Project 2
FYS-STK3155/4155

Magnus Bergkvam

November 17, 2023

1 Abstract

For the last decade especially, neural networks have been used for more and more purposes, with them often doing very well in highly complex tasks, with regression and classification being two of its common applications.

Motivated in exploring these two applications further, we have in this project tested out how well a basic neural network performs in a regression and a binary classification example. We then compared this with the linear models we explored in the previous project [1] (in the case of regression), as well as our own implementation of logistic regression (for classification). For this we have implemented our own neural network code from scratch, where we have used gradient methods paired with the backpropagation-algorithm. With gradient optimization being a central part of the development of both neural networks and logistic regression we explored ways we can most effectively optimize the cost function through the Newton-Raphson method, with the various hyperparameters being central. We also looked at various learning-rate schedulers, namely Adam, AdaGrad and RMSProp, and also explored with using regularization for our networks.

What we saw was that in the regression example (using the franke-function we explored in project 1 [1]), with regularization, we got some considerable performance gains compared to the linear models we tested in project 1, while for classification the neural network did emerge as better, but with a much finer margin. In the classification result we were able to predict 100% of the observations in the test-set for our final model, with the AdaGrad learning-rate scheduler and the sigmoid activation function with two hidden layers giving us the best fit. Throughout the project we have manifested the power of neural networks, while simultaneously demystified its components and the effect that some important hyperparameters have. It comes out that the learning-rate (and learning-rate scheduler) and regularization parameter are especially central, while other parameters are less central, but still noticeable.

2 Introduction

In the last 10 or so years, neural networks have really exploded in popularity. Various different types of neural networks have proved effective for various different applications all from speech-recognition, to image processing, to complex language tasks. But how powerful are they really in comparison to linear models or logistic regression when it comes to basic regression/classification, and how do we get the most performance out of them?

These are some of the things which we aim to tackle here. To do this we will implement a rather simple multi-layer perceptron model from scratch using gradient methods with the backpropagation algorithm for the training. We will then use our own neural network model for both regression and classification. For the regression we will be generating data from the franke-function as we did in project 1, and of course compare this to the results we got with the linear models we explored there. For the classification data we will try to classify whether a patient has cancer or not based on various medical features, where the wisconsin breastcancer data [2] is the data we will be using for that. Additionally we will explore various methods of gradient optimizations, as both the methods we will implement in this project (i.e. neural networks and logistic regression) have cost-functions which is typically minimized w.r.t. its parameters using some sort of gradient method. As already mentioned we will implement and explore performances based on various optimization algorithms, with the ones we will look at in this project being plain gradient descent, stochastic gradient descent, momentum, AdaGrad, RMSProp and Adam. For all our different methods we will compare our own implementations against established libraries, namely tensorflow and scikit-learn.

There are mainly two reasons why we are specifically focusing on using neural networks in this project. One is, as already mentioned, that they have become very popular in the last decade or so, with them being used for more and more tasks. With their usage increasing it becomes important to understand the way a model works, and how we get the most amount of performance out of them by tweaking the various different parts of the network. Additionally since they, in comparison to a lot of other models, are quite computationally expensive it is important to realize which of the parameters are the most relevant to tweak, and which are not so relevant, as this can save a lot of time. Another reason is because of the universal approximation theorem applying to neural networks. We will go more deeply into this in 3.7.5, but the essence is that a sufficiently big neural network should be able approximate any function multidimensional function to an arbitrary accuracy [3, s. 13.5]. We will see if this property is able to give the neural networks good results for the regression/classification examples and if we then are able to outperform models not having this property (linear models / logistic regression).

We will now get into the methods, before we represent our results, do some analysis on these results before we conclude. In both the methods and results we start by look deeper into the pure optimization parts, before going into the models themselves (i.e. neural networks and logistic regression).

3 Methods

3.1 Basic optimization data

In part of our analysis we will explore the different parameters for the different gradient methods. For this we will just use some simple arbitrary data generated from a fourth order polynomial. The polynomial I have chosen to generate the data from is:

$$f(x) = 4 - 6x^2 + x^4.$$

We then just generate a grid of 201 evenly spaced values $\boldsymbol{x}$ on $[-2, 2]$ and generate the response as follows:

$$\boldsymbol{y} = f(\boldsymbol{x}) + \boldsymbol{\epsilon}$$

, where $\epsilon_i \sim N(0, 0.2^2)$, and f is applied element-wise. We will use the linear regression and ridge regression for the models, and a design matrix with fourth-order polynomial features. This way analytical least-squares solution should give us optimal parameters close to $(4, 0, -6, 0, 1)^T$. We will mainly explore the effect that the learning-rate and regularization terms have, and also quickly see the effect of the amount of epochs and minibatch-size can have. We will also try out some different optimization methods, more specifically gradient descent, stochastic gradient descent, momentum and AdaGrad (more about these later), and see which performs the best, as well as how much of a difference the choice of method makes. When exploring the stochastic gradient descent models (see 3.5.2 for info on SGD) we will also utilize a time-decay function for the learning-rate (see 3.5.1 for info on time-decay), as this can potentially help convergence and speed of learning. For these cases we will use the following simple time-decay function:

$$f(t) = \frac{1}{t + 10}.$$

3.2 The franke function

The franke-function is a 2-dimensional function which we already have explored in project 1 [1], so I won't go too deeply into this here. The function is given by:

$$\begin{aligned} f(x, y) = {} & \frac{3}{4} \exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) + \frac{3}{4} \exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right) \\ & + \frac{1}{2} \exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) - \frac{1}{5} \exp\left(-(9x-4)^2 - (9y-7)^2\right). \end{aligned}$$

Again we will draw 200 of random observations $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, with $(x_1)_i \in [0,1]$, and $(x_2)_i \in [0,1]$ for $1 \leq i \leq 200$. The response will be generated by:

$$\boldsymbol{y} = f(\boldsymbol{x}_1, \boldsymbol{x}_2) + \boldsymbol{\epsilon}$$

, with $\epsilon_i \sim N(0, 0.1^2)$ (we apply f element-wise). We will try to fit a neural network to this data with this time only $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ as the response. In project 1 we added polynomial features of various degrees to fit to a model. Here we do not need to do this as we can just add enough nodes and the universal approximation theorem assures us that the network should do a good job of approximating the function.

3.3 The wisconsin cancer data

The wisconsin cancer data [2] is a dataset containing breast cancer cases, with a response-variable indicating wether the patient has breast-cancer or not, with some corresponding medical features. This is clearly a binary classification case, which makes this suitable for both neural networks and logistic regression models. We will use the dataset included in *sklearn.datasets* [4]. To load the data into python and splitting into train, validation and test sets, we can just use the following code:

```
X, y = load_breast_cancer(return_X_y=True)
y = y.reshape(len(y), 1)

# Train-test-validation splitting
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.3)
X_val, X_test, y_val, y_test = train_test_split(X_val, y_val, test_size=0.4)
```

, after importing *load_breast_cancer* from *sklearn.datasets* [4] that is.

We split into train, test and validation sets because we will evaluate a lot of models on the validation set, and then it becomes increasingly important to evaluate the final selected model on some test set in order for us to not get too optimistic of a final metric [5, s. 7.2]. The data is limited to 569 observations with each having 30 explanatory variables. This makes both the validation set and test set rather small, so with around 100 and 70 observations in each respectively. We then need to be a little skeptical towards the final metrics, as the metrics may be a little specific to the split of data.

3.4 The Newton-Raphson method

Very central to the numerical optimization we do is the Newton-Raphson method. This is a method which allows us to find the minimum of some function by using its gradient. The method has it's origins in the Taylor-expansion of the function we wish to minimize. In our case we use this to minimize cost-functions for the models we use. Let $C(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y})$ be some general cost-function depending on some design matrix $\boldsymbol{X}$ and a response vector $\boldsymbol{y}$. We wish to find the parameter $\boldsymbol{\theta}$ which minimizes this cost, given $\boldsymbol{X}$ and $\boldsymbol{y}$, i.e. we minimize C with respect to $\boldsymbol{\theta}$, while $\boldsymbol{X}$ and $\boldsymbol{y}$ are obviously fixed. We can then approximate the cost using a Taylor-expansion of second order around some value $\boldsymbol{\theta}^*$, in which case we get:

$$\begin{aligned}C(\boldsymbol{\theta}, \boldsymbol{X}, \boldsymbol{y}) &\approx C(\boldsymbol{\theta}^*; \boldsymbol{X}, \boldsymbol{y}) + \frac{\partial C(\boldsymbol{\theta}^*; \boldsymbol{X}, \boldsymbol{y})}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta} - \boldsymbol{\theta}^*) + \frac{1}{2}\frac{\partial^2 C(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y})}{\partial^2 \boldsymbol{\theta}}(\boldsymbol{\theta} - \boldsymbol{\theta}^*)^2 \\ &= C(\boldsymbol{\theta}^*; \boldsymbol{X}, \boldsymbol{y}) + \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^*)(\boldsymbol{\theta} - \boldsymbol{\theta}^*) + \frac{1}{2}\boldsymbol{H}_{\boldsymbol{\theta}}(\boldsymbol{\theta}^*)(\boldsymbol{\theta} - \boldsymbol{\theta}^*)^2\end{aligned}$$

Where $\nabla_{\boldsymbol{\theta}} C$ is the gradient of C w.r.t. $\boldsymbol{\theta}$ and $\boldsymbol{H}_{\boldsymbol{\theta}}$ is the Hesse-matrix of C w.r.t. $\boldsymbol{\theta}$. This should be a good approximation for C sufficiently close to $\boldsymbol{\theta}^*$. We then can find the gradient of this w.r.t. $\boldsymbol{\theta}$:

$$\frac{\partial C(\boldsymbol{\theta}, \boldsymbol{X}, \boldsymbol{y})}{\partial \boldsymbol{\theta}} \approx \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^*) + \boldsymbol{H}_{\boldsymbol{\theta}}(\boldsymbol{\theta}^*)(\boldsymbol{\beta} - \boldsymbol{\theta}^*)$$

If we set this to 0 to find the minimum we further get:

$$\boldsymbol{H}_{\boldsymbol{\theta}}(\boldsymbol{\theta}^*)(\boldsymbol{\theta} - \boldsymbol{\theta}^*) = -\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^*)$$

which leads to

$$\boldsymbol{\theta} - \boldsymbol{\theta}^* = -\boldsymbol{H}_{\boldsymbol{\theta}}(\boldsymbol{\theta}^*)^{-1}\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^*)$$

and finally

$$\boldsymbol{\theta} = \boldsymbol{\theta}^* - \boldsymbol{H}_{\boldsymbol{\theta}}(\boldsymbol{\theta}^*)^{-1}\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^*)$$

This tells us that the minimum of the Taylor-expansion of the cost-function, using any $\boldsymbol{\theta}^*$ is given as above. The problem with this is that this is using the Taylor-expansion, which is only reliable close to $\boldsymbol{\theta}^*$ so chances are that using this formula will not give us a global minimum. However it generally tends to bring us closer to the minimum we are looking after. What we then usually do is to start with some initial guess for the optimal $\boldsymbol{\theta}$, which we denote $\boldsymbol{\theta}^{(0)}$. We then for $k = 1, \dots$ let $\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}^{(k-1)} - \boldsymbol{H}_{\boldsymbol{\theta}}(\boldsymbol{\theta}^{(k-1)})^{-1}\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(k-1)})$ all the way until we have convergence.

3.5 Model optimizers and learning-rate schedulers

Something which often is problematic when using the Newton-Raphson method as described over is that we need to estimate the Hessian matrix and also invert it. First of all finding a way to calculate the second derivative can be a difficulty in and of itself, but it may also be computationally expensive to do so. Not only that, we also need to invert this matrix which can also become very computationally expensive, and perhaps also numerically unstable with higher dimensional hesse matrices (remember that the Hessian matrix is a square matrix with the same dimensions as $\boldsymbol{\theta}$ in both width and height). Inverting this matrix then may not be a problem if we only have a few parameters, but for example when dealing with logistic regression with lots of features, or neural networks with lots of hidden nodes and layers this becomes unviable. We therefore will use various methods to approximate this hessian matrix.

3.5.1 Ordinary gradient descent

Ordinary gradient descent is a very simple way which we can avoid calculating this hessian matrix. Here we simply approximate it with some numerical learning-rate, which we will throughout this report denote η. This way our algorithm becomes:

$$\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}^{(k-1)} - \eta \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(k-1)}).$$

Often we may just keep η fixed throughout all the iterations, but we can also make it depend on k. For example one may wish to start with a higher learning-rate and make this decrease with the number of iterations. This method I will refer to as using time-decay. When using gradient descent for training our model, η then becomes a hyperparameter in this model. Often this hyperparameter is rather important in determining the fit of the model. We often want a rather high learning-rate as this will lead to faster convergence towards some optimum, however if we make it too big we may not get convergence at all. I will mainly throughout the code simply set up a grid of learning-rates and fit to all of them, and simply choose the learning-rate which gives the best performance for some fixed amount of epochs, but there also exists other more advanced ways of tuning this parameter.

3.5.2 Stochastic gradient descent

Stochastic gradient descent is very similar to ordinary gradient descent, except the fact that instead of training for each epoch on the whole dataset we split the data into n randomly chosen mini-batches each of some specified size (this is what *minibatch_size* in our code and report will refer to). Then we do training over all of the n mini-batches for each epoch, while the rest of the method remains the same.

3.5.3 Momentum

One problem with gradient descent and stochastic gradient descent which one often can see is that it can be quite slow to learn. Ideally we would like the model to minimize the cost/loss as quickly as possible, and therefore one might want to use methods designed to accelerate this learning process. One relatively simple such method is using a momentum, which often accelerates learning quite drastically especially in cases with high curvatures and small consistent or noisy gradients [6, s. 8.3.2]. This method introduces a variable $\boldsymbol{v}$ which describes the direction and speed which the parameter $\boldsymbol{\theta}$ is moving in parameter space [6, s. 8.3.2], much like the role of velocity in physics. To calculate this $\boldsymbol{v}$ we need a hyperparameter $\alpha \in [0, 1)$ (we will use $\alpha = 0.9$ throughout this project) and we initialize $\boldsymbol{v} = \mathbf{0}$. Then for each learning-iteration we update $\boldsymbol{v}$ the following way:

$$\boldsymbol{v} = \alpha \boldsymbol{v}_{prev} - \eta \Delta_{\boldsymbol{\theta}} C(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta}).$$

We then can simply update the $\boldsymbol{\theta}$ as follows:

$$\boldsymbol{\theta} = \boldsymbol{\theta}_{prev} + \boldsymbol{v}$$

We then see that the value of the $\boldsymbol{v}$ will depend on the current cost-gradient w.r.t. $\boldsymbol{\theta}$ as well as the previous $\boldsymbol{v}$, which again depends on the previous cost-gradient w.r.t. $\boldsymbol{\theta}$ and the previous previous $\boldsymbol{v}$ and so on. Therefore we see that $\boldsymbol{v}$ is influenced by all the previous cost-gradients, weighted differently, in contrast to just using the current cost-gradient, and granted they point in a somewhat similar direction, the $\boldsymbol{v}$ will grow bigger and bigger in size. It is therefore not that difficult to see that this can give quicker learning.

Keep in mind that in theory there is nothing stopping us from combining momentum with any of the other methods we discuss here, however methods like Adam already incorporates a form of momentum [6, s. 8.5.3], so we might not get much quicker learning by using it, if any at all.

3.5.4 AdaGrad

AdaGrad is another way designed to give faster learning. This is an algorithm which uses adaptive learning-rates. These have a difference in that each axis of the parameter-space $\boldsymbol{\theta}$ essentially get a separate learning-rate [6, s. 8.5], while these separate learning-rates change/adapt throughout the learning process, which is a different approach to using momentum. AdaGrad is a rather simple way of achieving a good such adaptive learning-rate. It scales down the parameters with big historical updates, and scales up the ones with small corresponding historical updates up [6, s. 8.5.1]. For AdaGrad we introduce one variable $\boldsymbol{r}$ which we initially set to $\mathbf{0}$, as well as a small constant δ which we typically set to 10^{-7}. The algorithms is as follows until the stopping-criterion is met [6, algorithm 8.4]:

- Sample a minibatch of m observations from the training-set consisting of $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ as the explanatory variables and $\{\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{m}\}$.
- Calculate the gradient $\boldsymbol{g}$ of the cost w.r.t. $\boldsymbol{\theta}$ using this minibatch.
- Update the $\boldsymbol{r}$ to become $\boldsymbol{r}_{prev} + \boldsymbol{g} \odot \boldsymbol{g}$ ($\odot$ being the Hadamard-product).
- Calculate $\Delta\boldsymbol{\theta} = -\eta \frac{1}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$ and update $\boldsymbol{\theta}$ accordingly (i.e. set $\boldsymbol{\theta} = \boldsymbol{\theta}_{prev} + \Delta\boldsymbol{\theta}$).

3.5.5 RMSProp

RMSProp is much like AdaGrad, but is designed to be more robust for non-convex cost-function settings [6, s. 8.5.2], like typical neural nets. Unlike AdaGrad where we shrink the learning-rates using the sizes of all the previous updates, RMSProp focuses less on the history of updates from very long ago, but otherwise using much of the same approach. The RMSProp algorithm requires therefore a decay-rate ρ as a hyperparameter as well, along the $\boldsymbol{r}$ which we again initialize to be $\mathbf{0}$. Here we also need a small constant δ, which typically is set to 10^{-6}. Then we do the following until the stopping-criterion is met [6, algorithm 8.5]:

- Sample a minibatch of m observations from the training-set consisting of $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ as the explanatory variables and $\{\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{m}\}$.
- Calculate the gradient $\boldsymbol{g}$ of the cost w.r.t. $\boldsymbol{\theta}$ using this minibatch.
- Update the $\boldsymbol{r}$ to become $\rho \boldsymbol{r}_{prev} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}$ ($\odot$ being the Hadamard-product)
- Calculate $\Delta\boldsymbol{\theta} = -\eta \frac{1}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$ and update $\boldsymbol{\theta}$ accordingly (i.e. set $\boldsymbol{\theta} = \boldsymbol{\theta}_{prev} + \Delta\boldsymbol{\theta}$).

3.5.6 Adam

Adam is a very popular learning-rate scheduler, and it is probably the method we will tackle here which most often gives the best results. [6, s. 8.6.1]. We start the algorithm by initializing the parameter $\boldsymbol{\theta}$ which we would like to optimize the cost/loss w.r.t. (typically we initialize this by drawing it randomly from for example the normal distribution which is what we have done in this project). We also set beforehand $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$ and $t = 0$. Here the algorithms is as follows until we meet the stopping criterion [6, algortihm 8.7]:

- Sample a minibatch of m observations from the training-set consisting of $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ as the explanatory variables and $\{\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{m}\}$.
- Calculate the gradient $\boldsymbol{g}$ of the cost w.r.t. $\boldsymbol{\theta}$ using this minibatch.
- Increment t by 1 (i.e. set $t = t_{prev} + 1$).
- Update $\boldsymbol{s}$ to become $\rho_1 \boldsymbol{s}_{prev} + (1-\rho_1)\boldsymbol{g}$.
- Update $\boldsymbol{r}$ to become $\rho_2 \boldsymbol{r}_{prev} + (1-\rho_2) g \odot g$ ($\odot$ again being the Hadamard-product).
- Correct the bias in the first moment by: $\hat{\boldsymbol{s}} = \frac{\boldsymbol{s}}{1-\rho_1^t}$.
- Correct the bias in the second moment by: $\hat{\boldsymbol{r}} = \frac{\boldsymbol{r}}{1-\rho_2^t}$.
- Finally calculate $\Delta\boldsymbol{\theta} = -\eta \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ and update $\boldsymbol{\theta}$ accordingly (i.e. set $\boldsymbol{\theta} = \boldsymbol{\theta}_{prev} + \Delta\boldsymbol{\theta}$).

Here η is referred to as the step-size (but we will use the term learning-rate instead), ρ_1 and ρ_2 are the decay-rate which must be in $[0, 1)$ (typically we set the defaults to $\rho_1 = 0.9$ and $\rho_2 = 0.999$), δ is just a small constant (we set it to 10^{-8}). Note that we for all these three adaptive methods we can in theory just drop the minibatch sampling and still get a functioning algorithm, but this will often give worse results.

3.5.7 Object-oriented code for the learning-rate schedulers

Being one of the most central parts of the project, we have chosen to implement the learning-rate schedulers as an abstract base class in python. The abstract base class is implemented as follows:

```
class Scheduler(abc.ABC):
    def __init__(
        self,
        learning_rate: float,
        timedecay: typing.Callable[[int], float] = lambda _: 1,
        use_momentum=False,
        momentum_alpha=0.9,
    ):
        self._learning_rate = learning_rate
        self._momentum_alpha = momentum_alpha
        self._timedecay = timedecay
        self._update: np.ndarray | None = None
        self._use_momentum = use_momentum
        self.reset()

    @property
    def learning_rate(self):
        return self._learning_rate * self._timedecay(self._current_epoch)

    @abc.abstractmethod
    def _update_no_momentum(self, grad):
        pass

    def update(self, grad) -> np.ndarray:
        self._current_epoch += 1
        if self._use_momentum:
            if self._update is None:
                self._update = np.zeros_like(grad)

            self._update = self._update_no_momentum(grad) + self.momentum()
        else:
            self._update = self._update_no_momentum(grad)

        return self._update

    def momentum(self):
        return self._momentum_alpha * self._update

    def reset(self):
        self._current_epoch = 0
```

This way to add support for more schedulers we can just implement a separate class for the scheduler with the algorithm implemented by overriding *_update_no_momentum*, and overriding other methods as necessary, and it should work well with the rest of our code.

3.6 Automatic differentiation

What is clear is that in our optimization it will be central to calculate the gradient of some loss function with respect to some parameter, however it is not always easy to find analytical gradients by hand. Automatic differentiation is an algorithmic approach to calculating the gradients of functions. We will take some usage of this in our code through the python-library jax [7], but this is a somewhat minor part of the project so I won't go too much into detail about how this algorithm works in detail. Broadly said, automatic differentiation utilizes clever applications of the chain rule in order to calculate the gradients of functions without compromising on accuracy (it usually gives almost exactly the same results as an analytical expression). We will utilize this in some of our optimizations instead of calculating gradients by hand, and for adding support in our code so that users don't have to manually calculate gradients by hand, which can be time-consuming, and a potential for errors in the calculations. The reason we have not used it throughout all the codes is mainly due to the fact that using automatic differentiation often makes our code considerably more computationally expensive, and a lot of our programs are already computationally expensive enough.

3.7 Neural networks

There are many different implementations of neural networks out there. We will in this project focus on one of the simplest ones to implement, namely the multi-layer-perceptron. The multi-layer perceptron consists of a input layer with p nodes (granted we have p covariates), as many hidden layers as we want, each of which can have also as many nodes as we want, and then an output layer of the same dimensionality as our response. For our notation in this section we will assume we have L total layers (excluding the input layer), or equivalently $L-1$ hidden layers. Linking the layers together is a set of weights, where each node in the layer before is connected to each node in the next layer. Additionally each layer has its own weight matrix $\boldsymbol{W}^{(l)}$ and bias vector $\boldsymbol{b}^{(l)}$ for $1 \leq l \leq L$. Each layer also has its own activation function, which we denote $f^{(l)}$, again for $1 \leq l \leq L$. Typical choices of activation functions are sigmoid, relu or leaky relu, which are the ones we will explore in this project. For the final layer we instead call this the output function. The output function is typically tailored to whichever data we have. If we for example want to do regression we often just set this to the identity-function, i.e. we do not transform the activations at all. This is largely due to the fact that we then typically need an unbounded and continuous function when doing regression, and for most cases the identity function does the job well, while also being very simple. In the case where we are doing classification we this choice is no longer a good one as the identity function is not bounded, and we in that case want the output to be a probability between 0 and 1, so then a sigmoid-function for example is a good choice. Predictions are made using forward propagation (this algorithm we describe in 3.7.2), calculating the next layer activations based on the layer activation in the layer before, until we are at the final layer. Here we will follow the notation of *Deep Learning*[6] by denoting $\boldsymbol{a}^{(l)}$ as the activations in layer l before applying the activation functions and $\boldsymbol{h}^{(l)}$ as the final activations in layer l (i.e. after applying the activation function). Keep in mind that the final predictions then will be $\boldsymbol{h}^{(L)}$, while the activations in the inputs (which is just the values of the features) are denoted $\boldsymbol{h}^{(0)}$. For fitting the networks we use the backpropagation algorithm, where we for some given some design matrix $\boldsymbol{X}$ we fit $\hat{\boldsymbol{y}} = \boldsymbol{h}^{(L)}$ to match some target output $\boldsymbol{y}$ for a given cost-function $C(\hat{\boldsymbol{y}}, \boldsymbol{y})$ as good as possible.

3.7.1 Regularization

We will also implement basic support for regularization. This requires a regularization function $\Omega(\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ in this case corresponds to all the parameters. Additionally we add a regularization parameter λ which scales the extent of regularization we apply. Here we will only add l_2 regularization, so $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\boldsymbol{\theta}^T\boldsymbol{\theta}$. How the regularization is used we will go into mainly in 3.7.3, but we will also need the gradient of this regularizer with respect to the weights in each layer. For the l_2 regularization, we see that we have:

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2}\left(\sum_{l=1}^{L}\sum_{i=1}^{n_{i-1}}\sum_{j=1}^{n_i} {\boldsymbol{W}_{ij}^{(l)}}^2 + \sum_{l=1}^{L} {\boldsymbol{b}^{(l)}}^T\boldsymbol{b}\right)$$

. Taking the derivative with respect to $\boldsymbol{W}_{ij}^{(l)}$ we get:

$$\begin{aligned}\frac{\partial\Omega(\boldsymbol{\theta})}{\partial\boldsymbol{W}_{ij}}^{(l)} &= \frac{1}{2}\frac{\partial{\boldsymbol{W}_{ij}^{(l)}}^2}{\partial\boldsymbol{W}_{ij}^{(l)}} \\ &= \boldsymbol{W}_{ij}^{(l)}\end{aligned}$$

Hence we get that $\nabla_{\boldsymbol{W}^{(l)}}\Omega(\boldsymbol{\theta}) = \boldsymbol{W}^{(l)}$.
Keep in mind that we do not include regularization on biases. This is because we usually are ok with having sizeable values for the biases, more so that we are with having big values for weights, which often results in high degree of overfitting. Seeing as there are a lot more weights than biases, more model complexity is also reduced by penalizing the weights than the biases.

3.7.2 Forward propagation algorithm

With the description of the model out of the way, the forward propagation is actually rather simple. We assume we have $L-1$ hidden layers, with index L

being the output layer. We let us inspire by algorithm 6.3 in *Deep Learning* [6, p. 212], with some practical improvements for the code. Our algorithm becomes:

- Set $\boldsymbol{h}^{(0)} = \boldsymbol{X}$
- **for** $l = 1, \ldots, L$ **do**
- $\boldsymbol{a}^{(l)} = \boldsymbol{h}^{(l-1)}\boldsymbol{W}^{(l)} + \boldsymbol{b}^{(l)}$ (here $\boldsymbol{b}^{(l)}$ is added for every row).
- $\boldsymbol{h}^{(l)} = f^{(l)}(\boldsymbol{a}^{(l)})$ (here we apply f element-wise)
- **end for**
- $\hat{\boldsymbol{y}} = \boldsymbol{h}^{(L)}$

Keep in mind that here that $\boldsymbol{X}$, $\boldsymbol{y}$, $\boldsymbol{a}$-s, $\boldsymbol{h}$-s are allowed to be matrices. Here if $\boldsymbol{X}$ is a $n \times p$ matrix, it is easy to see that $\boldsymbol{a}^{(l)}$ and $\boldsymbol{h}^{(l)}$ are going to be $n \times n_l$ matrices (n_l being the amount of nodes in hidden layer l), i.e. we have that each row represents the activations in layer l for each observation. Then as a consequence since $\hat{\boldsymbol{y}} = \boldsymbol{h}^{(L)}$, we have that each row in the prediction matrix corresponds perfectly to the prediction of each row of the design/feature matrix.

3.7.3 Backward propagation algorithm

The backward propagation algorithm is a way for us to calculate the gradients of the weight and biases matrices/vectors. isn't all that much more complicated. Again we take big inspiration from *Deep Learning*, this time algorithm 6.4, with some tweaks. The backwards propagation algorithm then goes as follows:

- Set $\boldsymbol{g} = \nabla_{\hat{\boldsymbol{y}}} C(\hat{\boldsymbol{y}}, \boldsymbol{y})$
- **for** $l = L, \ldots, 1$ **do**
- Update $\boldsymbol{g}$ by setting $\boldsymbol{g} = \boldsymbol{g} \odot {f^{(l)}}'(\boldsymbol{a}^{(l)})$
- Calculate $\nabla_{\boldsymbol{b}^{(l)}} L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_{i=1}^{n} \boldsymbol{g}_{i*}$ (sum over all the rows)
- Calculate $\nabla_{\boldsymbol{W}^{(l)}} L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = {\boldsymbol{h}^{(l)}}^T \boldsymbol{g} + \lambda \nabla_{\boldsymbol{W}^{(l)}} \Omega(\boldsymbol{\theta})$
- **end for**

Then having gathered the gradients of the weights/biases from this we can then use whichever gradient-method we please to update the weights/biases, and continue this way until we have convergence (or the max amount of epochs is reached).

3.7.4 Weight and bias initialization

We see that at the core of neural networks are the weight-matrices and bias vectors, and seeing as we want to use an iterative method (namely the Newton-Raphson method) it becomes clear that we need to initialize the weights and biases to something. If we were to initialize all the weights/biases to be 0 (or some other constant) all of the neurons in the hidden layers will all have the same activations in the first iteration, and it becomes impossible for the model to learn properly. Our solution to this is to draw the weights randomly, for example from the standard-normal distribution as we have done in this project. This way we will be able to tackle the problem of the learning-process not starting properly. Another thing to take into consideration is that we ususally want the weights to be quite small initially, as very big weights can lead to overfitted models. For biases we often don't always bother initializing these randomly (although there is nothing hindering us to do so). What we will do is to initialize all the biases to the same value in every layer (we will set all of them to 0.1). Setting it to a small positive value will ensure that all the neurons are activated in the first training-cycle, which will also ensure all the neurons have some output which will be backpropagated in the first training-cycle [8, s. Weights and biases].

3.7.5 The universal approximation theorem

The universal approximation theorem tells us that if we have a neural network with one hidden layer and a non-linear activation-function in the hidden layer, if we have enough nodes in the hidden layer we should be able to approximate any continous multi-dimensional function to any accuracy [3, s. 13.5]. Keep in mind that this is merely to the training-data. A neural network can be able to approximate the training-data very well, without necessarily getting that great performance for new data.

3.8 Activation functions

For this project we will explore three different activation functions for the hidden layers.

3.8.1 Sigmoid

The sigmoid activation function limits the activations of the neurons between 0 and 1. The sigmoid activation function is given as the following:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

With the derivative being (this is quite straightforward to calculate):

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Additionally as mentioned this will also be used as an output-function when we are doing binary classification, due to the fact that the final neuron activations will be between 0 and 1.

3.8.2 Relu

Another popular activation function we will use is the relu activation function. Unlike the sigmoid function this does not have a upper bound. The function then can be expressed by:

$$\text{relu}(x) = \max(x, 0)$$

The derivative then becomes:

$$\text{relu}'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$$

Keep in mind that the derivative is not defined for $x = 0$. This however is usually unproblematic as we almost never get activations being exactly 0. In our implementation we will just set the derivative to 0 if the activation is exactly 0 (but again this should more or less never happen).

3.8.3 Leaky relu

The leaky relu is a generalization of the relu-function which unlike the relu-function doesn't completely neutralize negative values. Technically this makes this function not bounded below, however with δ being rather small severely limits the size of the negative activations, while no such shrinkage is applied for the positive activations of course. The leaky relu is given by:

$$\text{lrelu}(x) = \begin{cases} x & \text{if } x > 0 \\ \delta x & \text{if } x \leq 0 \end{cases}$$

The derivative then becomes:

$$\text{lrelu}'(x) = \begin{cases} \delta & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$$

We see one major difference between this and the relu is that for negative activations, the derivative of this will not be 0, but δ. This means that the network will be able to learn on activations which are negative as well, albeit likely slower than the positive cases. For this project we will be using $\delta = 10^{-3}$.

3.9 Cost functions

In this project we will use mainly two activation functions, namely mse/sse for regression and cross-entropy for binary-classification. We will now quickly go into these.

3.9.1 MSE/SSE

A natural choice of cost-function when dealing with regression tasks is the MSE. When the target dimensions are always 1 (which they are in this project), the MSE is defined like:

$$\text{mse}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i)^2 = \frac{1}{n}(\hat{\boldsymbol{y}} - \boldsymbol{y})^T(\hat{\boldsymbol{y}} - \boldsymbol{y})$$

The derivative with respect to $\hat{\boldsymbol{y}}$ then becomes:

$$\frac{\partial \text{mse}}{\partial \hat{\boldsymbol{y}}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{2}{n}(\hat{\boldsymbol{y}} - \boldsymbol{y})$$

We will also use the total sum of squares (SSE). Minimizing this is the same as minimizing the MSE, but with one less floating-point operation. The SSE is given by:

$$\text{sse}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_{i=1}^{n}(\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i)^2 = (\hat{\boldsymbol{y}} - \boldsymbol{y})^T(\hat{\boldsymbol{y}} - \boldsymbol{y})$$

and its derivative then becomes:

$$\frac{\partial \text{sse}}{\partial \hat{\boldsymbol{y}}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = 2(\hat{\boldsymbol{y}} - \boldsymbol{y})$$

The only potential problem with using SSE instead of MSE is that the gradient will then be dependent on the amount of observations we are using, which can be problematic with stochastic gradient descent. However in this project we will set a fixed size of each minibatch, which guarantees that each training-set sent into this cost/loss function will be of the same size, so this becomes unproblematic.

3.9.2 Cross-entropy

For binary classification we will use cross-entropy as out cost-function. Minimizing the cross-entropy is equivalent to maximizing the likelihood. We have the cross-validation given by:

$$\text{cross-entropy}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{i=1}^{n}(\boldsymbol{y}_i \log(\hat{\boldsymbol{y}}_i) + (1 - \boldsymbol{y})\log(1 - \hat{\boldsymbol{y}}_i)).$$

With the gradient w.r.t. $\hat{\boldsymbol{y}}$ then becoming:

$$\frac{\partial \text{cross-entropy}}{\partial \hat{\boldsymbol{y}}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = (\hat{\boldsymbol{y}} - \boldsymbol{y})/((1 - \hat{\boldsymbol{y}}) \odot \hat{\boldsymbol{y}}).$$

Here the division is element-wise. One thing to note is that we see that if one of our predictions $\hat{\boldsymbol{y}}_i$ is very close to 0 while the target value is 1 or vice versa, which especially can happen in the first iterations when using some sort of unbounded activation function in the last hidden layer, the gradient can become very big. If we get too big of a gradient this can be problematic for many reasons, with one being we can get values too big for the computer to calculate. In our code we will solve this by clipping the predictions in the cross-entropy-gradient, so that they don't become too close to 0 or 1 so that this doesn't become a problem. Additionally we will add a small δ inside the log-expression in the cross-entropy function for the same reason.

3.10 Accuracy score

For classification methods, after we have fitted the models and made predictions, we often want to evaluate a metric of how good our model really is. One intuitive such metric we will use a lot is the accuracy score. The accuracy-score essentially is just the proportions of datapoints we have classified correctly on some data. The accuracy score can then be defined as follows:

$$\frac{1}{n}\sum_{i=1}^{n} I(\hat{\boldsymbol{y}}_i = \boldsymbol{y}_i).$$

Using numpy this can simply be written as:

```
np.mean(y_pred == y)
```

, for some predictions *y_pred* and some target *y*. Keep in mind that while this can tell us the overall successes/failures of classifications it does not tell us what kinds of prediction errors we have made. For example in a binary-classification problem we may be interested in finding the amount of predictions we have made to be 1 while the target was 0, or vice versa, as one of these predictions errors may be more severe than the other. For these cases we have a confusion-matrix plotting the predictions against the target in a heatmap. We will use some such plots in this project as well.

3.11 Model complexities

One final thing to consider which will be very important for our results is the model-complexity of neural networks and logistic regression. Logistic regression requires just as many parameters as we have explanatory variables, so the model complexity is rather low, at least in the case where we don't have too many covariates. Neural networks however contain a lot of parameters so naturally the model is also much more complex. In both cases however we can introduce regularization constraints, so we are able to control the model complexities in both cases.

3.12 Logistic regression

The logistic regression model is a commonly used, and rather simple, way to perform classification tasks. With logistic regression we assume a binary response

0 and 1 typically and model the probability the follwing way:

$$Pr(Y_i = 1|\boldsymbol{x}_i) = \frac{1}{1+\exp(\boldsymbol{x}_i\boldsymbol{\beta})} = \sigma(\boldsymbol{x}_i\boldsymbol{\beta})$$

, where σ is obviously the sigmoid-function as already described. We then can do classification based on this, where we typically classify to 1 if this probability is bigger than 0.5 (this is the boundary we will use here) and 0 if lower than 0.5. In order to estimate the $\boldsymbol{\beta}$ parameters we use the likelihood for such a model, given by [5, s. 4.4]:

$$l(\boldsymbol{\beta}) = \sum_{i=1}^{n}(y_i\boldsymbol{x}_i\boldsymbol{\beta} - \log(1+\exp(\boldsymbol{x}_i\boldsymbol{\beta})))$$

, where the estimates $\hat{\boldsymbol{\beta}}$ become the ones which maximize this likelihood. The cost-function then becomes:

$$C(\boldsymbol{\beta}) = -\sum_{i=1}^{n}(y_i\boldsymbol{x}_i\boldsymbol{\beta} - \log(1+\exp(\boldsymbol{x}_i\boldsymbol{\beta}))).$$

To minimize we need the gradient given by [5, s. 4.4]:

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\sum_{i=1}^{n}\boldsymbol{x}_i^T(y_i - Pr(Y_i = 1|\boldsymbol{x}_i)) = -\boldsymbol{X}^T(\boldsymbol{y}-\boldsymbol{p}).$$

We then can use the Newton-Raphson method on this in order to estimate the $\boldsymbol{\beta}$.

We may also want to add regularization. To do this we can simply add a term $\frac{1}{2}\lambda\boldsymbol{\beta}^T\boldsymbol{\beta}$ to the cost-function, and add a term $\lambda\boldsymbol{\beta}$ to its gradient, just as we did with neural networks.

4 Results

4.1 General optimization

The **optimization_basic.py** in the project repo [9] performs various experimentation on some generated data as described in 3.1, and explores the effects the various parameters have when using gradient methods. For exploring the effect the amount of epochs has, we did a quick test by four-doubling the amount of epochs from one model to another. Table 1 shows the results of optimizing using GD and SGD with different amount of epochs. What we clearly see is that in general, the more amount of epochs we allow the smaller the cost will be (granted we have convergence for the specified learning-rate). This is intuitive as for each iteration we should get closer to the minimum of the cost (for the most part anyway), so doing more iterations must also naturally bring us closer to the minimum.

Epochs	100	400
COST GD	4.51	1.56
COST SGD	0.13	0.04

Table 1: The cost (mean-squared-error) of the model for plain gradient descent and stochastic gradient descent, trained with 100 epochs and 400 epochs. In both cases the model trained on more epochs is substantially smaller.

Minibatch-size	8	12	16	32
COST SGD (best)	0.0421	0.0612	0.1315	0.7856

Table 2: The cost (mean-squared-error) of the model for stochastic gradient descent trained on a grad of learning-rates for each minibatch-size with the learning-rate giving the lowest cost is chosen. We see that in this case the lower minibatch-sizes seem give faster learning. Here we have trained using 100 epochs.

For stochastic gradient descent we may also be interested in seeing if the size of the minibatches has an effect on the speed we are able to learn. Table 2 shows the costs of four different SGD models trained using equally many epochs, but with four different minibatch-sizes. Here we see that the lower minibatch-sizes seem to give a little quicker learning, but that this effect is not that enormous (perhaps with the exception of the difference between the minibatch-sizes of 8 and 32, where there is quite a difference).

We also explored how the learning-rate influences the performance of GD, SGD, GD with momentum and SGD with momentum, by training various such models on a grid of learning-rate. The costs of these functions plotted against the learning-rates at which the cost was obtained is shown in figure 1. We see generally that the bigger the learning-rate is the better the model is, however we also see that if the learning-rate becomes too big that the cost diverges towards infinity. We also see that in all the cases the learning-rate is very important, with the difference between the cost of a good and bad learning-rate being very big, even more so for the cases with added momentum.

Just from this plot we see quite a big difference between how quick the different optimization-methods work, with the momentum versions being much quicker to minimize the costs. Table 3 shows the costs for the different optimization methods. We see that in all cases adding momentum increases the speed of learning by quite a bit (perhaps less so with AdaGrad trained on stochastic minibatches, where this increase is quite marginal). We see that which method we use does impact the speed of learning by quite a lot, with the lowest cost (stochastic AdaGrad with momentum) being more than 100 times smaller than the highest cost (GD without momentum). Overall we can see that AdaGrad did seem to perform the best out of the methods we tried out.

We also check out the performance of each method under a different cost-function, namely using a ridge cost-function. Previously we have explored how

Method name	GD	SGD	AdaGrad (non-sgd)	AdaGrad (sgd)
COST (no momentum)	4.5150	4.1092	1.4655	0.0418
COST (with momentum)	0.0384	0.0488	0.0400	0.0385

Table 3: Various different learning-rate scheduling methods, with the methods being trained for a grid of learning-rates η, with the best cost being selected. Here for the stochastic gradient descent, we have used a timedecay for the learning-rate (plain gradient descent only). AdaGrad (sgd) here refers to using AdaGrad and training on stochastic mini-batches.

the performance depends on the hyperparameters isolated. In ridge regression we introduce another hyperparameter, namely the regularization parameter λ. Since we are dealing with ridge regression here it makes sense to explore the mse on some test-data, which is what we have done in the program. This now doesn't only depend on the learning-rate η anymore as previously, but also the regularization parameter λ (we keep the rest of the parameters fixed). Figure 2 shows a heatmap which tells us how the test MSE depends on both the η and λ for GD and SGD. What we see is that the best performance is quite sensetive to both the η and λ, and that a wrong η or λ alone can very much ruin the performance. Similarly figure 3 shows a corresponding plot for AdaGrad. Here we see that this is much more robust to higher learning-rates than the other methods, where it seems to take much more for this to diverge. This is probably related to the fact that AdaGrad vastly downscales the learning in directions with high historical updates (see 3.5.4) so this is able to combat divergence for the cost for the higher learning-rates.

For both the figure 2 and 3, focusing on the regularization parameter λ we see much of the same as we did with normal ridge, where a too low λ did not give that good results because of too little regularization, while a too high λ also caused subpar results due to too much regularization. However, we also see that we now have another dimension on top determining the performance, namely that the learning-rate must also be optimal for the model to be any good. Picking too low of a learning-rate will as seen already cause the cost to not converge, while picking a too big learning-rate will cause the cost to diverge. Picking a good model clearly requires us to do some kind of hyperparameter-tuning (like a grid-search for example) on both η and λ, as some η-values will cause a diverging cost for some, but not all, values of λ (as can be seen for $\eta = 1$ for the SGD case in figure 2).

4.2 The franke-function

Experimentation of using a neural network with the franke-function is done in the **franke_function.py** file in the project codes [9]. We here tried to fit an neural network as good as possible to minimize the MSE. Doing a grid-search on both the learning-rate η and the regularization parameter λ for each method using only stochastic gradient descent in this case. Figure 4 shows a

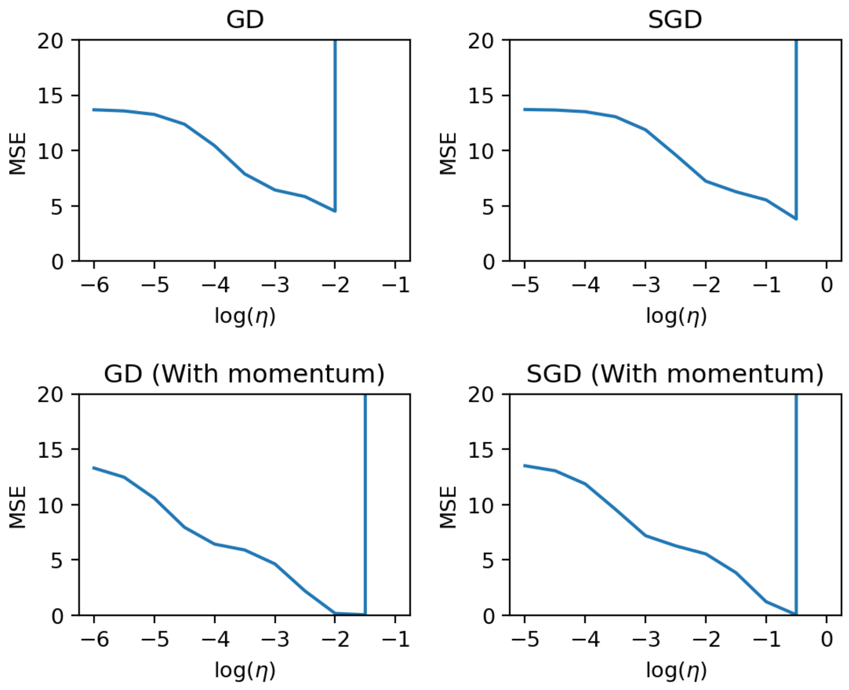

Figure 1: The costs trained on specific learning-rates. We see a general downward-moving trend, until we get a learning-rate higher than $\sqrt{10}$ for the SGD cases, and 10^{-2}, and $10^{-\frac{3}{2}}$ for the GD cases without and with momentum respectively, in which case after this we get a diverging cost.

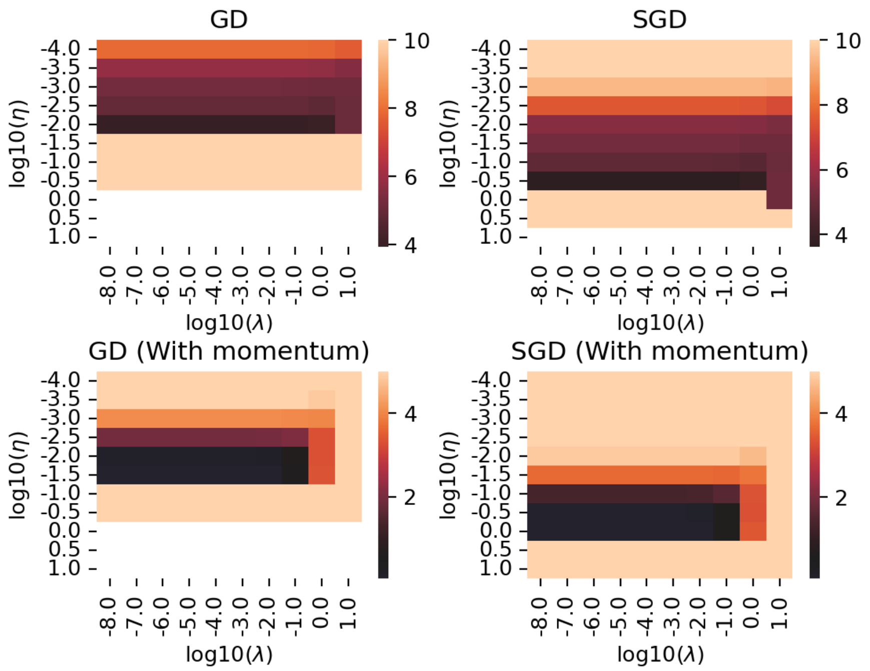

Figure 2: A heatmap of the test MSEs for various learning-rates and regularization paramters, using GD/SGD with and without momentum. Here we see that the lowest test MSEs for all the methods are somewhere in the middle of each heatmap. We also see that too low or high of a learning-rate η gives very poor or diverging costs, while too high or low regularization parameter λ also gives worse performance.

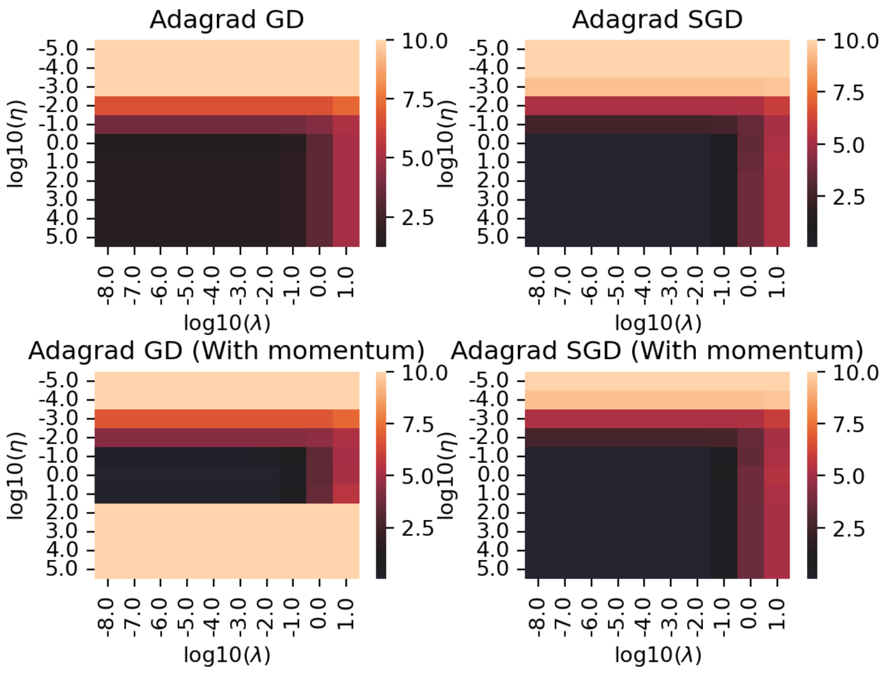

Figure 3: A similar heatmap to 2. We see much of the same patterns here, with one difference being that apart from the non-stochastic model without momentum, we are able to have much higher learning-rates without the costs diverging.

	Train	Test
MSE (best)	0.0068	0.0072
R^2 (best)	0.9244	0.9266

Table 4: The final metrics achieved for a neural network using SGD with momentum as optimization method. Here the learning-rate giving best train/test MSE/R^2 score were ≈ 0.00056, while best regularization-parameter was 0.0001 for the training-data and 0.001 for the test-data.

Library	sklearn	tensorflow
MSE (train)	0.0984	0.0726
MSE (test)	0.1091	0.0800

Table 5: The MSEs when using SGD neural networks in sklearn/tensorflow, and using the same optimal parameters we got for our own neural net (though we had to use a different regularization for tensorflow). We see we got a little better performance using tensorflow, but still much worse than our own implementation (see table 4).

heatmap of the MSE for some different values of both. Again here we see much of the same effect as we did in figure 2 and 3, with the best performance being dependent on having a fitting learning-rate and regularization parameter, but we can also see that we have quite a lot of learning-rates which get get quite good results for both train and test seemingly. Table 4 shows us some metrics we got when training neural networks on the franke-function, and choosing the best-performing model. In this case the best model (for the test-data) had $\eta \approx 0.00056$ and $\lambda = 0.001$. We get quite good results here, and we also see that the MSE and R^2 scores for the train and test-data are quite similar, which does indicate that the regularization works quite well (though one of course need to take into account that they were achieved at different values of λ). If we compare this to what we got for the linear methods in project 1 we had that for linear regression, ridge and lasso we had test MSEs of 0.0189, 0.0173 and 0.0165 respectively, with the corresponding R^2 scores being 0.8050, 0.8210 and 0.8303 [10, s. 4.1]. We clearly see that we have indeed achieved better performance in this case than the simple linear models we tackled in project 1.

We also compared our own results to those of Scikit-learn and tensorflow. Figure 5 shows the MSE achieved using these two methods as well (here I have dropped the R^2 scores, but they will be closely related to the MSEs). Interestingly we got quite a bit better results for our own code than these two libraries, however we have spent less time tweaking these models and their hyperparameters, so this is probably the reason for the worse performance here.

Lastly we also explored using different activation functions for the hidden layers. We here just used the same activation functions for all the hidden layers, with us testing out sigmoid, relu and leaky relu. Table 6 shows the MSE for the different hidden layers. From this we can see that the sigmoid gave us the

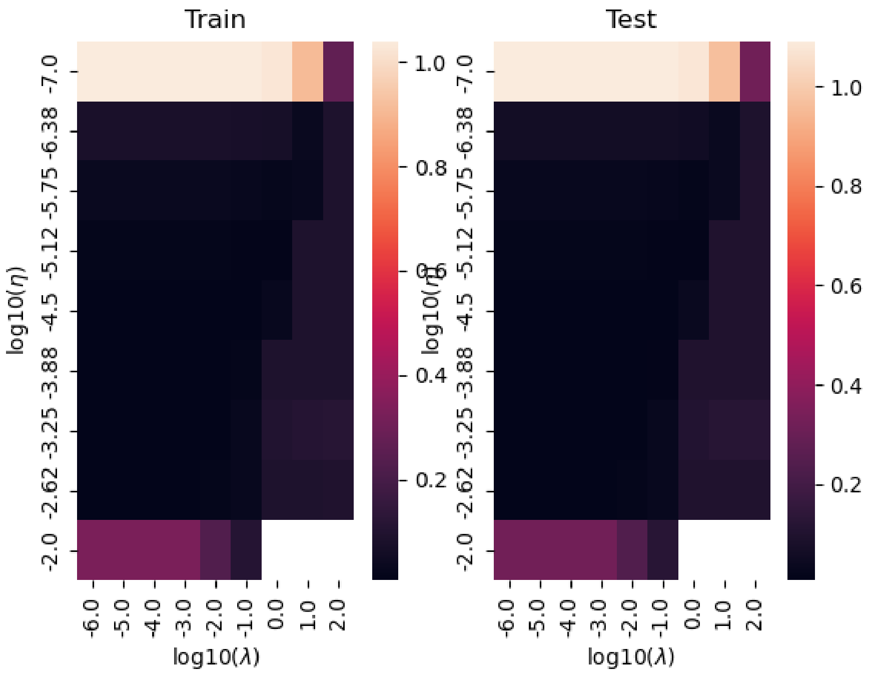

Figure 4: A heatmap showing the cost (MSE) for both train and test-data, depending the the learning-rates η and regularization parameters λ.

Activation function	sigmoid	relu	lrelu
MSE (test)	0.0086	0.0178	0.0164

Table 6: The test MSEs for three different activation functions. We see that sigmoid in particular gave about twice as low MSE as the two others, while relu and lrelu gave comparable results.

Hideen layer sizes	(100)	(10, 10)	(100, 100)	(50, 10)	(10, 10, 10)
Accuracy	1.0	1.0	1.0	1.0	
End of training loss	0.001246	0.017191	0.001211	0.007496	0.014259

Table 7: Accuracy and end-of-training loss metrics for neural networks with different hidden layer sizes, trained using max 200 epochs and using the sigmoid activation function. Here we have tested on a grid of learning-rates and picked the best accuracy/end-of-training loss. We see that all of the models are able to classify 100% of the training-data correctly, but we see that the loss (cross-entropy) is a little lower for some models than others. A lower end-of-training loss indicates that the model learns quicker.

best performance by a little bit, so it was correct using this in our experiments above.

4.3 Wisconsin Cancer data

4.3.1 Neural network

The **neuralnet_breastcancer.py** in the project code performs analysis for fitting neural networks on the wisconsin breast cancer data, in a classification task. Here we again explored various hyperparameters of the neural network to see which hyperparameters yielded the best result. Table 7 shows us the accuracy and end-of-training loss for some different hidden layer sizes trained on 200 epochs. Here we see that all of the different layer sizes we tested were able to classify all of the training-data correctly, so it doesn't seem to matter all that much which of them we choose, however we do see that a model with two hidden layers, each of 100 nodes did give the lowest end-of-training loss so this is the layer sizes we will be choosing for the rest of our experimentation.

It is also natural to explore which activation function we shall use for the hidden layers. Table 8 shows the accuracy and end-of-training loss using the three activation functions we explore in this project. We see that all the methods perform quite well on the training-data (when looking at the accuracy), but that the sigmoid gives a much lower end-of-training loss, so we have much quicker learning in this case. Therefore we choose to use the sigmoid forwards.

With the hidden layer sizes and the activation function in check we just need to choose the learning-rate scheduler, along with the learning-rates and regularization parameters for each of these schedulers. Table 9 shows the performance of different learning-rate schedulers in terms of prediction accuracy on

Activation function	sigmoid	relu	leaky relu
Accuracy	1.0	0.9774	0.9724
End of training loss	0.0012	207.2327	253.2846

Table 8: The accuracy and end-of-training loss (both on the training-data) for a neural network with 2 hidden layers, each with 100 nodes and training on 200 epochs. We see that overall the sigmoid does give the best results. Keep in mind that we have allowed for more epochs in the final model (2000 vs 500 epochs)

Method	SGD	AdaGrad	Adam	RMSProp
Train-accuracy	1.0	1.0	1.0	1.0
Valdiation-accuracy	0.9510	0.9608	0.9510	0.9510

Table 9: Training and validation accuracy. We see that AdaGrad gave slightly better results (one more correct prediction) than the other on the validation-data, while for the training-data all the schedulers are perfect.

both the training-data and validation data (both for the best respective η and λ). Here we see that all of the models managed to give 100% accuracy on the training-data, but none of them managed it on the validation data. We see that AdaGrad gave us the best results in this case, with the best learning-rate being 10^{-1} and the best regularization parameter being 10^{-7}. This will then be our final chosen model.

Table 10 shows the final accuracy of our selected model evaluated on a separate test-set in order to not get too optimistic final metrics (we have trained a lot of models). We here have fitted on both the previous training and validation data. What we see is that our own neural network was able to classify 100% of the observations correctly, while the tensorflow model was able to classify 98.55% of the observations correctly. Both of these are very good results. Figure 5 and 6 shows the confusion matrices for these two models. We unsurprisingly see that the confusion matrix for our own neural network is perfect, while the tensorflow version did make one wrong prediction, that being one false negative prediction.

Model	Project NN AdaGrad	Tensorflow AdaGrad
Test-accuracy	1.0	0.9855

Table 10: The final accuracy on the test-data for our own custom neural network built from scratch, and an equivalent model fitted in tensorflow. Both these models use the optimal η and λ found previously. We see that we get perfect, while almost perfect performance for the tensorflow model.

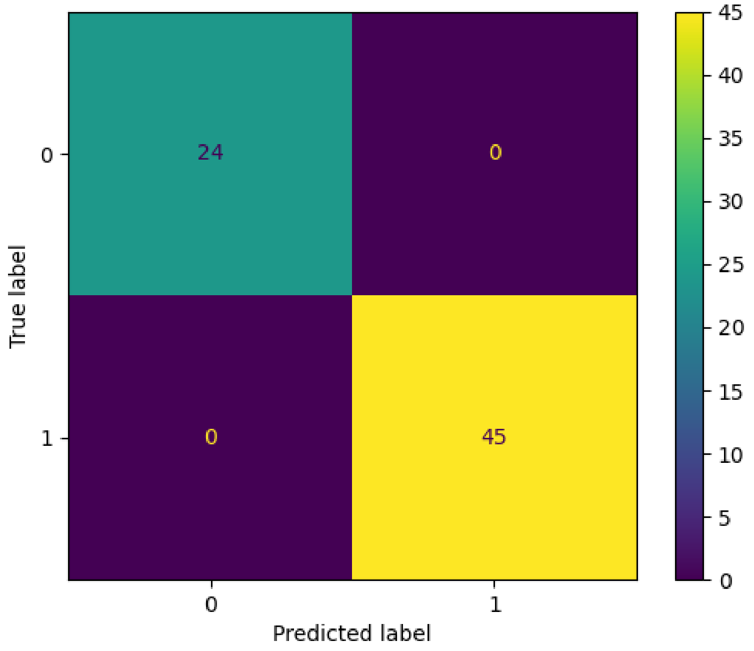

Figure 5: Confusion matrix for our own final AdaGrad model. We see that all the predictions are correctly classified.

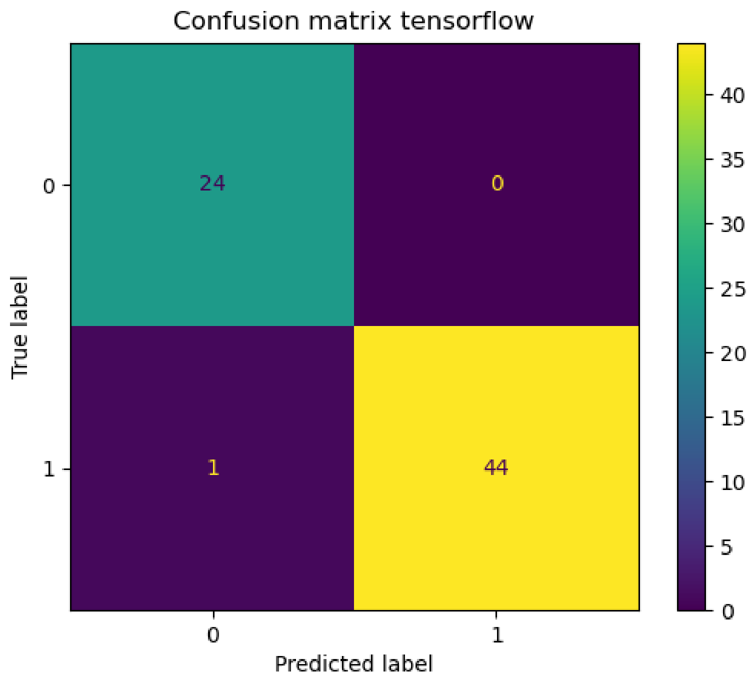

Figure 6: Confusion matrix for the final AdaGrad model using tensorflow.

Model	Project Logistic	Sklearn
Test-accuracy	0.9710	0.9710

Table 11: The results of using logistic regression (Sklearn and own implementation) on the cancer data. Here we have used $\lambda = 1.0$, which we found out gave the best results on some validation data. Furthermore for both models we allowed up to 2000 epochs and for the project model we used adam as the learning-rate scheduler with $\eta = 0.05$. We see that we get the same accuracy for our own model and the scikit-learn model.

4.3.2 Logistic regression

Finally we explored using logistic regression on this same data. Table 11 shows the accuracy we got on the validation data. The results are very good with there being very few wrong predictions. We also see that the estimated parameters for the Scikit-learn model and our own model gave pretty much the same parameters. Lastly figure 7 shows the confusion matrix for our own logistic regression implementation. We here see that we have one false positive and one false negative. The confusion matrix for the Scikit-learn model is exactly the same.

5 Analysis

While exploring the hyperparameters in a neural network model, we saw that perhaps the most central part in determining how good our models were was the learning-rate and regularization parameter, along with the learning-rate scheduling method. These findings agrees with the general optimization findings. Less important, but still noticeable were the minibatch-size, the size and number of the hidden layers and the activation functions. The hidden layer sizes not being that important is of course granted we are selecting between reasonably big neural networks - we would not expect a very small network to fit well to very complex data. We also clearly saw that, in general, the more epochs we train on the better the cost will generally be (at least on the training-data). We also saw that using *jax* instead of hand-computed gradients did not impact our results.

Another thing we saw was that in both the neural network for the franke-function, and for the breastcancer classification, the sigmoid activation-function gave us the best results, though in both cases relu and leaky relu did perform not that much worse. One thing to note is that the sigmoid activation function is more computationally heavy to use, as it involves taking an exponential, whereas relu and leaky relu just involve a conditional expression, each of which only contain a simple multiplication. This means that in general training models with the sigmoid will be more computationally expensive than the relu/lrelu versions. This way one would have to weight up if we then could train more epochs in the same time. Here we have not done this as we have just kept the

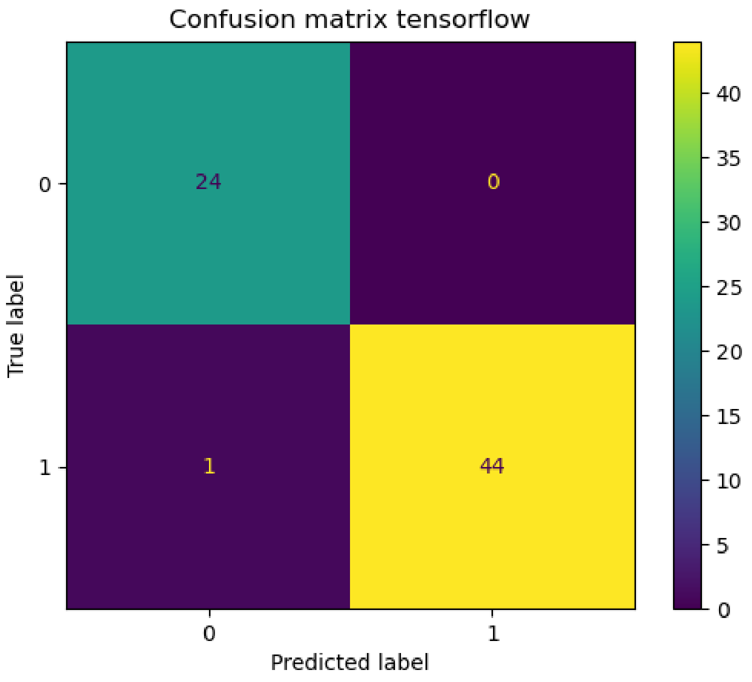

Figure 7: Confusion matrix for the final logistic regression model (using our own implementation).

amount of epochs the same for all these activation functions. If we then could train more epochs in the same time on a relu/lrelu this could very well become a better model.

We also see that the neural network we built did manage to outperform all the linear models which we explored in project 1 (with the same amount of observations in the train/test-data). This is likely due to the fact that in project 1 we mainly concentrated on fifth order polynomial features for our model, however seeing as our actual data is not generated from a polynomial, but the franke-function, the linear models can only do as well as a fifth-order polynomial (with interactions) can approximate the franke-function. Neural networks are not limited in the same way due to the universal approximating theorem. Here we can just give the network enough nodes and the network should be able to fit the training-data arbitrarily well, unlike the linear models. Here it seems like this property was enough for the neural network to come out on top, even for quite little data.

For both experiments, one possible improvement would be to test some different regularizers, like for example seeing if we get improvements in any of the experiments by using l_1 regularization instead of l_2 regularization. Another thing that could be interesting is to check out to which extent the amount of data here influences the performance of the neural network, for example for the franke-function. One would expect that as we increase the amount of data, that the neural network will eventually perform even better.

Another possible improvement is that we have only tweaked some of the hyperparameters of a neural network. Perhaps the most natural parameter to tweak which we have left out is the momentum parameter, which we just kept fixed at 0.9, but this is a parameter which can have some influence over the learning of the model. The different time-decays we have used has also been somewhat limited, as we have only used it with stochastic gradient descent, and even in this case we used only one quite basic time-decay. In general we have also used just simple grid-search for hyperparameter-tuning, while one could use more advanced techniques/libraries.

Lastly a potential problem I have already highlighted is how we have quite few observations in the cancer data case. This meant our validation data in particular became smaller than what is ideal. One could perhaps try a bigger split. An even better approach would be to simply do a train-test split and then use cross-validation for selecting the hyperparameters and then evaluate the final model on the test-set. The reason why I opted to not take this approach was mainly that cross-validation is much more computationally heavy, so running the programs would take much longer, and they already take quite a bit of time to run. I also took the same approach for logistic regression, as I wanted the conditions to be as equal as possible, but especially in the logistic regression case, cross-validation wouldn't even be particularly computationally expensive.

6 Conclusion

We see that when using gradient methods we have a lot of hyperparamters to tune, with some clearly being more important than others for how good of a model we are able to train. We have seen that especially important for models using gradient methods, neural networks included, is the learning-rate η, together with the learning-rate scheduling method, as well as the potential regularization parameter λ. Of course a high enough amount of epochs is also very important. Less important, but still noticeable is the minibatch-size (when using some sort of stochastic gradient descent), the type of activation function and the choice of hidden layer sizes.

Training on the franke-function and wisconsin cancer data we had that the neural network did perform better in both of these cases on the final test data, therefore manifesting how powerful these methods really can be. This is perhaps especially impressive since neither of these datasets were particularly data-heavy, and often complex models such as neural networks can struggle a bit with small amounts of data. In the case of the franke-function the performance was so much better that it seems like analysis on this particular function, and perhaps also terrains or other terrain-like functions, is better suited for something like a neural network than the linear models we looked at in project 1. The best model we obtained for predictions on new data for the franke-function was a neural network with two hidden layers each of 100 nodes, using SGD with momentum, and with $\eta \approx 0.00056$ and $\lambda = 0.001$. The corresponding test MSE for this model ended up at ≈ 0.0072.

In the logistic regression case, the logistic regression was able to closely compete with the neural networks in terms of prediction accuracy, with neural networks only just about coming out on top (also keep in mind that our test-data was quite limited, so the significance of how much better, if any better at all, is not clearly manifested). For this reason logistic regression (with some regularization) could still be a perfectly good model for this data, especially since logistic regression models are much easier and quicker to train, as well as the fact that they are better for inference, which is often relevant in the medical field. Incredibly though our neural network was able to classify all of the test-data correctly, so it was basically impossible to compete with this. The model which we were able to achieve this for was using AdaGrad (with stochastic minibatches), with $\eta = 0.1$ and $\lambda = 10^{-7}$, on a neural network with again two hidden layers, each with 100 nodes.

7 Appendix

7.1 Project code (github)

All the python programs described in the report and with the full source code can be found at: `https://github.com/magnouvean/ml-physics-projects/tree/main/project2/python`

References

[1] Project 1. `https://github.com/magnouvean/ml-physics-projects/tree/main/project1`, 2023.

[2] Breast cancer wisconsin (original). `https://archive.ics.uci.edu/dataset/15/breast+cancer+wisconsin+original`, 2023.

[3] 13. neural networks. `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter9.html`, 2023.

[4] sklearn.datasets.load_breast_cancer. `https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html`, 2023.

[5] Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[7] google/jax - github. `https://github.com/google/jax`, 2023.

[8] Week 41 neural networks and constructing a neural network code. `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week41.html`, 2023.

[9] Project 2 - python. `https://github.com/magnouvean/ml-physics-projects/tree/main/project2/python`, 2023.

[10] Project 1 - report. `https://github.com/magnouvean/ml-physics-projects/tree/main/project1/report/report_project1.pdf`, 2023.