Shallow Neural Networks

Danilo Comminiello

Neural Networks 2023/2024

October 11, 2023

Sapienza
Università di Roma

Table of contents

1 Learning in Higher Dimensional Space

The need for nonlinear models

For most of the problems met in real applications, linear models *cannot* model sufficiently well the data and lead to poor performance.

For example, recall that, given two jointly distributed random vectors $(\mathbf{y}, \mathbf{x}) \in \mathbb{R}^k \times \mathbb{R}^l$, we know that the optimal estimate of $\mathbf{y}$ given $\mathbf{x}$, in the mean-square-error sense (MSE), is the corresponding conditional mean, i.e., $\mathrm{E}\{\mathbf{y}|\mathbf{x}\}$, which in general is a *nonlinear function* of $\mathbf{x}$.

We mainly deal with nonlinear learning techniques based on the *mapping* of the input data to a new space, such that the original nonlinear task is *transformed* into a linear one.

Generalized linear models

Given $(y, \mathbf{x}) \in \mathbb{R} \times \mathbb{R}^l$, a *generalized linear estimator* $\hat{y}$ of y has the form:

$$\hat{y} = f(\mathbf{x}) = \sum_{k=1}^{K} w_k \phi_k(\mathbf{x}) + \theta_0, \tag{1}$$

where $\phi_1(\cdot), \ldots, \phi_K(\cdot)$ are *preselected* nonlinear functions.

A popular family of functions is the **polynomial** one, e.g.:

$$\hat{y} = \sum_{i=1}^{l} w_i x_i + \sum_{i=1}^{l-1} \sum_{m=i+1}^{l} w_{i,m} x_i x_m + \sum_{i=1}^{l} w_{i,i} x_i^2 + \theta_0. \tag{2}$$

Cover's theorem: spatially distributed points

We have already justified the expansion in eq. (1) by mobilizing arguments from the universal approximation theory.

However, in the context of *classification tasks*, looking at this expansion from a different angle provides an alternative and important interpretation.

Let us consider N points, $x_1, x_2, \ldots, x_N \in \mathbb{R}^l$, in general position, i.e., there is no subset of $l+1$ of them lying on a $(l-1)$-dimensional hyperplane.

For example, in the two dimensional space, any three of these points are not permitted to lie on a straight line.

Cover's theorem: definition

The number of groupings, denoted as $\mathcal{O}(N, l)$, that can be formed by $(l-1)$-dimensional hyperplanes to separate the N points in two classes, exploiting all possible combinations, is given by:

$$\mathcal{O}(N, l) = 2\sum_{i=0}^{l} \begin{pmatrix} N-1 \\ i \end{pmatrix},$$

where $$\begin{pmatrix} N-1 \\ i \end{pmatrix} = \frac{(N-1)!}{(N-1-i)!i!}.$$

Each one of these groupings in two classes is also known as a linear dichotomy.

Cover's theorem: example in two dimensional space

The figure below illustrates the Cover's theorem for the case of $N = 4$ points in the two dimensional space.

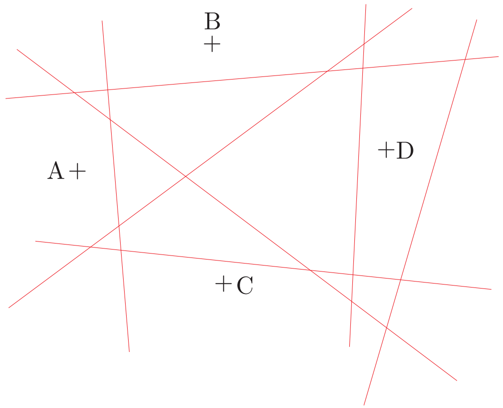

Figure 1: Possible groupings are: [(ABCD)], [A,(BCD)], [B,(ACD)], [C,(ABD)], [D,(ABC)], [(AB),(CD)], [(AC),(BD)]. Each grouping is counted twice, since it can belong to either ω_1 or ω_2 class. The total number of groupings is $\mathcal{O}(4,2) = 14$. The number of all possible combinations is $2^N = 16$. The grouping which is not counted is [(BC),(AD)], which is *not* linearly separable [1].

Cover's theorem: grouping linearly separable patterns

If $N \leq l+1$, then $\mathcal{O}(N,l) = 2^N$, i.e., all possible combinations are linearly separable.

Based on the previous theorem, given N points in $\mathbb{R}^l$, the probability of grouping these points in two *linearly separable classes* is:

$$P_N^l = \frac{\mathcal{O}(N,l)}{2^N} = \begin{cases} \frac{1}{2^{N-1}} \sum_{i=0}^{l} \begin{pmatrix} N-1 \\ i \end{pmatrix}, & N \geq l+1, \\ 1, & N \leq l+1. \end{cases}$$

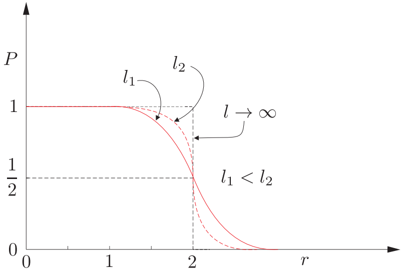

Figure 2: Set r: $N = r(l+1)$. The figure shows the probability, as a function of r, of grouping any of the data into two *linearly separable* classes. For $N > 2(l+1)$, the probability becomes *small*. For large values of l, and provided $N < 2(l+1)$, the probability tends to unity. Also, if $N \leq (l+1)$, all possible groupings in two classes are linearly separable [1].

Mapping in a higher-dimensional space

In order to exploit the previous theorem we consider N feature vectors $\mathbf{x}_n \in \mathbb{R}^l$, $n = 1, 2, \ldots, N$, so that the following mapping is performed:

$$\phi : \ \ \mathbf{x}_n \in \mathbb{R}^l \longmapsto \boldsymbol{\phi}\left(\mathbf{x}_n\right) \in \mathbb{R}^K, \quad K >> l$$

Then according to the Cover's theorem, *the higher* the value of K is *the higher* the probability becomes for the images of the mapping, $\boldsymbol{\phi}\left(\mathbf{x}_n\right) \in \mathbb{R}^K$, with $n = 1, 2, \ldots, N$, to be linearly separable in $\mathbb{R}^K$.

The expansion of a nonlinear classifier (for a binary classification task) is equivalent with a linear task after such a mapping, i.e.:

$$\hat{y} = \operatorname{sgn}\left(f\left(\mathbf{x}\right)\right), \qquad f\left(\mathbf{x}\right) = \sum_{k=1}^{K} w_k \phi_k\left(\mathbf{x}\right) + \theta_0 = \mathbf{w}^{\mathsf{T}} \begin{bmatrix} \boldsymbol{\phi}\left(\mathbf{x}\right) \\ 1 \end{bmatrix}, \tag{3}$$

where $\boldsymbol{\phi}\left(\mathbf{x}\right) = \begin{bmatrix} \phi_1\left(\mathbf{x}\right) & \phi_2\left(\mathbf{x}\right) & \ldots & \phi_K\left(\mathbf{x}\right) \end{bmatrix}^{\mathsf{T}}$.

Example of mapping in a higher-dimensional space I

Provided that K is *large enough*, the task is linearly separable in the new space, $\mathbb{R}^K$, with high probability, which justifies the use of a *linear classifier*, $\mathbf{w}$, in eq. (3).

Consider two patterns, x_1, x_2, that are *not* linearly separable in the two-dimensional space.

Hence, we map them in $\mathbb{R}^3$ by means of a nonlinear quadratic function, i.e.:

$$\begin{bmatrix} x_1 & x_2 \end{bmatrix}^\mathsf{T} \longmapsto \boldsymbol{\phi}(\mathbf{x}) = \begin{bmatrix} x_1 & x_2 & f(x_1, x_2) \end{bmatrix}^\mathsf{T}$$

$$\text{where} \qquad f(x_1, x_2) = 4 \exp\left(-\left(x_1^2 + x_2^2\right)/3\right) + 5$$

Then, the patterns become linearly separable in the new space.

Example of mapping in a higher-dimensional space II

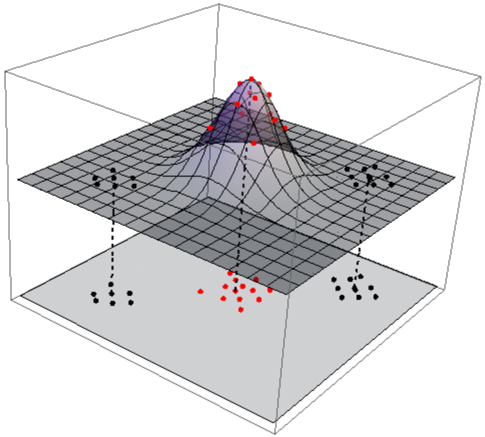

Figure 3: The patterns (red points in one class and black for the other), that are not linearly separable in the original two-dimensional plane, become linearly separable after the nonlinear mapping in the three-dimensional space; one can draw a plane that separates the “black” from the “red” points. [1]

Example of mapping in a higher-dimensional space III

After the mapping from the original l-dimensional space to the new K-dimensional one, the images of the points $\phi(\mathbf{x}_n)$, $n = 1, 2, \ldots, N$, lie on an l-dimensional surface (**manifold**) in $\mathbb{R}^K$.

Since l variables were originally chosen to describe each pattern (dimensionality, number of free parameters), the same number of free parameters will be required to describe the same objects after the mapping in $\mathbb{R}^K$.

In other words, after the mapping, we embed an l-dimensional manifold in a K-dimensional space, in such a way, so that the data in the two classes to become linearly separable.

2 Introduction to Neural Networks

Modeling complex tasks with deep learning

Deep learning is getting a lot of attention due to its favorable capabilities and it definitely represents one of the hottest topics in the machine learning field.

Deep learning can be understood as a subfield of machine learning that is concerned with training models based on artificial neural networks (NNs) with many layers efficiently.

The basic concept behind artificial NNs was built upon hypotheses and models of how the human brain works to solve complex problems.

In this lecture, we are going to learn the basic concepts of NNs so that we may deal with deep neural network (DNN) architectures that are particularly well suited for signal processing.

Shallow learning

All the models we adopted up to now are often denoted as *shallow*.

Shallow learning is characterized by a single level of processing, which often yields an output derived from an inner product between the weights and the input, e.g.:

$$\mathbf{y} = \mathbf{W}\mathbf{x}$$

Shallow models are linear classifier, regression filters, PCA and basic neural networks, among others [2].

Such methods allow us to use simple representations of the input.

Biological neuron

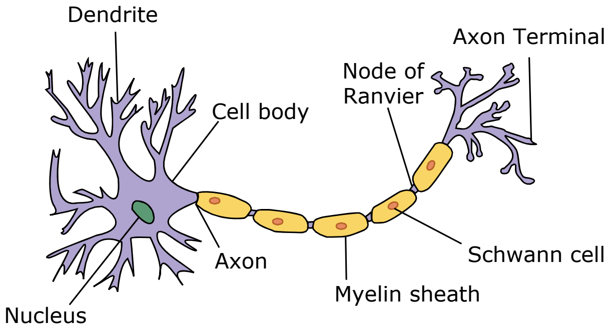

Figure 4: The biological neuron [3].

Warren McCulloch and Walter Pitts began to develop models of artificial neurons considering a cartoonish picture of a biological neuron, consisting of *dendrites* (input terminals), the *nucleus* (CPU), the *axon* (output wire), and the *axon terminals* (output terminals), enabling connections to other neurons via *synapses*.

Single-layer neural networks

We can represent shallow models as a single-layer neural network, where every input is connected to a single neuron (computed node).

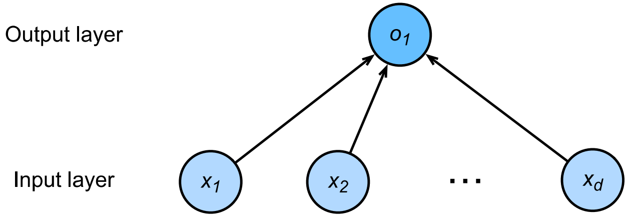

Figure 5: Diagram of a single-layer neural network [3]. The diagram indicates the connectivity pattern (here, each input is connected to the output) but not the values taken by the weights or biases.

Since for this model, every input is connected to every output (in this case there is only one output!), we can regard this transformation as a **fully-connected layer**, also commonly called a dense layer.

Single-layer neural networks with multiple outputs

We can generalize the previous diagram of Fig. 5 considering multiple output neurons.

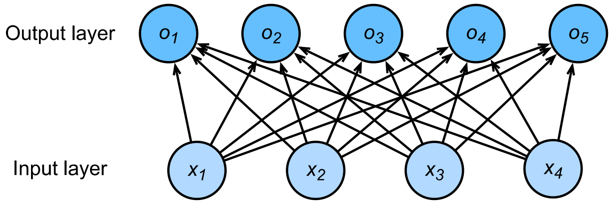

Figure 6: Single layer perceptron with 5 output units [3].

This model mapped our inputs directly to our outputs via a softmax regression involving a single linear transformation:

$$\mathbf{o} = \mathrm{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b}). \tag{4}$$

If output labels are linearly related to input data, then this approach would be sufficient. But linearity is a strong assumption.

Looking for deeper representation I

The way human brain learns is inherently hierarchical, thus being able to provide a *deeper* representation.

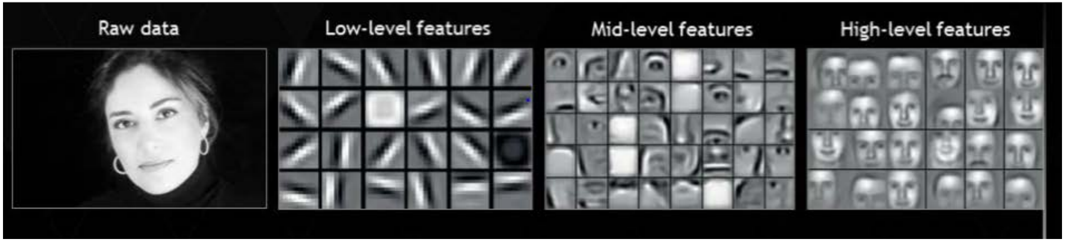

Figure 7: Learning "deeper" implies learning features of features of features...
Image source: Analytics Vidhya.

Looking for deeper representation II

On one hand, there might exist a representation of our data that would take into account the relevant interactions among our features, on top of which a linear model would be suitable.

However, we simply do not know *a priori* how to calculate it by hand.

With deep neural networks, we used observational data to jointly learn both a representation (via hidden layers) and a linear predictor that acts upon that representation.

3 Multilayer Perceptron

How to create deeper learning algorithms I

Thus, the simplest idea to “go deeper” would be trying to create multilayers networks by incorporating one or more hidden layers, as:

$$\mathbf{y} = \mathbf{W}_2 \mathbf{W}_1 \mathbf{x} \tag{5}$$

The easiest way to do this is to stack many fully-connected layers on top of each other.

Each layer feeds into the layer above it, until we generate an output.

Every input influences every neuron in the hidden layer, and each of these in turn influences every neuron in the output layer.

Multilayer perceptron

The resulting architecture is called a **multilayer perceptron** (MLP).

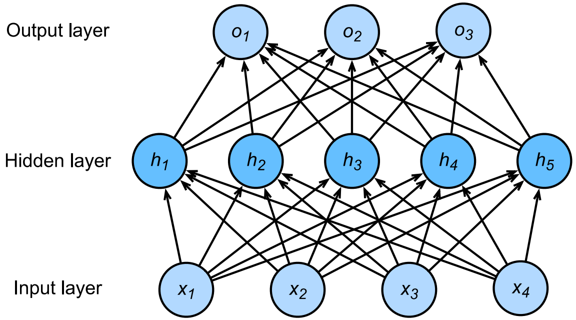

Figure 8: Multilayer perceptron with hidden layers. This example contains a hidden layer with 5 hidden units in it [3]. Note that layers are both fully connected.

Since the input layer does not involve any calculations, producing outputs with this network requires implementing the computations for each of the 2 layers (hidden and output).

Linear computation for the multilayer perceptron I

Formally, we calculate the output of a one-hidden-layer MLP (5) as follows:

$$\begin{aligned}\mathbf{h} &= \mathbf{W}_1\mathbf{x} + \mathbf{b}_1,\\ \mathbf{o} &= \mathbf{W}_2\mathbf{h} + \mathbf{b}_2,\\ \mathbf{y} &= \mathrm{softmax}(\mathbf{o}).\end{aligned} \tag{6}$$

Note that after adding this layer, our model now requires us to track and update two additional sets of parameters.

However, simply adding extra (linear) layers *does not* lead to something “deeper” as a result.

Linear computation for the multilayer perceptron II

The reason is plain.

The hidden units above are given by a linear function of the inputs, and the outputs (pre-softmax) are just a linear function of the hidden units.

A linear function of a linear function is itself a linear function. Thus, the only result would be a collapse of the layers into a single linear transformation.

Moreover, our linear model was already capable of representing any linear function.

Example of binary classification using linear MLP

If we consider a binary classification problem using a 2-layer neural network as (6), we will find out that:

- the first layer is a “feature” transform
- the second layer is a simple classifier.

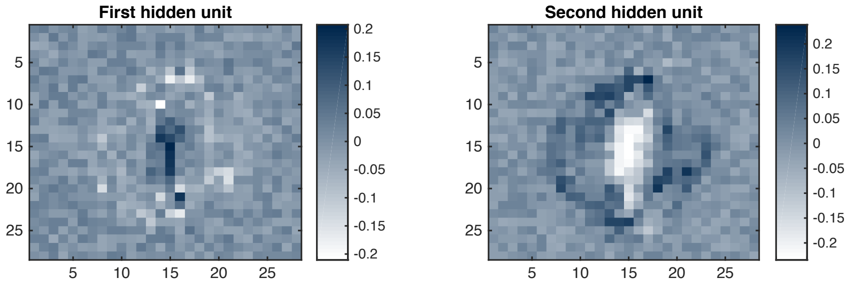

Figure 9: Representation of learned weights using linear multilayer networks [2].

From linear to nonlinear MLP

One fundamental step for *going deeper* is applying a nonlinear activation function σ to each hidden unit of the MLP:

$$\begin{aligned} \mathbf{h} &= \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1), \\ \mathbf{o} &= \mathbf{W}_2\mathbf{h} + \mathbf{b}_2, \\ \mathbf{y} &= \mathrm{softmax}(\mathbf{o}). \end{aligned} \tag{7}$$

This allows to stack transforms meaningfully.

In general, with these activation functions in place, it is no longer possible to collapse our MLP into a linear model.

To build more general MLPs, we can continue stacking such hidden layers, e.g., $\mathbf{h}_1 = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$ and $\mathbf{h}_2 = \sigma(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$, one atop another, yielding ever more expressive models (assuming fixed width).

Interpretation of a nonlinear MLP

In a multilayer network created according to (7):

- the first layer contains low-level features,
- second and other intermediate layers contain mid- and high-level features,
- the last layer is a classifier.

For certain choices of the activation function, it is widely known that MLPs are universal approximators.

However, this does not mean that we should try to solve all of your problems with single-layer networks.

In fact, we can approximate many functions much more compactly by using deeper (vs wider) networks.

Example of binary classification with 2-layer MLP I

If we consider a binary classification problem using a 2-layer MLP as (7), we will find out that:

- The **input layer** describes each pattern as a point in the feature space.
- The **first hidden layer** of nodes forms a *partition* of the input space and placed each input point in one of the regions, using a coding scheme of zeros and ones at the outputs of the respective neurons.

  This can be considered as a more abstract representation of our input patterns.

Example of binary classification with 2-layer MLP II

- The **second hidden layer** of nodes, based on the information provided by the previous layer, encodes information related to the classes.

  This is a further *abstraction of the representation*, which carries some type of semantic meaning.

  For example, it could provide information of whether a tumor is malignant or benign, in a related a medical application [1].

It turns out a hierarchical type of representations of the input patterns that mimics the physical mechanism on which brain intelligence is built upon.

Vectorization and minibatch

We can generalize the MLP considering a minibatch of inputs $\mathbf{X}$.

The calculations to produce outputs from an MLP with two hidden layers can thus be expressed:

$$\begin{aligned}\mathbf{H}_1 &= \sigma(\mathbf{W}_1\mathbf{X} + \mathbf{b}_1),\\ \mathbf{H}_2 &= \sigma(\mathbf{W}_2\mathbf{H}_1 + \mathbf{b}_2),\\ \mathbf{O} &= \mathrm{softmax}(\mathbf{W}_3\mathbf{H}_2 + \mathbf{b}_3).\end{aligned}$$

We define the nonlinearities σ and $\mathrm{softmax}$ to apply to its inputs in a row-wise fashion, i.e., one observation at a time.

Note that, often, the activation functions that we apply to hidden layers are not merely row-wise, but component wise.

4 ACTIVATION FUNCTIONS

Activation functions

An **activation function** σ decides whether a neuron should be *activated* or *not* by calculating the weighted sum and further adding bias with it.

They are differentiable operators to transform input signals to outputs, while most of them add nonlinearity.

Because activation functions are fundamental to deep learning, let's briefly survey some common activation functions.

ReLU function I

The most popular choice, due to both simplicity of implementation and its performance on a variety of predictive tasks, is the rectified linear unit (**ReLU**).

ReLUs provide a very simple nonlinear transformation.

Given the element z, the function is defined as the maximum of that element and 0.

$$\mathrm{ReLU}(z) = \max(z, 0).$$

ReLU function II

The ReLU function retains only positive elements and discards all negative elements (setting the corresponding activations to 0).

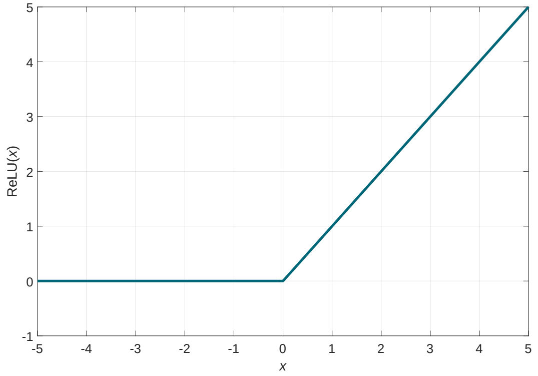

Figure 10: ReLU activation function. As it is possible to see, the ReLU is a piecewise linear function.

Derivative of the ReLU function

When the input is negative, the derivative of ReLU function is 0 and when the input is positive, the derivative of ReLU function is 1.

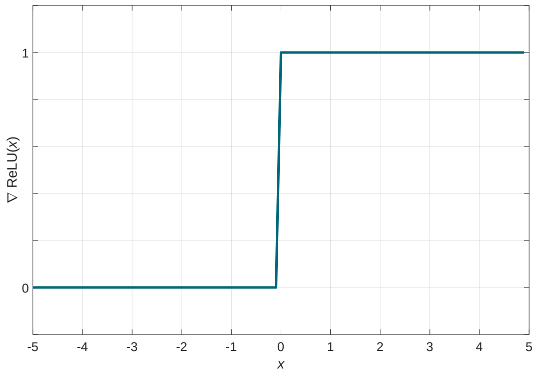

Figure 11: Derivative of the ReLU activation function. As it is possible to see, the ReLU is a piecewise linear function. Note that the ReLU function is *not differentiable* when the input takes value precisely equal to 0. In these cases, the left-hand-side derivative is taken by default thus saying that the derivative is 0 when the input is 0.

Sigmoid activation function I

The **sigmoid function** transforms its inputs, which values in the domain $\mathbb{R}$, to outputs that lie the interval $(0, 1)$.

For that reason, the sigmoid is often called a squashing function: it *squashes* any input in the range $(-\infty, \infty)$ to some value in $(0, 1)$:

$$\operatorname{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.$$

Earliest neural networks were provided by thresholding units, taking value 0 when its input is below some threshold and value 1 when the input exceeds the threshold.

Sigmoid activation function II

When attention shifted to gradient based learning, the sigmoid function was a natural choice because it is a smooth, differentiable approximation to a thresholding unit.

Sigmoids are still widely used as activation functions on the output units, when we want to interpret the outputs as probabilities for binary classification problems (see Lecture 9).

In many modern NNs, the sigmoid has mostly been replaced by the simpler and more easily trainable ReLU for most use in hidden layers.

However, as we will see, some neural architectures leverage sigmoid units to control the flow of information across time.

Sigmoid activation function III

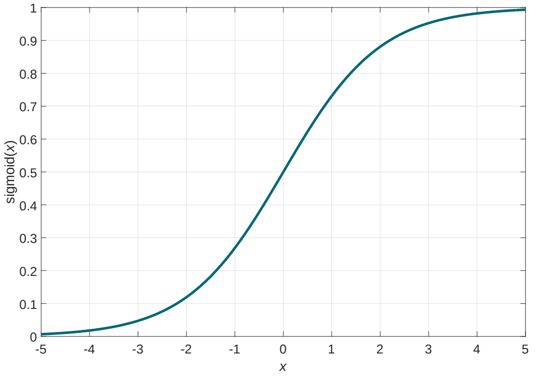

Figure 12: Sigmoid activation function. Note that when the input is close to 0, the sigmoid function approaches a linear transformation.

Derivative of the sigmoid function

The derivative of sigmoid function is given by the following equation:

$$\frac{d}{dx}\text{sigmoid}(x) = \frac{\exp(-x)}{(1+\exp(-x))^2} = \text{sigmoid}(x)\left(1-\text{sigmoid}(x)\right).$$

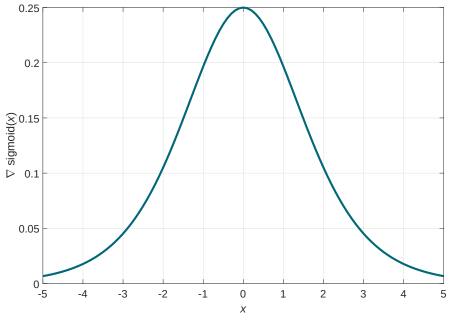

Figure 13: Derivative of the sigmoid activation function. Note that when the input is 0, the derivative of the sigmoid function reaches a maximum of 0.25.

Hyperbolic tangent function

The **hyperbolic tangent** (tanh) also squashes its inputs, but in $(-1, 1)$:

$$\text{tanh}(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$

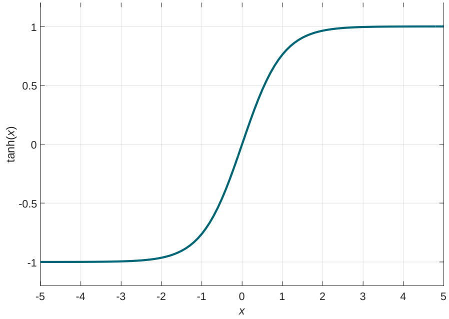

Figure 14: Hyperbolic tangent activation function. Note that when the input is close to 0, the sigmoid function approaches a linear transformation. Although the shape of the function is similar to the sigmoid function, the tanh function exhibits point symmetry about the origin of the coordinate system.

Derivative of the tanh function

The derivative of tanh function is given by the following equation:

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

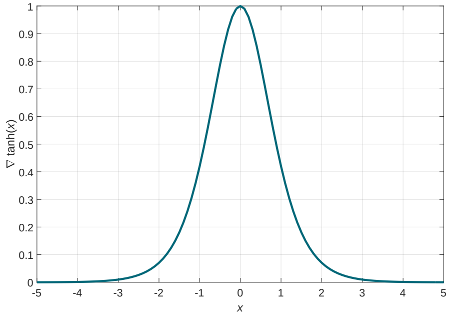

Figure 15: Derivative of the hyperbolic tangent function. Note that when the input is 0, the derivative of the hyperbolic tangent function reaches a maximum of 1. As the input diverges from 0 in either direction, the derivative approaches 0.

References

[1] S. Theodoridis, *Machine Learning. A Bayesian and Optimization Perspective*. Elsevier, 2nd ed., 2020.

[2] P. Smaragdis, "Machine Learning for Signal Processing course," 2018. University of Illinois at Urbana-Champaign.

[3] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive Into Deep Learning*. 0.7.1 ed., 2020.

[4] C. M. Bishop, "Training with noise is equivalent to Tikhonov regularization," *Neural Computation*, vol. 7, no. 1, pp. 108–116, 1995.

[5] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

Shallow Neural Networks

Neural Networks 2023/2024

Danilo Comminiello

https://sites.google.com/uniroma1.it/neuralnetworks2023

{danilo.comminiello, simone.scardapane}@uniroma1.it