# Project 2

**FYS-STK4155 — Applied data analysis and machine learning**

Anders Johansson
12th November 2018

**Abstract**

This project applies several statistical learning methods to data from the famous Ising model. Linear regression is used to determine the coupling, while classification of spin configurations as ordered or disordered is achieved with both logistic regression and multi-layered neural networks. Finally, neural networks are trained to predict the energy of a spin configuration.

LASSO regression is found to perfectly reproduce the Ising model and give an $R^2$ score of 1 for the correct choice of regularisation. Using 400 training samples requires $\lambda \in \left[10^{-3}, 10^{-2}\right]$, while 1000 allows for smaller values of $\lambda$. The largest viable $\lambda$ is $10^{-1}$ in both cases, after which the performance decreases rapidly. Visualisation of the coupling matrices shows LASSO being able to break the symmetry and return the original model, while Ridge returns a symmetric coupling matrix. Given enough training data, Ridge regression achieves the same performance.

A neural network with no hidden layers taking in couplings, in practice a linear regressor with stochastic batch gradient descent, also returns a symmetric coupling matrix. With a learning rate of 0.04, the neural network converges to give an $R^2$ score of 1 after 10 epochs. On a 4-core processor a batch size of 12 is found to minimise the training time. Additionally, a neural network with only one hidden layer is able to predict energies with an $R^2$ score of more than 0.98 when only fed the spins and not the couplings.

Logistic regression is able to classify spin configurations as ordered/disordered with an accuracy of approximately 70 %, while a neural network with one hidden layer with 400 neurons achieves perfect accuracy within 10 epochs. Consequently, neural networks proved to be superior, although the choice of hyperparameters, especially the learning rate, was crucial.

# Contents



# 1 Introduction

The Ising model is famous in many fields of science. Physicists study the model because it represents a magnetic lattice and can model ferromagnetic materials, and also because it has a large university class and therefore the same behaviour as many other models. Social scientists like the model because it is a simple correlated system, which can approximate correlated quantities such as dialects. The Metropolis algorithm, acknowledged as one of the premiere algorithms in computational science, is a method for finding the properties of the Ising model numerically.

This project does not deal with the simulation of a spin lattice, but rather the classification and analysis of pre-calculated spin configurations with corresponding energies and labels as ordered or disordered. Thus the Ising model can be used to verify various numerical methods for both regression and classification, and also measure their performance to infer their relative strengths and weaknesses.

The main goal is to compare simple neural networks with the different linear regression methods for regression and logistic regression for classification. The regression part of the problem is to find the coupling constants from given spin configurations and their energies, as well as prediction of energies. Binary classification of states as ordered or disordered is used as a classification problem.

Consequently, the first part of this report derives the necessary mathematical tools for both regression and classification, including the back propagation algorithm for training neural networks. Different methods for classification and regression are then applied to the Ising data provided in[2] and compared.

# 2 Theory and methods

See appendix B on page 21 for a review of linear regression, including terminology and regression problem statement.

## 2.1 The Ising model

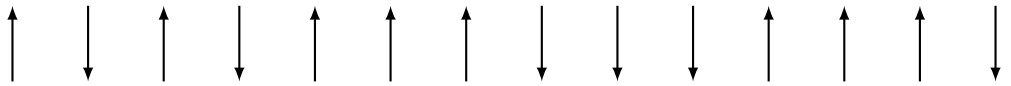


Figure 1: Example of a spin configuration in the one-dimensional Ising model. Spins pointing up and down represent $s = +1$ and $s = -1$.

The Ising model is a model for the energy of a system of binary spins, e.g. spins which only can take values $\pm 1$. Interactions are in this project assumed to be between nearest neighbours only, and the Hamiltonian is therefore given by

$$H = -J \sum_{\langle ij \rangle} s_i s_j, \tag{2.1}$$

where the sum runs over the nearest neighbours. In one dimension, this reduces to

$$H = -J \sum_{k=1}^{L} s_k s_{k+1}, \tag{2.2}$$

where $s_{L+1} = s_1$ in order to enforce periodic boundary conditions. According to the Boltzmann distribution, the probability of a state with energy $H$ is proportional to $\mathrm{e}^{-\beta H}$, where $\beta = 1/kT$ and $T$ is the temperature. At low temperatures, states with a low energy, i.e. ordered states, are preferred, while the probability distribution flattens for large temperature. Therefore, an unordered state is more likely at higher temperatures. The transition between preferring ordered and unordered states is called a *phase transition*. While no phase transition exists for the one-dimensional Ising model, the two-dimensional Ising model has a phase transition at $kT/J = 2.27$.

### 2.1.1 Regression problem

The performance of the different regression models can be evaluated on data created from the Ising model by "forgetting" the true model with a constant coupling coefficient $J$ and assuming that it is non-constant,

$$H = -\sum_{i,j} J_{ij} s_i s_j. \tag{2.3}$$

Given a system with $L$ spins, this gives $L^2$ basis functions on the form $-s_i s_j$ and $L^2$ coefficients $J_{ij}$ to be determined by regression to find an estimator of the energy of a spin configuration. Both linear regression methods and neural networks will be applied.

### 2.1.2 Classification problem

A dataset of two-dimensional spin configurations has been made available in [2]. These configurations have been created at different temperatures and are therefore labelled as either ordered or unordered. Logistic regression and neural networks can be used for this task.

## 2.2 Binary logistic regression

Binary logistic regression is applied to problems where there are two possible outcomes, e.g. $y_i = 1$ or $y_i = 0$. Given an input $\vec{x}_i$ (for example a spin configuration), a logistic regressor gives as output the probability $p_i$ that $y_i = 1$. The name logistic regression stems from the use of a logistic function as the value of $p$, where the logistic function is applied to a linear combination of functions evaluated in the output, i.e.

$$p_i = \frac{1}{1 + \mathrm{e}^{-\beta_j \phi_j(\vec{x}_i)}}. \tag{2.4}$$

The matrix $X$ consisting of $X_{ij} = \phi_j(\vec{x}_i)$ can be created in the same manner as in linear regression, and the equation above is straightforwardly vectorised as

$$\vec{p} = \frac{1}{1 + \mathrm{e}^{-X\vec{\beta}}}. \tag{2.5}$$

### 2.2.1 Cost function

If a given output $y_i$ is 1, $p_i$ gives the likelihood of this result as predicted by the logistic regressor. Similarly, if $y_i = 0$, $1 - p_i$ gives the likelihood. This can be combined into $l_i = p_i^{y_i}(1 - p_i)^{1-y_i}$. The product rule gives the total likelihood,

$$L = \prod_i l_i = \prod_i p_i^{y_i}(1 - p_i)^{1-y_i}, \tag{2.6}$$

which should be maximised by the logistic regressor. Since the logarithm is a strictly monotonous function, one can instead choose to maximise

$$\ln(L) = \sum_i y_i \ln(p_i) + (1 - y_i)\ln(1 - p_i), \tag{2.7}$$

a much nicer expression to differentiate. Finally, this can be turned into a minimisation problem with a minus sign, giving the final cost function

$$Q(\vec{\beta}) = -\sum_i (y_i \ln(p_i) + (1-y_i)\ln(1-p_i)), \tag{2.8}$$

called the cross-entropy.

### 2.2.2 Minimisation

Letting $t_i = X_{ij}\beta_j = \phi_j(\vec{x}_i)\beta_j$ and using the result

$$\frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{1}{1+\mathrm{e}^{-x}}\right) = \frac{\mathrm{e}^{-x}}{(1-\mathrm{e}^{-x})^2} = \frac{1+\mathrm{e}^{-x}-1}{(1+\mathrm{e}^{-x})^2} = \frac{1}{1+\mathrm{e}^{-x}} - \frac{1}{(1+\mathrm{e}^{-x})^2} = \frac{1}{1+\mathrm{e}^{-x}}\left(1-\frac{1}{1+\mathrm{e}^{-x}}\right), \tag{2.9}$$

the derivative of the cost function is

$$\frac{\partial Q}{\partial \beta_k} = \frac{\partial Q}{\partial p_j}\frac{\partial p_j}{\partial \beta_k} = \frac{\partial Q}{\partial p_j}\frac{\partial p_j}{\partial t_l}\frac{\partial t_l}{\partial \beta_k} = \sum_j \frac{\partial Q}{\partial p_j}\frac{\partial p_j}{\partial t_j}\frac{\partial t_j}{\partial \beta_k} = \sum_j \frac{\partial Q}{\partial p_j} p_j(1-p_j)X_{jk}. \tag{2.10}$$

Further,

$$\frac{\partial Q}{\partial p_j} = -\frac{y_j}{p_j} + \frac{1-y_j}{1-p_j} = \frac{-y_j(1-p_j)+p_j(1-y_j)}{p_j(1-p_j)} = \frac{p_j-y_j}{p_j(1-p_j)}, \tag{2.11}$$

giving

$$\frac{\partial Q}{\partial \beta_k} = (p_j - y_j)X_{jk}, \tag{2.12}$$

or, in matrix form,

$$\nabla_{\vec{\beta}} Q = X^T(\vec{p}-\vec{y}), \tag{2.13}$$

with $\vec{p} = 1/\left(1+\mathrm{e}^{-X\vec{\beta}}\right)$. The equation $\nabla Q = \vec{0}$ does not have an analytic solution, but the cost function can be minimised with the standard minimisation methods.

### 2.2.3 Regularisation

As with linear regression, a logistic regressor can be regularised by adding a term to the cost function in the hope of reducing variance and overfitting. One example is the same term as in Ridge regression,

$$Q = -\sum_i (y_i \ln(p_i) + (1-y_i)\ln(1-p_i)) + \tfrac{1}{2}\lambda\left\|\vec{\beta}\right\|_2^2, \tag{2.14}$$

which adds a simple term to the gradient of the cost function,

$$\nabla_{\vec{\beta}} Q = X^T(\vec{p}-\vec{y}) + \lambda\vec{\beta}. \tag{2.15}$$

## 2.3 Neural networks

Neural networks can be used for both classification and regression. They aim to be more accurate than linear and logistic regression by using a more complex set of functions for approximation. A neural network is a series of affine transformations and activation functions, where the activation function may for example be a logistic function.

Each pair of one affine transformation and one activation function is called a *layer*. The number of outputs from a layer is called the number of *neurons* in the layer. Mathematically, layer number $l$ contains a weight-matrix $W^l$, a bias vector $\vec{b}^l$ and an activation function $f^l$, takes input $\vec{a}^{l-1}$, applies the affine transformation $\vec{z}^l = W^l\vec{a}^{l-1} + \vec{b}^l$ and returns the output $f^l(\vec{z}^l)$.

Prediction is done recursively. The first layer, called the input layer, applies its affine transformation to the input vector and then its activation function. The second layer receives the output of the first layer and repeats the process. This is repeated up to and including the final (output) layer, whose activation function is tailored to either classification or regression.

### 2.3.1 Minimisation: Back-propagation

Similarly to logistic regression, the cost function should be minimised using some sort of gradient method. This requires taking the derivative of the cost function with respect to the fitting parameters, i.e. the weight matrices $W^l$ and the bias vectors $\vec{b}^l$ for neural networks. Since the prediction is calculated as a series of compositions of complicated functions, it is not possible to derive closed-form expressions of the derivatives of the cost function with respect to weights and biases. Instead, one can find the derivative with respect to the weights and biases of the *last* layer, and then derive a recursion relation which gives the derivatives at earlier layers as well. This is the famous *back-propagation* algorithm.

First, the derivatives of the cost function with respect to the biases can be calculated via the chain rule,

$$\frac{\partial Q}{\partial b_j^l} = \frac{\partial Q}{\partial z_k^l}\frac{\partial z_k^l}{\partial b_j^l} = \frac{\partial Q}{\partial z_k^l}\frac{\partial}{\partial b_j^l}\left(W_{ki}^l a_i^{l-1} + b_k^l\right) = \frac{\partial Q}{\partial z_k^l}\delta_{jk} = \frac{\partial Q}{\partial z_j^l} =: \delta_j^l. \tag{2.16}$$

Applying the chain rule again, $\delta_j^l$ is rewritten as

$$\delta_j^l = \frac{\partial Q}{\partial z_j^l} = \frac{\partial Q}{\partial a_k^l}\frac{\partial a_k^l}{\partial z_j^l} = \frac{\partial Q}{\partial a_k^l}\frac{\partial f^l\left(z_k^l\right)}{\partial z_j^l} = \frac{\partial Q}{\partial a_j^l}\frac{\partial f^l}{\partial z_j^l}. \tag{2.17}$$

Additionally, the derivatives with respect to the weight matrix elements are

$$\frac{\partial Q}{\partial W_{jk}^l} = \frac{\partial Q}{\partial z_i^l}\frac{\partial z_i^l}{\partial W_{jk}^l} = \delta_i^l\frac{\partial}{\partial W_{jk}^l}\left(W_{im}^l a_m^{l-1} + b_i^l\right) = \delta_j^l a_k^{l-1}. \tag{2.18}$$

These equations can be combined to give the derivatives of the cost function with respect to the weights and biases at the last layer, but do not say anything about the gradients at earlier layers, which would require knowledge of $\delta_j^l$ for $l < L$. This relation can be derived through one final application of the chain rule,

$$\delta_j^l = \frac{\partial Q}{\partial z_j^l} = \frac{\partial Q}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} = \delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial a_i^l}\frac{\partial a_i^l}{\partial z_j^l} = \delta_k^{l+1}\frac{\partial}{\partial a_i^l}\left(W_{km}^{l+1}a_m^l + b_k^l\right)\frac{\partial f^l(z_i^l)}{\partial z_j^l} = \delta_k^{l+1}W_{kj}^{l+1}f^{l\prime}(z_j^l). \tag{2.19}$$

The above results can be summarised on a matrix form as

$$\vec{\delta}^L = \nabla_{\vec{a}^L} Q \circ {f^L}'(\vec{z}^L), \tag{2.20}$$

which is easily calculated given a cost function $Q$ and final activation function $f^L$ ($\circ$ denotes the element-wise product),

$$\vec{\delta}^l = \left(W^{l+1,T}\vec{\delta}^{l+1}\right) \circ {f^l}'(\vec{z}^l), \tag{2.21}$$

which can be calculated recursively once $\vec{\delta}^L$ is known from the previous equation, and finally the gradients

$$\nabla_{\vec{b}^l} Q = \vec{\delta}^l \qquad \text{and} \qquad \nabla_{W^l} Q = \vec{\delta}^l \otimes \vec{a}^{l-1}. \tag{2.22}$$

Together, these four equations give a complete recipe of how to calculate the gradients of the cost function in a neural network.

### 2.3.2 Activation functions

The simplest activation function is the identity, whose derivative is 1. This is often used in the output layer for regression problems. Another activation function, which historically has been widely used in the intermediate (hidden) layers of the neural network, is the sigmoid or logistic function, whose derivative was shown earlier to be

$$f'(z) = f(z)(1 - f(z)). \tag{2.23}$$

Other activation functions include the hyperbolic tangent, with derivative

$$f'(z) = 1 - f(z)^2, \tag{2.24}$$

and the rectified linear unit (RELU),

$$f(z) = \begin{cases} z, & z \geq 0 \\ 0, & z < 0 \end{cases} \implies f'(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases}. \tag{2.25}$$

The rectified linear unit usually performs the best[2] and is used throughout this project (except in the output layer for classification).

### 2.3.3 Cost functions and output layers

When applying a neural network to regression problems, the natural choice of cost function is simply the squared deviation from the known outputs,

$$Q = \tfrac{1}{2}\left\|\vec{a}^L - \vec{y}\right\|_2^2, \tag{2.26}$$

with the gradient

$$\nabla_{\vec{a}^L} Q = \vec{a}^L - \vec{y}. \tag{2.27}$$

Regressing neural networks usually use the identity as the activation function for the output layer, giving

$${f^L}'(\vec{z}^L) = \vec{1} \tag{2.28}$$

and

$$\vec{\delta}^L = \vec{a}^L - \vec{y}. \tag{2.29}$$

Binary classification problems such as the one of the two-dimensional Ising model may use the same cost function as that of logistic regression, with the logistic function used as the activation function of the output layer in order to obtain a probability $a^L$ between zero and one. The cost function is then

$$Q = -\Big(y \ln\Big(a^L\Big) + (1-y)\ln\Big(1-a^L\Big)\Big) \tag{2.30}$$

with

$$\nabla_{\vec{a}^L} Q = \frac{\partial Q}{\partial a^L} = -\frac{y}{a^L} - \frac{1-y}{1-a^L} = \frac{a^L - y}{a^L(1-a^L)} \tag{2.31}$$

giving

$$\delta^L = \frac{\partial Q}{\partial a^L}\frac{\partial f^L}{\partial z^L} = \frac{a^L - y}{a^L(1-a^L)} \cdot f^L(z^L)\Big(1 - f^L(z^L)\Big) = a^L - y. \tag{2.32}$$

### 2.3.4 Implementation

As in the previous project, I have chosen to use an object-oriented approach and Fortran. This combination resulted in general, easy-to-use and efficient code for regression methods. The neural network is therefore similarly organised.

The most important class is the `neural_network` class. This contains methods for training, prediction and back propagation, which call corresponding methods on each element in the network's array of `layer` objects. Each `layer` has a weight matrix, a bias vector, some temporary output and input vectors, and an `activation_function` object. Subclasses of `activation_function` have been implemented for both the sigmoid function and the rectifying linear unit. These implement a method for evaluation and differentiation. Giving the last `layer` instances of different `activation_function` subclasses tailors the neural network to either classification or regression.

Coarrays, introduced in Fortran2008, were used to parallelise the neural network. Each batch in the stochastic batch gradient descent is divided among the available processes. When all processes have finished with their given amount of work, the resulting gradients are summed and used to update the weights. Synchronisation therefore only takes place between each batch. For large batch sizes, the speedup of parallelisation should be almost perfect.

## 2.4 Stochastic minimisation algorithms

See appendix B.5 on page 25 for a review of gradient descent and Newton's method.

Gradient descent and Newton's method are often suboptimal because of their tendency to get stuck in local minima. A simple extension is the stochastic batch-gradient descent, which selects batches of inputs from the training data with replacement and trains either a neural network or a logistic regressor on each batch. If $b$ batches are chosen out of the $N$ training data sets, there are $N/b$ input items to be chosen with replacement for each batch, and the gradient descent update step is

$$\vec{\beta}_{k+1} = \vec{\beta}_k - \alpha \nabla_{\vec{\beta}} Q\Big(X_{[\vec{\imath},:]}, \vec{y}_{\vec{\imath}}\Big), \tag{2.33}$$

where $Q\left(X_{[\vec{\imath},:]}, \vec{y}_{\vec{\imath}}\right)$ denotes the gradient applied to $\vec{x}_j$ for all $j \in \vec{\imath}$, $\vec{\imath}$ being a set of $N/b$ indices.

A further improvement is to add a so-called momentum, i.e. use a linear combination of the current and previous gradient rather than just using the current gradient,

$$\vec{v}_k = \gamma \vec{v}_{k-1} + Q\left(X_{[\vec{\imath},:]}, \vec{y}_{\vec{\imath}}\right), \qquad \vec{\beta}_{k+1} = \vec{\beta}_k - \alpha \vec{v}_k. \tag{2.34}$$

# 3 Results and discussion

## 3.1 Linear regression

One-dimensional binary lattices were generated randomly, and their energies were calculated. The regression methods were told to approximate the energies as linear combinations of spin-spin products, $E = -J_{ij} s_i s_j$. An interesting feature of this regression problem is the non-uniqueness of the coupling matrix. While the energy is generated from $E = -\sum_j J_{j,j+1} s_j s_{j+1}$, any matrix $J$ which fulfils $J_{j,j+1} + J_{j,j-1} = 1$ and $J_{ij} = -J_{ji}$ for $i \neq j \pm 1$ (requiring $J_{ii} = 0$) will give the same energy.

Widely different behaviours are seen for the three regression algorithms. Ordinary least squares gives strange results, while Ridge and LASSO regression give good, but different, results. Ridge regression, whose cost function includes the sum of the squares of the coefficients $\beta_j$, gives a symmetric coupling matrix because $0.5^2 + 0.5^2$ is the smallest squared sum of two numbers which add up to one. The resulting performance is an $R^2$ score around 0.8 with 1000 training samples when $\lambda$ is not chosen too large.

On the other hand, LASSO regression is able to perfectly reproduce the asymmetric coupling matrix used to generate the data with the correct choice of regularisation parameter, resulting in an $R^2$ score of 1. Using more training samples reduces the sensitivity to $\lambda$, while the peak performance is the same for 400 and 1000 samples.

When the number of samples reaches the number of parameters, $L^2 = 1600$, Ridge regression gives perfect energies because the underlying model of the energy is the same as the model used for fitting. Reducing the number of samples reduces the accuracy. LASSO regression, on the other hand, manages to find a perfect fit for certain values of $\lambda$ even with 400 samples.

Coupling matrices showing transitions and other interesting features are shown below. A complete set is given in figure 14 on page 19 and beyond.

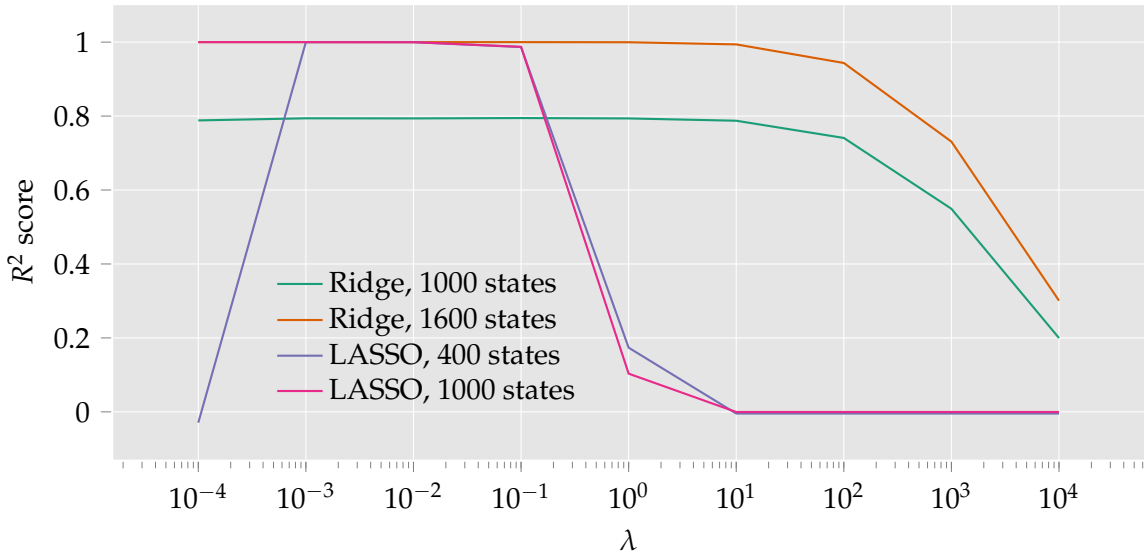



Figure 2: $R^2$ score on test data. While the regularisation of Ridge regression does not significantly improve the performance, as illustrated by the constant coupling matrices in figure 5 on the following page, the correct choice of regularisation parameter gives LASSO perfect predictive ability. This corresponds well with the coupling matrices shown in figure 5 and onwards, which show that LASSO regularisation gives the same, asymmetric coupling matrices as those of the underlying model.

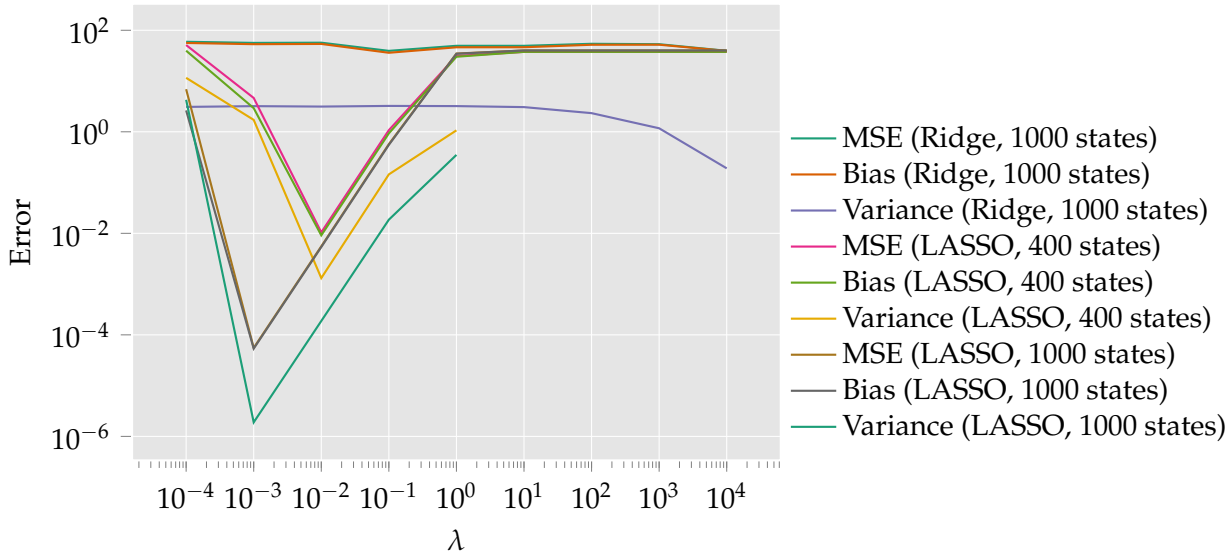
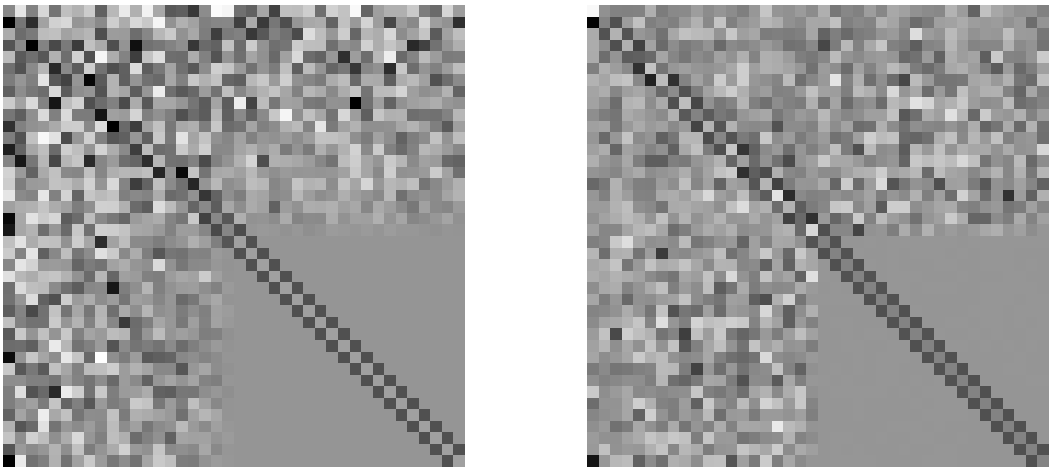
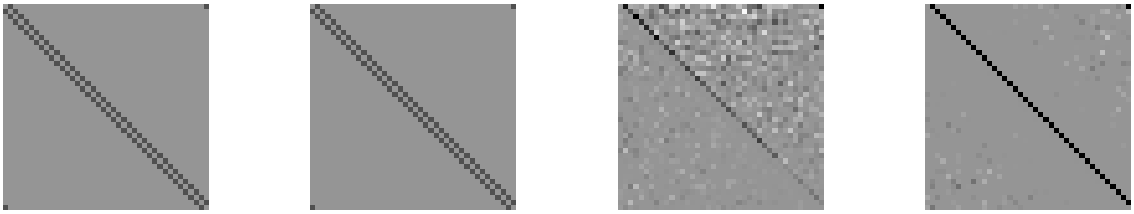



Figure 3: Bias-variance decomposition of the mean squared error for Ridge and LASSO regression. The bias and variance are expected to add up to the mean squared error, which is confirmed by the plot. Further, using a regularisation parameter $\lambda$ which is too large causes the mean squared error to increase because the approximating model becomes unable to fit the data, i.e. the bias increases, and the reduction of variance can not compensate sufficiently. As seen also in figure 2, LASSO outperforms Ridge regression for certain values of $\lambda$ when fewer samples than $L^2$ is used.



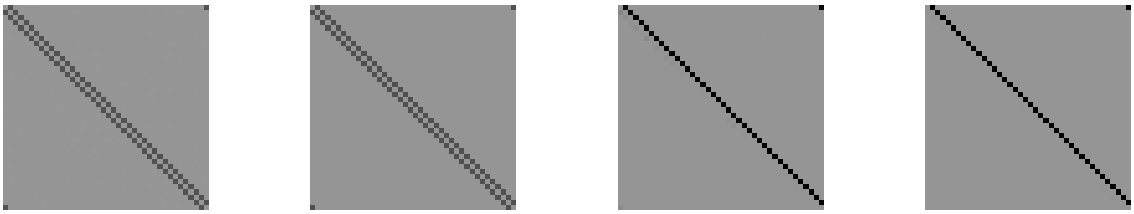
(a) 1000 states. (b) 1600 states.

Figure 4: Coupling matrix for ordinary least squares regression. White is negative one, black is one. It is clear that the lack of regularisation causes overfitting when few states are used as input. Additionally, ordinary least squares and Ridge regression find a coupling matrix whose super and sub diagonals are equal. The strange lower right quarter of the coupling matrix can be caused by either something wrong which I have done, or the use of `dgelsd`, which finds a singular value decomposition via the divide-and-conquer algorithm. Notice that most of the noise (that is, non-zero elements other than on the sub and super diagonals) is antisymmetric, so that these elements cancel and do not affect the energy.



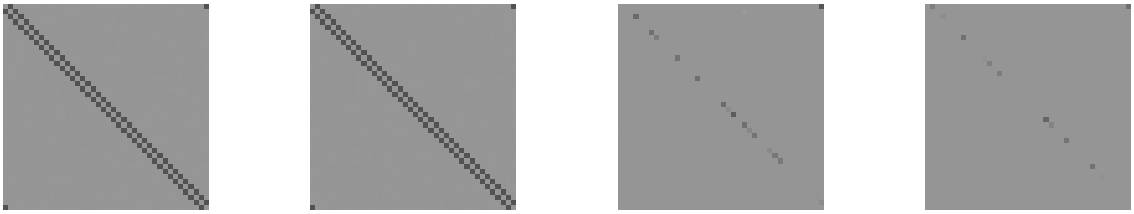
(a) Ridge regression, 1000 training states. (b) Ridge regression, 1600 training states. (c) LASSO regression, 400 training states. (d) LASSO regression, 1000 training states.

Figure 5: Coupling matrices for $\lambda = 1.00 \cdot 10^{-4}$.



(a) Ridge regression, 1000 training states. (b) Ridge regression, 1600 training states. (c) LASSO regression, 400 training states. (d) LASSO regression, 1000 training states.

Figure 6: Coupling matrices for $\lambda = 1.00 \cdot 10^{-3}$.



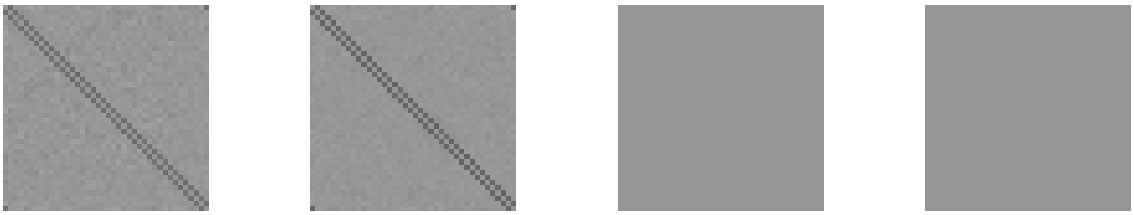
(a) Ridge regression, 1000 training states.

(b) Ridge regression, 1600 training states.

(c) LASSO regression, 400 training states.

(d) LASSO regression, 1000 training states.

Figure 7: Coupling matrices for $\lambda = 1.00$.



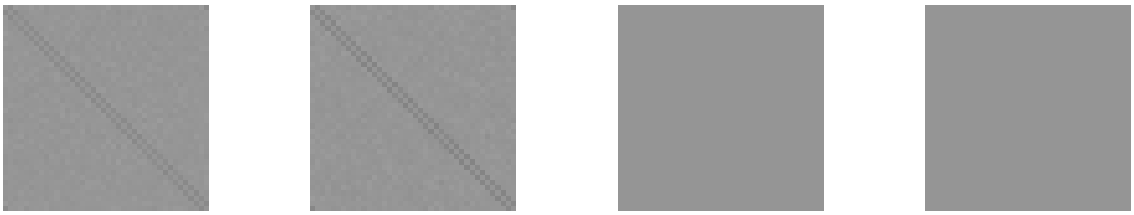
(a) Ridge regression, 1000 training states.

(b) Ridge regression, 1600 training states.

(c) LASSO regression, 400 training states.

(d) LASSO regression, 1000 training states.

Figure 8: Coupling matrices for $\lambda = 1.00 \cdot 10^3$.



(a) Ridge regression, 1000 training states.

(b) Ridge regression, 1600 training states.

(c) LASSO regression, 400 training states.

(d) LASSO regression, 1000 training states.

Figure 9: Coupling matrices for $\lambda = 1.00 \cdot 10^4$.

### 3.2 Logistic regression

Mehta et. al.[2] have provided a data set with 160 000 spin configurations on a $40 \times 40$ lattice. The states are labelled as ordered or disordered. Of these, 30 000 are considered *critical*, i.e. in the range between ordered and disordered (this phase transition is sharp only in the limit of infinite lattice size). The non-critical states were divided into training and test data, while the critical states were kept as an extra test set.

There are many parameters to be chosen in logistic regression. I chose to vary regularisation, learning rate and momentum to determine the optimal combination of parameters. The batch size could also

have been varied, but this was kept fixed for simplicity and reduction of runtime. Additionally, the batch size mainly determines the rate of convergence, while wrong choices for the other parameters often lead to no convergence at all. For each $\lambda$, an optimal learning rate and momentum was determined through a (parallel) parameter sweep.

Table 1: Accuracy on training and test data, as well as the critical states, for a variety of regularisations $\lambda$. Optimal momentums and learning rates are estimated through a simple parameter sweep. The number of stochastic gradient descent iterations was limited to 100, while the batch size was 32. Other prints from the program show that the choice of hyperparameters such as learning rate is very important for logistic regression to be better than guessing.

| $\lambda$ | Training accuracy | Test accuracy | Critical accuracy | Learning rate | Momentum |
|---|---|---|---|---|---|
| | 0.71 | 0.69 | 0.62 | $1.0 \cdot 10^{-2}$ | 0.20 |
| $1.0 \cdot 10^{-5}$ | 0.72 | 0.70 | 0.63 | $1.0 \cdot 10^{-2}$ | 0.60 |
| $1.0 \cdot 10^{-4}$ | 0.71 | 0.69 | 0.62 | $1.0 \cdot 10^{-2}$ | 0.40 |
| $1.0 \cdot 10^{-3}$ | 0.72 | 0.70 | 0.62 | $1.0 \cdot 10^{-2}$ | 0.80 |
| $1.0 \cdot 10^{-2}$ | 0.71 | 0.70 | 0.63 | $1.0 \cdot 10^{-2}$ | 0.80 |
| $1.0 \cdot 10^{-1}$ | 0.71 | 0.70 | 0.63 | $1.0 \cdot 10^{-1}$ | 0.00 |
| $1.0 \cdot 10^{0}$ | 0.68 | 0.67 | 0.68 | $1.0 \cdot 10^{-3}$ | 0.40 |
| $1.0 \cdot 10^{1}$ | 0.68 | 0.67 | 0.63 | $1.0 \cdot 10^{-3}$ | 0.40 |
| $1.0 \cdot 10^{2}$ | 0.51 | 0.51 | 0.50 | $1.0 \cdot 10^{-2}$ | 1.00 |

## 3.3 Neural network regression

### 3.3.1 Rediscovering the Ising model

A neural network with only an output layer and no activation function is identical to a linear regressor, except for the use of a minimisation algorithm instead of closed form solutions. If the same input is used as in section 3.1 on page 8, the weight matrix of such a neural network should be equal to the coupling matrix of the Ising model.

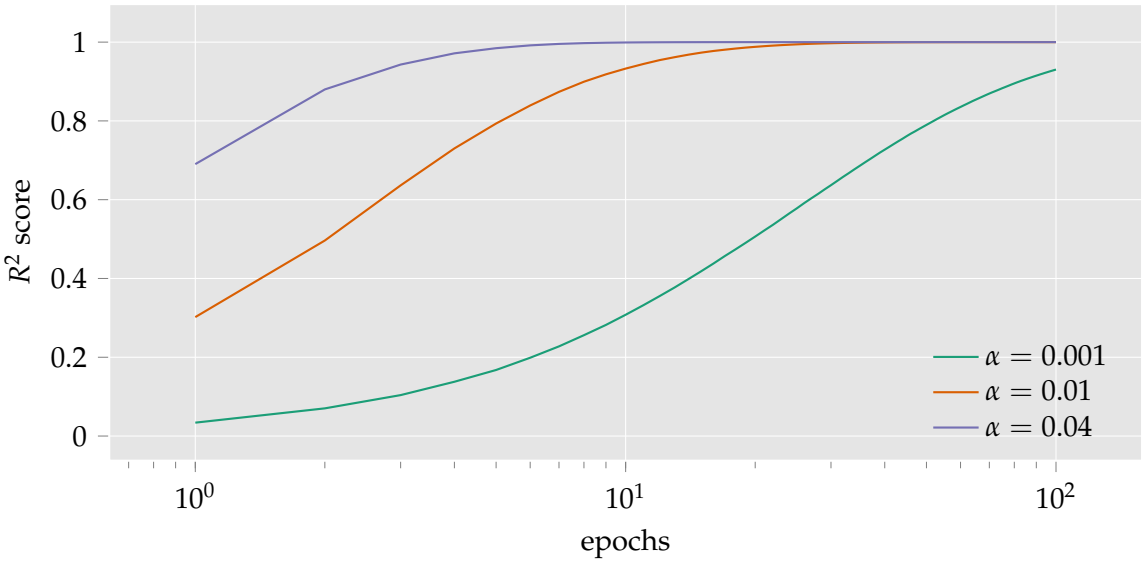



Figure 10: $R^2$ score on test data as a function of the number of epochs, i.e. the number of stochastic gradient descent iterations. 3000 states were used for training with a batch size of 300. With the right learning rate, the neural network converges quickly to the correct solution, while a badly chosen learning rate leads to slower convergence or even divergence. A learning rate of 0.05 ruins everything (not shown).

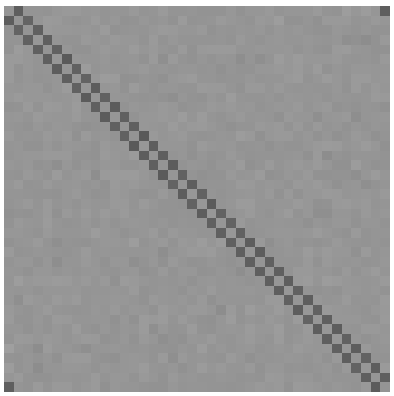


Figure 11: Weight matrix of the neural network's only layer, which should be equal to the coupling matrix $J$. The neural network gives a symmetric matrix with weights 0.5 on the super and sub diagonals.

Table 2: Optimal combination of learning rate and batch size for a few different values of $\lambda$. The given numbers of epochs and training time are determined by registering when the neural network gives $R^2 \geq 0.98$ on test data. Optimal parameters are defined as the parameters which give the shortest *training time*. A batch size of 1 always gives the smallest number of required epochs for this simple problem, but does not utilise the parallelisation of the neural network. These results are generated on a 4-core processor. Using more cores would lead to a larger optimal batch size. A suitable choice of regularisation parameter is occasionally seen to reduce training time and the required number of epochs, while too strong regularisation keeps the neural network from converging sufficiently.

| $\lambda$ | Learning rate | Batch size | Epochs for convergence | Training time [s] |
|---|---|---|---|---|
| | $1.0 \cdot 10^{-2}$ | 20 | 21 | $8.2 \cdot 10^{-2}$ |
| $1.0 \cdot 10^{-5}$ | $1.0 \cdot 10^{-2}$ | 12 | 14 | $8.2 \cdot 10^{-2}$ |
| $1.0 \cdot 10^{-4}$ | $1.0 \cdot 10^{-2}$ | 12 | 14 | $6.5 \cdot 10^{-2}$ |
| $1.0 \cdot 10^{-3}$ | $1.0 \cdot 10^{-2}$ | 12 | 13 | $6.1 \cdot 10^{-2}$ |
| $1.0 \cdot 10^{-2}$ | $1.0 \cdot 10^{-2}$ | 12 | 14 | $1.2 \cdot 10^{-1}$ |
| $1.0 \cdot 10^{-1}$ | $1.0 \cdot 10^{-2}$ | 12 | 14 | $6.5 \cdot 10^{-2}$ |
| $1.0 \cdot 10^{0}$ | $1.0 \cdot 10^{-2}$ | 40 | 53 | $4.2 \cdot 10^{-1}$ |

#### 3.3.2 Energy from a black body

The previous exercise did not use the full power of neural networks, i.e. their ability to take in some input, e.g. a spin system, and spit out some output, e.g. energy, without any knowledge of how the output should depend on the input — a "black box"-behaviour. Instead, the neural network was turned into a linear regression machine because it was told to apply an affine transformation to fit data which was known to be linear.

It is therefore interesting to see whether the neural network can predict energies when it is just given the spin configuration and no other knowledge. Since the energy has a non-linear dependence on the spins, this requires more than one layer. As seen in figure 12 on the following page, a modest neural network with one hidden layer with 400 neurons achieves an $R^2$ score above 0.98. Increasing the number of layers does not contribute significantly, since the energy is a linear combination of products of spins, requiring a total of two layers.

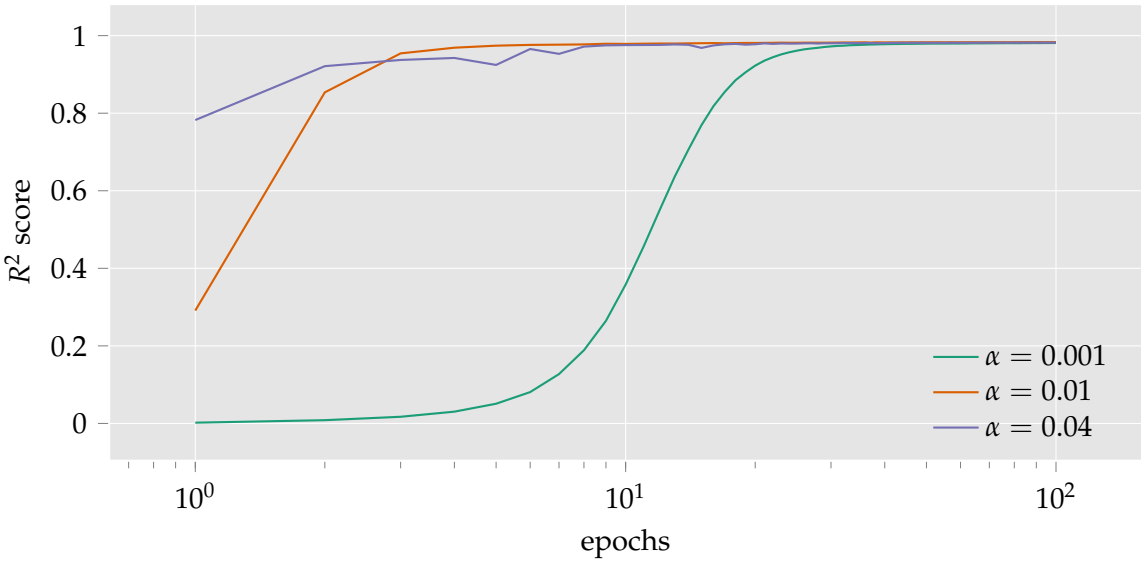



Figure 12: $R^2$ score on training data as a function of the number of epochs, i.e. the number of stochastic gradient descent iterations. 10 000 states were used for training with a batch size of 20, $\lambda = 0.001$ and $L = 40$. The neural network consists of one hidden layer with 400 neurons. Other simulations, especially when using more layers, show that while a learning rate of 0.04 initially converges the fastest, a smaller learning rate results in a better final $R^2$ score. An adaptive scheme may therefore perform better.

### 3.4 Neural network classification

The logistic regression in section 3.2 on page 11 showed mediocre performance, never achieving an accuracy above approximately 70 %. A neural network, with its series of affine transformations and activation functions, has a much more complex approximating function, more degrees of freedom and therefore a better chance of reaching a near-perfect predicting performance.

Figure 13 on the following page shows that this is indeed the case. Whereas logistic regression required all 130 000 spin configurations to reach an accuracy of 70 %, simple neural networks converge to perfect accuracy on test data within few epochs when using one tenth of the spin configurations. In addition to the improved accuracy, this means that the neural networks are more efficient despite their higher complexity, because they need less data and fewer epochs.

Interestingly, a deeper neural network does not seem to converge in fewer epochs. This may be due to the fact that the order/disorder in a lattice configuration only depends on quantities like the magnetisation, which is a first order polynomial of the spins (the sum), and the energy, which is a second order polynomial. A second order approximation only requires one hidden layer, since both the hidden and the output layer apply an affine transformation.

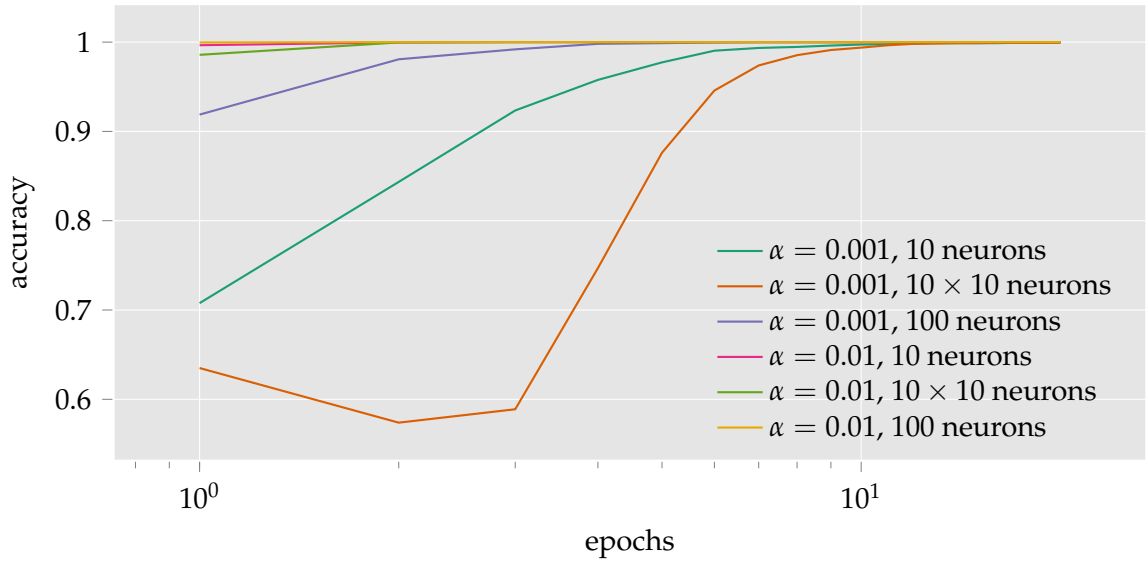



Figure 13: Accuracy on test data as a function of the number of epochs for a few neural network architectures and two different learning rates with batch size 20 and $\lambda = 0.001$. Only one tenth of the data provided by Mehta et. al.[2] was used. This amount of data rendered logistic regression useless, while the neural networks quickly converge to perfect accuracy with as little as one hidden layer with 10 neurons.

## 4 Summary and conclusion

Linear regression, logistic regression and neural networks have been apllied to the Ising model, both as a regression and as a classification problem.

The regression problem was best solved with LASSO regression, which was able to give a perfect $R^2$ score of 1 with as little as 400 states — much less than the number of parameters to be determined. The choice of regularisation parameter was, however, very important to achieve this accuracy. Ridge regression achieved similarly when given sufficient training data, even though it returns a symmetric coupling matrix. LASSO, on the other hand, was the only regression method giving the same coupling matrix as the one used to generate the data.

Neural networks also performed well on the regression problem, and neural networks are the only regression method capable of predicting energies when only given spin configurations and not the spin-spin couplings. An $R^2$ score above 0.98 was quickly achieved with only one hidden layer. When given couplings and only using an output layer, the neural network was reduced to a linear regressor and performed equally well as Ridge regression, giving the same coupling matrix.

The highest accuracy when classifying spin configurations as orderered or disordered was, by far, achieved by neural networks. Logistic regression failed to reach an accuracy above 70 % even when trained on 100 000 states, while neural networks achieved perfect accuracy when trained on one tenth of the data.

The conclusion of this project is to use knowledge of the system to choose a specialised method whenever possible, such as in the case of the energy being a linear combination of spin-spin couplings.

On the other hand, neural networks proved to be versatile and powerful tools for both classification and regression. They perform well when given the right parameters, which often have to be estimated by a parameter sweep.

Further work is definitely to do larger and more thorough simulations in order to determine the optimal hyperparameters such as batch size and learning rate, as these have proven to be crucial for good performance in neural networks. Additionally, adaptive and more advanced minimisation methods should be implemented, as an adaptive learning rate should be able to combine the fast initial convergence of a large learning rate with the improved precision of a smaller learning rate, thus giving both fast convergence and excellent performance.

# References

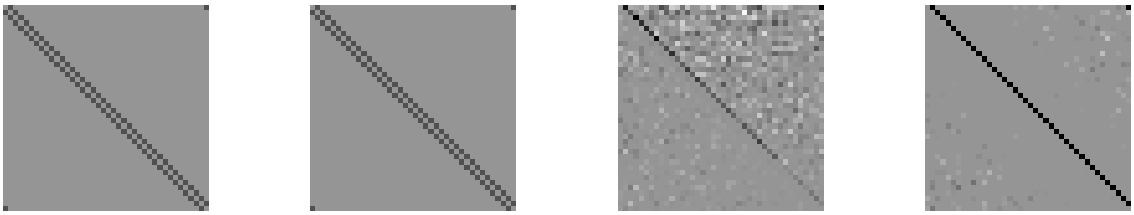
[1] Jerome Friedman, Trevor Hastie and Rob Tibshirani. "Regularization paths for generalized linear models via coordinate descent". In: *Journal of statistical software* 33.1 (2010), p. 1.

[2] Pankaj Mehta et al. "A high-bias, low-variance introduction to machine learning for physicists". In: *arXiv preprint arXiv:1803.08823* (2018).

[3] Wessel N van Wieringen. "Lecture notes on ridge regression". In: *arXiv preprint arXiv:1509.09169* (2015).

# Appendix

## A More coupling matrices



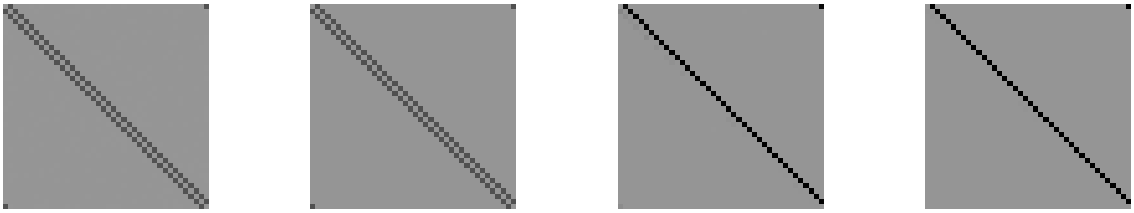
(a) Ridge regression, 1000 training states.

(b) Ridge regression, 1600 training states.

(c) LASSO regression, 400 training states.

(d) LASSO regression, 1000 training states.

Figure 14: Coupling matrices for $\lambda = 1.00 \cdot 10^{-4}$.



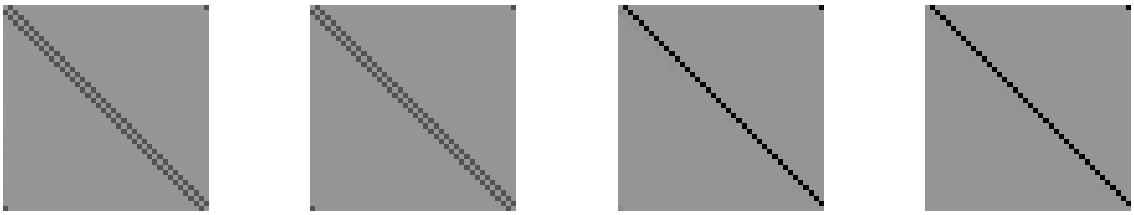
(a) Ridge regression, 1000 training states.

(b) Ridge regression, 1600 training states.

(c) LASSO regression, 400 training states.

(d) LASSO regression, 1000 training states.

Figure 15: Coupling matrices for $\lambda = 1.00 \cdot 10^{-3}$.



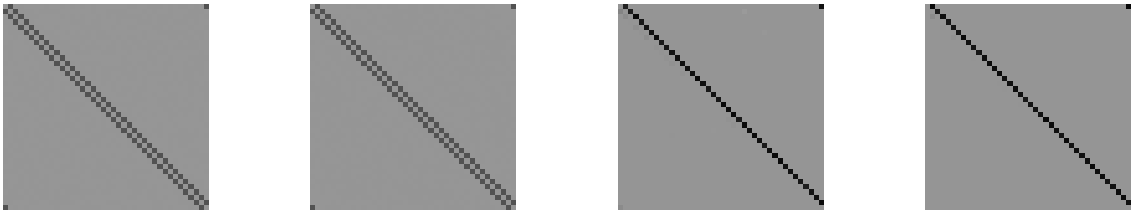
(a) Ridge regression, 1000 training states.

(b) Ridge regression, 1600 training states.

(c) LASSO regression, 400 training states.

(d) LASSO regression, 1000 training states.

Figure 16: Coupling matrices for $\lambda = 1.00 \cdot 10^{-2}$.



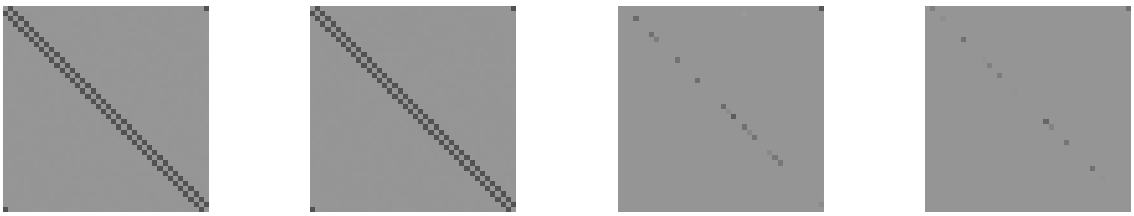
(a) Ridge regression, 1000 training states.

(b) Ridge regression, 1600 training states.

(c) LASSO regression, 400 training states.

(d) LASSO regression, 1000 training states.

Figure 17: Coupling matrices for $\lambda = 1.00 \cdot 10^{-1}$.



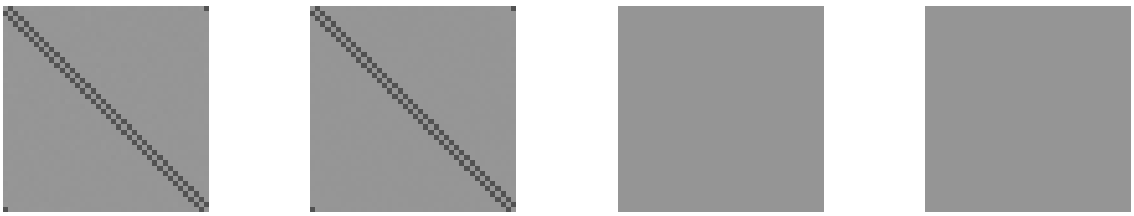
(a) Ridge regression, 1000 training states.

(b) Ridge regression, 1600 training states.

(c) LASSO regression, 400 training states.

(d) LASSO regression, 1000 training states.

Figure 18: Coupling matrices for $\lambda = 1.00$.



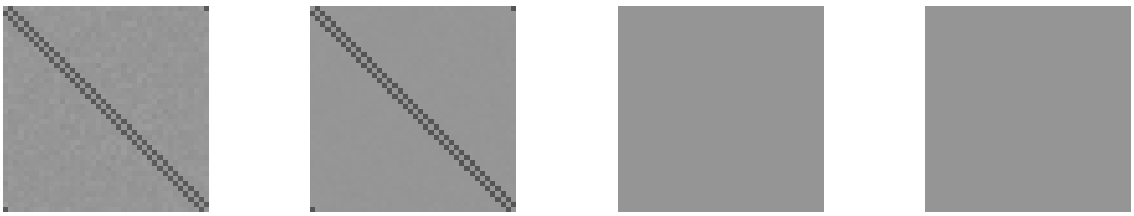
(a) Ridge regression, 1000 training states.

(b) Ridge regression, 1600 training states.

(c) LASSO regression, 400 training states.

(d) LASSO regression, 1000 training states.

Figure 19: Coupling matrices for $\lambda = 1.00 \cdot 10^{1}$.



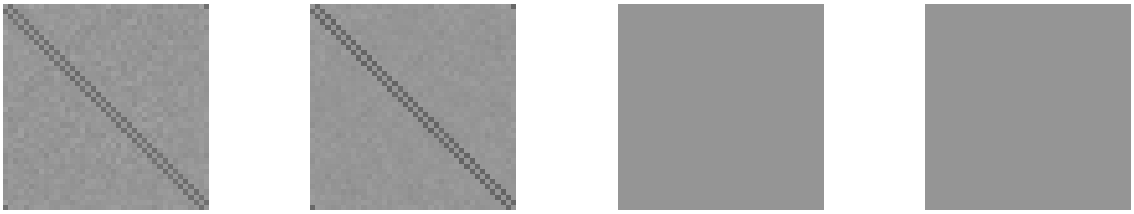
(a) Ridge regression, 1000 training states.

(b) Ridge regression, 1600 training states.

(c) LASSO regression, 400 training states.

(d) LASSO regression, 1000 training states.

Figure 20: Coupling matrices for $\lambda = 1.00 \cdot 10^{2}$.



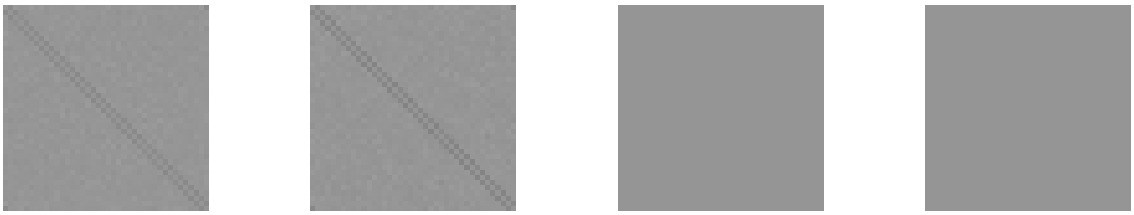
(a) Ridge regression, 1000 training states.

(b) Ridge regression, 1600 training states.

(c) LASSO regression, 400 training states.

(d) LASSO regression, 1000 training states.

Figure 21: Coupling matrices for $\lambda = 1.00 \cdot 10^3$.



(a) Ridge regression, 1000 training states.

(b) Ridge regression, 1600 training states.

(c) LASSO regression, 400 training states.

(d) LASSO regression, 1000 training states.

Figure 22: Coupling matrices for $\lambda = 1.00 \cdot 10^4$.

# B Review of linear regression

This section is mainly a copy-paste from project 1 and is included for reference.

## B.1 Fitting problem statement

The general problem is to fit a given set of data $D = \{(\vec{x}_i, y_i)\}_{i=1}^N$, where $\vec{x}_i$ are inputs of some dimensionality (2 in this project) and $y_i$ are scalar outputs. A set of basis functions $\{\phi_j\}_{j=1}^p$ is chosen, and the goal is to approximate the input data with the linear combination $\beta_j \phi_j(\vec{x})$. If the data set were generated by such a linear combination, there would exist parameters $\{\beta_j\}$ such that $\beta_j \phi_j(\vec{x}_i) = y_i$, which can be rewritten as the matrix equation

$$X\vec{\beta} = \vec{y}, \tag{B.1}$$

where $X \in \mathbb{R}^{N \times p}$ is the matrix with elements $x_{ij} = \phi_j(\vec{x}_i)$, and $\vec{\beta}$ and $\vec{y}$ are the vectors with elements $\beta_j$ and $y_i$.

In general, it will not be possible to find parameters $\beta_j$ which fulfill this, since the data set will not be generated from the simple basis functions. Consequently, one must find the $\beta_j$s which in some sense make $X\vec{\beta}$ as close to $\vec{y}$ as possible. The way in which to measure deviation from the perfect solution is

called the *cost function*, frequently written $Q(\vec{\beta}; D)$. Regression methods differ in the choice of cost function, while their aim is always to find the parameters $\beta_j$ which minimise the cost function.

When a regression method has been used to find $\vec{\beta}$, predicted values are denoted $\tilde{y}_i = X_{ij}\beta_j$, while the original, exact values are denoted $y_i$.

## B.2 Ordinary Least Squares

The ordinary least squares method is the simplest and most intuitive, since its cost function is simply the square of the error made by the fit,

$$Q(\beta; D) = \left\| \vec{y} - X\vec{\beta} \right\|_2^2, \tag{B.2}$$

where $\|\cdot\|_p$ denotes the usual $p$-norm.

### B.2.1 Geometric view

The geometric view of the equation $X\vec{\beta} = \vec{y}$ is to find the linear combination of the columns of $X$ equal to $\vec{y}$. This is not necessarily possible if the columns of $X$ do not span $\mathbb{R}^N$, which is not possible if $p < N$. Ordinary least squares seeks to find the linear combination of the columns of $X$ as close to $\vec{y}$ as possible. This is achieved when $X\vec{\beta}$ is equal to the projection of $\vec{y}$ onto the column space of $X$, i.e.

$$X\vec{\beta} = \text{Proj}_{\text{Col}\, X}\, \vec{y}. \tag{B.3}$$

By construction, this equation will always have a solution, although it may not be unique if the columns of $X$ are not linearly independent. Since $X\vec{\beta}$ is the projection of $\vec{y}$ onto the column space of $X$, the error, $\vec{y} - X\vec{\beta}$, is orthogonal to the columns of $X$, i.e. the rows of $X^T$. Consequently,

$$X^T\left(\vec{y} - X\vec{\beta}\right) = \vec{0} \implies X^T X\vec{\beta} = X^T\vec{y}, \tag{B.4}$$

which is a simple linear set of equations which can be solved for $\vec{\beta}$. This set of equations is called the *normal equations*. If $X$ is non-singular, the minimisation problem in ordinary least squares has the closed form solution

$$\vec{\beta}_{\text{OLS}} = \left(X^T X\right)^{-1} X^T\vec{y}. \tag{B.5}$$

### B.2.2 Minimisation view

The cost function for ordinary least squares can be written as

$$Q(\beta; D) = \left\| \vec{y} - X\vec{\beta} \right\|_2^2 = \sum_{i=1}^{N} \left(y_i - X_{ij}\beta_j\right)^2. \tag{B.6}$$

When this is minimised, $\nabla Q = \vec{0}$, where the gradient denotes differentiation with respect to the parameters $\beta_j$. The components of the gradient can straightforwardly be calculated from the cost

function,

$$\nabla_k Q = \frac{\partial}{\partial \beta_k}\big((y_i - X_{ij}\beta_j)(y_i - X_{ij}\beta_j)\big) = 2(y_i - X_{ij}\beta_j)\frac{\partial}{\partial \beta_k}(y_i - X_{ij}\beta_j) = -2X_{ik}(y_i - X_{ij}\beta_j), \quad \text{(B.7)}$$

which can be rewritten on vector form as

$$\nabla Q = -2X^T\Big(\vec{y} - X\vec{\beta}\Big). \quad \text{(B.8)}$$

Setting $\nabla Q = \vec{0}$ gives equation (B.4) on the previous page.

Ordinary least squares has been implemented by calling `dgelss`, which uses a singular value decomposition to prevent numerical instabilities.

### B.3 Ridge regression

The cost function used in Ridge regression is

$$Q_\lambda(\vec{\beta}; D) = \Big\|\vec{y} - X\vec{\beta}\Big\|_2^2 + \lambda\Big\|\vec{\beta}\Big\|_2^2 = \sum_{i=1}^{N}\big(y_i - X_{ij}\beta_j\big)^2 + \lambda\sum_{i=1}^{p}\beta_i^2, \quad \text{(B.9)}$$

which penalises solution vectors $\vec{\beta}$ where some coefficients are large. $\lambda$ is a parameter which should be chosen with great care. The error is as small as possible for $\lambda = 0$, as this will reproduce the solution found by ordinary least squares, but a non-zero value of $\lambda$ will give $\beta_j$s which yield more reasonable predictions for other values of $\vec{x}$ than those contained in the training data set $D$[2].

Using the derivative of the first term from equation (B.8), the gradient of the cost function is

$$\nabla Q_\lambda = -2X^T\Big(\vec{y} - X\vec{\beta}\Big) + 2\lambda\vec{\beta} = -2X^T\vec{y} + 2X^TX\vec{\beta} + 2\lambda\vec{\beta} = -2X^T\vec{y} + 2\Big(X^TX + \lambda I\Big)\vec{\beta}. \quad \text{(B.10)}$$

Minimisation requires $\nabla Q_\lambda = \vec{0}$, which gives

$$\Big(X^TX + \lambda I\Big)\vec{\beta} = X^T\vec{y}. \quad \text{(B.11)}$$

This can be solved for $\vec{\beta}$, and the closed-form solution to the minimisation of the Ridge cost function is

$$\vec{\beta}_{\text{Ridge}} = \Big(X^TX + \lambda I\Big)^{-1}X^T\vec{y}. \quad \text{(B.12)}$$

Having a non-zero $\lambda$ clearly reduces problems with linearly dependent columns of $X$ and singularity of $X^TX$.

Ridge regression has been implemented by calculating $X^TX$ and adding $\lambda$ on the diagonal, calculating $X^Ty$ and using `dposv` to find $\vec{\beta}$ using a Cholesky-decomposition, since $X^TX + \lambda I$ is positive definite.

### B.4 LASSO regression

The cost function used in LASSO regression is

$$Q_\lambda(\vec{\beta}; D) = \Big\|\vec{y} - X\vec{\beta}\Big\|_2^2 + \lambda\Big\|\vec{\beta}\Big\|_1 = \sum_{i=1}^{N}\big(y_i - X_{ij}\beta_j\big)^2 + \lambda\sum_{i=1}^{p}|\beta_i|, \quad \text{(B.13)}$$

which also penalises solution vectors $\vec{\beta}$ where some coefficients are large. LASSO regression has the advantage that certain choices of $\lambda$ will give a sparse solution, i.e. $\beta_j = 0$ for some values of $j$[3].

As with the other regression methods, the gradient of the cost function should now be differentiated and set to zero. Mathematicians will now point out that the absolute value is not differentiable at zero and start deriving ingenious minimisation methods to circumvent the problem. This is usually a non-issue in a physics course — I will now *define* the derivative to be

$$\frac{\mathrm{d}|x|}{\mathrm{d}x} := \operatorname{sgn} x := \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases} \tag{B.14}$$

and proceed happily.

The gradient of the cost function is thus

$$\nabla Q_\lambda = -2X^T\left(\vec{y} - X\vec{\beta}\right) + \lambda \operatorname{sgn}\left(\vec{\beta}\right), \tag{B.15}$$

and $\nabla Q_\lambda = \vec{0}$ does unfortunately not have a closed-form solution. A separate minimisation algorithm must therefore be used to find the parameters $\beta_j$ which minimise the cost function. The calculated gradient can be put straight into the gradient descent algorithm with a suitable step length. Alternatively, Newton's method can be used with the second derivative (Hessian) matrix of the cost function. The second derivative of the absolute value is even more problematic, which is solved by setting it equal to zero.

The Hessian matrix of the LASSO cost function is thus the same as the second derivative of the ordinary least squares cost function,

$$H_{kl} = H_{lk} = \frac{\partial^2 Q_\lambda}{\partial\beta_l \partial\beta_k} = \frac{\partial}{\partial\beta_l}(\nabla_k Q) \overset{\text{equation (B.7)}}{=} \frac{\partial}{\partial\beta_l}\left(-2X_{ik}\left(y_i - X_{ij}\beta_j\right)\right) = 2X_{ik}X_{il}, \tag{B.16}$$

which is the component form of the matrix equation

$$H = 2X^T X. \tag{B.17}$$

The gradient can now be rewritten as

$$\nabla Q_\lambda = -2X^T\vec{y} + H\vec{\beta} + \lambda \operatorname{sgn}\left(\vec{\beta}\right). \tag{B.18}$$

Equation (B.22) on the next page in Newton's method can then transformed into

$$H\left(\vec{\beta}_{i+1} - \vec{\beta}_i\right) = 2X^T\vec{y} - H\vec{\beta}_i - \lambda \operatorname{sgn}\left(\vec{\beta}_i\right) \implies H\vec{\beta}_{i+1} = 2X^T\vec{y} - \lambda \operatorname{sgn}\left(\vec{\beta}_i\right). \tag{B.19}$$

Since $H = 2X^T X$, this reduces to the normal equations of ordinary least squares (equation (B.4) on page 22) in the case of $\lambda = 0$, as it should.

Minimisation of the LASSO cost function has the benefit that the second derivative is independent of $\vec{\beta}$. $H$ can therefore be Cholesky-decomposed once, an $\mathcal{O}(p^3)$ operation, and the result can be used to solve the linear set of equations for each iteration, which is $\mathcal{O}(p^2)$ with a pre-Cholesky-decomposed matrix. The Cholesky-decomposition can be used instead of an LU-decomposition, reducing the

number of floating point operations by a factor of two, because $H = 2X^T X$ is positive definite when $X$ is non-singular.

Evaluation of the gradient only involves matrix-vector products, which is also an $\mathcal{O}(p^2)$ operation. The minimisation can, therefore, be done with one initial $\mathcal{O}(p^3)$ and then only $\mathcal{O}(p^2)$ operations per iteration, while the more general problem requires both the construction and the Cholesky-decomposition of $H$, an $\mathcal{O}(p^3)$ operation, for every single iteration.

This implementation gives both good performance and accurate results for small values of $\lambda$. Larger values, however, give rise to issues. The theory suggests that a large $\lambda$ should force some of the parameters $\beta_j$ to become zero, but the cost function is not differentiable, and certainly not double differentiable, when $\beta_j$ is zero. Consequently, Newton's method struggles to converge when some $\beta_j$s are zero in the true minimum of the LASSO cost function. Smarter methods such as coordinate descent[1] should therefore be used, as in e.g. scikit-learn.

## B.5 Minimisation methods

The important step in a regression method, or indeed in any statistical learning method, is to minimise the cost function. Certain cost functions, such as those of ordinary least squares and Ridge regression, admit analytical closed-form solutions for the parameters $\beta_j$ which minimise $Q(\vec{\beta}; D)$, while most methods, including LASSO regression and logistic regression, require usage of some general minimisation algorithm.

### B.5.1 Newton's method

Since the gradient of the cost function, $\nabla Q$, is a perfectly normal function from $\mathbb{R}^p$ to $\mathbb{R}^p$, it can be Taylor expanded around some point $\vec{\beta}_0$. This Taylor expansion can then be evaluated at a point $\vec{\beta}_0 + \delta\vec{\beta}$. To first order in $\delta\vec{\beta}$,

$$\nabla Q\Big(\vec{\beta}_0 + \delta\vec{\beta}\Big) = \nabla Q\Big(\vec{\beta}_0\Big) + H\Big(\vec{\beta}_0\Big)\delta\vec{\beta}, \tag{B.20}$$

where $H(\vec{\beta}_0)$ is the derivative of $\nabla Q$, i.e. the Hessian of $Q$, evaluated at $\beta_0$. Given a guess $\beta_0$ somewhere near the minimum of $Q$, a better estimate can be found by solving for the $\delta\vec{\beta}$ which makes $\nabla Q(\vec{\beta}_0 + \delta\vec{\beta}) = \vec{0}$, i.e.

$$H\Big(\vec{\beta}_0\Big)\delta\vec{\beta} = -\nabla Q\Big(\vec{\beta}_0\Big), \tag{B.21}$$

and, in general, solving

$$H\Big(\vec{\beta}_i\Big)\delta\vec{\beta}_i = -\nabla Q\Big(\vec{\beta}_i\Big), \tag{B.22}$$

letting $\vec{\beta}_{i+1} = \vec{\beta}_i + \delta\vec{\beta}_i$ and continuing the process until the method has converged sufficiently. The above equation is a simple linear set of equations.

### B.5.2 Gradient descent

Evaluating the Hessian matrix and inverting it can sometimes be infeasible, for example due to poor time complexity or being woefully undefined. In such cases, one can replace the Hessian $H$ with a number $1/\alpha$. Equation (B.22) on the previous page is then rewritten as

$$\vec{\beta}_{i+1} = \vec{\beta}_i - \alpha \nabla Q\left(\vec{\beta}_i\right), \tag{B.23}$$

with a simple geometric interpretation: By going a small step in the opposite direction of the gradient, the value of the cost function will decrease and $\vec{\beta}$ will approach the true minimum. A small step will guarantee convergence for a convex function at the cost of slow convergence, while a larger value may give faster convergence or not converge at all.

## B.6 Performance of regression methods

While the cost function is minimised by all regression methods, its value is not particularly meaningful. In particular, the cost functions in Ridge and LASSO regression depend on the parameter $\lambda$, and a measure independent of $\lambda$ should be used to determine the optimal $\lambda$. Other functions are therefore introduced to measure the performance of a given regression method and/or a choice of parameters.

The simplest measure is the mean square error,

$$\text{MSE}\left(\vec{\beta}, D\right) = \frac{1}{N}\left\|\vec{y} - \vec{\tilde{y}}\right\|_2^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - \tilde{y}_i)^2, \tag{B.24}$$

which should be as small as possible. Another measure is the $R^2$ score, defined as

$$R^2(\vec{\beta}, D) = 1 - \frac{\left\|\vec{y} - \vec{\tilde{y}}\right\|_2^2}{\left\|\vec{y} - \bar{y}\right\|_2^2} = 1 - \frac{\sum_{i=1}^{N}(y_i - \tilde{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}, \tag{B.25}$$

where $\bar{y}$ is the mean of the measured values. The $R^2$ score should be as close to 1 as possible.

Prediction and these measures of performance can be applied to two main types of data. Firstly, it can be applied to the data from which $\vec{\beta}$ was derived, which is called the training data. Ordinary least squares, corresponding to $\lambda = 0$ for the other methods, will, by definition, give the best results for this data set. Secondly, the performance can be measured for values not among the training data, called test data. Ridge and LASSO are expected to outperform ordinary least squares for small, non-zero values of $\lambda$ for this category of data.

## B.7 Resampling methods

Resampling methods are techniques to improve the prediction accuracy and obtain estimates for quantities such as the variance of $\vec{\beta}$ by repeatedly dividing the data set into training and test data. Two simple examples are $k$-fold cross-validation and bootstrapping. The former partitions the data set into $k$ partitions. One of these subsets is chosen as test data on which the performance is measured, while the rest are used as training data. This process is the repeated $k$ times, so that each subset is

used once as test data. Bootstrapping, on the other hand, repeatedly creates training data sets by randomly selecting $N$ values from the data set with replacement, while another, separate data set is used as test data each time, as illustrated by the snippet below.

```
!$omp parallel do private(fitter, tmp_real, indices, &
!$omp&                     y_selection, X_selection)
bootstraps: do i = 1, num_bootstraps
    if (.not. allocated(fitter)) fitter = self%fitter

    ! choose size(y_train) random indices
    call random_number(tmp_real)
    indices(:) = nint((N_train-1)*tmp_real) + 1

    ! select from training data
    y_selection(:)   = y_train(indices)
    X_selection(:,:) = X_train(indices,:)

    fitter%X = X_selection

    ! fit to selection
    call fitter%fit(y_values=y_selection)

    ! apply to test data, evaluate performance
    call fitter%predict(x_test, y_predictions(:,i), &
                        y_test, MSEs(i), R2s(i))

    betas(:, i) = fitter%beta
end do bootstraps
!$omp end parallel do
```

### B.8 Bias and variance

The choice of the number of basis functions, $p$, determines the complexity of the model to which the data set is fitted. A higher complexity makes the model a better approximation to the training data, which is said to reduce the *bias* of the model. On the other hand, a higher complexity may decrease the model's ability to predict reasonable values for test data, since a more complex model will be more affected by noise in the model. This is said to increase the model's *variance*. The best prediction performance is therefore achieved for a complexity which balances bias and variance, which is called the bias-variance trade-off[2].

Following [2], a mathematical manifestation of the bias-variance trade-off can be derived by assuming that the measured data set $D = \{(\vec{x}_i, y_i)\}_{i=1}^{N}$ is generated by a combination of some model $f(\vec{x})$ (for example Franke's function) and random noise, specifically

$$y_i = f(\vec{x}_i) + \varepsilon_i =: f_i + \varepsilon_i, \tag{B.26}$$

where the variables $\varepsilon_i$ are independent and normally distributed with variance $\sigma^2$ around zero. Given a data set $D$, a prediction $\tilde{y}_i^D$ can be made by any of the regression methods discussed above. An

expected mean square error can be found by averaging over all datasets $D$ and all noises $\vec{\varepsilon}$,

$$E_{D,\varepsilon}\left[\left\|\vec{y}-\vec{\tilde{y}}_D\right\|_2^2\right] = E_{D,\varepsilon}\left[\left\|\vec{y}-\vec{f}+\vec{f}-\vec{\tilde{y}}_D\right\|_2^2\right] = E_{D,\varepsilon}\left[\left\|\vec{y}-\vec{f}\right\|_2^2+\left\|\vec{f}-\vec{\tilde{y}}_D\right\|_2^2+2\left(\vec{y}-\vec{f}\right)\cdot\left(\vec{f}-\vec{\tilde{y}}_D\right)\right], \tag{B.27}$$

and using the linearity of the expectation value,

$$= E_{D,\varepsilon}\left[\left\|\vec{y}-\vec{f}\right\|_2^2\right]+E_{D,\varepsilon}\left[\left\|\vec{f}-\vec{\tilde{y}}_D\right\|_2^2\right]+2E_{D,\varepsilon}\left[\left(\vec{y}-\vec{f}\right)\cdot\left(\vec{f}-\vec{\tilde{y}}_D\right)\right] \tag{B.28}$$

$$= E_{D,\varepsilon}\left[\left\|\vec{\varepsilon}\right\|_2^2\right]+E_{D,\varepsilon}\left[\left\|\vec{f}-\vec{\tilde{y}}_D\right\|_2^2\right]+2E_{D,\varepsilon}\left[\vec{\varepsilon}\cdot\left(\vec{f}-\vec{\tilde{y}}_D\right)\right] \tag{B.29}$$

$$= \sigma^2+E_{D,\varepsilon}\left[\left\|\vec{f}-\vec{\tilde{y}}_D\right\|_2^2\right]+2\cancel{E_\varepsilon[\vec{\varepsilon}]}\cdot E_D\left[\left(\vec{f}-\vec{\tilde{y}}_D\right)\right] \tag{B.30}$$

$$= \sigma^2+E_D\left[\left\|\vec{f}-\vec{\tilde{y}}_D\right\|_2^2\right]. \tag{B.31}$$

This expression can be further decomposed by adding and subtracting the average prediction, $E_D\left[\vec{\tilde{y}}_D\right]$,

$$E_D\left[\left\|\vec{y}-\vec{\tilde{y}}_D\right\|_2^2\right] = \sigma^2+E_D\left[\left\|\vec{f}-E_D\left[\vec{\tilde{y}}_D\right]+E_D\left[\vec{\tilde{y}}_D\right]-\vec{\tilde{y}}_D\right\|_2^2\right] \tag{B.32}$$

$$= \sigma^2+\left\|\vec{f}-E_D\left[\vec{\tilde{y}}_D\right]\right\|_2^2+E_D\left[\left\|E_D\left[\vec{\tilde{y}}_D\right]-\vec{\tilde{y}}_D\right\|_2^2\right] \tag{B.33}$$

$$+2E_D\left[\left(\vec{f}-E_D\left[\vec{\tilde{y}}_D\right]\right)\cdot\cancel{\left(E_D\left[\vec{\tilde{y}}_D\right]-\vec{\tilde{y}}_D\right)}\right] \tag{B.34}$$

$$= \sigma^2+\left\|\vec{f}-E_D\left[\vec{\tilde{y}}_D\right]\right\|_2^2+E_D\left[\left\|E_D\left[\vec{\tilde{y}}_D\right]-\vec{\tilde{y}}_D\right\|_2^2\right]. \tag{B.35}$$

The first term, $\sigma^2$, represents the noise in the generated data, which no model can overcome. The second term measures the deviation of the average prediction from the true, noise-free model, which is the (squared) bias. Lastly, the third term measures how much the predictions vary and is called the variance. A complex model (large $p$) will minimise the second term, since a complex model will be better suited to fit the true model, $f$, while the third term will increase with model complexity since the fitting procedure will be more sensitive to noise in the different data sets.

Unfortunately, the above bias-variance decomposition requires knowledge of the exact model behind the data, $f$. A more computationally practical expression could have been derived by adding and subtracting $E_D\left[\vec{\tilde{y}}_D\right]$ in equation (B.27), which gives the expression

$$E_D\left[\left\|\vec{y}-\vec{\tilde{y}}_D\right\|_2^2\right] = \left\|\vec{y}-E_D\left[\vec{\tilde{y}}_D\right]\right\|_2^2+E_D\left[\left\|E_D\left[\vec{\tilde{y}}_D\right]-\vec{\tilde{y}}_D\right\|_2^2\right] \tag{B.36}$$

by manipulations analogous to the ones used above. This is another formulation of the bias-variance decomposition which can be used without any information about the underlying model. Dividing by $N$ gives the average MSE on the left-hand side.