Using Neural Networks, Linear and Logistic Regression to Mimic the Ising Model

Camilla Marie Baastad Niels Bonten

December 17, 2018

We compare MLP neural networks ability to perform regression analysis and classification against that of linear regression and logistic regression respectively. The test of our methods is the 1D Ising model in the regression case, and identifying phases of spin configurations. We find that Lasso regression outperforms MLP with an R^2 score of 0.9790 on test data using 3-fold cross-validation on 600 samples. In comparison, the MLP scored only 0.8400 using 10000 samples. In the classification case, however, the MLP excelled. It obtained an accuracy of 99.86 % against logistic regression's score of 85.86 %. The sigmoid proved to be the best activation function in terms of accuracy 99.86 %, but a comparable accuracy 99.39% we considered a clipped ReLu function that was six times faster to compute.

All code and data used in this project can be found at: `https://github.com/Niebon/FYS-STK4155_Project_2`.

Contents

Introduction

Statistical classification can be used for a lot of things, from sending shady e-mails to your spam filter, to assigning diagnoses. The multi-layer perceptron (MLP) is an interesting method that can be trained to behave as it is taught. It can do classification problems and can to a successful degree mimic other tools. We consider the Ising model for energies of spin configurations and mimic its results with various regression methods. Given a spin configuration s, the Ising model defines the energy of s as a certain linear expression.

There are two main objectives to this report. The first is to perform linear regression and MLP regression to mimic Ising energies for spin configurations. For the linear regression part, we assume a much more general basis and perform linear regression with respect to that basis, to see if we can still estimate the output of the Ising model. For this we further develop the code that we used in our previous project. [BB18] For the MLP part, we create a multi-layer perceptron class with a back-propagation algorithm using steepest descent. We compare the two methods with respect to R^2-scores, MSE, variance and bias using 3-fold cross validation. We find that the Lasso model performs excellent at 600 data points, whereas the MLP needs up to 10000 point to perform well on testing data.

The second objective is to classify the phase of spin configurations into either ordered or disordered using the 2-dimensional Ising model. The physics of the situation is that the phase depends on the temperature associated to the spin configuration, which again depends on the energy. We consider data consisting of spin configurations labeled either ordered or disordered and train logistic models and MLP's to guess correctly. We use stochastic gradient descent to train our models for batch sizes $\{1, 2, 4, 8, 16, 32, 64\}$. The MLP's perform very well in this task, even better than logistic regression. For our MLP in the classification problem we also experiment with different activation functions. We find that a clipped ReLu is a good alternative to a sigmoid.

In section 1 we present the Ising model. We describe logistic regression and MLP with backpropagation, and provide a brief presentation of the linear regression methods OLS, Ridge and Lasso. We also present stochastic gradient descent and the functions we will use to evaluate the error of our models. Section 2 contains a description of the two main classes in our programs as well as the most important functions. In section 3.1 we present the results we got when performing regression analysis using the 1-dimensional

Ising model. The results from our attempt to classify the phases of the 2-dimensional Ising model can be found in section 3.2.

1 Theory

In this section we present the Ising model, which will be used throughout this project. We give a brief explanation of linear regression, before presenting logistic regression and associated gradient methods. We proceed by presenting artificial neural networks using the multi-layer perceptron architecture and how to train the network using backpropagation. Finally we present methods for analysing the error of our model, both in the regression and the classification case.

This section rests heavily on the texts "Logistic regression", "Optimization and Gradient Methods" and "Neural networks, from the simple perceptron to deep learning and convolutional networks" by Morten Hjorth-Jensen, and the article "A high-bias, low-variance introduction to machine learning for physicists" by Mehta et al. "Optimization and Gradient Methods". [Hjo18b; Hjo18a; Hjo18c; Meh+18]

1.1 The Ising model

The Ising model is a model of ferromagnetism in statistical mechanics. The ferromagnetic metal is in some lattice state, and the magnetic moments of the electrons spin is in one of two states, +1 or -1. The interaction energy between neighbouring spins give rise to the Hamiltonian of the system.[1]

Definition 1.1. (Ising model) Let $\Lambda \subset \mathbb{Z}^n$ be a subset and let J_{xy} be physical constants with unit energy for $x, y \in \Lambda$. A **spin configuration** on Λ is a function $s \colon \Lambda \longrightarrow \{1, -1\}$. The **energy** $E[s]$ **of the spin configuration** s is defined as

$$E[s] := -\sum_{\substack{\langle x,y \rangle \\ x,y \in \Lambda}} J_{xy} s(x) s(y).$$

Here $\langle xy \rangle$ means that the sum goes over closest neighbors in Λ, i.e., those points which lie at most a distance 1 apart.

We will only consider the above version of the Ising model. However, if there is an external magnetic field present, then the Ising model may be be modified by adding an additional term for each $x \in \Lambda$.

Example 1.2. (1-dimensional Ising model) Letting $\Lambda = \mathbb{Z}$ we have the 1-dimensional Ising model. If we further let $J_{kl} = 1$ for all $k, l \in \mathbb{Z}$, then the energy for a spin configuration s simply becomes $E[s] = -\sum_{\langle kl \rangle} s(k) s(l)$.

Our goal in the first part of this project is to learn models to predicts the energies from the spin configurations using the 1-dimensional Ising model.

The two-dimensional Ising model exhibits a phase transition at some critical temperature. Below critical temperature, the system will be in a ferromagnetic phase. The configurations representing the sub-critical temperature states are called *ordered*. Above critical temperature, net magnetization is zero. These configurations are called *disordered*.

The second part of the project is a classification exercise, in which we want to, given a spin configuration, identify its phase using the 2-dimensional Ising model.

1.2 Linear regression

Consider a data set $\{y_i, \boldsymbol{x}^i = (x_1^i, x_2^i, ..., x_p^i)\}_{i=1}^n$ measured in p independent variables. Let f be the evaluation map of the vector $(x_1^i, x_2^i, ..., x_p^i)$ in the given basis $\mathcal{B}$. Then

$$\boldsymbol{X} = \begin{bmatrix} f(\boldsymbol{x}^1) \\ f(\boldsymbol{x}^2) \\ \vdots \\ f(\boldsymbol{x}^n) \end{bmatrix}$$

is the design matrix.

Linear regression assumes a linear relationship between the dependent and the independent variables. We will make use of Ordinary least squares (OLS), Ridge and Lasso regression.

OLS describes the relationship

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon},$$

and so our *cost function*, which we will try to minimize to find the best regression parameters, $\boldsymbol{\beta}$, becomes

$$C_{OLS}(\beta) = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}).$$

Ridge and Lasso are so called *regularization methods*, where we add a regularizer, λ, that shrinks the regression parameters. Ridge regression uses the L_2 norm of the regression parameter. The cost function becomes

$$C_R(\beta) = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta}.$$

Lasso regression uses the L_1 norm, and so the cost function becomes

$$C_L(\beta) = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) + \lambda|\boldsymbol{\beta}|.$$

For a complete description of the three linear regression methods, see our previous report on the subject. Here you will also find information about k-fold cross-validation, a resampling technique used in this project. [BB18]

1.3 Logistic regression

Consider again a data set $D = \{y_i, \boldsymbol{x}^i = (x_1^i, x_2^i, ..., x_p^i)\}_{i=1}^n$ measured in p independent variables, but let y_i be discrete and take only values $k = 0, 1, ..., K-1$ (K different classes). We want to be able to predict the output classes of a given design matrix, $\boldsymbol{X} \in \mathbb{R}^{n \times p}$, and ultimately obtain a model that is able to predict the output classes of unseen data. In this project we will only consider binary classification, in which there are two output classes.

In classification problems we model the *probability* that the data belongs to the class k, based on the input $\boldsymbol{x}^i$. In logistic regression, our probability function is given by the **sigmoid**,

$$p(t) = \frac{1}{1 + e^{-t}}.$$

We will return to the sigmoid function when discussing activation function for neural networks, in section 1.4.1.

Denote by $p(y_i = 1|\boldsymbol{x}^i; \boldsymbol{\beta})$ the probability of y_i being class 1 given the input vector $\boldsymbol{x}_i$ and weights $\boldsymbol{\beta}$. Using the sigmoid function, this probability is given as

$$p(y_i = 1|\boldsymbol{x}^i; \boldsymbol{\beta}) = \frac{1}{1 + e^{-\boldsymbol{x}^i \boldsymbol{\beta}}}.$$

As we consider binary classification only, the probability that y_i is the other class, 0, is given by

$$p(y_i = 0|\boldsymbol{x}^i; \boldsymbol{\beta}) = 1 - p(y_i = 1|\boldsymbol{x}^i; \boldsymbol{\beta}).$$

To define the cost function for logistic regression, we use the *Maximum Likelihood Estimation,* given by

$$P(D|\boldsymbol{\beta}) = \prod_{i=1}^{n} p(y_i = 1)^{y_i} (1 - p(y_i = 1))^{1-y_i}.$$

From this we can find the log-likelihood

$$\log(P(D|\boldsymbol{\beta})) = \sum_{i=1}^{n} y_i \log(p(y_i = 1))(1 - y_i) \log((1 - p(y_i = 1))).$$

We use the negative log-likelihood for a cost function, the so called *cross entropy cost function.*

Definition 1.3. (Cross entropy cost function) Given data $\mathcal{D} = \{y_i, x_i\}$ where $y_i \in \{0, 1\}$ depends on x_i, and the design matrix

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{X}_1^T \\ \boldsymbol{X}_2^T \\ \vdots \\ \boldsymbol{X}_n^T \end{bmatrix}$$

for a linear regression problem for $\mathcal{D}$, the **cost function** that solves the corresponding logistic regression problem is:

$$C(\boldsymbol{\beta}) = -\sum_{i=1}^{n} (y_i \boldsymbol{X}_i^T \boldsymbol{\beta} - \log(1 + \exp \boldsymbol{X}_i^T \boldsymbol{\beta})$$

1.3.1 Gradient methods

Many machine learning problems are rather similar. We start off with a data set, a model and a cost function. The cost function is meant to represent the error of our model, so that minimizing it will result in the regression parameters, $\boldsymbol{\beta}$, that yield the best model. In certain cases, like OLS and Ridge, we can solve for $\boldsymbol{\beta}$ analytically. In most cases, however, we depend on some computational method to compute the minimum.

Gradient descent is based on the fact that a function $C(\beta)$ decreases fastest from β in the direction of the negative gradient of C, $-\nabla C(\beta)$. Starting with an initial guess of the point β_0, we can compute the new approximations inductively by

$$\beta_{k+1} = \beta_k - \gamma_k \nabla C(\beta_k).$$

Here γ_k is often referred to as the **learning rate**.

The gradient descent method has some limitations. For instance, if our function is not convex, we run a high risk of getting stuck in local minima. Also, the method is very sensitive to choice of learning rate. If the learning rate is too small, our method will take too long to converge. Too large a learning rate will cause irregular behavior. We can however, repress these weaknesses to some degree, by introducing stochasticity. This leads us to *stochastic gradient descent.*

The cost function, which we want to minimize, can almost always be written as a sum over n datapoints $\{\mathbf{x}_i\}_{i=1}^n$ in the following way,

$$C(\beta) = \sum_{i=1}^{n} c_i(\mathbf{x}_i, \beta). \tag{1}$$

This means that the gradient can be computed as a sum over i-gradients

$$\nabla_\beta C(\beta) = \sum_{i}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta). \tag{2}$$

Stochasticity occurs by only taking the gradient on a subset of the data. We split the data into *mini-batches.* For a data set with n points, if we have M points in each mini-batch, we have n/M mini-batches, B_1, B_2, ... , $B_{n/M}$.

We now approximate the gradient by, instead of summing over all data points, we sum over the datapoints within one minibatch picked at random in each gradient descent step,

$$\nabla_\beta C(\beta) = \sum_{i \in B_k}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta). \tag{3}$$

The stochastic gradient descent step is thus given by

$$\beta_{k+1} = \beta_k - \gamma_k \sum_{i \in B_k}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta). \tag{4}$$

1.4 Neural networks

Artificial neural networks are computational structures that are able to learn from examples shown. There are several ways to build an artificial neural network. We will be using *multi-layer perceptron.* Such a neural network defines a complicated map from $\mathbb{R}^n$ to $\mathbb{R}^m$.

Definition 1.4. Let $f\colon \mathbb{R} \to \mathbb{R}$ be any function. The function f defines for all $k > 1$ the function $f\colon \mathbb{R}^k \to \mathbb{R}^k$ by

$$f(\mathbf{z}) = (f(z_1), \ldots, f(z_k)), \quad \mathbf{z} = (z_1, \ldots, z_k).$$

An **MLP** (multi-layer perceptron) $N_m^n(\psi)$ is a function $\mathbb{R}^n \longrightarrow \mathbb{R}^m$ which is defined as a composition of the form:

$$\mathbb{R}^n \xrightarrow{T_1} \mathbb{R}^{n_1} \xrightarrow{f} \mathbb{R}^{n_1} \xrightarrow{T_2} \mathbb{R}^{n_2} \xrightarrow{f} \mathbb{R}^{n_2} \xrightarrow{T_3} \cdots \xrightarrow{f} \mathbb{R}^{n_s} \xrightarrow{T_s} \mathbb{R}^{n_s} \xrightarrow{f} \mathbb{R}^m$$

such that $T_i(\boldsymbol{a}) = \boldsymbol{W}_i\boldsymbol{a} + \boldsymbol{b}_i$ is a linear transformation for all $i \in \{1, \ldots, s\}$ with $s \geq 1$. The entries of the matrix $\boldsymbol{W}$ are referred to as **weights** and the entries of $\boldsymbol{b}$ referred to as **bias**. The function f is called an **activation function**.

We may view an MLP as a sequence of functions:

$$\mathbb{R}^n \xrightarrow{f\circ T_1} X_1 \xrightarrow{f\circ T_2} X_2 \xrightarrow{f\circ T_3} \cdots \xrightarrow{f\circ T_s} \mathbb{R}^m$$

We then refer to the X_i as the **hidden layers** of the given MLP, and to $\mathbb{R}^n$ and $\mathbb{R}^m$ as respectively the **input layer** and **output layer**. Each copy of $\mathbb{R}$ in the Cartesian product $X_i = \mathbb{R} \times \cdots \times \mathbb{R}$ is called a **node**. The reason for this terminology becomes clear with fig. 1, which illustrates the intuition behind this construction.

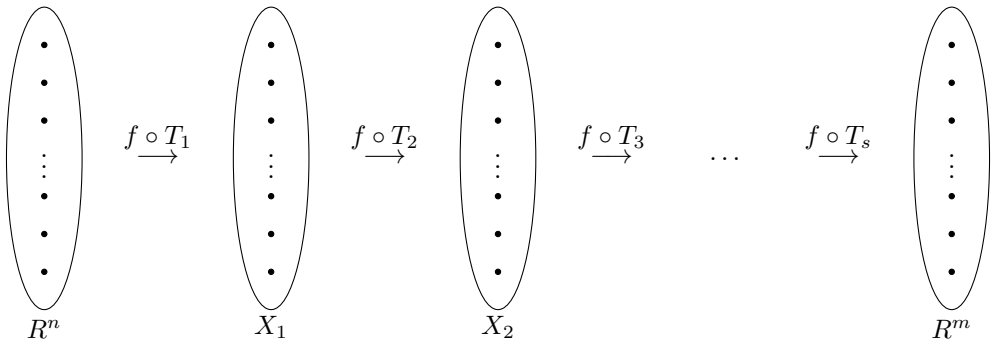

Figure 1: Illustration of MLP. Each layer defines maps to the nodes in the next layer. It is common to illustrate this by connecting the nodes by lines.

We note that the structure of an MLP consists of at least one layer of input nodes, one layer of output nodes, and at least one hidden layer in between the input and output layer. The input layer passes a signal on to the first hidden layer, which passes a signal on to the next hidden layer and so on, until reaching the output layer.

Definition 1.5. Let $s \geq 0$ be an integer. For a neural network N with n inputs, and m outputs, and successive hidden layers $l_1, \ldots, l_s$ with respectively $n_1, \ldots, n_s$ nodes, we define the **size** of N to be the ordered sequence of integers:

$$[n, n_1, \ldots, n_s, m]$$

Example 1.6. (i) The size of a network with 10 inputs, 5 outputs and one hidden layer with 4 nodes is $[10, 4, 5]$.

(ii) Let N be a neural network such that the size of N is $[10, 5, 8, 2]$ and with tanh as activation function. Then the network N determines a map $[0,1]^{10} \longrightarrow [0,1]^2$ that admits a factorization:

$$[0,1]^{10} \longrightarrow [0,1]^5 \longrightarrow [0,1]^8 \longrightarrow [0,1]^2$$

Neural networks are made to mimic the way biological neurons interact. A neuron receives input from other neurons, and is activated if this input exceeds some threshold value. An activated neuron emits an output that may activate other neurons.

Each neuron, or node, receives a vector of input values $\boldsymbol{a}$, which is determined by the output of the previous layer. Consider the node n_j^l in layer l. The input this node receives from node n_i^{l-1} in the previous layer $l-1$, is determined by the output a_i^{l-1}, of this node. The output is weighted by a weight w_{ji}^l, corresponding to the strength of the connection between these to nodes. Say layer $l-1$ consists of N nodes. Each of these nodes then contributes to the input node n_j^l receives.

Definition 1.7. (Activation) The **activation**, z_j^l, of node n_j^l in layer l is given by

$$z_j^l = \sum_{i=1}^{N} w_{ji}^l a_i^{l-1} + b_j^l,$$

where $w_{ji}^l a_i^{l-1}$ is the weighted output of the previous layer consisting of N nodes, and b_j^l is the bias.

The output of node n_j^l, a_j^l, is given by an activation function f. That is, $a_j^l = f(z_j^l)$. Activation functions are described in section 1.4.1.

Denote by $\boldsymbol{W}^l$ the matrix consisting of all weights associated with layer l, such that $(\boldsymbol{W}^l)_{ij} = w_{ij}^l$. Let $\boldsymbol{a}^{l-1}$ be the vector consisting of all outputs from the previous layer and let $\boldsymbol{b}^l$ be the vector consisting of the biases in layer l. The propagation of the signal from layer $l-1$ to layer l can thus be written as

$$\boldsymbol{a}^l = f(\boldsymbol{W}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l).$$

1.4.1 Activation functions

In [Hor91] Hornik shows that with even a single hidden layer, neural networks approximate all functions $f\colon X \to \mathbb{R}$ where $X \subset \mathbb{R}^k$ is compact (closed and bounded) arbitrary well under certain conditions on the activation function. The exact conditions are that the activation functions are *continuous, bounded* and *non-constant*. This opens up an wide array of activation function that one might test. Of course, for any given problem, we seek one that performs both well and is not too numerically expensive.

We consider the following activation functions of the form $\mathbb{R} \to [0, 1]$ and note that all are continuous, bounded and non-constant:

(i) Sigmoid: $f(z) = \frac{1}{1+e^{-z}}$

(ii) Tanh: $f(z) = \tanh z$

(iii) Clipped rectified linear unit (ReLu)

$$f(z) = \begin{cases} 0.9999, & \text{if } z > 0.9999, \\ 0.001, & \text{if } z < 0.001, \\ z, & \text{otherwise.} \end{cases}$$

(iv) Bump:

$$f(z) = \begin{cases} 0.9999, & \text{if } z > 0.9999, \\ 0.001, & \text{if } z < 0.001, \\ \sin \frac{\pi z}{2}, & \text{otherwise.} \end{cases}$$

(v) Hill:

$$f(z) = \begin{cases} 0.9999, & \text{if } z > 0.999,9 \\ 0.001, & \text{if } z < 0.001, \\ 1 + \sin \frac{3\pi z}{2}, & \text{otherwise.} \end{cases}$$

The sigmoid activation function will be used unless stated otherwise. Note that (iii), (iv) and (v) are not smooth.

We are motivated to only consider activation functions $f(z)$ such that $f'(z) \neq 0, 1$. The reason for this is that we will consider the derivative of the cross entropy cost function in the backwards propagation algorithm. Since it goes as

$$C'(y) = \frac{y - t}{y(1 - y)}$$

for the output y of the network and target t, we conclude that y may not take the values 0 and 1. This motivates in the definition for the mysterious clipped rectified linear unit and also for the functions labeled Bump and Hill. We require that $f'(y) \subset (0, 1)$, and model $(0, 1)$ numerically by $[0.001, 0.9999]$.

1.4.2 Backpropagation

We *train* the network by feeding it examples. Using a set of input and known, correct output, *targets*, we can train our network through supervised learning. A common way to do this is using *backpropagation*.

The idea is simple. We feed the input to our network and let it propagate through the layers until we reach the final output. We compare our estimation to the targets by evaluating some cost function. Computing the gradient of the cost function with respect to our parameters, i.e. weights and biases, indicates how we can adjust said parameters to yield a better model. We then adjust the parameters and repeat until satisfied or until we run out of data.

After feeding our network and evaluating the cost function $C(W^l)$, we need to find the derivatives of $C(W^l)$ with respect to the weights and biases in the output layer, L. We will find these through repeated utilization of the chain rule.

$$\begin{aligned}
\frac{\partial C(W^L)}{\partial w_{ij}^L} &= \frac{\partial C(W^L)}{\partial a_i^L} \frac{\partial a_i^L}{\partial w_{ij}^L} \\
&= \frac{\partial C(W^L)}{\partial a_i^L} \frac{\partial a_i^L}{\partial z_i^L} \frac{\partial z_i^L}{\partial w_{ij}^L} \\
&= \frac{\partial C(W^L)}{\partial a_i^L} f'(z_i^L) a_i^{L-1}.
\end{aligned}$$

Likewise,

$$\begin{aligned}
\frac{\partial C(W^L)}{\partial b_i^L} &= \frac{\partial C(W^L)}{\partial a_i^L} \frac{\partial a_i^L}{\partial b_i^L} \\
&= \frac{\partial C(W^L)}{\partial a_i^L} \frac{\partial a_i^L}{\partial z_i^L} \frac{\partial z_i^L}{\partial b_i^L} \\
&= \frac{\partial C(W^L)}{\partial a_i^L} f'(z_i^L).
\end{aligned}$$

Defining

$$\delta_i^L := \frac{\partial C(W^L)}{\partial a_i^L} f'(z_i^L),$$

we see that we can write $\frac{\partial C(W^L)}{\partial w_{ij}^L} = \delta_i^L a_i^{L-1}$ and $\frac{\partial C(W^L)}{\partial b_i^L} = \delta_i^L$. The factor $\frac{\partial C(W^L)}{\partial a_i^L}$ measures the rate of the change of the cost function with the i-th output node. Thus if the cost function does not depend much on the i-th output node, δ_i^L will be small.

Having found the gradient of the cost function with respect to the weights and biases of the output layer, we wish to propagate backward through all the layers. The equation

$$\delta_i^l = \frac{\partial C(W)}{\partial z_i^l}$$

holds for a general layer l. We want to express this in terms of δ_i^{l+1}. Using the chain rule and summing over all nodes in layer l, we obtain

$$\delta_i^l = \sum_{k=1}^{n} \frac{\partial C(W)}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_i^l} = \sum_{k=1}^{n} \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_i^l} = \sum_{k=1}^{n} \delta_k^{l+1} w_{ki}^l f'(z_k^l).$$

Summing up, the backpropagation algorithm goes as follows:

(i) Feed the network and let the data propagate through the network.

(ii) Evaluate the output by calculating all $\delta_i^L = \frac{\partial C(W^L)}{\partial a_i^L} f'(z_i^L)$.

(iii) Propagate backwards through the network using the equation $\delta_i^l = \sum_{k=1}^{n} \delta_k^{l+1} w_{ki}^l f'(z_k^l)$.

(iv) Finally, we update the weights and biases using stochastic gradient descent, as presented in section 1.3.1. Let

$$w_{ki}^l \leftarrow w_{ki}^l - \eta \delta_i^l a_k^{l-1},$$

$$b_i^l \leftarrow b_i^l - \eta \delta_i^l,$$

where η is the learning rate.

When doing regression analysis on the 1-dimensional Ising model, we will use the cost function

$$C = \frac{1}{2} \sum_{i=1}^{n} (y_i - t_i)^2,$$

where y_i are the estimates of the targets t_i. In this case,

$$\frac{\partial C}{\partial a_i^L} = a_i^L - t_i.$$

Using neural networks to classify the phases of the 2-dimensional Ising model, we will use the cross-entropy cost function described in **??**. For this cost function,

$$\frac{\partial C}{\partial a_i^L} = \frac{a_i^L - t_i}{a_i^L (1 - a_i^L)}.$$

1.5 Error analysis

When doing regression analysis, we will consider two common methods to estimate the error, namely *Mean Square Error* (MSE) and *the R^2 score function*. In the classification case, we will use the *Accuracy score* to evaluate our results.

1.5.1 Mean Square Error (MSE) and the R^2 score function

Let $\hat{y}_i$ is the estimate of the observation y_i, and let $\overline{\hat{y}} = \frac{1}{n}\sum_{i=1}^{n}\hat{y}_i$ be the mean of all the estimated values $\hat{y}_i$. The Mean Square Error is given by

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2,$$

which can be rewritten as

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \overline{\hat{y}})^2 + \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - \overline{\hat{y}})^2 - \frac{2}{n}\sum_{i=1}^{n}(y_i - \overline{\hat{y}})(\hat{y}_i - \overline{\hat{y}}).$$

Here the first term is what is commonly referred to as the $\text{Bias}^2(\hat{y})$, while the second term is $\text{Var}(\hat{y})$. The R^2 score function is given by

$$\text{R}^2(\hat{y}) = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \overline{\hat{y}})^2}.$$

The best possible score is $R^2(\hat{y}) = 1$, which is when our predictions equals the observed results.

For a complete description of the MSE and the R^2 score function, see our previous project. [BB18]

1.5.2 Accuracy score

Let $\boldsymbol{t} = (t_1, t_2, ..., t_n)$ be our targets and $\boldsymbol{y} = (y_1, y_2, ..., y_n)$ our estimates. The accuracy score of our model is thus given by

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

where

$$I(t_i, y_i) = \begin{cases} 1 & \text{if } t_i = y_i \\ 0 & \text{otherwise.} \end{cases}$$

A perfect classifier will have an accuracy score of 1.

2 Methods

This section contains a description of the two classes that makes up most of our code.

2.1 The class regdata

From project 1 ([BB18]) the class `linreg` needed modification so that it could perform regression with arbitrary bases, (not only those in two variables up to a certain degree). For this we give as functional argument a map from the independent variables to the basis (predictors).

Let $y_1, \ldots, y_n$ be independent variables and let $\boldsymbol{z} = f(y_1, \ldots, y_n)$ be a variable dependent on $y_1, \ldots, y_n$. We may write $\boldsymbol{z} = f(\boldsymbol{y})$ if $\boldsymbol{y} = (y_1, \ldots, y_n)$. Assume we have data of the above type stored in two arrays `Z` and `Y`, corresponding to the depended variables $\boldsymbol{z}$ and independent variables $\boldsymbol{y}$ respectively, so that `Z[i]` belongs to `Y[i]`.

An instance of the class `regdata` for the above data contains `Z` and `Y`, the design matrix `X` for a choice of basis, and functions to perform ordinary least square, Ridge and Lasso fits on the given data resulting in a coefficient `beta` so that `Z` is approximated by matrix multiplying `X` by `beta`. That is

$$\boldsymbol{z}_i \approx \boldsymbol{x}_i^T \boldsymbol{\beta} + \boldsymbol{\epsilon}.$$

Let `f` be a function that transforms `Y[i]` to a row `X[i]` in the design matrix. In other words, `f` represents the map that sends the set of independent variables $\{y_1, \ldots, y_n\}$ to the the basis $\mathcal{B}$ of choice. To initiate an instance of the type regdata for the above regression problem, we write `regdata(Y,Z,f)`.

Example 2.1. If our data is $\mathcal{D} = \{(z_i, (y_1, y_2)_i) | i = 1, \ldots n\}$ and our basis is $\mathcal{B} = \{1, y_1^2, y_1 y_2, 5y^5\}$, then the map f is defined by:

$$(y_1, y_2) \mapsto (1, y_1^2, y_1 y_2, 5y^5)$$

Then also $(1, y_1^2, y_1 y_2, 5y^5)_i$ is the i'th row vector in the design matrix corresponding to the pair $(y_1, y_2)_i$.

A new function `log_beta_SGD(beta_init=None, epoch_start = 0,epochs = 100, M=5)` was added. This guy performs stochastic gradient descent to minimize the cost function for logistic regression. There is an optional input argument `beta_init` which allows to begin the algorithm with a user specified `beta`. Then there is control over what epochs the algorithm runs over (affects learning rate), and the batch size `M`.

2.2 The class MLP

We make a class `MLP` to account for all MLP's of arbitrary size $[n, n_1, n_2, \ldots, n_s, m]$ and with arbitrary activation function f. To initiate an instance of `MLP`, we write `MLP(network_size, DC = 'default', A_DA = 'default')`. If set to `'default'`, the derivative of the cost function `DC` will be the standard cost function. If one wishes to use a different cost function, one simply has to pass the derivative of the desired cost function as an argument. If set to `'default'`, the pair `A_DA =[A,DA]` is treated respectively the sigmoid and the derivative of the sigmoid for feed-forward and backwards-progragation. If one wishes to change the activation function, then one simply needs to ass as argument a pair `[A,DA]` consisting of the desired activation function `A` and its directional derivative `DA`.

The reason that we take `DC` and `DA` as constructor arguments is because of the class function `backwards_propagation(input, target,step_size=0.01)`. In section section 1.4.2 we point out that to perform backwards propagation on a given MLP with activation function $f(z)$, we only need to know the the size of the network, the directional derivative of the desired cost function, and $\nabla f(\boldsymbol{z})$. The above mentioned class function performs one iteration of backwards propagation for the given input and target and step size.

Then there is: `train(training_input, training_output, epoch_start = 0, epochs = 100, number_batches = 1, completion_message = False)`. As the naming suggests, given a training input and output, this performs 100 epochs of training using stochastic gradient descent. If `completion_message` is set to `True`, then a message is printed to terminal every 5% of the training process to report progress.

The learning rate of the SGD based training can be controlled via the self instance `learn` of the class `learning_rate`. This is a class constructed by two arguments `t0` and `t1`, with a class function

`gamma(e,M,i)` which returns the output of the function:

$$\gamma(e, M, i) = \frac{t_0}{eM + t_1 i}$$

The function `train` is made so that it passes the correct number of batches `M` and epoch `e` to `gamma(e,M,i)`. This allows for easy control of the learning rate by modifying the parameters `t0` and `t1`.

3 Results and discussion

3.1 1D Ising model: modelling energy

In this subsection we present the results for the one dimensional Ising model. The data that we fit to is generated with $\Lambda = \{1, 2, \ldots, 40\}$, and with energy

$$E[s] = -\sum_{\substack{\langle x,y \rangle \\ x,y \in \Lambda}} s(x)s(y)$$

for all spin configurations s.

3.1.1 Linear regression

To make our results comparable to https://physics.bu.edu pankajm/MLNotebooks/HTML/NB_CVI-linreg_ising.html, we consider a sample of 600 randomly generated spin configuration, and evaluate our models using 3-fold cross validation. This ensures that the partition into training has a comparable ratio. An advantage of performing 3-fold cross validation is that our graphs represent averages of multiple fits.

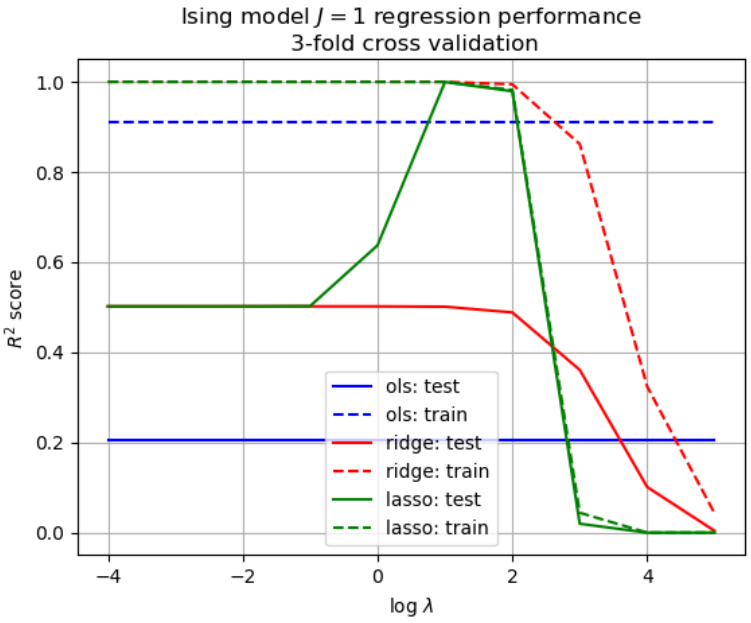

Figure 2: The R^2 scores of OLS, Ridge and Lasso using 3-fold cross-validation. The best model seems to be Lasso with $\lambda = 1$. The scores presented are clipped to stay within the unit interval.

As can be seen from fig. 2, our results resembles the ones found by Mehta et al. OLS and Ridge regression demonstrate a rather poor prediction ability. Using a lot more than 600 data points, we can obatin good results for OLS and Ridge. However, Lasso obtains a very good score for certain values of λ, even though we based our model on few data points.

Table 1: R^2-scores for the best models ($\lambda = 1$) from 3-fold cross validation using 600 data points.

$\lambda = 1$	Ordinary least square	Ridge	Lasso
R^2-score	0.206325	0.488431	0.979070

Our implementation of Lasso converges slowly (table 2) compared to ordinary least squares and ridge. We decided to cap the Shooting algorithm that we implemented at 25 iterations. This means that our notion of convergence for this algorithm is rather loose. On the other hand, it does not seem to be problematic in this case, given scores (fig. 2 and fig. 5). We observed that the shooting algorithm only needs 25 iterations for the λ around the optimal parameter $\lambda = 1$.

Figure 5 shows that the MSE for Lasso at the good values of λ is small. The bias and the variance however, are very large. Figure 3 and fig. 4 unsurprisingly shows that the MSE, bias and variance are all large for these models.

Table 2: Average time of 5 runs of ordinary least squares, Ridges and lassos on a sample of 400 spin configurations.

	Ordinary least square	Ridge	Lasso
$\bar{t}$ [seconds]	0.955149	0.934650	114.858

Table 2 shows the average time of all three linear regression algorithms. Obviously Lasso is a lot more computationally expensive than the other two, using over a 100 times more time to finish.

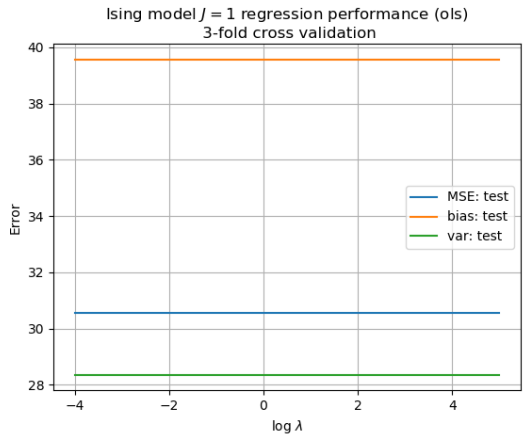

Figure 3: MSE, bias and variance of OLS regression using 3-fold cross-validation and 600 data points. OLS does not depend on λ.

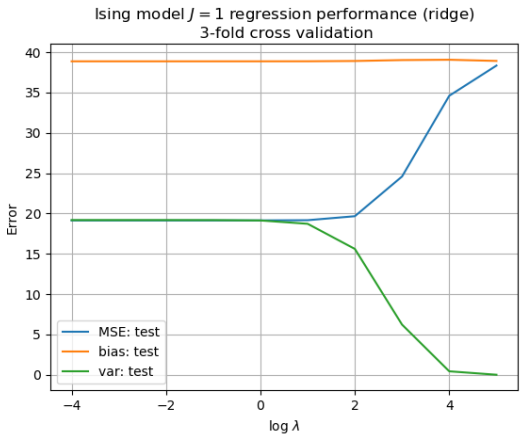

Figure 4: MSE, bias and variance of Ridge regression using 3-fold cross-validation and 600 data points for different values of λ.

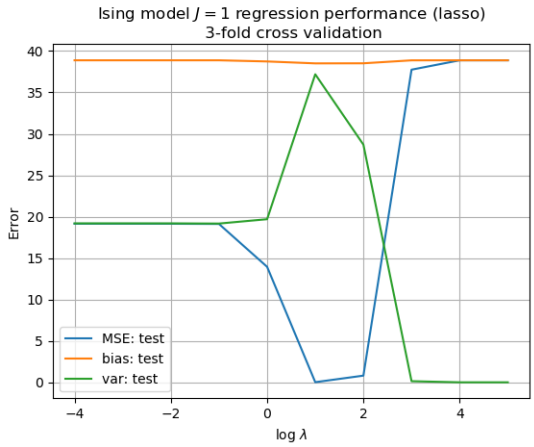

Figure 5: MSE, bias and variance of Lasso regression using 3-fold cross-validation and 600 data points for different values of λ.

3.1.2 MLP

If y denotes the output from the network, and t denotes the target value, then the cost function that we consider here is:

$$C(y) = \frac{(t-y)^2}{2}$$

After trail and error, the neural networks which gave us the best models were of the type $[L, nL, nL, 1]$ where $n \geq 1$, and where $L = 40$. Seemingly, higher n results in better models, but for few data points we found instances where raising n too much resulted in worse fits for the same number of epochs. Also, we found no benefits in changing the number of hidden layers. The learning rate that gave us the best results was $t_0 = t_1 = 1$. We do not mean to claim that no better models exist outside of the settings we tested.

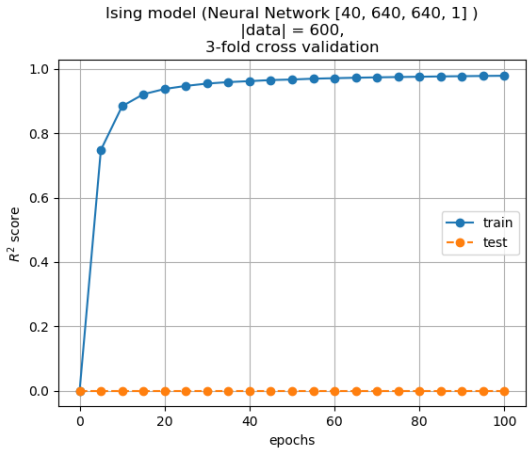

Figure 6: R^2 score of MLP regression of size [40,640,640,1] using 3-fold cross validation and 600 data points. The model works fine on the training data, but makes terrible predictions.

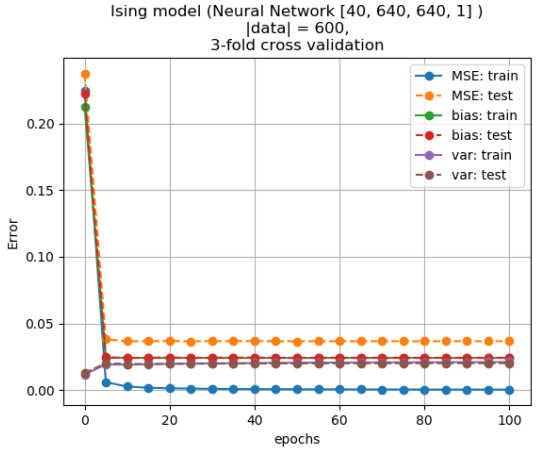

Figure 7: MSE, bias and variance of and MLP of size [40,640,640,1] using 3-fold cross-validation and 600 data points. The error is much lower than for OLS and Ridge regression.

Using the same amount of data as for linear regression fig. 2, we note from fig. 6 that the model performs extremely poorly on R^2 scores. Note that the scores presented are clipped to stay within the unit interval. We can see that the network finds a model that fits the training data fairly well. However, as we feed it too few data points, the model is makes terrible predictions, as is seen from the low R^2 score of the testing data. Figure 7 however, shows that in terms of MSE, bias and variance, MLP outperforms OLS and Ridge. Lasso is still better.

We see from comparing fig. 8 and fig. 9 to fig. 6 and fig. 7, that this is an example where an MLP needs to be fed with a lot of training data to make worthwhile predictions.

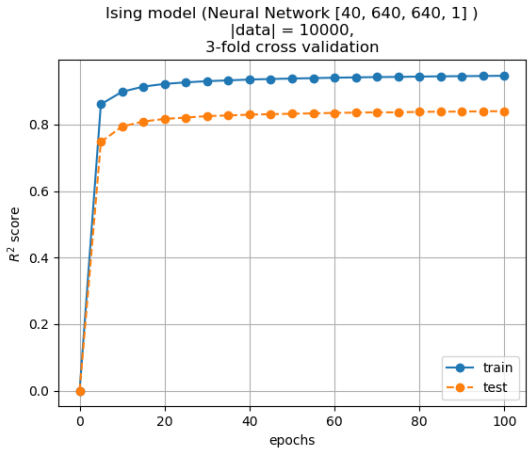

Figure 8: R^2 score of MLP regression of size [40,640,640,1] using 3-fold cross validation and 10000 data points. It yields acceptable results.

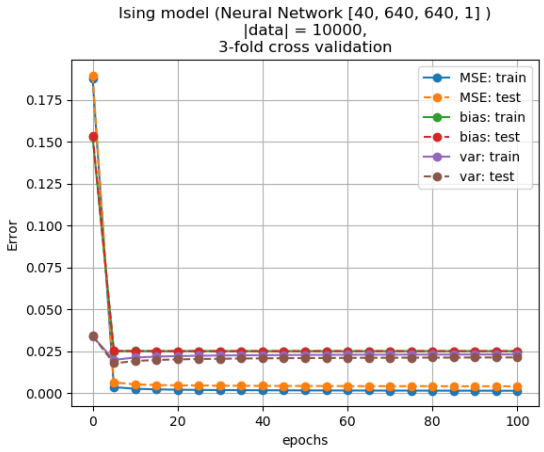

Figure 9: MSE, bias and variance of and MLP of size [40,640,640,1] using 3-fold cross-validation and 10000 data points. It yields acceptable results.

If we increase the number of data points, then the performance of the neural network increases significantly, as seen in fig. 8. So much that it becomes kind of comparable to Lasso after 100 epochs in terms of performance. This is also seen in terms of MSE in fig. 9.

The data for figures fig. 8 and fig. 9 took us roughly 4-5 hours to generate. An improvement here might lie in using stochastic gradient descent. The possibility of speeding up our stochastic gradient descent algorithm is discussed further in section 3.3

Table 3: 3-fold cross validation R^2-scores for neural networks on test data after 100 epochs of training for the two data set sizes we considered.

|data|	600	10000
R^2-score	-0.528647	0.840022

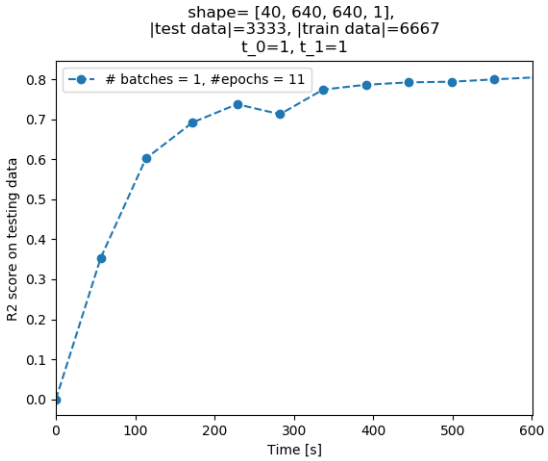

Figure 10: In the course of 10 minutes we are able to run 11 epochs of training. The R^2 score improves drastically in the course of the first few minutes, and then improves more slowly. After 2 minutes, corresponding to the time it takes take to run a lasso (table 2), we look at a R^2 score of about 0.6. In terms of speed, we can make the neural network find good results at a comparable rate as Lasso, however, as fig. 6 and fig. 8 show, we need to feed it more data to make good predictions on test data.

3.2 2D Ising model: classifying spin configurations

We turn over to the 2–dimensional Ising model. We let $\Lambda = \{1, \dots, 40\}^2$. We considered data from https://physics.bu.edu/ pankajm/MLReviewDatasets/isingMC/. The data consists of spin configurations on Λ divided into three types:

1. Ordered states (below some critical temperature).
2. Unordered states (above some critical temperature).
3. Ordered/unordered states (at some critical temperatur).

We model classification of a given spin configuration into the first two types, and disregarded classification of spin configurations at the critical temperature. The relation to the Ising model is that the energy of a spin configuration determines its associated temperature.

3.2.1 Logistic regression

For this problem, there seems to be no advantage in increasing the number of batches of the stochastic gradient descent (see fig. 11 and fig. 12). Using one single batch achieves a higher accuracy faster, in terms of epochs and in terms of time, than with mini batches.

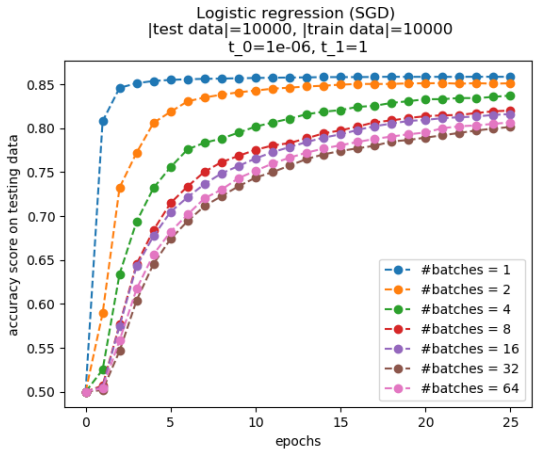

Figure 11: The accuracy score on test data after performing logistic regression using 10000 test and training data. Plotted against number of epochs for different number of mini-batches.

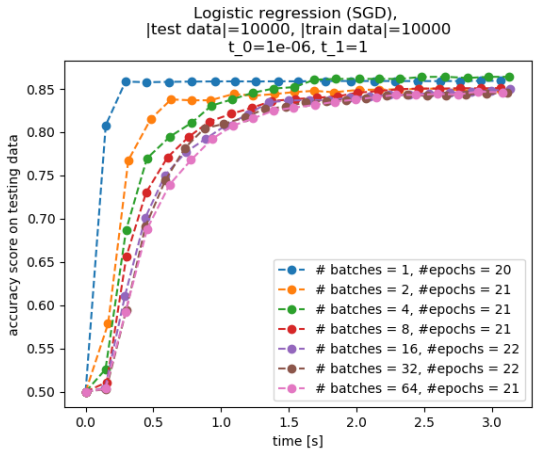

Figure 12: The accuracy score on test data after performing logistic regression using 10000 test and training data. Plotted against number of time for different number of mini-batches.

Table 4: The accuracy score of our best logistic regression model. The settings used were $t_0 = 10^{-6}$, $t_1 = 1$ and 25 epochs of training with a single batch (gradient descent) on 10000 training data points. The accuracy is measured on 10000 testing samples.

Best number of batches:	Accuracy
1	85.86%

We tried implementing Newton Raphson's method, but we ran into singular matrices for this data set after. We have not had the chance to add a penalty term, but if we had, then it would have been interested how it compared.

3.2.2 Neural network

Figures fig. 13-fig. 16 illustrate that choosing good learning rates can produce good models significantly faster. Seemingly after only 1 epoch. Table 5 summarizes the best model.

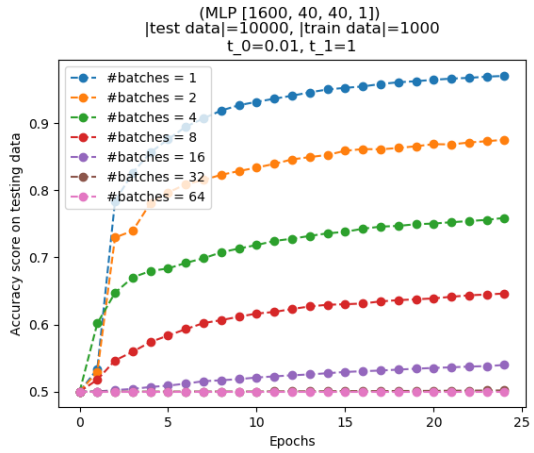

Figure 13: The accuracy gradually improves. The models with large batch sizes perform poorly.

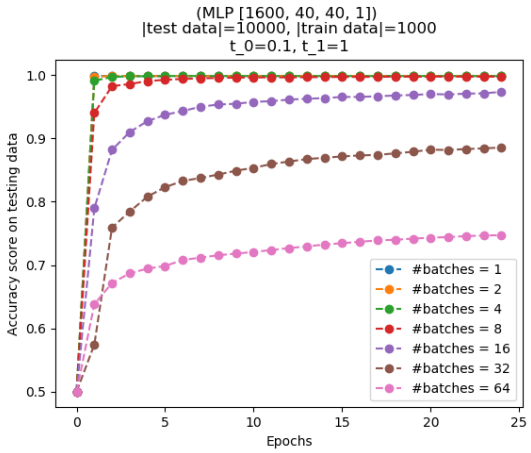

Figure 14: The accuracy shoots towards 100 % quick in the beginning if the model uses few batches.

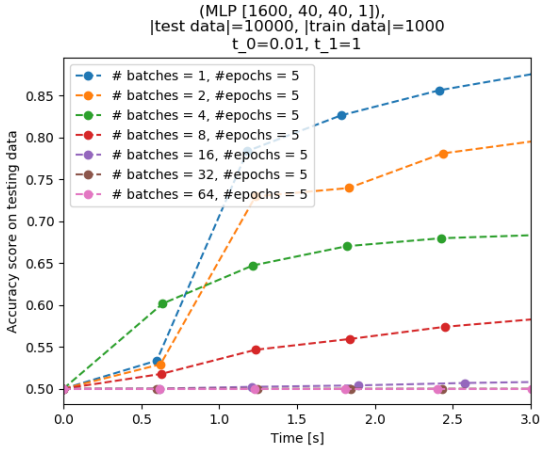

Figure 15: Accuracy for different number of mini batches in the first few seconds.

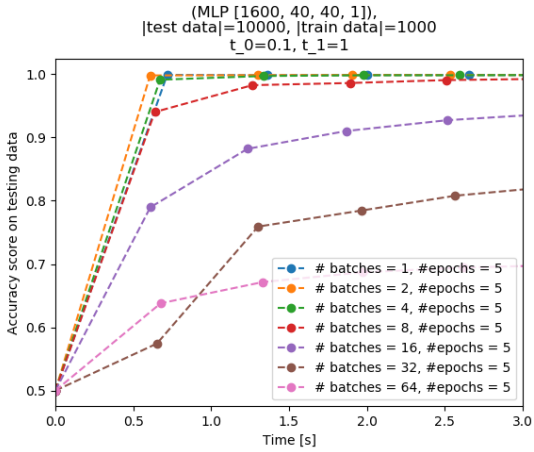

Figure 16: In the first epoch the models with batch size up to 4 are all practically good enough.

Table 5: The accuracy score of our best neural network model. The settings used were $t_0 = 0.1$, $t_1 = 1$ and 25 epochs of training with batches (stochastic gradient descent) on 1000 training data points. The accuracy is measured on 10000 testing samples.

Best number of batches:	Accuracy
1	99.86 %

3.2.3 Different activation functions (tanh, clipped ReLu, bump and hill)

In this subsection we consider alternative activation functions for our network in the classification problem. Again, the parameters that we present as optimal are chosen on the basis of trial and error. We note that the clipped rectified linear unit, performs excellent in terms of accuracy (fig. 18), and its speed beats the sigmoid by almost 6 times (fig. 16 and fig. 21). It offers a great alternative and is numerically cheaper to implement, however, it it seems that it is more sensitive to the number of batches used than the sigmoid. We see this seemingly asymptotic behavior in the accuracy/epoch ratio of the non-smooth activation functions.

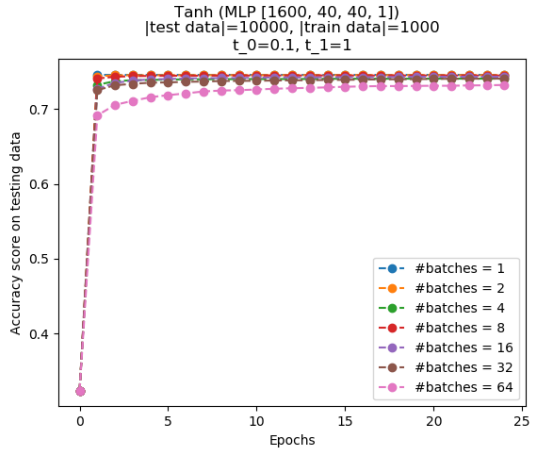

Figure 17: Tanh. Accuracy final epoch (on training data): 74.56 %. Best number of batches 1.

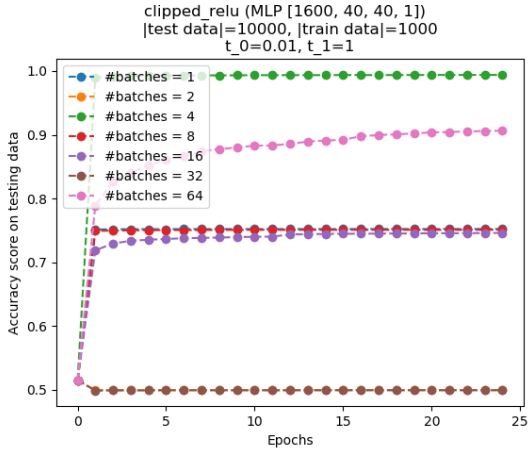

Figure 18: Clipped ReLu. Accuracy final epoch (on training data): 99.39%. Best number of batches: 4

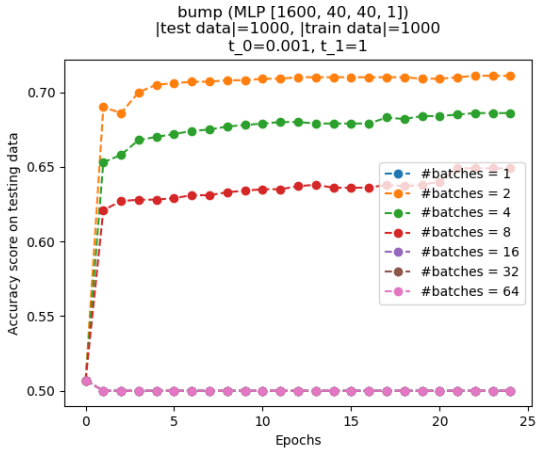

Figure 19: Bump. Accuracy final epoch (on training data): 71.10 % Best number of batches: 2

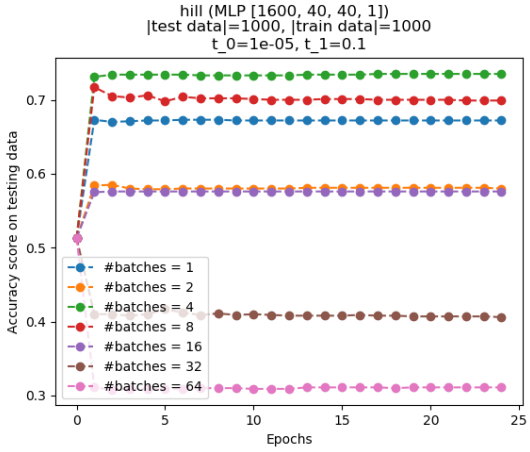

Figure 20: Hill. Accuracy final epoch (on training data): 73.5 % Best number of batches: 4

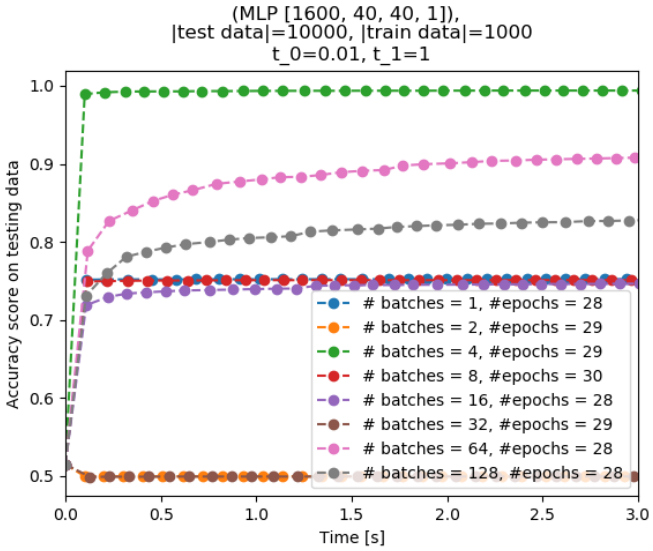

Figure 21: The clipped ReLu activation results in a SGD algorithm that is almost 6 times as fast as the one where we use a sigmoid function. See fig. 16. This is a huge improvement in terms of speed for a comparable accuracy. However, we see that this activation function is less stable in the sense that some of the batch configurations seem to cause slower learning.

3.3 Discussion about stochastic gradient descent

In the future, to seek out optimal learning rates, it might be interesting to fix a number of epochs, say 10, and plot R2-score/accuracy against varying learning rate parameters t_0 and t_1. One could simply iterate over a combination of t_0 and t_1 to find which combination yields the highest score. This will be

computationally more time consuming that what we have done, but it will give stronger estimates on an ideal learning rate, and offers a more sophisticated approach than what we do.

A possibility that we did not explore was to consider momentum based stochastic gradient descent. These methods also provide parameters which one may iterate over to find the optimal model.

As for the algorithm itself, in the literature we saw two version of it, and decided to study the one we saw in [Hjo18a]. That is, we iterate of all batches each epoch, and randomly pick one of the batches each iteration on which we train the MLP. The advantage of doing this is that no data points are left out in an epoch, but depending on the learning rate, not all data points count equally towards deciding the final weights and biases in the MLP. A plain disadvantage is that for batch sizes larger than 1, there is no speed improvement over steepest descent, since the number of calculations each epoch is unchanged. We see this experimentally in fig. 12, fig. 21, and fig. 10.

The other version that we saw, for example [Kon+14], iterates over a random subset of batches each epoch, on which to train the model. The advantage of this method over the one we went for is that it is faster to compute for a larger of batch size due to the possibility to skip batches. A disadvantage is that one risks disregarding important batches and therefore trains the MLP sub-optimally. Had we studied this method, it would also have been interesting to look for instances where a large batch number outperformed steepest descent significantly both in terms of speed and in terms of accuracy. Our code is easily modified to carry out the exact same plots with a modified SGD algorithm.

4 Conclusion

MLP's are very successful in tackling the classification problem that we presented. It performed better than logistic regression (85.86%). We see that the sigmoid (accuracy 99.86 %) and the clipped ReLu (accuracy 99.39%) both are very competent as activation functions. In the future, we see no reason not to test out clipped ReLu whenever a sigmoid is a good choice. It is definitely a nice gadget when one seeks to speed things up. As for the other activation functions that we tested; none seemed to offer any improvement over the sigmoid in this scenario.

A possible extension to the classification problem is to include the critical phase. Then it might be an idea to have the MLPs output layers consist of three nodes instead of only one, so that the one that flashes should correspond to exactly one of the three possibilities. As it stands, our code can do the above, but the logistic regression code needs modification. Another extension is to build upon our Newton Raphson algorithm to include an option for a penalty term to the cost function.

As for the mimicking the Ising model, our MLP's performed only so so. They required large chunks of training data, and many nodes in the hidden layers to get R^2-score higher than 0.8. This in turn meant that it took a long time to get results. Given more time and more nodes, we likely could have gotten higher scores with the settings we used. Lasso performed very well in the 3-fold cross validation that we ran, and on far fewer data points, so it was clearly a better alternative. For the same data size (600 samples) the Lasso had an R^2-score of 0.979070 on test data, whereas the MLP had a negative score -0.528647 after 100 epochs. On 10000 data sample, the MLP scored 0.840022. To give MLP's a new shot at the same problem, a suggestion might be to modify our stochastic gradient descent algorithm to only iterate over a randomly picked subset of the batches. This *might* speed things up and simulatiously give good results.

It is certainly possible to search for optimal parameters in in a more organized manner that what we have done here. Although being time consuming, we would have liked to try to iterate over learning rate parameters to find the optimal model. The current way we have implemented learning, which only depends on two parameters, may even result in 3D plots of score vs parameters which are nice to look at. Moreover, we could look into modifying how we iterate over batches in our implementation stochastic gradient descent and to include momentum based learning.

References

[BB18] Camilla Marie Baastad and Niels Bonten. *Linear Regression on Terrain Data*. https://github.com/CamillaBa/FYS-STK4155_Project_1/blob/master/Project1_FYS_STK4155.pdf. 2018.

[Hjo18a] Morten Hjorth-Jensen. *Data Analysis and Machine Learning Lectures: Optimization and Gradient Methods*. https://compphysics.github.io/MachineLearning/doc/pub/Splines/pdf/Splines-minted.pdf. [Online; accessed 11-November-2018]. 2018.

[Hjo18b] Morten Hjorth-Jensen. *Data Analysis and Machine Learning: Logistic Regression*. https://compphysics.github.io/MachineLearning/doc/pub/LogReg/pdf/LogReg-minted.pdf. [Online; accessed 11-November-2018]. 2018.

[Hjo18c] 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/pdf/NeuralNet-minted.pdf. [Online; accessed 11-November-2018]. 2018.

[Hor91] Kurt Hornik. "Approximation capabilities of multilayer feedforward networks". In: *Neural networks* 4.2 (1991), pp. 251–257.

[Kon+14] Jakub Konecny et al. "ms2gd: Mini-batch semi-stochastic gradient descent in the proximal setting. arXiv preprint". In: *arXiv preprint arXiv:1410.4744* (2014).

[Meh+18] Pankaj Mehta et al. "A high-bias, low-variance introduction to machine learning for physicists". In: *arXiv preprint arXiv:1803.08823* (2018).