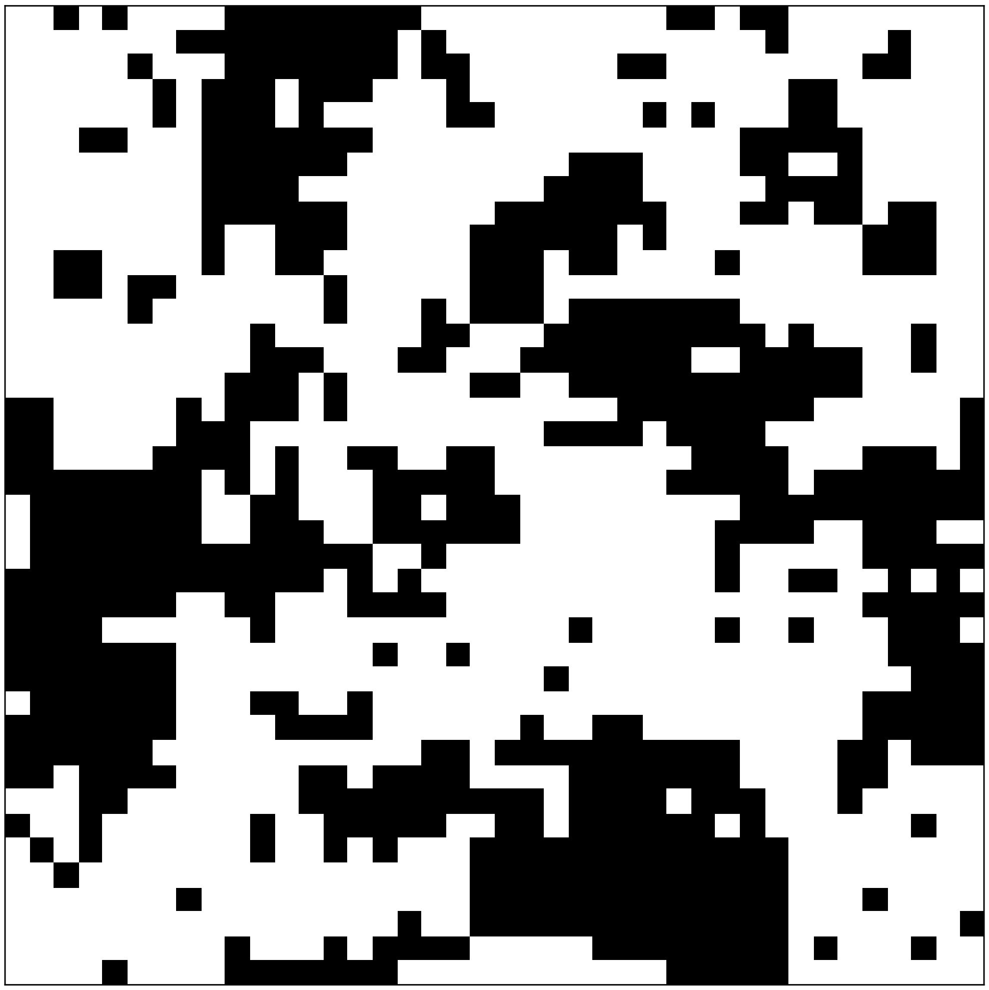
Classification and Regression - From linear and logistic regression to neural networks.

— Project 2 —

FYS-STK4155

Heine Aabø Halvard Sutterud

November 2018



Abstract

In this project we study the difference between standard regression methods such as ordinary least squares and logistic regression, and neural networks. We apply the methods to the Ising model in one and two dimensions. In the 1D case the energies of a given configuration is predicted, while in the latter the magnetic phase of the lattice is classified and predicted. One of the main results is that machine learning, despite all the hype, has its limitations. This is exemplified in the one-dimensional Ising problem, where Ordinary Least Squares (OLS) with and without regularization massively outperforms the neural network with an R2 score of 0.838 for 666 data points, while the optimal neural network gave an R2 score of 0.578, for 128 nodes and a learning rate of $\eta = 10^{-1}$, with the sigmoid activation function in the hidden layer. However, in the classification case, the neural network shows its strength by giving an accuracy of up to 99.4%, while our comparatively poorly performing logistics regression solver gives an accuracy of 68.8%. We consider the best hyperparameters for the classification network to be the RELU activation function with 8 hidden neurons and a learning rate of $\eta = 10^{-1/2}$, as this converged to the optimal value in only 4 epochs.

Contents

1 Introduction

Machine learning have been the source of extreme amounts of hype during the last years, with the solution of problems earlier deemed unsolvable by computers such as car navigation and playing Go. Underlying many of the methods which have nurtured this hype are advanced versions of the simple statistical algorithm we will demonstrate in this project; the multilayer perceptron model, also called a neural network. Even in its simple form, neural networks can outperform other methods when it comes to prediction, as we will show.

In this project we will study both classification and regression problems. We start simple with the regression problem, where we will be predicting the energies of the one-dimensional Ising model. This is first done using Ordinary least squares, lasso and ridge regression, and we will here reuse the code from project 1 [3]. Afterwards we use our newly implemented neural network to find the energies. The classification problem is to predict the phase of a lattice configuration of the 2D Ising model, which exhibits a phase transition at a critical temperature found analytically by Lars Onsager in 1944. We first implement logistic regression, solving the problem by minimizing a cross entropy cost function with gradient descent. This is then extended to a neural network problem, where gradient descent in combination with the backpropagation algorithm is used to find the global minima. In this case we show that the neural network can far outperform the logistic regression, using the relatively large data storage at hand.

We begin with an overview of the theory of the Ising model, logistic regression and neural networks, including some of the mathematical details. For a more in depth explanation and derivation of the backpropagation algorithms, Mehta et al [1] is an excellent read. We also cover the different methods of gradient descent used. Section 3 contains a short summary of our specific implementations, with references to the notebooks (part of supplementary materials) which reproduce the data of this article. Next we present the results in section 4, followed by a more comprehensive discussion in section 5 and concluding remarks.

2 Theory

2.1 Ising model

The Ising model is a matemathical model representing some features of a physical system. It depends on binary variables taking the values 1 and 0 or +1 of −1. In this project an interaction model for spins is considered. From quantum mechanics we know that particles can have intrinsic spin of two possible values, so the variables of the model are the spins of each particle in a lattice. The interactions between the particles is simplified to nearest neighbours only, with the same interaction strength for all pairs. The energy of the system can then be defined as

$$E = -J \sum_{<kl>}^{N} s_k s_l, \tag{1}$$

where J is the coupling constant representing the interaction strength between each spin pair, N is the total number of spins s, which can have values of ± 1.

From eq. (1) it is clear that the energy favour aligned spin pairs for $J > 0$, where the interactions are ferromagnetic, and misaligned spin pairs for $J < 0$, where the interactions are antiferromagnetic.

If the interaction strength is different for each spin pair, the energy becomes

$$E = -\sum_{j,k=1}^{N} J_{jk} s_j s_k \tag{2}$$

For the one dimensional Ising model no phase transition is exhibited. For the two dimensional model a phase transition occure at a critical temperature T_c, called the Curie temperature. For temperature below T_c the system is either ferromagnetic or antiferromagnetic, while above T_c it becomes paramagnetic. That is, the system of spins go from being ordered to disordered.

2.2 Logistic regression

If a data set only consist of discrete values $k = 0, 1, ..., K-1$, where K is the number of classes or different possible values, then prediction of new data should also be of these values. If we consider the simplest case with two classes, e.g. 0 and 1, a useful predictions would be the probability for it to be one of the two. The function evaluating this should then give values between 0 and 1. Usually the sigmoid function is used for this:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{3}$$

The regression model is typically $\hat{y} = \hat{X}\hat{\beta}$, and so the probability for some input, given by $\hat{X}_i\hat{\beta}$, to be of class 1 is then given as

$$p(y_i = 1|\hat{X}_i, \hat{\beta}) = \sigma(\hat{X}_i\hat{\beta}) \tag{4}$$

and the probability for it to be of class 0 is just

$$p(y_i = 0|\hat{X}_i, \hat{\beta}) = 1 - p(y_i = 1|\hat{X}_i, \hat{\beta}) \tag{5}$$

where the prediction is set to 1 if $p(y = 1|x) \geqslant 0.5$.

The total probability for all outcomes, where $y_i \in \{0, 1\}$, is then the product of all individual probabilities

$$P(\hat{y}|\hat{\beta}) = \prod_{i=1}^{n} [p(y_i = 1|\hat{X}_i, \hat{\beta})]^{y_i} [1 - p(y_i = 1|\hat{X}_i, \hat{\beta})]^{1-y_i} \tag{6}$$

so, the term containing the probability of it being the opposite value of y_i will cancel out. Utilizing the properties of logarithms give the log-likelihood

$$\mathcal{C}(\hat{\beta}) = \sum_{i=1}^{n} \Big(y_i \log p(y_i = 1|\hat{X}_i, \hat{\beta}) + (1 - y_i) \log \Big[1 - p(y_i = 1|\hat{X}_i, \hat{\beta})) \Big] \tag{7}$$

which can be rewritten and set to negative; this defines a cost function to minimize, called the cross-entropy:

$$\mathcal{C}(\hat{\beta}) = -\sum_{i=1}^{n} \Big(y_i(\hat{X}_i\hat{\beta}) - \log(1 + e^{\hat{X}_i\hat{\beta}}) \Big) \tag{8}$$

To avoid overfitting as a result of large parameters $\hat{\beta}$, a regularization term can be added, as with ridge and lasso regression, often called $L2$ and $L1$ regularization respectivly. With $L2$ regularization the cost function becomes

$$\mathcal{C}(\hat{\beta}, \lambda) = \mathcal{C}(\hat{\beta}) + \lambda \sum_{j=1}^{m} \beta_j^2 \tag{9}$$

The optimal set of parameters with respect to the cost function eq. (9), can be found using gradient descent. Here the first derivative of eq. (9) is needed:

$$\frac{\partial \mathcal{C}(\hat{\beta}, \lambda)}{\partial \beta_j} = \frac{\partial \mathcal{C}(\hat{\beta})}{\partial \beta_j} + 2\lambda \sum_{j=1}^{m} \beta_j \tag{10}$$

where

$$\begin{aligned} \frac{\partial \mathcal{C}(\hat{\beta})}{\partial \beta_j} &= -\sum_{i=1}^{n} \Big(y_i \hat{X}_i - p(y_i = 1|\hat{X}_i, \hat{\beta})\hat{X}_i \Big) \\ &= -\sum_{i=1}^{n} \Big(\hat{X}_i(y_i - p_i) \Big) \end{aligned} \tag{11}$$

since

$$\frac{\partial \log(1+e^{\hat{X}_i\hat{\beta}})}{\partial \beta_j} = \frac{\hat{X}_i e^{\hat{X}_i\hat{\beta}}}{1+e^{\hat{X}_i\hat{\beta}}} = p(y_i = 1|\hat{X}_i, \hat{\beta})\hat{X}_i \tag{12}$$

The performance of a logistic regression model can be measured with the accuracy score, given by

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n} \tag{13}$$

2.3 Gradient descent

Gradient descent is a optimization method for finding minima of a given function, where in machine learning problems this will be a cost function. If the derivative of the cost function with respect to the parameters β can be defined, then the direction of a minimum, local or global, will be where the derivative is negative. By iteratively taking steps in this direction a minimum is certain to be found if it exists. The process can be made faster by changing the step size, with several different approaches constituting different types of gradient descent. This can also make it easier to avoid local minima. The simpelest approach is to have a fixed step size, or learning rate η. More advanced approaches use adaptive learning rates depending on different properties of the gradient. For a fixed learning rate the iterative update of the parameters become:

$$\hat{\beta} \leftarrow \hat{\beta} - \eta \nabla \mathcal{C}(\hat{\beta}) \tag{14}$$

There are good reasons for using gradient descent instead of direct methods. For other problems in machine learning, like neural networks, the cost functions are often not convex and can have multiple local minima. In these cases a gradient descent with sufficient learning approach is an easy and reliable method for finding good weights.

2.3.1 Stochastic batch gradient descent

To prevent the algorithm from getting stuck in a local minima, we introduce some stochasticity in the training process. In stochastic gradient descent one estimates the derivative of the cost function by sampling a randomly sampled point, and then updates the weights correspondingly. This can then be further modified by updating the weights with the average over the derivative sampled at a subset of points, or minibatches. The expression for the updated parameters or weights is then

$$\hat{\beta} \leftarrow \hat{\beta} - \gamma \sum_{i \in B_k} \nabla \mathcal{C}(\hat{\beta})|_{\mathcal{L}_i}, \tag{15}$$

where $\mathcal{L}_i$ is the i-th data value part of the minibatch denoted B_k.

2.4 Neural network

In this project we will make use of the simplest type of artificial neural network, called a multilayer perceptron. This network consists of a series of fully connected layers, meaning that each node in one layer has incoming and outgoing connections to all the nodes in the previous and next layers respectively.

2.4.1 Perceptron

The original perceptron as invented by Frank Rosenblatt in 1957 outputs a single binary value from a set of binary input values, and so makes up a binary classifier. Along with each input x_i comes a weight ω_i that determines the importance of the input to the output. The output is decided by whether the sum $\sum_i^n \omega_i x_i$ is above or below some threshold value. The activation of the binary perceptron can then written as

$$\text{output} = \begin{cases} 1, & \text{if } \omega \cdot x + b > 0 \\ 0, & \text{otherwise,} \end{cases} \tag{16}$$

where we have added an intercept b called the bias. This bias is important, as it lets the breaks the proportionality of input of the perceptron and its output. In our case, we will consider more complicated perceptrons, where the activation is defined by a function other than the step function. Again the sigmoid function is a popular candidate, but in recent years the rectified linear unit, or RELU, has become more and more used. It is defined by

$$\text{RELU}(x) = \begin{cases} x, & x > 0 \\ 0, & \text{otherwise.} \end{cases} \tag{17}$$

The RELU has the advantage that its derivative does not saturate for large inputs. In addition to these two activation functions we will also use tangens hyperbolicus, or tanh, as activation.

2.4.2 Multilayer perceptron and feedforward networks

As mentioned in Nielsen's book on neural networks [2] the simple perceptron can replicate a `NAND` gate, and since all logic systems can be represented as `NAND` gates a system of perceptrons will do the same. This motivates the use of what is called a multilayer perceptron. Here the system of perceptrons is divided into layers, one input and output layer and an arbitrary number of hidden layers. Where the inputs to a perceptron in a hidden layer is the output of all the perceptrons in the previous layer. Each perceptron is then assigned an activation function $f(z)$ that takes the input $z = \omega \cdot x + b$. One key aspect of the neural network is that this activation function needs to be nonlinear, as subsequent linear transformations can be reduced to a single linear transformation. By having a nonlinear activation function, we vastly increase the class of functions we can reproduce from only linear functions to almost all functions, given a large enough network. The perceptron is now called a neuron, since these types of systems in many cases are modelled after a biological systems of neurons, in what is called deep learning.

This system now takes a set of input values and gives an output value, which is called a feedforward network. However, without good weights and biases the network will not perform well.

2.4.3 Backpropagation

Finding the optimal weights and biases for the network is done by first calculating the errors in the output and then minimizing it by propagating it through the backward through the weights of the network, in effect finding how much each weight contributed to this error. This process is called backpropagation. We are interested in finding the error of a given layer due to the weights. The error in last (L-th) layer is calculated by

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial z_j} = f'(z_j^L)\frac{\partial \mathcal{C}}{\partial a_j^L}, \tag{18}$$

where $f'(z_j^L)$ is the derivative of the activation function of the j-th neuron with respect to its input, and $a_j^L = f(z_j^L)$ is the activation of the j-th neuron. We also recognize this as the output of the network, while $\frac{\partial \mathcal{C}}{\partial a_j^L}$ is the derivative cost function with respect to this output. In the case of regression, this is just $a_j^L - t_j$, with t_j as the target value for that specific training set. We will come back to the expression in the classification case.

The error due to the previous ($l-1$-th) layer is found by propagating a single layer backwards with

$$\delta_j^{l-1} = \sum_k \delta_k^l w_{kj}^l f'(z_j^{l-1}). \tag{19}$$

This algorithm can then be used all the way until the input layer, where $z_j^{l-1} = x_j$, i.e. the input. The crucial realization is then that the error in the cost function due to the weights and the biases follow simple expressions in these δ, and as such we find the optimal update in weight- and bias-space to be

$$w_{jk}^l \leftarrow w_{jk}^l - \eta\delta_j^l a_k^{l-1}, \tag{20}$$
$$b_j^l \leftarrow w_j^l - \eta\delta_j^l \tag{21}$$

2.4.4 Softmax function

For multiclass classification, where the output layer can be represented as a vector of size K equal to the number of classes, the probability for an input to be in class k is given by the softmax function,

$$P(y_{ic} = 1 \mid \hat{x}_i, \hat{\theta}) = \frac{e^{(\hat{a}_i^{hidden})^T\hat{w}_c}}{\sum_{c'=0}^{C-1} e^{(\hat{a}_i^{hidden})^T\hat{w}_{c'}}} \tag{22}$$

Here $\hat{a}_i$ is the activation from the i-th layer. This is called the sofmax function. The negative log-likelihood (also known as multiclass cross entropy) then gives the cost function to be used in the backpropagation for the outer layer:

$$\mathcal{C}(\hat{\theta}) = -\log P(\mathcal{D} \mid \hat{\theta}) \tag{23}$$

Where

$$P(\mathcal{D} \mid \hat{\theta}) = \prod_{i=1}^{n}\prod_{c=0}^{C-1}[P(y_{ic}=1)]^{y_{ic}} \tag{24}$$

To be able to evaluate eq. (18), we then need the derivative of both the softmax function and multiclass cross entropy. Starting with the former,

$$\frac{\partial f(z_j)}{\partial z_i} = \frac{\partial}{\partial z_i}\frac{e^{z_j}}{\sum_k e^{z_k}} = f(z_j)(\delta_i j - f(z_i)), \tag{25}$$

where $\delta_i j$ is the kroenecker delta, which reduces to 1 in our case. The derivative of our current cost function is

$$\frac{\partial \mathcal{C}}{\partial a_j} = \frac{\partial}{\partial a_i^L}\left[-\sum_{i=1}^{n} y_i \log a_i^L + (1-y_i)\log\bigl(1-a_i^L\bigr)\right] \tag{26}$$
$$= \frac{-y_i(1-a_i^L)+(1-y_i)a_i^L}{a_i^L(1-a_i^L)} = \frac{a_i - y_i}{a_i^L(1-a_i^L)}. \tag{27}$$

Combining the two leaves us with the super nice expression for the errors in the last layers as

$$\delta_j^L = a_j^L - y_j \tag{28}$$

3 Methods and implementation

3.1 Linear regression model

From eq. (1) the model can be written on the form

$$E^i = \mathbf{X}^i \cdot \mathbf{J}, \tag{29}$$

where the design matrix $\mathbf{X}^i$ is composed of all the interactions $s_j^i s_k^i$ for the i-th configuration in the data set generated as the energies from random configurations of spins with $J = 1$. This model does not include the minus sign in eq. (1), so we expect to learn negative values of $\mathbf{J}$.

To perform the regression analysis we used our own code from project 1 for the OLS and ridge which can be found in `tools.py`. Here the lasso regression is done with `sklearn.linear_model.Lasso`.

3.2 Logistic regression model

This model is going to predict the phase of a given configuration of spins as either ordered or disordered. The output should then be 1 or 0 respectively, and the model is going to be on the form

$$p(E^i) = \frac{1}{1+e^{\mathbf{X}^i\cdot\mathbf{J}}} \tag{30}$$

which is just the sigmoid function. Here the data set is generated from Metha et al [1]. It is composed of energies $\mathbf{E}^i$ and their corresponding spin configurations $\mathbf{X}^i$.

We implemented our own algorithm for the logistic regression which can be found as `Logistic_Regression` in `logreg.py`. It uses gradient descent with an optional learning rate and L2-regularization to find the optimal parameters $\mathbf{J}$. Energies can then be predicted with the design matrix $\mathbf{X}^i$ and the performance of the model can be tested with the accuracy function, given by eq. (13).

3.3 Neural network

In the case of the neural networks, we again generated 1000 data sets consisting of random spin configurations and their energies in a $L = 40$ 1D ising model. This was split into a training set of 66% and test set of 33%. We used 200 mini-batches in our stochastic gradient descent, which corresponds to roughly 3 datapoints per batch on average. A grid search over several sizes of the hidden layer and learning rates was run, with both RELU and sigmoid as activation functions. TODO: Skriver again her, men har kanskje ikke skrevet det tidligere To represent the spin couplings, which was what we want the network to learn, we again used the outer product of the spins with themselves, and the output of the network was a single number with the identity as activation function. The cost function was the mean squared error.

In the two-dimensional Ising phase classification problem we trained the neural network on data from the Mehta et. al. [1]. The data consists of $L = 40$ 2D lattice grids generated with monte carlo for different temperatures $T \in 0.25, 0.5, \cdots, 3.75, 4$. We first labeled them as either ordered or disordered if they where below or above the critical temperature respectively, and set 33% of the data as test data. The input data was here the flattened array of spins, which means no topological information was presented to the network. As output was the two classes, either ordered or disordered.

One python function in `results_functions.py` implements stochastic gradient descent for both 1-dimensional regression and 2-dimensional classification.

4 Results

4.1 The one-dimensional Ising model

4.1.1 Using regression analysis

The ridge and lasso models were trained for several regularization values λ. The resulting $R2$-scores for the training and test set can be seen in fig. 1. The training set was composed of 666 samples and the test set of 334 samples, from a total data set of 1000 samples.

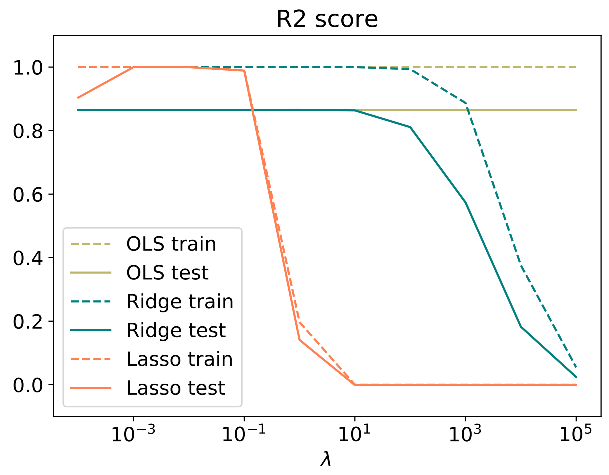


Figure 1: R2 score for different λ-values with OLS, ridge and lasso regression.

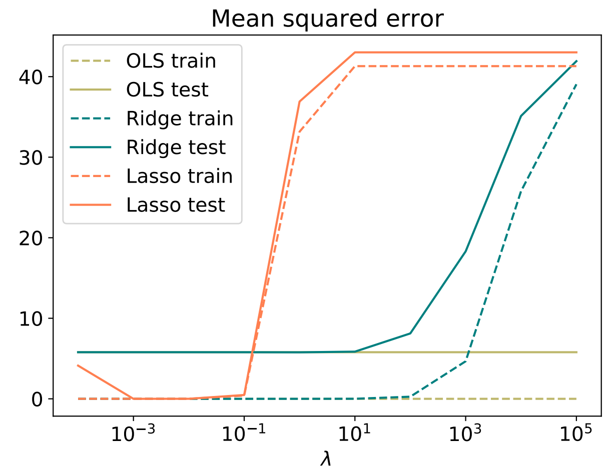


Figure 2: Mean squared error for different λ-values with OLS, ridge and lasso regression.

In Figures 3 to 5 the parameters of the different regression models are plotted for values of λ that give a good and bad $R2$-score.

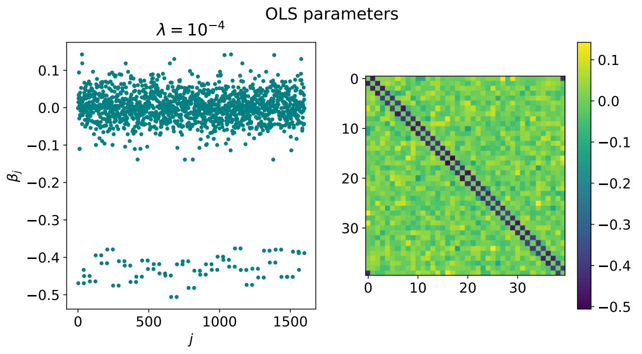


Figure 3: Parameters for OLS regression. The left figure is a matrix representations of the parameters, and the right is a plot of their values.

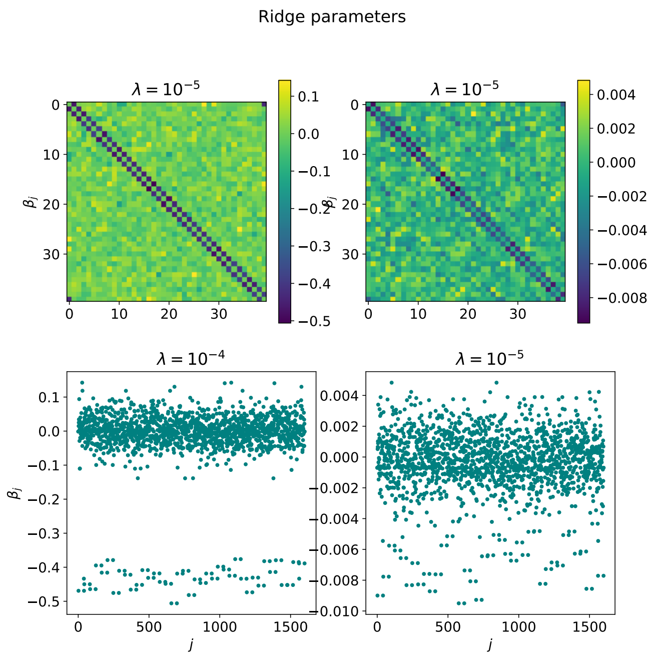


Figure 4: Parameters for ridge regression. Plotted for $\lambda = 10^{-4}$, that gave the best $R2$-score, and $\lambda = 10^{5}$, that gave the worst $R2$-score. The top figures are matrix representations of the parameters, and below their values are plotted.

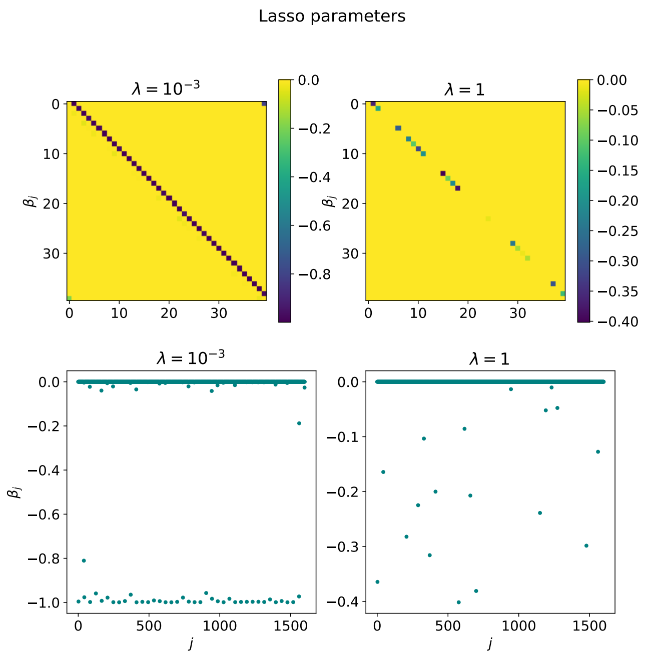


Figure 5: Parameters for lasso regression. Plotted for $\lambda = 10^{-4}$, that gave the best $R2$-score, and $\lambda = 1$, that gave the worst $R2$-score were some parameters still were different. The top figures are matrix representations of the parameters, and below their values are plotted.

Figure 6 show a bias-variance decomposition of ridge regression for different values of λ, here the data set was resampled using bootstrap.

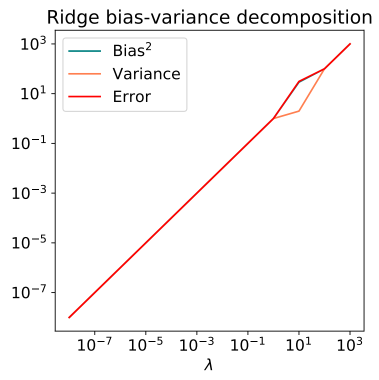
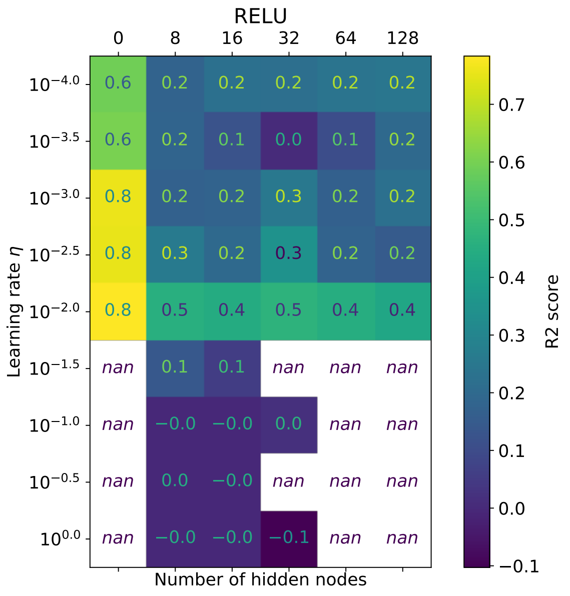


Figure 6: Bias-variance decomposition of ridge regression.

4.1.2 Using a neural network

The results of the neural network applied to the regression problem can be seen in fig. 7 and fig. 8 for relu and sigmoid respectively. The results of the best hyperparameters can be seen in table 1.



Learning rate η	0	8	16	32	64	128
$10^{-4.0}$	0.6	0.2	0.2	0.2	0.2	0.2
$10^{-3.5}$	0.6	0.2	0.1	0.0	0.1	0.2
$10^{-3.0}$	0.8	0.2	0.2	0.3	0.2	0.2
$10^{-2.5}$	0.8	0.3	0.2	0.3	0.2	0.2
$10^{-2.0}$	0.8	0.5	0.4	0.5	0.4	0.4
$10^{-1.5}$	nan	0.1	0.1	nan	nan	nan
$10^{-1.0}$	nan	−0.0	−0.0	0.0	nan	nan
$10^{-0.5}$	nan	0.0	−0.0	nan	nan	nan
$10^{0.0}$	nan	−0.0	−0.0	−0.1	nan	nan

Figure 7: R2 score of a grid search for a regression neural network trained with the sigmoid activation function in the hidden layer. Zero neurons in the hidden layer correspond to no hidden layer.

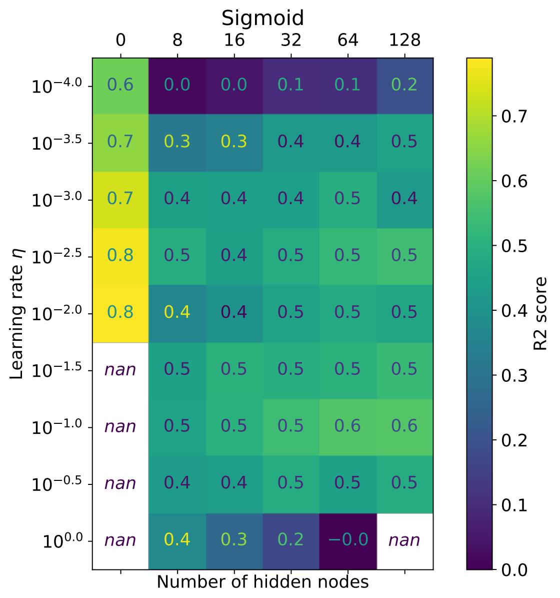


Figure 8: R2 score of a grid search over learning rate η and hidden layer size N_h for a regression neural network trained with the RELU activation function in the hidden layer.

4.2 The two-dimensional Ising model

4.2.1 Using logistic regression

The model was trained for several learning rates η and regularization values λ. To find the optimal values, the accuracy scores of the prediction of each combination was found from the test set. The maximal number of iterations in the gradient descent was set to 1000.

Table 2: Accuracy score from predictions on the same test set, for a selection of λ and η values.

	$\eta = 10^{-4}$	$\eta = 10^{-3}$	$\eta = 10^{-2}$	$\eta = 10^{-1}$
$\lambda = 0$	0.4827	0.4960	0.6860	0.6530
$\lambda = 10^{-4}$	0.4937	0.5060	0.6883	0.6527
$\lambda = 10^{-2}$	0.4963	0.5010	0.6877	0.6530
$\lambda = 10^{-1}$	0.4770	0.5043	0.6847	0.6537
$\lambda = 10^{2}$	0.4887	0.4973	0.6880	0.6597

As seen in table 2; $\eta = 10^{-2}$ and $\lambda = 10^{-4}$ gave the best accuracy equal to 0.6883. To comparison the `sklearn`-algorithm gave an accuracy of 0.6710 on the same test set. The results are also plotted in fig. 9.

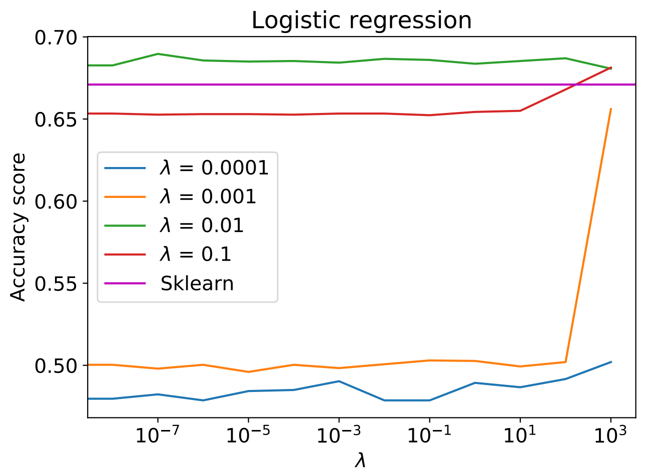


Figure 9: Accuracy score as a function of lambda for a selection of learning rates, compared with sklearn.

4.2.2 Using a neural network

The neural network generally performed much better at classification than the logistic regression for the current problem. Figures 10 to 12 show the misprediction rate of our implemented network as a function of eta and number of hidden nodes.

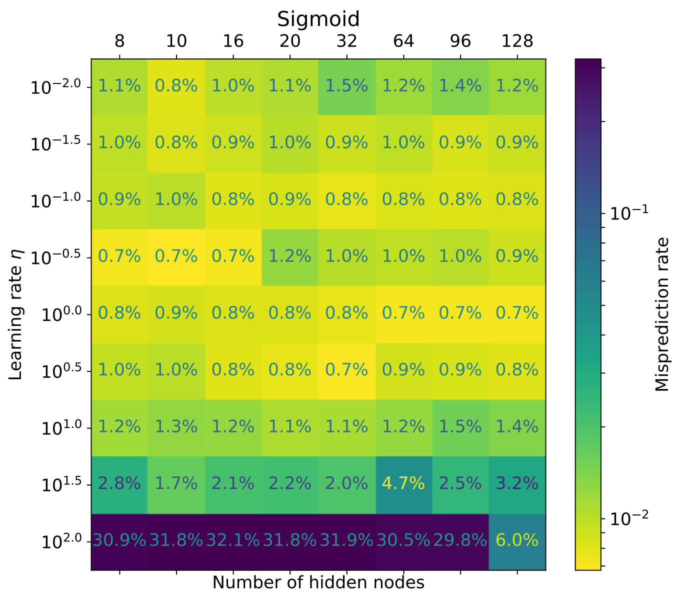


Figure 10: Misprediction rate (1 - Accuracy) of a grid search for a classification neural network trained with the sigmoid activation function in the hidden layer.

Table 1: Best results for regression neural network for different layer sizes.

		RELU			Sigmoid	
N_h	η	R2 Score	Epoch	η	R2 Score	Epoch
0	$10^{-2.0}$	0.784	199	$10^{-2.0}$	0.789	66
8	$10^{-2.0}$	0.456	8	$10^{-2.5}$	0.490	79
16	$10^{-2.0}$	0.441	7	$10^{-1.5}$	0.506	13
32	$10^{-2.0}$	0.470	8	$10^{-1.0}$	0.545	5
64	$10^{-2.0}$	0.448	20	$10^{-1.0}$	0.575	5
128	$10^{-2.0}$	0.422	7	$10^{-1.0}$	0.578	7

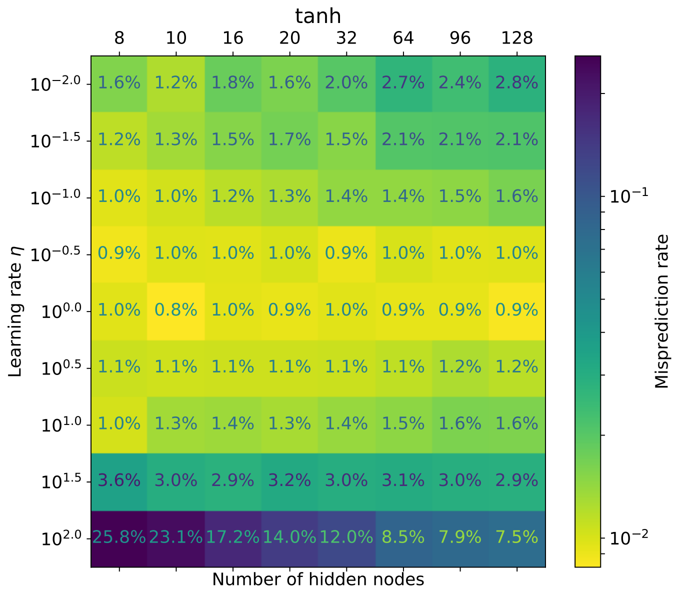


Figure 11: Misprediction rate (1 - Accuracy) of a grid search for a classification neural network trained with the tanh activation function in the hidden layer.

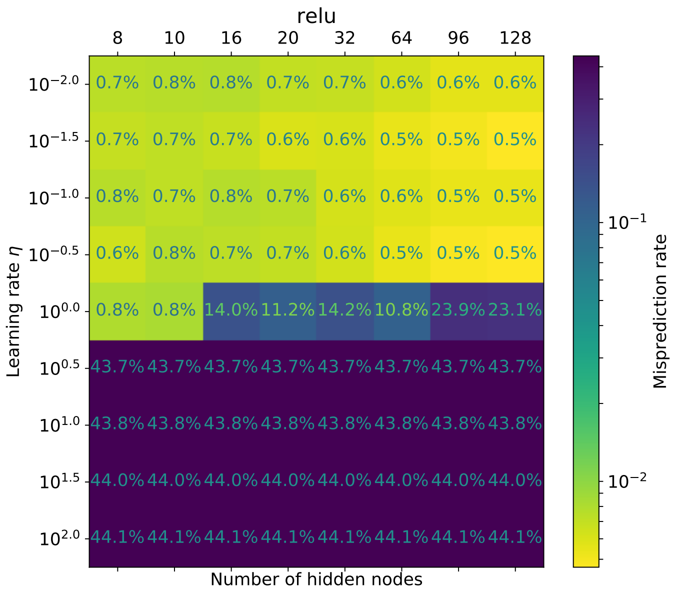


Figure 12: Misprediction rate (1 - Accuracy) of a grid search over learning rate η and neuralfor a classification neural network trained with the RELU activation function in the hidden layer.

Table 3: Optimal values after a grid search. Most of the activation functions have approximately the same optimal performance, but the optimal hyper parameters vary between function and layer sizes.

	RELU			sigmoid			tanh		
	η	Accuracy	Epoch	η	Accuracy	Epoch	η	Accuracy	Epoch
N_h									
8	$10^{-0.5}$	0.994	4	$10^{-1.0}$	0.991	28	$10^{-0.5}$	0.991	100
10	$10^{-1.5}$	0.993	16	$10^{-2.0}$	0.992	52	$10^{0.0}$	0.992	100
16	$10^{-1.5}$	0.993	8	$10^{0.0}$	0.992	6	$10^{0.0}$	0.990	83
20	$10^{-1.5}$	0.994	18	$10^{0.5}$	0.992	28	$10^{0.0}$	0.991	47
32	$10^{-1.5}$	0.994	21	$10^{0.5}$	0.993	36	$10^{-0.5}$	0.991	99
64	$10^{-0.5}$	0.995	27	$10^{0.0}$	0.993	41	$10^{0.0}$	0.991	21
96	$10^{-0.5}$	0.995	35	$10^{-0.5}$	0.993	36	$10^{0.0}$	0.991	10
128	$10^{-1.5}$	0.995	40	$10^{0.0}$	0.993	29	$10^{0.0}$	0.991	36

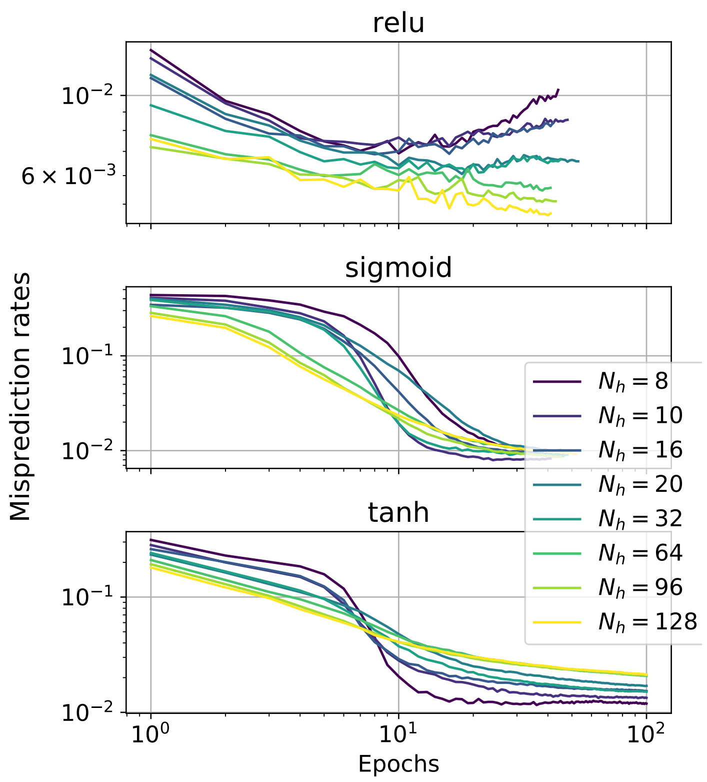


Figure 13: Misprediction rate (1 - Accuracy) for a given learning rate $\eta = 10^{-1.5}$ where all the networks performs reasonably well.

5 Discussion

5.1 Energy prediction

As seen in fig. 1, the lasso method performs much better than the ridge and ordinary least squares methods for some λ-values. However, it quickly falls down in performance when $\lambda > 10^{-1}$. For ridge the performance does not seem to be affected before $\lambda > 10$. From Figures 3 to 5 it is obvious that the differences between the models is a result of how they find their parameters, or coupling constants $\hat{\mathbf{J}}$. The coupling constants predicted by OLS and ridge move towards 0 and -0.5, while lasso predicts them towards 0 and -1.0. Their performance seem to get worse as the spread around these values increase. The OLS and ridge methods find symmetric coupling constants, while Lasso gives only entries on the upper diagonal. This invariance is expected, as the coupling direction is a gauge degree of freedom of the problem. However, it is not obvious why the different methods end up in different gauges. As Mehta et. al. states, different regularization parameters give different results because of the effect of the cost function on the optimal point in the parameter space.

The mean squared error seems very related to the $R2$-score, as can be seen in and fig. 1 fig. 2, where the MSE blow up when the $R2$ becomes bad. The MSE is however very high for ridge and OLS where it gives good $R2$-scores. Here it is also worth mentioning that OLS and ridge has a notable deviation in fig. 1 between training and test set, that can be an indication of overfitting. The reason for this could be the low number of samples used in the data set. When we tested it for a bigger data set, the deviation disappeared, but the differences between the performance of the methods blurred out. The bias-variance decomposition of the ridge method shows a logarithmic relationship between the mean squared error and the regularization parameter λ. Obviously a choosing a low λ will give minimal error.

We see that the neural net did not perform very well in the case of the regression problem. From figs. 7 and 8 it is clear that the best performing network is one with no hidden layer. In this case, the network reduces to an OLS-solver with gradient descent, as the activation function to the last layer is the identity. This can as such not be considered as results for the neural network.

Our network also has trouble with running for large values of η. This might be due to numerical instabilities in our implementation, but it might also happen in a convex minimization problem, if for example the learning rate times the gradient becomes more than twice as large as the distance to the minima, each step might produce a worse result than the previous.

5.2 Phase prediction

The logistic regression gave poor performance on predicting the phase, as seen in fig. 9. Still our model got slightly better predictions than that of Sklearn. However, our neural net performed way better in this problem. Almost every run only needed a single epoch to outperform the logistic regression, and then continued to improve after that. We could probably have improved the logistic regression by using stochastic gradient descent instead of just steepest descent, though as seen in Metha et. al. [1] this would still be bad in comparison to the neural network.

The result of the grid searches in figs. 10 to 12 shows that the different activation function exhibit slightly different behaviour. Especially RELU behaves badly for too large learning rates. One hypothesis is that because the value of the gradient is zero for $x < 0$, then there is a possibility that all weights to the hidden layer get set to negative values, causing a full stop in the training. If this is the case, it would probably have been amended if we used an ELU activation function instead, which has a non-zero derivative for negative function values.

There is a tendency for tanh and sigmoid to prefer smaller sizes of the hidden layer than RELU. Considering the number of epochs used as seen in table 3, this can be explained by the fact that RELU uses less epochs to obtain the same result, i.e. it trains and obtains optimal values faster. One possible explanation is that the derivative of the RELU activation function doesn't saturate for large function values.

Using a convolutional network we could probably increase our prediction rate in the classification case, as we could then present the network with information on topology, which decreases the amount of information the network would have to learn. However, already now the net was in some way able to learn the underlying structure of the spins, at least better than the logistic regression. This is the strength of the hidden layers, which may preprocess the input data before delivering it to the output for classification.

6 Conclusion

It is clear from the two methods that neural networks and ordinary regression methods have their advantages and disadvantages. When the complexity of the problem is small, ordinary least squares outperforms the network, seen by the R2 scores of 0.838 with OLS versus 0.578 with the network. In the case of the 1D Ising model, this is probably due the problem being a linear one, where the energies are a linear combination of the paired spins. Since we were able to provide the solver with this topological information, the solver outperformed the network which was a huge overkill.

In the classification problem we did not provide either solver with corresponding topologies, and in this case the network performed the best. In some way it was able to learn underlying structure of the spins in a more nuanced way than the logistic regression. The best network could predict the correct phase in 99.4% of the cases.

References

[1] Pankaj Mehta, Marin Bukov, Ching-Hao Wang, Alexandre G. R. Day, Clint Richardson, Charles K. Fisher, and David J. Schwab. A high-bias, low-variance introduction to Machine Learning for physicists. *arXiv:1803.08823 [cond-mat, physics:physics, stat]*, March 2018. arXiv: 1803.08823.

[2] Michael A. Nielsen. Neural Networks and Deep Learning. 2015.

[3] Halvard Sutterud and Heine Aabø. Repository for work on the course FYS-STK4155: Applied data analysis and machine learning at the University of Oslo - halvarsu/FYS-STK4155, November 2018. original-date: 2018-09-12T12:02:27Z.