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 4, 2021

ABSTRACT: Abstract coming soon.

Contents

1 Introduction

All the code, results and instructions on running the code can be found in our GitHub repository[1].

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, which tells us how well our model is (more on this here 2.3.3). 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

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

The algorithm for GD is then:

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

where η is what we call the learning rate. This algorithms finds (ideally), with each iteration, new $\boldsymbol{\beta}_{t+1}$ 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. Potentially we can also get stuck in a local minima.

[1] https://github.com/sigurdru/FYS-STK4155/tree/main/project1

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 methods 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 Logistic Regression

Gradient decent is used when we have a continuous output, like fitting a function to data. Logistic Regression is used for classification situations, meaning that we want to predict discrete outputs, i.e. true of false. In our case we want a model that takes in some values $\mathbf{x}_i$ and spits out zero or one. These values should correspond to the actual classification $y_i \in \{0, 1\}$ corresponding to true or false respectively. Lets define our model as

$$\sigma(s_i), \quad \text{where } s_i = \boldsymbol{x}_i^T \boldsymbol{w} + b_0 \equiv \boldsymbol{X}_i^T \boldsymbol{W} \tag{2.7}$$

Where $\boldsymbol{x}_i^T$ is our data/input, $\boldsymbol{w}$ and b_0 are parameters in the model. As a shorthand we also defined $\boldsymbol{W} = (b_0, \boldsymbol{w})$ and $\boldsymbol{X}_i = (1, \boldsymbol{x}_i)$. We also have σ which is some soft classifier that maps our output between zero and one (i.e. the Sigmoid (2.11)). The reason why we want a soft classifier and not a hard one (like $\sigma = 1$ if $s \geq 0$ and 0 otherwise.), is because then we can interpret the output as a probability. Here we also need a cost function to minimize. It is common to choose the cross entropy, which we derive in 2.3.3. However we will just use it without derivation for now. The cross entropy for this model is given as

$$C(\boldsymbol{W}) = \sum_{i=1}^{n} -y_i \log \sigma(\boldsymbol{X}_i^T \boldsymbol{W}) - (1 - y_i) \log \left[1 - \sigma(\boldsymbol{X}_i^T \boldsymbol{W})\right], \tag{2.8}$$

where n are the number of samples we want to classify. Now with a cost function and model in hand we are ready to minimize the cost function in order to find the optimal parameters for the model. We have a convex cost function, therefore a minimization leads to

$$\nabla_{\boldsymbol{W}} C(\boldsymbol{W}) = \sum_{i=1}^{n} \left[\sigma(\boldsymbol{X}_i^T \boldsymbol{W} - y_i)\right] \boldsymbol{X}_i = \boldsymbol{0}. \tag{2.9}$$

Thus the only thing left to do is perform an algorithm similar to 2.5, where we in this case are updating $\boldsymbol{W}_t$.

2.3 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.

2.3.1 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.

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

$$\sigma(x_1 w_1 + x_2 w_2 + \cdots + x_3 w_n + b), \tag{2.10}$$

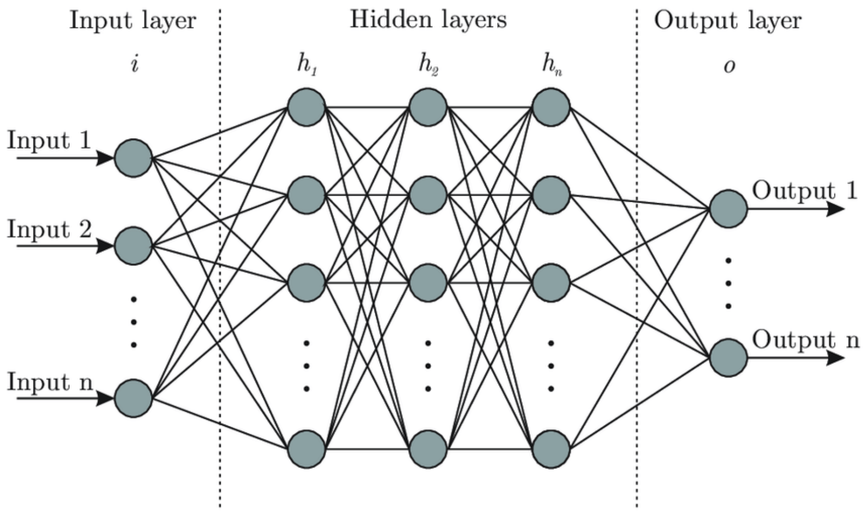

Figure 1. Basic outline of a neural network. It displays the different layers (input, hidden and output), nodes (gray circles) and the connection between the nodes (black lines).

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

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.

2.3.2 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[1] displayed in figure 2. The exact functions are as follows

[1] *FLERE?*

$$\text{Sigmoid: } \sigma(x) = \frac{1}{1+e^{-x}} \tag{2.11}$$

$$\text{RELU: } \sigma(x) = \max(0, x) \tag{2.12}$$

$$\text{Leaky RELU: } \sigma(x) = \begin{cases} \alpha x, & \text{if } x \leq 0 \\ z, & \text{otherwise} \end{cases} \tag{2.13}$$

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

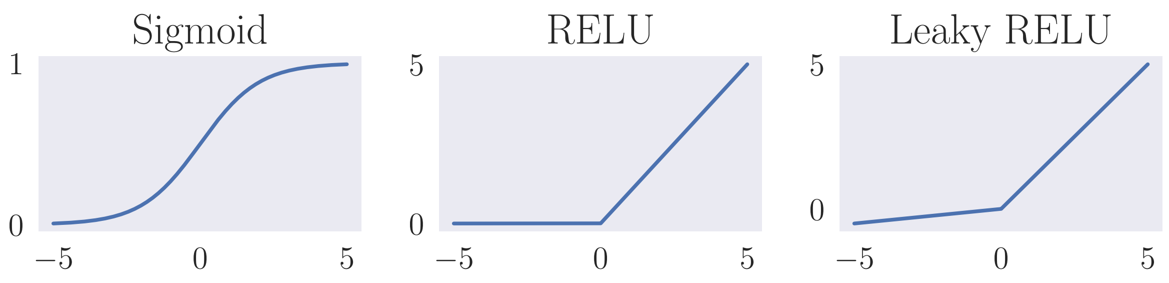

Figure 2. Some activation functions, namely Sigmoid, RELU and Leaky RELU.

2.3.3 Cost Function and Regularization

Before one can start training the data, we must have a cost function. This will tell us how well or poorly our network is performing, and is what we want to minimize when we train the network. For continuous date it is common to use mean square error (MSE) as the cost function. Which is just the difference between desired output ($\hat{\mathbf{x}}$) and actual output ($\mathbf{x}$), squared, divided by the number of datapoints n (see equation (2.14))

$$C(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{\mathbf{x}}_i - \mathbf{x}_i)^2. \tag{2.14}$$

One can also implement regularization, which helps prevent overfitting in the network. Common ones are L_1 and L_2 which is done by adding a regularization term to the end of the cost function (see equations (2.15) and (2.16)).

$$L_1: \quad C(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{\mathbf{x}}_i - \mathbf{x}_i)^2 + \lambda\sum_j |\mathbf{w}_j| \tag{2.15}$$

$$L_2: \quad C(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{\mathbf{x}}_i - \mathbf{x}_i)^2 + \lambda\sum_j \mathbf{w}_j^2 \tag{2.16}$$

[2]Here λ is some regularization parameter and $\mathbf{w}$ are weights.

[2] *RIKTIG?*

For classification scenarios one will often use cross-entropy as the cost function. This is given by equation (2.17).

$$C(\mathbf{x}) = \tag{2.17}$$

lol[3]

[3] *NOT DONE*

2.3.4 The Backpropagation Algorithm

With a desired cost function we are ready to train the neural network. This is done by the backpropagation algorithm. The method entails finding the derivative of the cost function, with respect to all parameters. When we have a neural network, we have thousands of parameters which can be tuned (weights and biases), meaning that we have to approximate the derivative somehow. The backpropagation algorithm does just that, by exploiting the layered structure displayed in figure 1.

Before we can embark on deriving the algorithm we will introduce some notation. We assume L total layers while $l = 1, \ldots, L$ indexes which one. Next we need to index the weights, nodes and biases. Let w_{jk}^l be the weight connecting k-th neuron in layer $l-1$ and j-th neuron in layer l. The index order in j and k are such that we can do matrix multiplication with index notation later down the road. We also let b_j^l be the j-th neuron bias in layer l. Thus the activation of the j-th neuron in layer l (a_j^l) becomes

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right) = \sigma(z_j^l), \quad z_j^l \equiv \sum_k w_{jk}^l a_k^{l-1} + b_j^l. \tag{2.18}$$

Here σ is an activation function.

Now the cost function will depend directly on the activation of the output layer (a_j^L). However the activation of the output layer depends on the previous layers, meaning that the cost function depends indirectly on all the previous layers. Lets define define the error Δ_j^L of the j-th neuron in layer L, as the change in cost function with respect to z_j^L.

$$\Delta_j^L \equiv \frac{\partial C}{\partial z_j^L} \tag{2.19}$$

We can similarly define the error of neuron j in layer l (Δ_j^l), as the change in the cost function with respect to z_j^l,

$$\Delta_j^l \equiv \frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l}\frac{d\sigma(z_j^l)}{dz_j^l}. \tag{2.20}$$

In the next few lines we are going to derive several equations needed for the algorithm, it will be apparent why after we have found them. Notice that (2.20) also can be written as

$$\Delta_j^l = \frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial b_j^l}\frac{\partial b_j^l}{\partial z_j^l} = \frac{\partial C}{\partial b_j^l}. \tag{2.21}$$

Because $\partial b_j^l / \partial z_j^l = 1$ from (2.18). Again using the chain rule we can rewrite (2.20)

$$\begin{aligned}\Delta_j^l = \frac{\partial C}{\partial z_j^l} &= \sum_k \frac{\partial E}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \Delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_j^l} \\ &= \left(\sum_k \Delta_k^{l+1} w_{kj}^{l+1}\right)\frac{d\sigma(z_j^l)}{dz_j^l}.\end{aligned} \tag{2.22}$$

To find the last equation, we differentiate the cost function with respect to the weight w_{jk}^l

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^j} = \Delta l a_k^{l-1}. \tag{2.23}$$

Now why have we done all this work, well because the equations (2.20), (2.21), (2.22) and (2.23) define what we call the backpropagation algorithm. Then, what exactly is the algorithm? In entails six steps:

1 Activation: First activate the neurons in the activation layer (a_j^1) with desired data.

2 Feedforward: Activate the nodes in following layers, this is done by equation (2.18).

3 Error at layer L: Calculate the error at the last layer using (2.20).

4 Backpropagate error: With (2.22) we can the calculate the error, iterating backwards in the network.

5 Calculate gradient: Find the gradient by using equations (2.21) and (2.23).

6 Update parameters: Update the parameters similarly to (2.5), however $\boldsymbol{\beta}_t$ are our weights and biases in this case.[4]

[4] *RIKTIG?*

One thing we haven't mentioned is how one initializes the network. In order to do the first step, activation, we have to give the network some initial weights and biases. [5]

[5] *HOW?*

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.