Project 2

Håkon Olav Torvik, Vetle Vikenes & Sigurd Sørlie Rustad

FYS-STK4155 – Applied Data Analysis and Machine Learning
Autumn 2021
Department of Physics
University of Oslo

November 2, 2021

Abstract: Abstract.

Contents

1 Introduction

2 Theory

In the theory-section we aim to give a brief explanation of the main concepts and terminology used in this report. For a more in-depth explanation we recommend reading the appropriate sections in [1], which has been of great inspiration and help for us throughout the project.

2.1 Gradient Decent

In this section we cover gradient decent and different variations of it. More specifically we describe gradient decent (GD), stochastic gradient decent (SGD) and adding momentum to the aforementioned methods.

2.1.1 Ordinary Gradient Decent

Gradient decent methods is often used to minimize the so-called cost/loss-function. Thus, lets say we have a cost function $C(\boldsymbol{\beta})$ which could be expressed as

$$C(\boldsymbol{\beta}) = \sum_{i=1}^{n} c_i(\mathbf{x}_i, \boldsymbol{\beta}). \tag{2.1}$$

Where n denotes the number of datapoints and $\mathbf{x}$ are the datapoints. The gradient with respect to the parameters $\boldsymbol{\beta}$ is then defined as

$$\nabla_{\boldsymbol{\beta}} C(\boldsymbol{\beta}) = \sum_{i=1}^{n} \nabla_{\boldsymbol{\beta}} c(\mathbf{x}_i, \boldsymbol{\beta}). \tag{2.2}$$

The algorithm for GD is then:

$$\begin{aligned} \mathbf{v}_t &= \eta \nabla_{\boldsymbol{\beta}} C(\boldsymbol{\beta}_t) \\ \boldsymbol{\beta}_{t+1} &= \boldsymbol{\beta}_t - \mathbf{v}_t, \end{aligned} \tag{2.3}$$

where η is what we call the learning rate. This algorithms finds (ideally), with each iteration, new β_{k+1} values which decreases the cost function. This is of course not always the case, and depends on the value of η. There are many potential problems when choosing the wrong learning rate. For example, if it is to big, our answer can diverge. If η is too small we will need too many iterations to reach the minima, or reach a local minima.

2.1.2 Stochastic Gradient Decent

Another challenge, which is reduced when using SGD, are the large number of computations needed when calculating the gradient. Instead of calculating the total derivative as in (2.2), we approximate it. This is done by performing the gradient on a subset of the data, called a minibatch. With n still denoting the total number of datapoints, we will have $N_B = n/M$ minibatches, where M is the size of each minibatch. The minibatches are denoted by B_k. Thus our approximated gradient, using a

single minibatch B_k is defined as

$$\nabla_\beta C^{MB}(\boldsymbol{\beta}) \equiv \sum_{i \in B_k} \nabla_\beta c(\mathbf{x}_i, \boldsymbol{\beta}). \tag{2.4}$$

Then the aim is to use this approximated gradient, for all N_B minibatches, to update the parameters $\boldsymbol{\beta}$, at every step k. Doing this for all N_B minibatches, are what we refer to as an epoch. The SGD algorithm then becomes very similar to (2.3), however with an approximated gradient.

$$\begin{aligned} \mathbf{v}_t &= \eta \nabla_\beta C^{MB}(\boldsymbol{\beta}_t) \\ \boldsymbol{\beta}_{t+1} &= \boldsymbol{\beta}_t - \mathbf{v}_t \end{aligned} \tag{2.5}$$

This not only speeds up the algorithms, it also helps prevent getting stuck in local minima because of the stochastic nature.

2.1.3 Adding Momentum

Still the method can be optimized even further by adding momentum. This is done by a small modification in the parameter $\mathbf{v}_t$ in equations (2.3) and (2.5). Namely we add a so-called mass term, to simulate the step sizes having momentum

$$\mathbf{v}_t = \eta \nabla_{\beta_t} C(\boldsymbol{\beta}) \rightarrow \mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta \nabla_\beta C(\boldsymbol{\beta}_t). \tag{2.6}$$

Here γ is what we could refer to as the mass, and is another free parameter. One of the benefits is for example that this lets us move faster in regions where the gradient is small.

2.2 Feed-Forward Deep Neural Networks

Neural networks are neural-inspired nonlinear models, which are taught by a way of supervised learning. We will in this section explain what we mean by non-linearity, the basic architecture of a neural network and how the network learns.

2.2.1 Architecture of Neural Networks

The structure we are going to use in this report is similar to that in figure 1. The gray circles are what we refer to as nodes. For now we just need to know that they hold some numerical value. One initializes the network by giving the nodes in the input layer numerical values. These values would correspond to some actual physical property, for example brightness of pixels in a picture. Then, each node in the input layer is connected to each node in the hidden layer h_1. In figure 1 we have three such hidden layers, where each node in one layer is connected to every node in the next layer. Now the nodes are connected through what we will refer to as weights, biases and activation functions (more on that later). The connections are what assigns the numerical value of the nodes in the next layer. Lastly we have the output layer, which outputs values dependent on the problem. If we have a classification situation, where we for example wanted to classify the type of animal in different pictures, then one node could correspond to a lion, next to a zebra and so fourth. By this we would know what animal the network *thinks* is in the picture by looking at what neuron has the highest numerical value.

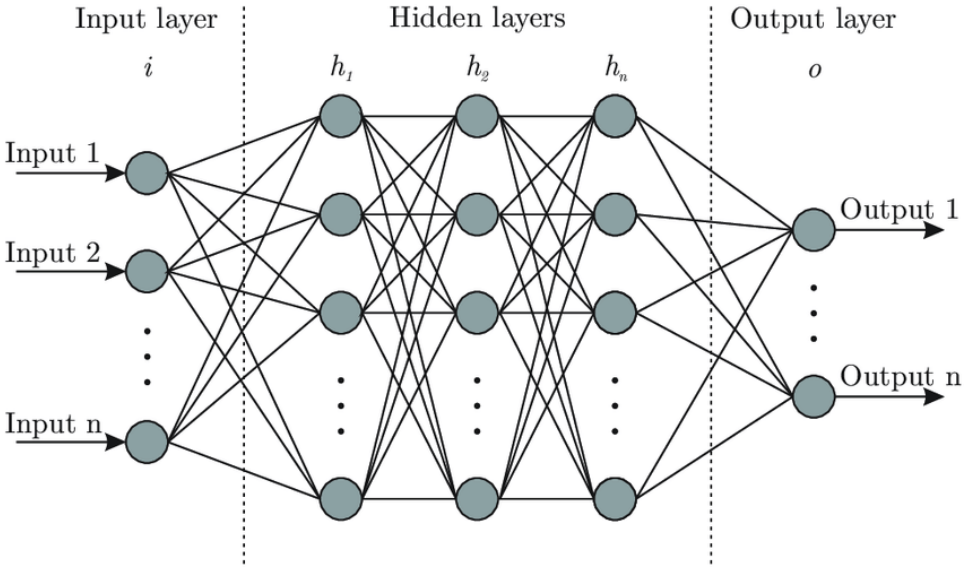

Figure 1. Basic outline of a neural network. It displays the different layers (input, hidden and output), nodes (gray circles) and the connection between the nodes (black lines).

(source: `https://www.researchgate.net/figure/Artificial-neural-network-architecture-ANN-i-h-1-h-2-h-n-o_fig1_321259051`)

We mentioned that the different nodes are connected through weights, biases and activation functions. Looking at figure 1, a neuron j in layer h_1 is connected to n input neurons, denoted by black lines. Each input neuron has a numerical value defined by the problem. The value neuron j in h_1 then gets is defined as

$$\sigma(x_1 w_1 + x_2 w_2 + \cdots + x_3 w_n + b), \tag{2.7}$$

where x_i are the values of neuron i in the input layer, w_i are the weights between neurons i and j, b is what we refer to as the bias and σ is the activation functions (more on them in the next section). Every neuron is connected like this, with different weights and biases. In this project the activation function is the same for each neuron. Note that when we train the data, what we are really doing, is adjusting these parameters to give a desired result. We cover how this is done in the back propagation algorithm section.

2.2.2 Activation Functions

The activation functions are where the non-linearity term comes in, because they are non-linear. Now there are many such functions, in our project we have implemented the ones displayed in figure 2. The exact functions are as follows

$$\text{Sigmoid: } \sigma(x) = \frac{1}{1+e^{-x}} \tag{2.8}$$

$$\text{RELU: } \sigma(x) = \max(0, x) \tag{2.9}$$

$$\text{Leaky RELU: } \sigma(x) = \begin{cases} \alpha x, & \text{if } x \leq 0 \\ z, & \text{otherwise} \end{cases} \tag{2.10}$$

Where α is some parameter which we have set to $\alpha = 0.1$ in figure 2.

2.2.3 Cost Function and Regularization

Before one can start training the data, we must have a cost function.

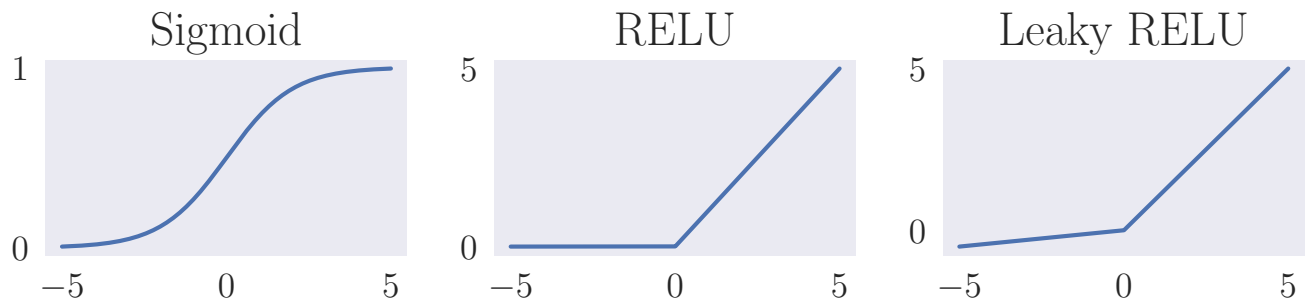

Figure 2. Some activation functions, namely Sigmoid, RELU and Leaky RELU.

2.2.4 The Back Propagation Algorithm

2.3 Logistic Regression

3 Methods

4 Results

5 Discussion

6 Conclusion

A Appendix

References

[1] Pankaj Mehta, Marin Bukov, Ching-Hao Wang, Alexandre G.R. Day, Clint Richardson, Charles K. Fisher, and David J. Schwab. A high-bias, low-variance introduction to machine learning for physicists. *Physics Reports*, 810:1–124, May 2019.