Profielwerkstuk

Machine Learning, Neural Networks and the evolution of Neural Networks through Augmenting Topologies

-

describing the underlying theory, methods and the particular application of these principles to making a computer teach itself how to play a video game

by Maxim L. van den Berg, Sam R.W. van Kampen,
Wilco M. Stam and Robin A.J. Wacanno

January 29, 2017

Abstract

Lekker abstract

Contents

1 Introduction

- Discuss why we chose the subject
- Short introduction subject
- Explain that we will assume the reader doesn't really know anything about neural networks yet.
- Define what the reader should now already. For example: "It is expected that basic matrix functions and basic programming syntax are understood."

2 Background information

2.1 The automation of algorithmic problem solving

2.1.1 History

Before giving some background information on the subjects which will be described in this report, it is useful to gain some knowledge about the historical context in this field of research. Therefore, some of the most significant advancements will be described in the following paragraphs, in addition to which their importance to the subject of this report will be discussed.

2.1.1.1 Bayes' Theorem

The history of machine learning begins not just decades but centuries ago. It begins with the posthumous publication of *An Essay towards solving a Problem in the Doctrine of Chances*, a work by the Reverend Thomas Bayes regarding the mathematical theory of probability. In the *Essay*, Bayes lays out a number of statistical propositions, the fifth of which delineates his seminal theorem. The *Essay* describes the following:

> *If there be two subsequent events, the probability of the* 2^{nd} $\frac{b}{N}$ *and the probability of both together* $\frac{P}{N}$*, and it being* 1^{st} *discovered that the* 2^{nd} *event has also happened, the probability I am right is* $\frac{P}{b}$.[1]

This symbolically implies the following[2]:

$$P(B|A) = \frac{P(A \cap B)}{P(A)}, \text{ if } P(A) \neq 0,$$

which leads to Bayes' Theorem for conditional probabilities:

$$\Rightarrow P(A|B) = \frac{P(B|A)P(A)}{P(B)}, \text{ if } P(B) \neq 0.$$

The theorem describes the relation between a prior probability $P(A)$ and a posterior probability $P(A|B)$ given $P(B|A)$, the probability of observing B when given that A is true:

$$P(A|B) \propto P(B|A)P(A).$$

A is called a conditional probability since the occurrence of event B influences the probability of A's occurrence. An intuitive example is the classification of fruit. When

[1] Bayes (1763).
[2] Stigler (1982).

given a random fruit and asked to determine whether it is an apple or an orange, the probabilities for either class is 50%. When given the additional evidence that the fruit is red, probabilities obviously shift in favour of the apple. The conditional probability of the fruit being an apple increases, given the evidence that the fruit is red.

This theorem is used in machine learning to create Naive Bayes classifiers. These are models that assign class labels to problem instances (sets of features $x_1, x_2, \ldots, X_n$), class labels drawn from a finite set, based on conditional probabilities.

Abstractly, a classifier based on conditional probability assigns instance probabilities

$$P(C_k|x_1, \ldots, x_n)$$

for each class C_k. It is, however, infeasible to base such a model on probability tables when the number of features n is large, or if a feature can take on a large number of values. We can use Bayes' Theorem to decompose the probability as follows:

$$P(C_k|x) = \frac{P(C_k)P(x|C_k)}{P(x)}$$

The numerator is equivalent to the joint probability $P(C_k, x_1, \ldots, x_n)$, which can be rewritten as follows:

$$\begin{aligned}
P(C_k, x_1, \ldots, x_n) &= P(x_1, \ldots, x_n, C_k) && (1)\\
&= P(x_1|x_2, \ldots, x_n, C_k)P(x_2, \ldots, x_n, C_k) && (2)\\
&= P(x_1|x_2, \ldots, x_n, C_k)P(x_2|x_3, \ldots, x_n, C_k)P(x_3, \ldots, x_n, C_k) && (3)\\
&= \ldots && (4)\\
&= P(x_1|x_2, \ldots, x_n, C_k)P(x_2|x_3, \ldots, x_n, C_k)\ldots P(x_{n-1}|x_n, C_k)\\
&\quad P(x_n, C_k)P(C_k) && (5)
\end{aligned}$$

Now, the "naive" part of Naive Bayes comes into play. The algorithm assumes each feature F_i is conditionally independent of every other feature F_j for $j \neq i$, given C. This means that

$$P(x_i|x_{i+1}, \ldots, x_n, C_k) = P(x_i|C_k)$$

And the joint probabilities above can be written as

$$\begin{aligned}
P(C_k|x_1, \ldots, x_n) &\propto P(C_k, x_1, \ldots, x_n) && (6)\\
&\propto P(C_k)P(x_1|C_k)P(x_2|C_k)\ldots && (7)\\
&\propto P(C_k)\prod_{i=1}^{n} P(x_i|C_k) && (8)
\end{aligned}$$

Under the above independence assumptions, the initial instance probability equation can be simplified to:

$$P(C_k|x_1, \ldots, x_n) = \frac{1}{P(x)}P(C_k)\prod_{i=1}^{n} P(x_i|C_k)$$

This probability model is then combined with a decision rule, a common one being choosing the hypothesis that is most probable, the *maximum a posteriori* decision rule. The corresponding classifier is the function that assigns a class $\hat{y} = C_k$ as follows:

$$\hat{y} = \underset{k \in \{1,\dots,K\}}{\operatorname{argmax}} \; p(C_k) \prod_{i=1}^{n} p(x_i | C_k).$$

2.1.1.2 Turing Test

In the year 1950, computer scientist Alan Turing published a seminal paper in the philosophical magazine *Mind*.[3] This publication proved an important step in the development of Machine Learning algorithms and Artificial Intelligence. In this paper Turing ponders the question whether machines are able to think. He does this by trying to rephrase the question, for a question such as *"Can a machine think?"* contains vague concepts. Concepts such as *'think'* and *'machine'*, which are not clearly and universally definable.

Turing proposes that, instead of asking this question, one should seek to find out whether a machine is able to play the so called **Imitation Game**. In Turing's version of this game, more commonly know as the **Turing Test**, there are three participants: a machine, a human and a judge (also a human). Let us call these A, B and C respectively. Each of the participants are held in isolated rooms and participant C is only able to communicate with participants A and B with a terminal, through which he is able to ask both of them questions. The aim of the game is for participant C to correctly identify whether participants A and B are human or machine. Participants A and B however, both try to convince participant C they are in fact human. Thus *'think'* has been redefined as being able to act indistinguishably from an entity that, without a doubt, is able to think: a human.

Now, Turing has only solved half of the problem; he still has to come up with a way to unambiguously define the word *'machine'*. This is vital for there are a number of objects which could classify as a machine but would not be suitable to take this test. Turing has an answer to this however, for he thinks this theory should only apply to binary machines. According to him there are two important reasons for using these kinds of machines. For one, they are already available; their existence is not debatable. The second reason is based on one of his theories, which states that any binary or digital machine can simulate any other digital machine. This means that, if *any* digital machine is able to pass the **Turing Test**, *all* of these machine should—in theory—be able to pass the test.

Thus Turing is able to ask the following, more specific question:

> *Is there a conceivable computer D, one with the adequate amount of computing power and the right program, that is able to play the Imitation Game as participant A against participant B—who is human—in such a way that judge C is unable to correctly determine whether participant A or B is the real human?*

In this way, the Turing Test has been used for decades to test the *'intelligence'* of certain Machine Learning algorithms and Artificial Intelligences.

[3] Turing (1950).

2.1.1.3 SNARC

One year after Turing's publication, Marvin Minsky finished building his Stochastic Neural Analogue Reinforcement Calculator, also know as SNARC. SNARC was a network of 40 randomly connected Hebb synapses. The SNARC was based on a model created by Warren McCulloch and Walter Pitts in 1943 wherein they proposed a model of artificial neurons where each neuron is characterized as being *on* or *off*, with a switch to *on occurring in response to stimulation by a sufficient number of neighbouring neurons. With the addition of a rule that modifies the connection strength between neurons the network gains the ability to improve it's performance. This rule is now called **Hebbian Learning**.

The SNARC consisted of 40 neurons connected randomly and each neuron consisted of a long term memory and a short term memory. The long term memory is the probability that if a signal comes into on of the inputs that then a signal will come out of the output. This probability varies from 0—if the potentiometer is turned down—to 1—if the potentiometer is turned all the way up. If a signal gets through then the short term memory activates which consist of a capacitor and if the operator of the SNARC liked the outcome it could reward the neurons that fired by pressing a button that changed the probability of a neuron firing based on the state of the short term memory. The network achieves this by running a chain powered by a motor along all the potentiometers which turns the shaft of the potentiometers when the electric clutch is engaged. This only happens when the neuron has received a signal and thus the capacitor holds a charge.

2.1.2 The basics of Machine Learning

2.1.3 Supervised and Unsupervised Learning

> *"[Machine Learning is] The field of study that gives computers the ability to learn without being explicitly programmed."*
> *Arthur Lee Samuel (1959), a pioneer in the field of computer gaming, artificial intelligence, and machine learning.*
>
> *"Well-posed Learning Problem: A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E."*
> *Tom M. Mitchell, a famous computer scientist and university professor.*

2.2 Neural networks - a brief introduction

- Biology lesson about the brain and how neural networks work alike, especially evolving neural networks

2.3 Examples of applications of Machine Learning and Neural Networks

- Examples of neural network implementation and what they have achieved

3 Machine learning

3.1 Mathematical theory and model

3.1.1 Introduction and overview

As discussed previously, machine learning refers to the study and construction of mathematical algorithms with the goal of learning from and making predictions on data. There

are two main basic types of machine learning models, of which other more complex algorithms are often based, and each with their own applications: linear regression and logistic regression. Both are conceptually and in terms of implementation very similar, and a good understanding of linear regression will also provide the knowledge to successfully grasp the concepts of logistic regression.

The basic function of linear regression is to model the relationship between one or more explanatory or independent values, more often referred to as input values and x-values, and a dependent variable, frequently referred to as the output value and y-value. A simple example for this would be the amount of rooms and the surface area of a house as x-values, and the price of this house as the y-value. These values serve as a training data for the algorithm, and are used to teach the program to make accurate predictions. These x-values and the y-value together are referred to as a data- or training set or simply 'the example data'. In order for the algorithm to work properly, the application of a great amount of data and thus a large dataset is crucial. The data corresponding to a single y-value will be referred to as a single training example.

Amount of rooms x-value 1	**Surface Area in m^2 x-value 2**	**House price in € y-value**
8	20	4000

Table 3.1: An example of a single training example

The algorithm will use this data to compute a hypothesis function using a process called gradient descent, which will attempt to minimize a cost function. What this means is that the linear regression algorithm will calculate a fit to the data set which is accurate to the data and not excessively complex, in order to be able to correctly predict a y-value for new, unlabelled x-values (x-values without a y-value). For example, considering this data set, which consists of one x-value and the y-value, the linear regression algorithm will compute the following graph:

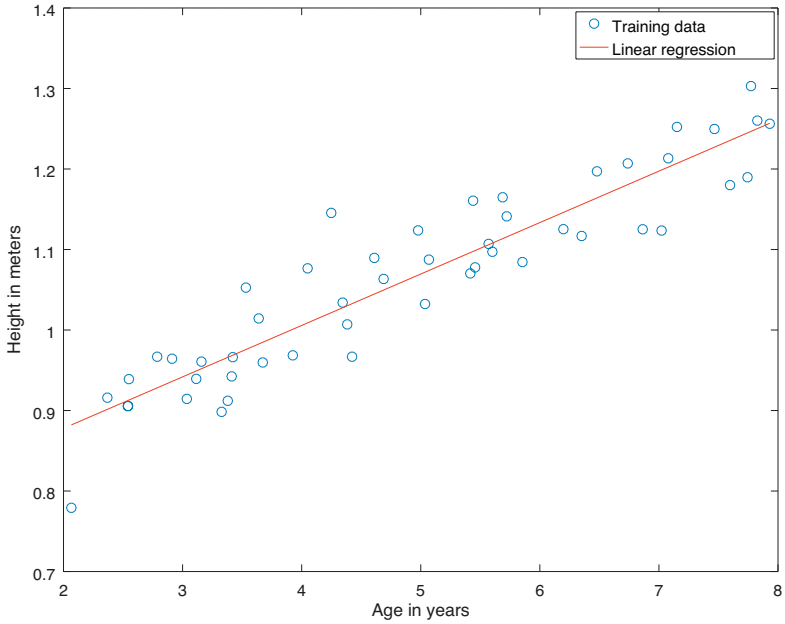


Figure 3.1: show graph with the added line

Note how the vector is a simplified average of the training data and can therefore be used to predict new y-values given unlabelled x-values.

In contrary to linear regression, were the dependent variable (or y-value) is a scaler, in logistic regression this variable is categorical, meaning it can only be one of a limited and usually fixed number of possible values. The most common application of this model is for binary dependent variables, were it can only take on a value of 0 or 1, or 'true' or 'false'. This concept is essential in the implementation of neural networks, and a more in depth explanation is provided in chapter 3.

3.1.2 Linear regression

3.1.2.1 Datasets

In order to gain an understanding of the mathematics of linear regression, it is useful to make use of an example dataset. For this example the relation between the age and height of various boys between the ages of two an eight.

Example m	**Age in years x-value**	**Height in meters y-value**
1	2.066	0.779
2	2.368	0.916
3	2.540	0.905
4	2.542	0.906
5	2.549	0.939
...	...	...
50	7.9306	1.2562

Table 3.1: A data set with 1 feature and 50 trainings examples

The x- and y-values will be denoted as $x^{(i)}$ and $y^{(i)}$ respectively, where i represents the number of the training example in the data set, ranging from 1 to m. A dataset accordingly can be seen as a list of m training examples $\{(x^{(i)}, y^{(i)}); i = 1, 2, \ldots, m\}$.[4] The variable i represents an index into the training set and is not an exponentiation.

This example considers only one x-value per dataset, coupled with one y-value. This is strictly for the explanation and demonstration of the basic machine learning concepts in this paper. Generally, there will always be more than one x-value, called **features** in this context, used, in order to encompass more complex data and compute more accurate and therefore useful predictions. In such scenarios the x-values are considered vectors in $\mathbb{R}^n$, where n is the number of x-values per training example. This, however, results in calculations and data of higher dimension, which are impossible to display in a clear and concise manner and are thus disadvantageous for the goals of this chapter. Keeping this in mind, the data of the example can be displayed more effectively and concise in a graph.

[4] Ng (n.d.).

3.1.2.2 The Hypothesis function

The goal of the algorithm will be to learn a function h that takes the given x-values in a training example to calculate a prediction for the y-value, where $h(x^{(i)}) \approx y^{(i)}$, meaning $h(x^{(i)})$ is a good as possible predictor of $y^{(i)}$. Traditionally, this function is called the hypothesis. Because the y-value is continuous, the learning problem of the example is called a regression problem.

The next step is defining the function h. The algorithm needs to factor in all the x-values for the prediction, while learning how to weigh each feature independently. Each x-value is therefore assigned to different variable, which the program will gradually adjust in order to increase the accuracy of the prediction. These parameters are noted as θ_a , where a is the number of the x-value it corresponds to. One extra parameter, θ_0, is also necessary to define h, as it functions as the starting position for the graph. The general function h will subsequently resemble:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$$

Where n is the amount of features.

Note that this function, when plotted with the x-values, will result in a linear graph (even in higher dimensions). Often, data will follow a different pattern, such as an exponential curve or a logistic curve. For this, the function can be altered accordingly to fit a specific relationship between the x-value and the y-value. For example, a term for an exponential relation might be $\theta_n(x_n)^2$, and a term for square root relation might be $\theta_n\sqrt{(x_n)}$. It is possible for the program to recognize these patterns itself, but the complex algorithms this requires extends beyond the scope of this paper.

When the first term, θ_0 , is considered to be $\theta_0 x_0$,with $x_0 = 1$, function h can be rewritten as:

$$h_\theta(x) = \sum_{i=0}^{n} \theta_i x_1$$

Where n is the amount of features.

The function for the example will simply be:

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 = \sum_{i=0}^{1} \theta_i x_i$$

3.1.2.3 The Cost function

In order for the program to update the parameters and increase the accuracy of the prediction, it needs to know the margin of error of its previous prediction. For this a square error term is used, which is defined as the difference between the two values squared. In this algorithm this function is referred to as the **cost function**. For a single training set it defines the cost function J:

$$J(\theta)^{(i)} = (h_\theta(x^{(i)}) - y^{(i)})^2$$

The cost function J for the whole training set can be defined as an average of the examples 1 through m. In addition, to simplify a forthcoming partial derivative, it is useful to multiply the function by $\frac{1}{2}$, for minimizing half of the error term will ultimately have the same result as minimizing the full error term:

$$J(\theta) = \frac{1}{2} \cdot \frac{1}{m} \cdot \sum_{i=0}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

The goal of the algorithm will be to minimize this function, through modification of the variables θ; more specifically in the case of the example, the variables θ_0 and θ_1. Relation between the h function and the θ variables can be analysed with help of the following graphs. Notice how different values of θ affect the hypothesis function and subsequently the cost function. The values of θ in graph 3.5 result in the lowest value of function J, indicating the function h to be a reasonable fit to the data.

3.1.2.4 Gradient descent

To achieve the goal of minimizing the cost function, a process called **gradient descent** is used. In principle, this entails that the parameters θ are updated in order to improve the hypothesis function. An important attribute of this process is that it is repeated a considerable number of times, wherein the parameters are updated only to a small extend each iteration. This step as well as other similar algorithms which serve the same objective are the reason why the great computational speeds computers provide are essential for all machine learning algorithms.

For reasons of this explanation, it is useful to visualize the cost function in a graph. The dependent variables of the function are the parameters θ. Because in the example these are comprised of θ_0 and θ_1, the figure is three dimensional.

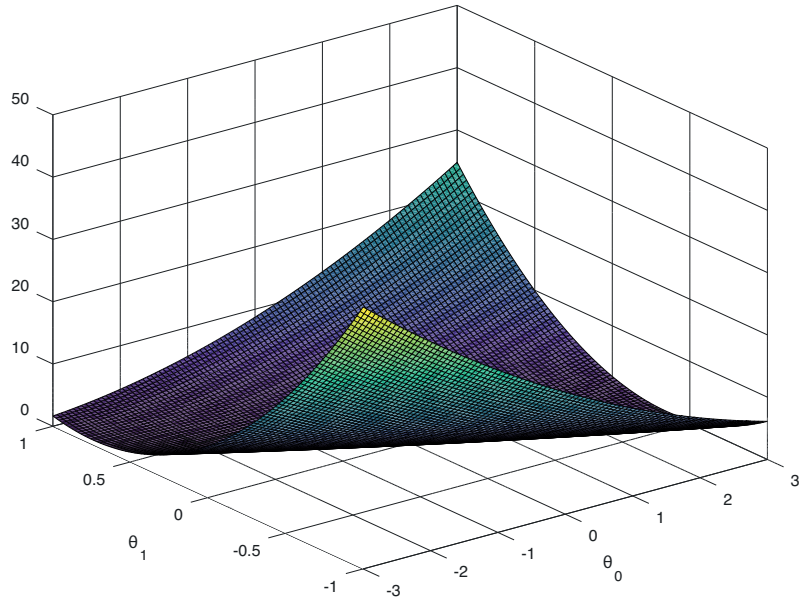


Figure 3.6:

As discussed previously, the goal of the algorithm is to minimize the cost function, which translates to finding a minimum in the plotted figure. As can be seen in figure 3.8, there won't always be just one minimum, but many different **local minima** can be present, while normally only one **global minimum** exists. The simple version of gradient descent in this introduction won't be able to distinguish between these. For now, however, this won't present any substantial problems. Additionally, a schematic rendering of the direction of the parameters and the cost function through the gradient descent algorithm is shown. Notice how each iteration reduces the outcome of the cost function, although in reality, the amount of iterations will be much larger instead of the 7 shown here.

Furthermore, to make finding the minimum for the example data set - which happens to only have one minimum - more apparent, it is beneficial to display the figure in a top-down view. From this figure, the "route" the parameters will elapse can be determined easily.

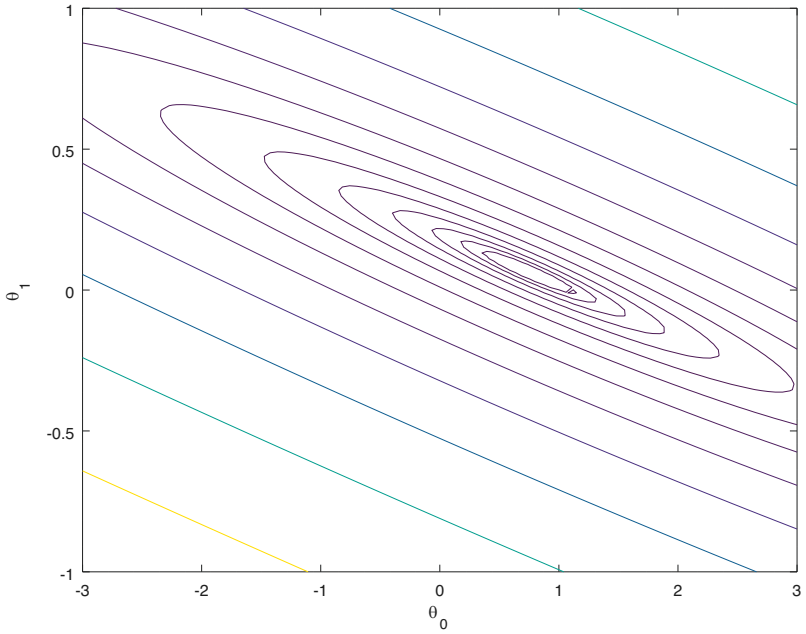


Figure 3.8: show graph with the added line

In order for the program to know which direction is towards a minimum, a **partial derivative** of the cost function is used. With this, the slope of the graph is calculated, in order for the program to know which direction is "uphill" and which direction is "downhill". Additionally, this will cause the parameters to gradually slow down their descent as they get closer to the minimum, where the slope is equal to zero. Note that in this process all features are computed separately. With this, the update function for each separate parameter θ for one iteration can be defined:

$$\theta_j := \theta_j - \alpha \cdot \frac{\partial}{(\partial \cdot \theta_j)} \cdot J(\theta) \quad (for\ all\ j)$$

Where j is the number of the feature, from 0 to the total amount of features
Where $\frac{\partial}{(\partial \cdot \theta_j)}$ indicates a partial derivative over $theta_j$
Where α is parameter indicating the learning rate.

The := symbol indicates an assignment computation. Because θ_j is used in the definition as well, this means the original value of θ_j will be updated in accordance to this original value.

Choosing a desirable learning rate α is an important part in this process, as it defines to which degree the parameters are updated each iteration. When it is too small, the program will take too long to compute, and when it is too large, the program might have trouble converging to a minimum, as the parameter will continually go over the it. Because all features are computed separately, it is possible to visualize this process.

The next step is to define $\frac{\partial}{(\partial \cdot \theta_j)} \cdot J(\theta)$. The inclusion of the previously defined cost function J gives:

$$\frac{\partial}{(\partial \cdot \theta_j)} \cdot \frac{1}{m} \cdot \frac{a}{m} \cdot \sum_{i=0}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

After differentiation, this formula becomes:

$$\frac{1}{m} \cdot \sum_{i=0}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

With these and as a final step, we can redefine the update formula:

$$\theta_j := \theta_j - \alpha \cdot \frac{1}{m} \cdot \sum_{i=0}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \quad (for\ each\ j)$$

3.1.2.5 Abridgement

In practice, the program first initializes the parameters θ to random values (often 0). Following this, it will compute the hypothesis function and update the parameters for i amount of iterations using the gradient descend algorithm with learning rate α, using the following formulas, until convergence. The outcome will be a hypothesis function which accurately predicts the y-value for new x-values.

$$h_\theta(x) = \sum_{i=0}^{n} \theta_i \cdot x_i$$

$$\theta_j := \theta_j - \alpha \cdot \frac{1}{m} \cdot \sum_{i=0}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \quad (for\ each\ j)$$

3.1.2.6 Vectorized implementation

For the machine learning to be as efficient as possible, the amount of calculations required for each iteration should be minimized. Especially for increasingly complex and large datasets, only a tiny decrease in the duration of a single iterations can lead to huge improvement for the whole process. An important aspect for essentially all machine learning algorithms is the vectorized implementation. Making use of a heavily optimized linear algebra library and matrix multiplications, an vectorized implementation creates a significant performance increase in any of the machine learning algorithms. For most programming languages, such a linear algebra library is easily available.

In basic terms, a vectorized implementation simply refers to an implementation where the computer calculates all learning examples 'at once', by positioning them in a single matrix. The hypothesis function will therefore be implemented as the product of two matrices, x^T and θ, instead of the sum of the products of two integers, θ_i and x_i, for all features and subsequently all training examples. Because of the exploitation of the matrix functionality, these computations are effectively identical. The matrix multiplication, however, is many times faster than a relatively slow for-loop and more compact in the programme as well. Furthermore, by arranging the data as a matrix, data sorting and storing can be done much more efficiently and orderly.

$$h_\theta(x) = \sum_{i=0}^{n} \theta_i \cdot x_i \quad (for\ all\ training\ examples) = x \cdot \theta$$

Note that the $\sum$ will be represented in code as a for-loop.It is important for the two matrices to be the appropriate proportion, in order to allow for the matrix multiplication.

The matrix x consists of all x-values of the training examples, and will be of $\mathbb{R}^{m \times n}$ dimensions, where m is the number of training examples and n the number of features (including x_0). Note the multiplication symbol expresses dimensionality, and does not indicate actual multiplication. The values for the weight values θ have just one value for each iteration, and are thus constant across the multiplications with all training examples in each iteration. θ will accordingly be of $\mathbb{R}^{n \times 1}$ dimensions. Following the multiplication, this computation will result in a matrix of dimensions $\mathbb{R}^{m \times 1}$ as well, containing the hypothesis value for all training examples.

$$\begin{array}{c|c} & \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix} \\ \hline \begin{bmatrix} x_0^{(1)} & x_1^{(1)} & x_2^{(1)} \\ x_0^{(2)} & x_1^{(2)} & x_2^{(2)} \\ \vdots & \vdots & \vdots \\ x_0^{(m)} & x_1^{(m)} & x_2^{(m)} \end{bmatrix} & \begin{bmatrix} x_0^{(1)} \cdot \theta_0 + x_1^{(1)} \cdot \theta_1 + x_2^{(1)} \cdot \theta_2 \\ x_0^{(2)} \cdot \theta_0 + x_1^{(2)} \cdot \theta_1 + x_2^{(2)} \cdot \theta_2 \\ \vdots \\ x_0^{(m)} \cdot \theta_0 + x_1^{(m)} \cdot \theta_1 + x_2^{(m)} \cdot \theta_2 \end{bmatrix} \end{array} = \begin{bmatrix} h_\theta^{(1)}(x) \\ h_\theta^{(2)}(x) \\ \vdots \\ h_\theta^{(m)}(x) \end{bmatrix}$$

Figure 3.10: The matrix multiplication table of the matrices x and θ, wherein the hypothesis values for all training examples are calculated. Following the standard rules for matrix multiplication, the first element of each row of the bottom left matrix is multiplied with the first element of each column of top right matrix, after which the product of the next elements of the respective row and column is added, until all each row has been included. In the bottom right matrix the result of this multiplication can be seen. This result is equal to the hypothesis function, confirming the validity of the use of the matrix multiplications of a vectorized implementation.

Accordingly, the cost function J can be updated for use of the the matrix h_θ, and a matrix y of the y-values of the training examples. Because, instead of with a one-by-one approach, all h-values are used simultaneously the sum can once again be omitted. The function can be rewritten as follows.

$$J(\theta) = \frac{1}{m} \cdot \sum_{i=0}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 = \frac{1}{m} \cdot (h_\theta - y)^2$$

Note the exponent 2 illustrates an element-wise exponentiation.

Lastly, since the the weight values, θ_0 to θ_n, have been converted to a matrix form as well, the weight update function must be updated similarly. It is once again possible to exploit the functionality of matrix multiplication for a more efficient computation, as the x-value present in the update function can be used to conveniently create a matrix with the corresponding dimensions.

$$\begin{array}{c|c} & \begin{bmatrix} h_\theta^{(1)}(x) \\ h_\theta^{(2)}(x) \\ \vdots \\ h_\theta^{(m)}(x) \end{bmatrix} \\ \hline \begin{bmatrix} x_0^{(1)} & x_0^{(2)} & \dots & x_0^{(m)} \\ x_1^{(1)} & x_1^{(2)} & \dots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & \dots & x_2^{(m)} \end{bmatrix} & \begin{bmatrix} \Delta\theta_0 \\ \Delta\theta_1 \\ \Delta\theta_2 \end{bmatrix} \end{array}$$

$$\theta_j := \theta_j - \alpha \cdot \frac{1}{m} \cdot \sum_{i=0}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \quad (for\ each\ j) \longrightarrow \theta =: \theta - \alpha \cdot (\frac{1}{m} \cdot x^T \cdot h_\theta)$$

One dimensional integers α and $frac1m$ are multiplied element-wise with the according matrices. The T in this formula represents the transpose of the matrix x. Because of standard matrix multiplication notation, the transposed matrix x^T is positioned before the h_θ term in the formula. With this, the complete learning algorithm can be represented with the following formula:

$$\theta := \theta - \alpha \cdot (\frac{1}{m} \cdot x^T \cdot ((x * \theta) - y)^T)$$

3.1.2.7 Sidenotes

While more data usually means more accuracy, because it is often possible that no correlation between the x and y values exists, all x-values should always be carefully considered and analyzed however. Even when no correlation exists, the algorithm will still factor the data in the calculations, which will result in less accurate results.

This paper won't go into detail of many other, more complex aspects of linear regression, on account of space constraints. If a more in depth understanding is required, it is recommended to read up on these concepts, such as feature scaling and the selection of a good learning rate to name a few, from other sources.

Furthermore, it is important to realize these forms of machine learning, linear and logistic regression, are relatively basic and are not used a lot nowadays in their original form. Many more complex and efficient algorithms have since taken their place. These algorithms are still based on the same concept however, and basic understanding of these concepts is crucial in the field. Neural networks are an example of this, since they are essentially a variation on logistic regression.

3.2 Implementation in the programming language Octave

3.2.1 Introduction and overview

The implementation of the linear regression algorithm is relatively simple, and only consists of a few lines. This will be a step by step commentation for a basic implementation as described in chapter 3.1 of this paper. As mentioned previously, the used programming language is GNU octave, but any programming language is of course suitable, provided it has an linear algebra library available. Octave comes pre installed with a greatly optimized linear algebra library and provides build-in functionality for graphs and other useful figures as well. For this programme the previously described example of chapter 3.1 is used once again.

3.2.2 Non-vectorized implementation in Octave

Before examining vectorized implementation, it is important to understand a basic implementation. Keep in mind a non-vectorized implementation is rarely used anymore and can make for a somewhat disconcerting programme.

The first step is loading the training examples and defining variables. The dataset contains 50 examples, meaning the x and y values, which are defined as an array, will have a length of 50. This example considers the data to be available as an array in a *.dat* file, although the initiating of this data depends on the manner in which it is stored on the computer. The x_0 is defined as 1.

Next, the learning rate α (alpha) and the amount of iterations i (max_iterations) are given the values 0.07 and 1500 respectively. Both values are changeable, and optimal values differ per situation. Unfortunately, the techniques for choosing the best values for these will not be discussed in this paper. For now, realize a smaller value of α will give the algorithm more accuracy, at the cost of speed. The amount of iterations is the amount of times gradient descend will run, and a higher value ensures the hypothesis function has converged to an optimal value, at the cost of the programme running for a greater amount of time.

The starting values for the weights θ are initiated as well, and are given a value of 0, although any value could be used. As a reminder, the example dataset contains only one feature, meaning 2 different θ values need to be initiated, θ_0 (theta0) and θ_1 (theta1).

```
%Inport Training Examples
x0 = 1;
x1 = load('x.dat');
y = load('y.dat');

%Set variables
m = length(y);
alpha = 0.07;
max_iterations = 1500;
theta0 = 0;
theta1 = 0;
```

The next step is defining the hypothesis function. From the general formula, $h_\theta(x) = \theta_0 \cdot x_0 + \theta_1 \cdot x_1 \ldots \theta_n \cdot x_n$, follows for the example the formula $h_\theta(x) = \theta_0 \cdot x_0 + \theta_1 \cdot x_1$. Because each training example has a hypothesis value, the function must be computed m amount of times. This is achieved by use of a for-loop, in which the hypothesis function

is defined as the array h[i].

```
%The hypothesis function
for i = 1:m
  h[i] = theta0 * x0 + theta1 * x1[i];
end
```

The cost function will be defined as the already partially differentiated formula $\frac{1}{m} \cdot \sum_{i=0}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$. For the sum, a for loop can be used, which goes over all the index values of $h[i]$ and adds them together. The $\frac{1}{m}$ is added in the next step, in order to correctly calculate the outcome. Because the formula has a weight parameter specific term ($x_j^{(i)}$), a separate cost function for each feature is necessary. In the programme, this will be defined as follows.

```
%The cost function
J0 = 0;
J1 = 0;
for i = 1:m
  J0 = J0 + (h[i] - y[i]) * x0;
  J1 = J1 + (h[i] - y[i]) * x1[i];
end
```

Lastly, the weight parameters are updated according to the the update function $\theta_j := \theta_j - \alpha \cdot \frac{1}{m} \cdot \sum_{i=0}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \quad (for\ each\ j)$, in which $\sum_{i=0}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$ is replaced with the in the programme defined J. Keep in mind this variable is different than the in section 3.1 defined cost function J. The parameters theta0 and theta1 will be updated as follows.

```
%The update function
theta0 = theta0 - alpha * 1/m * J0;
theta1 = theta1 - alpha * 1/m * J1;
```

The described programme functions as a single iteration of the gradient descend algorithm. In order to finish the full linear regression algorithm, a for-loop is added which encompasses the defined functions. Because for each iteration a new error term is necessary, the variables J0 and J1 need to be reset. The full programme will look as follows.

```
%Inport Training Examples
x0 = 1;
x1 = load('x.dat');
y = load('y.dat');

%Set variables
m = length(y);
alpha = 0.07;
max_iterations = 1500;
theta0 = 0;
theta1 = 0;

%Linear Regression
for i = 0:max_iterations
  for i = 1:m
    h[i] = theta0 * x0 + theta1 * x1[i];
  end
```

```
18
19    J0 = 0;
20    J1 = 0;
21    for i = 1:m
22      J0 = J0 + (h[i] - y[i]) * x0;
23      J1 = J1 + (h[i] - y[i]) * x1[i];
24    end
25
26    theta0 = theta0 - alpha * 1/m * J0;
27    theta1 = theta1 - alpha * 1/m * J1;
28  end
```

3.2.3 Vectorized implementation in Octave

This basic implementation presents a number of problems. For instance, a high number of features can lead to an incredible amount of theta variables, resulting in a high number of functions or, alternatively, nested for-loops, both of which decrease the clarity and increase inefficiency of the code. A vectorized implementation will solve these problems as well as improve performance significantly, as has been described in section 3.1. Additionally, because of the nature of this implementation, the programme is compatible with all possible number of features.

To start, the x, y and h values previously defined as arrays, can be converted to matrices. The variables y and h both contain one value for each training example, resulting in matrices dimensions $\mathbb{R}^{m\times 1}$, and $\mathbb{R}^{50\times 1}$ for the example. The matrix x includes all x values, and will accordingly have a width of $n+1$ (all features and x_0), resulting in a matrix of $\mathbb{R}^{m\times n}$, and $\mathbb{R}^{50\times 2}$ for the example. The first column is made up of the x_0 values, and all its elements equal one. This column won't be part of any dataset, and needs to be added to the matrix. This can be achieved with an Octave function *ones*(), which will return an matrix of ones of the specified dimensions.

```
1 %Inport Training Examples
2 x = load('x.dat');
3 y = load('y.dat');
4
5 %Set variables vectorized
6 m = length(y);
7 x = [ones(m, 1), x];
```

For each feature, one weight parameter θ_n needs to be defined. As described in section 3.1, this is achieved in a matrix of dimensions $\mathbb{R}^{n\times 1}$, and $\mathbb{R}^{2\times 1}$ for the example. Once again, the values of θ (theta) are initiated as zero. Because the height of the matrix needs to equal the width of the matrix x, the Octave function *width*() (which, as expected, returns the width of a matrix) is used followed by a transpose. This gives a matrix of the right dimensions. With the Octave function *zeros*(), all elements of the matrix are set to zero.

```
1 theta = zeros(size(x(1,:)))';
```

Next, the hypothesis function can be defined. All h values are computed at once, and no for-loop is necessary.

```
1 %The hypothesis function
```

```
h = x * theta;
```

The cost function is adjusted in a similar manner, and computes all values at once. Additionally, the different values for J can be represented in one function. A matrix multiplication with x^T will result in a matrix of the same dimension as the matrix theta, namely $\mathbb{R}^{n\times 1}$, and $\mathbb{R}^{2\times 1}$ for the example. In order to correctly execute this multiplication, the matrix x^T needs to be positioned before the matrix h. In Octave, a transpose is implemented with use of an apostrophe, resulting in x^T being represented as x' in the programme. For consistency, the $\frac{1}{m}$ term is again moved to the update function.

```
%The cost function
J = x' * (h - y)
```

Subsequently, the update function updates all weight parameters θ, present in different rows of the matrix theta, at once. The integers α (alpha) and $\frac{1}{m}$ are multiplied element-wise with the matrix.

```
%The update function
theta = theta - alpha * (1/m) * J;
```

Once again a for-loop is added to finish the linear regression algorithm.

```
%Linear regression vectorized implementation
for i = 1:max_iterations
  h = x * theta;
  J = x' * (h - y);
  theta = theta - alpha * (1/m) * J;
end
```

As a final step, the functions can be combined to shorten the whole algorithm to a single line.

```
%Inport Training Examples
x = load('x.dat');
y = load('y.dat');

%Set variables vectorized
m = length(y);
x = [ones(m, 1), x];
alpha = 0.07;
max_iterations = 1500;
theta = zeros(size(x(1,:)))';

%Linear regression vectorized and shortened implementation
for i = 1:max_iterations
  theta = theta - alpha * ((1/m) * x' * ((x * theta) - y));
end
```

Even for a linear regression algorithm with only two features, the vectorized implementation manages to shorten and optimize the programme by a considerable amount. This example served only as an easy introduction to the idea. The concepts of vectorized implementation play a big role in the implementation of neural networks and will return in the following chapter.

4 Neural networks

4.1 Mathematical theory and model

- Unsupervised versus supervised learning
- Full explanation of the supervised neural network
- formulas and alpha
- Explanation vectorized implementation
- Autoencoders

4.2 Implementation in the programming language Octave

- Step by step (First without vectorization.)
- Step by step with vectorized implementation
- Determining the amount of iterations and the value for alpha and labda.

4.3 Alternative algorithms to gradient descent

- Short explanation without a lot of details

5 Evolving neural networks and self-learning artificial intelligence

5.1 NEAT - Self improving neural networks

neural networks are brains, etc. etc. etc.

5.2 The application of neural network based artificial intelligence to video games

- Zeer belangrijk gedeelte. Hier leggen we het punt van het onderzoek eigenlijk uit, en waarom dit zou werken en waarom het goed is dat het zou werken
- Uitwerking
- theory

Andere punten:

- Which games is is applicable to?

5.3 Implementation in the programming language Python

6 Experiment: The application of neural network based artificial intelligence to video games

6.1 Choosing an emulator and transferring the program to the appropriate language

6.2 The set-up and time partition of the experiment

6.3 [after the experiment] Analysis of the data

- Did it work?
- How long did it take?
- Application on other levels and games
- How can the program be improved

7 Conclusion

8 Definitions

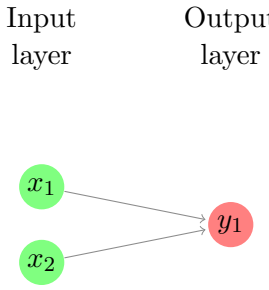
References

Bayes, T. (1763). An essay towards solving a problem in the doctrine of chances. *Phil. Trans.*, *53*(0), 370-418. Retrieved from `http://www.stat.ucla.edu/history/essay.pdf` doi: 10.1098/rstl.1763.0053

Ng, A. (n.d.). *Cs229 lecture notes - supervised learning.* Retrieved from `http://cs229.stanford.edu/materials.html`

Stigler, S. M. (1982). Thomas Bayes' Bayesian inference. *Journal of the Royal Statistical Society*, *Series A*(145), 250-258. Retrieved from `https://bit.ly/stiglerbayes`

Turing, A. M. (1950). Computing machinery and intelligence. *Mind*(49), 433-460. Retrieved from `https://www.csee.umbc.edu/courses/471/papers/turing.pdf`


Input
layer
Output
layer
x_1
x_2
y_1

Listing 1: Data definitions (nnrv.m)

```
W_1 = stdnormal_rnd(2,3);
W_2 = stdnormal_rnd(3,1);
b1 = stdnormal_rnd(1,3);
b2 = stdnormal_rnd(1,1);

data_x1 = [1; 0; 1; 1; 0; 0; 0; 1; 1; 0; 1; 0; 0; 0];
data_x2 = [0; 1; 0; 1; 0; 0; 1; 1; 1; 0; 0; 1; 1; 0];

data_y = [0; 0; 0; 1; 0; 0; 0; 1; 1; 0; 0; 0; 0; 0];
```

Listing 2: Implementation of the XNOR gate in a neural network in the Ocatve language (nnt1.m)

```
% An implementation of the XNOR-gate using a neural network
% Uses data values defined in nnrv.m
%
% by Maxim van den Berg, Sam van Kampen, Robin Wacanno and Wilco Stam

alpha = 1;
lambda = 0;
m = rows(data_x1);

function s = f(z)
  s = 1./(1 + exp(-z));
end


%
disp("W_2 (begin) =");
disp(W_2);
disp("W_1 (begin) =");
disp(W_1);
disp("b2 (begin) =");
disp(b2);
disp("b1 (begin) =");
disp(b1);
%Compute for display starting values
a1 = [data_x1, data_x2];
z2 = a1 * W_1 + b1;
a2 = f(z2);
z3 = a2 * W_2 + b2;
h = f(z3);
a3 = h;
disp("h (begin) =");
disp(h);

loops = 0;

deltas = [];

do
  loops += 1;
  % Compute the current output of the neural net
  a1 = [data_x1, data_x2];
```

```
42
43    z2 = a1 * W_1 + b1;
44    a2 = f(z2);
45
46    z3 = a2 * W_2 + b2;
47    h = f(z3);
48    a3 = h;
49
50    % Start computing deltas for backprop
51
52
53    delta3 = -(data_y - h)   .* (a3 .* (1 - a3)); % f'(z) = f(z)(1 - f(z)). See
          slide: http://i.imgur.com/K5QHTK9.png
54    delta2 = (delta3 * W_2') .* (a2 .* (1 - a2)); % Hoe zit het met de sum? to be
          continued als we meer layers toevoegen
55
56    gradW_2 = a2' * delta3;
57    gradW_1 = a1' * delta2;
58    gradb2 = sum(delta3);
59    gradb1 = sum(delta2);
60
61
62    W_2 = W_2 - alpha * (gradW_2 * 1/m) + lambda * W_2;
63    W_1 = W_1 - alpha * (gradW_1 * 1/m) + lambda * W_1;
64
65    b2 = b2 - alpha * (gradb2 * 1/m);
66    b1 = b1 - alpha * (gradb1 * 1/m);
67
68    deltas = [deltas mean(abs(gradW_2 * 1./m))];
69  until (mean(abs(delta3)) < 1e-4);
70
71
72  disp("h =");
73  disp(h);
74
75  disp("Number of required loops: ");
76  disp(loops);
77
78  disp("Rounded H, compared to data_y");
79  disp([round(h) data_y]);
80
81  disp("J / cost =");
82  disp(delta3);
83
84  disp("W_2 =");
85  disp(W_2);
86  disp("W_1 =");
87  disp(W_1);
88
89  disp("b2 =");
90  disp(b2);
91  disp("b1 =");
92  disp(b1);
```