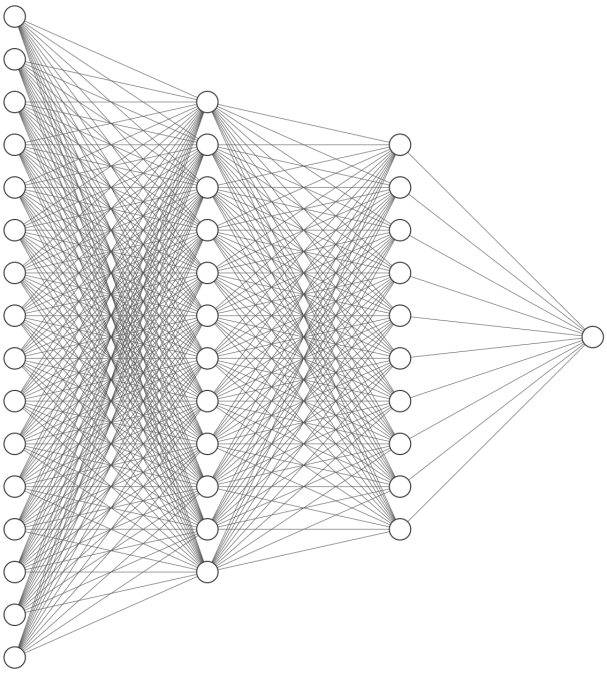
Danmarks Tekniske Universitet

DTU

22110 Project Report: Predicting disease with an Artificial Neural Network



Authors

Eirik Runde Barlaug - s221409
Sara Bakken Heiberg - s221630

November 22, 2022

Contents

1 Introduction

In this project, we have implemented a simple artificial neural network to predict whether certain variations of Single Nucleotide Polymorphisms (SNP) will lead to a disease or not. We implemented the algorithm from scratch as a working pure python neural network, and evaluated its performance. The main goal of the project is creating an ANN in Python with no use of libraries. Thus the focus lies on building the net as well as code quality. The model has therefore not been optimized beyond some satisfactory level.

1.1 Contributions

Both students has contributed equally (50% each) on all parts of the project - both the code base and the report.

1.2 Relevancy of the assigned task

Single nucleotide polymorphisms (SNP), or "snips", is a part of the human DNA which occur almost once every 1000th nucleotide on average [1]. These are the most common type of gene variation among single nucleotides in humans, and they can be used by scientists to locate genes that may be associated with certain diseases. Although most combinations of SNPs do not affect health, some variations may be indicators of diseases. To predict a certain disease based on a humans SNP-information is what we will be trying to achieve in this project.

The data set used for training and testing of the algorithm is a part of a project at DTU HealthTech. It contains different combinations of SNPs in humans and whether or not that human got some disease. For each observation there are 27 input variables, and one target. All inputs are floats, and the target is a binary variable where 1 means that the individual got the disease, and 0 means that the individual did not get the disease.

The predictions of the disease will be done by using an Artificial Neural Network, or an ANN for short. ANNs are computing systems inspired by the biological neural connections that constitute the human brain. A neuron accepts input signals via its dendrites, which pass the electrical signal down to the cell body. The axon carries the signal out to synapses, which are the connections of a cell's axon to other cell's dendrites. Artificial Neural Networks work in a similar way. ANNs can be considered as weighted directed graphs where the nodes work as neurons and the interconnection strength between them are weighted edges. The processing element of a neuron receives many signals and the weighted inputs are being summed. If it crosses a threshold, it goes as input to other neurons (or as output to the external world).

2 Theory

The following section will describe the necessary theory one must understand in order to fully comprehend and evaluate the codebase for this project. The section describes each

part of the neural net - everything from the overall structure to the details of each neuron. It is assumed that the reader has basic knowledge of calculus and linear algebra, as the explanations are somewhat math-intensive.

2.1 The architecture of the neural net

We will in this section describe the structure of an artificial neural network (ANN for short), which is the most basic form of a *feed forward network*. The ANN consists of units called *nodes* or *neurons*. The nodes are structured in *layers*, where each node in a given layer is connected to every node in the next layer through a set of weighted connections, called *edges*, or simply *weights*. The first layer of the ANN is called the *input-layer*, the final layer is called the *output-layer* and all layers in between are called *hidden layers*. An example of a simple ANN structure is given in figure 1.

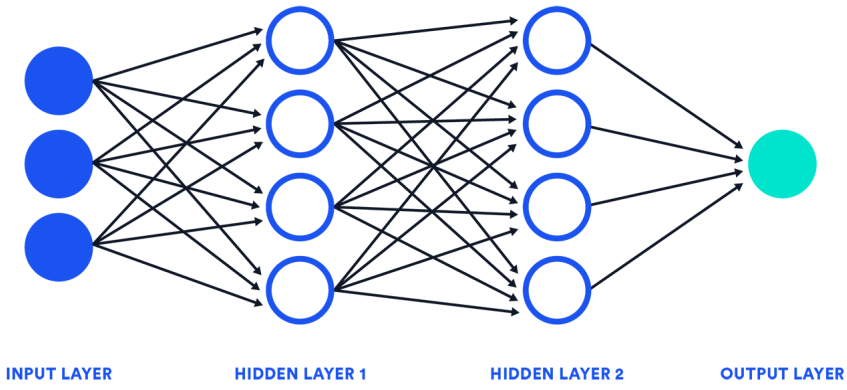


Figure 1: Example of a basic ANN structure.[2]

If the input layer of an ANN consists of N nodes, and the output layer of M nodes, the ANN acts as a function f that maps the input to the output as

$$f : \mathbb{R}^N \to \mathbb{R}^M. \tag{1}$$

In figure 1, we have a 3-dimensional input that is mapped to a single output node. Say, for example, the inputs are height, weight and shoe size of a person, and the output is if the person is a male (1) or a female (0), we then have a neural net that classifies a persons gender based on the mentioned attributes, or in other words a function that maps those three attributes to a gender for a given person (the input).

2.1.1 Weights and biases

We will now introduce some more complexity to the neural net model in figure 1. Suppose now, that we instead have a simple ANN as displayed in figure 2, with two input

nodes, i, two hidden nodes, h, and a single output node, o. Between the nodes are the *weights*. The magnitude of a given weight into a node determines how much that specific input influences the output of the node. The weights can be seen as the *strength of the connection* between two nodes.

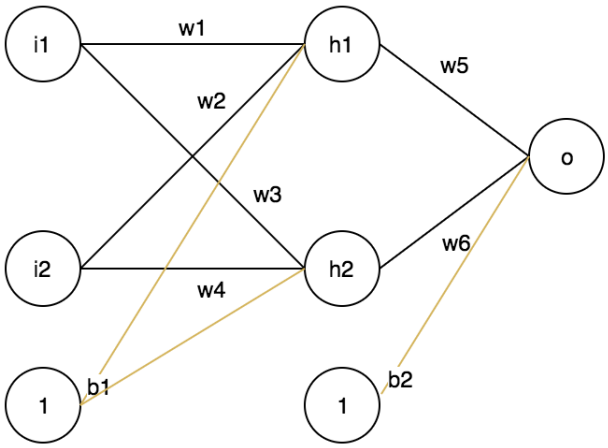


Figure 2: Simple three-layered ANN labeled with weights and biases.[3]

Take for instance h_1 in the figure above. If we let $w_1 = 1$ and $w_2 = 0.1$, then the output from i_1 (i.e. the first input of h_1) will influence the output of h_1 more than the output from i_2. Furthermore, the *biases* are always one, and are also multiplied by their own weights (b_i in the figure above). It is the weights and the biases that we want to tune to the right values in order to fit our model to the data, because these are the parameters that determine the shape of the function f in equation 1.

2.2 A single neuron

As mentioned, the nodes of the ANN are connected to each other through weighted edges. A single node is merely a mathematical function that takes inputs along with the weights and biases, and outputs a real number. The output of a single node, y, is the output of some *non linear* function h to which the input is the weighted sum of all inputs to the node. That is,

$$\begin{aligned} y &= f(x_1, x_2, ..., x_n, w_0, w_1, w_2, ..., w_n) \\ &= h(w_0 + x_1 w_1 + x_2 w_2 + ... + x_n w_n) \\ &= h\left(w_0 + \sum_{i=1}^{n} x_i w_i\right), \end{aligned} \tag{2}$$

where h is a non linear *activation function* and w_0 is the *bias*. With vector notation this may be simplified to

$$y = h(\boldsymbol{x}^\top \boldsymbol{w}), \tag{3}$$

where

$$\boldsymbol{x} = (1\ x_1\ x_2\ ...\ x_n)^\top, \boldsymbol{w} = (w_0\ w_1\ w_2\ ...\ w_n)^\top \in \mathbb{R}^{n\times 1}. \tag{4}$$

Visually, such a node may be described as in figure 3, where s simply represents the weighted sum of the inputs. All other notation is equal to the above formulation.

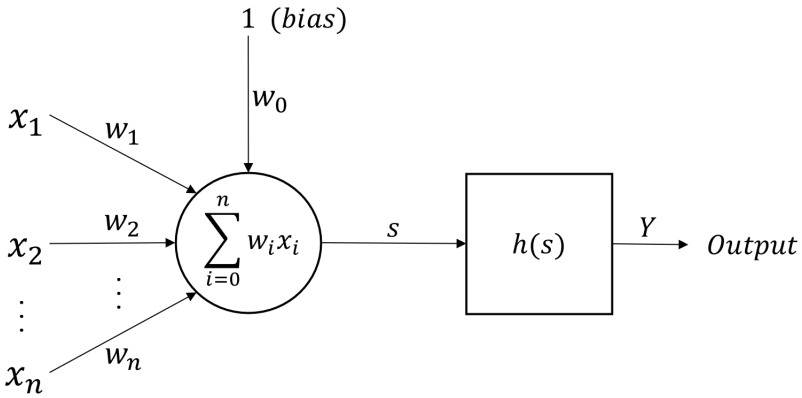


Figure 3: Visual representation of the input-output characteristics of a single node.

When the function h is the identity matrix, or simply 1, the node is referred to as a *perceptron*, and its output is simply the weighted sum of the input.

2.2.1 The activation function

The activation function is a real valued non linear function that takes the weighted sum of the n node-inputs and outputs a single real number. I.e., $h : \mathbb{R}^1 \to \mathbb{R}^1$. The activation function is needed to introduce non-linearity into the model, which is important if the output can not be represented as a linear combination of the input variables; this is often the case in real world classification and regression problems.

There are many common activation functions and some are better than others for certain tasks. For this project, we decided to use the sigmoid activation function,

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \tag{5}$$

which is continuously differentiable, and squeezes all input values into the range $[0, 1]$. Thus, small changes in weights and biases in the network will give small changes in the output, which is desirable for a stable convergence during training. Also, calculating the derivative of the activation function becomes important when backpropagating, as we will see in chapter 2.4.2. This is easy with the sigmoid function, due to the exponential.

2.3 Why biases are included

Let the sigmoid function be the activation function in a simple ANN consisting of a single input, a single output and no hidden nodes in between. The ANN without a bias is described by equation 6, and the ANN with a bias is described by equation 7.

$$f(x) = h(w_1 x) = \sigma(w_1 x) = \frac{1}{1 + e^{-w_1 x}} \tag{6}$$

$$f(x) = h(w_0 + w_1 x) = \sigma(w_0 + w_1 x) = \frac{1}{1 + e^{-(w_0 + w_1 x)}} \tag{7}$$

The two versions of the thought ANN is displayed in figure 4

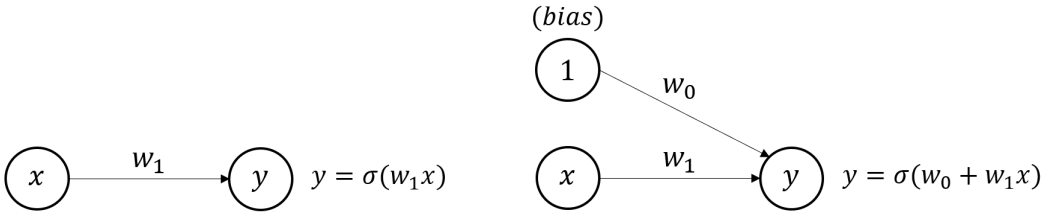


Figure 4: Two simple two-layered ANNs, with and without bias.

Without the bias, the weight w_1 could only change the slope of the activation function, this means that the intercept *must stay in place*, no matter what the weights of the network are. With an added bias w_0, the activation function is allowed to have different intercepts - in other words the activation function can be shifted in the x-direction if desirable, in addition to the slope being changed by w_1. The latter approach introduces a higher degree of flexibility in the network, which in turn results in better learning. In figure 5 some examples of this is displayed with the sigmoid function.

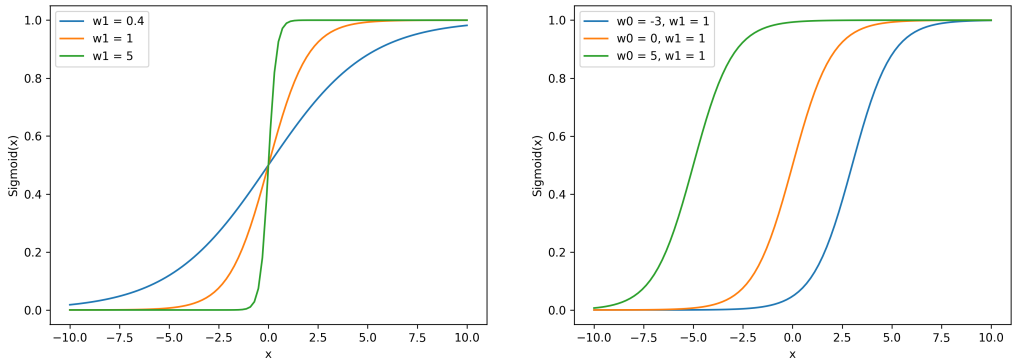


Figure 5: The sigmoid function with different weights and biases.

2.4 How the model is trained

Training the model comes down to minimizing the error between the predicted values and the target values for our data, without overfitting the model to the training data or making the model to biased. This means that the variance between the estimated model parameters across different samples of our data should neither have to much variance (then the model is overfitted to the training data) or to much bias (then the model is underfitted). Unfortunately, increasing one decreases the other. This is known as the bias-variance trade-off, and its proof is outside the scope of this report. The reader is, however, encouraged to study this phenomena to further enhance their understanding of the topic.

The training is supervised, which in essence means that we know the target values (or *true* values) of the predictions during the training. The training consists of two main phases: The *feed forward* and the *backpropagation*. During the former, a sample from the training data set is fed through the network, and a prediction is made. Then, some error between the estimated output and the target value is obtained. The error is then used to update the weights of the network in such a way that the error is minimized. This iteration continues until some average training error is below a predefined tolerance, or the iteration number has grown larger than some maximum number of iterations. The two mentioned phases will now be described in more detail.

2.4.1 The feed forward phase

A one-layer example of the feed forward

Recall the equation for a single neuron,

$$y = h(\boldsymbol{x}^\top \boldsymbol{w}). \tag{8}$$

Say y is the output of a single node, but we want the output for all nodes in a single layer. This may be done by gathering all the weights for all nodes in the layer in a matrix, such that the total output from the layer is presented by a vector $\boldsymbol{y}$, where each row i is the output from node i. Say we have M nodes in the layer, and N inputs. For a such given layer, we have the following structure.

$$\boldsymbol{y} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix} = H\left(\begin{bmatrix} \boldsymbol{w_0} & \boldsymbol{w_1} & \boldsymbol{w_2} & \cdots & \boldsymbol{w_M} \end{bmatrix}^\top \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \right) = H\left(\begin{bmatrix} \boldsymbol{w_0} & \boldsymbol{w_1} & \boldsymbol{w_2} & \cdots & \boldsymbol{w_M} \end{bmatrix}^\top \boldsymbol{x} \right), \tag{9}$$

$$\boldsymbol{y} = H(\boldsymbol{W}^\top \boldsymbol{x}), \tag{10}$$

where the function $H(\cdot)$ denotes the row-wise evaluation of the nonlinear activation function h. Notice that the $(N \times M)$ weight matrix $\boldsymbol{W}$ consists of the stacked weight vectors from equation 8. Furthermore, the indices indicate what neuron in the layer they are the input-

weighs to. Notice also how the weight matrix is transposed. All written out, a single layer input-output equation (10) (i.e. its *transfer function*) reads as follows:

$$\boldsymbol{y} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix} = H\left(\begin{bmatrix} w_{00} & w_{01} & w_{02} & \cdots & w_{0N} \\ w_{10} & w_{11} & w_{12} & \cdots & w_{1N} \\ w_{20} & w_{21} & w_{22} & \cdots & w_{2N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{M0} & w_{M1} & w_{M2} & \cdots & w_{MN} \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \right). \tag{11}$$

In equation 11, the indices ij means the ij-th weight connects the output from node j in the previous layer to node i in the current layer. Notice that the left-most column of the transposed weight matrix in fact is all the bias weights for this layer.

The general feed forward algorithm for a L-layered neural net

More generally, we can define the above example for a L-layered ANN. In the following, the superscripts denotes the layer number.
1: We define $\boldsymbol{a}^{(0)} = \boldsymbol{x}$ as the input to the neural net. Each node i in layer l (of size M) is given its own *activity* $z_i^{(l)} = \tilde{\boldsymbol{x}}^\top \boldsymbol{w}_i^{(l)}$, where $\boldsymbol{w}_i^{(l)}$ is the weights of the connections into node i and $\tilde{\boldsymbol{x}}$ is a vector of inputs to the node.
2: All the M activations are passed through the nonlinear function h and collected in a vector $\boldsymbol{a}^{(l)} = (h(z_0^{(l)})\ h(z_1^{(l)})\ h(z_2^{(l)})\ \ldots\ h(z_M^{(l)}))^\top = (a_0^{(l)}\ a_1^{(l)}\ a_2^{(l)}\ \ldots\ a_M^{(l)})^\top$. This vector is essentially the generalization of $\boldsymbol{y}$ in equation 11, but rather for an arbitrary layer l.
3: For all layers $l = 1, ..., L$ in the network, let $\boldsymbol{a}^{(l)} = H^{(l)}\left(\boldsymbol{W}^{(l)\top}\boldsymbol{a}^{(l-1)}\right) \in \mathbb{R}^M$ be the output from the M nodes in layer l, where M is an arbitrary integer. Often, the activation function $H^{(l)}$ is identical for all L layers.
4: Return the output from the net, namely $\boldsymbol{a}^{(\boldsymbol{L})}$.

Visually, if applied on a two-layered ANN, the algorithm may be described as seen in figure 6. As one can see, the whole acts as a function that takes an observation $\boldsymbol{x}$ and a set of weights $\boldsymbol{W} = (\boldsymbol{W}^{(1)},\ \boldsymbol{W}^{(2)})$ and returns a prediction $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{W})$.

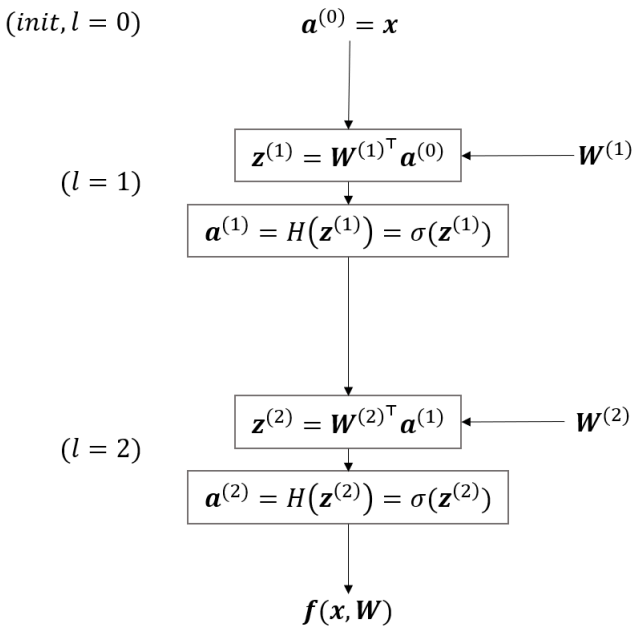


Figure 6: A visual representation of the feed forward algorithm on an ANN consisting of two layers.

2.4.2 The backpropagation phase

Now that the feed forward phase and some notation has been introduced, we may proceed to the core of the training - namely the backpropagation algorithm. The goal of the backpropagation is to update the weights and biases in the neural network in order to minimize the output error. When measuring how good the performance is, we use the concept of a *cost function*, $\boldsymbol{C}$. Here, it is given as the *Mean Square Error* (MSE) between a prediction and the true value for that prediction (12). The total cost of a single training example is the sum of the means of the squares of the differences between each actual and predicted output values.

$$\boldsymbol{C} = \frac{1}{2} \cdot (\hat{\boldsymbol{y}} - \boldsymbol{y})^2 \tag{12}$$

We aim to reduce the MSE by changing the weights such that the cost converges to a minimum. Whether we should increase or decrease the weights, is decided by using the *Gradient Descend algorithm*.

2.4.3 Gradient descent

The objective of the gradient descent algorithm is to start from a random point (random weights) and in an iterative manner find a way to update this point until we reach the lowest point of the cost function, i.e. find the weights that give the smallest error. The gradient of a function is the direction of the steepest ascent, i.e. the direction that increases the error

the most. For each iteration, we compute the gradient at the point we are at (which is given by the currently used weights). By taking a small step in the opposite direction, we hope to move towards the (local) minimum which will be where the gradient is equal to zero. The step size is called the *learning rate* and it is a scalar that determines how much the weights change in each iteration, and therefore how quickly the error of the ANN converges. Larger learning rates make the algorithm take big steps down the slope and it might jump across the minimum point, thereby missing it. It is therefore smart to choose a smaller learning rate, but not too small as it will make the progression very slow. To update the weights, the gradient is multiplied by this learning rate (α), and then subtracted from the old weights as

$$\boldsymbol{W}_{new} = \boldsymbol{W}_{old} - \alpha \frac{\partial \boldsymbol{C}}{\partial \boldsymbol{W}}. \tag{13}$$

Now we will be illustrating how backpropagation works by using a simple ANN consisting of one input layer, one output layer and one hidden layer. There is in other words four parameters needed to be updated and four gradients needed to be calculated. That is,

$$\frac{\partial \boldsymbol{C}}{\partial \boldsymbol{W}^{(1)}}, \frac{\partial \boldsymbol{C}}{\partial \boldsymbol{W}^{(2)}}, \frac{\partial \boldsymbol{C}}{\partial \boldsymbol{w_0}^{(1)}}, \frac{\partial \boldsymbol{C}}{\partial w_0^{(2)}}. \tag{14}$$

We use the chain rule to calculate the gradients backward through the layers of the neural network. Lets follow the paths of the weights $\boldsymbol{W}^{(1)}$ and $\boldsymbol{W}^{(2)}$, and the biases $\boldsymbol{w_0}^{(1)}$ and $w_0^{(2)}$ (where the former bias term is a vector and the latter is a scalar):

$$\frac{\partial \boldsymbol{C}}{\partial \boldsymbol{W}^{(2)}} = \frac{\partial \boldsymbol{C}}{\partial \boldsymbol{a}^{(2)}} \frac{\partial \boldsymbol{a}^{(2)}}{\partial \boldsymbol{z}^{(2)}} \frac{\partial \boldsymbol{z}^{(2)}}{\partial \boldsymbol{W}^{(2)}} = (\boldsymbol{a}^{(2)} - \boldsymbol{y}) \cdot \boldsymbol{a}^{(2)}(1 - \boldsymbol{a}^{(2)}) \cdot \boldsymbol{a}^{(1)} \tag{15}$$

$$\frac{\partial \boldsymbol{C}}{\partial w_0^{(2)}} = \frac{\partial \boldsymbol{C}}{\partial \boldsymbol{a}^{(2)}} \frac{\partial \boldsymbol{a}^{(2)}}{\partial \boldsymbol{z}^{(2)}} \frac{\partial \boldsymbol{z}^{(2)}}{\partial w_0^{(2)}} = (\boldsymbol{a}^{(2)} - \boldsymbol{y}) \cdot \boldsymbol{a}^{(2)}(1 - \boldsymbol{a}^{(2)}) \cdot 1 \tag{16}$$

$$\begin{aligned} \frac{\partial \boldsymbol{C}}{\partial \boldsymbol{W}^{(1)}} =& \frac{\partial \boldsymbol{C}}{\partial \boldsymbol{a}^{(2)}} \frac{\partial \boldsymbol{a}^{(2)}}{\partial \boldsymbol{z}^{(2)}} \frac{\partial \boldsymbol{z}^{(2)}}{\partial \boldsymbol{a}^{(1)}} \frac{\partial \boldsymbol{a}^{(1)}}{\partial \boldsymbol{z}^{(1)}} \frac{\partial \boldsymbol{z}^{(1)}}{\partial \boldsymbol{W}^{(1)}} \\ =& (\boldsymbol{a}^{(2)} - \boldsymbol{y}) \cdot \boldsymbol{a}^{(2)}(1 - \boldsymbol{a}^{(2)}) \cdot \boldsymbol{W}^{(2)} \cdot \boldsymbol{a}^{(1)}(1 - \boldsymbol{a}^{(1)}) \cdot \boldsymbol{x} \end{aligned} \tag{17}$$

$$\begin{aligned} \frac{\partial \boldsymbol{C}}{\partial \boldsymbol{w_0}^{(1)}} =& \frac{\partial \boldsymbol{C}}{\partial \boldsymbol{a}^{(2)}} \frac{\partial \boldsymbol{a}^{(2)}}{\partial \boldsymbol{z}^{(2)}} \frac{\partial \boldsymbol{z}^{(2)}}{\partial \boldsymbol{a}^{(1)}} \frac{\partial \boldsymbol{a}^{(1)}}{\partial \boldsymbol{z}^{(1)}} \frac{\partial \boldsymbol{z}^{(1)}}{\partial \boldsymbol{w_0}^{(1)}} \\ =& (\boldsymbol{a}^{(2)} - \boldsymbol{y}) \cdot \boldsymbol{a}^{(2)}(1 - \boldsymbol{a}^{(2)}) \cdot \boldsymbol{W}^{(2)} \cdot \boldsymbol{a}^{(1)}(1 - \boldsymbol{a}^{(1)}) \cdot 1 \end{aligned} \tag{18}$$

These expressions are obtained by following the chain rule straight forward. However the calculations are somewhat extensive, and not fully included in this report. After computing the gradients, we can update the weights and biases for the ANN, and iterate all over again until the error is sufficiently small. A visual representation of the backpropagation is given in figure 7.

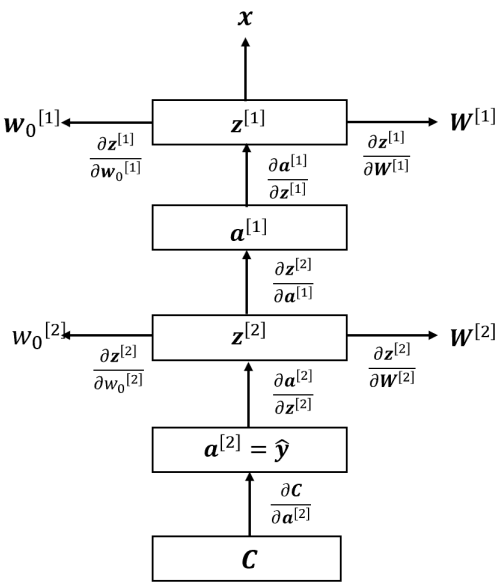


Figure 7: A visual representation of the backpropagation algorithm on an ANN consisting of two layers.

To summarize, each iteration begins with a forward pass that outputs the current prediction of the parameters, and then evaluates the error of the network by the cost function. Finally, backpropagation is done in order to update the parameters of the model.

3 Algorithm design

Our neural network consists of an input layer, one hidden layer and an output layer. There are 27 input nodes, 14 hidden nodes, and 1 output node that predicts the target variable.

For this project, we were not allowed to use built-in libraries. Hence we have implemented the help functions ourselves. In the file `helpers_matrix.py`, we have defined functions for matrix transpose (`transpose()`), dot product (`dot()`), scalar multiplication (`scalar_mult()`), element-wise difference (`diff()`), etc., which are used by the other central functions in the ANN. In the following, we will explain the main algorithms of our program.

3.1 Initialization of weights and biases

It is important to initialize the weights and biases appropriately to ensure a model with high accuracy. The function `init_weights_and_bias()` takes two adjacent layers as arguments, and assigns a pseudo-random number between 0 and 1 (following a normal distribution) to every parameter, i.e. to every weight and bias. Random initialization allows us to break symmetry and to make all the nodes in the neural network behave differently. The function returns the initial weights and biases between two layers in the network, represented

as a matrix and a vector, respectively. The weight matrix has number of rows equal to the number of nodes in the second layer, and number of columns equal to the number of nodes in the first layer. The bias vector has length equal to the number of nodes in the second layer.

3.2 The feed forward algorithm

As mentioned in subsection 2.4.1, the data enters the input layer, travels through the hidden layer, and eventually exits the output layer. This is implemented as the function `feed_forward()`. The function takes in the input observation ($\boldsymbol{x}$), a list containing all of the weight matrices in the network ($\boldsymbol{weights}$) and a list containing all the bias vectors ($\boldsymbol{biases}$) as parameters. For our two-layered network in particular, the weight list will contain two weight matrices and the bias list will contain two bias vectors (where the last vector is merely a scalar).

First, the function unpacks the different weight matrices and bias vectors. Then it multiplies the observation vector $\boldsymbol{x}$ with the weights of the first layer $\boldsymbol{w_1}$ and adds the corresponding biases $\boldsymbol{b_1}$. Next, the sigmoid function is applied on all elements of the output vector from the first layer. The result is then being multiplied with the weights of the second layer, $\boldsymbol{w_2}$, and the bias $\boldsymbol{b_2}$ is added. Once again, the result is passed through the sigmoid function, and we get the output of the second layer. The value of the end node is given as a float number between 0 and 1, and tells the probability of having a disease. This probability is the return value of the function.

3.3 The backpropagation algorithm

As described in subsection 2.4.2 the objective of the backpropagation algorithm is to update the weights and biases in the multi-layer feed forward network, based on the error between the output of the feed forward (i.e. the estimate: $\hat{y}$) and the correct target value y.

The function `backpropagate()` takes the input vector ($\boldsymbol{x}$), target vector ($\boldsymbol{y}$), and the weights and biases of the network. Just like the `feed_forward()` algorithm, it unpacks the weights and biases belonging to the two different layers, and then calculates the outputs from the linear functions ($\boldsymbol{z1}$ and $\boldsymbol{z2}$) and from the activation functions ($\boldsymbol{a1}$ and $\boldsymbol{a2}$). This is done for both layers, as described in figure 6. Next, the four gradients are calculated just as explained in section 2.4.3. In the end, all the weights and biases are updated and returned.

3.4 The training algorithm

The overall goal of supervised learning is to build a model that performs well on new data. We can simulate the experience of having new data by splitting the data into a training and a testing set. The data from 3 of the files given in the project description [4] are used as training set, while the fourth one is used as a test set. As the name indicates, the training set is used to train and estimate the parameters (weights and biases) of the model.

The function `train()` takes in the training data ($\boldsymbol{x\_train}$ and $\boldsymbol{y\_train}$), all the weights and the biases in the network, the learning rate described in 2.4.3 ($\boldsymbol{\alpha}$), the number

of times to iterate through all the training data (***num_epochs***) and the error tolerance (***tol***). It runs the feed forward and backpropagation algorithm, and calculates the mean square error from the forward pass. This is done until the number of iterations is equal to ***num_epochs*** (which is equal to 100 by default) *or* until the MSE is less than the tolerance ***tol*** (0.05 as default value). Between epochs, all the training data is shuffled to prevent the network from creating biases. The function returns the tuned weights and biases, in addition to the average errors for all iterations.

3.5 The prediction and evaluation algorithm

As already mentioned, the feed forward algorithm returns the probability that a specific SNP variations will lead to disease. In order to conclude, we must assign the predicted value either the value 0 or 1 (1=disease, 0=health). This is the purpose of the function `predict()`. It takes in the test data, and the tuned weights and biases of the network, and runs the feed forward function. If the predicted probability of disease is above 0.5, the SNP combination is classified as one leading to disease (i.e. it is given the value 1). If the prediction is lower or equal to 0.5, it is classified as a combination that will not give a disease (i.e. given the value 0). The function also calculates the error, and returns both the classified predictions (vector consisting of 0's and 1's for all input data) and the mean square errors (vector of MSE for all the predictions).

After the training is done and we have estimated the parameters of the ANN, we evaluate its performance on new, unseen data - namely the test data. This is exactly how we expect to use the ANN in practice; to predict whether new variations of SNP will lead to disease or not.

The function `evaluate()` evaluates the performance of the model based on the predicted and true class labels, i.e. if the ANN's predictions of whether certain variations of SNP leads to disease is correct or not. It takes in the test set (***x_test*** and ***y_test***), and the class labels (***y_pred***) and the errors predicted in the function `predict()` as parameters. Then it compares all the predicted values with the true values from the test set ***y_test*** and evaluates the overall performance of the ANN in terms of the accuracy measure:

$$\text{Accuracy} = \frac{\#\text{ correctly classified observations from the test set}}{\#\text{ observations in the test set}}. \tag{19}$$

3.6 Runtime analysis of key algorithms

In order to evaluate the runtime of the algorithm, we are interested in looking at its complexity. Big-***O*** notation is a way of expressing the algorithmic complexity and describes how the runtime of an algorithm changes as the input size grows (worst case). Hence, we are looking at the general case where we have n input nodes, m hidden nodes and k output nodes.

We start by looking at the time complexity of some of the helper functions:

- The time complexity of matrix multiplication for $M_{nm} \cdot M_{mk}$ is $\boldsymbol{O}(nmk)$.

- The time complexity of matrix transpose of M_{nm} is $\boldsymbol{O}(nm)$.
- The time complexity of element-wise difference between two vectors/matrices of size m is $\boldsymbol{O}(m^2)$.

In the below, we will be looking at the runtimes of the central algorithms, described in chapter 3.

init_weights_and_biases:
The function `init_weights_and_biases()` only initializes the weight matrix and bias vector between two layers in the network. So in general, the time complexity of the function is $\boldsymbol{O}(nm+m) = \boldsymbol{O}(nm)$, where n is the number of nodes in the first layer and m is the number of nodes in the second layer.
The initialization of the first layer of our ANN has a time complexity of $\boldsymbol{O}(nm+m) = \boldsymbol{O}(nm)$. Initializing the second layer has a time complexity of $\boldsymbol{O}(mk + k) = \boldsymbol{O}(mk)$.

feed_forward:
If there are n input nodes, m hidden nodes and k output nodes, there will be performed $n \cdot m$ multiplications in the first hidden layer. Each hidden node performs a linear combination of its inputs followed by the application of the sigmoid function. Hence, each hidden node j performs the following operation:

$$a_j = \sigma(\sum_{i}^{n} w_{ij} x_i), \tag{20}$$

where i is the input coming from the input node i, w_{ij} is the weight of the connection from the input node i to the hidden node j, and a_j is the output of node j. The sigmoid function (σ) always runs in the same amount of time, and it has a constant time complexity of $\boldsymbol{O}(1)$.

Then each node j in the next layer (i.e. the output layer), also performs a linear combination followed by the application of the activation function,

$$a_j = \sigma(\sum_{i}^{m} w_{ij} x_i). \tag{21}$$

Similarly to the previous reasoning, there are $m \cdot k$ multiplications at the output layer. Thus, the ANN will perform $(n \cdot m) + (m \cdot k)$ multiplications. The time complexity of the algorithm is then $\boldsymbol{O}(nm + mk)$. Note that it is the matrix multiplications that dominates the runtime of the algorithm.

backpropagation:
The backpropagation algorithm will also be dominated by the matrix multipliaction. Just like the `feed_forward` algorithm, it has a time complexity of $\boldsymbol{O}(nm + mk)$

train:
The train algorithm will run both forward pass and backpropagation on all of the training

data and shuffle the data in between epochs. Lets say there are i observations (i.e. number of variations of SNP) in the training set. The number of iterations is equal to the number of epochs (e). Shuffling of the data is done with `random.shuffle()` which uses the Fisher-Yates shuffle algorithm, this algorithm runs on $\boldsymbol{O}(n)$. The shuffling is done once every epoch, and both training and evaluation is done twice per observation i, for e iterations, unless the training is ended early due to low enough error. Hence, the time complexity of the train algorithm is given by $\boldsymbol{O}(e(2i(nm + mk) + 1))$.

predict:
For each observation in the test set, it predicts the output of the neural net, i.e. sends every observation through the forward pass algorithm once. If there are i observations in the test set, the algorithm will have a time complexity of $\boldsymbol{O}(i(nm + mk))$.

evaluate:
For each observation in the test set, it compares the predicted value against the real value, which takes constant time. Hence, the runtime is equal to the number of observations in the test set.

4 Program design

The following section will describe the overall structure of the program, so that the reader can maneuver the attached files with ease.

4.1 Modules and their responsibilities

The modules (with its files) and their responsibilities are summarized in table 1.

Module	**Files**	**Responsibility**
Utilities	`file_utils.py`	Utilities for writing weights and biases to a .txt-file and reading & transforming them back to matrices on the right format.
	`helpers_matrix.py`	Utility functions for matrix and vector operations. Linear algebra.
	`train_utils.py`	Utilities for training of the ANN, as well as some globals describing the structure of the net.
	`predict_evaluate_utils.py`	Utilities for prediction and evaluation.
Data reading	`read_data.py`	Reads the data, transforms it into a the correct shape and datatype, and divides it into training and test datasets.
Training	`train.py`	Initializes, creates and trains the ANN model based on the training data.
Prediction and evaluation	`predict_evaluate.py`	Predicts (classifies) the test data by using the tuned weights and biases read from file (i.e. the trained model), and evaluates the performance of the model.

Table 1: Summary of the modules with consisting files and belonging responsibilities.

4.2 Dependencies

In figure 8 the dependencies of the program is described. An arrow as $A \rightarrow B$ corresponds to a dependency where B is dependent on some data and/or functionality from A. The weights and biases are written to a file between the training and the prediction. This is quite important, because if not, the model must be re-trained each time the user wants to make a prediction on some test data or an entirely new observation. As of now,

the model must only be trained *once*, which saves time and makes the results reproducible. For comparison, the training (with $\alpha = 0.005$ and $n_{\text{epoch}} = 50$) took 36.975 seconds, while the prediction took 0.343 seconds. This testing was done with a train-test split of $^3/_4$ to $^1/_4$, on an Intel(R) Core(TM) i5-8265U CPU 1.60GHz (8 CPUs), 1.8GHz processor with 8 GB RAM.

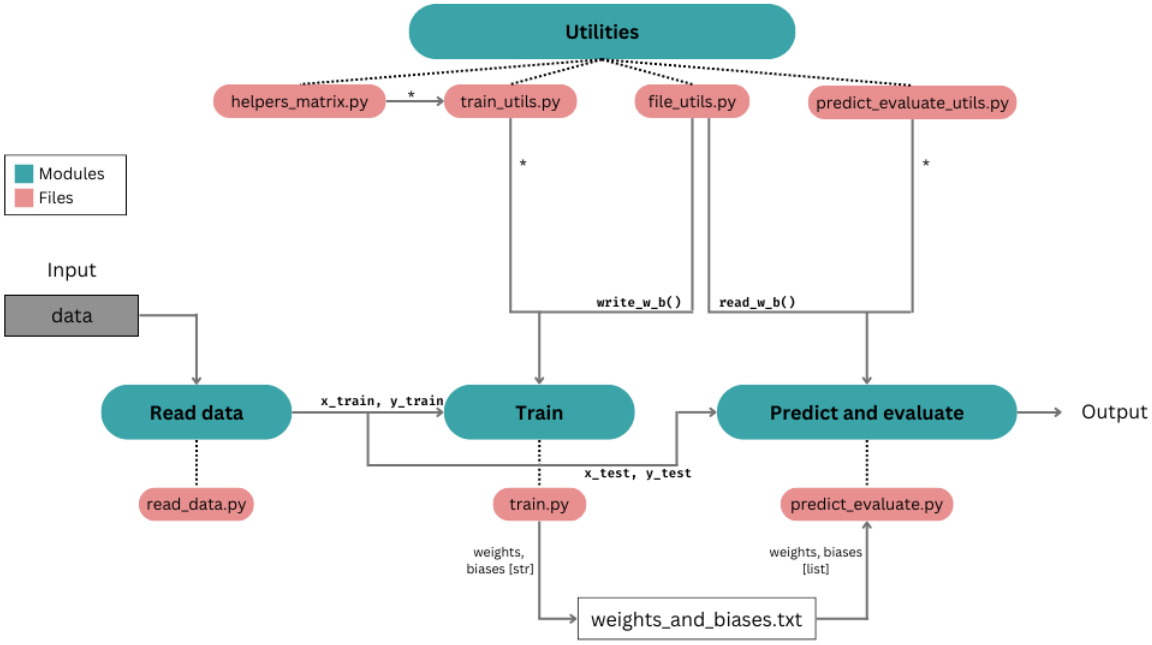


Figure 8: The dependecies between the modules of the program.

5 Program manual

The reader may, if needed, alter the shape of the data in `read_data.py`. Then, it should be run to read in the data from the data folder. Secondly, the file `train.py` should be run once in order to obtain a model that is saved to *weights_and_biases.txt*. The learning rate, α, and the number of epochs can be changed if wanted to. Finally, the file `predict_evaluate.py` should be run once to obtain predictions for the test data. Information on the accuracy of the model will then be printed to screen.

6 Conclusion

The ANN ended up with an accuracy of around 0.75-0.80, which we consider sufficient given the task. Not much time was spent on optimizing the hyperparameters of the network,

but some testing was done: We did check the training error for different step sizes, α. While testing we observed that with a smaller step size, the error converged slower, while with a large step size, the error trajectory became more noisy. This makes intuitively sense, given the theory. We ended up with $\alpha = 0.005$ as the final value, which gave us a middle ground between fast drops in error, but still a non-noisy result. More concisely, with our final model with $\alpha = 0.005$ and $n_{epoch} = 50$, we ended up with the following:

- Number of correct classified input data points: 998. That is 74.256% accurate
- Number of incorrect classified input data points: 346. That is 25.744% inaccurate.

where the testing was done on the fourth data set, i.e. 25% of the data was used for testing. Note that this configuration most likely is *not* the optimal one. We found the results satisfactory given the task, however, the model could be improved by e.g. utilizing different activation functions, using a different number of nodes and layers, or changing the error function and thus the backpropagation algorithm. To further improve the accuracy, the Binary Cross-Entropy Loss Function could have been used instead of the MSE - this is common when ANNs are used for binary classification.

References

[1] N. L. of Medicine (US), "What are single nucleotide polymorphisms (snps)?," *Medline Plus*, 2022.

[2] "Understanding neural networks." https://towardsdatascience.com/understanding-neural-networks-677a1b01e371. Accessed: 2022-10-25.

[3] "Learning mechanism of artificial neural networks: Backpropagation." https://cetinsamet.medium.com/learning-mechanism-of-artificial-neural-networks-backpropagat Accessed: 2022-10-25.

[4] "Artificial neural network." https://teaching.healthtech.dtu.dk/22110/index.php/Artificial_Neural_Network. Accessed: 2022-11-10.