Exploring the hyperspace of Machine Learning parameters

Eirik Ramsli Hauge, Joakim Kalsnes, Hans Mathias Mamen Vege

November 13, 2018

Abstract

An implementation of a multilayer perceptron neural network and logistic regression is presented, in which the effects of tuning different hyperparameters is made studying the 1D and 2D Ising model. As a verification, comparisons between 1D Ising model applied to ordinary least squares, Ridge and Lasso regression and the MLP is made, to which the MLP appears outperform the linear regression in certain aspects. For the 2D Ising model, we find that the optimal parameters for MLP seems to lie around a constant learning rate $\eta = 0.001$, L^2 regularization strength $\lambda = 0.1$, mini batch size in SGD(Stochastic gradient descent) for $N_{\text{mb}} = 20$, 20 hidden layer neurons. This produces after 500 epochs a accuracy of

1 Introduction

With the rapid expansion of computer power, data science is becomming an ever bigger part of modern society. Among the methods used to process data, some of the most popular are regression and classification. Through this paper we will look closely at the use of these methods when applied to data from the Ising model for spins. First of, we will use linear, ridge and lasso regression to estimate the coupling constant for the one-dimensional Ising model. Secondly, we move on to logistic regression and determining the phase of a spin matrix created by the two-dimensional Ising model. The latter problem is a classification problem and we will look further into it by creating a neural net. To learn more about the properties of a neural net, we will start by using the one-dimensional Ising model to find the optimal weights and biases in a regression case. Then we will apply what we learned by repeating the same classification as the logistic case, but this time we will train a neural net with a cross-entropy cost function to perform the task.

Our work is heavily inspired by the work of Metha et al. [6], and therefore it is only natural to compare our results to theirs. Lastly, we will give a detailed and in-depth analyzis of the algorithms used and how our home-made algorithms compares to those of Metha et al. and similar calculations done by standard libraries such as scikit-learn or TensorFlow.

2 Theory

2.1 The Ising Model

The Ising model is a way of modeling phase transitions at finite temperatures of magnetic systems. When modeling, we set up a chain or a lattice of particles and allow them to have either spin up or spin down. From this, one can sample energy and magnetization, and

measure several quantities such as the heat capacity or magnetic susceptibility. Our focus will be on predicting the energy coupling constant for a 1D lattice, and the phase of a 2D-lattice.

Periodic boundary conditions is given in both cases, such that $j = N = 0$, where j is the lattice site and N is the lattice size.

2.1.1 1-dimensional Ising model

Energy for a 1 dimensional Ising model is given as

$$E = -J \sum_{j=1}^{N} s_j s_{j+1} \tag{1}$$

where the N is the number of particles(or lattice size) and $s_j = \pm 1$ is the j'th spin. Our goal will be to predict J, but in order to do so, we must recast the problem as a linear regression problem.

We begin by labeling each site as coupled with J.

$$E_{\text{model}}[\boldsymbol{s}^i] = -\sum_{j=1}^{N} \sum_{k=1}^{N} J_{j,k} s_j^i s_k^i, \tag{2}$$

where i is the index over lattice configurations. The coupling strength $J_{j,k}$ can now be cast as a matrix, and we end up with

$$E_{\text{model}}^i \equiv \boldsymbol{X}^i \cdot \boldsymbol{J}, \tag{3}$$

where $\boldsymbol{X}^i$ is the design matrix consisting of all two-body interactions $\{s_j^i s_k^i\}_{j,k=1}^N$, and $\boldsymbol{J}$ the weight matrix we wish to find later using machine learning techniques.

2.1.2 2-dimensional Ising model

The 2D Ising model has its energy stated as,

$$E = -J \sum_{<kl>}^{N} s_k s_l, \tag{4}$$

where $< kl >$ indicates a sum over the nearest neighbors. That is, written out,

$$E = -J \sum_{i,j}^{N} 2 s_{i,j} (s_{i+1,j} + s_{i-1,j} + s_{i,j+1} + s_{i,j-1}), \tag{5}$$

where we have used the symmetry that $s_{i,j} s_{i+1,j} = s_{i+1,j} s_{i,j}$. J is, as in the 1D model, a coupling constant, but will not be our main focus when studying the 2D Ising model. This time, we will focus on its property of exhibiting phase transitions. Below a critical temperature of $T_C \approx 2.269$ found analytically by [8], the lattice will exhibit an ordered state, one in which the spins is *locked* or *frozen* into place. Above T_C the lattice exhibit a disordered phase, as the spins will be fluctuating randomly.

We will investigate the classification of states below $T < 2.0$ and $T > 2.5$. The phase for states between we will dub as being in a *critical phase*.

2.2 Logistic regression

In project 1 we used linear regression to predict a continuous output from a set of inputs [3, 11]. We used ordinary least squares regression (OLS), Ridge regression and Lasso regression, where the two latter impose a penalty to the OLS. In this project we will reuse the ideas and code of project 1, but we will also use neural networks to predict continuous variables. In addition we study situations where the outcome is discrete rather than continuous. This is a classification problem, and we will use logistic regression to model the probabilities of the classes.

2.3 Logistic regression

Just like a linear regression model, a logistic regression model computes a weighted sum of the predictor variables, written in matrix notation as $\boldsymbol{X}^T\beta$. However, the logistic regression returns the logistic of this weighted sum as the probabilities. For a classification problem with K classes, the model has the following form [4, p.119],

$$\begin{aligned}
\log\frac{Pr(G=1|X=x)}{Pr(G=K|X=x)} &= \beta_{10} \ + \ \beta_1^T x \\
\log\frac{Pr(G=2|X=x)}{Pr(G=K|X=x)} &= \beta_{20} \ + \ \beta_2^T x \\
&\vdots \\
\log\frac{Pr(G=K-1|X=x)}{Pr(G=K|X=x)} &= \beta_{(K-1)0} \ + \ \beta_{K-1}^T x
\end{aligned} \tag{6}$$

It is arbitrary which class is used in the denominator for the log-odds above. Taking the exponential on both sides and solving for $Pr(G=k|X=x)$ gives the following probabilities,

$$\begin{aligned}
Pr(G=k|X=x) &= \frac{exp(\beta_{k0} \ + \ \beta_k^T x)}{1+\sum_{l=1}^{K-1} exp(\beta_{l0} \ + \ \beta_l^T x)}, \ k=1,\ldots,K-1, \\
Pr(G=K|X=x) &= \frac{1}{1+\sum_{l=1}^{K-1} exp(\beta_{l0} \ + \ \beta_l^T x)},
\end{aligned} \tag{7}$$

and the probabilities sum to one. To classify the output we choose the class with the highest probability.

2.3.1 Fitting logistic regression model

The usual way of fitting logistic regression models is by maximum likelihood. The log-likelihood for N observations is defined as,

$$l(\theta) \ = \ \sum_{i=1}^{N} \log p_{g_i}(x_i;\theta), \tag{8}$$

where $p_k(x_i;\theta) \ = \ Pr(G=k|X=x_i;\theta)$ and $\theta \ = \ \{\beta_{10}, \beta_1^T, \ldots, \beta_{(K-1)0}, \beta_{K-1}^T\}$.

One very common classification problem is a situation with binary outcomes, either it happens or it does not. As we see from Equation 6 above, setting K=2 simplifies the model considerable, since there will now be only a single linear function. θ in Equation 8 will also

be simplified: $\theta = \beta = \{\beta_{10}, \beta_1^T\}$. The two-class case is what is used in this project, and the following discussion will assume the outcome has two classes.

We start by coding the two-class g_i with a 0/1 response y_i, where $y_i = 1$ when $g_i = 1$, and $y_i = 0$ when $g_i = 2$. Next, we let $p_1(x;\theta) = p(x;\beta)$, and $p_2(x;\theta) = 1 - p(x;\beta)$. The log-likelihood can then be written

$$\begin{aligned} l(\beta) &= \sum_{i=1}^{N} \{y_i \log p(x_i;\beta) + (1-y_i)\log(1-p(x_i;\beta))\} \\ &= \sum_{i=1}^{N} \{y_i \log \frac{p(x_i;\beta)}{1-p(x_i;\beta)} + \log(1-p(x_i;\beta))\} \\ &= \sum_{i=1}^{N} \{y_i \beta^T x_i + \log(1 - \frac{1}{1+exp(-\beta^T x_i)}\} \\ &= \sum_{i=1}^{N} \{y_i \beta^T x_i + \log(\frac{exp(1}{1+exp(\beta^T x_i)}\} \\ &= \sum_{i=1}^{N} \{y_i \beta^T x_i - \log(1+exp(\beta^T x_i))\}. \end{aligned} \tag{9}$$

This is the equation we want to maximize to find the best fit. Following the approach in Géron's book [2], we chose the equivalent approach of minimizing the following,

$$J(\beta) = -\frac{1}{N} \sum_{i=1}^{N} \{y_i \beta^T x_i - \log(1+exp(\beta^T x_i)).\} \tag{10}$$

This is just the negative of Equation 9, divided by the number of samples. This is our cost function, and dividing by the number of training samples finds the mean cost.

As with all cost functions, the goal is to minimize it. This introduces it's own potential pitfalls such as overfitting. In order to avoid that, we can introduce regularizations. Let us begin by looking at the optimization problem.

2.4 Optimization

As well as in logistic regression as in neural networks, minimizing(or maximizing depending on your setup) the cost function is a central problem, and can to some extent be stated as *the* problem. Having a method which is both efficient and converges towards the minimum is, luckily, not a new problem in computer science. In the logistic regression optimization, three main methods will be utilized, while in the neural network we will focus on the stochastic conjugate gradient descent. For logistic regression we will use a learning rate optimized gradient descent method, and

Gradient descent measures the local gradient of the cost function, with regards to the weights $\boldsymbol{w}$(or β. Since the gradient goes in the direction of fastest increase, we will go in the opposite direction, i.e. negative gradient. We start by choosing random values for $\boldsymbol{w}$ (since our cost function is convex any choice should give correct results), calculate the gradient, update the $\boldsymbol{w}$ values, and do this iteratively until the algorithm converges to a minimum. The size of the steps is important, and is determined by the learning rate. If the learning rate

is too small, we will need many iterations which is time consuming. However, if the learning rate is too high, we might overshoot and miss the minimum. One way to choose the learning rate is too let it depend on the size of the gradient. If the gradient is large, i.e a steep slope, the learning rate can be relatively high. When the gradient is small, the learning rate is also small.

2.4.1 Gradient descent

Gradient descent is set up in a general fashion as, utilizing an optimized learning rate η_k find and a scaling parameter γ. The scaling parameter γ is introduced in order to prevent results from blowing up.

Algorithm 1 Gradient descent.

Input: $\boldsymbol{X}$, $\boldsymbol{y}$, $\boldsymbol{w}_0$, η
Set initial weights, $\boldsymbol{w} = \boldsymbol{w}_0$
Set previous gradient with current, $\nabla_0 = \nabla_1$
while $i < N_{\text{max}}$ and $||\nabla \mathcal{C}(\boldsymbol{w})|| < \varepsilon$ **do**
    $z = \boldsymbol{X} \cdot \boldsymbol{w}$
    $p = \sigma(z)$
    $\nabla_0 = \nabla_1$, set previous gradient to current.
    $\nabla_1 = -\boldsymbol{X}^T \cdot (\boldsymbol{y} - \boldsymbol{p})/\gamma + \lambda f_L(\boldsymbol{w})/\gamma$
    $\boldsymbol{w}_0 = \boldsymbol{w}$
    Update learning parameter, η_k
    $\boldsymbol{w} = \boldsymbol{w}_0 - \eta_k \nabla_1$, update weights.
end while
Return $\boldsymbol{w}$

The full implementation with the optimized learning rate η_k can be seen in the article Barzilai and Borwein [1].

2.4.2 Stochastic gradient descent

Stochastic gradient descent is similar to gradient descent, expect that we first randomly shuffle our data, then divide our data into mini batches, N_{mb}. Then we run gradient descent on each of the mini batches, and take the average of the outputted gradients as our gradient descent step. We repeat this N_{epochs}.

Algorithm 2 Stochastic gradient descent(SGD).

for doi_e in N_{epochs} epochs
    Shuffle data $\boldsymbol{X}_{\text{train}}$, $\boldsymbol{y}_{\text{train}}$
    Split into mini batches, $\boldsymbol{X}_{\text{train}}$, $\boldsymbol{y}_{\text{train}}$
    for doi_{mb} in mini batches,
        Perform gradient descent step, and retrieve $\nabla \boldsymbol{w}_{i_{\text{mb}}}$
    end for
    Take the average of the $\nabla \boldsymbol{w}_{i_{\text{mb}}}$ and update the weight matrix $\boldsymbol{w}$.
end for

2.5 Regularization

Like we introduced Lasso and Ridge regression to avoid over fitting in Project 1, we can add a penalty term to the cost function in equation (10).

2.5.1 L^1 regularization

The L^1 regularization utilizes the Taxi-cab metric,

$$||\boldsymbol{a}||_1 = |a_0| + \cdots + |a_{n-1}| = \sum_{i=0}^{n-1} |a_i|,$$

and is defined as

$$\lambda ||\boldsymbol{w}||_1, \tag{11}$$

with $\boldsymbol{w}$ being the weight matrix. This is equivalent with β as seen in the logistic regression. Its derivative is given as,

$$\lambda \operatorname{sign}(\boldsymbol{w}) \tag{12}$$

where sign is simple the sign of $\boldsymbol{w}$.

For logistic regression, this becomes in the gradient of the Geron cost function (10),

$$\frac{\partial J(\beta)}{\partial \beta} = \frac{1}{N}\boldsymbol{X}^T(\boldsymbol{p} - \boldsymbol{y}) \ + \lambda \cdot \operatorname{sign}(\beta) \tag{13}$$

2.5.2 L^2 regularization

The L^2 regularization is given as the Euclidean norm of the weight matrix,

$$||\boldsymbol{a}||_2 = \left(\sum_{i=0}^{n-1} a_i^2 \right)^{1/2}, \tag{14}$$

and is given as

$$\lambda ||\boldsymbol{w}||_2^2, \tag{15}$$

with its following derivative

$$\lambda \cdot 2\boldsymbol{w}. \tag{16}$$

The 2 in front is often offset by redefining the L^2 norm with a factor half, and will not affect the final outcome.

Implementing the L^2 norm in the gradient of the Geron cost function(10), we get

$$\frac{\partial J(\beta)}{\partial \beta} = \frac{1}{N}\boldsymbol{X}^T(\boldsymbol{p} - \boldsymbol{y}) \ + \lambda \cdot 2\beta \tag{17}$$

2.5.3 Elastic net regularization

Elastic net regularization utilizes a linear combination of L^1 and L^2 regularization, and consists of adding a term

$$\lambda_1||\boldsymbol{w}||_1 + \lambda_2 + ||\boldsymbol{w}||_2^2 \tag{18}$$

The derivative of this w.r.t. $\boldsymbol{w}$ is simply the combined derivatives of L^1 and L^2. For our purposes we will set $\lambda_1 = \lambda_2$ in order to avoid having the hyper parameter space become too large.

2.6 Neural Networks

Among the many methods developed for machine learning, neural networks, and especially deep neural networks, are among the most popular. Neural networks were suggested already in 1943 [5] and have had many renaissances since. Currently we are experiencing such a renaissance, but in contrast to earlier periods of resurfaced interest, we now have the computer power to use neural nets efficiently.

A neural net bases itself loosely upon the biological model of neurons communicating together in the brain. A neuron cell contains most of what a normal cell contains, but it also has a long tail called an axon and some antenna like extension called dendrites. The axon of one cell can extend quite far and attach to some of the dendrites of another neural cell. Thus, the biological neural net consists of neural cells receiving input through their dendrites from many other cells and sending output through one output [2, p. 257].

The computed neural networks works in a similar way. We construct "neurons" or "nodes" which are ordered in different layers where each neuron in one layer is connected to all neurons in the next layer. Initially, we start with an input layer which we feed information. Following this initial layer we have one or many hidden layers before we reach the output layer. A neural network with two or more hidden layers are called deep neural networks [2, p. 263]. Each neuron contains an activation function which determines the strength of the output. In the early days, a step function was used as the activation function. However, one has found that the use of a activation function with a gradient, such as the logistic function used in logistic regression, gives a better neural net. This is due to the fact that we now can apply gradient descent when optimizing the neural net which is discussed below.

To activate a neuron, it needs an input. This input is provided by all the neurons in previous layers through "wires" connecting the neurons (think of the axon to a dendrite). Each of these "wires" is weighted and all connections between one layer and the next is affected by a bias term. Thus, the output of a neuron is given as [2, p. 260]

$$a = \sigma(z) \tag{19}$$

with

$$z = \boldsymbol{w}^T\boldsymbol{x} + b \tag{20}$$

where a is the output, $\boldsymbol{w}$ are the weights, $\boldsymbol{x}$ are the inputs and b is the bias term. Note that if we had used a step function instead of the logit(sigmoid) function in the equation above, the neuron would either give an output of 1 or nothing, i.e. 0. When we instead use the sigmoid function, the output can be in the range of 0 to 1.

Once all neurons have been calculated in a layer, we can move on to the next and continue until we reach our output layer. Each layer can have as many neurons as the user wants. Optimizing the number of neurons in each layer is an art and requires both experience and a bit of luck. The output layer needs one neuron for each class we wish to identify.

After initial calculation of the outer layer, you will most likely have an answer that is completely rubbish. It is clear that we have to optimize the neural net. As each neurons activation function is calculated using equation (19), we see that we can optimize the weights between the neurons and the bias between the layers. This is done through a method called backwards propagation where one uses the cost function to identify the magnitude of the error (the cost) of a neural net and then one goes backwards through the neural net to update the weights and biases.

The backpropegation is then summed up as following(heavily influenced by the work of Nielsen [7]),

- Compute the output error vector for the final layer (L) given by

$$\delta^L = \nabla_a \mathcal{C} \odot \sigma^{'}(\boldsymbol{w}^T \cdot \boldsymbol{x} + b)$$

  where $\mathcal{C}$ is the cost function and ∇_a is a vector who has components that are the partial derivatives $\frac{\partial \mathcal{C}}{\partial a_j^L}$ where a_j is the j'th output found by using equation (19) for a.

- Go back through all the previous layers l = L-1, L-2, ..., 2 and compute

$$\delta^l((\boldsymbol{w}^{l+1})^T \delta^{l+1}) \odot \sigma^{'}(\boldsymbol{w}^T \cdot \boldsymbol{x} + b)$$

  this is where we back propagate the error.

- Finally, find the gradient of the cost function for the two parameters we want to change, $\boldsymbol{w}$ and b by:

$$\frac{\partial \mathcal{C}}{\partial \boldsymbol{w}_{jk}^l} = a_k^{l-1} \delta_j^l$$
$$\frac{\partial \mathcal{C}}{\partial b_j^l} = \delta_j^l$$

  where k indicates the column of $\boldsymbol{w}^l$ as j indicates the row.

Once all layers have been adjusted through back propagation, one can run through the whole network again and repeat the process.

In order to understand parts of the back propagation algorithm for L layers and a general layer activation function $a^L(z)$, can take a quick look at how to proceed in deriving it. We start with some cost function $\mathcal{C}(w^L, y)$, where w^L is the output layer weights and y is the *true* values which we want to approach. In order to move towards optimal weights(and biases), we can perform a gradient descent by subtracting the gradient of the cost function w.r.t. the weights and biases. We begin by finding the gradient of the $\mathcal{C}$ for W^L. Using the chain rule we get,

$$\frac{\partial \mathcal{C}}{\partial w^L} = \frac{\partial z^L}{\partial w^L} \frac{\partial a^L}{\partial z^L} \frac{\partial \mathcal{C}}{\partial a^L} \tag{21}$$

From this we simply have to find each of the partial derivatives. Without specifying the cost function and the activation layer, we can only state that the first partial derivative is given as,

$$\frac{\partial z^L}{\partial w^L} = a^{L-1}$$

From this, we quickly see that we need to properly defined the cost function in relation to the ∂^L as we encountered earlier.

2.6.1 Mean square error(MSE) cost function

A common cost function may be the quadratic loss function(MSE) given by.

$$\mathcal{C}_{\text{MSE}} = \frac{1}{2N}\sum_j (y_j - a_j^L)^2 \tag{22}$$

where N are the total numbers of outputs in j and y are the true answers. For a single training sample, this becomes,

$$\mathcal{C}_{\text{MSE}} = \frac{1}{2}(a^L - y)^2 \tag{23}$$

We can then insert this expression into the cost chain rule expression (21), and we get that the partial derivative

$$\begin{aligned}\frac{\partial \mathcal{C}_{\text{CE}}}{\partial w^L} &= a^{L-1}(a^L - y)\sigma'(z^L) \\ &= a^{L-1}\delta^L\end{aligned}$$

If we now take the derivative for layer $L-1$, we get

$$\begin{aligned}\frac{\partial \mathcal{C}_{\text{CE}}}{\partial w^{L-1}} &= \frac{\partial a^{L-1}}{\partial w^{L-1}}\delta^L \\ &= \frac{\partial z^{L-1}}{\partial w^{L-1}}\frac{\partial a^{L-1}}{\partial z^{L-1}}\delta^L \\ &= a^{L-2}\sigma'(z^{L-1})\delta^L\end{aligned}$$

2.6.2 Cross entropy(CE) cost function

The CE cost function is given as

$$\mathcal{C}_{\text{CE}} = -\sum_{i=1} y_i \log a^L \tag{24}$$

with its derivative as

$$\frac{\partial \mathcal{C}_{\text{CE}}}{\partial a^L} = -\sum_{i=1}\frac{y_i}{a^L} \tag{25}$$

If we use softmax as the layer output activation function,

$$z_i = \frac{\exp z_i}{\sum_k \exp(z_k)} \tag{26}$$

with i, k being output classes and its derivative,

$$z_i = z_i(\delta_{ij} - z_j), \tag{27}$$

we can write the initial gradient δ^L as

$$\delta^L = y - a^L \tag{28}$$

If we set the number of output classes to be 2 such that we get binary classification, we get the cost functions for logistic regression,

$$\mathcal{C}_{lg} = -\left(y \log p + (1-y)\log(1-p)\right)$$

for one sample.

2.6.3 Activation layers

The hidden layer activations can greatly affect the outcome of the neural network. We will focus on four different hidden layer activations, the first being **sigmoidal activation**.

$$\sigma(z) = \frac{1}{1+\exp(-z)} \tag{29}$$

with its derivative

$$\sigma'(z) = \sigma(z)(1-\sigma(z)) \tag{30}$$

The **hyperbolic tangens** activation function is given as

$$\sigma_{\tanh}(z) = \tanh(z) \tag{31}$$

with its derivative

$$\sigma'_{\tanh}(z) = 1 - \tanh^2(z) \tag{32}$$

The **relu** or rectifier activation is given as,

$$\sigma_{\text{relu}}(z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{33}$$

with its derivative

$$\sigma'_{\text{relu}}(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{34}$$

The **Heaviside** activation function is given as

$$\sigma_{\text{Heaviside}}(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{35}$$

with its derivative

$$\sigma'_{\text{Heaviside}}(z) = 0 \tag{36}$$

2.6.4 Learning rate

When updating the weights and biases with SGD(Stochastic gradient descent), we did so by a learning rate parameter η. There are several way to define η, with the simplest one having $\eta =$ constant. Another option is one that is inversely decreasing as a function of the epochs. That is, for a given epoch i_e out a total N_{epochs}, we set the learning rate as

$$\eta(t_e) = \eta_0(1 - \frac{i_e}{1 + N_{\text{epochs}}}) \tag{37}$$

This will force the step size to decrease toward 0 as we close in on the maximum number of epochs N_{epochs}.

2.6.5 Weight initialization

When initializing weights and biases, we will look at two ways of how this can be done. The first is through a gaussian distribution, $(0, 1)$ which we will call *large*, as the biases will have large, spread-out distribution.

Then, we will use a gaussian distribution but divided by the number of training samples, $(0, 1/N_{\text{train}})$, dubbing that one to be called *default*, as this is the one we will by default use in our neural network.

The effect of these two is essentially shrinking in the initial search space, and we should expect them to converge at large epoch times.

2.6.6 Measuring the performance

The performance of a neural network(or any classifier), can in its simplest form be measured by the accuracy, which is defined as

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n}, \tag{38}$$

where n is the number of samples we are testing against, I is the indicater function, which returns 1 if the prediction t_i equals the true values y_i.

3 Implementation

Code can be found on [10].

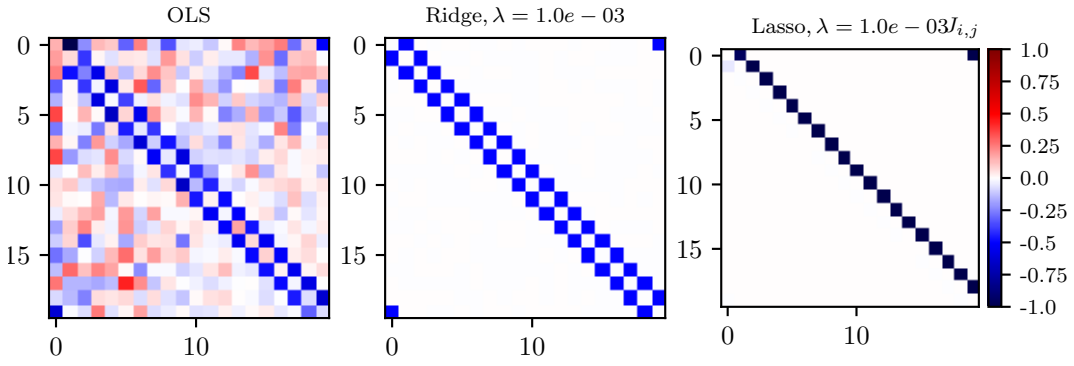
4 Results

Results for two different cases are being presented, one the one-dimensional Ising Model and another for the the two-dimensional Ising model. We begin with looking at the. 1D Ising model.

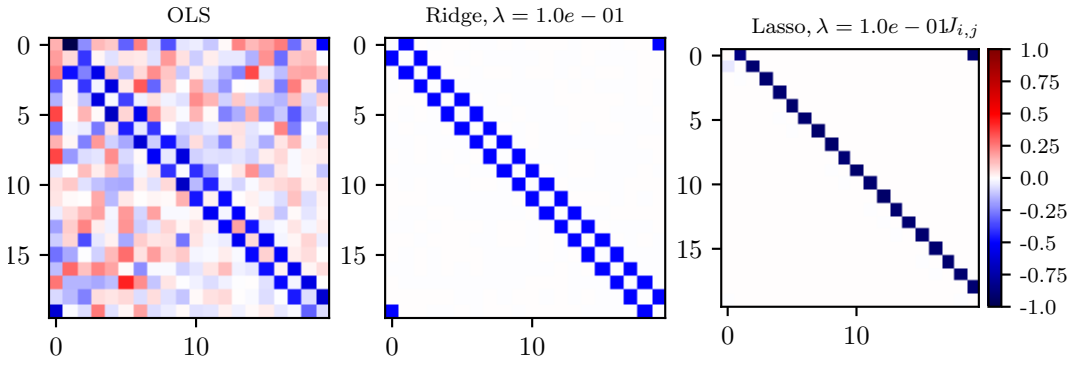
4.1 1D Ising model

4.1.1 Fitting with linear regression

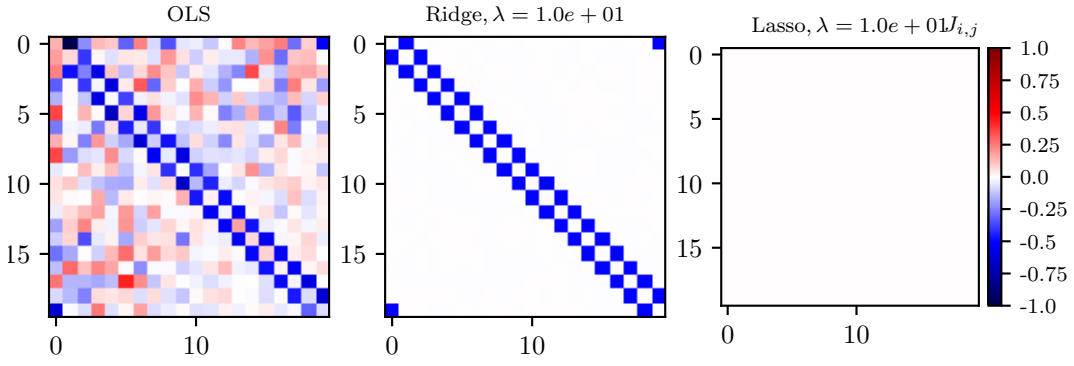
For linear regression we got coefficients of $\boldsymbol{J}$ in (3) as the following presented in figure 1,



(a) $\lambda = 10^{-3}$



(b) $\lambda = 10^{-1}$



(c) $\lambda = 10^{1}$

Figure 1: Heat map plots of the $\boldsymbol{J}$ in (3) retrieved from OLS, Ridge and Lasso. Gathered using $N_{\text{train}} = 5000$.

The R^2 score of the OLS, Ridge and Lasso can be seen in figure 2,

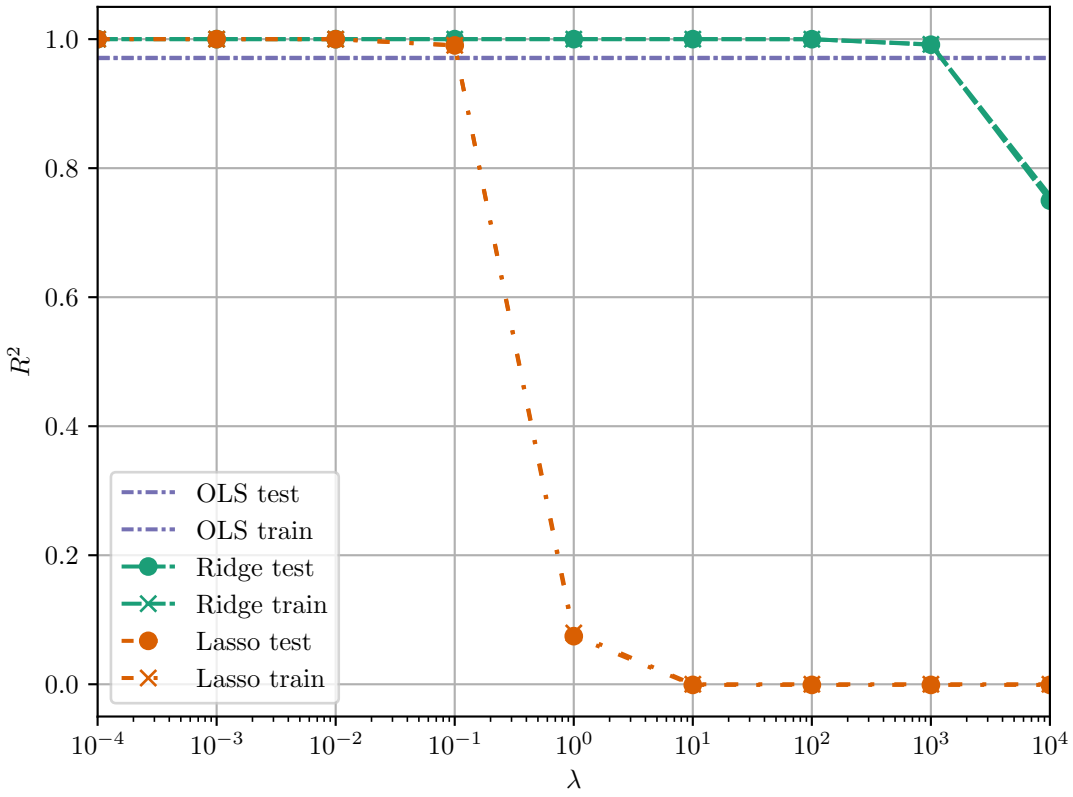
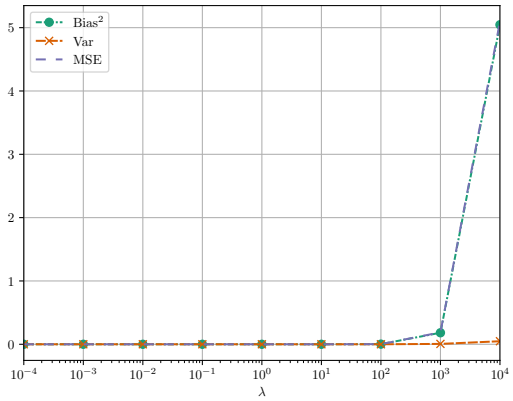
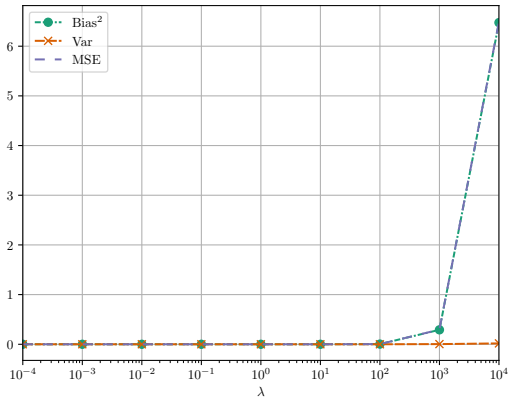


Figure 2: R^2 score for different Ordinary Least Squares(OLS), Ridge and Lasso regression. Retrieved $N_{\text{train}} = 5000$ and $N_{\text{test}} = 5000$ on a 1D Ising model of size $L = 20$.

The bias-variance decomposition for Ridge and Lasso using bootstrap and cross validation can be viewed in figure 3 and 4.

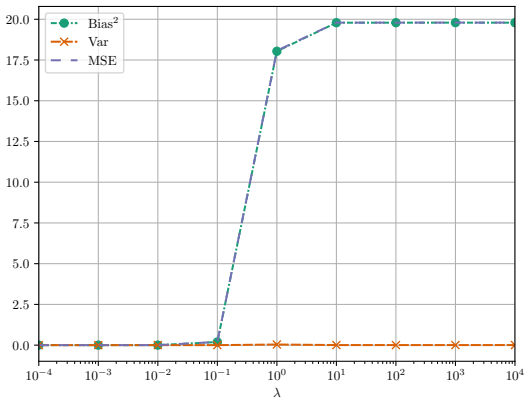


(a) Bootstrap.

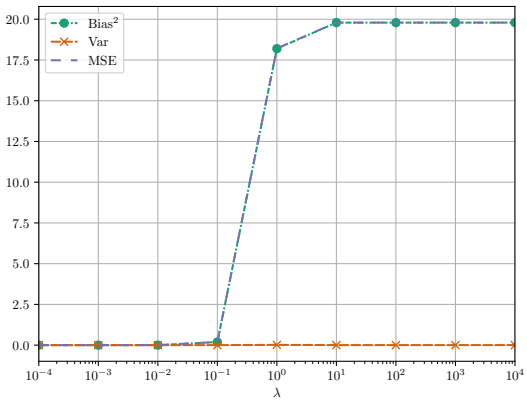


(b) k-fold Cross Validation.

Figure 3: A bias-variance decomposition of Ridge regression using bootstrapping3a and cross-validation3b.



(a) Bootstrap.

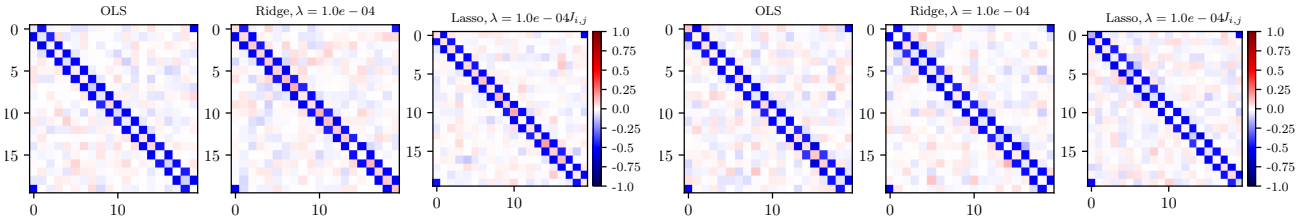


(b) k-fold Cross Validation.

Figure 4: A bias-variance decomposition of Lasso regression using bootstrapping4a and cross-validation4b.

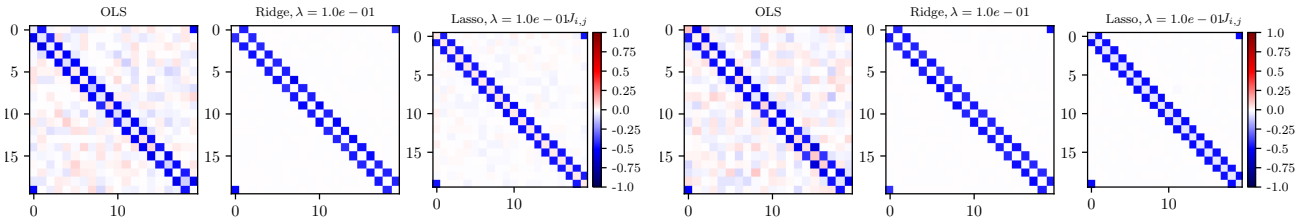
4.1.2 Fitting with a neural network

By setting the output activation function to the identity and by having zero hidden layers, we are essentially performing a regression analysis on the 1D Ising model. We generate the same amount of data by inputing the same RNG(random number generator) seed. A fit using $N_{\text{train}} = 400$, $N_{\text{train}} = 5000$ and $N_{\text{test}} = 5000$ for $\lambda = 10^{-3}, 10^{-1}, 10^{1}$ can be seen in figure **??**.



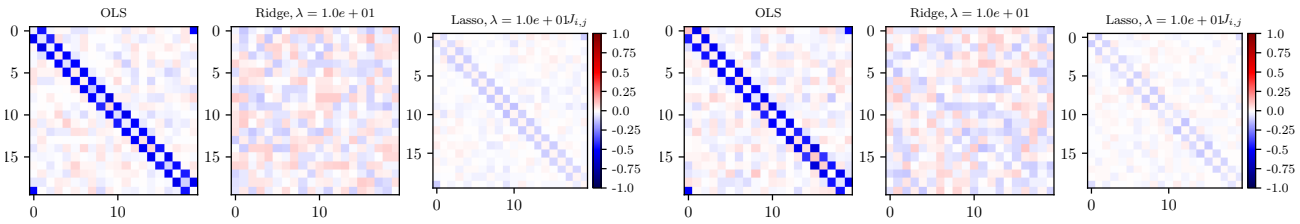
(a) $N_{\text{train}} = 400$, $\lambda = 10^{-3}$

(b) $N_{\text{train}} = 5000$, $\lambda = 10^{-3}$



(c) $N_{\text{train}} = 400$, $\lambda = 10^{-1}$

(d) $N_{\text{train}} = 5000$, $\lambda = 10^{-1}$

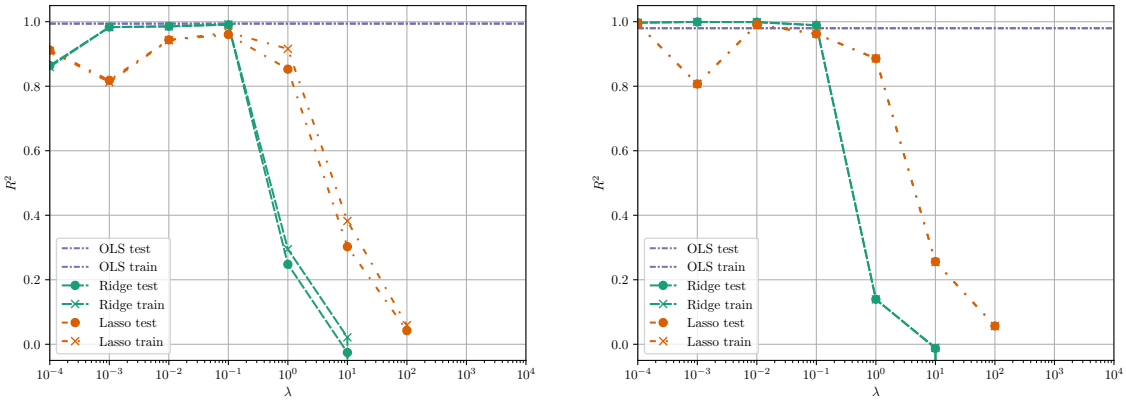


(e) $N_{\text{train}} = 400$, $\lambda = 10^{1}$

(f) $N_{\text{train}} = 5000$, $\lambda = 10^{1}$

Figure 5: Heat map plot of the coefficients of $\boldsymbol{J}$ in (3) using neural networks with different regularizations for $\lambda = 10^{-3}, 10^{-1}, 10^{1}$.

The R^2 score of the neural network using L^1, L^2 and no regularization can be seen in figure 6,



(a) $N_{\text{train}} = 400$ (b) $N_{\text{train}} = 5000$

Figure 6: R^2 score for the neural network using L^1 (Lasso), L^2 (Ridge) and no regularization (OLS). Retrieved $N_{\text{train}} = 400$ on the left and $N_{\text{train}} = 5000$ on the right, for a 1D Ising model of size $L = 20$.

4.2 2D Ising model

As stated in the section about the 2D Ising model 2.1.2, the classification will focus on evaluating the phases of different lattice configurations, and wetter or not it is below or above a critical temperature. We begin by listing the results from the logistic regression.

4.2.1 Classification through logistic regression

In logistic regression we investigated the behavior of the classification and compared it to that of SciKit Learn[9], using the standard logistic regression method[1] and SciKit Learn's SGD(Stochastic Gradient Descent) implementation [2]. This gave the results found in figure 7.

[1]See Logistic Regression documentation
[2]See SGD documentation

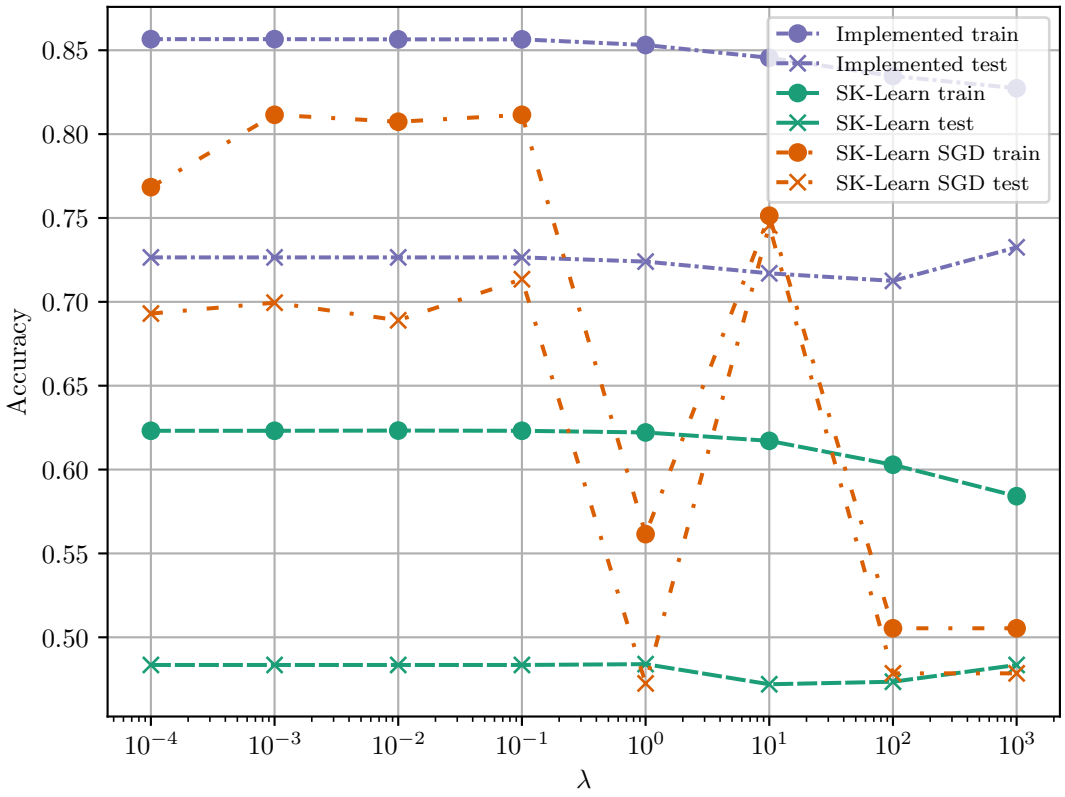


Figure 7: The accuracy for our implementation of logistic regression versus that of SciKit learn.

4.2.2 Classification through neural networks

For classifying the states through a neural network, we looked at several different hyper parameters. All runs were made using $N_{samples} = 10000$ except stated other wise. The training percent was 0.5. We start by comparing two different cost functions and their layer outputs,

- Cross entropy with softmax layer output(24)
- MSE with sigmoidal layer output.(23)

These cost functions following behavior for epochs seen in following figure 8,

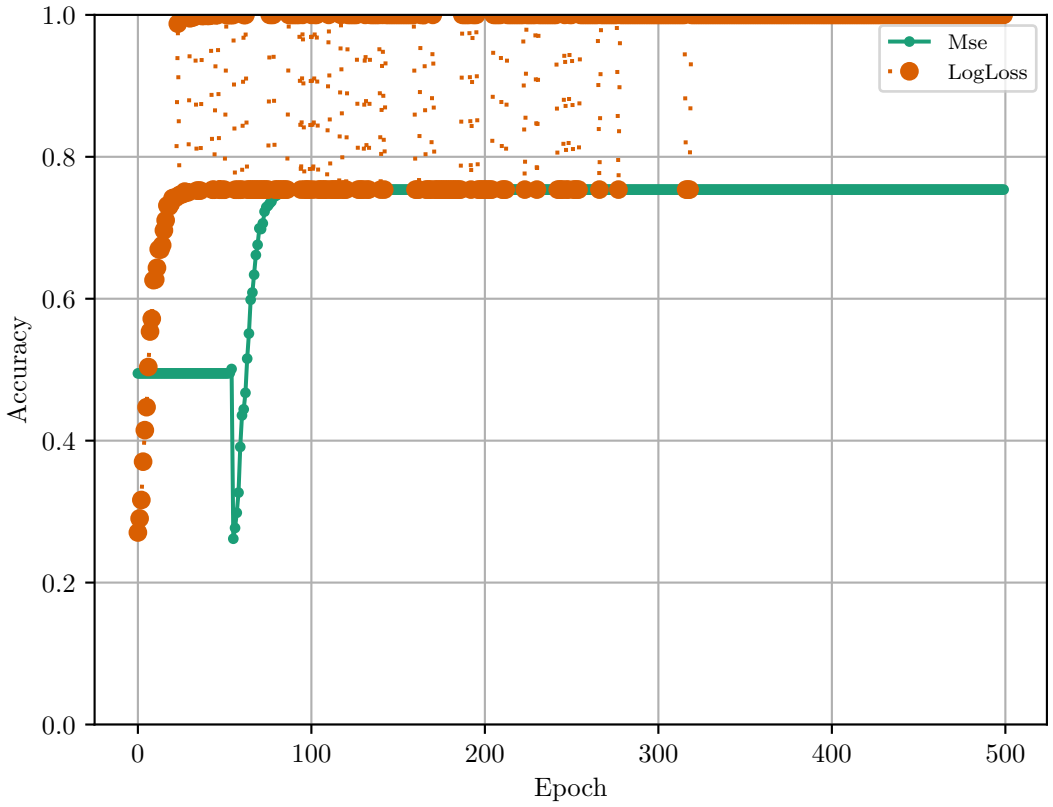


Figure 8: A comparison in accuracy scores between the MSE and (CE) Cross Entropy loss functions over 500 epochs. The output layer for MSE is sigmoidal, the output layer for CE is softmax. The learning parameter was $\eta = 0.001$ and we used the inverse learning rate(37).

We then wish to to investigate the effects of having different initial weights. Given the initial weights *large* and *default* as listed in section 2.6.5, we get the results as seen in figure 9,

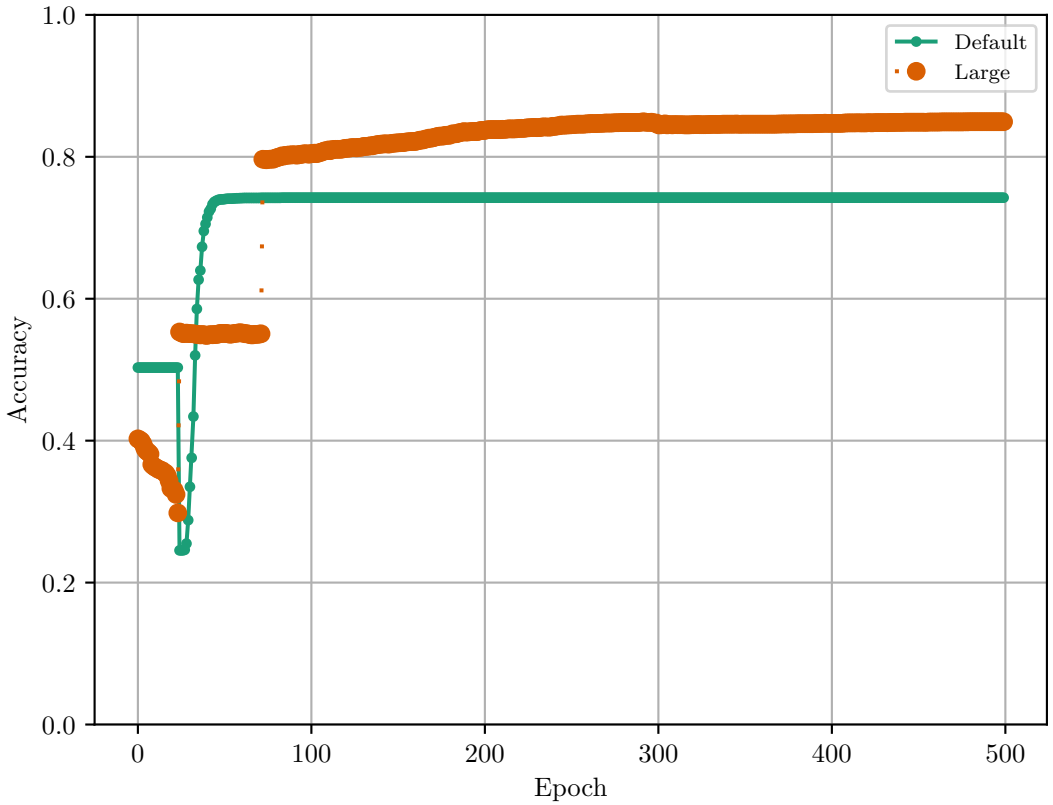


Figure 9: A comparison in accuracy scores between the initial weights *large* and *default* as listed in section 2.6.5. The run was for 500 epochs. The cost function was set to cross entropy and had softmax output activation. The learning parameter was $\eta = 0.001$ and we used the inverse learning rate(37).

An investigation into different layer activations2.6.3 was performed for both the MSE- and the CE-cost function. The results from MSE can be seen in figure

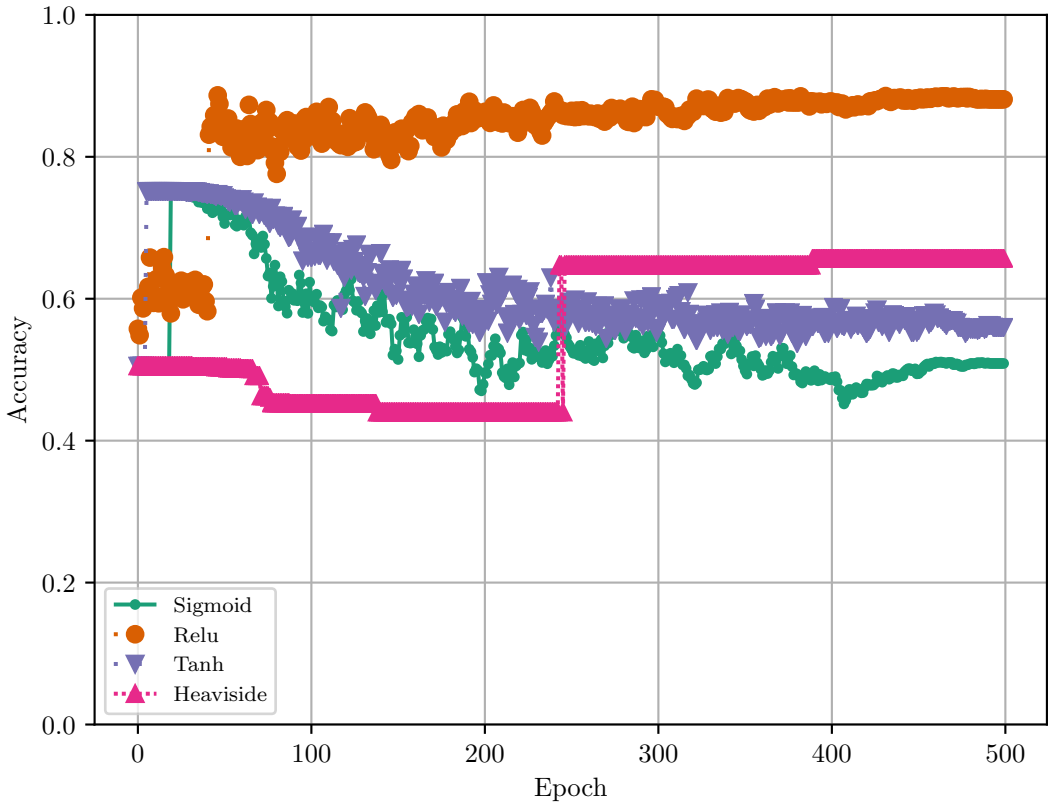


Figure 10: A comparison in accuracy scores between the hidden layer activation functions(see section 2.6.3) for MSE as cost function. The run was for 500 epochs. The learning rate was set with the inverse learning rate (37) with an $\eta_0 = 0.001$ and $\lambda = 0.0$.

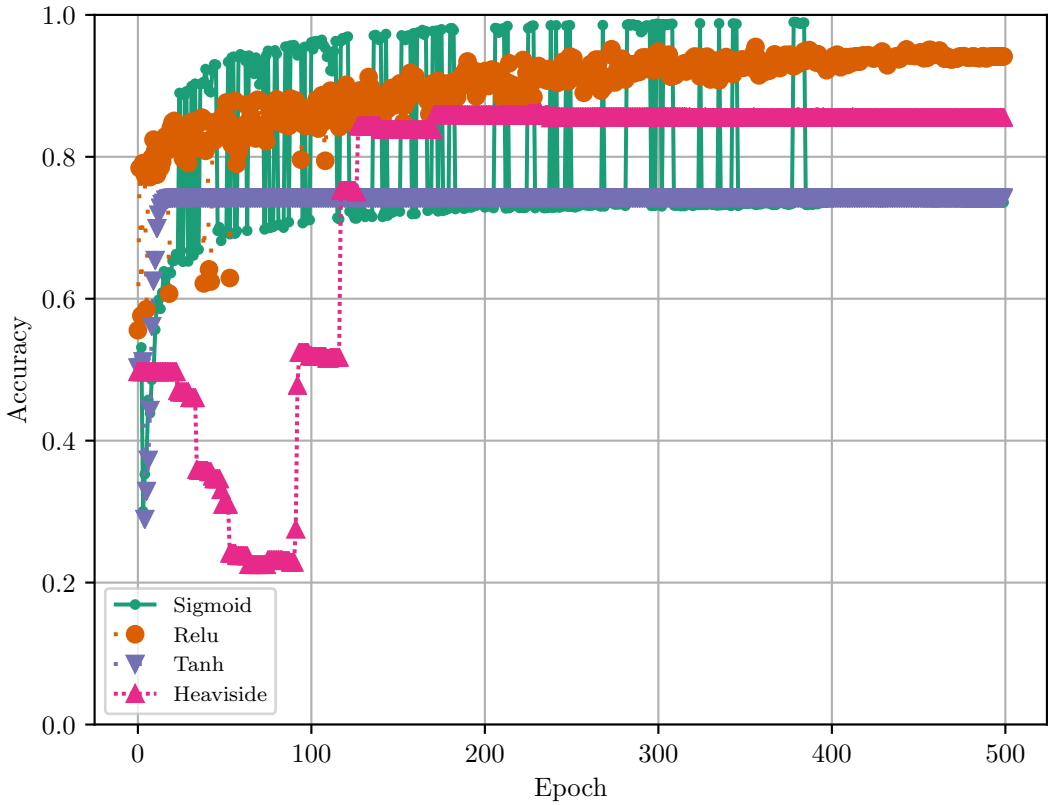


Figure 11: A comparison in accuracy scores between the hidden layer activation functions(see section 2.6.3) for cross entropy as cost function. The run was for 500 epochs. The learning rate was set with the inverse learning rate (37) with an $\eta_0 = 0.001$ and $\lambda = 0.0$.

We then move on to an investigation for different L^2 regularization strengths λ versus different constant learning rates η. A run with 500 epochs, cross entropy and sigmoidal hidden layer activation can be seen in figure 12,

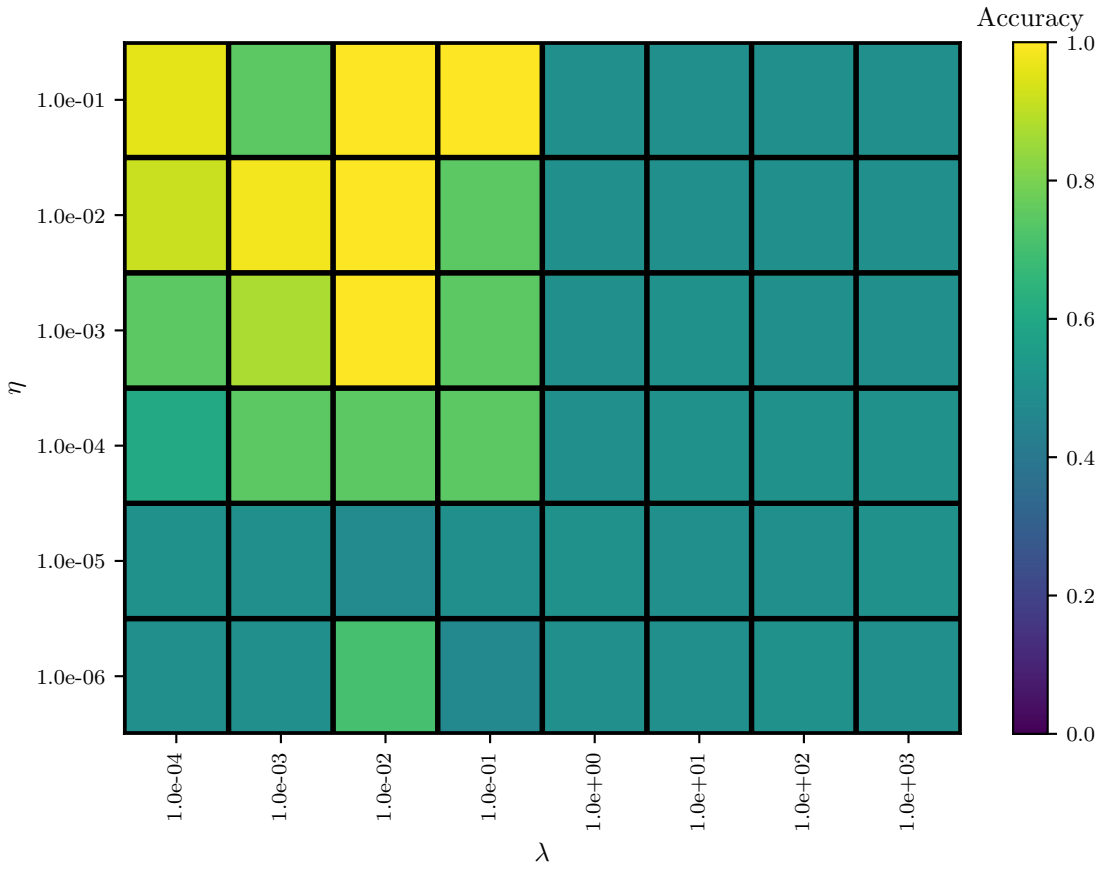


Figure 12: The accuracy as function of the L^2 regularization parameter λ and constant training rate η. The run was for 500 epochs and with cross entropy as cost function, softmax output and sigmoidal hidden layer activation. The hidden layer was 10 neurons large.

A comparison of the accuracy score(38) as a function of L^2 regularization parameter and hidden layer size(the neurons) can be viewed in figure 13.

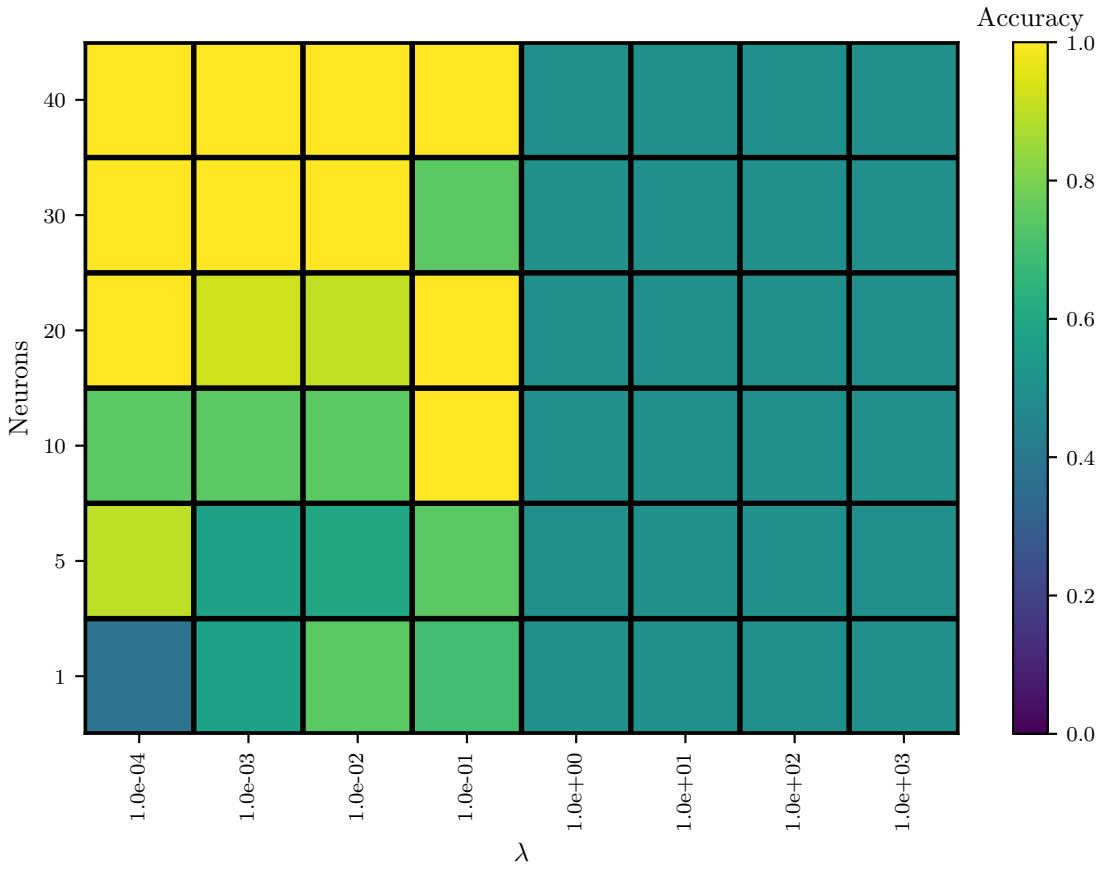


Figure 13: The accuracy score as function of the L^2 regularization parameter λ and the number of neurons. The run was for 500 epochs and with cross entropy as cost function, softmax output and sigmoidal hidden layer activation. The learning rate was set with the inverse learning rate (37) with an $\eta_0 = 0.001$.

The accuracy score(38) as a function of the hidden layer size(the neurons) and the training data size as percentage of of a $N_{\text{samples}} = 10000$ training data, can be viewed in figure 14.

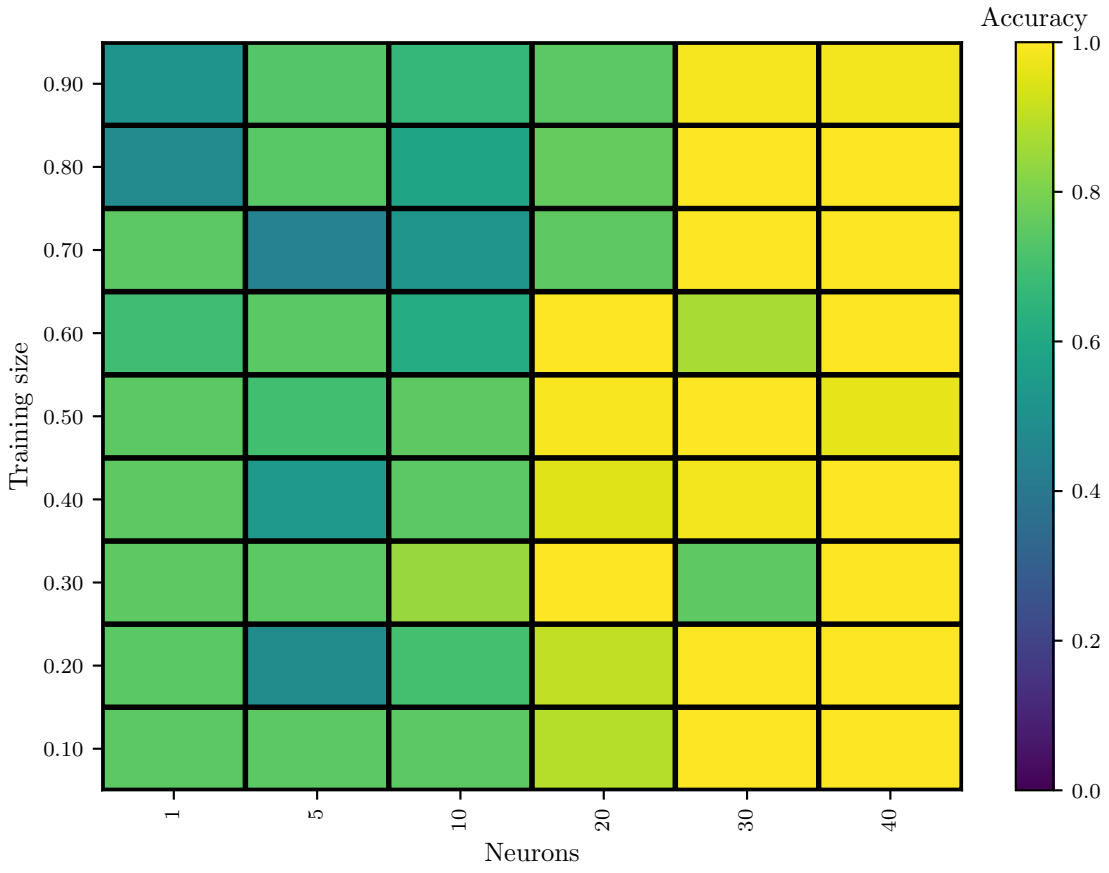


Figure 14: The accuracy score as function of the number of neurons and the training data size percentage of $N_{\text{samples}} = 10000$. The run was for 500 epochs and with cross entropy as cost function, softmax output and sigmoidal hidden layer activation. The learning rate was set with the inverse learning rate (37) with an $\eta_0 = 0.001$ and $\lambda = 0.0$.

The accuracy score(38) as a function of the hidden layer size(the neurons) and the learning rate η, can be viewed in figure 14.

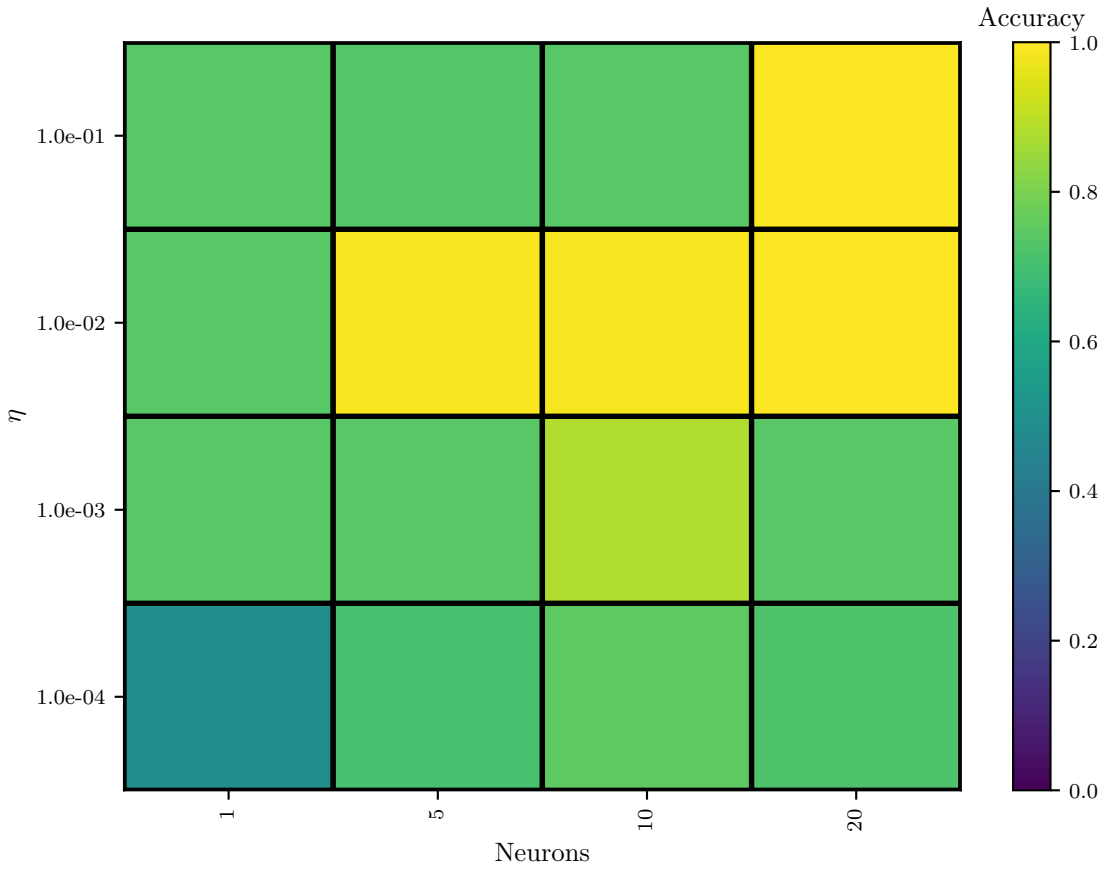


Figure 15: The accuracy score as function of the number of neurons and the learning rate η. The run was for 500 epochs and with cross entropy as cost function, softmax output and sigmoidal hidden layer activation. The regularization strength was set to $\lambda = 0.0$

.

The accuracy score(38) as a function of L^2 regularization strength λ and the mini batch size in the SGD2, can be viewed in figure 16.

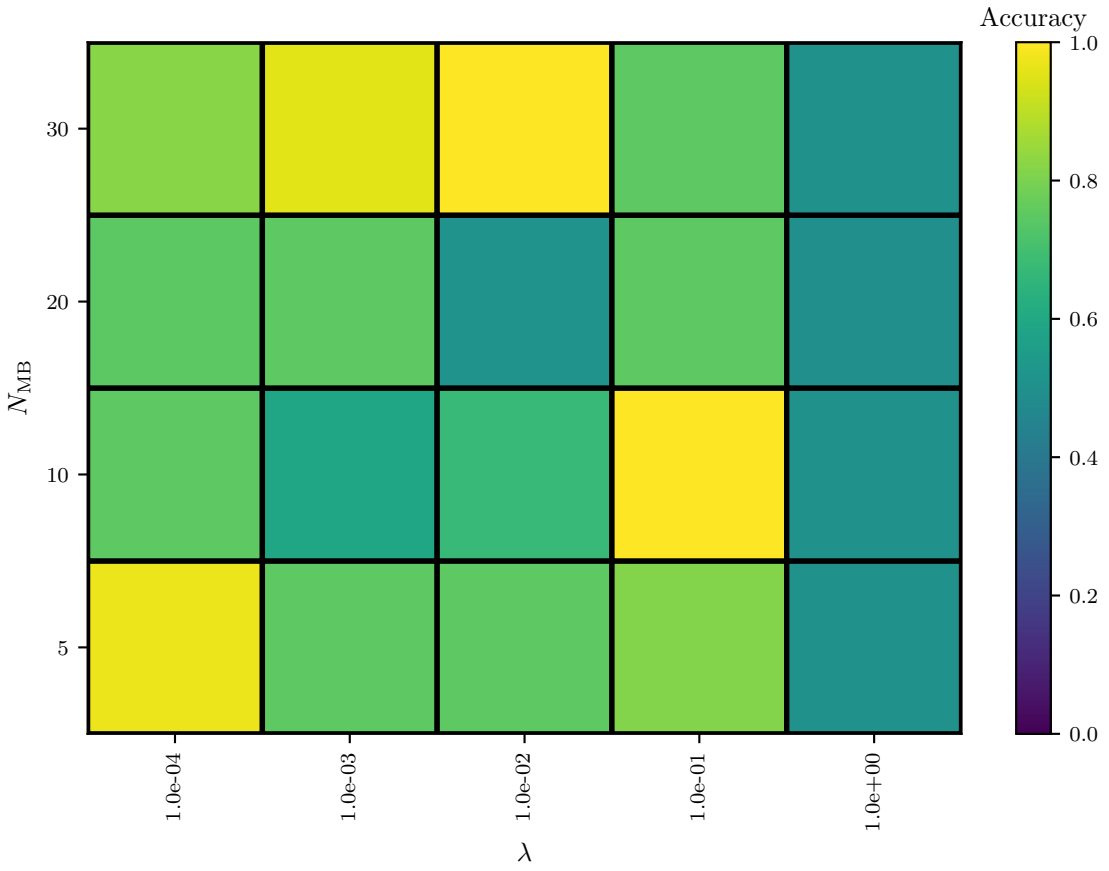


Figure 16: The accuracy score as function of the L^2 regularization strength λ and the mini batch size in the SGD2. The run was for 500 epochs and with cross entropy as cost function, softmax output and sigmoidal hidden layer activation and the inverse learning rate (37) with $\eta_0 = 0.001$.

After choosing following optimal parameters, we get

Parameters	Values
N_{neurons}	10
λ	0.1
N_{mb}	10
N_{epochs}	500
η (constant)	0.001

Table 1: Based on the different fitting procedures, these fit parameters were determined to be the best. The polynomial fitted was of degree 6.

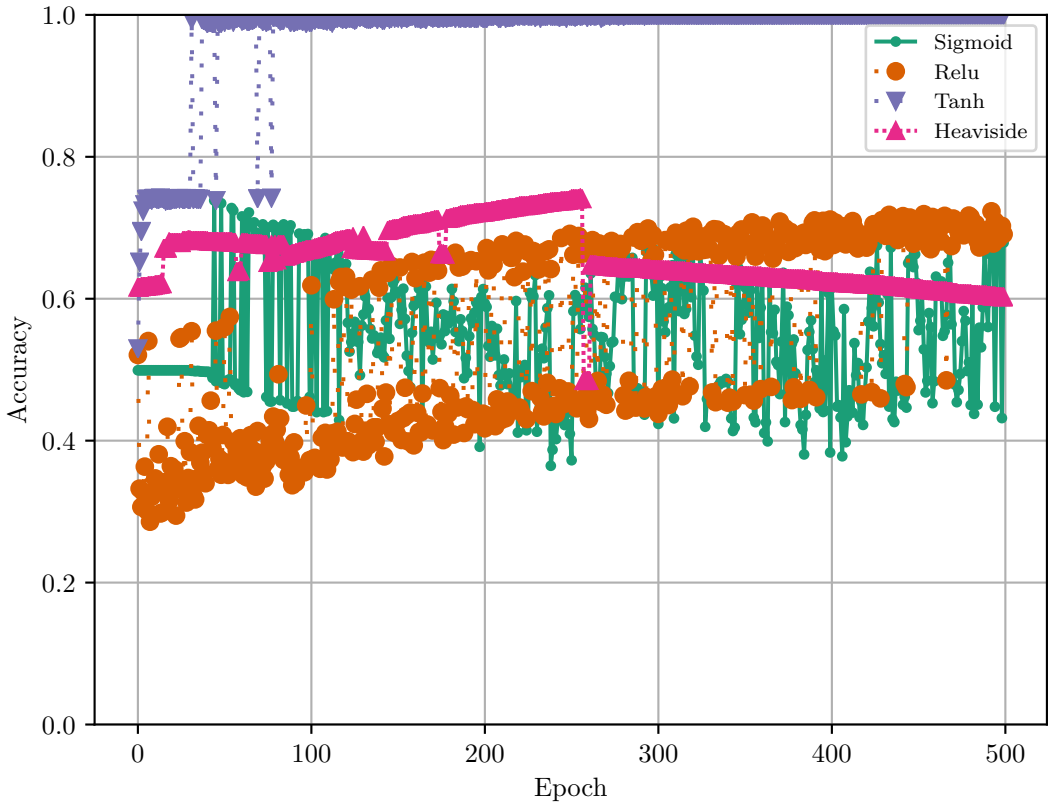


Figure 17: A comparison in accuracy scores between the hidden layer activation functions(see section 2.6.3) for MSE as cost function using *optimal parameters*. The optimal parameters can be viewed in table .

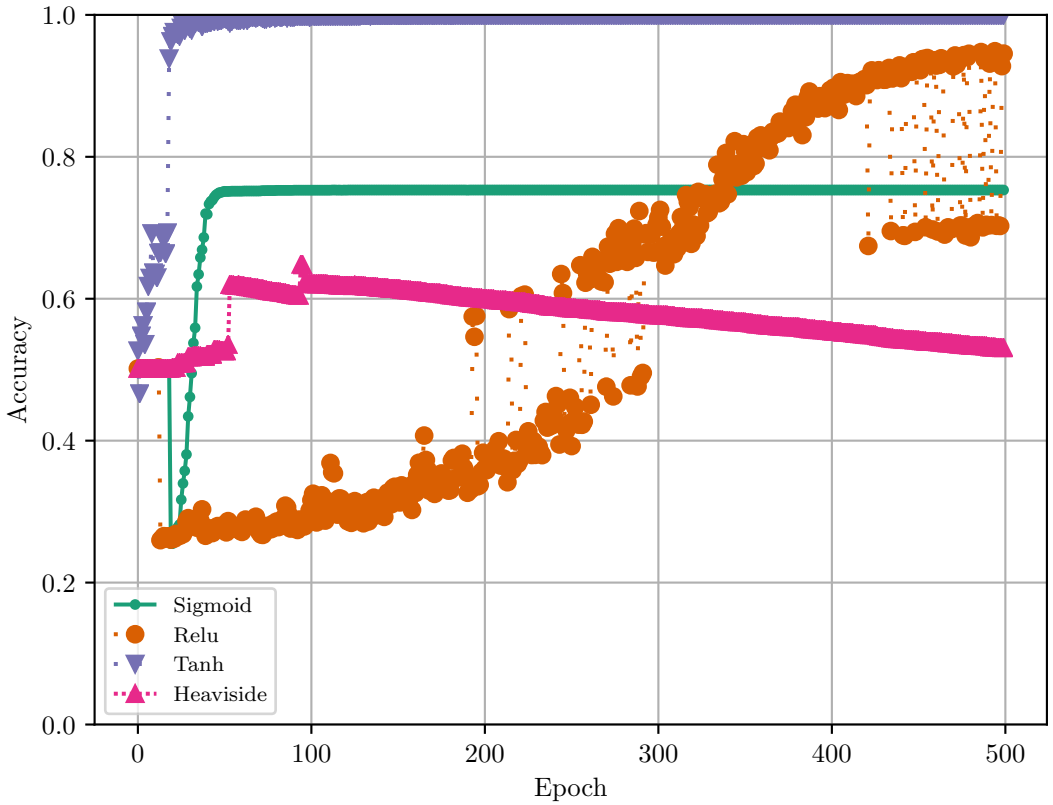


Figure 18: A comparison in accuracy scores between the hidden layer activation functions(see section 2.6.3) for cross entropy as cost function using *optimal parameters*. The optimal parameters can be viewed in table.

5 Discussion

5.1 1D Ising model

5.1.1 Fitting with linear regression

By examining the results from figure 1 illustrates the effect of different λ-values. The diagonal line illustrates the particle we are looking at and the pixels to its immediate left and right are the nearest neighbours. It is clear from this figure that even though we assumed that all particles interact with each other, only the neighbouring particles will have a profound effect on each other. For Lasso, it seems that the optimal λ is either 10^{-3} or 10^{-1}. This is agreeable with the results of Metha et al [6] who found 10^{-2} to be the optimal parameter. For Ridge and OLS we can not seem to see any big difference between the different λ's. The inability to find the best λ for OLS and Ridge and the fact that we do not have a clear optimal parameter for Lasso indicates that we should have gathered more data.

However, by looking at figure 2 we can see that there is little change in the R2-score for

different λ-values, indicating that it would not matter if we generated many more plots. They would most likely stay the same. It is also clear that Lasso is the most sensitive of the models, loosing ground to the other regression methods already at $\lambda = 10^{-1}$. The R2-score for Ridge starts to decline at about $\lambda = 10^3$, while OLS seems to maintain its quality over all λ's.

This trend is further confirmed by figure **??** and 4 where we have used Bootstrap and k-fold validation to verify our results. As we can see, the variance stays about the same for all λ-values, but MSE and therefore, the bias increases. This shift in bias also happens around $\lambda = 10^{-1}$ for Lasso and $\lambda = 10^3$ for Ridge as the R2-score did as well.

5.1.2 Fitting with a neural network

Repeating the calculations using a neural net with zero hidden layers and the identity function as the activation function gives us the opportunity to repeat the above calculations using a neural net instead. As we can see from figure 5, the difference in N_{train} had little effect, but the change in λ was significant. The change between OLS, Ridge and Lasso also represent a change in regularisation from no regularisation to L^2 to L^1 respectively.
In contrast to the linear regression case, there is a difference between $\lambda = 10^{-3}$ and $\lambda = 10^{-1}$. The diagonal lines once removed which signifies the nearest neighbours are about as strong for both parameters, however, there is a noticeable decrease in noise for the $\lambda = 10^{-1}$ case. Furthermore, we must note that the use of the parameter $\lambda = 10^1$ is more devastating for Ridge than Lasso this time around. Lasso looses a lot of information, but for Ridge the significant of nearest neighbours disappears completely. The same trend is once again evident for the R2-score plotted in figure 6. However, with this representation we can see that increasing N_{train} decreases the difference between the test and train data. This is as expected.

5.2 2D Ising model

5.2.1 Classification through logistic regression

From figure 7 we can compare the accuracy of our implemented logistic regression method to that of SKLearn. Furthermore, we can compare our results to those of Metha et al. [6]. What is peculiar in our case is that SKlearn with Stochastic Gradient Descent (SGD) is very varied over the different λ-values, while Metha et al. have quite stable values. Both standard SKlearn and the implemented methods have stable accuracies, with the implemented train being the clear winner. A reason for this unexpected variation in accuracy for SGD might be that we perform the analyses on less data than those of Metha et al. causing our SGD to jump more randomly back and forth for higher λ-values. If the experiment was to be repeated we would double check that the SKlearn with SGD was correctly used and give it more data. Our implemented data have performed better than those of Metha et al., which is positive. However, this might also indicate that we have implemented something wrong and are just lucky. If this experiment were to be repeated we would also double check our implementation and run it over more data to truly test if we are just lucky or if our method is actually better.

5.2.2 Classification through neural networks

To find the optimal solution using the neural net we implemented many different features to our Neural Net (NN). All these results can be viewed in figures 8 - 11. We then moved on to trying to find the optimal parameters in figures 12 - 16.

Our initial results in figure 8 gives us a comparison of using either MSE or LogLoss as a cost function. The latter cost function ca nthen be directly compared to our logistic regression results in figure 7. As we can see, the LogLoss accuracy increases significantly over the first 20-30 epochs before varying until we reach some more than 300 epochs. After this the accuracy is almost 1.0 which seems a bit to good to be true. The MSE stays at around 0.75, which is quite comparable to the results of the logistic regression above.
Comparing the initial weights between the default and large as in figure 9, we see a similar trend. For smaller number of epochs, the accuracy increases quite drastically until it becomes stable a bit before the 100 epochs mark. Once the stable accuracy is reached, the Large initial weights perform best, however, there is not that big of a difference. These results are also comparable to the logistic regression above and seem to be a bit worse than the implemented train, but better than the implemented test.

Furthermore, figures 10 and 11 indicates that changing the activation function will have an effect on the accuracy as well. This is to be expected. The Heaviside function was added for historical reasons and even though it is the loser when both cost functions are considered, it is clear why this function was the initial function which was used at the advent of Neural Nets and made scientists want more. In our modern times the other functions are more used and Relu is the clear accuracy winner for MSE and the Sigmoid and Relu are so close that they are almost indistinguishable for cross entropy. This indicates that Relu should be the optimal activation function, regardless of cost function.
No Neural Net is only made up of the activation and cost function, we also need to do an in-depth analyses of the different variables. From figure 12 it seems as though a smaller λ and larger is η is generally better until we reach a limit. From this figure it seems as though a learning rate of $\eta = 1.0e-02$ and $\lambda = 1.0e-02$ gives us the optimal accuracy.
However, there are more concerns to take into account. Figure 13 seems to point towards a decrease in λ and an increase in neurons is almost always better. It is also clear that even though it seems to increase, it also seems to reach a plateau where further increases in neurons and decreases in λ may have an effect. As this will also increase computational time, it seems as though $\lambda = 1.0e-02$ and 30 neurons gives us the optimal balance between high accuracy and decreased computational time.
Similarly, figure 14 can be interpreted as training sets around 0.5 and large number of neurons gives us the optimal accuracy. It also seems as though the number of neurons is the most significant parameter of these two as all neuron columns seem to have quite similar accuracy, independent of training size, compared to the difference in accuracy of the training size rows. In this case it seems as though a training size of 0.5 with 20 or 30 neurons would be optimal.
Given the changes in learning rate compared to number of neurons displayed in figure 15, we see similar tendencies. An increase in neurons and an increase in learning rate is positive. However, the effect of increasing learning rate decreases once we pass $\eta = 1.0e-02$. The optimal learning rate is therefore $\eta = 1.0e-02$, but the accuracy seems to be quite similar for number of neurons from five and above.

Finally, 16 gives us quite unambiguous results. As $\lambda = 1.0e-02$ has been dominating the previous results, it is tempting to see the same here as well. And there are indications that a batch size of 30 and $\lambda = 1.0e-02$ are the optimal values, however, there are no trends significant enough to say this with certainty.

6 Conclusion

References

[1] Jonathan Barzilai and Jonathan M. Borwein. Two-point step size gradient methods. *IMA Journal of Numerical Analysis*, 8(1):141–148, 1988. doi: 10.1093/imanum/8.1.141. URL `http://dx.doi.org/10.1093/imanum/8.1.141`.

[2] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.", 2017.

[3] Eirik Ramsli Hauge and Joakim Kalsnes. Fysstk4155: Project 1 github repository. https://github.com/eirikraha/fysstk3155—Project-1, October 2018.

[4] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[5] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943. ISSN 1522-9602. doi: 10.1007/BF02478259. URL `https://doi.org/10.1007/BF02478259`.

[6] P. Mehta, M. Bukov, C.-H. Wang, A. G. R. Day, C. Richardson, C. K. Fisher, and D. J. Schwab. A high-bias, low-variance introduction to Machine Learning for physicists. *ArXiv e-prints*, March 2018.

[7] Magnus Nielsen. How the backpropagation algorithm works. http://neuralnetworksanddeeplearning.com/chap2.html, October 2018.

[8] Lars Onsager. Crystal statistics. i. a two-dimensional model with an order-disorder transition. *Physical Review*, 65(3-4):117, 1944.

[9] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[10] Mathias Vege. Fysstk4155: Project 2 github repository. https://github.com/hmvege/FYSSTK4155-Project2, October 2018.

[11] Mathias Vege. Fysstk4155: Project 2 github repository. https://github.com/hmvege/FYSSTK4155-Project1, October 2018.