Perceptron and Activation Functions

TIN0255/09P9M80/D82 - Aprendizagem Profunda

2025.1 - Prof. Pedro Moura

Biological Neuroanatomy

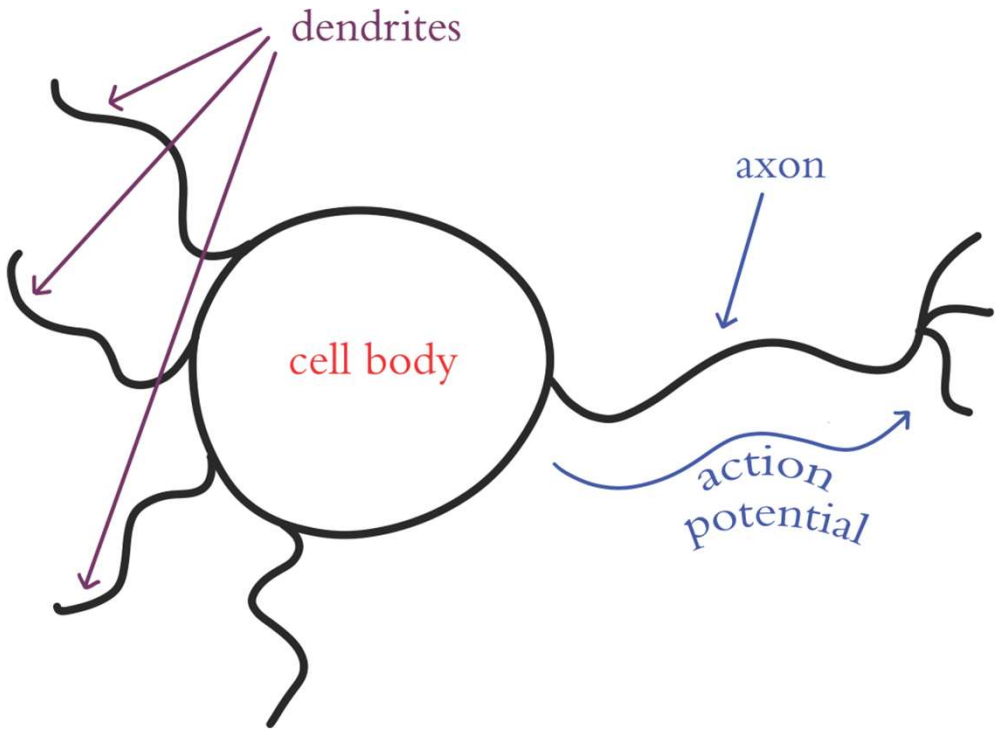
- As we saw previously, artificial neurons are inspired by biological ones.
- In this sense, we saw that a biological neuron receives inputs into its **cell body** from many **dendrites**.
- When the signal conveyed along a dendrite reaches the cell body → a small change in the voltage of the cell body.
- Some of them cause a positive change, other cause a negative change.

Biological Neuroanatomy

- Cumulative effect causes the voltage to increase from resting state (-70 millivolts) to the critical **threshold** of (-55 millivolts)
  - The **neuron fires** something called an **action potential** away from its cell body down its **axon**

- In sum:
  - Receive information from many other neurons
  - Aggreate this information via changes in cell voltage
  - Transmit a signal if the cell voltage crosses a threshold level

Biological Neuroanatomy



The anatomy of a biological neuron.

The Perceptron

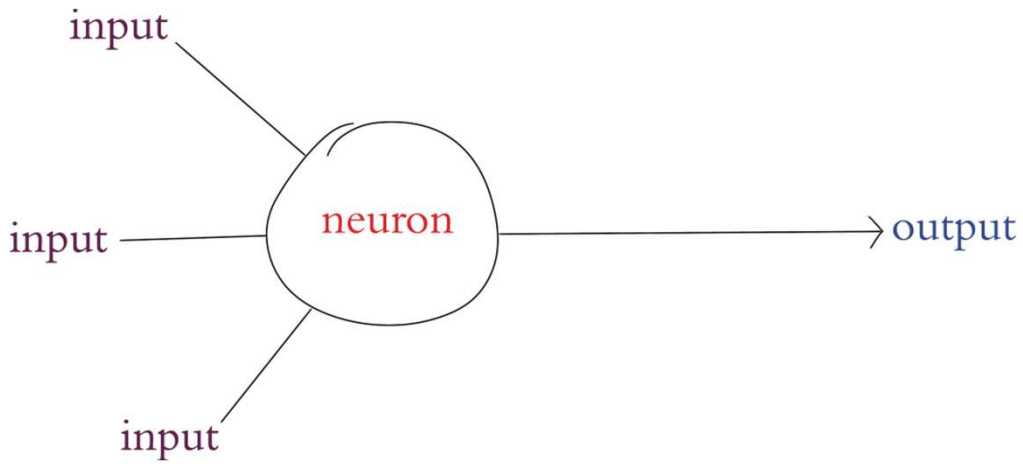
- In the late 1950s, the American neurobiologist Frank Rosenblatt published an article on **perceptron**.
  - An algorithm influenced by his understanding of biological neurons

- Analogous to its living inspiration, the **perceptron** can:
  - Receive input from many other neurons
  - Aggreate those inputs via a simple arithmetic operation called the **weighted sum**
  - Generate an output if this weighted sum crosses a threshold level, which can then be sent on to many other neurons

The Perceptron

The American neurobiology and behavior researcher Frank Rosenblatt.

The Perceptron

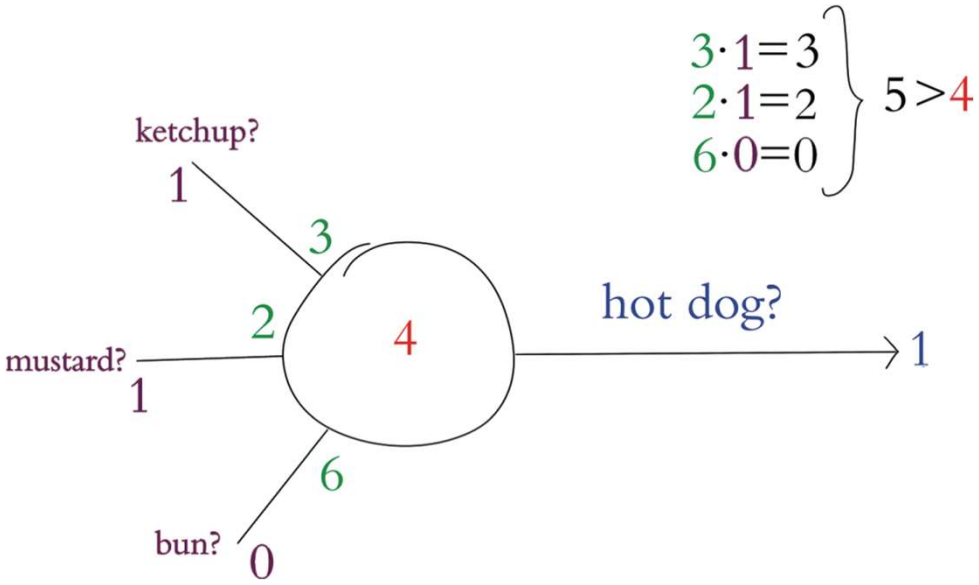


Schematic diagram of a **perceptron**, an early artificial neuron.

The Hot Dog Detector

- Let's work through a lighthearted example to understand how the perceptron algorithm works.
- We are going to look at a perceptron that is specialized in distinguishing whether a given object is a **hot dog** or **not a hot dog**.
- A critical attribute of a perceptrons is that they can only be fed **binary information** as inputs.
  - The perceptron independently **weights** each of these inputs.
- And their output is also restricted to being binary.

The Hot Dog Detector



First example of a hot dog-detecting perceptron: In this instance, it predicts there is indeed a hot dog.

The Hot Dog Detector

- To make a prediction as to whether the object is a hot dog or not, the perceptron independently **weights** each input.

- Let's determine the **weighted sum** of the inputs:
  - For the *ketchup*: 3 x 1 = 3
  - For *mustard*: 2 x 1 = 2
  - For *bun*: 6 x 0 = 0

- We can compute that the **weighted sum** is 5.

The Hot Dog Detector

- To generalize from this example, the calculation of the **weighted sum** of inputs is:

$$\sum_{i=1}^{n} w_i x_i$$

- w_i is the weight of a given input *i*
- x_i is the value of a given input *i*

The Hot Dog Detector

- The final step of the percetron algorithm is to evaluate whether the weighted sum of the inputs is greater than the neuron's **threshold**.

$$\sum_{i=1}^{n} w_i x_i \begin{cases} > threshold, \text{output } 1 \\ \leqslant threshold, \text{output } 0 \end{cases}$$

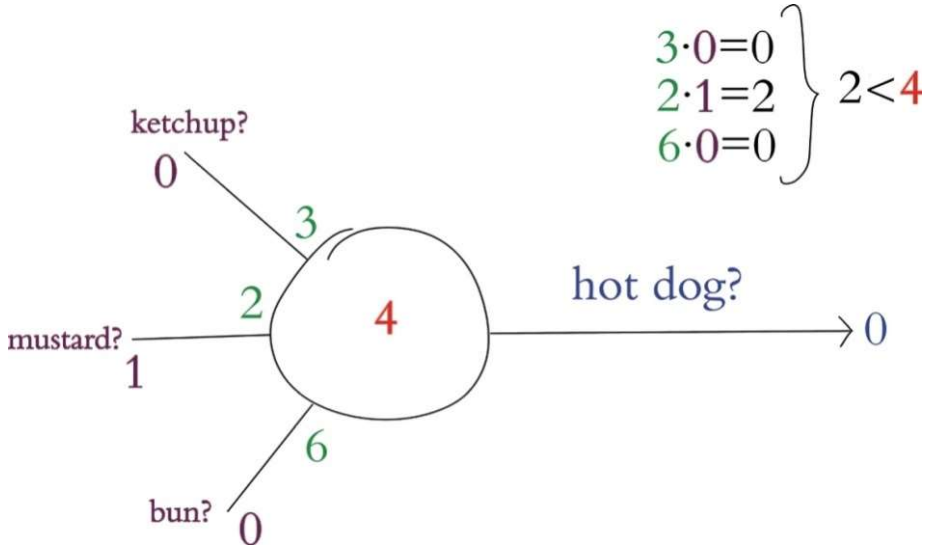
The Hot Dog Detector

- Knowing this, we can wrap up our last example.
  - The weighted sum of **5** is greater than the neuron's **threshold**, and so our hot dog-detecting perceptron outputs 1.

$$\sum_{i=1}^{n} w_i x_i \begin{cases} > \textit{threshold}, \text{output } 1 \\ \leqslant \textit{threshold}, \text{output } 0 \end{cases}$$

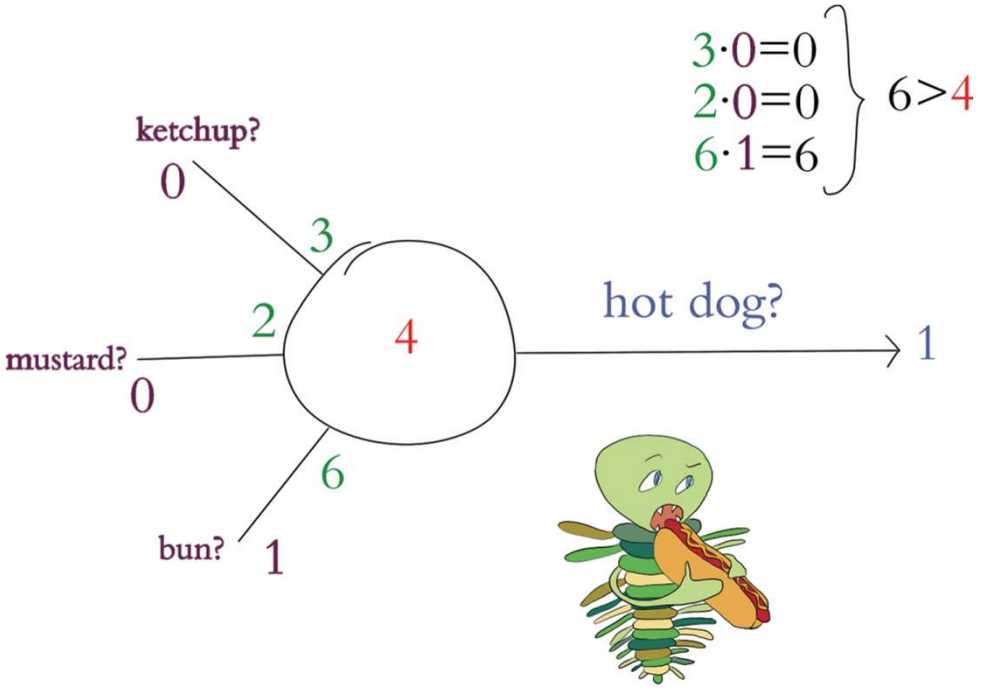
- Now let's see an object evaluated by the perceptron that includes mustard only.
  - It predicts this object is **not** a hot dog.

The Hot Dog Detector



Second example of a hot dog-detecting perceptron: in this instance, it predicts there is not a hot dog.

The Hot Dog Detector



Another example of a hot dog-detecting perceptron. What is going to happen here?

The Most Important Equation

- To achieve the formulation of a simplified an universal perceptron.
  - We introduce the **bias b** and which is given by:

$$b \equiv -threshold$$

- Together, a neuron's bias and its weights constitute all of its **parameters**.

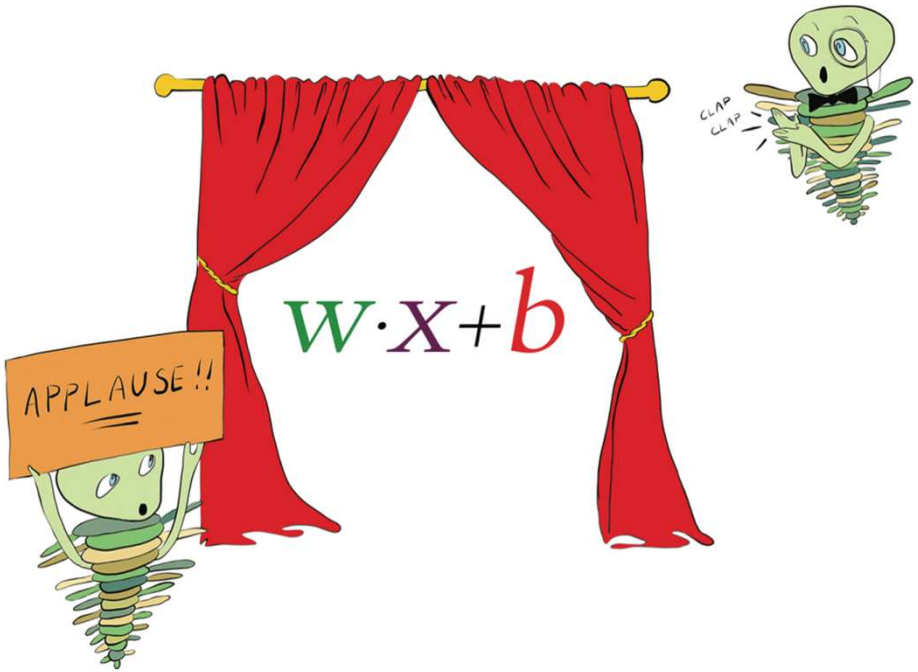
The Most Important Equation

- We now arrive at the most widely used perceptron equation:

$$\text{output} \begin{cases} 1 \text{ if } \boldsymbol{w} \cdot \boldsymbol{x} + \boldsymbol{b} > 0 \\ 0 \text{ otherwise} \end{cases}$$

- We substituted **b** in place of the neuron's threshold.
- Flipped **b** onto the same side of equation as all other variables.
- Used the **dot product** notation.

The Most Important Equation



The most important equation when learning Artificial Neural Networks.

Modern Neurons and Activation Functions

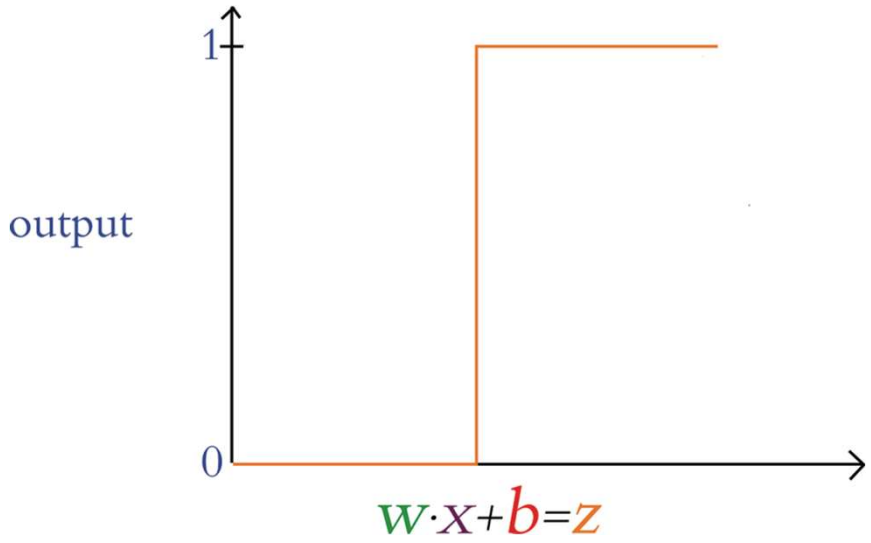
- Modern artificial neurons are not perceptrons.
  - Although simple, it is not used widely today.
- Two main restrictions:
  - It receives only **binary** inputs and provides only a **binary** output.
  - It only solves linearly separable problems.
- In many cases, we want to make predictions from inputs that are continuous variables.

Modern Neurons and Activation Functions

- A less obvious corollary of its restriction: it makes learning rather challenging.

- Let's create a variable **z**, such as **z = w . x + b**.

Modern Neurons and Activation Functions



The perceptron's transition from outputting zero to outputting one happens suddenly, making it challenging to gently tune w and b to match a desired output.

Modern Neurons and Activation Functions

- When **z** is any value less than or equal to zero, the perceptron outputs its smallest possible output: **0**.
- If **z** becomes positive to even the tiniest extent, the perceptron outputs its largest possible output, **1**.
- This sudden and extreme transition is not optimal during training
  - We make slight adjustments to **w** and **b** based on whether it appears the adjustment will improve the network's **output**.

Modern Neurons and Activation Functions

- With perceptron: the majority of adjustments makes no difference to **w** and **b**
  - **z** would generally be moving around at lower negative values lower than 0 or at positives values higher than **0**.
- Situation is even worse:
  - Slight adjustment to **w** or **b** will cause **z** to cross from negative to positive (or vice versa)
  - Leads to drastic swing int output from **0** all the way to **1** (or vice versa)
- **Percepton has no finesse: it's either yelling or it's silent.**

Modern Neurons and Activation Functions

- The approach we shall use to learn good parameter values for the network is **gradient descent**.
  - We then need functions that “play well” with it.

- Desired properties:

  1) The function is **continuous** and **differentiable** everywhere (or almost everywhere);

  1) The derivative of the function **does not saturate** (tends towards zero) over its expected input range.
     - Very small derivatives tend to stall out the learning process.

  1) The derivative **does not** explode (tends towards infinity), since this would lead to issues of numerical instability.

Modern Neurons and Activation Functions

- The step function does not satisfy conditions (2) and (3).
- Its derivative explodes at 0 and is 0 everywhere else.
- **Step function does not play well with gradient descent and is not a good choice for a deep neural network.**

The Sigmoid Neuron

- Alternative to the erratic behavior of the perceptron?
  - A gentle curve from 0 to 1.
- It is called the **sigmoid function**.

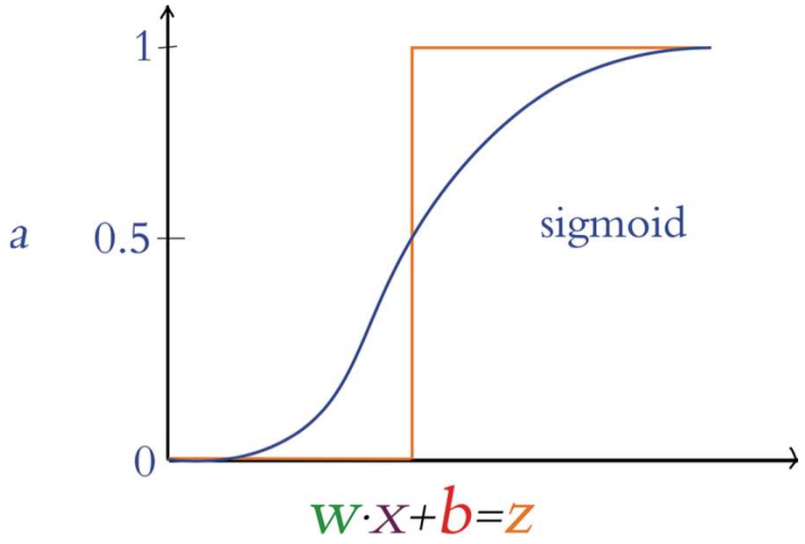
$$\sigma(z) = \frac{1}{1+e^{-z}}$$

- **z** is equivalent to **w . x + b**.
- It is an **activation function**.

The Sigmoid Neuron

- It is the canonical **activation function**.
  - Such that σ is generally used to denote **any** activation function.
- The output of any neuron's activation function is referred as its **activation**.
  - We use the term **a**.
- There's no need to memorize the sigmoid function.
  - We focus on understanding its behavior.
  - Let's play a bit!
- Increasingly large positive (negative) inputs will result in values that approach **1**/**0**.

The Sigmoid Neuron



The sigmoid activation function.

The Sigmoid Neuron

- Any artificial neuron that features the sigmoid function: **sigmoid neuron**.
- Advantage over the perceptron: small changes in a given sigmoid neuron's parameters **w** or **b** cause small changes in **z**, thereby producing similarly gradual changes in the neuron's activation.
- Exception: large negative or positive values → it will output **0**/**1**.
- **Saturation** still happens → subtle updates to the weights and biases during training will have little to no effect on the output.

The Sigmoid Neuron

- Sigmoid **saturates very quickly** as we move away from the **"critical region" around 0**.
- The **derivative goes towards zero** and gradient-based learning can stall out.
- **Weights almost stop changing**, once they get away from 0.
- Its value is in the **range [0, 1]** → it is possible to interpret the **outputs of the network as a probability**.

The Sigmoid Neuron

- Later, when we learn backpropagation, we shall see that **we need derivatives of activation functions and loss functions**.

- **Exercise** (13.2.1 from "Mining of Massive Datasets"): Show that for the logistic sigmoid σ, if $y = \sigma(x)$, then

$$\frac{dy}{dx} = y(1-y)$$

The Sigmoid Neuron

- Sigmoid function tends to vanish gradient.
- From the previous exercise, it is possible to see that its derivative has the following form:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

- When *x* grows to infinite large, $\sigma(x)$ tends to 1. Therefore:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$
$$= 1 \times (1 - 1) = 0$$

The Tanh Neuron

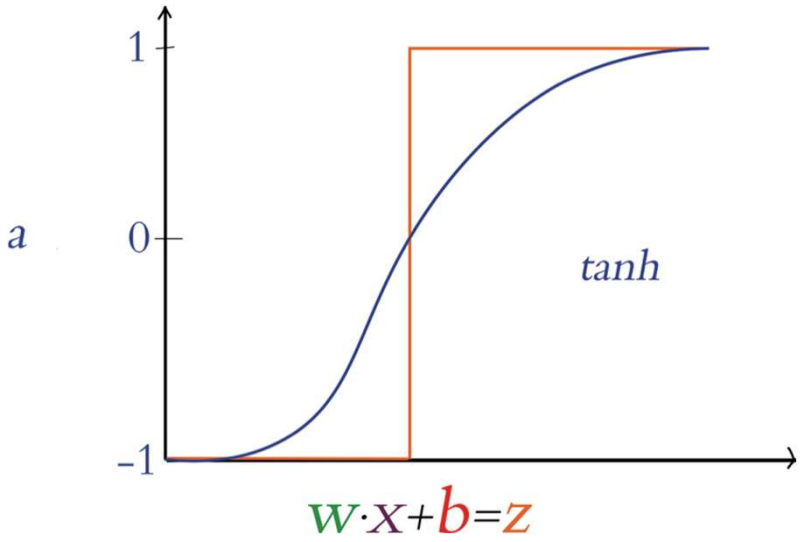
- A popular cousin of the sigmoid neuron is the **tanh neuron**.

$$\sigma(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

- The shape of the tanh curve is similar to the sigmoid curve, except that its output has the range [-1, 1].

- Its output tends to be centered near **0**.

The Tanh Neuron



The tanh activation function.

The Tanh Neuron

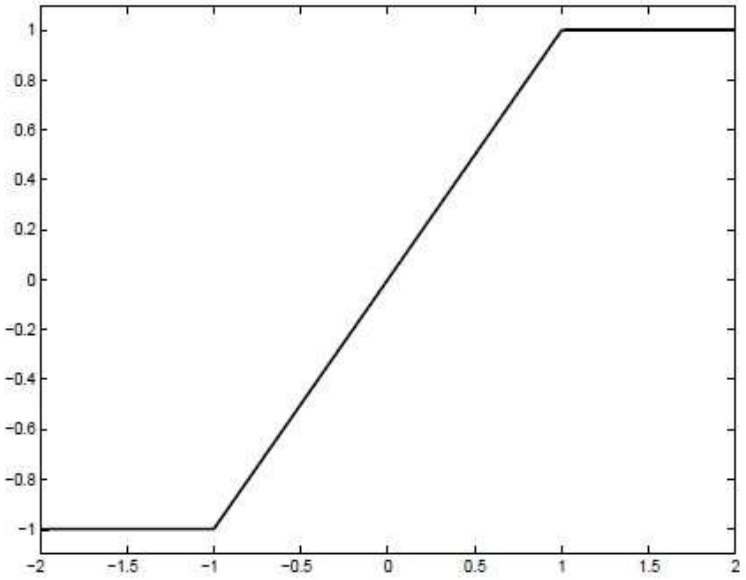
- It is easy to see that the two functions are **identical after shifting and scaling**.

- **Exercise** (13.2.2 from “Mining of Massive Datasets”): Show that if *y* = tanh(*x*) then

$$\frac{dy}{dx} = 1 - y^2$$

Hard Tanh Function

- It's a **piecewise version** of tanh.
- **It's much easier to compute!**



(b) Hard Tanh

$$\Phi(v) = \max\{\min[v, 1], -1\}$$

Hard Tanh Function

- Tanh is preferable to the sigmoid when the outputs of the computations are desired to be **both positive and negative**.
- **Its mean-centering and larger gradient (because of stretching) with respect to sigmoid makes it easier to train.**
- Sigmoid and Tanh have been the historical tools of choice for incorporating nonlinearity in the ANN.

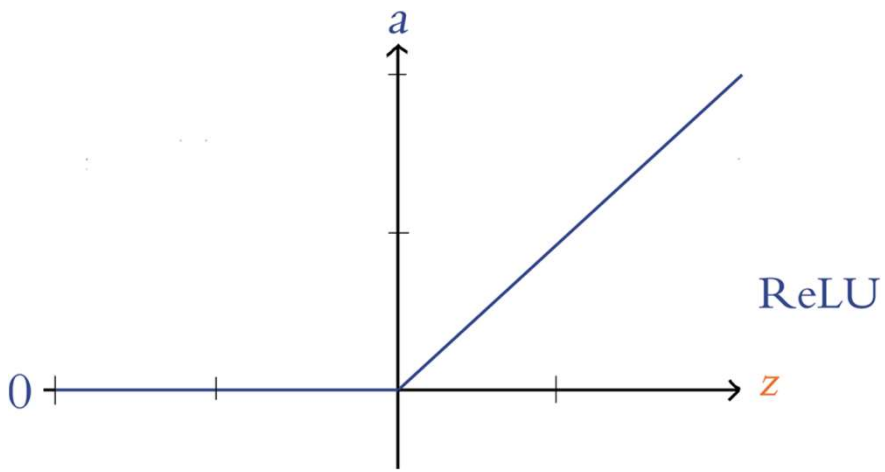
ReLU: Rectified Linear Units

- Another neuron is the **rectified linear unit**, or **ReLU neuron**.

$$a = max(0, z)$$

- Its shapes diverges glaringly from the sigmoid and tanh sorts.
  - It was inspired by properties of biological neurons and popularized within artificial neural networks by Vinod Nair and Geoff Hinton.

ReLU: Rectified Linear Units



The ReLU activation function.

ReLU: Rectified Linear Units

- It is one of the simplest functions to imagine that is **nonlinear**.
  - Its output does not vary uniformly linearly across all values of **z**.
  - It is in essence **two** distinct linear functions combined (one at negative **z** values, and the other at positive **z** values).
- This nonlinear nature is a **critical property** of all activation functions used within deep learning architectures.
- **These nonlinearities permit deep learning models to approximate any continuous function.**
- The relatively simple shape of the ReLU function's particular brand of nonlinearity works to its advantage.

ReLU: Rectified Linear Units

- Its name derives from the analogy to *half-wave rectification* in electrical engineering.
- It is not differentiable at 0 but it is differentiable everywhere else.
  - Including at points arbitrarily close to 0.
- In practice, we “set” the derivative at 0 to be either 0 (the left derivative) or 1 (the right derivative).
- It is the default choice in modern neural networks, replacing the sigmoid function.

ReLU: Rectified Linear Units

- **The popularity of ReLU derives from two properties:**

- 1. The gradient of ReLU remains constant and never saturates for positive **x**, speeding up training.
  - It has been found in practice that networks that use ReLU offer a significat speedup in training compared to sigmoid activation.

- 2. Both the function and its derivative can be computed using **elementary and efficient mathematical operations** (no exponentiation).

ReLU: Rectified Linear Units

- **Problem:** ReLU does suffer from a problem related to saturation of its derivative when **z** < 0.
- Once a neuron's input values become negative, it is possible that the neuron's output get "stuck" at 0 through the rest of the training.
- This is called the **dying ReLU problem**.
- If too many **z**'s get below zero then most of the neurons in network will simply output zero, i.e., **they die and thereby prohibiting learning**.
- **This can be handled, to some extent, by using Leaky-Relu.**

ReLU: Rectified Linear Units

- **Another problem: ReLU tends to blow up activation**.
- There is no mechanism to constrain the output of the neuron (as x itself is the output when $x > 0$).
- Large activations tend to highly influence neurons in the next layers: **Undesirable!**

ReLU: Rectified Linear Units

- There is a veritable zoo of activation functions available and the list is ever growing.

- We are now going to see some of the “advanced” activation functions, which are also provided by Keras, are:
  - **Leaky ReLU**
  - **Parametric ReLU**
  - **Exponential Linear Unit**

- All three of which are derivations from the ReLU function.

ReLU: Rectified Linear Units

- The **Leaky ReLU** attempts to fix this problem by defining the activation function as follows:

$$f(x) = \begin{cases} x, & \text{for } x \geq 0 \\ \alpha x, & \text{for } x < 0 \end{cases}$$

- **where α is an hyperparameter and it is typically a small positive value such as 0.01.**

- The **Parametric ReLU** (PreLU) makes alpha a **parameter to be optimized** as part of the learning process.

ReLU: Rectified Linear Units

- Improvement on both the original and leaky ReLU functions → **Exponential Linear Unit, or ELU**.

$$f(x) = \begin{cases} x, & \text{for } x \geq 0 \\ \alpha(e^x - 1), & \text{for } x < 0 \end{cases}$$

- where α is a **hyperparameter**.
  - **α is held fixed during the learning process, but we can repeat the learning process with different values of α to find the best value for our problem.**

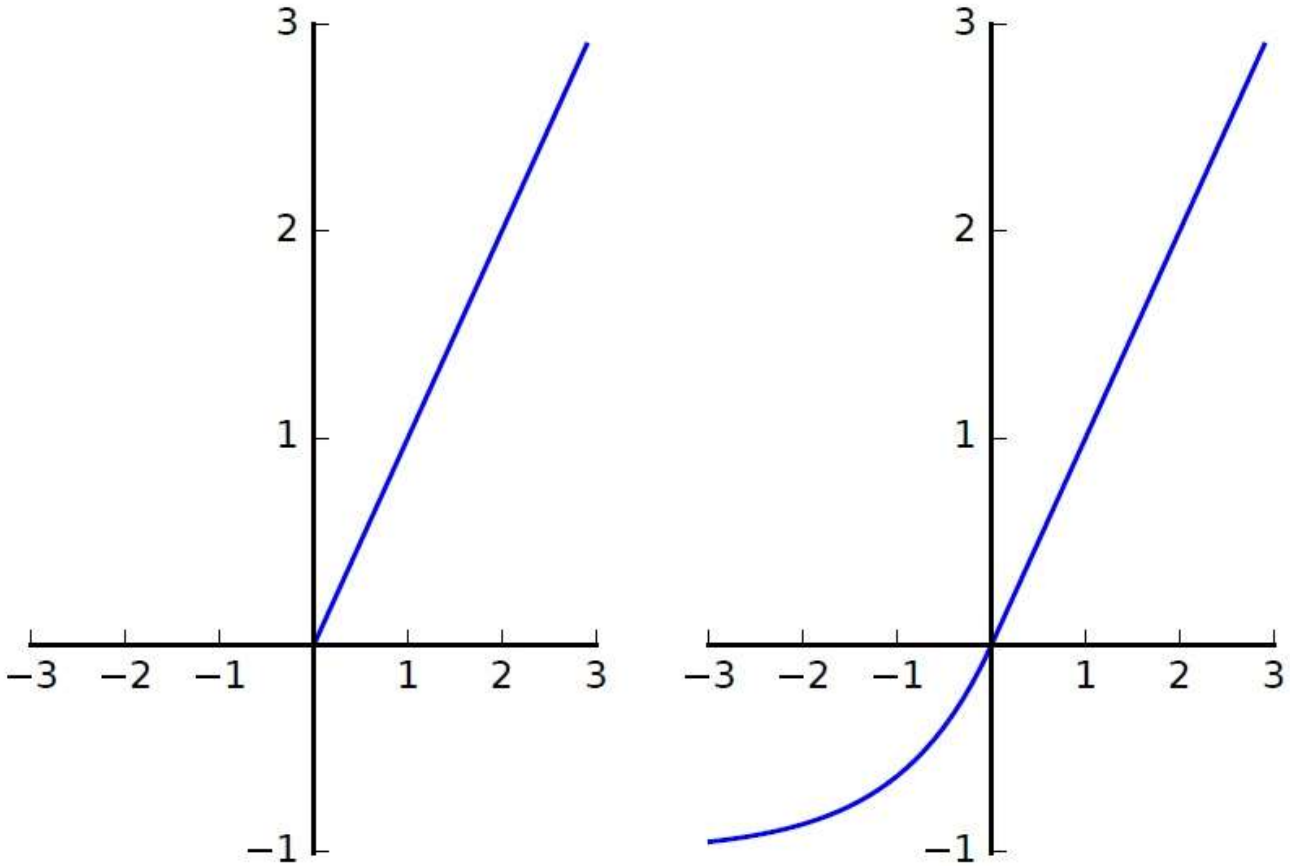
ReLU: Rectified Linear Units

- The node's value saturates to $-\alpha$ for large negative values of x, and a typical choice is $\alpha = 1$.

- **ELU's drive the mean activation of nodes towards zero, which appears to speed up the learning process compared to other ReLU variants.**

ReLU: Rectified Linear Units

- The ReLU (a) and ELU (b), with $\alpha = 1$ functions:



(a) (b)

Maxout Networks

- A recently proposed solution is the use of **Maxout units**.
- It leverages two coeficiente vectors $\bar{W}_1$ and $\bar{W}_2$ instead of a single one.
- The maxout unit computes the function $\max\{\bar{W}_1 \cdot \bar{X}, \bar{W}_2 \cdot \bar{X}\}$.
  - In case bias neurons are used, the maxout activaction is $\max\{\bar{W}_1 \cdot \bar{X} + b_1, \bar{W}_2 \cdot \bar{X} + b_2\}$.
  - It is **piecewise linear**.
- **It is a genelization of the ReLu, because the ReLU is obtained by setting one of the coeficient vectors to 0.**
- Even the Leaky ReLU can be shown to be a special case of maxout: we set $\bar{W}_2 = \overline{\alpha W_1}$ for $\alpha \in (0,1)$.

Maxout Networks

- **However, it does not saturate at all, and is linear almost everywhere.**
- (Goodfelllow *et* al., 2013) have shown that maxout networks are **universal function approximators**.
- **Maxout has advantanges over the ReLU**, and it enhances the performance of ensemble methods like **Dropout** (which we will learn later).
- **One drawback of maxout is that it doubles the number of parameters.**

Considerations

- ReLU and hard tanh functions **have largely** replaced the sigmoid and soft tanh functions in modern networks.
  - **Ease in training multi-layered networks with these activation functions.**
- Generally, nonlinear activation functions are very → they help in creating more **powerful compositions** of diferente types of functions.
- These functions are referred to as **squashing** functions: they map the outputs from na arbitrary range to bounded outputs.

Considerations

- **The use of a nonlinear activation plays a fundamental role in increasing the modeling power of the network.**
- **Network with only linear activations** → it would not provide better modeling power than a single-layer network.
- We will discuss this in details later in our course.

Choosing a Neuron

- Within a given hidden layer → you are able to choose any activation function you fancy.

- Constraint that you should select a nonlinear function (to be able to approximate any continuous function), we are nevertheless left with quite a bit of room for choice.

- Let's rank the neuron types we've seen, from those we recommend least through to those we recommend most:
  - **Perceptron:** with its binary inputs and the aggressive step of its binary output, is not a practical consideration for deep learning models.
  - **Sigmoid:** is an acceptable option, but it tends to lead to neural networks that train less rapidly than those composed of, say, tanh or ReLU neurons. **Limit your use to situations where it would be helpful to have a neuron provide output within the range of [0, 1].**

Choosing a Neuron

- **Tanh:** is a solid choice, since its 0-centered output helps deep learning networks learn rapidly.
- **ReLU:** is preferred because of how efficiently these neurons enable learning algorithms to perform computations → they tend to lead to **well-calibrated** neural networks in the **shortest period of training time**.

Exercises

- **Exercise** (13.2.3 from “Mining of Massive Datasets”): Show that

$$\tanh(x) = 2\sigma(2x) - 1$$

- This shows that tanh function is a **re-scaled sigmoid function** with both horizontal and vertical stretching, as well as vertical translation (Exercise 3, Chapter 1, “Deep Learning and Neural Networks”).

Exercises

- **Exercise** (13.2.4 from “Mining of Massive Datasets”): Show that

$$\sigma(x) = 1 - \sigma(-x)$$

Exercises

- **Exercise** (1.10.2 “Neural Networks and Deep Learning”): Show the following properties of the sigmoid and tanh activation functions (denoted by $\sigma(.)$ in each case):

(a) Sigmoid activation: $\sigma(-x) = 1 - \sigma(x)$

(b) Tanh activation: $\sigma(-x) = -\sigma(x)$

(c) Hard tanh activation: $\sigma(-x) = -\sigma(x)$

Referências

1. Capítulo 6 do livro *"Deep Learning Illustrated: A Visual, Interactive Guide to Artificial Intelligence"* de Jon Krohn, Grant Beyleveld e Aglaé Bassens.

2. Capítulo 13 do livro *"Mining of Massive Datasets"* de Jure Leskovec, Anand Rajaraman e Jeffrey Ullman.

3. Capítulos 1 e 4 do livro "*Neural Networks and Deep Learning: A Textbook"* de Charu C. Aggarwal. 2ª ed.