FYS-STK4155 - Applied data analysis and machine learning

Project 3

Even M. Nordhagen

November 22, 2018

- Github repository containing programs and results:

  `https://github.com/evenmn/FYS-STK4155`

Abstract

The aim of this project is to divide various sounds into ten categories, inspired by the Urban Sound Challenge. For that, we use classification methods spanning from logistic regression to random forests...

To estimate the error, Mean Square Error (MSE) and the R^2-score function are used in the regression case, and we find more reliable values by K-fold validation resampling. In the classification case, we will study the accuracy score since the outputs are binary. Gradient descent (GD) and stochastic gradient descent (SGD) are implemented as the minimization tools.

In linear regression, Lasso and Ridge regression gave smaller MSE (below 10^{-4}) compared to OLS, and only Lasso was able to recall the correct J-matrix. Using a neural network, a pure linear activation function on a hidden layer gave best results. For classification, logistic regression did not work well, but with a neural network we were able to obtain an accuracy above 99% for a test set far from the critical temperature and above 96% for a test set close to the critical temperature using three hidden layers with *Rectified Linear Units* (ReLU) and *leaky* ReLU activation functions.

Contents

1 Introduction

In the everyday life we are continuously surrounded by sounds, and usually the human brain is able to recognize what kind of sound it is based on experience. Artificial neural networks are again based on studies of the human brain, and if the artificial neurons work as the biological ones, there should be possible training a neural network recognizing sounds as well.

Lately, immense efforts have been put into this subject with the purpose of translating voice into text, leaded by technology companies like Google, Microsoft, Amazon and so on, who develop voice controlled virtual assistants. The technology is promising, the time saving using voice commands vs. keyboard commands is potentially huge and the market is enormous. Some even claim that virtual assistants will run our lives within 20 years. [`https://nordic.businessinsider.com/ai-assistants-will-run-our-lives-20-years-from-now-2018-1?r=US&IR=T`] If that is right is still uncertain, but what is certain is that the technology has great potential.

In this final project we will, based on the Urban Sound Challenge, sort sounds into classes using various classification methods. We use the same idea as when the great technology companies recognize voice, but we are going to differentiate between sounds made by **air conditioners**, **car horns**, **children**, **dogs**, **drills**, **engines**, **guns**, **jackhammers**, **sirens** and **street musicians**. In order to do that,

2 Theory

2.1 Sound analysis

2.1.1 Time domain

Sounds are longitudinal waves which actually are different pressures in the air. They are usually represented by functions giving pressure per time, where a pure tone has a repeating function. This function is obviously continuous, but since computers discretize functions, we will loose some information, see Figure ...

INSERT FIGURE

How much information we loose depends on the sampling frequency (or sampling rate), which is the number of sampling points per second. A rule of thumb is that one should have twice as high sampling rate as the highest sound frequency to keep the most important information. For instance, a human ear can perceive frequencies in the range 20-20000Hz, so around a sampling rate around 40kHz is known to be satisfying. Ordinary CD's use a sampling rate of 44.1kHz.

2.1.2 Frequency domain

Sometimes, a frequency domain rather than the time domain gives a better picture. On one hand, one looses the time dependency, but on the other one gets information about which frequencies that are found in the wave. One goes from the time domain to the frequency domain using Fourier transformations. Which one that gives the best results in our case is not clear.

INSERT FIGURE

2.2 Urban Sound Challenge

The Urban Sound Challenge is a classification challenge provided by Analytics Vidhya with the purpose of introducing curious people to a real-world classification problem. To receive the data set, one needs to register for the challenge and is then expected to summit the solutions in 31st of Desember 2018. For more practical information, see [`https://datahack.analyticsvidhya.com/contest/practice-problem-urban-sound-classification/`]

The data set consists of 8732 sound samples with a constant sampling rate of 22050Hz and the lengths are maximum 4 seconds, but vary. This makes the training session difficult, since we will get input arrays of different lengths. The samplings are distributed between ten classes, namely

- air conditioner
- car horn
- children playing
- dog bark
- drilling
- engine idling
- gun shot
- jackhammer
- siren
- street music

ADD SOME EXAMPLES

3 Methods

3.1 Classification methods

3.2 Logistic regression

Despite its name, logistic regression is not a fitting tool, but rather a classification tool. Traditionally, the perceptron model was used for 'hard classification', which sets the outputs directly to binary values. However, often we are interested in the probability of a given category, which means that we need a continuous *activation function*. Logistic regression can, like linear regression, be considered as a function where coefficients are adjusted with the intention to minimize the error. Here, the coefficients are called *weights*. The process goes like this: The inputs are multiplied by several weights, and by adjusting those weights the model can classify every *linear classification problem*. A drawing of the perceptron is found in figure (1).

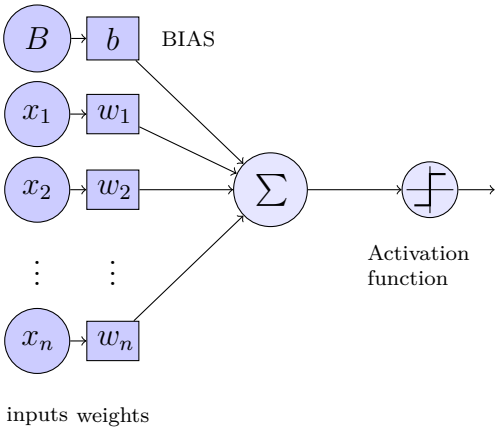

Figure 1: Logistic regression model with n inputs.

In logistic regression, we usually have one binary output node for each class, but for two categories one output node is sufficient, which can be fired or not fired.

Initially, one needs to train the perceptron such that it knows which outputs are correct, and for that one needs to know the outputs that correspond to the inputs. Every time the network is trained, the weights are adjusted such that the error is minimized.

The very first step is to calculate the initial outputs (forward phase), where the weights usually are set to small random numbers. Then the error is calculated, and the weights are updated to minimize the error (backward phase). So far so good.

3.2.1 Forward phase

Let us look at it from a mathematical perspective, and calculate the net output. The net output seen from an output node is simply the sum of all the "arrows" that point towards the node, see figure (1), where each "arrow" is given by the left-hand node multiplied with its respective weight. For example, the contribution from input node 2 to the output node follows from $X_2 \cdot w_2$, and the total net output to the output O is therefore

$$net = \sum_{i=1}^{I} x_i \cdot w_i + b \cdot 1. \tag{1}$$

Just some notation remarks: x_i is the value of input node i and w_i is the weight which connects input i to the output. b is the bias weight, which we will discuss later.

You might wonder why we talk about the net output all the time, do we have other outputs? If we look at the network mathematically, what we talk about as the net output should be our only output. Anyway, it turns out to be convenient mapping the net output to a final output using an activation function, which is explained further in section 3.2.5. The activation function, f, takes in the net output and gives the output,

$$out = f(net). \tag{2}$$

If not everything is clear right now, it is fine. We will discuss the most important concepts before we dive into the maths.

3.2.2 BIAS

As mentioned above, we use biases when calculating the outputs. The nodes, with value B, are called the bias nodes, and the weights, b, are called the bias weights. But why do we need those?

Suppose we have two inputs of value zero, and one output which should not be zero (this could for instance be a NOR gate). Without the bias, we will not be able to get any other output than zero, and in fact the network would struggle to find the right weights even if the output had been zero.

The bias value B does not really matter since the network will adjust the bias weights with respect to it, and is usually set to 1 and ignored in the calculations. [2]

3.2.3 Learning rate

In principle, the weights could be updated without adding any learning rate ($\eta = 1$), but this can in practice be problematic. It is easy to imagine that the outputs

can be fluctuating around the targets without decreasing the error, which is not ideal, and a learning rate can be added to avoid this. The downside is that with a low learning rate the network needs more training to obtain the correct results (and it takes more time), so we need to find a balance.

3.2.4 Cost function

The cost function is what defines the error, and in logistic regression the cross-entropy function is a naturally choice. [3] It reads

$$c(\boldsymbol{W}) = -\sum_{i=1}^{n} \Big[y_i \log f(\boldsymbol{x}_i^T \boldsymbol{W}) + (1 - y_i) \log[1 - f(\boldsymbol{x}_i^T \boldsymbol{W})] \Big] \tag{3}$$

where $\boldsymbol{W}$ contains all weights, included the bias weight ($\boldsymbol{W} \equiv [b, \boldsymbol{W}]$), and similarly does $\boldsymbol{x}$ include the bias node, which is 1; $\boldsymbol{x} \equiv [1, \boldsymbol{x}]$. Further, the $f(x)$ is the activation function discussed in next section.

The cross-entropy function is derived from likelyhood function, which famously reads

$$p(y|x) = \hat{y}^y \cdot (1 - \hat{y})^{1-y}. \tag{4}$$

Working in the log space, we can define a log likelyhood function

$$\log \Big[p(y|x) \Big] = \log \Big[\hat{y}^y \cdot (1 - \hat{y})^{1-y} \Big] \tag{5}$$

$$= y \log \hat{y} + (1 - y) \log(1 - \hat{y}) \tag{6}$$

which gives the log of the probability of obtaining y given x. We want this quantity to increase then the cost function is decreased, so we define our cost function as the negative log likelyhood function. [7]

Additionally, including a regularization parameter λ inspired by Ridge regression is often convenient, such that the cost function is

$$c(\boldsymbol{W})^+ = c(\boldsymbol{W}) + \lambda ||\boldsymbol{W}||_2^2. \tag{7}$$

We will later study how this regularization affects the classification accuracy.

3.2.5 Activation function

Above, we were talking about the activation function, which is used to activate the net output. In binary models, this is often just a step function firing when the net output is above a threshold. For continuous outputs, the logistic function given by

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{8}$$

is usually used in logistic regression to return a probability instead of a binary value. This function has a simple derivative, which is advantageous when calculating a large number of derivatives. As shown in section A.1, the derivative is simply

$$\frac{df(x)}{dx} = x(1-x). \tag{9}$$

$tanh(x)$ is another popular activation function in logistic regression, which more or less holds the same properties as the logistic function.

3.2.6 Backward phase

Now all the tools for finding the outputs are in place, and we can calculate the error. If the outputs are larger than the targets (which are the exact results), the weights need to be reduced, and if the error is large the weights need to be adjusted a lot. The weight adjustment can be done by any minimization method, and we will look at a couple of gradient methods. To illustrate the point, we will stick to the **gradient descent** (GD) method in the calculations, even though other methods will be used later. The principle of GD is easy: each weight is "moved" in the direction of steepest slope,

$$\boldsymbol{w}^{+} = \boldsymbol{w} - \eta \cdot \frac{\partial c(\boldsymbol{w})}{\partial \boldsymbol{w}}, \tag{10}$$

where η is the learning rate and $c(\boldsymbol{w})$ is the cost function. We use the chain rule to simplify the derivative

$$\frac{\partial c(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{\partial c(\boldsymbol{w})}{\partial out} \cdot \frac{\partial out}{\partial net} \cdot \frac{\partial net}{\partial \boldsymbol{w}} \tag{11}$$

where the first is the derivative of the cost function with respect to the output. For the cross-entropy function, this is

$$\frac{\partial c(\boldsymbol{w})}{\partial out} = -\frac{y}{out} + \frac{1-y}{1-out}. \tag{12}$$

Further, the second derivative is the derivative of the activation function with respect to the output, which is given in (9)

$$\frac{\partial out}{\partial net} = out(1-out). \tag{13}$$

The latter derivative is the derivative of the net output with respect to the weights, which is simply

$$\frac{\partial net}{\partial \boldsymbol{w}} = \boldsymbol{x}. \tag{14}$$

If we now recall that $out = f(\boldsymbol{x}^T\boldsymbol{w})$, we can write

$$\frac{\partial c(\boldsymbol{w})}{\partial \boldsymbol{w}} = [f(\boldsymbol{x}^T\boldsymbol{w}) - \boldsymbol{y}]\boldsymbol{x} \tag{15}$$

and obtain a weight update algorithm

$$\boldsymbol{w}^+ = \boldsymbol{w} - \eta \cdot [f(\boldsymbol{x}^T\boldsymbol{w}) - \boldsymbol{y}]^T\boldsymbol{x}. \tag{16}$$

where the bias weight is included implicitly in $\boldsymbol{w}$ and the same applies for $\boldsymbol{x}$.

3.3 Neural network

If you have understood logistic regression, understanding a neural network should not be a difficult task. According to **the universal approximation theorem**, a neural network with only one hidden layer can approximate any continuous function. [8] However, often multiple layers are used since this tends to give fewer nodes in total.

In figure (2), a two-layer neural network (one hidden layer) is illustrated. It has some similarities with the logistic regression model in figure (1), but a hidden layer and multiple outputs are added. In addition, the output is no longer probabilities and can take any number, which means that we do not need to use the logistic function on the outputs anymore.

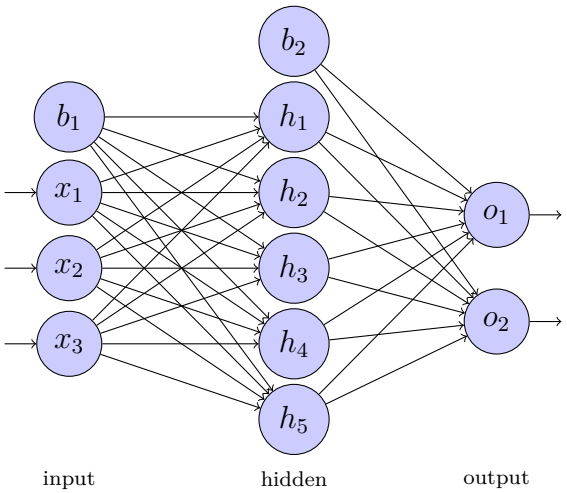

Figure 2: Neural network with 3 input nodes, 5 hidden nodes and 2 output nodes, in addition to the bias nodes.

Without a hidden layer, we have seen that the update of weights is quite straight forward. For a neural network consisting of multiple layers, the question

is: how do we update the weights when we do not know the values of the hidden nodes? And how do we know which layer causing the error? This will be explained in section 3.3.3, where one of the most popular techniques for that is discussed. Before that we will generalize the forward phase presented in logistic regression.

3.3.1 Forward phase

In section 3.2.1, we saw how the output is found for a single perceptron. Since we only had one output node, the weights could be stored in an array. Generally, it is more practical to store the weights in matrices, since they will have indices related to both the node on left-hand side and the node on the right-hand side. For instance, the weight between input node x_3 and hidden node h_5 in figure (2) is usually labeled as w_{35}. Since we have two layers, we also need to denote which weight set it belongs to, which we will do by a superscript ($w_{35} \Rightarrow w_{35}^{(1)}$). In the same way, $\boldsymbol{W}^1$ is the matrix containing all $w_{ij}^{(1)}$, $\boldsymbol{x}$ is the vector containing all x_i's and so on. We then find the net outputs at the hidden layer to be

$$net_{h,j} = \sum_{i=1}^{I} x_i \cdot w_{ij}^{(1)} = \boldsymbol{x}^T \boldsymbol{W}_j^{(1)} \tag{17}$$

where the $\boldsymbol{x}$ and $\boldsymbol{W}^{(1)}$ again are understood to take the biases. This will be the case henceforth. The real output to the hidden nodes will be

$$h_j = f(net_{h,j}). \tag{18}$$

Further, we need to find the net output to the output nodes, which is obviously just

$$net_{o,j} = \sum_{i=1}^{H} h_i \cdot w_{ij}^{(2)} = \boldsymbol{h}^T \boldsymbol{W}_j^{(2)} \tag{19}$$

We can easily generalize this. Looking at the net output to a hidden layer l, we get

$$\boldsymbol{net}_{h_l} = \boldsymbol{h}^{(\boldsymbol{l-1})^T} \boldsymbol{W}^{(l)}. \tag{20}$$

3.3.2 Activation function

Before 2012, the logistic, the tanh and the pur linear functions where the standard activation functions, but then Alex Krizhevsky published an article where he introduced a new activation function called *Rectified Linear Units (ReLU)* which outperformed the classical activation functions. [4] The network he used is now known as AlexNet, and helped to revolutionize the field of computer vision. [5]

After that, the ReLU activation function has been modified several times (avoiding zero derivative among others), and example of innovations are *leaky ReLU* and *Exponential Linear Unit (ELU)*. All those networks are linear for positive numbers, and small for negative numbers. Often, especially in the output layer, a straight linear function is used as well.

In figure (**??**), *standard RELU, leaky RELU* and *ELU* are plotted along with the logistic function.

3.3.3 Backward Propagation

Backward propagation is probably the most used technique for updating the weights, and is actually again based on equation (10). What differs, is the differentiation of the net input with respect to the weight, which gets more complex as we add more layers. For one hidden layer, we have two sets of weights, where the last layer is updated in a similar way as for a network without hidden layer, but the inputs are replaced with the values of the hidden nodes:

$$w_{ij}^{(2)+} = w_{ij}^{(2)} - \eta \cdot [f(h_i^T w_{ij}) - y_j]^T h_i. \tag{21}$$

We recognize the first part as δ_{ok}, such that

$$w_{ij}^{(1)+} = w_{ij}^{(1)} - \eta \cdot \sum_{k=1}^{O} \delta_{ok} \cdot w_{jk}^{(2)} \cdot f'(out_{hj}) \cdot x_i \tag{22}$$

where we recall δ_{ok} as

$$\delta_{ok} = -(t_{ok} - out_{ok}) \cdot f'(out_{ok}).$$

For more layers, the procedure is the same, but we keep on inserting the obtained outputs from various layers.

3.3.4 Summary

Since it will be quite a lot calculations, I will just express the results here, and move the calculations to Appendix B. The forward phase in a three-layer perceptron is

$$\begin{aligned} net_{hi} &= \sum_j w_{ji}^{(1)} \cdot x_j \\ out_{hi} &= \mathrm{f}(net_{hi}) \\ \\ net_{ki} &= \sum_j w_{ji}^{(2)} \cdot out_{hj} \\ out_{ki} &= \mathrm{f}(net_{ki}) \\ \\ net_{oi} &= \sum_j w_{ji}^{(3)} \cdot out_{kj} \\ out_{oi} &= \mathrm{f}(net_{oi}) \end{aligned} \tag{23}$$

which can easily be turned into vector form. The backward propagation follows from the two-layer example, and we get

$$w_{ij}^{(3)} = w_{ij}^{(3)} - \eta \cdot \delta_{oj} \cdot out_{ki}$$

$$w_{ij}^{(2)} = w_{ij}^{(2)} - \eta \sum_{k=1}^{O} \delta_{ok} \cdot w_{jk}^{(3)} \cdot f'(out_{kj}) \cdot out_{hi}$$

$$w_{ij}^{(1)} = w_{ij}^{(1)} - \eta \sum_{k=1}^{O} \sum_{l=1}^{K} \delta_{ok} \cdot w_{lk}^{(3)} \cdot f'(out_{kl}) \cdot w_{jl}^{(2)} f'(out_{hj}) \cdot x_i$$

where we again use the short hand

$$\delta_{oj} = (t_j - out_{oj}) \cdot f'(out_{oj}).$$

If we compare with the weight update formulas for the two-layer case, we recognize some obvious connections, and it is easy to imagine how we can construct a general weight update algorithm, no matter how many layers we have.

3.4 Minimization methods

In many cases we are not able to obtain an analytical expression for the derivative of a function, and to minimize it we therefore need to apply numerical minimization methods. An example is the minimum of the Lasso cost function, which does not have an analytical expression. In this section we will present two simple and similar methods, first ordinary gradient descent and then stochastic gradient descent.

3.4.1 Gradient Descent (GD)

The Gradient Descent method was mentioned already in the theory part, in equation (10), and this will therefore just be a quick reminder of the idea. The though is that we need to move in the direction where the cost function is steepest, which will take us to the minimum. The gradient always points in the steepest direction, and since we want to move in opposite direction of the gradient, the gradient is subtracted from the weights in every iteration. Updating the weights using gradient descent therefore reads

$$W_{ij}^{+} = W_{ij} - \eta \cdot \frac{\partial c(\boldsymbol{W})}{\partial W_{ij}} \tag{24}$$

where W_{ij}^{+} is the updated W_{ij} and η is the learning rate. An example implementation looks like

```
for iter in range(T):
        for i in range(N):
                y[i] = feedforward(X[i])
                gradient = (y[i] - t[i]) * df(y[i])
                W -= self.eta * gradient
```

3.4.2 Stochastic Gradient Descent (SGD)

We now turn to the stochastic gradient descent, which hence the name is stochastic. The idea is to calculate the gradient of just a few states, and hope that the gradient will work for the remaining states as well. It will probably not converge in fewer steps, but each step will be faster. We can express the SGD updating algorithm as

$$W_{ij}^{+} = W_{ij} - \frac{\eta}{N} \sum_{k=1}^{N} \frac{\partial c_k(\boldsymbol{W})}{\partial W_{ij}} \tag{25}$$

where we sum over all states in a so-called minibatch. After going through all minibatches, we say that we have done an epoch.

Because of the stochasticity, we are less likely to be stuck in local minima, and the minimization will be significantly faster because we calculate just a fraction of the gradients compared to ordinary gradient descent. Anyway, we expect this method to require more iterations than standard gradient descent.

```
for epoch in range(T):
        for i in range(m):
                random_index = np.random.randint(m)
                Xi = self.X[random_index:random_index+1]
                ti = self.t[random_index:random_index+1]
                yi = feedforward(Xi)
                gradient = (yi - ti) * df(yi)
                W -= self.eta * gradient
```

3.5 Error analysis

To find out how good our model is, we need to compare the outputs with the targets in one way or another. For continuous outputs, we typically calculate absolute error, relative error or mean square error, and often the R^2-score is evaluated to estimate how much of the data that is described by the model. In classification the outputs are binary classes, and we should only care about whether the model choose correct class or not. Therefore, the error is either 0 or 1, and we apply the accuracy score.

3.5.1 Accuracy score

The accuracy score is a very intuitive error estimation, and is just number of correctly classified images divided by the total number of images,

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(y_i = t_i)}{n}. \tag{26}$$

It is typically used to determine the error in classification, since the error is binary (the classification is either correct, or wrong).

4 Code

Using the language of machine learning: Python. Useful packages sklearn, keras, TensorFlow. Parallel processing using Dask. Pandas used in reading.

4.1 Code structure

4.2 Implementation

5 Results

Table 1: The accuracy obtained using neural network for classification. 'Train' is the training set, 'Test' is a test set far from critical temperature and 'Critical' is a test set close to the critical temperature. We used one minimization iteration ($T = 1$), learning rate $\eta = 1e - 4$ and regularization parameter $\lambda = 1e - 3$, used logistic activation function on output node and tried *pure linear*, *ReLU* and *Leaky ReLU* on the hidden layers. Notation "10+10" means two layers of 10 hidden nodes each.

		Accuracy		
	Hidden nodes	Train	Test	Critical
Linear	10	0.4371	0.4381	0.3333
	10+10	0.4379	0.4367	0.3333
	10+10+10	0.4371	0.4382	0.3333
ReLU	10	0.9930	0.9929	0.9655
	10+10	0.9935	0.9934	0.9679
	10+10+10	0.9935	**0.9938**	**0.9686**
Leaky	10	0.9931	0.9929	0.9656
	10+10	0.9932	0.9933	0.9668
	10+10+10	**0.9936**	0.9935	0.9685

6 Discussion

7 Conclusion

8 References

[1] E. M. Nordhagen. Project 1 - FYS-STK4155 - Applied data analysis and machine learning. https://github.com/evenmn/FYS-STK4155/tree/master/doc/Project1. (2018).

[2] S. Marsland. Machine Learning: An Algorithmic Perspective. Second Edition (Chapman & Hall/Crc Machine Learning & Pattern Recognition). (2014).

[3] P. Mehta, C.H. Wang, A.G.R. Day, C. Richardson, M.Bukov, C.K.Fisher, D.J. Schwab. A high-bias, low-variance introduction to Machine Learning for physicists. (2018).

[4] ImageNet Classification with Deep Convolutional Neural Networks. A. Krizhevsky, I. Sutskever, G. E. Hinton. (2012).

[5] K. Allen. How a Toronto professors research revolutionized artificial intelligence. thestar.com, retrieved November 3rd, 2018. (2015).

[6] ELU-Networks: Fast and Accurate CNN Learning on ImageNet. M.Heusel, D.A.Clevert, G.Klambauer, A.Mayr, K.Schwarzbauer, T.Unterthiner, S.Hochreiter. http://image-net.org/challenges/posters/JKU_EN_RGB_Schwarz_poster.pdf

[7] Logistic Regression-Detailed Overview. S.Swaminathan. https://towardsdatascience.com/logistic-regression-detailed-overview-46c4da4303bc

[8] Data Analysis and Machine Learning: Neural networks, from the simple perceptron to deep learning and convolutional networks. M.Hjorth-Jensen. https://compphysics.github.io/MachineLearning/doc/pub/NeuralNet/html/NeuralNet.html

A Appendix A - Derivation of activation functions

A.1 Sigmoid

If we differentiate the sigmoid function

$$f(x) = \frac{1}{1 + \exp(-x)} \tag{27}$$

from equation (8), and we will get

$$f'(x) = f(1 - f) \tag{28}$$

PROOF:

$$\begin{aligned}
f'(x) &= -1 \cdot (1 + \exp(-x))^{-2} \cdot -1 \cdot \exp(-x) \\
&= \frac{\exp(-x)}{(1 + \exp(-x))^2} \\
&= \frac{1 + \exp(-x) - 1}{(1 + \exp(-x))^2} \\
&= \frac{1 + \exp(-x)}{(1 + \exp(-x))^2} - \frac{1}{(1 + \exp(-x))^2} \\
&= \frac{1}{1 + \exp(-x)} - \frac{1}{(1 + \exp(-x))^2} \\
&= \frac{1}{1 + \exp(-x)} \left(1 - \frac{1}{1 + \exp(-x)}\right) \\
&= f(1 - f)
\end{aligned}$$

B Appendix B - Backward propagation