Regression and classification by regression methods and neural networks

Thomas Larsen Greiner (tagreine@student.matnat.uio.no)
https://github.com/tagreine/school/tree/master/FYS-STK4155/Project2

Hao Zhao (haozh@ifi.uio.no)
https://github.com/haozh7109/Machine_Learning_Course/tree/master

Hong Li (hong.li@geo.uio.no)
https://github.com/tagreine/school/tree/master/Project2

Abstract

This report compares regression methods and neutral networks in determining the value of the coupling constant for the energy of the one-dimensional Ising model and classifying the phase of the two-dimensional Ising model. These regression methods are linear regression methods, i.e. ordinary least squared (OLS), Ridge and Lasso regression. Additionally, we implement SVD in order to avoid the singularity problem of the design matrix in the regression methods. For the neural network, we build a simple network with two hidden layers. The results for regression show the Lasso regression outperforms all other methods in predicting energy states and in terms of estimator similarity. We believe the reason is the connection between the discrete nature of the model (only -1 and 1) and coupling constants, and the L_1 regularization leads to sparsity of the estimator, which fits the modeled problem quite well. In addition, the linear regression approach using L_1 regularization has less computational cost compared to the neural network. The regression also gives an explicit formulation and direct connection to the coupling constants, instead of just a function with weights and biases as by the neural network. The results for classification show the L_2 regularized neural network outperforms the logistic classifier, even while using L_1 regularization. The neural network gives an almost perfect fit with respect to both the training- and test dataset, which implies a good generalization of the classifiers prediction performance. However, the Tanh activation does not give good prediction accuracy, and is worse than the L_1 regularized logistic classifier. The reason could be implementation related (our own logistic classifier is not optimized) or other.

Introduction

Arthur Samuel introduced Machine learning (ML) almost 60 years ago. This field studies how computers can learn from data automatically. ML uses statistical techniques to enable the computer system to analyze and identify the patterns of the data, and to make the decisions with minimal human interventions. ML has been widely used in scientific research as well as commercial activities.

Machine learning mainly have three branches: supervised learning, unsupervised learning and reinforcement learning. In this report, we focus on supervised learning in regression and classification. The supervised learning algorithm uses the training samples with associated known output to infer the relation or function to be able to map the input and output by an iterative feed-forward propagation and error backward propagation process. The most widely used supervised learning algorithms include linear regression, logistic regression, naive Bayes, support vector machines and Neural Networks. In this report, we analyze the parameterization of linear regression problem, logistic regression/classification problem based on the 1D and 2D Ising model and compare the performance between the conventional regression methods with the Neural Network (Multiple Layer Perceptron) based method.

The report is divided into three following sections. They are 'Theoretical background', 'Results and discussion' and 'Conclusions'. The 'Theoretical background' is the theory for linear regression analysis, including ordinary least squares (OLS), Ridge regression, Lasso regression and singular value decomposition to avoid singularity issues in the design matrix, and neural network with algorithms for computing the forward- and backward propagation. We also add a short introduction on the theory of classification using logistic regression. Next follows 'Results and discussion' section from the regression analysis part using OLS, Ridge, Lasso and neural networks, with bootstrap resampling for model evaluation, followed by results and discussion section for the classification problem comparing a logistic classifier to a neural network classifier. The last section follows a summary of the conclusions.

Theoretical background

Linear regression

We start by defining our training set as $\{x^{(i)}, y^{(i)}\}$. The linear regression model y is defined as

$$y = \beta^T x + \hat{\epsilon} \tag{1}$$

where $\hat{\epsilon}$ is the noise term and β is the parameter we want to estimate in order to fit our model to the data. By expanding our feature-space, we will instead define our linear model (1) as

$$\boldsymbol{y} = \boldsymbol{x\beta} + \hat{\epsilon}$$

where $\boldsymbol{\beta}$ is a $(p \times 1)$-dimensional vector and $\boldsymbol{x}$ is a $(n \times p)$-dimensional matrix, which gives the design matrix

$$\boldsymbol{x} = \begin{bmatrix} x_{1,1} & \cdots & x_{1,p} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \cdots & x_{n,p} \end{bmatrix} \tag{2}$$

In order to estimate the parameters $\boldsymbol{\beta}$, we need to define a cost function for the model. In the case of linear regression, we can write the cost function, including L_2 regularization (Ridge regression), as follows (Hastie et al, 2001)

$$J(\boldsymbol{\beta}; \lambda_2) = (\boldsymbol{y} - \boldsymbol{x\beta})^T(\boldsymbol{y} - \boldsymbol{x\beta}) + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta} \tag{3}$$

where λ_2 is known as the penalty parameter. The Ridge estimation of the parameters $\boldsymbol{\beta}$ is then given by

$$\widetilde{\boldsymbol{\beta}} = (\boldsymbol{x}^T\boldsymbol{x} + \lambda_2\boldsymbol{I})^{-1}\boldsymbol{x}^T\boldsymbol{y} \tag{4}$$

leading to the estimated solution

$$\widetilde{\boldsymbol{y}} = \boldsymbol{x}\widetilde{\boldsymbol{\beta}} = \boldsymbol{x}(\boldsymbol{x}^T\boldsymbol{x} + \lambda_2\boldsymbol{I})^{-1}\boldsymbol{x}^T\boldsymbol{y} \tag{5}$$

where $\boldsymbol{I}$ is the identity matrix. It is also possible to add other penalizations to the estimators $\boldsymbol{\beta}$, such as the Lasso regression, which has a loss function that reads (van Wieringen, 2015)

$$L(\boldsymbol{\beta};\lambda_1) = (\boldsymbol{y} - \boldsymbol{x\beta})^T(\boldsymbol{y} - \boldsymbol{x\beta}) + \lambda_1\|\boldsymbol{\beta}\|_1 \tag{6}$$

where $\|\boldsymbol{\beta}\|_1$ is the L_1 norm of the estimators, and is just a sum of the absolute values. However, there exist no well-defined estimator for the of the Lasso regression method, and therefore need to be solved by other means, such as iterative methods, which could be a downside. The Lasso regression upside is its ability to lead to sparsity of its estimator, implying many zeros within $\widetilde{\boldsymbol{\beta}}$, which is in contrast to L_2 norm ($\|\boldsymbol{\beta}\|_2$) which will not contain zeros (van Wieringen, 2015).

In the case of singularity (of the matrix $\boldsymbol{x}^T\boldsymbol{x}$) as is the case for the Ising model, Ridge estimation works as it applies a small value to the diagonal elements of matrix $\boldsymbol{x}^T\boldsymbol{x}$, making it non-singular. Another approach is to use singular value decomposition, both for OLS ($\lambda = 0$) and Ridge regression. The OLS estimator is rewritten in terms of SVD as follows (van Wieringen, 2015)

$$\begin{aligned}
\widetilde{\boldsymbol{\beta}}_{\text{OLS}} &= (\boldsymbol{x^T x})^{-1}\boldsymbol{x}^T\boldsymbol{y} \\
&= (\boldsymbol{VDU}^T\boldsymbol{UDV}^T)^{-1}\boldsymbol{VDU}^T\boldsymbol{y} \\
&= (\boldsymbol{VD}^2\boldsymbol{V}^T)^{-1}\boldsymbol{VDU}^T\boldsymbol{y} \\
&= \boldsymbol{VD}^{-2}\boldsymbol{V}^T\boldsymbol{VDU}^T\boldsymbol{y} \\
&= \boldsymbol{VD}^{-2}\boldsymbol{DU}^T\boldsymbol{y}
\end{aligned} \tag{7}$$

and for the Ridge estimator in terms of SVD (van Wieringen ,2015)

$$\begin{aligned}
\widetilde{\boldsymbol{\beta}}_{\text{Ridge}} &= (\boldsymbol{x^T x} + \lambda\boldsymbol{I})^{-1}\boldsymbol{x}^T\boldsymbol{y} \\
&= (\boldsymbol{VDU}^T\boldsymbol{UDV}^T + \lambda\boldsymbol{I})^{-1}\boldsymbol{VDU}^T\boldsymbol{y} \\
&= (\boldsymbol{VD}^2\boldsymbol{V}^T + \lambda\boldsymbol{VV}^T)^{-1}\boldsymbol{VDU}^T\boldsymbol{y} \\
&= \boldsymbol{V}(\boldsymbol{D}^{\mathbf{2}} + \lambda\boldsymbol{I})^{\mathbf{-1}}\boldsymbol{DU}^T\boldsymbol{y}
\end{aligned} \tag{8}$$

An algorithm for computing the Ridge regression is presented in Appendix A. Note that for computing OLS we can use the results from $\widetilde{\boldsymbol{\beta}}_{\text{Ridge}}$ but set $\lambda = 0$, which gives the estimation given in (8).

Classification and logistic regression

The classification problem resembles the regression problem, except that the target values we want to predict in classification are discrete variables (categories), which could take a binary or multiclass form. In binary classification, which we have studied in this report, the target values in $\boldsymbol{y}$ takes only two values, 0 and$+1$, or -1 and$+1$. The ultimate goal in classification, given a training set $\{x^{(i)}, y^{(i)}\}$, is to predict the target value $y^{(i)} \in \{0,1\}$ by some input $\boldsymbol{x}^{(i)} \in \mathbb{R}^{p\times 1}$, typically via some non-linear function $\mathrm{p}(\boldsymbol{z})$, also known as the logistic function. The logistic function takes the form

$$\mathrm{p}(\boldsymbol{z}) = \frac{1}{1 + e^{-z}} \tag{9}$$

where $\boldsymbol{z}$ is defined as

$$\boldsymbol{z} = \sum_{j=1}^{p} \beta_j^T \boldsymbol{x}_j + \beta_0 = \boldsymbol{x}\boldsymbol{\beta} \tag{10}$$

where $\boldsymbol{\beta}$ are the parameters we want to estimate. Note that the model in equation 10 is similar to the model defined in linear regression, only that this model is passed through a non-linear function (9), constraining the values between 0 and 1. The function $\mathrm{p}(\boldsymbol{z})$ as defined above, is commonly known as the Sigmoid function.

As with linear regression, in order to estimate $\boldsymbol{\beta}$ we need to define a cost function for the model. Because of minimization of the cost function in logistic regression leads to non-linear equation for the estimators $\boldsymbol{\beta}$ (Jensen, 2018); the estimation problem needs to be solved by some iterative method for minimizing the cost function. For a given training example $\{x^{(i)}, y^{(i)}\}$ the cost function for logistic regression including L_2 regularization is defined as (Ng, 2018)

$$C(\beta) = -\frac{1}{m}\left(\sum_{i=1}^{m} y^{(i)} \log\left(p\left(x^{(i)}\right)\right) - \left(1 - y^{(i)}\ \right) \log\left(1 - p\left(x^{(i)}\right)\right)\right) + \lambda \frac{1}{2m}\sum_{j=1}^{p} \beta_j^2 \qquad (11)$$

where the second term is the L_2 regularization. We can rewrite it in vectorized form by including all training examples

$$C(\boldsymbol{\beta}) = -\frac{1}{m}(\boldsymbol{y}\log(\boldsymbol{p})\ - (1-\boldsymbol{y})\log(1-\ \boldsymbol{p})) + \lambda\frac{1}{2m}\boldsymbol{\beta}^T\boldsymbol{\beta} \qquad (12)$$

We take the partial derivatives with respect to the estimators $\boldsymbol{\beta}$

$$\begin{aligned}
&\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} \\
&= -\frac{1}{m}\left(\boldsymbol{y}\frac{\partial \log(\boldsymbol{p})}{\partial \boldsymbol{\beta}} - (1-\boldsymbol{y})\frac{\partial \log(1-\boldsymbol{p})}{\partial \boldsymbol{\beta}}\right)\frac{\partial \boldsymbol{p}}{\partial \boldsymbol{\beta}}\frac{\partial \boldsymbol{\beta}^T}{\partial \boldsymbol{\beta}}\boldsymbol{x} + \ \lambda\frac{1}{m}\boldsymbol{\beta} \\
&= -\frac{1}{m}\left(\boldsymbol{y}\frac{1}{\boldsymbol{p}} - (1-\boldsymbol{y})\frac{1}{\boldsymbol{1-p}}\right)\frac{\partial \boldsymbol{p}}{\partial \boldsymbol{\beta}}\frac{\partial \boldsymbol{\beta}^T}{\partial \boldsymbol{\beta}}\boldsymbol{x} + \lambda\frac{1}{m}\boldsymbol{\beta} \\
&= -\frac{1}{m}\left(\boldsymbol{y}\frac{1}{\boldsymbol{p}} - (1-\boldsymbol{y})\frac{1}{\boldsymbol{1-p}}\right)\boldsymbol{p}(1-\boldsymbol{p})\boldsymbol{x} + \lambda\frac{1}{m}\boldsymbol{\beta} \\
&= -\frac{1}{m}\big((\boldsymbol{y}-\boldsymbol{p})\boldsymbol{x} + \lambda\boldsymbol{\beta}\big) \qquad (13)
\end{aligned}$$

Finally, we update the estimators by gradient methods, such as gradient descent. The gradient descent optimization, including L_2 regularization, is defined as (Ng, 2018)

$$
\begin{aligned}
\widetilde{\boldsymbol{\beta}} &:= \widetilde{\boldsymbol{\beta}} - \eta \frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} \\
&:= \widetilde{\boldsymbol{\beta}} - \eta \left(-\frac{1}{m} \Big((\boldsymbol{y} - \boldsymbol{p}) \boldsymbol{x} + \lambda \widetilde{\boldsymbol{\beta}} \Big) \right)
\end{aligned} \tag{14}
$$

where η parameter controls the step-length of each gradient descent iteration, which is also known as the learning rate. The main idea behind gradient descent, is to adjust the estimators in the direction where the gradient of the cost function is steepest, which ensures a stepwise migration towards a local (or global) minimum. An algorithm for computing the logistic cost function and gradient optimization, including L_2 penalty is provided in Appendix A. Note that for computing OLS we can set $\lambda = 0$.

Neural network

The building blocks of a neural network is build up by an input layer, hidden layer(s) and an output layer. All layers consists of neurons, which are connected from the former layer via matrix–vector multiplication and non-linear activation functions. The forward computation from one layer to the next is known as forward propagation. In order to explain the forward propagation, we start with defining our training data as $\{x^{(i)}, y^{(i)}\}$, where a single input vector $x^{(i)} \in \mathbb{R}^{p\times 1}$ as presented in Figure 1. Feeding the network with n training examples, gives the input matrix as

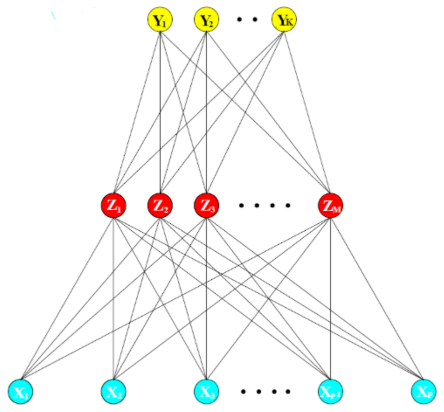


Figure 1: *Neural network architecture, with one hidden layer. Figure borrowed from Hastie et al (2001).*

$$
\boldsymbol{X} = \begin{bmatrix} | & | & | & | \\ \boldsymbol{x}^{(1)} & \boldsymbol{x}^{(2)} & \dots & \boldsymbol{x}^{(n)} \\ | & | & | & | \end{bmatrix} \tag{15}
$$

The input vectors are fed forward to all neurons in the next layer (hidden layer in Figure 1). The non-linear transform from the input layer to next, is defined by the equation

$$g\left(\boldsymbol{z}^{(2)}\right) = g\left(\begin{bmatrix} W_{11}^{(1)} & \cdots & W_{1p}^{(1)} \\ \vdots & \ddots & \vdots \\ W_{m1}^{(1)} & \cdots & W_{mp}^{(1)} \end{bmatrix}\begin{bmatrix} x_1^{(1)} \\ \vdots \\ x_p^{(1)} \end{bmatrix} + \begin{bmatrix} b_1^{(1)} \\ \vdots \\ b_m^{(1)} \end{bmatrix}\right) = g\left(\boldsymbol{W}^{(1)}\boldsymbol{x}^{(1)} + \boldsymbol{b}^{(1)}\right) \qquad (16)$$

where the function $g(\boldsymbol{z})$ is the non-linear activation function we compute before feeding forward to the next layer. The activation is commonly denoted for any layer l as $a^{(l)} = g\left(\boldsymbol{z}^{(l)}\right)$, which is a common notation when deriving the forward- and backward propagation algorithm. The matrix $\boldsymbol{W}^{(l)} \in \mathbb{R}^{m\times p}$ are the weights and $\boldsymbol{b}^{(l)} \in \mathbb{R}^{m\times 1}$ are the biases in layer l (with m neurons), and defines the parameters which we want to learn.

There exist many kinds of activation functions. Ng and Katanforoosh (2018) lists a few examples of the most familiar activation functions in the Deep learning course notes

$$g(\boldsymbol{z}) = \frac{1}{1+e^{-\boldsymbol{z}}} \qquad \text{(Sigmoid)}$$

$$g(\boldsymbol{z}) = \max(\boldsymbol{z}, 0) \qquad \text{(Relu)}$$

$$g(\boldsymbol{z}) = \frac{e^{\boldsymbol{z}} - e^{-\boldsymbol{z}}}{e^{\boldsymbol{z}} + e^{-\boldsymbol{z}}} \qquad \text{(Tanh)}$$

The forward propagation algorithm is described in more detail as

1. Set $\boldsymbol{x}^{(1)} = \boldsymbol{a}^{(1)}$. Note that in these examples, inputs are column vectors.
2. Compute the first argument $\boldsymbol{z}^{(2)} = \boldsymbol{W}^{(1)}\boldsymbol{a}^{(1)} + \boldsymbol{b}^{(1)}$
3. Compute the activation function $\boldsymbol{a}^{(2)} = g\left(\boldsymbol{z}^{(2)}\right)$ for the first hidden layer
4. Compute the argument in the next layers $\boldsymbol{z}^{(l+1)} = \boldsymbol{W}^{(l)}\boldsymbol{a}^{(l)} + \boldsymbol{b}^{(l)}$
5. Compute the activation function $\boldsymbol{a}^{(l+1)} = g\left(\boldsymbol{z}^{(l+1)}\right)$

An algorithm for computing the forward propagation with two hidden layers is presented in Appendix B. In order to train the weights and biases, we first need to define a cost function for the output and define its partial derivatives with respect to the weights and biases. In our example, we will use the cost function as described in the theory part for logistic regression. The cost function for the output layer, including L_2 regularization, is defined as

$$\begin{aligned} C(\boldsymbol{W},\boldsymbol{b}) = &-\frac{1}{m}\Big(\,\boldsymbol{y}\log\Big(g\big(\boldsymbol{z}^{(L)}\big)\Big) + \ (1-\boldsymbol{y})\log\Big(1-g\big(\boldsymbol{z}^{(L)}\big)\Big)\Big) \\ &+\frac{\lambda}{2m}\boldsymbol{W}^{(L-1)^T}\boldsymbol{W}^{(L-1)} \end{aligned} \tag{17}$$

The weights in each layer are updated using the same gradient method as explained for logistic regression, i.e. gradient descent. Updating the weights and biases for any layer l is done by computing the gradient descent optimization

$$\boldsymbol{W}^{(l)} \coloneqq \boldsymbol{W}^{(l)} - \eta\frac{\partial C(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{W}^{(l)}} \tag{18}$$

and

$$\boldsymbol{b}^{(l)} \coloneqq \boldsymbol{b}^{(l)} - \eta\frac{\partial C(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{b}^{(l)}} \tag{19}$$

We take the partial derivative with respect to the weights in the output layer

$$\begin{aligned} \frac{\partial C(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{W}^{L-1}} = &-\frac{1}{m}\boldsymbol{y}\frac{1}{g(\boldsymbol{z}^{(L)})}\frac{\partial g\big(\boldsymbol{z}^{(L)}\big)}{\partial \boldsymbol{z}^{(L)}}\frac{\partial \boldsymbol{z}^{(L)}}{\partial \boldsymbol{W}^{L-1}} \\ &-\frac{1}{m}\,(1-\boldsymbol{y})\frac{1}{1-g(\boldsymbol{z}^{(L)})}(-1)\frac{\partial g\big(\boldsymbol{z}^{(L)}\big)}{\partial \boldsymbol{z}^{(L)}}\frac{\partial \boldsymbol{z}^{(L)}}{\partial \boldsymbol{W}^{L-1}}+\frac{\lambda}{m}\boldsymbol{W}^{(L-1)} \\ = &-\frac{1}{m}\,\boldsymbol{y}\frac{1}{g(\boldsymbol{z}^{(L)})}g\big(\boldsymbol{z}^{(L)}\big)\Big(1-g\big(\boldsymbol{z}^{(L)}\big)\Big)\,\boldsymbol{a}^{(L)^T} \end{aligned}$$

$$+\frac{1}{m}\,(1-\boldsymbol{y})\frac{1}{1-g(\boldsymbol{z}^{(L)})}g\big(\boldsymbol{z}^{(L)}\big)\Big(1-g\big(\boldsymbol{z}^{(L)}\big)\Big)\,\boldsymbol{a}^{(L)^T}+\frac{\lambda}{m}\boldsymbol{W}^{(L-1)}$$

$$=-\frac{1}{m}\Big(\boldsymbol{y}-g\big(\boldsymbol{z}^{(L)}\big)\Big)\,\boldsymbol{a}^{(L)^T}+\frac{\lambda}{m}\boldsymbol{W}^{(L-1)} \tag{20}$$

The error for the output layer is defined as $\boldsymbol{\delta}^{(L)}=-\Big(\boldsymbol{y}-g\big(\boldsymbol{z}^{(L)}\big)\Big)$. The error in any layer l and partial derivative of the cost function generalized as

$$\boldsymbol{\delta}^{(l)}=\Big(\boldsymbol{W}^{(l)^T}\boldsymbol{\delta}^{(l+1)}\Big)\bullet g'\big(\boldsymbol{z}^{(l)}\big) \tag{21}$$

and

$$\frac{\partial C(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{W}^{(l)}}=\frac{1}{m}\Big(\boldsymbol{\delta}^{(l+1)}\boldsymbol{a}^{(l)^T}+\lambda\boldsymbol{W}^{(l)}\Big) \tag{22}$$

where • is the Hadamard product, which imply an element-wise multiplication between the arguments. This result comes from the partial derivatives with respect to the weights in any of the former layers. By introducing the chain-rule to the results derived in the forward propagation algorithm, we see that the partial derivative with respect to $\boldsymbol{W}^{(L-2)}$ gives (excluding regularization)

$$\begin{aligned}\frac{\partial C(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{W}^{(L-2)}}&=\frac{\partial C(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{a}^{(L)}}\frac{\partial \boldsymbol{a}^{(L)}}{\partial \boldsymbol{z}^{(L)}}\frac{\partial \boldsymbol{z}^{(L)}}{\partial \boldsymbol{a}^{(L-1)}}\frac{\partial \boldsymbol{a}^{(L-1)}}{\partial \boldsymbol{z}^{(L-1)}}\frac{\partial \boldsymbol{z}^{(L-1)}}{\partial \boldsymbol{W}^{(L-2)}}\\&=\big(g\big(\boldsymbol{z}^{(L-1)}\big)-\boldsymbol{y}\big)\boldsymbol{W}^{(L-1)}\bullet g'(\boldsymbol{z}^{L-1})\boldsymbol{a}^{(L-1)^{\mathrm{T}}}\\&=\Big(\boldsymbol{W}^{(L-1)^T}\boldsymbol{\delta}^{(L)}\Big)\bullet g'\big(\boldsymbol{z}^{(L-1)}\big)\boldsymbol{a}^{(L-1)^T}\\&=\boldsymbol{\delta}^{(L)}\boldsymbol{a}^{(L-1)^T}\end{aligned} \tag{23}$$

The scheme for estimating the errors in all layers and neurons, and respective gradients for weights and biases, is known as backpropagation. The task of the backpropagation algorithm is to determine the change to the weights and biases in terms of what relative proportions causes the most decrease to the cost. By following the course notes of Ng and Katanforoosh (2018) we can use the derivations above and the results from the forward propagation algorithm, to do the backward propagation algorithm. This is described in more detail as

1. Perform forward propagation (as explained), for all layers.
2. Compute the error in the output layer $\boldsymbol{\delta}^{(L)} = -\left(\boldsymbol{y} - g\left(\boldsymbol{z}^{(L)}\right)\right)$.
3. For layer $l = L-1, L-2, \dots, 2$

   compute the error: $\boldsymbol{\delta}^{(l)} = \left(\boldsymbol{W}^{(l)^T} \boldsymbol{\delta}^{(l+1)}\right) \bullet g'(\boldsymbol{z}^{(l)})$

4. Compute the partial derivatives with respect to weights and biases using the following results

   Weights: $\frac{\partial C(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{W}^{(l)}} = \frac{1}{m}\left(\boldsymbol{\delta}^{(l+1)} \boldsymbol{a}^{(l)^T} + \lambda \boldsymbol{W}^{(l)}\right)$

   Biases: $\frac{\partial C(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{b}^{(l)}} = \frac{1}{m}\boldsymbol{\delta}^{(l+1)}$

An algorithm for computing the backward propagation with two hidden layers is presented in Appendix B. In order to update the weights and biases we need to implement an optimization scheme, such as the gradient descent algorithm, which ensure a stepwise migration towards a local minimum. There are basically three approaches of gradient descent method; batch gradient descent, which takes the entire dataset and computes the gradient of the cost function; mini-batch gradient descent, which takes mini-batches of size $m > 1$; stochastic gradient descent, which is a special case of mini-batch gradient descent, with $m = 1$. Implementation of one mini-batch gradient descent is described in more detail as

1. Set $\boldsymbol{\Delta W}^{(l)} \coloneqq 0$ and $\boldsymbol{\Delta b}^{(l)} \coloneqq 0$.
2. For batch size $i = 1{:}m$
   a. Perform backward propagation to compute

   Weights: $\frac{\partial C(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{W}^{(l)}} = \frac{1}{m}\left(\boldsymbol{\delta}^{(l+1)} \boldsymbol{a}^{(l)^T}\right)$

   Biases: $\frac{\partial C(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{b}^{(l)}} = \frac{1}{m}\boldsymbol{\delta}^{(l+1)}$

   b. Store gradients

Weights: $\boldsymbol{\Delta W}^{(l)} := \boldsymbol{\Delta W}^{(l)} + \frac{\partial C(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{W}^{(l)}}$

Biases: $\boldsymbol{\Delta b}^{(l)} := \boldsymbol{\Delta b}^{(l)} + \frac{\partial C(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{b}^{(l)}}$

3. Update the weights

   Weights: $\boldsymbol{W}^{(l)} := \boldsymbol{W}^{(l)} - \eta\left(\frac{1}{m}\left(\boldsymbol{\Delta W}^{(l)} + \lambda \boldsymbol{W}^{(l)}\right)\right)$

   Biases: $\boldsymbol{b}^{(l)} := \boldsymbol{b}^{(l)} - \eta\left(\frac{1}{m}\boldsymbol{\Delta b}^{(l)}\right)$

Repeat the steps above in order to reduce the cost function. An algorithm for gradient descent is presented in Appendix B.

In order to evaluate the model performance, i.e. how well it predict the training data, we turn towards metrics such as MSE and the R^2 score. The MSE is defined as

$$\mathrm{MSE}(y, \tilde{f}(x)) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \tilde{f}(x_i)\right)^2 \tag{24}$$

and the R^2 score is defined as

$$\mathrm{R}^2(y, \tilde{f}(x)) = 1 - \frac{\sum_{i=1}^{n}\left(y_i - \tilde{f}(x_i)\right)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{25}$$

where y is the true model, $\tilde{f}(x)$ is the predicted model and $\bar{y}$ is the mean of the true model. However, for model evaluation in machine learning problems, the prediction error (MSE) measured on training data is not a good measure for model assessment (Hastie et al, 2001). There exists a couple of statistical tools, which provide model assessment in machine learning problems, known as resampling methods. The bootstrap method and the K-fold cross validation are the most common for model assessment in machine learning problems (James et al, 2013).

Suppose now that we split the jointly distributed set of measurements $\{x, y\}$ into a training set and an independent test set. The basic idea in bootstrapping is to randomly draw samples from your training set with replacement (i.e. the same sample can be drawn more than once), let's say B times, and refit the model for each bootstrap sample, and then apply the same set of predictors to test set. For bootstrap, the expected prediction error function reads (Hastie et al, 2001)

$$\mathrm{E}_{\mathrm{boot}}\left[\mathrm{L}\left(y_i, \tilde{f}^{*b}(x_i)\right)\right] = \frac{1}{B}\frac{1}{N}\sum_{i=1}^{B}\sum_{i=1}^{N}\mathrm{L}\left(y_i, \tilde{f}^{*b}(x_i)\right) \tag{26}$$

where $\mathrm{L}\left(y_i, \tilde{f}^{*b}(x_i)\right) = \left(y_i - \tilde{f}^{*b}(x_i)\right)^2$, i.e. the squared error between the model y_i and the predicted model $\tilde{f}^{*b}(x_i)$. Assessing the model performance is done by computing the average of the expected prediction error on both the training set and the test set at each point $\boldsymbol{x_i}$. The purpose of splitting the data into a training set and an independent test set, is that the prediction error for the training set tends to decrease using higher model complexity, but for the test set, the prediction error tends to have a turning point where the prediction error increases. As prediction increases for the test set using higher model complexity, the model is overfitting, i.e. the training model is not able to generalize the predictions. This is an inevitable consequence, and is related to the variance-bias tradeoff of the prediction, which we can derive from expected prediction error

$$\begin{aligned} E\left[L\left(\boldsymbol{y}, \tilde{f}(\boldsymbol{x})\right)\right] &= E[\boldsymbol{y}^2] - E[\boldsymbol{y}]^2 + E\left[\tilde{f}(\boldsymbol{x})^2\right] - E\left[\tilde{f}(\boldsymbol{x})\right]^2 \\ &+ E\left[\tilde{f}(\boldsymbol{x})\right]^2 - 2E\left[\tilde{f}(\boldsymbol{x})\boldsymbol{y}\right] + E[\boldsymbol{y}]^2 \\ &= \sigma^2 + \mathrm{Var}\left(\tilde{f}(\boldsymbol{y})\right) + \mathrm{Bias}^2\left(\tilde{f}(\boldsymbol{x}), \boldsymbol{y}\right) \end{aligned} \tag{27}$$

where we have used the following relationships

$$\sigma^2 = \mathrm{Var}(\boldsymbol{y}) = \mathrm{E}[\boldsymbol{y}^2] - \mathrm{E}[\boldsymbol{y}]^2$$

$$\mathrm{Var}\left(\tilde{f}(\boldsymbol{x})\right) = \mathrm{E}\left[\tilde{f}(\boldsymbol{x})^2\right] - \mathrm{E}\left[\tilde{f}(\boldsymbol{x})\right]^2$$

$$\mathrm{Bias}^2\left(\tilde{f}(\boldsymbol{x}), \boldsymbol{y}\right) = \left(\mathrm{E}\left[\tilde{f}(\boldsymbol{x})\right] - \boldsymbol{y}\right)^2 = \mathrm{E}\left[\tilde{f}(\boldsymbol{x})\right]^2 - 2\mathrm{E}\left[\tilde{f}(\boldsymbol{x})\boldsymbol{y}\right] + E[\boldsymbol{y}]^2$$

The first term is the irreducible error, which contain the variance of the input data. The bootstrap method was implemented using functions provided by Scikit learn, to split the model and evaluate the model performance.

Results and discussion

1D Ising: Regression analysis

In order to study a binary system, i.e a system which is only defined by two values, such as ± 1 or 0 and 1, using both regression analysis and classification, we will turn our focus towards a commonly used model known as the Ising model. The Hamiltonian function for the one-dimensional Ising model reads (Mehta et al, 2018)

$$H = -J \sum_{j=1}^{40} s_j s_{j+1} \tag{28}$$

where $s_j \in \{-1,1\}$, n is the total number of spins, J is known as the coupling constant. The coupling constant expresses the strength of the interaction between the adjacent spins. In physics the Hamiltonian is interpreted as state of energy, denoted $E[s]$, where in this context our aim is to learn the coupling constant J from the training data $D = (\{s_j\}_{j=1}^{40}, E[s_j])$. Before we start, we will rephrase the 1D equation (28) to include every pairwise interaction between all spin variables. The equation then reads (Mehta et al, 2018)

$$E[s^i] = -\sum_{j=1}^{40}\sum_{k=1}^{40} J_{j,k} s_j^i s_k^i \tag{29}$$

where $J_{j,k}$ is now a matrix of coupling constants. In order to train the $J_{j,k}$ we will define equation XXX in a form which is more familiar from a linear regression point

$$E[s^i] = \boldsymbol{X}^i \boldsymbol{J} \tag{30}$$

Where

$$\boldsymbol{J} = [J_{1,1}, J_{2,1}, \dots, J_{40,1}, J_{1,2}, J_{2,2}, \dots, J_{40,2}, \dots, J_{40,40}]^T \tag{31}$$

giving a $(40^2 \times 1)$-dimensional vector of the coupling constants corresponding to all the possible spin interactions given in the design matrix $\boldsymbol{X}^i$ $i = 1,2, \dots, 600$. The design matrix is now given by a (600×40^2)-dimensional matrix

$$\boldsymbol{X}^i = \begin{bmatrix} | & | & | & | & | & | & | & | & | & | \\ S_1^i S_1^i & S_2^i S_1^i & \dots & S_{40}^i S_1^i & S_1^i S_2^i & S_2^i S_2^i & \dots & S_N^i S_2^i & \dots & S_N^i S_N^i \\ | & | & | & | & | & | & | & | & | & | \end{bmatrix} \tag{32}$$

For the Ising model, the design matrix is singular, which force us to turn to other methods than the OLS estimator to provide a solution to the problem. In this case, we introduce the SVD for both the OLS and Ridge estimations. The SVD approach for OLS and Ridge is computed using the algorithms presented in Appendix A, and the Lasso estimator was computed by the "least_square_ridge_svd" approach within the scikitlearn package. For model assessment, the training data were randomly split into roughly 70/30 training-test set ratio, and find the best-fit model based on the prediction error from the test set, which gives the cross validation approach for model assessment. Results from the computed prediction error (MSE) and R^2 score using OLS, Ridge and Lasso are presented in Figure 3 and Figure 4 respectively. By considering the MSE and R^2 score for the test set, we observe that the Lasso estimation gives a better fit for the OLS and Ridge prediction error. This we believe is due to the sparsity of the coupling constant and the sparsity given by the Lasso estimators, which will therefore be favored over Ridge and OLS. The predicted estimators from OLS, Ridge and Lasso, by

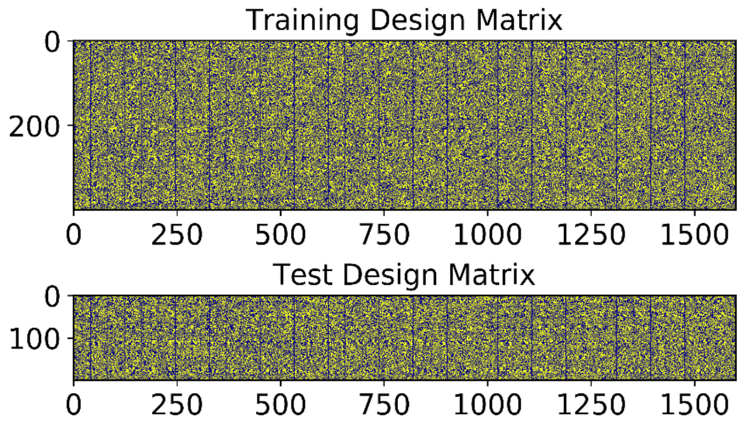


Figure 4: *The training- and test design matrices $\boldsymbol{X}_{train}, \boldsymbol{X}_{test} \in \{-1,1\}$ as explained in (23), and used for training the predictors.*

using a range of different values of the penalty parameter λ are presented in Figure 5, showing that the Lasso prediction resembles the ground truth very well.

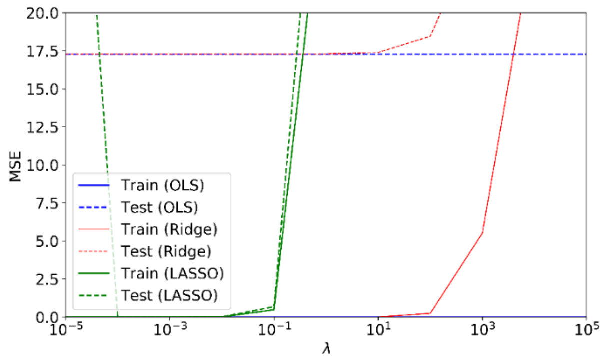


***Figure 5:** Computed prediction error (MSE) wrt model complexity using OLS, Ridge (L_2) and Lasso (L_1) estimation. For the OLS solution the Ridge penalty parameter was set to $\lambda_2 = 0$, giving a constant prediction error. Considering the test set, the prediction error from Lasso gives the overall best score but shows overfitting issues at penalty values $\lambda_1 \leq 10^{-4}$.*

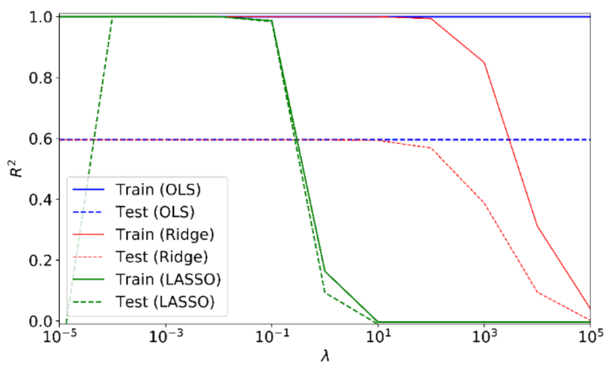


***Figure 6:** Computed R^2 score wrt model complexity using OLS, Ridge (L_2) and Lasso (L_1) estimation. The R^2 score from Lasso gives the overall best score considering the test set, but shows overfitting issues at the lowest penalty values, $\lambda \leq 10^{-4}$. We believe that the improved score for Lasso estimation is due to the sparse nature of the L_1 regularization, and since the coupling constants in this example is sparse, it will favor Lasso over Ridge and OLS.*

In order to assess our model using a different approach, we now turn to the bootstrap resampling technique, as explained in the theory section for model evaluation, and find the best-fit model based on the prediction error from the test set. In this example, the data was split using an 80/20 relationship on the training- and test set and used 200 bootstrap iteration.

The prediction error, variance and bias decomposition from the Ridge estimations and Lasso estimations, using bootstrap resampling, are presented in Figure 6 and Figure 7 respectively. The Lasso estimation shows significantly better results compared to the Ridge. The prediction error decreases as model complexity increases considering both the training and test set. For the Lasso estimation, the prediction error shows a turning point at $\lambda_1 = 10^{-2}$, which corresponds to the best fit. We see that variance increases as model complexity increases but is more or less constant from Ridge penalty values $\lambda_2 \leq 10^{2}$. We plot the results derived from the bootstrap predicted best fit in Figure 8, where we see that the predicted result, using Lasso with a penalty parameter of $\lambda_2 \leq 10^{-2}$, fits the ground truth perfectly.

For the Ising-model regression problem, showing a discrete nature in both its design matrix and its estimators, it seems likely that the L_1 regression approach would give the best model. This is possibly due to the Lasso estimators sparsity, leading to zeros within its estimators $\left(\tilde{\beta}_j(\lambda_1) = 0\right)_{Lasso}$, unlike the Ridge estimator, which can have estimators close to zero, but not equal to zero.

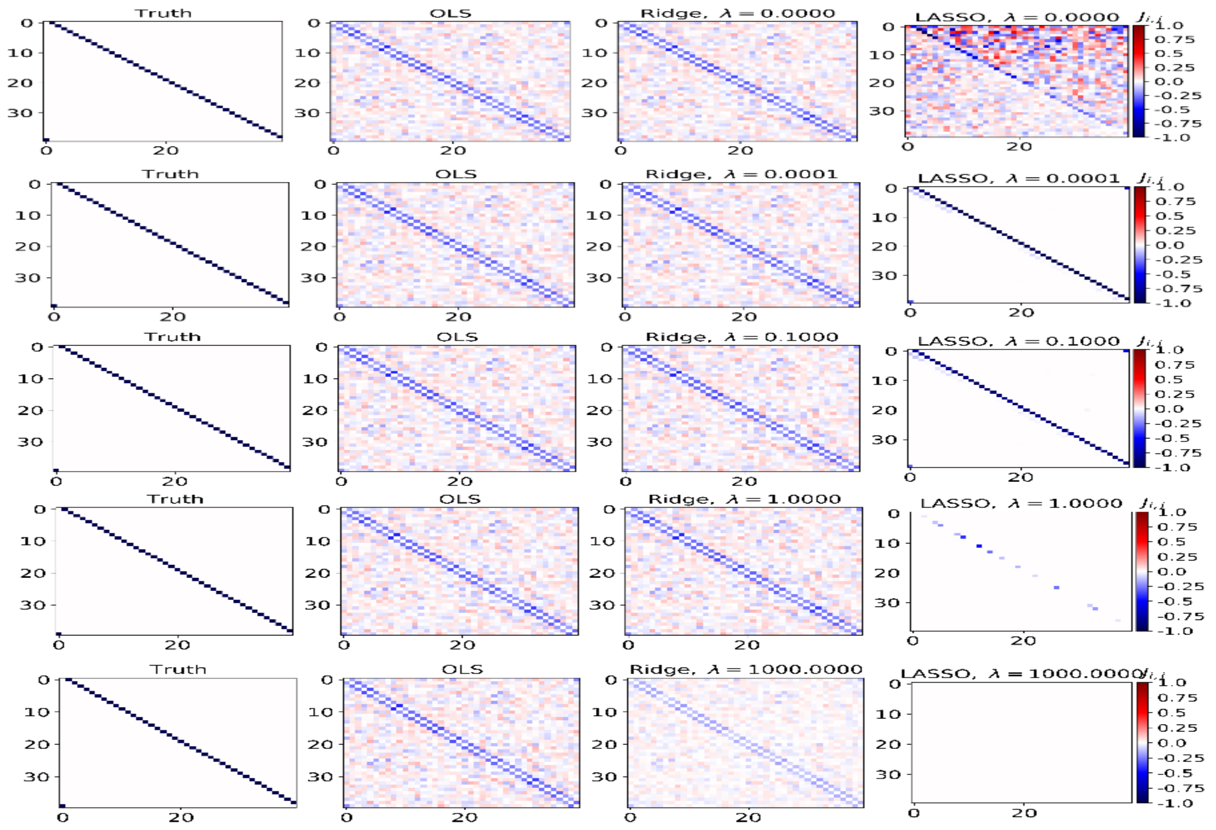
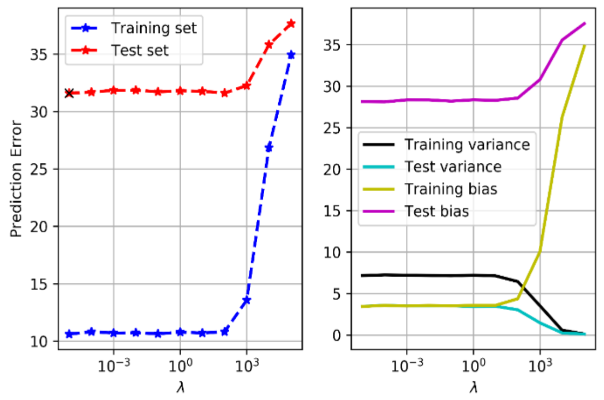


Figure 5: *Computed OLS, Ridge and Lasso estimators wrt model complexity. The Lasso estimator using a penalty parameter $\lambda \leq 0.1$ seems to give a good estimation. However, as we see in Figure 4, $\lambda \leq 10^{-4}$ gives overfitting and high prediction error.*



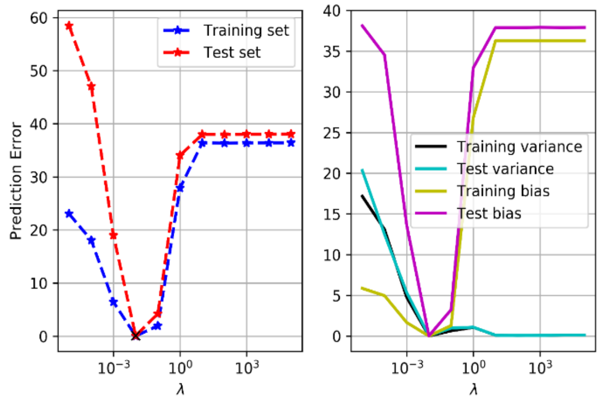


Figure 6: *Computed prediction error (MSE) (left) for training set and test set wrt to model complexity using Ridge. The training/test split was set to 80/20 and using 200 bootstrap iterations. The prediction error decreases as model complexity increases considering both the training and test set. We see that variance increases as model complexity increases but is merely constant from penalty values $\lambda \leq 10^2$. The bias shows similar trend but the bias decrease with model complexity. Considering the test set, the example shows a best fit for $\lambda \leq 10^{-5}$ (black cross).*

Figure 7: *Computed prediction error (MSE) (left) for training set and test set wrt to model complexity using Lasso. The training/test split was set to 80/20 and using 200 bootstrap iterations. The prediction error decreases as model complexity increases considering both the training and test set. However, its shows a turning point for $\lambda \leq 10^{-2}$, where prediction error increases again, due to overfitting. Considering the test set, the example shows a best fit for $\lambda \leq 10^{-2}$ (black cross).*

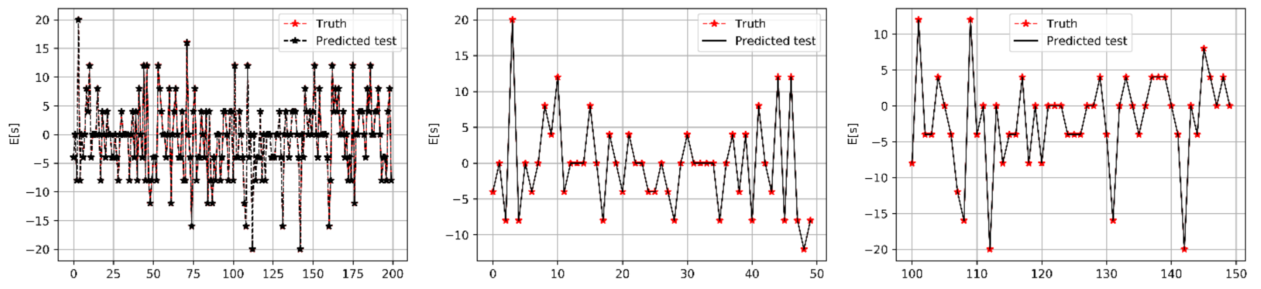


Figure 8: *Ground truth and predicted result, using L_1 estimation with a penalty parameter of $\lambda_1 = 10^{-2}$. We see that the trained function fits the data perfectly.*

1D Ising: Neural network regression

In this example, we implement a neural network with forward propagation, backpropagation and gradient descent optimizer, as explained in the theory section. The neural network was written as a two-layer network with L_2 regularization, and all activation functions (Relu, Sigmoid and Tanh) implemented with their respective derivatives (see Appendix B). When choosing an activation function for training, the function applies for all layers (including output). This is however a consequence from our implementation. It is possible to have different activation functions for different layers, and in the regression problem, it is also common to exclude the activation from the output layer.

In order to train the regressor, we use the same training data, and train/test split (approximately 70/30) as used in the linear regression example presented in the previous section. As the output layer for our neural net regressor includes an activation function, the training- and test set targets was normalized and reformed to better fit the outputs. For Sigmoid- and Relu activations we reformed the targets to

$$\boldsymbol{y} = \frac{\left(\frac{\boldsymbol{y}}{\max|\mathbf{y}|}\right) + 1}{2}$$

giving training targets for Sigmoid and Relu $\boldsymbol{y} \in [0,1]$. For Tanh we used the following transformation

$$\boldsymbol{y} = \frac{\boldsymbol{y}}{\max|\mathbf{y}|}$$

giving training targets for Tanh $\boldsymbol{y} \in [-1,1]$. The regressors weights and biases were trained using a network with two hidden layers including 20 neurons in each layer. The computed prediction error (MSE) and R^2 score using Relu, Sigmoid and Tanh activation functions with respect to to model complexity (represented by L_2 regularization) is presented in Figure 8 and Figure 9 respectively.

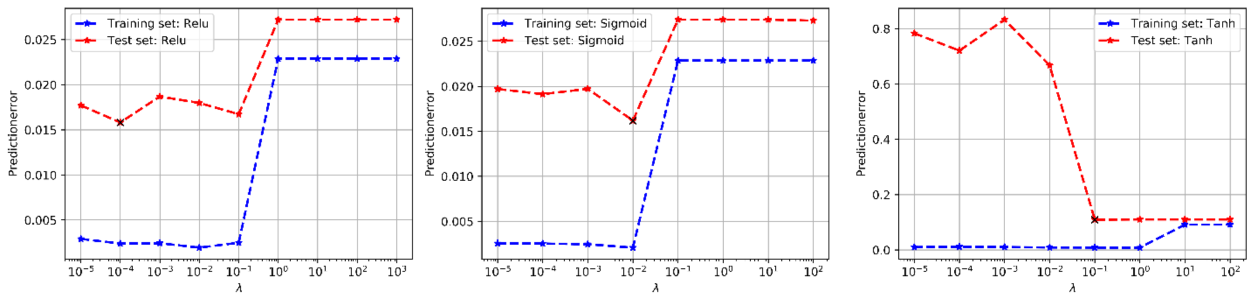


Figure 8: *Computed prediction error (MSE) using Relu (left), Sigmoid (middle) and Tanh (right) activation functions wrt to model complexity, for both training set and test set. The model complexity is represented by L_2 regularization in this case. The classifier was trained using train/test split was set to approximately 70/30, two hidden layers with 20 neurons in each. The training data targets was reformed to better resemble the output activation function characteristics, giving target values for Tanh $\boldsymbol{y} \in [-1,1]$ and $\boldsymbol{y} \in [0,1]$ for Sigmoid and Relu. Considering the test set, the example shows a best fit for Relu at $\lambda \leq 10^{-4}$ and at $\lambda \leq 10^{-2}$ for Sigmoid. For Tanh we have a best fit at $\lambda \leq 10^{-1}$, though the parameter does not show significantly larger prediction error for $\lambda \geq 10^{-1}$. Compared to the linear regression solution using L_1 regularization, this neural network gives a prediction error which is much higher. In addition, it was computationally much more expensive (longer time to get a good fit for the model).*

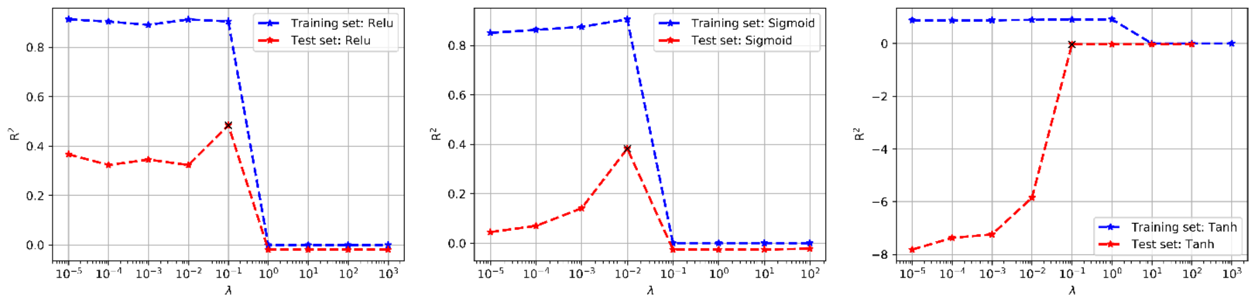


Figure 9: *Computed prediction error (MSE) using Relu (left), Sigmoid (middle) and Tanh (right) activation functions wrt to model complexity, for both training set and test set. The model complexity is only represented by L_2 regularization in this case. Compared to the linear regression solution using L_1 regularization, this neural network gives a prediction error which is much higher. In addition, it was computationally much more expensive (longer time to get a good fit for the model).*

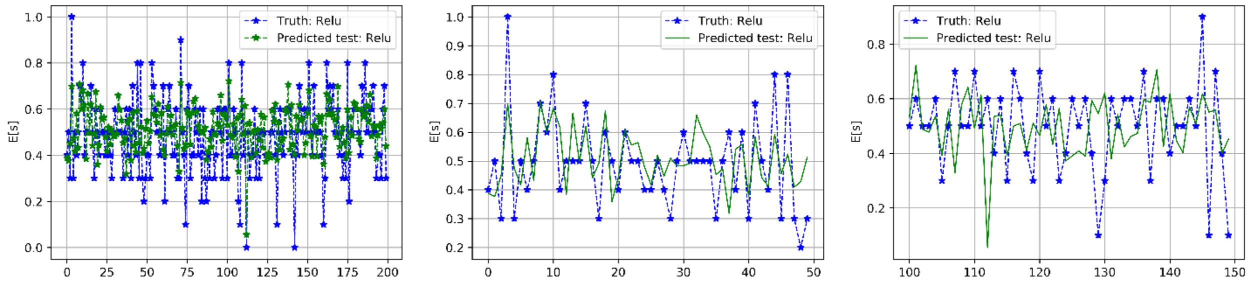


Figure 10: *Ground truth and predicted result, using neural network with Relu activations and L_2 regularization using a penalty parameter of $\lambda_1 = 10^{-4}$. We see that the trained function does not fit the data perfectly as we had with the Lasso estimation using linear regression as presented in Figure 8.*

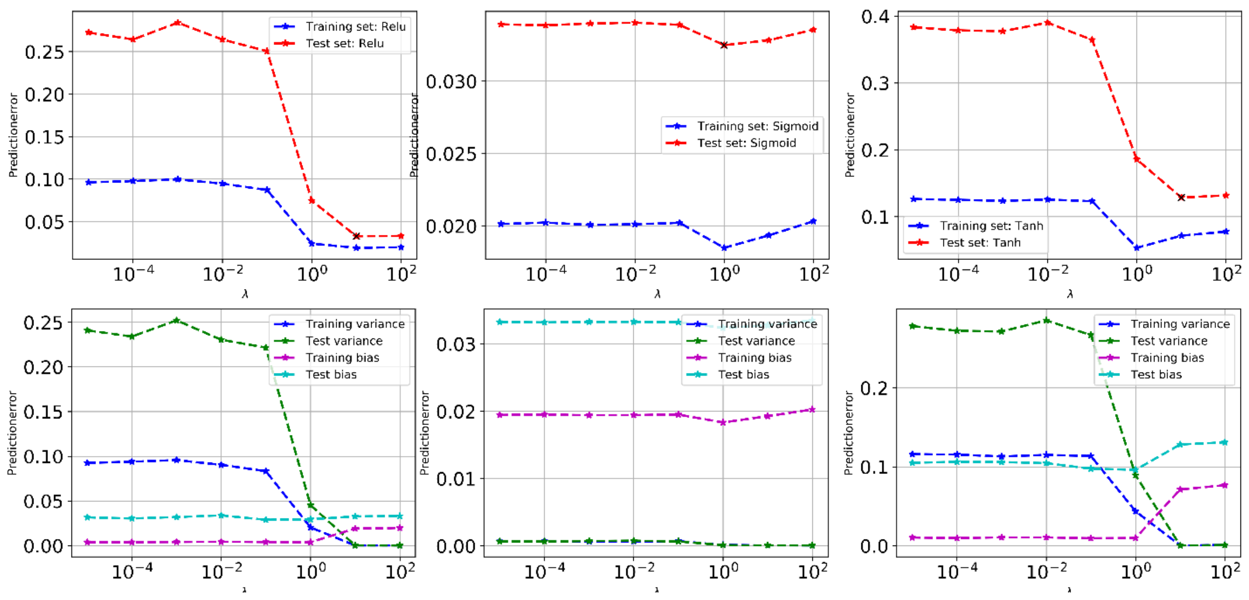


Figure 11: *Results from bootstrap resampling with 100 bootstraps, including computed prediction error (MSE) using Relu (left), Sigmoid (middle) and Tanh (right) activation functions wrt to model complexity, for both training set and test set. The model complexity is represented by L_2 regularization in this case. Compared to the linear regression solution using the Lasso estimation, the neural network gives a prediction error, which is much higher. In addition, it was computationally more expensive. From the variance and bias tradeoff, especially for the Relu and Tanh activations, the variance is the key factor for high prediction error at low penalty parameter values.*

We observed that the Lasso estimation for regression outperforms OLS, Ridge from the previous section, and compared to the neural network, for this type of problem. This relates both in prediction of energy states (Figure 8 and Figure 10), and estimator similarity. As mentioned, this possibly relates to the connection between the discrete nature of the model (only -1 and 1) and sparsity of the coupling constants, and the L_1 regularization leading to sparsity in its the estimators, which fits the modeled problem quite well. In addition, the linear regression approach using L_1 regularization have less computational cost compared to the neural network, and gives an explicit formulation and direct connection to the coupling constants, instead of just a function with weights and biases.

2D Ising: Logistic regression classification

In this section, we continue to study a binary system as in the regression analysis part, but now we turn our focus towards the 2D Ising model and the classification problem. The Hamiltonian function is somewhat different for the two-dimensional Ising model, and reads (Mehta et al, 2018)

$$H = -J \sum_{<ij>} s_i s_j , \quad s_i s_j \in \{-1,1\} \tag{33}$$

Which describes a 2D square lattice where $< ij >$ indicates a summation over the nearest neighbors. Our ultimate goal is to train a logistic regression classifier to see if we can classify the magnetic phase, and at which accuracy. The classification problem we will be studying is a binary classification problem, since we deal with only ordered or disordered phases. In the report we are not going to study the classification problem related to the magnetic phase at (or close to) the critical temperature$T_c/J = 2/\log(1+\sqrt{2}) \approx 2.26$. The dataset used in this report was downloaded from

https://physics.bu.edu/~pankajm/ML-Review-Datasets/isingMC/.

Data for classification: Ising2DFM_reSample_L40_T=All_labels.pkl, excluding the magnetic phases at the critical temperature. The training data, consists of k "images" $I^{(k)}$ defining a two-dimensional 40×40 square lattice, which corresponds to the phases we want to classify; disordered or ordered magnetic phases. The "images" were merged and transformed to a design matrix $\boldsymbol{x}$ where each individual "image" is structured as a row-vector

$$I^{(k)} = \begin{bmatrix} I_{1,1}^{(k)} & \cdots & I_{1,40}^{(k)} \\ \vdots & \ddots & \vdots \\ I_{40,1}^{(k)} & \cdots & I_{40,40}^{(k)} \end{bmatrix} \rightarrow \boldsymbol{x} = \begin{bmatrix} - & I_{i,j}^{(1)} & - \\ - & I_{i,j}^{(2)} & - \\ - & \vdots & - \\ - & I_{i,j}^{(k)} & - \end{bmatrix} \tag{34}$$

The target phases in $\boldsymbol{y}$ are structured as a column vector

$$\boldsymbol{y} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_k \end{bmatrix}, \qquad t_i \in \{0,1\}, i = 1,2,3, \dots, k \tag{35}$$

Figure 12 displays one ordered and one disordered phase, extracted from the design matrix $\boldsymbol{x}$. Before we train the classifier, the training data was separated into a training- and test set, with a train/test relationship of 80/20. As a first test, we trained the logistic classifier using L_2 regularization using only the train- and test set, with no statistical resampling method. As a second test we assess the model using bootstrap resampling, using both L_2 and L_1 regularization. We will distinguish between the regularization parameters of L_1 and L_2 by λ_1 and λ_2 respectively. The L_1 regularization leads to different estimators compared to L_2 regularization, as explained in the theory section. Due to sparse nature of the L_1, some of the estimators within $\widetilde{\boldsymbol{\beta}}_{Lasso}$ may be estimated to $\tilde{\beta}_j(\lambda_1) = 0$. This is in contrast to the L_2 estimators as $\tilde{\beta}_j(\lambda_2) \neq 0$ for all parameters in $\widetilde{\boldsymbol{\beta}}(\lambda_2)$.

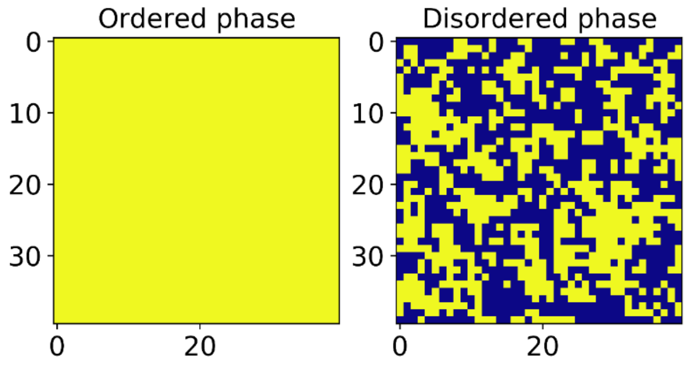


Figure 12: *A single training example $x^{(i)} \in \{-1, +1\}$ from the ordered magnetic phase and from the disordered magnetic phase, extracted from the training data.*

We trained the logistic classifier using batch gradient descent with batch size of 50, learning rate of $\eta = 0.05$ and by running 500 epochs. The logistic cost and gradient descent optimizer in this example (our own implementation) only includes L_2 regularization. The results are presented in Figure 13. The decision threshold for prediction was set to $\tilde{\mathbf{y}} = \begin{cases} 1 \; if \; \tilde{\mathbf{y}} \; \geq 0.5 \\ 0 \quad else \end{cases}$. The classifiers performance in relation to the test data shows low prediction accuracy, with a best fit of our model using a penalty parameter $\lambda_2 = 10^{-4}$ with an accuracy close to 55%.

We were not able to produce accuracy above 55% on our test set using our own implementation; testing different learning rate η, batch sizes and epochs. We could possibly improve the classifier by testing more testing to find an optimal training rate, batch size etc. However, as shown in the regression case, L_1 regularization could be a quick answer. The discrete characteristic of the Ising model, could possibly favor sparsity in its estimators $\widetilde{\boldsymbol{\beta}}$, also for the classification problem.

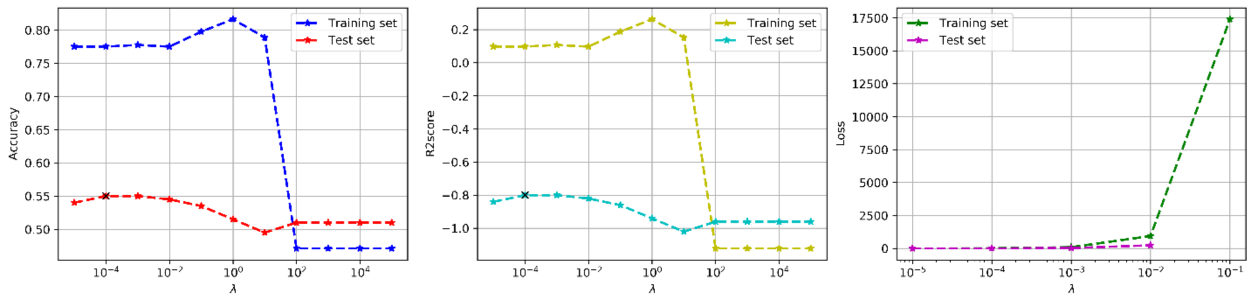


Figure 13: *Computed accuracy score, R^2 score and loss (logistic cost) wrt to model complexity λ_2, using logistic regression with L_2 regularization. The training data were split to a training- and test set with train/test split ratio of 0.8. The classifier was trained using batch gradient descent with batch size of 50, learning rate $\eta = 0.05$ and 500 epochs. The decision threshold for output probabilities from the logistic function was set to $\tilde{y}_{pred} = \begin{cases} 1\, if\ \tilde{y}\ \geq 0.5 \\ 0 \quad else \end{cases}$. The prediction of the test set shows a best fit using $\lambda_2 = 10^{-4}$. The trained classifier performance, is quite poor, showing low accuracy score for the test data. As we observed in the regression case, the Lasso estimation, or L_1 regularization outperformed the L_2 regularization prediction. In this case, we will also test the L_1 regularization, but using bootstrap resampling for model assessment.*

In order to compare the L_2 regularization and L_1 regularization, we now introduce the bootstrap resampling method for model assessment. In this case, the scikit-learn package linear_model.SGDClassifier was used, and with optimal learning rate. We trained and assessed the classifier using both L_2 and L_1 regularization with bootstrap resampling, as presented in Figure 14. The loss of the test set shows a best fit using L_2 at $\lambda_2 = 1$, and at $\lambda_1 = 10^{-2}$ for L_1 regularization. We see an increase in model accuracy by using L_1 compared to L_2, which could relate to the model characteristic and sparse nature of the L_1 regularization estimators, as we observed in the regression case. However, the accuracy score using the best-fit model only gave about 70% accuracy score, which is still a low accuracy.

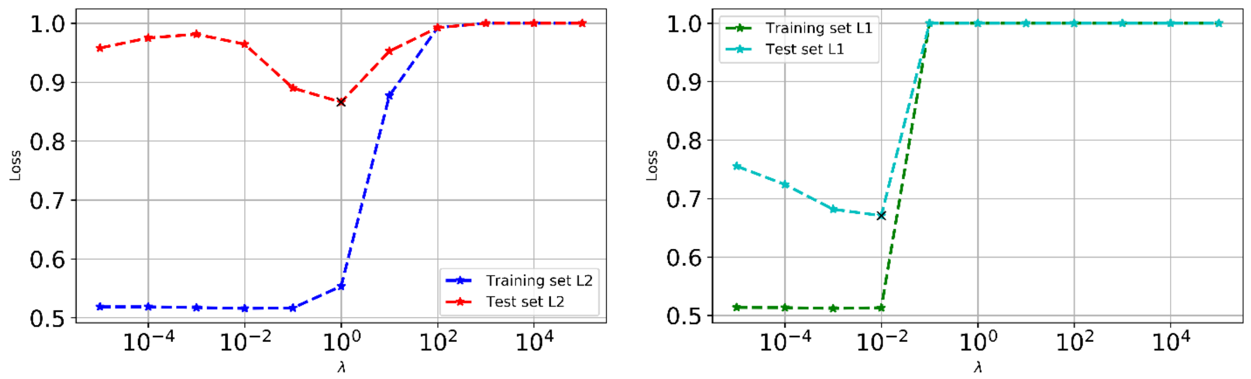


Figure 14: *Results from bootstrap resampling with 100 bootstraps, including computed logistic regression loss wrt to model complexity with L_2 regularization (left) and with L_1 regularization (right) for training and test set. The decision threshold for our prediction was set to $\tilde{y} = \begin{cases} 1\, if\ \tilde{y}\ \geq 0.5 \\ 0 \quad else \end{cases}$. The loss of the test set shows a best fit using L_2 at $\lambda_2 = 1$, and at $\lambda_1 = 10^{-2}$ for L_1 regularization. However, we see an increase in model accuracy by using L_1 compared to L_2, which could be related to the model characteristic and sparsity of the L_1 estimators, as we observed in the regression case. However, the accuracy score using the best fit model $\lambda_1 = 10^{-2}$ only gave about 70% accuracy score, but nevertheless it outperforms the L_2 regularization.*

Neural network classification

In this example, we implement a neural network with forward propagation, backpropagation and gradient descent optimizer, as explained in the theory section, and as done in the regression analysis section but in this case as a classification problem. The neural network was written as a two-layer network with L_2 regularization, and all activation functions (Relu, Sigmoid and Tanh) implemented with their respective derivatives (see Appendix B).

In order to train the classifier, we use the same training data, and train/test split of 80/20 as used in the logistic classification example presented in the previous section. No normalization was applied to the target values, so the target values reads as shown in (26) for all classification examples, i.e. $\boldsymbol{y} \in \{-1,1\}$. The classifiers weights and biases were trained using a network with two hidden layers including 20 neurons in each layer. The computed accuracy score, R^2 score and prediction error (MSE) using Relu, Sigmoid and Tanh activation functions with respect to to model complexity (represented by L_2 regularization) is presented in Figure 15, Figure 16 and Figure 17 respectively. We observe that the neural network is able to fit the test data almost perfectly, by using Relu and Sigmoid activation, and shows that it clearly outperforms the logistic classifier. However, we were not able to make the Tanh function perform better than the L_1 regularized logistic classifier, which gave around 65% accuracy score compared to 70% from the logistic classifier.

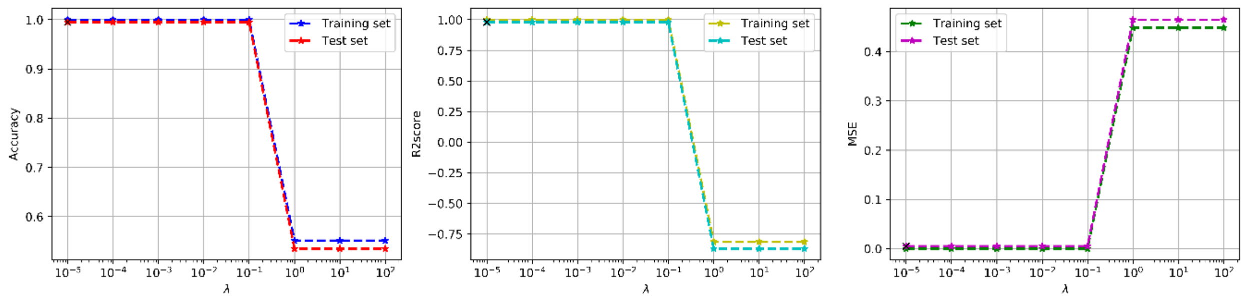


Figure 15: *Computed neural network model accuracy, R^2 score and prediction error (MSE) wrt to model complexity, which is represented by L_2 regularization parameter λ_2. The neural network classifier setup includes Sigmoid activation function for all layers, 2 hidden layers and 20 neurons in each hidden layer. The classifier was trained with batch gradient descent with batch size of 50, learning rate $\eta = 1$ and running 500 epochs. The decision threshold for our prediction was set to $\tilde{y} = \begin{cases} 1 \; if \; \tilde{y} \geq 0.5 \\ 0 \quad else \end{cases}$. We see that the neural network is able to fit the test data almost perfectly, which shows that it clearly outperforms the logistic classifier.*

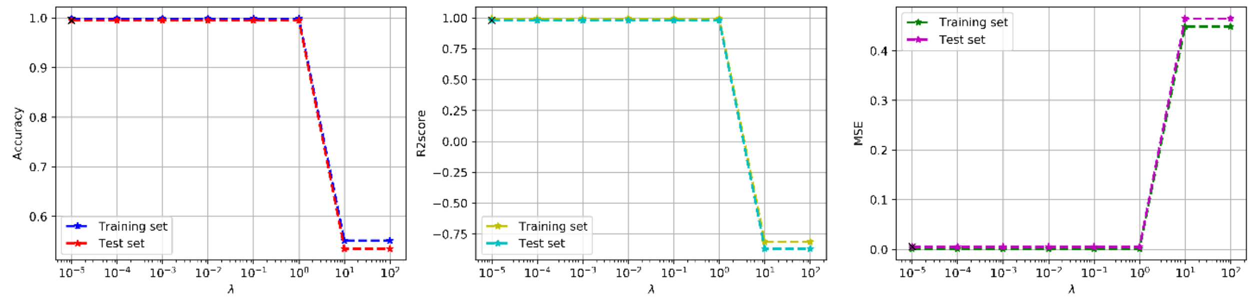


Figure 16: *Computed neural network model accuracy, R^2 score and prediction error (MSE) wrt to model complexity, which is represented by L_2 regularization parameter λ_2. The neural network classifier setup includes Relu activation function for all layers, 2 hidden layers and 20 neurons in each hidden layer. The classifier was trained with batch gradient descent with batch size of 50, learning rate $\eta = 0.1$ and running 500 epochs. The decision threshold for our prediction was set to $\tilde{y} = \begin{cases} 1\, if\ \tilde{y}\ \geq 0.5 \\ 0 \quad else \end{cases}$. We see that the neural network is able to fit the test data almost perfectly, which shows that it clearly outperforms the logistic classifier.*

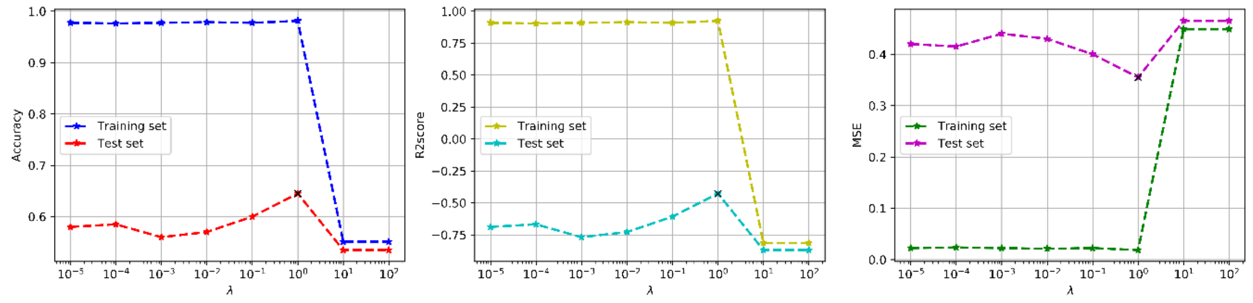


Figure 17: *Computed neural network model accuracy, R^2 score and prediction error (MSE) wrt to model complexity, represented by L_2 regularization parameter. The neural network classifier setup includes Tanh activation function for all layers, 2 hidden layers and 20 neurons in each hidden layer. The classifier were trained with batch gradient descent with batch size of 50, learning rate $\eta = 1$ and running 500 epochs. The decision threshold for our prediction was set to $\tilde{y} = \begin{cases} 1\, if\ \tilde{y}\ \geq 0.5 \\ 0 \quad else \end{cases}$. The classifier is not able to outperform the logistic classifier, using L_1 regularization, by using Tanh activation in all layers.*

Conclusions

In this report, we have tested different methods for training a function to predict energy states and estimating the coupling constant for the 1D Ising model. These methods includes linear regression using OLS, Ridge- and Lasso estimation, and by implementing SVD in order to avoid the singularity problem of the design matrix. In addition, we also implement a neural network with two hidden layers in order to compare its performance to the linear regression approach, on energy state predictions. What we found is that using the Lasso estimation for regression outperforms all other methods, OLS, Ridge and neural network, for this type of problem. This relates both in prediction of energy states, and estimator similarity. We believe the reason to be of the connection between the discrete nature of the model (only -1 and 1) and coupling constants, and the L_1 regularization leading to sparsity of the estimator, which fits the modeled problem quite well. In addition, the linear regression approach using L_1 regularization have less computational cost compared to the neural network, and gives an explicit formulation and direct connection to the coupling constants, instead of just a function with weights and biases.

We have also tested different methods for training a classifier the magnetic phase of the 2D Ising model.   For the classification problem, the L_2 regularized neural network outperforms the logistic classifier, even while using L_1 regularization. The neural network gives an almost perfect fit with respect to both the training- and test set, implying a good generalization of the classifiers prediction performance. However, the Tanh activation did not give good prediction accuracy, and was even outperformed by the L_1 regularized logistic classifier. The reason why the logistic classifier does not work well for this problem, using either L_1 or L_2 regularization, is not simple to answer. It could be implementation related (our own logistic classifier is not optimized) or other.

Due to limited time, we have not tested neural network performance on the regression problem using L_1 regularization, or for the classification problem. This could be implemented in project 3.

References

Hastie, T., Tibshirani, R., Friedman, J. (2001). *The Elements of Statistical Learning*. New York, NY, USA: Springer New York Inc.

Hjorth-Jensen, M. (2018). *Data Analysis and Machine Learning: Logistic Regression.* https://compphysics.github.io/MachineLearning/doc/pub/LogReg/pdf/LogReg-minted.pdf

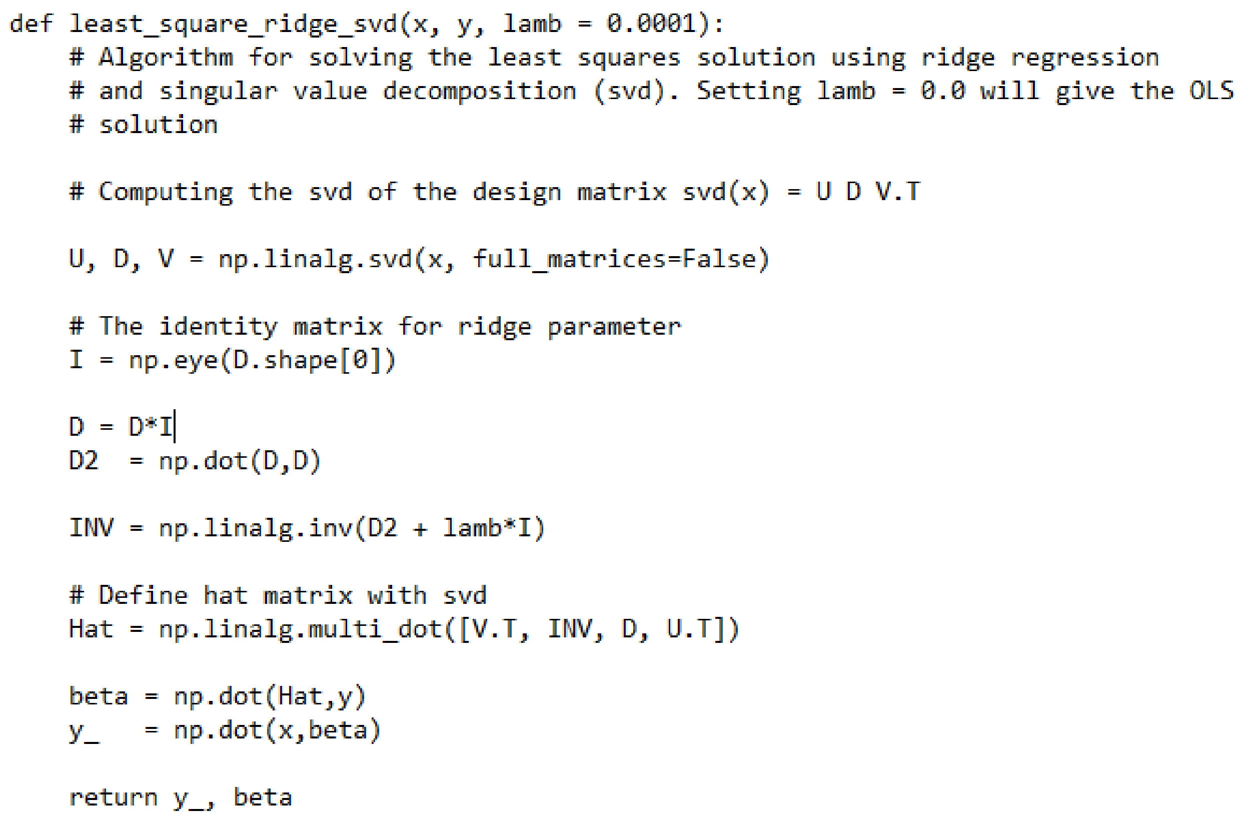
Ng, A. and Katanforoosh, K. (2018). *CS229 Lecture Notes: Deep Learning.* http://cs229.stanford.edu/notes/cs229-notes-deep_learning.pdf.

Mehta, P., Bukov, M., Wang, C., Day, A., Richardson, C., Fisher, C. K., Schwab, D. J., (2018). *A high-bias, low-variance introduction to Machine Learning for physicists,* arXiv:1803.08823

van Wieringen W. N. (2015). *Lecture notes on ridge regression*, arXiv:1509.09169

Appendix A

Algorithm for computing the ridge and OLS solution by using SVD:

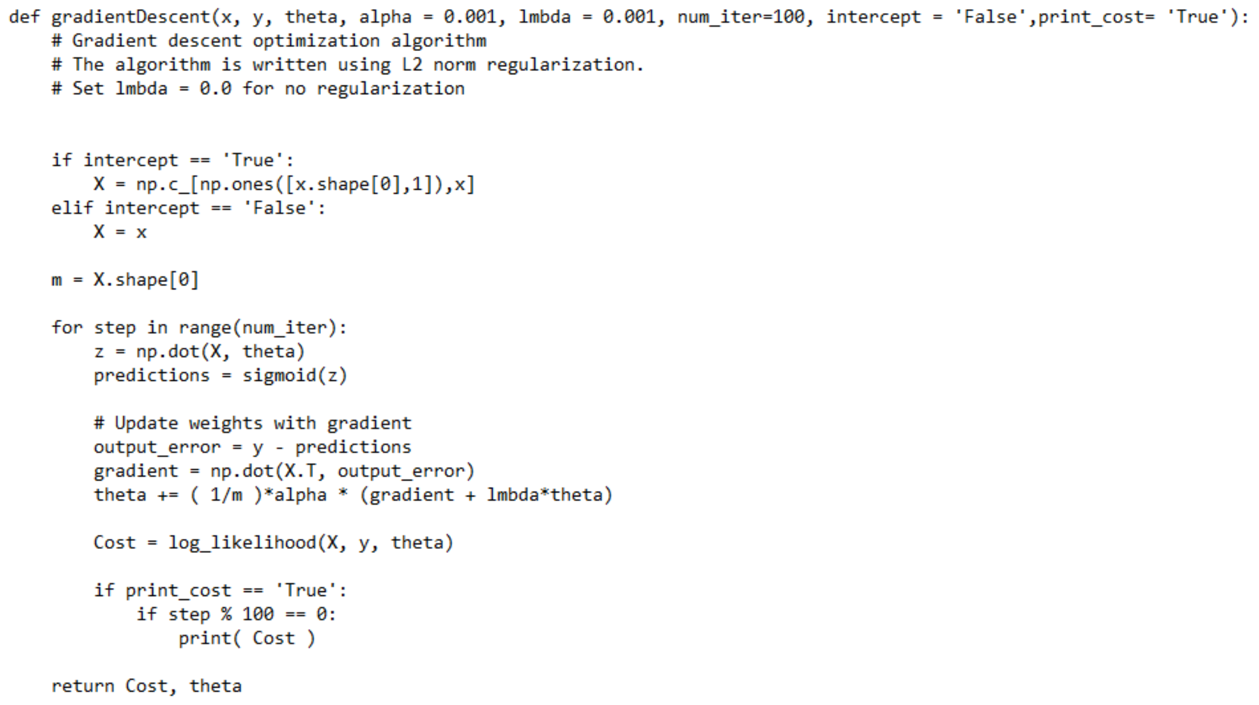


```
def least_square_ridge_svd(x, y, lamb = 0.0001):
    # Algorithm for solving the least squares solution using ridge regression
    # and singular value decomposition (svd). Setting lamb = 0.0 will give the OLS
    # solution

    # Computing the svd of the design matrix svd(x) = U D V.T

    U, D, V = np.linalg.svd(x, full_matrices=False)

    # The identity matrix for ridge parameter
    I = np.eye(D.shape[0])

    D = D*I
    D2  = np.dot(D,D)

    INV = np.linalg.inv(D2 + lamb*I)

    # Define hat matrix with svd
    Hat = np.linalg.multi_dot([V.T, INV, D, U.T])

    beta = np.dot(Hat,y)
    y_   = np.dot(x,beta)

    return y_, beta
```

Algorithm for computing the logistic cost:

```
def logistic_reg_cost(x, y, theta, lmbda = 0.01, intercept = 'False'):
    # Cost function for logistic regression
    # The algorithm is written using L2 norm regularization. Set
    # lmbda = 0.0 for no regularization

    m = len(y)

    if intercept == 'True':
        X = np.c_[np.ones([x.shape[0],1]),x]
    elif intercept == 'False':
        X = x

    # Compute the linear function
    z = X.dot(theta)
    # Compute the non-linear function
    g = sigmoid(z)

    # Regularization term
    RegCost = (lmbda/2*m)*np.transpose( theta ).dot( theta )
    # Logistic regression cost function
    Cost    = ( 1/m )*( - np.transpose( y ).dot( np.log( g ) )
              - np.transpose( ( 1 - y ) ).dot( np.log( 1 - g ) ) ) + RegCost

    return Cost
```

Algorithm for updating estimators through gradient descent:



```
def gradientDescent(x, y, theta, alpha = 0.001, lmbda = 0.001, num_iter=100, intercept = 'False',print_cost= 'True'):
    # Gradient descent optimization algorithm
    # The algorithm is written using L2 norm regularization.
    # Set lmbda = 0.0 for no regularization


    if intercept == 'True':
        X = np.c_[np.ones([x.shape[0],1]),x]
    elif intercept == 'False':
        X = x

    m = X.shape[0]

    for step in range(num_iter):
        z = np.dot(X, theta)
        predictions = sigmoid(z)

        # Update weights with gradient
        output_error = y - predictions
        gradient = np.dot(X.T, output_error)
        theta += ( 1/m )*alpha * (gradient + lmbda*theta)

        Cost = log_likelihood(X, y, theta)

        if print_cost == 'True':
            if step % 100 == 0:
                print( Cost )

    return Cost, theta
```

Appendix B

Algorithm for forward propagation with two layers:

```
def forwardProp(X,W1,W2,W3,bias1,bias2,bias3,activation = 'sigmoid'):
    # Forward propagation algorithm for neural network

    a1 = X.copy()
    # Hidden layers
    z2 = W1.dot(a1) + bias1
    if activation == 'sigmoid':
        a2 = sigmoid(z2)
    if activation == 'tanh':
        a2 = tanh(z2)
    if activation == 'Relu':
        a2 = Relu(z2)

    z3 = W2.dot(a2) + bias2
    if activation == 'sigmoid':
        a3 = sigmoid(z3)
    if activation == 'tanh':
        a3 = tanh(z3)
    if activation == 'Relu':
        a3 = Relu(z3)

    # Output layer
    z4 = W3.dot(a3) + bias3
    if activation == 'sigmoid':
        y_pred  = sigmoid(z4)
    if activation == 'tanh':
        y_pred  = tanh(z4)
    if activation == 'Relu':
        y_pred  = Relu(z4)


    return y_pred,z4,z3,z2,a3,a2,a1
```

Algorithm for backward propagation with two layers:

```
def costGrad(X,y,W1,W2,W3,bias1,bias2,bias3,activation = 'sigmoid'):
    # Backward propagation algorithm for neural network

    # Do forward propagation
    y_pred,z4,z3,z2,a3,a2,a1 = forwardProp(X,W1,W2,W3,bias1,bias2,bias3,activation)
    # Do backward propagation
    if activation == 'sigmoid':
        delta4 = -(y - y_pred)
        delta3 = np.multiply( np.dot( W3.T , delta4 ) , sigmoidGradient(z3) )
        delta2 = np.multiply( np.dot( W2.T , delta3 ) , sigmoidGradient(z2) )

    if activation == 'tanh':
        delta4 = -(y - y_pred)
        delta3 = np.multiply( np.dot( W3.T , delta4 ) , tanhGradient(z3) )
        delta2 = np.multiply( np.dot( W2.T , delta3 ) , tanhGradient(z2) )

    if activation == 'Relu':
        delta4 = -(y - y_pred)
        delta3 = np.multiply( np.dot( W3.T , delta4 ) , ReluGradient(z3) )
        delta2 = np.multiply( np.dot( W2.T , delta3 ) , ReluGradient(z2) )
    # Partial derivatives of cost function wrt weights and biases
    dCdW3  = np.dot( delta4 , a3.T)
    dCdW2  = np.dot( delta3 , a2.T)
    dCdW1  = np.dot( delta2 , a1.T)
    dCdb3 = np.sum(delta4,axis=1,keepdims=True)
    dCdb2 = np.sum(delta3,axis=1,keepdims=True)
    dCdb1 = np.sum(delta2,axis=1,keepdims=True)

    return dCdW3, dCdW2, dCdW1, dCdb3, dCdb2, dCdb1
```

Algorithm for gradient descent optimizer for neural network with two layers:

```
def gradientDescentOptimizer(X,params, grads,eta=0.001,lmbda=0.001):
    # Gradient descent optimizer for two layer neural network
    m = X.shape[0]

    # Parameters to update
    W1 = params[0]
    W2 = params[1]
    W3 = params[2]
    bias1 = params[3]
    bias2 = params[4]
    bias3 = params[5]

    dCdW1 = grads[0]
    dCdW2 = grads[1]
    dCdW3 = grads[2]

    dCdb1 = grads[3]
    dCdb2 = grads[4]
    dCdb3 = grads[5]

    # Derivatives of cost function including L2 regularization
    GradW1 = (1/m)*(dCdW1 + lmbda*W1)
    GradW2 = (1/m)*(dCdW2 + lmbda*W2)
    GradW3 = (1/m)*(dCdW3 + lmbda*W3)
    Gradb1 = (1/m)*(dCdb1)
    Gradb2 = (1/m)*(dCdb2)
    Gradb3 = (1/m)*(dCdb3)

    # Update weights and biases
    W1 = W1 - eta*GradW1
    W2 = W2 - eta*GradW2
    W3 = W3 - eta*GradW3
```