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 17, 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.

Gradient Decent:

In this section we will discuss the gradient decent algorithm and how it can be used to

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.

optimize the parameters of a model.
Data:
In this section we will discuss the data we will be using in this project.
Results:
In this section we will discuss the results we achieved with our models.
Conclusion:
In this section we will discuss the results we achieved with our models.

1. Gradient Descent

Gradient descent is an iterative optimization algorithm for finding the minimum of a function. The idea is to take steps in the direction of the negative gradient of the function. It is verry similar to the Newton-Raphson method for finding roots of a polynimal, but instead of using the second derivative, we use the first derivative.

optimizers

stocastic

1.1 Backpropagation and Chain Rule

Calculating derivatives is the bread and butter of machine learning. For the simpelest models, like linear regression, the derivatives are relatively easy and straight forward to calculate. We simply take the derivative of the cost function with respect to our parameters. However, for more complex models, like neural networks, the derivatives are not so easy to calculate. When the loss function is a composition of several functions, we need to invoke the chain rule.

Forward

$$\begin{tikzcd}
 & z_0 & a_0 & z_1 & a_1 & & \\
\boxed{x} \arrow[r] & \boxed{xw_0 + b_0} \arrow[r] & \boxed{\sigma(z_0)} \arrow[r] & \boxed{a_0w_1 + b_1} \arrow[r] & \boxed{\sigma(z_1)} \arrow[r] & \boxed{y} \arrow[r] & \boxed{L(y,t)}
\end{tikzcd}$$

If we want to calculate the derivative of the loss function with respect to w_0, we must go trough all the functions in the chain and calculate the derivative of each function with respect to the next function.

Backwards

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

Figure 1: chain rule

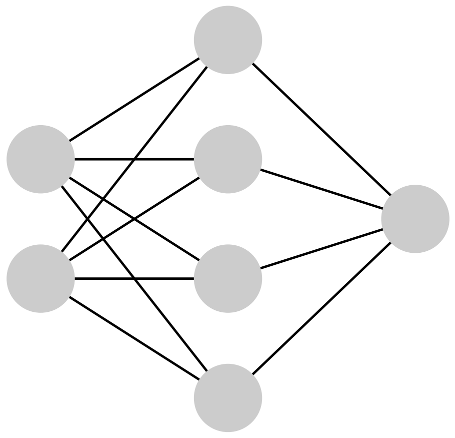


Figure 2: A model of a neural network with one hidden layer consisting of four nodes [7]

2. Neural Networks

As it turns out, the framework for neural networks is very similar to the framework for logistic regression. Neural networks can be interpeted as several logistic regression models glued together, huge benefit of this is that we can use the same code For both logistic and linear regression as well as neural networks.

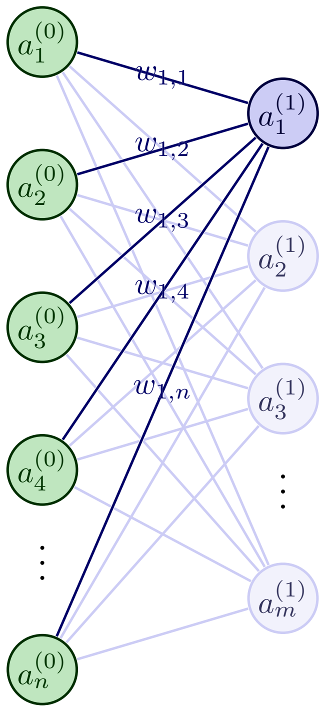
With backpropagation we have a way to calculate the derivatives of the loss function with respect to the parameters. This allows us to train our neural network. with one layer the network becomes a linear regression model. $y = x_0w_0 + x_1w_1 + x_2w_2 + ... + x_nw_n + b_0$ with a activation function at the end we can make it into a logistig regression model. This makes it very convenient to use the same code for both linear and logistic regression as well as for larger neural networks with hidden layers, by simply swapping out the different parts.

activation functions

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]



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

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

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

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

Figure 3

[7]

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

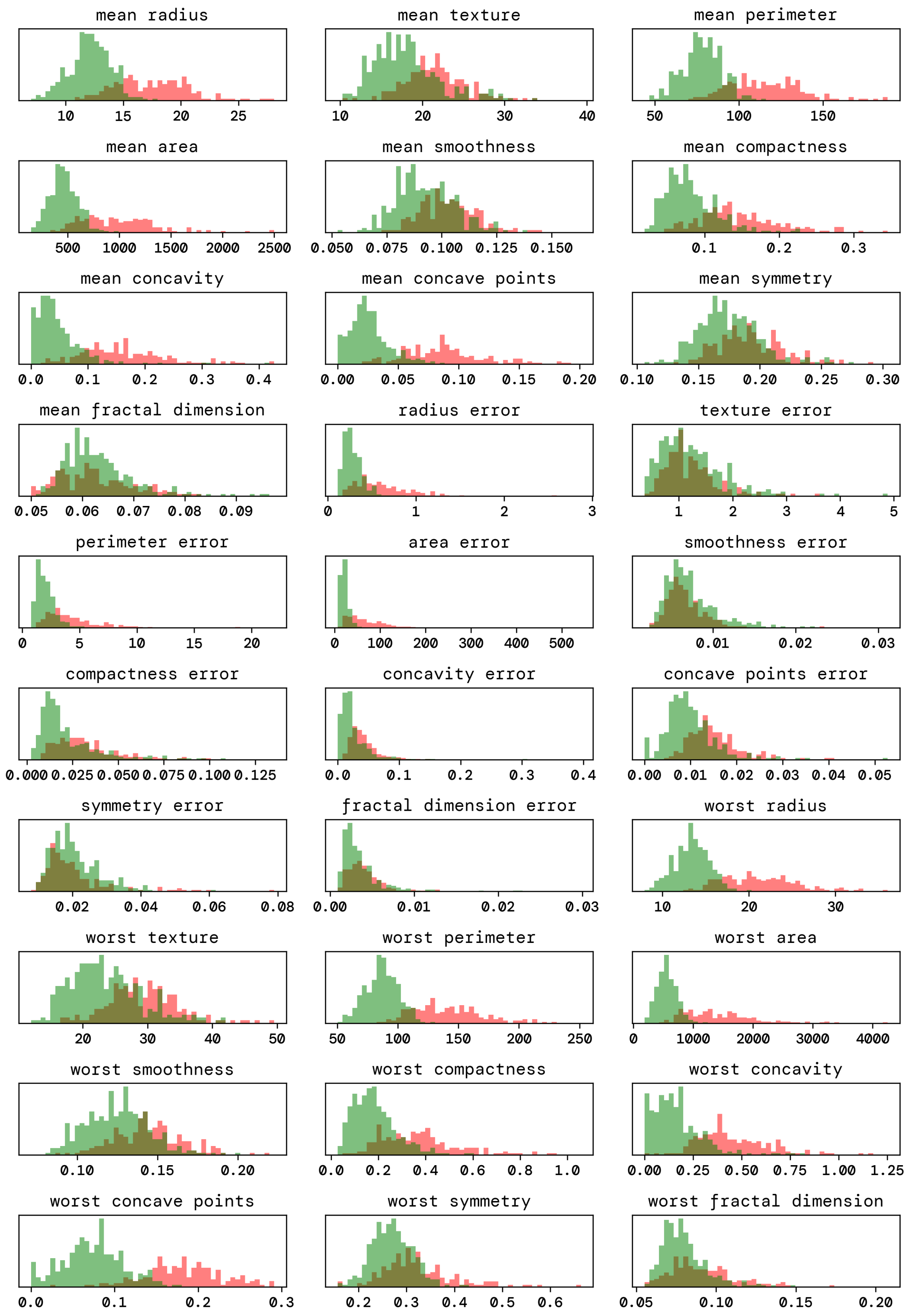


Figure 4: Feature histogram. Red (0) class is malignent, green (1) is benign

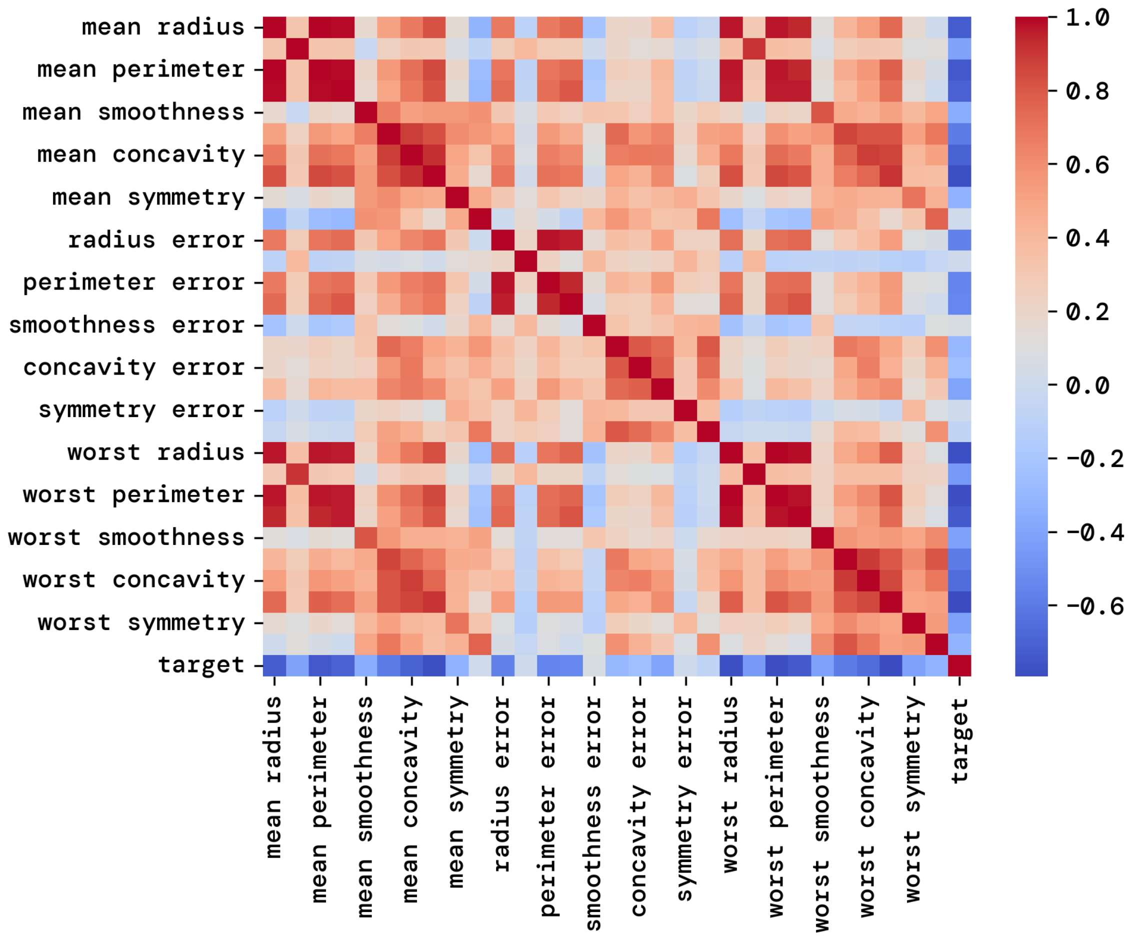


Figure 5: Feature correlation. Gives us an idea of the redundancy of the features. Only every other features is annotated but you can infer the missing lables from Figure 4

4. Results and Analysis

For classification we use the cross entropy loss function. We tune the hyperparameters in a grid searches with a maximum of two dimensions. As we find the optimal hyperparameters, we use the same optimal hyperparameters for the next grid search.

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. We can therefore not tune these hyperparameters independently. We must first find the optimal

due to our limited time and computational resources we have not been able to do a full grid search for all hyperparameters. we train each hyperparameter separatly, in some cases in a grid of two dimensions. we then use this as a utgangspunkt for fine tuning.

4.1 Hyperparameters

epochs found to be 200

when training for batch size we had to decrease the optimal learning rate due to unstable results for smaller batches

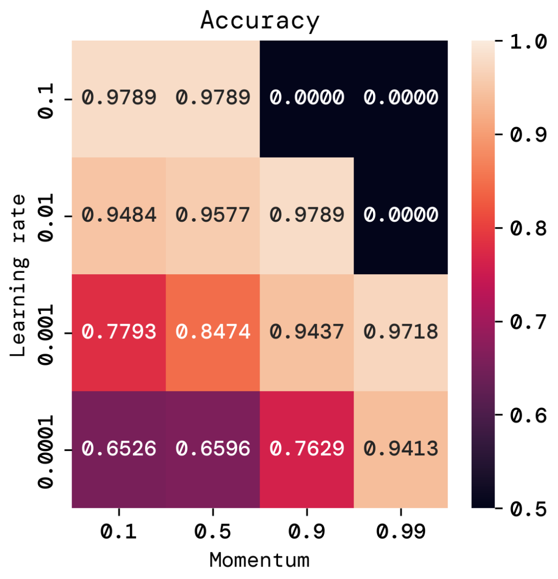


Figure 6: In this case havving momentum seems to be beneficial. We maxes out our testing range and found that 0.9 was the best value for momentum. momentum allows a higher learning rate

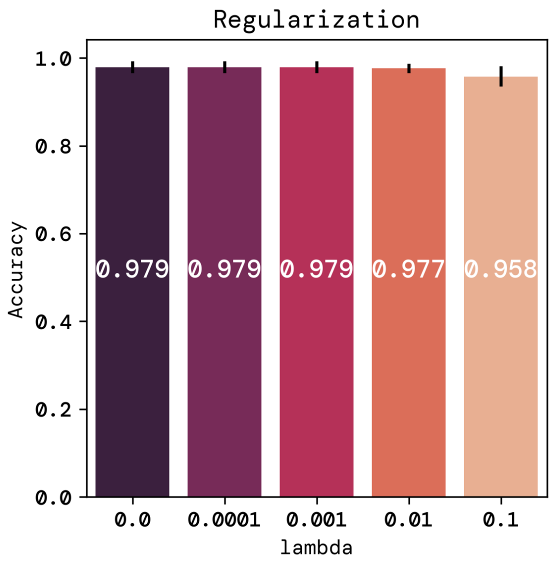


Figure 7: Introduction regualarization does not seem to yield any benefits, in fact having too much regualarization shunts performance

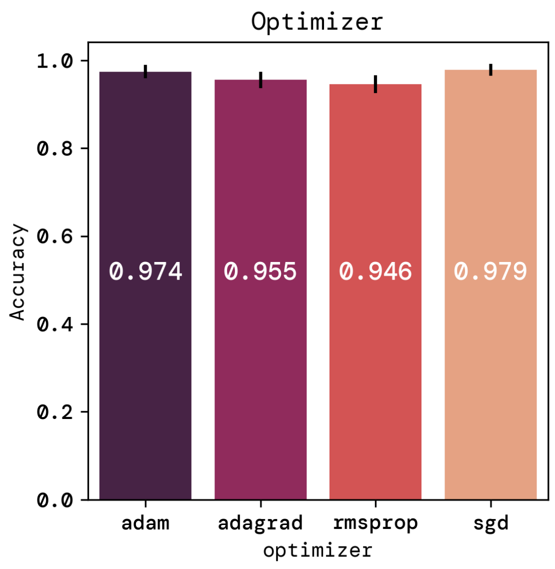


Figure 8: In this case havving momentum seems to be beneficial. We maxes out our testing range and found that 0.9 was the best value for momentum. momentum allows a higher learning rate

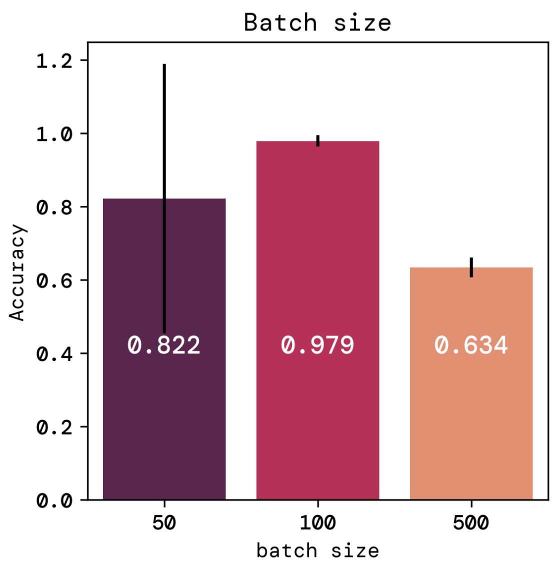
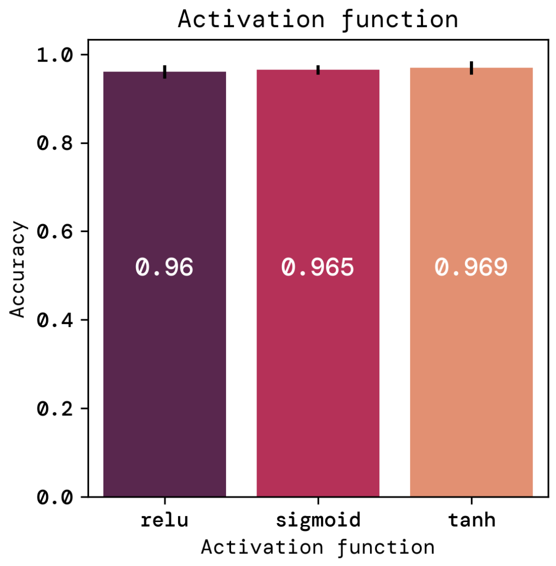


Figure 9: Introduction regualarization does not seem to yield any benefits, in fact having too much regualarization shunts performance

4.2 Final Evaluation and Comparisons

5. Conclusion

Lorem ipsum dolor sit amet, officia excepteur ex fugiat reprehenderit enim labore culpa sint ad nisi Lorem pariatur mollit ex esse exercitation amet. Nisi anim cupidatat excepteur officia. Reprehenderit nostrud nostrud ipsum Lorem est aliquip amet voluptate voluptate dolor minim nulla est proident. Nostrud officia pariatur ut officia. Sit irure elit esse ea nulla sunt ex occaecat reprehenderit commodo officia dolor Lorem duis laboris cupidatat officia voluptate. Culpa proident adipisicing id nulla nisi laboris ex in Lorem sunt duis officia eiusmod. Aliqua reprehenderit commodo ex non excepteur duis sunt velit enim. Voluptate laboris sint cupidatat ullamco ut ea consectetur et est culpa et culpa duis.



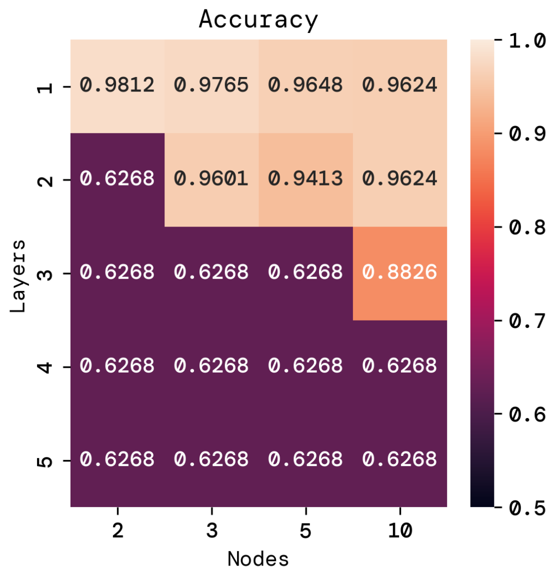


Figure 10: In this case havving momentum seems to be beneficial. We maxes out our testing range and found that 0.9 was the best value for momentum. momentum allows a higher learning rate

Figure 11: Introduction regualarization does not seem to yield any benefits, in fact having too much regualarization shunts performance

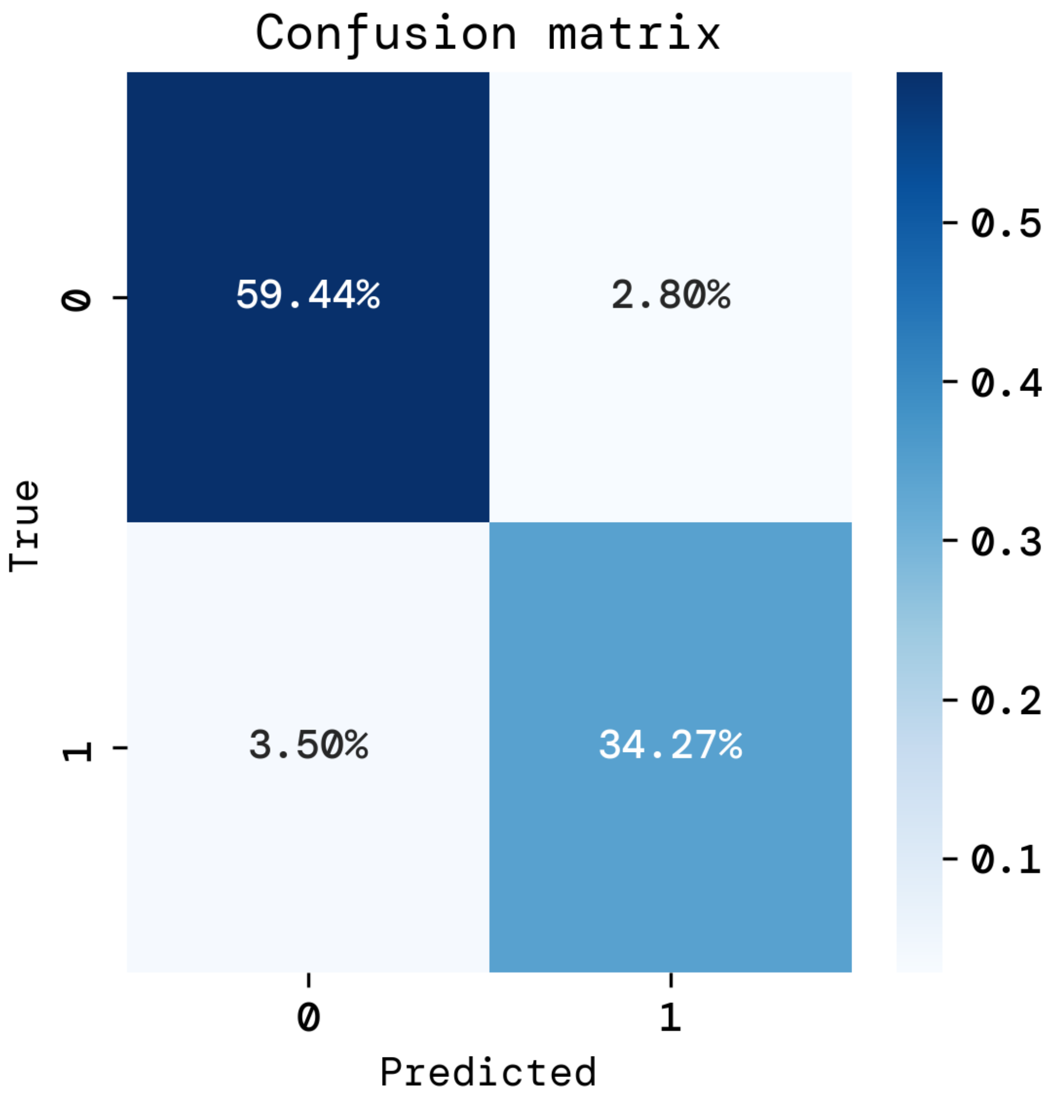

Confusion matrix
59.44%
2.80%
3.50%
34.27%
True
0
1
Predicted
0
1
0.5
0.4
0.3
0.2
0.1

Figure 12

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

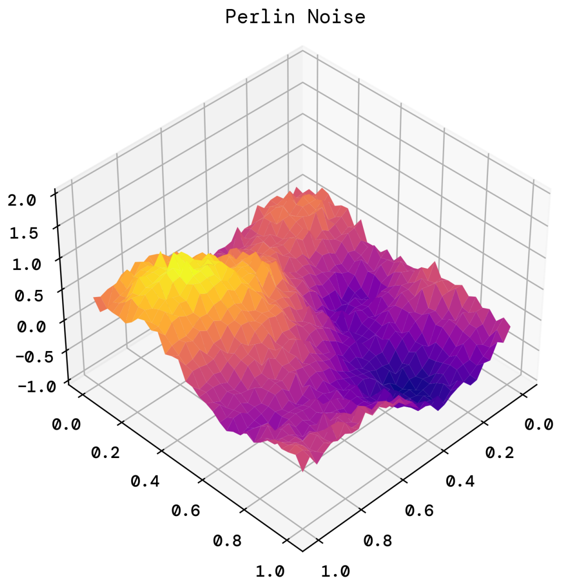
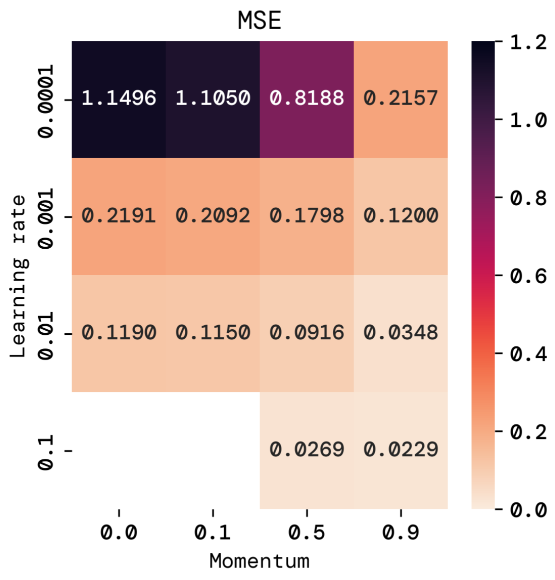


Figure 13



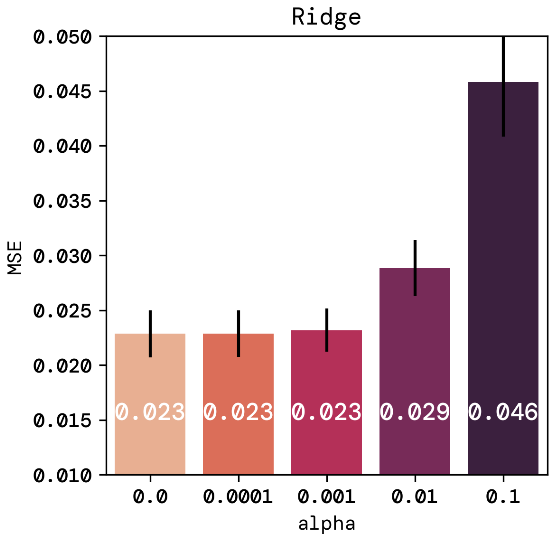


Figure 14: In this case havving momentum seems to be beneficial. We maxes out our testing range and found that 0.9 was the best value for momentum. momentum allows a higher learning rate

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

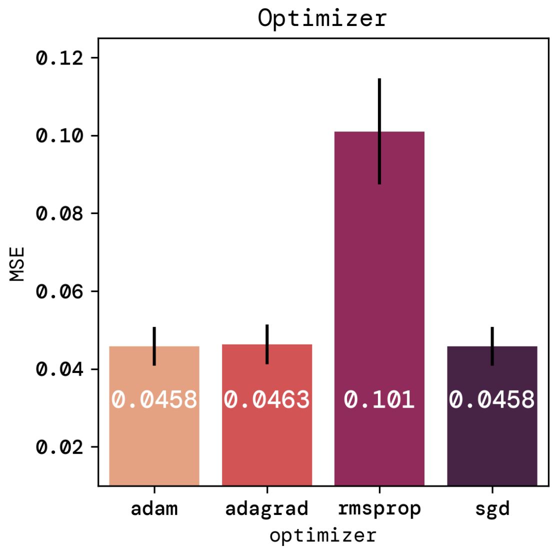


Figure 16: In this case havving momentum seems to be beneficial. We maxes out our testing range and found that 0.9 was the best value for momentum. momentum allows a higher learning rate

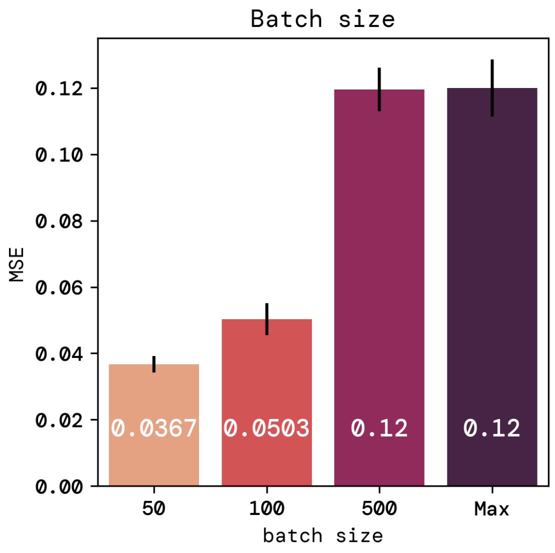


Figure 17: Introduction regualarization does not seem to yield any benefits, in fact having too much regualarization shunts performance

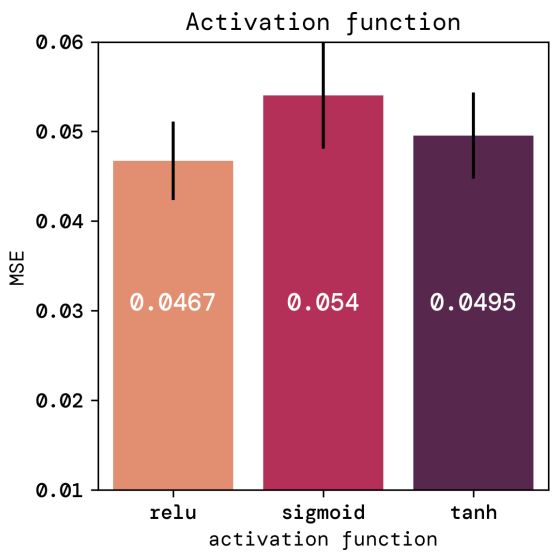


Figure 18: In this case havving momentum seems to be beneficial. We maxes out our testing range and found that 0.9 was the best value for momentum. momentum allows a higher learning rate

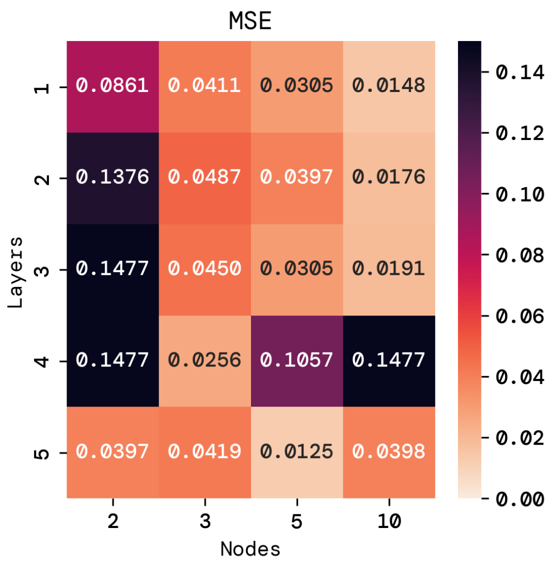


Figure 19: Introduction regualarization does not seem to yield any benefits, in fact having too much regualarization shunts performance

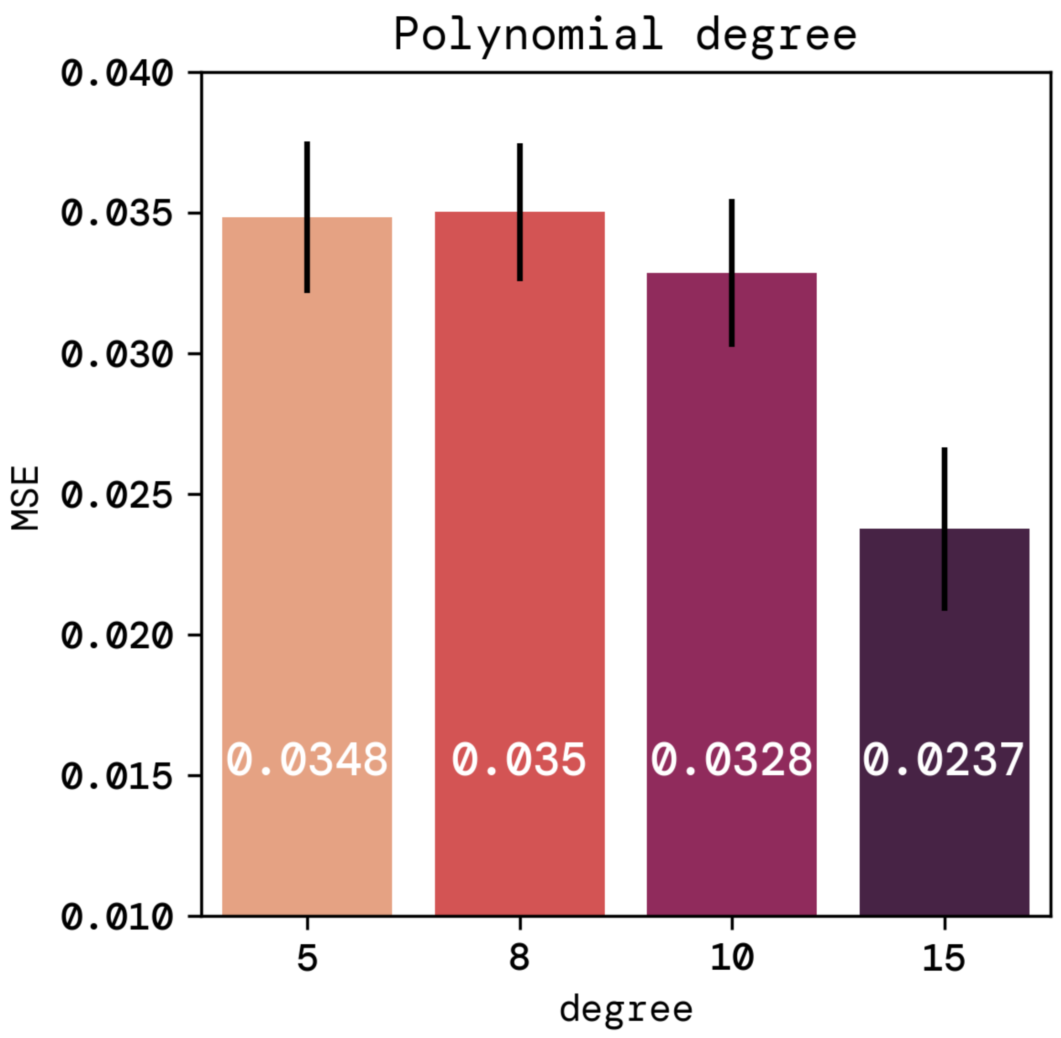

Polynomial degree
MSE
degree
0.040
0.035
0.030
0.025
0.020
0.015
0.010
5
8
10
15
0.0348
0.035
0.0328
0.0237

Figure 20

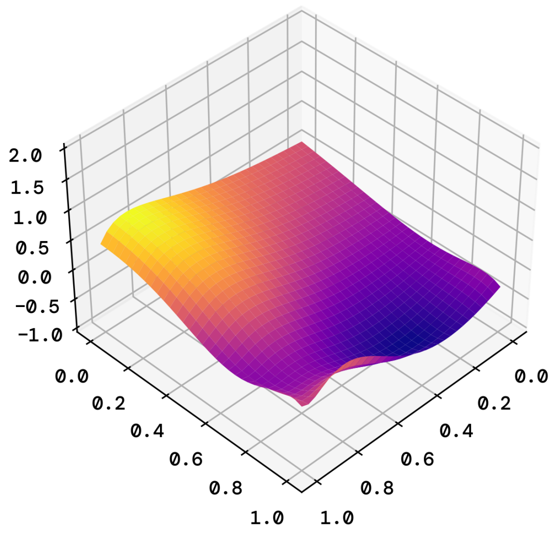


Figure 21: In this case havving momentum seems to be beneficial. We maxes out our testing range and found that 0.9 was the best value for momentum. momentum allows a higher learning rate

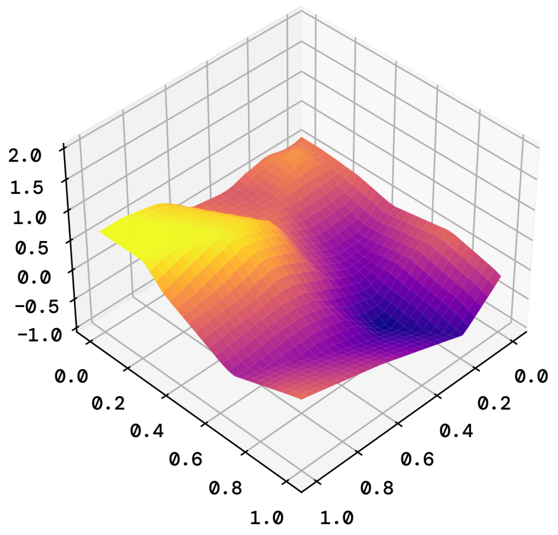


Figure 22: Introduction regualarization does not seem to yield any benefits, in fact having too much regualarization shunts performance

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 [2] 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.

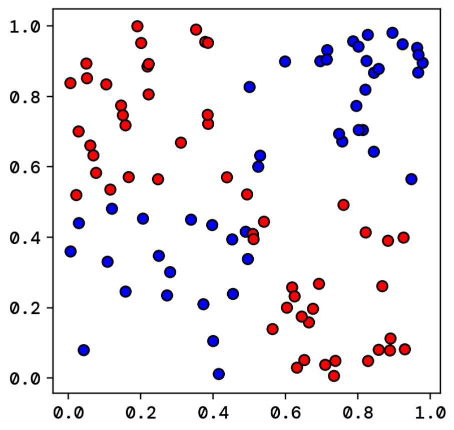


Figure 23

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

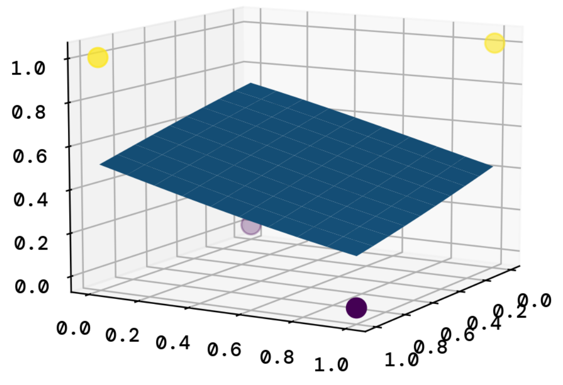


Figure 24

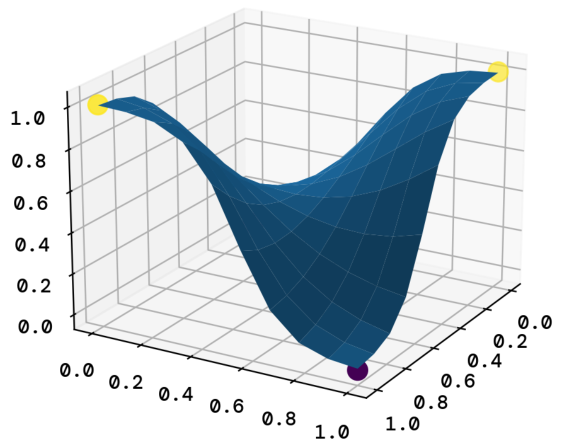


Figure 25

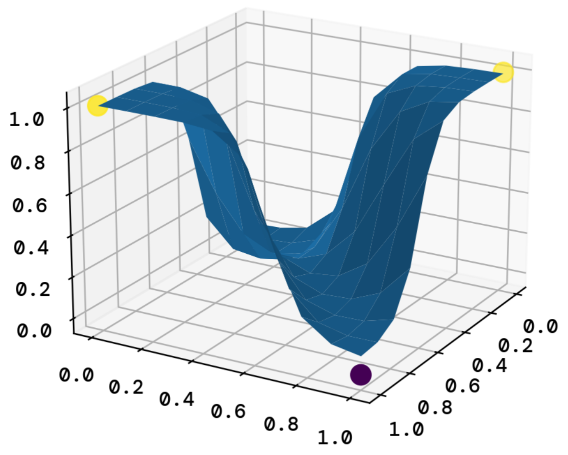


Figure 26

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.

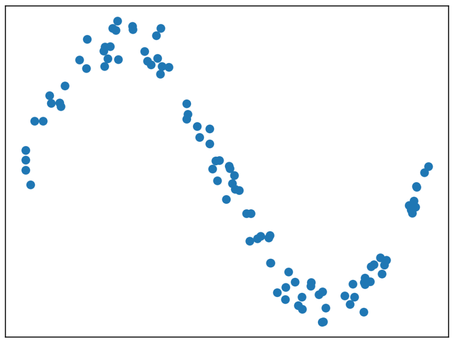


Figure 27: A sin function that we want to approximate

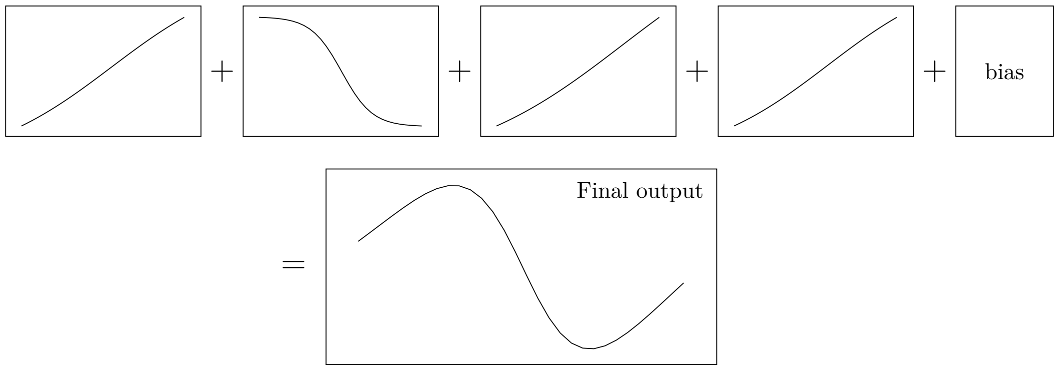


Figure 28: Four indivudual logistic regression outputs scaled and added together to produce a sin approximation

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.