2. Methods

2.1. An overview on Neural Networks

2.1.1. Core Structure

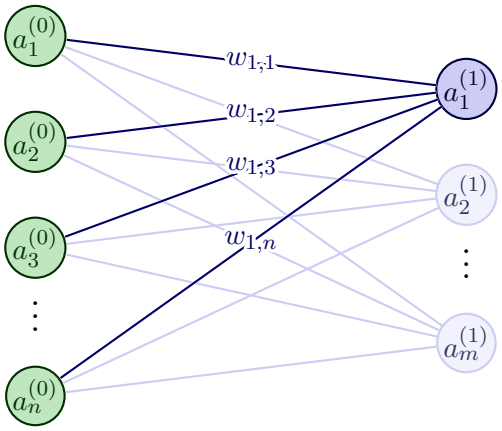
With the term *Artificial Neural Network* (NN), we refer to a computing system inspired by the biological neural networks constituting animal brains. These tools were first introduced in the second half of the XX century, and nowadays they are witnessing an amazing popularity.
They fall into the framework of *Deep Learning* (DL), which, in turn, is one of the specialization of *Machine Learning* (ML), and they enable to process the information in a surprisingly complete way. The main strength of NNs is that, while classical ML predictions (e.g. linear regression) work well only when the features (i.e. the inputs or a hand-made combination of them) are good, they can directly learn the features from the data and therefore fall in the *Feature Learning*[1] context.

The neuron

In order to understand how these tools work, we first introduce their basis components: **neurons**.
They consists in processing units, which:

1. take as input a finite number of signals a_j from input data or other previous neurons outputs;
2. compute their weighted sum with weights $w_{i,j}$, where j refers to the signal and i identifies the neuron;
3. subtract a threshold by adding a bias term b_i, which is a special weight relative to the neuron itself;
4. produce one single output z_i and apply a non-linear activation function ϕ obtaining a_i.

It is fundamental that the activation function ϕ is *non-linear*; else, the whole Neural Network would be only a highly factorized linear function of the input data.



$$\begin{pmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_m^{(1)} \end{pmatrix} = \sigma \left[\begin{pmatrix} w_{1,1}^{(0)} & w_{1,2}^{(0)} & \dots & w_{1,n}^{(0)} \\ w_{2,1}^{(0)} & w_{2,2}^{(0)} & \dots & w_{2,n}^{(0)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1}^{(0)} & w_{m,2}^{(0)} & \dots & w_{m,n}^{(0)} \end{pmatrix} \begin{pmatrix} a_1^{(0)} \\ a_2^{(0)} \\ \vdots \\ a_n^{(0)} \end{pmatrix} + \begin{pmatrix} b_1^{(0)} \\ b_2^{(0)} \\ \vdots \\ b_m^{(0)} \end{pmatrix} \right]$$

$$\mathbf{a}_1 = \sigma\left(\mathbf{W}_0^T \mathbf{a}_0 + \mathbf{b}_0\right), \quad \mathbf{W}_0 \in \mathbb{R}^{n \times m}$$

Figure 1: The neuron, basic component of a Neural Network.

Fully Connected Neural Networks

Neurons are stacked in layers, and in *Fully Connected Neural Networks (FCNN)*, as the one involved in our project, each neuron is connected to all the neurons of the next and the previous layer.
Now, let us define the structure of a Neural Network (NN), which enables the understanding of the already described process of features learning from data. As Figure 2 shows, the three main sections are:

- **Input Layer**, receiving N_i inputs. Input data can be, for example, domains, images, time-series ...;
- **Hidden Layers**: $L-1$ layers, each one with N_h nodes. Their task is to find suitable features;
- **Output Layer**, depending on the desired task (e.g. regression, classification ...), returns N_o output.

Using this notation, the output of the neuron i in layer l, which depends only on the activations $\{a_j^{(l-1)}\}_{j=1}^m$ of the previous layers, is:

$$a_i^{(l)} = \phi\left(\sum_{j=1}^{N_{l-1}} w_{j,i} a_j^{(l-1)} + b_i^{(l)}\right) \tag{1}$$

[1] In Machine Learning, Feature Learning or Representation Learning is a set of techniques that allows a system to automatically discover the representations needed for feature detection or classification from raw data. This replaces manual feature engineering and allows a machine to both learn the features and use them to perform a specific task.

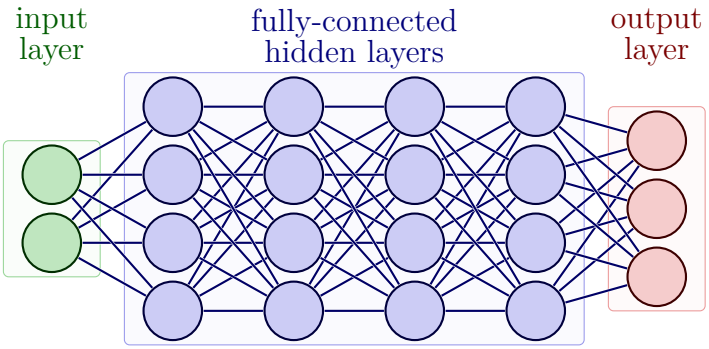


Figure 2: General structure of a Neural Network

Activation Functions

In this section we present some common choices for the activation function ϕ:

- **Sigmoid:** this function is smooth everywhere, but it presents a weak point: $|\sigma'(x)| \ll 1$ for $|x| \gg 1$, therefore the gradient is *vanishing*, and the learning can be slow for deep NNs.
- **Hyperbolic Tangent:** balanced version of the Sigmoid, which is differentiable and has a similar $S-$shape. It has the advantage to push the input values to 1 and -1 instead of 1 and 0.
- **Rectified Linear Unit (ReLU):** in this case, the gradient is not vanishing for $x > 0$, since the derivative is 1, but the function is not differentiable at $x = 0$ and for $x < 0$ the derivative is 0.
- **Leaky ReLU:** this slightly modified version of ReLU solves the issue of the $0-$gradient for $x < 0$.
- **Swish:** another variation on the ReLU, which, according to experiments, tends to work better than ReLU on deeper models across a number of challenging datasets.

Activation	Analytical Expression	Derivative of Activation
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	$\sigma'(x) = \sigma(x)(1-\sigma(x))$
Hyperbolic Tangent	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$	$\tanh'(x) = 1 - \tanh^2(x))$
ReLU	$\mathrm{ReLU}(x) = \max\{0, x\}$	$\mathrm{ReLU}'(x) = \mathbb{1}_{\mathbb{R}_+}(x)$
Leaky ReLU	$\mathrm{LR}(x) = \max\{\alpha x, x\}$	$\mathrm{RL}'(x) = \alpha + (1-\alpha)\mathbb{1}_{\mathbb{R}_+}(x)$
Swish	$\mathrm{S}(x) = x\sigma(x)$	$S'(x) = S(x) + \sigma(x)(1 - S(x))$

Table 1: Common Activation Functions for Neural Networks and their derivatives

Structure of our Neural Network

Reproducing the architecture of the Neural Network proposed in [11], in the project we built a *feed-forward fully connected* Neural Network with 2 hidden layers having the *same number of nodes* (usually 16 or 50), as sketched in Figure 3. The activation function chosen for hidden layers can be specified by the user; in the test cases proposed, we selected the *swish function*.

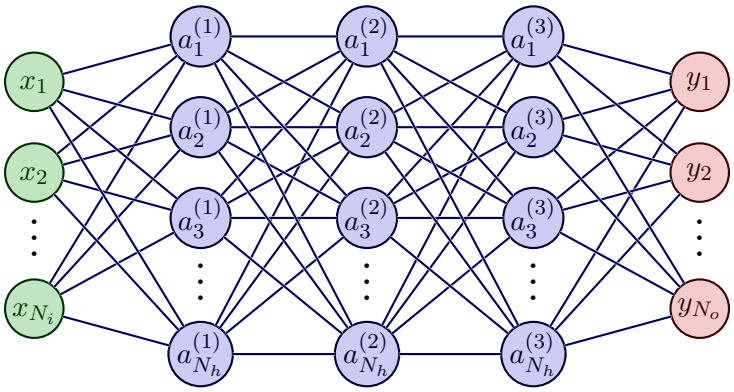


Figure 3: Architecture of a neural network with N_i input, 3 hidden layers and N_o output.

2.1.2. Backpropagation

In Machine Learning, the algorithm through which the NN learns the right weights is generally the so-called *backpropagation*. First of all, let us define the *loss* function $\mathcal{L}(f)$, which provides a quantitative measure of the performance of the NN by expressing the distance between the output estimated by the model $\mathbf{y}^* = f(\mathbf{x})$ and its true value $\mathbf{y}$ for all the data. The optimal solution is obtained by minimizing the Loss function, with respect to all weights and biases:

$$w_{i,j}^{*,(l)}, b_i^{*,(l)} = \underset{w_{i,j}^{(l)}, b_i^{(l)}}{\operatorname{argmin}} \mathcal{L}(f) \tag{2}$$

To perform this task, we have first to choose the *optimizer*, i.e. the optimization algorithm.
There exist several possibilities, with different memory consumption and time needed to reach convergence.
Whatever algorithm we choose, we need the computation of $\nabla\mathcal{L}$, in particular:

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} = \left(\frac{\partial \mathcal{L}_n}{\partial \mathbf{W}_l}\right)_{i,j} \qquad \frac{\partial \mathcal{L}_n}{\partial b_i^{(l)}} = \left(\frac{\partial \mathcal{L}_n}{\partial \mathbf{b}_l}\right)_i \qquad \forall (i,j,l) \tag{3}$$

This means that the number of derivatives that must be computed is of the order of $\mathcal{O}(K^2 L)$, supposing that we have K nodes in each of the L layers for the sake of simplicity; therefore, finding an efficient manner to compute the derivatives jointly becomes fundamental.

General Notations

To illustrate the backpropagation algorithm, let us introduce the following notations:

- **Weight matrices:** $\mathbf{W}_l$, where $\mathbf{W}_1 \in \mathbb{R}^{N_i \times N_h}$, $\mathbf{W}_l \in \mathbb{R}^{N_h \times N_h}$ $\forall\ 2 \leq l \leq L-1$, $\mathbf{W}_L \in \mathbb{R}^{N_h \times N_o}$.
- **Bias vectors:** $\mathbf{b}_l$, where $\mathbf{b}_l \in \mathbb{R}^{N_h}$ $\forall\ 1 \leq l \leq L$, $\mathbf{b}_L \in \mathbb{R}^{N_o}$.
- **Trainable Parameters**: $\boldsymbol{\theta}$ usually used to indicate weights and biases of all layers together.

With this notation,

$$\begin{aligned}
\mathbf{a}_1 &= f^{(1)}(\mathbf{a}_0) = \Phi(\mathbf{z}_0) = \Phi(\mathbf{W}_1^T \mathbf{a}_0 + \mathbf{b}_1) \\
\mathbf{a}_2 &= f^{(2)}(\mathbf{a}_1) = \Phi(\mathbf{z}_1) = \Phi(\mathbf{W}_2^T \mathbf{a}_1 + \mathbf{b}_2) \\
&\vdots \\
\mathbf{a}_L &= f^{(L)}(\mathbf{a}_{L-1}) = \Phi(\mathbf{z}_{L-1}) = \Phi(\mathbf{W}_L^T \mathbf{a}_{L-1} + \mathbf{b}_L)
\end{aligned}$$

Therefore, writing all in a more compact form, the output of the Neural Network is:

$$\mathbf{y}^* = f(\mathbf{x}) \text{ with } f = f^{(L)} \circ f^{(L-1)} \circ \cdots \circ f^{(2)} \circ f^{(1)} \text{ and } \mathbf{x} = \mathbf{a}_0,\ \mathbf{y}^* = \mathbf{a}_L \tag{4}$$

Loss Functions

All the test cases presented in this work fall into the regression framework, hence our dataset generally consist in a sequence of N couples of inputs $\{\mathbf{x}\}_{n=1}^N$ and targets $\{\mathbf{y}\}_{n=1}^N$ vectors.
The most common used loss function for regression problems is the *Mean Squared Error (MSE)*, defined as:

$$\text{MSE} = \frac{1}{N} \sum_{n=1}^{N} (\mathbf{y}_n - \mathbf{y}_n^*)^2$$

Using the MSE as *Loss Function* and exploiting the model we have built (written in Equation 4) to predict over a given dataset, we obtain the following equation:

$$\mathcal{L}_{MSE} = \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}_n \text{ where } \mathcal{L}_n = (\mathbf{y}_n - f(\mathbf{x}_n))^2 = (\mathbf{y}_n - f^{(L)} \circ f^{(L-1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(\mathbf{x}_n))^2 \tag{5}$$

It is important to notice that the loss can be seen as a function of all trainable parameters $\boldsymbol{\theta}$ and inputs $\mathbf{x}$ or as function of neural network outputs $\mathbf{y}^*$; the last interpretation will be very useful when dealing with backpropagation of the errors, because we can divide the contribution from the model itself and the loss. In both case targets $\mathbf{y}$ are involved because we are in a supervised learning[2] context but they are fixed values.

[2]Supervised Learning (SL) is a machine learning paradigm for problems where the available data consists of labelled examples, meaning that each data point contains features (covariates) and an associated label. The goal of supervised learning algorithms is learning a function that maps feature vectors (inputs) to labels (output), based on example input-output pairs.

Chain Derivatives

As said above, the quantities we are interested on are the partial derivatives in Equation 3; we will use the chain rule for the derivative computation for compound fractions and define as auxiliary quantities the outputs $\mathbf{z}_l$ and activation $\mathbf{a}_l$ of each layer l:

$$\mathbf{z}_l := \mathbf{W}_l^T \mathbf{a}_{l-1} + \mathbf{b}_l = \mathbf{W}_l^T \Phi(\mathbf{z}_{l-1}) + \mathbf{b}_l \tag{6}$$

Writing component-wise the derivatives, we get that:

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} = \sum_{k=1}^{N_l} \frac{\partial \mathcal{L}_n}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial w_{i,j}^{(l)}} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{i,j}^{(l)}} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} a_i^{(l-1)} = \delta_j^{(l)} a_i^{(l-1)} \tag{7}$$

$$\frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}} = \sum_{k=1}^{N_l} \frac{\partial \mathcal{L}_n}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial b_j^{(l)}} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)} \cdot 1 = \delta_j^{(l)} \tag{8}$$

$\delta_j^{(l)}$ represents the error due to the neuron j of layer l and can be easily computed in the following way:

$$\delta_j^{(l)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} = \sum_{k=1}^{N_{l+1}} \frac{\partial \mathcal{L}_n}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_{k=1}^{N_{l+1}} \delta_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}$$

$$\frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \frac{\partial}{\partial z_j^{(l)}} \sum_{i=1}^{N_l} w_{i,k}^{(l+1)} \Phi(z_i^{(l)}) = \sum_{i=1}^{N_l} w_{i,k}^{(l+1)} \frac{\partial \Phi(z_i^{(l)})}{\partial z_j^{(l)}} = \sum_{i=1}^{N_l} w_{i,k}^{(l+1)} \Phi'(z_i^{(l)}) \delta_{i,j} = \Phi'(z_j^{(l)}) w_{j,k}^{(l+1)}$$

$$\delta_j^{(l)} = \sum_{k=1}^{N_{l+1}} \delta_k^{(l+1)} \Phi'(z_j^{(l)}) w_{j,k}^{(l+1)} \quad \forall 1 \le j \le N_l, \quad \forall 1 \le l \le L-1 \tag{9}$$

Therefore, we could split the computation made by NN into two distinct phases:

- **Forward Pass**: the network is crossed from the input layer to the output layer with Equation 6 and generates the desired output of the model $\mathbf{y}_n^*$ for each sample $\mathbf{x}_n$.

$$\mathbf{z}_l = \mathbf{W}_l^T \mathbf{a}_{l-1} + \mathbf{b}_l \quad l = 1, \dots, L$$

$$\mathbf{a}_l = \Phi(\mathbf{a}_{l-1}) \quad l = 1, \dots, L$$

$$\mathbf{a}_0 = \mathbf{x}_n \in \mathbb{R}^{N_i}, \ \mathbf{y}_n^* = \mathbf{a}_L \in \mathbb{R}^{N_o}$$

- **Backward Pass**: now the NN is crossed in the opposite direction, computing recursively errors related to single neurons $\{\boldsymbol{\delta}_l\}_{l=1}^L$ of all the layers with Equation 9, then it is transferred to single weights and biases with Equation 7 and Equation 8. In vector form[3], we get:

$$\boldsymbol{\delta}_L = -2(\mathbf{y}_n - \mathbf{y}_n^*) \odot \Phi'(\mathbf{z}_L)$$

$$\boldsymbol{\delta}_l = (\mathbf{W}_{l+1} \boldsymbol{\delta}_{l+1}) \odot \Phi'(\mathbf{z}_l) \quad l = (L-1), \dots, 1$$

$$\frac{\partial \mathcal{L}_n}{\partial \mathbf{W}_l} = \boldsymbol{\delta}_l \odot \mathbf{a}_l \qquad \frac{\partial \mathcal{L}_n}{\partial \mathbf{b}_l} = \boldsymbol{\delta}_l \quad l = 1, \dots, L$$

It is important to notice that the choice of the loss function is involved directly only on the computation of the last error term $\boldsymbol{\delta}_L$ and that derivative of the activation function Φ is computed a lot of times, therefore we want it to be easily differentiable as it happens for the common choices shown in Table 1.

2.1.3. Optimizers

For a model, learning means solving the optimization problem of finding the values of the parameters that minimize the loss function as stated in Equation 2. Since Neural Networks often contain thousands of parameters, the exact solution is not available in closed form, hence we proceed with algorithms that reduce step after step the loss function in N iterations, also called *epochs*.

The choice of the optimizer, affects both the accuracy and the learning speed of the model. We now present some common available choices for classical Neural Networks, which will be useful to understand the more complex optimizers presented later in this project for Bayesian Neural Networks.

[3] Those Equations can be translated into vector form using the elementwise product $\odot$ i.e. the Hadamard Product

Gradient Descent

Every optimization algorithm, to which we refer as *optimizer*, updates all trainable parameters of the NN with the simple update rule $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \delta\boldsymbol{\theta}^{(t)}$, so the difference among optimizers consist in the term $\delta\boldsymbol{\theta}$.
The *Gradient Descent* (GD) algorithm originates from the simplest idea: the gradient of a function points to the direction of largest increase of the function, hence, to minimize the function, we take a step in the opposite direction. So, $\delta\boldsymbol{\theta} = -\gamma\mathcal{L}(\boldsymbol{\theta})$, where γ is called *Learning Rate* (LR) and is an hyper-parameter[4] of the NN.

Algorithm 1: Gradient Descent (GD)

Initialization: learning rate $\gamma > 0$, number of iterations N_{epochs}, initial state $\boldsymbol{\theta}^{(0)}$
for $t = 1, \dots, N_{epochs}$ **do**
  Compute the gradient of the full loss function $\nabla\mathcal{L}(\boldsymbol{\theta}^{(t-1)})$
  Update the trainable parameters: $\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t)} - \gamma\nabla\mathcal{L}(\boldsymbol{\theta}^{(t-1)})$

Stochastic Gradient Descent

In ML, many loss functions are formulated as a sum or mean over a function computed on single data samples, so a common practice is to use the batched version of the optimizer to speed up the computing.
To obtain it, we evaluate for each iteration the loss function not on the whole dataset, but only on a subset of it, called *batch* (and its cardinality is called *batch size*).

In the case of *Stochastic Gradient Descent* (SGD) we take the limit case and we choose a batch size of one and compute its loss contribute, denoted as $\mathcal{L}_n$; for example, for MSE, it is equal to $(\mathbf{y}_n - \mathbf{y}_n^*)^2$.

Algorithm 2: Stochastic Gradient Descent (SGD)

Initialization: learning rate $\gamma > 0$, number of iterations N_{epochs}, initial state $\boldsymbol{\theta}^{(0)}$
for $t = 1, \dots, N_{epochs}$ **do**
  Sample uniformly $n \in \{1, 2, \dots, N_{samples}\}$
  Compute the gradient of the loss on the n^{th} sample $\nabla\mathcal{L}_n(\boldsymbol{\theta}^{(t-1)})$
  Update the trainable parameters: $\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t)} - \gamma\nabla\mathcal{L}_n(\boldsymbol{\theta}^{(t-1)})$

In this way, we avoided the computation of the full gradient every time, but we lost the guarantee that the update is pointed towards the minimum at each iteration. However, we expect that on average the ensemble direction is the right one.

Adam Optimizer

Among the first order optimizers, the most popular choice is *Adam*, an empowered version of Stochastic Gradient Descent. This is an *adaptive learning rate method*, meaning that *Adam* is able to learn an optimal LR, which represents the width of displacements for each step of gradient descent. Its main strength is combining the pros of other two common optimizers derived from SGD:

- *AdaGrad*, that maintains a per-parameter LR improving performance on problems with sparse gradients;
- *RMSProp*, whose LR are adapted basing on the average of recent magnitudes of the gradients for the weights, so the algorithm performs well on online and non-stationary problems.

Adam can be defined as a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments and it is considered the best benchmark in the choice of optimizers.

Algorithm 3: Adaptive Moment Estimation (Adam)

Initialization: α, β_1, β_2, ϵ, number of iterations N_{epochs}, initial state $\boldsymbol{\theta}^{(0)}$, momenta $\mathbf{m}^{(0)}$, $\mathbf{v}^{(0)}$
for $t = 1, \dots, N_{epochs}$ **do**
  $\mathbf{m}^{(t)} = \beta_1\mathbf{m}^{(t-1)} + (1-\beta_1)\nabla\mathcal{L}(\boldsymbol{\theta}^{(t-1)})$
  $\mathbf{v}^{(t)} = \beta_2\mathbf{v}^{(t-1)} + (1-\beta_2)(\nabla\mathcal{L}(\boldsymbol{\theta}^{(t-1)}))^2$
  $\hat{\mathbf{m}} = (1-\beta_1)^{-1}\mathbf{m}^{(t)}$, $\hat{\mathbf{v}} = (1-\beta_2)^{-1}\mathbf{v}^{(t)}$
  $\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t-1)} - \alpha\frac{\hat{\mathbf{m}}}{\sqrt{\hat{\mathbf{v}}}+\epsilon}$

We reported in algorithm 3 *Adam*'s general scheme, using the following notation:

- $\mathbf{m}$, $\mathbf{v}$ are the first and second moment of the gradient, initialized at $\mathbf{0}$;
- $\hat{\mathbf{m}}$, $\hat{\mathbf{v}}$ are unbiased estimators of $\mathbf{m}$, $\mathbf{v}$;
- α, β_1, β_2, and ϵ are fixed hyper parameters.

[4] A more detailed explanation of parameters and hyper-parameters will be given in subsubsection 2.1.4

2.1.4. Training Neural Networks

In this section we present the traditional pipeline that is followed to improve the performances for the most of Deep Learning models used in literature. The success of the prediction task depends on model parameters; indeed, artificial Neural Networks are called *parametric models* because they capture all the information about their predictions within a finite set of parameters and they need to be trained to select the best parameters to represent the map we want to be learned by the model.

First of all, we need to clarify some distinctions among parameters, because a group of parameters is updated during learning, while others have to be fixed to a value. We therefore make a distinction into:

- **Trainable parameters**: they appear directly in the loss evaluation and typically gradients are taken with respect of this set of parameters. The mechanism of their choice is usually referred to as *training*. In most of the cases, they are the ensemble of weights and biases of the NN, but there can be exceptions. For example when dealing with inverse problems (presented in the dedicated section), there can be other trainable parameters called $\boldsymbol{\lambda}$, or in the transfer learning context not all NN parameters are trainable.
- **Hyper parameters**: they are all the other parameters necessary to build the model; their choice is called *tuning* and it is usually performed by hand with literature-based knowledge or through grid search-like algorithms. Some relevant examples are all the optimizers' parameters (e.g. learning rate for GD), the number of epochs, the architecture of the NN itself ...

Data Processing

A high quality of data can play a key role in the model performance. This is the reason why it may be useful to work with transformed versions of data, more friendly for the NN range of optimal domain, obtained through a preliminary *pre-processing*, or ask the network for an output that is easier to reconstruct, which then need to be restored into the desired one with some *post-processing*.

Moreover, it is an important practice in ML to partition the dataset into *training*, *validation* and *test* subset, because for the evaluation of the model performance it is not fair to use the data already seen and used to improve the model. This means that we cannot use the whole dataset for training, but we should reserve a part of them for assessing model goodness in an unbiased way; this could sound be penalizing in ML as it common to wish to train with as many data as possible, but in our application, as we will state in subsection 2.2, we rely also on the physical information from partial differential equations rather than on big number of data, hence we did not encounter problems of shortage of data.

Learning Workflow

After the pre-processing of the dataset, a precise method should be follow in order to code a ML model for the desired task which is capable of both accurate and generalizable predictions:

1. **Training phase**: updating the NN's trainable parameters in order to minimize the loss on the training dataset, during subsequent N_{epochs} iterations, increasing *accuracy*. This is the only phase in which the optimizer comes into play, and it requires the heaviest computations (mainly due to the gradients).
2. **Validating phase**: evaluating model's *generality* to ensure that the model did not overfit on the training set, that is, learn also noise or peculiarities in the data provided. This enables us to tune the hyperparameters in order to improve the model, also with the aid of performance metrics aside from the loss.
3. **Testing phase**: using unseen data is able to provide the true unbiased *evaluation* of model performance.

Approximation Power

A possible rising question could be: *what are the functions that Neural Networks are able to approximate?* The comforting answer, when the activation ϕ is sigmoid-like, is given by Barron's Approximation Result.

Theorem 2.1. *Given a function $f : \mathbb{R}^D \to \mathbb{R}$, let $\hat{f}$ be its Fourier Transform.*

$$\text{If } \exists C > 0 \text{ such that } \int_{\mathbb{R}^D} |\omega||\hat{f}(\omega)|d\omega \leq C. \text{ Then } \forall n \geq 1,\ \exists f_n,\ \{c_j\}_{j=1}^n \subset \mathbb{R}, \text{ with:}$$
$$f_n(x) = \sum_{j=1}^{n} c_j \phi(x^T w_j + b_j) + c_0 \text{ so that } \int_{|x|\leq r} |f(x) - f_n(x)|^2 dx \leq \frac{(2Cr)^2}{n}$$

This is equivalent to state that every NN with a single hidden layer and a sigmoid-like activation function ϕ can approximate with arbitrarily small error (in the sense of the L^2 norm any continuous function on a compact set, provided that a sufficient number of hidden neurons are employed. There exist a similiar result also for the pointwise approximation error, obtained by Shekhtman; in this case, the activation function is the ReLU.

2.2. Physics Informed Neural Networks

In this section we present *Physics Informed Neural Networks* (PINN), which represent a DL framework for solving problems involving partial differential equations. They have been introduced in the last 5 years, and in their development Karniadakis, Perdikaris and Raissi have been pioneers (see [9], [10]). Nowadays, these tools are being empowered even outside academia, as can be seen from solutions such as NVIDIA Modulus (see here). PINNs consist in Neural Networks that are trained to solve *supervised learning* tasks while respecting laws of physics governed by ordinal (ODE) or partial (PDE) differential equations.

In standard ML, Neural Networks are common in "big data" frameworks, when plenty of observations are available to train our model, but in some applications data may be extremely costly and difficult to evaluate, such as in the medical field. Therefore, the exploitation of the physical knowledge that we have on the phenomena, represented by differential models, plays a key role, since it enables us to work even in a "small data" regime.

2.2.1. Structure of a PINN

We now illustrate the general structure of Physics-Informed Neural Networks. Given a generic partial differential equation (linear/nonlinear, steady/unsteady ...), we introduce the following notation:

- Ω is a bounded domain in $\mathbb{R}^n$ and $[0, T]$ is the time domain;
- $\mathbf{x}$ and t are, respectively, space and time variable; $\mathbf{x} \in \Omega$, $t \in [0, T]$, used as inputs;
- $\mathbf{u}(\mathbf{x}, t)$ and $\mathbf{f}(\mathbf{x}, t)$ are the solution and parametric field, which can depend in general on space and time;
- $\mathcal{N}$ is the fully known differential operator applied to $\mathbf{u}$;
- $\boldsymbol{\lambda}$ represents the physical parameters of the problem which could be unknown.

The analytical formulation of the problem to solve is therefore:

$$\begin{cases} \mathcal{N}(\mathbf{u}(\mathbf{x}, t); \boldsymbol{\lambda}) = \mathbf{f}(\mathbf{x}, t) & \text{in } \Omega \times [0,\mathrm{T}] \\ + \text{ boundary conditions} & \text{on } \partial\Omega \times [0,\mathrm{T}] \\ + \text{ initial conditions} & \text{on } \Omega \times \{0\} \end{cases} \tag{10}$$

In Figure 4 we show how PINNs work: a classical NN is used to make a prediction on the solution $\mathbf{u}$ with its forward pass, then, during loss evaluation (MSE, in the example) a new regularization term is added: the *residual loss* denoted as MSE_R. The evaluation of residual is not made by NN layers but is performed by computing partial derivatives with automatic differentiation[5] on model outputs with respect to the inputs and then combine in the differential operator.

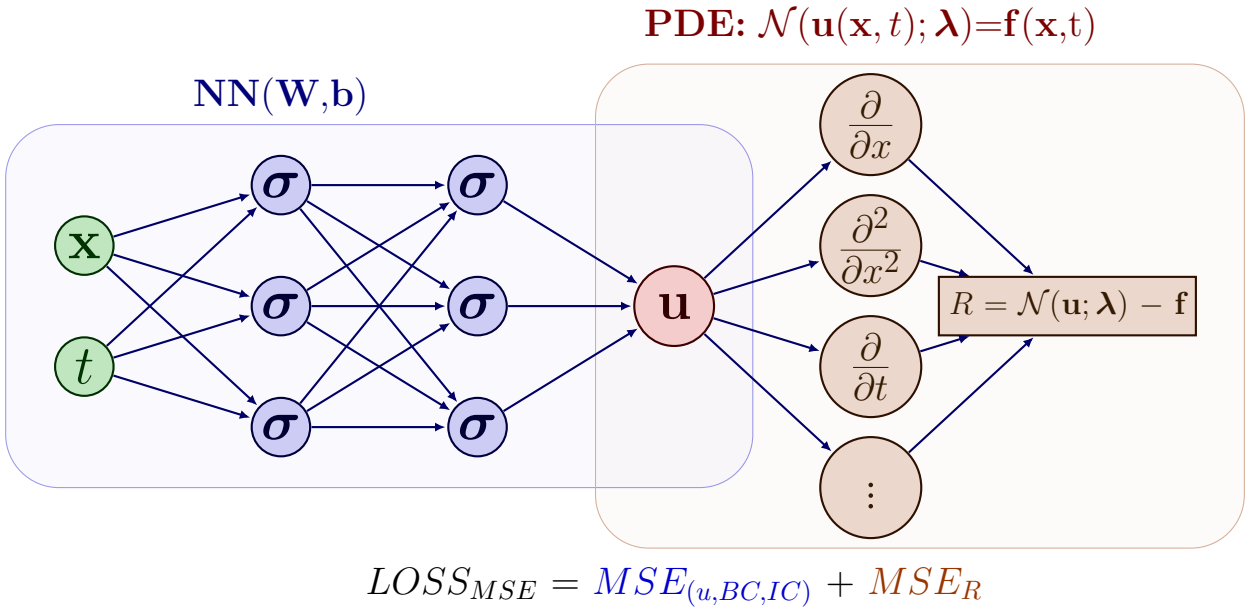


Figure 4: Structure of a Physics-Informed Neural Network.

[5] Automatic Differentiation (AD) is a set of techniques to evaluate the derivative of a function specified by a computer program. It exploits the fact that every computer program, regardlessly of its complications, executes a sequence of arithmetic operations and elementary functions.

2.2.2. Optimization Problems

Introducing a ML model on this kind of analytical problem allow us to consider the latter equivalent to solving an optimization problem. In our project we deal with a steady PDE-constrained problem, such as:

$$\begin{cases} \min_{u} \mathcal{L}(u) = ||u - t||_{L^2} \\ s.t.\ \mathcal{N}(u; \boldsymbol{\lambda}) = f \text{ in } \Omega \end{cases}$$

where we adopt the following notation:

- $\mathcal{L}$ the quantity to optimize;
- u the variable of minimization;
- t the target function we want to learn;
- f the parametric field in the PDE.

Discrete Problem

The first step is to discretize the t and f functions, because we will work with sparse measurements of them. Those will be available only in some points, denoted with $\mathbf{x}$, so we approximate the continuum problem with vectors $\mathbf{t} = \mathbf{t}(\mathbf{x})$ and $\mathbf{f} = \mathbf{f}(\mathbf{x})$ with variable of minimization $\mathbf{u} = \mathbf{u}(\mathbf{x})$. The discretized problem reads as:

$$\begin{cases} \min_{\mathbf{u}} \mathcal{L}(\mathbf{u}) = MSE(\mathbf{u}, \mathbf{t}) \\ s.t.\ \mathcal{N}(\mathbf{u}; \boldsymbol{\lambda}) = \mathbf{f} \quad \forall \mathbf{x} \end{cases} \tag{11}$$

Note that the L^2-norm is now replaced by its dicrete counter-part the MSE which play the role of empirical mean, computed on the set of observation $\mathbf{x}$.

Using a NN as model, we can rewrite the first equation in Equation 11 in term of parameter $\boldsymbol{\theta} = (\mathbf{W}, \mathbf{b})$ using it as the variable of minimization, resulting in:

$$\min_{\boldsymbol{\theta}} \mathcal{L}(\mathbf{u}), \text{ where } \mathbf{u}(\mathbf{x}; \boldsymbol{\theta}) = \mathcal{NN}(\mathbf{x}; \boldsymbol{\theta})$$

Finally, the optimization part of our problem depends only on the trainable parameters of the NN. The remaining part of Equation 11 is the differential costraint; to solve that, the PINN section of our model comes into play.

Differential Residuals

If the couple given by a function $\mathbf{u}$ and a parametric field $\mathbf{f}$ is the solution of the PDE, then we can define the residual of the operator as:

$$R \coloneqq \mathcal{N}(\mathbf{u}; \boldsymbol{\lambda}) - \mathbf{f} \tag{12}$$

Therefore, we can impose $R = 0$ through a loss function for values of $\mathbf{u}$ and $\mathbf{f}$ that approximate the exact ones with our prediction and help the model learning functions reasonably close to the correct ones by including *MSE(**R**,**0**)* in $\mathcal{L}$ during the training phase.

Boundary and initial conditions can be easily implemented just by considering domain points $\mathbf{x}$ belonging to the parabolic boundary. More details on the explicit formulation of the losses, regularization and on the dataset needed, will be provided in subsubsection 2.2.3 and case-specific sections.

Direct and Inverse Problems

When dealing with PINNs, we can consider both *direct* or *inverse* problems and is important to remark the difference, because we need to treat the regularization term with the introduction of PINN.
The difference between the two consists basically in the knowledge that we have on some parameters $\boldsymbol{\lambda}$, representing physical coefficients (such as a diffusion coefficient): in the direct setting they are known, while in the inverse one they need to be estimated from other measurements available.

In *direct problems* the value of physical parameters $\boldsymbol{\lambda}$ is known, so it can be overlooked as it was part of the differential operator $\mathcal{N}$, leading us to the following formulation:

$$\begin{cases} \min_{\theta} \mathcal{L}(\mathbf{u}) = MSE(\mathbf{u}, \mathbf{t}) + R(\mathbf{u}) \\ \text{with } \mathbf{u}(\mathbf{x}; \boldsymbol{\theta}) = \mathcal{NN}(\mathbf{x}; \boldsymbol{\theta}) \end{cases} \tag{13}$$

In the *inverse setting*, the model physical parameters $\boldsymbol{\lambda}$ are unknown and are reconstructed by the model. In this case, the optimization problem solved by the PINN is a generalization of the above one: in fact, it is only needed to include the set of physical parameters of interest $\boldsymbol{\lambda}$ among the residuals R, leading to:

$$\begin{cases} \min_{\theta,\lambda} \mathcal{L}(\mathbf{u};\boldsymbol{\lambda}) = MSE(\mathbf{u},\mathbf{t}) + R(\mathbf{u};\boldsymbol{\lambda}) \\ \text{with } \mathbf{u}(\mathbf{x};\boldsymbol{\theta}) = \mathcal{NN}(\mathbf{x};\boldsymbol{\theta}) \end{cases} \tag{14}$$

2.2.3. Dataset for PINNs

Training and validating a model for a PINN follows a similar schedule as the one stated in subsubsection 2.1.4, but with the addiction of some peculiarity, in fact computing the forward pass is a much more easy task the the whole evaluation of loss needed with the insertion of the residuals.
For validation and testing we can rely in the same way as before on *validation points* and *test points* without any issue but wee can distinguish three important subdatasets within the *training dataset*:

Fitting Points, i.e. points inside the domain Ω where we impose a value for the solution (obtained by the known analytical expression or, for example, from numerical simulations) and act in the classical training set fashion. Those are a few because we work in a *small data regime* and can be limited only to certain subdomain of Ω where is easier to take real measurements of u_{ex}. The loss to add is in this case:

$$\mathcal{L}_{fit} = \frac{1}{N_{fit}} \sum_{n=1}^{N_{fit}} (u(x_n,t_n) - u_{ex}(x_n,t_n))^2$$

Collocation Points, being the points inside the domain Ω where we impose the PDE constraints, by requiring to minimize the residual of the equation. Referring to Equation 12, our residual is:

$$\mathcal{R}(\mathbf{x},t) = \mathcal{L}(\mathbf{u}(\mathbf{x},t)) - \mathbf{f}(\mathbf{x},t))$$

and using MSE loss function as in Figure 4 we aim at minimizing the loss:

$$\mathcal{L}_{col} = \frac{1}{N_{col}} \sum_{n=1}^{N_{col}} \mathcal{R}(\mathbf{x}_n,t_n)^2$$

with $\{(\mathbf{x}_n,t_n)\}_{n=1}^{N_{col}}$ denoting the elements of the set of collocation points. Those points are totally avalaible beacuase there is no need of target function but they affect in the bigger way the computational cost for the evaluation of gradients with respect to inputs.

Boundary Points, on which we impose the exact value when we have a Dirichlet Boundary condition, or the minimization of the residual of the equation describing the Neumann condition; depending on than they are more similar to the first or second set presented respectively. For example, with a Dirichlet Boundary Condition of the form $u = f$ on Γ_D we add the loss:

$$\mathcal{L}_{bnd} = \frac{1}{N_{Bnd}} \sum_{n=1}^{N_{Bnd}} (u(x_n,t_n) - f(x_n,t_n))^2$$

Those points come from physical equations, involving only f and not u_{ex}, so we can use a lot of them without worries. The same can be done for the initial condition

Combination of the Losses

The final loss function that we want the net to minimize includes the effect of all the losses introduced above, with a weighted combination of them. For example, with M losses $\mathcal{L}_i$, we have

$$\mathcal{L} = \sum_{i=1}^{M} q_i \mathcal{L}_i$$

with q_i indicating the weight of the i-th loss.
Note that $\mathbf{q}$ is an hyperparameter of the net, and its choice will be crucial for the performance.

2.3. Bayesian Neural Networks

Bayesian Neural Networks (BNNs) represent a way to introduce *uncertainty quantification* (UQ) in the framework of modern deep-learning methods, which constitute incredibly powerful tools to tackle challenging problems, but they often operate as black boxes on the significance of their predictions.

Uncertainty Quantification

Up to now, we have presented methods to reconstruct functions with Neural Networks which are not able to provide information on the *confidence* of the prediction. With NNs, we are indeed unable to quantify the uncertainty of the prediction, because we obtain from the model just one sample of the solution, namely the network output correspondent to the given input, using as network parameters the ones chosen after the process of loss minimization with the already presented methods.

The main idea is to replace $\boldsymbol{\theta} \coloneqq \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L}$ with a suitable joint probability distribution and change the way we perform the forward pass. For the validation and test phase N new instances of $\boldsymbol{\theta}$ are sampled and they generate N different independent predictions, which are actually the output of a classical NN whose parameters are each member the set at a time. Finally, having N samples of the output, we take the relevant statistics and use the mean for prediction and the variance for the UQ.

The real burden occurs during the model training, because it is mandatory to have *ad hoc* algorithms for the BNN; this discussion is postponed to section 3, that is entirely devoted to describe the task.

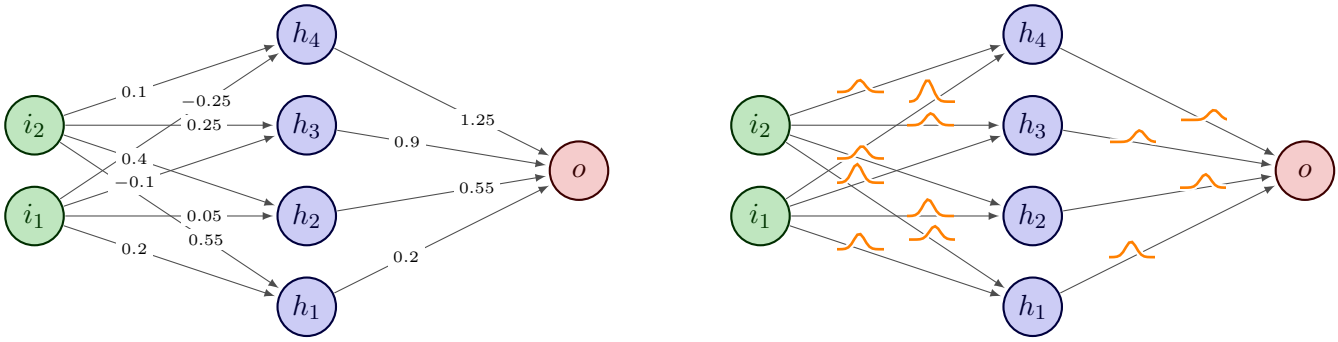


Figure 5: Difference between classical Neural Network (NN) and Bayesian Neural Network (BNN)

Bayesian Framework

The BNN framework is based on the idea of combining neural network theory with Bayesian inference. The fundamental concept on which the method relies comes indeed from Bayesian statistic, and it expresses the fact that prior beliefs influence posterior beliefs. This idea can be formalized into the following key theorem:

Theorem 2.2 (Bayes Theorem). *Given two events A, B, such that $\mathbb{P}(B) \neq 0$, the conditional probability of A given B, denoted as $\mathbb{P}(A|B)$, can be computed as*

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(B|A)\mathbb{P}(A)}{\mathbb{P}(B)}.$$

While working with BNNs, we will apply the same theorem in its continuous version to probability distributions (denoted with p), and the distribution of interest will be the one of the network parameters $\boldsymbol{\theta} \coloneqq \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L}$, because from it we can infer the distribution of the output.

Theorem 2.3 (Bayes Rule). *For two continuous random variables X and Y, let us denote by f_X and f_Y their probability density. Then, the density of the conditioned random variable $X|Y = y$ can be computed as follows:*

$$f_{X|Y=y}(x|y) = \frac{f_{Y|X=x}(y|x)f_Y(y)}{f_X(x)}, \qquad \text{where } f_{Y|X}(y|x)f_X(x) = f_{X,Y}(x,y) = f_{X|Y}(x|y)f_Y(y)$$

The Bayes theorem is used to update prior knowledge on the network parameters' distribution exploiting the information coming from the observed data/the physics. At the same time, it turns out that we can establish a straightforward relation between probability distributions and loss functions: the good quality of the prediction can be indeed either represented by

1. a high value of the probability density;
2. a low value of the loss function.

Hence, probability densities can be inserted into the NNs presented in subsection 2.1, by considering their opposite inside the loss function and recovering in this way the classical optimization problem framework.

2.3.1. Likelihood Distribution

We consider a scenario in which the dataset D consists of measurements of the external forces (denoted by D_f) and of the PDE solutions (denoted by D_u when internal and by D_b when on the boundary).
As in the case of the direct problem it is common not to rely on measurements of the solution inside the domain but only on the boundary, it is convenient to distinguish these points in the definition of the dataset D, which is then splitted into:

$$D = D_u \cup D_b \cup D_f.$$

It is reasonable to consider that the available measurements are affected by some *noise*, due to inevitable defaults of the measurement tools.
Moreover, in principle we shall not include any co-implication between the measurements of the different quantities, hence we assume that $\overline{\mathbf{u}}, \overline{\mathbf{f}}$ are independent gaussians, centered at the real hidden value $\mathbf{u}, \mathbf{f}$:

$$\overline{\mathbf{u}}^{(i)} = \mathbf{u}(\mathbf{x}^{(i)}) + \varepsilon_u^{(i)} \text{ with } \varepsilon_u^{(i)} \sim \mathcal{N}(0, \sigma_u^{(i)}) \qquad i = 1, \dots, N_u$$

$$\overline{\mathbf{u}}^{(i)} = \mathbf{u}(\mathbf{x}^{(i)}) + \varepsilon_b^{(i)} \text{ with } \varepsilon_b^{(i)} \sim \mathcal{N}(0, \sigma_b^{(i)}) \qquad i = 1, \dots, N_b$$

$$\overline{\mathbf{f}}^{(i)} = \mathbf{f}(\mathbf{x}^{(i)}) + \varepsilon_f^{(i)} \text{ with } \varepsilon_f^{(i)} \sim \mathcal{N}(0, \sigma_f^{(i)}) \qquad i = 1, \dots, N_f$$

and that the variances $\sigma_u^{(i)}, \sigma_b^{(i)}, \sigma_f^{(i)}$ of the noise terms are known.
In particular, following the choices made in [11], we set them to equal values $\sigma_u, \sigma_b, \sigma_f$, resulting in working with noises that are *independent and identically distributed* for each group. Since the measurements on $\mathbf{u}$ and $\mathbf{f}$ are assumed independent, the *likelihood* of data $p(D|\boldsymbol{\theta})$ is computed as

$$p(D|\boldsymbol{\theta}) = p(D_u|\boldsymbol{\theta})p(D_b|\boldsymbol{\theta})p(D_f|\boldsymbol{\theta}). \tag{15}$$

By the former assumption on gaussianity, each density factor reads as:

$$p(D_u|\boldsymbol{\theta}) = \prod_{i=1}^{N_u} \frac{1}{\sqrt{2\pi\sigma_u^2}} \exp\left(-\frac{(\overline{\mathbf{u}}^{(i)} - \mathbf{u}^{(i)})^2}{2\sigma_u^2}\right), \tag{16}$$

$$p(D_b|\boldsymbol{\theta}) = \prod_{i=1}^{N_b} \frac{1}{\sqrt{2\pi\sigma_b^2}} \exp\left(-\frac{(\overline{\mathbf{u}}^{(i)} - \mathbf{u}^{(i)})^2}{2\sigma_b^2}\right), \tag{17}$$

$$p(D_f|\boldsymbol{\theta}) = \prod_{i=1}^{N_f} \frac{1}{\sqrt{2\pi\sigma_f^2}} \exp\left(-\frac{(\overline{\mathbf{f}}^{(i)} - \mathbf{f}^{(i)})^2}{2\sigma_f^2}\right). \tag{18}$$

2.3.2. Prior Distribution

The initial knowledge on the distribution of network parameters $\boldsymbol{\theta}$ is summarized by $p(\boldsymbol{\theta})$, which is called *prior*.
A classic choice for NN parameters is considering $\boldsymbol{\theta}$ to be *a priori* distributed as a multivariate normal.
As stated in [11], a common choice in the Bayesian learning framework is to assume *independence* among each weight and bias of the neural network and *zero mean*. For what concerns the variance, the reference paper assumes it to be 1 for each network parameter, and in the proposed implementation the variance is set to

$$\sigma_\theta^2 = \frac{50}{N_l}$$

where N_l the number of neurons per layer. $N_l = 50$ is the choice made in the reference paper, hence we set the same variance if the number of neurons per layer is the same, and ensure a bigger flexibility (represented by a bigger variance) in the case of a network with less parameters. Therefore,

$$p(\boldsymbol{\theta}) = \prod_{i=1}^{N_\theta} \frac{1}{\sqrt{2\pi\sigma_\theta^2}} \exp\left(-\frac{(\boldsymbol{\theta}^{(i)} - 0)^2}{2\sigma_\theta^2}\right), \tag{19}$$

where N_θ is the total number of network parameters (considering both weights and biases).

2.3.3. Posterior Distribution

The Bayes theorem enables to reconstruct, starting from *prior* and *likelihood*, the quantity $p(\boldsymbol{\theta}|D)$, that is called *posterior* distribution of $\boldsymbol{\theta}$ and represents the update of the prior knowledge on the parameters' distribution after having obtained the information from data. The theorem's formulation reads then as:

$$p(\boldsymbol{\theta}|D) = \frac{p(D|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(D)}.$$

$p(D)$ represents the distribution of the dataset, but its calculation is typically unfeasible. Hence, we exploit the Bayes Theorem only to get the relation of proportionality:

$$p(\boldsymbol{\theta}|D) \propto p(D|\boldsymbol{\theta})p(\boldsymbol{\theta}) \tag{20}$$

The reconstruction of the posterior as presented above is then exploited to sample a set $\{\boldsymbol{\theta}_i\}_{i=1}^M$ where

$$\boldsymbol{\theta}_i \sim p(\boldsymbol{\theta}|D) \qquad i = 1, ..., M$$

However, computing the posterior directly is impossible in general with Neural Networks, therefore methods involving Markov Chain Monte Carlo or Variational Inference (as the ones presented in 3) can be used to sample directly from the posterior distribution.

Having reconstructed the posterior distribution of parameters, they are then used to reconstruct the posterior predictive distribution of the neural network - namely, the distribution of the network output $f_{\boldsymbol{\theta}}(x)$.
Indeed, by producing the neural networks output corresponding to the sampled parameters, the estimation of the mean and of the variance of the output f given a fixed input $\mathbf{x}^*$ is performed as follows:

$$\mathbb{E}[f|\mathbf{x}^*, D] \approx \frac{1}{M}\sum_{i=1}^{M} f_{\theta_i}(\mathbf{x}^*)$$

$$\mathrm{Var}[f|\mathbf{x}^*, D] \approx \frac{1}{M}\sum_{i=1}^{M}(f_{\theta_i}(\mathbf{x}^*) - \mathbb{E}[f|\mathbf{x}^*, D])^2$$

The two quantities obtain represent meaningful metrics for UQ, embedded into a probabilistic framework, and can be used in the production of credible confidence intervals for the neural network output.

2.4. Bayesian Physics Informed Neural Networks

The above discussion does not take into account any information coming from the physics, but relies simply on data. The Bayesian-Physics Informed Neural Network (B-PINNs) approach consists in adding to a BNN model the knowledge that the problem solution is required to satisfy given partial differential equations and boundary conditions. This method is implemented in the main papers of reference, that are [11], [7].

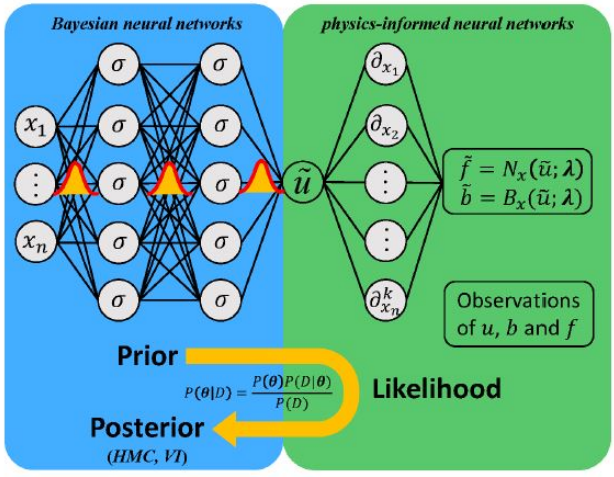


Figure 6: Structure of a B-PINN, extracted from the reference paper [11].

2.4.1. Physical Likelihood

They represent an integration of the PINNs into a Bayesian framework, as the constraint of the PDE appears as additional term of likelihood. In the formulation of the Bayes Theorem the information coming from the partial differential law is present, given the modification of the likelihood term:

$$p(\boldsymbol{\theta}|D,R) \propto p(D,R|\boldsymbol{\theta})p(\boldsymbol{\theta}). \tag{21}$$

The likelihood of data $p(D,R|\boldsymbol{\theta})$ is computed as

$$p(D,R|\boldsymbol{\theta}) = p(D|\boldsymbol{\theta})p(R|\boldsymbol{\theta}) \tag{22}$$

In Equation 22, $p(D|\boldsymbol{\theta})$ is the same quantity present in Equation 15 for the not physics-informed case, and $p(R|\boldsymbol{\theta})$ plays the same role in an artificial way, but representing the information coming from the physical law rather than from measurements.
Considering the set of *collocation points* $\{x_i\}_{i=1}^{N_{col}}$ already defined in subsection 2.2, the goal is to require that at these points the solution approximated by the neural network solves the partial differential equation.
Therefore, one can proceed similarly as done for the *fitting points* while imposing the solution to be close to the measurement, and it is common in the B-PINNs framework to consider this part of the likelihood to be gaussian as well:

$$p(R|\boldsymbol{\theta}) = \prod_{i=1}^{N_{col}} \frac{1}{\sqrt{2\pi\sigma_r^2}} \exp\left(-\frac{(\mathcal{R}(\tilde{\mathbf{u}}^{(i)}) - 0)^2}{2\sigma_r^2}\right). \tag{23}$$

In Equation 23, $\mathcal{R}$ denotes the residual as in subsection 2.2, while σ_r plays the role of σ_u or σ_f in the fitting case and represents an artificial uncertainty related to the physical knowledge. The better the model is in approximating the phenomenon, the smaller will σ_r be; note also that, by introducing differences among the values of uncertainty for data and for physical knowledge, one can give a different weight to the two components of the likelihood, and therefore ensure that the most reliable information (i.e., the one to whom less uncertainty is associated) weighs more and will be more determinant in the learning process.

2.4.2. Prior for Inverse Problems

The Bayesian framework can be applied to inverse problems as well, by considering that the model parameters $\boldsymbol{\lambda}$ (following the same notation as in subsection 2.2) follow a certain probability distribution.
We can assume to have a prior knowledge on $\boldsymbol{\lambda}$, and it can reasonably assumed to be independent from $\boldsymbol{\theta}$, because there is no relation between model and network parameters.
Hence, the prior term containing information on both of them $p(\boldsymbol{\theta},\boldsymbol{\lambda})$ can be computed as

$$p(\boldsymbol{\theta},\boldsymbol{\lambda}) = p(\boldsymbol{\theta})p(\boldsymbol{\lambda}).$$

With the above expression for the prior, the same discussion made in the case of the direct problem still holds, provided that we consider that the likelihood of data depends on fixed model parameters as well, and $p(D|\boldsymbol{\theta})$ is replaced by $p(D|\boldsymbol{\theta},\boldsymbol{\lambda})$. Consequently, the formulation of the Bayes theorem reads as

$$p(\boldsymbol{\theta},\boldsymbol{\lambda}|D,R) = \frac{p(D,R|\boldsymbol{\theta},\boldsymbol{\lambda})p(\boldsymbol{\theta})p(\boldsymbol{\lambda})}{p(D)p(R)}.$$

The latter can be rewritten with the all aforementioned simplifications as:

$$p(\boldsymbol{\theta},\boldsymbol{\lambda}|D,R) \propto p(D|\boldsymbol{\theta},\boldsymbol{\lambda})p(R|\boldsymbol{\theta},\boldsymbol{\lambda})p(\boldsymbol{\theta})p(\boldsymbol{\lambda}) \tag{24}$$

and we obtain a proportionality relation ready-to-use. Finally, naming $L = \log(p)$ for all the functions, we get:

$$Loss = L(D;\boldsymbol{\theta},\boldsymbol{\lambda}) + L(R;\boldsymbol{\theta},\boldsymbol{\lambda}) + L(\boldsymbol{\theta}) + L(\boldsymbol{\lambda}) = Loss_{data} + Loss_{pde} + Prior_{\theta} + Prior_{\lambda} \tag{25}$$