Neural Networks

Prof. Alessandro Lucantonio

Aarhus University - Department of Mechanical and Production Engineering

?/?/2023

Motivations

- Neural Networks (NN) are one of the most flexible ML tools.
- Universal approximators!
- Can manipulate real and discrete data $\rightsquigarrow$ regression and classification problems.
- Not a single model: many type of NN (e.g. MLP, CNN, RNN).
- Inspired by biological systems.

Perceptron - Visualization

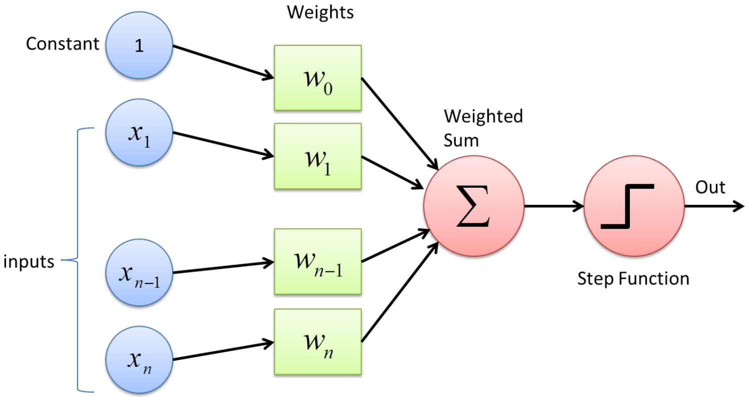


Figure: Representation of a perceptron.

Perceptron - Formal

Notation: n is as usual the number of features. x is an input sample of length $n+1$ with $x_0 = 1$.

Perceptron:

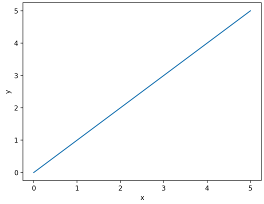
$$\begin{cases} n(\boldsymbol{x}) := \boldsymbol{w}^T \boldsymbol{x} = \sum_{j=0}^{n} w_j x_j, \\ h(\boldsymbol{x}) := f(n(\boldsymbol{x})). \end{cases}$$

n is the **net input** to the neuron.

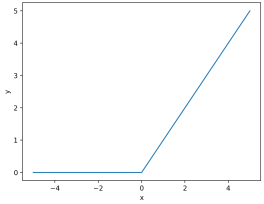
f is called **activation function**. Some common examples are:

- Linear: $f(t) = at + b$.
- ReLU (Rectified Linear Unit): $f(t) := \max\{0, t\}$.
- Sigmoid: $f(t) := \frac{1}{1+e^{-t}}$.
- Tanh (Hyperbolic tangent): $f(t) := \frac{e^{2t}-1}{e^{2t}+1}$

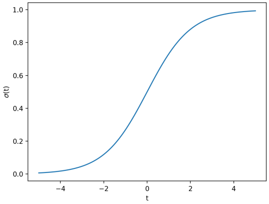
Activation Functions - Visualization



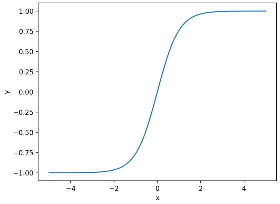
(a) Linear



(b) ReLU

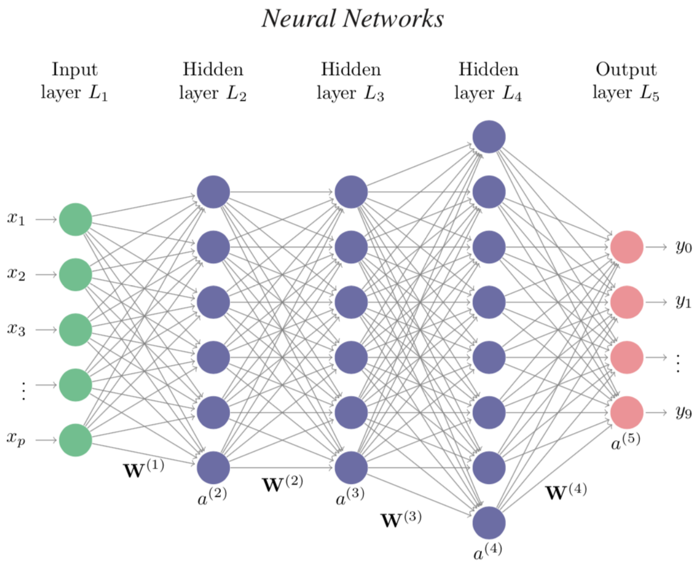


(c) Sigmoid



(d) Tanh

MultiLayer Perceptron (MLP) - Visualization



MLP - Formal

Notation: $\boldsymbol{a}^{(i)}$ is the input vector of the i-th layer and $\mathrm{W}^{(i)}$ is the weight matrix for each layer. m is the number of layers.

Parenthesis: Why do we have a weight matrix for each layer? Each layer i consists of m_i neurons, hence necessarily has m_i weight vectors, which we can arrange row-wise to form the matrix $\mathrm{W}^{(i)}$.

For each layer i compute

$$\begin{cases} n_i(\boldsymbol{a}^{(i)}) := \mathrm{W}^{(i)}\boldsymbol{a}^{(i)}, \\ h_i(\boldsymbol{a}^{(i)}) := f_i(n_i(\boldsymbol{a}^{(i)})). \end{cases}$$

Compact form:

$$h(\boldsymbol{x}) := f_{m-1}\big(\mathrm{W}^{(m-1)} f_{m-2}\big(\cdots f_1\big(\mathrm{W}^{(1)}\boldsymbol{x}\big)\cdots\big)\big)$$

MLP - Some comments

- This kind of NN are also called **feedforward NN**, since we transfer information from input to output.
- The hypothesis function is non-convex! This is due to the fact that composition of convex functions is not necessarily convex. Hence, in the learning problem we will have many local minima and saddle points.
- Theory tells us that MLP with just 1 hidden layer are universal approximators, in the sense that they approximate as well as you want "each" continuous function (not a formal statement, just to provide intuition).

Tips and Tricks - Is one layer really enough?

Theory suggests us that the answer is yes, but pay attention. There exists cases for which an exponential number of units (w.r.t. the input dimension) are required to approximate well the data.

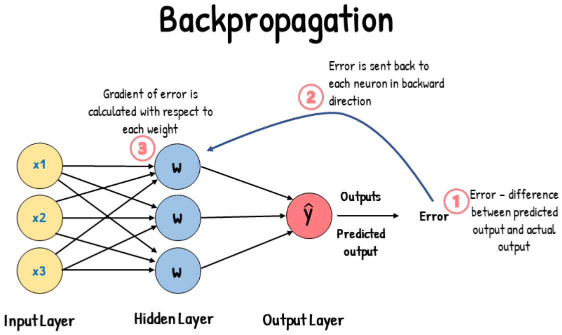
Introducing many layers helps to reduce the total number of units but complicate a bit the learning procedure (see later).

Then the question becomes: how to find the optimal number of layers? Validation is your best friend :)
Include in your model selection process as many different architectures/activation functions as you can.

Back-propagation - Intuition

Notation: $E(\mathrm{W}) = \frac{1}{N}\sum_{j=0}^{N} E_j(\mathrm{W})$ and $\mathrm{W} = (\mathrm{W}^{(1)}, \ldots, \mathrm{W}^{(m-1)})$.

To apply gradient descent we have to compute $\nabla E(\mathrm{W})$. In the case of NN gradient descent algorithm is called **back-propagation**, since it allows the information from the cost to then flow backward through the network in order to compute the gradient.



Backpropagation
Error is sent back to each neuron in backward direction
2
Gradient of error is calculated with respect to each weight
3
x1
x2
x3
w
w
w
ŷ
Outputs
Predicted output
Error
1
Error - difference between predicted output and actual output
Input Layer
Hidden Layer
Output Layer

Back-propagation - Formal

Formal computation only for $E = \text{MSE}$.

Goal: Our goal is to compute $\nabla E_i(\mathrm{W})$ or equivalently $\frac{\partial E_i}{\partial w_{jk}}$.

Chain rule:

$$\frac{\partial E_i}{\partial w_{jk}} = \underbrace{\frac{\partial E_i}{\partial (n_t)_j}}_{A} \underbrace{\frac{\partial (n_t)_j}{\partial w_{jk}}}_{B}.$$

B:

$$\frac{\partial (n_t)_j}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \sum_p w_{jp} a_p^{(t)} = a_k^{(t)}$$

Back-propagation - Formal

A:

$$\frac{\partial E_i}{\partial (n_t)_j} = \underbrace{\frac{\partial E_i}{\partial (h_t)_j}}_{C} \underbrace{\frac{\partial (h_t)_j}{\partial (n_t)_j}}_{D}$$

D:

$$\frac{\partial (h_t)_j}{\partial (n_t)_j} = \frac{\partial}{\partial (n_t)_j}(f_t(n_t))_j = f_t((n_t)_j) = f_t'((n_t)_j) = (f_t'(n_t))_j$$

C in the case of $t = m - 1$ (output layer):

$$\frac{\partial E_i}{\partial (h_{m-1})_j} = \frac{\partial}{\partial (h_{m-1})_j}(\boldsymbol{y}_i - h_{m-1}(\boldsymbol{x}^i))^2 = -2((\boldsymbol{y}_i)_j - (h_{m-1}(\boldsymbol{x}^i))_j)$$

Back-propagation - Formal

C in the case of $t = m - 2$ (notation):

$$\frac{\partial E_i}{\partial (h_{m-2})_j} = \sum_k \frac{\partial E_i}{\partial (n_{m-1})_k} \frac{\partial (n_{m-1})_k}{\partial (h_{m-2})_j}.$$

Note that the terms $\frac{\partial E_i}{\partial (n_{m-1})_k}$ have been already calculated in the previous case. The other term is easy:

$$\frac{\partial (n_{m-1})_k}{\partial (h_{m-2})_j} = \frac{\partial}{\partial (h_{m-2})_j} \sum_p w_{kp} (h_{m-2})_p = w_{kj}.$$

At this point we continue going backward and computing C for $t = m - 3, \ldots, 1$.

NN best practices

- Initialize all weights randomly near zero, but **avoid** to initialize it constantly zero. As a general recipe: the more you are random the more you do not introduce bias.
- Try random starting configurations. For each configurations do multiple runs and look to the mean $\pm$ std results.
- Number of epochs (i.e. number of iterations of SGD) are not an hyperparameter. Fix it to an appropriate number (e.g. 5000 or less) to avoid too slow convergences.
- Use regularization to counter overfitting.