From Classification and Regression to Neural Networks

J. G. Inácio,[1] J. A. Fløisand,[1] and J. E. Kings[1]

[1]*Department of Physics, University of Oslo, Norway, Sem Sælands vei 24, 0371 Oslo*

(Dated: November 15, 2021)

A code for a feedforward neural network is constructed with a flexible number of layers, nodes per layer and a vast array of activation functions to choose from. The training process is done by optimization methods, using the backpropagation algorithm for fast computation of gradients. We also solve classification and regression problems with optimization methods, compare their performance and discuss the update schedule for the learning rate. Lastly we apply the developed neural network to study the Wisconsin Breast Cancer data set and make real-world predictions as to whether tumours can be considered benign or malignant. We find that our implementation outperforms standard regression methods solved with mini-batch stochastic gradient descent, for both linear and logistic models. In the case of classification, we find that where the neural network reaches an accuracy of up to 99%, logistic regression gives an accuracy of only 94%. For linear regression problems where the neural network gives a mean squared error of 0.027, we get 0.075 with ridge regression solved with mini-batch stochastic gradient descent.

INTRODUCTION

The idea of neural networks dates as far back as the 1940's. Inspired by the way neurons connect and interact in the brains of living organisms, McCulloch *et al*. [1] proposed a model where all neurons are connected to each other, and through repeated use, strengthen their connection if it proves favourable. Later in 1958, psychologist Frank Rosenblatt introduced the idea of a perceptron [2], the Mark I Perceptron. This model is the foundation of most neural networks used today. Weighted inputs are evaluated by a special function to determine the state of activation of the neuron. This however posed a major drawback: with a single perceptron, only linear problems can be solved. In the following decade, the work of Bernard Widrow and Marcian Hoff [3], led to the creation of networks of perceptrons that are capable of solving non-linear problems. Their application of neural networks to a real-world problem (reducing noise over phone lines), is still in use today. Nowadays, neural networks are much more robust and able to solve a vast array of complex problems, from weather prediction to self-driving cars.

In this report we are not going to program a car to self-drive, but we are hoping to build a neural network that can correctly classify the state of a potential cancer patient given some features about the tumour. Our neural network will also be able to read handwritten numbers from one to ten.

We start by solving simple regression and classification problems with gradient descent (GD), mini-batch stochastic gradient descent (SGD) and RMSprop gradient descent based on momentum GD. We solve the ordinary least squares, ridge and logistic regression cases. We also compare the efficiency of the different optimization methods and the update schedule for the learning rate. This part is done without any involvement of a neural network.

We then describe our implementation of a feedforward neural network (FFNN), with a flexible number of layers, nodes per layer, activation functions, and so forth. The training process is achieved by the backpropagation algorithm, for the lightweight calculation of otherwise computationally expensive gradients, in conjunction with GD or SGD.

Lastly we apply this neural network code to a famous data set amongst the machine learning community, the Wisconsin Breast Cancer data set `https://www.kaggle.com/uciml/breast-cancer-wisconsin-data`. This data set consists of features, such as area, radius, texture, etc, taken from the medical imaging of tumours from about 500 patients that are either classified as malignant or benign. Our objective is to use the developed code for the neural network to predict whether or not a given individual has a malignant or benign tumour given the features of the tumour.

All of the codes used to get the re-

sults are available on the following GitHub page `https://github.com/jgci2000/FYS-STK4155-projects`, under the `project_2` folder.

METHODS

One of the benefits of neural networks is that they can be set up to "solve"[1] both classification and continuous problems. We will test our implementation against logistic regression for a classification problem, and against linear regression for a continuous problem. In this section, we will describe some of the important points that are used in our implementation. As we have already gone through regression in our previous report, we will mostly focus on logistic regression and FFNNs here.

Gradient Methods

When optimizing a function $f : \mathbb{R}^n \to \mathbb{R}$, we can find hard or impossible expressions to solve analytically. We are then in need of numerical method that give good approximations. If the function we are optimizing is differentiable, we can try to find where the gradient is zero. We will here mention two such methods, both of which can be thought of as fixed point iterations. We note that any maximization problem can be turned into a minimization problem. In our case this is done by looking at $g(\boldsymbol{x}) = \nabla f(\boldsymbol{x})$ instead.

For a function $g : \mathbb{R}^n \to \mathbb{R}^n$, *Newton's method* can be written as

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \big(J_g(\boldsymbol{x}^{(k)})\big)^{-1} g(\boldsymbol{x}^{(k)}),$$

where $J_g(\boldsymbol{x})$ denotes the Jacobian of g at $\boldsymbol{x}$. Two ways of deriving this are by setting a first order Taylor approximation of g equal to zero [4], or by finding the matrix Λ such that the fixed point iteration $\boldsymbol{x}^k - \Lambda g(\boldsymbol{x}^k) = \boldsymbol{x}^{(k+1)}$ is a contraction [5]. As our goal is to find a zero of ∇f, the function g above will now be the gradient ∇f, and the Jacobian of ∇f is the Hessian matrix $H_f(\boldsymbol{x})$. Thus, Newton's method takes the form

$$\boldsymbol{x}^{(k)} = \boldsymbol{x}^{(k)} - \big(H_f(\boldsymbol{x}^{(k)})\big)^{-1} \nabla f(\boldsymbol{x}^{(k)}).$$

In many cases, data sets and the number of parameters can be so large that computing the Hessian matrix and inverting it demands to much time. To handle this problem, we can use *simultaneous relaxation* instead [5]. simultaneous relaxation is defined by the recursion

$$\boldsymbol{x}^{(k)} = \boldsymbol{x}^{(k)} - \eta g(\boldsymbol{x}^{(k)}).$$

We can think of this as setting the Jacobian of g equal to a single number. For finding zeros of our gradient ∇f, this method is given by

$$\boldsymbol{x}^{(k)} = \boldsymbol{x}^{(k)} - \eta \nabla f(\boldsymbol{x}^{(k)}). \tag{1}$$

This method is known as *gradient descent* and the parameter η is called the *learning rate*. For (1) to be a contraction, we would need η to be less than the largest eigenvalue of $H_f(\boldsymbol{x})^{-1}$ for all $\boldsymbol{x}$.

It is not uncommon to work with so large data sets that even gradient descent is considered to slow. Moreover, both Newton's method and gradient descent will find local minimum, which need not be global. To solve this we introduce the method called *Stochastic Gradient Descent*. When doing stochastic gradient descent, we randomly choose a data point and do gradient descent for this data point alone. This way, we are able to have much faster computations, but we will not choose an optimal direction for the updated point in our iteration. Although this sounds bad, it will also allow the iterated points to be able to move away from local minimum.

The idea of stochastic gradient descent opens up for a similar method which we will call *mini-batch Stochastic Gradient Descent* (abbreviated by SGD). In this method, we choose M data points at random and do gradient descent for these data points. M is called the batch size of the method. This method benefits from being less computationally heavy, while also allowing us the benefits of gradient descent. We note that if we have N data points, then $\lfloor N/M \rfloor$ iterations is called an epoch.

[1] Solve is not a precise enough word to use here. Neural networks try to find a model to correctly predict the problem at hand.

An alternative to the constant learning rate η we suggested above would be a learning rate that depends on each iteration (see f.ex. [6]). This is known as *adaptive learning rate*. For decreasing learning rates, one major drawback is that our learning rate might approach zero before the method has had time to converge. We will study the learning rate

$$\eta(t) = \frac{t_0}{t_1 + t},$$

where t_0 and t_1 are parameters we can tune and t is the current epoch. Given the total number of epochs, initial learning rate η_0 and final learning rates η_1, we find the expression for t_0 and t_1 to be given by

$$\begin{cases} t_0 = \frac{\eta_0\eta_1}{\eta_0-\eta_1}\text{epochs} \\ t_1 = \frac{\eta_1}{\eta_0-\eta_1}\text{epochs} \end{cases}$$

This gives us more control over how fast and how much the learning rate should decrease.

Linear Regression

In the last project, we have studied extensively three regression models, thus in this section we will not explain the methods in a thorough fashion.

Linear regression models arise when trying to minimize the Mean Squared Error (MSE) given by

$$\ell(\boldsymbol{\beta}) = \mathbb{E}(\tilde{\boldsymbol{y}} - \boldsymbol{y})^2,$$

where $\tilde{\boldsymbol{y}}_i = \boldsymbol{x}_i\boldsymbol{\beta}$ is the prediction of our model and $\boldsymbol{y}$ is the target value. This problem is known as ordinary least squares. Adding a regularization parameter λ to the cost function,

$$\ell(\boldsymbol{\beta}) = \mathbb{E}(\tilde{\boldsymbol{y}} - \boldsymbol{y})^2 + \lambda\|\boldsymbol{\beta}\|_2^2,$$

is known as Ridge regression and

$$\ell(\boldsymbol{\beta}) = \mathbb{E}(\tilde{\boldsymbol{y}} - \boldsymbol{y})^2 + \lambda|\boldsymbol{\beta}|,$$

is known as Lasso regression.

Logistic Regression

This section will closely follow section 4.4.1 in Hastie *et al*. [7]. For a more thorough explanation we suggest reading the referenced book.

Assume we are given data $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N)$ which belongs to classes $K = 1, \ldots, k$. Each data point $\boldsymbol{x}_i$ is given as a vector, as it might include a number of different features depending on the specific problem. For instance, $\boldsymbol{x}_i$ could be values for the different features of cancer patient i, such as size of tumour, area, and more. As the classes are discrete, we do not expect the regression models we have studied before to give good results. Instead, we look at our data from a more probabilistic point of view, taking the probability of being in class K, given the data point x_i, $P(\text{class} = K|X = x_i)$. For the logistic regression model, we require these probabilities to be linear after a monotone transformation. All probabilities should sum up to one, meaning they must be individually contained in $[0, 1]$. For two classes $K = 0, 1$, the model is given by requiring

$$\ln\frac{P(\text{class} = 0|X = \boldsymbol{x})}{P(\text{class} = 1|X = \boldsymbol{x})} = \beta_{10} + \sum_{i=1}^{n}\beta_{1i}x_i = \boldsymbol{\beta}^T\boldsymbol{x}$$

and

$$P(\text{class} = 0|X = \boldsymbol{x}) + P(\text{class} = 1|X = \boldsymbol{x}) = 1,$$

where we for simplicity assume that $x_0 = 1$ for all data points $\boldsymbol{x}$. Writing $P(\text{class} = 0|X = \boldsymbol{x}) = p_0(\boldsymbol{x};\boldsymbol{\beta}_0)$ and $P(\text{class} = 1|X = \boldsymbol{x}) = p_1(\boldsymbol{x};\boldsymbol{\beta}_1)$, this gives

$$p_0(\boldsymbol{x};\boldsymbol{\beta}_0) = \frac{e^{\boldsymbol{\beta}_0^T\boldsymbol{x}}}{1 + \sum_{l=0}^{1} e^{\boldsymbol{\beta}_l^T\boldsymbol{x}}}$$
$$p_1(\boldsymbol{x};\boldsymbol{\beta}_1) = \frac{1}{1 + \sum_{l=0}^{1} e^{\boldsymbol{\beta}_l^T\boldsymbol{x}}},$$

and thus

$$p_1(\boldsymbol{x};\boldsymbol{\beta}_1) = 1 - p_0(\boldsymbol{x};\boldsymbol{\beta}_0).$$

Therefore, a single vector parameter $\boldsymbol{\beta}$ is needed. The general case is similar.

Continuing with the $K = 0, 1$ case, to find the optimal $\boldsymbol{\beta}$ we want to maximize the log-likelihood,

also called cross-entropy:

$$\ell(\boldsymbol{\beta}) = \sum_{i=1}^{N} \Big(y_i \ln p_0(\boldsymbol{x_i};\boldsymbol{\beta}) + (1-y_i)\ln(1-p_0(\boldsymbol{x_i};\boldsymbol{\beta})) \Big)$$
$$= \sum_{i=1}^{N} \Big(y_i \boldsymbol{\beta}^T \boldsymbol{x}_i - \ln(1+e^{\boldsymbol{\beta}^T \boldsymbol{x_i}}) \Big),$$

where $y_i \in \{0,1\}$ is the classification of the data point $\boldsymbol{x}_i$. To find the maximum, we look for the point where the derivative of $\ell(\boldsymbol{\beta})$ is zero:

$$\frac{d\ell(\boldsymbol{\beta})}{d\boldsymbol{\beta}} = \sum_{i=1}^{N} \boldsymbol{x_i}(y_i - p_0(\boldsymbol{x_i};\boldsymbol{\beta})) = 0.$$

To solve this equation, we can apply Newton's method, which requires the second derivative,

$$\frac{d^2\ell(\boldsymbol{\beta})}{d\boldsymbol{\beta} d\boldsymbol{\beta}^T} = -\sum_{i=1}^{N} \boldsymbol{x_i}\boldsymbol{x_i}^T p_0(\boldsymbol{x_i};\boldsymbol{\beta})(1-p_0(\boldsymbol{x_i};\boldsymbol{\beta})).$$

For Newton's method, we compute

$$\boldsymbol{\beta}^{(n+1)} = \boldsymbol{\beta}^{(n)} + \left(\frac{d^2\ell(\boldsymbol{\beta})}{d\boldsymbol{\beta} d\boldsymbol{\beta}^T} \right)^{-1} \frac{d\ell(\boldsymbol{\beta})}{d\boldsymbol{\beta}},$$

where $\boldsymbol{\beta}^{(0)}$ is an initial guess. As discussed in the section on gradient methods, it can be beneficial to do gradient descent instead. For gradient descent, we compute

$$\boldsymbol{\beta}^{(n+1)} = \boldsymbol{\beta}^{(n)} + \eta \frac{d\ell(\boldsymbol{\beta})}{d\boldsymbol{\beta}},$$

where $\boldsymbol{\beta}^{(0)}$ is an initial guess and η is the learning rate.

As for ordinary linear regression, we can hope to improve our results by adding a regularization term. For our implementation, we will include an ℓ_2 regularization term λ. Following Hastie *et al.* [7, section 4.4.4 and 11.5.2], this means that we instead maximize

$$\ell(\boldsymbol{\beta}) = \sum_{i=1}^{N} \Big(y_i \boldsymbol{\beta}^T \boldsymbol{x}_i - \ln(1+e^{\boldsymbol{\beta}^T \boldsymbol{x_i}}) \Big) - \lambda \left\| \boldsymbol{\beta} \right\|_2^2,$$

which gives

$$\frac{d\ell(\boldsymbol{\beta})}{d\boldsymbol{\beta}} = \sum_{i=1}^{N} \boldsymbol{x_i}(y_i - p_0(\boldsymbol{x_i};\boldsymbol{\beta})) - 2\lambda\boldsymbol{\beta}$$

and

$$\frac{d^2\ell(\boldsymbol{\beta})}{d\boldsymbol{\beta} d\boldsymbol{\beta}^T} = -\sum_{i=1}^{N} \boldsymbol{x_i}\boldsymbol{x_i}^T p_0(\boldsymbol{x_i};\boldsymbol{\beta})(1-p_0(\boldsymbol{x_i};\boldsymbol{\beta})) - 2\lambda I_n$$

where I_n denotes the $n \times n$ identity matrix.

Feedforward Neural Network

We will here discuss enough of the theory around neural networks to describe our implementation of a feedforward neural network. There is a vast supply of books and articles surrounding this topic. References we recommend include Nielsen [8], Hastie *et al.* [7], Bishop [9] and Goodfellow *et al.* [10].

The simplest possible neural network is the single perceptron model where we have one node with two inputs and one output, Figure 1. We consider that

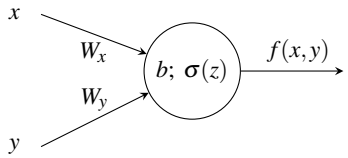

Figure 1: Single perceptron model

each input has an associated weight W_x and W_y, and that the node has an intrinsic bias b and an *activation function* σ. Given an input $z = W_x x + W_y y + b$, the activation function determines the output of the node $a = \sigma(z)$. This simple example can be thought of as a regression problem for the three parameters x, y and the intercept b.

In the last project, we studied linear models that can perform a polynomial fit to a well-behaved function. We can then build upon the simple perceptron model. If we now let the one node have n inputs, we can solve a linear regression problem of degree n, if we think of each input i as being x^i, and each weight as being a coefficient in the linear expression. With this setup, we can also solve binary classification problems, by scaling the output and interpreting it as a probability.

Building further upon this model, by adding an additional layer between the input and output layer, we can create a so-called multi-layer perceptron model, Figure 2. This we can solve even harder

problems, e.g. image recognition and K-class classification problems. This model is known as a feed-

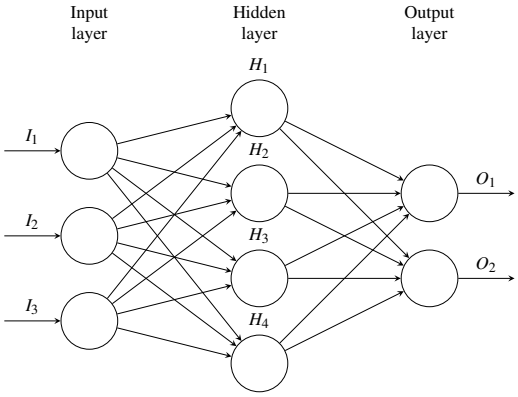

Figure 2: Multi-layer perceptron model with one hidden layer.

forward neural network. Such a network is assigned L layers (layers 1 through $L-1$ are the hidden layers, and layer L is the output layer), with n_l neurons (or nodes) in each layer. For each of the layers a weight matrix $W^{(l)}$ of size $n_l \times n_{l-1}$ is assigned, where the entry $W_{ij}^{(l)}$ represents the weight that connects node i from layer $l-1$ to node j into layer l. Each layer is also assigned a bias vector $b^{(l)}$ of size n_l. To evaluate the networks predictions, we apply a cost function. This will determine the accuracy or error of our model. We wish to minimize this error.

We can use such a model to make predictions using the feed forward process. The inputs can be denoted as $a^{(0)}$; then, for each layer, in turn, from 1 to L, the layer's output $a^{(l)}$ (vector of size n_l) is computed as

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)},$$
$$a^{(l)} = \sigma^{(l)}(z^{(l)}),$$

where $\sigma^{(l)}$ is the layer's activation function. For a regression problem, the output layer's activation function should be the identity function, $\sigma^{(L)} : z_i \mapsto z_i$, as we wish to allow all values in $\mathbb{R}$. On the other hand, for classification problems, as we desire a probabilistic output, the prediction should be contained in $[0,1]$. This way, we can use a sigmoid function,

$$\sigma^{(L)}(z_i) = \frac{1}{1+e^{-z_i}},$$

or any other function of the type $\mathbb{R} \to [0,1]$. A usual example is the softmax

$$\sigma^{(L)}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n_l} e^{z_j}}$$

or the hyperbolic tangent

$$\sigma^{(L)}(z_i) = \tanh(z_i).$$

Additionally, we explore using different activation functions in the hidden layers, amongst the sigmoid, the hyperbolic tangent, the rectified linear unit (ReLU) function

$$\sigma^{(l)}(z_i) = \begin{cases} z_i & z_i \geq 0 \\ 0 & z_i < 0, \end{cases}$$

the leaky ReLu

$$\sigma^{(l)}(z_i) = \begin{cases} z_i & z_i \geq 0 \\ \alpha z_i & z_i < 0, \end{cases}$$

and the exponential linear unit (ELU) function

$$\sigma^{(l)}(z_i) = \begin{cases} z_i & z_i \geq 0 \\ \alpha\,(e^{z_i} - 1) & z_i < 0, \end{cases}$$

with $0 < \alpha \ll 1$.

Typically, the weights and biases in the network are initialized randomly, corresponding to an initial guess. It is then highly unlikely to produce good predictions. In order to obtain accurate prediction with our network, the weights and biases are iteratively adjusted until the predictions given by the network are satisfactory. This process is called *training*. In this process we use gradient methods to find the minimum of the model's cost function. We are now faced with a problem of computing the gradients of the cost function with respect to all of the weights and biases that comprise the network. As the network has potentially hundreds of weights and biases all dependant on each other, this becomes

quite computationally heavy to do naively. Luckily, there is an algorithm that can efficiently compute the task at hand: *backpropagation*. A short overview is presented here; for an in-depth derivation of the expressions shown, see Appendix B.

Depending on the way we initialize the weights and biases, we might get a faster convergence or non-convergence of the employed gradient method. We initialize the biases to some small value ε (the value of which we can experiment with to find the best initial value), and the weights with uniform distribution between -1 and 1.

The way we update the weights and biases each epoch is given by

$$W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial C}{\partial W^{(l)}}$$
$$b^{(l)} \leftarrow b^{(l)} - \eta \frac{\partial C}{\partial b^{(l)}}.$$

To compute the gradients of the cost function with respect to the weights and biases, we used the backpropagation algorithm, which consists of computing

$$\frac{\partial C}{\partial z^{(L)}} = \frac{\partial C}{\partial a^{(L)}} \odot \sigma^{L\prime}(z^{(L)}),$$
$$\frac{\partial C}{\partial z^{(l)}} = \sigma^{(l)\,\prime}(z^{(l)}) \odot ((W^{(l+1)})^T \delta^{(l+1)}),$$
$$\frac{\partial C}{\partial b^{(l)}} = \delta^{(l)},$$
$$\frac{\partial C}{\partial W^{(l)}} = \delta^{(l)} (a^{(l-1)})^T,$$

where $\delta^{(l)} := \frac{\partial C}{\partial z^{(l)}}$.

Once all weights and biases throughout the network have been updated epoch times, the hope is that our cost function decreases.

Additionally, to prevent overfitting, we can introduce a regularization term in the cost function [7]. For an ℓ_2 regularization term the updated gradients will take the form

$$W^{(l)} \leftarrow W^{(l)} - \eta \left(\frac{\partial C}{\partial W^{(l)}} + \lambda W^{(l)} \right)$$
$$b^{(l)} \leftarrow b^{(l)} - \eta \left(\frac{\partial C}{\partial b^{(l)}} + \lambda b^{(l)} \right),$$

where λ is a hyperparameter for the regularization term, and C is the cost function without regularization.

IMPLEMENTATION

The core neural network implementation is written in an object-oriented, component-based manner. As reference previously, the code can be found on GitHub. The central `NeuralNetwork.Model.Model` class represents the full network, to be trained and run on a number of different continuous and classification problems.

The base `NeuralNetwork.Layer.Layer` abstract class, inherited by `NeuralNetwork.Layer.HiddenLayer` and `NeuralNetwork.Layer.OutputLayer` for syntax clarity, represents a single layer with n neurons, instances of which can be added as components to a `Model` instance.

When adding new layers to a `Model` instance, in order of forward execution (i.e. with the output layer at the end), the size of each layer is needed (as `Layer::_size`), as well as the size of the previous layer (determined automatically from the model's input size for the first layer, or from the last stacked layer's size), and one of the `NeuralNetwork.ActivationFunctions` sibling classes as the activation function, which provides the function itself and its first derivative.

The available activation functions for hidden layers are the sigmoid, hyperbolic tangent, ELU, ReLU and leaky ReLU. Regarding our softmax function implementation, one thing to note is that the $\sum_{j=1}^{n_l} e^{z_j}$ term is very quick to explode to very large numbers, yielding `NaN`s in the output as a result - instead, we compute softmax as e^S, with

$$\begin{aligned} S &= \ln(e^{z_i}) - \ln\left(\sum_{j=1}^{n_l} e^{z_j}\right) \\ &= z_i - \left(\ln\left(\sum_{j=1}^{n_l} e^{z_j}\right) - \ln\left(e^7\right)\right) - 7 \\ &= z_i - 7 - \ln\left(\sum_{j=1}^{n_l} \frac{e^{z_j}}{e^7}\right) \end{aligned}$$

Since $e^7 \approx 1100$ is a much larger number than individual e^{z_i} terms expected from normalized data points, this expression turns out much more stable computationally.

The feedforward step is done by the `Model::feed_forward` method, to which the inputs are

passed as a matrix of size $k \times s$, where $s =$ `Model::_input_size`, and k any number of data points; the output is given as a $k \times n^{(L)}$ matrix, where $n^{(L)}$ is the size of the output layer. This output is obtained by internally repeatedly calling the `Layer::forward` method on each layer sequentially, which computes the layer's output as

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$
$$a^{(l)} = \sigma(z^{(l)})$$

These results are obtained fully via NumPy matrix operations (including the activation function execution), in order to speed up execution.

Training the network, either via `Model::train`, to train the model once over a number of epochs with a given learning rate and hyperparameters, using either GD or SGD, or via `Model::grid_train` in order to vary any 2 parameters and train the network repeatedly to find the best fit, internally relies on the `Model::back_prop` method, itself calling `Layer::backward` on each layer in reverse order, which computes the weight and bias gradients, updating the individual neurons by

$$W^{(l)} \leftarrow W^{(l)} - \eta (a^{(l-1)} \delta^{(l)} + \lambda W^{(l)})$$
$$b^{(l)} \leftarrow b^{(l)} - \eta (\delta^{(l)\,T} + \lambda b^{(l)})$$

Where $z^{(l)}$ is the non-activated output of layer l as a vector of size n_l, $a^{(l)} = \sigma^{(l)}(z^{(l)})$, η the learning rate and λ the regularization parameter; the column vector error $\delta^{(l)}$ is given by

$$\delta_l = \begin{cases} C'(a^{(L)\,T}) \odot \sigma^{L\,\prime}(z^{(L)\,T}) & l = L \\ (\delta^{(l+1)} W^{(l+1)}) \odot \sigma^{(l)\,\prime}(z^{(l)\,T}) & l \neq L \end{cases}$$

The convenience method `Model::error` feeds forward once with the given inputs and returns $C(a^{(L)})$; i.e. the MSE of the output for regression problems, and the accuracy of the output for classification problems.

DISCUSSION

Regression on Franke's Function

In order to test the implementation of the optimization methods discussed above, we start by running our linear regression models on Franke's function [11]. The data we feed to our models is 1000 uniformly scattered points on $[0,1] \times [0,1]$, sampled as

$$z = F(x,y) + \varepsilon$$

where $F : \mathbb{R}^2 \to \mathbb{R}$ is Franke's function and $\varepsilon \sim N(0, \sigma)$ follows a normally distributed noise, with $\sigma = \frac{1}{4}$ throughout.

Table I: Time, in milliseconds, per epoch for SGD and GD for different learning rate values, averaged over 100 separate runs of 200 epochs each to reduce statistical noise. The size of each batch was set to 5. Computation time taken from system with an i7 4790 @ 3.60 GHz.

Method	SGD (200 batches)		GD	
η	0.01	0.001	0.01	0.001
Time per epoch (ms)	1.69	1.00	0.17	0.13
Time per step (ms)	$8.5 \cdot 10^{-3}$	$5.0 \cdot 10^{-3}$	0.17	0.13

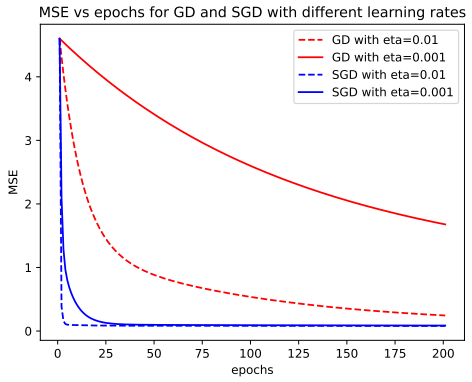

Figure 3: MSE for Franke's function as a function of epochs for Gradient Descent and Stochastic Gradient Descent methods for two different values of the learning rate η. The batch size for SGD is 5.

In Figure 3 we can find the comparison between the error of SGD and GD over epochs for two different learning rates. We see that SGD converges faster for both learning rates than GD. Furthermore,

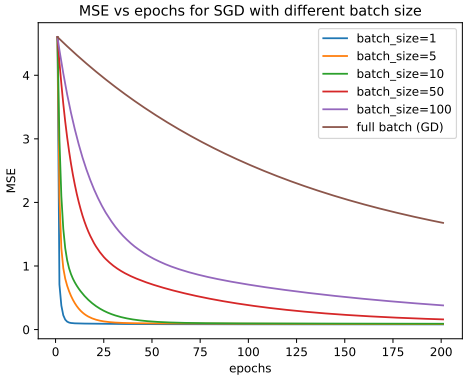

Figure 4: MSE for Franke's function as a function of epochs for SGD optimizer as a function of the batch size. The learning rate is 0.001. As the batch size increases, the MSE over epochs behaves like GD's MSE.

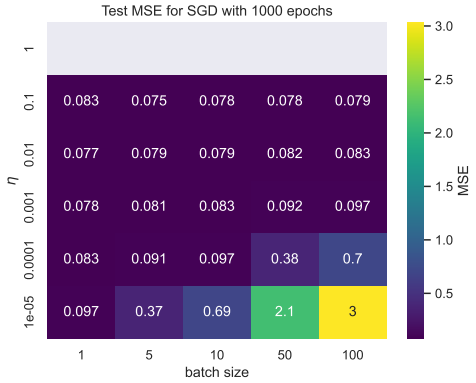

Figure 5: Grid search for the optimal MSE for Franke's function as a function of the batch size and learning rate η. The number of epochs used is 1000. The grey squares indicate that the method was not able to converge due to numerical overflow.

in Figure 4, we can see that as we increase the batch size, the convergence rate of SGD approaches the convergence rate of GD.

Since $\lfloor \frac{N}{M} \rfloor = \frac{1000}{5} = 200$ steps are performed in each epoch of the SGD method while only one is done with GD, it is expected for SGD to take longer per epoch than GD. Indeed, each SGD epoch takes around 9 times longer to compute compared to GD, Table I. That being said, the time per individual step with SGD is much lower than with GD since gradients are computed from 5 points at a time instead of 1000. Thus, SGD and GD achieve very similar MSEs after the same amount of steps, but SGD does so in a much faster fashion in terms of real time taken. Furthermore, due to the random nature of SGD, there is a possibility for encountered local minima to be overcome since there is no confinement to a predetermined path on the n-dimensional MSE surface. Knowing this, we can conclude that SGD is a more efficient and reliable method.

In Figure 5, we have plotted a grid search for the optimal MSE over different learning rates and batch sizes for the SGD method. Through the analysis of the figure, we can see that for learning rates lower that 10^{-4}, the MSE worsens, since the limited number of epochs did not enable the method to converge. Again showing that as the number of batches decreases, the convergence rate also decreases. Thus, for a fast and accurate convergence, we restrict ourselves to a batch size of 5 and learning rates higher than 10^{-4}. For large learning rates, the method is not able to converge due to numerical overflow.

Introducing ℓ_2 regularization and keeping the optimal mini-batch size found above, we can compare results produced by different combinations of parameters η and λ, as shown in Figure 6. In this particular case, ℓ_2 regularization is found to produce worse results than the original $\lambda = 0$ samples, and the best MSE is still found at $(\eta, \lambda) = (0.1, 0)$ after 1000 epochs. This result is to be expected, since we have already shown the polynomial fit to Franke's function to be better modeled by ordinary least squares than ridge regression.

We have also tested how an adaptive learning rate affects our models ability to fit Franke's function. We have found no benefits of using an adaptive scheme for the learning rate. Furthermore finding the optimal values for t_0 and t_1 might pose to be a tedious and unrewarding process, as the convergence rate does not change significantly.

In order to solve the same regression problem with our feedforward neural network implementation, we initialize a network with a single hidden layer with 30 neurons, with a sigmoid as the acti-

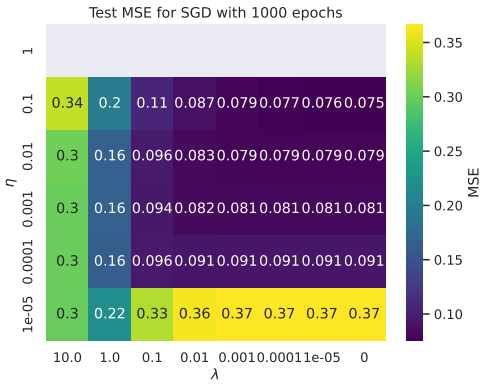

Figure 6: Grid search for the optimal MSE for Franke's function as a function of the learning rate η and ℓ_2 regularization parameter λ. The number of epochs used is 1000. The grey squares indicate that the method was not able to converge due to numerical overflow. This shows that the combination of batch size and learning rate is not optimal for these values - here in particular, $\eta = 1$ is too high of a learning rate.

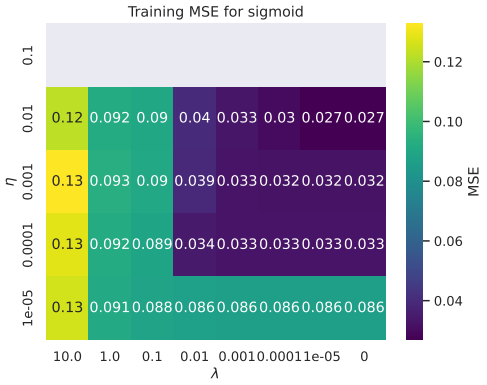

Figure 7: Grid search for the optimal MSE for Franke's function with our feedforward neural network as a function of the learning rate η and ℓ_2 regularization parameter λ. The network is composed of one hidden layer with 30 nodes and a sigmoid as an activation function. The number of epochs used is 1000. The grey squares indicate that the method was not able to converge due to numerical overflow.

vation function. The input layer is of size 2, and the output layer is of size 1, with a linear activation function, in order to fit points $f(x,y) = F(x,y) + \varepsilon$ as $a^{(L)} \stackrel{!}{=} f(a_0^{(0)}, a_1^{(0)})$. The network is trained via SGD, with again a batch size of 5.

Performing the same analysis as we did for OLS and Ridge with SGD, we have produced a grid search over various values of the learning rate and regularization parameter, Figure 7. We have found again, that a low regularization parameter produces the best results to fit Franke's function, with a learning rate higher than 10^{-4}. However, the MSE is about 2 times lower than with OLS and Ridge solved by SGD. We again observe that high learning rates cause numerical overflow.

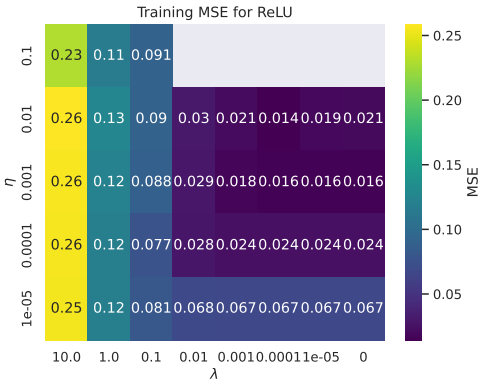

Figure 8: Grid search for the optimal MSE for Franke's function with our feedforward neural network as a function of the learning rate η and ℓ_2 regularization parameter λ. The network is composed of one hidden layer with 30 nodes and a ReLU as an activation function. The number of epochs used is 1000. The grey squares indicate that the method was not able to converge due to numerical overflow.

For a similar network with the activation function of the hidden layer set to a ReLU, we observe better results than with the sigmoid, especially with $\eta = 10^{-3}$ and $\lambda \leq 10^{-4}$, Figure 8.

Comparing the MSE obtained after a fixed number of epochs, varying the number of hidden layers, we observe a much lower MSE for 9 or less layers than for more; in particular, the best fits arise as a result of 2 to 6 20-node hidden layers in this case after 500 epochs, Figure 15. This shows that increasing the model complexity does not mean automatically

that get a lower error. This should be kept in mind when studying more complex problems.

Building a network with two hidden layers (both using the sigmoid as the activation function), we compare differences in predictions given by initializing the neurons' biases to either 0 or 1, Figure 14. The predictions over the first few epochs are much closer for a lower initial bias, but the two variants produce results very close to each other after a dozen iterations, although in the long run, the MSE for networks initialized with a higher bias tends to be ever so slightly lower than with a lower initial bias (in this case, $\text{MSE}_{\text{initial bias=1}} \approx 0.135$ and $\text{MSE}_{\text{initial bias=0}} \approx 0.140$).

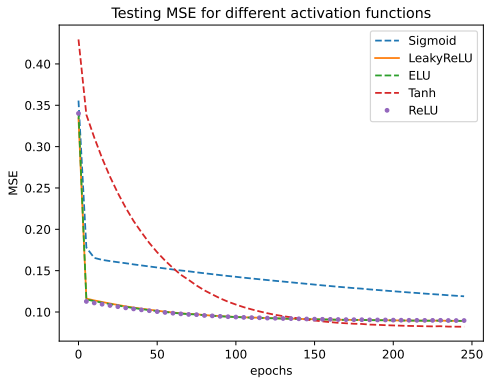

Figure 9: MSE for Franke's function with our feedforward neural network as a function of epochs for different activation functions. Here, the ELU and Leaky ReLU are both computed with $\alpha = 0.05$. The network is composed of a 2 hidden layers with 20 nodes each and an output layer. The first hidden layer has a sigmoid as an activation function and the second hidden layer can have different activation functions. The output layer as a linear function as the activation function.

From Figure 9, generally speaking, for the case of predicting Franke's function, the sigmoid, leaky ReLU, ReLU and ELU all converge at equivalent rates, with the sigmoid producing worse fits; using the hyperbolic tangent as the activation function results in a network that takes more time to converge, but results in a lower MSE. It's interesting to note that the ReLU, leaky ReLU and ELU produce largely identical results to each other (which incidentally makes this a difficult plot to show) over the same number of epochs in this particular setup.

Classification on the MNIST Data Set

In the last section, we have shown that out implementation of SGD and of a feedforward neural network is correct for linear regression type problems. In this section our aim is to validate its capabilities of classifying data correctly. For this we decided to use a subset of the MNIST data set, provided by SciKit Learn [12]. This data set contains about 1800 labeled 8 by 8 pixels images of handwritten digits form 0 to 10, Figure 10.

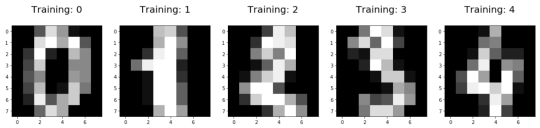

Figure 10: Example of data points from the MNIST data set.

To analyse this data set, we tested different neural networks architectures and found that the one that produced better results was a neural network with two hidden layers, with 20 nodes each and a sigmoid as the activation function. The output layer also uses a sigmoid as the activation function, but with 10 output nodes. We also used SGD as the gradient method, with a batch size of 5, over 500 epochs.

In Figure 11, we can observe the grid search for accuracy of our network on the classification of the MNIST data set, for different values of the learning rate and regularization parameter. We found that, instead of a lower learning rate, this classification problem performs better with a higher learning rate value.

A noteworthy aspect is that out neural network can successfully classify at best 98% the test data points.

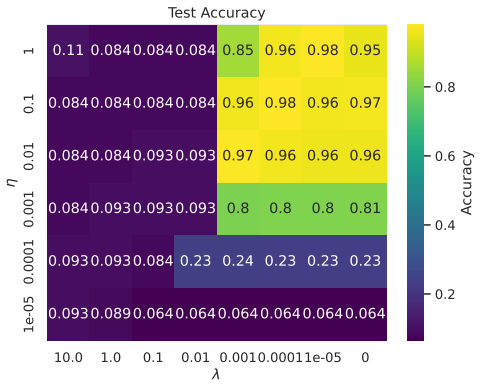

Figure 11: Grid search for the optimal MSE for the MNIST dataset with our feedforward neural network as a function of the learning rate and regularization parameter. The number of epochs was

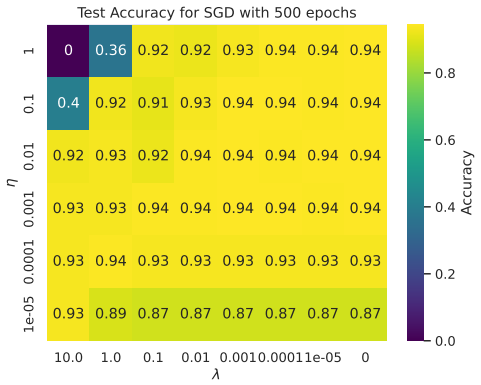

Figure 12: Grid search for the optimal MSE for the cancer data with standard logistic regression as a function of the learning rate η and ℓ_2 regularization parameter λ using SGD. The number of epochs used is 500.

Classification on the Wisconsin Breast Cancer Data Set

Having tested our implementation of a feedforward neural network on Franke's function and the MNIST data set, we now move on to the study of the famous Wisconsin Breast Cancer data set. This data set is comprised of features, such as area, radius, texture, etc, taken from medical imaging, of potentially tumours, labeled as malignant or benign.

In Figure 16, we can see the correlation matrix for the Breast Cancer data set. As it is expected, we find a high correlation between features such as mean radius and mean area or mean perimeter. To note, mean radius, mean area and mean perimeter all share a high correlation with mean concave points.

We first use standard logistic regression, where we initialize the features vector with a uniform distribution between 0 an 1, with SGD as the gradient method. In Figure 12, we see a grid search for the optimal values of the learning rate and regularization parameter. After 500 epochs, we find an accuracy of 94% at best with various hyperparameter combinations, especially at lower values of the regularization parameter $\lambda \leq 0.01$ and higher values of the learning rate $\eta \geq 10^{-3}$.

We repeat the same exercise with our feedforward neural network. We tested a vast array of architectures and found good results with a network with 2 hidden layers, each with 20 nodes and the sigmoid as the activation function. The output layer also uses a sigmoid as the activation function. We also used the cross-entropy as the cost function, since we are dealing with a classification problem.

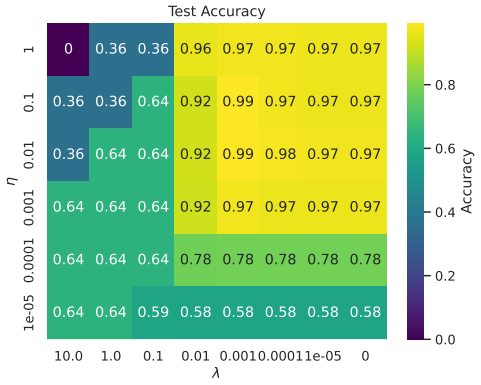

Figure 13: Grid search for the optimal MSE for the cancer data with our feedforward neural network as a function of the learning rate η and ℓ_2 regularization parameter λ using SGD. The number of epochs used is 500.

In Figure 13, we can see that the accuracy of the predictions, compared with logistic regression, increases to around 99%, after 500 epochs, for $\eta \in [0.1, 0.01]$, $\lambda = 10^{-3}$. However, the accuracy decreases faster as a function of the distance to the optimal learning rate and regularization parameter than with standard logistic regression: with $(\eta, \lambda) = (10^{-5}, 1)$, we obtain $\text{Accuracy}_{\text{SGD}} \approx 0.89$ whereas $\text{Accuracy}_{\text{FFNN}} \approx 0.64$. This is an indication that, while the neural network has the potential to perform better, it is more sensitive to the hyperparameters, meaning there's a higher guarantee to obtain better predictions with standard linear regression from a naive starting set of (η, λ) without the added overhead brought by the tweaking of these parameters.

We noted while testing out some architectures on our neural network, some of which we were expecting to fail, such as having no hidden layers and a linear function as the output layer's activation function, that the network could surprisingly reproduce 95% of the test Breast Cancer data set correctly. However, in this case, we cannot interpret the output of the neural network as probabilities anymore, since some are negative or larger than 1. This shows that even though the test accuracy might seem high, the interpretation might be wrong, so it is in the user's interest to always double-check the validity of the network.

CONCLUSION

We started by studying and comparing the performance of gradient methods, namely Gradient Descent and mini-batch Stochastic Gradient Descent. We found that mini-batch Stochastic Gradient Descent is more efficient and reliable than Gradient Descent given that the number of batches is relatively small and it can overcome confinement to local minima.

We implemented a code for a feedforward neural network with a flexible number of layers, nodes per layer and a vast array of activation functions to choose from. The training process is achieved by optimization methods, using the backpropagation algorithm for fast computation of gradients. This network was tested for both regression and classification problems to validate its performance. We also found that Rectified Linear Unit, leaky Rectified Linear Unit and Exponential Linear Unit as cost functions can provide a faster convergence of the mean squared error, while the hyperbolic tangent might give slower convergence.

Firstly, we applied both the neural network and standard linear regression models with mini-batch Stochastic Gradient Descent to data generated by Franke's function. We found that the linear model (MSE ≈ 0.075) performs slightly worse than our neural network (MSE ≈ 0.027).

We then applied this functioning network code to a subset of the MNIST data set and found that out network is able to achieve an accuracy of 98% on the recognition of low-resolution handwritten digits.

Lastly, we studied the Wisconsin Breast Cancer data set, and found some expected and some unexpected correlations between some of the features of the data set. We also applied standard logistic regression with mini-batch Stochastic Gradient Descent and a neural network to compare their performance. We found that logistic regression might give a worse accuracy (94%) than with the neural network (99%), although it is less susceptible to changes in hyperparameters.

We have shown that our feedforward neural network implementation is a suitable adaptive model to predict both regression- and classification-type problems. In particular, this machine learning implementation predicts better results than standard regression models for the Wisconsin Breast Cancer data set, our focus data.

[1] W. S. McCulloch and P. Walter, A logical calculus of the ideas immanent in nervous activity. bulletin of mathematical biophysics, Bulletin of Mathematical Biophysics (1944).

[2] F. Rosenblatt, The perceptron: a probabilistic model for information storage and organization in the brain., Psychological review **65**, 386 (1958).

[3] B. Widrow and M. E. Hoff, *Adaptive switching circuits*, Tech. Rep. (Stanford Univ Ca Stanford Electronics Labs, 1960).

[4] J. Bonnans, J. Gilbert, C. Lemarechal, and C. Sagastizábal, *Numerical Optimization: Theoretical and Practical Aspects*, Universitext (Springer Berlin Heidelberg, 2013).

[5] E. Süli and D. F. Mayers, *An Introduction to Numerical Analysis* (Cambridge University Press, 2003).

[6] Y. Bengio, Practical recommendations for gradient-based training of deep architectures (2012), arXiv:1206.5533 [cs.LG].

[7] T. Hastie, R. Tibshirani, and J. H. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed., Springer series in statistics (Springer, 2009).

[8] M. Nielsen, Neural networks and deep learning, `http://neuralnetworksanddeeplearning.com/index.html` (2019).

[9] C. M. Bishop, *Pattern Recognition and Machine Learning*, Information Science and Statistics (Springer New York, 2016).

[10] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, 2016) `http://www.deeplearningbook.org`.

[11] R. Franke, *A critical comparison of some methods for interpolation of scattered data*, Tech. Rep. (Naval Postgraduate School Monterey CA, 1979).

[12] G. Varoquaux, Recognizing hand-written digits, `https://scikit-learn.org/stable/auto_examples/classification/plot_digits_classification.html` (2021), accessed: 2021-11-13.

Appendix A: Figures

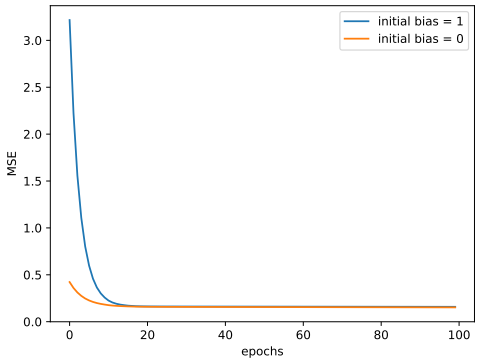

Figure 14: MSE for Franke's function with our feedforward neural network as a function of epochs for two different ways of initializing the bias vector. The network is composed of a 2 hidden layers with 20 nodes each and a sigmoid as the activation function, and an output layer.

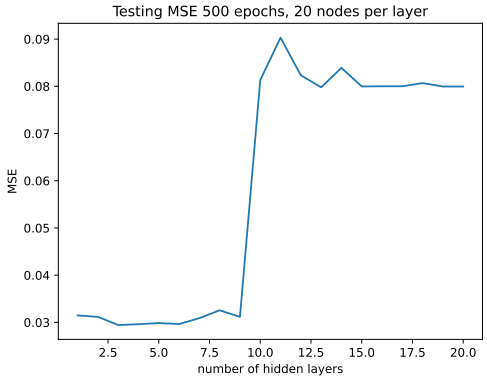

Figure 15: MSE for Franke's function with our feedforward neural network as a function of hidden layers after 500 epochs. Each hidden layer has 20 nodes, with the sigmoid as the activation function.

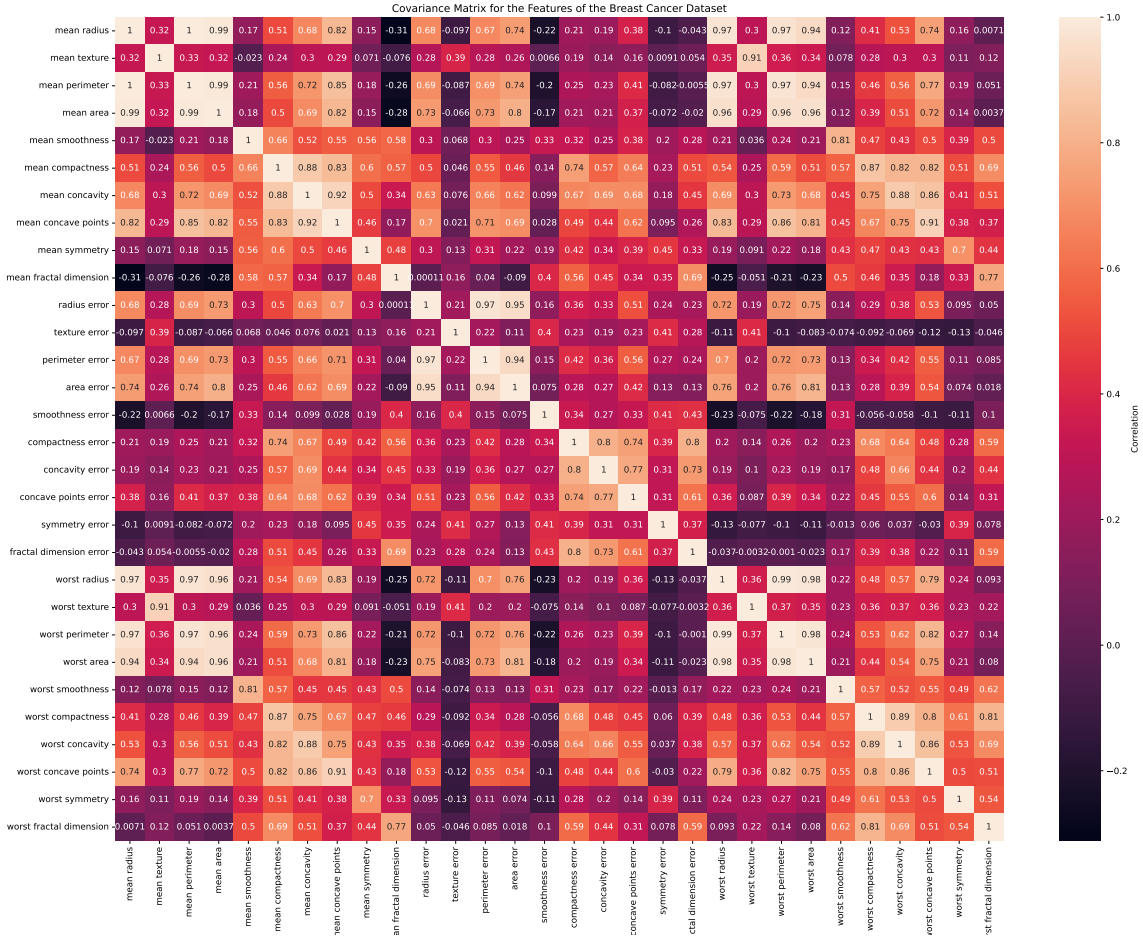

Figure 16: Correlation matrix for the features of the Wisconsin Brest Cancer data set.

Appendix B: Backpropagation

Define $z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}$, i.e. the weighted sum of the activation of the previous layer, and $a^{(l)} = \sigma^{(l)}(z^{(l)})$, where $\sigma^{(l)}$ is the activation function of layer l. Assume further that the cost fucntion can be written as

$$C(a_1^{(L)}(x_1),\dots,a_1^{(L)}(x_n),a_2^{(L)}(x_1),\dots,a_2^{(L)}(x_n),\dots,a_{n_L}^{(L)}(x_n)) = \sum_{i=1}^{n}\sum_{k_L}^{n_L} C_{i,k_L}(a_{k_L}^{(L)}(x_i)),$$

where $a_{k_L}^{(L)}(x_i)$ is the output of the activation of node $a_{k_L}^{(L)}$ with respect to the datapoint x_i, and C_{i,n_L} is the cost function using only the output $a_{k_L}^{(L)}$. Denoting $\delta_j^{(l)} := \frac{\partial C}{\partial z_j^{(l)}}$, backpropagation consist of computing

$$\frac{\partial C}{\partial z_j^{(L)}} = \frac{\partial C}{\partial a_j^{(L)}}\sigma^{(L)\prime}(z_i^{(L)}), \tag{B1}$$

$$\frac{\partial C}{\partial z_j^{(l)}} = \sigma^{(l)\prime}(z_j^{(l)})\cdot\sum_{i=1}^{n_l} W_{ij}^{(l+1)}\delta_i^{(l+1)}, \tag{B2}$$

$$\frac{\partial C}{\partial b_j^{(l)}} = \delta_j^{(l)}, \tag{B3}$$

$$\frac{\partial C}{\partial W_{jk}^{(l)}} = a_k^{(l-1)}\delta_j^{(l)}. \tag{B4}$$

We derive these equations for a single data point. It is from this easy to generalize to the full cost function. Let n_l denote the number of nodes in layer l, $\sigma^{(l)}$ be the activation function of layer l, $a_i^{(l)}$ be the activation $\sigma^{(l)}(z_i^{(l)})$ of node i in layer l (for $l=0$ the activation will be the input), $z_i^{(l)} = \sum_{j=1}^{n_l} W_{ij}^{(l)}a_j^{(l-1)} + b_i^{(l)}$ be the weighted sum of the inputs to node i in layer l. We let $z^{(l)} = (z_1^{(l)},\dots,z_{n_l}^{(l)})^T$, where $\sigma^{(l)}(z^{(l)}) = (\sigma^{(l)}(z_1^{(l)}),\dots,\sigma^{(l)}(z_{n_l}^{(l)}))^T$, and $a^{(l)} = (a_1^{(l)},\dots,a_{n_l}^{(l)})^T$. Denote the final layer by L. Notice that we can write the cost function as $C(a^{(L)}) = \sum_{k_L=1}^{n_L} C_{k_L}(a_{k_L}^{(L)})$. We begin by computing

$$\begin{aligned}\frac{\partial C}{\partial z_i^{(L)}} = \sum_{k_L=1}^{n_L}\frac{\partial}{\partial z_i^{(L)}}C_{k_L}(a_{k_L}^{(L)}) = \sum_{k_L=1}^{n_L}\frac{\partial C_{k_L}}{\partial a_{k_L}^{(L)}}\frac{\partial a_{k_L}^{(L)}}{\partial z_i^{(L)}} &= \frac{\partial C_i}{\partial a_i^{(L)}}\sigma^{(L)\prime}(z_i^{(L)}) \\ &= \sigma^{(L)\prime}(z_i^{(L)})\sum_{k_L=1}^{n_L}\frac{\partial C_{k_L}}{\partial a_i^{(L)}} = \frac{\partial C}{\partial a_i^{(L)}}\sigma^{(L)\prime}(z_i^{(L)}) := \delta_i^{(L)}\end{aligned}$$

In general we will have

$$\frac{\partial z_i^{(l)}}{\partial z_j^{(l-1)}} = \frac{\partial}{\partial z_j^{(l-1)}}\left(\sum_{k_{l-1}}^{n_{l-1}} W_{ik_{l-1}}^{(l)}\sigma^{(l-1)}(z_{k_{l-1}}^{(l-1)}) + b_i^{(l)}\right) = W_{ij}^{(l)}\sigma^{(l-1)\prime}(z_j^{(l-1)}).$$

To see how we get to the general case, we look at how we can compute $\delta_i^{(L-1)}$ and $\delta_i^{(L-2)}$. We get

$$\begin{aligned}
\frac{\partial C}{\partial z_i^{(L-1)}} &= \sum_{k_L=1}^{n_L} \frac{\partial C_{k_L}(\sigma^{(L)}(z_{k_L}^{(L)}))}{\partial z_i^{(L-1)}} = \sum_{k_L=1}^{n_L} \frac{\partial C_{k_L}(\sigma^{(L)}(z_{k_L}^{(L)}))}{\partial z_{k_L}^{(L)}} \frac{\partial z_{k_L}^{(L)}}{\partial z_i^{(L-1)}} = \sum_{k_L=1}^{n_L} \overbrace{\frac{\partial C_{k_L}(a_{k_L}^{(L)})}{\partial z_{k_L}^{(L)}}}^{=\delta_{k_L}^{(L)}} \frac{\partial z_{k_L}^{(L)}}{\partial z_i^{(L-1)}} \\
&= \sum_{k_L=1}^{n_L} \delta_{k_L}^{(L)} \frac{\partial}{\partial z_i^{(L-1)}} \left(\sum_{k_{L-1}}^{n_{L-1}} W_{k_L k_{L-1}}^{(L)} \sigma^{(L-1)}(z_{k_{L-1}}^{(L-1)}) + b_{k_L}^{(L)} \right) = \sum_{k_L}^{n_L} \delta_{k_L}^{(L)} W_{k_L i}^{(L)} \sigma^{(L-1)\prime}(z_i^{(L-1)}) \\
&= \left((W^{(L)})^T \delta^{(L)} \right)_i \sigma^{(L-1)\prime}(z_i^{(L-1)}) := \delta_i^{(L-1)}.
\end{aligned}$$

Next, we get

$$\begin{aligned}
\frac{\partial C}{\partial z_i^{(L-2)}} &= \sum_{k_L=1}^{n_L} \frac{\partial}{\partial z_i^{(L-2)}} C_{k_L} \Bigg(\sigma^{(L)} \underbrace{\Big(\sum_{k_{L-1}}^{n_{L-1}} W_{k_L k_{L-1}}^{(L)} \sigma^{(L-1)} \overbrace{\Big(\sum_{k_{L-2}}^{n_{L-2}} W_{k_{L-1} k_{L-2}}^{(L-1)} \sigma^{(L-2)}(z_{k_{L-2}}^{(L-2)}) + b_{k_{L-1}}^{(L-1)} \Big)}^{=z_{k_{L-1}}^{(L-1)}} + b_{k_L}^{(L)} \Big)}_{=z_{k_L}^{(L)}} \Bigg) \\
&= \sum_{k_L=1}^{n_L} \overbrace{\frac{\partial C_{k_L}}{\partial z_{k_L}^{(L)}}}^{=\delta_{k_L}^{(L)}} \frac{\partial z_{k_L}^{(L)}}{\partial z_i^{(L-2)}} = \sum_{k_L=1}^{n_L} \delta_{k_L}^{(L)} \sum_{k_{L-1}}^{n_{L-1}} W_{k_L k_{l-1}}^{(L)} \sigma^{(L-1)\prime}(z_{k_{L-1}}^{(L-1)}) \frac{\partial z_{k_{L-1}}^{(L-1)}}{\partial z_i^{(L-2)}} \\
&= \sum_{k_{L-1}}^{n_{L-1}} \frac{\partial z_{k_{L-1}}^{(L-1)}}{\partial z_i^{(L-2)}} \overbrace{\sum_{k_L=1}^{n_L} \delta_{k_L}^{(L)} W_{k_L k_{l-1}}^{(L)} \sigma^{(L-1)\prime}(z_{k_{L-1}}^{(L-1)})}^{=\delta_{k_{L-1}}^{(L-1)}} = \sum_{k_{L-1}}^{n_{L-1}} W_{k_{L-1} i}^{(L-1)} \sigma^{(L-2)\prime}(z_i^{(L-2)}) \delta_{k_{L-1}}^{(L-1)} \\
&= \left((W^{(L-1)})^T \delta_{k_{L-1}}^{(L-1)} \right)_i \sigma^{(L-2)\prime}(z_i^{(L-2)}) := \delta_i^{(L-1)}.
\end{aligned}$$

We now see the pattern: Applying the chain rule with z_i^k for all $k = L, \ldots, l+1$ and all nodes i in the respective layer, we are able to find δ_i^k after we change the order of summation. The mathematical derivation is

$$\begin{aligned}
\frac{\partial C}{\partial z_i^{(l)}} &= \sum_{k_L=1}^{L} \delta_{k_L}^{(L)} \sum_{k_{L-1}=1}^{n_{L-1}} W_{k_L k_{L-1}}^{(L)} \sigma^{(L-1)\prime}(z_{k_{L-1}}^{(L-1)}) \sum_{k_{L-2}=1}^{n_{L-2}} \cdots \sum_{k_{l+1}=1}^{n_{l+1}} W_{k_{l+2} k_{l+1}}^{(l+1)} \sigma^{(l+1)\prime}(z_{k_{l+1}}^{(l+1)}) \frac{\partial z_{k_{l+1}}^{(l+1)}}{\partial z_i^{(l)}} \\
&= \sum_{k_{l+1}=1}^{n_{l+1}} \underbrace{\cdots \sum_{k_{L-1}=1}^{n_{L-1}} \overbrace{\sum_{k_L=1}^{n_L} \delta_{k_L}^{(L)} W_{k_L k_{L-1}}^{(L)} \sigma^{(L-1)\prime}(z_{k_{L-1}}^{(L-1)})}^{\delta_{k_{l+1}}^{(l+1)}} W_{k_{L-1} k_{L-2}}^{(L-1)} \sigma^{(L-2)\prime}(z_{k_{L-2}}^{(L-2)}) \cdots W_{k_{l+2} k_{l+1}}^{(l+2)} \sigma^{(l+1)\prime}(z_{k_{l+1}}^{(l+1)})}_{\delta_{k_{l+1}}^{(l+1)}} W_{k_{l+1} k_l}^{(l+1)} \sigma^{(l)\prime}(z_i^{(l)}) \\
&= \sum_{k_{l+1}=1}^{n_{l+1}} \delta_{k_{l+1}}^{(l+1)} W_{k_{l+1} k_l}^{(l+1)} \sigma^{(l)\prime}(z_i^{(l)}) = \left((W^{(l+1)})^T \delta^{(l+1)} \right)_i \sigma^{(l)\prime}(z_i^{(l)}) := \delta_i^{(l)}.
\end{aligned}$$

In the same fashion, we find

$$\frac{\partial C}{\partial b_i^{(L)}} = \sum_{k_L=1}^{L} \frac{\partial C_{k_L}(a_{k_L}^{(L)})}{\partial z_{k_L}^{(L)}} \frac{\partial z_{k_L}^{(L)}}{\partial b_i^{(L)}} = \delta_i^{(L)}$$

and

$$\frac{\partial C}{\partial b_i^{(l)}} = \sum_{k_l=1}^{n_l} \delta_{k_l}^{(l)} \frac{\partial z_{k_l}^{(l)}}{\partial b_i^{(l)}} = \delta_i^{(l)}$$

for the biases, and

$$\frac{\partial C}{\partial W_{ij}^{(L)}} = \sum_{k_L=1}^{L} \frac{\partial C_{k_L}(a_{k_L}^{(L)})}{\partial z_{k_L}^{(L)}} \frac{\partial z_{k_L}^{(L)}}{\partial W_{ij}^{(L)}} = \delta_i^{(L)} a_j^{(L-1)}$$

and

$$\frac{\partial C}{\partial W_{ij}^{(l)}} = \sum_{k_l=1}^{n_l} \delta_{k_l}^{(l)} \frac{\partial z_{k_l}^{(l)}}{\partial W_{ij}^{(l)}} = \delta_i^{(l)} a_j^{(l-1)}$$

for the weights. We have thus proven equations (B1) to (B4), and we can easily write them as vector and matrix equations as follows:

$$\begin{aligned}
\frac{\partial C}{\partial z^{(L)}} &= \frac{\partial C}{\partial a^{(L)}} \odot \sigma^{L\prime}(z^{(L)}), \\
\frac{\partial C}{\partial z^{(l)}} &= \sigma^{(l)\prime}(z^{(l)}) \odot ((W^{(l+1)})^T \delta^{(l+1)}), \\
\frac{\partial C}{\partial b^{(l)}} &= \delta^{(l)}, \\
\frac{\partial C}{\partial W^{(l)}} &= \delta^{(l)} (a^{(l-1)})^T,
\end{aligned}$$

where $\odot$ denotes the Hadamard/elementwise product. If we now have n data points, the cost function will sum over the cost of each data points. If we let

$$\delta^{(l)} = \begin{pmatrix} \delta_1^{(l)}(x_1) & \delta_1^{(l)}(x_2) & \dots & \delta_1^{(l)}(x_n) \\ \delta_2^{(l)}(x_1) & \delta_2^{(l)}(x_2) & \dots & \delta_2^{(l)}(x_n) \\ \vdots & \vdots & & \vdots \\ \delta_{n_l}^{(l)}(x_1) & \delta_{n_l}^{(l)}(x_2) & \dots & \delta_{n_l}^{(l)}(x_n) \end{pmatrix} = \begin{pmatrix} \delta^{(l)}(x_1) & \delta^{(l)}(x_2) & \dots & \delta^{(l)}(x_n) \end{pmatrix}$$

and

$$a^{(l)} = \begin{pmatrix} a_1^{(l)}(x_1) & a_1^{(l)}(x_2) & \dots & a_1^{(l)}(x_n) \\ a_2^{(l)}(x_1) & a_2^{(l)}(x_2) & \dots & a_2^{(l)}(x_n) \\ \vdots & \vdots & & \vdots \\ a_{n_l}^{(l)}(x_1) & a_{n_l}^{(l)}(x_2) & \dots & a_{n_l}^{(l)}(x_n) \end{pmatrix},$$

then

$$\frac{\partial C}{\partial b^{(l)}} = \sum_{i=1}^{n} \delta^{(l)}(x_i) \quad \text{and} \quad \frac{\partial C}{\partial W^{(l)}} = \delta^{(l)}(a^{(l-1)})^T.$$