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 18, 2023

https://github.com/bragewiseth/MachineLearningProjects

Contents

https://github.com/bragewiseth/MachineLearningProjects

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

There are many real world applications and problems that require us to correctly differentiate between some classes. For example, in the field of cancer research, it is often necessary to classify a tumor as either benign or malignant. this can be done by extracting a set of features from the tumor and differentiating between the two classes based on how different classes posess slight variations across the features. However, some features may be more or less equal for both classes, and the amount of features may be large. These two factors may make it difficult for a human to classify the tumor. To tackle this, we can use *logistic regression*, that is, first regression and then clamp the output to a binary value (for the bianry case). We can do this with an activation function like the sigmoid function [1] or the heaviside function. This binary value is then the classification of the data point. However, in some cases it may be impossible for logistic regression to make a good classification. For example, if the data is not linearly separable, linear regression will not be able to draw a straight line that separates the two classes. And thus the output of the activation function will not be able to classify the data correctly. We explore this problem in more detail in Appendix B. As for now, a solution is using a *neural network* instead. Neural networks are able to approximate any continuous function, given enough neurons in the hidden layer. This makes them a powerful tool for classification problems. They can also be used for regression problems, which we will explore in Appendix A.

In project 1[12], we used an analytical expression to find the optimal parameters for linear regression. However, by using activation functions we lose our lineariy, and thus we can no longer use the same analytical expression to find the optimal parameters. Instead, we must use an iterative approach to find the optimal parameters. If our cost function is continuous and differentiable, we can use gradient descent to find the minimum of the cost function. All the models we will be using in this project are gradient descent based.

1. The sigmoid function does not output a binary value, but a value between 0 and 1. We can then set a threshold, for example 0.5, and say that if the output is above the threshold, we classify it as 1, and if it is below, we classify it as 0. We can interpret the output as the probability of the data point being 1.

The report is structured as follows: In Section 2 we will introduce the logistic regression model. In Section 3 we will introduce gradient descent and backpropagation. In Section 4 we will introduce neural networks. In Section 5 we will introduce the data we will be using. In Section 6 we will present our results and discuss them. In Section 7 we will conclude the report. In Appendix A we will explore regression with neural networks. In Appendix B we will explore the universal approximation theorem.

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.

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.

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.

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

$$\begin{tikzcd}
 & z_0 & a_0 & z_1 & a_1 & & \\
\boxed{x} \arrow[r] & \boxed{xw_0 + b_0} \arrow[r] & \boxed{\sigma(z_0)} \arrow[r] & \boxed{a_0 w_1 + b_1} \arrow[r] & \boxed{\sigma(z_1)} \arrow[r] & \boxed{y} \arrow[r] & \boxed{L(y,t)}
\end{tikzcd}$$

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

$$\boxed{\frac{\partial L}{\partial w_0}} = \boxed{\frac{\partial L}{\partial a_1}} \cdot \boxed{\frac{\partial a_1}{\partial z_1}} \cdot \boxed{\frac{\partial z_1}{a_0}} \cdot \boxed{\frac{\partial a_0}{\partial z_0}} \cdot \boxed{\frac{\partial z_0}{\partial w_0}}$$

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 output), 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.

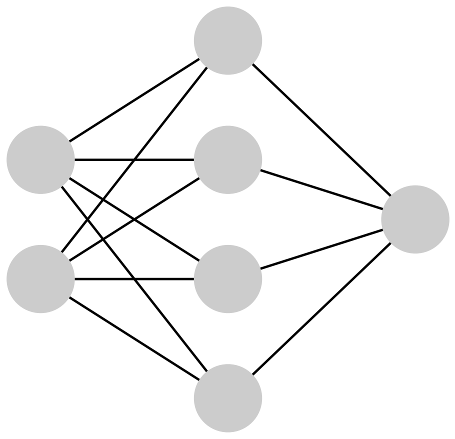
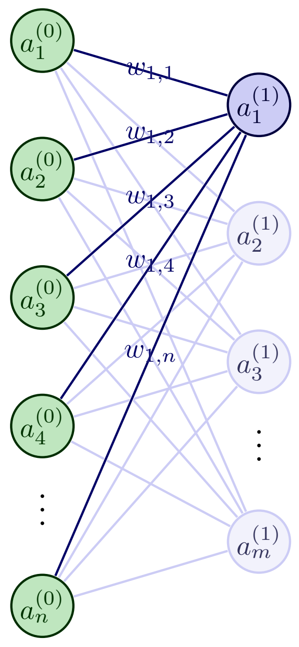


Figure 2: A model of a neural network with one hidden layer consisting of four nodes [7]

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 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



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

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

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

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

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.
[7]

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 classification we will be using the Wisconsin breast cancer dataset [cite here]. This dataset contains 569 data points with 30 features each. The 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. The features are computed for each cell nucleus: radius, texture, perimeter, area, smoothness, compactness, concavity, concave points, symmetry, fractal dimension. The mean, standard error and "worst" or largest (mean of the three largest values) of these features were computed for each image, resulting in 30 features. For example, field 3 is Mean Radius, field 13 is Radius SE, field 23 is Worst Radius.

All feature values are recoded with four significant digits. Missing attribute values: none. Class distribution: 357 benign, 212 malignant

malignant = cancer.data[cancer.target==0] benign = cancer.data[cancer.target==1]

the goal is to classify the tumors as either benign or malignant based on the features. a positive result means that the tumor is malignant, and a negative result means that the tumor is benign. in other words, we want to find the cases of cancer and classify them as positive

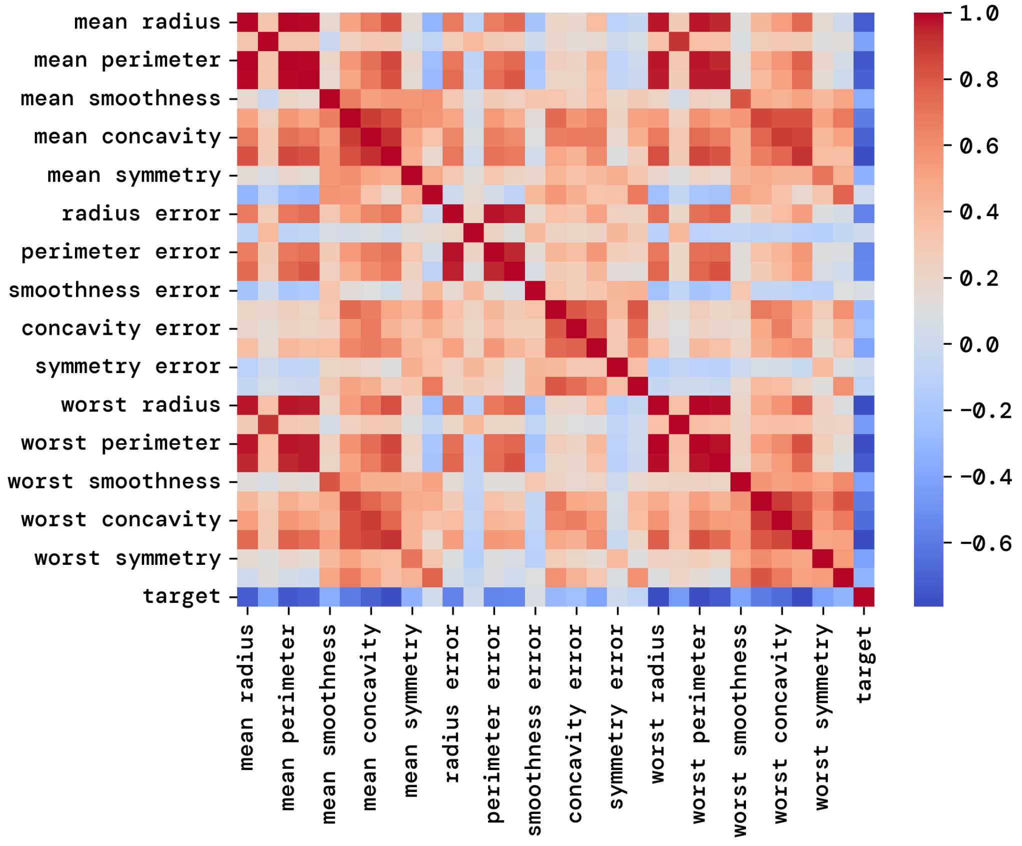


Figure 4: Feature correlation. Gives us an idea of the redundancy of the features. Only every other features is annotated but you can infer the missing lables from Figure 5

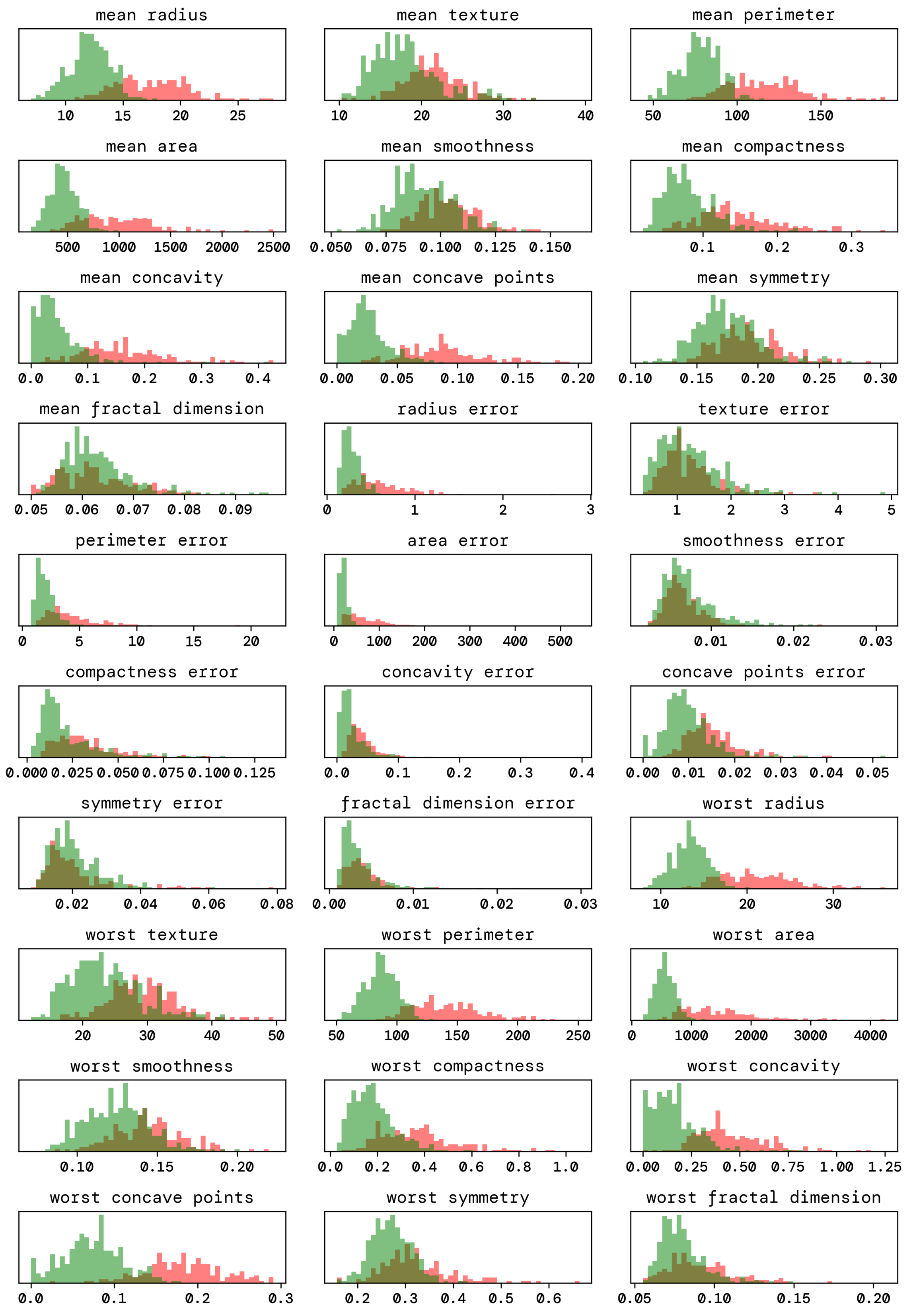


Figure 5: Feature histogram. Red (0) class is malignent, green (1) is benign

4. Results and Analysis

For classification we use the cross entropy loss function. we use automatic differentiation to calculate the gradient of the loss function with respect to the parameters. some of the hyperparameters are more dependent on each other than others. For example, the learning rate and the number of epochs the batch size and the learning rate. Optimally we can therefore not tune these hyperparameters independently. Due to our limited time and computational resources we have not been able to do a full grid search. At most we have been able to do a grid search of two dimensions. These indivudial searches are used as an indication of where the optimal hyperparameters lies.

4.1 Hyperparameters

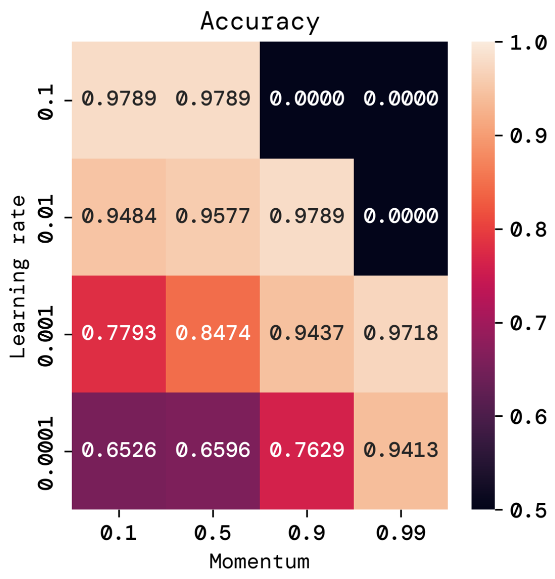


Figure 6:

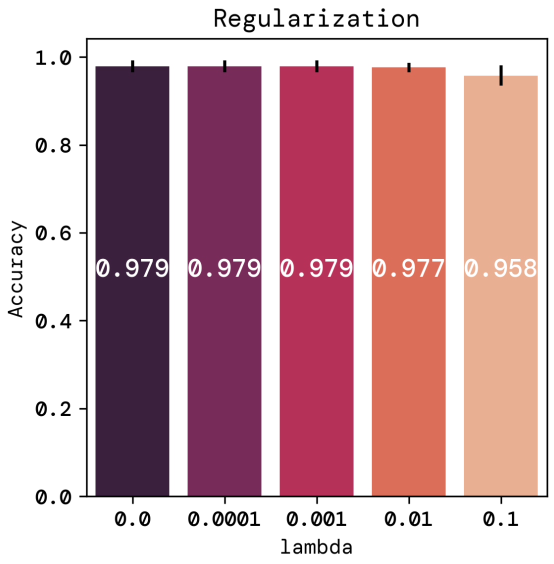


Figure 7:

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 regualarization does not seem to yield any benefits, in fact having too much regualarization decreases performance

The optimizer does not seem to have a large impact on the performance of the model. However, the Adam and seems to be slightly better than the others. The batch

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

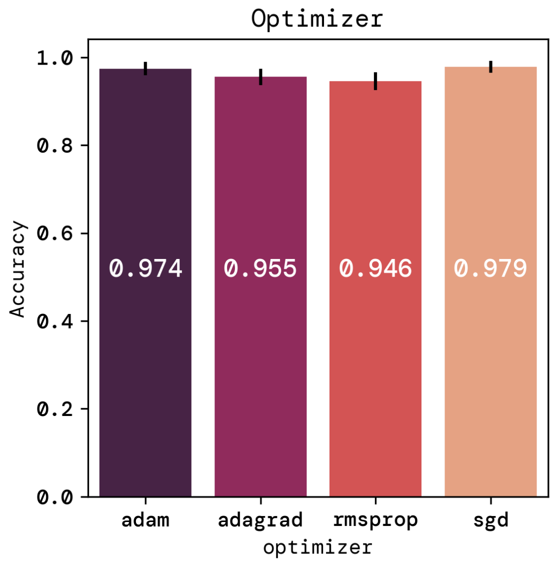


Figure 8: sgd (Standard gradient decent with momentum)

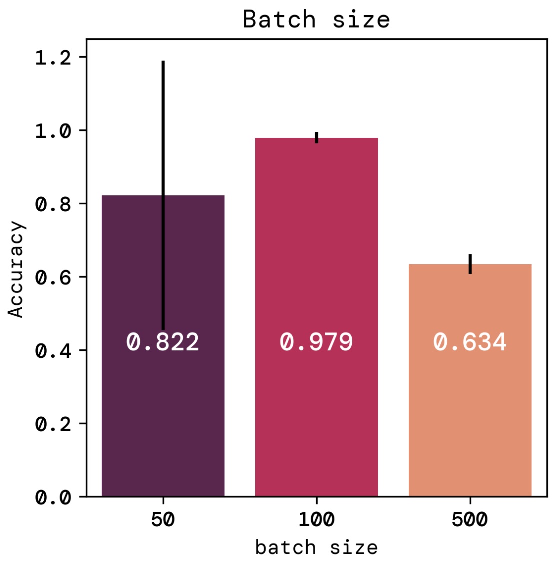


Figure 9:

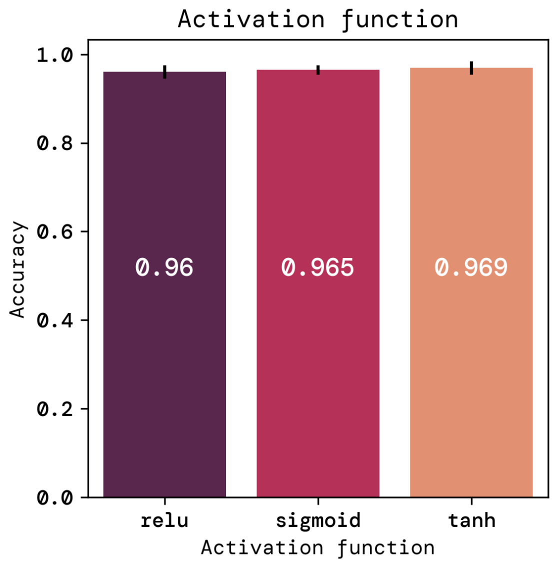
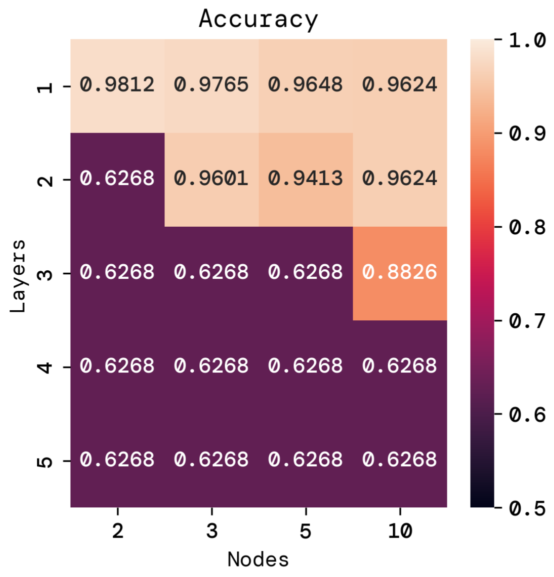


Figure 10:



Layers \ Nodes	2	3	5	10
1	0.9812	0.9765	0.9648	0.9624
2	0.6268	0.9601	0.9413	0.9624
3	0.6268	0.6268	0.6268	0.8826
4	0.6268	0.6268	0.6268	0.6268
5	0.6268	0.6268	0.6268	0.6268

Figure 11:

4.2 Final Evaluation and Comparisons

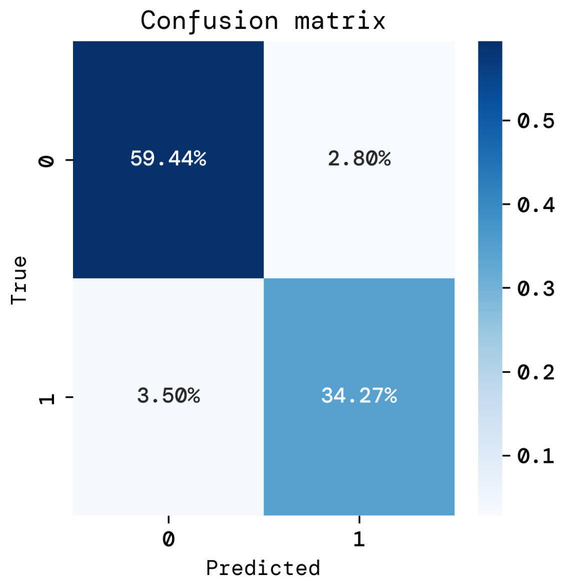


Figure 12: Confusion matrix for the best model. The model has an accuracy of 96% on the test set.

5. Conclusion

Appendix A

Appendix A. Regression with Neural Networks

In project 1[12], we found that we can fit lines or even polynimals that approximate the distribution of our data by solving an nice analytical expression for the optimal parameters β. We can in principle approximate any function with a polynomial, if we give ourselves infinite degrees of freedom. This is great but there are several limitations to this approach. First of all, we can not give ourselves infinite degrees of freedom, believe it or not. Secondly, what if we don't want to find a polynimal but rather classify our data into some classes? However the first problem still remains, we want to approximate any function, but don't want to use an infinite taylor series. We need a different approach. Instead of finding a single high degree polynimial, we can try to glue together a bunch of smaller line segments. For this we can use a *neural network*.

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

https://github.com/bragewiseth/MachineLearningProjects

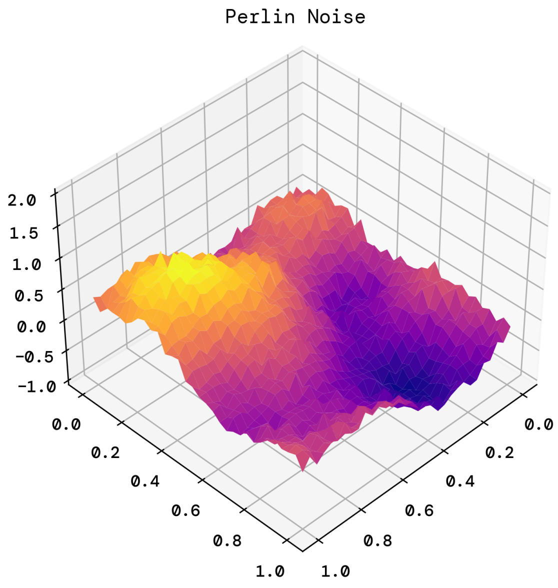
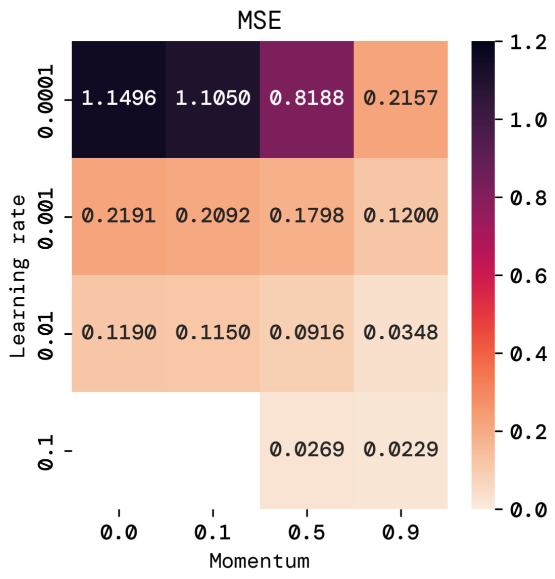


Figure 13



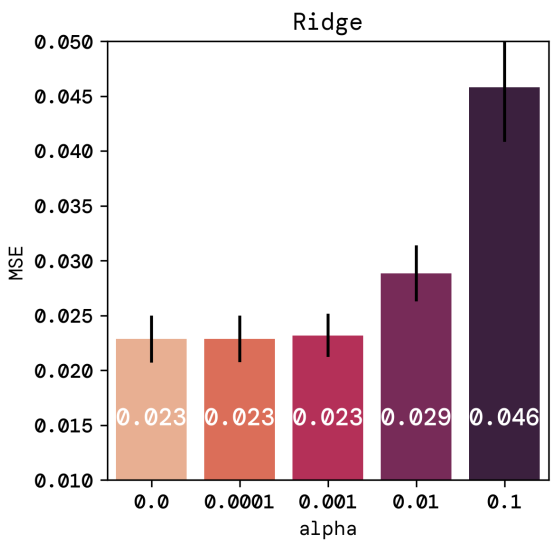


Figure 14: In this case havving momentum seems to be beneficial. We maxes out our testing range and found that 0.9 was the best value for momentum. momentum allows a higher learning rate

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

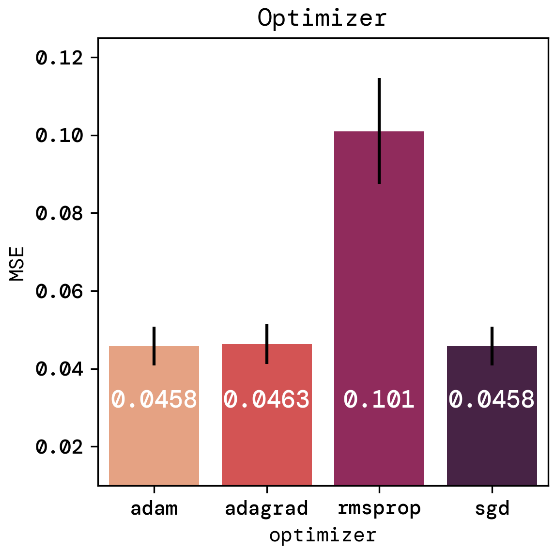


Figure 16: In this case havving momentum seems to be beneficial. We maxes out our testing range and found that 0.9 was the best value for momentum. momentum allows a higher learning rate

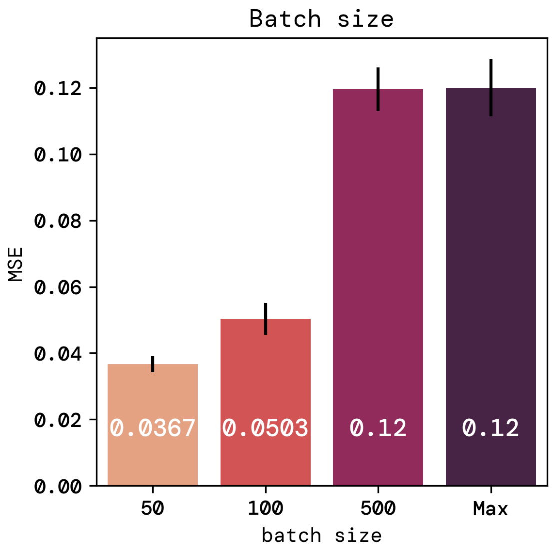


Figure 17: Introduction regualarization does not seem to yield any benefits, in fact having too much regualarization shunts performance

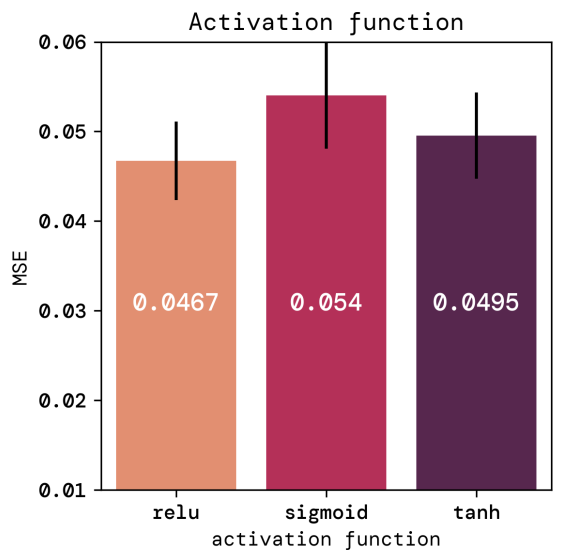


Figure 18: In this case havving momentum seems to be beneficial. We maxes out our testing range and found that 0.9 was the best value for momentum. momentum allows a higher learning rate

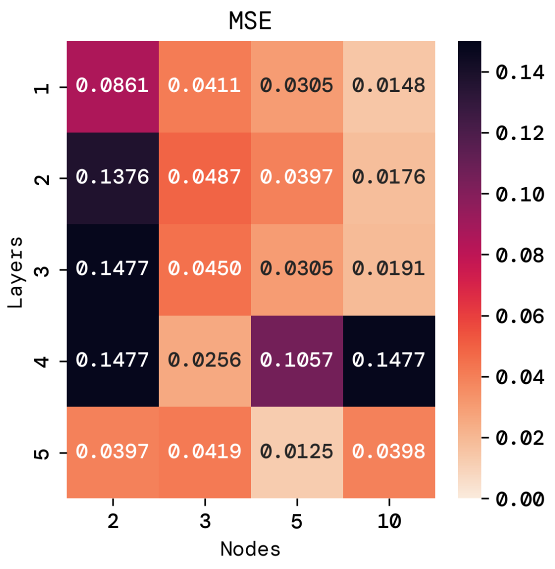


Figure 19: Introduction regualarization does not seem to yield any benefits, in fact having too much regualarization shunts performance

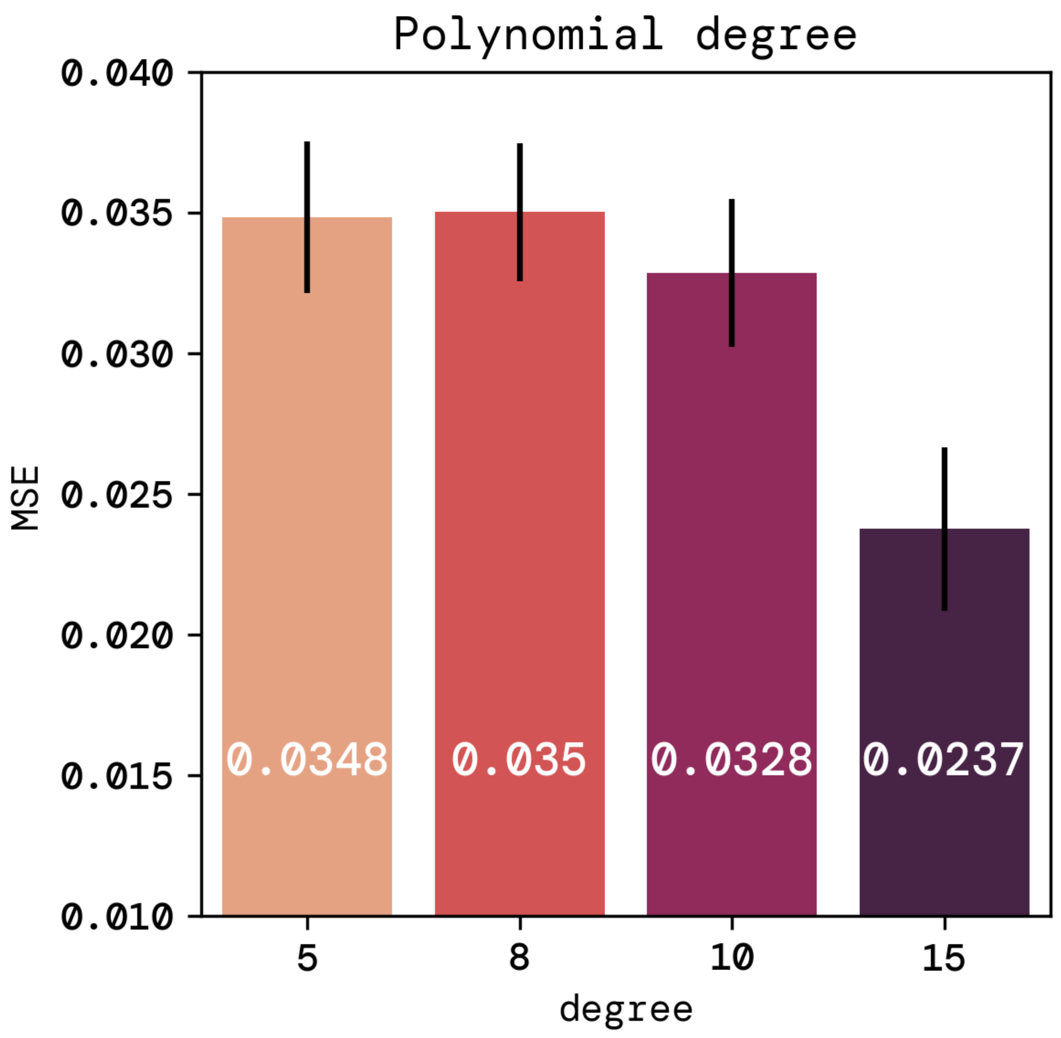

Polynomial degree
MSE
degree
0.040
0.035
0.030
0.025
0.020
0.015
0.010
5
8
10
15
0.0348
0.035
0.0328
0.0237

Figure 20

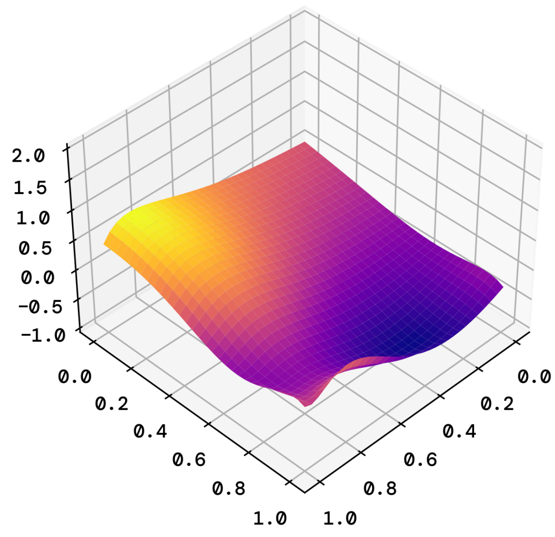


Figure 21: In this case havving momentum seems to be beneficial. We maxes out our testing range and found that 0.9 was the best value for momentum. momentum allows a higher learning rate

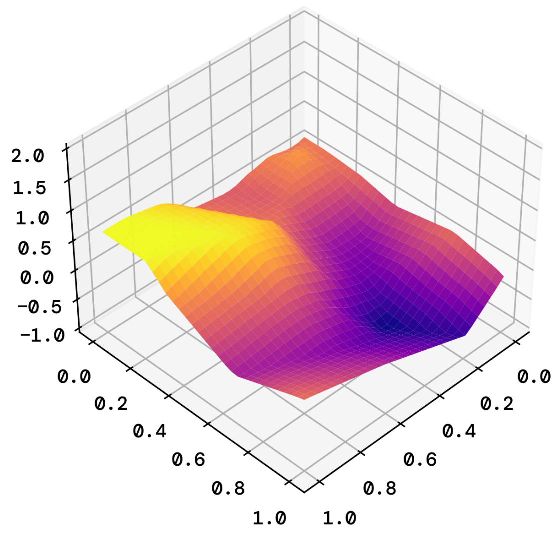


Figure 22: Introduction regualarization does not seem to yield any benefits, in fact having too much regualarization shunts performance

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 [6].

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.

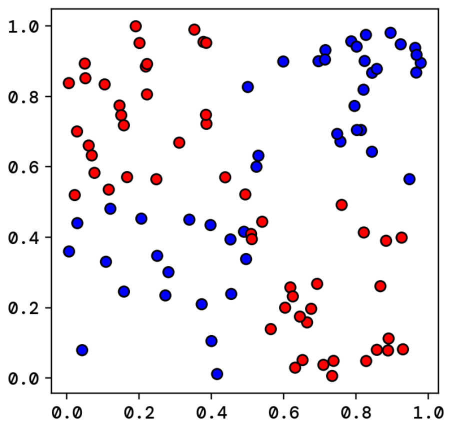


Figure 23

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

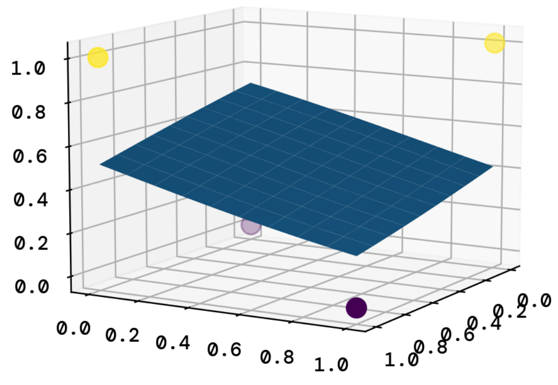


Figure 24

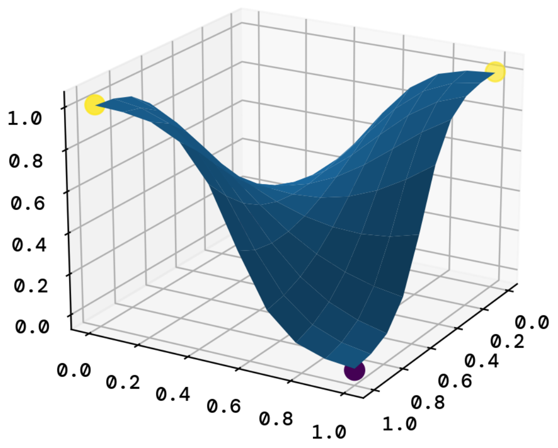


Figure 25

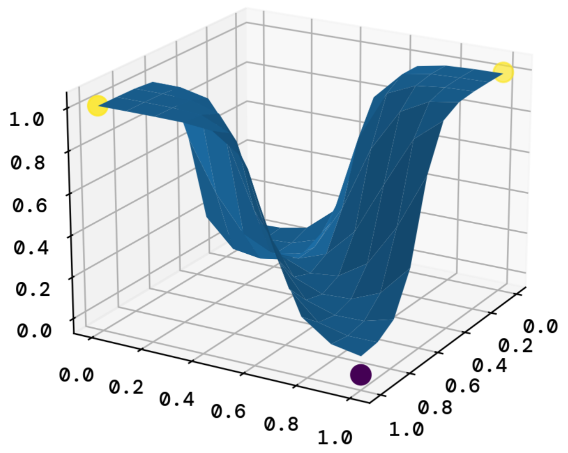


Figure 26

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.

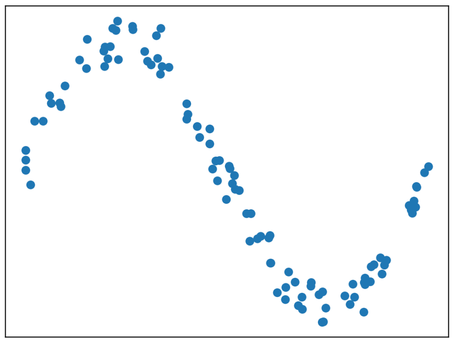


Figure 27: A sin function that we want to approximate

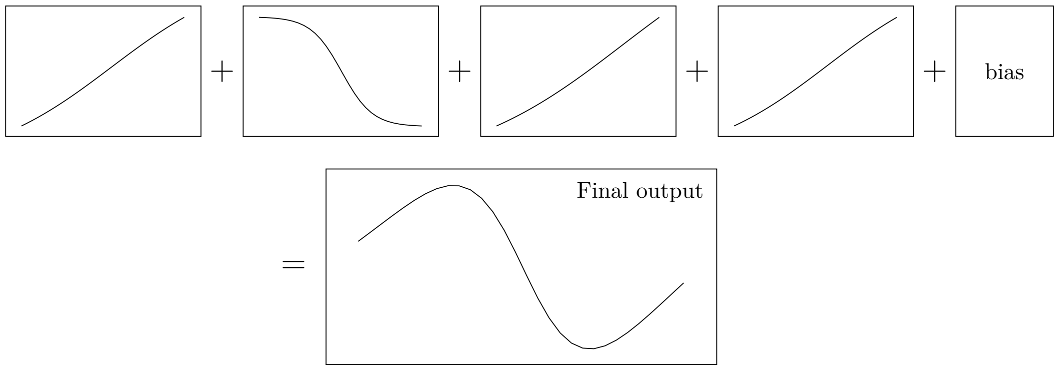


Figure 28: Four indivudual logistic regression outputs scaled and added together to produce a sin approximation

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

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

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

[4] 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.

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

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

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

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

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

[10] 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.

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

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