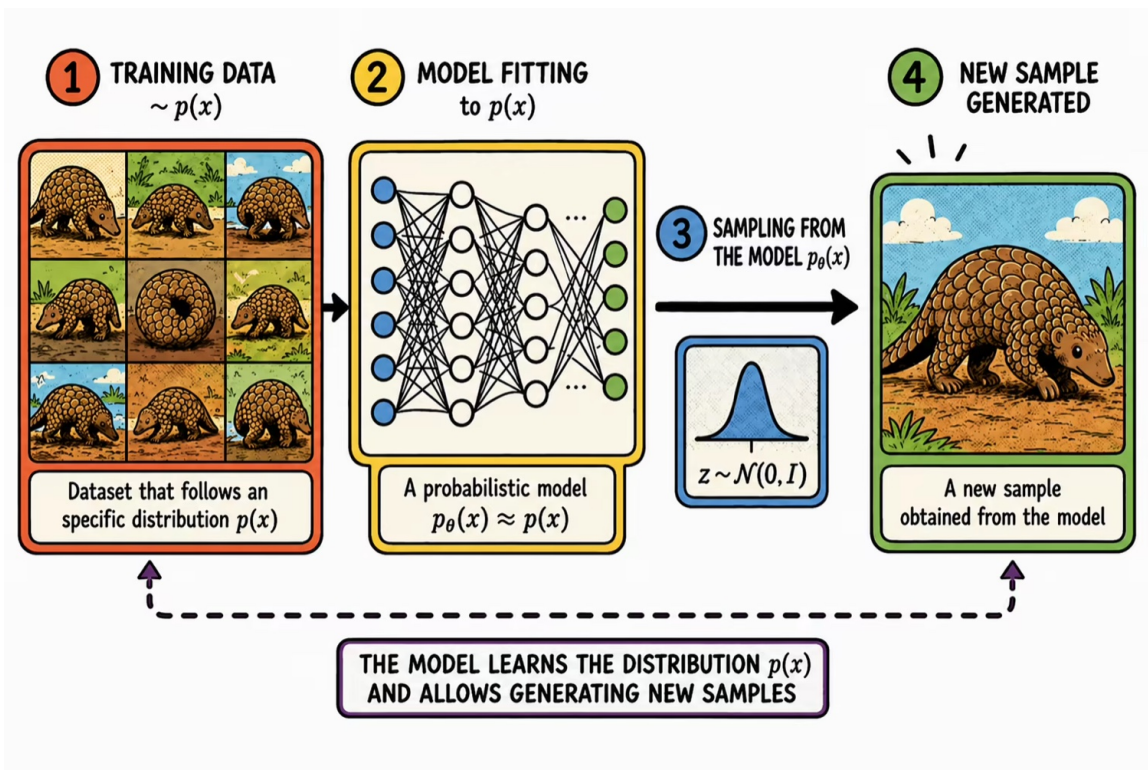


Figure 1.2. A generative model learns the distribution of pangolin images to create new samples

This process applies to a variety of domains. Below are some examples you're probably already familiar with.

1.2.1 Image generation

Nano Banana (Google), GPT Image 2 (OpenAI), DALL · E, Stable Diffusion, Adobe Firefly, Mid-journey, FLUX...

Note

The illustrations in this course were generated with [GPT Image 2](#).

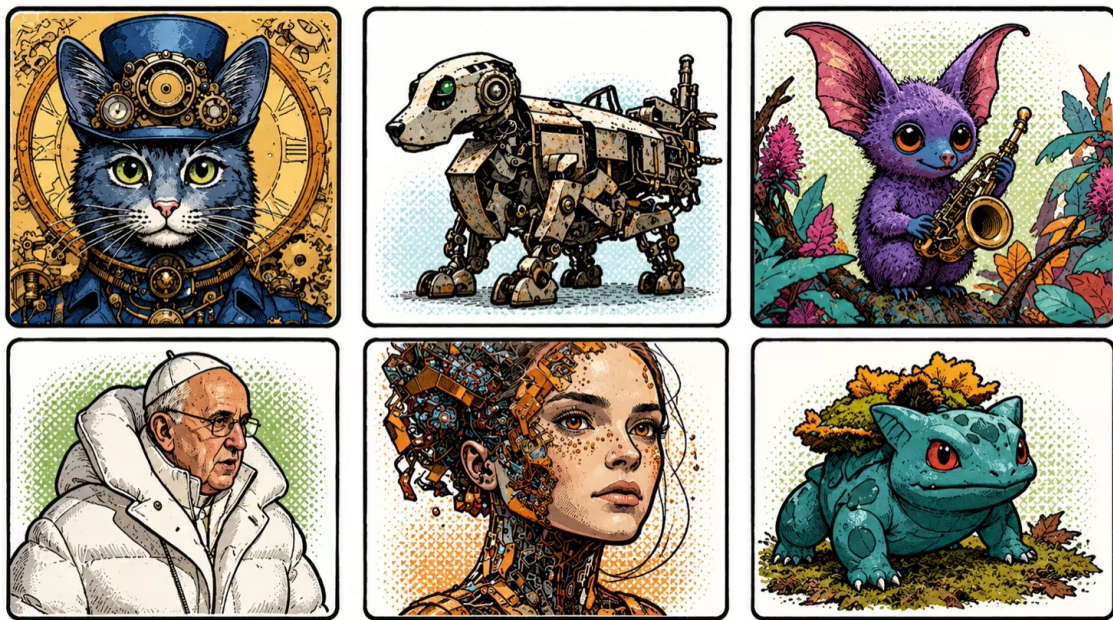


Figure 1.3. Examples of image generation

1.2.2 Text / code generation

ChatGPT, Claude, Gemini, Grok, Mixtral...

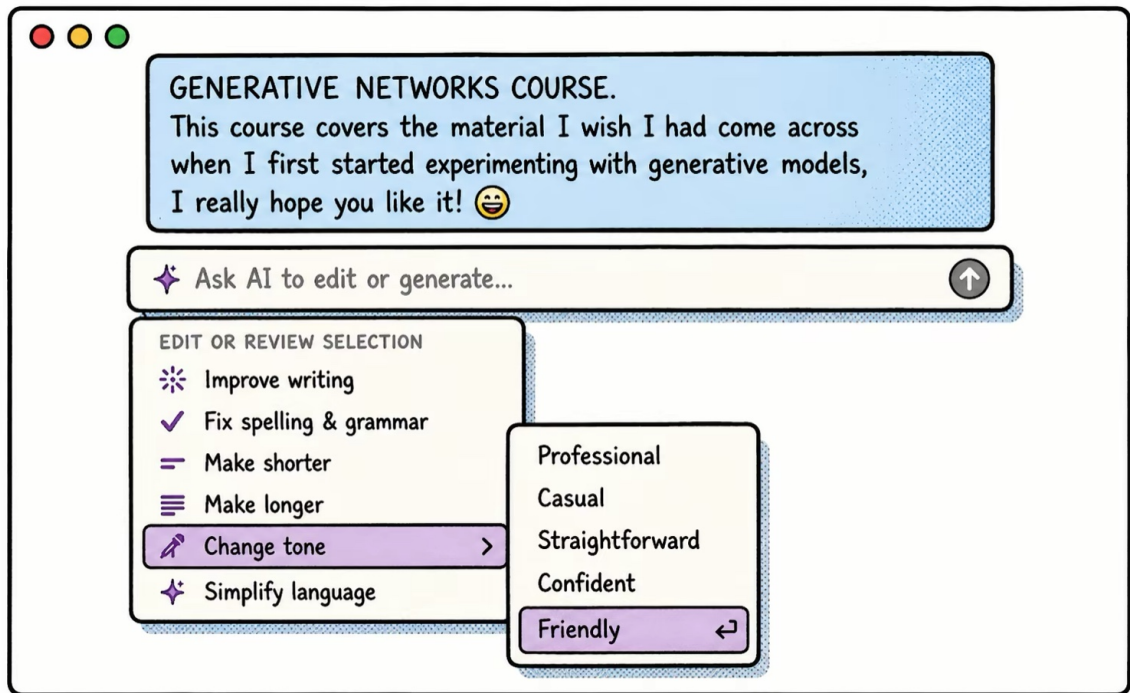


Figure 1.4. Example of text generation and editing assisted by Notion AI

1.2.3 Audio generation

Suno, ElevenLabs, WaveNet, VALL-E, Stable Audio, Udio...

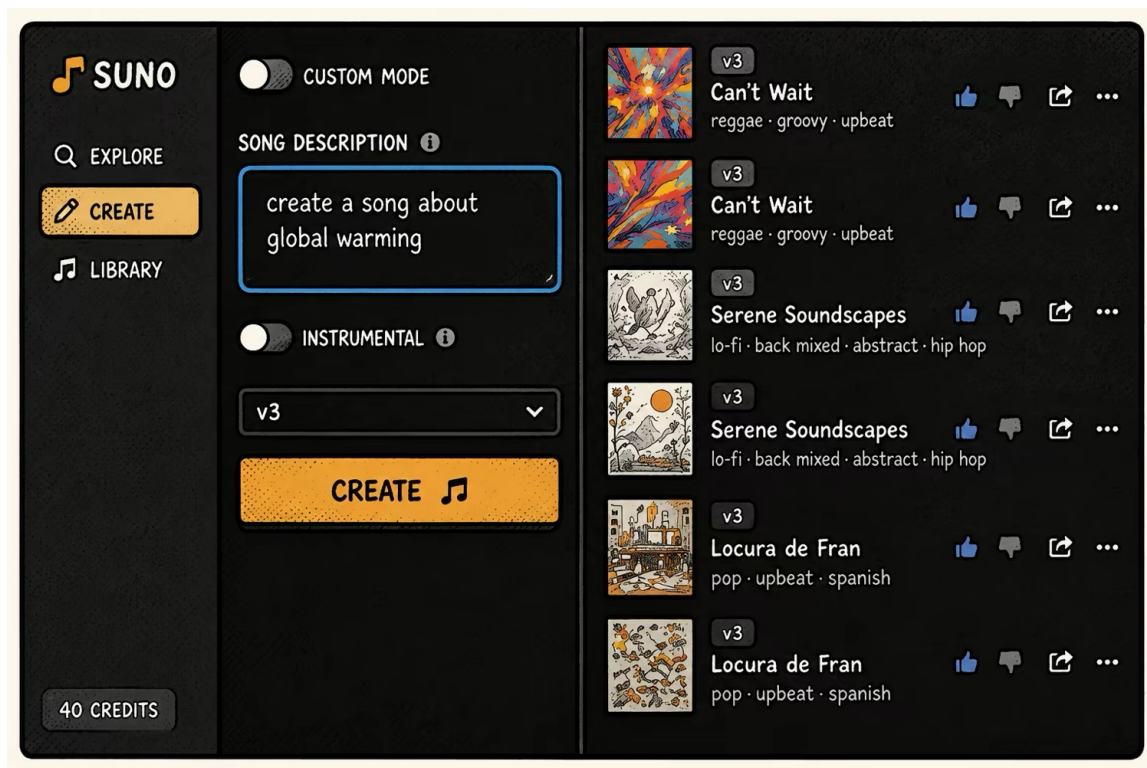


Figure 1.5. Suno interface for generating music from a text description

1.3 Conditional generative models

We generally want to control what gets generated. That's why conditional generative models are used, in the form $p(x | c)$. Some examples:

- $c = \text{text}$, $x = \text{image}$: a **text-to-image** model. Example: Nano Banana.
- $c = \text{image}$, $x = \text{text}$: an **image-to-text** model. Example: give ChatGPT an image and ask it to describe it.
- $c = \text{image}$, $x = \text{image}$: an **image-to-image** model. Example: image editing in [Adobe Firefly](#).
- $c = \text{audio}$, $x = \text{text}$: a **speech-to-text** model. Example: Whisper, or any voice dictation.
- $c = \text{text}$, $x = \text{audio}$: a **text-to-speech** model. Example: [ElevenLabs TTS](#).
- $c = \text{text}$, $x = \text{text}$: a **sequence-to-sequence** model. Example: translators (text generation models would also fall under this category).

1.4 Other applications

The nice thing about modeling a distribution is that it doesn't just let you generate data, but a lot of other things too.

1.4.1 Density estimation

This consists of evaluating the density that the model assigns to a data sample, i.e., computing $p(x)$. This is useful for problems such as data compression, anomaly detection, classification, model comparison, etc.

The simplest approach to density estimation is to use **kernel density estimation** or **KDE**. However, KDE runs into limitations in high-dimensional spaces (the **curse of dimensionality**), where the density of the data becomes extremely sparse.

Modern generative networks mitigate this limitation by learning compact, complex representations that make it possible to estimate or model density in a much more efficient and scalable way for high-dimensional data such as images, audio, or text.

1.4.2 Data imputation

This consists of "filling in" missing values in a data vector or matrix. For example, a simple way to fill in missing data in tabular data is to use the mean value of each column (mean imputation).

We can generalize this by learning a joint model over all the variables and, for each incomplete record, sampling the missing values X_m conditioned on the observed ones X_o : $p_{\theta}(X_m | X_o)$. This is called **multiple imputation**.

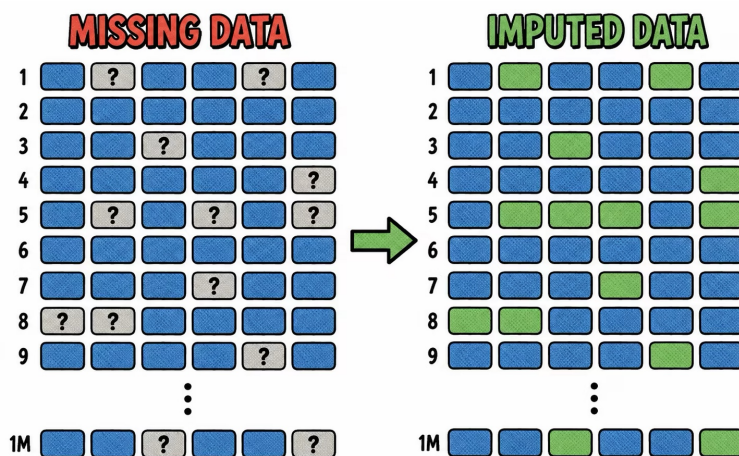


Figure 1.6. Missing variables are sampled conditioned on the observed variables

A generative model can be used to fill in more complex types of data, such as masked pixels in an image.

1.4.3 Representation learning

Some types of generative models have latent variables z , which can be understood as the causes that generated the observed data x . These variables generally reduce the dimensionality of the data and make it possible to discover patterns / [useful representations](#).

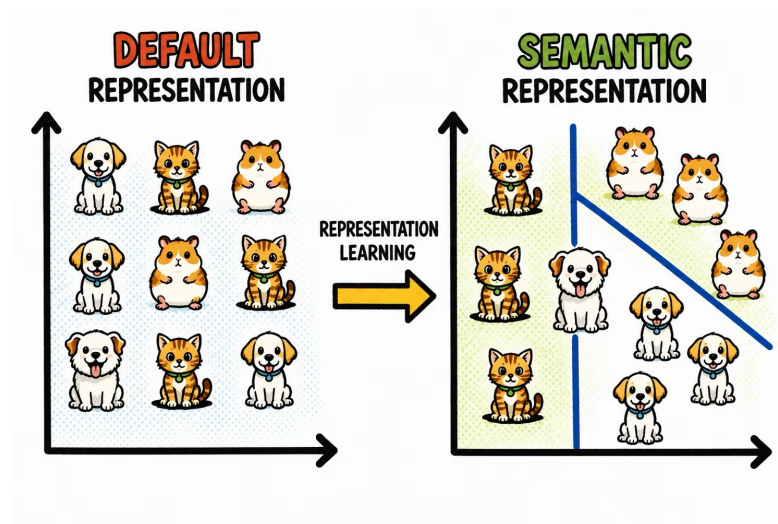


Figure 1.7. An encoder transforms observations into useful latent representations

1.4.4 Interpolation in the latent space

The latent space is where the latent variables live. It can be understood as the space into which models compress the data.

One of the most interesting capabilities of certain latent-variable models is the ability to generate samples with certain desired properties by interpolating between existing data points in the latent space.

In the following example we reduce the latent space to two dimensions: each chair is a point on a plane. We pick four samples and interpolate their coordinates. Decoding the intermediate points gives us representations that combine properties of the selected samples.

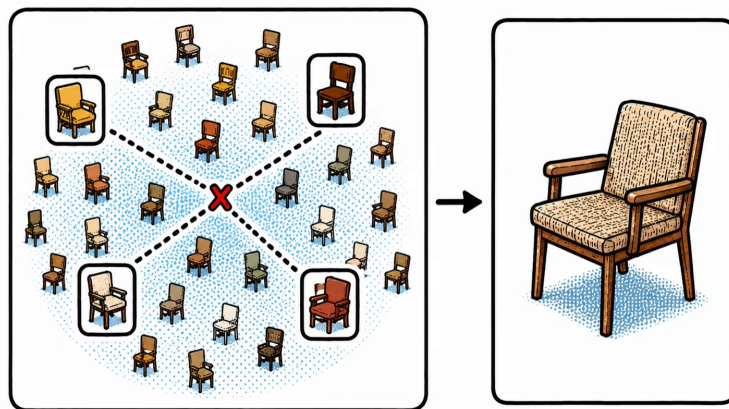


Figure 1.8. Interpolation between four points in a latent space of chairs and their decodings

1.4.5 Data compression

Models that can assign high probability to frequently occurring data and low probability to rare data can be used for data compression, since they can assign shorter codes to the most common elements. Compression is a beautiful field that overlaps directly with Information Theory. We'll cover some of this in the [Variational Autoencoders](#) chapter.

Note

Claude's name comes from Claude Shannon, author of the modern formulation of Information Theory¹.

1.5 Timeline

This timeline brings together a selection of particularly influential works in the evolution of generative models. Don't worry if you don't understand what they're about right now — it's a reference you can come back to later.

We'll also cover foundational models. However, the literature is much broader than that, so I've decided to include here works related to generative models, along with some architectures and training techniques (such as CLIP, LoRA, RLHF, or DPO) that, while not generative themselves, have been key to their evolution.

On the web

The interactive timeline of generative models is available in the web version of the course.

Probabilistic Machine Learning

There are many types of generative models. At a high level we can distinguish between:

- **Probabilistic Graphical Models (PGM)**: they represent, by means of a graph, the probabilistic relationships between observed (and/or latent) variables, showing how a joint distribution decomposes and what conditional independences exist between them.
- **Deep Generative Models (DGM)**: they use deep neural networks to model or generate data distributions. Some incorporate latent variables z , although this is not a requirement common to all families.

Of course, many hybrids are possible. From here on we'll focus on DGMs.

The main types of DGM are **variational autoencoders (VAE)**, **generative adversarial networks (GAN)**, **autoregressive models (ARM)**, **flow-based models** and **diffusion models**.

These models can be classified according to the following criteria:

- **Density**: models that can estimate the probability density function $p(x)$. For example, GANs model the data distribution implicitly, so they cannot.
- **Sampling**: models that can generate new samples from the modeled distribution. Models like VAEs and GANs allow fast sampling. However, ARMs, diffusion models or normalizing flows are (generally) slow to sample from.
- **Training**: what kind of method is used to estimate the parameters? For some models (like ARM and flows), an exact maximum likelihood estimation (MLE) can be performed. For other models it isn't so simple. For example, in the case of VAEs a lower bound on the likelihood is maximized.
- **Latents**: does the model use a latent vector z to generate x ? And, if so, does it have the same size as x or is it a compressed representation? For example, ARMs don't use latent representations; flows and diffusion do, but they aren't compressed representations.
- **Architecture**: what kind of neural network can be used? Are there any restrictions? In the case of flows, for example, only invertible neural networks can be used, where each layer has a tractable Jacobian.

Here's a brief summary of how the main models work. If this is the first time you're seeing this diagram, don't worry if you don't understand it yet. I recommend coming back to it as we move

forward.

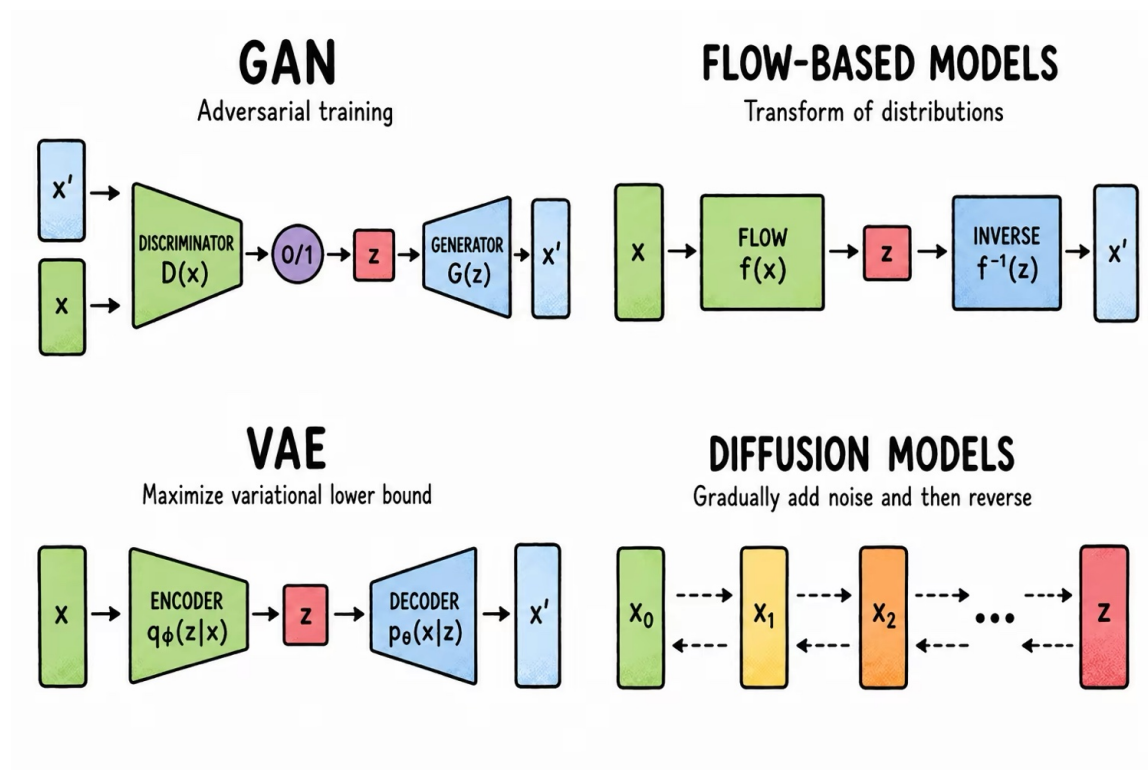


Figure 2.1. Overview of generative AI

In the coming chapters we'll make use of probabilistic notation. If it's something you already know well, you can skip straight to the first chapter. If not, in the rest of this article we'll do a quick review of probability.

2.1 Probability

There are two different interpretations of probability:

- **Frequentist:** probabilities represent frequencies of events that can occur multiple times. Example: the probability of rolling a 6 if you roll a die 600 times. We expect the number 6 to come up approximately 100 times. It's assumed that conditions are stable and that, if you repeat the experiment many times, the result will converge toward a fixed frequency (in this case $1/6$).
- **Bayesian:** probability is used to quantify uncertainty/ignorance about something. The Bayesian interpretation can be used to model uncertainty about one-off events that don't have long-run

frequencies.

Example: say we want to calculate the probability that the polar ice cap will melt in 2030. This event will either happen or not, but it can't happen repeatedly. Nevertheless, we should be able to quantify our uncertainty about this event.

From here on we'll adopt the Bayesian interpretation. Fortunately, the basic rules of probability theory are the same regardless of which interpretation is adopted.

Types of uncertainty

Uncertainty in predictions can fundamentally arise for two reasons:

- **Aleatoric uncertainty.** Arises from the intrinsic variability of the data.
Example: flipping a coin in a uniform way. You can measure the force with which you flip it, air resistance, gravity, etc., but there will always be uncontrollable microscopic fluctuations. It's not reduced with more data or better models.
- **Epistemic uncertainty.** Arises from our ignorance or lack of information about a system that, in theory, is deterministic (if we had all the information, we could know the outcome).
Example: predicting whether a patient has a disease based on an initial symptom. The patient either has the disease or doesn't. It's not a matter of chance, it's a matter of data. It's reduced by collecting more data (biopsies, scans, etc.).

2.2 Basic rules

2.2.1 Probability of an event

We define an event A as a state that either holds or doesn't. For example, A can be the event "it will rain tomorrow" or "it rained yesterday". The expression $\Pr(A)$ denotes the probability with which the event A is believed to be true. We require $0 \leq \Pr(A) \leq 1$, where $\Pr(A) = 0$ means the event won't occur and $\Pr(A) = 1$ means it will.

2.2.2 Random variables

Suppose X represents some unknown quantity, like the face a die will land on when rolled. If the value of X is unknown and/or can change, we call it a **random variable**. The set of possible values, denoted $\mathcal{X}$, is known as the **sample space** or state space. An **event** is a set of outcomes from a given sample space.

For example, if X represents the face of a die, $\mathcal{X} = \{1, 2, \dots, 6\}$. The event "seeing a 1" is denoted $X = 1$, the event "seeing an odd number" is denoted $X \in \{1, 3, 5\}$, the event "seeing a number between 1 and 3" is denoted $1 \leq X \leq 3$, etc.

2.2.3 Discrete random variables

If the sample space $\mathcal{X}$ is countable (finite or countably infinite), X is called a **discrete** random variable. Example: rolling a die, the number of customers entering a store, the number of heads in 5 coin flips...

The function that describes the probability distribution of discrete random variables is called the **probability mass function** or **pmf**:

$$p(x) = \Pr(X = x),$$

where x is a possible value within the sample space.

2.2.4 Continuous random variables

If $X \in \mathbb{R}$, it's called a **continuous** random variable. Example: time, weight, temperature...

In this case, you can't create a finite set of the different possible values the variable can take. However, they can be divided into different intervals.

Here, unlike the PMF, which tells us the probability of an exact value occurring, we use the **cumulative distribution function** or **cdf**, which tells us the probability of that value or any smaller one occurring:

$$P(x) = \Pr(X \leq x)$$

Note that an uppercase P is used to represent the cdf.

Using this, we can calculate the probability of being within any interval:

$$\Pr(a < X \leq b) = P(b) - P(a)$$

The function that describes the probability distribution of continuous random variables is called the **probability density function** or **pdf**, which coincides with the derivative of the cdf:

$$p(x) = \frac{d}{dx}P(x)$$

- The PMF (discrete) measures "bars" (exact points).
- The PDF (continuous) measures "areas" (intervals).

Given a pdf, we can calculate the probability that a continuous variable falls within a finite interval as:

$$\Pr(a < X \leq b) = \int_a^b p(x) dx = P(b) - P(a)$$

Example: Waiting time at a bus stop

Imagine the waiting time (in minutes) for the bus follows a uniform distribution between 0 and 10 minutes. Its PDF is constant: $p(x) = 1/10$ for $0 \leq x \leq 10$, and 0 otherwise. If we want to know the probability that the bus arrives between minute 2 and minute 5, $\Pr(2 < X \leq 5)$, we use the integral of the PDF:

$$\Pr(2 < X \leq 5) = \int_2^5 \frac{1}{10} dx$$

We calculate the integral:

$$\left[\frac{1}{10}x \right]_2^5 = \frac{5}{10} - \frac{2}{10} = \frac{3}{10} = 0.3$$

Therefore, there is a 30% probability that the bus arrives within that time interval.

2.2.5 Statistics

- **Mean** (μ): arithmetic average, represents the value you would get if you could distribute the total of the data equally among all the elements.
- **Variance** (σ^2): a measure of the spread of a distribution, it measures how far, on average, the values deviate from the mean. Since it squares the differences, its units are also squared (for example, if you measure in meters, the variance will be in square meters).
- **Standard deviation** (σ): the square root of the variance. It fixes the "problem" with the variance. It gives an idea of how far the data deviates from the average in the original units.

2.3 Bayesian inference

The term "inference" refers to the act of generalizing from sample data, usually with some degree of confidence. The term "Bayesian" is used to refer to inference methods that represent that confidence using probability theory and Bayes' theorem.

Bayes' theorem is a formula for calculating the probability distribution over possible values of an unknown quantity H given observed data Y :

$$p(H | Y) = \frac{p(Y | H) p(H)}{p(Y)}$$

- $p(H)$ represents what is known about the possible values of H before seeing any data; this is the **prior distribution**.
- $p(Y | H)$ represents the distribution over the possible outcomes Y we expect to see given H ; this is the **observation distribution**.
- If we fix it at the value $Y = y$ corresponding to the actual observations, but treat it as a function of H (rather than of Y), we get the **likelihood function**, $\mathcal{L}(H) = p(Y = y | H)$.

Multiplying the prior distribution $p(H)$ by the likelihood function $p(Y | H)$ for each h gives the **unnormalized joint distribution** $p(H, Y)$. It can be turned into a normalized distribution by dividing by $p(Y)$, which is known as the **marginal likelihood** (we'll see this later).

By normalizing the joint distribution, we get the **posterior distribution**, $p(H | Y)$, which represents what is known about the distribution after seeing evidence. In other words:

$$\text{posterior} \propto \text{prior} \times \text{likelihood}$$

Example: test for a rare disease

Imagine you want to know whether a patient has a rare disease (Hypothesis H) that affects only 1% of the population.

1. **Prior distribution**, $p(H)$: before taking the test, the probability that the patient is sick is 0.01.
2. **Likelihood**, $p(Y | H)$: the sensitivity is $p(Y = \text{pos} | H = \text{sick}) = 0,99$ and the false positive rate is $p(Y = \text{pos} | H = \text{healthy}) = 0,05$.
3. **Posterior distribution**, $p(H | Y)$: once a positive result is observed, we want to calculate the probability that the patient is actually sick.

You might confuse the 99% sensitivity with the probability of being sick after a positive result. However, applying Bayes' theorem we get:

$$p(H = \text{sick} | Y = \text{pos}) = \frac{p(Y = \text{pos} | H = \text{sick}) \cdot p(H = \text{sick})}{p(Y = \text{pos})}$$

Where:

- Prior $p(H = \text{sick}) = 0.01$ (only 1% of the population is sick).
- Likelihood $p(Y = \text{pos} | H = \text{sick}) = 0.99$ (the test detects 99% of sick people).
- False positives $p(Y = \text{pos} | H = \text{healthy}) = 0.05$ (the test is wrong for 5% of healthy people).

Step 1: Calculate the total probability of a positive result $p(Y = \text{pos})$

A positive can be a true positive (sick person who tests positive) or a false positive (healthy person who tests positive):

$$p(Y = \text{pos}) = (\text{Sensitivity} \times \text{Prevalence}) + (\text{False positive rate} \times \text{Healthy population})$$

$$p(Y = \text{pos}) = (0.99 \times 0.01) + (0.05 \times 0.99)$$

$$p(Y = \text{pos}) = 0.0099 + 0.0495 = 0.0594$$

Step 2: Calculate the Posterior

Now we substitute into Bayes' formula:

$$p(H = \text{sick} | Y = \text{pos}) = \frac{0.99 \cdot 0.01}{0.0594} \approx 0.166$$

Conclusion: Although the test has a sensitivity of 99%, the probability that the patient is sick after testing positive is approximately 16.7%. The low prevalence (the prior) "pulls" the result down very strongly. The number of healthy people in the population is so large that the 5% false positive rate outnumbers the true positives. This example shows that, in Bayesian inference, the observed data isn't the only thing that determines reality; our initial belief about how rare the event is plays a fundamental role.

Note

One challenge of the Bayesian approach is specifying a suitable prior distribution. There are different approaches such as conjugate priors, informative priors, hierarchical or empirical ones. We'll mainly use simple priors.

2.4 Probability distributions

There is a wide variety of probability distributions used for different types of models. At [this link](#) you can explore some of the most commonly used ones interactively.

A generative model learns a distribution over the data, $p(x)$ for $x \in \mathcal{X}$, or a joint distribution $p(x, y)$ when several variables are involved. In the following chapters we'll see different ways of representing, approximating and sampling from it.

CHAPTER 3

GAN Networks

Chapter code: github.com/NahuelCostaCortez/generative-networks

”The most interesting idea in the last 20 years in Machine Learning” — *[Yann LeCun, 2016]*

Until the arrival of Transformers and diffusion models, **Generative Adversarial Networks**³ had broadly impacted the generative landscape, showing great advances in many tasks such as image generation, natural language, and music.

Back in the day, websites like [thispersondoesnotexist](https://thispersondoesnotexist.com/) and its [variants](#) became famous. Today we are used to seeing realistic AI-generated images, but back then it was surprising to think that those faces did not correspond to real people, but had instead been generated by a model.

Nowadays, GANs are no longer the default option for many generation tasks, but they remain useful in specific niches, when fast sampling or translation between domains is needed.

They are inspired by game theory: two models, a **generator** and a **discriminator**, compete while also improving each other.

GENERATIVE ADVERSARIAL NETWORKS

@MIAMIAMIA0103

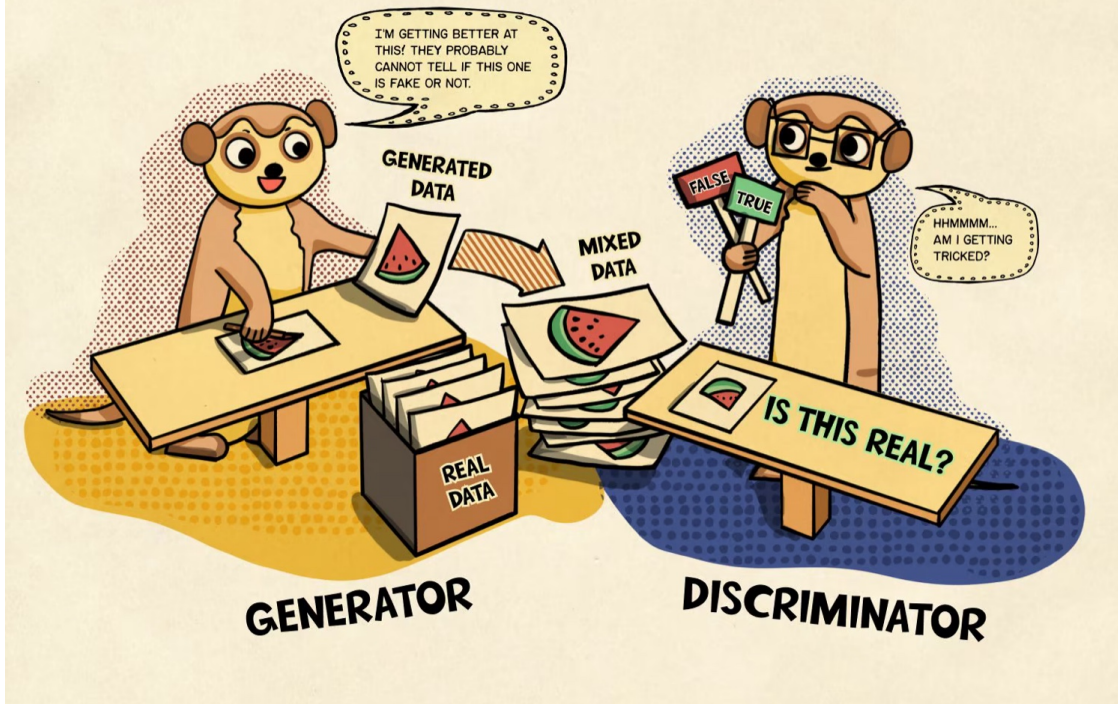


Figure 3.1. Source: [miamiamia0103](#)

3.1 Architecture

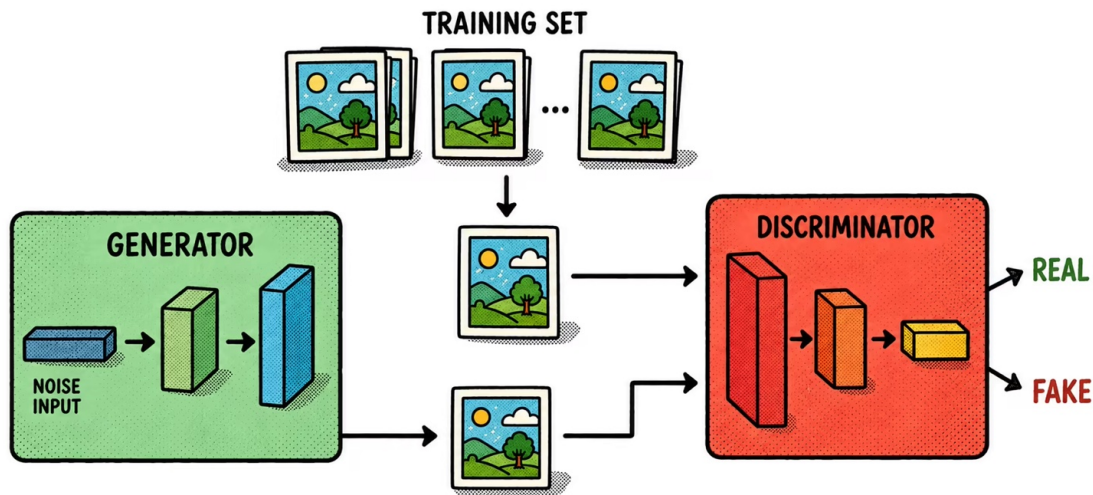


Figure 3.2. Architecture of a GAN

- **Generator network:** receives as input a vector of random numbers or Gaussian noise (which is why it is sometimes denoted z), from which it generates data.
- **Discriminator network:** it has to identify whether the data it receives is real (part of the training set) or fake (generated by the generator network).

[Playground](#)

3.2 Explanation

The discriminator's task is to differentiate whether the samples it receives correspond to the training data distribution ('real' data) or whether, on the contrary, they are samples generated by the generator ('fake' data). The generator's objective, therefore, is to produce samples that are as realistic as possible in order to try to fool the discriminator.

At the beginning of training, neither network is good at its task. The generator produces blurry, incoherent images, and the discriminator is not good at telling real data from fake data apart.

However, as training progresses, the generator produces better examples to try to fool the discriminator, and the discriminator, by seeing more real data, learns to tell it apart from generated data, so the two enter a zero-sum game.

Note

At the ideal equilibrium, the generated distribution matches the real distribution, $p_g = p_{\text{data}}$, and the discriminator assigns $D(x) = 1/2$ to any sample. At that point it can no longer distinguish between real and generated data, so neither network can improve unilaterally.

Normally, after training the discriminator is discarded, since the main interest lies in data generation. However, the discriminator is a good binary classifier that could be used for other tasks.

3.3 Applications

The most studied and popularly known application area is image generation. Here are some applications:

- **Conditional image generation:** for example, BigGAN⁴ can generate ImageNet samples conditioned on a specific class. StyleGAN⁵ is capable of generating high-resolution face images and introduces a style-based architecture that allows interpolation in the latent space and control of characteristics at different scales, such as overall face geometry, hairstyle, or texture.
- **Paired image-to-image generation:** data in the form (x_n, y_n) can be used to build conditional generative models $p(x | y)$. In some cases, the conditioning variable y has the same dimension as the output variable x . Thus, a model can be used to perform image-to-image translation, a process where an image is generated conditioned on the style of another. An example of this would be colorization, where the model takes a black-and-white photo and "fills in" the colors based on the patterns it has learned, or segmentation, where the model converts a color-label scheme (like a map where each color is an object) into a realistic photograph that exactly respects those shapes.
- **Unpaired image-to-image generation:** one limitation of conditional GANs is the difficulty of collecting paired data. It is much easier to collect unpaired data, for example, a set of daytime images D_x and a set of nighttime images D_y . We assume that the sets D_x and D_y come from the marginal distributions $p(x)$ and $p(y)$ respectively. Since these marginal distributions do not by themselves determine a unique correspondence between the two domains, models like CycleGAN learn transformations in both directions and introduce additional constraints, such as cycle consistency. This is called **unsupervised domain translation** and has applications such as style transfer.

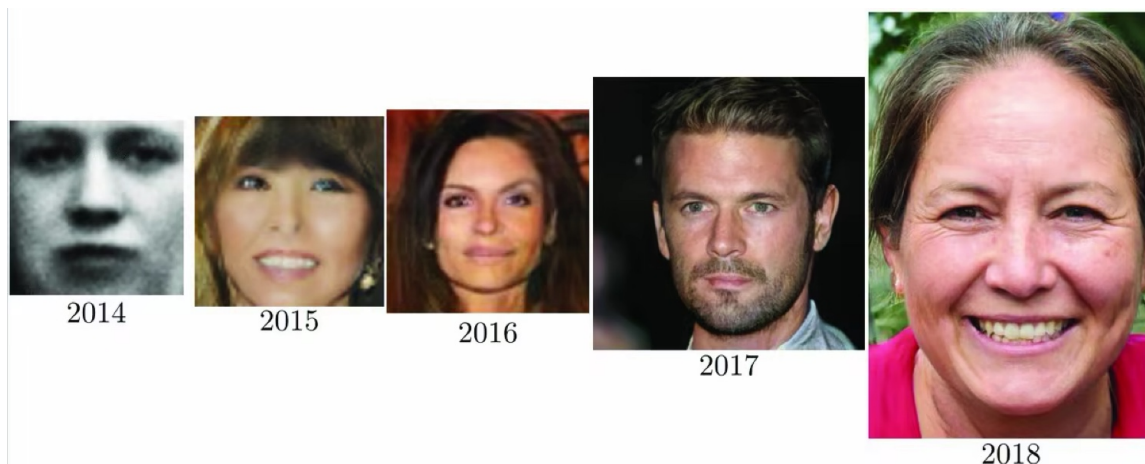


Figure 3.3. From left to right: original GAN (2014), DCGAN (2015), CoupledGAN (2016), ProgressiveGAN (2017), BigGAN (2018). [Source](#).

Applications in other domains

- **Video generation:** spatio-temporal coherence is obtained by ensuring that the discriminator has access to real data and to the generated sequences in order, thereby penalizing the generator when it generates individually realistic frames without respecting temporal order.
- **Audio generation:** GAN architectures have been developed to replicate instruments with GANSynth⁶, to convert voices⁷, and to directly generate waveforms with WaveGAN⁸.
- **Text generation:** there are several text-data tasks for which GAN-based approaches have been developed, such as conditional text generation and text style transfer. Text data is usually represented as discrete values (at the character or word level), indicating membership in a set of a given vocabulary.
- **Domain adaptation:** an important task in machine learning is correcting shifts in data distribution, since in many cases the distribution on which the model is evaluated at inference time is not exactly the same as the training data distribution. In general, the image generation approaches mentioned focus on pixel-level adaptations, such as pix2pix⁹. However, extensions of these approaches to the general domain adaptation problem seek to do so not only in the observed data space, but also at the feature level.

3.4 Loss function

On one hand, we want the discriminator to become an expert at identifying what is real and what is fake:

- Identifying the real: We **maximize** $\mathbb{E}_{x \sim p_r} [\log D(x)]$, forcing the discriminator to assign a probability close to 1 to any data coming from the real set.

- Identifying the fake: We **maximize** $\mathbb{E}_{z \sim p_z} [\log(1 - D(G(z)))]$, forcing the discriminator to assign a probability close to 0 to any data generated by G.

On the other hand, the generator is trained to increase its chances of producing a high probability for a fake example, thus **minimizing** $\mathbb{E}_{z \sim p_z} [\log(1 - D(G(z)))]$.

What is E and why use log?

- The symbol E means Mathematical Expectation (or expected value). It is simply an average. When we write $\mathbb{E}_{x \sim p_r} [\dots]$, we are just saying: "Calculate the average of [...] for the data x that comes from distribution r ".
- Small probabilities are often multiplied together, and multiplying very small numbers (e.g. 0.0001×0.0001) makes the result tend toward zero. The logarithm turns those multiplications into sums (e.g.: $\log(a \cdot b) = \log(a) + \log(b)$), which makes the calculations much more stable.
- In the case of GANs, if the discriminator is very confident about something but is wrong (for example, it says a fake datum is 99% real), the logarithm of a number close to zero ($\log(1 - 0.99)$ in this case) is a very large negative number. This forces the network to correct its error immediately.

By combining both parts, we are playing a 'minimax' game in which we must optimize the following loss function:

$$\begin{aligned} \min_G \max_D L(D, G) &= \mathbb{E}_{x \sim p_r(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))] \\ &= \mathbb{E}_{x \sim p_r(x)} [\log D(x)] + \mathbb{E}_{x \sim p_g(x)} [\log(1 - D(x))] \end{aligned}$$

Note

In practice, minimizing $\mathbb{E}_{z \sim p_z} [\log(1 - D(G(z)))]$ can produce very small gradients when the discriminator confidently identifies fake samples. For this reason, the generator is normally trained by maximizing:

$$\mathbb{E}_{z \sim p_z} [\log D(G(z))]$$

This loss has the same equilibrium point, but provides more useful gradients early in training.

3.5 Limitations

Although GANs have shown great success in generating realistic images, their training can be slow and unstable.

Each model updates its cost independently without taking the other "player" into account. The simultaneous gradient update of both models does not guarantee convergence. This can sometimes

lead to **mode collapse**. During training, the generator can collapse into a configuration in which it always produces the same outputs. Even though the generator may be able to fool the discriminator, it fails to learn to represent the complex real-world data distribution and gets stuck in a small space with extremely low variety.

Training a GAN faces a dilemma:

- If the discriminator is not good, the feedback the generator receives is not good either, and as a result the loss function does not represent reality (causing the generator to fail to properly model the real data distribution).
- If the discriminator is very good, the gradient of the loss function drops close to zero and learning becomes slow or even gets stuck (**vanishing gradient**).

Heads-up

This dilemma can make training GANs very difficult. To mitigate these difficulties, there are methods that make this process simpler².

CHAPTER 4

Variational Autoencoders

Chapter code: github.com/NahuelCostaCortez/generative-networks

Before delving into variational autoencoders, let's do a brief review of autoencoders.

4.1 Autoencoders

Autoencoders are a family of neural networks whose goal is to learn a simplified representation of the input data.

Their operation is based on **compressing and reconstructing data**. They generally consist of two networks:

1. An **Encoder** $g(\cdot)$ that compresses the input data x into a lower-dimensional space, known as the **latent space**, $z = g_\phi(x)$.
2. A **Decoder** $f(\cdot)$ that reconstructs the data from the latent space, $x' = f_\theta(g_\phi(x))$.

The parameters of each network (ϕ, θ) are optimized jointly so that the reconstructions are as similar as possible to the original data, $x \approx f_\theta(g_\phi(x))$.

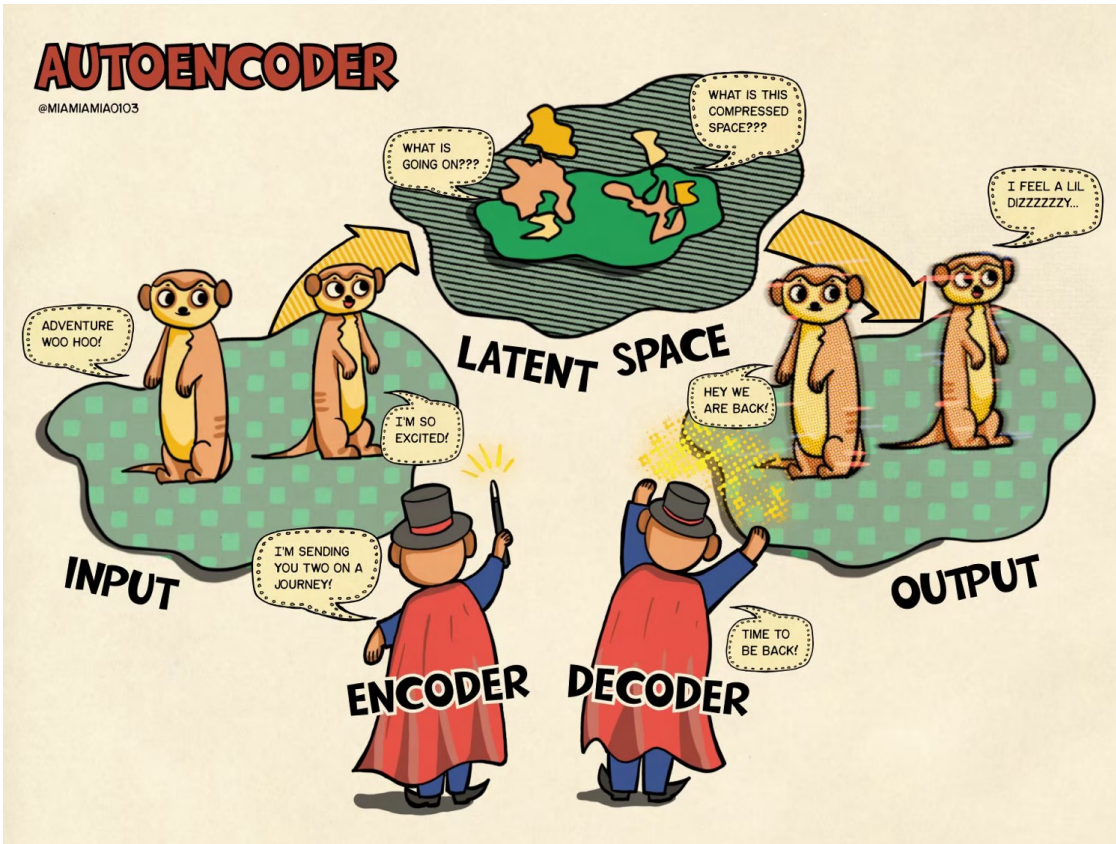


Figure 4.1. Source: [miamiamia0103](#)

The idea originated in the 1980s and was taken up again, among other works, by Hinton and Salakhutdinov¹⁰ in 2006. Today it has direct applications in tasks such as anomaly detection, manifold learning, compression, or data generation.

The loss function of a conventional autoencoder measures how well the model can reconstruct the original input once the data has been compressed into the latent space. Therefore, the goal is for the output $\hat{x}$ to be as similar as possible to the input x . The choice of error function depends mainly on the nature of the input data.

- **Mean Squared Error (MSE)**

This is the most commonly used loss function for continuous data (for example, images or time series).

The goal is to minimize the squared difference between the original input x and the reconstructed output $\hat{x}$:

$$MSE = \frac{1}{n} \sum_{i=1}^n (x_i - \hat{x}_i)^2$$

- **Binary Cross Entropy**

This is used when the input data is normalized in the range $[0, 1]$ and interpreted as probabilities, or when the data is binary. It is very common in autoencoders where the output layer has a sigmoid activation function.

$$BCE = -\frac{1}{n} \sum_{i=1}^n [x_i \log(\hat{x}_i) + (1 - x_i) \log(1 - \hat{x}_i)]$$

There are many variants of autoencoders, such as Denoising Autoencoders¹¹, Sparse Autoencoders¹², or Contractive Autoencoders¹³. From here on we will focus on **Variational AutoEncoders (VAE)**¹⁴ for their generative properties.

4.2 VAE

Conventional autoencoders map inputs to a fixed set of vectors. This means that if you sample a point in the latent space where no representation exists, the output will be unrealistic because the decoder doesn't know how to reconstruct an input from an empty region.

The novelty that VAEs introduce is that instead of mapping data to fixed vectors, they are mapped to a distribution.

Together with regularization towards a common prior, this favors a latent space that is continuous and structured. Therefore, you can choose a point z that doesn't correspond exactly to the projection of any training data, pass it through the decoder, and obtain a new, coherent sample x . That is, generating data in the form $z \rightarrow x$.

This is precisely a generative model of latent variables.

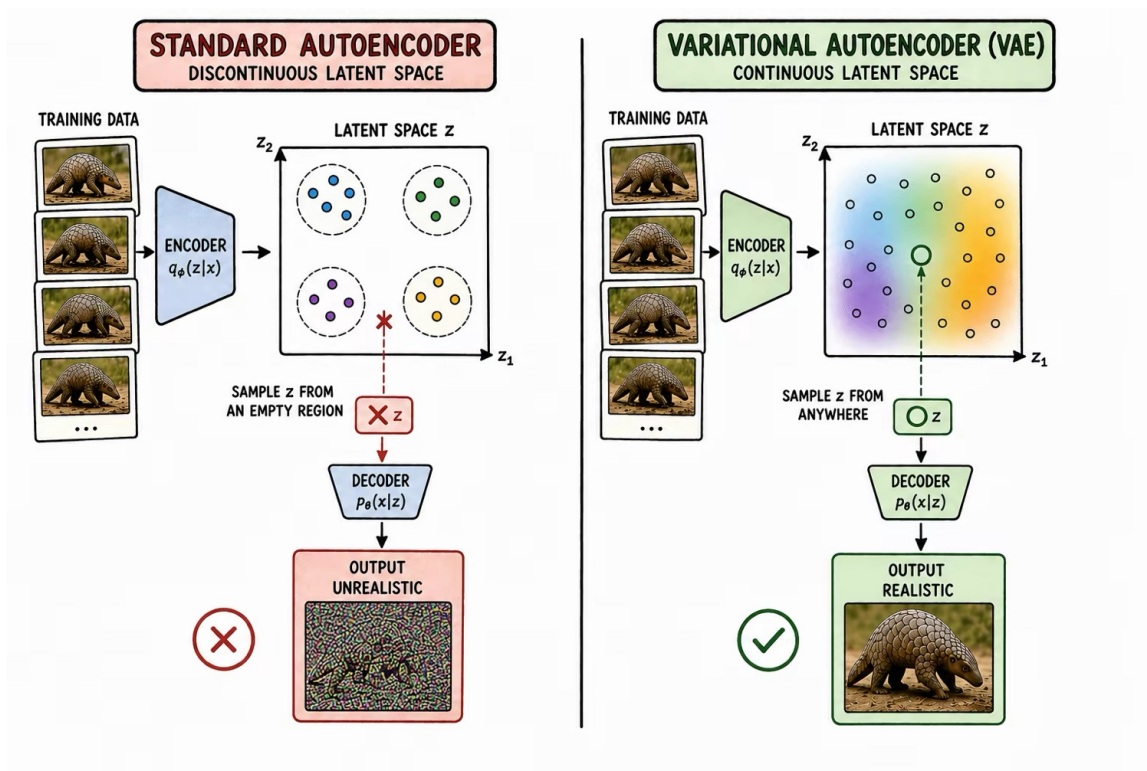


Figure 4.2. Comparison between an irregular latent space and a continuous, regularized one

The relationship between the input data x and the latent vector z is given by:

- **Prior:** $p(z)$
- **Likelihood:** $p_\theta(x | z)$
- **Posterior:** $p_\theta(z | x)$

The distribution of the data can then be modeled as:

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

Unfortunately, $p_\theta(x)$ is not easy to compute, since it is very costly to check and sum over all possible values of z (it is usually called 'intractable'). To solve this problem, VAEs resort to **variational inference**. Precisely, variational inference is one of the most widely used methods in Bayesian inference and is used to approximate intractable integrals. In other words, it is a technique used to approximate complex distributions.

The complex distribution we want to approximate is the posterior $p_\theta(z | x)$. To do this, a family of simple distributions $q_\phi(z | x)$ is assumed, and the approximation is posed as an optimization problem

in which we search within q_ϕ for the distribution that best approximates the target distribution. ϕ denotes the parameters of the chosen family of distributions.

In this way, we introduce:

- An encoder $q_\phi(z | x)$ that approximates the posterior $p_\theta(z | x)$.
- A decoder that models the conditional probability $p_\theta(x | z)$.

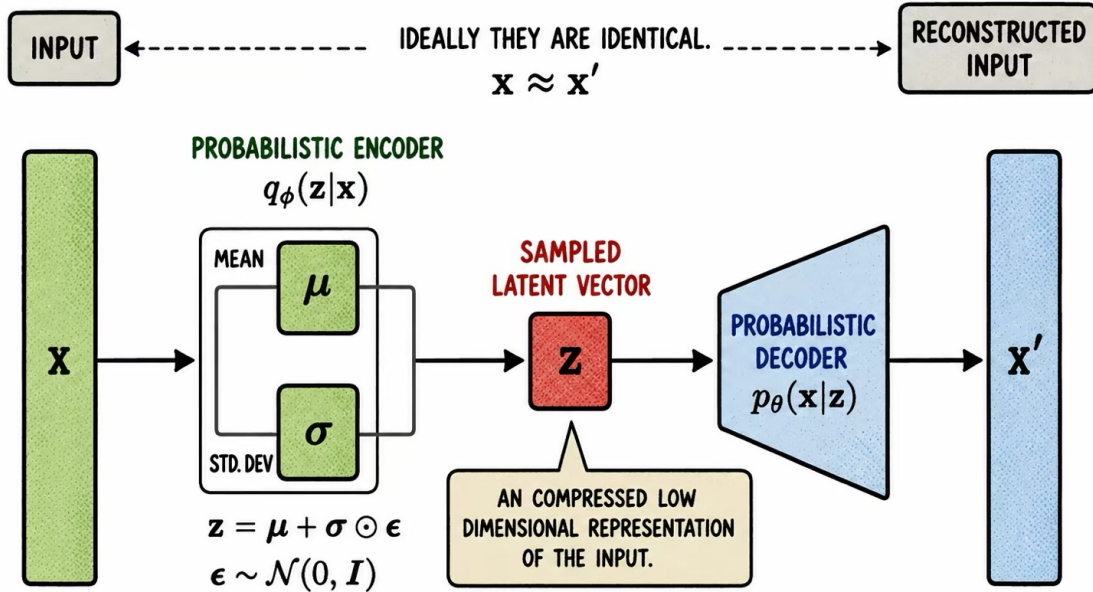


Figure 4.3. A VAE's encoder produces a mean and a variance to sample the latent variable

Note

During training we use backpropagation, but here we need to sample from the encoder's parameters ϕ to obtain z . The obstacle is that random operations cannot be differentiated.

The solution to this problem is the **reparametrization trick**. Instead of having the network directly generate the random value z , the process is split into two parts:

1. The predictable part (the network): The encoder learns the mean μ_ϕ and standard deviation σ_ϕ in a deterministic and differentiable way.
2. The random part (ϵ): An external noise source independent of the network's weights is introduced, following a standard Gaussian distribution $\epsilon \sim \mathcal{N}(0, I)$.

Once the network and the randomness are separated, they are simply combined with:

$$z = \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon$$

In this way, the randomness stays confined exclusively to ϵ , and since it is an external variable fixed during the computation, the network doesn't need to learn how to differentiate it.

4.2.1 Loss function: ELBO

The distribution estimated by the encoder, $q_\phi(z | x)$, must approximate the true posterior $p_\theta(z | x)$. To quantify the distance between these two distributions, the **Kullback–Leibler divergence** is used, $D_{\text{KL}}(Y \| X)$, which measures how much information is lost if distribution X is used to represent Y .

In this case we want to minimize $D_{\text{KL}}(q_\phi(z | x) \| p_\theta(z | x))$ with respect to ϕ .

Using Bayes' theorem, this can be rewritten as:

$$D_{\text{KL}}(q_\phi(z | x) \| p_\theta(z | x)) = \log p_\theta(x) - \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x | z)] + D_{\text{KL}}(q_\phi(z | x) \| p(z))$$

- The first term is the log-likelihood of the data, which we want to maximize.
- The second term is the expected log-likelihood of the data under the posterior approximated by the encoder.
- The third term is the KL divergence between the approximated posterior and the prior.

Combining these terms, we can define the loss function of a VAE as:

$$\mathcal{L}_{\text{ELBO}}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x | z)] - D_{\text{KL}}(q_\phi(z | x) \| p(z)),$$

known as the **Evidence Lower Bound (ELBO)**.

- The first term is the **reconstruction** of x , which tends to make the encoding-decoding scheme as efficient as possible by maximizing the log-likelihood $\log p_\theta(x | z)$ with sampling from $q_\phi(z | x)$ (encoder).
- The second term **regularizes** the latent variables z by minimizing the KL divergence between the variational approximation (encoder) and the prior distribution of z . Normally, the family of simple distributions q_ϕ we choose is a diagonal Gaussian. Therefore, when you pass an input x through the encoder, it returns the values of μ and σ for that specific input.

The ELBO is **maximized**; when we talk about the loss function we minimize its negative, $-\mathcal{L}_{\text{ELBO}}$.

4.2.2 Applications

The applications of VAEs are countless. Some examples beyond image generation are text modeling¹⁵, anomaly detection, remaining useful life estimation of industrial systems¹⁶, arrhythmia classification¹⁷, or molecular design¹⁸.

4.2.3 Posterior collapse

VAEs are known to be susceptible to **posterior collapse**. This is a problem that refers to the situation in which the posterior approximated by the encoder collapses onto the prior. That is, the encoder learns $q_\phi(z | x) \approx p(z)$ for any input x . This happens because the KL $\rightarrow$ the second term of the ELBO, is reduced to zero and the latent variables stop encoding information about the input data.

As a consequence, the decoder learns to ignore z entirely and generates reconstructions based solely on global statistics of the data, nullifying the usefulness of the latent space.

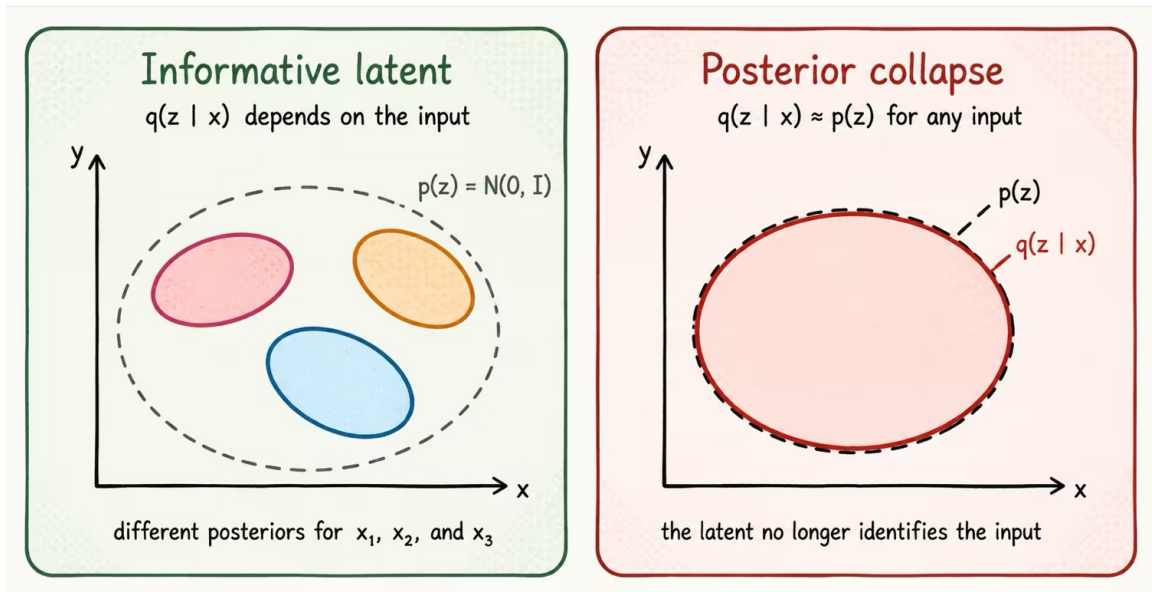


Figure 4.4. Comparison between an informative posterior and one that matches the prior during posterior collapse

4.2.4 Training tricks

There are several approaches to solving this problem, including:

- **KL Divergence Annealing:** gradually increases the weight of the KL divergence term in the loss function during training, allowing the model to first learn to reconstruct the data before

imposing strong regularization on the latent space.

- **Free Bits:** sets a minimum threshold for the KL divergence in each dimension of the latent variable, ensuring that each one contributes at least a "minimum amount of information" and preventing them from collapsing to trivial values.
- **Minimum Desired Rate:** similar to Free Bits, but applies a penalty to maintain a minimum amount of information in the latent space, forcing the compression not to lose relevance.
- **Limiting the decoder's capacity or adding connections from z :** reduces the possibility of the decoder explaining the data without resorting to the latent variable.

4.3 VAE Variants

VAEs trained with a pixel-by-pixel reconstruction loss tend to produce blurry images. Why does this happen?



Figure 4.5. Blurry reconstructions from a VAE

Typically, metrics such as RMSE or BCE are used to quantify the quality of the reconstructions, as we mentioned earlier. Therefore, the network's goal is to minimize these metrics.

The problem arises when, for the same latent representation, several plausible reconstructions exist. In that case, to minimize a pixel-by-pixel loss, the network can make a kind of cheat: predicting intermediate values between those alternatives. In an image, that conditional average between possibilities tends to produce blurry edges and details.

4.3.1 β -VAE

One way to control the balance between reconstruction quality and latent space regularization is to introduce a hyperparameter β that adjusts the weight of the KL term:

$$\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x | z)] - \beta D_{\text{KL}}(q_\phi(z | x) \| p(z))$$

This is what the β -VAE¹⁹ consists of.

If $\beta = 1$ is set, the objective function is the same one used in standard VAEs; if it is set to 0, the KL regularization disappears and the objective is reduced to the reconstruction term, bringing the behavior closer to that of a conventional autoencoder.

If $\beta < 1$ is used, more bits are stored about each input (that is, more importance is given to the data) and, therefore, images can be reconstructed in a less blurry way. On the other hand, an advantage of using $\beta > 1$ is that it encourages learning a "disentangled" latent representation. Intuitively, this means that each latent dimension represents a different factor/feature of the input data. For example, a model trained on photos of human faces might capture smoothness, skin color, hair color or length, emotion, whether glasses are worn, and many other relatively independent factors in separate dimensions.

In summary: designing a VAE is almost always a balancing act between "I want the reconstruction to look good" and "I want the latent space to make sense."

4.3.2 InfoVAE

To solve the problem of placing too much emphasis on the second term (regularization), causing latent representations that don't capture the structure of the data well and generated samples that lack diversity, **InfoVAE**²⁰ modifies the loss function by adding a **mutual information maximization** term between the latent variables z and the observed data x . The InfoVAE loss function can be written as:

$$\mathcal{L}_{\text{InfoVAE}} = \mathbb{E}_{p_{\text{data}}(x)} \left[\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x | z)] - (1 - \alpha) D_{\text{KL}}(q_\phi(z | x) \| p(z)) \right] - (\alpha + \lambda - 1) D_{\text{KL}}(q_\phi(z) \| p(z))$$

Where the first term is the reconstruction error, the second is the KL divergence between the approximated posterior distribution and the prior, and the third is the KL divergence between the marginal distribution of the latent variables and the prior distribution. The distribution $q_\phi(z) = \int q_\phi(z | x) p_{\text{data}}(x) dx$ is known as the **aggregated posterior** and describes how the encodings of the entire dataset are distributed in the latent space.

α and λ are hyperparameters that control the relative importance of each term. In this way, the objective favors z retaining information about x while still approximating the aggregated posterior to the prior.

4.3.3 Multimodal VAEs

Multimodal VAEs^{21, 22} are an extension of the standard VAE model to handle and learn joint representations of data coming from different modalities. These modalities can include text, images,

audio, video, etc. The main idea is to capture the dependencies and correlations between different types of data, learning a shared latent representation that can be used for tasks such as data generation or imputation.

Several approaches have been proposed so far for multimodal VAE learning. [This review](#) gathers some of the most influential works.

4.3.4 Hierarchical VAEs

One limitation of standard VAEs is that they usually use a simple prior, such as a standard Gaussian. One way to increase their flexibility is to introduce a hierarchy of latent variables. By making both the inference model (encoder) and the generative model (decoder) hierarchical, more complex distributions can be represented.

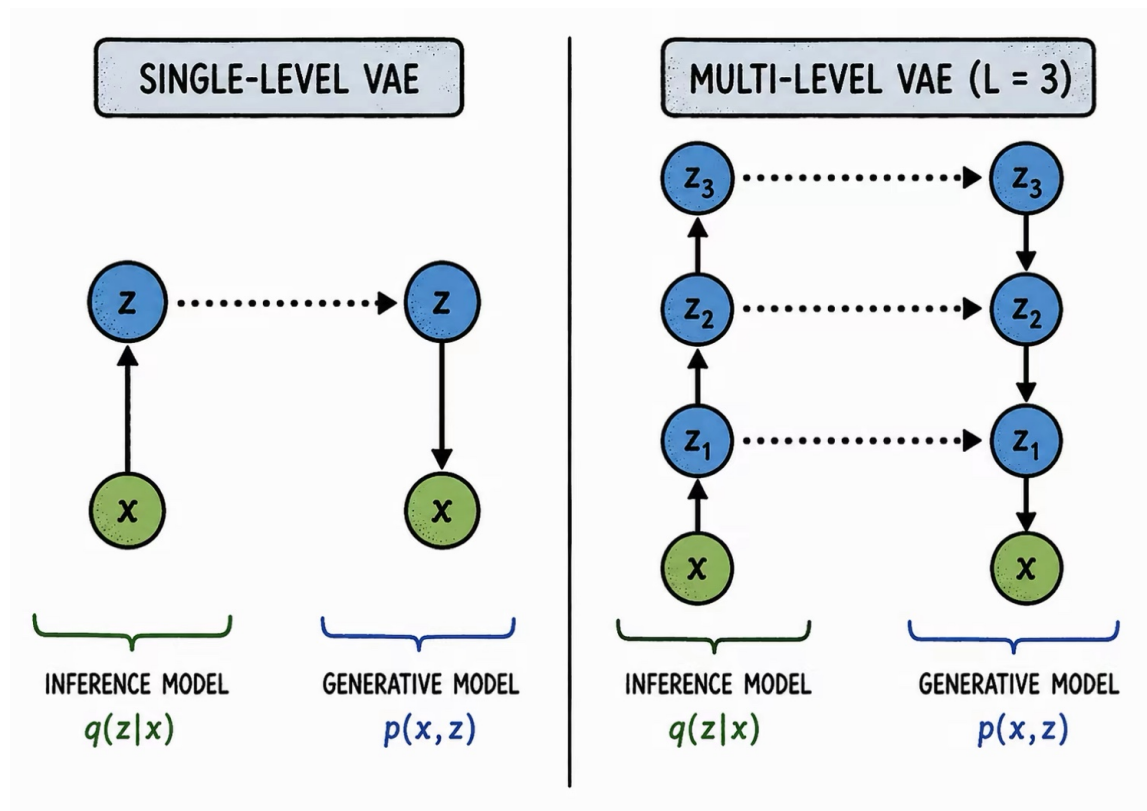


Figure 4.6. Hierarchical VAE with latent variables connected at different scales

The distribution we want to model then is:

$$p(x) = \int p(x | z_1) p(z_1 | z_2) \cdots p(z_k) dz_{1:k},$$

Therefore, the ELBO would be expressed as follows:

$$\mathcal{L}_{\text{ELBO}} = \mathbb{E}_{q(z_{1:k}|x)}[\log p(x | z_1)] - D_{\text{KL}}(q(z_{1:k} | x) \| p(z_{1:k}))$$

Many hierarchical models have been published, among which **VD-VAE (Very Deep VAE)**²³ stands out for image generation, and **Bit-Swap**²⁴ for data compression.

The VD-VAE architecture is a convolutional VAE. For each group of latent variables, the prior and posterior are diagonal Gaussians. In the decoder, it uses nearest-neighbor resampling instead of transposed convolutions and adds several design decisions to stabilize training. Together, this makes it possible to train a deep hierarchy using the original ELBO, without modifying the objective function.

4.4 Extra: Compression

In **Bit-swap**²⁴, a hierarchical VAE is presented, demonstrating that the bits-back compression scheme (BB-ANS) can be used with latent variable models. That is, this type of generative model can be used as efficient compression schemes. Let's take a look, but first: a bit of information theory.

4.4.1 Kolmogorov complexity

In information theory, the **Kolmogorov complexity** of an object x , denoted $K(x)$, refers to the length of the shortest possible description of that object on a 'universal computer'. It represents the inherent complexity of a piece of information x as a function of the length of the most concise binary program capable of generating it.

- A shorter binary program implies a higher level of regularity or predictability in the data.
- A longer binary program suggests a more complex, less compressible structure.

Similarly, the Kolmogorov complexity of x given y , $K(x|y)$, corresponds to the length of the binary program that, receiving y as input, produces x . Based on this idea, Bennett et al.²⁵ derived the **information distance** metric, $E(x, y)$:

$$E(x, y) = \max\{K(x | y), K(y | x)\} = K(x, y) - \min\{K(x), K(y)\} + O(\log K)$$

This metric essentially evaluates the existence of a simple program capable of transforming one object into another and, therefore, provides a measure of the relationship or amount of information shared between them. The simpler the conversion program, the more similar the objects are considered to be. However, Kolmogorov complexity is not theoretically computable but can only be approximated.

4.4.2 Compression

Compression techniques approximate Kolmogorov complexity, since they offer a practical means of representing data more concisely without losing essential information. The main goal of a lossless

compressor is to minimize the number of bits needed to represent the data, subsequently allowing its exact decompression.

Recommendation

Silicon Valley is a fun way to understand the importance of compression in society.

Compression methods achieve this goal by identifying and exploiting redundancies or patterns present in the data. To achieve the shortest possible compression length, symbols with higher probability receive shorter codes. According to Shannon's Source Coding Theorem¹, the length of the encoded bits is directly related to the entropy of the information source:

Shannon entropy

$$H(X) = - \sum_{x \in \mathcal{X}} p(x) \log_2 p(x)$$

What compression algorithms seek is to effectively model the "real" distribution of the data $p(x)$:

- A common strategy consists of dictionary-based methods, where patterns or sequences of data are identified and replaced with shorter codes. Among these are the Lempel-Ziv algorithms, which dynamically build a dictionary during compression²⁶.
- Context-based models, such as adaptive arithmetic coding, adjust the encoding probabilities based on the context of previously encoded symbols²⁷.
- Entropy coding methods, such as Huffman coding, assign shorter prefix codes to the most probable symbols²⁸.

Recent advances have incorporated Machine Learning techniques into compression algorithms. What models can we use to approximate the real distribution of the data?

→ **Exactly, VAEs!!**

4.4.3 Generative models as compressors

In a typical compression pipeline, data is represented as a sequence of symbols $X = x_1, x_2, \dots, x_n$ and encoded into a set of bits C . Afterwards, the inverse process is applied to recover the original data.

- Compression: $E(X) \rightarrow C$
- Decompression: $D(C) \rightarrow X'$

Lossless compression methods seek to minimize the number of bits without sacrificing information. For a symbol with probability $p(x)$, the ideal length is $-\log_2 p(x)$ bits.

The better the compression, the better the approximation of $p(x)$. In a probabilistic context, $p(x)$

can be expressed as the probability of observing the data under a given model:

$$p(x) = \int p(x|\theta)p(\theta)d\theta$$

where $p(x|\theta)$ is the probability of observing the data given a specific set of parameters, $p(\theta)$ is the prior distribution, and θ represents the model's parameters.

In the context of generative modeling, this expression is usually formulated in terms of the latent variable z :

$$p(x) = \int p(x|z)p(z)dz$$

Sounds familiar, right?

Using this notation, the process of sending information in a compression scheme ($E(X) \rightarrow C$) proceeds as follows:

1. The latent variable z is encoded using $p(z)$.
2. The sample x is encoded using $p(x|z)$.
3. Both z and x are transmitted.

On the other hand, receiving the information ($D(C) \rightarrow X'$) follows these steps:

1. z is decoded using $p(z)$.
2. x is decoded using $p(x|z)$.

The number of bits involved initially corresponds to $-\log_2 p(z) - \log_2 p(x | z)$. The **bits-back** method allows recovering approximately $-\log_2 q(z | x)$ bits used to select z ²⁹. This idea connects variational inference with probabilistic compression.

The compression process described above can be interpreted according to the BB-ANS scheme³⁰. Some extra bits are used to generate a sample z via the encoder $q(z|x)$; afterwards, z is used to encode x through $p(x|z)$, while the variable z itself is encoded using the prior distribution $p(z)$. The reverse process allows recovering z via $p(z)$, reconstructing x via $p(x|z)$, and recovering the initial extra bits using $q(z|x)$.

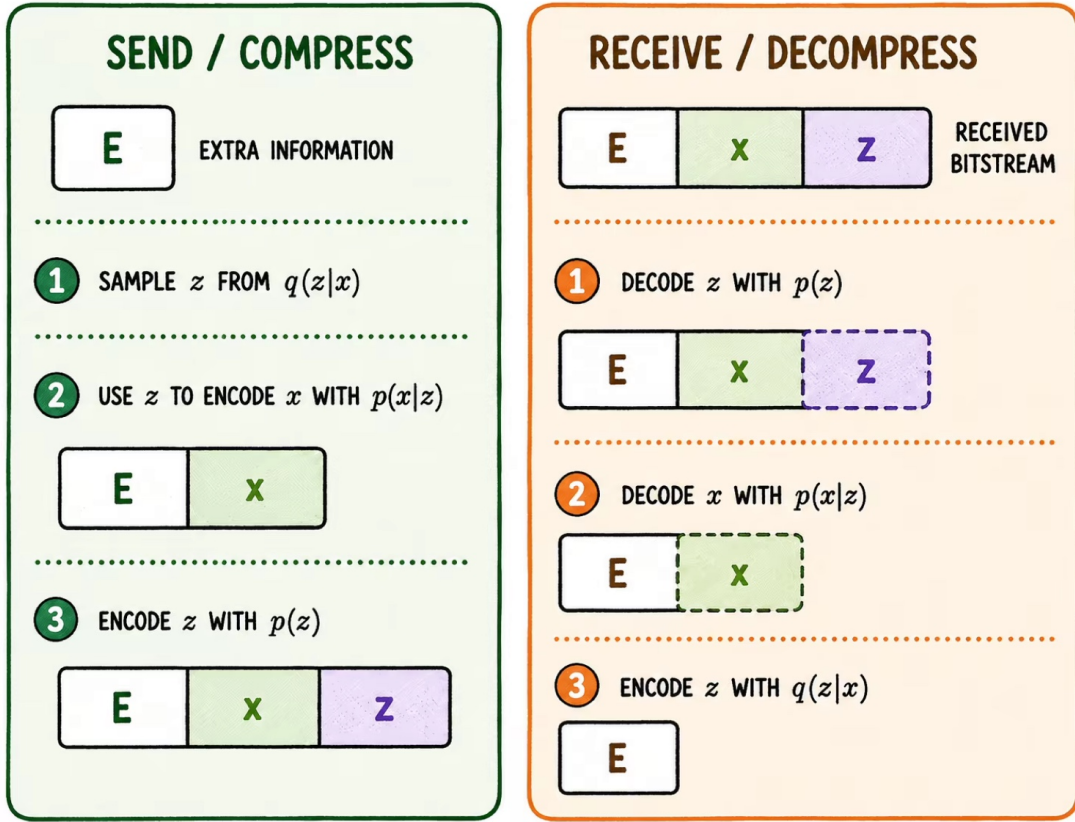


Figure 4.7. Bits-back encoding scheme with ANS for a latent-variable model

The final length of the bit stream for a single data point is given by:

$$N = n_{\text{extra}} + \log_2 q(z | x) - \log_2 p(x | z) - \log_2 p(z)$$

Notice that the costs $-\log_2 p(z)$ and $-\log_2 p(x | z)$ are transmitted, but the bits-back mechanism allows recovering approximately the $-\log_2 q(z | x)$ extra bits used to select z . **What matters here** is that $N - n_{\text{extra}}$ coincides with the negative ELBO (negative Evidence Lower Bound, nELBO), expressed in bits:

$$\begin{aligned} \mathbb{E}_{q(z|x)}[N - n_{\text{extra}}] &= \mathbb{E}_{q(z|x)} \left[\log_2 \frac{q(z | x)}{p(x, z)} \right] \\ &= -\mathcal{L}_{\text{ELBO}} / \log 2 \end{aligned}$$

Dividing by $\log 2$ converts nats into bits. This cost corresponds to the loss function we minimize in a VAE.

$$\begin{aligned}
\mathcal{L}_{\text{ELBO}} &= \mathbb{E}_{q(z|x)} \left[\log \frac{p(x, z)}{q(z | x)} \right] \\
&= \mathbb{E}_{q(z|x)} \left[\log \frac{p(x | z)p(z)}{q(z | x)} \right] \\
&= \mathbb{E}_{q(z|x)} [\log p(x | z)] + \mathbb{E}_{q(z|x)} \left[\log \frac{p(z)}{q(z | x)} \right] \\
&= \mathbb{E}_{q(z|x)} [\log p(x | z)] - D_{\text{KL}}(q(z | x) \| p(z))
\end{aligned}$$

- The first term corresponds to the reconstruction of x and favors an efficient encoding and decoding scheme by maximizing $\log p(x | z)$ with samples from $q(z | x)$.
- The second term is the regularization of the latent space by minimizing the KL divergence between the variational approximation and the prior distribution z .

As a result, minimizing the negative ELBO maximizes a lower bound of $\log p(x)$ and brings $q(z | x)$ closer to the model's posterior $p(z | x)$.

This equivalence demonstrates that a properly optimized latent variable model, such as a VAE, is directly applicable to compression tasks, since it minimizes the achievable code length in the context of bits-back coding.

Jiang et al., in NPC (*Non-Parametric learning by Compression*)³¹, proposed a method based on this premise for image classification, outperforming supervised and semi-supervised methods.

Similarly, in *Few-shot generative compression approach for system health monitoring*³² **I proposed a method based on compression and VAEs to diagnose cases where monitoring data is abundant, but there are very few labeled instances.**

CHAPTER 5

Flow-based Models

Chapter code: github.com/NahuelCostaCortez/generative-networks

Generative models differ in how they relate to the density of the data $p(x)$. **GANs** are implicit density models: they do not compute or optimize any likelihood, they only learn to sample through an adversarial game. **VAEs** are likelihood-based models, but their marginal likelihood is intractable and is optimized via a lower bound (the ELBO). **Flow-based models**, on the other hand, allow this density to be computed exactly and tractably.

As we already saw, a typical generative model with latent variables is defined by:

$$p(x) = \int p(x | z) p(z) dz,$$

and computing $p(x)$ is intractable because it is practically impossible to iterate over all values of z .

Flow-based models approach this problem through invertible transformations of distributions. There are two major families: classical **normalizing flows**, which apply a discrete sequence of invertible transformations, and **continuous normalizing flows (CNF)**, where the transformation is defined as the solution of an ordinary differential equation (ODE); we will look at the latter further on.

A good estimate of $p(x)$ makes it possible to efficiently carry out many tasks, such as sampling new, unobserved but realistic data points (data generation), inferring latent variables, or filling in incomplete data samples. Their use for detecting out-of-distribution (OOD) data has also been explored.

Note

A high likelihood does not guarantee that a sample belongs to the training distribution. In fact, it has been observed that these models can assign higher density to OOD data than to the training data itself³³.

5.1 Normalizing flows

Since we use backpropagation to train deep learning models, we expect the posterior probability distribution $p(z | x)$ to be simple enough that its derivative can be computed easily and efficiently. This is why the Gaussian distribution is usually used in latent-variable generative models, even though most real-world distributions are much more complicated than the Gaussian.

In a normalizing flow, to approximate the real distribution we start from a simple distribution $p_0(z_0)$, to which a series of invertible transformations is applied to turn it into a complex one $p_K(z_K)$. The variable z_i is repeatedly replaced by the new z_{i+1} to finally obtain a probability distribution matching the final target variable.

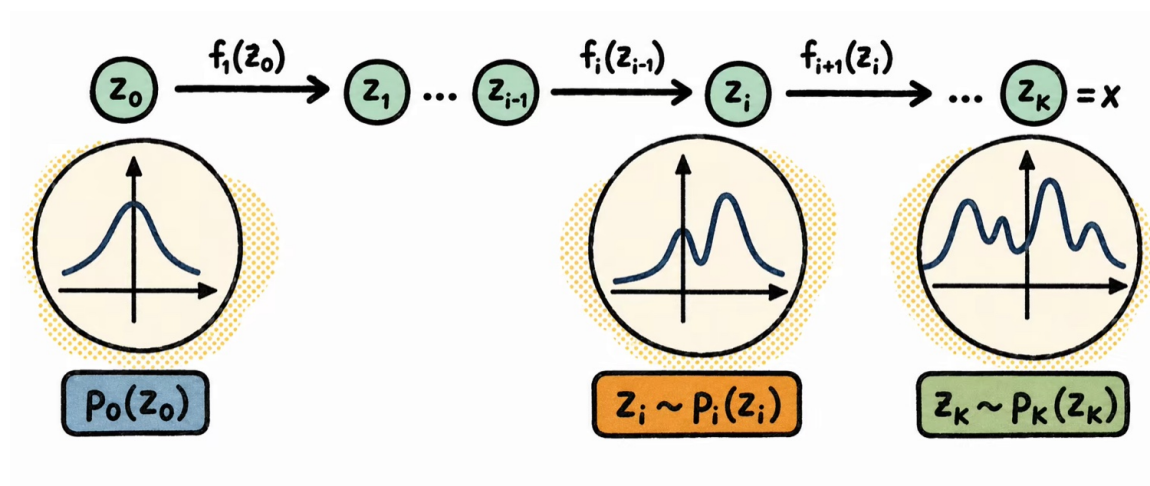


Figure 5.1. Sequence of invertible transformations from a base distribution to a complex distribution

Both the forward transformation and its inverse can be computed exactly. This makes density estimation possible. To do this, two things must be taken into account:

- **The density of the inversely transformed sample:** to obtain the transformed sample, the sequence of inverse transformations is applied to the original sample. Then, the density of this transformed sample is evaluated under the original simple distribution. That is, we check whether the resulting transformation is a Gaussian.
- **The change of volume due to the transformations:** as the transformations are carried out, the sample space is distorted. The change of volume is calculated by multiplying the absolute values of the determinants of the Jacobian matrices of each transformation.

By multiplying these two values (the density of the transformed sample and the change of volume), we obtain the density of the original sample under the complex distribution.

5.2 Applications

The most direct applications of normalizing flows are:

- **Density estimation:** to compute the exact density of the data (the one assigned by the model). They can be applied to fit multimodal densities to observed data. They can also be used as hybrid models that model the joint density of inputs and targets $p(x, y)$, unlike classification models, which only model $p(y | x)$, and density models, which only model $p(x)$. This is useful for tasks such as anomaly detection.
- **Data generation:** for different data modalities, including images, video, audio, text, and structured objects such as graphs.
- **Inference:** instead of blindly trying combinations in a simulator or digital twin until finding which parameters are compatible with an observed result, the flow learns to take the reverse path: starting from an observation, it approximates the posterior distribution over the parameters that could have generated it, without needing to run through the simulator thousands of times. This approach saves months of computation in fields such as cosmology³⁴ and computational neuroscience³⁵.

5.3 Basic linear algebra concepts

5.3.1 Jacobian matrix

Given a function mapping an n -dimensional input vector to an m -dimensional output vector, the matrix of all the first-order partial derivatives of this function is called the **Jacobian matrix**:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

It can be understood as a translator from one vector space to another.

Reminder

Partial derivatives measure how a function changes with respect to a specific variable, holding the others constant.

For example, if we have a function:

$$f(x, y) = x^2y + 3y$$

the partial derivative with respect to x is computed by treating y as a constant:

$$\frac{\partial f}{\partial x} = 2xy$$

and the partial derivative with respect to y is computed by treating x as a constant:

$$\frac{\partial f}{\partial y} = x^2 + 3$$

The symbol ∂ is used to distinguish them from ordinary derivatives.

A partial derivative is first-order when we differentiate the function only once with respect to a variable (as in the example above). If we differentiate a partial derivative again, we obtain a second-order derivative.

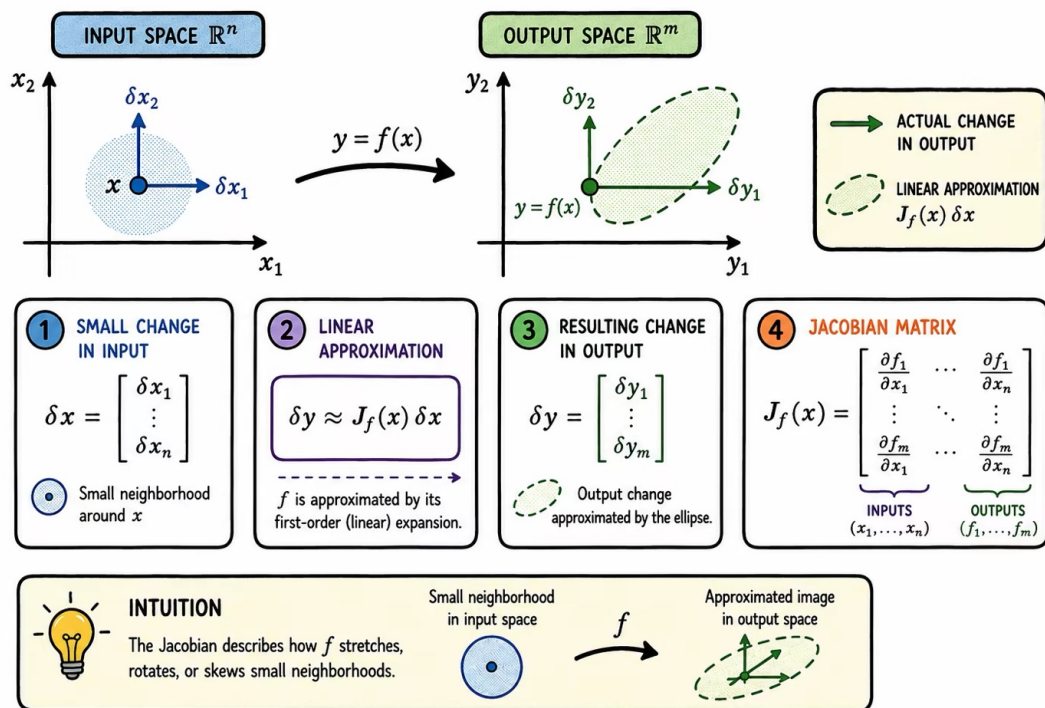


Figure 5.2. Jacobian matrix formed by the partial derivatives of a vector transformation

5.3.2 Determinant

The absolute value of the determinant (which only exists for square matrices) can be thought of as a measure of "how much multiplication by the matrix expands or contracts space."

The determinant of a square matrix M detects whether it is invertible:

- if $\det(M) = 0$ then it is not invertible (a singular matrix with linearly dependent rows or columns;

- or any row or column that is all 0);
- if $\det(M) \neq 0$, then M is invertible.

The determinant of a product is equivalent to the product of the determinants: $\det(AB) = \det(A)\det(B)$.

5.3.3 Change of variables theorem

When we transform a random variable, its **probability does not change**, but the space it lives in can stretch or compress.

Imagine a simple variable z with known density $p_Z(z)$, for example a Gaussian. Now we apply an invertible function:

$$x = f(z)$$

Since f is invertible, if we know a value of x we can also recover the value of z that produced it:

$$z = f^{-1}(x)$$

The question is: if we know the density in the original space, $p_Z(z)$, how do we compute the density in the new space, $p_X(x)$?

At first glance, one might think it is:

$$p_X(x) = p_Z(f^{-1}(x))$$

This tells us where x comes from, BUT the transformation may have changed the size of the regions of space.

Intuition

If a transformation stretches a region, the same probability ends up spread over a larger region. Therefore, the density decreases.

If a transformation compresses a region, the same probability ends up concentrated in a smaller region. Therefore, the density increases.

In one dimension, we can think of it in terms of small intervals. A little piece around z has length dz . After applying the transformation, that piece becomes another one around x with length dx .

The probability inside both pieces must be the same:

$$p_X(x) dx = p_Z(z) dz$$

Rearranging:

$$p_X(x) = p_Z(z) \left| \frac{dz}{dx} \right|$$

And since $z = f^{-1}(x)$:

$$p_X(x) = p_Z(f^{-1}(x)) \left| \frac{df^{-1}}{dx} \right|$$

The absolute value appears because a density cannot be negative. If the function flips the axis, for example from left to right, the derivative can be negative, but the size-change factor remains positive.

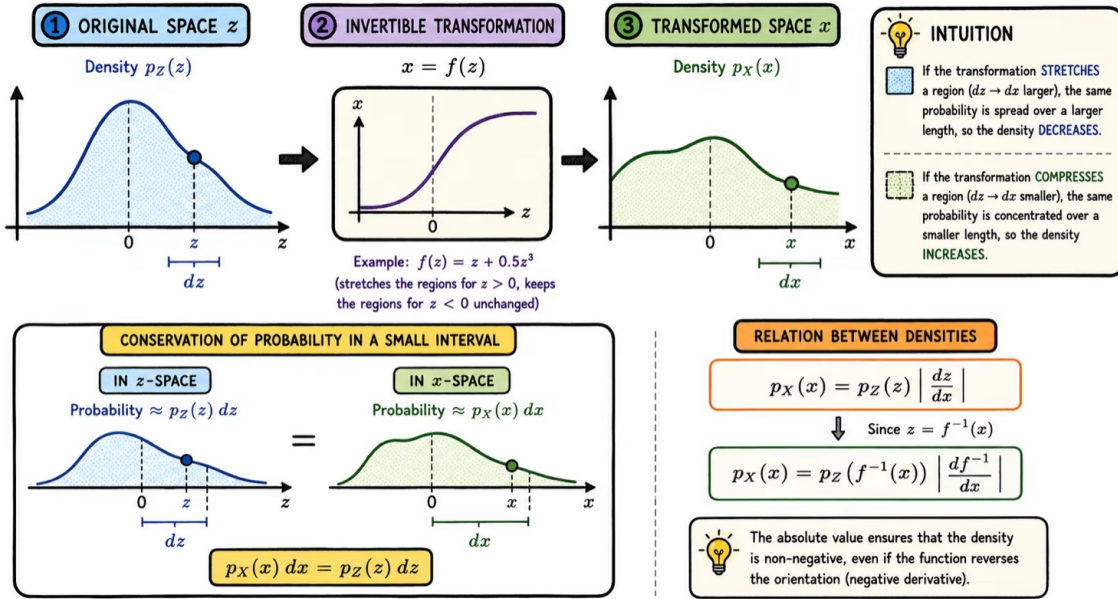


Figure 5.3. A transformation stretches and compresses regions, changing their density

In several dimensions the same thing happens, but the "little pieces" are no longer intervals, but small areas, volumes, or hypervolumes. This is where the Jacobian determinant comes in:

$$p_X(x) = p_Z(f^{-1}(x)) |\det J_{f^{-1}}(x)|$$

Where $J_{f^{-1}}(x)$ is the Jacobian matrix of the inverse function. Its determinant measures how much the volume locally expands or contracts when going from x back to z .

This is also often written using the forward transformation f :

$$p_X(x) = p_Z(z) |\det J_f(z)|^{-1}$$

This form is equivalent. If f stretches the volume by a factor of 3, then the density is divided by 3. If it compresses it by a factor of 3, then the density is multiplied by 3.

5.4 Application to normalizing flows

A normalizing flow does not perform a single transformation, but a chain of invertible transformations:

$$z_0 \xrightarrow{f_1} z_1 \xrightarrow{f_2} z_2 \xrightarrow{f_3} \dots \xrightarrow{f_K} z_K = x$$

We normally start with a simple distribution:

$$z_0 \sim p_0(z_0)$$

Then we apply transformations until we reach a sample that looks like real data:

$$x = z_K = f_K \circ f_{K-1} \circ \dots \circ f_1(z_0)$$

This is used to **generate** data: we sample a simple z_0 and progressively transform it to obtain x .

But the interesting thing about flows is that we can also take the reverse path to compute the exact density of a data point x :

$$x = z_K \xrightarrow{f_K^{-1}} z_{K-1} \xrightarrow{f_{K-1}^{-1}} \dots \xrightarrow{f_1^{-1}} z_0$$

Once we have z_0 , its density is easy to compute because it belongs to the initial distribution. All that remains is to correct for all the volume changes that occurred along the way.

For a single transformation:

$$z_i = f_i(z_{i-1})$$

Applying the change of variables theorem:

$$p_i(z_i) = p_{i-1}(z_{i-1}) \left| \det J_{f_i}(z_{i-1}) \right|^{-1}$$

This equation says:

- the density after the transformation depends on the density before the transformation
- if the transformation expands the space, the density decreases
- if the transformation compresses the space, the density increases

As we mentioned in the GAN chapter, in machine learning we usually work with logarithms of probabilities because:

- products of many small numbers tend to zero
- products become sums, which are easier to optimize

Taking logarithms:

$$\log p_i(z_i) = \log p_{i-1}(z_{i-1}) - \log \left| \det J_{f_i}(z_{i-1}) \right|$$

If we repeat this for the K transformations of the flow, we obtain:

$$\log p_X(x) = \log p_0(z_0) - \sum_{i=1}^K \log |\det J_{f_i}(z_{i-1})|$$

This is the central formula of normalizing flows.

It reads as follows:

- $\log p_X(x)$: the log-density of the real data point we want to evaluate
- $\log p_0(z_0)$: the log-density of the corresponding point in the simple distribution
- $\sum_i \log |\det J_{f_i}|$: the accumulated correction for all the volume changes introduced by the transformations

The path traveled by the random variables $z_i = f_i(z_{i-1})$ is the **flow**. The complete chain of successive distributions is what we call a **normalizing flow**.

Required properties for f

The transformation functions f_i must satisfy these properties:

1. Bijection: they must be **invertible**.
2. Their **Jacobian determinant must be easy to compute**.

5.5 Models based on Normalizing flows

Some popular normalizing flow architectures are Real NVP³⁶, Masked Autoregressive Flows³⁷, Glow³⁸ (by Kingma → author of the original VAE paper and creator of the Adam optimizer), and SurVAE³⁹.

The differences between these architectures lie in the transformations they apply and in specific network designs, but they all share the common goal of transforming a simple distribution into a more complex one.

5.6 Advantages and disadvantages

Advantages

- They allow computing the exact density (the one assigned by the model) of a data point, not just generating samples. This makes them useful for tasks such as anomaly detection, density estimation, or models where we want to explicitly know what probability the model assigns to an observation.
- Since the transformation is invertible, we can go in both directions: from noise to data to generate,

and from data to noise to evaluate density.

Limitations

- Each transformation must be invertible and have a Jacobian determinant that is easy to compute. In standard neural networks, most operations (such as ReLU activations or pooling transformations) are not invertible. This greatly restricts layer design and can make the model less flexible than other generative architectures.
- Computing the determinant of a dense $N \times N$ matrix has a complexity of $O(N^3)$. To avoid this, architectures are designed so that the Jacobian matrix is triangular (where the determinant is just the product of the diagonal), but this greatly restricts how the data can interact with each other, requiring much deeper and slower architectures to compensate for that loss of modeling capacity.

5.7 Flow Matching

As we have seen, normalizing flows build the model as a chain of invertible transformations:

$$z_0 \xrightarrow{f_1} z_1 \xrightarrow{f_2} \dots \xrightarrow{f_K} x$$

Flow Matching starts from a similar idea, but instead of asking:

What invertible transformations can I design to turn noise into data?

it asks:

What direction should each point follow to move from a simple distribution to the target distribution?

In this way, what Flow Matching learns is a **velocity field**.

Imagine we have many noise points, for example samples from a Gaussian, and many real points, for example images. We want to gradually move the noise points until they end up resembling the real data.

To describe this movement we introduce a time variable:

$$t \in [0, 1]$$

When $t = 0$, we are at the simple distribution:

$$x_0 \sim p_0$$

When $t = 1$, we want to be at the data distribution:

$$x_1 \sim p_{\text{data}}$$

In between we have intermediate points x_t . The model learns a function:

$$v_\theta(x_t, t)$$

This function receives two things:

- the current point x_t
- the current time t

And returns a vector indicating **where that point should move at that instant**.

Intuition

In normalizing flows we learned transformations.

In Flow Matching we learn velocities.

Generation consists of starting at noise and following those velocities until reaching a realistic sample.

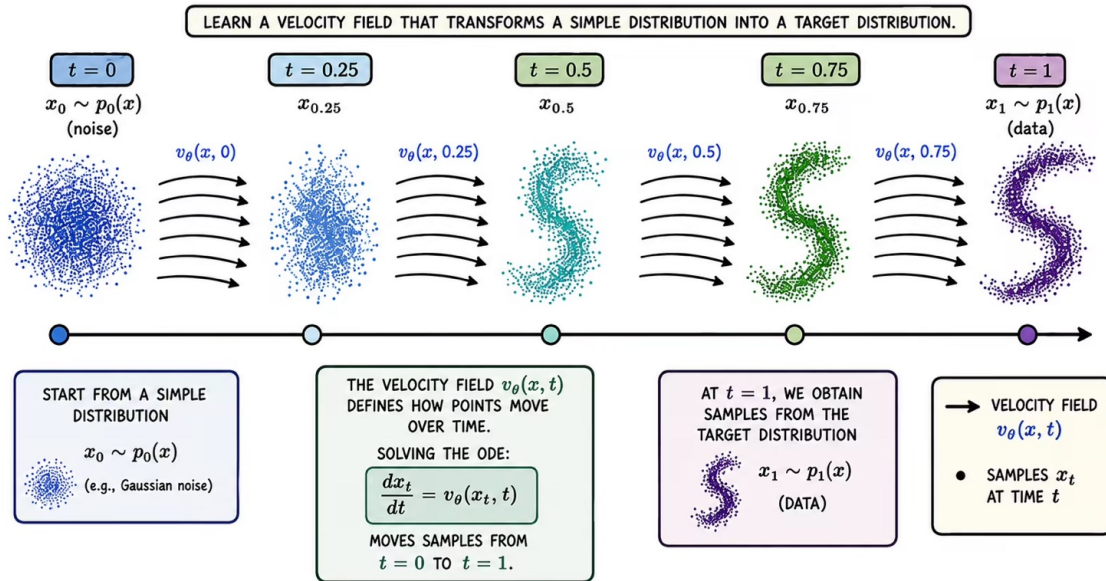


Figure 5.4. Velocity field transporting noise samples toward the data distribution

5.7.1 Training

To train the model we need to know what a good velocity would be at different intermediate points. To do this, we can pair a noise point z with a real data point x and draw a line between them:

$$x_t = (1 - t)z + tx$$

This equation represents an interpolation where:

- if $t = 0$, then $x_t = z$
- if $t = 1$, then $x_t = x$
- if t is between 0 and 1, then x_t is between the noise and the data

If the path is a straight line, the velocity that leads from z to x is:

$$u_t = x - z$$

Therefore, during training we can do the following:

1. Take a noise sample z
2. Take a real data point x
3. Choose a random time t between 0 and 1
4. Construct the intermediate point $x_t = (1 - t)z + tx$
5. Ask the network to predict the correct velocity $u_t = x - z$

The loss function can be written as:

$$\mathcal{L}(\theta) = \mathbb{E}_{z,x,t} [\|v_\theta(x_t, t) - u_t\|^2]$$

The idea is fairly direct: the network predicts an arrow, which is penalized if it points in a direction different from the one it should.

Important

Linear interpolation is a simple way to understand Flow Matching, but it is not the only one. In real models, more sophisticated paths between noise and data can be defined.

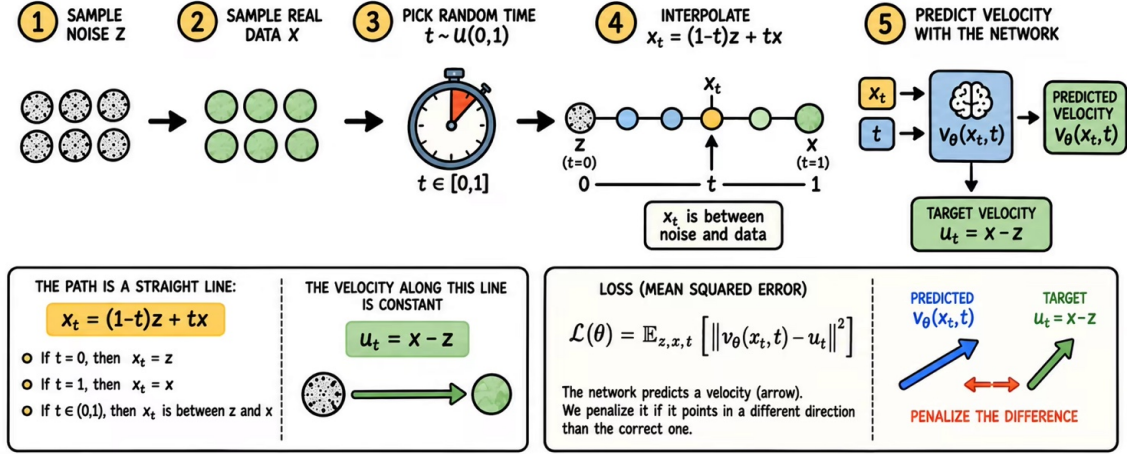


Figure 5.5. Summary of training and sampling via Flow Matching

5.7.2 Sample generation

Once the model is trained, generating a sample consists of solving an ordinary differential equation (ODE):

$$\frac{dx_t}{dt} = v_\theta(x_t, t)$$

This roughly means: "update x_t following the velocity predicted by the network at each instant." In practice:

1. We sample an initial noise point $x_0 \sim p_0$
2. We use the network to know where to move it
3. We take a small step in that direction
4. We repeat the process from $t = 0$ to $t = 1$

At the end we obtain x_1 , which should look like a sample from the data distribution.

5.7.3 Difference from Normalizing Flows

Normalizing flows and Flow Matching share the same intuition: both transform a simple distribution into a complex distribution. The difference lies in how they do it.

In **normalizing flows**:

- we use a finite sequence of invertible transformations

- we compute how the density changes with Jacobian determinants
- we can evaluate the density assigned by the model

In **Flow Matching**:

- we learn a continuous velocity field
- we generate data by following a trajectory from noise to data
- we do not need to manually design invertible layers with easy determinants

This is why Flow Matching is appealing: it allows the use of more flexible neural networks and training them with a relatively simple regression loss.

5.7.4 Continuous Normalizing Flows

Until now, when we talked about Normalizing Flows we were referring to discrete flows. However, there are also continuous flows, where, instead of applying a chain of discrete layers, the transformation is modeled as a continuous process in time governed by an ODE.

The problem is that training these flows required "simulating" the ODE forward and backward within each optimization step (known as simulation-based training or the use of Neural ODEs), which is computationally exhausting and very slow.

In this sense, the training reformulation proposed by Flow Matching (direct, linear probability paths and a simple regression loss) makes it possible to solve this problem during **training**: it is no longer necessary to simulate the ODE at every optimization step.

Note

This does not eliminate the cost in all cases: if afterward we want to **evaluate the exact likelihood** of a data point under the trained model, we still need to solve the ODE and integrate the divergence of the velocity field along the entire trajectory, just as in any CNF. What Flow Matching avoids is simulation during training, not the cost of evaluating the exact density afterward.

5.7.5 Optimal Transport

Once the central idea of Flow Matching is understood, it makes sense to talk about **Optimal Transport** as a way of choosing better paths between the initial distribution and the data distribution.

The intuition behind Optimal Transport is to transform one distribution into another **at the lowest possible cost**.

In Flow Matching this comes up when we construct paths between noise points and real points. If we pair points at random, the trajectories can cross a lot or take unnecessary detours. If we use the idea of optimal transport, we try to pair points in a more coherent way.

This is why variants such as **Optimal Transport Flow Matching** or **OT-CFM** appear in the literature. The original paper [Flow Matching for Generative Modeling](#)⁴⁰ already highlights the use of optimal-transport-based paths as a particularly interesting option, and later works such as

__Improving and Generalizing Flow-Based Generative Models with Minibatch Optimal Transport__⁴¹
 use optimal transport on mini-batches to build more useful pairings during training.

What matters is not to confuse the levels:

- **Flow Matching:** learns a velocity field that moves samples from noise to data.
- **Optimal Transport:** can help define more efficient paths or pairings for training that velocity field.

5.7.6 Relationship with diffusion models

Flow Matching is closely related to diffusion models, because both describe a process that connects noise with data.

In diffusion, one usually learns to invert a stochastic process that adds noise. The same model can parameterize noise, score, x_0 , or velocity, and these parameterizations can be converted into one another for a given noise path. For example, for the linear path used in Flow Matching, $x_t = (1 - t)x_0 + t\varepsilon$ with $\varepsilon \sim \mathcal{N}(0, I)$:

$$\varepsilon = \frac{x_t - (1 - t)x_0}{t}, \quad x_0 = \frac{x_t - t\varepsilon}{1 - t}, \quad s(x_t, t) = -\frac{\varepsilon}{t}, \quad v(x_t, t) = \varepsilon - x_0.$$

Given x_t and any one of the four quantities (noise ε , score s , x_0 , or velocity v), the others can be obtained without retraining the model. Other (nonlinear) noise paths give rise to analogous relationships but with different coefficients, as we will see in the next chapter.

In Flow Matching, the velocity field that transports samples from noise to data is learned directly.

Therefore, both frameworks are closely related: they differ in the objective and in how the path is specified, as we will see in the next chapter.

CHAPTER 6

Diffusion models

Chapter code: github.com/NahuelCostaCortez/generative-networks

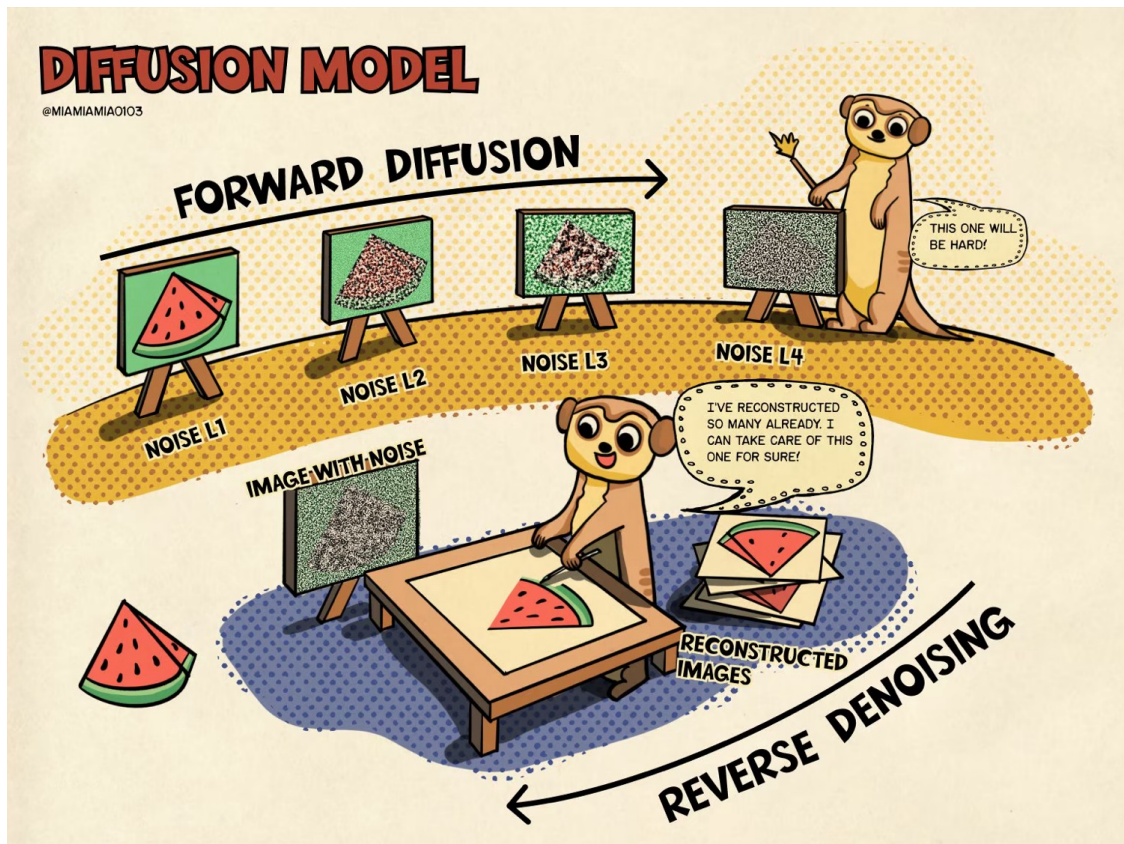


Figure 6.1. Source: [miamiamia0103](https://miamiamia0103.com)

6.1 Introduction

Diffusion models draw inspiration from [non-equilibrium thermodynamics](#). They add noise to the data and then learn a reverse process that generates samples from noise.

Specifically, a **diffusion process** is applied to gradually convert the observed data x_0 into a noisy version x_T by repeatedly passing the data through a stochastic encoder $q(x_t | x_{t-1})$. After enough steps, we end up with $x_T \approx \mathcal{N}(0, I)$, or any other convenient reference distribution. Then, a **reverse process** is learned by passing the noise through T steps of a decoder $p(x_{t-1} | x_t)$ until x_0 is generated again.

These models are similar to **hierarchical VAEs**, in the sense that they have an encoder and a decoder and follow a sequence of steps or hierarchies, and also to **normalizing flows**, since they apply a sequence of transformations to the data.

Non-Equilibrium Thermodynamics and Diffusion Models

Non-equilibrium thermodynamics studies how systems that are changing evolve irreversibly toward other states. Diffusion models use exactly that kind of dynamics, but applied to probability distributions (and with reversibility).

Remember the image from the beginning of the course? It should make much more sense now.

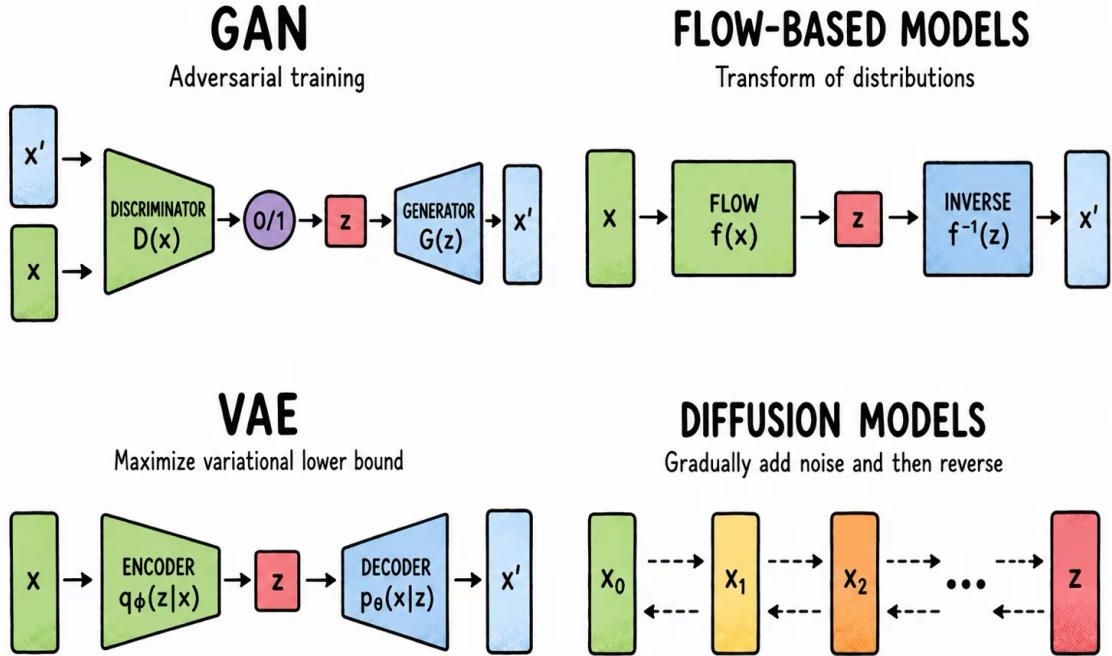


Figure 6.2. Overview of generative models

In diffusion models, the forward process in which we add noise is not learned but instead follows a fixed procedure, and the latent variable has the same dimensionality as the original data. On the other hand, the decoder is shared across all steps.

Several diffusion-based generative models have been proposed, including *diffusion probabilistic models*⁴², *noise-conditioned score network*⁴³, *denoising diffusion probabilistic models*⁴⁴, and *variational diffusion models*⁴⁵.

6.2 Encoder (forward diffusion)

Forward diffusion is the easy part because it doesn't involve training any model. It is a fixed procedure in which we decide how to add noise.

The process is usually defined as a simple linear Gaussian model:

$$q(\mathbf{x}_t | \mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t | \sqrt{1 - \beta_t} \mathbf{x}_{t-1}, \beta_t \mathbf{I}),$$

where the values of $\beta_t \in (0, 1)$ are chosen according to a **noise scheduler**.

Intuitively, β_t controls how much noise we add at each step t :

- if β_t is small, the image changes very little
- if β_t is large, we add a lot of noise at once

The **noise scheduler** decides how that amount of noise grows throughout the process. Adding a lot of noise from the start is not the same as adding it slowly and accelerating toward the end.

The joint distribution over the latent states is:

$$q(\mathbf{x}_{1:T} | \mathbf{x}_0) = \prod_{t=1}^T q(\mathbf{x}_t | \mathbf{x}_{t-1})$$

The distribution $q(\mathbf{x}_t | \mathbf{x}_0)$ is known as the **diffusion kernel**. Applying this to the input data distribution and then computing the resulting unconditional marginals is equivalent to Gaussian convolution:

$$q(\mathbf{x}_t) = \int q_0(\mathbf{x}_0) q(\mathbf{x}_t | \mathbf{x}_0) d\mathbf{x}_0$$

As t increases, the signal-to-noise ratio decreases and the marginals approach a simple distribution. In images, this process first removes high-frequency content (i.e., low-level details, such as texture), and then low-frequency content (high-level or "semantic" information, such as shape).

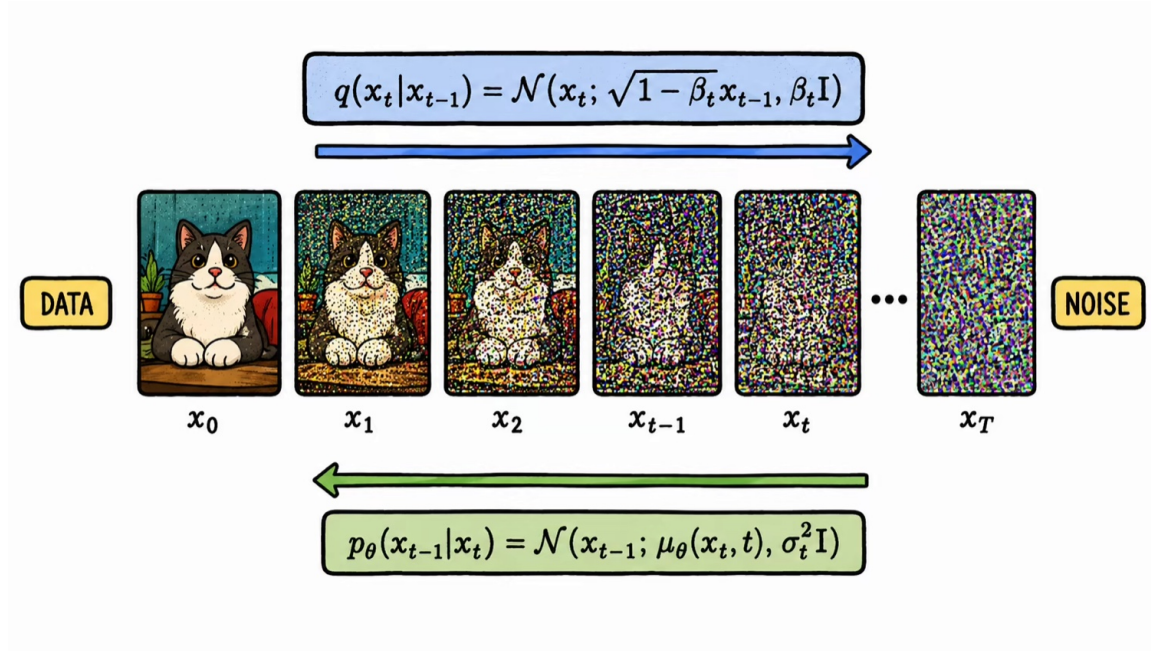


Figure 6.3. The forward process gradually adds noise and the reverse process learns to remove it

6.3 Decoder (reverse diffusion)

The ultimate goal of a diffusion model is for the reverse process (decoder) to learn the original data distribution, $p(x_0)$.

By adding noise in T steps, a sequence of intermediate distributions $q(x_t | x_0)$ is created, where x_t is the original data x_0 with an increasing amount of noise.

Generating a full image from scratch is hard, so the problem is reformulated to consist 'only' of removing a bit of noise from a given image. By doing it in several steps, we turn a complicated task into many simple ones.

In other words, to generate a new sample we don't know x_0 , so we train a model to approximate the reverse distribution:

$$p_\theta(\mathbf{x}_{t-1} | \mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1} | \mu_\theta(\mathbf{x}_t, t), \Sigma_\theta(\mathbf{x}_t, t))$$

The corresponding joint distribution over all generated variables is given by:

$$p_\theta(x_{0:T}) = p(x_T) \prod_{t=1}^T p_\theta(x_{t-1} | x_t), \quad \text{where } p(x_T) = \mathcal{N}(0, I)$$

To generate data from the model, a point $x_T \sim \mathcal{N}(0, I)$ is sampled and then the Markov chain is run backward, sampling $x_t \sim p_\theta(x_t | x_{t+1})$ until a sample is obtained in the original data space, x_0 .

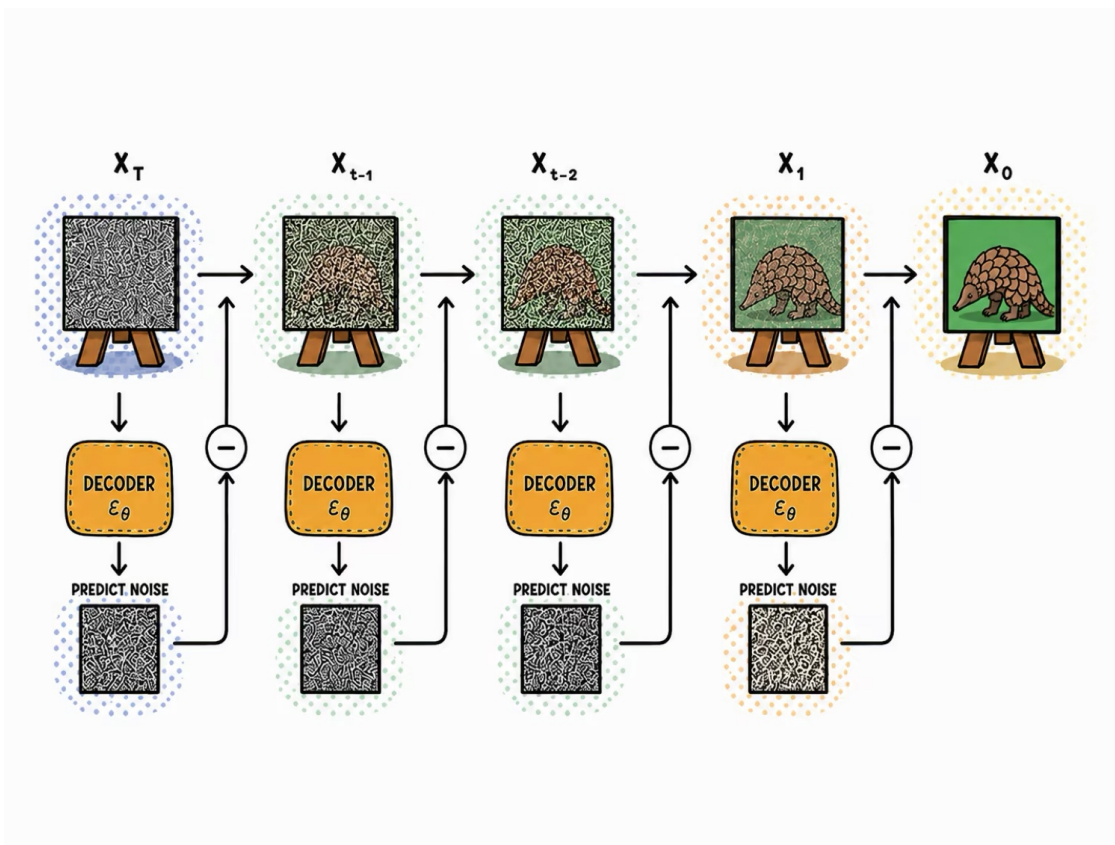


Figure 6.4. Stochastic reverse chain that transforms Gaussian noise into a data sample

6.4 Training

We have already defined the two processes that make up a diffusion model: the forward process adds noise through known transitions, and the reverse process progressively tries to recover the data. What's left is to learn the parameters θ of each reverse transition $p_\theta(x_{t-1} | x_t)$.

During training we start from a clean sample x_0 , so we can generate any noisy state x_t and compute what the step toward x_{t-1} should look like. The goal is to get the learned transition to approximate that reverse process. This idea is formulated probabilistically through the ELBO and ends up producing a simple noise-prediction loss.

6.4.1 The ELBO as a starting point

We want to maximize the log-likelihood of the data, $\log p_\theta(x_0)$, but computing it requires integrating over all latent states $x_{1:T}$. As in a VAE, we use the forward process $q(x_{1:T} | x_0)$ as the variational distribution and minimize the negative ELBO:

$$\mathcal{L}_{\text{VLB}} = \mathbb{E}_q \left[-\log \frac{p_\theta(x_{0:T})}{q(x_{1:T} | x_0)} \right]$$

*VLB stands for "Variational Lower Bound".

This expression can be decomposed into a sum of local problems:

$$\mathcal{L}_{\text{VLB}} = \underbrace{D_{\text{KL}}(q(x_T | x_0) \| p(x_T))}_{\mathcal{L}_T} + \sum_{t=2}^T \underbrace{D_{\text{KL}}(q(x_{t-1} | x_t, x_0) \| p_\theta(x_{t-1} | x_t))}_{\mathcal{L}_{t-1}} + \underbrace{-\log p_\theta(x_0 | x_1)}_{\mathcal{L}_0}$$

- $\mathcal{L}_T$ compares the last state of the forward process with the Gaussian prior and does not depend on θ if the noise scheduler is fixed.
- Each $\mathcal{L}_{t-1}$ compares the true reverse step, which we can compute during training because we know x_0 , with the learned reverse step.
- $\mathcal{L}_0$ corresponds to the final reconstruction step toward the data.

The distribution $q(x_{t-1} | x_t, x_0)$ is Gaussian and its mean can be expressed using the noise ϵ with which we constructed x_t :

$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

If we fix the variance of the reverse process and parameterize its mean using a network $\epsilon_\theta(x_t, t)$, each KL term that depends on θ becomes, up to constants, a squared-error noise-prediction term:

$$\mathcal{L}_{\text{VLB}} = \mathbb{E}_{x_0, \epsilon, t} \left[w_t \|\epsilon - \epsilon_\theta(x_t, t)\|^2 \right] + C,$$

where w_t is a weight determined by the noise scheduler and by the variance chosen for $p_\theta(x_{t-1} | x_t)$.

6.4.2 Simplified loss

Ho et al.⁴⁴ observed that dropping the weights w_t and sampling t uniformly produced better samples. This variant is known as $\mathcal{L}_{\text{simple}}$:

$$\mathcal{L}_{\text{simple}}(\theta) = \mathbb{E}_{x_0, \epsilon, t} \left[\|\epsilon - \epsilon_\theta(x_t, t)\|^2 \right]$$

This is the **loss function we will assume for the rest of the chapter**. It is a reweighted version of the variational objective that keeps the same denoising problem, but it is no longer exactly the original ELBO because it assigns the same explicit weight to every timestep.

In summary, during training:

1. We take an image x_0 .

2. We pick a random step t .
3. We add a known amount of noise ϵ to obtain x_t .
4. We feed x_t to the model.
5. The network predicts the noise it believes is in the image, $\epsilon_\theta(x_t, t)$, and we minimize $\mathcal{L}_{\text{simple}}$.

6.5 Example: Stable Diffusion

The original Stable Diffusion family⁴⁶ popularized latent diffusion. Instead of running the process directly on pixels, it runs it on a compressed **latent space**, which substantially reduces the computational cost.

This makes it a **latent diffusion model**.

The motivation is very practical. A 512×512 image with 3 channels has a very large dimensionality, so performing diffusion over all those pixels is expensive. Stable Diffusion first compresses the image to a smaller representation, performs the denoising process in that compressed space, and, at the end, decodes the result back to pixels.

6.5.1 Overview

We can think of Stable Diffusion as three main blocks:

1. **Text encoder**: converts the text prompt into a numerical representation.
2. **Image generator**: generates a latent representation of the image guided by the text.
3. **Image decoder**: reconstructs the latent representation into the target image dimension.

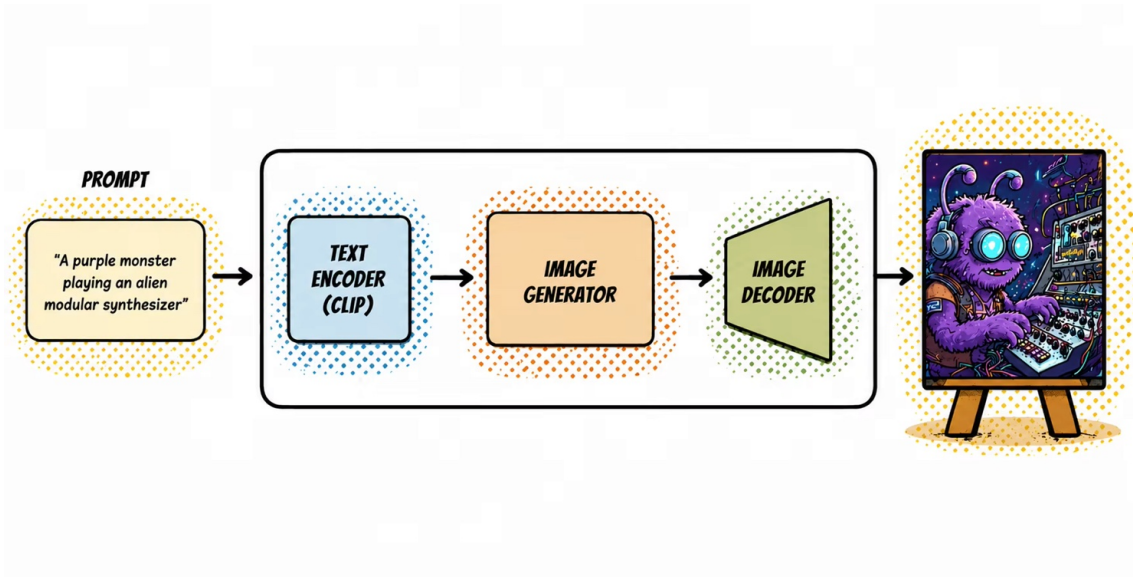


Figure 6.5. Stable Diffusion combines a text encoder, a latent denoiser, and an image decoder

6.5.2 Text encoder: CLIP

First, the model needs to convert prompts into vectors that a neural network can use.

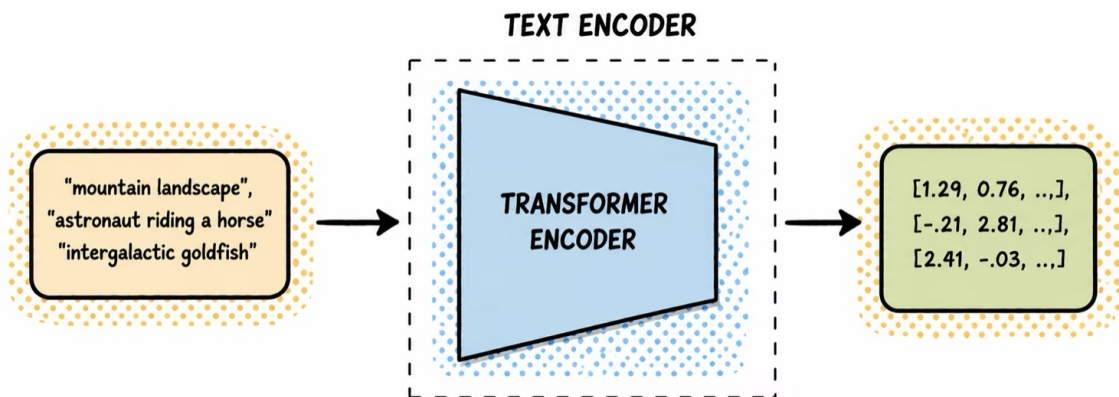


Figure 6.6. CLIP projects related images and text into a shared representation space

In the original Stable Diffusion, this block is CLIP's text encoder⁴⁷. CLIP jointly trains an image encoder and a text encoder to learn representations where an image and its description end up close together in vector space.

During training, image-text pairs are passed through their respective encoders and the embeddings are compared using cosine similarity. At the start, the similarity will be low, even if the text correctly describes the image, simply because the vector spaces of each encoder are different. What is then done is to update the weights of both networks so that, next time, the resulting embeddings are similar.

By repeating this operation, by the end of training the encoders end up producing embeddings in which, for example, an image of a pangolin and the phrase "a photo of a pangolin" end up in similar regions of the latent space.

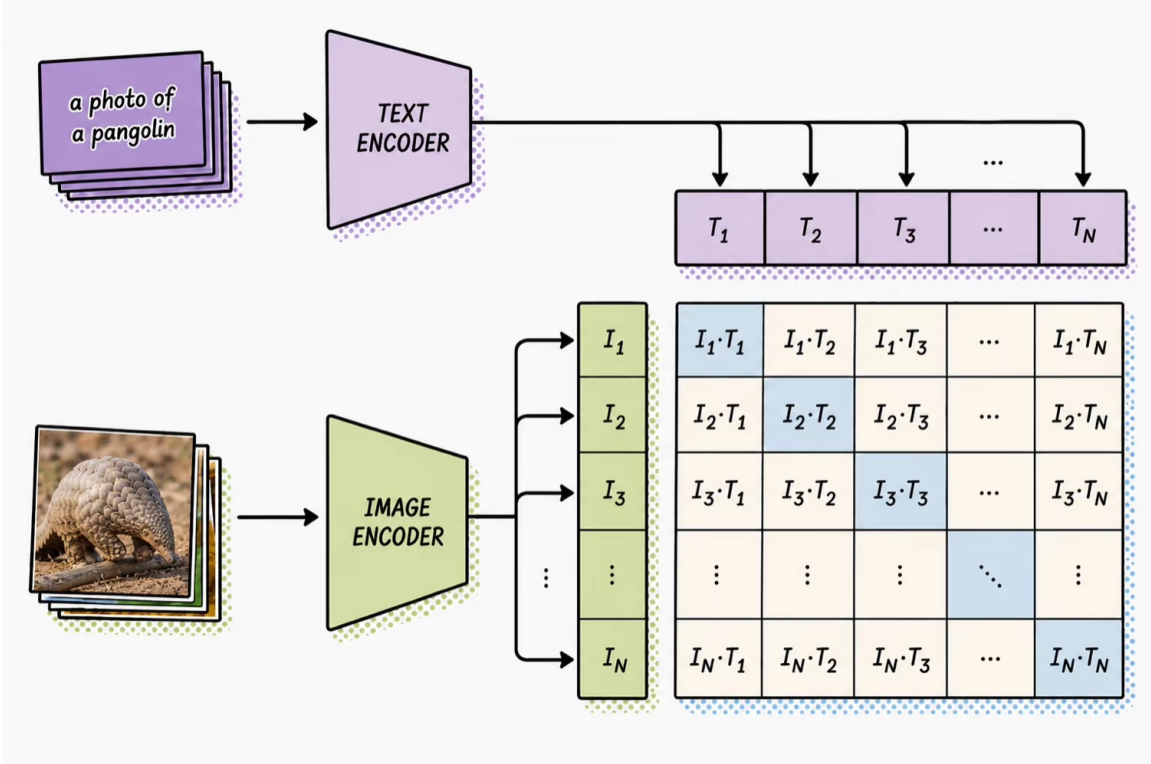


Figure 6.7. The text is tokenized and turned into contextual embeddings to condition the diffusion process

Note

Just like in word2vec, the training process must include negative examples. That is, images and descriptions that don't match, to which the model must assign low similarity scores (contrastive learning).

The **text encoder** is used to convert the prompt into a sequence of embeddings that contain contextual representations of the tokens. The generator will use it to guide the appearance of objects, styles, and relationships.

Example: if the prompt says "a dog with sunglasses on the beach", the text encoder generates a representation that will help the image generator know that concepts like dog, sunglasses, beach, outdoor light, sand, etc., should appear.

6.5.3 Image generator

The name "image generator" can be misleading because, in the Stable Diffusion family, this component does not produce pixels directly, but rather a matrix in **latent** space.

The core block is a **U-Net**. This network receives:

- a noisy latent z_t
- the timestep t
- the textual conditioning c produced by CLIP

And predicts the noise that should be subtracted:

$$\epsilon_{\theta}(z_t, t, c)$$

Then, a scheduler uses that prediction to compute a slightly less noisy latent:

$$z_t \rightarrow z_{t-1}$$

This process is repeated many times until a clean latent z_0 is obtained. This is not a visible image, so it is passed through the **decoder of a VAE** to convert it into pixels, $\hat{x} = D_{\text{VAE}}(z_0)$.

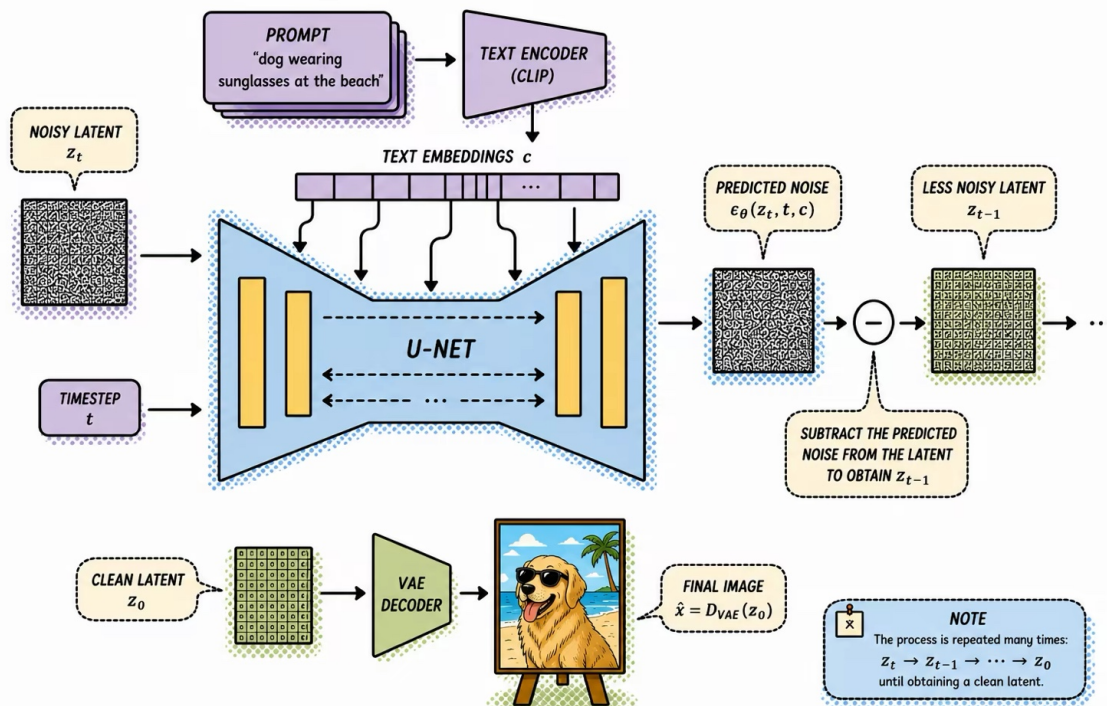


Figure 6.8. The U-Net predicts noise conditioned on the timestep and the text representation

The U-Net uses **cross-attention** mechanisms to mix the visual information of the latent with the textual information of the prompt. Thanks to this, the model can decide which parts of the image should attend to which parts of the text.

Cross-attention

Cross-attention lets the image "look" at the text while it is being generated.

For example, if the prompt contains "dog", "sunglasses", and "beach", different regions of the latent can use those words to form coherent objects and details.

6.5.4 Training

During training, the model mainly learns to remove noise in the latent space.

The typical process is:

1. A real image x and its associated text are taken.
2. The image is encoded with the VAE's encoder to obtain a latent z_0 .
3. The text is encoded with CLIP to obtain the conditioning c .
4. A random timestep t is chosen.
5. Noise is added to the latent z_0 to obtain z_t .
6. The U-Net receives (z_t, t, c) and predicts the added noise.
7. The real noise is compared with the predicted noise.

The loss is the same $\mathcal{L}_{\text{simple}}$ we just defined, but applied in the latent space and conditioned on the text:

$$\mathcal{L}_{\text{simple}}^{\text{latent}}(\theta) = \mathbb{E}_{z_0, \epsilon, t, c} [\|\epsilon - \epsilon_{\theta}(z_t, t, c)\|^2]$$

In many implementations, the VAE and the text encoder are already pretrained, so they are kept frozen. The trainable part consists of the U-Net's parameters for the denoising process.

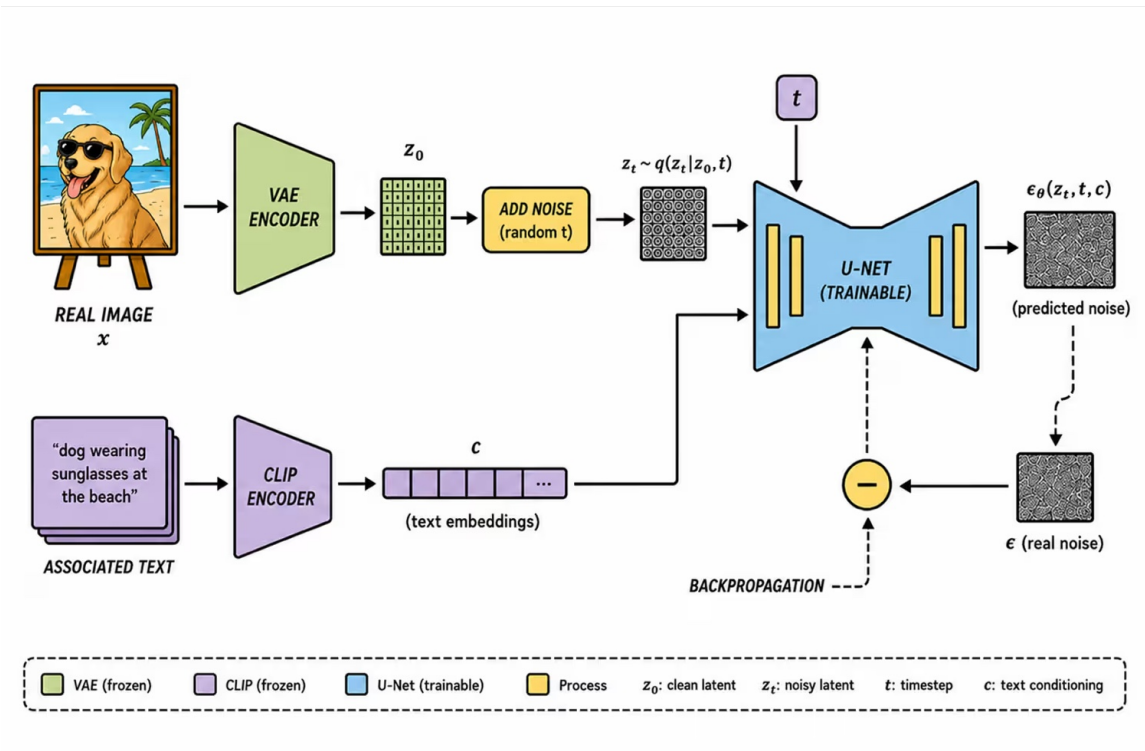


Figure 6.9. Training latent diffusion using known noise and the denoiser's prediction

6.5.5 Inference

Here we want to use the already-trained model to generate a new image.

The typical flow is:

1. We write a prompt.
2. CLIP converts the prompt into text embeddings c .
3. We sample a random initial latent $z_T \sim \mathcal{N}(0, I)$.
4. The U-Net predicts the noise in the current latent, conditioned on c .
5. The scheduler subtracts the predicted noise from the latent to make it slightly less noisy.
6. We repeat steps 4 and 5 for several timesteps.
7. The decoder converts the final latent z_0 into an image.

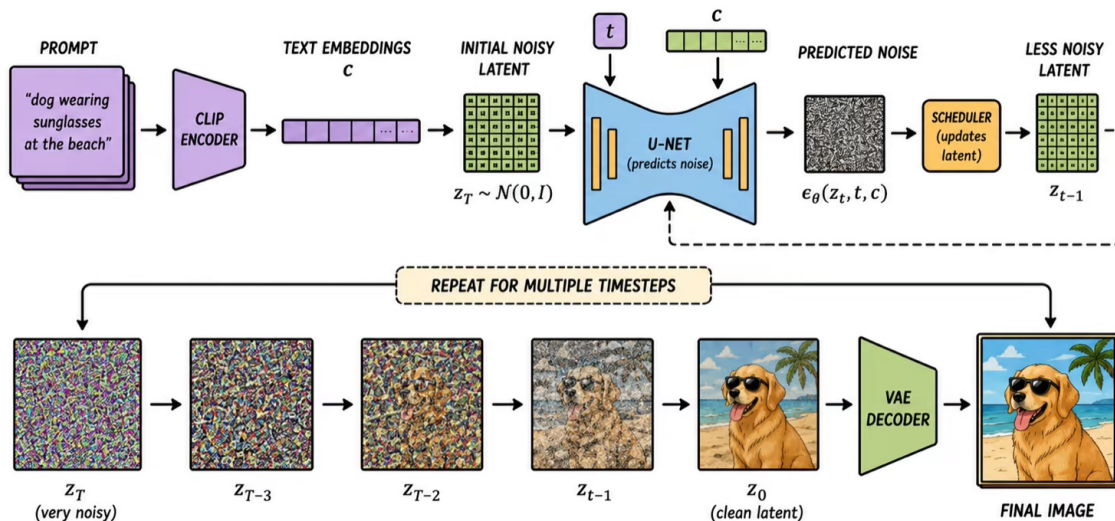


Figure 6.10. Iterative sampling in latent space followed by decoding into an image

6.6 Relationship to Flow Matching

Diffusion models and Flow Matching are closely related. Both start from a simple distribution, usually Gaussian noise, and learn to transform it into a target distribution. The most common way of presenting them differs in which parametrization the model uses to express that same objective,

although, as we'll see, all of them are convertible into one another.

In diffusion, the model learns to **predict the noise**:

$$\epsilon_{\theta}(x_t, t)$$

That is, it receives a noisy sample x_t and predicts which part of that sample corresponds to the added noise.

In Flow Matching, the model learns a **velocity**:

$$v_{\theta}(x_t, t)$$

That is, it receives an intermediate point x_t and predicts in which direction it should move to advance from the initial distribution toward the target distribution.

Intuition

Diffusion asks: "what noise do I need to remove?"

Flow Matching asks: "in which direction do I need to move?"

6.6.1 During training

In diffusion, we construct noisy examples by adding known noise to real data:

$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon$$

Since we know ϵ , we train the network to predict that noise.

In Flow Matching, we construct intermediate points between an initial sample z and a data point x :

$$x_t = (1 - t)z + tx$$

And we train the network to predict the velocity that would follow that path:

$$u_t = x - z$$

In both cases we build a supervised problem: we artificially generate an intermediate state and we know what the correct answer should be. In diffusion this target can also be expressed as a score, a clean sample x_0 , or a velocity v .

These parametrizations are not independent: given $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon$, they are related by

$$x_0 = \frac{x_t - \sqrt{1 - \bar{\alpha}_t} \epsilon}{\sqrt{\bar{\alpha}_t}}, \quad s(x_t, t) = -\frac{\epsilon}{\sqrt{1 - \bar{\alpha}_t}}, \quad v = \sqrt{\bar{\alpha}_t} \epsilon - \sqrt{1 - \bar{\alpha}_t} x_0.$$

Given x_t and any one of the four quantities (noise ϵ , score s , clean data x_0 , or velocity v), the others can be obtained without retraining the model. They are different parametrizations of the same training objective.

6.6.2 During generation

In diffusion, we start from noise and apply denoising steps:

$$x_T \rightarrow x_{T-1} \rightarrow \cdots \rightarrow x_0$$

In Flow Matching, we start from noise and follow a velocity field:

$$\frac{dx_t}{dt} = v_\theta(x_t, t)$$

Generation also proceeds from noise to data, but the mathematical starting point changes. Diffusion can be implemented as a stochastic chain or via its probability-flow ODE; Flow Matching directly defines and learns a velocity field over a probability path. The two frameworks are closely related.

CHAPTER 7

Transformers

Chapter code: github.com/NahuelCostaCortez/generative-networks

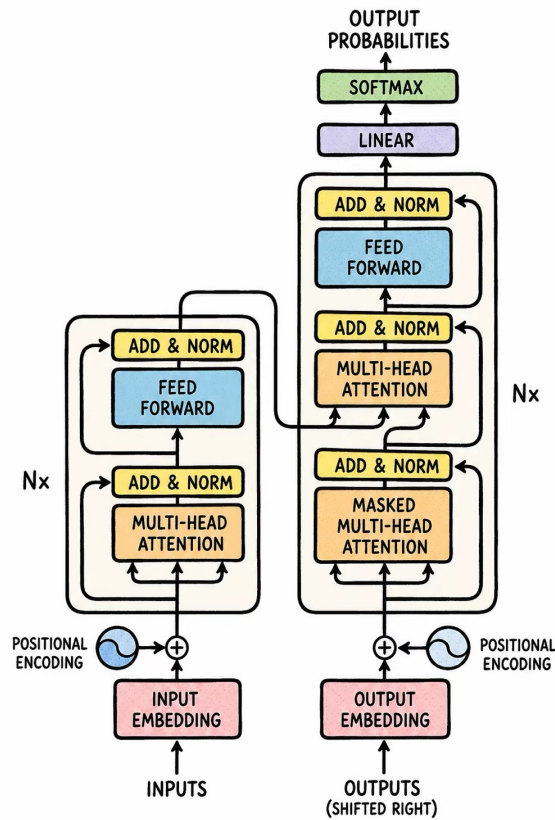


Figure 7.1. Architecture of a Transformer

7.1 Background

To understand the origin of Transformers we need to review some concepts we haven't covered yet.

7.1.1 From words to vectors

So far we have covered architectures mainly oriented toward images, which makes their processing easier because they already come in numerical format (images are matrices of pixels), but Transformers deal with text. Neural networks don't understand words, so the text has to be transformed into a numerical representation.

One-Hot Encoding

The most basic approach would be to associate each word with a number. For example, we could represent the sentence "The cat sat on the mat" as a vector "[25, 7, 4, 8, 27, 15]".

! Problem → If we fed those identifiers directly in as numeric values, we would be telling the model that words with a larger associated number carry more importance.

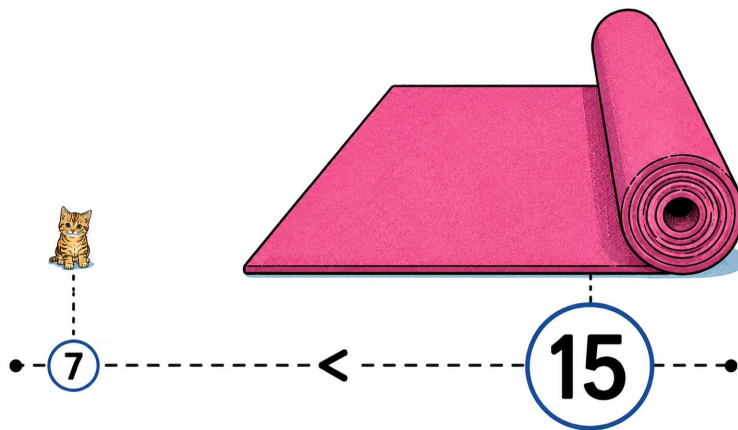


Figure 7.2. Two words represented by arbitrary numbers

💡 Instead, each word could be associated with a vector with as many positions as there are words in the vocabulary of the problem at hand. To represent a word, a 1 would be marked in the position that corresponds to it and 0 in the rest. For example, if the vocabulary only had 3 words we would have something like the following:

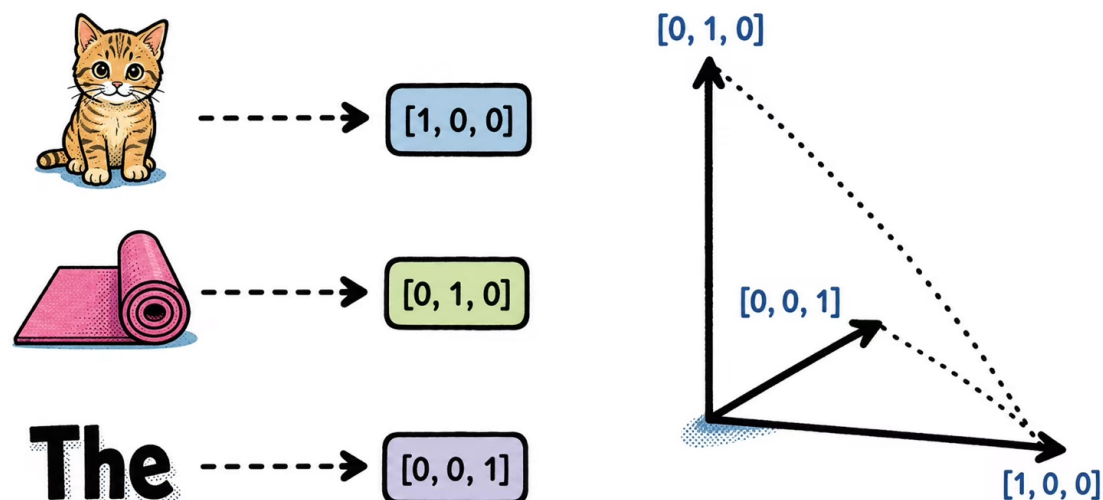


Figure 7.3. One-hot encoding of three words and their geometric representation

With this representation the distance between words is preserved! This happens regardless of the number of words (dimensionality) in the vocabulary.

! However:

- **Unnecessary storage:** a 10-word sentence with a 400-term vocabulary becomes a 10×400 matrix where most values are 0.
- **All words are equidistant:** this solves the previous problem, but words like "cat" and "box" should actually be closer together than "cat" and "sky".

Embeddings

Related to this last point, we as humans understand that some words are semantically more related to each other than others. "car", "mechanics" and "wheel" are conceptually closer to each other than "cloud" or "volcano".

For example, how would you mentally sort the images on the left?

One option could be by color and type of car → It would end up looking something like the image on the right.

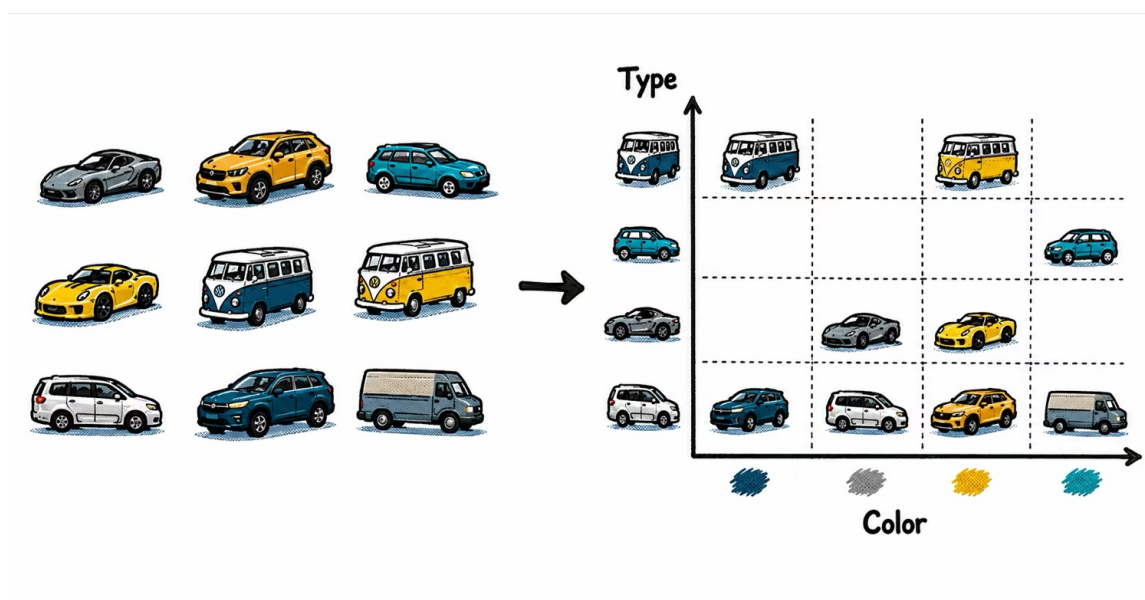


Figure 7.4. Example of sorting car images by color and type

Internally we follow a process of compressing and ordering data.

To begin with, each image has thousands of pixels, which our brain processes and orders until it obtains a much more compact representation. In other words, it has performed a dimensionality reduction, in this case down to two dimensions. This is precisely one of the operating principles of neural networks: compressing and ordering data until obtaining an internal representation (latent space or embedding) useful for downstream tasks.

This doesn't happen only with images; any data that can be converted into a numerical representation can be subject to this same process. We already saw that text can be converted into a numerical representation, whose dimensionality is very large, so it can be modeled to reduce/compress it.

At first, the embedding layer has no knowledge of how to organize the data, but, as it is trained, it learns representations suited to solving the task. For example, when analyzing whether a movie review is positive or negative, words like "happy", "satisfying" or "like" will tend to end up close together.

Note

Instead of always learning from scratch, we can start from pretrained embeddings. **Word2Vec**⁴⁸ popularized reusable static embeddings: each word keeps the same vector regardless of context. The [TensorFlow Embedding Projector](#) is a perfect resource for visualizing the similarities between embeddings (reduced to 3 dimensions) of different words.

7.1.2 Tokens

In practice, text is not split into words but into numeric units we call **tokens**.

Tokens are indices within a vocabulary: the number 1437 doesn't mean anything by itself, it's simply a position that points to an embedding. The embeddings for each token are initialized randomly and, as the model sees words during training, it adjusts the values of each vector to obtain the best representation of each one.

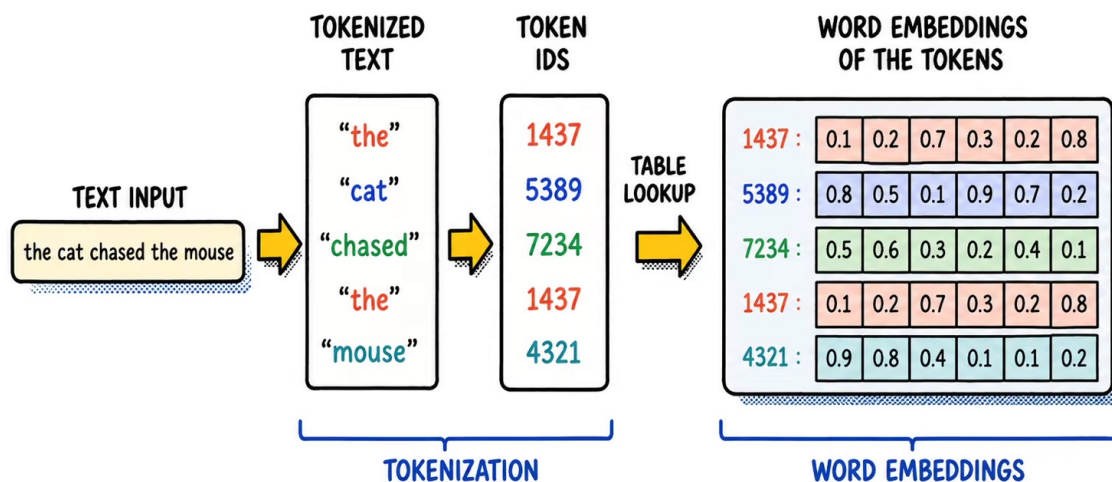


Figure 7.5. Tokenization pipeline and embedding lookup

Why split into tokens instead of words?

- Less unknown vocabulary (OOV): Allows processing new words, proper nouns or typos by splitting them into known fragments.
- Efficient vocabulary: Keeps the dictionary size within a manageable range (30k-256k tokens), avoiding excessive GPU memory consumption.
- Morphological flexibility: Makes it easier for the model to detect shared roots and suffixes in complex languages.
- Efficiency: frequent sequences tend to take up few tokens while uncommon words are split into smaller fragments.

There are different ways to tokenize text, so each model usually has its **own tokenizer** (typically

based on algorithms such as Byte Pair Encoding (BPE), SentencePiece (Unigram) or WordPiece), understood as a component that converts text into tokens.

7.1.3 Recurrent networks and Vanishing Gradient

Going back to the modeling side, many machine learning tasks were initially solved with simple neural architectures. Over time, architectures evolved to adapt to the nature of the data:

- **Images** → Convolutional Neural Networks (CNNs)
- **Data organized as graphs** → Graph Neural Networks (GNNs)
- **Sequential data**, such as text or time series → Recurrent Neural Networks (RNNs)

In sequential data, order matters. When processing a word, for example, it's necessary to take the previous words into account because they provide the context that allows it to be interpreted. Recurrent networks are precisely useful for this because they incorporate connections that reuse the information processed in previous steps (unlike a multilayer network, whose connections only move forward toward the output).

In an RNN, the first word is processed and the resulting state is fed into the processing of the next one. The process repeats until the whole sequence has been traversed. Thus, the state of the network at each step depends both on the current input and on the previous states, hence the name **recurrent network**.

The main problem with RNNs is that, when sequences are very long, the influence of the earliest elements on the last ones can become smaller and smaller. The network ends up "forgetting" information from the start of the sequence, a problem related to what is known as the *vanishing gradient*.

Vanishing gradient

During training, the error is propagated backward to compute how much each parameter should be adjusted. In long sequences, this computation chains together many derivatives. If their values are close to zero, their product becomes progressively smaller until the gradient practically vanishes. As a consequence, the earliest steps of the sequence receive barely any information to learn from.

LSTM and GRU architectures mitigate this problem through mechanisms that control which information is kept, updated or forgotten. Even so, each state depends on the previous one, which is a problem because it limits the handling of very distant dependencies and also makes parallel training impossible.

7.1.4 Attention mechanisms

Attention is a mechanism first introduced in *_Neural Machine Translation by Jointly Learning to Align and Translate_*⁵⁰ to represent the most relevant information of a sequence in a vector.

Think about how we translate into another language. We don't translate word by word; instead,

we focus on specific words of the original sentence to translate the current word. In other words, we fix our attention on specific parts of the sentence.

For example, in a sentence like "the player showed his cards":

- The relationship between "player" and "cards" takes us to the concept of a game
- "showed" as the verb of the sentence, is closely connected to the subject "player"

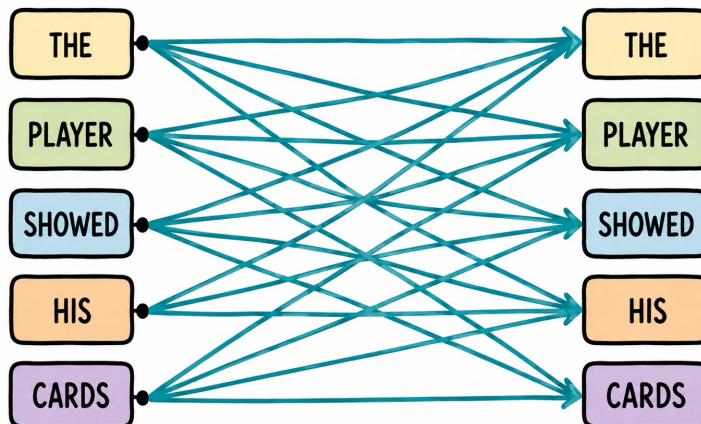


Figure 7.6. Attention connects the words player, showed and cards according to their contextual relationship

To search for these relationships, two projections are trained so that, given a sequence as input, they learn to generate two different vectors:

- **Identifier vector** 🔑 (**Key**): identifies the interesting properties that characterize each word.
- **Search vector** 🔓 (**Query**): describes the interesting properties that each word is looking for.

If there is some query word whose description is compatible with what the key word is looking for, a match should be generated. It's something equivalent to trying keys (identifier/key) in different locks (search/query), where the keys work with lesser or greater success.

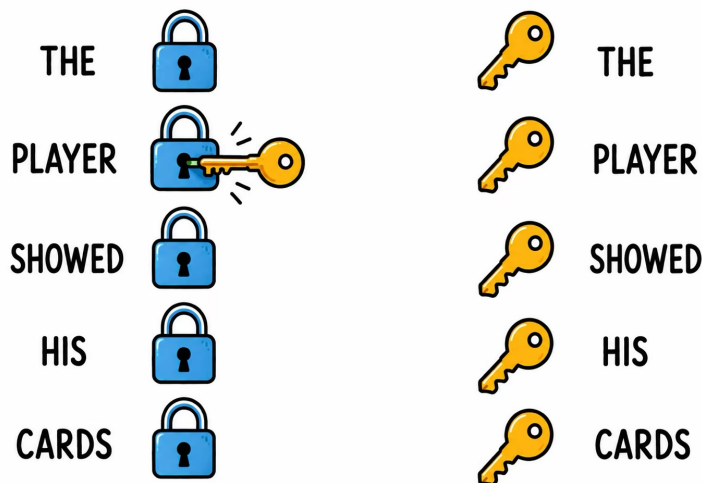


Figure 7.7. The Query and Key projections produce compatibility scores between tokens

To quantify compatibility, the dot product between the key and query vectors is used, QK^T . It will be larger the more the directions of the vectors coincide (if they coincide to a large degree, that value will be high).

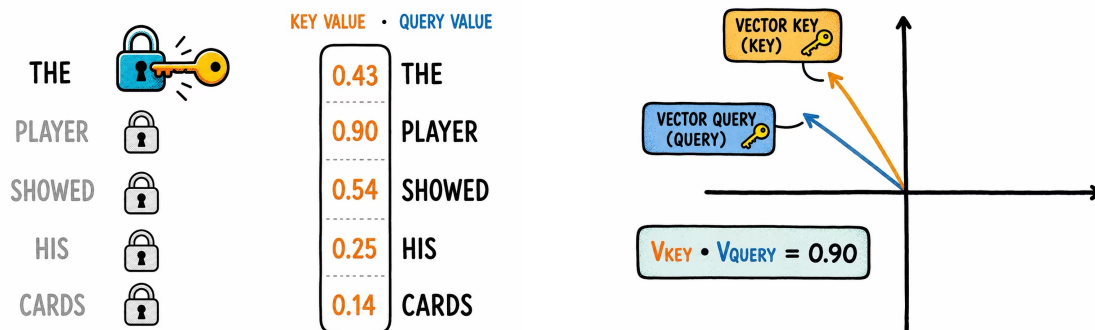


Figure 7.8. Key-query dot product (1) / Key-query dot product (2)

When working with high-dimensional vectors d_k , the sum of many products tends to produce extremely large scalar values. To smooth this out, it is divided by $\sqrt{d_k}$, normalizing the variance of the scores.

$$S = \frac{QK^T}{\sqrt{d_k}}$$

We know this as the **compatibility matrix or scores**. Then, the softmax function is applied along each row of S to transform the scores into a probability distribution (values between 0 and 1 that add up to 1).

$$A = \text{softmax}(S),$$

This is the **attention vector**. If a word's key fits the lock perfectly, it will get a weight close to 1 (almost total attention); if not, its weight will fall toward 0.

Note

There is one attention vector per word/token.

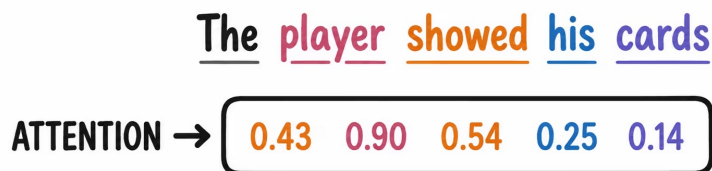


Figure 7.9. Attention vector for the token 'The'.

This way, unlike recurrent networks, words can be associated regardless of how far apart they are in the sentence.

Then, there is another projection that processes each of the tokens, generating a vector known as the **value vector (Value)**.

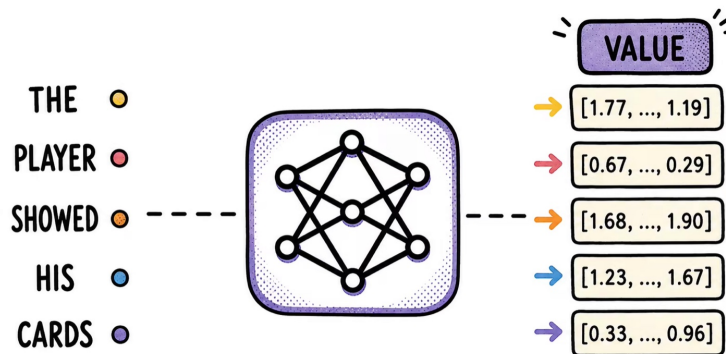


Figure 7.10. A vector is generated for each token.

If we compute the weighted sum of the product of the attention vector with the value vector, the result is a vector that captures the context of the sentence:

$$\text{Attention}(Q, K, V) = AV$$

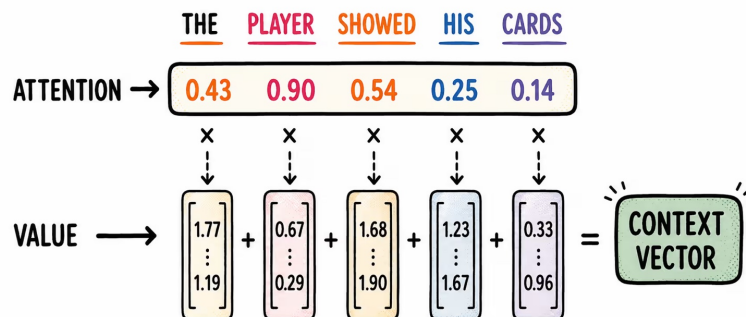


Figure 7.11. Weighted sum of five Value vectors to produce a contextualized vector

Note

In **multi-head attention**, several heads apply different projections in parallel. Each one can specialize in different relationships; their outputs are concatenated and projected again.

7.2 Transformers

We had seen that with recurrent networks we mainly had two problems arising from sequential processing:

- handling very distant dependencies (long texts) is limited
- parallel training becomes impossible

Attention mechanisms partly mitigate the first problem, but the second one remains.

Solution

Don't use recurrent networks → new architecture based on attention → **Transformers** : Attention is all you need⁴⁹.

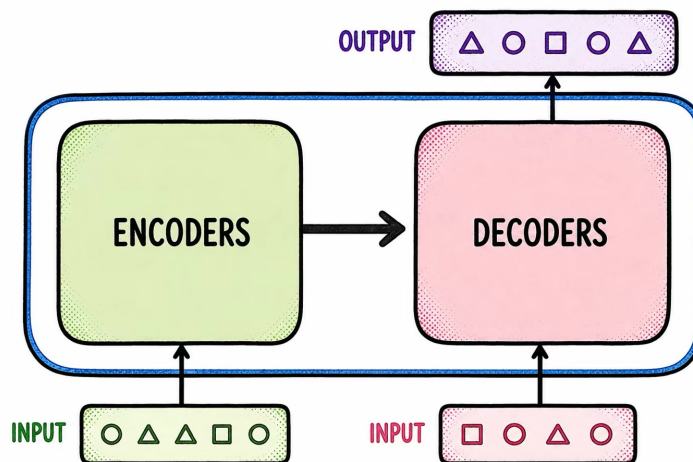


Figure 7.12. Simplified architecture of a transformer.

The inputs/outputs depend on the goal to be solved. Examples:

- **Translation.** Encoder input: sentence in language 1; decoder input: sentence in language 2; output: sentence in language 2 shifted by one unit.
- **Text prediction.** Encoder input: there is no encoder. Decoder input: sentence; output: next token/word.
- **Time series prediction.** Encoder input: time series (e.g.: T1, T2, T3, T4). Decoder input: continuation of the series (T4, T5); output: continuation of the series shifted by one unit (T5, T6).

7.2.1 Positional encoding

If everything is now processed at the same time, how do we know the order of the sequences?

Reminder

The sentences to be processed are not represented by words but by vectors that are learned in the embedding layer.

1st approach: one option would be to add to each embedding vector another vector whose components all mark the position it occupies (absolute positioning).

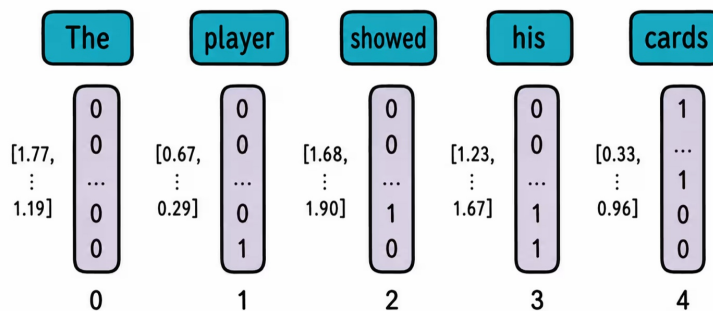


Figure 7.13. Absolute positioning

! Problem: if we have very long sequences, we would be adding a very large value to the last vector.

2nd approach: a solution to the previous problem would be to divide each positional vector by the number of words in the sentence, that is, to normalize the vectors.

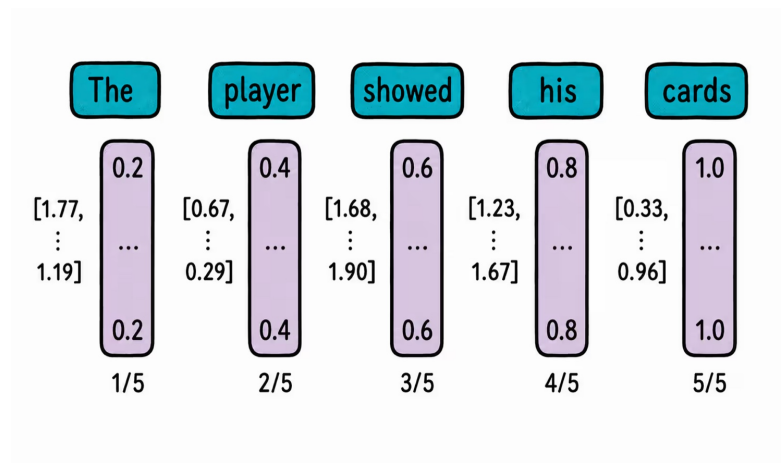


Figure 7.14. Normalization of the positional vectors

! Problem: what would happen if we have sequences of different lengths?
3rd approach: work only with 1s and 0s $\rightarrow$ binary encoding.

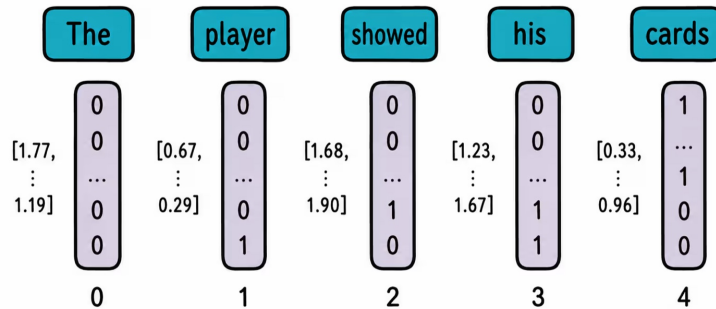


Figure 7.15. Binary encoding

This strategy is closer to the continuous nature of neural networks. In fact, these state changes can be represented as waves oscillating at different frequencies:

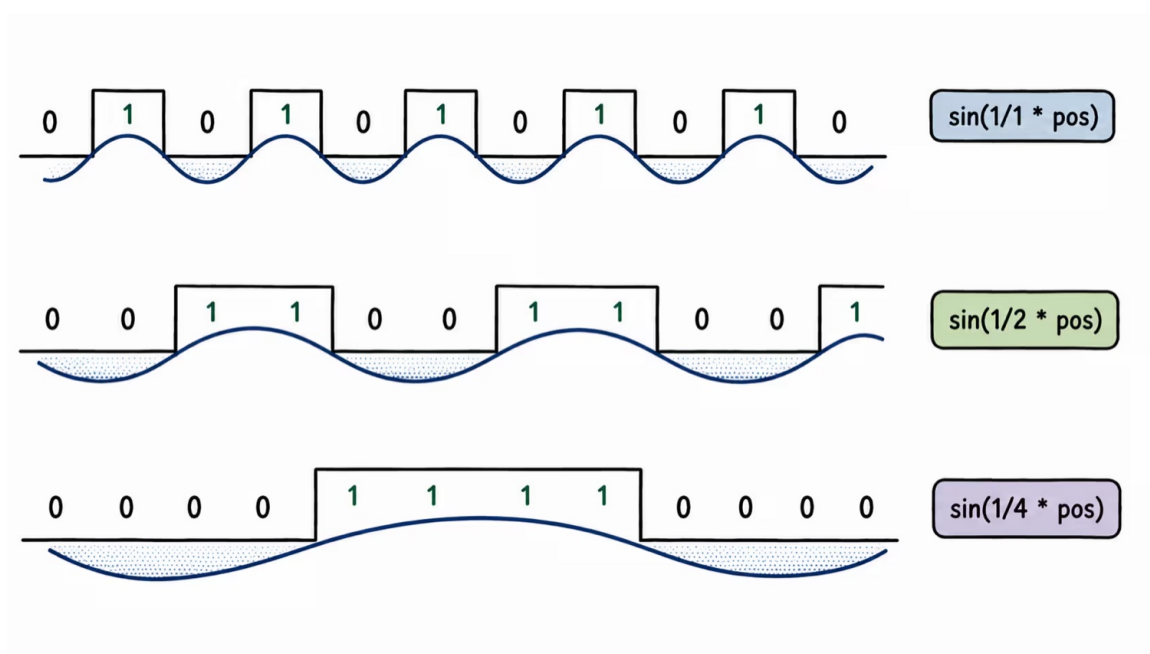


Figure 7.16. Positional encoding with waves

Therefore, positional encoding can be computed with a single sinusoidal function. The formula from the original paper is a bit more complex, but the intuition behind it is exactly the same:

$$PE_{(pos, 2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

$$PE_{(pos, 2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

This way, the Transformer can understand the order of the sequences passed as input (while still processing everything in parallel).

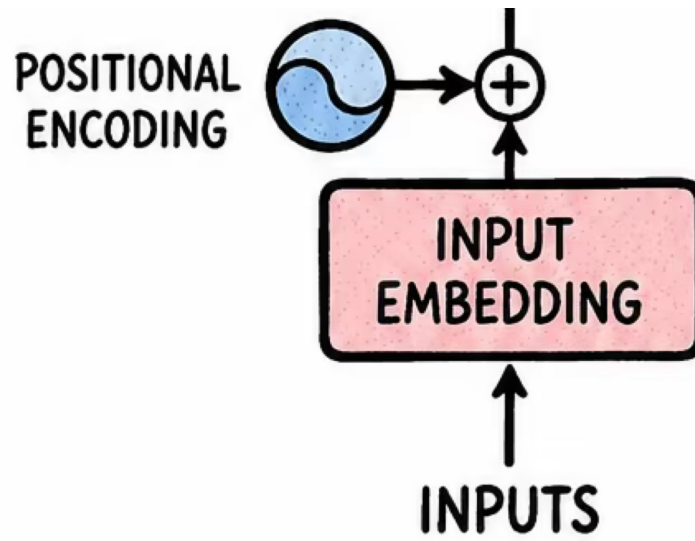


Figure 7.17. Sine and cosine waves of different frequencies encode the position of each token

7.2.2 Summary

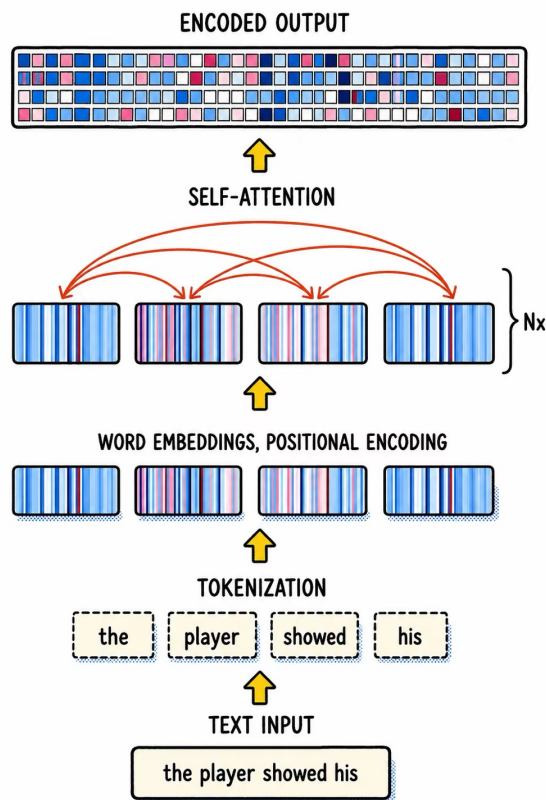


Figure 7.18. Processing up to the encoder output.

With this we already have a notion of the main components of the Transformer.

7.2.3 Learnable Positional Embeddings

In models such as BERT or GPT-2, the sinusoidal functions were replaced with learnable positional embeddings. That is, each position has an embedding that, instead of being deterministic, is trained, just like the word embeddings.

- **Advantage:** the model learns the best positional representation for its task.
- **Disadvantage:** it loses generalization to sequences longer than those seen during training.

7.2.4 RoPE (Rotary Positional Embedding)

Both sinusoidal encoding and learnable positional embeddings share the same strategy: they compute a position vector and **add** it to the word embedding before the sequence enters the Transformer. However, attention isn't interested in the absolute position of each word, but in the **relative distance** between the word asking (query) and the word being attended to (key). Adding a fixed position vector forces the model to infer that distance indirectly, by comparing two absolute positions.

RoPE⁵¹ is a strategy adopted by more recent models, in which, instead of adding the position, it **encodes it as a rotation** that is applied directly to the query and key vectors within each attention layer → never to the input embedding, and never to the value vector.

The geometric idea

Rotating a 2D vector by an angle θ around the origin doesn't change its length, only its orientation. RoPE takes advantage of exactly this property: it takes the query (or key) vector of a word at position m , splits it into pairs of components (x_{2i}, x_{2i+1}) , and rotates each pair by an angle proportional to m :

$$\begin{pmatrix} x'_{2i} \\ x'_{2i+1} \end{pmatrix} = \begin{pmatrix} \cos(m\theta_i) & -\sin(m\theta_i) \\ \sin(m\theta_i) & \cos(m\theta_i) \end{pmatrix} \begin{pmatrix} x_{2i} \\ x_{2i+1} \end{pmatrix}$$

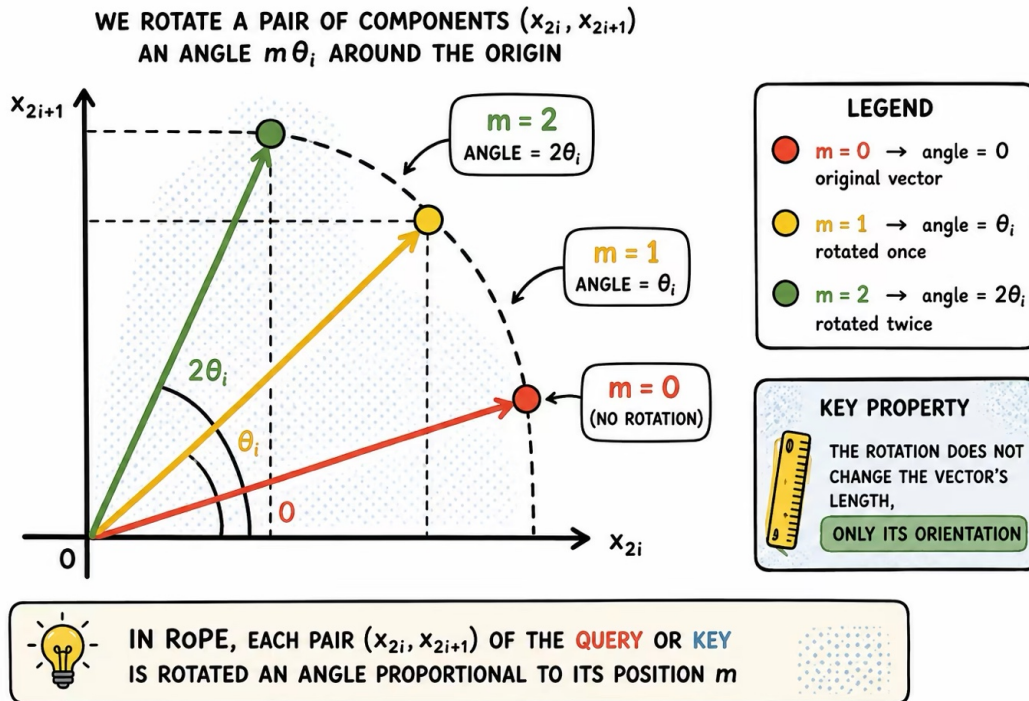


Figure 7.19. RoPE encodes position by rotating the components of Query and Key

Each pair of dimensions i rotates at a different frequency, $\theta_i = 10000^{-2i/d_{model}} \rightarrow$ the same frequencies used by sinusoidal encoding. The first pairs (high frequency) rotate quickly and distinguish nearby positions; the last ones (low frequency) rotate slowly and capture long-range relationships. A vector of d_{model} dimensions is thus treated as $d_{model}/2$ independent pairs, each rotating in its own 2D plane at its own speed. This process is applied equally to the query vector (with its position m) and to the key vector (with its position n) of each token.

Why this encodes relative position

This is RoPE's key result. When computing attention, what matters is the dot product between the rotated query at position m and the rotated key at position n . Calling R_m the rotation matrix at position m , and taking advantage of the fact that rotations are orthogonal ($R_m^\top = R_{-m}$):

$$(R_m q)^\top (R_n k) = q^\top R_m^\top R_n k = q^\top R_{n-m} k$$

The result **doesn't depend on m and n separately, but only on their difference $n - m$** : the relative distance between the two words. The absolute position disappears from the equation and only the relationship between positions survives, which is exactly what attention needs to know.

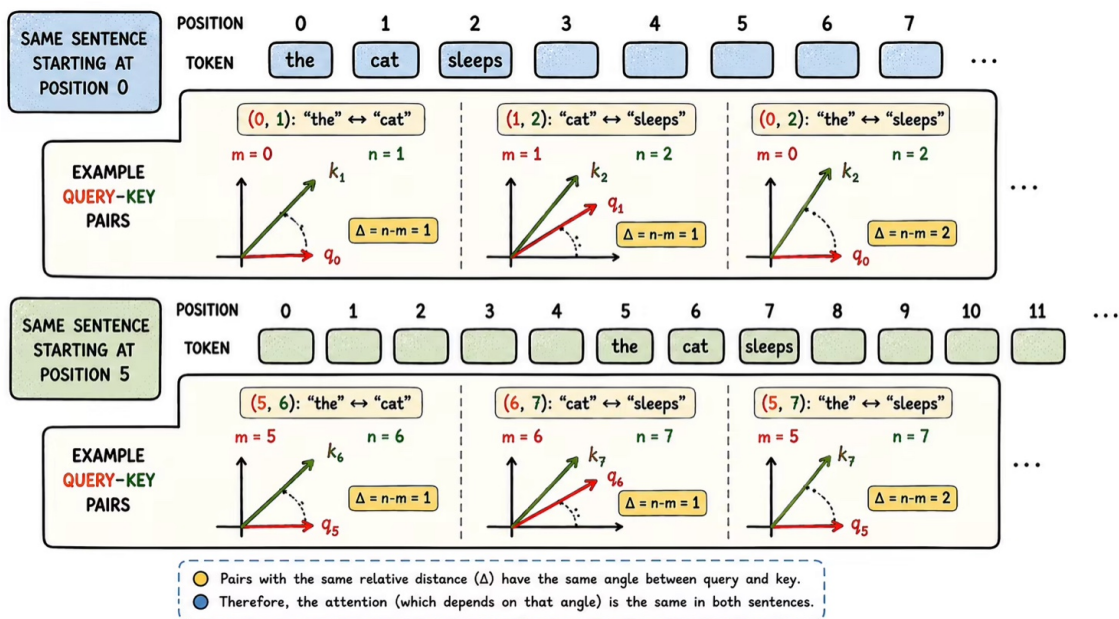


Figure 7.20. The angular difference between RoPE vectors represents a relative distance between positions

Advantages over the previous approaches

- **Encodes relative relationships natively**, without adding anything to the embedding or learning a position table.
- **Preserves the norm of the vectors**: a rotation doesn't change the length of the vector, so it doesn't distort the scale of the dot products the way adding a large position vector would.
- **Generalizes better to longer sequences** than those seen in training, because the mechanism isn't tied to a fixed range of learned positions (although extending the context far beyond training still requires additional adjustments, such as rescaling the frequencies — techniques like *NTK-aware scaling* or *YaRN*).
- **Adds no parameters**: unlike learnable positional embeddings, the rotations are computed with a fixed formula, not trained.

In summary, the flow that Q and K follow right before computing attention becomes: **embedding** → **linear projection to Q/K/V** → **rotate Q and K according to their position** → **dot product**.

7.2.5 Learning process

The original Transformer combines an encoder with an **autoregressive** decoder. In generation tasks, that decoder is trained to predict the next token of a sequence from the previous tokens.

Autoregressive model

An autoregressive model decomposes the probability of a full sequence as the product of conditional probabilities, predicting each element from the previous ones:

$$p(x_1, x_2, \dots, x_T) = \prod_{t=1}^T p(x_t | x_1, \dots, x_{t-1})$$

The decoder is given the target sequence itself as input, and that same sequence shifted by one position as output. This is what's known as **teacher forcing**: during training, at position t the model receives the real tokens $x_1, \dots, x_t$ as input (not the ones it would generate itself) and has to predict the token x_{t+1} .

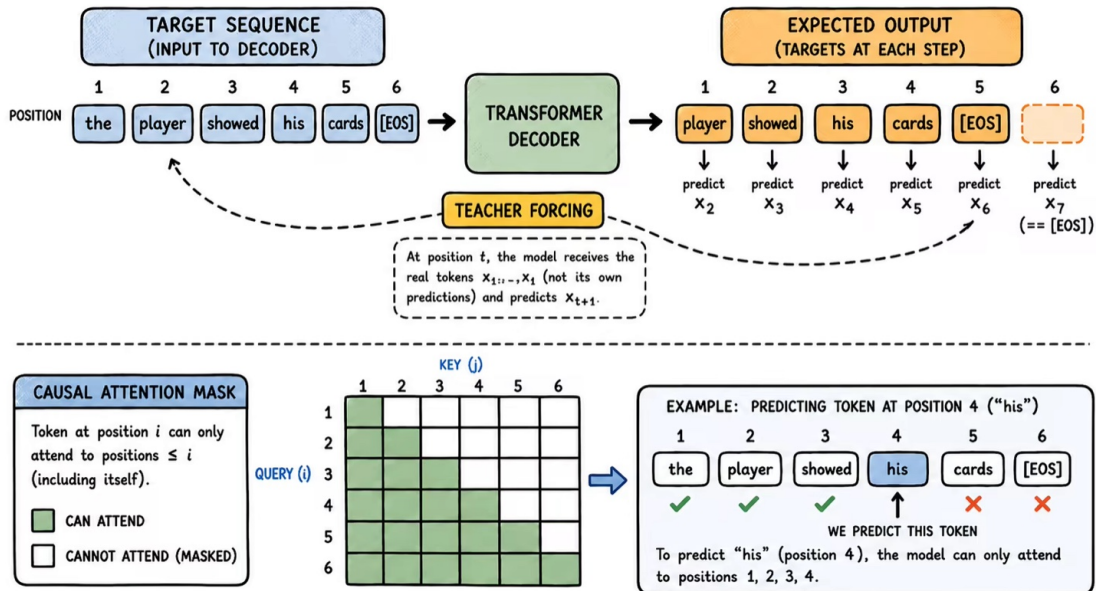


Figure 7.21. Teacher forcing shifts the target sequence to predict the next token

For this to work, two ingredients are needed:

- **Causal mask**: the decoder's self-attention is masked so that the prediction of token $t + 1$ cannot "cheat" by looking at future tokens → each position can only attend to itself and the previous ones.
- **Loss function**: at each position, the predicted probability distribution over the vocabulary (the

output of a final softmax layer) is compared with the real token using *cross-entropy*, and averaged across the whole sequence.

Thanks to the combination of teacher forcing and the causal mask, all positions of the sequence can be trained at once in a single forward pass → there is no need to wait for the model to predict token t in order to process $t + 1$.

Training $\neq$ inference

Teacher forcing only exists during training, because that's when we know the full target sequence in advance. During inference (generation) that "future" doesn't exist yet: the model generates a token, adds it to its own input sequence, and repeats the process autoregressively, one token at a time. That's why training is parallelizable, but generating remains sequential.

This same idea, training by predicting the next token from the previous context, is the one that, scaled up to massive amounts of unlabeled text, becomes the pretraining paradigm of foundational language models → we'll see it in detail in the [next chapter](#).

7.3 Extra: Vision Transformers

In ViTs ⁵², images are split into multiple sub-images (patches) that are passed to the Transformer encoder as if they were a sequence.

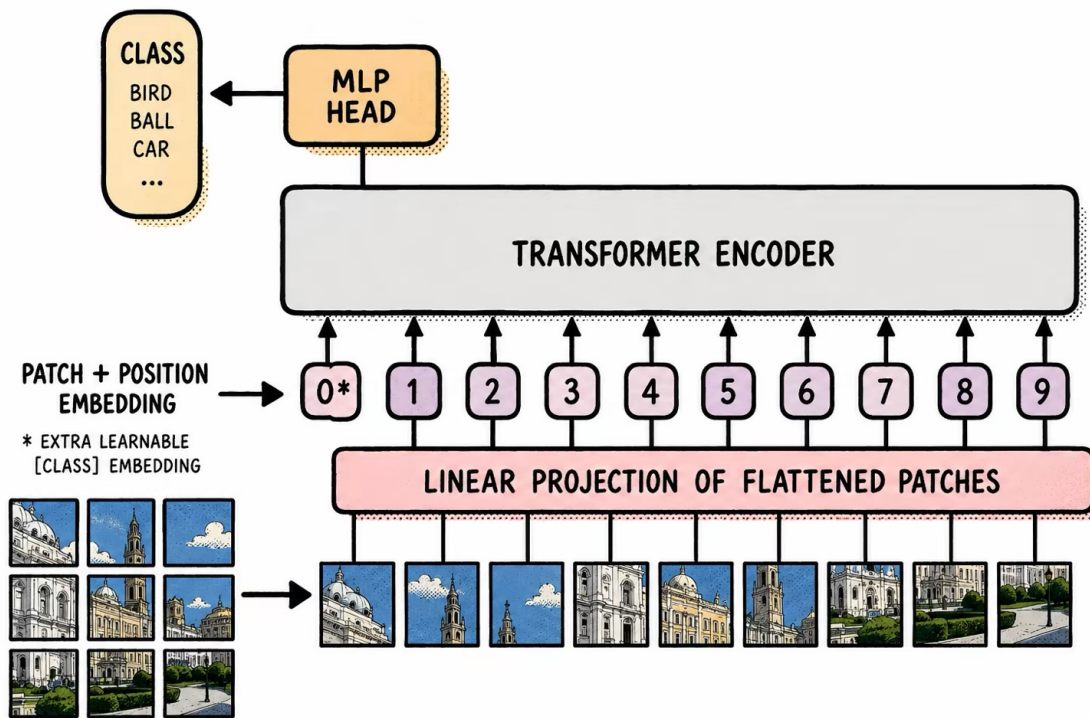


Figure 7.22. Vision Transformer splits an image into patches and processes them as a sequence

Again, since this is a Transformer that processes everything in parallel and therefore doesn't know the order of the data, it's necessary to provide it with that information.

Unlike text, where the order is sequential, in images the order between pixels matters, but it isn't strictly sequential between patches. For this reason, applying the original positional encoding doesn't make much sense. ViTs apply learnable Positional Embeddings to let the network learn the most suitable positional encoding.

That is, it's given the image and its position:

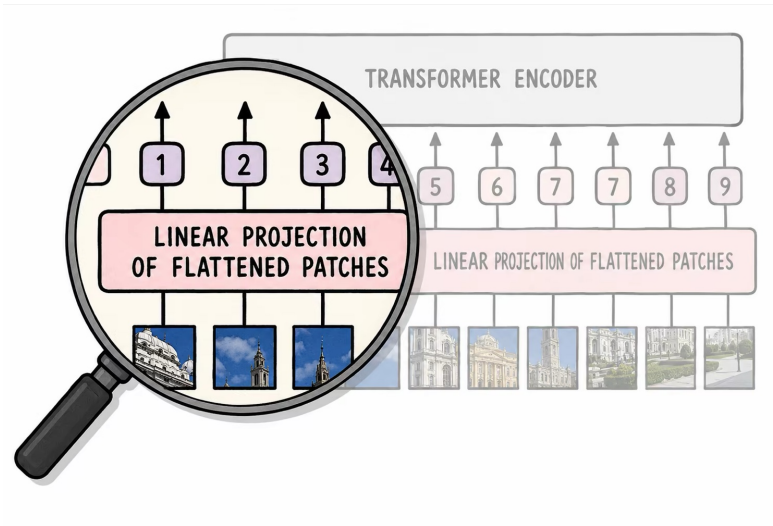


Figure 7.23. ViT adds learned positional embeddings to the sequence of patches

It isn't given the numbers 1, 2, 3... but rather each number is associated with a vector. These vectors are parameters that the model learns during training → learnable embeddings.

In this case, only the encoder can be trained with one of the learning paradigms we'll see in the next chapter. Once the model is pretrained, simple classifiers can be built on top of the representations obtained.

CHAPTER 8

Foundation Models

Chapter code: github.com/NahuelCostaCortez/generative-networks

In the previous chapter we saw how a Transformer can be trained autoregressively. Given a sequence of tokens, the model learns to predict the next one. This is a form of self-supervised learning. In this type of paradigm, if we train a sufficiently expressive model with a general task and a broad collection of data, the representations it learns can be reused to solve problems that were not explicitly defined during training.

This shift in perspective, from training a model for each task to **pretraining a base that can adapt to many tasks**, is the origin of foundation models.

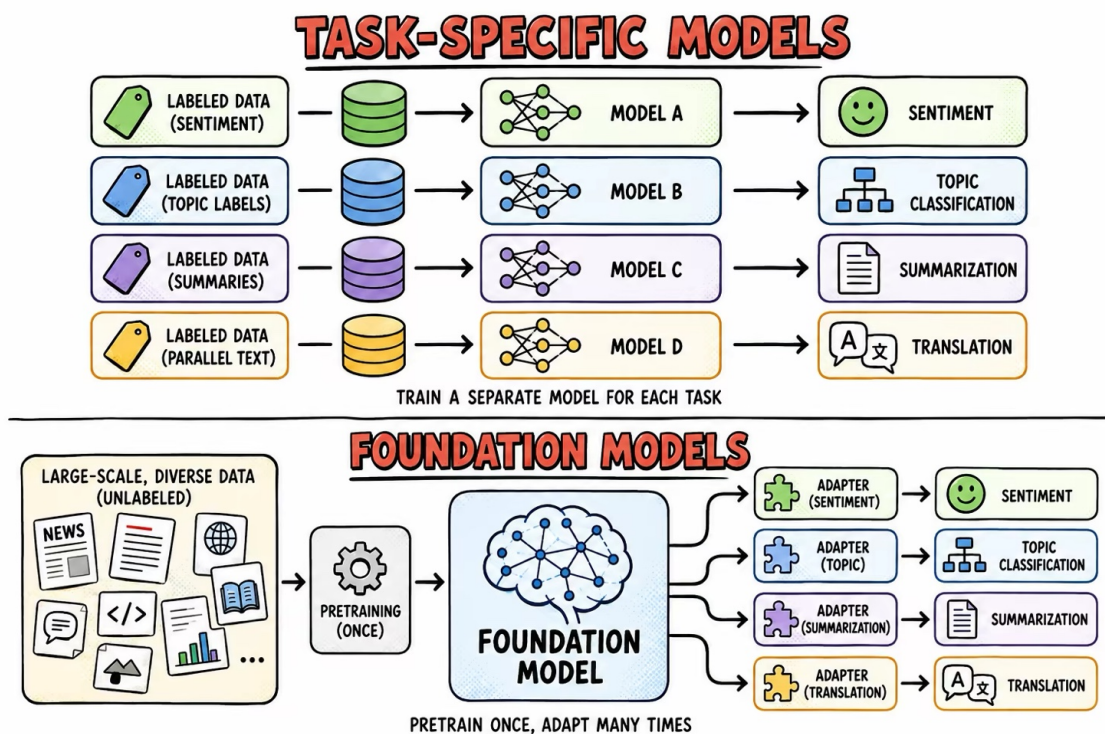


Figure 8.1. Pretraining, adaptation, and use cycle of a foundation model

8.1 Self-supervised learning

Imagine we want to classify the opinion of a review as positive or negative. Traditionally, what was done was to build a dataset of pairs (x, y) , where x is the text and y is its label, and from there train a model to approximate $p(y | x)$.

Now imagine that instead of classifying positive/negative we want to assign one to five stars $\rightarrow$ we would have to re-label all the data. Every problem would need new labels and, usually, a new training process.

This strategy works but:

- **Labels are expensive:** they require annotators and consistent criteria.
- **Knowledge stays fragmented:** each model only learns what it needs to solve its own task.
- **Reuse is limited:** a change in the categories or the domain can force us to label data and train again.

The problem is not supervised learning itself. Human labels are still essential. The problem is

using large amounts of task-specific supervision to learn from scratch everything that each task shares with the others.

Self-supervised learning has a history that predates modern training at scale, but the availability of large collections of text, images, audio, and video accelerated its adoption. This does not mean training happens without labels, but rather that the targets are obtained from the data itself instead of being manually annotated.

The transition did not happen overnight, nor did it eliminate supervised learning. It was an accumulation of ideas:

1. **Task-specific models:** one architecture and one labeled dataset for each task.
2. **Transfer learning:** reusing features learned on a large problem.
3. **Self-supervised pretraining:** learning most of the parameters with data that has no manual annotations.
4. **Pretraining at scale:** jointly increasing the diversity of the data, the capacity of the model, and the compute.
5. **General-purpose adaptation:** using fine-tuning, prompting, or in-context learning to address many tasks from the same base.

In natural language processing, models like BERT⁵⁵ showed that an encoder pretrained on unlabeled text could be adapted with few modifications to numerous tasks. On the other hand, the GPT family showed that scaling autoregressive decoders made it possible to reuse the same model through examples or instructions written directly in the context.

8.1.1 What is a foundation model

The term **foundation model** was popularized to describe models trained on broad, large-scale data that can adapt to a wide range of downstream tasks⁵⁶.

The word *foundational* does not imply that the model is finished or universal. It simply describes an incomplete base on which later systems are built.

We can think of four ingredients:

1. **Broad data:** covering many concepts, contexts, or modalities.
2. **General objective:** not tied to a single label or application.
3. **Scalable architecture:** can absorb more data and capacity without being redesigned for each task.
4. **Adaptability:** its representations or predictions can be reused in new problems.

Large, generative, and foundational are not synonyms

- A model can have many parameters and be specialized in a single task.
- A generative model may have been trained on a very narrow domain and not transfer well.

- A foundation model can be generative, like an LLM, or be designed mainly to produce representations, like a visual encoder.

LLMs are, therefore, one type of foundation model, but there are also foundation models for vision, audio, video, or multimodal systems.

Limitations of foundation models

- Large-scale pretraining requires a great deal of compute, energy, and financial resources.
- The provenance, quality, permissions, and representativeness of the data condition what the model learns.
- The model's biases, errors, and vulnerabilities can be inherited by the applications built on top of it.

8.1.2 Relevance of the Transformer

Self-supervised learning predates the Transformer and does not require this architecture. However, the Transformer brought together several properties especially suited to pretraining at scale:

- Parallelization during training.
- Long-range dependencies via self-attention.
- It can adapt to different modalities: text tokens, image patches, audio fragments, multimodal sequences...
- The same structure scales by increasing layers, dimensionality, data, and compute.

8.2 Encoders and decoders

The original Transformer⁴⁹ used both: the **encoder** to build a contextual representation of the input sequence, and the **decoder** to generate the output sequence. Modern variants can keep both components or use only one of them.

8.2.1 Encoder-only

In an encoder, each token can normally attend to all the other tokens in the input. The representation of a word can use both the preceding and the following words:

$$h_t = f_{\theta}(x_1, \dots, x_T)$$

This bidirectional visibility produces contextual representations that are very useful for:

- classification and regression
- information extraction
- semantic search
- segmentation or detection in images
- building embeddings for other networks

BERT⁵⁵ is a clear example of this.

8.2.2 Decoder-only

In a causal decoder, each position can only use itself and the previous positions:

$$h_t = f_{\theta}(x_1, \dots, x_t)$$

This restriction allows the output to be interpreted as a distribution over the next token and to generate a sequence step by step. It is the natural configuration for language models.

It is worth noting that a decoder is not limited to "writing text" — its internal states also contain useful representations. However, its training and usage setup is especially aligned with open-ended generation.

8.2.3 Encoder-decoder

The encoder processes a complete input and the decoder generates an output conditioned on it:

$$p(y \mid x) = \prod_{t=1}^{T_y} p(y_t \mid y_{<t}, \text{Enc}(x))$$

This separation is natural when the input and the output play different roles, as in translation, summarization, image captioning, or conversion between modalities.

8.2.4 Other compositions

Foundation models do not always fit into those three classic configurations. For example, CLIP⁵⁷ uses a **dual encoder** to process two modalities separately and project their representations into a shared space, in this case images and text. Another example is JEPA, where a **context encoder + target encoder + predictor** architecture can be used to predict a target representation.

8.3 Pre-training tasks

So far we have talked about *self-supervised learning* in general terms, but this only describes where the supervision comes from. However, there are different paradigms. To compare them it is useful to ask what information the model receives and what it must predict.

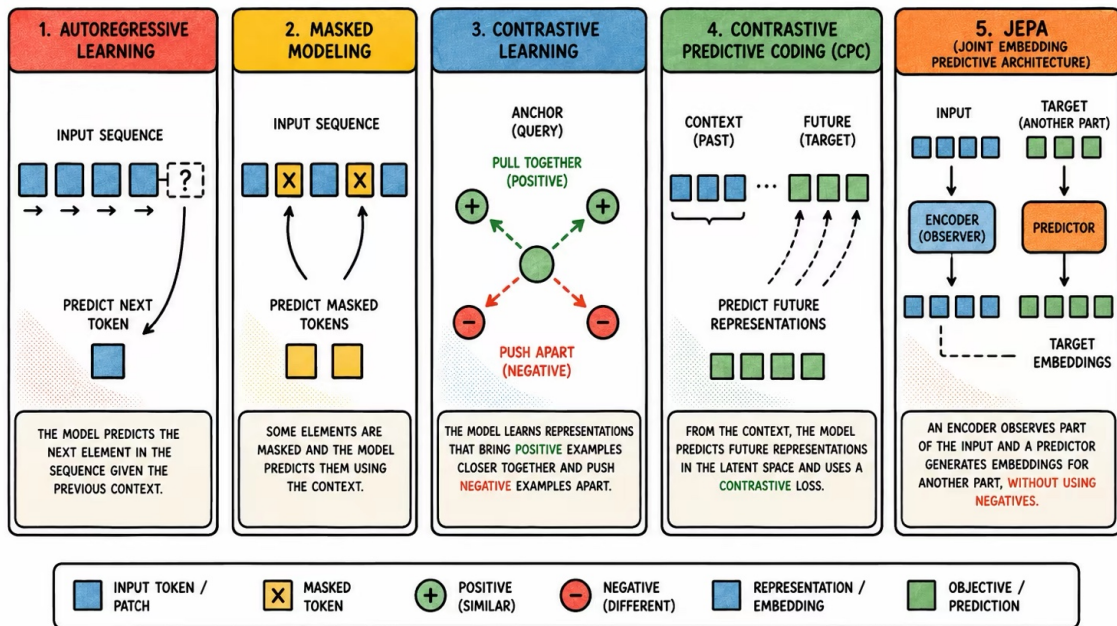


Figure 8.2. Comparison of autoregressive, masked, contrastive, and predictive objectives

8.3.1 Autoregressive learning

As we saw in the previous chapter, an autoregressive model factorizes the joint probability of a sequence:

$$p(x_1, \dots, x_T) = \prod_{t=1}^T p(x_t | x_{<t})$$

Its usual loss function is the cross-entropy between the actual next element and the predicted distribution:

$$\mathcal{L}_{AR} = - \sum_{t=1}^T \log p_{\theta}(x_t | x_{<t})$$

Each position acts as a new target, and the objective coincides with the generation procedure, which makes the causal decoder a direct generative model.

8.3.2 Masked modeling

Another possibility is to hide or corrupt part of the input and ask the model to reconstruct it. If M is the set of masked positions:

$$\mathcal{L}_{mask} = - \sum_{i \in M} \log p_{\theta}(x_i \mid x_{\setminus M})$$

In **masked language modeling (MLM)**, the encoder observes the context on both sides of the gap:

"The player [MASK] their cards" $\rightarrow$ "showed".

BERT uses this objective to pretrain an encoder that can later be adapted to supervised tasks.

The same idea can be carried over to other modalities. In images, **Masked Autoencoders (MAE)** are commonly used, which consist of removing a high proportion of patches, processing only the visible ones with an encoder, and using a decoder to reconstruct the missing pixels⁵⁸. Once pretraining is finished, the decoder is discarded and the encoder is kept as the foundation model.

Note

Masking only describes how an incomplete input is constructed, but it does not define what type of architecture to use. BERT uses an encoder; BART⁵³ and T5⁵⁴ follow an encoder-decoder scheme, and MAE uses an encoder with an auxiliary decoder.

8.3.3 Contrastive learning

Instead of reconstructing the data, the model can be taught which samples should have similar representations.

We usually start from a sample x , from which two related views $\tilde{x}_i$ and $\tilde{x}_j$ are generated. In an image these might be two crops with color changes; in a multimodal system they might be a photograph and its caption. They are then passed through an encoder that turns each view into an embedding:

$$z_i = g_{\theta}(\tilde{x}_i), \quad z_j = g_{\theta}(\tilde{x}_j)$$

A contrastive loss pulls the positive pair (z_i, z_j) together and pushes it apart from representations corresponding to other samples. A common form is InfoNCE⁵⁹:

$$\mathcal{L}_i = -\log \frac{\exp(\text{sim}(z_i, z_j)/\tau)}{\sum_{k \neq i} \exp(\text{sim}(z_i, z_k)/\tau)},$$

where τ is a temperature and sim is usually cosine similarity.

The encoder is taught which changes it should ignore and which relationships it should preserve. SimCLR⁶⁰, for example, treats two transformations of the same image as positive. CLIP aligns corresponding images and texts to build a space shared across modalities.

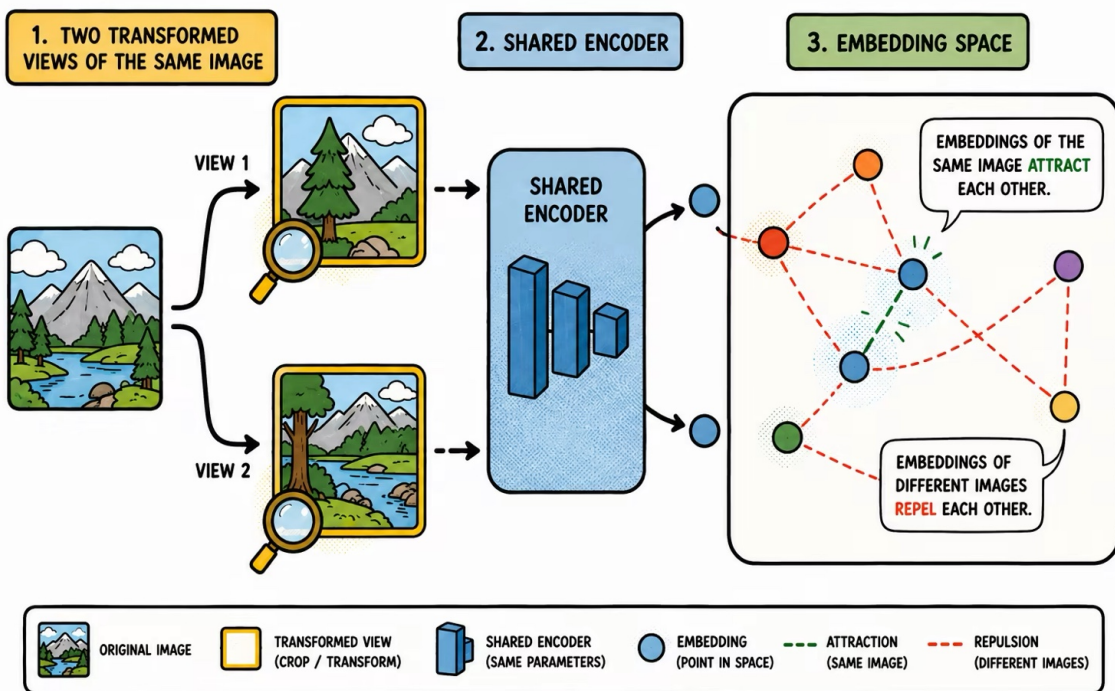


Figure 8.3. Contrastive learning of two views of the same observation

The choice of views is crucial. If two transformations are considered equivalent, the model will learn to be invariant to their differences. A poorly chosen transformation can eliminate precisely the semantic property that makes two images have something in common.

Contrastive does not always mean self-supervised

Contrastive describes how representations are compared, not where the supervision comes from. SimCLR is self-supervised because it automatically constructs the positive pair by applying two transformations to the same unlabeled image. CLIP, in contrast, learns from image-text pairs obtained from the web: it uses weak supervision, or supervision via natural language, since the correspondence between the two modalities provides an external signal that already exists in the data.

8.3.4 Contrastive Predictive Coding (CPC)

CPC⁵⁹ combines temporal prediction, latent representations, and contrastive learning.

First, an encoder transforms each observation into a representation z_t . Then, an autoregressive model summarizes the past into a context vector c_t :

$$z_t = g_{enc}(x_t), \quad c_t = g_{ar}(z_{\leq t})$$

From c_t , the model must identify the correct future representation z_{t+k} among several negative samples. It does not try to reconstruct anything, but instead retains the information that allows it to distinguish the real future from plausible alternatives.

CPC occupies an interesting position among several families:

- it is **autoregressive** because it summarizes the past to anticipate the future,
- it is **contrastive** because it distinguishes positive targets from negative ones,
- it is **predictive in latent space** because the target is a representation, not the full observation.

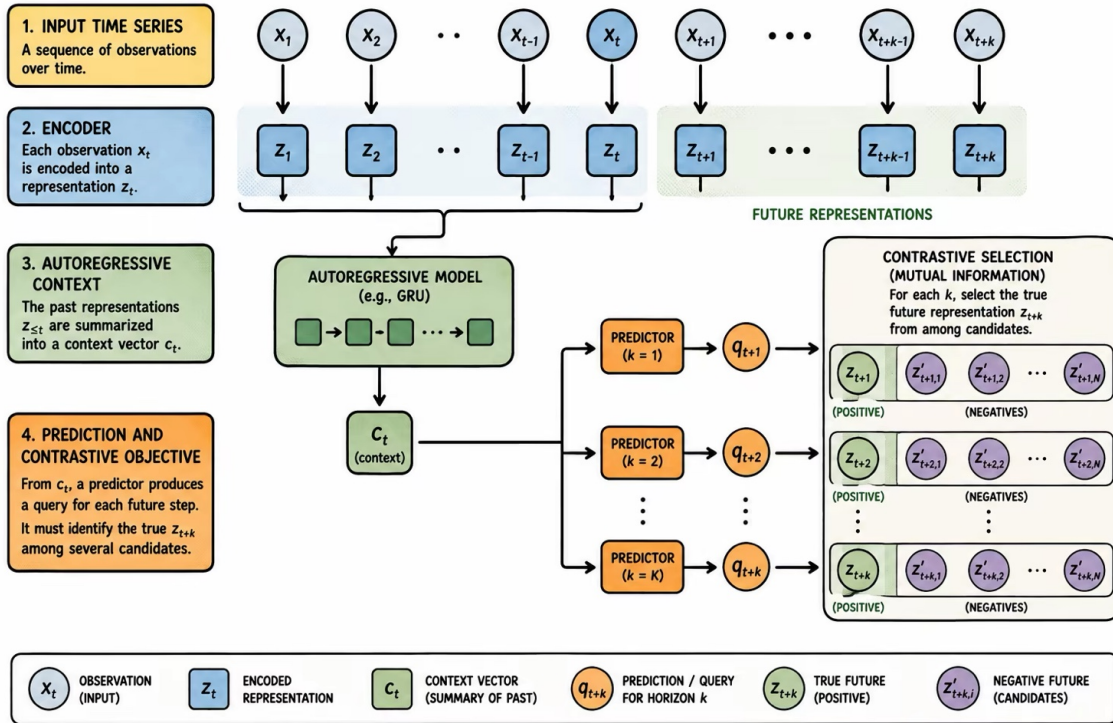


Figure 8.4. Contrastive Predictive Coding applied to a sequence of representations

8.3.5 From observations to representations

Reconstructing high-dimensional data can be unnecessarily difficult. To complete an image, a model might have to decide the exact texture of a wall or the position of every leaf on a tree. These are details that are hard to predict and perhaps irrelevant if what we later want is to recognize objects or understand a scene.

We can express the difference schematically:

$$\underbrace{\text{context} \rightarrow \text{missing datum}}_{\text{data space}}$$

$$\underbrace{\text{context} \rightarrow \text{target embedding}}_{\text{latent space}}$$

The second option allows the target to omit unpredictable details and retain more abstract properties. This is the main motivation behind **Joint-Embedding Predictive Architectures**.

8.3.6 JEPA

Joint-Embedding Predictive Architectures (JEPA)⁶¹ were put forward by Yann LeCun as part of a broader proposal for building machines capable of learning internal models of the world. We will first study JEPA as a pretraining method, and later connect these representations with agents, planning, and world models.

JEPA is not a specific network, but a family of architectures and objectives. We usually talk about three components:

1. A **Context Encoder** transforms the observed part x into a latent representation s_x .
2. A **Target Encoder** transforms another related view y into the target representation s_y .
3. A **Predictor** uses s_x and some information about the relationship between the two views to estimate s_y .

Schematically:

$$s_x = f_\theta(x), \quad s_y = f_{\bar{\theta}}(y),$$

$$\hat{s}_y = p_\phi(s_x, c), \quad \mathcal{L}_{JEPA} = D(\hat{s}_y, \text{sg}(s_y)),$$

where c describes which target is to be predicted, D measures the distance between representations, and sg indicates that the gradient does not update the target encoder.

During pretraining, both encoders observe related views of the same data. The Target Encoder and the Predictor are auxiliary training components and can be discarded afterward. What is worth keeping as the foundation model is the Context Encoder.

Why is the Context Encoder the resulting foundation model?

The Predictor only serves as a loss mechanism during training to force the encoders to learn relevant semantic representations. It is precisely the representations produced by the Context Encoder that contain the general knowledge of the world.

On the other hand, in some formulations it is considered that the same situation could have several compatible futures: the Predictor then receives an additional latent variable from which it selects among different possible predictions. This is not universal to all JEPA (the example we will see next, I-JEPA, is deterministic), but when it is present it can be worth keeping the Predictor as well, to anticipate the evolution of the latent state. This issue is especially important in video and robotics, where the future is not fully determined by the current observation.

Let's look at a more concrete example.

I-JEPA: learning from regions of an image

I-JEPA⁶² (*Image-based JEPA*) brings this principle to visual learning. From the same image it builds:

- a **context block**, made up of visible, spatially distributed patches,
- several **target blocks**, corresponding to regions that the context encoder cannot observe,
- a positional encoding for each target block.

The context encoder processes only the visible patches. The target encoder computes the representations of the target blocks, and the predictor tries to recover them from the context and their position. It does not reconstruct the original pixels, but instead predicts the embeddings that the target encoder would have produced for those regions.

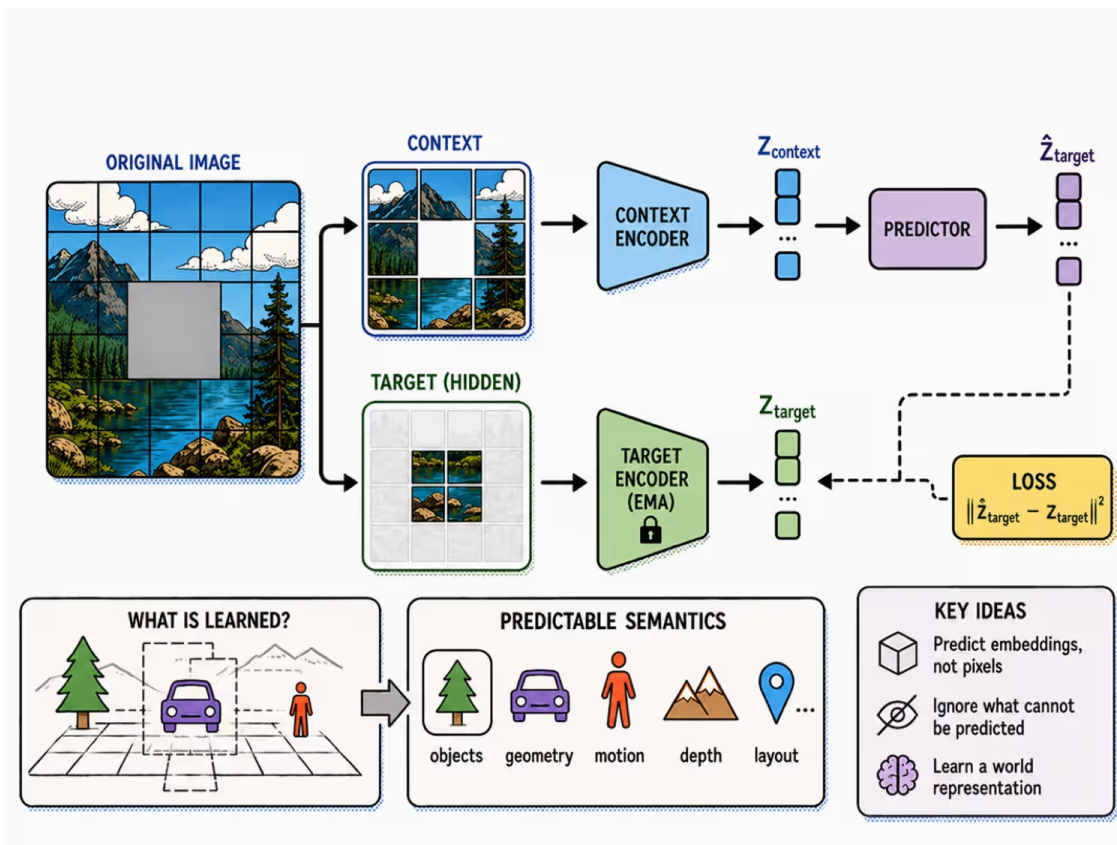


Figure 8.5. JEPA architecture with a context encoder, a target encoder, and a predictor

The masking pattern matters. If the targets are too small, the model can solve the task using local textures; if the context contains little information, the target becomes impossible to anticipate. I-JEPA uses relatively large target blocks to favor semantic properties, and a distributed context that provides enough information about the scene.

V-JEPA: From images to video

In **V-JEPA**⁶³, the related views come from a video and masking extends across space and time. The model predicts the representations of hidden spatio-temporal regions from the visible video. This forces the encoder to capture both appearance and motion, without reconstructing pixels, using text, relying on negative examples, or starting from a pretrained image encoder.

The collapse problem

During training there exists a useless solution that would nonetheless minimize the error: having every input produce exactly the same embedding. In that case, the predictor would always be correct without learning anything about the data:

$$f(x) = c \quad \forall x$$

This is called **complete collapse**, but there is a subtler variant: the embeddings can be different and yet concentrate almost all of their variability in a few dimensions. In this **dimensional collapse**, the model wastes much of the latent space and produces poor representations for downstream tasks.

Joint embedding methods need to prevent these problems. Contrastive approaches do so by pushing negative examples apart. Other methods use normalization, terms that enforce variance and decorrelation, a slowly updated target encoder, or *stop-gradient*.

I-JEPA uses two related mechanisms:

- The gradient of the loss updates the context encoder and the predictor, but not the target encoder.
- The parameters of the target encoder follow an **exponential moving average** of the context encoder's parameters. The target changes slowly and provides a more stable reference.

These mechanisms work, but they introduce design decisions: the speed of the moving average, asymmetry between branches, the predictor, and the masking strategy. Much of the research on joint embeddings has focused on finding recipes that avoid collapse without degrading the quality of the representations. This is precisely the problem LeJEPA tries to address.

8.3.7 LeJEPA

LeJEPA⁶⁴ (*Latent-Euclidean JEPA*) is a formulation proposed by Randall Balestriero and Yann LeCun that attempts to replace those heuristic recipes with an explicit criterion for how the latent space should be organized.

LeJEPA starts from two minimal requirements:

1. **Predictability**: representations of related views should be able to predict one another.
2. **Non-degeneracy**: embeddings must occupy the latent space in a way that is suitable for downstream tasks.

The first requirement provides invariance: two related views of the same sample should preserve the same semantic information. On its own, however, it allows all views and all samples to be represented by the same single point. The second requirement must rule out both complete collapse and dimensional collapse.

The authors analyze which distribution of embeddings would, under the assumptions of their theoretical framework, reduce the expected risk across a broad family of downstream tasks. Their result proposes an **isotropic Gaussian** as the objective.

What is an isotropic Gaussian?

It is expressed as

$$z \sim \mathcal{N}(0, I)$$

The embeddings form a multidimensional Gaussian distribution, centered at zero, as we have already seen several times in this course. **Isotropic** means that it has the same spread in every direction. Its covariance matrix is the identity I , so every direction has unit variance and there is no dominant axis.

Geometrically, in two dimensions it would look like a circular cloud, not a narrow ellipse. If we rotated the space, the distribution would keep the same shape.

This geometry prevents all points from coinciding, some dimensions from staying constant, or information from concentrating in a small subspace. Requiring only zero mean and identity covariance does not, however, control the whole shape of the distribution, since very different distributions can share those two first properties.

SIGReg

Checking and directly regularizing a distribution with hundreds or thousands of dimensions would be very costly. **Sketched Isotropic Gaussian Regularization (SIGReg)** reduces the problem to multiple one-dimensional checks.

For each random unit direction a_m , the embedding is projected:

$$u_m = a_m^\top z$$

If z follows an isotropic Gaussian, any projection u_m must follow a one-dimensional Gaussian $\mathcal{N}(0, 1)$. SIGReg samples several directions, compares the empirical distribution of those projections with the target Gaussian, and resamples directions during training. This way it can detect a collapsed direction without explicitly constructing or comparing high-dimensional densities.

The proposed implementation uses a test based on characteristic functions, related to the Epps–Pulley statistic. This choice provides bounded gradients and a cost that grows linearly with the number of samples, which makes it possible to apply the regularization to large embeddings and models.

The LeJEPa loss

Let $z_{n,v}$ be the representation of view v of sample n . LeJEPa builds a center μ_n from the V_g global views. For a given sample n , the following expression is the **per-sample predictive loss**:

$$\mu_n = \frac{1}{V_g} \sum_{v=1}^{V_g} z_{n,v}, \quad \mathcal{L}_{pred}^{(n)} = \frac{1}{V} \sum_{v=1}^V \|z_{n,v} - \mu_n\|_2^2$$

This simplified equation pulls together the views of the same sample. The full formulation averages

that term over the B samples in the batch and applies SIGReg separately to each view, using the embeddings from the whole batch:⁶⁵

$$\mathcal{L}_{LeJEP A} = \frac{1-\lambda}{B} \sum_{n=1}^B \mathcal{L}_{pred}^{(n)} + \frac{\lambda}{V} \sum_{v=1}^V \text{SIGReg}(\{z_{n,v}\}_{n=1}^B)$$

The two terms exert complementary forces:

- The average of $\mathcal{L}_{pred}^{(n)}$ pulls together the views of each sample and removes irrelevant variation.
- SIGReg organizes, for each view, the representations of the samples in the batch and prevents them from ending up at the same point or subspace.

The only hyperparameter specific to the combination is λ , which regulates the balance between predictability and diversity.

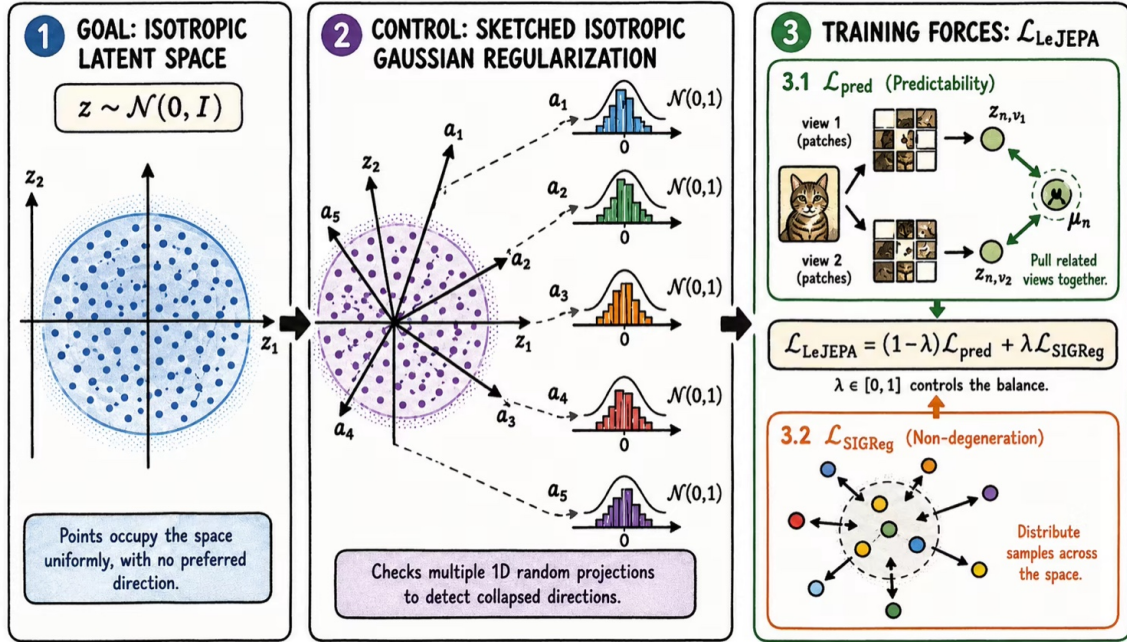


Figure 8.6. Statistical regularization of the representation space in LeJEP A

What it simplifies

By preventing collapse through an explicit objective on the distribution, LeJEP A's recommended recipe can use the same encoder for the different views and dispense with several usual components, such as:

- negative examples and large sample queues

- *teacher-student* networks and their moving-average schedule
- *stop-gradient*
- a predictor used solely to avoid collapse
- prototypes or special output dimensions

Another practical property is that the training loss reported by the authors correlates with the quality of the representations as measured by *linear probing*. In many self-supervised methods a lower loss does not necessarily imply a better encoder, which forces supervised evaluations to be run during development. If this relationship holds in new domains, it would make it possible to select models without relying on labels.

8.3.8 World models

We began the JEPA section by noting that these architectures were put forward as part of a proposal for building machines capable of learning internal models of the world. This idea connects directly with **world models**.

A **world model** is a learned model of the dynamics of an environment: given a representation of its current state, it predicts how it may evolve. If there is an agent acting on the environment, the prediction must also depend on the action it takes.

World model

An internal model that approximates the transitions/changes of the environment. It allows the consequences of an action to be anticipated without executing it in the real world.

In simplified form, it contains two pieces:

1. A **state encoder** transforms the observation x_t (or the recent history) into a latent state z_t .
2. A **transition model** predicts the next state from the current one and the action a_t :

$$z_t = E_\theta(x_{\leq t}), \quad \hat{z}_{t+1} = F_\phi(z_t, a_t)$$

Depending on the application, the world model can also predict rewards, events, or a distribution over future states instead of a single state.

What JEPA contributes

A world model needs a state space that retains what determines the evolution of the environment. Foundation models obtained via JEPA provide precisely a starting point for building it.

However, a pretrained encoder is not yet a world model. Learning dynamics requires temporally ordered data and a transition predictor. Actions are needed when we want a model conditioned on an agent's intervention and want to use it for control.

This relationship can be summarized as follows:

$$\underbrace{E_\theta}_{\text{JEPA foundation model}} + \underbrace{F_\phi(z_t, a_t)}_{\text{environment dynamics}} = \underbrace{\text{latent world model}}_{\text{internal simulator}}$$

Simulation and planning

Once learned, the world model can be used as a **latent simulator**. If we have a target state z_g , we can:

1. Encode the current observation as z_t and the goal as z_g .
2. Propose several action sequences.
3. Roll out their consequences in latent space via F_ϕ .
4. Measure which predicted final state ends up closest to z_g .
5. Execute the best action and plan again with the new observation.

The system only simulates the latent variables needed to compare their consequences. This can be much more efficient than generating full observations, and it avoids spending capacity on details that are irrelevant to the decision.

In a robotic arm, for example, z_t can encode the position of a cup and of the arm, the opening of the gripper, spatial relationships, and possible contacts. The world model allows the gripper's movements to be rehearsed before being executed physically. It can also be used as a simulator to train policies, generate hypothetical trajectories, or assess whether an action could cause a collision.

V-JEPA 2⁶⁶ is an example of this transition. After visual pretraining, its V-JEPA 2-AC variant incorporates robotic trajectories with actions to learn the dynamics $F_\phi(z_t, a_t)$. The predictor is then used to plan reaching, grasping, and placing tasks from a visual goal.

Note

The world model is not a complete agent. It needs a goal, a mechanism for proposing actions, a function for comparing them, and a planning procedure. Uncertainty does not disappear either just because we work with embeddings. If several semantically distinct futures exist, the model must represent multiple hypotheses or use latent variables.

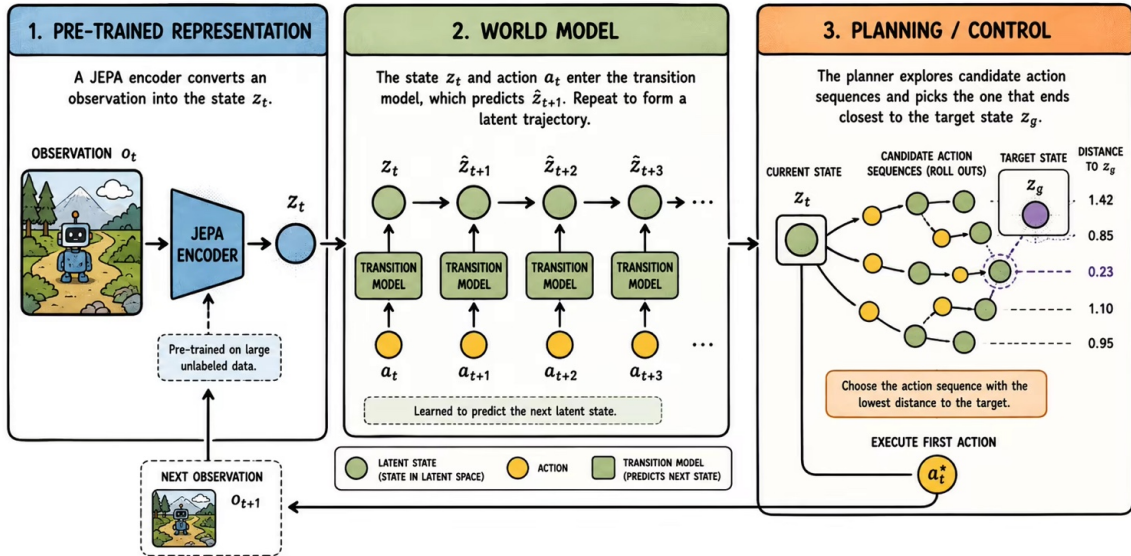


Figure 8.7. World model predicting future states in the representation space

8.4 Comparison

Paradigm	What is hidden or related	Target	Target space	How it avoids a trivial solution	Architecture
Autoregressive	The future	Next element	Data	Must explain the actual token	Decoder
Masked modeling	Parts of the input	Hidden tokens or pixels	Observation space	Must reconstruct the actual content	Encoder or encoder-decoder
Contrastive	Two related views	Correct pair	Latent	Negatives or batch distribution	Encoder
CPC	Future of the sequence	Correct future representation	Latent	Contrastive negatives	Predictive encoder
JEPA	Missing or future region	Target representation	Latent	Teacher, stop-gradient, or other mechanisms	Encoder
LeJEPA	Context and target views	Representation + regularized geometry	Latent	SIGReg	Encoder

**Note that this table does not describe fully mutually exclusive categories.*

8.5 Downstream tasks

The result of self-supervised learning is not an assistant, a classifier, or a search system. It is a base on which to build a useful system. For that, there are different options:

8.5.1 Linear probing

The encoder is frozen and a linear layer is trained on its representations with a labeled dataset. It is a common way to measure how much accessible information the embedding contains without modifying the pretrained model.

8.5.2 Fine-tuning

Training continues on all or part of the parameters using examples from the final task. It usually provides more adaptation capacity, but requires storing and updating a large model.

8.5.3 Efficient adaptation

Methods like LoRA⁶⁷ update a reduced number of parameters and keep most of the base model frozen. We will cover them in the [next chapter](#).

8.5.4 Prompting and in-context learning

In a generative model, the input itself can describe the task and include examples. The model adapts its behavior through the context without its parameters changing:

Classify the review as positive or negative. Review: "The story is slow, but the characters are wonderful."

This mechanism reduces the need to create a different output head for each task: many applications can be expressed through a common text or token interface.

8.5.5 Post-training and alignment

LLMs pretrained solely to predict the next token do not automatically behave like assistants. After pretraining, instruction data, demonstrations, and preferences are used to improve their ability to follow requests.

This is where techniques such as the following come in:

- **Supervised fine-tuning (SFT)** on instruction-response pairs.
- **RLHF**⁶⁸, which uses human preferences to train a reward signal and optimize the model.
- Direct preference optimization methods, such as DPO⁶⁹.

These techniques belong to **post-training**. They are not self-supervised paradigms comparable to MLM, CPC, or JEPA, because they use additional supervision to modify the behavior of an already pretrained base.

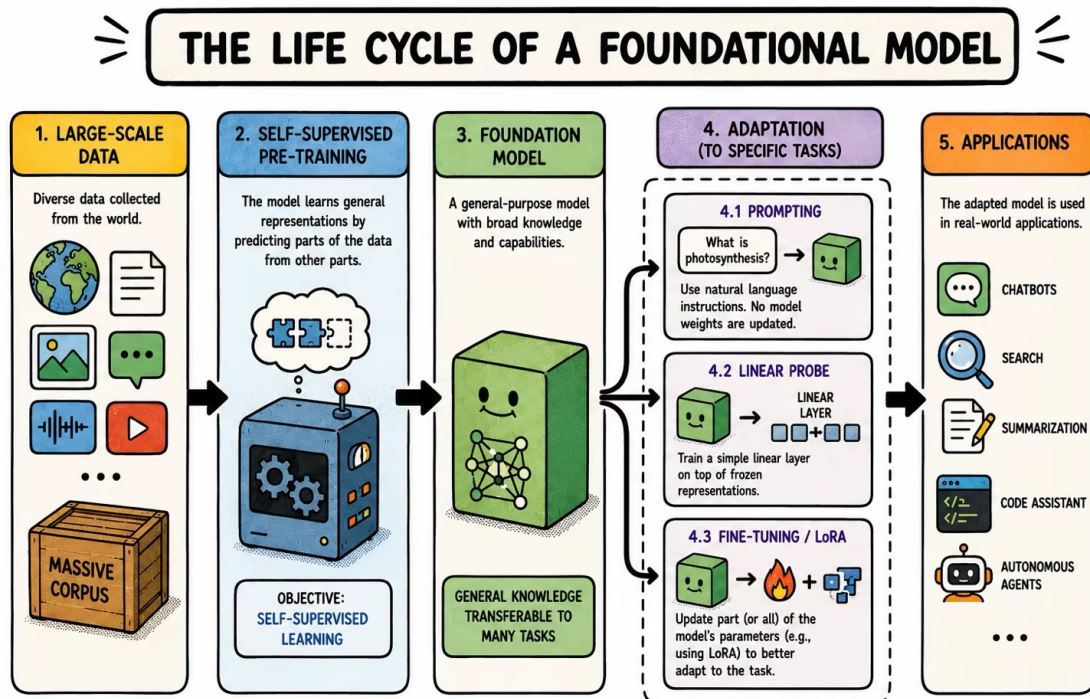


Figure 8.8. Life cycle of a foundation model from pretraining to deployment

CHAPTER 9

Efficiency and cost reduction

Chapter code: github.com/NahuelCostaCortez/generative-networks

Today's generative models can contain billions of parameters. Training, adapting, and serving them requires memory to store their weights and activations, and computing power to carry out the operations.

There is no single way to reduce this cost. In this chapter we will look at three complementary strategies:

1. **Distillation:** training a smaller model to reproduce the behavior of a more capable one.
2. **Quantization:** representing weights and, in some cases, activations with fewer bits.
3. **LoRA:** adapting a pretrained model by updating only a small number of parameters.

9.1 Knowledge distillation

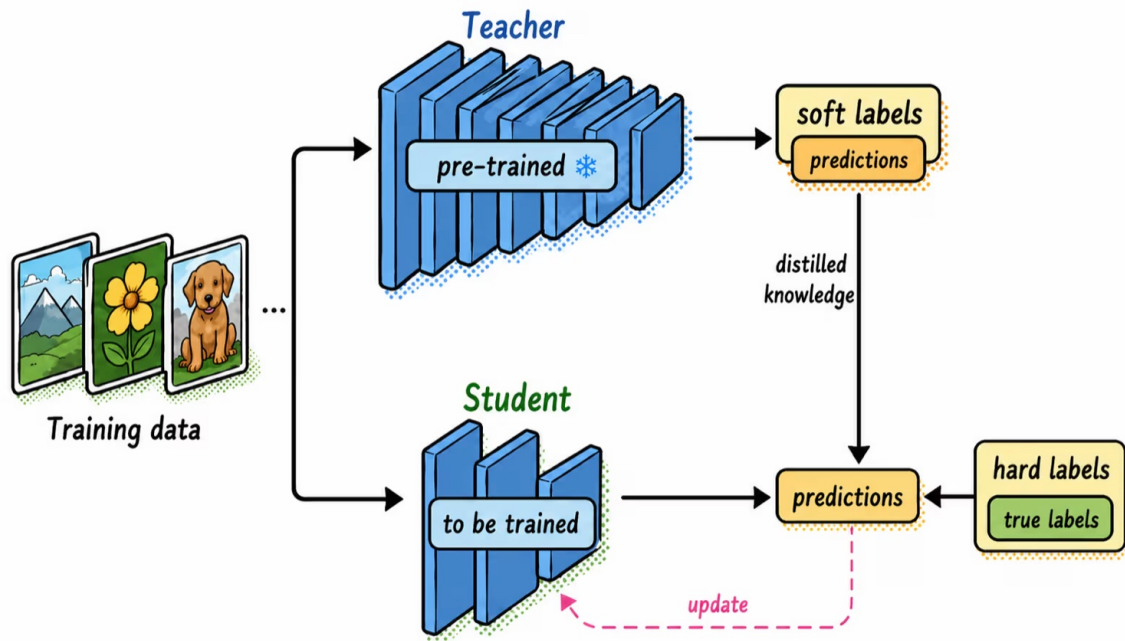


Figure 9.1. A teacher model transfers knowledge to a smaller student model

Knowledge distillation⁷⁰ is a compression technique in which a large model, the **teacher**, supervises the training of a smaller model, the **student**.

The goal is to approximate the function the teacher has learned. If the transfer works, the student retains a significant part of its quality but with less memory, lower latency, and a lower inference cost. This approach is useful for producing lightweight models that can be deployed on mobile devices, edge devices, or services with heavy request volumes.

9.1.1 Hard labels and soft targets

In a conventional classification problem, a hard label indicates only the correct class. For an image of a cat, for example, the target vector might be:

$$y = [1, 0, 0]$$

for the classes cat, leopard, and dog. The teacher, on the other hand, can produce a distribution such as $[0.80, 0.15, 0.05]$. These probabilities contain additional information: they show that, for the

teacher, a cat is more similar to a leopard than to a dog. These relationships between classes are usually referred to as **dark knowledge**.

Soft targets are obtained by applying *softmax* to the logits (the model's unnormalized outputs) with a **temperature** τ :

$$p_i^{(\tau)} = \frac{\exp(z_i/\tau)}{\sum_j \exp(z_j/\tau)}$$

When $\tau = 1$, the usual *softmax* is obtained. A higher temperature flattens the distribution and makes small probabilities visible, better exposing the similarities learned by the teacher.

What does the student learn?

The hard label only tells you which is the correct answer. The teacher's distribution also communicates how it distributes its uncertainty across different alternatives. The student learns to imitate that **output behavior**, not necessarily the internal mechanism or the exact reasoning of the teacher.

9.1.2 Loss function

Let $p_T^{(\tau)}$ and $p_S^{(\tau)}$ be the smoothed distributions of the teacher and the student. A common formulation combines two objectives:

$$\mathcal{L}_{\text{total}} = \alpha \tau^2 D_{\text{KL}}(p_T^{(\tau)} \| p_S^{(\tau)}) + (1 - \alpha) \text{CE}(y, p_S^{(1)})$$

1. The **distillation loss** forces the student to approximate the teacher's distribution. It can be expressed using Kullback–Leibler divergence or cross-entropy.
2. The **supervised loss** compares the student's prediction with the true label y .

The hyperparameter α controls the relative weight of both terms. The factor τ^2 compensates for the change in gradient scale produced by the temperature. It is not always essential to have human labels: unlabeled data or synthetic examples can also be used, training solely with the teacher's supervision.

9.1.3 Distillation in generative models

In an LLM, knowledge can be distilled at different levels:

- **Logits**: the student approximates the probability distribution over the next token. This approach requires being able to query the teacher's outputs and properly align vocabularies.
- **Sequences**: the teacher generates responses that become training data for the student. This is especially practical when only generated text is accessible.
- **Intermediate representations**: besides the output, hidden states or attention maps are aligned when the architectures allow it.

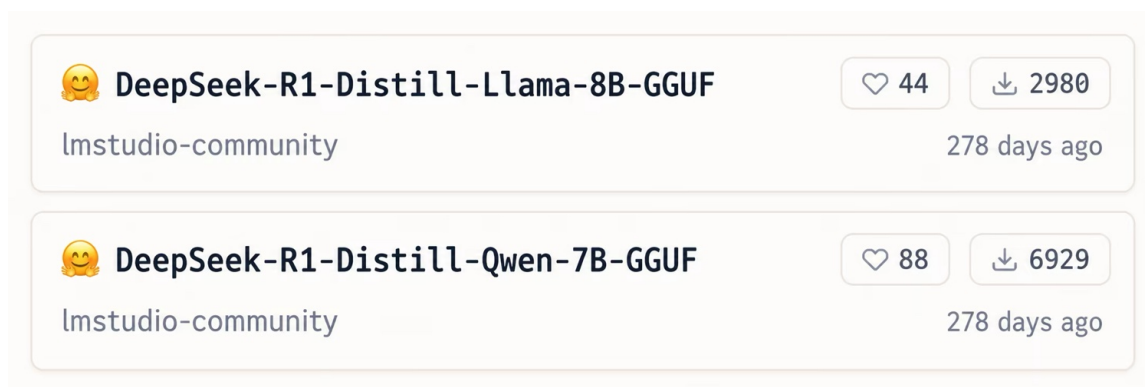


Figure 9.2. Family of small models distilled from DeepSeek-R1 responses

This is an example of sequence-level distillation: DeepSeek-R1 generates and filters solutions, and smaller models based on Qwen and Llama learn from those samples. They are not directly copying its logits.

Distillation has an upfront cost, the teacher must be run and the student must be trained, and it can transmit its errors or biases. Its usefulness depends on the accumulated savings during deployment offsetting that cost, and on the loss of quality being acceptable.

9.2 Quantization

Quantization is a technique for reducing computational and memory costs during inference by representing weights and activations with low-precision data types, such as 8-bit integers (`int8`) instead of the usual floating-point `float32` or `float16`.

Reducing the number of bits means that the resulting model requires less storage in memory, consumes less energy, and operations such as matrix multiplication can be performed much faster.

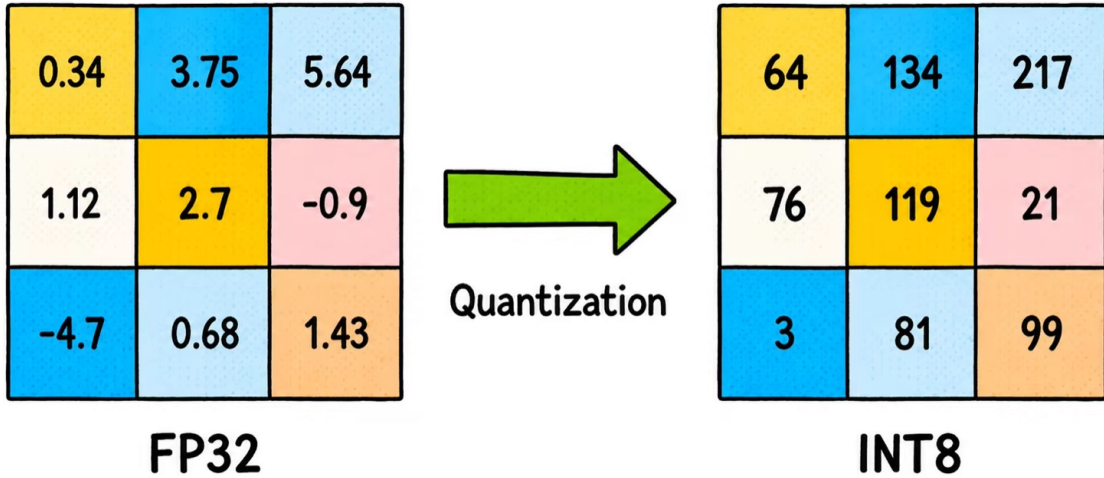


Figure 9.3. Continuous values grouped into a finite number of quantization levels

The most widely used standard method is uniform affine quantization, which maps real values onto a grid of integers through a linear transformation:

- Affine (asymmetric): allows the real value 0.0 to be aligned with an arbitrary integer z_0 (zero point), which avoids distortions when the data is not symmetrically centered around zero (as happens after ReLU-type activations).
- Uniform: the distance between two consecutive integers always represents the same fixed interval in the real-valued space (the scale factor s).

Mapping a real value x to an integer q is done as follows:

$$q = \text{clip}\left(\text{round}\left(\frac{x}{s}\right) + z_0, q_{\min}, q_{\max}\right)$$

During inference, an approximation of $\hat{x}$ is reconstructed:

$$\hat{x} = s(q - z_0)$$

9.2.1 Example - INT8 (range [0, 255])

Suppose the weights of a layer range between $x_{\min} = -1.0$ and $x_{\max} = 3.0$:

1. Scale (s): $s = \frac{x_{\max} - x_{\min}}{q_{\max} - q_{\min}} = \frac{3.0 - (-1.0)}{255 - 0} \approx 0.0157$
2. Zero point (z_0): $z_0 = \text{round}\left(-\frac{x_{\min}}{s}\right) = \text{round}\left(\frac{1.0}{0.0157}\right) = 64$
3. Quantize a value ($x = 1.2$):

$$q = \text{clip}\left(\text{round}\left(\frac{1.2}{0.0157}\right) + 64, 0, 255\right) = \text{clip}(76 + 64, 0, 255) = 140$$

4. Reconstruct ($\hat{x}$):

$$\hat{x} = 0.0157 \cdot (140 - 64) = 1.1932$$

The discrepancy between 1.2 and 1.1932 is the quantization error, generated by the rounding operation. To reduce this margin of error, the granularity at which s and z_0 are computed is adjusted:

- Per tensor: all values in a layer share a single pair (s, z_0) . It is very memory-efficient, but vulnerable to outliers.
- Per channel: each output channel has its own scale, balancing precision and computational cost.
- Per group: weights are divided into small blocks (e.g. 32 to 128 elements) with an independent scale, a technique fundamental to 4-bit LLMs (such as AWQ or GPTQ).

9.2.2 Which part of the model gets quantized?

1. **Weights only** (*weight-only*): considerably reduces memory and is common in LLMs. Activations are kept at higher precision.
2. **Weights and activations**: can speed up computation further, but activations are harder to quantize because they change with each input and can contain outliers.
3. **KV cache**: in autoregressive models with long contexts, quantizing the cache reduces the memory that grows with the number of tokens processed.

The theoretical storage savings are easy to estimate. For example, for a 7B model:

Weight format	Approximate memory
<code>float32</code> (32 bits)	28 GB
<code>float16</code> / <code>bfloat16</code> (16 bits)	14 GB
<code>int8</code> (8 bits)	7 GB
4 bits	3.5 GB

Accumulation data type

The storage format does not have to match the format used to accumulate results. In a matrix multiplication, many products are summed; even if each operand is `int8`, the accumulator is usually `int32` to avoid overflow. Similarly, many computations with `float16` or `bfloat16` accumulate in

float32.

The exact detail depends on the operator and the hardware. That is why the nominal weight format alone is not enough to infer the internal precision or the final speed.

9.2.3 PTQ and QAT

There are two main points at which quantization can be introduced:

- **Post-Training Quantization (PTQ)**: an already trained model is quantized. It is fast and may only require a small calibration set to estimate scales. Methods such as GPTQ⁷¹ correct part of the error when quantizing LLMs weight by weight or block by block.
- **Quantization-Aware Training (QAT)**: quantization operations are simulated during training so the model learns to tolerate the error⁷². It tends to better preserve quality at aggressive precisions, but it requires retraining.

Quantizing always introduces a trade-off between **size**, **speed**, and **quality**. Layers do not all have the same sensitivity, and it is sometimes worth keeping some of them at higher precision. Moreover, a model that is four times smaller will not necessarily be four times faster: latency can be dominated by sequential generation, data transfer, or the lack of native support for the chosen format.

9.3 Low-Rank Adaptation (LoRA)

When training large generative models, adjusting all the parameters can be very costly, especially when the only goal is to adapt the model to a specific task.

In full fine-tuning, every weight matrix W_0 must have all of its parameters updated:

$$W_{\text{updated}} = W_0 + \Delta W$$

If $W_0 \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$, training ΔW requires up to $d_{\text{out}}d_{\text{in}}$ parameters.

Low-Rank Adaptation (LoRA)⁶⁷ is an efficient fine-tuning method, or **PEFT** (*Parameter-Efficient Fine-Tuning*), that seeks to reduce this burden using a technique called low-rank decomposition.

The central idea is to freeze the original weight matrix W_0 (some or all of the model's matrices) and, instead, decompose the update ΔW into the product of two smaller matrices:

$$\Delta W = \frac{\alpha}{r} BA, \quad A \in \mathbb{R}^{r \times d_{\text{in}}}, \quad B \in \mathbb{R}^{d_{\text{out}} \times r}$$

where α is a hyperparameter that scales the contribution of the low-rank update (often set relative to r , hence the ratio α/r).

This way, instead of training $d_{\text{out}}d_{\text{in}}$ values, only $r(d_{\text{in}} + d_{\text{out}})$ need to be trained. This significantly reduces the number of parameters that need to be adjusted, making the process more efficient.

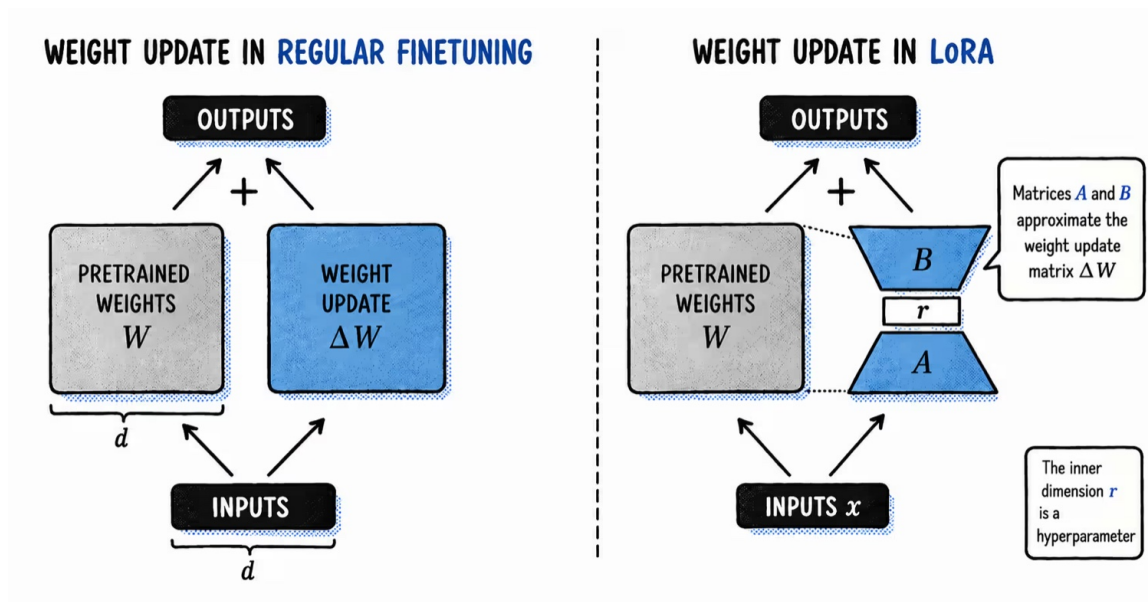


Figure 9.4. LoRA represents the update to a weight matrix using two low-rank matrices

Note

A and B do not represent the entire matrix W , only the change or update needed for the new task: ΔW .

Large models are generally overparameterized. Even though W_0 has enormous dimensions, the changes needed to learn a specific task live in a very low-dimensional subspace.

Example. Suppose we want to adapt a square weight matrix W_0 of size 4096×4096 (a typical LLM layer), with 16 777 216 parameters. Full fine-tuning of that matrix would update the 16.8 M parameters. With $r = 8$, LoRA trains:

$$8(4096 + 4096) = 65\,536$$

This represents approximately 0.39% of that matrix's parameters.

There is a trade-off in choosing the rank. A rank that is too small may fail to capture the target task well. A very large rank improves capacity, but also increases cost and the risk of overfitting.

LoRA does not compress the base model

LoRA reduces the **trainable** parameters, but running the model still requires loading the weights of W_0 .

In general, there are two options:

- Merge the weights: compute the product $\Delta W = \frac{\alpha}{r}BA$ and add it directly to the original matrix: $W_{\text{final}} = W_0 + \Delta W$. The model is saved with W_{final} as a single standard file. This eliminates any latency or compute overhead at inference, but modularity is lost.
- Keep the adapter decoupled: only the file with the weights A and B is saved. At inference, the base model W_0 is loaded into memory and the input travels through two paths in parallel. On one hand, the base path, where $h_{\text{base}} = Wx$ is computed, and on the other, the LoRA path, where the adjustment is computed as $B \cdot (A \cdot x)$. Both paths are then summed to obtain the final response.

To reduce memory usage during inference, LoRA can be combined with quantization. A popular combination is **QLoRA**⁷³: it keeps the base model frozen and quantized, for example to 4 bits, and propagates the gradient through higher-precision LoRA adapters. This reduces both the memory of the base model and the memory associated with training.

9.3.1 Contributions

- **Efficiency**: allows models to be adapted without an exhaustive retraining, saving time and computational resources.
- **Simplicity**: for each task, only an adapter needs to be saved instead of a full copy of the model.

9.4 How to choose a technique

Main goal	Technique	What is reduced	Main trade-off
Create a small model to serve many times	Distillation	Parameters and inference compute	Training a student and accepting some loss of quality
Adapt a model on a tight budget	LoRA	Trainable parameters and optimizer states	The base model is still required
Load or run a model with less memory	Quantization	Bits per weight and, optionally, per activation	Numerical error and dependence on hardware
Adapt a model that barely fits in memory	QLoRA	Base model memory and <i>fine-tuning</i> memory	More complex training and quantization operations

These techniques can be chained together. For example, one can distill a large model, adapt the

student with LoRA, and quantize the result for deployment.

References

1.

```
@article{shannon1948mathematical,  
  title={A Mathematical Theory of Communication},  
  author={Shannon, Claude E.},  
  journal={The Bell System Technical Journal},  
  volume={27},  
  year={1948}  
}
```
2.

```
@article{salimans2016improved,  
  title={Improved Techniques for Training GANs},  
  author={Salimans, Tim and Goodfellow, Ian and Zaremba, Wojciech and Cheung, Vicki and Radford,  
    ↪ Alec and Chen, Xi},  
  journal={arXiv preprint arXiv:1606.03498},  
  year={2016}  
}
```
3.

```
@article{goodfellow2014generative,  
  title={Generative Adversarial Nets},  
  author={Goodfellow, Ian J. and Pouget-Abadie, Jean and Mirza, Mehdi and Xu, Bing and  
    ↪ Warde-Farley, David and Ozair, Sherjil and Courville, Aaron and Bengio, Yoshua},  
  journal={Advances in Neural Information Processing Systems},  
  volume={27},  
  year={2014}  
}
```
4.

```
@article{brock2018large,  
  title={Large Scale GAN Training for High Fidelity Natural Image Synthesis},  
  author={Brock, Andrew and Donahue, Jeff and Simonyan, Karen},  
  journal={arXiv preprint arXiv:1809.11096},  
  year={2018}  
}
```
5.

```
@article{karras2019style,  
  title={A Style-Based Generator Architecture for Generative Adversarial Networks},  
  author={Karras, Tero and Laine, Samuli and Aila, Timo},  
  journal={arXiv preprint arXiv:1812.04948},  
  year={2018}  
}
```

6. `@article{engel2019gansynth,
 title={GANSynth: Adversarial Neural Audio Synthesis},
 author={Engel, Jesse and Agrawal, Kumar Krishna and Chen, Shuo and Gulrajani, Ishaan and
 ↪ Donahue, Chris and Roberts, Adam},
 journal={arXiv preprint arXiv:1902.08710},
 year={2019}
}`
7. `@article{kaneko2020cyclegan,
 title={CycleGAN-VC3: Examining and Improving CycleGAN-VCs for Mel-spectrogram Conversion},
 author={Kaneko, Takuhiro and Kameoka, Hirokazu and Tanaka, Kou and Hojo, Nobukatsu},
 journal={arXiv preprint arXiv:2010.11672},
 year={2020}
}`
8. `@article{donahue2018adversarial,
 title={Adversarial Audio Synthesis},
 author={Donahue, Chris and McAuley, Julian and Puckette, Miller},
 journal={arXiv preprint arXiv:1802.04208},
 year={2018}
}`
9. `@article{isola2017image,
 title={Image-to-Image Translation with Conditional Adversarial Networks},
 author={Isola, Phillip and Zhu, Jun-Yan and Zhou, Tinghui and Efros, Alexei A.},
 journal={arXiv preprint arXiv:1611.07004},
 year={2016}
}`
10. `@article{hinton2006reducing,
 title={Reducing the Dimensionality of Data with Neural Networks},
 author={Hinton, Geoffrey E. and Salakhutdinov, Ruslan R.},
 journal={Science},
 volume={313},
 year={2006}
}`
11. `@article{vincent2008extracting,
 title={Extracting and Composing Robust Features with Denoising Autoencoders},
 author={Vincent, Pascal and Larochelle, Hugo and Bengio, Yoshua and Manzagol, Pierre-Antoine},
 journal={International Conference on Machine Learning},
 year={2008}
}`
12. `@article{makhzani2013k,
 title={k-Sparse Autoencoders},
 author={Makhzani, Alireza and Frey, Brendan},
 journal={arXiv preprint arXiv:1312.5663},
 year={2013}
}`
13. `@article{rifai2011contractive,
 title={Contractive Auto-Encoders: Explicit Invariance During Feature Extraction},
 author={Rifai, Salah and Vincent, Pascal and Muller, Xavier and Glorot, Xavier and Bengio,
 ↪ Yoshua},
 journal={International Conference on Machine Learning},
 year={2011}
}`

14. `@article{kingma2013auto,`
`title={Auto-Encoding Variational Bayes},`
`author={Kingma, Diederik P and Welling, Max},`
`journal={arXiv preprint arXiv:1312.6114},`
`year={2013}`
`}`
15. `@article{yang2017improved,`
`title={Improved Variational Autoencoders for Text Modeling using Dilated Convolutions},`
`author={Yang, Zichao and Hu, Zhiting and Salakhutdinov, Ruslan and Berg-Kirkpatrick, Taylor},`
`journal={International Conference on Machine Learning},`
`year={2017}`
`}`
16. `@article{costa2022variational,`
`title={Variational Encoding Approach for Interpretable Assessment of Remaining Useful Life`
`↪ Estimation},`
`author={Costa, Nahuel and Sánchez, Luciano},`
`journal={Reliability Engineering & System Safety},`
`volume={222},`
`year={2022}`
`}`
17. `@article{costa2021semi,`
`title={Semi-Supervised Recurrent Variational Autoencoder Approach for Visual Diagnosis of`
`↪ Atrial Fibrillation},`
`author={Costa, Nahuel and Sanchez, Luciano and Couso, Ines},`
`journal={IEEE Access},`
`volume={9},`
`year={2021}`
`}`
18. `@article{de2023population,`
`title={Population analysis and the effects of Gaussian basis set quality and quantum`
`↪ mechanical approach: main group through heavy element species},`
`author={North, Sasha C. and Jorgensen, Kameron R. and Pricetolstoy, Jason and Wilson, Angela`
`↪ K.},`
`journal={Frontiers in Chemistry},`
`volume={11},`
`year={2023}`
`}`
19. `@article{higgins2017beta,`
`title={beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework},`
`author={Higgins, Irina and Matthey, Loic and Pal, Arka and Burgess, Christopher and Glorot,`
`↪ Xavier and Botvinick, Matthew and Mohamed, Shakir and Lerchner, Alexander},`
`journal={International Conference on Learning Representations},`
`year={2017}`
`}`
20. `@article{zhao2017infovae,`
`title={InfoVAE: Information Maximizing Variational Autoencoders},`
`author={Zhao, Shengjia and Song, Jiaming and Ermon, Stefano},`
`journal={arXiv preprint arXiv:1706.02262},`
`year={2017}`
`}`

21. `@article{wu2018multimodal,`
`title={Multimodal Autoencoder Predicts fNIRS Resting State From EEG Signals},`
`author={Sirpal, Parikshat and Damseh, Rafat and Peng, Ke and Nguyen, Dang Khoa and Lesage,`
`↪ Frédéric},`
`journal={Neuroinformatics},`
`volume={20},`
`year={2021}`
`}`
22. `@article{shi2019variational,`
`title={Variational Mixture-of-Experts Autoencoders for Multi-Modal Deep Generative Models},`
`author={Shi, Yuge and N, Siddharth and Paige, Brooks and Torr, Philip},`
`journal={Advances in Neural Information Processing Systems},`
`volume={32},`
`year={2019}`
`}`
23. `@article{child2020very,`
`title={Very Deep VAEs Generalize Autoregressive Models and Can Outperform Them on Images},`
`author={Child, Rewon},`
`journal={arXiv preprint arXiv:2011.10650},`
`year={2020}`
`}`
24. `@article{kingma2019bit,`
`title={Bit-Swap: Recursive Bits-Back Coding for Lossless Compression with Hierarchical Latent`
`↪ Variables},`
`author={Kingma, Friso H. and Abbeel, Pieter and Ho, Jonathan},`
`journal={International Conference on Machine Learning},`
`volume={97},`
`year={2019}`
`}`
25. `@article{bennett1998information,`
`title={Information Distance},`
`author={Bennett, Charles H. and Gács, Péter and Li, Ming and Vitányi, Paul M. B. and Zurek,`
`↪ Wojciech H.},`
`journal={IEEE Transactions on Information Theory},`
`volume={44},`
`year={1998}`
`}`
26. `@article{kosaraju2000compression,`
`title={Compression of Low Entropy Strings with Lempel-Ziv Algorithms},`
`author={Kosaraju, S. Rao and Manzini, Giovanni},`
`journal={SIAM Journal on Computing},`
`volume={29},`
`year={2000}`
`}`
27. `@article{moffat1998arithmetic,`
`title={Arithmetic Coding Revisited},`
`author={Moffat, Alistair and Neal, Radford M. and Witten, Ian H.},`
`journal={ACM Transactions on Information Systems},`
`volume={16},`
`year={1998}`
`}`

28. `@article{moffat2019huffman,`
`title={Huffman Coding},`
`author={Moffat, Alistair},`
`journal={ACM Computing Surveys},`
`volume={52},`
`year={2019}`
`}`
29. `@article{frey1997efficient,`
`title={Efficient Stochastic Source Coding and an Application to a Bayesian Network Source`
`↪ Model},`
`author={Frey, Brendan J. and Hinton, Geoffrey E.},`
`journal={The Computer Journal},`
`year={1997}`
`}`
30. `@article{townsend2019practical,`
`title={Practical Lossless Compression with Latent Variables using Bits Back Coding},`
`author={Townsend, James and Bird, Tom and Barber, David},`
`journal={arXiv preprint arXiv:1901.04866},`
`year={2019}`
`}`
31. `@article{jiang2022few,`
`title={Few-Shot Non-Parametric Learning with Deep Latent Variable Model},`
`author={Jiang, Zhiying and Dai, Yiqin and Xin, Ji and Li, Ming and Lin, Jimmy},`
`journal={Advances in Neural Information Processing Systems},`
`volume={35},`
`year={2022}`
`}`
32. `@article{costa2025few,`
`title={Few-shot Generative Compression Approach for System Health Monitoring},`
`author={Costa, Nahuel and Sánchez, Luciano},`
`journal={Reliability Engineering & System Safety},`
`year={2025}`
`}`
33. `@article{nalisnick2018deep,`
`title={Do Deep Generative Models Know What They Don't Know?},`
`author={Nalisnick, Eric and Matsukawa, Akihiro and Teh, Yee Whye and Gorur, Dilan and`
`↪ Lakshminarayanan, Balaji},`
`journal={arXiv preprint arXiv:1810.09136},`
`year={2018}`
`}`
34. `@article{alsing2019fast,`
`title={Fast likelihood-free cosmology with neural density estimators and active learning},`
`author={Alsing, Justin and Charnock, Tom and Feeney, Stephen and Wandelt, Benjamin},`
`journal={arXiv preprint arXiv:1903.00007},`
`year={2019}`
`}`

35. `@article{goncalves2019training,`
`title={Training deep neural density estimators to identify mechanistic models of neural`
`↪ dynamics},`
`author={Gonçalves, Pedro J. and Lueckmann, Jan-Matthis and Deistler, Michael and Nonnenmacher,`
`↪ Marcel and Ücal, Kaan and Bassetto, Giacomo and Chintaluri, Chaitanya and Podlaski,`
`↪ William F. and Haddad, Sara A. and Vogels, Tim P. and Greenberg, David S. and Macke, Jakob`
`↪ H.},`
`journal={eLife},`
`volume={9},`
`year={2020}`
`}`
36. `@article{dinh2016density,`
`title={Density estimation using Real NVP},`
`author={Dinh, Laurent and Sohl-Dickstein, Jascha and Bengio, Samy},`
`journal={arXiv preprint arXiv:1605.08803},`
`year={2016}`
`}`
37. `@article{papamakarios2017masked,`
`title={Masked Autoregressive Flow for Density Estimation},`
`author={Papamakarios, George and Pavlakou, Theo and Murray, Iain},`
`journal={arXiv preprint arXiv:1705.07057},`
`year={2017}`
`}`
38. `@article{kingma2018glow,`
`title={Glow: Generative Flow with Invertible 1x1 Convolutions},`
`author={Kingma, Diederik P. and Dhariwal, Prafulla},`
`journal={arXiv preprint arXiv:1807.03039},`
`year={2018}`
`}`
39. `@article{nielsen2020survae,`
`title={SurVAE Flows: Surjections to Bridge the Gap between VAEs and Flows},`
`author={Nielsen, Didrik and Jaini, Priyank and Hoozeboom, Emiel and Winther, Ole and Welling,`
`↪ Max},`
`journal={Advances in Neural Information Processing Systems},`
`volume={33},`
`year={2020}`
`}`
40. `@article{lipman2022flow,`
`title={Flow Matching for Generative Modeling},`
`author={Lipman, Yaron and Chen, Ricky T. Q. and Ben-Hamu, Heli and Nickel, Maximilian and Le,`
`↪ Matt},`
`journal={arXiv preprint arXiv:2210.02747},`
`year={2022}`
`}`
41. `@article{tong2023improving,`
`title={Improving and generalizing flow-based generative models with minibatch optimal`
`↪ transport},`
`author={Tong, Alexander and Fatras, Kilian and Malkin, Nikolay and Huguët, Guillaume and Zhang,`
`↪ Yanlei and Rector-Brooks, Jarrod and Wolf, Guy and Bengio, Yoshua},`
`journal={arXiv preprint arXiv:2302.00482},`
`year={2023}`
`}`

42. `@article{sohl2015deep,
 title={Deep Unsupervised Learning using Nonequilibrium Thermodynamics},
 author={Sohl-Dickstein, Jascha and Weiss, Eric A. and Maheswaranathan, Niru and Ganguli,
 ↪ Surya},
 journal={arXiv preprint arXiv:1503.03585},
 year={2015}
}`
43. `@article{song2019generative,
 title={Generative Modeling by Estimating Gradients of the Data Distribution},
 author={Song, Yang and Ermon, Stefano},
 journal={arXiv preprint arXiv:1907.05600},
 year={2019}
}`
44. `@article{ho2020denoising,
 title={Denoising Diffusion Probabilistic Models},
 author={Ho, Jonathan and Jain, Ajay and Abbeel, Pieter},
 journal={Advances in Neural Information Processing Systems},
 volume={33},
 year={2020}
}`
45. `@article{kingma2021variational,
 title={Variational Diffusion Models},
 author={Kingma, Diederik P. and Salimans, Tim and Poole, Ben and Ho, Jonathan},
 journal={arXiv preprint arXiv:2107.00630},
 year={2021}
}`
46. `@article{rombach2022high,
 title={High-Resolution Image Synthesis with Latent Diffusion Models},
 author={Rombach, Robin and Blattmann, Andreas and Lorenz, Dominik and Esser, Patrick and Ommer,
 ↪ Björn},
 journal={arXiv preprint arXiv:2112.10752},
 year={2021}
}`
47. `@article{radford2021learning,
 title={Learning Transferable Visual Models From Natural Language Supervision},
 author={Radford, Alec and Kim, Jong Wook and Hallacy, Chris and Ramesh, Aditya and Goh,
 ↪ Gabriel and Agarwal, Sandhini and Sastry, Girish and Askell, Amanda and Mishkin, Pamela
 ↪ and Clark, Jack and Krueger, Gretchen and Sutskever, Ilya},
 journal={arXiv preprint arXiv:2103.00020},
 year={2021}
}`
48. `@article{mikolov2013efficient,
 title={Efficient Estimation of Word Representations in Vector Space},
 author={Mikolov, Tomas and Chen, Kai and Corrado, Greg and Dean, Jeffrey},
 journal={arXiv preprint arXiv:1301.3781},
 year={2013}
}`

49. `@article{vaswani2017attention,`
`title={Attention Is All You Need},`
`author={Vaswani, Ashish and Shazeer, Noam and Parmar, Niki and Uszkoreit, Jakob and Jones,`
`↪ Llion and Gomez, Aidan N. and Kaiser, Lukasz and Polosukhin, Illia},`
`journal={arXiv preprint arXiv:1706.03762},`
`year={2017}`
`}`
50. `@article{bahdanau2014neural,`
`title={Neural Machine Translation by Jointly Learning to Align and Translate},`
`author={Bahdanau, Dzmitry and Cho, Kyunghyun and Bengio, Yoshua},`
`journal={arXiv preprint arXiv:1409.0473},`
`year={2014}`
`}`
51. `@article{su2021roformer,`
`title={RoFormer: Enhanced Transformer with Rotary Position Embedding},`
`author={Su, Jianlin and Lu, Yu and Pan, Shengfeng and Murtadha, Ahmed and Wen, Bo and Liu,`
`↪ Yunfeng},`
`journal={arXiv preprint arXiv:2104.09864},`
`year={2021}`
`}`
52. `@article{dosovitskiy2020image,`
`title={An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale},`
`author={Dosovitskiy, Alexey and Beyer, Lucas and Kolesnikov, Alexander and Weissenborn, Dirk`
`↪ and Zhai, Xiaohua and Unterthiner, Thomas and Dehghani, Mostafa and Minderer, Matthias and`
`↪ Heigold, Georg and Gelly, Sylvain and Uszkoreit, Jakob and Houlsby, Neil},`
`journal={arXiv preprint arXiv:2010.11929},`
`year={2020}`
`}`
53. `@article{lewis2019bart,`
`title={BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation,`
`↪ Translation, and Comprehension},`
`author={Lewis, Mike and Liu, Yinhan and Goyal, Naman and Ghazvininejad, Marjan and Mohamed,`
`↪ Abdelrahman and Levy, Omer and Stoyanov, Ves and Zettlemoyer, Luke},`
`journal={arXiv preprint arXiv:1910.13461},`
`year={2019}`
`}`
54. `@article{raffel2019t5,`
`title={Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer},`
`author={Raffel, Colin and Shazeer, Noam and Roberts, Adam and Lee, Katherine and Narang,`
`↪ Sharan and Matena, Michael and Zhou, Yanqi and Li, Wei and Liu, Peter J.},`
`journal={arXiv preprint arXiv:1910.10683},`
`year={2019}`
`}`
55. `@article{devlin2018bert,`
`title={BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding},`
`author={Devlin, Jacob and Chang, Ming-Wei and Lee, Kenton and Toutanova, Kristina},`
`journal={arXiv preprint arXiv:1810.04805},`
`year={2018}`
`}`

56. `@article{bommasani2021foundation,`
`title={On the Opportunities and Risks of Foundation Models},`
`author={Bommasani, Rishi and Hudson, Drew A. and Adeli, Ehsan and Altman, Russ and Arora,`
`↪ Simran and von Arx, Sydney and Bernstein, Michael S. and Bohg, Jeannette and Bosselut,`
`↪ Antoine and Brunskill, Emma and Brynjolfsson, Erik and Buch, Shyamal and Card, Dallas and`
`↪ Castellon, Rodrigo and Chatterji, Niladri and Chen, Annie and Creel, Kathleen and Davis,`
`↪ Jared Quincy and Demszky, Dora and Donahue, Chris and Doumbouya, Moussa and Durmus, Esin`
`↪ and Ermon, Stefano and Etchemendy, John and Ethayarajh, Kawin and Fei-Fei, Li and Finn,`
`↪ Chelsea and Gale, Trevor and Gillespie, Lauren and Goel, Karan and Goodman, Noah and`
`↪ Grossman, Shelby and Guha, Neel and Hashimoto, Tatsunori and Henderson, Peter and Hewitt,`
`↪ John and Ho, Daniel E. and Hong, Jenny and Hsu, Kyle and Huang, Jing and Icard, Thomas and`
`↪ Jain, Saahil and Jurafsky, Dan and Kalluri, Pratyusha and Karamcheti, Siddharth and`
`↪ Keeling, Geoff and Khani, Fereshte and Khattab, Omar and Koh, Pang Wei and Krass, Mark and`
`↪ Krishna, Ranjay and Kuditipudi, Rohith and Kumar, Ananya and Ladhak, Faisal and Lee, Mina`
`↪ and Lee, Tony and Leskovec, Jure and Levent, Isabelle and Li, Xiang Lisa and Li, Xuechen`
`↪ and Ma, Tengyu and Malik, Ali and Manning, Christopher D. and Mirchandani, Suvir and`
`↪ Mitchell, Eric and Munyikwa, Zanele and Nair, Suraj and Narayan, Avani and Narayanan,`
`↪ Deepak and Newman, Ben and Nie, Allen and Niebles, Juan Carlos and Nilforoshan, Hamed and`
`↪ Nyarko, Julian and Ogut, Giray and Orr, Laurel and Papadimitriou, Isabel and Park, Joon`
`↪ Sung and Piech, Chris and Portelance, Eva and Potts, Christopher and Raghunathan, Aditi`
`↪ and Reich, Rob and Ren, Hongyu and Rong, Frieda and Roohani, Yusuf and Ruiz, Camilo and`
`↪ Ryan, Jack and Ré, Christopher and Sadigh, Dorsa and Sagawa, Shiori and Santhanam, Keshav`
`↪ and Shih, Andy and Srinivasan, Krishnan and Tamkin, Alex and Taori, Rohan and Thomas,`
`↪ Armin W. and Tramèr, Florian and Wang, Rose E. and Wang, William and Wu, Bohan and Wu,`
`↪ Jiajun and Wu, Yuhuai and Xie, Sang Michael and Yasunaga, Michihiro and You, Jiaxuan and`
`↪ Zaharia, Matei and Zhang, Michael and Zhang, Tianyi and Zhang, Xikun and Zhang, Yuhui and`
`↪ Zheng, Lucia and Zhou, Kaitlyn and Liang, Percy},`
`journal={arXiv preprint arXiv:2108.07258},`
`year={2021}`
`}`
57. `@article{radford2021clip,`
`title={Learning Transferable Visual Models From Natural Language Supervision},`
`author={Radford, Alec and Kim, Jong Wook and Hallacy, Chris and Ramesh, Aditya and Goh,`
`↪ Gabriel and Agarwal, Sandhini and Sastry, Girish and Askell, Amanda and Mishkin, Pamela`
`↪ and Clark, Jack and Krueger, Gretchen and Sutskever, Ilya},`
`journal={arXiv preprint arXiv:2103.00020},`
`year={2021}`
`}`
58. `@article{he2021mae,`
`title={Masked Autoencoders Are Scalable Vision Learners},`
`author={He, Kaiming and Chen, Xinlei and Xie, Saining and Li, Yanghao and Dollár, Piotr and`
`↪ Girshick, Ross},`
`journal={arXiv preprint arXiv:2111.06377},`
`year={2021}`
`}`
59. `@article{oord2018cpc,`
`title={Representation Learning with Contrastive Predictive Coding},`
`author={Oord, Aaron van den and Li, Yazhe and Vinyals, Oriol},`
`journal={arXiv preprint arXiv:1807.03748},`
`year={2018}`
`}`

60. `@article{chen2020simclr,`
`title={A Simple Framework for Contrastive Learning of Visual Representations},`
`author={Chen, Ting and Kornblith, Simon and Norouzi, Mohammad and Hinton, Geoffrey},`
`journal={arXiv preprint arXiv:2002.05709},`
`year={2020}`
`}`
61. `@article{lecun2022path,`
`title={A Path Towards Autonomous Machine Intelligence},`
`author={LeCun, Yann},`
`journal={OpenReview preprint},`
`year={2022}`
`}`
62. `@article{assran2023ijepa,`
`title={Self-Supervised Learning from Images with a Joint-Embedding Predictive Architecture},`
`author={Assran, Mahmoud and Duval, Quentin and Misra, Ishan and Bojanowski, Piotr and Vincent,`
`↪ Pascal and Rabbat, Michael and LeCun, Yann and Ballas, Nicolas},`
`journal={arXiv preprint arXiv:2301.08243},`
`year={2023}`
`}`
63. `@article{bardes2024vjepa,`
`title={Revisiting Feature Prediction for Learning Visual Representations from Video},`
`author={Bardes, Adrien and Garrido, Quentin and Ponce, Jean and Chen, Xinlei and Rabbat,`
`↪ Michael and LeCun, Yann and Assran, Mahmoud and Ballas, Nicolas},`
`journal={arXiv preprint arXiv:2404.08471},`
`year={2024}`
`}`
64. `@article{balestrieri2025lejepa,`
`title={LeJEPA: Provable and Scalable Self-Supervised Learning Without the Heuristics},`
`author={Balestrieri, Randall and LeCun, Yann},`
`journal={arXiv preprint arXiv:2511.08544},`
`year={2025}`
`}`
65. `@article{balestrieri2025lejepa-formulation,`
`title={LeJEPA: Provable and Scalable Self-Supervised Learning Without the Heuristics},`
`author={Balestrieri, Randall and LeCun, Yann},`
`journal={arXiv preprint arXiv:2511.08544},`
`year={2025}`
`}`
66. `@article{assran2025vjepa2,`
`title={V-JEPA 2: Self-Supervised Video Models Enable Understanding, Prediction and Planning},`
`author={Assran, Mido and Bardes, Adrien and Fan, David and Garrido, Quentin and Howes, Russell`
`↪ and Komeili, Mojtaba and Muckley, Matthew and Rizvi, Ammar and Roberts, Claire and Sinha,`
`↪ Koustuv and Zholus, Artem and Arnaud, Sergio and Gejji, Abha and Martin, Ada and Hogan,`
`↪ Francois Robert and Dugas, Daniel and Bojanowski, Piotr and Khalidov, Vasil and Labatut,`
`↪ Patrick and Massa, Francisco and Szafraniec, Marc and Krishnakumar, Kapil and Li, Yong and`
`↪ Ma, Xiaodong and Chandar, Sarath and Meier, Franziska and LeCun, Yann and Rabbat, Michael`
`↪ and Ballas, Nicolas},`
`journal={arXiv preprint arXiv:2506.09985},`
`year={2025}`
`}`

67. `@article{hu2021lora,`
`title={LoRA: Low-Rank Adaptation of Large Language Models},`
`author={Hu, Edward J. and Shen, Yelong and Wallis, Phillip and Allen-Zhu, Zeyuan and Li,`
`↪ Yuanzhi and Wang, Shean and Wang, Lu and Chen, Weizhu},`
`journal={arXiv preprint arXiv:2106.09685},`
`year={2021}`
`}`
68. `@article{ouyang2022instructgpt,`
`title={Training language models to follow instructions with human feedback},`
`author={Ouyang, Long and Wu, Jeff and Jiang, Xu and Almeida, Diogo and Wainwright, Carroll L.`
`↪ and Mishkin, Pamela and Zhang, Chong and Agarwal, Sandhini and Slama, Katarina and Ray,`
`↪ Alex and Schulman, John and Hilton, Jacob and Kelton, Fraser and Miller, Luke and Simens,`
`↪ Maddie and Askell, Amanda and Welinder, Peter and Christiano, Paul and Leike, Jan and Lowe,`
`↪ Ryan},`
`journal={arXiv preprint arXiv:2203.02155},`
`year={2022}`
`}`
69. `@article{rafaellov2023dpo,`
`title={Direct Preference Optimization: Your Language Model is Secretly a Reward Model},`
`author={Rafailov, Rafael and Sharma, Archit and Mitchell, Eric and Ermon, Stefano and Manning,`
`↪ Christopher D. and Finn, Chelsea},`
`journal={arXiv preprint arXiv:2305.18290},`
`year={2023}`
`}`
70. `@article{hinton2015distilling,`
`title={Distilling the Knowledge in a Neural Network},`
`author={Hinton, Geoffrey and Vinyals, Oriol and Dean, Jeff},`
`journal={arXiv preprint arXiv:1503.02531},`
`year={2015}`
`}`
71. `@article{frantar2022gptq,`
`title={GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers},`
`author={Frantar, Elias and Ashkboos, Saleh and Hoefler, Torsten and Alistarh, Dan},`
`journal={arXiv preprint arXiv:2210.17323},`
`year={2022}`
`}`
72. `@article{jacob2017quantization,`
`title={Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only`
`↪ Inference},`
`author={Jacob, Benoit and Kligys, Skirmantas and Chen, Bo and Zhu, Menglong and Tang, Matthew`
`↪ and Howard, Andrew and Adam, Hartwig and Kalenichenko, Dmitry},`
`journal={arXiv preprint arXiv:1712.05877},`
`year={2017}`
`}`
73. `@article{dettmers2023qlora,`
`title={QLoRA: Efficient Finetuning of Quantized LLMs},`
`author={Dettmers, Tim and Pagnoni, Artidoro and Holtzman, Ari and Zettlemoyer, Luke},`
`journal={arXiv preprint arXiv:2305.14314},`
`year={2023}`
`}`