University of Oxford

Lincoln College

Doctorate of Philosophy

Particle Physics

Advanced Machine Learning Applications for the Higgs and Heavy Flavour Quarks at ATLAS

Candidate
Maxence DRAGUET

Supervisor
Daniela BORTOLETTO

2020-2024

Contents

List of Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
ATLAS	A Toroidal LHC ApparatuS
BDT	Boosted Decision Trees
CNN	Convolutional Neural Network
CPU	Core Processing Unit
DL	Deep Learning
DNN	Deep Neural Network
DT	Decision Trees
FPGA	Field-Programmable Gate Array
GAN	Generative Adversarial Network
GAT	Graph Attention Network
GNN	Graph Neural Network
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
HEP	High Energy Physics
LHC	Large Hadron Collider
LSTM	Long-Short Term Memory
MC	Monte Carlo
ML	Machine Learning
MLP	Multilayer Perceptron
MVA	Multivariate Analysis
NLP	Natural Language Processing
NN	Neural Network
PU	Pile-up
ReLU	Rectified Linear Units
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
VAE	Variational Auto-Encoder

Abstract

This confirmation of status report summarises some of the work I have carried out as part of the $VH(H \rightarrow c\bar{c})$ combined analysis of Run 2, with data from 2016 to 2018, as well as the development of the new jet flavour tagger called *DL1d*, designed to identify b- and c-jets with high efficiency.

CHAPTER 1

MACHINE LEARNING & DEEP LEARNING

This chapter is entirely dedicated to a review of relevant machine learning and deep learning methods in the context of High Energy Physics (HEP). As for other fields of science and technology, the recent advancements in the field of artificial intelligence have introduced many useful techniques that can be leveraged in particle physics. Before starting the review, some definitions of the different terms are presented. This is followed by presenting the most commonly used methods in particle physics: decision trees and deep neural networks. Finally, a word on optimisation technique is given at the conclusion of this chapter.

1.1 Definitions

1.1.1 Artificial Intelligence

Artificial Intelligence encapsulates any piece of software, any *program*, that aims to mimic an aspect of human intelligence, a non-exhaustive list of which includes:

- *Reasoning*, the ability to conduct logical thoughts and estabish their validity.
- *Inferring*, the ability to connect logical statements to induce or deduce new statements.
- *Creativity*, the ability to generate new information.
- *Acting*, the ability to perform a task or to change the environment.

Artificial Intelligence (AI) is a large field of study that studies these various aspects in many different subjects such as robotics, Natural Language Processing (NLP), computer vision, generative modelling, and reinforcement learning. Artificial intelligence can be broadly separated into three levels according to the performance of the created system:

1. *Narrow Intelligence:* representing artificial intelligence capabilities on a specific and specified problem, for which the underlying software is uniquely trained. This field includes *reactive AI*, where a model would be trained to output an optimal move based on current conditions only and *limited memory AI*, where a model is able to draw knowledge from past data to make decisions.

An example of the former is the IBM chess player Deep Blue, while the latter is most famously demonstrated by OpenAI's chatGPT chat-box and other machine learning models.

2. *General Intelligence:* representing an artificial intelligence capable of matching human problem-solving skills. In particular, this hypothetical setting would let a machine learn new tasks on its own and extrapolate from its existing knowledge, to *transfer learning* already acquired. Such a model would have the ability to adopt and combine several of the traits of intelligence.

3. *Super Intelligence:* describes a hypothetical type of intelligence able to exceed human abilities and exhibit independent control of thoughts.

Of these, currently only the first one is accessible and the second one is an ambitious focus of research. The inception of reactive AI, the first approach attempted, can be found in the research into games in the 50s and 60s. This paradigm saw the rise of algorithms capable of searching for the optimal move in large search space of possible actions using *heuristics*, human-passed knowledge on useful features of the specific environment of the game. For example, in chess one can use the point system assigning arbitrary values to each piece to help decide what action is worthwile (e.g., a queen is worth more than a simple pion). In this reactive approach, neither the rules of the game nor the decision process are learnt. The former is forced into the search logic and the latter is the outcome of the search process. These limitations and dependency on human-encoded intelligence restrict the potential of reactive AI to specfic well-controlled settings with a high degree of human understanding and low environment complexity. Limited memory AI revolutionised the field by removing the need for complete human control of the data interpretation, letting instead a well-crafted mathematical model abstract and represent the information internally. It opens the door to applications that would not otherwise realistaclly be feasible, such as autonomous driving, speech recognition, seamless robotics, etc. This new paradigm has also been observed to outperform reactive AI in all settings (e.g., in chess) and can be exploited in abstract scenarios where heuristics finding is impractical. For this reason, the focus of this chapter is on limited memory AI as exemplified by machine learning.

1.1.2 Machine Learning

Machine Learning (Machine Learning (ML)) underpins the field of narrow AI with limited memory capabilities. It represents a shift of paradigm in AI: by moving away from human-declared logic-based rules written in a specific syntax, the wholemark of reactive AI that executes statements such as

If x happens, do y,

for an input x and an output y, it automates the representation and update steps with mathematical models of both the dataspace ($\mathscr{D}$) and the learning process:

$$\forall x \in \mathscr{D}, \text{ do } f(x) = \hat{y}; \text{ update } f(x) \text{ given } (x,y),$$

where $\hat{y}$ is the prediction of the model. In this respect, both the internal representation of the rules and the decision-making is underpinned by the trained mathematical model f. Essentially, two distinct steps are applied to a mathematical model underpinned by modifiable parameters:

1. *Inferring:* the model has to give its prediction $\hat{y}$ on a new data point x: $f(x) = \hat{y}$.

2. *Learning:* the parameters of the model are updated based on a specified training or fitting procedure, depending on whether the training will be progressively exposed to the data points of a training dataset or directly exposed to entirety of the set. The objective is to align the output of the model $\hat{y}$ with the expected behaviour y: given the couple (x,y), let $f(x) = \hat{y} \rightarrow y$ under training convergence - this means the model f has to become an accurate estimator of the label y. Note that not every ML model requires a declared target output y, a paradigm referred to as *unsupervised* learning.

The training process closely depends on the type of model being deployed. These can be broadly separated into two fields:

- *Classical machine learning:* includes decision trees (Boosted Decision Trees (BDT) or Multivariate Analysis (MVA), random forest, ...), Support Vector Machine (SVM), logistic regression, kernel methods, k-Nearest Neighbours, ...
- D*eep Learning* (Deep Learning (DL)): these methods are based on a core logical module call the Artificial Neuron. This module is then stacked into layers of given width, meaning a given number of neurons, and several layers of such modules are then connected along depth.

DL is thus very much a part of ML, only constituting a specialised approach to building models on a core unit. Non-DL are often referred to as *classical* machine learning and still prove valuable in many application thanks to their ease of use and their ability to be deployed in context with small dataset sizes. The task of a ML model can be mutlifold:

- *Classification:* the task of assigning a label to a data sample, e.g., this jet is labelled a b-jet. The general case is multiclass, with n labels possible, while a particular and common case is the binary classification case ($n = 2$).
- *Regression:* the task of predicting a continuous variable based on a data sample, e.g., the momentum of the particle is 15 GeV/c.
- *Features extraction:* given a dataset with specific features, reconstruct new features, e.g., given a set of tracks, reconstruct the secondary vertex. This case is a multidimensional case of regression combined with classificiation (do the tracks share a vertex?).
- *Generation:* output samples from a distribution matching the training dataset distribution. E.g., Given a sample of 1 million $t\bar{t}$ events, sample 10 new data points from the underlying statistical model.
- *Anomaly detection:* identify and flag rare events in an unlabelled dataset.

There are different paradigms of ML, divided mostly along the lines of the amount of human intervention [1]:

- *Supervised learning:* the data used for training is endowed with the information the model must predict. In the learning step, the model is therefore optimised to make predictions that closely align with the target. Classificiation and regression are the most common tasks that fall under this realm.
- *Unsupervised learning:* the data is not endowed with extra-information the model must learn to predict but rather has underlying features that must be extracted. The model is therefore trained with an objective to optimise without human intervention, and should discover patterns and insights without any guidance. Generative models and clustering are prime examples.
- *Semi-supervised learning:* also called *weak supervision*, is a paradigm combining the supervised and unsupervised approaches. The model is mostly unsupervised but can benefit from some labelled cases or human input (a technique also named *active learning*). A prime example is a clustering tasks followed by a classification of the formed clusters. This is particularly fruitful when the cost of labelling the data is expensive, as is the case with real human data but thankfully not so in the case of particle physics data.
- *Self-supervised learning:* a machine instructs itself on what tasks should be learnt. The goal of the model is loosely defined and the learning process should include superficial objective to learn.

- *Reinforcement Learning:* this paradigm of ML is dedicated to the setting of a game theoretic environment. An agent must explore its environment and can act by choosing a specific policy - a method by which the agent selects an action given its current situation. In Reinforcement Learning (RL), the objective is for the agent to learn how to construct the best policy to satisfy a reward function and therefore obtain the best outcome for itself.

These different settings can be explored both in ML and more spefically in deep learning. The latter is currently considered to be the most performant one thank to its ease of scaling and is the main subject of research at the moment.

1.1.3 Deep Learning

Deep Learning (DL) combines a family of methods that have quickly grown in popularity around 2010, with widely advertised results on competitive benchmark tasks in pattern recognition, as exemplified by the super-human performance of the *DanNet* model [2] based on Convolutional Neural Network (CNN) [3]. The basis of any DL method is the Artificial Neuron, a logical unit built to mimic the functioning of a human neuron. These neurons are then combined into layers of any numbers of neurons (the width of the layer) and the layers themselves are stacked into depth, with deeper layer receiving as input the output of earlier neurons. Different DL models are constructed by modifying the structure of the layers - in particular, the input, output, and activation function used - and the transfer of information between neurons, be that between layers, depth-wise, or between neurons, width-wise. DL is specifically well-suited to the setting of the A Toroidal LHC ApparatuS (ATLAS) experiment, because:

- Large datasets of both real and simulated data are available.
- Thanks to advanced Monte Carlo (MC)-based simulation programs, the simulated data points are faithful representations of the real data.
- The data and data-model from which the data originates is well understood in physics, the former coming from measurements from well-calibrated detectors and the second from crafted theories of the field.
- The data exhibits reach features that can be approached as an image, a sequence or a graph, all three of which being the main data representation studied in deep learning.

Given how important this form of AI has become in all technological fields, this chapter is primarily dedicated to introducing some of its approaches most relevant to High Energy Physics (HEP).

1.2 Machine Learning Methods for Physics

High-energy physicists enjoy a special relationship with ML methods. Experimental particle physics largely relies on statistical analyses of complex and large datasets, be that simulated using MC methods or collected from sophisticated detector apparata. A typical physics analysis can be described as the combination of five main elements:

1. Data collection: real data is collected from a detector exposed to the underlying physics desired, e.g., at CERN placing and callibrating the ATLAS detector at an interaction point of the Large Hadron Collider (LHC) to collect proton-proton collision data.
2. Simulated data is generated to match the condition of collection of the real data in terms of detector effects and operational conditions such as energy, Pile-up (PU) and luminosity. This simulated data englobes the best of our current theorical knowledge of the law of physics.

3. The detector of a modern particle physics experiment is a complex set of sub-detectors sensitive to different physical phenomena, as described in chapter REF. This low-level information collected by different devices must be processed and recombined to generate *objects*, aggregated information that often hold physical meaning. For examples, from hits in the tracking detector a track can be fitted and some of its physical properties, such as p_T, reconstructed. This task corresponds to a mapping from *low*-level $\rightarrow$ *high*-level information to reconstruct interesting and physically meaningful features of the measured data.

4. An analysis strategy is established, with objective to similarly restrict the full datasets of both simulated and real data to a portion of the dataspace that is most sensitive to the studied *signal* or process. The sensitivity aspect underlies the need to take into account limited knowledge of theorical physics, limited precision of the apparata, limited statistics of both simulations and data collected, etc. To optimise the analysis, selection rules are derived based on physically accessible information, e.g., the centre-of-mass energy, presence of leptons, the transverse momentum p_T, and other high-level object reconstructed in the previous step.

5. With the optimally selected set of real and simulated datapoints, a statistical model is built to quantify the agreement of the measured data with the expectations from the theory under the conditions of the experiment. This is most often achieved through a likelihood computation.

Modern advanced machine learning has the potential to improve all steps of this process:

1. The operational side of running the detector and the accelerators can benefit from RL methods for improved control of the different electronic device. Triggers, an essential component of the ATLAS experiment described later in this thesis, can be upgraded to use sophisticated DL model running online thanks to a hardware back-bone built on Field-Programmable Gate Array (FPGA).

2. Simulating a dataset using a MC approach is a computationally intensive task. Each event must pass through a selection of probabilistic step, with only a simulated datapoint sastifying all requirements ending in the usable sample. This process can be sped-up and optimised significantly with more advanced and refined MC methods, but the cost remains significant to generate datasets of sufficient statistics. Generative AI has the potential to accelerate this step by giving statistical model that can be efficiently sampled. Generative Adversarial Network (GAN) and Variational Auto-Encoder (VAE) have been shown to perform the sampling step in a competitive amount of time. However, a key limitation of these stasticial approach is their limited ability to encorporate the sophisticated theorical model required to simulate the data, and any discrepancy or un-closure introduces levels of disagreements that are counter-productive for the final objective of the physics analysis.

3. ML is particularly well-suited for the object reconstruction task. Broadly, ML-based method can offer scalable, efficient, and precise solutions for objects reconstruction. Important examples in ATLAS are identifying particles in the detector (e.g., τ identification), reconstructing missing transverse energe (E_T) in the triggers, and classifying heavy-flavour jets - as exemplified in the next chapter of this thesis dedicated to flavour tagging.

4. Historically, physicists have relied on a cut-base approach to select their data: they analyse each of the relevant variables for the physics problem at hand to try and identify the best features to use to restrict the dataspace through manually-defined restrictions. For example, to a measure an interesting physical quantity of the process of Z-bosons decaying to two charged leptons l^+l^-, restricting the invariant mass of the lepton pair $m_{l^+l^-}$ to lie close the Z-boson rest mass $m_Z \approx 91.19$ GeV/c^2 is beneficial, as most of the signal will be found there. Machine learning is able to entirely bypass this manual operation, learning directly from an appropriate set of

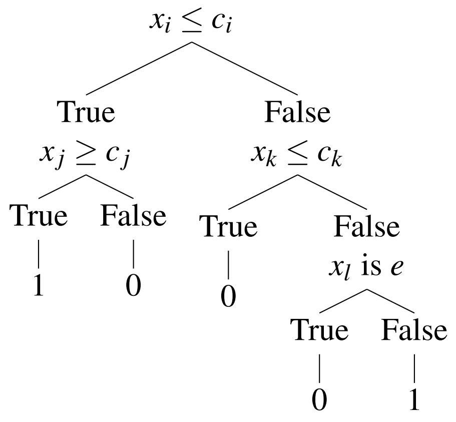
signal and background datasets with given features a transformation of the input data features to a discriminant optimising the separation of signal from background.

5. The likelihood function of the constructed statistical test, verifying the level of agreement between the real data and the theory through the simulated sample, can be directly learnt by a model given access to both sets. Furthermore, anamoly detection settings, such as those in the search for unknown resonances, can be derived using ML model in an unsupervised setting, thereby automating the discovery process and requiring only real data.

One of the main focus of this thesis can be broadly summarised as contributing to step 3 in the aformentioned list: developping DL-tools for improved object reconstruction. The analysis presented in the latter part of this document also introduces some classical ML technique of data selection - corresponding to step 4. The rest of this chapter now address a more detailed review of the relevant ML methods.

1.2.1 Decision Trees

Decision Trees (DT), also called *Classification and Regression Trees* (CART) are the bread-and-butter of any data analysis. They are simple to train, give a good ground performance for both classification and regression tasks, and are white box model - meaning there decision process is easy to interpret. Underlying the model is a recursive partitioning approach of the input space [1]. Labelling a partition step as *node*, the tree structure emmerges from a *root* state that is subsquently partitioned along different branches with one *leaf* per final region. The splits are done along a feature of the input space, and the method accept both discrete categorical values (e.g., the label of a lepton as e, μ, τ) and continuous values (e.g., m_l). For example, the following is a simple classification tree outputting the predicted class as 0 or 1:



At each node there is a learnt condition with x_i, x_j, x_k being continuous features of the dataset that are cut at the thresholds c_i, c_j, c_k and c_l is a categorical feature (e.g., is the lepton an electron). The leaf values are the output of the tree in different regions defined by the combination of successive selections - here a binary variable indicating a class. An example of a tree performing classification is shown in Figure 1.1, where a tree with two nodes is able to isolate most of the blue class from the red class with the region limited by green lines, corresponding to both conditions $x_1 \geq c_2$ and $x_2 \geq c_2$ being satisfied.

Finding the optimal set of partitions of a dataset is an NP-complete problem and therefore intractable for large datasets. To build a tree, a greedy approach must be adopted - meaning using a heuristic approach to find a satsifying solutions, e.g., successively choosing the most optimal step at each stage with no guarantee to find a global optimum instead of a local one. The chosen split is selected based on a defined *cost* function as suggested in Equation 1.1.

$$(j^*, t^*) = \arg \min_{j \in \{1,\dots,D\}, t \in T_j} \min \left(\text{cost}(\{x_i, y_i : x_{ij} \leq t\}) + \text{cost}(\{x_i, y_i : x_j > t\}) \right) \tag{1.1}$$

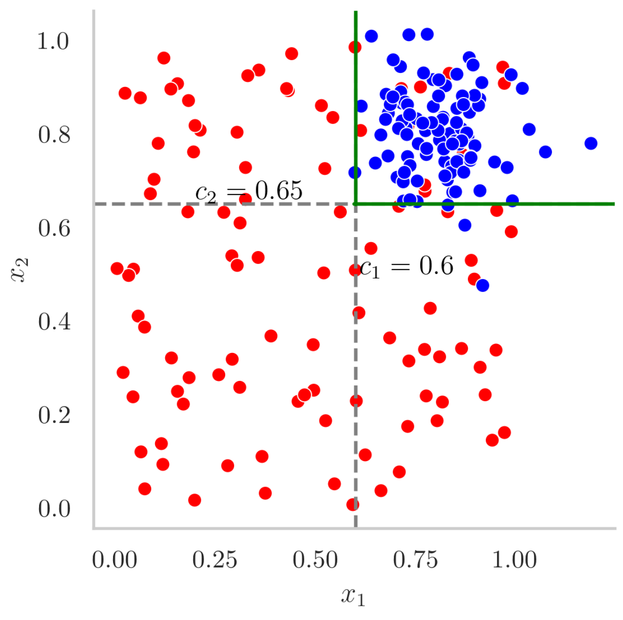


Figure 1.1: A binary classification problem with two features. A decision tree applies two successive cuts c_1 and c_2 to isolate most of the blue class from the red.

where T_j is the set of possible thresholds, x_j, y_j are the features and label (or regressive objective). For categorical variable, the inequality $x_j >< t$ is converted in a value equality $x_j == t$. The *cost* function will depend on the objective of the tree, with the regression case typically using the error function

$$cost(D) : \sum_{i \in D} (y_i - \bar{y})^2,$$

and for a classification the loss would be one of the following:

- *Missclassification rate:* $\frac{1}{|D|} \sum_{i \in D} \mathbb{I}(y_i \neq \hat{y})$, where D is the data in the leaf of the tree and $\mathbb{I}$ is the identity function: $\mathbb{I}(x) = 1$ if x is True, else 0.

- *Statistical entropy:* defining the class-condition probability as $\pi_c = \frac{1}{|D|} \sum_{i \in D} \mathbb{I}((y_i) \neq c)$, the entropy over the (C) classes is defined in Equation 1.2

$$H(\vec{\pi}) = -\sum_{c=1}^{C} \pi_c \log \pi_c, \tag{1.2}$$

  with $\vec{\pi}$ being a vector $(\pi_1, \pi_2, ..., \pi_C)$ with the class-condition probabilities as components.

- *Information Gain:* an equivalent formulation to the entropy, where the gain in information that should be maximised is the relative change in entropy by adding a selection on feature X_j over the current stage:

$$\text{Gain}(X_j < t, Y) = H(Y) - H(Y|X_j < t)$$

- **Gini:** computes and minimises the expected error rate:

$$\sum_{c=1}^{C} \pi_c (1 - \pi_c). \tag{1.3}$$

The pseudocode algorithm to train a DT with the update rule of Equation 1.1 is summarised in Algorithm 1.

DT can overfit a dataset: the model may tune itself to specific features of the training set that are not generalisable to any set samples from the true original distribution. Regularisation serves as an important step to avoid this often undesirable behaviour. For trees, a natural procedure to avoid

Algorithm 1 Recursive Procedure to Train a Decision Tree [1].

```
function FITTREE(node, $D$, depth)
    node.prediction $\leftarrow$ mean($\{y_i : i \in$D$\}$)
    $(j^*, t^*, D_L, D_R) \leftarrow$ split($D$)
    if not worthSplitting(depth, cost, $D_L$, $D_R$) then
        return node
    else
        node.left $\leftarrow$ FITTREE(node, $D_L$, depth + 1)
        node.right $\leftarrow$ FITTREE(node, $D_R$, depth + 1)
        return node
    end if
end function
```

overtraining is to interrupt the growth of the tree when it is no longer worth doing so - a criterion that is hard to decide *a priori* - or to *prune* the tree - removing nodes or branches that contribute little to the overall performance. A simpler way to regularise the performance by reducing the variance of the estimate of the model is to train several trees with different subsets of the data chosen randomly with replacement and aggregate the results into a single prediction. For example, for regression, one can take as output the average over each base learners:

$$y(x) = \frac{1}{N_l} \sum_{i=1}^{N_l} y_i(x),$$

for N_l base learner making prediction $y_i(x)$ given the input features x and for classification select the class by majority voting. This statistical technique of using ensemble of predictors is referred to as *bagging*. To further decorrelate the performance of the different predictors, these can be built on a subset of the input features and training datapoints, thereby forming a *random forest*.

1.2.2 Boosted Decision Trees

A popular extension to the simple decision trees approach is to introduce the concept of *boosting*, leading to a model referred to as a Boosted Decision Trees (BDT) or MVA in particle physics. Boosting is a greedy algorithm leveraging a weak learner or predictor (e.g., a DT) and applying it sequentially to weighted versions of the data, with a larger weight given to missclassified or miss-regressed datapoints, as per the use case. This method is hugely popular in data science, having earned the title *"best off-the-shelf classifier in the world"* [4]. Two particularly useful approaches are adaptive boosting (AdaBoost) [5] and gradient boosting [6], both combining an ensemble of M weak learners f_i $(i = 1, ..., M)$ into a strong learner F:

$$F(x) = \sum_{i=1}^{M} f_i(x).$$

For the following discussion, the model is built using a training dataset $\{(x_1, y_1), ..., (x_N, y_N)\}$ with input vectors $x_i \in \{\mathbb{R} \otimes \mathbb{D}\}^d$ of d features that are real or discrete ($\mathbb{D}$) and $y \in \mathbb{R}^d$ is a d-dimension real vector that serves as output to be predicted by the model.

AdaBoost

AdaBoost combines the M weak learners f_i with adaptive weights α_i to improve the ensemble performance as

$$F(x) = \sum_{i=1}^{M} \alpha_i f_i(x),$$

where F is the boosted model, and the successive boosting stages $F_T = \sum_{i=1}^{T \leq M} \alpha_i f_i(x)$ define stronger and stronger boosted variants of the model that combine weak learners f_i with a weight $\alpha_i \in \mathbb{R}$. At each iteration m of the training process ($m = 1, ..., M$), a weak learner f_m is fitted to the training set to minimise a loss function $L(y_i, F_m(x_i))$. The loss in AdaBoost is the exponential loss on the datapoints of Equation 1.4:

$$L(y, F_m(x)) = \sum_{i=1}^{N} \exp\left(-y_i F_m(x_i)\right) = \sum_{i=1}^{N} \exp\left(-y_i (F_{m-1}(x_i) + \alpha_m f_m(x_i))\right), \tag{1.4}$$

that the added new weak learner $\alpha_m f_m$ at step m has to minimise: $(\alpha_m, f_m) = \arg\min_{(\alpha_m, f_m)} L(y, F_m(x))$. The typical case for AdaBoost is binary classification with $y_i \in \{-1, 1\}$ but the algorithm can be generalised to other cases [1]. Equation 1.4 can be re-expressed as:

$$\sum_{i=1}^{N} w_{i,m} \exp\left(-\alpha_m y_i f_m(x_i)\right),$$

where $w_{i,m} = \exp\left(-y_i F_{m-1}(x_i)\right)$ can be interpreted as a weight applied to the datapoint indexed by i (x_i, y_i) at step m proportionaly to the error of the current strong learner. It can be shown that the weak learner f_m minimising the optimisation objective at step m is the one minimising the miss-classified weights sum error ε_m of the reweighted version of the dataset with weights $w_{i,m}$ [1], where:

$$\varepsilon_m = \sum_i w_{i,m} \mathbb{I}(y_i \neq f_m(x_i)).$$

For the first time step $m = 1$, these weights are initialised to $1/N$. The weights are then updated to

$$w_{i,m+1} = w_{i,m} e^{-\alpha_m y_i f_m(x_i)},$$

and renormalised so that $\sum_i w_{i,m+1} = 1$ before being assigned to each training sample i for the next step. The weak learner can be combined with the strong learner using an optimal weight α_m found by minimising the loss L of the combined learner:

$$\alpha_m = \frac{1}{2} \log \frac{1 - \varepsilon_m}{\varepsilon_m},$$

giving the overall update rule of Equation 1.5:

$$F_m(x) = F_{m-1}(x) + \alpha_m f_m(x), \tag{1.5}$$

combining the new weak learners f_m with optimal weight α_m to the current strong learner. The AdaBoost algorithm is summarised in Algorithm 2.

Algorithm 2 Adaboost for Binary Classification with Exponential Loss [1]

```
Initialise weights: $w_{i,1} = \frac{1}{N}$, where $N$ is the number of samples.
for $m = 1$ to $M$ do
    Minimise $\varepsilon_m = \sum_i w_{i,m} \mathbb{I}(y_i \neq f_m(x_i))$ on training set with weights $w_{i,m}$ to find $f_m(x)$.
    Compute $\alpha_m = \frac{1}{2} \log\left(\frac{1-\varepsilon_m}{\varepsilon_m}\right)$.
    Update weights: $w_{i,m+1} \leftarrow w_{i,m} \exp(-\alpha_m y_i f_m(x_i))$ and renormalised $\sum_i w_{i,m+1} = 1$.
end for
return $F(x) = \sum_{m=1}^{M} \alpha_m f_m(x)$
```

Gradient boosting

Gradient boosting is a generic approach which, contrary to AdaBoost, does not require a specific derivation for each loss function. The objective is to minimise the empirical risk, the expected value of the loss function L on the training set as shown in Equation 1.6

$$\hat{f} = \arg\min_{f} \mathbb{E}_{x,y} L(y, f(x)) \tag{1.6}$$

where $f(x) = (f(x_1), ..., f(x_N))$ is the output of the learner on the whole training set. As the name suggests, the approach leverages gradient descent to find the optimal $\hat{f}$. At step m, the gradient of the loss L is evaluated at $f = f_{m-1}$ as

$$g_{i,m} = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}},$$

which is then used to update the learner with a step

$$f_m = f_{m-1} - \alpha_m g_m,$$

where $g_m = (g_{1,m}, g_{2,m}, ..., g_{N,m})$ and the step-length α_m is chosen to minimise the residual loss $L(y, f_{m-1} - \alpha_m g_m)$. This implements functional gradient descent, and leads the model to fit the N datapoints of the set. This procedure naturally leads to overfitting the training set, an undesirable feature that is remedied by using a weak learner to approximates the negative gradient. In the specific case of gradient boosted decision trees, at step m a decision tree $h_m(x)$ is fitted to the pseudo-residuals $g_{i,m}$. This DT h_m at step m defines J_m disjoint regions through its leaves with predictions b_{jm} in each $j = 1, ...J_m$ region:

$$h_m(x) = \sum_{j=1}^{J_m} b_{jm} \mathbf{1}_{R_{jm}}(x),$$

where $\mathbf{1}_{R_{jm}}(x)$ is the indicator function - equals to 1 when $x \in R_{jm}$ and 0 otherwise. The update to the model is chosen so that:

$$f_m(x) = f_{m-1} + \alpha_m h_m(x),$$

with α_m selected by minimising the empirical risk of the updated model:

$$\alpha_m = \arg\min_{\alpha} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \alpha h_m(x_i)).$$

Algorithm 3 Gradient Boosting [1]

```
Initialise $f_0(x) = \arg\min_\alpha \sum_{i=1}^N L(y_i, \alpha)$
for $m = 1$ to $M$ do
    Compute the gradient residual for each $i = 1, ..., N$: $g_{i,m} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x_i)=f_{m-1}(x_i)}$
    Train weak learner $h_m$ on the dataset $\{(x_i, g_{i,m})\}_{i=1}^N$
    Compute $\alpha_m$ by minimising $\sum_{i=1}^N L(y_i, f_{m-1}(x_i) + \alpha h_m(x_i))$
    Update $f_m(x) = f_{m-1}(x) + \nu \alpha_m h_m(x)$
end for
return $f(x) = f_M(x)$
```

The full algorithm for Gradient Boosting is presented in Algorithm 3, where the update rule is added a *learning rate* hyperparameter ν to introduce regularisation and reduce the risk of overfitting. By keeping $0 < \nu \leq 1$, it limits the ability of the model to fully adapt to the training error, thereby improving generalisation. The price is a slower updating of the model and therefore a more demanding computational complexity. Further regularising techniques are bootstrap aggregation - training each weak learner on a random subset of the data -, limiting the number of leaves, or more generally penalising model of larger complexity - removing branches that do not reduce the loss by a minimal amount.

BDT resist better to overtraining thanks to the regularisation effect of boosting and the different techniques described in this section. Unfortuntaly, an undiserable feature of boosting is the loss of direct interpretability of the decision making but this is more than met by an appreciable *boost* in performance of the underlying model. An interesting property exhibited by all tree-based algorithms and many ML approaches is the ease to quantify the impact of a specific feature on the result. This technique of *feature importance* assigns a score to each input feature, typically the Gini importance of Equation 1.3, quantifying the reduction in entropy obtained by adding a feature, or the Shapley value, measuring the average marginal contribution of each feature.

Pros:

- *Adaptability to Different Distributions:* Boosting algorithms, such as AdaBoost and Gradient Boosting, can adapt well to different types of data distributions and can capture non-linear relationships in the data.
- *High Accuracy:* BDT easily achieve high accuracy in both classification and regression tasks, making them suitable for a wide range of applications.
- *Ensemble Learning:* The boosting technique leverages ensembling by combining multiple weak learners to create a strong learner, improving overall model performance.
- *Robustness to Overfitting:* Boosting helps mitigate overfitting, enhancing the generalisation of the model to unseen data. BDT typically perform reasonably well out-of-the-box.
- *Feature Importance:* BDT provide a measure of feature importance, aiding in feature selection and interpretability of the model.

Cons:

- *Sensitivity to Noisy Data:* BDT can be sensitive to noisy data and outliers, potentially leading to overfitting.
- *Computational Complexity:* Training multiple weak learners sequentially can be computationally expensive, especially for large datasets or deep tree structures.
- *Parameter Tuning:* BDT still require some fine-tuning of the hyperparameters, such as learning rate and tree depth, for optimal performance.
- *Black Box Nature:* The ensemble nature of boosted decision trees make them somewhat of a black box, sacrifing the perfect interpretability of DT for the sake of performance.

1.2.3 Artificial Neurons

The Artificial Neuron, also called the *perceptron* by its inventor Frank Rosenblatt in his seminal 1958 paper [7], is the logical gate that underpins most DL models. Notably, a Multilayer Perceptron (MLP) - also called a Deep Neural Network (DNN) - is a stacking of layers of artificial neurons. Inspired by biological principles, the perceptron, as shown in Figure 1.2, accepts multiple intputs and gives as output 1 if the combination of inputs exceeds a certain modifiable threshold, otherwise giving 0. This combination accepts weights to scale the input which, remarkably, can be adjusted during a training phase if the output of the perceptron is incorrect. Artificial neurons are a direct generalisation of this very principle, with the output no longer being thresholded but applied a chosen function f after being added a learnable bias term b.

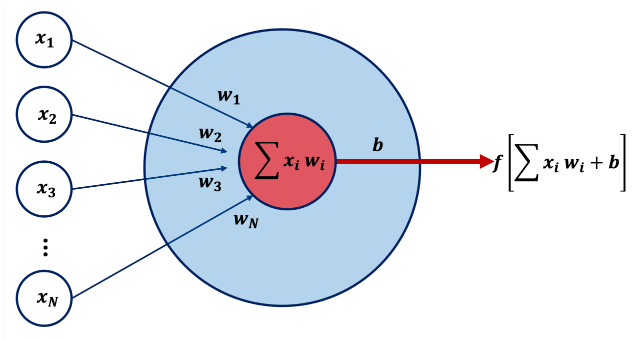


Figure 1.2: Schematics of a perceptron or an artifical neuron: the inputs x_i ($i = 1,...,N$) is multiplied by learnable weights w_i, summed and added a bias b before being passed to an output function f.

In this thesis, a *perceptron* shall refer to a single artificial neuron, which is equivalent to a logistic regression model. The interest of the artificial neuron stems from a significant theoretical result: stacks of artificial neurons are *universal function approximator* [8, 9], as is shown in the next section on deep neural network. This theoretical result is built on a mathematically advantageous function choice for f: the sigmoid σ, defined in equation 1.7 and shown in Figure 1.3:

$$\sigma(x) = \frac{1}{1+e^{-x}}. \tag{1.7}$$

Thanks to its property to map the range of real numbers to the [0, 1] range, this activation function is often used for numerical stability and to map some input to a probability distribution. A practical mathematical property of the sigmoid that is particularly relevant for DL is the ease to compute its derivative:

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

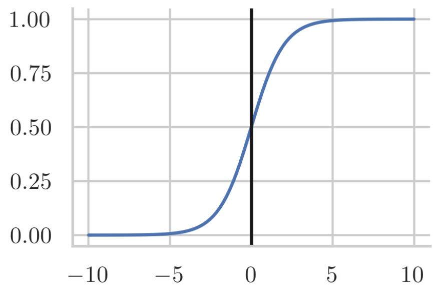


Figure 1.3: The sigmoid function σ.

The power of the artificial neuron comes from its ability to be efficiently combined into a well-ordered structure with powerful representation power. For an input $x \in \mathbb{R}^d$, a neuron individually applies an affine transformation $W_i^T x + b_i$, where $W_i \in \mathbb{R}^d$, $b_i \in \mathbb{R}$ are the weights and bias of the neuron i, that is passed through an activation function f for a total output of a single neuron $f(W_i^T x + b_i)$. Combining these operations, as shown in the next section, leads to a well-structure mathematical model that has the power to represent all well-behaved continuous functions.

1.2.4 Deep Neural Networks

A Deep Neural Network (DNN), also called Multilayer Perceptron (MLP), Artificial Neural Network (ANN), Neural Network (NN), or feed-forward neural network, is made by stacking layers of artificial neurons as shown in Figure 1.4 Each neuron in a layer receives as input the output of each neuron of the previous layer, and connects to each neuron of the next layer. Layers of artificial neurons that are placed between the input and output are said to be *hidden layers*. The particularlity of the design of this architecture is that layers of neurons connect to all neurons of the next layers only, defining a feed-forward computation graph from input x to output y. Mathematically, a single layers at depth i with m units given as input the previous layer of dim n at depth $i-1$ computes the transformation of Equation 1.8

$$a^i = f^i\left(W^{i^T} a^{i-1} + b_i\right), \tag{1.8}$$

where $W^i \in \mathbb{R}^{m\times n}$ is the matrix of weight of layer i - one row per unit of layer i, one column per unit of layer $i-1$, $b_i \in \mathbb{R}^m$ is the vector of bias, one per unit of layer i, f^i is the activation function of layer i, and $a^{i-1} \in \mathbb{R}^n$ are the n activated output of layer $i-1$. Note that strictly speaking the activation could be different for the units of the layer but is often unique per layer to accelerate matrix computations with vector operations.

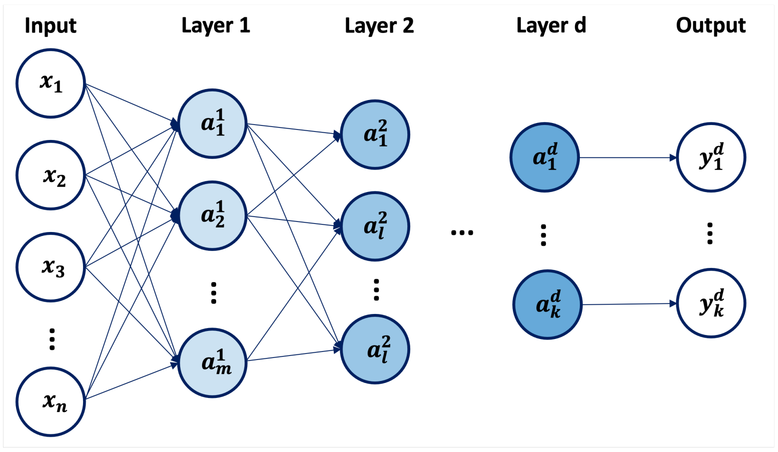


Figure 1.4: A deep neural network with d layers of width m, l ..., k. Each artificial neuron, represented by a ball of darkening blue along depth, computes an affine transformation of the input of the layer followed by an activation function. The input of the DNN is x and the output is y.

Neural networks implement a system computing combination of Equation 1.8. A powerful theoretical motivation for neural networks stems from the fact they are Universal Function Approximator. This theorem, or rather family of theorem depending on the type of activation f considered, establishes that neural network built with appropriate activation functions are able to approximate most well-behaving functions. The most famous such theorem states [8, 9]:

Theorem: *Let $C([0,1]^n)$ denote the set of all continuous function $[0,1]^n \to \mathbb{R}$ and σ be any sigmoidal activation function. Then the finite sum $f(x) = \sum_{i=1}^N \alpha_i \sigma(w_i^T x + b_i)$ is dence in $C([0,1]^n)$. In other words, given any $f \in C([0,1]^n)$ and $\varepsilon > 0$, $\exists$ a sum $G(x)$ of the above form for which:*

$$G(x) - f(x) < \varepsilon \quad \forall x \in [0,1]^n.$$

The statement of the theorem essentially is that any function defined over the n-dimensional unite hypercube $[0,1]^n$ can be approximated by an arbitrarily wide neural network. This result only requires σ to be sigmoidal or discriminatory in the sense:

$$\sigma(x) \to \begin{cases} 1 \text{ if } x \to \infty, \\ 0 \text{ if } x \to -\infty, \end{cases} \tag{1.9}$$

which is, for example, satisfied by the sigmoid function. Interestingly, similar theorems were derived for other important activation function commonly used in DL, for example for ReLu in [10]. Many flavours of DNN exist with different functions used. An important theoretical result is the requirement for the output function applied to the artificial neuron to possess some degree of *non-linearity*. A neural network with strictly linear functions can indeed be seen to collapse to a linear regression model. Such a function, when applied to the output of an artificial neuron, is said to *activate* it and is hence called *activation functions*. The most popular such functions, shown in Figure 1.5, are:

- The sigmoid function of Equation 1.7.
- The hyperbolic tangent function $tanh = \frac{e^x - e^{-x}}{e^x + e^{-x}}$.
- The Rectified Linear Units (ReLU) function of Equation 1.10

$$\text{ReLu}(x) = \max(0,x), \tag{1.10}$$

  note that the non-linearity here is strictly between positive and negative inputs, making this activation function one of the simplest that can be leveraged in DNN for universal function approximation. The deriviative of this function is indeed particularly easy to compute. A generalisation of ReLU called leakyReLU introduces a linear function in the negative range as $\text{leakyReLu} = \max(\alpha x, x)$, with $\alpha \in [0,1[$.

- The Exponential Linear Unit (ELU) function, shown in Equation 1.11, modifies the Leakage ReLU in the negative domain while keeping the saturation property:

$$\text{Elu}(x) = \begin{cases} x & \text{if } x >= 0, \\ \alpha(e^x - 1) & \text{otherwise}, \end{cases} \tag{1.11}$$

  where the a hyperparameter $\alpha > 0$.

- The softmax function. For an $x \in \mathbb{R}^n$, the softmax return a vector $\text{softmax}(x) = [..., \frac{e^{x_i}}{Z}, ...]$ (for $i = 1, ..., N$), where $Z = \sum_i e^{x_i}$. For a 2-dimensional x, the softmax is equivalent to the sigmoid function. In n-dimension, it maps each entry of x to the range $[0,1]$ and guarantees $\sum_i \text{softmax}(x)_i = 1$. The softmax is therefore helpful to define probability distributions over multidimensional outputs.

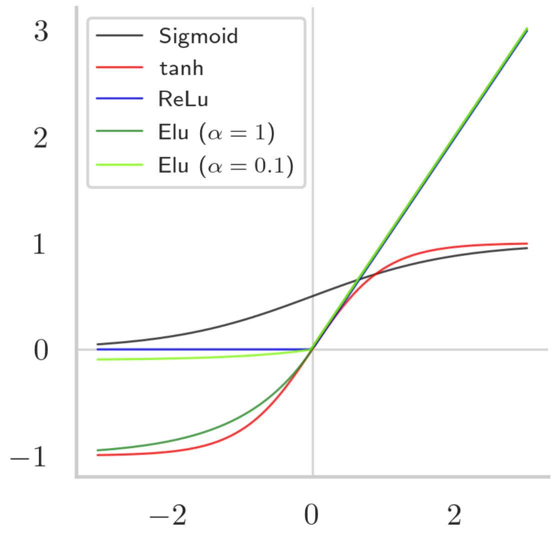


Figure 1.5: The most common non-linear activations used in deep learning.

Many more functions have been defined in the field of AI, with none seemingly offering a clear advantage over the ones listed here. The sigmoid is no-longer the choice of reference, due to its tendency to quickly saturates - meaning its gradient for large positive or negative input values vanishes in the sense that it tends to 0. The hyperbolic tangent *tanh* offers larger gradients thanks to its [-1, 1]range with steeper curvature and is often the activation of choice for autoregressive architecture, such as the Recurrent Neural Network (RNN). The ReLU function is the most widely used activation function as its derivative is extremelly efficient to compute - 1 for positive inputs and otherwise 0 - and does not suffer from vanishing gradients for positive values. Its fixed 0-value for negative input is a double edge sword: on one side, it helps the network regularises itself by deactivating neurons, on the other it could let some neurons being stuck in *off*-mode if the weights are such that its pre-activation value is always negative for the input data. For this reason, it is important to correctly initialises the weights of a DNN. An easy work around this limitation is the leakyReLu and the Elu activations, which both introduce some leakage

in the negative value as controlled by a parameter α.
While the Universal Function Approximator theorem gives a powerful endorsment of DL networks, it does not state how to derive such a netowrk to fit a specific function. The task of choising the right architecture depthwise and widthwise and the correct weights and biases must be approximated by a learning strategy updating the weights and biases to reduce the error between the output and a given target. Many other data models are also Universal Function Approximators and what sets appart DL-like models that are built on artificial neurons is the simplicity of the procedure to update their weights: thanks to their nice computational structure, under a reasonable choice of activation functions for each layer, the network is *differentiable*. This means a gradient can be computed on a function measuring the difference between the target y and the output $\hat{y}$ - often called a loss function when it should be minimised or a reward function when it is to be maximised - and *backpropagated* across each neuron of each layer to give a local update to be applied to each individual weight and biases. The recent rise of DL in AI can be traced back to improvements in making this backpropagation of the gradients with publicaly available software, such as PyTorch [11] and TensorFlow [12], implementing efficient algorithm to perform this essential step.
One of the main difficulty in training a neural network is the non-convexity of the loss function as a function of the model parameters. The large number of parameters typically implemented by neural networks require large dataset to correctly assign values to the parameters without suffering from overtraining - the property to only correctly describe the data of the training set without being able to generalise the performance to data not seen during the training phase. There are two main difficulties encountered when optimising a neural network: the non-convexity of the objective function means saddle points and local minima are typically abundant and the computational complexity due to the large number of parameters makes a single update using a large dataset expensive. To circumvent these, the *backpropagation* algorithm of Algorithm 4 is used [13].

Algorithm 4 Backpropagation Algorithm

```
function UPDATE($x$, $y$, $N$, $\mathscr{L}$)
    Forward step: propagate input $x$ through network $N$ to get prediction $\hat{y}$
    Loss: compute the loss or reward of $N$: $\mathscr{L}(y, \hat{y})$
    while $\exists$ a layer with no gradients do
        Take right-most layer required a gradient
        Take input of layer to its right
        Using the chain-rule of calculus, propagate gradient of next layer to that of current layer
        Store the gradient at the layer
    end while
end function
```

In summary, the backpropagation algorithm serves as an effective way to compute

$$\frac{\partial \mathscr{L}(\theta)}{\partial \theta} = \sum_{i=1}^{N} \frac{\partial l(\theta)}{\partial \theta},$$

where θ encapsulates all parameters of the model and there are N datapoints, with a per datapoint loss of l. The backpropagation algorithm starts with a forward pass through the network:

$$x \xrightarrow{\times W^1 + b^1} z^1 \xrightarrow{f^1()} a^1 \xrightarrow{\times W^2 + b^2} z^2 \xrightarrow{f^2()} a^2 \xrightarrow{\times W^3 + b^3} \dots \xrightarrow{\times W^d + b^d} a^d = \hat{y},$$

the loss is then computed $\mathscr{L}(y, \hat{y} | W^1, b^1, W^2, b^2, ..., W^d, b^d)$ before computing the gradient of each layer by starting from the right:

$$\text{grad}_{W^d, b^d} = \nabla_{W^d, b^d} \mathscr{L} \to \text{grad}_{W^{d-1}, b^{d-1}} = \nabla_{W^{d-1}, b^{d-1}} \text{grad}_{W^d, b^d} \to \dots \to \nabla_{W^1, b^1} \text{grad}_{W^2},$$

where the operation $\text{grad}_{W^{d-1},b^{d-1}} = \nabla_{W^{d-1},b^{d-1}}\text{grad}_{W^d,b^d}$ implies a use of the chain rule to obtain the local gradient, based on information already obtained. In the context of DL, the chain rule of multivariate calculus offers a transformation:

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial z}\frac{\partial z}{\partial x}, \tag{1.12}$$

for a function $h : \mathbb{R}^n \to \mathbb{R}^m$, composiong two functions $h = g \circ f$ defined as $f : \mathbb{R}^n \to \mathbb{R}^k$, $g : \mathbb{R}^k \to \mathbb{R}^m$, with $x \in \mathbb{R}^n$ and $z = f(x) \in \mathbb{R}^k$. Two types of updates are necessary:

- For a layer l, the activation can be unpacked:

$$\frac{\partial l}{\partial z^l} = \frac{\partial l}{\partial a^l}\frac{\partial a^l}{\partial z^l}$$

- For a layer l having access to next layer $l+1$ ($\frac{\partial l}{\partial z^{l+1}}$):

$$\frac{\partial l}{\partial z^l} = \frac{\partial l}{\partial z^{l+1}}\frac{\partial z^{l+1}}{\partial z^l} = \frac{\partial l}{\partial z^{l+1}}\frac{\partial z^{l+1}}{\partial a^l}\frac{\partial a^l}{\partial z^l},$$

  but since $z^{l+1} = W^{l+1}a^l + b^{l+1}$, this reduces to:

$$\frac{\partial l}{\partial z^l} = \frac{\partial l}{\partial z^{l+1}}W^{l+1}\frac{\partial a^l}{\partial z^l},$$

- Combining the two above, we can compute $\frac{\partial l}{\partial z^l}$ for each layer in backward order. One thus now requires the derivative with respect to the per-layer weights:

$$\frac{\partial l}{\partial W^l} = (a^{l-1}\frac{\partial l}{\partial z^l})^T,$$

  for the weight matrix - an outer product between a row vector $\frac{\partial l}{\partial z^l}$ and a column vector a^{l-1}. For the bias:

$$\frac{\partial l}{\partial b^l} = \frac{\partial l}{\partial z^l},$$

  the result is directly obtained from the row vector.

Once all the gradients are locally available, the next step is to update all the parameters to reduce the loss by taking a step in the direction opposed to the gradient, giving the largest reduction in loss. For example for a specific parameter w_{ij} at training step $t+1$:

$$w_{ij}^{T=t+1} \leftarrow w_{ij}^{T=t} - lr \times \text{grad}\left[w_{ij}^{T=t}\right], \tag{1.13}$$

where the *learning rate lr* controls how large a step is taken in the opposite direction of the gradient. Since the gradient of the earlier layers will be derived from the gradient of later layers, it is important for the gradients to respect some numerical stability to avoid the risk of vanishing gradient ($\to 0$) or exploiding gradients ($\to \infty$). This requires some care in the architecture choice and might motivate the use of a specific activation function over another. Note that strictly speaking, the activation function needs not be a continuous differentiable function (C^1), as the existence of a right or left derivative is sufficient, hence the function should at least be continuous (C^0).
Concerning the loss function $\mathscr{L}$, there is a lot of freedom in how to set it up - although it should be differentiable - and some typical choices are:

- The cross-entropy loss function - also called the logistic loss - is based on the notion of entropy, as defined in Equation 1.2, and is often used to assign probabilities in a classification with c classes:
$$-y_i \log \hat{y}_i,$$
where y_i is the real label of the datapoint and $\hat{y} \in [0,1]^C$ is the model prediction, respecting $\sum_i \hat{y}_i = 1$. Given the requirements on the output, it is typically combined with a softmax.

- the Mean Squared Error (MSE):
$$\frac{1}{N}\sum_i^N (y_i - \hat{y}_i)^2.$$

- the Mean Absolute Error (MAE):
$$\frac{1}{N}\sum_i^N |y_i - \hat{y}_i|.$$

To regularise the model, it is common to add terms to the loss function called *regulariser* in order to restric the model weights to a certain size. This can be achieved by adding to the loss $\mathscr{L}$ an L2-penalty
$$\lambda \sum_i w_i^2,$$
on the sum of the squared values of the weights, or an L1-penalty
$$\lambda \sum_i |w_i|,$$
where this last approach using the absolute value has the nice additional feature to make the network sparse - meaning to set un-necessary weights at 0. The amount of regularisation is controlled by the hyperparameter λ. Further regularisation can be obtained by randomly dropping out some connexions of the network during training, a technique called *dropout* and controled by the dropout probability p to incude an artificial neuron or not.

Pros:

- *Universal Function Approximators:* Feedforward networks, particularly deep ones, are known to be universal function approximators. They can approximate any continuous function to arbitrary precision given sufficient hidden units and appropriate activation functions.

- *Flexible Architecture:* The architecture of feedforward networks is flexible, allowing for customization in terms of the number of layers, the number of neurons in each layer, and the choice of activation functions. This flexibility makes them suitable for various tasks.

- *Feature Learning:* Feedforward networks can automatically learn hierarchical representations of features from the input data. This is beneficial for tasks where the underlying patterns are complex and not easily captured by handcrafted features.

- *Non-linearity Handling:* By introducing non-linear activation functions, feedforward networks can capture and model non-linear relationships in data, enabling them to solve more complex problems compared to linear models.

- *Availability of Optimisation Techniques:* Various optimisation techniques, such as gradient descent and its variants, are well-suited for training feedforward networks. This makes it possible to efficiently update the network weights during the training process.

Cons:

- *Limited Modeling of Sequential Data:* Feedforward networks are not naturally designed for handling sequential data and temporal dependencies. RNN and Transformer architectures are often preferred for tasks involving sequential information.
- *Fixed Input Size:* Traditional feedforward networks have a fixed input size. While techniques like padding or resizing can be employed, they might not be suitable for handling inputs of varying lengths in tasks like natural language processing.
- *Lack of Memory:* Feedforward networks do not have an inherent memory mechanism, which can be a limitation when dealing with tasks requiring the model to retain information over a sequence of inputs.
- *Vanishing and Exploding Gradients:* Training deep feedforward networks can be challenging due to the vanishing and exploding gradient problems. Gradients may become too small or too large during backpropagation, leading to difficulties in learning deep representations.
- *Need for Sufficient Labelled Data:* Feedforward networks often require a large amount of labelled data for effective training. In domains where labelled data is scarce, the performance may be limited.

This section introduces deep neural networks, the simplest feed-forward architecture constituted of artificial neurons stacked into layers to generate an output y based on an input x. There are many refinements to this base architecture, and some are explored in the next sections.

1.2.5 Recurrent Neural Networks

The first modification to the DNN considered in this thesis are Recurrent Neural Networks (RNN). These models were derived to work with sequences, such as occuring in natural language processing. The main change with the DNN architecture is in the way information is passed through the network: RNN are autoregressive models. The information flow is bi-directional: the computation sequentially processes through the same structure the input at given step with the output of the prior step. The advantage of this representaton is that this cyclical flow can be unfolded into a direct acyclical computational graph that, for a given sequence length, is equivalent to a DNN pass. Figure 1.6 presents the structure of an RNN-based network as well as its unfolding. The input x is a sequence of N tokens, and the length of different inputs x_i in the dataset can vary. The mathematical structures implemented by this architecture to generate an output y of length equal to the input is that of Equation 1.14 for the step t ($t = 1, ..., N$):

$$y_t = W(h_t) = W(V(x_t) + U(h_{t-1})), \tag{1.14}$$

where U, V, and W are three, potentially different, DNNs mapping taking at timestep t respectively the previous hidden state h_{t-1}, the current input token x_t and the new hidden state $h_t = V(x_t) + U(h_{t-1})$. The initial hidden state h_0 is usually initiated from a special mapping from the whole input x. An interesting feature of such a network is its ability to build an internal memory of previous inputs up to a timestep T thanks to the chain of hidden states $h_{t<T}$. To avoid having too large (exploding) or small (vanishing) gradients, which would not correctly update the weights of the model during gradient descent, the tanh function is often used as activation in RNN thanks to its smooth distribution and limitation to the range $[-1, 1]$. Unfortunately, due to the network relying on repetitive multiplications of numbers in the range $[-1, 1]$, the effect of much earlier timestep ($h_{t<<T}$) can be lost when processing later input at T. This process is referred to as *memory loss*. Thankfully, there are many main architectural modifications to RNN that improve their operational memory, of which the two main ones are Long-Short Term Memory (LSTM) and Gated Recurrent Unit (GRU).

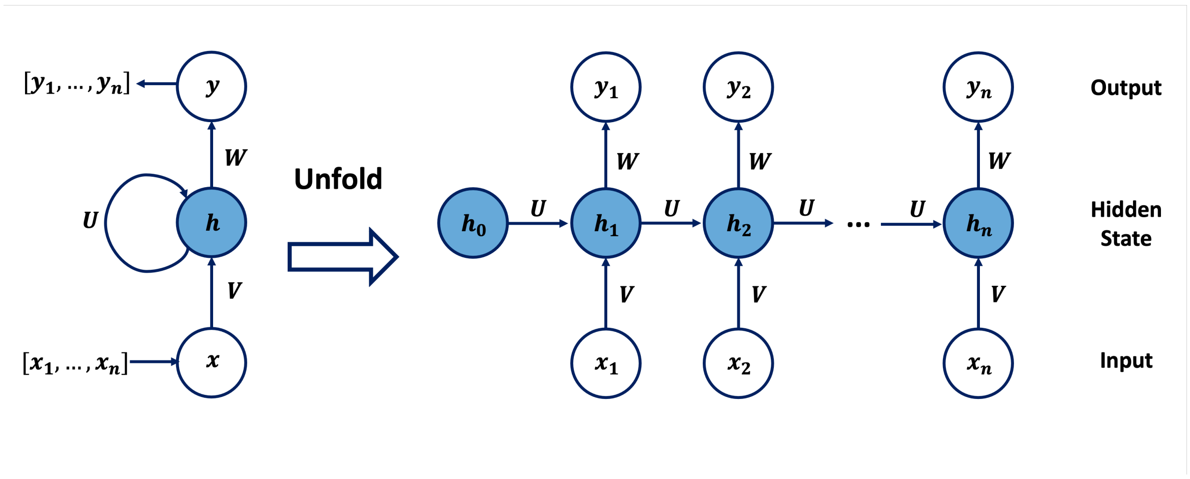


Figure 1.6: A recurrent neural network, using 3 feed-forward neural networks (DNN) U, V, W, to map the input sequence $x = [x_1, x_2, ..., x_N]$ to the output $y = [y_1, y_2, ..., y_N]$ using the internal hidden state h^t evovling for each timestep t. h_0 would typically be obtained by a mapping of the whole input sequence x.

Long-Short Term Memory

As shown in Figure 1.7, Long-Short Term Memory (LSTM) cells implement a specific architecture to propagate information along the sequence, with the introducion of a new control state c [14]. Three gates covering the forget, the input, and the output regulate the flow of information from the cell. In particular, the forget gate F decides what information to keep from prior states, by multiplying these values by a factor 1 and discards the rest through a multiplication by a factor close to 0. The input gate I is tasked with creating the new internal state of the cell and what information to store in it. Finally, the output gate O decides what information in the cell should be brought to the output. This selectivty of the LSTM cell to decide what to use from memory, what to keep in memory, and what to output give this architecture a much improved memory for long sequences which result in a much improved efficiency.

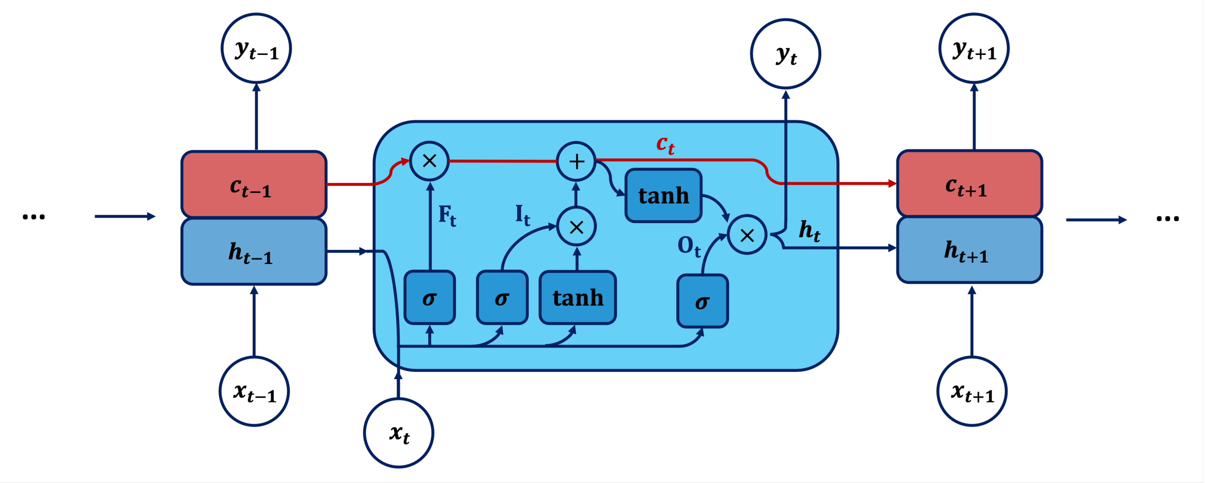


Figure 1.7: An LSTM cell. Arrows and lines that merge imply concatenation of the inputs, the $\times$, $+$, and tanh are element-wise operation and the σ are different layered transformations (1-layer feed-forward network). F_t is the forget gate of the memory cell c, I_t the input gate, and O_t the output gate.

Gated Recurrent Unit

The Gated Recurrent Unit (GRU) is another adventageous design to improve the memory of a recurrent-like network without using as many parameters as an LSTM [15]. There is no output gate and only one internal hidden state h_t is required. This architecture comes in different version, with the fully gated version presented in Figure 1.8. It only requires two gates: the *update gate* Z and the *reset gate* R. The former lets the model decide how much of the past information should be kept and the latter is used to forget past information.

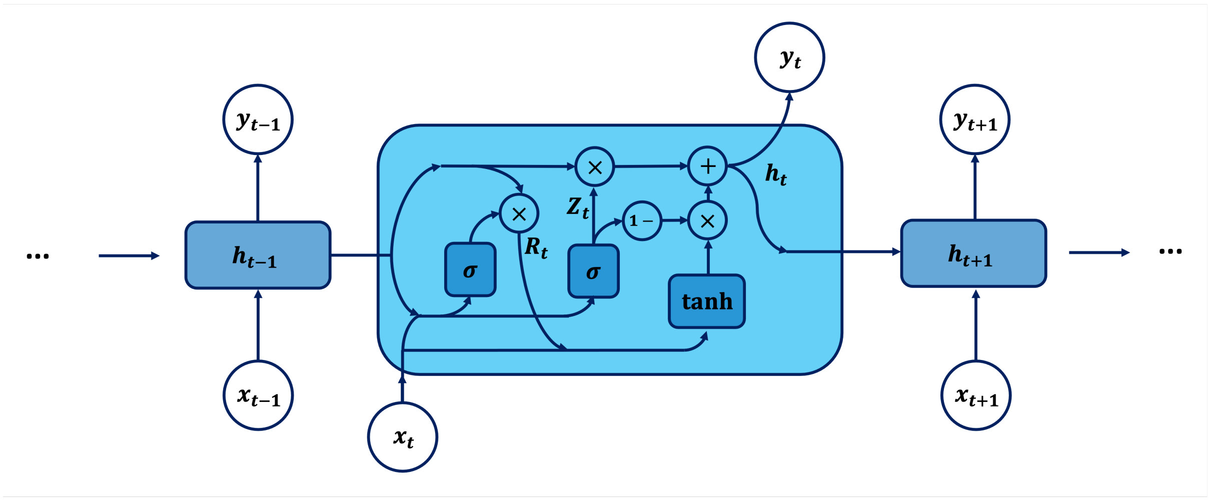


Figure 1.8: A Gated Recurrent Unit cell (fully-gated version). Arrows and lines that merge imply concatenation of the inputs, the $\times$, $+$, tanh, and “1 - ” are element-wise operation - the last one outputting the vector 1 minus its inputs - and the σ are two different layered transformations (1-layer feed-forward network) with sigmoid-like activation function. Z_t is the update gate and R_t the reset gate.

RNNs and their modification have been designed for ordered sequence analysis and have had great results in such settings. Ordered sequences are natural in language analysis, where a sentence such as *“the boy hit the car”* is meaninguflly different to its permutation *“the car hit the boy”*. The choice to sequentially analyse the tokens of a sequence with memory lets RNN-based operates simularly to a Turing Machine, endowing them with the powerful representational flexibility of Universal Turing Machine [16]. A significant drawback however is the impossibility to fully parallelise the analysis of the sequence due to its strict ordering, making RNNs expensive models to train even on highly parallelised hardware. The main motivation behind the Transformer design, introduced in section 1.2.8, was to fix this crucial weakness.

Pros:

- *Sequential Processing:* RNN are designed to handle sequential data, making them suitable for tasks with temporal dependencies.
- *Flexibility:* RNN can operate on input sequences of variable length.
- *Memory:* RNN have a memory mechanism that allows them to retain information about previous inputs.

Cons:

- *Vanishing and Exploding Gradients:* Training deep RNN can suffer from vanishing and exploding gradient problems, affecting the learning of long-term dependencies.
- *Limited Short-Term Memory:* Traditional RNN struggle to capture long-range dependencies due to their limited short-term memory.
- *Complexity:* While LSTM and GRU architectures can mitigate the two points above, the cost is a more complex architecture that is harder to train and requires more resources.
- *Interpretability:* The internal workings of a RNN is challenging to interpret.

1.2.6 Convolutional Neural Networks

Convolutional Neural Networks (CNN) [3, 17] have emerged as a powerful class of deep learning models that are particularly effective in computer vision tasks, including image and video analysis. The architecture of CNNs is roughly based on human visual processing. It consists of convolutional layers - implementing the fundamental convolution operation-, pooling layers, and feed-forward layers (DNN). This architecture, presented in Figure 1.9, enables CNN to automatically learn hierarchical representations of features while respecting properties of image-based data: spaciality (geometrical grounding), locality (neighbourhoods) and spatial symmetry.

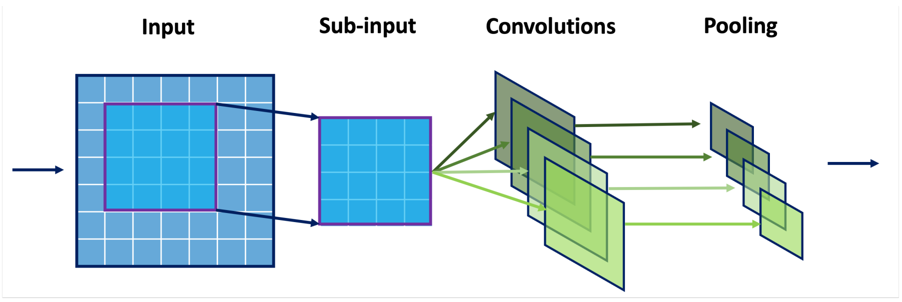


Figure 1.9: A layer of a convolutional neural network, implementing a convolution with 4 kernels followed by a pooling operation. This design can be stacked to create deep architecture, and combined with a feed-forward neural network, after flattening the output at some depth, before reaching the final loss function layer. In this example, the receptive field size is 4 and there are 4 different kernels used to analyse an input image of size 7 $\times$ 7.

The functioning of CNN involves the use of convolutional layers to extract local patterns and features from input data. Convolutional operations are applied to the input data by multiplying entry-by-entry a learnable *kernel* or *filter*, represented by a matrix of weights of smaller dimension than the total image size, with an equal size subpart of the input and applying an activation function. The size of the filter or kernel restricts the processing of the input to a given *receptive field* dimension, and this window is passed over the full input image by moving it by a defined *stride* length. Pooling layers are then used to reduce spatial dimensions and retain important features. This process can be parallelised for an image of multiple channel: e.g., a coloured image would be represented by a tuple of 3 values (for example, using the RGB coding) for each pixel. For classification and regression, a CNN based model would typically stack several convolutional and pooling layers before leading to a fully connected neural network "flattening" the last representation to make predictions. *Flattening* refers to the process of transforming the input image, represented as a matrix $\mathbb{R}^x \times \mathbb{R}^y$, into a vector $\mathbb{R}^{x \times y}$. CNN-based models, such as AlexNet [17] and ResNet [18], have demonstrated state-of-the-art performance in various computer vision tasks.
A main advantage of the convolution operation on an image of size $x \times y$ is the reduction of the number of artificial neurons required to process the image, which helps to regularise the network.

- A DNN given the flatten image would require $x \times y$ neurons.
- A CNN with k kernels of size $\alpha \times \beta$ would require $k \times \alpha \times \beta$ artificial neurons, that would be applied to different subpart of the image.

For example, for an image of size 100 × 100, a DNN would require 10,000 weights while a CNN can process the image with only 25 if a single kernel of size 5 × 5 is used. Typical pooling functions are the *maxPooling* or the *sumPooling*, which, respectively, takes the largest element or the sum in each window of their input, with specific hyperparameters governing the size and the movements of the window.

Pros:

- *Feature Learning:* CNN automatically learn hierarchical representations of features on multi-dimensional data, reducing the need for manual feature engineering.
- *Spatial Hierarchies:* Convolutional and pooling layers enable the model to capture spatial hierarchies in the input data while respecting the properties of images, namely locality: how close pixels are.
- *State-of-the-Art Performance:* CNN have achieved state-of-the-art performance in image classification, object detection, and segmentation tasks.

Cons:

- *Computational Complexity:* Training deep CNN can be computationally intensive, requiring substantial resources.
- *Large Datasets:* CNN often require large labelled datasets for effective training, which may not be available for every application.
- *Interpretability:* The internal workings of CNN can be challenging to interpret, making them somewhat of a "black box".

1.2.7 Graph Neural Networks

In the last few years, Graph Neural Network (GNN) have gained significant attention for their ability to model and analyse complex relationships within graph-structured data [19]. Originally designed for tasks such as node classification and link prediction, GNNs have found applications in diverse domains such as social networks modelling, recommendation systems, and physics, for modelling the dynamic of a N-body system, performing tracks reconstruction, and identifying particles.
GNNs operate on graph-structured data, where nodes or vertices represent entities, and edges represent relationships between these entities. The functioning of GNNs involves iterative aggregation of information from neighbouring nodes and updating of the edges, allowing them to capture local and global structures as defined by the graph. This is achieved through the use of message-passing mechanisms. An interesting feature of graphs is that the input information does not need to be given a rigid structure. Consequently, graph based-methods have a much greater representation power than image- or sequence-based ones. Graphs are in fact able to to represent abritrary relational structures, as defined by the graph through its directional weighted edges [20]. A particular feature arising from this property is that graph are permutation equivariant: the order of nodes can be rearranged without impact.

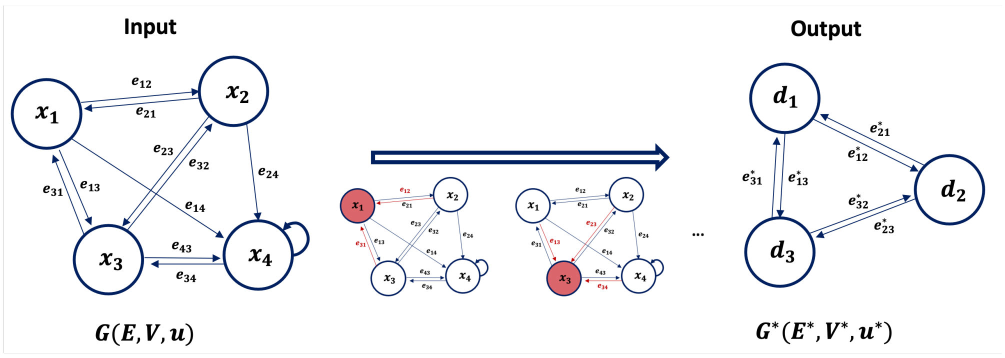


Figure 1.10: A graph neural network update on a directed graph $G(V,E,u)$ with a global representation u, four initial nodes $\in V$ and edges $e_{ij} \in E$ connecting nodes $i \to j$ (in the notation of the text, each node is given a different integer index k and written (e_k, r_k, s_k), with $r_k = j$ and $s_k = i$). By analysing the neighbours of each node, the graph is updated to a new graph $G^*(V^*, E^*, u^*)$.

Typically, a GNN architecture consists of multiple layers of message passing operations. Each layer updates the node representations by aggregating information from neighbouring nodes, as schematised in Figure 1.10. Different architectures implement different update processes for the graph, with popular GNN architectures being Graph Convolutional Networks (GCNs) [21] and Graph Attention Network [22]. In this thesis, the notation adopted is to represent a graph G has a tuple of three entires:

1. $E = \{(e_k, r_k, s_k)\}_{k=1:N^e}$ the set of edges, with each edge having a real vector of features $e_k \in \mathbb{R}^e$ and storing the index of the receiver (sender) as r_k (s_k).
2. $V = \{v\}_{i=1:N^v}$ the set of nodes, each node having a real vector of features $v_i \in \mathbb{R}^v$.
3. u, a global attribute of the graph modelled by a real vector of features $u \in \mathbb{R}^u$.

Algorithm 5 Steps of Computation in a Full Graph Network Block [20]

```
1:  function GraphNetwork($E, V, u$)
2:      for $k \in \{1 \dots N^e\}$ do
3:          $e^*_k \leftarrow \phi^e(e_k, v_{r_k}, v_{s_k}, u)$
4:      end for
5:      for $i \in \{1 \dots N^n\}$ do
6:          Let $E^*_i = \{(e^*_k, r_k, s_k)\}$ for $k = 1 : N^e$ where $r_k = i$
7:          $\overline{e^*_i} \leftarrow \rho^{e \to v}(E^*_i)$
8:          $v^*_i \leftarrow \phi^v(\overline{e^*_i}, v_i, u)$
9:      end for
10:     Let $V^* = \{v^*\}_{i=1}^{N^v}$
11:     Let $E^* = \{(e^*_k, r_k, s_k)\}_{k=1}^{N^e}$
12:     $\overline{e^*} \leftarrow \rho_{e \to u}(E^*)$
13:     $\overline{v^*} \leftarrow \rho_{v \to u}(V^*)$
14:     $u^* \leftarrow \phi_u(\overline{e^*}, \overline{v^*}, u)$
15:     return $(E^*, V^*, u^*)$
16: end function
```

The most general graph update algorithm to describe an update stage of a full GNN block is described in Algorithm 5. Essentially, for a given step the input is a graph $G(E,V,u)$ that is updated into a new graph $G^*(E^*,V^*,u^*)$ by first updating the edges $e \in E$ and then modifying the nodes $v \in V$ and the global representation u. The update rule leverages different neural networks ϕ and aggregation function ρ to update the graph. The aggregation function should accept a variable number of input with permutation invariance to output a single element per group, and is typically implemented to do the sum or max pooling. The update is decomposed into successive updates of:

- The edges, with a DNN ϕ^e mapping each of the input edge, their respective receiver and sender nodes and the global state u to output a new edge feature vector e_k^* for each edge k: $e_k^* = \phi^e(e_k, v_{r_k}, v_{s_k}, u)$. The new edges are stored in a set E^*.
- Before updating a vertex i represented by v_i, the E_i^* updated edges connecting to i ($i == r_k$) are pooled locally over the node as $\overline{e_i^*} = \rho^{e\to v}(E_i^*)$.
- The vertex is then updated with a DNN ϕ^v, mapping the pooled representation of the edges $\overline{e_i^*}$ connecting to the vertex being updated, the input vertex feature v_i and the global representation u to update $v_i \to v_i^* = \phi^v(\overline{e_i^*}, v_i, u)$. The new vertices are stored in a set V^*.
- The set of edges is updated by a global pooling $\overline{E^*} = \rho^{e\to u}(E^*)$.
- The set of vertices is updated by a global pooling $\overline{V^*} = \rho^{v\to u}(V^*)$.
- The global representation is updated by DNN ϕ^u mapping $u^* = \phi^u(\overline{e^*}, \overline{v^*}, u)$, where $\overline{e^*} = \rho^{e\to u}(E^*)$ and $\overline{v^*} = \rho^{v\to u}(V^*)$ are globalyl pooled updated edges ($\overline{E^*}$) and vertices ($\overline{V^*}$).

This formulation of a graph as a message-passing with edges update device is the most complete architecture of a GNN. The design is however flexible: for example, RNNs or CNNs can be used instead of DNN. Furthemore, many specialisations of the structures exist to reduce the degree of complexity of the model and avoid overfitting or converge issues, as listed in Figure 1.11. A notable example for this thesis is the Deep Set architecture [23], designed to runs specifically on sets where the ordering does not matter. It essentially simplifies the graph network by dropping altogether the edges and considering instead a fully connected graph with static edges, with an update of the global representation only based on pooled node information:

$$v_i^* = \phi^v(\overline{e_i^*}, v_i, u) = \phi^v(v_i, u),$$

$$\overline{V^*} = \rho^{v\to u} = \sum_i v_i^*,$$

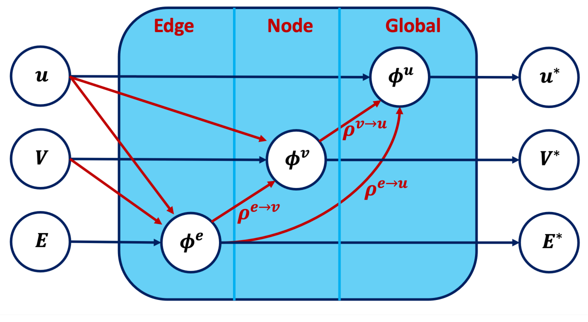
$$u^* = \phi^u(\overline{e^*}, \overline{v^*}, u) = \phi^u(\overline{v^*}, u).$$

This is somewhat simular to PointNet, a GNN designed to analyse sets of 3D points, that uses an analoguous update with max-aggregation instead of sum pooling after updating the nodes in two steps [24].

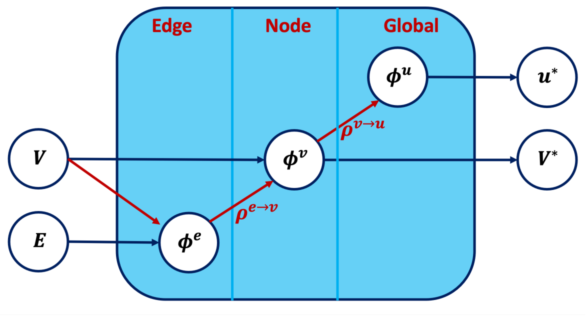
A different approach introduced in [25] defines the non-local neural network, unifying different types of *attention*-based architecture. Attention is an essential feature of modern deep learning: it refers to how an element is given a weighted version of the inputs, with weights standing for the degree of attention to be given to each different part of the input. This concept is not restricted to GNN but is easily encapsulated in this formalism. As will be shown in the next section, the Transformer is a special case of this non-local NN family. In this section only the Graph Attention Network (GAT) is introduced for the sake of conciseness [22]. GAT introduces the attention mechanism by having a learnable weighting of the neighbour of the node being updated. When updating node v_i, a score is computed for each of the connected neigbour of v_i by a NN mapping:

$$e(v_i, v_j) = \phi(v_i, v_j) = a^T \text{ leakyReLU}([Wv_i, Wv_j]),$$

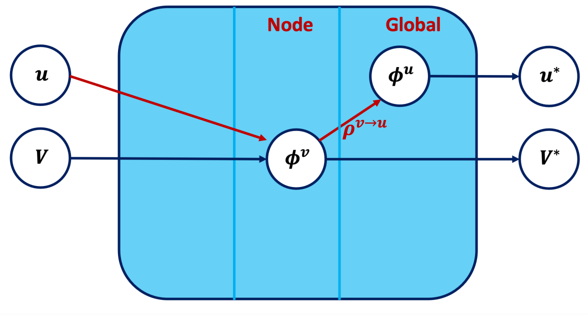
where leakyReLU is the modification of the ReLU with negative leakage, the nodes $v_i, v_j \in \mathbb{R}^d$ with j connected to i, and the operation implements an embedding of the two nodes to a dimension d' with



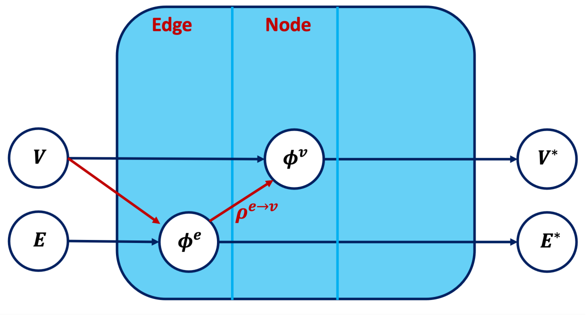
(a) Full GNN.



(b) Message-passing GNN.



(c) Deep Set.



(d) Non-local NN.

Figure 1.11: Some of the different types of GNN update rules, defining different GNN architecture [20].

two learnable parameters: $a \in \mathbb{R}^{2d'}$ and $W \in \mathbb{R}^{d' \times d}$. The operation $[,]$ stands for concatenation of the elements. These scores are then combined for each node i over its neighbours $\{j\}$ to give attention scores α_{ij}:

$$\alpha_{ij} = \text{softmax}_j(e(v_i, v_j)) = \frac{\exp(e(v_i, v_j))}{\sum_{j' \in \text{neighours of i}} e(v_i, v_{j'})}.$$

The final step is to leverage these attention weights when updating each node v_i:

$$v_i^* = \sigma\left(\sum_j \alpha_{ij}.W^v v_j\right),$$

where the sum over j is taken over neighbouring nodes of i, σ is any activation function and W^v is another matrix of learnable parameters.

Pros:

- *Modeling Graph Structure:* GNNs naturally handle graph-structured data, making them well-suited for tasks involving relationships between entities.
- *Transferability:* Pre-trained GNN models on one graph can be fine-tuned for related tasks on another graph.
- *State-of-the-Art Performance:* GNNs have achieved state-of-the-art results in various graph-related tasks, including node classification and link prediction.

Cons:

- *Computational Complexity:* Training GNNs can be computationally expensive, particularly for large graphs.
- *Limited Global Context:* Some GNN architectures may struggle to capture long-range dependencies in graphs, limiting their ability to consider global context.
- *Interpretability:* Similar to other deep learning models, GNN lack interpretability, making it challenging to understand the learned representations.

1.2.8 The rise of the Transformers

The Transformer architecture, introduced in 2017 [26], has become a foundational model for NLP tasks. It has significantly impacted the field, enabling the development of state-of-the-art models such as BERT [27] and GPT [28]. Note that the transformer is also spearheading a revolution in computer vision tasks thanks to the generalisation of the architecture into the Vision Transformer (ViT) [29].

The Transformer architecture is based on the mechanism of self-attention introduced in the last section. As mentionned in section 1.2.5 on RNN, it abandons sequential processing and adopts a parallelised approach, allowing for efficient computation on parallelisable hardware. The key components of the Transformer are the self-attention mechanism and position-wise feedforward networks. The self-attention mechanism allows the model to weigh the importance of different words or tokens in a sequence when making predictions for a specific token. This mechanism enables the model to capture long-range dependencies in the input data without the added complexity of LSTM or GRU. Strictly speaking, the input of a transformer is a sequence that is not ordered. In NLP, the importance of ordering is built-in using position-wise embedding to let the network decypher the index of the token in the input sequence. For the computer vision case, the Vision Transformer splits an input image x into patches of fixed size, that are flattened into a vector and mapped with a learnable positional embedding.

As presented in Figure 1.12, the general Transformer architecture consists of an encoder and a decoder, the main feature of auto-encoders model. The decoder works in an autoregressive way, combining the current output with an internal representation h built by the encoder to generate new output tokens y. Both the encoder and decoder are composed of multiple layers, each containing a multi-head self-attention mechanism and position-wise feedforward networks (DNN). The decoder is further endowed with a masked attention layer, for the output to compute self-attention with information accessible prior to the token's position. The attention mechanism allows the model to focus on different parts of the input sequence, while the feedforward networks provide additional non-linear transformations. Residual connexions are added to let the gradients propagate efficiently in depth and layer normalisation is used after each block to avoid vanishing or exploding gradients and improve training speed [30]. This type of normalisation scales each activation (each neuron) by substracting the empirical mean and dividing by the standard deviation per datapoint.
The attention mechanism maps the queries and a set of key-value pairs to an output as defined in Equation 1.15 and schematised in Figure 1.13a, with query $Q \in \mathbb{R}^{d_q \times d_k}$, key $K \in \mathbb{R}^{d_k \times d_v}$, and value $V \in \mathbb{R}^{d_v}$ and the output is a vector $\mathbb{R}^{d_q \times d_v}$. This combines d_q different queries of the d_k keys mapping to d_v values. Equation 1.15 is weighted sum of the values, based on a compatibilty function established by comparing the queries and keys:

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{Q^T.K}{\sqrt{d_k}}\right)V, \tag{1.15}$$

where the scaling by $\sqrt{d_k}$ is implemented to reduce the magnitude of the dot product $Q^T K$ and avoid landing in regions of saturation of the following softmax. This scaled dot-product attentino mechanism leverages the extensive optimisation of the associated matrix multiplication, making this operation less time and memory demanding than using a DNN mapping to compute the attention - a

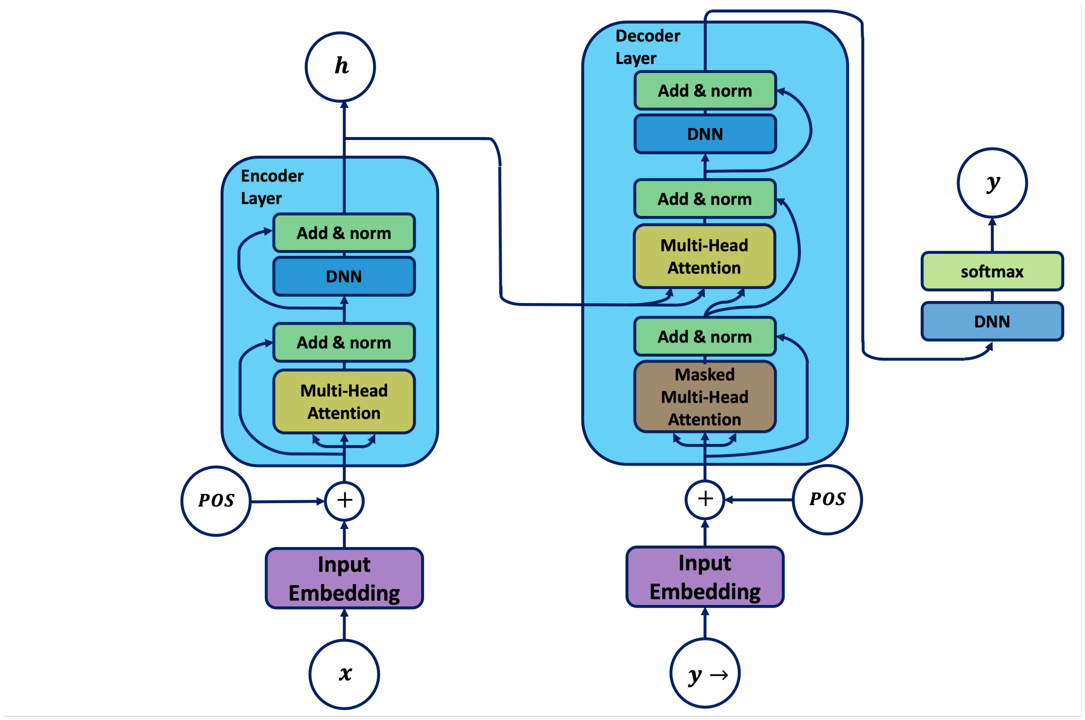


Figure 1.12: The full transformer architecture, combining an encoder and a decoder each made of an arbitrary number of layers. The input x is first embedded with a dedicated mapping which can, when order matters, be supplemented with positional embedding. The encoder generates an internal representation h that is passed to the decoder. This last component produces the next output using the internal representation y and the output shifted to the right - to force output token to only access prior information.

technique referred to as *additive attention* [31]. As shown in Figure 1.13b, multi-head attention runs the dot-product self-attention in parallel for h different heads, each head h_i implementing a separate projection from the input $x \in \mathbb{R}^{N\times d}$ to Q, K, V with linear transformation of respective weights $W_i^Q \in \mathbb{R}^{d\times d_k}$, $W_i^K \in \mathbb{R}^{d\times d_k}$, and $W_i^Q \in \mathbb{R}^{d\times d_v}$, where N is the length of the sequence, d is the model dimension and h the number of heads:

$$\begin{aligned} Q &= xW_i^Q, \\ K &= xW_i^K, \\ V &= xW_i^V, \\ h_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V). \end{aligned}$$

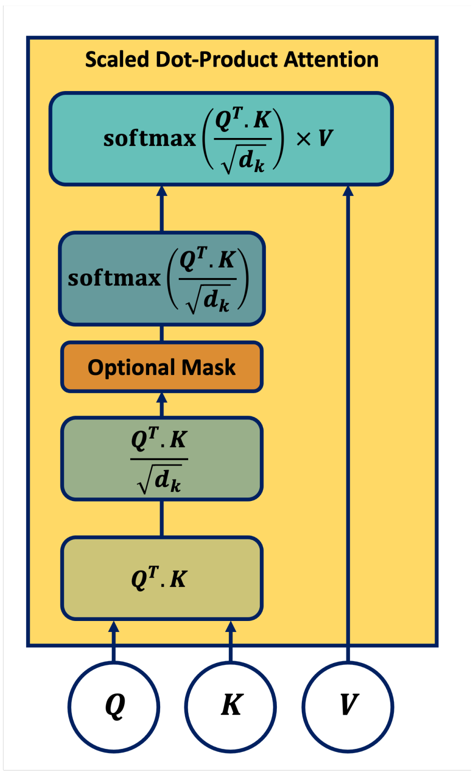
The multi-head module then concatenates the h different heads outputs and applies another linear tranformation of parameters $W^O \in \mathbb{R}^{hd_v\times d}$:

$$\text{Multi-Head Attention}(Q,K,V) = \text{Concatenate}(h_1, ...h_h)W^o.$$

Self-attention is a special case in which the multi-head does not receive a tuple (Q,K,V) but a single input $x \in \mathbb{R}^{N\times d_v}$ that is mapped out to the tuple.

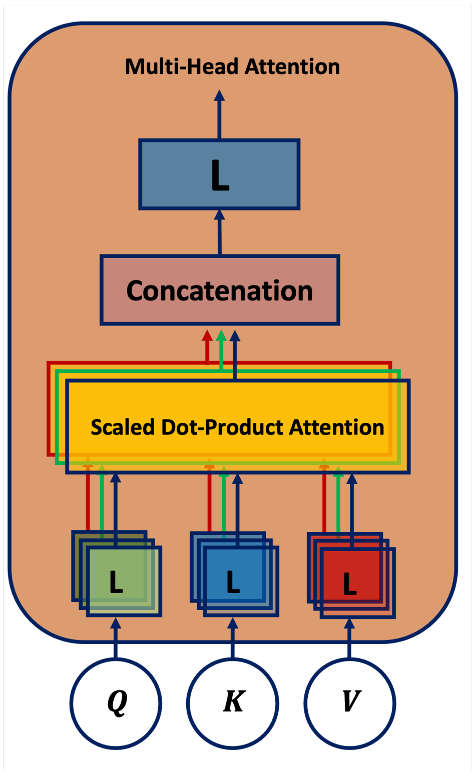
Pros:

- *Parallelisation:* The architecture allows for efficient parallelisation of training, speeding up the learning process.
- *Capturing Dependencies:* The self-attention mechanism enables the model to capture long-range dependencies in sequences.



Scaled Dot-Product Attention
$\text{softmax}\left(\frac{Q^T.K}{\sqrt{d_k}}\right) \times V$
$\text{softmax}\left(\frac{Q^T.K}{\sqrt{d_k}}\right)$
Optional Mask
$\frac{Q^T.K}{\sqrt{d_k}}$
$Q^T.K$
Q
K
V

(a) Scaled dot-product attention with optional masking.



(b) Multi-head attention module, where L stands for a linear transformation of the input.

Figure 1.13: Multi-head attention mechanism in a transformer. The core operation is an optionally masked scalled dot-product of the queries Q, keys K, and values V. A head consists in three (optionally different) linear projection of the tuple (Q,K,V), each leading to a separate scaled dot-product. The multi-head modules then concatenates all the different head results and finishes with another linear transformation.

- *Versatility:* The Transformer architecture has been successfully applied to various NLP tasks - including machine translation, text summarization, and language modeling - and computer vision tasks.

Cons:

- *Computational Complexity:* Training large Transformer models can be computationally expensive, requiring powerful hardware.
- *Interpretability:* The attention mechanism, while effective, can be challenging to interpret, making the model somewhat of a "black box."
- *Data Dependency:* Transformer models may require large amounts of data for effective training, which may limit their application in domains with limited labelled data.

1.3 Training and Optimising Deep Learning Models

Training and optimising neural network models involve a combination of selecting appropriate architectures, fine-tuning hyperparameters - which cannot be learnt by backpropagation-, and employing acceleration techniques to improve efficiency and convergence. In this section, key aspects of the training process are explored, and acceleration techniques commonly used in practice are discussed.

1.3.1 Training Algorithms

When optimising the learnable parameters of a model, different strategies can be deployed to update the weights. All mechanisms are refinement on the base gradient descent rule of Equation 1.13, and each method derived has different *pros and cons*. The two main approaches are:

- Stochastic Gradient Descent (SGD): the update rule is the same as that of Equation 1.13, with the only difference being in how the gradient of the weights is computed. Instead of deriving it for the whole dataset (full-batch), the expectation over a random sub-batch is taken, hence the stochastic behaviour:

$$\nabla w = \frac{1}{b}\sum_{s=1}^{b} \nabla w_s.$$

  A common observation is that for sub-batches - from now-on referred to as just *batch* - of sufficient size, the stasticial estimator of the gradient based on the batch is unbiased. This greatly accelerates the time it takes to compute the gradient and naturally splits the loop over the dataset into different iterations called *steps*, at which a batch is passed throught the network, a benefitial features in the cases of large datasets that would not fit in memory.

- Adam [32] Adam is an algorithm published in 2014 leveraging an adaptive moment estimation approach. The moment in this sense is analoguous to the physical moment and encapsulates the dynamic of the optimisation as stated by the gradient. The idea is that, for larger gradients, the slope is steep and can be quickly traversed, so that any slowing down due to a changing curvature of the objective function landscape could have a mitigated effect on the speed thanks to the inertia of the momentum. This is implemented as an exponentially decaying moving average: the moment m_w^t of weight w at step t is updated with a factor $\beta_1 \in [0,1[$ such that:

$$m_t \leftarrow \beta_1 m_w^{t-1} + (1-\beta_1)\nabla_w \mathscr{L}^t,$$

  where the previous contribution are successively mutliplied by the gradient forgetting factor β_1. Additionally, another element is taken into account in the gradient descent rule: the second moment $(\nabla_w \mathscr{L}^t)^2$. This tracks how steep the gradient is and, by multiplying the gradient update by a term inversely proportional to the second moment, accelerates the gradient updates in "flatter" regions of the objective landscape with the term:

$$v_t \leftarrow \beta_2 v_w^{t-1} + (1-\beta_2)(\nabla_w \mathscr{L}^{\sqcup})^2.$$

  To avoiding biasing the gradient update, both the momentum (first moment) and the second moment are corrected with

$$\hat{m}_w^t \leftarrow \frac{m_w^t}{1-\beta_1},$$

  and

$$\hat{v}_w^t \leftarrow \frac{v_w^t}{1-\beta_2}.$$

  The two contributions are then combined into a single gradient descent step following equation **??**

$$w^t \leftarrow w^{t-1} - lr \times \frac{\hat{m}_w^t}{\sqrt{\hat{v}_w^t} + \varepsilon}, \tag{1.16}$$

  where ε is added for numerical stability.

A key hyperparameter in the any gradient descent algorithm is the learning rate lr. There is no evident choice for this parameter and the most suitable values are very much derived on a case-by-case approach. A useful technique to let the training process converge to a good minimum of the loss function and avoid unsuitable local minima is to adapt a *learning rate schedule:* the learning rate is modified throughout training to resolve different part of the loss function. Initially, having a relatively larger lr allows the model to quickly update its weights in the direction of the minimum. If the rate is kept to high, the weights will not be able to approach the minimum and will "bounce' around the target. In order to correct this, the scheduler reduces the lr so that smaller steps can be taken later in the training to approach the chosen optimum. At the beginning, the rate is typically not set to its maximum to start the gradient process in a valley of interest. An equivalent change is to modify the batch size while keeping the lr fixed [33]. This has a regularising effect on the gradient: small batch sizes capture large variances and let the optimisation make drastic changes of orientation in the optimising function landscape, thereby avoid unsatisfactory local minima. Larger batch sizes regularise the direction of descent, thereby offering a lower variance but potentially biased estimates towards a minimum. Some method, such as Adam, have specific hyperparameters that should also be optimised with the procedure described later in this chapter.

1.3.2 Regularisation

Regularisation techniques are applied in the architecture and training procudure to prevent overfitting. Common methods include *dropout*, which randomly drops connexions during training with Bernouilli probability distribution of parameter p, and L2 (L1) regularisation, which penalises large weights proportionaly to a penalisation parameter λ times the sum of the squared (absolute value) of the weights. Both p and λ require careful optimisation as regularising the model can introduce bias and limit the overall performance. Batch normalisation is a technique that normalises the inputs of a layer over the batch, reducing internal covariate shift. It helps stabilise and accelerate the training process. This is distinct from the layer normalisation used in the Transformer architecture of section 1.2.8 as the normalisation is carried over the batch samples rather than the activation.

1.3.3 Hyperparameters Optimisation

There are several characteristics of the network that need to be optimised. Mainly:

- **Architecture Selection:** choosing the right architecture is crucial for the success of a NN. Factors to consider include the complexity of the task, the nature of the data, and the desired trade-off between model complexity and interpretability. Limits in computing power should be factored in. Elements of the architecture include the type of ML chosen (BDT, DNN, CNN, RNN, Transformer, ...), the choice of activation functions (ReLU, tanh, ...), and the number of layers and nodes.

- **Hyperparameter Tuning:** optimising hyperparameters - parameters that cannot be optimising through backpropagation of the gradients of the loss - is essential for achieving good model performance. Key hyperparameters include learning rate parameters, batch size, and regularisation parameters.

The optimisation process for both hyperparameter tuning and architecture selection is expensive: it requires training and testing models with different combinations of hyperparameters or model architecture to uncover the best peforming options. Techniques such as grid search, random search, and Bayesian optimisation can be employed to efficiently explore the hyperparameter space. Architecture search is usually performed by trials and attempts, and the literature offers insights and guidance into what models might best perform in specific situations.

1.3.4 Acceleration Techniques

Training an ML model is often a computationally demanding tasks that should be carried out more effectively on specifically designed hardware and with some tricks in the process.

- **Parallel Data Loading:** loading data in parallel can significantly speed up the training process. Libraries like TensorFlow and PyTorch provide functionalities for efficient parallel data loading. Instead of preparing a single batch, multiple batches can be loaded by different processing units to avoid this step becoming a bottleneck in performance.
- **Early Stopping:** to prevent overfitting and save computation time, early stopping involves monitoring the model's performance on a validation set and halting training when performance ceases to improve.
- **Hardware Accelerators:** specialised hardware accelerators can further accelerate training. The Graphics Processing Unit (GPU) architecture is design to perform simple mathematical operation in parallel. This is precisely the needs of NN optimisation, as each step consist in a matrix multiplication. Utilising GPU for training and inference can lead to substantial speedup compared to Core Processing Unit (CPU)-based training. Other types of hardware optimised for DL include the Tensor Processing Units (TPU) and the Neural Processing Units (NPU).
- **Transfer Learning:** transfer learning leverages pre-trained models on large datasets. Fine-tuning these models on specific tasks can significantly reduce the required training time and data. This is becoming increasingly fashionable, as larger *foundational* models trained on multiple tasks with large datasets can then be applied to a specific task, with the pre-trained weights either kept fixed or let free to be modified to optimise the new task. New modules can also be added on top the foundational model to specialise it to a specific use-case. Such large foundational models are already available in NLP (e.g., the Mistral 7B [34] - a 7 billion parameters transformer capable of analysing English sentences and code) and computer vision (for example, the Florence-2 model by Microsoft [35] - built on a Transformer).

Training and optimizing neural network models involve a combination of careful architectural choices, hyperparameter tuning, and the use of acceleration techniques. The selection of appropriate techniques depends on the specific requirements of the task, available resources, and the desired trade-offs between training time and model performance.

BIBLIOGRAPHY

[1] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012. ISBN: 0262018020.

[2] Dan Cireşan, Ueli Meier, and Juergen Schmidhuber. "Multi-column Deep Neural Networks for Image Classification". In: *Proceedings / CVPR, IEEE Computer Society Conference on Computer Vision and Pattern Recognition. IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (Feb. 2012). DOI: 10.1109/CVPR.2012.6248110.

[3] Yann LeCun et al. "Handwritten Digit Recognition with a Back-Propagation Network". In: *Advances in Neural Information Processing Systems*. Ed. by D. Touretzky. Vol. 2. Morgan-Kaufmann, 1989. URL: %5Curl%7Bhttps://proceedings.neurips.cc/paper_files/paper/1989/file/53c3bce66e43be4f209556518c2fcb54-Paper.pdf%7D.

[4] Leo Breiman. "Bagging predictors". In: *Machine Learning* 24.2 (1996), pp. 123–140. DOI: 10.1007/BF00058655. URL: %5Curl%7Bhttps://doi.org/10.1007/BF00058655%7D.

[5] Yoav Freund and Robert E Schapire. "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting". In: *Journal of Computer and System Sciences* 55.1 (1997), pp. 119–139. ISSN: 0022-0000. DOI: https://doi.org/10.1006/jcss.1997.1504. URL: https://www.sciencedirect.com/science/article/pii/S002200009791504X.

[6] Jerome H. Friedman. "Greedy function approximation: A gradient boosting machine." In: *The Annals of Statistics* 29.5 (2001), pp. 1189–1232. DOI: 10.1214/aos/1013203451. URL: https://doi.org/10.1214/aos/1013203451.

[7] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological Review* 65.6 (1958), pp. 386–408. ISSN: 0033-295X. DOI: 10.1037/h0042519. URL: %5Curl%7Bhttp://dx.doi.org/10.1037/h0042519%7D.

[8] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2.4 (1989), pp. 303–314. DOI: 10.1007/BF02551274. URL: %5Curl%7Bhttps://doi.org/10.1007/BF02551274%7D.

[9] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: \url{https://doi.org/10.1016/0893-6080(89)90020-8}. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[10] Zhou Lu et al. “The Expressive Power of Neural Networks: A View from the Width”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS’17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6232–6240. ISBN: 9781510860964.

[11] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: %5Curl%7Bhttp://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf%7D.

[12] TensorFlow Collaboration. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Tech. rep. 2015. URL: %7Bhttps://www.tensorflow.org/%7D.

[13] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323.6088 (1986), pp. 533–536. DOI: 10.1038/323533a0. URL: %5Curl%7Bhttps://doi.org/10.1038/323533a0%7D.

[14] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-term Memory”. In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[15] Kyunghyun Cho et al. “Learning phrase representations using RNN encoder-decoder for statistical machine translation”. In: *Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*. 2014.

[16] Stephen Chung and Hava Siegelmann. “Turing Completeness of Bounded-Precision Recurrent Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by M. Ranzato et al. Vol. 34. Curran Associates, Inc., 2021, pp. 28431–28441. URL: https://proceedings.neurips.cc/paper_files/paper/2021/file/ef452c63f81d0105dd4486f775adec81-Paper.pdf.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[18] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[19] Franco Scarselli et al. “The Graph Neural Network Model”. In: *IEEE Transactions on Neural Networks* 20.1 (2009), pp. 61–80. DOI: 10.1109/TNN.2008.2005605.

[20] Peter Battaglia et al. “Relational inductive biases, deep learning, and graph networks”. In: *arXiv* (2018). URL: https://arxiv.org/pdf/1806.01261.pdf.

[21] Thomas N. Kipf and Max Welling. “Semi-Supervised Classification with Graph Convolutional Networks”. In: *Proceedings of the 5th International Conference on Learning Representations* (ICLR). ICLR ’17. Palais des Congres Neptune, Toulon, France, 2017. URL: https://openreview.net/forum?id=SJU4ayYgl.

[22] Petar Veličković et al. “Graph Attention Networks”. In: *International Conference on Learning Representations* (2018). URL: https://openreview.net/forum?id=rJXMpikCZ.

[23] Manzil Zaheer et al. “Deep Sets”. In: *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/f22e4747da1aa27e363d86d40ff442fe-Paper.pdf.

[24] Charles R. Qi et al. “PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. July 2017.

[25] Xiaolong Wang et al. “Non-local Neural Networks”. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 7794–7803. DOI: 10.1109/CVPR.2018.00813.

[26] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa
Paper.pdf.

[27] Jacob Devlin et al. “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)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://aclanthology.org/N19-1423.

[28] Alec Radford et al. “Improving language understanding by generative pre-training”. In: (2018).

[29] Alexey Dosovitskiy et al. “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”. In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=YicbFdNTTy.

[30] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. cite arxiv:1607.06450. 2016. URL: http://arxiv.org/abs/1607.06450.

[31] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate”. In: *CoRR* abs/1409.0473 (2014). URL: https://api.semanticscholar.org/CorpusID:11212020.

[32] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. 2015. URL: http://arxiv.org/abs/1412.6980.

[33] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. “Don’t Decay the Learning Rate, Increase the Batch Size”. In: cite arxiv:1711.00489Comment: 11 pages, 7 figures. 2017. URL: https://openreview.net/pdf?id=B1Yy1BxCZ.

[34] Albert Q. Jiang et al. *Mistral 7B*. 2023. arXiv: 2310.06825 [cs.CL].

[35] Bin Xiao et al. *Florence-2: Advancing a Unified Representation for a Variety of Vision Tasks*. 2023. arXiv: 2311.06242 [cs.CV].