FYS-STK4155 - Applied data analysis and machine learning

Project 2

Tommy Myrvik and Kristian Tuv
November, 2018

Github repository for the project:
https://github.com/Myrvixen/Fys-Stk4155-Project2

Contents

1 Abstract

Fitting a model to a set of data is not only good for finding patterns in the data, but also for predicting the outcome of new data sampled from the same distribution. While methods like linear and logistic regression does the job in many cases, they have their limitations in that they strictly model linear dependencies between the inputs and regressors. Neural networks are more flexible in that they are also able to capture *non-linear* dependencies in the data, and are often able to find good representations in the data where the other two methods are rendered useless.

In this projects we have tested all three methods on both 1D and 2D configurations of spins using the Ising Model, where we were trying to predict the energies of different 1D configurations, and classify whether 2D configurations are ordered or disordered. We found that while both linear regression and neural networks are able to predict the energies perfectly, logistic regression really struggles to classify the 2D-states correctly. A neural network with only one hidden layer *easily* outperformed the latter in prediction accuracy, and careful tuning of the network hyperparameters yielded *nearly* perfect results.

2 Introduction

Supervised learning (SL) is the task of using an algorithm for mapping an input value to an output value via a mapping function. During training of the model we know the input values and output values, and we want to know the relationship between them given by the mapping function. When this mapping function has been found, it can be used to make predictions of new outputs from new sets of input data.
There are enough popular SL-algorithms in existence to make it tedious to list them all. We are instead going to have a closer look at three of the most popular ones

- Linear Regression
- Logistic Regression
- Neural Network

In this project we will study these SL-algorithms evaluated on the famous Ising model.

2.1 Goals

In this project we will study classification and regression problems evaluated on the Ising model. We will study the regression algorithms of Ordinary Least Squares, Ridge Regression and Lasso Regression evaluated on a one dimensional Ising model where we aim to reproduce the coupling constant of the energies. The aim is to do so without our model having any prior knowledge of the Ising model. We will instead use a physicist magical intuition as our point of origin.

We then move on to including logistic regression and multilayer perceptron code for classifying the two dimensional Ising model into ordered and unordered phases.
As a continuous sanity check we will follow closely the works of Mehta et al, arXiv 1803.08823.

2.2 Structure

We start by brushing the surface of molecular dynamics and the Ising model, before we go deeper into the mathematical and computational theory of the supervised learning methods mentioned.
We then present the data sets and implementations of our algorithms, before giving the results of our analysis.
Finally we conclude the project and discuss future implementations.

I Theory

3 Understanding the Ising model

In studies of matter, a central subject is the understanding of phase transitions. In a phase transition properties of the matter in question changes due to changes in external factors such as temperature or pressure. We will have a look at the magnetic behavior of a material above and below a the critical temperature where a phase transition takes place, by using the Ising Model. The Ising model is a stochastic model of a ferromagnet. It was introduced by Wilhelm Lenz, and solved in the 1 dimensional case by Ernst Ising, hence it's name.
The Ising model uses a set of binary variables that represent magnetic dipole moments of atomic spins that can be in one of two states, up or down, which we will represent by +1 and -1. The spins are arranged on a lattice[4] where the neighboring spins can interact with each other. Before delving into the intrinsics of the Ising model, we will have a look at the properties of our system.

3.1 Canonical ensemble

We will imagine examining a material where we keep the number of particles N, the volume V and the temperature T constant. This means we are working in the canonical ensemble[3].

In the canonical ensemble, the probability of a given combination of spin orientations

in our system, also called a microstate[1], is given by the Boltzmann distribution

$$P_i(\beta) = \frac{e^{-\beta E_i}}{Z} \tag{1}$$

Where $\beta = 1/k_b T$, k_b being the Boltzmann constant and T being the temperature in Kelvin. E_i is the energy of a given microstate i and Z is the partition function in the canonical ensemble summed over all microstates M, which acts as a normalization factor.

$$Z = \sum_{i=1}^{M} e^{-\beta E_i} \tag{2}$$

The energy of a microstate can be found by using the Ising model, as will be described below.

3.2 Periodic Boundary Conditions

When doing molecular dynamics simulations in studies of matter, we only have the processing power to simulate a tiny system om particles. To pretend we are simulating bigger systems and to avoid dealing with the boundaries of our system, we often use periodic boundary conditions.
Periodic boundary conditions are a way of pretending the system we simulate is infinitely large. For a three dimensional system, like in figure 1, where the particles are free to move around, we need to decide what happens if a particle moves outside the edge of the simulation box (the box in the middle of figure 1). With periodic boundary conditions it will reappear on the opposite side of the box. Looking closely at figure 1 we see that the boxes are all duplicates of the simulation box. When a particle reappears on the opposite side of the box, this will in practice mean we are duplicating the simulation box an infinite number of times. This also means that particles close to one side of the box will be influenced by particles on the other side of the box, which creates the illusion of an infinite system.

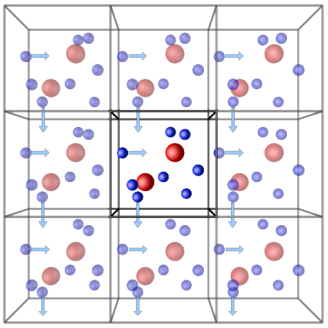


Figure 1: *A representation of periodic boundary conditions. The boxes are all duplicates of the middle box. Source:ISAACS*

[1]Say in a 1D-system of two particles, the four possible spin microstates are up-up, down-up, up-down and down-down

3.3 Ising Model

We are now ready to have a closer look at the Ising model.
In it's simplest form, the energy of the system is described by

$$E = -J \sum_{<kl>}^{N} s_k s_l \tag{3}$$

where s can take binary values, e.g ± 1, and represents spin, N is the total number of spins, J is the coupling constant describing the strength of the interactions between the the spin-spin products. In this simple model J is just a constant, but if the spin-spin interaction of neighboring spins are of different magnitudes, this can just as well be a vector.
$< kl >$ indicates we are only summing over the nearest neighbours in the system, where two sites k and l are neighbors if they differ by exactly one coordinate in either dimension.
Given a $J > 0$, it is energetically favourable for a spin-spin pair to be aligned, hence the orientation of a spin is likely to be aligned with a neighbouring spin.
Because we are in the canonical ensemble, the system in question will always be generated at a constant temperature T, given by the Boltzmann equation. As we will see below, the temperature of the system will greatly influence it's properties.
Consider a simple system of two particles. The possible microstates are

$$\uparrow\uparrow \qquad \downarrow\downarrow \qquad \uparrow\downarrow \qquad \downarrow\uparrow$$

Which gives the corresponding energies

$$E_1 = -J \qquad E_2 = -J \qquad E_3 = J \qquad E_4 = J$$

We see that the energies with aligned spins gives negative energies, and positive energies for opposite spins. This fact becomes very important when we consider the probabilities of finding different states given by the Boltzmann distribution (eq. (1)).
When the temperature goes towards zero, the negative energies, i.e the align spins, become more probable. If we move the system towards large temperatures, the positive energies becomes more probable, i.e a larger amount of the opposite facing spins are more probable.
What we can deduce from this, is that for low temperatures, there should be order in the system because the spins largely influence each other to be aligned. We call this the *ordered phase*.
For high temperatures we deduce the opposite effect. Spins are generally not aligned, and we will have trouble finding a lot of order in the system. We call this the *disordered phase*.

3.4 The Critical Phase

For high temperatures we deduced the system exhibiting a disordered phase. For low temperatures the system will be ordered. Then of course there must occur a phase

transition between the two phases.
Lars Onsager[5] showed that for a two dimensional system, a phase transition will occur on a critical temperature T_c given by

$$T_c = 2.269185T_0 \tag{4}$$

Where T_0 is defined by J/k_b.
Onsager derived this relationship for an infinite system, and for a finite system the critical temperature will be slightly larger[6]

$$T_c \approx 2.3T_0 \tag{5}$$

3.5 Other uses of the Ising model

As physicists we view s as the spin of a particle, however it is not hard to imagine using the Ising model for describing other binary issues. As an example, and as an analogy to keep in mind, the economist Thomas Schelling used an Ising-like model at low temperatures to describe the complex issue of self-segregation in cities in the United States.[2] Even without overarching factors like racism, people tend to self-segregate in societies according to religion, language, income etc.
We can formulate the Ising model like this.
The physicist way: A spin finds it energetically favourable to be aligned with it's neighbor, which leads to clustering at low temperatures.
The Schelling way: A person finds it pleasant to be more alike it's neighbors, which leads to segregation at low tolerance levels.

4 Machine learning methods

4.1 Linear Regression

In general, regression revolves around curve fitting, i.e. finding the best fit for a data set to a model of choice. This model can be a linear function, a polynomial, a sum of sinusoids, or anything in between. For our regression model to be *linear* doesn't mean that we fit the data to a first order polynomial, but rather that the models output is a linear product of the predictor variables $\hat{x} = [x_0, x_1, ...x_{p-1}]^T$ and the regression parameters $\hat{\beta} = [\beta_0, \beta_1, ...\beta_{p-1}]$ so that:

$$\tilde{y} = \sum_{i=0}^{p-1} \beta_i x_i \tag{6}$$

It's important to note here that the vector $\hat{x}$ is not a vector of different x points in the data set, but the representation of a single point in the model of choice, i.e. $\hat{x} = [1, x, x^2]^T$ for a second order polynomial.

In turn, an actual data point from the data set y can be written as:

$$y = \sum_{i=0}^{p-1} \beta_i x_i + \epsilon \tag{7}$$

where ϵ is the error/difference between the data and the model estimations:

$$y - \tilde{y} = \epsilon \tag{8}$$

When fitting a model to a data set, we also get a set of equations with the form described above, where the goal is to find a vector $\hat{\beta}$ that translates the input predictors into the data points in the best way possible. While the $\hat{\beta}$ vector is the same for all data point pair (x_i, y_i), each pair has it's own predictor vector $\hat{x}$ (i.e. each x value is represented in terms of the model), which can be gathered in a matrix $\hat{X}$ called the *design matrix*. Writing the set of equations for n (x, y) pairs using the elements in the design matrix we get:

$$\begin{aligned}
y_0 &= \beta_0 x_{00} + \beta_1 x_{01} + ... + \beta_{p-1} x_{0p-1} + \epsilon_0 \\
y_1 &= \beta_0 x_{10} + \beta_1 x_{11} + ... + \beta_{p-1} x_{1p-1} + \epsilon_1 \\
&\vdots \qquad\qquad\qquad \vdots \\
y_i &= \beta_0 x_{i0} + \beta_1 x_{i1} + ... + \beta_{p-1} x_{ip-1} + \epsilon_i \\
&\vdots \qquad\qquad\qquad \vdots \\
y_{n-1} &= \beta_0 x_{n-10} + \beta_1 x_{n-11} + ... + \beta_{p-1} x_{n-1p-1} + \epsilon_{n-1}
\end{aligned}$$

A convenient realization here is that all these equations can be gathered and written on matrix form like this:

$$\hat{y} = \hat{X}\hat{\beta} + \hat{\epsilon} \tag{9}$$

There are many ways to find an estimate for the optimal $\hat{\beta}$ for a given data set (with or without noise) and a model of choice, each with their pros and cons. Three of the most popular methods are the Ordinary Least Squares, Ridge and Lasso methods, and these will be outlined in the subsections of this chapter.

4.1.1 Ordinary Least Squares

In Ordinary Least Squares we are finding the β values that minimizes the residual sum of squares (RSS), which here is defined as the cost function Q:

$$Q(\beta) = \sum_{i=0}^{n-1} (y_i - \tilde{y}_i)^2 = \sum_{i=0}^{n-1} (y_i - \sum_{j=0}^{p-1} x_{ij}\beta_j)^2 \tag{10}$$

or on matrix form:

$$Q(\beta) = (\hat{y} - \hat{X}\hat{\beta})^2 = (\hat{y} - \hat{X}\hat{\beta})^T(\hat{y} - \hat{X}\hat{\beta}) \tag{11}$$

We can interpret the RSS as the sum of all the squared errors, or distances, from the data points y to the corresponding points $\tilde{y}$ in the regressed line. This is illustrated in figure 2. We want as small an error as possible and hence find the minimum of the cost function (here the RSS) with respect to β.

$$\frac{\partial Q(\beta)}{\partial \beta_j} = 0$$
$$\frac{\partial}{\partial \beta_j}\left(\sum_{i=0}^{n-1}(y_i - \sum_{j=0}^{p-1} x_{ij}\beta_j)^2\right) = 0$$
$$-2\sum_{i=0}^{n-1} x_{ij}(y_i - \sum_{j=0}^{p-1} x_{ij}\beta_j) = 0$$

which we on matrix form recognize as:

$$-2\hat{X}^T(\hat{y} - \hat{X}\beta) = 0$$
$$\hat{X}^T\hat{y} = \hat{X}^T\hat{X}\beta$$

$$\hat{\beta} = (\hat{X}^T\hat{X})^{-1}\hat{X}^T\hat{y} \tag{12}$$

Further, by differentiating again with respect to β_j one can show that:

$$\frac{\partial^2 Q}{\partial \beta^2} = 2\hat{X}^T\hat{X} \tag{13}$$

where the resulting matrix is positive definite (if we assume none of the columns in $\hat{X}$ are linearly dependent), which in turn proves that $Q(\beta)$ is at a *minimum* in equation (12).

Equation (12) finds the best fit to data it's given, and in general the more data points, the better fit. This sounds good, but can give really catastrophic results when applied in practice. There is a difference between data that the model has seen, and data it hasn't seen. If the data the model is trained on is full of noise, the OLS regression will usually fit to this noise, given enough model complexity. It doesn't look for underlying features in the data, it just fits to what's given to it. It's prediction on any point it hasn't seen before is thus likely to be completely off, making the once promising model a quite bad one. This makes the variance of the model quite high, where the prediction of new data will vary greatly, depending on the data it has been fitted to. We'll now look into a few regression methods that might be able to look past the noise in the data, giving better predictions of unseen data, namely the Ridge and Lasso methods.

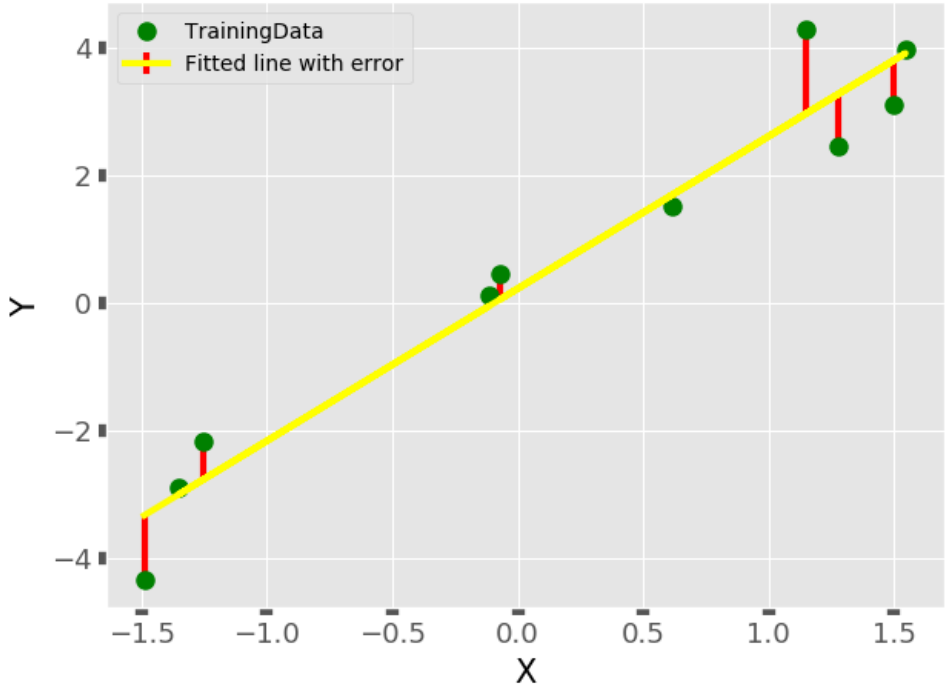


Figure 2: *Training data in green and a fitted line in yellow with the errors in red. We find the RSS by squaring these errors and summing them.*

4.1.2 Ridge

In OLS our startingpoint for minimizing our cost function was the RSS

$$Q(\beta) = RSS(\beta) = \sum_{i=0}^{n-1}(y_i - \sum_{j=0}^{p-1} x_{ij}\beta_j)^2 \tag{14}$$

The ridge regression method is a modification to OLS, where the cost function now has an extra term

$$Q(\beta) = \sum_{i=0}^{n-1}(y_i - \sum_{j=0}^{p-1} x_{ij}\beta_j)^2 + \lambda \sum_{j=0}^{p-1} \beta_j^2 \tag{15}$$

where the last term often is referred to as the *penalty term*. In this term we see that we have the factor β^2 (times a constant λ) that's effectively being added to the RSS. From this we can understand that large β values are bringing larger contributions to the cost function, and this is where the term *penalty* comes in. As the cost function is the quantity we want to minimize, it's beneficial for all the β values to be small, effectively penalizing the large ones and driving them down. Punishing the β values too much however sacrifices the fit given in the RSS-term, causing a low penalty, but high RSS. Minimizing the cost function then seeks to find a balance between a good general

fit (RSS) and the amplitude of the feature estimates (β values). This balancing makes complex models more viable, as the large β values that would cause overfitting in OLS are "shrunk" down so that more of the features are contributing at the same levels.

This process of shrinking the coefficients is called regularization, and the strength of this regulating is determined by the regularization constant, λ. The value of this constant varies from problem to problem, meaning there's no general "best value". For each new model we have to try out several values for λ, and see which one performs best.

Looking back on equation (15) we will derive an expression for the optimal $\hat{\beta}$ in the same way as we did for the OLS. Finding the minimum for the cost function gives us:

$$\frac{\partial Q(\beta)}{\partial \beta_j} = 0$$

$$\frac{\partial}{\partial \beta_j}\left(\sum_{i=0}^{n-1}(y_i - \sum_{j=0}^{p-1} x_{ij}\beta_j)^2 + \lambda \sum_{j=0}^{p-1}\beta_j^2\right) = 0$$

$$-2\sum_{i=0}^{n-1} x_{ij}(y_i - \sum_{j=0}^{p-1} x_{ij}\beta_j) + 2\lambda \sum_{j=0}^{p-1}\beta_j = 0$$

Writing this on matrix form gives:

$$-2\hat{X}^T(\hat{y} - \hat{\beta}\hat{X}) + 2\lambda\hat{\beta} = 0$$

$$(\hat{X}^T\hat{X} + \lambda)\hat{\beta} = \hat{X}^T\hat{y}$$

where we finally end up with the expression:

$$\hat{\beta}^{ridge} = (\hat{X}^T\hat{X} + \lambda I)^{-1}\hat{X}^T\hat{y} \tag{16}$$

where λ is actually $\lambda \cdot I_p$, so that the λ value is effectively added to the diagonal of the $\hat{X}^T\hat{X}$ matrix.

One thing to note about the ridge method (and also the lasso method discussed in the next section) is that it's more a method of *predicting* rather than *infering*. The OLS on one hand always finds the best fit to the points it's being *fitted on*, but it doesn't necessarily transfer well to points it *hasn't* seen before (it often actually gives catastrophic results as we'll see later). As the cost of the ridge method always will be higher than the one for OLS, it will give a sub-optimal fit for the points it's being fitted on. In contrary to OLS however, the ridge method is less likely to overfit to the training data, by using all features a little rather than using few features a lot. This makes the model less sensitive to changes in the data set, where it in practice lowers the variance as λ increases, at the cost of some bias. A model with a good choice of λ is likely to generalize better on noisy data, making it better at predicting unseen points.

4.1.3 Lasso

The Lasso regression method is another method that is very similar to the Ridge. The cost function looks like this:

$$Q(\beta) = \sum_{i=0}^{n-1}(y_i - \sum_{j=0}^{p-1} x_{ij}\beta_j)^2 + \lambda \sum_{j=0}^{p-1} |\beta_j| \tag{17}$$

where the only difference lies in the penalty term, where β^2 factor is replaced by the absolute value $|\beta|$. While the difference is quite subtle, it has a few consequences in terms of both further mathematical derivations, and the resulting regressions.

First of all, it's not possible to find a analytical expression for the $\hat{\beta}$ at which the cost function is minimized, as we've done for both the OLS and ridge. This means we have to use other minimization methods like gradient descent or Newton's method to try to find an approximation to the minimum. One will usually have to choose a random $\hat{\beta}$ vector to initialize the minimization, and depending on how close this guess is to the minimum, we might experience significant differences in number of iterations for the algorithm to converge. In some cases it won't converge at all. Such a method is needless to say a little more complicated to implement than the OLS and Ridge cases, so in this project we'll use Scikit Learn's Lasso regression function to make sure that the results we are getting are correct.

The Lasso methods effect on the resulting β values are much of the same as for Ridge, but there is a significant difference: While the Ridge drives the β values down towards, but not exactly 0, Lasso does just this. This is called feature selection, where some parameters can be set to 0, leaving only the more significant β values in the model. This makes the model simpler, and like the ridge, makes it less prone to over-fitting. As with Ridge, the parameter λ controls the strength of the penalization, and a stronger penalization lowers the variance (at the cost of some bias) also for this method.

For more information on the linear regression methods and their performances on different types of data sets, feel free to check out our report on **FYS-STK4155 - Project 1**[9].

4.2 Logistic regression

Logistic regression is a method closely related to linear regression, although the applications are somewhat different. Where linear regression aims to predict exact values like energy values, stock prices or height of a terrain (like we did in Project 1[9]), logistic regression on the other hand seeks to place every input example into predefined "bins", or *classes*. These classification problems can encompass anything from classifying an object in an image, to determining if a student is likely to show up to class after x hours of sleep. What we want as an output here is either just the label of the class the model

thinks the inputs belong to (called "hard classification") or probabilities/scores for each class (called "soft classification"). In the "soft" version the model also picks out a label as well, corresponding to the class with the highest probability, but the probabilities also gives us information on *how sure* the model is of it's pick. As we'll see shortly, the accuracy metric used to evaluate the model only cares about correct labeling, but the cost function actually needs the class scores as well. Logistic regression follows the same recipe as linear regression, but as we'll see, has one extra step.

In the same way, we get a linear combination of the regression parameters $\hat{\beta}$ and the regressors $\hat{X}$, so that

$$\hat{\tilde{y}} = \hat{X}\hat{\beta}. \tag{18}$$

In linear regression the vector $\hat{\tilde{y}}$ (which can also be called *activations* or *logits*) is a vector of length N, where N is the number of training examples. In logistic regression this vector can also be a matrix of dimension $N \times K$ depending on the number of classes K, which also means that $\hat{\beta}$ has dimension $p \times K$ (a set of βs for each class). It's normal however to only use regression parameters for the first $K-1$ classes so that the dimension of $\hat{\beta}$ is $p \times K - 1$. The reasoning for this is that the output of the regression will be in terms of probabilities as previously stated, so that all probabilities must sum up to 1. We can then calculate the probabilities for the $K-1$ first classes, and simply say that the probability for the last one is:

$$p(C = K|\boldsymbol{x}) = 1 - \sum_{k=1}^{K-1} p(C = k|\boldsymbol{x}) \tag{19}$$

The most common way to compute these probabilities is by applying the Softmax function to the class activations. This function can in general be written as:

$$p(C = k|\boldsymbol{x}) = \frac{\exp(\beta_{k0} + \sum_j \beta_{kj} x_j)}{\sum_{l=1}^{K} \exp(\beta_{l0} + \sum_j \beta_{lj} x_j)} \tag{20}$$

In the case where we only directly calculate the probabilities of the $K-1$ first classes the above equation take the form:

$$p(C = k|\boldsymbol{x}) = \frac{\exp(\beta_{k0} + \sum_j \beta_{kj} x_j)}{1 + \sum_{l=1}^{K-1} \exp(\beta_{l0} + \sum_j \beta_{lj} x_j)} \tag{21}$$

with the probability for the last class being

$$p(C = K|\boldsymbol{x}) = \frac{1}{1 + \sum_{l=1}^{K-1} \exp(\beta_{l0} + \sum_j \beta_{lj} x_j)}. \tag{22}$$

It can be shown however that the two approaches above are equivalent, but we'll come back to the proof of this in section 4.2.1 on binary classification, as it's easier to see in

this case.

The cost function for logistic regression compares the computed probabilities with the true label/class of the input. Often we'll see that these come as a single number corresponding to the true class label, while the output of our regression model may have K outputs, one for each possible class. It's common practice to "expand" the true label into a vector of the same size as the output, where it has value 1 at the true class, and 0 everywhere else. This kind of vector is called a *one-hot vector*, and for a model with four possible classes this vector can for instance look like this:

$$\hat{y} = [0, 1, 0, 0]$$

where the class at index 1 is the true one. It's on this form we would like the output from out model to look like as well, where it's 100% certain that the input belongs to one class, and 0% for the others. Unless our model perfectly describes the input data however, we will probably get scores for all classes that are different from 0 and 1. The difference between the desired output and the actual output gives rise to an error estimation, which we can define as our cost function for the problem. The most common one to use in logistic regression is the *negative log-likelihood*, or *cross-entropy* which is the more common name in the literature. It stems from the *Maximum Likelihood Estimation principle*, which aims to maximize the probability of seeing the observed data in a data set $\mathcal{D} = (y_i, x_i)$. The individual probability of a specific outcome can in general be written as:

$$P(\mathcal{D}|\hat{\beta}) = \prod_{i=1}^{n} p(y_i|x_i, \hat{\beta}) \tag{23}$$

The cross-entropy can be found by taking the negative log of these probabilities, and can in it's very simplest form be defined as:

$$H = -\sum_i y_i \log p(y_i|x_i, \hat{\beta}) \tag{24}$$

where the sum runs over all examples, with y_i being the "true" probabilities, and p_i the computed probability according to eqs. (20) - (22). It's common to denote this quantity as H, but as we're here treating it as a cost function, we will use C from now on. It's important to note that y_i still only takes the values 0 or 1, and is only 1 for the one true class. This means that it's only the probability p_i for the true class that gives contribution to the cross-entropy, and hence it's common in some literature to denote p_i as p_{y_i} instead.

We can also use regularization in the same way as in linear regression, where we simply add an extra term to the cost function. The Ridge and Lasso regularization are often referred to as *L2* and *L1* regularization respectively in the context of logistic regression and neural networks. From here on we'll use these terms when we're talking about these types of machine learning.

We will go more in detail about the cross-entropy and it's gradients in the next part about binary classifiers, as the results found in that case are easily generalized to any number of classes. As we in this project only will deal with binary labeled data, it's natural to delve into what simplifications can be made in this case.

4.2.1 Binary classifier

Restricting our classifier to only 2 classes we can simplify eqs. (20), (21) and (22) to:

$$p(k=0,1|\boldsymbol{x}) = \frac{\exp(\beta_{k0} + \sum_j \beta_{kj} x_j)}{\sum_{l=1}^{2} \exp(\beta_{l0} + \sum_j \beta_{lj} x_j)} \tag{25}$$

and

$$p(C=0|\boldsymbol{x}) = \frac{\exp(\beta_0 + \sum_j \beta_j x_j)}{1 + \exp(\beta_0 + \sum_j \beta_j x_j)} \tag{26}$$

$$p(C=1|\boldsymbol{x}) = \frac{1}{1 + \exp(\beta_0 + \sum_j \beta_j x_j)} \tag{27}$$

We can see that the probability for class 0 simply is the *sigmoid function*, which is defined as:

$$f(x) = \frac{e^x}{1+e^x}. \tag{28}$$

We will discuss this function more in section 4.3.3 on activation functions, but it has one property that's worth mentioning here. In the binary classification case, the model will chose class 0 if it's probability is larger than 0.5 . The sigmoid function has the nice property that it's rotationally symmetric around $f(x) = 0.5$, meaning it's distribution is equal above and below this line. Thus it has no bias towards choosing one class over the other.

Using the standard definition of the softmax function (eq. (25)) for a binary classifier as a staring point, we can show that it's equivalent with the approach in eqs. (26) and (27). For simplification we'll express the activations $\beta_0 + \beta_1 x_1 + ...$ on matrix form as $\hat{X}\hat{\beta}$, and $\hat{\beta}_0$, $\hat{\beta}_1$ as the weights corresponding to class 0 and 1 respectively. We then have:

$$p(k=0,1|\boldsymbol{x}) = \frac{\exp(\hat{X}\hat{\beta}_k)}{\exp(\hat{X}\hat{\beta}_0) + \exp(\hat{X}\hat{\beta}_1)} \tag{29}$$

We can divide both the numerator and denominator by $\exp(\hat{X}\hat{\beta}_1)$, and rearranging some yields some familiar results:

$$p(k=0,1|\boldsymbol{x}) = \frac{\exp(\hat{X}\hat{\beta}_k - \hat{X}\hat{\beta}_1)}{\exp(\hat{X}\hat{\beta}_0 - \hat{X}\hat{\beta}_1) + \exp(\hat{X}\hat{\beta}_1 - \hat{X}\hat{\beta}_1)}$$
$$= \frac{\exp(\hat{X}(\hat{\beta}_k - \hat{\beta}_1))}{1 + \exp(\hat{X}(\hat{\beta}_0 - \hat{\beta}_1))}$$

giving

$$p(k=0|\boldsymbol{x}) = \frac{\exp(\hat{X}(\hat{\beta}_0 - \hat{\beta}_1))}{1 + \exp(\hat{X}(\hat{\beta}_0 - \hat{\beta}_1))} = \frac{\exp(\hat{X}\hat{\beta})}{1 + \exp(\hat{X}\hat{\beta})} \tag{30}$$

$$p(k=1|\boldsymbol{x}) = \frac{\exp(\hat{X}(\hat{\beta}_1 - \hat{\beta}_1))}{1 + \exp(\hat{X}(\hat{\beta}_0 - \hat{\beta}_1))} = \frac{1}{1 + \exp(\hat{X}\hat{\beta})} \tag{31}$$

with

$$\hat{\beta} = -(\hat{\beta}_1 - \hat{\beta}_0). \tag{32}$$

This shows that the weight-vector $\hat{\beta}$ that's being trained in this "sigmoidic" approach is simply the difference between the weight-vectors $\hat{\beta}_0$, $\hat{\beta}_1$ one would get for each of the two classes in the standard softmax approach. This restriction actually means that we don't get more degrees of freedom, even though we introduce twice the number of weights. Although the results of the classifications will be the same in both cases, there is a difference in what the weights actually learn. The vector $\hat{\beta}$ is quite simple and direct in that it learns to separate between the two classes, while the vectors $\hat{\beta}_0$ and $\hat{\beta}_1$ each contain information on the characteristics of each class. The output of the sigmoid method directly gives information on the labeling as well, as it picks class 0 if the resulting probability is larger than 0.5. $\hat{\beta}_0$ and $\hat{\beta}_1$ on the other hand is only responsible for giving a score for their respective classes, and it would require an "outside force" to actually compare the activations and pick the largest one.

This all seems to argument that the direct "sigmoidic" version is the better one, as we get the same results for fewer regression parameters. More parameters also introduce the danger of increasing the variance of the model. It is however cases where it's desired to have weights for each class, as the characteristics of each class may be of interest.

Mehta et al [1] performed softmax classification on the MNIST[2] data set, and when the trained weights are plotted it can be seen that the images they create actually resemble the digits.

The cross-entropy in the binary case can be shown to be written as:

[2]MNIST is a data set of all hand-written digits 0-9

$$C(\hat{\beta}) = -\sum_{i=1}^{n}(y_i(\beta_0 + \sum_j \beta_j x_j) - \log\left(1 + \exp\left(\beta_0 + \sum_j \beta_j x_j\right)\right)) \tag{33}$$

Regularization of the weights can also be added to the cost function in the same manner as in the the linear regression case.

To minimize the cost function we need to find an expression for it's gradient with respect to the regression parameters β_0 and β_1. The derivatives can be found to equal

$$\frac{\partial C(\hat{\beta})}{\partial \beta_0} = -\sum_{i=1}^{n}\left(y_i - \frac{\exp\left(\beta_0 + \sum_j \beta_j x_j\right)}{1 + \exp\left(\beta_0 + \sum_j \beta_j x_j\right)}\right) \tag{34}$$

$$\frac{\partial C(\hat{\beta})}{\partial \beta_1} = -\sum_{i=1}^{n}\left(y_i x_i - x_i\frac{\exp\left(\beta_0 + \sum_j \beta_j x_j\right)}{1 + \exp\left(\beta_0 + \sum_j \beta_j x_j\right)}\right) \tag{35}$$

which on compact form can be written as

$$\frac{\partial C(\hat{\beta})}{\partial \hat{\beta}} = -\hat{X}^T(\hat{y} - \hat{p}). \tag{36}$$

To this gradient we can also add regularization terms, which are simply the derivatives of the terms discussed in the Ridge and Lasso sections (section 4.1.2, 4.1.3):

$$L_2 \Rightarrow 2\lambda\hat{\beta}, \qquad L_1 \Rightarrow \lambda\frac{\hat{\beta}}{|\hat{\beta}|} \tag{37}$$

The cost function has no analytic expression for it's minimum with respect to $\hat{\beta}$, so we'll have to follow the gradients by applying minimization methods to find an approximation for the minimum. Some of these methods will be explored in section 4.4.

4.3 Neural Networks

The brain consists of billions of nerve cells called neurons. The internal function of a neuron is quite complex, but it's essential purpose is simple. When exposed to external stimuli, like a smell, the electrical potential of a neurons membrane will change. When the membrane potential reaches a certain level it fires an electrical pulse which in transmitted to other neurons in nearby vicinity, and thereby raising their membrane potential. In this way information (the electrical signal) propagates through the brain.
We can imagine arranging neurons in layers, where all the neurons in one layer are connected to the neurons in the next. In a real brain the neurons in a single layer will also be connected to each other, but for simplicity we will pretend they are not.
A neuron will only fire if a certain membrane potential-threshold is reached, and all, or none, or some of the neurons in a layer might fire at the same time. Information is

fed through the network layer by layer until the information reaches a final output layer where an action is taken, say eating the delicious pie that generated the smell.
We can summarize this model of a neuron with the mathematical model

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b_i\right) = f(u) \tag{38}$$

where y is the output from a single neuron, u is the weighted sum of the signals x_i generated by the neurons in a previous layer or the input, b_i is the resting membrane potential and f is an activation function equating to the membrane potential threshold of the neuron.
We will use this layer model of the brain in constructing a neural network like the *Multilayer Perceptron*.

4.3.1 Multilayer Perceptron

There are several ways of constructing a neural network, but the most straight forward way is the so called fully connected feed forward network, also called a *multilayer perceptron* (MLP). As the name suggests, these models consists of multiple layers which processes the input, and produces an output. An MLP consist of *at least* three layers; one input layer, at least one *hidden layer*, and an output layer.

A visual representation of a MLP is shown in figure 3. To continue our brain analogy, every circle in the figure 3 represents a neuron. The output from a neuron is given by equation (38). The input layer consist of the data we are evaluating, say a picture. This input is weighted and processed according to eq. (38), and passed to the next layer, called a hidden layer. The output from the hidden layer can be passed along to another hidden layer where the process is repeated. This process is repeated through an arbitrary number of hidden layers before the last hidden layer eventually passes it to the output layer. We can design the output layer in any way we want, so we can in a sense (try to) make the network learn anything, as long as we provide comparable target outputs.

4.3.2 Feed-Forward

The process of feeding a neural network an input which is processed through each layer to generate an ouput is called *feed-forward*. The input to a neural network can consist of any number of examples N, each having p regressors. A single example hence has the shape

$$\hat{x}^T = [x_0, x_1, \ldots, x_p]. \tag{39}$$

This can for example be pixels of a picture or a certain spin configuration of the Ising model. We will here denote the number of inputs (or nodes) to a layer l as N_l.

Following eq (38) the output from node i in the first hidden layer will be

$$a_i^1 = f^1(z_i^1) = f^1\left(\sum_{j=1}^{N_0} w_{ij}^1 x_i + b_i^1\right) \tag{40}$$

where z is called the *activation* of the layer, and a the output of the layer after being evaluated by an activation function f. We also assume the activation function f to be the same within a single layer, but can be different in subsequent layers. The sum j runs over all outputs generated in the previous layer, and for the first layer, this would just be a sum over all the inputs to the neural network. The superscript indicates what layer we are evaluating.
The second hidden layer takes the output from the first hidden layer as input, and will thus give the output

$$a_i^2 = f^2(z_i^2) = f^2\left(\sum_{j=1}^{N_1} w_{ij}^2 a_i^1 + b_i^2\right) = f^2\left(\sum_{j=1}^{N_1} w_{ij}^2 f^1\left(\sum_{j=1}^{N_0} w_{ij}^1 x_i + b_i^1\right) + b_i^2\right) \tag{41}$$

This pattern can be extended to as many hidden layers as we wish and as we can see, the only independent variables are the input values.
The feed-forward process of equation (41) can be generalized as

$$a_i^l = f^l(u_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l a_j^{l-1} + b_i^l s\right) \tag{42}$$

This also applies for the last layer (output layer) L, where the only difference is in the activation function f. If we're doing a classification problem, the activation z^L will be squeezed between 0 and 1 by a sigmoid function, or else we simply do nothing. It's important not to use an activation function in the same manner as the other layers *in addition* to the sigmoid just mentioned, as this will just throw away information without an additional benefit. The output of the final layer a^L is usually denoted as $\tilde{y}$, as this is the output received from the "black box" that is the neural network.

4.3.3 Activation functions

The activation functions purpose is to introduce non-linearities into our otherwise linearly composed network. They are essential when we have one or more hidden layers, as they force the different layers to learn different things about the input data. If we don't use activation functions on the hidden layers it can be shown that the addition of these layers essentially has no effect in terms of increasing the complexity of our model. If we assume we have one hidden layer with no activation, the output of the network can be written as a simple linear combination of the weight matrices and biases:

$$\tilde{y} = \hat{W}_1(\hat{W}_0\hat{x} + \hat{b_0}) + \hat{b_1} \tag{43}$$
$$= \hat{W}_1\hat{W}_0\hat{x} + \hat{W}_1\hat{b_0} + \hat{b_1} \tag{44}$$
$$= \hat{W}\hat{x} + \hat{b} \tag{45}$$

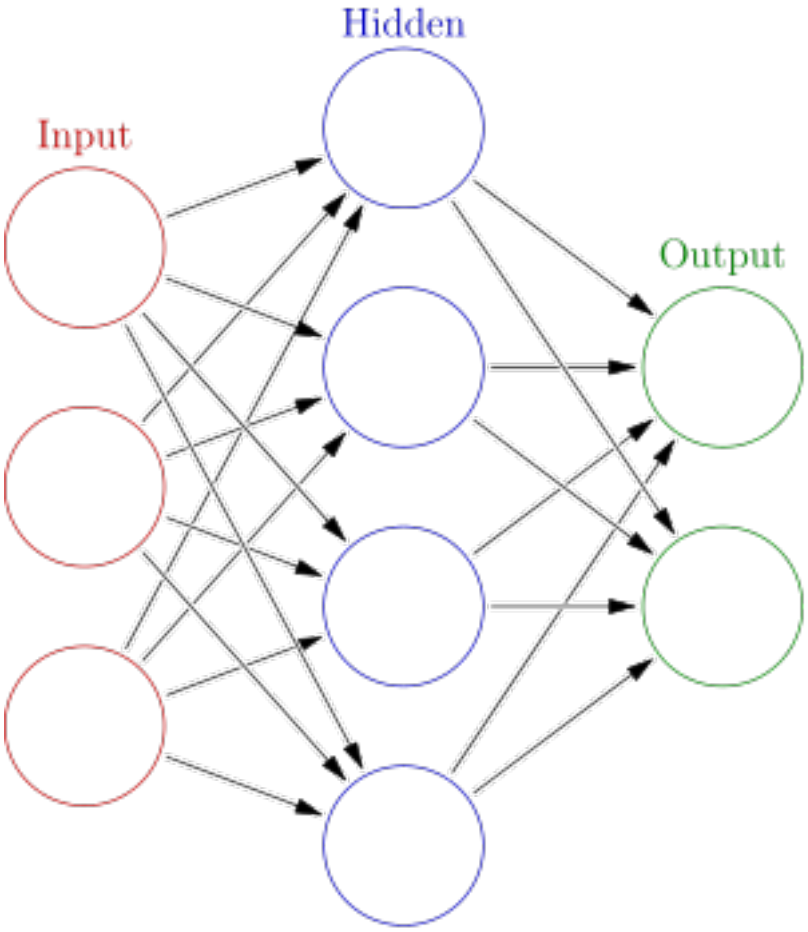


Figure 3: *Illustration of a Multilayer Perceptron. Every node in a layer is connected to all the nodes in the next layer, making it a fully connected neural network. Every neural network consists of an input layer, an output layer and zero or more hidden layers Source: Artificial Neural Network*

with $\hat{W} = \hat{W}_1\hat{W}_0$ and $\hat{b} = \hat{W}_1\hat{b_0} + \hat{b_1}$. By instead introducing a non-linearity onto the activation $\hat{W}_0\hat{x} + \hat{b_0}$ the transition from the first to the second line is no longer possible, and the model will actually be able to make use of both layers.

There are many possible activation functions one could use, and there is at the moment a lot of research going on exploring the performance and effects of different the ones. In this project we will use three of the most common ones, namely the sigmoid, tanh, and ReLU activation functions. The functional form of these can be seen in Figure 4.

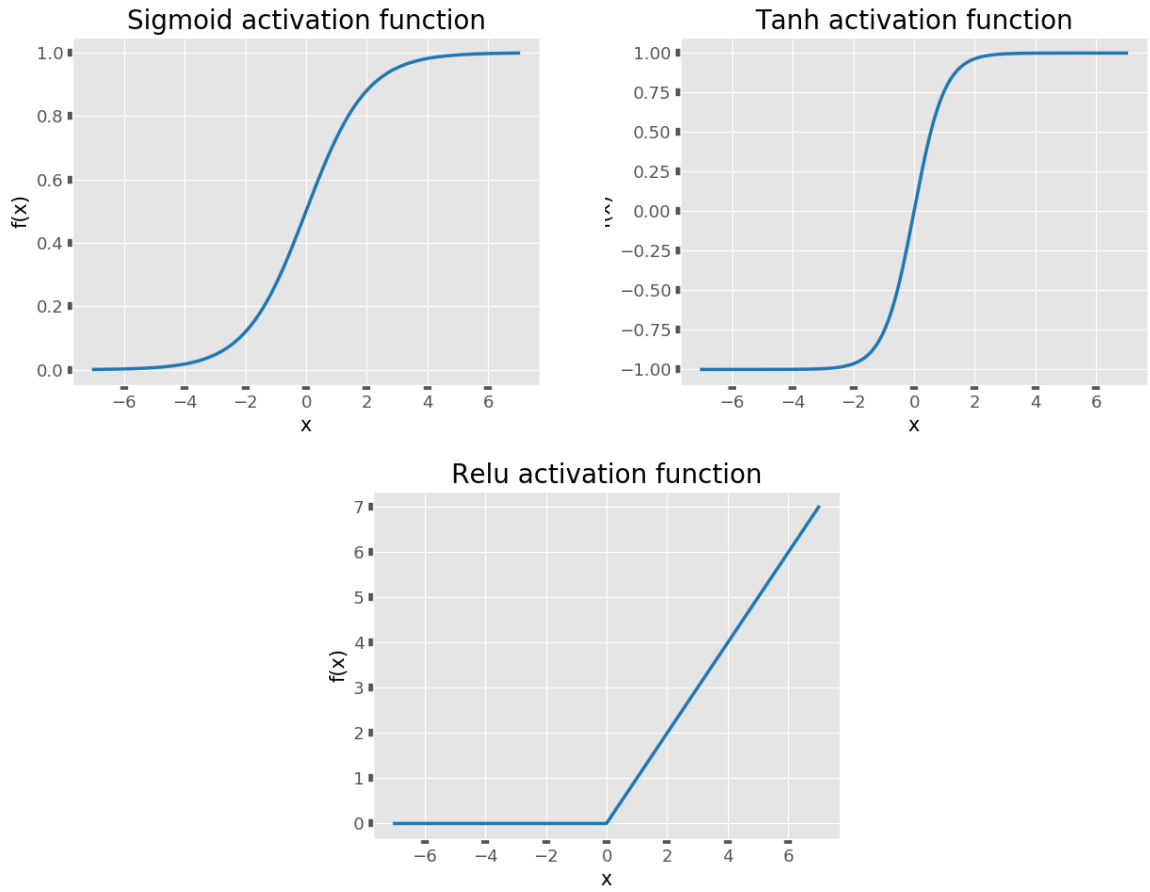


Figure 4: *Plots showing the behaviour of the sigmoid, tanh, and ReLU activation functions.*

All three of these have their pros, but also their potential shortcomings, and there is still no definite answer to which one is *the one* to choose when building the architecture of the network. The Standard course CS321n has a short, but good discussion of these functions[7], and we will list some of their observations here.

The sigmoid function, which was briefly mentioned in section 4.2.1, squashes the input to a value between 0 and 1. This means it's not zero-centered, but rather centered at $f(x) = 0.5$ as mentioned earlier, which is not an optimal property when dealing with a multilayer perceptron. As the output of a neuron always is positive, the gradient of the activation will also be positive, hence the update of the weights will be all positive or negative (depending on the overall gradient). This may create a zig-zag-like behaviour of the updating of the weights, but this effect is negated some when we average gradients over a batch.

The bigger shortcoming of the sigmoid function is a problem regarding vanishing gradients. By looking at the functional form in Figure 4 we see that the function flattens

out when $x > 2$ and $x < -2$. This means that the gradients in these regions are close to 0, leading to very small updates to the weights for each iteration. Hence, one should be careful not to initialize the weights with too large values, and preprocessing of the data would also help to circumvent this problem.

Even though it's function values can be a little expensive to compute as it uses exponentials, one advantage of the sigmoid is that it's derivative is very easy and cheap to calculate, which is one of the main reasons for it being widely used:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{46}$$

The tanh activation is very similar to the sigmoid, but this function is squeezing the input to the interval $[-1, 1]$ instead of $[0, 1]$. This has the wanted property in that it's output is zero-centered, creating a more dynamic updating of the weights. It does however suffer from the same vanishing gradient problem as the sigmoid, and from it's functional form it seems to saturate a little faster than it's competitor. This function is also a little expensive to compute, but it's derivative however is in turn quite cheap:

$$\tanh'(x) = 1 - \tanh(x)^2 \tag{47}$$

While the sigmoid and tanh activations are quite similar, the family of ReLU functions introduces a completely different functional form. It's form is quite peculiar, where it's linear in x in the region $x > 0$, and 0 everywhere else:

$$ReLU(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$$

It does however have some nice properties. Krizhevsky et al.[8] report that the ReLU function outperform both the sigmoid and tanh functions when training with stochastic gradient descent, a minimization method which will be discussed in section 4.4. It's also very cheap to calculate both the function values and it's gradient, which is simply:

$$ReLU'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$

It does however have dangerous drawback if not treated correctly. If the training step is too large (too large updates in weights), one might risk that the weights get updated to a distribution where the input to the ReLU function is always negative, meaning constant zero output and zero gradient flowing through to the weights. This means that these neurons are essentially "dead", and in severe cases, huge fractions of your network may be "non-operational". But as mentioned, as long as one is careful with the learning rate, the ReLU function is reported to perform really well, and is the preferred activation function among many researchers in the field.

There exists modifications to the standard ReLU function like the Leaky-ReLU and ELU, which both modify the part of the function which gives 0 to instead give a small

gradient to try to avoid some of the problems with dying neurons. In this project however, we will stick to the standard ReLU function, and be extra mindful of our learning rates.

4.3.4 Backpropagation

With the network having produced outputs through the feed-forward process and evaluated these with a cost function, we now need a way to calculate the gradients of the parameters in order to lower the cost, and communicate these back to each layer of parameters in the network. As we'll see, we'll have to run through the layers *backwards*, updating each weight matrix for each layer as we pass. This process is called *backpropagation*.

As the network is recursively built in that it needs the output from the previous layer in order to compute an output for the current layer during the feed-forward process, it's derivatives will also depend on each other in the same way during backpropagation, only this dependence goes the opposite way. This means that the derivatives of the parameters in one layer can be found by a series of chain-rules through the upper layers. We start at the computed cost function, and the presence of an activation function in the output layer depends on whether we're doing classification or not, as discussed in the feed-forward section. The gradients of the weights and biases in the output layer can be found from the following chain:

$$\frac{\partial C}{\partial W^L} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}\frac{\partial z^L}{\partial W^L} \tag{48}$$

$$\frac{\partial C}{\partial b^L} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}\frac{\partial z^L}{\partial b^L} \tag{49}$$

Now if we want to enter the next layer instead, we replace the last derivative with $\frac{\partial z^L}{\partial a^{L-1}}$. If we now write out a longer chain of derivatives, we will see a pattern emerge:

$$\frac{\partial C}{\partial a^0} = \left[\frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}\right]\left[\frac{\partial z^L}{\partial a^{L-1}}\frac{\partial a^{L-1}}{\partial z^{L-1}}\right]\left[\frac{\partial z^{L-1}}{\partial a^{L-2}}\frac{\partial a^{L-2}}{\partial z^{L-2}}\right]\dots \tag{50}$$

We see that in order to "traverse" the layers, the same kinds of derivatives will be calculated at each one, making it possible to write compact code. It's normal to call the brackets above the *errors* of the layers δ^l, with the first one containing the cost function being the *output error* δ^L. These errors can be written as:

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{51}$$

$$\delta^l = ((w^{l+1})^T\delta^{l+1}) \odot \sigma'(z^l) \tag{52}$$

where $\odot$ is the *Hadamard product* (element-wise mulitplication), and $\sigma'(z^l) = \frac{\partial a^l L}{\partial z^l}$.
At each layer we will compute the derivatives wrt. the weights and biases:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$$
$$\frac{\partial C}{\partial b_j^l} = \delta_j^l$$

before continuing on following eq. (50). This goes on until we reach the first hidden layer, where we see that the partial derivatives of the weights depends on the inputs:

$$\frac{\partial C}{\partial w_{jk}^1} = x_k\delta_j^1 \tag{53}$$

It's also important to use regularization in neural networks, and this is implemented in the same way as for linear and logistic regression. As we may have more than one weight matrix, we have to add the contributions from all weights in all layers to the cost function. This means that the gradients for all weight matrices we find during backpropagation gets an extra term each:

$$\frac{\partial C}{\partial w_{jk}^l} \mathrel{+}= \lambda w_{jk}^l, \qquad \frac{\partial C}{\partial w_{jk}^l} \mathrel{+}= \lambda\frac{w_{jk}^l}{|w_{jk}^l|} \tag{54}$$

for L2 and L1 regularization respectively.

4.3.5 Hyperparameters and initialization

Before any propagation in either direction can unfold, we have to actually set up and structure the neural network in terms of how many hidden layers we want, and how many nodes we have in each. This structure is called the *architecture* of our network.

We stated that we can use an arbitrary number of hidden layers in our neural network. This is true, but it is most likely not necessary. The universal approximation theorem states that given an activation function which is continuous, non constant and bounded, any continuous function can be approximated by a neural network within an arbitrarily small error using only a single hidden layer containing a finite number of neurons.[10]. This in practice means that we in most cases would not need more than one hidden layer for approximating a function with our neural network. There is a small catch however, we do not know how many neurons are needed, and in many cases it will be more computationally efficient to use several hidden layers with fewer nodes, than to use just one enormous layer. What we ultimately seek is to find the *sparsest* architecture in order to keep the variance and computational cost low, that still gives good results. What one will often see is that increasing the number of layers and nodes past a certain point gives little to no gain in model accuracy, rendering all the extra computations more or less useless.

The numbers of layers and nodes are examples of so-called *hyperparameters*, which are parameters that are set *before* training, in contrary to the weights and biases which are

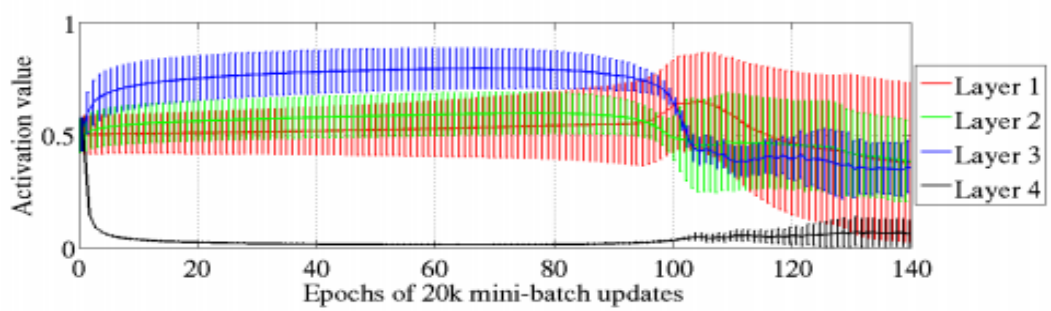


Figure 5: *Plot showing the mean output of the sigmoid activation function and it's standard deviation (vertical bars) found in Source:Understanding the difficulty of training deep feedforward neural networks, where the simulation was run for 300 test examples. The plot illustrates a common problem of the sigmoid function, where in the top layer (layer 4) the output quickly saturates slowing down the learning of the network for 100 epochs*

parameters that are adjusted *during* training. Learning rate and regularization strength are other examples of hyperparameters that we need to adjust in order make our network perform as well as possible. A common way to determine the values of these is to run cross-validation on the training set (discussed in **Project 1**[9]).

For a long time the idea of deep neural networks, i.e neural networks with a lot of hidden layers, was abandoned due to a common problem of unstable gradients. Gradients often get smaller as we get deeper into the network, that is the inputs to the layers tend to get large, and looking at the sigmoid function and the tanh function in figure 4 we see that a large input results in a gradient of approximately zero. In architectures like Recurrent Neural Networks[11], which is beyond the scope of this article, one would also encounter the problem of exploding gradients where the gradients grow bigger and bigger and the algorithm diverges.

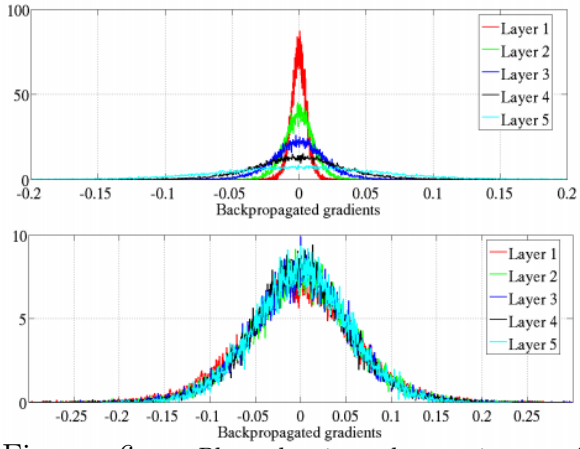


Figure 6: *Plot showing the variance of the backpropagating gradients found in Source:Understanding the difficulty of training deep feedforward neural networksevaluated using the tanh activation function. Top plot shows the variance through the layers with no normalization when initiating weights. Bottom plot show variance when weights have been normalized during initiation.*

In "Understanding the Difficulty of Training Deep Feedforward Neural Networks" Glorot and Bengio[12] found that the combination of using the logistic sigmoid activation function and initializing the weights using the standard normal distribution resulted in the variance of the output within a layer to increase from layer to layer until the activation function would saturate at the top. This would of course result in decreased learning potential for the network. In figure 5 Glorot and Bengio ran 300 simulations of the same network using sigmoid activation functions

in each layer and a standard normal distribution initialization of the weights[3]. As we can see, due to the increasing variance of the output values from the previous layer, the output quickly saturates for higher level layers. This pattern would happen with both the tanh activation function and the sigmoid function but was made worse by the sigmoid function due to it's mean being 0.5 not 0[13].

Glorot and Bengio proposed using a normalization scheme for initializing the weights, also called He initialization, which resulted in the variance staying approximately constant through the network and as a result the chance of saturation is severely reduced. In figure 6 we see the result of the variance when using the normalization scheme compared to initiating from the standard normal distribution.

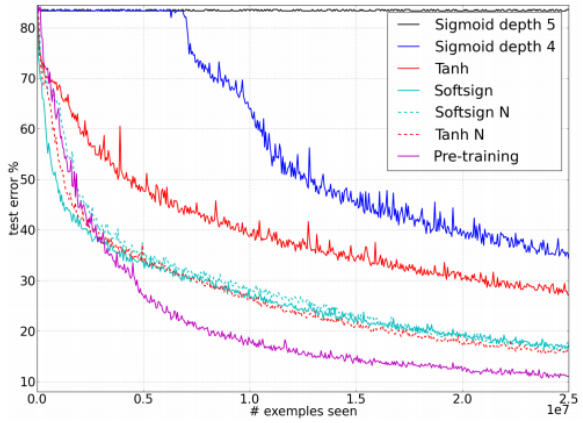


Figure 7: *Plot showing the effects on the final error when normalizing the initialization*

The effect they found of the normalization N on the learning ability of the network is show in figure 7, where the Sigmoid, Tanh and Tanh N (normalized) are the relevant simulations. As we can see, the normalization of the weights has a large impact on the final test error.
Their reasoning in proposing the normalization scheme was to make sure the variance of the input and output of a layer would be as close as possible, and the same should be true for the variance of the gradient backpropogation. They found it is not possible to guarantee both, unless the number of nodes for the input and output are the same. Because of this they proposed a compromise solution

Activation function	Uniform distribution [-r, r]	Normal distribution
Sigmoid	$r = \sqrt{\frac{6}{n_{inputs} + n_{outputs}}}$	$\sigma = \sqrt{\frac{2}{n_{inputs} + n_{outputs}}}$
Tanh	$r = 4\sqrt{\frac{6}{n_{inputs} + n_{outputs}}}$	$\sigma = 4\sqrt{\frac{2}{n_{inputs} + n_{outputs}}}$
ReLU	$r = \sqrt{2}\sqrt{\frac{6}{n_{inputs} + n_{outputs}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{inputs} + n_{outputs}}}$

This is not however a huge issue when dealing with networks with few hidden layers,

[3]The standard deviation in this plot is the variation of the 300 simulations, and is not the same as the variance we are talking about when saying the output variance increases.

and in this case a uniform or normal distribution between smaller values as $[-0.5, 0.5]$ is fine.

4.4 Minimization methods

After finding the derivatives of the cost function using back propagation, we now need to utilize these derivatives to find the minimum.

Lets imagine you are Captain Kirk stranded on a mountain, temporarily blinded and without your communicator. If you want to get back to the good life of space travel and instant food-generating machines, you need to find your way down to your crew as quick as possible. Perhaps your best strategy would be feel the ground around you and take a step in the direction of the steepest descent, then feel the ground again and repeat. This method is guaranteed to get you down to a local minimum, and if you are lucky this is also the global minimum where your crew is. If not, you're not only stranded, but you will have to make the food yourself.
We will now view different but quite similar approaches to this minimization problem.

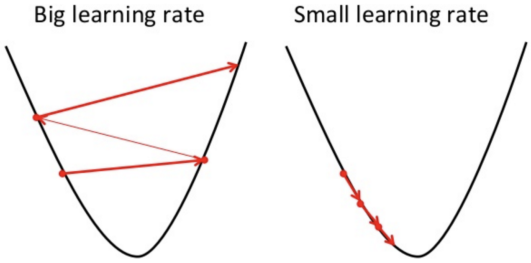


Figure 8: *Visualization of small and big learning rate. Choosing the learning rate too big, we might miss the minimum entirely and diverge. For too small learning rate we will move toward a minimum, but it might be to slow to converge with the amount of epochs we are willing to use in training. Source: Towards Data Science*

4.4.1 Gradient Descent

Gradient descent was essentially explained in the introduction. Using back propagation to find the derivatives, i.e the local gradient of the error function with regards to the parameters, we know the direction of steepest ascent, and we will just move in the opposite direction towards a local minimum. It is important to remember that even though we know the direction, we have no idea how far from the minimum we are and we do not know if the minimum is the global minimum or just a local one.

An important choice we have to make is how big of a step we want to take towards the minimum. This step is called the learning rate and choosing a suitable learning rate is essential for our model to converge. In figure 8 we see examples of small an big learning rates when descending to the minimum of a convex function. As we see, choosing a learning rate too big, we might jump across the valley and even end up in a point further

away from the minimum than before. If this is happening our model will diverge.
If we choose our learning rate too small on the other hand, we will indeed converge toward the minimum, but the trade-off is you might spend far more time on the descent than you are willing to do. The trick here is simply finding a goldy locks zone through trial and error.

Another consideration is the shape of our terrain. Not all cost functions are smooth convex functions, and like in figure 9, depending on where we start our descent we might end up in a local minimum or we might stagnate if hitting a plateau where the gradient is approximately zero.

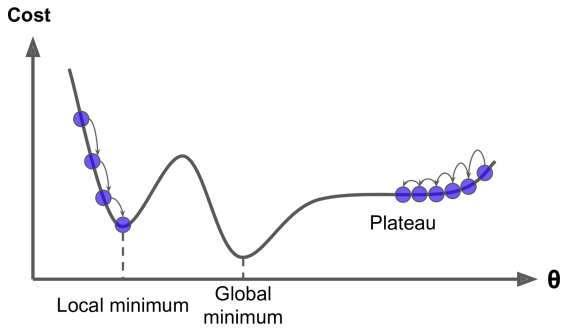


Figure 9: *Visualization of local and global minimum. In neural networks we are always looking for the optimal solution, which corresponds to the global minimum. However if our cost function is non convex we might be unlucky and get trapped in a local minimum. With a small learning step we might also essentially get trapped on a plateau Source: Towards Data Science*

Luckily several cost functions, like MSE can be shown to be convex functions, i.e if no hidden layers are used, the function only has one minimum. The function might however be severely elongated if we are using features with different scales, and the risk of hitting a plateau is always present.

As soon as hidden layers are used however, we need to be aware that the minimum we find is likely to only be a local minimum and we must either settle for this solution, or try another gradient descent method in an effort to avoid getting stuck in the local minimas.

4.4.2 Batch Gradient Descent

Batch gradient descent (BGD) is the most straight forward way of doing gradient descent. As the name implies, at each gradient step we are calculating the gradient for every point in the *entire* training set. The drawback of doing this is that it is terribly slow when computing on large data sets.
In section 4.2.1 we introduced the cross-entropy (equation 33) as a cost function, and derived it's derivative with respect to the weights β (equation 36). If we use the cross-entropy as a simple example, and calling the learning rate η we have the following equation for a single gradient descent step

$$\beta^{i+1} = \beta^i - \eta \frac{\partial C(\hat{\beta})}{\partial \hat{\beta}} = \beta^i - \eta \hat{X}^T(\hat{y} - \hat{p}) \tag{55}$$

where i represents the number gradient descent steps taken. Notice the entire trainingset $\hat{X}^T$ is being used.

4.4.3 Stochastic Gradient descent and Mini-Batch

In batch gradient descent, we calculated the gradient on the entire training set when doing a single gradient descent step. On the other extreme we can choose to only use a single training example chosen randomly with replacement for our descent. This is what is called *Stochastic Gradient Descent (SGD)*.
Because only one training instance has to be kept in memory during training, this is a much faster algorithm for training on big training sets. Another plus is the fact that due to random fluctuations in the algorithm, there is a chance that if we get stuck in a local minimum during the descent, we might actually bounce out of this minimum and continue our descent.

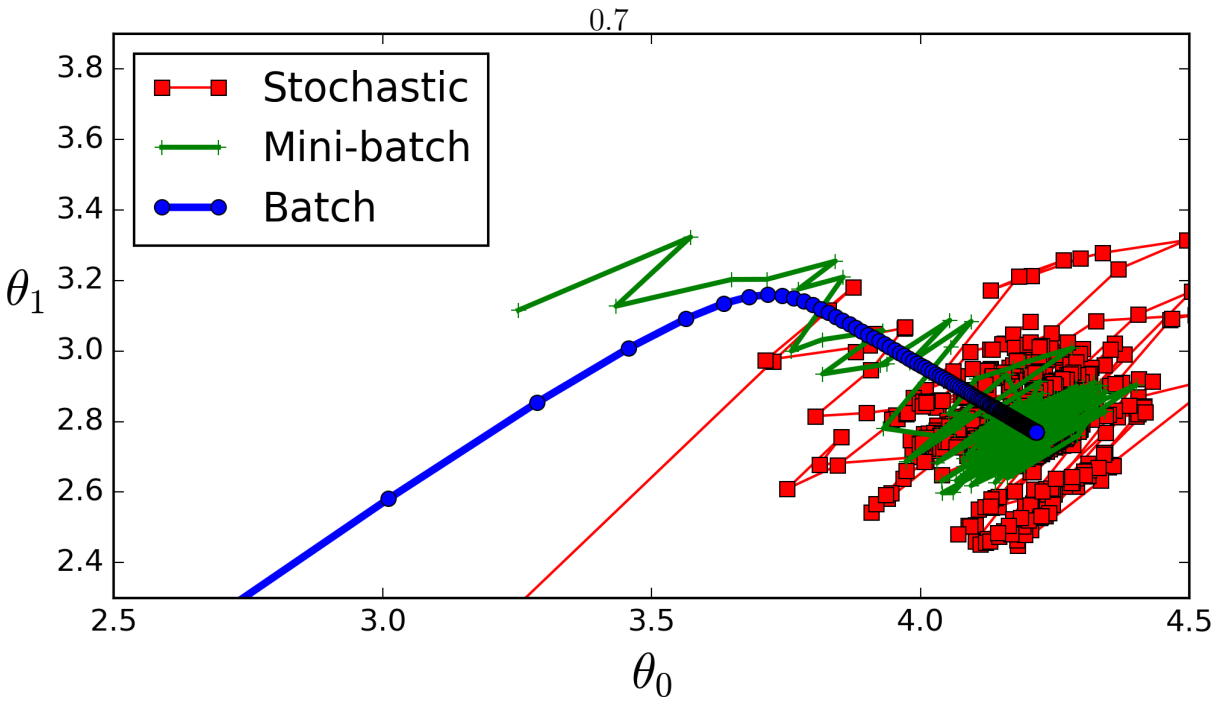


Figure 10: Figure showing Stochastic Gradient Descent, Batch Gradient Descent and Mini-Batch Gradient descent. We clearly see the differences in in randomness of the three algorithms. Source: Hands on machine learning with scikit-learn and tensorflow, Aurélien Géron, p.122

The biggest draw-back to this algorithm is also because of it's stochastic nature. Because of this randomness, the descent towards the minimum will be quite noisy compared

to batch gradient descent and we can not expect the algorithm to gently settle at the minimum like batch gradient descent does, instead it will keep bouncing around even though we have reached the minimum. The final parameters will still generally be good enough though, and because of the huge benefit of speeding up our descent SGD is preferred over BGD.
One can think of *Mini-Batch (MB)* as a happy medium of the two previous algorithms. We still choose a random sample from the training set with replacement but in contrast to SGD we are choosing more than a single sample. This will reduce the randomness of the descent, and we will to a larger degree settle in the minimum depending on the size of our mini-batch. This algorithm will be even faster than SGD because we can utilize things like hardware optimization of matrix operations or numpy's universal functions. In figure 10 we can see a visualization of the descent by all three algorithms. There is a clear difference in randomness for all three, and in how they settle at the minimum.

One important difference in between SGD and MB is the learning rate η. Because of the huge variance in the parameter update for SGD, we generally prefer using a lower learning rate to avoid divergence, and because of this the MB algorithm will generally converge quicker as we can risk taking bigger steps in the descent.
Of course there exists more advanced algorithms for gradient descent than these three, but when when having the option of the three presented here the Mini-Batch is generally the preferred one due to it's speed and reduced variance compared to SGD.

II Implementation and results

5 Implementation and data sets

5.1 Regression of the Ising model in 1D

When approaching the task of doing linear Regression on data from the one dimensional Ising model, we will imagine being handed a data set of N random spin configurations of L spins, with corresponding energies generated by the one dimensional Ising model given by equation 3

$$D = \{\{S_j\}_{i,j=1}^{L}, E_i\}_{i=1}^{N} \tag{56}$$

where the spins $S_j = \pm 1$ and the dataset D consist of $i = N$ spin-energy combinations.
If we did not know how the data was created and what it represents, then our task is the recreate the underlying model which creating the data using linear regression.
A typical physicist's approach might be to evaluate the pairwise interactions of all the variables like one would evaluate the potential energies using Newton's law of gravitation. This would correspond to a all-to-all Ising model.

$$E^i = -\sum_{j=1}^{L}\sum_{k=1}^{L} J_{j,k} S_j^i S_k^i \tag{57}$$

Now we can collect all the $J_{j,k}$ elements in a matrix J containing all the couplings between spin pairs. We can also collect all the spin pairs for a given spin system i in a matrix X^i. We can now calculate the energies of the given spin configuration by matrix multiplication like a typical linear regression problem

$$E^i = X^i J \tag{58}$$

By using linear regression or a neural network we will find the weights of the matrix J which reproduce the data generated by the Ising model.

5.2 Classification of the Ising model in 2D

When considering classification of the Ising model, we are no longer interested in the exact combination of the coupling constants creating a certain energy. We instead aim to classify whether the system is ordered or disordered, i.e if the system is below or above the critical temperature described in section $3.4T_c \approx 2.3T_0$.

Our data set consists of a range of an enseble of spin configurations which are labeled as 0 (disordered) or 1 (ordered). Some configurations are generated on temperatures around the critical temperature, which means these may carry some characteristics from both ordered and disordered states. We will leave these states out of the training procedure of our models, and instead train on the states that are definitely ordered or disordered. The critical configurations will hence act as an "extra" test set which will give it's own measure of how good our model *actually* is. We don't expect our models to perform *perfectly* on this set, but a good model should be able to get a rather good score on this as well.

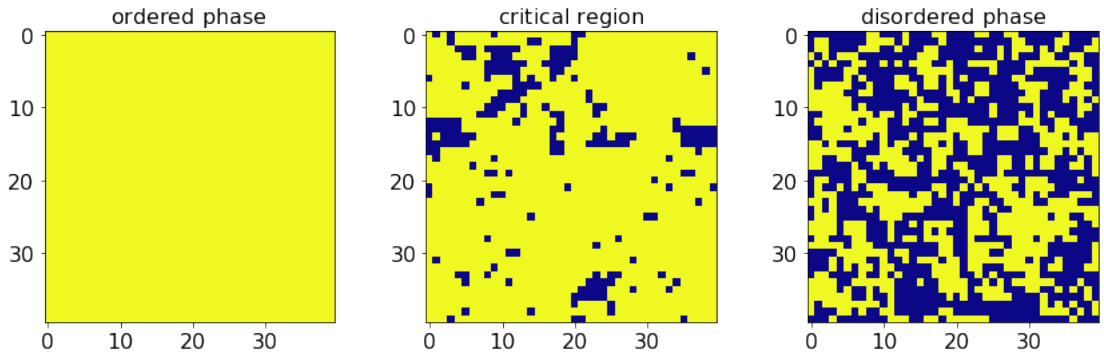


Figure 11: *Examples of what an ordered and disordered spin configuration looks like, as well as one in the critical "fringe" area in between. Source: Mehta et al.[1]*

6 Results

6.1 Linear regression

The first thing we explored was of what degree our linear regression schemes was able to predict the energies of a given spin-spin interaction matrix (i.e. $S_i S_j$), and also the coupling constant matrix $\hat{J}$. We aimed to recreate the results gotten by Mehta et al.[1], where they ran several experiments with different regularization strengths λ, and plotted the resulting $\hat{J}$ matrix ($\hat{\beta}$ in the context of linear regression theory).

Figure 12 shows a few plots generated by our program, and when we compare these with the corresponding plots in Mehta et al.[1], we see they look exactly the same. Table 1 shows a selection of the statistical results of the runs. All results in the table was obtained as averages through 100 bootstrap resamplings. We see that no model is able to describe the data very good at all, but while most models give terrible results, there is a few Lasso models that seem to be on the track of the real description. We see in the table that a Lasso model with $\lambda = 0.01$ gives the best results, which is the same reported by Mehta et al.[1].

Method	λ	MSE Train	MSE Test	R2 Train	R2 Test	Bias	Variance
OLS	0.0	1.232e-05	32.251	1.000	0.186	29.193	3.057
Ridge	0.01	1.023e-09	32.530	1.000	0.179	29.489	3.041
Ridge	0.1	1.017e-07	32.467	1.000	0.180	29.351	3.115
Ridge	1	1.028e-05	32.451	1.000	0.181	29.445	3.006
Ridge	10	0.0010	32.373	0.9997	0.183	29.393	2.980
LASSO	0.0001	1.134e-06	35.232	1.0000	0.1108	27.6153	7.616
LASSO	0.001	0.00011	8.958	0.9999	0.7739	4.3389	4.619
LASSO	0.01	0.00896	4.085	0.9997	0.8968	1.1792	2.906
LASSO	0.1	0.78082	11.884	0.9776	0.700	8.6709	3.213

Table 1: *Selection of results from the linear regression of the 1D Ising model.*

What's to note here is that these experiments were run on randomly generated spin samples with a total sample size of 400. We see that both the OLS and Ridge methods pick up on there being a strong nearest neighbor interaction (the two diagonal lines), but also weaker interactions between all other spin pairs. Since we know how the data was generated by using *only* nearest neighbor interactions (see section 5.1), we know that all these extra interactions don't really describe the data. This is also reflected in Table 1, where the MSEs on the test set are quite awful for all these models. Increasing the number of samples increases the performance for all models considerably, and with 10000 samples all models are sure of nearest neighbor interactions *only*, leading to close to perfect results.

What's really interesting here is that while OLS and Ridge chooses a coupling strength of 0.5 in *both* directions, Lasso fully commits to one direction, with a coupling strength

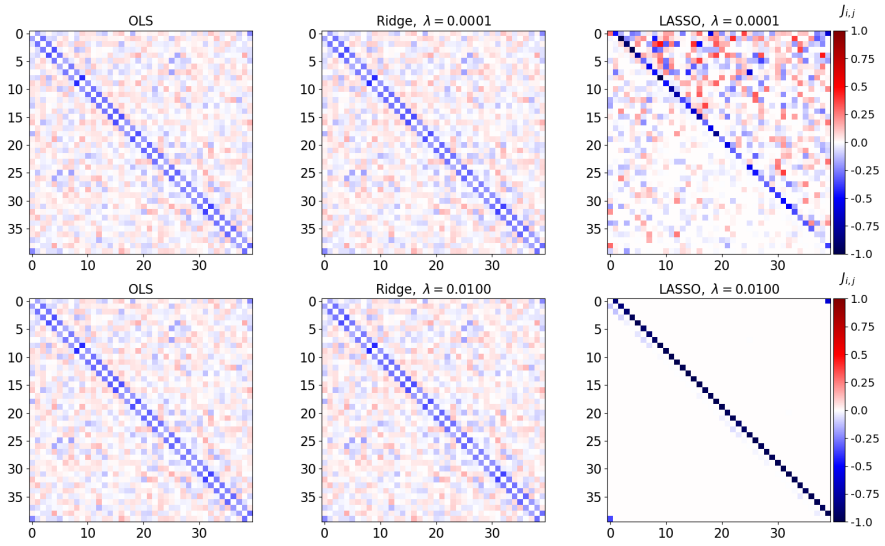


Figure 12: *Reproduced plots from Mehta et.al.[1] of the linear regression results for* $\lambda = 0.0001, 0.01$*, sample size = 400. Note: Increasing maximum number of iterations in Scikit Learn's Lasso-function makes the* $\lambda = 0.0001$ *plot look like the one with* $\lambda = 0.01$*.*

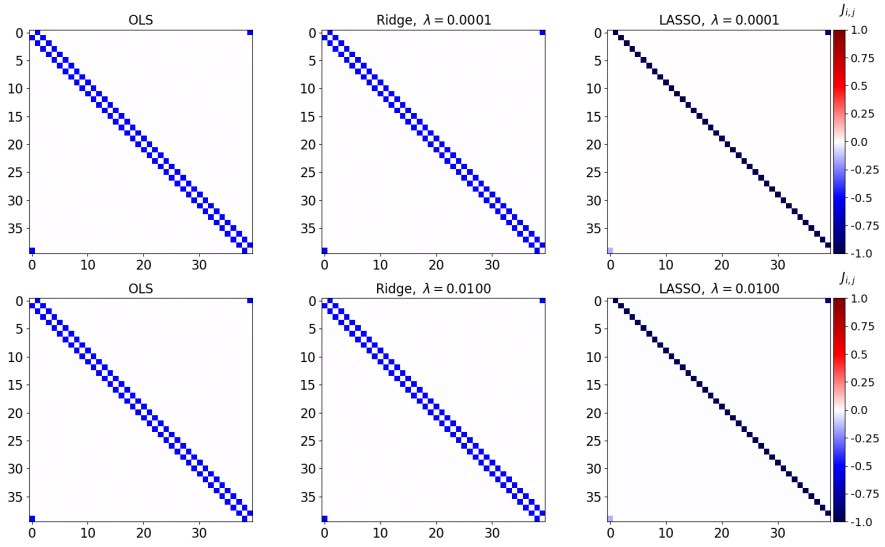


Figure 13: *Same plot as Figure 12, but now with a sample size 10000. While OLS and Ridge chooses a symmetric solution, Lasso picks out interactions in only one direction.*

of 1.0. Since we're using periodic boundary conditions, we can't practically tell if the interactions go one way or both, as they are equivalent under this condition. This means that both solutions are equally good, although they look different. The reason for the Lasso choosing one direction is not entirely clear to us, but it's likely that some other type of minimization algorithm is being used in Scikit Learn's Lasso function that prefers even sparser models than the ones gained in the OLS and Ridge regressions. As we'll see in the results of the neural network (section 6.3.1), the symmetry will stay intact during minimization using gradient descent.

6.2 Logistic regression

There are quite a few hyperparameters to set in logistic regression, and two of them are the batch size in the stochastic gradient descent method, and the learning rate. What's convenient is that these two in some sense go hand in hand, as a larger batch size usually gives smaller averaged gradients, hence requiring a larger learning rate to converge. Thus it's more important to find a good combination of them rather than tuning each one individually. We started off by testing different combinations, evaluating their performance the test and training sets in terms of cost, accuracy, and total calculation time. The results can be seen in Table 2.

Batch Size	L.rate	Train Cost	Test Cost	Train Acc	Test Acc	Wall Time
65000	0.001	2.62765	2.57877	0.642815	0.648431	105.32
65000	0.01	2.56531	2.52125	0.643677	0.648954	106.945
65000	0.1	10.4199	10.4775	0.491031	0.492431	105.839
650	0.001	1.76248	1.79178	0.662231	0.663554	106.239
650	0.01	0.68946	0.733002	0.713169	0.680292	108.185
650	0.1	8.47399	8.59113	0.46	0.443323	108.3
130	0.001	0.675466	0.712758	0.731985	0.690154	104.691
130	0.01	0.977319	1.00466	0.463662	0.432092	107.291
130	0.1	11.6634	11.5802	0.468138	0.456508	103.543
65	0.001	0.677694	0.711747	0.708154	0.666708	109.602
65	0.01	0.692559	0.726628	0.492585	0.460046	108.995
65	0.1	1.34054	1.42947	0.459477	0.435954	109.259
1	0.001	0.743638	0.785637	0.724123	0.698446	145.783
1	0.01	3.9193	4.18961	0.472785	0.460385	145.376
1	0.1	24.04	26.318	0.716492	0.701108	144.745

Table 2: *Results for different combinations of batch size and learning rate. All examples were run with the same weight initialization, and same number of epochs (80). Rows highlighted with green shows good candidates, orange shows "curiosities".*

In this test we have included batch sizes of 65000 (the whole training set in this case) and 1, corresponding to regular batch gradient descent and regular stochastic gradient

descent, as well as some sizes in between. In the table we have highlighted some combinations that are interesting, where the green ones show combinations that perform best. We see that "medium sized" batch sizes seems to perform the best, and this is also supported in many studies, for instance in Yoshua Bengio, 2012[14].

We have also marked a few cases with orange, which peaked our interests for additional reasons. The lower one (batch size 1, learning rate 0.001) shows rather good accuracy compared with the others, while the cost is a little, but not much worse. What's to note however is that the wall time for this batch size is much larger than the other ones, meaning that we're better off picking a faster configuration which perform almost just as well.

The first orange one (batch size 65, learning rate 0.01) shows costs of about the same levels as the green ones, but quite awful classification accuracy. This was an effect which we also observed during training, where the cost kept decreasing, but the accuracy suddenly dropping by 20%, before rising again at later epochs. We never found a good explanation for this behaviour, but it may show that although the cost on average is at a minimum, it may lie in a "fringe" area where it *barely* is able to classify correctly (remember that only a score larger than 0.5 is required to classify as class 0). In this case a small perturbation in the parameters may cause it to pick the opposite class.

It's also worth mentioning that since we only have 2 classes, wild guesses should on average give 50% prediction score. From the table we see than most models actually perform *worse* than this, and the best ones which give around 70% aren't actually that impressive either. This shows that a simple logistic regression model might not be complex enough to capture the characteristics of ordered/disordered spin-configurations, and that "heavier guns" in forms of neural networks may be required.

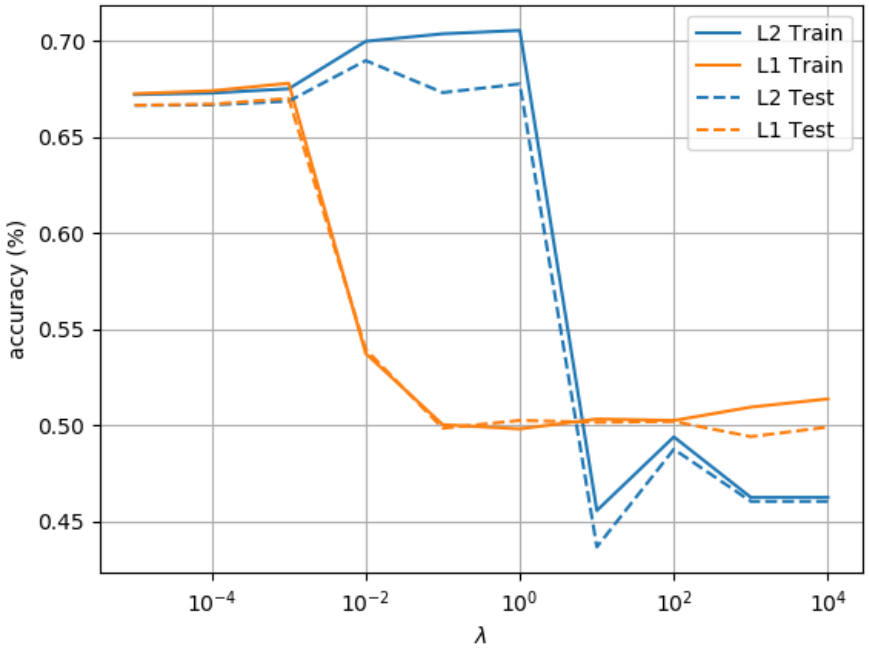


Figure 14: *Classification accuracy as function of regularization strength for both L1 (Lasso) and L2 (Ridge) regularization. All estimations were run with the same weight initializations and hyperparameters.*

Figure 14 shows how the accuracy on the test set react to different levels of regularization. If we again compare with the results on page 32 in Mehta et al.[1] (who only used L2 regularization), we note that the behaviours of the curves are quite similar, although there are some discrepancies for the lowest λs. It seems they in the article chose a number of epochs and decay of the learning rate so these the models aren't given enough time to converge to the same levels of accuracy as those with $\lambda = 10^{-3}, .., 10^{-1}$. We have shown in Table 2 that they should be on the same levels, as these accuracies are all acquired with no regularization. The behaviour for larger λs however is similar in that the accuracies drop down to 45-50% and those values are kept more or less constant as λ continues to increase. This is natural as the all weights at this point will be close to or exactly 0, simply giving 0 as output (and also predicted class label) every time. Given that the true labels of all configurations are around 50-50 split between ordered/unordered, it's fully expected that the models stabilize around this point with such heavy regularization.

From the figure we can also conclude that $\lambda = 10^{-3}$ and $\lambda = 10^{-2}$ seems to be the best choices for L1 and L2 regularizations respectively, as these give the best combinations of performance in terms of accuracy, and least amounts of overfitting.

At last we trained our logistic regression scheme on all the best performing hyperparameters we had found: L2 regularization with $\lambda = 10^{-2}$, batch size 130 and learning

rate $\eta = 0.001$ (this seemed to be the most stable combination). We let the model train for 200 epoch to make sure it was able to converge to the minimum, and measured the performance on the training and test sets, *as well* as the set containing the critical states. We got the following result:

Training Acc (%)	Test Acc (%)	Critical Acc (%)
73.0846	69.7046	59.2167

This indeed seems to be the maximum capacity of this simple logistic regression model, and we see that while the accuracy on training and test aren't great, the accuracy on the critical states are even worse.

6.3 Neural Net

As we know, neural networks are more flexible versions of the linear and logistic regression methods we have explored so far. If we look at how a neural network is built up, we can see that a neural net with no hidden layer and no activation in the output layer simply perform linear regression. The same goes for logistic regression, where a network with no hidden layers but sigmoid/softmax in the output layer does the same job. It's therefor useful to compare our results from our linear and logistic regression methods with our implementation of a neural network to validate that it works for both prediction and classification.

6.3.1 Validation

We start by validating that our neural network gives the same results as our linear regression model in section 6.1.

If we compare this plot with the data in Table 1 we see the same behavior. There's only one or two Lasso models that give somewhat decent results, while all of the other have really high MSEs. A thing to note that the preferred λ value for Lasso in this case is $\lambda = 0.1$, while in section 6.1 we found that the optimal λ to be 0.01. This might be explained by our results for the coupling matrix J shown in Figure 16.

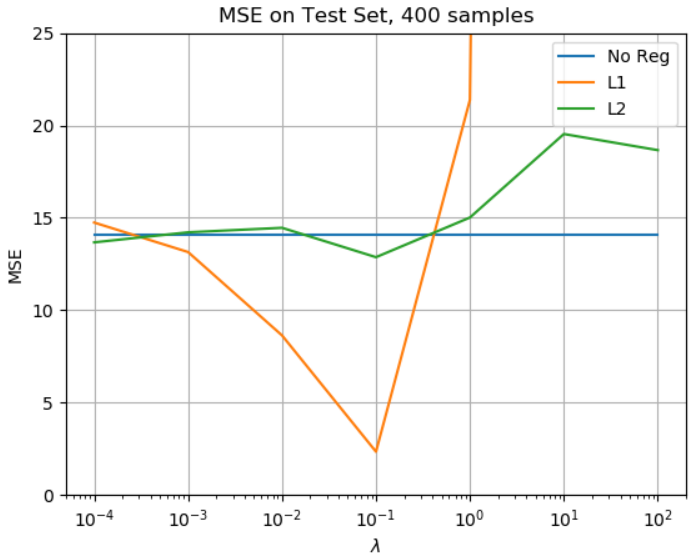


Figure 15: *MSE on the test set as function of regularization strength λ for L1, L2 and no regularization. Note that all of these values differ by a factor of 2 from those obtained in linear regression, simply because we have defined the errors like this.*

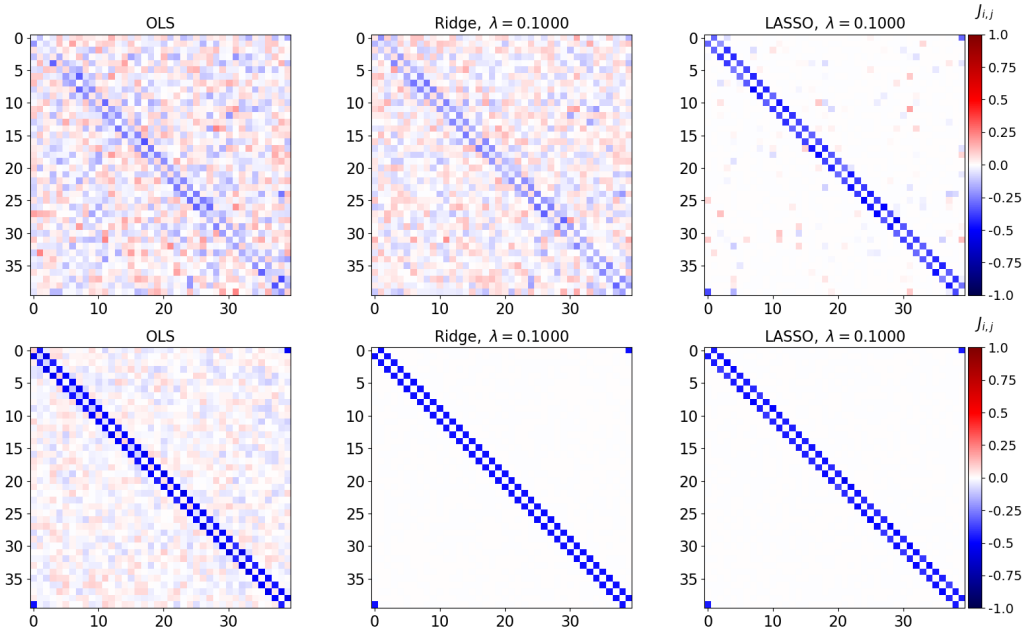


Figure 16: *Coupling constant matrix J, computed by the neural network on 400 (top) and 10000 (bottom) training examples.*

Figure 16 shows the results corresponding to those in Figures 12 13 (linear regression), but now computed with the neural network and $\lambda = 0.1$ for sample sizes of 400 and 10000. We see that the results for OLS and Ridge are quite noisy for lower sample size, and it doesn't seem to be some pattern in the noise like we see in the linear regression case, where the matrix is symmetric around the diagonal. This noise gives quite awful fits, as seen in in both Table 1 and Figure 15. It's interesting to note that while "OLS" and L2 regularization in the NN case are really noisy, L1 actually finds and select the correct interactions with minimal noise, given such a sparse data set. What's different here from linear regression is that L1 regularization here actually *don't* break the symmetry like Scikit Learn's algorithm does. Scikit's documentation states that it uses so-called "coordinate descent" instead of gradient descent, which might be the cause of the differing results. As stated before however, the solutions are equivalent, hence there's no difference in performance.

The bottom row of Figure 16 (10000 samples) is also interesting in that the "OLS" solution isn't noise-free, despite perfect performance. By examination one can see that where linear regression gave a *symmetric* solution, the neural network gives an *anti-symmetric* solution, where all terms on opposite sides of the diagonal effectively cancel each other out. The L1 and L2 solutions are similar with linear regression in that they are able to remove the noise, giving sparser models with effectively less variance.

As we don't have have the geometric/visual interpretations of the weights in the 2D case as we did in 1D, so we will here simply validate that out network perform just as good (or bad) in terms of cost and accuracy. We found just this, where the output of an example run gave the following accuracies.

Training Acc (%)	Test Acc (%)	Critical Acc (%)
71.1286	68.8615	61.4467

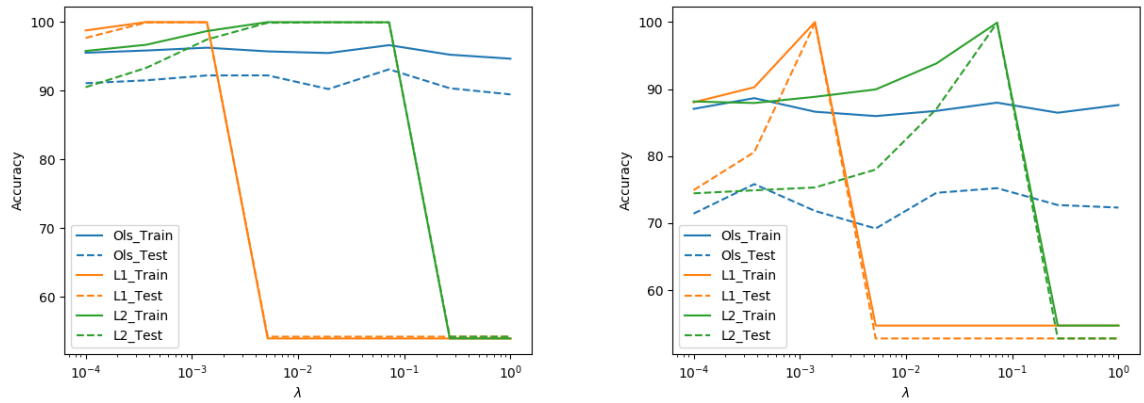
These are similar to the ones found in logistic regression, which both proves that our network works also in the classification case, and that we need more complexity to be able to classify this data set properly.

6.3.2 Quest for the perfect classifier

As we have seen, our simple linear regression model is able to predict the energies of different 1D spin configurations perfectly, which was also validated by the no-hidden-layer network. This means that adding layers or nodes to increase the complexity is meaningless in this case, and one could say that the use of a neural network at all is overkill. In the classification of phases of 2D spin configurations however it's clear that a simple logistic regression model does not suffice, and that we need a more complex model in the form of e.g. a neural network. The goal is to build a model that perfectly classifies all ordered/unordered phases, as well as performing well on the critical configurations.

There are several ways of defining what constitutes the best network.
We might be interested in which network gives the best test accuracy on similar data or which gives the best accuracy on difficult data sets like the critical data.
Do we values speed i.e reaching a certain accuracy threshold the quickest, or is it just the final error we care about no matter the time it takes?
We might value the network reaching a certain accuracy using less training data if we want to speed up our calculations.
For a binary classifier as we are working with in this project we might expect that for a large amount of training data, we will have a very large amount of possible models which all work very well. To better distinguish between the models we have decided the define the "goodness" of the model using only 10000 training examples and mainly evaluating the models on the critical data set. How quickly the minimum is reached has not been an evaluation criteria as long as the minimum is reached within a given amount of epochs.

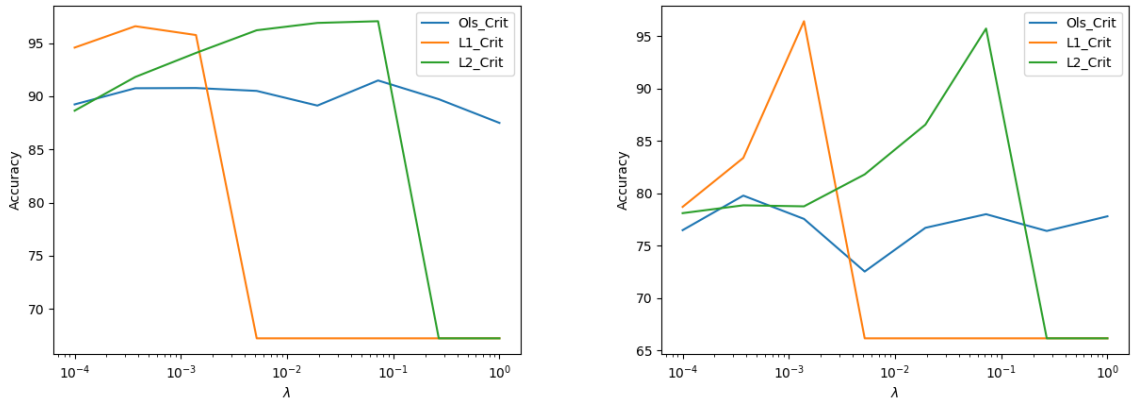
Fine tuning a neural network might seem as an insurmountable task, as the possible number of combinations of parameters are infinite. One thing which we know for certain however, is that as long as the network is not terribly designed, the network will perform better the more training examples it is allowed to see. As an illustration of this we show the accuracy of training test and critical data in Figure 17 and Figure 18. As we can clearly see, increasing the number of data points will in general increase the accuracy of the network. There is however a sweetspot around $\lambda = 10^{-3}$ for L_1 regularization and $\lambda = 10^{-1}$ for L_2 regularization, where 4000 training examples performs just as well as 10000 samples.
It is tempting to think that a this regularization will work just as well for different activation functions and architecture, but there is no guarantee that it will.



(a) *Accuracy of two layer neural network using the sigmoid activation function and 10 nodes in the hidden layer. The neural network was run for 10000 training examples*

(b) *Accuracy of two layer neural network using the sigmoid activation function and 10 nodes in the hidden layer. The neural network was run for 4000 training examples*

Figure 17: *In the figures L1 and L2 refers to the type of regularization, "ols" has been used to reference no regularization. We a clear difference between (a) and (b). As the number of training examples increases, we naturally can expect the test accuracy to increase as long as the regularization is not to extreme*



(a) *Accuracy of two layer neural network using the sigmoid activation function and 10 nodes in the hidden layer. The neural network was run for 10000 training examples*

(b) *Accuracy of two layer neural network using the sigmoid activation function and 10 nodes in the hidden layer. The neural network was run for 4000 training examples*

Figure 18: *In the figures L1 and L2 refers to the type of regularization, "ols" has been used to reference no regularization. We a clear difference between (a) and (b). As the number of training examples increases, we naturally can expect the test accuracy to increase as long as the regularization is not to extreme*

All networks in our evaluation was run for 10000 training examples, learning rate 0.01, batch size 200 and 200 epochs. The network cycling was done in the following way:

- All networks was cycled through 1, 2 and 3 hidden layer architecture
- All layers in a given network has the same number of nodes
- All networks was cycled through 10, 50 and 100 nodes in their given hidden layers
- Only a single type of activation function is used throughout a given network of up to 3 hidden layers
- All networks was cycled through L1, L2 and No regularization
- Every type of regularization was tested for a regularization strengths of 0.001, 0.01 and 0.1

Activation	Critical	λ	N	Reg	Test	Train
tanh	96.643	0.1	100.0	l2	99.944	99.91
tanh	96.437	0.1	50.0	l2	99.939	99.9
sigmoid, sigmoid	96.39	0	100.0	None	99.922	99.96
tanh, tanh	96.37	0.1	100.0	l2	99.938	99.9
tanh	96.23	0.1	10.0	l2	99.932	99.9
tanh, tanh, tanh	96.163	0.001	100.0	l1	99.945	99.91
relu, relu	90.253	0.001	10.0	l1	96.462	99.03
relu, relu	88.803	0.1	100.0	l2	91.48	96.73
relu, relu, relu	87.837	0.1	50.0	l2	94.126	98.19
relu, relu	86.8	0.1	50.0	l2	88.809	95.3
relu, relu, relu	86.19	0.001	100.0	l1	99.844	99.98
sigmoid	79.37	0.001	100.0	l1	77.45	81.7
sigmoid	77.757	0.01	100.0	l2	77.392	82.24
sigmoid	76.82	0.01	50.0	l2	73.605	78.01
sigmoid	76.627	0.001	50.0	l1	73.794	77.22

Table 3: *Best network combinations was ranked by critical accuracy for sigmoid, tanh, relu. For simplicity only one type of activation function was used throughout a single network and maximum depth was 3 hidden layers. The 5 best networks for each activation function combination has been combined to a single table and again sorted by critical accuracy. Hidden layers throughout a single network all had the same size and was tested for 10, 50 and 100 nodes in each layer. L1, L2 and No regularization was tested with regularization strengths of 0.001, 0.01 and 0.1. All networks was trained on 10000 samples.*

The 5 best results for each activation function architecture has been sorted by critical error in table 3. Activation shows the hidden layer structure, λ is the regularization

strength, N the number of nodes in each hidden layer, Reg is the type of regularization used, Critical, Test and Train are the accuracies of the network.
As we can clearly see, different combinations of the tanh activation function are all performing very well on the critical data set. The biggest difference between tanh and the others is the fact that tanh is zero centered, which seems to have a very big impact on the dataset.
We also see that for a small data set sigmoid architectures are generally very bad, especially combining several layers of sigmoid activations are a terrible idea as one can see by expecting the full data set in the github repository. However the sigmoid architecture marked in orange stands out, and it it not entirely clear why exactly this architecture performs so well. In figure 19 we can see the convergence rates of this architecture is very slow and it fluctuates badly between the epochs which would make it a bad candidate for our best model.
We can also see that using some kind of regularization is in general a good idea and L_2 regularization seems to have slight advantage over L_1.
Another observation is that in general we get better results by increasing the number of nodes, as we might expect from the universal approximation theorem.
As we can see from the table, the top 6 accuracies are almost indistinguishable and either one will be a good model for us. We have chosen the model marked in green as our best model. The reasoning behind this is that given a number of models which all perform equally well, we prefer a simple model as it is less prone to overfitting. This model only uses a single hidden layer of 10 nodes, which in addition makes it very quick to compute compared to the other models which we can see in figure 20. However, if we compare with figure 19 we can see that our choice of best model converges at about half the rate of the deeper layer tanh architectures, hence the speedup will be negligible.

Finally we ran our best model on the full dataset, giving the results:

Training Acc (%)	Test Acc (%)	Critical Acc (%)
99.9954	99.9708	96.78

Some models we tried out were in some cases able to give 100% on training and/or test, but they usually performed worse on the critical set. With the critical accuracy taken into consideration, the results above are the overall best ones we were able to get.

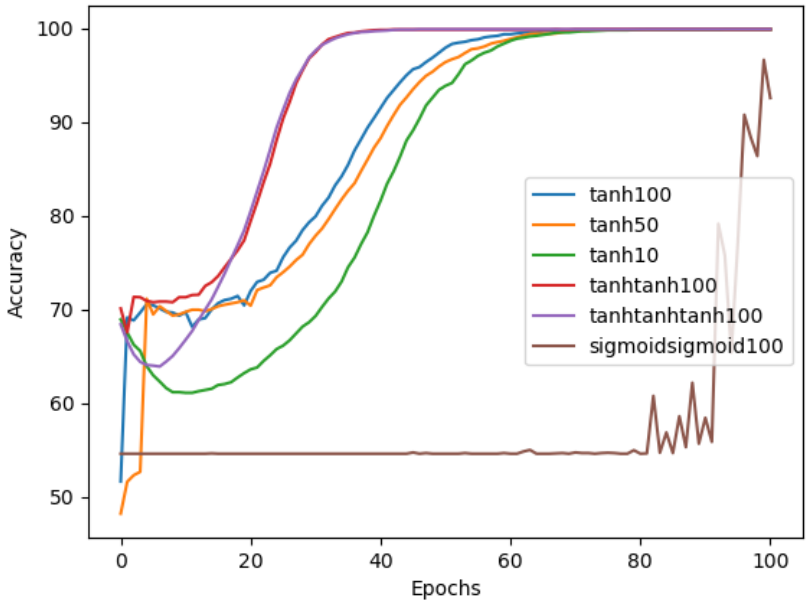


Figure 19: *Convergence rate of the six best models in table 3. The y axis is measured in the accuracy of the test error*

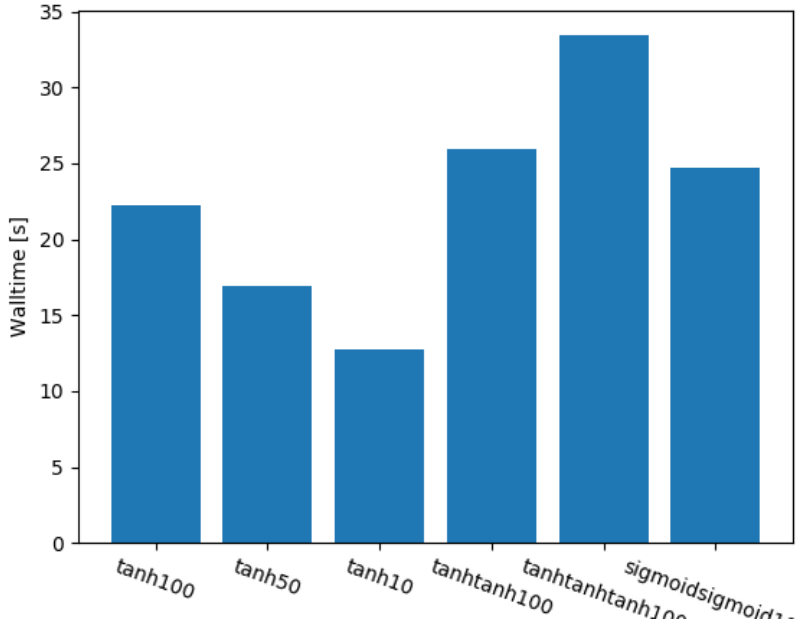


Figure 20: *Wall time of the 6 best models measured in seconds and run for 100 epochs.*

III Conclusions and future work

7 Summary and conclusion

In this project we have tested Ordinary Least Squares, Ridge Regression and Lasso Regression as methods for reproducing the energies given by the one dimensional Ising model. We found that for such a simple problem, all methods were able to find a model which perfectly reproduces the energies in the data set when given enough training examples. However, Ordinary Least Squares and Ridge did not break the symmetries of the nearest neighbors, which Lasso did. This seems to be a feature of Scikit learns minimization algorithm and not of Lasso regression in it self, as the symmetries was not broken when performing Lasso regression using a neural network. Both Scikits Lasso algorithm and our "Lasso neural network" did however vastly outperform OLS and Ridge for few training samples, as it found the nearest neighbor dependency almost perfectly for only 400 training samples.
As with the standard regression algorithms the neural network was also able to perfectly reproduce the Ising model when given enough training samples. One could argue however that using a neural network is a far too advanced method for such a simple problem.

We then used logistic regression for classifying the ordered and unordered states of the two dimensional Ising model. We found that logistic regression is not complex enough for this problem and was not able to classify the states to a satisfactory degree, where our best accuracy was 69.7 % evaluated on the Test set, and 59.2 % when evaluated on the critical data set.
A neural network of only one hidden layer was able to make nearly perfect predictions on the test set and an accuracy in the high 90s for the critical data set. Of the neural network hyperparameters and architectures we tested, we chose a simple network of one hidden layer with a tanh activation function as our best model which give a test accuracy of approximately 96 % on the critical data set.
In our studies we found an enormous amount of network architectures which all gave accuracies in the high 90s, and any one of these could have been used as a model without the result being much different. Hence we might conclude that a perfect feed-forward neural network might not exist for this problem, but there are a lot of networks which are "close enough".

8 Thoughts for future

We got very good results for a lot of architectures in this project. Even though most architectures which was showed enough training data could be tweaked to get approximately 96 percent accuracy on the critical test set, we never seemed to get above the 97 percent threshold. It's possible that the last states are those generated at temperatures very close to the critical temperature, leaving it almost impossible to tell if they are or-

dered or disordered just by looking at the spins. There are several things we could have tried to improve the performance further however:

When starting off we did not expect to be using as much as three hidden layers in our neural network architecture and because of this we did not implement a initiation scheme such as He initiation. As discussed in the theory, the variance within the layers gets very important when the networks get deep. This is probably more important for the convergence rate than for the total accuracy though.
We also did not implement different minimization methods such as momentum and Adam, and this is probably a very good starting point for increasing the accuracy.

Adding convolutional layers to our feed-forward neural network (or even just a pure convolutional neural network) might be able to push the accuracy the final distance, but we'll leave this for a possible future project.

V References

References

[1] Mehta et al, arXiv 1803.08823, *A high-bias, low-variance introduction to Machine Learning for physicists*, arXiv 1803.08823.

[2] Thomas C. Schelling. Journal of Mathematical Sociology. Dynamic Models of Segregation. Harvard University. Dynamic Models of Segregation

[3] Wikipedia, Canonical ensemble, 30.October 2018, Canonical Ensemble

[4] Wikipedia, Lattice(group), 30.October 2018, Lattice

[5] Fifty Years of the Exact solution of the Two-dimensional Ising Model by Onsager, Somendra M. Bhattacharjee and Avinash Khare, Institute of Physics, Sachivalaya Marg, Bhubaneswar 751 005, India. (February 1, 2008)

[6] FYS-STK4155 Morten Hjort-Jenssen, Department of Physics, University of Oslo, Norway, Oct 16, 2018

[7] CS231n - Convolutional Neural Networks for Viasual Recognision, Module 1: Neural Networks, Stanford, 2018

[8] Krizhevsky et al., *ImageNet Classification with Deep Convolutional Neural Networks*

[9] FYS-STK4155 - Project 1 Tommy Myrvik, Kristian Tuv, 2018, University of Oslo

[10] Approximation Capabilities of Multilayer Feedforward Networks, Kurt Hornik, Technische Universitiit Wien, Vienna, Austria, 1990

[11] Recurrent Neural Network, Wikipedia, 5. november 2018

[12] Understanding the difficulty of training deep feedforward neural networks, Xavier Glorot, Yoshua Bengio DIRO, Universite de Montreal, Montreal, Quebec, Canada

[13] Efficient Backprop Yann LeCun, Leon Bottou, Genevieve B Orr, Klaus-Robert Müller, Image Processing Research Deartments At and T labs, Springer, 1998

[14] Yoshua Bengio, arXiv 1206.5533, *Practical recommendations for gradient-based training of deep architectures*, 2012