

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 continuous 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

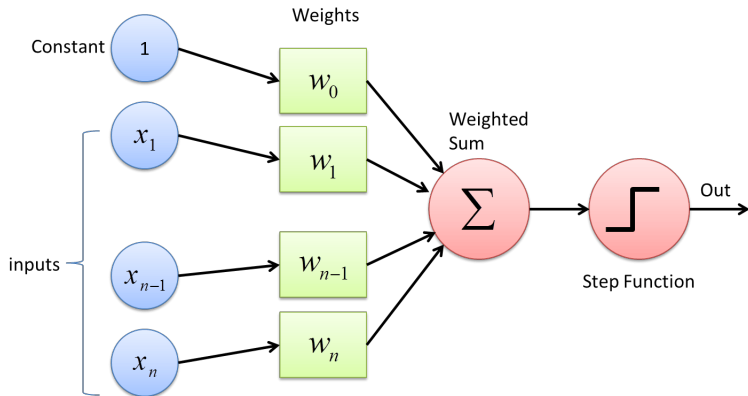
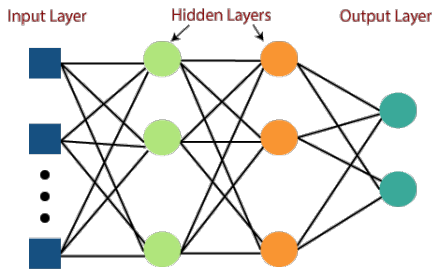


Figure: Representation of a perceptron.

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:

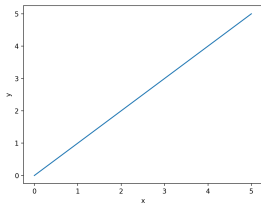
$$\begin{cases} z(\mathbf{x}) := \mathbf{w}^T \mathbf{x} + \mathbf{b} \\ h(\mathbf{x}) := f(z(\mathbf{x})) \end{cases}$$

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

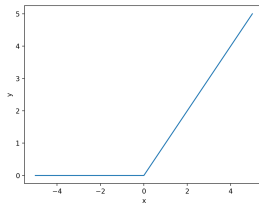
Examples of activation functions:

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

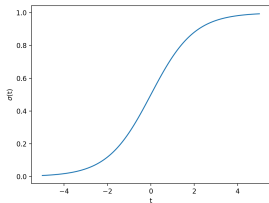
Activation functions - Plots



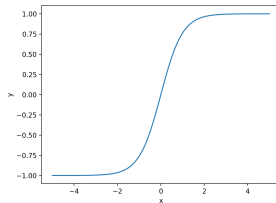
(a) Linear



(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 \leadsto 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:

- ▶ $\mathbf{a}^{(j)}$ is the output of the j -th layer
- ▶ $W^{(j)}$ 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 $j = 1, \dots, m - 1$ compute:

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

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

MLP representation - Multiple samples

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

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

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

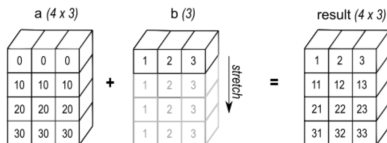


Figure: Array broadcasting in NumPy.

MLP representation - Example

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

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

Calculations with component notation:

$$z_i^{(1)} = W_{ik}^{(1)} x_k + b_i^{(1)}, \quad i = 1, 2, 3, 4$$

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

$$a_1^{(2)} = z_1^{(2)} = W_{1k}^{(2)} a_k^{(1)} + b_1^{(2)}$$

Learning XOR with an MLP

Architecture: 1 hidden layer containing 2 ReLU units.

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

A solution to the problem is:

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

Indeed, for the set of inputs

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

the output $\mathbf{w}^T \max\{0, XW + \mathbf{b}^T\}$ 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

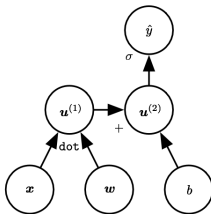


Figure: Example of computational graph of the function $\hat{y} = \sigma(\mathbf{w}^T \mathbf{x} + b)$.

Back-propagation - Example

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

$$u^{(1)} = \mathbf{w}^T \mathbf{x}$$

$$u^{(2)} = u^{(1)} + b$$

$$\hat{y} = \sigma(u^{(2)})$$

$$E = \text{MSE}(\hat{y} - y) = (\hat{y} - y)^2$$

- *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 \mathbf{w} and bias b .

$$\frac{\partial E}{\partial \hat{y}} = 2(\hat{y} - y)$$

$$\frac{\partial E}{\partial u^{(2)}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial u^{(2)}} = 2(\hat{y} - y) \sigma'(u^{(2)})$$

$$\frac{\partial E}{\partial u^{(1)}} = \frac{\partial E}{\partial u^{(2)}} \frac{\partial u^{(2)}}{\partial u^{(1)}} = 2(\hat{y} - y) \sigma'(u^{(2)})$$

$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial u^{(2)}} \frac{\partial u^{(2)}}{\partial b} = 2(\hat{y} - y) \sigma'(u^{(2)})$$

$$\frac{\partial E}{\partial \mathbf{w}} = \frac{\partial E}{\partial u^{(1)}} \frac{\partial u^{(1)}}{\partial \mathbf{w}} = 2(\sigma(\mathbf{w}^T \mathbf{x} + b) - y) \sigma'(\mathbf{w}^T \mathbf{x} + b) \mathbf{x}$$

Tips and Tricks - NN best practices

- ▶ Initialize all weights *randomly* near zero (or use other initialization techniques), but **do not** initialize uniformly to *break symmetry* and avoid *vanishing/exploding gradients*
- ▶ If a learning algorithm is suffering from *high bias*, getting more training data will not (by itself) help much. Instead, if a learning algorithm is suffering from *high variance*, getting more training data is likely to help
- ▶ Fewer features fixes *high variance* but not high bias; additional features fixes *high bias* but not high variance
- ▶ In general, use regularization to counter overfitting
- ▶ Small (big) NNs are more prone to underfitting (overfitting); use cross-validation to select network size

Basic Recipe for Machine Learning

- ▶ If your model has a high bias:
 - ▶ Try to make your NN bigger (number of hidden units, number of layers)
 - ▶ Try a different model that is suitable for your data
 - ▶ Try to increase the number of epochs
- ▶ If your model has a high variance:
 - ▶ More data
 - ▶ Try regularization
 - ▶ Try a different model that is suitable for your data

You should try the previous two points until you achieve low bias and low variance.