Neural Networks

Prof. Alessandro Lucantonio

Aarhus University

7/11/2023

Introduction

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

Perceptron

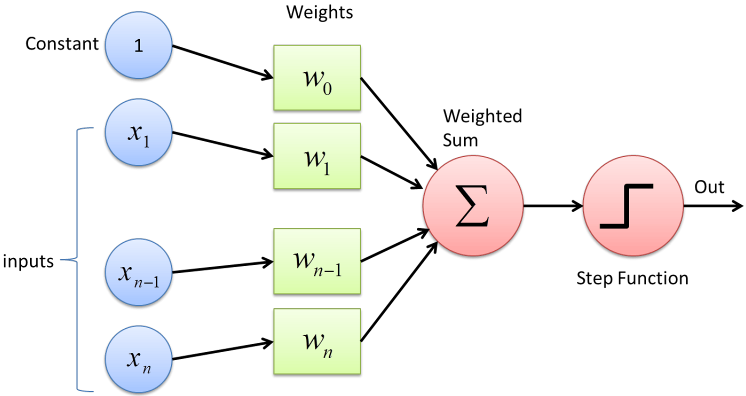
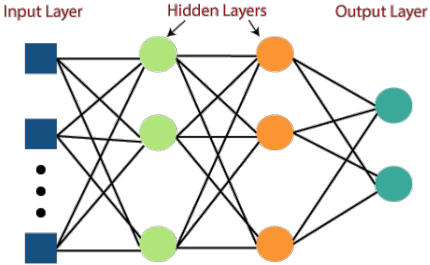


Figure: Representation of a perceptron.

Multi-Layer Perceptron (MLP)

The MLP is a fundamental type of NN: it consists of three types (input, hidden, output) of fully-connected layers such that information flows forward from the inputs to the output.



Perceptron - Formal

Operation of a single unit:

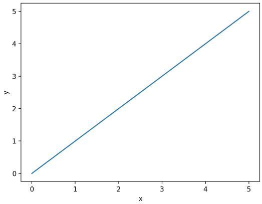
$$\begin{cases} z(\boldsymbol{x}) := \boldsymbol{w}^T \boldsymbol{x} + \boldsymbol{b} \\ h(\boldsymbol{x}) := f(z(\boldsymbol{x})) \end{cases}$$

with z the **net input** to the neuron, $\boldsymbol{b}$ is the **bias** and f is the **activation function**.

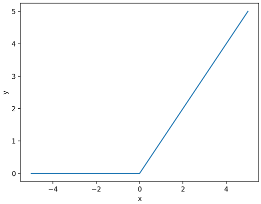
Examples of activation functions:

- 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): $f(t) := \frac{e^{2t}-1}{e^{2t}+1}$

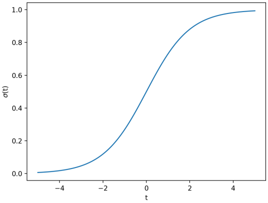
Activation functions - Plots



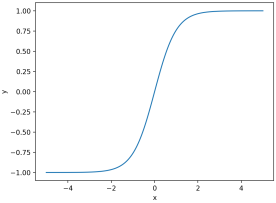
(a) Linear



(b) ReLU



(c) Sigmoid



(d) Tanh

Activation functions - Features

- ReLU: excellent default choice (easy to optimize because they are similar to linear units), the derivative remains large when active, disregard the non-differentiability
- Sigmoid: saturates when the argument is either very positive or very negative $\rightsquigarrow$ gradient-based learning may be hard, better not to use them as hidden units unless appropriate cost function can undo the saturation in the output layer (when output is a probability)
- Tanh: performs better than sigmoid when the latter must be used, similar to the identity near 0, composition of two tanh resembles a linear model as long as the argument is small (easier training)

MLP representation - Formal

Notation:

- $\boldsymbol{a}^{(j)}$ is the output of the j-th layer
- $\mathrm{W}^{(j)}$ is the weight matrix for the inputs of the j-th layer
- m is the number of layers (including input and output)

For each layer $j = 1, \ldots, m-1$ compute:

$$\begin{cases} \boldsymbol{z}^{(j)}(\boldsymbol{a}^{(j-1)}) := \mathrm{W}^{(j)}\boldsymbol{a}^{(j-1)} + \boldsymbol{b}^{(j)}, \\ \boldsymbol{a}^{(j)} := h^{(j)}(\boldsymbol{a}^{(j-1)}) = f^{(j)}(\boldsymbol{z}^{(j)}(\boldsymbol{a}^{(j-1)})). \end{cases}$$

with $\boldsymbol{a}^{(0)} = \boldsymbol{x}$ (notice: no 1 in the first entry) and $\boldsymbol{a}^{(m-1)}$ is the output of the network.

MLP representation - Multiple samples

For each layer $j = 1, \ldots, m-1$ compute:

$$\begin{cases} \mathrm{Z}^{(j)}(\mathrm{A}^{(j-1)}) := \mathrm{A}^{(j-1)}\mathrm{W}^{(j)} + (\boldsymbol{b}^{(j)})^T, \\ \mathrm{A}^{(j)} := h^{(j)}(\mathrm{A}^{(j-1)}) = f^{(j)}(\mathrm{Z}^{(j)}(\mathrm{A}^{(j-1)})). \end{cases}$$

with $\mathrm{A}^{(j)}$ the *matrix* of the outputs of the j-th layer (rows correspond to samples) and $\mathrm{A}^{(0)} = \mathrm{X}$ (**without** the column of ones). Here, $(\boldsymbol{b}^{(j)})^T$ is a *row vector* containing the biases (use *broadcasting* to extend it to N samples).

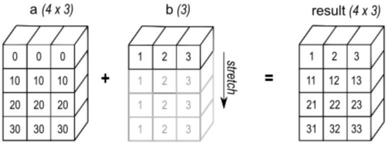


Figure: Array broadcasting in NumPy.

MLP representation - Example

Assume single sample, 3 inputs, 1 hidden layer/4 units, 1 output:

- $\boldsymbol{x}$ is a 3x1 vector
- $\boldsymbol{b}^{(1)}$ is a 4x1 vector
- $\mathrm{W}^{(1)}$ is a 4x3 matrix (each row contains the weights relative to a unit of the hidden layer)
- $\boldsymbol{z}^{(1)}$ is a 4x1 vector (each component corresponding to the net input of a unit of the hidden layer)
- $\boldsymbol{a}^{(1)}$ is a 4x1 vector (each component corresponding to the output of a unit of the hidden layer)
- $\mathrm{W}^{(2)}$ is a 1x4 matrix
- $\boldsymbol{b}^{(2)}$ is a 1x1 vector
- $\boldsymbol{a}^{(2)} = \boldsymbol{z}^{(2)}$ is a 1x1 vector (output)

Calculations with component notation:

$$\boldsymbol{z}_i^{(1)} = \mathrm{W}_{ik}^{(1)} \boldsymbol{x}_k + \boldsymbol{b}_i^{(1)}, \quad i = 1, 2, 3, 4$$

$$\boldsymbol{a}_i^{(1)} = h_i^{(1)}(\boldsymbol{z}^{(1)}), \quad i = 1, 2, 3, 4$$

$$\boldsymbol{a}_1^{(2)} = \boldsymbol{z}_1^{(2)} = \mathrm{W}_{1k}^{(2)} \boldsymbol{a}_k^{(1)} + \boldsymbol{b}_1^{(2)}$$

Learning XOR with an MLP

Architecture: 1 hidden layer containing 2 ReLU units.

Call $\mathrm{W} = \mathrm{W}^{(1)}$, $\boldsymbol{b}^{(1)} = \boldsymbol{b}$ and $\boldsymbol{w} = \mathrm{W}^{(2)}$. Set $\boldsymbol{b}^{(2)} = \mathbf{0}$.

A solution to the problem is:

$$\mathrm{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad \boldsymbol{b} = \begin{bmatrix} 0 \\ -1 \end{bmatrix} \quad \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$

Indeed, for the set of inputs

$$\mathrm{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

the output $\boldsymbol{w}^T \max\{0, \mathrm{XW} + \boldsymbol{b}^T\}$ is $[0, 1, 1, 0]^T$.

Exercise: try to solve the problem using *Linear Regression*.

MLP - Features

- This type of NN is also called **feedforward NN** because information flows from input to output without feedback
- The hypothesis function is *non-convex* because composition of convex functions is not necessarily convex
- Theory tells us that one-layer MLPs are **universal approximators**, *i.e.* they approximate *any continuous function* with any desired accuracy (not a formal statement), even though the layer may be infeasible large and may fail to learn and generalize correctly

Tips and Tricks - Is one layer really enough?

Theory suggests us that the answer is yes, but pay attention: *an exponential number of hidden units* (w.r.t. the input dimension) may be needed to approximate well the data, *i.e.* one hidden unit for each input configuration that needs to be distinguished.

- Empirically, increasing the *depth* results in better generalization for a wide variety of tasks (even though training is harder)
- Try different architectures in the model selection

Back-propagation

For network training via gradient descent, we need to compute the gradient of the cost with respect to the weights and biases.
We use **back-propagation**, which allows the information from the cost to then flow backward through the network.

- NNs are represented as **computational graphs**
- the *chain rule of Calculus* is used to compute derivatives by composing operations in a specific order that is highly efficient

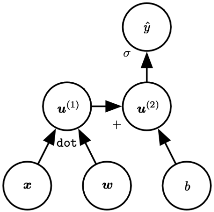


Figure: Example of computational graph of the function $\hat{y} = \sigma(\boldsymbol{w}^T\boldsymbol{x} + b)$.

Back-propagation - Example

- *Forward pass*: Compute the output E given the inputs $\boldsymbol{x}$ following the operations of the graph.

$$
\begin{aligned}
u^{(1)} &= \boldsymbol{w}^T \boldsymbol{x} \\
u^{(2)} &= u^{(1)} + b \\
\hat{y} &= \sigma(u^{(2)}) \\
E &= \mathrm{MSE}(\hat{y} - y) = (\hat{y} - y)^2
\end{aligned}
$$

- *Backprop*: For each operation (node) in the graph starting from the output and going backward, compute the gradient of the output E with respect to the inputs of that operation and propagate this information to the *parents* of the graph node to eventually compute the derivatives of E with respect to weights $\boldsymbol{w}$ and bias b.

$$
\begin{aligned}
\frac{\partial E}{\partial \hat{y}} &= 2(\hat{y} - y) \\
\frac{\partial E}{\partial u^{(2)}} &= \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial u^{(2)}} = 2(\hat{y} - y)\sigma'(u^{(2)}) \\
\frac{\partial E}{\partial u^{(1)}} &= \frac{\partial E}{\partial u^{(2)}} \frac{\partial u^{(2)}}{\partial u^{(1)}} = 2(\hat{y} - y)\sigma'(u^{(2)}) \\
\frac{\partial E}{\partial b} &= \frac{\partial E}{\partial u^{(2)}} \frac{\partial u^{(2)}}{\partial b} = 2(\hat{y} - y)\sigma'(u^{(2)}) \\
\frac{\partial E}{\partial \boldsymbol{w}} &= \frac{\partial E}{\partial u^{(1)}} \frac{\partial u^{(1)}}{\partial \boldsymbol{w}} = 2(\sigma(\boldsymbol{w}^T \boldsymbol{x} + b) - y)\sigma'(\boldsymbol{w}^T \boldsymbol{x} + b)\boldsymbol{x}
\end{aligned}
$$