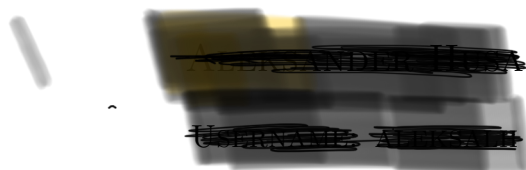


UNIVERSITY OF OSLO

Project 2

FYS-STK4155



✓ you
can
change
name
here

November 12, 2018

Contents

*you don't
need this,
but ok*

1	Introduction	3
2	Method	4
2.1	Ising Model	4
2.1.1	1D Model	4
2.1.2	2D Model	5
2.2	Linear Regression	5
2.2.1	Bias and Variance	6
2.3	Logistic Regression	7
2.3.1	Gradient Descent	8
2.4	The Multilayer Perceptron	9
2.4.1	Back Propagation	10
2.4.2	Momentum	11
2.4.3	Accuracy Score	11
2.4.4	Cross Validation	12
2.4.5	Earlystopping	12
2.4.6	Initializing Weights	12
3	Results	13
3.1	1D Ising Model	13
3.1.1	Linear Regression	13
3.1.2	Multilayer Perceptron	19
3.2	2D Ising Model	21
3.2.1	Logistic Regression	21
3.2.2	Multilayer Perceptron	23
4	Discussion	24
5	Conclusion	25
6	Appendix	26
6.1	GitHub	26
6.2	Deriving the Back Propagation	26
6.3	Source Code	28
6.3.1	Linear Regression (part b)	28
6.3.2	Logistic Regression (part c)	38
6.3.3	MLP for regression (part d)	44

6.3.4	MLP for classification (part e)	54
	References	66

Abstract

We explored the use of different methods for solving linear regression and classification on the 1D and 2D Ising models, respectively. We used the OLS, ridge and lasso methods for linear regression on the 1D Ising model, where the goal was to estimate the energy of the model using the spins as inputs. We also created a multilayered perceptron (MLP) to solve the linear regression on the 1D model. The lasso method, given the correct regularization parameters, performed the best on the linear regression case for the 1D Ising model. The MLP did not perform as well as the lasso method. The 2D Ising model was used for the classification. We used the spin configuration of the models to estimate the phase of the Ising model. It was attempted to classify the phase of the 2D model by using logistic regression methods, but these did not perform well. The MLP was trained on the 2D model and performed far better than the logistic regression. This project showed the importance of choosing the right algorithms to solve right problems.

1 Introduction

A constant problem in machine learning is to balance the complexity of the algorithms with their computation time. We always want the best results possible, but the computation time is often a limiting factor. While more complex methods often give great results, their computation time may be long. Simpler methods is often faster, but they might give worse results. This is the most intuitive explanation, but is it always true?

The goal of this project is to explore the use of different regression methods and neural networks. We will use logistic and linear regression as well as a multilayer perceptron (MLP) to solve classification and regression problems. The models will be tested on the 1D and 2D Ising models, commonly used in a variety of different fields of research. The inputs of the created models will be the spins in the Ising models. We will estimate the energy of the 1D model using our regression models and try to train our MLP to perform the same regression. MLP and logistic regression models are also designed to classify the phases in the 2D Ising model. Using different methods to solve problems on the same models give us an opportunity to see which methods

performs best.

We will first describe the methods and models used. The results will be presented after that, and a discussion and conclusion will be at the end of the report.

2 Method

2.1 Ising Model

The Ising model is a binary model which is used in several scientific disciplines. It was originally created to represent ferromagnetism. The values of the models can therefore represent atomic spins. We will use this terminology in this report. Each variable in the model can have 2 different values (+1 or -1 in this project) which represents the spins. These spins are normally arranged in a lattice formation which allows each spin to interact with their neighbours. The goal of this project is to explore the use of different machine learning techniques. The relatively simple form of the Ising model and the fact that it is well researched and widely used makes it ideal for this project. The Ising model can, in its simplest form, be written as:

$$E = -J \sum_{\langle kl \rangle}^N s_k s_l \quad (1)$$

E describes the energy of the system and J is a coupling constant which describe the strength of interaction between neighbours. The $\langle kl \rangle$ indicates that we only sum over the nearest neighbours and $s_k = \pm 1$ represents the spins. We will look at both the 1D and 2D models in this project. [5]

2.1.1 1D Model

The 1D Ising model with nearest neighbours interaction can be described as:

$$E = -J \sum_{j=1}^N s_j s_{j+1} \quad (2)$$

We can see that this is a chain where we sum the interactions between the neighbours in the chain. The data for the 1D model will be generated

with $J = 1$ and we will use regression to find J .

2.1.2 2D Model

The 2D Ising model is one of the simplest models that can be used to describe phase transitions. The spins will more often be aligned at lower temperatures and the system will be in an ordered phase. The system will enter a disordered phase over a critical temperature. There is a critical region around the critical temperature where the ferromagnetic correlation length diverges. This makes it harder to estimate phases using this data and we will not train our models using this data. We will, however, use the critical data to assess our models.

2.2 Linear Regression

We will use the same linear regression methods as in project 1. We will use the OLS, ridge and lasso methods to estimate the energy of a 1D Ising model and the bootstrap method will be used to assess the models. We will not describe these methods in detail as this is covered in project 1. The β parameters will be equivalent to the interaction constant, J , when we use the linear regression methods.

We will generate the Ising model data using the following code:

```
1 ##### define Ising model aprams
2 # system size
3 L=40
4 # create 10000 random Ising states
5 states=np.random.choice([-1, 1], size=(10000,L))
6 def ising_energies(states,L):
7     """
8     This function calculates the energies of the states in the
9     nn Ising Hamiltonian
10    """
11    J=np.zeros((L,L),)
12    for i in range(L):
13        J[i,(i+1)%L]=-1.0
14    # compute energies
15    E = np.einsum('...i,ij,...j->...',states,J,states)
16    return E
17 # calculate Ising energies
18 energies=ising_energies(states,L)
```

This code generates the energies of 1000 1D configurations with a coupling constant of $J = -1$ and only nearest neighbour interactions. There are 40 spins in this configuration. The task for the linear regression methods will be to find the J -value between the spins.

As we do not assume that only neighbours have interactions, we instead use the more general assumption

$$E_{model}[s^i] = - \sum_{j=1}^N \sum_{k=1}^N J_{j,k} s_j^i s_k^i \quad (3)$$

where i represents one of the 1000 configurations generated. Equation 3 sums over all the spins and allows for interactions between all the spins in the configuration. The regression will learn $40 \times 40 = 1600$ coupling constants, but we assume that the constants describing interactions between neighbours will be magnitudes larger than the others. We can describe the energy of the model in matrix form as

$$E_{model}^i \equiv \hat{X}^i \cdot \hat{J} \quad (4)$$

where \hat{X}^i is all the interactions between the spins and J is the coupling constants to be estimated. We expect to learn negative values of J since the model does not have a negative J . To generate the \hat{X} , which contains all the configurations generated, we will use

```
1 states=np.einsum('...i,...j->...ij', states, states)
```

where the input states are the spin configurations generated and the np.einsum function creates a 2D matrix with all the interactions between the spins. The output states are then split into training and test data. We can use the code from project 1 with the output energies generated and the \hat{X} defined.

2.2.1 Bias and Variance

We will use bootstrapping to find the bias and variance of our linear regression models. We will use the following definition:

$$MSE = E[(y - \hat{f}(x))^2] = (E[f(x) - \hat{f}(x)]^2) + (E[\hat{f}(x)^2] - E[\hat{f}(x)]^2) + \sigma^2 \quad (5)$$

The first term is the bias of the model squared. The second term is the variance and the last term is the irreducible error of the model. This error comes from noise and it cannot be removed completely.

2.3 Logistic Regression

We aim to classify the spin configurations with the logistic regression. The goal is to use spin configurations from a 40×40 2D Ising model and predict the phase of the model (ordered or disordered). We will use data from [Mehta et al, arXiv 1803.08823](#). This data contains spins of ordered, disordered and critical phases and labels to classify the data.

We will use the data and the labels to train our logistic model to predict the phase of the model based on the spins. A sigmoid function will be used to classify the data. Since we have two classes, our target can either be $y_i = 0$ or $y_i = 1$. The output of the sigmoid function gives the probability of $y_i = 1$. The function can be written as

$$p(y_i = 1|x_i\hat{\beta}) = \frac{1}{1 + \exp(-(\beta_0 + \beta_1 x_1))} \quad (6)$$

where x_i is the input and $\hat{\beta}$ only has 2 parameters. This can be extrapolated to include more inputs. The goal of the model is to match the predicted values (p) with the given labels for the set (y_i) using a given spin configuration (1600 values of x) and the learnable parameters ($\hat{\beta}$). We will use the cross entropy cost function to learn the β -parameters. This cost function C with regards to $\hat{\beta}$ can be written as (with the same number of inputs as in 6)

$$C(\hat{\beta}) = - \sum_{i=1}^n (y_i(\beta_0 + \beta_1 x_i) - \log(1 + \exp(\beta_0 + \beta_1 x_i))) \quad (7)$$

where i represents one of the configurations of spins. The cost function can also be fitted with a regularization term. We will fit it with a $L2$ regularization term in this project, which can be written as:

$$\lambda \sum_{i=1}^n \beta_i^2 \quad (8)$$

This cross entropy function is a convex function, and a local minima is therefore a global minima. If we minimize this cost function and write it with a matrix notation we get:

$$\frac{\partial C(\hat{\beta})}{\partial \hat{\beta}} = -\hat{X}^T(\hat{y} - \hat{p}) \quad (9)$$

\hat{y} is an n -length vector with the targets for the classifications, \hat{p} is an n -length vector with the $p(y_i|x_i,\hat{\beta})$ probabilities and \hat{X} is an $n \times m$ matrix containing the x_i values. m will be the number of inputs which in our case will be 1600.[1]

2.3.1 Gradient Descent

We will use gradient descent to minimize the cost function. Gradient descent is a method that follows the gradient to a local minima of a function. We want to minimize the cost by adjusting the adjustable parameters. The gradient descent can be written as:

$$x_{k+1} = x_k - \eta_k \nabla F(x_k) \quad (10)$$

x is the value to be optimized, ∇F is the gradient of the function and η is the learning rate which controls how fast the method learns new values. The method will converge slowly if the learning rate is too low and it will be irregular if the learning rate is too high. This method will converge slowly if we update the parameters after every input(sequential training), but it will escape local minima relatively easy. This means that this method explores many possible solutions, but it might discard a good solution. Another solution is to update the weights after all training data is used (batch method). This will make the method follow the gradient which most of the data "agrees" on. This will lead to a faster convergence, but the batch method is more likely to get stuck in local minima.

We will use a minibatch method in this project. This method balances the sequential and batch training methods. This means that we run through a number of the test data and then update the parameters. Then we do this for another minibatch until all the training data is used. After that, we reshuffle the data and do it again. One of these runs is called an *epoch*. We will run

a number of epochs until no improvements are shown. Our gradient descent for one minibatch, using equation 9, will be

$$\hat{\beta}_{n+1} = \hat{\beta}_n - \eta \frac{1}{N} \sum_i^N \hat{X}_i^T (\hat{y}_i - \hat{p}_i) \quad (11)$$

where N is the size of the minibatch. Using the average instead of the sum allows the use of the same learning rate between minibatch sizes. The minibatches gets randomly chosen at each epoch.

2.4 The Multilayer Perceptron

We will use a multilayer perceptron (MLP) to try to solve both a regression and a classification problem. The MLP is a simple feed-forward neural network. This means that the information only moves in one direction (forward). The MLP contains several perceptrons (or nodes) in at least 3 layers (2 layers with perceptrons). These 3 layers are called *input*-, *hidden*- and *output*-layers. More hidden layers can be added, but we will only use one hidden layer in this project. The perceptrons take the sum of all its inputs multiplied by weights, run the sum through an activation function and then gives an output based on the inputs. We will use the sigmoid function from equation 6 in this project as an activation function. A biological perceptron gives a value of either on or off. We can not use this functionality since we need a derivable function to teach the network. The sigmoid function works well since it gives a value between 0 and 1. The mathematical function of a node can be written as

$$y = f\left(\sum_{i=1}^n w_i x_i + b_i\right) \quad (12)$$

where y is the output of the node, f is an activation function, w are the weights of the inputs, x are the inputs and b is the bias for that node. The learnable bias makes it possible for the node to have a value even if all inputs are 0. The weights are the parameters the network adjust to learn.

The perceptrons in the hidden layer feed their outputs to the perceptrons in the output-layer, and the result of the network is calculated. We will use a linear activation function on the output nodes for the regression problem

and the same sigmoid function as equation 6 for the classification problem. Feed-forward function of our network can therefore be written as:

$$y_i = f^2 \left[\sum_{j=1}^M w_{ji} f^1 \left(\sum_{k=1}^K v_{jk} x_k + b_j^1 \right) + b_i^2 \right] \quad (13)$$

y_1 represents the outputs of one of the output nodes and x are the inputs of the network. f^2 can either be a linear or a sigmoid activation function, while f^1 is always a sigmoid function. v are the weights on the inputs given to the hidden layer and w are the weights between the hidden layer and the output nodes. K are the number of inputs and M are the number of nodes in the hidden layer. The b 's are learnable biases. [2]

2.4.1 Back Propagation

The challenge with a MLP is finding a way to adjust the weights in the hidden layers. We can easily find the error of the output, but it is harder to find out which nodes in the hidden layers contributed to that error. The back propagation was developed to solve this problem. The goal of the back propagation is to distribute the error of the output between all the different nodes. The back propagation is derived in section 6.2. We assume that all nodes in the same layer has the same activation function. The error of the outputs for a sigmoid activation function is

$$\delta_o^i = (y_o^i - t^i) y_o^i (1 - y_o^i) \quad (14)$$

where i denotes the different output nodes. y_o^i is the output of the network and t_1 is the target the network is trying to reach. Equation 14 will be replaced by $\delta_o^i = (y_i - t_i)$ if we use a linear activation function. The error of the hidden layer can then be calculated by

$$\delta_h^j = y_h^j (1 - y_h^j) \sum_{i=1}^N w_{ji} \delta_o^i \quad (15)$$

where y_h^j are the outputs of the nodes. We can see that we estimate the error of each node in the hidden layer by multiplying the error of the output with the weight between the hidden node and the output. We get the error of the

hidden node by taking the sum of all these weighted errors. [3] We can then update the weights of the output by:

$$w_{ji} \leftarrow w_{ji} - \eta \delta_o^i y_h^j \quad (16)$$

We update the weight of the hidden layer by:

$$v_{jk} \leftarrow v_{jk} - \eta \delta_h^j x_k \quad (17)$$

We also used minibatches in the MLP. The updated weights are

$$\hat{w} \leftarrow \hat{w} - \eta \frac{1}{N} \hat{\delta}_o \hat{y}_h \quad (18)$$

$$\hat{v} \leftarrow \hat{v} - \eta \frac{1}{N} \hat{\delta}_h \hat{x} \quad (19)$$

where N is the size of the minibatches. $\hat{\delta}_o$ will be a $P \times N$ matrix, where P is the number of outputs. \hat{y}_h will be a $N \times M$, where M is the number of nodes in the hidden layer. $\hat{\delta}_h$ will be an $M \times N$ matrix. \hat{x} will be a $N \times K$ matrix, where K is the number of inputs to the network. The biases are updated at the same time by adding them to the weight matrices, but they are not used to calculate the back propagation to the hidden layer.

2.4.2 Momentum

The MLP runs are implemented with momentum. This is implemented to help the network escape local minima. The momentum adds a term to the weight update so that the network is more likely to continue in the same direction as the previous update. The momentum is the same for both of the layers and, for the output layer, it can be written as

$$w_{ji} \leftarrow w_{ji} - \eta \delta_o^i y_h^j + \alpha \Delta w_{ji}^{t-1} \quad (20)$$

where Δw_{ji}^{t-1} is the previous change to the weight and α is an adjustable parameter to adjust the impact of the momentum.

2.4.3 Accuracy Score

We will use an accuracy score to asses the classification models. The score is calculated by

$$Accuracy = \frac{\sum_{i=1}^n I(t_i = y_i)}{n} \quad (21)$$

where I is the indicator function. This means that $I = 1$ if $t_i = y_i$ and zero otherwise. t_i is the target and y_i is the output of our MLP during the classification. This will give a value between 0 and 1, and will be the percentage of correct classifications if we multiply it by 100. 100% accuracy indicates that all the classifications are correct.

2.4.4 Cross Validation

We will use a 10-fold cross validation to assess our multilayer perceptrons, where we divide our data into 10 folds. 1 fold will be used for testing, 1 fold for validation and 8 folds for training. The test fold is the fold we use to test our final model in each run. The validation set is used during the training (but not used to train the model) to determine if we can stop the training of the model. The goal is to stop training before we overfit our model too much. The last 8 folds is used to train our model. The model is trained and assessed 10 times, where each fold is used for testing once.

2.4.5 Earlystopping

Both the MLP's created for this project is fitted with an earlystopping function. This function assess the model every 10 epochs using the validation set. The values of the best weights are always saved. The training is stopped if this function finds that no improvements are made (or if the models get worse) after a given number of epochs. The best weights are used on the final model.

2.4.6 Initializing Weights

It is important to initialize the weights in the MLP in a good way. We might get unstable solutions if the weights are too large or too small. We have taken a random number between $-\frac{1}{\sqrt{n}}$ and $\frac{1}{\sqrt{n}}$ to initialize the weights. n is the number of inputs to the layer. [3]

3 Results

3.1 1D Ising Model

We tried to use both linear regression and MLP to solve the regression problem of the 1D Ising model. We tried to estimate the energy of the model based on the spins and we assumed an interaction between all of the spins (even though only neighbouring spins had any interaction). The β -parameters can be seen as the interaction constant, J , when we use linear regression. There is not any such equivalence when we use the MLP to solve the 1D-model and we can therefore only assess the MLP with regards to the estimated energy.

3.1.1 Linear Regression

We got good results when using the linear regression methods on the 1D Ising model. Figure 1 and 2 show the J -constant between the spins. We can see that all methods show a stronger connection between the neighbours. We can see that this connection is "split" between two variables for the ridge and OLS methods (there is a connection constant between spin 1 and 2 and between spin 2 and 1), while the Lasso method shows one connection value between each neighbour. This is closer to the model we are trying to estimate, since we only have one J between spins.

We can also see that the ridge and OLS method perform similarly for lower values of λ . The ridge method gets worse when λ gets large. The lasso method has a near perfect J -constant when $10^{-3} \leq \lambda \leq 10^{-1}$

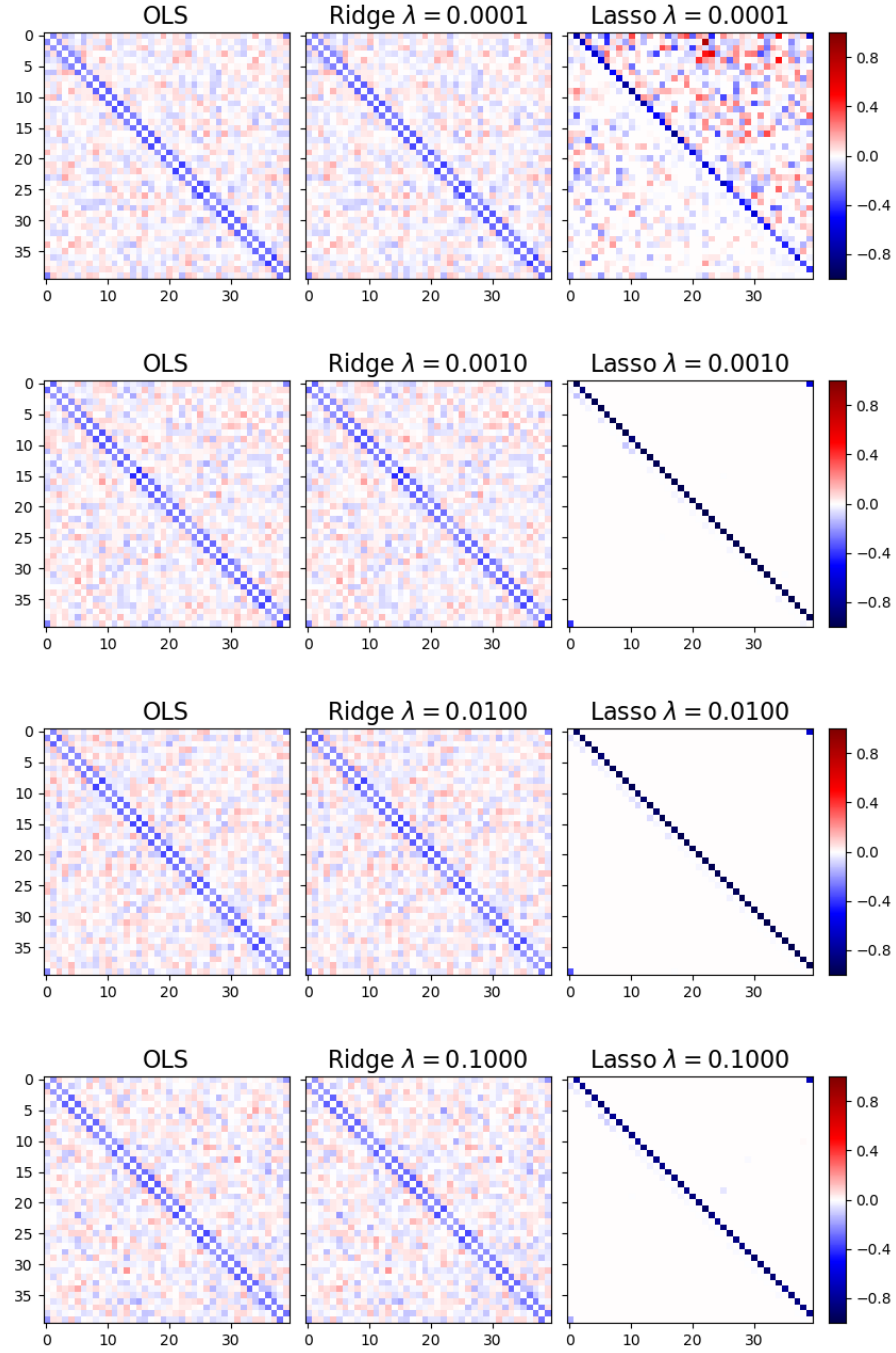


Figure 1: The estimated J -values (between all spins) for different values of λ .

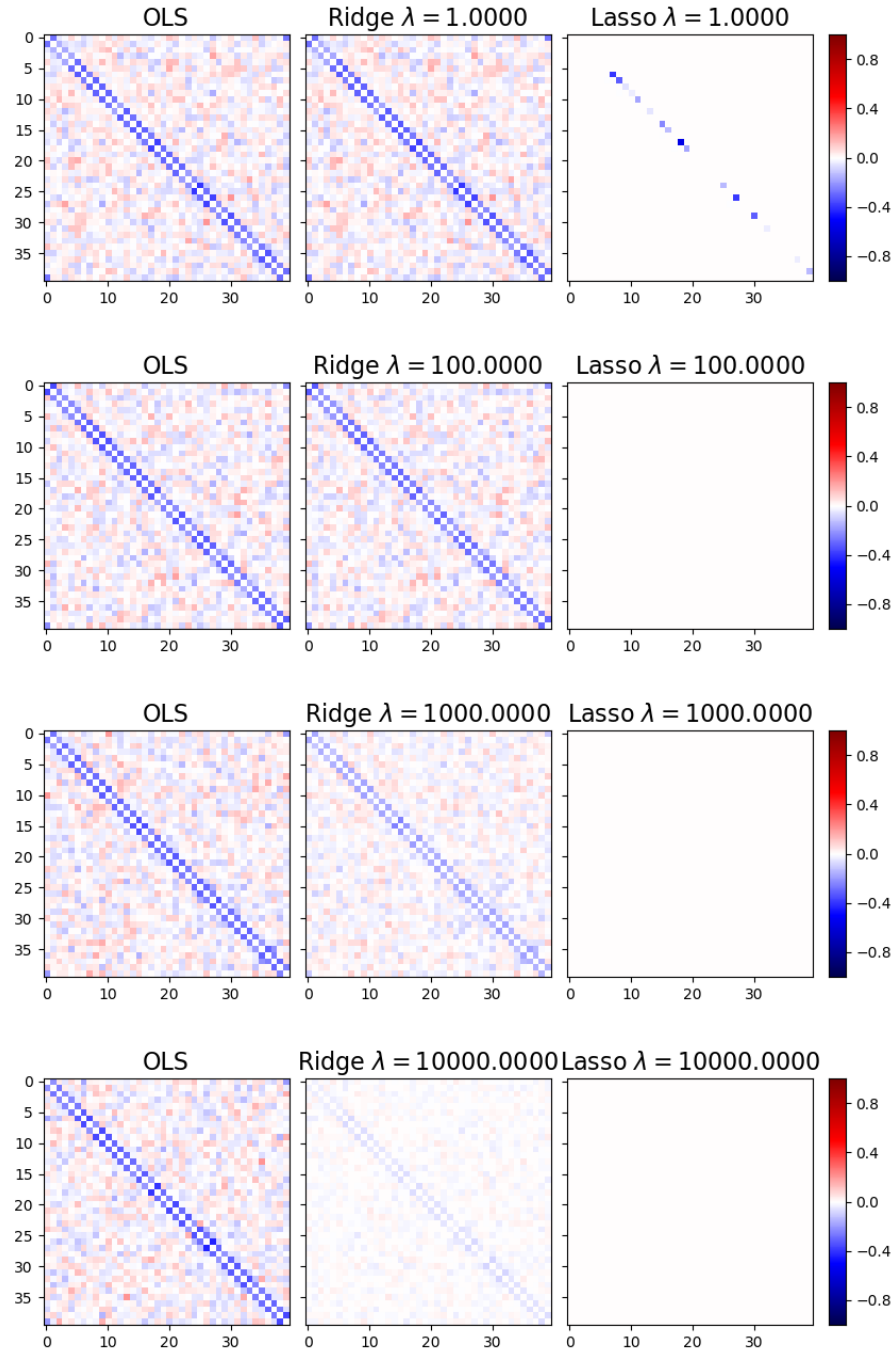


Figure 2: The estimated J -values (between all spins) for different values of λ .

Figure 4 shows the R^2 -scores and figure 5 shows the MSE -scores for the different methods for different values of λ on both training and test data. The ideal R^2 -score is 1 and the ideal MSE -score is 0. These figures and figure 1 and 2 show the same results regarding which method is the best.

We see that the lasso method performs near perfectly for $\lambda = 10^{-2}$ before it gets worse. The OLS and ridge methods perform similarly (until λ gets large) and very well on the training data, but perform badly on the test data. This indicates that these models overfit the data. Additionally, this result tells us that the fact that the lasso method sets irrelevant values to zero greatly improves the model. This seems reasonable since our model estimates the energy using interactions between all spins. Therefore, setting the interactions that are not between neighbours to zero will more closely represent the real model. These observations are supported by figure 3. We can see that bias and variance are almost zero when the lasso method uses the correct λ -value. We can also see that the variance decreases and the bias increases as the model becomes too simple for the higher values of λ . The ridge method gets higher bias when $\lambda > 100$

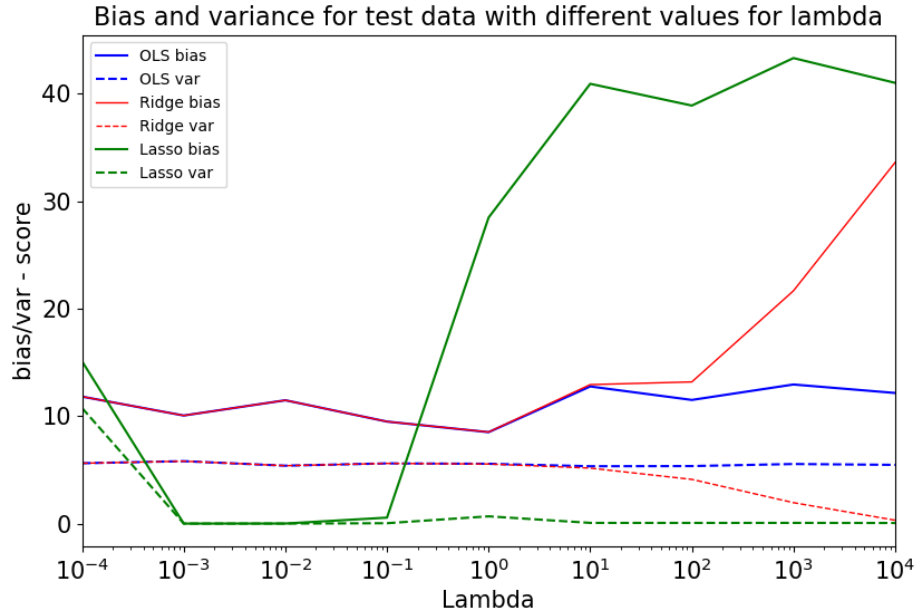


Figure 3: $Bias^2$ - and $variance$ -scores for test data for different values of λ .

The results from figure 1, 2 and 4 compares well with the result from [Mehta et al, arXiv 1803.08823](#), which can be found in their [jupyter notebook](#).

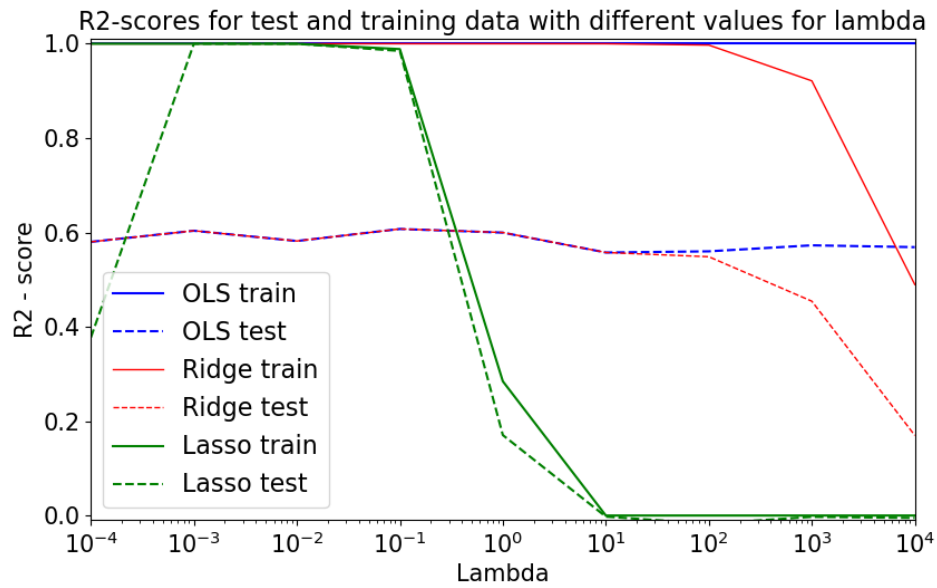


Figure 4: R^2 -scores for test and training data for different values of λ .

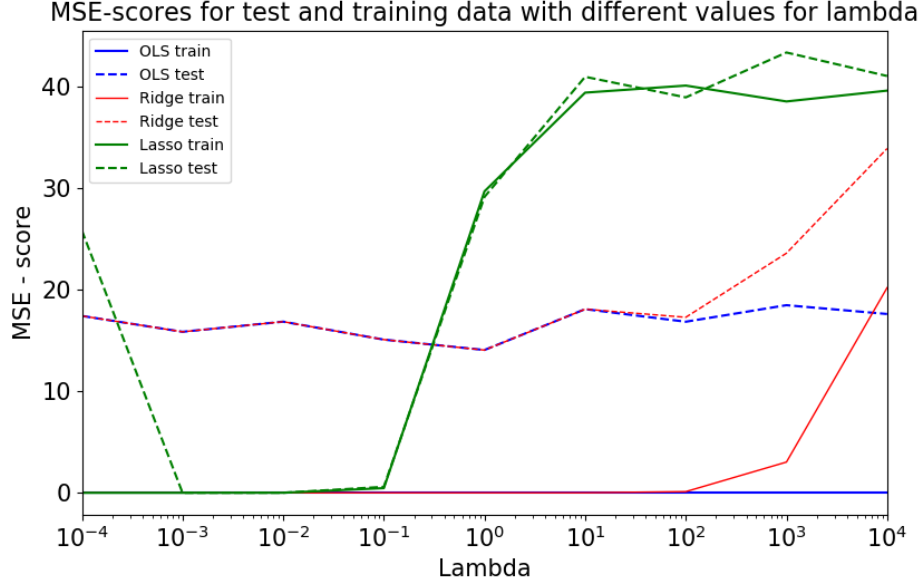


Figure 5: MSE -scores for test and training data for different values of λ .

3.1.2 Multilayer Perceptron

We also used a MLP to estimate the energies in the 1D Ising model. The MLP parameters can be found in table 1. Further tweaking of these values could improve the results, but these values gave the best results from the ones we tried.

Table 1: Parameters used for the MLP when used on the 1D Ising model

Learning rate (η)	0.01
Momentum (α)	0.5
Hidden layers	1
Input nodes	1600
Hidden nodes	20
Output nodes	1
Minibatch size	50

We ran a keras neural network to compare it to our result. Both the created MLP and the keras models performed very well on the training data

and poorly on the test data. This is an indication that the models overfit the data a lot. We can see that neither network provides great results from figure 6 and 7. Keras performs better than our MLP on training data, but it performs worse on the test data. This indicates that keras overfit the data even more than our MLP. The keras model also runs slower than the created MLP. This was improved by also fitting the keras model with earlystopping, but it did not improve the results from the model much.

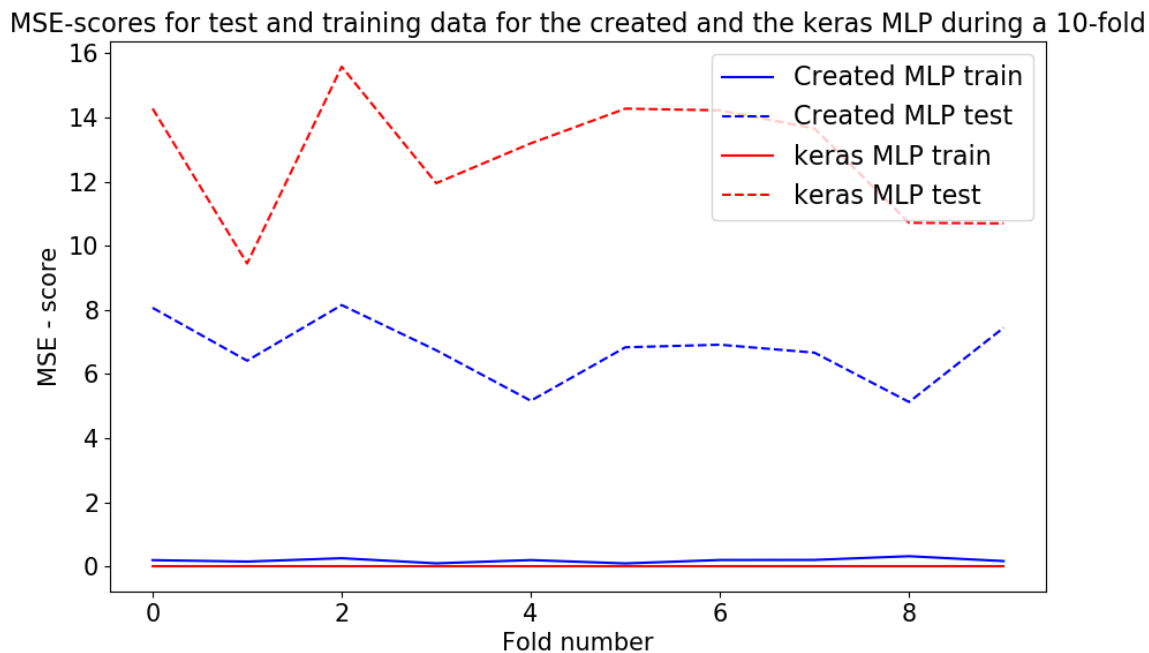


Figure 6: MSE -scores for test and training data for different runs of the 10-fold.



Figure 7: R^2 -scores for test and training data for different runs of the 10-fold.

3.2 2D Ising Model

The goal of the 2D Ising model was to classify the phases of the data using the spin configurations. We tried to use both logistic regression and our MLP to classify the data. The critical phase data was not used to train the models, but was used for assessment.

3.2.1 Logistic Regression

The logistic regression method did not manage to estimate the phases well. Several parameters were tested, but we finally used the ones described in table 2. We used a very low learning rate as a too high η did not manage to converge.

Table 2: Parameters used for logistic regression on the 2D Ising model

Learning rate (η)	0.00001
Inputs	1600
Outputs	1
Minibatch size	20

Figure 8 shows that the models perform poorly on the training, test and critical data. Neither the created models nor the logistic regression models from Scikit-learn manage to classify the data in a good way. It would therefore seem that the logistic regression method does not manage to classify the phases well. The figure also shows that the created method with L2 regularization shows a small improvement around $\lambda = 1$, but the performance becomes worse when $\lambda > 10$. The results from Scikit compares with the bilinear results in [Mehta et al, arXiv 1803.08823 jupyter notebook](#).

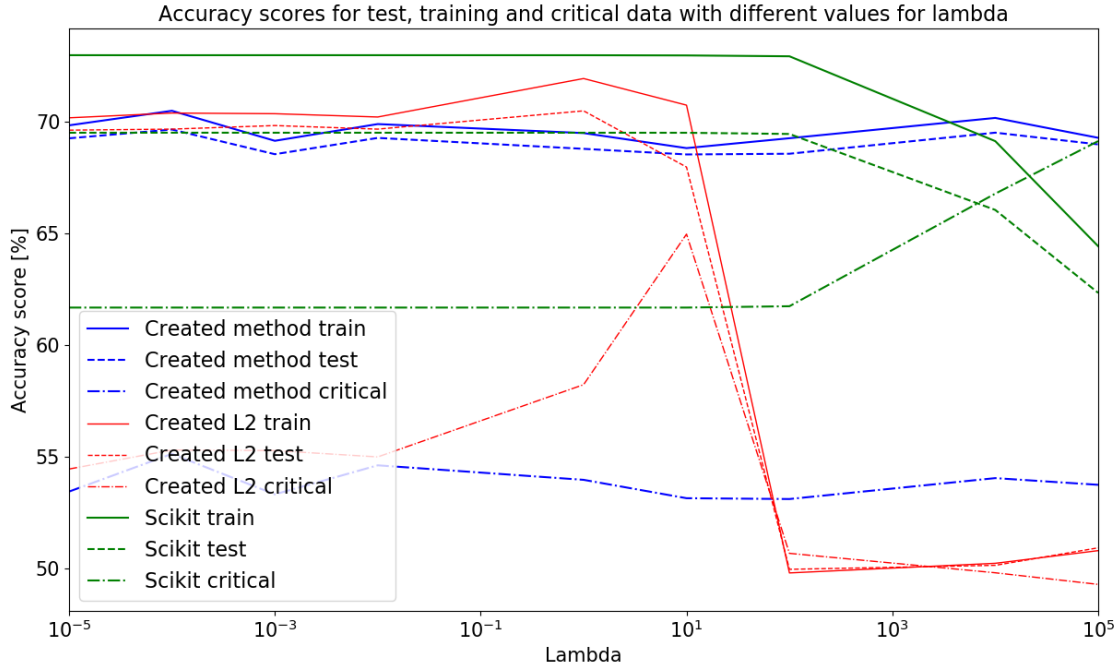


Figure 8: Accuracy scores for test, training and critical data for different values of λ .

3.2.2 Multilayer Perceptron

We also tried to use our MLP to classify the data and it performed much better than the logistic regression. The parameters used can be found in table 3. A keras network was used to compare the results. Both the created and the keras network managed to classify the data well. They also performed relatively well on the critical data.

Table 3: Parameters used for the MLP when used on the 2D Ising model

Learning rate (η)	0.1
Momentum (α)	0.7
Hidden layers	1
Input nodes	1600
Hidden nodes	12
Output nodes	2
Minibatch size	20

Figure 9 shows that both models perform well over the folds, but the created MLP is slightly better at predicting the critical data. This could probably be fixed by adjusting the parameters of the keras MLP even more.

Accuracy scores for test, training and critical data for the created and the keras MLP during a 10-fold

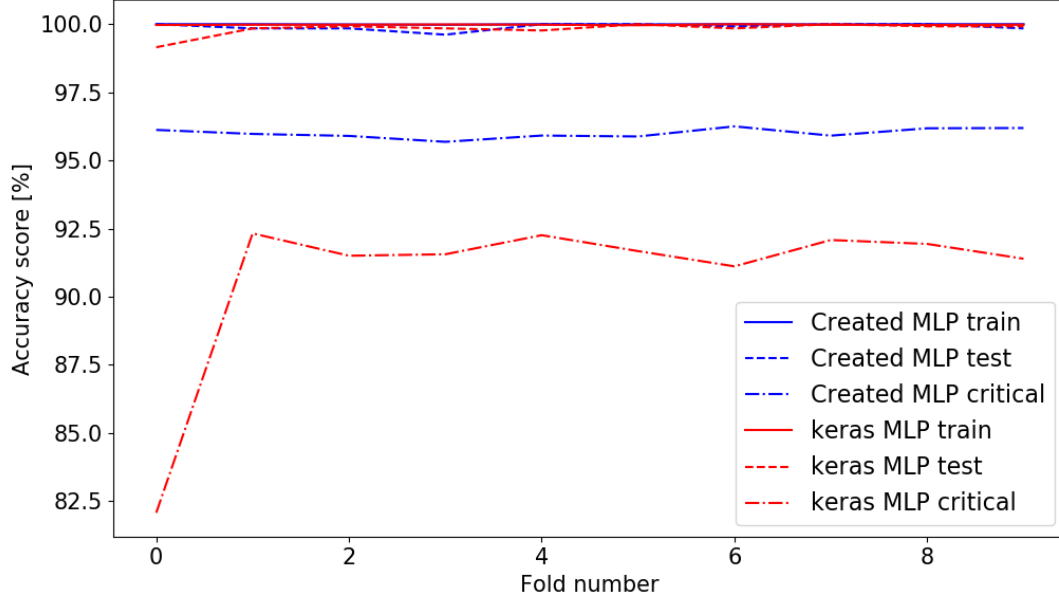


Figure 9: Accuracy scores for test, training and critical data for the created MLP and the keras MLP.

4 Discussion

We see from the results that the lasso method (with the right value of λ) outperforms the MLP for the 1D Ising model. This indicates that a linear function is enough to estimate the energy of the model. The MLP can learn more complex functions, but struggled to learn the simpler 1D models. We would predict that a network without a hidden layer would perform better than the MLP's used in this project, as they can only estimate linear functions and would perform better when used for linear regression.

The multilayer perceptrons outperformed the logistic regression on the 2D Ising model. It managed to get a near perfect estimation of the classes on the test data, and classifications of the critical data was relatively good. The created MLP was relatively fast and consistent. This indicates that the classification problem cannot be solved in a good way with a linear classifier, like

the logistic regression method, but a more complex model is needed. The number of inputs could be the reason for needing an especially low learning rate for the logistic regression model. It is reasonable that the output would vary greatly if a too large adjustment is made to the parameters.

The keras module from tensorflow was used for comparison in this project. The keras models performed worse than the created models for both the 1D and the 2D Ising models. They were also much slower than the MLP's that were created for this project. Fitting the keras models with earlystopping made them much faster, but it reduced the accuracy somewhat. The keras models are well designed machine learning models with many parameters and optimizers to adjust. It is therefore safe to assume that the keras models would greatly outperform the models created for this project, even if more time was spent on optimizing the keras models correctly.

Further work on this project could include the optimization of the keras models for a better comparison. Further adjusting of the created models could also improve the results found in this projects. There is also possible to explore more methods and neural networks to find ones that fit the problems in this project even better.

5 Conclusion

This project has made it clear that fitting the right solver to the right problem is critical when it comes to getting the best results. It is tempting to implement a neural network to any data and expect it to solve the problem in the best way, but this is clearly not always the best approach. It is probable that a well designed neural network could solve the 1D Ising model in a good way, but it is not necessary to design it when a simple linear regression model solves it near perfectly. A good understanding of the data and the problem that needs to be solved is therefore imperative when deciding which methods to use. This understanding will stop us from using more complexity and computation when less is needed.

6 Appendix

6.1 GitHub

The source code and test results can be found at the address ~~https://github.com/...~~

6.2 Deriving the Back Propagation

The back propagation is what allows us to train our multilayered network. We start by simply defining the error of our network as:

$$C(\hat{W}) = \frac{1}{2} \sum_{i=1}^n (y_i - t_i)^2 \quad (22)$$

Here we have n inputs and y_i are the outputs we get from our network by running it forwards. The t_i are the targets for our network to match. We can write the output (before activation) of a node as:

$$z_i^l = \sum_{j=1}^M w_{ij}^l a_j^{l-1} + b_j^l \quad (23)$$

l denotes the different layers, w are the weights, a_i^{l-1} are the outputs from the previous layer, b_j^l are the bias for the node and M are the number of nodes in the previous layer. We have used the a sigmoid function as our activation function and we will use the same function for all the layers in this derivation. The output of a node will therefore be:

$$a_j^l = f(z_j^l) = \frac{1}{1 + \exp(-z_j^l)} \quad (24)$$

We want to figure out how a change in the weights of the layers impact the output of the network. We will therefore look at the output layer ($l = L$) and from equation 22 we get:

$$C(\hat{w}^L) = \frac{1}{2} \sum_{i=1}^n (a_i^L - t_i)^2 \quad (25)$$

Now we want to know how this cost function changes with regards its weights. Applying the chain rule and since only the j set of weights impacts the j output we get:

$$\frac{\partial C(\hat{w}^L)}{\partial \hat{w}_{jk}^L} = (a_j^L - t_j) \frac{\partial a_j^L}{\partial \hat{w}_{jk}^L} \quad (26)$$

If we apply the chain rule to the last part we get:

$$\frac{\partial a_j^L}{\partial \hat{w}_{jk}^L} = \frac{\partial a_j^L}{\partial \hat{z}_j^L} \frac{\partial \hat{z}_j^L}{\partial \hat{w}_{jk}^L} \quad (27)$$

If we apply these two derivatives to equation 24 and equation 23 we get:

$$\frac{\partial a_j^L}{\partial \hat{z}_j^L} \frac{\partial \hat{z}_j^L}{\partial \hat{w}_{jk}^L} = a_j^L (1 - a_j^L) a_j^{L-1} \quad (28)$$

Now if we combine equation 26 and equation 28 we get

$$\frac{\partial C(\hat{w}^L)}{\partial \hat{w}_{jk}^L} = a_j^L (1 - a_j^L) (a_j^L - t_j) a_j^{L-1} \quad (29)$$

We define:

$$\delta_j^L = a_j^L (1 - a_j^L) (a_j^L - t_j) \quad (30)$$

Which is the error term of the j^{th} output. $\delta_j^L = (a_j^L - t_j)$ if we have a linear output. Since the derivative of the linear function is 1. We can write:

$$\frac{\partial C}{\partial \hat{w}_{jk}^L} = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial \hat{z}_j^L} \frac{\partial \hat{z}_j^L}{\partial \hat{w}_{jk}^L} \quad (31)$$

We know from 29 and 28 that $\frac{\partial \hat{z}_j^L}{\partial \hat{w}_{jk}^L} = a_j^{L-1}$. Using the term we got from equation 30 we get:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial \hat{z}_j^L} \quad (32)$$

The error of our general layer can be written as:

$$\delta_j^l = \frac{\partial C}{\partial \hat{z}_j^l} \quad (33)$$

We want to describe the error in terms of the next layer. The error of the node is also a function of all the error it contributed to in the next layer

(based on the weights between the nodes). This means we write the error of the node as a sum:

$$\delta_j^l = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial \hat{z}_j^l} \quad (34)$$

Combining this with equation 23 and 33 we get:

$$\delta_j^l = \sum_k \delta_j^{l+1} w_{kj}^{l+1} f'(z_j^l) \quad (35)$$

We can use the methods in section 2.4.1 now that we have the equations for the error terms of the nodes.

6.3 Source Code

6.3.1 Linear Regression (part b)

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3 from sklearn import metrics
4 from sklearn import linear_model
5 import random
6 import time as tm
7 from mpl_toolkits.axes_grid1 import ImageGrid
8
9
10 #perform the OLS regression
11 def OLS (X,Xte,zt,trainn):
12     """
13     Calculates the OLS regression using a polynomial.
14     Inputs: Train data, test data, training outputs and number
15     of training data.
16     Outputs: Beta parameters, predicted z on train data and z
17     predicted on test data.
18     """
19     #using SVD to find the pseudo inverse
20     u, s, vh = np.linalg.svd(X, full_matrices=False)
21     beta = vh.T @ np.linalg.pinv(np.diag(s)) @ u.T @ zt
22     zpred = X.dot(beta)
23     zpredtest = Xte.dot(beta)
24     return beta, zpred, zpredtest
25
26 #perform the ridge regression

```

```

25 def ridge (X,Xte,zt,lam,trainn):
26     """
27     Calculates the ridge regression using a polynomial.
28     Inputs: Train data, test data, training outputs, lambda
29     matrix and number of training data.
30     Outputs: Beta parameters, prediced z on train data and z
31     predicted on test data.
32     """
33     beta = np.linalg.inv(X.T.dot(X) + lam).dot(X.T).dot(zt)
34     zpred = X.dot(beta)
35     zpredtest = Xte.dot(beta)
36     return beta, zpred, zpredtest
37
38 #perform the lasso regression
39 def Lasso(X,Xte,zt,lamb,trainn):
40     """
41     Calculates the lasso regression using a polynomial.
42     Inputs: Train data, test data, training outputs, lambda
43     variable and number of training data.
44     Outputs: Beta parameters, prediced z on train data and z
45     predicted on test data.
46     """
47     lasso=linear_model.Lasso(alpha=lamb)
48     lasso.fit(X,zt)
49     zpred = lasso.predict(X)
50     zpredtest = lasso.predict(Xte)
51     beta = lasso.coef_
52     return beta, zpred, zpredtest
53
54 #function to calculate MSE and R2
55 #needs flattend z and zpred array
56 def MSER2 (z,zpred,n):
57     """
58     Calculates the MSe and R2 scores.
59     Inputs: Correct data, predicted data and number of data.
60     Outputs: MSE- and R2-score.
61     """
62     ze = 0 #z error
63     za = 0 #z sum of z - zaverage
64     zm = np.mean(z) #calcute the mean of z
65     #sum of error
66     for i in range(0,n):
67         ze = ze + (zpred[i] - z[i])**2
68         za = za + (z[i] - zm)**2

```

```

66     zMSE = ze/(n) #calcute MSE
67     zR = 1 - (ze/za) #calcute R2
68
69     return zMSE, zR,
70
71
72
73 #function to calculate Bias, variance and error terms of MSE
74 def VBE(zpred,z):
75     """
76     Calculates the bias and variance.
77     Inputs: Correct data and predicted data.
78     Outputs: bias and variance.
79     """
80     #gets the mean for the prediced values of z
81     zpm = np.mean(zpred, axis=1, keepdims=True)
82
83     #values to store sums
84     zv = np.zeros(len(z))
85     zb = 0
86     n = len(z)
87
88     for i in range(0,n):
89         zb += (z[i] - zpm[i])**2
90
91     for i in range(0,zpred.shape[1]):
92         zv += zpred[:,i]**2
93
94     #variance calculation
95     varz = np.mean((zv / zpred.shape[1]) - zpm**2)
96
97     #bias calculation
98     biasz = (zb / n)
99
100
101     # #used to compare methods
102     # bias = np.mean( (z.reshape(len(z),1) - np.mean(zpred, axis
103     # variance = np.mean( np.var(zpred, axis=1, keepdims=True) )
104     # print(bias,variance)
105     # print(biasz,varz)
106
107     return varz, biasz
108
109 def ising_energies(states,L):

```

```

110     """
111     This function calculates the energies of the states in the
nn Ising Hamiltonian
112     """
113     J=np.zeros((L,L),)
114     for i in range(L):
115         J[i,(i+1)%L]=-1.0
116
117     # compute energies
118     E = np.einsum('...i,ij,...j->...',states,J,states)
119
120     return E
121
122 # adjustable parameters
123 sampn = 1000 #number of samples
124 lamb = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000] #
    value of lambda
125 trainp = 0.7 #Number of training samples given as %
126 bootrun = 100 #times to run the bootstrap
127 nlevel = 0 #noise level
128
129
130 #not adjustable
131 L = 40
132 count = 0 #counter for figures
133 #get number of training samples
134 trainn = int(sampn*trainp)
135 #function for noise
136 N0 = np.random.normal(0,nlevel , sampn)
137
138
139 # create 10000 random Ising states
140 states=np.random.choice([-1, 1], size=(sampn,L))
141
142
143 #z calculated using energies
144 z = ising_energies(states,L) + N0
145
146
147 #get data for X matrix
148 states=np.einsum('...i,...j->...ij', states , states)
149 states=states.reshape((sampn,L*L))
150
151 #number of test data
152 testn = sampn - trainn

```

```

153
154 #array to store the r2 scores
155 train_r2_ols = np.zeros(len(lamb))
156 test_r2_ols = np.zeros(len(lamb))
157 train_r2_ridge = np.zeros(len(lamb))
158 test_r2_ridge = np.zeros(len(lamb))
159 train_r2_lasso = np.zeros(len(lamb))
160 test_r2_lasso = np.zeros(len(lamb))
161
162 #array to store the mse scores
163 train_mse_ols = np.zeros(len(lamb))
164 test_mse_ols = np.zeros(len(lamb))
165 train_mse_ridge = np.zeros(len(lamb))
166 test_mse_ridge = np.zeros(len(lamb))
167 train_mse_lasso = np.zeros(len(lamb))
168 test_mse_lasso = np.zeros(len(lamb))
169
170 #array to store bias and variance
171 bias_ols = np.zeros(len(lamb))
172 var_ols = np.zeros(len(lamb))
173 bias_ridge = np.zeros(len(lamb))
174 var_ridge = np.zeros(len(lamb))
175 bias_lasso = np.zeros(len(lamb))
176 var_lasso = np.zeros(len(lamb))
177
178 for l in lamb:
179
180     #creates array to store data
181     zeps = np.zeros((3,bootrun))
182     zMSE = np.zeros((3,bootrun))
183     zR = np.zeros((3,bootrun))
184     zMSEte = np.zeros((3,bootrun))
185     zRte = np.zeros((3,bootrun))
186     timeend = np.zeros(bootrun)
187     varz = np.zeros(3)
188     biasz = np.zeros(3)
189
190     #creates array to store the best beta coefficients
191     beta = np.zeros((3,bootrun,L*L))
192
193     #get random indices for test data
194     randi = random.sample(range(0,sampn),testn)
195
196     #array to store test data
197     xte = np.zeros((testn,L*L))

```



```

198     zte = np.zeros(testn)
199
200     #get the test data from the full set
201     for k in range(0,testn):
202         xte[k] = states[randi[k],:]
203         zte[k] = z[randi[k]]
204
205     #remove the test data from the sample som they are not
206     #used for training
207     xtr = np.delete(states ,randi ,axis=0)
208     ztr = np.delete(z ,randi)
209
210     #create array to store the test results
211     zpredtest_o_sum = np.zeros((testn ,bootrun))
212     zpredtest_r_sum = np.zeros((testn ,bootrun))
213     zpredtest_l_sum = np.zeros((testn ,bootrun))
214
215     for j in range(0,bootrun):
216
217         #set start time for the run
218         timestart = tm.time()
219
220         #get random indexes for the training data (with
221         #resampling)
222         randi =np.random.randint((trainn) , size=trainn)
223
224         #array to store the trainging data
225         xt = np.zeros((trainn ,L*L))
226         zt = np.zeros(trainn)
227
228         #takes trainging data with resampling
229         for k in range(0,trainn):
230             xt[k] = xtr[randi[k],:]
231             zt[k] = ztr[randi[k]]
232
233         #fit functions
234         #ridge
235         lam = np.identity(xt.shape[1]) * 1
236         beta[1,j,:] , zpred_r , zpredtest_r = ridge(xt ,xte ,zt ,lam ,
237 sampn)
238         print(beta[1,j,:10])
239
240         #lasso
241         beta[2,j,:] , zpred_l , zpredtest_l = Lasso(xt ,xte ,zt ,l ,

```

```

sampn)
241
242     #OLS
243     beta[0,j,:], zpred_o, zpredtest_o = OLS(xt,xte,zt,sampn)
244
245     #calculate MSE and R2
246     zMSE[0][j], zR[0][j] = MSER2(zt,zpred_o,len(zt))
247     zMSE[1][j], zR[1][j] = MSER2(zt,zpred_r,len(zt))
248     zMSE[2][j], zR[2][j] = MSER2(zt,zpred_l,len(zt))
249
250     #calculate MSE and R2 for test data
251     zMSEte[0][j], zRte[0][j] = MSER2(zte,zpredtest_o,len(zte
))
252     zMSEte[1][j], zRte[1][j] = MSER2(zte,zpredtest_r,len(zte
))
253     zMSEte[2][j], zRte[2][j] = MSER2(zte,zpredtest_l,len(zte
))
254
255     #sum the test results
256     zpredtest_o_sum[:,j] = zpredtest_o
257     zpredtest_r_sum[:,j] = zpredtest_r
258     zpredtest_l_sum[:,j] = zpredtest_l
259
260     #gets the times used to make and use model
261     timeend[j] = tm.time() - timestart
262
263
264
265     #function to calualte variance bias and error term
266     varz[0], biasz[0] = VBE(zpredtest_o_sum,zte)
267     varz[1], biasz[1] = VBE(zpredtest_r_sum,zte)
268     varz[2], biasz[2] = VBE(zpredtest_l_sum,zte)
269
270
271     #prints releevant data
272     print("OLS method")
273     print("Training data:")
274     print("MSE = %.3f    R2 = %.3f" %(np.mean(zMSE[0,:]), np.
mean(zR[0,:]))))
275     print("Test data:")
276     print("MSE = %.3f    R2 = %.3f" %(np.mean(zMSEte[0,:]), np.
mean(zRte[0,:]))))
277     print("Bias      : %.4f" %(np.mean(biasz[0])))
278     print("Variance  : %.4f\n" %(np.mean(varz[0])))
279

```

```

280
281     print("Ridge method with lambda=%0.4f" %l)
282     print("Training data:")
283     print("MSE = %0.3f    R2 = %0.3f" %(np.mean(zMSE[1, :]), np.
mean(zR[1, :]))))
284     print("Test data:")
285     print("MSE = %0.3f    R2 = %0.3f" %(np.mean(zMSEte[1, :]), np.
mean(zRte[1, :]))))
286     print("Bias          : %0.4f" %(np.mean(biasz[1])))
287     print("Variance     : %0.4f\n" %(np.mean(varz[1])))
288
289     print("Lasso method with lambda=%0.4f" %l)
290     print("Training data:")
291     print("MSE = %0.3f    R2 = %0.3f" %(np.mean(zMSE[2, :]), np.
mean(zR[2, :]))))
292     print("Test data:")
293     print("MSE = %0.3f    R2 = %0.3f" %(np.mean(zMSEte[2, :]), np.
mean(zRte[2, :]))))
294     print("Bias          : %0.4f" %(np.mean(biasz[2])))
295     print("Variance     : %0.4f\n" %(np.mean(varz[2])))
296
297     print("_____")
298
299     #store data for plotting
300     train_r2_ols[count] = np.mean(zR[0, :])
301     test_r2_ols[count] = np.mean(zRte[0, :])
302     train_r2_ridge[count] = np.mean(zR[1, :])
303     test_r2_ridge[count] = np.mean(zRte[1, :])
304     train_r2_lasso[count] = np.mean(zR[2, :])
305     test_r2_lasso[count] = np.mean(zRte[2, :])
306
307     train_mse_ols[count] = np.mean(zMSE[0, :])
308     test_mse_ols[count] = np.mean(zMSEte[0, :])
309     train_mse_ridge[count] = np.mean(zMSE[1, :])
310     test_mse_ridge[count] = np.mean(zMSEte[1, :])
311     train_mse_lasso[count] = np.mean(zMSE[2, :])
312     test_mse_lasso[count] = np.mean(zMSEte[2, :])
313
314     bias_ols[count] = np.mean(biasz[0])
315     var_ols[count] = np.mean(varz[0])
316     bias_ridge[count] = np.mean(biasz[1])
317     var_ridge[count] = np.mean(varz[1])
318     bias_lasso[count] = np.mean(biasz[2])
319     var_lasso[count] = np.mean(varz[2])
320

```

```

321
322     #method for plotting gotten from stack exchange. Vissited:
02-11-18
323     #url: https://stackoverflow.com/questions/13784201/
matplotlib-2-subplots-1-colorbar
324     #author: spinup. url: https://stackoverflow.com/users
/1329892/spinup
325     fig = plt.figure(figsize=(9.75, 3))
326
327     grid = ImageGrid(fig, 111,
328                       nrows_ncols=(1,3),
329                       axes_pad=0.15,
330                       share_all=True,
331                       cbar_location="right",
332                       cbar_mode="single",
333                       cbar_size="7%",
334                       cbar_pad=0.15,
335                       )
336
337     # Add data to image grid
338     grid[0].imshow(beta[0][j,:].reshape(L,L), cmap='seismic',
vmin=-1, vmax=1,)
339     grid[0].set_title('OLS', fontsize=16)
340
341     grid[1].imshow(beta[1][j,:].reshape(L,L), cmap='seismic',
vmin=-1, vmax=1,)
342     grid[1].set_title('Ridge  $\lambda = %.4f$ ' % l, fontsize=16)
343
344     im = grid[2].imshow(beta[2][j,:].reshape(L,L), cmap='seismic',
vmin=-1, vmax=1,)
345     grid[2].set_title('Lasso  $\lambda = %.4f$ ' % l, fontsize=16)
346     # Colorbar
347     grid[2].cax.colorbar(im)
348     grid[2].cax.toggle_label(True)
349
350     count += 1
351
352     plt.figure(count + 1)
353     # Plot our performance on both the training and test data
354     plt.semilogx(lamb, train_r2_ols, 'b', label='OLS train')
355     plt.semilogx(lamb, test_r2_ols, '—b', label='OLS test')
356     plt.semilogx(lamb, train_r2_ridge, 'r', label='Ridge train',
linewidth=1)
357     plt.semilogx(lamb, test_r2_ridge, '—r', label='Ridge test',
linewidth=1)

```

```

358 plt.semilogx(lamb, train_r2_lasso, 'g',label='Lasso train')
359 plt.semilogx(lamb, test_r2_lasso, '—g',label='Lasso test')
360
361
362 #fig.set_size_inches(10.0, 6.0)
363 plt.title("R2-scores for test and training data with different
           values for lambda", fontsize = 16)
364 plt.legend(loc='lower left',fontsize=16)
365 plt.ylim([-0.01, 1.01])
366 plt.xlim([min(lamb), max(lamb)])
367 plt.xlabel('Lambda',fontsize=15)
368 plt.ylabel('R2 - score',fontsize=15)
369 plt.tick_params(labelsize=15)
370
371 plt.figure(count + 2)
372 # Plot our performance on both the training and test data
373 plt.semilogx(lamb, train_mse_ols, 'b',label='OLS train')
374 plt.semilogx(lamb, test_mse_ols, '—b',label='OLS test')
375 plt.semilogx(lamb, train_mse_ridge, 'r',label='Ridge train',
               linewidth=1)
376 plt.semilogx(lamb, test_mse_ridge, '—r',label='Ridge test',
               linewidth=1)
377 plt.semilogx(lamb, train_mse_lasso, 'g',label='Lasso train')
378 plt.semilogx(lamb, test_mse_lasso, '—g',label='Lasso test')
379
380
381 #fig.set_size_inches(10.0, 6.0)
382 plt.title("MSE-scores for test and training data with different
           values for lambda", fontsize = 16)
383 plt.legend()
384 #plt.ylim([-0.01, 1.01])
385 plt.xlim([min(lamb), max(lamb)])
386 plt.xlabel('Lambda',fontsize=15)
387 plt.ylabel('MSE - score',fontsize=15)
388 plt.tick_params(labelsize=15)
389
390 plt.figure(count + 3)
391 # Plot our performance on both the training and test data
392 plt.semilogx(lamb, bias_ols, 'b',label='OLS bias')
393 plt.semilogx(lamb, var_ols, '—b',label='OLS var')
394 plt.semilogx(lamb, bias_ridge, 'r',label='Ridge bias',linewidth
               =1)
395 plt.semilogx(lamb, var_ridge, '—r',label='Ridge var',linewidth
               =1)
396 plt.semilogx(lamb, bias_lasso, 'g',label='Lasso bias')

```

```

397 plt.semilogx(lamb, var_lasso, '—g',label='Lasso var')
398
399
400 #fig.set_size_inches(10.0, 6.0)
401 plt.title("Bias and variance for test data with different values
         for lambda", fontsize = 16)
402 plt.legend()
403 #plt.ylim([-0.01, 1.01])
404 plt.xlim([min(lamb), max(lamb)])
405 plt.xlabel('Lambda',fontsize=15)
406 plt.ylabel('bias/var - score',fontsize=15)
407 plt.tick_params(labelsize=15)
408
409
410
411 plt.show()

```

6.3.2 Logistic Regression (part c)

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3 from sklearn import metrics
4 from sklearn import linear_model
5 import random
6 import time as tm
7 import pickle
8
9
10 def predict(x,beta):
11     """
12     Function that predicts labels based on the beta parameters
13     that has been
14     calculatted.
15     Inputs: datapoints and beta parameters
16     Output: predicted label
17     """
18     ypred = x @ beta
19     ypred = 1 / (1 + np.exp(-ypred))
20     ypred[ypred < 0.5] = 0
21     ypred[ypred >= 0.5] = 1
22     return ypred
23
24 def beta_update(lr ,x,y,beta):
25     """
26     Function to calculate the new beta parameters without

```

```

27 regularization.
28 Inputs: data, labels, beta parameters
29 Output: updated beta
30 """
31 output = 1 / (1 + np.exp(-(x @ beta)))
32 delta = x.T @ (output - y.reshape((len(y),1)))
33 new_beta = beta - lr * delta/len(y)
34 return new_beta
35
36 def beta_update_l2(lr, x, y, beta, lamb):
37     """
38     Function to calculate the new beta parameters with l1
39     regularization.
40     Inputs: data, labels, beta parameters, lambda
41     Output: updated beta
42     """
43     output = 1 / (1 + np.exp(-(x @ beta)))
44     delta = x.T @ (output - y.reshape((len(y),1)))
45     new_beta = beta - lr * (delta/len(y) + lamb * beta)
46     return new_beta
47
48 #adjustable parameters
49 trainp = 0.5 #percentage of the data to be used for training
50 minibatch = 20 #minibatch size
51 lr = 0.00001 #learning rate
52 lamb = [0.00001, 0.0001, 0.001, 0.01, 1, 10, 100, 10000, 100000]
53 #value of lambda
54 #lamb = [0.00001, 0.0001, 0.001, 0.01, 1] #a smaller lambda
55 #not adjustable
56 L = 40 #the root of the number of spins
57
58 #set the filenames
59 label_filename = 'Ising2DFM_reSample_L40-T=All_labels.pkl'
60 dat_filename = 'Ising2DFM_reSample_L40-T=All.pkl'
61
62 # Read in the labels
63 with open(label_filename, "rb") as f:
64     labels = pickle.load(f)
65
66 # Read in the corresponding configurations
67 with open(dat_filename, "rb") as f:
68     data = np.unpackbits(pickle.load(f)).reshape(-1, 1600).
69     astype("int")

```

```

68 # Set spin-down to -1
69 data[data == 0] = -1
70
71
72 # Set up slices of the dataset
73 ordered = slice(0, 70000)
74 critical = slice(70000, 100000)
75 disordered = slice(100000, 160000)
76
77 #creates a dataset without the critical data
78 datawo = np.concatenate((data[ordered], data[disordered]))
79 labelswo = np.concatenate((labels[ordered], labels[disordered]))
80
81 #creates array to store the critical data
82 critical_data = np.zeros((30000,L*L+1))
83 critical_data[:,0] = -1
84 critical_data[:,1:] = data[critical]
85 critical_label = labels[critical]
86
87 #randmoly shuffle the data
88 order = list(range(np.shape(datawo)[0]))
89 np.random.shuffle(order)
90 datawo = datawo[order,:]
91 labelswo = labelswo[order]
92
93 #find total samples and calculate the number of training data
94 sampn = len(labelswo)
95 trainn = int(sampn*trainp)
96
97
98 #split the data into training and test sets
99 xt = np.zeros((trainn,L*L+1))
100 xt[:,0] = -1
101 xt[:,1:] = datawo[:trainn,:]
102 yt = labelswo[:trainn]
103
104 xte = np.zeros((sampn-trainn,L*L+1))
105 xte[:,0] = -1
106 xte[:,1:] = datawo[trainn:,:]
107 yte = labelswo[trainn:]
108
109
110 #array to store the ac scores
111 train_ac = np.zeros(len(lamb))
112 test_ac = np.zeros(len(lamb))

```



```

113 citical_ac = np.zeros(len(lamb))
114 train_ac_l2 = np.zeros(len(lamb))
115 test_ac_l2 = np.zeros(len(lamb))
116 citical_ac_l2 = np.zeros(len(lamb))
117 train_ac_sci = np.zeros(len(lamb))
118 test_ac_sci = np.zeros(len(lamb))
119 citical_ac_sci = np.zeros(len(lamb))
120
121 count = 0
122 for l in lamb:
123
124     #initialize the beta parameters
125     beta = (2/np.sqrt(L*L+1)) * np.random.random_sample((L*L
126     +1,1)) -1/np.sqrt(L*L+1)
127     beta_l2 = (2/np.sqrt(L*L+1)) * np.random.random_sample((L*L
128     +1,1)) -1/np.sqrt(L*L+1)
129
130     #variable to store the best score
131     best_score = 0
132     best_score_l2 = 0
133     for k in range(0,50):
134
135         for i in range(0,trainn,minibatch):
136
137             #update beta parameters
138             beta = beta_update(lr,xt[i:i+minibatch,:],yt[i:i+
139             minibatch],beta)
140             beta_l2 = beta_update_l2(lr,xt[i:i+minibatch,:],yt[i
141             :i+minibatch],beta_l2,l)
142
143             print(beta_l2[k,:10])
144             #checks the score of the model and stores the best
145             parameters
146             temp_pred = predict(xt,beta)
147             temp_score = np.sum(yt.reshape((trainn,1)) == temp_pred)
148             / len(yt)
149             if temp_score > best_score:
150                 best_beta = beta
151                 best_score = temp_score
152
153             temp_pred = predict(xt,beta_l2)
154             temp_score = np.sum(yt.reshape((trainn,1)) == temp_pred)
155             / len(yt)
156             if temp_score > best_score_l2:
157                 best_beta_l2 = beta_l2

```

```

151         best_score_l2 = temp_score
152
153
154         #reshuffle the data so the model does not train the same
way
155         order = list(range(np.shape(xt)[0]))
156         np.random.shuffle(order)
157         xt = xt[order,:]
158         yt = yt[order]
159
160         #predicts the labels using beta
161         ypred = predict(xte,best_beta)
162         ypred_train = predict(xt,best_beta)
163         critical_ypred = predict(critical_data,best_beta)
164
165         #predicts the labels using beta_l2
166         ypred_l2 = predict(xte,best_beta_l2)
167         ypred_train_l2 = predict(xt,best_beta_l2)
168         critical_ypred_l2 = predict(critical_data,best_beta_l2)
169
170         #fitting a scikit model
171         scilearn = linear_model.LogisticRegression(penalty='l2',C=1/
1).fit(xt, yt)
172
173
174         #calualte the score of the predicted labels
175         test_ac[count]= (np.sum(yte.reshape((sampn - trainn,1)) ==
ypred) / len(yte))*100
176         train_ac[count]= (np.sum(yt.reshape((trainn,1)) ==
ypred_train) / len(yt))*100
177         citical_ac[count] = (np.sum(critical_label.reshape((len(
critical_label),1)) == critical_ypred) / len(critical_label))
*100
178
179         test_ac_l2[count]= (np.sum(yte.reshape((sampn - trainn,1))
== ypred_l2) / len(yte))*100
180         train_ac_l2[count] = (np.sum(yt.reshape((trainn,1)) ==
ypred_train_l2) / len(yt))*100
181         citical_ac_l2[count] = (np.sum(critical_label.reshape((len(
critical_label),1)) == critical_ypred_l2) / len(
critical_label))*100
182
183         #calualte the score of the scikit models
184         test_ac_sci[count] = scilearn.score(xte,yte) * 100
185         train_ac_sci[count] = scilearn.score(xt,yt) * 100

```

```

186     citical_ac_sci[count] = scilearn.score(critical_data ,
187                                           critical_label) * 100
188
189
190
191     #print results
192     print("Created minibatch method:")
193     print("Train score: %.4f" %train_ac[count])
194     print("Test score: %.4f" %test_ac[count])
195     print("Critical score: %.4f\n" %citical_ac[count])
196
197     print("Created minibatch with L2 regularization lambda = %.5
198 f:" %l)
199     print("Train score: %.4f" %train_ac_l2[count])
200     print("Test score: %.4f" %test_ac_l2[count])
201     print("Critical score: %.4f\n" %citical_ac_l2[count])
202
203     print("Scikit learn method lambda = %.5f:" %l)
204     print("Train score: %.4f" %train_ac_sci[count])
205     print("Test score: %.4f" %test_ac_sci[count])
206     print("Critical score: %.4f\n" %citical_ac_sci[count])
207
208     print("—————\n")
209
210     count += 1
211     plt.figure(count +1)
212     # Plot our performance on both the training and test data
213     plt.semilogx(lamb, train_ac , 'b',label='Created method train')
214     plt.semilogx(lamb, test_ac , '—b',label='Created method test')
215     plt.semilogx(lamb, citical_ac , '-.b',label='Created method
216 critical')
217     plt.semilogx(lamb, train_ac_l2 , 'r',label='Created L2 train',
218 linewidth=1)
219     plt.semilogx(lamb, test_ac_l2 , '—r',label='Created L2 test',
220 linewidth=1)
221     plt.semilogx(lamb, citical_ac_l2 , '-.r',label='Created L2
222 critical',linewidth=1)
223     plt.semilogx(lamb, train_ac_sci , 'g',label='Scikit train')
224     plt.semilogx(lamb, test_ac_sci , '—g',label='Scikit test')
225     plt.semilogx(lamb, citical_ac_sci , '-.g',label='Scikit critical'
226 )
227
228     plt.title("Accuracy scores for test, training and critical data

```

```

224     with different values for lambda", fontsize = 16)
225 plt.legend(loc='lower left', fontsize=16)
226 plt.xlim([min(lamb), max(lamb)])
227 plt.xlabel('Lambda', fontsize=15)
228 plt.ylabel('Accuracy score [%]', fontsize=15)
229 plt.tick_params(labelsize=15)
230
231 plt.show()

```

6.3.3 MLP for regression (part d)

```

1 import time
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import pickle
5 import sklearn.metrics as met
6
7
8 from tensorflow.keras.models import Sequential
9 from tensorflow.keras.layers import Dense
10 from tensorflow.keras import optimizers
11 from tensorflow.keras import callbacks
12
13 class mlp:
14     """
15     Class to store the neural network
16     """
17     def __init__(self):
18         """
19         function to initialize the network
20         """
21         #set if the network prints the progress
22         self.verbose = False
23         #toggle the keras nettwork
24         self.run_keras = True
25         #learning rate
26         self.eta = 0.01
27         #momentum factor
28         self.momentum = 0.5
29         #number of hidden nodes
30         self.hidden = 20
31         #number of inputs
32         self.ninput = 1600
33         #number of outputs
34         self.noutput = 1

```

```

35         #size of minibatch
36         self.minibatch = 50
37         #number of epochs between checking earlystopping
38         self.early = 10
39         #weights for hidden layer
40         self.v = (2/np.sqrt(self.ninput + 1)) * np.random.
random_sample((self.ninput + 1 ,self.hidden)) -1/np.sqrt(self
.ninput + 1)
41         #store previous deltas for momentum
42         self.vprev = np.zeros((self.ninput + 1 ,self.hidden))
43         #weights for output layer
44         self.w = (2/np.sqrt(self.hidden + 1)) * np.random.
random_sample((self.hidden + 1 ,self.noutput)) -1/np.sqrt(
self.hidden + 1)
45         #store previous deltas for momentum
46         self.wprev = np.zeros((self.hidden + 1 ,self.noutput))
47
48
49     def earlystopping(self , inputs , targets , valid , validtargets
):
50         """
51         The earlystopping function runs the training function a
number of epoch and evaluates the nn. The training
52         is stopped if the network show no sign of improvement. A
validation set is used to assess the model.
53         Inputs: training data, training labels , validation data,
validation labels.
54         Outputs: Number of runs before earlystopping and array
of accuracy scores.
55         """
56
57         #creates arrays to store MSE
58         last_MSE = np.zeros(10)
59         timestart = 0
60         overfit = 0
61
62         for k in range(0,10000):
63
64             #trains the program for using all the test data
65             for i in range(0,inputs.shape[0] ,self.minibatch):
66                 self.train(inputs[i:i+self.minibatch ,:] , targets [
i:i+self.minibatch ,:])
67
68
69

```

```

70         #reorder the data so the program is not trained in
the same way every epoch
71         order = list(range(np.shape(inputs)[0]))
72         np.random.shuffle(order)
73         inputs = inputs[order,:]
74         targets = targets[order,:]
75
76
77         #checks the model with the validation set every 100
epochs
78         if k%self.early == 0:
79             timend =time.time() - timestart
80
81             #gets the MSE and R2 score for the validation
set
82             MSE, R2 = mlp1.score(valid,validtargets)
83             #stores the last 10 ac scores
84             last_MSE[1:] = last_MSE[:9]
85             last_MSE[0] = MSE
86             diff = MSE - np.mean(last_MSE)
87
88
89             timestart = time.time()
90
91             #prints training and validation results if
verbose is true
92             if self.verbose:
93                 MSE_train, R2_train = mlp1.score(inputs,
targets)
94                 print("_____")
95                 print("After %d epochs"%k)
96                 print("Validation set:")
97                 print("MSE: %.5f    R2: %.5f" %(MSE, R2))
98                 print("Training set:")
99                 print("MSE: %.5f    R2: %.5f" %(MSE_train,
R2_train))
100
101
102             #cheks if no improvements are found and
increments overfit variable
103             if diff > 1e-10: overfit += 1
104             #overfit varibale is reset if an improvement is
found
105             else:
106                 overfit = 0

```

```

107         best_v = self.v
108         best_w = self.w
109
110         #stops if there is little change in MSE
111         if (overfit > 5 or MSE < 1e-3) and k>10 * self.
early:
112             self.v = best_v
113             self.w = best_w
114             break
115
116
117     return k
118
119 def train(self, inputs, targets):
120     """
121     Trains the network by running it forwards and then using
122     backpropagation to adjust the weights.
123     Inputs: training data, training labels.
124     """
125
126     #runs the program forward
127     outputH, outputO = self.forward(inputs)
128
129     #calculate output error
130     errOm = (outputO.T - targets)
131
132     #calculate hidden layer error
133     summerrm = self.w[1:,:] @ errOm.T
134
135     errHm = (outputH * np.subtract(1, outputH)) * summerrm
136
137     #put the bias in the hidden layer output
138     outputHmi = np.zeros((self.hidden + 1, self.minibatch))
139     outputHmi[0,:] = -1
140     outputHmi[1:,:] = outputH
141
142     #adjusting weight for the output
143     deltaw = self.eta * outputHmi @ (errOm/self.minibatch)
144     self.w = self.w - deltaw + self.momentum * self.wprev
145     self.wprev = deltaw
146
147     #put the bias in the inputs
148     inputsi = np.zeros((inputs.shape[1] + 1, self.minibatch))
149     inputsi[0,:] = -1
150     inputsi[1:,:] = inputs.T

```

```

150
151     #adjusting wheight for hidden layer
152     deltav = self.eta * inputsi @ (errHm/self.minibatch).T
153     self.v = self.v - deltav + self.momentum * self.vprev
154     self.vprev = deltav
155
156     #function to run the MPL forward
157     def forward(self , inputs):
158         """
159         Runs the network forwards.
160         Inputs: training data.
161         Outputs: Hiddel layer outputs and network output.
162         """
163         #puts bias in the inputs
164         inputss = np.zeros((inputs.shape[0] , inputs.shape[1] + 1)
165 )
166         inputss[:,0] = -1
167         inputss[:,1:] = inputs
168
169         #caculate the hidden layer output
170         outputhm = self.v.T @ inputss.T
171
172         outputhm = self.activate(outputhm)
173
174         #puts bias in the hiddel layer outputs
175         outputhmi = np.zeros((self.hidden + 1,inputs.shape[0]))
176         outputhmi[0,:] = -1
177         outputhmi[1:,:] = outputhm
178
179         #calculate the outputs of the MLP
180         outputom = self.w.T @ outputhmi
181
182         #returns the hidden layer output and the output output
183         return outputhm, outputom
184
185     def score(self , inputs , targets):
186         """
187         Calculates the MSE and R2 scores of the network.
188         Inputs: Data and labels that are going to be assessed.
189         Outputs: MSE- and R2-scores.
190         """
191         #runs the moodel forward
192         outputHi, outputOu = self.forward(inputs)
193
194         target_e = 0 #target error

```



```

194         target_a = 0 #target sum of target - target_avrage
195         target_avg = np.mean(targets) #calcute the mean of
targets
196         n = inputs.shape[0]
197
198         #sum of error
199         target_e = np.sum((outputOu.T - targets)**2)
200         target_a = np.sum((targets - target_avg)**2)
201
202         MSE = target_e/(n) #calcute MSE
203         R2 = 1 - (target_e/target_a) #calcute R2
204
205         return MSE ,R2
206
207     def activate(self , inputs):
208         """
209         Calculates the activation function for an output.
210         Inputs: outputs of a layer in the network
211         Outputs: the activation function result.
212         """
213
214         return 1 / (1 + ( np.exp( - inputs)))
215
216     def ising_energies(states ,L):
217         """
218         This function calculates the energies of the states in the
nn Ising Hamiltonian
219         """
220         J=np.zeros((L,L),)
221         for i in range(L):
222             J[i,(i+1)%L]=-1.0
223
224         # compute energies
225         E = np.einsum('...i,ij,...j->...',states,J,states)
226
227         return E
228
229
230     sampn = 1000 #number of samples
231     nlevel = 0 #noise level
232
233     N0 = np.random.normal(0,nlevel , sampn)
234     L = 40
235
236     # create 10000 random Ising states

```

```

237 states=np.random.choice([-1, 1], size=(sampn,L))
238
239 #gets the target energies using the ising_energies function
240 target = ising_energies(states,L) + N0
241 target = target.reshape(sampn,1)
242
243 #get data for X matrix
244 states=np.einsum('...i,...j->...ij', states, states)
245 states=states.reshape((sampn,L*L))
246
247
248 # Split data into k sets
249 foldsm = []
250 foldst = []
251 for i in range(0,sampn,int(sampn/10)):
252     foldsm.append(states[i:i+int(sampn/10),:])
253     foldst.append(target[i:i+int(sampn/10),:])
254
255
256
257 #array to store data
258 MSE_test = np.zeros(10)
259 R2_test = np.zeros(10)
260 MSE_train = np.zeros(10)
261 R2_train = np.zeros(10)
262 k = np.zeros(10)
263 mse_keras_test = np.zeros(10)
264 r2_keras_test = np.zeros(10)
265 mse_keras_train = np.zeros(10)
266 r2_keras_train = np.zeros(10)
267
268
269 for i in range(0,10):
270
271     # Test data is used to evaluate how good the completely
272     # trained network is.
273     test = foldsm[i]
274     test_targets = foldst[i]
275     sum = 0
276
277     valind = np.random.randint(9)
278     if valind >= i:
279         valind = valind + 1
280
281     # Validation checks how well the network is performing and

```

```

281     when to stop
282         valid = foldsm[valind]
283         valid_targets = foldst[valind]
284
285         sumind = 0
286         for j in range(0,10):
287             if j != i and j != valind: sumind = sumind + foldsm[j].
288             shape[0]
289
290         #Training data to train the network
291         train = np.zeros((sumind,1600))
292         train_targets = np.zeros((sumind,1))
293         placedind = 0
294         for j in range(0,10):
295             if j != i and j != valind:
296                 train[placedind:placedind+foldsm[j].shape[0],:] =
297                 foldsm[j]
298                 train_targets[placedind:placedind+foldsm[j].shape
299                 [0],:] = foldst[j]
300                 placedind = placedind + foldsm[j].shape[0]
301
302         # initialize the network
303         mlp1 = mlp()
304
305         #run the program
306         k[i]=mlp1.earlystopping(train , train_targets , valid ,
307         valid_targets)
308
309         #get the MSE and R2 for our model
310         MSE_test[i], R2_test[i] = mlp1.score(test , test_targets)
311         MSE_train[i], R2_train[i] = mlp1.score(train , train_targets)
312
313
314
315         #using keras for comparison
316         if mlp1.run_keras:
317             model = Sequential()
318             model.add(Dense(mlp1.hidden , input_dim=mlp1.ninput ,
319             activation='relu'))
320             model.add(Dense(1, activation='linear'))

```

```

320         sgd = optimizers.SGD(lr=mlp1.eta, momentum=mlp1.eta,
321                                decay=0.0, nesterov=False)
322         model.compile(loss='mse', optimizer=sgd)
323         earlystop = callbacks.EarlyStopping(monitor='val_loss',
324                                                min_delta=0, patience=1, verbose=0, mode='auto')
325         model.fit(train, train_targets, epochs=1000, verbose=0,
326                   batch_size=mlp1.minibatch, validation_data=(valid,
327                                                                valid_targets),
328                   callbacks=[earlystop])
329         keras_pred_test = model.predict(test)
330         keras_pred_train = model.predict(train)
331
332         mse_keras_test[i] = met.mean_squared_error(test_targets,
333                                                    keras_pred_test)
334         r2_keras_test[i] = met.r2_score(test_targets,
335                                         keras_pred_test)
336         mse_keras_train[i] = met.mean_squared_error(
337             train_targets, keras_pred_train)
338         r2_keras_train[i] = met.r2_score(train_targets,
339                                         keras_pred_train)
340
341         #print results for each fold
342         print("Created MLP:")
343         print("Test set MSE: %.4f R2: %.4f" %(MSE_test[i], R2_test[i]
344         ))
345         print("Train set MSE: %.4f R2: %.4f\n" %(MSE_train[i],
346         R2_train[i]))
347
348         if mlp1.run_keras:
349             print("keras MLP:")
350             print("Test set MSE: %.4f R2: %.4f" %(mse_keras_test[i],
351             r2_keras_test[i]))
352             print("Train set MSE: %.4f R2: %.4f\n" %(mse_keras_train
353             [i], r2_keras_train[i]))
354             print("_____\\n")
355
356         #print results
357         print("Created MLP")
358         print("Test data:")
359         print("Average MSE %.2f. Min MSE: %.2f. Max MSE: %.2f. MSE std:
360         %.2f"
361         %(np.mean(MSE_test), np.min(MSE_test), np.max(MSE_test), np.std(
362         MSE_test)))
363         print("Average R2 %.2f. Min R2: %.2f. Max R2: %.2f. R2 std: %.2f
364         ")

```

```

350 %(np.mean(R2_test), np.min(R2_test), np.max(R2_test), np.std(
    R2_test)))
351 print("Train data:")
352 print("Average MSE %.2f. Min MSE: %.2f. Max MSE: %.2f. MSE std:
    %.2f")
353 %(np.mean(MSE_train), np.min(MSE_train), np.max(MSE_train), np.
    std(MSE_train)))
354 print("Average R2 %.2f. Min R2: %.2f. Max R2: %.2f. R2 std: %.2f
    \n")
355 %(np.mean(R2_train), np.min(R2_train), np.max(R2_train), np.std(
    R2_train)))
356
357 if mlp1.run_keras:
358     print("keras MLP")
359     print("Test data:")
360     print("Average MSE %.2f. Min MSE: %.2f. Max MSE: %.2f. MSE
        std: %.2f")
361     %(np.mean(mse_keras_test), np.min(mse_keras_test), np.max(
        mse_keras_test), np.std(mse_keras_test)))
362     print("Average R2 %.2f. Min R2: %.2f. Max R2: %.2f. R2 std:
        %.2f")
363     %(np.mean(r2_keras_test), np.min(r2_keras_test), np.max(
        r2_keras_test), np.std(r2_keras_test)))
364     print("Train data:")
365     print("Average MSE %.2f. Min MSE: %.2f. Max MSE: %.2f. MSE
        std: %.2f")
366     %(np.mean(mse_keras_train), np.min(mse_keras_train), np.max(
        mse_keras_train), np.std(mse_keras_train)))
367     print("Average R2 %.2f. Min R2: %.2f. Max R2: %.2f. R2 std:
        %.2f")
368     %(np.mean(r2_keras_train), np.min(r2_keras_train), np.max(
        r2_keras_train), np.std(r2_keras_train)))
369
370 plt.figure(1)
371 # Plot MSE on both the training and test data
372 plt.plot(MSE_train, 'b', label='Created MLP train')
373 plt.plot(MSE_test, '—b', label='Created MLP test')
374 if mlp1.run_keras:
375     plt.plot(mse_keras_train, 'r', label='keras MLP train')
376     plt.plot(mse_keras_test, '—r', label='keras MLP test')
377
378
379
380 if mlp1.run_keras:
381     plt.title("MSE-scores for test and training data for the

```

```

        created and the keras MLP during a 10-fold", fontsize = 16)
382 else:
383     plt.title("MSE-scores for test and training data for the
        created MLP during a 10-fold", fontsize = 16)
384 plt.legend(fontsize=16)
385 plt.xlabel('Fold number',fontsize=15)
386 plt.ylabel('MSE - score',fontsize=15)
387 plt.tick_params(labelsize=15)
388
389
390 # Plot R2 on both the training and test data
391 plt.figure(2)
392 plt.plot(R2_train, 'b',label='Created MLP train')
393 plt.plot(R2_test, '--b',label='Created MLP test')
394 if mlp1.run_keras:
395     plt.plot(r2_keras_train, 'r',label='keras MLP train')
396     plt.plot(r2_keras_test, '--r',label='keras MLP test')
397
398
399 if mlp1.run_keras:
400     plt.title("R2-scores for test and training data for the
        created and the keras MLP during a 10-fold", fontsize = 16)
401 else:
402     plt.title("R2-scores for test and training data for the
        created MLP during a 10-fold", fontsize = 16)
403 plt.legend(fontsize=16)
404 plt.xlabel('Fold number',fontsize=15)
405 plt.ylabel('R2 - score',fontsize=15)
406 plt.tick_params(labelsize=15)
407
408
409 plt.show()

```

6.3.4 MLP for classification (part e)

```

1 import time
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import pickle
5 from sklearn import metrics
6
7 from tensorflow.keras.models import Sequential
8 from tensorflow.keras.layers import Dense
9 from tensorflow.keras import callbacks
10
11 class mlp:

```

```

12 """
13 Class to store the neural network
14 """
15 def __init__(self):
16     """
17     function to initialize the network
18     """
19     #set if the network prints the progress
20     self.verbose = False
21     #toggle the keras nettwork
22     self.run_keras = True
23     #learning rate
24     self.eta = 0.1
25     #momentum factor
26     self.momentum = 0.7
27     #number of hidden nodes
28     self.hidden = 12
29     #number of inputs
30     self.ninput = 1600
31     #number of outputs
32     self.noutput = 2
33     #size of minibatch
34     self.minibatch = 20
35     #number of epochs between checking earlystopping
36     self.early = 10
37     #weights for hidden layer
38     self.v = (2/np.sqrt(self.ninput + 1)) * np.random.
random_sample((self.ninput + 1 ,self.hidden)) -1/np.sqrt(self
.ninput + 1)
39     #store previous deltas for momentum
40     self.vprev = np.zeros((self.ninput + 1 ,self.hidden))
41     #weights for output layer
42     self.w = (2/np.sqrt(self.hidden + 1))* np.random.
random_sample((self.hidden + 1 ,self.noutput)) -1/np.sqrt(
self.hidden + 1)
43     #store previous deltas for momentum
44     self.wprev = np.zeros((self.hidden + 1 ,self.noutput))
45
46
47 def earlystopping(self , inputs , targets , valid , validtargets
):
48     """
49     The earlystopping function runs the training function a
number of epoch and evaluates the nn. The training
50     is stopped if the network show no sign of improvement. A

```

```

51     validation set is used to assess the model.
52     Inputs: training data, training labels, validation data,
53     validation labels.
54     """
55     #creates arrays to store MSE
56     last_ac = np.zeros(10)
57     timestart = 0
58     overfit = 0
59     for k in range(0,10000):
60         #trains the MLP using all the test data
61         for i in range(0,inputs.shape[0],self.minibatch):
62             self.train(inputs[i:i+self.minibatch,:],targets[
63             i:i+self.minibatch,:])
64
65         #reorder the data so the program is not trained in
66         the same way every epoch
67         order = list(range(np.shape(inputs)[0]))
68         np.random.shuffle(order)
69         inputs = inputs[order,:]
70         targets = targets[order,:]
71
72         #checks the model with the validation set every 100
73         epochs
74         if k%self.early == 0:
75             timend =time.time() - timestart
76
77             #get accuracy of the model using the validation
78             set
79             ac, conf = mlp1.confusion(valid,validdtargets)
80
81             #stores the last 10 ac scores
82             last_ac[1:] = last_ac[:9]
83             last_ac[0] = ac
84             diff = ac - np.mean(last_ac)
85
86             if self.verbose:
87                 #get accuracy of the model using the traing
88                 set
89                 ac_train, conf_train = mlp1.confusion(
90                 inputs,targets)

```



```

88         print("_____")
89         print("After %d epochs" %k)
90         print("Validation set accuracy: %.5f" %ac)
91         print("Training set accuracy: %.5f" %
ac_train)

92
93
94         #cheks if no improvements are found and
increments overfit variable
95         if diff <= 1e-10: overfit += 1
96         #overfit varibale is reset if an improvement is
found
97         else:
98             overfit = 0
99             best_v = self.v
100            best_w = self.w
101
102            #stops if there is little change in accuracy or
103            if overfit > 5 or ac == 100:
104                self.v = best_v
105                self.w = best_w
106                break
107            return k
108
109    def train(self, inputs, targets):
110        """
111        Trains the network by running it forwards and then using
backpropagation to adjust the weights.
112        Inputs: training data, training labels.
113        Outputs: Number of runs before earlystopping and array
of accuracy scores.
114        """
115        #create array to store errors
116        errOm = np.zeros((self.noutput,1))
117        errOmsum = np.zeros((self.noutput,1))
118        errHm = np.zeros((self.hidden,1))
119
120
121        #runs the program forward
122        outputH, outputO = self.forward(inputs)
123
124        #calculate output error
125        errOm = (outputO.T - targets) * outputO.T * np.subtract
(1, outputO.T)
126

```

```

127
128     #calculate hidden layer error
129     summerrm = self.w[1:,:] @ errOm.T
130     errHm = (outputH * np.subtract(1, outputH)) * summerrm
131
132     #put the bias in the hidden layer output
133     outputHmi = np.zeros((self.hidden + 1,self.minibatch))
134     outputHmi[0,:] = -1
135     outputHmi[1:,:] = outputH
136
137     #adjusting weight for the output
138     deltaw = self.eta * outputHmi @ (errOm/self.minibatch)
139     self.w = self.w - deltaw + self.momentum * self.wprev
140     self.wprev = deltaw
141
142     #put the bias in the inputs
143     inputsi = np.zeros((inputs.shape[1] + 1,self.minibatch))
144     inputsi[0,:] = -1
145     inputsi[1:,:] = inputs.T
146
147     #adjusting wheight for hidden layer
148     deltav = self.eta * inputsi @ (errHm/self.minibatch).T
149
150     self.v = self.v - deltav + self.momentum * self.vprev
151     self.vprev = deltav
152
153     #function to run the MPL forward
154     def forward(self , inputs):
155         """
156         Runs the network forwards.
157         Inputs: training data.
158         Outputs: Hiddel layer outputs and network output.
159         """
160         #puts bias in the inputs
161         inputss = np.zeros((inputs.shape[0],inputs.shape[1] + 1)
162         )
163         inputss[:,0] = -1
164         inputss[:,1:] = inputs
165
166         #caculate the hidden layer output
167         outputhm = self.v.T @ inputss.T
168
169         outputhm = self.activate(outputhm)
170
171         #puts bias in the hiddel layer outputs

```

```

171         outputhmi = np.zeros((self.hidden + 1, inputs.shape[0]))
172         outputhmi[0, :] = -1
173         outputhmi[1:, :] = outputhm
174
175         #calculate the outputs of the MLP
176         outputom = self.w.T @ outputhmi
177         outputom = self.activate(outputom)
178
179         #returns the hidden layer output and the output output
180         return outputhm, outputom
181
182     def confusion(self, inputs, targets):
183         """
184         Calculates the accuracy score and confusion matrix for
185         the network.
186         Inputs: Data and labels that are going to be assessed.
187         Outputs: accuracy score and confusion matrix.
188         """
189         #create confuison matrix
190         conf = np.zeros((self.noutput, self.noutput))
191
192         #creates a value to store the number of correct
193         #classifications
194         correct = 0
195
196         #runs the moodel forwards
197         outputHi, outputOu = self.forward(inputs)
198
199         #finds the target result and the estimated resiltis
200         tarind = np.argmax(targets, axis=1)
201         estind = np.argmax(outputOu, axis=0)
202
203         #for loop runs through the test input
204         for i in range(0, inputs.shape[0]):
205             #puts increments the values in the confusion matrix
206             #based on the results
207             conf[tarind[i]][estind[i]] = conf[tarind[i]][estind[
208                 i]] + 1
209
210             #if a value is placed in the diag then
211             #classification is correct
212             if tarind[i] == estind[i]:
213                 correct = correct + 1

```

```

212         #gets the percentage of correct classifications
213         percor = correct/inputs.shape[0]
214
215         return percor * 100, conf
216
217     def activate(self, inputs):
218         """
219         Calculates the activation function for an output.
220         Inputs: outputs of a layer in the network
221         Outputs: the activation function result.
222         """
223         return 1 / (1 + ( np.exp( - inputs)))
224
225     L = 40
226     label_filename = 'Ising2DFM_reSample_L40-T=All_labels.pkl'
227     dat_filename = 'Ising2DFM_reSample_L40-T=All.pkl'
228
229     # Read in the labels
230     with open(label_filename, "rb") as f:
231         labels = pickle.load(f)
232
233     # Read in the corresponding configurations
234     with open(dat_filename, "rb") as f:
235         data = np.unpackbits(pickle.load(f)).reshape(-1, 1600).
236         astype("int")
237
238     # Set spin-down to -1
239     data[data == 0] = -1
240
241     #generate onehot vector
242     target = np.zeros((np.shape(data)[0],2));
243     for x in range(0,2):
244         indices = np.where(labels==x)
245         target[indices,x] = 1
246
247     # Set up slices of the dataset
248     ordered = slice(0, 70000)
249     critical = slice(70000, 100000)
250     disordered = slice(100000, 160000)
251
252     #critical data
253     critical_data = data[critical]
254     critical_label = target[critical]
255

```

```

256 #ordered and disordered data
257 datawo = np.concatenate((data[ordered], data[disordered]))
258 labelswo = np.concatenate((target[ordered], target[disordered]))
259
260
261
262 #randomly shuffle the data
263 order = list(range(np.shape(datawo)[0]))
264 np.random.shuffle(order)
265 datawo = datawo[order,:]
266 labelswo = labelswo[order]
267
268
269 # Split data into k sets
270 foldsm = []
271 foldst = []
272 for i in range(0,13000,1300):
273     foldsm.append(datawo[i:i+1300,:])
274     foldst.append(labelswo[i:i+1300,:])
275
276
277
278 #arrays to store data
279 test_ac = np.zeros(10)
280 train_ac = np.zeros(10)
281 citical_ac = np.zeros(10)
282 k = np.zeros(10)
283 ac_test_keras = np.zeros(10)
284 ac_train_keras = np.zeros(10)
285 ac_critical_keras = np.zeros(10)
286
287 for i in range(0,10):
288
289     # Test data is used to evaluate how good the completely
290     # trained network is.
291     test = foldsm[i]
292     test_targets = foldst[i]
293     sum = 0
294
295     valind = np.random.randint(9)
296     if valind >= i:
297         valind = valind + 1
298
299     # Validation checks how well the network is performing and
300     # when to stop

```

```

299     valid = foldsm[valind]
300     valid_targets = foldst[valind]
301
302     sumind = 0
303     for j in range(0,10):
304         if j != i and j != valind: sumind = sumind + foldsm[j].
shape[0]
305
306     #Training data to train the network
307     train = np.zeros((sumind,1600))
308     train_targets = np.zeros((sumind,2))
309     placedind = 0
310     for j in range(0,10):
311         if j != i and j != valind:
312             train[placedind:placedind+foldsm[j].shape[0],:] =
foldsm[j]
313             train_targets[placedind:placedind+foldsm[j].shape
[0],:] = foldst[j]
314             placedind = placedind + foldsm[j].shape[0]
315
316
317     # initialize the network
318     mlp1 = mlp()
319
320
321     #run the mlp
322     k[i]=mlp1.earlystopping(train, train_targets, valid,
valid_targets)
323
324
325     #array to store accuracy scores
326     test_ac[i], mat = mlp1.confusion(test, test_targets)
327     train_ac[i], train_mat = mlp1.confusion(train, train_targets)
328     citical_ac[i], citical_mat = mlp1.confusion(critical_data,
critical_label)
329
330
331     if mlp1.run_keras:
332         #using kera for comparison
333         model = Sequential()
334         model.add(Dense(mlp1.hidden, input_dim=mlp1.ninput,
activation='relu'))
335         model.add(Dense(2, activation='sigmoid'))
336         model.compile(loss='binary_crossentropy', optimizer='sgd
', metrics=['accuracy'])

```

```

337         earlystop = callbacks.EarlyStopping(monitor='val_acc',
338         min_delta=1e-7, patience=20, verbose=0, mode='auto')
339         model.fit(train, train_targets, epochs=1000, verbose=0,
340         batch_size=100, validation_data=(valid, valid_targets),
341         callbacks=[earlystop])
342         keras_pred_test = model.predict(test)
343         keras_pred_train = model.predict(train)
344         keras_pred_critical = model.predict(critical_data)
345
346         #define values to be one or the other class
347         keras_pred_test[keras_pred_test < 0.5] = 0
348         keras_pred_test[keras_pred_test >= 0.5] = 1
349
350         keras_pred_train[keras_pred_train < 0.5] = 0
351         keras_pred_train[keras_pred_train >= 0.5] = 1
352
353         keras_pred_critical[keras_pred_critical < 0.5] = 0
354         keras_pred_critical[keras_pred_critical >= 0.5] = 1
355
356         ac_test_keras[i] = metrics.accuracy_score(test_targets,
357         keras_pred_test) * 100
358         ac_train_keras[i] = metrics.accuracy_score(train_targets
359         , keras_pred_train) * 100
360         ac_critical_keras[i] = metrics.accuracy_score(
361         critical_label, keras_pred_critical) * 100
362
363         #print results for each fold
364         print("
365         _____\n")
366         print("Own MLP:")
367         print("Train set accuracy: %.4f%% Test set accuracy: %.4f%%
368         Critical set accuracy: %.4f%%\n" %(train_ac[i], test_ac[i],
369         citical_ac[i]))
370
371         if mlp1.run_keras:
372             print("keras MLP:")
373             print("Train set accuracy: %.4f%% Test set accuracy: %.4
374             f%% Critical set accuracy: %.4f%%\n" %(ac_train_keras[i],
375             ac_test_keras[i], ac_critical_keras[i]))
376
377         print("
378         _____")

```

```

370
371 #print results
372 print("Created MLP:")
373 print("Average accuracy train data: %.2f%%. Min test: %.2f%%.
      Max test: %.2f%%. Test std: %.2f%%"
374 %(np.mean(train_ac), np.min(train_ac), np.max(train_ac), np.std(
      train_ac)))
375 print("Average accuracy test data: %.2f%%. Min test: %.2f%%. Max
      test: %.2f%%. Test std: %.2f%%"
376 %(np.mean(test_ac), np.min(test_ac), np.max(test_ac), np.std(
      test_ac)))
377 print("Average accuracy on critical data %.2f%%. Min test: %.2f
      %%. Max test: %.2f%%. Test std: %.2f%%"
378 %(np.mean(critical_ac), np.min(critical_ac), np.max(critical_ac),
      np.std(critical_ac)))
379
380
381 if mlp1.run_keras:
382     print("keras MLP:")
383     print("Average accuracy train data: %.2f%%. Min test: %.2f
      %%. Max test: %.2f%%. Test std: %.2f%%"
384     %(np.mean(ac_train_keras), np.min(ac_train_keras), np.max(
      ac_train_keras), np.std(ac_train_keras)))
385     print("Average accuracy test data: %.2f%%. Min test: %.2f%%.
      Max test: %.2f%%. Test std: %.2f%%"
386     %(np.mean(ac_test_keras), np.min(ac_test_keras), np.max(
      ac_test_keras), np.std(ac_test_keras)))
387     print("Average accuracy on critical data %.2f%%. Min test:
      %.2f%%. Max test: %.2f%%. Test std: %.2f%%"
388     %(np.mean(ac_critical_keras), np.min(ac_critical_keras), np.
      max(ac_critical_keras), np.std(ac_critical_keras)))
389
390
391 #plot the accuracy scores
392 plt.plot(train_ac, 'b', label='Created MLP train')
393 plt.plot(test_ac, '—b', label='Created MLP test')
394 plt.plot(critical_ac, '—b', label='Created MLP critical')
395 if mlp1.run_keras:
396     plt.plot(ac_train_keras, 'r', label='keras MLP train')
397     plt.plot(ac_test_keras, '—r', label='keras MLP test')
398     plt.plot(ac_critical_keras, '—r', label='keras MLP critical')
399
400
401 if mlp1.run_keras:
402     plt.title("Accuracy scores for test, training and critical

```



```

    data for the created and the keras MLP during a 10-fold",
    fontsize = 16)
403 else:
404     plt.title("Accuracy scores for test, training and critical
    data for the created MLP during a 10-fold", fontsize = 16)
405 plt.legend(fontsize=16)
406 plt.xlabel('Fold number', fontsize=15)
407 plt.ylabel('Accuracy score [%]', fontsize=15)
408 plt.tick_params(labelsize=15)
409
410
411 plt.show()

```

References

- [1] Morten Hjorth-Jensen. Data analysis and machine learning: Logistic regression. https://compphysics.github.io/MachineLearning/doc/pub/LogReg/html/_LogReg-bs009.html. Accessed: 2018-10-29.
- [2] Morten Hjorth-Jensen. Data analysis and machine learning: Neural networks, from the simple perceptron to deep learning and convolutional networks. <https://compphysics.github.io/MachineLearning/doc/pub/NeuralNet/html/NeuralNet-bs.html>. Accessed: 2018-11-01.
- [3] Stephen Marshland. *Machine Learning: An Algorithmic Perspective*, chapter 4, pages 71–108. Chapman & Hall CRC, 2 edition, 2015.
- [4] Pankaj Mehta, Marin Bukov, Ching-Hao Wang, Alexandre G.R. Day, Clint Richardson, Charles K. Fisher, and David J. Schwab. A high-bias, low-variance introduction to machine learning for physicists. 2018.
- [5] Wikipedia. Ising model. https://en.wikipedia.org/wiki/Ising_model#No_phase_transitions_in_finite_volume. Accessed: 2018-10-28.