How Does It Work?

Paul Lehaut

January 7, 2026

Contents

1 Introduction

Schematically : DL $\subset$ ML $\subset$ AI.

- AI : Set of theories/algorithms allowing computers task that may have required human intelligence.
- ML : Focuses on algorithms that improve automatically through experience. It studies function approximation where parameters are estimated from data rather than hard-coded.
- DL : A subset of ML where the approximating functions are artificial neural networks with multiple layers, allowing for hierarchical representation learning.

1.1 Machine Learning

Let's precise, first ML :

- The goal is to approximate a relation $\mathbb{P}$ between inputs $\mathcal{X}$ and outputs $\mathcal{Y}$ by using a function of our data : $h \in \mathcal{H} : \mathcal{X} \longrightarrow \mathcal{Y}$, where $\mathcal{H}$ is the set of available functions (the hypothesis).
- h will need to minimize what is called the **True Risk** :

$$R(h) = \mathbb{E}_{(x,y)\sim\mathbb{P}}(l(h(x), y)) = \int_{\mathcal{X}\times\mathcal{Y}} l(h(x), y) d\mathbb{P}(x, y)$$

  where l is a convex loss function.
- The main problem is to compute $R(h)$ while $\mathbb{P}$ is unknown.

To tackle that problem, we use a Monte Carlo estimate based on our data $S = ((x_i, y_i))_{1\leq i\leq n} \sim \mathbb{P}$ iid, we get the **Empirical Risk**:

$$\hat{R}_S(h) = \frac{1}{n}\sum_{i=1}^{n} l(h(x_i), y_i)$$

we then choose $\hat{h}_S$ such that : $\hat{h}_S \in \text{argmin}_h \hat{R}_S(h)$.

To avoid overfitting ($\hat{R}_S(\hat{h}_S) \approx 0$ while $R(\hat{h}_S) >> 0$) we admit that Generalization Gap $\approx \sqrt{\frac{|\mathcal{H}|}{n}}$.

We can then decompose the error of our learned predictor $\hat{h}_S$ into two orthogonal components relative to the Bayes optimal predictor h^* (the theoretical limit of performance):

$$h^*(x) = \text{argmin}_y \mathbb{E}_{Y\sim\mathbb{P}(\cdot|\ X=x)}(l(y, Y)).$$

The **Excess Risk** is:

$$R(\hat{h}_S) - R(h^*) = (R(\hat{h}_S) - \inf_{h\in\mathcal{H}} R(h)) + (\inf_{h\in\mathcal{H}} R(h) - R(h^*))$$

- **Bias** : $\inf_{h\in\mathcal{H}} R(h) - R(h^*)$ is the capacity of our hypothesis to approximate the truth.
- **Variance** : $R(\hat{h}_S) - \inf_{h\in\mathcal{H}} R(h)$ how far is our chosen $\hat{h}_S$ from the best $h \in \mathcal{H}$ (due to finite sampling).

This brings the following problem : as we increase the complexity of our model $\mathcal{H}$, we reduce the bias but we increase the variance.

To deal with this trade-off, we may try to minimize $\hat{R}_S(h) + \lambda\Omega(h)$ instead of $\hat{R}_S(h)$, where $\Omega(h) = ||h||_{L^2}$ for instance.

Typically $\mathcal{H}$ includes, for ML :

- linear/affine models : $h(x) = W\phi(x) + b$ where $b \in \mathbb{R}^d$, $W \in \mathcal{M}_{d,k}(\mathbb{R})$ and $\phi : \mathcal{X} \longrightarrow R^k$, used to deal with logistic regression for classification
- feature maps + linear head : $h(x) = w^T\phi(x)$ where $w \in \mathbb{R}^d$ and $\phi : \mathcal{X} \longrightarrow R^d$
- kernel methods : $h_\alpha(x) = \sum_{i=1}^{n} \alpha_i k(x, x_i)$ for a kernel k

1.2 Deep Learning

The main difference lies in the choice of $\mathcal{H}$, we will here use **Deep Neural Networks**. A neural network is a directed acyclic graph which describes a composition of differentiable functions.

For a network with L layers we have, for instance: $f_\theta(x) = f_L(f_{L-1}(...(f_1(x)...)))$ where each layer performs, in general, an affine transformation followed by a non-linear activation function (in most cases ReLU: $\sigma(z) = \max(z, 0)$):

$$f_l(x) = \sigma(W_l f_{l-1}(x) + b_l).$$

$\theta = \{W_l,\ b_l\}_{l=1}^L$ are the learnable parameters.

Early layers learn low-level features and deeper layers capture higher-level, more abstract features.

1.3 Representation Learning vs Feature Engineering

This disctinction is the fundamental reason why DL overtook classical ML in perceptual tasks.

Indeed, let's say we seek a predictor $f(x) = w \cdot \phi(x) \approx y$ with a feature map $\phi : \mathcal{X} \longrightarrow \mathbb{R}^k$. ϕ transforms raw data into a more useful representation, w is usually a linear classifier.

In classical ML (Feature Engineering), we need to fix ϕ, the problem is then to find the optimal weights w given phi. Thus, the performance is bounded by the quality of ϕ.

In DL (Representation Learning), we treat ϕ as a parameterized function ϕ_θ (a NN) and we learn both θ and w at the same time. The model discovers the optimal features by itself and for the specific task we are dealing with. On the other hand, it requires more data to perform.

2 Feedforward Networks

Often called Multi-Layer Perceptrons (MLPs), they are the foundational architecture of DL.

The **Universal approximation theorem** implies that a feedforward network with a single hidden layer can approximate any continuous function on a compact subset of $\mathbb{R}^n$, given appropriate activation functions.

Let's study an MLPs with L layers, the network defines a function $f : \mathbb{R}^{d_{in}} \longrightarrow \mathbb{R}^{d_{out}}$ so that:

$$x_0 = x,\ x_L = f(x) \text{ and, for each layer l: } x_l = \sigma_l(W_l x_{l-1} + b_l) = f_l(x_{l-1}, \theta_l).$$

The non-linearity introduced by σ allow the network to warp the input space to underline non-linear relationships between the features. Common choices for σ are ReLU or sigmoid ($\sigma(z) = \frac{1}{1+e^{-z}}$).

We define θ as the set of trainable parameters.

Training an MLP leads to minimizing the non-convex function:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(f_\theta(x_i), y_i)\ (J_{MSE}(\theta) = \frac{1}{n}\sum_{i=1}^{n} ||f_\theta(x_i) - y_i||^2 \text{ for instance.})$$

To find the optimal θ that minimizes $J(\theta) = \hat{R}_S(f_\theta)$ we use, in most cases, the **Stochastic Gradient Descent** (SGD):

$$\theta_{t+1} = \theta_t - \eta_t \nabla_\theta l(f_\theta(x_{i(t)}, y_{i(t)}))$$

where $i(t)$ is a random index.

Computing the gradient is done via **Backpropagation** which is an application of the chain rule :

$$\frac{\partial J}{\partial \theta_k}(\theta_k) = \frac{\partial J}{\partial x_k}(x_k)\frac{\partial x_k}{\partial \theta_k}(x_{k-1}, \theta_k) = \frac{\partial J}{\partial x_k}(x_k)\frac{\partial f_k}{\partial \theta_k}(x_{k-1}, \theta_k).$$

Where $\frac{\partial J}{\partial x_k}(x_k)$ is computed in the backward pass and x_k in the forward pass.

Networks are often regularized by the squared L2 loss of the learnable parameters, leading to minimizing for instance :

$$J(\theta) = J_{MSE}(\theta) + \lambda||\theta||_{L^2} \quad \text{where } \lambda \text{ is a scalar hyper-parameter.}$$

Most of the time we don't compute the full gradient but a **Batch** stochastic gradient descent :

$$\theta_{t+1} = \theta_t - \eta_t \frac{1}{B} \sum_{n=1}^{B} \nabla_\theta l(f_\theta(x_{i(t,b)}, y_{i(t,b)}))$$

where B is the batch size. We use this method to accelerate gradient computing (by computing in parallel on GPUs). Moreover, this method converges faster.

η_t is frequently called the **Learning Rate**, the most common strategy is to pick a high learning rate (that still leads to steady loss decrease, typically 10^{-2}, 10^{-3}) and keep it constant until convergence. The idea is that a hight lr increases chances of getting out of local minima. Once the losses seems to have converged, we can decrease the lr.

To improve the training stability and convergence, we often use **Batch Normalization** (BN), which both has estimated and learned parameters. It is frequently implemented just before non-linearities and has different types of behavior during training and inference.

- Training : the estimated parameters are, for a batch B, his mean and standard deviation $\mu_B = \bar{x}_n$ and $\sigma_B^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_B)^2$, we note a and b its learnable parameters and its output is :

$$y_i = a\frac{x_i - \mu_B}{\sigma_B} + b$$

- Testing : at this point, data might not be available in batches, we thus keep running estimates of μ and σ, which we may update during iteration after sampling a batch B by :

$$\mu \leftarrow m\mu + (1-m)\mu_B, \quad \sigma \leftarrow m\sigma + (1-m)\sigma_B \quad \text{where } m \text{ is a hyperparameter close to 1 (0.9 for instance).}$$

2.1 Convolutional Neural Networks (CNNs)

We consider a tensor $X \in \mathbb{R}^{C_{in} \times H \times W}$ as our input (Channels, Height, Width). Our trainable parameters form a kernel $K \in \mathbb{R}^{C_{out} \times C_{in} \times k \times k}$, a single output channel is given by:

$$Y_{u,v} = (K \star X)_{u,v} = \sum_{c=1}^{C_{in}} \sum_{i=1}^{k} \sum_{j=1}^{k} K_{c,i,j} X_{c,u+i-1,v+j-1} + b.$$

A standard CNN is thus a sequence of convolution, non-linearity and pooling repeated L times.

3 Transformer Architecture

We first introduce a new type of architecture.

3.1 Recurrent Neural Networks (RNNs)

Unlike MLPs where the internal state h_l depends only on the current input h_{l-1}, an RNN **maintains a 'memory'**.

Let $x = (x_1, ..., x_T)$ be a sequence of inputs where $x_t \in \mathbb{R}^d$. Then the RNN maintains a hidden state $h_t \in \mathbb{R}^k$ according to:

$$h_{t+1} = \sigma(W_{hh}h_t + W_{xh}x_{t+1} + b_h) \quad \text{and} \quad y_t^* = \phi(W_{hy}h_t + by) \quad \text{the output at time } t$$

To train this network we may use a total loss function: $\mathcal{L} = \sum_{t=1}^{T} l(y_t^*, y_t)$. To update W_{hh}, we need to compute $\frac{\partial \mathcal{L}}{\partial W_{hh}}$, which means computing, for each time step t:

$$\frac{\partial l_t}{\partial W_{hh}} = \sum_{i=1}^{t} \frac{\partial l_t}{\partial y_t^*} \frac{\partial y_t^*}{\partial h_t} \frac{\partial h_t}{\partial h_i} \frac{\partial h_i}{\partial W_{hh}}$$

or:

$$\frac{\partial h_t}{\partial h_i} = \prod_{j=i+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \approx \prod_{j=i+1}^{t} W_{hh}$$

which leads to a matrix exponentiation.

There are two cases:

- Eigenvalues λ of W_{hh} are such that $\max_\lambda |\lambda| < 1$, then $\prod_{j=i+1}^{t} W_{hh}$ tends to zero and the network forgets long-term dependencies
- $\max_\lambda |\lambda| > 1$ and the gradient explodes.

This spectral instability is the main reason why RNNs are difficult to train.

To tackle this problem we introduce the Transformer architecture.

3.2 Transformer Architecture

Let $X \in \mathcal{M}_{n,d}(\mathbb{R})$ be our input matrix of n vectors embedded in d dimensions. It is important to notice that transformer architecture treats the input as a set and does not have an inherent notion of order.

However, it is possible to fix it by injecting **Positional Encodings** into X (typically vectors containing $\sin \setminus \cos$ frequencies).

First, we project X into three distinct subspaces:

- **Queries** ($Q = XW_Q$) : what is Q_i looking for ?
- **Keys** ($K = XW_K$) : what does W_i mean ?
- **Values** ($V = XW_V$) : what does V_i give if selected ?

We often use two different sets of inputs, one to compute queries, and the other one to compute both keys and values (it is called cross-attention).

The **Attention Mechanism** works by mapping a query and set of key-value to an output:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \quad \text{with} \quad \text{softmax}(\frac{QK^T}{\sqrt{d_k}})_{i,j} = \frac{(\exp(QK^T)_{i,j}/\sqrt{d_k})}{\sum_{k=1}^{n} \exp((QK^T)_{i,k}/\sqrt{d_k})}$$

- QK^T computes the dot product between queries and keys, it measures the alignment between them
- $\frac{1}{\sqrt{d_k}}$ is the scaling factor if components of Q and K are independent with mean 0 and variance 1, their dot product has also a mean of 0, we then define d_k as its variance so that the global variance is 1
- finally, the output $\text{Attention}(Q, K, V)_i$ is a convex combination of all value vectors weighted by their relevance.

Still, a single dot product only captures one type of similitude. To capture multiple, we run the process multiple times with different matrices : $\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$.

3.3 Language Models

Most Language models are transfromers.

The problem is that text can't easily be converted into a continuous fixed-sized representation. We then represent it as a sequence of elementary objects named **Tokens**. A successfull method is to use vocabularies of tokens in-between characters and words (referred as a sub-word tokenization). A popular sub-word tokenization is the **Byte Pair Encoding** which defines tokens by :

- Define each charachters as a token
- Iterate until reaching a target number of tokens :

- For each pair of successive tokens, count the number of occurences
- The most common paire defines a new token
- Replace every possible pair with the new token.

There is typically $30-40$k tokens in modern deep networks, which are then projected into a space of typical dimension $512-4096$.

A LM obviously needs a positional encoding, a good way to add it is to use sine and cosine position embeddings PE. If we denote by d the tokens dimension, then $PE(pos) \in \mathbb{R}^d$ is defined for position pos of the token and $i \in \{1, ..., \frac{d}{2}\}$ by :

$$PE(pos)_{2i} = \sin(\frac{pos}{10000^{2i/d}}),\ PE(pos)_{2i+1} = \cos(\frac{pos}{10000^{2i/d}}).$$

We then process with two transformer architectures:

- **Encoder** : it takes as input a set of N tokens and outputs N tokens. It applies K transformers blocks, all with the same architecture, to the tokens:
  - A linear layer predicting key, queray and value tokens and computing multi-head-attention over them
  - The output for each token is added to the input token
  - Layer normalization is applied to the results
  - Outputs are then processed (independently) by a small MLPs (2 layers) with residual connection
  - Then a last layer normalization.
- **Decoder** : It works as the encoder but in each block, between the first linear layer and the MLP is added a multi-head-cross-attention layer (query tokens are computed from the current decoder tokens, while the keys and values are computed from the output tokens of the encoder).

3.4 Image classific Vision Transformers : ViT architecture

The Vit architecture is a transformer encoder. Here, we deal with linear embedding of image patches with 2D positional encoding. Moreover, the network takes as an input an additional learnable token : the **class token**. The only output considered to predict image classes is the output corresponding to the class token.