Astronomy & Astrophysics manuscript no. project2 ©ESO 2022
November 17, 2022

Classification and Regression: SGD, Neural Networks and Logistic Regression

Nanna Bryne[1, 2] and Johan Mylius Kroken[1, 2]

[1] Institute of Theoretical Astrophysics (ITA), University of Oslo, Norway
[2] Center for Computing in Science Education (CCSE), University of Oslo, Norway

November 17, 2022 GitHub repo link: `https://github.com/Johanmkr/FYS-STK4155colab/tree/main/project2`

ABSTRACT

We build a versatile neural network code in order to perform linear regression and binary classification tasks. We train the network by minising the loss function by performing plain and stochastic gradient descent (SGD) for a variety of optimisation algorithms. SGD with RMSProp optimiser perform best and is used in training. A network with 1 hidden layer of 30 neurons where $\eta = 10^{-1}$ and $\lambda = 10^{-4}$ which uses the sigmoid activation function trained for 700 epochs with 2 minibatches yield the best test MSE of 0.052 when trained to fit the noise Franke function, compared to an MSE of 0.15 for OLS. For the binary classification task the data is the Wisconsin Breast Cancer data. A neural network of 2 hidden layers of 10 neurons each where $\eta = 10^{-3}$ and $\lambda = 10^{-6}$ which uses the ReLU activation function trained for 900 epochs with 5 minibatches yield the best accuracy of 1. Logistic regression with $\eta = 10^{-3}$ and $\lambda = 10^{-8}$ also yield an accuracy of 1.

Contents

Notation and nomenclature

Datasets and outputs

$\mathcal{D}$	Dataset $\{X, \mathbf{y}\}$ of length n on the form $\{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \dots, (\mathbf{x}^{(n)}, y^{(n)})\}$
f	Function $f(\boldsymbol{\theta}; \mathbf{x})$ that gives the model's output $\hat{y}$ given some input $\mathbf{x}$
$\mathcal{L}$	Total loss function $\mathcal{L}(\hat{y}, y)$ where $\hat{y} = f(\boldsymbol{\theta}; \mathbf{x})$ is the output from our model (often written as $\mathcal{L}(\boldsymbol{\theta})$ for ease of notation)
n	$\in \mathbb{N}$; Number of samples in a dataset
p	$\in \mathbb{N}$; Number of features of the dependent variables in a dataset
$\mathbf{x}^{(i)}$	$\in \mathbb{R}^p$; The i^{th} example of the dependent variable $\mathbf{x}$
X	$\in \mathbb{R}^{n\times p}$; Feature matrix of n row vectors $\mathbf{x}$
$y^{(i)}$	$\in \mathbb{R}$; The i^{th} example of the independent variable y associated with $\mathbf{x}^{(i)}$
$\hat{y}^{(i)}$	$\in \mathbb{R}$; The output associated with $(\mathbf{x}^{(i)}, y^{(i)}) \in \mathcal{D}$
$\boldsymbol{\theta}$	$\in \mathbb{R}^p$; Parameter vector, or vector of coefficients

Syntax for steepest descent algorithms

$\mathbf{v}$	$\in \mathbb{R}^p$; Momentum in parameter space ($\mathbf{v}_k$ denotes the last update to $\boldsymbol{\theta}_k$)
$\mathcal{A}$	$= \nabla_\theta \mathcal{L} \in \mathbb{R}^p$; Magnitude and direction of steepest ascent in parameter space ($\mathcal{A}_k = \nabla_\theta \mathcal{L}(\boldsymbol{\theta}_k)$)

Network components

$\mathbf{a}$	$\in \mathbb{R}^{N_l}$; Activation argument of layer l
$\mathbf{b}^l$	$\in \mathbb{R}^{N_l}$; Bias term of neurons in layer l
g_l	$: \mathbb{R}^{N_l} \to \mathbb{R}^{N_l}$; Activation function associated with layer l, an affine transformation

$\mathbf{h}^l$	$\in \mathbb{R}^{N_l}$; Information associated with layer l
N_l	$\in \mathbb{N}$ Number of neurons in layer l
$W^{l\to l+1}$	$\in \mathbb{R}^{N_l \times N_{l+1}}$; Matrix of weights describing the mapping from layer l to layer $l+1$ ($W^{l+1\leftarrow l} \equiv [W^{l\to l+1}]^\intercal$)

Hyperparameters

L	Number of layers in an NN, not counting the input layer, or label of output layer
m	Number of minibatches in SGD
γ	Momentum factor (constant term)
η	Learning rate (global)
λ	Regularisation parameter (penalty parameter in Ridge regression)
ϱ_1, ϱ_2	Hyperparameters related to RMSProp ($\varrho \equiv \varrho_2$) and Adam

Miscellaneous

$\mathcal{N}(\mu, \sigma)$	Normal distribution with mean μ and standard deviation σ
$\text{Var}(A)$	Variance of A
$\nabla_{\xi} J$	Gradient of a function J with respect to $\boldsymbol{\xi}$
$\odot$	Elementwise Hadamard product

Acronyms

DAG	Directed acyclic graph
FFNN	Feedforward neural network
GD	Gradient descent
MSE	Mean squared error
NN	Neural network
OLS	Ordinary least squares
ReLU	Rectified linear unit
SGD	Stochastic gradient descent

1. Introduction

Supervised learning problems dealing with regression and classification both benefit, or even rely on, optimisation methods for locating minima of some loss function. A commonly used solution in machine learning is the stochastic gradient descent algorithm with its many subvariants. We aim to investigate these subvariants in their right on an arbitrary testing polynomial. The result of this investigation will be paramount in the further development of a feedforward neural network that will be used both for linear and logistic regression problems, but also binary classification problems. It can also be extended to deal with multivariate classification problem, although this is not done here.

Section 2 presents the theoretical background of the following analysis in section 3. Above lies the nomenclature of this paper, for reference. We summarise our results section 4. In more detail, in section 2.1 we describe the main ideas behind the methods of steepest descent. We move on to describe some basic theory concerning neural networks and the structure of a feedforward neural network in section 2.2. We connect this to linear regression in section 2.3, classification in 2.4 and logistic regression in 2.5. In section 2.6 we briefly state how models like these are validated in this paper. The analysis part is initiated with a simple regression problem in 3.1 concerning the steepest descent methods. We present how we build our neural network in 3.2, which we use to solve the more complex regression problem in section 3.3. We move on to a binary classification problem which we analyse with our neural network in section 3.4 and with logistic regression in section 3.5. We present some closing thoughts in section 3.6 before we summarise our main results in section 4.

There are three appendices to this report, A, B and C, all of which support our analysis through a number of figures. Additional results can be found in the figure folder on our Github repository, for the interested reader.

2. Theory

Linear regression assumes a linear relationship between a set of p features $\mathbf{x} \in \mathbb{R}^p$ and an observed value $y \in \mathbb{R}$. We assume there exists a *continous* function of the input $\mathbf{x}$ giving the output $\hat{y}$. The coefficients $\boldsymbol{\theta} \in \mathbb{R}^p$ that determine said function can be estimated using a variety of methods, as discussed in project 1. In any case, the aim is to minimise some loss function $\mathcal{L}(\boldsymbol{\theta}; \hat{y}, y)$ with respect to this parameter vector ($\boldsymbol{\theta}$) describing what we sacrifice by using this exact model.

What if the function we want to fit is *discontinous*? We consider the binary situation where the observed y only takes one of two discrete values; 0 or 1. Logistic regression proposes a model where the output $\hat{y}$ is obtained from a probability distribution and subsequently a befitting total loss function that can be minimised with respect to a set of parameters $\boldsymbol{\theta}$. Now, instead of using the method-specific regression algorithms for finding the optimal $\boldsymbol{\theta}$, we can change modus operandi and focus solely on the minimisation of some objective function (e.g. a loss function). For this purpose, we may use the very powerful procedure of steepest descent.

Where the actual (physical) relationship between some dependent and independent variable is not of utmost importance, and the main aim is to predict the outcome given some setting, supervised learning problems may also be solved using neural networks.

2.1. Stochastic gradient descent (SGD)

SGD and its subvariants are frequently used optimisation algorithms in machine learning (**?**). The more basic algorithm known as gradient descent (GD) is technically a specific case of SGD[1] follows the gradient of some objective function J downhill in some parameter space. The effect of introducing stochasticity is considered in section 2.1.3. The result is a flexible way to locate the minima of any smooth and differentiable J, exactly what we wished for. The ordinary least squares and Ridge schemes of linear regression that we discussed in project 1, are then implemented by using the mean squared error (MSE) function with an ℓ^2-norm regularisation term,

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{2n}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2 + \frac{\lambda}{2p}\sum_{j=1}^{p}\theta_j^2. \tag{1}$$

λ is the penalty term set to zero for OLS and a small positive value for Ridge regression. The output $\hat{y}^{(i)}$ is resulting from some function evaluated at $\mathbf{x}^{(i)}$, which for our purposes reads $\hat{y}^{(i)} = [\mathbf{x}^{(i)}]^\intercal\boldsymbol{\theta}$. In the following, said function is denoted $f = f(\mathbf{x};\, \boldsymbol{\theta})$.

[1] The case with number of minibatches $m = 1$.

2.1.1. Plain gradient descent

The most basic concept is that of steepest descent. In order to find a minimum of a function $J = J(\boldsymbol{\xi})$, we follow the steepest descent of that function in $\boldsymbol{\xi}$-space, i.e. the direction of the negative gradient $-\nabla_\xi J(\boldsymbol{\xi})$. We thus have the iterative scheme to find minima,

$$\boldsymbol{\xi}_{k+1} = \boldsymbol{\xi}_k - \eta_k \nabla_\xi J(\boldsymbol{\xi}_k), \tag{2}$$

where the learning rate η_k may follow a schedule in k or stay constant. In the following, we consider a constant global[2] learning rate $\eta_k = \eta$.

What we would like to minimise is the cost function $\mathcal{L}(\boldsymbol{\theta})$ which is a function of the parameters $\boldsymbol{\theta}$ which we are trying to estimate. If we define $\mathcal{A}_k \equiv \nabla_\theta \mathcal{L}(\boldsymbol{\theta}_k)$ to be the direction and magnitude of the steepest ascent in parameter space, eq. (2) reads

$$\begin{aligned}\boldsymbol{\theta}_{k+1} &= \boldsymbol{\theta}_k + \mathbf{v}_k\,;\\ \mathbf{v}_k &= -\eta\mathcal{A}_k,\end{aligned} \tag{3}$$

for substitutions $J \to \mathcal{L}$ and $\boldsymbol{\xi} \to \boldsymbol{\theta}$. For a sufficiently small η, this method will converge to a minimum of $\boldsymbol{\theta}$. However, since we may not know the nature of $\mathcal{L}$ in parameter space, there is a risk that said extremum is just a local and not a global minimum. The steepest descent method in eq. (3) is a deterministic method, which means we may get stuck in a local minimum. There are several ways around this, and one such way is to include an element of randomness in the computations, as we will see in section 2.1.3.

2.1.2. Momentum

From eq. (3) we have that the movement in parameter space is given by $\mathbf{v}_k$, which describes the direction and magnitude of the steepest descent in parameter space. Sometimes we might want to move larger distances in one step. This can be achieved by introduction momentum: We add an addition term to $\mathbf{v}_k$ which lets us rewrite eq. (3) as

$$\begin{aligned}\boldsymbol{\theta}_{k+1} &= \boldsymbol{\theta}_k + \mathbf{v}_k\,;\\ \mathbf{v}_k &= \gamma\mathbf{v}_{k-1} - \eta\mathcal{A}_k,\end{aligned} \tag{4}$$

where γ is a momentum parameter and η is the same global learning parameter as before. The basic idea is that with this method we "overshoot" the descending step length in the direction of the previous step, with a magnitude that is controlled by γ. By doing this, we may reach the desired minimum with fewer iterations.

2.1.3. Stochasticity

There are several weaknesses to the plain gradient descent, perhaps the largest is the computational expense of large datasets and its sensitivity of initial conditions and learning rates. If $\mathcal{L}(\boldsymbol{\theta})$ has numerous local minima, we will find one minimum only per set of initial conditions, and we have no good way of saying whether this minimum is global or not. One way of overcoming this is by adding stochasticity to the gradient descent algorithm.

[2] Emphasising "global" here to distinguish from the *actual* rate of learning (or step size) which may depend on the specific update rule we choose.

The main idea is that with the n data points which we have in a dataset $\mathcal{D}$, we can create m subsets, meaning that we have n/m [3] data points in each *minibatch*, denoted $\mathcal{B}_j$ for $j \in \{1, 2, \dots, m\}$, s.t. $\bigcup_{j=1}^m \mathcal{B}_j = \mathcal{D}$ and $\bigcap_{j=1}^m \mathcal{B}_j = 0$. We recognise that we may write the total cost function as a sum over all data points $\mathbf{x}^{(i)}$ for $i \in [1, n]$,

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{(\mathbf{x},y)\in\mathcal{D}} l\big(f(\mathbf{x};\,\boldsymbol{\theta}), y\big) = \sum_{i=1}^{n} l_i(\boldsymbol{\theta}), \tag{5}$$

where $l_i(\boldsymbol{\theta}) = l\big(f(\mathbf{x}^{(i)};\boldsymbol{\theta});\, y^{(i)}\big)$ is the per-example loss function. Thus, its gradient is written

$$\mathcal{A} = \nabla_\theta \mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{n} \nabla_\theta l_i(\boldsymbol{\theta}). \tag{6}$$

Now we may approximate the gradient of the cost function by only summing over the data points in a minibatch picked at random:

$$\begin{aligned}\mathcal{A}_k &= \sum_{j=1}^{m} \mathcal{A}_k^j\,;\\ \mathcal{A}_k \to \mathcal{A}_k^j &= \sum_{i:\mathbf{x}^{(i)}\in\mathcal{B}_j} \nabla_\theta l_i(\boldsymbol{\theta}_k)\,;\end{aligned} \tag{7}$$

The estimate $\mathcal{A}_k^j \approx \mathcal{A}_k$ can be used in our algorithm to ensure stochasticity and relieve computational pressure.

2.1.4. Optimising the learning rate

There is in general no way of knowing a priori what value η or γ should take. Tuning such hyperparameters makes up a significant part of the work in a supervised learning problems. We can mitigate parts of the struggle with hyperparameter adjustment by using an algorithm with adaptive learning rates. We present a few such schemes in short below, that is different ways of calculating $\mathbf{v}$ in eq. (3). All require an original learning rate η and a small number ε for numerical stability.

AdaGrad (**?**, algorithm 8.4), from "adaptive gradient (algorithm)", adapts η individually to the components of $\boldsymbol{\theta}$, scaling them as $\eta \to \eta' \sim {}^{\eta}/_{\nabla_\theta \mathcal{L}}$ (mind the simplification). This method is famously a trouper in convex settings, but has the unfortunate habit of prematurely deciding on the best model in a nonconvex setting. By default: $\varepsilon = 10^{-7}$.
RMSProp (**?**, algorithm 8.5), from "root mean squared propagation", is a modification to the AdaGrad algorithm. Where AdaGrad performs good, this one learns slow in comparison. However, RMSProp outperforms AdaGrad in nonconvex situations. This improvement introduces an additional hyperparameter ϱ, a decay rate controlling the length scale of the moving average. By default: $\varrho = 0.9$ and $\varepsilon = 10^{-7}$.
Adam (**?**, algorithm 8.7), from "adaptive moments", calculates the update based on both the first-order momentum, the same as in momentum, and the second-order momentum, much like in RMSProp. We need two hyperparameters ϱ_1 and ϱ_2 for this optimisation scheme, representing the decay rate of the first and second moment, respectively. By default: $\varrho_1 = 0.9$, $\varrho_2 = 0.999$ and $\varepsilon = 10^{-8}$.

The choice of default values is inspired by (**?**, chapter 8.5).

[3] Give or take; needs to be an integer.

2.2. Neural Network (NN)

We have so far discussed regression and given a lot of attention to the coefficients $\boldsymbol{\theta}$ that we assume describe some physical relationship between a set of feature values and a target. More complex problems require more complex models, and many phenomena may not even be possible to describe with a smooth function. Deep learning models usually pay more attention to the output, in this section denoted $\hat{\mathbf{y}}$ to allow multivariable outputs[4], after training, that is.

2.2.1. Basics

A feedforward NN (FFNN) is typically built by composing together several functions into a chain of function. Associated with this model is a directed acyclic graph (DAG) describing the explicit structure. The depth of the model is determined by the length of the abovementioned chain. Each function represents a layer in the network. The final layer of an FFNN is the output layer, and the layers between the input (prior to the first) and the output layer are called hidden layers. (?)

The structure of such a chain-based architecture is described by the $L-1$ hidden layers $\mathbf{h}^l \in \mathbb{R}^{N_l}$, $l = 1, 2, \dots, L-1$, given by

$$\mathbf{h}^0 = \mathbf{x}^{(i)}\,; \tag{8a}$$
$$\mathbf{h}^1 = g_1\big((W^{0\to 1})^\intercal \mathbf{h}^0 + \mathbf{b}^1\big)\,; \tag{8b}$$
$$\mathbf{h}^2 = g_2\big((W^{1\to 2})^\intercal \mathbf{h}^1 + \mathbf{b}^2\big)\,; \tag{8c}$$
$$\vdots$$
$$\mathbf{h}^L = g_L\big((W^{L-1\to L})^\intercal \mathbf{h}^{L-1} + \mathbf{b}^L\big)\,; \tag{8d}$$

where we defined $\mathbf{h}^0$ and $\mathbf{h}^L$ to be the input and output layer, respectively.

The matrix of weights $W^{l-1\to l} \in \mathbb{R}^{N_{l-1}\times N_l}$ applies weights and dimension corrections to the previous layer $\mathbf{h}^{l-1} \in \mathbb{R}^{N_{l-1}}$ so that the activation function $g_l : \mathbb{R}^{N_l} \to \mathbb{R}^{N_l}$ can accept the input. The bias $\mathbf{b}^l \in \mathbb{R}^{N_l}$ may be interpreted as a safety mechanism of the neurons to prevent their layer value to become zero, and is typically set to a small nonzero value (?).

2.2.2. Activation functions

A layer $\mathbf{h}^l$ has an associated activation $\mathbf{a}^l \in \mathbb{R}^{N_l}$ which is a function of the previous layer values, $\mathbf{h}^{l-1}$, the weights, $W^{l-1\to l}$, and the biases linked with each neuron, $\mathbf{b}^l$. The activation is passed as argument to the activation function g_l whose job is to perform the affine transformation from one layer to another in a NN. In eq. (8) the activation is $\mathbf{a}^l = (W^{l-1\to l})^\intercal \mathbf{h}^{l-1} + \mathbf{b}^l$, which is valid for $l = 1, 2, \dots L$. Note that the weight matrix $W^{l-1\to l}$ is associated with both the current and previous layer. We can rewrite the formula in eq. (8) as the more compact expression:

$$\mathbf{h}^0 = \mathbf{x}^{(i)}\,, \quad \mathbf{h}^l = g_l(\mathbf{a}^l)\,,\ l = 1, 2, \dots L\,;$$
$$\mathbf{a}^l = W^{l\leftarrow l-1}\mathbf{h}^{l-1} + \mathbf{b}^l\,; \tag{9}$$

where $W^{l\leftarrow l-1} \equiv (W^{l-1\to l})^\intercal$. The output is $\hat{\mathbf{y}}^{(i)} = \mathbf{h}^L \in \mathbb{R}^{N_L}$.

We present some examples of commonly used activation functions:

$$\sigma(\boldsymbol{\xi}) = \frac{1}{1+e^{-\boldsymbol{\xi}}} = 1 - \sigma(-\boldsymbol{\xi}) \tag{10a}$$
$$\tanh(\boldsymbol{\xi}) = \frac{e^{2\boldsymbol{\xi}}-1}{e^{2\boldsymbol{\xi}}+1} = 2\sigma(2\boldsymbol{\xi}) - 1 \tag{10b}$$
$$\mathrm{ReLU}(\boldsymbol{\xi}) = \max(0, \boldsymbol{\xi}) = \begin{cases} \boldsymbol{\xi}, & \boldsymbol{\xi} > 0 \\ 0, & \boldsymbol{\xi} \le 0 \end{cases} \tag{10c}$$
$$\mathrm{ReLU}^*(\boldsymbol{\xi}) = \begin{cases} \boldsymbol{\xi}, & \boldsymbol{\xi} > 0 \\ 0.01\boldsymbol{\xi}, & \boldsymbol{\xi} \le 0 \end{cases} \tag{10d}$$

Assuming some $\boldsymbol{\xi} \in \mathbb{R}^K$, the set of expressions (10) show some well-known activation functions $\mathbb{R}^K \to \mathbb{R}^K$. To translate into NN components, set $K \to N$, $\boldsymbol{\xi} \to \mathbf{a}$ and e.g. $g = \tanh$. The oldest and probably most famous is the slow-learning sigmoid function σ in eq. (10a). The hyperbolic tangent in eq. (10b) is closely related to the sigmoid, and is typically performing better (?). The ReLU (eq. (10c)) or leaky ReLU (eq. (10d)) activation function provides output of the type that is easy to interpret as it resembles the linear unit. ReLU typically learns fast, but has the the unfortunate habit of killing neurons. That is to say, some neurons are deactivated for any input. The leaky ReLU can omit this issue somewhat, but the hatch is a perfomance reduction.

2.2.3. Initialisation

Before we even start to calculate anything with our network we need to know how it should be initialised. For the bias, this is straightforward as all the $\mathbf{b}^l$, $l = 1, 2, \dots, L$ are initialised with a constant small though nonzero value. For the weights on the other hand, the initialisation depends on the choice of activation functions. All weight initialisations are drawn from normal distribution, but limits are put on the variance of these distributions. For the sigmoid and hyperbolic tangent activation functions, weights are drawn from a normal distribution centred around zero, with variance

$$\mathrm{Var}\left[W^{l-1\to l}\right] = \frac{2}{N_{l-1}+N_l}. \tag{11}$$

Similarly, for the ReLU and leaky ReLU activation function, the variance of the normal distribution centred around zero must satisfy

$$\mathrm{Var}\left[W^{l-1\to l}\right] = \frac{2}{N_{l-1}}. \tag{12}$$

We have that eq. (11) is known as *Xavier* initialisation, while eq. (12) is known as *He* initialisation (?).

2.2.4. Back propagation

The information in an FFNN accepting input $\mathbf{x}$ to produce output $\hat{\mathbf{y}}$ is flowing *forward* (?), hence the name. The initial information from $\mathbf{x}$ propagates through the hidden layers resulting in the production of $\hat{\mathbf{y}}$ which is the output of the final layer. This information flow is called forward propagation or forward pass. Training the network (tuning the

[4] For completeness. Will not be relevant to think of the output as a vector in our analysis.

weights and biases) consists of running forward propagation and compare the resultant output $\hat{\mathbf{y}}$ with the desired output $\mathbf{y}$, i.e. evaluate the loss function, $\mathcal{L}(\boldsymbol{\theta})$.

The art of back propagation is to reverse this process. We let $\mathcal{L}(\boldsymbol{\theta})$ provide information about the error of the output layer, that propagates backwards through the network in order to compute the gradient of the loss function for each layer, $\nabla^l_\theta \mathcal{L}(\boldsymbol{\theta})$. These gradients are used to update the weights and biases of each layer in such a way that when forward propagation is run again, the overall output loss will be lower. Over time, we propagate forwards a backwards in order to minimise the loss function, typically using stochastic gradient descent, as explained in section 2.1.3. The optimiser of choice (section 2.1.4) takes the gradients found from back propagation, (and hyperparameters) as inputs and optimises the weights and biases accordingly. An explanation of the back propagation algorithm is included in appendix E. For the curious reader, a thorough walkthrough of the back propagation algorithm can be found in (**?**, chapter 6.5).

2.3. Linear regression

When using a FFNN for a regression problem we opt to fit a function to a certain dataset. One approach to this is to let the number of features be the dimensionality of the input data, and the number of data points be the data points in the domain we are considering. In this investigation we will use the two dimensional Franke function from project 1[5] on a 20×20 grid (more on this later).

Since we want to fit a function to data points, the obvious way of measured the error from the output layer is by considering the MSE. Thus, our loss function will be as given in eq. (1) with a tunable hyperparameter λ.

We may use a varying number of hidden layers and neurons, depending on the data we are trying to fit. The architecture of the network is a problem dependant feature and must examined. The same goes for the activation functions described in eq. (10). However, the output function g_L can simply be the linear function: $g_L(\mathbf{a}^L) = \mathbf{a}^L$, befitting problems such as these.

2.4. Classification

For the case of classification, we typically have many features in the input data, which results in a more complex design matrix. The output of the network is structured into two main classes, binary and multivariate classification. In binary classification we have one single output node that ideally should be either 0 or 1. We achieve this by having a sigmoid output function (eq. (10a)). In multivariate classification on the other hand, we have multiple output nodes and a probability distribution between them. This probability distribution is found using the Softmax function. However, the dataset we will analyse is the Wisconsin Breast Cancer dataset (**?**) which needs binary classification. Thus, the main focus will be on the binary classification technique.

When evaluating the loss we use eq. (15) which is known as the cross entropy or log loss function which measures the performance of a model whose output is a value between 0 and 1. Another argument for using cross entropy as loss function is that we expect the error to follow a binomial distribution, rather than a normal distribution as in linear regression.

[5] Equation (10) in the paper.

2.5. Logistic regression

The standard logistic function $q : \mathbb{R} \to (0, 1)$ may be written as

$$q(\xi) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 \xi)}}, \tag{13}$$

and is indeed the sigmoid function in eq. (10a) if we substitute $\boldsymbol{\xi} \to \beta_0 + \beta_1 \xi$ ($\beta_0, \beta_1 \in \mathbb{R}$ arbitrary constants). This can be generalised to work with multivariable inputs $\mathbf{x} \in \mathbb{R}^p$: We include an intercept θ_0 and let $\boldsymbol{\theta} = (\theta_1, \theta_2, \ldots, \theta_p)$ be the same as before. Then,

$$q(\mathbf{x};\, \theta_0, \boldsymbol{\theta}) = \frac{1}{1 + e^{-(\theta_0 + \mathbf{x}^\intercal \boldsymbol{\theta})}}, \tag{14}$$

so that $\hat{y}^{(i)} = q\big(\mathbf{x}^{(i)}\big)$. When this is the case, the loss function simplies a bit, but we present the cross entropy for a general $\hat{y}$ resulting from a model as

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) = &-\frac{1}{2n}\sum_{i=1}^{n}\Big[y^{(i)}\log \hat{y}^{(i)} + \big(1 - y^{(i)}\big)\log\big(1 - \hat{y}^{(i)}\big)\Big] \\ &+ \frac{\lambda}{2p}\sum_{j=1}^{p}\theta_j^2,\end{aligned} \tag{15}$$

except for the last term which is an ℓ^2-norm regularisation term.

Now, if we want to perform a logistic regression analysis, given that we already have a flexible algorithm FFNN, we need only create a very simple FFNN of:

* p input neurons ($N_0 = p$)
* a single output neuron ($N_L = 1$)
* no hidden layers ($L = 1$)
* sigmoid as output activation function ($g_L = \sigma$)

In doing this, following eq. (9), we end up with the output

$$\hat{y} = \sigma(W^{L \leftarrow 0}\mathbf{x} + b^L) = q(\mathbf{x};\, b^L, W^{0 \to L}), \tag{16}$$

where we see that $\hat{y}, b_L \in \mathbb{R}$ and $W^{0\to L} \in \mathbb{R}^{p\times 1}$. Thus, we have a simple way of fitting a dataset using logistic regression without creating any more algorithms. If we need the coefficients $\theta_0, \boldsymbol{\theta}$, we can obtain them as $b^L, W^{0\to L}$.

2.6. Model validation

We have discussed some loss functions, but the way we will provide the error in a model has not yet been clarified. Assume we have divided the dataset $\mathcal{D}$ of n observed values into a training set $\mathcal{D}^{\text{train}}$ and a test set $\mathcal{D}^{\text{test}}$ of respective lengths n^{train} and n^{test}, s.t. $\mathcal{D}^{\text{train}} \cup \mathcal{D}^{\text{test}} = \mathcal{D}$ and $\mathcal{D}^{\text{train}} \cap \mathcal{D}^{\text{test}} = 0$, also implying $n^{\text{train}} + n^{\text{test}} = n$.

When we speak of the total loss, this concerns the training dataset, or more often subsets of which. Since we expect its error to be normally distributed around the target values, we evaluate our regression models using the MSE function

$$\text{MSE} = \frac{1}{n^{\text{test}}} \sum_{y \in \mathcal{D}^{\text{test}}} (\hat{y} - y)^2, \tag{17}$$

where $\hat{y}$ is the output we get from our model at corresponding $\mathbf{x} \in \mathcal{D}^{\text{test}}$. The MSE is then a function of our model, e.g. of the parameter vector $\boldsymbol{\theta}$. The same goes for the binary classification problem, only here it makes sense to use the accuracy score, counting correct predictions from our binomially distributed error

$$\text{Accuracy} = \frac{1}{n^{\text{test}}} \sum_{y \in \mathcal{D}^{\text{test}}} \mathbf{1}_y(\hat{y}), \tag{18}$$

where the indicator function can be written explicitly:

$$\mathbf{1}_y(\hat{y}) = \begin{cases} 1, & \hat{y} = y \\ 0, & \hat{y} \neq y \end{cases}; \tag{19}$$

Note that we have assumed that the computed output is already interpreted to binary classification:

$$\hat{y} \to \begin{cases} 0, & 0 < \hat{y} \leq 1/2 \\ 1, & 1/2 < \hat{y} \leq 1 \end{cases}; \tag{20}$$

3. Analysis

3.1. Gradient descent

Using the SGD algorithm, we perform an OLS regression on a dataset generated by a third order polynomial with some added noise,

$$F(x) = 2.0x + 1.7x^2 - 0.40x^3 + 0.10\mathcal{N}(0,1), \tag{21}$$

and we consider $n = 400$ data points in total, but save 20% of these for validation. We use the Vandermonde matrix $X \in \mathbb{R}^{400\times3}$ of row vectors $\mathbf{x} = (x, x^2, x^3)$ $(p = 3)$ so that the output becomes $\hat{y} = X\boldsymbol{\theta}$, where $\boldsymbol{\theta} \in \mathbb{R}^3$. We also scale the data via z-score normalisation. In particular, we aim to minimise the cost function in eq. (1) in $\boldsymbol{\theta}$-space with $\lambda = 0$ for which we need to tune the learning rate η. We perform the same analysis using the Ridge cost function, i.e. $\lambda > 0$ in eq. (1), but here we need to contemplate the penalty parameter λ as well as the learning rate η.

We want to try a variety of optimiser algorithms for different η's. For this very simple case, after a variety of simulations, we realise a few key takeaways:

a) SGD is much more robust GD.
b) The effect increasing λ has on the MSE is negligable.
c) All update rules seem to find an equally good model for $\eta \in [10^{-3}, 1]$.

Mainly, these things are found after performing OLS with GD, and both OLS and Ridge regression ($\lambda = 0.1$) with SGD with $m = 40$. More results than the ones presented in this paper can be found here. Item a) is not surprising, even though we might have exaggerated with the number of minibatches we chose. Point b) is expected as the function is so simple, however, we noticed that the algorithms learned slightly faster. The last key point c) indicates that the update rules are properly implemented in our code. In

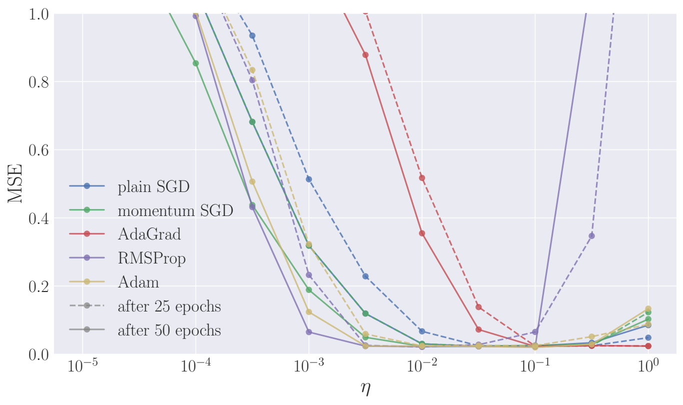


Fig. 1. The graphs show how the test MSE evolves as the global learning rate η increases for different update rules in the SGD algorithm. The penalty parameter is $\lambda = 0.1$, and we have used $m = 40$ minibatches. $\gamma = 0.5$ for the momentum SGD and the other relevant hyperparameters are set to their defaults in accordance with section 2.1.4. The dashed graphs show the MSE after 25 iterations and the solid graphs represent the MSE after 25 additional iterations. All solvers were initialised by the same random vector $\boldsymbol{\theta}_0$.

Figure 1 we present the result of the Ridge regression with SGD where we used $m = 40$ and $\lambda = 0.1$ and stopped after 50 complete epochs. The graphs are barely distinguishable from what we got with $\lambda = 0$. We see that RMSProp learns fast for small η, wheras AdaGrad seems to need a larger η to be able to converge. With momentum, the learning requires half of the iterations needed for the plain SGD to get the same MSE, a result we repeatedly have gotten for other experiments.

We will study more thoroughly the dependencies on the number of epochs, the number of minibatches (m) and the penalty parameter (λ) in the NN analysis of the Franke function in section 3.2.

3.2. Neural network

We will build our FFNN ((i)-(iii)) and solve a supervised learning problem ((iv)-(vi)) using the steps listed below (**?**).

(i) Collect and prepocess data, that is we extract 80% of the dataset and reserve the rest for validation. The data is then scaled using standard score normalisation[6] with respect to the training data.
(ii) Define the model and design its architecture. In practice, this means to decide on hyperparameters of the NN such as depth (L) and activation function(s) (g).
(iii) Choose loss function and optimiser. For regression we will use the MSE score (1) as the estimator of loss, whereas the classification problem estimates the loss according to the cross entropy (15). We will use SGD as optimiser, but we have various alternatives for the exact optimisation algorithm (see section 2.1.4).
(iv) Train the network to find the right weights and biases.
(v) Validate model, i.e. assess model performance by applying it on the test data.
(vi) Adjust hyperparameters, and if necessary review the network architecture. That is to say, if the result is not satisfactory even after tuning the hyperparameters, return to step (ii) and start over from there.

[6] Formula found in **?**, or page 6 of the project 1-report.

3.3. Regression problem

Our dataset is once again fictional as it is generated by the Franke function from project 1[7] with an added noise of $0.1\mathcal{N}(0, 1)$ for a set of coordinates in the plane. We split and standardise the 20×20 data points, which concludes step (i). Eventually we want to optimise the network architecture. However, in order to begin generating result we need an initial architecture that is not too computationally expensive, but yet versatile. We initialise the network with 3 hidden layers with 15, 10, and 5 neurons each, resulting in a depth of $L = 4$. The input layer has two nodes, $N_0 = 2$, representing the plane, whilst the output layer consist of a single node, $N_L = 1$. These features of our NN are not to be changed. We begin our analysis with the sigmoid function as activation function for the hidden layers and a linear output function. We use SGD with RMSProp as our optimiser of choice, dividing the training data into $m = 3$ minibatches. This concludes step (ii) and (iii). We then train our network for 250 epochs, which at this stage is a fair tradeoff between computational efficiency and fine tuning of the network. The trained model is tested against the test data and performance is recorded.

Now that our network is set up we are ready to tune it. The first task is to determine the hyperparameters η and λ given the architecture above. From Figure A.1 it is obvious that the current choice of architecture and training prefers relatively large learning rates and small regularisation parameters. The large error for smaller learning rates is most likely due the number of epoch being too small for these learning rates to find the minimum of the loss function. For now this result is satisfactory and we note the optimal parameters to be $\eta = 10^{-1}$ and $\lambda = 10^{-4}$.

Now we take a closer look at the architecture of the network and perform analysis of a model where we increase the number of hidden layers $L - 1$ with a fixed, but increasing number of neurons per layer N_l. Note that N_l is the same for $l = 1, 2, \ldots, L - 1$, and that still $N_0 = 2$ and $N_L = 1$. Figure A.2 shows the results, where it becomes apparent that for the optimal parameters mentioned above, a network with a simple architecture (few layers and neurons) is preferred with the optimal architecture being a network with one hidden layer that contains 30 neurons. The mean squared error is steadily decreasing, and we note the current value to be MSE $= 0.057$.

The next thing to decide is the activation function of the hidden layer. In Figure A.3 and Figure A.4 we have plotted the MSE as function of training epochs, for up to 1000 and 250 epochs respectively. For computational efficiency we want to train the network for as few epochs as necessary, but we should also check how increasing the training epochs affect the performance of the model. From these two figures we draw that the sigmoid and and hyperbolic tangent activations show the most promising results, with sigmoid being the most stable. Hence, we use sigmoid when performing the final analysis. In addition, we do not expect the model to perform significantly better if we increase the number of epochs. We also take note of the stochastic behaviour of the graphs, except for the ReLU graph. This could be since ReLU have the ability of killing neurons when its derivative is zero.

[7] Equation (10) in the report.

Having a model we need to decide on the best way of training it. Figure A.5 shows a heatmap of the number of epochs against the number of minibatches. Few minibatches and few epochs result in computational efficiency. However, there does not seem to be an overall preference. We therefore choose the best value of those presented and train the model over 700 epochs using 2 minibatches.

The final model consist of a network with one hidden layer that contains 30 neurons. The learning rate and regularisation is tuned to be $\eta = 10^{-1}$ and $\lambda = 10^{-4}$, where we use sigmoid as the activation function of the hidden layer. The model is trained over 700 epochs using 2 minibatches. This results in a test MSE of 0.052, which is significantly less than 0.15 as we got with normal OLS analysis in project 1. In order to validate our model we let the model fit a function to the original noise Franke function data. The result is shown in figure Figure 2, where the data points are shown with green triangles. As is visible in the figure, the model does quite a decent job of fitting this function to the data points for the given domain.

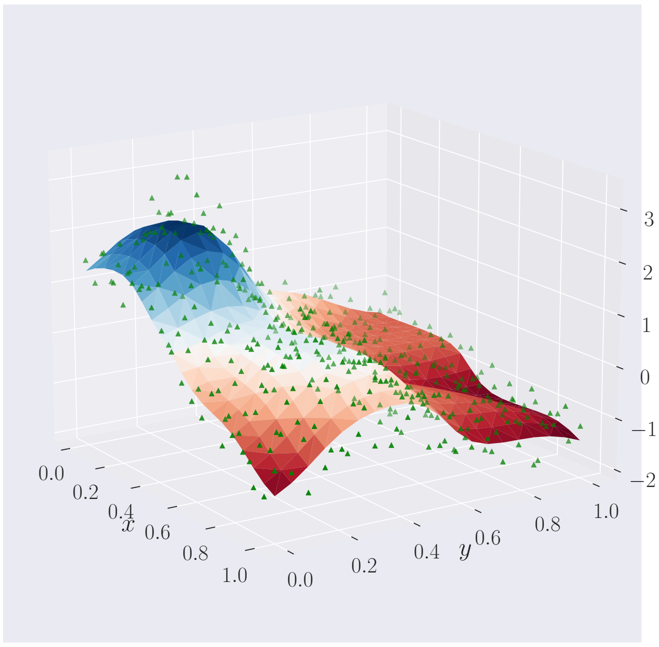


Fig. 2. Best fit of the Franke function (original data shown with green triangles). The model used has 1 hidden layer with 30 neurons, $\eta = 10^{-1}$, $\lambda = 10^{-4}$, with a sigmoid activation function trained for 700 epochs with 2 minibatches. This resulted in a test MSE of 0.052

It is perhaps surprising that the ReLU and leaky ReLU activation functions perform so badly. One could expect that these activation function should outperform the sigmoid and hyperbolic tangent activation function. A reason for the results obtained here could very well be implementation errors in the model and initialisation. In addition, the order in which we test and decide hyperparameters should perhaps also play a role. If we revert the order and say, start with a given η and λ and tune the architecture before tuning the hyperparameters the result could have been different. There are numerous ways of testing and deciding on parameters for such a network, and the procedure presented here is just one of many possibilities. In the end, the MSE obtained is lower than what we found for OLS, which is a satisfactory result. Also, proposed ideal architecture is not computationally expensive and simulations are thus easy to run.

3.4. Classification problem

For the classification problem we use the Wisconsin Breast Cancer dataset provided by Scikit-learn (?). In short, this dataset consists of 569 samples, each with $p = 30$ features, divided into 2 classes; those that were benign, and those that were malignant. This is thus a *binary* classification problem. The major differences to our network model will be the number of input neurons, which will now be $N_0 = 30$, and a single output neuron with a sigmoid output function, i.e. $N_L = 1$, $g_L = \sigma$. We will also use cross entropy in eq. (15) as our loss function.

For comparative reasons, we perform the analysis in a similar manner as with linear regression, and we start with the same model architecture; a network with $L - 1 = 3$ hidden layers of structure $N_1, N_2, N_3 = 15, 10, 5$. We minimise the loss function in the same way, using SGD with RMSProp as our chosen optimiser algorithm, starting with $m = 3$ minibatches. From Figure B.1 with find the optimal hyperparameters to be $\eta = 10^{-3}$ and $\lambda = 10^{-6}$. From Figure B.2 we find the the ideal architecture, given said hyperparameters, is a network with $L - 1 = 2$ hidden layers, each with $N_1 = N_2 = 10$ neurons. We also note that in this case there is no significant favourisation of the simpler networks, as it was for linear regression. However, we choose the simplest one, with the highest accuracy. Figure B.3 and Figure B.4 show the accuracy as function of training epochs for the four different activation functions, given the above architecture. We see that the accuracy when using ReLU is close to 1, when we increase the training epochs, and we deem this to be the optimal activation function. With the above conditions, the optimal model should be trained for 900 epochs using $m = 5$ according to Figure B.5, which gives an accuracy of 1[8].

The difficulties of finding an optimal model for a classification problem are similar to those of classification. The above architecture and tuning is only one of many ways of designing a model which would yield good results. It is challenging to be adamant that one model outperforms the others, since we have only tested for one, and also because the way we decide on the architecture and parameters could very well be switched around. Even for the model at hand, the heatmaps does not give a definite answer as to which set of hyperparameters or architecture is actually best, only how good they are. It is therefore wise to choose the simplest solution (most computationally efficient) that yields the most accurate result. However, the model at hand gives a descent result, and seem to work well.

3.5. Logistic regression

When performing the logistic regression analysis we notice that this is equivalent of having a neural network with no hidden layers and sigmoid as activation function. We use SGD with RMSProp as optimiser, so the only parameters we need to determine are the learning rate and regularisation. We use 250 epochs and $m = 3$ minibatches. From Figure C.1 we have that the optimal hyperparameters are $\eta = 10^{-3}$ and $\lambda = 10^{-8}$. The obtained accuracy is 1[9], which is satisfactory. We are thus pleased with this model and can deduce that it performs as well as the tuned neural network.

3.6. Closing words

We have so far given estimates of how good our models are in terms of generalisation MSE and accuracy based on an untouched part of the dataset. There are more bulletproof ways of giving these measurements, for instance the bootstrap or cross-validation methods we discussed in project 1. Especially alarming are the accuracy scores of 1, which in reality is not realistic. An FFNN such as ours should not give such perfect results, however, we know that the classification dataset we use is ideal for such analyses. That is to say, there is most likely a single feature (or maybe a several few) that discloses whether a person has cancer or not. That would be plain to see if we were to make a histogram the frequency of said feature(s) where we separate the malignant cases from the benign cases and see if they overlap at all. If they do not or only a little, then this would be a very important feature. In any case, we might have benefitted especially in this case from using some resampling methods.

In this analysis, we have favourised RMSProp as optimiser. This was due to the nice results in section 3.1 and because we aimed to reduce the amount of hyperparameters to tune. However, we cannot say for certain that this is the best optimiser for our loss functions.

4. Conclusion

For linear regression on the Franke function, the optimal model is an NN with $L-1 = 1$ hidden layer with the number of neurons $N_1 = 30$. The hyperparameters are tuned to be $\eta = 10^{-1}$, $\lambda = 10^{-4}$. We use the sigmoid activation function and train the model over 700 epochs using SGD with RMSProp optimiser with $\varrho = 0.9$ using $m = 2$ minibatches. The resulting performance measure MSE $= 0.052$. This is significantly better than 0.15, which was obtained with OLS on the same function, with comparable noise in project 1. This is only one of many models that could deliver an equally good result. It is fairly computational efficient, and accurately fits a function to the data.

For the classification task, we found the optimal model to be an NN with $L - 1 = 2$ hidden layers for which $N_1 = N_2 = 10$. The hyperparameters are tuned to be $\eta = 10^{-3}$ and $\lambda = 10^{-6}$. We use the ReLU activation function and train the model over 900 epochs using SGD with RMSProp optimiser ($\varrho = 0.9$) having $m = 5$ minibatches. The result is a model with Accuracy $= 1$, which is the best possible accuracy.

For logistic regression we use an NN with no hidden layers ($L-1 = 0$), and sigmoid activation function ($g_L = \sigma$). This as well manages to perform with Accuracy $= 1$, having $\eta = 10^{-3}$ and $\lambda = 10^{-8}$. Since this is a simpler model than the NN of depth $L = 3$, we may conclude that logistic regression does an equally good job in this case.

A continuation of this work would be to troubleshoot the code for implementation errors and perform more simulation of different network type, initialise the weights and

[8] Given to two significant digits, which would allow for a few false predictions given the size of our dataset.

[9] Again, to 2 significant digits, which may contain a few false predictions.

biases differently and tune the architecture and hyperparameters in dif ferent ways and orders. Another natural extension would be to make the neural network able to perform multivariate classification.

Code availability

The code is available on GitHub at `https://github.com/Johanmkr/FYS-STK4155colab/tree/main/project2`.

Appendix A: Regression figures

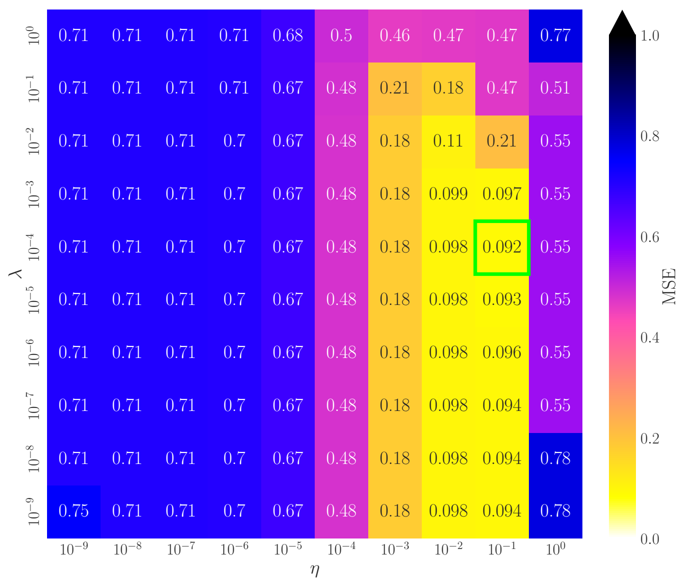


Fig. A.1. Heatmap of the MSE as function of learning rate η and regularisation parameter λ, using SGD with RMSProp as optimiser performing regression analysis of a 3 layered, 15-10-5 neurons, neural network.

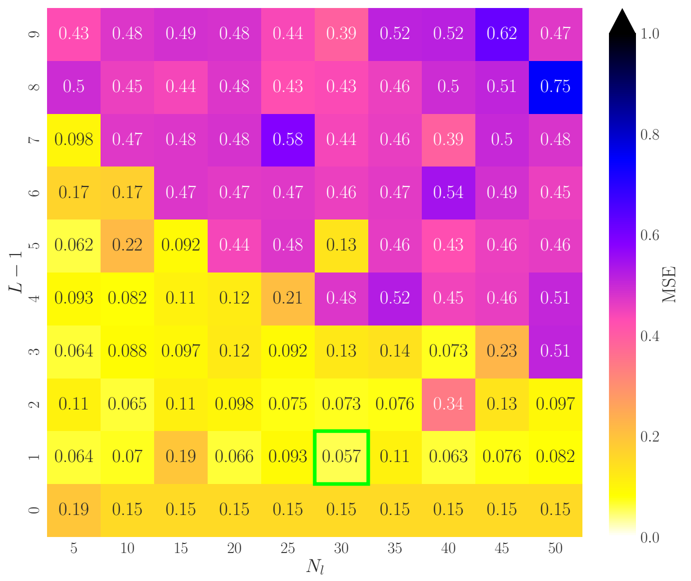


Fig. A.2. Heatmap of the MSE as function of hidden layers $L-1$ and neurons per layer N_l, using SGD with RMSProp as optimiser performing regression analysis with $\eta = 10^{-1}$ and $\lambda = 10^{-4}$.

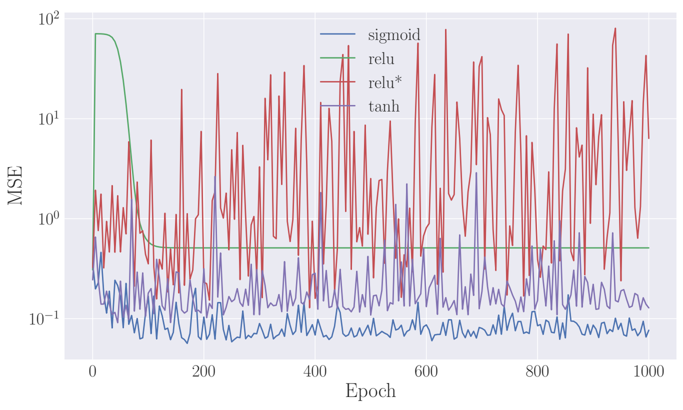


Fig. A.3. Plot of the MSE for up to 1000 epochs, using SGD with RMSProp as optimiser performing regression analysis with $L-1=1$ hidden layer with $N_l = 30$ neurons with $\eta = 10^{-1}$ and $\lambda = 10^{-4}$. The four different activation functions perform differently. Note the logarithmic MSE axis.

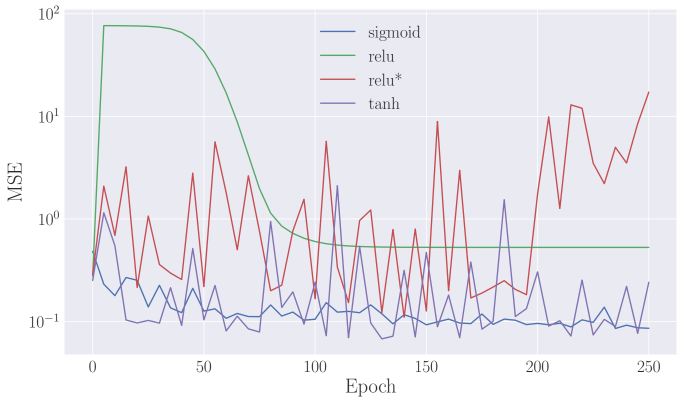


Fig. A.4. Plot of the MSE for up to 250 epochs, using SGD with RMSProp as optimiser performing regression analysis with $L-1=1$ hidden layer with $N_l = 30$ neurons with $\eta = 10^{-1}$ and $\lambda = 10^{-4}$. The four different activation functions perform differently. Note the logarithmic MSE axis.

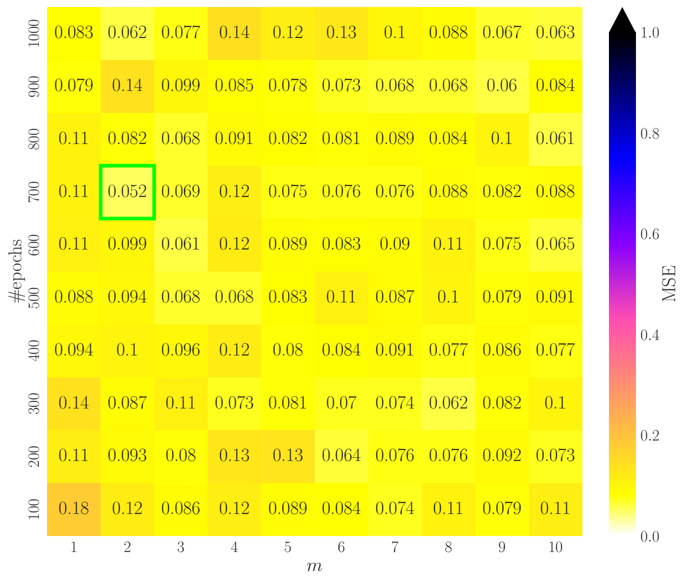
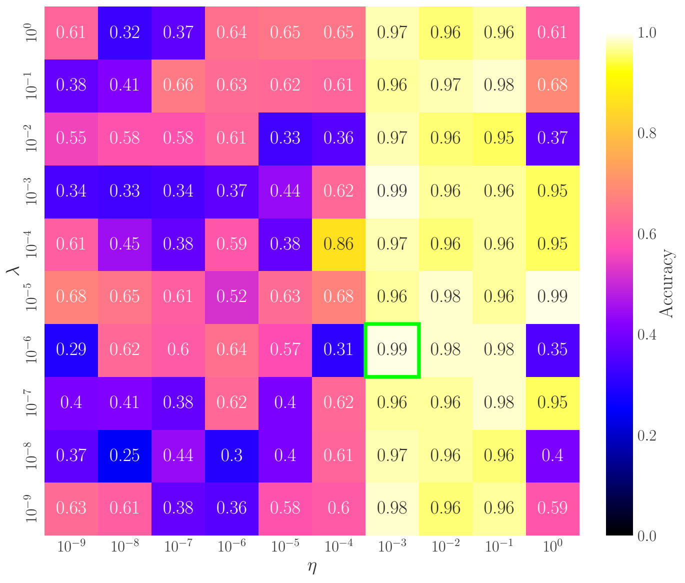


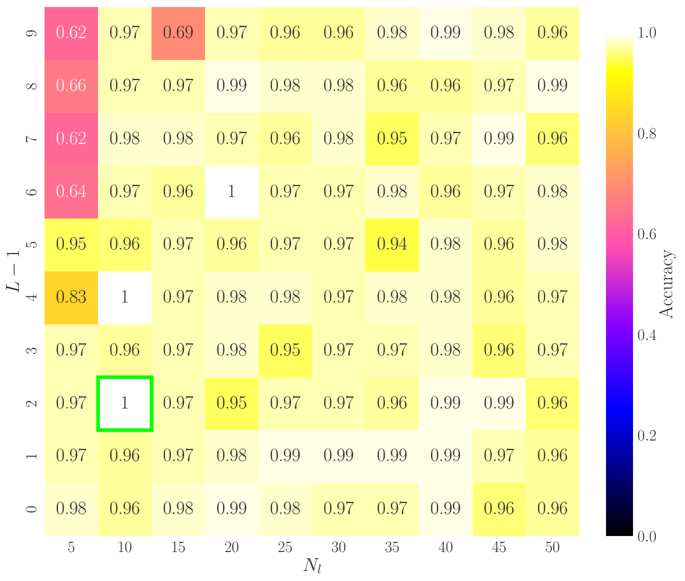
Fig. A.5. Heatmap of the MSE as function of the number of minibatches m and training epochs, using SGD with RMSProp as optimiser performing regression analysis with $L-1=1$ hidden layer with $N_l = 30$ neurons with $\eta = 10^{-1}$ and $\lambda = 10^{-4}$ using sigmoid as activation function.

Appendix B: Classification figures



λ \ η	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}	10^{0}
10^{0}	0.61	0.32	0.37	0.64	0.65	0.65	0.97	0.96	0.96	0.61
10^{-1}	0.38	0.41	0.66	0.63	0.62	0.61	0.96	0.97	0.98	0.68
10^{-2}	0.55	0.58	0.58	0.61	0.33	0.36	0.97	0.96	0.95	0.37
10^{-3}	0.34	0.33	0.34	0.37	0.44	0.62	0.99	0.96	0.96	0.95
10^{-4}	0.61	0.45	0.38	0.59	0.38	0.86	0.97	0.96	0.96	0.95
10^{-5}	0.68	0.65	0.61	0.52	0.63	0.68	0.96	0.98	0.96	0.99
10^{-6}	0.29	0.62	0.6	0.64	0.57	0.31	0.99	0.98	0.98	0.35
10^{-7}	0.4	0.41	0.38	0.62	0.4	0.62	0.96	0.96	0.98	0.95
10^{-8}	0.37	0.25	0.44	0.3	0.4	0.61	0.97	0.96	0.96	0.4
10^{-9}	0.63	0.61	0.38	0.36	0.58	0.6	0.98	0.96	0.96	0.59

Fig. B.1. Heatmap of accuracy as function of learning rate η and regularisation parameter λ, using SGD with RMSProp as optimiser performing regression analysis of a 3 layered, 15-10-5 neurons, neural network.



$L-1$ \ N_l	5	10	15	20	25	30	35	40	45	50
9	0.62	0.97	0.69	0.97	0.96	0.96	0.98	0.99	0.98	0.96
8	0.66	0.97	0.97	0.99	0.98	0.98	0.96	0.96	0.97	0.99
7	0.62	0.98	0.98	0.97	0.96	0.98	0.95	0.97	0.99	0.96
6	0.64	0.97	0.96	1	0.97	0.97	0.98	0.96	0.97	0.98
5	0.95	0.96	0.97	0.96	0.97	0.97	0.94	0.98	0.96	0.98
4	0.83	1	0.97	0.98	0.98	0.97	0.98	0.98	0.96	0.97
3	0.97	0.96	0.97	0.98	0.95	0.97	0.97	0.98	0.96	0.97
2	0.97	1	0.97	0.95	0.97	0.97	0.96	0.99	0.99	0.96
1	0.97	0.96	0.97	0.98	0.99	0.99	0.99	0.99	0.97	0.96
0	0.98	0.96	0.98	0.99	0.98	0.97	0.97	0.99	0.96	0.96

Fig. B.2. Heatmap of accuracy as function of hidden layers $L-1$ and neurons per layer N_l, using SGD with RMSProp as optimiser performing regression analysis with $\eta = 10^{-3}$ and $\lambda = 10^{-6}$.

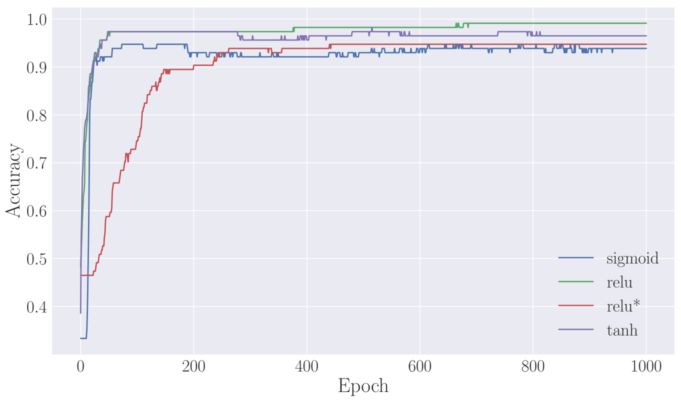


Fig. B.3. Plot of accuracy for up to 1000 epochs, using SGD with RMSProp as optimiser performing regression analysis with $L-1 = 2$ hidden layers with $N_l = 10$ neurons each with $\eta = 10^{-3}$ and $\lambda = 10^{-6}$. The four different activation functions perform differently.

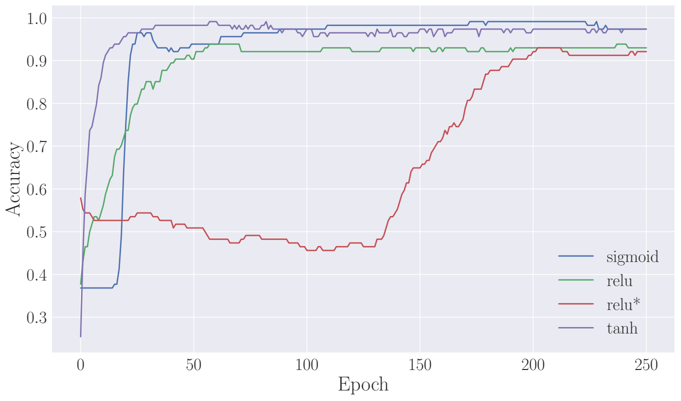


Fig. B.4. Plot of accuracy for up to 250 epochs, using SGD with RMSProp as optimiser performing regression analysis with $L-1 = 2$ hidden layer with $N_l = 10$ neurons with $\eta = 10^{-3}$ and $\lambda = 10^{-6}$. The four different activation functions perform differently.

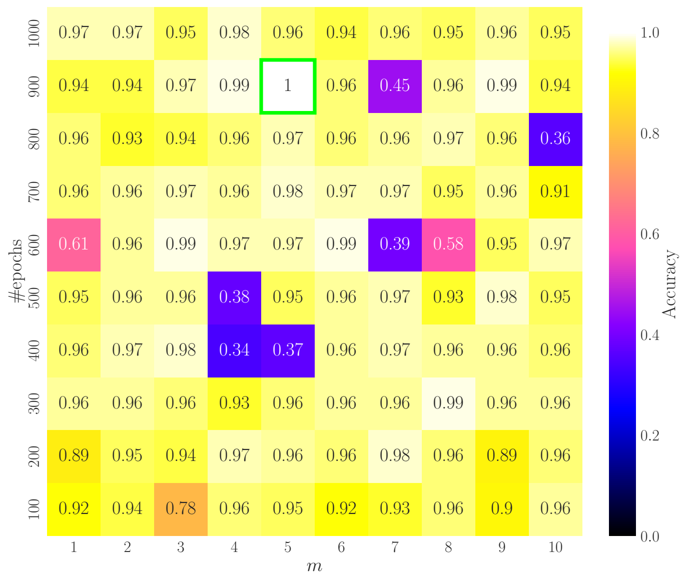


Fig. B.5. Heatmap of accuracy as function of the number of minibatches m and training epochs, using SGD with RMSProp as optimiser performing regression analysis with $L-1=2$ hidden layer with $N_l = 10$ neurons with $\eta = 10^{-3}$ and $\lambda = 10^{-6}$ using RELU as activation function.

Appendix C: Logistic regression figure

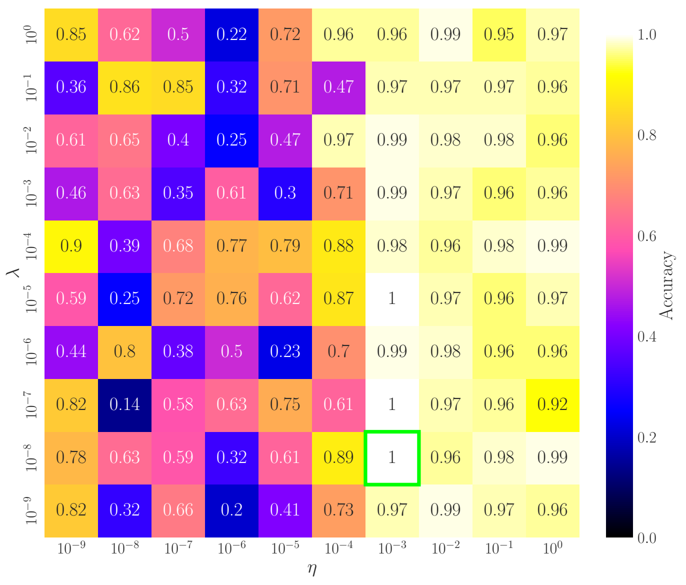


Fig. C.1. Heatmap of the MSE as function of learning rate η and regularisation parameter λ, using SGD with RMSProp as optimiser performing regression analysis of a network with no hidden layers using the sigmoid function as activation function. This is equivalent of performing logistic regression.

Appendix D: Optimiser algorithms

FIX CITATIONS

Adagrad[10]

Algorithm 1 Adagrad algorithm

```
Require: Global learning rate $\eta$
Require: Initial parameter $\boldsymbol{\theta}$
Ensure: Initialise gradient accumulation variable $\mathbf{r} = \mathbf{0}$
  while $|\mathbf{v}| > 10^{-10}$ do
      Accumulate squared gradient: $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g}$
      Compute update
  end while
```

Appendix E: Back propagation algorithm

The output $\hat{\mathbf{y}}$ of our NN is given by the layer value of the output layer $\mathbf{h}^L$ passed through an output function g_L: $\hat{\mathbf{y}} = g_L(\mathbf{h}^L)$. The output error is given by

$$\boldsymbol{\delta}^L = \frac{\mathrm{d}g_L(\mathbf{a}^L)}{\mathrm{d}\mathbf{a}^L} \odot \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} = \frac{\mathrm{d}g_L(\mathbf{a}^L)}{\mathrm{d}\mathbf{a}^L} \odot \frac{\partial \mathcal{L}}{\partial \mathbf{h}^L} \left(\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{h}^L} \right)^{-1}, \tag{E.1}$$

where $\odot$ is the element wise Hadamard product. This error is propagated through the layers of the network in backward order,

$$\boldsymbol{\delta}^l = \boldsymbol{\delta}^{l+1} W^{l+1 \to l} \odot \frac{\mathrm{d}}{\mathrm{d}\mathbf{a}^l} g_l(\mathbf{a}^l), \tag{E.2}$$

for $l = L-1, L-2, \ldots, 1$. Having found the error propagated through each layer, we update the weights and biases as follows:

$$\begin{aligned} \nabla_W^l &= (\boldsymbol{\delta}^l)^\intercal \mathbf{h}^{l-1} \\ \nabla_b^l &= \boldsymbol{\delta}^l \\ W^l &= \mathcal{U}(W^l, \nabla_W^l) \\ \mathbf{b}^l &= \mathcal{U}(\mathbf{b}^l, \nabla_b^l) \\ \implies \boldsymbol{\theta}^l &= \mathcal{U}(\boldsymbol{\theta}^l, \nabla_\theta^l)\,; \qquad \boldsymbol{\theta}^l = (W^l, \mathbf{b}^l), \end{aligned} \tag{E.3}$$

Here $\mathcal{U}(\boldsymbol{\theta}, \nabla_\theta)$ is a function that updates the parameter $\boldsymbol{\theta}$ according som some optimisation scheme, typically SGD with a favoured optimised. For reference, plain gradient descent reads: $\mathcal{U}(\boldsymbol{\theta}, \nabla_\theta) = \boldsymbol{\theta} - \eta\nabla_\theta$, where η is the learning rate.

[10] Algorithm 8.4 in Goodfellow et al.