# 1. Deep Learning

Deep learning (DL) is a subset of a machine learning that employes model called artificial neural networks (ANNs), which are inspired by structure and function of the human brain, to learn from data and make predictions. In human brain, information is transmitted through chemical signals. Dendrites collect charges from synapses, where these contributions accumulate in the postsynaptic neuron. When a threshold is surpassed, the neuron becomes active, firing electrical signals along its axon.

ANNs are composed of numerous interconnected nodes called neurons replicating the functioning of brain neurons (Figure 1.1). Each neuron integrates inputs from many other neurons, where these inputs are weighted by specific values representing their significance. Within a neuron, inputs are summed, and if a certain threshold is surpassed, the neuron generates an output signal, which is transmitted to connected neurons.

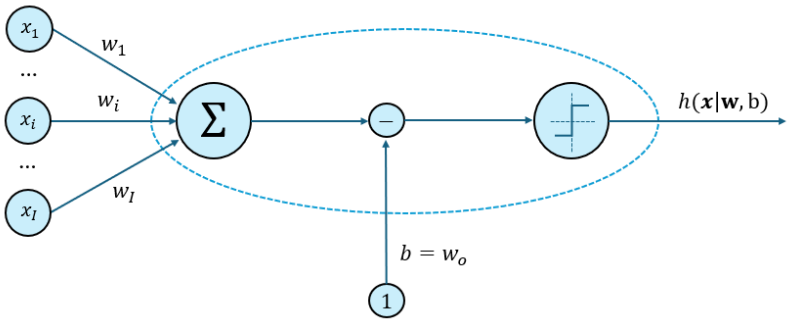


*Figure 1.1: artificial neuron representation; **x** are the inputs, **w** are the weights, Σ represents the summation operation, b is the threshold, and h is the stepwise function. The output is generated if the summation is greater than b.*

Mathematically, the behavior of an artificial neuron can be expressed as follows:

$$h(\boldsymbol{x}|\boldsymbol{w}, b) = h(\sum_{i=1}^{I} w_i * x_i - b) = h(\sum_{i=o}^{I} w_i * x_i) = h(\boldsymbol{w}^t \boldsymbol{x}) \tag{1.1}$$

Here, $\boldsymbol{x}$ is the vector of inputs from previous neurons, $\boldsymbol{w}$ represents the associated weights, $b$ is the bias term and $h$ is an arbitrary nonlinear function, also referred to as the activation function.

ANNs are structured as ensembles of connected neurons, with model parameters mainly representing weights and bias of each neuron. Remarkably, even the single neuron can constitute a model. In fact, the earliest DL model consisted of just one neuron. This model, known as the perceptron, was proposed by Rosenblatt in 1959. The perceptron receives input data and produces a prediction as output. However, its simplicity imposes limitations, particularly for complex tasks such as nonlinear classifications since its linear decision boundary is inadequate. To overcome its limitations, the concept of multilayer perceptron (MLP) was developed. A MLP is an ANN composed of multiple neurons organized in interconnected layers (Figure 1.2). The input layer receives the raw

data; one or more hidden layers process the information through a series of nonlinear transformations and an output layer produces the final result. Typically, ANNs are composed of a succession of many hidden layers, which is where the term “deep learning” originated. The use of many layers allows the network to learn hierarchical representations by transforming input data layer by layer. The MLP is also denoted as feedforward neural network (FNN) since the information flows thorough the network in one direction, from the input layer to the hidden layers, and finally to the output layer.

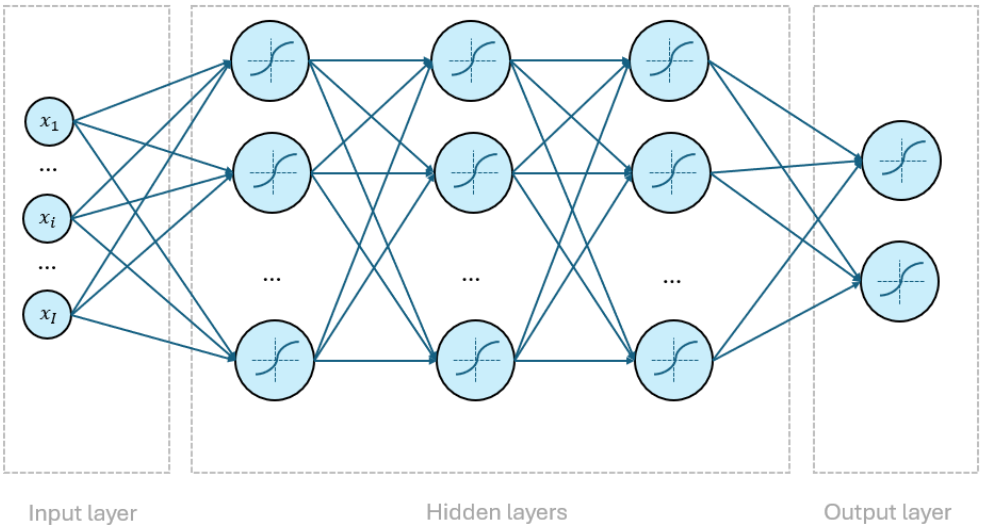


*Figure 1.2: MLP architecture. Input layer, Hidden layers (here 3) and Output layer are visible in the figure; each circle is an artificial neuron with its activation function, **x** is the input, blue arrows indicate neuron weights.*

Based on the MLP architecture, more complex models have been developed over the years to handle different types of data, resulting in diverse application across various fields. Particularly notable are convolutional neural networks (CNNs) which are suitable for handling images and are widely used in computer vision and medical diagnosis. Recurrent neural networks (RNNs) are designed for processing sequential data such as time series or text sequences and are utilized in applications like time series forecasting and natural language processing.

DL has gained popularity over the years for several reasons. One major motivation is the superior performance of ANNs and MLP, due to their complex architecture and behavior, compared to traditional ML models. Most importantly, ANNs eliminate the need for manual feature engineering, a necessary process for ML models. DL models such as CNNs automatically learn to extract features from raw data, while ML relies on hand-crafted features that demand time, domain knowledge and effort. Additionally, hand-crafted features often perform worse, as CNNs are trained to capture more complex and hierarchical patterns.
However, DL has some drawbacks. DL models necessitate large amounts of training data to achieve acceptable results, and their training requires significant computational resources. Moreover, ANNs are notorious for their lack of explainability, as it is very difficult to interpret how they reach their decisions. This may be problematic when not only the outcome is needed but also a clear understanding of the decision-making process.

## 1.1 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a specialized kind of ANNs designed to process data that has a known grid-like topology, such as images. As their name suggests, CNNs are simply neural networks that use convolution in place of a general matrix multiplication in at least one of their layers (Goodfellow et al., 2016). The mathematical operation of convolution for 2-D discrete data, such an image of width M and height N, can be formulated as:

$$S(i,j) = (I * K)(i,j) = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} I(m,n) K(i-m, j-n) \tag{1.2}$$

Where $I$ is often referred to as the input, $K$ as the kernel, $S$ as the feature map and $*$ denotes the convolution operation.

CNNs use layers of convolutions to detect the presence of specific features in images. The kernel slides over the image's width and height; at each position, the dot product is calculated between the kernel's weight and pixel values under the kernel (the kernel's receptive field). This process produces a single value that represents how strongly the feature described by the kernel is present in that region of image. As the kernel moves across the entire image, these values form the feature map, which indicates the presence and intensity of the feature at various image locations. CNNs frequently consist of multiple convolutional layers stacked on top of each other. This layered architecture allows the network to gradually interpret the visual information form the raw image data. In the initial layers, fundamental features such as edges, textures and color are detected. As the layers go deeper, they process inputs from the feature maps of previous layer, allowing the detection of more intricate patterns. It is important to emphasize how kernel weights are learned during training; this ensures that the learned representations are well-suited to the specific patterns in data to ultimately enhance model ability.

Typically, convolutional layers are interspersed by pooling layers. The aim of these layers is to gradually reduce the dimensionality of feature maps since smaller maps imply fewer parameters and reduced computational complexity. The typical forms of pooling are max pooling, which retains the maximum value within a certain image window, and average pooling, which takes the average value of pixels within the window.

Once the convolutional and pooling layers extract a reduced and meaningful representation of the input data, these representations are used as input to a series of fully connected layers (FCLs). FCLs use the previously extracted information to perform high-level reasoning and output predictions. These layers perform similarly to traditional MLPs, where each neuron is connected to every neuron of the previous layer. Ultimately, CNNs operate as FFNs similarly to MLPs. Essentially, a convolutional layer can be conceptualized as a hidden layer where the input is a feature map unrolled into a vector, and the output is the subsequent feature map. The weights are represented by kernel weights, with connection limited to local receptive fields, in contrast to the dense connection of FCLs.

FFNs, and therefore CNNs as a form of FFNs, are trained using the gradient descending (GD) algorithm and backpropagation (BP) algorithm. GD is an optimization algorithm used to minimize the model's loss function, which measures the difference between the model's prediction and the

actual values. GD works by calculating the gradient of the loss function with respect to the model parameters; parameters are then updated in the opposite direction of gradient, to descend the slope of the loss function towards a minimum. BP algorithm is instead used to calculate gradients required for GD. In BP, input data are forward propagated inside the network and an output is generated; the loss function is computed to measure the error between predicted and actual output. The error is then propagated backward through the network to compute the gradient of the loss function with respect to each weight. The backward direction is essential because it allows the reuse of partial derivative computations of one layer in the computations of previous layer, following the chain rule.

CNNs are particularly effective for image processing, although their reasoning can also be applied to time series. The most straightforward approach is to compute the time-frequency representation and treat it as an image. Alternatively, 1-D convolutions can be employed, with kernels operating along the time dimension, allowing the network to learn temporal patterns directly from raw time series.

## 1.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) are ANNs designed to process sequential data, such as time series or text sequences, where the order of data point is significant and meaningful. RNNs are characterized by their internal memory, which enables them to use previous outputs in the sequence to inform the processing of the current input. This is achieved through recurrent connections, where the output from certain nodes is fed back as input to the same nodes. Unlike traditional FNNs, which allow information to flow only in forward direction, RNNs are bi-directional ANNs, with the backward flow given by recurrent connections. Consequently, each hidden layer receives two kinds of input: the output from the previous layer and the output from the same layer at the previous time step (Figure 1.3).

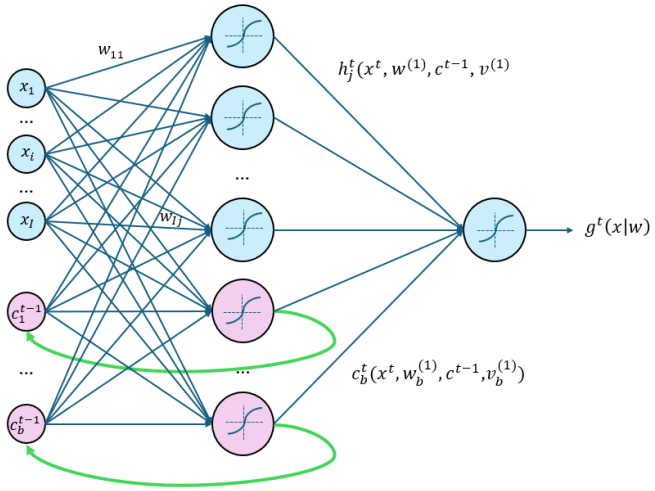


*Figure 1.3: the most general RNN architecture. Blue neurons are "forward" neurons, pink neurons are recurrent neurons; green lines indicate recurrent connection, with their inputs that can be fed both to recurrent and forward neurons.*

RRN training is quite similar to FFN training, as both rely on BP. However, RNNs are specifically trained using the backpropagation trough time (BPTT) algorithm(Werbos, 1990), which is an

extension of BP to account for the sequential nature of the input data. This procedure involves unrolling the computational graph of a RNN one time step at a time. The unrolled RNN is essentially a FNN where the same parameters are repeated across the entire unrolled network (Figure 1.4).

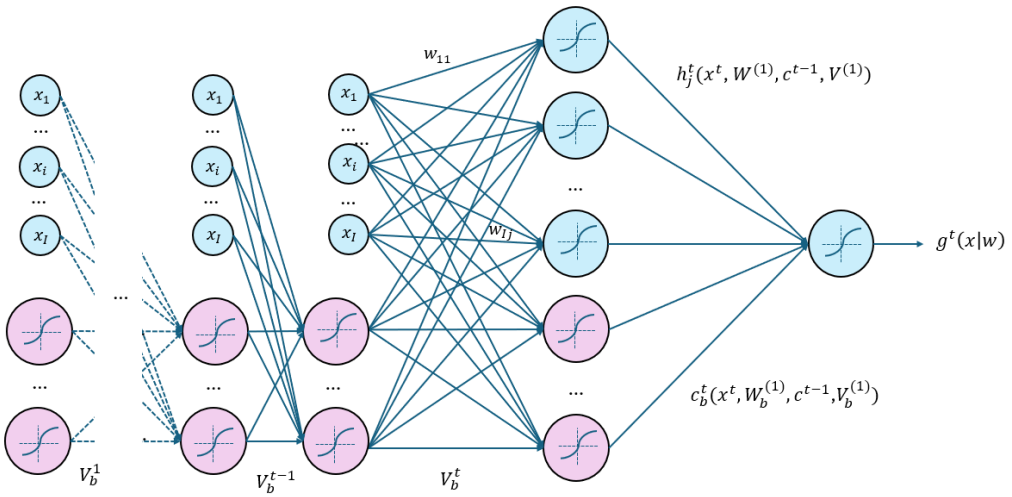


*Figure 1.4: unrolled RNN. The unrolling is done initializing weight replicas to be the same so for instance* $V_B^t = V_B^{t-1} = \ldots = V_B^1$

Once unrolled, the BP algorithm is applied in the same way as for a FNN. The gradient with respect to each parameter is calculated and summed across all instances where the parameter occurs in the unrolled network. The gradient descent algorithm then uses the average gradient to update each parameter:

$$V_b = V_b - \eta * \frac{1}{U} \sum_{u=1}^{U} \frac{\partial E^t}{\partial V_B^{t-u}} \tag{1.3}$$

Where $V_b$ is the recurrent weight, $\eta$ is the learning rate, $U$ is the length of the sequence to be processed and $\frac{\partial E^t}{\partial V_B^{t-u}}$ is the gradient with respect to $V_B^t$ calculated at different time steps.

However, RNNs are widely known for their inability to process excessively long sequences through BTTP. BTTP typically calculates gradients by multiplying weights and activation derivatives at each time step. As the input sequence progresses, these multiplicative terms accumulate.
Model weights can be unpredictably higher or lower than one while derivative values are always equal or less than one, especially with common activation functions such as ReLU, Thanh or sigmoid. This dynamic ultimately leads to vanishing gradients, where gradients become very small, slowing down or preventing training. Less frequently, it can cause exploding gradients, where the gradient becomes excessively large, leading to oscillating weights and unstable training.

### 1.2.1 Long Short-Term Memory Networks

Long short-term memory networks (LSTMs) are a type of RNN proposed in 1991 by Hochreiter & Schmidhuber (Hochreiter & Schmidhuber, 1997) to solve problems of processing long sequences.. LSTMs superiority relies in a novel recurrent network architecture in conjunction and an appropriate gradient-based learning algorithm. LSTMs are suitable for learning long input sequences without loss of short time capabilities, as they are designed to solve vanishing gradient problem and mitigate exploding gradient to some extent.

LSTM architecture is more complex than that of standard RNNs. To highlight the differences, we can consider that RNNs can be viewed as a chain of repeating modules (or cells). A standard RNN module is very simple, as it contains a single layer (Figure 1.5).

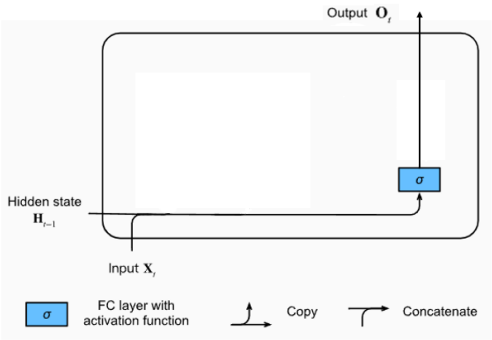


*Figure1.5: standard RNN cell. Hidden state (previous output layer) and current input are concatenated and processed through the layer.*

The LSTM module, instead of encapsulating a single layer, has four layers to effectively manage the interaction of current input and model memory. LSTM module utilizes a unique element called the cell state to stores long-term information. Within the module, the cell state is updated at every time step according to the current input and short-term information (i.e., the previous cell output or hidden state) in a mechanism regulated by gates. These gates serve as switches, controlling how one variable influences another variable within the cell. Gates are composed of a sigmoid layer followed by a pointwise multiplication operation. The sigmoid layer scales the input variable between 0 and 1. Those scaled value are then used in the multiplication operation to modulate the target variable. Within each cell there are three types of gates: forget gate, input gate and output gate (Figure 1.6).

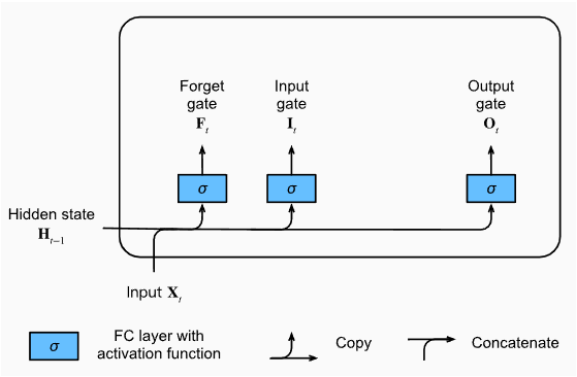


*Figure 1.6: Sigmoid layer of Forget gate, Input gate and Output gate before the pointwise multiplication.*

Gates within LSTM cell serve to control and regulate the information flow at different steps (Figure 1.7):

- The forget gate decides whether to retain or discard the current cell state. Specifically, the sigmoid layer takes as input the hidden state and the current input and its output multiplies cell state. Higher factors indicate more of the cell state is retained, while lower factors result in current information being discarded.

- The input gate determinates which portion of new information should be incorporated into the cell state. The new information is generated through a Tanh layer that receives current input and hidden state (input node). The input gate then processes this new information, determining its impact on the cell state.

- The output gate regulates which information from new cell state should be used as cell output. Initially, a candidate output is generated through a Tanh layer, which is then filtered by the output gate.

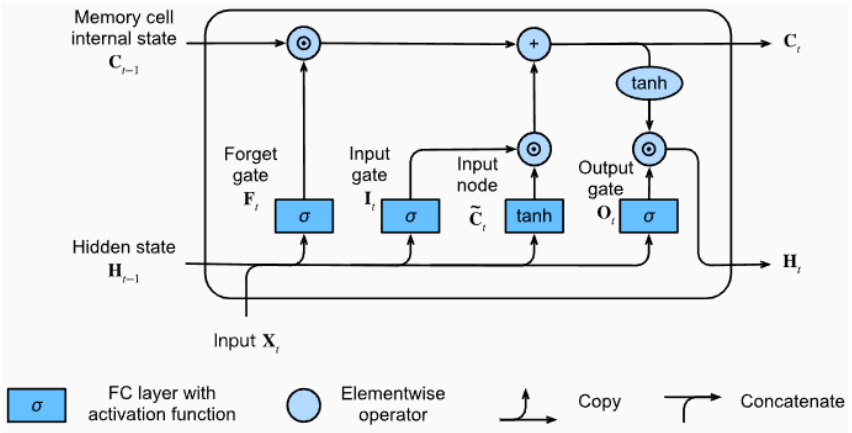


*Figure 1.7: LSTM cell and its entire inner representation; The LSTM cell processes the current input $X_t$, the hidden state $H_{t-1}$ and the memory cell $C_{t-1}$. It updates the memory cell and computes the output $H_t$.*

The mathematical operations within the LSTM cell can be formulated as:

$$F_t = \sigma(X_t W_{xi} + H_{t-1} W_{hi} + b_i) \tag{1.4}$$

$$I_t = \sigma(X_t W_{xf} + H_{t-1} W_{hf} + b_f) \tag{1.5}$$

$$O_t = \sigma(X_t W_{xo} + H_{t-1} W_{ho} + b_o) \tag{1.6}$$

$$\tilde{C}_t = \tanh(X_t W_{xc} + H_{t-1} W_{hc} + b_c) \tag{1.7}$$

$$C_t = F_t \odot C_{t-1} + I_t \odot \tilde{C}_t \tag{1.8}$$

$$H_t = \tanh(C_t W_{ch} + b_h) \odot O_t \tag{1.9}$$

LSTM are still trained through BTTP algorithm; however, their architecture solves the vanishing gradient issues that affects standard RNNs. Unlike standard RNNs, LSTM networks do not rely on the multiplication of activation derivatives and "static" weights over time. Instead, gradient calculations involve the multiplication of sums where sum components vary at each time step and may be arbitrarily lower or higher than one. This makes it less probable for the products to approach zero. Additionally, gradient calculations are influenced by the forget gate. During training, if gradient begins converging to zero, adjustments are made to the forget gate to increase its output, moving it closer to 1 and increasing gradients.

## Sources


Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735–1780.

Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, *78*(10), 1550–1560. https://doi.org/10.1109/5.58337