Artificial neural networks

In the search of the brain

Natural Language Processing

Piotr Fulmański

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
University of Lodz

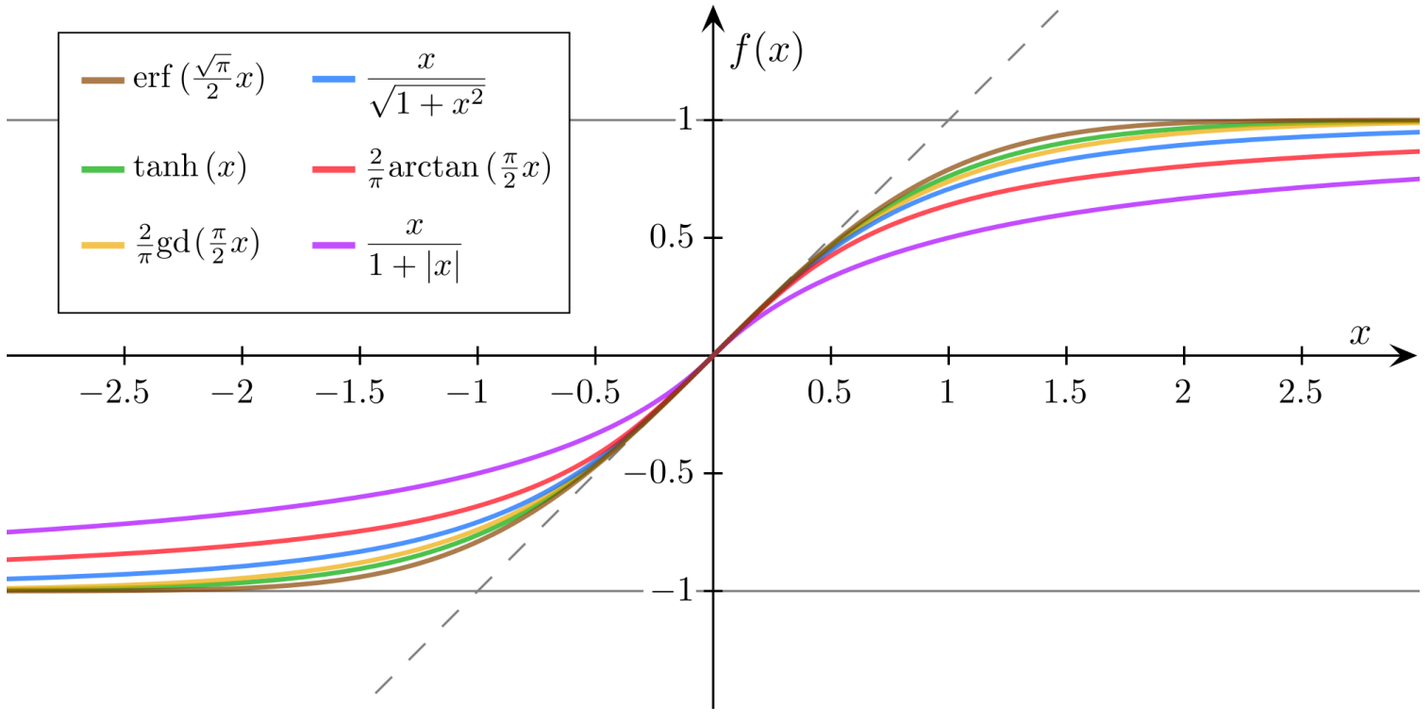
Lecture goals

- Sigmoid function, logistic function, softmax function
- Algorithmic method for searching a minimum of a function
- Artificial neural networks - fast introduction
- Interactive examples
- Practical examples

Some useful functions you have to know

Sigmoid function

A sigmoid function is a mathematical function having a characteristic "S"-shaped curve or sigmoid curve.



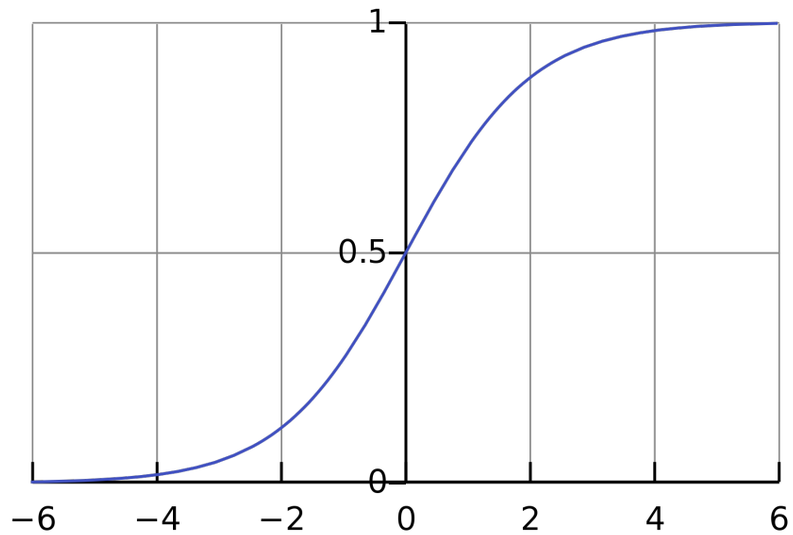
There are many functions of this type but all of them share the same set of properties. A sigmoid function is a bounded, differentiable, real function that is defined for all real input values and has a non-negative derivative at each point and exactly one inflection point.

You will focus on the one, given by formula:

$$f(x) = \frac{1}{1 + e^{\lambda(-x)}}$$

and called *logistic function*.

Logistic function



A logistic function or logistic curve is a curve (with equation:

$$f(x) = \frac{L}{1 + e^{-\lambda(x - x_0)}},$$

where:

- x_0 is the x value of the sigmoid's midpoint;
- L is the curve's maximum value;
- λ is the logistic growth rate or steepness of the curve.

The *standard logistic function* is the logistic function with parameters $\lambda = 1,\ x_0 = 0, L = 1,$

which yields:

$$f(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1} = \frac{1}{2} + \frac{1}{2}\tanh\left(\frac{x}{2}\right).$$

The standard logistic function has an easily calculated derivative (this derivative is known as the l*ogistic distribution)*:

$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) = \frac{e^x \cdot (1 + e^x) - e^x \cdot e^x}{(1 + e^x)^2} = \frac{e^x}{(1 + e^x)^2} = f(x)\big(1 - f(x)\big).$$

Because $1 - f(x) = f(-x)$ than you have:

$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) = f(x)f(-x)$$

For us the most important is this relation:

$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) = f(x)\big(1 - f(x)\big)$$

as it allows to calculate derivative if only value $f(x)$ is known. This is very important property allowing us to reduce number of computation in case of neural networks where first you propagate signals forward calculating neuron's activation and next you propagate error signals back calculating activation's derivative.

Logistic function

The **standard logistic function** with steepness parameter λ:

$$f(x) = \frac{1}{1 + e^{-\lambda x}}$$

with derivative:

$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) = \lambda f(x)\big(1 - f(x)\big)$$

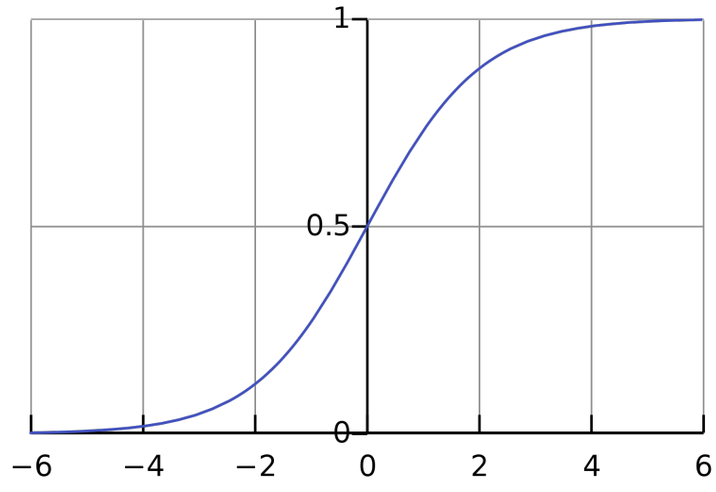
takes the values from interval $(0,1)$ which for some practical reasons is very undesirable feature. This function is sometimes called unipolar sigmoid function in contrast to its **bipolar** version of the form:

$$f(x) = \frac{2}{1 + e^{-\lambda x}} - 1$$

with derivative:

$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) = \lambda \frac{1}{2}\big(1 - (f(x))^2\big)$$

which takes the values from interval $(-1,1)$.



Both are often used as the activation function of an artificial neural network. Another frequent choice is:

$$f(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

with derivative:

$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) = 1 - \tanh^2(x).$$

Notice that:

$$\tanh(x) = \frac{2}{1 + e^{-2x}} - 1.$$

Softmax function

argmax

You can define $\arg\max(x), \arg\max : \mathbb{R}^n \to \mathbb{R}^n$ function as a one-hot representation of the argument x (assuming there is a unique max arg):

$$\arg\max(x_1, \ldots x_i, \ldots, x_n) = (y_1, \ldots, y_i, \ldots, y_n) = (0,\ldots,0,1,0,\ldots,0)$$

where the output coordinate $y_i = 1$ if and only if i is the index of the unique greatest (maximum) value of all $x_1, \ldots, x_n$.

For example:

$$\arg\max(4,8,11) = (0,0,1)$$

because the third argument is the maximum value of 4, 8 and 11.

This can be generalized to multiple $\arg\max$ values (multiple equal x_i being the maximum) by dividing the 1 by the total number of argument taking maximum value. For example:

$$\arg\max(4,3,2,4) = \left(\frac{1}{2},0,0,\frac{1}{2}\right) = (0.5,0,0,0.5)$$

because the first and fourth argument are both the maximum (of value 4). In case all arguments are equal, this is simply $\arg\max(x_1, \ldots, x_n) = \left(\frac{1}{n}, \ldots, \frac{1}{n}\right)$.

Softmax function

softmax

The $\operatorname{soft\,max}(x)$, $\operatorname{soft\,max} : \mathbb{R}^n \to \mathbb{R}^n$ function (also known as normalized exponential function) takes as input a vector x of n real numbers, and normalizes it into a *probability distribution* consisting of n probabilities proportional to the exponentials of the input numbers.

The standard (unit) $\operatorname{soft\,max}$ function is defined by the formula:

$$\operatorname{soft\,max}(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

for $i = 1,\ldots,n$ and $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$.

In words: you apply the standard exponential function to each element and normalize these values by dividing by the sum of all these exponentials; this normalization ensures that the sum of the components of the output vector is 1.

That is, prior to applying $\operatorname{soft\,max}$, even if some vector's components are negative or greater than one and all not sum to 1, after applying $\operatorname{soft\,max}$, each component will be in the interval $[0,1]$, and the components will add up to 1. In consequence, they *can be interpreted as probabilities*. Furthermore, the larger input components will correspond to larger probabilities.

Softmax function

softmax

```
import numpy as np
a = [4,8,11]
print( np.exp(a) / np.sum(np.exp(a)) )
b = [0.4,0.8,1.1]
print( np.exp(b) / np.sum(np.exp(b)) )


0.000867 0.047384 0.951747
0.221947 0.331106 0.446946
```

If you take an input of $(4,8,11)$, the $\mathrm{soft\,max}$ of that is $(0.000867,0.047384,0.951747)$. The output has most of its weight where the 11 was in the original input. This is what the function is normally used for: **to highlight the largest values and suppress values which are significantly below the maximum value.**

But be careful: $\mathrm{soft\,max}$ is not scale invariant, so if the input is divided by 10 to the form $(0.4,0.8,1.1)$ the $\mathrm{soft\,max}$ is $(0.221947,0.331106,0.446946)$. Note that in the second case largest value is not as much large compared to the second largest value as in the first example:

$$\frac{0.951747}{0.047384}100\,\% = 20.085 \cdot 100\,\% = 2008.5\,\%$$

$$\frac{0.446946}{0.331106}100\,\% = 1.3498 \cdot 100\,\% = 134.98\,\%$$

You can say that larger values create a probability distribution that is **more** concentrated around the positions of the largest input values.

Softmax function

softmax

You can say that larger values create a probability distribution that is **more** concentrated around the positions of the largest input values.

Sometimes $\operatorname{soft\,max}$ is defined as:

$$\operatorname{soft\,max}(x)_i = \frac{e^{\beta x_i}}{\sum_{j=1}^{n} e^{\beta x_j}}.$$

The greater β is, the probability distribution is **more concentrated around the positions of the largest input values**.

Softmax function

softmax

If you look closer to the first example, input of $(4,8,11)$ and its $\text{soft}\max$ of value $(0.000867,0.047384,0.951747)$ you can approximate the result by $(0,0,1)$.

Taking into account last definition of $\text{soft}\max$:

$$\text{soft}\max(x)_i = \frac{e^{\beta x_i}}{\sum_{j=1}^{n} e^{\beta x_j}}$$

it can be proved that as $\beta \to \infty$ $\text{soft}\max$ converges to $\arg\max$.

Other words, you can say that $\text{soft}\max$ is not a smooth maximum (a smooth approximation to the maximum function), but is rather a smooth approximation to the $\arg\max$ function and this is why sometimes it is called (correctly) $\text{soft}\arg\max$. The term *softmax* is used in machine learning.

Softmax function

Neural networks

The *softmax* function is used in various multiclass classification methods, such as:

- multinomial logistic regression (also known as softmax regression; a classification method that generalizes logistic regression to multiclass problems, i.e. with more than two possible discrete outcomes),

- multiclass linear discriminant analysis,

- naive Bayes classifiers,

- and **artificial neural networks**.

Because of its properties (interpretability as probability distribution) the softmax function is often used in the **final** layer of a neural network-based classifier. Such networks are commonly trained under a log loss (or cross-entropy) regime, giving a non-linear variant of multinomial logistic regression (a classification method that generalizes logistic regression to multiclass problems).

Softmax function

Derivative of softmax function

Now I will calculate derivative of softmax function as you will need it if you want to use it in neural networks (why you need derivatives I explain in the following part).

To simplify notation I will write:

$$S_i(x) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}.$$

Softmax is a vector function therefore, when you try to find its derivative, you talk about a Jacobian matrix, which is the matrix of all first-order partial derivatives of a vector-valued function:

$$\frac{\partial S}{\partial x} = \begin{bmatrix} \frac{\partial S_1}{\partial x_1} & \cdots & \frac{\partial S_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial S_n}{\partial x_1} & \cdots & \frac{\partial S_n}{\partial x_n} \end{bmatrix}.$$

Your goal is to compute $\frac{\partial S_i}{\partial x_k}$ for some arbitrary i and k. To make it simpler, apply first rule for calculating derivative of quotient: for $f(x) = \frac{g(x)}{h(x)}$ you have

$$f'(x) = \frac{g'(x)h(x) - g(x)h'(x)}{\big(h(x)\big)^2}.$$

In this case:

$$f(x) = S_i(x) = \frac{g_i(x)}{h(x)},$$

$$g_i(x) = e^{x_i},\ h(x) = \sum_{j=1}^{n} e^{x_j}$$

and you have to calculate

$$\frac{\partial}{\partial x_k} S_i(x) = \frac{\frac{\partial}{\partial x_k} g_i(x)h(x) - g_i(x)\frac{\partial}{\partial x_k} h(x)}{\big(h_i(x)\big)^2}.$$

Softmax function

Derivative of softmax function

You have to calculate $\frac{\partial}{\partial x_k} g_i(x)$ and $\frac{\partial}{\partial x_k} h(x)$.

Surprisingly calculating derivative of h is simpler than derivative of g_i. Computing the derivative of h with respect to x_k, no matter for which $k \in \{1,2,\ldots,n\}$, the answer will **always** be e^{x_k}

$$\frac{\partial h}{\partial x_k}(x) = \frac{\partial}{\partial x_k} \sum_{j=1}^{n} e^{x_j} = \sum_{j=1}^{n} \frac{\partial}{\partial x_k} e^{x_j} = e^{x_k}$$

because

$\frac{\partial}{\partial x_k} e^{x_j} = 0$ for all $j \neq k$ and

$\frac{\partial}{\partial x_k} e^{x_k} = e^{x_k}$ for $j = k$.

Softmax function

Derivative of softmax function

The derivative of g_i with respect to x_k equal to e^{x_k} **only** for $i = k$.

Otherwise, with respect to x_k, it is a constant and its derivative is 0.

Softmax function

Derivative of softmax function

Case 1 $i = k$

$$\frac{\partial h}{\partial x_k}(x) = e^{x_k}$$

$$\frac{\partial g_i}{\partial x_k}(x) = e^{x_k}$$

$$\frac{\partial}{\partial x_k} S_i(x) = \frac{\frac{\partial}{\partial x_k} g_i(x) h(x) - g_i(x) \frac{\partial}{\partial x_k} h(x)}{\left(h(x)\right)^2} = \frac{e^{x_k} h(x) - e^{x_k} e^{x_k}}{\left(h(x)\right)^2} = \frac{e^{x_k}\left(h(x) - e^{x_k}\right)}{\left(h(x)\right)^2} =$$

$$\frac{e^{x_k}}{h(x)} \cdot \frac{h(x) - e^{x_k}}{h(x)} = \frac{e^{x_k}}{h(x)} \cdot \left(1 - \frac{e^{x_k}}{h(x)}\right) = \frac{g_k(x)}{h(x)} \cdot \left(1 - \frac{g_k(x)}{h(x)}\right) =$$

$$S_k(x)\left(1 - S_k(x)\right)$$

So finally in this case, when $i = k$, you have:

$$\frac{\partial S_i}{\partial x_k}(x) = S_k(x)\left(1 - S_k(x)\right)$$

Softmax function

Derivative of softmax function

Case 2 $i \neq k$

$$\frac{\partial h}{\partial x_k}(x) = e^{x_k}$$

$$\frac{\partial g_i}{\partial x_k}(x) = 0$$

$$\frac{\partial}{\partial x_k} S_i(x) = \frac{\frac{\partial}{\partial x_k} g_i(x) h(x) - g_i(x) \frac{\partial}{\partial x_k} h(x)}{\Big(h(x)\Big)^2} = \frac{0 \cdot h(x) - e^{x_i} e^{x_k}}{\Big(h(x)\Big)^2} = \frac{-e^{x_i} e^{x_k}}{\Big(h(x)\Big)^2} =$$

$$-\frac{e^{x_i}}{h(x)} \cdot \frac{e^{x_k}}{h(x)} = -S_i(x) S_k(x) = S_i(x)(0 - S_k(x))$$

So finally in this case, when $i \neq k$, you have:

$$\frac{\partial S_i}{\partial x_k}(x) = -S_i(x) S_k(x) = S_i(x)(0 - S_k(x))$$

Softmax function

Derivative of softmax function

Summarizing

$$\frac{\partial S_i}{\partial x_k}(x) = \begin{cases} S_i(x)\big(1 - S_k(x)\big) & \text{for } i = k \\ S_i(x)\big(0 - S_k(x)\big) & \text{for } i \neq k \end{cases}.$$

Sometimes, this piecewise function can be put together using Kronecker delta function δ_{ik}

$$\frac{\partial S_i}{\partial x_k}(x) = S_i(x)\big(\delta_{ik} - S_k(x)\big),$$

where

$$\delta_{ik} = \begin{cases} 1 \text{ for } i = k \\ 0 \text{ for } i \neq k \end{cases}.$$

In the search of minimum function value

Analogy

A man in the blizzard.

Derivative

$$f'(a) = \lim_{x \to \infty} \frac{f(a+h) - f(a)}{h}$$

If x and y are real numbers, and if the graph of f is plotted against x, the derivative is the *slope* of this graph at each point.

$$m = \frac{\Delta y}{\Delta x}$$

$$f(x) = mx + b$$

See below links for some examples and animations:

`https://en.wikipedia.org/wiki/Derivative`

Gradient

The gradient of a scalar-valued differentiable function f of several variables is the vector field (or vector-valued function) ∇f whose value at a point p is the vector whose components are the partial derivatives of f at p. That is, for $f\colon \mathbf{R}^n \to \mathbf{R}$, its gradient $\nabla f\colon \mathbf{R}^n \to \mathbf{R}^n$ is defined at the point $p = (x_1, \ldots, x_n)$ in n-dimensional space as the vector:

$$\nabla f(p) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{bmatrix}.$$

Gradient descent

Gradient descent is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function.

The idea is to **take repeated steps in the opposite direction of the gradient** of the function at the current point, **because this is the direction of steepest descent**.

If the multi-variable function $F(\mathbf{x})$ is defined and differentiable in a neighborhood of a point $\mathbf{a}$, then $F(\mathbf{x})$ decreases fastest if one goes from $\mathbf{a}$ in the direction of the negative gradient of F at $\mathbf{a}$ ($-\nabla F(\mathbf{a})$):

$$\mathbf{a}_{n+1} = \mathbf{a}_n - \eta \nabla F(\mathbf{a}_n).$$

If $\eta \in \mathbb{R}_+$ is small enough, then $F(\mathbf{a_n}) \geq F(\mathbf{a_{n+1}})$.

See below links for some examples and animations:

`https://en.wikipedia.org/wiki/Gradient_descent`

See also [FulSD].

1D case

$$f(x) = x^2, f'(x) = 2x, \eta = 0.2$$

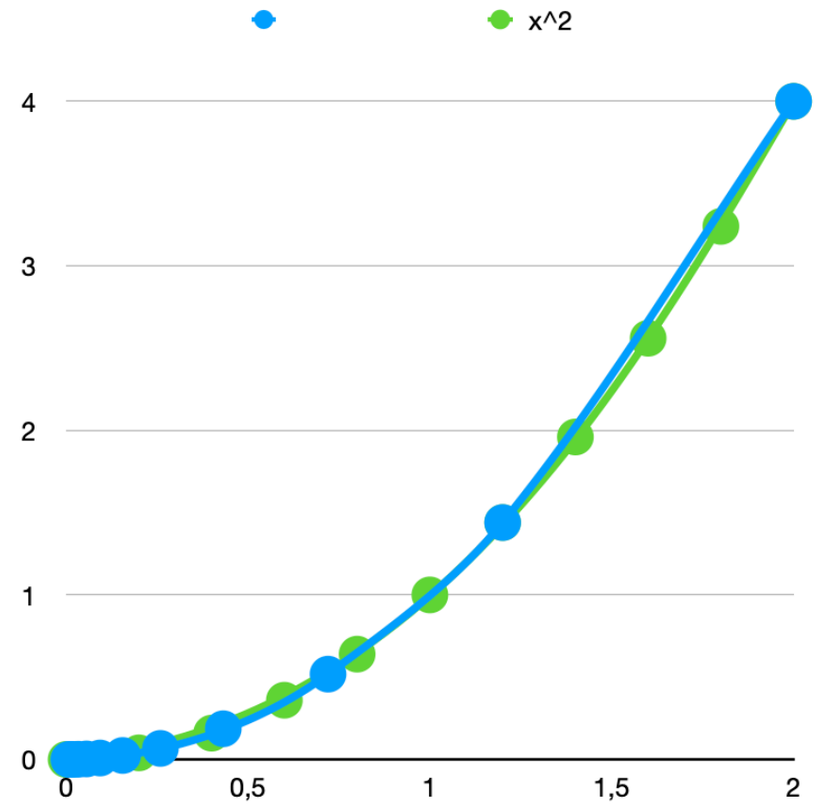


Tabela 1

	f(x)=x^2	f'(x)=2x	eta*f'(x)	x-eta*f'(x)	Eta
2	4	4	0,8	1,2	0,2
1,2	1,44	2,4	0,48	0,72	0,2
0,72	0,5184	1,44	0,288	0,432	0,2
0,432	0,186624	0,864	0,1728	0,2592	0,2
0,2592	0,06718464	0,5184	0,10368	0,15552	0,2
0,15552	0,0241864704	0,31104	0,062208	0,093312	0,2
0,093312	0,008707129344	0,186624	0,0373248	0,0559872	0,2
0,0559872	0,00313456656384	0,1119744	0,02239488	0,03359232	0,2
0,03359232	0,001128443962982	0,06718464	0,013436928	0,020155392	0,2
0,020155392	0,000406239826673	0,040310784	0,0080621568	0,0120932352	0,2
0,0120932352	0,000146246337602	0,0241864704	0,00483729408	0,00725594112	0,2
0,00725594112	0,000052648681536	0,01451188224	0,002902376448	0,004353564672	0,2

1D case

$$f(x) = x^2, f'(x) = 2x, \eta = 0.4$$

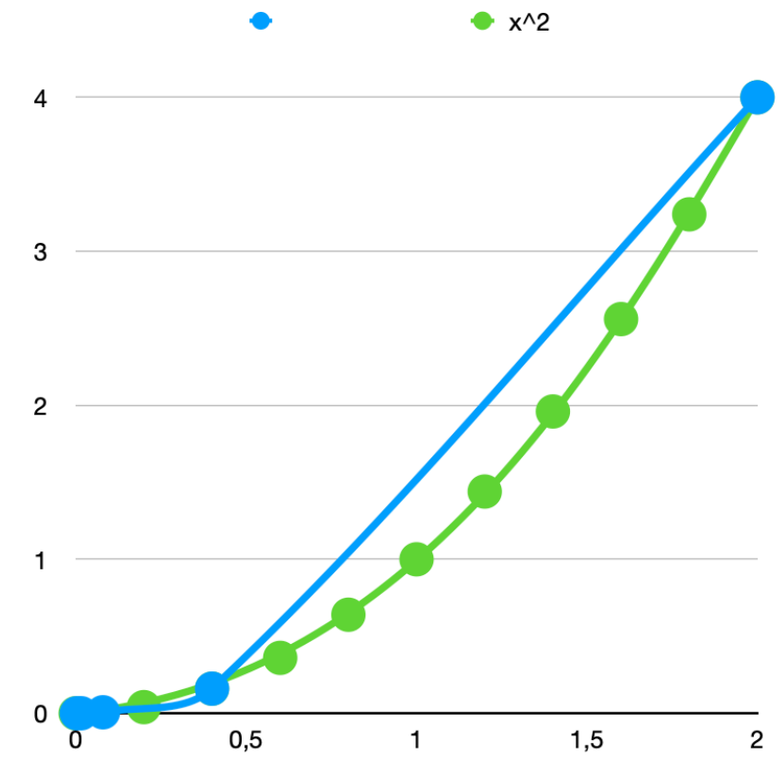


Tabela 1

X	f(x)=x^2	f'(x)=2x	eta*f'(x)	x-eta*f'(x)	Eta
2	4	4	1,6	0,4	0,4
0,4	0,16	0,8	0,32	0,08	0,4
0,08	0,0064	0,16	0,064	0,016	0,4
0,016	0,000256	0,032	0,0128	0,0032	0,4
0,0032	0,00001024	0,0064	0,00256	0,00064	0,4
0,00064	0,0000004096	0,00128	0,000512	0,000128	0,4
0,000128	0,000000016384	0,000256	0,0001024	0,0000256	0,4
0,0000256	0,00000000065536	0,0000512	0,00002048	0,00000512	0,4
0,00000512	0,000000000026214	0,00001024	0,000004096	0,000001024	0,4
0,000001024	0,000000000001048	0,000002048	0,0000008192	0,0000002048	0,4
0,0000002048	0,000000000000041	0,0000004096	0,00000016384	0,00000004096	0,4
0,00000004096	0,000000000000001	0,00000008192	0,000000032768	0,000000008192	0,4

1D case

$$f(x) = x^2, f'(x) = 2x, \eta = 0.5$$

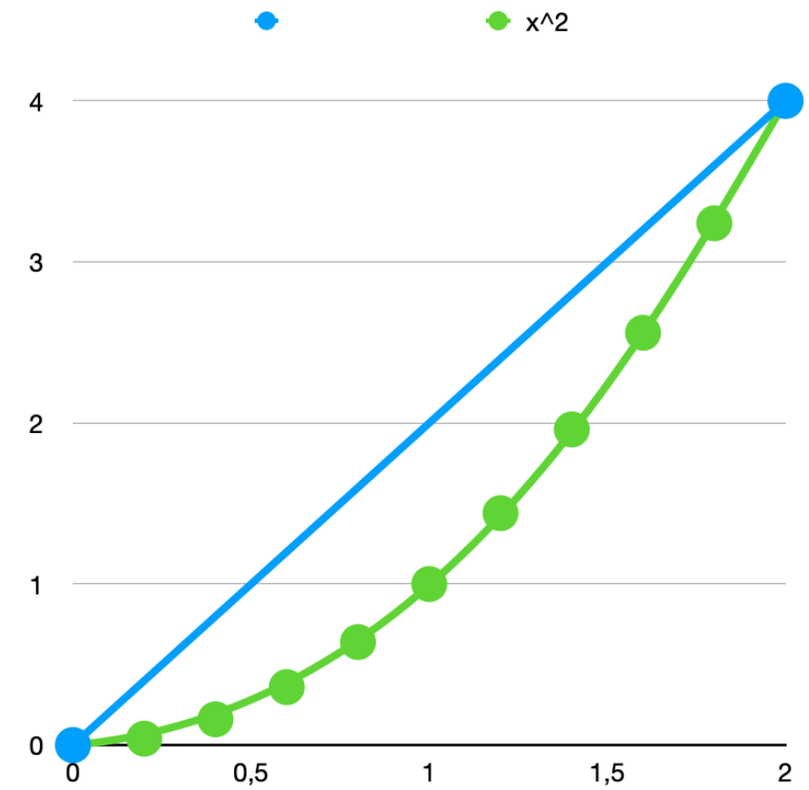


Tabela 1

X	f(x)=x^2	f'(x)=2x	eta*f'(x)	x-eta*f'(x)	Eta
2	4	4	2	0	0,5
0	0	0	0	0	0,5
0	0	0	0	0	0,5
0	0	0	0	0	0,5
0	0	0	0	0	0,5
0	0	0	0	0	0,5
0	0	0	0	0	0,5
0	0	0	0	0	0,5
0	0	0	0	0	0,5
0	0	0	0	0	0,5
0	0	0	0	0	0,5
0	0	0	0	0	0,5
0	0	0	0	0	0,5

1D case

$$f(x) = x^2, f'(x) = 2x, \eta = 1.0$$

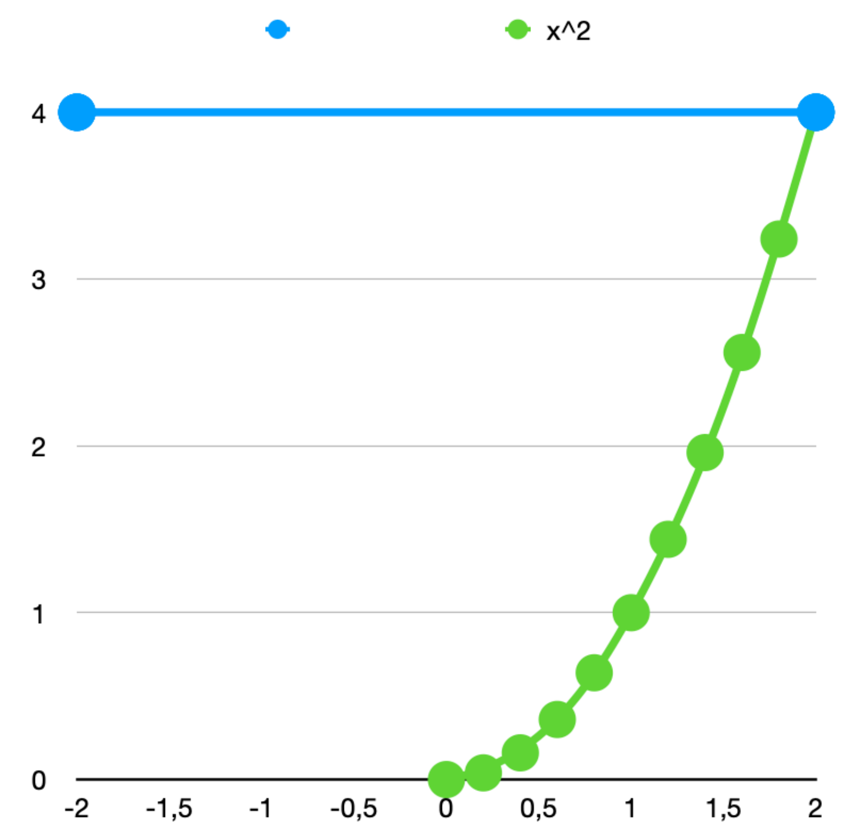


Tabela 1

	f(x)=x^2	f'(x)=2x	eta*f'(x)	x-eta*f'(x)	Eta
2	4	4	4	-2	1,0
-2	4	-4	-4	2	1,0
2	4	4	4	-2	1,0
-2	4	-4	-4	2	1,0
2	4	4	4	-2	1,0
-2	4	-4	-4	2	1,0
2	4	4	4	-2	1,0
-2	4	-4	-4	2	1,0
2	4	4	4	-2	1,0
-2	4	-4	-4	2	1,0
2	4	4	4	-2	1,0
-2	4	-4	-4	2	1,0

1D case

$$f(x) = x^2, f'(x) = 2x, \eta = 1.05$$

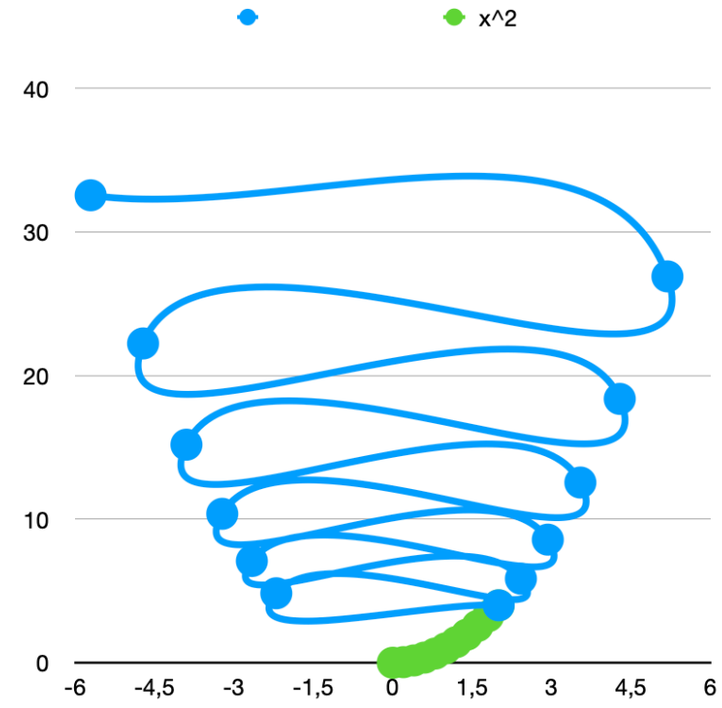


Tabela 1

	f(x)=x^2	f'(x)=2x	eta*f'(x)	x-eta*f'(x)	Eta
2	4	4	4,2	-2,2	1,05
-2,2	4,84	-4,4	-4,62	2,42	1,05
2,42	5,8564	4,84	5,082	-2,662	1,05
-2,662	7,086244	-5,324	-5,5902	2,9282	1,05
2,9282	8,57435524	5,8564	6,14922	-3,22102	1,05
-3,22102	10,3749698404	-6,44204	-6,764142	3,543122	1,05
3,543122	12,553713506884	7,086244	7,4405562	-3,8974342	1,05
-3,8974342	15,1899933433296	-7,7948684	-8,18461182	4,28717762	1,05
4,28717762	18,3798919454289	8,57435524	9,003073002	-4,715895382	1,05
-4,715895382	22,2396692539689	-9,431790764	-9,9033803022	5,1874849202	1,05
5,1874849202	26,9099997973024	10,3749698404	10,89371833242	-5,70623341222	1,05
-5,70623341222	32,5610997547359	-11,41246682444	-11,983090165662	6,276856753442	1,05

Artificial neural networks

Fast introduction

Into the learning

1. `prezentacja_ldi_2019.pdf` (artificial neuron, all-or-nothing rule, typical architectures).

2. Idea of backpropagation (continue with "red path" following "blue path" included in presentation) with emphasis on what does it mean *learn neural network* - adjust neural network's weights.

3. Explain what does *adjust neural network*'s *weights* mean - adjust to what? Adjust, so the neural network error is minimal - other words, find such a parameters of a neural network (weight are the only parameters we have), so it makes the smallest possible error

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t),$$

where $\Delta w_{ij}(t)$ is the most crucial part.

Fast introduction

1. Perceptron rule:

   $$\Delta w_i = \eta[t - f(net)]x_i$$

2. Define neural network error function:

   $$E(w) = \frac{1}{2}\sum_{k=1}^{m}(t_k - y_k)^2$$

3. Delta rule:

   $$\Delta w_i = \eta \underbrace{[t - f(net)]f'(net)}_{\delta} x$$

4. **Delta rule is important because it can be generalized (if only activation function is differentiable – this explains why you use sigmoid functions which approximates a stepwise functions).** Generalized delta rule:

   output layer l: $\delta_l = [t - f(net)]f'(net)$

   all other layers $i = 1,\ldots,l-1$: $\delta_i = \delta_{i+1} w f'(net)$

   modify weights in each layer $i = 1,\ldots,l$: $w_i = w_i + \eta\delta_i x_i$

Fast introduction

Other materials

- One of the following:
    - Group 1:
        1. [FulGrzAI], chapter 16 (16.1-16.5) i 17 (17.1-17.4)
    - Group 2:
        1. WSTĘP DO UCZENIA MASZYNOWEGO – WYKŁAD 1 (https://www.math.uni.lodz.pl/~kosmatka/wdum-wstep/)
        2. WSTĘP DO UCZENIA MASZYNOWEGO – WYKŁAD 2 (https://www.math.uni.lodz.pl/~kosmatka/wstep-do-uczenia-maszynowego-wyklad-2/)
- David Kriesel, *A Brief Introduction to Neural Networks*, 2007, http://www.dkriesel.com/en/science/neural_networks
- Michael Nielsen, *Neural Networks and Deep Learning*, http://neuralnetworksanddeeplearning.com

Interactive examples

Examples

Interactive examples

- `perceptronDemo.jar`

  $$\Delta w_i = \eta[t - f(net)]x_i$$

- `backpropDemo.jar`

  (generalized) delta rule:

$$\Delta w_i = \eta \underbrace{[t - f(net)]f'(net)}_{\delta} x$$

Examples

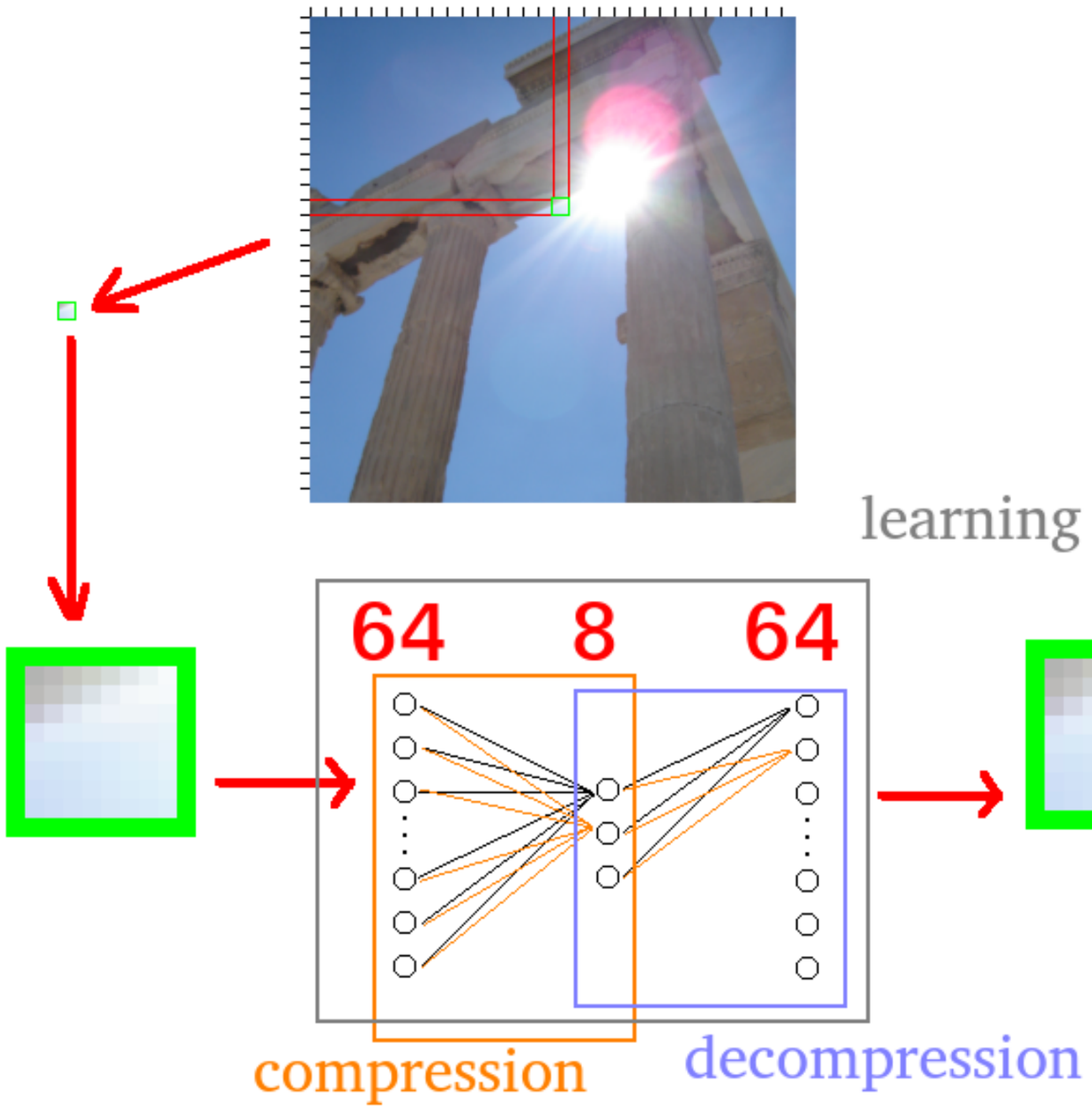
Interactive examples

- Feed-forward neural networks
    - Plane separation (`ssn1.jar`).
    - Simple image recognition (`ssn2.jar`).
    - XOR problem (`ssn3.jar`).
    - Simple game - tanks (`ssn4.jar`).
- Recurent neural networks
    - Hopfield neural network (`ssn6.jar`).
- Self-organizing neural networks
    - Example (`ssn7.jar`).
- Image compression (`ssn5.jar`).

Examples

Interactive examples - image compression

- Idea



- Image compression (`ssn5.jar`).

Neural networks in practice

Task 1

Tanks in Python

1. Implement missing code in Python.

2. Test if it works for game (macOS version).

Task 2

Solving XOR problem with Keras

- Code
    - `lecture_06_02.py` - XOR in Keras.

For more information about Keras, see for example [Bur, Cho, Mol] .

Bibliography

- [FulSD] Algorytm najszybszego spadku
  `https://fulmanski.pl/zajecia/sztuczna/zajecia_old/materialy/cw08/cw08.htm`
- [Bur] Christoph Burgdorf, *Understanding XOR with Keras and TensorFlow*,
  `https://blog.thoughtram.io/machine-learning/2016/11/02/understanding-XOR-with-keras-and-tensorlow.html`
- [Cho] Francois Chollet, *Deep Learning with Python*, Manning, 2018.
- [FulGrzAI] Piotr Fulmański, Marta Grzanek, *Sztuczna inteligencja. Podręcznik do wykładów i ćwiczeń*
  `https://fulmanski.pl/books/doc/ai_2010.pdf`
- David Kriesel, *A Brief Introduction to Neural Networks*, 2007,
  `http://www.dkriesel.com/en/science/neural_networks`
- [Mol] Jojo Moolayil, *Learn Keras for Deep Neural Networks*, Apress, 2019.
- Michael Nielsen, *Neural Networks and Deep Learning*,
  `http://neuralnetworksanddeeplearning.com`