

DEEP LEARNING NOTES



Deep Learning

Notes of Luigi Battista

A.Y. 2024/25

Preface

These notes accompany the **Deep Learning course (A.Y. 2024/25)**, taught by Prof. Vito Walter Anelli as part of the Master's program in Artificial Intelligence and Data Science at PoliBa – Polytechnic University of Bari.

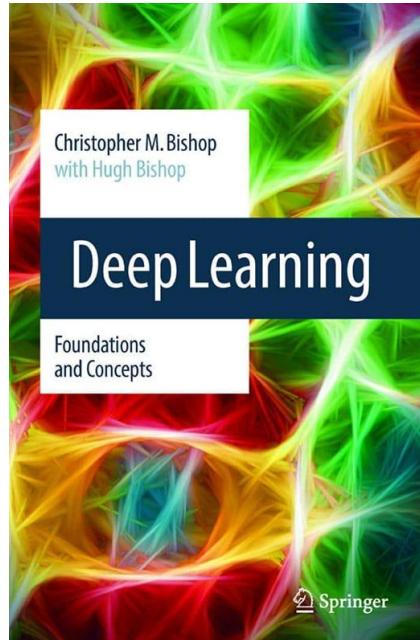
During the course, many Deep Learning concepts are introduced, but often in a somewhat **superficial** way: illustrated intuitively but without sufficient depth in the explanations. My goal in writing these notes is to provide a **more detailed and structured understanding**, explaining not just how things work, but **why** they work.

All topics covered in the course are included here, but I have added extensive additional explanations to fill gaps where the course presentation was less detailed. As a student myself, I initially struggled with some concepts, realizing that these difficulties stemmed from the limited depth of the course material. In some chapters, I have reorganized and rewritten the content entirely to improve clarity and completeness.

I strongly believe that a deep understanding of the **mathematical formulation** is essential. When something seems like magic in Deep Learning, there is almost certainly **mathematics behind it**. By exploring these foundations, we can ensure that our understanding is not superficial or alchemical, but precise and engineering-driven.

As in my notes for the [ML-Course](#), my objective here is to provide material that helps PoliBa students build a solid understanding of Deep Learning, enabling them to actively contribute to AI development rather than merely follow its evolution without the ability to design new methods and architectures.

My primary reference is [Deep Learning - Foundations and Concepts](#) by Chris Bishop (and his son), published in 2024. This book is, in my opinion, one of the best for combining intuitive explanations with correct mathematical formalism. Many of the concepts in these notes are drawn from it.



Other useful references include:

- [Understanding Deep Learning](#) by Simon J.D. Prince (University of Bath)
- [Dive into Deep Learning](#) by Aston Zhang, Zachary C. Lipton, Mu Li and Alexander J. Smola ([Amazon](#))

- **Research papers** cited throughout these notes. In practice, as a Machine Learning engineer, you will frequently need to consult and understand papers to keep up with new methods, improvements, and state-of-the-art techniques. **For this reason, I highly encourage you to read these papers alongside these notes**, as doing so will deepen your understanding, reinforce the concepts presented here, and help you develop the habit of engaging with current research.

I have also created a repository on my personal GitHub called [DL-Course](#), where the **most up-to-date version** of these notes can be found. In the future, I plan to add code examples directly linked to the theoretical concepts discussed here. If you find the repository helpful, I would greatly appreciate it if you could give it a  or share it with others.

The most important factor for success in Deep Learning is **truly understanding the theoretical concepts**. Without this, one risks becoming more of an alchemist than a Machine Learning engineer. While coding and experimentation are crucial, they will be far more meaningful when grounded in solid theory.

Finally, I want to emphasize that these notes were created not merely to pass the exam (which requires much less effort), but to genuinely understand the foundations of the subject. The time you invest in studying deeply now will save you time and effort in the future. Moreover, a solid understanding makes passing the exam significantly easier. With that said, let's embark on the study of this fascinating subject.

Contents

Preface	3
Introduction to Deep Learning	13
Computational Graphs in Deep Learning.....	15
ML Model in Supervised Learning.....	16
Loss Function.....	17
Model Optimization with Gradient Descent	18
Traditional Neural Network.....	21
Automatic Differentiation.....	25
Pytorch	26
Learning Representations.....	29
Linear Classifiers and their Limitations	29
Shallow Networks and the Need for Deep Architectures.....	30
Invariant Feature Learning	32
The Manifold Hypothesis.....	32
Disentangling Factors of Variation	33
Learning Hierarchical Representations.....	34
Activation Functions	35
Threshold - nn.Threshold()	36
Sigmoid - nn.Sigmoid().....	36
Vanishing and Exploding Gradients	36
Tanh - nn.Tanh()	37
ReLU- nn.ReLU()	38
LeakyReLU - nn.LeakyReLU()	39
PReLU - nn.PReLU().....	39
RReLU - nn.RReLU()	39
ELU - nn.ELU()	40
GELU - nn.GELU().....	41
SiLU – nn.SiLU	41
Softmax - nn.Softmax()	42
Softmin - nn.Softmin()	42
nn.LogSoftmax()	43
Saturating functions vs non-saturating functions	43
Cost Functions	44
nn.MSELoss()	44
nn.L1Loss()	44

nn.SmoothL1Loss()	45
L1 vs. L2 for Computer Vision	45
nn.NLLLoss()	46
Weights & Handling Imbalanced Classes	47
nn.CrossEntropyLoss()	47
nn.BCELoss()	49
nn.BCEWithLogitsLoss()	49
nn.KLDivLoss()	50
Hinge Loss	50
nn.HingeEmbeddingLoss()	50
nn.SoftMarginLoss()	51
nn.MultiLabelMarginLoss()	51
nn.TripletMarginLoss()	51
nn.MarginRankingLoss()	52
nn.CosineEmbeddingLoss()	53
Optimization	55
The Error Surface	55
Efficient Gradient Computation: Mini-Batch Learning	56
A bag of tricks for mini-batch gradient descent	57
1. Data Normalization	57
2. Initializing the Weights	58
3. Decorrelate the Input Components	60
4. Be Careful about Turning Down the Learning Rate	61
Challenges in Neural Network Optimization	61
Momentum	63
The Equations of the Momentum	63
Role of β	64
Nesterov Momentum	64
Why Does Momentum Work?	66
Separate Adaptive Learning Rates	67
Separate Adaptive Learning Rates based on Gradient Sign	67
Rprop: Using Only the Sign of the Gradient	68
Why Rprop Does Not Work Well with Mini-Batches?	69
RMSProp: A mini-batch version of Rprop	69
Adagrad	70
Adadelta	71
RMSProp Analogies	72

Adam	72
A closer look to Adam	73
Limitations of Adam.....	75
AdamW	75
Lion	76
A Brief Overview of "Hessian-Free" Optimization.....	77
Intuition Behind Curvature	77
Newton's method	78
Curvature (Hessian) Matrix.....	78
How to avoid Inverting a Huge Matrix.....	79
Conjugate Gradient	80
Normalization Layers	82
Batch Normalization	82
Does Scale and Shift Undo Normalization?	83
BatchNorm During Inference	83
Why Is Batch Normalization Effective?	83
Limitations of BatchNorm	84
Layer Normalization.....	84
Key Differences from BatchNorm	85
LayerNorm in CNNs.....	85
Instance Normalization.....	86
Group Normalization	86
Overview of Ways to Improve Generalization.....	88
Overfitting	88
Preventing Overfitting	88
Limit the Capacity of a Neural Net	88
Weight Decay	89
Early Stopping	89
Using Noise as a Regularizer	90
Ensemble of Models	90
How Does the Ensemble Compare with the Individual Predictors?.....	91
How to Encourage Diversity Among Predictors	91
Dropout	92
What Happens at Test Time?	93
Another way to think about Dropout.....	94
Convolutional Neural Networks	95
Image Data: What Makes Images Different?	95

Filters as Feature Detectors.....	98
Convolution and Cross-Correlation	100
Padding.....	102
Strided Convolutions	103
Convolutional Layer.....	105
1x1 Convolutional Layer.....	107
Local Connectivity and Parameters Sharing.....	108
More about Receptive Fields	108
Pooling Layer.....	110
FC Layer.....	111
Typical CNN Architecture	111
What Do Convolutional Filters Learn?	112
LeNet, AlexNet & VGG-16	114
ResNet.....	116
Other Computer Vision Tasks	120
Object Detection	121
Image Segmentation.....	124
Fully Convolutional Networks (FCNs).....	126
U-Net.....	128
Transfer Learning	130
Data Augmentation.....	131
Recurrent Neural Networks	132
Text Data: What Makes Text Different?	132
Word Embedding.....	133
Tokenization	134
How to Handle Sequential Data?	135
Recurrent Neuron	138
Backpropagation Through Time (BPTT)	139
The Problem of Long-Term Dependencies	141
# Idea: Using Gated Memory Cells	143
LSTM.....	143
LSTM Gradient Flow	146
GRU.....	147
RNN Applications	148
Bi-LSTM.....	151
Bi-LSTM with Attention (Att-BLSTM)	152
RNNs Limitations.....	153

Transformers	155
Attention	156
Intuition behind Attention	156
Attention Weights	157
Attention Mechanism	158
Self-Attention.....	159
Multi-head Attention.....	161
Encoder	162
Positional Encoding	164
Classification Head	168
Decoder.....	169
Encoder-Decoder Transformer	170
Training an Encoder-Decoder Transformer.....	172
GPT.....	173
BERT	175
Pre-Training.....	176
Finetuning.....	177
BERT for Features Extraction	178
GPT-2.....	179
GPT-3.....	181
Strategies for Next Token Sampling	182
ChatGPT	186
Large Language Models.....	188
Mixture of Experts.....	189
Retrieval-Augmented Generation.....	190
Chain of Thought	192
Vision Transformer.....	194
CLIP.....	196
Recent Advances in Transformers	199
KV Cache, Multi-Query and Grouped-Query Attention.....	199
Advancing Positional Embeddings	205
Multi-Head Latent Attention	209
Graph Neural Networks	213
What is a Graph?	213
Images as Graphs.....	214
Text as Graphs	215
Molecules as Graphs	215

Social Networks as Graphs	216
Graphs of Different Sizes.....	216
Graph-level Tasks, Node-level Tasks & Edge-level Tasks	217
The Challenges of Computation on Graphs	218
Node-Order Equivariance	220
Scalability & Sparsity	220
Building a Graph Neural Network.....	221
The Simplest GNN	221
Message Passing	222
GNN Predictions Using Information Aggregation.....	223
Learning Edge Representations	225
Adding Global Representations	226
Convolutional Graph Neural Network.....	227
Polynomial Filters on Graphs.....	228
ChebNet.....	233
Embedding Computation	234
The Shared Workings behind GNNs	234
Modern Graph Neural Networks.....	235
GCN	235
GAT.....	237
GraphSAGE.....	238
GIN.....	239
Autoencoders.....	242
Standard Autoencoder.....	242
Sparse Autoencoders.....	243
Denoising Autoencoders	243
Deep Autoencoder.....	244
Deep Autoencoder vs PCA	245
Generative Adversarial Networks	247
Generative Models.....	247
The Idea behind GAN – Indirect Approach.....	248
Architecture of a GAN & Adversarial Training.....	251
Mathematics behind GANs.....	252
Difficulties with Training GANs	254
GANs in Image Domain	256
DCGAN	256
Conditional GAN	257

CycleGAN	258
Other GANs.....	260
Variational Autoencoder.....	261
Motivation for VAEs: The Limitations of Standard Autoencoders	261
Latent Variables Models	264
Non-linear Latent Variable Model	267
VAE's Training	268
ELBO	268
Variational Inference.....	270
The Variational Autoencoder	272
Re-parametrization Trick	273
Comparison of VAE Loss and Standard Autoencoder Loss.....	274
Final Considerations and Applications of VAEs	276
Normalizing Flows	280
Introducing Normalizing Flows	280
Coupling Flows.....	282
Autoregressive Flows	285
Continuous Flows.....	286
Neural Differential Equations	286
Neural ODE Flows	287
Energy-based Models.....	289
How an Energy-Based Model Works	289
Energy Function Surface after Training	290
EBMs Inference	291
Latent-Variable EBM	292
Regularized Latent-Variable EBM	293
How Do we Model EBMs?	294
Training EBMs	296
Contrastive Divergence	298
Other Contrastive Approaches	299
Contrastive Joint Embedding	299
Non-Contrastive Embedding	300
Diffusion Models.....	302
Intuition Behind Diffusion Models	302
Denoising Diffusion Probabilistic Model.....	305
Forward Diffusion Process	305
Diffusion Kernel.....	309

Reverse Diffusion Process	310
Training Loss	312
Inference	317
Guided Diffusion	318
Classifier Guidance	319
Classifier-Free Guidance	320
Stable Diffusion	322
Other Conditioning Modalities in Stable Diffusion	324
Stable Diffusion v1 vs v2	328
Diffusion Score Matching	330
Score Loss	331
Sampling from a Score-Based Model	333
Score Matching vs Denoising Diffusion	334
Noise Variance Schedule	334
Continuous-Time Diffusion Models	336
Flow Matching	342
Bibliography	343

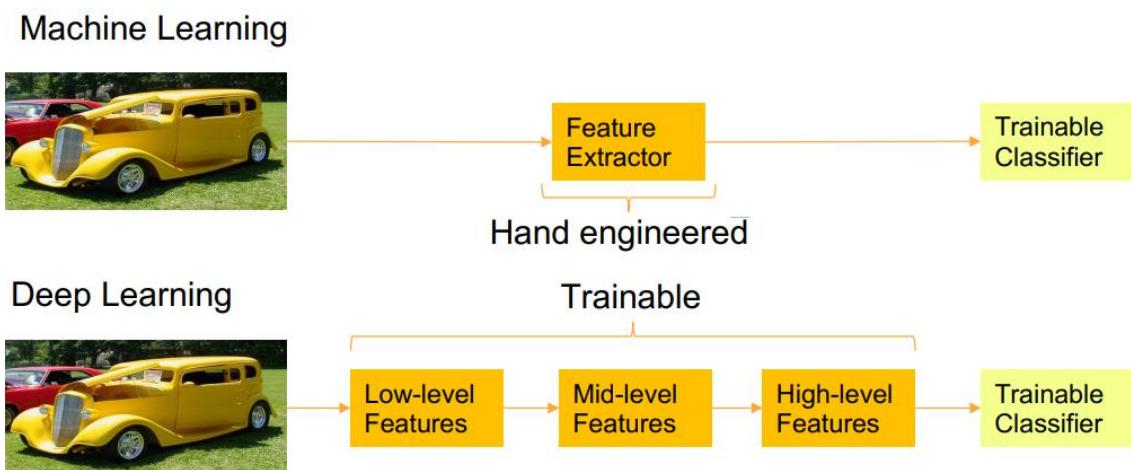
Introduction to Deep Learning

One of the greatest challenges in **Artificial Intelligence (AI)** has always been solving tasks that are easy for humans to perform, but difficult to formally describe using a set of mathematical rules. These tasks, which we often solve intuitively and effortlessly — such as recognizing spoken words or identifying faces in images — pose significant challenges for traditional computational approaches.

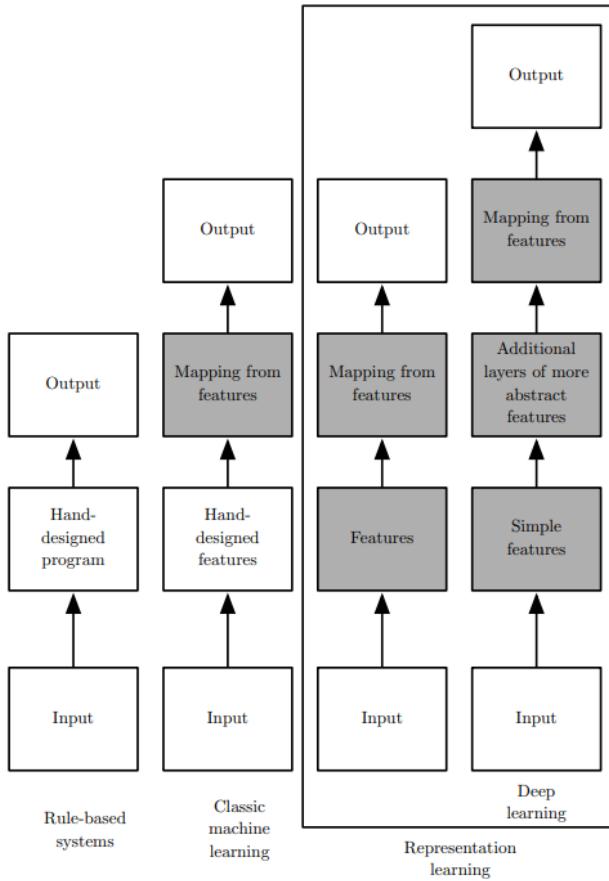
A promising solution is to enable computers to learn from experience and understand the world through a hierarchy of concepts, where each concept is defined in relation to simpler ones. By acquiring knowledge through experience, this approach eliminates the need for human operators to explicitly encode all the necessary knowledge into the system.

This hierarchical organization allows a computer to learn complex concepts by building them upon simpler ones. If we visualize this structure as a graph, where each layer represents an increasingly abstract level of understanding, we obtain a deep architecture. This depth is what characterizes **Deep Learning (DL)**, a subfield of *Machine Learning (ML)* designed to leverage these layered representations.

From Feature Engineering to Representation Learning: In traditional machine learning, many tasks can be solved by carefully designing and extracting the right set of features (*feature engineering*) and then feeding them into a relatively simple machine learning algorithm. However, for many problems it is difficult to determine which features should be extracted. For example, consider the task of detecting cars in images. We know that cars have wheels, so we might attempt to use the presence of a wheel as a feature. However, defining precisely how a wheel appears in terms of pixel values is challenging. While a wheel has a simple geometric structure, its visual representation can vary due to factors such as shadows, reflections on metallic surfaces, occlusions from fenders or other objects in the foreground, and more.



A more effective approach is to use ML not only to learn the mapping from input data to output but also to learn the representation itself. This means that the feature extraction process becomes *trainable* as well (rather than manually designed). This approach is known as **representation learning**.



Learned representations often yield significantly better performance than handcrafted ones. Moreover, they allow AI systems to quickly adapt to new tasks with minimal human intervention. A representation learning algorithm can discover a suitable set of features for a simple task in minutes or, for a more complex task, in hours or even months. In contrast, manually designing feature representations for complex tasks requires extensive time and human effort — sometimes decades for an entire research community.

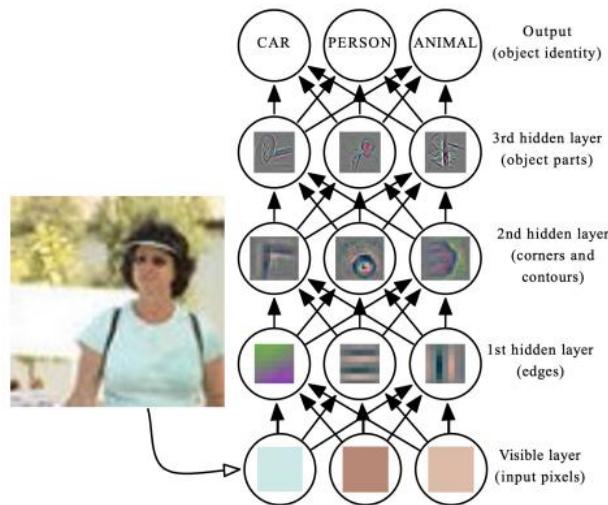
Deep learning addresses this fundamental challenge in representation learning by building **hierarchy representations** where more complex features are expressed as combination of other, simpler features. Each deep learning model should first be able to extract **low-level features**, then combine them into **mid-level features** and finally integrate these into **high-level features** that capture more abstract concepts. For example, a deep learning system processing an image might start by detecting basic visual patterns, then progressively build up to recognizing larger structures, and ultimately identifying a complete object.

8 9 0 1 2 3 4 7 8 9 0 1 2 3 4 5 6 7 8 6
4 2 6 4 7 5 5 4 7 8 9 2 9 3 9 3 8 2 0 5
0 1 0 4 2 6 5 3 5 3 8 0 0 3 4 1 5 3 0 8
3 0 6 2 7 1 1 8 1 7 1 3 8 9 7 6 7 4 1 6
7 5 1 7 1 9 8 0 6 9 4 9 9 3 7 1 9 2 2 5
3 7 8 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 0
1 2 3 4 5 6 7 8 9 8 1 0 5 5 1 1 0 4 1 9
3 8 4 7 7 8 5 0 6 5 5 3 3 3 3 9 8 1 4 0 6
1 0 0 6 2 1 1 3 2 8 0 7 8 4 6 0 2 0 3 6
8 7 1 5 9 3 2 4 9 4 6 5 3 2 8 5 9 4 6
6 5 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
8 9 0 1 2 3 4 5 6 7 8 9 6 4 2 6 4 7 5 5
4 7 8 9 2 9 3 9 3 8 2 0 9 8 0 5 6 0 1 0
4 2 6 5 5 4 3 4 1 5 3 0 8 3 0 6 2 7 1
1 8 1 7 1 3 8 5 4 2 0 9 7 6 7 4 1 6 8 4
7 5 1 2 6 7 1 9 8 0 6 9 4 9 9 6 2 3 7 1
9 2 2 5 3 7 8 0 1 2 3 4 5 6 7 1 0 1 2 3
4 5 6 7 8 0 1 2 3 4 5 6 7 8 9 2 1 2 1 3
9 9 8 5 3 7 0 7 7 5 7 9 9 4 7 0 3 4 1 4
4 7 5 8 1 4 8 4 1 8 6 6 4 6 3 5 7 2 5 9

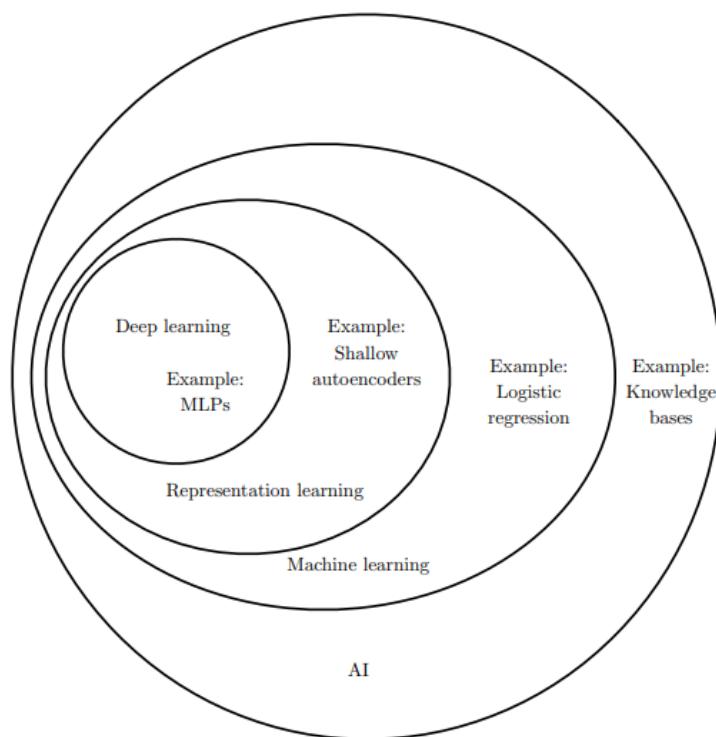
Example: The MNIST Dataset

The **MNIST dataset** is a large collection of handwritten digit images, each labeled with the corresponding number (0–9). This simple classification problem is one of the most widely used benchmarks in deep learning research. Despite being relatively easy to solve with modern techniques, MNIST remains popular as a fundamental test for new models.

Therefore, deep learning enables computers to construct complex concepts by building upon simpler ones. The Figure below illustrates how a deep learning system can represent the concept of an image of a person by combining simpler concepts, such as corners and contours, which are in turn defined in terms of edges.



Below is a Venn diagram that highlights how deep learning is a kind of representation learning, which is a kind of machine learning, which is used for many but not all approaches to AI. Each section of the Venn diagram includes an example of an AI technology.

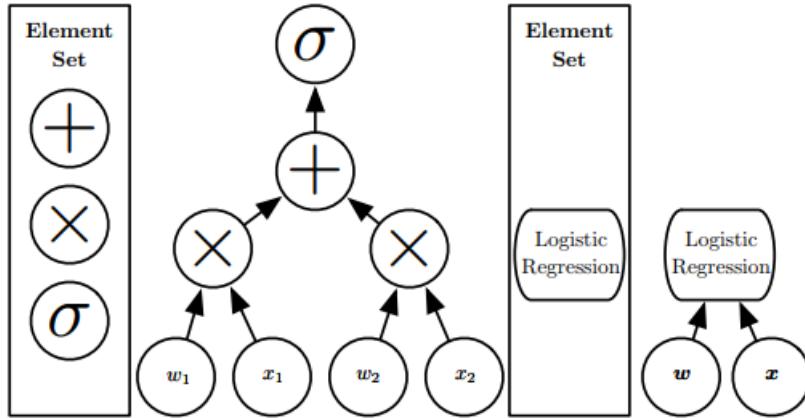


Computational Graphs in Deep Learning

One of the most effective ways to analyze a deep learning model is through **computational graphs**. A computational graph is a **directed acyclic graph (DAG)** where nodes represent operations or variables, and directed edges define the flow of data and dependencies between computations.

- An **operation** is a function applied to one or more variables.
- Our computational graph framework is accompanied by a set of allowed operations.
- More complex functions are built by **composing** simpler operations.
- If a variable y is computed by applying an operation to a variable x , we draw a directed edge from x to y .

For example, the computational graph for *logistic regression* is illustrated in the Figure below:



Depending on how we define the granularity of computational modules, we can represent the graph at different levels of abstraction. For instance, we might explicitly model individual operations such as **addition**, **multiplication** and the **sigmoid function** (Figure on the left), or we could treat **logistic regression as a single computational unit** (Figure on the right).

A key advantage of computational graphs is their role in **automatic differentiation**, which is crucial for training deep learning models. Frameworks like **PyTorch** leverage dynamic computational graphs to efficiently compute gradients during backpropagation. We will explore these details further in the following sections.

ML Model in Supervised Learning

A generic ML model in a **supervised learning** setting follows this process: it takes two inputs — the actual input x and the target value y . A **parameterized deterministic function** G is then applied to x , meaning that x is combined with the parameters θ through a deterministic function, such as a **linear combination** or another type of function. As output, this function produces the **predicted value** \bar{y} :

$$\bar{y} = G(x, \theta)$$

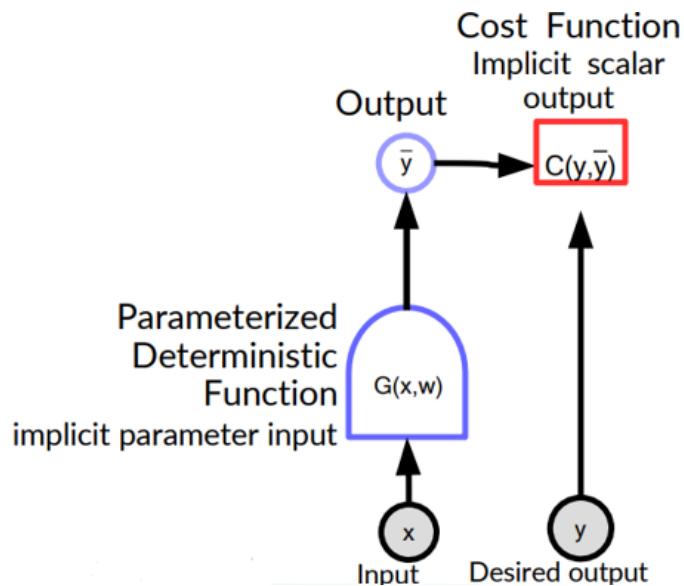
To measure how much the predicted value \bar{y} differs from the target value y , we apply another function called the **cost function** C . The cost function depends on the specific task and what we want to measure. Examples:

- **Linear Regression (MSE):**

$$\bar{y} = \sum_j \theta_j x_j \quad C(y, \bar{y}) = \|y - \bar{y}\|^2$$

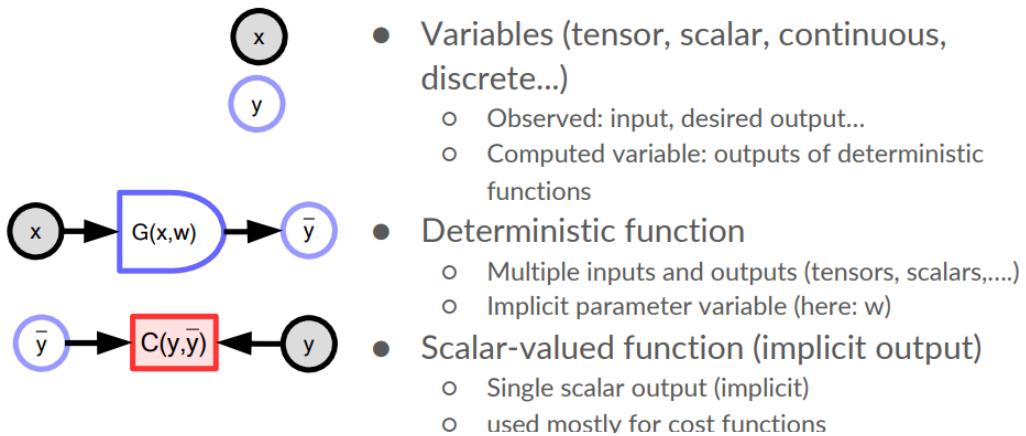
- **Logistic Regression (BCE):**

$$\bar{y} = \sigma\left(\sum_j \theta_j x_j\right) \quad C(y, \bar{y}) = -[y \log(\bar{y}) + (1 - y) \log(1 - \bar{y})]$$



Note: In the Figure above, we illustrate a single function representing the G block, but in general, this block can consist of **multiple functions** combined in different ways. Therefore, the computation of G may involve complex algorithms.

To represent our model using **computational graphs**, we introduce some fundamental notations:



Loss Function

Above we saw that to measure the discrepancy between \bar{y} and y we need to apply a cost function. If we want to avoid representing two different encapsulated functions, we can define a single function that takes x, y, θ as inputs. This is the **loss function** L , which is defined for a single sample as:

$$L(x, y, w) = C(y, G(x, \theta))$$

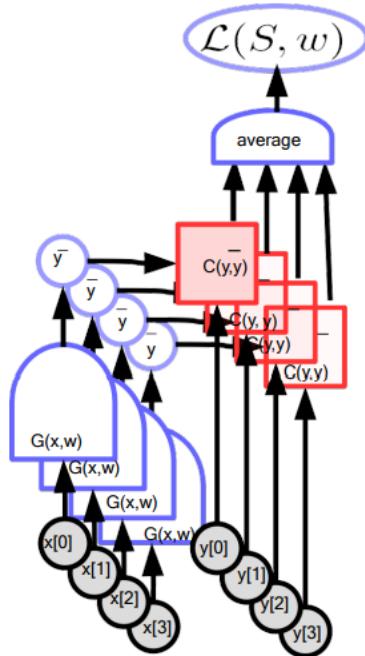
If we consider a **dataset S** , where each sample $x^{(i)}$ is associated with a label $y^{(i)}$:

$$S = \{(x^{(i)}, y^{(i)}) \mid i = 0, \dots, m - 1\}$$

we can compute the **average loss** over the entire dataset:

$$\mathcal{L}(S, \theta) = \frac{1}{m} \sum_{i=0}^m L(x^{(i)}, y^{(i)}, \theta)$$

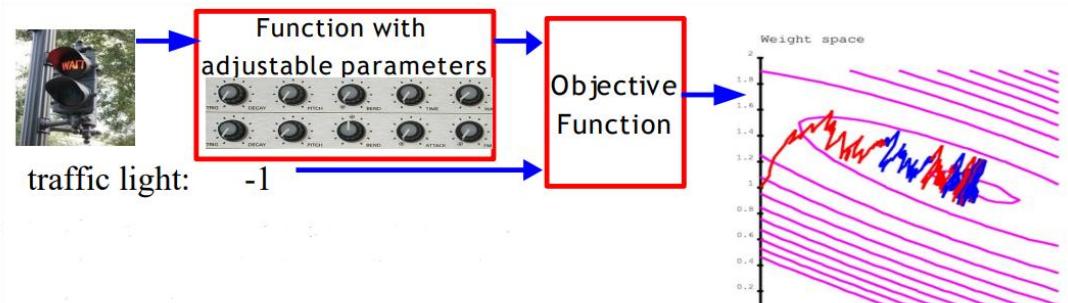
If we would represent the **average loss** in a computational graph, this would correspond to replicating the previously defined computational graph ***m times***, computing L for each sample, and then applying an **average block** that takes all these L values as inputs and computes the final average loss.



Model Optimization with Gradient Descent

Once we have defined the **model architecture** and the **loss function** to minimize, we can begin the **learning phase**. The goal of learning is to find the optimal parameters values that minimize the loss function.

Essentially, this process can be seen as solving an **optimization problem**, where the **objective function** is the **loss function**, and the **variables to optimize** are the parameters. What we want to do is adjust the parameters until we get better and better results.



A common approach to solving this optimization problem is with **Gradient Descent (GD)**. GD is an iterative optimization technique that helps adjust the parameters to find the minimum of the loss function. It does so by

following the **steepest descent direction** — the direction in which the loss decreases the fastest. This direction is given by the **gradient** $\nabla L(\theta)$, which is the vector of partial derivatives of the loss function:

$$\nabla L(\theta) = \begin{bmatrix} \frac{\partial L(\theta)}{\partial \theta_0} \\ \vdots \\ \frac{\partial L(\theta)}{\partial \theta_j} \\ \vdots \\ \frac{\partial L(\theta)}{\partial \theta_n} \end{bmatrix}$$

The core idea of **GD** is simple: at each iteration, we slightly adjust the parameters **in the direction indicated by the gradient**, gradually moving toward the minimum of the loss function.

GD tells us the most effective direction to update our parameters to minimize the loss. The gradient is the derivative vector that points in the direction of the **greatest increase** of the loss function — therefore, to minimize it, we move in the opposite direction (that's why we have the negative sign).

How does it work?

Initially, the parameters θ_j (for $j = 0, \dots, n$) are **randomly initialized** (or set using other techniques, as we will see in later chapters). Then, GD **iteratively updates** each parameter according to the **partial derivative** of the loss function with respect to that parameter:

$$\theta_j := \theta_j - \alpha \frac{\partial L(x, \theta)}{\partial \theta_j}$$

where α is the **learning rate**, a **hyperparameter** that controls the **step size** in each iteration. This process repeats until convergence.

GD is an optimization method that **only works locally**: at each step, it gives us a direction for how to increase or decrease the weights to reduce the loss, since we don't know at prior the shape of the loss surface.

The partial derivative $\frac{\partial L(x, \theta)}{\partial \theta_j}$ provides the **slope** of the loss function at the current point w.r.t to a given parameter θ_j , telling us how much the loss function changes when we slightly modify that parameter.

- If the derivative is **positive**, the loss function is increasing, so we **reduce** θ_j .
- If the derivative is **negative**, the loss function is decreasing, so we **increase** θ_j .

Intuition: We can think of GD as **descending a mountain in a thick fog**. In this metaphor, the fog represents our limited knowledge of the shape of the mountain (the shape of the loss function), while the GD is the process that allows us to descend into the valley.

Because the fog is so thick, we can only see the next step we can take in each possible direction. So, our strategy becomes to follow the direction of the **steepest descent** we can observe at that moment. This allows us to move in the direction that seems to lead to the bottom of the valley.

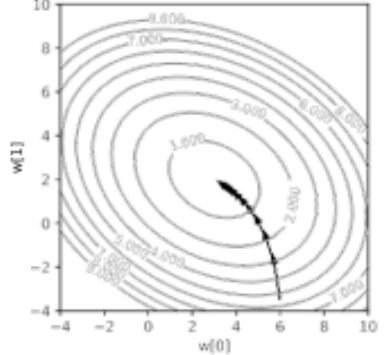
However, because our information is limited and subject to our reduced visibility (noise), the path we follow can be a bit random. Each sample we collect provides only a rough estimate of the direction, so, as we descend the mountain, our trajectory may have random deviations due to these uncertainties.

For the GD we have two main paradigms:

- **Full Batch Gradient Descent (FBGD):** At each iteration, FBDG computes the gradient using **all training samples** in the training set and then updates the parameters:

$$\theta := \theta - \alpha \frac{\partial \mathcal{L}(S, \theta)}{\partial \theta}$$

In FBDG, **convergence is ensured**. However, the computational cost of BGD per iteration **grows linearly** with the number of samples considered. Therefore, when dealing with large datasets, adopting FBDG would be too time-consuming, since we need to process all the data before performing a single update.



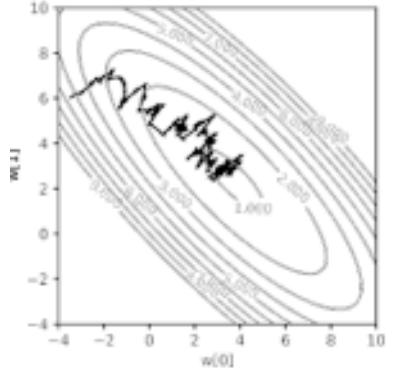
- **Stochastic Gradient Descent (SGD):** At each iteration we randomly **pick a single sample with replacement** from the training set to compute the gradient and update the parameters:

pick a i in $0, \dots, m - 1$, then update θ :

$$\theta := \theta - \alpha \frac{\partial \mathcal{L}(x^{(i)}, y^{(i)}, \theta)}{\partial \theta}$$

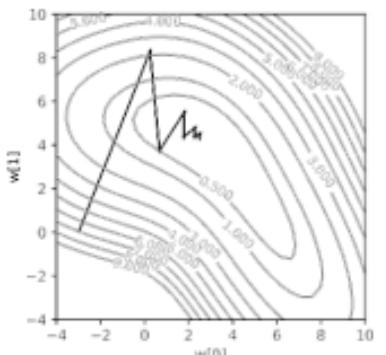
The gradient computed with SGD is a **stochastic approximation** of the full gradient. This means that, **on average**, the stochastic gradient is a good estimate of the gradient.

SGD significantly reduces the computational cost per iteration, making it much more efficient for large datasets. However, it introduces noise into the optimization process because the gradient is calculated using only one sample. This noise can cause the algorithm to oscillate around the minimum (a "zig-zag" effect), potentially preventing it from settling exactly at the global minimum.



To strike a balance between the stability of FBDG and the efficiency of SGD, practically we use a third option known as **Mini-Batch Gradient Descent (MBGD)**. Instead of computing the gradient using all training samples (as in FBDG) or a single sample (as in SGD), at each iteration MBGD considers a **small batch of randomly chosen training samples** to compute the gradient and update the parameters.

Using mini-batches reduces the variance in the updates compared to SGD, resulting in more stable convergence, while being computationally more efficient and faster than FBDG.



Mini-Batch GD significantly **smooths out the optimization path**, as shown in the Figure. It works very well if the samples chosen for each batch are representative of the overall dataset.

Traditional Neural Network

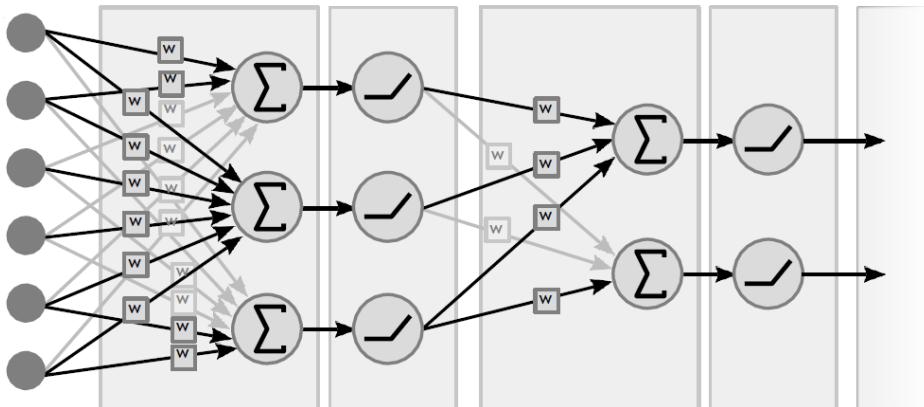
Below is a recap of how a standard **Neural Network** works, often also refers to as **Feed Forward Network (FFN)** or **Multilayer Perceptron (MLP)**, assuming you are already familiar with the basic concept. If not, I previously wrote a detailed chapter titled *Neural Networks* in my [ML Course notes](#), which you can refer to for a deeper explanation.

In a neural network, each unit (or neuron) receives a set of inputs and produces an output by computing a **weighted sum** of those inputs, followed by a **non-linear activation function**. One special input, x_0 (representing the intercept) is always set to 1 and is known as the **bias input**. The parameter associated with it, θ_0 , is called the **bias parameter b** , while the remaining parameters $\theta_1, \theta_2, \dots, \theta_n$ are known as **weights w** .

This distinction is not just semantic — it has practical importance. For example, as discussed in the ML course, regularization is typically applied only to the weights and not to the bias parameters. Another reason, which will become clearer in later sections, is that weights and biases are usually initialized differently at the start of training.

A **neural network** consists of multiple layers, each containing several **units (neurons)**. Between layers, we maintain **weight matrices W** , which define the mapping between one layer and the next, along with **bias vectors b** . Although bias vectors are not explicitly shown in the Figures below, they should always be taken into consideration (recall that at each unit bias is added to the weighted sum before applying the activation function).

The Figure below provides an example of a neural network.



During **forward propagation**, the input is passed through each layer of the network until the final output is obtained. Each unit in a layer receives input from the previous layer, computes the **weighted sum** of these inputs using the corresponding weight matrix (a matrix-vector product) and bias vector (added element-wise), and then applies a **non-linear activation function** (a point-wise product) such as ReLU, tanh, etc. (we will see them in detail in a next chapter). The result of this activation function becomes the output of the unit and serves as the input for the next layer.

Note: We use a non-linear activation function because it allows the network to handle data that are not linearly separable.

Mathematically, for a given unit i in layer l , the output is:

$$z_i^{[l]} = \sum_{j \in UP(i)} W_{ij}^{[l]} \cdot a_j^{[l-1]} + b_i^{[l]}$$
$$a_i^{[l]} = g(z_i^{[l]})$$

where:

- $z_i^{[l]}$ represents the weighted sum of the inputs for unit i in layer l and is generally called ***pre-activation***
- $a_i^{[l]}$ is the output after applying the *activation function* g and is generally called ***activation***

In a standard neural network, each unit is connected to every unit in the previous layer. Therefore, the term $UP(i)$ refers to the set of units in the previous layer that are actually connected to unit i of the layer l . When every unit in a layer receives input from *all* units in the preceding layer, this layer is called a **Fully Connected (FC) layer**.

Note: As we will see in later chapters, this is not always the case — sometimes a unit receives input from only a subset of units in the previous layer.

Once the forward pass is complete and the predicted output is obtained, we calculate how far it deviates from the target value using a cost function.

At this point, in order to update the network's parameters using gradient descent, we need to compute the gradients of the weigh matrix and bias vector of each layer. This is done using a procedure known as **backpropagation**, which systematically calculates these gradients by applying the chain rule.

The **chain rule** provides a way to compute the derivative of a composite function by relating the derivatives of its individual components. If we have a function composition $g((h(s)))$, the chain rule states that its derivative is:

$$g(h(s))' = g'(h(s)) h'(s)$$

In fact, the forward propagation can be seen as a sequence of function compositions, therefore during backpropagation, we compute the derivative of the cost function with respect to each parameter matrix/vector step by step, working **backward** from the output layer to the first layer. We do this by using the chain rule iteratively, leveraging previously computed partial derivatives to calculate those for earlier layers.

In matrix calculus ,depending on the layout convention adopted (that refers to how derivatives are arranged), the chain rule behaves differently:

- If we decide to follow the ***numerator layout*** we will have:

$$\frac{\partial g}{\partial s} = \frac{\partial g}{\partial h} \frac{\partial h}{\partial s}$$
- If we decide to follow the ***denominator layout*** we will have (practically we follow the reverse order):

$$\frac{\partial g}{\partial s} = \frac{\partial h}{\partial s} \frac{\partial g}{\partial h}$$

Pytorch and many other deep learning frameworks use the denominator layout, so here we decide to use it.

Note: In the slide of the course, instead is used the numerator layout.

In neural networks we will use the chain rule during backprop to compute the gradients of the parameters, i.e. the weight matrices and bias vectors in each layer. Therefore, we proceed as follows:

1. Starting from the last layer, we **first compute** $\frac{\partial C}{\partial z^{[l]}}$, which helps in computing the gradients of the parameters at that layer:

$$\frac{\partial C}{\partial z^{[l]}} = \frac{\partial z^{[l+1]}}{\partial z^{[l]}} \frac{\partial C}{\partial z^{[l+1]}}$$

2. Then, we **compute the gradients** $\frac{\partial C}{\partial W^{[l]}}$ and $\frac{\partial C}{\partial b^{[l]}}$, using the relationships:

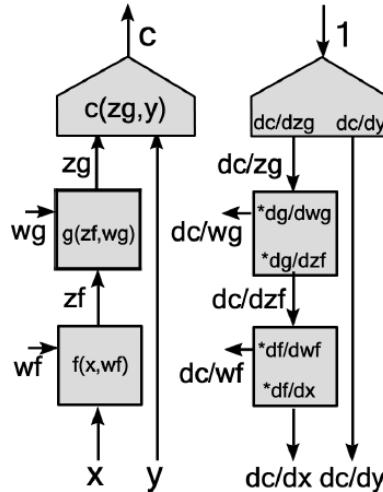
$$\frac{\partial C}{\partial W^{[l]}} = \frac{\partial z^{[l]}}{\partial W^{[l]}} \frac{\partial C}{\partial z^{[l]}}$$

$$\frac{\partial C}{\partial b^{[l]}} = \frac{\partial z^{[l]}}{\partial b^{[l]}} \frac{\partial C}{\partial z^{[l]}}$$

3. We repeat this process for each layer until we reach the first layer.

For instance, if we consider the neural network shown in the Figure, starting from the last layer we perform the backward pass calculating in this order the followings:

$$\begin{aligned} & \frac{\partial C}{\partial z_g} \\ & \frac{\partial C}{\partial W_g} = \frac{\partial z_g}{\partial W_g} \frac{\partial C}{\partial z_g} \quad \frac{\partial C}{\partial b_g} = \frac{\partial z_g}{\partial b_g} \frac{\partial C}{\partial z_g} \\ & \frac{\partial C}{\partial z_f} = \frac{\partial z_g}{\partial z_f} \frac{\partial C}{\partial z_g} \\ & \frac{\partial C}{\partial W_f} = \frac{\partial z_f}{\partial W_f} \frac{\partial C}{\partial z_f} \quad \frac{\partial C}{\partial b_f} = \frac{\partial z_f}{\partial b_f} \frac{\partial C}{\partial z_f} \end{aligned}$$



In the ML course, we denoted the generic $\frac{\partial C}{\partial z}$ with δ , representing the **error** of the units in a given layer and we saw that this error can be recursively computed from the error in the next layer. Each δ value is derived using the chain rule, but here we present them directly as $\frac{\partial C}{\partial z}$ because we had found explicit forms for these derivatives. In particular using the **denominator layout** we have found that:

- For the last layer L :

$$\delta^{[L]} = \frac{\partial C}{\partial z^{[L]}} = \frac{\partial C}{\partial a^{[L]}} \odot g'(z^{[L]})$$

- For any intermediate layer l :

$$\delta^{[l]} = \frac{\partial C}{\partial z^{[l]}} = \left((W^{[l+1]})^T \delta^{[l+1]} \right) \odot g'(z^{[l]})$$

- Once these were calculated it was then very easy to calculate the gradients of the parameters for each generic layer l :

$$\frac{\partial C}{\partial W^{[l]}} = \delta^{[l]} (a^{[l-1]})^T$$

$$\frac{\partial C}{\partial b^{[l]}} = \delta^{[l]}$$

Also, remember that when working with scalars, vectors and matrices it is crucial to ensure that **dimensions align correctly**. For example, considering the network above, let's compute $\frac{\partial C}{\partial z_f}$ assuming the following dimensions:

$$\frac{\partial C}{\partial z_f} = \frac{\partial z_g}{\partial z_f} \frac{\partial C}{\partial z_g}$$

$$[d_f \times 1] = [d_f \times d_g][d_g \times 1]$$

where d_f indicates the dimensionality (number of units) of layer f and d_g indicates the dimensionality (number of units) of layer g .

Therefore:

- z_g is a row vector, z_f is a row vector
- $\frac{\partial z_g}{\partial z_f}$ is a matrix of partial derivatives (Jacobian matrix):

$$\begin{bmatrix} (\partial z_g)_1 & (\partial z_g)_2 & \dots & (\partial z_g)_{d_g} \\ \hline (\partial z_f)_1 & (\partial z_f)_2 & \dots & (\partial z_f)_{d_f} \\ (\partial z_g)_1 & (\partial z_g)_2 & \dots & (\partial z_g)_{d_g} \\ \hline (\partial z_f)_2 & (\partial z_f)_2 & \dots & (\partial z_f)_{d_f} \\ \vdots & \vdots & \ddots & \dots \\ (\partial z_g)_1 & (\partial z_g)_2 & \dots & (\partial z_g)_{d_g} \\ \hline (\partial z_f)_{d_f} & (\partial z_f)_{d_f} & \dots & (\partial z_f)_{d_f} \end{bmatrix}$$

Each element of the Jacobian matrix represents a partial derivative of the i -th output with respect to the j -th input.

$$\left(\frac{\partial z_g}{\partial z_f} \right)_{ij} = \frac{(\partial z_g)_i}{(\partial z_f)_j}$$

- C is a scalar vector, z_g is a row vector
- $\frac{\partial C}{\partial z_g}$ is still a row vector

Multiplying the Jacobian matrix $\frac{\partial z_g}{\partial z_f}$ by the row vector $\frac{\partial C}{\partial z_g}$ we obtain a vector of dimensionality $[d_f \times 1]$, which is actually a one-dimensional Jacobian matrix.

Automatic Differentiation

At this point a natural question should arise: “If we already know the explicit forms of the derivatives to be taken into account then why do we express them using the chain rule? Could we simply use such expressions directly?”

The answer is that it would certainly make more sense to use such forms when dealing with simple neural networks such as the one we analyzed until here. However, what if we start to consider more complex neural networks (as we will do during this course) that introduce new different operations? Someone could say that a researcher somewhere maybe already did the calculus for us and computed them explicitly, but what if you want to implement a new architecture? In such cases you should find the explicit forms for these derivatives by hand, and this can be time-consuming and more importantly error prone. This is a big limitation on how quickly and effectively different architectures can be explored empirically.

To address these challenges, several alternatives were explored during the year including Numerical Differentiation or Symbolic Differentiation. Nowadays most of the deep learning frameworks including PyTorch make use of **Automatic Differentiation**.

Note: If you are interested in a deeper understanding of numerical and symbolic differentiation, I suggest you read the section 8.2 Automatic Differentiation of the recommended text book or also the following article.

Automatic Differentiation (autograd) is a technique used to automatically compute gradients given a computational graph. In essence, autograd dynamically builds a computational graph during runtime, recording all operations as they are executed. When performing forward propagation, the system not only computes the output but also tracks the derivative of each operation in the graph. This ensures that, during the backward pass, the derivatives of each operation are readily available. The explicit derivatives of basic functions are pre-defined and can be found in the [derivatives.yaml](#) file.

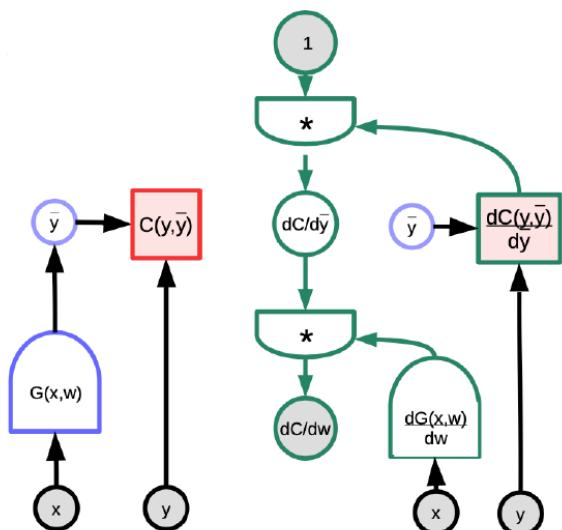
Each tensor stores a reference to its backward propagation function in the **grad_fn** property. This enables the computation of gradients by backpropagating through the computation graph, starting from the output. The system applies the chain rule to combine the derivatives of simpler operations recorded as it traverses the graph in reverse (**reverse-mode autodiff**). The gradients are only computed for the nodes in the graph where the **requires_grad** property is set to **True**. In the context of neural networks, this property is typically set for the parameters W and b , as we need to compute the gradients with respect to them.

For further information on automatic differentiation and how PyTorch’s autograd engine works, I recommend checking out the following resources: [Autograd tutorial](#) and [Overview of PyTorch Autograd Engine](#).

In the Figure on side is show how the computational graph built during forward prop (on the left) and its reverse computational graph built during back prop (on the right) look like.

Notes:

- From the Figure you can see that, at each step, we systematically apply the chain rule — multiplying the results of these backward operations — until we have computed all necessary gradients.
- In reality, instead of explicitly writing derivatives in mathematical notation, PyTorch represents them using internal functions like `<AddBackward>`, `<MulBackward>`, etc.



- We start with "1" at the root of the reverse computational graph because, by definition, the gradient of the cost function with respect to itself is always 1. Since the chain rule requires that we multiply derivatives, by initializing the root as "1," we ensure that the chain rule can be applied directly.

Pytorch

[PyTorch](#) is a powerful deep learning framework that simplifies the process of building complex neural networks. Thanks to its dynamic computation graph and automatic differentiation (autograd), PyTorch automatically handles gradient computations required for backpropagation, making model training more efficient and user-friendly.

Below is an example of how a simple neural network is implemented in PyTorch.

```
import torch
from torch import nn

# Initialize a random input tensor representing an image with 3 channels and spatial dimensions 10x20
image = torch.randn(3, 10, 20)
# Compute the total number of elements in the input tensor (flattened size)
d0 = image.nelement()

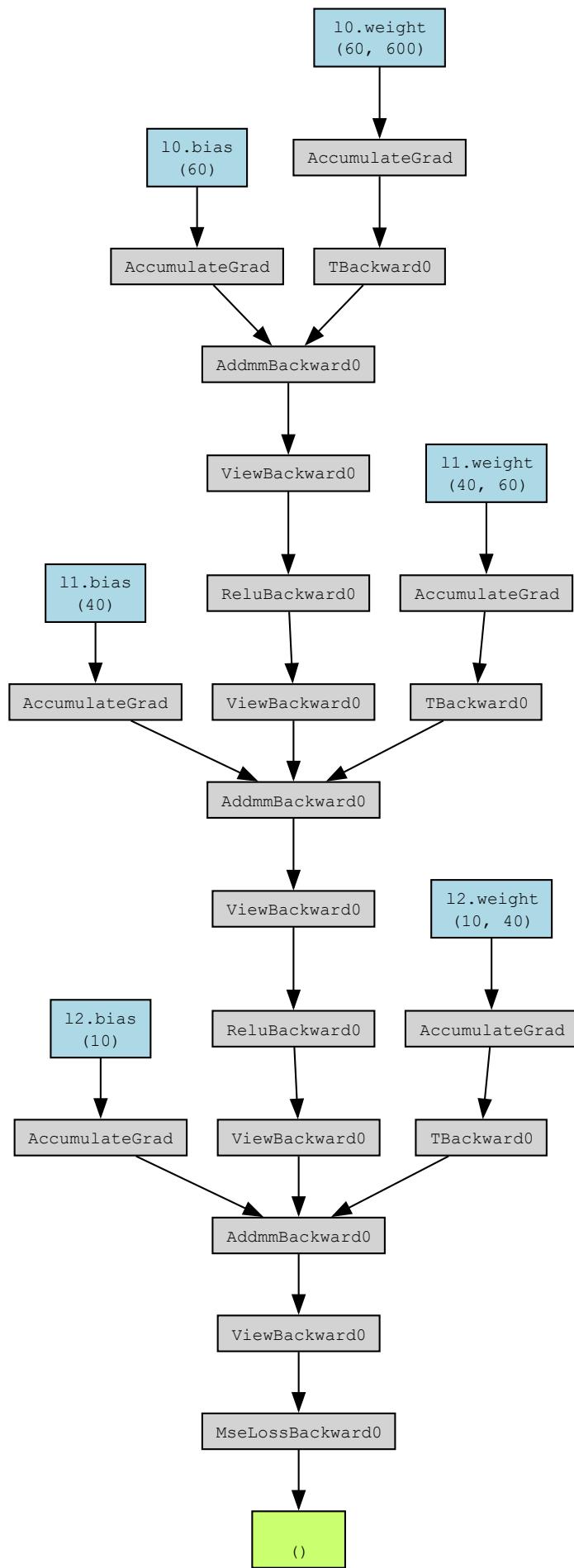
class MyNet(nn.Module):
    def __init__(self, d0, d1, d2, d3): # d0: Input dimension, di: hidden layer i dimension.
        super().__init__()
        # Define fully connected (FC) layers
        self.l0 = nn.Linear(d0, d1) # First linear layer (input -> hidden layer 1)
        self.l1 = nn.Linear(d1, d2) # Second linear layer (hidden layer 1 -> hidden layer 2)
        self.l2 = nn.Linear(d2, d3) # Third linear layer (hidden layer 2 -> output layer)
        # Note: nn.Linear applies a linear transformation to the input: X*W^T + b

    def forward(self, x):
        a0 = x.view(-1) # Flatten the input tensor to a 1D vector (required for linear layers)
        # Apply first FC layer (weighted sum + bias) followed by ReLU activation
        z1 = self.l0(a0)
        a1 = torch.relu(z1)
        # Apply second FC layer followed by ReLU activation
        z2 = self.l1(a1)
        a2 = torch.relu(z2)
        # Apply final FC layer (produces the output)
        z3 = self.l2(a2)
        return z3 # output tensor

# Instantiate the model with specified layer dimensions
model = MyNet(d0, 60, 40, 10)
# Perform a forward pass with the input image
out = model(image)
```

Note: Weight matrix and bias vector are implicitly defined within the ***nn.Linear*** itself. For them **requires_grad** is set automatically to **True**.

In the Figure below I also report the PyTorch execution graph and trace that would be obtained from this neural network. You can notice that the derivatives of the operations are tracked starting from the nodes having **requires_grad=True**, so in our case the weight matrix and bias vector of each layer (i.e. for instance the input is not tracked). Moreover, in the end I added a MSE loss, but depending on the objective of the task we want to perform could be another type of loss.



When working with PyTorch, it helps to think of neural networks as modular components that can be connected like **building blocks**. PyTorch provides a variety of modules that can be combined to build neural networks in a plug-and-play manner, including layers like Linear, activation functions such as ReLU and operations like Add, Max and LogSoftMax.

In later sections, we will explore different activation functions such as ReLU, sigmoid, etc. as well as various loss functions including squared error, cross-entropy, hinge loss, ranking loss, etc., all of which are essential for training deep learning models.

Directed Acyclic Graphs for Backpropagation: Any directed acyclic graph is OK for backprop? Yes, as long as there exists a well-defined partial order among the modules. The graph must be both directed and acyclic, meaning it cannot contain loops. If loops are present, they must be “unrolled” before applying backpropagation. This is exactly what happens in Recurrent Neural Networks (RNNs) when using Backpropagation Through Time (BPTT), a topic we will explore in a later chapter. Once unrolled, backpropagation can be applied as usual.

Learning Representations

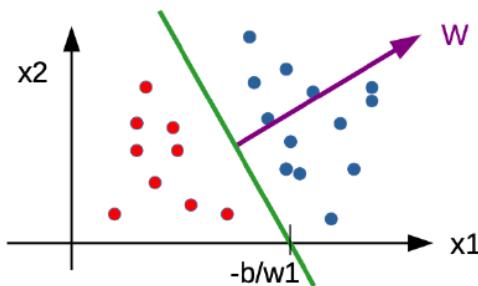
Linear Classifiers and their Limitations

One of the fundamental tasks in ML is to classify data points into different categories. Suppose we have a dataset consisting of two classes, where our goal is to learn a way to separate the positive (blue) data points from the negative (red) ones.

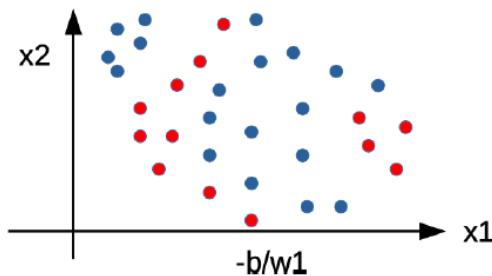
If the data points are **linearly separable**, a linear classifier is an effective solution. Mathematically, a linear classifier is defined as:

$$\bar{y} = \text{sign}\left(\sum_{j=1}^n w_j x_j + b\right)$$

This equation partitions the space into two half-spaces, separated by a **linear** hyperplane.



Once trained, the classifier predicts the class of a new test sample based on which side of the hyperplane it falls on. However, what happens when the data is **not linearly separable**? Consider the example below:



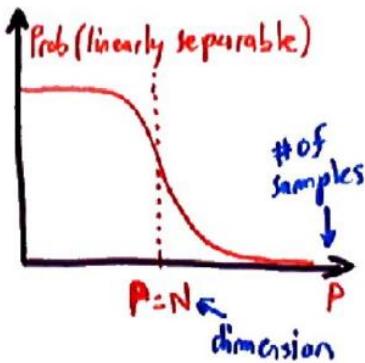
Here, a straight-line decision boundary cannot effectively separate the two classes.

To try to solve this problem, consider the following theorem on pattern separability.

Cover's Theorem on Pattern Separability (1966): *The probability that a random dichotomy of P points in N dimensions is linearly separable approaches zero as p becomes significantly larger than N .*

Therefore, given a certain collection of P samples, only about N are linearly separable. If P is larger than N , the probability that a random dichotomy is linearly separable is very, very small.

In simpler terms, as the number of data points increases beyond the number of available dimensions, it becomes increasingly unlikely that a linear boundary can separate the classes.



- Problem: there are 2^P possible dichotomies of P points.
- Only about N are linearly separable.
- If P is larger than N , the probability that a random dichotomy is linearly separable is very, very small.

So, coming back to our previous example, the problem is that we have **too many data points in only two dimensions**. To make this data linearly separable, it is necessary to increase the number of dimensions.

Therefore, what we could do is expand the dimension non-linearly. But how?

→ **Basic principle:** expanding the dimension of the representation so that things are more likely to become linearly separable.

Shallow Networks and the Need for Deep Architectures

During ML course, we explored how **Support Vector Machines (SVMs) with kernel functions** can handle non-linearly separable data. Instead of attempting to classify data directly in its original space, we apply a **transformation function** that maps the data into a higher-dimensional space where it becomes **linearly separable**. This allows us to use a simple **linear hyperplane** to separate the classes effectively.

However, explicitly computing this transformation is often **computationally expensive**. To avoid this, we leverage **kernel functions**, which compute the inner product in the transformed space **without explicitly performing the transformation**. Common kernel functions include:

- **Radial Basis Function (RBF)**
- **Polynomial Kernel (degree d)**
- **Gaussian Kernel**

Kernel functions operate **directly on the original data points** but yield results equivalent to computing their inner product in the transformed space:

$$K(X, X^i) = \Phi(X) \cdot \Phi(X^i)$$

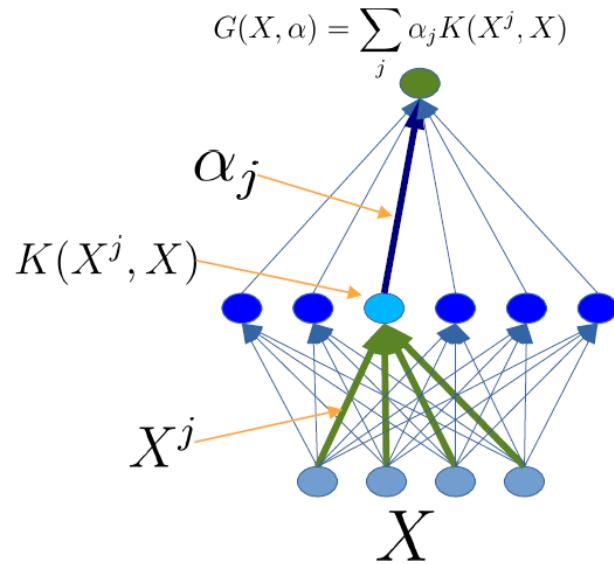
More generally, **machine learning algorithms that rely on kernel functions** are known as **kernel machines** and follow the form:

$$\bar{y} = \sum_{i=1}^m \alpha_i K(X, X^i)$$

Another key advancement in the last two decades has been the adoption of **shallow networks**, neural networks consisting of just **two layers**:

- Layer1: dot products + non-linear function;

- Layer2: linear



Mathematically, this can be expressed as:

$$\bar{y} = F(W^{-1} \cdot F(W^0 \cdot X))$$

Shallow networks are sometimes called “**universal approximators**” because, if they contain a sufficiently large number of neurons, they can approximate most continuous functions. However, in practice, **few useful functions can be efficiently represented with only two layers of reasonable size**.

To overcome this limitation, **Deep Neural Networks (DNNs)**, with multiple hidden layers, are able to represent a wide range of complex functions. A deep network can be expressed as:

$$\bar{y} = F(W^K \cdot F(W^{K-1} \cdot F(\dots F(W^0 \cdot X) \dots)))$$

Do we really need deep architectures?

A fundamental question in ML is whether deep architectures are truly necessary, given that shallow networks are already theoretically capable of approximating any continuous function. While it is mathematically possible for a shallow neural network to approximate any function, the key consideration is not just feasibility but also efficiency and practicality.

Deep neural networks have proven to be significantly more effective when dealing with complex machine learning tasks such as visual recognition, natural language processing and speech recognition. One major advantage of deep architectures is their ability to capture hierarchical patterns in data, making them particularly useful for structured information like images or text. Additionally, deep networks can represent complex functions more efficiently with fewer computational resources compared to shallow networks that would require an impractically large number of parameters to achieve the same results.

Bengio and LeCun (2007), in their paper *Scaling Learning Algorithms towards AI* [1], argued that deep architectures allow for more compact representations of wide families of functions compared to shallow architectures, as they can **trade space for time** — or more specifically, **breadth for depth**. While deep networks require more sequential computations (more layers), they often demand **less hardware in terms of parallel computation**. Their work demonstrated that deep architectures are often more efficient in terms of the number of computational components and parameters needed to represent common functions.

In practice, while shallow networks may suffice in some scenarios, deep architectures have consistently proven to be more efficient and scalable. They enable the learning of richer representations, achieve superior performance with fewer computational resources and are better suited for real-world applications that require handling large and complex datasets.

Invariant Feature Learning

Invariant features are the **essential properties of the input that remain unchanged** despite variations in conditions such as position, lighting or transformations. In DL, these are the features we aim to learn and extract to ensure our model generalizes well to new data.

For example, in face recognition, invariant features could be the proportions of the face, the distance between the eyes, the shape of the mouth and so on, which remain **consistent** regardless of changes in angle, lighting or facial expressions. Identifying these invariant properties of the face will make it easier for the model to recognize and distinguish faces.

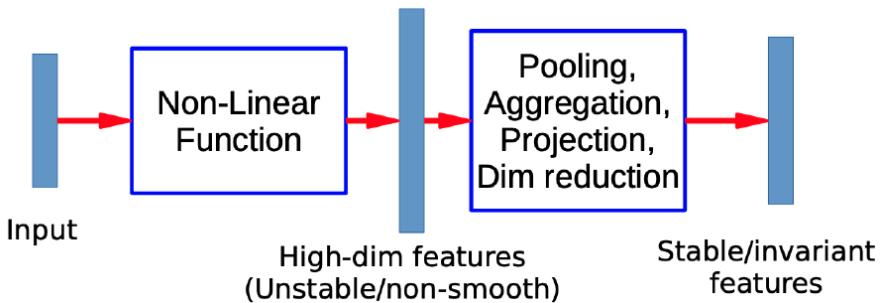
Basic Approach to Invariant Feature Learning:

1. **Embed the input non-linearly into a high(er) dimensional space:**

- In this space, features that were previously entangled **become more separable**, making it easier to distinguish meaningful patterns.

2. **Pool regions of the new space together:**

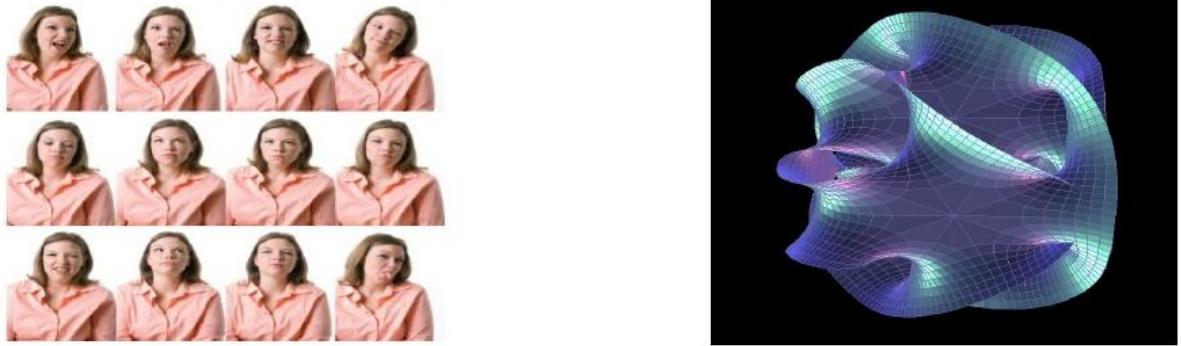
- This helps in bringing together things that are **semantically similar**.
- Like pooling in convolutional networks (that we will see in a next chapter), this step enhances invariance by **preserving essential patterns** (stable features) while ignoring unimportant variations (unstable features).



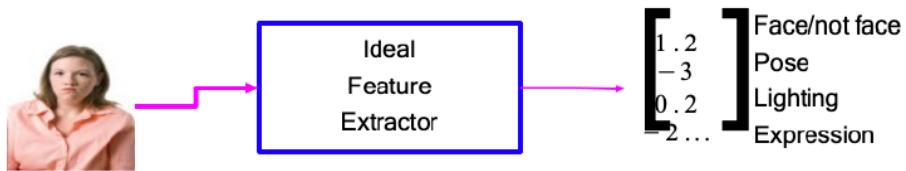
The Manifold Hypothesis

The **Manifold Hypothesis** posits that high-dimensional natural data (such as images of a face) tend to lie on low-dimensional manifolds, meaning they exist in subspaces with lower dimensions than the original data. This is because the variables in natural data are mutually dependent and the information they contain is frequently correlated.

For instance, face images can be represented in a space with fewer dimensions (i.e. < 56) compared to the original dimension of the images (1000x1000 pixels = 1,000,000 dimensions).



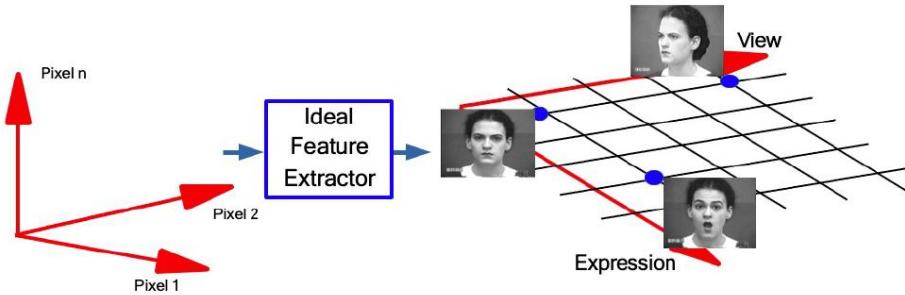
Each image can be viewed as a point in a high-dimensional space, where each pixel represents a dimension. However, there are only a limited number of facial expressions an individual can make, meaning the set of all possible images of a person's face does not reside in the high-dimensional space of one million pixels, but rather in a low-dimensional space defined by just a few dimensions.



The idea is that if the true geometric structure of high-dimensional data can be found and represented in a lower-dimensional space, learning becomes more efficient and accurate. This means that, rather than considering each dimension of an image as an independent feature, we seek combinations of features that represent the real underlying structure of the data, thus reducing the dimensionality of the problem.

Disentangling Factors of Variation

The concept of **Disentangling Factors of Variation** refers to the ability of a ML model to identify and separate different sources of variation within the data. In other words, it involves extracting data representations that remain invariant to unwanted variations, such as changes in position, lighting or facial expression in face images, so that the features relevant to the specific task at hand can be isolated.

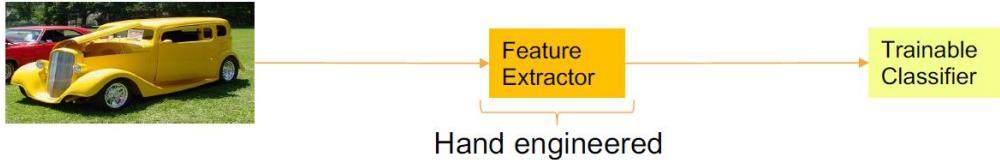


An “**Ideal Disentangling Feature Extractor**” would be a feature extractor that fully separates the different sources of variation in the data, representing each factor of variation independently from the others. However, in practice, it is difficult to achieve such a perfect feature extractor, and efforts are made to approximate this goal using different deep learning techniques.

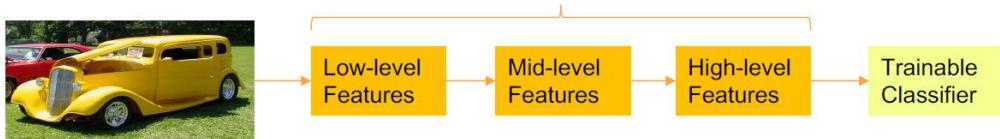
Learning Hierarchical Representations

In traditional ML, the typical approach involves manually extracting relevant features and then feeding them into a ML model. However, for many problems, it is difficult to determine which features should be extracted and moreover they are not so easy to define.

Traditional Machine Learning

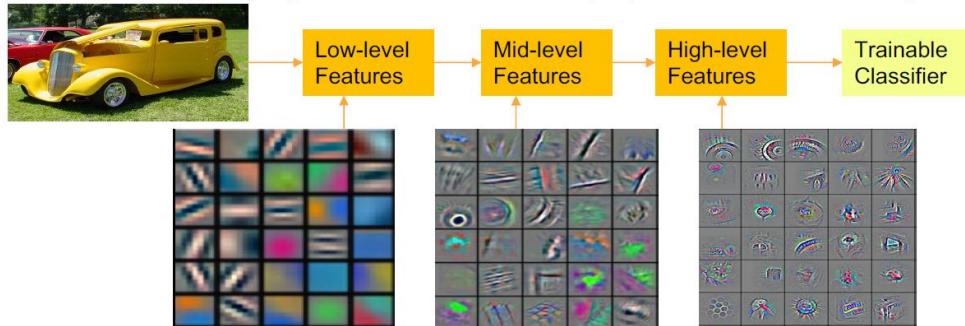


Deep Learning



DL takes a different approach starting from an intuition: natural data is inherently **compositional**, so it can make sense to efficiently represent them through a **hierarchical structure**. Therefore, instead of relying on hand-engineered features, deep learning models aim to **learn the representation of the data in a hierarchical manner** where more complex features emerge as combinations of simpler ones. The first layers of a deep network capture **low-level features** such as edges and textures. These are then combined in subsequent layers to form **mid-level features**, like shapes and contours. Finally, in the deepest layers, the network learns **high-level features** that represent more abstract concepts, such as object parts and complete objects.

Below in Figure there is a visualization of the features at different levels (low-level, mid-level, high-level) learned by a convolutional neural network trained on ImageNet ([we will explore convolutional networks in more detail in a later chapter](#)).



The ability to automatically learn and refine hierarchical features is one of the key strengths of DL, making it highly effective across various domains. Follow some examples of the hierarchy of features that are learned in different domains:

- **Image recognition:** Pixel → edge → texton → motif → part → object
- **Text:** Character → word → word group → clause → sentence → story
- **Speech:** Sample → spectral band → sound → ⋯ → phone → phoneme → word

Activation Functions

In neural networks, an **activation function** is a mathematical operation applied to a neuron's output, transforming the weighted sum of its input into a new value that is passed on to the next layer.

Different activation functions exhibit unique characteristics, and their effects become particularly pronounced in DNNs. In the Table below, we highlight some commonly used activation functions.

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
Tanh		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
Arctan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU) [2]		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [3]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

Activation functions must be **nonlinear** to enable neural networks to model complex functions. According to the *Universal Approximation Theorem*, a two-layer neural network with a nonlinear activation function can be proven to be a universal function approximator. In contrast, a neural network with linear activation functions in all layers would reduce to a linear model regardless of how many layers it has, severely limiting its expressive power. For instance, the **Identity** activation function does not satisfy this property. When multiple layers use the identity activation function, the entire network is equivalent to a single-layer model (this follows from the fact that the composition of successive linear transformations is itself a linear transformation).

Activation functions should also be **differentiable**, meaning they should have nonzero gradients in most parts of their domain. This property is essential for gradient-based optimization methods to ensure that gradients can flow through the network during backpropagation. For instance, the **Binary step** function is not differentiable at 0, and it differentiates to 0 for all other values, so gradient-based methods can make no progress with it.

Some activation functions are better suited for hidden layers, while others are more appropriate for output layers. In most cases, all the hidden units in a network will be given the same activation function, although in principle there is no reason why different choices could not be applied in different parts of the network.

Threshold - nn.Threshold()

Threshold is one of the oldest activation functions and is defined as:

$$\text{Threshold}(x) = \begin{cases} x, & x > \text{threshold} \\ v, & \text{otherwise} \end{cases}$$

It applies a threshold operation on the input values. Specifically, the function only keeps the input values greater than a certain *threshold* and resets the ones less than that with a *v* value (*threshold* and *v* are parameters that can be passed to nn.Threshold).

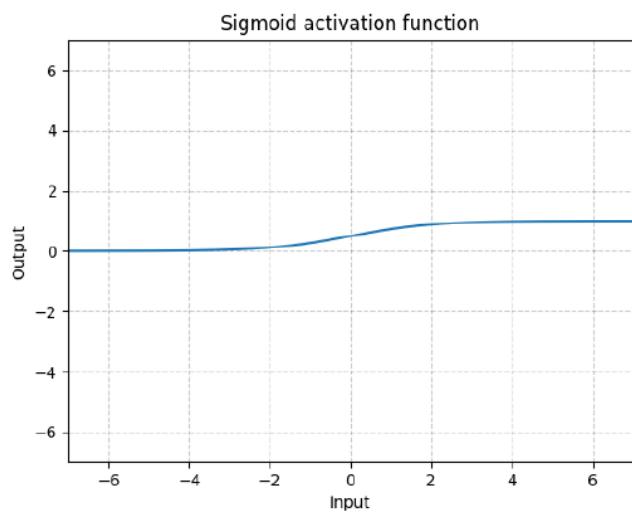
It is **non-differentiable**, so gradient-based methods can make no progress with it, and this is also the reason that prevented people from using backpropagation in the 60s and 70s when they were using binary neurons.

Sigmoid - nn.Sigmoid()

The **Sigmoid** is a simple, nonlinear and differentiable activation function defined as:

$$\text{sigmoid}(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

It maps any real-valued input to an output in the range **[0, 1]**, making it particularly useful when the output needs to represent a probability for a binary decision. Historically, the sigmoid was widely used in early multilayer neural networks; however, its popularity has waned in the years due to a significant drawback that affects its suitability for **deep** neural networks: its gradients go to zero exponentially when the inputs have either large positive or large negative values, and this iterated over many layers can cause the so called "**vanishing gradients**" problem. This makes it less effective as activation function for hidden layers in a deep neural network.



Vanishing and Exploding Gradients

The phenomena of vanishing gradients and exploding gradients occur when we try to train very deep neural networks. From the chain rule of calculus, we know that the gradient of a cost function with respect to the weight matrix of the first layer of the network ([following the denominator layout](#)) is given by:

$$\frac{\partial C}{\partial W^{[1]}} = \frac{\partial z^{[1]}}{\partial W^{[1]}} \cdots \frac{\partial z^{[L]}}{\partial z^{[L-1]}} \frac{\partial C}{\partial z^{[L]}}$$

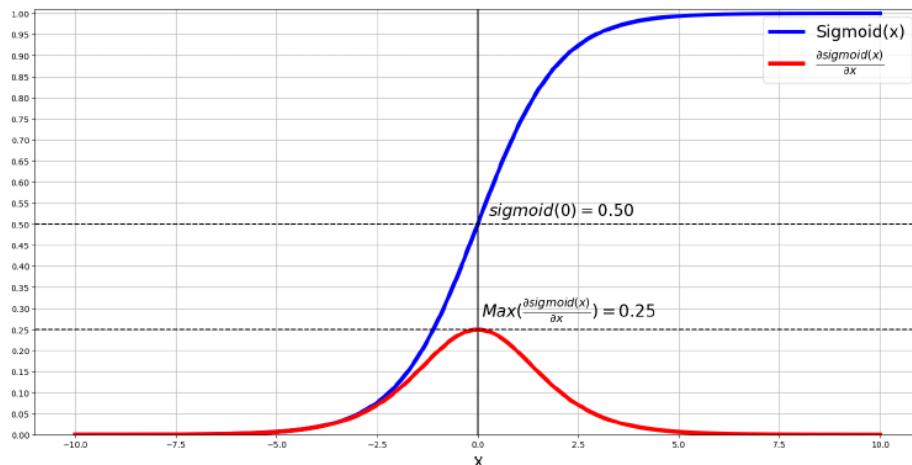
The product of a large number of such terms will tend towards 0 if most of them have a magnitude < 1 leading to **vanishing gradients** or will tend towards ∞ if most of them have a magnitude > 1 leading to **exploding gradients**. Consequently, as the depth of a network increases, error function gradients can tend to become either very small "shutting down" learning in earlier layers, or very large blowing up and so making training unstable.

To better show it, let's consider more in detail what happens for sigmoid. The derivative of sigmoid is:

$$\sigma'(x) = \sigma(x) \odot (1 - \sigma(x))$$

This derivative becomes very small when input values are far from zero, impacting the error term δ during backpropagation in a DNN, since it is part of it ($\delta^{[l]} = ((W^{[l+1]})^T \delta^{[l+1]}) \odot (\sigma'(x^{[l]}))$). Specifically:

- For large positive inputs, $\sigma(x) \approx 1$, we will have that the term inside the derivate $(1 - \sigma(x)) \approx 0$, so $\sigma'(x) \approx 0$ and as consequence $\delta \approx 0$.
- For large negative inputs, $\sigma(x) \approx 0$, we will have that the term inside the derivate $\sigma(x) \approx 0$, again resulting in $\sigma'(x) \approx 0$ and as consequence $\delta \approx 0$.



So, its derivate $\sigma'(x)$ becomes almost always very small — often close to zero. As a result, the gradients quickly diminish as they propagate backward through the multiple layers of the network, continuing to diminish and resulting in even smaller updates for earlier layers, exacerbating the vanishing gradient problem.

Exploding gradients, on the other hand, are generally not an issue with sigmoid activations because its derivative is inherently bounded (max at 0.25), and therefore it never assumes values that could lead to exponential growth of gradients.

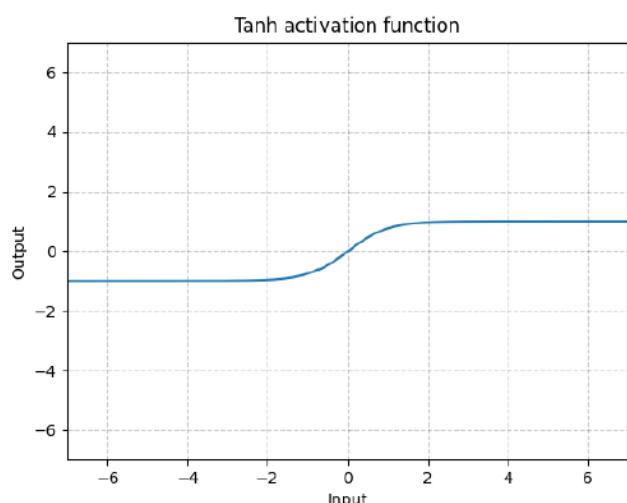
Tanh - nn.Tanh()

Tanh (Hyperbolic Tangent) is an activation function defined as:

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

This function is similar to the Sigmoid function, but it produces outputs in the range of $[-1, 1]$ rather than $[0, 1]$. As a result, its output has a zero mean, which helps to balance weight updates during training. In fact, convergence is usually faster if the mean of each input variable is close to zero (we will cover better this aspect in *Optimization* chapter).

Like Sigmoid, Tanh also suffers from the vanishing gradient problem, making it less suitable as activation function for the hidden layers of a deep neural network.



ReLU- nn.ReLU()

ReLU (Rectified Linear Unit) is an activation function defined as:

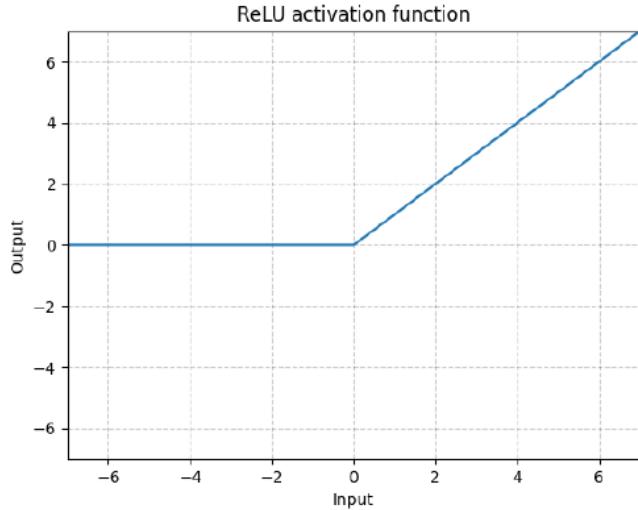
$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

ReLU outputs the input directly if it's positive and zero if the input is negative, for this reason is also known as **positive part**.

The derivative of ReLU is:

$$\text{ReLU}'(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

Since for $x \geq 0$, $\text{ReLU}'(x) = 1$, the gradient for positive values is consistently 1, allowing gradients to propagate as they are without shrinking and therefore avoiding the issue of small gradients (vanishing gradient).



Although the ReLU has a non-zero gradient for positive input values, this is not the case for negative inputs, which can mean that some hidden units receive no 'error signal' during training. This leads to the famous "**dying ReLU**" problem, where neurons with negative inputs always output zero and may stop learning because their gradients are zero. **ReLU variants** were introduced to mitigate this issue by allowing small negative gradients.

ReLU has another important drawback that it is **non-differentiable at $x = 0$** . At this point, the function abruptly shifts from outputting 0 for negative inputs to increasing linearly for positive inputs, resulting in an undefined slope. In other words, there is no unique derivative at $x = 0$ because the left- and right-hand limits do not match. However, this is generally not a major concern in practice: the exact value $x = 0$ is an isolated point, and neural networks rarely operate precisely at zero. Additionally, gradient descent methods manage this discontinuity by using **subgradient** approximations, which allow effective training despite the non-differentiability at that point.

Empirically, ReLU is one of the best performing activation functions and it is in widespread use. Many practical applications simply use ReLU units as the default unless the goal is explicitly to explore the effects of different choices of activation function.

Subgradients

Since standard gradient descent relies on derivatives, we need an alternative when dealing with non-differentiable points. **Subgradient descent** works just like regular gradient descent but uses a **subgradient** instead of an exact gradient. The update rule is:

$$x^{(k+1)} = x^{(k)} - \alpha_k g^{(k)}$$

where $g^{(k)}$ is any subgradient of f at $x^{(k)}$. The **subgradient** at a non-differentiable point is **any slope that lies below the function** (for convex functions), i.e. it is a lower bound.

Note: From calculus we recall that the **derivative** of a function at a point gives the slope of its tangent line.

In deep learning frameworks like PyTorch, **automatic differentiation handles non-differentiable points** by selecting appropriate subgradients. For more details, see the PyTorch [documentation](#).

LeakyReLU - nn.LeakyReLU()

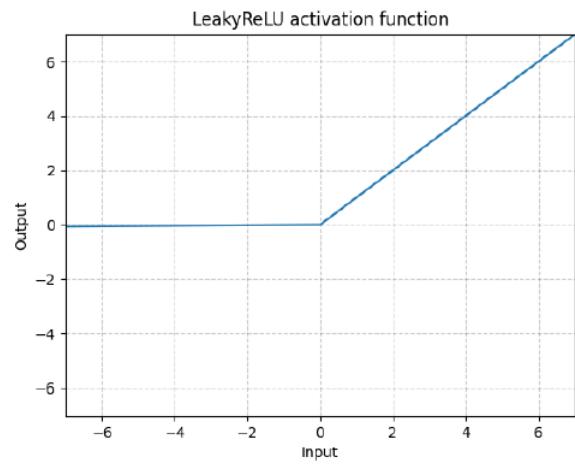
LeakyReLU is a variant of ReLU defined as:

$$\text{LeakyReLU}(x) = \begin{cases} x, & x \geq 0 \\ \alpha x, & x < 0 \end{cases}$$

where α is a **fixed** value (typically very small, e.g., $\alpha = 0.01$) that controls the slope of the linear segment of the function for negative input values ($0 < \alpha < 1$).



This ensures that derivatives are:



$$\text{LeakyReLU}'(x) = \begin{cases} 1, & x \geq 0 \\ \alpha, & x < 0 \end{cases}$$

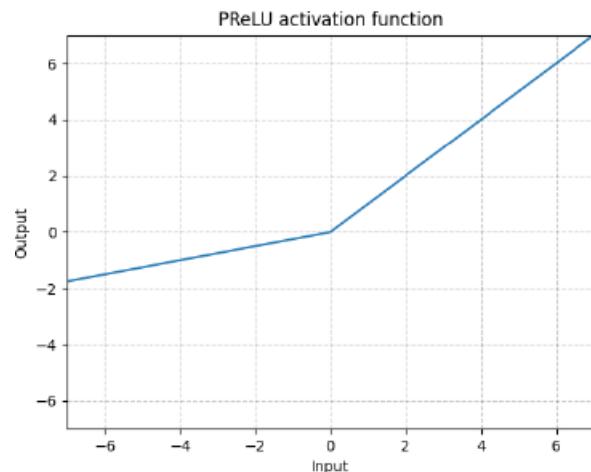
Therefore, unlike ReLU, this has a nonzero gradient for negative values, which prevents the dying ReLU problem by ensuring negative inputs contribute to a small gradient, allowing the neuron to continue learning even with negative inputs.

PReLU - nn.PReLU()

PReLU, or **Parametric ReLU**, is another variant of ReLU that is defined as:

$$\text{PReLU}(x) = \begin{cases} x, & x \geq 0 \\ ax, & x < 0 \end{cases}$$

Unlike LeakyReLU, where the slope of the negative segment is a fixed constant, in PReLU the slope value a is a parameter of the network that is learned during training. This allows the network to learn the best slope values for the network, making it more flexible and adaptable.



The intuition is that different layers may require different types of non-linearity. For instance, the authors when experimented in convolutional layers found that deeper layers generally prefer to have smaller coefficients, suggesting the model becomes more discriminative at later layers, while it wants to retain more information at earlier layers.

RReLU - nn.RReLU()

RReLU, or **Randomized Leaky ReLU**, is another variant of the ReLU function that introduces randomness into the negative slope during training. It is defined as:

$$RReLU(x) = \begin{cases} x, & x \geq 0 \\ \alpha x, & x < 0 \end{cases}$$

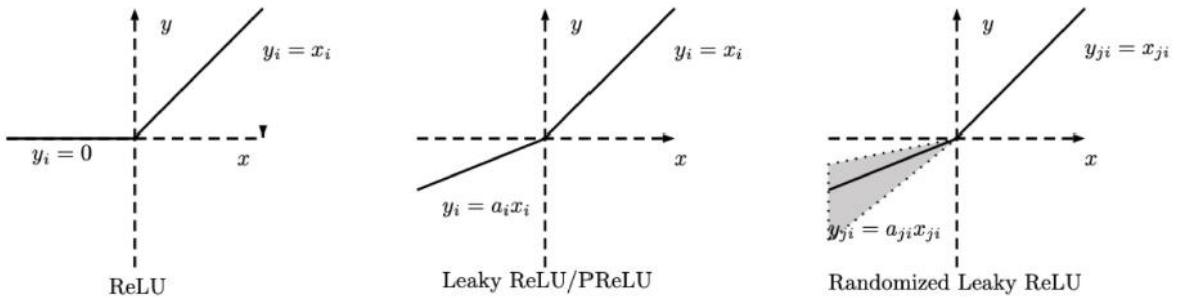
where α is a random value sampled from a uniform distribution, $\alpha \sim U(l, u)$ with $0 \leq l < u < 1$ (i.e. within a given interval $[l, u]$).

- During training, for each negative input, α is randomly sampled from the interval $[l, u]$. This stochastic behavior acts as a form of regularization and can help improve generalization by preventing overfitting.
- In the testing phase, the randomness is removed by fixing α to the average value of the interval, i.e., $\alpha = \frac{l+u}{2}$. This ensures that the output is deterministic.

Recap - Differences between LReLU, PReLU and RReLU:

All the three ReLU variants, i.e. LReLU, PReLU and RReLU, provide negative values in the negative part of their functions where:

- LReLU uses a slightly sloping fixed slope.
- PReLU learns the steepness of this slope.
- RReLU sets this slope to a random value between an upper and lower bound during training and to an average of these bounds during testing.



ELU - nn.ELU()

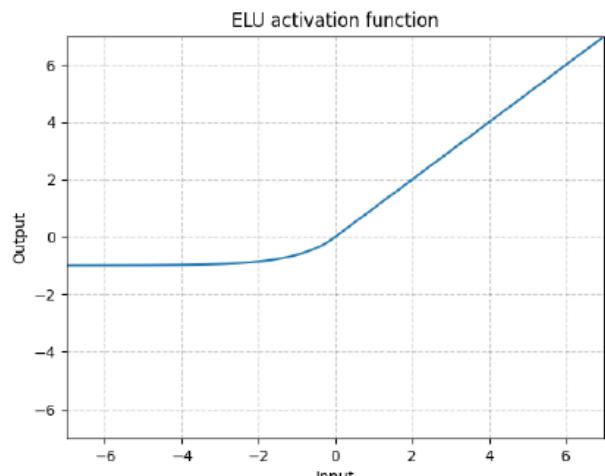
ELU (Exponential Linear Unit) is an activation function defined as:

$$ELU(x) = \max(0, x) + \min(0, \alpha(e^x - 1)) = \begin{cases} x, & x \geq 0 \\ \alpha(e^x - 1), & x < 0 \end{cases}$$

where α is a positive parameter ($\alpha > 0$) that adjusts the slope of the function in the negative region.

Compared to ReLU variants (such as Leaky ReLU), ELU has an exponential negative slope leading to a slight curvature for negative values instead of being flat, which helps to bring the mean activation closer to zero and speed up learning. Moreover, it is differentiable in each point of its domain.

Its variants (CELU, SELU) are simply different parametrizations of ELU, designed to further enhance performance under various conditions.



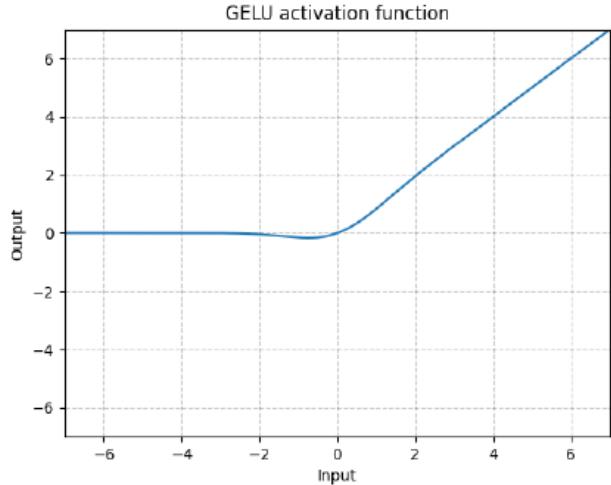
GELU - nn.GELU()

GELU (Gaussian Error Linear Unit) is a smooth, non-monotonic activation function defined as:

$$GELU(x) = x \cdot \Phi(x)$$

where $\Phi(x)$ is the Cumulative Distribution Function (CDF) for Gaussian Distribution.

The GELU activation can be understood as weighting each input by the probability that it is positive under a Gaussian distribution. Instead of applying a hard cutoff (as ReLU does by zeroing all negative values) or a fixed exponential scaling (as in ELU), GELU smoothly scales inputs according to their magnitude. In practice, this means small negative values are only partially suppressed, while larger positive values are passed through almost unchanged. This smooth, probabilistic weighting of inputs allows GELU to capture subtler patterns that rigid activations might miss.



GELU is a **non-monotonic** activation function, meaning that its output does not always increase as the input increases. This happens because the function involves the Gaussian CDF, which introduces a small dip in the curve for negative values where the output slightly decreases before rising again. By contrast, functions like ReLU and ELU are **monotonic**: as the input grows, the output always grows (or at least never decreases). GELU's slight non-monotonicity gives the network more expressive power, allowing it to capture more intricate patterns and approximate complex functions that strictly monotonic activations might overlook.

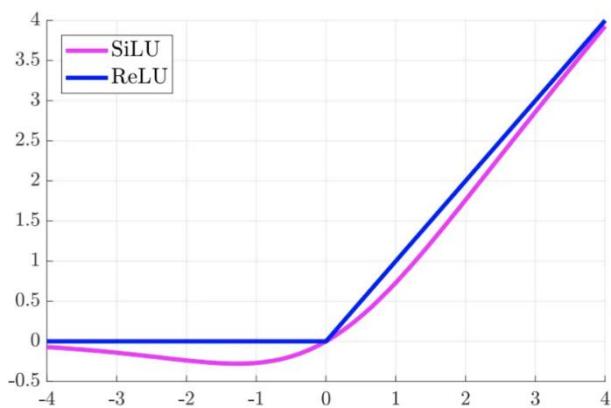
GELU has been shown to improve performance in various tasks, particularly in architecture like Transformers (e.g., BERT and GPT). Its properties often translate into faster convergence and potentially higher accuracy compared to networks using ReLU or ELU.

SiLU – nn.SiLU

Sigmoid Linear Unit (SiLU) is a smooth, non-monotonic activation function that has gained popularity for its efficiency and strong empirical performance. It is defined as:

$$SiLU(x) = x \cdot \sigma(x)$$

where $\sigma(x)$ is the logistic sigmoid.



SiLU is often compared to GELU because of their similar shapes and empirical behavior. Same as GELU, SiLU is non-monotonic, which can help networks model more complex input-output relationships. Both perform well in practice, with GELU remaining the default in many Transformer-based models, while SiLU is common in CNNs and diffusion models (e.g., EfficientNet, many YOLO variants, DDPM UNet). SiLU can also be **slightly more computationally efficient** than GELU, making it attractive in large-scale settings.

Softmax - nn.Softmax()

Softmax is an activation function commonly used in the final layer of neural networks for multi-class classification. It takes as input a vector of real values, also known as **logits**, and produces another vector (having the same dimension as input) where each element represents the probability that the input belongs to the respective class (therefore produces a probability distribution across the classes). This function is defined as:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

where x_i is the raw output (logit) for the i -th class.

This transformation ensures that the elements of the output vector are all positive and sum to 1 (in the range [0,1]), so that it can be interpreted as a probability distribution.

Notice that Softmax amplifies differences between inputs, assigning higher probabilities to larger values.
Example:

$$x = [1, 2, 3]$$

$$\text{Softmax} = [0.09003057, 0.24472847, 0.66524096]$$

Additional Considerations:

- The **Sigmoid function is just a special** (simplistic) **case of the Softmax function**. In fact, if we consider a binary setting with inputs x_1 and x_2 and if we set $x_2 = 0$, then softmax reduces to:

$$s(x_1) = \frac{e^{x_1}}{e^{x_1} + e^{x_2}} = \frac{e^{x_1}}{e^{x_1} + e^0} = \frac{e^{x_1}}{e^{x_1} + 1} = \frac{e^{x_1}}{e^{x_1}(1 + e^{-x_1})} = \frac{1}{1 + e^{-x_1}}$$

which is the familiar sigmoid function.

- The **Softmax function is sometimes called "soft argmax"** because it is a "soft" version of argmax. In other words, argmax returns the index of the element with the maximum value in a vector, while Softmax returns a probability distribution over all elements, assigning higher probabilities to larger values. In this way, the output of softmax not only highlights the maximum element but also provides a measure of confidence for each value, making it a "softer" version of the hard argmax.
- The best cost function to use in combination with Softmax is **Cross-Entropy**.

Softmin - nn.Softmin()

Softmin is the counterpart of softmax that emphasizes lower values rather than higher ones. It transforms a vector of real values into a probability distribution that highlights the smallest values. The softmin function is defined as:

$$\text{Softmin}(x_i) = \frac{\exp(-x_i)}{\sum_j \exp(-x_j)}$$

While softmax is used to identify the most likely class, softmin can be useful in tasks where one needs to select the minimum value. Softmin also amplifies differences between inputs, but it assigns higher probabilities to lower values. Example:

$$x = [1, 2, 3]$$

$$\text{Softmin} = [0.66524096, 0.24472847, 0.09003057]$$

nn.LogSoftmax()

LogSoftmax is defined as the logarithmic function of the Softmax function:

$$\text{LogSoftmax}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right)$$

Rather than returning probabilities, LogSoftmax outputs the **log probabilities**. This logarithmic transformation ensures that the output values are bound and less prone to the numerical instability that arises from exponentiating large logits. It is generally used in combination with the **Negative Log-Likelihood Loss (NLLLoss)** which expects log probabilities as input. Combining LogSoftmax with NLLLoss is generally more numerically stable than applying Softmax followed by a Cross-Entropy loss, as the logarithm is applied directly to the logits before computing the loss, rather than after applying the Softmax function.

Beyond the functions discussed above, there are many other activation functions, such as Softplus, Hardtanh, CELU, SELU, ReLU6, Softsign, Tanhshrink, Hardshrink, Softshrink, and LogSigmoid. However, I chose not to describe them in detail here in order to keep the focus on the most relevant activations.

Saturating functions vs non-saturating functions

An activation function f is **non-saturating** if

$$\lim_{|x| \rightarrow \infty} |\nabla f(x)| \neq 0$$

meaning that its derivative does not approach zero as the input x tend to become very large or very small. This property is beneficial because it helps maintain meaningful gradients during backpropagation, thereby reducing the risk of the vanishing/exploding gradient problem. For instance, the ReLU function is non-saturating in its positive region (for $x > 0$, its derivative is 1).

In contrast, an activation function is **saturating** if

$$\lim_{|x| \rightarrow \infty} |\nabla f(x)| = 0$$

Functions like tanh and sigmoid fall in this category since their derivatives approach zero as x becomes very large or very small. In general, non-saturating activation functions are preferred in hidden layers of deep neural networks because they are less likely to suffer from the vanishing/exploding gradient problem.

Note: It is worth noting that even if a function is considered 'saturating' that doesn't automatically mean that it has 'saturated' (saturating \neq saturated).

Lastly, I want to conclude this chapter underlining that there is **no universally “best” activation function**. Apart from well-known cases such as preferring ReLU over sigmoid to mitigate vanishing gradients, the choice between activation functions like SiLU and GELU typically depends on the specific task and is best determined through hyperparameter tuning and empirical evaluation.

Cost Functions

In DL, **cost functions** (often referenced interchangeably with **loss functions**) play a crucial role in training models by guiding the optimization process. A cost function quantifies how well a model's predictions align with the actual target values, providing feedback that helps the model improve its performance.

Different deep learning tasks require specific cost functions, each designed to address the unique requirements of the problem at hand. In this chapter, we will focus on the most commonly used cost functions implemented in PyTorch. It is assumed that you are already familiar with the concept of activation functions, as well as with some basic cost functions such as *Mean Squared Error* (MSE), *Mean Absolute Error* (MAE), *Binary Cross-Entropy* (BCE), and *Kullback-Leibler* (KL) divergence, which were covered in [ML Course](#). However, many other loss functions exist for various applications, so it is always a good idea to consult the PyTorch documentation to find the most suitable loss for your task.

nn.MSELoss()

In PyTorch, **nn.MSELoss()** measures the **Mean Squared Error (MSE)** between predicted values x and target values y . It is also referred to as **L2 loss** since it is based on the **squared L2 norm** of a vector v . When using a mini-batch of N samples, it computes an individual loss for each sample in the batch, resulting in an N -element loss vector. We can choose to either retain this vector as-is or apply a reduction method to aggregate the losses into a single value.

- If we choose not to reduce it (i.e. set `reduction='none'`), the loss is computed for each sample in the batch and returned as an N -element tensor:

$$l(x, y) = L = \{l_1, \dots, l_N\}^T$$

where N is the batch size and the generic element i in the loss vector is therefore given by:

$$l_i = (x_i - y_i)^2$$

- If we choose to reduce it to a single value, we have two main options (note that the default value is `reduction='mean'`):
 - `reduction='mean'`: returns the mean of all individual losses.
 - `reduction='sum'`: returns the sum of all individual losses (therefore avoids division by N).

$$l(x, y) = \begin{cases} \text{mean}(L), & \text{if } \text{reduction} = \text{"mean"} \\ \text{sum}(L), & \text{if } \text{reduction} = \text{"sum"} \end{cases}$$

nn.L1Loss()

In PyTorch, **nn.L1Loss()** measures the **Mean Absolute Error (MAE)** between predicted values x and target values y . It is referred as **L1 loss** since it is based on the **L1 norm** of a vector v . For a batch of size N , if **unreduced** (i.e. set `reduction='none'`) the loss is:

$$l(x, y) = L = \{l_1, \dots, l_N\}^T$$

where the generic element i in the loss vector is therefore given by:

$$l_i = |x_i - y_i|$$

It also has reduction options of 'mean' and 'sum' similar to what `nn.MSELoss()` have.

L1 loss is more robust to outliers and noise compared to L2 loss. In fact, in L2 loss (MSE) the errors of those outlier/noisy points are squared, so the cost function gets very sensitive to outliers.

A key drawback of L1 loss is that it is not differentiable at zero. Since the absolute value function has a sharp corner at $x = 0$, its gradient is undefined at that point. This motivates the following SmoothL1Loss.

nn.SmoothL1Loss()

The **SmoothL1Loss** uses *L2 loss* (squared error) when the absolute error is smaller than a threshold (defined by β) and switches to *L1 loss* (absolute error) for larger errors. In this way transitions smoothly near zero, avoiding the non-differentiability issue of the L1 loss.

For a batch of size N , if unreduced (i.e. set `reduction='none'`) the loss is:

$$l(x, y) = L = \{l_1, \dots, l_N\}^T$$

with:

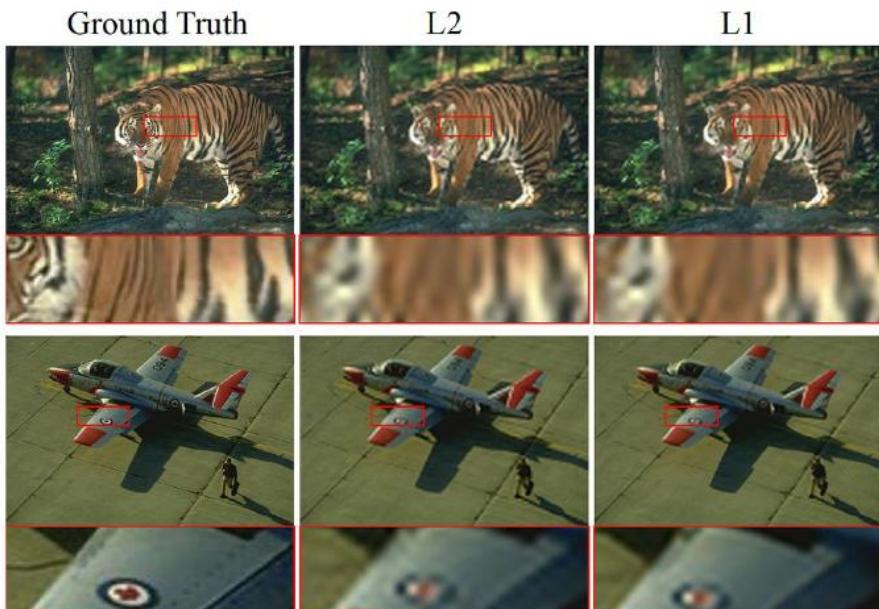
$$l_i = \begin{cases} 0.5(x_i - y_i)^2 / \beta, & \text{if } |x_i - y_i| < \beta \\ |x_i - y_i| - 0.5 * \beta, & \text{otherwise} \end{cases}$$

where β is a hyperparameter that defines the threshold where the loss transitions from L2 to L1 (by default $\beta = 1$). It also has reduction options of 'mean' and 'sum'.

It is less sensitive to outliers than MSE Loss and transitions smoothly near zero, avoiding the non-differentiability issue that occurs in L1 loss.

L1 vs. L2 for Computer Vision

When making predictions in *Computer Vision* (CV) the choice between L1 loss (absolute error) and L2 loss (mean squared error) can significantly affect the quality of the results.



For instance, let's consider [image restoration](#), which is a CV task aimed at improving the quality of images by recovering missing or degraded information. In such a case:

- **L2 loss (MSE):** Optimizing an image restoration method to minimize the mean per-pixel squared difference often leads to **blurry images**. The problem lies in the shape of the loss as it approaches zero. The closer the error is to zero, the smaller the gradient is, meaning that small deviation from the ground truth ([important for sharpness](#)) is not penalized as much.
- **L1 loss (MAE):** On the other hand, L1 loss leads to **sharper images**. L1 has constant gradients, which means that with the loss approaching zero, the gradient will not diminish, resulting in sharper-looking images.

Because of these characteristics, **L1 loss** is often a better choice for image restoration tasks.

!!! Note: If you didn't already get it, here I want underline that neither `nn.L1Loss()` nor `nn.MSELoss()` apply **L1** or **L2 regularization** to the model's weights automatically. If you want to include **weight regularization** (like L1 or L2 penalty) in PyTorch, you need to add them **explicitly to the total loss**.

nn.NLLLoss()

`nn.NLLLoss()`, or **Negative Log Likelihood Loss**, is a loss function used for multi-class classification problems with C classes. It measures how well the predicted log-probabilities align with the true class labels.

Note that, mathematically, the input of NLLLoss should be (log) likelihoods (like LogSoftmax), but PyTorch doesn't enforce that, so be careful when using it. Obtaining log-probabilities in a neural network is easily achieved by adding a `LogSoftmax` layer in its last layer. Therefore, NLLLoss and LogSoftmax are generally designed to be used in combination.

Note: Instead of explicitly applying LogSoftmax and then using NLLLoss, you can also directly use `CrossEntropyLoss`, which combines both steps in one function (and avoids you adding an extra layer).

For a batch of size N , if unreduced (i.e. set `reduction='none'`) the loss can be described as:

$$l(x, y) = L = \{l_1, \dots, l_N\}^T$$

with:

$$l_i = -w_{y_i} x_{i,y_i}$$

where x is the input, y is the target and w is the weight.

Note: The target for `nn.NLLLoss()` should be a class index in the range $[0, C - 1]$, where C is the number of classes.

If reduction is not 'none' (default 'mean'), then:

$$l(x, y) = \begin{cases} \sum_{i=1}^N \frac{1}{\sum_{i=1}^N w_{y_i}} l_i, & \text{if reduction} = \text{"mean"} \\ \sum_{i=1}^N l_i, & \text{if reduction} = \text{"sum"} \end{cases}$$

This loss function has an optional argument **weight** that can be passed in using a 1D Tensor assigning weight to each of the classes (each sample in the batch has a target class y_i and the corresponding weight w_{y_i} is taken from this weight tensor). This is useful when dealing with an **imbalanced** training set.

Weights & Handling Imbalanced Classes

Weighting Class: When dealing with **imbalanced datasets**, where some classes have significantly fewer samples than others, using a **weight vector** can help balance the contribution of each class during training. For example, in a medical dataset, **common flu** cases might be far more frequent than **lung cancer** cases. To ensure the model does not become biased toward the majority class, we can **increase the weight** for underrepresented categories, giving them a stronger influence on the loss function.

Balancing Class Frequency in Training: As an alternative, rather than manually adjusting class weights, a more effective method is to **equalize class frequency during training** by structuring how minibatches are formed:

1. Store samples of each class in separate buffers.
2. Then, when creating a minibatch, sample the **same number of samples** from each buffer, ensuring each class is equally represented.
3. If a smaller buffer runs out of samples to use, we iterate through the smaller buffer from the beginning again until every sample of the larger class is used.

This method ensures that **all classes contribute equally** during training while leveraging stochastic gradients effectively.

Considerations:

- A **common mistake** is to artificially balance the dataset by **discarding** excess samples from the majority class. This is **not recommended**, as it wastes valuable data. Instead, use **resampling techniques** like the one described above.
- A potential downside of equalizing class frequency during training is that the model **loses awareness of real-world class distributions** (real-world data are made up of these unbalanced classes and therefore if we equalize class frequency during training our neural network model would not be able to know the relative frequency of the real data). To solve this, after training with balanced minibatches, we can **fine-tune the model for a few additional epochs at the end using the actual class frequencies**. This step helps the model **recalibrate its output layer**, allowing it to adjust its predictions based on real-world distributions while still benefiting from balanced learning.

nn.CrossEntropyLoss()

Cross-Entropy (CE) loss is a widely used loss function for **multi-class classification tasks**. It measures the difference between the predicted distribution (from the model) and the true distribution (usually represented as a one-hot encoded target vector). For a single sample, CE loss is defined as:

$$CE(x, y) = - \sum_{c=1}^C y_c \log(s(x_c))$$

where:

- x_c is the raw output (logit) of the model for the c -th class,
- y_c is the corresponding element of the one-hot encoded target vector,

- $s(x_c)$ is the softmax of the logits, converting them into a valid probability distribution:

$$s(x_c) = \frac{\exp(x_c)}{\sum_{k=1}^C \exp(x_k)}$$

The softmax is necessary because the model's raw outputs (logits) are unconstrained real numbers. Softmax transforms them into probabilities that sum to 1, allowing CE loss to compare the predicted and true distributions meaningfully. Intuitively, CE penalizes the model when it assigns low probability to the correct class and rewards it when it assigns high probability.

In Pytorch, `nn.CrossEntropyLoss()` combines `nn.LogSoftmax` and `nn.NLLLoss` into a single function. The reason for merging these functions is to improve numerical stability during gradient computation. Applying softmax separately can produce values very close to 0 or 1, whose logarithms can approach $-\infty$ or 0, causing extremely large gradients during backpropagation. By combining both operations, PyTorch ensures stable gradients and avoids numerical saturation issues.

For a batch of size N , if unreduced (i.e., with reduction='none'), the loss is:

$$l(x, y) = L = \{l_1, \dots, l_N\}^T$$

where:

$$l_i = -w_{y_i} \log \frac{\exp(x_{i,y_i})}{\sum_{c=1}^C \exp(x_{i,c})}$$

Here, x represents unnormalized class scores (logits) and an optional `weight` parameter can be used to assign different importance to each class.

+ **Relationship with KL Divergence:** A physical interpretation of the Cross Entropy Loss is related to the Kullback–Leibler (KL) divergence, where we are measuring the divergence between two distributions. In the context of classification, these distributions are represented by the predicted distribution (x vector) and the target distribution (a one-hot vector with 0 on the wrong classes and 1 on the right class). Therefore, we can formalize Cross Entropy in terms of distributions p (target distribution) and q (predicted distribution) as:

$$H(p, q) = - \sum_i p(x_i) \log(q(x_i))$$

Now, let's recall that *Entropy* is used to measure the uncertainty of a system, which is defined as:

$$H(p) = - \sum_i p(x_i) \log(p(x_i))$$

with $p(x_i)$ as the probabilities of different states x_i of the system. From an information theory point of view, $H(p)$ is the amount of information needed for removing the uncertainty.

Now look at the definition of *KL Divergence* which quantifies how much one probability distribution q differs from another p :

$$\mathcal{D}_{KL}(p||q) = \sum_i p(x_i) \log \frac{p(x_i)}{q(x_i)} = \sum_i p(x_i) \log(p(x_i)) - \sum_i p(x_i) \log(q(x_i))$$

where the **first term** is the entropy of distribution p and the second term can be interpreted as the expectation of distribution q in terms of p , capturing how well q approximates p .

Note: The KL divergence tells us how different q is from p from the perspective of p .

From the definitions, we can easily see the relationship that relates cross-entropy with KL divergence:

$$H(p, q) = \mathcal{D}_{KL}(p||q) + H(p)$$

If $H(p)$ is a constant, then minimizing $H(p, q)$ is equivalent to minimizing $\mathcal{D}_{KL}(p||q)$. Therefore, minimizing KL divergence is equivalent to minimizing cross-entropy when entropy remains constant.

A further question follows naturally: “How the entropy can be a constant?”. In a ML task, we begin with a dataset P_D which represents the problem to be solved, and the learning purpose is to make the model estimated distribution P_{model} as close as possible to true distribution of the problem P_{truth} . Since P_{truth} is unknown, we approximate it using P_D , expecting that $P_{model} \approx P_D \approx P_{truth}$, and minimize $\mathcal{D}_{KL}(P_D||P_{model})$. And luckily, in practice the dataset D is given, which means its entropy H_D is fixed as a constant.

+ **nn.AdaptiveLogSoftmaxWithLoss()**: In tasks with an extremely large number of classes (e.g., millions), computing the standard softmax for Cross-Entropy loss becomes very costly in terms of both memory and computation, since it requires evaluating the exponential and normalization over all classes. To address this issue, PyTorch provides `nn.AdaptiveLogSoftmaxWithLoss()`, a variant of Cross-Entropy loss with an efficient approximation of the softmax function. In particular, it uses techniques such as hierarchical class clustering to reduce the number of computations while producing the same loss values as standard CE. The method is detailed in the paper *Efficient Softmax Approximation for GPUs* introduced by Edouard Grave et al. [2].

nn.BCELoss()

The **nn.BCELoss()** (**Binary Cross-Entropy Loss**) is a special case of cross-entropy used when there are only two classes (binary classification problems). For a batch of size N , if unreduced (i.e., with reduction='none'), the loss is defined as:

$$l(x, y) = L = \{l_1, \dots, l_N\}^T$$

where:

$$l_i = -w_i[y_i \log(x_i) + (1 - y_i) \log(1 - x_i)]$$

This loss function assumes that both the predicted values x and target values y are probabilities.

nn.BCEWithLogitsLoss()

The **nn.BCEWithLogitsLoss()** (**BCE Loss with Logits**) is a version of the binary cross entropy loss that takes raw scores (**logits**) that haven't gone through sigmoid, so **it does not assume x is between 0 and 1. It is then passed through a sigmoid σ to ensure it is in that range**. The loss function is more likely to be numerically stable when combined like this.

For a batch of size N , if unreduced (i.e., with reduction='none'), the loss is defined as:

$$l(x, y) = L = \{l_1, \dots, l_N\}^T$$

where:

$$l_i = -w_i[y_i \log \sigma(x_i) + (1 - y_i) \log(1 - \sigma(x_i))]$$

nn.KLDivLoss()

The **nn.KLDivLoss()** (**Kullback-Leibler Divergence Loss**) is a loss function used to compute the KL Divergence between a predicted distribution and a target distribution, where generally the target is a one-hot distribution. For a batch of size N , if unreduced (i.e., with reduction='none'), the loss is defined as:

$$l(x, y) = L = \{l_1, \dots, l_N\}^T$$

where:

$$l_i = y_i(\log y_i - \log x_i)$$

This loss also assumes that both the predicted values x and the target values y are expressed as probabilities.

One drawback of this function is that it is not merged with a softmax or log-softmax, which may lead to potential numerical stability issues during computation.

Hinge Loss

Hinge loss is commonly used in ML for **binary classification** or **similarity tasks**. The core idea is to not only classify correctly but also enforce a **margin** between classes or between similar and dissimilar pairs. For a single example, the hinge loss is typically defined as:

$$l = \max(0, margin - y \cdot x)$$

where x is the model output, $y \in \{-1, 1\}$ is the target label and *margin* defines how far the outputs of the positive and negative classes should be separated.

Intuitively:

- If the model predicts correctly and the score is beyond the margin, the loss is 0. — the model is confident enough.
- If the prediction is correct but the score is within the margin, the loss encourages the model to push it further away from the decision boundary.
- If the prediction is incorrect, the loss increases proportionally to how far it is from satisfying the margin.

The **margin** is crucial because it encourages the model not just to classify correctly but to do so **with confidence**, enforcing a buffer zone around the decision boundary. This often leads to **better generalization**.

Several variants of hinge loss have been developed to accommodate different tasks and training methodologies, such as multi-class classification, ranking, etc., each adapting the basic hinge formulation to specific requirements.

nn.HingeEmbeddingLoss()

The **nn.HingeEmbeddingLoss()** (**Hinge Embedding Loss**) is designed for measuring **similarity between two inputs**. It works by pushing apart dissimilar ones.

Given a pairwise distance x between two inputs (e.g., L1 or L2 distance between two feature vectors) and a target $y \in \{-1, 1\}$ indicating whether the pair should be similar (1) or dissimilar (-1), for i -th sample in the mini-batch this loss is defined as:

$$l_i = \begin{cases} x_i, & \text{if } y_i = 1 \\ \max(0, margin - x_i), & \text{if } y_i = -1 \end{cases}$$

In practice, for positive labels ($y_i = 1$), the distance x_i is taken as it is, while for negative labels ($y_i = -1$), the loss penalizes only distances smaller than the margin, encouraging dissimilar inputs to be separated by at least the margin.

nn.SoftMarginLoss()

nn.SoftMarginLoss() (**Soft Margin Loss**) is a loss function used for optimizing a binary classification problem based on a logistic loss between input tensor x and the target tensor y , where y contains values of either 1 or -1.

$$loss(x, y) = \sum_i \frac{\log(1 + \exp(-y[i] * x[i]))}{x.nelement()}$$

Given a set of positive and negative samples, the function aims to optimize their separation, ensuring a margin of at least 1 ($margin = 1$). Specifically, it **tries to make $\exp(-y[i] * x[i])$ for the correct $x[i]$ smaller than for any incorrect sample**.

nn.MultiLabelMarginLoss()

nn.MultiLabelMarginLoss() (**Multi-Class Hinge Loss**) is a **margin-based loss** function used for **multi-label classification**, where each input sample can belong to multiple classes simultaneously.

Note: Don't confuse multi-class classification with multi-label classification: in multi-class classification a sample can appertain to one of classes, instead in multi-label classification each input can have a variable number of classes (e.g., an image containing both "dog" and "cat").

This margin-based loss accommodates inputs with a varying number of target labels. In this case, multiple categories require high scores, and the loss function computes the hinge loss for each correct category, summing the results across all target labels. In other words, this loss extends the concept of Hinge Loss to multiple target labels by encouraging the model to assign higher scores to correct classes while ensuring a sufficient margin between correct and incorrect classes scores.

For a mini-batch of size N , with input tensor x representing raw scores (logits) for each class and target tensor y containing indices of the ground-truth classes for each sample, the loss is computed as:

$$loss(x, y) = \sum_{ij} \frac{\max(0, 1 - (x[y[j]] - x[i]))}{x.size(0)}$$

where $x[y[j]]$ is the predicted score for the correct class and $x[i]$ represents scores of all possible classes. The function ensures that the predicted score for correct classes is **at least 1 unit greater** than the score for incorrect classes.

nn.TripletMarginLoss()

nn.TripletMarginLoss() (**Triplet Margin Loss**) is a loss function used to measure the relative similarity between samples by ensuring that similar examples are closer together while pushing dissimilar ones further apart.

A **triplet** consists of:

- **Anchor (a):** The reference sample.
- **Positive sample (p):** A sample similar to the anchor.
- **Negative sample (n):** A sample different from the anchor.

The objective of this loss is to minimize the distance between the anchor and the positive sample while maximizing the distance between the anchor and the negative sample, ensuring that the gap between them is at least a given margin.

The loss function for each sample in a mini-batch is defined as:

$$L(a, p, n) = \max(0, d(a_i, p_i) - d(a_i, n_i) + margin)$$

where d represents the distance metric, commonly the Euclidean distance.

This loss function tries to send the first distance toward 0 and the second distance larger than some margin. However, the only thing that matters is that the distance between the good pair is smaller than the distance between the bad pair.

nn.MarginRankingLoss()

Margin-based losses can also be used for **ranking tasks**, where the goal is to **order items correctly** rather than predict exact values. In ranking, each input corresponds to a score assigned to an item, and the model's objective is to ensure that “positive” items are ranked higher than “negative” items according to some criterion (e.g., relevance, preference, or similarity).

In other words, the model doesn't need to predict the exact score for each item — it only needs to maintain the **correct relative ordering**. For example, in a recommendation system, a relevant item should have a higher score than an irrelevant one, and in a search engine, more relevant documents should appear higher in the ranking.

PyTorch provides **nn.MarginRankingLoss()** (**Margin Ranking Loss**) to enforce this ordering by also ensuring that one input is larger than the other by at least a specified margin. The loss function for each pair of samples in the mini-batch is:

$$loss(x_1, x_2, y) = \max(0, -y * (x_1 - x_2) + margin)$$

where:

- x_1, x_2 are the scores assigned by the model for two items (e.g., a positive and a negative item).
- y is a label indicating the expected relationship:
 - $y = 1$ enforces that x_1 should be higher than x_2 ,
 - $y = -1$ enforces the opposite

Intuitively, the loss behaves like a hinge:

- If the difference $y * (x_1 - x_2)$ **exceeds the margin**, the loss is 0 (the ranking is correct and confident).
- If it is **smaller than the margin**, the loss increases linearly, penalizing the model for not maintaining the desired ordering with sufficient separation.

nn.CosineEmbeddingLoss()

Cosine similarity measures the **cosine of the angle** between two vectors in a high-dimensional space, capturing how similar they are based on **direction** rather than **magnitude**. It is defined as:

$$\cos(x_1, x_2) = \frac{x_1 \cdot x_2}{\|x_1\| \|x_2\|}$$

where:

- $x_1 \cdot x_2$ is the dot product of the two vectors,
- $\|x_1\|$ and $\|x_2\|$ are their respective magnitudes (Euclidean norms).

Cosine similarity ranges from **-1 to 1**:

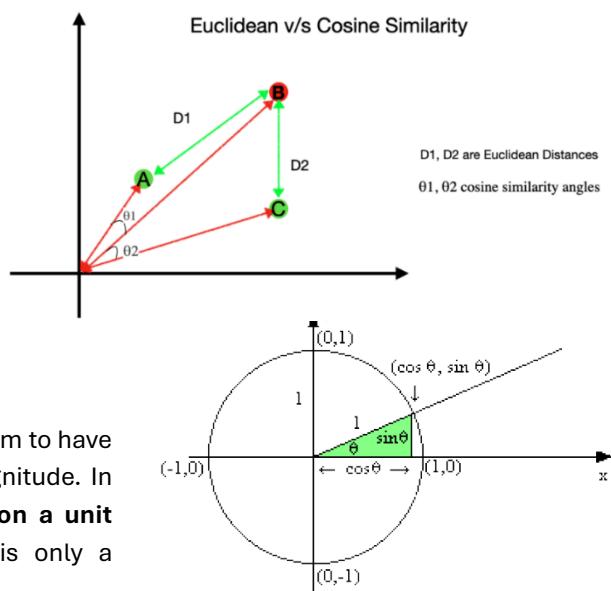
- 1 indicates the vectors are perfectly aligned (highly similar)
- 0 indicates they are orthogonal (unrelated)
- -1 indicates they are diametrically opposed (completely dissimilar).

Why not use Euclidean distance as similarity measure?

Euclidean distance calculates the straight-line distance between two points in space. While this may seem like a natural way to measure similarity it has some **limitations**, especially in high-dimensional spaces. In fact, in high-dimensional spaces the **magnitude** of the vectors can heavily influence their Euclidean distance: if one vector is much longer than another, its distance to other vectors will be artificially increased, even if the direction (or similarity) between the vectors remains unchanged. In other words, vectors that are similar in direction can appear dissimilar if their magnitudes are very different.

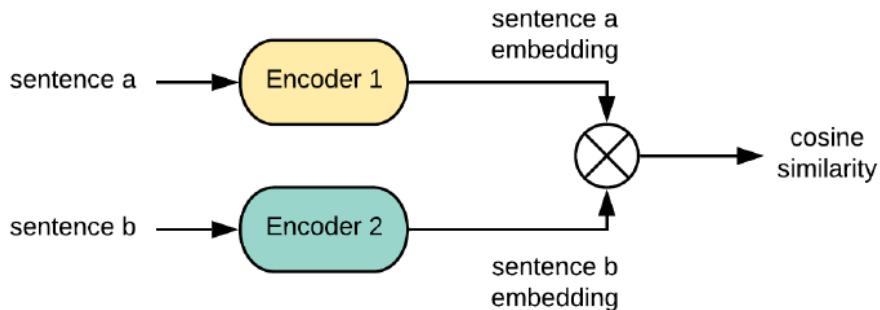
For example, imagine two vectors that point in the same direction, but one is much longer than the other. Euclidean distance would consider these vectors far apart because it measures the straight-line distance between their endpoints, which is large due to the difference in magnitude. However, from a similarity perspective, these vectors are essentially pointing in the same direction — they represent the same pattern or orientation. Cosine similarity, on the other hand, considering only the **direction** of the vectors (ignoring their magnitudes), it would indicate that the two vectors are highly similar, accurately reflecting that they point in the same direction.

In the Figure on side, Euclidean distance would consider points A and B far apart, while B and C appear closer. However, this can be misleading: cosine similarity correctly identifies A and B as highly similar (since their directions align) and B and C as less similar.



Notice that it is by **normalizing** the vectors (scaling them to have unit length) that we can remove any influence of magnitude. In fact, after normalizing the vectors, we **place them on a unit hypersphere**, where the similarity between them is only a function of their directions and depend on their **angle**.

Pytorch provides **`nn.CosineEmbeddingLoss()`** (**Cosine Embedding Loss**), a loss function used to measure the similarity or dissimilarity between two input vectors based on cosine similarity.



For each sample, the Cosine Embedding Loss is defined as:

$$\text{loss}(x, y) = \begin{cases} 1 - \cos(x_1, x_2), & \text{if } y = 1 \\ \max(0, \cos(x_1, x_2) - \text{margin}), & \text{if } y = -1 \end{cases}$$

where *margin* is a real number between -1 and 1 . A value between 0 and 0.5 is suggested and if it is missing, the default value is 0 .

Let's analyze the loss:

1. For $y = 1$ (positive pair), the term $1 - \cos(x_1, x_2)$ is basically the normalized Euclidean distance and is used to encourage the vectors to be as **aligned** as possible:
 - When the cosine similarity between the two vectors is **1** (meaning the vectors are identical), the term $1 - \cos(x_1, x_2)$ becomes **0**, indicating no loss — the vectors are perfectly aligned.
 - As the cosine similarity decreases (i.e., the vectors become less similar), the value of $1 - \cos(x_1, x_2)$ increases, which in turn **increases the loss**. This encourages the model to reduce the distance between similar vectors, pulling them closer together to align them.
2. For $y = -1$ (negative pair), the term $\max(0, \cos(x_1, x_2) - \text{margin})$ helps to **increase the distance** between dissimilar vectors, ensuring they are pushed apart as much as necessary (specified by the margin):
 - **If cosine similarity is greater than margin**, the term $\cos(x_1, x_2) - \text{margin}$ is positive, so the loss increases (penalty is applied).
 - **If cosine similarity is less than or equal to margin**, $\cos(x_1, x_2) - \text{margin} \leq 0$, so $\max(0, \dots)$ returns $0 \rightarrow$ the loss is zero (no penalty)

Optimization

Optimization in DL differs significantly from traditional optimization techniques: in classical optimization, the goal is often to directly minimize or maximize a well-defined function. However, in DL, optimization is typically **indirect** — we do not optimize the final performance measure (such as accuracy on a test set) directly. Instead, a common way to frame DL as an optimization problem is by minimizing the **expected loss over the training data**. In fact, since the true data distribution is unknown, we approximate it using the empirical distribution, which is defined by the available training set. The process of minimizing this empirical loss is known as **empirical risk minimization**. Rather than optimizing the true risk, we optimize the empirical risk, assuming that improvements in training performance will generalize well to new, unseen samples.

During training, our objective is to find the optimal values for the weights and biases of the neural network — values that minimize empirical loss in the hope that they lead to accurate predictions on the test set. For convenience, we group all learnable parameters, including weights and biases, into a single vector θ . The optimization process then seeks to minimize a chosen loss function, often referred to as an **error function $E(\theta)$** .

Said that, in this chapter we will explore a set of techniques that will enable us to optimize our training to make it more stable, speed-up and help to address some issues that we can have when dealing with complex optimization landscapes.

The Error Surface

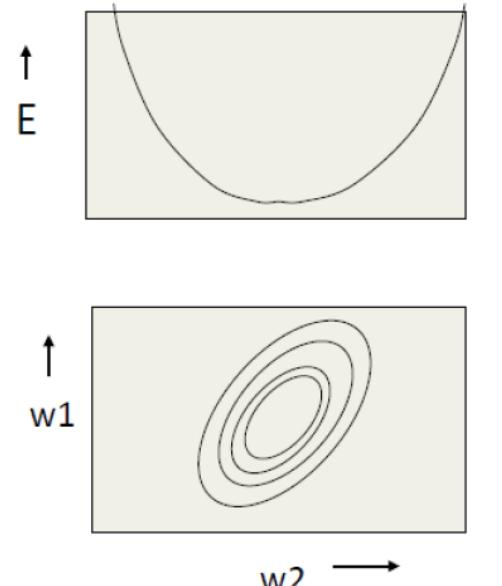
To better understand the optimization process, it is useful to have a geometrical picture of the **error function**, which we can view as a **surface** sitting over ‘parameters space’.

This **error surface** lies in a space with a horizontal axis for each parameter and one vertical axis for the error. So, in this space, points on the horizontal plane correspond to different settings of the parameters and the height corresponds to the error that you're making with that set of parameters.

The objective of training is to adjust parameters in a way that minimizes this error, effectively finding the lowest point on the surface. A natural way to achieve this is by following the **negative gradient** of the error function, which provides the direction of the steepest descent (i.e. the direction of the greatest rate of decrease of the error function) and that is what gradient descent methods do.

Analyzing the shape of the error surface is really important in DL since it can give us insights into how a neural network learns and what challenges it might face. To develop an intuition for this, we first consider the simplest case: a **linear neuron with squared error** (e.g. MSE). The error surface in this case forms a **quadratic bowl**. If we take a **vertical cross-section**, it appears as a parabola (top Figure), and if we take a **horizontal cross-section**, it forms an ellipse (bottom Figure).

This property holds for linear systems with squared error, but for more complex, multi-layer non-linear networks, the error surface becomes much more complicated. However, as long as the parameters remain relatively small, the error surface remains smooth and can still be **locally approximated** by a portion of a quadratic bowl. This

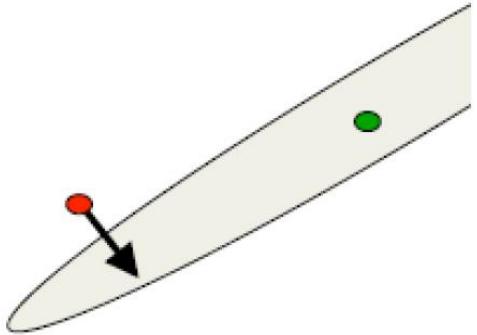


local approximation is useful because, even in non-linear networks, the learning dynamics are influenced by the curvature of the error surface in similar ways.

When considering **convergence speed** in full-batch learning, the quadratic bowl provides useful insights. In an ideal scenario, the natural approach would be to simply move "downhill" along the gradient to reduce the error. However, there is a problem: **the direction of steepest descent does not always lead directly to the minimum unless the surface is a perfect circle.**

To illustrate this, consider an **elliptical error surface**:

- The gradient is **very large along the short axis of the ellipse**, but we only need to move a small distance in that direction.
- Conversely, the gradient is **small along the long axis**, yet that is the direction in which we need to travel a greater distance.



This mismatch causes inefficient updates, as the steepest descent direction is **almost perpendicular** to the optimal path toward the minimum (green point, i.e. the center of the ellipse).

You might wonder why studying such **linear** systems is relevant when optimizing complex, **non-linear** neural networks. The answer is that **even in multi-layer non-linear networks, similar challenges arise**. While their error surfaces are not perfect quadratic bowls, they exhibit the same key properties locally: they **tend to be highly curved in some directions and much flatter in others**. This leads to the same kind of inefficiencies in gradient-based optimization, affecting how quickly and effectively a neural network can learn.

Efficient Gradient Computation: Mini-Batch Learning

As we discussed earlier, gradient descent methods aim to minimize the error function by adjusting the parameters in the direction of the steepest descent. However, computing the gradient over the entire dataset at each step (**full batch learning**) can be highly inefficient, especially when the dataset is large and contains redundant information.

If a dataset has a lot of redundancy, the gradient calculated on one half of the data will be almost identical to the gradient computed on the other half. This means that computing the gradient over the entire dataset is often unnecessary and computationally wasteful. Instead, we can achieve faster and more efficient training by computing the gradient on a subset of the data, updating the parameters and then computing the gradient on the remaining data using the updated parameters.

The extreme version of this approach updates parameters by using the gradient computed after every single training example and is known as "**online**" learning (i.e. SGD). While this allows for rapid updates, it can introduce **high variance** in the updates, leading to oscillations that make the training unstable. On the other hand, we can use **mini-batch learning**, where the gradient is computed over a small batch of training examples before updating the parameters. Mini-batches strike a balance between efficiency and stability, reducing variance while still allowing for faster updates compared to full-batch training.

Mini-batches also have significant computational advantages. Modern hardware, particularly GPUs, is optimized for matrix-matrix multiplications, which makes computing gradients over a batch of training examples highly efficient.

Note: However, it must be said that there are many clever methods to speed up learning when using full batch computation on all training data, such as using non-linear conjugate gradient method (we will see some details about it at the end of the chapter).

When using mini-batches careful attention is needed to balance class distributions within each batch to ensure that each class is equally represented in the mini-batches.

For **large** neural networks with **very large and highly redundant** training sets, it is **nearly** always best to use mini-batch learning.

The mini-batches may need to be **quite big** when adapting fancy methods, such as adaptive optimizers (that we will see soon), as they often rely on more precise estimates of the gradient to function properly.

A bag of tricks for mini-batch gradient descent

When using mini-batch gradient descent, several challenges can arise that slow down learning or make optimization inefficient. However, there are several practical "tricks" that can significantly improve convergence and stability.

1. Data Normalization

In many datasets, input variables can span very different ranges. For example, in health data, a patient's height might be measured in meters (e.g., 1.8 m), while their blood platelet count could be in hundreds of thousands per microliter (e.g., 300,000 platelets/ μL). Such disparities can make gradient descent training much more challenging.

Changes in weights associated with large-magnitude inputs produce much larger changes in the output. This results in an **error surface with very different curvatures along different axes**, making gradient descent inefficient.

A standard approach to address this issue, as seen in the ML course, is **z-score normalization**, which rescales each input feature to have zero mean and unit variance. For each input feature, we compute its mean μ_j and standard deviation σ_j over the training set:

$$\begin{aligned}\mu_j &= \frac{1}{N} \sum_{i=1}^N x_j^{(i)} \\ \sigma_j^2 &= \frac{1}{N} \sum_{i=1}^N (x_j^{(i)} - \mu_j)^2\end{aligned}$$

where N is the number of training samples. Then, each input is normalized as:

$$\hat{x}_j^{(i)} = \frac{x_j^{(i)} - \mu_j}{\sigma_j}$$

Note: This procedure effectively **shifts** the inputs by subtracting the mean and **scales** them by dividing by the standard deviation.

As a result of this procedure, all features contribute roughly equally to the gradient updates, improving the stability and efficiency of training.

!!! Importantly, the same mean and standard deviation computed from the training set should be applied to validation and test data in order to ensure that all inputs are scaled in the same way.

2. Initializing the Weights

If all neurons in a neural network start with identical parameters (weights and biases), they will receive the same gradients during backpropagation. As a result, they will update their parameters in exactly the same way, preventing them from learning distinct features.

To **break this symmetry**, weights should be initialized with small random values, typically drawn from uniform distribution in the range $[-\epsilon, \epsilon]$ or from a zero-mean Gaussian distribution of the form $\mathcal{N}(0, \epsilon^2)$.

The choice of ϵ is important and several heuristics have been proposed to determine an appropriate value. Two widely used methods are **Xavier initialization** and **He initialization**, both of which account for variance.

Ideally, we would like the variance of activations to remain quite stable as information propagates through the layers — neither decays to zero nor grows significantly. If this balance is not maintained, the network can suffer from **vanishing or exploding gradients**, making training ineffective.

Therefore, the goal of these two methods is to initialize the weights in a way that **maintains a stable variance across layers** (it should remain mostly **constant**).

To mathematically guarantee this we should enforce that:

- **Forward propagation:** The variance of activations should remain constant across layers:

$$\forall l, \quad \text{Var}[a^{[l]}] = \text{Var}[a^{[l+1]}]$$

- **Backward propagation:** The variance of activations' gradients should remain constant:

$$\forall l, \quad \text{Var}\left[\frac{\partial E}{\partial a^{[l+1]}}\right] = \text{Var}\left[\frac{\partial E}{\partial a^{[l]}}\right]$$

By developing the first equation mathematically, you end up with the conclusion that to maintain the same variance across layers during forward propagation, we need to initialize the weights in a way to ensure that:

$$\forall l, \quad \text{Var}(W^{[l]}) = \frac{1}{n^{[l]}} = \frac{1}{\text{fan_in}}$$

where **fan-in** is the number of input connections to the given neuron in that layer. This actually results in the so-called **LeCun initialization**, introduced by LeCun et al. in "Efficient Backprop" (1998) [3].

Next, developing the second equation mathematically, we find that to maintain consistent variance during backward propagation, we need to initialize the weights in a way to ensure that:

$$\forall l, \quad \text{Var}(W^{[l]}) = \frac{1}{n^{[l+1]}} = \frac{1}{\text{fan_out}}$$

where **fan-out** is the number of output connections from the given neuron to neurons in the next layer.

At this point we can think about balance variance during both forward and backward propagation by averaging these two values:

$$\text{Var}(W^{[l]}) = \frac{2}{\text{fan_in} + \text{fan_out}}$$

Okay, now that the variance of the distribution is known, we can initialize the weights using either a **normal** or **uniform** distribution:

3. **Normal Distribution:** If $W^{[l]} \sim \mathcal{N}(0, \sigma^2)$ then given that for normal distribution we have $\text{Var}[W^{[l]}] = \sigma^2$, we can find the standard deviation σ as:

$$\frac{2}{\text{fan_in} + \text{fan_out}} = \sigma^2 \rightarrow \sigma = \sqrt{\frac{2}{\text{fan_in} + \text{fan_out}}}$$

4. **Uniform Distribution:** If $W^{[l]} \sim U(-a, a)$ then given that for uniform distribution we have $\text{Var}[W^{[l]}] = \frac{1}{3}a^2$, we can find the bound a as:

$$\frac{2}{\text{fan_in} + \text{fan_out}} = \frac{1}{3}a^2 \rightarrow a^2 = \frac{6}{\text{fan_in} + \text{fan_out}} \rightarrow a = \sqrt{\frac{6}{\text{fan_in} + \text{fan_out}}}$$

This results in the famous **Xavier-Glorot initialization** (or simply **Xavier initialization**) introduced by Xavier Glorot et al. in "*Understanding the Difficulty of Training Deep Feedforward Neural Networks*" (2010) [4]. Therefore, the Xavier initialization suggests initializing weights by sampling from either:

- a **normal distribution** $\mathcal{N}(0, \sigma^2)$, where $\sigma = \sqrt{\frac{2}{\text{fan_in} + \text{fan_out}}}$
- a **uniform distribution** $U(-a, a)$, where $a = \sqrt{\frac{6}{\text{fan_in} + \text{fan_out}}}$

while the biases are typically initialized to zero.

Note: In PyTorch, initializations can be adjusted with a **gain** factor, which is an optional scaling factor to maintain different variance factors for the different layer's weight across the network. This allows for flexible initialization when variance is not strictly 1 between consecutive layers.

Xavier initialization was designed to make the variance of a layer's outputs equal to the variance of its inputs. This approach worked well when using activation functions like logistic sigmoid or also tanh which were commonly used at the time. Both are symmetric around zero, making this balance in variance appropriate.

However, when **ReLU and its variants** became popular, Xavier initialization no longer worked as effectively. This is because ReLU introduces asymmetry by zeroing out negative values, effectively cutting the distribution of activations in half and reducing their variance. To address this, **He Kaiming initialization** (often referred to as simply **He initialization**) was introduced by He Kaiming et al. in their 2015 paper, "*Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*" [5].

Since ReLU sets all negative values to zero, the variance of the activations is effectively halved. To compensate for this, He initialization scales the variance by a factor of 2, ensuring that the activations remain stable across layers.

In their forward propagation derivation, the authors concluded that:

$$\forall l, \quad \text{Var}[W^{[l]}] = \frac{2}{n^{[l]}} = \frac{2}{\text{fan_in}}$$

In the backward propagation derivation, they concluded that:

$$\forall l, \quad \text{Var}[W^{[l]}] = \frac{2}{n^{[l+1]}} = \frac{2}{\text{fan_out}}$$

Interestingly, the authors also found empirically that it is sufficient to use either the forward or backward signal scaling **alone**, as both correctly scale the activations and gradients through the network. Therefore, the choice of scaling factor for He initialization is flexible and either **fan-in** or **fan-out** can be used independently to achieve the desired effect. Therefore, considering for instance the *fan_in* (which is generally the most used), the He initialization suggests in case of use of ReLU (or its variants) to initialize weights by sampling from either:

- A **normal distribution** $\mathcal{N}(0, \sigma^2)$, where $\sigma = \sqrt{\frac{2}{fan_in}}$
- A **uniform distribution** $U(-a, a)$, where $a = \sqrt{\frac{6}{fan_in}}$

while the biases are typically initialized to zero.

Therefore, these methods ensure that the weights for each unit are initialized by drawing values from a distribution that is scaled according to the number of incoming (fan-in) and outgoing (fan-out) connections, which ensure activations and their gradients maintain a stable variance across layers.

Note: If you'd like a deeper understanding, I recommend reading both original papers, along with these helpful articles:

- <https://towardsdatascience.com/xavier-glorot-initialization-in-neural-networks-math-proof-4682bf5c6ec3/>
- https://pouannes.github.io/blog/initialization/#mjax-eqn-eqfwd_K

3. Decorrelate the Input Components

A more advanced, but highly effective technique for improving learning efficiency is **decorrelating the input components**. This approach ensures that the error surface becomes circular, at least for a linear neuron, making gradient descent much more efficient.

In many datasets, different input features are often correlated with each other. If we examine how two input components vary together across the entire training set, we often find a strong correlation. This correlation can make learning more difficult because it skews the error surface into an elongated shape, causing inefficient updates during training.

There are several different ways to decorrelate inputs. A reasonable method is to use **Principal Components Analysis (PCA)**:

- Drop the principal components with the smallest eigenvalues (this achieves some dimensionality reduction).
- Divide the remaining principal components by the square roots of their eigenvalues.

For a linear neuron, this converts an axis-aligned elliptical error surface into a circular one. Once the error surface is circular, the gradient of the error function will always point directly toward the minimum, allowing for much faster and more stable learning.

Note: This technique is primarily useful when working with **structured data** (e.g., tabular datasets, sensor readings, financial data) where input features often exhibit strong correlations. For unstructured data like images, text or audio, such preprocessing is rarely needed because similar benefits are typically achieved through **learned feature extraction**.

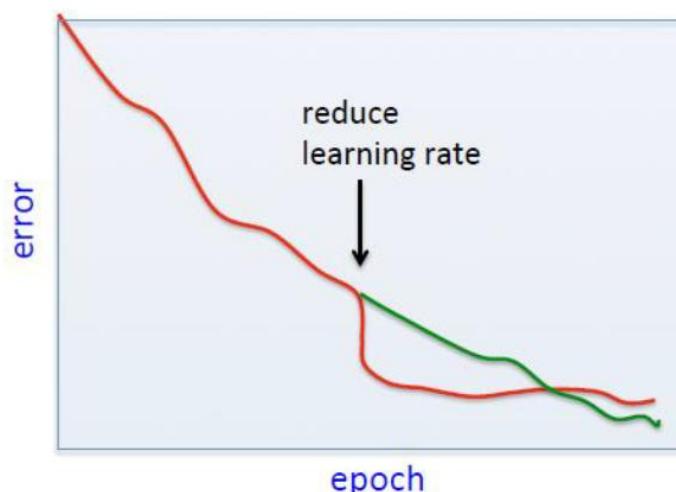
4. Be Careful about Turning Down the Learning Rate

As we will see, **learning rate α** has a very high impact on how effectively a neural network learns. A higher learning rate allows for **larger steps** in parameter updates, while a lower learning rate results in **smaller, more cautious steps**. Selecting an appropriate learning rate is essential for balancing fast convergence and stability.

At the start of training, the parameter space is vast, so large step sizes are beneficial for quickly exploring the direction of optimization. However, a large learning rate near the end of the training can cause instability, leading to oscillations or even divergence.

In practice, an effective strategy is to start with a **higher learning rate** and then schedule a gradual decrease in the learning rate over time, a technique known as **learning rate scheduling**. Examples of learning rate schedules include linear, power law and exponential decay.

Therefore, towards the end of training, it is beneficial to reduce the learning rate to stabilize learning and minimize fluctuations caused by gradient variations in different mini-batches. However, reducing the learning rate too soon can slow down learning prematurely, preventing the model from reaching its best performance.

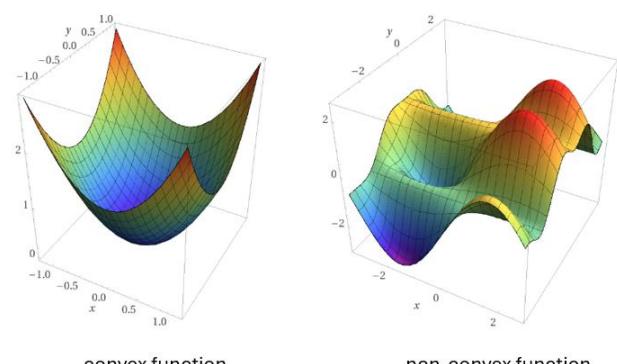


It can be very helpful in practice to monitor the **learning curves** looking at how the error function evolves during the gradient descent iteration to ensure that it is decreasing at a suitable rate.

Note: Other backpropagation strategies can be found in [3], as well as more recent research papers that continue to refine best practices for neural network optimization.

Challenges in Neural Network Optimization

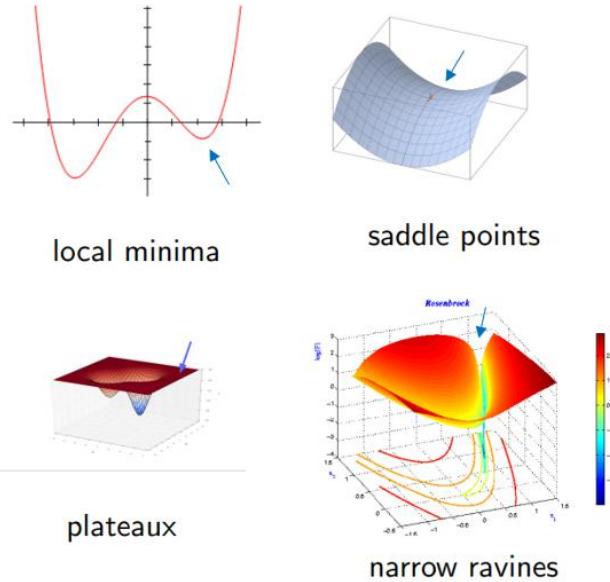
In some cases, optimization is relatively straightforward because the error surface has a **convex shape**—meaning it has a single, well-defined minimum. We like convex functions because they have a great property: any **local minimum is also a global minimum**, guaranteeing an optimal solution. In fact, a convex function resembles a smooth bowl: no matter where you start, following the gradient downhill will always lead to the lowest point, ensuring convergence to the global minimum.



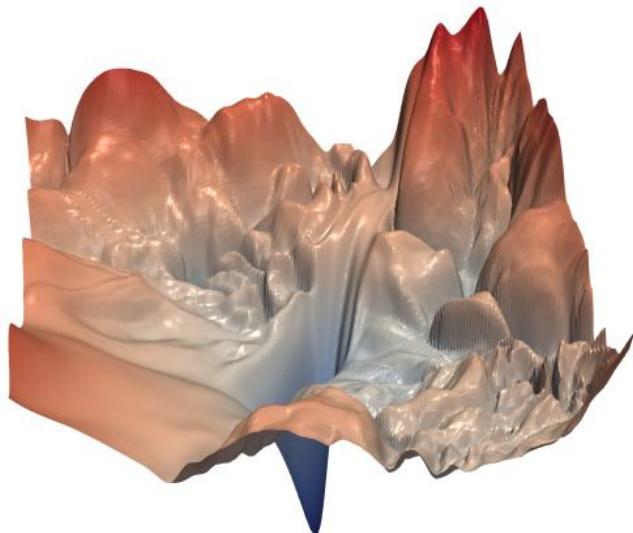
However, **neural networks** introduce a much more complex landscape, as their error surfaces are typically **non-convex** and highly irregular. Therefore, instead of a smooth bowl, we now have a **rugged terrain** filled with multiple **local minima**, **saddle points**, **plateaus** and **narrow ravines** — all of which can make optimization significantly more difficult.

The first challenge is that in non-convex optimization we will have numerous **local minima** along the error surface. This makes it significantly harder to determine whether an optimal solution has been reached risking to be trapped in one of them. In addition to local minima, neural network optimization is hindered by several other problematic regions in the loss landscape:

- **Saddle Points:** These are points where the gradient is zero, but they are neither a minimum nor a maximum. They can slow down optimization significantly, as the gradients in their vicinity become small, leading to **minimal updates**.
- **Plateaus:** These are **flat regions** where the gradient is nearly zero, causing training to slow down or even **stall**.
- **Narrow Ravines:** These are elongated, steep valleys in the loss surface, where the optimization process can oscillate between steep walls rather than smoothly progressing toward the minimum.



For example, the Figure below illustrates a real error surface for a deep neural network. As you can see the optimization landscape is highly complex, posing significant challenges in reaching an optimal or near-optimal solution.

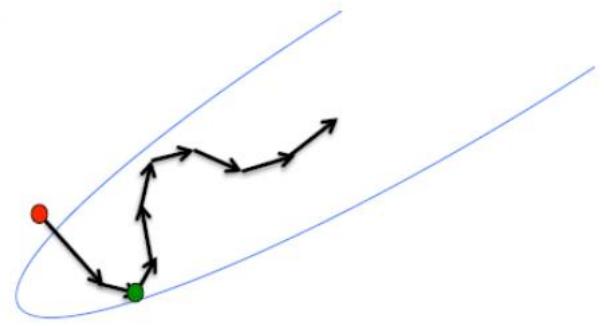


Over the years, various optimization techniques have been developed to address these issues. Now, we will explore some of the most important ones, starting with *momentum*, a method that “accelerates” gradient descent helping into escape flat regions or local minima.

Momentum

Momentum was first introduced by Boris Polyak in 1964 in his seminal work on accelerating iterative optimization methods [6]. The method, often referred to as Polyak's **Heavy Ball Method**, adds a fraction of the previous update to the current one to reduce oscillations and speed up convergence.

To build intuition, imagine a ball rolling down a hilly landscape that represents the loss surface of a neural network. The ball's position in the horizontal plane corresponds to the parameter vector.

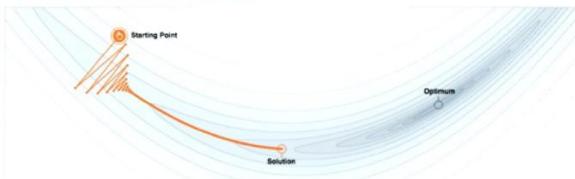


Initially, the ball moves in the direction of the steepest descent (just like gradient descent). However, as the ball gains velocity, it no longer strictly follows the steepest descent path. Instead, its momentum makes it keep going in the previous direction, even if the gradient momentarily changes.

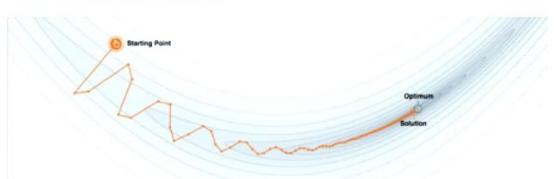
This property of momentum provides three key benefits:

1. **Damping Oscillations:** In regions with high curvature, where gradients fluctuate in opposite directions (back and forth), momentum smooths these oscillations (which are very common in standard SGD), leading to more stable updates.
2. **Speeding Up Convergence:** It accelerates movement in directions with a gentle but consistent slope, reducing training time and leading to faster convergence.
3. **Escaping Local Minima and Flat Regions:** In areas where gradients are small, such as local minima, saddle points or plateaus, momentum acts like inertia — carrying velocity from previous updates and pushing the optimizer forward, preventing it from getting stuck.

Without momentum



With momentum



The Equations of the Momentum

In standard GD, parameters are updated at each step using only the current gradient:

$$\Delta\theta^{(t)} = -\alpha \nabla E(\theta^{(t)})$$

$$\theta^{(t+1)} = \theta^{(t)} + \Delta\theta^{(t)}$$

where α is the learning rate and $\nabla E(\theta^{(t)})$ is the gradient of the error function with respect to parameters θ at time step t .

With momentum, we incorporate past updates to influence the direction of movement. This is done using the following update equations:

$$\Delta\theta^{(t)} = \beta \Delta\theta^{(t-1)} - \alpha \nabla E(\theta^{(t)})$$

$$\theta^{(t+1)} = \theta^{(t)} + \Delta\theta^{(t)}$$

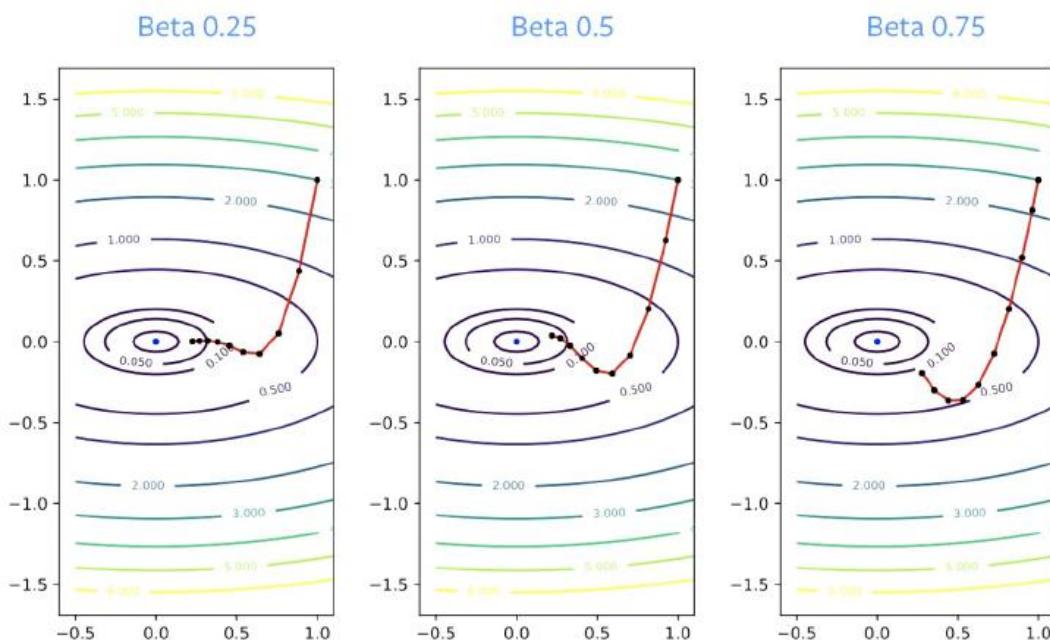
where:

- $\Delta\theta^{(t-1)}$ is the **momentum term**, representing the previous parameter update. It **accumulates information from past gradients** and can be thought as a **running average of the gradients**.
- β is the **momentum coefficient** and determines how much past updates influence the current update.

At each update step, the momentum term is updated by combining the current gradient with the past momentum. The parameters are then updated by moving in the direction of the new momentum (rather than just using the gradient).

Role of β

The parameter β known **momentum coefficient** or **dampening factor** controls how much past updates influence the current update. β has to be greater than zero, because if it is equal to zero, you are just doing gradient descent. It also has to be less than 1, otherwise everything will blow up. Smaller values of β produce faster changes in direction, while for larger values it takes longer to change trajectory. A typical value used in practice is 0.9 (or even 0.99).



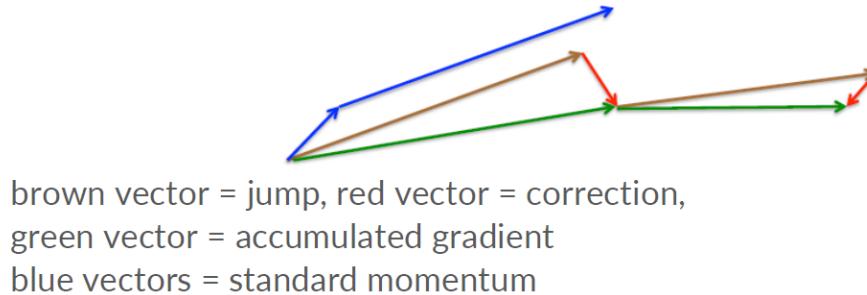
Nesterov Momentum

Momentum is a great method that improves optimization and makes convergence faster. However, the convergence can be further accelerated using a modified version of momentum called **Nesterov Momentum** proposed by Ilya Sutskever (2012) [7].

The standard momentum method first computes the gradient at the current location then take a step that is amplified by adding momentum from the previous step. With the Nesterov method, we change the order of these:

1. First make a big jump in the direction of the previously accumulated gradient (look ahead).

- Then measure the gradient where you end up (at this new position) and make a correction (before making the final update).



The update rules for **Nesterov momentum** are as follows:

- Look ahead** by making a temporary parameter update (jump) based on the previously accumulated momentum:

$$\hat{\theta}^{(t)} = \theta^{(t)} + \beta \Delta\theta^{(t-1)}$$

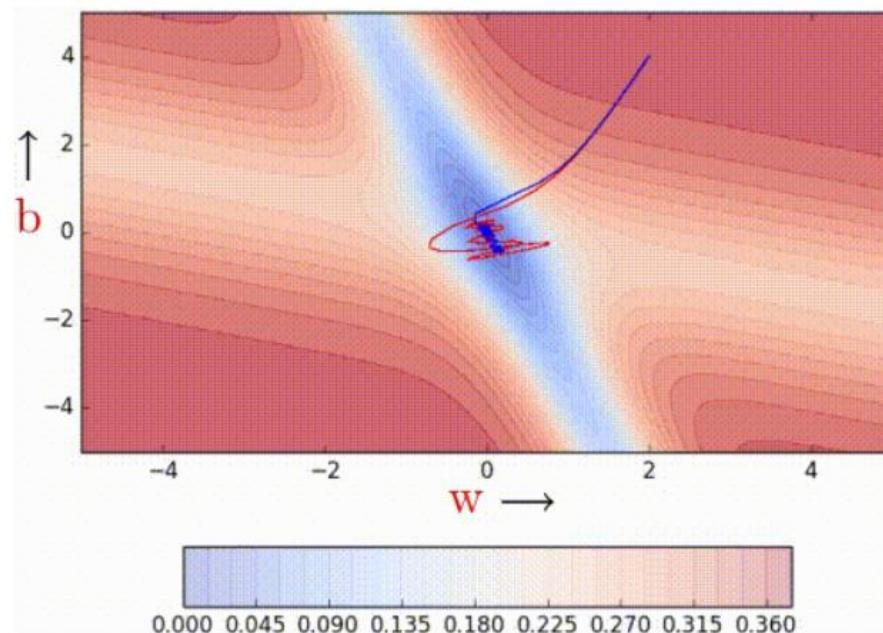
- Compute the gradient at this new look-ahead position:**

$$\Delta\theta^{(t)} = \beta \Delta\theta^{(t-1)} - \alpha \nabla E(\hat{\theta}^{(t)})$$

- Update the parameters** using the newly computed momentum:

$$\theta^{(t+1)} = \theta^{(t)} + \Delta\theta^{(t)}$$

From the Figure below (<https://medium.com/data-science/learning-parameters-part-2-a190bef2d12>), you can see that **Nesterov momentum** (blue) takes **smoother and smaller U-turns** compared to standard momentum (red).



By **looking ahead**, Nesterov momentum anticipates the optimal direction and adjusts its course more efficiently than standard momentum-based gradient descent, reducing oscillations and **making more precise corrections**. This leads to faster convergence and more stable updates.

Why Does Momentum Work?

Nesterov momentum can achieve **accelerated convergence** if the hyperparameters are chosen carefully. However, this theoretical acceleration proof applies only to **convex problems** and does not directly extend to neural networks, which are highly non-convex.

Some believe that standard momentum itself is an **accelerated** optimization method. In reality, momentum provides acceleration **only for quadratic functions**. Moreover, acceleration does not work well with SGD because **SGD introduces noise** and acceleration doesn't work well with noise. Therefore, though some bit of acceleration is present with Momentum SGD, it alone is not a good explanation for the high performance of the technique.

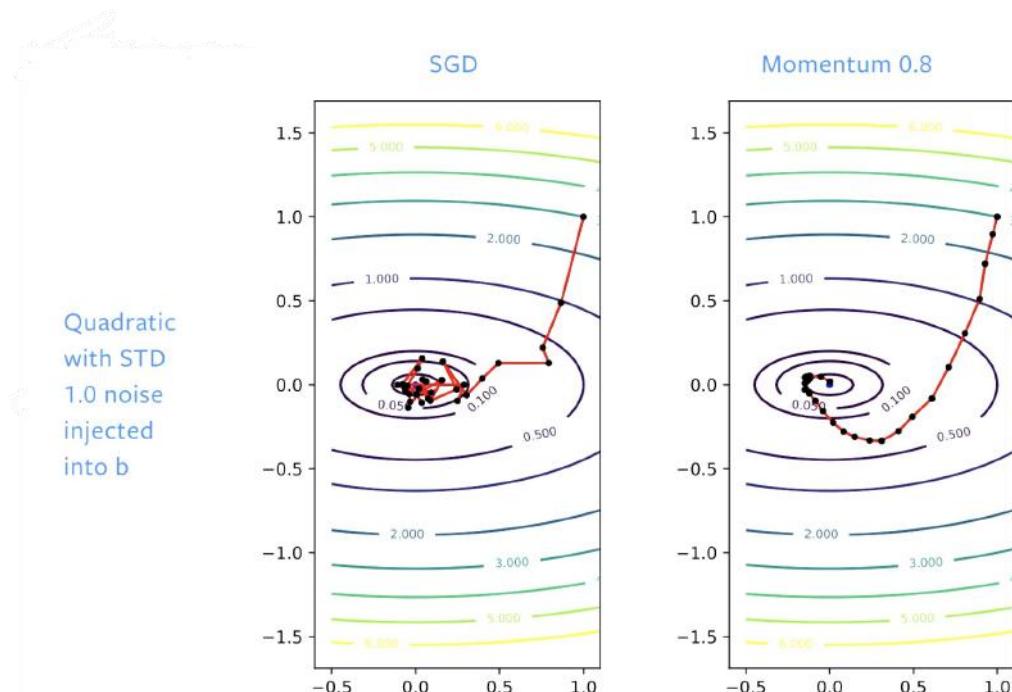
A more **practical and likely explanation** for why momentum work is **noise smoothing**. Momentum **averages gradients over time**, acting as a **running average** rather than relying on noisy individual updates. In theory, to make SGD effective, we would need to average all step updates:

$$\bar{\theta}^{(t)} = \frac{1}{T} \sum_{t=1}^T \theta^{(t)}$$

However, the advantage of **SGD with momentum** is that this explicit averaging becomes no longer necessary. Momentum adds naturally **smoothing** in the optimization process, ensuring that each update is a better approximation of the optimal solution. With standard SGD, you often need to average multiple updates before making a meaningful step in the right direction.

→ Therefore, the final answer is that **both acceleration and noise smoothing contribute to the high performance of momentum.**

Now, notice that without momentum, SGD initially makes good progress but starts to bounce around when it reaches a flatter region or the bottom of a loss valley. Adjusting the learning rate can make bouncing around slower, but momentum smooths out the steps so that there is no bouncing around.



Separate Adaptive Learning Rates

Selecting an appropriate learning rate is one of the most challenging aspects of training a deep neural network due to its significant impact on model performance. The difficulty arises from the structure of the error surface, which defines how the loss function changes with respect to the model's parameters. The curvature of the error surface can vary dramatically across different directions which makes choosing a single global learning rate α problematic. Some directions are steep, meaning small changes in the parameters result in large variations in the error, while others are relatively flat, making the model less sensitive to changes in those parameters.

Ideally, the learning rate should be large in flat directions, where gradients are small but consistent, to accelerate convergence. In contrast, it should be small in steep directions, where gradients are large but unstable, to prevent oscillations.

Although momentum-based optimization methods can help navigate this landscape by smoothing updates and accelerating convergence, it does not fully address the challenges posed by varying curvatures.

To better handle these challenges, **adaptive learning rate** techniques have been developed. These methods assign a unique learning rate to each parameter dynamically, adjusting it throughout training based on the local characteristics of the error surface.

Separate Adaptive Learning Rates based on Gradient Sign

One way to proceed could be to use a **global learning rate (set manually)** multiplied by an appropriate **local gain** that is determined empirically for each parameter:

$$\Delta\theta_{ij} = -\alpha g_{ij} \frac{\partial E}{\partial \theta_{ij}}$$

where α is the (global) learning rate and g_{ij} is an adaptive local gain that adjusts the learning rate individually for each parameter.

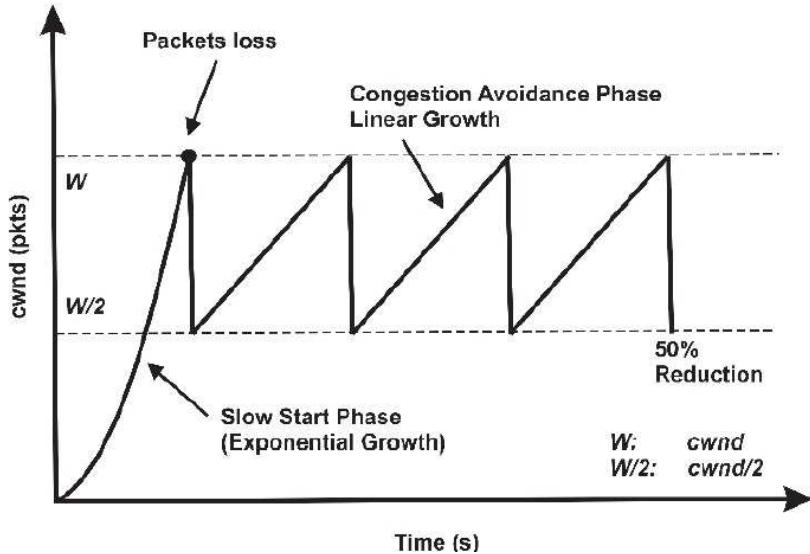
A practical way to determine individual learning rates is to start with an **initial local gain of 1 for each parameter**. Then, we update the local gain dynamically based on the gradient's behavior:

- **If the gradient for that parameter does not change sign** (i.e., the product of consecutive gradients is positive, meaning we are moving in the same direction), the local gain is **increased gradually** using a **small additive factor**.
- **Otherwise** (the two gradients have opposite signs, indicating oscillations) the local gain is **reduced sharply** using a strong multiplicative factor. This ensures that big gains decay quickly when oscillations start.

This update rule is expressed as:

$$\text{if } \left(\frac{\partial E^{(t-1)}}{\partial \theta_{ij}} \frac{\partial E^{(t)}}{\partial \theta_{ij}} \right) > 0 \\ \text{then } g_{ij}^{(t)} = g_{ij}^{(t-1)} + 0.05 \\ \text{else } g_{ij}^{(t)} = g_{ij}^{(t-1)} * 0.95$$

This mechanism is analogous to the **AIMD** (Additive Increase, Multiplicative Decrease) algorithm used in **TCP congestion control**, where the goal is to balance efficient data transmission, while avoiding congestion. AIMD **gradually increases** data transmission until congestion occurs, at which point it **rapidly reduces** the rate to prevent network overload.



Similarly, **adaptive learning rates cautiously increase when learning is stable and quickly decrease when instability arises**.

Tricks to Improve Adaptive Learning Rates:

1. **Limit local gain values to lie in a reasonable range**, such as $[0.1, 10]$ or $[0.01, 100]$, to prevent extreme fluctuations in learning rates.
2. **Use full-batch learning or big mini-batches** to ensure that changes in the sign of the gradient are not mainly due to the sampling error of a mini-batch.
3. **Combine adaptive learning rates with momentum** by considering the agreement in sign not only for the current gradient for each parameter, but also the **velocity** of each parameter.
4. Moreover, consider that adaptive learning rates only deal with **axis-aligned effects** (i.e., per-coordinate partial derivatives), while **momentum accounts for directionality** in a more global manner.

Rprop: Using Only the Sign of the Gradient

We saw that the magnitude of the gradient can vary significantly between different parameters and fluctuate during training, making it difficult to choose a single global learning rate. In full-batch learning, one way to address this issue is by relying only on the **sign** of the gradient rather than its magnitude.

Resilient Propagation (Rprop), introduced in 1993 by Riedmiller et al. [8], builds on this idea. Instead of directly using gradient values, Rprop adjusts the learning rate separately for each parameter based on how the gradient sign changes over time:

- If the gradient for a particular parameter maintains the same sign across two consecutive steps (it means that updates are consistently moving in the right direction), the learning rate for that parameter is **increased multiplicatively** (e.g., by a factor of 1.2).

- Otherwise (If the gradient **changes sign**) the learning rate is **decreased multiplicatively** (e.g., by a factor of **0.5**).

Mathematically, the parameter update rule in Rprop is:

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^+ \Delta_{ij}^{(t-1)}, & \text{if } \frac{\partial E}{\partial \theta_{ij}}^{(t-1)} \frac{\partial E}{\partial \theta_{ij}}^{(t)} > 0 \\ \eta^- \Delta_{ij}^{(t-1)}, & \text{if } \frac{\partial E}{\partial \theta_{ij}}^{(t-1)} \frac{\partial E}{\partial \theta_{ij}}^{(t)} < 0 \\ \Delta_{ij}^{(t-1)}, & \text{otherwise} \end{cases}$$

where $0 < \eta^- < 1 < \eta^+$. Here η^+ represents the factor by which the learning rate increases when consecutive gradients have the same sign, while η^- represents the factor by which the learning rate decreases when the gradient sign changes.

Why Rprop Does Not Work Well with Mini-Batches?

Rprop was designed for **full-batch learning**, where gradients are computed over the entire dataset. However, when applied to **mini-batches**, this approach leads to problems. For example, consider a parameter that receives a gradient of +0.1 in nine mini-batches and -0.9 in the tenth. In Rprop, this parameter would be **incremented nine times and decremented only once**. The overall effect is that the parameter **increases significantly**, even though the average gradient over all ten mini-batches is close to zero. This happens because Rprop treats each mini-batch independently rather than averaging gradients across multiple mini-batches.

The key challenge is to find an algorithm able to retain the robustness of Rprop, while maintaining the efficiency of mini-batch learning and the effective averaging of gradients over mini-batches.

RMSProp: A mini-batch version of Rprop

Basically, Rprop is equivalent to using the gradient for updates, but normalizing (dividing) it by its magnitude. This effectively removes the influence of the gradient's magnitude and focuses only on its sign, making learning more robust to variations in gradient scale.

The problem with Rprop that we have when we consider mini-batches is that we divide by a different number for each mini-batch (this number can fluctuate significantly between mini-batches leading to instabilities between batches). So why not force the number we divide by to be very similar for adjacent mini-batches? This is basically the intuition behind **RMSProp** (which stands for **Root Mean Square Propagation**) introduced in 2012 by G. Hinton. in one of his lectures (**Note:** RMSProp was never published in a formal paper!).

RMSProp maintains a **moving average of the squared gradient** for each parameter, computed as:

$$SquaredGradAvg(\theta_j^{(t)}) = \gamma SquaredGradAvg(\theta_j^{(t-1)}) + (1 - \gamma) \left(\frac{\partial E}{\partial \theta_j}^{(t)} \right)^2$$

and, rather than directly dividing by the gradient magnitude (which varies unpredictably across mini-batches in Rprop), RMSProp divides the gradient by the **square root of this running average**:

$$\theta_j^{(t+1)} = \theta_j^{(t)} - \frac{\alpha}{\sqrt{SquaredGradAvg(\theta_j^{(t)})}} \left(\frac{\partial E}{\partial \theta_j}^{(t)} \right)$$

This running average provides a more consistent estimate of the gradient scale over time, ensuring that the normalization factor changes smoothly between mini-batches.

Note: The use of the square root makes the learning work much better (Tijmen Tieleman).

If we indicate the partial derivative of the loss function w.r.t. to the parameter θ_j at the time step t (i.e., $\frac{\partial E}{\partial \theta_j}^{(t)}$) with $g_j^{(t)}$, we obtain a more general (and often used) form of RMSProp:

$$\mathbb{E}[g_j^2]^{(t)} = \gamma \mathbb{E}[g_j^2]^{(t-1)} + (1 - \gamma) g_j^{2(t)}$$

$$\theta_j^{(t+1)} = \theta_j^{(t)} - \frac{\alpha}{\sqrt{\mathbb{E}[g_j^2]^{(t)} + \epsilon}} g_j^{(t)}$$

where:

- $\mathbb{E}[g_j^2]^{(t)}$ is the running average of the squared gradient for parameter θ_j at time step t .
- γ is a smoothing coefficient similar to the Momentum coefficient β
- ϵ is a small constant (usually in the order of 10^{-8}) introduced to ensure numerical stability when $\mathbb{E}[g_j^2]^{(t)}$ is close to 0.

Adagrad

In many ML problems, especially those involving high-dimensional feature spaces, not all features contribute equally to the learning process. Some features appear frequently and dominate the updates, while others are rare but highly informative. **Adagrad** (which stands for **A**daptive (**s**ub)**gadapting the learning rate for each parameter individually based on the history of gradients. It performs larger updates for infrequent features and smaller updates for frequent ones, effectively allowing the model to learn faster for less common but important features.**

In its update rule, Adagrad modifies the general learning rate α at each time step t for every parameter θ_j based on the **past gradients that have been computed for θ_j** :

$$\theta_j^{(t+1)} = \theta_j^{(t)} - \frac{\alpha}{\sqrt{G_{jj}^{(t)} + \epsilon}} g_j^{(t)}$$

where:

- $G^{(t)} \in \mathbb{R}^{d \times d}$ is a **diagonal matrix** where **each diagonal element jj** is the **sum of the squares of the gradients w.r.t. θ_j up to the time step t** :

$$G^{(t)} = \sum_{\tau=1}^t \begin{bmatrix} g_1^{(\tau)} (g_1^{(\tau)})^T & 0 & \cdots & 0 \\ 0 & g_2^{(\tau)} (g_2^{(\tau)})^T & \ddots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & g_d^{(\tau)} (g_d^{(\tau)})^T \end{bmatrix}$$

- **ϵ is a smoothing term** that avoids division by zero (usually on the order of $1e-8$).

The term $G^{(t)}$ ensures that the learning rate adapts to the geometry of the optimization process. Instead of treating all parameters equally, it rescales the gradient updates based on how frequently and strongly each parameter has been updated in the past:

- If a parameter has consistently received large gradients, the denominator $\sqrt{G_{jj}^{(t)} + \epsilon}$ increases, causing the learning rate for that parameter to decrease.
- Conversely, if a parameter has received small or infrequent gradients, its learning rate remains relatively large, ensuring that rare but important features continue to be learned effectively.

One of the benefits of AdaGrad is that it eliminates the need to manually tune the learning rate; most leave it at a default value of 0.01. However, its main weakness is the **accumulation of the squared gradients in the denominator**: since every added term is positive, the accumulated sum keeps growing during training, causing the learning rate to shrink and becoming infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge.

Adadelta

Adadelta, proposed in 2012 by M. Zeiler (Google/NYU) [10], is an extension of Adagrad that **seeks to reduce its aggressive, monotonically decreasing learning rate**. Instead of accumulating *all* past squared gradients, Adadelta **restricts the window of accumulated past gradients to some fixed size ω** .

Instead of inefficiently storing ω previous squared gradients, **the sum of gradients is recursively defined as a decaying average of all past squared gradients**. The **running average** of the squared gradient for parameter θ_j at time step t , i.e. $\mathbb{E}[g_j^2]^{(t)}$, then **depends** (as a fraction γ) **only on the previous average and the current gradient**:

$$\mathbb{E}[g_j^2]^{(t)} = \gamma \mathbb{E}[g_j^2]^{(t-1)} + (1 - \gamma) g_j^{2(t)}$$

We now simply replace the ii element of the diagonal matrix $G^{(t)}$ with the decaying average over past squared gradients for the parameter θ_i , i.e. $\mathbb{E}[g_j^2]^{(t)}$:

$$\theta_j^{(t+1)} = \theta_j^{(t)} - \frac{\alpha}{\sqrt{\mathbb{E}[g_j^2]^{(t)} + \epsilon}} g_j^{(t)}$$

As the denominator is just the root mean squared (RMS) criterion of the gradient (after adding a small constant ϵ)

$$RMS[g_j]^{(t)} = \sqrt{\mathbb{E}[g_j^2]^{(t)} + \epsilon}$$

we can replace it with the criterion short-hand:

$$\theta_j^{(t+1)} = \theta_j^{(t)} - \frac{\alpha}{RMS[g_j]^{(t)}} g_j^{(t)}$$

At this point, the authors highlight a subtle issue: the units in this parameter update $\Delta\theta_j$ **do not match** the units of the parameter θ_j . Ideally, any update we add to a parameter should have the same units as the parameter itself, but dividing by gradient magnitudes (as above) breaks this consistency. To restore unit consistency, Adadelta introduces a second exponentially decaying average, this time not of squared gradients but of **squared parameter updates**:

$$\mathbb{E}[\Delta\theta_j^2]^{(t)} = \gamma \mathbb{E}[\Delta\theta_j^2]^{(t-1)} + (1 - \gamma)\Delta\theta_j^{2(t)}$$

From this, we define the RMS of parameter updates:

$$RMS[\Delta\theta_j]^{(t)} = \sqrt{\mathbb{E}[\Delta\theta_j^2]^{(t)} + \epsilon}$$

Since $RMS[\Delta\theta_j]$ at time t is unknown, **we approximate it with the RMS of parameter updates until the previous time step ($t - 1$)**. Replacing the learning rate α in the previous update rule with $RMS[\Delta\theta_j]^{(t-1)}$ finally yields the Adadelta update rule:

$$\begin{aligned}\Delta\theta_j^{(t)} &= -\frac{RMS[\Delta\theta_j]^{(t-1)}}{RMS[g_j]^{(t)}} g_j^{(t)} \\ \theta_j^{(t+1)} &= \theta_j^{(t)} + \Delta\theta_j^{(t)}\end{aligned}$$

By incorporating the second moving average in the nominator of the update rule, Adadelta builds an update rule where the units cancel out correctly.

Moreover, you should notice that with Adadelta, we **do not even need to set a default learning rate**, as it has been eliminated from the update rule.

RMSProp Analogies

RMSProp and Adadelta have both been developed **independently around the same time** stemming from the need to resolve Adagrad's radically diminishing learning rates. RMSProp in fact **is identical to the first update vector of Adadelta** that we derived above:

$$\begin{aligned}\mathbb{E}[g_j^2]^{(t)} &= \gamma \mathbb{E}[g_j^2]^{(t-1)} + (1 - \gamma)g_j^{2(t)} \\ \theta_j^{(t+1)} &= \theta_j^{(t)} - \frac{\alpha}{\sqrt{\mathbb{E}[g_j^2]^{(t)} + \epsilon}} g_j^{(t)}\end{aligned}$$

Moreover, notice that basically RMSProp also divides the learning rate by an exponentially decaying average of squared gradients.

Adam

Adaptive Moment Estimation (Adam), proposed by Diederick et. al. in 2014 [11], is an adaptive learning rate optimization algorithm which computes adaptive learning rates for each parameter by combining ideas from both RMSProp (or first update of Adadelta) and SGD with momentum.

In fact, in addition to storing an exponentially decaying average of past squared gradients $v^{(t)}$ like RMSProp, Adam also keeps an exponentially decaying average of past gradients $m^{(t)}$, similar to momentum:

$$\begin{aligned}m_j^{(t)} &= \beta_1 m_j^{(t-1)} + (1 - \beta_1)g_j^{(t)} \\ v_j^{(t)} &= \beta_2 v_j^{(t-1)} + (1 - \beta_2)g_j^{2(t)}\end{aligned}$$

$m^{(t)}$ and $v^{(t)}$ are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method (we will see soon better why we talk about uncentered variance).

So, Adam stores two moving averages for each parameter during training, $m_i^{(t)}$ and $v_i^{(t)}$. As both are initialized as vectors of 0s, **the authors of Adam observed that they were biased towards zero** in the early stages of training (during the initial time steps), especially when the decay rates β_1 and β_2 were close to 1. Therefore, they counteracted these biases by computing **bias-corrected first and second moment estimates**:

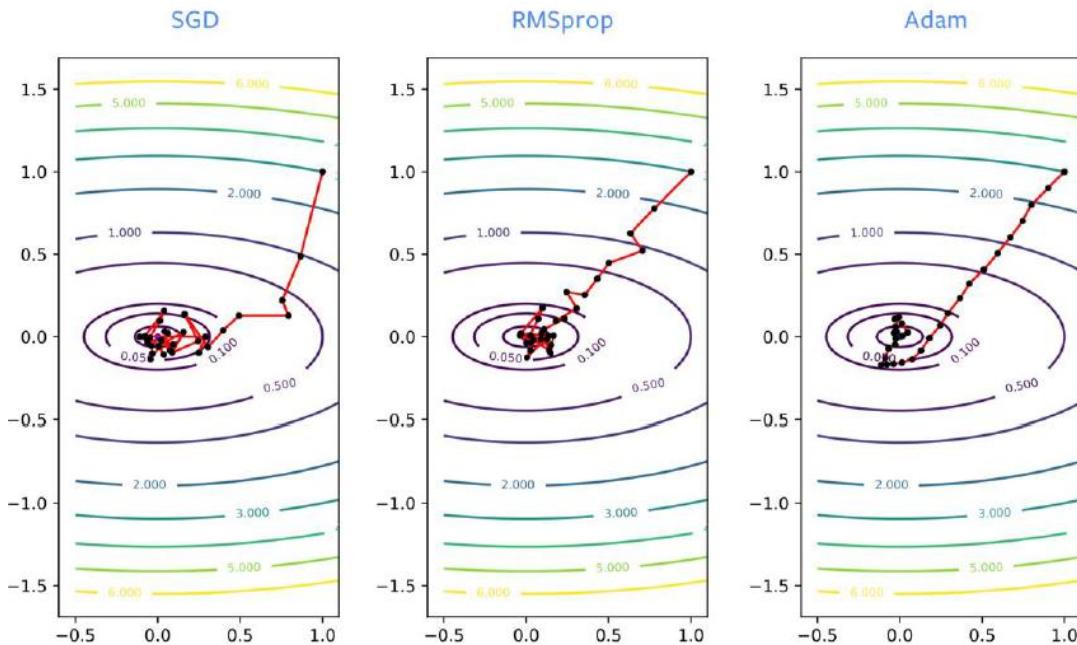
$$\hat{m}_j^{(t)} = \frac{m_j^{(t)}}{1 - \beta_1^t}$$

$$\hat{v}_j^{(t)} = \frac{v_j^{(t)}}{1 - \beta_2^t}$$

They then used these to update the parameters just as we have seen in RMSprop (or in Adadelta), which yields the Adam update rule:

$$\theta_j^{(t+1)} = \theta_j^{(t)} - \frac{\alpha}{\sqrt{\hat{v}_j^{(t)}} + \epsilon} \hat{m}_j^{(t)}$$

The authors propose default values of 0.9 for β_1 , 0.999 for β_2 and 10^{-8} for ϵ . They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms.



A closer look to Adam

1. Why is Bias Correction Necessary?

We said above that Adam keeps track of two exponential moving averages, $m_i^{(t)}$ and $v_i^{(t)}$. Since these moving averages are initialized at zero, they are **biased towards zero** in the early training steps. To correct this, Adam applies **bias corrections**, which effectively rescale these estimates before using them for parameter updates.

To understand why this rescaling is needed, consider what happens at first time step $t = 1$, where both $m_i^{(0)}$ and $v_i^{(0)}$ start at 0. Supposing to have $\beta_1 = 0.9$ and $\beta_2 = 0.999$, then:

$$m_j^{(1)} = \beta_1 m_j^{(0)} + (1 - \beta_1) g_j^{(1)} = 0.9 \cdot 0 + 0.1 \cdot g_j^{(1)} = 0.1 \cdot g_j^{(1)}$$

$$v_j^{(1)} = \beta_2 v_j^{(0)} + (1 - \beta_2) g_j^{2(1)} = 0.999 \cdot 0 + 0.001 \cdot g_j^{2(1)} = 0.001 \cdot g_j^{2(1)}$$

Ideally, the first moment estimate at this step should be equal to $g_j^{(1)}$, but instead, it is scaled down by 0.1. The same happens for the second moment estimate that at this step should be equal $g_j^{2(1)}$, but instead, it is scaled down by 0.001. Using bias-corrected versions, we remove these biases, scaling them by $(1 - 0.9) = 0.1$ and $(1 - 0.999) = 0.001$, ensuring that after the first step:

$$\hat{m}_j^{(1)} = \frac{m_j^{(1)}}{1 - \beta_1^1} = \frac{0.1 \cdot g_j^{(1)}}{1 - 0.9^1} = \frac{0.1 \cdot g_j^{(1)}}{0.1} = g_j^{(1)}$$

$$\hat{v}_j^{(1)} = \frac{v_j^{(1)}}{1 - \beta_2^1} = \frac{0.001 \cdot g_j^{2(1)}}{1 - 0.999^1} = \frac{0.001 \cdot g_j^{2(1)}}{0.001} = g_j^{2(1)}$$

Moreover, notice that the correction terms involve **powers** β_1^t and β_2^t . Early on, when t is small, these powers are significant and the correction is large. As t increases, $\beta_1^t, \beta_2^t \rightarrow 0$, so the correction factor tends to 1. This actually ensures that bias corrections matters only in the initial steps — the very purpose they were designed for.

2. Interpreting the Second Moment (Uncentered Variance $v^{(t)}$)

Let's recall that the variance of a random variable X is defined as:

$$\text{Var}(X) = \mathbb{E}[X^2] - \mathbb{E}[X]^2$$

where $\mathbb{E}[\cdot]$ denotes the expectation.

In Adam, the second moment estimate $v^{(t)}$ is called the **uncentered variance** because it just tracks the exponential moving average of the square of the gradients ($\mathbb{E}[g^2]$) without subtracting square of the mean of the gradients ($\mathbb{E}[g]^2$).

Note: This terminology is not entirely accurate. In practice, what Adam actually tracks is a **running average**, which is an approximation rather than the **true expectation**. The mathematical expectation involves averaging over an **infinite** number of samples, whereas the moving average is an estimate that converges over time. That is also why $m^{(t)}$ and $v^{(t)}$ are called **estimates**, since they are computed using finite past gradients rather than an infinite distribution.

3. How Adam Adapts Step Sizes

Let's recall that the variance quantifies how much the gradients vary around their means. Adam dynamically adjusts step sizes based on gradient variance:

- **Stable gradients (small variance):** If gradients remain approximately constant (i.e., the loss landscape is smooth, like walking down a gentle slope), the variance of the gradients is approximately 0 and the uncentered variance $v^{(t)}$ is therefore approximately equal to $m_j^{2(t)}$ ($v_j^{(t)} \approx m_j^{2(t)}$). This means that in the update rule $m_j^{(t)} / \sqrt{v_j^{(t)}}$ is around 1 and therefore the step size is approximately equal to the learning rate α , allowing normal gradient descent behavior.

- **Rapidly changing gradients (high variance):** On the other hand, if gradients are changing rapidly then $\sqrt{v_j^{(t)}}$ is much larger than $m_j^{(t)}$, and therefore the step size is much smaller than α (preventing the optimizer from making overly large updates in unstable regions).

Summing up, this means that Adam is able to adapt step sizes for each individual parameter from estimating the first and second moments of the gradients: when the gradients do not change much and “we do not have to be careful walking down the hill”, the step size is of the order of α , if they do and “we need to be careful not to walking the wrong direction”, the step size is much smaller.

Limitations of Adam

Adam works well, but it has some drawbacks:

1. **Generalization Issue:** Despite its strong performance during training, Adam does not always generalize well to unseen data. It has been observed that neural networks trained with Adam can achieve near-zero loss on the training set but still perform worse on test data compared to models trained with SGD. Therefore, it has shown to produce **higher generalization errors** compared to SGD, and this is particularly evident on image problems. Factors could include that it finds the closest local minimum, or less noise in ADAM, or its structure, for instance.
2. **Increased Memory Usage:** Adam requires maintaining **three buffers** (for first-moment estimates, second-moment estimates and parameters), whereas SGD only needs one. While this is not an issue for small models, it can become a constraint when training a model on the order of several giga bytes in size, in which case it might not fit in memory.
3. **More Hyperparameters to Tune:** Adam introduces two momentum-related hyperparameters (β_1 and β_2), whereas SGD with momentum requires tuning only one. This makes Adam more complex to fine-tune for optimal performance.

AdamW

Many deep learning libraries implement **L2 regularization** by adding a weight decay term directly to the gradients. However, this approach, while effective for **SGD**, does not work as intended for adaptive optimizers like Adam. In fact, applying *L2* as described above to the standard Adam update, we'll get:

$$\theta_j^{(t+1)} = \theta_j^{(t)} - \alpha \frac{\beta_1 m_j^{(t-1)} + (1 - \beta_1)(\nabla f^{(t)}(\theta_j^{(t)}) + \lambda \theta_j^{(t)})}{\sqrt{v_j^{(t)}} + \epsilon}$$

where λ is the regularization coefficient which defines the rate of the weight decay and $\nabla f^{(t)}(\theta_j^{(t)})$ is the gradient of the loss with respect to the parameters at step t (i.e. it is equivalent to the notation $g_j^{(t)}$).

Notice that, basically we did:

$$\begin{aligned} g_j^{(t)} &= \nabla f^{(t)}(\theta_j^{(t)}) + \lambda \theta_j^{(t)} \\ m_j^{(t)} &= \beta_1 m_j^{(t-1)} + (1 - \beta_1)g_j^{(t)} \\ \theta_j^{(t+1)} &= \theta_j^{(t)} - \alpha \frac{m_j^{(t)}}{\sqrt{v_j^{(t)}} + \epsilon} \end{aligned}$$

Note: Here just for simplify notation we neglected the bias-corrections.

As you can see in standard Adam **the weight decay is normalized by \sqrt{v} as well**. If the gradient of a certain weight is large (or is changing a lot), the corresponding v will be large too. As result, the weight will be regularized less than weights with small and slowly changing gradients!

This unintended scaling makes L2 regularization **less effective in Adam** compared to SGD, which explains why SGD often yields models with **better generalization**.

To address this issue, **AdamW**, a modified version of Adam proposed by Loshchilov et al. in 2017 [12], **decouples weight decay from the adaptive learning rate**.

In AdamW, instead of applying weight decay within the gradient update, weight decay is applied **after** computing the parameter-wise step size (see the green term in line 12). The weight decay (or regularization term) does not end up in the moving averages and is thus only proportional to the weight itself. This results in a more effective and consistent regularization effect, leading to better generalization performance.

Algorithm 2 Adam with L₂ regularization and Adam with decoupled weight decay (AdamW)

```

1: given  $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$ 
2: initialize time step  $t \leftarrow 0$ , parameter vector  $\theta_{t=0} \in \mathbb{R}^n$ , first moment vector  $m_{t=0} \leftarrow \theta$ , second moment
   vector  $v_{t=0} \leftarrow \theta$ , schedule multiplier  $\eta_{t=0} \in \mathbb{R}$ 
3: repeat
4:    $t \leftarrow t + 1$ 
5:    $\nabla f_t(\theta_{t-1}) \leftarrow \text{SelectBatch}(\theta_{t-1})$                                  $\triangleright$  select batch and return the corresponding gradient
6:    $\mathbf{g}_t \leftarrow \nabla f_t(\theta_{t-1}) \cancel{+ \lambda \theta_{t-1}}$ 
7:    $\mathbf{m}_t \leftarrow \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1) \mathbf{g}_t$                                  $\triangleright$  here and below all operations are element-wise
8:    $\mathbf{v}_t \leftarrow \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2$ 
9:    $\hat{\mathbf{m}}_t \leftarrow \mathbf{m}_t / (1 - \beta_1^t)$                                           $\triangleright \beta_1$  is taken to the power of  $t$ 
10:   $\hat{\mathbf{v}}_t \leftarrow \mathbf{v}_t / (1 - \beta_2^t)$                                           $\triangleright \beta_2$  is taken to the power of  $t$ 
11:   $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$                                  $\triangleright$  can be fixed, decay, or also be used for warm restarts
12:   $\theta_t \leftarrow \theta_{t-1} - \eta_t \left( \alpha \hat{\mathbf{m}}_t / (\sqrt{\hat{\mathbf{v}}_t} + \epsilon) \cancel{+ \lambda \theta_{t-1}} \right)$ 
13: until stopping criterion is met
14: return optimized parameters  $\theta_t$ 

```

Note: The term $\eta^{(t)}$ is just a **scaling factor** introduced in AdamW to also account for a possible scheduling of both the learning rate α and the regularization coefficient λ .

Lion

The **Lion optimizer**, introduced by Google Brain researchers [13] on 13 February 2023, is a new optimization algorithm **discovered via a symbolic / program search** approach, rather than handcrafted design. Its name stands for “*EvoLved Sign Momentum*” because it blends momentum (as in methods like SGD with Momentum) with a *sign* operation ([below we’ll see this more in detail](#)).

To efficiently explore the large space of possible optimizers during the program search, the authors employed **regularized evolution**, a type of evolutionary algorithm. In this approach, a population of P candidate algorithms is maintained and gradually improved through cycles. In each cycle:

1. a subset $T < P$ of candidates is sampled at random and the best performer is selected as the parent (tournament selection);

2. This parent is then copied and mutated (apply mutation) to produce a child algorithm, which is added to the population, while the oldest algorithm is removed.

While evolutionary search usually starts from random candidates, the authors **warm-started** the initial population using **AdamW**, which accelerated convergence and guided the search toward promising regions.

Additional **efficient search techniques**, such as symbolic representation of update rules and compact program functions (e.g., **linear interpolation** for combining momentum and gradient), allowed the algorithm search to be both tractable and interpretable.

Lion only tracks momentum and, unlike other adaptive optimizers, updates are taken in the **direction of the momentum's sign** (element-wise). This means the *magnitude* of the update is the same for each parameter, and only the *direction* matters. Pseudocode:

Algorithm 1 AdamW Optimizer

```

given  $\beta_1, \beta_2, \epsilon, \lambda, \eta, f$ 
initialize  $\theta_0, m_0 \leftarrow 0, v_0 \leftarrow 0, t \leftarrow 0$ 
while  $\theta_t$  not converged do
     $t \leftarrow t + 1$ 
     $g_t \leftarrow \nabla_{\theta} f(\theta_{t-1})$ 
    update EMA of  $g_t$  and  $g_t^2$ 
     $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ 
     $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ 
    bias correction
     $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ 
     $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ 
    update model parameters
     $\theta_t \leftarrow \theta_{t-1} - \eta_t (\hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon) + \lambda \theta_{t-1})$ 
end while
return  $\theta_t$ 

```

Algorithm 2 Lion Optimizer (ours)

```

given  $\beta_1, \beta_2, \lambda, \eta, f$ 
initialize  $\theta_0, m_0 \leftarrow 0$ 
while  $\theta_t$  not converged do
     $g_t \leftarrow \nabla_{\theta} f(\theta_{t-1})$ 
    update model parameters
     $c_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ 
     $\theta_t \leftarrow \theta_{t-1} - \eta_t (\text{sign}(c_t) + \lambda \theta_{t-1})$ 
    update EMA of  $g_t$ 
     $m_t \leftarrow \beta_2 m_{t-1} + (1 - \beta_2) g_t$ 
end while
return  $\theta_t$ 

```

Given that it only keeps track of the momentum (first moment), it is **more memory-efficient** than Adam (which also keeps the second moment). In terms of performance, it has been found that Lion's effectiveness increases with larger **batch sizes** during training. Another noteworthy characteristic is that Lion generally requires a smaller **learning rate** compared to Adam. This is a consequence of the choice to use the sign function due to the fact that the **norm** of the update produced by the **sign function** is inherently larger. For more detail, refer to the paper.

A Brief Overview of "Hessian-Free" Optimization

Hessian-Free optimizers [14] are often used to train complex models, such as Recurrent Neural Networks (RNNs). They involve intricate mathematical methods, so here we will just focus on building an intuitive understanding without delving too deeply into the complex math behind it. This section is based on a [lecture](#) given by G. Hinton at University of Toronto.

Intuition Behind Curvature

When training the parameters of a neural network, our goal is to navigate as far down the error surface as possible. A natural question arises: "If we choose a direction to move in and we keep going in that direction, how much does the error decrease before it starts rising again?". To answer this, we need to understand how the error behaves with respect to the curvature of the surface and the gradient.

For now, let's simplify the situation by assuming the curvature of the error surface is constant (i.e., it's a quadratic error surface). We can also assume that the magnitude of the gradient decreases as we move down the gradient, meaning the error surface is convex upward.

In this scenario, the **maximum reduction in error** when moving along a particular direction depends heavily on the **ratio of the gradient to curvature**. Therefore, a good direction to move in is one that offers a **high ratio of gradient to curvature**, even if the gradient itself is small.

To better illustrate this, imagine a steep gradient with a relatively high curvature. In this case, although the gradient is large, the **curvature** is also large, which means the error may only decrease a little, and we may end up overshooting the minimum.

Now consider a scenario with a gentler gradient, but the curvature in this region is much lower. Despite the smaller gradient, the **ratio of gradient to curvature** is higher, which means that by moving in this direction, the error decreases more significantly, and we move toward the minimum more effectively.

Now, the question is: "How can we identify such directions (like the second example), where even if the gradient is small, the curvature is even smaller?".

Let's start by looking at a well-known approach, the Newton's method.

Newton's method

Newton-Raphson method, also simply known as **Newton's method**, helps us to address a fundamental issue in optimization with steepest descent: using just the gradient doesn't always lead us in the optimal direction, especially when the curvature varies.

We know that if the error surface is quadratic and has circular cross-sections, the gradient direction is actually fine. In fact, it points directly toward the minimum, so gradient descent works well in this case. However, in real-world error surfaces, the curvature is often not uniform; the surface may be elongated in certain directions (forming ellipses), and the gradient points in the wrong direction.

The idea behind Newton's method is to apply a **transformation** that turns ellipses into circles. This adjustment will allow us to take steps as if we were navigating a circular error surface.

To achieve this, Newton's method multiplies the gradient vector by the inverse of the **curvature matrix H** (also called the **Hessian matrix**):

$$\Delta\theta^{(t)} = -\alpha H(\theta^{(t)})^{-1} \frac{\partial E^{(t)}}{\partial \theta}$$

So, the steps we need to take are calculate the Hessian, take its reverse, multiply it by the gradient and then move some distance in that direction. If the error surface is truly quadratic and we select the learning rate α correctly, we'll reach the minimum in a single step.

Of course, that single step involves something complicated which is inverting that Hessian matrix.

Unfortunately, the problem with this is that even if we have only a million parameters in our neural network, the curvature (Hessian) matrix will have trillion terms, so it is totally infeasible to invert it.

Curvature (Hessian) Matrix

The **Hessian** is the matrix of the second-order partial derivatives of the loss function (scalar field) with respect to the parameters of a neural network. In essence, it captures the **curvature** of the error surface. That's a good strike

for us since the more we know about the loss function, the better our optimization methods can be at navigating it. In mathematical terms, the elements of the Hessian matrix can be written as:

$$H_{ij} = \frac{\partial^2 E}{\partial \theta_i \partial \theta_j} = \frac{\partial}{\partial \theta_j} \left(\frac{\partial E}{\partial \theta_i} \right)$$

This represents how the gradient of the error function with respect to one parameter ($\frac{\partial E}{\partial \theta_i}$) changes as we move in the direction of another parameter (θ_j).

The curvature (Hessian) matrix consists of both **diagonal** and **off-diagonal** elements:

- **Diagonal elements:** These terms represent how the gradient changes in the direction of each parameter as we adjust that specific parameter. For instance, H_{ii} indicates how the gradient with respect to θ_i changes as we adjust θ_i . These terms give us information about how sensitive the error surface is along each individual parameter direction.
- **Off-diagonal elements:** These terms show how the gradient in one direction changes as we move in a different direction. For instance, H_{ij} (where $i \neq j$) tells us how the gradient with respect to θ_i changes as we modify θ_j . This captures the **interactions** between parameters — how the movement in one direction affects the gradient in another direction.

	i	j	k
i	$\frac{\partial(\partial E / \partial w_i)}{\partial w_j}$		
j	$\frac{\partial(\partial E / \partial w_j)}{\partial w_i}$		
k			$\frac{\partial^2 E}{\partial w_k^2}$

If the error surface is a circular bowl, the off-diagonal terms will be zero, indicating that the gradient in one direction does not change when moving in another direction. In such cases, the gradient descent method works well because each step along a parameter direction leads directly toward the minimum. However, in a more **elliptical** error surface, the **off-diagonal elements are non-zero**. In this case, if you move in one direction, the gradient in another direction changes. This is what we call a **twist** in the error surface — when traveling in one direction, the gradient in another direction is affected.

So, what's going wrong with the steepest descent when you have an elliptical error surface? It is that if we travel in one direction the gradient in another direction changes and so if I update one of the parameters, at the same time I'm updating all the other parameters. All those other updates will cause a change in the gradient for the first parameter and that means when I update it, I may make things worse, i.e. the gradient may have actually reversed the sign due to the changes in all other parameters. As result, as we get more and more parameters we need to be more and more cautious about changing each one of them because the simultaneous changes to all the other parameters can change and mess up the gradient of a parameter. In simpler words, the curvature matrix determines the sizes of these interactions, i.e. how much each parameter's update influences the gradients of all the other parameters.

How to avoid Inverting a Huge Matrix

So, we must deal with curvature, we can't just ignore it, and we'd like to deal with it without actually inverting it because the curvature matrix has too many terms to be of use in a big neural network.

One thing we could do is get some benefit from just using the terms along the **leading diagonal** of the curvature matrix (Le Cun) and making our step size depend on that leading diagonal. That helps a bit, it will get us to make different step sizes for different parameters. However, the diagonal terms are only a tiny fraction of the interactions (they are the self-interactions), so we are ignoring most of the terms in the curvature matrix when we do that.

Another thing we could do is try to approximate the curvature matrix with a matrix of much lower rank that captures the main aspects of the curvature matrix. That's what is done in Hessian-free methods, LBFGS and many other methods that try to do an approximate second order method for minimizing the error.

In the HF methods, **we make an approximation to the curvature matrix and then, assuming that approximation is correct** (i.e. assume the error surface is quadratic), **we minimize the error using an efficient technique called Conjugate Gradient**. Then, once we get close to the minimum on this approximation of the curvature, we make another approximation to the curvature matrix and we use conjugate gradient to minimize again.

Note: For RNNs it's important to add a penalty for changing any of the hidden activities too much. That would prevent us from changing a parameter early on that causes huge effects later in the sequence. We don't want to get effects that are too big and if we look at the changes in the hidden activities we can prevent them by penalizing those changes. If we put a quadratic penalty on those changes, we can combine them with the rest of Hessian-free methods.

Conjugate Gradient

Conjugate Gradient (CG) is a very clever method that, instead of going straight to the minimum in one step by multiplying by the inverse of the curvature matrix like Newton's method, tries to minimize one direction at the time.

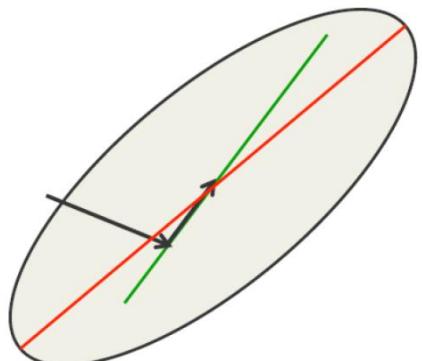
The process begins by selecting the direction of the steepest descent and searches for the minimum along this direction, which may require re-evaluating the error and the gradient multiple times. Once the minimum along this first direction is found, a second new direction is chosen to be conjugate to the previous one. **“Conjugate” means that as you go in the new direction, you do not change the gradients in the previous directions.**

It's a funny idea since we can see the conjugate direction as the one you can go in in a sense that it doesn't have a twist, i.e. you go in that direction and the gradient in the first direction doesn't change.

Therefore, with conjugate gradient method we:

- Use a sequence of steps each of which finds the minimum along one direction.
- **Make sure that each new direction is “conjugate” to the previous directions** so you do not mess up the minimization you already did.

Consider the elliptical error surface in Figure on side, where the red line represents its major axis. Our goal is to reach the minimum efficiently. We begin by taking a step in the direction of the steepest descent (first arrow) and, as we minimize in this first direction, we move slightly across the bottom of the ellipse until reaching a local minimum.



An interesting property emerges: the gradient in the direction of the first step (first arrow) is zero at all points on the green line. Therefore, if we move along the green line (second arrow) we don't mess up the minimization we already did in the first direction. So, we move along this second direction until reaching a local minimum.

In this two-dimensional space since we are at the minimum in the first direction, and simultaneously we are at the minimum in the second direction, while still maintaining the previously achieved minimum, means we actually reach the global minimum.

Extending this idea to higher dimensions, we continue selecting new directions that are **conjugate** to previous ones — ensuring that each new step optimizes without interfering with past minimizations. By iterating this

process across as many directions as there are dimensions in our space, we ultimately reach the global minimum of the error surface.

Conjugate gradient is guaranteed to find the global minimum of an N-dimensional **quadratic** surface after **only N steps** (it is very efficient). Why? It does so because it manages to get the gradient to be zero in N different direction, that they are not orthogonal directions, but they are independent one of another and that is sufficient to be at the global minimum. More importantly, after **many less than N steps** (always in a quadratic surface) it has typically got the error very close to the minimum value and that's why we use it → we are not going to do the full steps that would be as expensive as inverting the whole matrix, we're going to do with many less steps and get quite close to the minimum.

Conjugate gradient **can be applied directly to a non-quadratic error surface** (like the one of a multilayer nonlinear neural network) **and it usually works quite well → non-linear conjugate gradient**

Conjugate gradient is essentially a batch method, but you can apply it with large mini-batches and, when you do that, you do many steps of conjugate gradient on the same large mini-batch and then you move on the next large mini-batch.

The HF optimizer uses conjugate gradient for minimization on a genuinely quadratic surface where it excels (it works much better for that for a non-linear surface). This genuinely quadratic surface that HF is using is the quadratic approximation to the true surface.

Normalization Layers

In the previous chapter, we've seen the importance of **normalizing input data** — standardizing features to have zero mean and unit variance helps networks converge faster and more reliably. The same principle can be extended beyond the input data, applying it also to the variables in each hidden layer of a deep neural network.

As neural networks get deeper, hidden activations across layers can vary widely in scale, making optimization difficult and leading to unstable gradients. By normalizing the activations within each hidden layer, we can help **Maintain a more stable distribution of values** as they propagate through the network. This improves gradient flow, accelerates convergence and mitigates problems such as vanishing or exploding gradients, which are common in very deep networks.

Unlike input normalization, which is done once before training, **hidden activation normalization** must happen **dynamically during training**, since the distribution of activations changes as weights are updated at every iteration. This is where **normalization layers** come in: they automatically normalize intermediate values during forward passes, helping to stabilize training.

Batch Normalization

Batch Normalization (BatchNorm), introduced by Ioffe and Szegedy in 2015 [15], was the first widely adopted normalization layer. It operates by normalizing the **pre-activations** of each layer, using the empirical mean μ_j and variance σ_j^2 computed independently for each feature dimension **across the mini-batch**. Specifically, given a mini-batch of pre-activations $B = \{z^{(1)}, \dots, z^{(N)}\}$ (with N being the size of the mini-batch), the batch normalization for a specific feature $z_j^{(i)}$ proceeds as follows:

1. **Compute mini-batch mean:**

$$\mu_j = \frac{1}{N} \sum_{i=1}^N z_j^{(i)}$$

2. **Compute mini-batch variance:**

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (z_j^{(i)} - \mu_j)^2$$

3. **Normalize:**

$$\hat{z}_j^{(i)} = \frac{z_j^{(i)} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

where ϵ is a small constant (e.g., 10^{-5}) added for numerical stability.

4. **Scale and Shift:**

$$\check{z}_j^{(i)} = \gamma \hat{z}_j^{(i)} + \beta$$

where γ (scale) and β (shift) are **learnable parameters** specific to **each feature dimension**. These parameters allow the network to learn the optimal scale and mean for the normalized pre-activations, restoring the **representational flexibility** that could be lost by strict normalization to zero mean and unit variance. They are learned through backpropagation alongside other network parameters.

Does Scale and Shift Undo Normalization?

At first glance, it might seem like the learnable scale and shift (γ, β) **undoes** the effects of normalization. After all, if the model can simply rescale and re-center the pre-activations, isn't it undoing what normalization just did? **Not quite**, the crucial difference is in the way the parameters evolve during training. In the original unnormalized network, the mean and variance of activations are **emergent properties** — they depend on a complex, entangled function of the layer's weights and biases. These statistical properties change unpredictably during training, making it **difficult to control or optimize them directly**.

In contrast, with BatchNorm the **mean and variance are explicitly normalized** and **any necessary representational flexibility is reintroduced through just the learnable parameters γ and β** , which turn out to be **much easier to learn** during gradient descent. So, these parameters allow the model to adapt the scale and offset of pre-activations **as needed**, while still benefiting from the stability and improved optimization dynamics provided by normalization. Put simply:

- Without BatchNorm: Pre-activations might have arbitrary and unstable distributions, which are hard to train.
- With BatchNorm: Distributions are standardized first, then “fine-tuned” through γ and β , which are much easy to learn via GD.

BatchNorm During Inference

Once the network is trained and we want to make predictions on new data, we face a problem: during inference, we typically process **one sample at a time**, not full mini-batches. But Batch Normalization relies on **batch statistics** — the mean and variance of pre-activations across a mini-batch — which are no longer available during inference. To solve this, we could in principle compute the true mean μ_j and true variance σ_j^2 for each layer by passing the entire training set through the network after training. However, this would be **computationally expensive** and impractical for large datasets. Instead, BatchNorm solves this by **maintaining exponentially moving averages (EMA) of the batch statistics during training**:

$$\bar{\mu}_j^{(t)} = \eta \bar{\mu}_j^{(t-1)} + (1 - \eta) \mu_j$$

$$\bar{\sigma}_j^{(t)} = \eta \bar{\sigma}_j^{(t-1)} + (1 - \eta) \sigma_j$$

Here $\bar{\mu}_j$ and $\bar{\sigma}_j$ indicate exponentially moving averages and $\eta \in [0, 1]$ is the decay rate (commonly close to 1, e.g., 0.99). These moving averages are **NOT used during training**, but they are updated at each step and saved as part of the layer's state. During inference, the learned parameters γ, β and the stored moving averages $\bar{\mu}_j, \bar{\sigma}_j$ are used to compute:

$$\hat{z}_j = \frac{z_j - \bar{\mu}_j}{\sqrt{\bar{\sigma}_j^2 - \epsilon}} \quad \text{then} \quad \tilde{z}_j = \gamma \hat{z}_j + \beta$$

Why Is Batch Normalization Effective?

BatchNorm has proven to be one of the most impactful innovations in DL, enabling faster training, better convergence and improved generalization. However, **why it works** so well remains an open research question. Batch normalization was originally motivated by noting that updates to weights in earlier layers of the network change the distribution of values seen by later layers, a phenomenon called **internal covariate shift**. However, later studies [16] suggest that **covariate shift** is not a significant factor and that the improved training results from

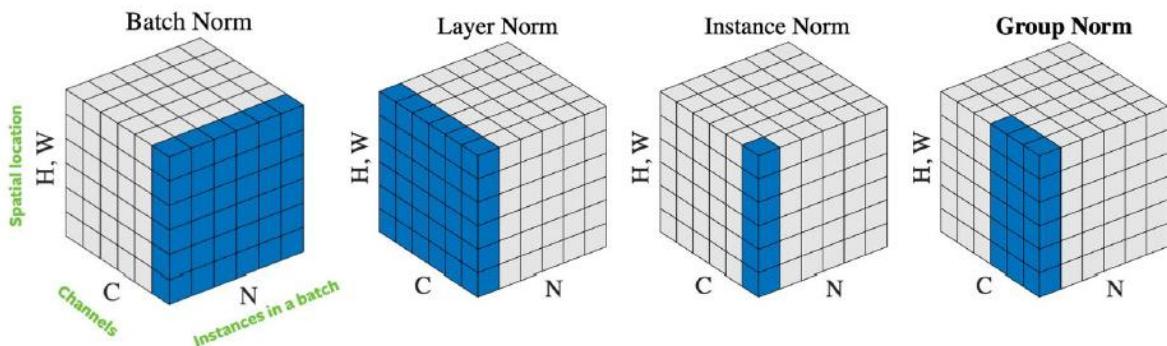
an improvement in the **smoothness of the error function landscape**, making gradients more predictable and stable. Nevertheless, normalization clearly provides several practical benefits:

- **Accelerated training:** Networks with normalization layers are **easier to optimize**, allowing for the use of **larger learning rates**, therefore speeding up the training of neural networks.
- **Implicit regularization:** The **mean/std estimates** used in BatchNorm **are noisy** due to the randomness of the samples in the batch. This extra “noise” acts as a form of **regularization**, improving generalization.
- **Reduced sensitivity to weight initialization:** Networks become less fragile to the scale of initial weights.

As a result, normalization lets you be more “careless” – you can combine almost any neural network building blocks together and have a good chance of training it without having to consider how poorly conditioned it might be.

Limitations of BatchNorm

One of the main limitations of Batch Normalization is its strong dependence on mini-batch size. When the batch size is too small (e.g., fewer than 8–16 examples), the computed mean and variance become too noisy. Also, for very large training sets, the minibatches may be split across different GPUs, making global normalization across the mini-batch inefficient. These limitations have motivated the development of alternative normalization techniques that are **independent of the batch size**.



Layer Normalization

Layer Normalization (LayerNorm) is a technique that normalizes the pre-activations **across the features** (i.e., hidden units) of a **single data point**, rather than across the mini-batch as in BatchNorm. It was introduced by Hinton et al. (2016) [17] in the context of RNNs where the distributions change after each time step making batch normalization infeasible. Since then, LayerNorm has proven highly effective in other architectures as well — most notably in Transformers, where it is now a standard component.

Given a single input pre-activation vector for one instance $z^{(i)} = [z_1^{(i)}, \dots, z_D^{(i)}]$ with D features (hidden units), LayerNorm:

- **Compute layer mean** (mean over all layer’s features) **for instance**:

$$\mu^{(i)} = \frac{1}{D} \sum_{j=1}^D z_j^{(i)}$$

- **Compute layer variance for instance**:

$$(\sigma^{(i)})^2 = \frac{1}{D} \sum_{j=1}^D (z_j^{(i)} - \mu^{(i)})^2$$

- **Normalize:**

$$\hat{z}_j^{(i)} = \frac{z_j^{(i)} - \mu^{(i)}}{\sqrt{(\sigma^{(i)})^2 + \epsilon}}$$

- **Scale and Shift:**

$$\check{z}_j^{(i)} = \gamma \hat{z}_j^{(i)} + \beta$$

Again, γ and β are learnable scale and shift parameters.

Key Differences from BatchNorm

- **Normalization scope:**

- **BatchNorm** normalizes **across the batch** for each feature.
- **LayerNorm** normalizes **across features** for each **individual instance**.

- **Applicability:**

- LayerNorm depends only on the current input instance's features, making it effective even with mini-batch size $N = 1$. This is a major advantage for RNNs and Transformers, where time steps or sequence lengths can vary making consistent batching difficult, and in situations where memory limits force small batch sizes.

- **Consistency in training and inference:**

- Unlike BatchNorm, **LayerNorm does not rely on batch statistics, so the same computation is used during both training and inference.** There's no need to store moving averages of the mean and variance.

LayerNorm in CNNs

In RNNs and Transformers, the input is typically a 3D tensor of shape:

$$(batch\ size, sequence\ length/n^{\circ}\ time\ steps, feature\ dimension) \ or \ (N, T, D)$$

LayerNorm is applied across the feature dimension D — i.e., across the hidden units — for **each individual instance** and for each token (or time step). However, in **CNNs**, the input is usually a **4D tensor** of shape:

$$(batch\ size, channels, height, width) \ or \ (N, C, H, W)$$

Here, each instance contains **C feature maps**, each with spatial dimensions $H \times W$. Therefore, **LayerNorm in CNNs** is typically applied **across all channels and spatial locations per individual instance**. That is, for each input $z^{(i)} \in \mathbb{R}^{C \times H \times W}$ corresponding to instance i , we have:

$$\begin{aligned} \mu^{(i)} &= \frac{1}{CHW} \sum_{c=1}^C \sum_{h=1}^H \sum_{w=1}^W z_{c,h,w}^{(i)} \\ (\sigma^{(i)})^2 &= \frac{1}{CHW} \sum_{c=1}^C \sum_{h=1}^H \sum_{w=1}^W (z_{c,h,w}^{(i)} - \mu^{(i)})^2 \end{aligned}$$

While this usage works, LayerNorm is less common in CNNs compared to GroupNorm or InstanceNorm, which we detail below.

Instance Normalization

Instance Normalization (InstanceNorm) [18] can be seen as applying LayerNorm per-channel in convolutional layers. It normalizes across spatial dimensions (height and width) independently for **each channel** and **each instance** in the batch. It was introduced in the context of **image style transfer**, where the style of an image is affected by the **contrast statistics** of feature maps. The authors observed that the **mean and variance of each channel** influence the style of the generated image. By normalizing each channel independently (removing instance-specific contrast), and later applying style-specific statistics (i.e., "denormalization"), the model can effectively transfer styles between images.

Formally, for each input $z \in \mathbb{R}^{C \times H \times W}$ corresponding to instance i , the normalization is applied **channel-wise** as follows:

$$\begin{aligned}\mu_c^{(i)} &= \frac{1}{HW} \sum_{h=1}^H \sum_{w=1}^W z_{h,w}^{(i)} \\ (\sigma_c^{(i)})^2 &= \frac{1}{HW} \sum_{h=1}^H \sum_{w=1}^W (z_{h,w}^{(i)} - \mu_c^{(i)})^2 \\ \hat{z}_{c,h,w}^{(i)} &= \frac{z_{c,h,w}^{(i)} - \mu_c^{(i)}}{\sqrt{(\sigma_c^{(i)})^2 + \epsilon}} \\ \check{z}_{c,h,w}^{(i)} &= \gamma_c \hat{z}_{c,h,w}^{(i)} + \beta_c\end{aligned}$$

Here, γ_c and β_c are learnable parameters **per-channel** (in LayerNorm they were across all channels).

Group Normalization

Group Normalization (GroupNorm) [19] is a compromise between LayerNorm and InstanceNorm. It divides the channels into **groups of fixed size** and performs normalization across spatial dimensions and **channels within each group, for each instance**. It was introduced to overcome the limitations of BatchNorm in scenarios where **small batch sizes** are unavoidable — a common situation in computer vision tasks such as object detection, segmentation, and video processing, where **memory constraints often restrict batch size**.

Formally, for a pre-activation $z^{(i)} \in \mathbb{R}^{C \times H \times W}$ (corresponding to instance i) GroupNorm proceed as follows:

1. Divide Channels into Groups:

Let the number of groups be G . Each group contains C/G consecutive channels.

Note: G must be chosen so that C is divisible by G .

2. Compute Mean and Variance per Group:

For each group $g \in \{1, \dots, G\}$, define the set of channels C_g belonging to group g . Then compute:

$$\begin{aligned}\mu_g^{(i)} &= \frac{1}{(C/G)HW} \sum_{c \in C_g} \sum_{h=1}^H \sum_{w=1}^W z_{c,h,w}^{(i)} \\ (\sigma_g^{(i)})^2 &= \frac{1}{(C/G)HW} \sum_{c \in C_g} \sum_{h=1}^H \sum_{w=1}^W (z_{c,h,w}^{(i)} - \mu_g^{(i)})^2\end{aligned}$$

3. Normalize, Then Scale and Shift:

Each element in the group is normalized and then transformed using learnable parameters γ_c, β_c (**per channel**):

$$\hat{z}_{c,h,w}^{(i)} = \frac{z_{c,h,w}^{(i)} - \mu_g^{(i)}}{\sqrt{(\sigma_g^{(i)})^2 + \epsilon}}$$

$$z_{c,h,w}^{(i)} = \gamma_c \hat{z}_{c,h,w}^{(i)} + \beta_c$$

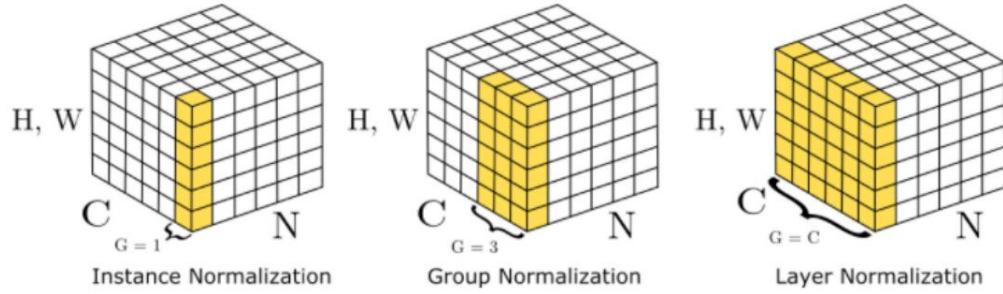
In practice, the **group size** is almost always **32**:

1. **Empirically validated:**

The original GroupNorm paper showed that $G = 32$ strikes a practical balance between **computational efficiency** and **model performance** across a wide range of vision tasks (like detection and segmentation) and network architectures.

2. **Good trade-off between LayerNorm and InstanceNorm:**

- If $G = 1$: GroupNorm becomes **InstanceNorm**, which normalizes each channel independently.
- If $G = C$: GroupNorm becomes **LayerNorm**, which normalizes across all channels.
- Choosing $G = 32$ gives a **middle ground**, allowing for cross-channel normalization without being as sensitive as InstanceNorm or as aggressive as LayerNorm.



Overview of Ways to Improve Generalization

Overfitting

In ML, **generalization** refers to model's ability to perform well on unseen data — not just on the training examples it was built on. One of the major challenges in achieving good generalization is overfitting.

When we train a model, the data it sees contains real patterns — the underlying regularities that describe the relationship between input and output — but it also contains random noise, or sampling error. This happens because any given dataset is just one possible sample of many, and it inevitably includes accidental patterns that don't reflect the true distribution.

The problem is that a model, when learning from the training data, cannot distinguish between true patterns and these spurious, sample-specific irregularities. If the model is highly flexible, it has the capacity to fit both the real and accidental regularities which leads to **overfitting**.

Preventing Overfitting

To prevent overfitting, several strategies can be used:

- **Get more training data:** Almost always the best bet if you have enough compute power to train on more data. In fact, more data reduces the impact of random sampling errors and helps the model capture genuine patterns more reliably
- **Use a model that has the right capacity:** one that's powerful enough to fit the true regularities, but not so flexible to also fit spurious regularities. Striking this balance is crucial for good generalization.
- **Average many different models:** This can be done by using models with different forms or by training the same model on different subsets of the training data and then averaging their predictions.

Limit the Capacity of a Neural Net

When it comes to neural networks specifically, there are several practical techniques to control their **capacity** and avoid overfitting:

- **Architecture: Limit the number of hidden layers and the number of units per layer.**
- **Weight regularization:** Discourage weights to become larger through L_2 or L_1 penalties. Both techniques constrain the model's complexity by keeping the weights small.
- **Early stopping:** Start the training with small weights and stop the learning as soon as performance on a validation set starts to degrade, preventing the model from overfitting to the training data.
- **Noise: Add noise** to the weights or the activities. Noise makes the model less sensitive to minor variations and helps it focus on more robust patterns.

In practice, a combination of several of these methods is used, as no single technique is universally sufficient on its own.

Weight Decay

One of the most effective ways to regulate model complexity is by **limiting the size of the weights**, which helps the model focus on capturing meaningful patterns rather than memorizing spurious or instance-specific patterns. A widely used method to achieve this is **weight decay**, implemented through **L2 regularization**. It works by adding a penalty term to the original loss function that penalizes the sum of the squared weights (**squared L2 norm**):

$$E_{reg}(w) = E(w) + \frac{\lambda}{2} \|w\|_2^2$$

where λ is the regularization coefficient that governs the relative importance of the regularization term with respect to the model error term.

It is called *weight decay* because it encourages the weight values to decay toward zero (but not exactly zero) unless they have big error derivatives. In fact, this term keeps the weights small pushing them near zero, while only the most important weights (the ones related to the true data regularities) will remain significant.

The actual complexity of the model is determined by the choice of the regularization coefficient λ :

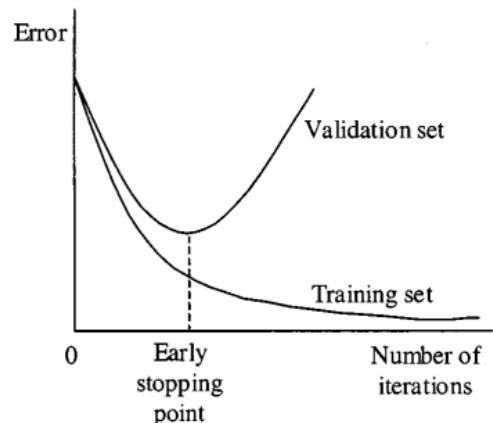
- When λ is set to a very **high value**, the penalty imposed on the model weights is so large that all parameters are reduced to almost zero. As a result, the model becomes **extremely simple**, with very limited ability to fit the training data (possible underfitting).
- In contrast, when λ is **very low** (near zero), the penalty on the model weights becomes negligible. This leads to a model with **maximum flexibility**, where all parameters are free to fully fit the training data. However, this can lead to excessive fitting to the training data and thus to a high probability of **overfitting**.

In a sense, λ determines the degree of overfitting. The problem of determining the optimal model complexity is shifted from that of finding the appropriate number of learnable parameters to that of finding the right balance between model complexity and generalization ability, through choosing an appropriate value for λ .

Another commonly used penalty term is the **L1 regularization**, which penalizes the sum of the absolute values (**L1 norm**) of the individual weights. This encourages some weight values to become **exactly zero**, effectively performing a feature selection.

Early Stopping

The training of DL models involves an iterative reduction of the error function defined with respect to a set of training data. Although the error function evaluated using the training set often shows a broadly monotonic decrease as a function of the iteration number, the error measured with respect to validation set often shows a decrease at first, followed by an increase as the network starts to overfit — capturing noise and spurious patterns in the training data. Therefore, to obtain a network with good generalization performance, training should be **stopped at the point where the validation error is lowest**. This is the central idea behind **Early stopping**. Instead of using the model parameters obtained at the final training iteration, we track the validation error throughout training and each time an improvement is observed, we save a copy of the current model parameters. Once the validation error stops improving for a predefined number



of iterations, training is halted and the model is rolled back to the best-performing parameters (the ones that lead to the best recorded validation error).

Early stopping is an easy-to-implement technique that does not require any substantial changes in the training procedure or the introduction of additional parameters to be learned during the process. In fact, it involves only a single hyperparameter, the number of iterations after which to stop learning. Typically, this value is chosen empirically using a validation set.

Beyond its practical simplicity, early stopping also serves as an **implicit regularization technique**. By halting training early, the model is prevented from reaching its full capacity to fit the training data, which reduces the risk of overfitting. In effect, it constrains the model's complexity by limiting how far the weights can grow. This connection to **model complexity** becomes clearer when considering the dynamics of weight growth during training in neural networks. At initialization, weights are small, and most units operate in their linear regime, leading to low model complexity. As training progresses, weights grow, non-linear behaviors emerge, and the network becomes more expressive. By stopping early, we halt this growth before the network reaches its full representational power. This mechanism closely resembles the effect of **weight decay**, which also constrains the model by keeping the weights small. In both cases, the result is the same: a model that generalizes better by avoiding unnecessary complexity.

Using Noise as a Regularizer

One interesting way to regularize a model — that is to control its complexity and reduce overfitting — is by introducing **random noise into the inputs during training**. Specifically, we can add **Gaussian noise** to each input value, perturbing it slightly. The Figure illustrates this idea for a simple net with a linear output unit directly connected to the inputs. Each input value x_i is perturbed by Gaussian noise:

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

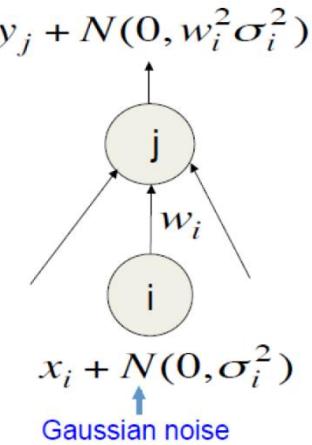
This noisy input is then multiplied by the weight w_i as it passes through the neuron. In particular **the variance of the noise is amplified by the squared weight** before going into the next layer. As a result, the contribution to the output at unit j becomes:

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

Now, since we are minimizing the **squared error** between the predicted output and the target, this added noise makes the error larger. As a result, the learning process is incentivized to **reduce large weights**, because large weights would amplify the noise more, increasing the overall error. In this way, minimizing the squared error when inputs are noisy naturally encourages smaller weights, achieving an effect similar to **L2 regularization**.

So, adding noise to inputs **indirectly** penalizes large weights and so **constrains the effective size of the weights**, helping to reduce overfitting and improve generalization.

Note: This technique is for instance used in *Denoising Autoencoders*, where noise is added to inputs during training to encourage the learning of more robust representations ([we will see better this in a later chapter](#)).



Ensemble of Models

When we train several different models to solve the same problem, a natural idea is to pick the single model that performs best on some validation set. However, instead of selecting the single best one, a more effective strategy

is to **combine their predictions**, creating what's known as an **ensemble** (or sometimes a **committee**) of models. This improves generalization by averaging out the individual errors of the models. This averaging process can be motivated by considering the **bias-variance trade-off**.

Let's recall from the ML Course that in regression tasks, the squared error can be decomposed into three components:

- (squared) **Bias** — the error due to incorrect or overly simplistic assumptions in the model (the bias measures how far the model's average predictions are from the true outputs). A high bias indicates the model cannot capture the underlying patterns in the data.
- **Variance** — the error due to sensitivity to small fluctuations in the training data (the variance measures how much the model's predictions fluctuate with different training sets). A high-variance model captures not only the underlying patterns but also the noise in the training set.
- **Irreducible error** — the natural noise inherent in the data, which no model can eliminate.

The bias term is big if the model has too little capacity to fit the data, while the variance term is big if the model has so much capacity that it is good at fitting the sampling error in each particular training set.

By combining multiple models, we can average out the variance without increasing the bias too much. This allows us to safely use individual models with high capacity (low bias, high variance) such as neural networks because their overfitting tendencies are reduced when combined. In this way, combining models effectively **reduces variance while retaining low bias**.

This is the key insight: **Ensemble methods allow us to use powerful (high-capacity) models without suffering as much from overfitting, because averaging helps cancel out their individual fluctuations.**

How Does the Ensemble Compare with the Individual Predictors?

On any one test case, some individual predictors may be better than the ensemble. In fact, different models could excel in different test cases, depending on how they have captured various patterns or noise in the data. However, when we **average the predictions over many test cases**, the combined predictor typically performs **better than any individual model**. This is especially true if the individual predictors disagree a lot, since errors of the individual models are less likely to align, making it easier for averaging to smooth them out. So, **we should try to make the individual predictors disagree** without making them much worse individually.

The goal is to increase diversity among models without sacrificing too much individual quality — this maximizes the benefit we get from averaging and it's a key factor in building an effective ensemble.

How to Encourage Diversity Among Predictors

Diversity can be introduced in various ways:

1. **Randomness during training:** Let the learning algorithm get stuck in different local optima. A dubious hack (but worth a try), since it can introduce variation.
2. **Architectural differences:** Use different types of models (e.g., neural networks, decision trees, Gaussian processes, support vector machines), or in case of neural network you can make them different by vary architectural choices like:
 - Number of hidden layers
 - Number of neurons per layer

- Activation functions
- Regularization methods (e.g., different types or strengths of weight penalty)
- Learning algorithms

3. Data variation:

a. Bagging (Bootstrap Aggregating): Train different models on different subsets of the data:

- Construct different training sets by using sampling with replacement: a,b,c,d,e → a,c,c,d,d. These datasets are known as **bootstrapping datasets**.
- Each data set can then be used to train a model (this way, each model sees a slightly different view of the data) and the predictions of the resulting models are averaged. This procedure is known as **bootstrap aggregation** or **b bagging**.

A popular example of ensemble models that uses bagging are **Random Forests**, which use it to train diverse decision trees. It has shown to cope well with the overfitting problem of single decision trees. We could use bagging with neural nets, but it's very expensive.

b. Boosting: Train a sequence of **low-capacity models**, where each model is trained to correct the errors made by the previous ones:

- Each base model is trained using a **weighted form of the data set** in which the weighting coefficient associated with each data point depends on the performance of the previous models. In particular, we **boost up-weight cases that previous models got wrong** by giving them greater value. So, like that the error for that training cases will be higher enforcing the model to give them more importance.
- Once all the models have been trained, their predictions are then combined using a weighted majority vote (classif. task) or by averaging their outputs (regress. task).

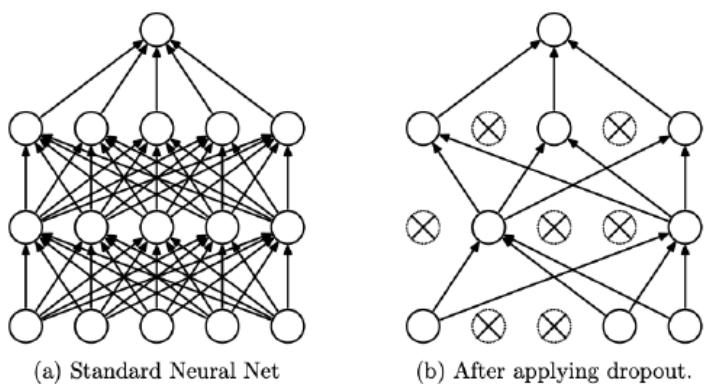
In practice, while ensemble methods are powerful and often lead to **significant performance gains**, they come at an important cost: multiple models have to be trained and then predictions have to be evaluated for all the models, thereby **increasing the computational cost of both training and inference**. How significant this depends on the specific application scenario.

Dropout

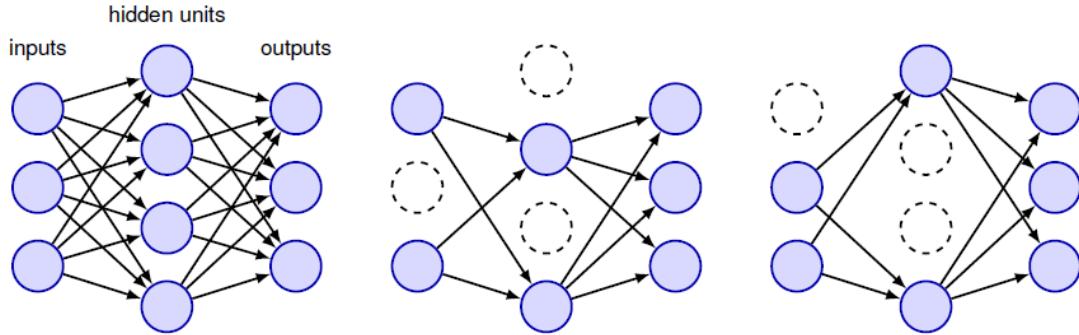
A very effective and widely used form of regularization is **Dropout**, introduced by Srivastava et al. in 2014 [20]. It can be viewed as an implicit way to perform approximate model averaging over exponentially many models without having to train multiple models individually. It has broad applicability and is computationally cheap.

The main idea behind dropout is to **randomly disable** (i.e., "drop out") **certain units** (neurons) in the network **during training**. This forces the model to become less reliant on any single neuron, which helps avoid overfitting to the training data.

During training each time a data point is presented to the network, a new random choice is made for which nodes to omit. Therefore, the network does not learn from the dropped-out neurons during that iteration.



For instance, the Figure below shows a simple network along with examples of pruned networks in which subsets of nodes have been omitted.



Dropout is applied to both hidden nodes and input nodes, **but NOT output nodes**, and is equivalent to setting the output of a dropped node to 0. It can be implemented by defining a mask vector $R_i \in \{0,1\}$ which multiplies the activation of each non-output node j for a given data point i . Each entry of the mask is set to 1 with probability p (the retention probability) and to 0 with probability $1 - p$ (the dropout rate).

Thus, each neuron is **retained with a certain probability p and “dropped” with probability $1 - p$** . A value of $p = 0.5$ (i.e., there's a 50% chance of dropping a neuron) seems to work well for the hidden nodes, whereas for the inputs a value of $p = 0.8$ is typically used.

During training, as each data point is presented to the network, a new mask is created and the forward and backward propagation steps are applied on that pruned network to obtain error function gradients, which are then used to update the weights, for example by SGD.

For a network with H non-output nodes, there are 2^H possible pruned networks, but only a small fraction of these networks will ever be considered during training. This differs from conventional ensemble methods in which each of the networks in the ensemble is independently trained to convergence.

Another difference is that the exponentially many networks that are implicitly being trained with dropout are not independent but share their parameter values with the full network, and hence with each other. The sharing of the weights means that every model is very strongly regularized, making dropout an even better regularizer than techniques like L_2 or L_1 regularization.

Finally, note that training can take longer with dropout since the individual parameter updates are very noisy (**and so you need to train the model for more time**).

What Happens at Test Time?

At test time, we want to make predictions using the entire network, without no longer randomly dropping any units. This is because we want to evaluate the network to its full capacity. However, during training each node was only active with a probability p and so we need in some way to compensate for this. This raises the question: how do we handle dropout when we no longer apply random masking? Answer: By **re-scaling the output weights** in the network so that the expected input to each node is roughly the same during testing as it would be during training, compensating for the fact that in training a proportion of the nodes would be missing.

In particular, if a node was kept with probability p during training, the outgoing weights from that node are multiplied by p during testing. This ensures that for any hidden unit the expected output (under the distribution used to drop units at training time) is the same as the actual output at test time.

For instance, if $p = 0.5$ for a hidden node, during testing the weights of that node are scaled by 0.5, accounting for the fact that just half neurons were present during training.

Thus, during testing, all neurons are active, but the weights are adjusted to ensure that the network behaves similarly to how it was trained.

!!! Note: In PyTorch with p is indicated the probability of dropping and not of keeping!

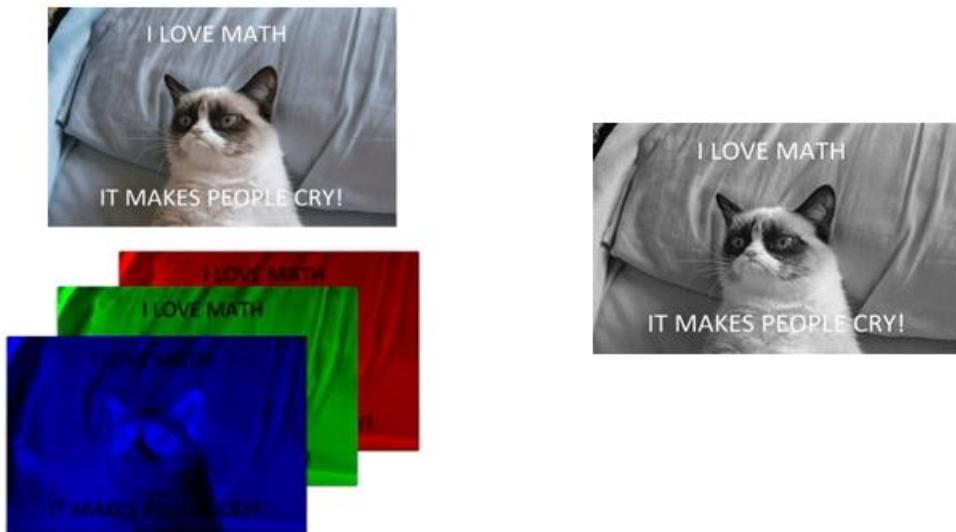
Another way to think about Dropout

A deeper motivation behind dropout is that it **reduces co-adaptation** among neurons. In a standard network without dropout, neurons can end up **co-adapting** to each other — each neuron may rely on the presence of others to make accurate predictions. This leads to overfitting because the neurons become overly specialized for the training data. With dropout, each node **cannot rely on the presence of other specific nodes** and must instead **make useful contributions in a broad range of contexts**, thereby reducing co-adaptation and specialization.

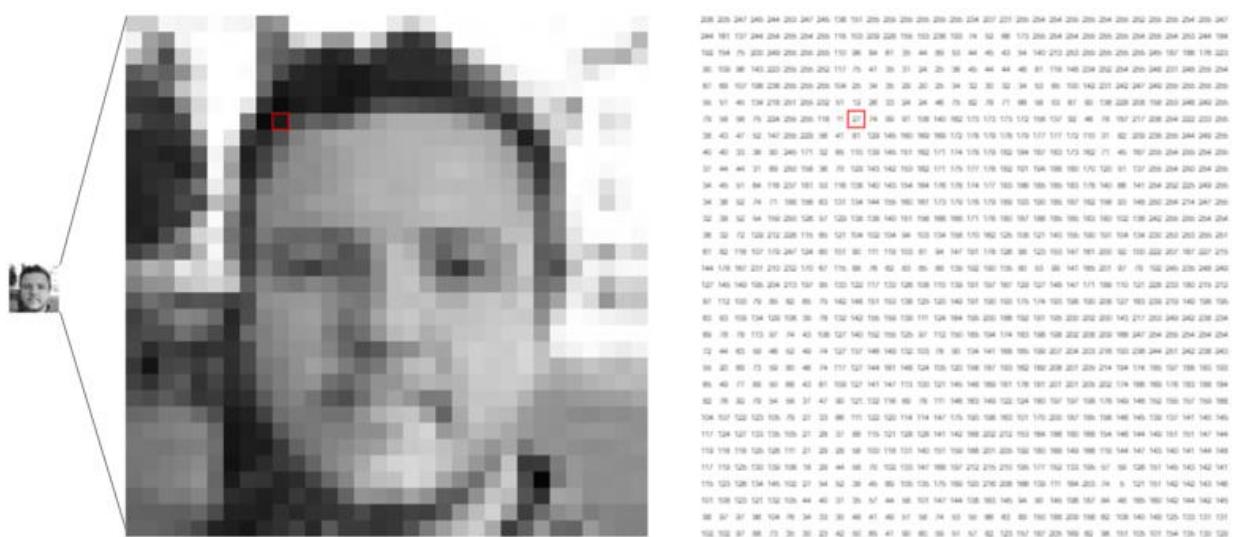
Convolutional Neural Networks

Image Data: What Makes Images Different?

Image data is a unique type of data with distinct characteristics that set it apart from other forms of information. One of the fundamental differences is its structure: images are inherently three-dimensional, consisting of **width**, **height**, and depth — commonly referred to as **channels**. In fact, each image is made up of a rectangular array of pixels, in which each pixel has either a **grey-scale** intensity (right Figure) or more commonly a triplet of **red**, **green**, and **blue (RGB)** channels each with its own intensity value (left Figure).



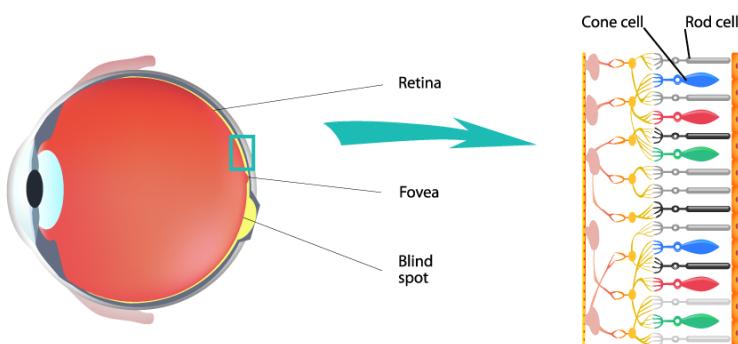
These pixel intensities are always non-negative numbers that also have some maximum value corresponding to the limits of the camera or other hardware device used to capture the image. In fact, generally, intensities are represented with finite precision, for example as 8-bit numbers represented as integers in the range $0, \dots, 255$. For instance, in the Figure Below we can see the 8-bit representation of pixel intensities for a $1 \times 32 \times 32$ image (grayscale image).



Note: If you're wondering why **RGB** is used to represent images on computers, the choice is inspired by how our eyes perceive color — though it's not an exact match, just a close approximation.

Our retinas contain two types of photoreceptive cells:

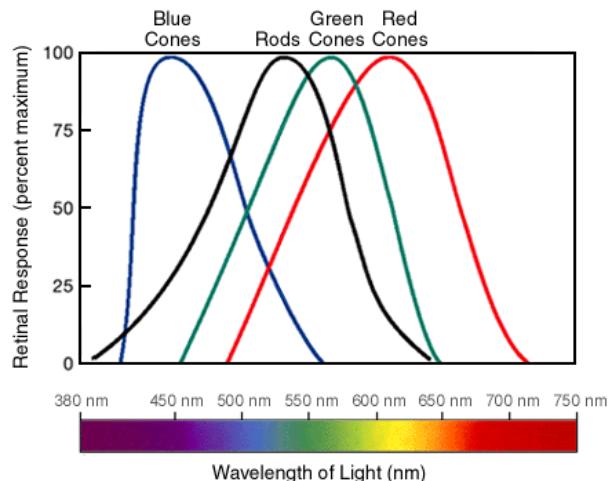
- **Rod cells**, which are highly sensitive to light, but do not detect color. They are useful in low-light conditions (which is why it's difficult to distinguish colors at dusk).
- **Cone cells**, which are responsible for color vision. There are three types, each sensitive to different wavelength ranges:
 - **Short-wavelength cones (S-cones)** – most sensitive to violet and blue light
 - **Medium-wavelength cones (M-cones)** – most sensitive to greenish-yellow light
 - **Long-wavelength cones (L-cones)** – most sensitive to orangish-red light



Our brain interprets color based on how strongly each type of cone is stimulated. These three types of cones are often labeled by the color they are most sensitive to: **blue, green and red**.

But if we only have three types of cones, how do we perceive so many different colors? The key lies in the overlap of their responses. For example, we see yellow when both green and red cones are stimulated, while blue cones remain inactive. This blending of signals allows us to perceive millions of distinct colors.

However, it's important to underline again that the way cone cells respond to light does not perfectly align with the red, green and blue primaries used in screens, as for instance M-cones are more sensitive to greenish-yellow than pure green and L-cones peak in an orangish-red range rather than pure red. Despite these differences, RGB works well for screens and cameras because mixing different intensities of red, green and blue light can approximate most of the colors our eyes perceive.



If you are interested, at this [Link](#) there is an interesting video that resume very briefly and visually how human eyes see the colors.

Images generally have a high dimensionality, with typical cameras capturing images comprising tens of megapixels. This makes it challenging to apply a standard fully connected architecture to image data as it will require a model with a vast number of parameters that would be infeasible to train.

To see this, look at how many weights we will have if we just consider a very small image such as one from the CIFAR-10 dataset, which consists of color images of size 32×32 pixels ($3 \times 32 \times 32$). A single neuron in the first hidden layer of a fully connected neural network processing this image would require $3 \times 32 \times 32 = 3072$ weights. Moreover, consider that more realistically images have generally larger sizes, i.e. an image $3 \times 256 \times 256$ would result in neurons with nearly **200,000** weights. All these weights are just for one neuron, however we would almost certainly want to have several such neurons, so the parameters would add up quickly! Clearly, this full connectivity is wasteful, and the huge number of parameters would quickly lead to overfitting.

Beyond computational inefficiency, this fully connected approach also fails to take account of the highly structured nature of image data, in which the relative positions of different pixels play a crucial role. We can see this because if we take the pixels of an image and randomly permute them, then the result no longer looks like a natural image. Similarly, if we generate a synthetic image by drawing random values for the pixel intensities independently for each pixel, there is essentially zero chance of generating something that looks like a natural image. This suggests that local correlations in an image are essential; nearby pixels are more likely to have similar colors and intensities compared to pixels that are far apart. Recognizing and leveraging these patterns allows us to build neural networks with far fewer parameters while improving their ability to generalize.

Another key property of image data is **translation equivariance** — if a particular portion of an image such as an eye in a face appears at one location, the same set of pixel values in a different location should still be recognized as an eye. We would like a neural network able to generalize what it has learned in one location to all possible locations in the image, without needing to see examples in the training set of the corresponding feature at every possible location. Similarly, we want the network to exhibit **invariance to small transformations**, such as minor shifts or rotations, so that an object remains recognizable even if its position slightly changes.

A fully connected network would have to learn any invariances and equivariances by example, which would require huge data sets. Therefore, by designing an architecture that is able to incorporate these **inductive biases** about the structure of images, we can reduce the data set requirements dramatically and also improve generalization with respect to symmetries in the image space.

Lastly, image data exhibits a natural **hierarchical structure**. A face in an image includes elements such as eyes, and each eye has structure such as an iris, which itself has structure such as edges and color variations.

At the lowest level of the hierarchy, a node in a neural network could detect



the presence of a feature such as an edge using information that is local to a small region of an image, and therefore it would only need to see a small subset of the image pixels. More complex structures further up the hierarchy can be detected by composing multiple features found at previous levels. A key point, however, is that although we want to build the general concept of hierarchy into the model, we want the details of the hierarchy, including the type of features extracted at each level, to be learned from data, not hand-coded. Hierarchical models fit naturally within the deep learning framework, which already allows very complex concepts to be extracted from raw data through a succession of possibly very many ‘layers’ of processing in which the whole system is trained end-to-end.

To fully exploit the two-dimensional structure of images and embed **inductive biases**, we need to incorporate concepts such as hierarchy, local connectivity, equivariance and invariance. **Convolutional Neural Networks (CNNs)**, introduced for the first time by Le Cun in 1998 [21], are designed precisely for this purpose, making them the most effective architecture for processing image data.

Filters as Feature Detectors

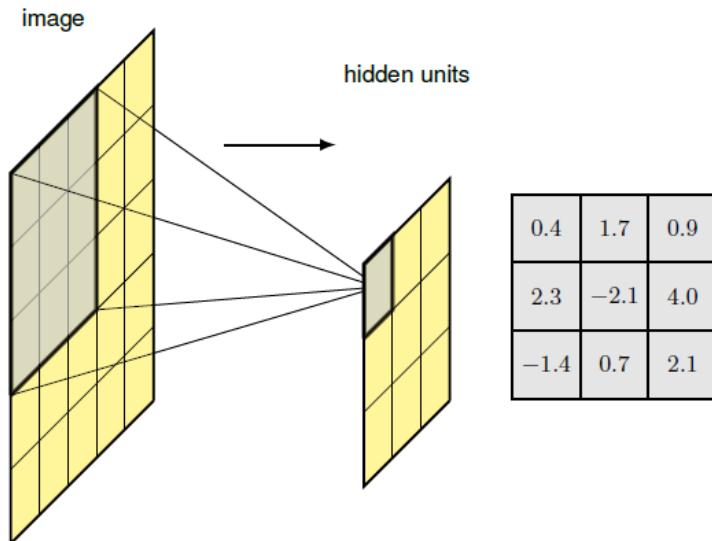
To simplify our discussion, let's first consider grayscale images, which have only a single channel. We'll extend these ideas to multiple channels later.

Consider a single unit in the first layer of a neural network that takes as input just the pixel values from a small rectangular region, or patch, from the image. This patch is referred to as the **receptive field** of that unit, and it captures the notion of locality (the idea that nearby pixels contain meaningful relationships). We would like weight values associated with this unit to learn to detect some useful low-level features. The output of this unit follows the standard neural network operation: a weighted linear combination of input values, followed by a nonlinear activation function:

$$a = \text{ReLU}(w^T x + b)$$

Here, x is the vector of pixel values in the receptive field and we assume a **ReLU** activation function (since it's the most common used), though other activation functions could be used.

Because there is one weight associated with each input pixel, the weights themselves form a small *two-dimensional grid* known as a **filter**, sometimes also called a **kernel**. This filter itself can be visualized as an image, as shown in the Figure below.

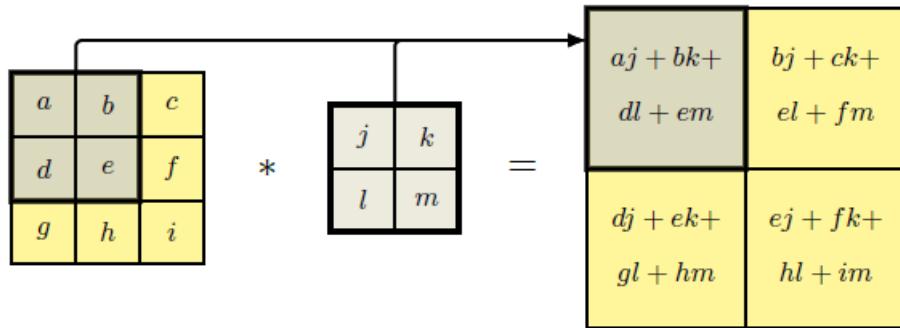


Now, suppose that w and b for this hidden unit are fixed and ask: *for which value of the input image patch x does this hidden unit produce the largest output response?* The maximum output response from this hidden unit occurs when x values align with the ones of filter w , meaning the unit is most activated when it encounters a pattern that closely resembles its filter. Note that the ReLU generates a non-zero output only when $w^T x$ exceeds a threshold of $-b$, and therefore the unit effectively acts as a **feature detector** that signals when it finds a sufficiently good match to its kernel.

Moreover, note that this approach is also **translation equivariant**: if we apply the same filter at multiple locations across the image, the unit will produce the same response whenever it encounters the same pattern, regardless of its position. In fact, if a local patch of an image produces a particular response in the unit connected to that

patch, then the same set of pixel values at a different location will produce the same response in the corresponding translated location in the output.

This idea forms the foundation of CNNs, where the same filter is systematically applied across the entire image through an operation known as **convolution**. We will explore convolution in more detail in the next section, but for now, you can picture it as a process where the filter starts at the top-left corner of the image, slides horizontally along each row and then shifts downward to the next row. At each position, the filter is multiplied element-wise with the corresponding region of the input image and the results are summed to produce an output value. This operation is illustrated in Figure below.



Note: A convolutional layer by itself does not include an activation function like ReLU — it only performs the convolution operation. However, it is typically followed by a non-linear activation function such as ReLU. In the discussion above, we included the activation function just to illustrate how a filter can act as a feature detector by responding strongly to certain input patterns.

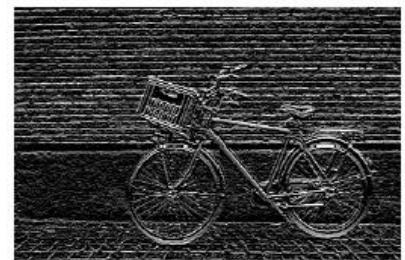
Now, let's come back to filter discussion. Consider the problem of detecting edges in images using a fixed, hand-crafted filter. Intuitively, a **vertical edge** appears where there is a significant change in pixel intensity as we move horizontally across the image. We can measure this by convolving the image with a **3×3 edge detection filter** of the form:

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

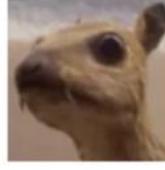
Similarly, **horizontal edges** can be detected using the **transpose** of this filter:

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

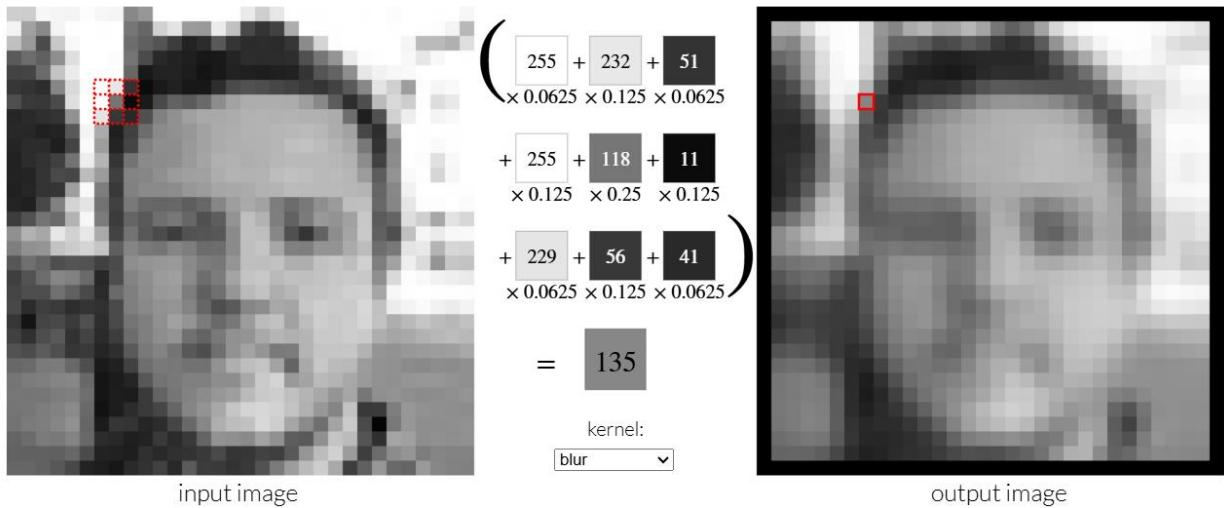
The Figure below illustrates the results of applying these two filters to a sample image. In the central image, note that a vertical edge that corresponds to an **increase** in pixel intensity produces a **positive response** in the output (represented by a light color), whereas an edge corresponding to a **decrease** in intensity produces a **negative response** (represented by a dark color). A similar effect occurs in the image on the right for horizontal edges.



The values within a filter determine what features it detects. Different filters can be designed to extract edges, enhance colors, blur an image or sharpen details. In image processing, these predefined filters have various names and purposes. For example, the **Gaussian Blur** filter smooths an image, the **Sharpen** filter enhances details and **Edge Detection** filters highlight object edges like shown in Figure below.

<i>Original</i>	<i>Gaussian Blur</i>	<i>Sharpen</i>	<i>Edge Detection</i>
$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$
			

You can also explore how different filters affect images interactively at [this link](#).



CNNs basically make use of filters of this type at different levels of architecture, but with an important difference: instead of relying on hand-crafted filters, they learn the optimal filter values during training from the data itself. This allows the network to automatically discover patterns and build a **hierarchy of features**, extracting increasingly complex representations from raw image data through multiple layers of processing.

Convolution and Cross-Correlation

Convolution is a mathematical operation widely used in digital signal processing to determine the level of similarity or correlation between two signals. Given two continuous signals, $x(t)$ and $k(t)$, their convolution is defined as:

$$(x * k)(t) = \int_{-\infty}^{\infty} x(\tau) \cdot k(t - \tau) d\tau$$

This operation computes the integral of the pointwise product between a signal $x(t)$ and a time-shifted version of the other signal $k(t - \tau)$.

In the case of discrete signals, we instead have a discrete convolution:

$$(x * k)(n) = \sum_{i=-\infty}^{\infty} x(i) \cdot k(n-i)$$

Here, x represents the input data, often called the **input feature map**, and k represents the **filter** or **kernel**. The result of the convolution, $(x * k)(n)$, is called the **output feature map**.

It's important to remember that a key property of convolution is **commutativity**, meaning that the order of the input and kernel can be switched without changing the result. This property holds for both continuous and discrete convolutions:

$$(x * k)(t) = (k * x)(t) = \int_{-\infty}^{\infty} k(\tau) \cdot x(t - \tau) d\tau$$

$$(x * k)(n) = (k * x)(n) = \sum_{i=-\infty}^{\infty} k(i) \cdot x(n-i)$$

In practical applications, we often use **kernels of finite size**. Suppose the kernel k has an odd length K ; in this case, the discrete convolution becomes:

$$(k * x)(n) = \sum_{i=-K/2}^{K/2} k(i) \cdot x(n-i)$$

Here we have considered a kernel with an odd length for simplicity, so that the central element $k(0)$ is well defined. Expanding this expression:

$$(k * x)(n) = k(-K/2) \cdot x(n + K/2) + \dots + k(0) \cdot x(n) + \dots + k(K/2) \cdot x(n - K/2)$$

We can observe that the leftmost element of the kernel, $k(-K/2)$, is multiplied by the rightmost element of the shifted input, $x(n + K/2)$, while the rightmost element of the kernel, $k(K/2)$, is multiplied by the leftmost element of the shifted input, $x(n - K/2)$. This operation is equivalent to **flipping** the kernel (rotating it 180° around its center) and then multiplying each element of x by the corresponding element of k .

In practice, however, a CNN does not directly use the convolution operation, but rather a similar operation called **cross-correlation**. The discrete cross-correlation between an input x and a kernel k of finite size K is defined as:

$$(k * x)(t) = \sum_{\tau=-K/2}^{K/2} k(\tau) \cdot x(t + \tau)$$

This operation is similar to convolution, with the difference that the kernel is not flipped (we have a “+” instead of a “-”). Each element of the input x is directly multiplied by the corresponding element of the kernel k :

$$(k * x)(n) = k(-K/2) \cdot x(n - K/2) + \dots + k(0) \cdot x(0) + \dots + k(K/2) \cdot x(n + K/2)$$

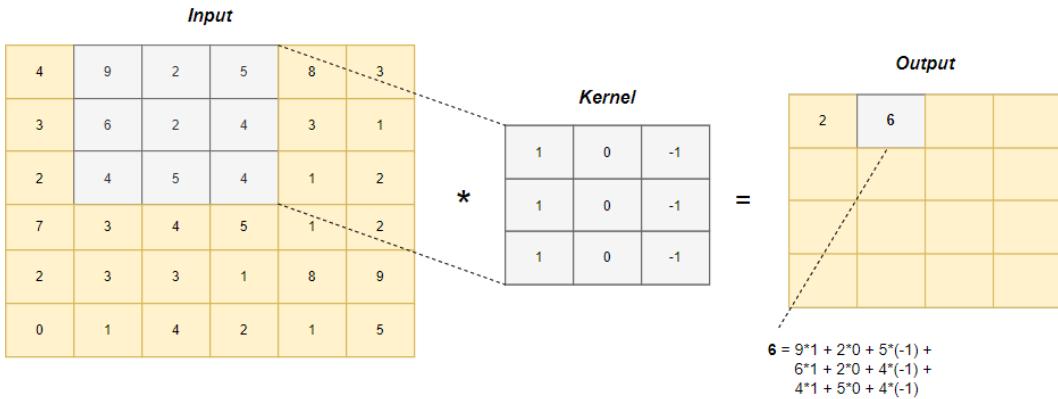
We can extend cross-correlation to higher-dimensional inputs and kernels. Suppose we have an input x and a kernel k , both D -dimensional, with dimensions along each axis defined by (n_1, \dots, n_D) and (K_1, \dots, K_D) , respectively. The cross-correlation between them is given by:

$$(k * x)(n_1, \dots, n_D) = \sum_{i_1=-K_1/2}^{K_1/2} \dots \sum_{i_D=-K_D/2}^{K_D/2} k(i_1, \dots, i_D) \cdot x(n_1 + i_1, \dots, n_D + i_D)$$

In the two-dimensional case (**2D**), often used in CNNs, where x represents an image of size $H \times W$ and k is a two-dimensional filter of size $K_H \times K_W$, this expression becomes:

$$(k * x)(h, w) = \sum_{k_h=-K_h/2}^{K_h/2} \sum_{k_w=-K_w/2}^{K_w/2} k(k_h, k_w) \cdot x(H + k_h, W + k_w)$$

This operation slides the kernel over the input image, multiplying the kernel weights element-wise with the corresponding image pixel values and summing the result to form the output. An example of this operation is shown in the Figure below, where it is illustrated the application of a 3×3 filter on a 6×6 image.



Therefore, although the operation actually used in CNNs is cross-correlation (where the kernel is not flipped), it is common in ML literature to refer to this operation as "convolution". For simplicity, we will follow this common convention in our discussion of CNNs, even though we are, in practice, referring to cross-correlation.

Padding

As we can see from the Figure above, the output feature map produced by a convolution operation is generally smaller than the original input feature map. If the input image has dimensions $H \times W$ pixels and we apply a kernel of size $K \times K$ (filters are usually chosen to be square), the resulting output feature map O will have dimensions:

$$d(O) = (H - K + 1) \times (W - K + 1)$$

For the example shown in the Figure above applying a 3×3 filter to a 6×6 image (input feature map) produces an output feature map of size $(6 - 3 + 1) \times (6 - 3 + 1) = 4 \times 4$.

However, in some cases, we might want to obtain an output feature map with the **same dimensions as the original input**. To achieve this, we use **padding**, which consists of adding extra pixels along the borders of the input image.

In general, this means adding P values on both sides of the image along each dimension. For example, considering the horizontal dimension, the size of the output feature map becomes:

$$W + 2P - K + 1$$

A common choice for the values of the added padding pixels is to set them to zero, after first subtracting the mean from each image so that zero represents the average value of the pixel intensity. This technique is commonly called **zero-padding**.

In our case, if P pixels of zero-padding are added to the image from the Figure above, the output feature map will have dimensions:

$$d(O) = (W + 2P - K + 1) \times (H + 2P - K + 1)$$

as shown in the Figure below.

0	0	0	0	0	0	0	0
0	4	9	2	5	8	3	0
0	3	6	2	4	3	1	0
0	2	4	5	4	1	2	0
0	7	3	4	5	1	2	0
0	2	3	3	1	8	9	0
0	0	1	4	2	1	5	0
0	0	0	0	0	0	0	0

When the value of P is chosen so that the output feature map has the **same dimensions as the input feature map**, we obtain what is called a "**same convolution**". This requires $P = (K - 1)/2$, an operation known as **same padding**. On the other hand, if no padding is used (i.e., $P = 0$), the operation is called a "**valid convolution**" (which corresponds to the standard convolution described earlier).

Strided Convolutions

In typical image processing applications, images may have a very large number of pixels, and since the kernels are often relatively small (with $K \ll H, W$), the convolutional feature map will be similar in size to the original image — or identical if same padding is used.

However, in some cases, it may be useful to **significantly reduce the size of the feature map** relative to the original image to gain more flexibility in designing convolutional network architectures. One way to achieve this is by using another important parameter of convolution, called the **stride**.

The **stride S** indicates the step size by which the kernel moves over the image during the convolution process. Normally, the kernel moves one pixel at a time ($S = 1$), but by using a larger stride S , the kernel moves by bigger steps. This reduces the size of the output feature map since fewer positions of the kernel are computed. The size of the output feature map along, for example, the horizontal dimension — taking both padding and stride into account — can be calculated as follows:

$$\frac{W + 2P - K}{S} + 1$$

By applying the same stride S both horizontally and vertically for the example above, the output feature map will have dimensions:

$$d(O) = \left(\frac{H + 2P - K}{S} + 1 \right) \times \left(\frac{W + 2P - K}{S} + 1 \right)$$

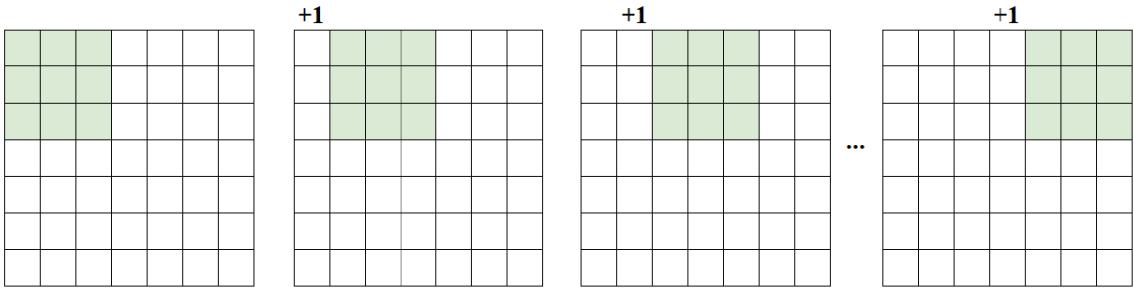
Note: In general, when processing large images with relatively small filters, using a larger stride results in reducing the feature map size by roughly a factor of $1/S$ compared to the original image size.

On the Applicability of Stride

Let's consider an image of size $H \times W = 7 \times 7$ — for simplicity, we'll denote $H = W = N$ — and a filter of size 3×3 (so $K = 3$). Now, depending on the stride value we choose, we get different outcomes:

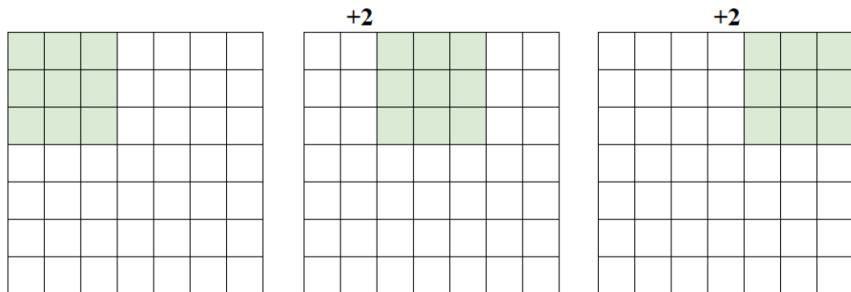
- $S = 1$

Applying a stride of 1 works without any issue → **Resulting output size: 5×5**



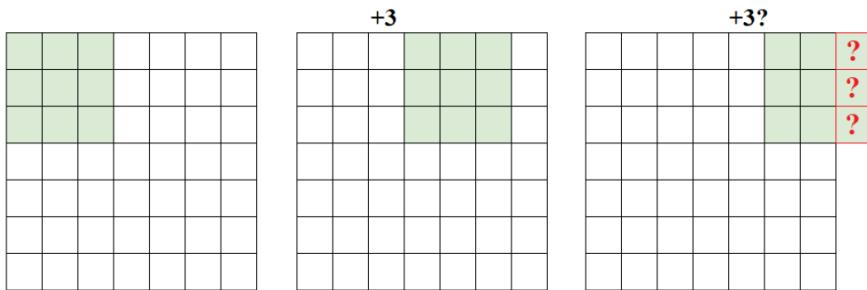
- $S = 2$

Applying a stride of 2 also works without any problem → **Resulting output size: 3×3**



- $S = 3$

A stride of 3 **cannot be applied directly**, because it doesn't evenly fit the image.



For a stride to be valid (i.e., cover the image completely without partial overlaps), the output feature map size must be an integer:

$$\text{Output size} = \frac{N - K}{S} + 1 \rightarrow \text{Integer}$$

So, this is the formula used to verify whether a given stride can be properly applied to the image.

Let's apply this condition to each stride from the example above:

- $S = 1$

$$\frac{7 - 3}{1} + 1 = 5 \quad (\text{OK})$$

- $S = 2$

$$\frac{7 - 3}{2} + 1 = 3 \quad (\text{OK})$$

- $S = 3$

$$\frac{7 - 3}{3} + 1 = 2.33 \quad (\text{NO})$$

To handle cases like $S = 3$, where the formula doesn't give an integer, we can use **zero-padding** to increase the effective size of the image and make the stride compatible. We must choose an appropriate padding value P so the output size becomes valid.

Let's try adding a padding of $P = 1$:

$$\frac{N + 2P - K}{S} + 1 = \frac{7 + 2 - 3}{3} + 1 = 3 \quad (\text{OK})$$

So, by carefully choosing the right padding value, we can enable the use of stride values that would otherwise be incompatible with the image dimensions.

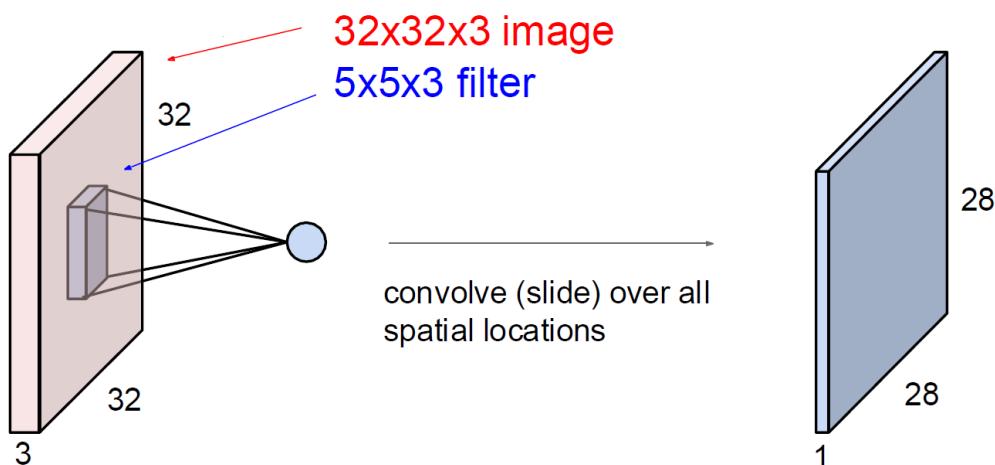
Convolutional Layer

So far, we have considered convolutions applied to a grayscale image, so an image with just one channel. For a color image, there are instead three channels corresponding to the red, green and blue components. We can easily extend convolutions to handle multiple channels by increasing the dimensionality of the filter accordingly. If an input image has dimensions $H \times W$ and C_{in} channels, it can be represented as a tensor of size $C_{in} \times H \times W$. To perform a convolution on this data, the filter must also have the same number of channels. Specifically, a filter designed for this input will have dimensions $C_{in} \times K \times K$. In other words, each slice of the filter operates on the corresponding input channel. The results obtained from all channels are then summed together, producing a single output feature map.

For instance, in Figure below we can see that applying a $3 \times 5 \times 5$ filter to a $3 \times 32 \times 32$ CIFAR-10 image results in an output feature map of size $1 \times 28 \times 28$.

Note: The value 28 comes from using $P = 0$ (no padding) and $S = 1$ (stride of 1). In fact, recalling the formula for the output spatial dimension $d(O) = (N + 2P - K)/S + 1$, in our case we will have $(32 + 2 * 0 - 5)/1 + 1 = 28$.

In terms of parameters, considering a filter of size $C_{in} \times K \times K$ we will have $C_{in}K^2$ weight parameters plus a bias parameter, for a total of $(C_{in}K^2 + 1)$ parameters.



In the case just described, we obtained a single output feature map because we considered the application of a single filter. Such a filter is conceptually analogous to a single hidden node in a fully connected network, capable

of detecting only one type of feature. To make the model more expressive and able to capture different types of features, it is sufficient to use multiple filters of this type.

A **convolutional layer** basically applies C_{out} different filters, all with the same size $C_{in} \times K \times K$. Each filter has its own independent set of parameters, which gives rise to its own independent feature map, for a total of C_{out} output feature maps. The total number of parameters of the layer will be given by $C_{out}(C_{in}K^2 + 1)$, since each filter has its own $C_{in}K^2$ weights and its own bias.

Therefore, a convolutional layer performs an operation that can be formally described as follows:

$$O(i) = b(i) + \sum_{j=0}^{C_{in}-1} k(i,j) * x(j), \quad \forall i = 1 \dots C_{out} - 1$$

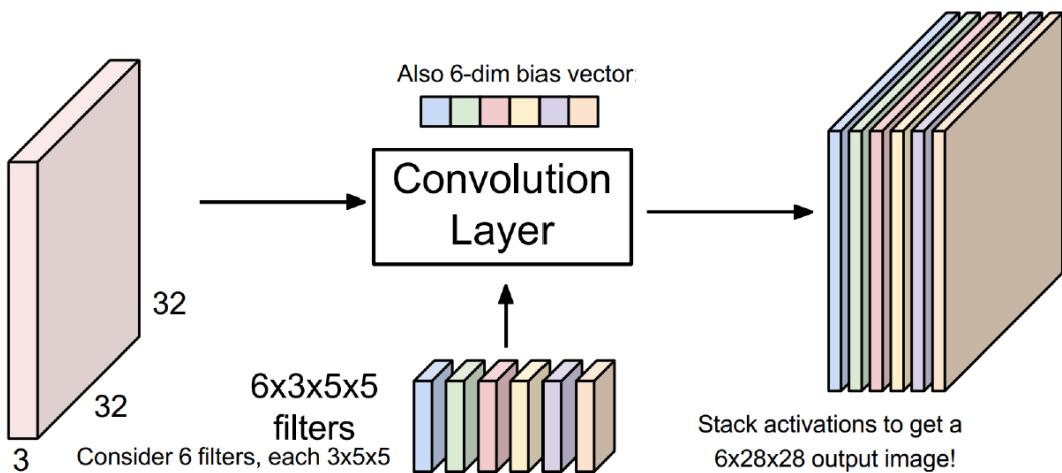
where $x(j)$ represents the j -th channel of the input feature map, $k(i,j)$ is the slice related to the j -th input channel of the i -th filter and $b(i)$ is the bias associated with the i -th output channel.

Note: I want highlight that the number of channels in the filter (C_{in}) must match the number of channels in the input feature map x . The number of output channels C_{out} , instead, is a hyperparameter of the layer. Additionally, kernel size, stride and padding are also hyperparameters of the layer.

This means that, for each filter, we first compute the cross-correlation between each corresponding input channel and the filter's channel slice — and this is why the number of channels in the kernel must match that of the input. The results from all input channels are then summed together and a bias term is added element-wise. This process is repeated for each of the C_{out} filters in the layer, producing a final output feature map with C_{out} channels.

Therefore, each filter in the convolutional layer will produce an output feature map of the same spatial dimensions, and these maps are stacked together along the depth (channels) dimension, defining the number of channels of the next layer.

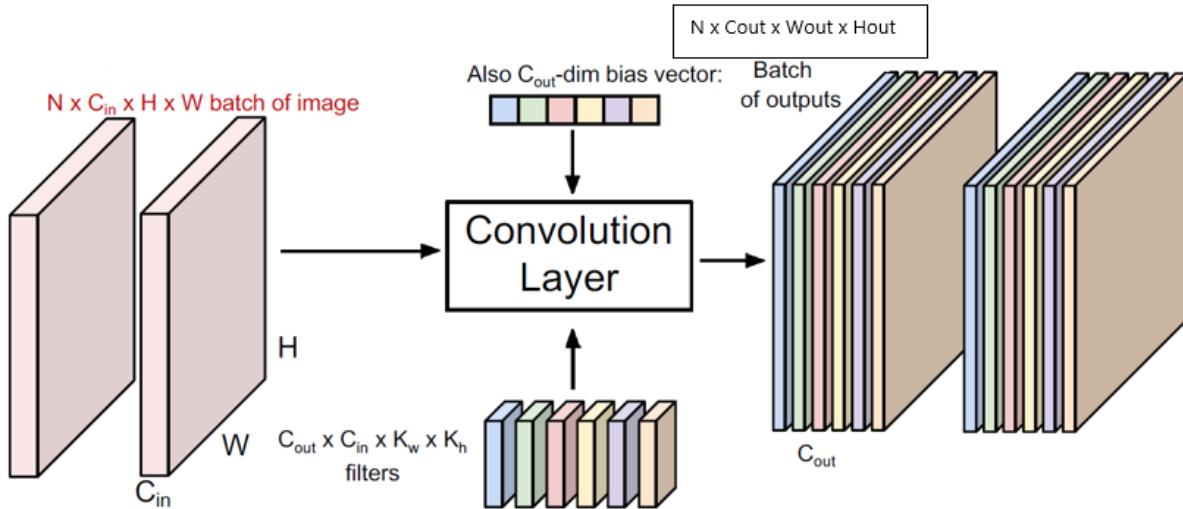
For example, again considering a $3 \times 32 \times 32$ CIFAR-10 image, if we apply **6 filters** of size $3 \times 5 \times 5$, with zero padding and stride equal to 1, we obtain an output feature map of size $6 \times 28 \times 28$. In practice, the application of each filter generates a feature map of size $1 \times 28 \times 28$, and these are stacked together along the channel dimension, resulting in $6 \times 28 \times 28$.



Finally, in the case of a mini-batch — when processing **N images** at a time — the operation extends to all images in the batch. The complete expression becomes:

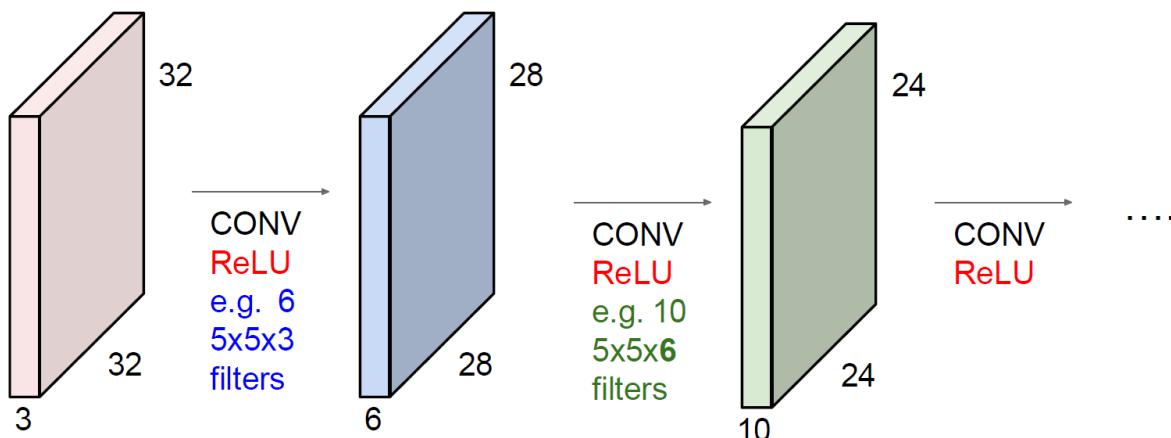
$$O(n, i) = b(i) + \sum_{j=0}^{C_{in}-1} k(i, j) * x(n, j), \quad \forall i = 0, \dots, C_{out} - 1 \quad \forall n = 0, \dots N - 1$$

where N is the batch size and n identifies the n -th image in the considered batch.



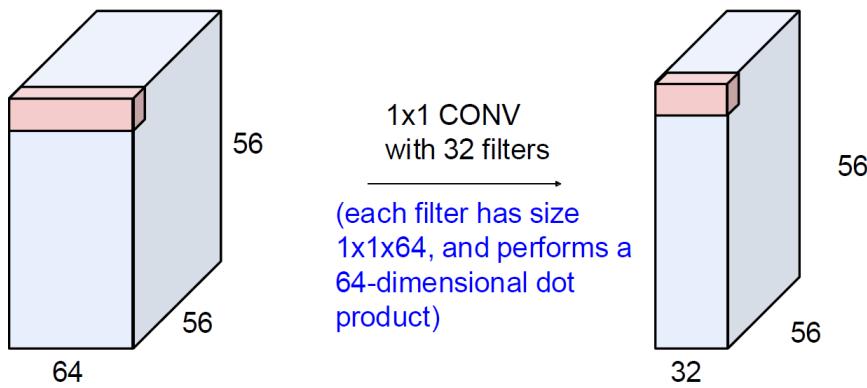
To enable the network to capture increasingly complex and hierarchical features in the data, CNNs stack multiple convolutional layers one after the other, where each convolutional layer is generally followed by an activation function such as a ReLU.

Note: We recall that activation functions apply multiple-wise transformations, so the dimensions are kept the same.



1x1 Convolutional Layer

A particularly useful concept when designing convolutional networks is the **1 × 1 convolutional layer**. This is essentially a convolutional layer where the filter size is $K = 1$, meaning it processes one pixel at a time across all its channels. One application for 1×1 convolutions is simply to change the number of channels (typically reduce them) without changing the size of the feature maps, by setting the number of output channels to be different to the number of input channels.



Local Connectivity and Parameters Sharing

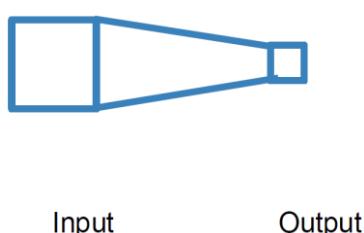
Compared to a standard fully connected network, convolutional architectures offer two key advantages:

1. **Local connectivity** – Instead of connecting every neuron to every input pixel, a convolutional layer focuses on local regions of the input. Each unit in a convolutional layer is connected only to a small, localized region of the previous layer, known as its *receptive field*. The size of this receptive field — which corresponds to the filter size — is a hyperparameter. Importantly, while connectivity is local in the spatial dimensions (width and height), it is always *full* along the depth dimension: each filter spans the entire depth (channels) of the input. This introduces an asymmetry in how spatial and depth dimensions are treated: **local in 2D space but fully connected across depth**.
2. **Parameter Sharing** – The same filter is applied across different regions of the input feature map. This means that the weights are **shared** by all the units that use the same filter as it moves across the image, drastically reducing the total number of parameters compared to a fully connected layer. Importantly:
 - **Filters do not share weights with each other** — each filter has its own unique set of parameters and learns to detect a distinct feature. E.g. the weights of filter A are not used by filter B but are independent of each other.
 - **Within a given filter, however, the weights are shared across different spatial positions.** So, if the same filter is applied to two different regions A and B of the image (with $A \neq B$), the output values are computed using the same weights.

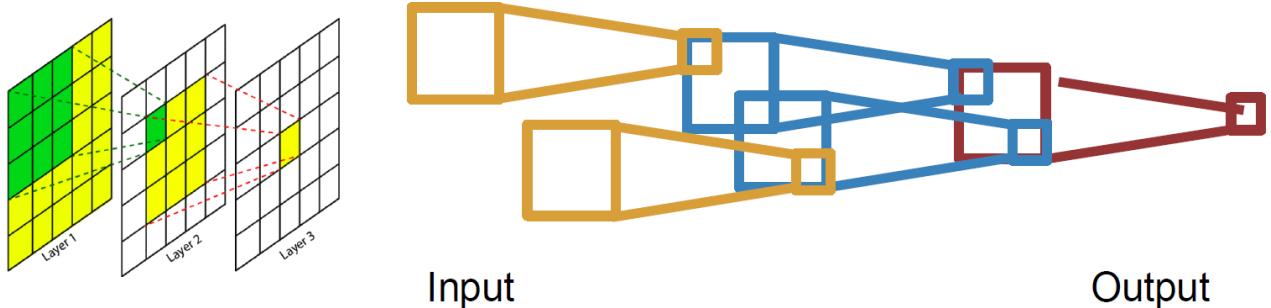
This parameter sharing is one of the main reasons convolutional networks are so efficient for image processing tasks — it not only reduces the number of parameters significantly but also allows the network to detect patterns regardless of their position in the image.

More about Receptive Fields

As already talked above, one of the core ideas behind CNNs is that each unit in a given layer corresponds to a **receptive field** in the previous layer.



Starting from the input image as we move deeper into the network, the receptive field associated with each point in the successive layers **increases in size relative to the input layer**. This is because each layer aggregates information from its own local neighborhood, which is itself built on neighborhoods from the previous layer, gradually expanding the effective field of view.



Consider a convolutional layer with kernel size K , so that each output unit depends on a $K \times K$ region from the previous layer. When stacking multiple layers such as that, i.e. with the same kernel size K ([supposing a stride 1 and no padding](#)), each subsequent convolution adds $K - 1$ to the size of the receptive field. Specifically, after L such layers, the size of the receptive field becomes:

$$recep_field^{(L)} = 1 + L(K - 1)$$

Example:

Let's assume the kernel size is $K = 3$. Then:

- For the **first hidden layer**:

$$recep_field^{(1)} = 1 + 1 * (3 - 1) = 3$$

- For the **second layer**:

$$recep_field^{(2)} = 1 + 2 * (3 - 1) = 5$$

- For the **third layer**:

$$recep_field^{(3)} = 1 + 3 * (3 - 1) = 7$$

- ...

- At **layer $L = 10$** :

$$recep_field^{(10)} = 1 + 10 * (3 - 1) = 21$$

This means that **one point in the 10-th layer corresponds to a 21×21 region in the original input image**. As we go deeper, each unit "sees" more of the input — and eventually, one unit may cover the **entire input image**, which is essential for capturing global context and therefore make reliable high-level predictions (i.e. for classification tasks).

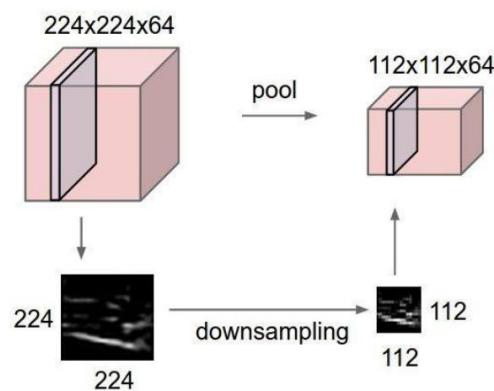
Problem: For large input images, stacking many small-kernel convolutional layers can become **computationally expensive**, since many layers are needed before the receptive fields of the output units can “see” the whole image. And this is not good news since each layer introduces additional parameters, increasing model complexity and training time.

Solution: **Downsampling** inside the network. Basically, the idea is to progressively reduce the spatial dimensions while increasing the receptive field more efficiently. We can achieve this through:

- **Strided convolutions:** reduce the amount of information stored by increasing the Stride. **Note:** While this can help reduce computation, it must be used carefully — a stride that's too large will discard too much spatial information, harming the model's performance.
- **Pooling**

Pooling Layer

The **pooling layer** is a fundamental component of CNNs, primarily used for **reducing the spatial dimensions (down-sampling)**. Its main purpose is to make the feature maps smaller and more manageable while retaining the most important information.

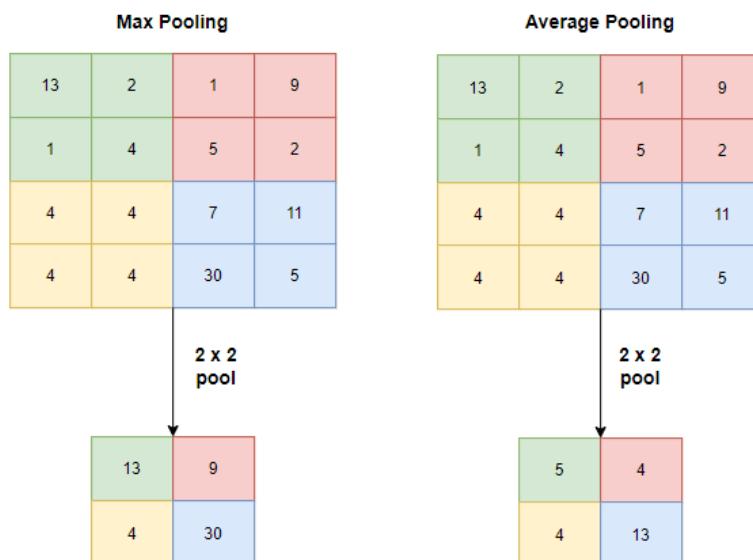


Pooling layers are typically inserted after one or more convolutional layers.

Like a convolutional layer, a pooling layer arranges units in a grid, where each unit receives input from a local **receptive field** in the previous feature map. Here too, the size of the pooling window (filter) and the stride can be chosen (but **not the padding parameter**). The key difference is that the pooling layer performs a simple, **fixed aggregation function** over the inputs in each receptive field and thus there are no learnable parameters in pooling.

Two of the most common pooling operations are:

- **Max pooling**, which outputs the maximum value within the receptive field.
- **Average pooling**, which outputs the average of the values within the receptive field.



It's important to note that pooling is applied independently to each channel in the feature map ([same as a convolutional filter](#)). For instance, if we have a feature map with 8 channels, each of size 64×64 , and we apply a max-pooling operation with a 2×2 window and stride 2, the result will be a tensor of size $8 \times 32 \times 32$. In practice, this halved the spatial resolution ([due to stride=2](#)) of each channel!

In addition to reducing the size of the feature maps, pooling layers play a crucial role in [introducing spatial invariance into the network](#). Spatial invariance means that the model becomes less sensitive to small translations or distortions in the input — which is often desirable in visual tasks like image classification.

For example, imagine an object in an image which shifts slightly to the left or right. Without spatial invariance, this small change in position could result in completely different feature representations in deeper layers, making the model fragile and overly dependent on exact pixel alignments. We want the output of the network to be robust to small translations of the input — in other words, the prediction should not change if the object in the image shifts slightly. Pooling helps achieve this by summarizing the activations within local regions, so that minor variations in the position of a feature don't significantly alter the output.

Take *max pooling* as an example: it selects the most prominent activation within a local region, regardless of its exact position. This means that [as long as a feature is somewhere within the pooling window, it will still be detected with the same strength](#). Therefore, max pooling helps the network retain the information that "a feature is present" without being overly concerned with where exactly it appears.

Therefore, this spatial invariance introduced by pooling is particularly beneficial for building more robust and generalizable representations, considering that real-world data often involves objects that can appear in slightly different positions, scales or orientations.

FC Layer

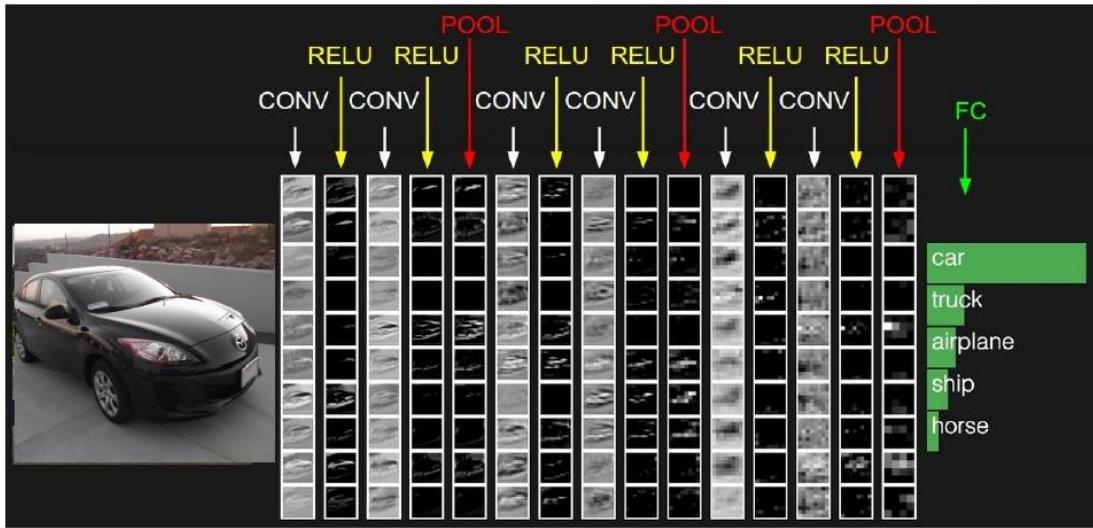
In many applications, the final goal of a network is to make predictions about the image as a whole — for instance, assigning a class label in image classification — and so the output units of the network need to combine information from across the whole of the input image. This is typically achieved by adding one or two standard **fully connected (FC) layers** at the final stages of the network, in which each unit is connected to every unit in the previous layer.

Important note: Convolutional and pooling layers produce multi-dimensional outputs, typically structured as (channels, height, width). However, fully connected layers expect a **one-dimensional vector** as input, since each neuron in an FC layer connects to a flat list of numbers rather than a spatially organized grid. To bridge this gap, the multi-dimensional tensor is [flattened](#) into a single vector before passing it to the fully connected layers.

Example: Suppose that the final convolutional layer outputs a tensor of shape $[128, 7, 7]$. *Flattening* converts this into a vector of size $128 \times 7 \times 7 = 6272$. This 6272-dimensional vector is then fed into the FC layer.

Typical CNN Architecture

The typical architecture of a CNN generally consists of a sequence of **convolutional layers followed by non-linear activation functions such as ReLU**, periodically interleaved with **pooling layers that reduce the spatial dimensions**. In many cases, **FC layers are added in the final stages of the network**, as illustrated in the Figure below.

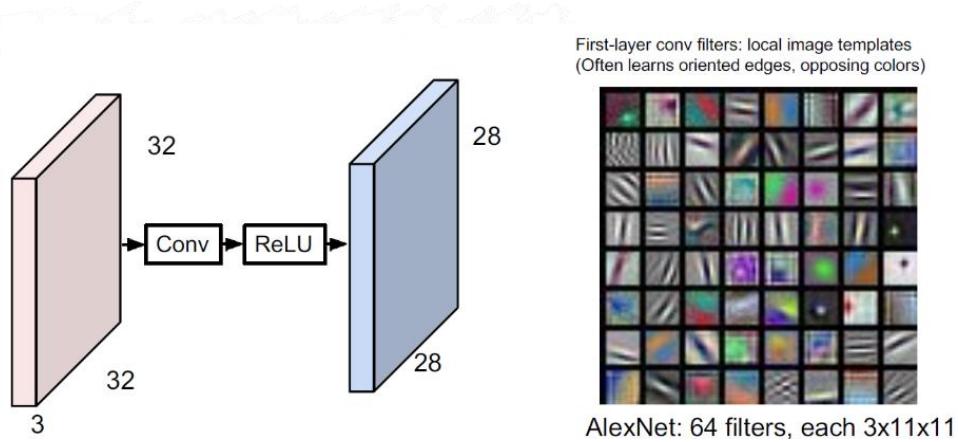


It is important to highlight that while **convolutional layers benefit from parameter sharing**, resulting in a **significantly lower number of parameters**, **FC layers do not share this property**. In fact, even though convolutional layers can be numerous, **the majority of CNN's parameters are typically concentrated in the final FC layers**.

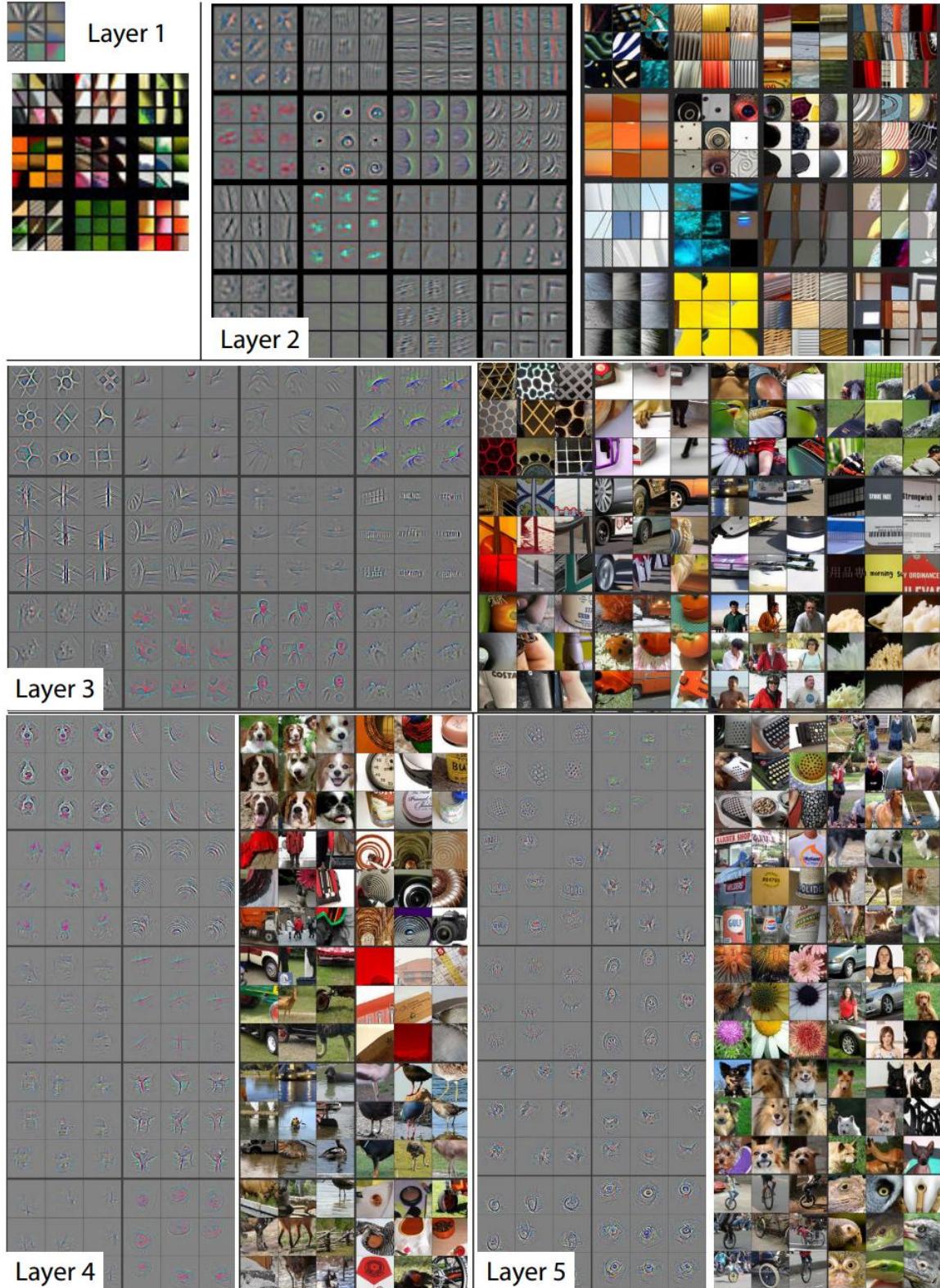
What Do Convolutional Filters Learn?

To understand what these filters learn, it's important to look at the result of applying the activation function over these feature maps since these activations can be interpreted as a measure of strength with which a certain feature is detected in a particular region of the input image.

In the Figure below, we see an example of this process: an input image passes through a convolutional layer followed by a ReLU activation. The resulting output is a set of activation maps (one per filter), where each filter emphasizes particular patterns it has learned to recognize.

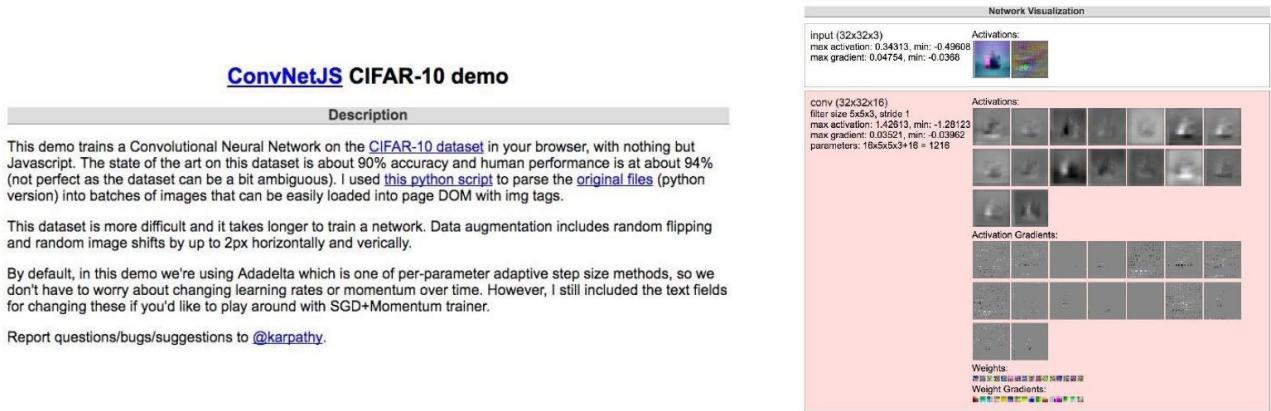


For example, in **AlexNet** — a network we'll explore in the next section — the first convolutional layer uses **64 filters**, each of size $3 \times 11 \times 11$. At this stage, the filters typically learn to detect simple, localized patterns such as **oriented edges and contrasting colors**, which are common in natural images. As you can see from the Figure each filter learned to respond to specific low-level features present in small regions of the image. These filters then become building blocks for the features at the next layers which will combine them to form increasingly complex visual representations. In fact, as we move deeper into the network, the filters start combining these simpler patterns to detect more abstract and meaningful structures — from textures and object parts to entire object as shown in the Figure below.



Note: The illustrations shown above are the result of an insightful technique for visualizing the features learned at different layers of a CNN introduced by Zeiler & Fergus (2013) [22]. They make use of **transpose convolutions**, that practically make the reverse of a convolution, i.e. instead of mapping pixels to features do the opposite ([we will better see them when we talk about U-Net](#)). The paper also shares several key observations and insights about CNN features learning that it's well worth reading.

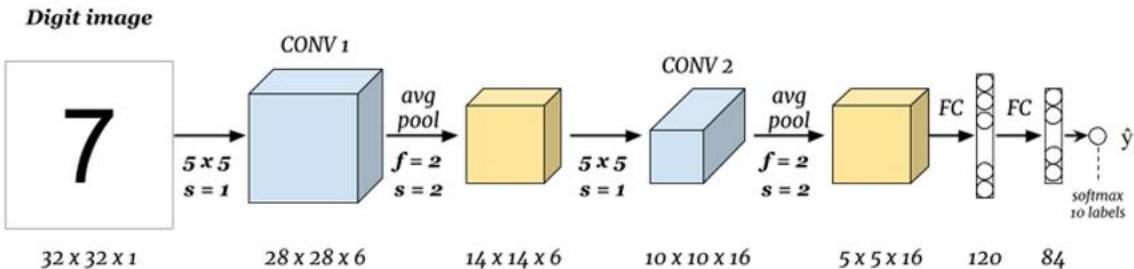
A nice example of this progressive feature learning process can be seen in the ConvNetJS demo for CIFAR-10, available at this [link](#). This interactive demo shows, in real time, how a CNN trained on the CIFAR-10 dataset activates different filters across its layers.



LeNet, AlexNet & VGG-16

CNNs were the first deep neural networks (networks with more than two layers of learnable parameters) to be successfully applied to practical tasks.

The first CNN was **LeNet** [23], designed by Yann LeCun in 1998, which was used to classify **low-resolution grayscale images of handwritten digits**. In this case, there were 10 possible classes (digits 0–9).

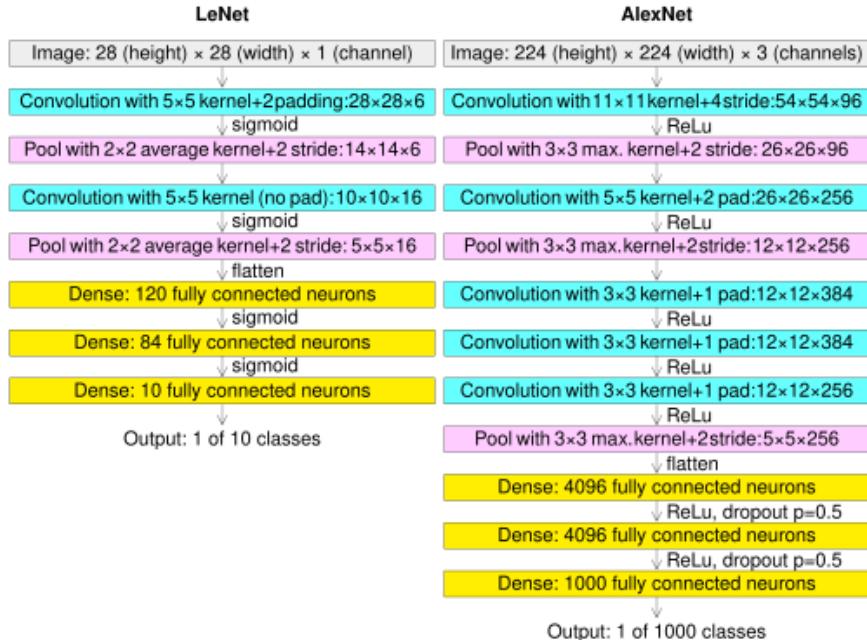


The development of more advanced CNN architectures was later accelerated by the introduction of the **ImageNet** dataset [24], which contains around 14 million labeled images across nearly 22,000 categories. This was a far larger dataset than those previously available, and the breakthroughs driven by ImageNet highlighted the importance of large-scale data, along with well-designed models equipped with appropriate inductive biases, in building successful deep learning solutions.

A subset of this dataset, consisting of **1,000 non-overlapping classes**, became the basis for the annual **ImageNet Large Scale Visual Recognition Challenge (ILSVRC)**. This benchmark significantly pushed the limits of CNN architectures, as the problem's complexity increased dramatically with the presence of so many categories and so if the classes were evenly distributed, a random guess would result in a 99.9% error rate. The dataset includes 1.28 million training images, 50,000 validation images, and 100,000 test images. Classifiers are designed to produce a ranked list of predicted output classes for each test image, with performance reported using **top-1 and top-5 error rates** (meaning a prediction is considered correct if the true class appears as the top prediction or within the top five predicted classes respectively).

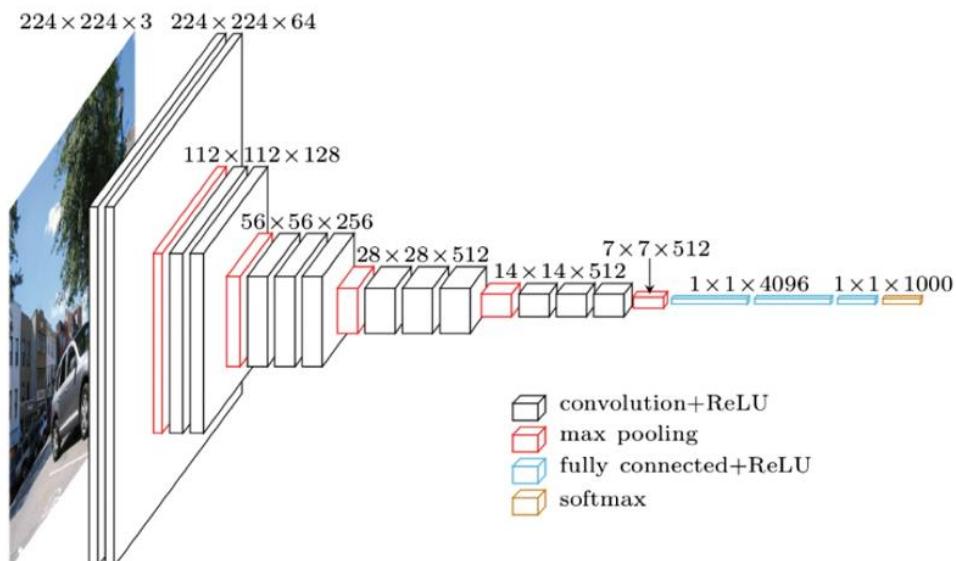
A major breakthrough came with **AlexNet** [25], a CNN architecture designed by graduate students Alex Krizhevsky and Ilya Sutskever, under the supervision of Geoffrey Hinton. This model **won the 2012 ILSVRC competition**,

reducing the top-5 error rate to a new record of 15.3%. Key aspects of this model included the **use of ReLU activation functions**, **GPU-accelerated training** and **dropout regularization**. Notably, AlexNet was also the **first CNN to operate on RGB images rather than grayscale images**, marking a significant evolution in CNN-based image classification.



In the following years, further advancements made error rates even lower, reaching around **3%** (achieved with **ResNets** that we will see next), slightly surpassing human-level performance on the same dataset (estimated around 5%). This is partly due to the difficulty humans face in distinguishing between subtly different categories — for example, different species of mushrooms.

As an example of a typical modern CNN architecture, we can examine **VGG-16** in detail, as many subsequent models followed its architectural design principles. **VGG-16** [26] consists of **16 learnable layers**, based on a small number of simple, consistent design choices that lead to a relatively uniform structure, minimizing the number of hyperparameter decisions required.



The model takes as input a 224×224 RGB image, followed by a **sequence of convolutional layers interleaved with pooling layers** for downsampling. All convolutional layers use **3×3 filters with stride 1 and padding 1**, while **max pooling layers use 2×2 filters with stride 2**, halving the spatial resolution at each step. Notably, VGG-16 follows a **systematic increase in the number of output channels** (i.e. the number of filters) at each stage, progressing through multiples of 2: **64, 128, 256** and **512**. The convolutional stack is followed by **three FC layers**, the last of which outputs **1,000 units** to support classification over 1,000 categories. All learnable layers (convolutional and FC) use ReLU nonlinearities, except for the final output layer, which uses a softmax activation.

An important point to underline is that earlier CNNs typically used fewer convolutional layers, relying instead on larger receptive fields to ensure that higher-level units could "see" the entire input image. For instance, AlexNet employed **11×11 filters with a stride of 4** in its initial layers, allowing it to quickly expand the receptive field. This design was effective in increasing the field of view early in the network, but it came at the cost of reduced spatial precision and flexibility.

However, as demonstrated in VGG-16, an alternative strategy is to **stack multiple layers with smaller receptive fields**, such as **3×3 filters with stride 1**. The advantage of this approach is that it allows the receptive field to grow gradually while **preserving more fine-grained spatial information** at each step. Additionally, using multiple small filters introduces **more nonlinearities** (via activation functions), which increases the network's expressive power and ability to model complex patterns — all while maintaining a relatively manageable number of parameters due to parameters sharing. This design principle — combining **depth, modularity and gradual receptive field growth** — has become a foundational strategy in modern CNN architectures, favoring **deep, layered structures over aggressive early downsampling**.

ResNet

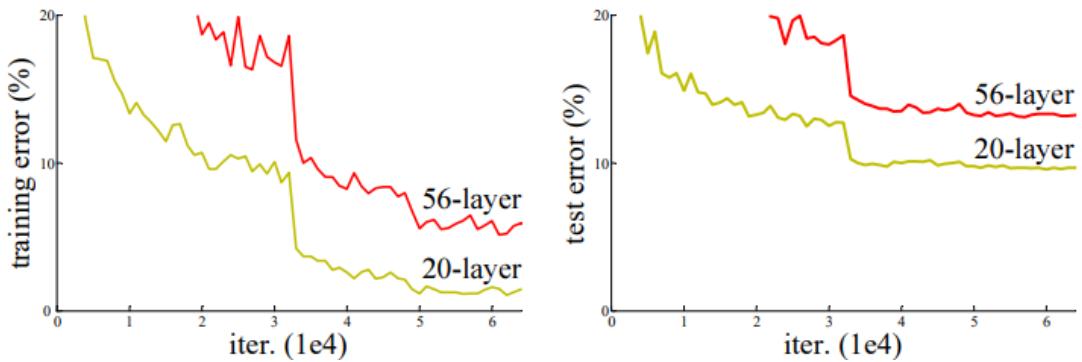
After the success of VGG and other deep convolutional networks, it became clear that increasing network depth was a promising way to improve performance. Deeper models could progressively extract richer, multi-level feature representations — from low-level edges and textures to high-level object parts — and this was key in achieving state-of-the-art results on challenging benchmarks like ImageNet.

In theory, adding more layers should allow a model to learn better representations and reduce both training and test errors. At the time, the best-performing models on ImageNet were pushing depths of up to 30 layers, showing noticeable improvements. But then when researchers tried to push up that limit something opposite happened: adding more layers actually degraded performance — the error rate, instead of decreasing steadily, started increasing again as the networks grew deeper.

So, at the time the authors of **Residual Net (ResNet)** [27] posed an important question: "**Is learning better networks as easy as stacking more layers?**"

An obstacle to answering this question was the notorious problem of vanishing/exploding gradients, which hamper convergence from the beginning. However, by the time of ResNet, techniques like **normalized weights initialization** and intermediate **normalization layers** such as *Batch Norm* (BN) were already in place, allowing networks with tens of layers to converge properly for SGD with backpropagation. In fact, the ResNet authors verified that gradients maintained healthy norms with BN, and both forward and backward signals propagated without vanishing.

At this point, one might suspect **overfitting** was the cause of the performance degradation — perhaps the deeper model was simply fitting the training data too well, leading to poor generalization. But surprisingly, this wasn't the case. Experiments showed that adding layers actually **increased training error** as well as test error. If overfitting were the issue, training error would decrease while test error increased — but here both were getting worse.



The authors reasoned about this and they come up with an hypothesis: “Could we actually have a very deep network which can be at least as accurate as a moderately deep network?”

To prove that this should actually be possible, they conducted an experiment. Consider two networks:

- A shallower baseline model (like VGG-16)
- A deeper version of the same model, created by simply adding at the end of the baseline model some extra layers initialized to perform the **identity function**

By construction, the deeper model should be able to at least mimic the shallower one by simply passing the input through these identity layers and therefore it should produce no higher training error than its shallower counterpart (it should be equal or lower). However, the result of the experiment, in practice, shown that traditional optimizers (like SGD with momentum at the time) failed to find solutions in deep models that were unable to find solutions that are comparably good or better than the shallower counterpart (or unable to do so in feasible time), even though such solutions clearly existed. This suggested that the degradation problem wasn’t caused by overfitting or convergence issues due to bad gradients. The problem was more subtle: **deeper models are just harder to optimize, even though they have strictly more expressive power** and simply stacking more layers on a well-performing shallow model **doesn’t guarantee improvement** — even when the added layers are theoretically redundant like identity mappings in this case.

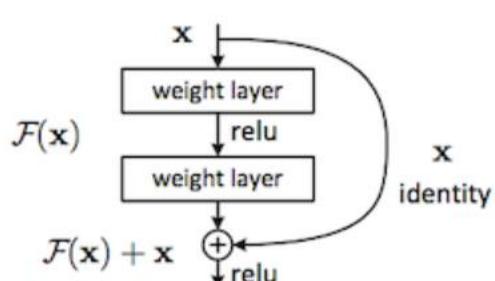
To tackle this, ResNet introduced a simple but powerful idea: Instead of hoping each few stacked layers directly fit a desired underlying mapping, we could have them learn the residual mapping.

Formally, denoting the desired underlying mapping as $\mathcal{H}(x)$, we want to let the stacked nonlinear layers fit another mapping $\mathcal{F}(x) = \mathcal{H}(x) - x$.

This can be also rearranged as:

$$\mathcal{H}(x) = \mathcal{F}(x) + x$$

from which comes the genial implementation idea of ResNet. In fact, this simple reformulation can be easily realized by feedforward neural networks via “skip connections”. In practice, the input x is passed forward unchanged (identity mapping) and added to the output of the stacked layers, producing the final result. The authors termed these identity shortcuts **residual connections**.



This makes it easier for the network to learn identity mappings when needed. If the optimal function at a certain stage is close to the identity, then the residual function $\mathcal{F}(x)$ just needs to be close to zero — a task that is often easier to optimize than fitting $\mathcal{H}(x)$ directly.

After adding these residual connections, the degradation problem disappeared. The authors were able to train extremely deep networks — over 100 layers — and achieved a top-5 error of just **3.57%** on ImageNet, **winning the ILSVRC 2015 competition**. They even trained networks with over 1000 layers, something previously considered infeasible. Interestingly, the 1000-layer model performed worse than the 100-layer version, and *this time*, the performance drop was due to overfitting: very low training error (<0.1%) but higher test error compared to 100-layer counterpart.

What made ResNet revolutionary was its simplicity: there was no complicated infrastructure behind it, but just some residual connections between layers. Notably these residual connections **add neither extra parameter nor computational complexity**, yet they radically improve the trainability of very deep networks.

Each combination of a function and a residual connection, such as $\mathcal{F}(x) + x$, is generally referred to as a **residual block**.

Note that the dimensions of x and $\mathcal{F}(x)$ must be equal in the equation above for the two to be added correctly. In fact, consider that sometimes the input and output dimensions could differ (e.g., when changing the input/output channels) and in such cases the simple matrix addition would not be possible. To address these cases there are two solutions:

1. **Linear projection:** perform a linear projection by multiplying an additional non-square matrix W of learnable parameters by the shortcut connection to match the dimensions, so the residual block becomes:

$$\mathcal{H}(x) = \mathcal{F}(x) + Wx$$

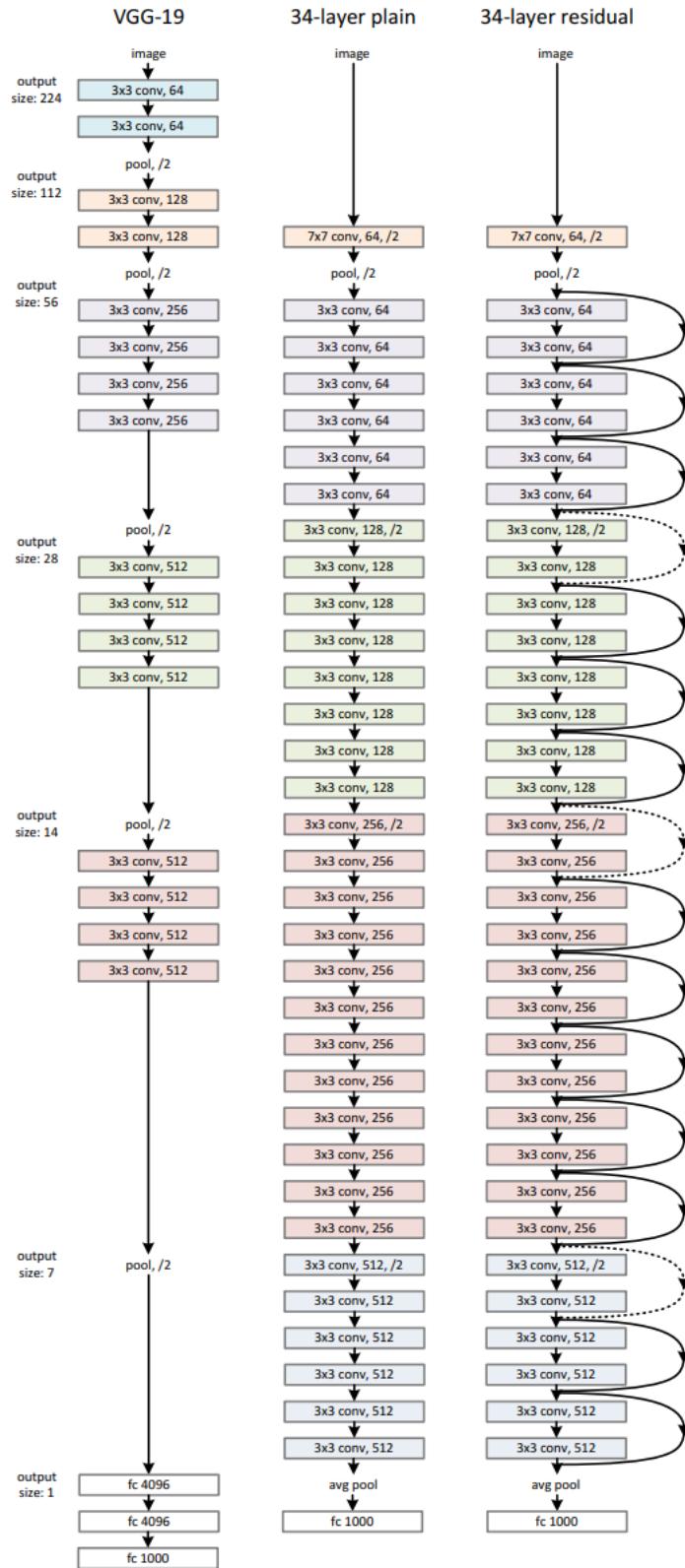
2. **Zero-padding:** apply padding to the input so its shape matches the output. This is often preferable because it doesn't introduce extra parameters, although the overhead of adding W is typically negligible in practice ([they are just few additional parameters](#)).

Both strategies are valid, and the choice depends on the design of the network.

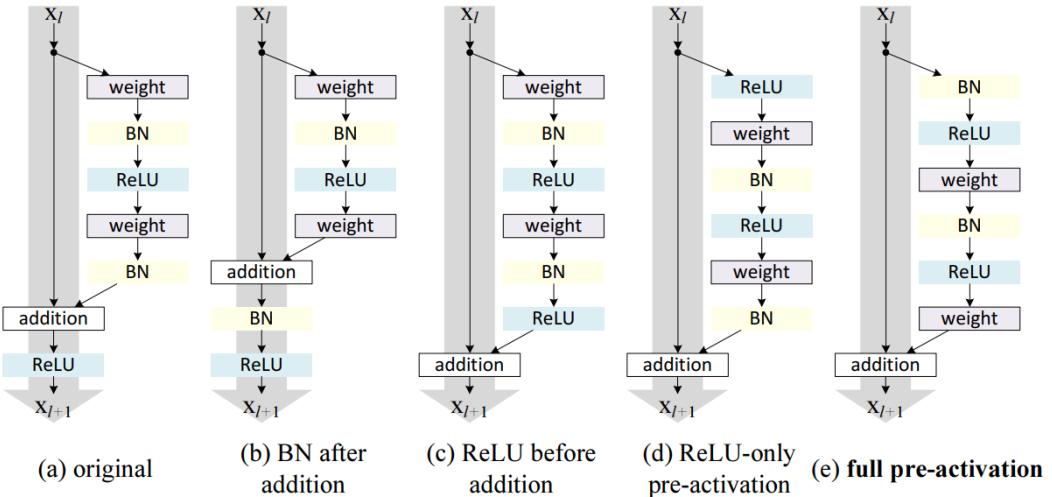
In the Figure below, we compare a **34-layer baseline convolutional network** inspired by VGG-19 (which is a VGG-16 with 3 additional convolutional layers at the end) with its **residual counterpart**, created by introducing **residual connections** every two layers.

In Figure solid lines represent **identity shortcut connections**, used when the input and output dimensions are the same, while dotted lines indicate shortcuts adjusted with **linear projection** or **zero-padding** used in cases where the dimensions of input and output don't match.

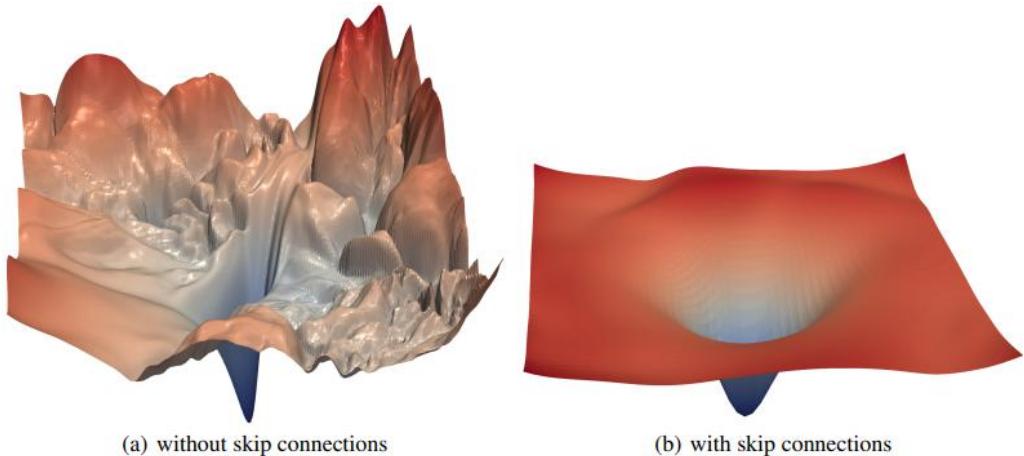
Building on this idea, the authors were also able to extend the architecture to even deeper networks leading to the famous **ResNet-101**. It uses a residual connection each three layers. These deeper models consistently outperformed their shallower counterparts (like the 34-layer version) by significant margins, without suffering from the degradation problem that had previously plagued deep networks.



A last important design consideration regards where to place the residual connections within the residual block. Several possibilities have been explored, as illustrated in Figure below. It was observed that the **"full pre-activation"** design, where both **Batch Normalization (BN)** and **ReLU** activation are applied **before** the weight layers, achieved the best results. This corresponds to the right-most residual block configuration shown in the Figure [28].



Li et al. [29] developed a way to visualize error surfaces directly, which showed that the effect of the residual connections is to create smoother error function surfaces, as shown in Figure below.



In conclusion, ResNet broke through a depth barrier that had previously limited deep learning models. Their design had a profound and lasting impact on modern DL architectures, going beyond image recognition and extending to a wide range of problems in both vision and non-vision domains.

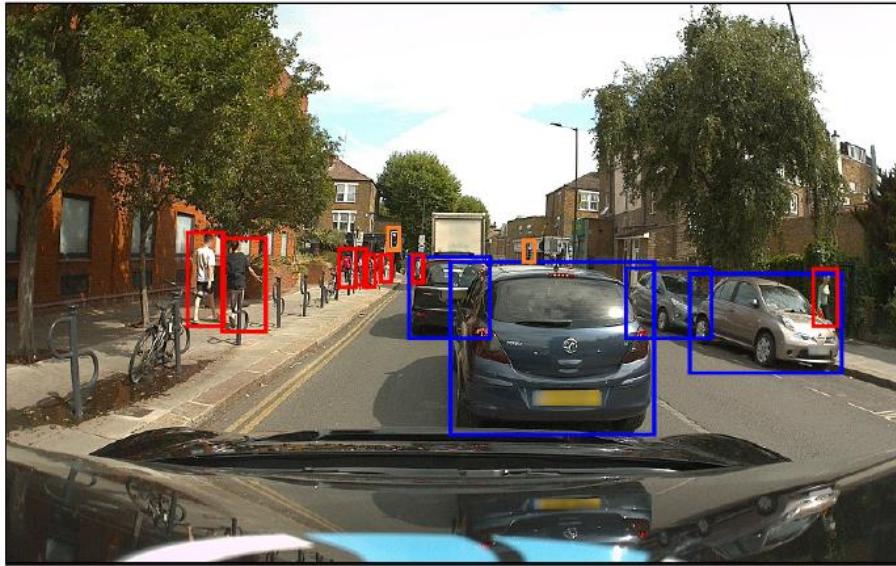
In fact, it's fair to say that without the foundational idea behind ResNet, we wouldn't have many of today's most advanced models that consist of thousands and thousands of layers — including LLMs.

Other Computer Vision Tasks

We have primarily motivated the design of CNNs through the image classification problem, where each image is assigned a single class label. This approach is well-suited to datasets like ImageNet, where each image is typically dominated by a single object. However, CNNs have a much broader range of applications, leveraging their inherent inductive biases. More generally, the convolutional layers of a CNN, trained on large image datasets for a specific task, can learn internal representations that are highly transferable. As a result, CNNs can be fine-tuned for a variety of specialized tasks ([we will talk more in detail about this idea in the end of the chapter](#)). Two of the most prominent of these tasks are *object detection* and *semantic segmentation*.

Object Detection

Many images contain multiple objects from one or more classes, and in such cases, we may want not only to classify the presence of each object but also to determine their precise locations within the image. This is the goal of **object detection**: predicting both *what* objects are present and *where* they are. This is achieved by drawing a **bounding box** around each object - a rectangle that tightly encloses the object, allowing us to localize it - and then assigning a class label for the object within that box, as illustrated in Figure.



Note: Here the bounding boxes were colored based on the object classes within them: blue boxes correspond to the class ‘car’, red boxes to the class ‘pedestrian’ and orange boxes to the class ‘traffic light’. Sometimes you can also find the explicit label placed on the corner of the bounding box.

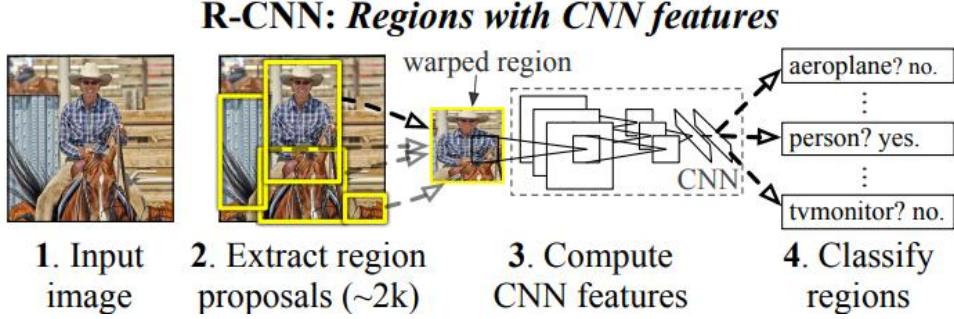
A bounding box is usually defined by a vector of four values: $b = (b_x, b_y, b_w, b_h)$, where b_x and b_y indicate the coordinates of the box center, while b_w and b_h denote its width and height. These values can be expressed either in pixels or as normalized values between 0 and 1, with the convention that (0, 0) corresponds to the top-left corner and (1, 1) to the bottom-right corner of the image.

In the simplest case, where an image is assumed to contain just one object from a predefined set of C classes, object detection can be approached by using a CNN with C outputs for classification (via a softmax layer), along with four additional outputs for predicting the bounding box coordinates. The network is trained using a **classification loss** (typically cross-entropy) and a **localization loss** (often MSE or smooth L1 loss) for bounding box regression.

However, this **single-object setup breaks down** when images contain **multiple objects**, especially if they belong to different classes. A model limited to one prediction might only detect one or two of the objects, **failing to detect the rest**, or even **confusing overlapping or nearby objects**. In such scenarios, the network’s output becomes ambiguous and loses the ability to **confidently disambiguate and localize all objects**, particularly when they vary in size, location or class.

To handle multiple objects, an early solution was the **sliding window** technique: a small, fixed-size window systematically moves across the image. At each location, the windowed region is fed into a CNN for classification, and if an object is detected with high confidence, the window’s position defines a bounding box. While this method aligns with the idea of CNNs scanning for features spatially, it’s **computationally expensive**, requiring a forward pass through the CNN at every possible position and scale.

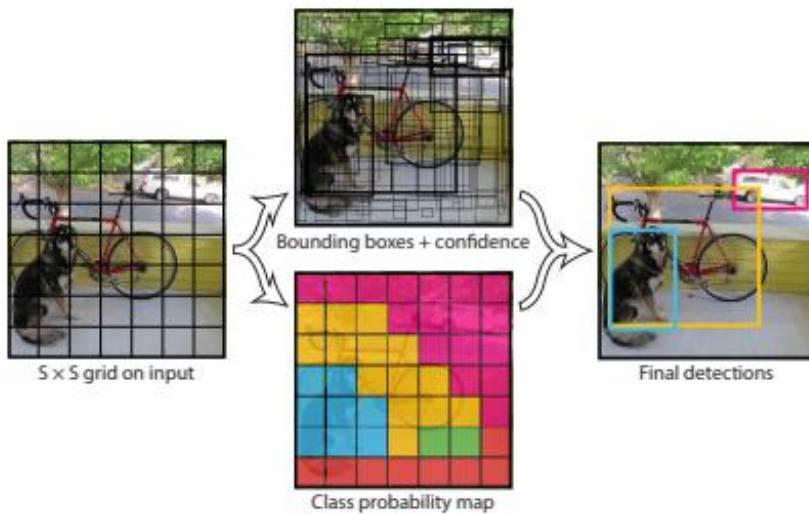
To reduce this cost, more advanced strategies were developed. Instead of scanning the entire image exhaustively, these approaches first identify a **smaller set of promising regions**, known as **region proposals**, which are then passed through the full CNN. This led to **R-CNN (Region-based CNN)** [30], which uses **Selective Search** to propose ~2000 candidate regions per image. Each region is then processed independently by a CNN to extract features and classify objects.



While more efficient than sliding windows, R-CNN was still **slow**, as it required running the CNN on each region separately. This inefficiency was addressed by **Fast R-CNN** [31], which computes convolutional features for the **entire image once**, then **reuses these features** to classify each region proposal via **region-of-interest (RoI) pooling**. This significantly sped up inference but still relied on the slow, hand-crafted Selective Search algorithm to generate proposals.

The breakthrough came with **Faster R-CNN** [32], which introduced the **Region Proposal Network (RPN)** — a lightweight CNN that **learns to generate object proposals** directly from feature maps. The RPN replaces Selective Search entirely and shares convolutional features with the detection network, enabling **end-to-end training** and significantly improving both speed and accuracy.

An even more streamlined approach was introduced by **YOLO (You Only Look Once)** [33]. YOLO reformulates detection as a **single regression problem**: the input image is divided into a grid, and for each cell, the network predicts a fixed number of bounding boxes along with class probabilities and confidence scores.



Importantly, YOLO **does not generate region proposals at all** — instead, it directly predicts objects in one forward pass through the network. This leads to extremely fast inference, suitable for **real-time applications**. However, this speed comes with trade-offs. Since each grid cell can predict only a limited number of bounding boxes, YOLO struggles with **detecting small or overlapping objects**, or scenes with **many objects** close together. These limitations may lead the model to **miss detections or reduce confidence** in its predictions. Nevertheless, later versions like YOLOv2, YOLOv3, YOLOv5 and beyond have addressed many of these issues and achieved remarkable **improvements in both speed and accuracy**.

Object detection continued to evolve rapidly, with models like **SSD (Single Shot MultiBox Detector)** [34] and **EfficientDet** [35] offering various trade-offs between computational efficiency and detection accuracy.

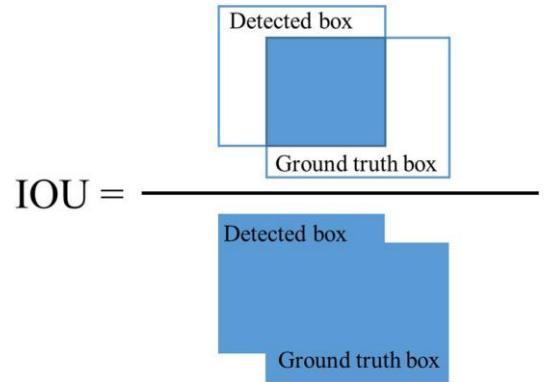
Though a full exploration of object detection is beyond the scope of this course I will stop here and conclude with just some other essential concepts that are fundamental to both evaluating object detectors and refining their predictions during inference:

- **Intersection over Union (IoU):** When a model is trained to predict bounding boxes, we need a meaningful way to evaluate how accurate those predictions are. In classification, this is relatively straightforward — we can use log-likelihood or accuracy based on predicted class probabilities. But in object detection, the key challenge is not only determining *what* object is present, but also *where* it is located.

To measure the alignment between a predicted bounding box and a ground-truth (annotated) bounding box, we use a metric called **Intersection over Union (IoU)**. This is defined as the ratio of the area of intersection between the predicted and ground-truth boxes to the area of their union:

$$IoU = \frac{\text{Area of Intersection}}{\text{Area of Union}} = \frac{|b_{pred} \cap b_{gt}|}{|b_{pred} \cup b_{gt}|}$$

The IoU value ranges from 0 to 1, where 1 means a perfect overlap and 0 means no overlap at all. A prediction is usually considered correct if its IoU with the ground truth is above a certain threshold (commonly 0.5). IoU penalizes both small overlaps and excessive regions outside the target, making it a more robust evaluation metric than using just the intersection area alone.



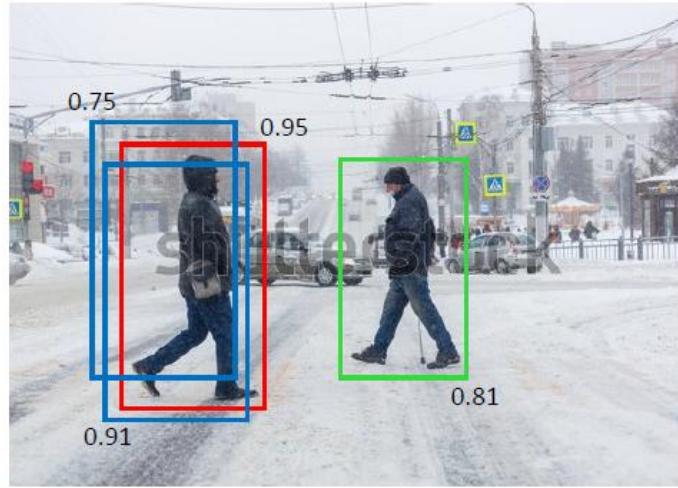
- **mean Average Precision (mAP):** Building on IoU, **mean Average Precision (mAP)** is widely used as a summary metric to evaluate object detection performance. It combines both **precision** (the proportion of predicted boxes that are correct) and **recall** (the proportion of ground-truth boxes that are detected), computed over various confidence thresholds and IoU thresholds. mAP provides a holistic view of both detection accuracy and coverage and is the standard metric in benchmarks such as PASCAL VOC and MS COCO.

- **Non-Maximum Suppression (NMS):** Object detectors typically generate multiple predictions for the same object — especially around its center, where many overlapping boxes might have high confidence. To eliminate this redundancy and output a clean set of distinct detections, a post-processing step called **Non-Maximum Suppression (NMS)** is applied at **inference time**. The process is straightforward:

1. For each object class, sort all predicted bounding boxes by their confidence scores in descending order.
2. Select the box with the highest confidence — this will be kept as a valid detection.
3. Compare this box with all remaining boxes using **IoU**. Discard any boxes that have a high IoU (e.g., > 0.5) with the selected one, as they likely refer to the same object.
4. Repeat this process with the next-highest scoring box and continue until all boxes are either kept or discarded.

The result is a set of **non-overlapping, high-confidence detections**, with only one bounding box retained per object instance. Importantly, NMS only removes boxes that significantly **overlap** — so if multiple

objects of the same class appear in different parts of the image (i.e., with low IoU between their boxes), they will all be preserved.



For instance, in the Figure above, **NMS will retain the red box**, which has the overall **highest confidence score**, and also **keep the green box**, which likely corresponds to a **different object** due to its **low IoU** with the red one. In contrast, **the blue boxes**, which have high overlap (IoU) with the red box but lower confidence, will be **suppressed** as redundant detections.

Image Segmentation

Image segmentation, in literature also known as **semantic segmentation**, takes this a step further: instead of classifying image regions it assigns a class label to every individual pixel in the image. The result is a dense, pixel-level classification map that allows us to identify the category of each object — and its precise location — within the scene.

To achieve this, the model outputs a feature map with the same spatial dimensions as the input image but with a different number of channels. While the input typically has three channels (RGB), the output **has as many channels as there are classes**. If there are C classes, the output is a C -channel feature map where each channel represents the predicted probability that a given pixel belongs to a specific class.

At inference time, the model's prediction can be visualized by assigning a unique color to each class and coloring each pixel according to the class with the highest predicted probability, as shown in Figure below.

Therefore, the network is trained using a **segmentation map** as target variable. This is a 2D array where each element corresponds to a pixel in the input image and contains the class index that pixel belongs to.



We have already seen that most CNNs use several levels of down-sampling — via strided convolutions or pooling — so that as the number of channels increases, the size of the feature maps decreases, keeping the overall size and cost of the network tractable while allowing the network to extract semantically meaningful high-order features from the image. However, segmentation requires that the final output has the same spatial resolution as the input. But how can we reach that?

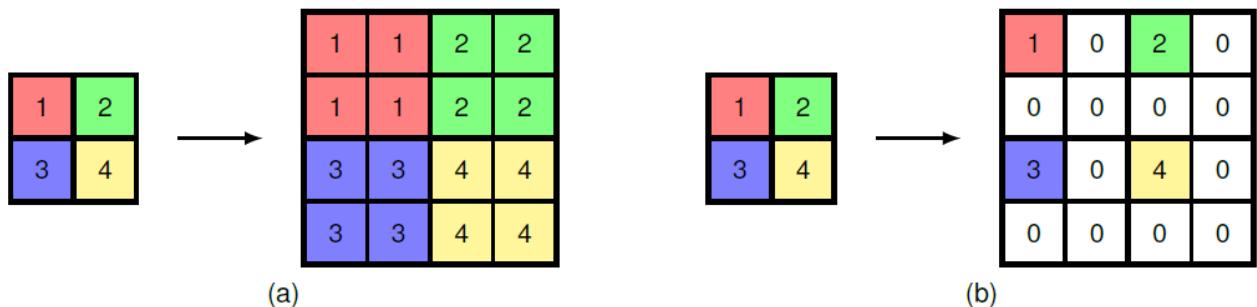
Early segmentation methods attempted to solve this using complex pipelines with proposal generation, region-based processing or post-processing steps such as Conditional Random Fields. A key breakthrough came with the use of **learnable upsampling convolutions** used to transform the low-resolution, high-dimensional representation back into a full-resolution segmentation map.

Up-sampling

Up-sampling is the process of increasing the spatial resolution of feature maps, essentially reversing the reduction in size caused by operations like strided convolutions and pooling. However, reversing these down-sampling operations is not straightforward. The challenge lies in how to reconstruct higher-resolution feature maps from smaller, compressed representations without losing important information.

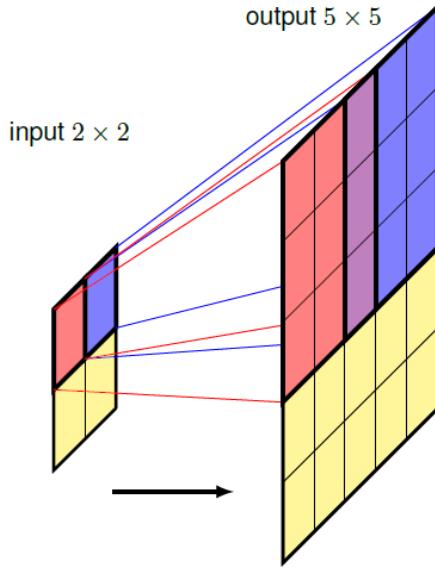
To better understand this, let's first look at an up-sampling analogue of pooling, where the output layer has a larger number of units than the input layer, for example with each input unit corresponding to a 2×2 block of output units. The question is then what values to use for the outputs. To find an up-sampling analogue of average pooling, we can simply copy over each input value into all the corresponding output units, as shown in Figure (a). We see that applying average pooling to the output of this operation regenerates the input.

For max-pooling, we can consider the operation shown in Figure (b) in which each input value is copied into the first unit of the corresponding output block and the remaining values in each block are set to zero. Again, we see that applying a max-pooling operation to the output layer regenerates the input layer.



Now, the up-sampling methods considered above are fixed functions, much like the average-pooling and max-pooling down-sampling operations. We can also use a **learned up-sampling** that is analogous to strided convolution for down-sampling.

In strided convolution, each unit on the output map is connected via shared learnable weights to a small patch on the input map, and as we move one step through the output array, the filter is moved two or more steps across the input array, and hence the output array has lower dimensionality than the input array. For up-sampling, we can think of using a filter that connects one pixel in the input array to a patch in the output array and then choose the architecture so that as we move one step across the input array, we move two or more steps across the output array. This is illustrated for 3×3 filters and an output stride of 2 in the Figure below. Note that there are output cells for which multiple filter positions overlap, and the corresponding output values can be found either by summing or by averaging the contributions from the individual filter positions.

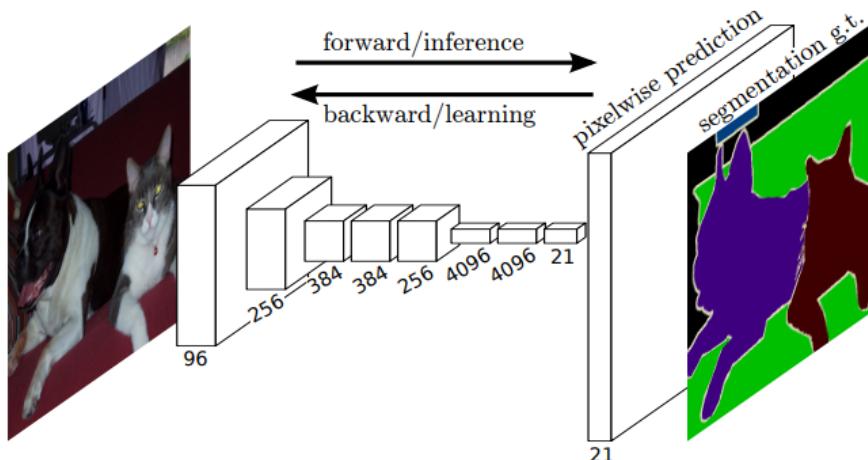


This up-sampling is called ***transpose convolution*** because, if the down-sampling convolution is expressed in matrix form, the corresponding up-sampling is given by the transpose matrix. Note that this is sometimes also referred to as '***deconvolution***', but it is better to avoid this term since deconvolution is widely used in mathematics to mean the inverse of the operation of convolution used in functional analysis, which is a different concept.

An important aspect that should be underlined is that, unlike average or max upsampling, transpose convolutions are parameterized and learned from data.

Fully Convolutional Networks (FCNs)

The 2015 paper “*Fully Convolutional Networks for Semantic Segmentation*” [36] introduced a groundbreaking approach to semantic segmentation. The authors proposed **Fully Convolutional Networks (FCNs)**, which can take input images of any size and produce dense, pixel-level output maps of matching dimensions. This design enabled **end-to-end training** without relying on complicated post-processing steps.

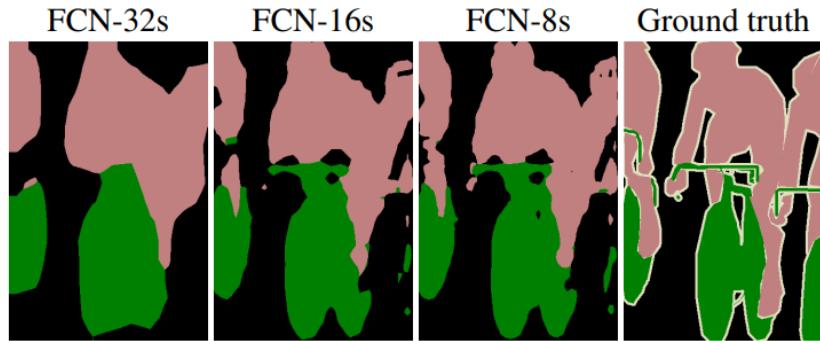


Before FCNs, popular classification networks like AlexNet and VGG ended with fully connected layers, which required fixed-size inputs and were unsuitable for producing per-pixel predictions. FCNs overcame this limitation by converting the fully connected layers of pretrained classification models, such as VGG16, into **1×1 convolutional layers**. This transformation allowed the networks to process images of arbitrary size while maintaining spatial information throughout the architecture. After converting the fully connected layers into 1×1 convolutions, the authors added an additional 1×1 convolutional layer with 21 output channels, corresponding

to the 20 object classes of the PASCAL VOC dataset plus a background class. This produced a coarse class score map at roughly 1/32 of the input image resolution. To recover the original resolution, a single transposed convolution with a stride of 32 was applied to upsample the score map, yielding dense, per-pixel predictions.

While this basic upsampling restored spatial dimensions, it lacked fine-grained spatial precision. As the network processed the image through multiple downsampling layers, valuable spatial details were lost — acceptable for classification but problematic for dense prediction tasks like image segmentation. To address this, FCNs introduced **skip connections** that fuse the high-resolution, low-level features from earlier layers with the upsampled coarse outputs, enhancing localization accuracy.

The authors experimented with three progressively more refined versions of the network named FCN-32s, FCN-16s and FCN-8s which differ in how they combine features from different layers to improve spatial accuracy.



The simplest version, FCN-32s, is the standard network that we saw above without skip connections. Therefore, it takes the coarse output from the final layer — at about 1/32 of the original image resolution — and directly upsamples it using a single transposed convolution to produce the full-resolution segmentation map. The resulting segmentation overly coarse due to lost spatial details.

To improve on this, FCN-16s incorporates an additional step: it fuses the upsampled coarse output with features extracted from an earlier layer in the network, specifically from the layer known as pool4, which retains spatial information at 1/16 resolution. Before merging, pool4 features are passed through a 1×1 convolution to match the number of classes (21 channels). The network then combines these features by summing them element-wise with the upsampled final output. This fused map, now at a higher resolution, is upsampled once more to produce the final full-resolution segmentation. By integrating these finer features, FCN-16s achieves better localization of object boundaries compared to FCN-32s.

Taking this idea even further, FCN-8s introduces another skip connection from an even earlier layer, pool3, which operates at 1/8 of the input resolution. Like with pool4, these pool3 features are first transformed via a 1×1 convolution to the class dimension and then fused by element-wise addition with the upsampled output of FCN-16s. After this fusion, the resulting feature map is upsampled by a factor of 8 to restore the original input size. By progressively combining information from multiple depths of the network, FCN-8s produces more precise and detailed segmentations, especially along object edges and smaller structures.

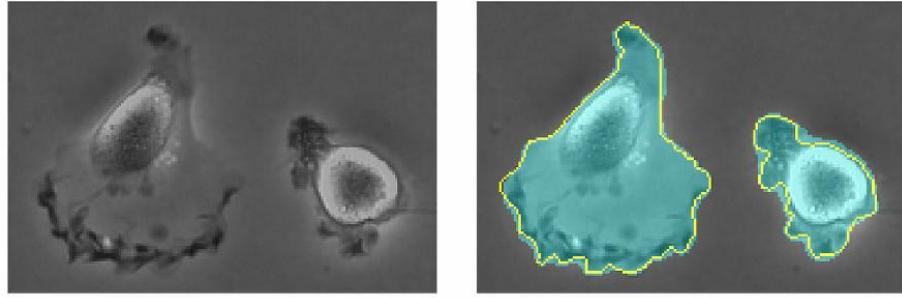
Through these variants, the paper demonstrated that combining coarse semantic information from deeper layers with finer spatial details from earlier layers leads to significant improvements in segmentation quality.

Note: The term **fully convolutional** reflects the fact that all layers in the network — both for down-sampling and up-sampling — are implemented using just convolutions. A network such that can take an arbitrarily sized image and will output a segmentation map of the same size.

U-Net

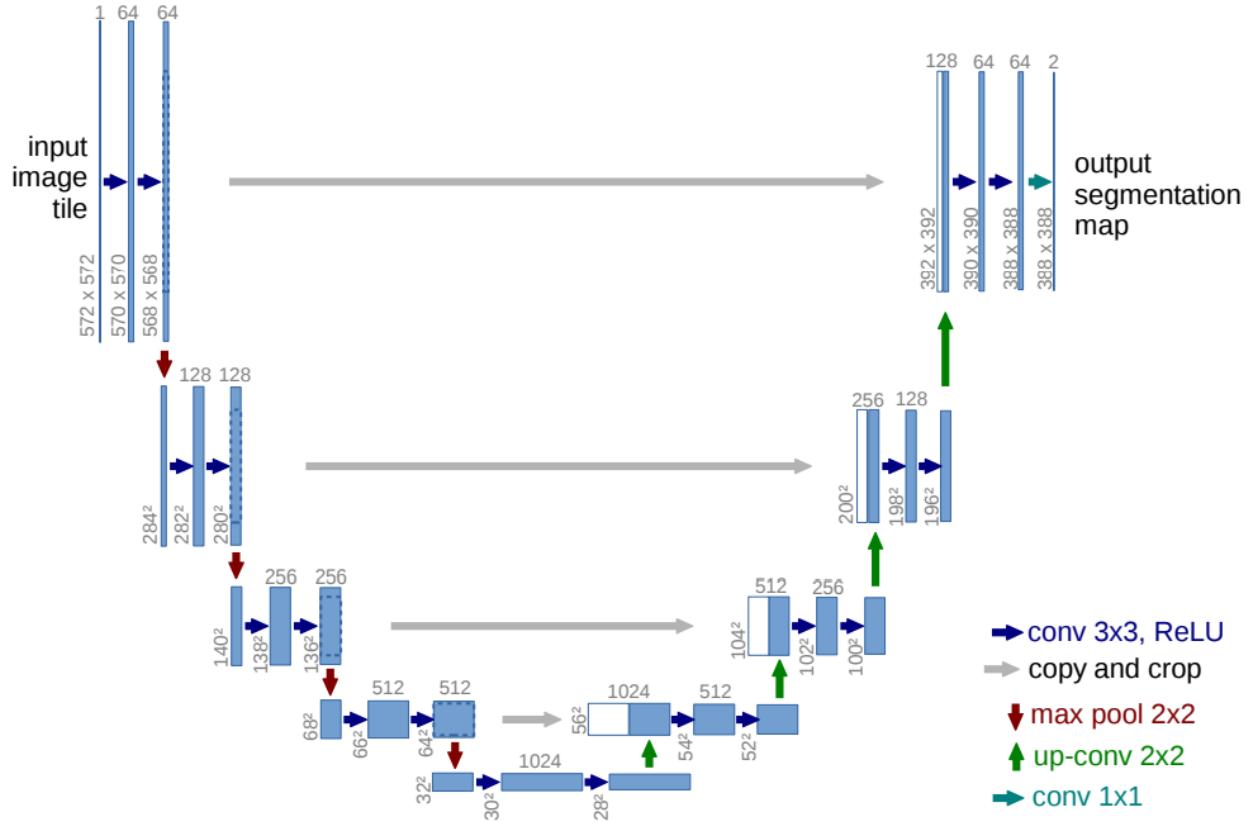
U-Net, introduced by Ronneberger et al. (2015) in the paper “*U-Net: Convolutional Networks for Biomedical Image Segmentation*” [37], extend the FCN architecture with some modifications to further emphasize the preservation and recovery of spatial information lost during down-sampling.

Originally developed for biomedical image segmentation — specifically for accurately segmenting cells in medical images — U-Net has since been widely adopted across various domains for its ability to produce precise, high-resolution segmentations.



Note: An interesting fact is that U-Net performs well even when **trained on very few images**, making it especially useful in fields where annotated data is scarce.

The U-Net architecture is named because of its characteristic U-shaped structure, illustrated in Figure.



The core concept is that for each down-sampling layer in the **contracting path** there is a corresponding upsampling layer in the **expansive path**, and the final set of channel activations at each down-sampling layer is

concatenated with the corresponding first set of channels in the up-sampling layer via **skip connections**, thereby giving those layers access to higher-resolution spatial information that was preserved earlier in the network.

A 1×1 convolutions is used in the final layer of U-net to reduce the number of channels down to the number of classes, which is then followed by a softmax activation function.

Note: The network does not include any FC layer, just convolutions and max-pooling in the contracting path and transpose convolutions in the expansive path.

Important Clarifications About U-Net

Before concluding, it is crucial to address two widespread misconceptions about the U-Net architecture:

- **U-Net Does Not Use Residual Connections:** A common misunderstanding is to confuse **skip connections** with **residual connections**, but these are fundamentally different concepts.
 - **Skip connections** broadly refer to any shortcut connections in a neural network that bypass one or more layers, feeding outputs directly to later layers. In the case of U-Net, skip connections specifically link corresponding layers in the *contracting path* and the *expansive path* that share the same spatial resolution. These connections **concatenate** feature maps, allowing the network to recover fine-grained spatial information lost during down-sampling.
 - **Residual connections**, on the other hand, are a special type of skip connection that perform **element-wise addition** between the input to a layer and its output, primarily to facilitate gradient flow and enable training of very deep networks.
- **U-Net and Autoencoders Are Two Different Concepts:** In a later chapter, we will analyze how autoencoders work. At first glance, U-Net might resemble the symmetrical architecture of an autoencoder, but they are fundamentally different in both purpose and function:
 - Autoencoders aims to learn meaningful, lower-dimensional representations (known also as latent representation) of the input and for doing that they accomplish a reconstruction task: the input is “compressed” into a latent representation using an encoder and then expand it back using a decoder so that the goal is for the output to closely resemble the original input. Therefore, the motivation behind an autoencoder to re-expand the lower-dimensional representation is just to permit the reconstruction task (which is an **unsupervised task**). On the other hand, the motivation for U-Net to re-expand the feature maps is to produce an output of the same size as the input in order to perform pixel-wise classification — a **supervised task**. In fact, U-Net is built more directly on the foundation of Fully Convolutional Networks (FCNs) rather than autoencoders.
 - Moreover, in autoencoders, **encoder and decoder are separate components**: the encoder transforms each input sample into a lower-dimensional representation, and the decoder reconstructs the output solely from this representation. This clear separation allows the encoder and decoder to be used independently — for example, using the encoder for feature extraction and the decoder for generation. For autoencoders, this separation is crucial; otherwise, it defeats the entire purpose of image compression.In U-Nets, however, **this is not the case**. There, the output mapping also depends directly on the input space via **skip connection**. This means there are no real "encoder" and "decoder" parts, in the sense of mapping the sample onto some well-defined latent space and then computing the output from it. You can NOT split a U-Net into parts and use them separately!

Transfer Learning

Once a CNN has been trained on a large-scale dataset such as ImageNet, the internal representations it learns — particularly in the earlier layers — are not only effective for the original task but also broadly transferable to other related tasks. This observation underpins **transfer learning**, a technique that enables a model trained on one task to be adapted to another, often with significantly less labeled data. It is particularly valuable in settings where the target task suffers from data scarcity.

To better understand this, consider how CNNs process images. The early layers tend to learn general-purpose features such as edges, textures or basic shapes, which are common across many visual domains. In contrast, the deeper layers tend to specialize in features that are more task specific. The key idea behind transfer learning is that these low-level, learned features can be retained and reused when training a model for a new task. Rather than training a model from scratch — an approach that requires large volumes of labeled data and extensive computational resources — one can instead take a pre-trained model and adapt it to the new task using significantly less data and training time.

This approach typically involves two main stages: **pre-training** and **fine-tuning**.

Pre-training refers to the initial phase in which a CNN is trained on a large and generic dataset for a general-purpose task, such as classification across thousands of object categories (e.g., ImageNet). This results in a network whose parameters encode rich and versatile representations of visual data. These parameters then serve as a starting point for the second stage: **fine-tuning**.

In its simplest form, fine-tuning is applied to a new dataset for the *same task* as the one used in pre-training. For example, a network pre-trained on ImageNet for image classification can be fine-tuned on a smaller dataset such as CIFAR-10, which also involves classifying images but contains fewer classes and far less labeled data. This can be reached by freezing the convolutional layers of the pre-trained network (i.e., their weights are kept fixed) and training from scratch only the final FC layers using the new dataset. This strategy is particularly effective when labeled data for the new dataset is limited, as it reduces the risk of overfitting by retaining the robust general-purpose features learned during pre-training. If more data is available, it can be beneficial to unfreeze and update also some of the deeper convolutional layers, allowing the network to gradually adjust its representations to better suit the specifics of the new task.

Fine-tuning can also be applied to adapt the model to a **downstream task** — a new task that follows the initial pre-training and may involve a different objective altogether. For instance, a network pre-trained for image classification can be fine-tuned for object detection or semantic segmentation. Although the goals of these tasks differ from those of the original classification task, they often share the same input modality (e.g., natural images), which allows the model to reuse its earlier learned features. This reuse not only improves performance but also reduces the training time and data requirements for the downstream task.

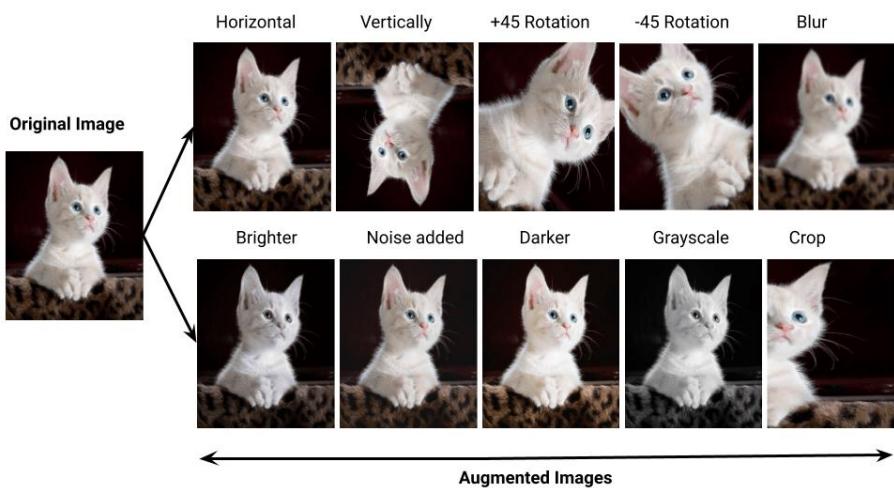
Fine-tuning is typically performed with a lower learning rate and fewer iterations, to ensure that the early acquired features are only subtly modified rather than entirely overwritten.

Importantly, transfer learning is a **general paradigm** that extends well beyond computer vision tasks. In natural language processing, for example, models like BERT and GPT are pre-trained on massive text corpora on tasks such as next-word or masked-word prediction and later fine-tuned for specific downstream tasks like sentiment analysis, question answering or summarization. Similar approaches are applied in speech processing and other domains. Across all of these fields, the guiding principle remains the same: begin with a model that has learned useful, general-purpose features through pre-training and then refine it through fine-tuning to meet the demands of a specific downstream task. This strategy not only accelerates learning and reduces data requirements but also provides a robust and scalable foundation for a wide array of real-world applications. Therefore, in this paradigm, a model trained on a broad and resource-intensive task can serve as a powerful foundation for more specialized tasks.

Data Augmentation

When classifying an object in images, the assignment of a particular class should be independent of the object's position in the image, i.e., it should be translation invariant. Similarly, changes in the object's size within the image should not affect its classification, thus ensuring scale invariance. Exploiting such symmetries to create inductive biases can significantly improve the performance of DL models. **Data augmentation** provides an efficient and practical way to **encourage these invariances without explicitly modifying the model architecture**.

This technique works by expanding the training set with transformed replicas of the original data, applying transformations consistent with the desired invariances, while preserving the original labels. In doing so, data augmentation effectively acts as a form of **regularization, reducing the risk of overfitting** by exposing the model to a broader set of variations and making it less likely to memorize spurious details of the training data.



It is often beneficial to consider multiple invariances simultaneously in addition to translation and scale invariance. Therefore, various techniques are employed, including flipping, cropping, translations, rotations, intensity and color balance changes, noise injections and others. The most suitable techniques to use often depend on the use case being considered. In recent years, beyond basic transformations, more advanced techniques such as CutMix, MixUp and adversarial augmentation [38] have been developed to enrich data in more sophisticated ways.

Note: It is important to be cautious with the choice of augmentation methods to apply, since certain augmentations can inadvertently alter the semantics of the input — for instance, flipping a handwritten digit 6 vertically might turn it into a 9.

Although data augmentation originated and is most widely used in computer vision, the concept extends to other domains such as audio, text and tabular data, where domain-specific transformations can be similarly beneficial.

Recurrent Neural Networks

Deep neural networks we have seen so far were designed all under a fundamental assumption: each input example is processed **independently** of the others. After handling one input, the network's internal state is reset before processing the next. While this approach works well for tasks where examples are unrelated — like recognizing objects in images — it falls short when we deal with **sequential data**. In many real-world tasks, inputs are not independent: understanding one element often depends on those that came before it. Examples include words in a sentence or frames in a video. Just to cite another, standard networks are built to handle inputs of **fixed size** (e.g. images) making it difficult to process sequences of varying lengths — like sentences of different word counts or videos of different durations. Therefore, it is desirable to extend these models to handle data with sequential structure.

This is precisely the motivation behind **Recurrent Neural Networks (RNNs)** [39]. RNNs were originally developed to process **sequential text data** — since text offers one of the most intuitive ways for thinking about sequential dependencies — but, as already anticipated above, they can be applied to any domain where inputs are sequentially structured.

Therefore, before we dive into how RNNs work, we will first make a little digression on text data as it is important to understand how we can handle this type of data. After that, we will delve into how RNNs operate and how they address the above limitations.

Text Data: What Makes Text Different?

Among the different types of sequential data we may encounter, **text** is one of the most prominent and widely studied. Text data is fundamentally sequential — words appear in a specific order, and understanding a given word often requires knowledge of the words that came before it. This makes text a natural domain for exploring how models process sequences over time. However, text is not just any sequence. It belongs to the broader domain of **natural language**, where the input consists of words, phrases, sentences and documents that are rich in structure, context and meaning.

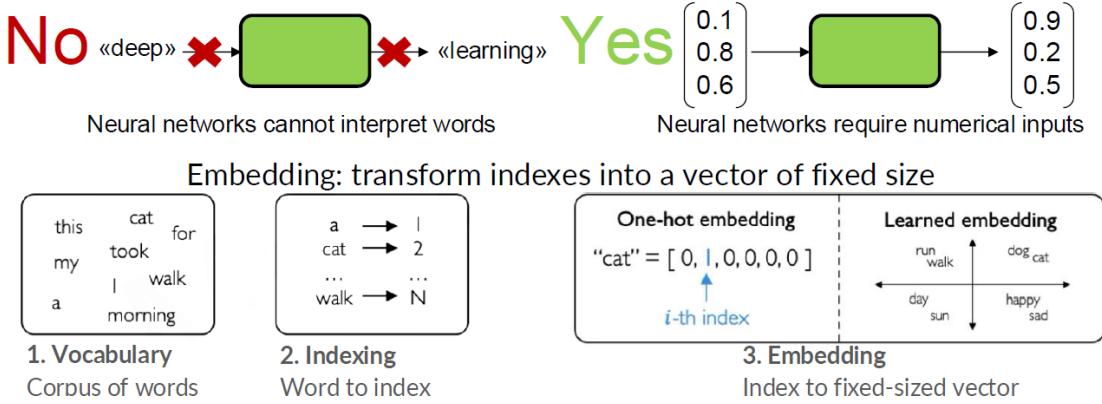
Natural language introduces unique challenges. Unlike simple sequences of numbers or symbols, language is inherently ambiguous, context-dependent and shaped by human logic and culture. Words may have multiple meanings depending on context, and subtle variations in word order or phrasing can drastically change a sentence's intent. This complexity is what makes the field of **Natural Language Processing (NLP)** both fascinating and essential. NLP aims to develop algorithms and models that allow machines to understand, interpret and generate human-like language.

Applications of NLP are now deeply embedded in many technologies we use daily. For instance, *sentiment analysis* seeks to determine whether a piece of text conveys positive or negative emotion. *Machine translation* enables communication across languages. *Named entity recognition* identifies mentions of people, places or organizations within a sentence. *Text summarization* reduces long articles into shorter, informative summaries.

First, there is a challenge that needs to be addressed: how can we convert raw text into a format that DL models can work with? We can't feed words as they are into a deep neural network as it expects numerical representations as input. Therefore, we need to represent words as **numerical vectors**.

One simple approach could be to define a fixed **dictionary** (sometimes also referred as **vocabulary**) of all the words we can encounter in a corpus of text and then adopt **one-hot encoding**, i.e. introduce vectors of length equal to the size of the dictionary along with a 'one-hot' representation for each word, in which the k -th word in the dictionary is encoded with a vector having a 1 in position k and 0 in all other positions. For example, if 'king' is

the third word in our dictionary then its vector representation would be $[0,0,1,0, \dots, 0]$. An obvious problem with a one-hot representation is that a realistic dictionary might have several hundred thousand entries leading to vectors of very high dimensionality. Also, it does not capture any similarities or relationships that might exist between words. Both issues can be addressed by mapping each dictionary's word into a lower dimensional vector (typically a few hundred dimensions), often referred as **word embedding**.



Word Embedding

Embeddings building process starts by defining a matrix W of size $N \times V$, where N is the dimensionality of the embedding space and V is the dimensionality of the dictionary. Then, for each one-hot encoded input vector x_i we can calculate the corresponding embedding using:

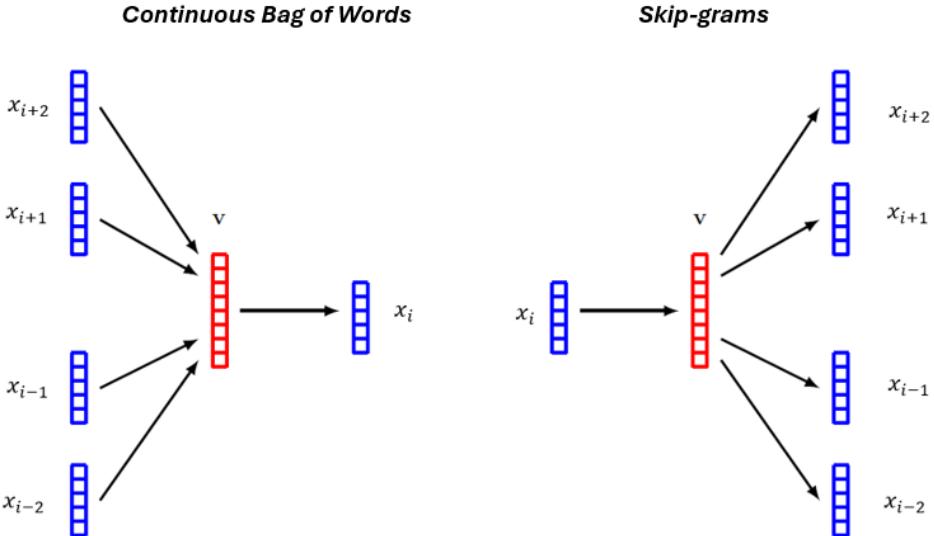
$$v_i = Wx_i$$

All we need is to **learn a meaningful embedding matrix W** starting from data (e.g. a corpus of text). There are many approaches for doing this, for instance in the ML course we studied a popular method called *Word2Vec*. Here, we'll briefly recap the core idea of that approach, assuming you're already familiar with the general concept. For a more detailed explanation, please refer to the *Embeddings* section of the [ML Course notes](#).

Note: In the ML course, we represented one-hot vectors as **columns**, whereas here we represent them as **rows**. As a result, the weight matrix W appears transposed, and some dot product operations are reversed in order. However, the underlying concepts remain the same.

Word2Vec [40] adopts a simple two-layer neural network to learn the embedding matrix W . A training set is constructed in which each sample is obtained by considering a ‘window’ of M adjacent words in the text, where a typical value might be $M = 5$. There are two variants of this approach. In **Continuous Bag of Words**, the target variable for network training is the middle word and the remaining context words form the inputs, so that the network is being trained to ‘fill in the blank’. A closely related approach, called **Skip-grams**, reverses the inputs and outputs, so that the center word is presented as the input and the target values are the context words. These models are illustrated in the Figure below.

Once the model is trained, the embedding matrix W is given by the transpose of the second-layer weight matrix for the continuous bag-of-words approach and by the first-layer weight matrix for skip-grams. **Words that are semantically related are mapped to nearby positions in the embedding space**. This is to be expected since related words are more likely to occur with similar context words compared to unrelated words. For example, the words ‘city’ and ‘capital’ might occur with higher frequency as context for target words such as ‘Paris’ or ‘London’ and less frequently as context for ‘orange’ or ‘polynomial’. The network can more easily predict the probability of the missing words if ‘Paris’ and ‘London’ are mapped to nearby embedding vectors.



It turns out that the learned embedding space often has an even richer semantic structure than just the proximity of related words, and that this allows for simple *vector arithmetic*. For example, this led to the famous ‘magic’ word analogy task where the concept that ‘*king is a man as queen is a woman*’ can be expressed through operations on the embedding vectors. If we use $v(\text{word})$ to denote the embedding vector for ‘word’, then we find:

$$v(\text{king}) - v(\text{man}) + v(\text{woman}) \approx v(\text{queen})$$

Word embeddings were originally developed as natural language processing tools in their own right. Today, they are more likely to be used as pre-processing steps for deep neural networks. In this regard they can be viewed as the first layer in a deep neural network. They can be fixed using some standard pre-trained embedding matrix, or they can be treated as an adaptive layer that is learned as part of the overall end-to-end training of the system. In the latter case the embedding layer can be initialized either using random weight values or using a standard embedding matrix.

Tokenization

One problem with using a fixed dictionary of words is that it cannot cope with words not in the dictionary or which are misspelled. It also does not take account of punctuation symbols or other character sequences such as computer code. An alternative approach that addresses these problems would be to work at the level of characters instead of using words, so that our dictionary comprises upper-case and lower-case letters, numbers, punctuation and symbols such as spaces and tabs. A disadvantage of this approach, however, is that it discards the semantically important word structure of language, and the subsequent neural network would have to learn to reassemble words from elementary characters. It would also require a much larger number of sequential steps for a given body of text, thereby increasing the computational cost of processing the sequence.

We can combine the benefits of character-level and word-level representations through a pre-processing step known as tokenization. Tokenization transforms a sequence of text — including words and punctuation — into a sequence of tokens, which are typically small units of meaning. These tokens may include common words in their entirety, along with fragments of longer words as well as individual characters that can be assembled into less common words. This approach allows the model to efficiently handle both frequent and rare words by breaking them into manageable, reusable components.

An additional benefit is that variations of the same word can have related representations. For example, ‘cook’, ‘cooks’, ‘cooked’, ‘cooking’ and ‘cooker’ are all related and share the common element ‘cook’, which itself could be represented as one of the tokens.

Note: Tokenization is a flexible concept that extends beyond text to other modalities. For example, in the Transformer chapter we'll see how a similar idea can be applied to images in the **Vision Transformer (ViT)** architecture, where image patches are treated as tokens.

There are many approaches to tokenization. As an example, a technique called **Byte Pair Encoding (BPE)** [41] that is used for data compression, can be adapted to text tokenization by merging characters instead of bytes. It starts with the individual characters and iteratively merges them into longer strings.

How does BPE works?

The list of tokens is first initialized with the list of individual characters. Then a body of text is searched for the most frequently occurring adjacent pairs of tokens and these are replaced with a new token. To ensure that words are not merged, a new token is not formed from two tokens if the second token starts with a white space. The process is repeated iteratively. Initially the number of tokens is equal to the number of characters, which is relatively small. As tokens are formed, the total number of tokens increases, and if this is continued long enough, the tokens will eventually correspond to the set of words in the text. The total number of tokens is generally fixed in advance, as a compromise between character-level and word-level representations. The algorithm is stopped when this number of tokens is reached.

The Figure below illustrates an example of the process of tokenizing natural language text with BPE. In this case, the most frequently occurring pair of characters is `<p>` and `<e>`, which occurs four times and so these form a new token `<pe>` that replaces all the occurrences of those two characters. Note that `<Pe>` is not included in this since uppercase `<P>` and lower-case `<p>` are distinct characters. Next the token `<ck>` is added since the pair `<c>` and `<k>` occurs three times. This is followed by tokens such as `<pi>`, `<ed>` and `<per>`, all of which occur twice, and so on.

Peter Piper picked a peck of pickled peppers
Peter Piper picked a peck of pickled peppers

More advanced tokenization techniques have been developed to better capture the complexities of natural language. Popular NLP libraries such as **SpaCy** provide a lot of additional robust and customizable tokenization functionalities.

Once the input text has been tokenized, it is common in practical applications to introduce additional **special tokens** such as `<start>`, `<eos>`, `<pad>`, `<cls>`, etc. These *special tokens* serve specific functional roles within a model. We will explore the purpose of many of these tokens in more detail in the Transformer chapter.

To summarize, in modern deep learning pipelines for natural language processing, the raw input text is first transformed into a sequence of tokens via a suitable tokenization algorithm. Special tokens may then be added to this sequence as needed for the task. Finally, the tokens are converted into embeddings which serve as the actual input to neural models.

How to Handle Sequential Data?

To introduce this concept, we will focus on text data. In the next for simplicity, we use word-level representations (not tokens) as this makes it easier to illustrate and motivate key concepts.

Let's consider a classic sequence modeling task: **predicting the next word in a sequence**: given a sequence of words, we would like to predict the word that comes next.

"This morning I took my cat for a walk."
given these words predict the next word

How could we tackle this kind of problem?

Idea #1: Use a fixed window

A simple idea is to predict the next element in a sequence based only on a **fixed number of previous elements** — for example, using the last two words to predict the next word.

"This morning I took my cat for a walk."
given these predict the
two words next word

Note: Just to say, each word can be represented using *one-hot encoding*

[1 0 0 0 0 0 1 0 0 0]
for a
 ↓
 prediction

or as a *word embedding* (obtained for instance with Word2Vec).

Problem #1: Can't capture long-term dependencies

Using a small, fixed window only gives the model access to a limited context. However, many tasks require information from the distant past to accurately predict the correct word.

Example:

"France is where I grew up, but I now live in Boston. I speak fluent ____."

To predict "French," the model needs to remember information from the distant past ("France"), not just the last few words.

Idea #2: Use Bag-of-Words

Another approach is to represent the sequence as a **bag of words**, a vector representation where each position corresponds to a word in a predefined dictionary and the value at each position indicates the count of its occurrences in the input sentence.

"This morning I took my cat for a walk."
↓
Bag of words
[1 0 0 1 0 1 1 0 0 1 0 0 0 1 1 0 1]
↓
prediction

Problem #2: Counts don't preserve order

This method completely discards word order, which can be crucial for meaning.

Example:



The food was good, not bad at all.

vs.



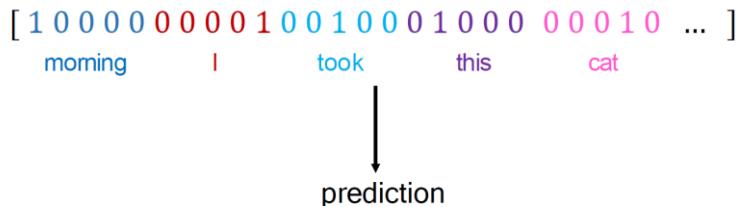
The food was bad, not good at all.

A bag-of-words model would see these two sentences as identical, even though they have opposite meanings.

Idea 3: Use a really big fixed window

If a small window is too narrow, what about using **a very large window** — perhaps the entire sequence so far?

"This morning I took my cat for a walk."



Problem #3: No parameter sharing

With this approach, every position in the window would have its own set of parameters. Things we learn about the sequence won't transfer if they appear elsewhere in the sequence.

Example:

[1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 ...]
this morning took the cat

[0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 1 ...]
this morning

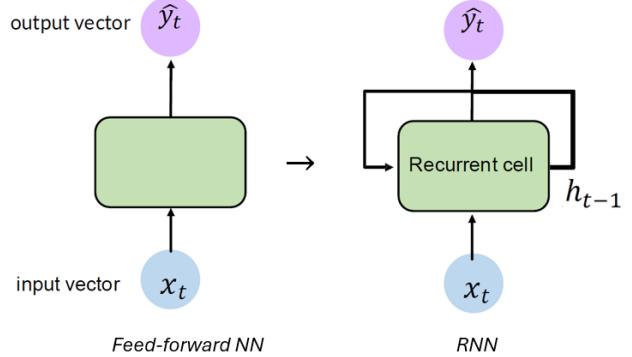
The shortcomings of these naive approaches highlight the need for a better model. **RNNs** solve these problems by introducing two key innovations:

- **A shared hidden state** that efficiently stores information across time.
- **Non-linear dynamics** that allow to update the hidden state in complex ways.

In extended forms like **LSTMs** (Long Short-Term Memory networks) and **GRUs** (Gated Recurrent Units), RNNs are even better equipped to handle long-range dependencies, selectively remembering or forgetting information through the sequence. More of this in later sections.

Recurrent Neuron

In a feed-forward neural network, each neuron only receives input from neurons in the previous layer and sends output to neurons in the next layer. **Recurrent Neural Networks (RNNs)** modify this structure by introducing cyclic connections known as **recurrent connections**. Each neuron in an RNN can send outputs not just to the next layer, but also **back to itself**. In particular, recurrent connections dynamically transfer information across adjacent time steps, potentially forming “self-connections” from a neuron to itself over time. This effectively gives each neuron a form of memory allowing them to process information from the past.



To implement this, an RNN neuron introduces a hidden state h associated with each time step that stores information about the sequence up to the previous time step. The hidden state time by time is updated based on the current input and the past hidden state, allowing information to flow through the sequence.

```
my_rnn = RNN()
hidden_state = [0, 0, 0, 0]

sentence = ["I", "love", "recurrent", "neural"]

for word in sentence:
    prediction, hidden_state = my_rnn(word, hidden_state)

next_word_prediction = prediction
# >>> "networks!"
```

At each time step t , the RNN updates its hidden state by combining the current input $x^{(t)}$ with the previous hidden state $h^{(t-1)}$. The updated hidden state $h^{(t)}$ is then linearly projected to produce the output vector $y^{(t)}$:

$$h^{(t)} = \tanh(x^{(t)}W_{xh} + h^{(t-1)}W_{hh} + b_h)$$

$$y^{(t)} = h^{(t)}W_{hy} + b_y$$

where W_{xh} represents the weight matrix between the input and the hidden state, W_{hh} denotes the recurrent weight matrix between the hidden state and itself across adjacent time steps, W_{hy} is the weight matrix mapping the hidden state to the output, and b_h and b_y are bias vectors for the hidden and output layers, respectively.

These two steps — updating the hidden state and producing the output vector — are the fundamental computations that always take place during the RNN forward pass. Then, in case we want to perform a specific task such as **next-word prediction**, we further apply a softmax to obtain a probability distribution over the vocabulary:

$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$

where $\hat{y}^{(t)}$ represents the predicted probability distribution (we will explore this in more detail with an example when discussing Transformers). In other tasks (e.g., regression), different output transformations may be applied instead of softmax.

Below is a simple implementation of an RNN cell in PyTorch (from scratch):

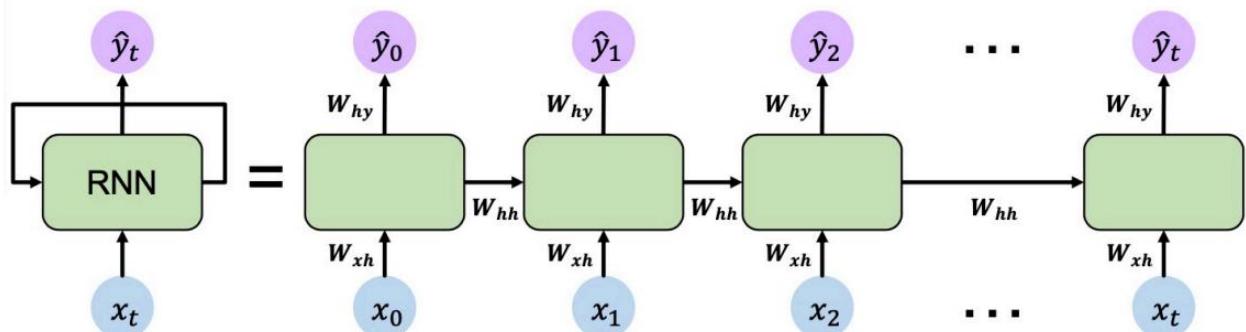
```
class MyRNNCell(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(MyRNNCell, self).__init__()
        # Initialize weight matrices
        self.W_xh = nn.Parameter(torch.randn(hidden_dim, input_dim))
        self.W_hh = nn.Parameter(torch.randn(hidden_dim, hidden_dim))
        self.W_hy = nn.Parameter(torch.randn(hidden_dim, output_dim))
        # Initialize hidden state to zeros
        self.h = torch.zeros(hidden_dim, 1)

    def forward(self, x):
        # Update hidden state
        self.h = torch.tanh(self.W_xh @ x + self.W_hh @ self.h)
        # Compute output
        y = self.W_hy @ self.h
        # Return the current hidden state and output
        return self.h, y
```

A key feature of RNNs is **parameter sharing across time steps**: the same set of parameters is reused at each time step (i.e. the same W_{xh} , W_{hh} , W_{hy} , b_h , b_y). This not only significantly reduces the number of model parameters, but it also enables to capture equivariances in the sequence, helping generalizing better across different positions in a sequence.

Backpropagation Through Time (BPTT)

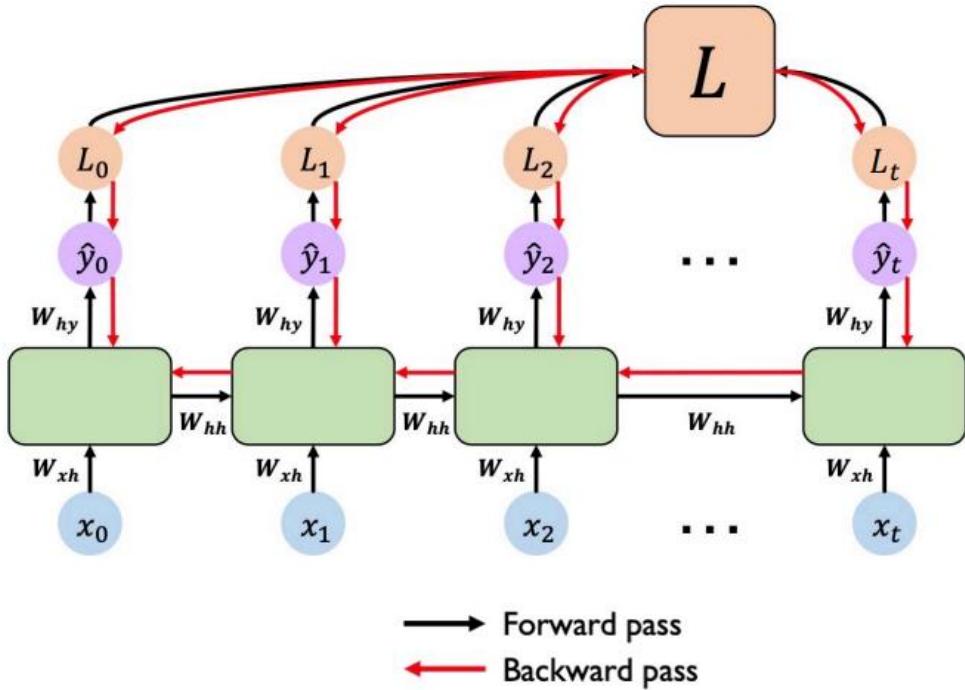
To understand how RNNs learn, it is helpful to visualize them “unrolled” across time. Unrolling an RNN means expanding it into a sequence of layers, where in this case each “layer” represents the network’s operation at a single time step. This unrolled structure forms a computational graph across time, resembling a deep feedforward network — with the crucial difference that **the same parameters are shared** across all time steps, and each hidden state depends not only on the current input but also on the previous hidden state (which helps in creating dependencies across time).



Consequently, training the RNN through time steps can be performed via SGD using gradients computed via backpropagation and evaluated via automatic differentiation just like with traditional neural networks. This specific adaptation of backpropagation **applied to the unrolled computational graph** of an RNN over multiple time steps is called **Backpropagation Through Time (BPTT)**. During BPTT, gradients are computed over the entire sequence to adjust the shared parameters, enabling the network to learn temporal dependencies from sequential data.

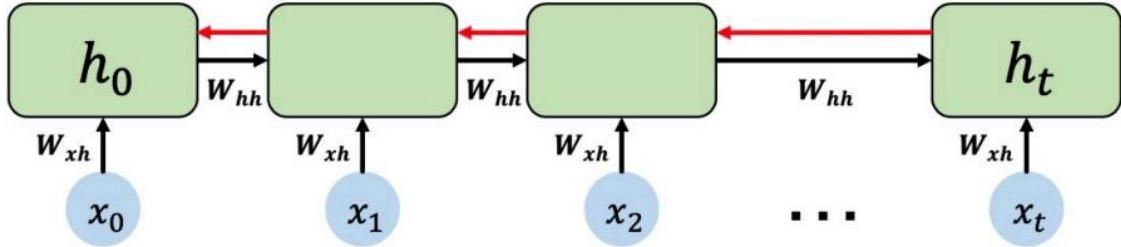
Conceptually, the entire training procedure using BPTT can be broken down into the following main steps:

1. **Forward Pass:** The RNN processes the input data sequentially, one time step at a time. At each time step t , the network computes a hidden state $h^{(t)}$ and an output $y^{(t)}$ based on the current input $x^{(t)}$, the previous hidden state $h^{(t-1)}$ and its weights and biases. Supposing a task of next-word prediction the output is then passed through a softmax to produce the predicted output $\hat{y}^{(t)}$ for that time step. This process is repeated for each time step in the sequence.
2. **Loss Computation:** After processing the entire sequence, the network's predictions $\hat{y}^{(t)}$ are compared against the true labels $\bar{y}^{(t)}$ at each time step in this case via a cross-entropy loss. The **total loss L** is then obtained by summing these individual losses.
3. **Backward Pass (Gradients Computation Through Time):** Next, we work backwards through the time sequence (starting from the final time step up to the first), computing the gradients of the parameters with respect to the total loss for each time step. Specifically, at each time step this involves computing partial derivatives of the loss with respect to the network parameters (i.e. W_{hh} , W_{xh} , W_{hy} and b_h , b_y). The gradients for each parameter computed at each step are **accumulated** and, once they have been computed for each time step, they are **summed** to obtain the overall gradient for each network parameter.
4. **Parameter Update:** Finally, the parameters are updated via SGD (or one of its variants) using the resulting overall gradients.



The Problem of Long-Term Dependencies

The main problem with RNNs is that they are difficult to train for very long sequences, due to the problems of “**exploding gradients**” and “**vanishing gradients**” that occur with very deep unrolled computational graphs.

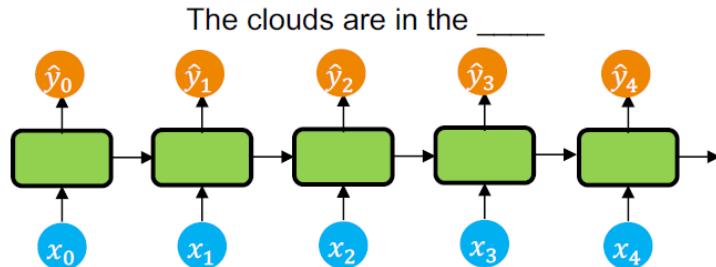


In the years **exploding gradients** were often mitigated using a technique called **gradient clipping**, which caps the gradient values at a predefined threshold to prevent them from becoming too large. However, the more critical and persistent issue is the **vanishing gradients**, which persisted for long time outlining the need for more sophisticated solutions.

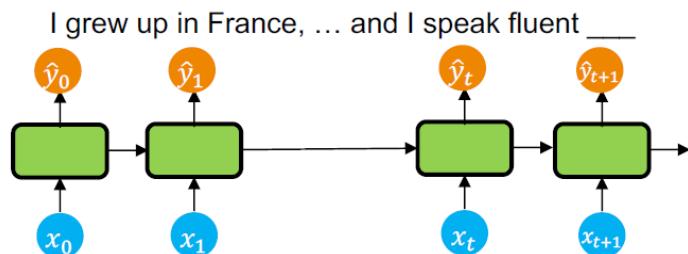
While feedforward networks are typically shallow enough to avoid this effect (they have only a few hidden layers), RNNs, even with adequate initialization, must propagate information across many time steps and therefore trained on long sequences (e.g. 100 time steps) can easily run into this problem.

Why have we vanishing gradients problem in RNNs?

If we have short-length sequences there are no problems and our RNN works well,



but as the length of the sequence increases the situation becomes different, since the longer the sequence the smaller the gradients become for first time steps in the sequence, complicating the learning of **long-term dependencies**.



In fact, due to vanishing gradients, while in the last time steps the information is still reliable, in the first time steps it becomes less reliable because the gradients become smaller and smaller.

Basically, in this process of updating the neural network, the dominant gradients are those of the short-term dependencies, because they are the ones that have larger gradients and therefore are updated more often, but the

long-term dependencies are lost because the gradients become too small → **a bias is created on the parameters, which only capture the short-term dependencies.**

As a result, for a RNN it becomes very difficult to detect that the current target output depends on an input from many time steps ago → RNNs **have difficulty handling long-term dependencies.**

Let's analyze this problem also from another perspective.

Imagine we input the sentence:

What time is it ?

Each word in the sequence is passed into the RNN one at a time. At first timestep, the word "**What**" is processed and a hidden state is generated. Then, at the next time step, the word "**time**" is input and the RNN combines this new word with the previous hidden state to produce an updated hidden state and output.



This continues with "**is**", then "**it**" and finally "**?**", each time building on the accumulated information from previous steps. However, here's where the problem starts to emerge. With each new word, the network updates its hidden state by combining the new input with the previous state — but **the influence of earlier words starts to fade**. When the input "**?**" is processed, the impact of "**What**" is already significantly diminished, as the hidden state's representation of that word has been repeatedly transformed and scaled down through the sequence. The farther back in time an input occurs, the less it contributes to the current state.



So, if understanding the sentence correctly depends on remembering "**What**" while processing "**?**", a standard RNN might fail to do so. This failure isn't due to a flaw in logic but due to how information fades through time in the model — **a direct consequence of the vanishing gradient problem**.

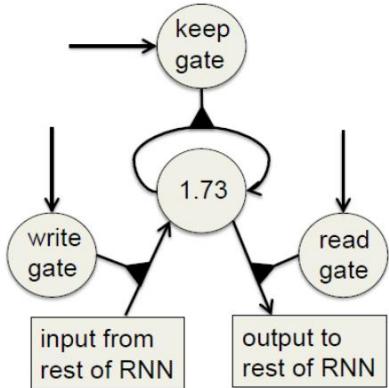
To handle this, the network would need a way to find a way to decide whether a piece of information is important or not, in order to possibly maintain its value and make it meaningful for the network. It should be able to remember which parts of the sequence are important and retain that information over long distances. Unfortunately, basic **RNN cells weren't designed to make such decisions** — they treat all incoming information uniformly, without the ability to distinguish between **what should be remembered and what can be forgotten**.

Idea: Using Gated Memory Cells

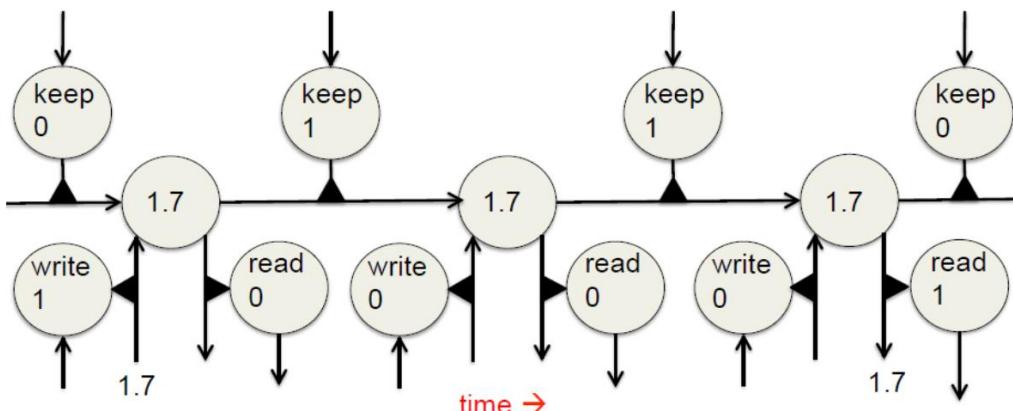
To address these issues researchers introduced more advanced RNN architectures that incorporate **gating mechanisms**. The central idea is to give the network a way to **control the flow of information over time**, allowing it to selectively **retain relevant data and discard what is no longer needed**.

The intuition behind a **gated memory cell** is simple yet powerful: information is managed through **gates** that determine when to write new information, when to keep existing information and when to read or output stored information.

- The **write gate** controls whether new input is allowed into the memory.
- The **keep gate** decides how much of the current content should be retained.
- The **read gate** determines whether the stored information is passed on to influence the output.



Below is an example of information flow through a gated memory cell across time steps.



At the **first time step**, the write gate is active (write = 1), so the input value (e.g., 1.7) enters the memory cell. However, we do not need to use this information immediately, so the read gate is off (read = 0), and the keep gate is on (keep = 1) to store the value. At the **second time step**, we want to continue storing the same information (keep = 1), but do not update it (write = 0) or read from it (read = 0). We repeat this for some other pairs of time steps keeping always that same information (1.7). At the final time step, we activate the read gate (read = 1) to retrieve the information, and we stop keeping it (keep = 0) as it's no longer needed.

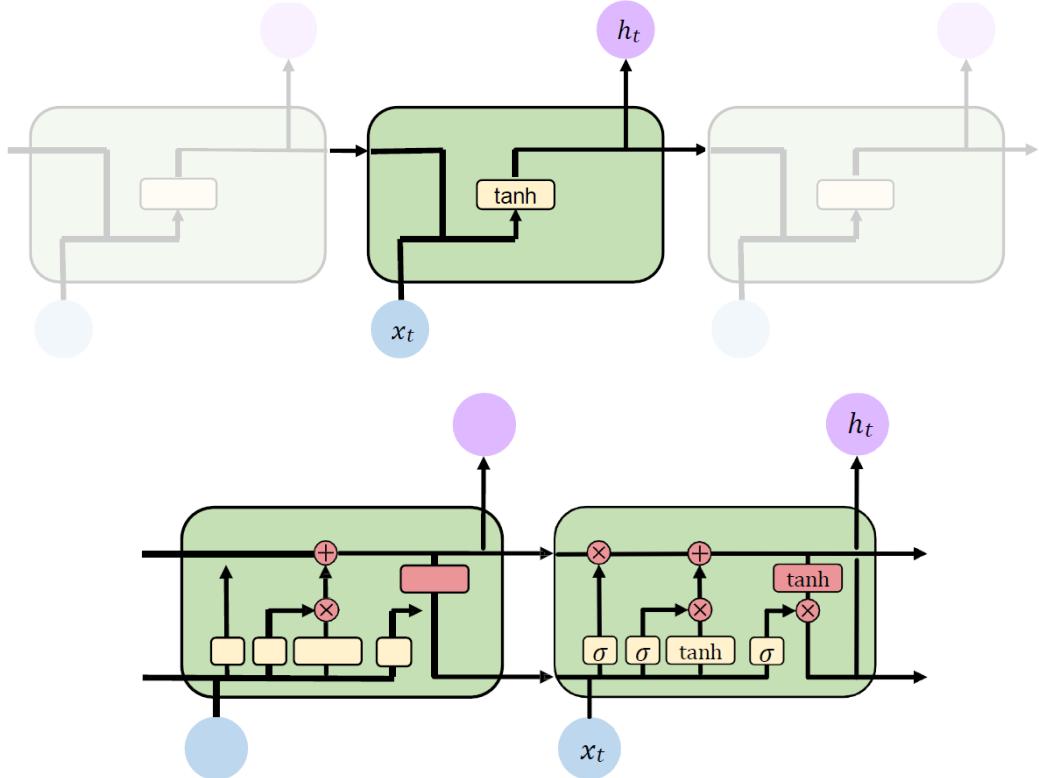
As we can see in this way we were able to keep in memory the input value 1.7 for many time steps. Therefore, this kind of controlled memory management allows the network to **preserve and reuse specific pieces of information over long sequences**, which is crucial for learning dependencies across distant time steps.

Long Short-Term Memory (LSTM) networks — and later, *Gated Recurrent Units (GRUs)* — are the most well-known examples of RNN architectures that use these gating mechanisms

LSTM

Long Short-Term Memory (LSTM) network, developed by Hochreiter and Schmidhuber (1997) [42], has a structure similar to that of a standard RNN, but with each recurrent node replaced by a **gated memory cell**. In particular, in an LSTM each gated memory cell is equipped with an internal state and a set of multiplicative gates that determine

whether a given input should influence the internal state (**input gate**), whether the internal state should discard some information (**forget gate**) and whether the internal state should influence the cell's output (**output gate**).



The input data to the LSTM gates are the input to the current time step and the hidden state of the previous time step. Three fully connected layers, each with a sigmoid activation function, determine the values of the input gate, forget gate and output gate. The sigmoid activation function produces values in the interval [0,1], adjusting the amount of information each gate should allow to pass. Mathematically, the gate values at time step t are calculated as follows:

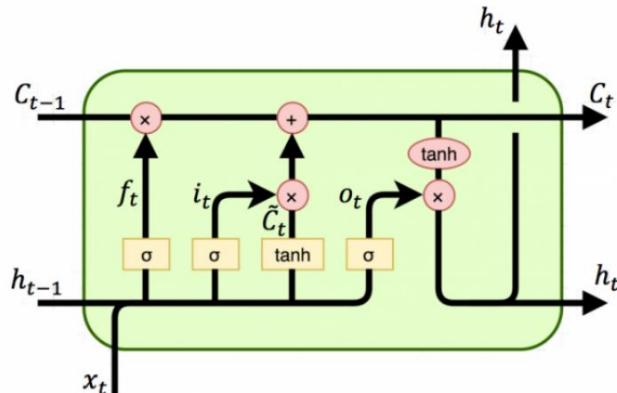
$$i^{(t)} = \sigma(x^{(t)}W_{xi} + h^{(t-1)}W_{hi} + b_i)$$

$$f^{(t)} = \sigma(x^{(t)}W_{xf} + h^{(t-1)}W_{hf} + b_f)$$

$$o^{(t)} = \sigma(x^{(t)}W_{xo} + h^{(t-1)}W_{ho} + b_o)$$

Additionally, there is an input node, typically computed by a \tanh activation function, which generates a vector of new candidate values $\tilde{c}^{(t)}$ to be added to the cell state:

$$\tilde{c}^{(t)} = \tanh(x^{(t)}W_{xc} + h^{(t-1)}W_{hc} + b_c)$$



In LSTMs, the **input gate** $i^{(t)}$ governs the importance of the new data $\tilde{c}^{(t)}$, determining how much of this data will be incorporated into the internal state of the cell. An input gate with a value of 1 indicates that all information from $\tilde{c}^{(t)}$ should be retained because it is relevant, while a value of 0 indicates that this information should be completely discarded because it is not relevant.

On the other hand, the **forget gate** $f^{(t)}$ controls how much of the previous cell state $c^{(t-1)}$ will be retained. So, a forget gate with a value of 1 indicates that the value of the previous cell state should be completely retained, while a value of 0 indicates that this value should be completely eliminated from the internal state of the cell, i.e., forgotten.

The internal state of cell $c^{(t)}$ is updated as follows:

$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \tilde{c}^{(t)}$$

The resulting new internal state of the cell incorporates both previously stored relevant information, retained through the value of the forget gate, and new information selected through the value of the input gate. The forget and input gates provide the model with the flexibility to learn when to keep this value unchanged and when to change it in response to subsequent inputs. For example, if the forget gate is always 1 and the input gate is always 0, the internal state of the memory cell will remain unchanged at each subsequent time step. This design helps mitigate the problem of “vanishing gradients”, making models much easier to train, especially when dealing with long sequences of data.

Finally, the hidden state $h^{(t)}$ of the memory cell is calculated as follows:

$$h^{(t)} = o^{(t)} \odot \tanh(c^{(t)})$$

When the **output gate** $o^{(t)}$ value is close to 1, the cell's internal memory is allowed to freely influence subsequent phases of the network. Conversely, for output gate values close to 0, the current memory is prevented from influencing other parts of the network at the current time step. It is important to note that a memory cell can accumulate information for many time steps without influencing the rest of the network as long as the output gate assumes values close to 0 and then influence the network at a subsequent time step as soon as the output gate transitions to values close to 1.

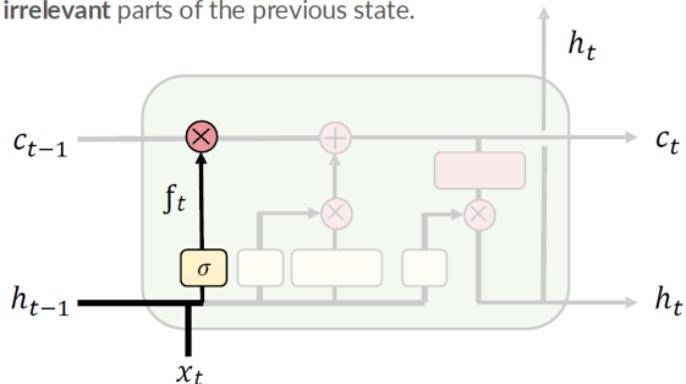
The cell's new internal state $c^{(t)}$ and the new hidden state $h^{(t)}$ will then be used as inputs for the next time step.

Recap:

- 1) Forget
 - 2) Store
 - 3) Update
 - 4) Output
- LSTM forgets irrelevant parts of the previous state.

- Use previous cell output and input
- Sigmoid: value 0 and 1 “completely forget” vs. “completely keep”

ex: Forget the gender pronoun of previous subject in sentence.

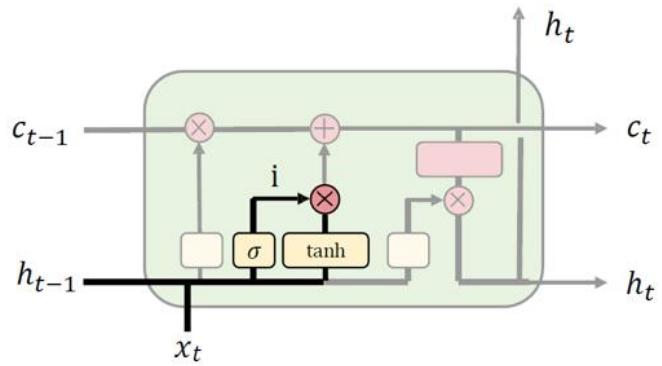


1) Forget 2) Store 3) Update 4) Output

LSTMs store relevant new information into the cell state

- Sigmoid layer: decide what values to update
- Tanh layer: generate new vector of "candidate values" that could be added to the state

ex: Add gender of new subject to replace that of old subject.

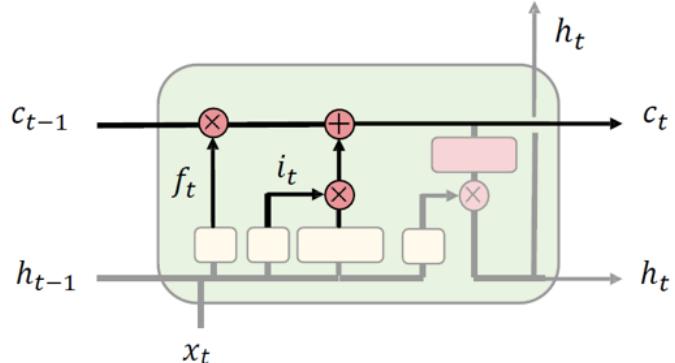


1) Forget 2) Store 3) Update 4) Output

LSTMs selectively update cell state values

- Apply forget operation to previous internal cell state: $c_{t-1} * f_t$
- Add new candidate values, scaled by how much we decided to update: $i_t * \hat{c}_t$

ex: Actually drop old information and add new information about subject's gender.

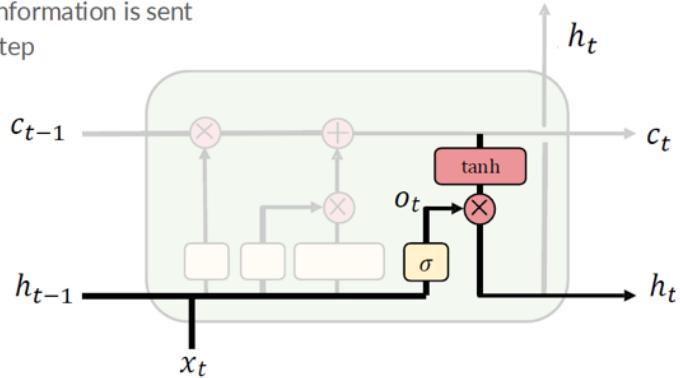


1) Forget 2) Store 3) Update 4) Output

The **output gate** controls what information is sent to the next time step

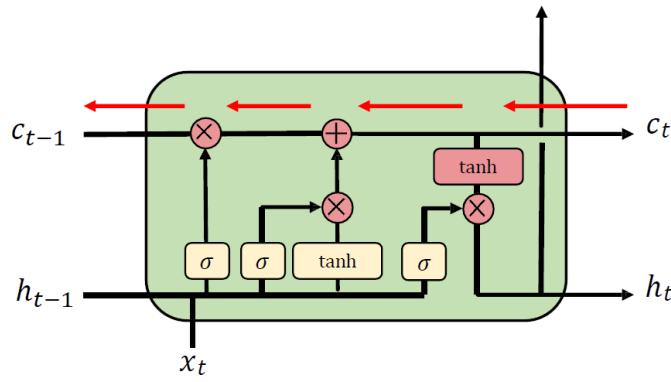
- Sigmoid layer: decide what parts of state to output
- Tanh layer: squash values between -1 and 1
- $o_t * \text{tanh}(c_t)$: output filtered version of cell state

ex: Having seen a subject, may output information relating to a verb.

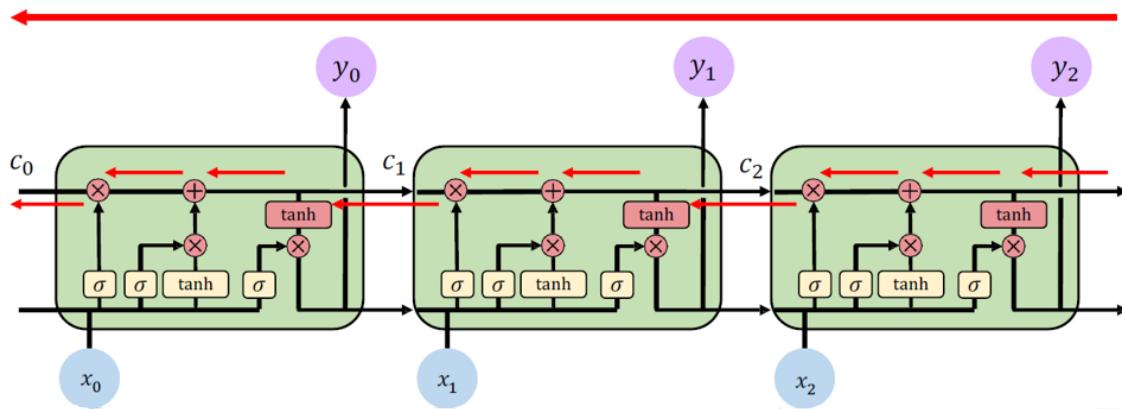


LSTM Gradient Flow

One of the key advantages of LSTMs lies in how they handle gradient propagation during backpropagation. Specifically, the update from the current cell state $c^{(t)}$ to the previous cell state $c^{(t-1)}$ involves only **element-wise operations**, without any matrix multiplications or non-linear activation functions. Since the cell state update **bypasses non-linearities**, the gradients can flow backward through time **without diminishing**, allowing the network to retain useful signals across many time steps.



As a result, LSTMs provide a **stable and uninterrupted gradient flow** during training, making them particularly well-suited for capturing **long-term dependencies** in sequential data.



LSTM: Key Concepts

1. Use gates to control the flow of information

- **Forget gate:** Discards information that is no longer relevant.
- **Input gate:** Selectively adds new information to the cell state.
- **Output gate:** Filters and sends out relevant information from the cell state.

2. Efficient gradient propagation:

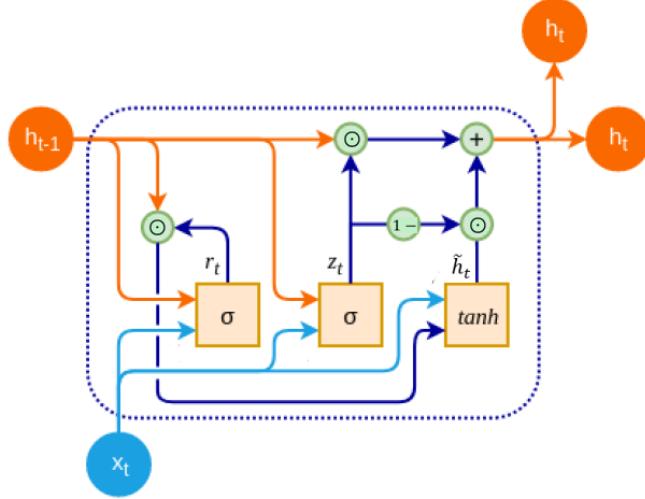
- Backpropagation from $c^{(t)}$ to $c^{(t-1)}$ relies only on **element-wise operations (no non-linearities)**. This enables **uninterrupted gradient flow**, preventing vanishing gradients and supporting the learning of long-range dependencies.

GRU

The **Gated Recurrent Unit (GRU)**, proposed by Cho et al. (2014) [43], is a variant of LSTMs that simplifies the information flow and requires fewer tensor operations. Unlike LSTMs, GRU uses a single hidden state instead of two, merging the internal state of the cell $c^{(t)}$ with the hidden state $h^{(t)}$. It also combines the input gate and the forget gate into a single gate called the **update gate** and replaces the output gate with a **reset gate**.

Gates Roles:

- The **reset gate** controls how much of the previous hidden state should be forgotten at the current time step.
- The **update gate** controls how much of the new candidate activation should be incorporated into the hidden state.



How Does It Work?

At each time step t , the **reset gate** $r^{(t)}$ and the **update gate** $z^{(t)}$ are first computed as follows:

$$r^{(t)} = \sigma(x^{(t)}W_{xr} + h^{(t-1)}W_{hr} + b_r)$$

$$z^{(t)} = \sigma(x^{(t)}W_{xz} + h^{(t-1)}W_{hz} + b_z)$$

Next, the reset gate is integrated into the computation of the hidden state update, similarly to the mechanism used in a standard RNN, resulting in the definition of the candidate hidden state $\tilde{h}^{(t)}$:

$$\tilde{h}^{(t)} = \tanh(x^{(t)}W_{xh} + (r^{(t)} \odot h^{(t-1)})W_{hh} + b_h)$$

In this way, the influence of previous hidden states can be modulated by element-wise multiplication of $r^{(t)}$ with $h^{(t-1)}$. The result is a *candidate* hidden state, since the action of the update gate still needs to be considered.

At the end the effect of the update gate $z^{(t)}$ is incorporated. It determines to what extent the new hidden state $h^{(t)}$ should be influenced by the previous hidden state $h^{(t-1)}$ (weighted by $z^{(t)}$) versus the candidate hidden state $\tilde{h}^{(t)}$ (weighted by $1 - z^{(t)}$). This leads to the final GRU update equation:

$$h^{(t)} = z^{(t)} \odot h^{(t-1)} + (1 - z^{(t)}) \odot \tilde{h}^{(t)}$$

The result of this operation is the new hidden state $h^{(t)}$. We can notice that:

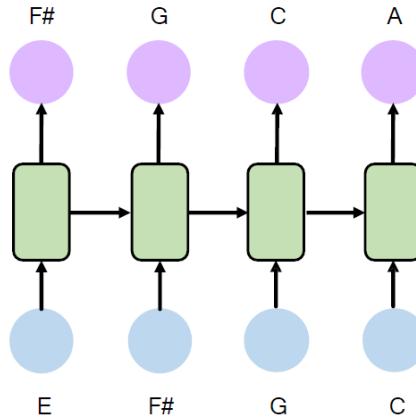
- When $z^{(t)}$ is close to 1, most of the previous hidden state $h^{(t-1)}$ is retained.
- When $z^{(t)}$ is close to 0, the new hidden state $h^{(t)}$ closely approximates the candidate hidden state $\tilde{h}^{(t)}$.

RNN Applications

Given their strong ability in modeling sequential data, RNNs are used in a wide variety of domains. Below are some notable applications:

- **Music generation**

In music generation, an RNN is trained on a dataset of musical sheet and learns to *predict the next note* in a sequence based on the preceding musical context. At each time step the RNN cell uses its hidden state information to predict the new note.

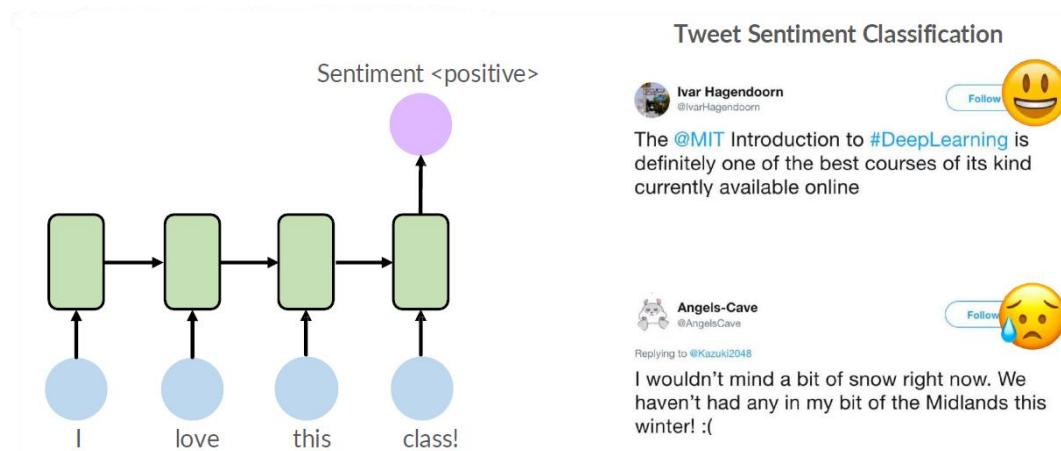


Given an input sequence such as a musical score, the RNN can be able to generate a new melody or arrangement that has some coherence and cohesion with the overall musical style and structure.

- **Sentiment classification**

RNNs are also used for **sentiment classification**, i.e. for classifying the polarity of a given sentence (does it have a positive or a negative sentiment?)

For example, when analyzing user reviews, the input is a sequence of words from the review text. The RNN processes this sequence and outputs a probability of having positive sentiment.



This is especially useful for analyzing opinions in social media, product reviews or customer feedback.

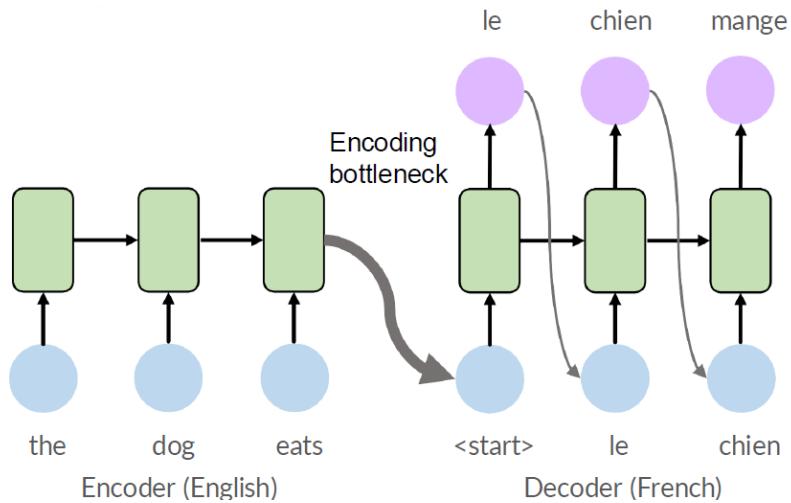
- **Machine translation**

Machine translation is another major application of RNNs. In this task, an RNN-based model, typically an encoder-decoder architecture, is trained to process the sequences of words in a source language and translate them into another language. This type of architecture was introduced for the first time by Cho et al. in *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation* (2014) [44]. The encoder processes a variable-length input sequence in one language and encodes it into

a fixed-length vector representation (or hidden state), which the decoder uses to generate the translated sentence, also of variable length.

Note: Specifically, the decoder generates the translated sentence in an **autoregressive** manner — that is, it produces one word at a time, and at each step, it conditions its prediction on all previously generated words. Each predicted word is fed back into the decoder as input for generating the next word. We will explore this in more detail when we discuss the Transformer architecture.

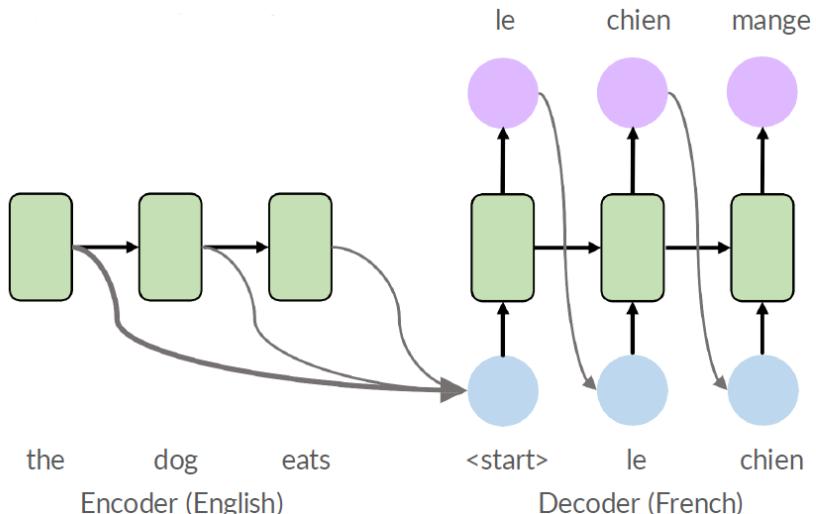
In particular, the encoder in this architecture is an RNN that sequentially reads each word of the input sequence. As each word is read, the hidden state of the RNN updates based on the input word and the previous hidden state. Once the end of the input sequence is reached, the last hidden state of the encoder serves as a summary of the whole input, which is passed to the decoder. The decoder is another RNN that generates the output sequence by predicting each subsequent word, based not only on the previous predicted word in the output sequence, but also conditioned on the encoder's final hidden state. In this way, the decoder utilizes both the previous output and the context from the encoder to generate the translation.



However, this approach faces a limitation, i.e. the **encoding bottleneck**: since the information from the encoder is “compressed” just into the last hidden state before being passed to the decoder, there is a risk that important details from the input sequence may be lost, especially for longer sentences. This constraint makes it difficult for the model to capture and retain all relevant information within a single state.

To overcome this, **attention mechanisms** were introduced. Attention mechanism was for the first time introduced exactly in this context by Bahdanau et al. in *Neural Machine Translation by Jointly Learning to Align and Translate* (2014) [45]. We will explore attention mechanisms in more detail in the Transformer chapter as it is the fundamental component of its architecture, but the core idea behind **attention** is to allow the model to assign an "attention weight" to each word in the input sequence, prioritizing those most relevant to generating the correct output.

With attention, the encoder is no longer responsible from the burden of having to encode all information in the source sentence into a fixed length vector (the last encoder hidden state). Instead, the decoder can now dynamically access all hidden states of the encoder and select relevant information throughout the input sequence as needed.



Effectively, attention provides the network with **learnable memory access**, allowing it to retrieve the most critical information needed to produce better outputs, enhancing its ability to generate accurate translations

Bi-LSTM

Consider the following two sentences:

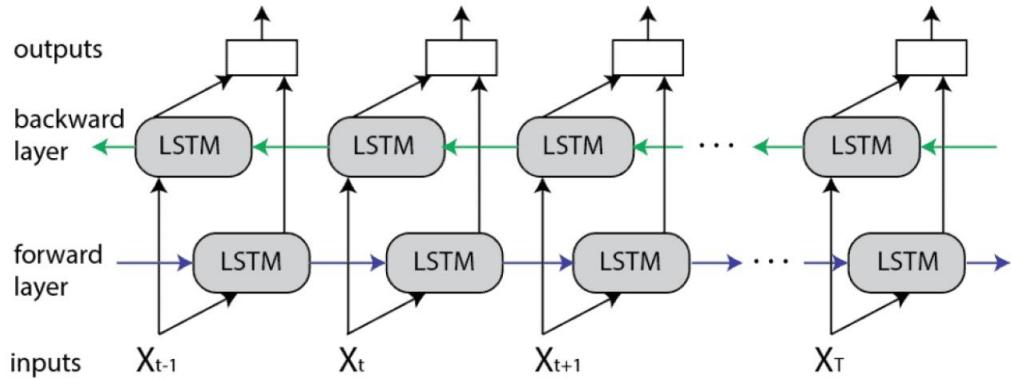


There's an ambiguity around the word *Teddy* in both sentences. In the first sentence *Teddy* refers to a stuffed animal, while in the second sentence *Teddy* refers to Theodore, a person's name.

Just by looking at the previous context it is unclear, when processing the word *Teddy*, whether it's part of a person's name or not. To do that we need the information about the following tokens. In fact, if we have the possibility to look at the words that follow it, we can resolve this ambiguity.

For instance, in the first sentence, knowing that the next word is *bears* helps clarify that *Teddy* is not a person's name. In the second sentence, knowing that the following word is *Roosevelt* helps us identify *Teddy* as part of a name. This shows that, in some tasks like classification (e.g. sentiment classification), it's not necessary to predict the next token, but rather to understand the current token in the context of both the preceding and following words. This is the intuition behind the Bi-LSTM model.

A **Bidirectional LSTM** or **Bi-LSTM**, introduced by Schuster et al. (1997) [46], is an extension of the standard LSTM model, which processes data in both forward and backward directions. This enables the model to capture context from both the past and future, making it particularly useful for tasks where understanding the full context is crucial.

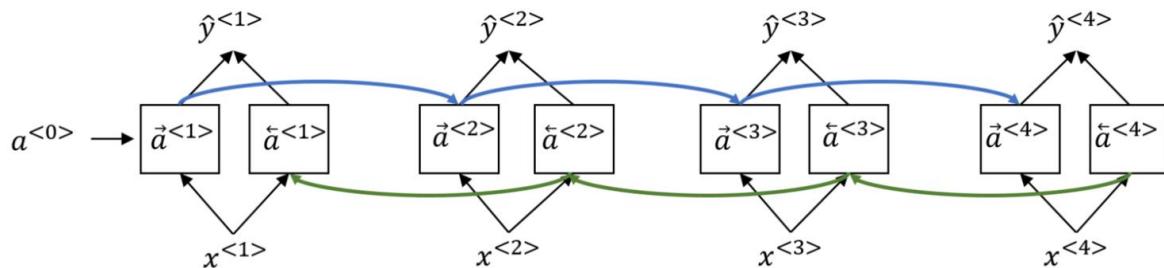


How Bi-LSTM Works?

Bi-LSTM consists of two LSTM layers, a **forward LSTM layer** and a **backward LSTM layer**.

During the forward pass, the forward LSTM computes the activation values by processing the input sequence and updating its hidden state as it moves from the start to the end. Similarly, the backward LSTM computes activation values by processing the sequence in reverse, from the end to the start.

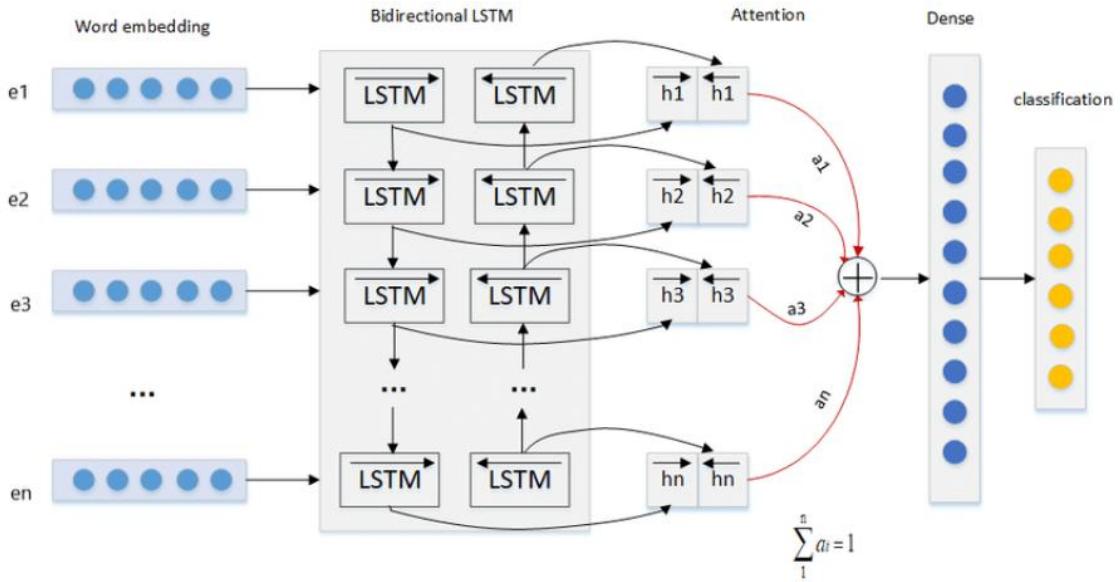
After a single pass of forward propagation, the prediction for each time step t are obtained by combining the activations of both LSTM layers for the corresponding time step. This is usually done through **concatenation** or another transformation like **element-wise summation**.



The major advantage of Bi-LSTM is that it allows us to consider words from both directions when making a prediction, effectively increasing the amount of context available to the model (e.g. knowing what words immediately *follow* and *precede* a word in a sentence). This makes it a good fit for many language-related applications like machine translation. On the downside, because tokens from both directions are considered, the whole sequence needs to be processed before a prediction can be made. This makes it unsuitable for tasks like real-time speech recognition.

Bi-LSTM with Attention (Att-BLSTM)

After discussing the Bi-LSTM model, it's natural to extend this architecture to incorporate an **Attention Mechanism** for enhanced performance. The combination of Bi-LSTM with Attention, referred as **Att-BLSTM** (2016) [47], is particularly useful for tasks that require capturing long-range dependencies and assigning varying importance to different parts of the input sequence. Attention mechanisms help the model focus on the most relevant words in a sentence, improving its ability to make more accurate predictions.



How Bi-LSTM with Attention Works?

The process begins by converting each word in the sentence into a word embedding. This is typically done by multiplying a word's one-hot vector representation with an embedding matrix. These word embeddings are then fed into a **Bi-LSTM** to capture both past and future context for each word in the sequence.

As we've seen with the Bi-LSTM, the forward LSTM processes the sequence from left to right, while the backward LSTM processes it from right to left. The output vectors from both directions are then combined, in this case by taking an element-wise sum.

This combined representation now contains context from both directions, which improves the model's understanding of each word in the sentence. However, simply concatenating or summing the output of the Bi-LSTM might not be enough. In some cases, certain words are more important than others when making a prediction. This is where the attention mechanism can come to help.

The **Attention Mechanism** is applied after the Bi-LSTM layer to allow the model to weigh the importance of each word in the sequence dynamically. These weights are used to create a weighted sum of the Bi-LSTM outputs, which results in a **single, contextually rich representation of the entire sentence**. Once this sentence-level representation is obtained, it is typically passed through one or more FC layers to perform the final task, such as classification.

RNNs Limitations

RNNs and their variants were among the first effective architectures designed to handle **sequential data**. They introduced the ability to process inputs of variable lengths and maintain a form of memory (through recurrent connections), making them well-suited for tasks like language modeling.

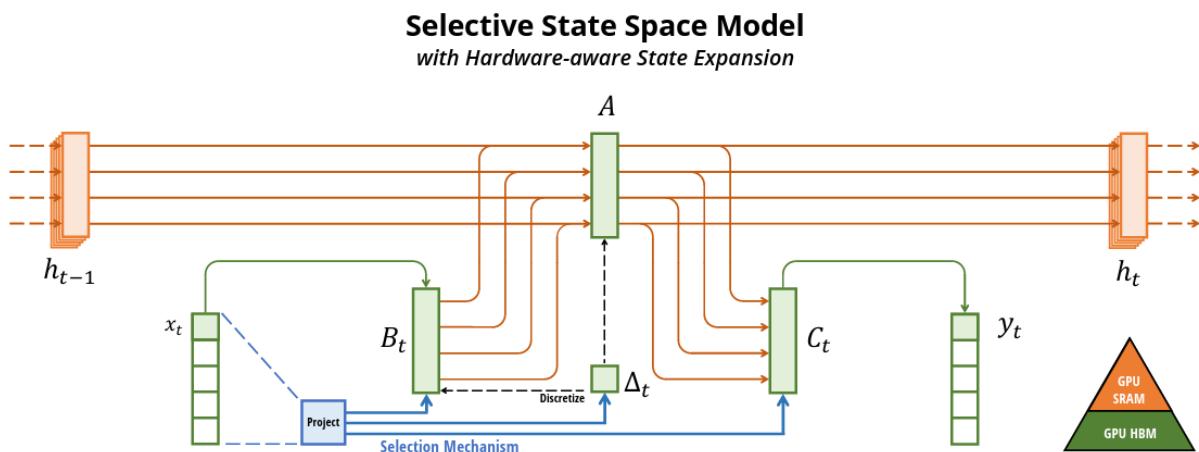
However, RNNs suffer from several important limitations:

1. **Difficulty Modeling Long-Range Dependencies:** Even advanced variants like LSTMs still have a limited ability to model long-range dependencies and can degrade in performance as the temporal gap increases.
2. **Sequential Signal Path Length:** The computation in RNNs is inherently sequential — each time step depends on the previous one. As a result, the signal path through the network grows linearly with the

sequence length, therefore learning long-term dependencies becomes increasingly difficult as the model must propagate gradients through many time steps.

3. **Training Difficulties:** RNNs are notoriously difficult to train. Traditional gradient descent methods often fail due to unstable gradient dynamics. To address this, researchers explored **Hessian-Free Optimization**, which attempted to leverage curvature information to make more informed updates and therefore improve convergence. Although effective in some cases, these techniques are computationally expensive and complex to implement, limiting their practical adoption.
4. **Lack of Parallelism:** Due to their inherently sequential nature, RNNs do not support parallel computation across time steps within a single training example. This prevents them from making efficient use of modern highly parallel hardware based on GPUs.

In next years, many of these limitations have been addressed by replacing RNNs with **Transformers**, which excel at modeling long-range dependencies through attention mechanisms and allow for high parallelization. However, in recent years a new wave of architectures has brought RNN-like models back into the spotlight. One notable example is **Mamba**, introduced in 2023 by *Albert Gu and Tri Dao* in the paper “*Mamba: Linear-Time Sequence Modeling with Selective State Spaces*” [48].



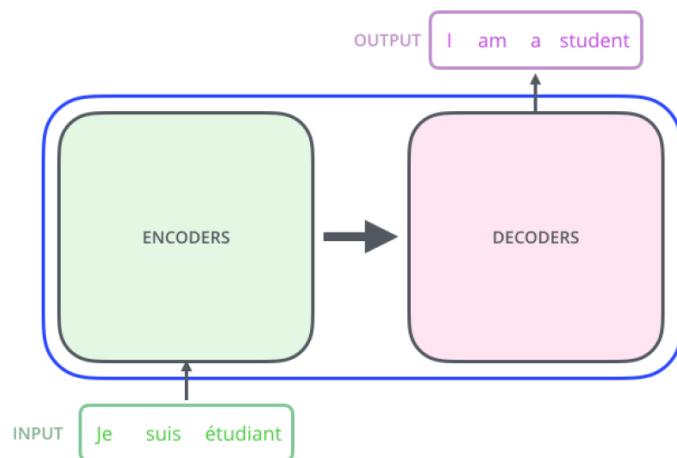
As you will see in the next chapter, while Transformers have become the dominant architecture in recent years thanks to their flexibility, they come with a significant challenge: **the quadratic complexity of the attention mechanism**, which makes handling very long sequences computationally expensive. Mamba addresses this limitation by leveraging **State-Space Models (SSMs)** combined with **selective mechanisms** to achieve **near linear-time complexity** for sequence modeling, while also permitting parallelization. This makes it highly efficient for extremely long sequences, positioning it as a strong alternative to Transformers in scenarios where quadratic attention becomes a bottleneck.

For a more intuitive and visual explanation of Mamba and its connection to state-space models, I recommend reading this excellent blog post: [A Visual Guide to Mamba and State Space Models](#). If you plan to explore this topic further, it's best to do so after understanding Transformers, as Mamba's design choices directly address many of the challenges inherent in attention-based models.

Transformers

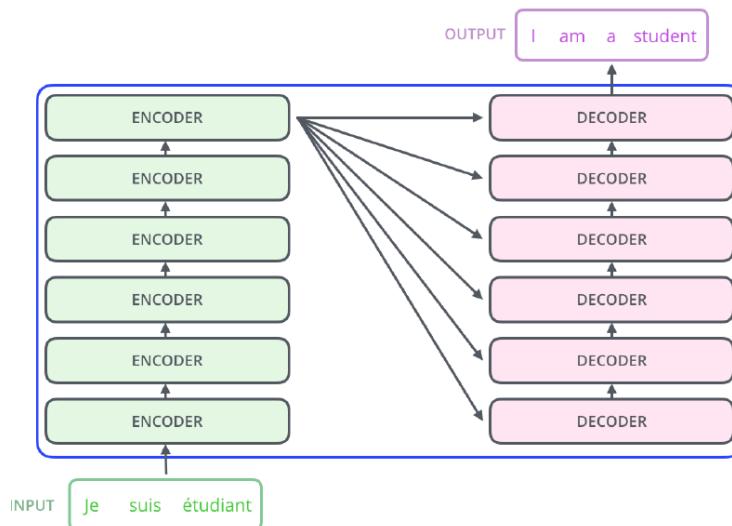
Transformers are among the most influential advancements in deep learning. Initially introduced in the field of NLP, they have since demonstrated outstanding performance across a wide range of domains beyond language, including computer vision, audio processing and more.

The original Transformer architecture was proposed by Vaswani et al. (Google Brain) in their groundbreaking 2017 paper, “**Attention is All You Need**” [49]. It was specifically designed for the task of **machine translation** — taking a sentence in one language as input and producing its translation in another language as output. Hence, it adopts an encoder-decoder structure where the encoder takes as input the sentence in the original language and generates an attention-based representation. On the other hand, the decoder attends over the encoded information and generates the translated sentence in an **autoregressive manner** (soon will understand better these aspects).



Both the encoder and the decoder consist of N **identical blocks** stacked sequentially.

Note: These blocks are **not weight-shared**, meaning each has its own learnable parameters, although their structure is the same within each stack.



Since the Transformer architecture integrates several important components that operate together, we will now break down the model step by step — starting with the core idea of attention and gradually building up to a detailed understanding of each of the major components. Rather than strictly following the slides used in the course —

which are based on the well-known blog post [The Illustrated Transformer](#) by Jay Alammar — I've chosen a different approach. While that resource offers an excellent visual and beginner-friendly introduction, it tends to gloss over deeper explanations, particularly regarding the motivations behind key architectural choices. My aim here is to provide a more comprehensive and rigorous perspective, focusing not only on what the model does but also on **why** it was designed this way. For similar reasons, I've also opted not to follow the slides when explaining how GPT and BERT work. In the case of GPT, in particular, I believe many of the most important aspects are **skipped entirely**. That said, I think that the slides based [The Illustrated GPT-2](#) for GPT and [The Illustrated BERT](#) for BERT can still be useful to get a visual grasp of some of the core concepts.

Attention

At the heart of the Transformer architecture lies the concept of **attention**. Introduced by Bahdanau as an enhancement to RNNs for machine translation, attention was initially used to help models selectively focus on relevant parts of the input when generating each word in the output. Later, Vaswani et al. demonstrated that it was possible to discard recurrence structure and **rely solely on attention mechanisms**, achieving remarkable improvements in both speed and performance. Today, attention-based models like Transformers have effectively replaced RNNs in almost all applications.

Intuition behind Attention

To build an intuitive understanding of attention, let's consider an example from natural language processing, where the idea was first applied. Take the following two sentences:

I swam across the river to get to the other bank.

I walked across the road to get cash from the bank.

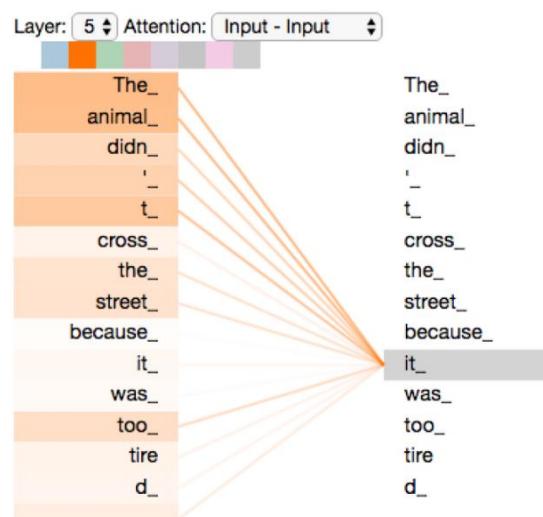
Here the word ‘bank’ has different meanings in the two sentences. However, this can be detected only by looking at the context provided by other words in the sequence. We also see that some words are more important than others in determining the interpretation of ‘bank’. In the first sentence, the words ‘swam’ and ‘river’ most strongly indicate that ‘bank’ refers to the side of a river, whereas in the second sentence, the word ‘cash’ is a strong indicator that ‘bank’ refers to a financial institution. We see that to determine the appropriate interpretation of ‘bank’, a neural network processing such a sentence should attend to, in other words rely more heavily on, specific words from the rest of the sequence.

Let's also consider another sentence:

The animal didn't cross the street because it was too tired

What does ‘it’ in this sentence refer to? Is it referring to the ‘street’ or to the ‘animal’? Thanks to attention, the model will be able to look at other positions in the input sequence and understand which are the more relevant that lead to a better encoding for this word. So, in this case when encoding “it,” the model attends more strongly to the word “animal”, allowing it to infer that “it” most likely refers to the animal.

In Figure on side, we can see an **attention map** that shows that when the model processes the word ‘it’, the highest attention weights are for ‘The’ and ‘animal’, so the model is focusing more on these than the other words in the input sentence.



These examples capture precisely the role of attention: it enables the model to selectively focus more on the relevant parts of the input. Having built the intuition, we're now ready to explore how this concept can be implemented in practice.

Attention Weights

First let's recall that we don't fill words directly into a neural network, but some numerical representation of those words. In particular, we saw that we can use **embeddings** to map words, or more precisely *tokens*, into continuous vectors in an embedding space.



These embedding vectors are typically generated in a **pre-processing step**, and they serve as the initial input to the Transformer model. Therefore, the input to a Transformer is a sequence of vectors $\{x_i\}$ of dimensionality D (*latent dim*), where $i = 1, \dots, N$, being the embeddings of the tokens of the input sentence.

The Transformer's goal is to transform each input vector x_i into a new vector z_i , always of dimension D but now in a **richer embedding space** that captures more meaningful and context-aware representations. In particular in the transformation step, the value of z_i obtained should depend not only on x_i , but also on the dependence of it with all the other vectors in the set. With attention this dependence should be stronger for those inputs x_j that are particularly important for determining the modified representation of z_i . A simple way to achieve this is to define each output vector z_i to be a **weighted sum** of all input vectors:

$$z_i = \sum_{j=1}^N a_{ij} x_j$$

where the coefficients a_{ij} are called **attention weights** and determine how much attention z_i should pay to x_j . These coefficients should be close to zero for input vectors that have little influence on the output vector z_i and largest for inputs that have most influence. We therefore want the attention weights to be constrained to be non-negative to avoid situations in which one coefficient can become large and positive while another coefficient compensates by becoming large and negative. We also want to ensure that if an output pays more attention to a particular input, this will be at the expense of paying less attention to the other inputs, and so the attention weights should also be constrained to sum to unity. Thus, the attention weights must satisfy the following two constraints:

- **Non-negativity:**

$$a_{ij} \geq 0$$

This prevents the model from canceling out one influence with a negatively weighted one, ensuring all contributions are additive.

- **Normalization (Sum-to-One):**

$$\sum_{j=1}^N a_{ij} = 1$$

This ensures the weights form a proper probability distribution, so that if one token receives more attention, others must receive less.

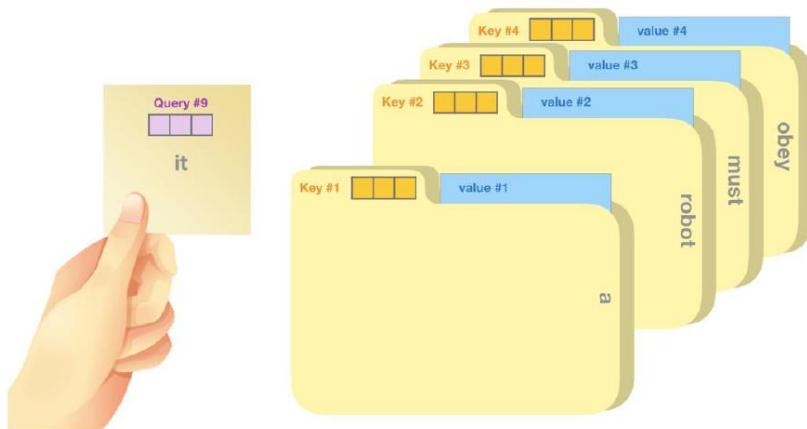
Now let's see how these weights a_{ij} are actually computed and how they can be learned in a way that captures relevance between tokens.

Attention Mechanism

To build the formulation for the attention mechanism let's consider an analogy with *information retrieval*.

Consider the problem of choosing which movie to watch in an online movie streaming service such as Netflix. One approach would be to associate each movie with a list of features describing things such as the genre (comedy, action, etc.), the names of the leading actors, the length of the movie and so on. The user could then search through a catalogue to find a movie that matches their preferences. We could automate this by encoding the features of each movie in a vector called the **key**. The corresponding movie file itself is called a **value**. Similarly, the user could then provide their own personal vector of values for the desired features, which we call the **query**. The movie service could then compare the *query* vector with all the *key* vectors to find the best match and send the corresponding movie to the user in the form of the *value* file.

We can think of the user ‘attending’ to the particular movie whose key most closely matches their query and return of it its value. This would be considered a form of *hard attention* in which a single value vector is returned. For the Transformer, we generalize this to **soft attention** in which we use continuous variables to measure the degree of match between *queries* and *keys* and we then use these variables to weight the influence of the *value* vectors on the outputs.



Now, to see how much a query vector q_i should attend each key vector k_j , we need to work out how *similar* each couple of these vectors are. One simple **measure of similarity** is to take their **dot product** $q_i \cdot k_j$. This score tells us how aligned the two vectors are — the higher the dot product, the more relevant k_j is to q_i .

Given this and the need to satisfy the constraints we discussed earlier — namely, that attention weights must be **non-negative** and **sum to one** — we can think to compute the **attention weights** as:

$$a_{ij} = \text{softmax}(q_i \cdot k_j) = \frac{\exp(q_i \cdot k_j)}{\sum_{l=1}^N \exp(q_i \cdot k_l)}$$

Here we applied the **softmax function** to the similarity scores computed via dot products because it ensures that:

- Each weight a_{ij} is **non-negative**, due to the exponential.
- The weights for each query **sum to 1**, due to the normalization across all keys in the denominator.

Note: In this context, softmax does **not have a probabilistic interpretation** — it is simply a convenient way to normalize the attention weights appropriately.

Bringing everything together, we can now express the basic formulation of attention mechanism as:

$$\mathbf{z}_i = \sum_{j=1}^N \mathbf{a}_{ij} \mathbf{v}_j = \sum_{j=1}^N \text{softmax}(\mathbf{q}_i \cdot \mathbf{k}_j) \mathbf{v}_j$$

Here we built-up a general formulation because based on what the query, key and values are we can have different types of attention (self-attention, cross-attention, etc. as we will see soon).

Self-Attention

Now that we understand the basic mechanism of attention, a natural question arises: *What exactly are the query, key and value vectors in our case?*

Since we're dealing with a sequence of tokens, we can view the query vector as the input token x_i that we are actually processing. To compute how much this token should attend to others in the sequence, we compare it to every other token vector x_j . We can view each of these tokens vectors x_j as a value vector and we can also use the vector x_j directly as the key vector for that token. Making an analogy with the example seen above it would be analogous to using the movie itself to summarize the characteristics of the movie. Using the attention definition above, we can therefore obtain the transformed output vector z_i for the considered input vector x_i as:

$$z_i = \sum_{j=1}^N \text{softmax}(x_i \cdot x_j) x_j$$

Since we use the same sequence of tokens to determine the *queries, keys and values* this process is known as **Self-Attention**. Moreover, because the measure of similarity between query and key vectors is given by a dot product, this is also sometimes referred to as *Dot-product Self-Attention*.

Now, if we combine the input vectors into a matrix X of dimensions $N \times D$ in which the i -th row comprises the token embedding x_i , we can rewrite the formulation above in matrix notation as:

$$Z = \text{Softmax}(XX^T)X$$

where the resulting output matrix Z is also of size $N \times D$ (same as X).

However, as formulated above, this transformation is **fixed** and **non-learnable** — it contains no trainable parameters. Moreover, each feature within a token embedding contributes equally to the similarity calculation, which is limiting. Ideally, we would like the network to have the flexibility to focus more on some features than others when determining token similarity score.

We can address both issues if we apply **learnable linear transformations** to the original input vectors analogous to a **Linear Layer** (FC) in a standard neural network:

$$\tilde{X} = XW + b$$

where W is a $D \times D$ matrix of learnable weight parameters and b is a $D \times 1$ vector of learnable bias parameters. In particular, we can define separate query, key and value matrices (and their biases) each having their own independent linear transformations:

$$Q = XW^Q + b^Q$$

$$K = XW^K + b^K$$

$$V = XW^V + b^V$$

where the weight matrices W^Q , W^K and W^V and the bias vectors b^Q , b^K and b^V represent parameters that will be learned during the training of the final Transformer architecture. For now, let's ignore the bias vectors, as they are added element-wise and do not affect the dimensionality, but focus on the weight matrices, as they are more interesting in terms of dimensionality choices.

Here the matrix W^K has dimensionality $D \times D^K$ where D^K is the length of the key vector. The matrix W^Q must have the same dimensionality $D \times D^K$ as W^K so that we can form dot products between the query and key vectors.

Similarly, W^V is a matrix of size $D \times D^V$, where D^V governs the dimensionality of the output vectors. If we set $D^V = D$, so that the output representation has the same dimensionality as the input, this will facilitate the inclusion of residual connections (which we discuss later) and also facilitate multiple layers to be stacked on top of each other.

With these learnable transformations, we can reformulate self-attention as:

$$Z = \text{Softmax}(QK^T)V$$

Scaled Self-Attention

When we do self-attention, we compare queries and keys using their dot product QK^T and then feed those dot products into a softmax. Now, notice that if our query and key vectors are long (large $D_K = D_Q$), their dot product can become very large simply because we're summing over more dimensions. This is a problem for the softmax:

- If the numbers inside softmax are **large**, its outputs become overly **sharp** (almost one-hot): one position gets a probability close to 1, and all the others close to 0. That kills the gradients for most positions, so the model learns very poorly.
- Conversely, if the numbers are **tiny**, softmax becomes too **flat** and so it produces a nearly uniform distribution which makes the model struggle to focus attention where it's needed.

To solve this, we can **scale** the dot product dividing it by $\sqrt{D_K}$. This effectively **normalizes** the variance of the dot product, keeping it roughly independent of the vector dimension. As a result, the values passed to softmax remain in a stable range, keeping the outputs balanced — not too sharp, not too flat — and ensuring well-behaved gradients.

Where does $\sqrt{D_K}$ come from?

Assume each component of a query Q and a key K is an independent random variable with mean 0 and variance 1. The dot product between Q and K , each of dimension D_K , is:

$$Q \cdot K = \sum_{i=1}^{D_K} Q_i \cdot K_i$$

We know that the variance of a sum of independent random variables equals the sum of the variances of those variables. Thus, since $\text{Var}(Q_i \cdot K_i) = 1 \forall i$, the variance of the dot product is:

$$\text{Var}(Q \cdot K) = D_K$$

and, as consequence, the standard deviation is:

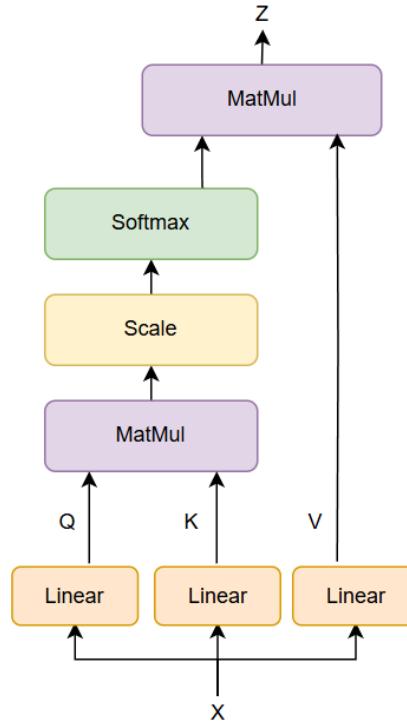
$$\text{Std}(Q \cdot K) = \sqrt{D_K}$$

Therefore, dividing the dot product by its standard deviation ($\sqrt{D_K}$) normalizes its scale, ensuring its variance is approximately 1 regardless of dimensionality.

Considering the *scaling*, the attention computation then becomes:

$$Z = \text{Softmax} \left(\frac{QK^T}{\sqrt{D_K}} \right) V$$

This is called **Scaled Dot-product Self-Attention** and represents the final form of the self-attention computation used in Transformer architectures.



Multi-head Attention

The scaled dot product attention described so far allows each output vector to attend to data dependent patterns of input vectors based on learned dependencies. This single computation is referred to as an **attention head**. However, often there might be multiple patterns of attention that are relevant at the same time and a single weighted average given by this single attention head is not a good option for it. In fact, using a single attention head can lead to averaging over these effects, thereby diluting important distinctions.

To address this limitation, the Transformer architecture introduces **Multi-head Attention**, which extends the attention mechanism to operate across multiple learned attention heads (similarly to multiple filters in CNNs).

Instead of computing attention once, multi-head attention computes it multiple times in parallel, using different learned projections of the same input. This means that for each attention head, we learn a separate sub-set of **query**, **key** and **value** transformations. Each head can then focus on different aspects of the input, effectively capturing different **representation subspaces**.

Formally, suppose we have H attention heads indexed by $h = 1, \dots, H$. For each of them we define separate query, key and value matrices:

$$Q_h = XW_h^Q + b_h^Q$$

$$K_h = XW_h^K + b_h^K$$

$$V_h = XW_h^V + b_h^V$$

Each head then computes its output using scaled dot-product attention:

$$Z_h = \text{Attention}(Q_h, K_h, V_h) = \text{Softmax}\left(\frac{Q_h K_h^T}{\sqrt{D_K}}\right) V_h$$

Therefore, we end-up with **H different output matrices Z_h** , each of dimension $N \times D/H$. However, we would like to have a single matrix (one for each token) of the same shape as the input (i.e. $N \times D$) so that we can pass the result onto subsequent layers in the network. Preserving the dimensional consistency will also be a requirement for residual connections. So, we need a way to condense these H outputs down into a single matrix.

To resolve this, we **concatenate** the outputs from all heads along the feature dimension:

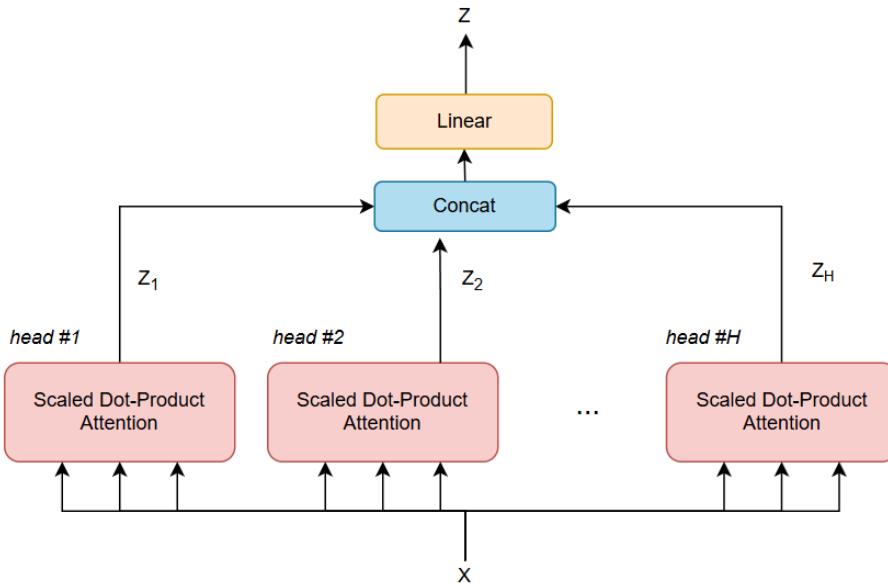
$$Z_{\text{concat}} = \text{Concat}[Z_1, Z_2, \dots, Z_H]$$

The result is then linearly transformed using an additional *Linear layer* to give a combined output that captures the information from all heads:

$$Z = Z_{\text{concat}} W^O + b^O$$

Here the weight matrix W^O and the bias vector b^O are also learned during the training process.

This produces the final output Z having dimensions $N \times D$ (the same shape as the original input matrix X) and resulting in a token's representation that has now been enriched with multiple perspectives — capturing diverse patterns of dependencies from the input — by attending to various relationships captured across multiple heads.



Encoder

Now that we've laid the groundwork, we can introduce how an **Encoder** in a Transformer is structured.

As mentioned earlier when introducing Transformer architecture, the encoder is composed by stacking **N identical blocks**. This is done because we know that neural networks benefit greatly from depth since it significantly enhances the network's capacity to learn complex patterns. However, stacking many layers

introduces challenges in training, particularly with gradient flow. To address this, each encoder block (following the principles established in ResNets) incorporates **residual connections** that help to maintain a smooth gradient flow through the network during backpropagation. In particular, a first residual connection is introduced to bypass the multi-head structure.

Note: Of course, for this to work, it is necessary that the output dimensionality is the same as the input dimensionality, namely $N \times D$.

This is then followed by **layer normalization**, ensuring that the distribution of activations remains well-behaved across layers (helps prevent exploding or vanishing gradients and accelerates convergence). The resulting transformation can be written as:

$$Y = \text{LayerNorm}[Z(X) + X]$$

where Z is defined by the multi-head self-attention seen above.

We are still missing a key component of the encoder. We have seen that the attention mechanism creates linear combinations of the value vectors, which are then linearly combined to produce the output vectors. Also, the values are linear functions of the input vectors, and so we see that the outputs of an attention layer are constrained to be linear combinations of the inputs. Although nonlinearities are introduced via the softmax function by making the outputs depend nonlinearly on the inputs through it, the output vectors are still constrained to lie in the subspace spanned by the input vectors and this limits the expressive capabilities of the attention layer.

To overcome this limitation, the above transformation is followed by a standard nonlinear **feed-forward neural network (FFN)**. This FFN introduces **additional nonlinearity** into the model. A typical configuration uses a two-FC layers network with a ReLU activation between the layers. However, this needs to be done in a way that preserves the ability of the transformer to process sequences of variable length (in fact a FFN requires to specify the dimensionality of the input). To achieve this, the same FFN is shared across all positions in the sequence. This means that each row of Y (corresponding to a token embedding) is processed independently and in parallel, but all pass through the same learned FFN.

Therefore, while the multi-head self-attention layer allows tokens to attend to each other and introduces dependencies between them, the FFN processes each token **independently**, adding representational power without interfering with sequence order or variable-length support.

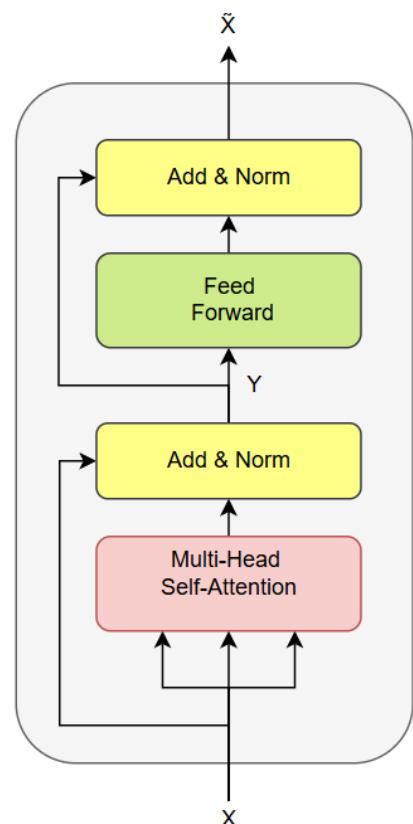
→ There are dependencies between the token embeddings in the multi-head self-attention layer, while the FFN does not have those dependencies.

To further improve this sublayer, the FFN is also wrapped in a **residual connection**, followed again by **layer normalization**, producing the final output of the encoder block:

$$\tilde{X} = \text{LayerNorm}(FFN(Y) + Y)$$

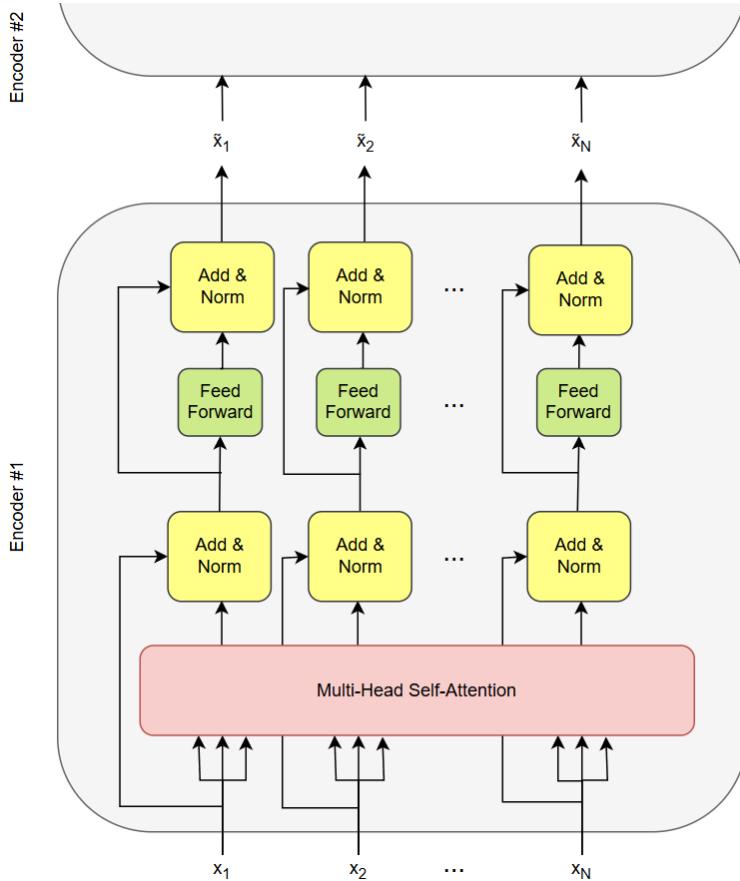
This completes the structure of a Transformer **encoder block** which consists of:

1. Multi-head self-attention with residual connection and layer normalization.
2. Feed-forward neural network, also with residual connection and layer normalization.



Note: In the Figure above, the structure may not make it visually obvious that **each row of Y** (i.e., each token embedding) is processed independently by the FFN. To make this clearer, the next Figure separates the different embeddings within X , illustrating how each one flows through the same shared FFN independently.

The best thing here is, unlike the case of RNN, in a Transformer each of these embeddings is independent of one another. So, we can apply parallelization and that makes all the difference!



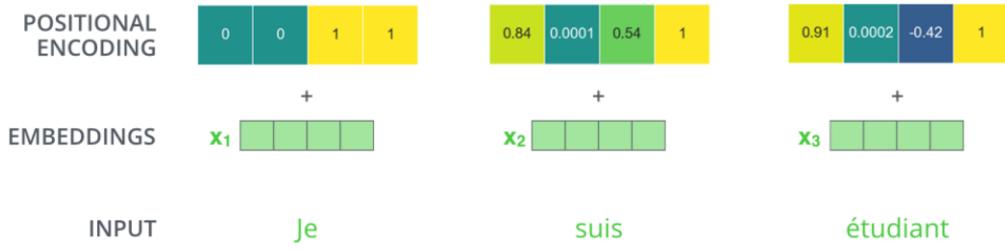
Positional Encoding

An important aspect we should have noticed is that Multi-Head Attention, as it is, has no inherent mechanism for capturing the order of tokens in the input sequence. This poses a significant challenge when working with sequential data, where the order of tokens carries crucial meaning. So, we need to find a way to inject information about token positions into the network. However, we also wish to retain the powerful properties of the attention layers that we have carefully constructed so far.

To address this limitation, Transformers employ a technique called **positional encoding**: the core idea is that, without changing the transformer architecture, what we can do is encode the token order in the input data itself. To do it we can construct a positional embedding PE associated with each input position pos and then combine this with the associated input token embedding. One obvious way to combine these vectors would be to concatenate them, but this would increase the dimensionality of the input space and hence of all subsequent attention spaces, creating a significant increase in computational cost. Instead, we can simply **add** the position embeddings onto the token embeddings to give:

$$\tilde{x}_{pos} = x_{pos} + PE_{pos}$$

This addition requires that the positional embeddings have the same dimensionality as the token embeddings.



At first it might seem that adding position information onto the token vector would corrupt the input vectors and make the task of the network much more difficult. However, some intuition as to why this can work well comes from noting that two randomly chosen uncorrelated vectors tend to be nearly orthogonal in spaces of high dimensionality, indicating that the network is able to process the token identity information and the position information relatively separately.

Note also that, because of the residual connections across every layer, the position information does not get lost in going from one transformer layer to the next. In fact, residual connections are essential to keep information about the original sequence. Removing the residual connections would mean that this information is lost after the first attention layer, and with a randomly initialized query and key vector, the output vectors for a given position has no relation to its original input. All outputs of the attention are likely to represent similar/same information, and there is no chance for the model to distinguish which information came from which input element.

Okay, said that, the next step is to determine how to construct the positional embeddings PE_{pos} . A naive solution would be to assign an increasing integer value to each position in the sequence (i.e. 1,2,3,...). However, this has the problem that the magnitude of the value increases without bound and therefore may start to corrupt the embedding vector significantly. Also, it may not generalize well to new input sequences that are longer than those used in training, since these will involve coding values that lie outside the range of those used in training. Another option might be to assign a number in the range (0,1) to each token in the sequence, which keeps the representation bounded. However, this representation is not unique for a given position as it depends on the overall sequence length.

Ideally, positional encodings should meet several criteria: they should provide unique, bounded representations for each position; they should generalize to sequences longer than those used in training; and they should have a consistent way to express the number of steps between any two input vectors irrespective of their absolute position because the relative position of tokens is often more important than the absolute position.

There are various methods for incorporating positional information into Transformer models, but the approach introduced by Vaswani et al. in the original paper is one of the most widely adopted. This method, known as **sinusoidal positional encoding**, is based on encoding positions using a combination of sine and cosine functions at different frequencies.

To build intuition for this method, it helps to first consider how numbers are represented in binary. For example:

0: 0000	5: 0101
1: 0001	6: 0110
2: 0010	7: 0111
3: 0011	8: 1000
4: 0100	9: 1001

In this representation, you can observe a pattern in how each bit (column) changes:

- The rightmost bit (Least Significant Bit) alternates with every number (frequency=1/2)
- The second bit from the right alternates every two numbers (frequency=1/4)
- The third bit alternates every four numbers (frequency=1/8)
- And so on...

This pattern — based on different frequencies — is fundamental to sinusoidal positional encoding. However, instead of using discrete binary digits, this method employs **smooth periodic functions**, specifically sine and cosine waves, to represent positional information in a continuous and differentiable manner.

In sinusoidal positional encoding, for each position in the sequence, a corresponding vector is created. Each component of this vector is computed using either a sine or a cosine function. The frequency of the function varies with each dimension of the vector. More formally, for a given position pos , we have an associated position-encoding vector where each of its dimensions i is given by:

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

Here, D is the dimensionality of the model's embeddings, which ensures that the positional encoding vector and the token embedding vector have the same shape, allowing them to be added together element-wise. Each dimension of the positional encoding corresponds to a sinusoid with a specific wavelength.

The authors of the Transformer model chose to scale the sine and cosine functions using powers of $1/10000$ in the denominator so to encode a wide range of frequencies from very high (rapid changes) to very low (slow changes) across dimensions. This design spreads the frequencies logarithmically across the embedding dimensions, which results in a *geometric progression* of wavelengths from 2π to $10000 \cdot 2\pi$.

Therefore, the elements (dimensions) of each positional embedding are given by a series of sine and cosine functions of steadily increasing wavelength. Moreover, the values of these sine and cosine functions are always bounded between -1 and 1. This ensures that the positional encodings don't grow or shrink excessively for very long sequences.

A plot of the position-encoding embeddings is shown in the Figure below, where the y-axis represents the token position in the input sequence and the x-axis corresponds to the dimensions of the positional embedding:

- Each row corresponds to the encoding for a particular position, similar to how each binary pattern represents a specific number.
- Each column corresponds to one dimension in the embedding vector, analogous to bit positions in binary representation.

The color scale in the plot reflects values that oscillate smoothly between -1 and 1, which can be viewed as a continuous analog to the discrete 0s and 1s in binary representation.

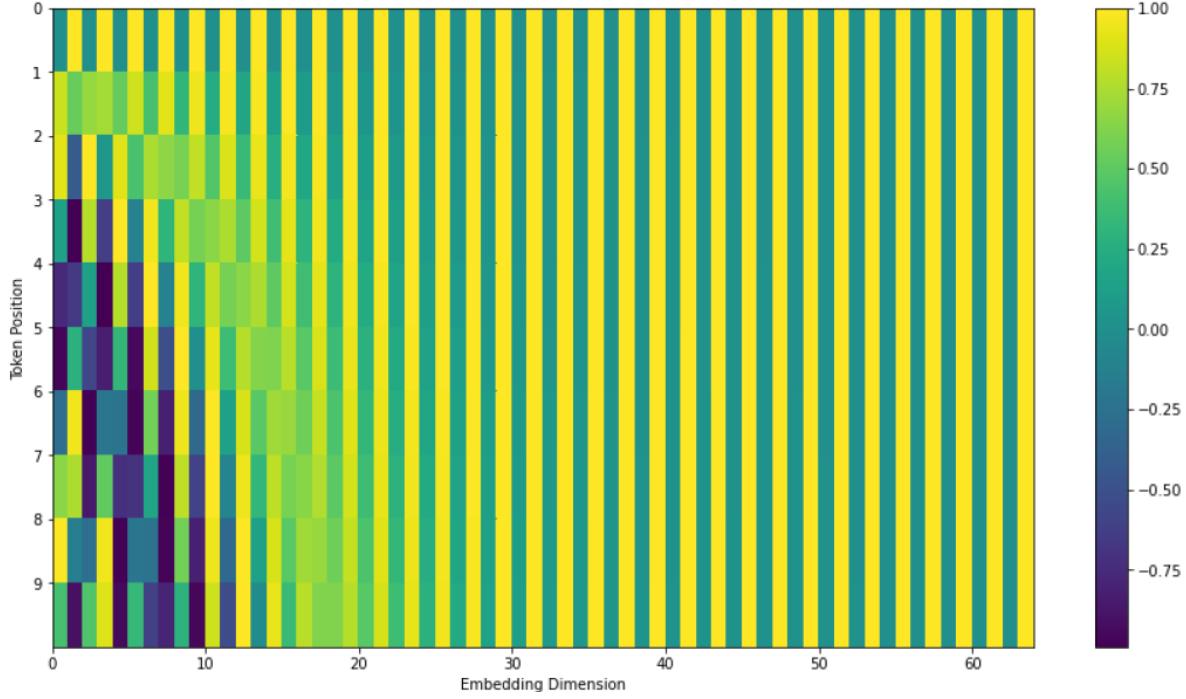
Notably, you should observe that:

- The leftmost columns (lower dimensions) change rapidly, like the least significant bits in binary.
- The rightmost columns (higher dimensions) change more slowly, analogous to the most significant bits in binary.

Therefore, in the first part of the plot, the differences between the positions are evident, indicating that these positions are particularly significant. However, as the size of the embedding increases, the differences between

the various positions become less and less marked because the effect of the sinusoidal part becomes less and less relevant.

Furthermore, we can notice that there is an alternating sequence of 0 (dark green) and 1 (yellow) towards the right (i.e. towards the end of the vector elements) of the plot. This is due to the alternating use of sine and cosine functions in the encoding formula, which creates a checkerboard-like structure as the embedding index increases.



One particularly valuable property of sinusoidal positional encoding is that, *for any fixed offset k , the embedding at position $pos + k$ (PE_{pos+k}) can be represented as a linear combination of the embedding at position pos (PE_{pos}),* in which the coefficients do not depend on the absolute position but only on the value of k . This means the network has the potential to **implicitly infer relative positions**, which makes it easier to learn relationships that depend on how far apart tokens are, rather than their absolute locations.

Note: Despite this property, sinusoidal encoding is still an **absolute positional encoding** — it does not explicitly represent relative distances. Instead, the ability to infer relative positions arises indirectly from its mathematical structure. Later in this chapter, we will explore alternative positional encoding methods, such as **relative positional encodings** and **RoPE**, which **explicitly** address relative positioning and often perform better in tasks requiring long-range dependencies.

Another common approach to positional encoding is the use of **learned positional encodings**. This is done by having a vector of weights at each token position that can be learned in conjunction with the rest of the model parameters during training. However, this approach does not meet the criteria we mentioned earlier of generalizing to longer input sequences as the encoding will be untrained for positional encodings not seen during training. Therefore, this approach is best suited in situations where the input length remains relatively constant during both training and inference.

During their study, the authors of the Transformer's paper also experimented with the use of learned positional encodings and found that both approaches produced comparable results. Ultimately, they adopted the sinusoidal approach for the reasons just discussed above.

Classification Head

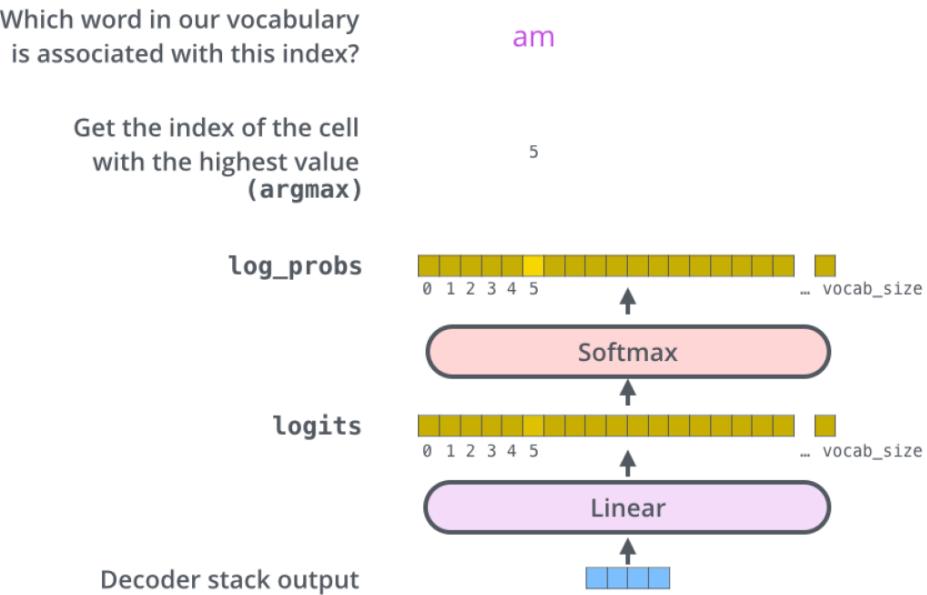
Before introducing the decoder in detail, it's helpful to first discuss an additional component placed on top of the final decoder block: the **classification head**. We introduce this component early because it's tightly connected to how the Transformer decoder works.

The classification head consists of a **Linear layer**, which makes a linear transformation of the decoder output using a matrix W^Y of dimensionality $D \times K$, followed by a **Softmax** activation function:

$$\hat{Y} = \text{Softmax}(\tilde{X}W^Y + b^Y)$$

where \tilde{X} indicate the output from the decoder — specifically, a matrix where each row corresponds to the embedding of an output token at a particular time step.

The Linear layer is used to project the output of the decoder in a much larger vector called the **logits vector**. This is chosen to have a dimensionality equal to the length of the dictionary so that each element of it represents a logit (unnormalized score) for a unique token in the dictionary. For instance, if the dictionary is made of K tokens, this vector will be of length K . The softmax function then transforms this vector into a **probability distribution** over all tokens in the dictionary — ensuring all values are non-negative and sum to 1. The element in the resulting vector with the highest probability is chosen, and the token associated with it is produced as the output for that time step.



Let's better understand this.

Basically, the goal of the decoder part in the original Transformer is to generate output sequences of tokens in the translated language and this is done in an autoregressive manner. This means that the model generates tokens one at a time, and each prediction is conditioned on all the previously generated tokens.

At time step n , the decoder takes as input the sequence of tokens t_1, t_2, \dots, t_{n-1} that it has already produced. Its output is then passed to the classification head to obtain **conditional probability distribution** over the vocabulary. This distribution assigns a probability to every possible token in the vocabulary, reflecting how likely each token is to be the next word given the preceding context:

$$p(t_a|t_1, t_2, \dots, t_{n-1}), p(t_b|t_1, t_2, \dots, t_{n-1}), \dots$$

There are several strategies for selecting the next token from this distribution. For now, to keep the explanation simple, we assume a greedy approach: select the token with the highest probability as the next token t_n . This predicted token is then appended to the previously generated sequence, and the updated sequence is fed back into the decoder as input to predict the next token t_{n+1} . This process repeats until the model predicts a special token, the “end-of-sequence” token denoted as **<EOS>**, signaling that the decoder has completed its output.

Let's make this concrete with an example. Suppose the current generated sequence is:

“Thanks for all the”

The decoder processes this sequence, and its output is passed to the classification head which produces a probability distribution over the vocabulary which looks like:

$$p(\text{fish}|\text{Thanks, for, all, the}) = 0.13, \quad p(\text{support}|\dots) = 0.84, \quad p(\text{gun}|\dots) = 0.01, \dots$$

Since “support” has the highest probability, the model selects it as the next token, extending the sequence to:

“Thanks for all the support”

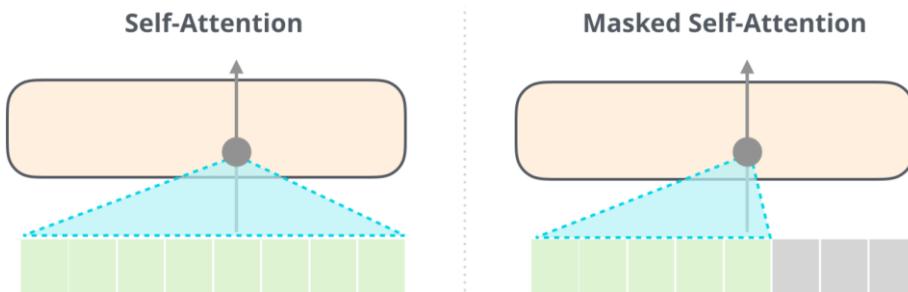
This is then again given in input to the decoder to predict the next token in the output sequence and this process continues token by token until the **<EOS>** token is generated.

Decoder

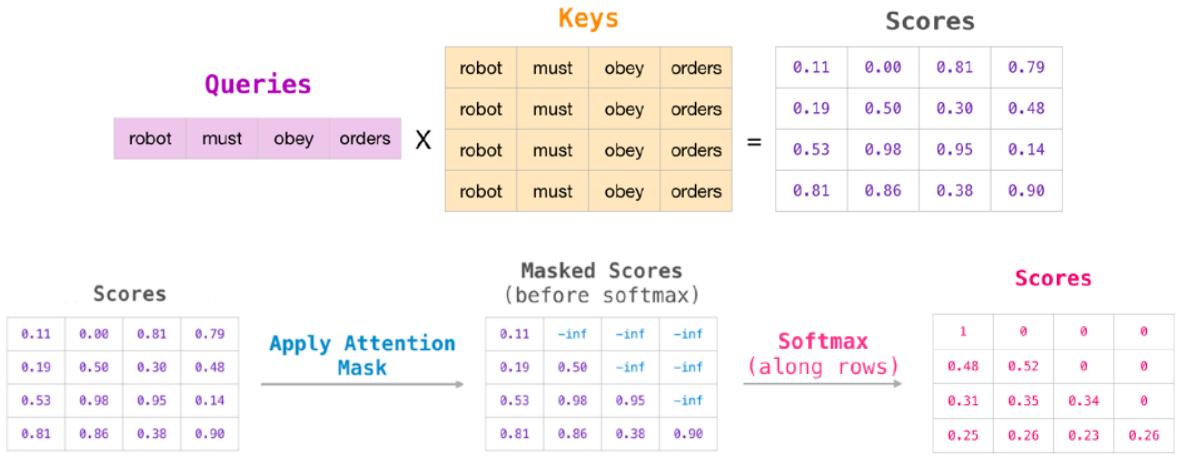
As previously mentioned, the decoder's primary role is to generate the output sequence **autoregressively** — that is, one token at a time, using the tokens it has already generated. Therefore, it takes as input the tokens generated so far. Like with the encoder inputs, the decoder inputs are first embedded and then positional encoding is added to them. Importantly, for the first time step, the decoder (that haven't generated any token yet) receives in input a special **<START>** token to initiate the generation process. As the sequence is generated, each newly predicted token is fed back into the decoder for the next step.

Like the encoder block, each decoder block consists of a similar set of sub-layers. It includes two multi-headed attention layers and a feed-forward layer, where each sub-layer is followed by residual connections and layer normalization. The key differences from an encoder block lie in the two multi-head attention layers adopted:

- **Masked Multi-Head Self-Attention:** The first attention layer in the decoder uses a **masked self-attention** mechanism. It is similar to the encoder's self-attention, with one major difference: **it can only attend to earlier positions in the output sequence**.



This is achieved by applying a **causal mask** (also known as a **look-ahead mask**) that prevents the model from accessing future tokens. In practice, this mask is implemented as a matrix, called **masked attention matrix**, that assigns a value of $-\infty$ to all positions representing future tokens, causing the softmax function to output 0 attention weights for them (see Figure below). This ensures that each token can only attend to preceding tokens, preserving the autoregressive property and preventing the model from “cheating” by looking ahead.

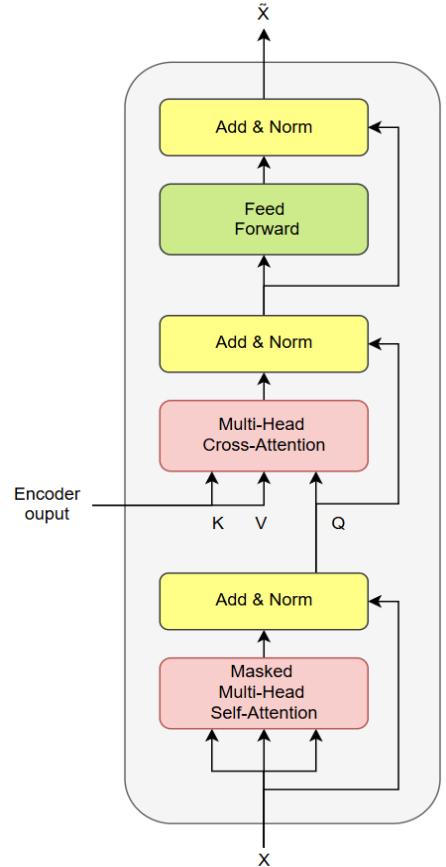


Note: The word "*causal*" comes from causality — the principle that a cause precedes its effect. In the context of sequence generation, this means a token can only be influenced by previous tokens that came before it, not by those that follow.

- **Cross-Attention (Encoder-Decoder Attention):** The second attention layer allows the decoder to attend also to the encoder's outputs. This mechanism is referred to as **cross-attention** or **encoder-decoder attention**. Unlike self-attention, in which queries, keys and values all come from the same sequence, in cross-attention:
 - **Queries** come from the sequence being generated (or from the decoder's previous block output)
 - **Keys and Values** come from the output of the last encoder block of the encoder stack

This layer enables the decoder to align its generation with relevant parts of the input sequence, effectively guiding the output based on the encoded input context.

After these two multi-head attention layers, the resulting output is finally passed through a feed forward network (that is exactly as in the encoder), which refines the information and enhances the overall representation.



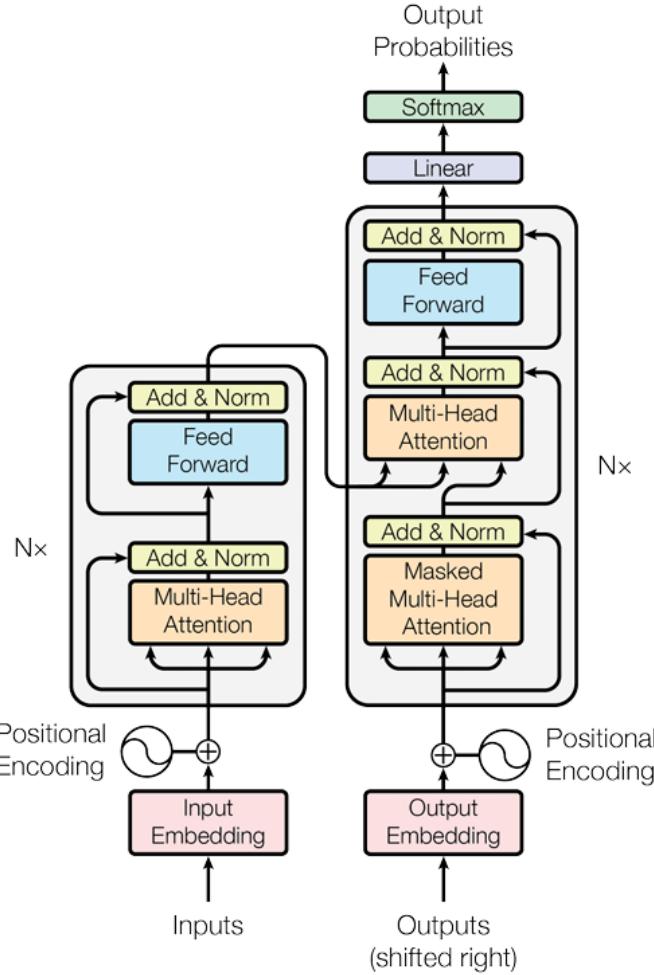
Encoder-Decoder Transformer

The overall architecture proposed in the paper is illustrated in the Figure below. It consists of a stack of $N = 6$ **identical layers** for both the encoder and the decoder.

To represent input and output tokens as vectors, the model uses **learned embeddings** of dimension $D = 512$. Sequences fed into both the encoder and the decoder are limited to a maximum length of 512 tokens. This limitation stems from the use of positional encodings, which were precomputed up to a fixed length of 512 in order to be added element-wise to input embeddings.

In the multi-head attention layers, the model uses **$H = 8$ attention heads**. Given the overall embedding dimension of 512, each head uses projections with dimensions $D_Q = D_K = D_V = \frac{D}{H} = 64$. This ensures that the

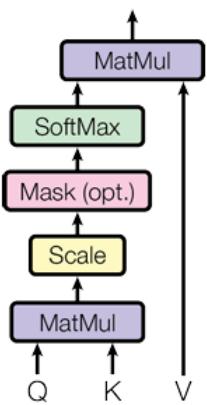
output of all heads when concatenated results in a final attention output of size $8 * 64 = 512$, matching the original embedding dimension.



One aspect that often goes unnoticed in the Transformer architecture is the **optional masking of specific entries in the attention mechanism** (shown in the original *Scaled Dot-Product Attention* Figure as "Mask (opt.)"). This type of masking is particularly important when working with **batches of sequences** that vary in length.

In NLP tasks, **input sequences** (sentences) often differ in length. However, in order to take advantage of the **parallel processing capabilities of GPUs**, we need to process multiple sequences simultaneously in **batches**. To make this possible, we **pad shorter sequences** with a special <PAD> token so that all sequences in a batch are the same length.

While this padding enables batching and parallel computation, it introduces a problem: the **padding tokens carry no semantic meaning**, yet they are included in the attention computation. If not handled properly, the model might waste attention capacity or even learn undesirable behaviors by attending to these meaningless <PAD> tokens. To prevent this, we apply an **attention mask** which ensures that the attention mechanism **ignores padded positions** by setting their attention logits to $-\infty$ so that when the softmax is applied, these positions get zero attention weight, effectively removing them from consideration.



Training an Encoder-Decoder Transformer

As already mentioned, before training a Transformer model, a preprocessing phase is required to construct the vocabulary. This is done through a process called *tokenization*, applied to the large corpus that will serve as training data. In the original Transformer, the authors used **Byte-Pair Encoding (BPE)** on the WMT 2014 English-French dataset, which contains 36 million sentence pairs. The result of this tokenization is a vocabulary of $K = 32000$ tokens used for both the source (English) and target (French) sequences ([shared](#)). In this tokens count are also included the special tokens of `<START>`, `<EOS>` and `<PAD>`. This vocabulary is therefore fixed before training and remains unchanged throughout.

When preparing data for training, particularly for sequence-to-sequence tasks like machine translation, the source sentence (e.g., in English) is passed to the encoder exactly as it is, without any special modifications. On the other hand, the target sentence (in French) is used in two ways within the decoder. First, it is kept as the target output, representing the ground truth that the model is trying to predict. Second, the target sequence is used as input to the decoder, but with a modification made to account the decoder auto-regressive nature: the sequence is shifted to the right by one position. This ensures that for each position i , the input x_i corresponds to the output y_{i+1} with target output x_{i+1} . To achieve this, the special `<START>` token is prepended to the beginning of the input sequence, and the final token is dropped to maintain the correct sequence length.

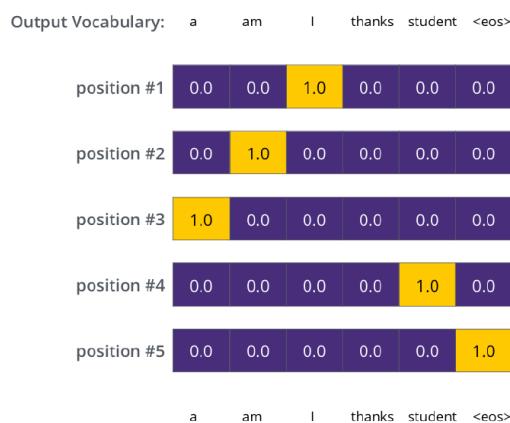
For instance, if the target sequence is [`<je>`, `<suis>`, `<étudiant>`, `<EOS>`, `<PAD>`, `<PAD>`], the decoder input becomes [`<START>`, `<je>`, `<suis>`, `<étudiant>`, `<EOS>`, `<PAD>`], while the target output remains [`<je>`, `<suis>`, `<étudiant>`, `<EOS>`, `<PAD>`, `<PAD>`].

Note: Of course, the `<PAD>` tokens even if present are ignored during loss computation.

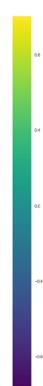
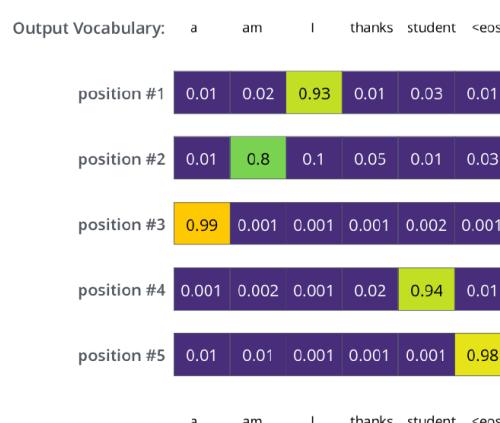
The decoder's predicted output is compared to the target output using **cross-entropy loss**. In practice, the targets are not represented as full one-hot vectors but rather as integer token IDs. For example, instead of encoding the correct token as [0, 0, 1, 0], we simply pass the integer index 2 (assuming the correct token is at position 2 in the vocabulary). Internally, the loss function behaves as though it were compared against a one-hot vector: it extracts the predicted probability for the correct token's index and takes the negative logarithm of that value. This is mathematically equivalent to using one-hot vectors but avoids the memory and computational overhead. For instance, if the model outputs predicted probabilities [0.1, 0.3, 0.5, 0.1] and the ground-truth token has index 2, the loss is computed as $-\log(0.5)$. No explicit one-hot representation is needed.

To help build intuition, the Figure below shows a simplified example where the ground-truth tokens are represented as one-hot vectors for illustrative purposes (image on the left). Alongside this, the model's predicted outputs after training are shown (image on the right), highlighting the kind of distributions the model is expected to learn by the end of training.

Target Model Outputs



Trained Model Outputs



Finally, during training, sentence pairs were batched together based on approximate sequence length to optimize computation and reduce the amount of padding. Each batch contained around 25,000 source tokens and 25,000 target tokens, enabling efficient use of GPU parallelism.

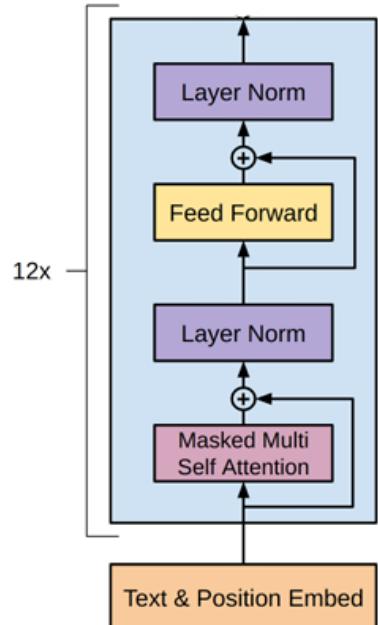
GPT

The first GPT model, introduced by Radford et al. (OpenAI) in 2018 with the paper *Improving Language Understanding by Generative Pre-Training* [50], marked the beginning of the **Generative Pretrained Transformer (GPT)** series. Its architecture was based on the **decoder portion** of the original Transformer model, with some modifications — most notably, the **removal of cross-attention** mechanisms, as the model was designed to operate in a purely autoregressive manner without the need to attend to encoder outputs.

Internally, GPT consists of a stack of identical blocks, each composed of **masked multi-head self-attention** followed by feed-forward layers. After the final transformer block, a linear projection and softmax are used to produce a probability distribution over the vocabulary for the next token prediction.

The architecture uses **12 transformer layers**, **12 attention heads** and a **768-dimensional embedding**. Each feed-forward layer expands the hidden representation to **3072 dimensions**, applies a **GeLU non-linearity** (instead of a ReLU) and projects it back down to the model dimension (so to permit residual connections to be added).

The model is often referred to as a "*decoder-only Transformer*" because it adopts only the decoder component of the original Transformer architecture. However, many researchers — including myself — consider this terminology misleading. The model no longer performs decoding in the traditional sense of sequence-to-sequence architectures. Instead, it operates in a purely autoregressive fashion, predicting tokens from left to right based solely on prior context. For this reason, a more precise and meaningful term would be **Causal Transformer**. This alternative naming better reflects the model's left-to-right causal structure, and there's a growing movement within the research community to adopt this terminology. I also strongly advocate for this shift in nomenclature.



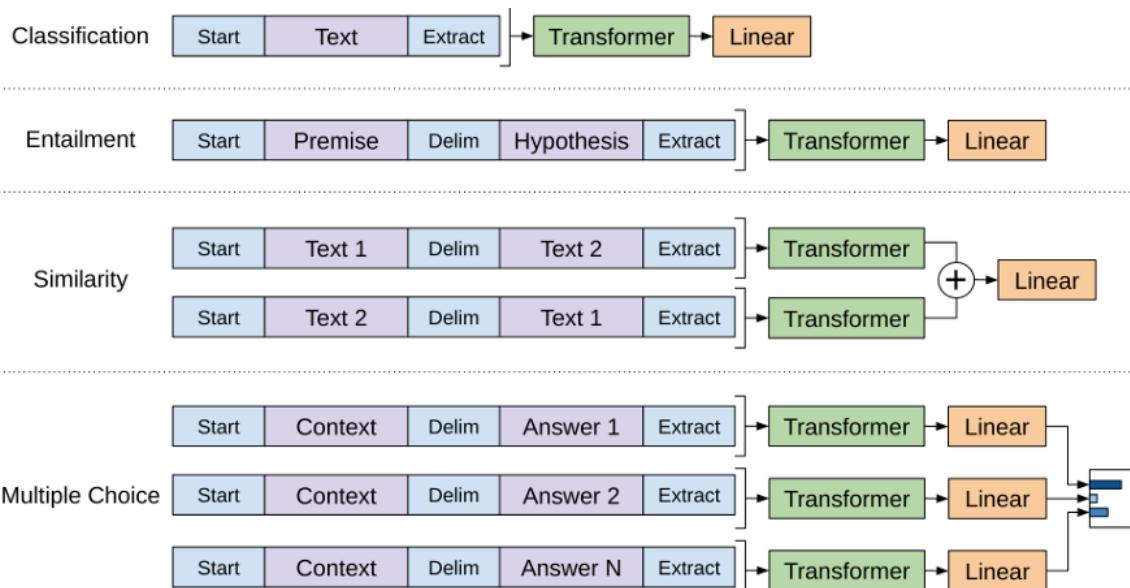
GPT does not introduce any significant architectural innovations, but what makes it particularly interesting is its training. The training process is divided into two main phases: pre-training and fine-tuning.

During **pre-training**, the model learns by **next token prediction** task in a sequence in an **autoregressive** manner. For the pre-training the data comes from **BooksCorpus**, and tokenization is performed using **BPE** with a vocabulary of **40,000 tokens**. Pre-training was conducted over **100 epochs**, using batches of 64 sentences.

The second phase, **fine-tuning**, aligns the pretrained model to specific downstream tasks using a small amount of labeled data. Crucially, this is done **trying to not change the model architecture as much as possible**. Instead, **GPT uses input transformations** to reformat task-specific inputs into a sequence that the pre-trained model can process. Here we list some of these input transformations shown in the Figure below:

- **Textual entailment:** For entailment tasks, the premise and hypothesis token sequences are concatenated with a delimiter token (<\$>) in between.

- **Similarity:** For sentences similarity tasks, there is no inherent ordering of the two sentences being compared. To reflect this, the input sequence is modified to contain both possible sentence orderings with a delimiter in between and process each independently to produce two sequence representations which are added element-wise before being fed into the linear output layer.
- **Question Answering:** For question answering tasks, we are given a context document, a question and a set of possible answers. The document context and question are concatenated with each possible answer, adding a delimiter token in between to get $[context; question; <\$>; answer_k]$. Each of these sequences are processed independently with the model and then normalized via a softmax layer to produce an output distribution over possible answers.



Fine-tuning was performed using smaller batch sizes (e.g., 32), a lower learning rate and just 3 epochs per task.

At this point, it's important to make a clarification — especially if you're reading the original paper — since it introduced some terminology that is incorrect. The GPT-1 paper refers to pretraining as an *unsupervised learning* phase because it does not use human-labeled data. However, this terminology is imprecise. In reality, the task of predicting the next token actually provides a **supervision signal** — it's just derived from the data itself, not from human annotation. This makes the approach an instance of self-supervised learning, rather than unsupervised learning. **Self-Supervised Learning (SSL)** is a subset of supervised learning where the targets (labels) are automatically obtained from the input data. For example, consider the sentence “*The cat sat on the mat*”. During training, the model learns by setting each next token as the target:

- Context: “*The*” → Target: “*cat*”
- Context: “*The cat*” → Target: “*sat*”
- Context: “*The cat sat*” → Target: “*on*”
- Context: “*The cat sat on the*” → Target: “*mat*”

At every step, the supervision signal is clear — the next word in the sequence — but it is derived directly from the data itself rather than from external labels. This approach contrasts with unsupervised learning, which typically refers to clustering, dimensionality reduction or density estimation without any label.

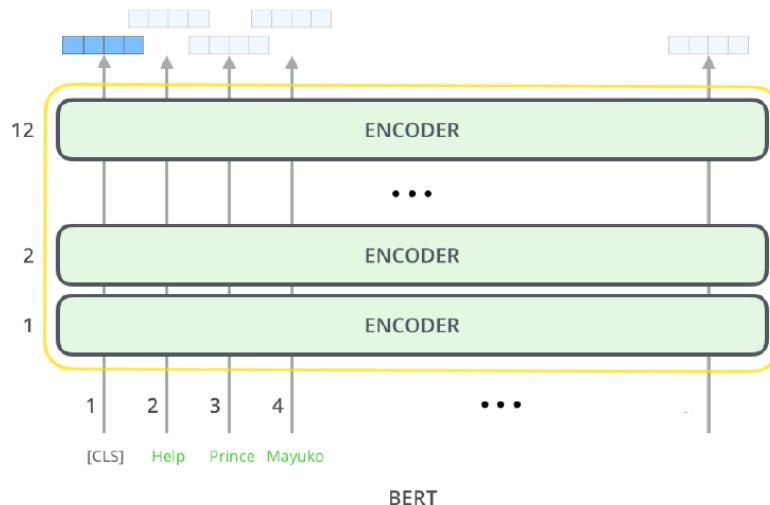
In summary, the central contribution of the GPT-1 paper is not a novel architectural innovation, but rather a shift in learning paradigm: it demonstrates that **a language model, trained on a general-purpose objective like next-word prediction, can be effectively adapted to a variety of downstream tasks through fine-tuning — without the need for task-specific architectures**. While GPT-1 set the foundation, the more impactful breakthroughs would emerge with GPT-2 and especially GPT-3.

BERT

GPT uses a unidirectional (left-to-right) architecture to learn general language representations during pre-training phase. This means that every token can only attend to previous tokens in the input sequence. While essential for generative new sentences, this constraint can be sub-optimal for sentence-level tasks and could be very harmful when applying finetuning based approaches to token-level tasks such as question answering, where it is crucial to incorporate context from both directions.

BERT (Bidirectional Encoder Representations from Transformers), introduced by Devlin et al. (Google) in 2018 [51], overcomes this limitation by using a **bidirectional Transformer encoder**, allowing the model to consider both the preceding and following context simultaneously. Soon we will see how we can ensure this behavior.

BERT architecture mirrors the original Transformer encoder design.



The authors presented two versions of BERT:

- **BERT Base** having 12 layers, 12 attention heads and an embedding dimension of 768, resulting in 110 million parameters;
- **BERT Large** having 24 layers, 16 attention heads and an embedding dimension of 1024, resulting in 340 million parameters.

BERT Base was chosen to have the same embedding dimension as GPT for comparison purposes.

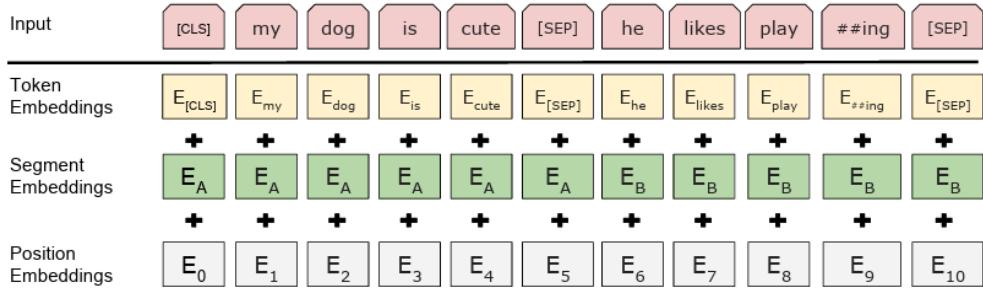
BERT, as GPT, follows a unified architecture across different tasks where there is minimal difference between the pre-trained architecture and the final downstream architecture.

To support a wide variety of downstream tasks, BERT's input representation is designed to encode either a single sentence or a pair of sentences (e.g., Question, Answer) as a single token sequence. **Therefore, in this context, “sequence” refers to the input token sequence to BERT model, which may be a single sentence or two sentences packed together, rather than an actual linguistic sentence.**

The first token of every sequence is always a special classification token <CLS> which is used during fine-tuning as an aggregate representation of the entire sentence for classification tasks. During pre-training, however, the output corresponding to this token is NOT used.

When a sentence pair is provided, the two sentences are concatenated and separated by a special <SEP> token. Additionally, to help the model distinguish between the two parts of the input, BERT introduces **segment embeddings**: a learned embedding is added to each token to indicate whether it belongs to sentence A or B.

Therefore, each token's input representation is constructed by summing the corresponding token, segment and position embeddings:



Pre-Training

We saw in GPT that each token can only attend to tokens to its left through masked self-attention and this was required to permit the training of the model via a next-token prediction task. However, given its bidirectional nature, BERT cannot be pre-trained with the same next-token prediction task since bidirectional conditioning would allow each token to indirectly "see itself" when predicting the next word, which would undermine the learning objective.

To enable truly bidirectional context modeling, BERT retains standard (non-masked) self-attention but modifies the pre-training objective. Instead of predicting the next token, BERT is trained using a **masked token prediction** task (in the paper it is called **Masked Language Modeling**, or in short **Masked LM**).

In this setup, some percentage of the input tokens – typically 15% – is masked at random with a **[MASK]** token and the model is trained to predict the original vocabulary id of the masked tokens based only on their surrounding context. To do this, the final output vectors corresponding to the mask tokens are fed into a softmax over the vocabulary and the predictions are compared with the original tokens using cross-entropy loss.

For example, in this setup an input sequence might be:

I <mask> across the river to get to the <mask> bank.

and the network should predict 'swam' at output node 2 and 'other' at output node 10. Notice that in this case only two of the outputs contribute to the error function and the other outputs are ignored.

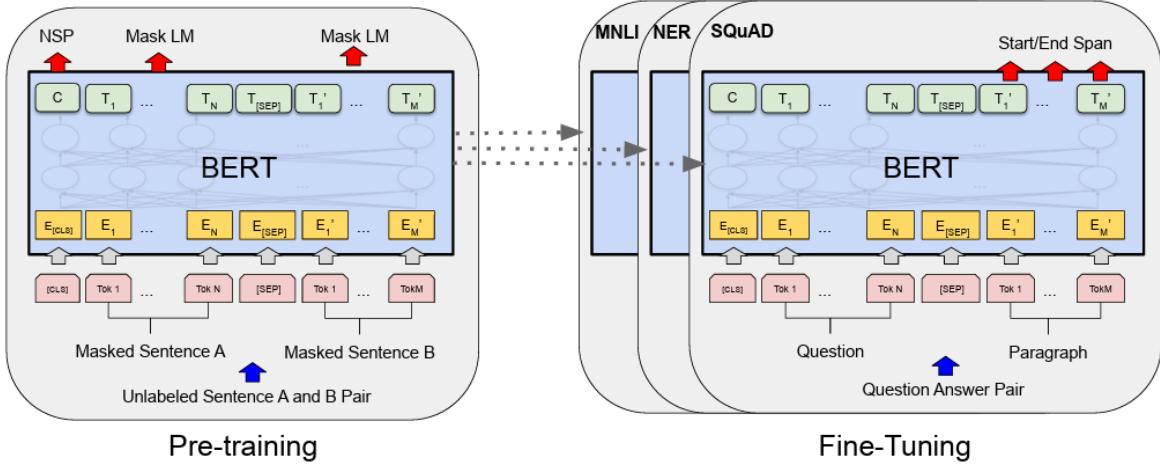
Although this approach allows us to obtain a bidirectional pre-trained model, a downside is that we are creating a mismatch between pre-training and fine-tuning, since the <MASK> token does not appear during fine-tuning. To mitigate this, BERT do not always replace "masked" words with the actual <MASK> token. Instead, of these 15% randomly selected tokens:

- 80% are actually replaced with <MASK>
- 10% with a random token
- 10% remain unchanged (use the original token).

In addition to masked token prediction, BERT introduces a second pre-training task called **Next Sentence Prediction (NSP)**. Many downstream applications, such as *Question Answering* and *Natural Language Inference*, require understanding relationships between sentences — something masked token prediction alone cannot capture. To train BERT for such tasks, next sentence prediction is formulated as a binary classification problem: given two sentences A and B, the model predicts whether B is the actual next sentence that follows A (labeled *IsNext*) or a randomly selected sentence (labeled *NotNext*). To permit this, during pre-training half of the sentence pairs (50%) are chosen to be positive examples (i.e. sentences that are consecutive in the corpus) and the other half are chosen to be negative examples (i.e. sentences randomly drawn from the corpus and that therefore are not consecutive). Since it is a binary classification task a BCE loss is used.

Finetuning

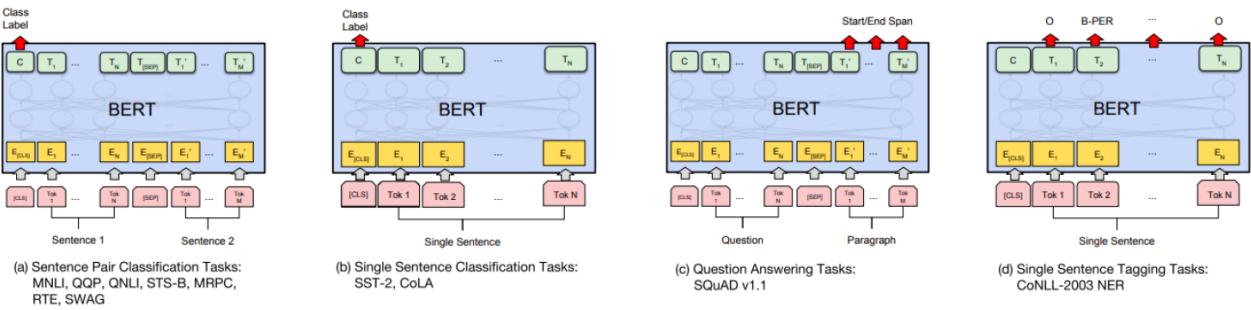
Once pre-training is complete, BERT can be fine-tuned on specific downstream tasks using labeled data. Although each downstream task results in a different model, all of them are initialized from the same pre-trained weights. During fine-tuning, the entire model — including all pre-trained parameters — is updated end-to-end.



Fine-tuning BERT is straightforward. Task-specific inputs and outputs are adapted to the model and a small task-specific layer is added on top.

For classification tasks, just the output corresponding to the <CLS> token is used, while for token-level tasks like sequence labeling or QA, the token-specific outputs are used directly and, in both cases, these are then passed to the given output layer. The output layer is typically a simple linear layer followed by a softmax.

Compared to pre-training, fine-tuning is relatively inexpensive and typically requires only a few epochs over relatively small datasets. In the Figure below there are illustrations of fine-tuning BERT on different down-stream tasks.



BERT for Features Extraction

BERT was designed to produce rich, contextualized embeddings that capture meaning based on both left and right context. These embeddings serve as powerful representations that can enhance performance across a variety of downstream NLP tasks.

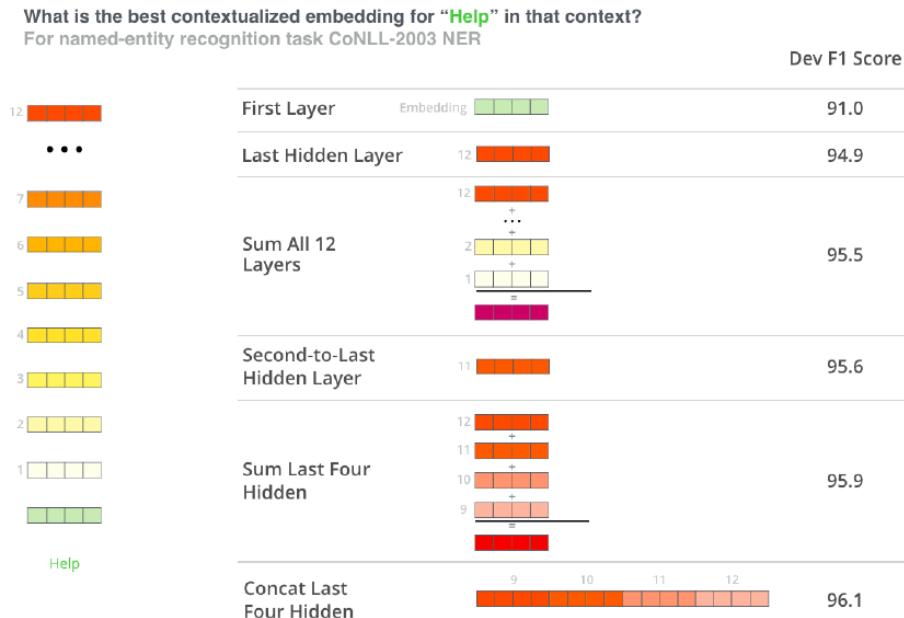
While fine-tuning BERT on a specific task is the most common usage, it's not the only approach; in fact, BERT can also be used as a **features extractor**. In this setup, a pre-trained BERT model is used to generate contextualized token embeddings, which are then passed as input features to another model without modifying BERT's parameters.

The original BERT paper explored this feature-based approach in tasks such as Named Entity Recognition (e.g., CoNLL-2003). The authors took output vectors from different layers of the BERT model and tested multiple strategies for extracting token-level embeddings: just use the last hidden layer; use the second-to-last layer; use a weighted sum of all layers; use a weighted sum of last four layers; concatenate the top four layers.

These embeddings were then passed as input to a randomly initialized two-layer BiLSTM with 768-dimensional hidden states. These variants were compared to a fully fine-tuned BERT model. The results showed that, although fine-tuning performs better, the feature-based approach can achieve highly competitive results.

Among the options tested, the best-performing method involved **concatenating the final four hidden layers** of BERT, yielding results close to the fine-tuned baseline — just 0.3 F1 points behind fine-tuning in the best case.

	Dev F1 Score
Fine-tuning approach	
BERT _{LARGE}	96.6
BERT _{BASE}	96.4
Feature-based approach (BERT _{BASE})	
Embeddings	91.0
Second-to-Last Hidden	95.6
Last Hidden	94.9
Weighted Sum Last Four Hidden	95.9
Concat Last Four Hidden	96.1
Weighted Sum All 12 Layers	95.5



Because of their rich semantic properties, BERT embeddings have found use beyond traditional classification tasks. For example, they can be used to compute sentence similarity — helping search engines like Google to match semantically similar queries even when there's no exact word overlap. They also serve as high-quality inputs to generative systems, such as models for text-to-image synthesis, where deep semantic understanding of language is critical.

Of course, since BERT is an **encoder-only** model, it cannot generate sequences like causal Transformers (e.g., GPT). But this is by design — BERT was not intended for generation, and its strength lies in producing high-quality comprehension and contextual awareness embeddings.

GPT-2

A core limitation of the GPT-1 approach was that, despite the model being architecturally task-agnostic, achieving strong performance on downstream tasks still required fine-tuning on task-specific large, labeled datasets. In other words, each new task necessitated a new dataset and dedicated fine-tuning step.

Therefore, GPT-2 authors in their paper *Language Models are Unsupervised Multitask Learners* (Radford et al., 2019) [52] set out to answer a fundamental question: “*Can we design a single model that performs well across a wide range of tasks without needing supervised data for each one?*”

Therefore, the authors investigated whether a single, large-scale model could exhibit **multitask learning** behavior — that is, the ability to handle different tasks — simply by training on large and diverse unlabeled text in a self-supervised manner, without requiring fine-tuning for each individual task.

Traditionally, learning to perform a single task can be expressed in a probabilistic framework as estimating a conditional distribution $p(\text{output}|\text{input})$. Since a general system should be able to perform many different tasks, even for the same input, it should condition not only on the input but also on the task to be performed. That is, it should model $p(\text{output}|\text{input}, \text{task})$. Task conditioning was often implemented at an architectural level in the years, but the authors here tried to implement it by introducing task information directly into the input sentence (using input transformations as we saw for GPT-1) without modifying the architecture. For example, a translation training example can be written as the sequence (*translate to french, english text, french text*). Likewise, a reading comprehension training example can be written as (*answer the question, document, question, answer*).

The most important aspect to underline is that model training is entirely focused on predicting the next token in a sequence. The model is never explicitly said which part corresponds to the task description, the input text or the expected output. Instead, it learns to identify these distinctions by recognizing patterns within the data during the training. Essentially, if the model becomes skilled at completing sequences and if tasks are presented as sequences where the desired output naturally follows the input, it can implicitly learn to perform those tasks.

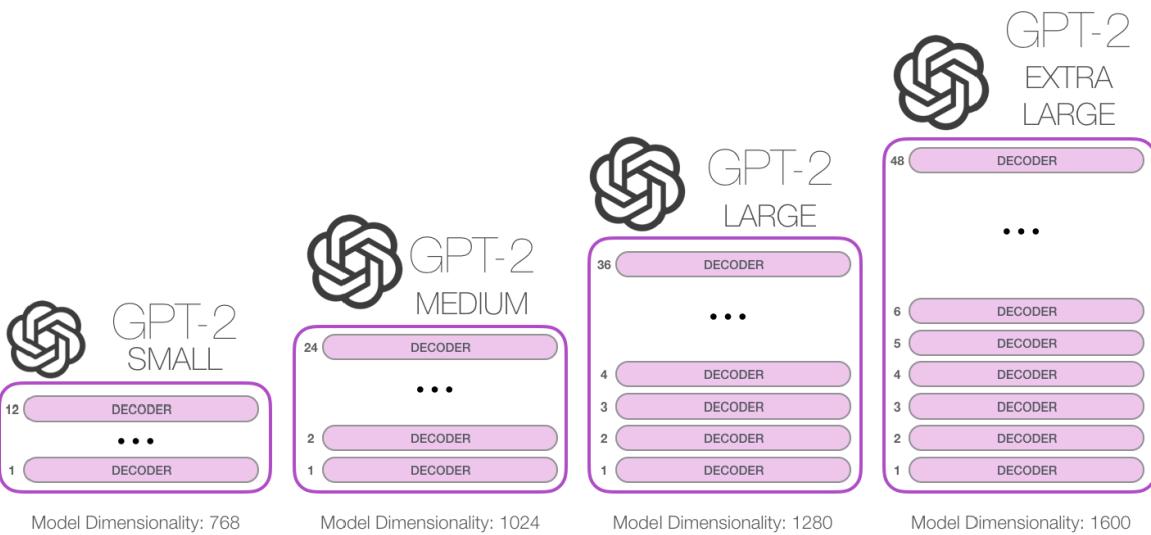
Interestingly, this formulation allows the same training objective (predict next token) to serve both for pretraining and eventually for fine tuning.

The dataset used for training GPT-2 was a critical point to permit this behavior. Previous language models had typically been trained on relatively narrow sources of text like news, Wikipedia or books. GPT-2’s authors wanted to push beyond that by constructing a training dataset that captured as much linguistic diversity as possible. They began by collecting web pages but recognized that much of the raw web is low quality — filled with spam, incoherent content and irrelevant material. To address this, they filtered the content using a clever heuristic: they only kept web pages that had been linked to from Reddit posts that received at least three upvotes. This simple metric — user engagement — served as a *proxy for quality*. This filtered dataset, called **WebText**, ended up with over 8 million documents and approximately 40 GB of text after cleaning and deduplication.

Moreover, another important contribution of GPT-2 was its approach to tokenization: they decided to use a **byte-level version of BPE**. In the traditional BPE the process begins by treating individual characters as the initial units and then iteratively merges frequently occurring sequences of characters into larger tokens. While this approach works well in smaller-scale settings, applying it to a massive and diverse dataset like WebText would result in an unmanageably large vocabulary. To avoid this explosion in vocabulary size, the GPT-2 team applied BPE directly to raw bytes. Since there are only 256 possible byte values in text represented in Unicode format, this choice ensures

that the base vocabulary (the one from which BPE starts) is kept small in size (just these 256 possible byte values) and that at the same time it is able to represent any Unicode string without needing complex preprocessing steps. From this base of byte-level symbols, the model then constructed larger and more meaningful tokens by learning which byte sequences occurred most frequently and merging them. The final vocabulary built in this way results in **50,257 tokens** — a size small enough to remain manageable for large-scale training.

The authors introduced four versions of the GPT-2 model, each progressively larger in terms of size and parameter count (capacity). The smallest version used 12 transformer layers and a model dimension of 768 (the same as GPT-1), with 117 million parameters. The next size used 24 layers and 1024-dimensional embeddings, totaling 345 million parameters (equivalent in size to the largest model from BERT). A third configuration featured 36 layers with a width of 1280, yielding 762 million parameters. Finally, the largest model — officially referred to as **GPT-2** — employed 48 layers and a model dimension of 1600, reaching 1.5 billion parameters. This model had more than ten times the number of parameters of GPT-1.



Although the architecture followed the same fundamental design as GPT, a few modifications were introduced. Layer normalization was moved to the input of each sub-block, similar to a pre-activation residual network and an additional layer normalization was added after the final self-attention block. A modified initialization which accounts for the accumulation on the residual path with model depth is used. In particular, the weights of residual layers at initialization are scaled by a factor of $1/\sqrt{N}$ where N is the number of residual layers.

When tested over different language model tasks spacing from question-answering, summarization and even (machine) translation, these models reached important performance over the different benchmark test datasets even against other models that instead were trained with precise and quality labels.

This finding supports the central insight of the GPT-2 paper: high-capacity models trained to maximize the likelihood of a sufficiently varied text corpus begin to learn how to perform a surprising amount of tasks without the need for explicit supervision, i.e. without any fine-tuning, but just by being given the right prompt. This is even more empathized with in GPT-3 paper when it is given the name of **context-learning**. This was the first **key insight** that changed how the NLP community viewed model training — moving from fine-tuning to prompt-based learning.

GPT-3

Humans do not require large supervised datasets to learn most language tasks – a brief directive in natural language (e.g. “please tell me if this sentence describes something happy or something sad”) or at most a tiny number of demonstrations (e.g. “here are two examples of people acting brave; please give a third example of bravery”) is often sufficient to enable a human to perform a new task to at least a reasonable degree of competence.

One potential route towards addressing these issues is *meta-learning* – which in the context of language models means the model develops a broad set of skills and pattern recognition abilities at training time and then uses those abilities at inference time to rapidly adapt to or recognize the desired task. GPT-2 attempted to do this via what we call “***in-context learning***”, using the text input of a pretrained language model as a form of task specification: the model is conditioned on a natural language instruction and/or a few demonstrations of the task and is then expected to complete further instances of the task simply by predicting what comes next. Doing it the model started to show very important results. Moreover, since in-context learning relies on the model internalizing a broad range of patterns and skills, it was reasonable to expect that these abilities would also improve significantly with increased scale.

GPT-3 authors in their paper *Language Models are Few-shot Learners* (Brown et al., 2020) [53] tested these intuitions by training an unprecedentedly large autoregressive language model with 175 billion parameters, which they named **GPT-3**. To evaluate its capabilities, GPT-3 was tested on over two dozen natural language processing tasks under three conditions: ***few-shot learning***, where several examples are provided in the prompt; ***one-shot learning***, where only a single example is given; and ***zero-shot learning***, where only a natural language instruction is provided without any examples.

Note: The concepts of few-shot, one-shot and zero-shot learning were not introduced by the GPT-3 paper itself; these paradigms were already well-established and explored in many contexts like *meta-learning*, *metric learning* and *transfer learning*. GPT-3’s contribution was to show these abilities emerging in large-scale language models without explicit task-specific finetuning.

Few-shot, one-shot and zero-shot learning refer to some capabilities that often emerge in very large, well-trained models: the ability to handle a new downstream task — sometimes with surprising accuracy — when given only a few, one or even zero examples in the prompt, even if that task was never explicitly part of the training objective. However, this should not be mistaken for “learning with little or no data”. These capabilities arise precisely because the model **has already been trained on massive, diverse datasets**. Through pretraining, it acquires statistical patterns, semantic relationships and general characteristics of the underlying data distribution, which it can later leverage to generalize to new tasks. Personally, I don’t like the use of the term “*learning*” in this context because during inference the model is not actually updating its parameters or acquiring new knowledge — nothing is being “learned”. What we observe really is an emergent capability: the model adapting its outputs based on the patterns present in the prompt, thanks to the knowledge it has already internalized during training. Without sufficiently large and varied training data, the model could not build the rich representations necessary to make these generalizations, and such performance on unseen tasks would not be possible.

Moreover, also the dataset was expanded (“scaled”) resulting in a significantly **larger and more diverse dataset** (~570GB after filtering) obtained from a mix of filtered Common Crawl, WebText, Books1, Books2 and English Wikipedia.

GPT-3, relying solely on in-context learning, achieved state-of-the-art results on most of the NLP benchmarks considered, **without any gradient updates or fine-tuning**. Interestingly, the results showed that performance consistently improved with the addition of natural language task descriptions and with more examples included in the prompt used model’s context. In fact, authors described GPT-3 as a “*Few-shot learner*”.

Beyond task performance, GPT-3 demonstrated also important on-fly reasoning abilities such as unscrambling words, performing simple arithmetic and using newly introduced words correctly after only a single definition. In the few-shot setting, it could even generate synthetic news articles that human evaluators found difficult to distinguish from real ones. This highlighted GPT-3's capability to produce coherent and contextually appropriate text at a scale previously unseen.

A key insight from GPT-3 is that **carefully phrased prompts** can drastically affect performance. This laid the groundwork for the field of **prompt engineering**, which aims to design a good form for a prompt that results in high-quality output for a given downstream task.

In essence, GPT-3 solidified the notion of large language models as **meta-learners** — models that adapt to new tasks implicitly by leveraging their pre-trained knowledge, rather than through explicit retraining. Moreover, it demonstrated that scaling up language models — increasing their parameter count and training data following observed **scaling laws** — enables the emergence of surprisingly versatile and adaptable language understanding capabilities. The paper also acknowledged and began addressing ethical challenges, including biases, toxic language generation and factual inaccuracies inherent in such models. The authors raised concerns about **fairness, safety and potential misuse**, setting the stage for ongoing research into responsible AI deployment.

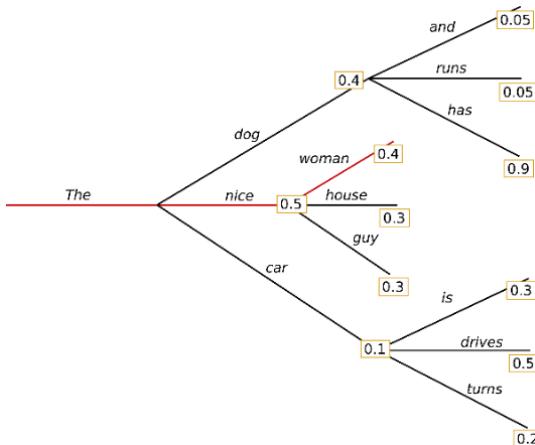
Strategies for Next Token Sampling

We have seen that the output of a causal Transformer is a probability distribution over all possible token values for the next token in the sequence, from which a particular value for that token must be chosen to extend the sequence. There are several options for selecting the value of the token based on the computed probabilities. One obvious approach, called **greedy search**, is simply to select the token with the highest probability. This has the effect of making the model deterministic, in that a given input sequence always generates the same output sequence. Note that simply choosing the highest probability token at each step is not the same as selecting the highest probability sequence of tokens overall. In fact, to find the most probable sequence, we would need to maximize the joint distribution across all the tokens in the sequence, which is given by:

$$p(y_1, \dots, y_N) = \prod_{i=1}^N p(y_i | y_1, \dots, y_{i-1})$$

If there are N steps in the sequence and the number of token values in the dictionary is K , then the total number of possible sequences is $O(K^N)$, which grows exponentially with the length of the sequence and hence finding the single most probable sequence is infeasible. By comparison, greedy search has a cost of $O(KN)$, which is linear in the sequence length.

Let's consider an example (here we suppose that tokens are words just for simplicity):



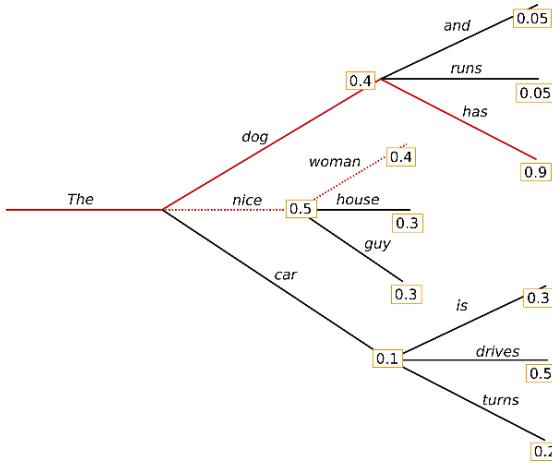
To alleviate this problem, one technique that has the potential to generate higher probability sequences than greedy search is the *beam search*.

Instead of choosing the single most probable token value at each step, **Beam search** [54] maintains a set of B partial sequences (called ***hypotheses***) at each step, where B is called the *beam width*.

In particular, at each time step the algorithm keeps track of the B most probable sequences so far. Then, for each of these sequences, it considers all possible next tokens and extends them, generating up to B^2 new hypotheses. At the end it is selected the most probable sequence between these hypotheses where the probability of a certain hypothesis sequence is obtained by multiplying the probabilities at each step of the sequence.

Since the probabilities are ≤ 1 then multiplying many of them tend to reduce the values, this can introduce a length bias: in fact, a long sequence will generally have a lower probability than a short one, biasing the results towards short sequences. For this reason, the sequence probabilities are generally normalized by the corresponding lengths of the sequence before making comparisons.

Therefore, Beam search reduces the risk of missing hidden high probability word sequences by keeping the most likely B of hypotheses at each time step and eventually choosing the hypothesis that has the overall highest probability. Let's illustrate this with $B = 2$:



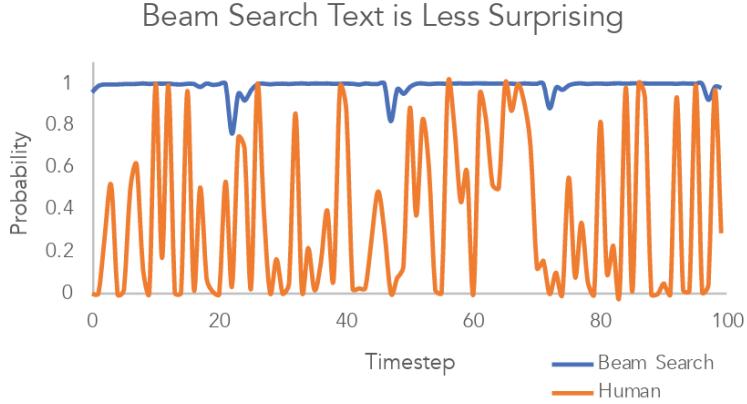
At time step 1, besides the most likely hypothesis (*<The>*, *<nice>*), beam search also keeps track of the second most likely one (*<The>*, *<dog>*). At time step 2, beam search finds that the token sequence (*<The>*, *<dog>*, *<has>*), which has a probability of 0.36 is higher than (*<The>*, *<nice>*, *<woman>*), which has a probability of 0.2. Thus, beam search was able to find the most likely token sequence in our example!

Beam search has cost $O(BKN)$, which is again linear in the sequence length. However, the cost of generating a sequence is increased by a factor of B , and so for very large language models, where the cost of inference can become significant, this makes beam search much less attractive.

One problem with approaches such as greedy search and beam search is that they limit the diversity of potential outputs and can even cause the generation process to become stuck in a loop, where the same sub-sequence of words is repeated over and over. For instance, referring the example above the output could generate a sentence that contains repetitions of same tokens multiple times, i.e. (*<The>*, *<nice>*, *<woman>*, *<The>*, *<nice>*, *<woman>*, *<The>*, *<nice>*, *<EOS>*).

Beam search is better suited for tasks with a predictable output length as in machine translation or summarization. Indeed, the original Transformer used beam search for machine translation. However, in open-ended generation where the desired output length can vary greatly, e.g. dialog or story generation, these methods often fail to produce varied or creative responses and are prone to repetitive patterns.

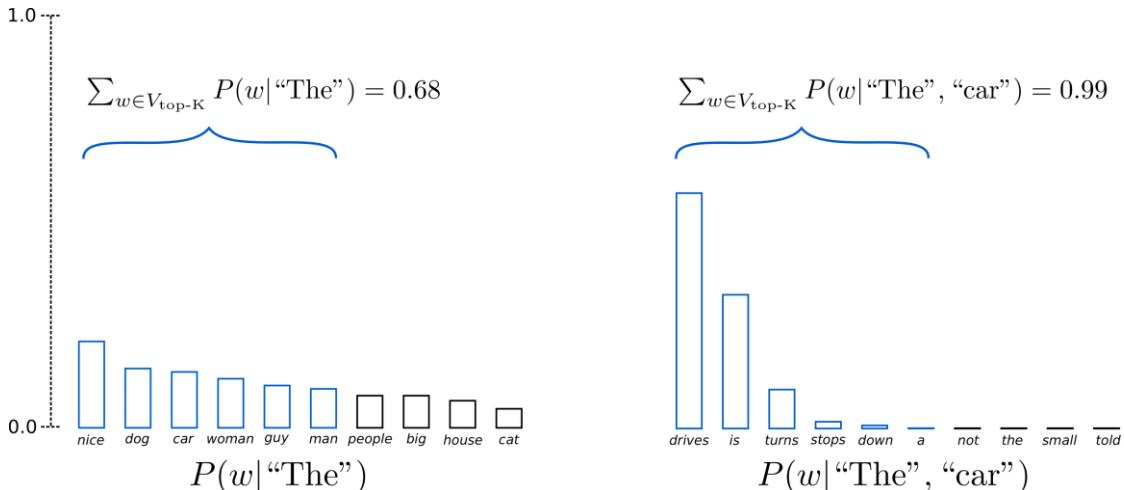
Moreover, as argued in Ari Holtzman et al. (2019) [55], high-quality human language does not follow a distribution of high probability next words, but instead it tends to involve lower probability and hence be more surprising with respect to a given model that automatically generate text. In essence, human readers value text that surprises them; predictability tends to diminish the perceived quality of the output. The authors show this nicely by plotting the probability that a model would give to human text vs. what beam search does. From the Figure we can see that human-generated sequences are assigned lower probabilities by language models than those produced by beam search, suggesting that maximizing likelihood does not necessarily equate to human-like generation.



To solve this, instead of trying to find a sequence with the highest probability, we can think to generate successive tokens simply by randomly sampling from the softmax distribution at each step. The first thing we should notice is that language generation using sampling is not *deterministic* anymore. Moreover, this naive randomly picking sampling approach leads to sequences that are likely incoherent or nonsensical. This is because token vocabularies are typically very large in size and therefore it is created a long tail of many token states each of which has a very small probability but which in aggregate account for a significant fraction of the total probability mass. This leads to the problem in which there is a significant chance that the system will make a bad choice for the next token, resulting in poor tokens which degrade the output quality.

To address this, an idea could be to sample not from all the distribution, but to restrict the sampling pool to only the top K tokens with the highest probabilities, for some choice of K , and then sample from these according to their renormalized probabilities (i.e. in this way the probability mass is redistributed among only those K tokens). This sampling scheme, introduced by Fan et. al (2018) [56], is known as **Top-K sampling**. For instance, GPT2 adopted *Top-K sampling* scheme, which was one of the reasons for its success in story generation.

To better illustrate how Top-K sampling works, let's revisit and slightly modify the example seen above extending the range of tokens used for both sampling steps from 3 to 10 tokens.



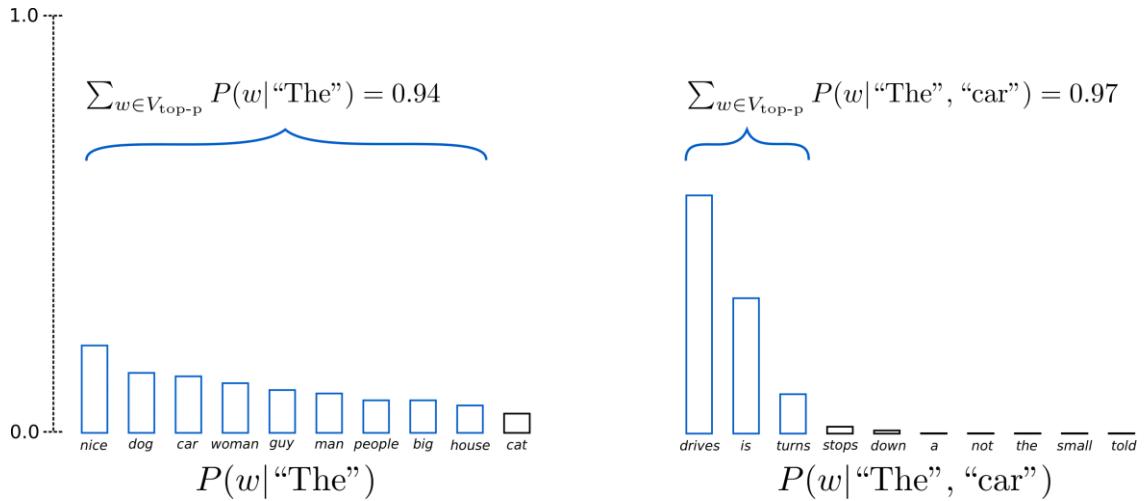
Assume we set $K = 6$ in both sampling steps which means we limit our sampling pool to 6 tokens.

We can notice that while the 6 most likely tokens encompass only two-thirds of the whole probability mass in the first step, it includes almost all of the probability mass in the second step. Nevertheless, we see that it successfully filters out the rather weird candidates (<not>, <the>, <small>, <told>) in the second sampling step.

However, this fixed-size cutoff introduces certain issues. In the step $t = 1$, limiting the sampling pool to the top 6 tokens eliminates the possibility to sample (<people>, <big>, <house>, <cat>), which seem like reasonable candidates. On the other hand, in step $t = 2$ the method includes the arguably ill-fitted tokens (<down>, <a>) in the sample pool of tokens. Thus, limiting the sample pool to a fixed size K could endanger the model to produce gibberish for sharp distributions and limit the model's creativity for flat distribution. This intuition led Ari Holtzman et al. (2019) [55] to create **Top- p sampling**, also known as **nucleus sampling**.

Instead of selecting a fixed number of top tokens, Top- p sampling chooses the smallest possible set of tokens whose cumulative probability exceeds a threshold p . The probability mass is then redistributed among this set of tokens and the model sample from it. This approach allows the number of candidate tokens to dynamically grow or shrink depending on the certainty of the model's predictions. When the distribution is uncertain (flat), the sampling set expands to include more options; when the model is confident (sharp), it narrows the set accordingly.

Let's see how this impacts the above example.



Assume we set $p = 0.92$ as threshold (which means that *Top- p* sampling picks the *minimum* number of tokens able to exceed $p = 92\%$ of the probability mass). In the first sampling step, it needs to consider the 9 most likely tokens to exceed the threshold, while in the second sampling step it only has to pick the top 3 tokens. It keeps a wide range of tokens where the next word is arguably less predictable, e.g. $P(w|<\text{The}>)$, and only a few tokens when the next token seems more predictable, e.g. $P(w|<\text{The}>, <\text{car}>)$.

While in theory, *Top- p* seems more elegant than *Top- K* , both methods work well in practice. *Top- p* can also be used in combination with *Top- K* , which can avoid very low ranked to never be chosen. For instance, GPT-3 uses both in combination.

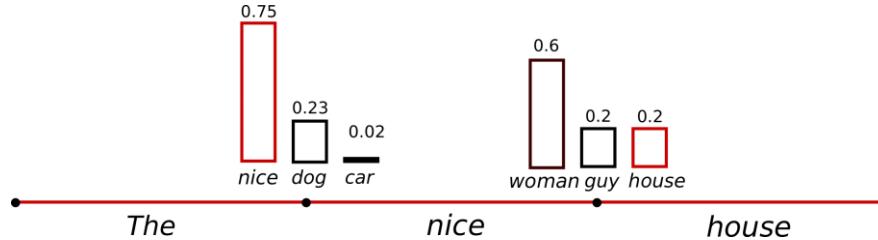
Finally, another method to control generation diversity is **temperature scaling**. This method introduces a scalar parameter T called **temperature** into the definition of the softmax function so that:

$$y_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

and then sample the next token from this modified distribution.

Note: In the formula z_i denotes the logits (unnormalized scores).

At $T = 1$, the distribution is unaltered, i.e. we keep our softmax distribution as it is. Lowering the temperature ($T < 1$) **sharpens** the distribution, making high-probability tokens more likely and low-probability ones less likely. In particular, at $T = 0$, the distribution collapses, resulting in a distribution that is concentrated on the most probable token, with all other token having zero probability, and hence this becomes equal to a greedy search selection. On the other hand, as $T \rightarrow \infty$, the distribution becomes **uniform** across all tokens (much smoother), maximizing randomness. An illustration of applying temperature of $T < 1$ to our first example could look as follows:



The conditional next word distribution of step $t = 1$ becomes much sharper (increasing the likelihood of high probability words and decreasing the likelihood of low probability words) leaving almost no chance for word <car> to be selected.

In practice, temperature scaling is used very often in combination with Top-K or Top-p sampling to fine-tune the balance between diversity and coherence.

Note: All examples referenced in this section are adapted from HuggingFace's blog post "[How to Generate Text](#)", which also provides implementation guidance in code.

ChatGPT

We previously saw how GPT-3 demonstrated impressive generative capabilities through self-supervised pretraining alone, without the need for any task-specific fine-tuning. This capability, learned purely from vast amounts of unlabeled text, allowed it to perform a surprising range of language tasks, from translation to question answering, simply by being prompted in the right way. However, despite its power, GPT-3 was not optimized to follow instructions or carry out sustained conversations in a natural, human-like way.

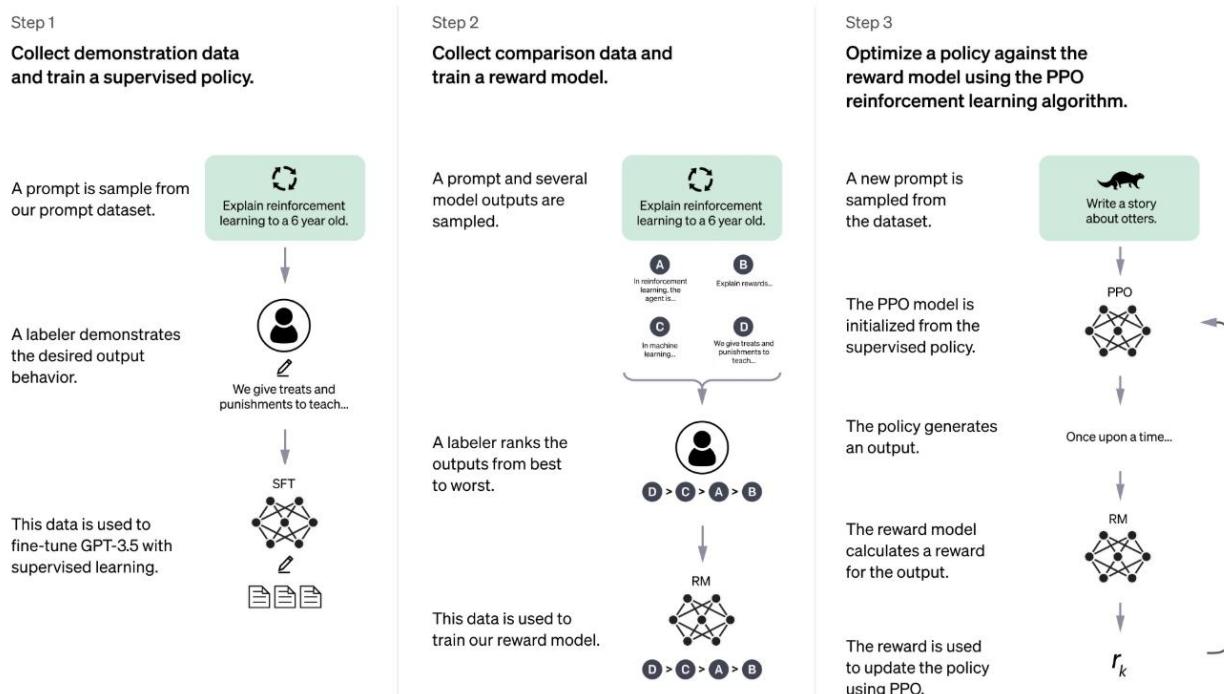


ChatGPT, introduced by OpenAI in late 2022, builds directly on these foundations. It is a fine-tuned and aligned version of the GPT-3.5 model ([soon we will understand what we mean with “aligned”](#)), specifically trained to respond to user prompts in a conversational and context-aware manner. All of us know today what ChatGPT is: a widely used **AI chatbot** capable of holding coherent multi-turn conversations across countless topics. For example, a typical chat looks like the one shown in the Figure above. However, beyond this familiar interface, it is also worth understanding — at least at a high level — how ChatGPT works under the hood. While going into full technical detail is outside the scope of this course, I’ll outline the core ideas.

ChatGPT is a sibling model of **InstructGPT** (Ouyang et al., 2022) [57], which was one of the first large-scale attempts to fine-tune and align a language model like GPT-3 to follow explicit **user instructions**. Both models rely on a key technique known as **Reinforcement Learning from Human Feedback (RLHF)**, which was the real breakthrough in **aligning language models with human expectations**.

Reinforcement learning is an AI subject in which an agent learns to make better decisions by receiving feedback in the form of rewards or penalties, depending on how good its actions are. In the context of conversational models, this idea is adapted to improve the quality of responses: human annotators act as a kind of “**reward signal**” by ranking different model outputs from best to worst. These preferences are then used to train a **reward model** that estimates how useful or appropriate a response is and this model guides the language model during alignment.

ChatGPT follows the same overall methodology as InstructGPT, but with slight differences in how its training data was collected. The process began with a first **supervised fine-tuning phase**: human AI trainers created example conversations by playing both roles — the user and the assistant — sometimes using model-generated suggestions to help formulate their replies. This newly created dialogue data was then combined with the InstructGPT dataset transformed into a dialogue format. Then it follows an **alignment phase** ([which we can see as a special fine-tuning](#)) done via *reinforcement learning* to align the model with human expectations: to train the reward model, OpenAI collected comparison data, where human trainers ranked multiple responses to the same prompt based on quality. These ranked samples came from real conversations with the model. Given these rankings, a reward model was trained to estimate which completions were better. Finally, this reward model was used to align the assistant using *Proximal Policy Optimization (PPO)*, a reinforcement learning algorithm. Several rounds of this process were carried out, gradually aligning the model’s behavior with human preferences.



Since ChatGPT was released **without an official research paper**, those interested in understanding how it works may find it helpful to read the InstructGPT paper, as ChatGPT builds directly on the same methodology. Moreover,

for a more informal but insightful perspective, I also recommend this blog article: [Understanding the Evolution of ChatGPT – Part 3](#).

Large Language Models

GPT-3 was one of the first examples of what we now call **Large Language Models (LLMs)** — very large transformer-based neural networks designed for natural language processing. The term "large" refers to the model's **number of parameters**, which can reach **hundreds of billions or even trillions**.

Training such models is computationally expensive, but their extraordinary capabilities justify the cost. In fact, we saw during our analysis of the GPT series that the impressive increase in performance of the GPT models through successive generations has come primarily from an increase in scale and an expansion of the datasets considered. They showed that increasing the size of the training data set, along with a commensurate increase in the number of model parameters, leads to improvements in performance that outpace architectural improvements or other ways to incorporate more domain knowledge. Their rise has been driven not only by the availability of massive datasets but also by significant advances in **massively parallel training hardware** — most notably GPUs and similar accelerators — organized in large clusters with high-speed interconnects and substantial onboard memory.

More importantly, a key factor for the rapid development of such LLMs is their ability to be **pretrained using self-supervised learning**. This means they learn from raw text without requiring manually labeled data, effectively allowing them to "label" their own inputs. This dramatically expands the volume of usable training data and justifies the use of very deep networks with a huge number of parameters.

A pre-trained model with broad capabilities learned during its pre-training is often referred also as a **foundation model**. These serve as **base models** for downstream **fine-tuning** on specific tasks using smaller, often labeled datasets. Many widely-used LLM applications today are based on such fine-tuned versions of openly released foundation models. Because fine-tuning requires significantly less data and computation than pretraining, it has become accessible even to organizations without massive computational resources.

We also saw that to further improve performance and alignment with user expectations, techniques like **Reinforcement Learning with Human Feedback (RLHF)** have been developed to fine-tune LLMs using **human feedback**. This led to development of **conversational chatbots** like **ChatGPT**, but then followed many others such as Google's [Gemini](#), Anthropic's [Claude](#), xAI's [Grok](#) and Mistral AI's [LeChat](#). These tools allow users to interact with AI using a natural language dialogue, making them very accessible to broad audiences. Because these models are explicitly trained to follow instructions and respond helpfully to user prompts, they are often referred to as **Instructed LLMs**.

Current state-of-the-art models, such as Open AI's GPT-4o, Gemini 2.5 or Anthropic's Claude 3.5, as evolved in scale shown that LLMs can exhibit increasingly **emergent behaviors**. Despite being extremely powerful, such LLMs had one common characteristic: they were **closed source**. In fact, when LLMs first began to gain widespread recognition, many of the most powerful LLMs were only accessible via paid APIs (e.g., OpenAI's API) and the ability to research and develop such models was limited to selected individuals or labs. However, despite the initial emphasis on proprietary technology, the LLM research community gradually began to create open-source variants of popular language models such as the GPT series. Even if these early open-source LLMs were inferior to the best proprietary models, they laid the foundation for greater transparency in LLM research and catalyzed the development of many subsequent models. For example, nowadays we have some important **open-source alternatives** to proprietary models, such as Meta's [LLaMA](#), Mistral AI's [Mistral](#) and TII's [Falcon](#) series, which continue to improve rapidly, promising to further redefine the frontiers of AI and **democratize** access to these technologies.

Beyond this, in the last years there have been many innovations around LLMs, in the following I want you to report some of them.

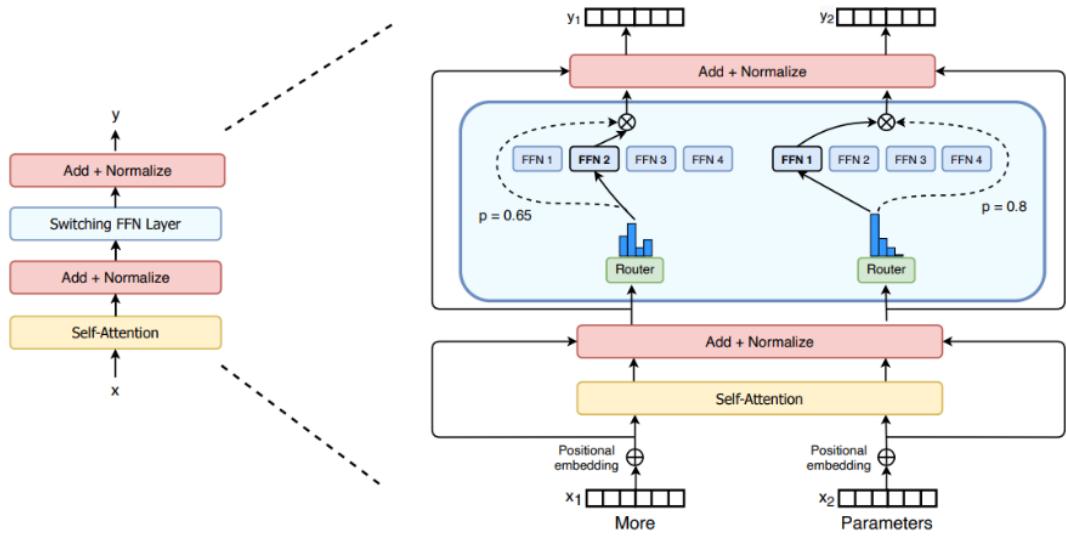
Mixture of Experts

As the size of the models increased over and over during the last years, researchers began exploring techniques to **reduce computational demands** without sacrificing performance. One of the most notable innovations is the **Mixture of Experts (MoE)** architecture [58].

MoE models enable large-scale neural networks to significantly cut down on computation during both training and inference by employing **conditional computation**: instead of activating the entire network for every input, only a small subset of specialized sub-networks, or “**experts**”, are activated as needed.

In the context of Transformer models, a Mixture of Experts architecture typically consists of two main components:

- **Sparse MoE layers:** These replace the standard dense FFN layers. A MoE layer contains **multiple experts** (e.g., 8), each of which is itself a neural network — often implemented as a FFN, though they can be more complex or even **hierarchical MoEs** (i.e., MoEs within MoEs).
- **A gating network (or router):** This learns to dynamically select which experts should process a given input token. For example, in the Figure below, the token “More” is sent to the second expert, and the token “Parameters” is sent to the first network. How to route a token to an expert is one of the big decisions when working with MoEs - the router is composed of learned parameters and is trained at the same time as the rest of the network.



A more recent example of an LLM utilizing the MoE architecture is **Mixtral 8x7B**. This model includes 8 expert networks, each containing 7 billion parameters. For each token processed at a given layer, the router activates only two out of the eight experts. Their outputs are then combined and passed to the next layer. Importantly, the selected experts may vary from layer to layer and from token to token, allowing for highly flexible and efficient computation.

The key advantage of MoEs lies in **sparsity**: only a fraction of the model's total parameters is used at any time, making it possible to scale up the model's total capacity without a proportional increase in computational cost. This means you can have a model with hundreds of billions of parameters but use only a small fraction of them per inference step — dramatically improving speed and efficiency.

However, this efficiency comes with a caveat: **all experts still need to be loaded into memory**. So, while MoEs save on compute during inference, they do not reduce the memory (RAM/VRAM) requirements of the model.

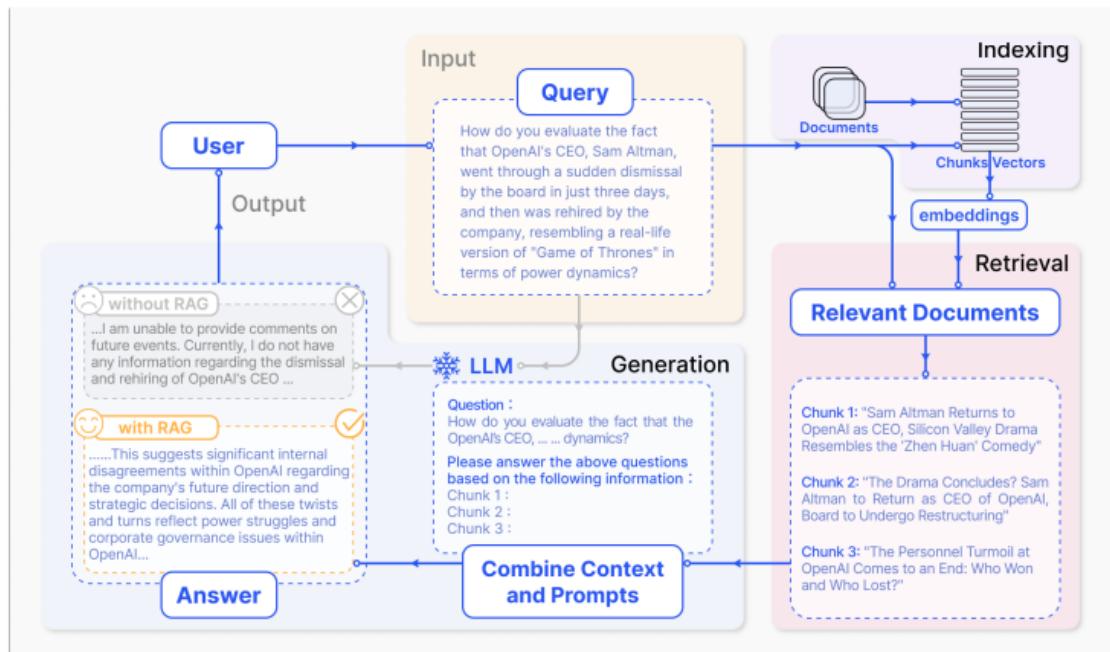
It's worth noting that **Mixture of Experts is not exclusive to LLMs** and the concept predates their rise. However, the scale and demands of modern LLMs have made MoEs particularly valuable, and they have become a cornerstone technique in the quest to make ever-larger models practical and efficient. More about them can also be found in this intuitive blog post "[A Visual Guide to Mixture of Experts](#)", which explains the core ideas and some new advancement.

Retrieval-Augmented Generation

As discussed earlier, large pre-trained language models have demonstrated the ability to learn and store a significant amount of factual knowledge within their parameters. However, their capacity to **access** and **manipulate** this knowledge with precision remains limited. On **knowledge-intensive tasks**, these models often underperform compared to more specialized architectures. They also lack the ability to **provide provenance** for their decisions and **updating their world knowledge** unless retrained or fine-tuned, which is expensive and inefficient. Another significant limitation is that, when lacking certain information, LLMs can produce **hallucinations** — plausible-sounding outputs that are **factually incorrect**. These outputs are difficult to verify and undermine the model's reliability.

To address these challenges, **Retrieval-Augmented Generation (RAG)** has emerged as a powerful framework. Introduced by Meta AI in 2020 with the paper "*Retrieval-Augmented Generation for Knowledge-Intensive Tasks*" [59], RAG provides a solution to mitigate some of these issues by augmenting LLMs with access to **external knowledge bases** at inference time. Rather than relying solely on static, parameterized model knowledge, RAG models **dynamically retrieve relevant documents** to ground their responses in real-world data.

A RAG system generally involves the following steps, as shown in the Figure below:



1. **Knowledge Base Construction:** Before generation begins, a curated **knowledge base** must be created. This can consist of internal documentation, web pages, academic papers or other domain-specific texts. The documents are divided into manageable “chunks” (typically 200–500 words) to optimize relevance during retrieval.

Each chunk is passed through a **text embedding model** (e.g. OpenAI's text-embedding-3-large or open-source models like Sentence-BERT), producing semantic vectors that capture the meaning of each chunk. These vectors, along with metadata and the original text, are stored in a **vector database** for efficient retrieval.

2. **Retrieval at Query Time:** When a user poses a query (i.e. the question which the LLM should respond to), the following steps occur:
 - **Step 1 - Query Embedding:** The user query is embedded into a vector using the same model used during knowledge base construction.
 - **Step 2 - Similarity Search:** A **semantic similarity search** is performed against the vector database (using cosine similarity or advanced vector search methods) to identify the top-K most relevant chunks.
 - **Step 3 - Context Construction:** The retrieved chunks (typically 3–5) are returned as context to support the generation task. These will be used to **augment the LLM's prompt**.
3. **Prompt Construction:** The retrieved documents are combined with the user's original query as additional context to build a comprehensive prompt for the LLM. A typical prompt structure might look like:

Question:

[User's question]

Context:

Chunk 1: [Relevant document snippet]

Chunk 2: [Relevant document snippet]

Chunk 3: [Relevant document snippet]

Answer:

This augmentation grounds the model's response in external information, thereby improving factual accuracy and transparency.

4. **Generation:** The constructed prompt is passed in input to the LLM. The model generates an answer that is **explicitly grounded in the retrieved documents** as the final output, reducing hallucinations and improving domain accuracy and trustworthiness.

RAG has gained significant traction in recent years due to its **flexibility** and **effectiveness**. It allows organizations to plug in their own data — such as internal wikis or customer service logs — without retraining the model. Moreover, RAG enables a sort of **explainability**, as retrieved sources can be shown alongside the generated answer, offering users visibility into the system's reasoning.

Beyond retrieving relevant information, the effectiveness of a RAG system hinges critically on **how the final prompt is constructed** before it's passed to the language model. The behavior of the model can be significantly influenced by **modifying the user's input prompt**, often by pre-pending a carefully crafted **prefix prompt** — a short instruction or context that guides the model toward the desired form, tone or constraints of the output. For example, this prefix might instruct the model to follow a particular structure, emphasize citation of sources, ignore irrelevant documents or synthesize conflicting sources. As such, **prompt engineering** plays a central role in unlocking the full potential of LLMs.

Chain of Thought

In RAG, we saw that concatenating multiple contextually relevant chunks into the prompt allows LLMs to produce more coherent and grounded answers. While this improves accuracy, it still focuses on *final answers* without explaining the reasoning behind them. In some cases, however, we want models to go beyond simply answering correctly — we want them to **explain the reasoning** behind their conclusions.

This is where **Chain of Thought (CoT) prompting** comes in. Introduced by Wei et al. in 2022 with the paper “*Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*” [60], CoT prompting has gained widespread attention in the past few years. The key idea is simple but powerful: instead of prompting the model to directly output an answer, we prompt it to “**think out loud**”, generating **intermediate reasoning steps** that lead to the final solution.

This technique mirrors how humans solve complex tasks — by breaking problems into smaller, logical steps. CoT prompting is especially useful for tasks involving multi-step reasoning such as math word problems, logic puzzles and commonsense reasoning. By providing **reasoned exemplars in the prompt**, the model learns how to structure its thought process.

Example: Without vs. With CoT

Without CoT Prompting:

Prompt:

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: The answer is 11

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

A:

Model output:

The answer is 27 (Incorrect)

With CoT Prompting:

Prompt:

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. $5 + 6 = 11$. The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

A:

Model output:

The cafeteria had 23 apples originally. They used 20 to make lunch. So, they had $23 - 20 = 3$. They bought 6 more apples, so they have $3 + 6 = 9$. The answer is 9. (Correct)

The model guided to walk through the reasoning, now took the right answer.

Over time, CoT prompting has evolved into multiple variants to suit different use cases. Some of them are:

- **Few-shot CoT prompting:** Show several examples with reasoning steps in the prompt before asking the model to solve a new problem.

Example:

Prompt:

Q: The odd numbers in this group add up to an even number: 16, 11, 14, 4, 8, 13, 24.

A: Adding all the odd numbers (11, 13) gives 24. The answer is True.

Q: The odd numbers in this group add up to an even number: 17, 9, 10, 12, 13, 4, 2.

A: Adding all the odd numbers (17, 9, 13) gives 39. The answer is False.

Q: The odd numbers in this group add up to an even number: 15, 32, 5, 13, 82, 7, 1.

A:

Model output:

Adding all the odd numbers (15, 5, 13, 7, 1) gives 41. The answer is False. (Correct)

- **Zero-shot CoT prompting:** Just add "Let's think step by step" to the original prompt — this surprisingly boosts reasoning even without examples.

Examples:

Without zero-shot CoT prompting:

Prompt:

Q: I went to the market and bought 10 apples. I gave 2 apples to the neighbor and 2 to the repairman. I then went and bought 5 more apples and ate 1. How many apples did I remain with?

A:

Output:

The answer is 11 apples (Incorrect)

With zero-shot CoT prompting:

Prompt:

Q: I went to the market and bought 10 apples. I gave 2 apples to the neighbor and 2 to the repairman. I then went and bought 5 more apples and ate 1. How many apples did I remain with? Let's think step by step.

A:

Output:

First, you started with 10 apples. You gave away 2 apples to the neighbor and 2 to the repairman, so you had 6 apples left. Then you bought 5 more apples, so now you had 11

apples. Finally, you ate 1 apple, so you would remain with 10 apples. The answer is 10 apples. (Correct)

Surprisingly, even this small prompt tweak enables significantly more accurate and thoughtful answers — especially valuable when you can't use few-shot examples.

If you're curious to explore further, this blog post offers a great introduction: [A Visual Guide to Reasoning LLMs](#). Despite the enthusiasm around these reasoning-capable LLMs, there is ongoing debate about their actual capacity to "think". Recent research has highlighted important criticisms. For example, several papers have shown that the reasoning steps generated as explanation of the answer by the model **does not always reflect its true internal reasoning process**. In other cases, models appear to produce accurate answers not because they reasoned effectively, but due to **data leakage** — that is, because the math problems or logic puzzles used in evaluation were **present in some form in the training data**, introducing **contamination** and making performance appear better than it truly is. This raises questions about whether the model is genuinely solving problems or simply recalling memorized patterns.

There is a growing body of work investigating these limitations. For a brief overview of some of the criticisms and challenges, I recommend watching this [video](#), which highlights several of these concerns.

Note: The video is in Italian, so for foreign students the easiest solution is to use YouTube's automatic subtitles: click the CC icon → Subtitles/CC → **Auto-translate** → select *English*.

To conclude, while LLMs are undeniably powerful and currently at the forefront of deep learning research and commercial interest, they represent only one facet of a much broader and richly diverse field. DL encompasses a wide array of architectures, techniques and applications, each offering unique insights and innovations. Today, LLMs dominate the spotlight, both in academia and industry, often overshadowing other promising areas. However, I encourage you, the reader, to **not be stuck with LLMs** and instead delve into the many other domains within DL. I firmly believe that some of the most significant breakthroughs — particularly in areas like explainability — will emerge not from a single architecture, but from the integration and collaboration of **multiple paradigms working together**.

Vision Transformer

Although the Transformer architecture was originally developed as a replacement for RNNs in natural language processing, its influence has rapidly expanded across nearly every domain of DL. What makes Transformers so powerful is their general-purpose design: they make few assumptions about the structure of the input data, unlike for example CNNs which rely on strong inductive biases such as translation equivariance and locality. This architectural flexibility has enabled Transformers to achieve state-of-the-art performance across a variety of modalities — including text, images, video and audio.

The core architecture of the Transformer has remained largely unchanged over time and across these domains. As a result, the key innovations in adapting Transformers to new data modalities lies not in altering the architecture itself, but in how the input and output data are represented: as long as we can tokenize the input and decode the output tokens appropriately, a Transformer can be used!

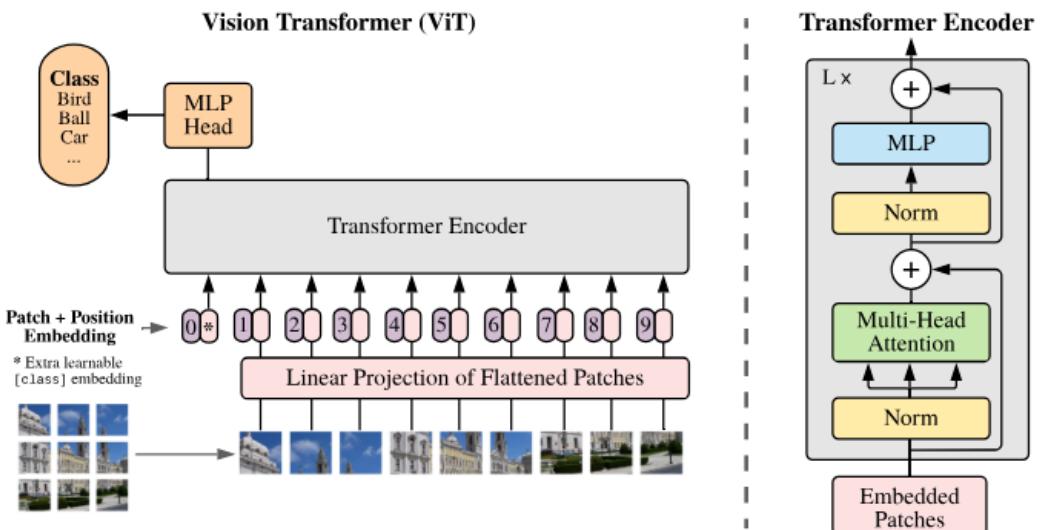
Here, as an example of this adaptability we will analyze the **Vision Transformer (ViT)**, introduced by Dosovitskiy et al. (2020) in the paper *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale* [61], which was the first Transformer model to be applied directly to images.

Since a Transformer expects to receive a sequence of tokens as input, applying a Transformer to images requires to decide how to convert an input image into tokens. A naïve choice would be to use each pixel as a token, followed by a linear projection. However, the memory required by a standard transformer implementation grows quadratically with the number of input tokens and so this approach is generally infeasible. Instead, the authors considered performing tokenization by splitting the image into a **grid of patches of the same size**.

Suppose the images have dimension $x \in \mathbb{R}^{H \times W \times C}$ where H and W are the height and width of the image in pixels and C is the number of channels (typically equal to 3 for RGB). Each image is split into non-overlapping patches of size $P \times P$ (where $P = 16$ is the common choice proposed by the paper) and then ‘flattened’ into a one-dimensional vector, which gives a representation $x_p \in \mathbb{R}^{N \times (P^2 C)}$, where $N = HW/P^2$ is the total number of patches for one image. Each patch is then linearly projected into an embedding, forming a sequence of patch embeddings suitable for Transformer input.

To encode positional information, ViT adds **learned positional embeddings** to each patch token. Unlike Transformers in NLP, which typically handle variable-length sequences, ViT operates on fixed-size images and thus always receives a fixed number of tokens. In NLP tasks, learned positional embeddings are often avoided because they struggle to generalize across varying sequence lengths. However, since ViT deals with inputs of fixed size, this limitation does not apply — making learned positional embeddings an effective and suitable choice.

The architecture of ViT is shown in Figure below. It closely resembles a standard **Transformer encoder**, with a few modifications: layer normalization is applied before the self-attention and feedforward sublayers (*pre-norm*) and the activation function in the feedforward is GELU instead of ReLU.



The primary goal of the ViT authors was to showcase the effectiveness of the architecture on image classification tasks. To enable classification, they introduced a learnable “classification token” — similar to the `<CLS>` token used in BERT — which is prepended to the sequence of patch embeddings. After passing through the Transformer encoder, the output corresponding to this token serves as a compact representation of the entire image. This representation is then fed into a final output layer, consisting of a linear projection followed by a softmax function, to predict the image’s class label.

It’s important to note that ViT is trained **exclusively to predict the class label** in a supervised manner, using a cross-entropy loss on labeled images. No masking of tokens (needed for masked token prediction in BERT) is adopted. The outputs of the other patch tokens **are not directly used** for classification; instead, their role is to provide contextual information and support the classification token through the attention mechanism.

Same as BERT, ViT features the “Base” (ViT-B) and “Large” (ViT-L) versions. Each of them can feature a different input patch size (for example ViT-L/16 means that the “Large” variant is adopted with an input patch size of 16×16).

ViT departs significantly from CNNs in terms of architectural bias. CNNs incorporate strong two-dimensional inductive biases that help them learn efficiently from relatively small datasets. In contrast, ViT has almost no built-in assumptions about spatial structure — aside from the use of fixed-size patches. However, at the same time this means it typically requires a much larger training dataset than a comparable CNN as it has to learn the geometrical properties of images from scratch. Interesting to note is that, because there are no strong assumptions about the structure of the inputs, transformers are often able to converge to a higher accuracy when trained at scale, as they can learn patterns directly from data without being limited by rigid architectural priors. This illustrates a fundamental trade-off in DL: models with fewer inductive biases tend to require more data, but offer greater flexibility and performance potential.

CLIP

The advantage of having a single architecture that is capable of processing many different kinds of data makes Transformer architecture extremely appealing for **multimodal learning** — where multiple data types (e.g., text, images, audio) are processed jointly. For example, this shared architecture underpins models capable of describing images in natural language and vice versa (as in CLIP) or even generating images from text prompts (as in DALL-E or Stable Diffusion). Moreover, because the same core architecture is used throughout, information from different modalities can be fused not only at the input or output level, but even within intermediate layers of the model, enabling richer and more integrated representations. To illustrate these ideas, we will analyze CLIP here. Stable Diffusion will be explored later on, as it requires additional concepts that will be introduced progressively in the coming chapters.

CLIP (Contrastive Language–Image Pretraining) is a **multimodal neural network** introduced by OpenAI in 2021 with the paper “*Learning Transferable Visual Models From Natural Language Supervision*”[62]. It learns to **connect images and natural language** by training on **pairs of images and their corresponding text captions**, enabling it to map both modalities into a **shared (joint) embedding space**. For example, an image of a dog and the sentence “*an image of a dog*” would end up close together in this space.



Unlike traditional supervised learning approaches — which rely on fixed label sets (e.g., the 1000 categories in ImageNet) — CLIP is trained using **natural language supervision** in a **self-supervised** fashion. This enables it to understand visual content through language and generalize to new tasks it wasn’t explicitly trained for.

To train CLIP, the authors constructed a dataset called **WIT (WebImageText)** consisting of 400 million (image, text caption) pairs collected from publicly available internet sources. The captions are often noisy, but this weak supervision is compensated by massive scale.

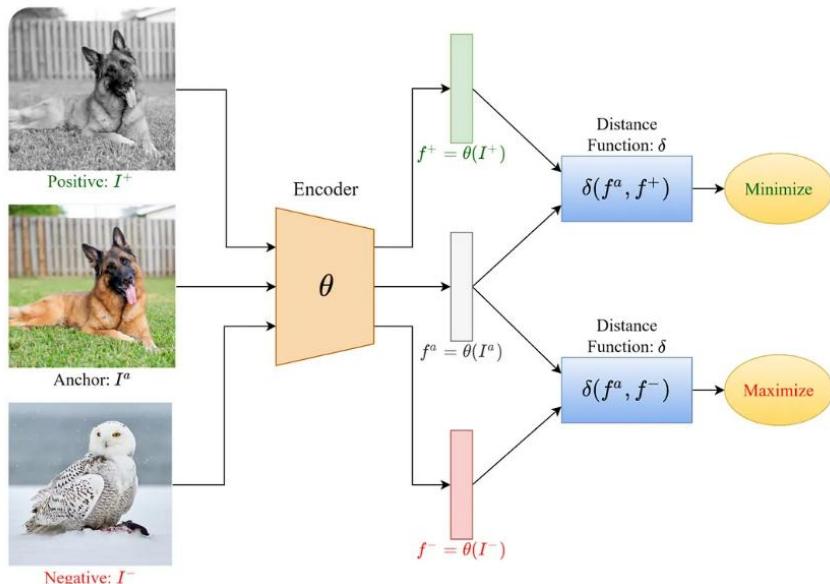
Learning from natural language has several potential strengths over traditional supervised approaches:

- **Scalability:** Natural language is abundant on the internet and far easier to collect than high-quality, crowd-sourced labels.
- **Language grounding and Zero-shot capabilities:** The model doesn't just learn visual representations, but it also learns how those visuals relate to **natural language**. This gives it **zero-shot capabilities**, meaning it can generalize to new tasks without being explicitly trained for them simply from exposure to diverse and abundant image–text pairs. For example, the authors demonstrated that CLIP achieves impressive zero-shot performance on ImageNet—despite never being explicitly trained on its 1.28 million labeled examples. Similar results were observed across various other benchmarks. Moreover, zero-shot evaluation of task-agnostic models is much more representative of a model's capability since it provides a more realistic and comprehensive measure of a model's generalization ability.

CLIP architecture consists of two main components, an **Image Encoder** – typically a **Vision Transformer (ViT)** or a ResNet – and a **Text Encoder** – an **encoder Transformer**. Both encoders are trained **from scratch** without initializing the image encoder with ImageNet weights or the text encoder with pre-trained weights. For all architectures minor modifications were made as described in the paper.

The authors initially tried to jointly train both the encoders to predict the caption of an image. However, they found it did not scale well to large datasets (such as WIT) and so they opted instead for a **contrastive representation learning** approach, which is more efficient for large-scale data. The goal of contrastive representation learning is to learn an embedding space in which similar sample pairs stay close to each other while dissimilar ones are far apart.

In a standard contrastive learning approach, you give the model an (**anchor, positive, negative**) **triplet**, where *anchor* is an image from one class, say a dog, *positive* is an alternative image from the same class, a dog, and *negative* is an image of another class, say a bird. You then embed the images, and you train the model in such a way that the distance between the two embeddings for the same class (the dog), $distance(anchor, positive)$, gets **minimized** and the distance between the two embeddings of different class, (the dog and bird), $distance(anchor, negative)$ gets **maximized**. This encourages the model to output very similar embeddings for the same objects and dissimilar embeddings for different objects.



The same approach can be applied to text as well as combination of text and images. For instance, for CLIP, for a single training example, the anchor could be an image of a dog, the positive could be the caption “*an image of a dog*” and the negative could be the caption “*an image of a bird*”.

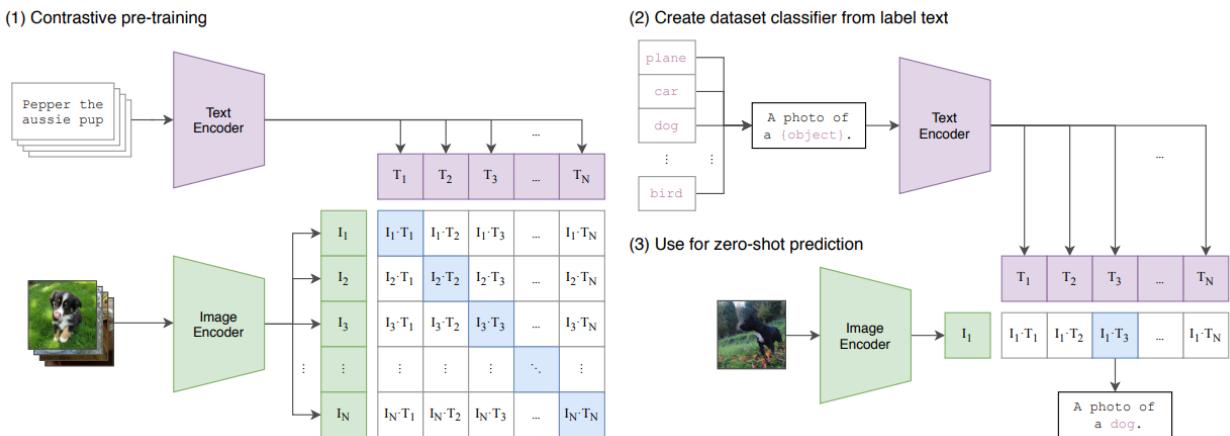
More specifically, CLIP uses a **multi-class N-pair loss**, which scales this contrastive idea across a full batch of examples (see Figure-1 below):

- Given a batch of N (**image, text**) pairs:
 - Each image is encoded to a vector I_i
 - Each caption is encoded to a vector T_j
- **Cosine similarity** is computed between **all** image–text pairs in the batch: $\text{sim}(I_i, T_j)$ for all i, j
- The model is trained to:
 - Assign high similarity to the correct matching pairs (I_i, T_i)
 - Assign low similarity to mismatched pairs (I_i, T_j) where $j \neq i$

This is done using a **symmetric cross-entropy loss** over image-to-text and text-to-image similarities.

This encourages CLIP to learn a **joint embedding space** where aligned image–text pairs are close and mismatches are distant.

Note: Unlike traditional contrastive learning setups that require triplets (anchor, positive, negative), CLIP learns from **implicit negatives** present in the batch.



Once trained, during inference CLIP can be, for example, used for **image classification without fine-tuning** (see Figure 2-3):

1. Provide a list of class labels and construct natural language prompts of the form “a photo of {class}”. E.g. “*a photo of a cat*”, “*a photo of a dog*”, “*a photo of a car*”, ...
2. Encode these prompts using the **text encoder**
3. Encode a new test image using the **image encoder**
4. Compare the image’s embedding to each class prompt using **cosine similarity**
5. The class whose text prompt is most similar to the image (highest cosine similarity) is the prediction

Interestingly, this is an example of **zero-shot classification**, meaning the model can correctly assign labels to images for classes it has never been explicitly trained to recognize, simply by leveraging the general concepts it learned during training.

CLIP is important because it represents a major step toward building general-purpose models that can “understand and reason” across different modalities. Unlike traditional vision systems that require task-specific fine-tuning, CLIP demonstrates strong generalization: it **performs well across a wide variety of datasets and tasks without needing to be retrained for each one**. What makes this possible is its ability to connect vision and language in a shared representation space, which allows it to interpret images in terms of natural language — a far more flexible and expressive supervision signal than fixed class labels.

This **multimodal understanding** gives CLIP a kind of real-world flexibility: it can recognize and describe objects or concepts it has never explicitly been trained on, simply by matching visual content to textual descriptions.

Recent Advances in Transformers

We conclude this chapter by exploring several architectural innovations that have recently pushed the efficiency and scalability of Transformer architectures. In particular, we will look at techniques designed to reduce memory usage, accelerate inference and improve handling of long sequences.

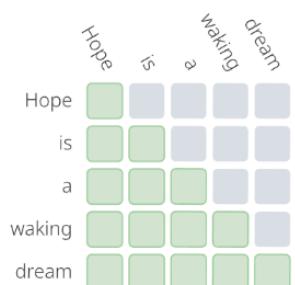
KV Cache, Multi-Query and Grouped-Query Attention

In autoregressive transformer models (e.g., GPT), text is generated one token at a time. At each decoding step, the model attends over all previously generated tokens. However, this process involves significant redundant computation, particularly in the attention mechanism. The **KV Cache**, along with **Multi-Query Attention (MQA)** and **Grouped-Query Attention (GQA)**, are optimizations that address this inefficiency. The following section draws from this detailed [blog post](#), which walks through each step with code and examples. Here, we will focus on presenting the core ideas directly.

KV Cache

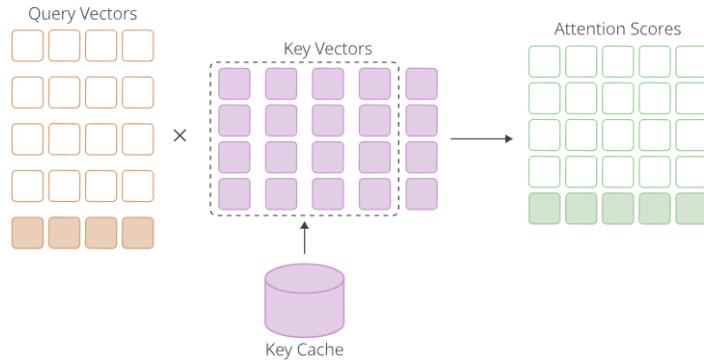
In standard multi-head self-attention, each attention head computes the *attention scores* between the current token’s query vector and all keys generated so far. This means that for each new token, the model must compute attention over all previously generated tokens.

However, there’s an important observation: At each decoding step, only the new query vector and the key/value vectors for the latest token are new — all prior keys and values remain unchanged.

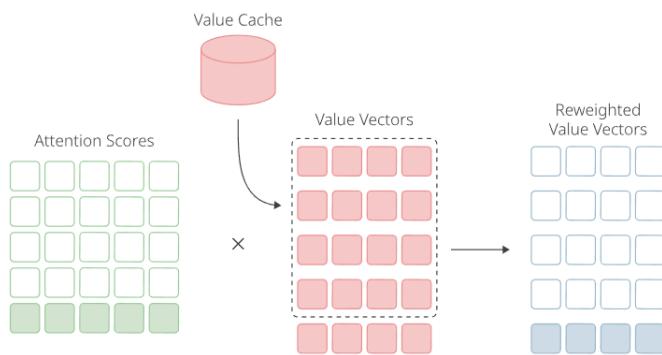


This means we don’t need to recompute previous keys and values at each step. We can **cache** them:

- **Key Cache:** Since attention scores are dot products between queries and keys and only the most recent query is new, we can reuse previously computed key vectors and just compute the new one for the latest token.



- **Value Cache:** Similarly, the value vectors used in attention can be cached. Only the latest value vector needs to be computed at each step.



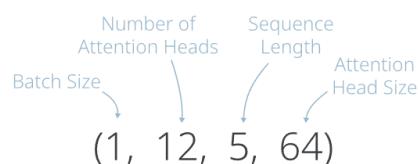
Together, the **Key Cache** and **Value Cache** form the **KV Cache** [63] — a mechanism to store and reuse key and value vectors for each layer during inference — which dramatically reduces redundant computation. It's so effective that it's **enabled by default** in [transformers](#) library. The KV Cache can be turned off by setting `use_cache=False` when loading the model.

Notes:

- The above discussion refers **just to autoregressive Transformers**, given that for models like BERT the attention is computed bidirectionally just one time.
- KV Cache is used **just during inference** phase since in training we still need to process the entire input in parallel.

Each transformer layer maintains **its own KV Cache**, typically shaped as:

$$[batch\ size, num\ heads, sequence\ length, head\ dim]$$



So, the **total memory footprint** of the KV Cache is:

$$Memory = 2 \times l \times b \times n \times s \times h \times 2$$

where:

- l : number of layers
- b : batch size
- n : number of attention heads
- h : head dimension
- s : sequence length
- First $\times 2$: number of caches per layer (i.e. one cache for **keys**, one for **values**)
- Second $\times 2$: number of bytes per floating point value assuming each model parameter is a **16-bit floating point value** (i.e., 2 bytes per float)

For example, the table below shows the size of the KV Cache for various versions of GPT-3.

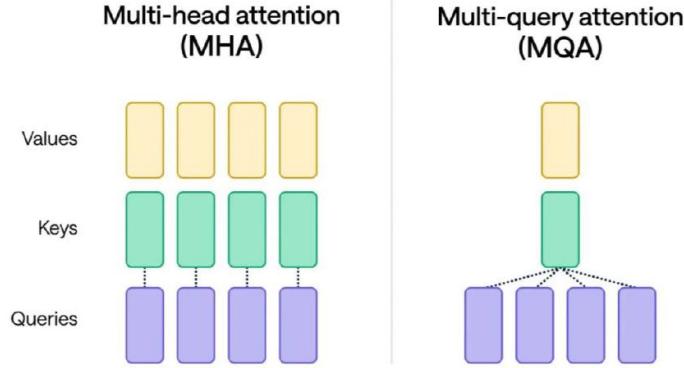
Batch Size:
1
Sequence Length:
1024

Model	Parameter Count	KV Cache Size
GPT-3 Small	125M	36.000 MB
GPT-3 Medium	350M	96.000 MB
GPT-3 Large	760M	144.000 MB
GPT-3 XL	1.3B	288.000 MB
GPT-3 2.7B	2.7B	320.000 MB
GPT-3 6.7B	6.7B	512.000 MB
GPT-3 13B	13B	800.000 MB
GPT-3	175B	4.500 GB

As model sizes get larger and sequence lengths get longer, the amount of memory consumed by the KV Cache becomes a major contributor to memory usage. **Multi-Query Attention (MQA)** and **Grouped-Query Attention (GQA)** are two techniques that combat this problem.

Multi-Query Attention

In standard *Multi-Head Attention*, each attention head computes its own independent set of query, key and value vectors. In contrast, **Multi-Query Attention (MQA)** [64] simplifies this by allowing each head to have **its own query vectors**, while **sharing the same key and value vectors** across all heads.



This simple change brings a substantial benefit during autoregressive inference. Since all heads reuse the same key and value vectors, the memory required to store the **KV Cache** can be reduced by a factor of n , where n is the number of attention heads. Specifically, the size of the KV Cache when using MQA becomes:

$$Memory = 2 \times l \times b \times s \times h \times 2$$

Let's look again at the size of the KV Cache for various versions of GPT-3, this time with a new column showing the size of the KV Cache when using MQA:

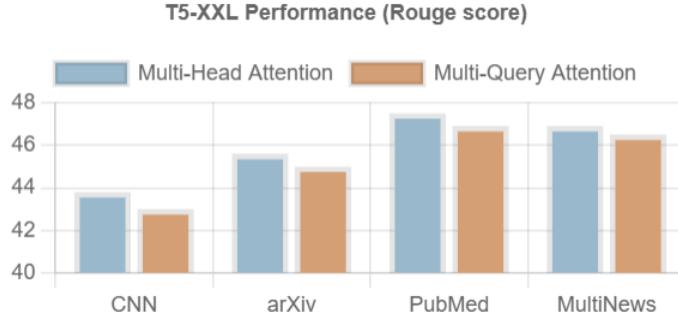
		Batch Size:	Sequence Length:	
		1	1024	
Model	Parameter Count	KV Cache Size (MHA)		KV Cache Size (MQA)
GPT-3 Small	125M	36.000 MB		3.000 MB
GPT-3 Medium	350M	96.000 MB		6.000 MB
GPT-3 Large	760M	144.000 MB		9.000 MB
GPT-3 XL	1.3B	288.000 MB		12.000 MB
GPT-3 2.7B	2.7B	320.000 MB		10.000 MB
GPT-3 6.7B	6.7B	512.000 MB		16.000 MB
GPT-3 13B	13B	800.000 MB		20.000 MB
GPT-3	175B	4.500 GB		48.000 MB

MQA offers a major reduction in KV Cache size and was even adopted in the PaLM model from Google Research [65]. This technique, however, has since been surpassed in popularity by the similar Grouped-Query Attention technique, primarily for two reasons:

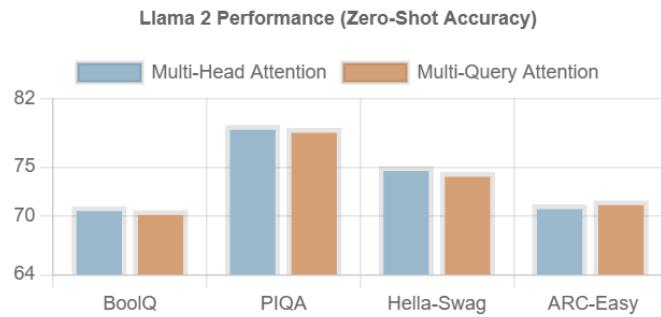
1. Model Performance Degradation:

Sharing key and value vectors reduces the representational capacity of each attention head. Empirical studies have shown that this can lead to **decreased model performance**, even when total parameter count is held constant.

- In their study, Ainslie et al. [66] demonstrated that using MQA in T5-XXL [67] led to lower scores on several summarization benchmarks compared to standard MHA.



- Similarly, in experiments with their model LLaMA 2, Touvron et al. [68] tested a version of the model using MQA with a larger feed-forward network to compensate for the parameter reduction. Despite this, the MQA variant consistently underperformed standard MHA across multiple benchmarks, reporting lower accuracy on average.



These results suggest that while MQA is attractive from a memory and speed perspective, it may not match the expressiveness of traditional MHA, especially in high-capacity models.

2. Inefficient Parallelization

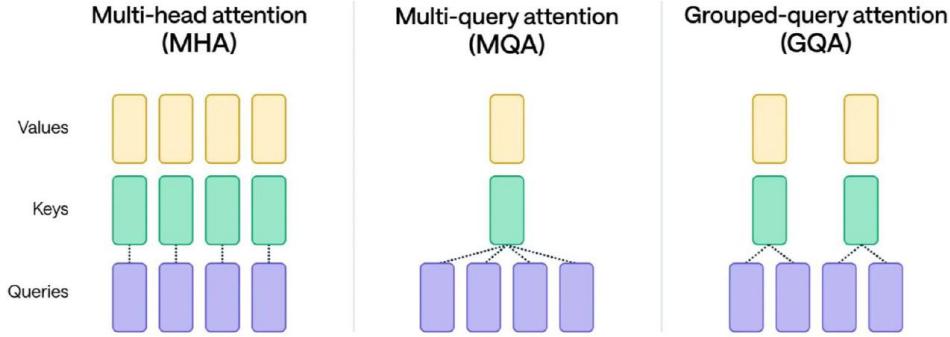
Large-scale transformer models often require **model parallelism** to fit into GPU memory. One common strategy is **Tensor Parallelism** [67], where attention heads are split across GPUs.

- For example, a model with 96 attention heads (like GPT-3 175B) could be distributed across 8 GPUs, with 12 heads per GPU.
- In standard MHA, each GPU independently computes attention for its assigned heads and only exchanges outputs. But with MQA, since **all heads share the same key and value vectors**, those vectors must be **replicated** (cached) **across all GPUs**.

This leads to **redundant computation and storage** on each device, negating some of the parallel efficiency and memory savings that MQA aims to provide.

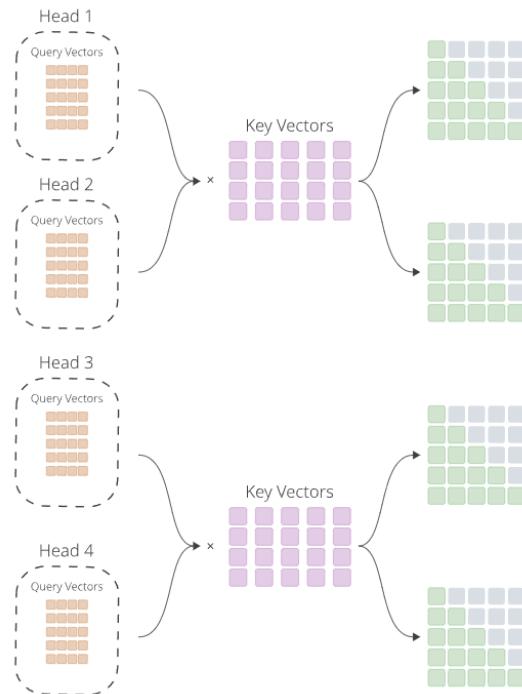
Grouped-Query Attention

Grouped-Query Attention (GQA) [66] is essentially a generalized definition that encompasses both standard MHA and MQA. It introduces a flexible middle ground between the two by allowing the number of unique Key and Value vectors to be defined as a configurable hyperparameter.

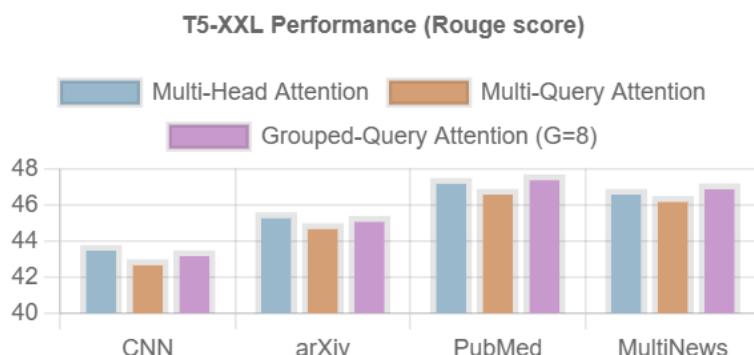


In MHA, the number of unique Key and Value vectors is equal to the number of attention heads; in MQA, the number of unique Key and Value vectors is equal to 1.

In GQA, the number of unique Key and Value vectors is equal to a **hyperparameter G** , the number of **Groups**. For example, if the number of attention heads is 4 and $G = 2$, then there will be two unique sets of Key and Value vectors, each of which will be used by two attention heads:



Using GQA with a group size of 8, the authors measured the performance of T5-XXL on the same summarization tasks as above, comparing the results to both MHA and MQA:



Results showed that GQA offers a **slight improvement in performance over MQA**, while still offering a **significant reduction in KV Cache size**. In fact, when operating in a multi-GPU environment with tensor parallelism, GQA can essentially get these performance gains for free by **setting G equal to the number of GPUs**.

The Table below, again, shows the size of the KV Cache for the variants of GPT-3, this time with GQA included:

Batch Size:	Sequence Length:	Number of Groups:		
1	1024	8		
Model	Parameter Count	KV Cache Size (MHA)	KV Cache Size (MQA)	KV Cache Size (GQA)
GPT-3 Small	125M	36.000 MB	3.000 MB	24.000 MB
GPT-3 Medium	350M	96.000 MB	6.000 MB	48.000 MB
GPT-3 Large	760M	144.000 MB	9.000 MB	72.000 MB
GPT-3 XL	1.3B	288.000 MB	12.000 MB	96.000 MB
GPT-3 2.7B	2.7B	320.000 MB	10.000 MB	80.000 MB
GPT-3 6.7B	6.7B	512.000 MB	16.000 MB	128.000 MB
GPT-3 13B	13B	800.000 MB	20.000 MB	160.000 MB
GPT-3	175B	4.500 GB	48.000 MB	384.000 MB

Since its introduction in October 2023, GQA has been adopted in several popular open-source transformer models:

- The Falcon series of models [69] uses GQA with 1 group (i.e. MQA) for their 7B parameter model and 8 groups for their 40B and 175B parameter models.
- The Llama 2 models [68] employ GQA with 1 group in the 7B and 13B variants and 8 groups in the 70B variant.
- The Mistral 7B model [70] and Mixtral 8x7B13 [71] each use GQA with 8 groups.

Advancing Positional Embeddings

One of the key components of Transformers are positional embeddings. But why are they needed at all? The fact is that the self-attention mechanism by itself is **permutation-invariant**. This means that given only the token embeddings, self-attention does not inherently understand the *order* of tokens in a sequence. Without additional information, it treats the sequence as an unordered “**bag of tokens**”. However, as we have discussed extensively, in language and many other sequential data domains, **order matters** — “the cat chased the dog” is very different from “the dog chased the cat”. **Positional embeddings** solve this problem by injecting information about token positions directly into the model’s representations, so that then attention layers can take token order into account. The following section is based on the explanations and insights from this [blog post](#).

Absolute Positional Encoding (Recap)

Previously, we covered two types of **absolute positional encoding**:

- **Sinusoidal (Fixed):** This method uses deterministic patterns of sine and cosine waves at various frequencies to generate a unique positional vector for each position in a sequence. These are **not learned** during training.
- **Learned (Trainable):** With this approach, each position in the sequence is assigned a unique, trainable vector. These vectors are optimized alongside the model's other parameters during the training process.

Both methods are considered "**absolute**" because they assign a unique, fixed vector to each specific position in the input sequence. This positional vector is then combined with the token's word embedding, typically through an element-wise sum, before the resulting embedding is fed into the attention mechanism.

Absolute positional encodings were introduced in the original Transformer paper and were widely used in popular models such as BERT, GPT and RoBERTa [72].

Limitations of absolute positional embeddings:

- **Fixed maximum length:** Learned embeddings (just them, not the sinusoidal ones) can only represent positions up to the length they were trained for. Beyond that, they cannot generalize.
- **Lack of relative information:** Each positional embedding is independent of others. The model treats position 1 vs. 2 the same way as position 2 vs. 500 — it has no notion of “closeness” or “farther apart”. This lack of relative positioning can hinder the model’s ability to reason about structure, especially in long sequences.

Relative Positional Embedding

Relative positional embeddings [73] focus on the distance between pair of tokens rather than their absolute position in a sequence. This approach understands that a token's relationship to another is defined by how far apart they are, not their specific index in the sentence.

For example, consider the sentence “I am a student”. Relative positional embeddings would note that “I” is three positions away from “student” and one position from “am”, completely ignoring their absolute positions of 1 and 4.

This method has a significant advantage: it generalizes better to sequences longer than those seen during training. This makes it particularly effective for models designed to handle a wide range of sequence lengths.

Some notable models that use relative positional embeddings include Transformer-XL, T5 and DeBERTa.

How It Works

Instead of adding a positional embedding to the token embedding itself, relative positional embeddings modify the **attention scores** within the self-attention mechanism. They introduce a **bias term** into the attention score calculation that incorporates the relative distance between tokens.

Recall that the attention score between two tokens, say token i and token j , is calculated using their query and key vectors. Relative positional embeddings add a bias term B_{ij} to this score based on their distance $(j - i)$. A common implementation (the one used in T5’s code) looks like this:

$$A_{ij} = \frac{Q_i K_j^T}{\sqrt{d_k}} + B_{ij}$$

Case Study – The T5 Model: The T5 model provides a good example of relative positional embeddings. It uses a **learnable bias**, represented as a floating-point number, **for each possible relative position offset between two tokens**. For example, a bias B_1 might represent the relative distance between any two tokens that are exactly one position apart, regardless of their absolute positions in the sequence.

During attention computation, T5 constructs a **relative position bias matrix** where each entry B_{ij} corresponds to the bias associated with the relative offset $(j - i)$ between tokens i and j . This bias matrix is **added directly** to the raw attention scores (as we have seen above) in the self-attention layer.

Despite their theoretical appeal, relative positional embeddings pose certain practical challenges:

1. **Performance Issues:** Benchmarks comparing T5's relative embeddings with other types have shown that they can be slower, particularly for longer sequences (due to the extra computation and bias matrix construction at every layer).
2. **Complexity in Key-Value Cache Usage:** Each additional token changes the relative distances for *all* other tokens in the sequence; the bias matrix must be recomputed or adjusted at each decoding step. This complicates the use of KV caching.

Due to these engineering complexities, **relative embeddings haven't been widely adopted, especially in larger language models.**

RoPE

Rotary Positional Embedding (RoPE), introduced in the RoFormer paper (2021) [74], is a positional encoding method that blends the strengths of both absolute and relative embeddings. Instead of adding position information to token embeddings, RoPE injects positional context **directly into the self-attention mechanism** (as with relative embeddings) by **rotating** query and key vectors using position-dependent sinusoidal transformations.

Given a token at a certain position in the sequence, RoPE applies a **rotation** to its respective key and query vectors based on its position in the sequence. This rotation is achieved by multiplying these vectors with a **position-dependent rotation matrix** whose elements are derived from sine and cosine functions. The rotated key and query vectors are then used to compute the attention scores in the usual way (dot product followed by softmax) and the rest of the computation in the Transformer happen as usual.

The Rotation Matrix

In the simplest case — 2D space — a rotation matrix for an angle θ is:

$$R(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

Multiplying $R(\theta)$ with a 2D vector $x = (x_1, x_2)$ rotates it by θ without changing its length (i.e., changes its **direction** but preserves its **magnitude**):

$$R(\theta)x = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \cos \theta x_1 - \sin \theta x_2 \\ \sin \theta x_1 + \cos \theta x_2 \end{bmatrix}$$

Now, above we said that we multiply query and key vectors for this rotation matrix. We know that query and key vectors are obtained from the given token embedding x via a multiplication of it with the query weight matrix W^Q and the key weight matrix W^K , respectively:

$$q = \begin{bmatrix} W_{11}^Q & W_{12}^Q \\ W_{21}^Q & W_{22}^Q \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad k = \begin{bmatrix} W_{11}^K & W_{12}^K \\ W_{21}^K & W_{22}^K \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Let's consider just the rotation of the query since the same applies for the key. If we apply the rotation matrix to it, we are rotating the query vector as:

$$R(\theta)q = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} W_{11}^Q & W_{12}^Q \\ W_{21}^Q & W_{22}^Q \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

In all above math, we assumed the token embedding x is happening at position 1! If it is happening at an **arbitrary position m** then the rotation matrix will contain m in it:

$$R(\theta)q = \begin{bmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{bmatrix} \begin{bmatrix} W_{11}^Q & W_{12}^Q \\ W_{21}^Q & W_{22}^Q \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

This is what injects **positional information** — every position has its own rotation angle.

Mathematical Proof of Relativity

Now, let's prove that RoPE is relative. To do so, we need to demonstrate that the attention score between two token embeddings depends only on their relative positions, not their absolute positions.

Let's define the RoPE operation for a vector x at position p :

$$RoPE(x, p) = \underbrace{\begin{bmatrix} \cos p\theta & -\sin p\theta \\ \sin p\theta & \cos p\theta \end{bmatrix}}_{R_p} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Consider two tokens (query and key) at positions m and n :

$$q_m = RoPE(q, m)$$

$$k_n = RoPE(k, n)$$

We calculate the attention score as their dot-product:

$$score = q_m \cdot k_n = (R_m q) \cdot (R_n k) = q^T R_m^T R_n k$$

Rotation matrices have this nice property that:

$$R_m^T R_n = R_{m-n}$$

therefore, substituting it the attention score becomes:

$$score = q^T R_m^T R_n k = q^T R_{m-n} k = q^T \begin{bmatrix} \cos(m-n)\theta & -\sin(m-n)\theta \\ \sin(m-n)\theta & \cos(m-n)\theta \end{bmatrix} k$$

As you see, the score is a function of relative positions which is the difference of positions between m and n .

Rotation Matrix for Higher Dimensions

In real models, embedding dimensions are much larger than 2 ($d \gg 2$). So how would the rotation matrix change? The authors in Roformer paper handles this by applying **independent 2D rotations to consecutive pairs of dimensions**:

$$R_{\theta,m}^d = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \dots & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \dots & 0 & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & \cos m\theta_{d/2} & -\sin m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \dots & \sin m\theta_{d/2} & \cos m\theta_{d/2} \end{pmatrix}$$

That is, for position m and with embedding dimension d , the rotation matrix is composed of **$d/2$ rotation matrices each 2-by-2**.

Another important point is that since this construction is **sparse**, the authors recommended a computationally efficient way to compute its product with a token embedding vector x by:

$$R_{\theta,m}^d x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \odot \begin{pmatrix} \cos m\theta_1 \\ \cos m\theta_1 \\ \cos m\theta_2 \\ \cos m\theta_2 \\ \vdots \\ \cos m\theta_{d/2} \\ \cos m\theta_{d/2} \end{pmatrix} + \begin{pmatrix} -x_2 \\ x_1 \\ -x_4 \\ x_3 \\ \vdots \\ -x_d \\ x_{d-1} \end{pmatrix} \odot \begin{pmatrix} \sin m\theta_1 \\ \sin m\theta_1 \\ \sin m\theta_2 \\ \sin m\theta_2 \\ \vdots \\ \sin m\theta_{d/2} \\ \sin m\theta_{d/2} \end{pmatrix}$$

This variant applies sine and cosine **element-wise** rather than performing a full matrix multiplication.

Multi-Head Latent Attention

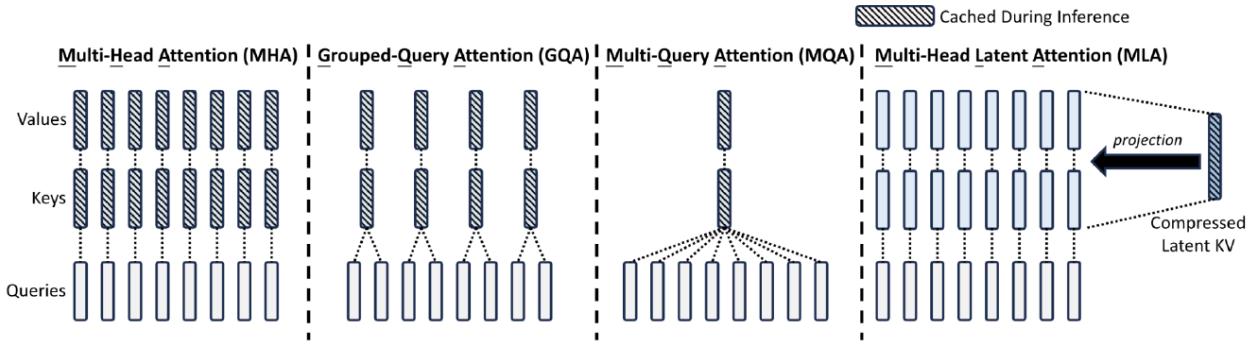
DeepSeek is a family of LLMs developed by the Chinese company DeepSeek. It gained massive attention in **early 2025** for **achieving performance** comparable to state-of-the-art models like GPT-4, while being **significantly cheaper to train and run**. This was achieved through a strong focus on **memory- and compute-efficient design**, rather than simply scaling up model size.

What further sets DeepSeek apart was its commitment to openness: unlike many closed-source models, DeepSeek released **open weights and inference code**, making cutting-edge LLM technology broadly accessible.

Its focus on efficiency sparked massive interest in the AI community because it challenged the prevailing notion that “bigger always means better”, showing instead that **smarter architecture design and optimization matter just as much as raw size**.

Here we’ll focus on **Multi-Head Latent Attention (MLA)** — one of DeepSeek’s most impactful innovations — introduced in **DeepSeekV2** (2024) [75] and later retained in more recent DeepSeekV3 [76] and DeepSeekR1 [77]. This section is based on this [blog post](#).

The core idea behind MLA is to **compress the attention input h_t** into a low-dimensional *latent* vector of dimension d_c , where $d_c \ll (d_h h_n)$. Later, when we need to calculate attention, this compact **latent representation is projected back into the high-dimensional space to recover the keys and values**. Because only the latent vector needs to be stored — instead of the full high-dimensional representations — this approach drastically reduces memory usage.



Formally, let c_t^{KV} denote the latent vector of the original key/value, W^{DKV} the **down-projection** matrix compressing from $(h_n \cdot d_h)$ to d_c (with “ D ” here we indicate *downsampling*), and W^{UK} , W^{UV} the **up-projection** matrices expanding from d_c back to the high-dimensional space (with “ U ” here we indicate *upsampling*):

$$\begin{aligned} c_t^{KV} &= W^{DKV} h_t \\ k_t^C &= W^{UK} c_t^{KV} \\ v_t^C &= W^{UV} c_t^{KV} \end{aligned}$$

Similarly, queries can be compressed and then up-projected as:

$$\begin{aligned} c_t^Q &= W^{DQ} h_t \\ q_t^C &= W^{UQ} c_t^Q \end{aligned}$$

Decoupled RoPE

As we discussed before, RoPE is a common choice for training generative models to handle long sequences. However, **applying MLA directly with RoPE introduces a compatibility issue**.

To see this more clearly, consider what happens when we calculate attention: when we multiply q^T with k , the matrices W^Q and W^{UK} will appear in the middle, and their combination equivalents to a single mapping dimension from d_c to d . In the original paper, the authors describe this as W^{UK} being mathematically “absorbed” into W^Q . As a result, we do not need to store W^{UK} in the cache, further reducing memory usage.

However, this will not be the case when we take the rotation matrix into consideration, as RoPE will apply a rotation matrix on the left of W^{UK} and this rotation matrix will end up in between the transposed W^Q and W^{UK} . This rotation matrix is position-dependent, meaning the **rotation matrix for each position is different**. As a result, W^{UK} **can no longer be absorbed by W^Q** .

To address this conflict, the authors propose what they call “**a decoupled RoPE**” by introducing **additional query vectors along with a shared key vector** and **use these additional vectors in the RoPE process only**, while keeping the **original keys isolated by the rotation matrix**.

The entire process of MLA can be summarized as below:

$$\begin{aligned} c_t^Q &= W^{DQ} h_t \\ [q_{t,1}^C; q_{t,2}^C; \dots; q_{t,n_h}^C] &= q_t^C = W^{UQ} c_t^Q \\ [q_{t,1}^R; q_{t,2}^R; \dots; q_{t,n_h}^R] &= q_t^R = \text{RoPE}(W^{QR} c_t^Q) \\ q_{t,i} &= [q_{t,i}^C; q_{t,i}^R] \end{aligned}$$

$$c_t^{KV} = W^{DKV} h_t$$

$$[k_{t,1}^C; k_{t,2}^C; \dots; k_{t,n_h}^C] = k_t^C = W^{UK} c_t^{KV}$$

$$k_t^R = RoPE(W^{KR} h_t)$$

$$k_{t,i} = [k_{t,i}^C; k_{t,i}^R]$$

$$[v_{t,1}^C; v_{t,2}^C; \dots; v_{t,n_h}^C] = v_t^C = W^{UV} c_t^{KV}$$

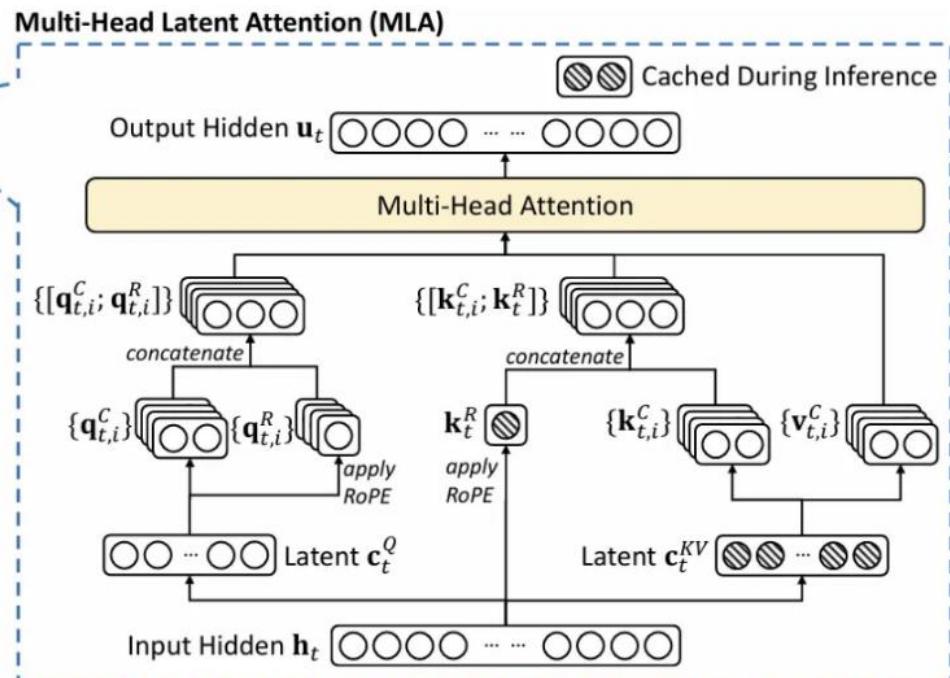
$$o_{t,i} = \sum_{j=1}^t \text{Softmax}_j \left(\frac{q_{t,i}^T k_{j,i}}{\sqrt{d_h + d_h^R}} \right) v_{j,i}^C$$

$$u_t = W^O [o_{t,1}; o_{t,2}; \dots; o_{t,n_h}]$$

where

- 1 – 4 equations describe how to process query tokens.
- 5 – 6 equations describe how to process key tokens.
- 7 – 8 equations describe how to use the additional shared key for RoPE. Be aware that the output of equation 6 is **not involved in RoPE**.
- equation 9 describes how to process value tokens.
- remaining equations describe the attention output as usual.

During this process, **only c_t^{KV}, k_t^R need to be cached!** This process can be illustrated more clearly with the flowchart below:



Performance of MLA

The table below compares the number of elements needed for KV cache (per token) as well as the modeling capacity between MHA, GQA, MQA and MLA, demonstrating that MLA could indeed achieve a better balance between memory efficiency vs. modeling capacity.

Interestingly, [the modeling capacity for MLA even surpasses that of the original MHA](#).

Attention Mechanism	KV Cache per Token (# Element)	Capability
Multi-Head Attention (MHA)	$2n_h d_h l$	Strong
Grouped-Query Attention (GQA)	$2n_g d_h l$	Moderate
Multi-Query Attention (MQA)	$2d_h l$	Weak
MLA (Ours)	$(d_c + d_h^R)l \approx \frac{9}{2}d_h l$	Stronger

More specifically, the Table below shows the performance of MHA, GQA and MQA on 7B models, where MHA significantly outperforms both MQA and GQA.

Benchmark (Metric)	# Shots	Dense 7B w/ MQA	Dense 7B w/ GQA (8 Groups)	Dense 7B w/ MHA
# Params	-	7.1B	6.9B	6.9B
BBH (EM)	3-shot	33.2	35.6	37.0
MMLU (Acc.)	5-shot	37.9	41.2	45.2
C-Eval (Acc.)	5-shot	30.0	37.7	42.9
CMMLU (Acc.)	5-shot	34.6	38.4	43.5

The authors also conduct analysis between MHA vs. MLA, and results are summarized in the Table below, where MLA achieves better results overall.

Benchmark (Metric)	# Shots	Small MoE w/ MHA	Small MoE w/ MLA	Large MoE w/ MHA	Large MoE w/ MLA
# Activated Params	-	2.5B	2.4B	25.0B	21.5B
# Total Params	-	15.8B	15.7B	250.8B	247.4B
KV Cache per Token (# Element)	-	110.6K	15.6K	860.2K	34.6K
BBH (EM)	3-shot	37.9	39.0	46.6	50.7
MMLU (Acc.)	5-shot	48.7	50.0	57.5	59.0
C-Eval (Acc.)	5-shot	51.6	50.9	57.9	59.2
CMMLU (Acc.)	5-shot	52.3	53.4	60.7	62.5

For a deeper understanding of MLA, I strongly recommend reading the original paper. Additionally, this [video overview](#) offers an excellent high-level and visual summary of all the concepts we've covered in this section — including MHA, KV Cache, MQA, GQA and finally MLA — making it a great resource to consolidate their understanding.

Graph Neural Networks

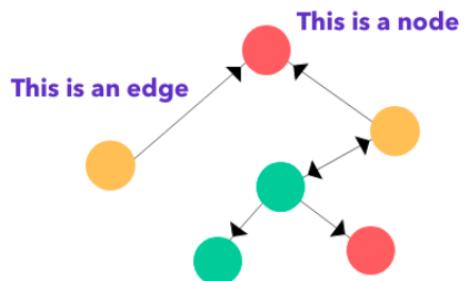
In previous chapters, we explored structured data in the form of sequences (e.g. text sentences) and images — corresponding to one- and two-dimensional arrays of variables, respectively. More generally, however, many types of structured data are best represented as **graphs**. In this chapter, we turn our attention to **Graph Neural Networks (GNNs)** [78], a class of models designed to work directly with graph-structured data.

GNNs have been actively developed for over a decade, offering powerful tools for learning on relational data. This chapter builds on the intuitive and visually rich explanations provided by the *Distill.pub* blog posts, [Introduction to GNNs](#) and [Understanding GNNs](#). Since those resources provide many helpful visualizations, I suggest starting with this chapter to gain an intuitive understanding. Afterward, I strongly recommend moving on to the discussion in the recommended textbook — specifically the chapter *Graph Neural Networks* — for a more rigorous treatment and a more focused view of the **core concepts behind GNNs**.

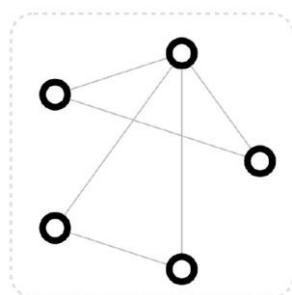
What is a Graph?

A **graph** is a flexible structure consisting of **nodes** (also known as **vertices**) and **edges**. Edges represent the relationships we can define between different nodes.

- The **black arrows** on the edges represent **the kind of relationship** between the nodes. It shows whether a relationship is **mutual** or **one-sided**.
- Graphs can be directed (the order of connection between nodes is important) or undirected (the order of connection is not important).
- Directed graphs can be unidirectional or bidirectional in nature.

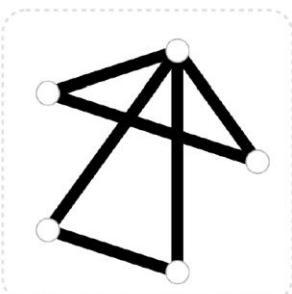


In a graph, we can define attributes for our nodes, which are usually represented as **embeddings**.



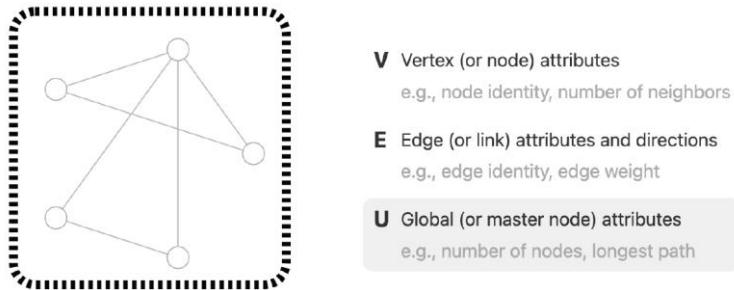
- **V** Vertex (or node) attributes
e.g., node identity, number of neighbors
- **E** Edge (or link) attributes and directions
e.g., edge identity, edge weight
- **U** Global (or master node) attributes
e.g., number of nodes, longest path

Edges can also be represented as embeddings.



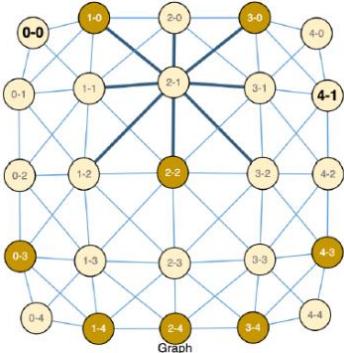
- **V** Vertex (or node) attributes
e.g., node identity, number of neighbors
- **E** Edge (or link) attributes and directions
e.g., edge identity, edge weight
- **U** Global (or master node) attributes
e.g., number of nodes, longest path

We can also have attributes that are related to the entire graph, thus obtaining a global embedding of the graph.



Images as Graphs

We typically think of images as rectangular grids, representing them as **rectangular grids with image channels** (e.g., 244x244x3 floats).



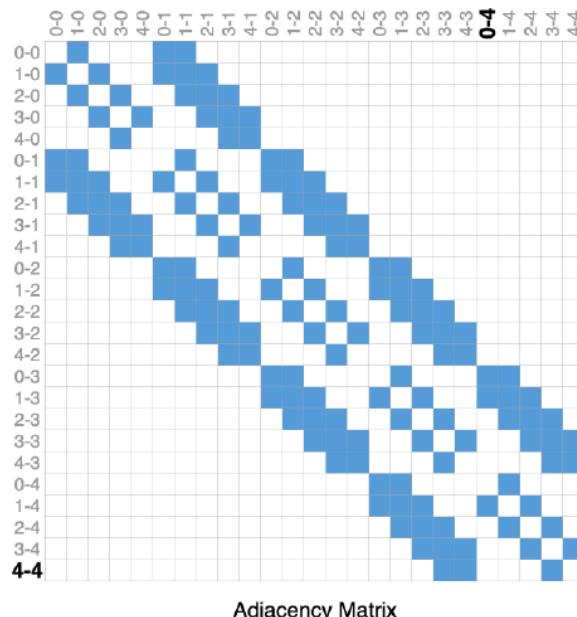
Another way to think of images is as **graphs** with a regular structure, where **each pixel represents a node and is connected via an edge to adjacent pixels**.

Each non-border pixel has exactly 8 neighbors and the information stored in each node is a 3-dimensional vector representing the pixel's RGB value.

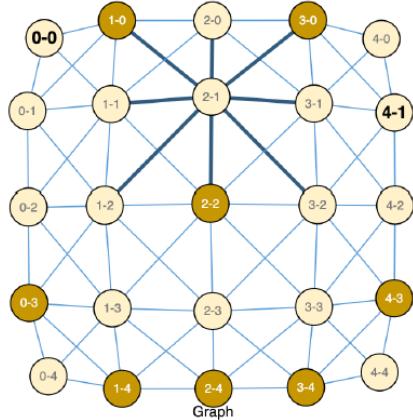
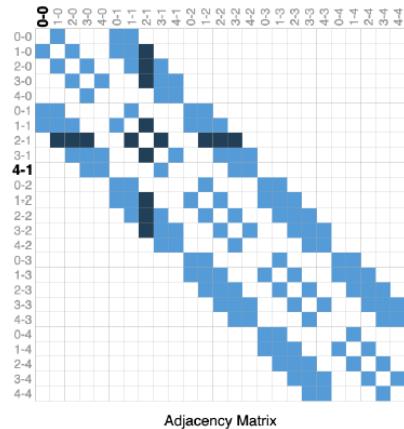
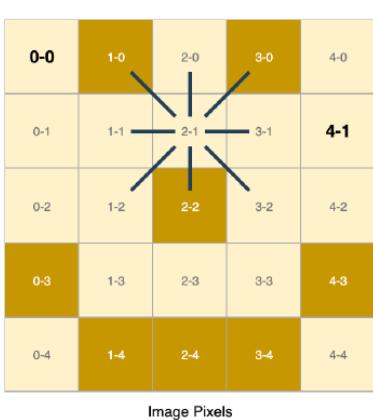
0-0	1-0	2-0	3-0	4-0
0-1	1-1	2-1	3-1	4-1
0-2	1-2	2-2	3-2	4-2
0-3	1-3	2-3	3-3	4-3
0-4	1-4	2-4	3-4	4-4

Image Pixels

Another way to visualize the connectivity of a graph is through its **adjacency matrix**: essentially, we sort the nodes — in this case, each of the 25 pixels in a simple 5x5 image — and **fill an $n \times n$ matrix with a 1** (or a color in the case of the Figure below) **if two nodes share an edge**.

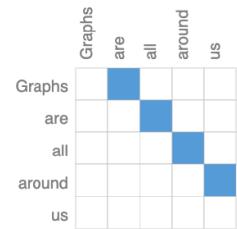


So, essentially, each of these three different representations below are different views of the same data.



Text as Graphs

Text can also be represented as a graph, but in this case the solution is slightly different because our adjacency matrix is much smaller. In fact, the adjacency matrix for text appears as a **simple diagonal line**, as each word is connected only to the previous and next word.

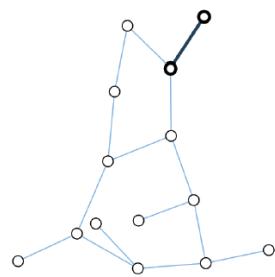
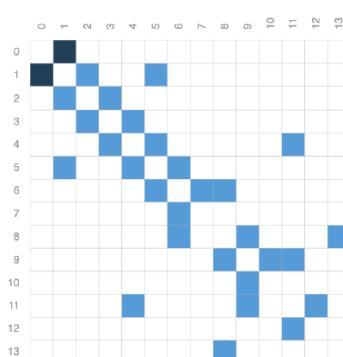
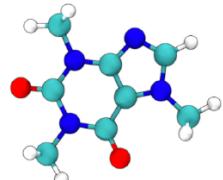


Molecules as Graphs

We can also imagine representing biological examples, e.g. **molecules**, in a graph.

If we consider the different atoms in a molecule, we can consider the distance of all the different covalent bonds we might have with respect to the atoms. Then, based on these bonds, we could create a 3-D object, and this could be represented as a graph where **each node is an atom and the edges are the covalent bonds holding the atom together**.

For example, here is the adjacency matrix for a caffeine molecule.

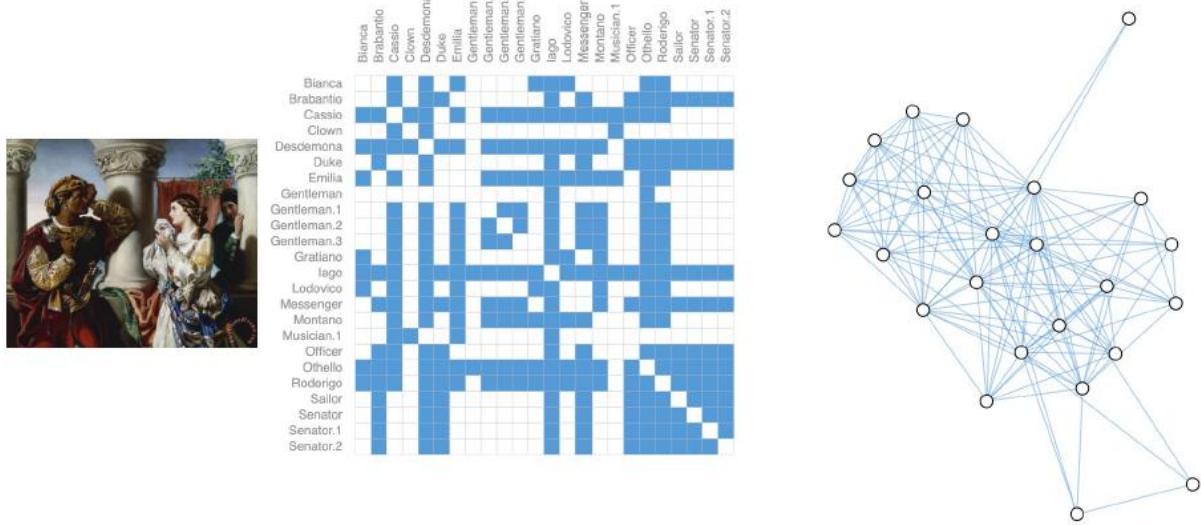


Social Networks as Graphs

Another use of data that can be structured as a graph is **social networks**.

- Social networks are tools to study patterns in collective behavior of people, institutions and organizations.
- We can build a **graph representing groups** of people by modelling **individuals as nodes and their relationships as edges**.

For example, here is the social network from Shakespeare's *Othello*. We've plotted whether the different actors or characters meet in the story. So here we can see the adjacency matrix and the graph.



Graphs of Different Sizes

Graphs can be very different from each other, meaning we can have very large graphs, like those of *Meta* or *Instagram*, or smaller graphs like the *Karate Club* social network.

Dataset	Domain	Edges per node (degree)					
		graphs	nodes	edges	min	mean	max
karate club	Social network	1	34	78		4.5	17
qm9	Small molecules	134k	≤ 9	≤ 26	1	2	5
Cora	Citation network	1	23,166	91,500	1	7.8	379
Wikipedia links, English	Knowledge graph	1	12M	378M		62.24	1M

The Karate Club has a maximum number of edges per node of 17 and an average of 4.5. In contrast, Wikipedia has an average of about 60 edges per node and the **maximum number of edges per node is 1 million!**

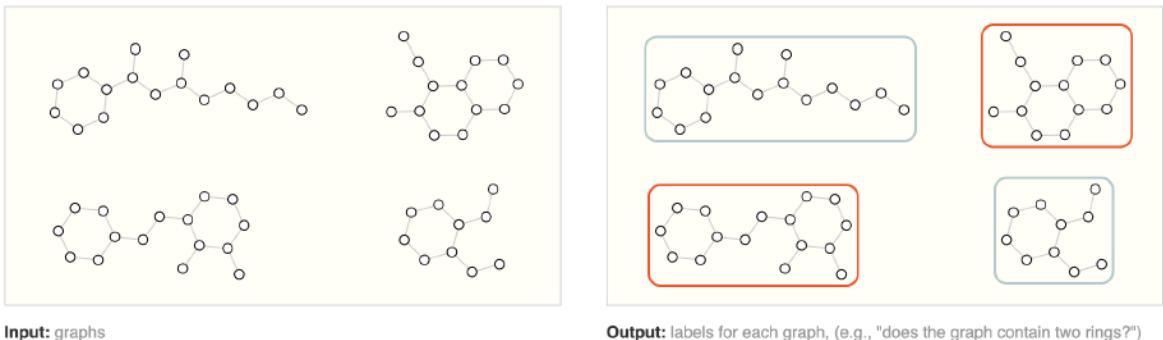
So, we want to find ML techniques that can work with these different types of graphs.

Graph-level Tasks, Node-level Tasks & Edge-level Tasks

In a graph we can define different types of tasks:

- **Graph-level tasks:** With this type of task, the goal is to **predict the property of an entire graph**.

For example, for a molecule represented as a graph, we might want to create a classifier that determines whether a given molecule contains at least two rings.

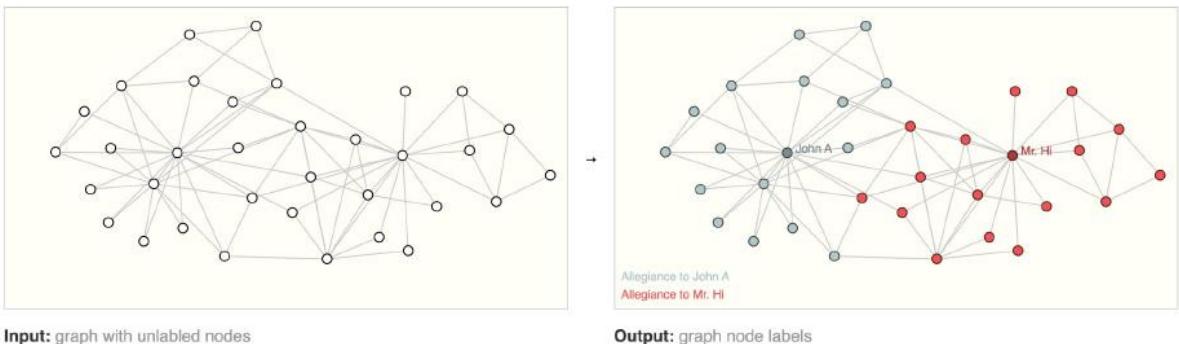


This is analogous to image classification problems with MNIST and CIFAR, where we want to **associate a label to an entire image**. With text, a similar problem is sentiment analysis.

- **Node-level tasks:** With this type of task, we want to identify the role or category of each node within the graph (for example, **assign each node to a specific category**).

A classic example of a node-level prediction problem is Zach's Karate Club.

In this case, there's a feud between two characters, Mr. Hi (Instructor) and John H (Administrator), and each club member is asked to side with one of them. The prediction problem here is to classify whether a given member will side with one or the other after the feud.

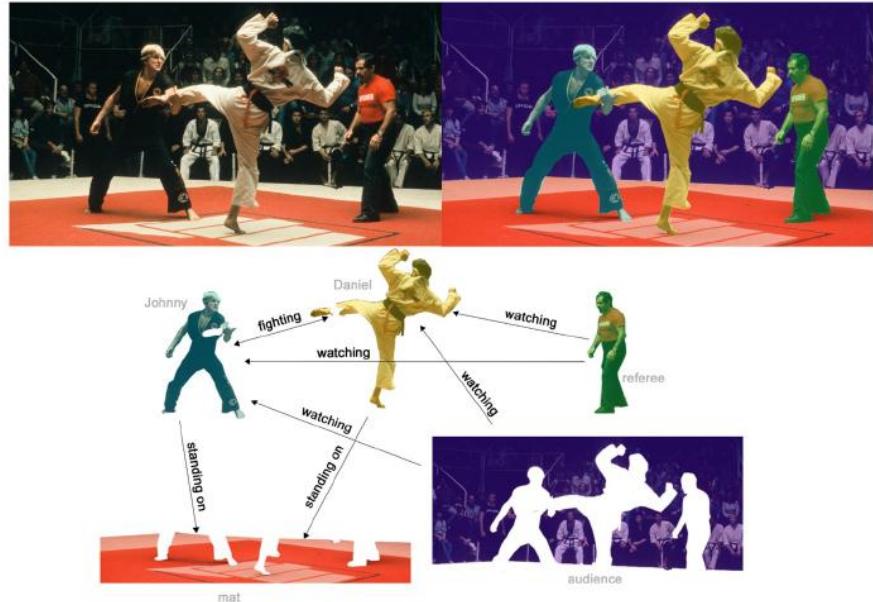


In this case, the distance between a node and the Instructor or Administrator is highly correlated to this label.

Following the image analogy, node-level prediction problems are analogous to image segmentation, where we try to label the role of each pixel in an image. With text, a similar task would be to predict the parts-of-speech of each word in a sentence (e.g., noun, verb, adverb, etc.).

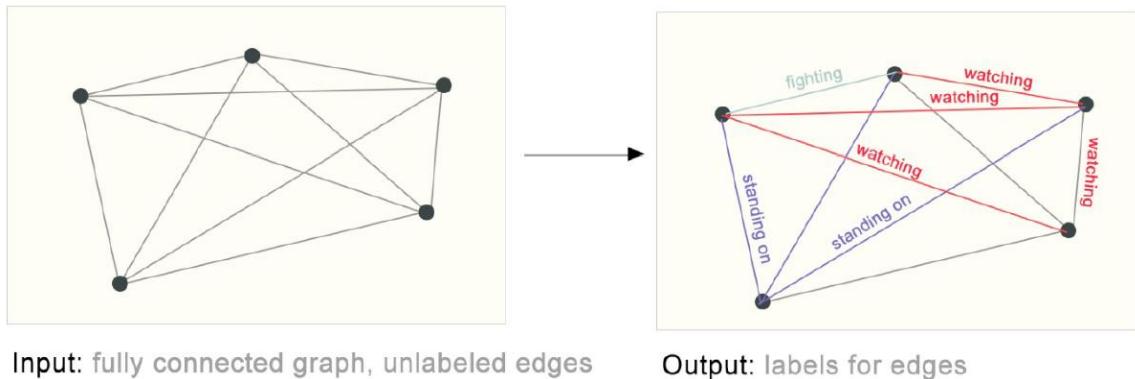
- **Edge-level task:** With this type of task, we want to **label or classify the edges of a graph**.

An example of edge-level inference is **image scene understanding**. Imagine we have a segmented image containing several elements: opponent A (Daniel), opponent B (Johnny) and the judge.



Each of these elements is a node, and we know they are related to each other through some activity. The judge is watching Daniel and Johnny, the two contestants are standing on stage mat, and they are fighting.

What we want to do is predict which of these nodes share an edge or what the value of that edge is. To do this, we can imagine that the graph is initially fully connected, so **we can make predictions and then, based on a threshold, we can prune the edges to arrive at a sparse graph**.



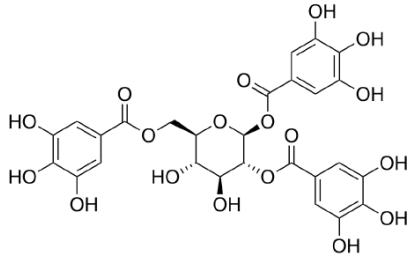
The Challenges of Computation on Graphs

Graphs are very different from the type of input we've analyzed so far, which is why they are studied in a different branch of ML and why it took many decades to develop high-level graph machine learning models.

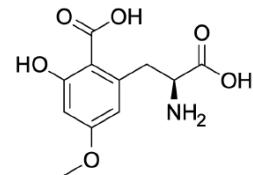
Let's look at the advantages and disadvantages of using graphs.

The advantage of using graphs is that they are **extremely flexible mathematical models**; we can represent practically anything with them, but this also means they **lack a consistent structure** (they can be very different from each other).

For example, consider two different graphs, in this case graphs of molecules:



A non-toxic 1,2,6-trigalloyl-glucose molecule.



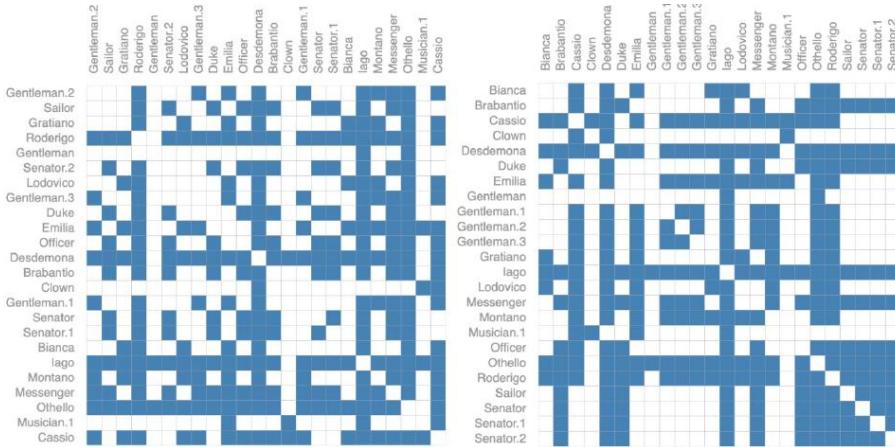
A toxic caramboxin molecule.

These two are completely different graphs. Looking at them, the following problems immediately emerge:

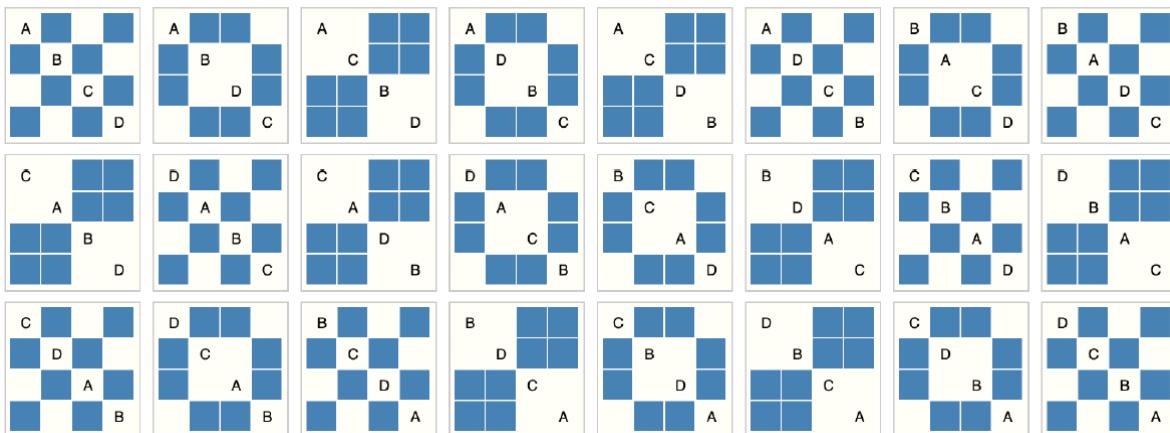
- We could have a **different number of atoms** or the atoms could be labeled differently because they are of **different types**.
- Each atom could have a **different number of connections** and even if they have the same connections, they could have **different strengths**.

Representing graphs in a processable format is **non-trivial** and the final representation chosen often depends on the specific problem we want to solve.

For example, the Othello graph we saw previously **can be described equivalently with these two adjacency matrices**:



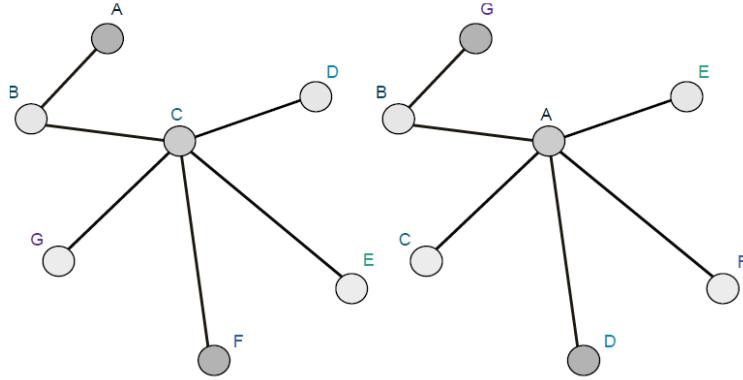
But not only! In general, it **can also be described with any other possible permutation** of the nodes.



So here we have a representation problem because we can't use a single adjacency matrix and use the techniques we've seen so far; we'd have to consider every possible permutation of these, and that's impractical.

Node-Order Equivariance

What we want is for our graph algorithms to be **node-order equivariant**, meaning they behave consistently under permutations of the node ordering. In other words, suppose we take a graph and apply a permutation A to its nodes, then run our algorithm and obtain a result. If we then apply a different permutation B to the same graph and run the algorithm again, the same difference in terms of permutation between A and B should persist between the results.

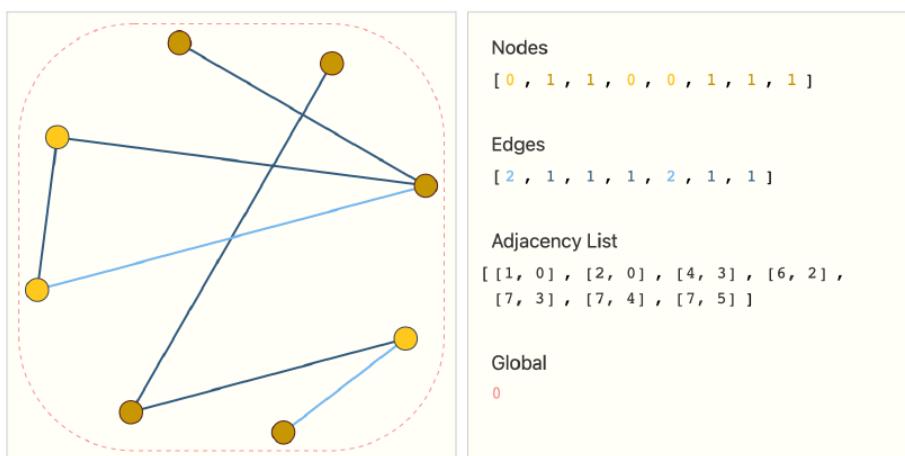


This property ensures that the algorithm's behavior is independent of how we label or order the nodes. Soon we will see that to achieve permutation equivariance, we can adopt aggregation operations such as **sum**, **mean** or **max**, which naturally respect node permutations.

Scalability & Sparsity

We've seen the problems we can encounter with a relatively small graph, like Othello, so we could imagine the difficulties we might encounter if we considered **much larger graphs** like Meta or X social networks, which have over a billion users. Luckily, **natural graphs are "sparse"**. This allows the use of methods to efficiently compute representations of nodes within the graph. Further, we will see that **many methods permit to have significantly fewer parameters** in comparison to the size of the graphs they operate on.

So, on the one hand, we want to exploit the sparseness of graphs; on the other, we want to use models that scale well with increasing connections or vertices. How can we do this? One elegant and memory-efficient way to do this is to use "**adjacency list**", another type of representation that lists the graph connections. The k -th entry of an adjacency list is indicated as a tuple (i, j) , representing the connectivity of an edge e_{ij} between nodes n_i and n_j . Below, we can see how information in a graph could be represented under this specification:



An important aspect in the use of an adjacency list is that since the number of edges is much smaller than the number of entries required for an adjacency matrix (which has n^2 entries), we avoid computing and storing the disconnected parts of the graph.

It should also be noted that the Figure uses scalar values per node/edge/global, but **most practical tensor representations use embeddings for each graph attribute**. In that case, instead of a node tensor of size $[n_{nodes}]$, we will be dealing with node tensors of size $[n_{node}, node_{dim}]$. The same applies to the other graph attributes.

Building a Graph Neural Network

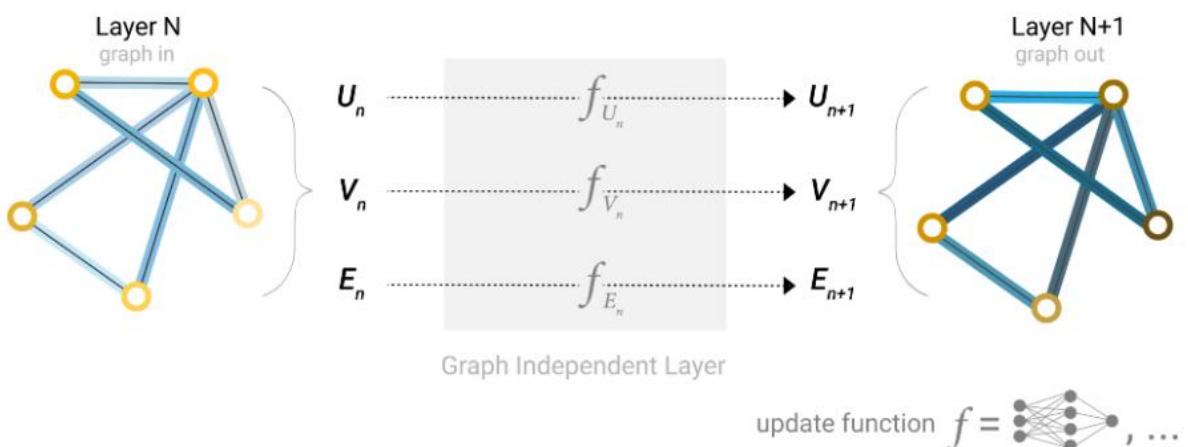
A **Graph Neural Network (GNN)** can be described as an optimizable transformation on all attributes of the graph (nodes, edges, global-context) that preserves graph symmetries (permutation invariances). So our objective will be to understand how to ensure it.

We're going to build GNNs using the “**message passing neural network**” framework proposed by Gilmer et al. (2017) [79] using the Graph Nets architecture schematics introduced by Battaglia et al. (2018) [80]. These GNNs adopt a “**graph-in, graph-out**” architecture meaning that these types of models accept a graph as input, with information loaded into its nodes, edges and global-context, and progressively transform these embeddings, without changing the connectivity of the input graph. To achieve this they use a message pass scheme which we will describe in next sections.

The Simplest GNN

We start with the simplest GNN architecture, which allow us to **learn new embeddings for all graph attributes** (nodes, edges, global context), but **without yet considering the graph's connectivity**.

This GNN applies a separate **multilayer perceptron (MLP)** on each component of a graph; we call this a **GNN layer**. The goal of each MLP is to transform the current embedding of a component (e.g., a node) into a **new learned embedding**. This means that we first apply the MLP to each node vector (f_{V_n}), obtaining a new embedding for each of them. The same process is also applied to the edges (f_{E_n}), obtaining a specific embedding for each edge of the graph. Finally, we apply the MLP to the global context vector (f_{U_n}), obtaining a single embedding representing the entire graph.



Once we have the new embeddings for each component of the graph, we can stack these "GNN layers" together, obtaining an updated representation of the graph with improved embeddings for each component.

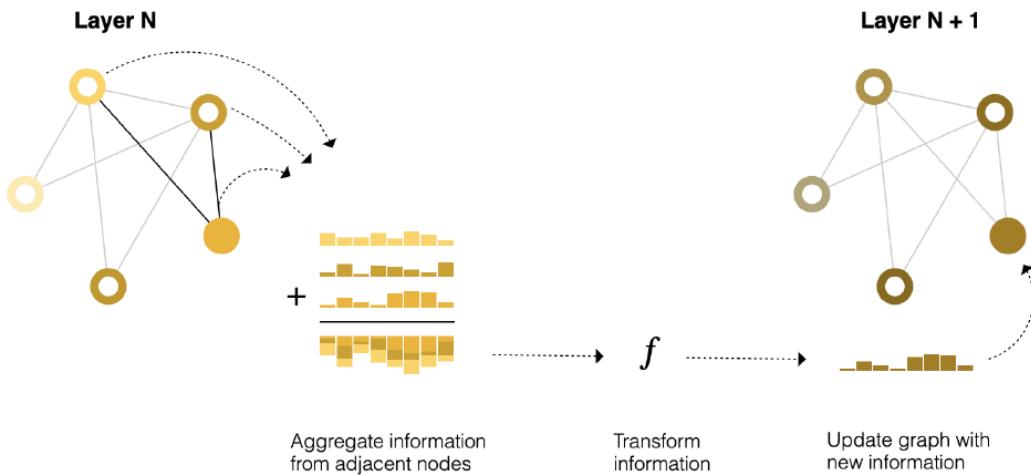
Note: It's important to note that since a GNN doesn't change the connectivity of the input graph, the GNN's output graph will have the same structure as the input graph, with the same adjacency list and the same number of attribute vectors. However, the embeddings associated with the nodes, edges and global context will have been updated and learned during the process of passing through the GNN layers.

Message Passing

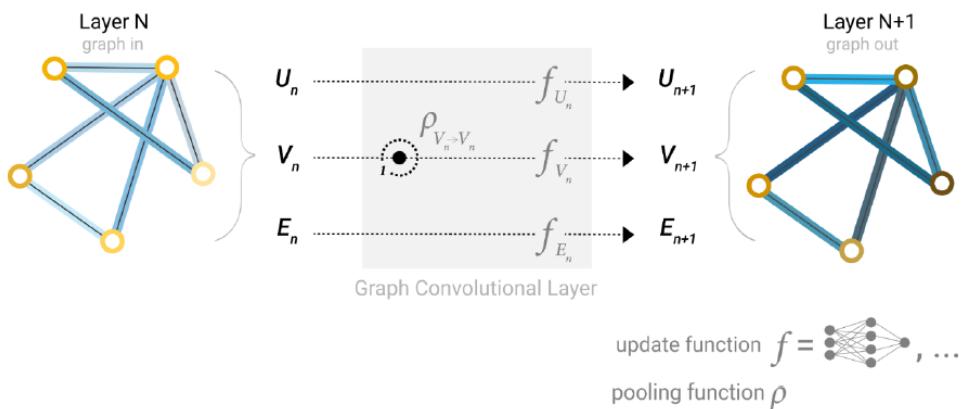
The simple GNN above processes nodes, edges, and globals independently. To exploit **graph connectivity**, we need a way for elements to communicate with their neighbors. This strategy is known as **message passing**. The fundamental idea is that **neighboring nodes exchange information and influence each other's updated embeddings**.

Message passing works in three steps:

1. **For each node in the graph, gather all the neighboring nodes embeddings** (or "messages"). This allows us to gain insight into each node's local context and its interaction with neighboring nodes.
2. **Aggregate all the gathered messages using an *aggregation function***, such as the sum. This way, we obtain an aggregate representation of the information from neighboring nodes.
3. **Apply an *update function***, often implemented as a neural network, **to update the node embedding with the new information coming from the aggregated neighbors**.



We can update the architecture diagram seen previously to include this new source of information for nodes:



Here the letter ρ represents the *aggregation* operation and $\rho_{V_n \rightarrow V_n}$ denotes that we are gathering information from nodes to nodes.

Note: It is important to notice that in the aggregation step we are considering a **sum** over the neighbors. Since summation is a **commutative operation**, the outcome is **equivariant with respect to the order of the nodes**, i.e. the order of the neighbors doesn't matter. In other words, even if we rearrange the order of the nodes in the graph, the updated embeddings produced by message passing remain unchanged. Therefore, the overall goal we set ourselves—that of obtaining algorithms that work consistently regardless of the specific organization of the nodes within the graph—has been effectively achieved.

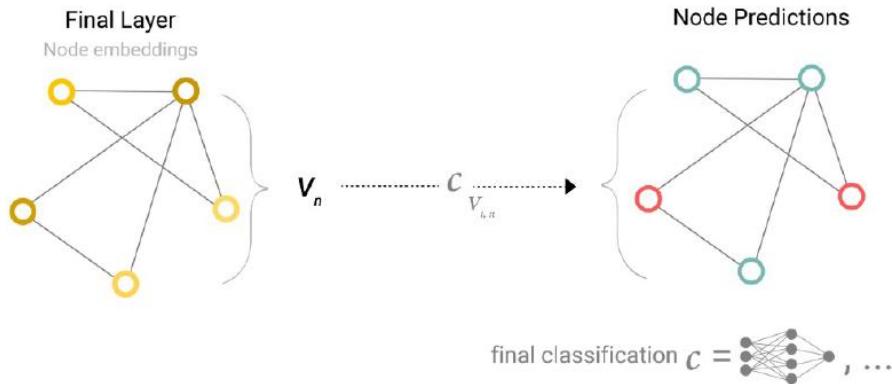
By stacking message passing layers, nodes progressively incorporate information from larger neighborhoods. **After k layers, a node embedding reflects information up to k hops away.** For instance, using three layers, a node has information about the nodes three steps away from it. By stacking more message passing GNN layers together, a node can eventually incorporate information from across the entire graph.

This process can be applied not only to nodes but also to edges, depending on what features we want to propagate across the graph. In the end of the chapter will build more elaborate variants of message passing in GNN layers that yield GNNs of increasing expressiveness and power.

GNN Predictions Using Information Aggregation

We've built a simple GNN, but how do we make predictions in any of the tasks we described above?

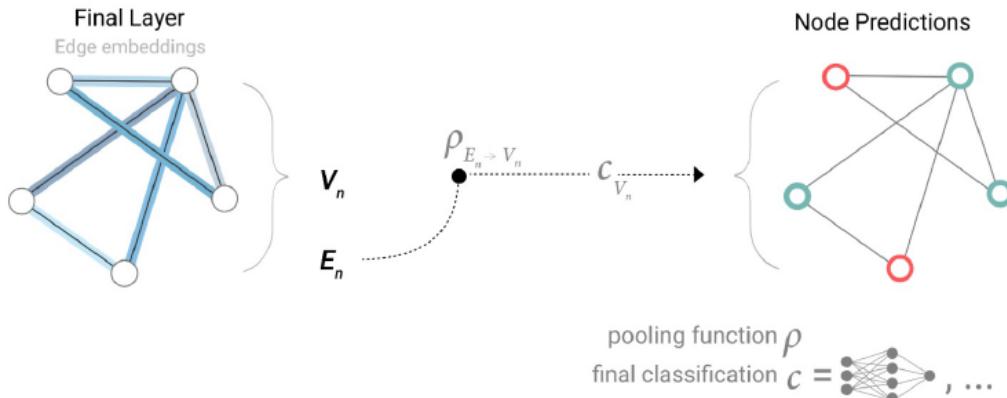
We'll consider the case of binary classification, but this approach can easily be extended to multi-class classification or regression case. If the task is to make binary predictions, for example on nodes, and the graph already contains node information, the approach is straightforward—**for each node embedding, apply a classifier c .**



So, essentially, we're taking the node embeddings of this graph and classifying them (shown in red or green in the Figure). However, it's not always so simple. For example, we might have information about the graph's edges, but no information about the nodes, yet we still need to make predictions on nodes. In this case, we need a way to collect information from the edges and give them to nodes for predictions. To do this, we can make use of **aggregation**:

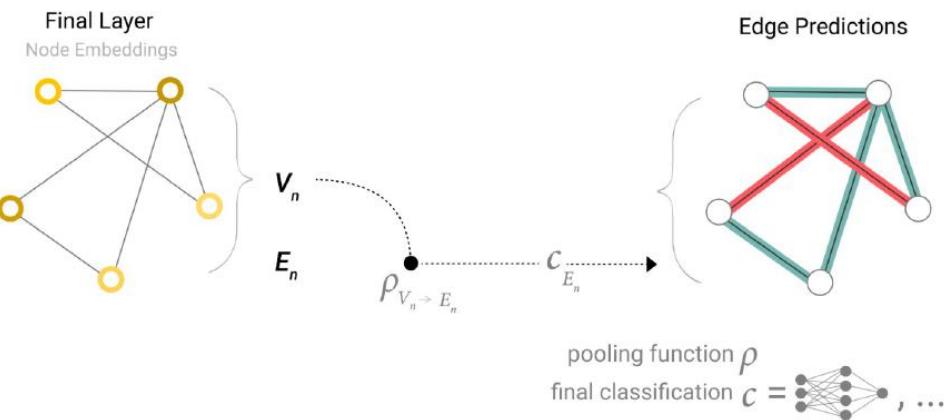
1. For each element of the graph to be aggregated, in this case edges, *gather* each of their embeddings and concatenate them into a matrix.
2. The gathered embeddings are then **aggregated**, usually via a **sum** operation.

So, if we only have edge-level features and want to make binary predictions about the nodes, **we can use aggregation to route (or pass) information to where it needs to go**. Then, as we did before, we can apply a classifier and use its output to make binary predictions about the nodes (in this case):

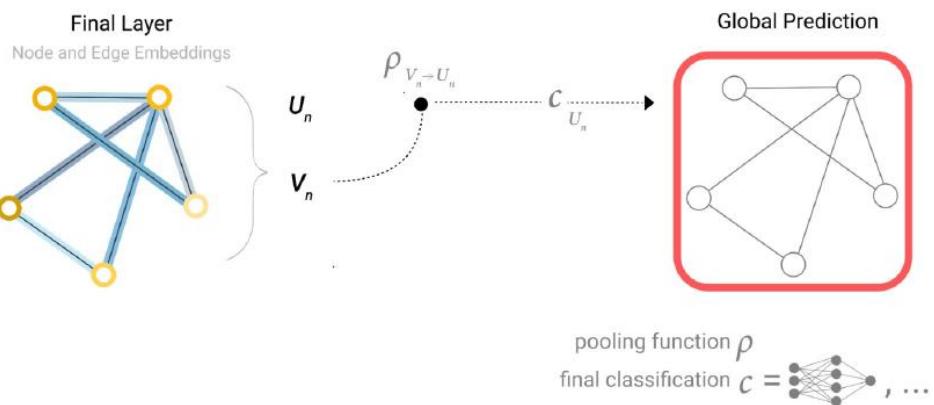


Note: In this case we denote with $\rho_{E_n \rightarrow V_n}$ the fact that we are gathering information from edges to nodes.

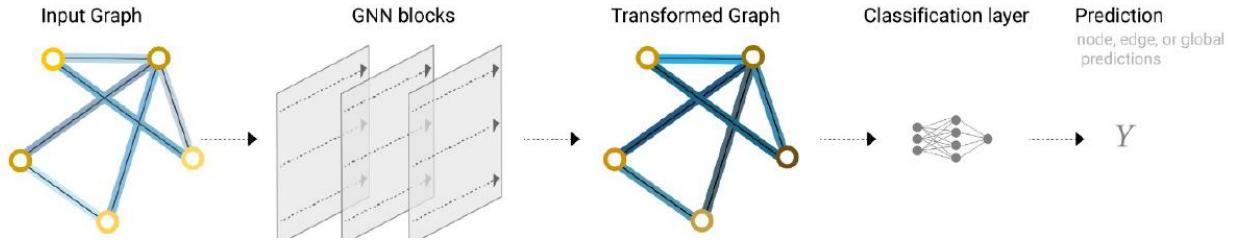
Similarly, if we only have node-level features and we are trying to predict binary edge-level information, the model looks like this:



Or if we only have node-level features and need to predict a global binary property, we need to gather all available node information together and aggregate them.



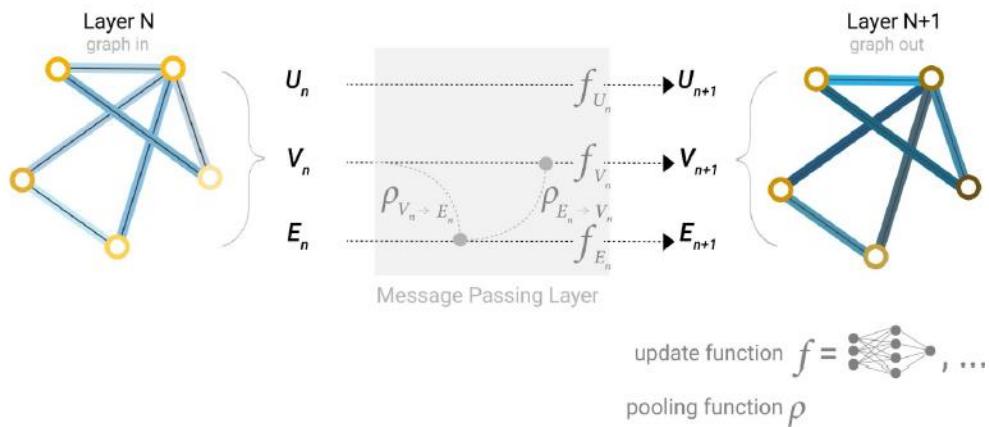
In our examples, the classification model c can be easily replaced with any differentiable model or adapted to multi-class classification using a generalized linear model.



Learning Edge Representations

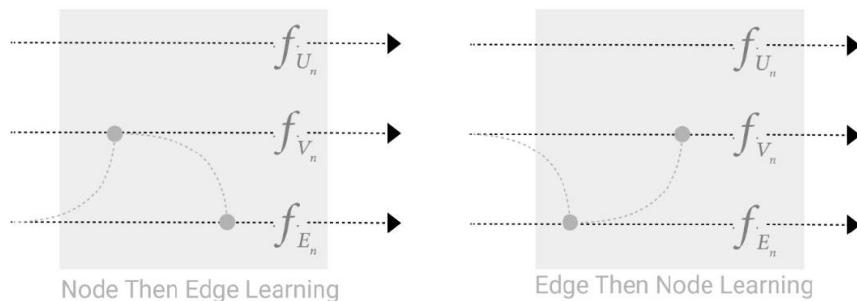
Above, we routed information between nodes and edges only at the prediction stage. However, we can think of **sharing information between nodes and edges within the GNN layer using message passing**.

If we now want to incorporate information from neighboring edges, we can follow the same approach for message passing used for neighboring nodes as previously, but applied to edges: first *aggregate* the edge information, then transforming it using an *update function* and at the end storing the results obtained (thus obtaining new embeddings).



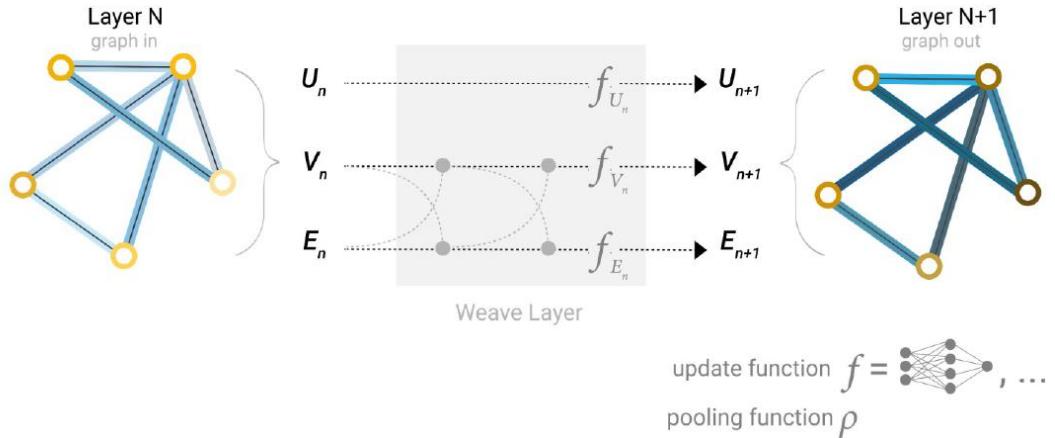
However, the node and edge information stored in a graph **are not necessarily the same size or shape**, so it is not immediately clear how to combine them. One way is to learn a **linear mapping** from the space of edges to the space of nodes, and vice versa. Alternatively, one may **concatenate** them together before the update function.

Moreover, it's important to consider the order in which we update the graph attributes, i.e., the node and edge embeddings, when constructing GNNs. In fact, we might want to update the node embeddings first and then the edge embeddings, or vice versa. The important thing is to maintain consistency with the chosen update order.



The order in which attributes are updated is generally a **design decision** when constructing GNNs. This is an open area of research with a variety of solutions - for example, we could update attributes in a '**weave**' fashion where we have four updated representations that get combined into new node and edge representations:

- node-to-node (linear),
- edge-to-edge (linear),
- node-to-edge (edge layer),
- edge-to-node (node layer).

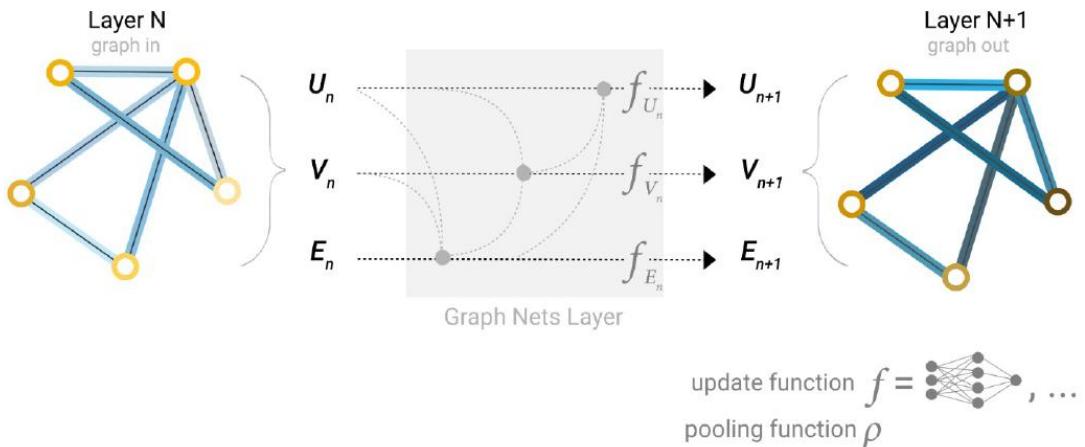


Adding Global Representations

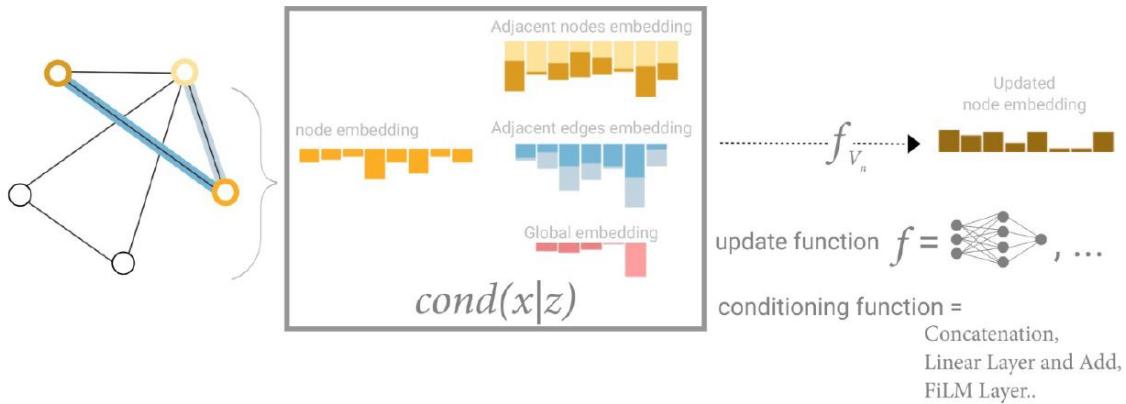
There is **one flaw** in the networks we've described so far: **nodes that are far in the graph may never be able to efficiently pass information to each other**, even if we apply message passing several times.

Before we said that, for one node if we have k layers, information propagates up to at most k steps away. This can be a problem in situations where the prediction task depends on nodes, or groups of nodes, that are far apart. One solution would be to **allow all nodes to pass information to each other**. Unfortunately, **for large graphs, this operation quickly becomes computationally expensive** (although this approach, called "virtual edges", has been used for small graphs such as molecules).

A **better solution** to this problem is to use the so-called **global representation of the graph (U)**, sometimes also referred as *master node* or *context vector*. **This global context vector is connected to all other nodes and edges in the network** and can **act as a bridge between them**, building up a representation for the graph as a whole. This creates a richer and more complex representation of the graph than could have otherwise been learned.

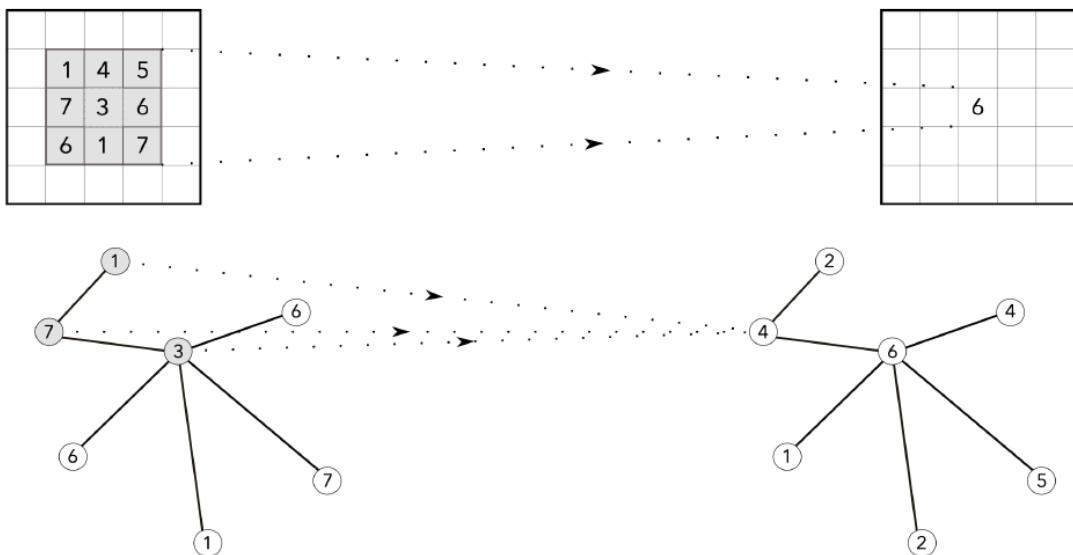


So, now we have a graph framework where all attributes — nodes, edges and the global context — have learned representations, which we can leverage during aggregation. For instance, when updating a node's embedding, we can condition it on all these possible sources of information. A simple approach is to **concatenate** these embeddings. Alternatively, we may also map them to the same space via a **linear mapping and add** them together or apply a **feature-wise modulation layer (FiLM)**, which dynamically adjusts each feature using learned *scale* and *shift* parameters. This modulation acts similarly to an attention mechanism, allowing the model to weigh and combine information sources in a flexible, context-sensitive way.



Convolutional Graph Neural Network

Now we want to introduce another important aspect: the analogy between classical convolutions and **graph convolutions**. In classical convolution, such as that used in CNNs, the operation consists of aggregating local information in a given area (e.g., a **filter**) and combining this information to obtain a new representation. Similarly, when we apply convolution to graphs, we are trying to obtain an aggregated representation of the nodes and their local relationships in the graph.



How can we implement these convolutions in graphs? One way to do this is to use **polynomial filters**. These filters are applied to the neighborhood of a node in the graph, similar to how localized filters in CNNs are applied to adjacent pixels in an image. The idea is to capture local information in the neighborhood of each node and combine this information to obtain a new representation of the node itself.

We begin by introducing the idea of constructing polynomial filters on the neighborhood of nodes. Later, we will see how more recent approaches extend this idea with more powerful mechanisms.

Polynomial Filters on Graphs

Given a graph G , let us fix an arbitrary ordering of the n nodes of G . Let A denote the **0-1 adjacency matrix** of G . Using this matrix, we can construct the **diagonal degree matrix D of G** , computed as follows:

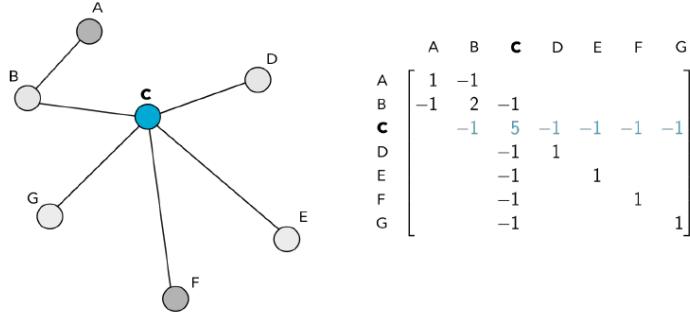
$$D_v = \sum_u A_{vu}$$

where A_{vu} indicates the element in the row corresponding to v and the column corresponding to u in matrix A . Each element of A indicates whether an edge exists between node v and node u . If an edge exists, the element is 1; otherwise, it is 0.

Note: The degree of a node v (i.e. D_v) represents the number of edges incident on that node, i.e., the number of nodes directly connected to v .

Then, the **graph Laplacian L** is defined as the $n \times n$ square matrix obtained by subtracting the adjacency matrix A from the diagonal degree matrix D :

$$L = D - A$$



The graph Laplacian gets its name from being the discrete analog of the [Laplacian operator](#) from calculus. It encodes the same information as the adjacency matrix A , but has unique properties. The graph Laplacian plays a fundamental role in various mathematical problems involving graphs, including random walks, spectral clustering and diffusion, to name a few.

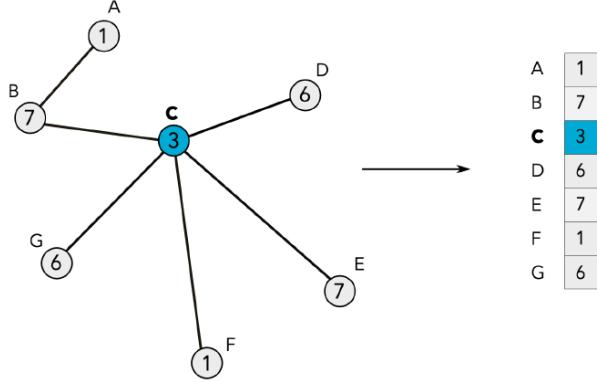
Now that we understand what the graph Laplacian is, we can build polynomials of the form:

$$p_w(L) = w_0 I_n + w_1 L + w_2 L^2 + \cdots + w_d L^d = \sum_{i=0}^d w_i L^i$$

Each polynomial of this form can alternately be represented by its vector of coefficients $w = [w_0, \dots, w_d]$. **Note that for every w , $p_w(L)$ is an $n \times n$ matrix, just like L .** These polynomials can be thought of as the equivalent of "filters" in CNNs, and the coefficients w as the weights of the "filters".

For ease of exposition, we will focus on the case where the nodes have one-dimensional features: each of the x_v for $v \in V$ is just a real number. [The same ideas apply when each of the \$x_v\$ is a higher-dimensional vector.](#)

Using the previously chosen ordering of the nodes, we can then stack all the node features x_v to get a feature vector $x \in \mathbb{R}^n$.



Once we have constructed the feature vector x , we can define its convolution with a polynomial filter p_w as:

$$x' = p_w(L)x$$

To understand how the coefficients w affect the convolution, let's consider the 'simplest' polynomial: when $w_0 = 1$ and all other coefficients are 0. In this case, x' is simply x :

$$x' = p_w(L)x = \sum_{i=0}^d w_i L^i x = w_0 I_n x = x$$

Now, if we increase the degree of the polynomial and consider the case where instead $w_1 = 1$ and all other coefficients are 0. Then x' is just Lx , and so:

$$\begin{aligned} x'_v &= (Lx)_v = L_v x \\ &= \sum_{u \in G} L_{vu} x_u \\ &= \sum_{u \in G} (D_{vu} - A_{vu}) x_u \\ &= D_v x_u - \sum_{u \in \mathcal{N}(v)} x_u \end{aligned}$$

We see that the features at each node v are combined with the features of its immediate neighbors $u \in \mathcal{N}(v)$. Thus, the resulting feature vector x' incorporates information from both the node itself ($D_v x_u$) and its adjacent nodes in the graph ($\sum_{u \in \mathcal{N}(v)} x_u$).

Note: For those familiar with Laplacian convolutions on images, this is exactly the same idea. When x is an image, $x' = p_w(L)x$ is exactly the result of applying a "Laplacian filter" to x .

At this point, a natural question to ask is: how does the degree d of the polynomial influence the behavior of the convolution? Indeed, it's not too difficult to show that:

$$\text{dist}_G(v, u) > 1 \Rightarrow L_{vu}^i = 0$$

that is, **the convolution will only be affected by nodes no further than d , so for greater values the Laplacian will have the value 0**. In fact, if we convolve x with $p_w(L)$ of degree d to obtain x' , we get:

$$x'_v = (p_w(L)x)_v = (p_w(L))_v x$$

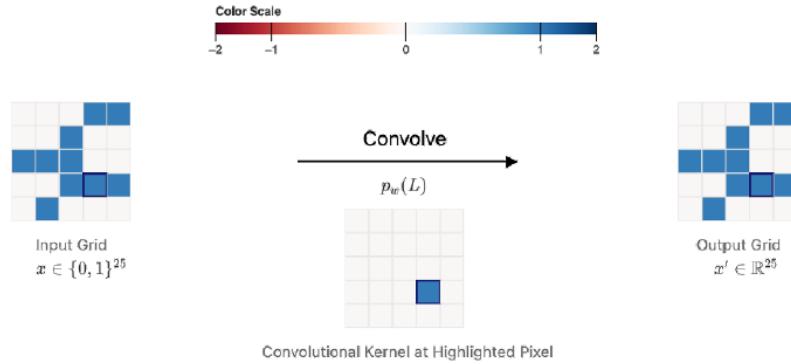
$$= \sum_{i=0}^d w_i L_v^i x$$

$$= \sum_{i=0}^d w_i \sum_{u \in G} L_{vu}^i x_u$$

$$= \sum_{i=0}^d w_i \sum_{\substack{u \in G \\ dist_G(u,v) \leq 1}} L_{vu}^i x_u$$

Effectively, **the convolution at node v only occurs with nodes u which are not more than d hops away**. Thus, these **polynomial filters are localized**. The degree of localization is completely governed by d : the higher the degree, the greater the number of neighboring nodes considered.

Below are several examples in which, given the input grid x , the polynomial coefficients are varied to see how the result x' of the convolution changes.



$$p_w(L) = \sum_{i=0}^2 w_i L^i = 1I + 0L + 0L^2.$$

w_0
1

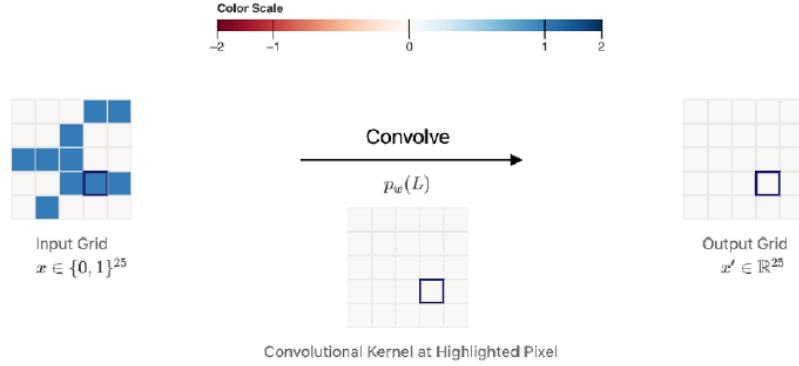
w_1
0

w_2
0

Choice of Laplacian

Unnormalized L Normalized \bar{L}

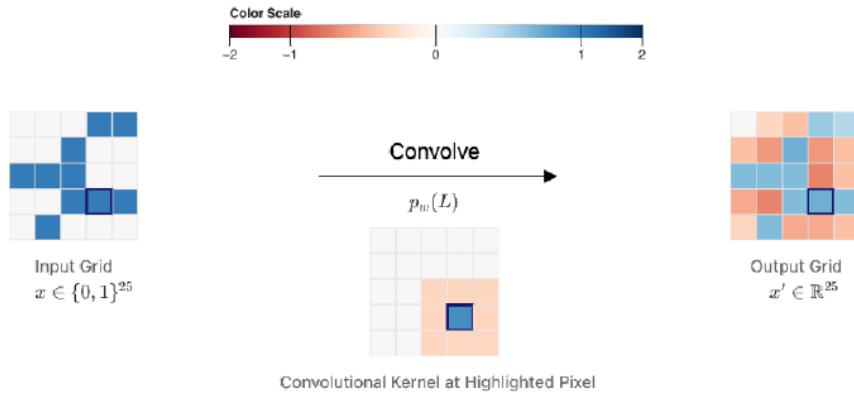
When $w_0 = 1$ and all other coefficients are zero $(1, 0, 0)$, this means that only the identity term I of the polynomial filter is present in the convolution operation. In this case, the convolution operation does not involve neighborhood information or movements beyond the current node. The resulting x' will be equal to the input x (as we mentioned previously).



Choice of Laplacian

Unnormalized L Normalized \tilde{L}

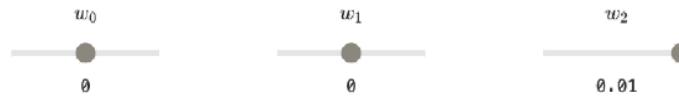
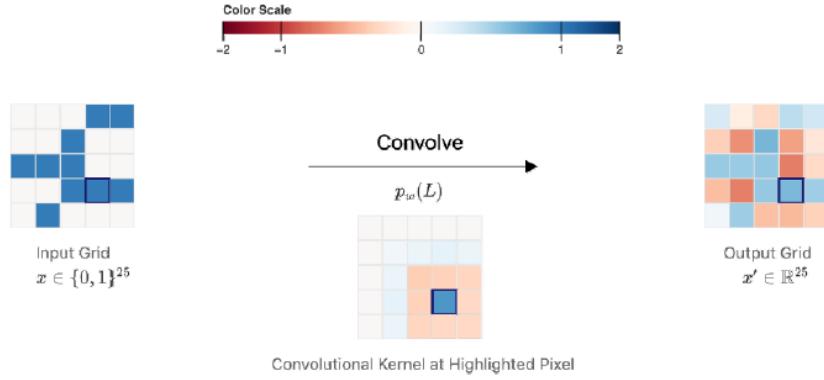
When all coefficients are zero $(0, 0, 0)$, this means that the polynomial filter has no influence. In this case, the convolution operation will result in x' being a zero vector, indicating that the features of all nodes are completely ignored.



Choice of Laplacian

Unnormalized L Normalized \tilde{L}

When $w_1 = 0.1$ and all other coefficients are zero $(0, 0.1, 0)$, it implies that the convolution operation considers just the nodes immediately adjacent to it given them an importance of 0.1. The resulting x' will incorporate information just from its neighbors with a weight of 0.1 assigned to their features. This indicates that the features of neighboring nodes have a moderate influence on the resulting features of the central node.

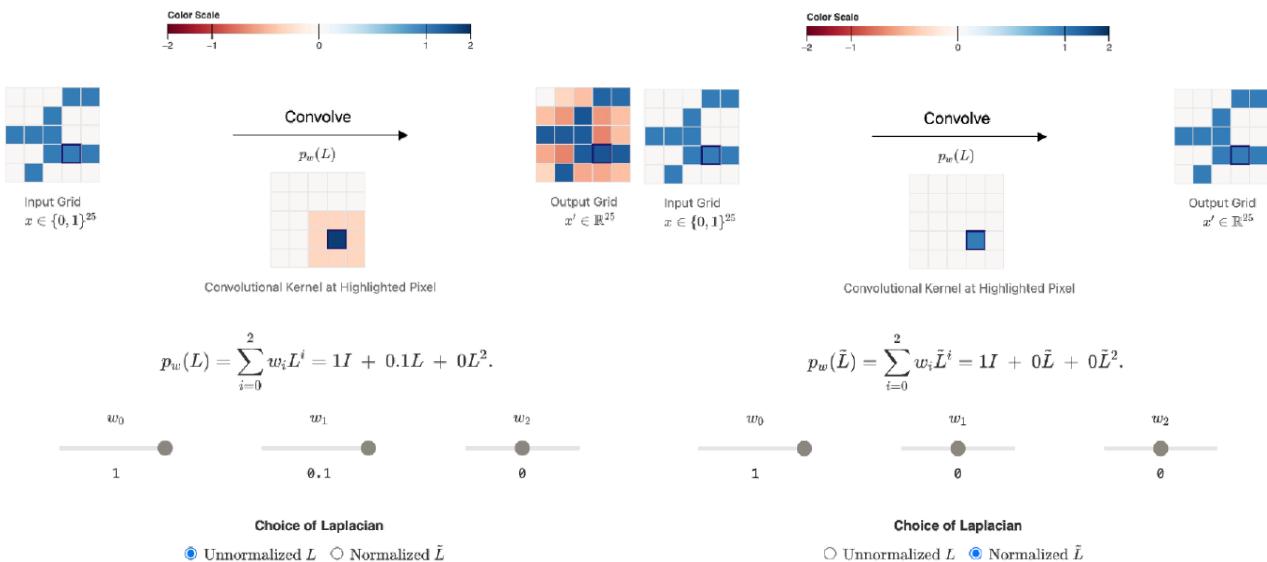


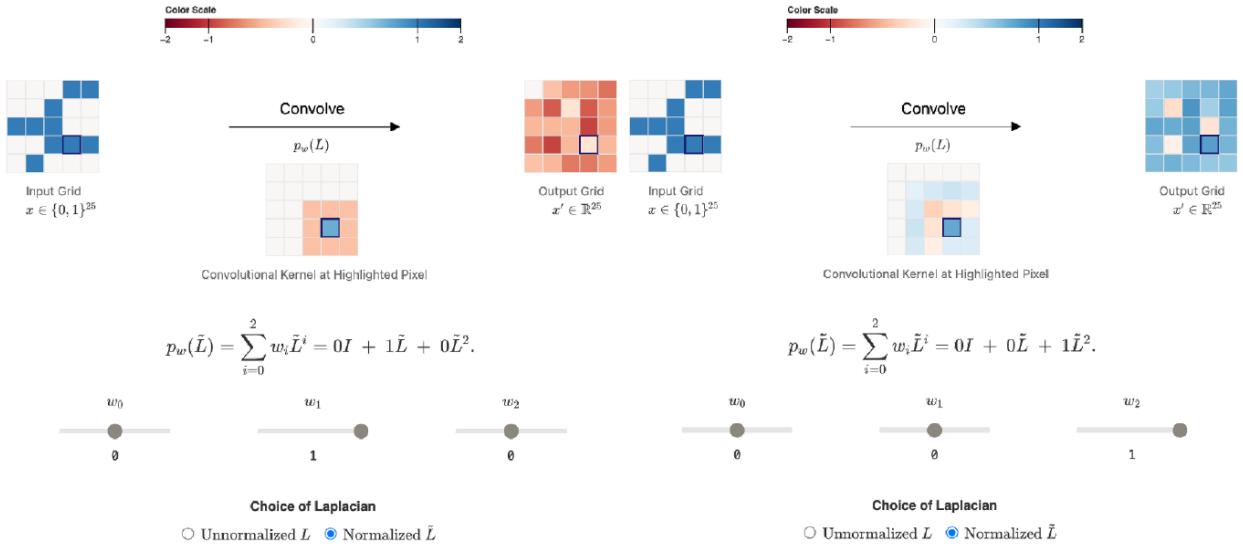
Choice of Laplacian

● Unnormalized L ○ Normalized \tilde{L}

When $w_2 = 0.01$ and all other coefficients are zero (0, 0, 0.01), the convolution operation extends its reach to nodes just beyond its immediate neighbors. The resulting x' will reflect a combination of the features 1 step away from its immediate neighbors. However, the weight assigned to the features of distant nodes is much smaller (0.01) indicating a lighter influence from these distant nodes, as indicated by the light blue portion in Figure.

By adjusting these coefficients, we can control the extent and intensity of the influence of neighboring nodes in the graph. More examples are given below.





ChebNet

A special case of GNN is ChebNet. **ChebNet** [81] refines this idea of polynomial filters by looking at polynomial filters of the form:

$$p_w(L) = \sum_{i=0}^d w_i T_i(\tilde{L})$$

where T_i is the degree- i [Chebyshev polynomial of the first kind](#) and \tilde{L} is the normalized Laplacian.

How can the Laplacian be normalized?

We can analyze the eigenvalues, take the maximum eigenvalue and normalize with respect to that:

$$\tilde{L} = \frac{2L}{\lambda_{max}(L)} - I_n$$

What is the motivation behind these choices?

- We know that the Laplacian L is *positive semi-definite*, meaning that all eigenvalues of L are always not lesser than 0. If $\lambda_{max}(L) > 1$, the entries in the powers of L can rapidly increase in size. However, if we normalize the Laplacian L , we obtain a \tilde{L} version that is effectively a scale-down version of L , with eigenvalues **guaranteed to be in the range $[-1, 1]$** . Therefore, in such a way we prevent the entries of powers of \tilde{L} from blowing up.
- The Chebyshev polynomials have certain interesting properties that make interpolation more numerically stable (we won't talk about this in more depth here).

Polynomial Filters are Node-Order Equivariant: The polynomial filters we considered here are actually independent of the order of the nodes. For example, if we consider the polynomial filter of degree $p_w = 1$, and therefore the first term of the Laplacian, we realize that this term involves a **sum** of the characteristics of the neighbors of each node and therefore the overall measure is equivariant with respect to the order of the nodes.

Of course, a similar reasoning can also be applied to polynomials of higher degree; the result is always the same: the entries of the powers of L are equivariant with respect to the order of the nodes.

Embedding Computation

To build a GCN, we can stack ChebNet (or other polynomial filter) layers one after the other along with nonlinearities, similar to what we do in traditional CNNs. In particular, if we have K different polynomial filter layers, each with its own learnable weights $w^{(k)}$, we will perform the following calculation:

Start with the original features.

$$\mathbf{h}^{(0)} = \mathbf{x}$$

Color Codes:

- Computed node embeddings.
- Learnable parameters.

Then iterate, for $k = 1, 2, \dots$ upto K :

$$p^{(k)} = p_{w^{(k)}}(L)$$

Compute the matrix $p^{(k)}$ as the polynomial defined by the filter weights $w^{(k)}$ evaluated at L .

$$g^{(k)} = p^{(k)} \times \mathbf{h}^{(k-1)}$$

Multiply $p^{(k)}$ with $\mathbf{h}^{(k-1)}$: a standard matrix-vector multiply operation.

$$\mathbf{h}^{(k)} = \sigma(g^{(k)})$$

Apply a non-linearity σ to $g^{(k)}$ to get $\mathbf{h}^{(k)}$.

We start with our input x , which we denote with $h^{(0)}$, and we calculate the Laplacian L . Then, for each of the K different polynomial filter layers, we combine the Laplacian with the weights $w^{(k)}$ to obtain the polynomial filter $p^{(k)}$. Once we have the filter, we convolve the current output $h^{(k-1)}$ and the considered polynomial filter $p^{(k)}$. We then apply a nonlinear activation function, obtaining the new output $h^{(k)}$.

It's important to note that **these networks reuse the same filter weights across different nodes**, exactly mimicking weight-sharing in CNNs which reuse weights for convolutional filters across a data grid.

The Shared Workings behind GNNs

ChebNet was a breakthrough in learning localized filters over graphs, and it motivated many to think of graph convolutions from a different perspective. Returning back to the result of convolving x with the polynomial kernel $p_w(L) = L$ and focusing on a particular node v we had seen that:

$$\begin{aligned} (Lx)_v &= L_v x \\ &= \sum_{u \in G} L_{vu} x_u \\ &= \sum_{u \in G} (D_{vu} - A_{vu}) x_u \\ &= D_v x_u - \sum_{u \in N(v)} x_u \end{aligned}$$

As we noted before, this is a 1-hop localized convolution, but more importantly, we can think of this convolution as arising of two steps:

- Aggregating over immediate neighbor features x_u .
- Combining them with the node's own feature x_v .

Indeed, if we think of a single layer of a standard GNN, we have precisely these phases:

1. **Node aggregation:** For each node i , the **aggregation function** $f^{(l)}$ combines the representations of the neighboring nodes j and the current node i . This can be denoted as:

$$m_i^{(l)} = \sum_{j \in \mathcal{N}(i)} f^{(l)}(h_j^{(l-1)}, h_i^{(l-1)}, e_{ij})$$

Here, $h_i^{(l-1)}$ represents the representation of node i at the previous layer, $h_j^{(l-1)}$ represents the representation of neighboring node j at the previous layer, e_{ij} indicates the edge between nodes i and j , and $f^{(l)}$ is a learnable function.

2. **Node updating:** The aggregation results are then used to update the representation of each node i :

$$h_i^{(l)} = g^{(l)}(h_i^{(l-1)}, m_i^{(l)})$$

Here, $g^{(l)}$ is the **update function**, a learnable function that updates the representation of node i based on its previous representation and the aggregated messages.

So, considering multiple layers, we have aggregation, update, aggregation, update and so on.

Of course, we can also have a **graph-level readout function**, which aggregates the node representations to obtain a graph-level representation ([the one we have called context vector](#)), if necessary.

Modern Graph Neural Networks

We've seen that at the core of GNNs lies a simple, but powerful idea: nodes update their representations by **aggregating and combining** information from their neighbors. This process, often referred to as **message passing**, defines the essential behavior of most GNNs. However, at this point, one might ask: "Can we design different types of aggregation and combination steps?".

Indeed, what we call "GNNs" today actually refers to a broad family of models that share this general message-passing paradigm but differ in how they implement the aggregation and update steps. Obviously, what we want for any GNN variant is that it must be node-order equivariant. By stacking multiple layers of message passing, where each layer aggregates information from 1-hop neighbors, GNNs gradually expand a node's "**receptive field**" — after K layers, a node can incorporate information from up to K -hop neighbors.

This iterative message-passing mechanism forms the backbone of many modern GNN architectures. In the following sections, we'll explore some of the most popular ones:

- **Graph Convolutional Network (GCN)**
- **Graph Attention Network (GAT)**
- **Graph Sample and Aggregate (GraphSAGE)**
- **Graph Isomorphism Network (GIN)**

GCN

We have the same starting point for all the different GNN variants we will see. In particular, at step 0 the initial embedding of each node v , denoted $h_v^{(0)}$, is simply set to its input features.

$\textcolor{orange}{h}_v^{(0)}$	$=$	$x_v \quad \text{for all } v \in V.$	Color Codes:  Embedding of node v .  Embedding of a neighbour of node v .  (Potentially) Learnable parameters.
Node v 's initial embedding.		... is just node v 's original features.	

Then, in a **Graph Convolutional Network (GCN)**, for each step k from 1 to K , we update embedding of each node v using the following two steps:

$$\textcolor{orange}{h}_v^{(k)} = \textcolor{teal}{f}^{(k)} \left(\textcolor{teal}{W}^{(k)} \cdot \frac{\sum_{u \in \mathcal{N}(v)} \textcolor{purple}{h}_u^{(k-1)}}{|\mathcal{N}(v)|} + \textcolor{teal}{B}^{(k)} \cdot \textcolor{orange}{h}_v^{(k-1)} \right) \quad \text{for all } v \in V.$$

$\textcolor{orange}{h}_v^{(k)}$ Node v 's embedding at step k .	$\sum_{u \in \mathcal{N}(v)} \textcolor{purple}{h}_u^{(k-1)}$ Mean of v 's neighbour's embeddings at step $k - 1$.	$\textcolor{orange}{h}_v^{(k-1)}$ Node v 's embedding at step $k - 1$.
--	---	--

Basically:

- **Node Aggregation:** Sum the embeddings of all neighboring nodes $u \in \mathcal{N}(v)$ from the previous step $k - 1$, normalize this sum and multiply it by learnable weight matrix $W^{(k)}$. We also add the embedding of node v at step $k - 1$ multiplied by another learned weight matrix $B^{(k)}$.
- **Node Update:** We then apply a non-linear activation function $f^{(k)}$ to the result of the aggregation step, producing the new embedding at the step k .

It's important to note that for each layer k , the same activation function $f^{(k)}$ and weight matrices $W^{(k)}, B^{(k)}$ are **shared across all nodes** in the graph. This ensures that the number of learnable parameters involved in the GCN model is not tied to the size of the graph, allowing the model to scale effectively. Therefore, even if $f^{(k)}, W^{(k)}, B^{(k)}$ become quite large, this is not a problem.

Moreover, in the formula above, we applied a **normalization factor**:

$$f \left(W \sum_{u \in \mathcal{N}(v)} \frac{h_u}{|\mathcal{N}(v)|} + B \cdot h_v \right)$$

where $|\mathcal{N}(v)|$ denotes the number of nodes in the neighborhood of node v .

Without such normalization, a raw summation would give disproportionately large influence to nodes with many neighbors compared to those with few. This imbalance can cause numerical instability, particularly in domains like social networks, where neighborhood sizes can differ by several orders of magnitude. Therefore, normalization is introduced to balance the contributions from neighbors and prevent high-degree nodes from dominating the aggregation.

In reality, in the original paper [82], the normalization is made more sophisticated: it accounts not only for the degree of the target node v , but also for the degree of each of its neighbors u :

$$f \left(W \sum_{u \in \mathcal{N}(v)} \frac{h_u}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}} + B \cdot h_v \right)$$

This means that the scaling depends both on how many neighbors v has and on how many neighbors each of those neighbors has in turn. In other words, the normalization takes into account not just v 's immediate neighborhood, but also the connectivity of its neighbors — effectively considering the neighbors of neighbors.

The motivation is that if a neighbor u itself has many connections, its individual contribution to v 's update should be down-weighted, since the information from u is being spread across many other nodes. Conversely, neighbors with fewer connections can exert relatively stronger influence.

After K layers of message passing, we obtain the final node embeddings $h_v^{(K)}$, which can be used for downstream tasks such as **node classification**. Typically, a prediction layer is applied to each final node embedding:

$$\hat{y}_v = \text{PREDICT}(h_v^{(K)})$$

where PREDICT is another neural network that is learned alongside the GCN model.

GAT

Graph Attention Networks (GATs), introduced by Veličković et al. (2017) [83], extend GCNs by incorporating an attention mechanism that learns to weight the importance of neighboring nodes adaptively, rather than treating all neighbors equally

Here too, the initial embedding of each node v is simply set to its input features

$$h_v^{(0)} = x_v \quad \text{for all } v \in V.$$

Node v 's
initial
embedding.

... is just node v 's
original features.

Then, for each step k from 1 to K , we update the embedding of node v as follows:

$$h_v^{(k)} = f^{(k)} \left(W^{(k)} \cdot \left[\sum_{u \in \mathcal{N}(v)} \alpha_{vu}^{(k-1)} h_u^{(k-1)} + \alpha_{vv}^{(k-1)} h_v^{(k-1)} \right] \right) \quad \text{for all } v \in V.$$

Node v 's embedding at step k .	Weighted mean of v 's neighbour's embeddings at step $k - 1$.	Node v 's embedding at step $k - 1$.
---	---	---

where, in addition, we have $\alpha^{(k)}$, which represent the **attention weights** generated by an attention mechanism $A^{(k)}$, normalized such that the sum over all neighboring nodes of each node v is 1:

$$\alpha_{vu}^{(k)} = \frac{\mathbf{A}^{(k)}(\mathbf{h}_v^{(k)}, \mathbf{h}_u^{(k)})}{\sum_{w \in \mathcal{N}(v)} \mathbf{A}^{(k)}(\mathbf{h}_v^{(k)}, \mathbf{h}_w^{(k)})} \quad \text{for all } (v, u) \in E.$$

Note: The attention network $\mathbf{A}^{(k)}$ returns an attention matrix (a matrix of weights). From this matrix, we select only a single normalized element.

Here too, predictions can be made at each node by using the final computed embedding $\mathbf{h}_v^{(K)}$:

$$\hat{y}_v = \text{PREDICT}(\mathbf{h}_v^{(K)})$$

where PREDICT is generally another neural network, learned alongside the GAT model.

In the GAT model too, the function $f^{(k)}$, the weight matrices $\mathbf{W}^{(k)}$ and additionally the attention mechanism $\mathbf{A}^{(k)}$ are shared across all nodes for every step k . This allows the model to scale well, since the number of parameters does not depend on the size of the graph.

Note: It is important to note that here we have discussed a variant of the GAT that adopts a **single-head attention**. The multi-head attention variant follows a similar principle. Furthermore, the choice of attention mechanism can vary depending on specific needs.

GraphSAGE

Graph Sample and Aggregate (GraphSAGE), introduced by Hamilton et al. (2017) [84], is an inductive framework for generating node embeddings that scales to large graphs (e.g. social networks) by sampling a fixed-size neighborhood and aggregating their features rather than relying on the entire graph structure.

Here too, the initial embedding of a node v at step 0 simply corresponds to its original features.

$$\mathbf{h}_v^{(0)} = \mathbf{x}_v \quad \text{for all } v \in V.$$

Node v 's	... is just node v 's
initial	original features.
embedding.	

Then, for each step k from 1 to K , we updated the embedding of node v as follows:

$$\mathbf{h}_v^{(k)} = f^{(k)} \left(\mathbf{W}^{(k)} \cdot \left[\mathbf{AGG}_{u \in \mathcal{N}(v)} (\{\mathbf{h}_u^{(k-1)}\}), \mathbf{h}_v^{(k-1)} \right] \right) \quad \text{for all } v \in V.$$

Node v 's	Aggregation of	... Node v 's
embedding at	v 's neighbour's	embedding at
step k .	embeddings at	step $k - 1$.
	step $k - 1$...	
	... concatenated	
	with ...	

Here, specifically, the embeddings of all neighboring nodes $u \in \mathcal{N}(v)$ from the previous step $k - 1$ are aggregated using the \mathbf{AGG} (aggregation) function. The key idea is to **learn node embeddings by sampling and aggregating**

features from a node's local neighborhood where the AGG function determines **how** the information from neighboring nodes is combined at each step.

There are several ways to aggregate neighbor embeddings; in particular, the original GraphSAGE paper considers the following choices for the AGG function:

- **Mean** (similar to GCN): The embedding is aggregated using an average of the neighbors:

$$W_{\text{pool}}^{(k)} \cdot \frac{\mathbf{h}_v^{(k-1)} + \sum_{u \in \mathcal{N}(v)} \mathbf{h}_u^{(k-1)}}{1 + |\mathcal{N}(v)|}$$

- **Dimension-wise Maximum**: For each dimension of the embedding, takes the **maximum value** across all neighbors:

$$\max_{u \in \mathcal{N}(v)} \{\sigma(W_{\text{pool}}^{(k)} \mathbf{h}_u^{(k-1)} + b)\}$$

Notice that both mean and max aggregations are permutation-invariant, being operation independent of the neighbors order.

After aggregation, GraphSAGE combines the aggregated neighbor embedding with the node's own embedding (usually via concatenation), applies a nonlinearity and proceeds to the next layer.

Here, too, predictions can be made at each node by using the final computed embedding $\mathbf{h}_v^{(K)}$:

$$\hat{y}_v = \text{PREDICT}(\mathbf{h}_v^{(K)})$$

where $PREDICT$ is typically another neural network, learned alongside the GraphSAGE model.

Here too, for each step k , the functions $f^{(k)}$, AGG and the matrix $W^{(k)}$ are shared across all nodes. This allows the GraphSAGE model to scale well, since the number of parameters does not depend on the size of the graph.

As we started to anticipate, GraphSAGE performs '**neighbourhood sampling**': irrespective of how large the actual neighborhood around a node is, a fixed-size random sample of the neighborhood is taken.

This choice is motivated by the fact that, as mentioned previously, GraphSAGE is also used for large social networks. For example, considering the case of Wikipedia, we saw that the average number of edges per node was 60, but the maximum could be as high as 1 million. Therefore, if we were to tackle a node with 1 million neighbors, the computation would take a long time. The idea of taking just a fixed-size random sample of the neighbors, such as 150 or 50, allows us to handle extremely large graphs. Obviously, this increases the variance in the computed embeddings but allows GraphSAGE to be used on very large graphs.

GIN

The **Graph Isomorphism Network (GIN)**, introduced by Xu et al. (2018) [85], is a GNN introduced to address a fundamental question in graph learning:

“How expressive can a GNN be in distinguishing between different graph structures?”

To understand this, we need to consider the concept of **graph isomorphism**.

Two graphs G_1 and G_2 are said to be **isomorphic** if they are structurally identical, meaning there exists a **one-to-one mapping between their nodes** that preserves all the edges.

In other words, if you can **relabel** the nodes of one graph and make it look exactly like the other, then they are isomorphic. Formally, there exists a bijection $f: V_1 \rightarrow V_2$ such that:

$$(u, v) \in G_1 \Leftrightarrow (f(u), f(v)) \in G_2$$

Conversely, if two graphs are **non-isomorphic**, they have **different structures** and cannot be matched through any node relabeling.

Why is this Important for GNNs?

In graph learning, we often want to make predictions based on graph's structure. If a GNN **cannot distinguish two non-isomorphic graphs**, it may assign them the same representation, leading to **incorrect or ambiguous predictions**. However, many existing GNN architectures (like basic GCNs or GraphSAGE) use **aggregation functions** (like mean or max) that can cause **different structures to collapse into identical representations** — especially when node features are similar or absent. This lack of expressive power limits their ability to fully capture the richness of graph topology.

GIN was proposed as a solution to this limitation. Its architecture was carefully designed to match the **discriminative power of the Weisfeiler-Lehman (WL) test**, a classical algorithm used to test graph isomorphism. The WL test works by repeatedly updating node labels based on the multiset of neighboring labels. If, after several iterations, two graphs still look the same under this process, the test cannot distinguish them. However, if the labels diverge, the test recognizes that graphs are definitely non-isomorphic. GIN mimics this behavior by:

- Using **sum aggregation** (which is injective over multisets),
- Incorporating a **learnable self-node weighting** term,
- Applying a **nonlinear MLP** to allow more complex transformations.

This gives GIN **maximal discriminative power** among GNNs using neighborhood aggregation and, in theory, it can **distinguish any pair of graphs that the WL test can**.

Here too, the initial embedding of a node v at step 0 simply corresponds to its original features.

$$h_v^{(0)} = x_v \quad \text{for all } v \in V.$$

Node v 's
initial
embedding.

... is just node v 's
original features.

For each step k from 1 to K , we update the embedding of node v as follows:

$$h_v^{(k)} = f^{(k)} \left(\sum_{u \in \mathcal{N}(v)} h_u^{(k-1)} + (1 + \epsilon^{(k)}) \cdot h_v^{(k-1)} \right) \quad \text{for all } v \in V.$$

Node v 's embedding at step k .	Sum of v 's neighbour's embeddings at step $k - 1$.	Node v 's embedding at step $k - 1$.
---	---	---

In this case:

- The aggregated neighbor embedding is obtained via a **sum**. The authors found that *sum* is strictly more expressive in distinguishing graph structures, avoiding information loss that may occur with mean or max pooling, making GIN more **injective** in theory.
- We have a term $1 + \epsilon^{(k)}$ that multiplies the node v 's embedding. In particular, here ϵ is a learnable parameter that weights the importance of node v 's own features compared to its neighbors. It has the same importance if $\epsilon = 0$.
- Here, the $f^{(k)}$ is implemented as a **multi-layer perceptron (MLP)** followed by a non-linearity to allow for complex, learned transformations. According to the authors, one layer is **not sufficient** for graph learning in general.

Here too, predictions can be made on each node using the computed final embedding $h_v^{(K)}$:

$$\hat{y}_v = \text{PREDICT}(\mathbf{h}_v^{(K)})$$

where *PREDICT* is generally another neural network, learned alongside the GIN model.

Here too, for each step k , the function $f^{(k)}$ and the real-valued parameter $\epsilon^{(k)}$ are shared across all nodes. This allows the GIN model to scale well, since the number of model parameters does not depend on the size of the graph.

Autoencoders

During the ML Course, we introduced *Autoencoders* as models designed for **dimensionality reduction** [86]. Here, we revisit them with a new perspective, focusing on their deeper connection to data representation. Let's begin by recalling how a basic autoencoder works.

Standard Autoencoder

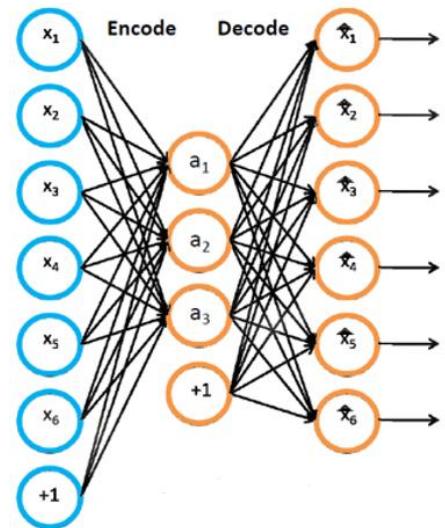
An **Autoencoder (AE)** consists of a neural network having the **same number of output units as inputs** and made of two main components:

- an **encoder**, which deterministically maps the input x into a representation z :
$$z = f(Wx + b)$$
- a **decoder**, which deterministically maps this representation back to a reconstruction y , which has the same shape as the original input:
$$y = g(W'z + b')$$

The network is trained to make the output y as close as possible to the input x :

$$y \approx x$$

However, for the autoencoder to learn **non-trivial representations**, **some form of constraint** must be imposed, otherwise the network could simply learn to copy the input directly to the output without extracting meaningful structure (i.e. learn the identity mapping).



A constraint could be to make the dimensionality of the representation z smaller than that of the input x , thereby forcing the model to learn a representation of the input that retains only the most salient features. This is the core idea behind the **standard autoencoder** (see Figure). Later, we'll explore other types of constraints, such as sparsity or noise.

So, autoencoders learn to encode input data x into a lower-dimensional representation z and then decode this representation outputting a reconstruction y that should be as close as possible to the original input x . Since the number of hidden units is smaller than the number of inputs, a perfect reconstruction of all input vectors is not in general possible. We therefore determine the network parameters (W and b) by minimizing an error function that captures the degree of mismatch between the input vectors and their reconstructions. This is known as a **reconstruction error** and is generally computed as a *sum-of-squares*:

$$\min_{W,b} \left(\frac{1}{2} \sum_i \|x^{(i)} - y(x^{(i)}; W, b)\|^2 \right)$$

Importantly, autoencoders operate in an **unsupervised learning** setting — we don't require any labels. In fact, the network is trained using only the inputs x .

Since the latent representation z is not directly observable, it is often referred to as a **hidden representation** or **latent variable**. This concept is deeply connected to the **manifold hypothesis**, which we recall suggests that high-dimensional real-world data (such as images, speech or text) lies near a much lower-dimensional, non-linear

manifold embedded in the high-dimensional input space. This manifold is shaped by a few meaningful factors of variation — for instance, lighting, pose or object identity in images.

Autoencoders are designed to **learn a parameterization of this manifold**. By compressing the input into a low-dimensional latent space, the encoder **captures the directions in data space that correspond to meaningful variations, while ignoring irrelevant or redundant directions where the data shows little or no variation**. The decoder then learns to map these compact latent codes back into the original input space, effectively modeling how the data is structured.

Why Reconstruct the Input? We've seen above that an autoencoder is trained to reconstruct its input, but why is this useful? What's the practical motivation behind learning to replicate the input? The **reconstruction task** serves as a **proxy objective**: it provides a training signal that forces the network to **learn a meaningful latent representation**. The idea is that, if the model can successfully reconstruct the input from a compressed code z , then z must capture the most relevant and informative features of the data.

This learned representation can then be used for a variety of downstream tasks. For instance, in an image classification task, instead of feeding the full image into a classifier, we can use the encoder to extract a low-dimensional feature vector and pass it to a classification head. This not only reduces computational complexity but often improves performance by filtering out irrelevant noise.

In short, the **goal of an autoencoder is not merely to copy the input**, but to learn a **compact, structured representation** of the data that preserves its essential information. Reconstruction is just the tool that enables this learning.

Sparse Autoencoders

Instead of limiting the number of units in the hidden layer, an alternative way to constrain the internal representation is to **use a regularizer to encourage a sparse representation**, leading to a lower effective dimensionality. That is the idea behind **Sparse autoencoders** [87]. A simple choice is the **l_1 regularizer** since this encourages sparseness, giving a regularized error function of the form:

$$\min_{w,b} \left(\frac{1}{2} \sum_i \|x^{(i)} - y(x^{(i)}; W, b)\|^2 + \lambda \sum_j |a_j| \right)$$

Here, a_j denotes the activation of the j -th neuron and λ is a regularization coefficient controlling the strength of the sparsity constraint. Note that, unlike traditional regularization which penalizes the weights of the network, this formulation **penalizes the activations**. This encourages the network to use only a small number of neurons' activations for any given input, leading to **sparse representations**.

Denoising Autoencoders

Another constraint that forces the model to discover interesting internal structure in the data is to use a **Denoising Autoencoder** [88]. The idea is to take each input vector $x^{(i)}$ and to **corrupt it with noise** to give a modified vector $\tilde{x}^{(i)}$ which is then fed as input to an autoencoder to give an output $y(\tilde{x}^{(i)}; W, b)$. The network is **trained to reconstruct the original noise-free input vector** by minimizing the reconstruction error with respect to the **original clean input**:

$$\min_{w,b} \left(\frac{1}{2} \sum_i \|x^{(i)} - y(\tilde{x}^{(i)}; W, b)\|^2 \right)$$

A common type of corruption is to add **zero-mean Gaussian noise** to each input feature independently, with the noise level controlled by the variance. Other types of noise may include masking random input values or applying salt-and-pepper noise. By learning to denoise the input data, the autoencoder is forced to learn **robust, underlying features** of the data. For instance, in image data, the network can learn that nearby pixel intensities tend to be correlated, allowing it to infer and correct noise-corrupted pixels.

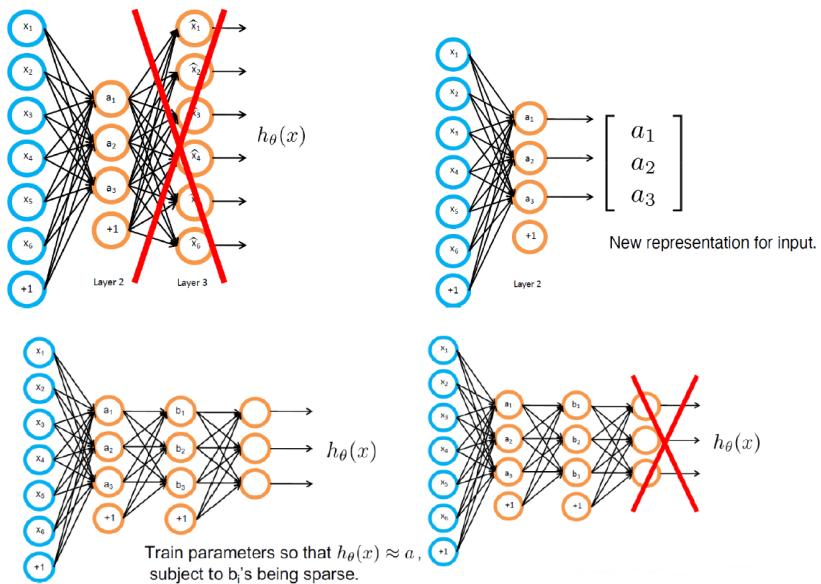
Deep Autoencoder

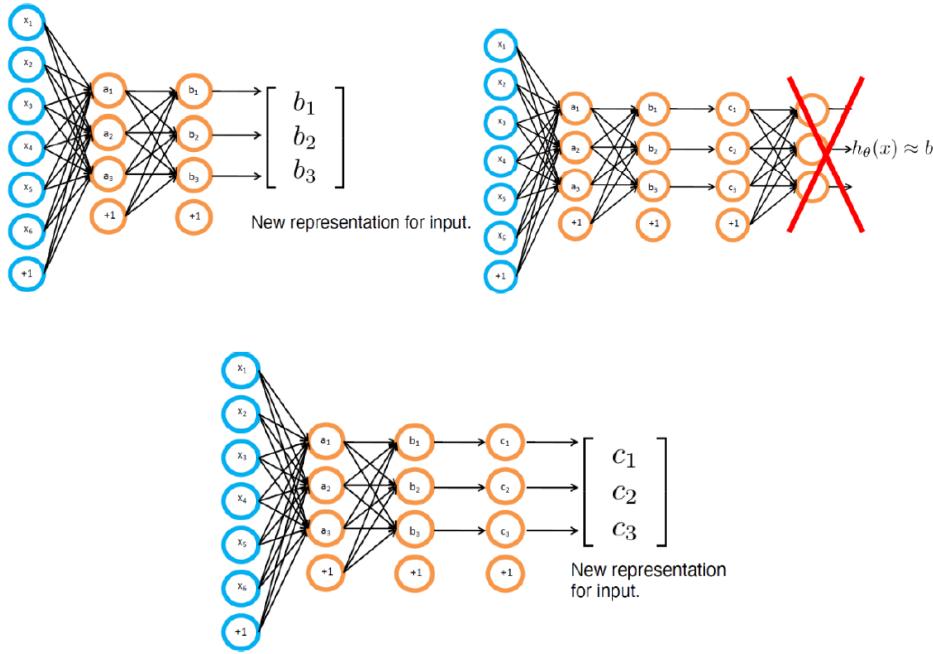
A simple autoencoder with just one hidden layer ([shallow net](#)) can only capture basic patterns in the data. To model more complex structures, we can **stack multiple layers** in both the encoder and decoder, forming a **deep autoencoder**. In this setup, the encoder progressively reduces the dimensionality of the input through successive layers, while the decoder symmetrically reconstructs the data back from the latent representation.

This deeper structure allows the network to learn **more complex, hierarchical and abstract representations** of the data. Each layer transforms the data into a more refined representation improving the quality of the final latent representation ([similar to what we observed in CNNs for image data](#)). Non-linear activation functions, such as ReLU, are typically used to enable the network to capture non-linear dependencies in the data.

Nowadays, training autoencoders end-to-end is straightforward, however, this wasn't always the case. In the early days of DL, training deep autoencoders was much more challenging. Many of the techniques we now take for granted—such as ReLU activations, proper weight initialization methods like Xavier or He initialization and robust optimizers like Adam—**had not yet been developed**.

At that time, increasing network depth often led to severe **vanishing gradient problems**, making it difficult to train deep networks effectively using standard backpropagation. To address this, researchers adopted a technique known as *layerwise training*. The core idea behind **layerwise training** [89] was to build the network **one layer at a time**. Each pair of adjacent layers was treated as a shallow autoencoder and trained independently. After training one layer, a new layer would be added on top, using the learned representations from the previous layer as input. This process was repeated, gradually stacking layers and building a deeper network.

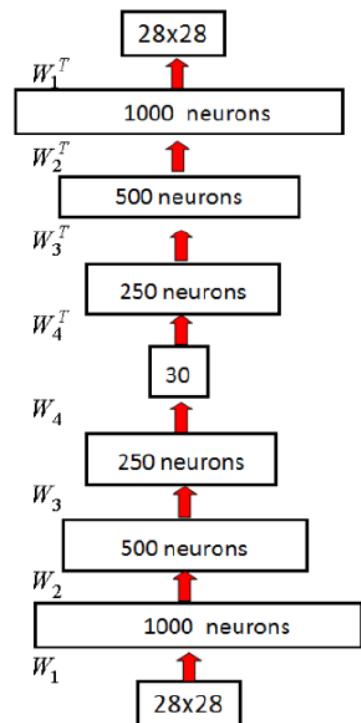




The key benefit of the **layerwise training approach** was that the **lower layers were trained in isolation**, before the full network depth introduced vanishing gradient issues. This meant that their weights were initialized to **meaningful values**, having already learned useful representations from the data. Later, when additional layers were added and gradients started to vanish during backpropagation, the **early layers still retained their effectiveness** — even if they received **little to no gradient updates**. In other words, because those layers were already performing well, they didn't require significant updates to contribute meaningfully to the overall network's performance. This made layerwise training a **smart workaround** for mitigating the vanishing gradient problem. That said, although layerwise pretraining was an important innovation at the time, it is now **largely obsolete**. With the development of improved initialization schemes, activation functions and optimization algorithms, **end-to-end training** has become the standard and more effective approach. For those interested in exploring this historical approach further, I recommend reviewing the "Autoencoders" section in the [ML Course notes](#).

In the Figure on side, we show a deep autoencoder applied to image data. Here, a 28×28 image is passed through several encoder layers, ultimately producing a latent representation consisting of only 30 features.

The use of multiple layers allows the network to learn a latent representation able to capture a rich, hierarchical encoding of the input data.



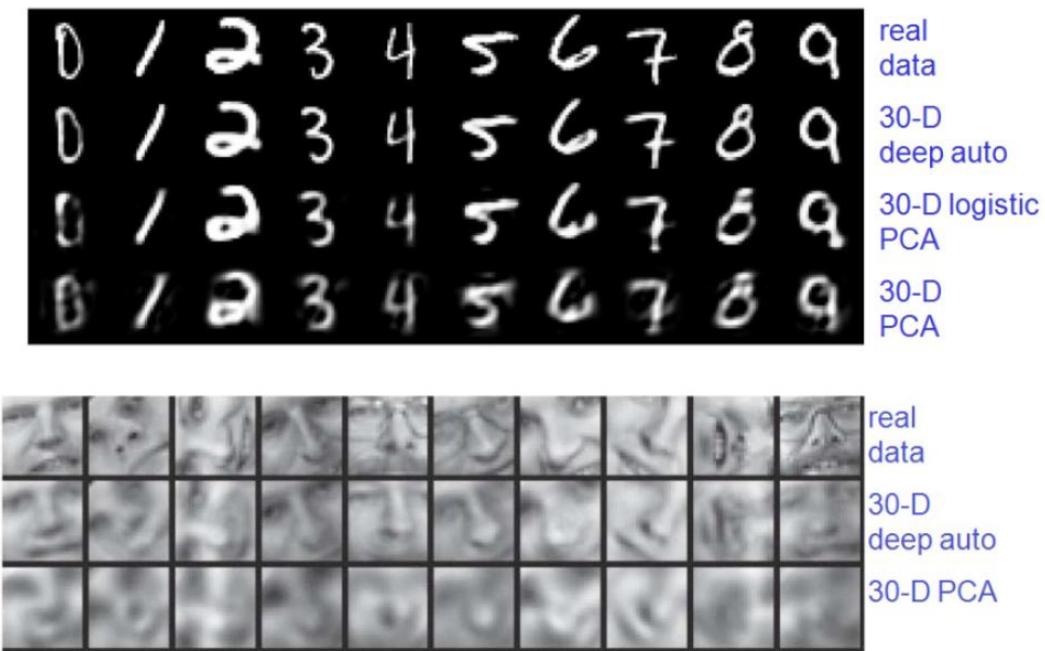
Deep Autoencoder vs PCA

Multiple layers with non-linear activations allow the deep autoencoders to **model complex, non-linear relationships** in the data. This makes them significantly more powerful than a single-layer (**linear**) autoencoder, which — in the absence of non-linear activation functions — is essentially just performing PCA. In fact, a single-layer autoencoder with purely linear activations learns the **same subspace** as PCA and thus performs **linear dimensionality reduction**.

While **Kernel PCA** extends PCA to non-linear settings by projecting data into a high-dimensional space using a **predefined kernel**, autoencoders offer a more **flexible and adaptive alternative**. Instead of relying on a fixed kernel function, autoencoders **learn the non-linear transformation directly from the data**, enabling them to better tailor the representation to the task at hand.

The Figure below [86] compares image reconstructions obtained using three methods: a **deep autoencoder**, **logistic PCA** and **standard PCA**, all with the same latent dimensionality (30). For simple datasets like MNIST, even linear methods like PCA can perform adequately. However, for **more complex datasets**, such as human face images, PCA reconstructions tend to appear **blurry and lack fine details** — often missing critical features like eyes, glasses or facial contours.

In contrast, **deep autoencoders** excel at capturing the hierarchical and abstract features of complex data, leading to **more accurate and visually detailed reconstructions**. This means that the autoencoder has learned a **richer and more meaningful latent space** — one that better preserves the essential features of the input.



To conclude, the ability to learn **task-specific, non-linear embeddings** makes autoencoders particularly powerful tools for modern representation learning.

Generative Adversarial Networks

Generative modeling in machine learning refers to the task of learning a probability distribution from observed data (e.g. training set) and then generating new samples that are likely to come from that same distribution. For instance, a generative model trained on images of animals could synthesize new, realistic-looking images of animals — even ones it has never encountered before.

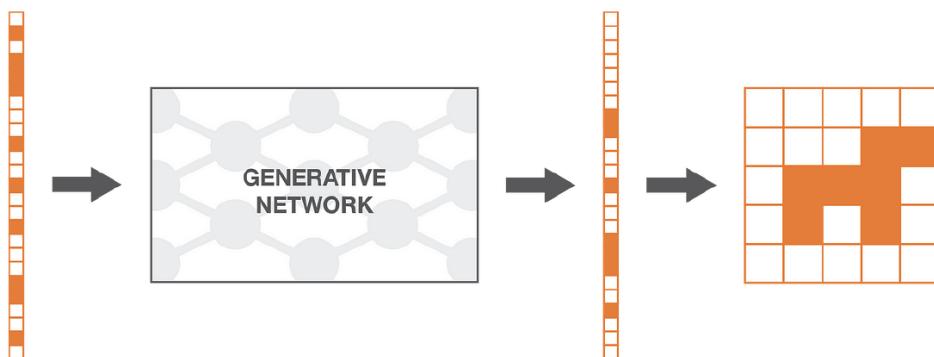
We have already seen an important family of deep generative models: autoregressive LLM's based on Transformers (e.g. GPT-3). In the next chapters, we will focus on other four major types of deep generative models: **Generative Adversarial Networks (GANs)**, **Variational Autoencoders (VAEs)**, **Normalizing Flows** and **Diffusion Models**.

Before diving into GANs, it is helpful to revisit a foundational concept shared by many generative models: **how to generate complex random variables from simpler, tractable ones**.

Generative Models

Imagine we have a dataset of real grayscale images of dogs, each composed of $n \times n$ pixels. We can represent each image as a vector in \mathbb{R}^N , where $N = n \times n$, by flattening the image — for example, stacking its columns into a single vector. However, not every point in this high-dimensional space corresponds to something that looks like a dog. In fact, the vast majority of vectors, if interpreted as images, will just appear as random noise. The ones that do resemble actual dogs form a small, highly structured region in this space and they are governed by a specific, unknown probability distribution. This is what we refer to as the **true data distribution**: the distribution that naturally generates the real dog images we observe in our dataset.

The objective of a generative model is to learn this true data distribution so that it can generate new, synthetic images that are indistinguishable from real ones. However, this distribution is extremely complex and cannot be written down analytically or sampled from directly. What we can do, instead, is start with a much simpler, known distribution — such as a **standard Gaussian or uniform distribution** — from which sampling is easy and well understood. Then, the task becomes learning a transformation that maps points from this simple distribution into the structured region of the space where realistic images live. If this transformation is successful, the distribution of these new outputs — called the **generated distribution** — will closely match the true data distribution of real data (e.g. dogs).



Input random variable
(drawn from a simple
distribution, for
example uniform).

The generative network
transforms the simple
random variable into
a more complex one.

Output random variable
(should follow the targeted
distribution, after training
the generative network).

The output of the
generative network
once reshaped.

This idea — learning a function that transforms a simple input distribution into a complex output distribution — lies at the heart of modern generative modeling. Since the transformation is typically non-linear, high-dimensional

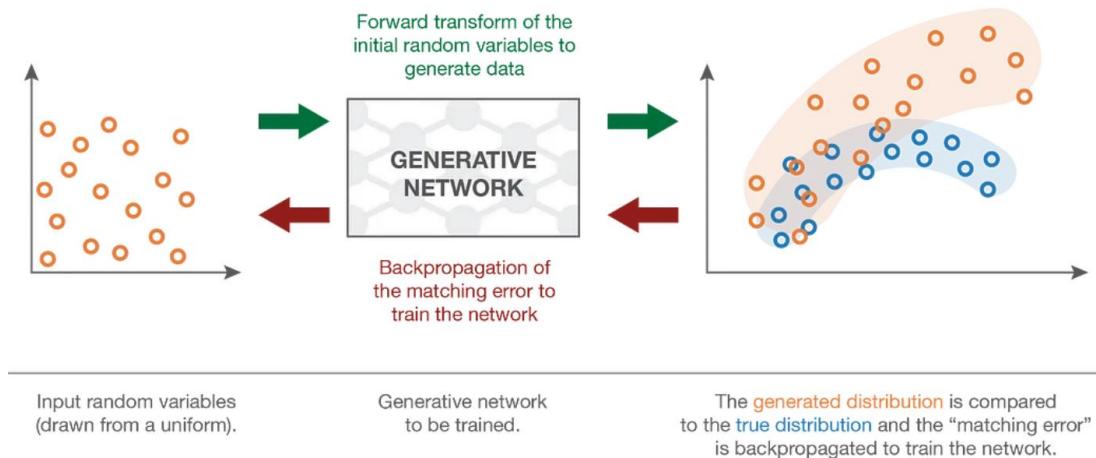
and intractable in closed form an idea could be to **learn it from the data**. As is the case in most cases, a very complex function naturally implies modeling with a **neural network**. Therefore, the idea is to use a neural network to learn and model the transformation function.

However, modeling in full high-dimensional data space (e.g., pixel space) is both inefficient and not useful. What we instead want is to deal with a much **lower-dimensional, non-linear manifold** able to capture the most meaningful factors of variation. To model this effectively, we introduce a **lower-dimensional latent space \mathbb{R}^d** , where $d \ll N$, and define a **simple prior distribution** over this space, commonly a standard Gaussian. We then sample a **latent vectors z** from this prior and use a neural network to **map these latent vectors to the high-dimensional data space \mathbb{R}^N** . In doing so, the network learns a transformation that reshapes the simple prior into a complex distribution over realistic data. For instance, if trained successfully on dog images, the network will follow the correct "dog probability distribution" (see Figure above), producing new synthetic dog images that are visually indistinguishable from real data.

Now that we've defined this architecture, we need to train it. One intuitive way to do this is by directly comparing the generated distribution to the true data distribution. This approach, which we refer to as the "**direct method**", proceeds as follows:

1. begin by generating a batch of random vectors sampled from the known simple distribution
2. feed them through our generative neural network to produce a batch of synthetic dog images
3. take a batch of real dog images from our dataset and then compare the two batches — the generated one and the real one — using a statistical measure of distributional distance (i.e. KL Divergence or total variation).
4. use backpropagation to make one step of gradient descent to lower the distance between true and generated distributions and update the parameters of the network accordingly.

We repeat these steps iteratively during the training process.



The Idea behind GAN – Indirect Approach

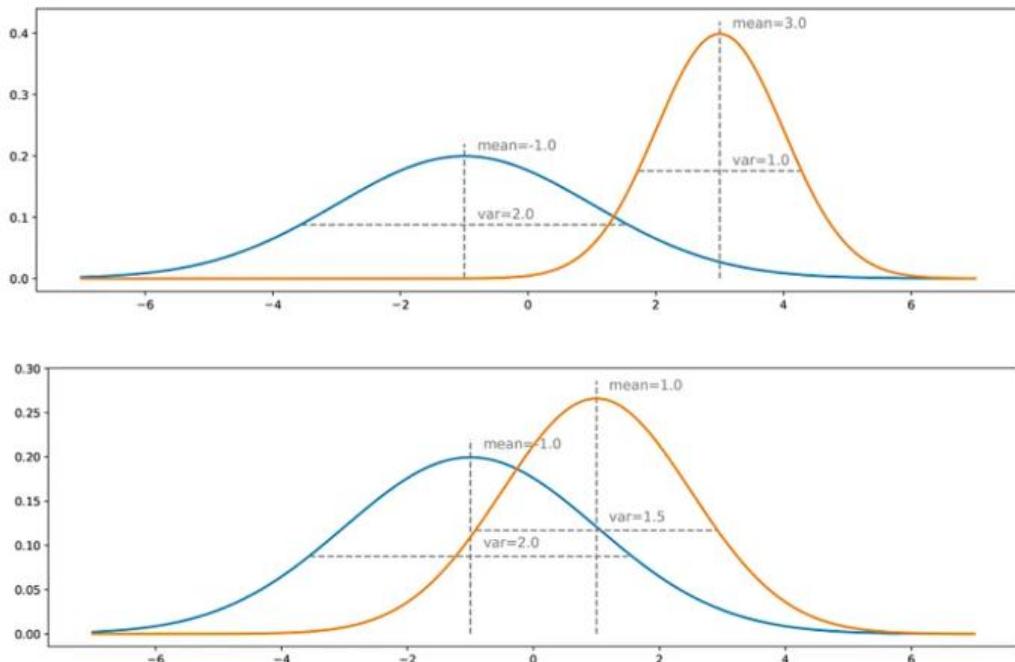
The "direct" training approach described above — aligning the generated distribution with the true data distribution via an explicit distance metric — is conceptually intuitive. However, in practice, **comparing high-dimensional distributions** is both computationally intensive and statistically fragile. This is especially true for image data, where distances like the KL divergence or total variation can be difficult to estimate accurately and efficiently.

Generative Adversarial Networks (GANs), introduced by Ian Goodfellow et al. in 2014 [90], provide an elegant "**indirect**" solution to this problem. Rather than computing a distance metric between the real and generated

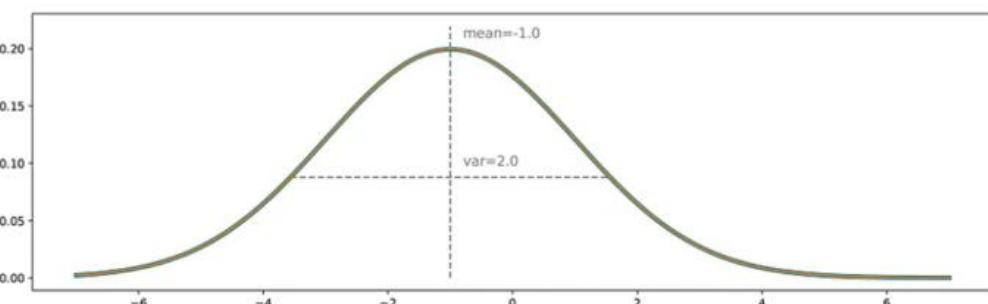
data distributions, GANs use an **indirect approach** which takes the form of a **proxy task** over the two distributions. The training of the generative network is then carried out with respect to this task, in such a way that it forces the generated distribution to become progressively closer to the true distribution. This proxy task in GANs is a **discrimination task between real and generated samples** or we could also describe it as a “non-discrimination” task since we want the discrimination to fail as much as possible.

In GAN architecture, we have a **discriminator** that takes both real and generated samples and **tries to classify them as accurately as possible**, and a **generator** that is trained to **fool the discriminator as effectively as possible**. Let's better understand this.

Suppose we have a true distribution, for example, a one-dimensional Gaussian, and we want a generator that can sample from this probability distribution. What we called “**direct**” **training method** would then consist in **iteratively adjusting the generator through gradient descent steps to correct** (reduce) **the measured difference/error between the true and generated distributions**. This is better shown in the Figures below where the blue curve represents the true distribution and the orange one is the generated distribution.



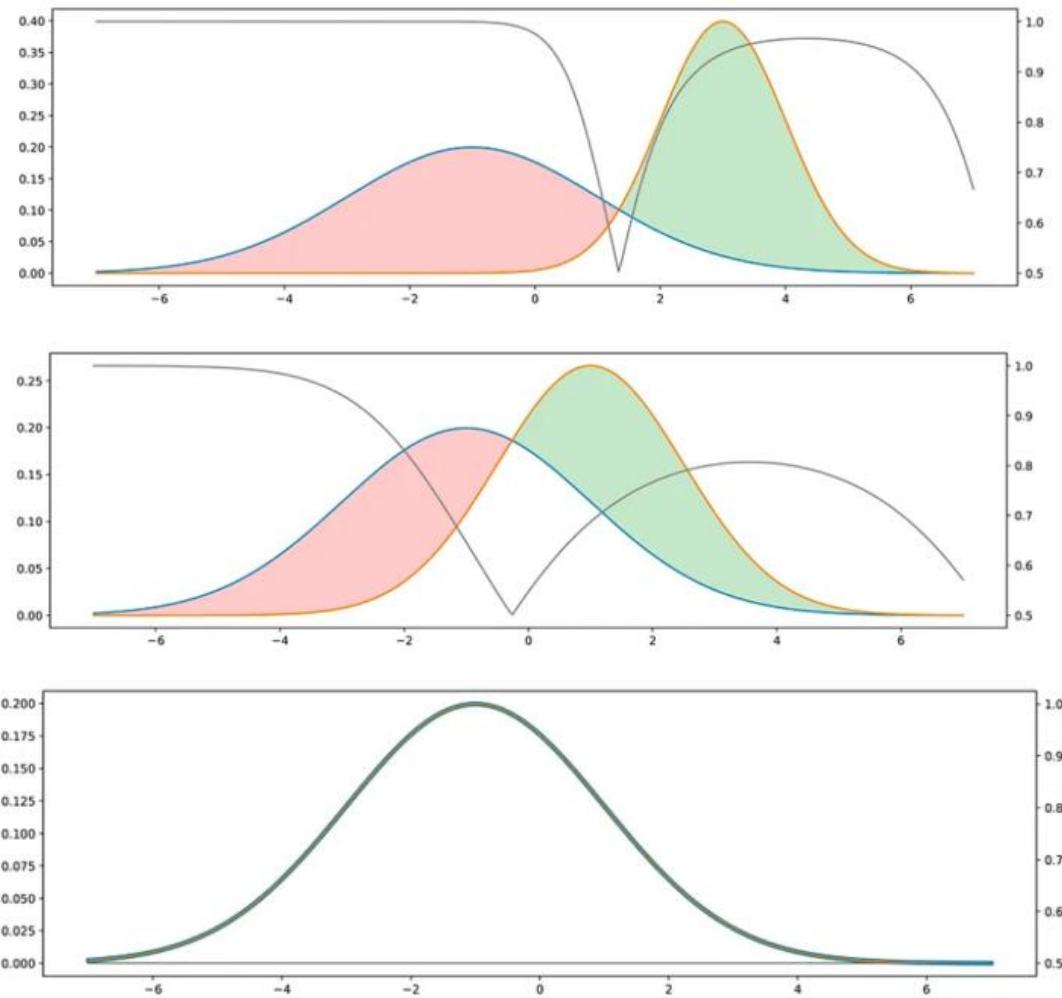
Assuming the optimization process is perfect, we should end up with the generated distribution matching exactly the true distribution.



In the “**indirect**” **approach**, we must also take into account the presence of a discriminator. We assume for now that this discriminator acts as an oracle that knows exactly how the true and generated distributions are made, and is able, based on this knowledge, to assign a class (“true” or “generated”) to any given point.

If the two distributions are far apart, the discriminator will be able to classify most of the presented points easily and with high confidence. In order to fool the discriminator, **the generated distribution must be brought closer to the true one**.

The discriminator will face the greatest difficulty when the two distributions are identical at all points: in this situation, for every point, there is an equal chance of it being either “true” or “generated”, leaving the discriminator with no meaningful information to base its decision on and so the best the discriminator can do is guess randomly — akin to flipping a coin — if they are true or generated, which yields correct predictions only half the time on average (50% prob).



Again, the blue curve represents the true distribution, and the orange one is the generated distribution. Additionally, in grey, plotted along a secondary y-axis on the right, is the **probability that the discriminator assigns to a point being true**, if it chooses the class with the higher density at each point (assuming that “true” and “generated” data are present in equal proportions).

The closer the two distributions become, the more often the discriminator is wrong. During training, the goal is to “move the green area towards the red area”.

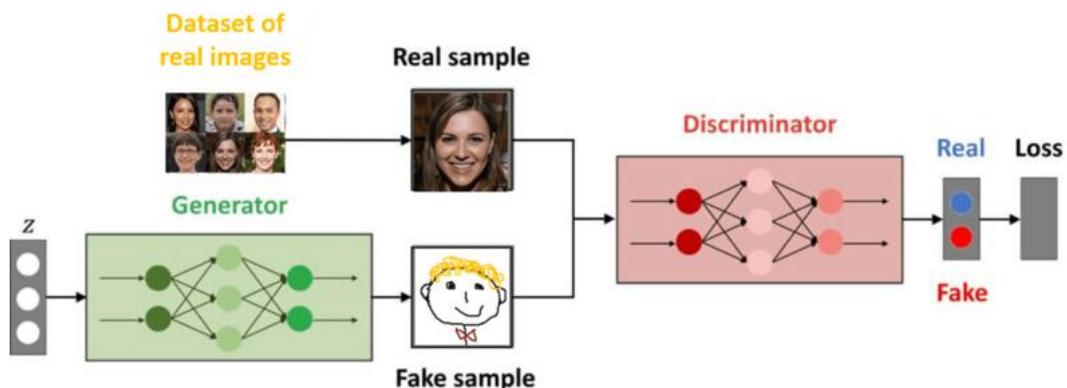
At this point, it becomes reasonable to question whether this “indirect” method is actually a good idea. Indeed, **it seems to be more complicated** (we must optimize the generator based on a proxy task rather than directly based on the distributions) and **it requires a discriminator that we consider in our current explanation as a given oracle, but that is, in reality, neither known nor perfect**.

1. Regarding the first concern, the difficulty of directly comparing two probability distributions based on samples counterbalances the apparent higher complexity of indirect method.
2. As for the second concern, it is obvious that the discriminator is not known; however, **it can be learned!**
→ another neural network can be used for it.

Architecture of a GAN & Adversarial Training

The generator is a neural network that models a transformation function. It takes as input a simple random variable and, once trained, must produce a random variable that follows the target distribution. In other words, it must generate new data that closely resembles the real data it was trained on.

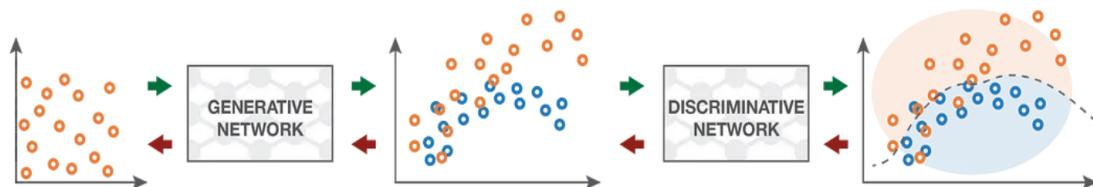
Since this transformation function is complex and unknown, we model the discriminator using another neural network. This neural network models a discriminative function: it takes in input “true” and “generated” data and outputs the probability that the generated data looks like the “true” one. In practice, it behaves as a **binary classifier**, and its loss function is typically modeled using **binary cross-entropy**.



Once defined, the **two networks can then be trained jointly with opposite objectives**:

- The discriminator is trained to correctly identify generated data as fake, minimizing its classification error.
- The generator, on the other hand, is trained to **fool the discriminator**. Since we want the discriminator to incorrectly label fake data as real, it makes sense to define the generator's loss as the **opposite** of the discriminator's loss. Thus, **the generator is trained to maximize the discriminator's classification error**.

■ Forward propagation (generation and classification) ■ Backward propagation (adversarial training)



Input random variables.

The generative network is trained to **maximise** the final classification error.

The **generated distribution** and the **true distribution** are not compared directly.

The discriminative network is trained to **minimise** the final classification error.

The classification error is the basis metric for the training of both networks.

At each iteration of the training process, the weights of the generative network are updated to increase the classification error (through **gradient ascent** on the generator's parameters), while the weights of the

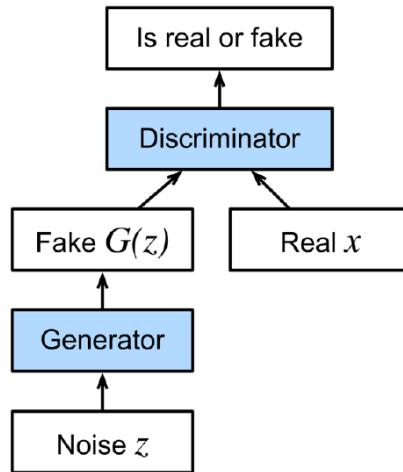
discriminative network are updated to decrease this error (through **gradient descent** on the discriminator's parameters).

These opposite goals and the implied notion of **adversarial training** of the two networks explain the name of "**adversarial networks**": **both network tries to beat each other**, and in doing so, they both improve over time. The competition between them makes these two networks "progress" with respect to their respective goals.

From a game theory point of view, we can think of this setting as a **minimax two-player game** where the **equilibrium state** corresponds to the situation in which **the generator produces data that perfectly matches the target distribution and the discriminator assigns a probability of 1/2 (0.5) to each point being "true" or "generated"**. This equilibrium point is known as the **Nash Equilibrium**.

Mathematics behind GANs

Let's now delve into the mathematical formulation that governs GAN training process.



The **discriminator is a binary classifier** designed to determine whether a given input **is real** (from real data) **or fake** (generated by the generator). Typically, the discriminator outputs a scalar prediction $o \in \mathbb{R}$, such as using a **fully connected layer with hidden size 1** (one output unit) **and then applies sigmoid function to obtain the predicted probability $\hat{y} \in [0,1]$** . The discriminator is trained to **minimize the binary cross-entropy (BCE) loss**:

$$L(y, \hat{y}) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$$

Assume the label y is **1 for the true data** and **0 for the generated (fake) data**. During training of the discriminator:

- For real data, the label is $y = 1$ and the prediction is $\hat{y} = D(x)$, so the associated loss is:

$$L_{real} = L(D(x), 1) = -\log(D(x))$$

- On the other hand, for the data generated by the generator, the label $y = 0$ and the prediction is $\hat{y} = D(G(z))$, therefore the associated loss is:

$$L_{fake} = L(D(G(z)), 0) = -\log(1 - D(G(z)))$$

Combining both, we get the overall discriminator loss:

$$L_D = L_{real} + L_{fake} = -\log(D(x)) - \log(1 - D(G(z)))$$

The discriminator is therefore trained to **minimize** this loss:

$$\min_D \left\{ -\log(D(x)) - \log(1 - D(G(z))) \right\}$$

The generator starts by **sampling a latent vector $z \in \mathbb{R}^d$ from a simple distribution**, such as a standard Gaussian, often referred to as the *noise distribution*. Next, the generator G maps this latent vector into a new data space, producing a fake sample $G(z)$. The goal of the generator is to **fool the discriminator** into believing this fake data is real, meaning it tries to make the discriminator's output satisfy $D(G(z)) \approx 1$. Defining the loss for the generator is less straightforward than for the discriminator, which is a binary classifier trained using the BCE loss. However, since the generator's objective is adversarial — essentially the opposite of the discriminator's — we can **smartly model the generator loss as the opposite of the discriminator loss**. In particular, since the generator has **no influence over** the real data x and thus cannot affect $D(x)$, its objective focuses solely on the term involving the generated data, $D(G(z))$:

$$L_G = -\log(1 - D(G(z)))$$

Therefore, the generator is trained to **maximize** the **cross-entropy loss when $y = 0$** (false data):

$$\max_G \left\{ -\log(1 - D(G(z))) \right\}$$

As a result, the GAN training process can be framed as a **minimax (zero-sum) game** between the generator and the discriminator, where both networks optimize a shared objective but with opposing goals:

$$\min_D \max_G \left\{ -\log(D(x)) - \log(1 - D(G(z))) \right\}$$

Note that this loss function is applied individually to each data point in the dataset. However, to consider the entire data distribution in the dataset, the equation is generalized to the expected values for the entire dataset:

$$\min_D \max_G \left\{ -\mathbb{E}_{x \sim p_{data}(x)} [\log(D(x))] - \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))] \right\}$$

Note: In the formulation above you may ask why to proceed with a minmax formulation even though the generator only influences the second term. Is it legit to bundle both terms in a single function? Since G only affects the second term (via $G(z)$), it **implicitly learns to maximize just $-\mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))]$** , while D affecting both terms tries to minimize the full loss.

The formulation more commonly found for the GAN, which is also the one used in the original paper [90], is:

$$\min_G \max_D \left\{ \mathbb{E}_{x \sim p_{data}(x)} [\log(D(x))] + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))] \right\}$$

This is **equivalent** to the one seen above, differing only in sign and the order of optimization. In the next we will use this notation.

In practice, **GAN training** involves alternating updates between the two networks. Specifically, we update the generator's parameters while keeping the discriminator's parameters fixed and then update the discriminator's parameters while keeping the generator's parameters fixed. This alternating process is repeated for many iterations, in each case taking just one gradient descent step over a mini-batch.

Over time, if training is successful, the generator becomes so good that the discriminator can no longer reliably distinguish real from fake data and hence will always produce an output of 0.5. This situation represents a **Nash Equilibrium** of the adversarial game.

Once training is complete, the **discriminator network is discarded** and the generator network can be used to synthesize new examples in the data space by sampling from the latent space and propagating those samples through the trained generator network.

Difficulties with Training GANs

Although GANs have demonstrated impressive capabilities in generating high-quality images, they are **notoriously difficult to train successfully**. This is primarily due to the adversarial nature of the training process. Unlike standard models that minimize a clear loss function like a **likelihood-based error**, GANs involve a two-player minimax game where **one model's progress may come at the expense of the other**. As a result, the loss function may not decrease consistently over time, but instead it can go up as well as down during training, making training behavior unstable and difficult to monitor. Below are some of the main challenges commonly encountered when training GANs:

- **Vanishing Gradients:**

In practice, **minimizing** the original generator loss

$$\min_G \left\{ \log \left(1 - D(G(z)) \right) \right\}$$

can lead to **vanishing gradients** issues, particularly in the **early stages of training**. This happens because, at the beginning, the generator is typically weak and produces poor-quality outputs that are **easily distinguishable from real data**. As a result, the discriminator confidently classifies these generated samples as fake, outputting values close to 0, i.e. $D(G(z)) \approx 0$. When this occurs, the term $\log(1 - D(G(z)))$ **saturates** and its gradient becomes very small, causing the generator to receive almost no feedback. With such weak gradients, the generator struggles to improve, effectively **stalling** the training process.

To overcome this issue, a common practical trick (**non-saturating heuristic**) is to change the generator's objective to **maximize**:

$$\max_G \left\{ -\log(D(G(z))) \right\}$$

This modified loss provides **stronger gradients**: the gradient is **large** when $D(G(z)) \approx 0$, meaning the generator receives a **strong learning signal** right when it needs it most.

In essence, although the generated sample is **detected as fake**, the modified loss **maximizes the probability that the generated sample is classified as real**. Both serve the same end goal — fooling the discriminator — but the second provides more effective gradients during early training.

- **Mode Collapse:**

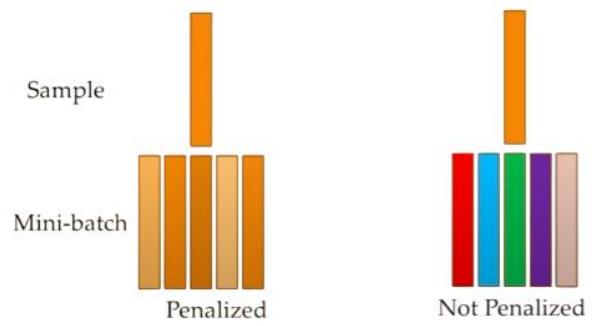
One of the most common challenges that you can encounter when training a GAN is **mode collapse**. This happens when the generator learns to map different latent vectors z to a **limited subset of possible outputs (modes)**, ignoring the full diversity of the data distribution. Instead of generating a wide variety of samples, **the generator collapses to producing only a few types of outputs**.

Why does this happen? The generator may discover that producing a specific type of output that is actually able to “fool” the discriminator is **sufficient to minimize its loss** and it exploits this by focusing on those output. As a result, **the generator fails to cover the full range of the true data distribution.**

For example, if a GAN is trained on handwritten digits, mode collapse might cause it to generate only the digit ‘3’ repeatedly. The discriminator might be fooled because these ‘3’s look real, but the generator is clearly not capturing the diversity of digits in the data.

In extreme cases, the generator may output just one or a few samples. The discriminator assigns a probability of 0.5 to these, and training **stalls** because the discriminator can no longer provide useful feedback to improve the generator.

A common strategy to mitigate mode collapse is to use **minibatch discrimination** (also called **minibatch features**). This technique involves evaluating each generated sample in the context of other samples within the same minibatch. If the generated samples are too similar to each other — that is, if the generator produces nearly identical outputs within a batch — the model is **penalized**. This encourages the generator to produce a more diverse set of outputs by pushing it to explore and capture different modes of data distribution, thus preventing it from collapsing to a single or few repeated outputs.



- **Non-convergence and instability**

The nature of the adversarial training makes GANs inherently **unstable**. This instability stems from the **lack of a clear global objective**. The generator and discriminator are constantly adapting to each other, so the gradient landscape is **non-stationary**.

- **Balance between the discriminator and generator**

The discriminator and generator networks may become **unbalanced** during training, leading to one network overpowering the other and preventing the other network from learning effectively. Since the two models are adversaries in a zero-sum game, any significant disparity in their performance can destabilize the learning dynamics.

- **Sensitivity to Hyperparameters**

GAN training is **notoriously sensitive to hyperparameter choices**. Small changes in these factors can lead to **dramatically different outcomes**, ranging from successful convergence to complete failure. Again, this sensitivity arises because GANs optimize two competing objectives, and so slight misalignments in their dynamics can have dramatic effects.

- **Lack of a Proper Evaluation Metric**

While GANs are capable of generating visually impressive images (as we will see soon), a fundamental challenge remains: **how do we objectively measure their performance?**

It would be nice to quantitatively evaluate the model; however, GANs are not born with a good objection function that can inform us of the training progress (i.e. we cannot trust validation loss). Without a good evaluation metric, it is like working in the dark:

- No good sign to tell when to stop;
- No good indicator to compare the performance of multiple models.

In the absence of a precise evaluation metric, often we proceed by visual inspection to diagnose eventual mode collapse or sample quality deterioration over time. Therefore, in GANs we often rely on a **qualitative evaluation**, rather than a quantitative one. Moreover, this approach poses important questions: do GANs truly do good generations or just generations that appeal/fool to the human eye? Are humans good discriminators for the converged generator?

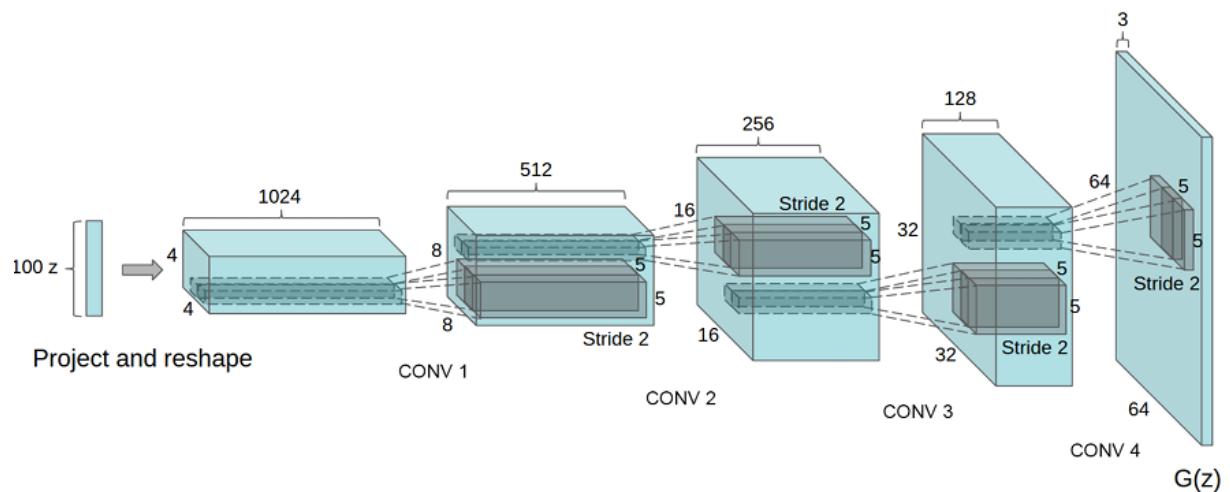
GANs in Image Domain

The foundational concept of GANs has sparked a vast amount of research, leading to numerous algorithmic innovations and a wide range of applications. Among these, one of the most successful and widespread uses of GANs is in **image generation**.

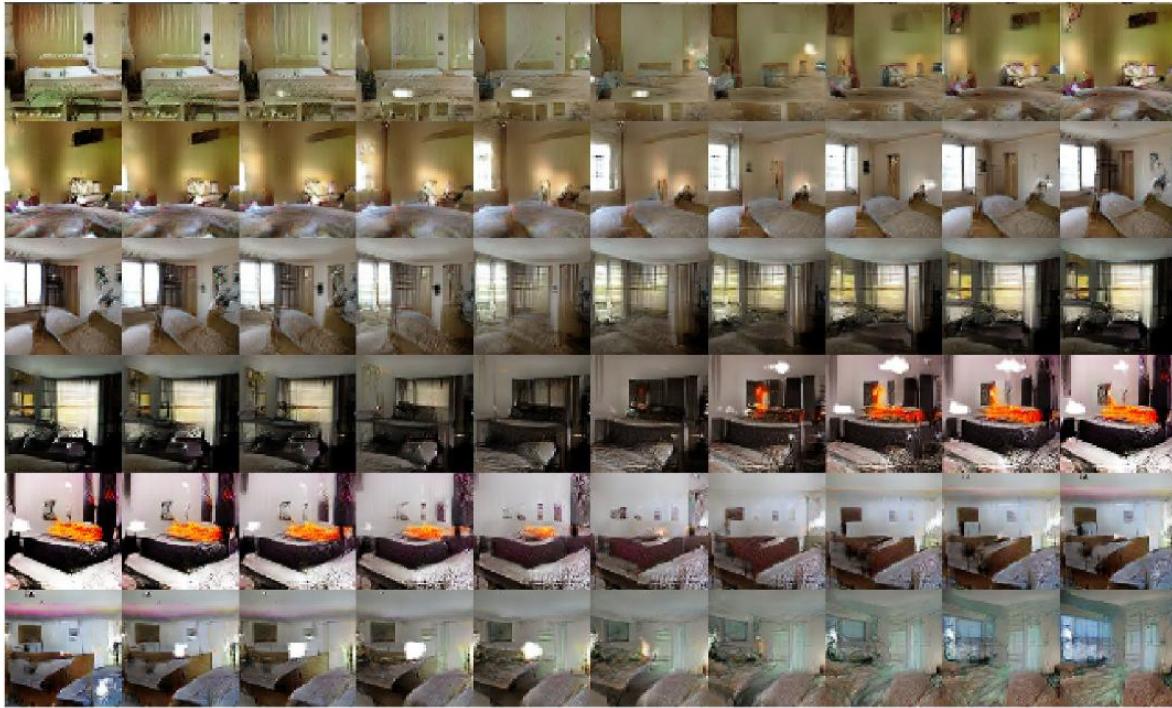
DCGAN

Early GAN models used FC networks for the generator and discriminator. However, there are many benefits to using CNNs, especially for images of higher resolution. This insight led to the development of the **Deep Convolutional GAN (DCGAN)** by Radford et al. [91], introduced just a year after the original GAN paper.

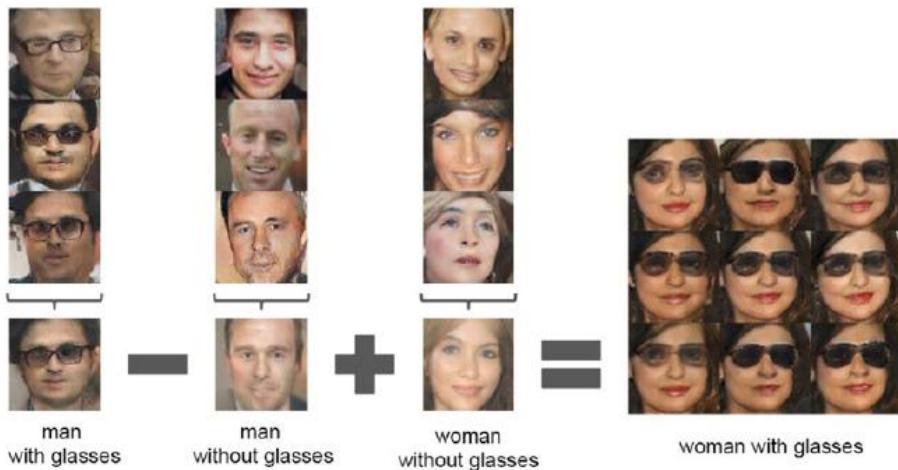
The discriminator network takes an image as input and provides a scalar probability as output, so in DCGAN a **standard convolutional network** is appropriate. The generator network needs to map a lower-dimensional latent space into a high-resolution image, and so a **network based on transpose convolutions** is used, as illustrated in Figure below. High quality images can be obtained by progressively growing both the generator network and the discriminator network starting from a low resolution and then successively adding new layers that model increasingly fine details as training progresses.



The authors shown that when the DC-GAN is trained on a dataset of bedroom images and random samples from the latent space are propagated through the trained generator, the generated images also look like bedrooms, as expected. In addition, however, the latent space has become **organized** in ways that are **semantically meaningful**. For example, if we follow a smooth trajectory through the latent space and generate the corresponding series of images, we obtain smooth transitions from one image to the next, as seen in Figure below.



Moreover, it is possible to identify directions in latent space that correspond to semantically meaningful transformations in the generated images. For example, when trained on face images, one direction might control the orientation of the face, whereas other directions might correspond to changes in lighting or the degree to which the face is smiling or not. These are called **disentangled representations** and allow new images to be synthesized having specified properties. The Figure below shows an example of such semantic manipulations from a DCGAN trained on CelebA dataset — a collection of celebrity faces annotated with attributes like hair color and facial expression — where attributes like gender or the presence of glasses correspond to particular directions in latent space, allowing targeted **image synthesis** with desired attributes.



The architectural innovations introduced by DCGAN have had a lasting impact, influencing many subsequent GAN models that adopted its design principles.

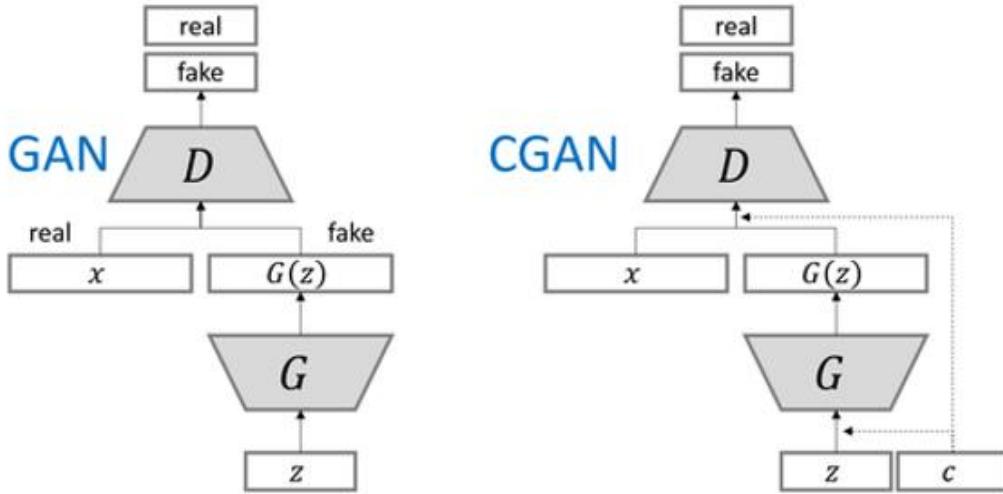
Conditional GAN

The standard GAN, as it is, generates samples from the **unconditional distribution $p(x)$** . For instance, if trained on dog images, it can generate synthetic images of dogs, but without control over specific attributes such as the

dog's breed. To incorporate such control, **Conditional GANs (cGANs)**, introduced by *Mirza and Osindero (2014)* [92], extend the GAN framework by introducing a **conditioning variable c** . This variable can represent labels, text descriptions, class categories or any auxiliary information relevant to the data. The model then learns to generate data from the **conditional distribution $p(x|c)$** , enabling it to produce samples that conform to specific conditions such as generating only huskies or golden retrievers, rather than random dogs.

To achieve this, **both** the generator G and the discriminator D are modified to **take c as an additional input**:

- The generator receives both a latent vector z and the conditioning vector c and outputs $G(z|c)$, a synthetic data sample conditioned on c .
- The discriminator takes as input either a real data sample x and a generated sample $G(z|c)$, **along with** the same conditioning vector c and outputs a probability indicating whether the sample is real or fake **under that condition**.



The cGAN objective function becomes:

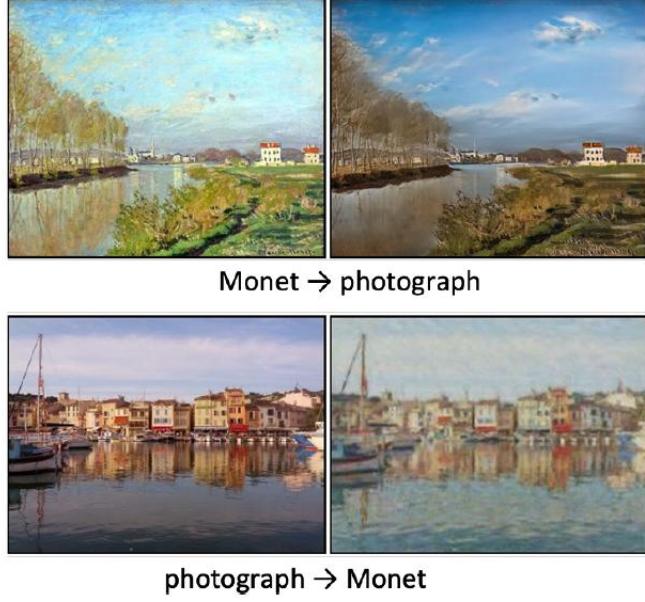
$$\min_G \max_D \left\{ \mathbb{E}_{x \sim p_{data}(x)} [\log(D(x|c))] + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z|c)))] \right\}$$

During training, labeled examples in the form of pairs $\{(x_i, c_i)\}$ are used. Once trained, images from a desired category can be generated by setting c to the corresponding label vector.

Compared to training separate GANs for each class, this has the advantage that **shared internal representations** can be learned jointly across all classes, thereby making **more efficient use of the data**. Furthermore, conditional generation has been shown to **reduce mode collapse** by providing explicit structure and supervision to guide the generator.

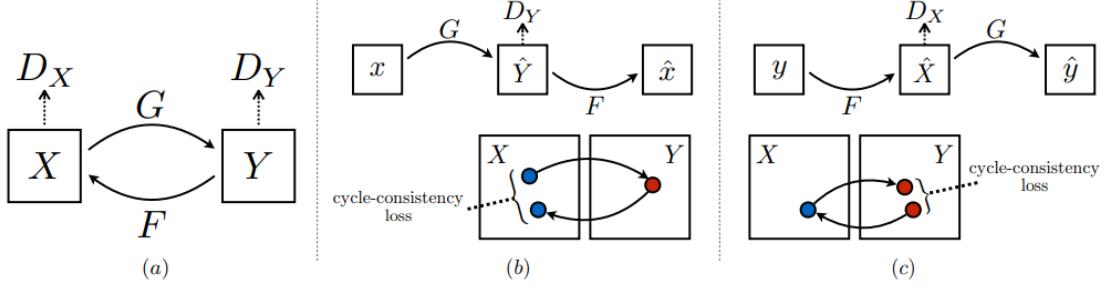
CycleGAN

Imagine you are tasked with transforming a photograph into a Monet painting that captures the same scene — **not just mimicking the painterly style but doing so in a way that preserves the semantic content of the original image**. Likewise, you might want to perform the inverse transformation: converting a Monet painting into a photorealistic depiction of the imagined landscape. This problem, known as ***image-to-image translation***, often falls under the broader umbrella of ***style transfer***, though it requires more than simply applying superficial aesthetics — the transformation must remain faithful to the underlying structure of the scene.



CycleGAN, introduced by Zhu et al. in 2017 [93], provides an elegant solution to this challenge, especially in the **unpaired** setting — that is, when we have no direct one-to-one correspondences between photographs and their painted equivalents. Unlike supervised translation models that require such aligned pairs, CycleGAN is designed to work with two **independent** datasets: one of photographs and one of Monet paintings, with no need for images to correspond in content.

At the heart of CycleGAN is the idea of learning two mappings: one from the domain of photographs (call it domain X) to the domain of Monet paintings (domain Y) and another in the reverse direction, from Y back to X .



To achieve this, CycleGAN makes use of two conditional generators, G_X and G_Y , and two discriminators, D_X and D_Y . The generator G_X takes as input a sample painting $y \in Y$ and generates a corresponding synthetic photograph x , whereas the discriminator D_X distinguishes between synthetic and real photographs.

Similarly, the generator G_Y takes a photograph $x \in X$ as input and generates a synthetic painting y , whereas the discriminator D_Y distinguishes between synthetic paintings and real ones. The discriminator D_X is therefore trained on a combination of synthetic photographs generated by G_X and real photographs, whereas D_Y is trained on a combination of synthetic paintings generated by G_Y and real paintings.

If we were to train this architecture using only standard GAN objectives, the result might be visually plausible paintings and photographs — but with no guarantee that a painting generated from a photograph actually reflects the content of the original scene. In other words, the model could simply generate arbitrary paintings that look Monet-like without maintaining any meaningful relationship to the input photo.

To address this, CycleGAN introduces an additional term in the loss function called the **cycle consistency loss**. The goal is to ensure that when a photograph is translated into a painting and then back into a photograph it should

be close to the original photograph, thereby ensuring that the generated painting retains sufficient information about the photograph to allow the photograph to be reconstructed. Similarly, when a painting is translated into a photograph and then back into a painting it should be close to the original painting.

Formally the cycle consistency loss is defined as:

$$L_{cyc}(G_X, G_Y) = \mathbb{E}_{x \sim p_{data}(x)} [\|G_X(G_Y(x)) - x\|_1] + \mathbb{E}_{y \sim p_{data}(y)} [\|G_Y(G_X(y)) - y\|_1]$$

where $\|\cdot\|_1$ denotes the l_1 norm. This cyclic consistency loss term is incorporated into the total loss function along with the standard GAN loss terms, resulting in the following total loss function:

$$L_{GAN}(G_X, D_Y, X, Y) + L_{GAN}(G_Y, D_X, Y, X) + \lambda L_{cyc}(G_X, G_Y)$$

where the coefficient λ determines the relative importance of the cycle consistency error.

Once trained, CycleGAN learns to perform image-to-image translation in a way that is not only stylistically convincing, but also structurally faithful to the source image. Remarkably, this is accomplished without the need for paired data — a significant advancement that has enabled a wide range of applications such as artistic style transfer.

Other GANs

Over the years significant advances have been made in GAN architectures and training methodologies, leading to models that are increasingly stable, powerful and capable of producing **high-fidelity outputs** (see Figure below). Notable examples include **Progressive GANs (ProGAN)** [94] and **StyleGAN** [95], each introducing architectural innovations such as progressive growing and style-based modulation.



Alongside these developments, other impactful improvements come from the use of **non-saturating loss functions** — most notably the **Wasserstein GAN (WGAN)** [96] (and its gradient-penalized variant **WGAN-GP** [97]) and **hinge-based losses** [98] [99] — that have played a key role in stabilizing training and addressing issues like mode collapse.

However, as emphasized earlier, training GANs remains a notoriously challenging task. They are highly sensitive to hyperparameters, suffer from non-convergent dynamics and often require large datasets (they are very **data-hungry models**) and careful architectural tuning. Despite these challenges, continued research into architectures, regularization techniques and loss formulations made GANs advance their ability to model complex data distributions with increasing realism and control.

Variational Autoencoder

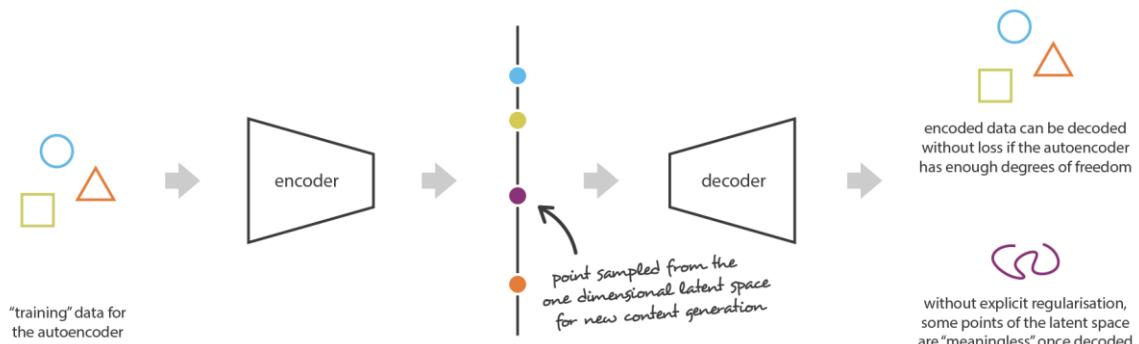
In the previous chapter, we saw that GANs are generative models capable of producing samples that are statistically indistinguishable from real data by **implicitly** (indirectly) learning the data distribution through **adversarial training**. In contrast, **Variational Autoencoders (VAEs)** are **explicit** (direct) probabilistic generative models that aim to maximize the **likelihood of the observed data**, rather than solving a proxy task such as the discrimination problem in GANs. VAEs learn the underlying structure of the data by mapping it into a **structured latent space**, from which new samples can be drawn directly after training.

The VAE was originally introduced by Kingma and Welling in 2013 [100] and later received a more comprehensive formalization and thorough evaluation in their 2019 paper, “*An Introduction to Variational Autoencoders*” [101]. Although the name suggests an autoencoder architecture — which, it must be said, is an integral component of the VAE — it is more accurate to describe the VAE as a **nonlinear latent variable model**. For this reason, after introducing the limitations of standard autoencoder and the intuition behind what we want a VAE to achieve, we will begin by introducing latent variable models and then delve into the specific case of the nonlinear latent variable model; the autoencoder architecture, as we will see, will be a natural choice to implement this type of model.

Motivation for VAEs: The Limitations of Standard Autoencoders

In a conventional autoencoder, once training is complete, we obtain an encoder that maps input into a latent representation and a decoder that reconstructs the original data from this latent code. One might wonder: could we simply sample a random point from the latent representation and decode it to generate new data? If the latent representation was **regular enough** (well “organized” by the encoder during the training process) it might seem plausible: we could sample a point randomly from it and decode that to generate a new, meaningful output (i.e. the decoder would then act more or less like the generator of a GAN). However, in practice, this assumption **rarely holds**, instead the latent space of a standard autoencoder is generally irregular and poorly structured, which makes sampling from it unreliable.

The distribution of encoded points depends heavily on the data, the architecture and the dimensionality of the latent space. Since autoencoders are trained solely to minimize reconstruction loss, they are **not incentivized to structure (organize) the latent space in any meaningful way for generative purposes**.



The model tends to exploit any **overfitting opportunities** available to optimize its reconstruction performance. As a result, it often learns to encode inputs into distinct points or clusters — frequently with discontinuities between them — simply because this helps the decoder reconstruct the inputs more accurately. Consequently, the latent space tends to form sparse, arbitrarily scattered clusters, leaving large portions unused. If we attempt to generate

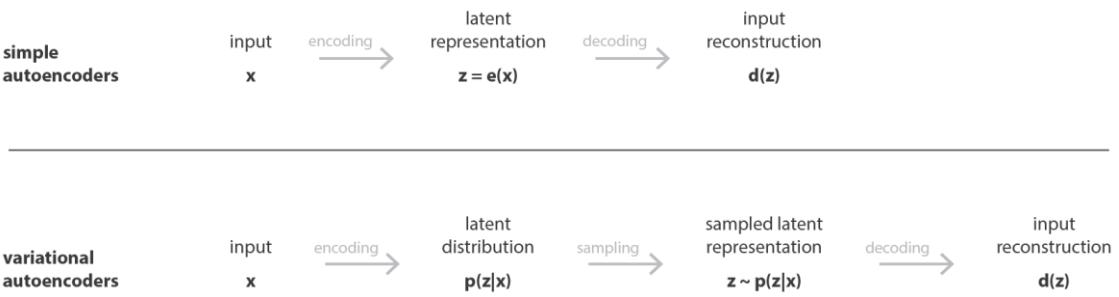
new data by sampling random points from this space, we are likely to land in regions the decoder has never encountered — resulting in **nonsensical outputs**.

In order to be able to use the decoder of our autoencoder for generative purpose, we have to be sure that the encoded latent space is “regular enough”. One possible solution to obtain such regularity is to introduce **explicit regularization** during the training process in a way that ensures the latent space is **continuous, smooth** and **densely covered**. This is precisely what VAEs aim to achieve.



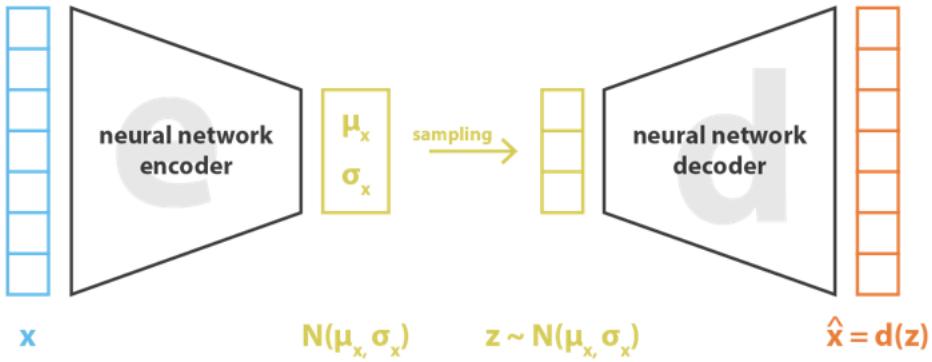
A VAE, like a traditional autoencoder, consists of two components: an encoder that maps input data to a latent representation and a decoder that reconstructs the data from that latent representation. However, the key innovation of VAEs lies in how they perform this encoding. Instead of mapping an input to a single fixed point in the latent space, a VAE encodes it as a **probability distribution over the latent space**. In particular, during training, for each input

- the encoder first produces the parameters of a probability distribution — typically a multivariate Gaussian — characterized by a mean vector and a covariance matrix.
- a sample is then drawn from this distribution and passed to the decoder, which attempts to reconstruct the original input from this sampled point.
- the reconstruction error is then computed and backpropagated through the network to update both the encoder and decoder parameters.



Choosing Gaussian distributions for the latent space is both mathematically convenient and computationally efficient. By encoding inputs as distributions with non-zero variance rather than as fixed points, the model can express a form of uncertainty and variability in the latent representation. More importantly, this allows us to impose a regularization term during training that pushes the encoded distributions to remain close to a predefined prior, usually a standard multivariate distribution with zero mean and unit variance, $\mathcal{N}(0, I)$.

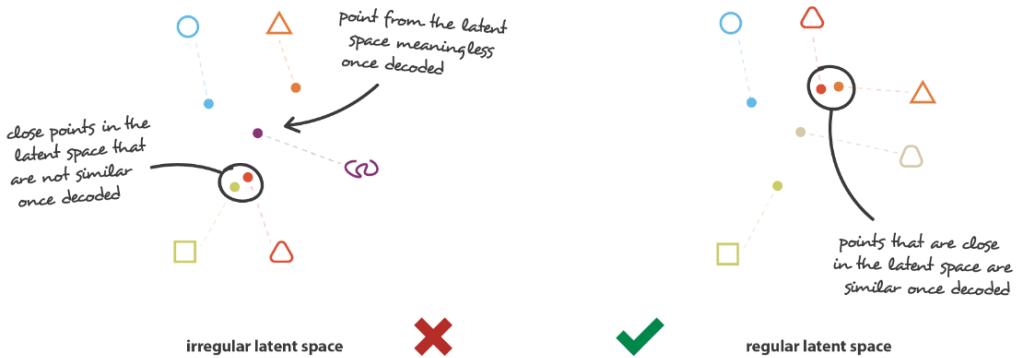
The VAE loss function, therefore, consists of two components. The first one is the **reconstruction term**, which ensures that the decoder can accurately reconstruct the input data from the latent representation. The second one is the **regularization term**, which penalizes deviations of the encoded distributions from the chosen prior distribution. This penalty is typically formulated as the **Kullback-Leibler (KL) divergence** between the encoder's output distribution and the prior distribution. The KL divergence measures how much the learned distribution differs from the desired prior and encourages the latent space to be well-structured and regular.



$$\text{loss} = \|x - \hat{x}\|^2 - \text{KL}[N(\mu_x, \sigma_x), N(0, I)] = \|x - d(z)\|^2 + \text{KL}[N(\mu_x, \sigma_x), N(0, I)]$$

The motivation for this regularization is rooted in two fundamental properties expected from the latent space in order to make generative process possible:

- **Continuity:** Close points in the latent space should yield similar contents upon decoding.
- **Completeness:** Points sampled from the latent space should result in meaningful “contents” once decoded.

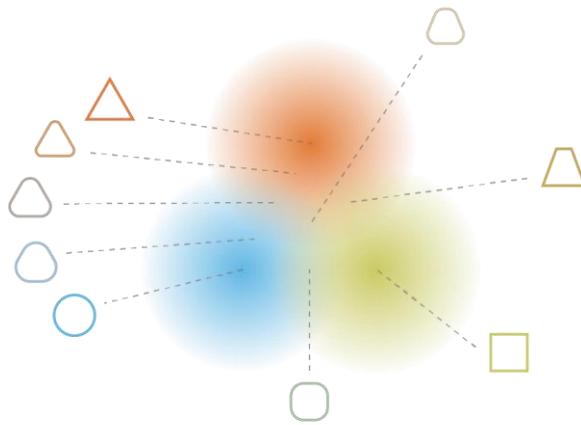


The only fact that VAEs encode inputs as distributions instead of simple points is not sufficient to ensure continuity and completeness. Without a well-defined regularization term, the model can learn (in order to minimize its reconstruction error) to “ignore” the distributions that are returned and behave almost like classic autoencoders (leading to overfitting). In fact, the encoder can either return distributions with tiny variances (that would tend to be punctual distributions) or return distributions with very different means (that would then be really far apart from each other in the latent space). In both cases, distributions are used the wrong way (cancelling the expected benefit) and continuity and/or completeness are not satisfied.

So, in order to avoid these effects we have to regularize both the covariance matrix and the mean of the distributions returned by the encoder. Given that we want to be close to a standard normal distribution, we require

the covariance matrices to be close to the identity, preventing punctual distributions, and the mean to be close to 0, preventing encoded distributions to be too far apart from each others. This encourages as much as possible returned distributions to “overlap”, satisfying in this way the expected continuity and completeness conditions. Naturally, as for any regularization term, this comes at the price of a higher reconstruction error on the training data. The trade-off between the reconstruction error and the KL divergence can however be adjusted as we will see at the end of the chapter.

To conclude, we can observe that the continuity and completeness enforced by regularization tend to create a smooth “transition” of information throughout the latent space. For instance, a point located halfway between the means of two encoded distributions coming from different training data should be decoded in something that is somewhere between the data that gave the first distribution and the data that gave the second distribution as it may be sampled by the autoencoder in both cases.



Latent Variables Models

Let’s begin by asking a fundamental question: why do we use latent variable models at all? When we observe data — say, images, speech or text — we see only the surface: the pixels, the sounds, the words. But we often suspect that something deeper is at play. These observable data points, which we denote as x , are likely the result of *hidden* or *latent* variables z that capture high-level features of these observed data points. These can be then used to generate and explain our observed data points when used in combination.

In **latent variable models**, we therefore try to explain the observed data by assuming that each datapoint x is generated from some latent variables z , via a conditional distribution $p(x|z)$. This relationship can be viewed as:



So, if we have access to these latent features z , we can sample new data points that resemble the observed data — like generating new images. However, in practice, we don’t know what z is for a given x . What we would really like to do is **invert** the generative process: starting from an observed data point x , we want to recover the underlying z that might have generated it, i.e. we say that we are “learning to represent” data x via z .

Now, let’s introduce a key concept in probabilistic modeling: **marginal likelihood**. This is crucial for understanding how latent variable models define the distribution of observable data.

Suppose we have a dataset describing eye color distributions for different nationalities. You can imagine this as a table where each entry $p(x, z)$ gives the probability of a person having a certain eye color x *and* a certain nationality z :

	Dutch	Greek	Chinese	Indian	Italian	German	US	Spanish
Brown	0.02	0.03	0.02	0.01	0.07	0.03	0.01	0.00
Blue	0.09	0.01	0.03	0.03	0.08	0.04	0.03	0.06
Green	0.01	0.01	0.08	0.06	0.07	0.08	0.06	0.06

This is a joint probability distribution $p(x, z)$ and therefore, being a *pdf*, the values of all entries together must sum to 1.

Now, suppose we want to know the distribution of one of our observed variables regardless of the values of the latent variable. In our case this will mean we want to know how likely each eye color is, regardless of nationality. To achieve this, we need to apply the **marginalization operation**, which allows us to compute the probability of an observed outcome by summing over all possible values of the latent variable:

$$p(x) = \sum_z p(x, z)$$

For instance, if we want to compute the probability of having brown eyes, we have to sum up the probabilities of having brown eyes while being Dutch, Greek, Chinese and so forth until we have considered all possible nationalities:

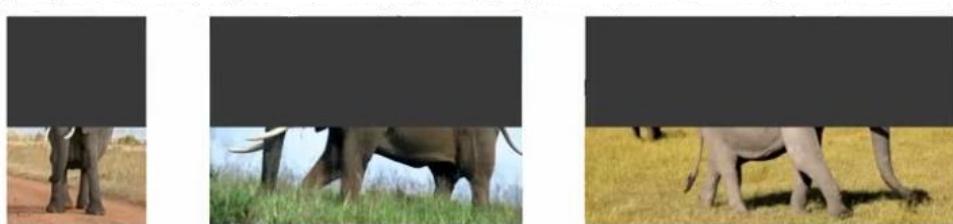
$$p(\text{Brown}) = p(\text{Brown}, \text{Dutch}) + p(\text{Brown}, \text{Greek}) + p(\text{Brown}, \text{Chinese}) + \dots$$

Doing this for all eye colors, we get:

Colors	
Brown	0.19
Blue	0.37
Green	0.44
Total	1.00

This operation gives us the marginal distribution $p(x)$, the probability of eye color independently of nationality. The resulting quantity is known as the **marginal likelihood**, as it reflects the likelihood of the observed data after integrating out — or marginalizing over — the latent variable.

Now, think of a more visual example. Imagine you have a set of images where only the bottom half is visible (this is our observed data x), while the top half is missing (this is our latent variable z).



Suppose we have a model that describes how the top and bottom halves of an image interact, i.e., we know $p(x, z)$. What we'd like to compute is how likely it is to observe each bottom half of the image, regardless of what the top half could be. To do that we can again marginalize out z :

$$p(x) = \int_z p(x, z) dz$$

Because the latent variable here is continuous (not categorical like nationalities), we use an integral instead of a sum. This integral considers *all possible completions* of the top half of the image and adds up their probabilities weighted by how likely each completion is.

For example, one possible top half might contain a large elephant; another might have two smaller elephants; yet another might have none at all. By accounting for all these possibilities, the marginal probability $p(x)$ tells us how likely it is to observe that bottom half of an image under all plausible upper halves.

Therefore, **latent variable models** take an indirect approach to describing a probability distribution $p(x)$ over a multi-dimensional variable x . Instead of directly writing the expression for $p(x)$, they model a joint distribution $p(x, z)$ of the observed data x and an unobserved latent variable z . They then describe the probability of $p(x)$ as a marginalization of this joint probability so that:

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

Typically, the joint probability $p(x, z)$ is broken down using the rules of conditional probability into the *likelihood* of the data with respect to the latent variables term $p(x|z)$, also known as **conditional likelihood**, and the **prior** $p(z)$:

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

This might seem like a roundabout, but it gives us flexibility because by choosing simple forms for $p(z)$ and $p(x|z)$, we can model very complex distributions $p(x)$.

For example, we could think to model the joint distribution as a weighted sum of Gaussian components, i.e. a **Mixture of Gaussians**. We recall from the ML Course that in 1-D (univariate) mixture of Gaussians, the latent variable z is discrete and selects which Gaussian component a data point belongs to. Suppose there are K components, each represented by a Gaussian with its own mean μ_k and variance σ_k^2 , then:

- The prior $p(z)$ is a categorical distribution with one probability π_k for each possible value of z .

$$p(z = k) = \pi_k$$

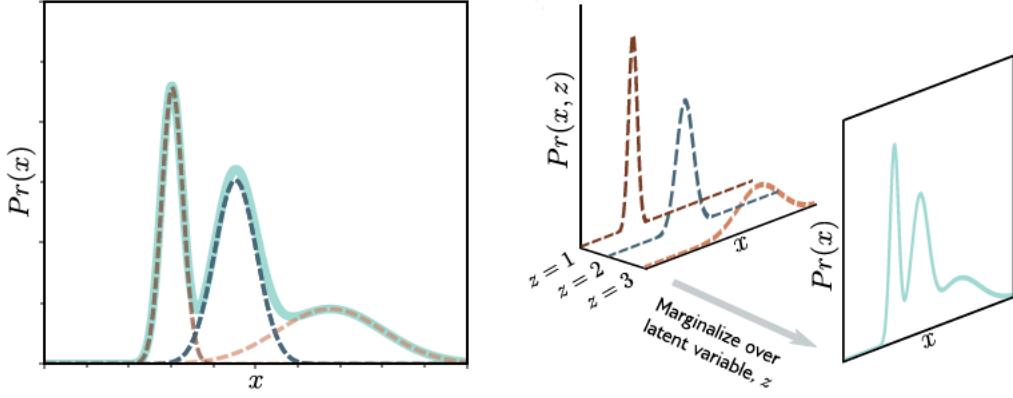
- The conditional likelihood for a specific value of z , namely $p(x|z = k)$, is modeled by a Gaussian distribution with parameters μ_k and σ_k^2 :

$$p(x|z = k) = \mathcal{N}_x[\mu_k, \sigma_k^2]$$

The marginal likelihood $p(x)$ is given by marginalization over the latent variable z . Here, the latent variable is discrete, so we sum over its possible values to marginalize:

$$\begin{aligned} p(x) &= \sum_{k=1}^K p(x, z = k) \\ &= \sum_{k=1}^K p(z = k) p(x|z = k) \\ &= \sum_{k=1}^K \pi_k \cdot \mathcal{N}_x(\mu_k, \sigma_k^2) \end{aligned}$$

As we can see in the Figure below, with this structure, we can describe a complex multi-modal probability distribution from a combination of simple expressions for the likelihood and prior.



We could think of using such a model to describe our latent space, however mixture of gaussians have two important limitations:

- they are discrete, while we would like to work with data that is continuous;
- they are linear, which limits their ability to capture complex, nonlinear patterns in data like images.

Non-linear Latent Variable Model

Now let's consider the **non-linear latent variable model**, which is what the VAE actually learns. In the nonlinear latent variable model, both the observed data x and the latent variable z are continuous and multivariate. In particular, in the case of the VAE:

- The **prior** $p(z)$ is a standard multivariate normal distribution:

$$p(z) = \mathcal{N}_z(\mathbf{0}, I)$$

- The **conditional likelihood** $p(x|z, \varphi)$ is a normal distribution as before, but its mean is a nonlinear function $f(z, \varphi)$ of the latent variable implemented through a neural network parameterized by φ and its covariance is assumed to be spherical $\sigma^2 I$:

$$p(x|z, \varphi) = \mathcal{N}_x(f(z, \varphi), \sigma^2 I)$$

The model $f(z, \varphi)$ describes the important aspects of the data and the remaining unmodeled aspects (unexplained variability) are attributed to the noise $\sigma^2 I^1$.

The data probability $p(x, \varphi)$ is found by marginalizing over the latent variable z :

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

¹ The choice of a **spherical covariance** is adopted to simplify the model in several ways. First, it simplifies the computations, since the covariance matrix is diagonal and uniform, avoiding the estimation of a full matrix, which would require learning more parameters and would entail a higher computational cost. Furthermore, sphericity implies that the residual noise is **isotropic**, i.e. with equal intensity (uniform) in all directions and uncorrelated across dimensions of x , which makes the math and optimization easier to manage. This means that, once the model has captured the nonlinear relationship via the function $f(z, \varphi)$, the remaining variability is treated as simple, independent Gaussian noise with same intensity in each dimension.

$$\begin{aligned}
&= \int p(x|z, \varphi)p(z)dz \\
&= \int \mathcal{N}_x(f(z, \varphi), \sigma^2 I) \cdot \mathcal{N}_z(0, I)dz
\end{aligned}$$

This can be viewed as an **infinite weighted sum** (i.e., an infinite mixture) **of spherical Gaussians** with different means, where the weights are $p(z)$ and the means are the network outputs $f(z, \varphi)$.

VAE's Training

How can we find the optimal parameters φ ? By **maximizing** the **log-likelihood** over an observed dataset $\{x_i\}_{i=1}^N$. For simplicity, let's assume that the variance term σ^2 in the likelihood expression is known, so our optimization focuses solely on learning φ ([later we will see how we can compute it](#)):

$$\hat{\varphi} = \operatorname{argmax}_{\varphi} \left[\sum_{i=1}^N \log(p(x^{(i)}|\varphi)) \right]$$

Since we are using a latent variable model, we express the likelihood inside the summation by marginalizing over the latent variables z :

$$= \operatorname{argmax}_{\varphi} \left[\sum_{i=1}^N \log \left(\int p(x^{(i)}, z|\varphi) dz \right) \right]$$

Unfortunately, the integral inside our expression is **intractable**: there's no closed-form expression and evaluating it numerically for each x_i is computationally expensive.

ELBO

To make progress, we define a **lower bound on the log-likelihood**. This is a function that is always less than or equal to the log-likelihood for a given value of φ and will also depend on some other parameters θ . Eventually, we will build a network to compute this lower bound and optimize it. Fortunately, to mathematically define it, we can make use of the *Jensen's inequality*.

Jensen's inequality says that:

A concave function $h(\cdot)$ of the expectation of data y is greater than or equal to the expectation of the function of the data:

$$h(\mathbb{E}[y]) \geq \mathbb{E}[h(y)]$$

In our case, the **logarithm is a concave function** and the **integral represents an expectation** with respect to a distribution over the latent variable, so we can think of using it to derive the lower bound for our log-likelihood.

We start by multiplying and dividing the log-likelihood by an arbitrary probability distribution $q(z)$ over the latent variables:

$$\log \left(\int p(x, z|\varphi) dz \right) = \log \left(\int q(z) \frac{p(x, z|\varphi)}{q(z)} dz \right)$$

Then by Jensen's inequality:

$$\log \left(\int q(z) \frac{p(x, z|\varphi)}{q(z)} dz \right) \geq \int q(z) \log \left(\frac{p(x, z|\varphi)}{q(z)} \right) dz$$

where the right-hand side is termed the **Evidence Lower BOund** or **ELBO**.

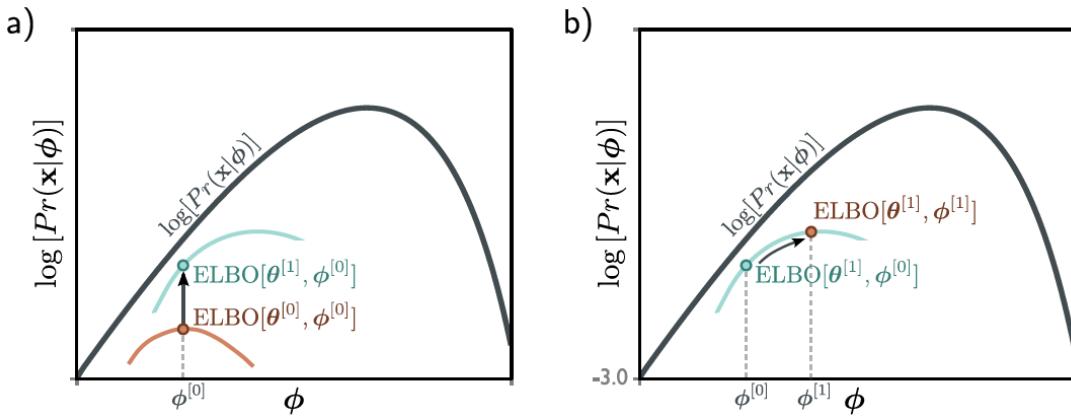
Note: It is referred to as the "evidence" because the marginal likelihood $p(x|\varphi)$ is also known as the **evidence** in the context of Bayes' rule.

In practice, the distribution $q(z)$ has parameters θ , so the ELBO can be written as:

$$ELBO[\theta, \varphi] = \int q(z|\theta) \log\left(\frac{p(x, z|\varphi)}{q(z|\theta)}\right) dz$$

Therefore, to learn the nonlinear latent variable model, we can now aim to **maximize** this bound with respect to both φ and θ .

To better understand the intuition behind ELBO, consider that the original log-likelihood of the data is a function of the parameters φ and that we want to find its maximum. For any fixed θ , the ELBO is still a function of parameters, but one that must lie below the original likelihood function. When we change θ , we modify this function and, depending on our choice, the lower bound may move closer or further from the log-likelihood (Figure a). Instead, when we change φ , we move along the lower bound function (Figure b).



The ELBO is **tight** when, for a fixed value of φ , the ELBO and the likelihood function coincide. To find the distribution $q(z|\theta)$ that makes the bound tight, we can factor the numerator of the log term in the ELBO using the definition of conditional probability we have discussed before:

$$\begin{aligned} ELBO[\theta, \varphi] &= \int q(z|\theta) \log\left(\frac{p(x, z|\varphi)}{q(z|\theta)}\right) dz = \int q(z|\theta) \log\left(\frac{p(z|x, \varphi)p(x|\varphi)}{q(z|\theta)}\right) dz \\ &= \int q(z|\theta) \log(p(x|\varphi)) dz + \int q(z|\theta) \log\left(\frac{p(z|x, \varphi)}{q(z|\theta)}\right) dz \\ &= \log(p(x|\varphi)) + \int q(z|\theta) \log\left(\frac{p(z|x, \varphi)}{q(z|\theta)}\right) dz \\ &= \log(p(x|\varphi)) - D_{KL}(q(z|\theta)||p(z|x, \varphi)) \end{aligned}$$

Here, the first integral disappears between the second and third lines since $\log(p(x|\varphi))$ does not depend on z (recall that we are marginalizing over z) and the integral of the probability distribution $q(z|\theta)$ is 1. In the last line, we simply used the definition of the KL divergence.

This equation shows that the ELBO is the **original log-likelihood $\log(p(x|\varphi))$ minus the KL divergence $D_{KL}(q(z|\theta)||p(z|x, \varphi))$** . We recall the KL divergence measures the "distance" between the distributions and can only take **non-negative values** (≥ 0). It follows that the **ELBO is a lower bound on $\log(p(x|\varphi))$** . The KL distance will be 0 and the lower bound will be "tight" when $q(z|\theta) = p(z|x, \varphi)$. The quantity $p(z|x, \varphi)$ is the **posterior** distribution on the latent variables z given the data observations x ; it tells us which values of the latent variable may have been responsible for the data point.

The ELBO equation can also be expressed in another way by considering the lower bound as the reconstruction error minus the distance from the prior:

$$\begin{aligned} \text{ELBO}[\theta, \varphi] &= \int q(z|\theta) \log\left(\frac{p(x, z|\varphi)}{q(z|\theta)}\right) dz = \int q(z|\theta) \log\left(\frac{p(x|z, \varphi)p(z)}{q(z|\theta)}\right) dz \\ &= \int q(z|\theta) \log(p(x|z, \varphi)) dz + \int q(z|\theta) \log\left(\frac{p(z)}{q(z|\theta)}\right) dz \\ &= \int q(z|\theta) \log(p(x|z, \varphi)) dz - D_{KL}(q(z|\theta)||p(z)) \end{aligned}$$

where the joint distribution $p(x|z, \varphi)$ has been factored into the conditional probability $p(x|z, \varphi)p(z)$ in the first row and the definition of KL divergence is used again in the last row. This formulation is the one used in the variational autoencoder!

In this, the first term measures how well the data distribution $p(x|z, \varphi)$ aligns with the observed data x for all possible values of the latent variables z and is called **reconstruction loss**. The second term measures the degree of correspondence between the auxiliary distribution $q(z|\theta)$ and the prior $p(z)$ and is called **regularization loss**.

Relation to the EM algorithm: Above we saw that the bound is tight when the distribution $q(z|\theta)$ matches the true posterior distribution $p(z|x, \varphi)$. So, one may ask why not use the **Expectation-Maximization (EM)** algorithm to find such a solution? In principle, we could alternate between:

- i. fixing φ and updating θ so that $q(z|\theta)$ equals the posterior $p(x|z, \varphi)$
- ii. fixing θ and updating φ to maximize the ELBO.

This approach is viable in models like the mixture of Gaussians, where the posterior distribution can be computed in closed form. However, in the case of nonlinear latent variable models — such as the one used in VAEs — the posterior does not have a tractable analytical form and so this method is inapplicable.

Variational Inference

We previously saw that the ELBO becomes tight when the distribution $q(z|\theta)$ is equal to the posterior $p(x|z, \varphi)$. In principle, we could compute the posterior using **Bayes's rule**:

$$p(z|x, \varphi) = \frac{p(x|z, \varphi)p(z)}{p(x|\varphi)}$$

However, in practice this is **intractable** because the denominator $p(x|\varphi)$, i.e. our marginal likelihood, involves an integral over all possible values of z , which is generally not analytically solvable.

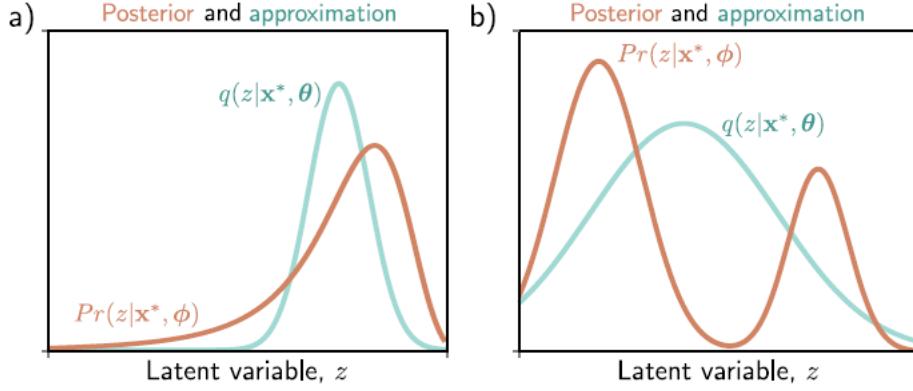
To address this, one solution could be to find an **approximation** using a procedure known as **variational inference** (sometimes also referred to as **amortized inference**), a technique that approximates the true posterior with a simpler, tractable distribution. Specifically, we select a **variational family** Q (Gaussian distributions in the case of VAEs) and aim to find the member of this family — denoted as $q(z|\theta)$ and called **variational distribution** — that is closest to the true posterior $p(x|z, \varphi)$ in terms of KL divergence.

A common choice for $q(z|\theta)$ is a multivariate normal distribution with a diagonal covariance matrix, since it is computationally efficient and interpretable. That is:

$$q(z|\theta) = \mathcal{N}_z(\mu, \text{diag}(\sigma^2))$$

This approximated distribution is not always going to be a great match to the posterior but will be better for some values of μ and σ^2 than others. For instance, the Figure below illustrate two scenarios:

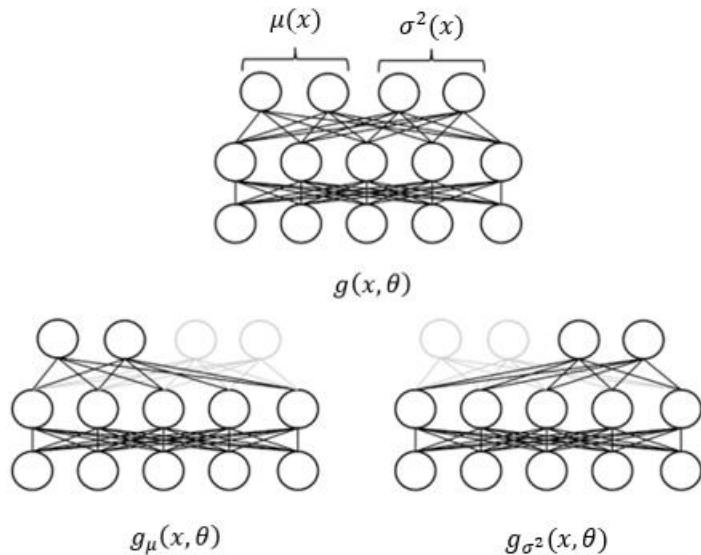
- a) sometimes, the approximation (cyan curve) is good and lies close to the true posterior (orange curve).
- b) however, if the posterior is multi-modal, then the Gaussian approximation will be poor.



To adapt the variational distribution to each data point we make it **data-dependent**: we use a neural network g with parameters θ , to predict the values of μ and σ^2 based on the observed data x . In other words, the neural network takes the data x as input and outputs the parameters μ and σ^2 of the variational distribution $q(z|\theta)$. Since the optimal choice for $q(z|\theta)$ was the posterior $p(z|x)$, and this depends on the data example x , the variational approximation should do the same, so we choose:

$$q(z|x, \theta) = \mathcal{N}_z(g_\mu(x, \theta), g_{\sigma^2}(x, \theta))$$

where $g_\mu(x, \theta)$ and $g_{\sigma^2}(x, \theta)$ correspond to the predicted mean and covariance for the variational distribution. These functions can either be two entirely separate neural networks or may share some subset of parameters. In VAE they are generally implemented as a single network that shares a common set of layers and then **branches at the end into two outputs**: one producing the mean μ and the other the covariance σ^2 for the data point x considered. For simplicity, we may refer to the entire network as $g(x, \theta)$, encompassing both $g_\mu(x, \theta)$ and $g_{\sigma^2}(x, \theta)$, as illustrated in the Figure below:



The Variational Autoencoder

We are now ready to describe the VAE. The goal is to train a model that **maximizes** the **ELBO**:

$$ELBO[\theta, \varphi] = \int q(z|x, \theta) \log[p(x|z, \varphi)] dz - D_{KL}[q(z|x, \theta) || p(z)]$$

The first term still involves an intractable integral, but since it is an expectation with respect to $q(z|x, \theta)$, we can approximate it by sampling. More specifically, we can try to approximate the integral over z with a simple **Monte Carlo estimate**:

$$\int q(z|x, \theta) \log[p(x|z, \varphi)] dz = \mathbb{E}_{q(z|x, \theta)}[\log(p(x|z, \varphi))] \approx \frac{1}{L} \sum_{l=1}^L \log p(x|z^{(l)*}, \varphi)$$

where $z^{(l)*}$ is the l -th sample extracted from $q(z|x, \theta)$ over the total L extracted. For a very approximate estimate, we could just use a single sample z^* from $q(z|x, \theta)$:

$$\mathbb{E}_{q(z|x, \theta)}[\log(p(x|z, \varphi))] \approx \log p(x|z^*, \varphi)$$

Using a single sample ($L = 1$) leads to a **high-variance estimate**, but this is often mitigated by averaging over **large mini-batches** during training (until now we were considering a single data sample x , but later we will consider a mini-batch of the training set). It is true that increasing the number of samples L can reduce the variance of the estimate, however it comes at the cost of slower training and, in some cases, may even hinder optimization. The authors in the original VAE paper found that using a single sample to approximate the expectation $\mathbb{E}_{q(z|x, \theta)}[\log(p(x|z, \varphi))]$ is sufficient when training with stochastic gradient descent and large enough batches. For this reason, using $L = 1$ remains a **standard choice** not only in the original paper, but also in most of the later VAE-based variants.

The second term is the KL divergence between the variational distribution $q(z|x, \theta) = \mathcal{N}_z(g_\mu(x, \theta), diag(g_{\sigma^2}(x, \theta)))$, which here we rewrite for simplicity in notation as $q(z|x, \theta) = \mathcal{N}_z(\mu(x), diag(\sigma^2(x)))$, and the prior $p(z) = \mathcal{N}_z(0, I)$, and can be evaluated in **closed form** analytically as:

$$D_{KL}[q(z|x, \theta) || p(z)] = -\frac{1}{2} \sum_{k=1}^K \left(1 + \log(\sigma_k^2(x)) - \mu_k^2(x) - \sigma_k^2(x) \right)$$

where K is the dimension of the latent space. Note: Here I skipped the mathematical steps on how this closed-form formulation is obtained as it is not that important for the continuation of the discussion.

To summarize, we aim to build a model that computes the ELBO for a data point x . To evaluate the ELBO we:

- compute the mean $\mu(x)$ and variance $\sigma^2(x)$ of the variational distribution $q(z|x, \theta)$ for this data point x using the network $g(x, \theta)$,
- draw a sample z^* from this distribution

$$z^* \sim q(z|x, \theta) = \mathcal{N}_z(\mu(x), diag(\sigma^2(x)))$$

and we pass it through the network $f(z^*, \varphi)$ to compute $p(x|z^*, \varphi)$

- finally, we compute the ELBO using

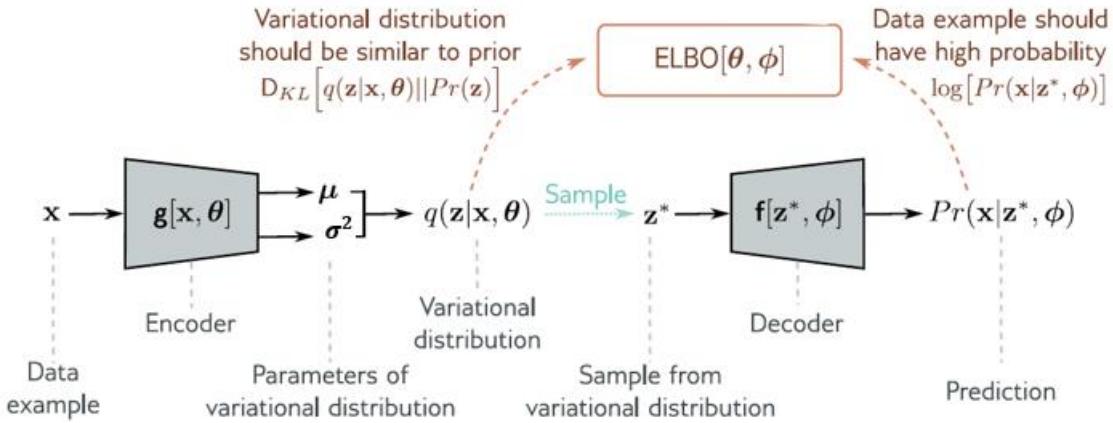
$$ELBO[\theta, \varphi] \approx \log p(x|z^*, \varphi) + \frac{1}{2} \sum_{k=1}^K (1 + \log(\sigma(x)_k^2) - \mu(x)_k^2 - \sigma(x)_k^2)$$

Although this formulation applies to a single data point x , in practice we aim to **maximize the expected ELBO over the entire dataset**. This is typically done by computing the average ELBO over **mini-batches** of samples during training. When training on a batch of N samples $\{x^{(i)}\}_{i=1}^N$, the total loss we want to maximize becomes:

$$\mathcal{L}(\theta, \varphi) = \sum_{i=1}^N \left(\frac{1}{L} \sum_{l=1}^L \log p(x^{(i)} | z^{(il)*}, \varphi) + \frac{1}{2} \sum_{k=1}^K \left(1 + \log (\sigma(x^{(i)})_k^2) - \mu(x^{(i)})_k^2 - \sigma(x^{(i)})_k^2 \right) \right)$$

where the sum over L is kept in this formulation just for generality, though in practice we often set $L = 1$.

The architecture associated with the model we build so far is shown in Figure below:



It should now be clear why this is called a variational autoencoder. It is “**variational**” because it computes a Gaussian approximation of the posterior distribution through the *variational inference* procedure. It is an “**autoencoder**” because it starts with a data point x , computes a lower-dimensional latent vector z from this and then uses this vector to recreate the data point x as closely as possible. In this context, the mapping from the data to the latent variable through the network $g(x, \theta)$ is called the **encoder** and the mapping from the latent variable to the data through the network $f(x, \varphi)$ is called the **decoder**. The VAE computes the ELBO as a function of both φ and θ and by **maximizing it** jointly trains both the encoder and decoder. The gradients with respect to φ and θ are evaluated using automatic differentiation. The model parameters are then updated iteratively through optimization algorithms such as SGD or Adam, by passing mini-batches of data through the model. Once the model is trained, the encoder network is discarded and new data points are generated by sampling from the prior $p(z)$ and forward propagating them through the decoder network to obtain samples in the data space.

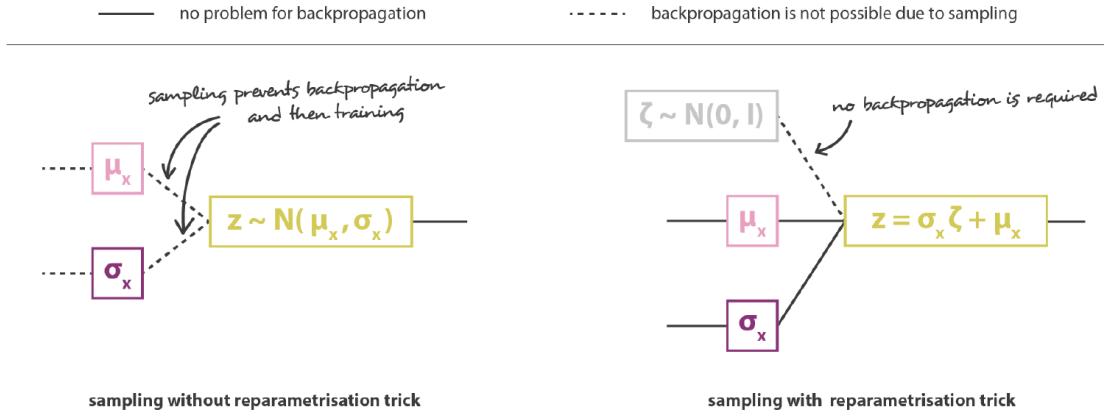
Re-parametrization Trick

There is one more complication; the network involves a sampling step from the variational distribution which makes it difficult to differentiate through this stochastic component. However, differentiating past this step is necessary to update the parameters θ that preceded it in the network.

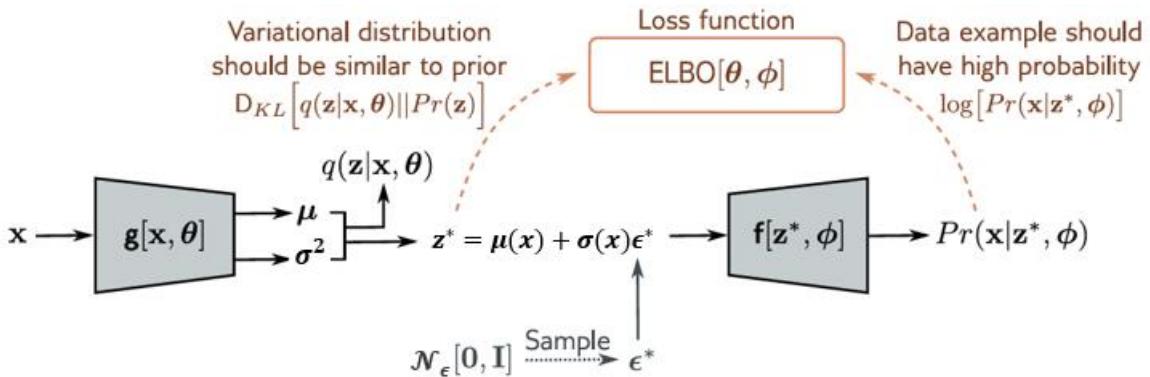
Fortunately, this problem can be solved using the so-called **re-parametrization trick**. This technique consists in separating the sampling process into a separate branch of the network that draws a sample ϵ^* from a standard distribution $\mathcal{N}_\epsilon(0, I)$ (i.e. we can use the prior $p(z)$) and then use the relation:

$$z^* = \sigma(x) \odot \epsilon^* + \mu(x)$$

to obtain a sample from the desired distribution. Now we can compute the derivatives as usual because the backpropagation algorithm **does not need to pass down the sampling branch** (see Figure).



The final VAE architecture looks like the one shown in the following Figure.



In light of this, therefore, to compute the ELBO we:

- compute the mean $\mu(x)$ and variance $\sigma^2(x)$ of the variational distribution $q(z|x, \theta)$ for the given data point x using the encoder network $g(x, \theta)$,
- extract a sample ϵ^* from the prior $\mathcal{N}_\epsilon(0, I)$ and combine it with the predicted parameters $\mu(x)$ and $\sigma^2(x)$ using the *re-parametrization trick* to obtain a sample from the variational distribution z^*
- pass z^* through the decoder network $f(z^*, \phi)$ to compute $p(x|z^*, \phi)$,

Finally, I want to underline that, when working with image data, the networks used to implement a VAE typically involve **convolutional layers** for the encoder and **transpose convolutions** for the decoder.

Comparison of VAE Loss and Standard Autoencoder Loss

To better understand how a VAE differs from a standard autoencoder, it's useful to directly compare their respective loss functions.

The VAE loss function for a batch of N data points can be defined in its more general form as:

$$\mathcal{L}_{VAE}(\theta, \phi) = \sum_{i=1}^N \left(\mathbb{E}_{q(z^{(i)}|x^{(i)}, \theta)} [\log(p(x^{(i)}|z^{(i)}, \phi))] - D_{KL}[q(z^{(i)}|x^{(i)}, \theta) || p(z^{(i)})] \right)$$

where we saw that the expectation is typically approximated using a single sample $z^{(i)*}$ drawn from $q(z^{(i)}|x^{(i)}, \theta)$ accordingly to re-parametrization trick:

$$\mathcal{L}_{VAE}(\theta, \varphi) = \sum_{i=1}^N \left(\log(p(x^{(i)}|z^{(i)*}, \varphi)) - D_{KL}[q(z^{(i)*}|x^{(i)}, \theta)||p(z^{(i)*})] \right)$$

In order to interpret this expression more concretely, let's unpack the log-likelihood term, $\log(p(x^{(i)}|z^{(i)*}, \varphi))$. We recall that we modeled it as a Gaussian distribution with mean given by the decoder $f(z^{(i)*}, \varphi)$ and fixed spherical variance $\sigma^2 I$:

$$p(x^{(i)}|z^{(i)*}, \varphi) = \mathcal{N}_{x^{(i)}}(f(z^{(i)*}, \varphi), \sigma^2 I) = \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left(-\frac{1}{2\sigma^2} \|x^{(i)} - f(z^{(i)*}, \varphi)\|^2\right)$$

From this, the log-likelihood becomes:

$$\begin{aligned} \log(p(x^{(i)}|z^{(i)*}, \varphi)) &= -\frac{1}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \|x^{(i)} - f(z^{(i)*}, \varphi)\|^2 \\ &= \text{const} - \frac{1}{2\sigma^2} \|x^{(i)} - f(z^{(i)*}, \varphi)\|^2 \end{aligned}$$

The first term is constant with respect to the model parameters, so it doesn't affect optimization. Therefore, it can be ignored and so we get:

$$\log(p(x^{(i)}|z^{(i)*}, \varphi)) \approx -\frac{1}{2\sigma^2} \|x^{(i)} - f(z^{(i)*}, \varphi)\|^2$$

Now, the KL divergence term compares the approximate variational posterior $q(z^{(i)*}|x^{(i)}, \theta) = \mathcal{N}_z(g_\mu(x^{(i)}, \theta), \text{diag}(g_{\sigma^2}(x^{(i)}, \theta)))$ against the prior $p(z^{(i)*}) = \mathcal{N}_z(0, I)$, so we can also write it more explicitly as:

$$D_{KL}[q(z^{(i)*}|x^{(i)}, \theta)||p(z^{(i)*})] = D_{KL}\left[\mathcal{N}_z\left(g_\mu(x^{(i)}, \theta), \text{diag}(g_{\sigma^2}(x^{(i)}, \theta))\right)||\mathcal{N}_z(0, I)\right]$$

Therefore, putting these terms together, the VAE can be also thought of as to **maximize** this loss:

$$\max_{\theta, \varphi} \mathcal{L}_{VAE} = \max_{\theta, \varphi} \sum_{i=1}^N \left(-\frac{1}{2\sigma^2} \|x^{(i)} - f(z^{(i)*}, \varphi)\|^2 - D_{KL}\left[\mathcal{N}_z\left(g_\mu(x^{(i)}, \theta), \text{diag}(g_{\sigma^2}(x^{(i)}, \theta))\right)||\mathcal{N}_z(0, I)\right] \right)$$

which can be transformed into a **minimization** problem by changing the sign:

$$\min_{\theta, \varphi} \mathcal{L}_{VAE} = \min_{\theta, \varphi} \sum_{i=1}^N \left(\underbrace{\frac{1}{2\sigma^2} \|x^{(i)} - f(z^{(i)*}, \varphi)\|^2}_{\text{reconstruction loss}} + \underbrace{D_{KL}\left[\mathcal{N}_z\left(g_\mu(x^{(i)}, \theta), \text{diag}(g_{\sigma^2}(x^{(i)}, \theta))\right)||\mathcal{N}_z(0, I)\right]}_{\text{regularization loss}} \right)$$

Now, let's compare this with the loss function used in a standard autoencoder. The AE tries to directly minimize the reconstruction error between the input and the decoder output:

$$\min_{\theta, \varphi} \mathcal{L}_{AE} = \min_{\theta, \varphi} \frac{1}{2} \sum_{i=1}^N \underbrace{\|x^{(i)} - f(g(x^{(i)}, \theta), \varphi)\|^2}_{\text{reconstruction loss}}$$

where $g(\cdot, \theta)$ is the encoder network mapping $x^{(i)}$ to a latent code and $f(\cdot, \varphi)$ is the decoder network reconstructing $x^{(i)}$ from that code.

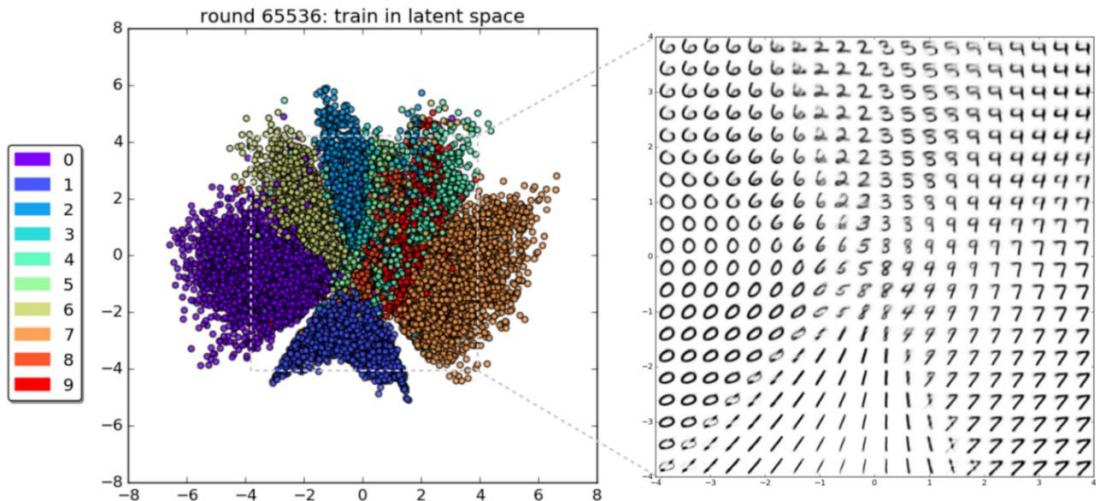
Both VAE and AE, therefore, aim to minimize reconstruction error expressed as a squared error difference between the original data point and the decoded data point. In this sense, they appear quite similar. However, the key

distinction lies in how each model treats the latent space: while AEs rely on a **deterministic encoding**, where each input maps to a fixed latent vector, VAEs use a **stochastic latent space** in which each input is mapped to a distribution over latent variables, from which a sample is drawn during training.

Moreover, the VAE also introduces a **regularization term** in the form of the KL divergence, which measures how much the approximate posterior distribution $\mathcal{N}_z\left(g_\mu(x^{(i)}, \theta), \text{diag}(g_{\sigma^2}(x^{(i)}, \theta))\right)$ deviates from the chosen prior $\mathcal{N}_z(0, I)$. This term therefore acts as a **regularizer**, preventing the encoder from arbitrarily shaping the latent space and encouraging instead the latent representations to stay close to the prior. This key difference gives the VAE desirable generative properties, such as the ability to generate coherent samples by simply drawing from a prior defined as $\mathcal{N}_z(0, I)$.

Final Considerations and Applications of VAEs

The Figure below shows how a VAE trained on MNIST is able to learn **continuous latent representations** of handwritten digits. This continuity means that interpolating between the latent representations of two digits yields smooth transitions; intermediate latent codes generate digits that look like blends of the two, while points farther from high-density regions produce outputs that are visually distorted.



While VAEs have seen significant development, their popularity and adoption have been somewhat overshadowed by the explosive growth of GANs. One reason for this is the higher degree of complexity in VAEs theoretical basis compared to the simplicity of the adversarial training concept that rules GANs. Nonetheless, VAEs offer unique advantages grounded in probabilistic modeling. First it is much **easier to train and to evaluate** with respect to a GAN, given that it is based on **data likelihood** instead of less grounded adversarial training.

Second, it shows greater ability to learn meaningful latent representations that can be manipulated in surprisingly flexible and interpretable ways. For instance, VAEs support meaningful **semantic interpolation**. After training on the CelebA dataset we can perform linear operations in the latent space. By computing the average latent code for "blond hair" faces and subtracting the average code for "non-blond hair" faces, we can obtain a direction Δ representing this transformation. Adding increasing multiples of Δ to a latent code of a neutral face results in a gradual transition to blond hair



Similar manipulations can also gradually add sunglasses to a face



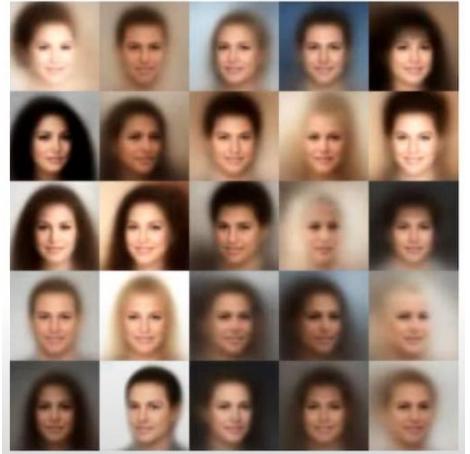
or combine multiple attributes like gradually changing hair color and adding sunglasses simultaneously:



Beyond interpolation, VAEs can be used for **resynthesis and attribute manipulation**. In this case, an input image is projected into the latent space using the encoder, and its corresponding mean vector is used as a representation. By navigating this space — e.g., manipulating the latent space in directions representing smiling/neutral (horizontal) and mouth open/closed (vertical) — we can generate new versions of the same face with altered expressions, as shown in the Figure below.



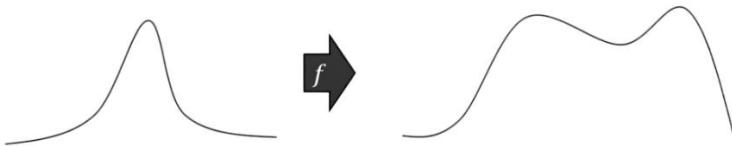
During training of VAEs, one common issue is **posterior collapse**. This happens when the variational distribution $q(z|x, \theta)$ converges to the prior $p(z)$, effectively ignoring the input data x . In this situation, the latent variable z carries little to no information about the data, and the model relies almost entirely on the decoder. A symptom of this phenomena is that reconstructions tend to be **blurry** (see Figure). Mathematically, this corresponds to the KL divergence term shrinking towards zero, i.e. $D_{KL}[q(z|x, \theta) || p(z)] \approx 0$.



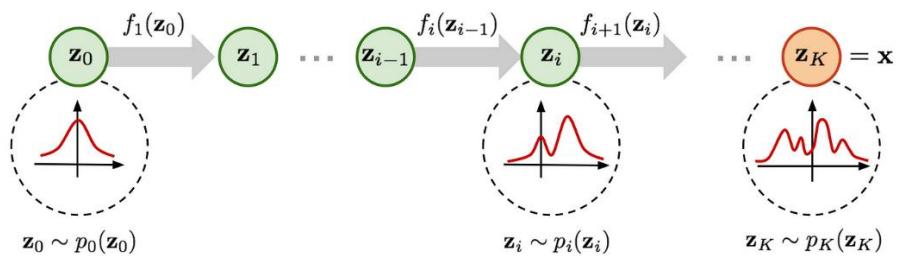
At the opposite extreme, another problem arises when the latent code is **not sufficiently regularized**. Here, the encoder perfectly preserves information about the input, leading to very sharp and accurate reconstructions. However, because the approximate posterior $q(z|x, \theta)$ deviates significantly from the prior $p(z)$, the model fails to generate meaningful samples when drawing from the prior. In this case, the KL divergence is large, and generated samples look unrealistic.

Both of these issues can be mitigated by introducing a weighting coefficient β to control the regularization effectiveness of the KL divergence, where typically $\beta > 1$ in accordance with **β -VAE** [102]. If the reconstructions look poor then β can be increased, whereas if the samples look poor then β can be decreased.

Despite these strengths, new data samples obtained from **vanilla VAEs** are generally **low-quality**: this is partly because of the naïve Gaussians used to model the variational posterior. In fact, a powerful improvement is to **replace the standard Gaussian variational distribution with a Normalizing flow** [103]. We will talk in detail about it in the next chapter, but the core intuition behind it is straightforward: as we've seen, true posterior distributions are often intractable, requiring us to resort to approximations. However, when there is a significant mismatch between the true distribution and the chosen approximation, the model's performance suffers. In the context of VAEs, if the true posterior is multi-modal or has a complex, irregular shape, approximating it with a single Gaussian — as is done in vanilla VAEs — is insufficient. A single Gaussian cannot adequately capture such complexity. Normalizing flows offer a solution: if we begin with a simple distribution, like a Gaussian, and apply a series of appropriately chosen transformations, we can morph it into a more flexible and expressive density that better matches the true posterior.

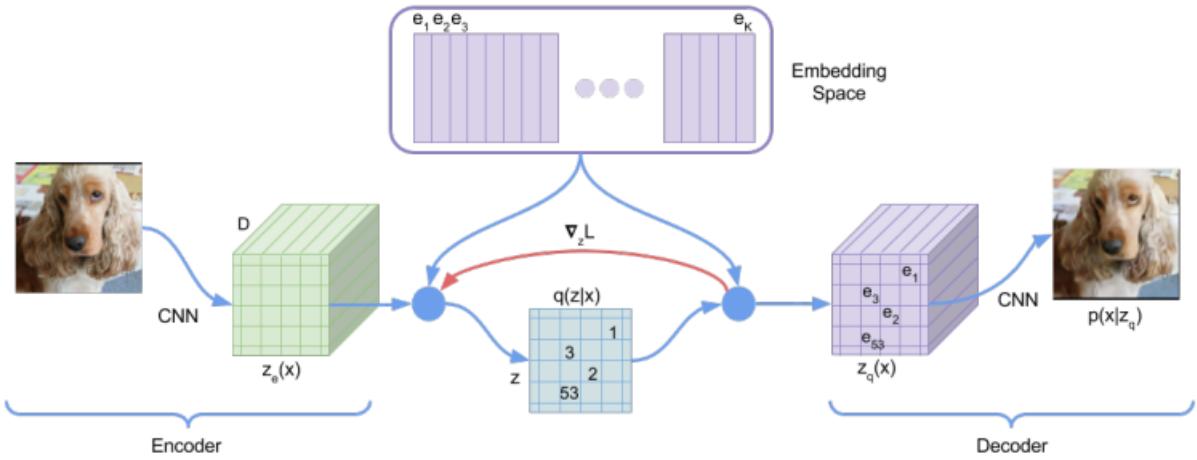


By applying such transformations repeatedly, we can, in principle, approximate arbitrarily complex distributions.



Of course, this raises important questions: what kinds of transformations should be used, and do we ensure the resulting distribution remains valid and tractable for likelihood computation? These are critical design considerations that fall beyond the scope of this chapter, so they will be discussed in depth in the next one.

Finally, a particularly influential extension of VAE is the **Vector Quantized VAE (VQ-VAE)** [104]. Unlike standard VAEs, which rely on continuous latent variables, VQ-VAEs introduce a **discrete latent space** implemented via a **learned codebook** of embeddings. During encoding, each input is mapped to the **nearest entry in the codebook**, effectively compressing the input into a **discrete sequence of tokens**. This discrete representation is especially powerful because it enables models to **combine VAE-like reconstruction with discrete generative modeling techniques** such as autoregressive Transformers.

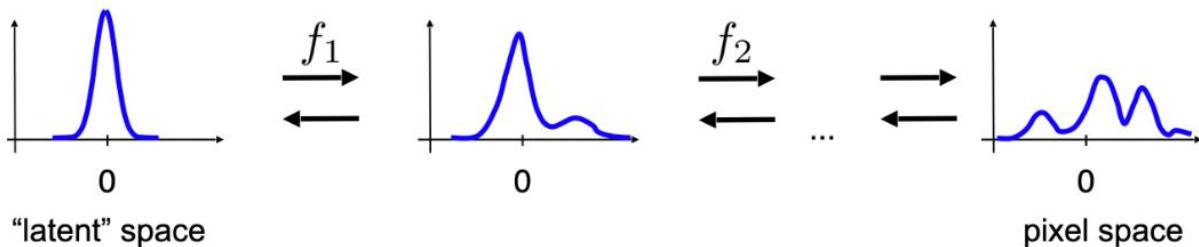


VQ-VAE played a key role in early multimodal models. For example, **DALL-E 1** [105] used a VQ-VAE to encode images into discrete tokens before modeling them with a Transformer, allowing the generation of **high-quality, coherent images from text prompts** — a breakthrough in text-to-image synthesis.

Normalizing Flows

In the previous chapters, we studied two complementary paradigms for generative modeling, **GANs** and **VAEs**: adversarial training with implicit likelihoods (GANs), and variational inference with approximate likelihoods (VAEs). However, both approaches come with limitations. GANs can generate highly realistic samples but lack an explicit likelihood function, making model evaluation and training stability difficult. VAEs, on the other hand, provide tractable likelihood approximations but often struggle with producing sharp, high-quality samples due to their reliance on approximate inference.

Normalizing Flows (NFs) [106] [107] offer a middle ground by constructing models that are both *likelihood-based* and expressive. They allow us to explicitly compute exact log-likelihoods while still generating flexible and realistic samples. At their core, normalizing flows are built upon the idea of transforming a simple probability distribution into a more complex one using a sequence of invertible transformations.



This chapter closely follows the corresponding chapter in the recommended textbook, which provides an excellent explanation of the core concepts. Although Normalizing Flows are not covered in the course lectures, I strongly recommend studying them, as they offer important insights and connections to topics such as **flow matching**, which we will discuss in detail in the last chapter.

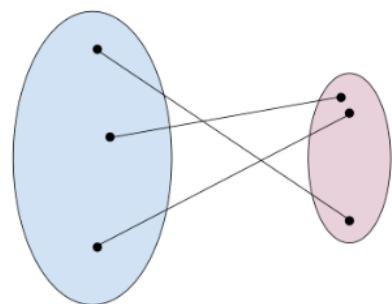
Introducing Normalizing Flows

Suppose we define a distribution $p_z(z)$ over a latent variable z together with a nonlinear transformation $x = f(z, w)$, where f is given by a neural network with parameters w , which transforms the latent space into the data space. Assuming $p_z(z)$ is a simple distribution such as a Gaussian, sampling from such a model is easy as each latent sample $z^* \sim p_z(z)$ is simply passed through the neural network to generate a corresponding data sample $x^* = f(z^*, w)$.

To train such a model, we can think to evaluate its likelihood. However, to do so, we need the corresponding data distribution $p_x(x)$, which requires the **inverse mapping**

$$= f^{-1}(x, w)$$

which exists only if f is **bijective**, i.e. only if it ensures that each value of x corresponds to a unique value of z and viceversa.



If we are able to guarantee that invertibility, we can then use the **change of variables formula** to calculate the data density:

$$p_x(x|w) = p_z(f^{-1}(x, w)) |\det J(x)|$$

where $J(x)$ is the Jacobian of the inverse mapping:

$$J(x) = \begin{bmatrix} \frac{\partial f_1^{-1}(x, w)}{\partial x_1} & \dots & \frac{\partial f_1^{-1}(x, w)}{\partial x_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_D^{-1}(x, w)}{\partial x_1} & \dots & \frac{\partial f_D^{-1}(x, w)}{\partial x_D} \end{bmatrix}$$

Equivalently, this is often written as:

$$p_x(x|w) = p_z(f^{-1}(x, w)) \left| \frac{\partial f^{-1}(x, w)}{\partial x} \right|$$

We will continue to refer to z as a ‘latent’ variable even though the deterministic mapping means that any given data value x corresponds to a unique value of z whose value is therefore no longer uncertain. The mapping function $f(z, w)$ will be defined in terms of a special form of neural network, whose structure we will discuss shortly.

An important implication of invertibility is that the latent space and data space must have the **same dimensionality**, which can lead to large models for high-dimensional data such as images. Moreover, computing the determinant of a general $D \times D$ Jacobian requires $O(D^3)$ operations, so practical flows impose special structure on f to make determinant evaluation more efficient.

Given a training set $\mathcal{D} = \{x^{(i)}\}_{i=1}^N$, the log likelihood function of our model is:

$$\begin{aligned} \ln p(\mathcal{D}|w) &= \sum_{i=1}^N \ln p_x(x^{(i)}, w) = \\ &= \sum_{i=1}^N \left\{ \ln p_z(f^{-1}(x^{(i)}, w)) + \ln |\det J(x^{(i)})| \right\} \end{aligned}$$

Thus, training our neural network amounts to maximizing this likelihood with respect to w .

To be able to model a wide range of distributions, we want the transformation function $x = f(z, w)$ to be highly flexible, and so we use a deep neural network architecture. We can ensure that the overall function is invertible if we make each layer of the network invertible. To see this, consider three successive transformations, each corresponding to one layer, of the form:

$$x = f^A(f^B(f^C(z)))$$

Then the inverse function is given by:

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

Moreover, the determinant of the Jacobian for such a layered structure is also easy to evaluate in terms of the Jacobian determinants for each of the individual layers by making use of the chain rule of calculus:

$$J_{ij} = \frac{\partial z_i}{\partial x_j} = \sum_k \sum_l \frac{\partial f_i^{C^{-1}}}{\partial f_i^{B^{-1}}} \frac{\partial f_i^{B^{-1}}}{\partial f_i^{A^{-1}}} \frac{\partial f_i^{A^{-1}}}{\partial x_j}$$

We recognize the right-hand side as the product of three matrices, and the determinant of a product is **the product of the determinants**. Therefore, the log determinant of the overall Jacobian will be the sum of the log determinants corresponding to each layer:

$$\ln |\det J(x)| = \ln |\det J^A(x)| + \ln |\det J^B(x)| + \ln |\det J^C(x)|$$

This approach to modelling a flexible distribution is called a **Normalizing Flow (NF)** because the transformation of probability densities through successive invertible mappings is somewhat analogous to the flow of a fluid. Also, the effect of the inverse mapping is to transform the complex data distribution into a normalized form, typically a Gaussian or normal distribution.

In the next sections we will discuss the core concepts from the two main classes of normalizing flows used in practice: **coupling flows** and **autoregressive flows**. We will also look at flows that use **neural differential equations** to define invertible mappings, leading to **continuous flows**.

Coupling Flows

Our goal is to design a single invertible transformation layer, so that by composing many such layers we obtain a highly flexible class of invertible functions. Consider first a simple linear transformation of the form:

$$x = az + b$$

This is easy to invert, giving:

$$z = \frac{1}{a}(x - b)$$

However, linear transformations are closed under composition, meaning that a sequence of linear transformations is equivalent to a single overall linear transformation. Furthermore, applying a linear transformation to a Gaussian distribution produces another Gaussian. Thus, no matter how many linear layers we stack, the resulting distribution remains Gaussian.

The question is whether we can retain the invertability of a linear transformation while allowing additional flexibility so that the resulting distribution can be non-Gaussian. One solution to this problem is given by a form of normalizing flow model called **real NVP** [108], which is short for ‘*real-valued non-volume-preserving*’. The idea is to partition the latent-variable vector z into two parts $z = (z_A, z_B)$, so that if z has dimension D and z_A has dimension d , then z_B has dimension $D - d$. The output vector x is similarly partitioned, i.e. $x = (x_A, x_B)$ where x_A has dimension d and x_B has dimension $D - d$. For the first part of the output vector, we simply copy the input:

$$x_A = z_A$$

The second part of the vector undergoes a linear transformation, but now the coefficients in the linear transformation are given by nonlinear functions of z_A :

$$x_B = \exp(s(z_A, w)) \odot z_B + b(z_A, w)$$

where $s(z_A, w)$ and $b(z_A, w)$ are the real-valued outputs of neural networks, and the exponential ensures that the multiplicative term is non-negative. Note that we have shown the same vector w in both network functions. In practice, these may be implemented as separate networks with their own parameters, or as one network with two sets of outputs ([similary to what we have seen for the VAE's encoder](#)).

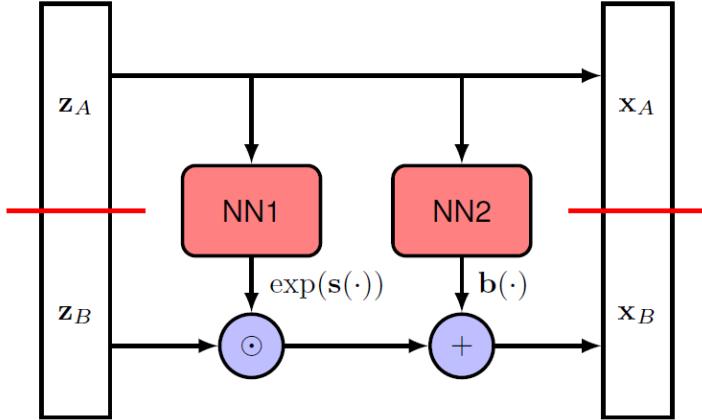
Due to the use of neural network functions, the value of x_B can be a very flexible function of x_A . Nevertheless, the overall transformation is easily invertible: given a value for $x = (x_A, x_B)$ we first compute

$$z_A = x_A$$

then we evaluate $s(z_A, w)$ and $b(z_A, w)$, and finally we compute z_B using

$$z_B = \exp(-s(z_A, w)) \odot (x_B - b(z_A, w))$$

The overall transformation is illustrated in Figure below. Note that there is no requirement for the individual neural network functions $s(z_A, w)$ and $b(z_A, w)$ to be invertible.



Importantly, the logarithm of the Jacobian-determinant is easy-to-calculate, given that for this transformation we have:

$$J = \begin{bmatrix} I_{d \times d} & 0_{d \times (D-d)} \\ \frac{\partial z_B}{\partial z_A} & \text{diag}(\exp(-s(x_A, w))) \end{bmatrix}$$

where:

- The **top left block** corresponds to the derivatives of z_A with respect to x_A and hence is given by the $d \times d$ identity matrix.
- The **top right block** corresponds to the derivatives of z_A with respect to x_B and these terms vanish.
- The **bottom left block** corresponds to the derivatives of z_B with respect to x_A and contains complicated expressions involving the neural network functions.
- The **bottom right block** corresponds to the derivatives of z_B with respect to x_B , which are given by a diagonal matrix whose diagonal elements are given by the exponentials of the negative elements of $s(x_A, w)$.

We therefore see that the Jacobian matrix so obtained is a **lower triangular** matrix, meaning that all elements above the leading diagonal are zero. For such a matrix, the determinant is just the product of the elements along the leading diagonal, and therefore it does not depend on the complicated expressions in the lower left block. Consequently, the determinant of the Jacobian is simply given by the product of the elements of $\exp(-s(x_A, w))$:

$$\det(J) = \prod_{j=1}^{D-d} \exp(-s(x_A, w))_j = \exp\left(\sum_{j=1}^{D-d} -s(x_A, w)_j\right)$$

A clear limitation of this type of transformation is that the value of z_A is unchanged by the transformation. However, this is easily resolved by adding another layer in which the roles of z_A and z_B are reversed, as illustrated in Figure below. This double-layer structure can then be repeated multiple times to facilitate a very flexible class of generative models.

Training proceeds by maximizing the log-likelihood. We first create mini-batches of data points. Then for each single data point x , the contribution to the log-likelihood is obtained from the change-of-variables formula:

$$\begin{aligned} \ln p_x(x|w) &= \ln p_z(f^{-1}(x, w)) |\det J(x)| = \\ &= \ln p_z(f^{-1}(x, w)) + \ln |\det J(x)| \end{aligned}$$

If the base distribution is a standard Gaussian $p_z(z) = \mathcal{N}(0, I)$, then we know that the log density is given by:

$$\ln p_z(z) = \text{const.} - \frac{1}{2} \|z\|^2$$

where the inverse transformation $z = f^{-1}(x, w)$ is obtained by applying a sequence of inverse coupling layers.

Note: We recall that for a general Gaussian with mean μ and variance σ^2 , the log density is

$$\ln p_z(z) = \text{const.} - \frac{1}{2\sigma^2} \|z - \mu\|^2$$

Which in the special case of a standard Gaussian ($\mu = 0, \sigma^2 = 1$) simplifies to the expression above.

Similarly, the log of the Jacobian determinant is given by a sum of log determinants for each layer where each term is itself a sum of terms of the form $-s_l(x_A, w)$:

$$\ln|\det J(x)| = \sum_{l=1}^L \sum_{j=1}^{D-d} (-s_l(x_A, w))_j$$

where L indicates the number of coupling layers.

Putting everything together, the log-likelihood over a batch can be defined as:

$$\max_{\theta} \mathcal{L}(w) \quad \text{where } \mathcal{L}(w) = \sum_{i=1}^N \log p_x(x^{(i)}, w) = \sum_{i=1}^N \left[-\frac{1}{2} f^{-1}(x^{(i)}, w) + \sum_{l=1}^L \sum_{j=1}^{D-d} (-s_l(x_A^{(i)}, w))_j \right]$$

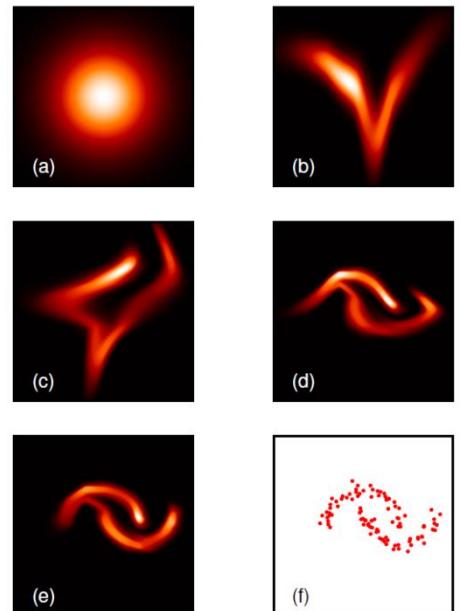
Gradients of the log likelihood can be evaluated using automatic differentiation, and the network parameters updated by stochastic gradient descent (or one of its variants).

The real NVP model is part of a broader class of normalizing flows called **coupling flows**, in which the linear transformation we saw above for x_B is replaced by a more general form:

$$x_B = h(z_B, f^{-1}(z_A, w))$$

where $h(z_B, f^{-1})$ is a function of z_B that is efficiently invertible for any given value of f^{-1} and is called the *coupling function*. The function $f^{-1}(z_A, w)$ is called a *conditioner* and is typically represented by a neural network.

We can illustrate the real NVP normalizing flow using the well-known *two moons* dataset (see Figure on side). Here a two-dimensional Gaussian distribution is transformed into a more complex distribution by using two successive layers each of which consists of alternate transformations on each of the two dimensions. In particular, the Figure shows (a) the Gaussian base distribution, (b) the distribution after a transformation of the vertical axis only, (c) the distribution after a subsequent transformation of the horizontal axis, (d) the distribution after a second transformation of the vertical axis, (e) the distribution after a second transformation of the horizontal axis and (f) the final two-moons dataset on which the model was trained.



Autoregressive Flows

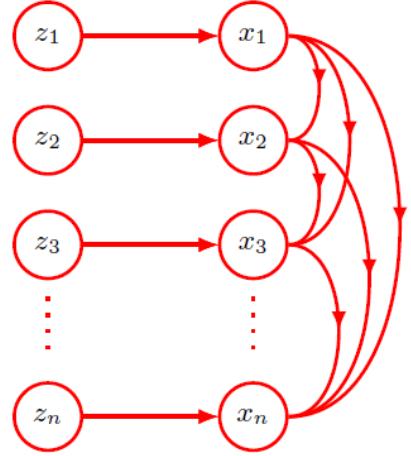
A related formulation of normalizing flows can be motivated by noting that any joint distribution over a set of variables can always be expressed as the product of conditional distributions, one for each variable. Given an ordering of the variables in the vector $x = (x_1, \dots, x_D)$, we can write:

$$p(x_1, \dots, x_D) = \prod_{n=1}^D p(x_n | x_{1:n-1})$$

where $x_{1:n-1}$ is a notation shorthand for the sequence (x_1, \dots, x_{n-1}) , i.e., all variables from x_1 up to x_{n-1} . This factorization can be used to construct a class of normalizing flow called a **masked autoregressive flow (MAF)** [109], where each variable is defined by

$$x_n = h(z_n, f_n^{-1}(x_{1:n-1}, w_n))$$

Here $h(z_n, \cdot)$ is the coupling function, which is chosen to be easily invertible with respect to z_n , and f_n^{-1} is the conditioner, which is typically represented by a deep neural network. The term “*masked*” refers to the use of a single neural network to implement to represent all these transformations thanks to the adoption of a binary masks that enforce the autoregressive structure by setting certain network weights to zero. The structure of a masked autoregressive flow is illustrated in Figure on side.



In this case the inverse mapping needed to evaluate the likelihood function is given by

$$z_n = h^{-1}(x_n, f_n^{-1}(x_{1:n-1}, w_n))$$

These inverse computations can be parallelized efficiently on modern hardware, since each z_n can be evaluated independently once the conditioners are known. Moreover, the Jacobian matrix of this transformation is **upper triangular**, meaning its determinant reduces to the product of diagonal entries and can therefore also be evaluated efficiently.

However, **sampling** from a MAF is inherently sequential: to compute x_n , all preceding variables x_1, \dots, x_{n-1} must first be evaluated, which slows down generation. To overcome this, one can instead use an **inverse autoregressive flow (IAF)** [110], in which the transformation is defined as

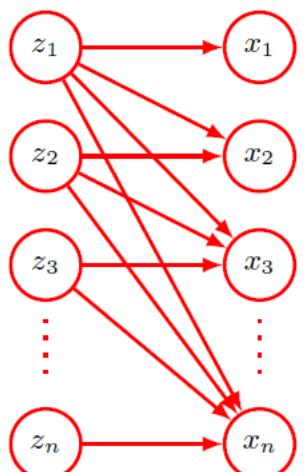
$$x_i = h\left(z_i, \tilde{f}_i^{-1}(z_{1:i-1}, w_i)\right)$$

as illustrated in Figure on side. In this case, sampling is efficient because, for a given choice of z , all x_n can now be computed in parallel. However, computing the inverse transformation (which is needed to evaluate the likelihood)

$$z_i = h^{-1}\left(x_i, \tilde{f}_i^{-1}(z_{1:i-1}, w_i)\right)$$

is now sequential, making likelihood evaluation slow.

Therefore, whether a masked autoregressive flow or an inverse autoregressive flow is preferred depends on the application. For instance, if the goal is **density estimation**, such as modeling the probability of images or text sequences, a **MAF** is more suitable because it allows fast likelihood evaluation. On the other hand, if the task is **generative modeling**, such as producing new realistic images or audio signals, an **IAF** is preferable since it enables fast sampling from the model.



We see that coupling flows and autoregressive flows are closely related. Although autoregressive flows introduce considerable flexibility, this comes with a computational cost that grows linearly in the dimensionality D of the data space due to the need for sequential sampling. Coupling flows can be viewed as a special case of autoregressive flows in which some of this generality is sacrificed for efficiency by dividing the variables into two groups instead of D groups.

Continuous Flows

The final approach to normalizing flows that we consider in this chapter will make use of deep neural networks defined in terms of an **ordinary differential equation (ODE)**. Conceptually, this can be thought of as a neural network with an **infinite number of layers**. We first introduce the concept of a neural ODE then we see how this can be applied to the formulation of a normalizing flow model.

Neural Differential Equations

We have seen that neural networks are especially useful when they comprise many layers of processing, and so we can ask what happens if we explore the limit of an infinitely large number of layers. Consider a residual network where each layer of processing generates an output given by the input vector with the addition of some parameterized nonlinear function of that input vector:

$$z_{t+1} = z_t + f(z_t, w)$$

where $t = 1, \dots, T$ labels the layers in the network. Note that we have used the same function at each layer, with a shared parameter vector w , because this allows us to consider an arbitrarily large number of such layers while keeping the number of parameters bounded. Imagine that we increase the number of layers while ensuring that the changes introduced at each layer become correspondingly smaller. In the limit, the hidden-unit activation vector becomes a function z_t of a continuous variable t , and we can express the evolution of this vector through the network as a differential equation:

$$\frac{dz(t)}{dt} = f(z(t), w)$$

where t is often referred to as ‘time’. This formulation is known as a **Neural ODE** [111].

Note: Here “ordinary” means that there is a single independent variable t , in contrast, for instance, to a partial differential equation, which depends on multiple variables.

If we denote the input to the network by the vector z_0 , then the output z_T is obtained by integrating the ODE:

$$z(T) = \int_0^T f(z(t), w) dt$$

This integral can be evaluated using standard **numerical integration methods**. The simplest is the **Euler’s forward integration method**, which discretizes time into small intervals of size Δt and approximates the integral as:

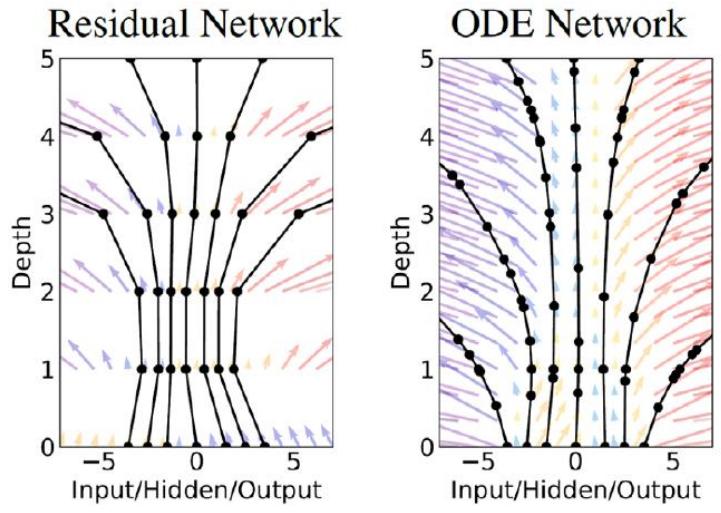
$$z(t + \Delta t) \approx z(t) + \Delta t f(z(t), w)$$

At each step, the change in the hidden state is estimated using the derivative $f(z(t), w)$ evaluated at the current point. However, it is important to note that this is an approximation, introducing an **error** that increases with larger

Δt and a larger number of integration steps. Therefore, smaller step sizes Δt are generally required to maintain accuracy, but this comes at the cost of increased computation.

In practise, more sophisticated **adaptive step-size ODE solvers**, such as [Runge-Kutta Fehlberg](#) or [Dormand-Prince](#) methods, are generally used. These methods dynamically adjust the step size Δt based on the estimated error at each step. This allows the solver to take larger steps in regions where the solution changes slowly and smaller steps where the dynamics are more complex, achieving a balance between accuracy and efficiency. Unlike uniform-step methods, the evaluation points in adaptive solvers are **not evenly spaced** in t .

Therefore, in the context of neural ODEs, the number of function evaluations required by the solver effectively replaces the notion of **network depth** in traditional feedforward networks, with each evaluation corresponding to a “virtual layer”. A comparison between a standard layered neural network and a neural differential equation is illustrated in the Figure on side. On the left, a residual network with five layers is shown, including trajectories for several starting values of a single scalar input. On the right, the continuous neural ODE is visualized for the same starting values of the scalar input. In particular, we can see that the function is not evaluated at uniformly-spaced time intervals, but instead the evaluation points are chosen **adaptively** by the numerical solver, varying depending on the input, in contrast to the uniformly spaced layers of the conventional network.



Neural ODE Flows

We can make use of a neural ordinary differential equation to define an alternative approach to the construction of tractable normalizing flow models. We saw that a neural ODE defines a highly flexible transformation from an input vector $z(0)$ to an output vector $z(T)$ in terms of a differential equation of the form

$$\frac{dz(t)}{dt} = f(z(t), w)$$

If we define a base distribution over the input vector $p(z(0))$, then the neural ODE propagates this distribution forward through time to give a distribution $p(z(t))$ for each value of t , ultimately yielding a distribution over the output vector $p(z(T))$. The authors showed that, for neural ODEs, the evolution of the log-density can be computed by integrating the differential equation:

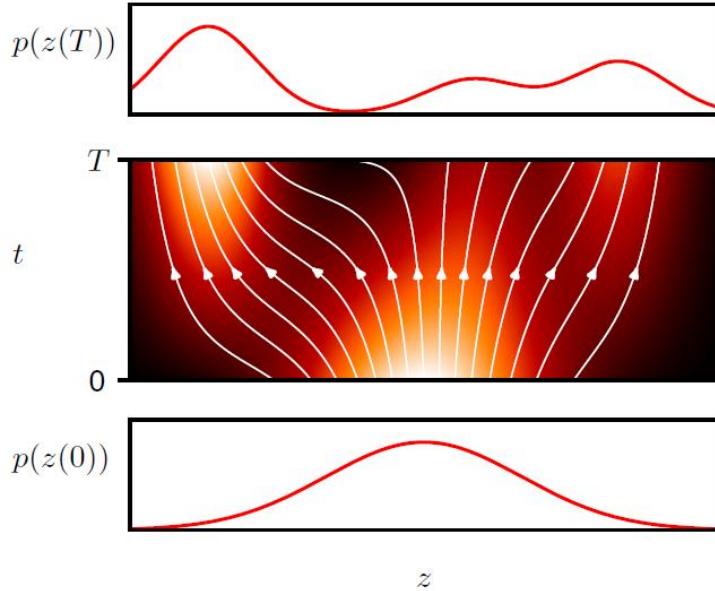
$$\frac{d \ln p(z(t))}{dt} = -\text{Tr}\left(\frac{\partial f}{\partial z(t)}\right)$$

where $\partial f / \partial z(t)$ represents the Jacobian matrix with elements $\partial f_i / \partial z_j$. This integration can be performed using standard ODE solvers.

Note: From linear algebra, we recall that the **trace** of a square matrix is the **sum of its diagonal elements**.

Similarly, samples from the output density can be obtained by sampling from the base distribution $p(z(0))$, which is chosen to be a simple distribution such as a Gaussian, and propagating them to the output by integrating the ODE again using the same ODE solver.

The resulting framework is known as a ***Continuous Normalizing Flow*** and is illustrated in Figure below. Continuous normalizing flows can be trained using the ***adjoint sensitivity*** method for neural ODEs, which can be viewed as the continuous time equivalent of backpropagation ([you can find more of this on the textbook](#)).



Because the evolution of the log-density involves the **trace** of the Jacobian rather than its determinant (as in discrete normalizing flows), this formulation may appear more computationally efficient. In general, computing the determinant of a $D \times D$ matrix requires $O(D^3)$ operations, whereas computing the trace requires only $O(D)$. However, if the determinant is lower diagonal, as in many forms of normalizing flow, then the determinant is the product of the diagonal terms and therefore also involves $O(D)$ operations. Then, since evaluating the individual elements of the Jacobian matrix requires separate forward passes, each costing $O(D)$ operations, computing either the trace or the determinant (for a lower triangular matrix) takes an overall cost of $O(D^2)$ operations. This cost can be further reduced using **Hutchinson's trace estimator** [112], which approximates the trace of a matrix A as

$$Tr(A) = \mathbb{E}_\epsilon[\epsilon^T A \epsilon]$$

where ϵ is a random vector whose distribution has zero mean and unit covariance, for example, a Gaussian $\mathcal{N}(0, I)$. For a given ϵ , the matrix-vector product $A\epsilon$ can be computed efficiently in a single pass using reverse-mode automatic differentiation. We can then approximate the trace using a finite number of samples as:

$$Tr(A) \approx \frac{1}{M} \sum_{m=1}^M \epsilon_m^T A \epsilon_m$$

In practice we can set $M = 1$ and just use a single sample, which is refreshed for each new data point ([similarly to what we saw for VAE](#)). Although this is a noisy estimate, this might not be too significant since it forms part of a noisy stochastic gradient descent procedure. Importantly it is **unbiased**, meaning that the expectation of the estimator is equal to the true value.

Energy-based Models

Energy-Based Models (EBMs), originally conceptualized by Yann LeCun [113], offer a flexible and unified framework for addressing a broad spectrum of ML problems. They model learning problems through the lens of **energy minimization** rather than explicit probability estimation. To do that, EBMs define an **energy function** over **input-output pairs**, where **lower energy corresponds to more plausible configurations** and learning consists in **shaping this energy landscape such that desired outputs are assigned lower energy than undesired ones**.

Importantly, EBMs provide a principled way to approach such problems by **avoiding the need to define normalized probability distributions explicitly**. Instead, they assign **scalar energy scores** to input-output pairs, making it possible to model complex dependencies and structure, in both discrete and continuous domains. This energy-based formulation accommodates tasks that require constraint satisfaction during inference — such as generating grammatically correct translations or semantically coherent predictions — by treating **inference itself as an optimization problem** over the energy landscape searching for configurations that minimize the energy.

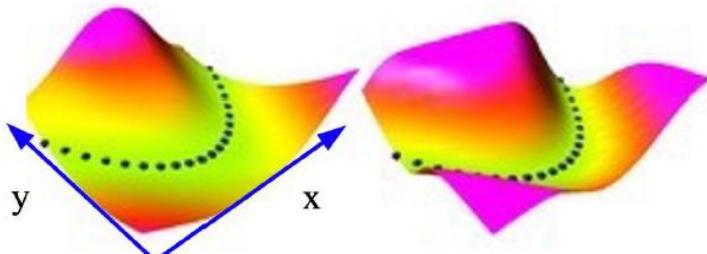
In this chapter, I have chosen to adopt a reverse approach to presenting EBMs, following the structure of the slides. Rather than starting with formal definitions, we will first explore **what kinds of problems EBMs can solve** and the **advantages they bring in addressing the challenges discussed earlier** and only then turn to their **mathematical formulation and training methods**.

There is much to say about EBMs, but due to time constraints — and because a fully unified theoretical framework is still emerging — I will keep the discussion aligned with the slides while adding insights to clarify their **true potential**. EBMs have long been an intriguing research topic and have recently experienced a strong resurgence in interest.

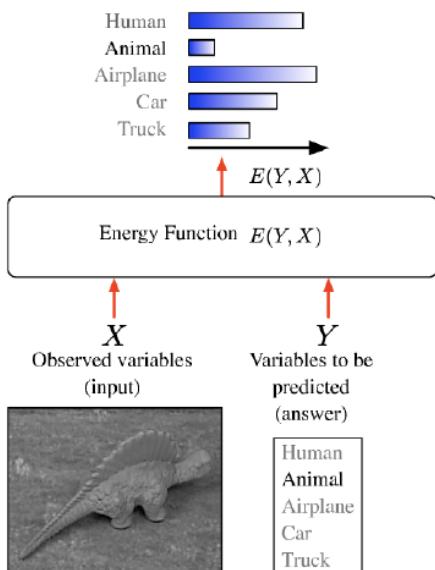
The slides draw inspiration from [113] and from Yann LeCun's [lectures](#) at NYU. For those who want to dive deeper into LeCun's perspective, I highly recommend the recent paper "*Introduction to latent variable energy-based models: a path toward autonomous machine intelligence*" [114], which also introduces **Joint Embedding Predictive Architecture (JEPA)** [115] — a promising direction for representation learning that has emerged in last years.

How an Energy-Based Model Works

EBMs offer a general framework for **modeling the dependencies between variables by assigning a scalar energy value to each configuration of those variables**. These configurations can include pairs such as inputs and outputs or even combinations involving latent variables. The core principle behind EBMs is that desirable or observed configurations — those consistent with the data — should be assigned low energy, whereas all other configurations should be associated with higher energy. The goal of learning in EBMs is to **shape the energy landscape in such a way that real data samples lie within low-energy regions, while unlikely or incompatible configurations occupy regions of higher energy**.



Rather than directly mapping inputs x to outputs y , as in traditional classification or regression models, the EBM framework seeks to determine whether a particular pair (x, y) is compatible. In other words, we are interested in determining whether a proposed output y is appropriate for a given input x . This can be framed as the problem of identifying outputs for which the energy function $E(x, y)$ is low. For example, we might want to know whether a particular high-resolution image y is a plausible enhancement of a low-resolution input x or whether a text sequence y is a suitable translation of another given text sequence x .



To formalize this, we define an **energy function** $E: X \times Y \rightarrow \mathbb{R}$, which maps input-output pairs to scalar energy values. This function should take lower values when the output y is well-aligned or compatible with the input x and higher values when the compatibility between the two is weaker.

Importantly, this energy function plays a role in inference rather than directly in learning. That is, after having learned this energy landscape during training, **inference consists of finding the value(s) of y that make $E(x, y)$ small for a given input x** , which can be expressed as:

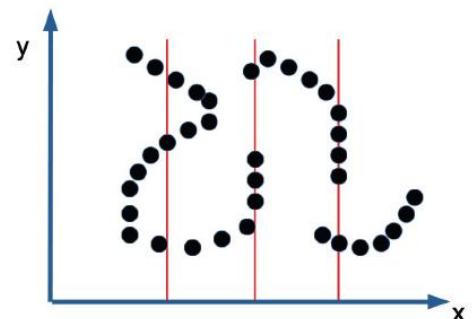
$$\hat{y} = \arg \min_y E(x, y)$$

Note: There may be multiple valid values of y (solutions) for a single x .

Conceptually, an EBM differs from traditional feed-forward models.

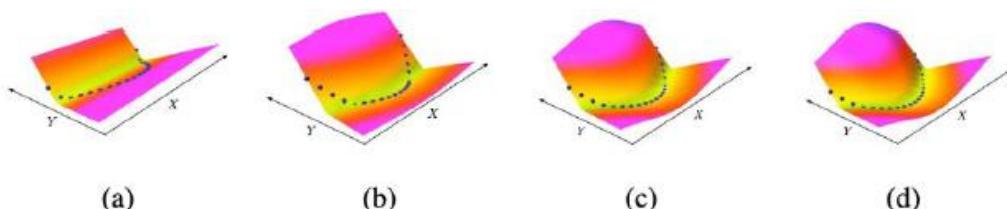
While a feed-forward model is an explicit function that deterministically computes an output y from an input x , an EBM defines an **implicit function** that captures the dependency between x and y .

As a result, EBMs can naturally accommodate multiple possible outputs for a single input, which classical models struggle to handle due to their one-to-one mapping structure. For instance, in Figure on side, where data points are represented as dots in the input-output space, we can see that for a given x we can have different corresponding y .

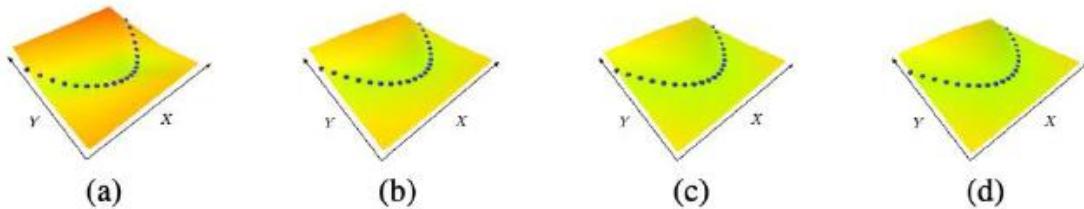


Energy Function Surface after Training

After a “good training”: After successful training, the energy surface over the space of input-output configurations should display low energy around regions corresponding good data combinations we trained the model to and high energy values for everywhere else.

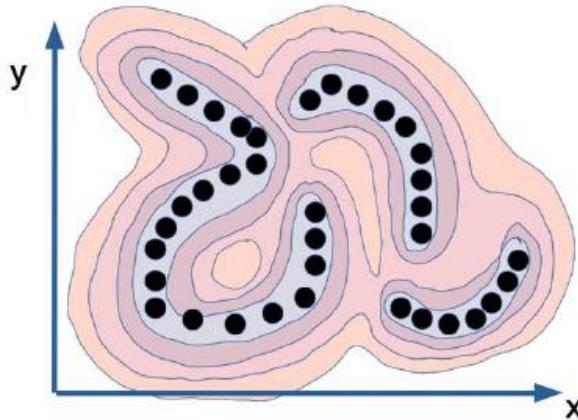


After a “bad training”: In contrast, poorly trained models may fail to learn meaningful input-output dependencies. For instance, if a model ignores the input and produces nearly identical outputs for every configuration, the resulting energy landscape **collapses to a flat surface**. This situation, known as **energy collapse**, leads to an energy function that provides no useful information — its surface remains constant and often near zero across the entire domain.

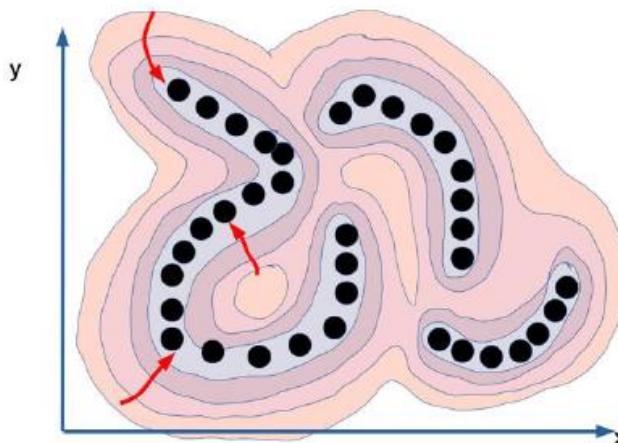


EBMs Inference

Inference in EBMs consists of navigating this energy surface to identify suitable outputs. For example, the Figure below shows the energy surface obtained after having trained the EBM on our datapoints in which the energy is low around the data points and higher elsewhere.



After having computed this surface effectively, we can then use optimization techniques such as gradient descent to find low-energy outputs during inference. When the output variable y is continuous and the energy function is smooth and differentiable, **gradient-based inference** becomes a natural choice. Starting from an arbitrary point in the input-output space, we **iteratively update y by descending along the gradient of the energy function**, keeping the input x fixed. This process allows us to explore the space of possible outputs and **converge toward configurations that minimize energy and thus are more likely under the model** (see Figure below).



Through this approach, EBMs provide a unified and flexible mechanism for inference that does not rely on enumerating or scoring all possible outputs explicitly. Instead, they exploit the geometry of the energy function, allowing us to reason about and discover compatible output configurations in a principled and efficient way.

Latent-Variable EBM

Latent-Variable EBMs extend the standard EBM framework by introducing an additional unobserved, or *latent*, variable z . The role of the latent variable is to capture hidden factors of variation that are not directly observable in the input x , but which are crucial for generating or understanding the output y .

Ideally, the latent variable should encode independent, disentangled factors of variation relevant to the task at hand. However, to ensure the model remains interpretable and generalizable, it is important to constrain the capacity of the latent space. If the latent variable is allowed to encode too much information, the model may overfit by embedding all necessary knowledge for prediction into z , effectively ignoring the structure of x and losing the benefits of modeling input-output dependencies (we will see better this in next paragraph).

The output y is now assumed to depend not only on the input x , but also on this latent variable z . The energy function thus becomes a function of all three components:

$$E(x, y, z) = C(y, \text{Dec}(\text{Pred}(x), z))$$

Note: Here, $C(\cdot)$ represents a “compatibility” cost ([later we will see which losses to use in general](#)) between the prediction and the ground truth.

Therefore, latent variables serve as auxiliary information — if we had access to them during inference, many tasks would become significantly easier.

For example, consider the problem of recognizing handwritten text. Human readers naturally perceive boundaries between characters or words, even in the absence of spaces. A machine, however, lacks this innate segmentation ability. In such cases, a latent variable can be used to represent the boundaries between words or characters in the text. Knowing where these boundaries lie would allow the model to better interpret ambiguous or continuous input, such as cursive handwriting or unsegmented text.

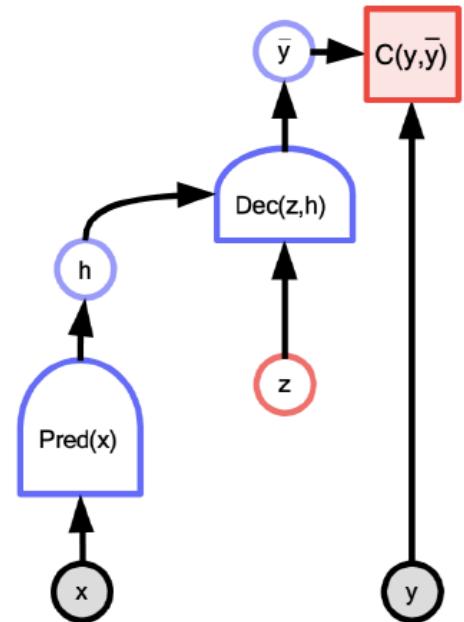
To illustrate this,

- imagine to have to read the following hand written word, it helps to know where the characters are.

- imagine to recognize speech, it helps to know where the words and phonemes are.

Youcantreadthisifyoudontunderstandenglish
Vousnepouvezpaslirececisivousneparlezpasfrançais

In both cases a latent variable z could represent the positions where characters/word boundaries occur, helping the model segment the sentence correctly and recognize the individual characters/words.



You can't read this if you don't understand English
 Vous ne pouvez pas lire ceci si vous ne parlez pas français

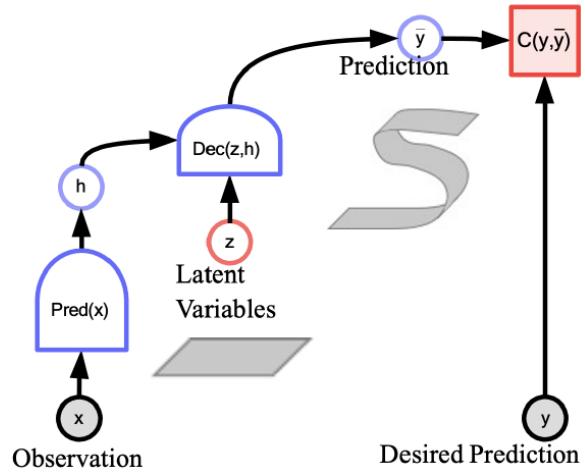
By incorporating a latent variable that models segmentation hypotheses, the system can explore various subdivisions of the input, searching for the one that minimizes overall recognition loss. Suboptimal or random subdivisions, in contrast, will produce high energy due to poor recognition performance.

Therefore, the use of latent variables allows us to encode such background knowledge implicitly within the model, providing a more powerful and flexible mechanism for interpreting ambiguous inputs. During inference, the model seeks the optimal configuration of both y and z that minimizes the energy function. That is, it solves:

$$\hat{y} = \arg \min_{y,z} E(x,y,z)$$

This joint optimization allows the model to reason over hidden structure, improving its ability to generate accurate and meaningful outputs.

Another big advantage of allowing latent variables is that by varying the latent variable z over a set, we can make the prediction output y vary over the manifold of possible predictions as well (see Figure on side).



Regularized Latent-Variable EBM

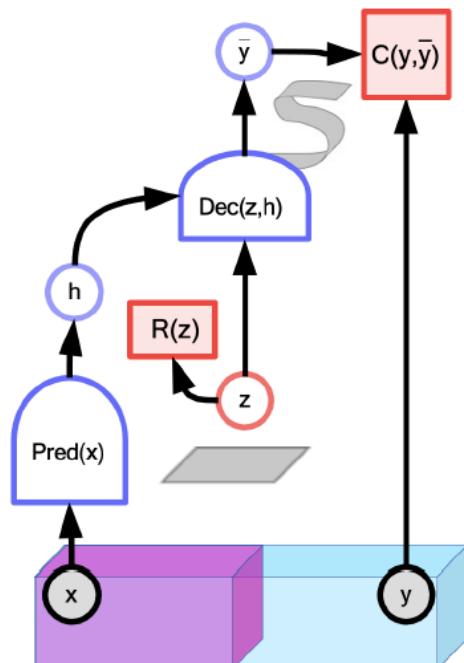
When incorporating latent variables into EBMs a critical challenge arises from the potential expressive power of the latent space itself. If the latent variable z is allowed too much flexibility, it may encode excessive information about the target output y , thereby undermining the intended dependency between the input x and output y . In such a case, for any input x , it becomes possible to perfectly reconstruct the true output y by selecting an appropriate value of z . This results in **overfitting**: the model relies too heavily on z to explain y , rather than learning meaningful mappings between x and y .

To address this, it is necessary to constrain the information-carrying capacity of the latent variable. A principled approach is to introduce a **regularization term** on z , penalizing overly informative or complex configurations. The goal of this regularization is to limit the extent to which z can freely encode arbitrary features, forcing the model to represent relevant structure in the energy function itself rather than hiding it in z .

The regularized energy function typically takes the form:

$$E(x,y,z) = C(y, Dec(Pred(x), z)) + \lambda R(z)$$

Here, $R(z)$ is a regularization term penalizing complex or overly informative latent codes and λ is a *hyperparameter* that modulates the trade-off between reconstruction accuracy and latent simplicity. By tuning λ one can balance the expressiveness of the latent space against the risk of overfitting. Without such regularization, the energy landscape could become flat and degenerate, allowing



arbitrary reconstructions that bear no meaningful relationship to x , effectively bypassing the model's learning objective.

There are several strategies for regularizing the latent variable z to control its information capacity:

- **Effective dimensionality reduction:** Constrain the latent space by limiting the number of active dimensions in z , thereby reducing its capacity to encode information.
- **Quantization/discretization:** Force the values of z to become discrete or quantized. This enforces a form of structural simplicity in the latent representation.
- **Sparsity through norm constraints:** Apply sparsity-inducing regularizers, such as the ℓ_0 norm (which penalizes the number of non-zero components) or the ℓ_1 norm. These encourage the model to use fewer latent dimensions to explain the data.
- **Maximize lateral inhibition or component competition:** Introduce inhibitory interactions among latent components to encourage competition, thereby selecting only the most relevant subset of features to contribute to the final output.
- **Stochastic Perturbation with Norm Control:** Add noise to the latent variable z while simultaneously constraining its ℓ_2 norm, preventing them from encoding arbitrary fine-grained details.

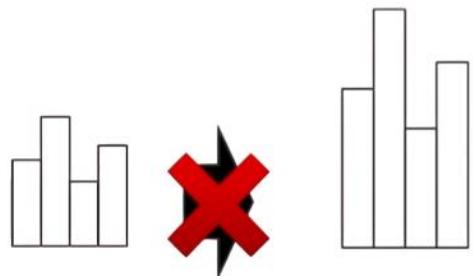
How Do we Model EBMs?

Now that we have a conceptual idea of what EBMs aim to do, let's dive into how we actually model them. This is the most important aspect to understand from this chapter, because it will help reveal the true advantage of EBMs later on.

In order to model an EBM, it needs to respect a probability distribution formulation. For this, the model must satisfy two key axioms of probability:

- **Non-negativity:** The probability distribution needs to assign any possible value of (x, y) a non-negative value, i.e. $p(y|x) \geq 0$.
- **Normalization:** The total probability across all possible values of y (given x) must sum/integrate to 1:
 - For continuous variables: $\int_y p(y|x) dy = 1$
 - For discrete variables: $\sum_y p(y|x) = 1$

Note that the requirement that probabilities sum (or integrate) to 1 is a **normalization step** that ensures predictions improve **relatively**, not just absolutely. Think of it in this way: suppose you have a scoring function that gives as prediction a score of 20 or 30. It might seem natural to interpret higher scores as better. But what if the model simply learns to inflate scores to 200 or more? The scores would be higher, but this wouldn't mean the model is actually better — it's just inflated the scores uniformly. We don't want the model to learn to trivially increase all outputs. Instead, we want the model to assign **higher likelihoods to better predictions relative to others**. In other words, the model should improve **relatively** and for doing that the total "probability mass" must remain fixed at 1 to force the model to make non-trivial improvements.



Non-negativity is easy to ensure. For instance, we can apply transformations like squaring or exponentiation to a neural network output:

- $g_\theta(x, y) = f_\theta^2(x, y)$

- $g_\theta(x, y) = \exp(f_\theta(x, y))$

where f_θ is a neural network. However, these functions are not normalized, i.e. they do not integrate (or sum) to 1. To fix this, we can normalize them by dividing by the total volume under the function — i.e., basically by taking the integral/summation of the function over all possible outputs y :

$$p_\theta(y|x) = \frac{1}{\text{volume}(g_\theta(x; \cdot))} g_\theta(x, y) = \begin{cases} \frac{1}{\int_y g_\theta(x, y) dy} g_\theta(x, y), & \text{if } y \text{ is continuous} \\ \frac{1}{\sum_y g_\theta(x, y)} g_\theta(x, y), & \text{if } y \text{ is discrete} \end{cases}$$

Examples:

- $g_{\theta=(\mu, \sigma)}(y) = e^{-\frac{(y-\mu)^2}{2\sigma^2}} \rightarrow \text{volume}(g_\theta) = \sqrt{2\pi\sigma^2} \rightarrow \text{Gaussian}$
- $g_{\theta=\lambda}(y) = e^{-\lambda y} \rightarrow \text{volume}(g_\theta) = \frac{1}{\lambda} \rightarrow \text{Exponential}$

In general, we want to find convenient forms for g_θ to be able to compute that sum/integral **analytically** — in closed form. This means we don't need to approximate the integral numerically, which would be much harder and less efficient. The above examples are ideal since we know their closed form solutions (that's also why they are often used), however dealing just with well-known distributions can be restrictive since they are not able to express complex domain knowledge (i.e. what if we deal with complex multimodal true data distributions?) or flexible interactions between variables.

In EBMs, we are interested in finding the probability of outputs y given inputs x data **implicitly** using an energy function $E(x, y)$, where lower energy values correspond to higher probabilities. This function is typically parameterized by a neural network, $E_\theta(x, y)$, which maps input-output pair to a scalar energy value.

To ensure that it respects the two probability axioms seen above, what we can do is define an energy function and divide by its volume:

$$p_\theta(y|x) = \frac{e^{-E_\theta(x, y)}}{Z_\theta(x)}$$

Here $Z_\theta(x)$ is a **normalizing constant** known as the **partition function** and defined as:

$$Z_\theta(x) = \begin{cases} \int_y e^{-E_\theta(x, y)} dy, & \text{if } y \text{ is continuous} \\ \sum_y e^{-E_\theta(x, y)}, & \text{if } y \text{ is discrete} \end{cases}$$

In this formulation $E_\theta(x, y)$ defines an **unnormalized probability** and $-E_\theta(x, y)$ is the actual **energy function** where negative sign is chosen so that we assign real data to low energy values and negative samples to high energy values. This construction guarantees that we obtain a valid probability distribution, making the EBM mathematically well-posed.

Note: Why do we pick **exponential**? It's because it couples well with maximum log-likelihood, since the log cancels the exponential. Moreover, exponential functions are widespread in statistical physics and naturally appear in many probabilistic models.

Why EBMs?

Unlike traditional likelihood-based models, which require us to define explicit normalized probability distributions, EBMs allow us to work with **unnormalized models**. We focus on learning the energy function $E_\theta(x, y)$ directly,

which gives us significant **flexibility** — we can design arbitrary neural networks that map input-output pairs into scalar energy values, tailored to the specific structure of the problem we're solving. Another advantage is that EBMs naturally support **latent variables** (as previously discussed), allowing richer modeling capabilities.

However, the main challenge is that training such models often requires computing the *partition function* Z_θ , which can be **intractable** for complex or high-dimensional distributions. Likelihood-based models address this challenge by choosing well-known probability distributions — such as Gaussian, uniform or exponential — for which the normalization constants (i.e., volumes) can be computed analytically. However, as we've discussed, **EBMs** aim to move beyond this limitation, allowing us to define more flexible and expressive models without being constrained to predefined distribution families.

So now we're left with two major challenges in training EBMs:

1. How to **handle the partition function** since without a way to deal with Z_θ , we can't ensure that the model defines a valid probability distribution?
2. And then, as a direct consequence of the first, how to **train** the model effectively?

We will now see that we can find a way to solve both these problems with some mathematical tricks.

Training EBMs

Now, to train EBMs a first attempt could be to **maximize the log-likelihood** of observed data $\{x_i, y_i\}_{i=1}^N$. This gives the following objective:

$$\max_{\theta} \mathcal{L}(\theta) \quad \text{where } \mathcal{L}(\theta) = \sum_{i=1}^N \log p_\theta(y_i|x_i)$$

Using the EBM formulation we have:

$$\mathcal{L}(\theta) = \sum_{i=1}^N \log \frac{e^{-E_\theta(x_i, y_i)}}{Z_\theta(x_i)} = \sum_{i=1}^N (\log e^{-E_\theta(x_i, y_i)} - \log Z_\theta(x_i))$$

This formulation requires $p_\theta(y|x)$ to be a **normalized probability distribution**, which means computing the partition function Z_θ . However, this is usually **intractable** for arbitrary energy functions $E_\theta(x, y)$, since it requires integrating over all possible outputs y :

$$Z_\theta(x) = \int_y e^{-E_\theta(x, y)} dy \quad (\text{continuous case})$$

We don't know Z_θ , but there's still a way around that. By taking the gradients, we get:

$$\nabla_\theta \mathcal{L}(\theta) = - \sum_{i=1}^N \nabla_\theta E_\theta(x_i, y_i) - \sum_{i=1}^N \nabla_\theta \log Z_\theta(x_i)$$

Let's concentrate in computing $\nabla_\theta \log Z(\theta)$:

$$\begin{aligned} \nabla_\theta \log Z(\theta) &= \frac{1}{Z(\theta)} \nabla_\theta Z(\theta) \\ &= \frac{1}{Z(\theta)} \nabla_\theta \int_y e^{-E_\theta(x, y)} dy = \frac{1}{Z(\theta)} \int_y \nabla_\theta e^{-E_\theta(x, y)} dy \end{aligned}$$

Applying the chain rule:

$$\nabla_{\theta} e^{-E_{\theta}(x,y)} = -e^{-E_{\theta}(x,y)} \nabla_{\theta} E_{\theta}(x,y)$$

so:

$$= -\frac{1}{Z(\theta)} \int_y -e^{-E_{\theta}(x,y)} \nabla_{\theta} E_{\theta}(x,y) dy = -\int_y \frac{e^{-E_{\theta}(x,y)}}{Z(\theta)} \nabla_{\theta} E_{\theta}(x,y) dy$$

Now, recognizing that this is the expectation of $\nabla_{\theta} E_{\theta}(x,y)$ under the model distribution $p_{\theta}(y|x)$ we get:

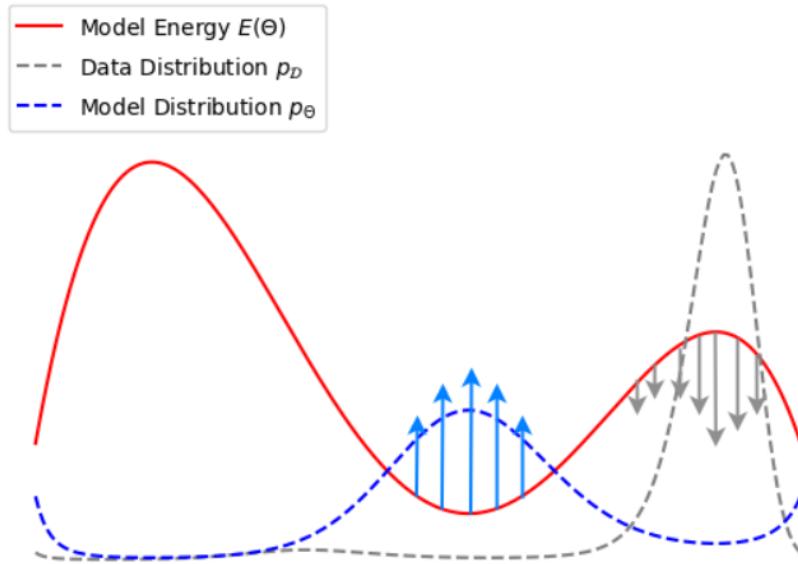
$$\nabla_{\theta} \log Z(\theta) = -\mathbb{E}_{y \sim p_{\theta}(y|x)} [\nabla_{\theta} E_{\theta}(x,y)]$$

So, plugging this back into the full expression for $\nabla_{\theta} \mathcal{L}(\theta)$ we get:

$$\nabla_{\theta} \mathcal{L}(\theta) = -\sum_{i=1}^N \nabla_{\theta} E_{\theta}(x_i, y_i) + \sum_{i=1}^N \mathbb{E}_{y \sim p_{\theta}(y|x_i)} \nabla_{\theta} E_{\theta}(x_i, y)$$

This gradient no longer depends on the intractable partition function Z_{θ} . It consists of two complementary terms:

1. The first term $-\sum_{i=1}^N \nabla_{\theta} E_{\theta}(x_i, y_i)$ **pushes down the energy $E_{\theta}(x_i, y_i)$ of true data pairs (x_i, y_i)** , thereby increasing the model's probability density on observed examples.
2. The second term $N \mathbb{E}_{y \sim p_{\theta}(y|x)} \nabla_{\theta} E_{\theta}(x, y)$ **pushes up the energy $E_{\theta}(x_i, y)$ of other samples y drawn from the model distribution** and therefore decreases the probability density assigned to outputs generated by the model itself (non-data samples).



Maximizing the log-likelihood with this gradient therefore **sculpts the energy landscape** lowering energy where the true data density exceeds the model density and raising it where the model density exceeds the data density.

Okay we ripped off from the partition function, however we still have a little problem to solve: the second term in the gradient involves an expectation over the model distribution, which is typically intractable to compute exactly. To address this, we can resort to **sampling-based methods** to approximate the expectation.

One widely adopted approach in this context is **Stochastic Gradient Langevin Dynamics (SGLD)**, also simply known as **Langevin sampling**, a form of *Markov Chain Monte Carlo (MCMC)*. It provides a procedure to sample from a given distribution, in our case $p_{\theta}(y|x)$, using only its **score function**, defined as the gradient of the log-probability with respect to the variable being sampled (in this case, the output y):

$$s_\theta(x, y) \approx \nabla_y \log p_\theta(y|x)$$

Note: We will revisit in more detail the *score function* and *Langevin sampling* in the context of *score matching diffusion models* in the Diffusion chapter.

Note: It is worth emphasizing that this gradient is taken with respect to the data variable y and is therefore not the usual gradient with respect to the learnable parameters θ .

Specifically, what we want to do is initialize the chain by drawing an initial value y_0 sampled from a given prior distribution and then iterate the following K Markov chain steps:

$$y_{k+1} = y_k + \eta \nabla_y \log p_\theta(y_k|x) + \sqrt{2\eta} \epsilon_k, \quad k = 0, \dots, K$$

where $\epsilon_k \sim \mathcal{N}(0, I)$ and the parameter η controls the step size. Each iteration of the Langevin equation takes a step in the direction of the gradient of the log likelihood and then adds Gaussian noise. This process stochastically descends the energy landscape with added noise to prevent getting stuck in local minima. It can be shown that, in the limits of $\eta \rightarrow 0$ and $K \rightarrow \infty$, the final y_K obtained with this procedure converges to a sample from the distribution $p_\theta(y|x)$. In other words, the error is negligible when η is sufficiently small and K is sufficiently large.

However, Langevin dynamics requires running a long Markov chain to generate the sample, which can be computationally expensive and so we need to consider practical approximations.

Contrastive Divergence

To reduce the computational cost of a full MCMC, **Contrastive Divergence (CD)** [116] is generally adopted. Instead of running MCMC to equilibrium, CD approximates this expectation using a **short MCMC chain**, i.e. using a few steps of Langevin sampling (even as few as one step – CD-1), which is computationally much less costly.

In **CD-K**, the idea is:

1. Start from a training pair (x_i, y_i) .
2. Initialize the chain at $y_0 = y_i$
3. Run K steps of Langevin dynamics to get a "negative sample" \tilde{y}_i .
4. Approximate the gradient as:

$$\nabla_\theta \mathcal{L}_{CD}(\theta) \approx -\nabla_\theta E_\theta(x_i, y_i) + \nabla_\theta E_\theta(x_i, \tilde{y}_i)$$

This provides a **contrast** between observed pairs and model-generated pairs — hence the name. More compactly:

$$\nabla_\theta \mathcal{L}_{CD}(\theta) \approx -\nabla_\theta E_\theta(x, y^+) + \nabla_\theta E_\theta(x, y^-)$$

where (x, y^+) is a real pair and (x, y^-) is a pair with a contrastive negative sample obtained after a few steps of Langevin dynamics.

Note: In the discussion above, we dealt with continuous distributions, which is why we used Langevin sampling. In the case of **discrete distributions**, a similar procedure for CD can be applied using **Gibbs sampling** instead. If you are interested in more details about Gibbs sampling, you can refer to the “Sampling” chapter in the recommended textbook.

Contrastive Divergence is not consistent in a statistical sense — it does **not follow the true gradient of the likelihood** (since as we said above for ensuring it, we should have a sufficient large K , which here is not the case) — but it has been found to work well **empirically** in many practical scenarios.

More interestingly, since the negative samples used in CD are typically generated via only a few MCMC steps starting from real data, they tend to stay **close to the data manifold**. This means the gradient shapes the energy function primarily around the regions of data support. This can prove to be **sufficiently effective** for tasks such as **discrimination** (which was the original use case when Hinton introduced it), but is expected to be less effective in learning a generative model.

An extension of CD is *Persistent Contrastive Divergence* (PCD) [117], which to improve sampling efficiency stores previously generated negative samples in *replay buffers*, avoiding the need to restart Langevin chains from scratch at each iteration.

Note: Of course, the mathematical framework discussed so far can be naturally extended to **latent-variable EBMs** $E_\theta(x, y, z)$.

Other Contrastive Approaches

An important observation is that the **partition function** $Z_\theta(x)$ cancels out when computing **likelihood ratio** between two outputs for the same input. For example, for a positive pair (x^+, y^+) and a negative pair (x^+, y^-) :

$$\frac{p_\theta(y^+|x)}{p_\theta(y^-|x)} = \frac{e^{-E_\theta(x,y^+)}/Z_\theta(x)}{e^{-E_\theta(x,y^-)}/Z_\theta(x)} = e^{-(E_\theta(x,y^+)-E_\theta(x,y^-))}$$

The partition function disappears entirely.

This property opens the door for training EBMs using a variety of **contrastive learning objectives**, since only relative comparisons of energies matter. For example, with a set of positive pairs and contrastive negatives (random, corrupted, or adversarial), one can use a **hinge loss**:

$$\sum_i \max(0, E_\theta(x_i, y_i^+) - E_\theta(x_i, y_i^-) + m)$$

where m is a **margin** enforcing a minimum energy gap between positive and negative pairs.

Suitable contrastive loss functions must enforce a **non-zero margin** between positive and negative pairs to avoid energy collapse. The contrastive loss functions can be calculated pairwise for specific data sets as the hinge loss we seen above, but lately, an increasing interest is given to group-based objectives such as **InfoNCE**, where the loss depends on the relative energies of an entire set of samples rather than on a single pair.

Aside from contrastive methods, other common approaches for training EBMs involve:

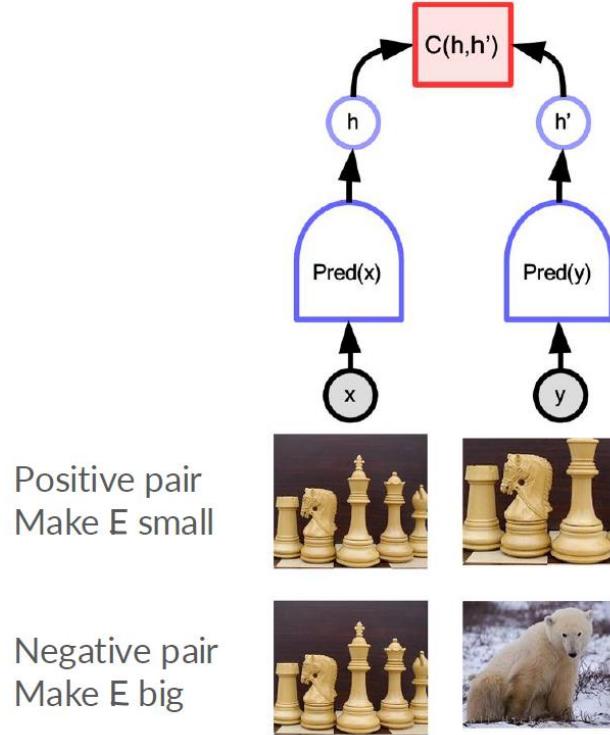
- **Score Matching** and **Noise Contrastive Estimation (NCE)**, which bypass direct computation of the partition function
- **Regularized or architectural methods**, where energy collapse is prevented not through explicit negatives, but via **constraints on the model architecture** or **additional regularization terms**.

We will revisit this latter family of methods in the upcoming sections on **non-contrastive embedding methods**, where architectural design choices play a key role in ensuring meaningful representations without the use of negatives.

Contrastive Joint Embedding

Contrastive embedding methods are designed to learn representations by comparing *pairs* of data. Suppose we start with a pair (x, y) , where x is an image and y is a transformed (augmented) version of x that preserves its semantic content (such as a crop, rotation or color jitter). This is called a **positive pair** because both samples

convey the same underlying information. To learn useful representations, we pass both x and y through a shared neural network to obtain two feature vectors, h and h' . Since the content is the same, we want these vectors to be *similar*. We enforce this by applying a **similarity-based loss** — typically using cosine similarity or dot product — designed to bring h and h' closer in the embedding space. This effectively lowers the energy of samples that lie on the true data manifold. However, we also need to *increase* the energy for off-manifold or unrelated samples. To do this, we introduce **negative pairs** — for example, two images from different classes — and push their embeddings *apart*. This contrast between positive and negative pairs allows the network to structure the embedding space meaningfully, pulling similar data together and pushing dissimilar data apart.



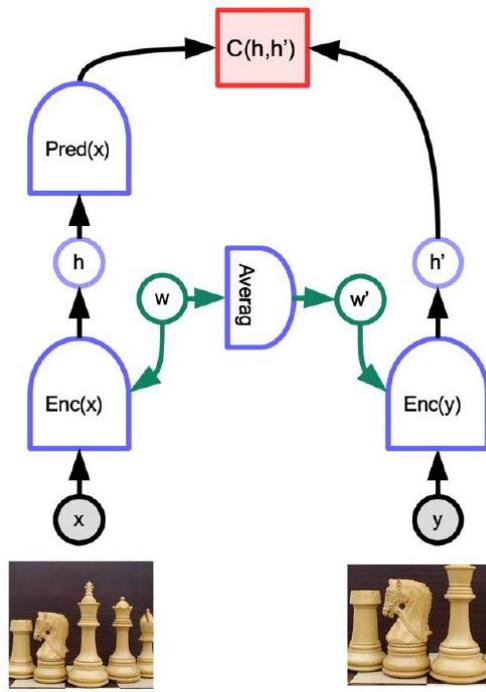
Architectures such as Siamese networks and techniques from *metric learning* are commonly used [118] [119]. However, it also introduces new challenges, notably the need for **hard negative mining** — finding informative negative samples that truly challenge the model.

Successful examples of *contrastive joint embedding* methods in vision include DeepFace [120], PIRL [121], **MoCo** [122] and **SimCLR** [123]. These techniques also extend beyond vision — for instance, using *temporal proximity* as a signal in video and audio [124].

Non-Contrastive Embedding

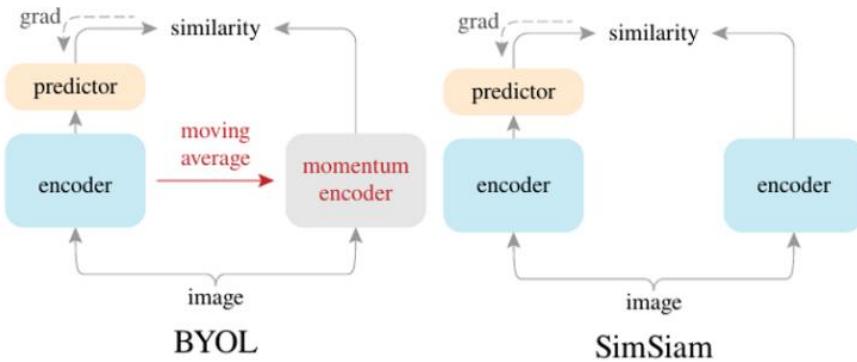
Given the complexity of contrastive learning, especially the challenges of selecting, storing and mining hard negative samples, a natural question arises: Can we learn useful embeddings **without using negative examples**? Surprisingly, the answer is **yes**. **Non-contrastive methods** aim to achieve similar goals (learning semantically meaningful embeddings) without relying on explicit negative sampling. These methods typically rely on **positive pairs only** — e.g., different views or augmentations of the same input. Instead of pulling these pairs together while pushing others apart, they design the training objective in a way that avoids energy collapse without requiring explicit repulsion.

A typical non-contrastive approach uses **Siamese-like architectures with small variations** — such as using different weights on the two network branches — and stabilizes learning using architectural or statistical tricks such as applying **stop-gradient operations** or **momentum encoders** to prevent representational collapse.



Prominent examples include [125]:

- **BYOL (Bootstrap Your Own Latent)** of Google DeepMind, in which one branch is a *momentum encoder* (an exponential moving average of the main encoder).
- **SimSiam** where *stop-gradient* is applied to one side during training.



Introducing asymmetry between the two network paths they become able to break the trivial symmetry that would allow all outputs to collapse to a constant vector. Moreover, since *non contrastive* methods do not require negative samples, non-contrastive methods **eliminate the need for hard negative mining**, simplifying training and often improving stability.

Lastly, it is worth noting that many of the models we have encountered so far — such as denoising autoencoders, variational autoencoders and others — can be reinterpreted through the lens of EBMs. For further discussion of these connections, see the lectures and papers cited at the beginning of this chapter.

Diffusion Models

In this final chapter, we explore a powerful class of generative models known as **diffusion models**, which have rapidly become the state of the art across numerous domains — generating not only images but also videos, music and, more recently, text. While the framework is broadly applicable, we'll focus here on image generation as our main illustrative example.

In the course the core material used to explain diffusion models consists of two online resources:

- [The Illustrated Stable Diffusion – Jalamar](#)
- [How Stable Diffusion Works – stable-diffusion-art.com](#)

These, in my view, are among the **worst attempts** to explain how diffusion models actually work. So, in this chapter, I'll try to do much better by exposing the real core concepts behind them.

Intuition Behind Diffusion Models

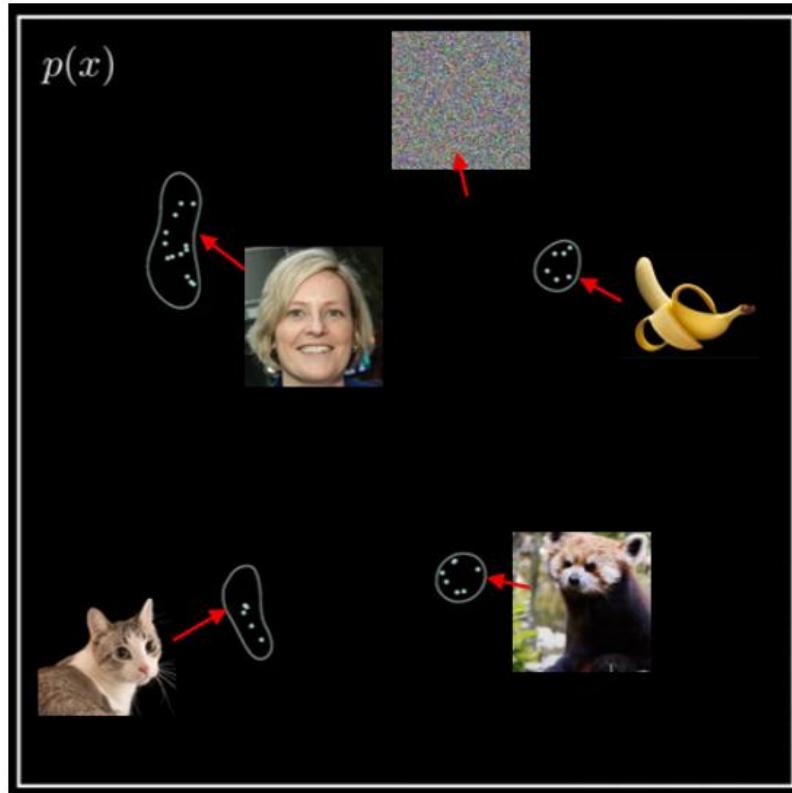
When generating data, we aim in some way to learn the *true data distribution*, denoted as $p(x)$. This distribution is incredibly complex — so much so that we can't really find a single expression to describe it completely and there are probably no such expressions anyway. Still, even without an explicit formulation, we want to generate new images, which is equivalent to sampling new points from this underlying distribution.

Let's consider the specific case of images. We can think of this distribution as encompassing all natural images. In other words, we're dealing with *image space* — the space of all possible images of a given size. Suppose we fix this size at 1000×1000 pixels. This results in a 1-million-dimensional space, where each axis represents the value of one pixel in the image. In the Figure below, we provide a simplified 2D representation. Of course, this 2D grid can only truly represent two pixels, but for the sake of intuition, let's pretend it represents the full 1-million-dimensional image space. Pixel values range from 0 to 255, representing intensity. Thus, the space of all possible images is confined within a 1-million-dimensional hypercube, where each axis (pixel) spans values between 0 and 255. Each point in this high-dimensional space corresponds to a potential image. For instance, a cat image might be located at one point, while a random noise image — e.g. constructed by sampling each pixel value from a Gaussian distribution — might reside elsewhere. A good image generator needs to learn what differentiates a good-looking image from a nonsensical one. It must learn to generate images that are closer — in some sense — to regions in image space where "good" images reside.

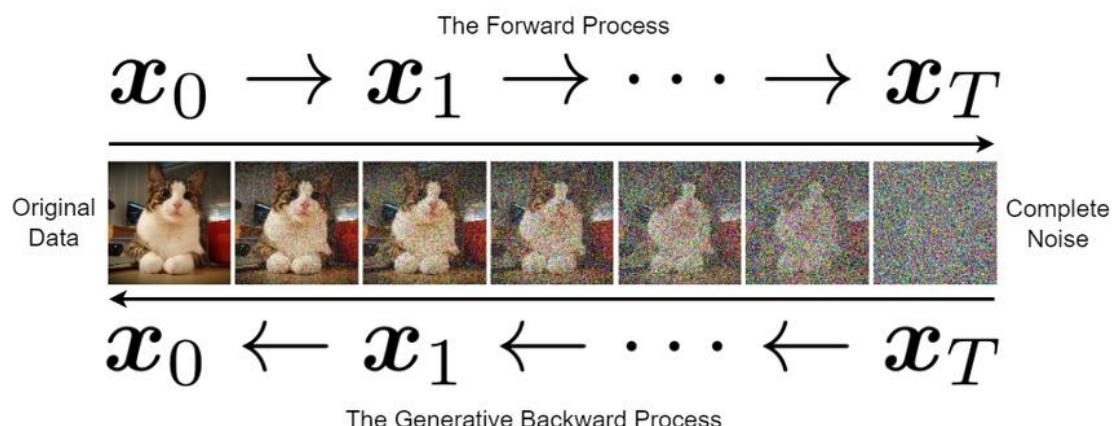
The first step is to collect a large dataset of high-quality images — the kind we'd want our model to generate. Then, we can examine where these images lie within image space by plotting their pixel values on this high-dimensional grid — one pixel per axis. When we do this, we will observe two important aspects:

1. **Most of the image space is empty** — This is because the **vast majority of possible images don't correspond to anything coherent or realistic**. Good images occupy only a tiny subset of this vast space because they must follow strict patterns. For example, nearby pixels often have similar values. Most randomly generated images violate such constraints.
2. **Images of the same type cluster together** — For example, banana images form a **cluster** separate from cat images. The banana cluster lies far from the cat cluster, although banana images are close to each other within their own cluster.

Of course, this is a simplification, but it's a helpful way to conceptualize high-dimensional image space. So, if we randomly sample a point in image space, it's almost certainly going to fall into the "empty" areas — far from any meaningful image cluster and result in **random noise**.



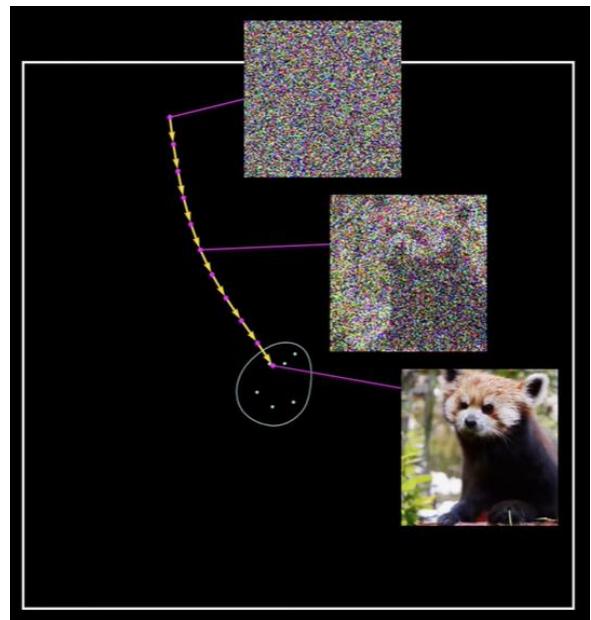
Diffusion models are a class of generative models that address this problem with a very interesting approach: they start with a real image and gradually corrupt it, **adding Gaussian noise** over many steps until it becomes indistinguishable from pure noise. Then, a deep neural network is trained to reverse this corruption step by step — removing the noise in small increments. Once trained, if successful, the network can then begin at pure noise and **denoise** its way back to a plausible image.



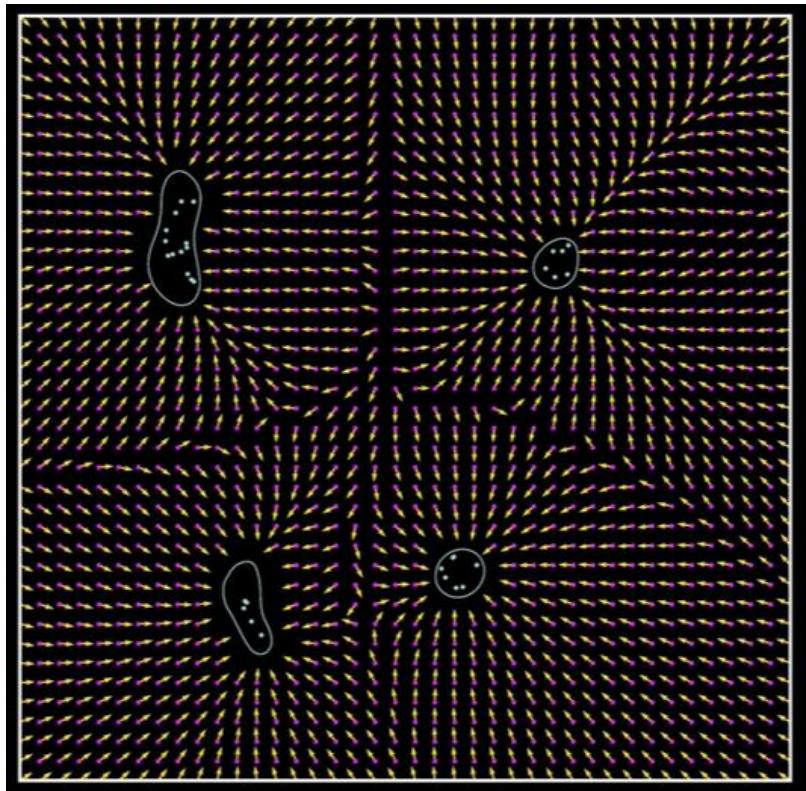
This reverse process can be interpreted geometrically also under another perspective: learning a **vector field** over image space. At every point, the vector field gives a direction — a small update that nudges the image to look slightly more realistic. We can imagine it saying things like: "make pixel 1 a bit brighter, pixel 2 a bit darker" and so on — for all one million pixels. These directions are not random, they are carefully learned so that, starting from any noisy image, following the vector field eventually leads you to a nearby region of meaningful images.

For example, imagine a random point in image space that happens to lie closer to the region where red panda images cluster. As you apply the model's vector guidance — step by step — it gradually steers the image toward that cluster. After enough steps, you land inside the red panda region. This is how the red panda in the Figure below was generated: not because the model explicitly intended to generate a red panda, but because the initial noise sample was closer to that region of image space.

Note: This happens because, in a standard diffusion model, we simply learn to reverse the noise process without any form of conditioning — meaning the model generates useful images purely from noise, but without control over *what* it generates. Later, we'll see how this process can be **conditioned** to guide the output (e.g., by using class labels or even text prompts).



Through training, the **diffusion model effectively learns a dense vector field over image space** — one that maps every possible image to the direction most likely to lead toward a region of high data density. In other words, it learns how to get from arbitrary noise to a good image. By repeating this process across many training samples, the model learns the entire vector field. It becomes capable of navigating from almost any starting point in image space to some region of meaningful images.



This idea — of following directional toward dense regions of image space — is more conceptually related to another perspective on diffusion models called **score matching generative modeling**. We will cover this perspective next as it can be a little more difficult to grasp at first glance.

Denoising Diffusion Probabilistic Model

We start by following step by step the research paper that made diffusion models so popular, called **Denoising Diffusion Probabilistic Models**, or **DDPM**, introduced by Jonathan Ho et al. in 2020 [126]. That's also why diffusion models are sometimes referred to as *denoising diffusion probabilistic models*. Despite its reputation, this paper did not introduce diffusion models, nor did it coin the term. The earliest known reference to diffusion models as we know them comes from a 2015 paper titled “*Deep Unsupervised Learning Using Non-Equilibrium Thermodynamics*” by Jascha Sohl-Dickstein et al [127].

In 2015, GANs were considered state-of-the-art in image generation research and diffusion models received little attention. In contrast, the 2020 DDPM paper quickly gained traction in the field of image/video generation. The main contribution of this paper was to take diffusion models from a theoretical concept previously limited to simple toy datasets into a state-of-the-art technique for image generation. On the technical side, it mainly introduced a much simpler training objective compared to the 2015 version. This made the method more accessible and highlighted interesting connections to earlier works on denoising models.

Diffusion models are **easy to train**, they scale well on parallel hardware, and they avoid the challenges and instabilities of GANs’ adversarial training while producing results that have quality comparable to or better than them. However, as we will see, **generating new samples can be computationally more expensive** than GANs due to the need for multiple forward passes through the decoder network (often hundreds or even **thousands**). Now, let’s examine how the DDPM paper formalizes this process. My explanation is partly inspired by this [video](#), although I have adapted the mathematical notation to align more closely with the paper and added additional insights for clarity.

Forward Diffusion Process

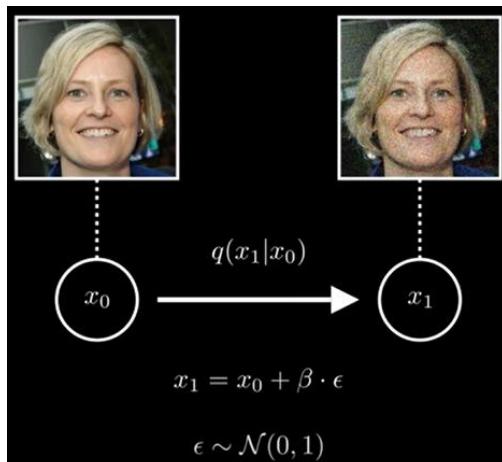
Let’s begin with an initial image denoted as x_0 , representing the clean, noise-free data. The goal of the forward diffusion (or *noising*) process is to gradually corrupt this image by adding Gaussian noise over a sequence of time steps, ultimately transforming it into pure noise, i.e. a standard normal distribution $\mathcal{N}(0, I)$. This process is central to diffusion models and is designed to allow the model to later learn how to reverse it.

To generate the next image x_1 , we use a conditional distribution $q(x_1|x_0)$, where noise is added to x_0 as follows:

$$x_1 = x_0 + \beta \cdot \epsilon$$

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

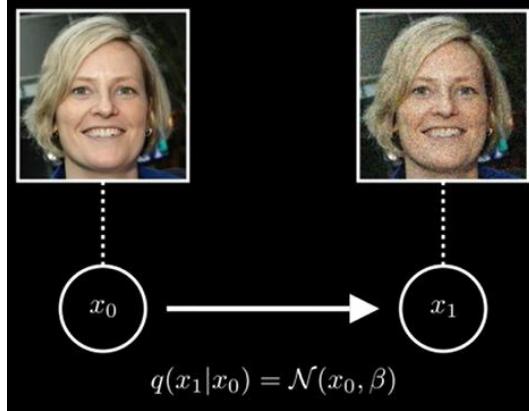
where β is a scalar that controls the **amount of noise** to add.



Therefore, the distribution describing x_1 is just a Gaussian centered on x_0 with noise of variance β :

$$q(x_1|x_0) = \mathcal{N}(x_0, \beta I)$$

So, rather than seeing this as adding noise to x_0 , we can just imagine it as taking a sample from a Gaussian distribution of variance β centered on x_0 .

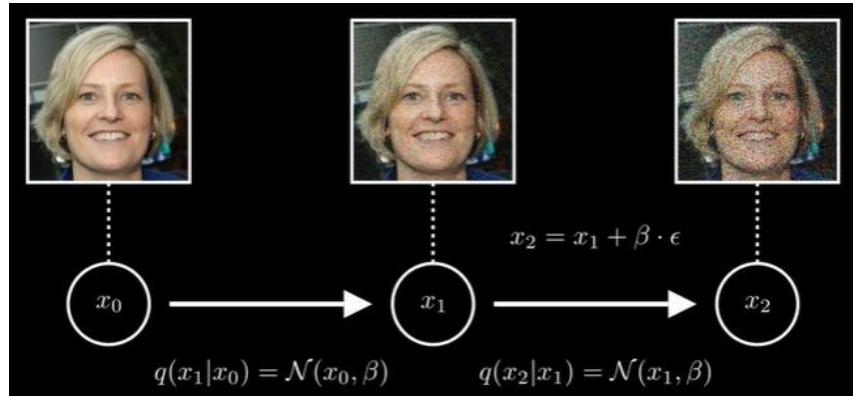


This same process is repeated for subsequent steps. For example, x_2 is generated from x_1 by adding the same amount of noise:

$$x_2 = x_1 + \beta \cdot \epsilon$$

which is the same as taking a sample from a Gaussian distribution centered on x_1 with variance β :

$$q(x_2|x_1) = \mathcal{N}(x_1, \beta I)$$



We continue this noising process step by step until we reach a final time step T (a hyperparameter). To represent the full forward process, we consider the joint probability distribution over all intermediate variables from x_1 to x_T , conditioned on the initial image x_0 :

$$q(x_1, x_2, \dots, x_T | x_0)$$

Using Bayes' theorem, we can decompose this joint probability into the product of the individual conditional distributions:

$$q(x_1, \dots, x_T | x_0) = q(x_1|x_0)q(x_2|x_1, x_0) \dots q(x_T|x_{T-1}, \dots, x_0)$$

However, because the forward diffusion process is actually a **Markov chain** where each step only depends on the previous one this expression further simplifies to:

$$q(x_1, \dots, x_T | x_0) = q(x_1|x_0)q(x_2|x_1) \dots q(x_T|x_{T-1})$$

To simplify notation, we denote the entire sequence x_1 through x_T as $x_{1:T}$. Using this, the full forward diffusion process becomes:

$$q(x_{1:T}|x_0) = q(x_1|x_0)q(x_2|x_1) \dots q(x_T|x_{T-1})$$

And now let's rewrite this product in an even more compact form using the standard product notation:

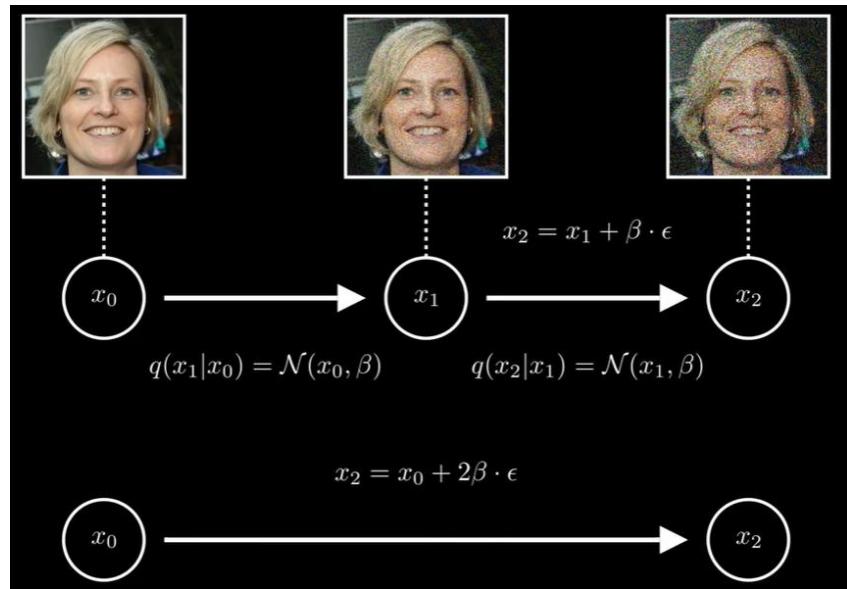
$$q(x_{1:T}|x_0) = \prod_{t=1}^T q(x_t|x_{t-1})$$

Now, coming back to our previous discussion, since we just sum the noise terms, we can go straight from x_0 to x_2 . We created x_2 by adding twice as much noise

$$x_2 = x_0 + 2\beta \cdot \epsilon$$

which is the same as sampling from a Gaussian centered on x_0 with variance $2\beta I$:

$$q(x_2|x_0) = \mathcal{N}(x_0, 2\beta I)$$

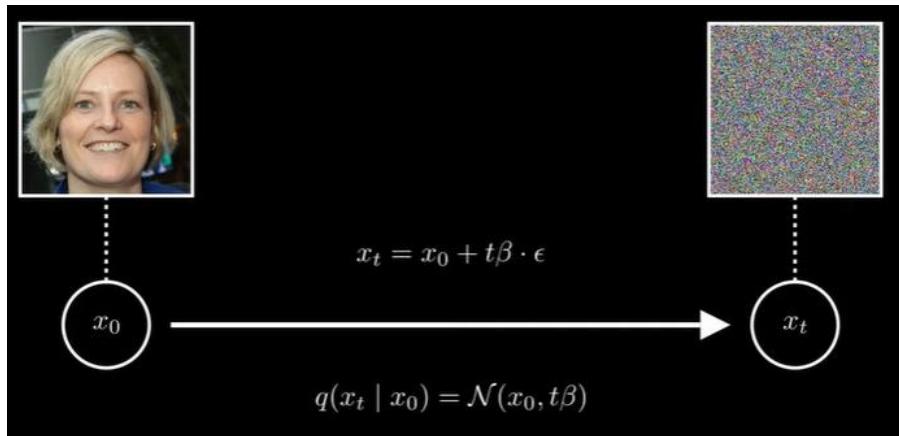


If we think a little, we can write a simple formula to describe what happens at any given step. We simply need to add t times β amount of Gaussian noise to x_0

$$x_t = x_0 + t\beta \cdot \epsilon$$

so, at time step t , $q(x_t|x_0)$ is a Gaussian distribution centered at x_0 with variance $t\beta I$.

$$q(x_t|x_0) = \mathcal{N}(x_0, t\beta I)$$



Now we have an issue with this process. What we wanted was a process that gradually transforms our data into a standard distribution. Clearly that's not what's happening here. The mean always stays fixed at x_0 and is never modified by the process, while the variance keeps increasing indefinitely according to $t\beta$.

So, in this setup there is no chance that our diffusion process converges to the standard distribution $\mathcal{N}(0, I)$. In this diffusion process the variance just keeps growing without bound, in the literature we often say that the “variance explodes” and we call this process the **variance exploding diffusion**.

So, we need a way to rewrite the diffusion process so that it gradually transforms the data into a standard normal distribution as time progresses:

$$q(x_t | x_{t-1}) \xrightarrow{t \rightarrow \infty} \mathcal{N}(0, I)$$

To achieve this, **we must ensure that the mean of the distribution converges to 0 and the variance converges to 1**. To achieve it, the authors of the DDPM paper proposed the following update rule for the forward process:

$$x_t = \sqrt{1 - \beta_t} x_{t-1} + \sqrt{\beta_t} \epsilon_t$$

Here, $\epsilon_t \sim \mathcal{N}(0, I)$ and we write ϵ_t (with subscript) to emphasize that a new noise sample is drawn at each time step which is not the same as done in the naïve approach discussed before (i.e. now we have that $\epsilon_1 \neq \epsilon_2 \neq \dots \neq \epsilon_T$).

This formulation corresponds to the following Gaussian conditional distribution:

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

where $\beta_t \in (0, 1)$ are variance coefficients chosen according to a **noise schedule**, typically increasing over time such that:

$$\beta_1 < \beta_2 < \dots < \beta_T$$

With this setup, you can verify that as β_t increases, the mean gradually moves toward zero and the variance approaches one, ensuring that the data distribution converges to $\mathcal{N}(0, I)$.

Why use $\sqrt{1 - \beta_t}$ and $\sqrt{\beta_t}$? While other formulations are mathematically possible, the authors of the DDPM paper opted for this specific choice due to its **simplicity** and effective behavior in practice. The authors proposed this **linear noise schedule**, however alternative schedules have then been proposed. One notable example is the **cosine noise schedule**, introduced in the paper “*Improved Denoising Diffusion Probabilistic Models*” [128].

The following Figure shows the difference between using a linear schedule and cosine schedule for the forward diffusion process. A linear schedule is displayed in the first row, while the second row demonstrates the improved cosine schedule.



The authors of the *Improved DDPM* paper argue that the cosine schedule offers superior performance. A linear schedule may lead to a rapid loss of information in the input image. As a result, this generally leads in an abrupt diffusion process. In contrast, the cosine schedule provides smoother degradation, hence allowing the later steps to operate on images that are not completely overwhelmed by noise.

Diffusion Kernel

As discussed above, the forward diffusion process in DDPM is defined by the recursive formula:

$$x_t = \sqrt{1 - \beta_t} x_{t-1} + \sqrt{\beta_t} \epsilon_t$$

Now, a natural question arises: Can we derive a **closed-form expression for $q(x_t|x_0)$** which allows us to directly draw samples x_t given initial data point x_0 without computing the intermediate variables x_1, \dots, x_{t-1} ? This would be highly beneficial — especially when t is large — since computing each step sequentially can be time-consuming. The resulting expression is known in literature as the **diffusion kernel**.

To derive an expression for $q(x_t|x_0)$, consider the first two steps of the forward process:

$$\begin{aligned} x_1 &= \sqrt{1 - \beta_1} x_0 + \sqrt{\beta_1} \epsilon_1 \\ x_2 &= \sqrt{1 - \beta_2} x_1 + \sqrt{\beta_2} \epsilon_2 \end{aligned}$$

Substituting the first equation into the second, we get:

$$\begin{aligned} x_2 &= \sqrt{1 - \beta_2} x_1 + \sqrt{\beta_2} \epsilon_2 = \\ &= \sqrt{1 - \beta_2} (\sqrt{1 - \beta_1} x_0 + \sqrt{\beta_1} \epsilon_1) + \sqrt{\beta_2} \epsilon_2 = \\ &= \sqrt{1 - \beta_2} (\sqrt{1 - \beta_1} x_0 + \sqrt{1 - (1 - \beta_1)} \epsilon_1) + \sqrt{\beta_2} \epsilon_2 = \\ &= \sqrt{(1 - \beta_2)(1 - \beta_1)} x_0 + \sqrt{1 - \beta_2 - (1 - \beta_2)(1 - \beta_1)} \epsilon_1 + \sqrt{\beta_2} \epsilon_2 \end{aligned}$$

Since the last two terms are independent samples from zero-mean normal distributions with variances $1 - \beta_2 - (1 - \beta_2)(1 - \beta_1)$ and β_2 respectively, the sum of these terms remains Gaussian. In particular, the mean of this sum is zero and its variance is the sum of the component variances. Thus, the overall result can be also rewritten as:

$$x_2 = \sqrt{(1 - \beta_2)(1 - \beta_1)} x_0 + \sqrt{1 - (1 - \beta_2)(1 - \beta_1)} \epsilon$$

where ϵ is also a sample from a standard normal distribution. If we continue this process by substituting this equation into the expression for x_3 and so on, we can show that:

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

where

$$\alpha_t = 1 - \beta_t \quad \bar{\alpha}_t = \prod_{s=1}^t \alpha_s = \prod_{s=1}^t (1 - \beta_s)$$

We can equivalently write this in probabilistic form as:

$$q(x_t|x_0) = \mathcal{N}(\sqrt{\bar{\alpha}_t} x_0, (1 - \bar{\alpha}_t) I)$$

For any starting data point x , the variable x_t is normally distributed with known mean and variance. Consequently, if we don't care about the history of the evolution through the intermediate variables x_1, \dots, x_{t-1} , it is easy to generate samples from $q(x_t|x_0)$.

Again, you can easily see that with this formulation after many steps the image becomes indistinguishable from Gaussian noise, and in the limit $T \rightarrow \infty$ we have:

$$q(x_t|x_0) = \mathcal{N}(0, I)$$

If we keep the variances small so that $\beta_t \ll 1$ then the change in the image vector between steps will be relatively small and hence it should be easier to learn to invert the transformation. However, since the variances at each step are small, we must use a **large number of steps** to ensure that the distribution over the final x_t obtained from

the forward noising process will still be close to a standard Gaussian and this increases the cost of generating new samples. In practice, T may be set to $T = 1000$.

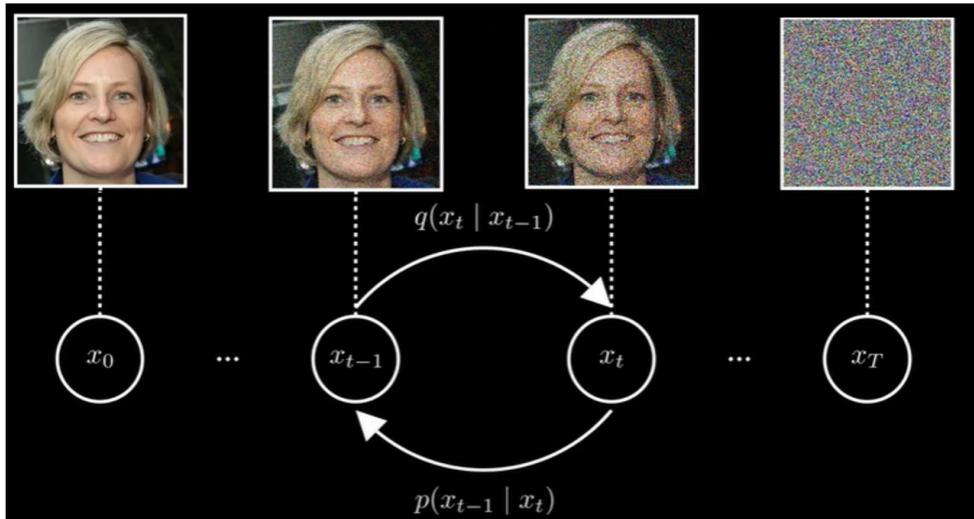
So, we have now defined a proper forward diffusion process that will **progressively transform any distribution into a standard Gaussian**. This process is conceptually similar to the **encoder in a VAE**, except that here it is **fixed and non-parametric** rather than learned.

Let's now move on to how we can **reverse** this forward diffusion process to recover data from noise.

Reverse Diffusion Process

So far, we described the forward diffusion process which gradually transforms a clean image x_0 into a sample of Gaussian noise x_t using a sequence of fixed conditional distributions $q(x_t | x_{t-1})$. Now, we turn our attention to the **reverse diffusion process** (or **denoising**): the goal is to learn a probabilistic model that can gradually denoise a sample x_t , step by step, to recover an image resembling the original x_0 . We represent this reverse diffusion process using a probability distribution p , which models the conditional probability of transitioning from x_t to x_{t-1} :

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



Unlike the forward process — where all parameters are fixed — for the reverse process our goal is to find the best parameters θ that allow us to remove the noise effectively and as you've probably guessed these parameters θ will be the ones of a neural network.

$$p_\theta(x_{t-1}, x_t)$$

Just as the forward process defines a Markov chain using conditionals $q(x_t | x_{t-1})$, the reverse process is also **Markov chain** (since each de-noising step depends only on the next step along the chain), modeled by a chain of conditionals $p_\theta(x_{t-1}, x_t)$. We can express the complete reverse diffusion process in a more compact form as:

$$p_\theta(x_{0:T}) = p_\theta(x_T) \prod_{t=1}^T p_\theta(x_{t-1} | x_t)$$

Note that the reverse process is **not conditioned on x_0** — unlike the forward process — because we start from pure Gaussian noise and aim to recover an image without knowing the original x_0 .

This reverse step is conceptually analogous to the **decoder in a VAE**, but here it is learned through a sequence of denoising steps instead of a single step.

Now, to approximate each of these reversed distributions we can think of using a normal distribution of the form:

$$p_\theta(x_{t-1}, x_t) = \mathcal{N}(\mu_\theta(x_t, t), \sigma^2 I)$$

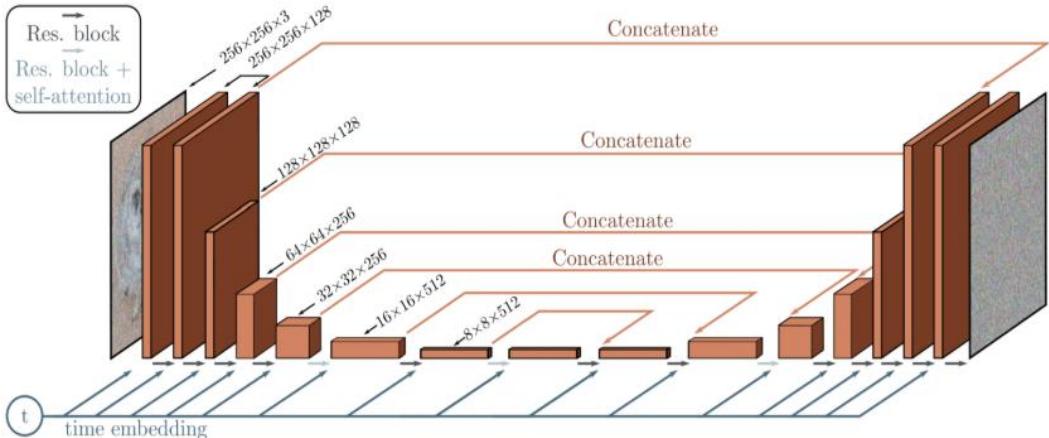
where μ_θ is the prediction of a deep neural network governed by a set of parameters θ and the variance is chosen to be spherical $\sigma_t^2 I$ with σ_t^2 typically set to match the forward variance, i.e. $\sigma_t^2 = \beta_t$, and is **not learned**. Thus, we can also directly write this expression as:

$$p_\theta(x_{t-1}, x_t) = \mathcal{N}(\mu_\theta(x_t, t), \beta_t I)$$

Therefore, the neural network **only needs to learn the mean** of the reverse distribution. The variance is predefined and varies across timesteps according to the noise schedule, which simplifies things even further.

Note that the network takes the step index t explicitly as an input so that it can account for the variation of the variance β_t across different steps of the chain. This allows us to use a **single network** to invert all the steps in the Markov chain, instead of having to learn a separate network for each step.

Lastly, given that in each step in the reverse process the provided output has the same dimensionality as the input a common choice for the neural network architecture used to model $\mu_\theta(x_t, t)$ is a ***U-Net***.



Regarding the U-Net implementation details, the main architectural choices involve:

- The **contracting** (downsampling) and **expanding** (upsampling) paths have the same number of layers, incorporating a bottleneck block between them.
- Each **contracting** stage consists of two Residual Blocks with convolutional layers, while each **expansive** stage includes three residual blocks along with transpose convolutional layers.
- The expansive path connects to each stage in the contracting path through **skip connections**.
- **Self-attention** is used at selected feature resolutions to enhance long-range dependencies.
- Additionally, the timestep t is encoded into a **time embedding** using a **sinusoidal positional encoding**.

These time embeddings help the neural network to gain certain information of at which state (step) is the image currently at. This is useful to know if more or less noise is currently present in the image, making the model subtract

more or less noise. We recall that in lower timesteps the forward diffusion process adds less noise than in higher timesteps.

Training Loss

Now, how do we actually train the neural network that models the reverse diffusion process? As we did for VAE we can think to maximize the log likelihood of the data under our generative model. This means finding the set of parameters θ that maximizes the (log) probability of generating real samples x_0 from our data distribution using the neural network:

$$\max_{\theta} \log p_{\theta}(x_0)$$

Equivalently, we can formulate this as **minimizing the negative log-likelihood**, which is the optimization objective followed by the authors of the DDPM paper:

$$\min_{\theta} -\log p_{\theta}(x_0)$$

Let's derive an expression for this negative log-likelihood.

Generally speaking, when dealing with joint probabilities a useful first step is often to marginalize the distribution with respect to the other variables:

$$-\log p_{\theta}(x_0) = -\log \int p_{\theta}(x_{0:T}) dx_{1:T}$$

In other words, we integrate out x_1, x_2, \dots, x_T to obtain the marginal probability of x_0 .

However, this integral is **intractable** in practice — there are far too many possible diffusion trajectories for which we would need to compute $p_{\theta}(x_{0:T})$. But as in VAEs, we can apply a very well-known trick: we multiply and divide by the density representing our forward process $q(x_{1:T}|x_0)$:

$$= -\log \int q(x_{1:T}|x_0) \frac{p_{\theta}(x_{0:T})}{q(x_{1:T}|x_0)} dx_{1:T}$$

Now we can rewrite this integral as an expectation making the expression slightly more compact:

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

Since the negative log function is convex, we can make use of *Jensen's inequality* to move the log inside the expectation at the cost of introducing an inequality:

$$-\log \mathbb{E}_{q(x_{1:T}|x_0)} \left[\frac{p_{\theta}(x_{0:T})}{q(x_{1:T}|x_0)} \right] \leq \mathbb{E}_{q(x_{1:T}|x_0)} \left[-\log \frac{p_{\theta}(x_{0:T})}{q(x_{1:T}|x_0)} \right]$$

This gives us an **upper bound** on the negative log-likelihood, which is equivalent to the **negative Evidence Lower Bound (negative ELBO)**:

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

Therefore, to train the diffusion model we can **minimize** this negative ELBO loss, which approximates the negative log-likelihood of the data.

Simplifying the Negative ELBO

Now we'll try to make sense of this training objective by modifying and simplifying its expression further. I'm going to skip the detailed algebraic derivation for brevity — although it mostly involves applications of Bayes' rule and standard Gaussian identities. If you're interested in full derivation, it's provided in **Appendix A** of the DDPM paper's supplementary material.

Through a series of transformations, the negative ELBO objective can be rewritten as:

$$\mathbb{E}_q \left[D_{KL}(q(x_T | x_0) || p(x_T)) + \sum_{t>1} D_{KL}(q(x_{t-1} | x_t, x_0) || p_\theta(x_{t-1} | x_t)) - \log p_\theta(x_0 | x_1) \right]$$

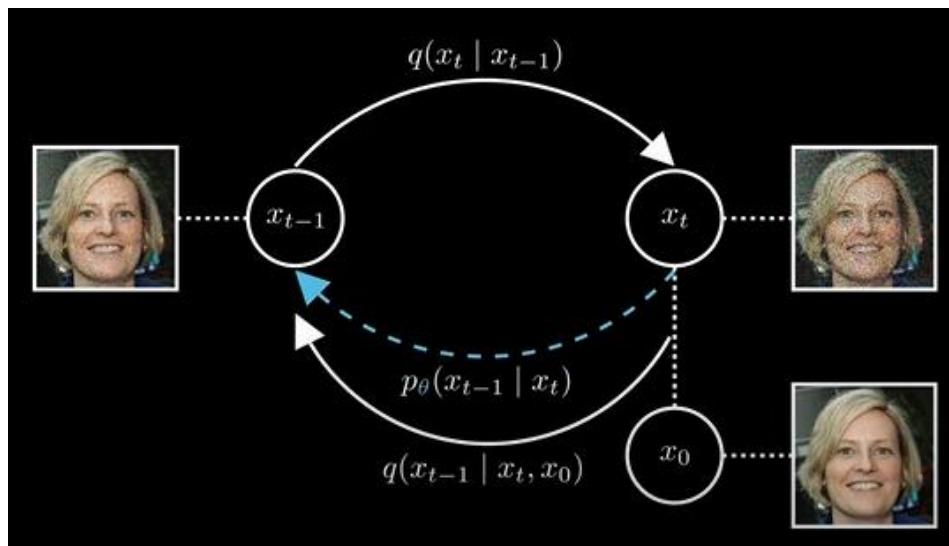
This form might look a bit intimidating at first, but as usual there's some intuition behind it. We can simplify this expression further:

- The first term on the left doesn't depend on our model parameters θ so we can ignore it when optimizing the model.
- The last term on the right instead does depend on the parameters θ of the network and represent the likelihood of recovering the original clean image x_0 from the nearly clean sample x_1 . Since the noise level at step $t = 1$ is very low, this term tends to be small and contributes little to learning. In practice, omitting it has little effect on performance, so it's often dropped.

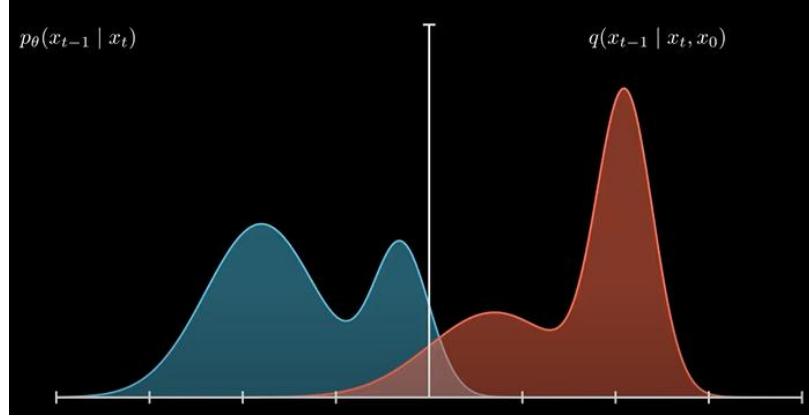
That leaves us with the **core training loss**, a big sum of KL divergences between some distributions:

$$\mathbb{E}_q \left[\sum_{t>1} D_{KL}(q(x_{t-1} | x_t, x_0) || p_\theta(x_{t-1} | x_t)) \right]$$

We know that $p_\theta(x_{t-1} | x_t)$ is our reverse diffusion step, but what about the other distribution, $q(x_{t-1} | x_t, x_0)$? It's the exact reverse distribution of the forward process $q(x_t | x_{t-1})$, but **conditioned on the original image x_0** . In other words, if we know both x_t (the noisy image) and the original clean image x_0 , we can compute exactly what the denoised image x_{t-1} should look like.



So why not just use this true posterior directly? Because at inference time, we're trying to generate x_0 from noise — so we **don't** have access to the original image. Using the true posterior would be cheating, since it assumes knowledge of the ground truth we're trying to generate. However, during training we *do* know x_0 , so we can use the true posterior to guide learning. So, what we're really doing with this loss function is **training the neural network to match the true posterior**, the one we could compute if we had access to x_0 , even though in practice the network only sees x_t .



Now about the expression of this true posterior, I'll skip the derivations again because it's mostly algebra and Bayes rule application. In the end it turns out that the **true posterior** is Gaussian with a mean $\tilde{\mu}_t(x_t, x_0)$ and a variance $\tilde{\beta}_t I$:

$$q(x_{t-1} | x_t, x_0) = \mathcal{N}(\tilde{\mu}_t(x_t, x_0), \tilde{\beta}_t I)$$

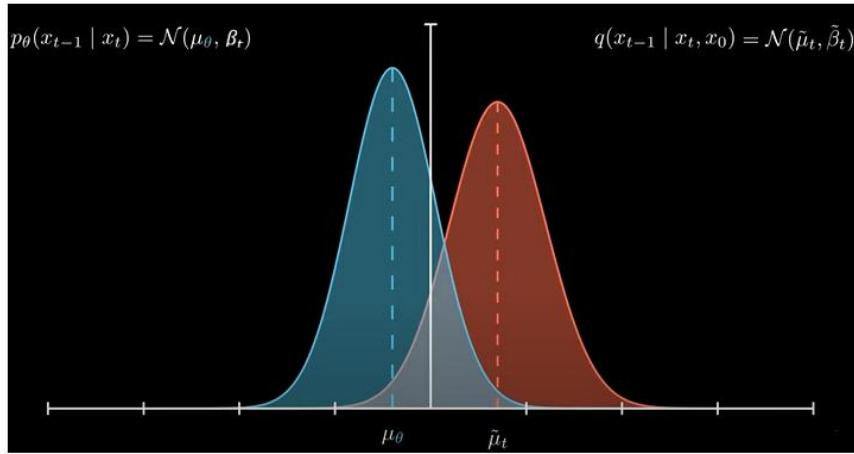
$$\text{where } \tilde{\beta}_t = \frac{1 - \bar{\alpha}_{t-1}}{1 - \bar{\alpha}_t} \beta_t.$$

We recall that for our **approximate posterior** we chose:

$$p_\theta(x_{t-1}, x_t) = \mathcal{N}(\mu_\theta(x_t, t), \beta_t I)$$

where, with this convenient form, we need to **just learn the mean** μ_θ of the distribution.

Since we're trying to match two Gaussian distributions but can only control the mean of the approximate one, the best we can do is bring its mean as close as possible to the mean of the true posterior.

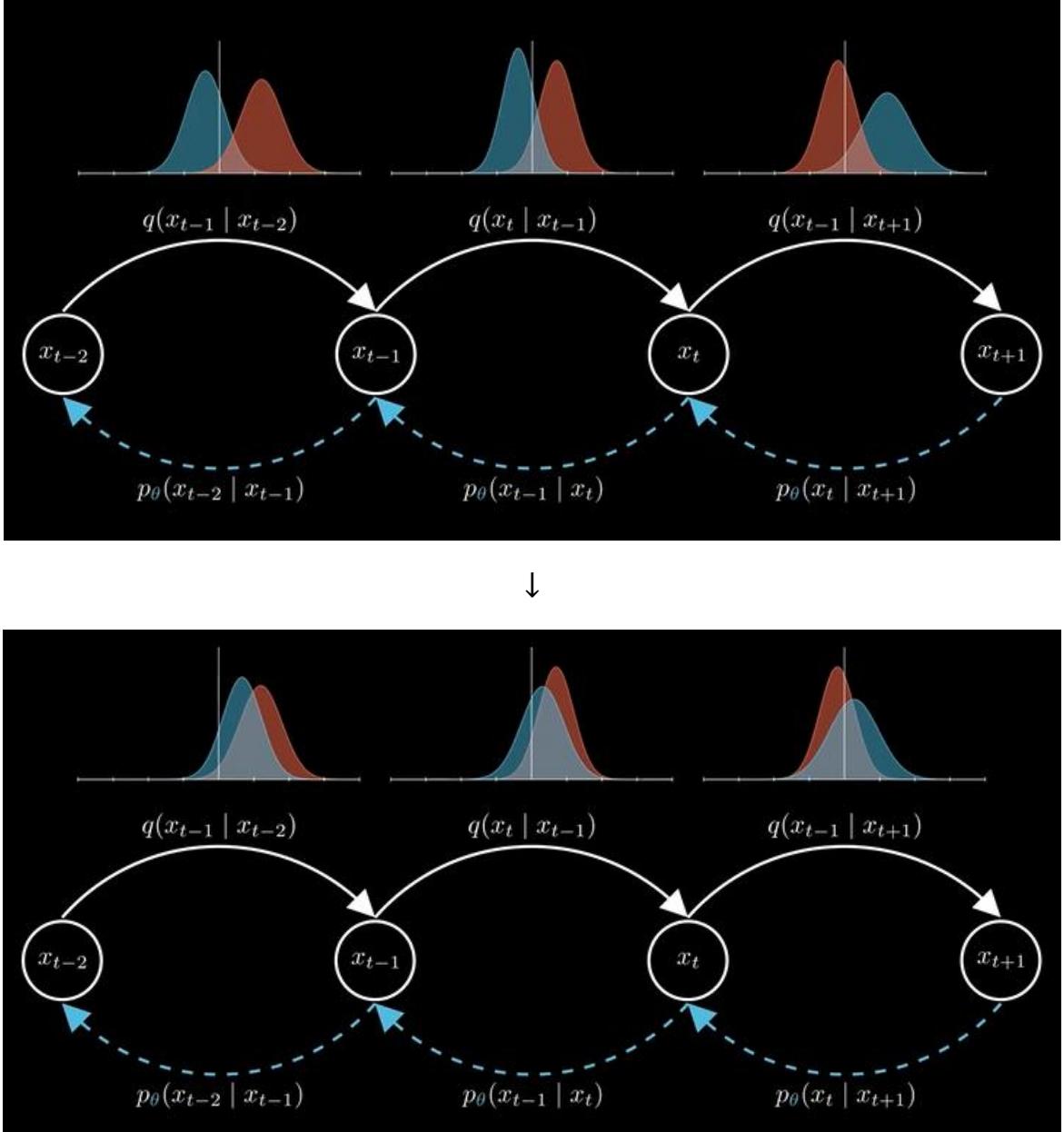


Said differently, we aim to **minimize the distance between the two means** and in fact that's exactly how the KL divergence behaves. In fact, each of these KL divergence term simplifies to a **squared error** between the means of the two distributions:

$$D_{KL}(q(x_{t-1} | x_t, x_0) || p_\theta(x_{t-1} | x_t)) = \frac{1}{2\sigma_t^2} \|\tilde{\mu}_t(x_t, x_0) - \mu_\theta(x_t, t)\|^2$$

So, at each timestep t , minimizing the KL divergence simply means bringing the predicted mean μ_θ closer to the true mean $\tilde{\mu}_t$.

Zooming out, the overall training objective — minimizing the sum of these KL terms — is just encouraging the network to match the true posterior means as closely as possible at every step.



Final Expression for the Diffusion Loss

Now that we have a better intuition of the negative ELBO, we'll simplify it further to state the final training objective. We just saw that each of the KL divergence terms in this sum simplifies into a squared distance, so overall we're minimizing a sum of squared distances:

$$\begin{aligned} & \mathbb{E}_q \left[\sum_{t>1} D_{KL}(q(x_{t-1}|x_t, x_0) || p_\theta(x_{t-1}|x_t)) \right] \\ & \rightarrow \mathbb{E}_q \left[\sum_{t>1} \frac{1}{2\sigma_t^2} \|\tilde{\mu}_t(x_t, x_0) - \mu_\theta(x_t, t)\|^2 \right] \end{aligned}$$

Each of these terms measures how far the predicted mean μ_θ is from the true posterior mean $\tilde{\mu}$ and the distance is weighted by a factor that depends on $\sigma_t^2 = \beta_t$.

We do have a closed form expression for the true posterior mean $\tilde{\mu}$, but it is a bit complicated (here I skip the math steps used for obtaining it):

$$\tilde{\mu}_t(x_t, x_0) = \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1 - \bar{\alpha}_t}x_0 + \frac{\sqrt{\alpha_t}(1 - \bar{\alpha}_{t-1})}{1 - \bar{\alpha}_t}x_t$$

As you can see it's a weighted combination of x_0 and x_t , which is not very convenient to work with.

Now recalling the expression of kernel diffusion we have obtained at the beginning:

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

and applying a tiny bit of algebra, we can also write x_0 as:

$$x_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}(x_t - \sqrt{1 - \bar{\alpha}_t}\epsilon)$$

If we plug this into the expression of $\tilde{\mu}_t$, we also get an expression that depends only on x_t and ϵ :

$$\tilde{\mu}_t = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}}\epsilon\right)$$

Then comes the **key idea**. Since the denoiser also has access to x_t , we can apply the same trick to the network's predicted $\tilde{\mu}_t$ and we write it as a function of x_t and a predicted noise term ϵ_θ :

$$\mu_\theta = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}}\epsilon_\theta(x_t, t)\right)$$

What we did was simply to express $\tilde{\mu}_t$ and μ_θ in function of x_t and ϵ/ϵ_θ . Now if we plug both of these expressions into the loss, the terms in x_t cancel out, revealing our final objective function:

$$\begin{aligned} & \mathbb{E}_q \left[\sum_{t>1} \frac{\beta_t^2}{2\sigma_t^2 \alpha_t (1 - \bar{\alpha}_t)} \|\epsilon - \epsilon_\theta(x_t, t)\|^2 \right] = \\ &= \mathbb{E}_q \left[\sum_{t>1} \frac{\beta_t^2}{2\sigma_t^2 \alpha_t (1 - \bar{\alpha}_t)} \|\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon, t)\|^2 \right] \end{aligned}$$

where we have rewritten x_t using the diffusion kernel in the second line.

We're left with a much simpler loss, a plane **squared distance between ϵ and ϵ_θ** . So, **ϵ_θ becomes the network's estimate of the noise that was added to x_0 to produce x_t** and minimizing this loss just means making this prediction as accurate as possible.

We can do a final simplification. As you can see, summing up over thousands of time steps for every sample is pretty expansive. Instead of summing up all time steps, we can simply **pick a single random time step t for each sample**, and we know that for a large number of samples considered during training, this will converge towards the original objective:

$$\mathbb{E}_q \left[\frac{\beta_t^2}{2\sigma_t^2 \alpha_t(1-\bar{\alpha}_t)} \|\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t)\|^2 \right]$$

Moreover, since now **we are not computing expectations over the entire diffusion chain anymore**, we can sample directly from $q(x_t|x_0)$ in closed form (by taking a training data point x_0 , sampling $\epsilon \sim \mathcal{N}(0, I)$ and then computing $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$) and therefore re-write the expectation as:

$$\mathcal{L}(\theta) = \mathbb{E}_{x_0, \epsilon, t} \left[\frac{\beta_t^2}{2\sigma_t^2 \alpha_t(1-\bar{\alpha}_t)} \|\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t)\|^2 \right]$$

Lastly, by absorbing all the time-dependent scalar coefficients into a single weighting term γ_t , we can rewrite the loss as:

$$\mathcal{L}(\theta) = \mathbb{E}_{x_0, \epsilon, t} [\gamma_t \|\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t)\|^2]$$

and this is the **final loss expression**.

In practice, the DDPM authors proposes to **set $\gamma_t = 1$ for all t** , which leads to the so-called **simplified objective**:

$$\mathcal{L}_{simple}(\theta) = \mathbb{E}_{x_0, \epsilon, t} [\|\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t)\|^2]$$

This choice was **empirically motivated** by the observation that it produces **high-quality samples** while avoiding unnecessary complexity associated with the weighting terms. However, more recent work, such as *Kingma & Gao (2023)* [129], has explored to also treat γ_t as a **learnable parameter**, potentially improving likelihood estimation and thus enhance generative performance.

Note: In many papers, this loss is often reported in terms of x_t rather than x_0 (i.e. before applying diffusion kernel), so here I report also it just for completeness:

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

Inference

Once the network has been trained, at inference time we can generate new samples by running the reverse diffusion process. The process begins by sampling a vector of pure noise $x_T \sim p(x_T) = \mathcal{N}(0, I)$ and then iteratively denoising it through each step of the learned reverse Markov chain finally obtaining a noise-free sample x_0 .

Note: To perform inference correctly, it's essential to use the exact same noise schedule $\{\beta_1, \dots, \beta_T\}$ and all derived constants, like α_t and $\bar{\alpha}_t$, that were used during training. If these differ the noise levels at each step will be inconsistent and model's predictions won't correspond properly to the learned denoising behavior.

In particular, at each reverse diffusion step t , we perform the following steps to generate x_{t-1} from x_t :

1. First, we feed x_t and timestep t into the neural network to predict $\epsilon_\theta(x_t, t)$, i.e. the noise that was added to x_t .
2. Using this predicted noise, we estimate the mean $\mu_\theta(x_t, t)$ using
$$\mu_\theta(x_t, t) = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}} \epsilon_\theta(x_t, t) \right)$$
3. Finally, we generate a sample x_{t-1} from the approximate posterior $p_\theta(x_{t-1}, x_t) = \mathcal{N}(\mu_\theta(x_t, t), \beta_t I)$ by adding noise scaled by the variance so that

$$x_{t-1} = \mu_\theta(x_t, t) + \sqrt{\beta_t} \epsilon$$

where $\epsilon \sim \mathcal{N}(0, I)$. Note that this is equivalent to sample from a Gaussian centered at $\mu_\theta(x_t, t)$ with variance $\beta_t I$.

This reverse diffusion step is repeated for the same number of T steps used during training (as we said typically $T = 1000$).

Note that:

- At the beginning (step 1) the network predicts the *total* noise that was added to the original data sample x_0 to produce x_t , i.e. $\epsilon_\theta(x_t, t)$. However, during sampling (step 2), we don't remove all of this noise at once. Instead, we **subtract only a fraction of it** — specifically,

$$\frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}} \epsilon_\theta(x_t, t)$$

— when computing the $\mu_\theta(x_t, t)$ and then we **add fresh Gaussian noise** (step 3) with variance β_t to generate x_{t-1} ($\sqrt{\beta_t} \epsilon$).

- At the final timestep of the reverse process — when $t = 1$ and we want to generate x_0 — we do **not add any additional noise** since we are aiming to generate a noise-free sample, so we simply compute:

$$x_0 = \mu_\theta(x_t, t) = \frac{1}{\sqrt{\alpha_1}} \left(x_1 - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_1}} \epsilon_\theta(x_1, t) \right)$$

This yields the final synthetic data sample, free from added noise.

Diffusion models have shown to achieve significantly better results compared to state-of-the-art generative models based at the time on GANs. However, this improvement comes at a computational cost: unlike GANs, which typically require a single forward pass through the generator network to generate an image, **diffusion models must perform thousands of sequential inference steps**, making the sampling process considerably slower and more resource intensive. For example, generating 50,000 images of size 32×32 with a DDPM can take around 20 hours, whereas a GAN can produce the same amount in less than a minute.

So, in essence, diffusion models trade increased computational effort for superior image quality. However, what researchers really wanted was to maintain this high visual fidelity while drastically reducing the number of inference steps. Remarkably, just a few months after the release of the original DDPM paper, this challenge was addressed with the introduction of **accelerated sampling** techniques, such as **Denoising Diffusion Implicit Models (DDIM)** introduced by Song et al. [130], that enable high-quality samples in only a handful of steps without sacrificing image fidelity. This was then further pushed up in next years as we will also see at the end of the chapter.

Guided Diffusion

The basic diffusion model we've described so far is **unconditional** — it learns to generate samples from the overall data distribution without any control over the kind of image it produces. The output entirely depends on the random initial noise x_T we sampled since, based on how "close" the noise values are close to certain well-behaved image values, it can lead toward a given region of image space rather than another. For instance, one sample might result in a cat, another in a mountain and another in a car. So, we don't have a way to steer it toward a particular region of that space corresponding to a particular type of output we want.

Naturally, this lack of control is a limitation in many practical applications. We'd often like to generate images with specific properties: maybe we want an image of a particular class (say, a dog rather than a plane), or one that matches a text prompt or even an image conditioned on a sketch. This is where **guided diffusion** comes in.

Note: Although *guided diffusion* was mathematically derived from the formulation based on **diffusion score matching** (which will be addressed in a later section), I've chosen to introduce it here to maintain continuity with the narrative of classical DDPM we made so far, especially since the next section on **Stable Diffusion** also builds more naturally on the DDPM formulation. Of course, once score-based diffusion is introduced later, you'll be able to revisit this guidance technique and reinterpret it through the lens of score matching.

Classifier Guidance

Instead of just modeling $p(x)$ we want to generate samples from the **conditional distribution** $p(x|y)$, where y might be a class label or some other form of conditioning input. That way, the generated image is not just *likely under the data distribution* but *likely given the condition*.

Unfortunately, we don't have direct access to this conditional distribution. However, by applying Bayes' rule, we can re-express it in a more convenient form:

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)}$$

Taking the logarithm and then the gradient with respect to x gives:

$$\begin{aligned} \log p(x|y) &= \log p(y|x) + \log p(x) - \log p(y) \\ \nabla_x \log p(x|y) &= \nabla_x \log p(y|x) + \nabla_x \log p(x) - \underbrace{\nabla_x \log p(y)}_0 \end{aligned}$$

where $\nabla_x \log p(y) = 0$ since $p(y)$ is independent of x . This identity is extremely valuable: it says that to compute the gradient of the conditional log-probability ([what we care about when sampling in diffusion score matching models](#)) we can simply add to the unconditional gradient term $\nabla_x \log p(x)$, which we already know how to estimate with a diffusion model ([in particular with a diffusion score matching model as we will see in a next paragraph](#)), a new **conditioning gradient term** $\nabla_x \log p(y|x)$, which reflects how compatible a condition y is with the given input x . Here comes the clever twist: the term $p(y|x)$ is exactly what a **classifier** model gives in output! If we have a trained classifier that predicts the probability of label y given an image x , and if it's differentiable, we can compute $\nabla_x \log p(y|x)$ easily.

This leads to the **Classifier Guidance** method introduced by Dhariwal and Nichol (2021) [131]: Given an unconditional diffusion model and a **classifier** trained to predict y from noisy images x_t , we can modify the sampling process by injecting the classifier's gradient into the reverse diffusion step. Moreover, the influence of the classifier can be controlled by introducing a hyperparameter γ , called the **guidance scale**, which controls the importance given to the classifier gradient. So, at the end, we obtain:

$$\nabla_x \log p(x|y) = \nabla_x \log p(x) + \gamma \nabla_x \log p(y|x)$$

If $\gamma = 0$ we recover the original unconditional diffusion model $p(x)$, whereas if $\gamma = 1$ we obtain the true conditional model $p(y|x)$. For values of $\gamma > 1$ we **sharpen** the conditional distribution, pushing the model to generate outputs more consistent with the label y . This sharpening effect can be understood more intuitively by reversing our derivation. In fact, if we revert the gradient and the logarithm operations that we used to go from Bayes' rule to classifier guidance, we get the following expression for the guided distribution:

$$p_\gamma(x|y) \propto p(x) \cdot p(y|x)^\gamma$$

Here, the classifier term $p(y|x)$ is raised to the power γ , which can be interpreted as **temperature scaling**. When $\gamma > 1$, the distribution becomes more peaked: probability mass is concentrated around the modes of $p(y|x)$ and less likely regions are suppressed. In effect, this lowers the temperature of the conditional term, making the model more confident in its outputs.

Following this view, classifier guidance thus acts as a sort of mechanism to selectively sharpen the part of the distribution influenced by the conditioning signal — without affecting the underlying data distribution $p(x)$. The result is that samples become **more aligned with the target label**. However, this comes with a **tradeoff**: as γ increases, **sample diversity decreases**. The model favors "easy" examples — those that the classifier can label confidently — at the expense of variety and richness in the generated outputs.

While classifier guidance is a powerful technique, it comes with several notable limitations. It requires a **separate classifier** to be trained and more importantly this classifier needs to be capable of making accurate predictions across various levels of noise — a challenging requirement, since standard classifiers are typically trained only on clean images. Therefore, training such a noise-robust classifier can be cumbersome. Moreover, even if we have a such noise-robust classifier on hand, classifier guidance is inherently limited in its effectiveness: most of the information in the input x is not relevant to predicting y , and as a result, taking the gradient of the classifier w.r.t. its input can yield arbitrary (and even adversarial) directions in input space. This stems from the fact that the classifier can ignore a lot of the image structure while still classifying correctly.

To address these limitations, we turn to a more elegant alternative that **eliminates the need for a separate classifier: classifier-free guidance**.

Classifier-Free Guidance

Instead of explicitly training a separate classifier to estimate $p(y|x)$, the idea behind **Classifier-Free Guidance (CFG)**, introduced by Jonathan Ho & Tim Salimans (2022) [132], is to train a **single diffusion model** able to model **both** the conditional distribution $p(x|y)$ and the unconditional distribution $p(x)$.

But how do we can reach this? Let's revisit Bayes' rule and rearrange it in the opposite direction to express the conditional probability $p(y|x)$ in terms of $p(x|y)$ and $p(x)$:

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)}$$

$$\rightarrow \nabla_x \log p(y|x) = \nabla_x \log p(x|y) - \nabla_x \log p(x)$$

Plugging this into the earlier classifier guidance formula, we can rewrite it entirely in terms of the gradients of the conditional and unconditional distributions:

$$\nabla_x \log p(x|y) = \nabla_x \log p(x) + \gamma \nabla_x \log p(y|x)$$

$$\rightarrow \nabla_x \log p(x|y) = \nabla_x \log p(x) + \gamma (\nabla_x \log p(x|y) - \nabla_x \log p(x)) =$$

$$= (1 - \gamma) \nabla_x \log p(x) + \gamma \nabla_x \log p(x|y)$$

The insight here is that both $\nabla_x \log p(x)$ and $\nabla_x \log p(x|y)$ can now be derived from diffusion model training. However, we still have a problem: this requires training two separate models, one conditional and one unconditional. To get around this, the authors propose a simple trick: **train a single conditional diffusion model with conditioning dropout**. During training, we randomly drop the condition y a certain percentage of the time (typically 10-20%). When dropped, we replace it with a placeholder value that represents "no condition". As a result, the model learns to generate images with or without guidance using the same network.

One might think that this approach would compromise conditional modeling performance, but in practice, the impact appears to be minimal. Empirical results consistently show that classifier-free guidance yields significantly higher quality outputs than traditional classifier guidance. This improvement stems from a fundamental difference: while a classifier $p(y|x)$ can ignore most of the input vector x as long as it makes a good prediction of

y , classifier-free guidance is based on modeling the conditional density $p(x|y)$, which requires the model to assign high probability to *all* relevant aspects of x .

An additional strength of classifier-free guidance is that it eliminates the need to train a separate, noisy classifier that can lead to adversarial gradients. Since both gradient terms come from the same generative model, the conditioning remains robust and aligned throughout the sampling process. Moreover, *conditioning dropout* is trivial to implement during training.

The classifier-free guidance integrates conditioning information directly into the generative model, which not only simplifies the architecture but also broadens the ways and the type of information we can use for conditioning. In practice, this usually takes the form of adding an embedding y in a similar way to how the timestep embedding is added. For example, when building a text-guided diffusion model, the conditioning input can be a general text sequence — a prompt — and not simply a selection from a predefined set of class labels. The model can incorporate this prompt by feeding in a representation generated by a Transformer-based language model and using it in two complementary ways: or by concatenating the textual embedding with the input to the denoising network or also allowing cross-attention layers within the denoising network to attend to the text token sequence (we will see practically how this can be done with Stable Diffusion).

Lastly, I want to underline another aspect. Recall the combined gradient:

$$\nabla_x \log p(x|y) = (1 - \gamma) \nabla_x \log p(x) + \gamma \nabla_x \log p(x|y)$$

For $\gamma = 0$ we recover the unconditional model and for $\gamma = 1$ we get the standard conditional model, similarly to what is used in classifier guidance. However, it is when $\gamma > 1$ that the magic happens. This is where the model begins to strongly emphasize the conditioning signal, producing results that are not only more detailed but also more semantically faithful to the prompt.

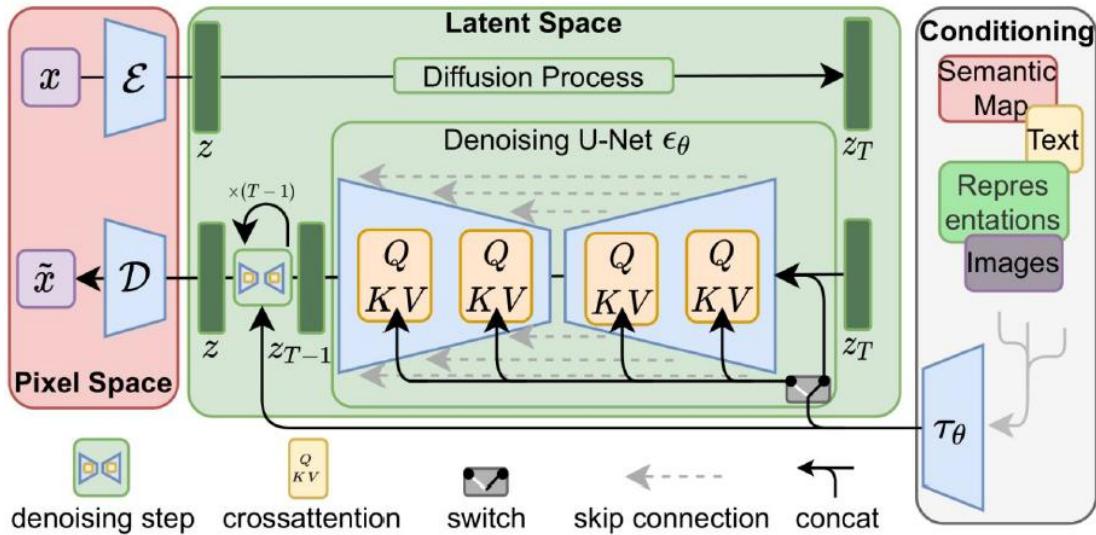


For example, in the Figure above are shown the results obtained from OpenAI’s **GLIDE** [133], a model which uses classifier-free guidance conditioned on a text prompt. It is important to see how increasing the guidance scale intensifies the influence of the conditioning, producing outputs that are sharper, more detailed and better aligned with the prompt. Take, for example, the prompt “a stained glass window of a panda eating bamboo”. At a guidance scale of $\gamma = 1$ (images on the left), the generated image might have the feel of a stained glass window, but no clear panda. However, increasing the guidance scale to $\gamma = 3$ (images on the right) results in a vivid, well-defined panda surrounded by bamboo with vivid colors that perfectly match the description.

Note: It is worth noting that there was only a very brief window of time between the publication of the classifier-free guidance paper and the OpenAI's GLIDE model. GLIDE effectively showcased the power of this technique, so that the idea has sometimes been attributed to the latter!

Stable Diffusion

Traditional diffusion models operate directly in pixel space, where the reverse process involves iteratively denoising a full-resolution image — typically by passing it through a U-Net at every diffusion step. While powerful, this setup becomes computationally demanding, particularly when generating high-resolution images or using many denoising steps T . Each step involves expensive operations in a high-dimensional space, making the entire sampling process slow and resource-intensive. This inefficiency prompted the development of **Stable Diffusion**, initially introduced as the **Latent Diffusion Model (LDM)** (2022) [134], which significantly improves scalability by shifting the diffusion process from the high-dimensional image space to a learned latent space. This shift is motivated by the observation that the process of a generative model can be decoupled into two conceptual stages: a first stage that handles “perceptual compression”, removing high-frequency pixel-level detail while retaining structure, and a second that focuses on “semantic generation”. Instead of applying diffusion in raw pixel space, LDM first maps images into a **perceptually equivalent**, but much lower-dimensional latent space using a **pre-trained VAE** and then, at this latent space, the actual diffusion process takes place. This reduces the dimensionality of the data the diffusion model must process, significantly improving efficiency without sacrificing output quality.



In particular, the following steps are taken:

- First, the **VAE encoder** \mathcal{E} maps the input image $x \in \mathbb{R}^{C \times H \times W}$ into a latent representation $z = \mathcal{E}(x) \in \mathbb{R}^{c \times h \times w}$
- Note:** The authors found that for inputs consisting of RGB images of dimensions 512×512 using a spatial resolution of 64×64 and 4 latent channels provided the optimal tradeoff between compactness and semantic expressiveness, since this configuration preserves enough structure for rich image generation while substantially reducing computational overhead. This compression ratio is the result of *empirical investigations* into various spatial and channel configurations, and you can check the paper for all comparisons.
- Once in the latent space, the **diffusion process mirrors that of standard DDPM**, but now in latent space rather than in the image space. The latent z is gradually noised over T steps to yield $z_T \sim \mathcal{N}(0, I)$ and the model learns to denoise z_T back to z using a U-Net.

- The **VAE decoder** \mathcal{D} then maps the z obtained from the diffusion process back to the image space $\tilde{x} = \mathcal{D}(z)$, yielding the final generated image.

In traditional diffusion models, we saw that the training objective is expressed as:

$$\mathcal{L}_{DM} = \mathbb{E}_{x, \epsilon \sim \mathcal{N}(0,1), t} [\|\epsilon - \epsilon_\theta(x_t, t)\|^2]$$

whereas in LDMs, since diffusion operates on latents, the objective becomes:

$$\mathcal{L}_{LDM} = \mathbb{E}_{\epsilon(x), \epsilon \sim \mathcal{N}(0,1), t} [\|\epsilon - \epsilon_\theta(z_t, t)\|^2]$$

One of the most powerful aspects of Stable Diffusion is its ability to generate images conditioned on text prompts or other auxiliary data. This is implemented via **classifier-free guidance (CFG)**, allowing the model to condition on additional information without relying on an external classifier. During training, the diffusion model is exposed to paired data (x_i, y_i) , where x_i is an image and y_i is the corresponding conditioning signal (e.g., a text prompt). To ensure that the model can operate both with and without conditioning, **conditioning dropout** is employed: the condition y is randomly dropped during 10% of training steps. This trains a single network to handle both unconditional and conditional generation seamlessly.

Stable Diffusion supports **multiple conditioning modalities**, each integrated differently into the denoising U-Net:

- Text prompts:** Given a text prompt y , this is first encoded into 768-dimensional embedding using a frozen **CLIP text encoder** (in the released version; BERT was used in the original paper). This embedding is then injected into the U-Net denoising model via **cross-attention layers**, where at each layer the latent representation attends to the prompt embedding. In particular, using the (flattened) latent representation $\varphi_i(z_t)$ as queries and the text embedding $\tau_\theta(y)$ as both keys and values, the attention is computed as:

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

with projections from both the latent and text embedding learned through weight matrices:

$$Q = W_Q^{(i)} \varphi_i(z_t), \quad K = W_K^{(i)} \tau_\theta(y), \quad V = W_V^{(i)} \tau_\theta(y)$$

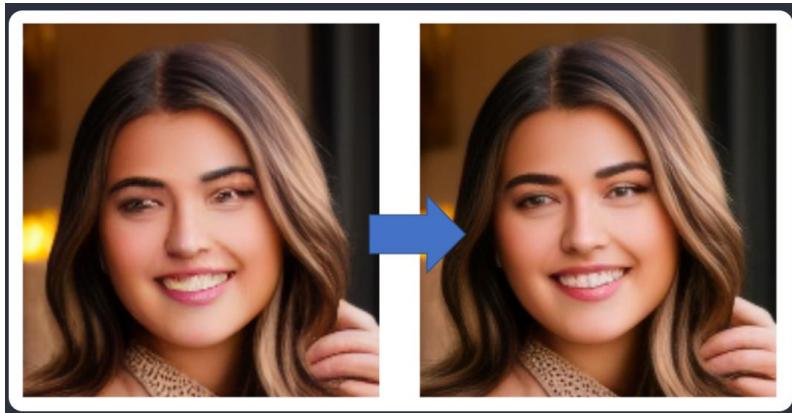
This allows the model to align latent features with the semantic content of the prompt at multiple levels in the U-Net, thereby generating images that are both semantically relevant and visually coherent.

- Other Conditioning Modalities:** For spatially structured inputs, such as semantic maps or partially masked images used for inpainting, conditioning is typically implemented by **concatenating** them with the latent representation.

This flexible conditioning mechanism allows Stable Diffusion to support a wide range of generation tasks — such as text-to-image, image-to-image, inpainting, depth-to-image ([which we'll talk about soon](#)) — by effectively integrating signals from multiple modalities into a unified generative framework.

Stable Diffusion was trained on LAION-400M, a large-scale dataset of image–text pairs which is publicly available and curated for high diversity and quality. Interestingly, only the image data of the dataset was used during the **pretraining phase of the VAE**; the textual information was introduced later during conditional training of the diffusion model. The VAE itself was trained to compress images into a latent space, learning to retain perceptually important details while discarding high-frequency noise and redundancy. However, this compression inevitably

results in some loss of fine detail. In practice, the VAE decoder becomes responsible for recovering and painting in those subtle visual elements — particularly for high-fidelity regions like eyes and faces. This is why, in many applications of Stable Diffusion, the decoder is further **fine-tuned separately** to enhance visual quality of the generated images. These **refined VAE decoder weights**, sometimes referred to simply as "**VAE files**", play a key role during inference in enabling the generation of fine-grained visual features such as eyes and facial details (or even textures) when mapping the denoised latent representation z back to image space.



Moreover, after training the base model, the authors performed an **additional fine-tuning phase** using the LAION-Aesthetics subset, biasing the model to generate images that exhibit greater aesthetic appeal.

Regarding image resolution, it's worth noting that Stable Diffusion v1 is trained specifically on **512×512** images. When generating images larger than this native size, artifacts can emerge — for instance, repeated objects such as duplicated heads. To mitigate this, it's typically recommended to constrain at least one image dimension to 512 pixels and then use an **AI upscaler** to increase the resolution afterward.

During inference, the generation process follows a now-familiar structure: starting from pure noise z_T in the latent space and a conditioning input y , the model iteratively denoises the latent through the trained U-Net. If conditioning on text, the prompt is first embedded using a frozen CLIP text encoder and the resulting embeddings guide the denoising through cross-attention layers at each level of the U-Net. For other modalities, such as segmentation maps or masked regions in inpainting, the conditioning input is concatenated directly with the latent tensor. After the latent vector is fully denoised to produce z , the VAE decoder maps it back into the image space to generate the final image x .

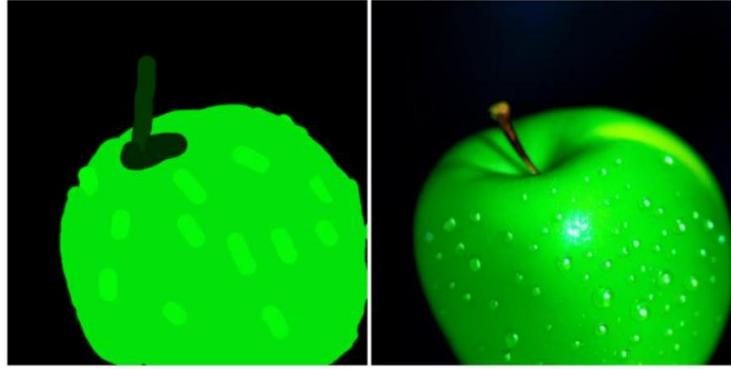
Other Conditioning Modalities in Stable Diffusion

While text prompts are the most common form of conditioning in Stable Diffusion permitting **text-to-image** task, the model architecture is flexible enough to support a wide range of additional conditioning types such as images, masks, depth maps and many others.

Note: It's important to note that **all** the generative tasks we'll discuss below involve **conditioning at inference time**. During training, the model is primarily trained on text-to-image pairs — such as those from the LAION dataset — with some additional image-conditioned examples possibly introduced during fine-tuning. However, once the model is trained, its weights are **frozen** and different conditioning strategies are applied **only during the sampling process**.

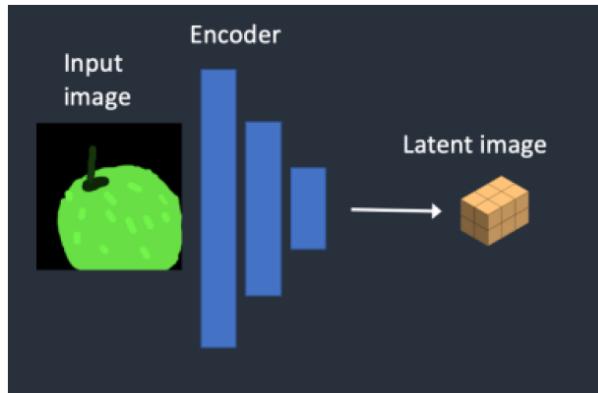
Image-to-Image Generation

In **image-to-image** tasks, both a source image and a text prompt are provided as inputs. The goal is to guide the generation process using the structural information of the input image, while refining or altering its appearance based on the prompt. For example, a rough sketch combined with the prompt “*a photo of a perfect green apple with stem, water droplets, dramatic lighting*” can yield a polished, photorealistic apple.



The process unfolds as follows:

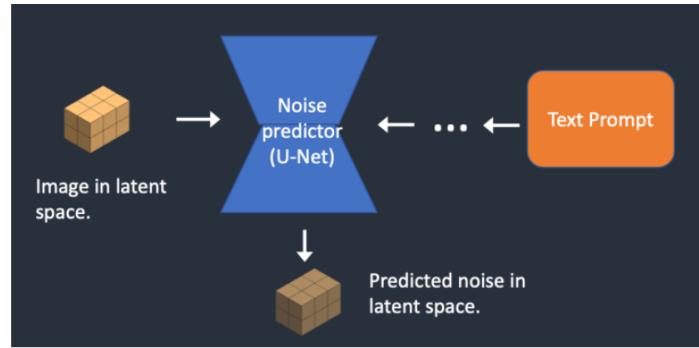
1. The input image is first encoded into the latent space using the VAE encoder.



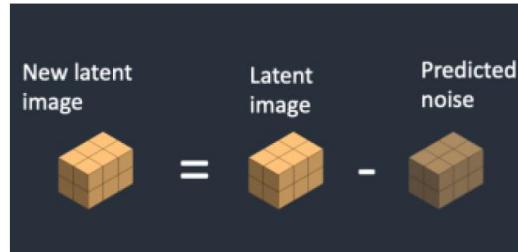
2. Controlled noise is then added **once** to this latent representation and the amount of noise is determined by a **denoising strength** parameter, which effectively determines how much of the original image is preserved:
 - A denoising strength of 0 means no noise is added, and the model attempts minimal modification.
 - A value of 1 adds maximum noise, effectively turning the latent into pure noise and allowing for complete reinterpretation by the model.

Technically, this is implemented by choosing a timestep t proportional to the denoising strength — higher values of t correspond to greater noise and less preservation of the original image. **Note:** This means that then the diffusion model will begin to **denoise from that timestep t** down to 0.

3. Parallelly, the **text prompt** is encoded into an embedding with CLIP.
4. The U-Net takes the noisy latent and the text embedding as input (via **cross-attention**) and predicts the residual noise.

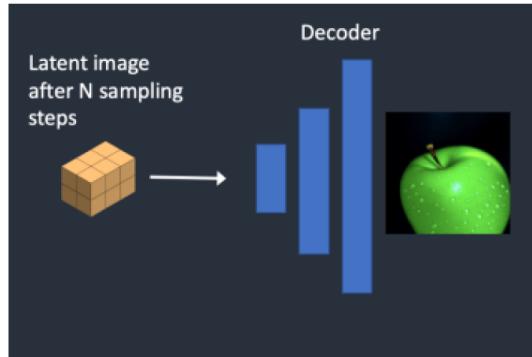


This predicted noise is subtracted from the latent obtaining the new latent image.



This denoising step is repeated over a number of timesteps.

- The final denoised latent is decoded by the VAE decoder back into image space, producing the final image.



This technique is ideal for translating rough sketches or stylized versions into more refined outputs guided by textual instructions.

Inpainting

Inpainting is essentially a special case of image-to-image generation, where only a portion of the image is modified. In this task the user masks out the region to be edited and the model has the goal to fill it in coherently based on the rest of the image (unmasked context) and a prompt.

The process unfolds as follows:

- The input image is first encoded into a latent representation using the VAE encoder.
- Noise is then **selectively added to the latent vector**, based on the binary mask and the specified **denoising strength**. The masked regions receive more noise (up to full noise at strength = 1), while the **unmasked regions are left intact**.
- This partially noised latent is passed into the U-Net denoiser:

- The **text prompt** is encoded with CLIP and injected into the U-Net via **cross-attention**.
 - The **binary mask** is often **concatenated to the latent** or passed as an additional channel to guide the denoising process, informing the model which areas are allowed to change.
4. The U-Net predicts the noise residual at each step. That residual is subtracted from the current latent and this denoising loop is repeated over a number of timesteps.
 5. Once the denoising is complete, the refined latent is passed through the VAE decoder to obtain the final image in pixel space.

This method allows for precise edits like object removal or replacing objects with other ones specified in the prompt.

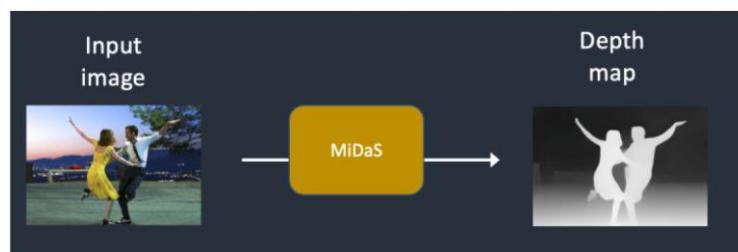


Depth-to-Image

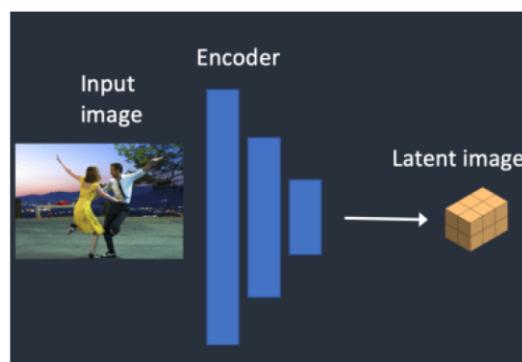
Depth-to-image builds on image-to-image generation by incorporating a *depth map* of the source image as an additional conditioning input. A **depth map** is an image where each pixel contains a value that indicates the distance (or depth) between the corresponding point in the image and the observer. The depth map provides a structural prior, helping the model preserve or reinterpret spatial layouts more realistically.

The process unfolds as follows:

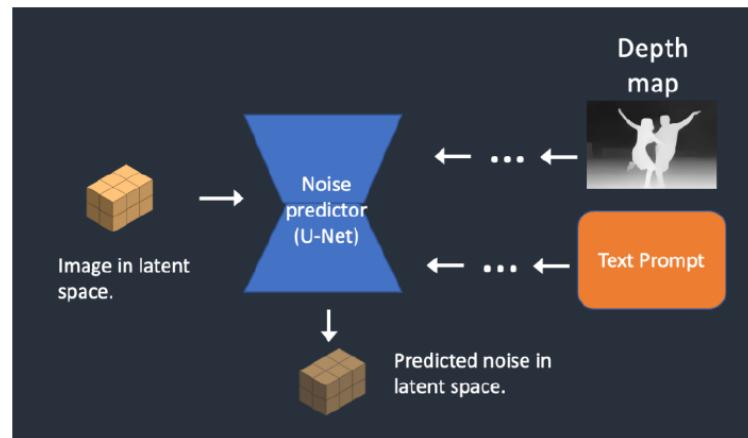
1. A pre-trained model like **MiDaS** is used to estimate a depth map from the source image.



2. The source image is encoded into latent space via the VAE encoder.



3. Noise is added to this latent representation, controlled by the denoising strength (as in image-to-image).
4. The noisy latent representation is then passed to the U-Net, which is **conditioned on both the text embedding (via cross-attention)** and the **depth map (concatenated to the latent)**, to predict the residual noise.



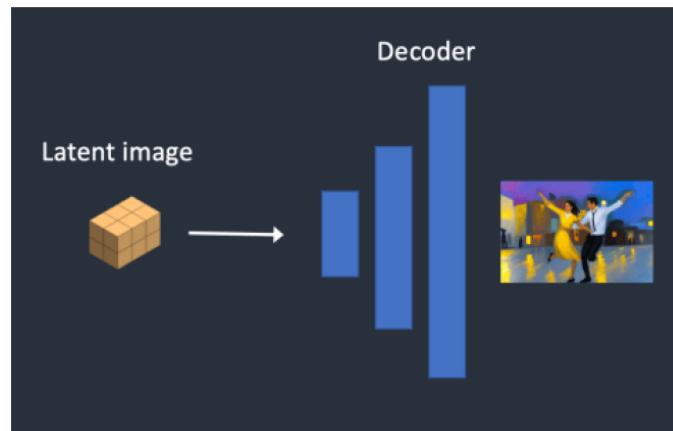
The predicted noise is then subtracted from the latent obtaining the new latent image.

$$\text{New latent image} = \text{Latent image} - \text{Predicted noise}$$

The equation shows the calculation: New latent image equals Latent image minus Predicted noise.

This denoising step is repeated over a number of timesteps.

5. The final denoised latent is decoded back into image space via the VAE decoder.



This technique allows for enhanced spatial realism in generated images, making it particularly effective for tasks that require consistent object geometry and depth-aware compositions.

Stable Diffusion v1 vs v2

Stable Diffusion was originally developed through a partnership between researchers at **CompVis** (University of Heidelberg) and **RunwayML**, in collaboration with **EleutherAI** and **LAION**. The first public release, **Stable Diffusion v1**, launched in August 2022, quickly gained popularity due to its high-quality image generation and open-source nature.

Following the success of v1, **Stability AI** was formally established as an independent company to fund, support and scale development of open generative models such as Stable Diffusion. It then led the creation and release of **Stable Diffusion v2** in November 2022, introducing several architectural and data improvements based on lessons learned from the original version.

Stable Diffusion v2 differs from its first version primarily in the text encoder, training data and the resulting model behavior during image generation:

- **Model Differences**

Stable Diffusion v1 uses OpenAI's CLIP ViT-L/14 as its text encoder, which contains approximately 63 million parameters. In contrast, Stable Diffusion v2 switches to **OpenCLIP** [135], a larger and fully open-source alternative trained by LAION, with versions scaling up to 354 million parameters. This switch was made for two main reasons:

- **Improved text encoding:** The larger OpenCLIP models lead to richer and more detailed text embeddings, which in turn improve image quality and prompt adherence.
- **Transparency and reproducibility:** Although Open AI's CLIP models are open-source, these models were trained with **proprietary data**. OpenCLIP model, on the other hand, is trained on open data, which gives researchers **more transparency in studying and optimizing the model**. This transparency is better also for long-term development.

- **Training Data Differences**

Stable Diffusion v1.4 was trained on a combination of LAION datasets:

- 237K training steps at 256×256 resolution on the LAION-2B-en dataset
- 194K steps at 512×512 resolution on LAION-high-resolution
- 225K steps at 512×512 on “LAION-aesthetics v2 5+”
- Text conditioning was randomly dropped 10% of the time for classifier-free guidance training (*conditioning dropout*).

In contrast, **Stable Diffusion v2** was trained on a more refined and filtered dataset:

- 550K steps at 256×256 resolution on a subset of LAION-5B, filtered for adult content using the LAION-NSFW classifier (with $p_{unsafe} \leq 0.1$) and images with aesthetic scores ≥ 4.5
- 850K steps at 512×512 on the same dataset, restricted to images with native resolution $\geq 512\times 512$
- 150K steps with a v-objective loss (see DDPM paper) on the same dataset
- Training was resumed for another 140K steps at 768×768 resolution

- **Outcome and Behavior Differences**

Users generally find it **harder to use Stable Diffusion v2 to control styles and generate celebrities**. Although StabilityAI did not explicitly filter out artist and celebrity names, their effects are much weaker in v2. This behavioral change likely stems from **differences in the training data**:

- OpenAI's CLIP models may have benefited from proprietary datasets containing high-quality, curated images of celebrities and artwork, leading to better learned representations for those subjects. Their data is probably highly filtered so that everything and everyone looks fine and pretty.

- In contrast, v2's training data — though more open and ethically filtered — may contain less of this specific content or have less consistent representations, resulting in weaker prompt associations for such subjects.

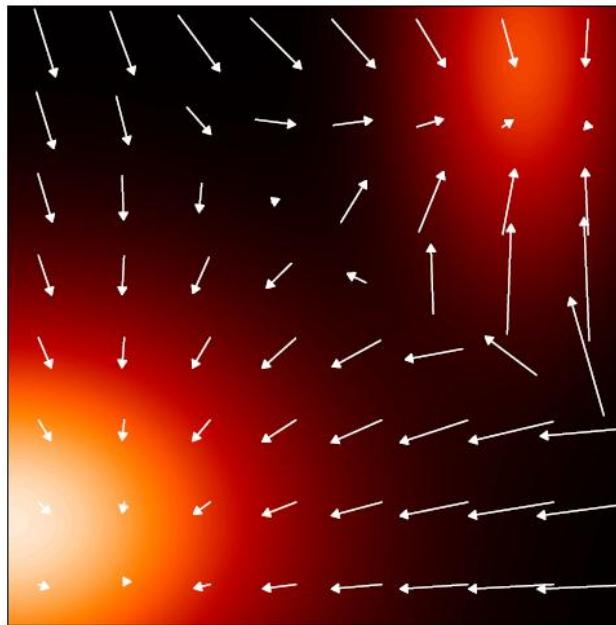
Diffusion Score Matching

The denoising diffusion models discussed so far are closely connected to another family of deep generative models developed largely independently, known as **score-based models**, introduced for the first time by Song and Ermon (2019) [136]. These models are grounded in the technique of **score matching**, which relies on the **score function** (also called the **Stein score**). The score function is defined as the gradient of the log-density with respect to the input data x :

$$s(x) = \nabla_x \log p(x)$$

It's crucial to emphasize that this gradient is taken with respect to the data vector x , not the model parameters. This contrasts with expressions like $\nabla_\theta \log p(x)$, which arise in maximum likelihood estimation and describe how to adjust model parameters θ to increase the likelihood of the observed data. Instead, $\nabla_x \log p(x)$ tells us how to adjust the **input** to increase its likelihood under the model.

Importantly, the score function $s(x)$ is a vector-valued function with the **same dimensionality as x** . Each element $s_i(x) = \partial \log p(x) / \partial x_i$ corresponds to the influence of the i -th element of x on its log-likelihood. If x is an image, then $s(x)$ can also be visualized as image-shaped vector field pointing in the direction of increasing density in data space. The Figure below illustrates this idea in two dimensions for a 2D Gaussian mixture, showing the probability density as an heatmap and the associated score vectors as arrows pointing toward high-probability regions.



Therefore, the score function can be thought of as a vector field over the image space, pointing in the direction where the data density increases most rapidly — that is, toward regions of higher probability. Its magnitude tends to be large when a point is far from the data distribution and small when near it.

This leads to the core intuition behind score-based models: once the model is trained to estimate these score vectors accurately, we can use them to steer sampling trajectories toward regions of high data likelihood, effectively generating samples that resemble real data.

Note: It's also important to notice that the score function **does not require the underlying distribution $p(x)$ to be normalized** because the *partition function* is removed by the gradient operator and so there is considerable flexibility in the choice of model. We can easily see this by parameterizing a score-based model as an energy-based model:

$$s_\theta(x) = \nabla_x \log p_\theta(x) = \nabla_x \log \left(\frac{e^{-E_\theta(x)}}{Z_\theta} \right) = -\nabla_x E_\theta(x) - \underbrace{\nabla_x \log Z_\theta}_{=0} = -\nabla_x E_\theta(x)$$

Since $\nabla_x \log Z_\theta = 0$, the score model is independent of Z_θ . This significantly expands the class of models we can work with, enabling flexible parameterizations without the need for explicitly normalizable distributions.

Score Loss

To train a score-based model, we aim to approximate the **true score function $\nabla_x \log p(x)$** of the distribution $p(x)$ that generated the data using a **parameterized model $s_\theta(x)$** . This model, often implemented as a neural network, is trained to produce a **vector field** that points in the direction of increasing data likelihood.

Therefore, we need to define a loss that aims to match the model predicted score $s_\theta(x)$ with the true score $\nabla_x \log p(x)$. For instance, we can use the **expected squared error**:

$$\mathcal{L}(\theta) = \mathbb{E}_{p(x)} \left[\frac{1}{2} \|s_\theta(x) - \nabla_x \log p(x)\|^2 \right] = \frac{1}{2} \int \|s_\theta(x) - \nabla_x \log p(x)\|^2 p(x) dx$$

This formulation provides significant modeling flexibility: in fact, since [\(as we saw above\)](#) the only requirement is $s_\theta(x)$ to be a vector-valued function with the **same dimensionality as its input**, we can use any neural network architecture for it that respects this constraint. Thus, score-based generative modeling can represent complex distributions simply by learning their score functions.

One problem with the loss function seen above is that we cannot minimize it directly because we do not know the true data score $\nabla_x \log p(x)$. All we have is a finite dataset $\mathcal{D} = \{x_1, \dots, x_N\}$. From these samples, we can define an empirical distribution $p_{\mathcal{D}}(x)$ which places all its mass exactly on the observed data points. Formally, this empirical distribution can be defined as a sum of **Dirac delta functions** centered at each data point:

$$p_{\mathcal{D}}(x) = \frac{1}{N} \sum_{i=1}^N \delta(x - x_i)$$

We recall that the Dirac delta function $\delta(x)$ has two key properties:

- $\delta(x) = 0$ for all $x \neq 0$, meaning it is zero everywhere except at zero.
- $\int \delta(x) dx = 1$, meaning the total “area” under the spike is exactly 1.

In our case so you can imagine this empirical distribution as infinitely tall and narrow spikes that “pick out” values at each dataset point. Although this representation correctly reflects the dataset's discrete nature, it creates a problem: $p_{\mathcal{D}}(x)$ is **not differentiable**, because the delta function itself is not a differentiable function of x . This means we cannot compute $\nabla_x \log p(x)$ needed for the score matching loss. We can address this by introducing a noise model to ‘smear out’ the data points and give a smooth, differentiable representation of the density. In particular, we can **smooth out** the empirical distribution by convolving it with a **noise kernel $q(\tilde{x}|x, \sigma)$** :

$$q_\sigma(\tilde{x}) = \int q(\tilde{x}|x, \sigma) p(x) dx$$

where a common choice for the noise kernel is a Gaussian of the form:

$$q(\tilde{x}|x, \sigma) = \mathcal{N}(\tilde{x}|x, \sigma^2 I)$$

In other words, this operation corresponds to “smooth” each data point of $p(x)$ by adding Gaussian noise with variance σ^2 , i.e. $\tilde{x} = x + \sigma \cdot \epsilon$ where $\epsilon \sim \mathcal{N}(0, I)$.

The key advantage of this smoothing is that the resulting distribution $q_\sigma(\tilde{x})$ is now differentiable with respect to \tilde{x} and so its score $\nabla_{\tilde{x}} \log q_\sigma(\tilde{x})$ is well defined. **We can see $q_\sigma(\tilde{x})$ as a smooth, differentiable approximation of $p(x)$.**

Therefore, instead of minimizing the original loss involving $p(x)$, we can think to minimize the corresponding loss with respect to the **smoothed density**:

$$\mathcal{L}(\theta) = \frac{1}{2} \int \|s_\theta(\tilde{x}) - \nabla_{\tilde{x}} \log q_\sigma(\tilde{x})\|^2 q_\sigma(\tilde{x}) d\tilde{x}$$

By substituting the definition of $q_\sigma(\tilde{x})$, this loss can be rewritten as:

$$\mathcal{L}(\theta) = \frac{1}{2} \int \int \|s_\theta(\tilde{x}) - \nabla_{\tilde{x}} \log q(\tilde{x}|x, \sigma)\|^2 q(\tilde{x}|x, \sigma) p(x) d\tilde{x} dx$$

Replacing the unknown $p(x)$ with the empirical distribution $p_D(x)$, we obtain:

$$\mathcal{L}(\theta) = \frac{1}{2N} \sum_{i=1}^N \int \|s_\theta(\tilde{x}) - \nabla_{\tilde{x}} \log q(\tilde{x}|x_i, \sigma)\|^2 q(\tilde{x}|x_i, \sigma) d\tilde{x}$$

Importantly, in the case we choose the Gaussian kernel, its true score $\nabla_{\tilde{x}} \log q(\tilde{x}|x_i, \sigma)$ has a known closed form:

$$\begin{aligned} \nabla_{\tilde{x}} \log q(\tilde{x}|x_i, \sigma) &= \nabla_{\tilde{x}} \ln \mathcal{N}(\tilde{x}|x_i, \sigma^2 I) = \\ &= \nabla_{\tilde{x}} \left(-\ln(2\pi\sigma^2) - \frac{1}{2\sigma^2} (\tilde{x} - x_i)^2 \right) = \\ &= -\nabla_{\tilde{x}} \frac{1}{2\sigma^2} (\tilde{x} - x_i)^2 = \\ &= -\frac{1}{\sigma^2} (\tilde{x} - x_i) = \\ &= -\frac{1}{\sigma^2} (x_i + \sigma \cdot \epsilon - x_i) = \\ &= -\frac{1}{\sigma} \epsilon \end{aligned}$$

where in the fifth equation we used the definition of $\tilde{x} = x_i + \sigma \cdot \epsilon$, where $\epsilon \sim \mathcal{N}(0, I)$. Thus, the true score becomes:

$$\nabla_{\tilde{x}} \log q(\tilde{x}|x_i, \sigma) = -\frac{1}{\sigma} \epsilon$$

We can now substitute this closed-form expression for the Gaussian score into our differentiable objective and rewrite the loss as an expectation over the noise ϵ :

$$\begin{aligned} \mathcal{L}(\theta) &= \frac{1}{2N} \sum_{i=1}^N \mathbb{E}_{\epsilon \sim \mathcal{N}(0, I)} \left[\left\| s_\theta(x_i + \sigma \cdot \epsilon) + \frac{1}{\sigma} \epsilon \right\|^2 \right] = \\ &= \frac{1}{2N} \sum_{i=1}^N \mathbb{E}_{\epsilon \sim \mathcal{N}(0, I)} \left[\frac{1}{\sigma} \|\epsilon - \tilde{s}_\theta(x_i + \sigma \cdot \epsilon)\|^2 \right] \end{aligned}$$

where in the last line we modified the score model to be of the form $\tilde{s}_\theta(x) = -\sigma s_\theta(x)$ and we used the property of σ being positive (since the standard deviation is always positive by definition!), which helped us to simplify the expression. Of course, while implementing, we can parameterize \tilde{s}_θ using a neural network directly, we do not have to parameterize s_θ first and then rescale it, there is no need for that.

Learning the score model \tilde{s}_θ by optimizing the above loss is called **denoising score matching**. Score matching allows us to learn a generative model in a completely different manner: instead of **matching distributions**, we can perform **matching scores** — effectively turning the problem into a **regression task**.

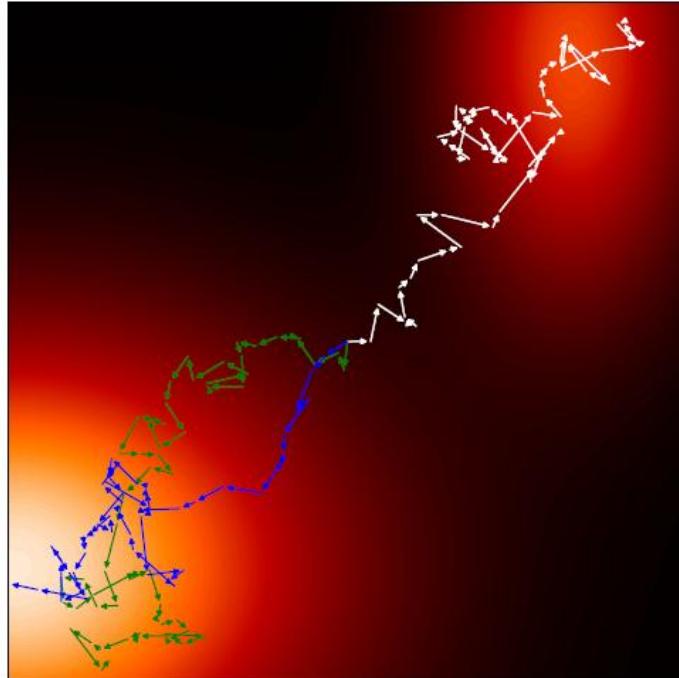
Sampling from a Score-Based Model

Once we have trained a score-based model, at inference time we want to generate new samples from the learned distribution. A natural choice for this is **Langevin dynamics** which relies on the score function, making it well-suited for score-based models. To perform sampling, we begin by drawing an initial value $x_0 \sim \mathcal{N}(0, I)$ and then apply K steps of Langevin dynamics:

$$x_{k+1} = x_k + \eta \nabla_x \log p_\theta(x_k) + \sqrt{2\eta} \epsilon_k, \quad k = 0, \dots, K$$

where η is the step size.

Over the course of these steps, the sample gradually moves toward regions of higher probability, eventually reaching a point x_K where the learned model assigns high density — ideally close to a mode of true data distribution. The Figure below illustrates this with three sample trajectories obtained using Langevin dynamics for the distribution we saw in Figure about the two-mixtures of Gaussians, starting from the center of the plot and converging toward the two high-density regions.



Why Langevin dynamics? Now, one might wonder: why use Langevin dynamics specifically for sampling? After all, once we've trained our score-based model, it seems natural to perform **gradient ascent** on the approximate score function to move toward regions of higher data density. That is, we could iteratively update the sample using:

$$x_{k+1} = x_k + \eta \nabla_x \log p_\theta(x_k)$$

where $\nabla_x \log p_\theta(x_k)$ is approximated by the learned score function $s_\theta(x_k)$. However, if we follow this purely deterministic update rule, we will always converge toward a local maximum of the estimated density and we won't generate diverse samples from the distribution. In fact, no matter where we start, we will tend to end up in the same high-probability region, generating identical or nearly identical samples.

To avoid this and encourage sample diversity, Langevin dynamics adds a small stochastic perturbation to each step, expressed as Gaussian noise ϵ_k scaled by the gradient step size $\sqrt{2\eta}$:

$$x_{k+1} = x_k + \eta \nabla_x \log p_\theta(x_k) + \sqrt{2\eta} \epsilon_k$$

This tiny dose of stochasticity ensures that our trajectory remains random enough to avoid converging at a few fixed points even after many iterations.

Score Matching vs Denoising Diffusion

We can see that the score loss so obtained measures the difference between the neural network score prediction and the noise ϵ . Therefore, this loss function has the same minimum as the form used in the denoising diffusion model, with the score function s_θ playing the same role as the noise prediction ϵ_θ up to a constant scaling $-\frac{1}{\sigma}$. If we look at the objective for denoising diffusion:

$$\mathbb{E}_{x,\epsilon,t} [\gamma_t \|\epsilon - \epsilon_\theta(x_t, t)\|^2]$$

and that of the score matching:

$$\frac{1}{2N} \sum_{i=1}^N \mathbb{E}_{\epsilon \sim \mathcal{N}(0,I)} \left[\frac{1}{\sigma} \|\epsilon - \tilde{s}_\theta(x_i + \sigma \cdot \epsilon)\|^2 \right]$$

we can clearly see they are very similar (especially, if we set $\gamma_t = \frac{1}{\sigma}$)! In fact, we can turn score matching into (almost) denoising diffusion. For doing it we should also need to introduce a variance schedule σ_t^2 and then modify the score model to be "aware" of this schedule, that is exactly what we will do in the next section.

Noise Variance Schedule

We have seen how to learn a score function from data and how to generate new samples using Langevin dynamics. However, we can identify three potential problems with this approach [136]:

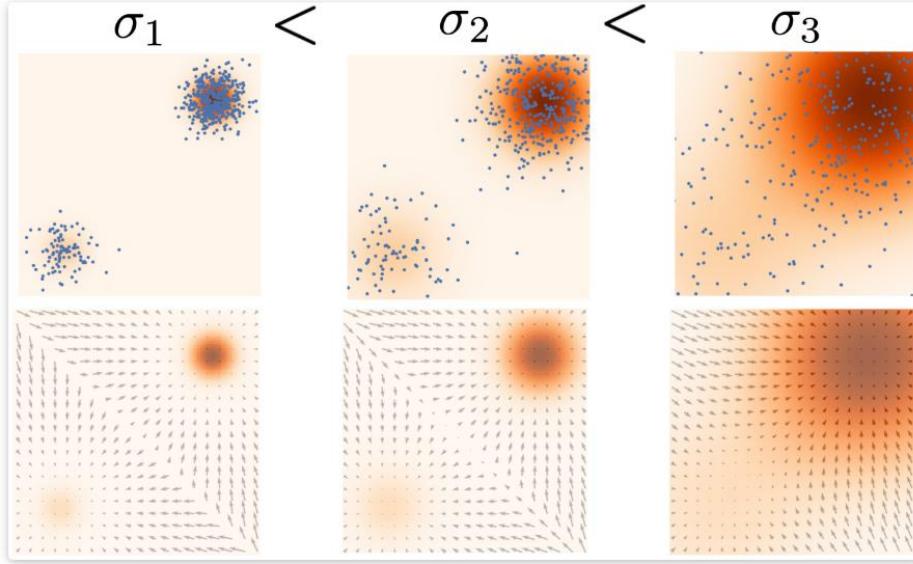
- First, if the data distribution lies on a **manifold of lower dimensionality** than the data space, the probability density $p(x)$ will be zero at points off the manifold and so in these regions the **score function is undefined** since $\log(p(x))$ is undefined.
- Second, in regions of low data density, the estimate of the **score function may be inaccurate** since the loss function is weighted by the density $p(x)$ and so in those regions the model receives very little signal.
- An inaccurate score function can lead to **poor trajectories when using Langevin sampling**: in fact, when sampling with Langevin dynamics, our initial sample is highly likely to fall in low density regions and so having an inaccurate score-based model will derail Langevin dynamics from the very beginning of the procedure, preventing it from generating high quality samples that are representative of the data.

All three problems can be addressed by choosing a sufficiently large value for the noise variance σ^2 used in the training objective. In fact, adding noise in this way "smears out" the data distribution: when the noise magnitude

is sufficiently large, it can perturb data points so that they populate low data density regions, allowing the model to learn more reliable score estimates in these regions.

However, the choice of noise scale presents a trade-off. A **larger variance** can obviously cover more low-density regions for better score estimation, but it will introduce a significant distortion of the original distribution and this itself introduces inaccuracies in the modelling of the score function. A **smaller variance**, in contrast, causes less corruption of the original data distribution, but does not cover the low-density regions as well as we would like.

To balance this trade-off, we can use multiple scales of noise simultaneously, each related to the specific variance considered. Specifically, we introduce a **noise schedule**, i.e. a sequence of increasing variances $\sigma_1^2 < \sigma_2^2 < \dots < \sigma_T^2$ [136] such that σ_1^2 is sufficiently small that the data distribution is accurately represented, whereas σ_T^2 is sufficiently large to address the challenges of undefined or poorly estimated scores in low data density regions.



The score network is then modified to take the **discrete time step t** as an additional input instead of the variance σ_t^2 directly, i.e., $s_\theta(x, t)$. The corresponding variance σ_t^2 is retrieved from the predefined noise schedule associated with each t . The model is trained using a weighted sum of denoising score-matching losses at different noise scales. For a single data point x , this weighted sum takes the form:

$$\sum_{t=1}^T \lambda_t \mathbb{E}_{\epsilon \sim \mathcal{N}(0, I)} \left[\frac{1}{\sigma_t} \|\epsilon - \tilde{s}_\theta(x + \sigma_t \cdot \epsilon, t)\|^2 \right]$$

where λ_t are weighting coefficients often chosen to be $\lambda_t = \sigma_t^2$.

Considering the entire dataset, therefore, the full training objective becomes:

$$\mathcal{L}(\theta) = \frac{1}{2N} \sum_{i=1}^N \sum_{t=1}^T \lambda_t \mathbb{E}_{\epsilon \sim \mathcal{N}(0, I)} \left[\frac{1}{\sigma_t} \|\epsilon - \tilde{s}_\theta(x_i + \sigma_t \cdot \epsilon, t)\|^2 \right]$$

We can notice that this multi-scale noise training procedure closely mirrors that used to train DDPMs. Same as in DDPM, a common architectural choice for the score network is a ***U-Net***, conditioned on the time step t so that it can adjust its predictions according to the corresponding noise level in the schedule.

Once the score network is trained, we can generate samples at inference time using a sampling procedure called **annealed Langevin dynamics**. This consists of applying Langevin dynamics sequentially across decreasing noise levels $t = T, T - 1, \dots, 1$, each with a few steps of Langevin sampling per scale.

It is conceptually very similar to sampling in denoising diffusion models. The key difference is that diffusion models typically perform one "denoising" step per noise level, while annealed Langevin dynamics runs multiple Langevin updates per noise level.

Continuous-Time Diffusion Models

We have seen that diffusion models often benefit from using a large number of noise steps T — typically several thousand. It is therefore natural to ask what happens if we consider the limit of an infinite number of steps, much as we did for infinitely deep neural networks when we introduced *Neural ODE* in the Normalizing Flows chapter. Taking this limit in diffusion models leads to a **continuous-time** formulation.

By generalizing the finite sequence of noise scales to an infinite continuum, we gain two key advantages:

1. **Exact log-likelihood computation** (and not just an approximation), and
2. **Higher quality samples**, due to finer-grained modeling of the data distribution.

To reach this limit, we must ensure that the noise variance σ_t^2 at each step decreases smoothly with the infinitesimal step size. This naturally leads us to represent diffusion models using **stochastic differential equations (SDEs)**, as proposed by Song et al. (2021) [137]. From this perspective, both score-based models and DDPMs can be seen as **discrete approximations** to an underlying continuous-time SDE.

Stochastic Differential Equations

In diffusion models, the forward process gradually corrupts data with random Gaussian noise over time, a process that is inherently **stochastic**, not deterministic. Therefore, to faithfully represent the randomness of the forward corruption process we cannot just rely on ODEs. **SDEs** naturally model systems with uncertainty, incorporating both deterministic and random components in their dynamics.

An SDE describes how a random variable \tilde{x} changes over an infinitesimal time interval dt , as follows:

$$d\tilde{x} = f(\tilde{x}, t)dt + g(t)dw$$

Here $f(\tilde{x}, t) \in \mathbb{R}^d$ is a vector-valued function called the **drift coefficient**, $g(t) \in \mathbb{R}$ is a real-valued function called the **diffusion coefficient**, w denotes a standard Wiener process (also called Brownian motion) and so dw represents an infinitesimal increment of this process.

The drift term governs the deterministic part of the evolution (similar to what we see in ODEs), while the diffusion term is stochastic (due to the standard Wiener process).

For our purposes, we don't need to know much about Wiener processes beyond the fact that they behave like continuous-time Gaussian noise: that is the difference between two time points follows a normal distribution with variance proportional to the time interval:

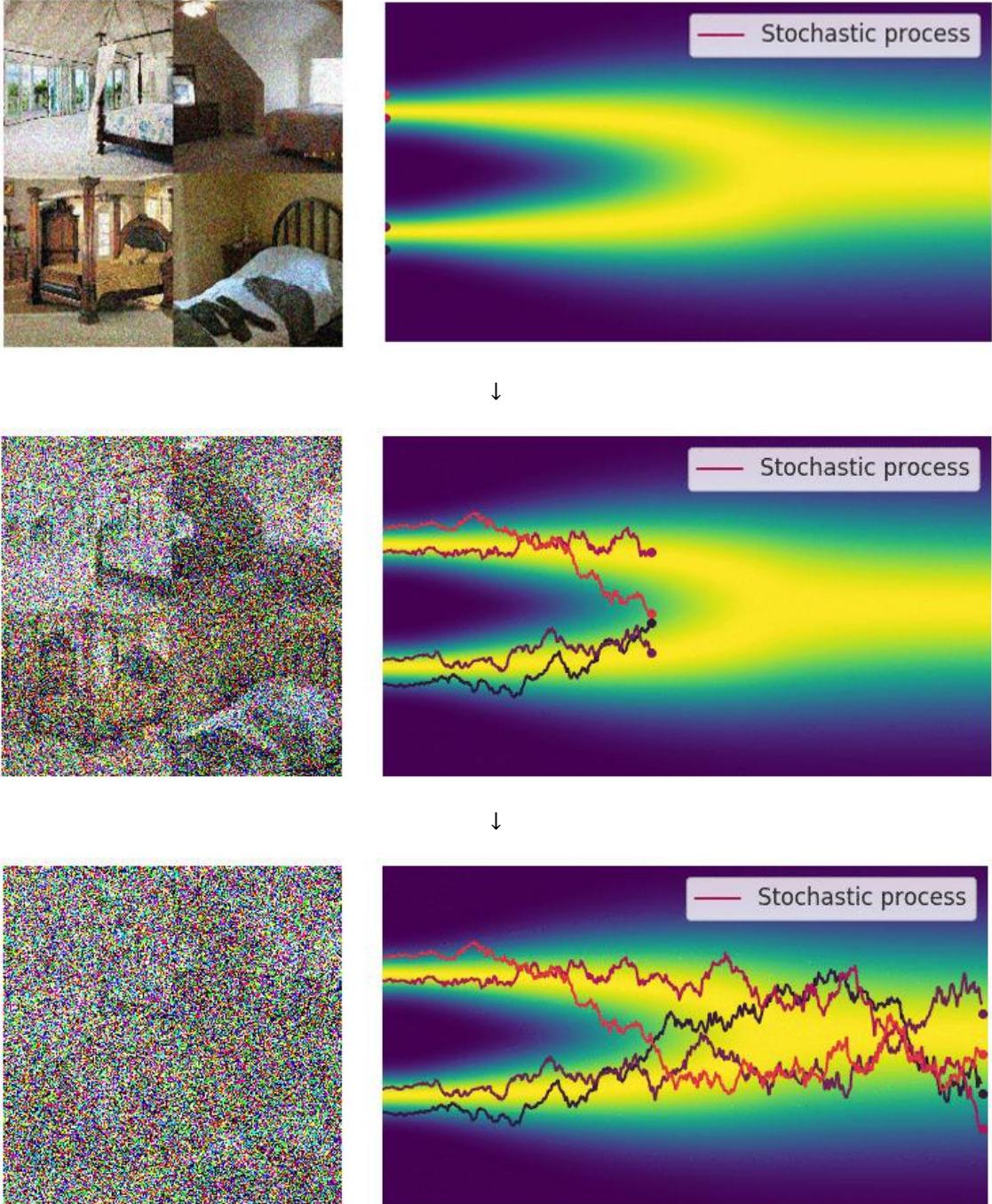
$$w_{t+\Delta} - w_t \sim \mathcal{N}(0, \Delta)$$

In our context, since we deal with **infinitesimal time steps** dt , we have that $\Delta = dt$. This means that dw , an infinitesimal increment of the Wiener process, behaves like a small Gaussian noise with variance dt :

$$dw \sim \mathcal{N}(0, dt)$$

As a result, the **diffusion term** $g(t)dw$ can be interpreted as adding **Gaussian noise** at each infinitesimal time step, with **standard deviation** $g(t)\sqrt{dt}$.

The solution to an SDE is not a single path but a **stochastic process**: a continuous collection of random variables $\{\tilde{x}_t\}_{t \in [0, T]}$ tracing stochastic trajectories as t evolves from 0 to T .



At each time t , we can define a **marginal probability density function** $p_t(\tilde{x})$, which describes the distribution $\tilde{x}(t)$ across different sample paths. This is analogous to what we had in discrete-time score-based models where each time step $t = 1, \dots, T$ corresponds to a different noise level σ_t and we defined:

$$q_{\sigma_t}(\tilde{x}) = \int q(\tilde{x}|x, \sigma_t)p(x)dx$$

as the data distribution perturbed by Gaussian noise of scale σ_t . Now, in the continuous-time setting, we can think of $p_t(\tilde{x})$ as the **continuous analogue** of $q_{\sigma_t}(\tilde{x})$. Instead of injecting noise at a finite number of discrete levels, we now smoothly evolve the noise over a continuous time interval $t \in [0, T]$ via the SDE.

Note: The specific forms of the *drift* and *diffusion* components are hand-designed. There are many valid ways to add noise; hence, multiple SDE formulations exist, each with different trade-offs in terms of sample quality, training stability or computational efficiency. In [137] the authors proposed three SDEs that have been found to work particularly well for generative modeling of images: **Variance Exploding (VE) SDE**, **Variance Preserving (VP) SDE** and **Sub-VP SDE**. Each of these corresponds to a different way of injecting and scaling noise over time.

We won't go into all details here, but for example, in **VE SDE** the drift and the diffusion are chosen to be:

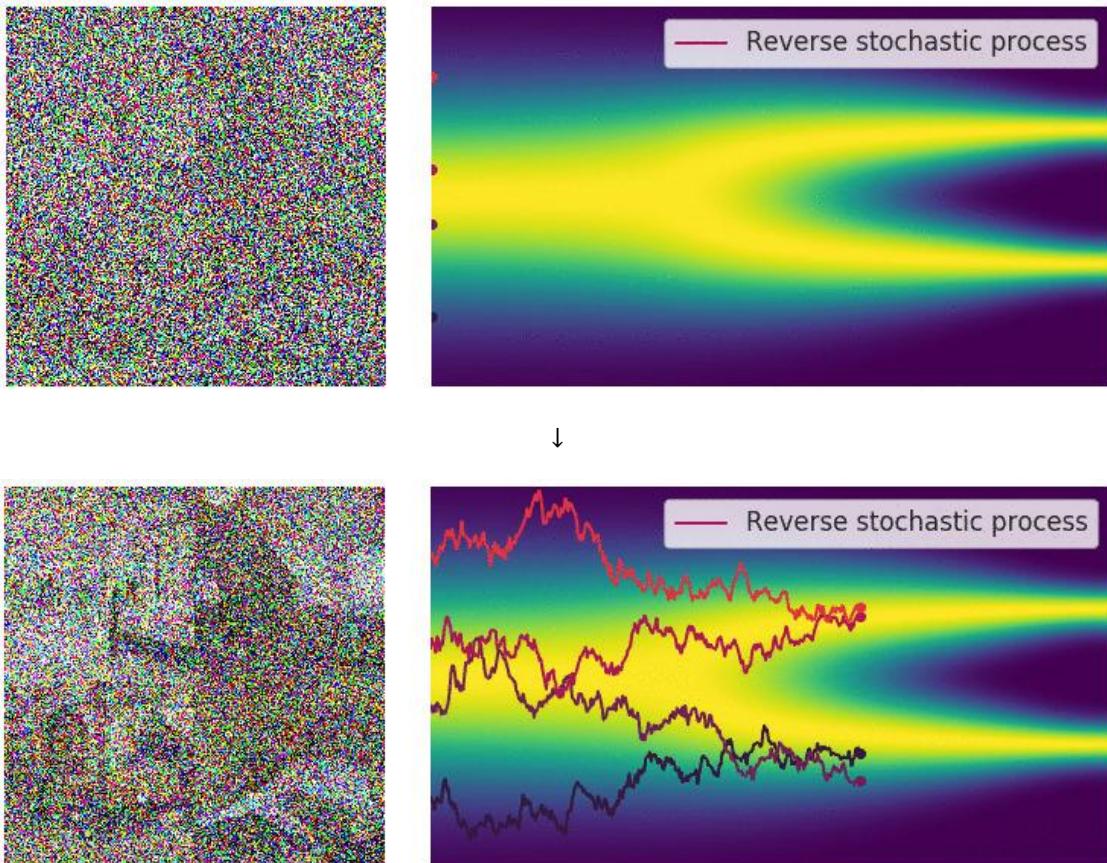
- $f(\tilde{x}, t) = 0$
- $g(t) = \sqrt{\frac{d[\sigma_t^2]}{dt}}$

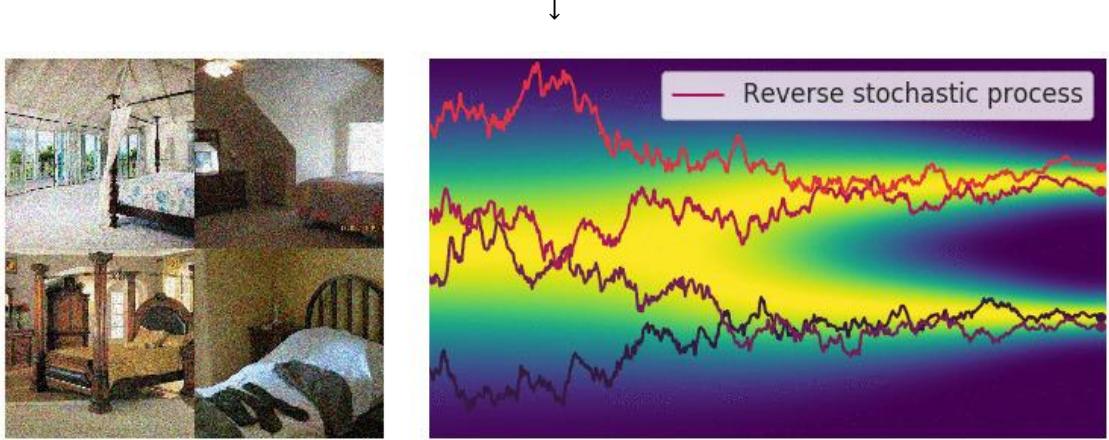
Plugging these into the general SDE formulation gives:

$$d\tilde{x} = \sqrt{\frac{d[\sigma_t^2]}{dt}} dw$$

As you can see, in this case there is **no deterministic drift** — the evolution is driven purely by noise.

Recall that with a finite number of noise scales, we can generate samples by reversing the perturbation process with annealed Langevin dynamics — that is, running Langevin steps across decreasing noise levels. For infinite noise scales, we can analogously reverse the perturbation process for sample generation by solving the **reverse SDE**.

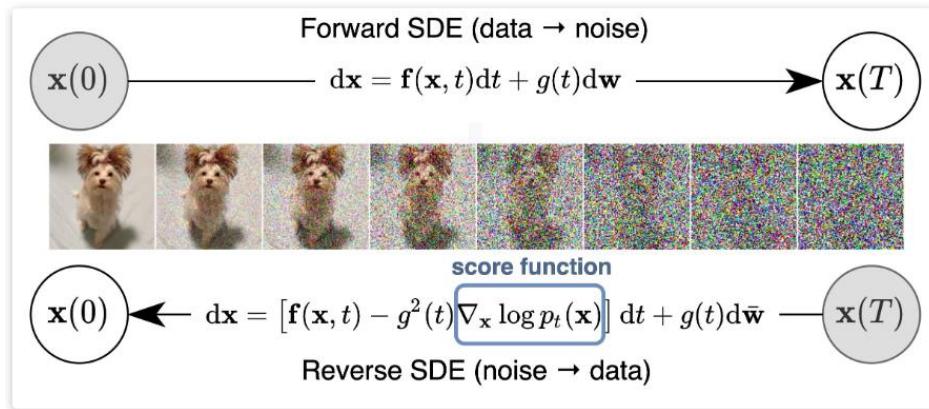




Importantly, any SDE has a corresponding *reverse SDE* [138], whose closed form is given by:

$$d\tilde{x} = [f(\tilde{x}, t) - g^2(t)\nabla_{\tilde{x}} \log p_t(\tilde{x})]dt + g(t)dw$$

Here dt represents a negative infinitesimal time step, since the SDE needs to be solved in reverse from $t = T$ to $t = 0$. In order to compute the reverse SDE, we need to estimate $\nabla_{\tilde{x}} \log p_t(\tilde{x})$, however we can recognize that it is exactly the **score function of $p_t(\tilde{x})$** .



So, what we can do is train a **time-dependent score-based model** $s_\theta(\tilde{x}, t)$, such that $s_\theta(\tilde{x}, t) \approx \nabla_{\tilde{x}} \log p_t(\tilde{x})$. We can think to train $s_\theta(\tilde{x}, t)$ using *denoising score matching*, i.e. the training objective is a continuous weighted combination of the denoising score-matching losses at the different noise scales:

$$\mathbb{E}_{t \sim \mathcal{U}(0, T)} \mathbb{E}_{p_t(\tilde{x})} [\lambda_t \|\nabla_{\tilde{x}} \log p_t(\tilde{x}) - s_\theta(\tilde{x}, t)\|^2]$$

where

- $\mathcal{U}(0, T)$ denotes a uniform distribution over the time interval $[0, T]$
- λ_t is a positive weighting function typically chosen to be $\lambda_t \propto 1/\mathbb{E} [\|\nabla_{x(t)} \log p(x(t)|x(0))\|^2]$ to balance the magnitude of different score matching losses across time.

Once the score-based model $s_\theta(\tilde{x}, t)$ is trained, we can plug it into the expression for the reverse SDE to obtain an **estimated** reverse-time stochastic process.

Now, since this SDE is continuous in time, we cannot solve it exactly in closed form. Instead, to solve an SDE numerically, we need to **discretize** the time variable. The simplest and most commonly used method for solving SDEs numerically is the **Euler–Maruyama method**. It is a stochastic generalization of Euler's method for ODEs, with an added noise term to account for the randomness introduced by the diffusion component.

To apply the Euler–Maruyama method to our estimated **reverse SDE**, we discretize the time interval $[0, T]$ into small, fixed time steps. Since we are solving the reverse process, we move **backward in time**, starting from $t = T$ and stepping toward $t = 0$. At each time step, the update rule is:

$$\begin{aligned}\Delta\tilde{x} &\leftarrow [f(\tilde{x}, t) - g^2(t)s_\theta(\tilde{x}, t)]\Delta t + g(t)\sqrt{|\Delta t|}\epsilon_t \\ \tilde{x} &\leftarrow \tilde{x} + \Delta\tilde{x} \\ t &\leftarrow t + \Delta t\end{aligned}$$

Here $\epsilon_t \sim \mathcal{N}(0, I)$ and $\Delta t < 0$ is a small negative step size, since we're integrating in reverse.

While Euler–Maruyama is simple and effective, more advanced solvers can be used for improved accuracy or stability. Some of these include the [Milstein solver](#) and the [stochastic Runge–Kutta solver](#).

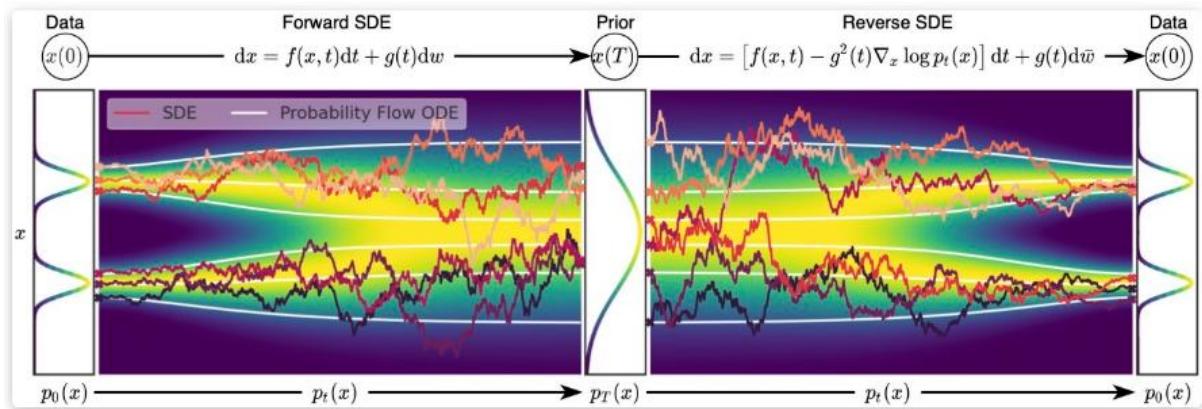
Probability Flow ODE

Despite being capable of generating high-quality samples, samplers based on SDE solvers do **not allow for exact log-likelihood computation** in score-based generative models. This is because the randomness inherent in the stochastic process makes it difficult (or intractable) to compute how a sample density evolves through the stochastic process.

To address this, always in [137], Song et al. showed that for every diffusion process governed by an SDE, there exists a corresponding deterministic ODE — called the **Probability Flow ODE** — whose trajectories have the same marginal probability densities $\{p_t(\tilde{x})\}_{t \in [0, T]}$ as the SDE. This means it is possible to convert any SDE into an ODE without changing its marginal distributions. The Probability Flow ODE of an SDE is given by:

$$\frac{d\tilde{x}}{dt} = f(\tilde{x}, t) - \frac{1}{2}g^2(t)\nabla_{\tilde{x}} \log p_t(\tilde{x})$$

The Figure below illustrates the sample trajectories generated by both the SDE and its corresponding Probability Flow ODE. While the SDE trajectories are inherently noisy due to the stochastic nature of the process, the **ODE trajectories** are noticeably **smoother** because they follow a deterministic path. Despite this difference, **both processes transform the data distribution into the same prior distribution (and vice versa)** and that is because they share the same set of **marginal distributions**.



This ODE-based formulation comes with several powerful benefits:

- When $\nabla_{\tilde{x}} \log p_t(\tilde{x})$ of the formula above is replaced by its learned approximation $s_\theta(\tilde{x}, t)$, the **Probability Flow ODE** becomes a **special case of a Neural ODE**. More specifically, it falls under the class of **continuous normalizing flows**. This is because the Probability Flow ODE defines a continuous, **fully**

invertible transformation that maps the data distribution $p_0(\tilde{x})$ to the prior noise distribution $p_T(\tilde{x})$, while preserving the same marginal distributions as the original SDE at every time step.

- As such, the Probability Flow ODE inherits all properties of Neural ODEs or continuous normalizing flows, including **exact log-likelihood computation**. Specifically, we can leverage the instantaneous *change-of-variable* formula to compute the unknown data density p_0 from the known prior density p_T with **numerical ODE solvers**.
- Because it's an ODE, we can make use of **adaptive step-size ODE solvers** which can drastically reduce the number of function evaluations, enabling to generate samples faster ([a smaller number of steps](#)) and with better quality.

Flow Matching

Content coming soon!

I'll update this section as soon as I have some time. In the meantime, if you find this work useful, please consider supporting it by giving the repository a .

Bibliography

- [1] Y. Bengio, Y. LeCun e others, «Scaling learning algorithms towards AI,» *Large-scale kernel machines*, vol. 34, n. 5, pp. 1-41, 2007.
- [2] A. Joulin, M. Cissé, D. Grangier, H. Jégou e others, «Efficient softmax approximation for GPUs,» in *International conference on machine learning*, PMLR, 2017, pp. 1302-1310.
- [3] Y. LeCun, L. Bottou, G. B. Orr e K.-R. Müller, «Efficient backprop,» in *Neural networks: Tricks of the trade*, Springer, 2002, pp. 9-50.
- [4] X. Glorot e Y. Bengio, «Understanding the difficulty of training deep feedforward neural networks,» in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, JMLR Workshop and Conference Proceedings, 2010, pp. 249-256.
- [5] K. He, X. Zhang, S. Ren e J. Sun, «Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,» in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026-1034.
- [6] B. T. Polyak, «Some methods of speeding up the convergence of iteration methods,» *Ussr computational mathematics and mathematical physics*, vol. 4, n. 5, pp. 1-17, 1964.
- [7] I. Sutskever, J. Martens, G. Dahl e G. Hinton, «On the importance of initialization and momentum in deep learning,» in *International conference on machine learning*, pmlr, 2013, pp. 1139-1147.
- [8] M. Riedmiller e H. Braun, «A direct adaptive method for faster backpropagation learning: The RPROP algorithm,» in *IEEE international conference on neural networks*, IEEE, 1993, pp. 586-591.
- [9] J. Duchi, E. Hazan e Y. Singer, «Adaptive subgradient methods for online learning and stochastic optimization.,» *Journal of machine learning research*, vol. 12, n. 7, 2011.
- [10] M. D. Zeiler, «Adadelta: an adaptive learning rate method,» *arXiv preprint arXiv:1212.5701*, 2012.
- [11] D. P. Kingma e J. Ba, «Adam: A method for stochastic optimization.,» in *International Conference on Learning Representations*, 2015.
- [12] I. Loshchilov e F. Hutter, «Decoupled weight decay regularization,» *arXiv preprint arXiv:1711.05101*, 2017.
- [13] L. Chen, B. Liu, K. Liang e Q. Liu, «Lion secretly solves constrained optimization: As lyapunov predicts,» *arXiv preprint arXiv:2310.05898*, 2023.
- [14] J. Martens e others, «Deep learning via hessian-free optimization.,» in *Icml*, vol. 27, 2010, pp. 735-742.
- [15] F. Bach, «Batch normalization: Accelerating deep network training by reducing internal covariate shift,» in *Proc 32nd Int Conf Mach Learn*, vol. 37, 2015, p. 448.
- [16] S. Santurkar, D. Tsipras, A. Ilyas e A. Madry, «How does batch normalization help optimization?,» *Advances in neural information processing systems*, vol. 31, 2018.
- [17] J. L. Ba, J. R. Kiros e G. E. Hinton, «Layer normalization,» *arXiv preprint arXiv:1607.06450*, 2016.

- [18] D. Ulyanov, A. Vedaldi e V. Lempitsky, «Instance normalization: The missing ingredient for fast stylization,» *arXiv preprint arXiv:1607.08022*, 2016.
- [19] Y. Wu e K. He, «Group normalization,» in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 3-19.
- [20] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever e R. Salakhutdinov, «Dropout: a simple way to prevent neural networks from overfitting,» *The journal of machine learning research*, vol. 15, n. 1, pp. 1929-1958, 2014.
- [21] Y. LeCun, L. Bottou, Y. Bengio e P. Haffner, «Gradient-based learning applied to document recognition,» *Proceedings of the IEEE*, vol. 86, n. 11, pp. 2278-2324, 2002.
- [22] M. D. Zeiler e R. Fergus, «Visualizing and understanding convolutional networks,» in *European conference on computer vision*, Springer, 2014, pp. 818-833.
- [23] Y. LeCun, L. Bottou, Y. Bengio e P. Haffner, «Gradient-based learning applied to document recognition,» *Proceedings of the IEEE*, vol. 86, n. 11, pp. 2278-2324, 1998.
- [24] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li e L. Fei-Fei, «Imagenet: A large-scale hierarchical image database,» in *2009 IEEE conference on computer vision and pattern recognition*, ieee, 2009, pp. 248-255.
- [25] A. Krizhevsky, I. Sutskever e G. E. Hinton, «Imagenet classification with deep convolutional neural networks,» *Advances in neural information processing systems*, vol. 25, 2012.
- [26] K. Simonyan e A. Zisserman, «Very deep convolutional networks for large-scale image recognition,» *arXiv preprint arXiv:1409.1556*, 2014.
- [27] K. He, X. Zhang, S. Ren e J. Sun, «Deep residual learning for image recognition,» in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770-778.
- [28] K. He, X. Zhang, S. Ren e J. Sun, «Identity mappings in deep residual networks,» in *European conference on computer vision*, Springer, 2016, pp. 630-645.
- [29] H. Li, Z. Xu, G. Taylor, C. Studer e T. Goldstein, «Visualizing the loss landscape of neural nets,» *Advances in neural information processing systems*, vol. 31, 2018.
- [30] R. Girshick, J. Donahue, T. Darrell e J. Malik, «Rich feature hierarchies for accurate object detection and semantic segmentation,» in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580-587.
- [31] R. Girshick, «Fast r-cnn,» in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440-1448.
- [32] S. Ren, K. He, R. Girshick e J. Sun, «Faster r-cnn: Towards real-time object detection with region proposal networks,» *Advances in neural information processing systems*, vol. 28, 2015.
- [33] J. Redmon, S. Divvala, R. Girshick e A. Farhadi, «You only look once: Unified, real-time object detection,» in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779-788.
- [34] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu e A. C. Berg, «Ssd: Single shot multibox detector,» in *European conference on computer vision*, Springer, 2016, pp. 21-37.

- [35] M. Tan, R. Pang e Q. V. Le, «Efficientdet: Scalable and efficient object detection,» in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 10781-10790.
- [36] J. Long, E. Shelhamer e T. Darrell, «Fully convolutional networks for semantic segmentation,» in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431-3440.
- [37] O. Ronneberger, P. Fischer e T. Brox, «U-net: Convolutional networks for biomedical image segmentation,» in *International Conference on Medical image computing and computer-assisted intervention*, Springer, 2015, pp. 234-241.
- [38] T. Kumar, R. Brennan, A. Mileo e M. Bendechache, «Image data augmentation approaches: A comprehensive survey and future directions,» *IEEE Access*, 2024.
- [39] J. L. Elman, «Finding structure in time,» *Cognitive science*, vol. 14, n. 2, pp. 179-211, 1990.
- [40] T. Mikolov, K. Chen, G. Corrado e J. Dean, «Efficient estimation of word representations in vector space,» *arXiv preprint arXiv:1301.3781*, 2013.
- [41] R. Sennrich, B. Haddow e A. Birch, «Neural machine translation of rare words with subword units,» *arXiv preprint arXiv:1508.07909*, 2015.
- [42] S. Hochreiter e J. Schmidhuber, «Long short-term memory,» *Neural computation*, vol. 9, n. 8, pp. 1735-1780, 1997.
- [43] K. Cho, B. Van Merriënboer, D. Bahdanau e Y. Bengio, «On the properties of neural machine translation: Encoder-decoder approaches,» *arXiv preprint arXiv:1409.1259*, 2014.
- [44] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk e Y. Bengio, «Learning phrase representations using RNN encoder-decoder for statistical machine translation,» *arXiv preprint arXiv:1406.1078*, 2014.
- [45] D. Bahdanau, «Neural machine translation by jointly learning to align and translate,» *arXiv preprint arXiv:1409.0473*, 2014.
- [46] M. Schuster e K. K. Paliwal, «Bidirectional recurrent neural networks,» *IEEE transactions on Signal Processing*, vol. 45, n. 11, pp. 2673-2681, 1997.
- [47] P. Zhou, W. Shi, J. Tian, Z. Qi, B. Li, H. Hao e B. Xu, «Attention-based bidirectional long short-term memory networks for relation classification,» in *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 2: Short papers)*, 2016, pp. 207-212.
- [48] A. Gu e T. Dao, «Mamba: Linear-time sequence modeling with selective state spaces,» *arXiv preprint arXiv:2312.00752*, 2023.
- [49] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, {. Kaiser e I. Polosukhin, «Attention is all you need,» *Advances in neural information processing systems*, vol. 30, 2017.
- [50] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever e others, «Improving language understanding by generative pre-training,» 2018.
- [51] J. Devlin, M.-W. Chang, K. Lee e K. Toutanova, «Bert: Pre-training of deep bidirectional transformers for language understanding,» in *Proceedings of the 2019 conference of the North American chapter of the*

association for computational linguistics: human language technologies, volume 1 (long and short papers), 2019, pp. 4171-4186.

- [52] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever e others, «Language models are unsupervised multitask learners,» *OpenAI blog*, vol. 1, n. 8, p. 9, 2019.
- [53] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell e others, «Language models are few-shot learners,» *Advances in neural information processing systems*, vol. 33, pp. 1877-1901, 2020.
- [54] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey e others, «Google's neural machine translation system: Bridging the gap between human and machine translation,» *arXiv preprint arXiv:1609.08144*, 2016.
- [55] A. Holtzman, J. Buys, L. Du, M. Forbes e Y. Choi, «The curious case of neural text degeneration,» *arXiv preprint arXiv:1904.09751*, 2019.
- [56] A. Fan, M. Lewis e Y. Dauphin, «Hierarchical neural story generation,» *arXiv preprint arXiv:1805.04833*, 2018.
- [57] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray e others, «Training language models to follow instructions with human feedback,» *Advances in neural information processing systems*, vol. 35, pp. 27730-27744, 2022.
- [58] W. Fedus, B. Zoph e N. Shazeer, «Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity,» *Journal of Machine Learning Research*, vol. 23, n. 120, pp. 1-39, 2022.
- [59] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. K{"u}ttler, M. Lewis, W.-t. Yih, T. Rockt{"a}schen e others, «Retrieval-augmented generation for knowledge-intensive nlp tasks,» *Advances in neural information processing systems*, vol. 33, pp. 9459-9474, 2020.
- [60] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou e others, «Chain-of-thought prompting elicits reasoning in large language models,» *Advances in neural information processing systems*, vol. 35, pp. 24824-24837, 2022.
- [61] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly e others, «An image is worth 16x16 words: Transformers for image recognition at scale,» *arXiv preprint arXiv:2010.11929*, 2020.
- [62] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark e others, «Learning transferable visual models from natural language supervision,» in *International conference on machine learning*, PMLR, 2021, pp. 8748-8763.
- [63] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le e R. Salakhutdinov, «Transformer-xl: Attentive language models beyond a fixed-length context,» *arXiv preprint arXiv:1901.02860*, 2019.
- [64] N. Shazeer, «Fast transformer decoding: One write-head is all you need,» *arXiv preprint arXiv:1911.02150*, 2019.
- [65] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann e others, «Palm: Scaling language modeling with pathways,» *Journal of Machine Learning Research*, vol. 24, n. 240, pp. 1-113, 2023.

- [66] J. Ainslie, J. Lee-Thorp, M. De Jong, Y. Zemlyanskiy, F. Lebr{\o}ne S. Sangha, «Gqa: Training generalized multi-query transformer models from multi-head checkpoints,» *arXiv preprint arXiv:2305.13245*, 2023.
- [67] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li e P. J. Liu, «Exploring the limits of transfer learning with a unified text-to-text transformer,» *Journal of machine learning research*, vol. 21, n. 140, pp. 1-67, 2020.
- [68] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale e others, «Llama 2: Open foundation and fine-tuned chat models,» *arXiv preprint arXiv:2307.09288*, 2023.
- [69] E. Almazrouei, H. Alobeidli, A. Alshamsi, A. Cappelli, R. Cojocaru, M. Debbah, {\o}. Goffinet, D. Hesslow, J. Launay, Q. Malartic e others, «The falcon series of open language models,» *arXiv preprint arXiv:2311.16867*, 2023.
- [70] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock e others, «Mistral 7B,» *arXiv preprint arXiv:2310.06825*, vol. 3, 2023.
- [71] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand e others, «Mixtral of experts,» *arXiv preprint arXiv:2401.04088*, 2024.
- [72] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer e V. Stoyanov, «Roberta: A robustly optimized bert pretraining approach,» *arXiv preprint arXiv:1907.11692*, 2019.
- [73] P. Shaw, J. Uszkoreit e A. Vaswani, «Self-attention with relative position representations,» *arXiv preprint arXiv:1803.02155*, 2018.
- [74] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo e Y. Liu, «Roformer: Enhanced transformer with rotary position embedding,» *Neurocomputing*, vol. 568, n. Elsevier, p. 127063, 2024.
- [75] A. Liu, B. Feng, B. Wang, B. Wang, B. Liu, C. Zhao, C. Dengr, C. Ruan, D. Dai, D. Guo e others, «Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model,» *arXiv preprint arXiv:2405.04434*, 2024.
- [76] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan e others, «Deepseek-v3 technical report,» *arXiv preprint arXiv:2412.19437*, 2024.
- [77] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi e others, «Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning,» *arXiv preprint arXiv:2501.12948*, 2025.
- [78] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner e G. Monfardini, «The graph neural network model,» *IEEE transactions on neural networks*, vol. 20, n. 1, pp. 61-80, 2008.
- [79] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals e G. E. Dahl, «Neural message passing for quantum chemistry,» in *International conference on machine learning*, Pmlr, 2017, pp. 1263-1272.
- [80] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner e others, «Relational inductive biases, deep learning, and graph networks,» *arXiv preprint arXiv:1806.01261*, 2018.

- [81] M. Defferrard, X. Bresson e P. Vandergheynst, «Convolutional neural networks on graphs with fast localized spectral filtering,» *Advances in neural information processing systems*, vol. 29, 2016.
- [82] T. Kipf, «Semi-Supervised Classification with Graph Convolutional Networks,» *arXiv preprint arXiv:1609.02907*, 2016.
- [83] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, Y. Bengio e others, «Graph attention networks,» *stat*, vol. 1050, n. 20, pp. 10-48550, 2017.
- [84] W. Hamilton, Z. Ying e J. Leskovec, «Inductive representation learning on large graphs,» *Advances in neural information processing systems*, vol. 30, 2017.
- [85] K. Xu, W. Hu, J. Leskovec e S. Jegelka, «How powerful are graph neural networks?,» *arXiv preprint arXiv:1810.00826*, 2018.
- [86] G. E. Hinton e R. R. Salakhutdinov, «Reducing the dimensionality of data with neural networks,» *science*, vol. 313, n. 5786, pp. 504-507, 2006.
- [87] A. Ng e others, «Sparse autoencoder,» *CS294A Lecture notes*, vol. 72, pp. 1-19, 2011.
- [88] P. Vincent, H. Larochelle, Y. Bengio e P.-A. Manzagol, «Extracting and composing robust features with denoising autoencoders,» in *Proceedings of the 25th international conference on Machine learning*, 2008, pp. 1096-1103.
- [89] Y. Bengio, P. Lamblin, D. Popovici e H. Larochelle, «Greedy layer-wise training of deep networks,» *Advances in neural information processing systems*, vol. 19, 2006.
- [90] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville e Y. Bengio, «Generative adversarial nets,» *Advances in neural information processing systems*, vol. 27, 2014.
- [91] A. Radford, L. Metz e S. Chintala, «Unsupervised representation learning with deep convolutional generative adversarial networks,» *arXiv preprint arXiv:1511.06434*, 2015.
- [92] M. Mirza e S. Osindero, «Conditional generative adversarial nets,» *arXiv preprint arXiv:1411.1784*, 2014.
- [93] J.-Y. Zhu, T. Park, P. Isola e A. A. Efros, «Unpaired image-to-image translation using cycle-consistent adversarial networks,» in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2223-2232.
- [94] T. Karras, T. Aila, S. Laine e J. Lehtinen, «Progressive growing of gans for improved quality, stability, and variation,» *arXiv preprint arXiv:1710.10196*, 2017.
- [95] T. Karras, S. Laine e T. Aila, «A style-based generator architecture for generative adversarial networks,» in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 4401-4410.
- [96] M. Arjovsky, S. Chintala e L. Bottou, «Wasserstein GAN,» *arXiv:1701.07875*, vol. 685, 2017.
- [97] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin e A. C. Courville, «Improved training of wasserstein gans,» *Advances in neural information processing systems*, vol. 30, 2017.
- [98] T. Miyato, T. Kataoka, M. Koyama e Y. Yoshida, «Spectral normalization for generative adversarial networks,» *arXiv preprint arXiv:1802.05957*, 2018.

- [99] J. H. Lim e J. C. Ye, «Geometric gan,» *arXiv preprint arXiv:1705.02894*, 2017.
- [100] D. P. Kingma e M. Welling, «Auto-encoding variational bayes,» *arXiv preprint arXiv:1312.6114*, 2013.
- [101] D. P. Kingma, M. Welling e others, «An introduction to variational autoencoders,» *Foundations and Trends in Machine Learning*, vol. 12, n. 4, pp. 307-392, 2019.
- [102] I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. Botvinick, S. Mohamed e A. Lerchner, «beta-vae: Learning basic visual concepts with a constrained variational framework,» in *International conference on learning representations*, 2017.
- [103] D. Rezende e S. Mohamed, «Variational inference with normalizing flows,» in *International conference on machine learning*, PMLR, 2015, pp. 1530-1538.
- [104] A. Van Den Oord, O. Vinyals e others, «Neural discrete representation learning,» *Advances in neural information processing systems*, vol. 30, 2017.
- [105] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen e I. Sutskever, «Zero-shot text-to-image generation,» in *International conference on machine learning*, Pmlr, 2021, pp. 8821-8831.
- [106] I. Kobyzev, S. J. Prince e M. A. Brubaker, «Normalizing flows: An introduction and review of current methods,» *IEEE transactions on pattern analysis and machine intelligence*, vol. 43, n. 11, pp. 3964-3979, 2020.
- [107] G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed e B. Lakshminarayanan, «Normalizing flows for probabilistic modeling and inference,» *Journal of Machine Learning Research*, vol. 22, n. 57, pp. 1-64, 2021.
- [108] L. Dinh, J. Sohl-Dickstein e S. Bengio, «Density estimation using real nvp,» *arXiv preprint arXiv:1605.08803*, 2016.
- [109] G. Papamakarios, T. Pavlakou e I. Murray, «Masked autoregressive flow for density estimation,» *Advances in neural information processing systems*, vol. 30, 2017.
- [110] D. P. Kingma, T. Salimans, R. Jozefowicz, X. Chen, I. Sutskever e M. Welling, «Improved variational inference with inverse autoregressive flow,» *Advances in neural information processing systems*, vol. 29, 2016.
- [111] R. T. Chen, Y. Rubanova, J. Bettencourt e D. K. Duvenaud, «Neural ordinary differential equations,» *Advances in neural information processing systems*, vol. 31, 2018.
- [112] W. Grathwohl, R. T. Chen, J. Bettencourt, I. Sutskever e D. Duvenaud, «Ffjord: Free-form continuous dynamics for scalable reversible generative models,» *arXiv preprint arXiv:1810.01367*, 2018.
- [113] Y. LeCun, S. Chopra, R. Hadsell, M. Ranzato, F. Huang e others, «A tutorial on energy-based learning,» *Predicting structured data*, vol. 1, n. 0, 2006.
- [114] A. Dawid e Y. LeCun, «Introduction to latent variable energy-based models: a path toward autonomous machine intelligence,» *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2024, n. 10, p. 104011, 2024.
- [115] M. Assran, Q. Duval, I. Misra, P. Bojanowski, P. Vincent, M. Rabbat, Y. LeCun e N. Ballas, «Self-supervised learning from images with a joint-embedding predictive architecture,» in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 15619-15629.

- [116] G. E. Hinton, «Training products of experts by minimizing contrastive divergence,» *Neural computation*, vol. 14, n. 8, pp. 1771-1800, 2002.
- [117] T. Tieleman, «Training restricted Boltzmann machines using approximations to the likelihood gradient,» in *Proceedings of the 25th international conference on Machine learning*, 2008, pp. 1064-1071.
- [118] J. Bromley, I. Guyon, Y. LeCun, E. S{\\"a}ckinger e R. Shah, «Signature verification using a " siamese" time delay neural network,» *Advances in neural information processing systems*, vol. 6, 1993.
- [119] S. Chopra, R. Hadsell e Y. LeCun, «Learning a similarity metric discriminatively, with application to face verification,» in *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, vol. 1, IEEE, 2005, pp. 539-546.
- [120] Y. Taigman, M. Yang, M. Ranzato e L. Wolf, «Deepface: Closing the gap to human-level performance in face verification,» in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 1701-1708.
- [121] I. Misra e L. v. d. Maaten, «Self-supervised learning of pretext-invariant representations,» in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 6707-6717.
- [122] K. He, H. Fan, Y. Wu, S. Xie e R. Girshick, «Momentum contrast for unsupervised visual representation learning,» in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 9729-9738.
- [123] T. Chen, S. Kornblith, M. Norouzi e G. Hinton, «A simple framework for contrastive learning of visual representations,» in *International conference on machine learning*, PMLR, 2020, pp. 1597-1607.
- [124] G. W. Taylor, I. Spiro, C. Bregler e R. Fergus, «Learning invariance through imitation,» in *CVPR 2011*, IEEE, 2011, pp. 2729-2736.
- [125] X. Chen e K. He, «Exploring simple siamese representation learning,» in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2021, pp. 15750-15758.
- [126] J. Ho, A. Jain e P. Abbeel, «Denoising diffusion probabilistic models,» *Advances in neural information processing systems*, vol. 33, pp. 6840-6851, 2020.
- [127] J. Sohl-Dickstein, E. Weiss, N. Maheswaranathan e S. Ganguli, «Deep unsupervised learning using nonequilibrium thermodynamics,» in *International conference on machine learning*, pmlr, 2015, pp. 2256-2265.
- [128] A. Q. Nichol e P. Dhariwal, «Improved denoising diffusion probabilistic models,» in *International conference on machine learning*, PMLR, 2021, pp. 8162-8171.
- [129] D. Kingma e R. Gao, «Understanding diffusion objectives as the elbo with simple data augmentation,» in *Advances in Neural Information Processing Systems*, vol. 36, 2023, pp. 65484-65516.
- [130] J. Song, C. Meng e S. Ermon, «Denoising diffusion implicit models,» *arXiv preprint arXiv:2010.02502*, 2020.
- [131] P. Dhariwal e A. Nichol, «Diffusion models beat gans on image synthesis,» *Advances in neural information processing systems*, vol. 34, pp. 8780-8794, 2021.
- [132] J. Ho e T. Salimans, «Classifier-free diffusion guidance,» *arXiv preprint arXiv:2207.12598*, 2022.

- [133] A. Nichol, P. Dhariwal, A. Ramesh, P. Shyam, P. Mishkin, B. McGrew, I. Sutskever e M. Chen, «Glide: Towards photorealistic image generation and editing with text-guided diffusion models,» *arXiv preprint arXiv:2112.10741*, 2021.
- [134] R. Rombach, A. Blattmann, D. Lorenz, P. Esser e B. Ommer, «High-resolution image synthesis with latent diffusion models,» in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 10684-10695.
- [135] M. Cherti, R. Beaumont, R. Wightman, M. Wortsman, G. Ilharco, C. Gordon, C. Schuhmann, L. Schmidt e J. Jitsev, «Reproducible scaling laws for contrastive language-image learning,» in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2023, pp. 2818-2829.
- [136] Y. Song e S. Ermon, «Generative modeling by estimating gradients of the data distribution,» *Advances in neural information processing systems*, vol. 32, 2019.
- [137] Y. Song, J. Sohl-Dickstein, D. P. Kingma, A. Kumar, S. Ermon e B. Poole, «Score-based generative modeling through stochastic differential equations,» *International Conference on Learning Representations*, 2021.
- [138] B. D. Anderson, «Reverse-time diffusion equation models,» *Stochastic Processes and their Applications*, vol. 12, n. 3, pp. 313-326, 1982.