Project 2

FYS-STK4155

Erik Joshua Røset & Oskar Idland
University of Oslo, Department of Physics
(Dated: November 7, 2024)

In this project, we extend our exploration of machine learning techniques to include advanced optimization algorithms and neural networks. We implement and analyze gradient descent methods, including Stochastic Gradient Descent (SGD), as well as feed-forward neural networks. Our analysis focuses on the mathematical foundations of backpropagation, activation functions, and regularization techniques. We apply these methods to both regression and classification tasks, using synthetic data generated through the Franke function and the Wisconsin Breast Cancer dataset. By comparing our implementations with established libraries like Scikit-learn and PyTorch, we gain insights into the practical considerations of neural network development and optimization. Our results demonstrate the importance of activation functions, initialization strategies, and regularization methods in training effective neural networks. We also highlight the impact of optimization techniques on model convergence and generalization, providing a comprehensive overview of the fundamental building blocks of modern machine learning systems.

https://github.com/Oskar-Idland/FYS-STK4155-Projects

I. INTRODUCTION

II. THEORY

In this project, we extend our previous exploration of machine learning techniques into more advanced optimization algorithms and neural networks. Gradient descent methods, along with their various enhancements, are crucial in training neural networks, which are used to model complex, non-linear relationships. While the foundational principles of regression analysis and optimization were addressed in our previous report, here we focus specifically on gradient descent with and without momentum, and tuning methods such as decaying learning rates, AdaGrad, RMSprop, and ADAM. Additionally, the fundamentals of neural networks will be discussed, emphasizing their relevance to the current project.

A. Gradient Descent Methods

Gradient descent is a fundamental optimization algorithm used to minimize a cost function $C(\beta)$, where β represents the model parameters. Unlike the closed-form solutions available for OLS and Ridge regression in Project 1, many modern machine learning models, particularly neural networks, require iterative optimization methods. The core principle of gradient descent is to iteratively update the parameters in the direction that minimizes the cost function. Given a cost function $C(\beta)$ and current parameters β_t at iteration t, the basic update rule is:

$$\beta_{t+1} = \beta_t - \eta \nabla_\beta C(\beta_t), \tag{1}$$

where $\eta > 0$ is the learning rate that controls the size of the update step, and $\nabla_\beta L(\beta_t)$ is the gradient of the cost function with respect to β . The gradient represents the direction of steepest ascent in the cost landscape, so moving in the negative gradient direction ensures we minimize the cost function.

1. *Gradient Descent*

In its simplest form, "plain" gradient descent (GD) computes the gradient using the entire dataset:

$$\nabla_\beta C(\beta_t) = \sum_{i=1}^{n} \nabla_\beta C_i(\beta_t). \tag{2}$$

Where n is the total number of training examples and $C_i(\beta_t)$ is the cost for the i-th example. While this approach would provide the most accurate estimate of the gradient, it has several significant limitations:

- Computational Cost: For large datasets, computing the gradient over all n data points becomes extremely expensive, as each update step requires a complete pass through the dataset.
- Local Minima: Since the algorithm is deterministic, it will converge to a local minimum of the cost function if it converges at all. In machine learning applications, where we often deal with highly non-convex cost landscapes, this can lead to suboptimal solutions.
- Initial Conditions: The final solution strongly depends on the initialization of parameters. Different starting points may lead to different local minima, making the choice of initialization crucial.
- Uniform Step Size: The learning rate η is uniform across all parameter dimensions, which can be problematic when the cost surface has different curvatures in different directions. This often

forces us to use a conservatively small learning rate determined by the steepest direction, significantly slowing down convergence in flatter regions.

2. Stochastic Gradient Descent

To address these limitations, Stochastic Gradient Descent (SGD) introduces randomness into the optimization process. Instead of using the entire dataset for each update, SGD uses randomly selected subsets of the data, called mini-batches:

The fundamental idea that SGD is built on is that the cost function can be written as the average of the cost functions for individual training examples. It then follows that the gradient can be computed as a sum over individual gradients. We can then approximate the gradient by only computing the gradient for a single minibatch:

$$\nabla_\beta C(\beta_t) \approx \sum_{i \in B_k} \nabla_\beta c_i(\mathbf{x}_i, \beta_t). \tag{3}$$

The entire dataset can be split into n/M minibatches (B_k). The size M of the minibatch represents a key parameter choice in SGD. When $M = n$, we recover the "plain" gradient descent method, while $M = 1$ represents pure stochastic gradient descent where updates are made using a single randomly chosen data point. The choice of minibatch size M thus allows us to balance between the accurate but computationally expensive gradient estimates of batch gradient descent and the noisy but frequent updates of pure SGD. In practice, minibatch sizes are often chosen to be much smaller than n to maintain the computational efficiency and stochastic nature of SGD while reducing the variance in gradient estimates compared to when $M = 1$. This gives us the update rule for SGD:

$$\beta_{t+1} = \beta_t - \eta \sum_{i \in B_k} \nabla_\beta c_i(\mathbf{x}_i, \beta_t). \tag{4}$$

Only processing a subset of the data at each iteration facilitates more frequent parameter updates, as the required computational power for each iteration is significantly reduced. The inherent noise in gradient estimates can also help the optimizer escape poor local minima and saddle points. However, the algorithm becomes highly sensitive to the choice of learning rate. Too large values can cause divergence, while too small values lead to slow convergence.

B. Advanced Optimization Methods

These limitations motivate two key enhancements to the basic SGD algorithm. The uniform step size problem can be addressed by introducing momentum, which helps the optimizer maintain velocity in consistent directions while damping oscillations in regions of varying curvature. The learning rate sensitivity issue can be tackled through adaptive learning rate methods, which automatically adjust the learning rate based on the observed geometry of the cost function during training.

1. Momentum

A key limitation of basic gradient descent methods is their uniform step size across all directions, which can lead to slow convergence, especially in regions where the cost surface has different curvatures in different directions. Momentum addresses this by accumulating a velocity vector that helps accelerate convergence and dampen oscillations:

$$v_{t+1} = \gamma v_t + \eta \nabla_\beta C(\beta_t), \tag{5}$$

$$\beta_{t+1} = \beta_t - v_{t+1}, \tag{6}$$

where γ (typically 0.9) is the momentum coefficient that determines how much of the previous velocity is retained. This modification provides several advantages:

- Faster convergence in regions where the gradient is consistent
- Reduced oscillations in directions of high curvature
- Ability to escape shallow local minima

2. Learning Rate Tuning Methods

Choosing the right learning rate η is critical to the success of gradient descent algorithms. If the learning rate is too large, the optimization may overshoot the minimum, while if it is too small, convergence will be very slow. Several adaptive learning rate methods have been proposed to dynamically adjust the learning rate during training:

Decaying Learning Rate One simple approach is to gradually decrease the learning rate as training progresses, using a schedule such as:

$$\eta_t = \frac{\eta_0}{1 + \lambda t},$$

where η_0 is the initial learning rate, t is the iteration, and λ is the decay rate. This helps ensure that larger updates are made at the start of training when far from the optimum, and smaller, more precise updates are made later.

AdaGrad (Adaptive Gradient Algorithm): AdaGrad adapts the learning rate for each parameter based on the historical gradients. It assigns a smaller learning rate to frequently updated parameters and a larger rate to less frequently updated ones:

$$\beta_{t+1} = \beta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla_\beta L(\beta_t),$$

where G_t is the sum of the squares of the past gradients, and ϵ is a small constant to avoid division by zero. AdaGrad is well-suited for sparse data but may become overly conservative in later iterations due to the cumulative sum of gradients.

RMSprop To address AdaGrad's diminishing learning rates, RMSprop uses a moving average of the squared gradients to scale the learning rate. This helps maintain a balance between fast convergence and smooth parameter updates:

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1-\beta) g_t^2,$$

$$\beta_{t+1} = \beta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \nabla_\beta L(\beta_t),$$

where β (typically 0.9) controls the decay rate of the moving average, and $E[g^2]_t$ is the moving average of the squared gradients.

Adaptive Moment Estimation (ADAM) ADAM combines the advantages of both AdaGrad and RMSprop by keeping track of both the first moment (mean) and the second moment (uncentered variance) of the gradients:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) \nabla_\beta L(\beta_t),$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\beta L(\beta_t))^2,$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t},$$

$$\beta_{t+1} = \beta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t.$$

ADAM is one of the most widely used optimization algorithms today due to its ability to dynamically adjust learning rates and its robustness in practice.

C. Neural Networks

Having established the mathematical framework for optimizing model parameters through gradient descent and its variants, we now turn our attention to neural networks - a powerful class of models whose training relies heavily on these optimization techniques. Neural networks represent a significant advancement beyond the linear models discussed in our previous work, offering the capability to model complex, non-linear relationships in data. These networks can be trained for both regression tasks, where we predict continuous values similar to our previous project, and classification tasks where we predict discrete categories. While linear regression models are constrained to linear combinations of input features, neural networks can approximate arbitrary continuous functions through the composition of multiple non-linear transformations. This property, formally stated in the universal approximation theorem [**?**], means that neural networks can theoretically approximate any continuous function to arbitrary accuracy given sufficient size.

This approximation capability is achieved through a network of interconnected nodes organized into layers, where each node applies a non-linear transformation to its input. The network learns by adjusting parameters called weights and biases, which determine how information flows and is transformed between nodes. During training, these parameters are iteratively adjusted to minimize a cost function, allowing the network to learn patterns directly from the data without explicit feature engineering.

1. Feed-Forward Neural Networks

A feed-forward neural network (FFNN) is structured into three distinct types of layers: the input layer, hidden layers, and output layer. The input layer represents the input data, with each node corresponding to a feature in the dataset. The output layer produces the final prediction, with its structure depending on the task - a single node for regression problems or multiple nodes for classification tasks. Between these lie the hidden layers, where the network learns to identify and extract relevant patterns from the input data. The term "hidden" stems from the fact that while we observe the inputs and final outputs during training, we do not directly observe what features these intermediate layers learn to represent.

The FFNN generalizes our previous linear models by defining the model output $\tilde{y}$ as a nested series of transformations of the input vector $x \in \mathbb{R}^n$. The fundamental computation at each node takes the form:

$$z = wx + b \tag{7}$$

$$a = f(z) \tag{8}$$

Where w are the weights, b is the bias, and $f(\cdot)$ is a non-linear activation function. This computation can be extended to a layer of neurons by organizing the weigths into a matrix $W \in \mathbb{R}^{m \times n}$ and the biases into a vector $\mathbf{b} \in \mathbb{R}^m$:

$$\mathbf{z} = W\mathbf{X} + \mathbf{b} \tag{9}$$

$$\mathbf{a} = f(\mathbf{z}) \tag{10}$$

Where $\mathbf{X} \in \mathbb{R}^{b \times n}$ represent a batch of input vectors and $\mathbf{a} \in \mathbb{R}^{b \times m}$ is the layer output and $\mathbf{b}$ is a vector of biases. For a network with L layers, we can express the complete forward propagation as:

$$\begin{aligned}
\mathbf{z}^1 &= W^1\mathbf{X} + \mathbf{b}^1 \\
\mathbf{a}^1 &= f(\mathbf{z}^1) \\
\mathbf{z}^2 &= W^2\mathbf{a}^1 + \mathbf{b}^2 \\
\mathbf{a}^2 &= f(\mathbf{z}^2) \\
&\cdots \\
\mathbf{z}^L &= W^L\mathbf{a}^{L-1} + \mathbf{b}^L \\
\mathbf{a}^L &= f(\mathbf{z}^L)
\end{aligned} \tag{11}$$

This iterative process of matrix multiplications and non-linear transformations enables the network to progressively build more complex representations of the input data at each layer. The final output $\mathbf{a}^L$ is then used to compute the cost function, which is minimized during training to learn the optimal weights and biases.

2. Initialization and Regularization

The weights and biases represent the parameters that the network learns during training. Each weight w_{ij}^l represents the strength of the connection between node i in layer $l-1$ and node j in layer l, determining how much influence the output from one node has on the next. The bias b_j^l for each node j in layer l allows the network to shift its activation function, providing an additional degree of freedom in fitting the data. Together, these parameters define how information is transformed as it moves through the network.

The choice of initial weights can significantly impact whether a network learns effectively or fails to train. Poor initialization can lead to either vanishing or exploding gradients, particularly in deep networks. If initial weights are too small, the signals shrink as they pass through each layer until they become too weak to drive meaningful learning. Conversely, if weights are too large, the signals grow exponentially, saturating activation functions and stalling learning. A common approach is to draw initial weights from a normal distribution with zero mean and a variance scaled by the number of input connections to each neuron.

$$w_{jk}^l \sim \mathcal{N}\left(0, \frac{1}{n_{in}}\right), \tag{12}$$

where n_{in} is the number of inputs to the layer. This scaling helps maintain a similar scale of activations and gradients across layers at the start of training. Biases are typically initialized from a normal distribution with a small scale factor to break symmetry while keeping activations in a reasonable range:

$$b_j^l = 0 \text{ or } b_j^l = 0.01. \tag{13}$$

This small random initialization for biases helps prevent all neurons in a layer from developing identical behavior during early training.

While Project 1 introduced regularization in the context of Ridge regression, neural networks benefit from similar techniques to prevent overfitting. The most common approach is L2 regularization (weight decay), which adds a penalty term to the cost function:

$$\begin{aligned}
C(\theta) &= \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}_i(\theta) \\
&\to \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}_i(\theta) + \lambda||w||_2^2 \\
&= \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}_i(\theta) + \lambda\sum_{ij} w_{ij}^2.
\end{aligned} \tag{14}$$

Where λ is the regularization parameter, N is the number of training samples, and the sum runs over all weights in the network. This modification encourages the network to use smaller weights and distribute the learned patterns across multiple nodes rather than relying too heavily on any single connection.

3. Activation Functions

The non-linear activation functions are fundamental components that enable neural networks to approximate complex functions. By introducing non-linearity between layers, these functions allow the network to learn and represent patterns that would be impossible with purely linear transformations. The careful selection of activation functions significantly impacts both the network's expressive capability and its training dynamics.

An activation function $f(\cdot)$ takes the weighted sum z of a node's inputs and transforms it into an output signal $a = f(z)$. For our implementations we consider the following activation functions.

Identity: The identity function, simply defined as

$$f(z) = z,$$

passes its input unchanged. While this function is rarely used in hidden layers due to its linearity, it serves as the standard activation function for regression tasks in the output layer.

Sigmoid: The Sigmoid function, defined as

$$\sigma(z) = \frac{1}{1+e^{-z}}, \tag{15}$$

maps any real input the the interval $(0, 1)$. This smooth, continuously differentiable function has historically been

popular due to its biological inspiration and interpretable output range that naturally corresponds to probability-like quantities. However, it suffers from several drawbacks:

- For inputs with large magnitudes, the function becomes nearly flat, which can complicate the training process
- Its output is not zero-centered, which can introduce systematic bias during training.
- The exponential computation is relative expensive.

Despite these limitations, the sigmoid remains useful in the output layer for binary classification tasks where probability interpretation is desired.

Rectified Linear Unit (ReLU): The ReLU function, defined as

$$f(z) = \max(0, z), \tag{16}$$

has become the default choice for many neural network architectures. As it is a very simple threshold function, it is computationally efficient. It enables sparse activation, as all negative inputs are mapped to exactly zero. However, it suffers from the "dying ReLU" problem, where neurons can become inactive and stop learning if they receive consistently negative inputs.

Leaky Rectified Linear Unit (Leaky RELU): To address the "dying ReLU" problem, the Leaky RELU function introduces a small slope for negative inputs:

$$f(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha z & \text{if } z \leq 0 \end{cases}, \tag{17}$$

where α is a small constant (typically 0.01). This modification prevents complete deactivation of neurons while maintaining the computational efficiency of the standard ReLU. By allowing a small, non-zero output for negative inputs, Leaky ReLU ensures that neurons can continue to participate in the learning process even when receiving predominantly negative inputs.

Softmax: Used primarily in the output layer for multi-class classification, the softmax function generalizes the sigmoid to multiple classes:

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}. \tag{18}$$

Where K is the number of classes. The softmax function produces a probability distribution, I.e. the output values sum to 1, which is useful for interpreting the network's output as class probabilities.

The choice of activation function is a critical design decision that can significantly impact the network's performance. For the hidden layers, the ReLU and Leaky ReLU functions are often preferred due to their computational efficiency and generally better training performance than sigmoid. Cite While the activation function for the output layer depends on the task, the softmax function is commonly used for multi-class classification tasks, while the sigmoid function is suitable for binary classification problems and in regression problems one often uses the identity function.

4. Backpropagation

The backpropagation algorithm provides an efficient method for computing gradients in neural networks, enabling the network to learn from its errors by adjusting weights and biases. While the forward pass computes predictions, backpropagation determines how each parameter should be adjusted to reduce the prediction error. The algorithm derives its name from the way it propagates error gradients backward through the network, from the output layer to the input layer.

To understand backpropagation, we start with a cost function C that measures the network's prediction error. For regression tasks, this is typically the mean squared error (MSE). The goal is similar to that of the gradient descent algorithms: to minimize the cost function. To achieve this, we compute the gradient of the cost function with respect to the network's parameters, the weights $\frac{\partial C}{\partial w^l_{jk}}$ and $\frac{\partial C}{\partial b^l_j}$ for each weight and bias in the network.

The key insight behind backpropagation is the chain rule, which allows us to decompose these calculations into smaller, more manageable steps. We introduce an intermediate quantity, the error term δ^l_j, which represents the error in node j in layer l.

$$\delta^l_j = \frac{\partial C}{\partial z^l_j}. \tag{19}$$

Where z^l_j is the weighted input to the activation function for node j in layer l. This error term represents how a change in the node's input affects the overall cost.

For the output layer, the error term is directly computed:

$$\delta^L_j = \frac{\partial C}{\partial a^L_j}\frac{\partial a^L_j}{\partial z^L_j} = (a^L_j - y_j) f'(z^L_j). \tag{20}$$

Where $f'(\cdot)$ is the derivative of the activation function. For the hidden layers, the error term is recursively computed based on the error terms of the subsequent layer:

$$\delta^l_j = \sum_k w^{l+1}_{kj} \delta^{l+1}_k f'(z^l_j). \tag{21}$$

This equation shows how errors propagate backward: each node's error is a weighted sum of the errors in the next layer, scaled by the local derivative of its activation function.

Once we have computed these error terms, the gradient for any weight or bias follows a simple pattern:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l, \tag{22}$$

and

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \tag{23}$$

In these equation the simplicity of the gradient expressions is evident. The gradient for a weight is the product of the error at its target node and the activation from its source node, while the gradient for a bias is simply at the error at its node.

During backpropagation, regularization terms are also added to the weight gradients to prevent overfitting, while the bias gradients remain unchanged. This selective regularization of weights but not biases reflects the different roles these parameters play in the network: weights determine the importance of connections, while biases set activation thresholds.

The complete backpropagation algorithm can then be summarized in four steps:

- Forward Pass: Compute activations for all layers
- Output Error: Calculate error terms δ^L for the output layer
- Backpropagation: Compute error terms δ^l for each hidden layer
- Gradient Computation: Calculate the gradients using the error terms

This process provides the gradients needed for the various optimization methods discussed earlier to update the network's parameters. The efficiency of backpropagation comes from its clever reuse of intermediate calculations through the chain rule, avoiding redundant computations that would occur in a more naive approach to gradient calculation.

5. *Learning Rate Tuning*

The learning rate tuning methods discussed in section II B 2 for gradient descent optimization are equally applicable to neural network training. The same challenges of balancing convergence speed with stability apply, and methods like AdaGrad, RMSprop, and Adam are commonly used to automatically adjust learning rates during neural network training.

D. Logistic Regression

While neural networks provide a powerful framework for both regression and classification tasks, understanding logistic regression offers valuable insights into many core concepts of deep learning. It can be viewed as a single-layer neural network and introduces key ideas like activation functions and alternative cost functions to the SGD method.

For binary classification problems where outcomes take only two values $\{0, 1\}$, we model the probability of a positive outcome using the sigmoid function (which we encountered in section II C 3 as a neural network activation function):

$$p(y = 1|x, \theta) = \frac{e^{\theta_0+\theta_1 x}}{1 + e^{\theta_0+\theta_1 x}}, \tag{24}$$

$$p(y = 0|x, \theta) = 1 - p(y = 1|x, \theta). \tag{25}$$

For a dataset $D = \{(y_i, x_i)\}$ with binary labels $y_i \in \{0, 1\}$, assuming independent observations, the likelihood is:

$$P(D|\theta) = \prod_i [p(y_i = 1|x_i, \theta)]^{y_i} [1 - p(y_i = 1|x_i, \theta)]^{1-y_i}. \tag{26}$$

Taking the logarithm gives the cross entropy cost function which is also used in classification for neural networks:

$$C(\theta) = -\sum_i \Big[y_i \log(p(y_i = 1|x_i, \theta)) + (1 - y_i) \log(1 - p(y_i = 1|x_i, \theta))\Big]. \tag{27}$$

Using matrix notation with design matrix X, target vector y, and predicted probabilities p, the gradient takes a form similar to what we saw for neural networks:

$$\frac{\partial C(\theta)}{\partial \theta} = -X^T(y - p). \tag{28}$$

Like in neural networks, we often add regularization terms to prevent overfitting. The optimization can be performed using any of the gradient-based methods discussed earlier, though Newton-Raphson's method is traditionally preferred when computationally feasible due to its faster convergence near the minimum.

The logistic regression model illustrates several key concepts that carry over to neural networks: the use of non-linear activation functions to constrain outputs, the importance of choosing appropriate cost functions for classification tasks, and the application of gradient-based optimization methods. Furthermore, many of the training considerations we will discuss next, such as batch processing and learning rate tuning, apply equally to logistic regression and neural networks.

E. Training Considerations for Optimization Methods

While gradient descent, neural networks and logistic regression may appear quite different at first glance, they share many fundamental training elements and challenges. Understanding these commonalities helps illuminate the underlying principles of machine learning optimization

1. Cost Functions and Optimization Goals

All three methods aim to minimize a cost function that measures prediction error. For regression tasks, we prefer the MSE discussed in Project 1 because its quadratic form makes it differentiable and provides stronger penalties for large errors. However, binary classification tasks often employ the logistic regression cost function.

For classification tasks, while we optimize the cross-entropy cost function during training, we typically evaluate model performance using the accuracy score - the fraction of correct predictions. For a binary classification problem with predictions $\hat{y}_i$ and true values y_i, the accuracy is defined as:

$$\text{Accuracy} = \frac{1}{n}\sum_{i=1}^{n} \mathbb{I}(\hat{y}_i = y_i), \tag{29}$$

where $\mathbb{I}$ is the indicator function that equals 1 if $y_i = \hat{y}_i$ and 0 otherwise. For neural networks and logistic regression with sigmoid output, we typically threshold the output probability at 0.5 to obtain binary predictions:

$$\hat{y}_i = \begin{cases} 1 & \text{if } \sigma(x_i) \geq 0.5 \\ 0 & \text{if } \sigma(x_i) < 0.5 \end{cases} \tag{30}$$

While accuracy provides an intuitive measure of model performance, it is worth noting that we do not optimize it directly during training. This is because accuracy is not differentiable (due to the threshold operation), making it unsuitable for gradient-based optimization. Instead, we optimize the cross-entropy cost function, which provides a smooth, differentiable measure that encourages the model to output probabilities close to 0 or 1 for the correct classes.

2. Batch Processing and Stochasticity

They all benefit from stochastic updates using mini-batches. This approach offers several advantages:

- Reduced memory requirements compared to full-batch methods
- More frequent parameter updates
- Introduction of beneficial noise that can help escape local minima
- Potential for better generalization

The choice of batch size involves similar tradeoffs for all cases: smaller batches provide more frequent updates but noisier gradient estimates, while larger batches give more stable gradients but slower convergence. The optimal batch size often depends on both the problem structure and computational constraints.

3. Training Progress

They typically measure progress in terms of epochs - complete passes through the training data. The number of epochs needed depends on factors like dataset size, model complexity, and the difficulty of the task. Training continues until either a maximum number of epochs is reached or some convergence criterion is met. However, the interpretation of "convergence" can differ between methods - neural networks may require more epochs due to their non-linear nature and larger parameter space.

4. Hyperparameter Selection

All approches require careful tuning of hyperparameters that control the learning process:

- Learning rates and their scheduling
- Regularization strength λ
- Optimization algorithm parameters (momentum, decay rates, etc.)
- Batch size and number of epochs

These hyperparameters often interact in complex ways. For example, the optimal learning rate typically decreases with batch size to compensate for more accurate gradient estimates. Similarly, stronger regularization (higher λ) may require more epochs or higher learning rates to reach convergence. Understanding these interactions is crucial for both gradient descent and neural network optimization.

The common thread through all these considerations is the fundamental challenge of optimization: balancing computational efficiency, model accuracy, and generalization ability. Whether using simple gradient descent or complex neural networks, success depends on carefully managing these tradeoffs through proper choice and tuning of training parameters. This understanding provides a unified framework for approaching machine learning optimization, regardless of the specific method employed.

III. METHODS & IMPLEMENTATION

A. Regression Analysis

To explore the optimization methods discussed in the theory section II, we first consider a regression problem using the Franke function. This function is a widely used benchmark for regression tasks, as it provides a smooth, non-linear surface that can be sampled to generate noisy data. We generate a dataset by sampling the Franke function with added Gaussian noise, which we then use to train and evaluate our regression models. We fit our model to this data with a polynomial design matrix, after splitting it into training and test sets, and evaluate the model's performance using metrics like the mean squared error (MSE) and R^2 score.

Regression models have many parameters, making it impossible (or at least impractical) to iterate over all possible combinations. We use grid search to explore two parameters at a time. We then zoom in on the most promising regions to find the optimal parameters. After finding a suitable set of parameters, we continue the search for the next set of parameters.

1. Plain Gradient Descent

No Momentum: To be sure we understand the basic principles of gradient descent, we start by implementing a simple gradient descent algorithm for linear regression. We use the MSE as our cost function and compute the gradient of the cost function with respect to the model parameters. We then update the parameters using the gradient and iterating over a set of learning rates.

Momentum: We then extend our implementation to include momentum. Now we iterate over all possible values of the momentum and learning rate to find the optimal combination for our model. After creating a general overview of the entire parameter space, we zoom in on the most promising regions to find the optimal parameters. This process will be repeated for the other optimization methods as well.

2. Stochastic Gradient Descent

No Momentum: Continuing with stochastic gradient descent, we implement a simple version of the algorithm and apply it to our regression problem. We then begin by iterating over all possible learning rates and batch sizes to find the optimal combination. We then zoom in on the most promising regions to find the optimal parameters.

Momentum: Secondly, we use the ideal learning rate, and then iterate over possible values of the momentum to find the optimal combination. We then zoom in on the most promising regions to find the optimal parameters.

Moving on to the more advanced optimization methods, we implement AdaGrad, RMSprop, and Adam. We follow the same procedure as before, iterating over all possible learning rates and momentum values to find the optimal combination. We note the optimal values to use in the neural network implementation.

3. AdaGrad

a. No Momentum: For both the regular and stochastic versions of AdaGrad, we iterate over all possible learning rates to find the optimal value.

b. Momentum: Using the optimal values found in the previous step, we then iterate over all possible values of the momentum to find the optimal value to pair with the learning rate.

4. RMSprop & Adam

a. No Momentum:

B. Neural Networks

In order to explore the optimization methods discussed in the theory section, we consider again a regression problem on the Franke function, and also a classification problem on the Wisconsin Breast Cancer dataset. The hyperparameter space is vast, and in order to effectively produce results we narrow down the possibilities by first deciding on a singular network model to base our work on. This entails initializing our models with pseudo-random weights sampled from a univariate normal distribution with zero mean and a variance of one. The same goes for the biases, but with an added constant of 0.01 in order to break symmetry. With the focus on exploring the influence of hyperparameters, we choose to keep the network architecture simple, with a single hidden layer and a fixed number of nodes. This allows us to focus on the effects of the learning rate, batch size, and regularization strength on the network's performance.

1. Franke Function Regression

We first validated our neural network implementation using the Franke function regression problem. Input data was generated by sampling the Franke function on a uniform grid of 100×100 points over the unit square $[0, 1] \times [0, 1]$. To simulate measurement noise, we added Gaussian noise with mean zero and standard deviation $\sigma = 0.01$ to the function values.

The input coordinates (x, y) were used to generate a design matrix with polynomial features up to degree 4, chosen to match the complexity of the Franke function's structure. The dataset was split into training (80%) and test (20%) sets using scikit-learn's `train_test_split function`. Both input features and

target values were standardized using StandardScaler, with the scaling parameters computed only from the training set to prevent data leakage.

The network was configured with a single hidden layer of 15 nodes, empirically chosen as a balance between model complexity and computational efficiency. The output layer consisted of a single node for regression. We initialized weights from a normal distribution $\mathcal{N}(0, 1)$ and biases with a small constant offset of 0.01 to break symmetry in the network's initial state.

For our systematic hyperparameter exploration, we established a standard configuration:

- Activation function: Sigmoid in the hidden layer
- Cost function: Mean Squared Error (MSE)
- Optimizer: AdaGrad with momentum ($\eta = 0.001$, $\gamma = 0.8$)
- Regularization rate: $\lambda = 0.001$

These baseline parameters were chosen based on preliminary experiments showing stable convergence without excessive overfitting. This configuration served as our control point, allowing us to systematically vary individual parameters while holding others constant.

Epochs vs batch size: Our first step was to explore the relationship between the number of epochs and the batch size. We iterated over epoch sizes $(100, 500, 1000, 2000)$ and number of batches $(1, 10, 20, 50, 100)$ and calculated the score metrics for each combination. From here we went further with using 500 epochs and 20 batches for the rest of the analysis.

Learning rate (η) vs Regularization (λ): This was conducted in similar fashion to the previous step, with the learning rate η ranging from values ($1e^{-4}$ to $1e^{-1}$ and 0.5), and the regularization parameter λ ranging from ($1e^{-5}$ to $1e^{-1}$).

Schedulers: To compare the different learning rate schedulers discussed in section II B 2, we implented one instance of each of them with the standard learning rate. For plain momentum and Adagrad with momentum, we used the parameters $\gamma = 0.8$ again. For Adam, we used the parameters $\rho = 0.9, \rho_2 = 0.999$. RMPprop had the same ρ value as Adam. To ensure numerical stability, we used a small value of $\epsilon = 1e^{-8}$ for all the optimizers. The momentum and decay parameters were chosen based on common values in deep learning literature that typically show good performance across different problems.

Training progression was monitored by computing both MSE and R^2 scores on the training data at each epoch. Convergence was defined quantitatively as the point where the relative difference between the mean MSE of the last 10 epochs and the current MSE fell below 10^{-2}, providing a relative convergence criterion for comparing optimizer efficiency. For final model evaluation, we computed these metrics on the held-out test set to assess generalization performance.

2. Breast Cancer Classification

For the classification task, we used the Wisconsin Breast Cancer dataset, which contains 30 features computed from digitized images of fine needle aspirates (FNA) of breast mass, with binary labels indicating malignant or benign tumors. The data was split into training and test sets using an 80-20 split. Unlike the Franke function data, only the input features were standardized since the target values are binary.

We maintained the same basic network architecture as in the regression task, initially using a single hidden layer with 15 nodes. The sigmoid activation function was employed in both the hidden and output layers, with the output layer producing a single value representing the probability of malignancy. For this binary classification problem, we used the logistic regression cost function eq. (27).

Given the relatively small size of the dataset compared to the Franke function case, we reduced the training duration to 20 epochs with 20 batches. The initial configuration used a regularization parameter $\lambda = 0.01$ and the Adam optimizer with learning rate $\eta = 0.001$, and momentum parameters $\rho = 0.9$, $\rho_2 = 0.999$. Model performance was evaluated using both training and test set accuracy scores.

Our investigation proceeded in three main stages:

Learning Rate and Regularization Parameter: Learning Rate and Regularization: We performed a grid search over learning rates and regularization parameters, both ranging from 10^{-5} to 10^{-1}. This explored the balance between model convergence speed and stability versus overfitting prevention.

Network Architecture: We systematically investigated the effect of network depth and width by varying:

- Number of hidden layers: 1 to 3 layers
- Neurons per layer: 5 to 25 neurons, in steps of 5

Each configuration was trained with the previously determined optimal learning rate and regularization parameter to isolate the effect of architecture changes.

Activation Functions: Maintaining the 15-neuron architecture, we compared three different activation functions in the hidden layers:

- Sigmoid
- ReLU
- Leaky ReLU

This comparison was performed across networks with 1 to 3 hidden layers to understand how different activation functions affect the network's learning capacity at varying depths. The output layer retained the sigmoid activation function to maintain proper probability outputs for classification.

C. Logistic Regression

To evaluate different approaches to classification of the Wisconsin Breast Cancer dataset, we implemented logistic regression alongside our neural network. While our neural network used a single hidden layer, logistic regression represents an even simpler model that can be viewed as a neural network without any hidden layers, using only a sigmoid activation function on the output.

Using the same preprocessed dataset as in our neural network analysis (standardized features, 80-20 train-test split), we implemented logistic regression with the standard sigmoid function and logistic regression cost function.

To enable direct comparison with our neural network results, we performed a similar grid search over:

- Learning rates η: $[10^-5,10^{-4}, 10^-3, 10^{-2}, 10^{-1}]$
- Regularization λ: $[10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}]$

We maintained consistency with our other implementations by using stochastic gradient descent with batch size 20 and 20 epochs. The model's performance was evaluated using the accuracy score on both training and test sets, allowing comparison between the simpler logistic regression and the single-hidden-layer neural network approach.

D. Program

1. Code Structure

All the source code that we developed and used to produce our results is available on our GitHub repository, linked in section VI in `Project_2/src`. The `README.md` file contains the entire project file structure. To replicate our exact results, use the provided `requirements.txt` file to install the necessary packages. Our code is divided into the following files and notebooks.

Regression Analysis `RegressionModel.py` contains the classes for the regression models.

`regression_anal.ipynb` contains the analysis of the Franke function data, which we used to validate our implementation.

Neural Networks `FFNN.py` contains the classes for the neural networks.

`nn_Franke.ipynb` contains the analysis of the Franke function to validate the implementation of neural networks.

`nn_breast_cancer.ipynb` contains the analysis of the breast cancer data using neural networks. This is where we truly optimize the hyperparameters of the neural networks.

`activation_funcs.py` contains the activation functions used in the neural networks.

`cost_funcs.py` contains the cost functions used in the neural networks.

Miscellaneous `utils.py` contains utility functions used throughout the project. This includes plotting, data generation and other repetitive tasks.

E. Tools

All our code is written in Python (3.12) [**?**], and we used scikit-learn [**?**] and py-torch to test against our models. To vectorize our code we used `numpy` [**?**], and for visualization we used `matplotlib.pyplot` [**?**]. All python packages and their versions can be found in our `requirements.txt`. Code completion and debugging was done in Visual Studio Code [**?**] with additional assistance of GitHub Copilot [**?**]. We used `git` [**?**] for version control, and `GitHub` [**?**] for remote storage of our code.

IV. RESULTS & DISCUSSION

A. Regression Analysis

1. *Plain and Stochastic Gradient Descent*

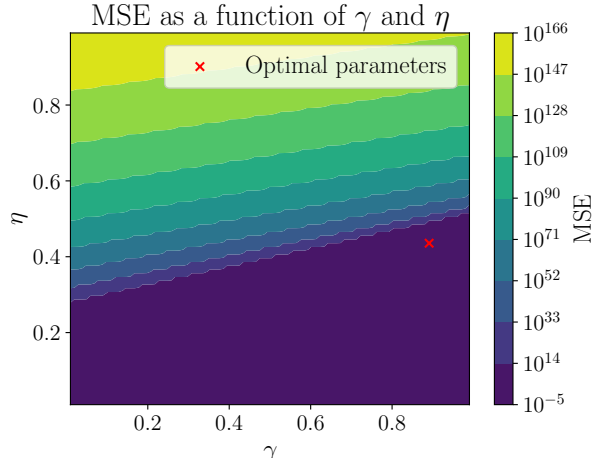

FIG. 1: Overview of the entire parameter space, plotting learning rate against momentum, using plain gradient descent. The optimal parameters are marked with a red cross, to use as a starting point for further analysis.

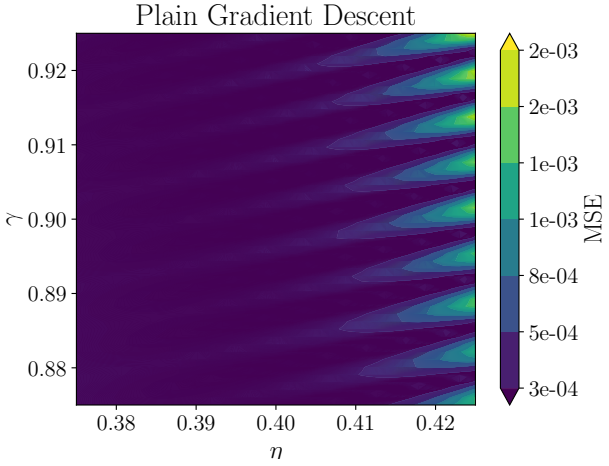

FIG. 2: Narrowed down parameter space from fig. 1.

Looking at fig. 1 we found a region of parameters in the lower-right corner, which gave the lowers MSE values. Using the best parameters found marked with a cross as a starting point, we found a good interval of values as seen in fig. 2, with an order of magnitude around 10^{-4} and 10^{-3}. In the following results, we only present the best intervals. The entire overview can be found the appendix #TODO: Reference .

Continuing our search for the optimal batch number, we found between 5 and 20 batches to work well. In

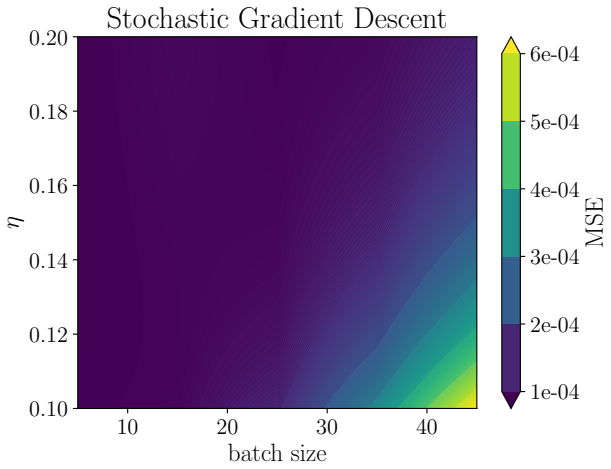

FIG. 3: Plotting batch size, against learning rate, using stochastic gradient descent.

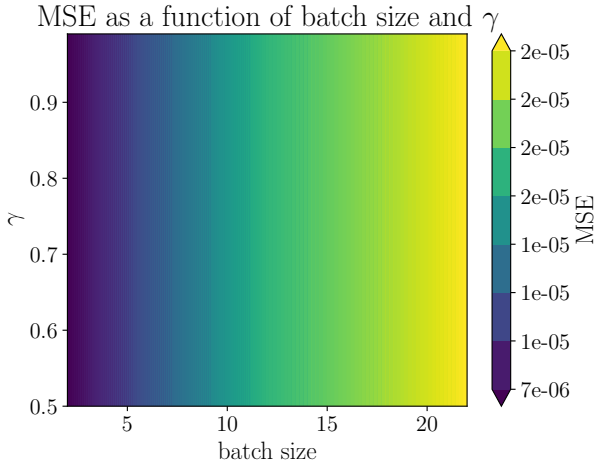

FIG. 4: Plotting the batch size against the momentum, using stochastic gradient descent. Using a learning rate of 0.2

fig. 3 we have no momentum. As a learning of 0.2 gave good results, we used this as our fixed learning rate in fig. 4 to find an optimal number of batches as a function of momentum. We see that the effect of changing the momentum is almost negligible, and that the number of batches is the most important parameter to tune. Even though we see a clear gradient in the plot, all the MSE values are in the same order of magnitude around 10^{-5}. The same is true in fig. 3, where all learning rates between 0.1 and 0.2 give MSE values around 10^{-4}.

2. *AdaGrad*

Exploring further with fig. 5 and fig. 6 we found that the stochastic performed better then plain gradient descent version with about 4 orders of magnitude. The

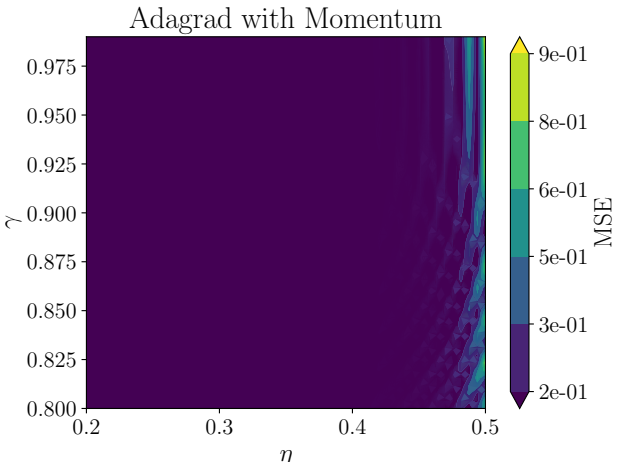

FIG. 5: Plotting the learning rate against the momentum, using regular AdaGrad

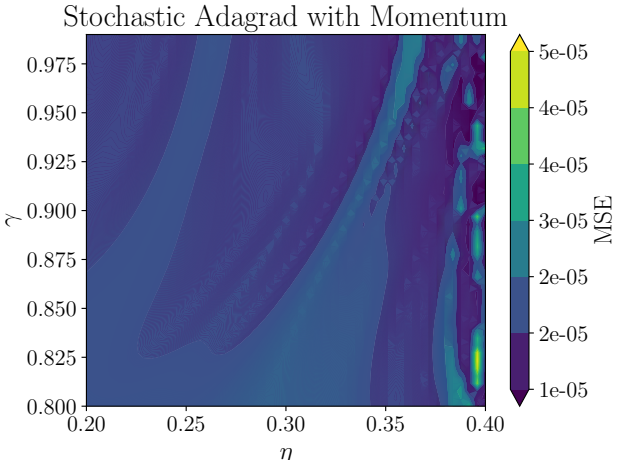

FIG. 6: Plotting the learning rate against the momentum, using stochastic AdaGrad. The batch size is set to 20.

stochastic version had an MSE of around 10^{-5}, while the plain version had an MSE of around 10^{-1}.

3. *RMSprop*

4. *Adam*

From fig. 7 and fig. 8 it seems that both versions underperformed with an MSE with around $10^{-}-1$. Looking closer at the MSE values of the stochastic variant, we found some spots with MSEs with a value around 10^{-5}.

Our feed-forward neural network demonstrated consistently strong performance on the Franke function regression task across a range of hyperparameters. As shown

Lastly, fig. 9 and fig. 10 performed similar as RMSprop, in general there where too high MSE values, but with spots in the same order of magnitude.

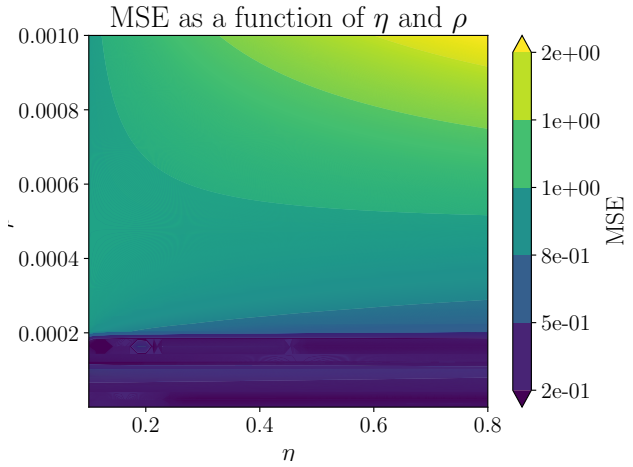

FIG. 7: Plotting the learning rate against the decay rate, using RMSprop

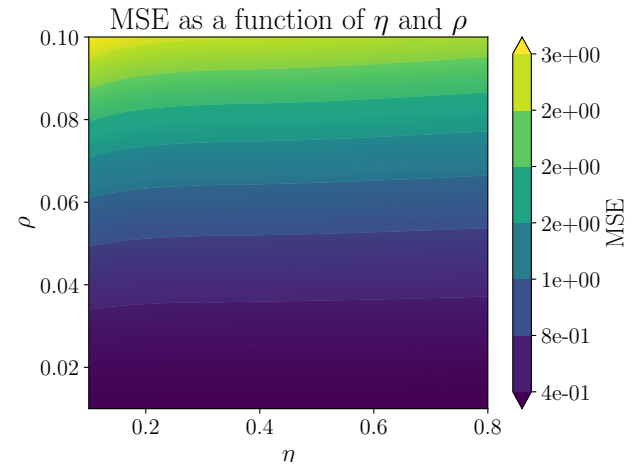

FIG. 8: Plotting the learning rate against the decay rate, using stochastic RMSprop. The batch size is set to 20.

B. Neural Networks Regression

Our analysis of feed-forward neural networks began with the Franke function regression problem, allowing us to validate our implementation and explore the effects of various hyperparameters. The results demonstrate that our neural network implementation achieves robust performance across a wide range of configurations, with optimal models achieving MSE values around 10^{-3} and R^2 scores above 0.95.

in fig. 11, the model achieves stable MSE scores around 10^{-3} and R^2 scores above 0.99 for combinations of learning rates η in the range 10^{-1} to 10^{-3} and regularization

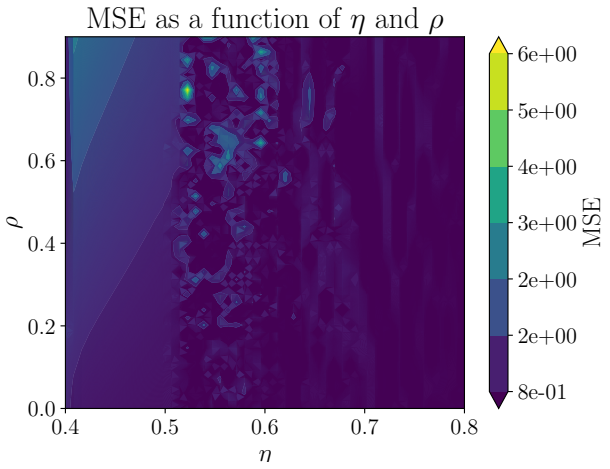

FIG. 9: Plotting the learning rate against the decay rate. We have set the same value for ρ_1 and ρ_2, using Adam

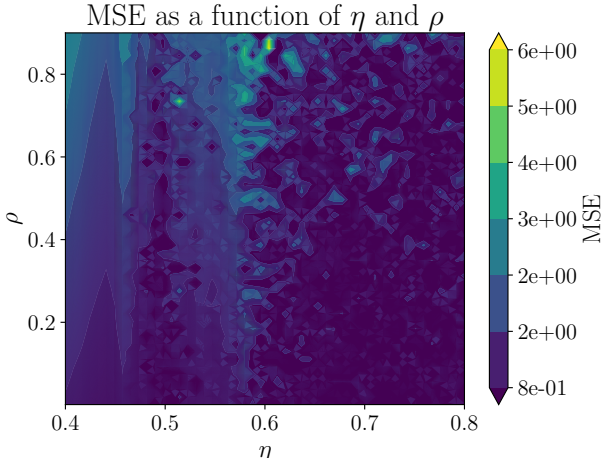

FIG. 10: Plotting the learning rate against the decay rate, using stochastic Adam. The batch size is set to 20, and the same value for ρ_1 and ρ_2 is used.

The comparison of different optimization schedulers in fig. 12 shows that all implemented methods have varying success with Adam and RMSprop performing the best. The marked convergence points indicate that both Adagrad methods are slower to converge than the other methods, while RMSprop reach its point around just 100 epochs. As the criteria for convergence is chosen somewhat arbitrarily, as describe in section III B 1, the marked crosses may not showcase where the schedulers converge, but gives insight into when the learning rates are slowing down relative to eachother as the networks are training. The figure also suggest that given enough epochs, both Adagrads, Adam and RMSprop will converge to an mse of just short of 10^{-3} and an R^2 score of 0.95. While constant learning rate and plain momentum will perform worse.

Testing different activation functions (fig. 13), reveals similar performance levels among non-linear functions. The sigmoid activation in the hidden layer performs slightly better with an MSE of 0.005 and R^2 of 0.995, but ReLU and Leaky ReLU follow closely with MSE around 0.011 and R^2 of 0.989. As expected, removing the non-linear capability of the network by using the identity function in the hidden layer results in significantly worse performance, with an MSE of ≈ 0.09 and R^2 of ≈ 0.91.

Comparing our results with the PyTorch implementation in fig. 14, we see that our model performed similarly to the PyTorch model, with an MSE around 10^{-3} and an R^2 score around 0.95. The main takeaway from this is that our models were faster to reach stable performance, wile the PyTorch model took more epochs to become stable at slightly better performance. This might suggest that the PyTorch model is more robust to overfitting, strengths λ in 10^{-3} to 10^{-5} . The model only shows significant performance degradation with the highest tested learning rate ($\eta = 0.5$), suggesting robust behavior across most of the hyperparameter space.

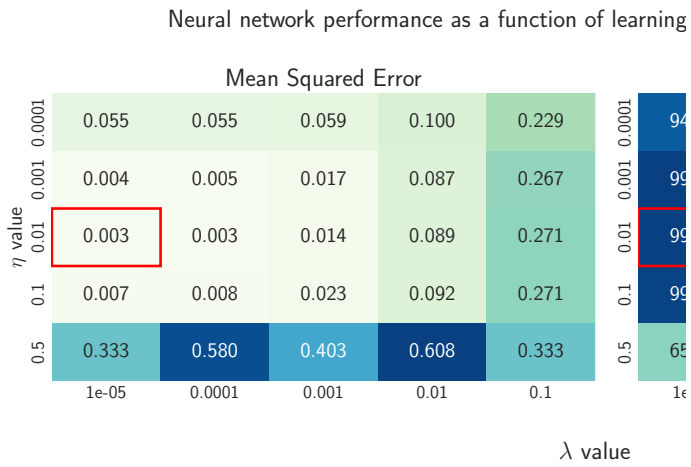

FIG. 11: The effect of learning rate and regularization strength on the Franke function regression problem. The plot shows the MSE and R^2 scores for different combinations of learning rate and regularization strength, with the optimal values highlighted in red.

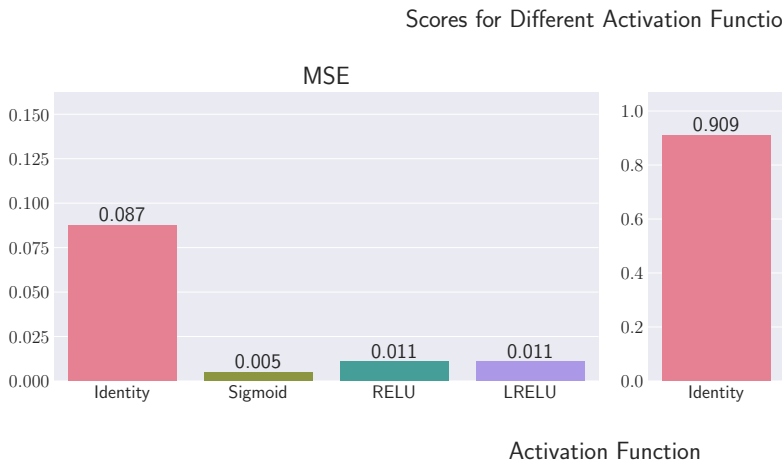

FIG. 13: The effect of different activation functions on the Franke function regression problem. The plot shows the test MSE and R^2 scores for different activation functions.

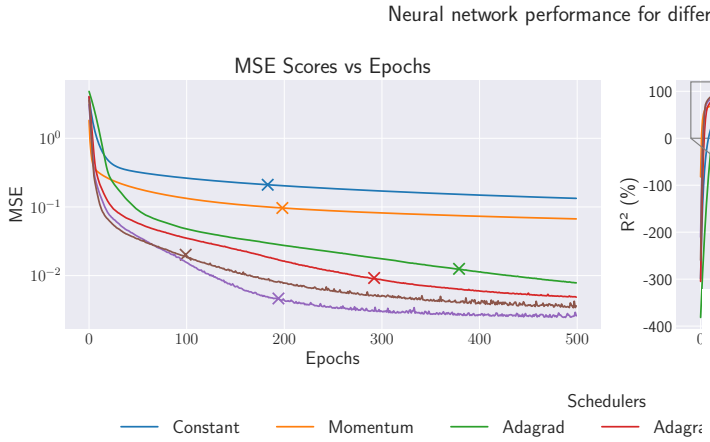

FIG. 12: The effect of different learning rate schedulers on the Franke function regression problem. The plot shows the training MSE and R^2 scores for different learning rate schedulers as a function of epochs. The schedulers MSEs are marked with a cross indicating at which epoch convergence was reached.

but at the cost of longer training times.

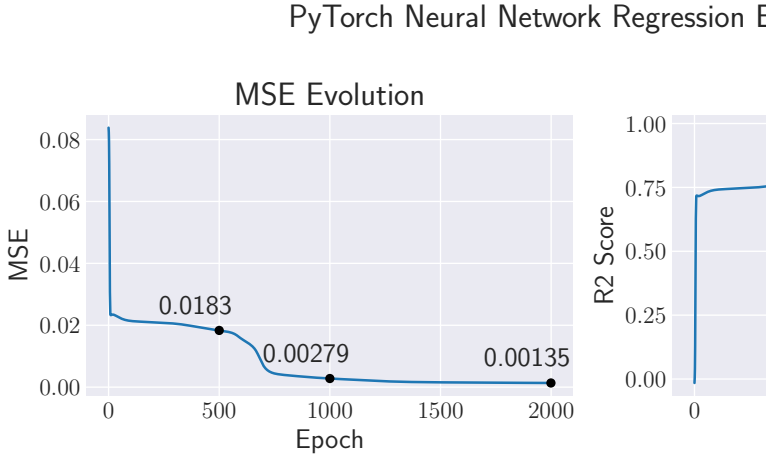

FIG. 14: PyTorch neural network regression on the Franke function. The plot shows the training and test MSE and R^2 as a function of epochs. The plots are annotated with the values at epochs 500, 1000 and 2000

C. Neural Networks Classification

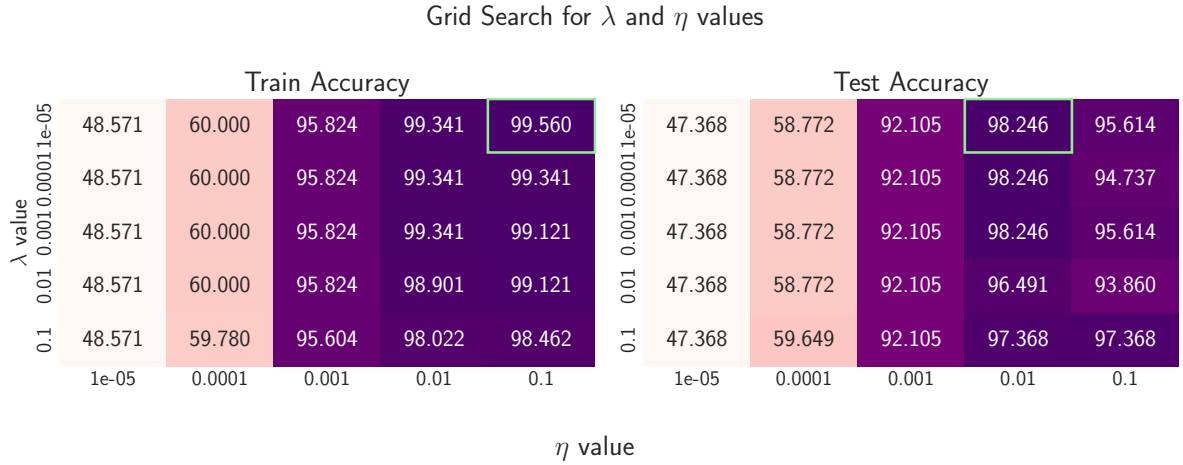

FIG. 15: Model performance for different learning rates and regularization strengths on the breast cancer classification problem. The plot shows the training and test accuracy scores for different combinations of learning rate and regularization strength. The optimal values are highlighted in green.

Our feed-forward neural network achieved strong classification performance on the Wisconsin Breast Cancer dataset. As shown in fig. 15, the model maintains high accuracy ($< 95\%$) across a broad range of hyperparameter combinations. Performance is most dependent on the learning rate, having optimal performance for η between 10^{-1} and 10^{-2}, while the regularization strength is less critical with a slight trend for better performance with lower λ values. In the figure, the optimal values for training accuracy and test accuracy differ. Since the dataset is small, the model is prone to overfitting and this could be the reason for the discrepancy.

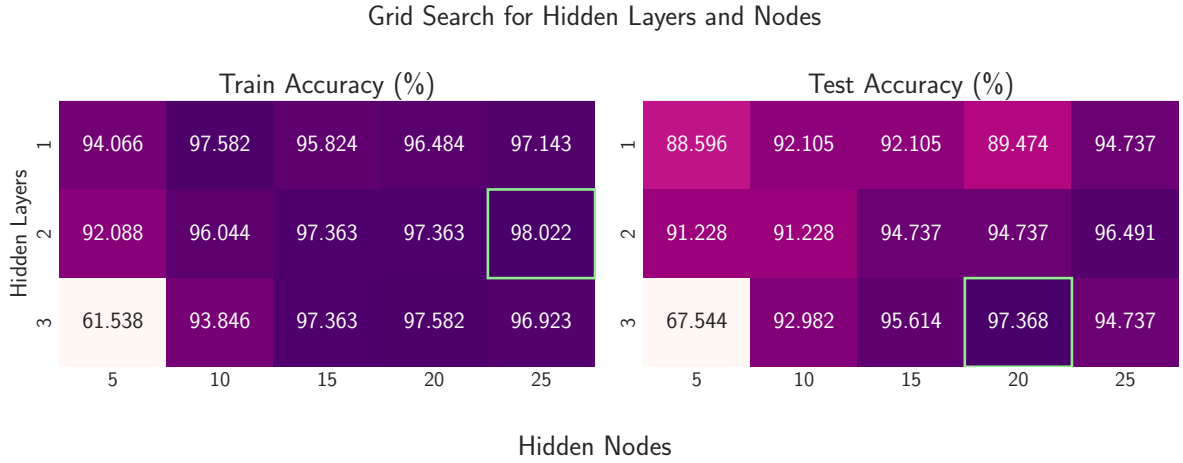

FIG. 16: Model performance for different network architectures on the breast cancer classification problem. The plot shows the training and test accuracy scores for different numbers of hidden layers and nodes. The optimal values are highlighted in green.

The exploration of network architectures demonstrates that model performance remains robust across different configurations. Networks with 2-3 hidden layers and 15-25 nodes per layer consistently achieve test accura-

cies above 96%. Notably, very small networks (5 nodes) show degraded performance, particularly with three layers where training accuracy drops to 61.5%, suggesting insufficient model capacity. Our initial assumptions that it would suffice to have a small network for this dataset are somewhat backed up by the results, but the faster runtimes do have a tradeoff in performance.

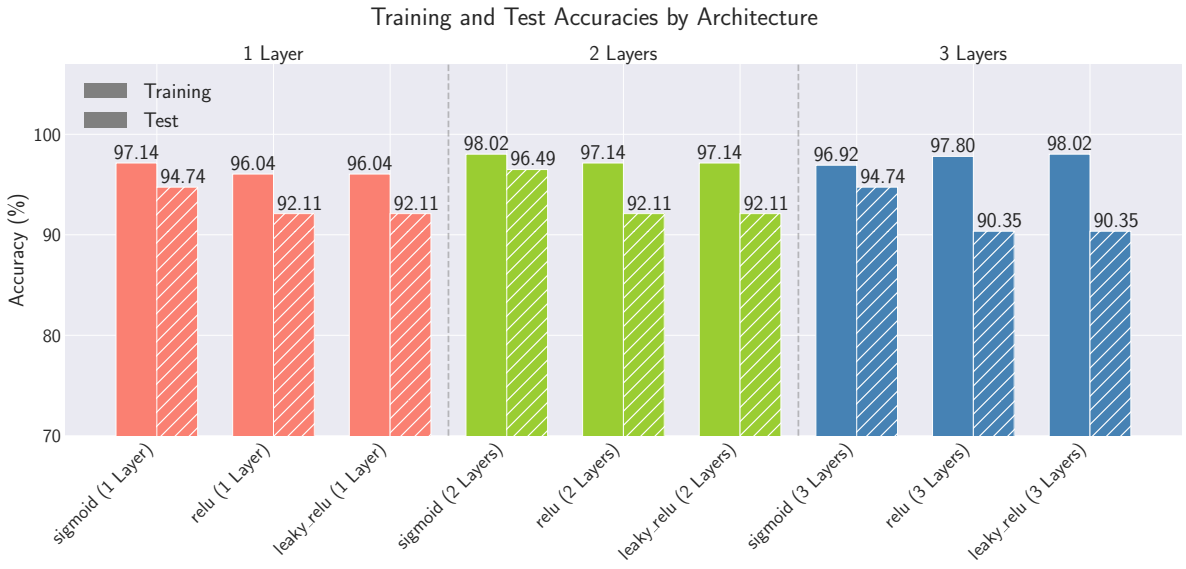

FIG. 17: Model performance for different activation functions on the breast cancer classification problem. The plot shows the training and test accuracy scores for different activation functions and numbers of hidden layers.

All tested activation functions perform similarly well, with accuracy variations within 2-3 percentage points. In single-layer configurations, sigmoid achieves slightly better performance, while ReLU and Leaky ReLU trends to better performance in deeper networks. This is only true for the training data, as all test accuracies degrade with more layers. This again suggest the danger of overfitting, and that the dataset is too small to support deep networks.

From fig. 18, we see that the PyTorch model is a more stable model, taking more time to reach high performance levels, but manages to hold a test accuracy of $< 98\%$ after 500 epochs. As our models are often scoring above 98% after only 20 epochs. A possible explanation for this is that the PyTorch model is a more advanced model, increasing the difficulty to train the network compared to our relatively simple model for a small data set as the Wisconsin Breast Cancer data.

D. Logistic Regression

Our logistic regression implementation achieves high performance on the breast cancer dataset, with test accuracies consistently above 96% across most hyperparameter combinations. fig. 19 shows that performance is robust across a wide range of learning rates and regularization strengths, with optimal test accuracy of 98.2% achieved at $\eta = 0.0001$ and a regularization strength of 0.1. The model demonstrates good generalization, with test accuracies closely matching training accuracies across the parameter space.

The confusion matrix from scikit-learn's implementation (fig. 20) shows excellent classification performance with only 3 misclassified samples out of 114 test cases. The model correctly identified 41 out of 43 malignant cases and 70 out of 71 benign cases, demonstrating balanced performance across both classes. The similar performance between our implementation and scikit-learn's validates our approach while suggesting that the classification task may be well-suited for linear decision boundaries.

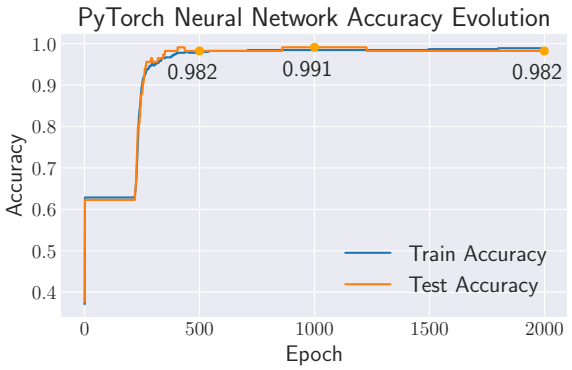

FIG. 18: PyTorch neural network classification on the breast cancer dataset. The plot shows the training and test accuracy as a function of epochs. The figure is annotated with the test accuracies at epochs 500, 1000 and 2000.

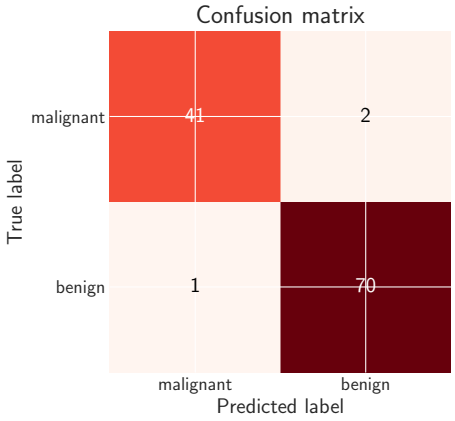

FIG. 20: Confusion matrix for the breast cancer classification problem with Skikit's Logistic Regression. The plot shows the confusion matrix for the test set, with the number of true positives, true negatives, false positives, and false negatives.

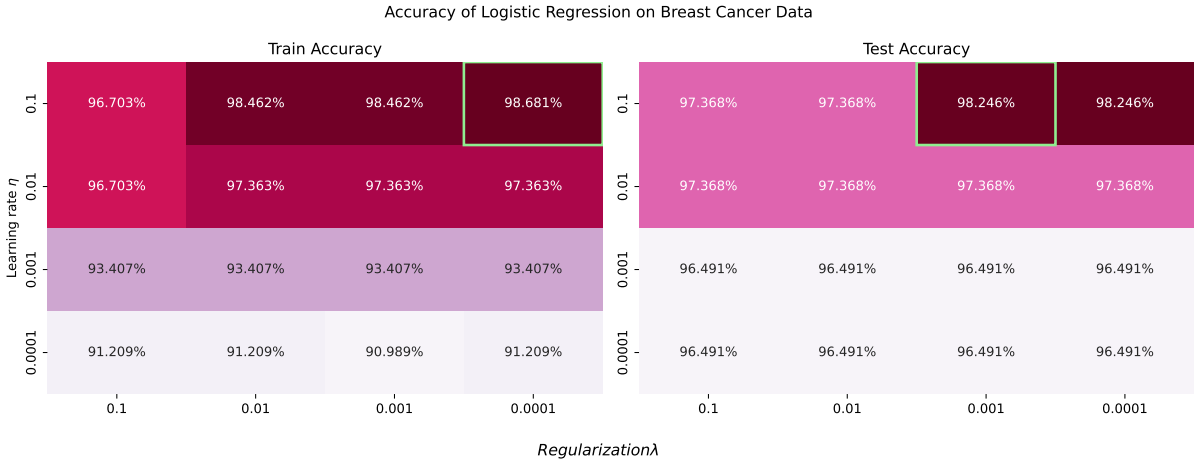

FIG. 19: Model performance for different learning rates and regularization strengths on the breast cancer classification problem using logistic regression. The plot shows the training and test accuracy scores for different combinations of learning rate and regularization strength. The optimal values are highlighted in green.

V. CONCLUSION

In our study, our regression analysis proved mostly successful. With the exceptions of finding a good range of values for the Adam and RMSprop schedulers, although individual values were found. The neural network analysis proved to be more successful, performing well across the board. Our classification model were exceptional and scored better than the PyTorch implementation. Although on such a small dataset as the breast cancer dataset, one should be critical to the amount of training possible.

VI. CODE

Link to our GitHub repository: https://github.com/Oskar-Idland/FYS-STK4155-Projects