FYS-STK4155 - Applied data analysis and machine learning

Project 2

Even M. Nordhagen

November 8, 2018

- Github repository containing programs and results:

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

Abstract

The aim of this project is to study the performance of linear regression in order to fit a two dimensional polynomial to terrain data. Both Ordinary Least Square (OLS), Ridge and Lasso regression methods were implemented, and for minimizing Lasso's cost function Gradient Descent (GD) was used. A fourth method was to minimize the cost function of Ridge using GD. The fitted polynomial was visualized and compared with the data, the Mean Square Error (MSE) and R^2-score were analyzed, and finally the polynomial coefficients were studied applying visualization tools and Confidence Intervals (CI). To benchmark the results, we used Scikit Learn.

We found the self-implemented OLS and Ridge regression functions to reproduce the benchmarks, and Lasso was close to reproducing the benchmark as well. However, the difference between results produced by standard Ridge regression and when minimizing its cost function is large. The OLS regression method is considered as the most successful due to its small MSE and high R^2-score.

Contents

1 Introduction

As late as in 1944 Ising model, the Norwegian-American physicist Lars Onsager

We will in this project classify whether a two-dimensional Ising grid is above or below the critical temperature with deep learning as a tool. Feed-Forward Neural Networks (FNN) is our tool of choice, and we investigate the performance as the sigmoid function is replaced with more modern activation functions and we adjust the hyperparameters. Various minimization methods will also be studied.

2 Theory

2.1 Linear regression

In linear regression, the dependent variable y_i is a linear combination of the parameters, and for a dependent variable this can be written as

$$y_i = \sum_j X_{ij}\beta_j. \tag{1}$$

In principle, x_{ij} can be an arbitrary function of the arguments x_i, and in this project the x_{ij}'s are all spin interactions in the Ising model ($x_{ij} = s_i s_j$).

We will study Ordinary Least Square (OLS) regression, Ridge regression and Lasso regression. The OLS cost function is known as

$$c(\vec{\beta}) = \sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\right)^2 \qquad \text{OLS} \tag{2}$$

which is minimized when

$$\vec{\beta} = (\hat{X}^T\hat{X})^{-1}\hat{X}^T\vec{y}. \tag{3}$$

Similarly, the Ridge cost function is

$$c(\vec{\beta}) = \sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\right)^2 + \lambda\sum_{j=1}^{p}\beta_j^2 \qquad \text{Ridge} \tag{4}$$

where λ is called the penalty. This is minimized when

$$\vec{\beta} = (\hat{X}^T\hat{X} + \lambda I)^{-1}\hat{X}^T\vec{y}, \tag{5}$$

and finally the Lasso cost function is given by

$$c(\vec{\beta}) = \sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\right)^2 + \lambda\sum_{j=1}^{p}\beta_j \qquad \text{Lasso.} \tag{6}$$

Linear regression is detailed in the project 1 report, [1].

2.2 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 set the output directly to a binary value. 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 solve every *linear problem*. A drawing of the perceptron is found in figure (1).

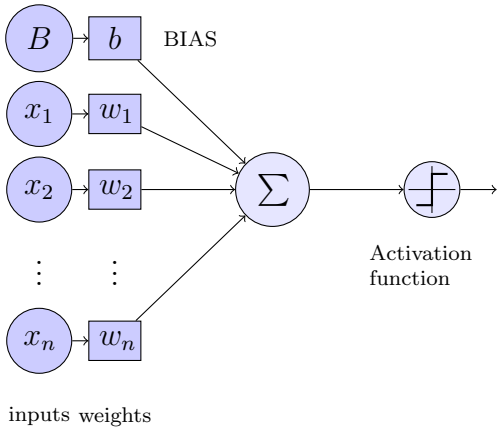


Figure 1: Logistic regression model with n inputs.

Initially, one needs to train the network 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, where the weights usually are set to small random numbers. Then the error is calculated, and the weights are updated to minimize the error. So far so good.

2.2.1 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

$$net = \sum_{i=1}^{I} x_i \cdot w_i + b \cdot 1. \tag{7}$$

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, to make the algorithm easy to implement, mapping the net output to a final output is standard practice. You do not need to care too much about this right now, the mapping is done with an activation function and is explained further in section 2.2.5. The activation function, f, takes in the net output and gives the output,

$$out = f(net). \tag{8}$$

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

2.2.2 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]

2.2.3 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.

2.2.4 Cost function

The cost function is what defines the error, and we could in principle have used OLS, ridge and so on. However, in logistic regression, the cross-entropy function is known to give good results and we will mainly stick to it [3]:

$$c(\boldsymbol{W}) = -\sum_{i=1}^{n} \Big[y_i \log f(\boldsymbol{x}_i^T \boldsymbol{W}) + (1 - y_i) \log[1 - f(\boldsymbol{x}_i^T \boldsymbol{W})] \Big] \tag{9}$$

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

$$p(y|x) = \hat{y}^y \cdot (1 - \hat{y})^{1-y}. \tag{10}$$

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

$$\log \Big[p(y|x) \Big] = \log \Big[\hat{y}^y \cdot (1 - \hat{y})^{1-y} \Big] \tag{11}$$

$$= y \log \hat{y} + (1 - y) \log(1 - \hat{y}) \tag{12}$$

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. [https://towardsdatascience.com/logistic-regression-detailed-overview-46c4da4303bc]

2.2.5 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

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{13}$$

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

$$\frac{df(x)}{dx} = x(1 - x). \tag{14}$$

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

2.2.6 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 few 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,

$$\boldsymbol{w}^+ = \boldsymbol{w} - \eta \cdot \frac{\partial c(\boldsymbol{w})}{\partial \boldsymbol{w}}, \tag{15}$$

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

$$\frac{\partial c(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{\partial c(\boldsymbol{w})}{\partial out} \cdot \frac{\partial out}{\partial net} \cdot \frac{\partial net}{\partial \boldsymbol{w}} \tag{16}$$

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

$$\frac{\partial c(\boldsymbol{w})}{\partial out} = -\frac{y}{out} + \frac{1-y}{1-out}. \tag{17}$$

Further, the second derivative is the derivative of the activation function with respect to the output, which is given in (14)

$$\frac{\partial out}{\partial net} = out(1-out). \tag{18}$$

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

$$\frac{\partial net}{\partial \boldsymbol{w}} = \boldsymbol{x}. \tag{19}$$

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

$$\frac{\partial c(\boldsymbol{w})}{\partial \boldsymbol{w}} = [f(\boldsymbol{x}^T\boldsymbol{w}) - \boldsymbol{y}]\boldsymbol{x} \tag{20}$$

and obtain a weight update algorithm

$$\boldsymbol{w}^+ = \boldsymbol{w} - \eta \cdot [f(\boldsymbol{x}^T\boldsymbol{w}) - \boldsymbol{y}]^T\boldsymbol{x}. \tag{21}$$

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

2.3 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. [https://compphysics.github.io/MachineLearning/doc/pub/NeuralNet/html/NeuralNet.html] However, often multiple layers are used since this often gives 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.

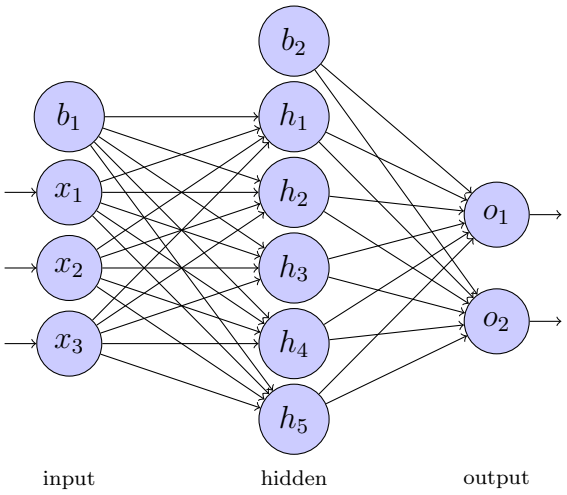


Figure 2: Neural network with 3 input nodes, 5 hidden nodes and 2 output nodes, in addition to the bias nodes.

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 2.3.3, where one of the most popular techniques for that is discussed. Before that we will generalize the forward phase presented in logistic regression.

2.3.1 Forward phase

In section 2.2.1, 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. When it comes to a neural network, it is more convenient 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

$$net_{h,j} = \sum_{i=1}^{I} x_i \cdot w_{ij}^{(1)} = \boldsymbol{x}^T \boldsymbol{W}^{(1)} \tag{22}$$

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

$$h_j = f(net_{h,j}). \tag{23}$$

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

$$net_{o,j} = \sum_{i=1}^{H} h_i \cdot w_{ij}^{(2)} = \boldsymbol{h}^T \boldsymbol{W}^{(2)} \tag{24}$$

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

$$net_{h_l,j} = \boldsymbol{h}^{(\boldsymbol{l-1})^T} \boldsymbol{W}^{(l)}. \tag{25}$$

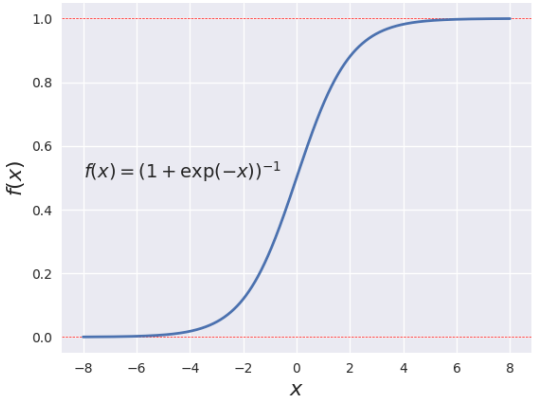
2.3.2 Activation function

Before 2012, the logistic and the tanh 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.

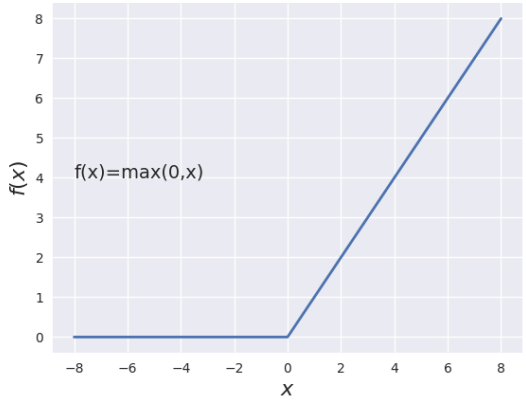
In figure (3), *standard RELU, leaky RELU* and *ELU* are plotted along with the sigmoid function.

2.3.3 Backward Propagation

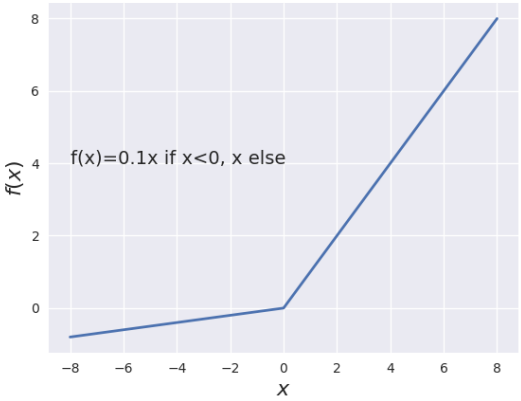
Backward propagation is probably the most used technique to update the weights, and is actually again based on equation (15). What differs, is the differentiation



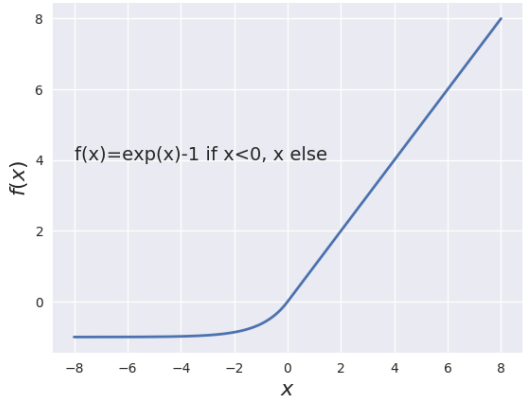
(a) Sigmoid



(b) ReLU



(c) Leaky ReLU



(d) ELU

Figure 3: Some more or less popular activation functions

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 set of weights, where the last layer is updated in a similar way as for a network without hidden layer...

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

$$w_{ij}^{(1)} \rightarrow w_{ij}^{(1)} - \eta \cdot \sum_{k=1}^{O} \delta_{ok} \cdot w_{jk}^{(2)} \cdot out_{hj}(1 - out_{hj}) \cdot x_i \tag{26}$$

where we recall δ_{ok} as

$$\delta_{ok} = -(t_{ok} - out_{ok}) \cdot out_{ok}(1 - out_{ok}).$$

2.3.4 Summary

Since it will be quite a lot calculations, I will just express the results here, and move the calculations to Appendix C. Let us start with the forward propagation:

$$\begin{aligned}
net_{hi} &= \sum_j w_{ji}^{(1)} \cdot i_j + b_{1i} \cdot 1 \\
out_{hi} &= \text{sig}(net_{hi}) \\
\\
net_{ki} &= \sum_j w_{ji}^{(2)} \cdot out_{hj} + b_{2i} \cdot 1 \\
out_{ki} &= \text{sig}(net_{ki}) \\
\\
net_{oi} &= \sum_j w_{ji}^{(3)} \cdot out_{kj} + b_{3i} \cdot 1 \\
out_{oi} &= \text{sig}(net_{oi})
\end{aligned} \tag{27}$$

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

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

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

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

where we again use the short hand

$$\delta_{oj} = (t_j - out_{oj}) \cdot out_{oj}(1 - out_{oj}).$$

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.

Now over to the problem we want to solve using neural networks.

2.4 Ising model

The Ising model is widely used to study phenomena in statistics, physics, economics etc..

2.4.1 One dimension

In one dimension, it can be viewed as a sequence of two binary numbers, for instance ± 1, which we will call spins. Each element interacts with its nearest neighbors by the formula

$$E_{i,i+1} = J_{i,i+1} s_i s_{i+1} \tag{28}$$

where s_i and s_{i+1} are the values of the respective elements and $J_{i,i+1}$ is the interaction coefficient. The total energy of an Ising model is just the sum of all interaction energies, which becomes

$$E = \sum_i J_{i,i+1} s_i s_{i+1}. \tag{29}$$

This particular model is basically setting the interaction between two non-neighbors to zero, but we could generalize this:

$$E = \sum_{ij} J_{i,j} s_i s_j. \tag{30}$$

where J now is an interaction matrix. We recognize this as an inner product where $\boldsymbol{x}$ is a vector containing all $s_i s_j$ elements and $\boldsymbol{J}$ as the J-matrix discussed above flatten out,

$$E = \boldsymbol{x}^T \boldsymbol{J}. \tag{31}$$

We could extend this to a system of n Ising models, by introducing a matrix $\boldsymbol{X} = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n]$ which contains all $\boldsymbol{x}$-vectors

$$\boldsymbol{E} = \boldsymbol{X}^T \boldsymbol{J}. \tag{32}$$

If we now know the spin configuration of the models ($\boldsymbol{X}$) and the corresponding energies, we can estimate the $\boldsymbol{J}$-vector using linear regression.

2.4.2 Two dimensions

In two dimensions, each spin has four neighbors instead of two, which increases the calculation difficulties significantly. In this case, the Ising model is commonly written as

$$E = \sum_{<ij>} J_{i,j} s_i s_j \tag{33}$$

where the notation <> restricts the interactions to be between nearest neighbors. In a two-dimensional Ising model, phase transitions can occur, which was first described analytically by Lars Onsager. He showed that the phase transition occurs at

$$T_c = \frac{2J}{k \ln(1+\sqrt{2})} \approx 2.269 \tag{34}$$

for a 2D Ising grid without boundaries. The Ising model we deal with is limited, such that we cannot avoid the boundary problems. The critical temperature is then $T_c \approx 2.3$ in units of energy.

3 Methods

3.1 Resampling techniques

A resampling technique is a way of estimating the variance of data sets without calculating the covariance. As we saw in section section 3.3, the true covariance

is given by a double sum which we will avoid calculating if possible. There are different ways of doing this, and we have already went through several of them in this course:

- Jackknife resampling
- K-fold validation
- Bootstrap method
- Blocking method.

For this particular project we have been focusing on the bootstrap and the K-fold validation methods, so only they will be covered here.

3.1.1 Bootstrap method

When we construct a data set, we usually draw samples from a probability density function (PDF) and get a set of samples $\vec{x}$. If we draw a large number of samples, the sample variance will approach the variance of the PDF. The bootstrap method turns this upside down, and tries to estimate the PDF given a set data set, because if we know the PDF, we know in principle everything about the data set.

The assumption we need to make, is that the relative frequency of x_i equals $p(x_i)$, which is reasonable (for instance, think about how the histogram looks like when we draw samples from a normal distribution). In this project the vector that we want to find, $\vec{\beta}(x, y)$, is a function of two set of variables. Fortunately, they are independent of eachother, so we can safely apply the independent bootstrap on them separately. The independent bootstrap goes as

1. Draw n samplings from the data set $\vec{x}$ with replacement and denote the new data set as $\vec{x}^* = \{x_1, x_2, \ldots, x_n\}$
2. Compute $\vec{\beta}(\vec{x}^*) \equiv \vec{\beta}^*$
3. Repeat the procedure above K times
4. The average value of all K $\vec{\beta}^*$'s are stored in a new vector $\vec{\bar{\beta}}^*$
5. Finally, the variance of $\vec{\bar{\beta}}^*$ should correspond to the sample variance

[9]

The implementation could look something like this

```
def bootstrap(data, K=1000):
dataVec = np.zeros(K)
for k in range(K):
dataVec[k] = np.average(np.random.choice(data, len(data)))
Avg = np.average(dataVec)
Var = np.var(dataVec)
Std = np.std(dataVec)

return Avg, Var, Std
```

which is strongly inspired by the lecture notes. [10]

It is also worth to mention that we should hold back a small part of the original data set for testing, since we need to test the model on data that it has not seen before.

3.1.2 K-fold validation method

K-fold validation is a method that has more describing name than many of its fellow methods. The idea is to make the most use of our data by splitting it into k folds and training k times on it. Every time we train, we leave out one of the folds, which gonna be our validation data. This validation data needs to be different every time, and we are therefore restricted to k unique training sessions. A typical overview looks like this:

- Split data set into k equally sized folds
- Use the $k-1$ first folds as training data, and leave the k'th fold for validation
- Use the $k-2$ first folds plus the k-th fold as training data, and leave the $(k-1)$'th fold for validation
- Continue until all folds are used as training data

[7]. An example implementation of K-fold validation, and in fact the function used in this project, can be seen below.

```
def k_fold(x, y, z, K=10):
xMat = np.reshape(x, (K, int(len(x)/K)))      # Reshape x-coords
yMat = np.reshape(y, (K, int(len(y)/K)))      # Reshape y-coords
zMat = np.reshape(z, (K, int(len(z)/K)))      # Reshape z-coords

MSE_train = 0                                   # Declare MSE_train variable
MSE_test  = 0                                   # Declare MSE test variable
R2_train  = 0                                   # Declare R2 train variable
R2_test   = 0                                  # Declare R2 test variable

for i in range(K):
xMatNew = np.delete(xMat, i, 0)              # Ignore test sets
yMatNew = np.delete(yMat, i, 0)              # Ignore test sets
zMatNew = np.delete(zMat, i, 0)              # Ignore test sets
```

```
xVecNew = xMatNew.flatten()                    # Reshape training sets
yVecNew = yMatNew.flatten()                    # into vectors
zVecNew = zMatNew.flatten()

order5 = Reg_2D(xVecNew, yVecNew, zVecNew, Px=5, Py=5)
beta_train = order5.ols()

MSE_train += MSE(xVecNew, yVecNew, zVecNew, beta_train)
MSE_test += MSE(xMat[i], yMat[i], zMat[i], beta_train)

R2_train += R2(xVecNew, yVecNew, zVecNew, beta_train)
R2_test += R2(xMat[i], yMat[i], zMat[i], beta_train)

return MSE_train/K, MSE_test/K, R2_train/K, R2_test/K
```

3.2 Minimization methods

Suppose we have a very simple model trying to fit a straight line to data points. In that case, we could manually vary the coefficients and find a line that fits the points quite good. However, when the model gets more complicated, this can be a time consuming activity. Would it not be good if the program could do this for us?

In fact there are multiple techniques for doing this, where the most complicated ones obviously also are the best. Anyway, in this project we will have good initial guesses, and are therefore not in need for the most fancy algorithms. We will stick to gradient descent in this project, which might be the simplest method for our purposes.

3.2.1 Gradient Descent

The Gradient Descent method was mentioned already in the theory part, in equation (15), and this will therefore just be a quick reminder of the idea. Perhaps the simplest and most intuitive method for finding the minimum is the gradient descent method (GD), which reads

$$\beta_i^{\text{new}} = \beta_i - \eta \cdot \frac{\partial Q(\beta_i)}{\partial \beta_i} \tag{35}$$

where β_i^{new} is the updated β and η is a step size, in machine learning often refered to as the learning rate. The idea is to find the gradient of the cost function $Q(\vec{\beta})$ with respect to a certain β_i, and move in the direction which minimizes the cost function. This is repeated until a minimum is found, defined by either

$$Q(\beta_i) < \varepsilon \tag{36}$$

or that the change in β_i for the past x steps is small. The algorithm for this minimization method is thus as follows:

```
while dbeta > epsilon:
e = z - X.dot(beta)
debeta = 2*X.T.dot(e) -np.sign(beta)*q*lambda*np.power(abs(beta), q-1)
beta += eta*dbeta
```

3.2.2 Stochastic Gradient Descent (SGD)

It will probably not converge in fewer steps, but each step will be faster.

$$\beta_i^{\text{new}} = \beta_i - \eta \cdot \frac{\partial Q(\beta_i)}{\partial \beta_i} \tag{37}$$

3.3 Error analysis

To find out how good the fit is, we could potentially compare the polynomial plot to the data plot and decide whether the fit is good or not. However, that approach is risky in the sense that it is hard to determine how good a fit really is, and we therefore rely on error estimates. There are many ways to calculate the error, where some frequently used methods are

- the absolute error
- the relative error
- the mean square error (MSE)
- the R^2 score function
- the accuracy score function

In this report we will study only the MSE and R^2 score function.

3.3.1 Mean Square Error (MSE)

The most popular error estimation method is the mean square error, also called least squares. We study the MSE in order to compute the reduction of the cost function, because it is basically the standard cost function, used in OLS.

$$\text{MSE}(\vec{\beta}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \tilde{y}(\vec{\beta}))^2 \tag{38}$$

Compared to least absolute value, the points far away from the fitted line are weighted stronger.

3.3.2 R^2 score function

The R^2 score function is a measure of how close the data are to the fitted regression line, and is a widely used quantity within statistics. In entirety it reads

$$R^2(\vec{y}, \tilde{\vec{y}}) = 1 - \frac{\sum_{i=1}^{N}(y_i - \tilde{y})^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}. \tag{39}$$

and is a ratio between the explained variation and total variation. If it is, on one hand, able to explain all the variations the score is 1, which is the best. On the other hand, if it is not able to describe any of the variations, the R^2-score is low. We are therefore fighting for a high R^2-score.

3.3.3 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,

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(y_i = t_i)}{n}. \tag{40}$$

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

4 Code

The code is mainly implemented in Python, due to its neatness and flexibility. The numpy package has a lot of functions which are both fast and convenient for machine learning purposes, and there exist some packages that provide easy and fast machine learning tools such as Scikit-Learn, Keras and Tensorflow. However, the neural networks that we implement from scratch will not be as fast in Python as in a low-level language, and they where therefore implemented in C++ as well.

4.1 Code structure

4.2 Algorithm

The actual algorithm consists of the results above, and we need to order them correctly. To increase the performance and the neatness, I will present a vectorized implementation (i.e. use linear algebra to solve it instead of sum over piecewise

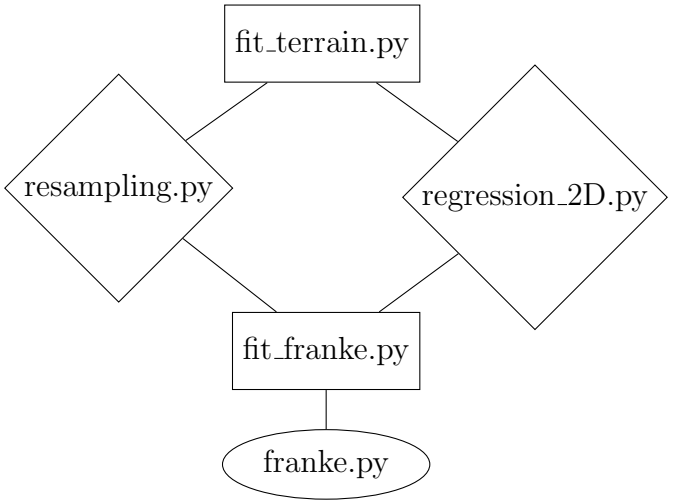


Figure 4: Code structure

elements). For the forward propagation (when we get the outputs), it is easy to see that it can be vectorized,

$$net_j = \sum_i (w_{ij} \cdot X_i + b_i) \quad \Rightarrow \quad net = W \cdot X + b. \tag{41}$$

where *net* is a vector. Also the backward propagation (when we update the weights) can be vectorized. Firstly the differentiation of the error function can be vectorized:

$$\frac{\partial E_{TOT}}{\partial out_i} = -(t_i - out_i) \quad \Rightarrow \quad \frac{\partial E_{TOT}}{\partial out} = -(t - out) \tag{42}$$

where we have vectors on both sides. Those vectors will have the same length as $out(1 - out)$, such that we can define a quantity δ^o and apply the Hadamard multiplication:

$$\delta^o = \frac{\partial E_{TOT}}{\partial out} \odot \frac{\partial \text{sigmoid}(out)}{\partial out} = -(t - out) \odot out(1 - out) \tag{43}$$

which will be used in both the weight update and the bias update. To fulfil the weight update, we need to take the outer product of δ^o with the inputs X_i, and we then have $\partial E_{TOT}/\partial W$, which of course has the same dimensions as the weight matrix.

$$W^+ = W - \eta(\delta^o \otimes X) \tag{44}$$

$$b^+ = b - \eta \cdot \delta^o \tag{45}$$

It should look something like this:

- **Declare all variables, define inputs, targets and quantities, and initialize the weights**
  #Iterations = ...
  Eta = ...

  X = [[...] ... [...]]
  t = [[...] ... [...]]

  W = 2*random(0,1) - 1
  b = 2*random(0,1) - 1

- **Training - calculate outputs for all samples and then update weights. Repeat #Iterations times**
  for i ¡ #Iterations
  for j ¡ #Samples
  net = X*W + b
  out = sigmoid(net)

  W = W + eta*(t - out)*out(1-out)*X
  b = b + eta*(t - out)*out(1-out)

- **Recall - Use the weight to find answer to problems with unknown answers**
  X = [[...] ... [...]] Inputs with unknown outputs
  net = X*W + b
  ans = sigmoid(out)

4.3 Implementation

For the algorithms above, there exist huge possibilities for vectorization, and basically all the loops can be avoided using vectorization provided by numpy.

```
def linear(X, t, T, eta = 0.1):

I = len(X[0])
O = len(t[0])
M = len(X)

W = 2*np.random.random([I, O]) - 1
b = 2*np.random.random(O) - 1

for iter in range(T):
for i in range(M):
```

```
net = np.dot(X[i], W) + b
out = sigmoid(net)

deltao = -(t[i] - out)*sig_der(out)
W = W - eta * np.outer(np.transpose(X[i]), deltao)
b = b - eta * deltao
return W, b
```

where sigmoid is the sigmoid function used in the weight calculations above and sig_der is its derivative. To get this working, the input array X and the target array t need to be numpy arrays, and a call to the function can be done like this

```
X = np.array([[0,0], [0,1], [1,0], [1,1]])
t = np.array([[0], [1], [1], [1]])

W, b = linear(X, t, 1000)
```

which can be recognized as an OR gate. The logic functions are popular examples when it comes to neural network, because (1) they are easy to work with (2) they display the constraints of a single perceptron in a easy way. In next section we are going to discuss this, and why we need multi-layer perceptrons for certain logic functions. However, back to the implementation. Since we now know the weights, we can use them to recall, which is simply done by

```
X_ = np.array([0,1])
net = np.dot(X_, W) + b
out = sigmoid(net)
print(out)
>>> 1
```

which should give 1 according to the OR gate. We could train this network to understand NOR, AND and NAND gates as well, but what happens if you try to teach it the XOR gate?

For complete programs, please visit
`https://github.com/evenmn/Machine-Learning`.

5 Results

Some general stuff here

5.1 Determine the energy in the Ising model

We have determined the energy of the Ising model using both linear regression (OLS, Ridge and Lasso) and neural networks, where we first train the model on a set, and then validate the model on a test set. In project 1, we found our implementation of OLS, Ridge and Lasso to give the same result as Scikit Learn, and we will in this project stick to Scikit Learn due to its quickness. [Ref. project 1] Let's start with linear regression.

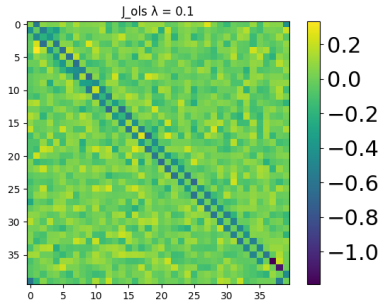
5.1.1 Linear regression

We used linear regression to find the $\boldsymbol{J}$-matrix, and from that we found the energies. In table (1), the errors between obtained energy and true energy are listed for the training set, the test set and the K-fold cross-validation resampling. Further, the $\boldsymbol{J}$-matrix is visualized in figure (5) and the R^2-score is plotted as function of λ in figure (6).

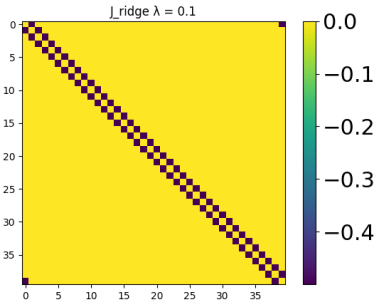
Table 1: Mean Square Error and R^2-score presented for OLS, Ridge and Lasso on the Ising model. 'Train' means the results obtained from the training set, containing 8000 states, and 'Test' means the results obtained from the test set, containing 2000 states. The 'K-fold'-columns are results from K-fold resampling with 10 folds. For Ridge and Lasso, we used $\lambda = 1e-3$ (penalty). See text for more information.

	MSE			**R2**		
	Train	Test	K-fold	Train	Test	K-fold
OLS	0.3006	0.3006	0.3739	0.9925	0.9927	0.9906
Ridge	1.731e-13	2.150-13	1.618e-13	1.000	1.000	1.000
Lasso	4.029e-5	4.189e-5	4.0346e-5	1.000	1.000	1.000

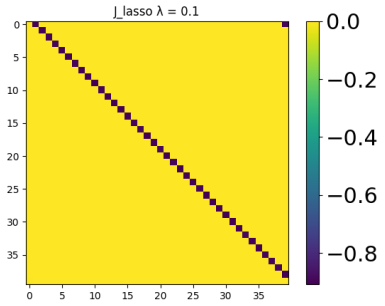
We observe that the MSE is quite low, especially for Ridge and Lasso. The R^2-score is correspondingly high. Below, in figure (5), the $\boldsymbol{J}$-matrix is visualized for all the linear regression methods and $\lambda \in [1e-1, 1e-6]$.



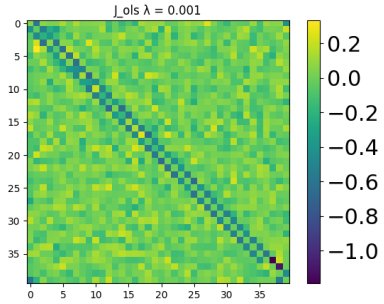
(a) OLS, $\lambda = 0.1$



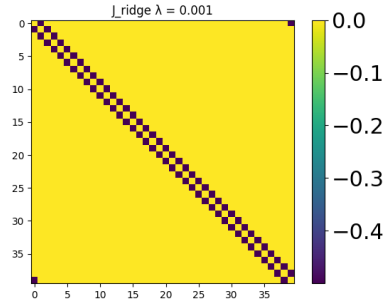
(b) Ridge, $\lambda = 0.1$



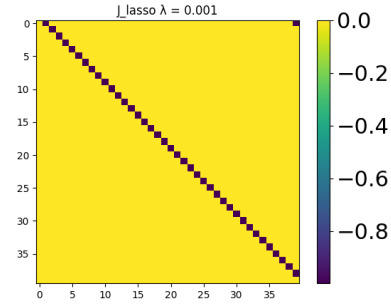
(c) Lasso, $\lambda = 0.1$



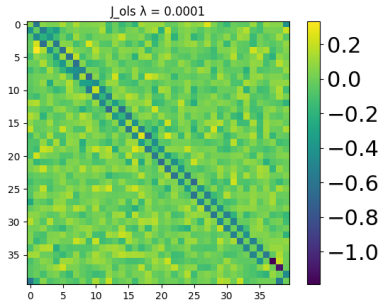
(d) OLS, $\lambda = 0.001$



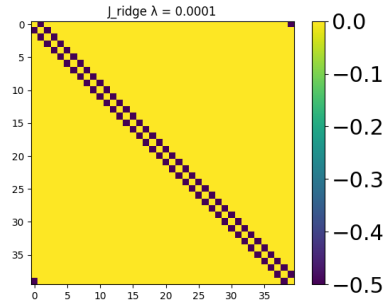
(e) Ridge, $\lambda = 0.001$



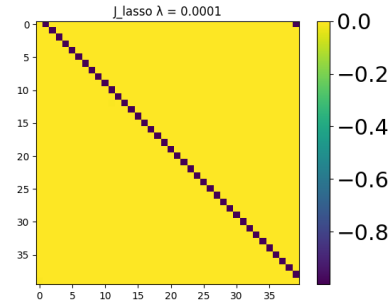
(f) Lasso, $\lambda = 0.001$



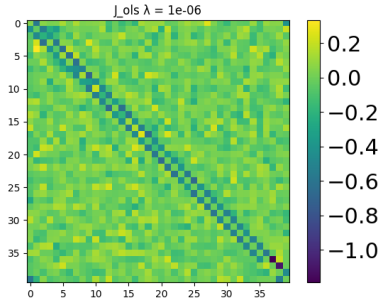
(g) OLS, $\lambda = 0.0001$



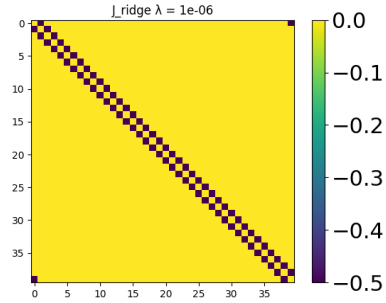
(h) Ridge, $\lambda = 0.0001$



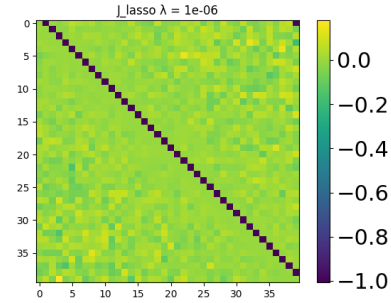
(i) Lasso, $\lambda = 0.0001$



(j) OLS, $\lambda = 0.000001$



(k) Ridge, $\lambda = 0.000001$



(l) Lasso, $\lambda = 0.000001$

Figure 5: The J-matrix obtained from OLS, Ridge and Lasso with $\lambda \in [1e-1, 1e-6]$. The models where trained on 8000 randomly chosen Ising states.

We observe that the $\boldsymbol{J}$ obtained form Lasso is superdiagonal, while for Ridge it is sub-superdiagonal. For OLS, the matrix tends to be sub-superdiagonal, but the off-diagonal elements are not fully zero.

Finally, the R^2-score is plotted as a function of the penalty, λ, in figure (6).

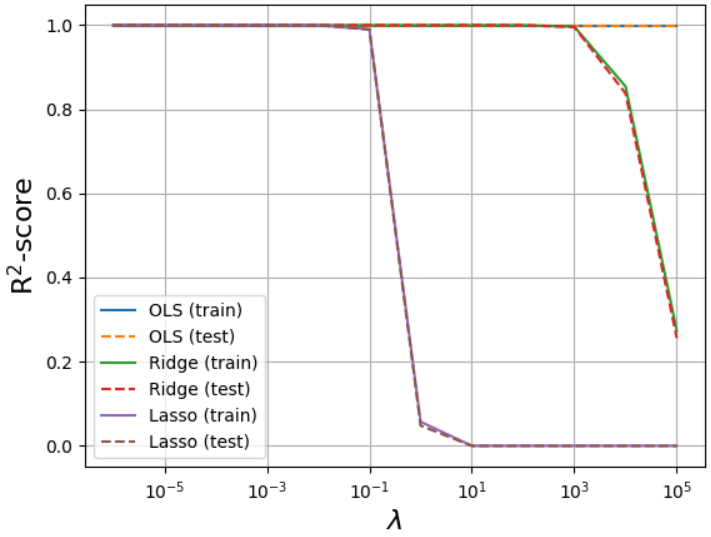


Figure 6: The R^2-score as a function of the penalty.

5.1.2 Neural networks

We now repeat what we have done above, jump over to a neural network approach.

Table 2: Mean Square Error and R^2-score presented for OLS, Ridge and Lasso on the Ising model. 'Train' means the results obtained from the training set, containing 8000 states, and 'Test' means the results obtained from the test set, containing 2000 states. The 'K-fold'-columns are results from K-fold resampling with 10 folds. For Ridge and Lasso, we used $\lambda = 1e-3$ (penalty). See text for more information.

	MSE			**R2**		
Hidden nodes	Train	Test	K-fold	Train	Test	K-fold
0	3.529e-4	4.454e-4	3.610e-4	1.000	1.000	1.000
2	0.009128	0.009651	0.009128	0.8977		
5	0.01439	0.01489	0.01555	0.8387		
10	0.01439	0.01489	0.01555	0.8387		
2 + 2	0.01439	0.01489	0.01555	0.8387		
5 + 5	0.01439	0.01489	0.01555	0.8387		
10 + 10	0.01439	0.01489	0.01555	0.8387		
10 + 10 + 10	0.01439	0.01489	0.01555	0.8387		

5.2 Classifying the phase

Removed states around critical temperature.

5.2.1 Logistic regression

Table 3: Accuracy score

	GD		**SGD**	
Iterations	Train	Test	Train	Test
10	3.529e-4	4.454e-4	3.610e-4	1.
100	0.009128	0.009651	0.009128	0.8977
1000	0.01439	0.01489	0.01555	0.8387

Add lambda as accuracy score plot

5.2.2 Neural networks

6 Discussion

Comparing my linear regression results to Mehta, I obviously not get the same results. For some reason my test errors are much less than what he obtained for

the test sets, which is suspicious. Cannot find the error in my code.

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 professor's research revolutionized artificial intelligence. thestar.com, retrieved November 3rd, 2018. (2015).

[6] Simulation stimulation. L. Bastick. https://www.sumproduct.com/thought/simulation-stimulation

[7] Machine Learning Crash Course: Part 4 - The Bias-Variance Dilemma. D. Geng, S. Shih. https://ml.berkeley.edu/blog/2017/07/13/tutorial-4/ (2017).

[8] Regression Analysis: How Do I Interpret R-squared and Assess the Goodness-of-Fit? http://blog.minitab.com/blog/adventures-in-statistics-2/regression-analysis-how-do-i-interpret-r-squared-and-assess-the-goodness-of-fit (2013).

[9] Bootstrap Methods: Another Look at the Jackknife. B. Efron. Ann. Statist., Volume 7, Number 1, 1-26. (1979).

[10] Lecture notes FYS-STK4155: Regression Methods. https://compphysics.github.io/MachineLearning/doc/pub/Regression/pdf/Regression-minted.pdf M. Hjorth-Jensen. (2018).

[11] Scikit-learn: Machine Learning in Python. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thiron, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay. Journal of Machine Learning Research, Volume 12. (2011).

A Appendix A - Derivation of activation functions

A.1 Sigmoid

If we differentiate the sigmoid function

$$f(x) = \frac{1}{1 + \exp(-x)} \tag{46}$$

from equation (**??**), and we will get

$$f'(x) = f(1 - f) \tag{47}$$

PROOF:

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

A.2 Cross-entropy

B Appendix B - Backward propagation

B.1 Single layer

Since the energy is not directly dependent on w_{ij}, we need to do a change of variables such that we get derivatives that we can calculate. Probably the best choice is

$$\frac{\partial E_{TOT}}{\partial w_{ij}} = \frac{\partial E_{TOT}}{\partial out_j} \cdot \frac{\partial out_j}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ij}}. \tag{48}$$

Standard differentiating of the error function from section **??** gives

$$\frac{\partial E_{TOT}}{\partial out_j} = -(t_j - out_j) \tag{49}$$

which is a neat expression as pointed out above. To obtain the second fraction, we need to decide which sigmoid function to use. In the calculations I will use the first mentioned in section 2.2.5, that reads

$$out_j = \frac{1}{1 + \exp(-net_j)}. \tag{50}$$

and we have already found that

$$\frac{\partial out_j}{\partial net_j} = out_j(1 - out_j). \tag{51}$$

Furthermore the netto output is given by equation (7),

$$net_j = \sum_{i=1}^{I} X_i \cdot w_{ij} + b_j \cdot 1, \tag{52}$$

which simply gives

$$\frac{\partial net_j}{\partial w_{ij}} = X_i. \tag{53}$$

We end up with

$$\frac{\partial E_{TOT}}{\partial w_{ij}} = -(t_j - out_j) \cdot out_j(1 - out_j) \cdot X_i \tag{54}$$

and the weight updating formula

$$w_{ij}^{+} = w_{ij} + \eta \cdot (t_j - out_j) \cdot out_j(1 - out_j) \cdot X_i \tag{55}$$

Similarly we need to update the bias weights, with a formula similar to that one in equation (15):

$$b_i^{+} = b_i - \eta \cdot \frac{\partial E_{TOT}}{\partial b_i}. \tag{56}$$

We do the same change of variables and the only thing that changes is the last part, $\partial net_i / \partial b_i$, which can easily be found to be 1. Then

$$b_i^{+} = b_i + \eta \cdot (t_i - out_i) \cdot out_i(1 - out_i) \tag{57}$$

and all the basics for the single perceptron are set.

B.2 Two-layer

Backward propagation is probably the most used technique to update the weights, and is actually based on equation (15), but we neglect the x_i:

$$w_{ij}^{\dagger} = w_{ij} - \eta \cdot \frac{\partial E_{TOT}}{\partial w_{ij}}. \tag{58}$$

To work the partial derivative out, we need to do a change of variables. I will focus on a 2-layer network, but the principle is the same for networks of more layers. The method will be slightly different for the two set of weights (W1 and W2), so I will do it seperately for them. We start with W2, which is naturally since we move backwards (hence backward propagation).

$$\frac{\partial E_{TOT}}{\partial w_{ij}} = \frac{\partial E_{TOT}}{\partial out_{oj}} \cdot \frac{\partial out_{oj}}{\partial net_{oj}} \cdot \frac{\partial net_{oj}}{\partial w_{ij}} \tag{59}$$

$$= \delta_{oj} \cdot \frac{\partial net_{oj}}{\partial w_{ij}} \tag{60}$$

where δ_{oj} is the same for all weights that go to the same output $oj \Rightarrow$ we will have O different δ_{oj}'s. Further recall the error function from equation (**??**), and recognize that

$$\frac{\partial E_{TOT}}{\partial out_{oj}} = -(t_{oj} - out_{oj}) \tag{61}$$

where t_{oj} are the targets. Then we have the sigmoid function where we get out_{oj} when we send in net_{oj}:

$$out_{oj} = \frac{1}{1 + \exp(-net_{oj})} \tag{62}$$

such that

$$\frac{\partial out_{oj}}{\partial net_{oj}} = out_{oj}(1 - out_{oj}) \tag{63}$$

as mentioned in section 1 and proven in Appendix A. We end up with

$$\delta_{oj} = -(t_{oj} - out_{oj}) \cdot out_{oj}(1 - out_{oj}) = \frac{\partial E_{TOT}}{\partial net_{oj}}. \tag{64}$$

We also have that

$$net_{oj} = \sum_k w_{kj} \cdot out_{hk} + b_{2j} \cdot 1 \tag{65}$$

such that

$$\frac{\partial net_{oj}}{\partial w_{ij}} = out_{hi} \tag{66}$$

and

$$\frac{\partial E_{TOT}}{\partial w_{ij}} = \delta_{oj} \cdot out_{hi}. \tag{67}$$

The specific updating algorithm is the following

$$w_{ij}^{(2)} \rightarrow w_{ij}^{(2)} - \eta \cdot \delta_{oj} \cdot out_{hi} \tag{68}$$

with

$$\delta_{oj} = -(t_{oj} - out_{oj}) \cdot out_{oj}(1 - out_{oj})$$

We are now set for updating the last weights W2, see section **??** for a overview of the algorithm, section **??** for a short description of how to vectorize the algorithm, or if you want to go straight to the implementation, you should check out section **??**.

However, we also need to update the remaining weights, W1, what are we waiting for? The approach is the same as above with considering how much the total error will change when changing one of the weights and doing a change of variables

$$\frac{\partial E_{TOT}}{\partial w_{ij}} = \frac{\partial E_{TOT}}{\partial out_{hj}} \cdot \frac{\partial out_{hj}}{\partial net_{hj}} \cdot \frac{\partial net_{hj}}{\partial w_{ij}}. \tag{69}$$

A difference is that the error for each of the hidden nodes is dependent on the error at each output nodes, such that now E_{TOT} needs to be split up in O terms

$$E_{TOT} = E_{o1} + E_{o2} + \ldots + E_{oO}. \tag{70}$$

This causes

$$\frac{\partial E_{TOT}}{\partial out_{hj}} = \frac{\partial E_{o1}}{\partial out_{hj}} + \frac{\partial E_{o2}}{\partial out_{hj}} + \ldots + \frac{\partial E_{oO}}{\partial out_{hj}} \tag{71}$$

where we again need to do a change of variables on each of those

$$\frac{\partial E_{ok}}{\partial out_{hj}} = \frac{\partial E_{ok}}{\partial net_{ok}} \cdot \frac{\partial net_{ok}}{\partial out_{hj}}. \tag{72}$$

We could have done another change of variables above to avoid equation (73), but this is neater

$$\frac{\partial E_{ok}}{\partial net_{ok}} = \frac{\partial E_{ok}}{\partial out_{ok}} \cdot \frac{\partial out_{ok}}{\partial net_{ok}}. \tag{73}$$

Now it gets more similar to the first weight updates again:

$$\frac{\partial E_{ok}}{\partial out_{ok}} = -(t_{ok} - out_{ok}) \tag{74}$$

and

$$\frac{\partial out_{ok}}{\partial net_{ok}} = out_{ok}(1 - out_{ok}). \tag{75}$$

Further we need to calculate $\partial net_{ok}/\partial out_{hj}$, which is done by

$$net_{ok} = \sum_i w_{ik}^{(2)} \cdot out_{hi} + b_{2k} \cdot 1 \tag{76}$$

and

$$\frac{\partial net_{ok}}{\partial out_{hj}} = w_{jk}^{(2)} \tag{77}$$

and we end up with

$$\frac{\partial E_{ok}}{\partial out_{hj}} = -(t_{ok} - out_{ok}) \cdot out_{ok}(1 - out_{ok}) \cdot w_{jk}^{(2)}. \tag{78}$$

That was the difficult part, as we have seen before

$$\frac{\partial out_{hj}}{\partial net_{hj}} = out_{hj}(1 - out_{hj}) \tag{79}$$

and since

$$net_{hj} = \sum_l w_{lj} \cdot x_l \tag{80}$$

where x_i is input i, we obtain

$$\frac{\partial net_{hj}}{\partial w_{ij}} = x_i \tag{81}$$

and we finally end up with

$$\frac{\partial E_{TOT}}{\partial w_{ij}^{(1)}} = \sum_{k=1}^{O} -(t_{ok} - out_{ok}) \cdot out_{ok}(1 - out_{ok}) \cdot w_{jk}^{(2)} \cdot out_{hj}(1 - out_{hj}) \cdot x_i. \tag{82}$$

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

$$w_{ij}^{(1)} \rightarrow w_{ij}^{(1)} - \eta \cdot \sum_{k=1}^{O} \delta_{ok} \cdot w_{jk}^{(2)} \cdot out_{hj}(1 - out_{hj}) \cdot x_i \tag{83}$$

where we recall δ_{ok} as

$$\delta_{ok} = -(t_{ok} - out_{ok}) \cdot out_{ok}(1 - out_{ok}).$$