Physics Informed Neural Networks (PINNs)

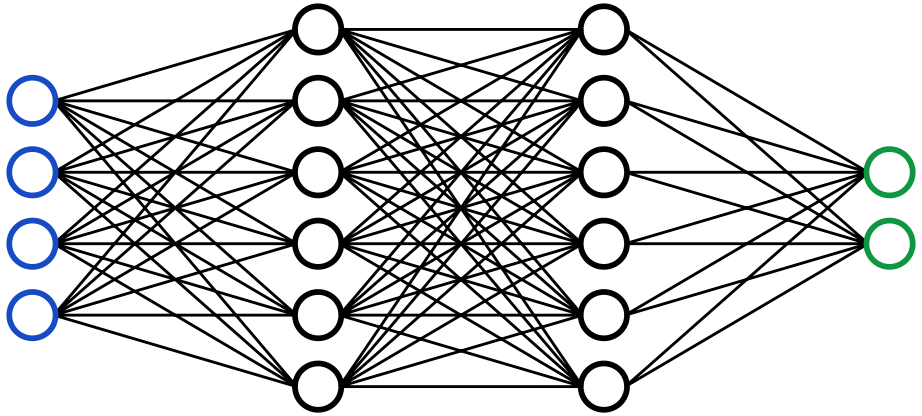
Neural Networks

Neural networks (NNs) are a class of machine learning algorithms that are inspired by the structure and function of the human brain. They consist of interconnected nodes (neurons) organized in layers, where each connection has an associated weight.

From a mathematical point of view, a NN is a function defined as a composition of simpler functions. In the picture, the blue circles are the input variables, the green circles are the output variables, all the lines represent affine transformations and the black circles are nonlinear (simple) activation functions.

$$NN(x) := \sigma(A_N \dots \sigma(A_2\sigma(A_1 x + b_1) + b_2) \cdots + b_N)$$

where σ is a nonlinear activation function, $A_i \in \mathbb{R}^{\ell_i \times \ell_{i-1}}$ for $i = 1, \dots, N$ and $b_i \in \mathbb{R}^{\ell_i}$ are the weights and biases of the network. N is the depth of the network, i.e. the number of layers, and ℓ_i is the number of neurons in the i-th layer.



Neural Networks properties

Though very simple, NN have proven to be very powerful in the last years. In particular, they can approximate functions very well, within few layers.

Universal Approximation Theorem

A feedforward neural network $NN : \mathbb{R}^n \to \mathbb{R}$ with a single hidden layer containing a finite number ℓ_2 of neurons can approximate any continuous function f on compact subsets of $\mathbb{R}^n$ to any desired degree of accuracy. In other words, for any $\varepsilon > 0$ there exists $\ell_1 \in \mathbb{N}$ such that

$$\max_{x \in K \subset \mathbb{R}^n} |NN(x) - f(x)| < \varepsilon.$$

Optimality of universal approximation theorem

More precise estimations are available, but they are far from the capability of these networks. What has been observed is that adding layers increases very quickly the approximation capability of the network.

Training a Neural Network: cost function

In order to make NN close to a function f, we have to train it to learn the *right* weights A_i and biases b_i, to minimize the error between the output of the network and the function f.
This is done by minimizing the cost function on a training set $X_{train} = \{(x_i, y_i)\}_{i=1}^{N_{train}}$:

$$\mathcal{L}(\theta) = \frac{1}{N_{train}} \sum_{i=1}^{N_{train}} \left(NN(x_i) - y_i\right)^2$$

where $\theta \in \mathbb{R}^{N_{param}}$ is the vector of all weights A_i and biases b_i for $i = 1, \dots, N$ collected all together.

Here, I have used the mean square error (MSE) as a **cost function**.
In general more complicate **loss functions** can be used to measure the error between one input and one output.

Minimization problem

What we want to solve is: find

$$\theta^* = \arg\min_{\theta} \mathcal{L}(\theta).$$

Training a Neural Network: forward pass and backpropagation

To find a minimum of the function we need to *put to 0* the gradient, so we use a gradient descent procedure:

$$\theta_i^{n+1} = \theta_i^n - \eta \nabla_{\theta_i} \mathcal{L}(\theta^n)$$

for all $i = 1, \dots, N_{param}$. η is called learning rate, to be tuned a little bit. How to compute the gradient?

Forward pass

$$h_0 := x$$

$$\begin{cases} f_k := A_k h_{k-1} + b_k & \forall k = 1, \dots, N \\ h_k := \sigma(f_k) & \forall k = 1, \dots, N \end{cases}$$

$$NN(x) = h_N$$

$$\mathcal{L}(\theta) = \frac{1}{N_{train}} \sum_{i=1}^{N_{train}} \left(NN(x_i) - y_i \right)^2$$

Backward propagation

We can easily compute the gradient recursively using the chain rule!

$$
\begin{aligned}
\nabla_\theta \mathcal{L}(\theta) =& ? \\
g_{h_N} :=& \nabla_{h_N} \mathcal{L} = 2h_N \\
g_{f_N} :=& \nabla_{f_N} \mathcal{L} = \sigma'(f_N) \nabla_{h_N} \mathcal{L} = \sigma'(f_N) g_{h_N} \\
g_{b_N} :=& \nabla_{b_N} \mathcal{L} = \nabla_{f_N} \mathcal{L} \nabla_{b_N} f_N = g_{f_N} \\
g_{A_N} :=& \nabla_{A_N} \mathcal{L} = \nabla_{f_N} \mathcal{L} \nabla_{A_N} f_N = g_{f_N} h_{N-1} \\
g_{h_{N-1}} :=& \nabla_{h_{N-1}} \mathcal{L} = \nabla_{f_N} \mathcal{L} \nabla_{h_{N-1}} f_N = g_{f_N} A_N \\
g_{f_{N-1}} :=& \nabla_{f_{N-1}} \mathcal{L} = \sigma'(f_{N-1}) \nabla_{h_{N-1}} \mathcal{L} = \sigma'(f_{N-1}) g_{h_{N-1}} \\
& \dots \\
g_{h_k} :=& g_{f_{k+1}} A_{k+1} \\
g_{b_k} = g_{f_k} :=& \sigma'(f_k) g_{h_k} \\
g_{A_k} :=& g_{f_k} h_{k-1}
\end{aligned}
$$

Other details

Stochastic gradient descent

In practice, it has been proven that a stochastic gradient descent algorithm provides less problems with local minima and reaches more easily other lower minima. It consists of splitting the training set in batches and to alternatively train on each of them. Hence, it is often used, also in its many variants (e.g. ADAM).

Dimension of optimization problem

The space where we look for θ is very very high dimensional and it is really hard to visualize or understand how to look for minima. Moreover, one could expect overfitting, since we have often more parameters than data, but stochastic gradient descent + this huge space makes things work.

PINN