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 5, 2018

Abstract

Contents

I. 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.

Somethingsomething background deep neural networks

Apply it to one-dim Ising and classification of two-dim Ising.

II. 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.

A. 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

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

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

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

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

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

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

is called the *sigmoid* of $\beta\cdot\mathbf{x}$. Also note that the sigmoid satisfies

$$\lim_{t\to\infty}\sigma(t) = 1 \quad (4)$$

$$\lim_{t\to-\infty}\sigma(t) = 0 \quad (5)$$

which justifies its use as a model for probabilities.

The logistic model can now be used for classification by predicting a class using the estimated probabilities according to

$$\hat{y}_i = \begin{cases} 1 & \text{if } p\left(y=1|\mathbf{x}^{(i)}\right) \geq 0.5 \\ 0 & \text{if } p\left(y=1|\mathbf{x}^{(i)}\right) < 0.5. \end{cases} \quad (6)$$

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

$$\begin{aligned} L(\beta) &= \prod_{i:y_i=1} p(y_i=1|\mathbf{x}^{(i)}) \prod_{i:y_i=0} p(y_i=0|\mathbf{x}^{(i)}) \\ &= \prod_{i=1}^n p(y_i=1|\mathbf{x}^{(i)})^{y_i}(1-p(y_i=1|\mathbf{x}^{(i)}))^{1-y_i}. \end{aligned} \quad (7)$$

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

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

$$\begin{aligned} l(\beta) &= \log(L(\beta)) \\ &= \sum_{i=1}^n y_i p(y_i=1|\mathbf{x}^{(i)}) + (1-y_i)(1-p(y_i=1|\mathbf{x}^{(i)}). \end{aligned} \quad (8)$$

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

$$f'(x^*) = 0, \quad f''(x^*) < 0. \quad (9)$$

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

$$\frac{d}{dx}\log(f(x)) = \frac{f'(x)}{f(x)} \tag{10}$$

$$\Rightarrow \frac{d}{dx}\log(f(x^*)) = 0 \tag{11}$$

$$\frac{d^2}{dx^2}\log(f(x)) = \frac{f''(x)f(x) - f'(x)^2}{f(x)^2} \tag{12}$$

$$\Rightarrow \frac{d^2}{dx^2}\log(f(x^*)) < 0, \tag{13}$$

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

$$C(\beta) \equiv -l(\beta). \tag{14}$$

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.

However, the gradient of the cross entropy can be given in closed-form

$$\nabla_\beta C(\beta) = -X^T\left(\mathbf{y} - \mathbf{p}\right). \tag{15}$$

Here we have defined

$$\mathbf{y} \equiv (y_1, \ldots, y_n), \tag{16}$$

$$\mathbf{p} \equiv \Big(p(y_1 = 1|\mathbf{x}^{(1)}), \ldots, p(y_n = 1|\mathbf{x}^{(n)})\Big) \tag{17}$$

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

B. 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.

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,

$$\sum_{i=1}^{N} w_i p_i = \mathbf{w}^T\mathbf{p}. \tag{18}$$

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

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

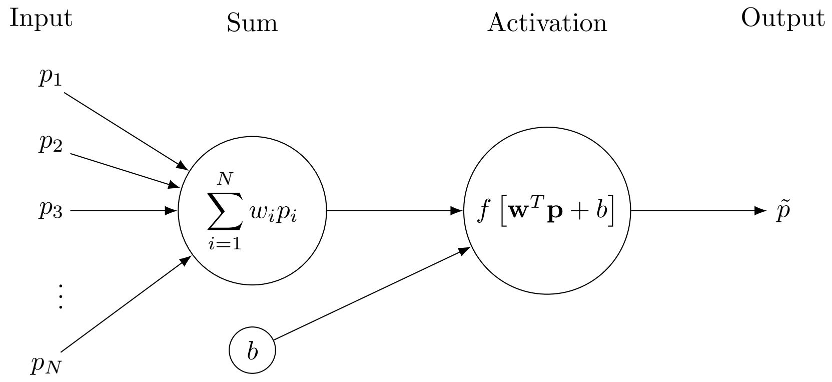

FIG. 1. A model neuron, a constituent part of the artificial neural network model. The input from the previous layer $\mathbf{p}$ multiplied by corresponding weights $\mathbf{w}$ and summed. Then the bias b is added, and the activation function f is applied to the resulting $\mathbf{w}^T\mathbf{p} +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

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 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

$$\text{input } \to f\left(\mathbf{w}^T\mathbf{p} + b\right) = \tilde{p} \ \to \ \text{output}. \tag{19}$$

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,

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

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

$$\mathbf{y}^k = f(W^k\mathbf{y}^{k-1} + \mathbf{b}^k) = f\left(\begin{bmatrix} w_{11}^k & w_{12}^k & \cdots & w_{1N}^k \\ w_{21}^k & w_{22}^k & \cdots & w_{2N}^k \\ \vdots & \vdots & \ddots & \vdots \\ w_{N1}^k & w_{N2}^k & \cdots & w_{NN}^k \end{bmatrix} \begin{bmatrix} y_1^{k-1} \\ y_2^{k-1} \\ \vdots \\ y_N^{k-1} \end{bmatrix} + \begin{bmatrix} b_1^k \\ b_2^k \\ \vdots \\ y_N^k \end{bmatrix} \right) \tag{20}$$

or in Einstein notation (no sum over k implied)

$$y_i^k = f\big(w_{ij}^k y_j^{k-1} + b_i^k\big). \tag{21}$$

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

It is clear from Eq. (20) 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

$$\hat{y}=\sum_{j=1}^{M} w_{1j}^L f_L\left(\sum_{k=1}^{M} w_{jk}^{L-1} f_{L-1}\left(\sum_{i=1}^{M} w_{ki}^{L-2} f_{L-2}\left(\dots f_1\Big(w_{m1}^1 x_1 + b_m^1\Big)\dots\right) + b_i^{L-2}\right) + b_k^{L-1}\right) + b_1^L\,. \tag{22}$$

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 $f_L, f_{L-1}, \dots, f_1$. From looking at Eq. (22), 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*.[2] 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).[3]

C. 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

$$f_\mathrm{H}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}, \tag{23}$$

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,

$$f_\mathrm{S}(x) = \frac{1}{1+\mathrm{e}^{-x}}, \tag{24}$$

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

$$f_\mathrm{t}(x) = \tanh(x) = \frac{\mathrm{e}^x - \mathrm{e}^{-x}}{\mathrm{e}^x + \mathrm{e}^{-x}}. \tag{25}$$

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

$$f_\mathrm{ReLU}(x) = \max(0, x), \tag{26}$$

and is popular mostly because of its application to training *deep* (many, large layers) neural networks.[4] Many variants of the ReLU exist, among

[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).

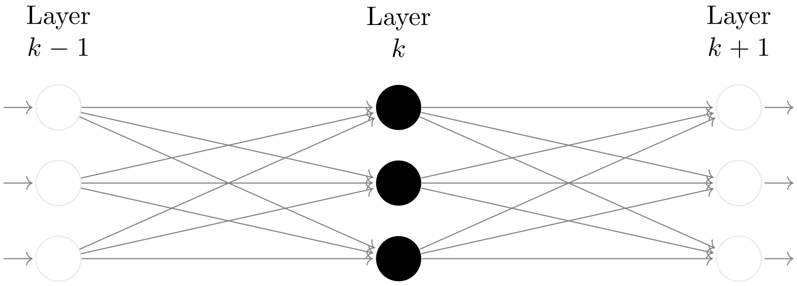

FIG. 2. Schematic representation of a single ANN layer. Each neuron of the layer indexed k is connected from behind to all neurons in layer $k-1$. The connection weights can be organized into a matrix, W^{k-1}, and the action of layer k can be succinctly stated as $f(W^k\mathbf{p}^{k-1}+\mathbf{b}^k)$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

the most well known are the leaky ReLU,

$$f_{\text{leaky ReLU}}(x;\alpha)=\begin{cases}x & \text{if } x\geq 0\\ \alpha x & \text{if } x<0,\end{cases} \tag{27}$$

the noisy ReLU

$$f_{\text{NReLU}}(x)=\max(0,x+\mathcal{N}(0,\sigma)), \tag{28}$$

and the exponential linear unit

$$f_{\text{ELU}}(x;\alpha)=\begin{cases}x & \text{if } x\geq 0\\ \alpha(\mathrm{e}^x-1) & \text{if } x<0.\end{cases} \tag{29}$$

D. 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

$$g_{ij}^k\equiv\frac{\partial C(\hat{\mathbf{y}})}{\partial w_{ij}^k},\qquad\text{and}\qquad h_i^k\equiv\frac{\partial C(\hat{\mathbf{y}})}{\partial b_i^k}. \tag{30}$$

The most common algorithm for computing these derivatives is the **backpropagation** algorithm.[5] 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.

E. Backpropagation

Before we can apply the backpropagation algorithm, we need to perform a forward pass of the

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

network given some input vector $\mathbf{x} \in \mathbb{R}^{n_{\rm f}}$, where $n_{\rm f}$ 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.

$$a_j^l = f_l(z_j^l), \tag{31}$$

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

$$z_j^1 = \sum_{i=1}^{n_{\rm f}} w_{ij}^1 x_i + b_j^1. \tag{32}$$

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

$$z_j^l = \sum_{i=1}^{N_{l-1}} w_{ij} x_i + b_j^l, \tag{33}$$

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,

$$\begin{aligned}\frac{\partial C(W^L)}{\partial w_{jk}^L} &= \frac{\partial C(W^L)}{\partial a_j^L}\left[\frac{\partial a_j^L}{\partial w_{jk}^L}\right] \\ &= \frac{\partial C(W^L)}{\partial a_j^L}\left[\frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L}\right] \\ &= \frac{\partial C(W^L)}{\partial a_j^L} f_L'(z_j^L) a_k^{L-1}, \end{aligned} \tag{34}$$

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

$$\begin{aligned}\frac{\partial C(W^L)}{\partial a_j^L} &= \frac{\partial}{\partial a_j^L}\left[\frac{1}{2}\sum_{i=1}^{N_{\rm O}} (a_j^L - t_j)^2\right] \\ &= a_j^L - t_j, \end{aligned} \tag{35}$$

and

$$\begin{aligned}\frac{\partial z_j^L}{\partial w_{jk}^L} &= \frac{\partial}{\partial w_{jk}^L}\left[\sum_{p=1}^{N_L} w_{jp}^L a_j^{L-1} + b_j^L\right] \\ &= a_j^{L-1}. \end{aligned} \tag{36}$$

We define the quantity in Eq. (34) (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

$$\begin{aligned}\delta_j^L &= \frac{\partial C(W^L)}{a_j^L}\frac{\partial f^L}{\partial z_j^L} = \frac{\partial C(W^L)}{a_j^L}\frac{\partial a_j^L}{\partial z_j^L} \\ &= \frac{\partial C(W^L)}{\partial z_j^L} = \frac{\partial C(W^L)}{\partial b_j^L}\frac{\partial b_j^L}{\partial z_j^L} && (37) \\ &= \frac{\partial C(W^L)}{\partial b_j^L}, && (38)\end{aligned}$$

where we used that

$$\begin{aligned}\frac{\partial b_j^L}{\partial z_j^L} &= \left[\frac{\partial z_j^L}{\partial b_j^L}\right]^{-1} \\ &= \left[\frac{\partial}{\partial b_j^L}\sum_{i=1}^{N_{L-1}} w_{ij}^L a_i^{L-1} + b_j^L\right]^{-1} \\ &= 1. \end{aligned} \tag{39}$$

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

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \tag{40}$$

holds for any layer, not just the output as in Eq. (37). Relating this to derivatives w.r.t. the layer $l+1$ z_js, we find

$$\begin{aligned}\delta_j^l = \frac{\partial C}{\partial z_j^l} &= \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{z_j^l} \\ &= \sum_k \delta_k^{l+1}\frac{\partial z_k^{l+1}}{z_j^l} \\ &= \sum_k \delta_k^{l+1} w_{kj}^{l+1}\frac{\partial f^l}{\partial z_j^l}, \end{aligned} \tag{41}$$

with

$$\begin{aligned}\frac{\partial z_k^{l+1}}{\partial z_j^l} &= \frac{\partial}{\partial z_j^l}\left[\sum_{i=1}^{N_l} w_{ik}^{l+1} a_k^l + b_k^{l+1}\right] \\ &= \frac{\partial}{\partial z_j^l}\left[\sum_{i=1}^{N_l} w_{ik}^{l+1} f^l(z_k^l) + b_k^{l+1}\right] \\ &= w_{jk}^{l+1} f^l(z_j^l). \end{aligned} \tag{42}$$

The rest of the backpropagation scheme is essentially iterating Eq. (41), 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.

F. Exploding / vanishing gradients and weight initialization

He initialization.[6] Xavier initialization.[7]

G. 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

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma_k \nabla F(\mathbf{x}_k), \quad \gamma_k > 0 \tag{43}$$

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

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma_k \nabla F(\mathbf{x}_k), \quad k \geq 0. \tag{44}$$

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 $\{\mathbf{x}_i\}_{i=1}^n$,

$$C(\theta) = \sum_{i=1}^{n} c_i(\mathbf{x}_i, \theta). \tag{45}$$

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

$$\nabla_\theta C(\theta) = \sum_{i}^{n} \nabla_\theta c_i(\mathbf{x}_i, \theta). \tag{46}$$

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

$$\begin{aligned}\nabla_\theta C(\theta) &= \sum_{i=1}^{n} \nabla_\theta c_i(\mathbf{x}_i, \theta) \\ &\to \sum_{i \in B_k}^{n} \nabla_\theta c_i(\mathbf{x}_i, \theta). \end{aligned} \tag{47}$$

Thus a gradient descent step now looks like

$$\theta_{j+1} = \theta_j - \gamma_j \sum_{i \in B_k}^{n} \nabla_\theta c_i(\mathbf{x}_i, \theta) \tag{48}$$

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

$$\gamma_j(t; t_0, t_1) = \frac{t_0}{t + t_1} \tag{49}$$

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.[8] 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.[9]

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]) 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.

III. 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 show phase transition. In particular, these phases can be labeled as ordered or disordered. Thus it is interesting to investigate whether logistic regression or neural networks can be trained to to classify the phases.

H. The one-dimensional Ising model

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

I. The two-dimensional Ising model

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

IV. RESULTS AND DISCUSSION

Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Donec odio elit, dictum in, hendrerit sit amet, egestas sed, leo. Praesent feugiat sapien aliquet odio. Integer vitae justo. Aliquam vestibulum fringilla lorem. Sed neque lectus, consectetuer at, consectetuer sed, eleifend ac, lectus. Nulla facilisi. Pellentesque eget lectus. Proin eu metus. Sed porttitor. In hac habitasse platea dictumst. Suspendisse eu lectus. Ut mi mi, lacinia sit amet, placerat et, mollis vitae, dui. Sed ante tellus, tristique ut, iaculis eu, malesuada ac, dui. Mauris nibh leo, facilisis non, adipiscing quis, ultrices a, dui.

J. Learning the one-dimensional Ising Hamiltonian

Morbi luctus, wisi viverra faucibus pretium, nibh est placerat odio, nec commodo wisi enim eget quam. Quisque libero justo, consectetuer a,

[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,

$$\mathrm{Var_u}(f) = \mathrm{E}\left[\left(f\right)^2\right] \neq \mathrm{E}\left[\left(f - \langle f \rangle\right)^2\right] = \mathrm{Var}(f). \quad (50)$$

```
    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, $m_j \leftarrow \beta_1 m_{j-1} + (1 - \beta_1) g_j$.
(4) Update biased second moment estimate, $v_j \leftarrow \beta_2 v_{j-1} \quad + (1 - \beta_2) g_j^2$.
(5) Compute the bias-corrected first moment esimate, $\hat{m}_j \leftarrow \frac{m_j}{1-\beta_1^j}$.
(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.

feugiat vitae, porttitor eu, libero. Suspendisse sed mauris vitae elit sollicitudin malesuada. Maecenas ultricies eros sit amet ante. Ut venenatis velit. Maecenas sed mi eget dui varius euismod. Phasellus aliquet volutpat odio. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Pellentesque sit amet pede ac sem eleifend consectetuer. Nullam elementum, urna vel imperdiet sodales, elit ipsum pharetra ligula, ac pretium ante justo a nulla. Curabitur tristique arcu eu metus. Vestibulum lectus. Proin mauris. Proin eu nunc eu urna hendrerit faucibus. Aliquam auctor, pede consequat laoreet varius, eros tellus scelerisque quam, pellentesque hendrerit ipsum dolor sed augue. Nulla nec lacus.

K. Classifying phases of the two-dimensional Ising model

Suspendisse vitae elit. Aliquam arcu neque, ornare in, ullamcorper quis, commodo eu, libero. Fusce sagittis erat at erat tristique mollis. Maecenas sapien libero, molestie et, lobortis in, sodales eget, dui. Morbi ultrices rutrum lorem. Nam elementum ullamcorper leo. Morbi dui. Aliquam sagittis. Nunc placerat. Pellentesque tristique sodales est. Maecenas imperdiet lacinia velit. Cras non urna. Morbi eros pede, suscipit ac, varius vel, egestas non, eros. Praesent malesuada, diam id pretium elementum, eros sem dictum tortor, vel consectetuer odio sem sed wisi.

V. 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.

REFERENCES

[1] Morten Ledum. "A Computational Environment for Multiscale Modelling". MA thesis. 2017.

[2] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8.

[3] J.M. McDonald and N.A. Weiss. *A course in Real Analysis*. 2nd ed. Academic Press, 2013. ISBN: 0-12-387774-1.

[4] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 2011, pp. 315–323.

[5] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323 (Oct. 1986).

[6] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[7] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[8] Ning Qian. "On the momentum term in gradient descent learning algorithms". In: *Neural Networks* 12.1 (1999), pp. 145–151. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(98)00116-6.

[9] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).