# Less is More: Comparing Neural Networks with Regression for Classification and Approximation

Oscar Atle Brovold, Eskil Grinaker Hansen, Håkon Ganes Kornstad

November 4, 2024

`Project GitHub Repository`

**Abstract**

Artificial neural networks offer flexibility for solving a variety of prediction and decision problems. However, they can also be computationally costly and energy-intensive. While advances in computational power have brought Machine Learning forward, challenges remain on the energy consumption side. In this paper, we aim to investigate how a feed-forward neural network (FFNN) compares to traditional regression methods, which are more lightweight in terms of computation and easier to implement. We start by coding a FFNN from scratch and train it using custom optimizers, including Stochastic Gradient Descent (SGD), RMSProp, and other variants. We approximate synthetic data generated from Franke's Function, and then apply the model to classify real health conditions using the well-documented Breast Cancer Wisconsin (Diagnostic) dataset. We then implement a logistic regression model for comparison. Our findings show that a well-tuned FFNN can effectively model both problems. For Franke's Function, our optimized Neural Network is able to achieve an Mean Squared Error (MSE) of approximately 0.01195. This is comparable to results from previous testing with an analytic linear regression model. For classification with the breast cancer dataset, we achieve an accuracy of around 0.982. The logistic regression model provides slightly better accuracy in comparison. We conclude that traditional methods are preferable for some classification or approximation tasks. They can produce comparable results to neural networks, while possibly offering lower computational costs, reduced energy consumption, and simpler implementation.

## INTRODUCTION

The human brain is a remarkable organ, with billions of neurons and trillions of synapses intricately connected in a vast network. Humans can recall previously stored information and use it to reason forward in time about unseen challenges in a matter of milliseconds. Perhaps even more impressive, the brain operates on a mere 15–20 watts. [1] It's no surprise, then, that the concept of artificial neural networks is intertwined with such magic and mystery, as it serves as the gateway into artificial intelligence and the modeling of the brain.

In our first paper *Regression Analysis and Resampling Techniques in Applied Machine Learning* [2], we explored three regression methods with their cost functions, testing their approximation capabilities on synthetic data from Franke's Function [3] and then on real terrain data. Using the best result from these findings as our benchmark, we now turn to evaluating the performance of an Artificial Neural Network. So called feed-forward neural networks (FFNN) are flexible enough to serve a vast array of different tasks, from function approximation to cancer data classification. This flexibility, however, comes with a cost: With a more advanced machine comes a more complex control panel. Artificial Neural Networks introduce more hyperparameters that need to be fine-tuned. With the time constraints for this report however, we will only be tapping into the wisdom of parameter tweaking. To save time, and at the same time ensuring the best possible results, we will try to alter the parameters as systematically as possible, creating plots along the way to support our

reasoning.

In addition to accuracy, we remain aware that runtime and model simplicity can contribute to overall computational efficiency. In larger-scale operations, these factors can influence energy consumption, making simpler models a green alternative, if they can achieve a comparable performance.

We will begin by setting up an FFNN for modelling Franke's Function, investigating ideal network sizes, use of optimizers, activation functions and hyperparameters for this approximation task. After a brief comparison with our previous findings, we will show the flexibility of an FFNN, by reconfiguring it as a classifier for diagnosing breast cancer, based on 30 feature variables. Again, fine-tuning of the setup is essential, requiring both domain knowledge and mathematical and statistical insight.

Finally, we will test a logistic regression model on the same data to evaluate its effectiveness in diagnosis. Through this study, we aim to show that while an FFNN can yield satisfactory results for diverse tasks, more traditional regression models may offer advantages for the datasets in this paper, particularly in computational efficiency, just like our brain.

We will benchmark our FFNN against an off-the-shelf neural network implemented in PyTorch and compare our logistic regression model with scikit-learn's implementation. Throughout testing, we also monitor energy consumption using the `CodeCarbon` [4] package. While we won't incorporate these measurements directly in this report, we aim to raise awareness about energy use in machine learning. This may serve as a foundation for further integration of energy considerations in our future reports.

## THEORETICAL BACKGROUND

### OPTIMIZATION ALGORITHMS

**Plain gradient descent**

Given a cost function, the goal of the gradient descent is to minimize this function by iteratively adjusting the coefficients. The gradient descent algorithm computes the gradient of the cost function with respect to the coefficients, which points in the direction of the steepest ascent. The algorithm then updates the coefficients in the opposite direction, progressively minimizing the cost function.

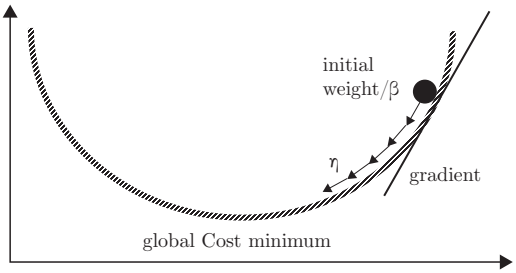



**Figure 1:** In machine learning, finding the optimal solution for a given objective function can be seen as moving downhill in search of the global cost minimum wrt. the model parameters. [5]

For each iteration, the update for the coefficients is:

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \eta \frac{\partial}{\partial \boldsymbol{\beta}} \mathcal{C}(\boldsymbol{\beta})$$

Here, $\eta$ is the step size (learning rate), while $\frac{\partial}{\partial \boldsymbol{\beta}} \mathcal{C}(\boldsymbol{\beta})$ is the derivative of the cost function, with respect to the coefficients. Choosing the right step size is crucial; if $\eta$ is too small, the gradient descent can be slow and computationally expensive. If $\eta$ is too large, the algorithm may overshoot and diverge. [5].

**Gradient descent with momentum**

To expand on plain gradient descent, we can add a **momentum** term that keeps track of the change in direction at each iteration. This allows the algorithm to build velocity in directions of consistent descent, leading to faster convergence. The update rule for gradient descent with momentum is:

$$\boldsymbol{v} \leftarrow \gamma \boldsymbol{v} + \eta \boldsymbol{g}$$
$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \boldsymbol{v}$$

where $\boldsymbol{v}$ is the velocity, $\gamma$ is the momentum parameter controlling the contribution of the velocity, and $\eta$ is the learning rate. $\boldsymbol{g}$ is the gradient of the cost function at the current iteration. [5]

**Stochastic gradient descent, with and without momentum**

Calculating the plain gradient descent may become very computationally expensive for a big set of input data. One common work-around is to use Stochastic Gradient Descent (SGD).

SGD updates the weights by using one random sample from the input at a time.

This process repeats for a set number of epochs or until a treshold value between old and new weights is achieved [6]. Although each step is less precise, using one sample enables faster progress, ultimately guiding us to the minimum more efficiently.

The method is often used in training neural networks, however it is common to use a **mini batch** of the entire input instead of only one sample. This allows the training to remain less computationally expensive, along with each step being more accurate. The learning rate for each step should decrease over time, to enusre convergence [7]. Just as with plain gradient descent, it is possible to include a momentum term to the updated weights.

**AdaGrad**

AdaGrad is a method to tune the learning rate as a function of time. To achieve this, we introduce the accumulated squared gradient $\boldsymbol{r}$:

$$\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$$

The general expression for AdaGrad is now:

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \eta \frac{1}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g} \tag{1}$$

To ensure numerical stability, a small constant $\delta$ is introduced, normally set to $10^{-7}$. [7]

Equation 1 shows that step-size for each weight is inverse proportional to the previous gradients. The method's strengths lies in its ability to adaptively change the learning rate, as a function of time.

**RMSProp**

As AdaGrad performs best in convex optimization, RMSProp was introduced to be applied in non-convex settings. Specifically, RMSProp expands on AdaGrad in introducing a decay rate $\rho$. With $\rho$ we modify the accumulation variable $\boldsymbol{r}$:

$$\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g} \tag{2}$$

Now, the history of gradients is regulated by $\rho$, where larger values of $\rho$ cause the optimizer to retain more of the past squared gradients, while smaller values place more emphasis on the recent gradients. The current squared gradient is scaled by $1 - \rho$, meaning that RMSProp "forgets" older gradients, and that recent gradients have a greater influence. [8]

**Adam (Adaptive Moment Estimation)**

Combining techniques from RMSProp with an added momentum, Adam defines the accumulated moment estimates as:

$$\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1) \boldsymbol{g}$$
$$\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2) \boldsymbol{g} \odot \boldsymbol{g}$$

Since the moment estimates are initialized to zero, they can be significantly biased in the early iterations. We therefore apply bias correction to $\boldsymbol{s}$ and $\boldsymbol{r}$:

$$\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$$
$$\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$$

Here $t$ is a time step, initialized to zero, updated with $t \leftarrow t + 1$ for each iteration. Finally we update the coefficients:

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \eta \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$$

Empirical evidence suggest that Adam is robust and effective across various tasks. However, it lacks a comprehensive theoretical framework that fully explains its success. [7]

## FEED-FORWARD NEURAL NETWORKS

In a neural network, input data is propagated through multiple layers and neurons to produce a final prediction, known as the **forward pass**. This structure is depicted in Figure 3. To fully understand how this process works, it is essential to examine some fundamental components first.

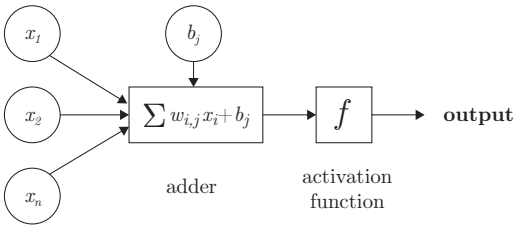



**Figure 2:** An artificial neuron. The neuron receives multiple inputs $x_1, x_2, \ldots, x_n$, each multiplied by a weight $w_{i,j}$. These weighted inputs are then summed together along with a bias term $b_j$, and then passed through an activation function $f$, which introduces non-linearity and determines the output of the neuron.

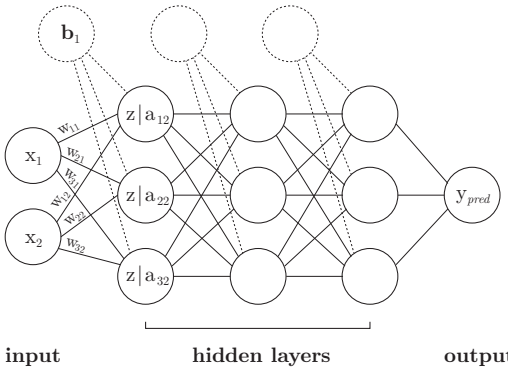



**Figure 3:** A Neural Network is a graph of neurons, represented as nodes in this diagram. Each neuron in one layer is fully connected to the neurons in the subsequent layer, with the connections represented by weighted edges. The $n$ internal layers of nodes are called **hidden layers**, making it a **deep neural network** for $n > 1$. During a forward pass, the network calculates the linear combinations of all input values with their corresponding weights and biases (shown as dashed lines), followed by applying an activation function at each neuron. [9]

**Activation functions**

To activate and thereby weigh each hidden layer in a deep network, we use activation functions, also known as **units**. The **Rectified Linear Unit (ReLU)** is commonly used because it acts as a switch, turning the neuron on or off based on the input. Its linear form,

$$g(z) = \max(0, z),$$

makes it simple to differentiate with a derivative of 1 whenever $z > 0$. The properties of ReLU renders neurons effectively dead for negative input values. To address this, several alternatives exist, including **Leaky ReLU**, where non-zero function values exist even for negative inputs.

Another widely used unit, is the **Sigmoid Function**, given by:

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

This function smoothly maps the input $z$ to a value between 0 and 1, also creating an effective switch which is differentiable. [7]

Yet another activation function, the **Softmax function**, is commonly applied to the output layer of a neural network. It transforms its input $\boldsymbol{z}$ into a probability distribution across different classes. It is defined as:

$$\sigma(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Here, each output value is normalized, ensuring that the sum of all outputs equals 1. Softmax effectively "activates" the most likely class, making it particularly useful in multi-class classification. [10]

**Cost Function: Cross Entropy**

In our first paper [2], we discussed the cost functions used for linear regression problems. However, for classification tasks we aim to find the model that maximize the probability of correct classification. From a statistical perspective, this involves minimizing a cost function that represents the negative **log-likelihood**. This cost function is known as the cross entropy loss:

$$\mathcal{C}(\boldsymbol{\theta}; x) = -\mathbb{E}_{(x,y)\sim\hat{p}_{\text{data}}} \log(p_{\text{model}}(y|x)),$$

where $\boldsymbol{\theta}$ are the model parameters, $p_{\text{model}}(y|x)$ is the model's predicted probability for the output $y$ given $x$ as input, and $\hat{p}_{\text{data}}$ is the the empirical distribution defined by the data set. [7]

In the case of binary classification, we can simplify the expression to:

$$\mathcal{C}(\boldsymbol{\theta}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}),$$

where $y$ is the target value, true or false, and $\hat{y}$ is the model's predicted probability of a true output. [10]

**Forward propagation**

In the **forward pass** of a neural network, the output of each neuron is computed as a linear combination of the input values, weights, and biases. For a specific neuron in the $\text{L}^{\text{th}}$ layer, this can be expressed as:

$$z_i^L = w_{i1} a_1^{L-1} + w_{i2} a_2^{L-1} + \ldots + w_{ij} a_j^{L-1} + b_i^L,$$

where $z_i^L$ is the linear combination for neuron $i$ in layer $L$, $w_{ik}$ are the weights, $\boldsymbol{a}^{L-1}$ are the activations from the previous layer $L-1$ and $b_i^L$ is the bias term.

Since neural networks often need to model non-linear behaviors, an **activation function** $f$ can be applied to the linear combination $z_i^L$. The output of an individual neuron is then given by:

$$a_i^L = f(z_i^L),$$

where $a_i^L$ is the activated output of the neuron in layer $L$. [11]

The final hidden layer is then connected to the output layer, where the model produces its prediction, $y_{\text{pred}}$, based on the input data. It is common practice to use one type of activation function in the hidden

layers and a different one in the output layer, depending on the problem. The model's prediction is then evaluated using an appropriate **cost function**.

To improve the network's performance, the weights and biases need to be adjusted. This is done through a backward pass, commonly called **backpropagation**, where the gradients of the cost function with respect to the network parameters are computed and used to update the parameters, thereby minimizing the cost function. [9]

### Backpropagation

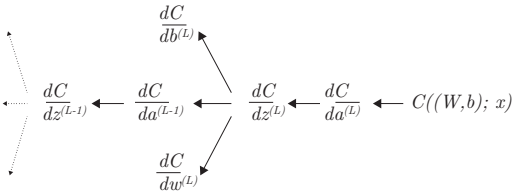



**Figure 4:** During backpropagation, the gradients of the loss with respect to the bias $\boldsymbol{b}$ and weights $\mathbf{W}$ are computed through the chain rule. This entails propagating the gradient $\frac{\partial \mathcal{C}}{\partial z}$ backwards, calculating the gradients with respect to $a$ and $z$, which are then used to update weights and biases recursively. [12]

The purpose of backpropagation is to iteratively adjust the network's weights and biases to minimize the loss function, enabling the model to learn from training data and improve its predictions over time. The backpropagation algorithm begins at the output layer, calculating the gradient of the loss with respect to each of the network's variable parameters. Once all gradients are computed, we use gradient descent to update these parameters, gradually refining the model's accuracy through successive iterations. The gradients we set to find are:

$$\frac{\partial \mathcal{C}}{\partial \mathbf{W}}, \frac{\partial \mathcal{C}}{\partial \boldsymbol{b}},$$

for all layers in the network. The derivatives for the output layer can be expressed, using the chain rule, as:

$$\frac{\partial \mathcal{C}}{\partial \mathbf{W}^L} = \frac{\partial \boldsymbol{z}^L}{\partial \mathbf{W}^L} \frac{\partial \boldsymbol{a}^L}{\partial \boldsymbol{z}^L} \frac{\partial \mathcal{C}}{\partial \boldsymbol{a}^L} \tag{3}$$

$$\frac{\partial \mathcal{C}}{\partial \mathbf{b}^L} = \frac{\partial \boldsymbol{z}^L}{\partial \mathbf{W}^L} \frac{\partial \boldsymbol{a}^L}{\partial \boldsymbol{z}^L} \frac{\partial \mathcal{C}}{\partial \boldsymbol{a}^L} \tag{4}$$

As we see in figure 4, we move recursively through the layers to find the derivatives, with:

$$\frac{\partial \mathcal{C}}{\partial \boldsymbol{a}^{L-1}} = \frac{\partial \boldsymbol{z}^L}{\partial \boldsymbol{a}^{L-1}} \frac{\partial \boldsymbol{a}^L}{\partial \boldsymbol{z}^L} \frac{\partial \mathcal{C}}{\partial \boldsymbol{a}^L} \tag{5}$$

We then replace $\frac{\partial \mathcal{C}}{\partial \boldsymbol{a}^L}$ from Equation 3 and 4 with the gradient calculated from Equation 5 and subtracting 1 from every L. This process repeats for every hidden layer in the network.

### One-Hot Encoding

[class 1, class 2, class 3] ⟶

| class 1 | class 2 | class 3 |
|---|---|---|
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

**Figure 5:** In classification problems, arranging classes in a sequential array can introduce unintended bias, as the algorithm might interpret a higher index as a higher *value* or *rank*. To circumvent this issue, we use **one-hot encoding**, where each class is represented by a unique binary vector. This eliminates any **positional bias** by treating each class as an independent binary feature.

### PyTorch

PyTorch is an open-source library developed by Meta AI. It includes a range of internal functions, including an autograd feature for automatic differentiation. PyTorch can run on both CPUs and GPUs, with GPUs often providing better performance due to their optimized handling of tensor operations. In PyTorch, neural networks can be built with `torch.nn`, optimizers are available through `torch.optim`, and batch processing is handled by the `DataLoader` class. [13]

### Logistic regression

Logistic regression is commonly used in classification tasks. When provided with a sufficient amount of independent variables, a logistic regression model can effectively classify what category a sample belongs to. The binary logistic regression model takes a set amount of input variables and returns a number between 0 and 1, where target label 0 corresponds to `false`, and label 1 to `true`. Usually, a number closer to 0 than 1 indicates the sample belonging to class 0 and vice versa. [11]

Logistic regression has a few properties in common with linear regression, but differs on its cost function, weight optimalization and output. As cost function for a binary regression model, the *binary cross entropy* can be utilized. In order to optimize the model we can use gradient descent on the cost function with respect to the weights. The initial output of the function is given by:

$$\boldsymbol{z} = \mathbf{X}\boldsymbol{\beta} + b \tag{6}$$

However, we stated that a logistic regression

model should produce a number between 0 and 1, that is why $z$ is run through the previously discussed *sigmoid* function. [14]

**Wisconsin Breast Cancer Database.**

Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image. [15] The data set contains 569 instances with 30 numerical features for each sample. The ground truth label is binomial, indicating malignant or benign (`true/false`).

## INITIAL TESTING WITH OPTIMIZERS

To gain a deeper understanding of the different optimization algorithms discussed in the theory, we implement them and use the following polynomial function for prediction:

$$f(x) = 1 + 3x + 2x^2$$

The goal is to explore the differences between various optimizers and gain insights into parameter-tuning. It is important to highlight that we test these optimizers on a simple second-order function, with only three weights to optimize. The specific optimizer that performs best in this case, along with the parameter tuning strategies we use, may not directly transfer to more complex models like neural networks. However, this process helps develop essential intuition.

As neural networks become more complex, reducing runtime becomes increasingly important. We aim to use an optimizer that can reach a given threshold with fewer iterations. Our experiments with the second-degree polynomial suggest that more sophisticated optimizers, such as Adam, Adagrad, and RMSProp, perform the best. These optimizers achieve a very low MSE within the first few iterations. However, it is crucial to note that their performance varies significantly depending on the choice of hyperparameters. We note that poorly chosen parameters can lead to suboptimal results, regardless of the number of iterations, and may even cause the algorithms to diverge. Please refer to the `repository` for plots.

## APPROXIMATING FRANKE'S FUNCTION WITH A NEURAL NETWORK

Fine-tuning a neural network involves changing parameters that are highly interconnected: tweaking one parameter often requires adjustments to others. To make this process manageable within time constraints, we need to establish a general framework. For Franke's Function, we will focus on using **stochastic gradient descent** without momentum, which allows us to concentrate on specific hyper parameters such as batch size, epochs, initialization strategies, regularization, and the structure of hidden layers. We begin with a train/val test (see Figure **F1**), gradually moving on to finer details to optimize the network's performance.

**Cross Validation**

To ensure consistency throughout our analysis, we employ the cross validation resampling technique [2]. We use scikit-learn's [16] built in cross validation functions, where we set `folds = 5` as a standard.

**Bias initialization**

Parameter initialization can directly affect the behaviour of our neural network, where well-suited initialization methods can reduce the time for the network to converge during training. We employ different methods of initialization on the network's biases to gain an understanding of what may or may not work. Figure **F2** shows how a network behaves as each model is using its own method. From the figure, we propose that bias initialization is negligible, as long as we use one of the recommended methods [7]. We choose to initialize our biases within a standard normal distribution.

**Epochs and batch size relation**

To achieve both efficient training and accurate models, we find it useful to evaluate our model's predictive performance while maintaining a low runtime. We develop a method that examines both the final loss and the time taken to obtain the optimal model for different epochs and batch sizes. Specifically, we set a target number of iterations, vary the batch size, and then use the following formula to find the corresponding number of epochs:

$$\text{epochs} = \frac{\text{iterations} \times \text{batch size}}{\text{samples}} \tag{7}$$

This approach ensures that the weights and biases in the network are updated an equal number of times. It is now meaningful to choose the batch size and epochs in a way that preserves accuracy while minimizing runtime.

From figure **F4** we observe that lower batch sizes significantly improve runtime. As shown in figure **F3**, the MSE does not vary notably as batch size decreases. Based on this we propose using a batchsize $\leq 2^3$ together with an appropriate number of epochs, as given by Equation 7. This will now serve as a starting point for further hyperparameter tuning.

**Setting the hidden layer structure**

An important consideration when implementing a neural network is the structure of the layers. We use a linear activation function for the output layer, as it is well-suited for approximating continuous values like heights. Furthermore, we focus on the number of hidden layers, the number of neurons within each layer, and the choice of activation functions. The previously discussed interdependence makes it challenging to find a single configuration that generalizes well. However, through experimentation with various parameter combinations, we can develop a general intuition for what works effectively and what does not. As a framework for this analysis, we use Stochastic Gradient Descent, learning rate = 0.01, epochs = 30, batch size = 4 and weights and biases standard normally distributed.

When examining Figure **F5**, we find that using up to two hidden layers seems to hit an optimal balance. When more than two hidden layers are used with the sigmoid activation function, the MSE begins to increase, suggesting that the network may become too complex and struggle with generalization. For ReLU and LeakyReLU, we find that the network tends to diverge as the number of layers increases. We propose that this is due to the linear relationship of ReLU and LeakyReLU for values where $x > 0$, which may lead to instability in deeper networks.

We now move on to examine the number of parameters for two hidden layers, i.e increasing number of neurons per layer, as shown in Figure **F6**. For the sigmoid activation, we can increase the network complexity significantly without the MSE beginning to increase. However, it is important to note that it reaches a plateau around 500 parameters, which corresponds to around 20 neurons per layer. If we can choose a network which performs equally well, but with less complexity, this would be an obvious choice. On the other side, ReLU and LeakyReLU achieves a lower MSE for less complex networks than sigmoid, but when the number of parameters reaches around 300, corresponding to around 15 neurons per layer, it struggles with convergence.

On a final note, we once again recognize that all the information received from these plots are highly dependent of the choice of learning rate, optimizer, initialization strategies, and other factors. However, for this specific network, we propose that ReLU and LeakyReLU in combination with a less complex network, could prove beneficial. Nonetheless, we will use the sigmoid activation function, as it does not struggle with divergence, and produces generally more stable results.

**Effect of regularization**

With a general intuition in place, we now explore the effects of adding an L2-regularization term to the backpropagation process. Figure **F7** shows that a regularization parameter in the range $10^{-7} \leq \lambda \leq 10^{-4}$ can sometimes reduce the MSE, although selecting an inappropriate value may significantly worsen model performance. We also find that the effectiveness of regularization depends on the learning rate. For instance, a learning rate of 0.1 with $\lambda = 10^{-7}$ yields the lowest MSE, while the same regularization value with other learning rates leads to an increase in MSE. Therefore, we suggest that adding regularization should be considered when designing a neural network, as it may enhance performance under the right conditions.

**Concluding the testing on Franke's function**

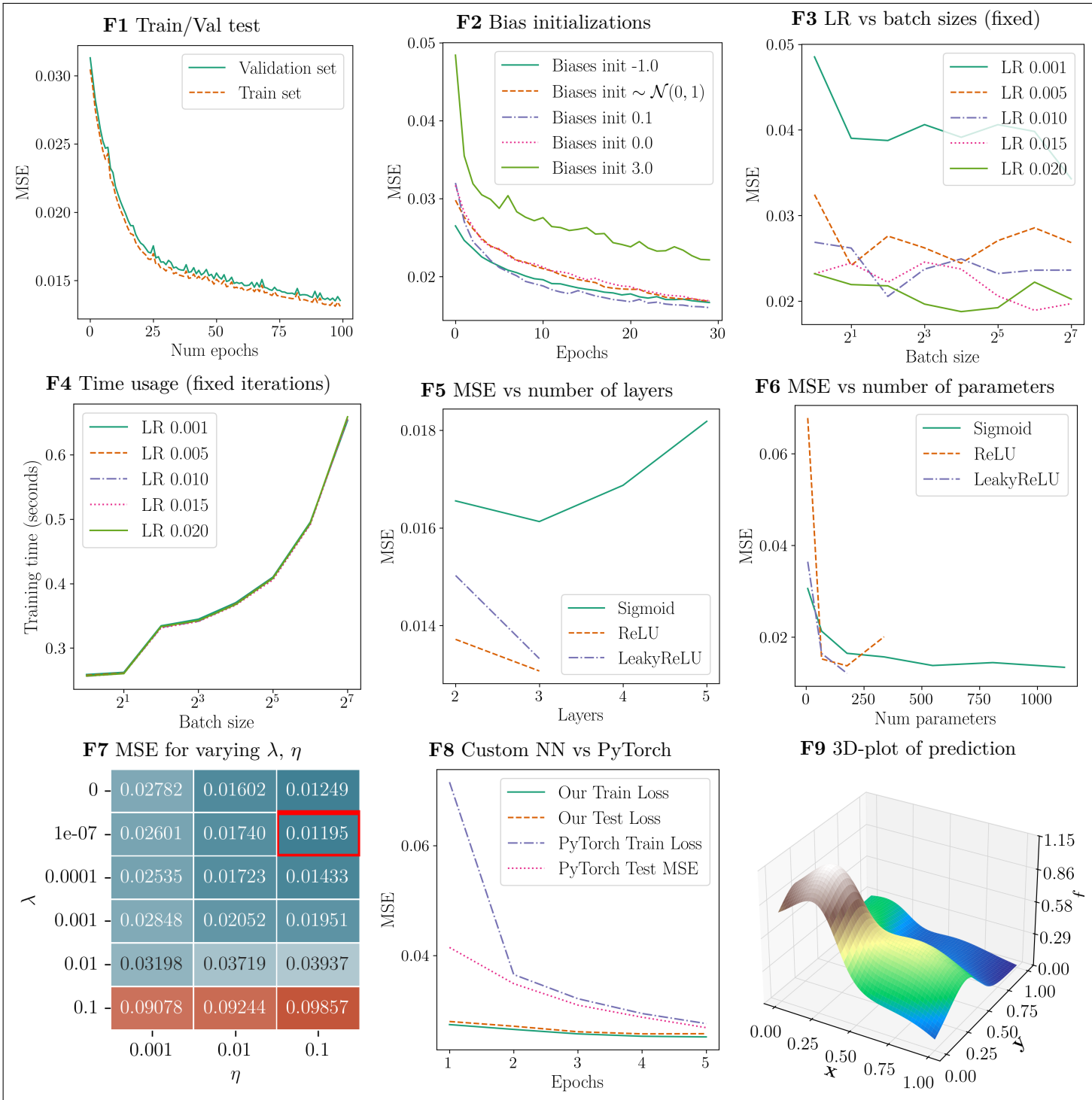
Finally, we suggest that Stochastic Gradient Descent with learning rate = 0.1, regularization parameter $\lambda = 10^{-7}$, two hidden layers with 10 neurons each using sigmoid as activation function, a batchsize of 4, and 30 epochs provides a good setup for training the neural network for Franke's function. Training the network with these parameters and then making predictions produces z-values within the scope of the terrain displayed in **F9**.

We are now ready to compare our linear regression analysis from our first paper [2] with this neural network implementation.

| Dataset | FFNN | Regression |
|---|---|---|
| Franke (MSE) | 0.01195 [F7] | 0.01034 [2] |

Table 1: Results from FFNN and linear regression on modeling Franke's function.

From the results shown in Table 1, we see that the FFNN does well, but the simpler analytical model we set up in our first project [2] did even better. With that, we propose that linear regression with regularization offers a simpler, computationally efficient solution, yielding reasonable accuracy when using moderate polynomial complexity. While neural networks may be somewhat excessive for modeling Franke's function, their adaptability allows them to handle it effectively.




**Testing FFNN with Franke's function:** **F1** MSE over epochs. 2 hidden layers with 10 neurons. Sigmoid activations. Epochs: 100, lr: 0.01, batch size: 4, optimizer: SGD, loss: MSE. **F2** Different bias initializations. 2 hidden layers with 10 neurons, and Sigmoid activations. $k$ folds with $k = 5$. Epochs: 30, lr: 0.01, batch size: 4, optimizer: SGD, loss: MSE. **F3** $\log_2$-plot of learning rates vs. batch sizes for fixed iterations ($i = 10000$). 2 hidden layers with 10 neurons, and sigmoid activations. $k$ folds with $k = 5$. Learning rate: 0.01, optimizer: SGD, loss: MSE. **F4** $\log_2$-plot of time usage vs. batch size for fixed iterations ($i = 10000$). 2 hidden layers with 10 neurons, and sigmoid activations. $k$ folds with $k = 5$. Loss function: MSE. **F5** MSE vs. number of layers. 10 neurons in each layer. $k$ folds with $k = 5$. Epochs: 30, lr: 0.01, batch size: 4, optimizer: SGD, loss: MSE. The plot clearly shows how the ReLU and LeakyReLU activation functions collapse for a number of layers larger than 3. **F6** MSE vs. number of parameters. 2 hidden layers with 10 neurons. $k$ folds with $k = 5$. Epochs: 30, lr: 0.01, batch size: 4, optimizer: SGD, loss: MSE. Similarly, the ReLU and LeakyReLU activation functions fail when the total network size (number of parameters) grows larger, while Sigmoid holds. **F7** Heat plot of MSE given varying regularization terms $\lambda$ and varying learning rate $\eta$. 2 hidden layers with 10 neurons. $k$ folds with $k = 5$. Epochs: 30, batch size: 4, optimizer: SGD, loss: MSE. **F8** Plot showing how our custom-built FFNN benchmarks against PyTorch. Custom NN: 2 hidden layers with 10 neurons. $k$ folds with $k = 5$. Epochs: 5, lr: 0.01, batch size: 4, optimizer: SGD, loss: MSE. PyTorch trained similarly. Note that the weights have been initialized manually using `torch.nn.init.normal` with `mean=0.0, std=1.0`. **F9** Final prediction by our custom NN. 2 hidden layers with 10 neurons and sigmoids. Epochs: 30, lr: 0.01, batch size: 4, optimizer: SGD, loss: MSE, $\lambda$ = 1e-7

## TESTING THE BREAST CANCER WISCONSIN (DIAGNOSTIC) DATASET

With our neural network now producing satisfactory results for Franke's Function data, we will switch to a classification problem. Specifically, we will fine-tune our network for the **Breast Cancer Wisconsin (Diagnostic) dataset**. We start off by exploring which insights transfer well from our findings with Franke's Function, and gradually move over to more specific tuning of the Breast Cancer dataset. As with Franke's Function, we will resample with cross validation.

### Framework for Breast Cancer Wisconsin (Diagnostic)

Initially, we reproduce the bias initialization plot for this dataset. (please refer to the `repository` for plots). We generally observe the same results as for Franke's Function, namely that if we initialize around 0, the effect on the performance is negligible. Therefore, we will continue to initialize biases using a standard normal distibution.

In our analysis, we also run preliminary testing with various optimization algorithms. As noted by Goodfellow, often the most effective optimizer is the one you are most familiar with [7]. Thus, partly arbitrary, we choose to concentrate on the RMSProp algorithm for this breast cancer classification problem. RMSProp involves two hyperparameters, which is less than Adam and more than SGD. We suggest that this strikes a good balance for our further analysis.

For the inital testing, we look at number of parameters for different activation functions (SGD in figure **BC1** and RMSProp in figure **BC2**). Without any fine-tuning, RMSProp demonstrates promising performance. Based on the insights from figure **BC2**, we propose using the sigmoid activation function, as it appears to work well in combination with RMSProp for this classification task. For the output layer, we use the softmax activation function. We also observe that two hidden layers with 32 neurons each, seem to produce robust results, this is also demonstrated in figure **BC3**.

When choosing the batch size, we generally observe similar patterns to those found with Franke's Function. (again, please refer to the `repository` for plots) Specifically, a batch size $\leq 2^3$ proves beneficial, as it significantly reduces runtime while still yielding high-accuracy results. This assumes that we adjust the number of epochs to achieve a target number of iterations (Equation 7).

### Tuning RMSProp

With our general framework now in place for this dataset, we are prepared to delve into fine-tuning specific parameters within the RMSProp algorithm.

As a starting point, we create a heat map (Figure **BC4**) with learning rates along the x-axis, decay rates along the y-axis, and the accuracy of the validation set for each combination. We find that a learning rate of 0.01 combined with a decay rate of 0.9 maximizes accuracy at 0.9802.

To further strengthen our analysis, we now explore the effects of an added L2-regularization term $\lambda$. The impact of this, in combination with a varying learning rates, can be studied in figure **BC5**. It is difficult to provide one general take-away, as regularization sometimes proves beneficial, and sometimes not. For the combination that maximizes the accuracy, we have a regularization of $\lambda = 10^{-7}$. Therefore, as with Franke's Function, we suggest exploring the effect of a regularization term, as it can sometimes increase accuracy.

Once again, we note that these findings are not absolute, and more thourough analysis of these parameters in combination with the previously discussed parameters could provide an even better accuracy.

Based on our finds, we recommend using RMSProp with a learning rate of 0.05, a regularization term of $10^{-7}$, and a decay rate of 0.9. The network setup includes two hidden layers with 32 neurons each, using sigmoid activation, a batch size of 4, and 30 epochs. This configuration tends to yield a well-trained neural network for the Breast Cancer Wisconsin (Diagnostic) dataset. The resulting predictions produce the confusion matrix shown in Figure **BC6**.

Our intuition is that further analysis with Bayes' formula on this result could provide some additional insights, however time constraints limit us from doing so.

### Testing with Logistic Regression

When applying logisitic regression to predict cancer diagnoses, our goal is to identify a set of parameters that maximize accuracy. We implement a logistic regression model using Stochastic Gradient Descent,

with a scheduled learning optimizer. [17] Figure **BC7** illustrates that multiple training configurations can yield high validation accuracy. The heatmap displays validation accuracy across various batch sizes and regularization parameters. We find that the results vary across runs, likely due to random parameter initialization, and the tendency of gradient descent to get stuck in local minima. Nevertheless, our primary aim is to acheive a highly accurate model.

We store the five top-performing models and proceed to testing. As shown in Figure **BC8**, all models achieve high accuracy, with one model standing out as the best. This model will be our final choice.

Finally, we cross-validate our results with Scikit-Learn's own `SGDClassifier`, which is similar to our logistic regression implementation. From this we obtain similar outcomes.

**Comparison of FFNN with Logistic regression**

We are now prepared to compare our logistic regression analysis with the neural network implementation.

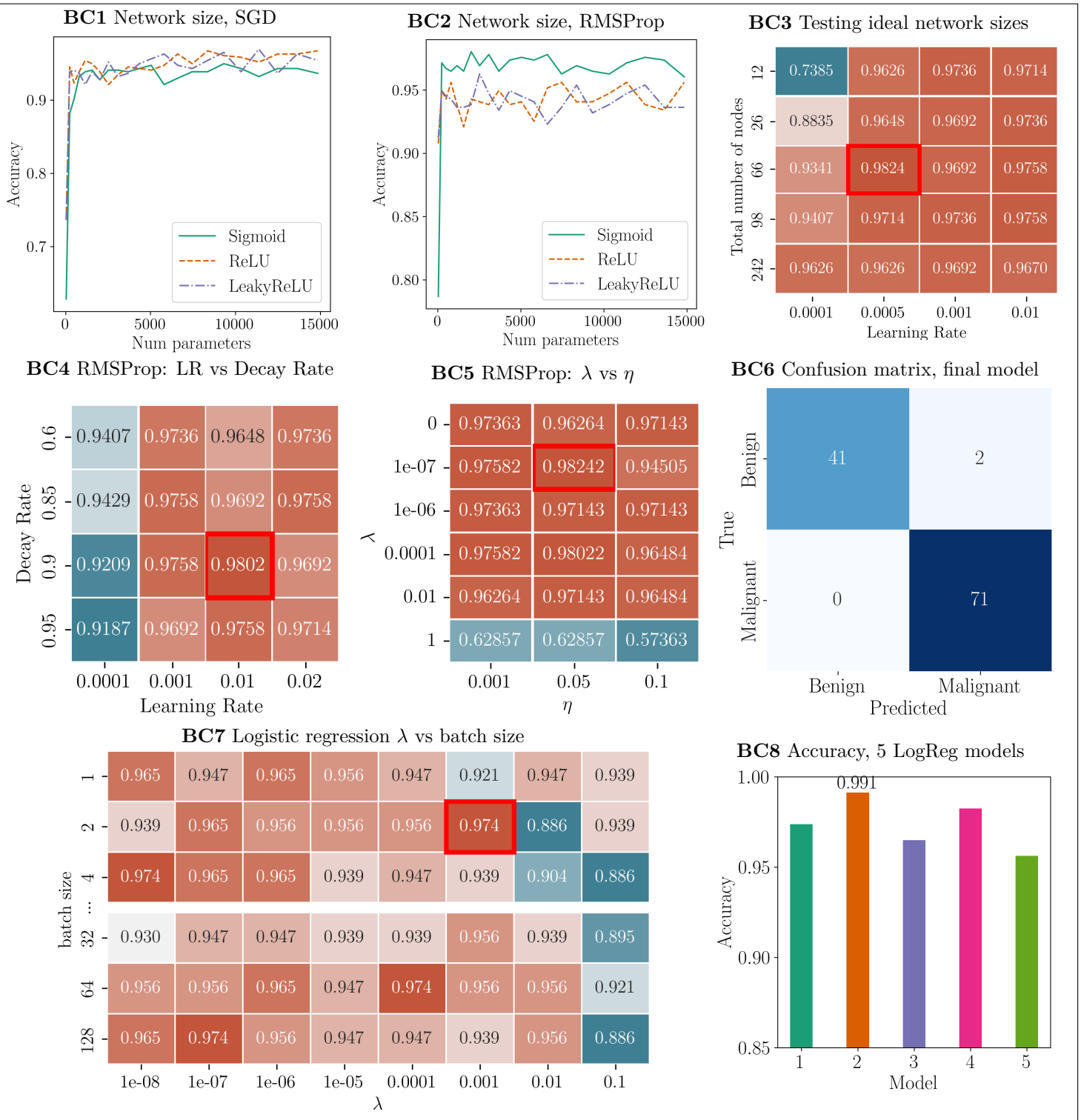
| **Dataset** | **FFNN** | **Logistic** |
|---|---|---|
| BCW (Acc.) | 0.9824 [BC5] | 0.991 [BC8] |

Table 2: Comparing the results of our feed forward network and the logistic regression model

From studying the results in Table 2, we see that both logistic regression and the neural network achieve high accuracy on the Breast Cancer Wisconsin (Diagnostic) dataset. The feed-forward neural network may require more fine tuning of hyperparameters, reaching similar accuracy levels when properly optimized. Since logistic regression provides a simpler setup, this would now be our obvious choice. This is on par with our findings with Franke's Function in the previous section.

**Note on Benchmarking with PyTorch**

For both the approximation and the classification tasks, a PyTorch network is set up with the identical layer depth, activation functions, optimizer, and with the same hyper parameters as our custom networks. Special care is taken in initializing the weights and biases, to ensure that these are normally distributed $\sim \mathcal{N}(0, 1)$, equal to our models. Benchmarking with PyTorch renders only slightly better results (one example is shown in Figure **F8**), and we take this as proof of valid findings from our own implemented model. Please refer to our `repository` for further plots.




**Testing FFNN with Breast Cancer Wisconsin (Diagnostic) dataset: BC1** Accuracy vs. number of parameters, using SGD optimizer. 2 hidden layers with 32 neurons. Epochs: 30, lr: 0.001, batch size: 4, loss: Cross Entropy Loss. $k$ folds with $k = 5$. **BC2** Accuracy vs. number of parameters, using RMSProp optimizer. 2 hidden layers with 32 neurons. Epochs: 30, lr: 0.001, $\rho = 0.9$, batch size: 4, loss: Cross Entropy Loss. $k$ folds with $k = 5$. **BC3** Testing different network sizes vs learning rates: 12: `[10, 2]`, 26: `[16, 8, 2]`, 66: `[32, 32, 2]`, 98: `[64, 32, 2]`, 242: `[128, 64, 32, 16, 2]`. Epochs: 30, batch size: 4, loss: Cross Entropy Loss. $k$ folds with $k = 5$. **BC4** Learning rate vs. tuning of RMSProp's $\rho$ parameter. 2 hidden layers with 32 neurons. Epochs: 30, batch size: 4, loss: Cross Entropy Loss. $k$ folds with $k = 5$. **BC5** Testing different values of the regularization term $\lambda$ vs. different learning rates ($\eta$). 2 hidden layers with 32 neurons. Epochs: 30, batch size: 4, loss: Cross Entropy Loss. $k$ folds with $k = 5$. **BC6** Confusion matrix for final network, after a prediction run on the test set, resulting in accuracy = 0.9824. The labels *Benign* and *Malignant* correspond to `False/0` and `True/1` in the dataset. Setup: 2 hidden layers with 32 neurons. Epochs: 30, batch size: 4, lr: 0.01, $\rho$: 0.9, $\lambda$: 1e-7, loss: Cross Entropy Loss. **BC7** Heatmap showing best accuracy based on batch sizes and regularization term $\lambda$. SGD, Cross Entropy Loss, epochs: 40, learning scheduler. **BC8** Bar plot showing accuracy from independently run Logistic regression models, $n = 5$.

## CONCLUSION

In this project, we set out to investigate the application of feed-forward neural networks in two different tasks: approximating synthetic data generated from Franke's Function and classifying real medical data from the Breast Cancer Wisconsin (Diagnostic) dataset. After extensive experimentation and model fine-tuning, we achieved strong results in both cases. We then evaluated the performance of our optimized neural networks by comparing them with traditional machine learning techniques: linear regression for Franke's Function, and logistic regression for the breast cancer dataset. This comparison helped us build intuition on the strengths and limitations of each approach.

| **Dataset** | **FFNN** | **Log/Lin** |
|---|---|---|
| FF (MSE) | 0.01195 [F7] | 0.0103 [2] |
| BCW (Acc.) | 0.982 [BC5] | 0.991 [BC8] |

Table 3: Comparison of Neural Network, Linear Regression, and Logistic Regression on Franke's Function (FF) and Cancer Data (BCW)

Our findings indicate that the neural network performs well on both tasks; however, traditional machine learning techniques slightly outperforms it in each case, as shown in Table 3. Although the neural network demonstrates impressive adaptability across these two diverse tasks, when considering simplicity, the traditional techniques are clearly the optimal choice for these specific problems. Nevertheless, our intuition is that a neural network would generalize more effectively to more complex tasks.

With additional time, we would have investigated the limitations of these traditional machine learning techniques, exploring the complexities required for a neural network to surpass them.

On a final note: Bringing our machine learning knowledge further into Neural Networks has provided insight into a model that can do almost everything. However it comes with its matehematical challenges, and training a machine learning model takes a significant amount of time. Choosing the correct path or order to solve this puzzle can feel like a mystic process, almost calling for a near-magical intuition. Then, when it is finally time to train the model, using your finest hardware could consume several kilowatt-hours of power. This can explain why AI data centers are so power-hungry. Let's remind ourselves about the brain's consumption again: If you spent an hour reading our report, you will have used 20 watt, corresponding in *kcals* to half a banana. The increasing global power consumption of the entire machine learning field is pushing us into realising that feeding everything into a deep neural network is not necessarily the future. [18] For certain types of data, simpler, analytical models combined with sufficient domain knowledge, and statistical and mathematical insights *could* prove a viable alternative.

## REFERENCES


[1] Lex Fridman. Yann Lecun: Meta AI, Open Source, Limits of LLMs, AGI & the Future of AI (Podcast #416, 2023).

[2] O.A. Brovold, E.G. Hansen, H.G. Kornstad. Regression Analysis and Resampling Techniques in Applied Machine Learning, November 2024.

[3] R. Franke. A Critical Comparison of Some Methods for Interpolation of Scattered Data. Technical Report NPS-53-79-003, Naval Postgraduate School, 1975.

[4] CodeCarbon.io.

[5] Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong. *Mathematics for machine learning.* Cambridge University Press, Cambridge, UK New York, NY, 2020.

[6] Rauf Bhat. Gradient Descent With Momentum, October 2020.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016.

[8] Geoffrey Hinton. Lecture 6a - Overview of mini-batch gradient descent. *Neural Networks for Machine Learning.*

[9] Paolo Perrotta. *Programming machine learning: from coding to deep learning*. Pragmatic Bookshelf, Raleigh, North Carolina, 2020. OCLC: 1155169278.

[10] Trevor Hastie, Robert Tibshirani, and J. H. Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer series in statistics. Springer, New York, NY, 2nd ed edition, 2009.

[11] What Is Logistic Regression? | IBM, August 2021.

[12] 3Blue1Brown. Backpropagation calculus | Chapter 4, Deep learning, November 2017.

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, and others. PyTorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, volume 32, 2019.

[14] Daniel Jurafsky and James Martin. Chapter 5 -Logistic Regression. August 2024.

[15] Olvi Mangasarian William Wolberg. Breast Cancer Wisconsin (Diagnostic), 1993.

[16] scikit-learn/sklearn/linear_model/_base.py at f5aac217372759d4d35d69934199be878c3bcc65 · scikit-learn/scikit-learn.

[17] Morten Hjorth-Jensen. Applied Data Analysis and Machine Learning — Applied Data Analysis and Machine Learning, 2021.

[18] Jeroen Van Der Donckt, Jonas Van Der Donckt, Emiel Deprost, Nicolas Vandenbussche, Michael Rademaker, Gilles Vandewiele, and Sofie Van Hoecke. Do not sleep on traditional machine learning. *Biomedical Signal Processing and Control*, 81:104429, March 2023.

## APPENDIX

**Derivation of binary cross entropy loss, with sigmoid as activation function**

$$L = -\boldsymbol{y}\log(\hat{\boldsymbol{y}}) - (1-\boldsymbol{y})\log(1-\hat{\boldsymbol{y}})$$

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w} &= \frac{\partial z}{\partial w}\frac{\partial a}{\partial z}\frac{\partial \mathcal{L}}{\partial a} = X^\top(\hat{\boldsymbol{y}} - \boldsymbol{y}) \\
\frac{\partial \mathcal{L}}{\partial a} &= \frac{\boldsymbol{y}}{\hat{\boldsymbol{y}}} + \frac{1-\boldsymbol{y}}{1-\hat{\boldsymbol{y}}} \\
\frac{\partial a}{\partial z} &= \sigma(z)(1-\sigma(z)) = \hat{\boldsymbol{y}}(\mathbf{1}-\hat{\boldsymbol{y}}) \\
\frac{\partial \mathcal{L}}{\partial z} &= -\boldsymbol{y}(1-\hat{\boldsymbol{y}}) + (1-\boldsymbol{y})\hat{\boldsymbol{y}} \\
&= -\boldsymbol{y} + \boldsymbol{y}\hat{\boldsymbol{y}} + \hat{\boldsymbol{y}} - \boldsymbol{y}\hat{\boldsymbol{y}} \\
&= \hat{\boldsymbol{y}} - \boldsymbol{y} \\
\frac{\partial z}{\partial w} &= X^\top
\end{aligned}$$

**Calculation for iterations given epochs and batch-size**

When training a neural network, varying the batch size for a fixed number of epochs can lead to significant differences in the total number of iterations. To maintain a consistent number of iterations across different batch sizes, it's helpful to adjust the number of epochs accordingly.

You can calculate the necessary number of epochs for a given batch size, sample size, and target total iterations using the formula:

$$\text{epochs} = \frac{\text{iterations} \times \text{batch size}}{\text{samples}}$$

**Backprop algorithm**

The algorithm for backpropagation starts with the output layer. After a forward pass, the gradient of the cost function $J$ is then calculated:

$$g \leftarrow \nabla_{\hat{y}} J = \nabla_{\hat{y}} \mathcal{L}(\hat{y}, y),$$

where $\hat{y}$ is the prediction, $y$ is the ground truth and $\mathcal{L}$ is the loss we wish to minimize.

Then, for every layer $k$, starting reversed from the output layer we then calculate the gradient of $J$ w.r.t the activation unit:

$$g \leftarrow \nabla_{a^{(k)}} J = g \odot f'(a^{(k)}),$$

where $\odot$ represents element-wise multiplication. We calculate the gradients for the weights and biases, using regularization tecniques if necessary.

Finally, we propagate the gradients to the next layer, calculating them based on the activation unit in the lower layer:

$$g \leftarrow W^{(k)\top} g,$$

where $W^{(k)\top}$ is the transposed weight matrix from the current layer. [7]