FYS-STK4155 - Applied data analysis and machine learning

Project 2

Even M. Nordhagen

November 12, 2018

- Github repository containing programs and results:

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

Abstract

In this project we first determine the energy in the one-dimensional Ising model using linear regression and deep learning. We will investigate the performance of both Ordinary Least Square (OLS), Ridge regression and Lasso regression, before we turn to a deep feed-forward neural network (FNN) with various activation function and hyper parameters. Secondly, we classify a two-dimensional Ising model using logistic regression and deep learning, and vary the regularization parameter.

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}), 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

The Ising model is the simplest theoretical description of a ferro magnet, where the spins can either point up or down. Even though the phase transition in the two dimensional Ising model was known in the early part of the twentieth century, the phase transition was first described analytically by the Norwegian-American physicist Lars Onsager in 1944. He found the critical temperature to be at $T_c = 2.269$ in units of energy.

Given a spin lattice, one can in principle calculate the energy using the Ising energy formula

$$E = \sum_{<ij>} J_{i,j} s_i s_j, \tag{1}$$

but this formula scales as N^D where N is the number of spins and D is the number of spatial dimensions. Would it not be wonderful if we could train a model once, and then calculate the energy of all Ising configurations correctly? What is even worse, is determining the phase of the spin lattice, where lattices below the critical temperature are ordered and lattices above the critical temperature are disordered. It might be possible to train a model recognizing the phase as well?

In this project we investigate different regression and classification approaches, where we first use linear regression to estimate the energy of a spin lattice including Ordinary Least Square (OLS), Ridge and Lasso regression. Thereafter, we see if we can repeat the exercise using a Feed-Forward Neural Network (FNN), where we focus on how the performance is dependent on the hyper parameters and the activation functions. The results were evaluated using Mean Square Error (MSE) and the R^2-score function, and we also used K-fold cross-validation to obtain more precise error estimations.

Furthermore, the classification problem is approached by logistic regression and again FNN. A regularization inspired by Ridge regression was added, giving us another hyper parameter to optimize. In classification, the output is binary, and we turn to the accuracy score to evaluate the results. Both Gradient Descent (GD) and Stochastic Gradient Descent (SGD) were used to minimize the cost function for both the regression and classification problems.

The background theory, including linear regression, logistic regression, neural network and Ising model, is described in the *Theory* section. In the *Methods* section, resampling techniques, minimization algorithms and methods for error estimation are detailed, and the code in described in section 4, *Code*. All the results are collected in the respective section, which are discussed in the *Discussion* section. A brief conclusion is given in *Conclusion.*

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

$$y_i = \sum_j X_{ij}\beta_j. \tag{2}$$

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 ($x_{ij} = s_i s_j$). The β_j's are the variable coefficients.

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

$$c(\vec{\beta}) = \sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\right)^2 \qquad \text{OLS} \tag{3}$$

which is minimized when

$$\vec{\beta} = (\hat{X}^T\hat{X})^{-1}\hat{X}^T\vec{y}. \tag{4}$$

Similarly, the Ridge cost function is

$$c(\vec{\beta}) = \sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\right)^2 + \lambda\sum_{j=1}^{p}\beta_j^2 \qquad \text{Ridge} \tag{5}$$

where λ is called the penalty. This is minimized when

$$\vec{\beta} = (\hat{X}^T\hat{X} + \lambda I)^{-1}\hat{X}^T\vec{y}, \tag{6}$$

and finally the Lasso cost function is given by

$$c(\vec{\beta}) = \sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\right)^2 + \lambda\sum_{j=1}^{p}\beta_j \qquad \text{Lasso.} \tag{7}$$

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 sets 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 classify every *linear classification problem.* A drawing of the perceptron is found in figure (1).

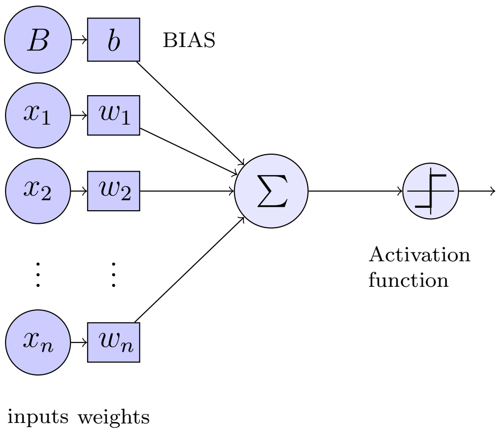


Figure 1: Logistic regression model with n inputs.

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 (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.

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

$$net = \sum_{i=1}^{I} x_i \cdot w_i + b \cdot 1. \tag{8}$$

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,

$$out = f(net). \tag{9}$$

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]:

$$c(\boldsymbol{W}) = -\sum_{i=1}^{n} \Big[y_i \log f(\boldsymbol{x}_i^T \boldsymbol{W}) + (1 - y_i) \log[1 - f(\boldsymbol{x}_i^T \boldsymbol{W})] \Big] \tag{10}$$

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

$$p(y|x) = \hat{y}^y \cdot (1 - \hat{y})^{1-y}. \tag{11}$$

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

$$\log \Big[p(y|x) \Big] = \log \Big[\hat{y}^y \cdot (1 - \hat{y})^{1-y} \Big] \tag{12}$$

$$= y \log \hat{y} + (1 - y) \log(1 - \hat{y}) \tag{13}$$

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]

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

$$c(\boldsymbol{W})^+ = c(\boldsymbol{W}) + \lambda \boldsymbol{W}. \tag{14}$$

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

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

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{15}$$

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

$$\frac{df(x)}{dx} = x(1-x). \tag{16}$$

$tanh(x)$ 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,

$$\boldsymbol{w}^{+} = \boldsymbol{w} - \eta \cdot \frac{\partial c(\boldsymbol{w})}{\partial \boldsymbol{w}}, \tag{17}$$

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

$$\frac{\partial c(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{\partial c(\boldsymbol{w})}{\partial out} \cdot \frac{\partial out}{\partial net} \cdot \frac{\partial net}{\partial \boldsymbol{w}} \tag{18}$$

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

$$\frac{\partial c(\boldsymbol{w})}{\partial out} = -\frac{y}{out} + \frac{1-y}{1-out}. \tag{19}$$

Further, the second derivative is the derivative of the activation function with respect to the output, which is given in (16)

$$\frac{\partial out}{\partial net} = out(1-out). \tag{20}$$

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

$$\frac{\partial net}{\partial \boldsymbol{w}} = \boldsymbol{x}. \tag{21}$$

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

$$\frac{\partial c(\boldsymbol{w})}{\partial \boldsymbol{w}} = [f(\boldsymbol{x}^T\boldsymbol{w}) - \boldsymbol{y}]\boldsymbol{x} \tag{22}$$

and obtain a weight update algorithm

$$\boxed{\boldsymbol{w}^+ = \boldsymbol{w} - \eta \cdot [f(\boldsymbol{x}^T\boldsymbol{w}) - \boldsymbol{y}]^T\boldsymbol{x}.} \tag{23}$$

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.

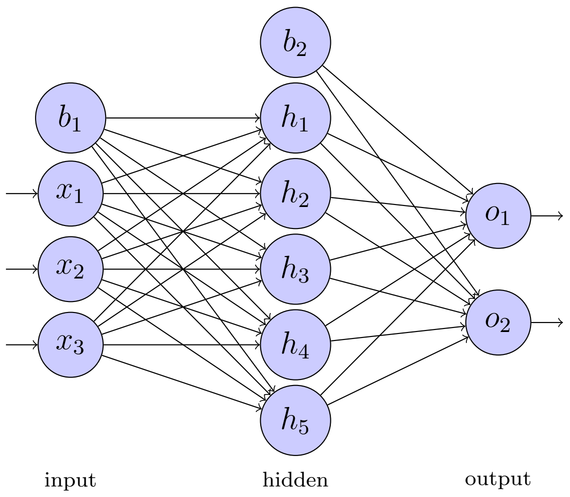


Figure 2: Neural network with 3 input nodes, 5 hidden nodes and 2 output nodes, in addition to the bias nodes.

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

$$net_{h,j} = \sum_{i=1}^{I} x_i \cdot w_{ij}^{(1)} = \boldsymbol{x}^T \boldsymbol{W}^{(1)} \tag{24}$$

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

$$h_j = f(net_{h,j}). \tag{25}$$

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

$$net_{o,j} = \sum_{i=1}^{H} h_i \cdot w_{ij}^{(2)} = \boldsymbol{h}^T \boldsymbol{W}^{(2)} \tag{26}$$

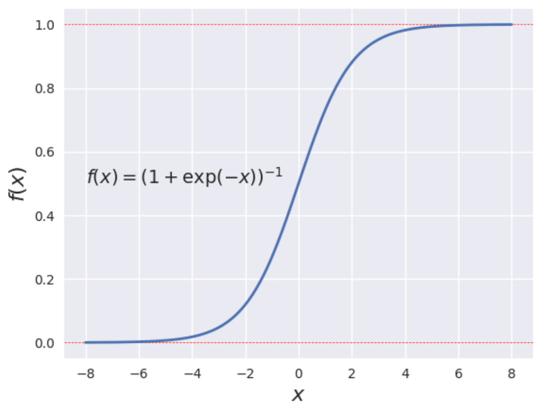
We can easily generalize this. Looking at the net output to a hidden layer l, we get

$$net_{h_l,j} = \boldsymbol{h^{(l-1)}}^T \boldsymbol{W}^{(l)}. \tag{27}$$

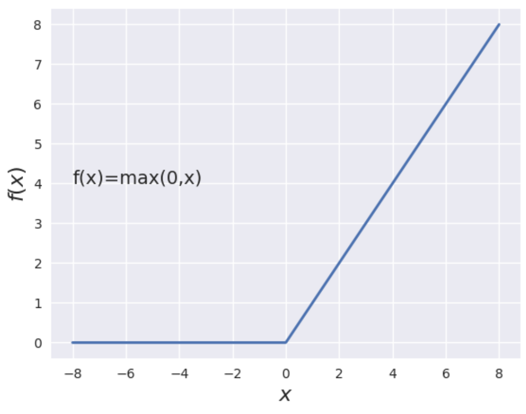
2.3.2 Activation function

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. Often, especially in the output layer, a straight linear function is used as well.

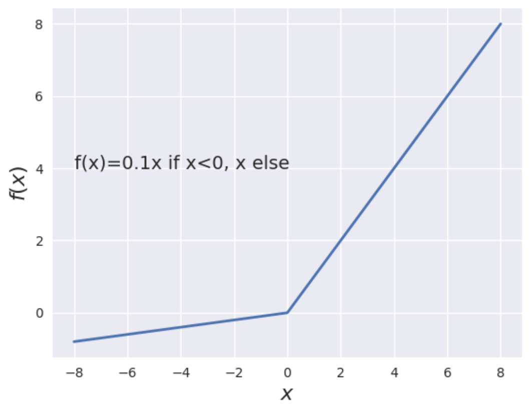
In figure (3), *standard RELU, leaky RELU* and *ELU* are plotted along with the logistic function.



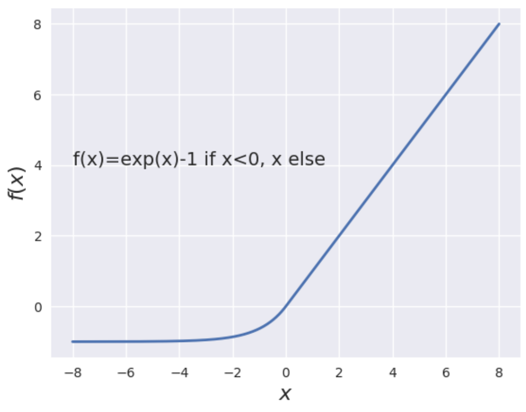
(a) logistic



(b) ReLU



(c) Leaky ReLU



(d) ELU

Figure 3: Some more or less popular activation functions

2.3.3 Backward Propagation

Backward propagation is probably the most used technique for updating the weights, and is actually again based on equation (17). 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:

$$w_{ij}^{(2)+} = w_{ij}^{(2)} - \eta \cdot [f(h_i^T w_{ij}) - y_j]^T h_i. \tag{28}$$

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

$$w_{ij}^{(1)+} = w_{ij}^{(1)} - \eta \cdot \sum_{k=1}^{O} \delta_{ok} \cdot w_{jk}^{(2)} \cdot out_{hj}(1 - out_{hj}) \cdot x_i \tag{29}$$

where we recall δ_{ok} as

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

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

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. The forward phase in a three-layer perceptron is

$$\begin{aligned}
net_{hi} &= \sum_j w_{ji}^{(1)} \cdot x_j \\
out_{hi} &= \mathrm{f}(net_{hi}) \\
\\
net_{ki} &= \sum_j w_{ji}^{(2)} \cdot out_{hj} \\
out_{ki} &= \mathrm{f}(net_{ki}) \\
\\
net_{oi} &= \sum_j w_{ji}^{(3)} \cdot out_{kj} \\
out_{oi} &= \mathrm{f}(net_{oi})
\end{aligned} \tag{30}$$

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

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

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.., and is a binary model with interactions based on the binary values. For instance, it can be used as a simple ferro magnetic model.

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

$$E_{i,i+1} = J_{i,i+1} s_i s_{i+1} \tag{31}$$

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

$$E = \sum_i J_{i,i+1} s_i s_{i+1}. \tag{32}$$

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

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

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,

$$E = \boldsymbol{x}^T \boldsymbol{J}. \tag{34}$$

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

$$\boldsymbol{E} = \boldsymbol{X}^T \boldsymbol{J}. \tag{35}$$

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

$$E = \sum_{<ij>} J_{i,j} s_i s_j \tag{36}$$

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

$$T_c = \frac{2J}{k \ln(1+\sqrt{2})} \approx 2.269 \tag{37}$$

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, which is often is a very expensive calculation. There are various methods for this, including the K-fold cross-validation, Bootstrap and Blocking methods. In this particular project, we will focus on the K-fold cross-validation.

3.1.1 K-fold validation method

K-fold validation is a method which 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 train our model 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. Then calculate the MSE and the R^2-score of the training and test set.
- Use the $K-2$ first folds plus the K-th fold as training data, and leave the $(K-1)$'th fold for validation. Calculate the MSE and R^2-score of training and test set
- Continue until all folds are used as training data
- Return the average training and test MSE, and the average training and test R^2-score. Typically one is interested in the average test errors.

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

```
def k_fold(X, E, T, h, eta, K):
        '''K-fold validation resampling based on neural netowrk'''

        MSE_train = 0
        MSE_test = 0
        R2_train = 0
        R2_test = 0

        Xmat = np.reshape(X, (K, int(len(X)/K), len(X[0])))
        Emat = np.reshape(E, (K, int(len(X)/K)))
        for i in range(K):
                Xnew = np.delete(Xmat, i, 0)
                Enew = np.delete(Emat, i, 0)

                X_train = np.reshape(Xnew, (len(Xnew)*len(Enew[0]), len(X[0])))
                E_train = np.reshape(Enew, (len(Xnew)*len(Enew[0])))

                obj = nn.NeuralNetwork(X_train, E_train, T, h, eta)
                W = obj.solver()
                E_train_tilde = obj.recall(X_train)
                E_test_tilde = obj.recall(Xmat[i])

        MSE_train += MSE(E_train_tilde, E_train)
        MSE_test += MSE(E_test_tilde, Emat[i])

        R2_train += R2(E_train_tilde, E_train)
        R2_test += R2(E_test_tilde, Emat[i])

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

3.2 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 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.

3.2.1 Gradient Descent

The Gradient Descent method was mentioned already in the theory part, in equation (17), 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

$$W_{ij}^{+} = W_{ij} - \eta \cdot \frac{\partial c(\boldsymbol{W})}{\partial W_{ij}} \tag{38}$$

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
```

3.2.2 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

$$W_{ij}^{+} = W_{ij} - \frac{\eta}{N}\sum_{k=1}^{N}\frac{\partial c_k(\boldsymbol{W})}{\partial W_{ij}} \tag{39}$$

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
```

3.3 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.

3.3.1 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.

$$\mathrm{MSE}(\vec{y}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - t_i)^2 \tag{40}$$

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

$$R^2(\vec{y}, \tilde{\vec{y}}) = 1 - \frac{\sum_{i=1}^{N}(y_i - t_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}. \tag{41}$$

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,

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(y_i = t_i)}{n}. \tag{42}$$

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

In order to reuse the code where possible, we ended up with several functions that communicate in criss-cross. The main functions are called *find_energy.py* and *classifier.py*, where the former estimates the energy of a Ising lattice using linear regression and neural network, getting the data from *Ising_1D.py*. The latter estimates the phase of the Ising model using logistic regression and neural networks, getting the data from *Ising_2D.py* (taken from [3]). *error_tools.py* consists of functions that returns the error, including MSE, R2 and Accuracy, which communicates with *resampling.py* as well. Both main functions are largely dependent on *neural_network.py*, which gets the optimization functions from *optimization.py* and the activation functions from *activation.py*.

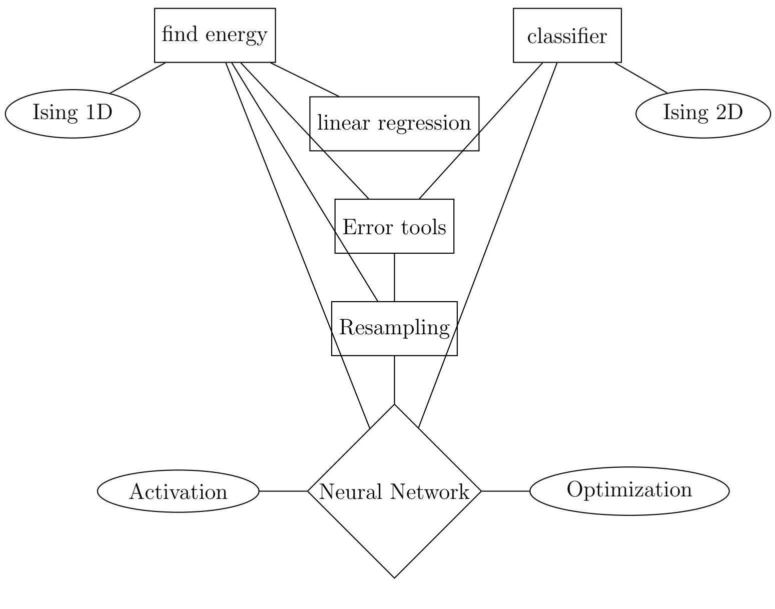


Figure 4: Code structure

4.2 Run the code

The neural network was implemented generally, and should be able to handle an arbitrary number of hidden layers and optional hyper parameters and activation functions. However, a few assumptions were made to maintain some neatness, for instance we assume that all hidden layers are affected by the same activation function, and the cost function is fixed apart from the regularization parameter. By default, the neural network class takes a set of input arrays X, corresponding targets t, number of iterations T, number of hidden nodes h, which is either an integer or a list of integer if one want multiple hidden layers. No hidden layer by default. *eta* is the learning rate and *lamb* is the regularization parameter, both set to 1e-4, and $f1$ and $f2$ are the output activation function and the activation function on hidden layers respectively. They are optional functions, and in *activation.py* the *pure linear*, *logistic*, *tanh*, *ReLU*, *Leaky ReLU* and *ELU* functions are implemented. Finally, *opt* is the optimizer, which is GD by default, but also SGD is implemented in *optimization.py*.

```
NeuralNetwork(X, t, T, h=0, eta=1e-4, lamb=1e-4, f1=none, f2=none, opt=GD)
```

By calling the solver, the network will minimize the weights with respect to the cost function and return the weights. The weight matrix contains the bias weights as well, which saves us from a few lines of code. To recall outputs, the recall function is called given an arbitrary input with the same shape as the input which was used in training. A general call could look like

```
import numpy as np
import neural_network as nn
from activation import *
from optimization import *

# Data set
X_train = np.array([...])          # Inputs
t_train = np.array([...])          # Targets
X_test = np.array([...])           # Inputs
t_test = np.array([...])           # Targets

# Parameters
T = 10           # Number of iterations
h = [10,10]      # Number of hidden nodes
eta = 1e-4       # Learning rate
lamb = 1e-4      # Regularization parameter

# Activation functions
f1 = logistic         # Activation on output layer
f2 = Leaky_ReLU       # Activation on hidden layer(s)
opt = SGD             # Minimization function

obj = nn.NeuralNetwork(X_train, t_train, T, h, eta, lamb, f1, f2, opt)
obj.solver()                                          # Obtain optimal weights
y_train = obj.recall(X_train)                         # Recall training phase
y_test = obj.recall(X_test)                           # Recall test phase
Error = error_estimator(t_test, y_test)               # Error estimation
```

4.3 Implementation

The algorithm used is based on the theory in section 2.3, where we vectorize to maximize the computational speed. For the forward phase, the implementation looks like

```
'''Feed forward'''
out = [np.insert(X, 0, 1)]                          # Add bias
for i in range(H):                                      # Loop over hidden layers
        net = np.dot(out[-1], W[i])                             # Net output to layer i
        Out = f2(net)                                   # Output to layer i
        out.append(np.insert(Out, 0, 1))                # Add bias
net = np.dot(out[-1], W[-1])                            # Final output
out.append(f1(net))                                     # Final output with f1
```

where all the "self"'s from the object orientation are removed due to legibility. Similarly, the backward propagation can be implemented like this

```
'''Backward propagation'''
deltah = [(out[-1] - t) * df1(out[-1])]
for j in range(H):
        delta = W[H-j].dot((deltah[-1]).T) * df2(self.out[H-j])
        deltah.append(delta[:-1])
deltah=deltah[::-1]
```

5 Results

In this section all the results are collected, where we first determine the energy in the one-dimensional Ising model, and the find the phase. We use linear regression and neural networks, and logistic regression and neural networks respectively for the two cases.

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. [1] Let's start with linear regression.

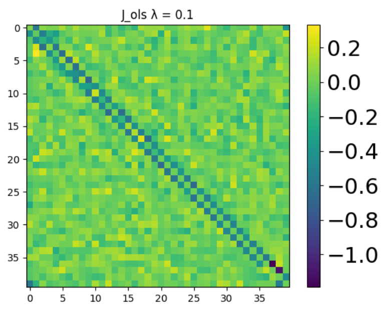
5.1.1 Linear regression

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).

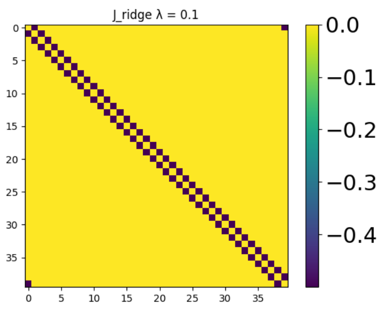
Table 1: Mean Square Error and R^2-score presented for OLS, Ridge and Lasso on the Ising model. 'Train' means the results obtained from the training set, containing 8000 states, and 'Test' means the results obtained from the test set, containing 2000 states. The 'K-fold'-columns are results from K-fold 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

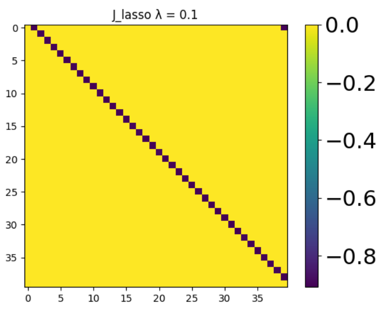
We observe that the MSE is quite low, especially for Ridge and Lasso. The R^2-score is correspondingly high. Below, in figure (5), the $\boldsymbol{J}$-matrix is visualized for all the linear regression methods and $\lambda \in [1e - 1, 1e - 6]$.



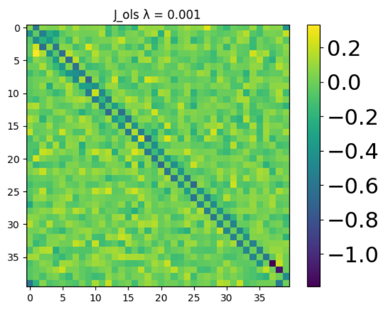
(a) OLS, $\lambda = 0.1$



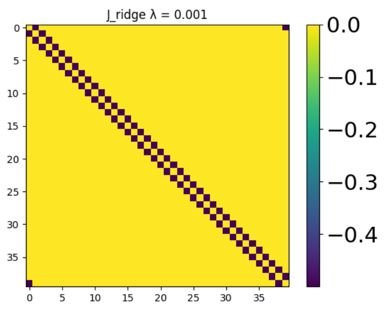
(b) Ridge, $\lambda = 0.1$



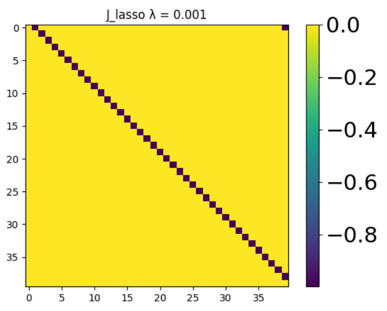
(c) Lasso, $\lambda = 0.1$



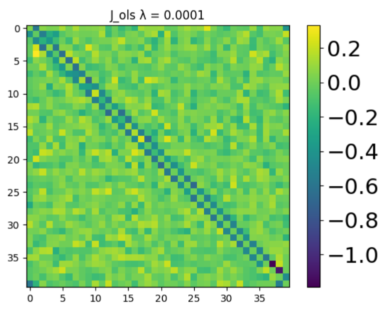
(d) OLS, $\lambda = 0.001$



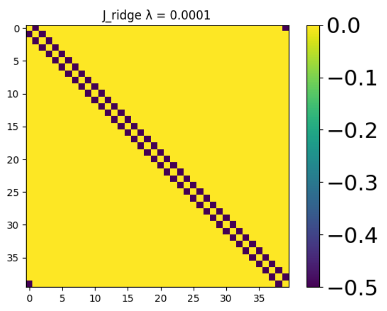
(e) Ridge, $\lambda = 0.001$



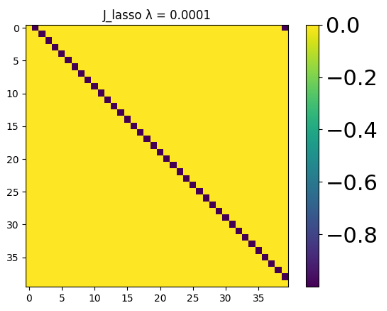
(f) Lasso, $\lambda = 0.001$



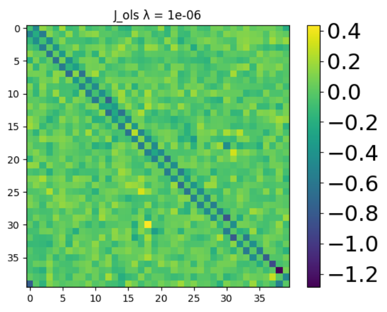
(g) OLS, $\lambda = 0.0001$



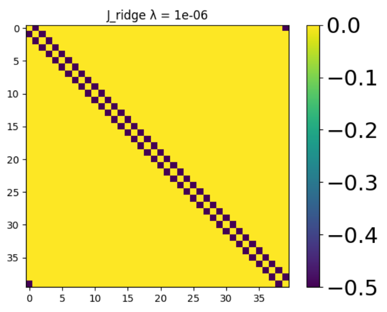
(h) Ridge, $\lambda = 0.0001$



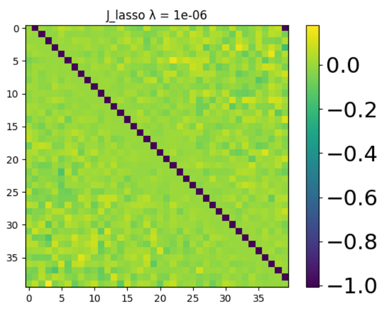
(i) Lasso, $\lambda = 0.0001$



(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).

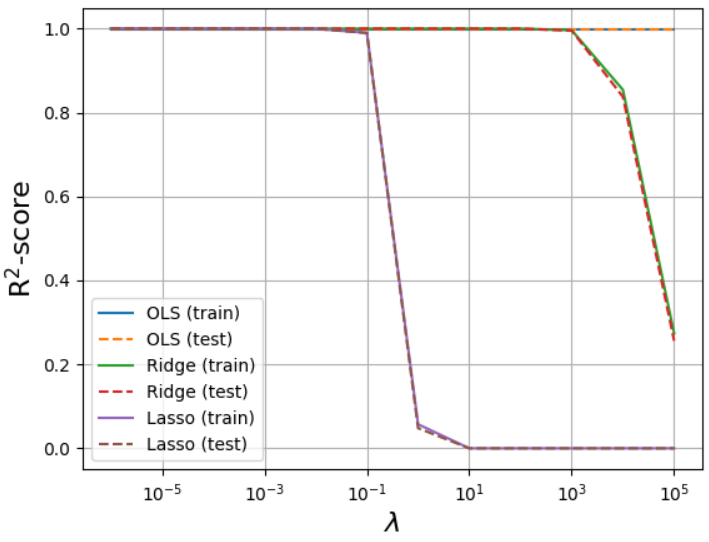


Figure 6: The R^2-score as a function of the penalty.

5.1.2 Neural networks

We now want to train a neural network on different Ising states, with the energies as the targets. In table (2), the MSE and R^2-score are given for various activation functions in the hidden layer and various number of hidden nodes for the training set, test set and using K-fold cross validation. By far the lowest MSE is obtained when using the pure linear cost function in the hidden layer, and apparently the more nodes the lower MSE. This does not apply for the other activation functions.

Table 2: Mean Square Error and R^2-score obtained using a neural network on the training set, test set and from K-fold cross validation with $K = 10$. Various activation functions *Pure linear*, *ReLU*, *Leaky ReLU* and *ELU* were used in the hidden layer, and the number of hidden nodes, H, was set to 0, 5 and 10. The learning rate was set to $\eta = 0.0001$ and we used $T = 5$ minimization iteration in GD.

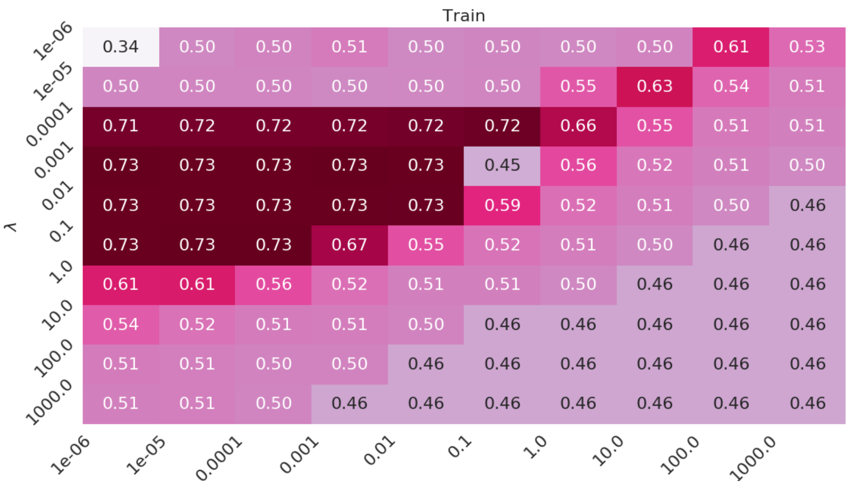
		MSE			**R2**		
	H	Train	Test	K-fold	Train	Test	K-fold
Linear	0	3.244e-4	5.573e-4	1.206e-4	1.000	1.000	1.000
	5	1.545e-9	2.690e-9	2.375e-3	1.000	1.000	1.000
	10	6.905e-11	7.848e-11	3.932e-10	1.000	1.000	1.000
ReLU	0	3.196e-4	5.511e-4	1.188e-4	1.000	1.000	1.000
	5	8.618e-1	1.107e-0	2.405e-1	0.9779	0.9712	0.9939
	10	1.600e-1	1.797e-1	1.994e-1	0.9959	0.9953	0.9943
Leaky	0	3.182e-4	5.481e-4	1.174e-4	1.000	1.000	1.000
	5	4.128e-1	4.627e-1	2.230e-1	0.9894	0.9880	0.9943
	10	1.965e-1	2.115e-1	2.394e-1	0.9949	0.9945	0.9939
ELU	0	3.449e-4	6.472e-4	1.198e-4	1.000	1.000	1.000
	5	2.414e-1	2.546e-1	1.463e-1	0.9405	0.9342	0.9535
	10	1.223e-1	2.281e-1	5.359e-1	0.9970	0.9941	0.9862

5.2 Classifying the phase

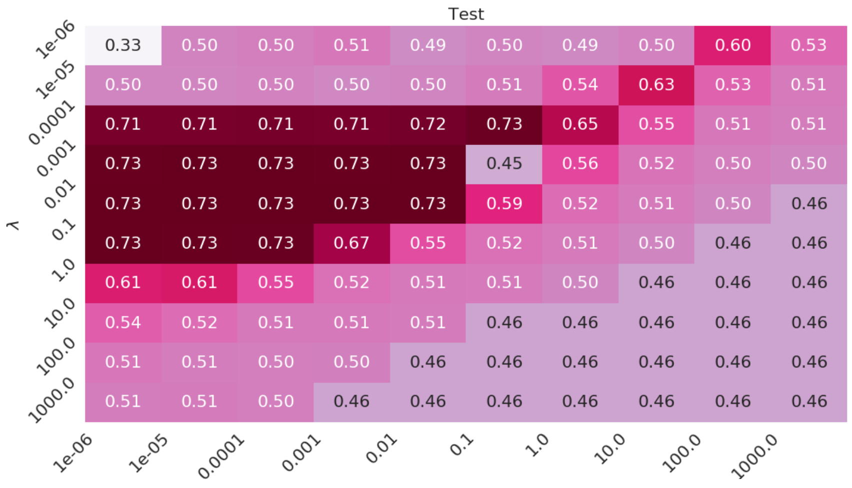
We now turn to the classification problem, where we want to find whether a Ising lattice is ordered or disordered. We exclude Ising lattices around critical temperature in our training set ("Train"), and test our model on unseen data ("Test"). Additionally, we also test our model on lattices which are close to the critical temperature ("Critical"). We first use logistic regression with logistic activation, and thereafter turn to neural network with various activation functions on the hidden layer and logistic activation on the output.

5.2.1 Logistic regression

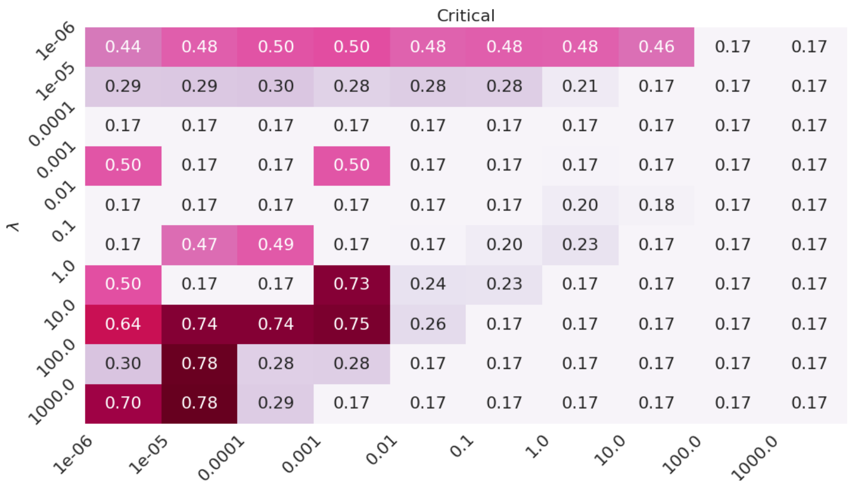
Our first approach is the logistic regression. For this, we calculate the accuracy for various learning rates and regularizations, given in figure (7) for all three data sets. What we observe, is that the model is really sensitive and the accuracy vary a lot. For some specific learning rates and regularization parameters, we get good result even for the critical set.



(a) Train



(b) Test



(c) Critical

Figure 7: The accuracy score as function of the learning rate $\eta \in [1e-6, 1e3]$ and regularization parameter $\lambda \in [1e-6, 1e3]$ for the training set (a), test set (b) and critical set (c).

5.2.2 Neural networks

Our second and final approach is classification using neural network. In the same way as when we used neural network in regression, we will investigate various activation functions in the hidden layer(s).

Table 3: 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 $\lambda = 1e - 3$.

		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

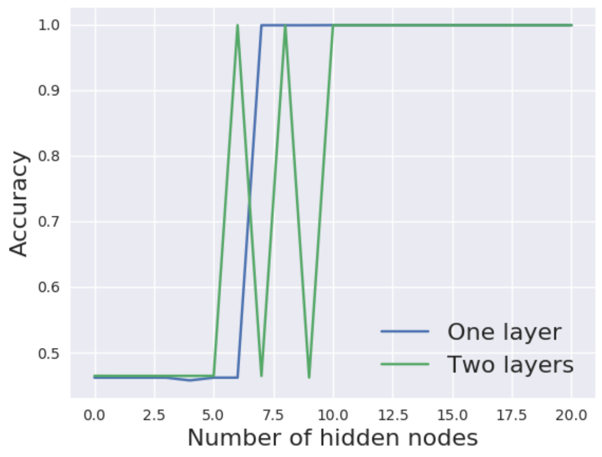


Figure 8: Accuracy as a function of number of hidden nodes, for both one and two layers. X-axis indicates number of hidden nodes in each layer, where the two-layer perceptron has an equal number of nodes in each layer.

6 Discussion

In table (1), one can see that the obtained results are quite satisfying, especially for Ridge and Lasso. However, I obviously do not get the same results as for Metha et. al. [3], who over all got a small MSE for the training set, but could not keep it up for the test set using OLS and Ridge. I have checked if the train and test sets are mixed up somewhere, but could not find the error.

Further, we observe that only Lasso is able to obtain the correct J-matrix, which is because it does not conserve symmetries. For OLS and Ridge, we get sub-superdiagonal matrices, with values 0.5 instead of 1.0. The result is an interaction 0.5 between particle i and j, and then an interaction 0.5 between j and i, which should give the same as just an interaction 1.0 between i and j since we apply periodic boundary conditions. In figure (6), we can see that the lower penalty the better, which we expect to be valid until the models approach OLS.

When applying the neural network, linear cost function on the hidden nodes gives the lowest MSE because the function is linear below x=0 as well. The more nodes we get, the lower MSE we get for this function. The remaining functions are quite consistent, probably because they have similar form (low but nonzero derivative below zero), but they give best results without any hidden layer, which is linear regression. We also tried multiple layers, but apparently a model with a hidden layer is complex enough.

For classification, the logistic regression did not work well, with a maximum accuracy of 0.73. The spin configurations are not linear with the phase, such that a model without a hidden layer is not complex enough. In figure (7), we also see that the model is very sensitive, and the MSE is jumping up and down as we vary the regularization and learning rate. An optimal learning rate is not too small, neither too large, which is as expected since we often will walk past the minimum if the learning rate is too large, and it takes time to reach the minimum if the learning rate is small. We also observe that we get relatively good accuracy for some specific values of λ and η for the critical set, something I believe happen by accident.

Unlike when we applied a neural network on regression, using a pure linear activation function on the hidden layer(s) causes a poor performance. On the other hand, the ReLU and leaky ReLU activation functions give surprisingly good results with accuracy above 99%! The reason could be that the the derivative below zero is much lower for the latter functions, which gives a faster training. [6] Further, we observe that the accuracy slightly increase when adding more layers, an effect that probably could be recreated by more hidden nodes in one layer according to the universal approximation theorem. An accuracy of 99.9% for the critical set is also higher than expected, and shows that there is a distinctive difference between the ordered and disordered lattices also close to the critical temperature.

In the end, I added a plot which shows how the R^2-score depends on the number of hidden nodes. Apparently, the accuracy score is stable above a certain number of nodes for both one and two hidden layers, so perhaps it does not matter how many hidden nodes we use as long as we are above this unstable area.

7 Conclusion

Machine learning is a difficult field, with unlimited of variational options. The conclusion...

For the energy recognition, both linear regression and neural networks give great results, which we would expect since the energy is linear with respect to the spin configurations. What is more surprising is how good accuracy we are able to get for the classification problem, in particular when using neural networks. We obtained an accuracy above 99% for the test cause with only one minimization iteration, taking just a few seconds in a high-level language.

Even though the results are satisfying in this project, we are not 100% satisfied before we can guarantee obtaining the correct phase. I believe that other activation functions together with more complex optimization techniques can increase the test accuracy to 100%. If I had time, I would have like to implement a momentum method or a conjugate gradient method to see if we were able to reach the minimum faster.

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] 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

A Appendix A - Derivation of activation functions

A.1 Sigmoid

If we differentiate the sigmoid function

$$f(x) = \frac{1}{1 + \exp(-x)} \tag{43}$$

from equation (15), and we will get

$$f'(x) = f(1 - f) \tag{44}$$

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

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

$$\frac{\partial E_{TOT}}{\partial w_{ij}} = \frac{\partial E_{TOT}}{\partial out_j} \cdot \frac{\partial out_j}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ij}}. \tag{45}$$

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

$$\frac{\partial E_{TOT}}{\partial out_j} = -(t_j - out_j) \tag{46}$$

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

$$out_j = \frac{1}{1 + \exp(-net_j)}. \tag{47}$$

and we have already found that

$$\frac{\partial out_j}{\partial net_j} = out_j(1 - out_j). \tag{48}$$

Furthermore the netto output is given by equation (8),

$$net_j = \sum_{i=1}^{I} X_i \cdot w_{ij} + b_j \cdot 1. \tag{49}$$

We end up with

$$\frac{\partial E_{TOT}}{\partial w_{ij}} = -(t_j - out_j) \cdot out_j(1 - out_j) \cdot X_i \tag{50}$$

and the weight updating formula

$$\boxed{w_{ij}^{+} = w_{ij} + \eta \cdot (t_j - out_j) \cdot out_j(1 - out_j) \cdot X_i} \tag{51}$$

Similarly we need to update the bias weights, with a formula similar to that one in equation (8):

$$b_i^{+} = b_i - \eta \cdot \frac{\partial E_{TOT}}{\partial b_i}. \tag{52}$$

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

$$b_i^+ = b_i + \eta \cdot (t_i - out_i) \cdot out_i(1 - out_i) \tag{53}$$

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 (17), but we neglect the x_i:

$$w_{ij}^\dagger = w_{ij} - \eta \cdot \frac{\partial E_{TOT}}{\partial w_{ij}}. \tag{54}$$

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).

$$\frac{\partial E_{TOT}}{\partial w_{ij}} = \frac{\partial E_{TOT}}{\partial out_{oj}} \cdot \frac{\partial out_{oj}}{\partial net_{oj}} \cdot \frac{\partial net_{oj}}{\partial w_{ij}} \tag{55}$$

$$= \delta_{oj} \cdot \frac{\partial net_{oj}}{\partial w_{ij}} \tag{56}$$

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 (10), and recognize that

$$\frac{\partial E_{TOT}}{\partial out_{oj}} = -(t_{oj} - out_{oj}) \tag{57}$$

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

$$out_{oj} = \frac{1}{1 + \exp(-net_{oj})} \tag{58}$$

such that

$$\frac{\partial out_{oj}}{\partial net_{oj}} = out_{oj}(1 - out_{oj}) \tag{59}$$

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

$$\delta_{oj} = -(t_{oj} - out_{oj}) \cdot out_{oj}(1 - out_{oj}) = \frac{\partial E_{TOT}}{\partial net_{oj}}. \tag{60}$$

We also have that

$$net_{oj} = \sum_k w_{kj} \cdot out_{hk} + b_{2j} \cdot 1 \tag{61}$$

such that

$$\frac{\partial net_{oj}}{\partial w_{ij}} = out_{hi} \tag{62}$$

and

$$\frac{\partial E_{TOT}}{\partial w_{ij}} = \delta_{oj} \cdot out_{hi}. \tag{63}$$

The specific updating algorithm is the following

$$\boxed{w_{ij}^{(2)} \rightarrow w_{ij}^{(2)} - \eta \cdot \delta_{oj} \cdot out_{hi}} \tag{64}$$

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.

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

$$\frac{\partial E_{TOT}}{\partial w_{ij}} = \frac{\partial E_{TOT}}{\partial out_{hj}} \cdot \frac{\partial out_{hj}}{\partial net_{hj}} \cdot \frac{\partial net_{hj}}{\partial w_{ij}}. \tag{65}$$

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

$$E_{TOT} = E_{o1} + E_{o2} + \ldots + E_{oO}. \tag{66}$$

This causes

$$\frac{\partial E_{TOT}}{\partial out_{hj}} = \frac{\partial E_{o1}}{\partial out_{hj}} + \frac{\partial E_{o2}}{\partial out_{hj}} + \ldots + \frac{\partial E_{oO}}{\partial out_{hj}} \tag{67}$$

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

$$\frac{\partial E_{ok}}{\partial out_{hj}} = \frac{\partial E_{ok}}{\partial net_{ok}} \cdot \frac{\partial net_{ok}}{\partial out_{hj}}. \tag{68}$$

We could have done another change of variables above to avoid equation (69), but this is neater

$$\frac{\partial E_{ok}}{\partial net_{ok}} = \frac{\partial E_{ok}}{\partial out_{ok}} \cdot \frac{\partial out_{ok}}{\partial net_{ok}}. \tag{69}$$

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

$$\frac{\partial E_{ok}}{\partial out_{ok}} = -(t_{ok} - out_{ok}) \tag{70}$$

and

$$\frac{\partial out_{ok}}{\partial net_{ok}} = out_{ok}(1 - out_{ok}). \tag{71}$$

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

$$net_{ok} = \sum_i w_{ik}^{(2)} \cdot out_{hi} + b_{2k} \cdot 1 \tag{72}$$

and

$$\frac{\partial net_{ok}}{\partial out_{hj}} = w_{jk}^{(2)} \tag{73}$$

and we end up with

$$\frac{\partial E_{ok}}{\partial out_{hj}} = -(t_{ok} - out_{ok}) \cdot out_{ok}(1 - out_{ok}) \cdot w_{jk}^{(2)}. \tag{74}$$

That was the difficult part, as we have seen before

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

and since

$$net_{hj} = \sum_l w_{lj} \cdot x_l \tag{76}$$

where x_i is input i, we obtain

$$\frac{\partial net_{hj}}{\partial w_{ij}} = x_i \tag{77}$$

and we finally end up with

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

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

$$w_{ij}^{(1)} \to w_{ij}^{(1)} - \eta \cdot \sum_{k=1}^{O} \delta_{ok} \cdot w_{jk}^{(2)} \cdot out_{hj}(1 - out_{hj}) \cdot x_i \tag{79}$$

where we recall δ_{ok} as

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