Machine learning applied to the one- and two dimensional Ising model.

FYS-STK4155: Project 2

Morten Ledum & Håkon Kristiansen
github.com/mortele/FYS-STK4155

November 8, 2018

Abstract

Contents

INTRODUCTION

There are many problems that require a probability estimate as output. This could for example predicting whether a person will develop a specific disease given genetic information. Another example, which we will examine closer, is to predict if a given spin-configuration generated from the two-dimension Ising model is ordrered or disordered. Problems of this type are referred to as *classification* problems.

Classification is fundamentally different from the regression problems we studied previously, in the sense that the predicted outcome only takes values across discrete categories. Thus, we will need different tools than that of linear regression. In this work we first consider *logistic regression* as a method for classification.

Artificial neural networks (ANN or simply NN) are essentially ubiquitous in modern technology today. Whenever you use a computer—whether you are using Youtube, searching Google, exchanging currency in a bank, interacting with a virtual assistant (such as Amazon's *Alexa*, Google's *Google assitant*, or Apple's *Siri*), editing photos, etc.—you are most likely interacting with a neural network. As a subfield of AI and machine learning research, NNs represent models which can learn to predict outcomes of new input data, by being repeatedly shown series of input/output pairs to *learn* from. Individual network models fall under the category of *narrow AI*, as each model is only able to do one (or a few) highly specialized tasks it was designed for. In the past few decades, such narrow AI NN models have reached super-human performance in a wide range of applications, e.g. board games (e.g. chess and go), visual pattern recognition (e.g. traffic sign recognition), parsing handwritten text, etc.

We will use NNs for both regression and classification analysis in the present project. Unlike for the linear models, the fundamental structure of the model and the training remains the same for NNs when interchanging regression $\rightleftharpoons$ classification, and the only change needed is exchanging the cost function employed in the training.

Our test system for this project is the Ising model, invented in 1920 by German physicist Wilhelm Lenz and solved (the 1D case) in 1925 by Ernst Ising.[Ising1925] The much more interesting (and computationally much more challenging) two-dimensional version was not solved until Lars Onsager tackled the problem in 1944.[onsager1944crystal] The Ising model—a lattice of spins with a local, nearest neighbors, interaction energy—is a simple but enormously important model system in theoretical physics as it is the first and simplest statistical mechanics model system exhibiting a phase transition which may be solved analytically.[mccoy2012importance] We will use regression and classification analysis to estimate the interaction energy parameter and the total energy, and classifying sub-critical ordered and supercritical disordered states of the system.

THEORY

In the following we outline the theory of the present work. We consider logistic regression as a model for classification problems. Furthermore, neural networks are discussed both in the context of regression analysis and classification. The theoretical aspects of linear regression have been discussed in previous work and is not repeated here.

In contrast to the linear regression model, we can not find the optimal parameters of the logistic or neural network models analytically. Thus, we have ro rely on numerical methods for optimization. In particular we will give a brief summary gradient descent methods.

Logistic regression

Suppose that we are given a dataset $\{(\mathbf{x}^{(i)}, y_i)\}_{i=1}^n$ where we have p predictors for each data sample $\mathbf{x}^{(i)} = \{x_1^{(i)}, \cdots, x_p^{(i)}\}$. The responses/outcomes y_i are discrete and can only take values from $k = 0, 1, \ldots, K-1$ (i.e K classes). The goal is to predict the output classes given n samples each containing p predictors. Throughout this section we assume that there are just two possible outcomes, i.e $y_i \in \{0, 1\}$.

In logistic regression, in contrast to linear regressions, we model the *probability* that y_i belongs to class 1, given $\mathbf{x}^{(i)}$. Let $p(y|x)$ denote the probability of event y given x, then the *logistic model* is

$$p\left(y=1|\mathbf{x};\beta\right) = \frac{1}{1+e^{-\beta\cdot\mathbf{x}}} \tag{1}$$

$$p\left(y=0|\mathbf{x};\beta\right) = 1 - p\left(y=1|\mathbf{x};\beta\right). \tag{2}$$

Here $\beta = (\beta_0, \beta_1, \ldots, \beta_p)$ are the parameters of the model. Note the appearance of the intercept

term β_0. In order to keep notation compact $\mathbf{x}^{(i)}$ can be augmented to incorporate the intercept by adding a 1 to each sample, i.e

$$\mathbf{x}^{(i)} \to \{1, x_1^{(i)}, \ldots, x_p^{(i)}\}.$$

The term $\beta \cdot \mathbf{x} = \beta_0 + \sum_{k=1}^{p} \beta_k x_k$ is known as the *log-odds* and the function

$$\sigma(\beta \cdot \mathbf{x}) = \frac{1}{1 + e^{-\beta \cdot \mathbf{x}}} \tag{3}$$

is called the *sigmoid* of $\beta \cdot \mathbf{x}$.

The logistic model can now be used for classification using the estimated probabilities

$$\hat{y}_i = \begin{cases} 1 & \text{if } p\left(y = 1 | \mathbf{x}^{(i)}\right) \geq 0.5 \\ 0 & \text{if } p\left(y = 1 | \mathbf{x}^{(i)}\right) < 0.5. \end{cases} \tag{4}$$

Training the logistic model

How do we train the logistic model? The answer is to use the prinicple of *maximum likelihood*. Under the assumption that every sample $\mathbf{x}^{(i)}$ is indenpendent, the likelihood is given by

$$\begin{aligned} L(\beta) &= \prod_{i:y_i=1} p(y_i = 1 | \mathbf{x}^{(i)}) \prod_{i:y_i=0} p(y_i = 0 | \mathbf{x}^{(i)}) \\ &= \prod_{i=1}^{n} p(y_i = 1 | \mathbf{x}^{(i)})^{y_i} (1 - p(y_i = 1 | \mathbf{x}^{(i)}))^{1-y_i} \\ &= \prod_{i=1}^{n} \sigma(\beta \cdot \mathbf{x}^{(i)})^{y_i} (1 - \sigma(\beta \cdot \mathbf{x}^{(i)}))^{1-y_i} \end{aligned} \tag{5}$$

Then, the parameters β are chosen to maximize the likelihood.

It turns out that it is easier to work with the *log-likelihood*

$$\begin{aligned} l(\beta) &= \log(L(\beta)) \\ &= \sum_{i=1}^{n} \Big[y_i \sigma(\beta \cdot \mathbf{x}^{(i)}) \\ &\qquad + (1 - y_i)(1 - \sigma(\beta \cdot \mathbf{x}^{(i)})) \Big]. \end{aligned}$$

Maximizing the logarithm of a function is equivalent to maximizing the function itself.

In order to see this, let $f(x)$ be a real valued function and let x^* be a maximum point of $f(x)$, i.e

$$f'(x^*) = 0, \quad f''(x^*) < 0. \tag{6}$$

Furthermore, assume that $f(x) > 0$ and consider $\log(f(x))$. Taking derivatives we have that

$$\frac{d}{dx} \log(f(x)) = \frac{f'(x)}{f(x)} \tag{7}$$

$$\Rightarrow \frac{d}{dx} \log(f(x^*)) = 0 \tag{8}$$

$$\frac{d^2}{dx^2} \log(f(x)) = \frac{f''(x)f(x) - f'(x)^2}{f(x)^2} \tag{9}$$

$$\Rightarrow \frac{d^2}{dx^2} \log(f(x^*)) < 0, \tag{10}$$

where the last inequality follows from the fact that we assumed $f'(x^*) = 0$, $f''(x^*) < 0$ and $f(x) > 0$. Hence, x^* also maximize $\log(f(x))$.

Thus, taking β to maximize the log-likelihood is equivalent to maximizing the likelihood itself. Finally, we take our cost function to be the so-called *cross-entropy* which is defined as the negative log-likelihood

$$C(\beta) \equiv -l(\beta). \tag{11}$$

Then, β is found by *minimizing* the cross-entropy.

Note here that we can not find a analytical solution for the maximizer. This means that we have to use a numerical opimization algorithm, such as gradient descent which we discuss later, to find the optimal parameters.

The gradient of the cross-entropy can be given in closed-form in terms of its components

$$\frac{\partial}{\partial \beta_j} C(\beta) = -\sum_{i=1}^{n} \mathbf{x}_j^{(i)} \left(y^{(i)} - \sigma(\beta \cdot \mathbf{x}^{(i)}) \right) \tag{12}$$

or more compactly as

$$\nabla_\beta C(\beta) = -X^T (\mathbf{y} - \mathbf{p}). \tag{13}$$

Here we have defined

$$\mathbf{y} \equiv (y_1, \ldots, y_n), \tag{14}$$

$$\mathbf{p} \equiv \left(\sigma(\beta \cdot \mathbf{x}^{(1)}), \ldots, \sigma(\beta \cdot \mathbf{x}^{(n)}) \right) \tag{15}$$

and $X \in \mathbb{R}^{n \times (p+1)}$ is the design-matrix containing $\mathbf{x}^{(i)}$ as its i-th row.

Convexity of the cross-entropy

It is well known that any local minimum of a convex function is also a global minimum. This has the important consequence that if a minimum is found we are sure that the solution is

optimal. For a multivariate function convexity is guaranteed if the corresponding Hessian matrix of second partial derivatives is positive semidefinite.

We show here that the cross-entropy (11) is a convex function. In the following we write $p_i \equiv \sigma(\beta \cdot \mathbf{x}^{(i)})$.

The components of the Hessian is

$$\begin{aligned}\frac{\partial^2 C(\beta)}{\partial\beta_k\partial\beta_j} &= -\sum_{i=1}^{n} \mathbf{x}_j^{(i)} \frac{\partial}{\partial\beta_k}\left(y^{(i)} - p_i\right) \\ &= -\sum_{i=1}^{n} \mathbf{x}_j^{(i)}\mathbf{x}_k^{(i)} p_i(p_i - 1).\end{aligned}$$

Written on matrix form we have that the Hessian $\nabla_\beta^2 C(\beta)$ is given by

$$\nabla_\beta^2 C(\beta) = -\sum_{i=1}^{n} \mathbf{x}^{(i)}(\mathbf{x}^{(i)})^T p_i(p_i - 1). \quad (16)$$

Note here that by $\mathbf{x}^{(i)}(\mathbf{x}^{(i)})^T$ we refer to the *outer product* of the i-th row of the design matrix X.

Now, a matrix $A \in \mathbb{R}^{n\times n}$ is positive semidefinite if $\mathbf{z}^T Az \geq 0$ for all $\mathbf{z} \in \mathbb{R}^n$. In particular, we have

$$\begin{aligned}\mathbf{z}^T\nabla_\beta^2 C(\beta)\mathbf{z} &= -\sum_{i=1}^{n} \mathbf{z}^T\mathbf{x}^{(i)}(\mathbf{x}^{(i)})^T\mathbf{z} p_i(p_i - 1) \\ &= \sum_{i=1}^{n} ||(\mathbf{x}^{(i)})^T\mathbf{z}||^2 p_i(1 - p_i) \geq 0,\end{aligned}$$

where the last inequality follows since we have a sum of non-negative terms. Thus, the Hessian of the cross-entropy is positive semi-definite which shows that the cross-entropy is a convex function.

Neural networks [1]

Artificial neural networks (sometimes just neural networks) are computational models with the ability to *learn* from examples it is shown. The structure of the networks are inspired by biological networks constituting animal brains. Artificial neural networks fall under the category of machine learning—a subfield of artificial intelligence—and we will, in the following, expose the precise mechanism of the model learning.

[1] This section follows chapter 7 of [**ledum2017computational**] because I am lazy.

Such neural networks can be created in numerous ways, but we will focus exclusively on the most common architecture, namely *multilayer perceptrons* (MLP). The MLP neural networks are built from *layers* of connected *neurons*. In the artificial network, an input value (possibly a vector) is fed into the network model and then propagated through the layers, being processed through each neuron in turn. We will deal only with *feed forward* ANNs, meaning information always flows through the net in one direction only—essentially there are no loops. The entire ANN produces an output value (possibly a vector), which means we can think of it as a complicated function $\mathbb{R}^n \mapsto \mathbb{R}^m$. As we will see, it is possible to write down a closed form expression for this function and it is—crucially—possible to devise an efficient algorithm for calculating the gradient of the entire function w.r.t. any of the internal parameters.

Neurons and layers

A neuron is simply a model function for propagating information through the network. Inspired by biological neurons, the artificial neuron "fires" if it is stimulated by a sufficiently strong signal. The artificial neuron receives a vector of input values $\mathbf{p}$. If the neuron is part of the very first hidden layer (this will be expanded upon shortly), the input is simply the input value(s) to the NN. If one or more layers preceded the current one, $\mathbf{p}$ is a vector of outputs from the neurons in the previous layer.

The neuron is connected to the previous layers' neurons, and the strength of the connection is represented by a vector of weights, $\mathbf{w}$. Let us now consider a neuron which we will label by the index k. The output from neuron i (of the preceding layer), p_i, is multiplied by the weight corresponding to the i—k connection, w_i. The combined weight vector multiplied by the input vector gives part of the total activation of the neuron,

$$\sum_{i=1}^{N} w_i p_i = \mathbf{w}^T\mathbf{p}. \quad (17)$$

The remaining part is known as the bias, b_k. This is a single real number. There is one for each neuron, and it acts as modifier making the neuron more or less likely to fire independently of the input.

The total input is passed to an activation (or transfer) function, which transforms it in some

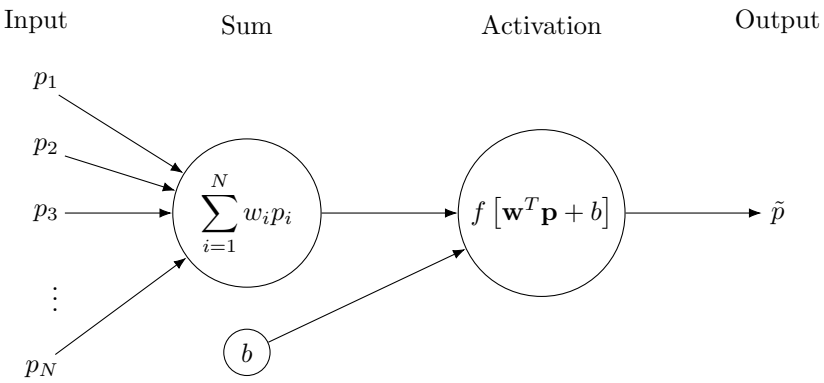

FIG. 1. A model neuron, a constituent part of the artificial neural network model. The input from the previous layer $\mathbf{p}$ multiplied by corresponding weights $\mathbf{w}$ and summed. Then the bias b is added, and the activation function f is applied to the resulting $\mathbf{w}^T\mathbf{p}+b$. The output $\tilde{p}$ goes on to become input for neurons in the next layer.

specified way, yielding the neuron *output* $\hat{p}_k$. This in turn becomes input for the neurons in subsequent layers.

Various different activation functions f are used for different purposes. The function may be linear or non-linear, but should vanish for small inputs and *saturate* for large inputs. For reasons that will become clear shortly, the conditions we enforce on f is continuity, boundedness, as well as non-constantness. We also demand it be monotonically increasing. Numerous different transfer functions are in popular use today, and we will outline some of them in section C.

In total, the action of a single neuron can be written

$$\text{input } \rightarrow f\left(\mathbf{w}^T\mathbf{p}+b\right)=\tilde{p} \rightarrow \text{ output.} \quad (18)$$

A schematic representation of the single neuron connected to the previous and acting as input for the next layers is shown in Fig. 1.

The full artificial neural network is built up of layers of neurons. Data is fed sequentially through the network, starting in the input layer (the input values can be thought of as the first layer), through the *hidden* layers, and ending up in the output layer. The propagation needs to happen simultaneously across the network, as layer k needs the fully computed output of layer $k-1$ before the activations can be calculated.

A layer is—put simply—a collection of neurons, all of which are connected to the previous layer's neurons and the next layer's neurons. Let us label the individual neurons in layer k by index i, i.e. n_i^k. The bias of neuron i is then denoted b_i^k, and the weights connecting n_i^{k-1} to n_j^k is called w_{ji}. For each neuron there is a corresponding weight, so the weight vector is denoted $\boldsymbol{w}_i^k$. The combination of all weight vectors for layer k thus makes a matrix, which we will denote by a capital W^k,

$$W^k=\begin{pmatrix} w_{11}^k & w_{12}^k & w_{13}^k & \dots & w_{1N}^k \\ w_{21}^k & w_{22}^k & w_{23}^k & \dots & w_{2N}^k \\ w_{31}^k & w_{32}^k & w_{33}^k & \dots & w_{3N}^k \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{N1}^k & w_{N2}^k & w_{N3}^k & \dots & w_{NN}^k \end{pmatrix},$$

or more compactly $(W^k)_{ij}=w_{ij}^k$. The collection of all biases for layer k is $\mathbf{b}^k$. In this notation, we may write the propagation of the signal from layer $k-1$ to layer k as

$$\mathbf{y}^k=f(W^k\mathbf{y}^{k-1}+\mathbf{b}^k)=f\left(\begin{bmatrix} w_{11}^k & w_{12}^k & \dots & w_{1N}^k \\ w_{21}^k & w_{22}^k & \dots & w_{2N}^k \\ \vdots & \vdots & \ddots & \vdots \\ w_{N1}^k & w_{N2}^k & \dots & w_{NN}^k \end{bmatrix}\begin{bmatrix} y_1^{k-1} \\ y_2^{k-1} \\ \vdots \\ y_N^{k-1} \end{bmatrix}+\begin{bmatrix} b_1^k \\ b_2^k \\ \vdots \\ y_N^k \end{bmatrix}\right) \quad (19)$$

or in Einstein notation (no sum over k implied)

$$y_i^k = f\big(w_{ij}^k y_j^{k-1} + b_i^k\big). \tag{20}$$

In all of the preceeding three equations, application of f indicates *element wise* functional evaluation.

It is clear from Eq. (19) that propagation through an entire layer can be thought of as a matrix-vector product, a vector-vector summation, and a subsequent application of the transfer function f element-wise on the resulting vector.

A schematic representation of a layer consisting of three artificial neurons in a fully connected ANN is shown in Fig. 2.

The full network

A collection of L layers connected to eachother forms a full *network*. Note carefully that the network is nothing more than a (somewhat complex) function. If a single input and a single output value is specified, the action of the NN can be written out in closed form as

$$\hat{y} = \sum_{j=1}^{M} w_{1j}^L f_L\left(\sum_{k=1}^{M} w_{jk}^{L-1} f_{L-1}\left(\sum_{i=1}^{M} w_{ki}^{L-2} f_{L-2}\left(\ldots f_1\Big(w_{m1}^1 x_1 + b_m^1\Big)\ldots\right) + b_i^{L-2}\right) + b_k^{L-1}\right) + b_1^L\,. \tag{21}$$

Here, we have taken each layer to consist of M neurons. The scalar x_1 denotes the input value, while $\hat{y}$ is the NN output. The transfer functions (which are *not* assumed to all be the same) are denoted $f_L, f_{L-1}, \ldots, f_1$. From looking at Eq. (21), the usefulness of the model is in no way obvious. But it turns out that for an ANN with at least one hidden layer populated with a finite number of neurons is a *universal approximator*.[HORNIK1989359] This holds under the aforementioned assumptions on f. Being a universal approximator means (in this context) that the NN function can be made to be arbitrarily close to any continuous Borel-measurable function (essentially *any* function we are likely to encounter).[mcdonald]

Activation functions

Without any transfer functions, i.e. $f_l(x) = x$ for all layers l, the full network would simply be a linear transformation. In order to introduce non-linearities in our model, we employ one (or more) of a large set of possible activations. As mentioned, we require that these functions are continous and at least once (almost everywhere[2]) differentiable, in order for the backpropagation scheme of section E to work.

The following is an incomplete outline of activation functions in common use today. The simplest possible activation, the identity transformation $f_{\mathrm{I}}(x) = x$, is commonly used for the output layer in regression networks. A simple (Heaviside) step function, $f_{\mathrm{H}}(x) = H(x)$, with

$$f_{\mathrm{H}}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}, \tag{22}$$

is sometimes used in the output layer of classification networks, but seldom in hidden layers because of the vanishing gradient making it impossible to train with backpropagation. The sigmoid function,

$$f_{\mathrm{S}}(x) = \frac{1}{1 + \mathrm{e}^{-x}}, \tag{23}$$

is commonly used as hidden layer activations along with its sibling the hyperbolic tangent

$$f_{\mathrm{t}}(x) = \tanh(x) = \frac{\mathrm{e}^{x} - \mathrm{e}^{-x}}{\mathrm{e}^{x} + \mathrm{e}^{-x}}. \tag{24}$$

The tangent activation is a simple rescaling of the sigmoid, with $f_{\mathrm{t}}(x) = 2f_{\mathrm{s}}(2x) - 1$.

Simpler than the sigmoid, the family of activations known as *rectifiers* consist of piecewise linear functions which are popular nowadays. The basic variant, the rectified linear unit (ReLU) is defined as

$$f_{\mathrm{ReLU}}(x) = \max(0, x), \tag{25}$$

and is popular mostly because of its application to training *deep* (many, large layers) neural networks.[glorot2011deep] Many variants of the ReLU

[2] A function with a property *almost everywhere* (a.e.) denotes a function which satisfies this property everywhere, **except** possibly on a set of measure zero (such as e.g. a single point).

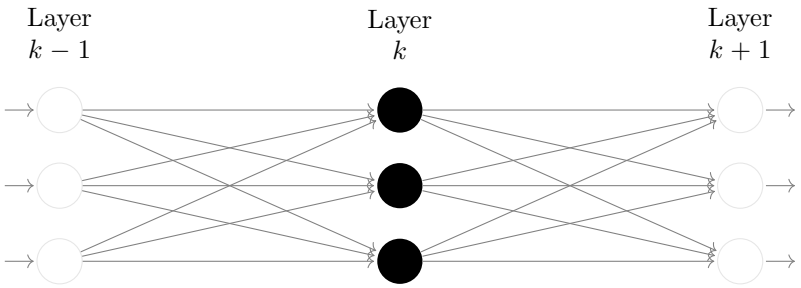

FIG. 2. Schematic representation of a single ANN layer. Each neuron of the layer indexed k is connected from behind to all neurons in layer $k-1$. The connection weights can be organized into a matrix, W^{k-1}, and the action of layer k can be succinctly stated as $f(W^k \mathbf{p}^{k-1} + \mathbf{b}^k)$ where element-wise operation is assumed for the activation f.

exist, among the most well known are the leaky ReLU,

$$f_{\text{leaky ReLU}}(x;\alpha) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0, \end{cases} \tag{26}$$

the noisy ReLU

$$f_{\text{NReLU}}(x) = \max(0, x + \mathcal{N}(0,\sigma)), \tag{27}$$

and the exponential linear unit

$$f_{\text{ELU}}(x;\alpha) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(\mathrm{e}^x - 1) & \text{if } x < 0. \end{cases} \tag{28}$$

Training neural networks

Knowing that ANNs can be universal approximators is not helpful unless we can find a systematic way of obtaining suitable parameters to approximate any given function $g(x)$. This is where *training* comes in. Teaching a NN to approximate a function is conceptually simple, and involves only three steps:

Assume input x and corresponding *correct* output y is known.

(1) Compute output $\text{NN}(x) = \hat{y}$ of the artificial neural network, and evaluate the *cost* function, $C(\hat{y}, y)$.

(2) Compute the gradient of $C(\hat{y})$ w.r.t. all the parameters of the network, w_{ij}^k and b_j^k.

(3) Adjust the parameters according to the gradients, yielding a better estimate $\hat{y}$ of the true output value y.

(4) Repeat (1)—(4).

The training scheme is known as *supervised learning*, because the NN is continually presented with x, y pairs, i.e. an input and an expected output. The cost (or objective or loss) function determines how happy the network is with it's own performance. In general, the output of the neural network is a vector of values, $\mathbf{y}$, and the cost function is taken across all outputs. In Eq. (**??**), the network produces N_{O} outputs for each input (which itself may be a vector).

Step (3) is easy to understand, but complex in practice. In order to update the network weights and biases, a measure of the expected change in the total output is needed. Otherwise, any change would just be done at random[3]. This means we need to compute the set of derivatives

$$g_{ij}^k \equiv \frac{\partial C(\hat{\mathbf{y}})}{\partial w_{ij}^k}, \quad \text{and} \quad h_i^k \equiv \frac{\partial C(\hat{\mathbf{y}})}{\partial b_i^k}. \tag{29}$$

The most common algorithm for computing these derivatives is the **backpropagation** algorithm.[backprop] The method works by first pushing an input through the ANN, and computing the derivatives of the cost function w.r.t. the last layer weights and biases. The network is then traversed backwards, and the gradient w.r.t. all neuron parameters is found by repeated application of the chain rule.

[3] This is a possible approach, yielding a class of *genetic* optimization algorithms. We will not discuss such schemes in the present work.

Backpropagation

Before we can apply the backpropagation algorithm, we need to perform a forward pass of the network given some input vector $\mathbf{x} \in \mathbb{R}^{n_\text{f}}$, where n_f denotes the number of features in the input data. We consider—for the moment—the case of a single input only ($n_\text{inputs} = 1$). During the forward pass we calculate the activations a^l of layer l, i.e.

$$a_j^l = f_l(z_j^l), \tag{30}$$

where z_j^l is the sum of a weighted sum of inputs from the previous layer and the bias of layer l,

$$z_j^1 = \sum_{i=1}^{n_\text{f}} w_{ij}^1 x_i + b_j^1. \tag{31}$$

Assuming the layers have N_l number of neurons, the calculated z_j^l of subsequent layers takes the form

$$z_j^l = \sum_{i=1}^{N_{l-1}} w_{ij} x_i + b_j^l, \tag{32}$$

where we note that $W^l \in \mathbb{R}^{N_{l-1} \times N_l}$.

After performing the forward pass, we calculate the cost function and its derivative w.r.t. the weights in the output layer W^L,

$$\begin{aligned}\frac{\partial C(W^L)}{\partial w_{jk}^L} &= \frac{\partial C(W^L)}{\partial a_j^L}\left[\frac{\partial a_j^L}{\partial w_{jk}^L}\right] \\ &= \frac{\partial C(W^L)}{\partial a_j^L}\left[\frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L}\right] \\ &= \frac{\partial C(W^L)}{\partial a_j^L} f_L'(z_j^L) a_k^{L-1},\end{aligned} \tag{33}$$

where we used that (note $y_j = a_j^L$, i.e. the activations of the final layer)

$$\begin{aligned}\frac{\partial C(W^L)}{\partial a_j^L} &= \frac{\partial}{\partial a_j^L}\left[\frac{1}{2}\sum_{i=1}^{N_\text{O}}(a_j^L - t_j)^2\right] \\ &= a_j^L - t_j,\end{aligned} \tag{34}$$

and

$$\begin{aligned}\frac{\partial z_j^L}{\partial w_{jk}^L} &= \frac{\partial}{\partial w_{jk}^L}\left[\sum_{p=1}^{N_L} w_{jp}^L a_j^{L-1} + b_j^L\right] \\ &= a_j^{L-1}.\end{aligned} \tag{35}$$

We define the quantity in Eq. (33) (apart from a_k^{L-1} as δ_j^L, meaning $\partial C/\partial w_{jk}^L = \delta_j^L a_k^{L-1}$. Applying the chain rule to δ_j^L yields the derivative of the cost function w.r.t. the output layer biases as

$$\begin{aligned}\delta_j^L &= \frac{\partial C(W^L)}{a_j^L}\frac{\partial f^L}{\partial z_j^L} = \frac{\partial C(W^L)}{a_j^L}\frac{\partial a_j^L}{\partial z_j^L} \\ &= \frac{\partial C(W^L)}{\partial z_j^L} = \frac{\partial C(W^L)}{\partial b_j^L}\frac{\partial b_j^L}{\partial z_j^L} \qquad (36) \\ &= \frac{\partial C(W^L)}{\partial b_j^L}, \qquad (37)\end{aligned}$$

where we used that

$$\begin{aligned}\frac{\partial b_j^L}{\partial z_j^L} &= \left[\frac{\partial z_j^L}{\partial b_j^L}\right]^{-1} \\ &= \left[\frac{\partial}{\partial b_j^L}\sum_{i=1}^{N_{L-1}} w_{ij}^L a_i^{L-1} + b_j^L\right]^{-1} \\ &= 1.\end{aligned} \tag{38}$$

We have thus found the derivatives of the cost function w.r.t. both the weights and biases in the output layer, W^L and $\mathbf{b}^L$.

The equation

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \tag{39}$$

holds for any layer, not just the output as in Eq. (36). Relating this to derivatives w.r.t. the layer $l+1$ z_js, we find

$$\begin{aligned}\delta_j^l = \frac{\partial C}{\partial z_j^l} &= \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{z_j^l} \\ &= \sum_k \delta_k^{l+1}\frac{\partial z_k^{l+1}}{z_j^l} \\ &= \sum_k \delta_k^{l+1} w_{kj}^{l+1}\frac{\partial f^l}{\partial z_j^l},\end{aligned} \tag{40}$$

with

$$\begin{aligned}\frac{\partial z_k^{l+1}}{\partial z_j^l} &= \frac{\partial}{\partial z_j^l}\left[\sum_{i=1}^{N_l} w_{ik}^{l+1} a_k^l + b_k^{l+1}\right] \\ &= \frac{\partial}{\partial z_j^l}\left[\sum_{i=1}^{N_l} w_{ik}^{l+1} f^l(z_k^l) + b_k^{l+1}\right] \\ &= w_{jk}^{l+1} f^l(z_j^l).\end{aligned} \tag{41}$$

The rest of the backpropagation scheme is essentially iterating Eq. (40), and computing—for each layer—the gradients $\partial C/\partial w_{ij}^l = \delta_i^l a_j^{l-1}$ and $\partial C/\partial b_i^l = \delta_i^l$. Once the gradients are known,

updating the weights and biases to improve the performance of the network (making the cost function smaller) can be done by e.g. gradient descent schemes, c.f. section G.

Exploding / vanishing gradients and weight initialization

Typical transfer functions are constant or close to constant on most of $\mathbb{R}$, and only changes appreciably in a tiny region around the origin. This means that a fully *saturated* neuron with input $z_j \gg 1$, or a *dead* neuron with input $z_j \ll -1$ will most likely exhibit very small gradients and change very little durign training. This means the neurons are essentially wasted, they only add a constant input to the neurons in the subsequent layer; a job already performed by the bias b_{j+1}. In order to avoid this effect, it is important to initialize the weight matrices in the network in a smart way.

In the useful region around the origin, we may assume that the transfer functions are roughly linear. In order for the signal to propagate through the network usefully, we essentially want the mean value of the z_js to vanish, and the variance to be on the order of 1. Let us now consider an input vector X of n components. If we take the weights to be random—as is the case in the first forward pass—then W is a random vector of weights W_i. With the previous assumption about linearity of the transfer function, we get the activation

$$A = W_1X_1 + W_2X_2 + \cdots + W_nX_n, \tag{42}$$

which has variance

$$\begin{aligned}\mathrm{Var}(A) &= \mathrm{Var}(W_1X_2 + \cdots + W_nX_n) \\ &= n\mathrm{Var}(W_i)\mathrm{Var}(X_i),\end{aligned} \tag{43}$$

where we assumed that the inputs and the weights are all independent, identically distributed, with vanishing mean. If we want the activation to have variance on the order of 1, then we must require that the variance of the weights is

$$\mathrm{Var}(W_i) = \frac{1}{n}. \tag{44}$$

Performing the same analysis with the backpropagated signal yields the same result,

$$\mathrm{Var}(W_i) = \frac{1}{n'}, \tag{45}$$

with n' being the amount of weights in the *next* network layer.

With this in mind, Glorot & Bengio suggested initializing weights with average variance:**glorot2010understanding**

$$\mathrm{Var}(W_i^l) = \frac{1}{n_l + n_{l+1}}. \tag{46}$$

In the case of sigmoid or hyperbolic tangent transfer functions, we may realize this variance by initializing $w = \mathcal{U}(-r, r)$ with

$$\begin{aligned}r_{\mathrm{sigmoid}} &= \sqrt{\frac{6}{n_{\mathrm{in}} + n_{\mathrm{out}}}}, \\ r_{\tanh} &= 4\sqrt{\frac{6}{n_{\mathrm{in}} + n_{\mathrm{out}}}},\end{aligned}$$

where $\mathcal{U}(a, b)$ denotes a uniform distribution between a and b and n_{in} (n_{out}) is the number of neurons in the current (next) layer.

With rectifying linear units, the weight initialization scheme changes slightly. As the ReLU transfer function vanishes across half of $\mathbb{R}$, He et al.**he2015delving** suggest doubling the variance of the weights in order to keep the propagating signal's variance constant, i.e.

$$\mathrm{Var}(W) = \frac{2}{n_{\mathrm{in}}}. \tag{47}$$

We may realize this by initializing weights $w = \mathcal{N}(0, 1)\sqrt{2/n_{\mathrm{in}}}$, where $\mathcal{N}(\mu, \sigma)$ denotes the normal distribution with mean μ and standard deviation σ.

Gradient Descent

Almost every problem in machine learning and data science starts with a dataset X, a model $g(\theta)$, which is a function of the parameters θ and a cost function $C(X, g(\theta))$ that allows us to judge how well the model $g(\theta)$ explains the observations X. The model is fit by finding the values of θ that minimize the cost function. Ideally we would be able to solve for θ analytically, however this is not possible in general and we must use numerical methods to compute the minimum.

The method of steepest descent

The basic idea of gradient descent is that a function $F(\mathbf{x})$, $\mathbf{x} \equiv (x_1, \cdots, x_n)$, decreases fastest if one goes from $\mathbf{x}$ in the direction of the negative gradient $-\nabla F(\mathbf{x})$. It can be shown that if

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma_k \nabla F(\mathbf{x}_k), \quad \gamma_k > 0 \tag{48}$$

for γ_k small enough, then $F(\mathbf{x}_{k+1}) \leq F(\mathbf{x}_k)$. This means that for a sufficiently small γ_k we are always moving towards smaller function values, i.e a minimum.

This observation is the basis of the method of steepest descent, which is also referred to as just gradient descent (GD). One starts with an initial guess $\mathbf{x}_0$ for a minimum of F and compute new approximations according to

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma_k \nabla F(\mathbf{x}_k), \quad k \geq 0. \tag{49}$$

The parameter γ_k is often referred to as the step length or the learning rate in the context of ML.

Ideally the sequence $\{\mathbf{x}_k\}_{k=0}$ converges to a *global* minimum of the function F. In general we do not know if we are in a global or local minimum. In the special case when F is a *convex function, all local minima are also global minima*, so in this case gradient descent can converge to the global solution. The advantage of this scheme is that it is conceptually simple and straightforward to implement.

However the method in this form has some severe limitations:

- In machine learing we are often faced with non-convex high dimensional cost functions with many local minimum. Since GD is deterministic we will get stuck in a local minimum, if the method converges, unless we have a very good intial guess. This also implies that the scheme is sensitive to the chosen initial condition.
- Note that gradient is a function of $\mathbf{x} = (x_1, \cdots, x_n)$ which makes it expensive to compute numerically.
- GD is sensitive to the choice of learning rate γ_k. This is due to the fact that we are only guaranteed that $F(\mathbf{x}_{k+1}) \leq F(\mathbf{x}_k)$ for *sufficiently* small γ_k. The problem is to determine an optimal learning rate. If the learning rate is chosen to small the method will take a long to converge and if it is to large we can experience erratic behavior.
- Many of these shortcomings can be alleviated by introducing randomness. One such method is that of Stochastic Gradient Descent (SGD).

Stochastic Gradient Descent

Stochastic gradient descent (SGD) and variants thereof address some of the shortcomings of the Gradient descent method discussed above.

The underlying idea of SGD comes from the observation that the cost function, which we want to minimize, can almost always be written as a sum over n datapoints $\{\mathbf{x}_i\}_{i=1}^n$,

$$C(\theta) = \sum_{i=1}^{n} c_i(\mathbf{x}_i, \theta). \tag{50}$$

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

$$\nabla_\theta C(\theta) = \sum_{i}^{n} \nabla_\theta c_i(\mathbf{x}_i, \theta). \tag{51}$$

Now, stochasticity/randomness is introduced by only taking the gradient on a subset of the data called minibatches. If there are n datapoints and the size of each minibatch is M, there will be n/M minibatches. We denote these minibatches by B_k where $k = 1, \cdots, n/M$.

As an example, suppose we have 10 datapoints $(\mathbf{x}_1, \cdots, \mathbf{x}_{10})$ and we choose to have $M = 5$ minibathces, then each minibatch contains two datapoints. In particular we have $B_1 = (\mathbf{x}_1, \mathbf{x}_2), \cdots, B_5 = (\mathbf{x}_9, \mathbf{x}_{10})$. Note that if you choose $M = 1$ you have only a single batch with all datapoints and on the other extreme, you may choose $M = n$ resulting in a minibatch for each datapoint, i.e $B_k = \mathbf{x}_k$.

The idea is now to approximate the gradient by replacing the sum over all datapoints with a sum over the datapoints in one the minibatches picked at random in each gradient descent step

$$\begin{aligned} \nabla_\theta C(\theta) &= \sum_{i=1}^{n} \nabla_\theta c_i(\mathbf{x}_i, \theta) \\ &\rightarrow \sum_{i \in B_k}^{n} \nabla_\theta c_i(\mathbf{x}_i, \theta). \end{aligned} \tag{52}$$

Thus a gradient descent step now looks like

$$\theta_{j+1} = \theta_j - \gamma_j \sum_{i \in B_k}^{n} \nabla_\theta c_i(\mathbf{x}_i, \theta) \tag{53}$$

where k is picked at random with equal probability from the interval $[1, n/M]$. An iteration over the number of minibathces n/M is commonly referred to as an epoch. Thus it is typical to choose a number of epochs and for each epoch iterate over the number of minibatches.

Taking the gradient only on a subset of the data has two important benefits. First, it introduces randomness which decreases the chance

that our opmization scheme gets stuck in a local minima. Second, if the size of the minibatches are small relative to the number of datapoints ($M < n$), the computation of the gradient is much cheaper since we sum over the datapoints in the k-th minibatch and not all n datapoints.

A natural question is when do we stop the search for a new minimum? One possibility is to compute the full gradient after a given number of epochs and check if the norm of the gradient is smaller than some threshold and stop if true. However, the condition that the gradient is zero is valid also for local minima, so this would only tell us that we are close to a local/global minimum. However, we could also evaluate the cost function at this point, store the result and continue the search. If the test kicks in at a later stage we can compare the values of the cost function and keep the θ that gave the lowest value.

Another approach is to let the step length γ_j depend on the number of epochs in such a way that it becomes very small after a reasonable time such that we do not move at all.

As an example, let $e = 0, 1, 2, 3, \cdots$ denote the current epoch and let $t_0, t_1 > 0$ be two fixed numbers. Furthermore, let $t = e \cdot m + i$ where m is the number of minibatches and $i = 0, \cdots, m-1$. Then the function

$$\gamma_j(t; t_0, t_1) = \frac{t_0}{t + t_1} \tag{54}$$

goes to zero as the number of epochs gets large. I.e. we start with a step length $\gamma_j(0; t_0, t_1) = t_0/t_1$ which decays in "time" t.

In this way we can fix the number of epochs, compute θ and evaluate the cost function at the end. Repeating the computation will give a different result since the scheme is random by design. Then we pick the final θ that gives the lowest value of the cost function.

Gradient descent improvements: Adam

While the stochastic gradient descent alleviates some of the problems intrinsic in the basic steepest descent method, it still has some problems. The *stochasticity* allows it to possibly make jumps over small barriers, essentially transitioning into other basins of more optimal local minima. However, the SGD method struggles with navigating surfaces in parameter space in which the gradient is much steeper in one direction than the other(s). In this case, the iterations $\theta_i = (\alpha_i, \beta_i)$ will rapidly oscillate between over/under-shooting α_i values (with the steep gradient), while slowly making progress towards the minimum in β_i (the shallow gradient).

We can help the SGD overcome this problem by introducing a *momentum* term.[qian1999monentum] Instead of recomputing the gradient at each iteration, we keep a part of the change at the previous time step, essentially giving the optimization a momentum—accelerating the minimization in *parameter space directions* in which the gradient is not steep, but consistently has a small value aimed steadily in one direction. It also hampers the rapid oscillating solutions in *parameter space directions* in which we are close to the optimium, and the steep gradient makes the SGD over/under-shoot the solution at every other iteration.

Each minibatch, the parameter update changes to

$$\theta_{j+1} = \theta_j - [\eta \nabla_\theta c_i(\mathbf{x}_i, \theta_{j-1})] - \gamma_j \nabla_\theta c_i(\mathbf{x}_i, \theta_j),$$

with the momentum term η usually set to a value close to 1.0, e.g. $\eta = 0.9$. The momentum term may be extended with a *Nesterov accelerated gradient* scheme, which basically adds adaptive momentum term.

While introducing momentum into the SGD method improves the scheme, we are still disregarding a lot of previous—possibly relevant—information when we re-compute the gradient at each iteration and throw away all the history of previous gradients computed. In general, we should be able to use past moments of the calculated in previous iteration steps as a guide for the current gradient step in order to improve performance. This is exactly the motivation behind the Adam (Adaptive moment estimation) optimizer.[kingma2014adam]

The Adam scheme uses a moving, exponentially decaying, average of the first and second moments of the gradient to compute individual adaptive learing rates for each parameter independently. The moving average of the gradient m_j is an estimate of the mean of the gradient, while the moving average of the gradient squared v_j is an estimate of the (uncentered[4])

[4]The uncentered variance estimate of $\{f_i\}_{i=1}^N$, $\mathrm{Var_u}(f)$ is the variance computed assuming $\langle f_i \rangle = 0$, meaning the average is *not* subtracted for each sample like usual,

$$\mathrm{Var_u}(f) = \mathrm{E}\left[\Big(f\Big)^2\right] \neq \mathrm{E}\left[\Big(f - \langle f \rangle\Big)^2\right] = \mathrm{Var}(f). \tag{55}$$

At step $j = 0$, initialize $m_0 = v_0 = 0$.

(1) Iterate the step counter $j \leftarrow j + 1$.

(2) Calculate the gradient $g_j \leftarrow \nabla_\theta c(\mathbf{x}_i, \theta_j)$.

(3) Update biased first moment estimate, $\quad m_j \leftarrow \beta_1 m_{j-1} + (1 - \beta_1) g_j$.

(4) Update biased second moment estimate, $v_j \leftarrow \beta_2 v_{j-1} \ + (1 - \beta_2) g_j^2$.

(5) Compute the bias-corrected first moment esimate, $\quad \hat{m}_j \leftarrow \frac{m_j}{1-\beta_1^j}$.

(6) Compute the bias-corrected second moment esimate, $\hat{v}_j \leftarrow \frac{v_j}{1-\beta_2^j}$.

(7) Update parameter vector, $\theta_j \leftarrow \theta_{j-1} - \alpha \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \varepsilon}$.

Algorithm 1. The *Adam* optimizer for stochastic optimization. The constants β_1, β_2, and ε take appropriate default values $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\varepsilon = 10^{-8}$. The initial step size α may be taken to be 0.001. The gradient squared, g_j^2, denotes the elementwise square.

variance of the gradient. At the first iteration, we assign $m_0 = v_0 = 0$, which means the estimates are inherently biased towards zero. In the Adam scheme, this is counteracted by computing bias-corrected estimates $\hat{m}_j$ and $\hat{v}_j$. The Adam optimization is outlined in algorithm 1.

MODEL SYSTEMS

In this work we apply the machine learning algorithms discussed to the one- and two-dimensional Ising model. Linear regression and neural networks are used to estimate the coupling constant of the one-dimensional Ising model.

The two-dimensional Ising model is known to exhibit a first order phase transition at the critical *Curie temperature*. In particular, the states of the model at sub-critical temperatures present as ordered, with large regions of spins aligned. At super-critical temperatures, the spins are disordered and essentially randomly uncorrelated. Thus it is interesting to investigate whether logistic regression or neural networks can be trained to to classify the phases.

The Ising model

The Ising model can be thought of as a microscopic model of a ferromagnetic metal. The solid state metal exists in some energy-preferable lattice state—for simplicity, we assume a square lattice—and the magnetic dipole moments of the electrons due to their intrinsic spins are taken to occupy fixed positions on this lattice. In more complicated models (such as the Potts model[potts*1952]) the spins are allowed to take many different values, but in the basic Ising model the spins are restricted to being aligned either up or down (+1 or −1). The quantum mechanical exchange interaction energy between spins gives rise to the (no external magnetic field) simple model Hamiltonian for the system

$$H(\mathbf{s}) = -\sum_{i,j} J_{ij} s_i s_j, \tag{56}$$

where J_{ij} is the interaction energy matrix governing the size of the interaction between spins i and j, while the state of the system (alignment of all spins) is denoted by $\mathbf{s} = \{s_i\}_{i=1}^N$. If J_{ij} is positive, the energetically most efficient state of the system is realized if spins s_i and s_j are aligned in the same direction. Taking the total magnetization $\mathcal{M}$ to be the sum of all the spins (in the sense that spins aligned up are represented by +1, and downward facing spins are represented by −1), this J_{ij} would facilitate spontaenous magnetization and thus model a ferromagnet. If $J_{ij} \leq 0$, the model system is a paramagnet (ideal paramagnet in the case of equality), giving rise to a $\mathcal{M} = 0$ state unless an external field is applied.

The simplest case of *nearest neighbor* interac-

tions only is realized if J takes the form

$$J_{ij} = J_0\Big[\delta_{i,j+1} + \delta_{i,j-1} + \delta_{1,i}\delta_{j,N} + \delta_{N,i}\delta j, 0\Big],$$

where the last term represents the periodic boundary conditions on i and j. The δs here denote Kronecker deltas. We note that the constant J_0 must carry dimensions of energy per squared magnetic dipole moment. From now on we will assume $J > 0$ in the rest of the project.

An example of a 2D Ising lattice is shown in Fig. 3.

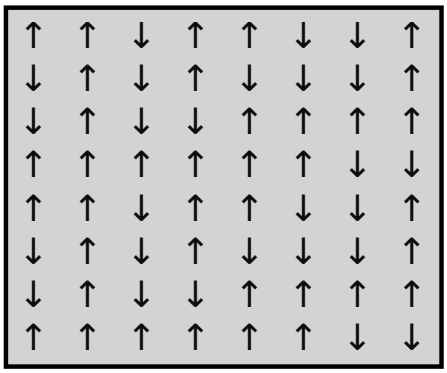

FIG. 3. Example of a 2D square lattice of spins representing a (subset of a) full magnet. Counting the number of spins pointing up and the number pointing down yields 38 total ↑s and 26 ↓s, meaning we would in this case have a net magnetization in the ↑ direction.

The energetically most favorable state of the Ising lattice is one in which all spins are aligned either up or down. The energy of both *all up* and *all down* is the same—in the 1D case $E_{\text{ground}} = -NJ_0$—meaning there are two ground states. Assuming we put the spin model in thermal contact with a large heat bath at temperature T, but keep the number of spins and their occupied volume constant, we may write down the canonical partition function of the Ising model by considering every possible state of the lattice ($\mathcal{S}$) weighted by their respective Boltzmann factors,

$$Z = \sum_{\mathbf{s}_k \in \mathcal{S}} \mathrm{e}^{-H(\mathbf{s}_k)/k_\mathrm{B}T}. \tag{57}$$

Since flipping every spin in the lattice simultaneously does not affect the energy, there exists a *flipped* version of every single state in $\mathcal{S}$ with the same energy and the same Boltzmann factor. When calculating canonical expectation values, the sum is taken over every possible state, so finding the expected magnetization of the Ising lattice will always yield zero since

$$\begin{aligned}\langle \mathcal{M} \rangle &= \frac{1}{Z} \sum_{\mathbf{s}_k \in \mathcal{S}} \mathcal{M}(\mathbf{s}_k) \mathrm{e}^{-H(\mathbf{s}_k)/k_\mathrm{B}T} \\ &= \frac{1}{Z} \sum_{\mathbf{s}_k, \mathbf{s}'_k \in \mathcal{S}} \Big[\mathcal{M}_k - \mathcal{M}_{k'}\Big] \mathrm{e}^{-\beta E_k} = 0,\end{aligned}$$

with $\mathcal{M}(\mathbf{s}) = \mathcal{M}(\mathbf{s}')$ and $\mathbf{s}'$ representing the lattice state $\mathbf{s}$ with every single spin flipped. We here used $\beta = 1/k_\mathrm{B}T$ and defined $H(\mathbf{s}) \equiv E_k$. Note that this is independent of the temperature.

However, the two ground states both have $\mathcal{M} \neq 0$. This constitutes a spontaneous symmetry breaking: When cooling the model from $T > T_\mathrm{c}$ (the critical or *Curie* temperature) to a temperature $T < T_c$, the magnetization will transition from a $\mathcal{M} = 0$ state (super-critical, disordered) to an ordered $\mathcal{M} = \pm N$ ground state. At (or around) $T = T_c$ the energetic spin up/down symmetry is spontaneously broken, and the Ising lattice falls into one of the two possible ground states. This symmetry may be *explicitly* broken by applying an external magnetic field $\mathbf{B}$, which adds a term $\propto \sum_{i=1}^{N} \mathbf{B} \cdot s_i$ to the Hamiltonian, breaking the $\mathbf{s} \rightleftharpoons \mathbf{s}'$ symmetry.

An example of this symmetry breaking is visualized in Fig. 4. Using the Metropolis algorithm to sample a pseudo-time evolution for a 2D Ising model with $N \times N$ spins, $N = 1000$. Note that starting from a disordered state, the all-spins up ground state is randomly chosen by the system at $T < T_c$.

At the Curie temperature T_c, the 2D Ising model shows a second order *phase transition*. Being a second order transition, the second derivatives of the free energy diverge. These include the spesific heat and the magnetic susceptibility,

$$\begin{aligned} c_V &= \left(\frac{\partial E}{\partial T}\right)_{V,N} = -T\left(\frac{\partial^2 F}{\partial T^2}\right)_{V,N} \\ \chi &= \left(\frac{\partial \mathcal{M}}{\partial B}\right)_{T,V} = \left(\frac{\partial^2 F}{\partial B^2}\right)_{T,V} \end{aligned}$$

The free energy F here denotes the Helmholtz free energy.

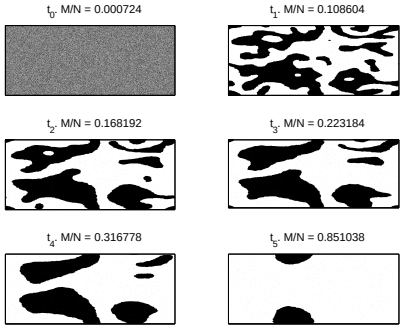

FIG. 4. Results of application of the Metropolis algorithm to the Ising model with a randomized starting lattice. White pixels denote spin up, while black pixels denote spin down. The lattice used is $N \times N$ spins large, where $N = 1000$. t_i denote "time", i.e. the number of Monte Carlo cycles performed, but t_i and t_{i+1} are not necessarily equidistant for every i. $\mathcal{M}$ denotes the sum of all the spins (1 or -1) such that $\mathcal{M}/N^2$ is the relative magnetization of a single spin in the lattice. The temperature used was 1.2 in units of $k_{\mathrm{B}}T/J_0$, where k_{B} is the Boltzmann constant.

RESULTS AND DISCUSSION

Regression analysis on the one-dimensional Ising Hamiltonian

Since we have already validated the implementations of the linear regression methods in project 1, we go straight to the applications on the Ising model.

In the following we will take the 1D Ising model interaction matrix J to be equal to

$$J = J_0 \begin{pmatrix} 0 & 1 & 0 & 0 & \dots & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \end{pmatrix} \tag{58}$$

with $J_0 = 1$ being the coupling constant. We will start out by applying the regression schemes developed in project 1 to the 1D spin lattice and estimate the coupling constant J_0. We generate $N_s = 1000$ sample lattice configurations (with the lattice size $L = 40$) $\mathbf{s}_k$ and compute their energies as $H(\mathbf{s}_k) = \mathbf{s}_k^T J \mathbf{s}_k$. For each configuration, we construct the design matrix by taking every possible combination of two spins multiplied. The rows of the design matrix take the following form

$$\mathrm{Row}_i(X) = \Big(s_1 s_1,\, s_1 s_2, \dots, s_1 s_N,\, s_2 s_1,\, s_2 s_2,\, s_2 s_3, \dots, s_N s_{N-1},\, s_N s_N\Big). \tag{59}$$

The samples are split into a training set and a validation (test) set. We use a 50/50 split, so the training set has size $N_t = 500$ and the validation set has size $N_v = 500$. The target data we fit against is the energy, E_k. Since the input is in the form of pairs of spins, we can directly interpret the found optimal β parameters to be the full interaction matrix J. Inspired by the paper by Mehta et al.,[mehta2018highbias] we calculate the coefficient of determination, R^2 score, for the three different regression schemes, with varying degrees of regularization λ. This is shown in Fig. 5. The reader is referred to the project 1 report for the definitions of the OLS/Ridge/Lasso schemes, λ, and the R^2 score.

Shown in the figure are the R^2 score as calculated from the training and the validation set for all three schemes. As expected, the OLS scheme is the best possible fit to the *training* data. However, for a better gauge of overall model performance is the fit on untrained validation data. Here we see that the Lasso regression scheme with small regularization parameters dominates the other two methods. For $\lambda \sim 10^{-1}$—10^{-4} the Lasso scheme exhibits an $R^2 \simeq 1$, while the OLS and Ridge methods

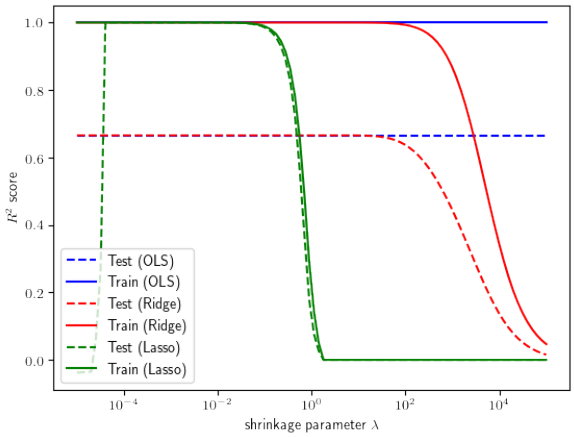

FIG. 5. Coefficient of determination, R^2 score, for the ordinary least squares, the Ridge, and the Lasso regression schemes applied to the 1D Ising model energy. Inspired by a similar figure in [**mehta2018highbias**]. As expected, the OLS scheme outperforms the other methods on the *training* set, but the Lasso regularization scheme is the only scheme which performs adequately on the validation set (for regularization parameter λ in the range of approximately 10^{-1}—10^{-4}).

fail to break $R^2 \sim 0.7$ across the entire broad range of λ values tried. The ridge regression scheme performs worse than the OLS—on both the training *and* validation sets—for $\lambda \gtrsim 10^2$. For smaller regularization parameters, the two are essentially indistinguishable.

As the optimized β parameters can be interpreted as the estimated interaction matrix J, it is natural to visualize the form of the β matrices for each method. This is shown in Fig. 6, where we perform fits for varying values of regularization λ. It is interesting to note that in the cases of OLS and Ridge regression, the solutions found exhibit a new symmetry that the input J matrix does not. In the generated input data, only the upper $i = j+1$ diagonal of J is populated with the values $J_0 = 1$. However, the computed energy is the same if we take J to have both the $i = j+1$ and $i = j-1$ diagonals populated by $J_0/2$. This is what is found by the two mentioned schemes. The Lasso method, on the other hand, finds the single diagonal version of J as it prefers performing variable selection—essentially zeroing out any unnecessary βs not absolutely needed—to find the optimal parameters.

TODO: Add bias-variance tradeoff analysis for the linear regression schemes!

Neural network regression

We now turn our attention to regression analysis using neural networks. Training neural networks is a lot more tricky than optimizing linear regression models, so preparing our training samples requires a bit more finesse in this case. The statistical mechanical multiplicity of lattice states with net magnetization and energies close to $E, \mathcal{M} \sim 0$ is **very** large compared the multiplicity of the ground state (2) and low/high $E, \mathcal{M}$ states. This means that if we just randomly generate N arrays filled with L values of -1 or $+1$, we will generate a vast majority of states with $E, \mathcal{M}$ close to zero, and almost no states with extreme $E \sim \pm J_0 L, \mathcal{M} \sim \pm L$. This yields an inherent problem, as the NN can not be expected to approximate well sample states which it has only seen *a few* times before in training. This problem of unbalanced training data can be solved easily in this case (but alas not in general, when generating new data is not as easy as pressing a button) by being smart about picking samples.

The ground state(s) have energy $-J_0 L$, and changing any spin in a row of three aligned spins ($\uparrow\uparrow\uparrow \to \uparrow\downarrow\uparrow$ or $\downarrow\downarrow\downarrow \to \downarrow\uparrow\downarrow$) yields an energy difference of $\Delta E = 4J_0$. The maximal energy state is the one in which alternating spins are always un-aligned, i.e. $\ldots \uparrow\downarrow\uparrow\downarrow\uparrow \ldots$, and this state has energy $+J_0 L$. For $J_0 = 1$, that means there are $2L/4$ total possible energies for the 1D lattice. Starting in the *maximal* energy state (which itself is doubly degenerate), we can generate a spin lattice of energy $J_0 L - 4J_0 \mathcal{F}$ by flipping $\mathcal{F}$ of the even-indexed spins only. Note carefully that if we start with the ground state and flip $\mathcal{F}$ spins at random, we are in no way guaranteed to end up with a state of energy $-J_0 L + 4J_0 \mathcal{F}$. In fact, this would only happen if you randomly picked only odd or only even numbered spins to flip, because otherwise you would perform at least one operation which would result in an energy difference $\Delta E \neq 4J_0$ like e.g. $\uparrow\uparrow\downarrow \to \uparrow\downarrow\downarrow$ which gives an energy difference of $\Delta E = 0$ (the

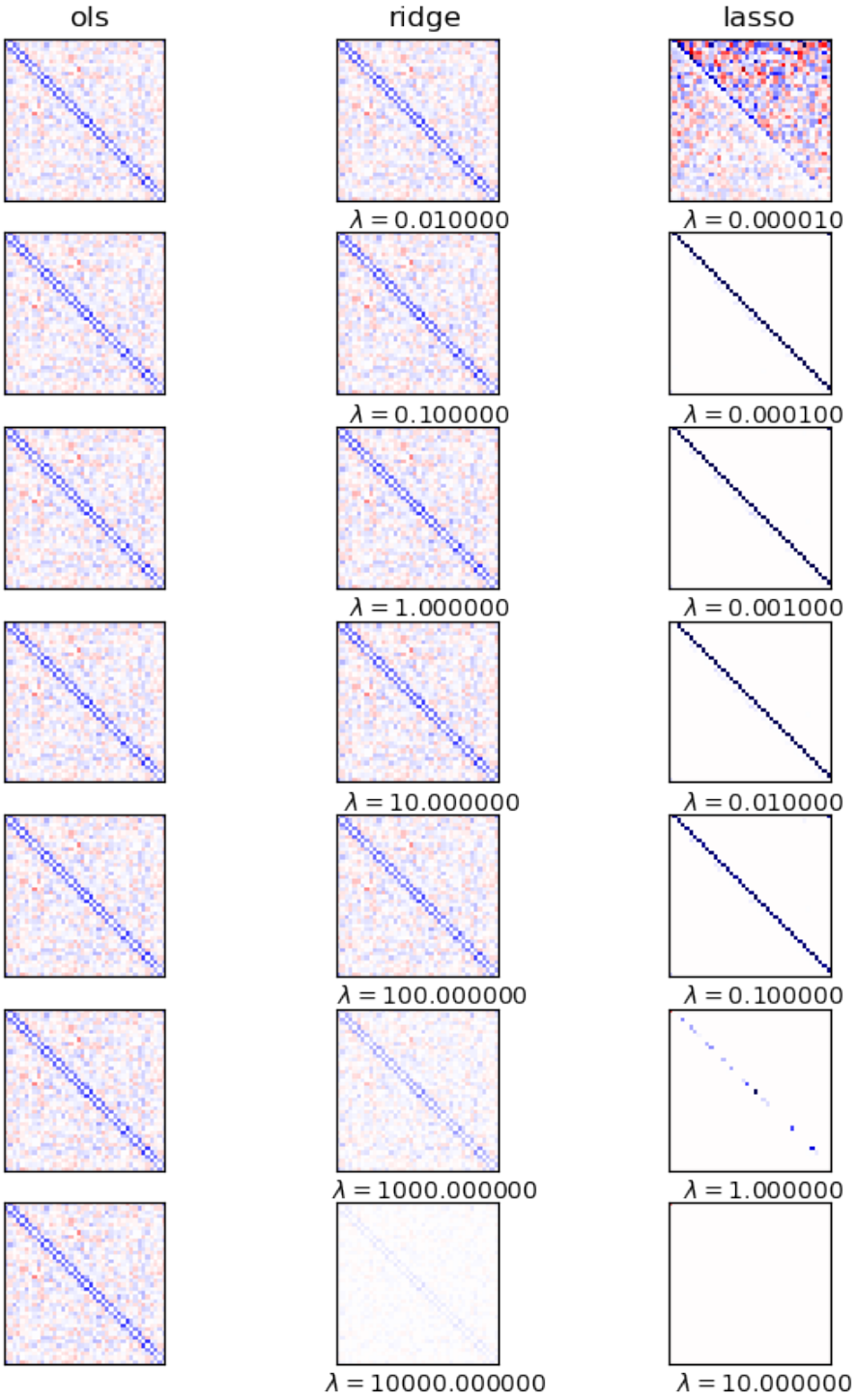

FIG. 6. Visualizations of the J matrix estimates gotten from the linear regression schemes discussed in section I. The left column shows the OLS solution, the middle shows the Ridge regression solution, while the right hand side shows the interaction matrix as calculated by the Lasso regression scheme. Since the OLS method is independet of the regularization parameter λ, all the OLS matrices are equal. We note that in the OLS and Ridge cases, the a symmetric J matrix is found, with values $0.5J_0$ on the $i = j + 1$ and $i = j - 1$ diagonals, while the input matrix only contains the upper $i = j + 1$ diagonals. The latter is found by the Lasso scheme. Inspired by a similar figure in [**mehta2018highbias**]. Please note carefully that the value of λ used in the Ridge regression is three orders of magnitude larger than that of Lasso in each row of this figure.

two states are related by a flip of all three spins, and a left-right mirroring, neither of which affect the energy).

Only in the case of starting from the maximal energy state and picking only odd or only even indexed spins to flip can we be sure to get a lattice of a specific energy. As an example, consider the $L = 19$ example shown in the following, where we randomly pick $\mathcal{F} = 3$ even numbered spins (↓s) to flip. We denote the spins to be flipped by a $\odot$

$$\begin{aligned}
&\text{Before:} \quad \uparrow \underset{\odot}{\downarrow} \uparrow \downarrow \uparrow \downarrow \uparrow \underset{\odot}{\downarrow} \uparrow \downarrow \uparrow \downarrow \uparrow \downarrow \uparrow \underset{\odot}{\downarrow} \uparrow \downarrow \uparrow \downarrow \quad E = J_0 L \\
&\text{After:} \quad \uparrow \uparrow \uparrow \downarrow \uparrow \downarrow \uparrow \uparrow \uparrow \downarrow \uparrow \downarrow \uparrow \downarrow \uparrow \uparrow \uparrow \downarrow \uparrow \downarrow \quad E = J_0(L-12)
\end{aligned} \tag{60}$$

We flipped the three spins with indices 2, 8, and 16 and ended up with a $\Delta E = 3 \cdot 4J_0$. Preparing $N/2$ such samples in the before state, and then iterating over $\mathcal{F} = 0, 1, 2, \ldots, L/2$, applying $\mathcal{F}$ even indexed random spin flips to N/L of the $N/2$ total samples will yield bins of N/L random configurations all with energies $E = J_0(L - 4\mathcal{F})$. For the other $N/2$ samples, we start off with the *other* maximal energy state (all spins flipped), and instead flip successively $\mathcal{F}$ even indexed—but now $\uparrow \rightarrow \downarrow$ instead of the opposite—to not bias the sampling towards one class of configurations. All in all, this yields a set of N samples with even probabilities to take on any of the $2L/4$ possible energies. In contrast, sampling in the naive but simple way overwhelmingly yields samples in the range around $E \sim 0$. Illustrating this, the cumulative probability of picking a random sample with energy E out of the sample set for both versions of the data set generation is shown in Fig. 7. We call the method of sampling outlined in this section *even sampling*. In this plot, a $L = 1000$ lattice is sampled $N \approx 1000$ times. A similar effect (though less extreme) is exhibited by smaller grids.

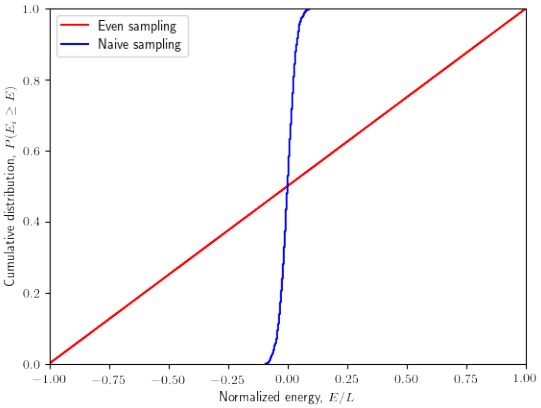

FIG. 7. Cumulative probability density function for picking a random sample with energy E_i out of the total data set, for the straight forward sampling method and the even sampling described in section I. Note that a linear cumulative distribution means the probability density is flat, i.e. uniform probability. The naively generated data set exhibits a probability density function sharply peaked around $E = 0$.

Having devised a method of sampling states which will be helpful for the NN training, we move on to the training. First off, we prepare networks with L inputs, a single hidden layer of L^2 neurons, and an output with identity activations. For the moment, we use sigmoid activations for the hidden layer. We prepare a data set of $N = 5000$ lattice configurations of *evenly* sampled energies, and reserve $N_v = 1000$ of them for validation purposes. The Adam optimizer is run for a total of 1000 epochs, with

a batch size of 1000, and an initial step size $\alpha = 0.001$. The mean squared difference between the target energies and the NN output is used as the cost function for training with the bakcpropagation algorithm. For varying lattice sizes $L = 10, \ldots, 40$ we perform the fit and visualize the output. This is shown in Fig. 8. Note that the energies are normalized (divided by L) and take values in $[-1, 1]$, to help the NN during training.

From the plots in Fig. 8, it is pretty clear that our trained networks leave a lot to be desired. One remedy it increasing the data set size for training, and increasing the training time (number of epochs). In order to compare against the linear regression models we discussed earlier, we investigate now how much training is necessary in order to surpass the performance of the Lasso regression at $\lambda = 10^{-2}$. This was deemed the best of the linear methods earlier.

We consider small enough system sizes L for the training to be feasible (training time on a late–2013 MacBook pro running OSX10.11.6 approximately less than 60 minutes), and keep going until the MSE has reached an apparent plateu. Comparing the mean squared error of the trained NN against the $\lambda = 10^{-2}$ for varying small lattices is shown in Fig. 10. It is clear that in some cases, the NN scheme is able to surpass the performance on the validation sets. However, as the lattice grows in size, the required training time becomes unfeasible, as shown in Fig. 9. It is hard to justify an order of magnitude or two drop in MSE at the cost of over 1500 times increase(!) in preparation time (the Lasso scheme does not require more than at most 2.2 s for even the largest of these lattices).

Recovering the J matrix in the neural network model

As a closing remark for the neural network regression section, we note the apparent difficulty of recreating the J matrix using the neural network scheme. Even using identity activations, the design matrix as network input (each sample being L^2 values of every combination of $s_i s_j$) and a single hidden layer with one neuron (making the first network weight $L \times L$), and freezing the intercepts at 0, we are still not able to recreate the J matrix. As an example, training a $L = 5$ model like this yields an MSE that vanishes to machine precision ($\sim 10^{-30}$), however the resulting weight matrix takes the form:

$$W = \begin{pmatrix} 0.59 & -1.77 & 0.18 & 0.22 & 0.83 \\ 0.81 & -0.14 & -1.65 & -0.39 & -0.72 \\ -0.18 & 0.69 & -0.85 & -0.48 & -0.52 \\ -0.22 & 0.39 & -0.48 & 0.81 & -1.03 \\ -1.79 & 0.72 & 0.52 & 0.07 & -0.41 \end{pmatrix}$$

It is not at all obvious why this matrix should yield essentially zero error in the energy for every single $L = 5$ state available. In order to make sense of this, we note a *key observation*: The energy is given by the quadratic form $\mathbf{s}^T J \mathbf{s} = E$ and any quadratic form $\mathbf{x}^T A \mathbf{x}$ is invariant under the exchange of A and the symmetric part of the matrix, $(A + A^T)/2$. As an example of this, the symmetric part of the standard J interaction matrix used so far in the present work is (zeros ommitted)

$$\frac{J + J^T}{2} = -\frac{1}{2} \begin{pmatrix} & 1 & & & & 1 \\ 1 & & 1 & & & \\ & 1 & & 1 & & \\ & & 1 & & & \\ & & & & \ddots & \\ & & & & & 1 \\ 1 & & & & 1 & \end{pmatrix}, \qquad (61)$$

which we have already discussed: This is is the symmetric version found by the OLS and Ridge regression methods. In the following, we denote the symmetric part of a square matrix A by

$$\operatorname{Sym} A = \frac{A + A^T}{2}, \qquad (62)$$

while the remaining, non-symmetric, part of A is called the *skew*-symmetric (or anti-symmetric) part of A,

$$\operatorname{Skew} A = \frac{A - A^T}{2}. \qquad (63)$$

Note that $A = \operatorname{Sym} A + \operatorname{Skew} A$, with $(\operatorname{Sym} A)^T = \operatorname{Sym} A$ and $(\operatorname{Skew} A)^T = -\operatorname{Skew} A$.

As the aforementioned W matrix recovers the energy to machine precision, that must necessarily mean that $\mathbf{s}^T W \mathbf{s} = \mathbf{s}^T J \mathbf{s} = E$. Computing the symmetric part of W yields an interesting result:

$$\begin{aligned} \operatorname{Sym} W &= -\frac{1}{2} \begin{pmatrix} D_1' & 1 & & & 1 \\ 1 & D_2' & 1 & & \\ & 1 & D_3' & 1 & \\ & & 1 & D_4' & 1 \\ 1 & & & 1 & D_5' \end{pmatrix} \\ &= \operatorname{Sym} J + D. \end{aligned} \qquad (64)$$

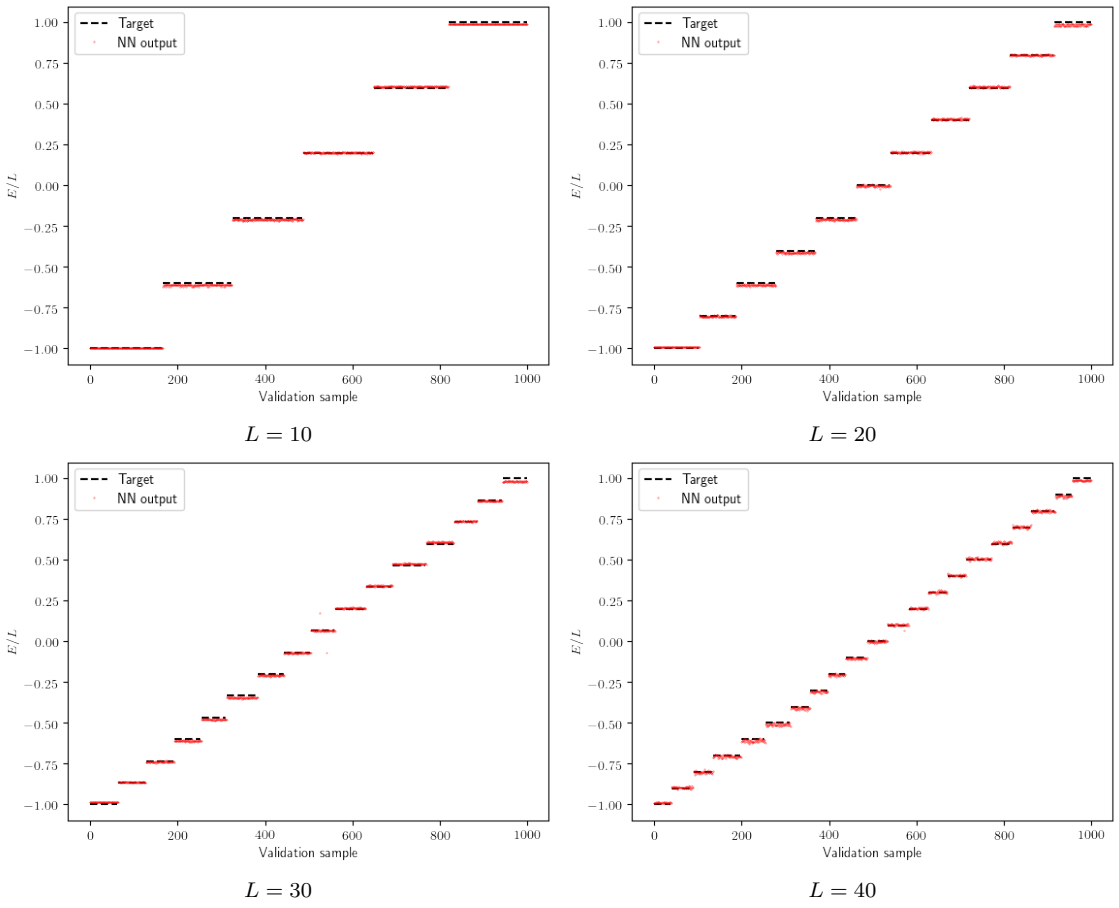

FIG. 8. Raw NN output and target validation energies for different lattice sizes after training for 1000 epochs with a data set of $N = 5000$ samples. The energies are normalized by L. We note that the NN output roughly follows the true target, but more training is necessary to reach very good results.

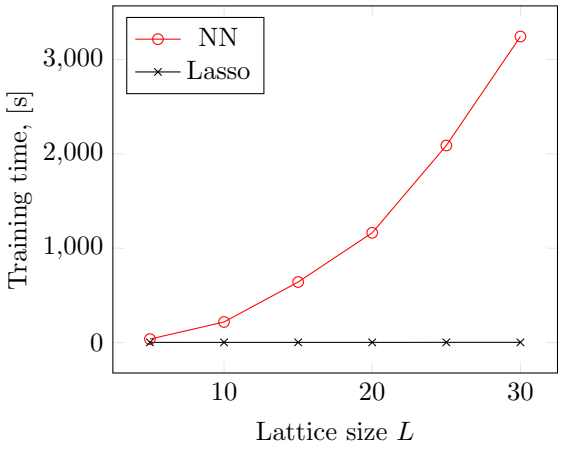

FIG. 9. Training time for the regression neural networks trained to estimate the energy of 1D Ising lattice configurations for different system sizes L. Even though the NN may outperform the linear schemes in terms of the mean squared error on a validation set—for small L—it is hard to justify the cost–benefit ratio in terms of training time required. The Lasso time peaks at around 2 s.

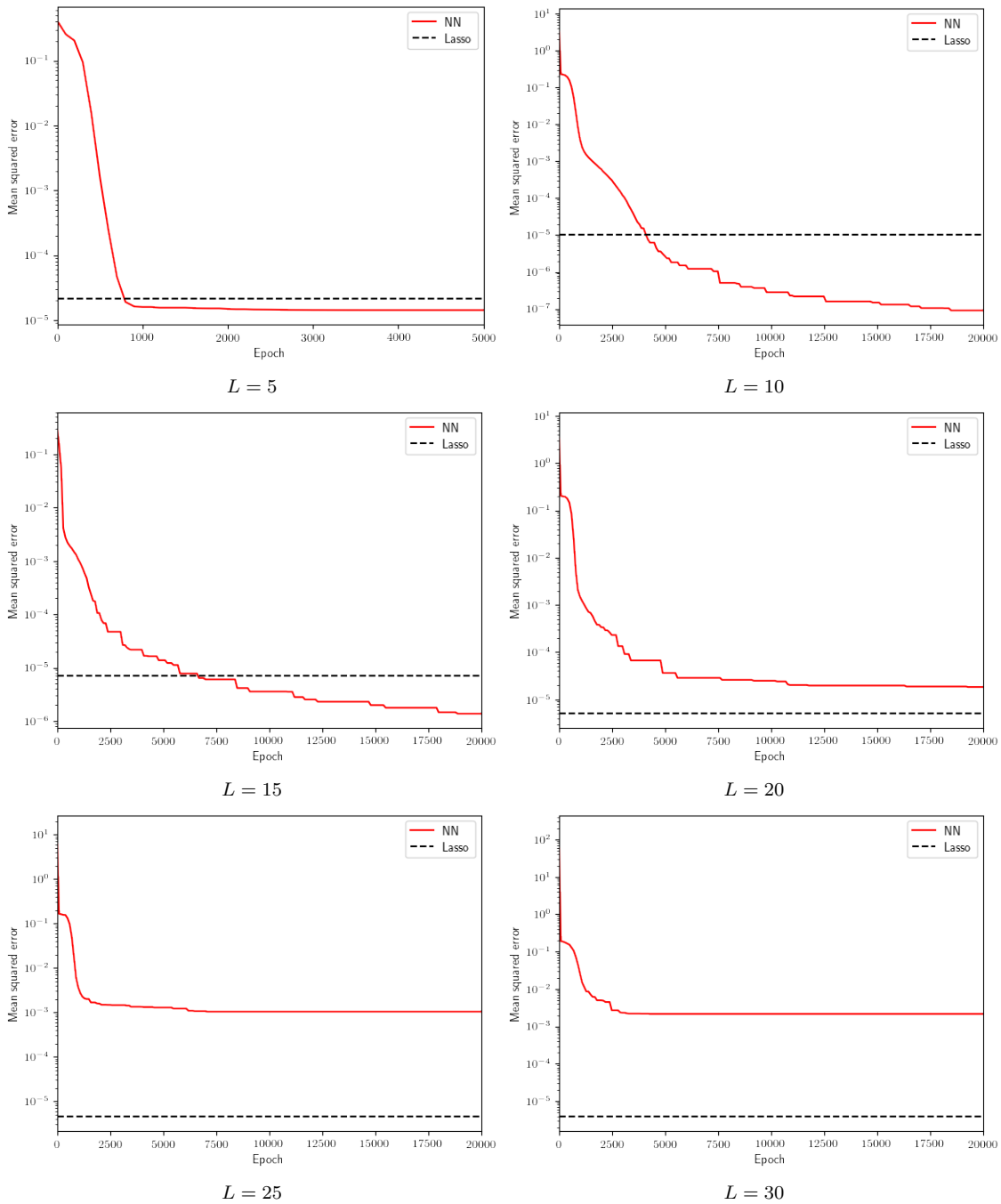

FIG. 10. Mean squared error calculated from trained networks applied to small–medium sized Ising lattices. We note that for sufficiently small system sizes, the NN outperforms the Lasso scheme (which was shown to be the best performer of the linear models). However, as the lattice growns, the training time required either makes training unfeasible or the training is simply not able to reduce the MSE to below that of the Lasso scheme (a plateu is hit before the NN performance surpasses that of the Lasso method). The training time for each network is shown in Fig. 9.

As we can see, we are tantalizingly close to recreating the symmetric part of J—the only difference is the diagonal elements, which we may organize in a matrix D with elements[5] D_i. It is obvious that since the original equation $\mathbf{s}^T W \mathbf{s} = \mathbf{s}^T J \mathbf{s}$ holds, and any quadratic form is invariant under a $A \mapsto \mathrm{Sym}\, A$ transformation, then

$$\begin{aligned} E &= \mathbf{s}^T W \mathbf{s} \\ &= \mathbf{s}^T \mathrm{Sym}\, W \mathbf{s} \\ &= \mathbf{s}^T (\mathrm{Sym}\, J + D) \mathbf{s} \\ &= \mathbf{s}^T \mathrm{Sym}\, J \mathbf{s} + \mathbf{s}^T D \mathbf{s} \\ &= E + \mathbf{s}^T D \mathbf{s} \end{aligned} \tag{65}$$

which can only be true if $\mathbf{s}^T D \mathbf{s} = 0$. In general, quadratic forms are not invariant under $A \mapsto A + D$, with D a diagonal matrix. However, in our special case—recall that the input state can only take values $s_i = \pm 1$—it is, if and only if $\mathrm{Tr}\, D = 0$.

To see that this is the case, consider the D term of Eq. (65), which itself is a quadratic form

$$\mathbf{s}^T D \mathbf{s} = \sum_{i=1}^{L} s_i s_i D_i, \tag{66}$$

and note that this corresponds to self-interaction in the Ising lattice (spin i interacting with spin i, itself). Since $s_i^2 = 1$ for any $s_i \in \{-1, 1\}$, we find that

$$\mathbf{s}^T D \mathbf{s} = \sum_{i=1}^{L} s_i^2 D_i = \mathrm{Tr}\, D. \tag{67}$$

Of course, the W matrix found by our neural network has vanishing trace, explaining how it is able to recover the energy perfectly.

In general, any regression scheme (linear or NN-based or otherwise) is not concerned with recovering the matrix J, but rather with finding any matrix A in the class of $L \times L$ matrices, which satisfy

$$\Big\{ A \in \mathbb{R}^{L \times L} \,:\, \mathrm{Sym}\, A = \mathrm{Sym}\, J + D, \\ D = \mathrm{diag}(D_1, \ldots, D_L),\ \mathrm{Tr}\, D = 0 \Big\}. \tag{68}$$

This whole ordeal outlines a fundamental difficulty in working with machine learning algorithms such as the present one; it is essentially a *black box*. Even when it works perfectly, understanding *why* or *how* it works may involve a considerable amount of effort.

Finally we note a curiosity: The $\mathrm{Tr}\, D = 0$ demand is simply an artifact of the way we train the model. If we add a constant α to all energies when training, $E \mapsto E + \alpha$, we would recover it as $\mathrm{Tr}\, D = \alpha$. Physically, a constant shift in the energy has no impact on the dyanamics of a system, and physically nothing prevents us from adding such terms. In the Ising model, we could interpret the self-interaction of spin s_i as the *self-energy* of spin i, i.e. the energy necessary for spin s_i to simply exist in the lattice regardless of it's interaction with the other spins.

Classifying phases of the two-dimensional Ising model

Logistic regression analysis on the two-dimensional Ising model

We consider the 2D Ising model on a 40×40 lattice. There are three possible types of states: ordered $T/J < 2.0$, critical $2.0 \leq T/J \leq 2.5$ and disordered $T/J > 2.5$.

Mehta et al.[mehta2018highbias] have made avaiable a data set where Monte Carlo sampling is used to prepare 10^4 states at sixteen uniformly spaced temperatures $T/J \in [0.25, 4.0]$.

The logistic regression model is trained using a set of ordered and disordered states, while the critical states are left out. This training set is split 50/50 in a training and test set. When the training procedure is finished, the performance of the model is measured on both the test set and the set of critical states not used in the training phase.

In Fig.11 we have computed the training, test and critical accuracy for regularization parameters $\lambda \in [10^{-5}, 10^5]$. The results are in excellent agreement those obtained by Mehta using scikit-learn's *logistic regression* functionality [6].

For small λ we observe that the accuracies are essentially unaffected by the regularization strength. However for λ roughly in the range $10 - 100$ we see a slight increase in the critical accuracy while the training and test accuracy stays the same. On the other hand, for large λ we observe an increasing critical accuracy at the cost of decreasing training and test accuracy. Furthermore, we note that in general the model seems to perform roughly 10% worse on

[5] The D_i and the D'_i elements shown in Eq. (64) are related by $D_i = -D'_i/2$, where we simply absorbed the constant into D'_i for ease of notation.

[6] https://scikit-learn.org/

the critical states compared with the ordered and disordered states.

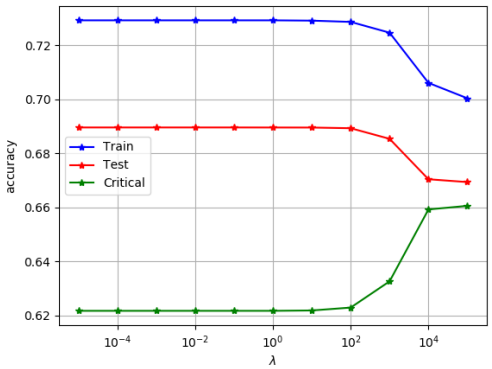

FIG. 11. Training, test and critical accuracy plotted as function of regularization parameter λ. We remark that for small λ the accuracies are essentially unaffected by the regularization. However for λ roughly in the range $10 - 100$ we see a slight increase in the critical accuracy while the training and test accuracy stays the same. On the other hand, for large λ we observe an increasing critical accuracy at the cost of decreasing training and test accuracy.

CONCLUSION

Sed feugiat. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Ut pellentesque augue sed urna. Vestibulum diam eros, fringilla et, consectetuer eu, nonummy id, sapien. Nullam at lectus. In sagittis ultrices mauris. Curabitur malesuada erat sit amet massa. Fusce blandit. Aliquam erat volutpat. Aliquam euismod. Aenean vel lectus. Nunc imperdiet justo nec dolor.