FYS-STK3155/4155
Project 2
Simplicity Versus Complexity: Traditional Statistical Methods and Neural Networks for Regression and Classification

Ben René Bjørsvik, Max Dahl, Egil Rolstad, and Asgeir Kråkenes

November 2024

Full project material available in our GitHub repository.

Contents

1 Abstract

Machine learning has become essential in advancing fields like healthcare, finance, and scientific research, where data-driven insights can lead to impactful solutions. Despite the success of complex neural networks in achieving high performance across various domains, challenges related to interpretability, computational efficiency, and energy consumption remain significant. Here, we demonstrate that traditional approaches can match or even exceed neural network performance in both regression and classification tasks. Using a Feed Forward Neural Network (FFNN) with gradient descent optimization techniques, we compare its performance against Ordinary Least Squares and Ridge regression on a regression problem that approximates the Franke function. For the classification task, we apply the FFNN and a logistic regression model to the Wisconsin Breast Cancer dataset. Our findings show that, in many cases, simpler methods are preferable for practical machine learning solutions, given their comparable accuracy, improved interpretability, and lower computational requirements. These results highlight the importance of model selection based on context.

2 Introduction

The scientific and public significance of neural networks was yet again demonstrated earlier this year when the Nobel Prize in Physics was awarded to renowned researchers contributing to the field [1]. Nevertheless, machine learning as a

whole faces challenges when it comes to interpretability, high energy consumption, and inherent biases [2]. Therefore, alternative approaches that can mitigate these issues and deliver comparable results are valued solutions. Building on this, we aim to demonstrate how less complex methods, which address some of these challenges more effectively, can achieve strong performance compared to neural networks in socially beneficial applications. Specifically, we here focus on identifying malignant tumors in the widely studied Wisconsin Breast Cancer dataset [3].

In this project, we implement a Feed Forward Neural Network (FFNN) and apply it to both a regression and a classification problem. The regression problem consists of attempting to approximate the Franke function, which is defined in 3.1. We compare the results from our FFNN model with the Ordinary Least Squares (OLS) and ridge regression models from [4]. This comparison shows that while the FFNN model achieves good results, it is out-performed by the much simpler OLS and ridge regression models.

The classification problem is based on the Wisonsin Breast Cancer Data Set, which is detailed in 3.9. This data set has been studied in several papers, such as [5]. In [5], the authors apply a deep convolutional neural network to the classification problem, achieving a test accuracy of 100%. Our FFNN model achieves similar results, after tuning of various hyper parameters using 5-fold cross-validation. We also implement a logistic regression classifier, and apply it to the same dataset. Similar to the regression problem, the FFNN model performs well, but is not able to outperform the simpler logistic regression model. Our findings imply that in many cases, simpler methods such as OLS and logistic regression are to be preferred over more complex models such as an FFNN.

We begin our paper by presenting various regression and classification models. Then, we give a discussion of different gradient descent methods that are used in the training of neural networks. This is followed by an overview of neural network training and architecture, as well as more detailed description of how we have implemented our FFNN model. Finally, we present our results, and give a discussion of the various models used.

3 Theory and methods

3.1 The Franke function

Let $\boldsymbol{z} \in \mathbb{R}^n$ be our response variable. In the first part of this report, $\boldsymbol{z}$ will represent the output of the Franke function, with and without stochastic noise.

The Franke function is defined as:

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

for $x, y \in [0,1]$. In the first part of the report where we use the Franke function to test our implementations, we will generate x and y by respectively drawing 100 samples uniformly over the interval $[0,1]$. We will then store every combination of x and y in a matrix $X \in \mathbb{R}^{10000\times 2}$.

3.2 OLS

Ordinary Least Squares (OLS) is a standard method for performing regression. In this model, we assume that the target vector $\boldsymbol{z}$ can be expressed as a linear combination of transformations of the input data $\mathbf{X}$, plus some stochastic noise. Mathematically, this assumption is formulated as:

$$\boldsymbol{z} = \beta_0 + \beta_1 h_1(\mathbf{X}) + \cdots + \beta_m h_m(\mathbf{X}) + \epsilon, \tag{1}$$

where $m \in \mathbb{N}$, $h_i : \mathbb{R}^{n\times p} \longrightarrow \mathbb{R}^{n\times 1}$ are transformations of the input features, and $\epsilon \sim N(\mathbf{0}, \Sigma)$ represents stochastically distributed noise with mean $\mathbf{0}$ and covariance matrix $\Sigma = \sigma^2\mathbf{I}$.

If $\tilde{\mathbf{X}} = h(\mathbf{X})$ represents our transformed design matrix, the parameters that minimize the mean squared error (MSE), which we define as $\|\boldsymbol{z} - \tilde{\mathbf{X}}\boldsymbol{\beta}\|^2$, can be determined using:

$$\hat{\boldsymbol{\beta}} = (\tilde{\mathbf{X}}^T\tilde{\mathbf{X}})^{-1}\tilde{\mathbf{X}}^T\boldsymbol{z}. \tag{2}$$

The model assumption in eq. (1), combined with the result in eq. (2), makes OLS a straightforward model to analyze and interpret. For instance, it can be shown that $\mathbb{E}(\hat{\boldsymbol{\beta}}) = \boldsymbol{\beta}$, meaning that $\hat{\boldsymbol{\beta}}$ is an unbiased estimator of $\boldsymbol{\beta}$. In this report, we transform our input data $\mathbf{X}$ into polynomials of varying degree d. For example, when $d = 2$, our design matrix $\tilde{\mathbf{X}}$ will have row vectors of the form $\begin{bmatrix}1 & x & y & x^2 & y^2 & xy\end{bmatrix}$. We utilize the `PolynomialFeatures` class from `sklearn.preprocessing` to efficiently obtain these polynomial transformations.

3.3 Ridge Regression

Ridge regression is similar to OLS, with the primary difference being the cost function we aim to minimize. In OLS, we use the MSE, whereas in Ridge regression, we want to find the parameters $\hat{\boldsymbol{\beta}}$ that minimize the function $\|\boldsymbol{z} - \tilde{\mathbf{X}}\boldsymbol{\beta}\|^2 + \lambda\|\boldsymbol{\beta}\|^2$ for some $\lambda \in \mathbb{R}$. Essentially, this cost function imposes a penalty on large (absolute) values of $\hat{\boldsymbol{\beta}}$. It can be shown that the parameters minimizing this penalized version of the MSE are given by:

$$\hat{\boldsymbol{\beta}} = (\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}} + \lambda\mathbf{I})^{-1}\tilde{\mathbf{X}}^\top\boldsymbol{z}, \tag{3}$$

where $\mathbf{I}$ is the identity matrix.

Our implementation of Ridge regression will be very similar to our implementation of OLS. The main difference is that $\hat{\boldsymbol{\beta}}$ will be found using eq. (3), and we will also vary the value of λ to study its impact on the MSE and R^2 (see Section 3.5). Additionally, it is crucial to scale the input data to prevent coefficients of large magnitude from dominating the penalty term [6]. In our code, we scale our input data by subtracting the mean from each column and dividing by the standard deviation. In other words, $\tilde{X}_{ij} = \frac{\tilde{X}_{ij} - \bar{X}_i}{s_{X_i}}$ for all $i \in [1, n]$, $j \in [1, m]$, where $\bar{X}_i$ and s_{X_i} represent the mean and standard deviation of the i-th column, respectively.

3.4 Logistic regression

We now shift our focus to binary classification and introduce another regression method: logistic regression. The logistic regression model uses the logistic function, also known as the sigmoid function (see Section 3.7.3), to model the relationship between input data and a binary outcome (0 or 1).

Let $\boldsymbol{X}$ denote the input data, where $\boldsymbol{X}$ is a matrix with n rows and p columns, representing n samples and p features. Let $\boldsymbol{\theta}$ be the vector of coefficients. The logistic function is mathematically defined as:

$$\sigma(\boldsymbol{z}) = \frac{1}{1 + e^{-\boldsymbol{z}}}, \quad \text{where} \quad \boldsymbol{z} = \theta_0 + \theta_1 X_1 + \theta_2 X_2 + \cdots + \theta_p X_p, \tag{4}$$

where $p \in \mathbb{N}$, and the output $\sigma(\boldsymbol{z})$ represents the probability of the binary outcome being 1, given $\boldsymbol{X}$.

In logistic regression, we use the following cost function:

$$C(\boldsymbol{\theta}) = -\frac{1}{n} \sum_{i=1}^{n} \left[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right], \tag{5}$$

where y_i is the target variable, and $\hat{y}_i$ is the predicted probability of the i'th observation being 1. There is no analytical expression for the minimizer of 5. Instead we use gradient descent to find the optimal parameters. Taking the gradient of (5) with respect to $\boldsymbol{\theta}$, we get:

$$\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) = \frac{1}{n} \mathbf{X}^\top \left(\hat{\boldsymbol{y}} - \mathbf{y} \right).$$

To prevent overfitting, we add an L^2-regularization term to the cost function:

$$C(\boldsymbol{\theta}) = -\frac{1}{n} \sum_{i=1}^{n} \left[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right] + \frac{\lambda}{2n} \sum_{j=1}^{p} \boldsymbol{\theta}_j^2, \tag{6}$$

where λ controls the regularization strength. The gradient then becomes:

$$\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) = \frac{1}{n}\mathbf{X}^\top (\hat{\boldsymbol{y}} - \mathbf{y}) + \frac{\lambda}{n}\boldsymbol{\theta}. \tag{7}$$

3.5 Performance metrics

3.5.1 MSE

The Mean Squared Error (MSE) is a standard way to measure model performance. Given a target $\boldsymbol{y}$ and a prediction $\hat{\boldsymbol{y}}$ we define the MSE as:

$$\sum_{i=1}^{n} (\boldsymbol{y}_i - \hat{\boldsymbol{y}}_i)^2. \tag{8}$$

3.5.2 R-squared

The R^2 score, also known as the coefficient of determination, measures how much of the variation in the data that our model is able to explain. It is defined as:

$$R^2 = 1 - \frac{\sum_{i=1}^{n} (\boldsymbol{y_i} - \hat{\boldsymbol{y}}_{\boldsymbol{i}})^2}{\sum_{i=1}^{n} (\boldsymbol{y_i} - \bar{y})^2}, \tag{9}$$

where $\bar{y}$ is the mean of the observed values.

R^2 ranges from 0 to 1, where 1 indicates a perfect fit and 0 indicates that the model performs no better than a fit equal to the average of the observed values.

3.5.3 Accuracy

For classification tasks, accuracy measures the proportion of correct predictions:

$$\textit{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \tag{10}$$

This metric is relevant for our logistic regression and neural network classification tasks.

3.6 Optimization

3.6.1 Gradient Descent

Gradient descent is an optimization algorithm used to minimize a cost function. By iterating over the data and updating the weights of a neural network or regression coefficients in the opposite direction of the gradient of the cost function with respect to the weights, scaled by a learning rate, we shift the weights in the direction of steepest descent. This way, the weights are adjusted to minimize the cost function.

Let $\mathbf{w}$ be the weights of our model, $F(\mathbf{w})$ be the cost function, and η denote the learning rate, a parameter that controls the step size for each optimization step. The weight update rule for gradient descent is given by:

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \eta \nabla F(\mathbf{w}_n). \tag{11}$$

Gradient descent begins with an initial guess $\mathbf{w}_0$ for the weights. For a sufficiently small η, it ensures that $F(\mathbf{w}_{n+1}) \leq F(\mathbf{w}_n)$. This means that the cost function decreases for each iteration. After a given number of iterations, also known as epochs, we eventually encounter a local minimum. The magnitude of the learning rate is here central. For a given scenario or data, this means that a learning rate that is too large may surpass the global minimum, while a learning rate that is too small may lead to slow convergence or converge to a local minimum.

3.6.2 Stochastic Gradient Descent

Stochastic Gradient Descent is a variation of standard Gradient Descent, and differs by how it uses training data for weight updates. To clarify, let us define three key terms.

An epoch has been referred to as an iteration, and refers more precisely to one complete pass through the entire training dataset. Generally, multiple epochs are required to train a model adequately. Then, a batch is a subset of the data [7]. The batch size can range from 1 to the size of the entire dataset, n. For efficiency, the dataset is often divided into smaller mini-batches, as implemented in part a) of our deliverables.

In standard Gradient Descent 3.6.1, weight updates occur once per epoch using the gradient of the cost function calculated over the entire dataset 11. For large datasets, this approach can be computationally intensive compared to the reward, as calculating the loss over every data point in each epoch is costly, and then not efficient if only earning one single update from it. To address this, SGD calculates the cost function and computes the gradient for only the mini-batch (subset) of all the data, and updates the weights for each mini-batch. For example, with $n = 100$, standard Gradient Descent would compute the cost function over the whole dataset and update weights once per epoch, while SGD with a batch size of 1 would update weights 100 times per epoch, and a mini-batch size of 10 would yield 10 updates per epoch, with the cost functions being calculated on the data included in each mini-batch.

By using frequent updates from random subsets, SGD can converge faster, although at the cost of noise in the optimization process. This noise can help avoid local minima and explore more of the solution space, potentially improving the final result.

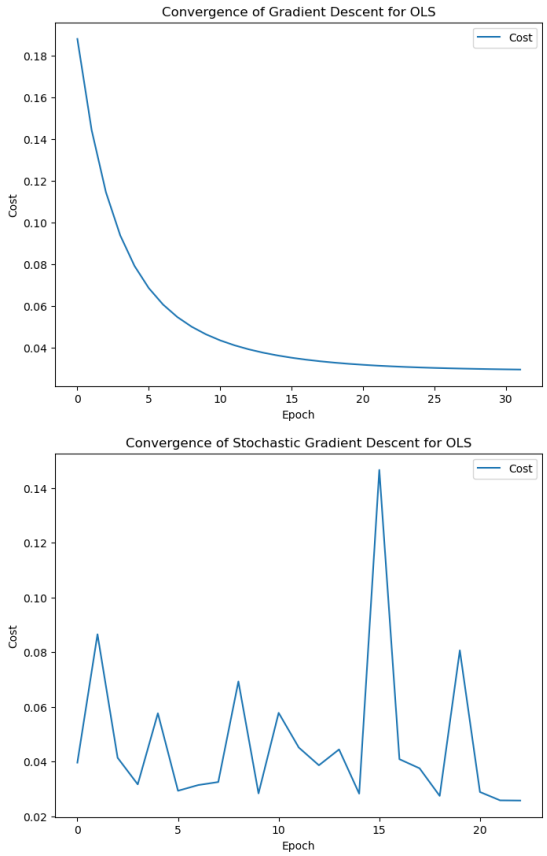

Figure 1: A simple comparison of convergence and learning stability between Gradient Descent and Stochastic Gradient Descent for independently tuned OLS and Ridge. Cost is plotted versus the number of epochs for the model to reach a given tolerance for the Franke Function.

3.6.3 Momentum

The Momentum technique is a method designed to accelerate the learning rate in training compared to standard Gradient Descent. By accumulating a moving average of the past gradients, Momentum makes the model maintain direction in the gradient's downhill path, reducing oscillations.

In Gradient Descent, each update to the model parameters is based on the current gradient 11. Momentum, on the other hand, uses the gradient's history by adding a fraction of the previous update to the current update. This adds some momentum in the direction of past and consistent gradients, which smoothens updates.

The momentum method update rule is defined as follows:

$$v_t = \gamma v_{t-1} + \eta \nabla w_t$$

$$w_{t+1} = w_t - v_t,$$

where v_t represents the accumulated velocity (momentum) at step t and γ is the momentum coefficient.

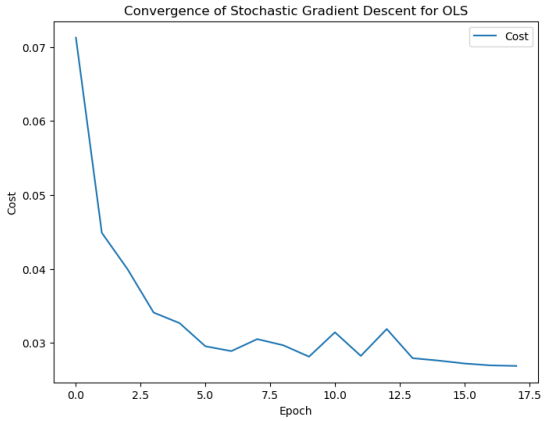

Figure 2: An example of a tuned SGD with momentum optimization process, illustrating a smoother learning process compared to the SGD plot in Figure 1. The cost is plotted versus the number of epochs for the model to reach a given tolerance for the Franke Function.

3.6.4 Adaptive Tuning of Learning Rate

Several optimization methods use adaptive tuning of the learning rate in the optimization process. By this, we mean that the step size is dynamically changed based on the gradients that are encountered during training. This approach means that our learning rate is adjusted from situation to situation, and can improve convergence and prevent oscillations or slowdowns in training.

The *Adagrad* (Adaptive Gradient) algorithm modifies the learning rate by accumulating the sum of squares of past gradients for each parameter. This scaling forces the learning rate to decrease for parameters with frequently large gradients, reducing the risk of too big steps, hence useful for sparse data. Adagrad's update rule is:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla w_t,$$

where G_t is the sum of squared gradients up to step t, η is the initial learning rate, and ϵ is a small constant to avoid division by zero.

RMSprop (Root Mean Square Propagation) is designed to improve Adagrad by using a moving average of squared gradients rather than a cumulative sum. This prevents the learning rate from becoming too small and allows for more stable training over time. RMSprop's update rule is:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla w_t$$

with

$$G_t = \alpha G_{t-1} + (1 - \alpha)(\nabla w_t)^2,$$

where α is the decay rate, typically around 0.9 - as used in a) - and G_t representing the exponentially weighted average of past squared gradients.

The *Adam* (Adaptive Moment Estimation) optimizer is designed to combine the advantages of both Momentum and RMSprop, and is a more comprehensive method than the two latter. It maintains both a moving average of gradients (like Momentum) and a moving average of squared gradients (like RMSprop). Additionally, a bias correction (denoted by the hats in the equations below) is applied to the estimates of these moments to account for the small number of observations on which the initial updates are based, as the denominator increases with each step t in the bias terms [7]. *Adam*'s update rule is:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

with

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla w_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla w_t)^2$$
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t},$$

where m_t and v_t are the moving averages of the gradient and squared gradient respectively, β_1 and β_2 are decay rates (typically 0.9 and 0.999), and $\hat{m}_t$ and $\hat{v}_t$ are the bias-corrected estimates.

3.7 Neural Networks

Neural networks have become fundamental tools in machine learning due to their ability to approximate complex functions across a wide range of applications. Similar to the regression methods discussed in Sections 3.2, 3.3 and 3.4, Neural networks aim to find optimal parameters that minimize a cost function, but with a more complex architecture allowing for non-linear relationships.

3.7.1 Feed-forward Neural Networks

The feed-forward neural network (FFNN) extends the concepts of linear models discussed in Section 3.2 by introducing multiple layers of transformations. Let $\mathbf{X} \in \mathbb{R}^{n \times p}$ be our input data matrix and $\mathbf{z} \in \mathbb{R}^n$ our response variable. Similar to eq. (1), we assume the target vector can be expressed as a series of non-linear transformations of the input data [7]:

$$\mathbf{z} = f_L(\mathbf{W}^L f_{L-1}(\mathbf{W}^{L-1} \cdots f_1(\mathbf{W}^1 \mathbf{X} + \mathbf{b}^1) + \mathbf{b}^{L-1}) + \mathbf{b}^L) + \boldsymbol{\epsilon} \tag{12}$$

where L is the number of layers, f_l are non-linear activation functions, $\mathbf{W}^l$ are weight matrices, and $\mathbf{b}^l$ are bias vectors. The error term $\boldsymbol{\epsilon}$ follows the same assumptions as in Section 3.2.
For each layer l, we compute [7]:

$$\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l \tag{13}$$

$$\mathbf{a}^l = f_l(\mathbf{z}^l) \tag{14}$$

where $\mathbf{a}^0 = \mathbf{X}$ represents the input layer, and $\mathbf{a}^L$ represents the final output.

3.7.2 Cost Functions and Regularization

Similar to Ridge regression in Section 3.3, we define a cost function with regularization:

$$C(\theta) = \frac{1}{n} \sum_{i=1}^{n} (\mathbf{z}_i - \mathbf{a}_i^L)^2 + \lambda \sum_{l=1}^{L} \|\mathbf{W}^l\|^2 \tag{15}$$

where $\theta = \{\mathbf{W}^l, \mathbf{b}^l\}_{l=1}^L$ represents all network parameters (the weights and biases), and λ serves the same regularization purpose as in 3. When $\lambda = 0$, this is reduced to the MSE.

3.7.3 Activation Functions

In Neural networks, activation functions introduce non-linearity into the model, enabling the network to learn complex patterns in the data.
Activation functions used in the hidden layers are:
The sigmoid function, mapping to the interval $(0, 1)$:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{16}$$

The Rectified Linear Unit (ReLU):

$$\text{ReLU}(z) = \max(0, z) \tag{17}$$

Leaky ReLU:

$$\text{LReLU}(z) = \begin{cases} z & \text{if } z > 0 \\ \delta z & \text{otherwise} \end{cases} \tag{18}$$

where δ is a small positive constant.
For the output layers, the choice depends on the task at hand:
In classification problems, the softmax function normalizes outputs to probabilities:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{19}$$

For regression problems, the identity function $f(z) = z$ provides unbounded linear output.

3.7.4 Training Algorithm

The optimization process follows the gradient descent framework described in Section 3.6, but requires a more sophisticated algorithm called *backpropagation*. For each layer l, we compute (where $\odot$ represents the Hadamard product)[7]:

1. Output Layer Error ($l = L$):

$$\boldsymbol{\delta}^L = \nabla_{\mathbf{a}^L} C \odot f_L'(\mathbf{z}^L) \tag{20}$$

2. Hidden Layer Error Propagation:

$$\boldsymbol{\delta}^l = ((\mathbf{W}^{l+1})^T \boldsymbol{\delta}^{l+1}) \odot f_l'(\mathbf{z}^l) \tag{21}$$

3. Parameter Updates:

$$\mathbf{W}^l \leftarrow \mathbf{W}^l - \eta \frac{\partial C}{\partial \mathbf{W}^l} \tag{22}$$

$$\mathbf{b}^l \leftarrow \mathbf{b}^l - \eta \frac{\partial C}{\partial \mathbf{b}^l} \tag{23}$$

where:

$$\frac{\partial C}{\partial \mathbf{W}^l} = \frac{1}{n} (\mathbf{a}^{l-1})^T \boldsymbol{\delta}^l + \lambda \mathbf{W}^l \tag{24}$$

$$\frac{\partial C}{\partial \mathbf{b}^l} = \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{\delta}_i^l \tag{25}$$

This extends the gradient descent method from Section 3.6.1 to handle multiple layers of parameters.

3.8 Our Implementation

Our implementation is a slightly modidified version of the FFNN class in [7]. This implementation is based on the backpropagation algorithm, using various stochactic gradient descent methods for optimizing the parameters.
The implementation consists of two main components:

1. **Network Architecture**
   We implement the feed-forward architecture that allows for varying network depths and layer sizes. Our implementation supports several activation and cost functions.

2. **Training Process**
   The training follows the backpropagation algorithm outlined above. For each training iteration:

   - Feed-forward computation stores both z and a matrices for each layer
   - Gradient computation using `Autograd` [8]
   - Weight updates using the chosen optimization method from Section 3.6

We initialize weights using a random standard normal distributions and set small bias values (0.01). Our implementation tracks both training and validation metrics during optimization, computing MSE and R^2 scores for regression tasks and accuracy for classification problems.

Our implementation balances the theoretical framework from Section 3 with practical considerations needed for reliable neural network training. Our code includes error handling and progress monitoring, making it suitable for both educational purposes and practical applications.

The test in the appendix 6.2 indicate that our code works as expected.

3.9 The Wisconsin Breast Cancer Data Set

Finally, we introduce the dataset on which we test our classification models: the Wisconsin Breast Cancer Data Set [3]. This dataset is well-known and easily available in Scikit-learn's built-in datasets [9]. It is used to distinguish between malignant and benign tumors, based on cell characteristics. The dataset contains characteristics measurements of cells sampled from multiple tumors, such as radius, texture, compactness and concavity. For each sample, the dataset also contains mean, standard error and mean of the three largest values for every characteristic.

In terms of size, the dataset consists of 569 samples, each with 30 numeric features, namely the cell characteristics described above. The target is binary, i.e. whether a tumor sample is malignant or benign, denoted 0 (n=212) and 1 (n=357), respectively. From the number of different target values we see that the dataset is fairly balanced.

The dataset poses a well-defined classification problem: based only on the cell characteristics from a given tumor, can we predict whether the tumor is malignant or benign? To answer this problem, we apply our FFNN model, and

compare it to a simpler (L^2-penalized) logistic regression model. Both the FFNN and logistic models are compared to the `MLPClassifier` and `LogisticRegression` models in the Scikit-learn library.

Training FFNN and logistic regression models for the problem at hand requires finding optimal hyperparameters. Before finding these parameters, we split the dataset into a training and test set. In what follows, we briefly explain how we searched for such optimal hyperparameters.

3.9.1 Tuning the FFNN model

Relevant hyperparameters for the FFNN model are the following:

- Number of hidden layers
- Number of nodes in hidden layers
- Activation function in hidden layers
- Scheduler for optimizing gradient
- Number of batches used in updating of weights
- Penalization parameter λ
- Learning rate η

We can tell that this creates a vast space of possible configurations of hyperparameters - a space too large to search extensively. Due to time and computational constraints, we decided to search through all combinations of the following hyperparameters for the FFNN model:

- Number of hidden layers: $[2, 5, 8]$
- Number of nodes in hidden layers: $[5, 10, 30]$
- Activation functions: [identity, sigmoid, softmax, RELU, LRELU]
- Number of batches: $[5, 10, 20]$
- Learning rate η: $[0.1, 0.01, 0.001]$
- Penalization λ: $[0.001, 0.01, 0.1, 1.0, 5.0]$

We can tell that all the hidden layers have the same number of nodes, meaning that we have only considered a certain kind of network structure. For all combinations of hyperparameters, we chose the *Adam* solver with decay rate parameteres $\beta_1 = \beta_2 = 0.99$. All the same hyperparameteres were used in the comparison with the Scikit-learn's `MLPClassification` model.

For reliable performance results of the different hyperparameter combinations, we performed 5-fold cross-validation. The cross-validation was performed *only*

on the training set to avoid data leakage - we do not want our choice of hyperparameters to be informed by the test set. We used Scikit-learn's `KFold` [9] function to easily retrieve fold indices in the validation process, and we used *accuracy*, as defined in eq. (10), as a measure of model performance.

3.9.2 Tuning the penalized logistic regression model

Since the penalized logistic regression model is a simpler one, it has considerably fewer hyperparameters to tune. We chose the following hyperparameters to search over:

- Number of batches used in gradient descent: $[5, 10, 20]$
- Learning rate η: $[0.1, 0.01, 0.001]$
- Penalization λ: $[0, 0.001, 0.01, 0.1, 1.0, 5.0]$

For our own implementation, we used the *Constant* solver to optimize the gradient. In Scikit-learn's `LogisticRegression` implementation, the *Constant* solver is not an option, which made us use their default solver *Limited-memory BFGS* (lbfgs) [10].

Similar to the FFNN hyperparameter search, we performed 5-fold cross-validation on the training set for reliable performance results. Also here we used *accuracy* as a measure of model performance.

3.9.3 Scaling of features

As mentioned, we split the data into training and test set before searching for optimal hyperparameters. Since we are assessing the use of regularization parameter λ, we scaled the numerical cell characteristic features in both training and test set, separately.

4 Results

4.1 Testing regression implementations

To assess our FFNN implementation, we applied it to a regression problem using the Franke function as target. After varying the degree of the polynomial transformation, we found $p = 8$ to be the optimal choice. In the appendix 6.1, we can see the results of applying our FFNN model to the Franke function with varying values for the learning rate and regularization parameter. Here we have used the Sigmoid funciton as activation function. We see that $\eta = 0.01$ and $\lambda = 0$ are the optimal choices both in terms of MSE and R^2 score, achieving an MSE of 0.004. We have not seen indications of our FFNN model overfitting the Franke function data. It is therefore reasonable that the model performs best without regularization.

We have also studied the effect of changing the activation function in the hidden layers of the FFNN model. The alternatives we considered were the RELU and Leaky RELU functions. Additionally, we studied how our FFNN implementation performed compared to the `MPLRegressor` class in `sklearn.neural_network`, as well as our own OLS and Ridge regression implementation from our previous project [4]. The results of these comparisons, presented in Table 3 6.1, show that changing the activation function in the hidden layers leads to a significant drop in performance. This may indicate that the RELU family of activation functions are not well-suited for this specific problem or network architecture, possibly resulting in issues such as vanishing gradients or inactive neurons.

Table 3 in 6.1 shows that our optimal FFNN model performs very well on the Franke function data, even marginally outperforming the `MLPRegressor` class from Scikit-Learn. This being said, neither of these models perform as well as OLS and Ridge regression. Figure 3 shows that both OLS and FFNN capture much of the variability in the Franke function data.

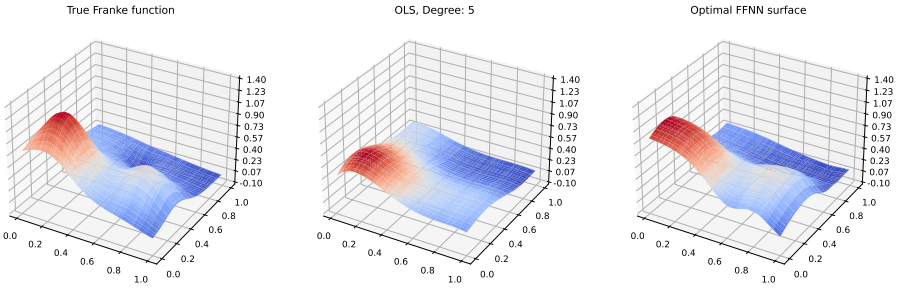

Figure 3: Regression surfaces obtained with OLS and FFNN, compared to the true Franke function surface

4.2 Application to the Wisconsin Breast Cancer dataset

We applied our Neural Network and Logistic regression implementations to the Wisconsin Breast Cancer dataset. For clarity in what follows, we repeat the some key properties of the dataset:

- It has a total of 569 samples, each with 30 numeric features
- Upon splitting in train/test set, we obtain a training set of size 426 and an a test set of 143 samples
- We standard-scaled the features

We now proceed to the tuning and comparison of the different models.

4.2.1 Tuning the Neural Network model

We chose six different hyperparameters to tune for, which gave a total of 2025 different combinations. After 5-fold cross-validation on the training set, we obtained the following results for the different hyperparameters:

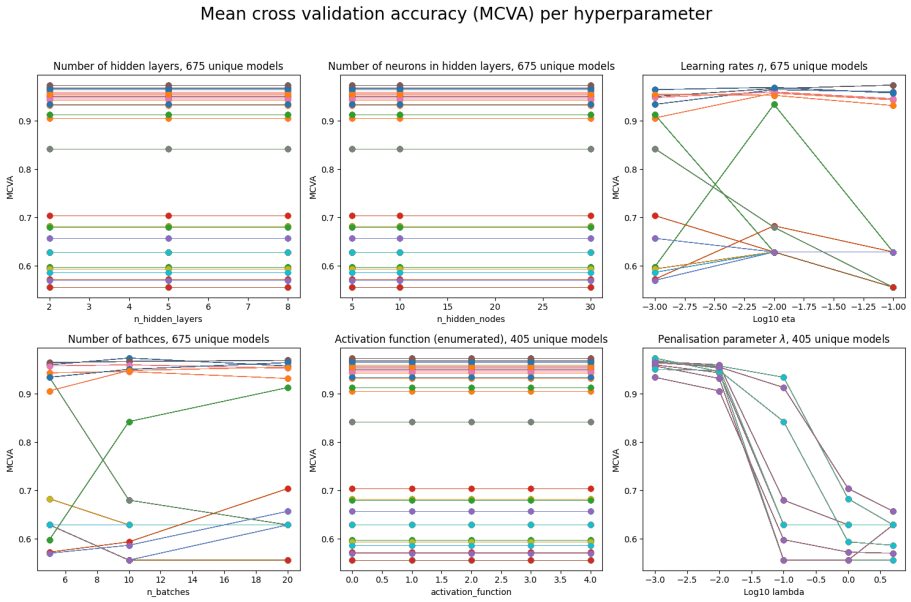

Figure 4

In the figure above, the lines indicate a given combination for all other hyperparameters than the one being evaluated. For example, in the lower right plot we vary over the penalization parameter λ. Each line connected by dots is a model whose hyperparameters are fixed, except that of λ. In this way we can assess which hyperparameter influences the mean cross-validation accuracy (MCVA).

We see from fig. 4 that the number of hidden layers, number of neurons in hidden layers and the activation functions has close to no influence on the MCVA. This means that when assessing the models which vary only over these particular hyperparamters, we see no change. We thus limit the following assessment of hyperparameters to the learning rate η, number of batches and penalization parameter λ:

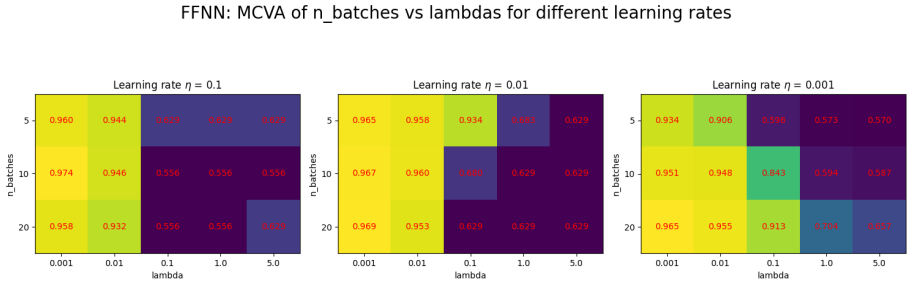

Figure 5

From fig. 5 we can tell that a learning rate $\eta = 0.01$, number of batches of 20 and $\lambda = 0.001$ performed best in the cross-validation setting. We now compare with models with the same hyperparameters which are trained on the full training set and evaluated on the test set:

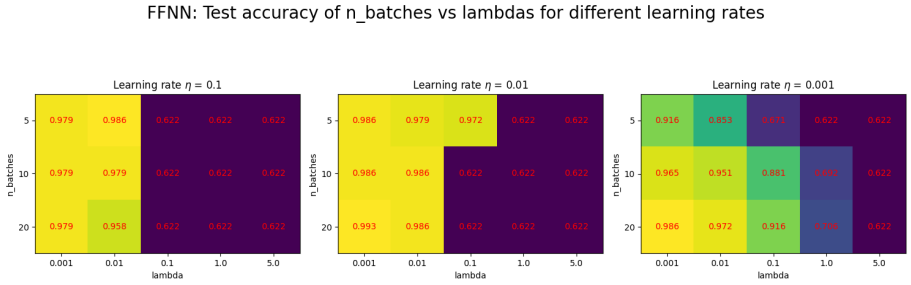

Figure 6

We see that the same values for η, number of batches and λ as in the cross-validation setting give the best accuracy of 0.993. We also see that for increasing penalization λ, the accuracies tends to decrease. Using Scikit-learn's `MLPClassifier` with the same parameters and training and test set gives the following results:

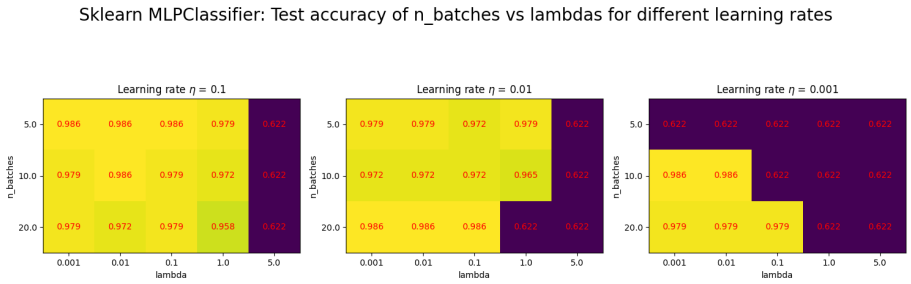

Figure 7

Comparing the performance of our own implementation to that of Scikit-learn, we see the same trend of descreasing accuracy with increasing penalization λ. The maximum accuracy of Scikit-learn's implementation is 0.986, slightly below the highest accuracy of our own model.

4.2.2 Tuning the Logistic Regression model

We searched the hyperparameters for the logistic regression model similar to how we did for the FFNN model. After 5-fold cross-validation on the training set, we get the following results:

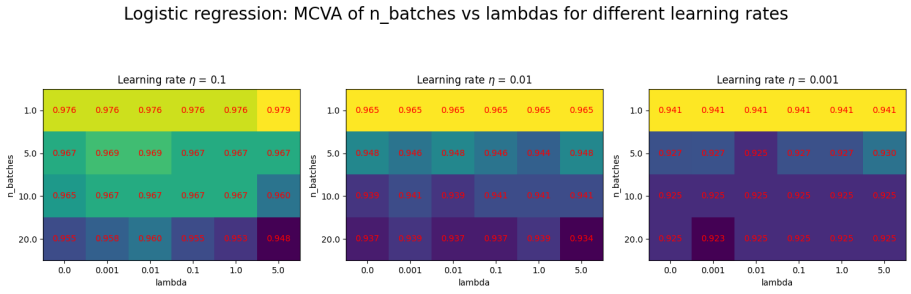

Figure 8

Here it seems like the the optimal choice of number of batches is 1, $eta = 0.1$ and $\lambda = 5.0$. Training the models on the full training set and evaluating on the test set gave the following results:

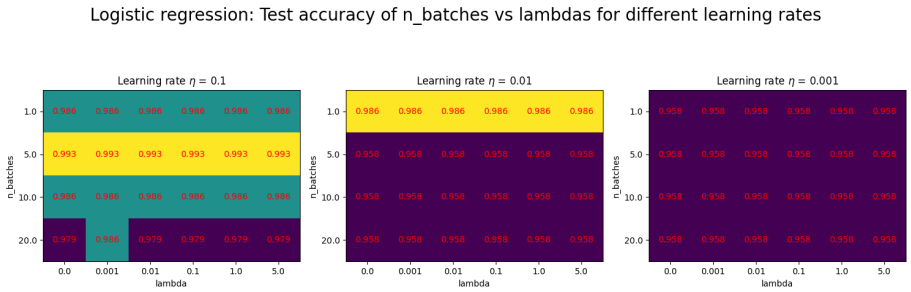

Figure 9

For the total dataset, it is clear that the optimal number of batches is 5 and $\eta = 0.1$, conflicting with what we would expect from the cross-validation results. Training and testing on the same hyperparameters with Scikit-learn's implementation gives the following:

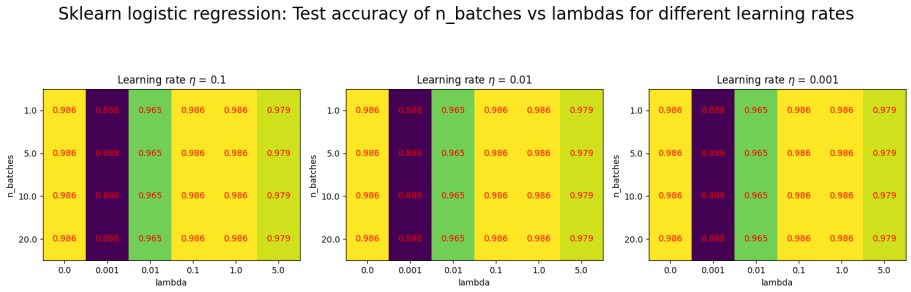

Figure 10

We see that our own implementation is superior to that of Scikit-learn in terms of maximum accuracy.

5 Discussion and conclusion

In the regression task using the Franke function, our FFNN implementation performed well with an MSE of 0.004. However, both OLS and Ridge regression demonstrated superior performance with simpler architectures, suggesting that the additional complexity of neural networks may not always justify their computational cost in regression problems.

For the Wisconsin Breast Cancer classification task, our FFNN achieved a respectable accuracy of 0.993, slightly exceeding Scikit-learn's MLPClassifier

(0.986). However, our logistic regression model demonstrated comparable performance with significantly less computational cost. Notably, we observed that several FFNN hyperparameters, including network depth and width, showed minimal impact on model performance.

When comparing FFNN with logistic regression, our results challenge the assumption that more complex models yield better results. The logistic regression model achieves similar performance while offering several advantages, including i) better interpretability of model parameters, ii) lower computational requirements iii) fewer hyperparameters to tune, and iv) more stable training processes.

Simultaneously, our analysis posses some limitations. When tuning our FFNN for the classification problem on the Wisconsin breast cancer data set, our search for the right tuning of our FFNN was narrowed down by assumptions we made. Specifically, these assumptions were mainly based on the trend we observed in fig. 4, but also on assumptions we did not further examine, for instance the consistent choice of *Adam* as our scheduler. We could also have investigated the FFNN without regularization for classification.

Additionally, comparing our implementations to the Scikit-learn library, this logistic regression comparison could have been more valid. This is because the Scikit-learn library lacks the option for a constant scheduler, essentially resulting in a comparison of two different methods. In addition to this, although we are able to conclude on our findings, one can question whether our results, which are specific to two particular problems, can be generalized broadly.

Our findings demonstrate that the most sophisticated model is not always the best choice. Particularly in medical applications like cancer detection, where interpretability and reliability are crucial, simpler models can represent a more responsible choice. Traditional statistical methods can match or exceed neural network performance while maintaining better interpretability and computational efficiency, suggesting that model selection should carefully consider the specific requirements and constraints of each application.

6 Appendix

6.1 Regression implementation

Eta	Lambda	MSE
0.010000	0.000000	0.004122
0.001000	0.000000	0.006741
0.000100	0.000000	0.015578
0.001000	0.010000	0.015707
0.000100	0.010000	0.028144
0.001000	0.100000	0.042575
0.010000	0.010000	0.044413
0.010000	0.100000	0.076330
0.000100	0.100000	0.086743

Table 1: MSE of FFNN model with varying hyper parameters

Eta	Lambda	R2
0.010000	0.000000	0.946798
0.000100	0.000000	0.798930
0.001000	0.010000	0.797264
0.000100	0.010000	0.636746
0.000100	0.100000	0.496374
0.001000	0.100000	0.450478
0.001000	0.000000	0.450478
0.010000	0.010000	0.426763
0.010000	0.100000	-4.621323

Table 2: R2 score of FFNN model with varying hyper parameters

Method	MSE
OLS	0.001875
Ridge	0.003015
Optimal FFNN	0.003889
Skleran MLP	0.005566
RELU	0.432025
LRELU	1.367926

Table 3: Performance of various regression models

6.2 Testing FFNN

Testing FFNN Implementation

```
#Testing FFNN code
test_x = np.hstack((np.zeros(50), np.ones(50))).reshape(-1, 1)
test_y = test_x
test_structure = (1, 20, 20, 1)
test_scheduler = func.Constant(eta = 0.01)
test_FFNN = func.FFNN(dimensions=test_structure,
    hidden_func=func.sigmoid, output_func=func.identity,
    cost_func=func.CostOLS)
test_scores = test_FFNN.fit(test_x, test_y, test_scheduler, epochs=2000,
    tol = 0)
final_train_error = test_scores["train_errors"][-1]
print(np.abs(final_train_error) < 10e-8)
```

```
True
```

7 Bibliography

References

[1] *The Nobel Prize in Physics 2024 - Press Release*. The Nobel Prize. 2024. URL: https://www.nobelprize.org/prizes/physics/2024/press-release/ (visited on 11/08/2024).

[2] *News Article on OUS Research: Researchers' Recent Achievements*. Oslo University Hospital - Research Department. 2023. URL: https://www.ous-research.no/home/ous/news/23168 (visited on 11/08/2023).

[3] William H. Wolberg et al. *Breast Cancer Wisconsin (Diagnostic)*. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5DW2B. 1993.

[4] B. Bjørsvik et al. "Fitting Heights: Regression Models for Terrain Prediction". In: *FYS-STK3155/4155 Fall 2024 Project 1* (Oct. 2024).

[5] Patnala S. R. Chandra Murty et al. "Integrative hybrid deep learning for enhanced breast cancer diagnosis: leveraging the Wisconsin Breast Cancer Database and the CBIS-DDSM dataset". In: *Scientific Reports* 14.1 (Nov. 2024), p. 26287. ISSN: 2045-2322. DOI: 10.1038/s41598-024-74305-8. URL: https://doi.org/10.1038/s41598-024-74305-8.

[6] Robert Tibshirani Jerome Friedman Trevor Hastie. *The Elements of Statistical Learning*. New York, NY, US: Springer, 2017. Chap. Model Selection and the Bias–Variance Tradeof, pp. 37–38.

[7] Morten Hjorth-Jensen. *Applied Data Analysis and Machine Learning*. 2024. Chap. Stochastic Gradient Descent, Batches and mini-batches, Momentum based GD, Automatic differentiation, Backpropagation. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html.

[8] Maclaurin et al. "Autograd: Effortless Gradients in Numpy". In: *AutoML Workshop Short Paper* (2015), pp. 1–9. URL: https://indico.ijclab.in2p3.fr/event/2914/contributions/6483/subcontributions/180/attachments/6060/7185/automl-short.pdf.

[9] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[10] Dong C. Liu and Jorge Nocedal. "On the limited memory BFGS method for large scale optimization". In: *Mathematical Programming* 45.1–3 (Aug. 1989), pp. 503–528. ISSN: 1436-4646. DOI: 10.1007/bf01589116. URL: http://dx.doi.org/10.1007/BF01589116.