Exploring the performance of neural networks relative to classical models

Bjørnstad, Johannessen, and Merlid

Department of Physics; UiO

(Dated: November 4, 2024)

Neural networks is the primary type of models used in complex machine learning problems, as they are extremely agile and can perform well in a wide range of scenarios, including both regression and classification tasks. They do however require large amounts of data to be trained and it is hard to interpret how they make their predictions. Classic models as linear and logistic regression are easier to interpret and simpler to fit to the data. In this paper we compare the performance of linear regression, with both ordinary least squares and ridge, to neural networks with a range of architectures and activation functions, on the noisy Franke data set (Franke, 1979), where we reuse our previous implementations of linear regression models (Bjørnstad *et al.*, 2024). Furthermore we implement logistic regression, and compare its performance with neural networks on a classification task, based on the Wisconsin breast cancer data set (Wolberg *et al.*, 1993). We find that our best neural networks perform better than linear regression at predicting the Franke data, however we emphasize the fact that neural networks are computationally heavier and less interpretable than the linear regression models. Our best neural network model achieves an R^2 value of 0.83, far outperforming the best linear regression model at an R^2 of 0.37. We find that the accuracy of the neural network trained to solve a classification problem is 0.97, equal to logistic regression. We prefer the logistic regression model as it has better recall.

Contents

I. Introduction

As our society has grown gradually more data-driven in the last decades, the number of important systems and decisions which are driven by data models is only increasing. Neural networks, being large adaptive machine learning models able to use large amounts of data to make accurate predictions, have rapidly moved from being a theoretical niche to being at the front of everybody's minds. For a wide range of fields, including healthcare, finance, education and transportation - neural networks are currently the cause of revolutionary changes.

Even though they have only shot to fame in more recent years, neural networks are not a new concept. The idea of making computational models inspired by neurons in the brain originated in 1943, in a paper by neurophysiologist Warren McCulloch and mathematician Walter Pitts (Macukow, 2016, p. 4). However, at the time neither the computational power or the algorithms required to train such networks were available. An important step was made when the backpropagation algorithm was popularized in 1986 (Macukow, 2016, p. 5), but it was not before the last decade neural networks exploded in popularity.

Neural networks have gradually replaced linear and logistic regression models, and are now being being used to solve an increasing range of novel problems. However, it is not a given that they will always perform better than classic, simpler models. Neural networks also require significantly more computational power, and often more data to be trained. In this paper we will apply neural networks alongside linear regression for a regression problem, and logistic regression for a classification problem to compare and analyze the performance.

For the regression problem we consider the Franke function (Franke, 1979). We implement a framework for testing different network architectures, activation functions and gradient descent algorithms to find the best possible neural network model for the problem. We then compare its overall mean squared error on a test set with the results we got for linear regression models in our previous implementations (Bjørnstad *et al.*, 2024).

In the classification case we consider the Wisconsin

breast cancer data (Wolberg *et al.*, 1993), containing data on breast cancer patients, as well as whether or not they had the disease. We implement logistic regression, and compare it to our best performing neural network model.

We start out by covering the theoretical background on which our work is based. After that we describe our implementations of the different methods, how they were tested and give reasons for decisions made during the process. The results will follow, including an interpretation and a critical discussion on which type of models perform best, based on different measures. Based on this we conclude on the quality of the neural network models on these problems, compared to linear and logistic regression.

II. Theory

A. Linear Regression

This subsection is reused from our previous work (Bjørnstad et al.*, 2024, p. 2).*

Linear regression bases itself on the assumption of a linear relationship between the *predictors* and the *response* (Fahrmeir *et al.*, 2013, p.21-26). Given a data set of p input variables (commonly called predictors), $X = [x_1, x_2, \ldots, x_p]$ and a data set of output variables (commonly called the response) one seeks a linear model on the form

$$\tilde{y} = \hat{\beta}_0 + \sum_{j=1}^{p} x_j \hat{\beta}_j. \tag{1}$$

The scalar $\tilde{y}$ is the prediction of the response, $\hat{\beta}_0$ the estimated intercept, and each $\hat{\beta}_j$ the estimated coefficient belonging to its corresponding predictor x_j.

This equation is commonly written in vector form,

$$\tilde{\mathbf{y}} = \boldsymbol{X}^T \hat{\boldsymbol{\beta}}, \tag{2}$$

where given n different sets of input/output-variables (data points), $\tilde{\mathbf{y}}$ is the response vector and $\boldsymbol{X}$ is a $n \times (p+1)$ matrix called the *design matrix*. Here the $'+1'$ column in $\boldsymbol{X}$ is a row of ones for inclusion of the intercept in $\hat{\boldsymbol{\beta}}$, and the residual p columns each of the p predictor variables.

The "true" model is assumed the form

$$y = \beta_0 + \sum_{j=1}^{p} x_j \beta_j + \epsilon. \tag{3}$$

Our linear regression models are always estimations of Eq. 3 above, and for real-life data there's no way of knowing the true $\boldsymbol{\beta}$ (for generated data there will be exceptions). ϵ is an irreducible *error term* or *residual term* representing all variance in the data not explainable by the linear model; any variation due to randomness and noice is included in this term. It is assumed $\epsilon \sim \mathcal{N}(0, \sigma^2)$.

The main objective when solving linear regression problems, is finding the optimal coefficients $\boldsymbol{\beta}$ that minimizes an error measure between y_i and $\tilde{y}_i$. How such an optimal solution evinces is dictated by the definition of the *cost function*, or *loss function*, which is simply metrics chosen to measure how much the predictions deviate from the "truth". A cost function, $\text{Cost}(f, \mathcal{D})$, is used to describe such a metric measuring a group of data points, while a loss function, $L(y, \hat{y})$, describes a metric regarding a single data instance. A cost function can commonly be expressed in terms of a loss function

$$\text{Cost}(f, \mathcal{D}) = \frac{1}{n} \sum_{i=1}^{n} L(y_i, \hat{y}_i), \tag{4}$$

as the average of the loss function over the data. Through different choices of cost function one ends up with different methods for estimation, resulting in different models for the same data set.

1. OLS

This subsection is reused from our previous work (Bjørnstad et al.*, 2024, p. 4).*

Ordinary least squares (OLS) is a linear regression method that seeks to minimize the following cost function (Hjorth-Jensen, 2024, Linear Regression):

$$C(\boldsymbol{\beta}) = \sum_{i=0}^{n-1} (y_i - \tilde{y}_i)^2 \tag{5}$$

From Eq. 5 the equation for the optimal $\boldsymbol{\beta}$ can be derived. This is done by taking the derivate of the cost function w.r.t. $\boldsymbol{\beta}$ and finding the minimum, which yields the optimal $\boldsymbol{\beta}$ for OLS

$$\hat{\boldsymbol{\beta}}_{\text{OLS}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} = \boldsymbol{H} \mathbf{y} \tag{6}$$

The $\boldsymbol{H}$ is popularly called the Hessian matrix, which is stated for OLS specifically in Eq. 6, but the term "Hessian matrix" is a general term for a square matrix of double derivatives.

Ordinary least squares provides an unbiased estimation - meaning the expected value of the estimated betas is equal to the true betas.

2. Ridge

This subsection is reused from our previous work (Bjørnstad et al.*, 2024, p. 4-5).*

An extension of the ordinary least squares method is to add a penalization term to the cost function (Hastie *et al.*, 2009, p.61-68). There are many reasons why this is often preferred.

Firstly, when OLS is performed it is assumed that the matrix $\boldsymbol{X}^T\boldsymbol{X}$ in Eq. 6 is invertible. This may not always be the case due to correlation between the predictors in the data set, or if $p > n$. In these cases the matrix will not be full rank, i.e. not invertible. A mathematical fix to this is to add a (small) number λ along the diagonal:

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X} - \lambda\boldsymbol{I})^{-1}\mathbf{X}^T\mathbf{y} \tag{7}$$

These methods also help to reduce overfitting. Intuitively this is a result of penalizing "too good of a fit" on the training data, meaning it's harder for models to get overfit.

Eq. 7 is the general equation for the coefficients in penalized regression, where the parameter λ controls the regularization. Among the many types of choices for penalization metrics are the L1-norm penalty, also known as Lasso, and the L2-norm penalty, known as Ridge. Different types of penalties yields in different properties and interpretations related to the resulting models.

In Ridge regression, the L2-norm penalty gives us the following cost function:

$$C(\boldsymbol{\beta}) = \sum_{i=0}^{n-1}\left(y_i - \sum_{j=1}^{p-1} X_{ij}\beta_j\right)^2 + \lambda\sum_{j=1}^{p-1}\beta_j^2 \tag{8}$$

This can alternatively be expressed as two equations, where the restraint on $\boldsymbol{\beta}$ is explicitly stated:

$$C(\boldsymbol{\beta}) = \sum_{i=1}^{N}\left(y_i - \sum_{j=1}^{p-1} X_{ij}\beta_j\right)^2 \tag{9}$$

$$\sum_{j=1}^{p}\beta_j^2 \leq t, \tag{10}$$

Here the value of t is directly related to the value of λ.

Ridge regression puts a penalty on all the β-terms except the intercept, which is held out during training. Had it been included, the model would depend on the chosen origin and this dependency undermines the principle of shift invariance (Hastie *et al.*, 2009, p. 63).

In the ideal case, when the design matrix $\mathbf{X}$ is orthogonal, one has $\mathbf{X}^T\mathbf{X} = \boldsymbol{I}$. From Eq. 7 gets that:

$$\boldsymbol{\beta}_{Ridge} = (\boldsymbol{I} - \lambda\boldsymbol{I})\mathbf{X}^T\mathbf{y} \tag{11}$$

From Eq. 11, it immediately follows that, in the case of $\mathbf{X}$ orthogonal, one gets the relation:

$$\boldsymbol{\beta}_{Ridge} = \frac{1}{1+\lambda}\boldsymbol{\beta}_{OLS} \tag{12}$$

B. Classification

Regression is one of the two main applications in which statistical learning is used for prediction; *classification* is the other. Classification aims to, evidently, model a way to classify a data point. As one might infer the two have a lot of conceptual overlap, and in a sense they represent two sides of estimation; the continuous and the discrete (Hastie *et al.*, 2009, p. 10). Many methods for classification even utilize some of the same methods used for regression, only now including extra step(s) to wrap or classify the computations.

Similarly as with linear regression, one has a design matrix X, and wishes to predict an outcome for each data point based on these values (Hjorth-Jensen, 2024, Logistic Regression). Only now, the goal is not a numerical value, but rather to classify data points into one of K classes.

While there is a wide range of classification tasks, the most common are binary classification problems (Hastie *et al.*, 2009, p.121). These normally constitute yes/no, true/false, diseased/not diseased, etc. and are usually represented 1/0 correspondingly. Often methods for classification use a probabilistic approach, calculating some sort of probability for each class.

1. Cross Entropy and Maximum Likelihood

When choosing a cost function for the training of a classification model least squares or similar methods are no longer an option. These functions account for neither the discreteness or the nonlinearity of the classification task. As a lot of methods for classification necessitate a probabilistic interpretation of the output, one needs a cost function that reflects the probability of each class accurately. In particular, our goal is to make the predicted probabilities of the correct classes as close to 1 as possible, while minimizing the probabilities of incorrect classes. Here *cross-entropy loss* becomes useful.

Entropy stems from the field of information theory, introduced as something akin to a measure of uncertainty (Shore and Johnson, 1981). Building on this, cross-entropy was later proposed as a quantitative measure of how far a predicted probability distribution deviates from the true distribution that it aims to model. The standard formulation of cross entropy is often referred to as binary cross-entropy, used in binary classification tasks. For one data point this loss function is expressed mathematically as (Hastie *et al.*, 2009, p.31)

$$L(y, \hat{y}) = -\left(y \cdot \log(\hat{y}) + (1-y)\cdot\log(1-\hat{y})\right) \tag{13}$$

Here, y represents the true label (1 when belonging to the class, and 0 otherwise), and $\hat{y}$ is the predicted probability of the class. When the true label $y = 0$, the loss becomes $L = -log(1-\hat{y})$ which penalizes large values and pushes

the prediction toward 0. Similarly when $y = 1$ we get $L = -log(\hat{y})$, which in turn penalizes small values and pushes the prediction towards 1.

For computations over larger batches the cost function averages over the loss for each data point, providing the equation:

$$C(\mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{N}\sum_{i=1}^{N}(y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)) \tag{14}$$

Binary classification problems are most often encountered, but cross-entropy loss can be extended to multi-class classification. This will however not be covered in this report.

The optimization of a binary classification problem can also be viewed as a *maximum likelihood estimation* (MLE) of a Bernoulli distribution (Goodfellow *et al.*, 2016, p.129-133). The goal is to maximize the likelihood of observing the true labels, given the input and our weights and biases, such that the observed data is the most probable. For one data point the posterior probability is given

$$Pr(y_i|x_i, \theta) = \hat{y_i}^{y_i}(1 - \hat{y_i})^{1-y_i} \tag{15}$$

θ represents the models parameters; the weights and biases. Rescaling this probability does not shift the maximum, so to ease computations we take the log of both sides to obtain the log-likelihood. Additionally, maximizing an equation is equivalent to minimizing it's negative counterpart. This yields the loss function

$$L(y, \hat{y}) = -\left(y \cdot \log(\hat{y}) + (1 - y) \cdot \log(1 - \hat{y})\right) \tag{16}$$

Trivially, this is the same as the equation for cross entropy stated in Eq. 13.

For batches of data points, we have the maximum likelihood given

$$\mathcal{L}(\theta) = \prod_{i=1}^{N} P(y_i|x_i; \theta) = \prod_{i=1}^{N} \hat{y_i}^{y_i}(1 - \hat{y_i})^{1-y_i} \tag{17}$$

By the same logic as for one data point given above (and the rules of logarithms), the resulting cost function is then given

$$C(\mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{N}\sum_{i=1}^{N}(y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)) \tag{18}$$

Which, again, is trivially the same as the cost function for binary cross-entropy stated in Eq. 14.

C. Logistic Regression

Logistic regression is a method that seeks to use linear functions to model the posterior probabilities of an instance belonging to a class - while still maintaining a legal probability distribution with values ranging $[0, 1]$ and summing to one (Hastie *et al.*, 2009, p.119).

Standard implementation of logistic regression is commonly a binary case. The aim is thus to decide whether an instance belongs to a class, or not (Jurafsky and Martin, 2024, p. 78). Borrowing from standard linear regression, the posterior probability for a data instance belonging to the class, i.e. $G = 1$, given x can be expressed as:

$$\Pr(G = 1|X = x) = \beta_0 + \beta^T x \tag{19}$$

However, this equation yields values ranging anywhere on the real number line, and thus fulfills neither of the aforementioned criterion for the distribution. This prompts the introduction of the *sigmoid function*, also called the logistic function;

$$\sigma(z) = \frac{1}{1 + \exp(-z)} = \frac{\exp(z)}{1 + \exp(z)} \tag{20}$$

By taking values in the real numbers, and forcing them between $[0, 1]$, one can ensure the former part of the distribution criteria - leaving us with the following expression for the first posterior probability.

$$\Pr(G = 1|X = x) = \frac{\exp(\beta_0 + \beta_1 x_1)}{1 + \exp(\beta_0 + \beta_1 x_1)} \tag{21}$$

The latter part of the criterion is ensured trough the definition of the second class probability;

$$\begin{aligned}\Pr(G = 0|X = x) =& 1 - \Pr(G = 1|X = x) \\ =& \frac{1}{\exp(\beta_0 + \beta_1 x_1)}\end{aligned} \tag{22}$$

The final step in this classifier is in many ways the simplest one; the decision with the highest probability is chosen. This will arbitrarily be the equivalent of looking only at the posterior probability of the instance belonging to the class, $P = Pr(G = 1)|X = x)$ with a decision boundary of 0.5. For $P \geq 0.5$ the decision then falls to "yes", and the opposite when $P < 0.5$. Which side of the decision the midpoint is assigned to is of no consequence to the models performance, but it is commonly rounded upwards; set as belonging to "yes" category.

Logistic regression can be generalized for the non-binary case, which is then called *multinomial logistic regression*. This generalization is done by comparing the posterior probabilities of the instance belonging to each of K classes respectively. Multinomial logistic regression will not be further covered in this report.

D. Gradient descent

In the case of linear regression and least squares (section II.A), the optimal coefficients for minimizing the cost

function can easily be found analytically. This is however not always the case; for more complicated methods, like logistic regression and neural networks, the steps for optimizing the model are not as simple and in many cases can only be approximated numerically (Hjorth-Jensen, 2024, Week 40). One method for this type of numeric optimization, and arguably the most popular one, is *gradient descent* (GD). In addition to benefits of efficiency and performance, gradient descents triumphs in it's versatility as it can be used for a wide range of optimization problems - including linear regression, logistic regression, neural networks, and many more.

Imagining the graph of a cost function as a hilly landscape; global and local minima as valleys, poorer solutions as hills - GD seeks to land in the deepest valley, i.e. the global minimum. By looking at the slope of the landscape one can keep taking steps in the steepest descending direction, and this way hopefully end up at this minimum.

$$-\nabla_\theta C(\theta) = -\begin{bmatrix} \frac{\partial C}{\partial \theta_1} \\ \frac{\partial C}{\partial \theta_2} \\ \vdots \\ \frac{\partial C}{\partial \theta_n} \end{bmatrix} \tag{23}$$

More precisely, one finds the gradient of the cost function with respect to the parameters of the model - which provides the direction of steepest increase. The negative of the gradient will then provide the direction of steepest decrease, as given for a general cost function in Eq. 23. θ represent the entire model, while the θ_is represent each of the parameters present in the model.

$$\theta_{t+1} = \theta_t - \eta_t \nabla_\theta C(\theta_t) \tag{24}$$

When updating the parameters in the model, we now use this gradient combined with a new parameter η, as shown in Eq. 24. η is commonly known as the *learning rate*, which we will discuss further below.

For a model containing only a few data-points and parameters computing the gradient in each step is feasible. When considering larger models and data sets with more parameters and training points, and complicated numerical processes computing the gradients, standard gradient descent becomes very computationally heavy, and is often avoided. It is often replaced by *stochastic gradient descent* (SGD).

In literature there are two main views on how SGD is motivated, with slight variations for the implementation. Both methods are however rooted in the same underlying theory; aiming to compute compute approximations of the gradients using subsets of the training data, and thereby reducing computational cost.

The first approach is presented in Goodfellow *et al.* (2016, p. 291). For each iteration (*epoch*) of the training process one *mini-batch* of size k is randomly chosen. Each epoch a new batch is randomly chosen. The motivation behind this approach is to achieve a good approximation to the gradient in each step, while making each step less computationally heavy.

An alternative approach can be found in Raschka *et al.* (2022, p. 47). Here, the training data is split into m equal mini-batches in each epoch, and these mini-batches are looped over; calculating the gradients and updating the model for each batch. In this way, each epoch is about as computationally heavy as in normal gradient descent, however the model parameters are updated more frequently, and thus convergence is reached faster (Raschka *et al.*, 2022, p. 47).

In essence, the two versions are virtually equivalent. Each step in the inner loop of Raschka's version and each epoch in Goodfellow's, are effectively the same. For coinciding values of k and m, the batches will even be congruent for each iteration. Raschka's method may converge slightly faster by updating parameters more frequently, while Goodfellow's introduces a slightly higher degree of randomness by choose each batch with replacement. However, as the size of the dataset and number of epochs increases, this difference becomes negligible. For the rest of the report, any mention of SGD is to be understood as an implementation by Goodfellow's version.

1. Learning rate

For both GD and SGD, the gradients are multiplied by a *learning rate* η_t when updating the model. This learning rate is a positive scalar, usually chosen of small value to make each update relatively small (Goodfellow *et al.*, 2016, p. 84). As a gradient is only representative for the change at the exact point of calculation, making too large of an update to the model based on a gradient is risky. By changing the model to much, i.e. moving it too far, one could be moving to areas of the function that are topographically very different than the starting point. In the worst case, a large step could overshoot the minimum.

While having too large of a learning rate could result in overshooting the minimum, and hence not reaching convergence; too small of a learning rate also has it's disadvantages. The smaller the learning rate, the more iterations are required to approach the minima, causing the training to become computationally heavier. To find a balance between the two, some opt for choosing a learning rate by trial error.

Since the drawbacks of a low learning rate are most unfavorable in the early iterations, and a too high learning rate mostly causes problems in the later iterations, an intuitive solution is to start with a higher learning rate and reduce it after a set number of iterations.

$$C(\theta_t - \eta \nabla_\theta C(\theta_t)) \tag{25}$$

Another approach is calculating the function in Eq. 25 for different values of η, and choosing the η resulting in the smallest value. This technique is called *line search* (Goodfellow *et al.*, 2016, p. 84). As computing every possible value of η would resemble an exhaustive search, line search still requires us to manually choose a set of learning rates, and every step in the iteration is limited to choose one of those learning rates.

A more common alternative is to opt for *adaptive learning rate* algorithms.

2. Adaptive learning rates

Both regular and stochastic gradient descent faces several issues when trying to approach the global minimum. Problematic points in the loss function, such as local minima and saddle points can cause the iterative procedure to lose progress (Ketkar, 2017, p. 116-117). In addition, as described above, a constant learning rate is hard to tune, and can cause issues both being too large and too small within the same optimization problem. To address these issues, a series of algorithmic variations have been proposed GD/SGD (i.e. Eq. 24) to replace the step of updating the model. Some of these algorithms requires us to initialize different parameters to be updated in each iteration. If nothing else is explicitly stated, all the iteratively updated parameters in the algorithms below (except θ) are initiated as 0 or 0-vectors.

A common technique in these algorithms, is to utilize information from the previous updates to improve the next one. One of the more simple algorithms achieving this, is the *momentum* algorithm. In this algorithm, one uses a fraction of the update from the previous step, to help us guide the next.

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta_t \nabla_\theta C(\theta_t) \\ \theta_{t+1} &= \theta_t - v_t \end{aligned} \tag{26}$$

This method replaces the update-step of Eq. 24 with Eq. 26. Intuitively one could imagine going full force down a hill (towards the global minimum); with standard implementations of GD or SGD, one could easily be stopped by local minima along the way. By including some of the previous momentum, one heightens the chance of escaping the local minimum and continuing toward the global minimum. Note that this algorithm requires us to tune two constants (η and γ), instead of only one constant as in the original case.

Other optimizing methods focus on adjusting the learning rate for each iteration. One such algorithm is the *AdaGrad* algorithm, which sums the squares of all previous gradients, and scales the learning rate in each iteration by the inverse of the square root of this sum (Goodfellow *et al.*, 2016, p. 303).

$$\begin{aligned} g_t &= \nabla_\theta C(\theta_t) \\ G_t &= G_{t-1} + g_t^2 \\ \theta_{t+1} &= \theta_t - \eta_t \frac{g_t}{\sqrt{G_t} + \delta} \end{aligned} \tag{27}$$

The update-step of AdaGrad is displayed in Eq. 27, where g_t^2 is g_t squared element-wise. The δ in Eq. 27 is a small constant added for numerical stability. It is commonly chosen to be around the range 10^{-7} to 10^{-8} (Goodfellow *et al.*, 2016, p. 304). It is also possible to combine the momentum and AdaGrad algorithms, by replacing the gradient in momentum with the update in the last line of the AdaGrad algorithm.

The *RMSProp* algorithm is an improved version of the AdaGrad algorithm, designed to keep the learning rate from shrinking to fast in non-convex areas of the cost function (Ketkar, 2017, p. 122).

$$\begin{aligned} g_t &= \nabla_\theta C(\theta_t) \\ G_t &= \rho G_{t-1} + (1-\rho) g_t^2 \\ \theta_{t+1} &= \theta_t - \eta_t \frac{g_t}{\sqrt{G_t} + \delta} \end{aligned} \tag{28}$$

RMSProp is implemented by the algorithm described in Eq. 28. Compared to AdaGrad, RMSProp uses a exponentially decaying average, discarding history from the extreme past (Goodfellow *et al.*, 2016, p. 304). The algorithm introduces a new parameter ρ controlling this moving average. RMSProp can also be combined with momentum as described for the AdaGrad algorithm.

One of the most widely used adaptive learning rate optimization algorithms in the recent years is *Adam*. This algorithm computes the updates by maintaing weighted averages of both g_t and g_t^2 (Ketkar, 2017, p. 123).

$$\begin{aligned} g_t &= \nabla_\theta C(\theta_{t-1}) \\ m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\ \theta_t &= \theta_{t-1} - \eta_t \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \end{aligned} \tag{29}$$

While Adam has more hyper-parameters than any of the above-mentioned algorithms, these parameters have intuitive interpretations and require little tuning (Kingma and Ba, 2017). Adam combines a weighted momentum (m_t) with a weighted average of the second order gradients (v_t) similar to RMSProp.

While optimization algorithms for adaptive learning rates are widely discussed in the literature, there is no general consensus on what algorithms are best for different

applications (Schaul *et al.*, 2014). Hence, testing different algorithms for your specific data-set, model and problem formulation, is the way to go.

3. Exploding gradients

Some models, such as neural networks with many layers (see section II.E), often have extremely steep cliff-like regions (Goodfellow *et al.*, 2016, p. 285). This can cause the gradients to become extremely large, causing the update-step to move the parameters too far. This phenomenon is known as *exploding gradients*. While this may cause serious problems for the learning process, it has an easy solution. Through a process called *gradient clipping*, we simply rescale the norm of the gradients whenever it goes above some threshold (Ketkar, 2017, p. 93).

If g denotes the gradient and c is our threshold, we set $\hat{g} = \frac{c}{|g|}g$ if $|g| > c$, and use $\hat{g}$ as the gradient in the learning step. As the most important information supplied by the gradient is the direction towards a lower value of the cost function, this method causes no harm by scaling the gradient. The size of the gradient is used only as a hint to how far we might have to move to find the minima, but if this value proposes a very large step, gradient clipping intervenes, making dangerously large steps less likely to happen (Goodfellow *et al.*, 2016, p. 286).

E. Feed Forward Neural Networks

Although methods like linear regression for continuous tasks and logistic regression for classification are suitable for a lot of applications, a more adaptive and versatile method is necessary to broaden the possibilities of use.

Neural networks (NN) make no assumptions about underlying data-structures, and can be fit for a broad range of tasks, including regression and classification (Hjorth-Jensen, 2024, Neural networks). Through *layers* of *nodes* (often called *units*), the network imitates the process of neurons firing in the brain, and can be fit to many complex problems.

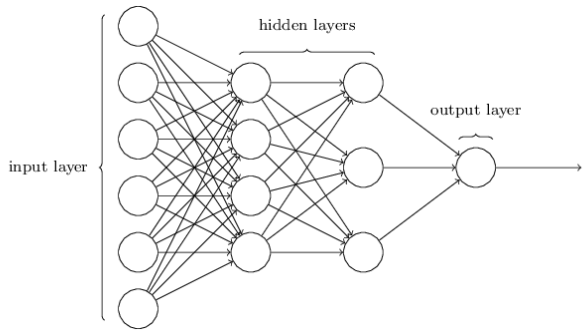

FIG. 1: Illustration of a generic neural network. (Taken from Nielsen, 2015, Ch.1).

A general NN consists of an input layer, k hidden layers and an output layer - all consisting of a varying number of nodes or units, as visualized in Fig. 1. The number of nodes in a layer is denoted as the width of the layer, and the number of layers in a network is denoted the depth of the network. Hidden layers are not hidden in the sense that they're not a visible part of the model; but rather that when using a trained model one never interacts with anything past the input and output layers, thus the internal nodes appear as hidden. Between the nodes we have connections called *weights*. These determine the influence from one particular node to another.

The weighted model stems from the mathematical function for an artificial neuron given as follows;

$$y = f\left(\sum_{i=0}^{N} wx_i\right) \tag{30}$$

Here f is some *activation function*that takes a weighted sum of N inputs and maps them to an output. Activation functions are further covered in the next subsection, II.E.1.

How many layers an NN has, how many nodes each layer has and how they're connected make up what's called the *architecture* of the neural network (Nielsen, 2015, Ch.1). These decisions are normally made case-by-case depending on the task at hand, input size, desired output size, etc. Apart from compatible dimensions there are no clear choices for what constitutes a "better" architecture, however more complex tasks are commonly better solved by more "complex" networks; i.e. more layers and more nodes.

It is sometimes specified wether a network is *fully connected* or not. A fully connected network, or layer, is simply connected in a way that each node receives the output from every single node in the preceding layer. Fig. 1 is an example of a fully connected NN. However, a fully connected network can always act as a "not-fully connected" network simply by having some of it's weight set to 0 and thereby nullifying the corresponding connections.

The simplest model for an artificial neural network is called a *feed forward neural network* (FFNN), where logically enough the flow of information moves only in one direction; forward. Consider for instance a FFNN with one hidden layers. The input-layer receives the input X, which is then passed through the first weights W_0 to the second layer. Here the weighted sum is passed through an activation function f_1 and passed via W_1 to the output layer. In the output layer the weighted sum is passed through f_2, before finally being output from the model. For each node n the process can be formally expressed as:

$$y = f\left(\sum_{i=0}^{N} w_{0n}x_i + b\right) \tag{31}$$

(Ketkar, 2017, p.17). Here one can clearly see the similarities to Eq. 30. b represents what is called a *bias term*. Looking back to the structural similarities of a biological NN, the bias allows one to skew "how easy it is to get the neurons to fire". I.e., by altering the bias terms one can shift the the output values in a desired direction - in theory. This is however easier said than done, and directly altering either weights or biases directly are not common procedure. Instead one relies on a good choice of cost function and the training of the NN to produce fitting parameters.

1. Activation functions

The activation functions are the key to how NNs generalize beyond linear methods. As might be inferred, these activation functions should **not** be linear (Goodfellow *et al.*, 2016, p.168). For an NN with one hidden layer and activation functions given as:

$$\begin{aligned} f_1(\mathbf{x}) &= \mathbf{W_0}^T\mathbf{x} + b_1 \\ f_2(\mathbf{x}) &= \mathbf{W_1}^T\mathbf{x} + b_2 \end{aligned} \tag{32}$$

The entire network could then be expressed:

$$\begin{aligned} F(\mathbf{x}) &= f_2(f_1(\mathbf{x})) \\ &= \mathbf{W_1}^T(\mathbf{W_0}^T\mathbf{x} + b_1) + b_2 \\ &= \mathbf{W_1}^T\mathbf{W_0}^T\mathbf{x} + \mathbf{W_1}b_1 + b_2 \\ &= \mathbf{W}^T\mathbf{x} + b \end{aligned} \tag{33}$$

The entire NN itself is just one big chain of all it's weights and activation functions nested. Were the activation functions to be linear, the network itself would be linear (as shown in Eq. 33) - undermining the fundamental purpose of an NN. Beyond non-linearity there are few hard restrictions on the activation functions, granted some are to be preferred. The rest of this section will use the convention of $\mathbf{z}$ representing the vector of "raw output" from the previous layer. Raw output simply refers to the values before they're passed through the activation function. An activation function applied to the entire vector, is simply applied element-wise - i.e. the input and output are congruent and

$$f(\mathbf{z})_i = f(z_i) \tag{34}$$

One example of a common activation function for classification problems is the Sigmoid function (Eq. 20), as used in logistic regression. Tweaking the architecture of an NN, one can deduce that logistic regression is congruent to a FFNN with no hidden layers and the Sigmoid function as the activation function.

Alternatively the *softmax* function is another common function for classification (Hjorth-Jensen, 2024, Logistic Regression). This is an extension of the Sigmoid function, and is also what's used in place of the sigmoid function for multinomial linear regression. The softmax function normalizes a vector of input values so that the resulting outputs sum to 1. While softmax can be used in internal nodes, it is more commonly used for the output layer. For an input vector of size K the softmax function for each element i is given as:

$$f(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{35}$$

For internal nodes and some regression tasks the activation function *Rectified Linear Unit* (ReLU) is commonly used (Hjorth-Jensen, 2024, Building a feed forward neural network);

$$f(\mathbf{z})_i = \max\{0, z_i\} \tag{36}$$

Alternatively ReLU can be expressed as

$$f(\mathbf{z})_i = \begin{cases} z_i & \text{if } z_i > 0 \\ 0 & \text{if } z_i \leq 0 \end{cases} \tag{37}$$

As opposed to both the sigmoid and the softmax functions, ReLU does not *saturate* for large positive values. The saturation of a function is an issue where the values of the function approaches its maximum or minimum, resulting in vanishing gradients and thereby poor convergence. However, ReLU suffers from an issue called *the dying ReLU*. In essence this refers to nodes stagnating at zero; effectively dying.

One solution to the problem of the dying ReLU is to use *leaky ReLU* (Goodfellow *et al.*, 2016, p. 190);

$$f(\mathbf{z})_i = \max\{0, z_i\} + \alpha \min\{0, z_i\} \tag{38}$$

Here α is typically sat to a small value, like 0.01. Similiar to ReLU, Leaky ReLU can alternatively be expressed as;

$$f(\mathbf{z})_i = \begin{cases} z_i & \text{if } z_i > 0 \\ \alpha z_i & \text{if } z_i \leq 0 \end{cases} \tag{39}$$

This variation of ReLU allows for some portion of the negative values to be included, mitigating the risk of dying nodes.

The output layer of a NN will often have a different activation function than it's predecessors. While the choice of activation functions in the hidden layers have less direct consequences to the type of output, the last activation dictates exactly this. For classification tasks one often has either the sigmoid, the softmax or similar functions for the output layer. Commonly less restrictive functions are used for regression problems, to avoid limiting the possible output scope of the network. Some even choose to use the identity function, or similarly no activation function, for the last layer.

2. Initialization of weights and biases

To start training the NN, and navigating the landscape of the cost function, one needs a starting point (Goodfellow *et al.*, 2016, p.297). For initialization of weights and biases there are a lot of different conventions. How the values are initialized can affect things like the optimization itself, convergence and the generalization of our model. One of the most important aspects is to ensure that no nodes are the same; for nodes with the same activation function, same input values and same initial values, most training methods will update these in the same manner. This again will result in a more narrow search field were not all possibilities in the cost landscape can be reached. Although there exists ways to work around this problem it is commonly preferred to avoid this issue as best one can. Larger (distinct) initial weights will be more effective against symmetry, but will on the other hand heighten the risk of exploding gradients (section II.D.3) and saturated nodes (II.E.1).

Arguably the most common method across the field, is random initialization. As the wording entails, the weights and biases are then randomly initialized. The values are commonly selected from a Gaussian distribution, with mean zero and standard deviation 1. This method alleviates the chances of symmetrical nodes, while also mitigating the risk of too large weights. Among other methods for initialization many are specialized for specific activation functions, providing benefits like faster convergence and moderation of saturation effects.

3. Choice of cost function for neural networks

In addition to initialization of weights and biases, the training of the model largely depends on choice of cost function, which again depends on the task at hand. How a cost function looks has in itself no restrictions, but some are more suited than others. The choice of cost function is arbitrarily what dictates the landscape that is navigated in the optimization of a network. Most popular optimization techniques require a cost function that can be expressed in terms of a loss function, i.e. that can be computed for each data point on its own. For networks to be used on regression problems one commonly uses MSE (see section II.G.1), and for classification problems cross entropy (section II.B.1) is by far the common choice.

4. Training the network

For the actual training of the network, in other words finding the optimal weights and biases, one normally uses gradient descent (see section II.D). The first step in this process is finding the gradients of the cost function to navigate the parameter landscape. As each layer of a NN can be expressed as a function of its previous layer, one can imagine these expressions to quickly become quite complicated. Furthermore we have to find the partial derivatives for all weights and biases. The most used solution to this intricate problem, is called *backpropagation* (Goodfellow *et al.*, 2016, p. 200). While many libraries provide implementations of automatic gradients, such as PyTorch's Autograd (Ansel *et al.*, 2024) and Tensorflow's GradiantTape (Abadi *et al.*, 2015), it is beneficial to understand how manual backpropagation works. Not only is this useful for debugging and ensuring code behaves as expected, but it is essential for understanding the training process.

The fundamental concept of backpropagation is the use of the chain rule from calculus. Since the network's output is a composition of functions, the chain rule allows us to systematically propagate backwards through the network and compute the derivatives effectively (Nielsen, 2015, Ch.2).

Denoting the output of layer (l) as $a^{(l)}$ and the weights and biases for that layer as $W^{(l)}$ and $b^{(l)}$, respectively. For a given layer, the input $z^{(l)}$ to the activation function is:

$$z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)} \tag{40}$$

Here $a^{(l-1)}$ is the output (or activation) from the previous layer. The activation $a^{(l)}$ is then given by applying the activation function f to $z^{(l)}$:

$$a^{(l)} = f(z^{(l)}) \tag{41}$$

To calculate how each weight $W^{(l)}$ and bias $b^{(l)}$ affects the overall cost L, we start from the final layer and compute the gradient of the cost with respect to each parameter in the network, working backward through each layer. The gradient of the cost with respect to $W^{(l)}$ is obtained using the chain rule:

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial a^{(l)}} \cdot \frac{\partial a^{(l)}}{\partial z^{(l)}} \cdot \frac{\partial z^{(l)}}{\partial W^{(l)}} \tag{42}$$

where:

- $\frac{\partial L}{\partial a^{(l)}}$ is the gradient of the cost with respect to the layer's activation
- $\frac{\partial a^{(l)}}{\partial z^{(l)}}$ is the derivative of the activation function applied at layer l
- $\frac{\partial z^{(l)}}{\partial W^{(l)}}$ is the derivative of the weighted sum with respect to the weights.

By recursively applying this process from the output layer back to the input layer, back-propagation computes all necessary gradients efficiently, enabling gradient descent to adjust each parameter and reduce the cost.

F. Scaling

Scaling data before giving it to a model is often beneficial, and sometimes a necessity (Hjorth-Jensen, 2024, Linear Regression). Some methods, like ordinary least squares, are invariant to scaling and thus it does not affect the performance in any way. For methods like ridge regression, which penalizes large coefficients, scaling the data mitigates problems related to parameters of unequal magnitudes being penalized unfairly compared to each other. For neural networks, scaling the data can be beneficial for convergence of the model, and choice of scaler normally depends on the activation functions used in the model (Goodfellow *et al.*, 2016, Ch.7). Additionally models are often sensitive to extreme values / outliers, and scaling the data can help alleviate the effects of this.

It's considered good practice to not scale the intercept, as it represents a constant term without variability and scaling it will often worsen it's performance, or defeat it's purpose entirely.

A common choice for scaling is to use the *standard scaler*. This method involves subtracting the mean and dividing by the standard deviation, for each feature (i.e. parameter) respectively. Each of the columns will then have mean equal to zero and a standard deviation of one after the scaling. For the j-th feature one gets the transformation as

$$\mathbf{x_j} = \frac{\mathbf{x_j} - \mu_j}{\sigma_j} \tag{43}$$

Standard scaling is well suited for methods that assume a underlying normal distribution, such as linear regression. For datasets with large outliers, standard scaling is also beneficial as it makes the data less susceptible to the effects of extreme values. A variation of the standard scaler is sometimes used, including only subtraction of the mean and omitting the division of standard deviation. This is commonly called *zero centering*.

Alternatively the *min-max scaler* is another common option.

$$\mathbf{x_j} = \frac{\mathbf{x_j} - \min(\mathbf{x_j})}{\max(\mathbf{x_j}) - \min(\mathbf{x_j})} \tag{44}$$

Max-min scaling is ideal for methods that makes no assumptions about underlying data distributions, like NNs. Shifting the data into the range $[0, 1]$ can also be beneficial for convergence of NNs when using activation functions like the sigmoid function. However, it's worth keeping in mind that the min-max scaler is sensitive to outliers.

G. Evaluation measures

1. Regression

This subsection is reused from our previous work (Bjørnstad et al., 2024, p. 4).

$$\text{MSE} = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2. \tag{45}$$

The *mean squared error* (MSE) is a popular error measure for linear regression models, and is defined as in Eq. 45, where n denotes the number of data points in the training data, and the prediction for the i-th data point is denoted $\tilde{y}_i$. Comparing the expression above (Eq. 45) to Eq. 5, it's clear how the OLS regression method is inherently designed to minimize MSE.

R^2 is a measure of how well the model explains the variance present in the data.

$$R^2 = 1 - \frac{RSS}{TSS} = 1 - \frac{\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2}{\sum_{i=0}^{n-1}(y_i - \bar{y}_i)^2}, \tag{46}$$

$R^2 \in (-\infty, 1]$. If $R^2 = 1$ the model perfectly explains all variance, whereas a value of 0 would mean does not explain any of the variance. A prediction $\tilde{\mathbf{y}} = \bar{\mathbf{y}}$, would result in $R^2 = 0$. If $R^2 < 0$, the model is worse than a straight line. The numerator is the sum of the squared residuals, also called RSS. The denominator is the total sum of squares, in short TSS (Martin, 2022, p. 29).

2. Classification

While the mean squared errors and R^2 measures could give us a rough grasp about the quality of a classification model, there are several other methods better suited to measure the model's performance. One such measure is *accuracy*.

$$\text{accuracy} = \frac{\text{correct predictions}}{\text{total predictions}} \tag{47}$$

Accuracy is simply defined as correct predictions divided by total predictions (see Eq. 47), and is a good measure on how often the model is correct. There is however an important weakness with accuracy, especially for certain types of data. This stems from the fact that the two types of errors in a binary classification model, *false positives* and *false negatives*, are in many cases not equally bad. Furthermore, many data sets are very skewed, with one outcome having many more data points than the other. A prime example of this is models based on medical data sets, trying to predict whether a patient has some disease, such as the Wisconsin breast cancer data set (Wolberg *et al.*, 1993). Clearly, in most of these cases, it is worse if the models wrongly predict that a sick patient is healthy, than the other way around.

Predicted / Truth	**Disease**	**No disease**
Disease	true positive (TP)	false negative (FN)
No disease	false positive (FP)	true negative (TN)

TABLE I: Confusion matrix for medical data

An example is the Norwegian mammography program, where only 3% of the initially tested will receive follow up tests or treatment (Kreftregisteret, 2024). In this case, a model predicting everyone to be healthy would obviously be very bad, however it would get an accuracy score of 0.97, which is considered very good. It is also much worse to send a person with the disease home, than to send a healthy person to a follow-up consultation. Hence we need a better solution to measure the quality of the model.

A naive solution that solves the problem of skewed data sets, is removing data points from the majority set, in order to make the data set less skewed. This often a bad solution for two reasons. It reduces the original dataset by a lot of data points (in the Norwegian mammography program example a completely equally distributed data set would contain at most 6% of the data points), which is of course much worse for training the model. Secondly, it does not consider the fact that the two different error types are not equally bad.

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{48}$$

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{49}$$

For these kind of models, measures such as *precision* and *recall* are popular. They both focus on the true positives, and ignore true negatives (see table I). Precision (Eq. 48) is the fraction of predicted positives where the true values are positive, while recall (Eq. 49) is the fraction of positive cases that are predicted positive by the model (Goodfellow *et al.*, 2016, p. 418). The equations uses the definitions of true positives (TP), false positives (FP) and false negatives (FN) as shown in table I. Clearly, recall is an important measure for medical cases, as you do not want to send home sick patients. That being said, a model must also have a certain level of precision in order to be useful. Predicting every instance to be true, would yield a high recall value - but trivially it would not be a very helpful model.

$$F = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}} \tag{50}$$

A compromise between precision and recall is the *F-score* (Eq. 50). An F-score close to 1 suggests that the model is good, while an F-score close to 0 is a sign of a bad model. Even though the F-score is good at evaluating the quality of a binary classification model, it provides little about what is wrong with the model, giving you no insight to if the error stems from the precision or the recall, or both. Hence, one will have to also calculate the precision and recall separately to figure out why a model has a low F-score.

III. Method

A. Data

1. Regression data

We have generated the data for the regression problem using the Franke function (Franke, 1979, p. 13).

$$\begin{aligned} f(x,y) =& \frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) \\ &+ \frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)^2}{10}\right) \\ &+ \frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) \\ &- \frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right) \end{aligned} \tag{51}$$

$$f^*(x,y) = f(x,y) + \frac{3}{10}\mathcal{N}(0,1) \tag{52}$$

The Franke function is defined by Eq. 51, on the domain $[0,1]^2$. In order to challenge the models, and test them for weaknesses such as overfitting, we have added a stochastic noise term to the function, as can be seen in Eq. 52, where $f(x,y)$ is as defined in 51. For the Franke data, we use 101 data points along each axis, for a total of 10201 data points (i.e. 101^2). The Franke function with noise is plotted in Fig. 2. All further references to the Franke function in the results and discussions refers to the Franke function with noise (Eq. 52).

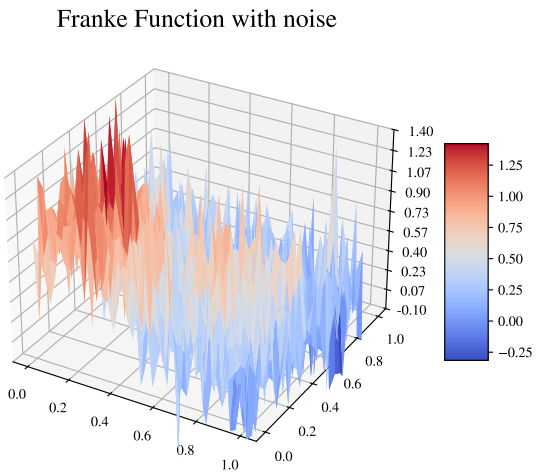

FIG. 2: The Franke function for $x, y \in [0, 1]$ with added noise term: $(3/10)\mathcal{N}(0, 1)$.

2. Classification data

For the classification problem, we test the models using the Wisconsin breast cancer diagnostics data set (Wolberg *et al.*, 1993). The data set contains 30 different features of digitized images computed from a fine needle aspirate of a breast mass, describing the characteristics of the cell nuclei in the images. It also contains a column describing the diagnosis of the breast tissues; malignant or benign. There are 569 patients registered in the data set, with no missing data. 357 of the patients have diagnosis benign, while 212 are malignant. More details about how the data is produced is available in Street *et al.* (1993).

3. Data handling

To make sure the models learn the underlying structure of the data, instead of just copying the set of data points, we split our data in a training set and a test set. This is common practice in machine learning problems, and is known as *train-test split*. By setting aside 30% of the data as test data, we make sure that the model can, after being trained on the training set, predict new data points it has never seen before with a reasonable accuracy. When presenting accuracy and error numbers in the results used for comparing models, we use the measures on the test data unless indicated otherwise.

The Franke data has input in the range $[0, 1]$ for both variables. This means that min-max scaling (Eq. 44) would have no effect on the data, as it is already min-max scaled. We tested using standard scaling (Eq. 43), however this gave worse results, hence we decided on not scaling the data.

For classification, we also tested both min-max scaling (Eq. 44) and standard scaling (Eq. 43), described in section II.F. When testing logitstic regression and the neural networks, it became clear that min-max was superior in both cases.

B. Implementation of the models

1. Linear regression

For linear regression, we reuse our earlier implementation of ordinary least squares regression and ridge regression for the Franke data set. A more detailed discussion of this implementation can be found in Bjørnstad *et al.* (2024).

2. Neural network

We implement a flexible neural network class in our code, enabling it to work for both classification and regression problems, with different structures, cost functions, activation functions and learning rate algorithms. Having cost functions, number and size of layers, activation functions, mini-batch sizes and gradient descent methods as arguments when setting up the class, allows us to easily explore different models. We input the network shape as an array of integers, where the first element represents the input shape, the last element represents the output shape and the rest represent the sizes of the hidden layers. The argument for mini-batch sizes for SGD defaults to normal gradient descent if set to 0.

While having implemented backpropagation with manual derivatives to find the gradients, we mainly use the `grad` function from `JAX` (Bradbury *et al.*, 2018) to get the gradients more efficiently. Our tests includes a control check that the gradients form `JAX` and our manual gradients are identical.

The initialization of the parameters is done with a normal distribution with expectation 0 and standard deviation 1. While there exists more advanced initialization schemes (see section II.E.2), we decided not to spend time implementing these. This is due to the fact that we are using several different activation functions, and many of the initialization schemes are specialized at being good for one particular activation function. We also use relatively simple neural networks structures with a relatively small amount of hidden layers and nodes, meaning it does not cost us much extra time or computation power needing to spend a few more iterations training them.

When using SGD, we have only one update in each epoch matching the first view in the discussion in section II.D. We make the mini-batch by making a list of all the indices of data points, shuffling it using `NumPy`'s shuffle method (Harris *et al.*, 2020), and selecting the first m indices (where m is the size of the mini-batch). Before running the parameters and gradients through the adaptive learning rate algorithm of choice, we flatten/ravel the

parameter object into an array from `NumPy` (Harris *et al.*, 2020), enabling us to perform mathematical operations on the entire set of parameters at once in an effective way. After updating the the parameters, we reshape them into the original structure.

Due to neural networks sometimes taking long to train, and the stochastic nature in initialization and mini-batches, we have implemented methods to save and load a network. This is done through methods in the neural network class, and all the required information about the networks i stored as a `json` file. We also made it possible to save a half-trained networks, and resume the training after loading it, including storing the epoch number. Since the networks in this paper are relatively small and fast to train, we did not actively use these methods, however they will be useful in further expansions and applications of this code base.

3. Logistic regression

As explained in the theory, a logistic regression model is in essence a neural network with no hidden layers and the sigmoid function as activation. Our implementation is inspired by this; we make the layers similarly as in the neural network code, just limited to only having one layer. Furthermore, we implemented cross entropy as the cost function inside the class, adding an l_2 regularization part to it, with parameter λ defaulting to 0.

We initiate our weights and biases using a normal distribution with expectation 0 and standard deviation 1. While we could have used specialized initiation schemes as described in section II.E.2, we reached convergence fast enough with the naive approach to not prioritize that. We use `JAX` (Bradbury *et al.*, 2018) to get the numerical gradients while training the model, and implement our mini-batch code as in the neural network case. We also implement custom functions for flattening and reshaping the weights and biases, but we use different functions for logistic regression than for neural networks, since we can make them much simpler and somewhat faster as logistic regression has an easier structure.

C. Exploration and testing

1. Exploration of the properties of a neural network

For the Franke function data, we use mean squared error as our cost function. For the classification case, i.e. the cancer data, we opt for binary cross entropy as our cost function.

As a measure of performance, R^2 is used for the Franke function data, while accuracy is used for the Wisconsin breast cancer data set during the exploration. In the end, we calculate accuracy, recall, precision, and F-score for the classification case.

Sigmoid is always utilized as the activation function in the final layer for the classification. For the regression (Franke data), we do not apply any activation function in the final layer. Formally, our activation function is the identity.

First, we want to find a good optimizer to use for the exploration of the other moments. To do this, we initialize our neural network model with a learning rate of 0.005 and two hidden layers of size 4 and 8 respectively. We train the models for 300 epochs through the entire exploration process, before increasing it in the final selected model. We use ReLU as activation function i both the hidden layers. For the Cancer data, sigmoid is used at the final layer, whereas for the Franke function data, we do not use an activation (i.e. identity). We use a batch size of 200 and 1000 for the Cancer data and the Franke function data respectively.

We use the following optimizers: constant, momentum, Adam, Adagrad with momentum and RMSprop. A sixth optimizer, Adagrad, is also implemented, but performs so badly on the neural network for the regression case that it surpresses the visibility in the plots. We therefore remove it completely. We train six different models, one for each of the optimizers, for the two different data sets. We choose the optimizer that performs best for use in further explorations of the neural network.

Next, we explore the structure of the network. We choose to test two and three hidden layers with each having the size 8 or 24. This amounts to 12 different combinations. These values were chosen after trial and error. With the exploration taking quite a lot of computation time, this seems like a good set of values to explore the properties of the structure of the neural networks. We find the network structure that produces the best performing model in terms of accuracy or R^2, and keep this accuracy for further exploration.

After that, we study the activation functions of the hidden layers. At the final layer we choose to always use sigmoid for the classification problem and the identity function for the regression problem. Three different activation functions is tested: ReLU, sigmoid, and leaky ReLU. For two hidden layers, these three activation functions can be combined in eight ways. Here, as in the previous steps, we find the best set of activation functions for each problem and continue with these.

Furthermore, we conduct a grid search for the best combination of initial learning rate and batch size. For both problems, we test the following learning rates: {0.01, 0.001, 0.0001}. For the regression problem, we test the batch sizes {1000, 3000, 5000}, whereas {100, 200, 300} are tested in the classification problem. This is simply because we have a lot more data in the regression problem, seeing as the data is generated from the Franke function expression. This grid search is conducted for the three best-performing optimizers from the first step. A combination of initial learning rate and batch size is chosen

for each of the three optimizers and brought to the final stage of the exploration.

For the final step, we train the three models for a total of 500 epochs, one for each optimizer. All three use the network structure and set of activation functions found to be the best. They each have their own best combination of learning rate and batch size. We find the best model among the three and record the last accuracy or R^2. For the classification case we also find the recall, precision and F-score.

2. Exploration of the logistic regression

For the logistic regression model, we conduct tests for what optimizer to use and the size of the initial learning rate. This is done as a grid search with the values {0.1, 0.01, 0.001} for the learning rate and the following six optimizers: constant, momentum, Adam, Adagrad, Adagrad with momentum, and RMSprop. The different models are trained for 50 epochs. The version that yields the highest accuracy is chosen. We take these parameters and continue our exploration of logistic regression by testing different λ-values in the for the ridge regression penalty. Seeing as $\lambda = 0$ also is tested, this amounts to checking whether a penalty is favorable or not. For the λ value that gives the highest accuracy, we train a final model for 500 epochs and record the accuracy, recall, precision and F-score at the final step.

IV. Results and discussion

For the plots presented in this results section, we will begin displaying data starting from epoch 50. This way we can evaluate the convergence in the later epochs without the low accuracy/R^2 in the first few epochs completely dominating the plots.

A. Franke function data

1. Neural network

Based on the initial search, using the momentum optimizer seems to be a good choice for further investigation. Furthermore, we note that the constant optimizer and the RMSprop optimizer make up the best three together with the momentum optimizer.

Fig. 3 shows the R^2 for models with different network structures, but otherwise the same parameters. It is interesting to notice that for two layers with a size of 24 each we get the model which yields the highest R^2 score. We tested for two and three layers and layer sizes of 8 and 24. It seems fewer but bigger layers are favorable. The reason for this might be that due to the large amount of noise, deeper networks might be prone to overfitting.

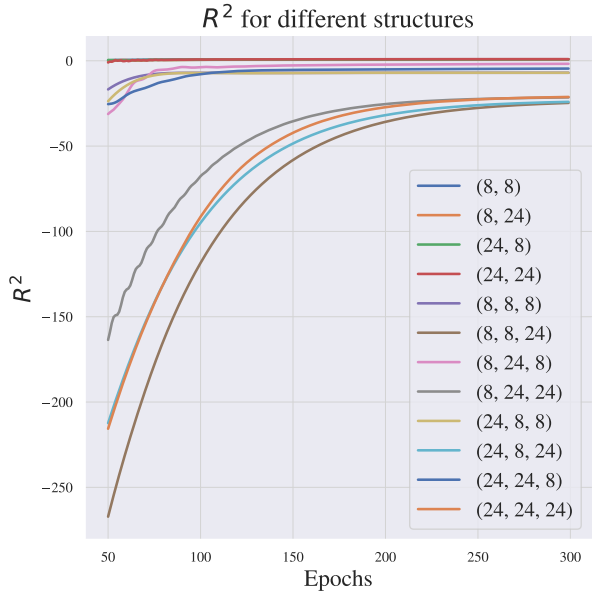

FIG. 3: R^2 for the neural network for different number of layers and sizes for each layer. The best one is (24,24).

A network with few but wide layers allows for capturing complex interactions, while still being shallow enough to avoid over-complication and overfitting.

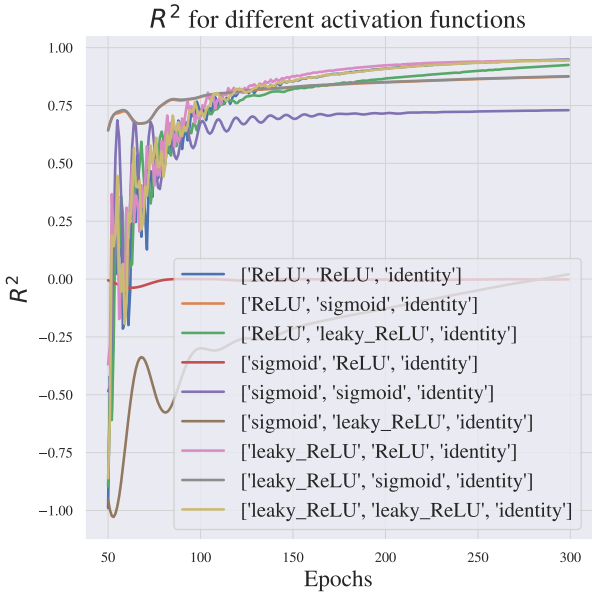

FIG. 4: The R^2 for different choices of activation functions for the two hidden layers. Identity is always used as the final activation function. The best R^2 is found when using ReLU at both hidden layers.

Using the network structure found in the previous step, we try different activation functions for the two hidden layers. Fig. 4 visualizes how the different models perform. The three worst performing models are those with sigmoid as the activation function at the first hidden layer. This is likely due to the fact that sigmoid is not designed for problems where one might want target values outside the range [0,1]. In the case where the sigmoid outputs something close to 1, and the backpropagation pushes it to become even larger, we might experience vanishing gradients, causing the training to halt before our wanted convergence point. The four best ones are the ones that combines leaky ReLU and ReLU (both orders) and just one or the other. The very best model has ReLU at both hidden layers.

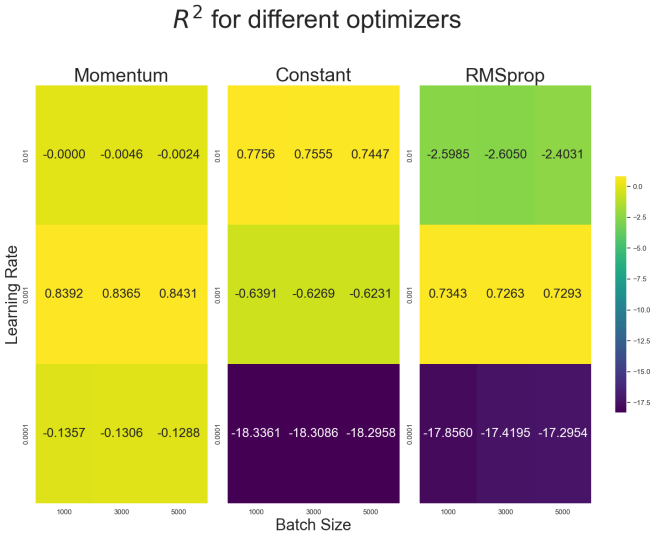

FIG. 5: The R^2 on the test set for different sets of learning rates and batch sizes for the three best optimizers.

Using (2,24,24,1) as the network structure and (ReLU, ReLU, identity) as the activation functions, the grid search for the best combination of initial learning rate and batch size is shown in Fig. 5. The three best optimizers, constant, momentum, and RMSprop, are used. We look for the combination that gives the highest R^2 on the test set for each of the optimizers.

For both momentum and RMSProp, the optimal seems to be a balanced learning rate of order of magnitude 10^{-3}. With constant learning rate however, a higher learning rate of magnitude 10^{-2} is preferred. As momentum has an extra term adding to the gradient, while RMSProp divides the gradient on a factor larger than one, we would have expected the optimal learning rates of these models to be on either side of that of the constant optimizer (momentum being lower, RMSProp higher).

The best combination of learning rate and batch size for the constant optimizer is (0.01, 1000). For the momentum optimizer we find it to be (0.001, 5000). Lastly, the model trained with RMSprop optimizer obtains the highest R^2 with the combination (0.001,1000). The impact of the batch size is not very large, which could be due to the case that 1000 data points is enough to get a representative subset of the total data. Smaller batch sizes performing slightly better could be due to some degree of randomness improving the learning steps.

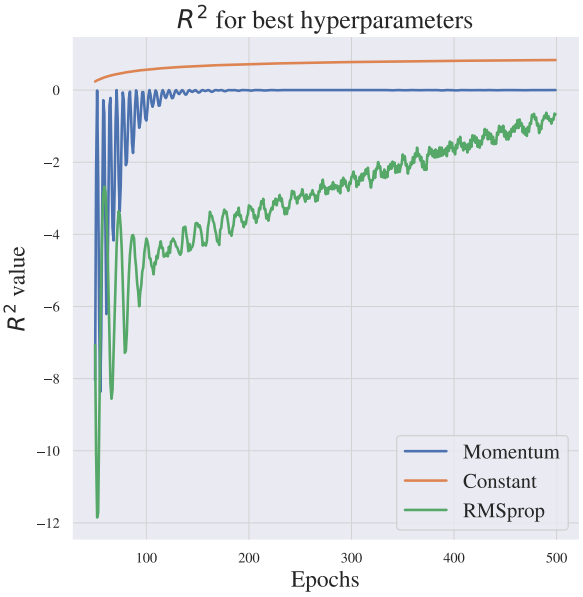

FIG. 6: The R^2 on the test set for the final three models.

The best model reaches an R^2 score of 0.83. This model is trained with the constant optimizer.

2. Comparison to linear regression

In our previous work, Bjørnstad *et al.* (2024), the maximal R^2 for the ordinary least squares model was found to be 0.37. This is far lower than the $R^2 = 0.83$ obtained from the neural network. This shows that our neural network model performs substantially better than any linear regression model. The added flexibility and possible complexity in the neural networks make them superior to linear regression when the data to be modeled does not contain linear dependencies.

B. Breast cancer data

1. Neural network

Based on the initial search, using the Adam optimizer seems to be a good choice for the further investigation.

The optimizers RMSprop and Adagrad with momentum make up the best three together with Adam.

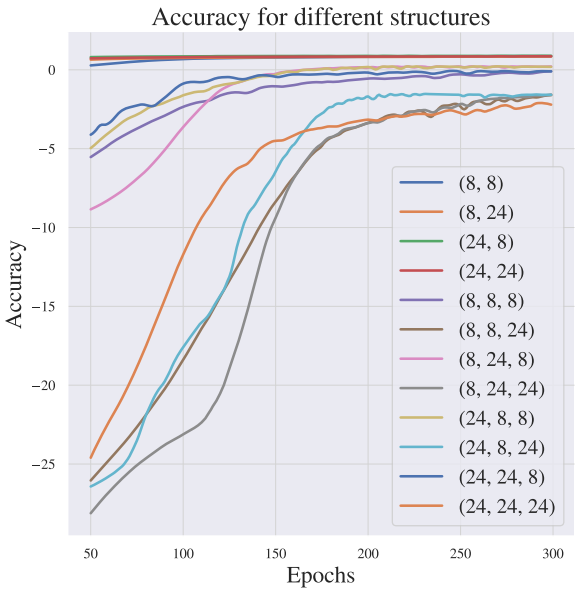

FIG. 7: The accuracy for the neural network for different number of layers and sizes for each layer. The best one is (24,8).

Using the Adam optimizer, we test for the best structure of the network. As with the regression problem, two hidden layers are preferred over three. The best performing network, the one which gives the highest accuracy, is the one with (24, 8) as the hidden layers. The four best ones are the ones with only two layers, clearly shown on Fig. 7. They majorly outperform the models with three hidden layers. The reason for this could again be overfitting in the overly complex models.

Keeping the two hidden layers of size (24,8), Fig. 8 shows how different choices of activation functions for the hidden layers lead to different accuracies. Sigmoid is always used as the activation function as the final layer. The best model is the one with (sigmoid, leaky ReLU) at the hidden layers.

Given a network structure of (30,24,8,1) and (sigmoid, leaky ReLU, sigmoid) as the activation functions, we tested for the best combination of learning rate and batch size for each of the three best optimizers. We get the best results using AdaGrad with momentum, with learning rate 0.001, and batch sizes 100 and 200 performing exactly equal. Note that as we only test this on a test set containing 171 data points, the accuracy will follow a seemingly very discrete scheme according to number of correct predictions divided by 171, hence it is expected to see different models with the exact same accuracy. It is possible that the models with learning rate 0.0001 are

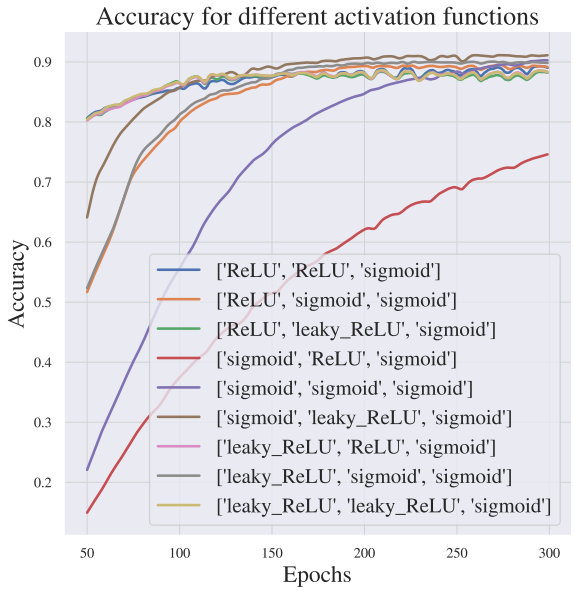

FIG. 8: The accuracy for different choices of activation functions for the two hidden layers. Sigmoid is always used as the final activation function. The best R^2 is found for (sigmoid, Leaky ReLU).

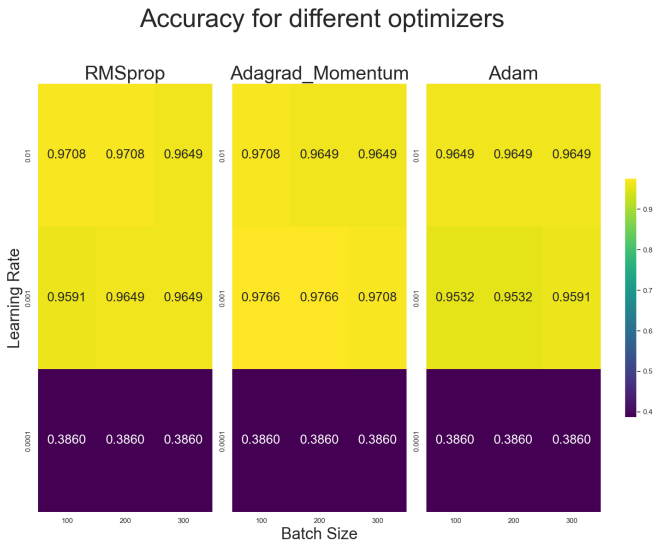

FIG. 9: The accuracy on the test set for different sets of learning rates and batch sizes for the three best optimizers.

all caught by the same local minima.

The best model for the classification problem given our method of exploring the different aspects, gives an accuracy of 0.97. This is the with RMSprop optimizer.

In Fig. 10 we clearly see the issue with having a small data set: each of the seemingly large jumps in accuracy

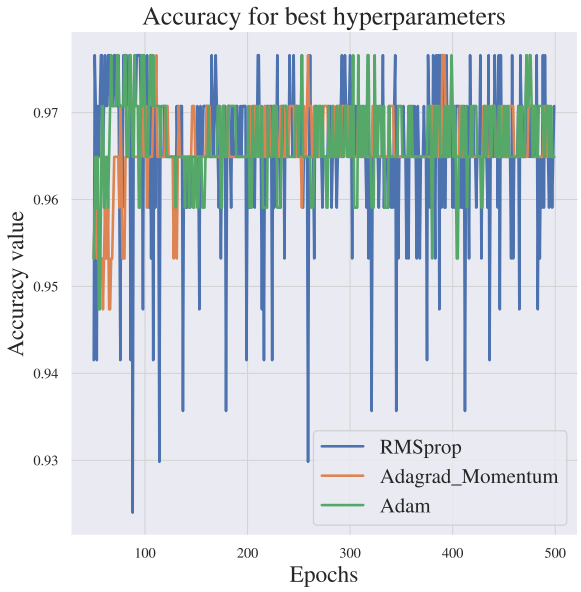

FIG. 10: The accuracy on the test set for the final three models.

does indeed represent just one patient each. It is not unexpected to see the models losing/gaining one correct prediction while closing in on its point of convergence, it simply looks weird in this case due to few data points.

2. Comparison to logistic regression

Our final logistic regression model has an accuracy of 0.97. This is the same as for the neural network.

We will consider the additional metrics recall, precision and F-score. These are presented in table II.

	NN	**LogReg**
Accuracy	0.97	0.97
Precision	0.98	0.96
Recall	0.94	0.97
F-score	0.96	0.96

TABLE II: Scores for classification problem

We see that both models has the exact same accuracy; predicting 166 of the 171 test subjects correctly. They also achieve identical F-scores, hence one could say that they are equally good. However, when looking at the precision, we observe that the neural network is performing slightly better. For the recall, the logistic regression slightly outperforms the neural network. As the discussion in section II.G.2 concluded, all errors are not equally bad; in particular it is worse to predict an ill patient to be healthy than the opposite.

In this case, the recall is thus more important than the precision, since the recall tells how many of the ill patients are predicted to be ill. The difference between the two models is that while both are as likely to perform a correct prediction for a randomly chosen patient, the neural network is slightly more likely to categorize an ill person as healthy, while the logistic regression model categorizes more healthy patients as ill. As discussed, it is quite clear that you would rather want the latter. This, combined with the logistic regression model being easier to design, requiring less tuning and being easier to interpret, we find this to be the better model.

Since a logistic regression model technically is a neural network with no hidden layers, we clearly could have found a better neural network than we did. We do believe that a way to possibly design better neural networks for this task, is to replace our current loss function by one punishing false negatives harder than false positive, making the model prone to prioritize a high recall over high precision.

V. Conclusion

From our experiments, we have the general impression that neural netwroks are performing better than the classical regression methods. Especially in the regression problem on the Franke function data, neural networks excel in comparison to linear regression. We observe that while the linear regression models struggle to grasp the structure of the data, the neural networks seem to make accurate prediction, achieving an R^2 value of 0.77, which is very good considering the large amounts of noise included in the data set. We experience that relatively shallow neural networks perform best at this problem, in particular those with 2 hidden layers, with ReLU as the superior activation function. On the optimizer part, we get the quite surprising result that a constant learning rate achieves the best performance. Our belief is that this comes down to the simplicity of the problem, and that some of the other methods causes the learning rate to slow down too much in the early iterations. As we mostly trained our models with relatively few epochs in the exploration process, we may have favored the models with quick convergence early too much.

In the classification problem, our neural networks and the classical method logistic regression has similar performance. As discussed, the neural network has a slightly higher precision, while logistic regression does somewhat better in the recall measure. This means that a positive prediction made by the neural network is more likely to be correct, however the neural networks is also more prone to wrongly categorize patients as healthy. Since both models achieve the exact same F-score, the decision on which is best is made upon assessing the importance of the types errors, and considering how much we value other prop-

erties such as computation time and interpretability of the models. Due to the fact that recall is more important in medical data sets (as discussed in section II.G.2) and the larger degree of interpretability in logistic regression models, we conclude that the logistic regression model is better. That said, we note that while we are likely to have found the absolute best logistic regression model (or at least very close to it), there are infinite opportunities within the realm of neural networks and there sure to exists a better model than ours. It should also be noted that the logistic regression model is technically also a neural network with no hidden layers. We did only test neural networks with 2-3 hidden layers, hence we could not have found that model, but we might have if we also tested neural networks with 0 hidden layers.

A possible future improvement for neural networks in the classification case is implementing a custom cost function penalizing false negatives harder than false positives. As our neural network achieves the same accuracy and F-score as the logistic regression model, while having a lower recall and higher precision, it is likely that this could immediately lead to a better model than the logistic regression. This also reiterates the point of neural networks being highly flexible, and the possibility of finding even better neural networks being very high.

We conclude that while neural networks performs substantially better than linear regression at predicting the noisy Franke function, they just barely equalize the performance of logistic regression in the classification problem. While the high flexibility of neural networks open possibilities for finding better models than we have, it also makes it harder finding those models. As we have not found a neural network comfortably outperforming the logistic regression model we found without much testing, we conclude that logistic regression might be a just as good alternative to neural networks for these kinds of problems.

References

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, R. Jozefowicz, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, M. Schuster, R. Monga, S. Moore, D. Murray, C. Olah, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng (2015), "TensorFlow, Large-scale machine learning on heterogeneous systems," .

J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, M. Suo, P. Tillet, E. Wang, X. Wang, W. Wen, S. Zhang, X. Zhao, K. Zhou, R. Zou, A. Mathews, G. Chanan, P. Wu, and S. Chintala (2024), in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)* (ACM).

J. Bjørnstad, G. Johannessen, and M. Merlid (2024), "Applying linear regression methods to terrain data," .

J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang (2018), "JAX: composable transformations of Python+NumPy programs," .

L. Fahrmeir, T. Kneib, S. Lang, and B. Marx (2013), *Regression: Models, Methods and Applications* (Springer Berlin Heidelberg).

R. Franke (1979), *A critical comparison of some methods for interpolation of scattered data.* (No. NPS53-79-003. Naval Postgraduate School Monterey CA.).

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

C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. Fernández del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant (2020), Nature **585**, 357–362.

T. Hastie, R. Tibshirani, and J. H. Friedman (2009), *The elements of statistical learning: Data mining, Inference, and Prediction*, 2nd ed., Springer series in statistics (Springer, New York, NY).

M. Hjorth-Jensen (2024), "Applied data analysis and machine learning," .

D. Jurafsky, and J. H. Martin (2024), *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*, 3rd ed., online manuscript released August 20, 2024.

N. Ketkar (2017), *Deep Learning with Python: A Hands-on Introduction* (Apress, Berkeley, CA).

D. P. Kingma, and J. Ba (2017), "Adam: A method for stochastic optimization," arXiv:1412.6980 [cs.LG].

Kreftregisteret (2024), "Nøkkeltall," .

B. Macukow (2016), in *Computer Information Systems and Industrial Management*, edited by K. Saeed, and W. Homenda (Springer International Publishing, Cham) pp. 3–14.

P. Martin (2022), *Linear Regression: An Introduction to Statistical Models* (SAGE Publications Ltd).

M. A. Nielsen (2015), *Neural Networks and Deep Learning*, accessed: 2023-10-31.

S. Raschka, Y. Liu, and M. V. (2022), *Machine Learning with PyTorch and Scikit-Learn* (Packt).

T. Schaul, I. Antonoglou, and D. Silver (2014), "Unit tests for stochastic optimization," arXiv:1312.6055 [cs.LG].

J. Shore, and R. Johnson (1981), IEEE Transactions on Information Theory **27** (4), 472–482.

W. N. Street, W. H. Wolberg, and O. L. Mangasarian (1993), in *Electronic imaging*.

W. Wolberg, O. Mangasarian, N. Street, and W. Street (1993), "Breast Cancer Wisconsin (Diagnostic)," UCI Machine Learning Repository.

A. Appendix: Source code

The code used for this project is available at https://github.com/GauteJ1/FYS-STK-projects. Instructions for running the code are located in `project_2/README.md`. All plots and results in this paper are easily reproducible in the Jupyter notebook files in this repository, by following the instructions mentioned above.