FYS-STK 4155 Project 2

Aleksander N. Sekkelsten[1]
[1]Faculty of Mathematics and Natural Sciences, University of Oslo

Abstract

This project examines the application of neural networks and logistic regression for both regression and classification tasks, employing three distinct approaches: a manually constructed neural network with a manually constructed backpropagation algorithm, a manually coded neural network using Zygote's automatic differentiation, and a model implemented using the Flux library in Julia. Various optimization algorithms-Gradient Descent, Stochastic Gradient Descent, Momentum, AdaGrad, RMSProp, and ADAM-were also implemented and compared. Regression tasks were benchmarked using Franke's function and a polynomial function, while classification was evaluated using the Wisconsin Breast Cancer and MNIST datasets. The results reveal that the manually coded network with backpropagation achieved superior performance, notably attaining an accuracy of 91.39% on the MNIST dataset compared to 85.43% with Zygote and 79.46% with Flux. This project highlights the significant impact of gradient computation methods on model performance, offering valuable insights into the strengths and limitations of each approach for various machine learning tasks.

1 Introduction

In the era of big data and artificial intelligence, machine learning has become an indispensable tool for analyzing complex datasets and developing predictive models. Neural networks, renowned for their ability to model non-linear relationships, have gained prominence across diverse applications, from image recognition to natural language processing. Meanwhile, logistic regression remains a staple for interpretable binary and multiclass classification tasks due to its simplicity and effectiveness.

Implementing neural networks and logistic regression models can be approached in various ways, each presenting trade-offs between performance, computational efficiency, and ease of development. Manual implementation offers granular control and the potential for optimization tailored to specific tasks but demands significant effort and expertise. Automatic differentiation tools like Zygote facilitate gradient computations, streamlining the development process while potentially introducing computational overhead. High-level libraries such as Flux in Julia provide abstractions that simplify model building and training but may not offer the same level of performance optimization achievable through manual coding.

This project investigates the impact of these different implementation methods on the performance of neural networks and logistic regression models across both regression and classification tasks. Specifically, we employ three approaches:

- Manually constructed neural networks with custom backpropagation.
- Manually coded neural networks utilizing Zygote's automatic differentiation.
- Models implemented using the Flux library in Julia.

We apply these methods to a spectrum of problems, starting with the regression task of fitting a one-dimensional polynomial. Here, we compare various optimization algorithms—ADAM, RMSProp, SGD, GD, Adagrad, and Momentum—without the use of Flux or an FFNN, to assess their performance in a fundamental setting. We then extend our analysis to the more complex Franke's function, employing both linear regression and FFNNs, and examine the role of different gradient computation methods. For both regression problems, Zygote gradients were compared to analytical ones from linear regression. Ridge and OLS regression are also compared.

In the classification domain, we utilize the Wisconsin Breast Cancer dataset and the MNIST handwritten digits dataset. By implementing logistic regression and FFNN models across the three approaches, we aim to evaluate how the choice of optimization algorithms, gradient computation methods, and network architectures affects model accuracy and efficiency.

This project aims to provide practical insights into the considerations involved in implementing neural networks and logistic regression in Julia. By emphasizing the significance of gradient computation methods, optimization techniques, and regularization parameters, we hope to guide practitioners in selecting the most appropriate methods for their specific machine learning tasks.

The remainder of this report is organized as follows: Section 2 details the methodologies and experimental setups, Section 3 presents the results and analyses, Section 4 offers a comprehensive discussion of the findings, and Section 5 concludes with the main insights and suggestions for future work.

2 Method

2.1 Franke's Function and Data Generation

Franke's function, a well-known test function for surface approximation problems, is used to generate synthetic data. The function is defined as:

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

We sample the function over a grid of x and y values, each ranging from 0 to 1, with 100 points in each dimension, creating a 100×100 grid. The resulting target values, Z, are computed by evaluating Franke's function at each grid point. To simulate real-world conditions, we add Gaussian noise with a standard deviation of $\sigma = 0.02$

Linear Regression

For the regression task, **Ordinary Least Squares (OLS)** regression was implemented as a baseline model. OLS seeks to minimize the residual sum of squares between the observed and predicted values by solving the following equation:

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Where $\mathbf{X}$ represents the design matrix of input features, $\mathbf{y}$ represents the output vector, and $\hat{\beta}$ are the estimated coefficients.

Ridge Regression

- **Ridge Regression** introduces an ℓ_2-norm penalty to the OLS loss function, which reduces overfitting by shrinking the regression coefficients:

$$\hat{\beta}_{\text{ridge}} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$$

Where λ is a regularization parameter and $\mathbf{I}$ is the identity matrix.

In this project, both methods were implemented, both with their analytical gradients, and numerically calculated ones. This was done on Franke's function.

2.2 Logistic Regression

Logistic Regression is a statistical method used primarily for classification tasks. It estimates the probability that a given input belongs to a particular class, using a logistic (sigmoid) or softmax function to map predictions to a range between 0 and 1. This method can be applied to both binary and multiclass classification problems.

For binary classification, logistic regression uses the **sigmoid function**, which transforms the linear combination of input features into a probability:

$$\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}}$$

where $z = \mathbf{w}^\top\mathbf{x} + b$ represents the linear combination of input features $\mathbf{x}$, weights $\mathbf{w}$, and bias b. The output $\hat{y}$ is the predicted probability that the input belongs to the positive class, making logistic regression a suitable model for binary classification tasks.

For **multiclass classification**, logistic regression is extended using the **softmax function**, which generalizes the sigmoid to multiple classes by mapping a vector of scores into a discrete probability distribution:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \quad \text{for } i = 1, 2, \ldots, K$$

where K is the total number of classes. The softmax function produces a probability for each class, with the highest probability typically selected as the predicted class.

Logistic regression is trained using a **cross-entropy loss function**, which measures the difference between the predicted probabilities and the true class labels. For binary classification, the binary cross-entropy loss is applied, while multiclass classification uses the categorical cross-entropy loss:

$$L = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)] \quad \text{(binary)}$$

$$L = -\frac{1}{N} \sum_{n=1}^{K} \sum_{i=1}^{N} y_i^{(n)} \log(\hat{y}_i^{(n)}) \quad \text{(multiclass)}$$

where K is the number of classes, N is the number of samples, $y^{(n)}$ is the true label, and $\hat{y}^{(n)}$ is the predicted probability for class n. This loss function encourages the model to maximize the probability of the correct class, resulting in a model that can classify new data with high accuracy.

Logistic regression is widely used due to its simplicity, interpretability, and efficiency, making it a strong choice for baseline models in classification tasks. It serves as the foundation for more complex models and can be extended to capture intricate relationships in data.

2.3 Feed-Forward Neural Network

A Feed-Forward Neural Network (FFNN) is an extension of logistic regression models, capable of modeling more complex relationships by introducing hidden layers and non-linear activation functions:

- Input Layer: Consists of neurons representing each feature in the input data.
- Hidden Layers: One or more layers of neurons that apply non-linear transformations to the inputs. These layers enable the network to learn complex patterns.
- Output Layer: Produces the final predictions.

FFNNs are often used for regression tasks in addition to classification tasks due to their generalizability.

2.3.1 Activation Functions:

Activation functions introduce non-linearity into the network, allowing it to learn non-linear relationships between inputs and outputs. In addition to Sigmoid described above, **ReLU** is widely used in hidden layers due to its computational efficiency:

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

Backpropagation

We implemented **backpropagation** from scratch to compute the gradients and update the weights of the neural network. The key steps are:

1. **Forward Pass**: The input data propagates through the network layer by layer, where each neuron computes:

   $$z = \mathbf{w}^T \mathbf{x} + b$$

   Here, $\mathbf{x}$ represents the input, $\mathbf{w}$ are the weights, and b is the bias term. The output is passed through the activation function to produce the final prediction.

2. **Loss Calculation**: For regression, the **mean squared error (MSE)** was used as the loss function:

$$L = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

For classification, we used the **binary cross-entropy**, and **cross-entropy** loss as described earlier.

3. **Backward Pass (Backpropagation)**: The gradients of the loss function with respect to the weights and biases were computed using the chain rule. Specifically:

For each layer l, the gradient of the loss with respect to the weights $\mathbf{W}$ and biases b is:

$$\frac{\partial L}{\partial \mathbf{W}^l} = \delta^l \cdot \mathbf{a}^{l-1}$$

$$\frac{\partial L}{\partial b^l} = \delta^l$$

Where δ^l is the error term for layer l, and $\mathbf{a}^{l-1}$ is the activation from the previous layer. The error term δ^l is computed as:

$$\delta^l = \frac{\partial L}{\partial z^l} \cdot \sigma'(z^l)$$

For the output layer in regression, $\sigma'(z) = 1$, and for classification, $\sigma'(z)$ is the derivative of the sigmoid function.

4. **Gradient Descent** : The network weights were updated using **Gradient Descent**, **Stochastic Gradient Descent**, **AdaGrad**, **Momentum**, **RMSProp** and **ADAM**. Gradient Descent is shown here, and is the core of all methods:

$$\mathbf{W}^l = \mathbf{W}^l - \eta \frac{\partial L}{\partial \mathbf{W}^l}$$

Where η is the learning rate.

2.4 Wisconsin Breast Cancer Dataset

The *Wisconsin Breast Cancer Dataset* is a widely used dataset for binary classification tasks in machine learning, particularly for medical diagnosis problems. It contains information about breast cancer biopsies, where each sample is labeled as either benign or malignant, representing the absence or presence of cancer, respectively. The dataset was obtained from the University of Wisconsin Hospitals and contains 569 samples with 33 features.

The features include various characteristics of cell nuclei present in digitized images of fine needle aspirates of breast masses. These features are computed from real-world clinical data and include measurements such as:

- *Radius*: Mean of distances from the center to points on the perimeter.
- *Texture*: Standard deviation of gray-scale values.
- *Perimeter*, *Area*, *Smoothness*, and *Compactness*.
- *Concavity*, *Symmetry*, and *Fractal dimension*.

Each feature was normalized and scaled prior to feeding into the machine learning models. For this project, we used the **Wisconsin Breast Cancer Dataset** to train and evaluate both the feed-forward neural network (FFNN), and logistic regression models. The dataset was split into training and test sets, with stratification ensuring that the class distribution was preserved in both sets. The main metric for evaluating the performance on this dataset was accuracy and a confusion matrix.

2.5 MNIST Dataset

The *MNIST* (Modified National Institute of Standards and Technology) dataset is a well-known benchmark in the field of machine learning and computer vision, particularly for evaluating models on image classification tasks. It contains 70,000 grayscale images of handwritten digits, where each image is a 28x28 pixel matrix, and each pixel represents a gray-scale value between 0 and 255. The goal of the task is to correctly classify each image into one of 10 digit classes (0 through 9).

The dataset is pre-divided into a training set and a test set:

- 60,000 training images.
- 10,000 test images.

Each image in the dataset corresponds to a single handwritten digit. In this project, we utilized the MNIST dataset to evaluate the performance of our feed-forward neural network (FFNN), as well as logistic regression in a multi-class classification setting.

The softmax activation function was applied to the output layer to generate a probability distribution over the 10 possible classes. The model's performance was evaluated based on the accuracy score and the confusion matrix, with particular focus on identifying patterns of misclassification across different digit classes.

2.6 Flux.jl

Flux.jl is a flexible and efficient machine learning library written in the Julia programming language. It provides a high-level, user-friendly interface for building and training various types of neural networks, while also giving users the flexibility to work directly with Julia's advanced numerical computing capabilities.

Flux emphasizes a minimalistic design, allowing users to define neural networks using straightforward Julia code.

Key features of *Flux.jl* include:

- Built-in utilities for automatic differentiation via Julia's *Zygote* package, enabling efficient gradient computation for backpropagation.
- Simple integration with other Julia packages for optimization, data handling, and visualization.
- A wide range of optimizers such as stochastic gradient descent (SGD), Adam, and RMSprop, among others.

In this project, Flux.jl was used to implement and train custom feed-forward neural networks (FFNN) for both regression and classification tasks. Its performance was compared to the manually constructed Backprapagation and FFNN. Flux's ability to leverage Julia's performance also contributed to efficient model training, particularly for large datasets like MNIST.

2.7 Zygote.jl and Automatic Differentiation

Zygote.jl is a state-of-the-art automatic differentiation (AD) library in Julia, and it serves as the backbone for gradient computation in the *Flux.jl* ecosystem.

Zygote uses a technique called *source-to-source transformation*, which transforms the code itself to generate gradients. This enables differentiation through any Julia function, including custom loss functions, control flow, and even recursive functions. Zygote supports both forward-mode and reverse-mode differentiation, but it primarily focuses on reverse-mode differentiation, which is more efficient for deep learning where the number of parameters is typically much larger than the number of outputs.

The key advantages of using *Zygote.jl* include:

- Gradients can be calculated through complex operations, including loops, conditionals, and recursion, without any manual intervention.
- By leveraging Julia's just-in-time (JIT) compilation and native performance, Zygote provides fast and efficient gradient computation.
- Zygote integrates smoothly with other Julia libraries and is the default AD engine in Flux.jl, enabling end-to-end neural network training and optimization.

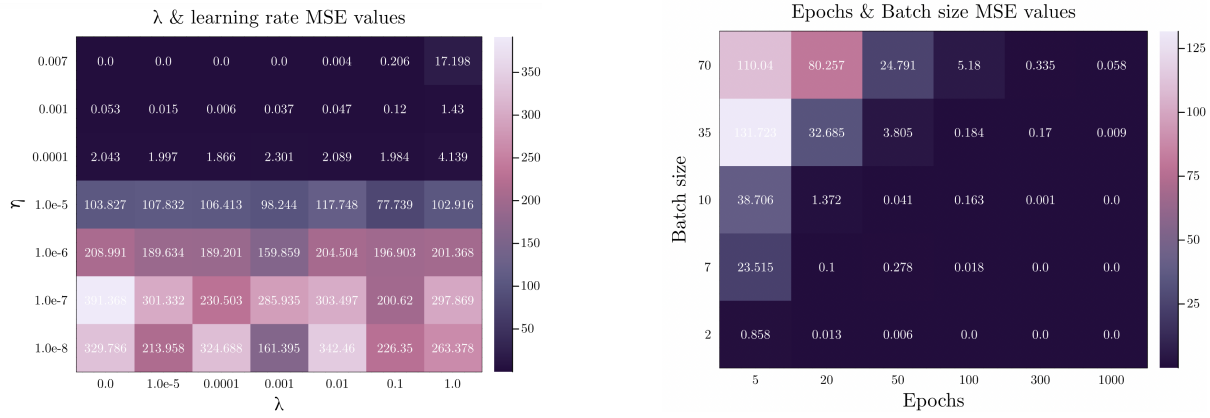
In this project, Zygote was used to compute the gradients of the loss functions with respect to the neural network parameters. This was compared to a pure Flux-written code, and the manual backpropagation algortihm.

3 Results

The results acquired in this project are three-fold. First, we applied the various optimiazation methods, with a Zygote and analytical gradient to both Ridge and OLS linear regression on a polynomial. For Franke's function(a 2d linear regression task), we incorporated a FFNN as well. Then we applied both the FFNN, and logistic regression to a binary classification task (The Wisconsin Dataset). And lastly, a multiclass classification problem (The MNIST dataset).

3.1 One dimensional polynomial

The models are asked to fit a second order polynomial. They will more precicely, take in one input x, and return the polynomial coefficients, which are a_0, a_1, a_2 in second order. After tuning each model with different parameters, ADAM produced the best results with a final loss of around 10^{-16}. After that, Momentum, AdaGrad, RMSProp and SGD all produced very similar results with a loss of around $10^{-6} - 10^{-9}$. GD was slightly more ineffective, with a final loss of around 10^{-4}. These performance differences are very minimal, and we will discuss the models further in the two dimensional, more complex case. However, it is worth noting that the more technical models like ADAM and RMSProp required more hyperparameter fine-tuning to acquire good results, whereas SGD and GD gave consistent results.



(a) MSE values for various values for the learning rate η and ridge penalty value λ, with 300 epochs and batch size of 20

(b) MSE values for various number of epochs and batch sizes with $\eta = 0.001$, and λ=0.001

Figure 1: Hyperparameter turning for SGD on a one dimensional polynomial

The hyperparameter tuning done for SGD in particlar is shown in figure 1. First, λ-values (corresponding to the ridge regression penalty term) were chosen in the range from 0 (OLS regression) to 1.0. Then several η values (learning rates) were chosen, and it was found that learning rates that surpassed 0.007 diverge quickly due to the exploding gradient problem.

In figure 1a, it is easy to see that the learning rate is a crucial parameter, with a too small one corresponding to very static learning that would require an enormous amount of epochs. If too large, the gradients can quickly diverge however, even for a simple problem such as this. The λ values also show differences however, the MSE is larger if too large, such as 1. And it is lowest for 0.0001-0.001, hence outperforming OLS regression.

In figure 1b, it is shown that if the number of epochs is low, and the batch size is large (corresponding to few batches), the training algortihm doesn't have time to converge properly. When the epochs go over 50 or more, the number of batches does not impact the results very significantly. With performance and computational cost in mind, training for 300 epochs and a batch size of around 10 would yield effective training and results for this task.

For other methods however, like ADAM, a learning rate of 0.001 slowed down the learning dramatically, and converged better for 0.1, which diverged for SGD. The parameters, should in other words be tuned individually for different methods. The results presented here account for SGD.

3.2 Franke's function

This function is more complex. We fit the function using linear regression and an FFNN. In addition to the analytical gradient, a Zygote gradient was applied to linear regression, inside a FFNN code, and a Flux model.

The models were asked to fit the function with a fifth-order 2 dimensional polynomial. The input size is x, y, and the output is the polynomial parameters which is of size 21 now.

The results of Linear regression is shown in table 1 and table 2 for both analytical and zygote gradients.

Method	**Analytical Gradients**	**Zygote Gradients**
GD	0.7598	0.7570
SGD	0.0236	0.0180
Momentum	0.0109	0.0108
Adagrad	0.0103	0.0100
RMSprop	0.0084	0.0084
Adam	0.0045	0.0045

Table 1: $\lambda = 0$

Method	**Analytical Gradients**	**Zygote Gradients**
GD	0.6829	0.7419
SGD	0.0172	0.0174
Momentum	0.0124	0.0128
Adagrad	0.0119	0.0120
RMSprop	0.0123	0.0124
Adam	0.0174	0.0174

Table 2: $\lambda = 1 \times 10^{-3}$

It is clear that the difference between the two gradient calculations is very small, but the difference between the models are larger. Since the linear regression task is quite complex, simple models such as GD and SGD have a poor performance here, espescially compared to some of the more advanced methods. These require more hyperparameter fine tuning, but were set to standard values for simplicity. Apart from GD and SGD, Ridge regression consistently performs worse than OLS, suggesting that the penalty term λ has minimal impact in mitigating overfitting. However, Ridge does show improvement for GD and SGD, indicating that simpler optimization methods may be more susceptible to overfitting the data compared to more advanced techniques.

Introducing now the FFNN, the model had an input dimension of 2 (x,y), one hidden layer with 30 nodes and a sigmoid activation function, and then the output layer without an activation function. This is necessary to allow complete flexibility in estimating the polynomial parameters. An activation like ReLU would limit the output values to strictly > 0, and sigmoid would restrict it to the interval $[1, 0]$.

All models were trained in Ridge regression and OLS regression, achieving similar results, shown in table 3

Method	$\lambda = 0$	$\lambda = 10^{-3}$
SGD	0.011	0.0117
Momentum	0.0073	0.0097
Adagrad	0.0073	0.0092
RMSprop	0.0037	0.0063
Adam	0.0007	0.0008

Table 3: Comparison of MSE values from FFNN for different optimization methods with $\lambda = 0$ and $\lambda = 10^{-3}$

The models have much better results with a FFNN than with linear regression. In linear regression, more simple mehtods such as GD and performed better with Ridge regression than with OLS. In all other

cases OLS outperformed Ridge. The results of OLS regression on FFNN is visualized in figure 2.

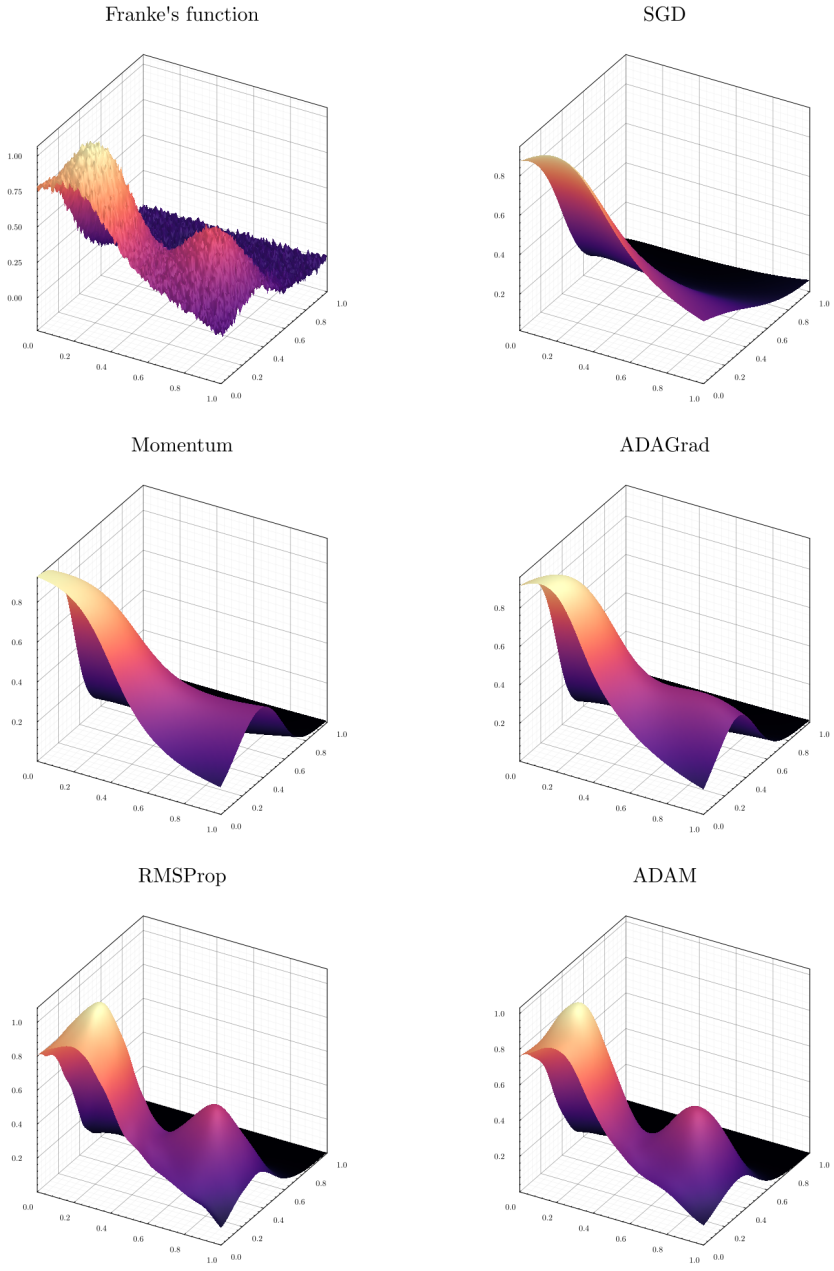
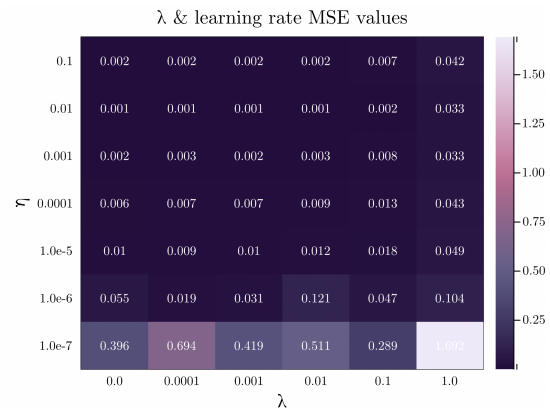
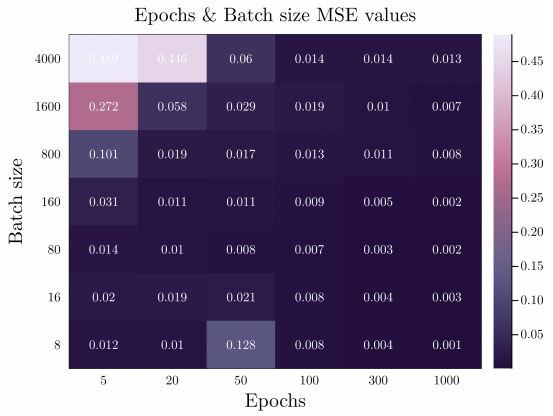


Figure 2: Franke's function predictions from the manually designed FFNN with Zygote gradient.

Finally, the experiment repeated the hyperparameter analysis conducted for the one-dimensional polynomial, as shown in Figure 3. The results align closely with those in Figure 1. Notably, the model's dependency on λ is nearly eliminated, except at high values around 1, which highlights the strong generalizability of the FFNN. The model performs effectively with or without the Ridge penalty, while the linear regression model demonstrates greater sensitivity. Additionally, the model shows less dependence on batch size and number of epochs, leading to quick and efficient convergence.



(a) MSE values for various values for the learning rate η and ridge penalty value λ, with 300 epochs and batch size of 80



(b) MSE values for various number of epochs and batch sizes with $\eta = 0.001$, and λ=0.001

Figure 3: Hyperparameter turning the FFNN with SGD on Franke's function

3.3 Wisconsin Breast Cancer

This is a multidimensional dataset, shown in table 4

Row	Variable	Mean	Min	Median	Max	Nmissing	Eltype
1	id	3.03718e7	8670	906024.0	911320502	0	Int64
2	diagnosis	-	B	-	M	0	String1
3	radius_mean	14.1273	6.981	13.37	28.11	0	Float64
4	texture_mean	19.2896	9.71	18.84	39.28	0	Float64
5	perimeter_mean	91.969	43.79	86.24	188.5	0	Float64
6	area_mean	654.889	143.5	551.1	2501.0	0	Float64
7	smoothness_mean	0.0963603	0.05263	0.09587	0.1634	0	Float64
8	compactness_mean	0.104341	0.01938	0.09263	0.3454	0	Float64
9	concavity_mean	0.0887993	0.0	0.06154	0.4268	0	Float64
10	concave_points_mean	0.0489191	0.0	0.0335	0.2012	0	Float64
11	symmetry_mean	0.181162	0.106	0.1792	0.304	0	Float64
12	fractal_dimension_mean	0.0627976	0.04996	0.06154	0.09744	0	Float64
13	radius_se	0.405172	0.1115	0.3242	2.873	0	Float64
22	fractal_dimension_se	0.0037949	0.0008948	0.003187	0.02984	0	Float64
23	radius_worst	16.2692	7.93	14.97	36.04	0	Float64
24	texture_worst	25.6772	12.02	25.41	49.54	0	Float64
25	perimeter_worst	107.261	50.41	97.66	251.2	0	Float64
26	area_worst	880.583	185.2	686.5	4254.0	0	Float64
27	smoothness_worst	0.132369	0.07117	0.1313	0.2226	0	Float64
28	compactness_worst	0.254265	0.02729	0.2119	1.058	0	Float64
29	concavity_worst	0.272188	0.0	0.2267	1.252	0	Float64
30	concave_points_worst	0.114606	0.0	0.09993	0.291	0	Float64
31	symmetry_worst	0.290076	0.1565	0.2822	0.6638	0	Float64
32	fractal_dimension_worst	0.0839458	0.05504	0.08004	0.2075	0	Float64
33	Column33	-	-	-	-	569	Missing

Table 4: Summary statistics of the dataset

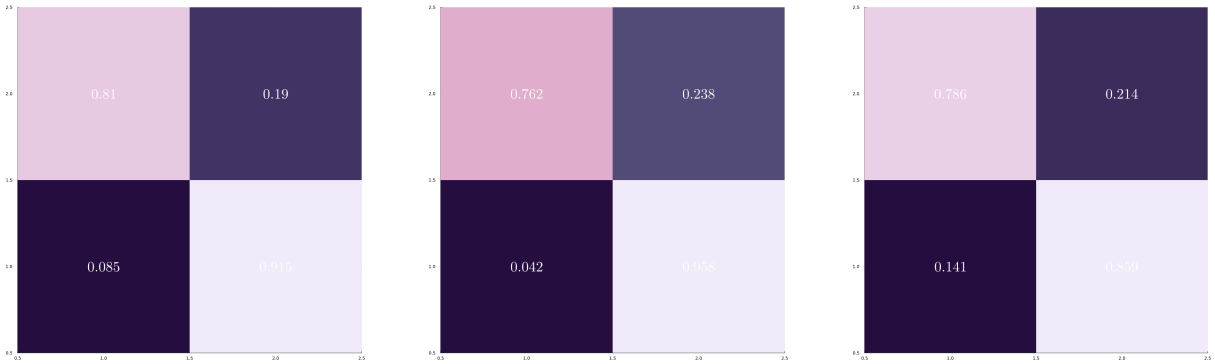
It has 33 features (columns), and a bitwise output (Malignant, Benign $\rightarrow$ 1,0). The last column is simply *missing* values, and must be dropped. The first column are the patient id's, and were dropped as well. The column with Malignant/Begning values are the target, and were taken as y. Thus, there are 30 remaining columns which represent x features. The columns have a high varierty in range, some range between $10^{-3} - 10^{-2}$, others between $10^2 - 10^3$. In order to treat all columns equally, we normalized

each one of them by standardization. Since we do not need to recover the data after prediction, this is done very easily.

Hence, the FFNN takes in an input of size 30, and returns a single number, applied on the sigmoid activation function, which scales it between 0 and 1. If the result is above 0.5, we interpret the result as a 1, and else, a 0. We also had a single hidden layer with 64 nodes and ReLU activation. The loss of the model was calculated with the *binary cross-entropy loss*.

The models were trained for 300 epochs, a a batch-size of 80, and a learning-rate of 0.001. The Logistic regression has an input of size 30 (for 30 features), and an output of 1 with a sigmoid activation function.

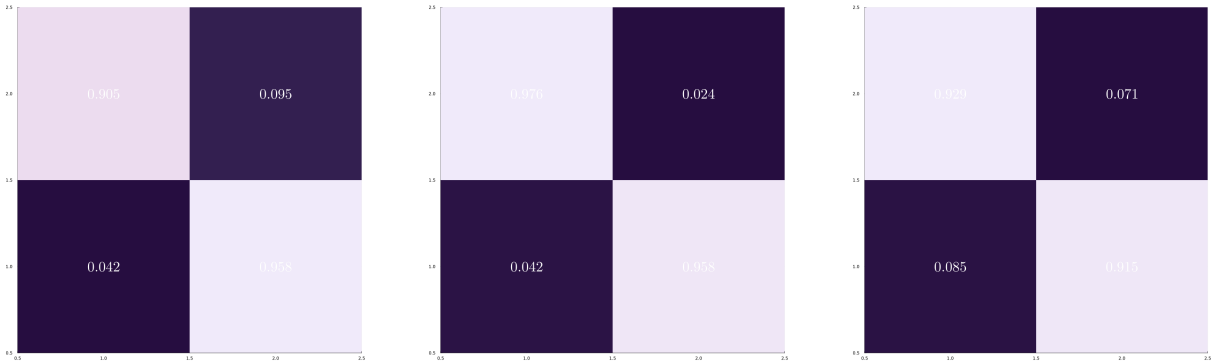
The results of logistic regression are shown in figure 4. All approaches produced very similar results, although Flux produced slighlty better results, and Zygote slightly worse, with the completly manual backprop in the middle. The difference is very small, and not much can be said about there being any fundamental difference between their performance.



(a) Flux model (b) Zygote Gradient (c) Manual Backprop.

Figure 4: Confusion matrix for evaluating the performance of the Logistic Regression with three approaches

The FFNN result is shown in figure 5. Overall, the three approaches gave better results than the logistic regression, due to to the expressive power of the additional hidden layer. Comparing the various approaches however, FFNN gives even more similar performances, only differing in results on the scale of around 0.07%. Flux and manual backpropagation is showing the best results, but they are hard to distinguish. .



(a) Flux model (b) Zygote Gradient (c) Manual Backprop.

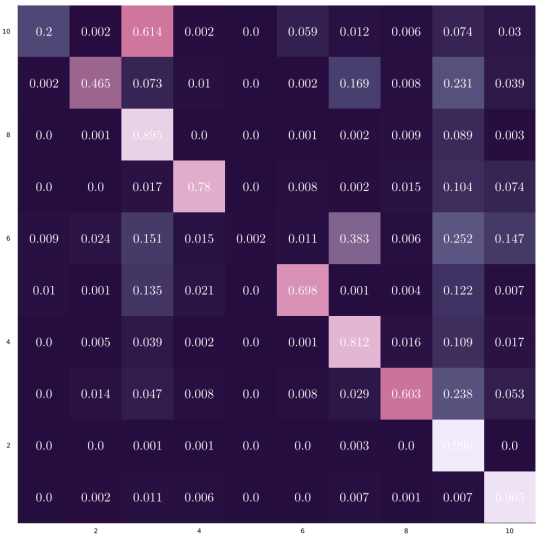
Figure 5: Confusion matrix for evaluating the performance of the FFNN with three approaches

3.4 MNIST

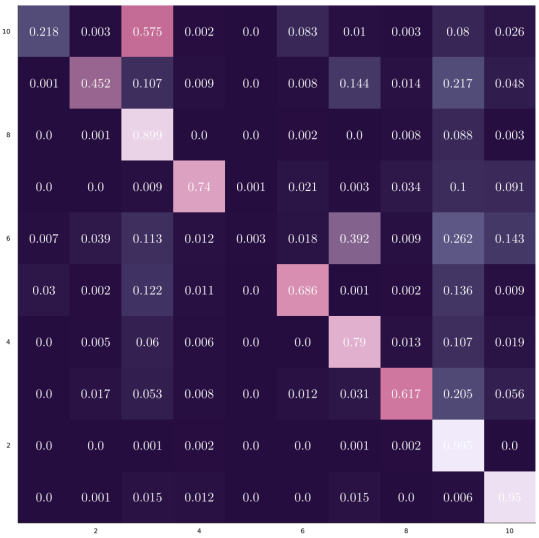
This is an even more complex dataset, and a staple in machine learning. The standard approach is a convolutional nerual network, but since we are applying only a FFNN, we need to *flatten* the images of size 28×28 to a vector with size 784, which is now the input size. The output, is of size 10, representing the probabilty of each class. The highest probabilty is interpreted as the *prediction* of the model. Once

again, the architecture, is rather small, with a first layer projecting the 784 vector to 32 nodes, with a sigmoid activation, and then to the output of 10 nodes, with a softmax activation. This was identical for the Flux model, as well as our own. We used SGD as optimizer. Since the values inside the data range from 0 to 255, we scaled it by dividing each value by 255. The models were trained for 200 epochs, with a batch-size of 80, and a learning-rate of 0.1.

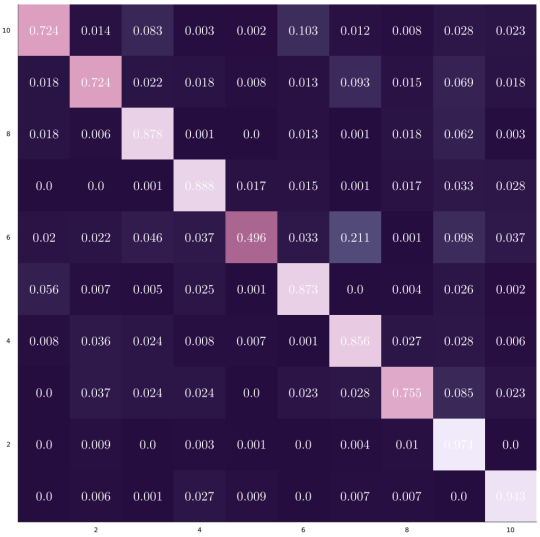
The results of logistic regression are presented in terms of confusion matrices in figure 6. Zygote and Flux gave very similar results, and backpropagation achieved better results.



(a) Flux model



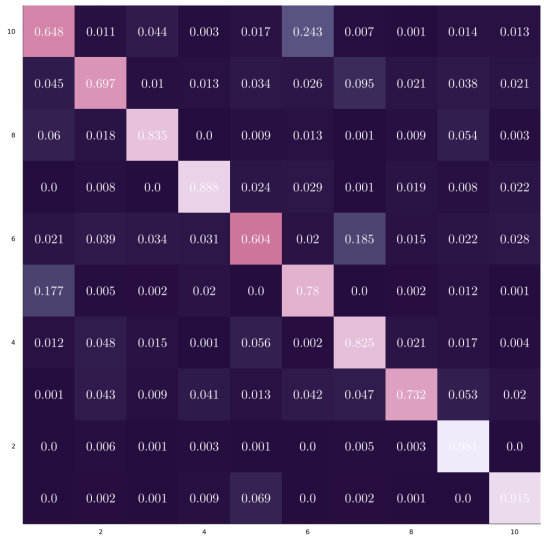
(b) Zygote Gradient



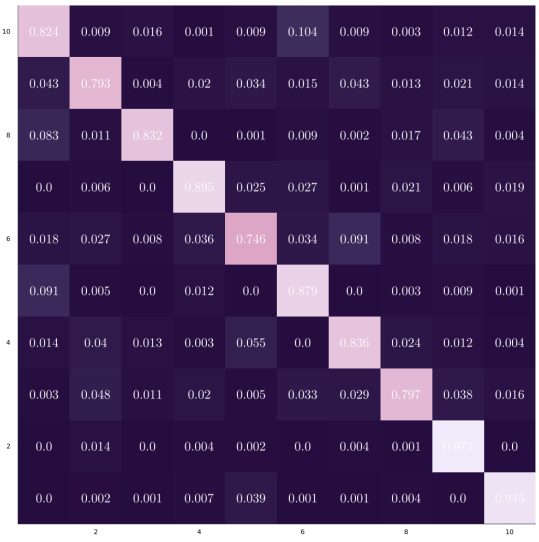
(c) Manual Backprop.

Figure 6: Confusion matrix for evaluating the performance of the three approaches for Logistic regression

For the FFNN, the result can be studied in figure 7. All models performed better than logistic regression, but maybe suprisingly, show a similar trend as logistic regression: the Flux-model, gave visibly more poor results than the Zygote gradient, which again gave poorer results than the manual Bakcpropagation. Accross different runs, the results stayed relatively stable. On average, the Backpropagation gave a resulting accuracy of 91.39%, Zygote 85.43% and Flux 79.46%.



(a) Flux model



(b) Zygote Gradient

(c) Manual Backprop.

Figure 7: Confusion matrix for evaluating the performance of the three approaches for a FFNN

4 Discussion

The primary objective of this project was to evaluate and compare different approaches to implementing feedforward neural networks (FFNNs) and logistic regression models across various regression and classification tasks. Specifically, we assessed the performance of manually coded backpropagation, the use of automatic differentiation through Zygote, and the employment of the Flux library. The tasks ranged from fitting a simple one-dimensional polynomial to tackling complex datasets like the MNIST handwritten digits.

4.1 Comparative Performance Across Methods

Our results indicate that the choice of implementation method—manual backpropagation, Zygote gradients, or Flux—has varying impacts depending on the complexity of the task. In simpler tasks such as the one-dimensional polynomial fitting, differences between optimization algorithms were minimal, and the method of gradient computation had negligible effects on performance. ADAM outperformed other optimizers, achieving a final loss of approximately 10^{-16}, while others like SGD and GD showed slightly higher losses.

As the complexity increased with the Franke's function, the advantages of more sophisticated optimization algorithms became apparent. Advanced methods like ADAM and RMSprop significantly outperformed simpler ones like GD and SGD in linear regression tasks. The use of FFNNs further improved performance, reducing the mean squared error (MSE) compared to linear regression models. Interestingly, the inclusion of a Ridge penalty (λ) did not consistently enhance performance, suggesting that overfitting was not a significant issue for most optimization methods, except perhaps for the simpler GD and SGD.

In the binary classification task using the Wisconsin Breast Cancer dataset, all three approaches—manual backpropagation, Zygote, and Flux—yielded comparable results in both logistic regression and FFNN implementations. The FFNN models, benefiting from an additional hidden layer, outperformed logistic regression models, highlighting the expressive power added by non-linear transformations.

The MNIST dataset presented a more challenging multiclass classification problem. Here, the manual backpropagation method consistently outperformed both Zygote and Flux implementations, achieving an average accuracy of 91.39%, compared to 85.43% for Zygote and 79.46% for Flux. This trend was observed in both logistic regression and FFNN models, with the gap in performance being more pronounced in the FFNN implementations.

4.2 Implications of Gradient Computation Methods

The superior performance of manual backpropagation in the MNIST task suggests that custom implementations can be more efficient or better optimized for specific problems. One possible explanation is that manual backpropagation allows for greater control over the computational graph and memory management, potentially leading to more efficient gradient computations. In contrast, automatic differentiation tools like Zygote, while highly flexible and easier to implement, may introduce overhead or less optimized computational paths, especially in complex models.

Flux, being a high-level library, abstracts many underlying computations, which can be both an advantage and a drawback. While it simplifies model implementation and accelerates development, it may not provide the same level of optimization as a carefully tuned manual implementation.

4.3 Role of Hyperparameters and Model Architecture

Hyperparameter tuning emerged as a critical factor influencing model performance across all tasks. The learning rate (η) was particularly influential; too small a value led to slow convergence, while too large a value caused divergence due to exploding gradients. The optimal learning rate varied between optimization algorithms—ADAM, for instance, required a higher learning rate than SGD to achieve optimal performance.

The impact of the Ridge penalty (λ) was more nuanced. In simpler tasks, introducing a penalty term sometimes improved performance for basic optimization methods like GD and SGD by mitigating overfitting. However, in more complex models and with advanced optimization algorithms, the penalty term had minimal or even negative effects on performance, suggesting that these models were robust to overfitting without regularization.

The architecture of the neural networks also played a significant role. Adding hidden layers and increasing the number of nodes enhanced the model's ability to capture complex patterns in the data, as evidenced by the improved performance of FFNNs over logistic regression models in both the Franke's function, the Wisconsin Breast Cancer dataset and MNIST. However, this also increased the computational complexity and the need for careful hyperparameter tuning.

4.4 Dataset Complexity and Model Generalization

The varying performance across different datasets underscores the importance of model selection in relation to dataset complexity. The FFNN models showed significant improvements over linear and logistic regression models in tasks involving non-linear and high-dimensional data. The MNIST dataset, with its high dimensionality and complexity, highlighted the limitations of simpler models and the need for architectures capable of capturing intricate patterns.

Moreover, the MNIST results suggest that the benefits of manual backpropagation become more pronounced as the complexity of the task increases. This could be due to the ability to tailor the backpropagation algorithm more closely to the specifics of the dataset and model architecture, potentially avoiding generic computations that may not be optimal for the task at hand.

4.5 Limitations and Future Work

While the manual backpropagation method demonstrated superior performance in complex tasks, it comes with the trade-off of increased implementation time and potential for errors. Automatic differentiation tools like Zygote and libraries like Flux offer significant advantages in terms of ease of use and rapid development, which are critical factors in real-world applications.

Future work could explore optimizing the use of automatic differentiation and high-level libraries by fine-tuning their parameters and leveraging their advanced features. Additionally, experimenting with deeper and more complex network architectures, such as convolutional neural networks for image data like MNIST, could provide further insights into the interplay between model complexity, implementation methods, and performance. Comparison of initialization techniques, different activation functions and model architectures could also be explored. The more modern Julia package Lux should also be compared to the manual code and Flux.

5 Conclusion

This project demonstrates that the method of implementing neural networks—whether through manual backpropagation, automatic differentiation via Zygote, or using high-level libraries like Flux—can significantly impact performance, particularly in complex tasks. Our experiments across regression and classification problems reveal several key findings.

Firstly, in simpler tasks such as fitting a one-dimensional polynomial, all methods performed comparably, with minimal differences in loss values. This suggests that for straightforward problems, the choice of implementation method may be less critical, and ease of implementation can be prioritized.

However, as task complexity increased, the differences became more pronounced. In the Franke's function regression and the binary classification task using the Wisconsin Breast Cancer dataset, advanced optimization algorithms and manual backpropagation showed improved performance over automatic differentiation and library-based implementations. This indicates that manual control over gradient computations and network architecture can lead to more efficient learning and better results in moderately complex problems.

Most notably, in the multiclass classification task using the MNIST dataset, the manually implemented neural network with backpropagation significantly outperformed both the Zygote-based and Flux-based models. The manual implementation achieved an accuracy of 91.39%, compared to 85.43% with Zygote and 79.46% with Flux. This highlights the potential performance benefits of manual implementations in handling high-dimensional and complex data, possibly due to more optimized computations and the ability to tailor the algorithm closely to the specific task.

The pros and cons of each method are evident from these findings. Manual backpropagation offers superior performance and the ability to customize the learning process extensively, but it requires considerable effort, expertise, and is prone to implementation errors. Automatic differentiation tools like Zygote provide a balance, offering flexibility and reducing the likelihood of errors, though they may introduce computational overhead. High-level libraries like Flux significantly simplify model development and deployment but may not always achieve the highest possible performance without extensive tuning.

For future work, exploring ways to optimize automatic differentiation and library-based methods could bridge the performance gap observed in complex tasks. Investigating the impact of deeper or more complex network architectures, such as convolutional neural networks for image data, could also provide further insights. Additionally, applying these methods to a broader range of datasets and tasks would help generalize the findings and potentially reveal new considerations in the choice of implementation method.

In conclusion, the choice between manual implementation and the use of automatic differentiation or high-level libraries should be informed by the specific requirements of the task at hand. For simpler problems or when rapid development is essential, using libraries like Flux may be most appropriate. For complex tasks where performance is critical, investing the additional effort in manual implementation may yield significant benefits. Balancing these considerations is key to effectively leveraging neural networks and logistic regression models in practical applications.

Acknowledgements

Almost everything in this project is inspired by the theory in the GitHub repository by Morten Hjorth-Jensen Hjorth-Jensen (2024a). The theory in this section is based on the Jupyter Book by Morten

Hjorth-Jensen Hjorth-Jensen (2024b). I also acknowledge OpenAI's ChatGPT for assisting in drafting and refining parts of this project OpenAI (2024). The code can be found here Sekkelsten (n.d.)

References

Hjorth-Jensen, M. (2024a). *Applied data analysis and machine learning.* Retrieved from https://github.com/CompPhysics/MachineLearning (Accessed: 2024-10-07)

Hjorth-Jensen, M. (2024b). *Applied data analysis and machine learning.* Retrieved from https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html (Accessed: 2024-10-07)

OpenAI. (2024). *Chatgpt conversation with aleksander sekkelsten.* Retrieved from https://chatgpt.com/share/672897dc-2e44-8001-b6cc-37e50ade240a (Accessed: 2024-11-04)

Sekkelsten, A. (n.d.). *UIO Numerical work.* Retrieved from https://github.com/Im2ql4u/Machine-learning-UIO/tree/main/Projects/Project%202