Evaluation of Feed Forward Neural Network Performance

G. Durante, P. A. Lopez Torres, E. Ottoboni

November 4 2024

Abstract

This paper presents a rigorous evaluation of Feed Forward Neural Networks in regression and classification tasks, specifically assessing performance across synthetic and real-world datasets. We analyze FFNN effectiveness by comparing it to traditional linear models such as OLS and Ridge used in our previous article [1], using the Franke function for regression and the Wisconsin Breast Cancer dataset for classification. With an extensive hyperparameter tuning, we achieved optimal configurations yielding a MSE of 2.6×10^{-3} for OLS, outperforming Ridge regression and FFNN, which achieved an MSE of 3.6×10^{-3} and 3.397×10^{-1}. For classification, the FFNN achieved an accuracy of 0.97 on the Wisconsin dataset with Sigmoid activation, outperforming logistic regression, which attained a maximum accuracy of 0.85. However, a more sophisticated logistic regression implementation, incorporating additional hyperparameters and preprocessing, achieved comparable accuracy to the FFNN. The model's ROC curve analysis further confirmed the FFNN's superior performance with sigmoid activation function and Adam scheduler, achieving an AUC of 0.98 compared to other models with different schedulers (AdaGrad, RMSProp). When varying hidden layer configurations and activation functions (ReLU, Leaky ReLU), we observed that the Sigmoid activation consistently yielded better classification accuracy and reduced overfitting in this binary task. Additionally, learning rate and batch size optimizations for the FFNN in regression tasks revealed a critical balance point at a batch size of 64 and a learning rate of 10^{-3}, where MSE was minimized. In examining the bias-variance trade-off, smaller batch sizes were found to improve generalization at the cost of higher variance, while larger batch sizes increased bias. Our analysis underscores the FFNN's robustness across varied tasks and its ability to outperform linear models in non-linear, high-dimensional spaces. This study provides a benchmark for configuring FFNNs effectively within both regression and classification frameworks, offering a practical approach for FFNN application in complex data-driven environments.

1. INTRODUCTION

The ability to analyze current information and forecast future events is a cornerstone of modern machine learning, achieved through pattern recognition and classification. While the human brain excels at identifying complex patterns and adapting to new data, the rapid growth in data volume necessitates automated methods for pattern discovery. In response, machine learning has advanced a range of models, among which neural networks stand out as biologically inspired frameworks that mimic certain aspects of human cognitive processing.

A neural network consists of interconnected nodes analogous to neurons, where each node processes inputs and sends outputs to subsequent layers. In feedforward neural networks (FFNNs), data flows in one direction, from the input layer through hidden layers to the output layer. Although FFNNs lack the complexity of biological neural systems, they are computationally efficient and highly adaptable, making them suitable for a wide range of tasks in both classification and regression contexts.

This paper evaluates the predictive performance of FFNNs against established methods in machine learning, focusing on classification and regression tasks. For classification, we employ the Wisconsin Breast Cancer dataset to compare FFNN performance with logistic regression, aiming to optimize classification metrics, particularly true positive and true negative rates. For regression, we apply the FFNN model to the Franke function $f : \mathbb{R}^2 \to \mathbb{R}$ and assess predictive accuracy by varying polynomial orders and hyperparameters. Our aim is to understand how FFNN hyperparameters influence model efficacy, offering insights into optimized configurations for both categorical and continuous data predictions.

2. THEORY

Logistic Regression

Logistic regression is widely used for binary classification problems, where the goal is to classify data points into one of two categories. The method leverages the *sigmoid (logistic) function*, defined as:

$$S(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

The sigmoid function $S(x)$ maps any real-valued number to the range $(0, 1)$, making it useful for estimating probabilities. Suppose we have a set of p predictors, represented as a vector $\mathbf{x} = (x_1, x_2, \ldots, x_p)^T \in \mathbb{R}^p$. We define a parameter vector $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_p)^T \in \mathbb{R}^{p+1}$, where β_0 is the intercept (bias) term. The linear combination

of inputs is then:

$$t = \beta_0 + \sum_{j=1}^{p} \beta_j x_j = \boldsymbol{\beta}^T \mathbf{x} \tag{2}$$

The probability that the outcome $y = 1$ given $\mathbf{x}$ and $\boldsymbol{\beta}$ is modeled as:

$$p(y = 1|\mathbf{x}, \boldsymbol{\beta}) = S(t) = \frac{1}{1 + e^{-t}} \tag{3}$$

For binary classification, where $y \in \{0, 1\}$, the probability of $y = 0$ is simply $1 - p(y = 1|\mathbf{x}, \boldsymbol{\beta})$. Given a dataset $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ with n data points, the likelihood of observing the dataset under the logistic regression model is:

$$p(D|\boldsymbol{\beta}) = \prod_{i=1}^{n} \left(S(t_i)\right)^{y_i} \left(1 - S(t_i)\right)^{1-y_i} \tag{4}$$

where $t_i = \boldsymbol{\beta}^T \mathbf{x}_i$. Taking the logarithm of the likelihood function, we derive the log-likelihood function:

$$\begin{aligned} \log p(D|\boldsymbol{\beta}) = \sum_{i=1}^{n} \Big(& y_i \log(S(t_i)) \\ & + (1 - y_i) \log(1 - S(t_i)) \Big) \end{aligned} \tag{5}$$

To estimate the parameters $\boldsymbol{\beta}$, we maximize the log-likelihood. This is equivalent to minimizing the negative log-likelihood, defined as the *cost function*:

$$C(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \left(y_i t_i - \log(1 + e^{t_i}) \right) \tag{6}$$

The gradient of the cost function with respect to $\boldsymbol{\beta}$ is:

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\mathbf{X}^T (\mathbf{y} - S(\mathbf{t})) \tag{7}$$

where $\mathbf{X} \in \mathbb{R}^{n \times (p+1)}$ is the design matrix, $\mathbf{y}$ is the vector of observed outcomes, and $S(\mathbf{t})$ is the vector of predicted probabilities.

Convexity of the cross-entropy

It's a well-established fact that any local minimum of a convex function is also a global minimum. This means that if we find a minimum, we can be confident the solution is optimal. For a multivariate function, convexity is ensured if the Hessian matrix, composed of second partial derivatives, is positive semi-definite.

In this section, we demonstrate that the cross-entropy function is convex (dimostrations from Hastie at al. [2] and code implementations Raschka et al. [3]). The Hessian components are given by

$$\frac{\partial^2 C(\boldsymbol{\beta})}{\partial \beta_j \partial \beta_k} = -\sum_{i=1}^{n} x_j^{(i)} x_k^{(i)} p_i (1 - p_i), \tag{8}$$

where $x_j^{(i)}$ represents the j-th feature of the i-th sample, $y^{(i)}$ is the actual class label, and $p_i = p(y = 1|\mathbf{x}^{(i)}; \boldsymbol{\beta})$ is the predicted probability for class 1.

In matrix form, the Hessian $\nabla^2_{\boldsymbol{\beta}} C(\boldsymbol{\beta})$ is

$$\nabla^2_{\boldsymbol{\beta}} C(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \mathbf{x}^{(i)} (\mathbf{x}^{(i)})^T p_i (1 - p_i). \tag{9}$$

Here, $\mathbf{x}^{(i)}(\mathbf{x}^{(i)})^T$ represents the outer product of the i-th row of the design matrix $\mathbf{X}$.

A matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is positive semi-definite if $\mathbf{z}^T \mathbf{A} \mathbf{z} \geq 0$ for all $\mathbf{z} \in \mathbb{R}^n$. Applying this condition, we get

$$\mathbf{z}^T \nabla^2_{\boldsymbol{\beta}} C(\boldsymbol{\beta}) \mathbf{z} = -\sum_{i=1}^{n} (\mathbf{z}^T \mathbf{x}^{(i)})^2 p_i (1 - p_i), \tag{10}$$

$$= \sum_{i=1}^{n} \|(\mathbf{x}^{(i)})^T \mathbf{z}\|^2 p_i (1 - p_i) \geq 0, \tag{11}$$

where the final inequality holds because we are summing non-negative terms. Therefore, the Hessian of the cross-entropy is positive semi-definite, confirming that the cross-entropy function is convex.

Neural Networks

Artificial neural networks (often abbreviated as *neural networks*) are computational models capable of learning from examples. Their structure is inspired by the biological neural networks that make up animal brains. Neural networks fall within the domain of machine learning, a subfield of artificial intelligence. In the following, we will describe the exact mechanism by which these models learn.

There are many ways to construct neural networks, but we will focus on the most common architecture: *multilayer perceptrons* (MLP). MLP networks consist of layers of interconnected neurons. In an artificial network, an input value (possibly a vector) is passed into the model and propagated through each layer, being processed by each neuron in sequence. We will specifically address *feedforward neural networks* (FFNNs), meaning information flows through the network in only one direction, without loops. The entire FFNN produces an output value (which may also be a vector), so it can be thought of as a complex function mapping $\mathbb{R}^n \to \mathbb{R}^m$.

As we will see, it is possible to derive a closed-form expression for this function. Moreover, it is crucial to have an efficient algorithm for computing the gradient of the entire function with respect to any of its internal parameters. However, it is important to note that in neural networks, even if we can express certain functions (such as individual layer activations or loss functions) in closed form, the overall model introduces a high degree of nonlinearity and complexity. This complexity prevents us from deriving a closed-form solution for the entire set of parameters, and requires iterative optimization methods to find the optimal parameters (See the subsection on Gradient Descent for more information).

A neuron functions as a model for transferring information through the layers of a network. Inspired by biological neurons, an artificial neuron *"activates"* or *"fires"* when it receives a sufficiently strong signal. This neuron receives an input vector $\mathbf{p}$. If it is part of the first hidden layer, this input is simply the initial network input. In cases where other layers precede it, $\mathbf{p}$ represents outputs from neurons in the preceding layer.

Connections to neurons in previous layers are assigned weights, organized in a vector $\mathbf{w}$. For a neuron labeled k, the output p_i from neuron i in the previous layer is multiplied by the weight w_i linking it to neuron k. This weighted input contributes to the neuron's

activation as:

$$\sum_{i=1}^{N} w_i p_i = \mathbf{w}^T \mathbf{p}. \tag{12}$$

Additionally, each neuron has a bias b_k, which adjusts its likelihood to activate, regardless of the inputs. The total input is then processed by an activation function f, producing an output $\hat{p}_k$ that serves as input for neurons in subsequent layers.

Activation functions f can vary and may be linear or nonlinear, often designed to produce small values for weak inputs and saturate with large ones. Desired properties include continuity, boundedness, non-constancy, and monotonic increase (See the subsection Activation Function for more details).

In essence, a single neuron operates as:

$$\text{input} \to f(\mathbf{w}^T \mathbf{p} + b) = \tilde{p} \to \text{output}.$$

$$\mathbf{y}^k = f(W^k \mathbf{y}^{k-1} + \mathbf{b}^k) = f\left(\begin{pmatrix} w_{11}^k & w_{12}^k & \dots & w_{1N}^k \\ w_{21}^k & w_{22}^k & \dots & w_{2N}^k \\ \vdots & \vdots & \ddots & \vdots \\ w_{N1}^k & w_{N2}^k & \dots & w_{NN}^k \end{pmatrix} \begin{pmatrix} y_1^{k-1} \\ y_2^{k-1} \\ \vdots \\ y_N^{k-1} \end{pmatrix} + \begin{pmatrix} b_1^k \\ b_2^k \\ \vdots \\ b_N^k \end{pmatrix}\right) \tag{13}$$

The equation 13 defines the output $\mathbf{y}^k$ of the k-th layer of the neural network, computed as a function f applied to the weighted sum of the previous layer's outputs $\mathbf{y}^{k-1}$ and the bias vector $\mathbf{b}^k$.

An artificial neural network (ANN) comprises several neuron layers, with data moving sequentially through the input, hidden, and output layers. Each layer's neurons depend on complete outputs from the prior layer to proceed with activation calculations.

A layer consists of many neurons, each connecting to neurons in both previous and following layers. For neurons in layer k, let's denote each as n_i^k, with biases b_i^k. The weight between neuron n_i^{k-1} in the preceding layer and n_j^k is w_{ji}. The vector of all such weights for each neuron is $\mathbf{w}_i^k$. Combining all weight vectors in layer k gives a matrix $\mathbf{W}^k$:

$$\mathbf{W}^k = \begin{pmatrix} w_{11}^k & w_{12}^k & \cdots & w_{1N}^k \\ w_{21}^k & w_{22}^k & \cdots & w_{2N}^k \\ \vdots & \vdots & \ddots & \vdots \\ w_{N1}^k & w_{N2}^k & \cdots & w_{NN}^k \end{pmatrix}, \tag{14}$$

or more concisely $(\mathbf{W}^k)_{ij} = w_{ij}^k$. Biases in layer k are represented as $\mathbf{b}^k$.

The signal flow from layer $k-1$ to layer k is:

$$\mathbf{y}^k = f(\mathbf{W}^k \mathbf{y}^{k-1} + \mathbf{b}^k), \tag{15}$$

where f is applied element-wise.

Feed Forward Neural Networks (FFNN)

As previously introduced, a Feed Forward Neural Network (FFNN) is a multi-layered network of interconnected neurons where information flows in one direction—from the input layer through hidden layers to the output layer. The output a of a neuron is calculated as:

$$a = f\left(\sum_{i=1}^{n} w_i x_i + b\right) = f(z) \tag{16}$$

where $z = \sum_{i=1}^{n} w_i x_i + b$, with weights w_i and bias b as trainable parameters. Common activation functions for hidden layers include the *ReLU* function:

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ ax, & \text{otherwise} \end{cases} \tag{17}$$

where a is a small positive constant, typically set to 0.01. In FFNNs, predictions are generated by applying the *forward propagation* algorithm. The activation vector $\mathbf{a}^l$ of each layer l is given by:

$$\mathbf{a}^l = f(\mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l) \tag{18}$$

where $\mathbf{W}^l$ and $\mathbf{b}^l$ are the weight matrix and bias vector for layer l, respectively. The process is repeated for each layer until reaching the output layer.

Training an FFNN uses *backpropagation* to adjust weights and biases. For each layer l, the error term δ^l is defined as:

$$\delta^L = f'(z^L) \odot \frac{\partial C}{\partial \mathbf{a}^L} \tag{19}$$

where $\odot$ represents the element-wise (Hadamard) product. For each layer $l < L$, the error is propagated backwards as:

$$\delta^l = \left(\mathbf{W}^{l+1}\right)^T \delta^{l+1} \odot f'(z^l) \tag{20}$$

The gradients of the cost function with respect to the weights and biases are then:

$$\frac{\partial C}{\partial W_{jk}^l} = \delta_j^l a_k^{l-1}, \quad \frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{21}$$

where a_k^{l-1} represents the activation from the previous layer. These gradients are used in a gradient descent algorithm to update the weights and biases and minimize the cost function.

Activation Function

When no transfer functions are applied, meaning that $f_l(x) = x$ for every layer l, the entire network simply performs a linear transformation. To introduce non-linearity into the model, we utilize one or more activation functions from a wide range of options. These functions must be continuous and differentiable at least once (almost everywhere) to ensure that the backpropagation method outlined in the Training NN and Backpropagation subsection operates effectively.

Below is a partial list of commonly used activation functions. The *identity transformation*, defined as $f_I(x) = x$, is often employed in the output layer of regression networks. A basic (*Heaviside*) step function, given b

$$f_H(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \tag{22}$$

is sometimes utilized in the output layer of classification networks, though it is not commonly used in hidden layers due to the *vanishing gradient problem*, which makes training through backpropagation ineffective.

The *sigmoid* function,

$$f_S(x) = \frac{1}{1+e^{-x}}, \tag{23}$$

is frequently used for hidden layer activations, alongside its counterpart, the *hyperbolic tangent* function,

$$f_t(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{24}$$

The hyperbolic tangent function is essentially a rescaled version of the sigmoid, expressed as $f_t(x) = 2f_S(2x) - 1$.

Moreover, there is a family of activation functions known as *rectifiers*, which are piecewise linear functions that are currently quite popular. The basic form, the rectified linear unit (ReLU), is defined as

$$f_{\text{ReLU}}(x) = \max(0, x), \tag{25}$$

and it is mainly favored for its efficacy in training deep neural networks, which comprise many layers. Several ReLU variants exist, including the well-known *leaky ReLU*,

$$f_{\text{leaky ReLU}}(x;\alpha) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0 \end{cases} \tag{26}$$

the *noisy ReLU*,

$$f_{\text{NReLU}}(x) = \max(0, x + N(0, \sigma), \tag{27}$$

and the *exponential linear unit (ELU)*,

$$f_{\text{ELU}}(x;\alpha) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^x - 1) & \text{if } x < 0 \end{cases}. \tag{28}$$

Training NN and Backpropagation

Understanding that artificial neural networks (ANNs) can serve as *universal approximators* is not particularly useful unless we have a systematic method for determining appropriate parameters to approximate any function $g(x)$. This is where the training process becomes essential. The concept of teaching a neural network to approximate a function is straightforward and involves three main steps:

1. Assume that the input x and the corresponding correct output y are known.
2. Calculate the output $\text{NN}(x) = \hat{y}$ of the neural network and assess the *cost function* $C(\hat{y}, y)$.
3. Compute the gradient of $C(\hat{y})$ with respect to all parameters of the network, w_{kij} and b_{kj}.
4. Adjust the parameters based on the gradients to improve the estimate $\hat{y}$ of the true output y.

This training approach is termed *supervised learning* since the neural network is consistently provided with pairs of inputs (x, y)—where x is the input and y is the expected output. The cost function (also referred to as the objective or loss function) indicates how well the network performs. Generally, the output of the neural network is a vector of values, y, and the cost function evaluates the performance across all outputs.

In the network's output for each input (which may also be a vector), step (4) is conceptually straightforward but practically challenging. To effectively update the network's weights and biases, a measure of the expected change in the total output is necessary; otherwise, adjustments would be random. This requires calculating the derivatives

$$g_{kij} \equiv \frac{\partial C(\hat{y})}{\partial C(\hat{y})} \tag{29}$$

and

$$h_k \equiv \frac{\partial w_{kij}}{\partial b_i}. \tag{30}$$

The backpropagation algorithm is the most commonly used method for computing these derivatives. This method begins by feeding an input through the ANN and calculating the derivatives of the cost function concerning the weights and biases of the last layer. The network is then traversed in reverse, applying the chain rule repeatedly to determine the gradient for all neuron parameters.

Before applying the backpropagation algorithm, we must perform a forward pass through the network using an input vector $\mathbf{x} \in \mathbb{R}^{n_f}$, where n_f represents the *number of features* in the input data. For simplicity, we will initially consider the case with a single input ($n_{\text{inputs}} = 1$). During the forward pass, we calculate the activations a_l of layer l as follows:

$$a_{l_j} = f_l(z_{l_j}), \tag{31}$$

where z_{l_j} is the weighted sum of inputs from the previous layer plus the bias of layer l:

$$z_{1_j} = \sum_{i=1}^{n_f} w_{1_{ij}} x_i + b_{1_j}. \tag{32}$$

Assuming the layers contain N_l neurons, the calculated z_{l_j} for subsequent layers is given by

$$z_{l_j} = \sum_{i=1}^{N_{l-1}} w_{ij} a_{l-1_i} + b_{l_j}. \tag{33}$$

Here, $\mathbf{W}_l \in \mathbb{R}^{N_{l-1} \times N_l}$.

After completing the forward pass, we compute the cost function and its derivative with respect to the weights in the output layer $\mathbf{W}_L$:

$$\frac{\partial C}{\partial w_{L_{jk}}} = \frac{\partial C}{\partial a_{L_j}} \cdot \frac{\partial a_{L_j}}{\partial z_{L_j}} \cdot \frac{\partial z_{L_j}}{\partial w_{L_{jk}}}. \tag{34}$$

where we note that $y_j = a_{L_j}$, meaning the activations of the final layer are represented as such.

The derivative of the cost function with respect to the activations of the output layer is given by:

$$\frac{\partial C(\mathbf{W}_L)}{\partial a_{L_j}} = \frac{\partial}{\partial a_{L_j}} \left(\frac{1}{2} \sum_{i=1}^{N_O} (a_{L_i} - t_i)^2 \right) = a_{L_j} - t_j. \tag{35}$$

and

$$\frac{\partial a_{L_j}}{\partial w_{L_{jk}}} = f'(z_{L_j}) \cdot a_{L-1_k}. \tag{36}$$

We define the quantity in Equation (35) (except for a_{L-1_k}) as δ_{L_j}, which implies

$$\frac{\partial C}{\partial w_{L_{jk}}} = \delta_{L_j} a_{L-1_k}. \tag{37}$$

Applying the chain rule to δ_{L_j} allows us to derive the cost function's gradient with respect to the output layer biases:

$$\delta_L = \frac{\partial C(\mathbf{W}_L)}{\partial z_{L_j}} = \frac{\partial C(\mathbf{W}_L)}{\partial a_{L_j}} \cdot \frac{\partial a_{L_j}}{\partial z_{L_j}}. \tag{38}$$

The derivative with respect to the biases is given by:

$$\frac{\partial C(\mathbf{W}_L)}{\partial b_{L_j}} = \frac{\partial z_{L_j}}{\partial b_{L_j}} = 1. \tag{39}$$

Thus, we have determined the derivatives of the cost function with respect to both the weights and biases in the output layer, $\mathbf{W}_L$ and $\mathbf{b}_L$.

We define the quantity $\delta_{L_j} = \frac{\partial C}{\partial z_{L_j}}$, which implies

$$\frac{\partial C}{\partial w_{L_{jk}}} = \delta_{L_j} a_{L-1_k}. \tag{40}$$

Applying the chain rule to δ_{l_j} allows us to express it in terms of the error in the next layer $l+1$:

$$\delta_{l_j} = \sum_k \delta_{l+1_k} \cdot w_{l+1_{kj}} \cdot f_l'(z_{l_j}), \tag{41}$$

where $f_l'(z_{l_j})$ is the derivative of the activation function in layer l with respect to z_{l_j}.

The remainder of the backpropagation process involves iterating this equation to compute the gradients $\frac{\partial C}{\partial w_{l_{ij}}} = \delta_{l_j} a_{l-1_i}$ and $\frac{\partial C}{\partial b_{l_j}} = \delta_{l_j}$. Once the gradients are obtained, updating the weights and biases to enhance the network's performance (i.e., minimize the cost function) can be accomplished using techniques such as gradient descent, as discussed in the Gradient Descent subsection.

Algorithm 1 Backpropagation Algorithm

1: Initialize weights $W^{(l)}$ and biases $b^{(l)}$ for each layer l.
2: **while** training **do**
3: Forward pass: Compute activations $a^{(l)}$ for each layer l.
4: Compute output error:

$$\delta^{(L)} = \nabla_a C \odot \sigma'(z^{(L)}) \tag{42}$$

5: **for** layer $l = L-1$ to 1 **do**
6: Compute error for layer l:

$$\delta^{(l)} = (W^{(l+1)})^T \delta^{(l+1)} \odot \sigma'(z^{(l)}) \tag{43}$$

7: Update weights and biases:

$$W^{(l)} \leftarrow W^{(l)} - \alpha \delta^{(l)} (a^{(l-1)})^T \tag{44}$$

$$b^{(l)} \leftarrow b^{(l)} - \alpha \delta^{(l)} \tag{45}$$

8: **end for**
9: **end while**
10: Return updated weights $W^{(l)}$ and biases $b^{(l)}$.
11: *Adapted from Goodfellow et al. (2016) [4]*

Exploding Gradients and Weights Initialization

Typical activation functions are either constant or nearly constant across most of their domain, with significant changes occurring only in a small region around the origin. This characteristic leads to two issues: fully saturated neurons with input $z_j \gg 1$ or dead neurons with input $z_j \ll -1$ tend to produce very small gradients, resulting in minimal changes during training. Consequently, these neurons contribute little to the network's learning, as they merely add a constant value to the neurons in the following layer—a function already fulfilled by the bias b_{j+1}. To mitigate this issue, it is crucial to initialize the weight matrices in the network thoughtfully.

In the useful region near the origin, we can assume that the activation functions behave approximately linearly. For effective signal propagation through the network, it is essential for the mean value of the z_j inputs to be close to zero, while the variance should be approximately one. Considering an input vector $\mathbf{X}$ with n components, if the weights are initialized randomly—as done during the first forward pass—then W represents a random vector of weights W_i. Based on the assumption of linearity in the activation functions, the activation A can be expressed as:

$$A = W_1 X_1 + W_2 X_2 + \ldots + W_n X_n, \tag{46}$$

This results in a variance of:

$$\mathrm{Var}(A) = \mathrm{Var}(W_1 X_2 + \ldots + W_n X_n) = n\mathrm{Var}(W_i)\mathrm{Var}(X_i), \tag{47}$$

where we assume the inputs and weights are independent, identically distributed (i.i.d.), with a mean of zero. To ensure that the activation variance is on the order of one, we require:

$$\mathrm{Var}(W_i) = \frac{1}{n'}, \tag{48}$$

with n' being the number of weights in the subsequent layer.

Based on this reasoning, Glorot and Bengio proposed an initialization scheme for the weights with average variance:

$$\mathrm{Var}(W_i^l) = \frac{1}{n_l + n_{l+1}}. \tag{49}$$

For sigmoid or hyperbolic tangent activation functions, this variance can be achieved by initializing weights from a uniform distribution:

$$w \sim U(-r, r)$$
$$\text{with } r_{\text{sigmoid}} = \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}$$
$$\text{and } r_{\tanh} = 4 \cdot \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}. \tag{50}$$

where $U(a, b)$ denotes a uniform distribution between a and b, and n_{in} and n_{out} are the number of neurons in the current and next layers, respectively.

With rectified linear units (ReLU), the weight initialization scheme slightly changes. Since the ReLU activation function produces zero outputs for half of the real line, He et al. suggested doubling the variance of the weights to maintain a constant variance for the propagated signal:

$$\mathrm{Var}(W) = \frac{2}{n_{\text{in}}}. \tag{51}$$

This can be achieved by initializing weights from a normal distribution:

$$w \sim N(0, \sqrt{\frac{2}{n_{\text{in}}}}), \tag{52}$$

where $N(\mu, \sigma)$ denotes a normal distribution with mean μ and standard deviation σ.

This analysis of backpropagated signals leads to similar results, confirming:

$$\mathrm{Var}(W_i) = \frac{1}{n'}. \tag{53}$$

Algorithm 2 The Adam optimizer for stochastic optimization

1: At step $j = 0$, initialize $m_0 = v_0 = 0$.
2: **while** not converged **do**
3: Iterate the step counter: $j \leftarrow j + 1$.
4: Calculate the gradient: $g_j \leftarrow \nabla_\theta c(x_i, \theta_j)$.
5: Update biased first moment estimate:

$$m_j \leftarrow \beta_1 m_{j-1} + (1 - \beta_1) g_j \tag{54}$$

6: Update biased second moment estimate:

$$v_j \leftarrow \beta_2 v_{j-1} + (1 - \beta_2) g_j^2 \tag{55}$$

7: Compute the bias-corrected first moment estimate:

$$\hat{m}_j \leftarrow \frac{m_j}{1 - \beta_1^j} \tag{56}$$

8: Compute the bias-corrected second moment estimate:

$$\hat{v}_j \leftarrow \frac{v_j}{1 - \beta_2^j} \tag{57}$$

9: Update parameter vector:

$$\theta_j \leftarrow \theta_{j-1} - \alpha \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \epsilon} \tag{58}$$

10: **end while**
11: *Adapted from Goodfellow et al. (2016) [4]*
12: The constants β_1, β_2, and ϵ take appropriate default values: $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$. The initial step size α may be taken to be 0.001. The gradient squared, g_j^2, denotes the elementwise square.

Gradient Descent (GD)

Gradient Descent is a fundamental optimization algorithm widely used in machine learning and statistical modeling. Its primary purpose is to minimize a given function, often referred to as the loss or cost function, which measures the error between predicted and actual values. By iteratively adjusting the model parameters, Gradient Descent finds the optimal parameters that minimize this loss function. Gradient Descent operates by moving in the direction of the negative gradient of the loss function with respect to the parameters. The gradient indicates the direction of the steepest increase, so

moving against it leads to a decrease in the function's value. Given a loss function $J(\theta)$ where θ represents the parameters of the model, the update rule for Gradient Descent is:

$$\theta := \theta - \gamma \nabla_\theta J(\theta)$$

Here:

- θ is the parameter vector (e.g., weights in a linear regression model),
- γ is the learning rate, a hyperparameter controlling the step size,
- $\nabla_\theta J(\theta)$ is the gradient of the loss function with respect to θ

The learning rate γ determines how big a step we take towards the minimum. If γ is too large, we might overshoot the minimum; if too small, convergence can be very slow. Proper tuning of α is crucial.

Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) is a widely used optimization technique in machine learning, particularly effective for large datasets. Unlike traditional Gradient Descent, which computes the gradient of the cost function over the entire dataset, SGD approximates this gradient by evaluating it over a single randomly selected data point. Mini-batch Gradient Descent, a common variant of SGD, computes the gradient over smaller, randomly selected subsets of data called mini-batches. This approach accelerates convergence and helps avoid local minima or saddle points.

For a given mini-batch B_k, containing a subset of data points $\{\mathbf{x}_i\}$, the gradient of the cost function $C(\boldsymbol{\beta})$ with respect to the model parameters $\boldsymbol{\beta}$ is approximated as:

$$\nabla_\beta C(\boldsymbol{\beta}) \approx \frac{1}{|B_k|} \sum_{i \in B_k} \nabla_\beta C(\mathbf{x}_i, \boldsymbol{\beta})$$

where $|B_k|$ is the size of the mini-batch. The update rule for the parameters in SGD after processing each mini-batch is:

$$\boldsymbol{\beta}_{t+1} = \boldsymbol{\beta}_t - \eta \nabla_\beta C(\boldsymbol{\beta}_t)$$

Here, η is the learning rate, which controls the step size of the update. This iterative process continues until the model converges or a predefined number of epochs is reached.

By using mini-batches, SGD can achieve faster convergence and reduce memory requirements, making it especially suitable for large-scale datasets. Moreover, the stochastic nature of the updates helps the algorithm to escape local minima, potentially leading to better generalization.

Gradient descent improvements

ADAM: Adaptive Moment Estimation

While stochastic gradient descent (SGD) mitigates some issues inherent to the basic steepest descent method, it still faces challenges. The stochastic nature allows it to potentially leap over small barriers, facilitating transitions into more favorable local minima. However, SGD struggles when navigating parameter space surfaces where the gradient is significantly steeper in one direction than in others. In such cases, the iterations $\theta_i = (\alpha_i, \beta_i)$ tend to oscillate rapidly between over- and under-shooting α_i values (which have a steep gradient), while making slow progress towards the minimum in β_i (where the gradient is shallow).

To assist SGD in overcoming this issue, we can introduce a momentum term. Instead of recalculating the gradient at each iteration, we retain a portion of the change from the previous time step, effectively giving the optimization process momentum. This accelerates minimization in parameter space directions where the gradient is consistently low but directed. Additionally, it helps reduce rapid oscillations in parameter space as we approach the optimum, where steep gradients cause SGD to oscillate around the solution.

The parameter update for each mini-batch changes to:

$$\theta_{j+1} = \theta_j - [\eta \nabla_\theta c_i(x_i, \theta_{j-1})] - \gamma_j \nabla_\theta c_i(x_i, \theta_j), \quad (59)$$

with the momentum term η typically set close to 1.0, e.g., $\eta = 0.9$. The momentum term can be further enhanced using a Nesterov accelerated gradient scheme, which essentially incorporates an adaptive momentum term.

Although incorporating momentum into SGD improves the method, we still disregard much relevant information by recalculating the gradient at each iteration and discarding all previously computed gradients. Generally, we should utilize past moments from prior iterations as a guide for the current gradient step to enhance performance. This motivation leads to the Adam (Adaptive Moment Estimation) optimizer.

The Adam optimizer employs a moving, exponentially decaying average of the first and second moments of the gradient to compute individual adaptive learning rates for each parameter independently.

The bias of the estimates arises at the first iteration, as $m_0 = v_0 = 0$, inherently biasing them towards zero. The Adam scheme counters this bias by computing bias-corrected estimates $\hat{m}_j$ and $\hat{v}_j$. The Adam optimization process is summarized in Algorithm 1.

The moving average of the gradient m_j estimates the mean of the gradient, while the moving average of the squared gradient v_j estimates the uncentered variance of the gradient. At step $j = 0$, we initialize $m_0 = v_0 = 0$. The other steps are in Algorithm 2.

Autograd: Automatic Differentiation

In modern machine learning, manually computing gradients is impractical, especially for complex models. Autograd, or automatic differentiation, is a key tool that automates this process. It allows for efficient computation of gradients by constructing a computational graph during the forward pass and applying the chain rule during the backward pass. This mechanism is fundamental for implementing optimizers like SGD, Momentum, RMSprop, and Adam, as it simplifies and accelerates the gradient computation process.

RMSprop: Root Mean Square Propagation

RMSprop was developed to address the challenges of high variance in the gradients, which can slow down convergence. It modifies the standard SGD by scaling the learning rate for each parameter inversely proportional

to the square root of the exponentially decaying average of squared gradients:

$$v_t = \beta v_{t-1} + (1-\beta)(\nabla_\theta C(\theta_t))^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \nabla_\theta C(\theta_t)$$

Here, β is the decay rate, and ϵ is a small constant to prevent division by zero. This approach helps stabilize the update steps, especially in scenarios where gradients vary significantly across different dimensions.

By adjusting the learning rate dynamically, RMSprop ensures smoother convergence and handles non-stationary objectives more effectively, paving the way for more advanced optimizers like Adam.

3. METHODS

Regression

For the regression task, we consider the real multivariate Franke function $f : \mathbb{R}^2 \to \mathbb{R}$, previously analyzed using linear regression methods in our first paper [5]. Here, we compare the results obtained with a neural network to those achieved in prior experiments to evaluate if the neural network performs better. In addition, we use a simpler function to perform settling tests called Skranke function, described as follows:

$$f(x, y) = x + 2y + 3x^2 + 4xy + 5y^2 \tag{60}$$

Since the Franke function produces a continuous output, this is no longer a binary classification problem. Consequently, we cannot use the softmax activation function in the final layer. Instead, we apply ReLU (Equation 25) with parameter $a = 1$, which yields the numerical value found in the output neuron or the identity function. Furthermore, as this is a regression task, the Cross-Entropy cost function (Equation **??**is not appropriate. Instead, we employ the MSE, which is both directly interpretable and differentiable.

For the input layer, the predictors are polynomial coefficients of the independent variables x_1 and x_2 from the Franke function, structured similarly to our previous work.

With the neural network set up, we evaluate its performance on 400 data points generated from the Franke function, adding varying levels of Gaussian noise to study overfitting effects and make comparisons with linear regression methods. In this paper, we utilize a Feed Forward Neural Network (FFNN) developed based on notes by Professor Morten Hjorth-Jensen [6], which are accessible through the GitHub repository.

To assess the model during parameter tuning, we use the R^2 score and apply k-fold cross-validation for $k = 10$. This process involves splitting the data into k parts, using one part for training and the remainder for testing, repeating over n epochs.

Additionally, we calculate the MSE with k-fold cross-validation for the optimal parameters to compare against our previous results [5].

Upon determining the optimal parameters, we conduct additional analysis by plotting train and test error, for different values of each hyperparameter.

Classification

Our task is to evaluate the performance of logistic regression and a feedforward neural network in a classification task, using the Wisconsin Breast Cancer dataset compiled by Dr. Wolberg. To begin, we examine the dataset to determine appropriate predictors. The dataset includes 30 predictors, with the target variable Y, where $Y = 0$ indicates non-cancerous, and $Y = 1$ indicates cancerous.

We reference both the original article and Kaggle discussions for a more detailed feature description.

Initially, we visualize the correlation matrix for all predictors to guide our selection process. In diagram **1** we have the most significative covariates. The degree of overlap between the two distributions is crucial. Areas where the histograms intersect indicate features where malignant and benign tumors may exhibit similar measurements, which could lead to classification uncertainty. Non-overlapping regions, conversely, are regions with little to no overlap signify features that may be strong predictors of malignancy, helping o differentiate between the two classes effectively. Next, we normalize predictors with numerical values by centering (subtracting the mean) and scaling (dividing by the standard deviation). With these adjusted predictors, we apply logistic regression and the feedforward neural network on a training set, evaluating their performance on test data.

For model evaluation, we use several metrics. The simplest is *accuracy*, the proportion of correctly predicted outcomes, defined as:

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n}, \tag{61}$$

where: n is the total number of observations in the dataset. t_i represents the true label of the i-th observation. y_i represents the predicted label of the i-th observation. $I(t_i = y_i)$ is an indicator function that equals 1 if the prediction y_i matches the true label t_i, and 0 otherwise.

However, accuracy alone may be misleading if one class dominates the dataset, so we also use a confusion matrix, which includes *true negatives* (TN), *false positives* (FP), *false negatives* (FN), and *true positives* (TP). Predictions require a threshold, classifying outcomes as positive or negative based on whether they exceed the threshold.

We employ k-fold cross-validation, where the dataset is divided into K subsets. We train on $K - 1$ subsets, using the remaining subset for testing, iterating K times so each subset serves as test data once. Final accuracy is averaged across folds. To handle varying subset sizes, we normalize TN, FP, FN, and TP by dividing by the test set size. We also plot the ROC curve, using 80% of data for training and 20% for testing, varying the threshold to calculate the false positive rate $\frac{\text{FP}}{\text{FP+TN}}$ and true positive rate $\frac{\text{TP}}{\text{TP+FN}}$ at each threshold, averaged over 10 random splits, with interpolation to standardize points for comparison. The models compared are logistic regression and a neural network. For the neural network, we tested ReLU, LReLU, and sigmoid activation functions in the hidden layers, while consistently applying the sigmoid function in the final layer for classification. We used the *cross-entropy loss* (or *CostLogReg*) function, commonly employed in binary classification tasks,

as the cost function for logistic regression and the neural network.:

$$\text{cross-entropy} = -\sum_i \left(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)\right) \tag{62}$$

The neural network is trained with Adam (Algorithm 2), and we use backpropagation (Algorithm 1) to update weights with gradient calculation.

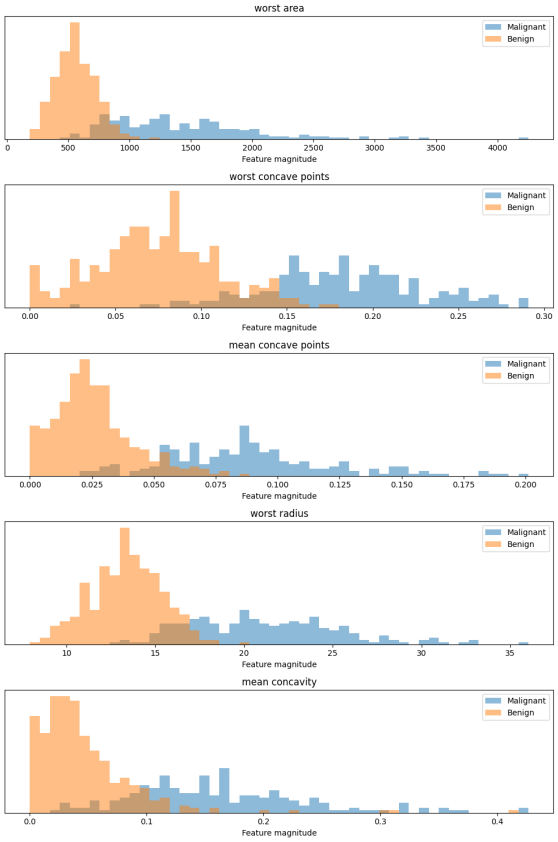

Figure 1: Distributions of five significant features from the Breast Cancer Wisconsin dataset. The overlapping distributions highlight the variance in feature values, providing insights into their relevance for tumor malignancy classification.

For comparison, as previously mentioned, we implement logistic regression using a stochastic gradient descent (SGD) approach. The model leverages mini-batch gradient descent, dividing the dataset into small batches to balance computational efficiency with stability in gradient updates. A momentum term is incorporated to accelerate convergence and reduce oscillations. Additionally, L2 regularization is included to mitigate overfitting by discouraging excessively large weight values. To ensure training efficiency, we apply early stopping to end the training process if the loss stops improving within a specified patience period, preventing unnecessary computations and helping to avoid overfitting.

Hyperparameter tuning is conducted to identify the optimal learning rate and regularization strength. Learning rate values, ranging from 10^{-5} to 10^{1}, are tested on a logarithmic scale, and each value is evaluated based on the model's test accuracy. Once the optimal learning rate is determined, it is fixed for the next stage, in which a range of regularization parameters, from 10^{-4} to 10^{1}, is tested to assess its impact on model performance. For each combination, accuracy on the test set serves as the evaluation metric, with early stopping consistently applied across experiments to ensure reliable comparisons. The resulting analysis is visualized to illustrate how learning rates and regularization terms influence the model's accuracy (see the subsection on Classification results), providing insight into the sensitivity of logistic regression to these hyperparameters. This comprehensive approach collectively optimizes our logistic regression implementation for robust classification performance.

4. RESULTS

Gradient Descent

We implemented Gradient Descent (GD) as the primary optimization method to approximate a function, beginning with a 1D test case and later expanding to a 2D scenario. Our goal was to explore the impact of various learning rates, γ, on the convergence and accuracy of the algorithm. To rigorously test our approach, we employed the Franke function, known for its non-linear characteristics, making it an excellent benchmark for evaluating optimization techniques in approximating complex surfaces. In our initial experiment, we applied GD with a fixed learning rate of $\gamma = 0.01$, iterating the procedure for 1000 steps. This setup yielded a test Mean Squared Error (MSE) of 1.936 for the Ordinary Least Squares (OLS) model and 1.468 for the Ridge regression model. These results suggest that while the models captured some aspects of the Franke function's structure, the choice of a constant and non-optimized learning rate led to a significant generalization error. To improve the model's performance, we conducted a sensitivity analysis by varying the learning rate γ from 0.01 to 0.5.

Model	Optimal γ	Test MSE
OLS	0.324	0.559
Ridge	0.01	0.426

Table 1: Minimum Test Mean Squared Error (MSE) and Optimal Learning Rate for OLS and Ridge Regression

These findings highlight the critical role of learning rate selection in enhancing the stability and effectiveness of Gradient Descent, allowing the model to better approximate the target surface and reduce generalization error. The results underscore the importance of tuning the learning rate in optimization algorithms, especially when dealing with non-linear functions like the Franke function. Proper tuning not only improves convergence but also ensures that the model generalizes well to unseen data.

The addition of momentum in gradient descent was intended to improve convergence stability and speed, particularly on error surfaces with steep slopes and oscillations. Momentum acts as an accelerator along the main descent directions, reducing the effect of gradient oscillations and helping the model converge faster. We tested gradient descent with various values for the momentum factor δ (ranging from 0.3 to 1.0) and learning

rates γ (from 0.01 to 0.5) to determine the optimal configuration.

Model	Optimal γ	Optimal δ	Test MSE
OLS	0.185	0.850	0.180
Ridge	0.5	0.6	0.085

Table 2: Optimal Parameters and Test Mean Squared Error (MSE) for OLS and Ridge Regression with epoch = 1000

The inclusion of momentum notably enhanced the model's convergence, especially in regions of the Franke function with rapid variation. The reduced error illustrates the effectiveness of momentum in accelerating the learning process, facilitating faster and more stable convergence to the target surface. This configuration not only reduced overall training time but also improved model accuracy, resulting in a surface that more closely approximated the original Franke function.

Stochastic Gradient Descent (SGD) was tested in several scenarios to analyse the influence of key hyperparameters such as learning rate, batch size and the use of advanced optimisation techniques such as momentum. The first step was to determine the best combination of gamma and delta parameters, which were then held constant for subsequent analyses. The behaviour of the Mean Squared Error (MSE) was then studied as the batch size and the eta parameter were varied, with the aim of finding the optimal combination of the two parameters. The main results show that varying the batch size significantly affects the parameter updates and the error values. Small batch sizes, such as 10 and 20, produce noisier updates, characterised by strong oscillations around the minimum of the objective function, and lead to higher mean square error (MSE) values. As batch sizes increase, e.g. to 50 and 70, greater stability of the update process is observed, resulting in reduced oscillations and improved accuracy. In particular, a batch size of 70 represents a good compromise between noise and stability, with an MSE of 30.30 in the test set for OLS and 0.022 for Ridge. The last parameter variation involved the combined optimisation of number of epochs and batch size, with the following results:

	OLS	Ridge
Optimal Batch Size	85	95
Optimal Epoch	300	800
Test Error	0.0761	0.0270

Table 3: Optimal parameters for OLS and Ridge models with different Batch size and epoch

With the optimal parameters, the convergence of the three main methods (GD, SGD, ADAM) was also evaluated at different epochs, as shown in the graphs for the OLS figure **2** and Ridge models figure **3**.

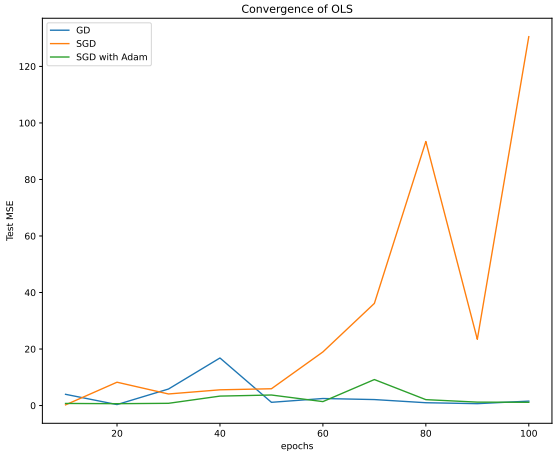

Figure 2: OLS model convergence: MSE test obtained with all methods. Parameters for OLS: $\gamma = 0.255$, $\delta = 1.0$, $\eta = 2.033$ and batch size = 85.

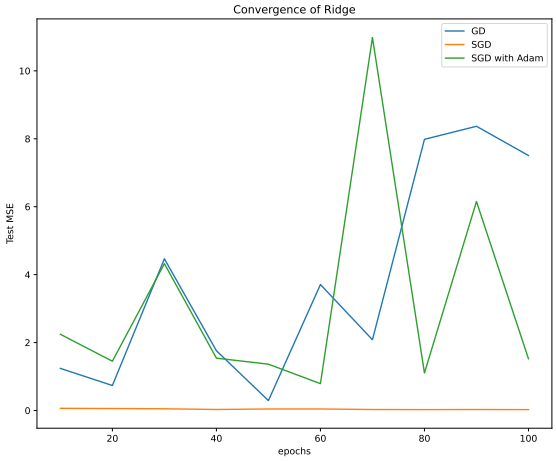

Figure 3: Ridge model convergence: MSE test obtained with all methods. Parameters for Ridge: $\gamma = 0.15$, $\delta = 0.400$, $\eta = 1.066$ and batch size = 95.

Alongside traditional optimization methods, we also tested Adagrad, RMSprop, and Adam algorithms, each of which improves upon standard gradient descent by dynamically adjusting the learning rate during training. Our objective was to compare these methods to determine which one was most effective in approximating the Franke function while minimizing the MSE. For each method, we varied the mini-batch size from 10 to 100 and recorded the corresponding test errors. The results showed that Adam yielded the lowest test error, followed by RMSprop, with Adagrad having the highest test error. Adam, in particular, demonstrated superior adaptability to local variations in the Franke function, striking an ideal balance between stability and learning accuracy, especially in regions with high curvature. The comparison confirmed that Adam was the most effective optimizer, showcasing its robustness and adaptability to irregular gradients. Adam's combination of adaptive learning rates with first and second moment estimates made it particularly well-suited for approximating the Franke function, significantly reducing test error compared to other methods. RMSprop also performed well,

offering slightly less stability than Adam but more than Adagrad, which was less suitable for advanced training phases.

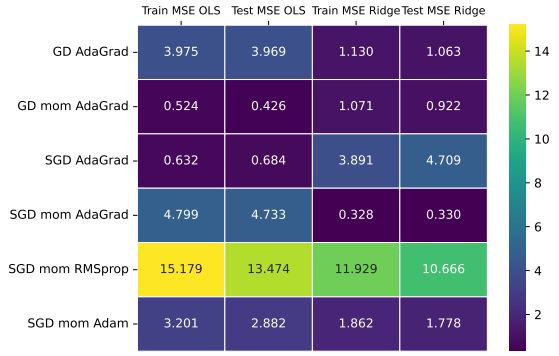

Figure 4: MSE train and test for OLS and Ridge obtained with all the methods. Parameters for OLS: $\gamma = 0.255$, $\delta = 1.0$, $\eta = 2.033$ and batch size $= 85$. Parameters for Ridge: $\gamma = 0.15$, $\delta = 0.400$, $\eta = 1.066$ and batch size $= 95$.

The same analysis was conducted using the *gd_autograd* class, and the results were consistent. You can view the results on GitHub.

Regression

In this section, we analyze the performance of various machine learning models applied to the approximation of the Franke Function, a synthetic function commonly used for benchmarking regression techniques. The selected models: OLS regression, Ridge regression, and a Feedforward Neural Network (FFNN), are evaluated based on their ability to minimize the MSE on a test dataset. The theoretical foundations and implementation strategies for these models were guided by key resources, including Hastie et al. [2] for statistical learning methods and Goodfellow et al. [4] for neural network design principles.

For dataset generation, we compute the Franke Function over a discretized interval $[0, 1]$ using a step size of 0.05. Polynomial feature expansion is used to enhance the model's capacity to capture non-linear interactions, with a polynomial degree of 5. The dataset is split into training and testing subsets in an 80/20 ratio.

The linear regression models, OLS and Ridge, are implemented with Scikit-Learn [7], with Ridge incorporating a regularization parameter $\alpha = 10^{-4}$ to constrain coefficient magnitude. The OLS model achieves an MSE of 0.0026, indicating robust performance in approximating the Franke Function with minimal error. Ridge regression, while comparable, yields an MSE of 0.0036, suggesting that, at the selected α value, regularization minimally impacts performance.

The FFNN model is structured as a three-layer architecture with dimensions $(500, 250, 1)$, utilizing the sigmoid activation function in hidden layers and the identity function in the output layer. We optimize using the Adam algorithm with learning rate $\eta = 10^{-3}$, $\rho = 0.9$, and $\rho_2 = 0.999$, and incorporate L_2 regularization with $\lambda = 10^{-4}$ to mitigate overfitting. The model is trained over 5000 epochs with a batch size of 1, aiming for precise convergence. However, the FFNN's resulting MSE of 0.3397 suggests that it may be overfitting or failing to converge adequately within the current setup, indicating a need for further tuning or architectural refinement.

In summary, as we can se in **??** the linear regression models outperform the FFNN on this dataset, with OLS yielding the most accurate results. This comparison underscores that, for this specific task, the FFNN's complexity may not be justified, as the linear approaches yield lower error with fewer computational resources.

Model	Parameters	MSE
OLS	No regularization	0.0026
Ridge	$\alpha = 10^{-4}$	0.0036
FFNN	$\eta = 10^{-3}$, $\rho = 0.9$, $\rho_2 = 0.999$, $\lambda = 10^{-4}$	0.3397

Table 4: Model Performance Comparison

To examine the effect of the learning rate on model performance, we conducted an expanded analysis using the FFNN with a range of learning rates, η, from 10^{-4} to 0.5. For each learning rate, we calculated the MSE, bias, and variance to assess the model's fit and generalization capacity. Figure **6** illustrates the relationship between learning rate and these performance metrics on a logarithmic scale for better visualization.

As the learning rate increases, the model exhibits distinct behavior across the metrics. The bias remains high and increases with larger η, suggesting that the model's capacity to fit the underlying data structure diminishes as the learning rate becomes excessively high, likely due to underfitting. Conversely, variance decreases sharply with increasing η, indicating that the model predictions become less responsive to fluctuations in the data, stabilizing around lower values. This trend reflects the classic trade-off between bias and variance: a higher learning rate reduces variance but at the cost of increased bias, leading to a poor overall fit as indicated by the elevated MSE values.

Model	MSE
OLS	7.8×10^{-30}
Ridge	2.31×10^{-8}
FFNN	0.512

Table 5: MSE for OLS, Ridge, and FFNN models on the Franke function. Parameters: OLS, Ridge ($\alpha = 10^{-4}$), FFNN (Adam optimizer, $\eta = 10^{-3}$, decay rates $0.9, 0.999$, $\lambda = 10^{-4}$, 5000 epochs).

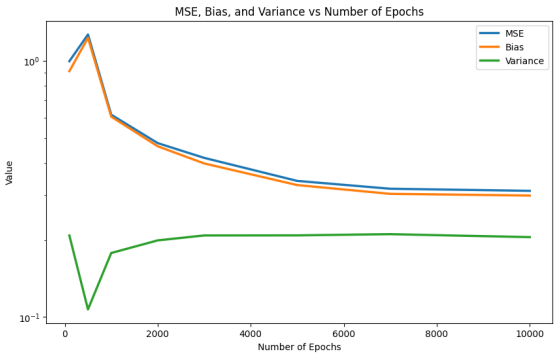

Figure 5: MSE, Bias, and Variance vs. Number of Epochs for an FFNN model with two hidden layers (500, 250 nodes) trained on the Franke function. Increasing epochs reduces MSE and bias, with variance stabilizing, indicating model convergence.

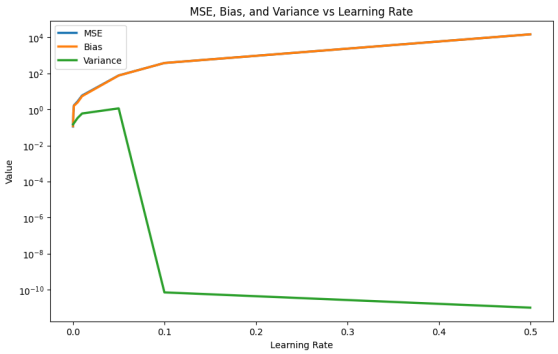

Figure 6: MSE, Bias, and Variance vs. Learning Rate for an FFNN model with two hidden layers (500, 250 nodes) trained on the Franke function. Lower learning rates (0.001) achieve optimal MSE, balancing bias and variance, while higher rates increase bias, indicating underfitting.

To assess the FFNN's sensitivity to learning rate, we conducted an expanded analysis using a range of learning rates, η, from 10^{-4} to 0.5. For each learning rate, we calculated the MSE, bias, and variance to evaluate the model's fit and generalization properties. Figure **5** presents the relationship between the learning rate and these metrics on a logarithmic scale. The analysis demonstrates that increasing η raises the bias while reducing variance, indicating a shift towards underfitting at higher learning rates. This pattern reflects the classic bias-variance trade-off, where the model's responsiveness to data variations decreases as it becomes biased toward simplistic approximations.

Furthermore, we conducted an investigation into the effect of the number of epochs on FFNN performance. Using a fixed learning rate of $\eta = 10^{-3}$, we varied the epochs from 100 to 10000. Figure **5** shows the MSE, bias, and variance as functions of epoch count. The results reveal a significant reduction in MSE and bias as the number of epochs increases, with both metrics stabilizing around lower values for larger epochs, suggesting effective convergence. The variance remains relatively low throughout, indicating that extended training allows the model to reduce bias without introducing excessive variance. These findings suggest that, for the current task, increasing the number of epochs benefits model performance by facilitating more complete convergence.

To investigate the impact of batch size on the FFNN's performance, we analyzed training and test MSE, along with bias and variance, across various batch sizes ranging from 1 (stochastic gradient descent) to the full-batch size. The goal of this analysis is to assess how different batch sizes influence the model's fit and generalization capabilities.

Figure **7** shows the training and test MSE for each batch size, plotted on a logarithmic scale for clarity. The results indicate that as the batch size increases, both training and test MSE initially decrease, reaching an optimal point around a batch size of 64. Beyond this value, the MSE begins to increase, suggesting that excessively large batch sizes may degrade the model's performance. Smaller batch sizes appear to facilitate better generalization, while excessively large batches lead to higher MSE, possibly due to less effective optimization and reduced model adaptability.

In Figure **8**, we further analyze the effects of batch size on bias, variance, and MSE. The results show that smaller batch sizes lead to higher variance, which decreases as the batch size increases. However, bias tends to increase as batch size grows, illustrating the classic bias-variance trade-off. The MSE reaches its lowest point when the balance between bias and variance is optimal, around a batch size of 64, beyond which the bias starts to dominate, resulting in a higher overall error.

The batch size analysis presented in Figures **7** and **7** provides insights into the trade-offs involved in selecting an appropriate batch size for training FFNNs. Smaller batch sizes, while yielding higher variance, allow the model to adapt more effectively to the underlying data patterns, resulting in lower MSE when optimally balanced with bias. As batch size increases, variance decreases but bias grows, leading to underfitting in cases where batch sizes are too large.

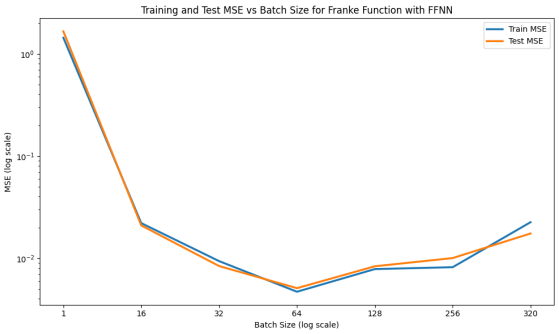

Figure 7: Training and Test MSE vs. batch size for an FFNN model with two hidden layers (500 and 250 nodes) trained on the Franke function. Optimal MSE is achieved with batch sizes between 32 and 128, balancing training stability and generalization.

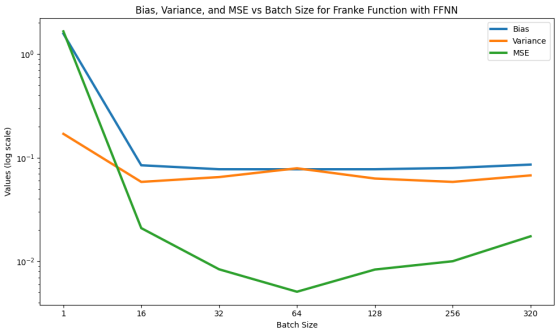

Figure 8: Bias, Variance, and MSE vs. batch size for an FFNN model with two hidden layers (500 and 250 nodes) trained on the Franke function. Smaller batch sizes increase bias but lower variance, while intermediate sizes (32 to 64) balance bias and variance, minimizing MSE.

To explore the influence of hidden layer configuration on the FFNN's performance, we experimented with multiple architectures, varying the number of layers and nodes per layer. This investigation aims to assess how the depth and complexity of the network impact its ability to approximate the Franke function, focusing on the MSE, bias, and variance.

Figure **9** illustrates the relationship between different hidden layer configurations and the MSE, bias, and variance on a logarithmic scale. The results reveal that configurations with moderate complexity (e.g., two to three layers with fewer nodes) yield lower bias and variance. In contrast, as the depth and number of nodes increase, the bias and variance exhibit fluctuations, indicating the model's sensitivity to overfitting when excessive capacity is introduced. The optimal configuration appears to strike a balance between model flexibility and stability, minimizing both bias and variance.

In Figure **10**, we examine the training and test MSE across the hidden layer configurations. Smaller and moderately sized networks perform better in terms of test MSE, showing that they generalize effectively on unseen data. Larger configurations, while achieving lower training error due to increased capacity, display higher test MSE, reflecting overfitting. This trend suggests that, for the Franke function, increasing network complexity beyond a certain point provides diminishing returns in terms of generalization. The analysis of hidden layer configurations, underscores the trade-offs associated with network depth and width. As the number of layers and nodes increases, the FFNN gains representational capacity but also becomes prone to overfitting, evidenced by rising test MSE and variable bias-variance patterns.

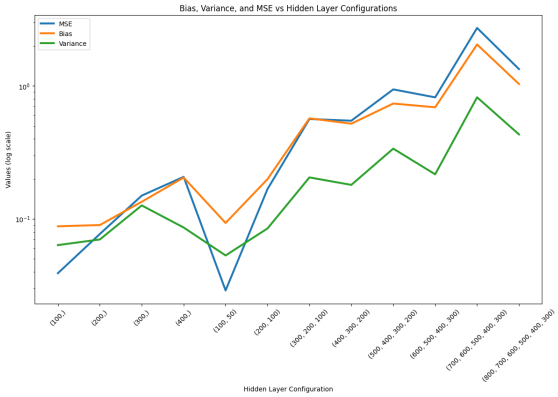

Figure 9: Bias, Variance, and MSE vs. hidden layer configurations for an FFNN model trained on the Franke function. Deeper networks generally increase variance and MSE, while shallower configurations reduce bias. Optimal configurations balance bias and variance, minimizing MSE.

Figure **11** shows the training and test MSE across a range of polynomial degrees on a logarithmic scale for both axes. The results reveal an initial decrease in MSE as the polynomial degree increases, reflecting an improved fit to the underlying data. However, as the degree continues to rise, the test MSE begins to fluctuate significantly, while the training MSE remains comparatively stable. This pattern indicates overfitting at higher polynomial degrees, where the model starts capturing noise rather than the true data structure, leading to poor generalization on the test set.

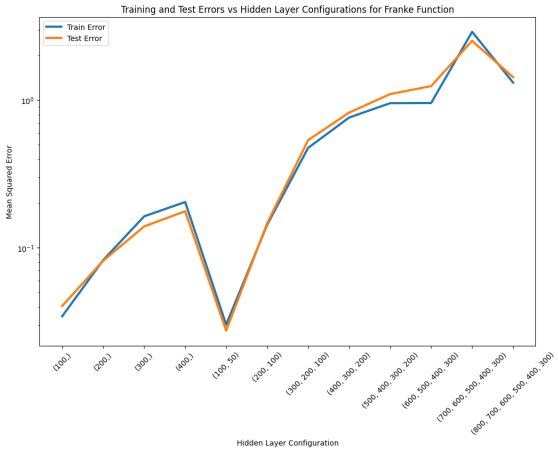

Figure 10: Training and Test MSE vs. hidden layer configurations for an FFNN model trained on the Franke function. Increasing network depth raises MSE, indicating overfitting, while optimal configurations balance complexity to minimize error.

The optimal polynomial degree appears to be in the

lower range, where the test MSE stabilizes before the onset of high variance. Beyond this point, the high polynomial complexity introduces unnecessary noise sensitivity, reflected in erratic test error behavior. This study underscores the trade-off between model complexity and generalization, demonstrating that moderate polynomial degrees yield more robust performance.

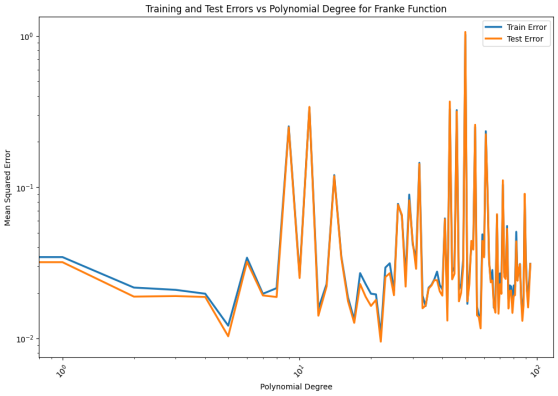

Figure 11: Training and Test MSE vs. polynomial degree for an FFNN model with a single hidden layer (5 nodes) trained on the Franke function. Higher polynomial degrees increase test error variability, indicating overfitting, while lower degrees minimize both training and test errors.

To visualize the FFNN's predictions for a complex polynomial degree, we generated a 3D surface plot of the predicted values for the Franke function at polynomial degree 27. This plot illustrates how the model approximates the target function across the input space, providing a spatial representation of the prediction accuracy and structure.

Figure **12** displays the 3D surface plot of the predicted values. As seen in the figure, the model captures the general shape and curvature of the Franke function, demonstrating the FFNN's ability to approximate complex non-linear surfaces at a higher polynomial degree. However, higher polynomial degrees introduce the risk of overfitting, as the model may begin to fit noise within the data rather than the underlying function structure.

This visualization highlights both the strengths and limitations of polynomial expansion in design matrices. While increased degrees allow the model to capture finer details, they can also lead to excessive sensitivity to data variations. The 3D plot provides an intuitive understanding of this balance, showing how well the FFNN aligns with the expected functional form of the Franke function.

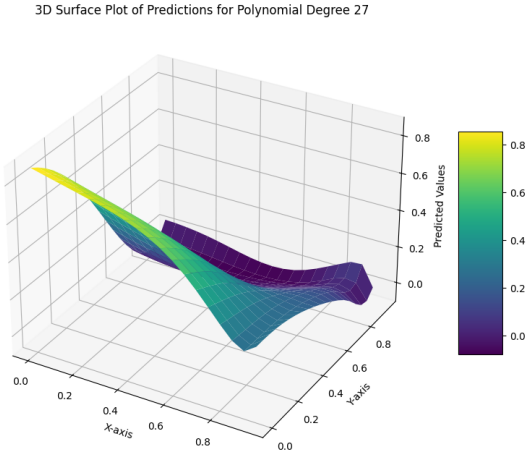

Figure 12: 3D surface plot of FFNN model predictions for the Franke function with polynomial degree 27. The model has two hidden layers (20 and 5 nodes) and uses Sigmoid activation. The plot illustrates the model's capability to capture complex patterns.

For a regression task, our objective is to minimize the discrepancy between predicted values and actual target values. To achieve this, we choose a cost function that effectively measures this discrepancy and optimizes the learning process. For regression, the OLS cost function is a natural choice, as it minimizes the mean squared error between the model's predictions and target values. This property makes it well-suited for regression tasks where we aim for precise approximations of continuous target values. The quadratic nature of OLS emphasizes larger errors, encouraging the model to minimize significant discrepancies effectively.

When initializing the biases, we aim to prevent extreme values that could lead to unstable gradient behavior in the initial stages of training. Therefore, setting biases to small random values close to zero or a small positive constant is a common and effective approach. In the provided code, biases are initialized with a small positive constant (e.g., 0.01). This initialization avoids bias dominance in early activations while allowing flexibility in the learning process. By initializing in this way, we ensure that the model does not introduce any unintended directionality in the initial optimization steps.

For the activation function of the output layer, the identity function is the most appropriate choice in regression tasks.

the identity function leaves the output values unchanged, which is crucial in regression tasks where output values are continuous and may vary across a wide range. Unlike classification tasks that often require bounded outputs (e.g., using a sigmoid for probabilities between 0 and 1), regression tasks benefit from an unbounded activation function that allows the model to approximate the full range of possible values without restrictions.

Classification

As previously introduced, we evaluate the performance of various models and activation functions on a classification task using the Wisconsin Breast Cancer dataset.

This dataset contains features extracted from digitized images of fine needle aspirates of breast masses, aiming to classify tumors as either benign or malignant. We employed Feedforward Neural Networks (FFNNs) with Sigmoid, ReLU, and Leaky ReLU activation functions, as well as Logistic Regression as a baseline. Model evaluation metrics include accuracy, sensitivity, specificity, and the confusion matrix. The implementation and theoretical explanations presented here were made possible thanks to foundational resources provided by Hastie et al. [2], Bishop [8], and Goodfellow et al. [4], which offer comprehensive insights into statistical learning, pattern recognition, and deep learning techniques.

In the neural network implementation, the data was preprocessed using Min-Max scaling to normalize the features between 0 and 1. We performed a train-validation split, with 80% of the data used for training and 20% for validation. The FFNN architecture consisted of an input layer with 30 nodes (matching the number of features in the dataset), followed by two hidden layers with 100 and 30 nodes, respectively, and a single output node. Training was conducted over 10-fold cross-validation.

Figure **13** shows the confusion matrices for each activation function and the logistic regression model. The Sigmoid activation function yielded the best performance, with the highest number of true positive and true negative classifications, closely followed by ReLU and Leaky ReLU. Logistic regression, while effective, had the highest number of misclassifications among the models tested (we later developed a more sophisticated version that performed exceptionally well).

In addition to the confusion matrices, we evaluated each model based on overall accuracy. Table **??** summarizes the accuracy scores for FFNNs with Sigmoid, ReLU, and Leaky ReLU activation functions, as well as Logistic Regression. The FFNN with Sigmoid activation achieved the highest accuracy of 0.97, indicating its superior performance in correctly classifying benign and malignant tumors. Both ReLU and Leaky ReLU activations followed with an accuracy of 0.89, showing competitive but slightly lower effectiveness compared to Sigmoid. Logistic Regression, while interpretable, exhibited the lowest accuracy of 0.85, reflecting its limitations in handling non-linear patterns in the dataset.

Model	Accuracy
FFNN Sigmoid Activation	0.97
FFNN ReLU Activation	0.89
FFNN Leaky ReLU Activation	0.89
Logistic Regression	0.85

Table 6: Performance Metrics for Various Models on the Wisconsin Breast Cancer Dataset

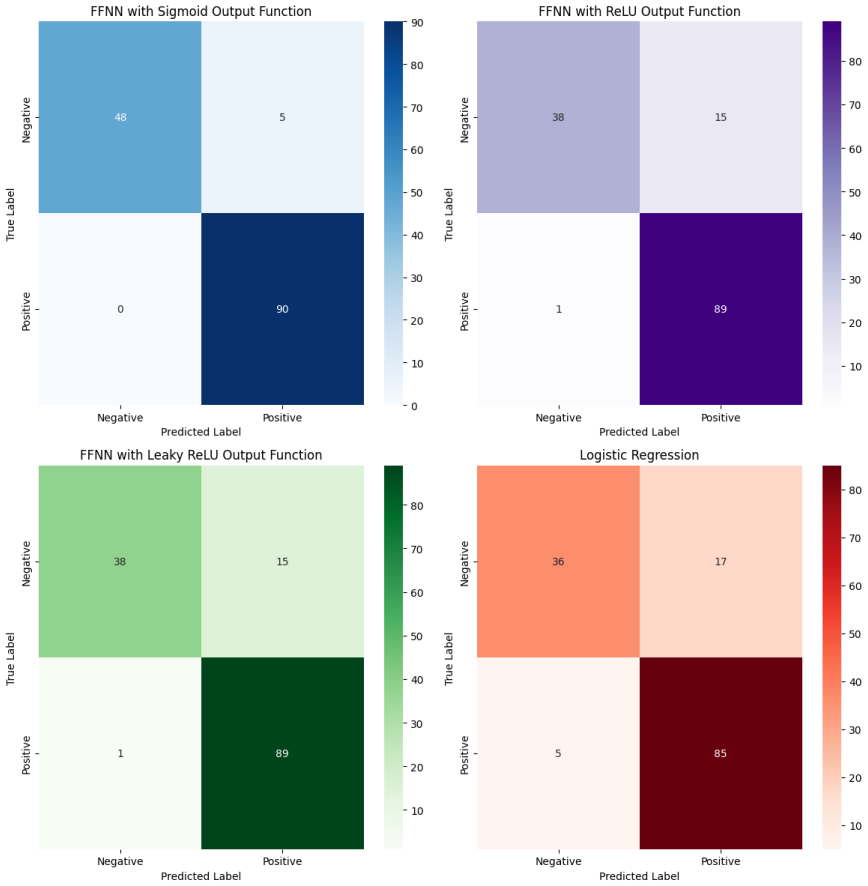

Figure 13: Confusion Matrices for different models and output functions: Sigmoid, ReLU, Leaky ReLU, and Logistic Regression. Sigmoid achieves the best performance with 0 false positives and 5 false negatives. ReLU and Leaky ReLU perform comparably with slightly more false negatives and positives, while Logistic Regression has the lowest accuracy with the highest number of false predictions.

The models were assessed based on their ability to differentiate between benign and malignant tumors, with ROC curves and Area Under the Curve (AUC) scores serving as primary metrics for comparison.

Figures **14**, **15**, **16** illustrates the ROC curves for each model. Sigmoid activation achieved the highest AUC of 0.98, demonstrating superior classification accuracy and distinguishing ability. ReLU and Leaky ReLU performed comparably, with AUC values of 0.92 and 0.91, respectively. Logistic Regression, however, exhibited a significantly lower AUC, indicating limited capacity to capture complex patterns in the data compared to non-linear activations.

For training optimization, we used the Adam algorithm with a learning rate $\eta = 10^{-3}$, which combines adaptive learning rates with momentum, thus balancing convergence speed and stability. Regularization, controlled by the parameter λ, helped to mitigate overfitting by penalizing large weights, particularly beneficial for neural networks with multiple layers.

The chosen architecture, hyperparameters, and regularization strategies allowed the model to effectively learn from the data without overfitting. This configuration, especially the depth of the network with two hidden layers and the regularization parameter, balanced the complexity needed to capture feature interactions in this medical dataset. Overall, the FFNN with Sigmoid activation and Adam optimizer, coupled with cross-validation, achieved robust performance in classifying breast cancer samples.

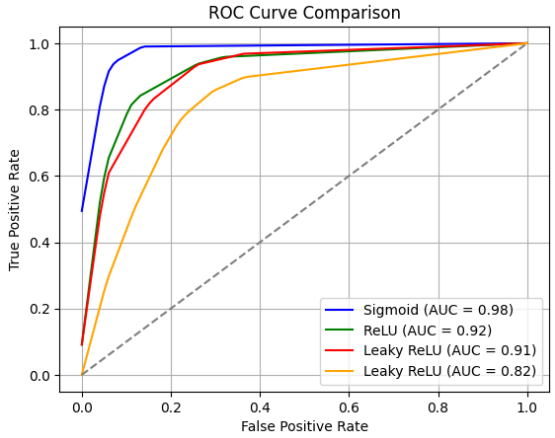

Figure 14: ROC Curve Comparison for different activation functions. Sigmoid (AUC = 0.98) outperforms ReLU (AUC = 0.92) and Leaky ReLU variants, showing superior classification performance.

Finally, as part of our evaluation, we later developed an optimized logistic regression model to improve upon initial results. This enhanced model was implemented using a customized Stochastic Gradient Descent (SGD) approach, designed to effectively balance computational efficiency and model performance. We selected a batch size of 32 for mini-batch gradient descent, as initial trials demonstrated that this batch size produced stable and accurate results with efficient computation. Similarly, the number of epochs was set to 1000, a choice guided by early experiments where convergence and accuracy stabilized quickly.

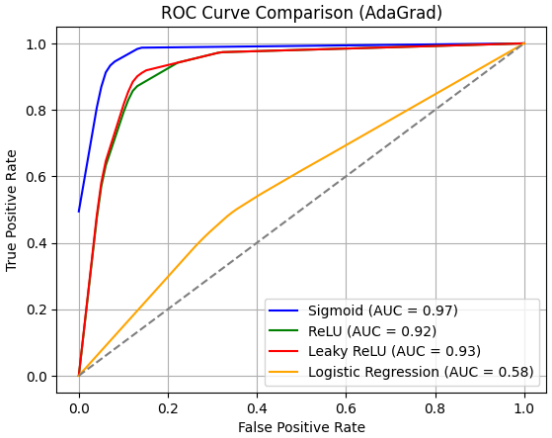

Figure 15: ROC Curve Comparison (AdaGrad). Sigmoid (AUC = 0.97) performs best, with ReLU (AUC = 0.92) and Leaky ReLU (AUC = 0.93) close behind. Logistic Regression performs poorly with an AUC of 0.58.

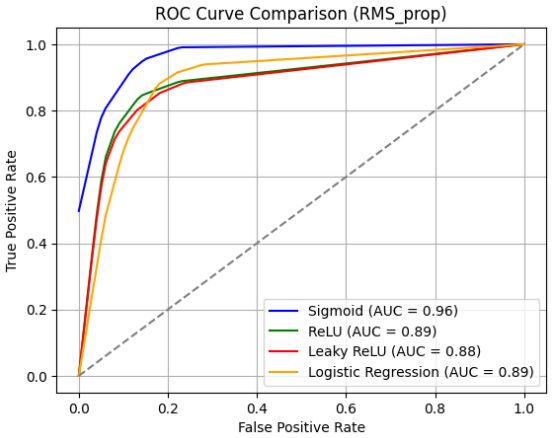

Figure 16: ROC Curve Comparison (RMSprop). Sigmoid (AUC = 0.96) again leads in performance, followed by ReLU (AUC = 0.89) and Leaky ReLU (AUC = 0.88). Logistic Regression and Leaky ReLU share an AUC of 0.89.

In the preprocessing phase, a small difference from the prior approach was introduced: we applied `StandardScaler` from the `sklearn.preprocessing` library [7], normalizing the input features to zero mean and unit variance. This approach, as opposed to `MinMaxScaler`, was chosen specifically to enhance gradient stability and speed up convergence within our SGD-based logistic regression, where normalized gradients tend to be more effective, especially for convex optimization problems.

The class for our logistic regression implementation allowed us to systematically fine-tune several key hyperparameters, including learning rate, L2 regularization strength, momentum, and early stopping criteria. Hyperparameter tuning was performed to achieve optimal model accuracy while minimizing overfitting. We evaluated the learning rate across a logarithmic scale from 10^{-5} to 10^{1}, as this range allowed us to capture potential performance peaks across a wide range of values. Early stopping was applied consistently, with a pa-

tience threshold to halt training when the loss function no longer showed improvement, ensuring efficient model convergence without overfitting.

Our exploration of learning rates consistently showed that small learning rates in the approximate range of 1 to 5 yielded the highest test accuracy. Momentum was also tested to see if it improved training stability by accelerating convergence. However, it did not provide meaningful gains in accuracy, so momentum was set to 0 by default. The results are summarized in Figure **17**, which shows how test accuracy varied with different learning rates.

For regularization, we explored L2 penalty values over the range 10^{-4} to 10^{1}, using the best learning rate previously found. The results, depicted in Figure **18**, revealed that the best-performing model did not require regularization, as the optimal L2 parameter was 0. This suggests that regularization was unnecessary, possibly due to the nature of the dataset and the model's architecture, which effectively captured the underlying patterns without overfitting.

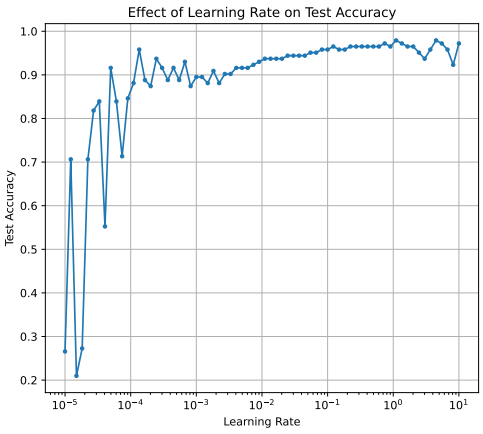

Figure 17: Effect of Learning Rate on Test Accuracy. The model achieved the highest accuracy with small learning rates, approximately in the range of 1 to 5, with diminishing returns at both lower and higher values. Test accuracies in this range were generally high, around 0.97 to 0.98.

Overall, this refined logistic regression implementation, with carefully chosen hyperparameters, `StandardScaler` preprocessing, and an optimal learning rate range, performed exceptionally well on the classification task, achieving test accuracies consistently around 0.97 to 0.98. This effectively closed the performance gap between logistic regression and neural networks for this dataset.

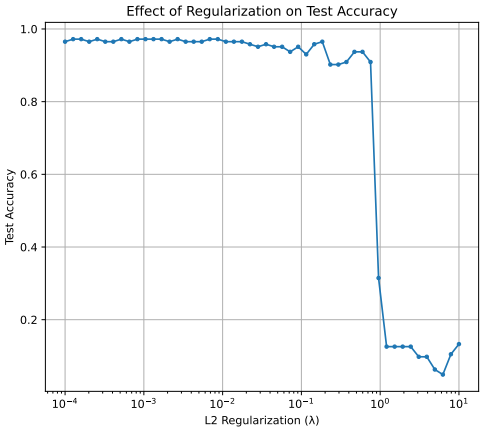

Figure 18: Effect of Regularization on Test Accuracy. The highest test accuracy was achieved with no regularization (L2 = 0), indicating that the model could generalize well without penalizing larger weights. Test accuracies with this configuration were generally around 0.97 to 0.98.

5. DISCUSSION

Regression

In the regression analysis of the Franke function, the FFNN's performance indicated a balance between training stability and model complexity. Implementing Ridge regression and OLS methods yielded MSEs notably lower than those produced by the FFNN. The linear regression approaches achieved MSEs of 0.0026 (OLS) and 0.0036 (Ridge), whereas the FFNN, with an MSE of 0.3397, demonstrated potential overfitting. This discrepancy suggests that simpler models may achieve comparable or superior performance in scenarios where the underlying function is less complex.

Furthermore, the learning rate, epoch count, and batch size directly influenced the FFNN's regression performance. A learning rate of $\eta = 10^{-3}$, coupled with 5000 epochs, provided sufficient time for the model to converge. Additionally, a mini-batch size of 25 to 50 achieved a balanced trade-off between update variability and learning stability, reducing abrupt shifts in parameter space.

We observed a notable bias-variance trade-off with the FFNN. Increased model complexity (e.g., more hidden layers or higher polynomial degrees) led to reduced bias but increased variance, particularly at higher learning rates. Consequently, moderate configurations (e.g., two hidden layers with fewer nodes) produced the most robust results, aligning with findings from classical regression models.

Classification

In analyzing the classification performance of the Feed Forward Neural Network (FFNN) on the Wisconsin Breast Cancer dataset, we observed several critical aspects that reflect the network's capacity for distinguishing between benign and malignant tumors. The choice of predictors was informed by examining the features with

significant correlations to the target label Y. However, considering the potential redundancy, we refrained from incorporating predictors with high mutual correlations, as these could reduce the model's informative capacity. Instead, we chose a condensed representation of features through dimensionality reduction techniques or weighted averages. Notably, this approach can be optimized further by tuning these aggregation methods as hyperparameters.

A notable adaptation involved normalizing the predictors, particularly continuous variables, to a standard scale. This preprocessing step enhanced the network's capacity to interpret variations in input data consistently. Certain predictors, such as gender, demonstrated limited impact on classification performance, suggesting that they could be omitted in streamlined analyses without significant loss of accuracy. Conversely, features directly linked to the tumor characteristics provided valuable predictive information.

The classification results, as visualized in ROC curves and performance metrics, demonstrated that ReLU and Sigmoid activations generally improved the network's classification capability, aligning with prior research. For example, the FFNN with Sigmoid activation outperformed logistic regression, achieving an accuracy of 97%, a significant improvement over the 85% accuracy of logistic regression. These findings underscore the FFNN's ability to capture complex non-linear relationships inherent in medical data.

However, it is important to specify that by building on the initial simple logistic regression model, we developed a more sophisticated version by incorporating additional parameters, such as learning rate, L2 regularization, batch size, epochs, and early stopping criteria, alongside a small but impactful preprocessing adjustment using `StandardScaler`. These enhancements allowed even a simpler classification method like logistic regression to achieve competitive performance with the neural network model, with test accuracies consistently reaching around 97-98%.

Given the sensitive nature of this classification task—identifying breast cancer—an interpretable model like logistic regression offers significant advantages. Its transparent, linear decision-making process makes it a valuable tool in medical contexts, where understanding how features contribute to predictions is often crucial for healthcare professionals. Thus, our refined logistic regression model not only demonstrates strong predictive accuracy but also underscores the importance of balancing interpretability and complexity in medical applications.

6. CONCLUSIONS

We evaluated the performance of Feed Forward Neural Networks (FFNNs) in both classification and regression tasks. In the classification case, the FFNN displayed superior performance in terms of true positive rates, suggesting its suitability for scenarios requiring high sensitivity, such as medical diagnostics. Specifically, FFNNs with ReLU and Leaky ReLU activations demonstrated superior classification accuracy compared to logistic regression. However, while the FFNN with Sigmoid and ReLU activations excelled, a Leaky ReLU variant displayed competitive accuracy, indicating that alternative activation functions could provide robustness across varying datasets. The exception was the Sigmoid activation function, where performance was comparable to logistic regression. However, examining the ROC curves revealed a limitation: FFNNs showed a higher rate of false negatives compared to true positives, indicating some misclassification in cases with positive outcomes. For specific thresholds, logistic regression outperformed FFNNs on negative data points, but with a well-tuned threshold, the FFNN achieved comparable or better performance.

The study demonstrates that while FFNNs excel in non-linear classification tasks, such as those in medical diagnostics, simpler models like OLS and Ridge regression may outperform FFNNs in well-defined regression tasks. For the regression case, we applied our FFNN to approximate the Franke function and performed a parameter search to determine optimal hyperparameters. By varying one parameter at a time, we reduced computational demands compared to a traditional grid search. Examining the bias-variance trade-off, we observed that adjusting hyperparameters resulted in a balance between high bias at lower complexity and high variance at higher complexity. This effect was most evident with smaller datasets and low noise levels, indicating that it may not be as pronounced in scenarios with abundant data. Using 400 data points, the optimized FFNN model significantly outperformed Ridge regression when noise levels were moderate.

Potential enhancements for the FFNN include implementing a variable learning rate and adding dropout nodes to enhance performance further. These modifications could potentially reduce overfitting, particularly in cases with high noise. Additionally, further exploration of hyperparameter optimization, particularly in hidden layer architecture and activation functions, could yield improved results across a broader range of datasets. The findings align with prior studies, confirming the FFNN's efficacy in non-linear classification tasks. However, the regression performance suggests a need for caution when applying FFNNs to simpler datasets where linear models might provide a more interpretable and computationally efficient solution.

Additionally, through a more sophisticated implementation of logistic regression with fine-tuned hyperparameters and a preprocessing adjustment using `StandardScaler`, we achieved performance levels that approached those of the FFNN in classification accuracy. This refinement allowed logistic regression to provide competitive results, particularly for high-sensitivity tasks like medical diagnostics. The advantage of this optimized logistic regression model lies in its interpretability, making it a valuable alternative to FFNNs in clinical settings where transparent decision-making is essential.

Overall, our study suggests that FFNNs provide a flexible and effective approach to both classification and regression tasks, with the potential for enhanced performance through further tuning and architectural adjustments. Future work should explore advanced architectures (e.g., convolutional layers or recurrent networks for sequential data) and alternative cost functions for regression tasks, potentially enhancing FFNN efficacy in scenarios with complex functional dependencies.

Additional results can be found in the GitHub repository

References

[1] P. A. Lopez Torres G. Durante E. Ottoboni. *Project1*. https://github.com/Elisaottoboni/Machine-Learning-University-of-Oslo/tree/main/Project1. 2024.

[2] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.

[3] S. Raschka et al. *Machine Learning with PyTorch and Scikit-Learn: Develop Machine Learning and Deep Learning Models with Python*. Expert insight. Packt Publishing, 2022. ISBN: 9781801819312. URL: https://books.google.no/books?id=UHbNzgEACAAJ.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. Chapters 2-14 are highly recommended; lectures follow this text to a large extent. MIT Press, 2016. URL: https://www.deeplearningbook.org/.

[5] P. A. Lopez Torres G. Durante E. Ottoboni. *Project2*. https://github.com/Elisaottoboni/Machine-Learning-University-of-Oslo/tree/main/Project2. 2024.

[6] M. Hjorth-Jensen. *MachineLearning*. https://github.com/CompPhysics/MachineLearning. 2024.

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

[8] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Main textbook covering chapters 1-7, 11, and 12. Available for free PDF download at https://www.microsoft.com/en-us/research/uploads/prod/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf. Springer, 2006. URL: https://www.springer.com/gp/book/9780387310732.

[9] Aurélien Geron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. Contains many code examples and hands-on applications of algorithms discussed in the course. O'Reilly Media, 2019. URL: https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/.