FYS-STK4155 Project on Machine Learning
Part 2: Classification and Regression, from linear and logistic-regression to neural networks

University of Oslo, Norway

Svein-Magnus Lommerud and Oliwia Malinowska

Fall 2022

Abstract

In this report we continue to study important machine learning methods, just like in Project 1. For this specific project we will utilize different methods to solve regression and classification problems. First we will test different types of gradient decent methods such as stochastic gradient decent and momentum gradient decent on our regression problem. Later on, we will implement a Feed Forward Neural Network algorithm and compare it to the earlier methods used. In order to determine how our neural network worked we played around with the number of hidden nodes used. We noticed that by increasing the number of nodes to five in each hidden layer made our total MSE score more consistent and lower, but by adding to many nodes we lost the ability to accurately predict new values. Regular OLS preformed better than the Stochastic Gradient Decent method and Neural Network, although the latter two were more consistent.

Introduction

Our world consists of complicated systems connected with each other. It could be a weather system connected with migration of animals, or nervous system connected with the rest of the body. These systems are extremely nuanced in their structure, elements they consist of, and processes that make them behave as they do. To make a model of such a system we need to take into account all those ingredients and find out how they influence each other. This is impossible without the power of a computer and statistical methods that can help us grasp the huge amount of data.

In this part of the project, we will continue to study regression and classification, two very important elements in machine learning. In the previous part, we've studied linear regression methods, including Ordinary Least Square, Ridge and Lasso methods. Now, we will dive in to an alternative approach, called Neural Network, that compared to OLS, Ridge and Lasso, creates a more complex model consisting of multiple layers of nodes simulating neurons in the brain.

In this paper we will perform classification and regression with help of feed-forward neural network. To get there, we will base our code on linear regression algorithms studied in **project1**, like Ordinary Least Square, Ridge and Lasso regression methods.

We will begin our project by introducing Stochastic Gradient Decent, which is important for setting up our Logistic Regression on Feed-Forward Neural Network code. Then we will write our own code for the Neural Network using forward propagation algorithm and the Sigmoid activation function. We will compare efficiency of our Neural Network model with Stochastic Gradient Descent and Ordinary Least Square models for different polynomial degrees. Final step will be to explore how number of nodes in the hidden layers influences the learning capacity of our Neural Network model.

Theory

Gradient Descent

Gradient Descent is a popular optimization technique in Machine Learning and Deep Learning, and it can be used with most, if not all, of the learning algorithms. A gradient is the slope of a function. It measures the degree of change of a variable in response to the changes of another variable. Mathematically, Gradient Descent is a convex function whose output is the partial derivative of a set of parameters of its inputs. The greater the gradient, the steeper the slope. We have the iterative step in the gradient decent that is given by the equation.

$$\beta_{k+1} = \beta_k - \eta \nabla_\beta C(\beta_k), k = 0, 1, 2.... \tag{1}$$

Where β is the parameters of the function, ∇ is the learning rate of the algorithm and $\beta C(\beta_k)$ is the gradient of the cost function. We will describe what a cost function is later on. Following equation 1 each gradient step for the cost function is given by:

$$\nabla_\beta C(\beta_k) = \sum_{i=1}^{n} \nabla_\beta C_i(x_i \beta_k) \tag{2}$$

Starting from an initial value, Gradient Descent is run iterative to find the optimal values of the parameters to find the minimum possible value of the given cost function.

Stochastic Gradient Descent

Stochastic gradient decent (SGD) is one of three types of gradient decent. We already know that stochastic means that something has to do with random probability. Compared to regular gradient decent where we use the entire data set for each iteration we take a random sample which is called a batch instead. It is obvious this becomes a much more viable method when we are dealing with large amounts of data. Due to the random selection of samples SGD contains a lot more noise in its path towards the minimum, but this does not mater as the purpose is only to locate the minimum. For Stochastic Gradient Descent the gradient step for the cost function is given by:

$$\beta_{j+1} = \beta_j - \eta_j \sum_{i=1}^{n} \nabla_\beta C_i(x_i, \beta_k) \tag{3}$$

Momentum Gradient Descent

Its also common to use another version of Stochastic gradient decent called momentum Gradient Decent. By using a momentum term to remember which direction we are moving we are normally able to calculate the gradient faster than standard gradient decent and stochastic. The momentum gradient decent step is given by:

$$v_j =_{j-1} + \eta_{j-1} \nabla_\beta C(\beta_{j-1}) \tag{4}$$

$$\beta_{j+1} = \beta_j - v_j \tag{5}$$

Deep Forward Neural Networks

Deep Forward Neural Networks are crucial in deep machine learning models. The neural network is built up by layers consisting of various number of nodes. Nodes are meant to behave like neurons in the human brain by taking the weighted sum of its input and then apply it to something called an activation function in order to get an output. We will explain the activation function in greater detail at a later stage.

So how is a Deep Forward Neural Network built? Lets take a look of a simple network. For example, we can have a chain consisting of functions f_1, f_2 and f_3 such that the final chain is given by $f(\mathbf{x}) = f_3(f_2(f_1(\mathbf{x})))$. Here, the most inner function, f_1 is the input (first) layer of the network, f_2 the second and f_3 the third, and since it is the last one, it's called the output layer. Number of layers (functions) tells us how deep the network is.

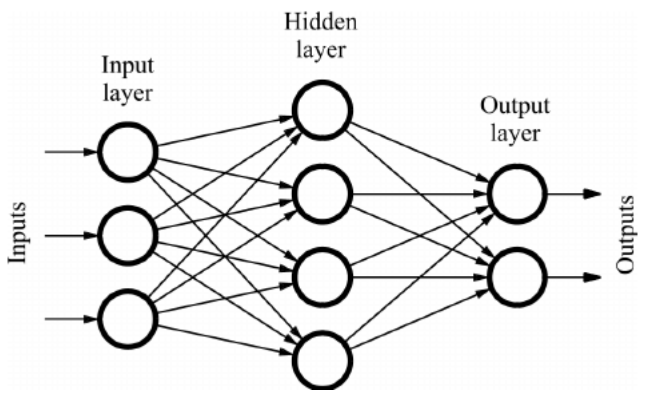

Figure 1: Simple Deep Forward Neural Network

The layers between the input and output layer are called hidden layers since the training data doesn't give us the desired output values for each of them.

As we can see on figure 1 the information only flows forward, from the input layer, through the hidden layers and to the output layer. Hence the name forward network.

Activation functions

As mentioned earlier nodes uses a activation function in order to determine its output. An activation function is a mathematical function meant to emulate neurons in the brain. The purpose of this function is to determine if the input that is passed through a node is important for the prediction model or if it should be disregarded. We have multiple functions that could be used as a activation function, but for this project we will be looking at three different functions. These three are Sigmoid function, The Rectified Linear Unit(RELU) function and the Leaky RELU.

The sigmoid function is known for its characteristic S shaped curve. The function is given by $f(x) = \frac{1}{1+e^{-x}}$ and allows us to preform a relative scaling of the data since the output of the function for all real numbers is between 0-1. The Sigmoid function has traditionally been used in Neural networks, but its not usefull in all cases. This is where the RELU function comes into play.

The RELU function returns output that is continuous compared to the binary output of the Sigmoid function. Its given by the function $f(z) = max(0, z)$. Meaning that if the input is larger than 0 it returns the input value. For all inputs lesser or equal to 0 the function will return 0. This means we have a linear function for all inputs greater than 0. This will come in handy later on.

The Leaky RELU function is the same function as the RELU, but it has been modified to allow returning some negative numbers, if the input is negative, and not zero as in the case of the normal RELU.

Backpropagation and cost function

Neural networks utilizes a set of biases and weights in order to get god results. So in order to find good values for our weights and biasas we use something called back propagation. Backpropagation calculates the gradient of the cost (error) function compared to the weights of the neural network. The algorithms acts backwards through the network, from the output layer of weights to the input layer of weights, reusing parts of the computation for one layer for the computation at next layer.

Cost function is an optimization tool. It says how accurate our model is to connect the predicted and actual values. Our goal is to create a cost function for every method and try to minimize it so that our model gets as accurate as possible.

Logistic Regression

Logistic Regression is a machine learning algorithm used for classification problems. Normally it would classify the data into two or more classes. By estimating the probability that a a sample belong to a specific class.

Methods

Gradient Descent

Starting from an initial value, Gradient Descent is run iterative to find the optimal values of the parameters to find the minimum possible value of the given cost function.

A mathematical approach to minimize the function would be to look at the equation:
$x_1 = x_o - \alpha \nabla f(x_o)$

where x_o is the starting point and α is a positive distance in the direction of the negative gradient.

We can rewrite this into a more general equation
$x_{n+1} = x_n - \alpha \nabla f(x_n)$

Starting from an initial guess x_o we keep improving little by little until we find a local minimum. This process may take thousands of iterations.

It's important to note that gradient decent has its limitations. It will only find the local minimum, meaning as soon as the algorithm finds some point that's at a local minimum we could risk it will never escape as long as it only finds a point where the direction points up. There are some other issues with the step size α where we will either overshoot the minimum with a too large step size or with a small step size we could risk having to run a large number of iterations if the function has a large number of variables. The last condition for use of gradient decent is that it needs a function that is differentiable anywhere so our gradient is always defined.

Stochastic Gradient Descent

There is a way to boost the speed of learning for SGD. By creating an update rule by modifying the SGD method where we introduce a velocity component v and a μ representing friction. The component v is the parameter we are trying to optimize and the component μ is used to control the velocity preventing it from overshooting.

By allowing the search to build inertia we are able to overcome a lot of the noise generated by the SGD method. This, in turn, will reduce the number of iterations needed to minimize the function making it even more effective than regular SGD. To make the method even more effective we can add history to the parameter. This change is based on the metaphor of momentum from physics where acceleration in a direction can be accumulated from past updates. Additionally, we add a hyperparameter that controls the amount of history (momentum) to include in the update equation, i.e. the step to a new point in the search space. The value for the hyperparameter is defined in the range 0.0 to 1.0 and often has a value close to 1.0, such as 0.8, 0.9, or 0.99. A momentum of 0.0 is the same as gradient descent without momentum.

Deep Forward Neural Networks

Building our neural network

Using output from the Franke's function and random x and y inputs, we build our neural network in the following way. We define the input layer, two hidden layers and one output layer, that will consist of $\hat{y}$, which is an estimate of the y (the output from the Franke's function). Number of nodes in each layer depend on the polynomial degree in each iteration. Those are our hyper parameters defining the structure of our network. What happens between one input node and one hidden node is that the value from the input node is multiplied by a certain weight and transferred to the hidden node. The weights are the parameters our model will learn on during the training, and the starting values for the weights are random numbers.
The value of one hidden node consists of the sum of values from the input nodes

connected to it, multiplied by the corresponding weights. The hidden node (and later the output node) applies then an activation function, for example a sigmoid function that allows us to model a non-linear system of parameters.

Forward propagation

Our input data x and y are stored in an X matrix. Each element of the matrix will be multiplied by the corresponding weight and then added together with the other values stored in each hidden node. The resulting matrix consisting of sums of weighted input values is activity of the first hidden layer. Then, we can apply the sigmoid activation function to each element of the activity matrix. Proceeding to the output layer, we need to multiply our results from each node of the hidden layer by new corresponding weights. By applying the activation function again at this step we end up with the final result, $\hat{y}$.

Gradient descent

Using a cost function, we determine how good our model is based on the test data. We minimize the cost function (train our network) by manipulating the weights.

To improve the calculation time for finding the optimal weights giving the lowest cost function, we choose one reference point and check the rate of change of the cost according to the changing weights to find in which direction, compared to the reference point, is downhill. Continuing in the correct direction, adding steps until the cost stops decreasing, is the gradient descent method used to decrease the computation time.

The complication in case of non-convex functions (functions with local minima) is due to the gradient descent method being designed to stop at a minimum, which means that the search for the lowest cost can be stopped before it reaches the actual lowest point. To avoid this problem, we choose a cost function that is the sum of squared errors (with natural convex behaviour). Another possible solution to the problem is using the stochastic gradient descent, mentioned previously, which separates the data so that we search for a minimum in the cost function for one part at a time.

Backpropagation

The error in the output value is backpropagated to each weight. Weights that have the largest contribution to the final error will have larger activation and larger rate of change of the cost, and therefore will be changed the most by the gradient descent method.

Results and Discussion

Stochastic Gradient Descent

If we take a look at figure (2) we notice that Ordinary Least Square (OLS) seems to have a smaller Mean Squared Error (MSE) value for all 30 polynomial degrees, compared to Stochastic Gradient Descent (SGD). However, the MSE plot for the SGD is more stable, especially for polynomial degrees zero to 13. This comes as no surprise due to the fact that the OLS uses the integrated matrix pseudo inverse (pinv) from numpy to bypass singular matrix issues which in turn will factor out some of the errors for OLS. We also noticed the time it took to run the SGD compared to OLS was much longer which is something one should investigate further.

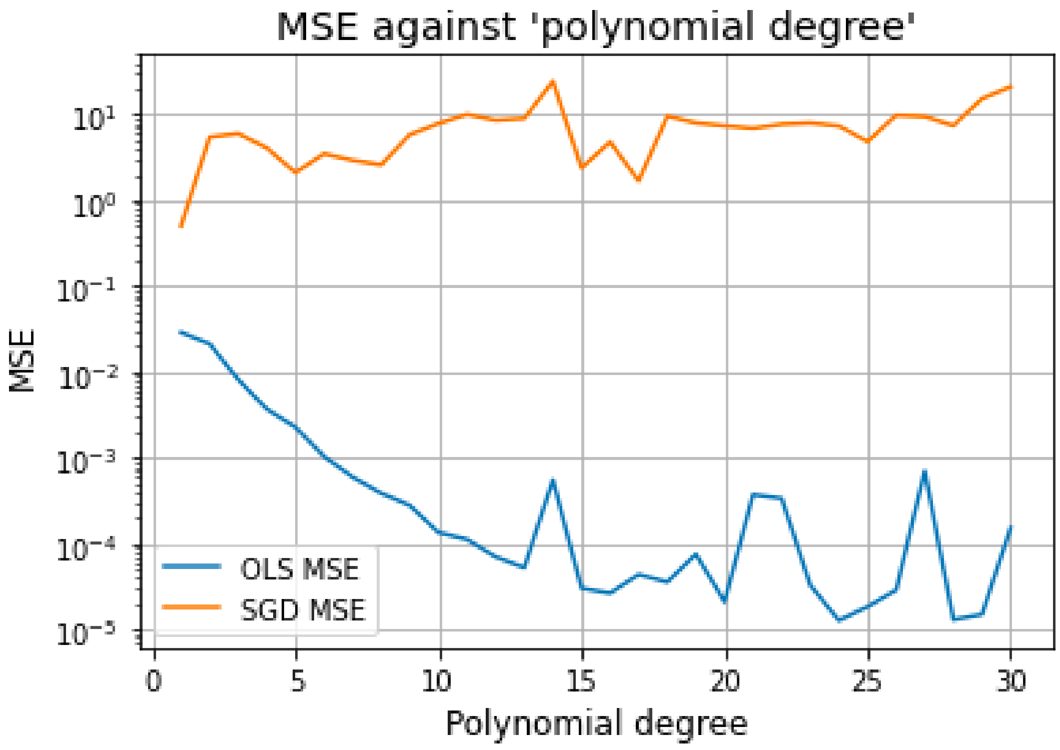

Figure 2: OLS (train) and SGD (train) MSE depending on the polynomial degree.

Neural Network

The number of nodes in the input layer follows the pattern of the triangular numbers for polynomial degree.

The network consists of two hidden layers and one output node. Each hidden layer consists of 5 nodes and the output layer consists of a varying number of nodes for each polynomial degree.

The Mean Squared Error is smaller for Neural Network model compared to the Stochastic Grade Descent and larger compared to the Ordinary Least Square model, as we can see in figure (3).

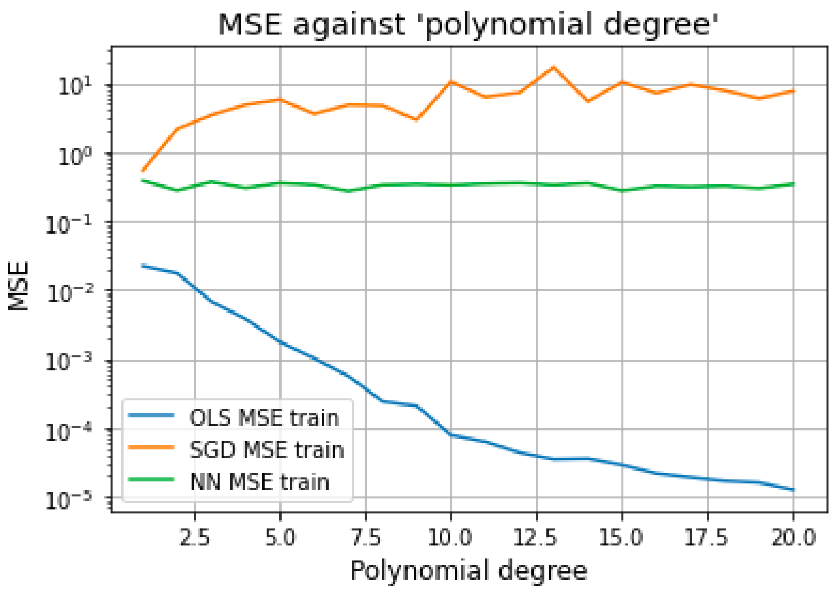

Figure 3: NN MSE compared to OLS and SGD MSE, depending on the polynomial degree. Number of nodes per hidden layer = 5.

The Mean Squared Error increases slightly and the plot flattens with increasing number of hidden nodes, since by increasing the number of nodes, we improve the learning capacity of the model. The plot is very smooth for two hidden layers with 10 nodes each (figure (4)), and gets smoother for larger number of nodes.

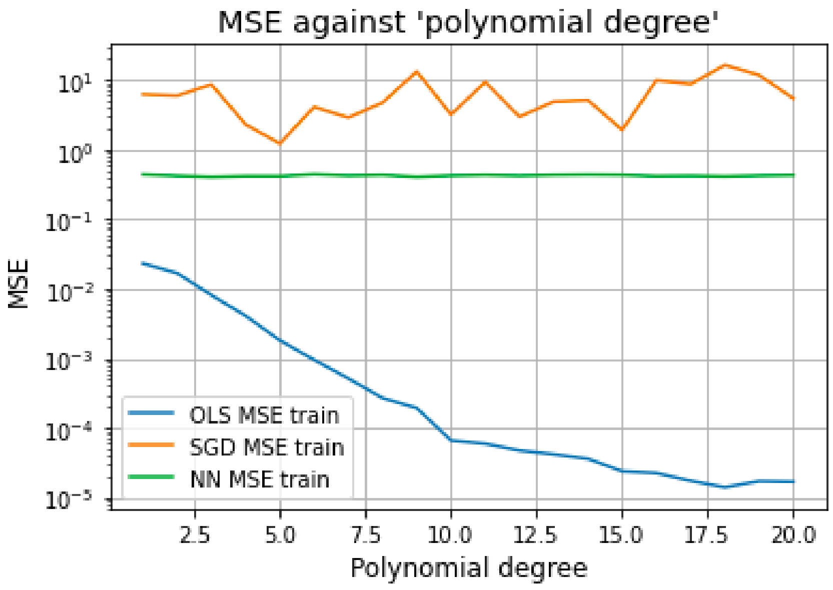

Figure 4: NN MSE compared to OLS and SGD MSE, depending on the polynomial degree. Number of nodes per hidden layer = 10.

The Mean Squared Error of the training and testing in figures (5) with 5 nodes in each of the two hidden layers and (6) with 10 nodes in each of the two hidden layers.

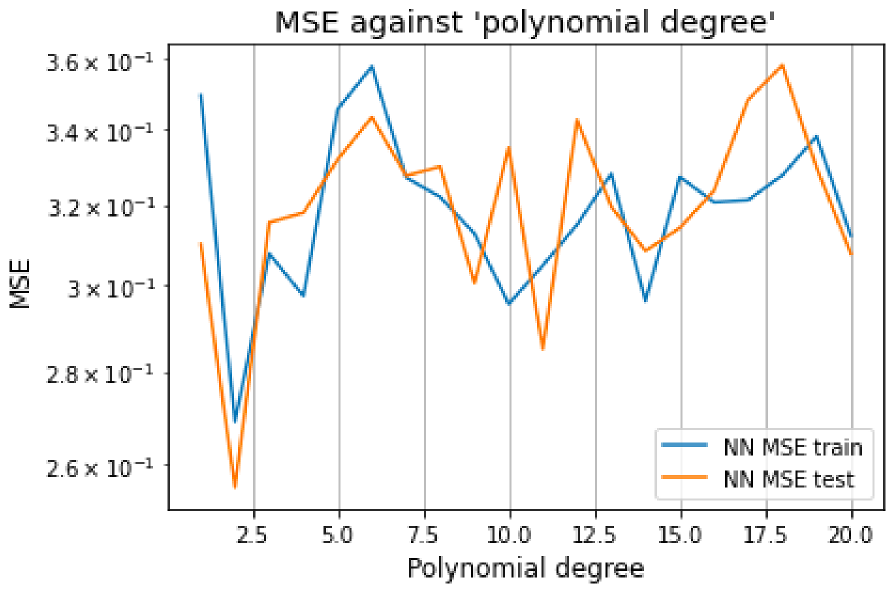

Figure 5: NN MSE train test, depending on the polynomial degree. Number of nodes per hidden layer = 5.

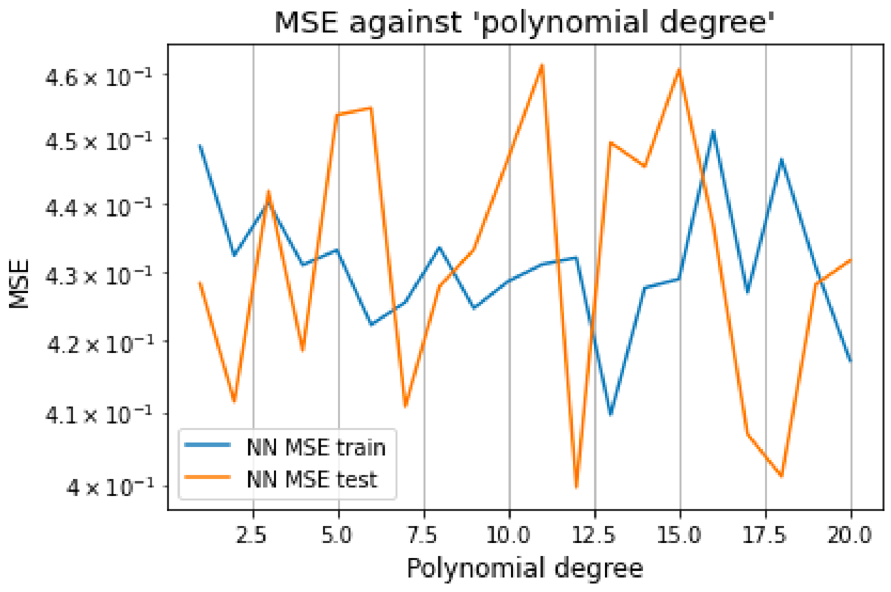

Figure 6: NN MSE train test, depending on the polynomial degree. Number of nodes per hidden layer = 10.

There is a large difference between these two. With larger number of nodes, the model is less effective in predicting new values. 10 nodes per layer leads to a larger total error than 5 nodes per layer.

Conclusion

We have performed regression using Gradient Decent, Stochastic Gradient Decent and momentum gradient decent and compared them to Ordinary Least Square regression. We saw that the Mean Squared Error scores for OLS were lower, but more inconsistent than that of SGD. We then tested our Feed Forward Neural Network. Our results showed that the Neural network performed better than SGD, but worse than OLS, based on the MSE scores. Compared to SGD and OLS the Neural network had a significantly more stable MSE score for increasing polynomial degree. We saw that the most optimal number of nodes per hidden layer for our neural network was five and if we went above ten nodes in each hidden layer our MSE score became higher. For future endeavors we should test the rest of the activation functions such as RELU and Leaky RELU in order to compare them to the Sigmoid function.

References

- S.M. Lommerud, O. Malinowska, "FYS-STK4155 Project on Machine Learning Part 1: Regression analysis and resampling methods", University of Oslo, Fall 2022, `https://github.com/Lommerud/FYS-STK-4155/tree/main/Project%201`, Accessed: 18.11.2022
- M. Hjorth-Jensen, "Applied Data Analysis and Machine Learning. Week 39: Optimization and Gradient Methods", 2022, `https://compphysics.github.io/MachineLearning/doc/pub/week39/html/week39.html`, Accessed: 18.11.2022
- M. Hjorth-Jensen, "Applied Data Analysis and Machine Learning. Week 40: Neural Networks.", 2022, `https://compphysics.github.io/MachineLearning/doc/pub/week40/html/week40.html`, Accessed: 28.10.2022
- Wikipedia, "Neural network", 2022, `https://en.wikipedia.org/w/index.php?title=Neural_network&oldid=1119396315`, Accessed 03.11.2022