University of Oslo

FYS-STK3155

Project 2

Brage Wiseth bragewi@ifi.uio.no
Felix Cameren Heyerdahl felixch@ifi.uio.no
Eirik Bjørnson Jahr eiribja@ifi.uio.no

November 19, 2023

https://github.com/bragewiseth/MachineLearningProjects

Contents

Abstract

In this study, we explore the effectiveness of neural networks in solving classification and regression problems, contrasting their performance with traditional logistic regression models. Utilizing the Wisconsin breast cancer dataset, we compared the accuracy of these methods in tumor classification. Our results show that neural networks achieved a classification accuracy of 95%, compared to logistic regression's 96%. Both are similar to the 96% achieved using SKLearn's models. Additionally, we examined the application of neural networks to regression problems, finding that they can approximate 2-dimensional perlin nose, with a mean squared error of 0.02, compared to linear regression's 0.04. This shows that both neural networks and logistic regression are powerful tools for classification tasks, but that neural networks seems to be more viable for complex regression tasks. Offering a more flexible framework for approximating functions

Keywords: Regression, Classification, Neural Networks

Introduction

Real-world applications often require accurate classification between different classes. A notable example is in cancer research, where it's crucial to classify tumors as either benign or malignant. This classification typically involves analyzing a set of features extracted from the tumor. However, these features may not always clearly distinguish between classes, particularly when they are similar for both or when there is a large number of features. This complexity can pose a challenge for human analysis. To address this, *logistic regression* is often employed. This method involves using regression to produce a continuous output, which is then transformed into a binary value through an activation function, such as the sigmoid function[1] or the Heaviside function. The resulting binary value represents the data point's classification. However, logistic regression has its limitations, particularly when dealing with data that is not linearly separable. In such cases, logistic regression cannot effectively draw a boundary to separate the classes, leading to classification errors. This issue is examined in more detail in Appendix B. As an alternative, *neural networks* offer a more versatile solution. Capable of approximating any continuous function with a sufficient number of neurons in the hidden layers[5], neural networks provide a robust tool for both classification and regression problems. Their ability to learn complex, non-linear relationships makes them particularly powerful for tasks where traditional logistic regression falls short. The application of neural networks in regression is explored in Appendix A.

In our initial project[11], we utilized an analytical approach to derive optimal parameters for linear regression. However, the introduction of activation functions in more complex models, such as neural networks, breaks this linearity. Consequently, the same analytical methods cannot be applied to determine optimal parameters. Instead, we shift towards an iterative approach. For cost functions that are continuous and differentiable, gradient descent becomes a viable method to locate the function's minimum. The models explored in this project predominantly rely on gradient descent.

The structure of this report is as follows:

- In Section 2, we introduce the logistic regression model.

[1] The sigmoid function outputs a value between 0 and 1. By setting a threshold, such as 0.5, values above this threshold can be classified as 1, and those below as 0. This output can be interpreted as the probability of the data point belonging to class 1.

- In Section 3, we discuss gradient descent and backpropagation, key techniques in the optimization of machine learning models.
- In Section 4, we delve into the fundamentals of neural networks.
- Section 5 covers the data used in our experiments and analyses.
- Section 6 presents our results and provides a discussion on their implications.
- Section 7 concludes the report, summarizing key findings and insights.
- Appendix A explores regression using neural networks.
- Appendix B investigates the universal approximation theorem and its applications.

1. Gradient Descent

Gradient descent is an iterative optimization algorithm used to find the minimum of a function. The fundamental principle behind gradient descent is to iteratively move towards the minimum of the function by taking steps proportional to the negative of the gradient (or approximate gradient) at the current point. Unlike the Newton-Raphson method, which is used for finding the roots of a function and involves the second derivative, gradient descent primarily utilizes the first derivative. While Newton-Raphson converges faster under certain conditions due to its use of second-order information, gradient descent is more widely applicable as it does not require the computation of the second derivative, making it simpler and more computationally efficient in many scenarios. In the context of machine learning, gradient descent is employed to minimize a cost function, which is a measure of how far a model's predictions are from the actual outcomes. The algorithm updates the parameters of the model in a way that the cost function is gradually reduced.

There are several variants of gradient descent, each with its unique characteristics:

- **Batch Gradient Descent:** This form computes the gradient of the cost function with respect to the parameters for the entire training dataset. While this can be computationally expensive [2], it produces a stable gradient descent convergence.
- **Stochastic Gradient Descent (SGD):** In contrast to batch gradient descent, SGD updates the parameters for each training example one by one. It can be faster and can also help to escape local minima, but the path to convergence can be more erratic.
- **Mini-batch Gradient Descent:** This is a compromise between batch and stochastic gradient descent. It updates the parameters based on a small, randomly selected subset of the training data. This variant provides a balance between the efficiency of SGD and the stability of batch gradient descent.

[2] depending on the way computational resources are utilized, it is possible to parallelize the computation of the gradient for each training example, making it more efficient. For our case using batch gradient descent on a GPU is much faster than using stochastic gradient descent.

The choice of gradient descent variant and its parameters, like the learning rate, significantly impacts the performance and convergence of the algorithm. An appropriately chosen learning rate ensures that the algorithm converges to the minimum efficiently, while a poorly chosen rate can lead the algorithm to diverge or get stuck in a local minimum. Furthermore, advanced optimization algorithms based on gradient descent, such as Adam and RMSprop, have been developed to address some of the limitations of the traditional gradient descent method, offering adaptive learning rates and momentum features.

1.1 Backpropagation and Chain Rule

In machine learning, particularly in training neural networks, calculating derivatives is a fundamental process. For simpler models, such as linear regression, deriving these derivatives is relatively straightforward. We typically compute the gradient of the cost function with respect to the model parameters. However, when dealing with more complex models like neural networks, the process of calculating these derivatives becomes more intricate due to the architecture of the network. Neural networks consist of multiple layers of interconnected nodes, where each node represents a mathematical operation. The output of one layer serves as the input for the next, creating a chain of functions. When the loss function of the network is a composition of several such functions, applying the chain rule becomes essential for computing gradients. This process is known as backpropagation.

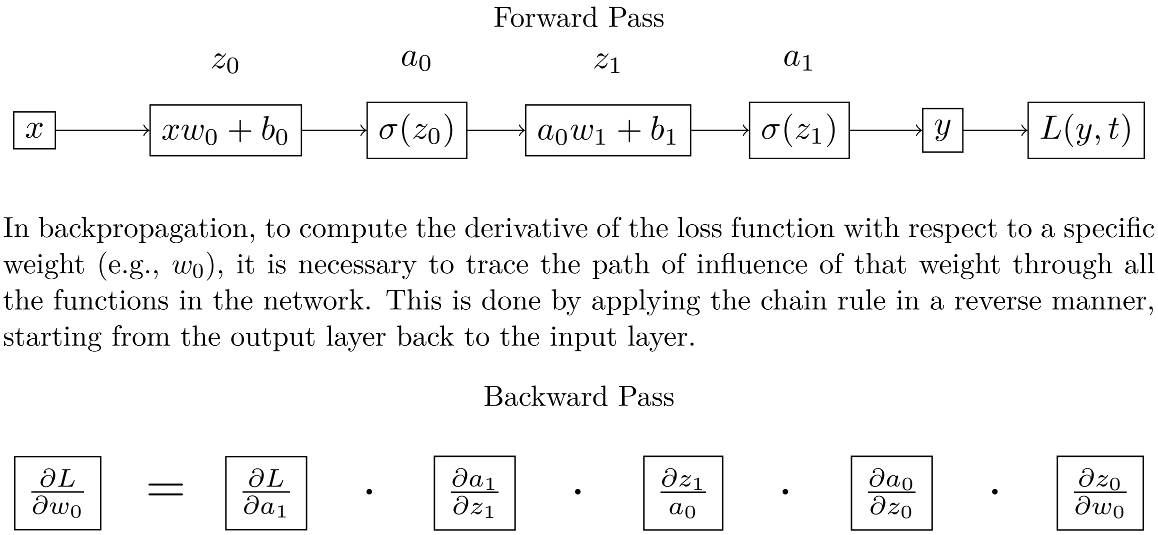

Forward Pass
z_0
a_0
z_1
a_1
x
$xw_0 + b_0$
$\sigma(z_0)$
$a_0 w_1 + b_1$
$\sigma(z_1)$
y
$L(y,t)$
In backpropagation, to compute the derivative of the loss function with respect to a specific weight (e.g., w_0), it is necessary to trace the path of influence of that weight through all the functions in the network. This is done by applying the chain rule in a reverse manner, starting from the output layer back to the input layer.
Backward Pass
$\frac{\partial L}{\partial w_0} = \frac{\partial L}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{a_0} \cdot \frac{\partial a_0}{\partial z_0} \cdot \frac{\partial z_0}{\partial w_0}$

Figure 1: Illustration of the Chain Rule in Forward and Backward Passes

Key Concepts in Backpropagation:

- **Forward Pass:** In the forward pass, inputs are passed through the network, layer by layer, to compute the output. Each node's output is a function of its inputs, which are the outputs of nodes in the previous layer.
- **Backward Pass:** The backward pass is where backpropagation is applied. After computing the loss (the difference between the predicted output and the actual out-

put), we propagate this loss backward through the network. This involves applying the chain rule to compute partial derivatives of the loss with respect to each weight in the network.

- **Gradient Descent:** The gradients computed through backpropagation are used to update the weights of the network. This is typically done using gradient descent or one of its variants, where the weight update is proportional to the negative gradient, aiming to minimize the loss function.
- **Activation Functions:** The role of activation functions in each neuron is crucial. They introduce non-linearities into the model, allowing neural networks to learn complex patterns. Derivatives of these activation functions play a significant role in the backpropagation process.

By iteratively performing forward and backward passes and updating the model parameters using gradient descent, neural networks can effectively learn from data. Backpropagation is thus a cornerstone technique in neural network training, enabling these models to capture and learn from complex patterns in data.

2. Neural Networks

Neural networks extend the principles of logistic regression to more complex architectures, enabling the modeling of a wider range of nonlinear relationships. At their core, neural networks can be conceptualized as a series of logistic regression models interconnected in a network structure. This similarity allows for a degree of code reusability across logistic regression, linear regression, and neural network implementations.

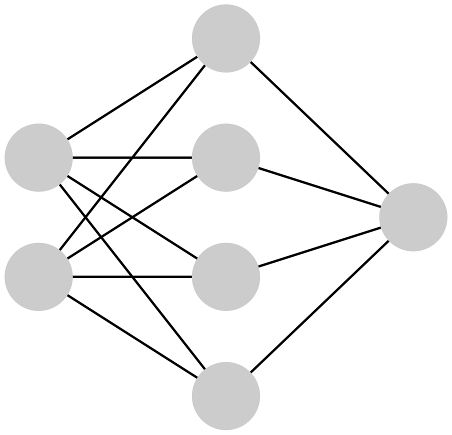
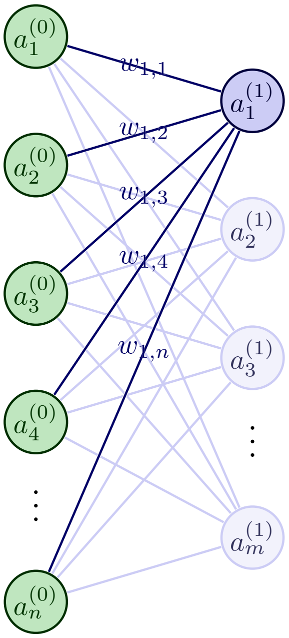


Figure 2: A model of a neural network with one hidden layer consisting of four nodes [6]

One of the defining features of neural networks is the use of activation functions. These functions introduce non-linearity into the model, allowing the network to learn complex patterns. Common examples of activation functions include the sigmoid, tanh, and ReLU



$$= \sigma\left(w_{1,0}a_0^{(0)} + w_{1,1}a_1^{(0)} + \ldots + w_{1,n}a_n^{(0)} + b_1^{(0)}\right)$$
$$= \sigma\left(\sum_{i=1}^{n} w_{1,i}a_i^{(0)} + b_1^{(0)}\right)$$

$$\begin{pmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_m^{(1)} \end{pmatrix} = \sigma\left[\begin{pmatrix} w_{1,0} & w_{1,1} & \ldots & w_{1,n} \\ w_{2,0} & w_{2,1} & \ldots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,0} & w_{m,1} & \ldots & w_{m,n} \end{pmatrix} \begin{pmatrix} a_1^{(0)} \\ a_2^{(0)} \\ \vdots \\ a_n^{(0)} \end{pmatrix} + \begin{pmatrix} b_1^{(0)} \\ b_2^{(0)} \\ \vdots \\ b_m^{(0)} \end{pmatrix}\right]$$

$$a^{(1)} = \sigma\left(\mathbf{W}^{(0)}a^{(0)} + \mathbf{b}^{(0)}\right)$$

Figure 3: Illustration of how a single node gets its value during a forward pass. This shows that matrix operations can be used for more efficient calculations. When a is a batch of inputs, these calculations become matrix-matrix multiplications, significantly speeding up both training and inference.
[6]

functions. Each has unique characteristics that make them suitable for different types of neural network architectures. In a neural network with no hidden layers, the model essentially becomes a linear regression model, represented by the equation $y = x_0w_0 + x_1w_1 + x_2w_2 + ... + x_nw_n + b_0$. By introducing an activation function at the output layer, the model transforms into a logistic regression model, suitable for binary classification tasks. The flexibility of neural networks arises from their ability to incorporate multiple hidden layers with a variety of activation functions, enabling them to capture complex relationships in data. Backpropagation is the mechanism through which neural networks learn. By calculating the derivatives of the loss function with respect to the network's parameters, backpropagation allows for the adjustment of these parameters in a way that minimizes the loss. This process is essential for training neural networks and is applicable across different network architectures, from simple single-layer networks to deep, multi-layered structures. To summarize, neural networks are a powerful extension of logistic and linear regression models, capable of handling a broad spectrum of tasks from simple regression to complex pattern recognition. Their versatility and adaptability, combined with the efficiency of matrix operations, make them a cornerstone technology in the field of machine learning.

3. Data

For our classification task, we will utilize the widely recognized Wisconsin Breast Cancer Dataset[12]. This dataset comprises 569 data points, each with 30 distinctive features. These features are derived from digitized images of a fine needle aspirate (FNA) of breast masses, focusing on various characteristics of the cell nuclei depicted in the images. The dataset quantifies several attributes for each cell nucleus, including radius, texture, perimeter, area, smoothness, compactness, concavity, concave points, symmetry, and fractal dimension. For each attribute, three types of measurements are provided: the mean, standard error, and the ”worst” or largest (which represents the mean of the three largest values). This results in a total of 30 features for each data point. For instance, field 1 represents the Mean Radius, field 11 is Radius SE, and field 21 corresponds to the Worst Radius. All feature values in this dataset are recorded with four significant digits, and there are no missing attribute values. The class distribution within the dataset is as follows: 357 benign cases and 212 malignant cases. The primary objective is to classify these tumors as either benign or malignant based on the features provided. In this context, a positive result indicates a benign tumor, while a negative result signifies a malignant tumor. Essentially, our goal is to accurately identify cases of cancer and classify them as negative.

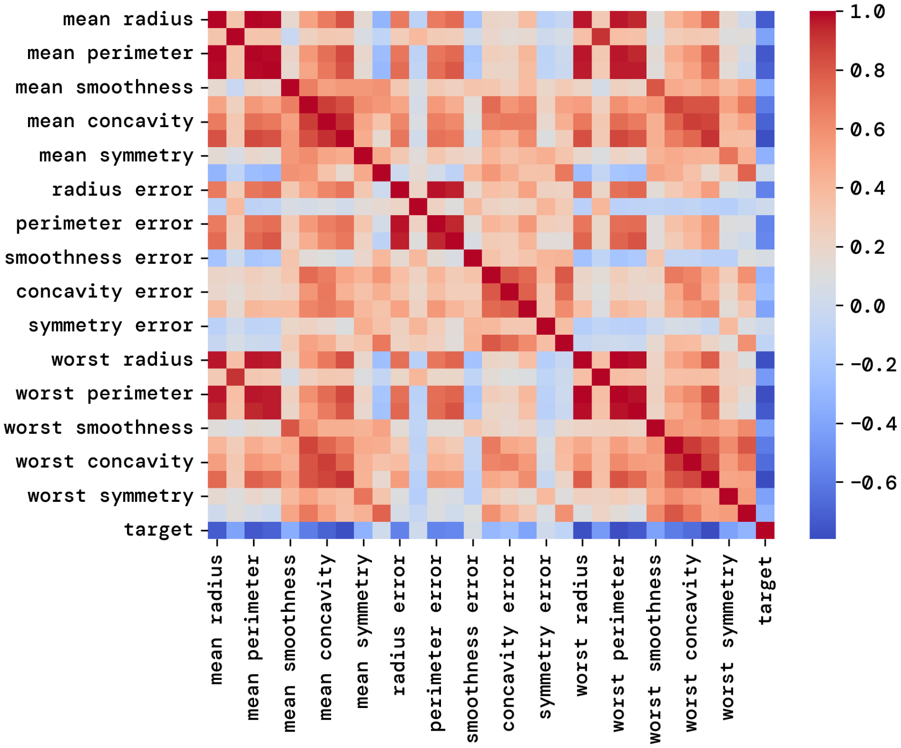


Figure 4: Feature correlation matrix. This provides insight into the redundancy among features. Only every other feature is annotated, but the missing labels can be inferred from Figure 5.

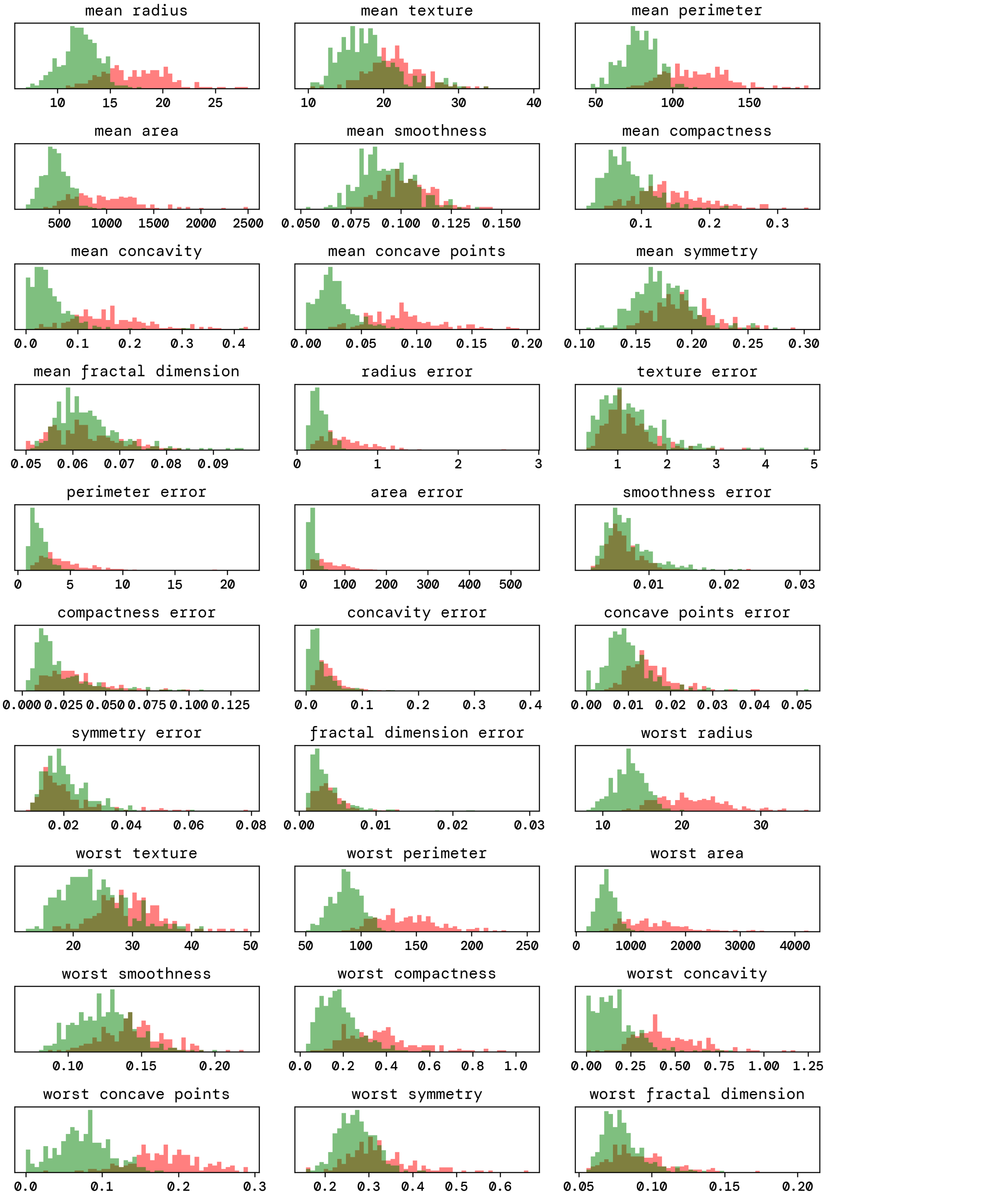


Figure 5: Feature histogram. The red class (0) represents malignant cases, while the green class (1) indicates benign cases.

4. Results and Analysis

In our experiments, we employed the cross-entropy loss function, The gradient of the loss function with respect to the model's parameters was computed using automatic differentiation. We used the SKLearn[9] library to perform cross-validation for a more robust evaluation of our models. We used k-fold cross-validation with $k = 6$ folds. The results presented in this section are based on the average of the results from the 6 folds. The bar plots also show the standard deviation of the results from the 6 folds.

4.1 Hyperparameters

During our hyperparameter tuning phase, we observed interdependencies between certain hyperparameters. Notably, the learning rate and the number of epochs showed a significant correlation, as did the batch size and the learning rate. Such dependencies imply that these hyperparameters cannot be optimized independently for the most effective training process. Ideally, a comprehensive grid search across multiple dimensions of hyperparameters would be conducted. However, due to constraints in time and computational resources, our experiments were limited. We were able to conduct grid searches, but these were restricted to two dimensions at most. While this limitation prevented a thorough exploration of the hyperparameter space, the results from these partial grid searches provided valuable insights. They served as indicators, pointing us towards regions in the hyperparameter space where optimal settings might exist. The analysis of these results suggests that while we have identified promising hyperparameter settings, a more exhaustive search might yield further improvements. We use the same base hyperparameters when we tune the other hyperparameters.

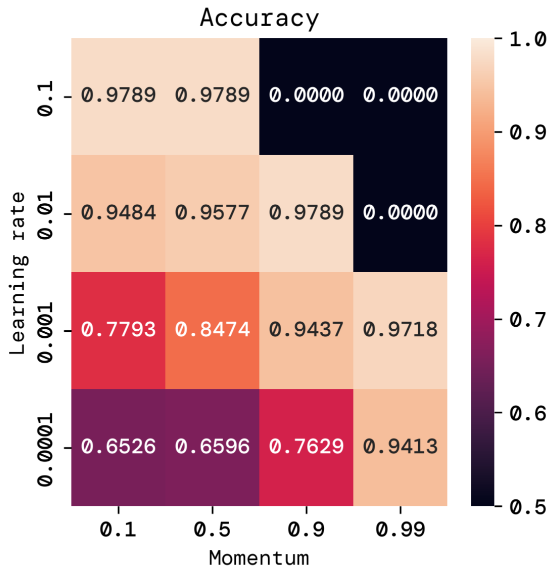


Figure 6: Accuracy versus learning rate and momentum.

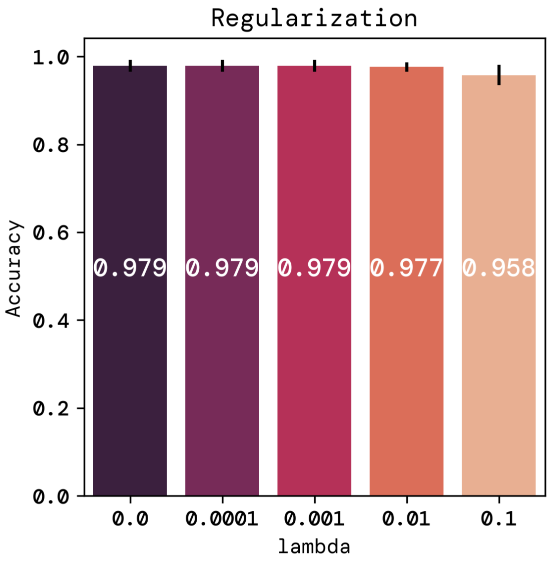


Figure 7: Accuracy versus regualarization.

Our findings indicate that incorporating momentum is beneficial, particularly with a value of around 0.9, which emerged as the most effective in our tests. This momentum allows for a higher learning rate, enhancing the training process. In contrast, excessive regularization seems to adversely affect model performance.

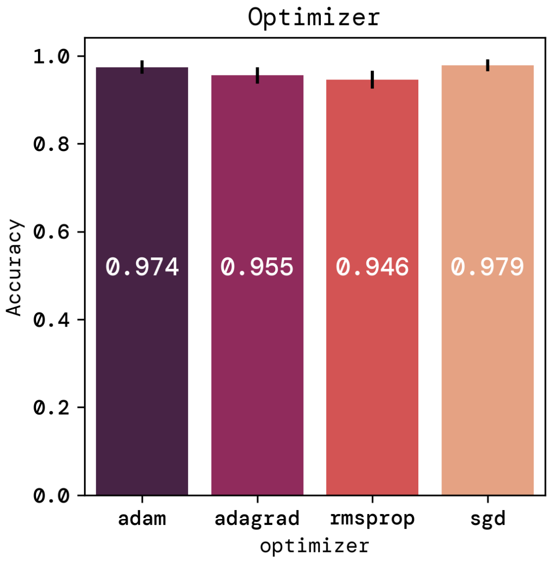


Figure 8: Model accuracy across different optimizers. here sgd is standard gradient descent.

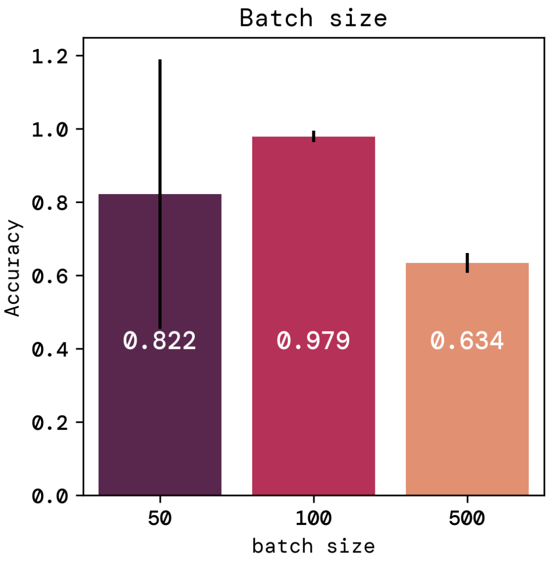


Figure 9: Accuracy based on batch size.

The choice of optimizer did not significantly impact the model's performance. However, Adam and standard gradient decent with momentum showed a slight advantage over others. The batch size also played a role, with certain sizes yielding better accuracy. With a batch size of 50, the model accuracy varied greatly between folds.

In order for our model to correctly classify the data, it is essential that the last layer's activation function is a sigmoid function. This is because the sigmoid function outputs a value between 0 and 1, which can be interpreted as the probability of the data point belonging to class 1. The choice of activation function for the hidden layers did not significantly impact the model's performance. We used one hidden layer with 2 nodes when testing actovation functions.

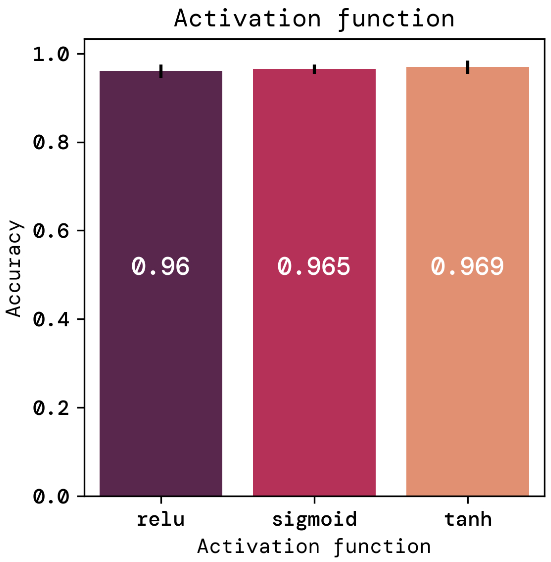


Figure 10: Model performance with different activation functions.

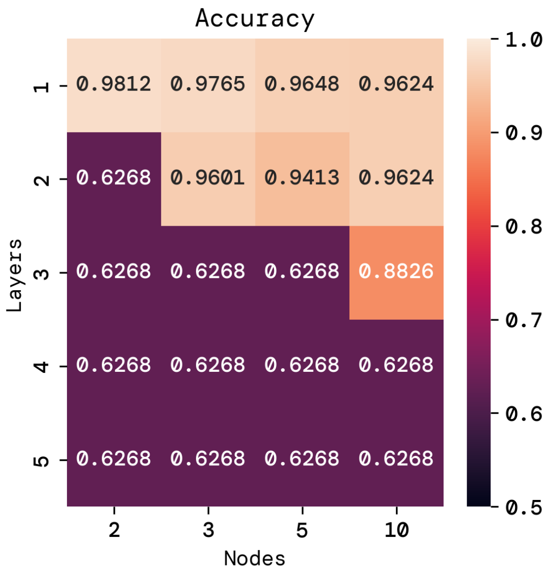


Figure 11: Accuracy in relation to the number of layers and nodes.

4.2 Final Evaluation and Comparisons

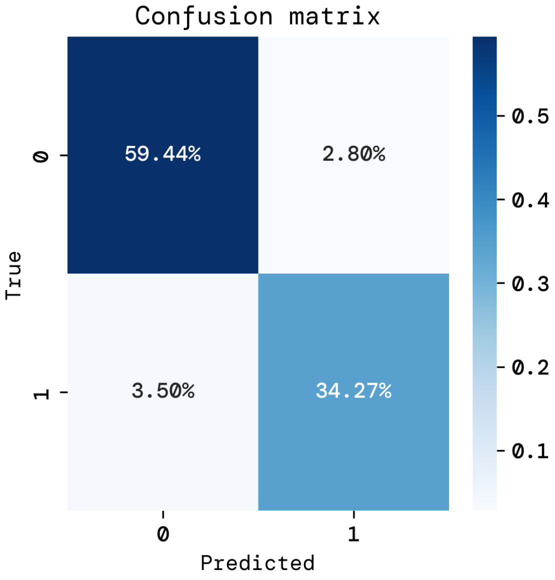


Figure 12: Confusion matrix for the best model. The model has an accuracy of 96% on the test set.

https://github.com/bragewiseth/MachineLearningProjects

5. Conclusion

Appendix A

Appendix A. Regression with Neural Networks

In our previous project[11], we explored the fitting of linear models and polynomials to data, deriving optimal parameters β through an analytical approach. This method demonstrated that, theoretically, polynomials can approximate any function, given a sufficiently high degree of freedom. However, this approach has practical limitations. Providing a model with infinite degrees of freedom is not feasible due to computational constraints and the risk of overfitting. How can we approximate complex functions without resorting to high-degree polynomials? A promising solution is to construct models that are not based on a single high-degree polynomial but are instead composed of simpler, interconnected components. This is where neural networks come into play. Neural networks offer a flexible architecture that can approximate a wide range of functions. By 'gluing together' multiple simple functions, such as linear segments in the case of basic neural networks, they can model complex relationships in the data. Unlike polynomials, neural networks are not limited to a specific functional form and can be adapted for both regression and classification tasks. Their layered structure, where each layer can learn different aspects of the data, allows them to capture both simple and intricate patterns.

Thus, neural networks provide a powerful and versatile alternative to polynomial regression, capable of handling diverse data modeling challenges, as we will explore in this section.

Data

For regression we will be using perlin noise to generate our dataset

Solving a regression problem with linear regression and neural networks using gradient optimization.

Results and Analysis

Hyperparameters

Final Evaluation and Comparisons

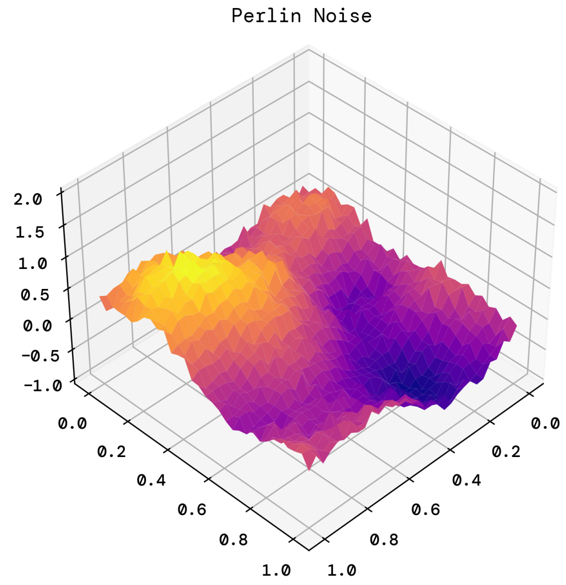


Figure 13

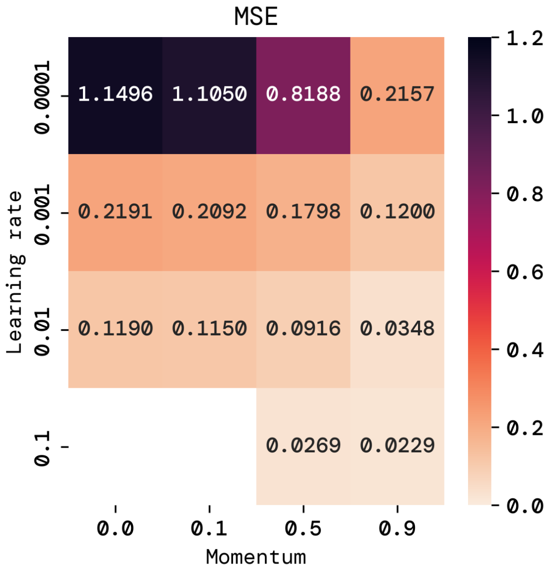


Figure 14: In this case havving momentum seems to be beneficial. We maxes out our testing range and found that 0.9 was the best value for momentum. momentum allows a higher learning rate

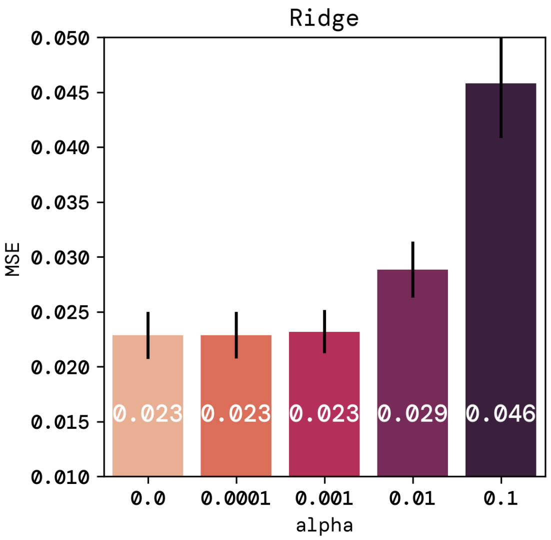


Figure 15: Introduction regualarization does not seem to yield any benefits, in fact having too much regualarization shunts performance

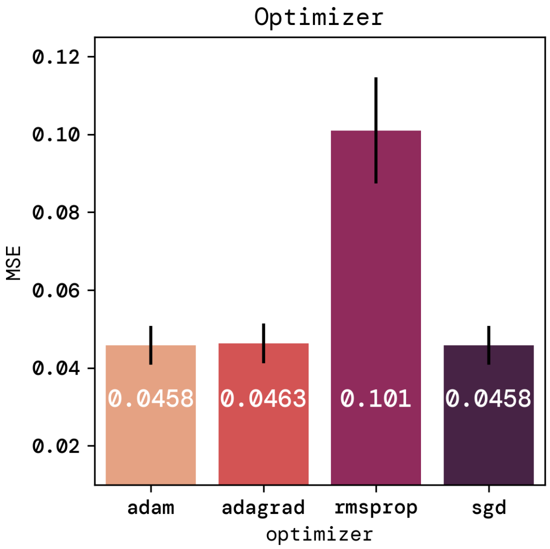


Figure 16: In this case havving momentum seems to be beneficial. We maxes out our testing range and found that 0.9 was the best value for momentum. momentum allows a higher learning rate

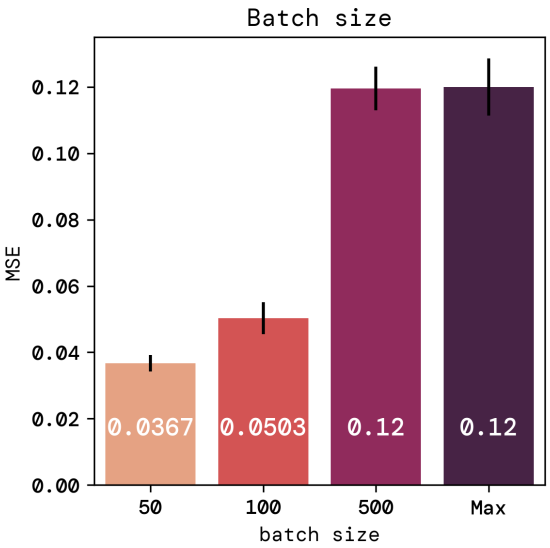


Figure 17: Introduction regualarization does not seem to yield any benefits, in fact having too much regualarization shunts performance

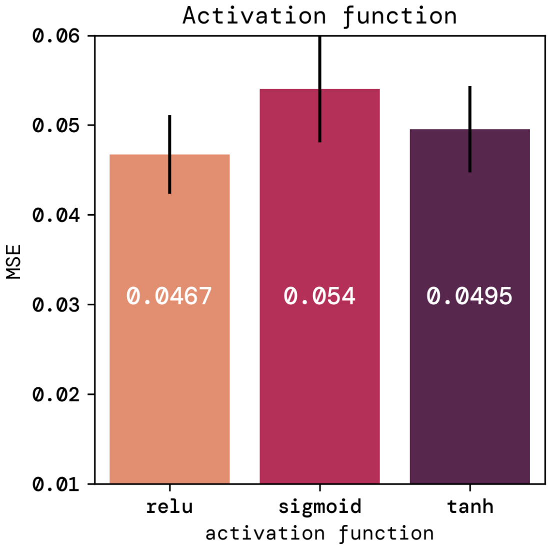


Figure 18: In this case havving momentum seems to be beneficial. We maxes out our testing range and found that 0.9 was the best value for momentum. momentum allows a higher learning rate

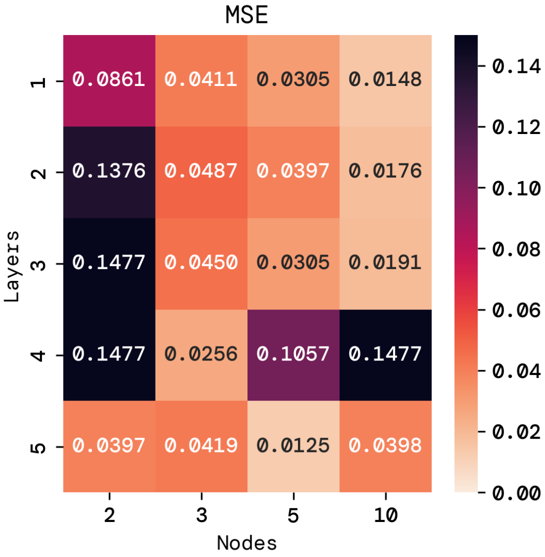


Figure 19: Introduction regualarization does not seem to yield any benefits, in fact having too much regualarization shunts performance

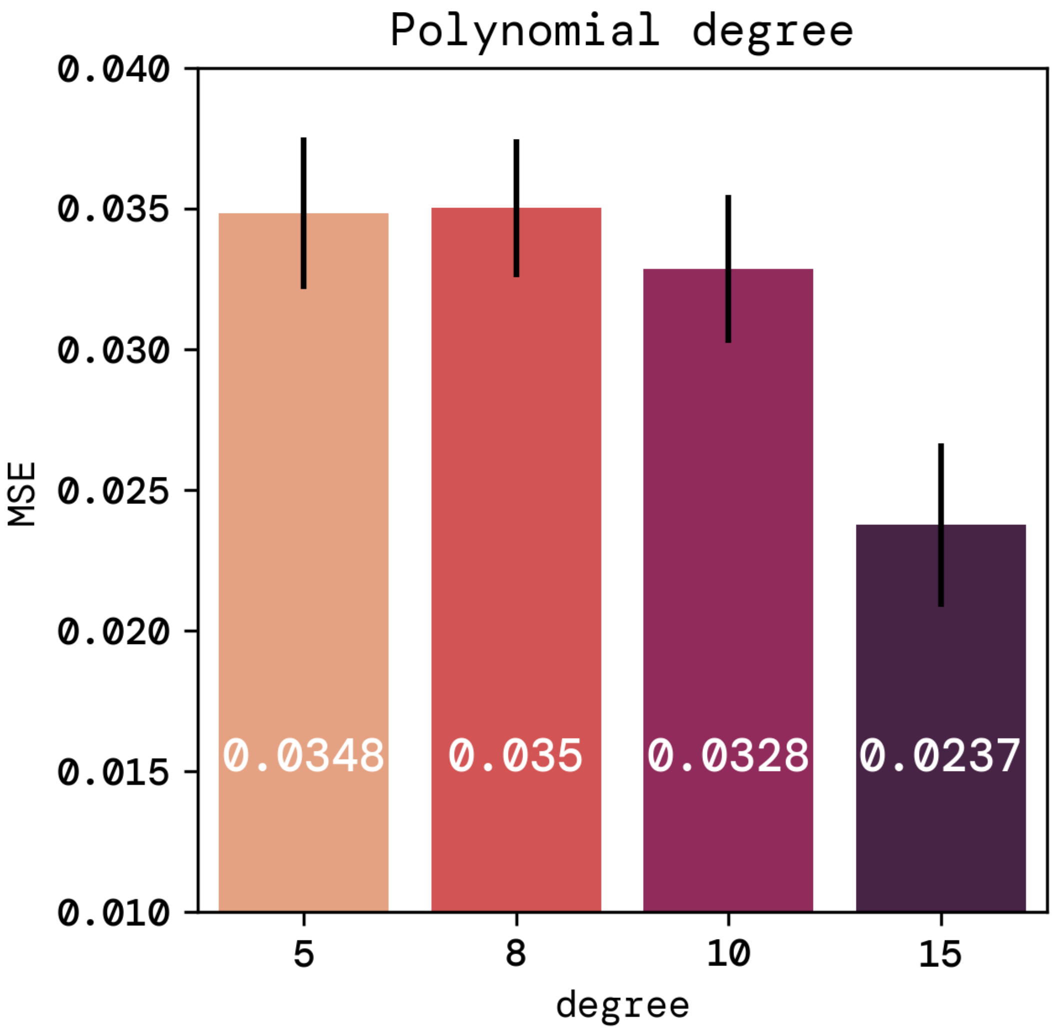

Polynomial degree
MSE
0.040
0.035
0.030
0.025
0.020
0.015
0.010
0.0348
0.035
0.0328
0.0237
5
8
10
15
degree

Figure 20

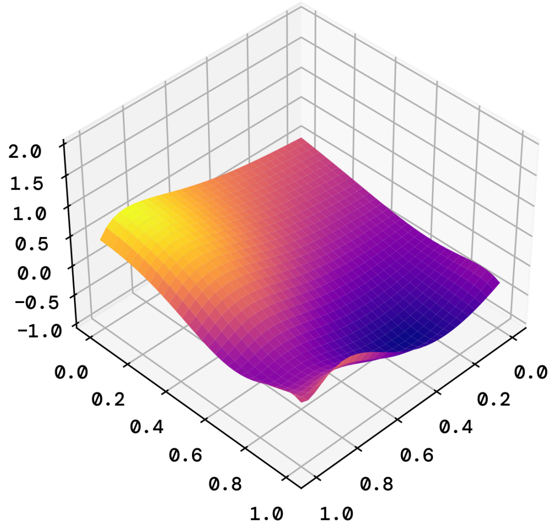


Figure 21: In this case havving momentum seems to be beneficial. We maxes out our testing range and found that 0.9 was the best value for momentum. momentum allows a higher learning rate

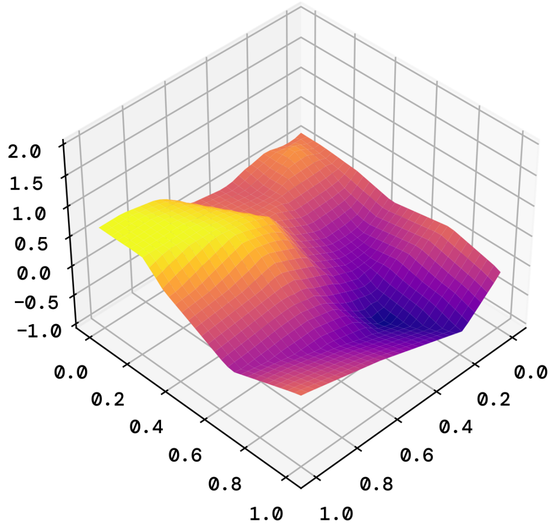


Figure 22: Introduction regualarization does not seem to yield any benefits, in fact having too much regualarization shunts performance

Appendix B

Appendix B. Universal Approximation

Where is the limit of what we can learn with a neural network? Can we approximate any function? There are several ways to approximate a function. For periodic functions it may be wise to use a Fourier series, another option is to use a Taylor series. If these mehtods don't float your boat, we can in fact use a neural network instead. A neural network with only one hidden layer can approximate any continuous function given enough neurons in the hidden layer. The more neurons, the higher the resolution of the Approximation [5].

XOR VS Perceptron

A perceptron is a simple model of a neuron. It takes in a set of inputs, multiplies them by a set of weights and sums them up to produce an output. The output is then passed through the heaviside [3] step function to produce a binary output. The perceptron can be used to solve simple classification problems. However, it is not able to solve the XOR problem. XOR is not linearly separable, meaning that it is not possible to draw a straight line that separates the two classes. This is a problem for the perceptron, as it can only draw straight lines. One way to solve this problem is to add some polynomial features to the data. This will allow us to draw a curved line that separates the two classes. However, this is not a scalable solution. NNs can also learn xor, without the need for a polynomial feature. This is important because it is not always obvious what feature engineering is required to solve a problem. Neural networks can learn the relevant features from the data. Manually adding polynomial features is not a scalable solution for high-dimensional data, whereas neural networks can learn arbitrarily complex functions. Adding polynomial featurer is also prone to bias problems.

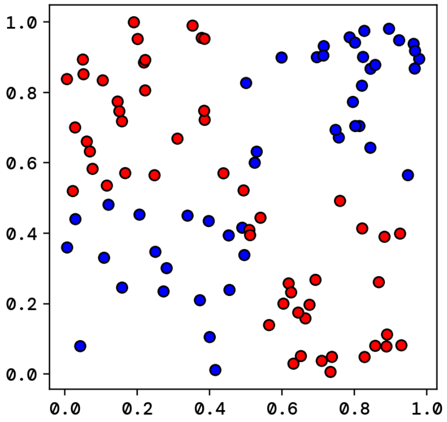


Figure 23

[3] We use sigmoid instead of the heaviside, but the outcome of the classification is the same.

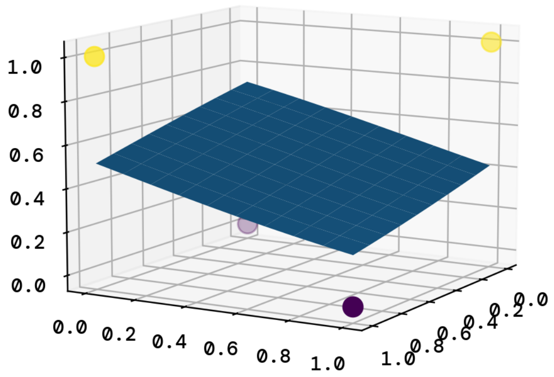


Figure 24

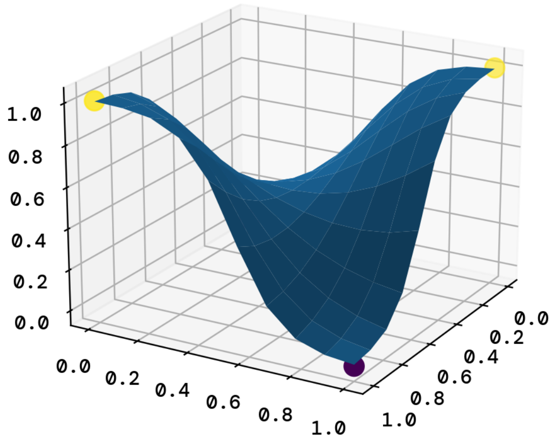


Figure 25

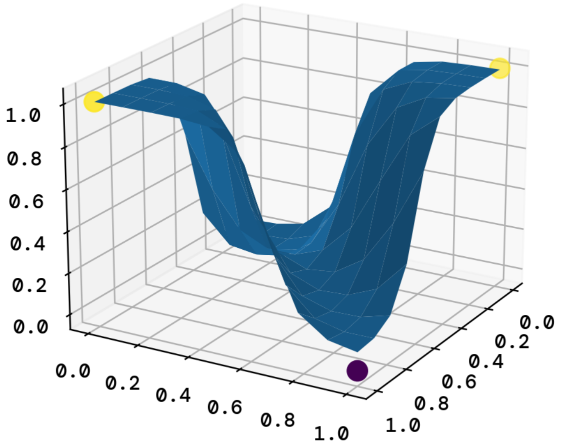


Figure 26

Combining Neurons

We want to approximate this sin function with a neural network. How might that look like? We can use a single neuron to approximate a line $aw + b$, putting it trough a sigmoid function we get some non-linearity. Every node in the first hidden layer (and consequents layers) is then a line put trough a sigmoid function. The final output of a single hidden layer network is just a linear combination of these curves. We can add more nodes to get more curves, but we are still We can chose whatever activation function we want as long as it is non-linear. Relu strugles to learn this sin function, but sigmoid works fine.

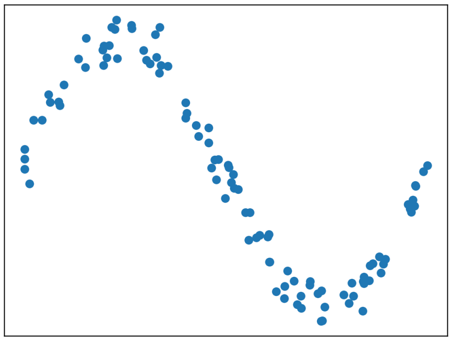


Figure 27: A sin function that we want to approximate

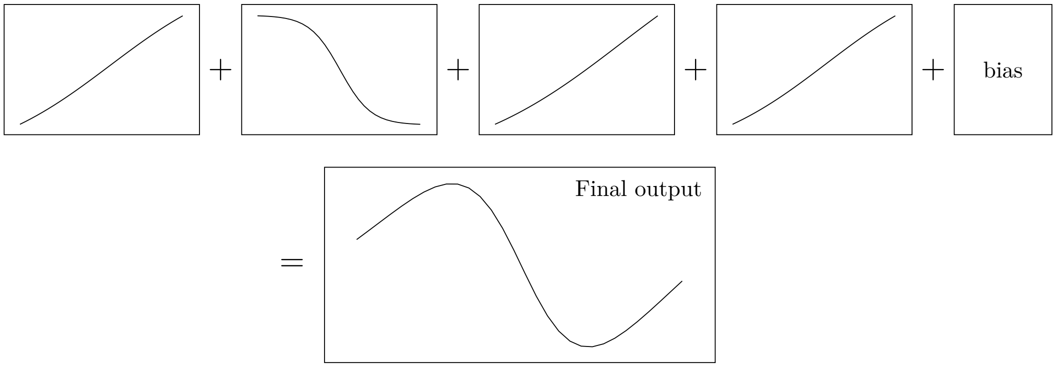


Figure 28: Four indivudual logistic regression outputs scaled and added together to produce a sin approximation

Bibliography

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1 edition, 2007.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. Book in preparation for MIT Press.

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

[4] Morten Hjorth-Jensen. Fys-stk4155/3155 applied data analysis and machine learning. `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html`, 2021.

[5] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[6] Izaak Neutelings. Tikz code for neural networks. `https://tikz.net/neural_networks/`, 2021. modified to fit the needs of this project.

[7] Michael A. Nielsen. Neural networks and deep learning, 2018.

[8] OpenAI. Chat-gpt. `https://openai.com/chatgpt`, 2023. Some of the code and formulations are developed with the help from Chat-GPT.

[9] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[10] Wessel N. van Wieringen. Lecture notes on ridge regression, 2023.

[11] Brage Wiseth, Felix Cameren Heyerdahl, and Eirik Bjørnson Jahr. MachineLearning-Projects. `https://github.com/bragewiseth/MachineLearningProjects`, November 2023.

[12] W. Wolberg, O. Mangasarian, N. Street, and W. Street. Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository, 1995. DOI: https://doi.org/10.24432/C5DW2B.