

GAME FARS

THE AI WHICH CAN PLAY GAMES

KANTONSSCHULE WOHLLEN
MATURA-PROJECT

Creating an AI which can play games

Brian Funk and Silvan Metzker

supervised by
Patric ROUSSELOT and Mark HEINZ

26. Oktober, 2020

Contents

1	Abstract	3
2	Introduction	3
3	Basics of Artificial Neuronal Intelligence	4
3.1	What is an Artificial Intelligence?	4
3.2	Mathematics Behind an Artificial Neuronal Network	5
3.2.1	Neuronal Network	5
3.2.2	Learning	9
3.2.3	Example	11
4	Reinforcement Learning	19
4.1	Introduction to Reinforcement Learning	20
4.2	Deep Q-Networks (DQNs)	21
4.2.1	Introduction to Q-Learning	21
4.2.2	Applying the Basics of Deep Learning	21
5	Implementation	22
5.1	Python	22
5.2	TensorFlow	22
5.3	NumPy	23
5.4	Matplotlib	23
5.5	Pygame	23
5.6	Variables	24
5.6.1	Gamma $[\gamma]$	24
5.6.2	Epsilon $[\epsilon]$ and Decay	24
5.6.3	Number of Episodes $[N]$	25
5.6.4	Alpha $[\alpha]$	25
5.6.5	Experience and Copy Step	25
5.6.6	Batch Size	26
6	Basic Structure of Code	26
7	The Model	27
7.1	The Class MyModel	27
7.2	The Class DQN	29

8 Graphical User Interface	33
9 Games	35
9.1 Tic-tac-toe	35
9.1.1 Rules	35
9.1.2 Implementation	36
9.1.3 Interaction with the AI	36
9.2 Snake	37
9.2.1 Rules	37
9.2.2 Implementation	38
9.2.3 Interaction with the AI	39
9.3 Space Invaders	42
9.3.1 Rules	42
9.3.2 Implementation	43
9.3.3 Interaction with the AI	44
10 Results	46
10.1 Tic-tac-toe	46
10.1.1 Expected Random Win Rate	46
10.1.2 Achieved Win Rate	49
10.2 Snake	51
10.2.1 Achieved Score	52
10.3 Space Invaders	53
10.3.1 Achieved Score	53
11 Discussion	54
Glossary	56

1 Abstract

The term *artificial intelligence* (*AI*) is widely used. Many know this wording in the context of a science fiction movie or when it comes to data analysis. Google uses AI to provide their customers with the optimal advertisement. It works, but hardly anyone can imagine how it works. By programming and testing such an intelligence, we try to understand how exactly this can work. How well can an AI perform the tasks given? Where will we face limitations? These and many other questions are answered in this report.

We succeeded in implementing an AI in various games. But it doesn't work equally well for all of them. We encountered the limits of intelligent retrieval sooner than we thought. But one of the goals of this project is to simply understand the technology. Even though the products are not perfect, we gained a lot of knowledge and thus achieved our main goal.

2 Introduction

The goal of this project is to create an AI that can play games. This is achieved by utilizing a practice called *machine learning*. Machine learning is defined as a technique that can learn by using *computational power*. It's a generic term mainly used to describe AIs which learn by analyzing huge amounts of data. The most human-like strategy in machine learning is *deep learning*. It's structure is similar to a human brain and does surprisingly well at mimicking the brain's abilities to learn.

This report is structured in the same way, we encountered the problems. First, the theoretical foundation has to be laid out. This will be done in chapter 3. To make it more tangible, this theoretical part will be backed up by a real-life example. There are many different ways to perform machine learning. Those range from just taking values out of a table, to a methodology similar to a human brain. To decide which type of machine learning to use, a clear difference has to be made. This will be discussed in chapter 4. As soon as it was clear, which type will be used, it had to be implemented. How this exactly happened, is stated in the implementation chapter. To gain some insights, the subsequent chapter shows some basic coding.

The whole project can now be split into three parts:

- Artificial intelligence
- Graphics
- Games

Each part will be discussed in chapters 7, 8 and 9. What this implementation yields and how it performs with each game is recorded in chapter 10. The games are increasingly complex, to test the capability of handling complex information.

3 Basics of Artificial Neuronal Intelligence

The upcoming section will explain, what is necessary in order to understand how an AI works. The first part will show what types of AI there are and which is being used in this project. The second part will explain in detail, how the mathematics behind neuronal networks work.

3.1 What is an Artificial Intelligence?

An AI is the ability of a digital device to execute tasks, which are related to human beings and animals. This vague description can be taken in many ways, so a closer description would be an *artificial narrow intelligence*. [1] Artificial narrow intelligence (ANI) is a type of AI, which can only be applied to one narrow task. This is the kind of AI that currently exists. It can do a task very well, even better than humans. For example an AI, can detect brain tumors way more accurately than an experienced neurosurgeon is expected to do. It's revolutionary and does very well at one certain task, however, it cannot tell the difference between a cat and a dog. This is considered an artificial narrow intelligence. An *artificial general intelligence* (AGI) on the other hand, is an AI that can perform all tasks that a human can fulfill. This type of AI is not yet discovered. The step after this, would be an *artificial super intelligence* (ASI), this is the kind of AI that is superior in every way to a human. It's also considered as the type of AI which could lead to the extinction of the human race. The closest approach to one of those higher levels of an AI was achieved by one of the so-called artificial *neuronal networks* (ANNs). [2]

3.2 Mathematics Behind an Artificial Neuronal Network

This chapter will explain in detail how neuronal networks mathematically function. This basic knowledge is important if one wants to understand the outputs of a neuronal network and how they are computed.

3.2.1 Neuronal Network

A model that is used for many self-learning applications is called artificial neuronal network or just neuronal network. This model mimics the human brain and tries to describe the learning-behaviour as accurately as possible. Although it has biological foundations, it can be described in a purely mathematical way. To understand this, certain terms have to be defined mathematically and linguistically. Similarly to the brain, a *neuron* takes multiple *inputs* and passes an output forward, which is somehow dependant on the input. The human brain uses an electrical currents to transfer information. The AI does this numerically. So a number gets passed from one neuron to another. The inputs of a neuron are multiplied by a certain factor. Later in the learning process, these factors are the values, that are going to be optimized and changed. By increasing or decreasing these values, one node can either have more or less impact. If a junction has the *weight* of 0, the incoming number doesn't affect the network at all. The neuron takes all the inputs and adds them together. This sum gets forwarded to the *activation function*, which calculates the value according to the function. This transformed value gets forwarded to the next neuron and the whole process starts again. Two neurons can make a connection, which is used to pass the numerical *outputs* of the function to the next neuron. These neurons build up *neuronal layers* and determine its size by the amount of neurons. One of the properties of the neuronal layers is, that all neurons from one neuronal layer don't connect to each other. But they do connect to the previous and subsequent neuronal layers. The neuronal layers can be categorized in three different ways: input, output and *hidden layers*. The hidden neuronal layers describe all neuronal layers, except the in- and *output layers*. [3]

The neuronal network consists of all parts mentioned before, as can be seen in Figure 1. Every circle or ellipse represents one neuron. The first neuronal layer is the *input layer* and has as many nodes, as the *input layer size* n .

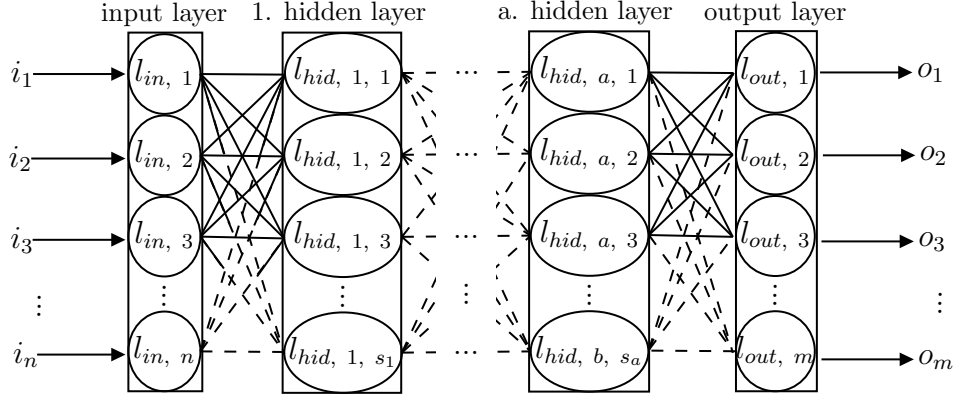


Figure 1: The relation between the different layers.

This means if you want to forward binary information of four variables, the input size and therefore the input layer would be in the size of four neurons. This also holds for the *output layer size*, which is denoted by m . The *amount of hidden layers* a and its size s can be chosen arbitrarily. Although there are no constraints for the hidden layers, the amount and size can hugely affect the efficiency and capability to learn. In Figure 2, the structure of a neuron

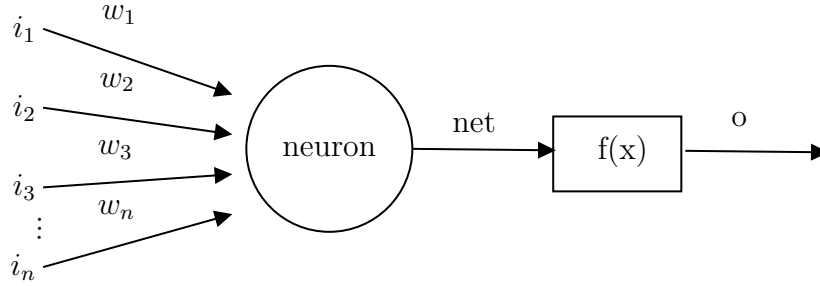


Figure 2: A visualization of a neuron.

can be seen. The abbreviation i stands for input, w refers to weight and o to the output of the neuron. The inputs and weights can be described in the form of a *vector*. The output o gets weighted accordingly and functions as input for the following neurons. Each component represents one input or

weight. This yields the vectors \vec{i} and \vec{w} .

$$\vec{i} = \begin{pmatrix} i_1 \\ i_2 \\ \dots \\ i_n \end{pmatrix} \quad \vec{w} = \begin{pmatrix} w_1 \\ w_2 \\ \dots \\ w_n \end{pmatrix} \quad (1)$$

The sum, in figure 2 referenced as *net*, is determined by the weighted summation of all the inputs.

$$\sum_{x=1}^n i_x \cdot w_x \quad (2)$$

The equation 2 shows the *net value* in the form of a sum. Due to the properties of the *scalar product*, this sum can be rewritten in the form of this vector operation:

$$net = \vec{i} \cdot \vec{w} \quad (3)$$

This weighted sum gets forwarded to the activation function. By replacing the net value with the definition stated in equation 3 and $f(x)$ being the activation function, the output is defined as follows:

$$\begin{aligned} o &= f(net) \\ o &= f(\vec{i} \cdot \vec{w}) \end{aligned} \quad (4)$$

The final output, or an intermediate result, of a hidden layer can also be described in a vector form.

$$\begin{pmatrix} o_1 \\ o_2 \\ \dots \\ o_m \end{pmatrix} = \begin{pmatrix} f(net_1) \\ f(net_2) \\ \dots \\ f(net_m) \end{pmatrix} \quad (5)$$

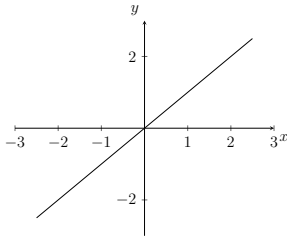
As one can see in formula 3, the net value can also be described in a vector form. By expanding the stated definition, it can be formulated such that multiple values are defined in one vector. Although it yields the same result, the output vector \vec{o} can be calculated easier, when using net values in a vector form.

$$\begin{pmatrix} net_1 \\ net_2 \\ \dots \\ net_m \end{pmatrix} = \begin{pmatrix} w_{1,1} & w_{2,1} & \dots & w_{n,1} \\ w_{1,2} & w_{2,2} & \dots & w_{n,2} \\ \dots & \dots & \dots & \dots \\ w_{1,s} & w_{2,s} & \dots & w_{n,s} \end{pmatrix} \cdot \begin{pmatrix} i_1 \\ i_2 \\ \dots \\ i_n \end{pmatrix} \quad (6)$$

Formula 6 is a stacked up version of equation 3. Every row represents all the weights from one specific neuron. The output of this neuron is used as input in the next one. It acts as output as well as an input, but nevertheless, it is denoted by i . A noticeable remark is, that it fits with the definition of the scalar product, which requires the dimension of the first vector to be the same as the dimension of the second vector. Which is the case, as it can be seen. The letter s denotes the *layer size* of the layer, from which the output gets calculated of. However, there are different kinds of neurons, depending on their activation functions. Depending on the context, different functions are more efficient to use. The most common and popular types of such are: [3]

Linear

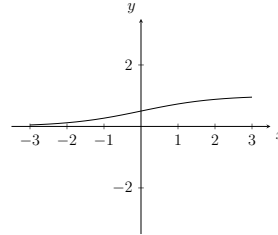
$$f(x) = x$$



Mostly used in input layer.

Sigmoid

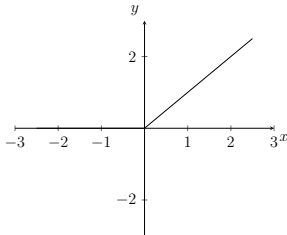
$$f(x) = \frac{1}{1 + e^{-x}}$$



Mostly used in hidden layers.

Rectified linear unit (ReLU)

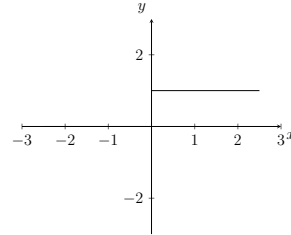
$$f(x) \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$$



Mostly used in hidden layers.

Binary step

$$f(x) \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$



Mostly used in output layers.

The derivative of these functions is important to make a learning process possible. It is used for the *backpropagation*. Therefore *linear function* and *binary step function* are not really used in the hidden layers. Although the *sigmoid neuron* is biologically more plausible, the *Rectified linear unit (ReLU)* functions have shown to be more efficient for hidden layers. [4] Sigmoid yields the property of always being finite whereas, Rectified linear unit (ReLU) diverges into infinity. Additionally, the easily calculable derivative of Rectified linear unit (ReLU) tends to converge faster to the wanted value.

3.2.2 Learning

To learn, the neuronal network gets optimized by changing the weights w . To change these values appropriately for the task given, a goal has to be set. This goal is important to calculate how much the actual output differs from the wanted results. It is denoted by t , which stands for *target*. The neuronal network can be compared to a function, which converts n inputs to m outputs.

$$f(x_1, x_2, x_3, \dots, x_n) \rightarrow (y_1, y_2, y_3, \dots, y_m) \quad (7)$$

With the data given, the network compares the yielded output values to the target values by calculating the error. A similar formula is used to calculate the variance in statistics.

$$E = \frac{1}{m} \sum_{i=1}^m (y_i - t_i)^2 \quad (8)$$

To make backpropagation possible, two of the three values (input, output and target) have to be constant. When the input vector \vec{i} and the target \vec{t} are fixed, the only variable is \vec{w} . This can be used to analyze how much the output differs from the target for given weights.

Reformulating yields a function, which is only dependant on \vec{w} . By plugging this formula into the equation, which assigns the numerical error, it can be optimized to minimize the error. The function is denoted by:

$$E_{\vec{i}\vec{t}}(\vec{w}) \quad (9)$$

The *optimization* happens due to the adjustment of the weights. As a result of the dependency, the outputs change with them. Which means by changing

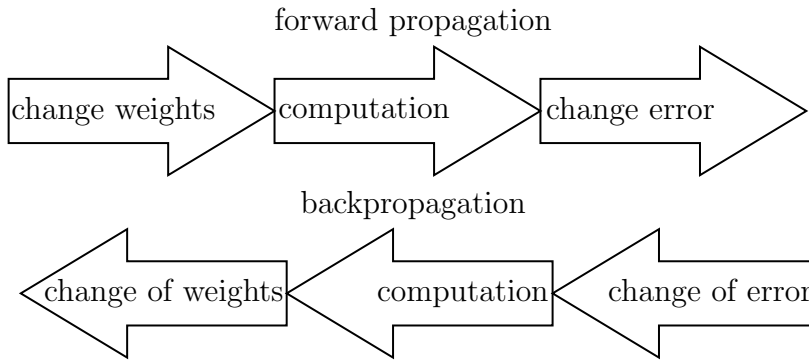


Figure 3: The difference between forward and back propagation.

\vec{w} , we can achieve a *convergence* or *divergence* of the target values. So how does this happen?

The forward propagation determines the change of the error when changing the weights. But when the weights should be optimized, backpropagation is used. As can be seen in Figure 3, the data gets passed in the opposite direction. So now, one can use the change of the error to get them according to the change of weights. This adjustment has to be made to yield a specific error.

The network should be doing better after this process, so the error should get smaller. In general, it does not matter how fast this value gets smaller. The function E is multidimensional, which allocates the error to the input and weights. To be optimized, the *gradient descent* can be used. This function is denoted by ∇f . The gradient descent of a more-dimensional function describes its steepest ascent. [5]

In this context the gradient descent always points in the direction, where the error increases the most. This would lead to a divergence of the outputs and the targets. So the negative gradient descent can be used to find the direction, in which the error decreases the fastest. Adding a multiple of the negative gradient descent and the old weights together, the resulting weights are ensured to be closer to the target value than the old one. But this only holds when the factor, with which the gradient descent gets multiplied with, is not too large. In the worst-case scenario, the new \vec{w} does not perform better, because it overshoots the optimal point. Another unlucky case would be if the greatest descent of the function would lead to a local *minimum*, which

would achieve worse results than other minima. This can be prevented by fully optimizing the function multiple times with random weights. To complete the learning process, the optimization has to be executed repeatedly until the outputs approximates the targets sufficiently. Mathematically this step can be defined as follows:

$$w_{new} = w_{old} - \alpha \cdot \nabla fE(w_{old}) \quad (10)$$

The constant α is called *learning rate* and defines how fast this approximation should happen. By choosing a large value, the trade-off for the time gained is the diminishing accuracy. The target values can be calculated in different ways which have their advantages and disadvantages.

For example, an AI, which looks for patterns in pictures and has to state if the picture shows something specific. The target then is defined by the data to be recognized, in our case an object. Therefore you would label all pictures according to which object is seen in the picture. The neuronal network improves the ability to recognize this specific object or pattern with every picture shown, by comparing the actual label to the output of the neuronal network. An example would be, that it should recognize a dog in a picture. The network gets all the pixels of the pictures and the labels as inputs. It should output 'dog' or 'not a dog'. By converting this label to a 0 for 'not a dog' and 1 for a 'dog', the numerical value can be used as the target value. By repeatedly showing pictures and recreating the optimization process stated in equation 10, the network recognizes a dog increasingly better.

3.2.3 Example

Let's create an AI, which can solve a simple task.[6] By doing so, the behaviour of a simple neuronal network can be demonstrated. The task looks like the following. The input comes in the form of a list with three elements. Every element is determined by the number 0 or 1. The output the AI should return, is one of the three numbers. Which number is determined by a . If, for example, $a = 1$, then only the first number in the list is significant. So the output should be the same number as in the first element of the list. A few examples will be given below to demonstrate the output for each input.

$a = 1$	$a = 2$	$a = 3$
$[1, 1, 0] \mapsto 1$	$[1, 1, 0] \mapsto 1$	$[1, 1, 0] \mapsto 0$
$[0, 1, 1] \mapsto 0$	$[0, 1, 1] \mapsto 1$	$[0, 1, 1] \mapsto 1$
$[1, 0, 1] \mapsto 1$	$[1, 0, 1] \mapsto 0$	$[1, 0, 1] \mapsto 1$
$[0, 1, 0] \mapsto 0$	$[0, 1, 0] \mapsto 1$	$[0, 1, 0] \mapsto 0$

The network will receive the value exactly in the form shown above. So, for instance the list $[1, 1, 0]$ and its goal $\mapsto 1$. It is not clear, if this task can be solved with just one neuron. But to keep it simple, the first attempt will just have one neuron in the hidden layer. The input layer will have the same size as the input itself. Which is three, hence three input numbers. The activation function is linear. This means the input won't be transformed. For this reason, we can assume there is no activation function for this layer. The same counts for the output layer. The goal is, to get one number returned, so the size will be one.

The network has in total four connections and five nodes. The three connections between the input layer and hidden layer have one weight each. In comparison the human brain has 10^{15} connections, which are called synapses. The amount of neurons in a human brain has been estimated to something around 10^{11} . So the neuronal network is primitive. Due to its simplicity the whole network can be sketched as it is shown below in fig 4.

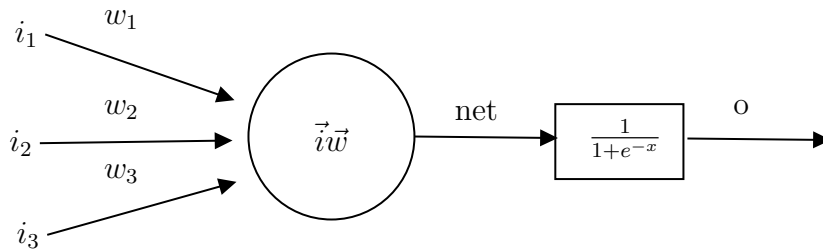


Figure 4: The structure of the neuronal network. It contains three input layer nodes, one hidden layer node and one output layer node.

The three inputs and weights can be written in the vector form. This makes calculations with many inputs easier, due to the properties a vector has. But there are alternative ways, of calculating the sum of a node. To reassure, that

the vector form, leads to the right solution, both ways are used to calculate the solution. Later on, only the vector form will be used. The node sums all the incoming weighted inputs up. There are three weighted inputs:

$$\begin{aligned}w_{i_1} &= i_1 \cdot w_1 \\w_{i_2} &= i_2 \cdot w_2 \\w_{i_3} &= i_3 \cdot w_3\end{aligned}$$

The sum of each weighted input, is the net value.

$$\begin{aligned}net &= w_{i_1} + w_{i_2} + w_{i_3} \\net &= (i_1 \cdot w_1) + (i_2 \cdot w_2) + (i_3 \cdot w_3)\end{aligned}$$

Now the same calculations will be done with vectors. First the vectors have to be defined, to perform operations with them. We define an input vector \vec{i} , with each element being one input. The same accounts for the weight vector \vec{w} . So both vectors are defined as follows:

$$\vec{i} = \begin{pmatrix} i_1 \\ i_2 \\ i_3 \end{pmatrix} \qquad \vec{w} = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix}$$

As it is written in Figure 4, the net is defined as dot product between the input and weight vector.

$$\begin{aligned}net &= \vec{i} \vec{w} \\net &= \begin{pmatrix} i_1 \\ i_2 \\ i_3 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix}\end{aligned}$$

By definition:

$$net = (i_1 \cdot w_1) + (i_2 \cdot w_2) + (i_3 \cdot w_3)$$

As can be seen, both ways of calculating the net value, yield the same value. This property holds for any dimension.

Firstly one must generate an output, to know how efficient the weights are, in solving the task. Thus, the first input has to be made. In the beginning, the weights are all set to 0.5. They will change with every *iteration* to make the whole structure more suitable for the task. The learning rate is set equal to the error. This leads to a decreasing learning rate when the target and output

converges and allows them to get increasingly precise. Although the number could be arbitrary, the efficiency of learning is dependant on this variable. It is not yet known, how this learning rate will work in this environment. One can assume, it will work because the learning rate is commonly adjusted in this way. For the first attempt, just the middle number is significant. This means $a = 2$. Putting the first example in the untrained networks yields:

$$net = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0.5 \\ 0.5 \\ 0.5 \end{pmatrix} = 1$$

$$o = \frac{1}{1 + e^{-net}}$$

$$o \approx 0.731059$$

The goal was, to achieve the same second number as in the input. The number was one. So the error can be calculated, by the difference of the output and target value. This step is important, because one can conclude, how well the weights are adjusted to the task.

$$E \approx 0.268941$$

In the next step, this error should be made as small as possible. A decreasing error means that the output is closer to the goal. Forward propagation describes the process of getting the output and the according error. In general, the inputs of this functions are \vec{i} and \vec{w} . The output of these sets of inputs produce is o . Now, the opposite is needed. The output is known. The weights should get adjusted, so \vec{i} has to be fixed. A function is needed to allocate \vec{i} and o to \vec{w} . Just knowing the specific values of the weights is not sufficient. One needs to know, how much every weight changed the output. This can be done by differentiation. This process is then called backpropagation. It is an efficient way of changing some numbers and looking at how the results change. If the wanted output occurs, one can change in the same manner again and would probably receive an even smaller error than before.

The more-dimensional equivalent of a slope is the gradient descent. One can imagine a three-dimensional map. At any point, the gradient points in the direction, where one can gain height the fastest. With enough time, one will get to the summit of a mountain. This is the highest point in this area. Mathematically speaking, it is a maximum. But the maximum is not wanted. This point yields the worst possible output. So, one has to go downwards

on this imaginary map, in the opposite direction of the gradient. One loses height the fastest, going in that direction. This will be done until a valley (minimum) has been found. It is the lowest point one can be in this area. This point refers to the output with the smallest error. It could be, that this valley is high in comparison to other valleys. To prevent going repeatedly in the higher valley instead of the lower, one can redo this process with different starting positions. The gradient is calculated by taking the derivative with respect to every dimension. The example has three dimensions. For simplicity the dimensions x, y and z denote the weights w_1, w_2 and w_3 .

$$\begin{aligned} f^x &= \frac{e^{-x-y}}{(e^{-x-y} + 1)^2} \\ f^y &= \frac{e^{-x-y}}{(e^{-x-y} + 1)^2} \\ f^z &= 0 \end{aligned}$$

The derivative of a dimension describes the impact it has on the change. The function f^x for example describes how x (or w_1) changed the overall output. By looking at all three dimensions one can say, that this solution makes sense. The last digit of the input was zero. Thus the according weight, did not contribute to the resulting output. By condensing this into a vector form, one gets the gradient.

$$\nabla f = \begin{pmatrix} f^x \\ f^y \\ f^z \end{pmatrix}$$

Plugging the output into the gradient yields the vector, which points in the direction of the greatest ascent. As stated in the example with the valley, one can minimize the error, by going in the opposite direction of the gradient. Mathematically speaking 'opposite' is the negative vector.

$$\nabla f = \begin{pmatrix} 0.104994 \\ 0.104994 \\ 0 \end{pmatrix} \quad (11)$$

The new weights can be obtained by subtracting a multiple of the gradient. These new weights are assured, with some exceptions, to be more efficient.

$$\begin{pmatrix} 0.5 \\ 0.5 \\ 0.5 \end{pmatrix} - 0.268941 \cdot \begin{pmatrix} 0.104994 \\ 0.104994 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.971763 \\ 0.971763 \\ 0.5 \end{pmatrix}$$

These new weights should perform better, than the initial ones. This can be easily checked by plugging the new weights into the network. The process is the same. Firstly, one has to calculate the net value. Plugging the result into the sigmoid function yields the general output.

$$\begin{aligned}
 net &= \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0.971763 \\ 0.971763 \\ 0.5 \end{pmatrix} = 1.943556 \\
 o &= \frac{1}{1 + e^{-net}} \\
 o &\approx 0.871739
 \end{aligned}$$

The input is the same as in the first iteration, so the target value is the same. The difference between the goal and the output is the error.

$$E \approx 0.125261$$

As can be seen, the error is smaller. So in this iteration the optimization was successful. However, not all iterations have to be rewarding. When the learning rate is too large, it can 'overshoot' the goal. One can prevent this with a decreasing α . In this example, it's done by setting this factor equal to the error. So as more accurate, the output gets, the smaller the change will be.

To train the network in general and not just for this particular case, the other three examples have to be used. The whole process can be condensed further, by making four iterations in one step. For every of the four iterations, the weights get updated once. The results are the same. To check if the network functions, an example can be used which was not part of the training set. The test sample is $[1, 0, 0]$. The following table shows the behaviour of the network introduced before. For the training input, only the four examples discussed before were used. To test the network's accuracy, the error of the test sample is calculated. It gets denoted by $E_{[1,0,0]}$. The weights and improved versions are also shown in the table. Iterations refers to the amount the whole optimizing process was done.

iterations	w_1	w_2	w_3	w_1^{new}	w_2^{new}	w_3^{new}	$E_{[1,0,0]}$
1	0.5	0.5	0.5	0.4091	0.6944	0.4091	0.6008
2	0.4091	0.6944	0.4091	0.3083	0.8616	0.3083	0.5764
3	0.3083	0.8616	0.3083	0.2032	1.0092	0.2032	0.5506
4	0.2032	1.0092	0.2032	0.0997	1.1425	0.0997	0.5249
5	0.0997	1.1425	0.0997	0.0026	1.2648	0.0026	0.5006
20	-0.6143	2.3232	-0.6143	-0.6340	2.3703	-0.6340	0.3465
100	-1.1661	3.8099	-1.1661	-1.1692	3.8186	-1.1692	0.2369
1000	-1.8129	5.7082	-1.8129	-1.8132	5.7089	-1.8132	0.1402
10000	-2.4101	7.4901	-2.4101	-2.4102	7.4901	-2.4102	0.0823
100000	-2.9926	9.2345	-2.9926	-2.9926	9.2345	-2.9926	0.0477

Table 1: Some values of the network at several iterations.

As can be seen in Table 1, in all the shown cases the error is smaller in the subsequent iteration, than in the original one. The new weights calculated in one round, act as new weights in the next round. To understand certain patterns one has to recall the examples stated before and their use as inputs.

The notation $w_{n,1}$ refers to all the inputs, which were weighted by w_n and were 1. For example the only relevant input for w_1 is i_1 . For w_2 it is i_2 and so on. Of the four examples used to train the AI, two 'ones' were forwarded as the first input. So $w_{1,1}$ is equal to 2. Whereas 3 times a 'one' gets weighted by w_2 . Namely the first, second and fourth example. The Table 2 shows the $w_{n,1}$ -values for all the exam-

	i_1	i_2	i_3
1. Example	1	1	0
2. Example	0	1	1
3. Example	1	0	1
4. Example	0	1	0
associated weight	w_1	w_2	w_3
amount of 1	2	3	2
$w_{n,1}$	2	3	2

Table 2: The different $w_{n,1}$ -values for the examples.

ples. With this knowledge, a certain tendency of the weights can also be determined and explained. The weight of the significant digit increases, whereas the other two decrease. This result seems plausible because in the end, the whole process is just dependant on the significant digit. To be sure that this tendency is not a coincidence, the whole process can be done again, but with a being 1 or 3. This can be easily changed by adjusting the target values. One can just use the second column of the examples shown before.

All three cases confirmed, that the weight of the significant digit will increase, and the other two decrease. This confirmation can be seen in Figure 5, 6 and 7.

Interesting to note is, that there is a certain symmetry. As can be seen in Figure 5, 6 and 7 the symmetry is easy to spot. The largest absolute change of these values always is in the significant weight. The adjustment of this weight is always positive, whereas the other two get increasingly smaller. The reason for the more reactive adjustment of the positive numbers is caused by the sigmoid function. A negative number will always yield a smaller absolute value than its positive equivalence. Furthermore, a negative number plugged into the sigmoid function added together with the positive counterpart of the original number plugged in, will be equal to 1. Whereas negative numbers will always yield values smaller than 0.5.

If the two insignificant weights have the same amounts of 1, then both get decreased equally. This can be seen with a being 2. The first and third weights increase in the same way. This is also the reason why only three lines can be seen in figure 6,

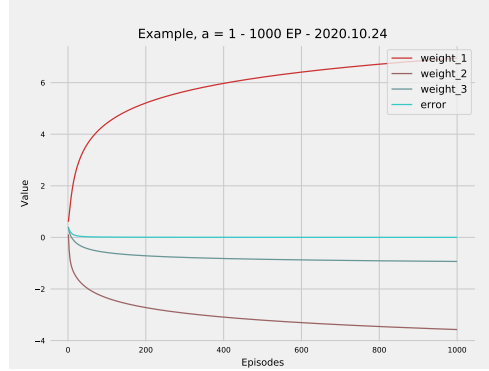


Figure 5: The weights and error with a being 1

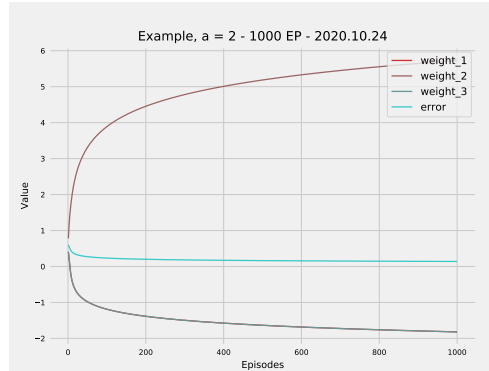


Figure 6: The weights and error with a being 2

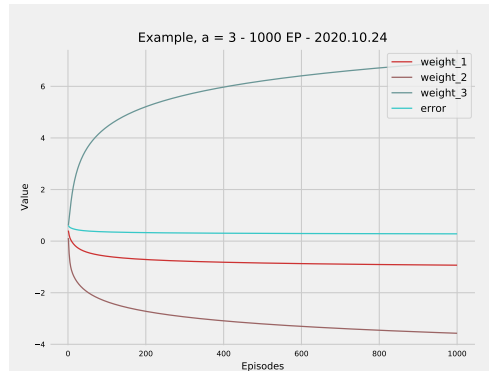


Figure 7: The weights and error with a being 3

and not four. In the other two cases, the weights are not symmetric. One of the insignificant inputs uses one more 'one' to train than the other. As shown in table 2. As a result, this weight decreases faster. This phenomenon can be seen in the case, where $a = 1$ or $a = 3$ holds. This tendency also can be observed numerically. The same conclusions as before can be drawn. When training the network with the three examples until 10'000 iterations the weights and error will be as stated in Table 3.

a	w_1	w_2	w_3	$E_{[1,0,0]}$	$w_{1,1}$	$w_{2,1}$	$w_{3,1}$
1	11.3165	-5.5673	-1.5690	10^{-5}	2	3	2
2	-2.9926	9.2345	-2.9926	0.0477	2	3	2
3	-1.5690	-5.5673	11.3165	0.1723	2	3	2

Table 3: Some values of the network with a being 1, 2 or 3 with 10000 iterations.

How can this be explained? As discussed before to do backpropagation the derivative of every dimension, or input is used. In the specific case where the input of a weight is equal to zero, the derivative also will be zero. Thus the gradient, which is used to adjust the weights, will have a zero in this dimension. This leads to not changing this weight at all. So when doing the training with these four examples, the second weight will react stronger, because it gets adjusted in three out of the four cases. This is not true for the other two weights, where they only get updated in half of all examples. This rather simple example shows how the general approach of the neuronal network works and its adjustment depending on the task. There are some major differences, when working with more nodes and connections. But in general the approach is the same.

4 Reinforcement Learning

The advantage of using neuronal networks instead of other algorithms, can be the ability to generalize abstract information. To recognize patterns in data, that cannot be described mathematically. In *deep Q-learning*, one describes this goal by giving rewards if the AI plays well. To first understand what deep Q-learning is, one needs to understand the basics of *reinforcement learning* first. This and deep Q-learning will be explained in this section.

4.1 Introduction to Reinforcement Learning

The concept of reinforcement learning (RL), is to record multiple attempts of any kind to later decide, which resulted into the best possible outcome. The main idea is, to map situations to actions. And then giving a positive *reward*, if the action was beneficial or a negative reward, if the action was detrimental. [7]

For that reason, at every attempt we record three main categories:

- State [s]
- Action [a]
- Reward [r]

If we take tic-tac-toe as an example, a *state* would be the playing field. So in a programmatic approach, one may take a list of nine elements representing nine fields. Each element then holds either a zero, one, or two. A zero could stand for an empty field, one would be a cross and two a naught. This list of nine elements would then define a state.

An action, in this case, could be a number between zero and eight, representing the index, in which field is to be placed the next symbol.

But the AI is missing one crucial part to learn successfully, the feedback. Feedback in RL is given as a reward of an action. It completes the thought behind reinforcement learning. It's like training a dog, if it does well you give it a reward in form of tasty treats. The same principle applies to RL, if the actor plays well and for example scores three in a row in tic-tac-toe, you give the agent a reward of 100 points. You can take this concept one step further and penalize the actor with -100 points if it loses. The principle of reinforcement learning then is, to use the collected data and process it in a way, to form a strategy. The strategy in RL is called *policy*. It's a function that calculates the most beneficial action, given a state [s]. [8]

This strategy can only work well if the actor can take action, that affects the state. Also, the state should relate to the goal in any way. If the problem can be expressed in a so-called *Markov decision process*, as sensation, action, and goal, the problem is suitable for a RL approach. [7]

There are different approaches to reinforcement learning. One can use a probabilistic method to calculate the best suitable action. Or one could calculate all possibilities if there are finite. These methods try to approach the

policy. In this case, a deep learning algorithm was used to approximate the policy. For more details continue to the next chapter. [9]

4.2 Deep Q-Networks (DQNs)

To first understand how deep Q-learning works, one needs to understand the principles of *Q-learning*. In the following section, this basic principle will be given. Subsequently deep Q-learning will be explained in detail.

4.2.1 Introduction to Q-Learning

Q-learning is a RL algorithm, which is operating *off-policy*. This means that the RL algorithm learns from actions that are outside of it's policy. The 'Q' stands for 'quality', which in this context means how beneficial an action is, in obtaining some future reward. The Q-learning algorithm is trying to learn a policy which maximises rewards. At the beginning the Q-learning algorithm will be taking random action, this phase is called *exploration*. This holds the purpose of increasing the variety of *experience* the agent encounters. The opposite of exploration is *exploitation*. It is considered as taking the most beneficial action the AI proposed. The most beneficial action is called a *greedy action*. [10]

$$Q_{new}(s, a) = Q(s, a) + \alpha \cdot [R(s, a) + \gamma \cdot Q'_{max}(s', a') - Q(s, a)] \quad (12)$$

The greedy action is selected using a *Q-table*. It is a table with all actions separated into columns and all possible states as rows. For each state-action pair an expected future reward will be listed. Each of those values can be calculated using a bellman equation (equation 12). Those values are referred to as the *Q-values* and represent how favorable an action is. The bellman equation will be explained in closer detail in the next chapters. [11]

4.2.2 Applying the Basics of Deep Learning

Deep Q-learning is fundamentally the same as Q-learning. But instead of a Q-table representing the policy, a deep learning AI strives to approximate the perfect policy. But what remains, is the bellman equation (equation 12). The bellman equation shows how Q-values are updated. In the same way as a Q-table, the deep Q-learning agent has it's own *memory*. It is filled up to a certain value and if it exceeds the value, the program will remove

older values to make room for the new memories. From that memory in certain steps, the deep learning algorithm will take random values and check for the optimal value of the new state $[s']$ and the new reward $[r']$ it could have taken. Then it weighs this value by *gamma* $[\gamma]$, the variable for future reward and subtracts it from the reward. [12]

This value will get subtracted by the actual Q-values it predicts. Which will then get inserted into the machine learning algorithm, in our case TensorFlow. There it will be weighted by the learning rate $[\alpha]$. Gamma and *alpha* as well as all other important variables for deep Q-learning are described more comprehensively in section 5.6.

5 Implementation

The implementation in informatics refers to the way something has been integrated. How this project tackled visuals, logic and the neuronal network will be elucidated in the following chapters.

5.1 Python

The first decision when approaching such a project is which programming language to use. In this case Python [13] was the favorable choice, because of the familiarity with the language. It's a simple language, yet a powerful tool. Python is considered best practice in scientific applications. All that, whilst being one of the most popular languages, yielding many results for a wide variety of problems. That's why Python was used for this project.

5.2 TensorFlow

The second decision one has to take for such a project is which machine learning framework to use. TensorFlow [14] is an end-to-end open-source platform for machine learning. It was developed by the Google Brain Team. Today it's considered as one of the most important machine learning frameworks. It works in many programming languages including Python. TensorFlow allows a relatively simple implementation of machine learning. [15]

In this project, the focus was laid on TensorFlow 2, the second version of TensorFlow. Therefore a basis of any TensorFlow 2 code was required. The basic example of a TensorFlow model is *CartPole*. It can be described as an

environment, containing a cart, which moves along a frictionless track. A pole which stands upright at first is attached via a joint to the cart. The machine learning algorithm should then balance this joint, by moving the cart. This project used such an example by author Siwei Xu [8] as the basis of the TensorFlow implementation. Subsequently, this example was adapted to the project's use. [8]

5.3 NumPy

NumPy [16] is a library for Python which makes numerical operations and working with large data structures easier, hence the name 'numerical' and 'python'. In this project, NumPy is used to work with arrays and for the use of some mathematical operations. These operations are more efficient when using the library than coding it in Python yourself. Additionally, it enables an efficient and easy evaluation of operations with large matrices. [16]

5.4 Matplotlib

Matplotlib [17] is a library for python which is widely used to make static, interactive, or animated visualizations. The project uses this library solely for the evaluations. The data, which were gathered during a training phase, gets plotted using this library. [17]

5.5 Pygame

This project contains games that are played by ANNs. Each of those games has been created in a way, that makes it possible to play them *headless*. Headless is a term that is commonly used in informatics to describe a system or program that runs without a graphical user interface (GUI or just UI). This for the simple reason, that an AI should be able to train without having to wait for the GUI to catch up. That way the AI can train as time efficiently as possible, resulting in possibilities to train the ANNs on a larger number of games.

Nevertheless, it is important to be able to see, what an AI is doing. This can resolve errors more intuitively, also errors in the game logic. Additionally, it can show people that have never coded before in a non-abstract version, how the AI works. So for the graphical interface, Pygame [18] is a good choice. It is resource-efficient in comparison to other options in Python and offers

features such as key-presses, mouse position, and mouse clicks. Although when comparing it to other non-Python interface builders, it's quite *low-level*. This means that you only get the most basic commands, such as the ability to draw rectangles and circles with manual coordinates. Pygame acts on a frame-by-frame basis. Meaning you have to redraw your scene every fraction of a second, for example, 30 times/frames per second (30 fps). A motion has to be described in coordinates moving pixel by pixel every certain time step. Also when analysing mouse-presses, you have to check whether or not the mouse is in a certain area or not to detect if the user clicked on a button. On the one hand, this gets complicated very quickly, but on the other hand, you get all possibilities without limitations. Pygame gives you the creative freedom to design everything the way you'd like it to be. [18]

5.6 Variables

In this section, the most important variables are explained in more detail. For understanding how an AI works, it is important to understand which variable does what.

5.6.1 Gamma [γ]

Gamma determines the importance of future rewards by multiplying them with a constant numerical factor from 0 to 1. The product is added to the reward to make it numerically more gaining, to plan for the future.

5.6.2 Epsilon [ϵ] and Decay

Epsilon determines the *exploration rate* with a numerical value from 0 to 1. If a random number in the same interval is smaller, the agent will take random action, otherwise, the choice that the AI considered as optimal, the so-called greedy action, will be chosen. This value gets continuously decreased over time, by a constant decay-factor (see Figure 8). Thus making the chance smaller to take random action increasing with time. The randomness is used to ensure the diversity of the first few decisions. This does not guarantee that the network will find the most efficient strategy, but it decreases the likelihood of repeatedly choosing a relatively good tactic, which is, in comparison with other efficient ways, worse.

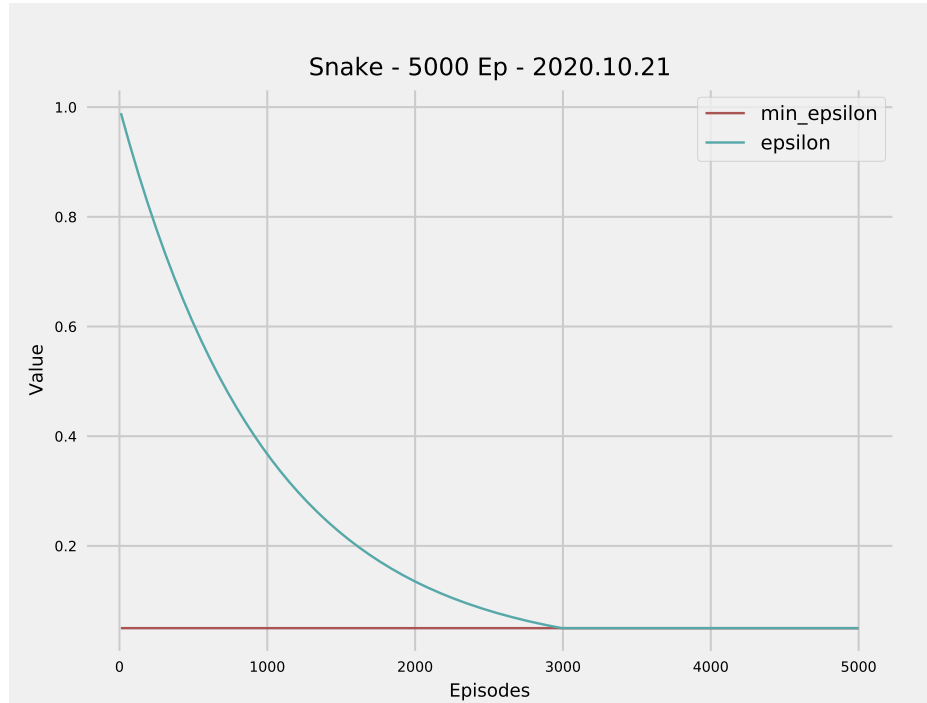


Figure 8: The decay of epsilon over time, example of game snake multiplying epsilon by 0.999 every episode. Stopping at a minimum epsilon of 0.05.

5.6.3 Number of Episodes [N]

N determines the number of games that are used to train the AI. A higher number usually means better performance, although it can also lead to reverse generalisation because the AI tries to learn all possible solutions instead of generalising.

5.6.4 Alpha [α]

Alpha determines how drastically the weights in the neuronal network get changed. This by multiplying the optimizing change by the constant factor alpha. The weights get changed to minimize the loss function.

5.6.5 Experience and Copy Step

The experience of a *DQN* model is like the memory of events in a human brain. Our DQN model saves the state, taken action, reward, the new state

and if the game is done (and therefore the last step). There are two variables *min experience* and *max experience*. The min experience variable defines how many events, it must have saved, that it actually processes the inputs. Max experience defines how many events are getting saved in total until it starts deleting the oldest one every time a new one gets added. This DQN actually consists of two separate neuronal networks. Firstly the *TrainNet* (Train Network), the network that is the most up-to-date and decides which actions to take.

The other one is called *TargetNet* (Target Network), it's meant for training purposes only. This network is structurally identical to the TrainNet. It will be periodically updated every certain step. *Copy step* defines the frequency of these updates. The purpose of the TargetNet is to avoid abrupt changes in strategy.

5.6.6 Batch Size

The *batch size* determines how many states should be processed at once. This can give the AI a sense of motion. It also gives the AI the ability to check what has happened in the previous states. By increasing this value one can improve the AIs vision. But it also drastically increases the input size which leads to longer training time. Additionally, a large batch size leads to more changes in the gradient. Because it gets calculated as many times less as the value batch size with the same training sample.

6 Basic Structure of Code

Readability is an important factor in coding too. Therefore this project went with an object-oriented layout of the code. This means that the data is structured in objects, for example in classes. This code has one class which is the model itself `MyModel` and `DQN`, those classes will be explained in more detail in section 7. In the file `games.py`, every game has its class.

These classes handle all the game inputs, processing and generate the output for the AI and the player. Then there's `train_dqn.py` and `train_dqn_vs_dqn.py`, which both explain two training methods with a class each. Some minor additional subprograms are used. For example `log.py`, was implemented to generate log files and plot the results of a training session as a graph.

This type of structure holds the advantage of having the possibility to gen-

eralize code and easily switch out a game or change the model. Additionally, it further improves readability.

7 The Model

In neuronal networks, the brain of the AI is called the model. It can be described as a container, which contains all weights, nodes, and layers of the AI. This section will explain how one can build a model underlined with code examples. To further understand the AI which was used in this project, a closer look into the code will be taken. It's the part where theory meets practice, therefore chapter 3 is precursory to this chapter. This chapter explains how a TensorFlow model can be created.

7.1 The Class MyModel

The class `MyModel` creates a neuronal network using `keras`. Keras is a library that makes the process of setting up a TensorFlow model more straightforward. `MyModel` can be divided into two parts. Where it gets called for the first time, the whole network gets created and initialized, which is the first part. The creation follows the mathematical model discussed before. The second part makes it possible to call this network and to get optimal output.

Listing 1: Example - Creation of a neuronal network using Keras.

```
1 def __init__(self, num_states, hidden_units, num_actions):
```

On initialisation of the class, some inputs in form of parameters are needed. The input `self` passes on the class, which enables the function to access the class it's located in. This format is used by Keras and is required to interfere with it and use kera-specific functions. The model uses the same three kinds of neuronal layers as explained in the mathematical part.

```
2     self.input_layer =  
        tf.keras.layers.InputLayer(input_shape=(num_states,))
```

The layer where the data gets passed on is the input layer. `num_states` is the variable that defines the size of the input layer and respectively the size of the data that will be inserted into the neuronal network. Keras requires a tuple for defining the size of this layer, which gets passed

on with `input_shape`. It could also be multidimensional but for the project's purposes, one dimension is sufficient.

```

4     self.hidden_layers = []
5     for i in hidden_units:
6         self.hidden_layers.append(tf.keras.layers.Dense(
7             i, activation='relu', kernel_initializer='RandomNormal'))

```

The second type is the *hidden units*. This type is required to have more than just one layer. So the parameter `hidden_units` passes on a list, in which every numerical element represents one layer with the size of its value. This happens by firstly clearing all the hidden layers, which can be seen in line four. Secondly a `for` loop iterates over all elements of `hidden_units`. For every iteration the procedure is the same. The size of the layer gets passed on. Additionally the activation function gets set to ReLU.

The `kernel_initializer` determines, in which state the weights should be in the beginning. The project uses solely `RandomNormal`, which sets the weights to a random numerical value. Why this is optimal, can be seen in section 3.2.2.

```

8     self.output_layer = tf.keras.layers.Dense(
9         num_actions, activation='linear',
10        kernel_initializer='RandomNormal')

```

The output layer is the last of the three layer types. The same procedure as before gets applied, except with one change. The activation function is now set to linear instead of the rectified linear unit. The model has just one output layer so this will happen once.

The first part of the class is now completed. To make it callable an additional definition has to be added.

Listing 2: Example - Making the model callable

```

1 def call(self, input):
2     z = self.input_layer(input)
3     for layer in self.hidden_layers:
4         z = layer(z)
5     output = self.output_layer(z)
6     return output

```

The inputs of this call function consists of the model itself, which gets passed

by `self` and the inputs one wants to have processed. The structure of the project assures that the inputs have the same dimension as the input layer size. So the inputs can be stored in the variable `z`. The format of `z` is set by `Keras` as a *tensor*, which is a specific kind of vector. The input layer passes the values on to the first hidden units. By iterating the hidden layers the values get passed on, by taking the output of one hidden layer as the input for the next. This can be seen in lines three and four. In line five the last hidden layer passes the value to the output layer and saves this value in `output`. This variable is returned.

7.2 The Class DQN

The structure of this class is more complex than the `MyModel` class due to conditions it has to fulfill. As the class as discussed before has an initialisation part, in which all variables get set. `MyModel` gets called for the first time in this step to create a neuronal network. The most important part is the memory, also known as experience, which is stored as a dictionary. It is structured in the following way:

Listing 3: Example - The memory of the neuronal network

```
1 experience = {'s': [], 'a': [], 'r': [], 's2': [], 'done': []}
```

At every step of the game one of these dictionaries will be created and then added to a table of all memories the agent is remembering. As can be seen in the code fragment above five types of information get stored. The first string `'s'` saves the state of the game in the beginning. `'a'` stands for the action the AI had taken and `'r'` for the rewards it yielded. What the action lead to, gets stored in `'s2'`. If the game is over, `'done'` will change to `True`. Otherwise, it is set to `False`. The memory is used to train the model. To get a brief overview the following functions are defined in the class see the following list:

1. `predict`
2. `get_action`
3. `get_q`
4. `get_prob`

5. `add_experience`
6. `copy_weights`
7. `train`

The first function `predict` makes a prediction with the model discussed in chapter 7.1. The next three functions do similar things, except the output differs slightly. For all of them `epsilon` and the state are inputs. Whereas `epsilon` is used to decide whether, a random action should be taken or not. In `get_action` the output is defined by the maximal value, the network can achieve in this state. `get_q` outputs the raw Q-values it predicted. It also returns the boolean value `True`, if a random action was taken. The function `get_prob` returns the normalized probability calculated by the Q-values, which therefore yield a value between zero and one.

The function `add_experience` is responsible for storing the experience necessary and gets called every time the agent has made a step in a game. As soon as this function gets called, a new experience dictionary will be stored. If the memory exceeds the value stated in the variable `max_experience`, the oldest memory will get deleted. The sixth function `copy_weights` copies all the weights from the normal neuronal network to the training network. Why this is important, is explained in chapter 5.6.5.

The most complex and important function of this class is `train`. Firstly, a random memory gets selected to analyse. Then the variable `batch_size` determines how many memories after the selected memory should be taken into consideration. These were stored before by the `add_experience` function. To take the future rewards into account, the `TargetNet` evaluates the maximum reward possible by the action taken. In those memories where the action was optimal, the values get incremented by `gamma` times the value computed by the `TargetNet`, which represents the future rewards.

Listing 4: Example - Optimization of the network

```

1 with tf.GradientTape() as tape:
2     selected_action_values = tf.math.reduce_sum(
3         self.predict(states) * tf.one_hot(actions,
4         self.num_actions), axis=1)
5
6     loss = tf.math.reduce_mean(tf.square(actual_values -
7         selected_action_values))

```

```

5
6 variables = self.model.trainable_variables
7 gradients = tape.gradient(loss, variables)
8 self.optimizer.apply_gradients(zip(gradients, variables))
9 return loss

```

In line 1 the Tensorflow function `GradientTape` is used. This function automatically observes any object of the type `tf.Variable` to compute the according gradient descent. [19]

To better understand line 2, let's take a closer look at every part of it. At first, `self.predict(states)` gets all Q-values of the according state again. This then gets element wisely multiplied by the one-hot encoded action. *One-hot encoding* is, when you take a categorical value and display it in a table of binary values (see Table 4).

Table 4: visualization of one-hot encoding.

Label encoding		one-hot encoding			
Color		green (1)	red (2)	yellow (3)	blue (4)
green (1)	→	1	0	0	0
red (2)		0	1	0	0
blue (4)		0	0	0	1
green (1)		1	0	0	0
yellow (3)		0	0	1	0

But why should one do this extra effort? Let's think of the following situation: One wants to input four colors into a neuronal network. Now you can only input numerical values, not strings. So one may attach an identical number to each color. Green is one, red is two, yellow is three and blue is four. This could work, but it has one flaw. The neuronal network could conclude that blue is the most superior colour there is since it has the highest number attached. Analogously the deep learning AI could say that green is the least favorable category. To defeat this behaviour, machine learning researchers use one-hot encoded tables (as example see Table 4).

Now back to line 2, the program element wisely multiplies (\odot) the one-hot encoded action previously taken by the AI with the newly predicted Q-values. Then the tensor gets added up along the y-axis using `tf.math.reduce_sum`. [20]

This complex procedure is a strategy to get the re-evaluated Q-value of the

action taken. See the following example to understand more closely:

```

Nr. of possible actions (self.num_actions) = 4
Action previously taken = 2
one-hot encoded action = [0, 0, 1, 0] (1)
Q-values calculated by the AI = [1'150, 1'535, 1'220, 773] (2)
(1)  $\odot$  (2) = [0, 0, 1'220, 0]
tf.math.reduce_sum([0,0,1'220,0], axis=1) = 1'220

```

On line 4, the program calculates the loss of the action taken. To understand this line, there's a need to define the variable **actual_values**. This variable is being calculated, by predicting the maximal Q-value with the TargetNet of the new state multiplied by gamma. That should represent the future reward weighted by gamma. This then gets added to the reward of the current memory if it is not a memory of the last step in a game. If this memory is the last step of a game, the **actual_value** variable will be just the reward of that memory. So this is the way how the **actual_value** gets calculated:

$$\text{actual_value} = R(s, a) + \gamma \cdot Q'_{max}(s', a')$$

This is enough to understand line 4. The **actual_value** gets subtracted from the **selected_values**, which is a re-evaluated Q-value of the action taken. That subtraction is then squared and saved as the loss:

$$\text{loss} = [R(s, a) + \gamma \cdot Q'_{max}(s', a') - Q(s, a)]^2$$

As seen in line 4 **tf.math.reduce_mean** is taken, this because multiple values (amount of **batch_size**) get processed at once, therefore it will take the mean of all values in those arrays, to create a general loss.

The trainable variables of the network are the weights and they get allocated to the container **variables** on line six.

Depending on the weights the loss differs. This implies, that to minimize the loss, the derivative with respect to the loss, has to be calculated. Which happens on line seven. Due to the special format of the **tape**, the rather simple subtraction has to be executed using **optimizer.apply_gradients**, as can be seen in line eight. This model can be used for almost all kinds of applications, including all three of our games. [21]

8 Graphical User Interface

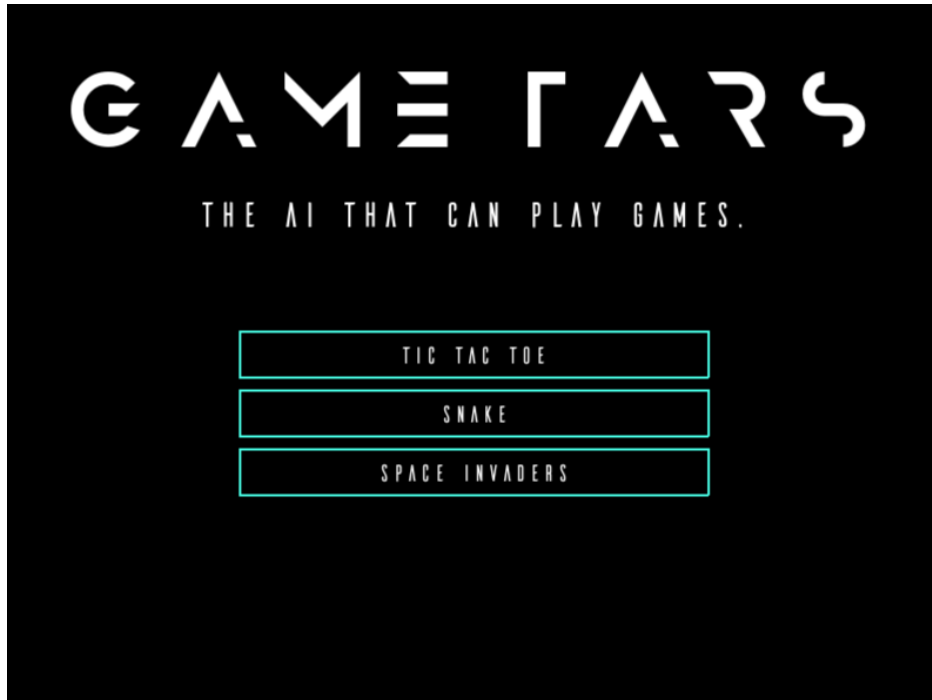


Figure 9: Title screen of Game TARS.

The Graphical User Interface (GUI) is a visual way to interact with the computer. In this project it was implemented using `pygame` (see Figure 9 for our title screen).

The most important variable for the functioning of the interface is `currentScreenfunction`. It gets called multiple times per second and if the content gets changed the screen gets updated to the specified page. Each screen has its own function for example each game itself (see Figure 10, 11 and 13 as examples). One way to change the screen functions is by using a button that has to be defined. Most screen functions are similar, thus the button example is representative of all the other ones. The following extract is a shortened version of the definition. All purely visual additions are excluded. `updateFieldVariable()` function:

Listing 5: Example - Definition of the button function

```

1  def addButton(self, message, x_center, y_center, w, h ,
    action=None):
2      mouse = pygame.mouse.get_pos()
3      x = x_center - 0.5*w
4      y = y_center - 0.5*h
5
6      # Detect mouse hover
7      if y < mouse[1] < y+h and x < mouse[0] < x+w:
8          # Detect mouse press
9          if self.mouseDidPress and action != None:
10             if message == 'Previous':
11                 self.page -=1
12             elif message == 'Next':
13                 self.page +=1
14             elif message != 'Previous' and message != 'Next':
15                 self.currentScreenFunction = action
16             self.mouseDidPress = False

```

In the first line, the name of the function is defined. The variables in the brackets have to plug in to call the function without an error. `self` stands for the class, which is in this case `pygame`. It enables to use the properties defined by the class.

The `x_center` and `y_center` define where the button should be centered. The width and height are passed by the variables `w` and `h`. The `action` allocates a specific function to the button. The `None` enables to input nothing for this variable. If nothing is specified. The variable `mouse` saves the cursor position.

In the next step, the `x` and `y`-positions get calculated to use the function `pygame.rect`. This function draws a rectangle and require that the position is defined by the upper left corner, not in the center as the button function demands. The if-statement checks if the cursor is within the button.

If the mouse is pressed and it's location inside the button it checks what the message of the button is. If the message is previous the `self.page` variable gets subtracted by one. If it's next, one gets added. When both cases don't occur, the `self.currentScreenfunction` is set to the action saved in this variable. Changing `self.currentScreenFunction` to `False` prevents the button to be pressed multiple times before the action can be executed.

9 Games

In this section, the games used will be described in detail. As well as how the game was implemented into code and what the AI receives and outputs. All games use the same deep-q learning network, but just with different inputs, outputs, hidden layers and variables.

9.1 Tic-tac-toe

In the present section a close explanation of what tic-tac-toe is, to how we implemented the AI, will be shown.

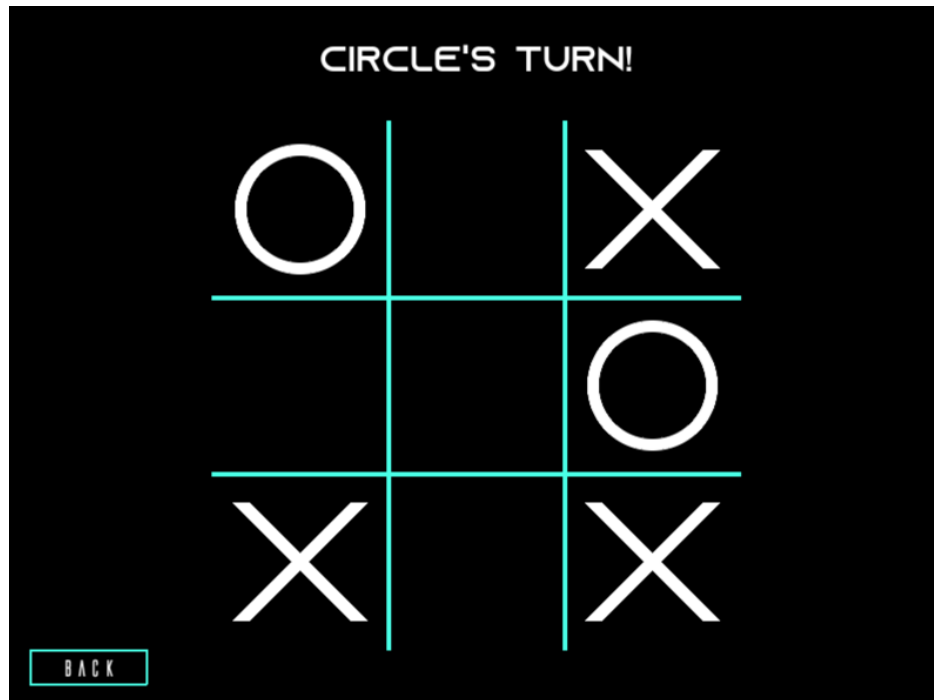


Figure 10: TicTacToe interface.

9.1.1 Rules

Tic-tac-toe is a board game, in which two players can make a move in alternating succession. Moving in this game means placing a cross or naught in

one of the nine fields (see Figure 10). Which symbol each player has to place, is predefined and unalterable for the ongoing game. A field which is taken by any of the two denotations, can't be used anymore. To win, a player has to get three of the own symbols in a row horizontally, vertically, or diagonally. The game is over if one of the players wins or all fields are occupied by a sign, which means the game ended in a tie.

9.1.2 Implementation

The idea is, that the whole game takes place in an array of nine elements, representing the nine fields of tic-tac-toe. There's a function `step()` which you can call to advance the game one step further. This function has the parameter `action`. The outer function first asks the user which action to take and then calls `step(action)` with `action` being the index of the field the player chose. Then the step function will check if the move is valid. If so, it will set a one at the index. After that, it will choose a move until it encounters a valid move for the second player, only if the game is not done yet. The algorithm will set the number two at the respective index. Finally, the function will return if the game is done and when true who won. Additionally, it returns the current state which is an array of nine fields.

9.1.3 Interaction with the AI

To make an AI which can play tic-tac-toe some data has to be distributed in both directions. The game passes the current state of the game to the network. The information will be used to make a decision, usually referred to as an action. This action will be transferred back to the game itself. The game will evaluate the new state according to its rules given. Then all the necessary information gets returned. This new information will then be passed back into the AI as an input. This happens as long as the game is not done.

There are four opponents, which the game can provide for the network to learn. The first is a real-life player. Although it is possible to train it that way, it is far slower than any computer equivalence. Therefore training an AI by human interactions would be time-consuming or require high amounts of people. For that reason one can utilise computational power. Depending on the agent, the game's settings and the computational power available, a computer needs less than a second to complete a game. This can be kept

running continuously. Computational power makes it possible to simulate over 50'000 games within an hour on the hardware available for this project. The human counterpart is much slower and needs breaks. This and the fact that machine learning needs huge amounts of data to improve, is why in most cases the AI won't be trained by the first option.

When utilising computational power one has to define the actions the computer should take as an opponent. The most simple and efficient way to determine the actions of the opponent is taking random action. Random action yields the advantage of being multiple times faster than most other computer-generated opponents.

A bit more complex and therefore slower is the *min max algorithm*, which is also known as the perfect strategy for tic-tac-toe. The algorithm calculates all possible outcomes of the current state and chooses the option which minimizes the opponent's chance to win the most. In this process, it tries to maximize its score. When playing against the min max algorithm, the network can't win. The best score it can get is by achieving a draw.

The last training method involves a duplicated neuronal network. So network to be trained learns from a network with the same structure but evolves independently. In this case, both networks try to beat each other. The data that the network receives is minimal. It gets the state discussed in chapter 9.1.2 converted to a one-hot encoded list. The generated action of the AI is the location of its move, which gets passed on to the game.

9.2 Snake

In this section it will be explained what snake is and which rules it possesses. Additionally how it was implemented and which problems were encountered while training.

9.2.1 Rules

Snake is a single-player game, in which the player pilots a snake in a field with four walls (see Figure 11). The field size, in this case, can be varied accordingly but is usually set to a 20×20 grid. The goal is to survive as long as possible and gain length by eating randomly occurring apples. The game is over, if the player controls the snake in such a way, that it collides with itself or one of the borders of the game. There are certain versions that allow the transform this game into a multiplayer one. The same rules apply but

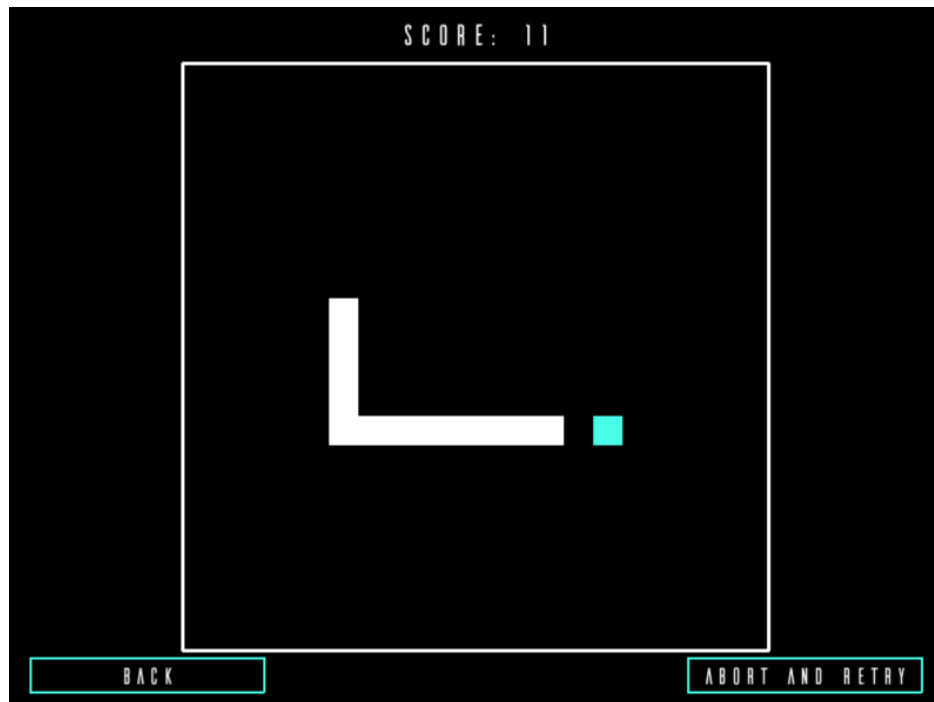


Figure 11: Snake interface.

one can also lose by colliding with the other snake.

9.2.2 Implementation

The implementation is quite simple. The game consists of a 20×20 grid, so what happens first, is that the apple gets placed randomly by selecting a value between zero and 400, not including 400. This is then saved as the index of the apple.

Every pixel of the snake is saved as its index in an array. There's no need to save the whole field of the snake as an array for game logic but it's necessary to display it afterward. This is getting handled by the `updateFieldVariable()` function:

Listing 6: Example - Create a snake field every second of size `self.field_size`

```

1  def updateFieldVariable(self):
2      # Create empty field of field_size
3      self.field = [0]*(self.field_size**2)

```

```
4
5     # Create apple
6     self.field[self.apple] = 2
7
8     # Create snake
9     for i in self.snake:
10         self.field[i] = 1
```

So after each step, a one-dimensional array with 400 zeros gets saved as `self.field`. The apple then gets set at index `self.apple`, which is represented as number two. Then the array with the indices of the snake (`self.snake`) gets iterated and at every index of snake, the field gets set to one representing the snake's body.

Also after each step, an action is read. This action then gets evaluated by checking if the snake is directly surrounding an object. If the snake heads in this direction, the game will be ended. If the snake's index of the head (last element of `self.snake`) is equal to the index of the apple, no element gets deleted. Otherwise, the first element gets deleted (last part of the snake). Additionally, the index of the field where the snake is heading is added to the list. This process runs until the player dies.

9.2.3 Interaction with the AI

Unlike tic-tac-toe the game snake doesn't have several options to train, because standard snake allows just one player. Firstly one needs to define the input for the deep learning AI.

This can be approached in different ways. One being more complicated than the other, but possibly yielding better accuracy. At first, the complete field of the snake game as input seems to be the most intuitive possibility. In more detail, this means giving the AI complete vision to the entirety of the game. This was the approach that was taken at first for this problem. Although this method has one flaw. Giving the complete field as input means, given a 20×20 field, passing on 400 inputs to the AI. This strategy can work, but inputs of higher scale mean more processing power is necessary to train the AI to perform well.

The lack of such power for this project entailed moving to a different methodology. It would be necessary to simplify the input so far, that less processing power is required for the AI to be successful. This turned out to be a better

solution given the situation. The new inputs were defined by twelve binary values. Of these twelve binary values three groups can be determined. Each of which consisting of a sequence of four values. Representing the directions up, right, down, and left in from the perspective of the snake's head. So the twelve states each binary are as follows:

- State 1-4 The apple is on the up, right, down or left side of the snake's head.
- State 5-8 An obstacle is directly up, right, down or left the snake's head.
- State 9-12 The direction of the snake's head is heading up, right, down or left.

As one can see, state 1-4 shows in which direction the snake should be heading to get to the apple. Each of the four states represents a direction, starting at top, and continuing 90° clockwise each state. If the apple is in a particular direction, it will be represented by a one, if no apple is in this direction zero will be chosen. If for example, the apple is to the left of the snake's head the input 1-4 would be $[0, 0, 0, 1]$.

State 4-8 shows for each direction, if there is an obstacle directly next to the snake's head. This can be a wall, or the snake's body itself. [22]

State 9-12 on the other hand shows the direction it's heading. As one can see in Figure 12, the head is moving left towards an apple. The only direction no obstacle appears to be directly opposed to is left. Therefore the representation in twelve states would be as follows:

$$[0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1]$$

This strategy performed very well. More on the performance in chapter 10.2. But it had one issue, the snake kept closing itself in a circle. This issue cannot be solved by rewards, since the AI simply can't see if it is enclosing itself based on its current inputs.

So to solve this, the inputs are needed to be changed. Therefore we tried out the other more resource-intensive method. This involves giving each field in the 20×20 grid a separate input. As previously stated this requires 400 inputs.

Additionally, a second problem arises. If the AI only gets the field as input, it does not know where the snake is currently heading since it does not know what the previous position was. Therefore it doesn't even know where the snake's head is currently located. This problem can be fixed by utilising the

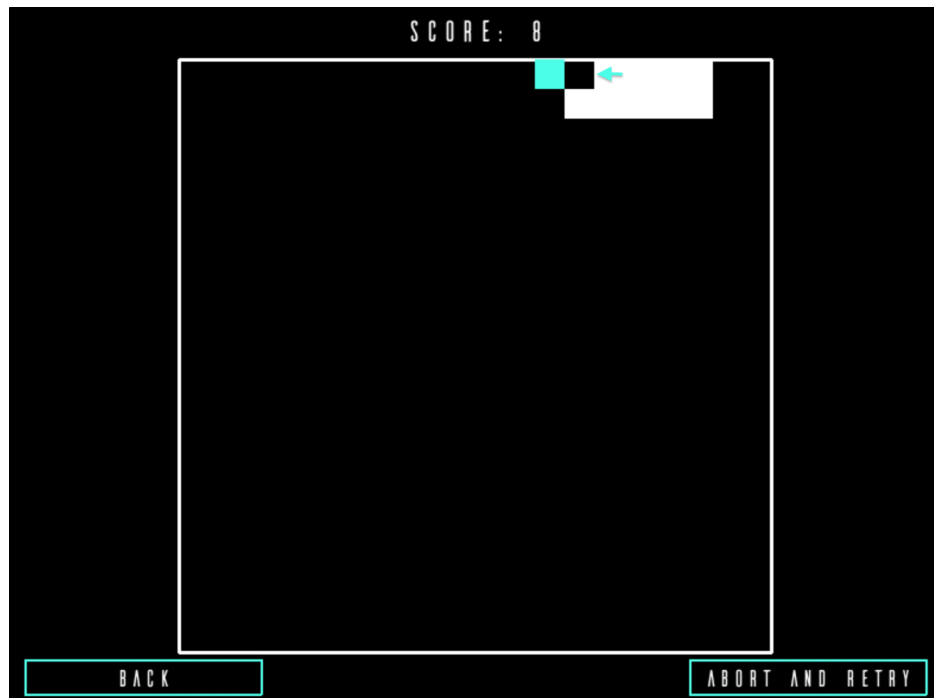


Figure 12: 20×20 Grid of Snake, the arrow representing the direction the snake's head.

variable batch size. This variable gives the AI a sense of motion and allows it to determine the snake's head. But it has one flaw, by increasing this value to two the number of inputs will be duplicated, resulting in an input size of 800. To defeat this problem the decision was made to shrink the field size to 10×10, decreasing the number of inputs to 200. This still did not result in the performance anticipated.

After analysing the situation, the conclusion was made, that the rewards must be improved. The rewards present were as follows: In short, eating

<code>reward_apple = 1000</code>	Snake collects an apple
<code>reward_death = -100</code>	Snake dies (runs into wall or itself)
<code>reward_closer = 10</code>	Snake gets closer to the apple
<code>reward_further = -15</code>	Snake gets further away from the apple

an apple will be rewarded. Colliding will be punished. Getting closer to the apple will also be rewarded, whilst the opposite gets punished a bit

stronger. These rewards describe the objectives of the game well, but it leads to misbehaviour by the AI. The problem was, that the agent was going in circles. This behaviour was very long-lasting and slowed down training drastically because the games needed a longer time to finish. Therefore a new reward `reward_repetitive = -50` was needed. This negative reward was used if the AI ended up in the same position as one of the previous six positions.

This improved not only training time, but also succeeded in a quite competent deep learning AI which plays snake. Back to the original problem, the reason why switching the inputs came apparent. The snake was enclosing itself. In this project, this was approached by adding a separate negative reward if the snake encloses itself. This however did not work well, probably due to limitations in computational power for training.

9.3 Space Invaders

Analogously to the previous chapter space invaders will be analysed. Including the explanation of the implementation and the problems encountered whilst training a capable deep learning AI.

9.3.1 Rules

Space invaders is the most complicated game of the three implemented. As in snake, it's a single-player game in which the player has to survive as long as possible. The competitor now controls a ship, which can move right and left. It also has the ability to fire upwards. The enemies are other spaceships that try to shoot the player's ship (see Figure 13). They have the same limitation, which means they can just move right, move left, and fire downwards. There are many different versions of the game, though in the following wording the implemented version is described. This adaptation is slightly simplified to make the implementation easier. Every shot costs an object one life when it gets hit by the shot. The enemy spacecraft have one heart and the ship, which the player controls three hearts. They come in three different levels and for every increase of level, the chance of spawning gets smaller. Though the possibility of firing gets increased with the levels. When all the enemies are destroyed new one will be generated. The game is over when the player's ship gets demolished.



Figure 13: Space invaders interface.

9.3.2 Implementation

As in the other games the essential data is saved in a matrix. Every element of it has a numerical value in the range from 0 to 9. Every number define the most important state as follows:

- 0 nothing
- 1 player's ship
- 2 enemy level 1
- 3 enemy level 2
- 4 enemy level 3
- 5 player's bullet
- 6 enemy's bullet

8 air

9 markers

The shapes of the objects are separately stored in matrices with the numbers listed before. A function sets different objects in the matrix at a certain position. This function is used to move the object by calling it and passing on the position and the object. The center of every matrix is permanently set to nine. This marks the center. In every processing step, all the elements of the matrix are checked for the number nine. If one is found the position will be saved in a list.

The next step is to identify what object the marker belongs to by looking around the marker for the numbers one to eight. The positions get assigned to a state and stored in three different lists, saving in the first the player, in the second the enemies, and in the third the shots. All elements of the matrix get replaced by zero to reset it. Now depending on the input, different actions are executed.

If the player pressed the left-key or right-key the position of the ship gets just shifted left respectively right if the field allows it. By calling the function and adding or subtracting a certain value. If the move is not allowed this step gets skipped leaving the ship at the same position as before. If no movement is triggered the player can fire. Again the function is called and sets a players' bullet above the position of the competitor.

In the next step, the shots are getting moved by doing the same procedure as with the player's movement. If the state is a six, which means the projectile belongs to the enemies, the shot gets moved one position down. If it is a five it gets moved one upwards. Additionally, it gets checked if a bullet is going to hit any object. If so the bullet gets deleted and one life subtracted from the spacecraft. When all opponents are demolished new ones get generated. This by calling a function that returns the level by a decreasing chance with increasing level. This random enemy gets allocated to a random position if there is space for it. This step can be made as long as the player has more than zero lives.

9.3.3 Interaction with the AI

As in snake, there are two possibilities to pass the data to the AI. One would be to forward the whole state at a specific moment. But with this method, the problem would arise that different objects can hardly be distinguished.

The first attempts used different numerical values for the objects. Positive integers were assigned to friendly objects, such as the player's ship and its projectile. The neutral element zero was assigned to the air.

Finally, the negative integers occurred only for hostile objects. The increase in levels of the opponents was signified with decreasing negative numbers. The enemy with the highest level has a larger impact than the others. Although this was well thought-through it didn't work. All the models which were trained with this method had chosen one action - right, left, or fire - and executed it constantly.

This led to players who were always going to the right or left respectively. The abstraction of the game state was the key element in making the game comprehensible for the neuronal network. Three ways of abstractions were implemented, differentiating in the information they forward to the network. The following data get returned:

1. Abstraction with four states:
 - (a) The nearest ship to the right
 - (b) The nearest ship to the left
 - (c) If an opponent is above
 - (d) If a hostile projectile is above
2. Abstraction with five states:
 - (a) The nearest ship to the right
 - (b) The nearest ship to the left
 - (c) If an opponent is above
 - (d) If a hostile projectile is right to the ship
 - (e) If a hostile projectile is left to the ship
3. Abstraction with four states and the position
 - (a) x-coordinates of the player
 - (b) The nearest ship to the right
 - (c) The nearest ship to the left
 - (d) If a hostile projectile is right to the ship

- (e) If a hostile projectile is left to the ship

Next to something always means, being one unit away from it. Whereas having something above refers to the whole x-coordinate, independently of the object's relative height. From these three abstractions, the second one worked best. This data gets passed on by a list.

Additionally, the second one uses immediate rewards in the following two situations. If the agent moves the ship under an enemy and shoots, a reward will be given, whereas when the ship is under a hostile projectile and doesn't move away, the agent gets punished.

10 Results

To give an overview over the results, all data gets stored in files. If necessary this stored information can be plotted into a graphic using *matplotlib*. The results of the following chapter are visualized by this library.

10.1 Tic-tac-toe

In tic-tac-toe a random player can achieve quite adequate scores. For that reason, it will first be analysed which probabilities a random player has. Then it will be shown how the AI performs comparatively.

10.1.1 Expected Random Win Rate

Win rate describes the amount of the games won by the AI in percent. To distinguish random choices from well-planned actions the probability of winning has to be determined. Thus yielding the expected win rate from an agent that takes purely random actions. The solution shown was discovered by Marcello Cammarata. [23]

The only fact which is important is which player starts. But if naught or cross begins is irrelevant. To make it simpler one can assume that cross starts. There are in total 9 fields in tic-tac-toe. The player that starts can at most occupy 5 positions and the second with 4. The natural question now is to ask in how many ways can this 5 crosses be distributed in the 9 fields. One position can only be claimed once. This can be calculated using the

formula for combinations without repetition.[24]

$$C_9^5 = \frac{9!}{5!(9-5)!} = 126$$

Among this amount 16 are ties. The amount of wins for the second player can be calculated by counting all configurations in which the player wins for sure. This is only the case for the diagonal triad because they prevent simultaneously a possible win for the other player.

Some configurations don't yield a definite win for a certain player. This amount is the wins for the second player except the diagonal wins. Because the first player could have won before. There are 6 layers where such a triad can lead to a win. Naught can occupy 4 fields and 3 of them are used for the win so the fourth can be set in the remaining 6 positions. This yields the total combination of $6 \cdot 6 = 36$. In this specific configuration, both players have the chance to possibly win because all the ties are excluded. From all the 126 combinations the configurations which lead to a win for X is the difference of all the ones mentioned before, which equals $126 - 16 - 12 - 36 = 62$.

By looking at which player completes first the triad, the number of wins for both players can be determined in these 'unsure' cases. Assuming that a player wins after a specific amount of draws means that the other player wins later in the game. This won't happen in the real-life game because as soon as one player wins the game is over. The term 'later' isn't the same for both players. When the first player makes his third move and the second also his third, the O's move is after the X's. But that doesn't work the other way around. There are four cases that have to be distinguished:

1. X completes at 3rd draw, O completes at 3rd or 4th move
2. O completes at 3rd draw, X completes at 4th move
3. X completes at 4th draw, O completes at 4th move
4. O completes at 4th draw, X completes at 5th move

For the first two cases, the amount can be determined by counting how many configurations are in such a way that this state is achieved. In general, the amount is determined by the product of the ways the triad can be distributed and it's frequency. The frequency is equal to the configurations possible for

the later win of the other player. When n denotes the round, the amount can be mathematically described as a formula.

$$Win_x = C_2^{n-1} \cdot \left(\sum_{i=0}^{5-n} C_{4-i}^2 \right)$$

$$Win_y = C_2^{n-1} \cdot \left(\sum_{i=0}^{6-n} C_{5-i}^2 \right)$$

Plugging the values in this formula yields the amount.

1. $Win_x^{n=3} = C_2^2 \cdot (C_2^2 + C_3^2) = 1 \cdot (1 + 3) = 4$
2. $Win_y^{n=3} = C_2^2 \cdot (C_3^2 + C_4^2) = 1 \cdot (3 + 6) = 9$
3. $Win_x^{n=4} = C_3^2 \cdot (C_4^2) = 3 \cdot 3 = 9$
4. $Win_y^{n=4} = C_3^2 \cdot (C_4^2) = 3 \cdot 6 = 18$

Thus, the probability that the first player wins is equal to $\frac{(4+9)}{40} = \frac{13}{40}$, whilst the probability of an O win is $\frac{(9+18)}{40} = \frac{27}{40}$. The final probabilities can now be computed. The chance that the first player wins is:

$$\frac{62 + 36 \cdot \left(\frac{13}{40}\right)}{126} = 0.584850$$

For the second player the likelyhood is:

$$\frac{12 + 36 \cdot \left(\frac{27}{40}\right)}{126} = 0.288275$$

Finally, the chance that none of the players win is the rest or:

$$\frac{16}{126} = 0.126875$$

To determine an improvement of the AI one should see the win rate of approximately 60% in the beginning and increasing over time. If the AI does not learn properly the win rate should be sticking around this combinatorial value.

10.1.2 Achieved Win Rate

The first few models which tried to master tic-tac-toe failed in doing so. This approach was to gradually adapt various examples to tic-tac-toe. These presets were known to function well for other tasks, but the adapted versions didn't for tic-tac-toe. After three examples in which the implementation was different tic-tac-toe was still not successfully played. After an analysis following problems had been found:

1. The AI choose fields which were occupied
2. The overall behaviour was random
3. Changing fundamental variables didn't affect the learning-rate
4. The learning-rate was approximately constant around the expectation

To prevent the first failure from happening a function was implemented to detect invalid moves and replace it with a random valid one. This correction leads to the second problem. But even though both flaws were fixed the AI did not perform better. To detect if the neuronal network at least changes its behaviour the important variables were revised. These adjustments didn't improve the learning capability. Additionally the learning-rate didn't show any form of tendency in either direction.

By linear *regression* an tendency can be seen but with increasing iterations the regression was increasingly close to zero. This means that in the long run, neither a deterioration nor an improvement happens.

The flaws suggest the agent acts purely random. To back this up a purely random network was implemented and this yielded similar results. The linear regression, in Figure 14 plotted as a dotted line, shows the trend. In the graph shown above, the random agent, plotted on the left in Figure 14 had even a greater improvement than the AI itself. So something fundamental had to be wrong. At this point, the immediate reward was introduced. It functions in the same way, as a normal reward but when the neuronal network makes a well-situated move, it will get a reward instantly. An example would be that points were given for preventing the opponent to win. This has proven to be the game-changing adjustment. This leads to actual learning one would expect the AI to have. To optimize the network even further, adjustments were made. The final result can be seen in Figure 15.

Important to note is that in the beginning, the neuronal network wins

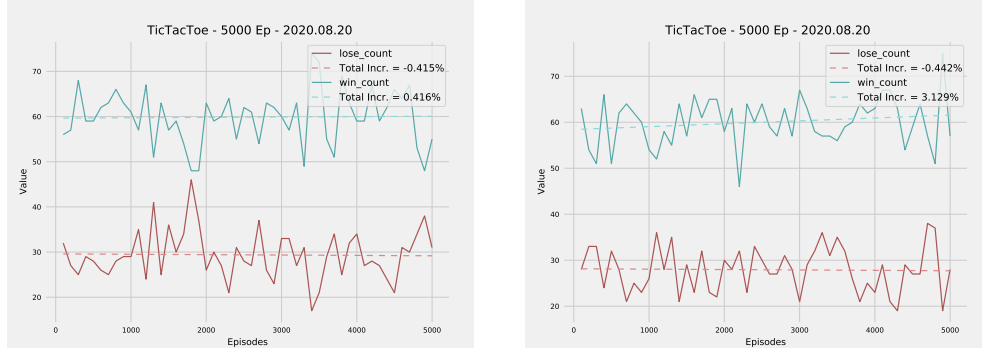


Figure 14: A comparison between the achieve winrate (left) and a purely random agent (right).

approximately 60% of all games. This coincides with the expected win rate. Because in the first few iterations the AI has not learned anything so the games are won by pure chance. This value gets improved quickly. After just 400 games the network improved itself by 18%. It won't get under this threshold earlier than 500 *episodes* later. Such outliers are normal when an AI gets trained.

The increase fluctuates and can even be a decrease. But nevertheless in increases from 59% win rate in the first 100 games to 91.5% in the last 100 iterations. Mathematically this can be well described by a regression function. Normally it has just one dimension but it can also be done in more. The function which describes the regression in four dimensions for the data points collected in Figure 15 is:

$$f(x) = (4.8 \cdot 10^{-15})x^4 + (-1.2 \cdot 10^{-10})x^3 + (1.3 \cdot 10^{-6})x^2 + (7.8 \cdot 10^{-2})x + 18.4$$

The coefficients itself are not important but the magnitude is. Note that coefficients for larger exponents get increasingly smaller. This implies that, in this case, the linear regression is sufficient to make a pretty accurate approximation for the improvement of the learning rate.

Although these percentages look promising one has to keep in mind that the network faced a random opponent. One can also train the network with an algorithm or even a human itself. So by combinations of these three options, a theoretically perfect AI could be trained. However, this was not achieved. In most cases, the neuronal network would still win. But there are a few combinations that can beat the network, every single time. It was tried to rectify this problem by making exactly this moves the network couldn't

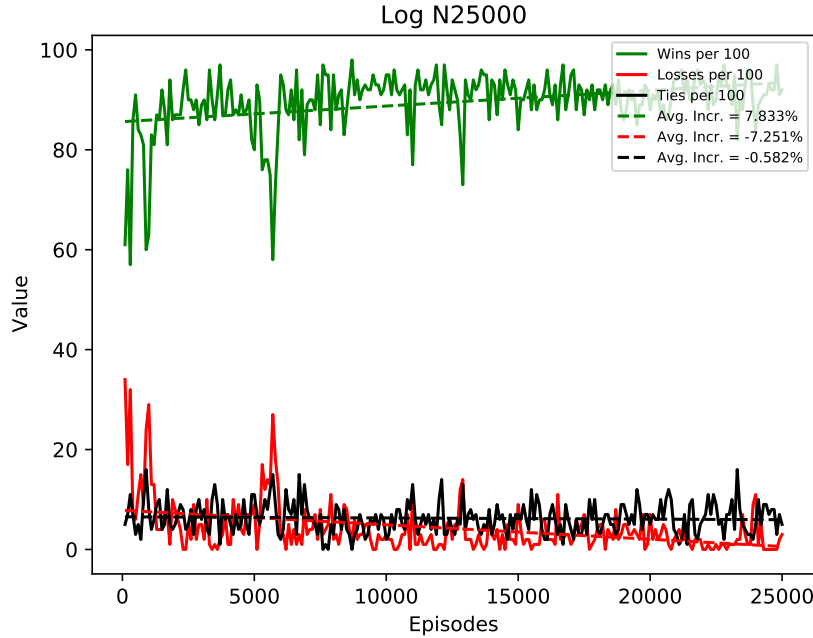


Figure 15: The learning rate of the final tic-tac-toe model.

handle. In the end, the problem was fixed but another flaw occurred. To make a flawless computational opponent one could try to train the AI with more hidden layers. If this can solve any kind of flaw is not clear.

10.2 Snake

In contrast to tic-tac-toe, an improvement of an AI playing snake can be easily be spotted because the score of a random agent would be approximately 0. The chance of accidentally crossing the spot of an apple and simultaneously have survived for that long is fairly small.

10.2.1 Achieved Score

At first snake was not performing well, although now it has greatly improved its performance. As explained in chapter 9.2.2 two types of inputs were tried. Firstly a simplified input and secondly the whole field. Both versions at the end performed similarly. But the major difference is the field size. The first approach was trained using a 20×20 field but can be expanded to any arbitrary field size. The second approach is limited to a 10×10 grid due to restrictions of available computational power.

As seen in Figure 16 and 17 the performance is similar. The average amount of apples the AI collected in the first 100 games by the second method is 1.050 apples. This average number has increased to a stunning 15.937 apples in the last 100 games, indicating an increase of 1'417.810%.

The growth is more noticeable with the first method. The first method started with a rounded average of 0.007 apples and ended with an average of staggering 23.320 apples, this is an increase of 333'042.857%. When looking at Figure 16 and 17, it's visible that the strongest improvement happened in the first half. After that, the improvement is slowing down. It seems that the simplified approach worked slightly better since it has a higher average score at the end of the training.

However, those two training graphs cannot be compared quantitatively, since the first method was trained in a 20×20 grid whilst the second method uses a 10×10 grid. It's more challenging for the AI to score quantitatively high in a smaller field. That's the case since the total points are smaller. Additionally,

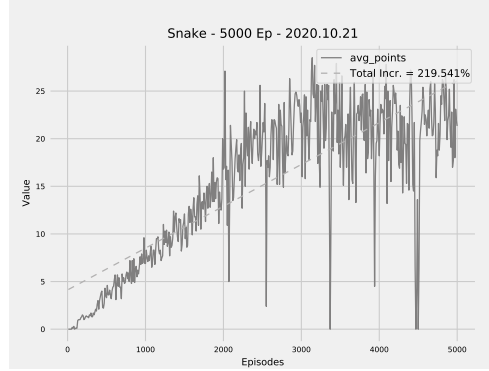


Figure 16: Method 1 of a 20×20 grid.

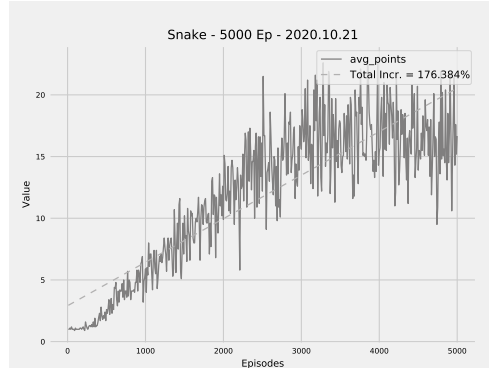


Figure 17: Method 2 of a (10×10) grid.

the snake more quickly takes up a higher percentage of the field, blocking and enclosing itself quicker. When taking a look at how the two AIs play snake, they're very similar. They both have the same issue, they play very well not making any mistakes, however at one point usually around 20 points the snake encloses itself.

This even though an enclosing penalty is in place for both AIs. Why the first method is not able to recognize this mistake, is clear, since it only sees close surroundings. But why the second method encounters this mistake is unclear. It could be, that the amount of 5'000 episodes is too small, for it to learn avoiding this mistake. Alternatively, the reward could be too small.

10.3 Space Invaders

As with snake an improvement of the AI is easy to spot in space invaders and its therefore obsolete to calculate the probability.

10.3.1 Achieved Score

The success at the beginning of this project was low. The AI developed a behaviour that had three outcomes: continuously shooting, continuously going left, or continuously right. The decision of which of the listed behaviours to take was seemingly random. The score was not zero, since sometimes it decided to shoot and by random, a ship would be above. After adding immediate rewards and simplifying the state the AI drastically improved. When taking a look at its playing behaviour it sometimes dies immediately. Nevertheless, in most situations, it scores around five to seven enemy kills. Sometimes even around 20-30 kills.

However in Space Invader not the kills count, but the score. It's calculated by kills which are weighted by the enemy level. There are three levels, the score you get is as follows:

- Level 1 Enemy: 4'000 points
- Level 2 Enemy: 9'000 points
- Level 3 Enemy: 16'000 points

The AI was trained in two steps of 500 episodes, the first time it started with a completely empty model. This can be seen in Figure 18. The second time the first model was loaded and training was being continued based on the previous model. In the first training step, the average score of the first ten games is 8'359.636. In the last ten games, the score was 43'983.667, stating a high increase. The second training was starting with an average score 45'063.090 and ended with an average of 70'166.683 points. When combining both sets of training an increase of around 739.351%

When taking a look at Figure 13 a clear upwards trend in both diagrams is noticeable. Those numbers seem very promising, but one has to account for the fact that these figures show the average.

So the fact that it sometimes dies instantly, is not represented. While of course some scores above 200'000 which do occur around every ten games, are also not visible in these figure.

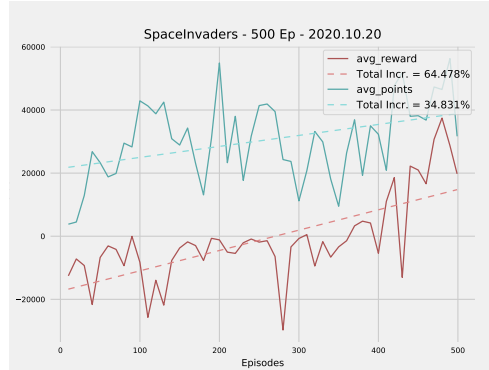


Figure 18: Space invaders, 500 episodes, first training.

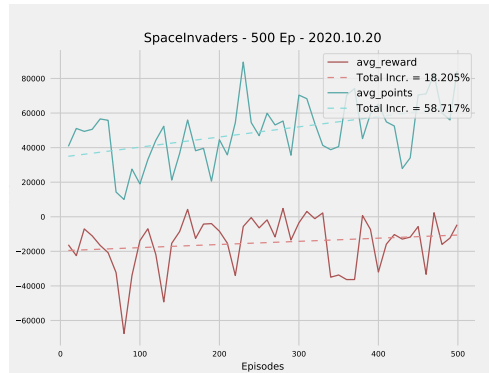


Figure 19: Space invaders, 500 episodes, second training, first training as basis.

11 Discussion

To attain a better understanding concerning the topic of artificial intelligence, we posed ourselves some questions, we strived to answer. Some of them can be answered. Some of them can not. In general, we can proudly say that all of the programs function. Though they are far from perfect. The AI which plays tic-tac-toe can be beaten. The snake-AI does not win the

game. Space invaders has some flaws too. Nevertheless, they function, this in itself is a huge achievement. The following table summarizes the gaining. As can be seen, the AI seems to work pretty well.

Game	Score in the beginning	Score in the end	Increase
TicTacToe	59.000	91.500	55.085%
Snake (20×20)	0.007	23.320	333'042.857%%
Snake (10×10)	1.050	15.937	1'417.810%
SpaceInvaders	8'359.636	70'166.683	739.351%

Table 5: The achieved growth categorized.

Additionally, we learned what the boundaries for the achieved AI is. But we can not make assumptions about the limits of neuronal networks in general. We found that after a specific degree of complexity our network can't handle that much data. But that does not mean that other models can not. The abstraction used to make more complex games comprehensible for the network can be reused for more complex tasks.

What also can be learned is, that multiple entities, with simple rules, can form complex structures. Similarly to 'Conway's game of life', one entity follows strict and simple rules. It can't form anything by itself. But when there are more entities and interact in a certain way, complex behavior can be achieved. In the case of a neuronal network, this complexity can theoretically surpass the human brain itself. Which ironically was its template.

No human can yet understand how an AI works. So, the leading questions: 'How does an AI work?' isn't the easiest one. But in the end, we got insights into the functioning of networks.

That is indeed not the whole picture but at least a fundamental part of it. This relatively easy model worked. In the complexity and amount of information, one receives today, a simple model can make many lives easier. Maybe not by playing games but the concept is the same. So one could adapt it, to almost everything one could need.

Glossary

activation function A function, which generates the output in a neuron by the summed up weighted inputs. 5, 7, 8, 12, 28, 61

AI Abbreviation for artificial intelligence. 3–5, 11, 20, 21, 23–26, 29, 31, 35–37, 39, 44, 46, 48–51, 53–56, 60

alpha Synonym for learning rate. 22, 25

amount of hidden layers [a] The amount which defines how many hidden layers get stacked after each other. 6

ANN Abbreviation for artificial neuronal network. 4, 23, 60

artificial general intelligence Type of AI, that can perform all tasks a human can perform. 4

artificial intelligence The ability of a digital device to execute tasks, which are related to human beings and animals.. 3, 4, 54, 56

artificial narrow intelligence The typical AI we encounter everyday, it's very good at a certain task, but can only perform this task. 4

artificial super intelligence Type of AI which is superior to the human brain and can perform all tasks a human is capable of. 4

backpropagation Backpropagation computes the gradient of the loss function with respect to the weights of the network. This for a single input–output example. It does so efficiently, unlike a naive direct computation of the gradient, with respect to each weight individually.[25]. 9, 14, 19

batch size A variable which defines the amount of previous inputs that get processed at every time step. By increasing this value, one can raise a sense of motion. 26, 41

binary step function An activation function type, which is mostly used for output layers. The function can only yield the value 0 or 1. 9

- CartPole** A prime example of machine learning. The task is, to balance a pole on a cart, which can be moved along a horizontal rail. 22
- computational power** Calculation powers of a computer. Usually refers to the capability of computers, to deal with high amounts of complex calculations. 3, 36, 37
- convergence** In the context of neuronal networks, it describes the behaviour of two functions such that the difference of their value get increasingly smaller. 10, 58
- copy step** The variable defines in which frequency the TargetNet should be updated. For further details read chapter 5.6.5. 26, 61
- deep learning** Type of machine learning AI which utilises neuronal networks. 3, 21, 22, 31, 39, 57
- deep Q-learning** Approximating the Q-table used in Q-learning by applying deep learning. 19, 21, 22
- deep Q-learning network** A neuronal network which uses a deep learning method. 57
- divergence** In the context of neuronal networks, it describes the behaviour of two functions, such that the difference of their value get increasingly larger. 10
- DQN** Abbreviation for *deep Q-learning network*. 25, 26, 61
- environment** A model which takes inputs as actions and returns a respective state. 23, 61
- episode** The machine learning terminology usually referred to as iteration in informatics. More specifically defining one round, or in reinforcement learning one game. 50, 53, 54
- experience** Also known as memory. Defines all states, actions, new states and rewards a agent has encountered. For further details read chapter 5.6.5. 21, 25, 26, 29, 30

exploitation The act of taking the most beneficial action, also known as the greedy action. 21

exploration When the AI takes random action, to discover inexperienced possibilities. 21

exploration rate $[\epsilon]$ When the AI takes random action, to discover inexperienced possibilities. 24

gamma $[\gamma]$ Gamma determines the importance of future rewards by acting as a factor from 0 to 1. For further details read chapter 5.6.1. 22, 24, 30, 32

gradient descent The gradient describes the slope of multidimensional functions. The negative slope points in the direction with the fastest convergence. 10, 14, 31

greedy action Taking the action the AI considers most beneficial. 21, 24, 58

headless The practice of running a program without a graphical user interface. 23

hidden layer One of the three neuronal layer types a neuronal network uses. The hidden layer does not have any connections with the in- or output. 5–9, 12, 28, 29, 35, 56, 58, 61, 65

hidden units A synonym for hidden layer. 28, 29

input Defines a possibility or act of receiving data which is later processed by a computer. 5–9, 11, 26–31, 34, 35, 44, 60

input layer The neuronal layer of a neuronal network where one inputs the information for the AI to train. 5, 6, 8, 12, 27, 29, 59, 65

input layer size $[n]$ The amount of states which are used to process the incoming data. 5

iteration In programming referred to as the amount of cycles a program goes through. 13, 16, 17, 28, 49, 50, 57

- layer size** [s] Describes the amount of neurons in a neuronal layer. 8
- learning rate** [α] Factor which defines how drastically weights should change each train step. For further details read chapter 3.2.1 or 5.6.4. 11, 13, 16, 22, 56
- linear function** An activation function type, which is mostly used for input layers. 9
- low-level** Level of programming which is close to the processor. 24
- machine learning** A practice which combines computational power with the ability to learn. 3, 22, 23, 31, 37, 57
- Markov decision process** Model which describes decision making, by observing an environment and reacting upon it, with an action. 20
- matplotlib** A library for creating static, animated and interactive visualizations in Python. 46
- max experience** This variable defines how much information should be saved in the memory. For further details read chapter 5.6.5. 26
- memory** Also known as experience, defines all states, actions, new states and rewards a agent has encountered. For further details look chapter 5.6.5. 21, 22, 25, 29, 30, 32, 59
- min experience** This variable defines what the threshold is, that the neuronal network begins to learn. For further details read chapter 5.6.5. 26
- min max algorithm** A type of algorithm yielding the optimal decision an agent can take. It calculates all possibilities, to find the most beneficial one. 37
- minimum** The point of a function where the lowest y-value is achieved. There are two types of minima. The global minimum yields the lowest value for the whole function. A local minimum however achieves this just in close range. 10
- net value** [net] The value which a neuron produces, before it gets forwarded to the activation function. 7

- neuron** A junction, which generates an output by a given input. The term can refer to a biological junction and its abstracted mathematical model. 5, 6, 8, 12, 56, 59–61
- neuronal layer** The structure multiple neurons build up. It has the property that all neurons, which are part of this layer, connect with all other neurons from the previous and following layers. 5, 27, 58–60
- neuronal network** Algorithm which mimics the brains capability to learn. Also known as artificial neuronal network (ANN). 4, 5, 9, 11, 12, 19, 25–27, 29–31, 37, 49, 50, 55–61
- off-policy** Deep learning algorithm which learns from actions which are located outside of its policy. 21
- one-hot encoding** A method to quantify categorical data. It produces a vector with the length of the data set and 0 and 1 as elements. For further details look chapter 7.2. 31
- optimization** The process of improving the AI by changing the weights. 9
- output** $[o]$ The final output of the neuron, after being passed to the activation function. 5, 6, 9–11, 26, 29, 30, 35, 60
- output layer** The neuronal layer which passes on the final output of the neuronal network. 5, 8, 12, 28, 29, 56, 65
- output layer size** $[m]$ The amount of states which are used to describe every action the agent can take. 6
- policy** Function which describes the optimal strategy. 20, 21
- Q-learning** Utilising a Q-table to determine the most beneficial action. 21, 57
- Q-table** A table consisting of Q-values. 21
- Q-value** A number which represents how favorable an action is depending on it's state. 21, 22, 30–32, 60

- Rectified linear unit (ReLU)** An activation function which is used in a neuron. The function is piecewise linear. The output of the function yields 0, until a certain threshold is reached. After this point the function is linear. It is mostly used in hidden layers. 9
- regression** A regression is a way to approximate data in a function to determine its tendencies.. 49
- reinforcement learning** An AI which tries to learn off of rewards given by the environment. 19, 20, 57
- reward** A numerical value which represents a favorability of an action. A negative reward is not favorable, whilst a high reward is favorable. 20–22, 24, 25, 29, 30, 32
- scalar product** A scalar product, or also known as dot product. It allocates a numerical value to two vectors. This number can be used to determine the angle between the input vectors. 7, 8
- sigmoid neuron** A neuron with the sigmoid activation function. This type of neuron is mostly used in hidden layers. 9
- state** Output of a environment describing its current status. 20–22, 25, 26, 28–32, 36, 37, 43, 44, 47, 60
- target** The target numerically defines what the neuronal network has to strive for. It is used to make learning possible and lead it in a certain direction. 9–11
- TargetNet** One of the two neuronal networks in the DQN model. Structurally identical to the TrainNet, which get updated every certain step defined by copy step. For further details look chapter 5.6.5. 26, 32, 57
- tensor** Tensor is a quantity which can classify scalars, vectors or analogue objects in a unified schema. Tensors are often used by the library `tensorflow` to make calculations. 29, 31
- TrainNet** One of the two neuronal networks in the DQN model. The TrainNet decides which action should be taken. For further details look chapter 5.6.5. 26, 61

vector A mathematical concept which describes the relative position of a point to another. In the machine learning context a vector is a way to condense the information of a layer into a mathematical object. 6–9, 29, 61

weight [w] A factor which the input gets multiplied with. The weights change to approximate the target values as close as possible. 5–11, 16, 25, 28, 30, 32, 60

References

- [1] B.J. Copeland. Artificial intelligence. *Britannica*, <https://www.britannica.com/technology/artificial-intelligence>, 2020.
- [2] Tannya D. Jajal. Distinguishing between narrow ai, general ai and super ai. *Medium*, <https://medium.com/mapping-out-2050/distinguishing-between-narrow-ai-general-ai-and-super-ai-a4bc44172e22>, 2018.
- [3] Knut Hinkelmann. Neural networks. *Medium*, http://didattica.cs.unicam.it/lib/exe/fetch.php?media=didattica:magistrale:kebi:ay_1718:ke-11_neural_networks.pdf, 2019.
- [4] Ayyüce Kızrak. Comparison of activation functions for deep neural networks, 2019.
- [5] Wikipedia contributors. Gradient - wikipedia, 2020. [https://de.wikipedia.org/wiki/Gradient_\(Mathematik\)](https://de.wikipedia.org/wiki/Gradient_(Mathematik)).
- [6] Neuronal network in 11 lines of python - github, 07.12.2015. <https://iamtrask.github.io/2015/07/12/basic-python-network/>.
- [7] Richard S Sutton, Andrew G Barto, et al. *Reinforcement Learning an introduction*. MIT press Cambridge, 2nd edition, 2018.
- [8] Jeremy Zhang. Reinforcement learning — implement tictactoe. *towards data science*, <https://towardsdatascience.com/reinforcement-learning-implement-tictactoe-189582bea542>, 2019.
- [9] Robert Moni. Reinforcement learning algorithms — an intuitive overview. *Medium*, <https://medium.com/@SmartLabAI/reinforcement-learning-algorithms-an-intuitive-overview-904e2dff5bbc>, 2019.
- [10] Andre Violante. Simple reinforcement learning: Q-learning. *Medium*, <https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56>, 2019.
- [11] Ayush Singh. An introduction to q-learning: reinforcement learning. *Medium*, <https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/>, 2019.

- [12] Jordi TORRES.AI. The bellman equation v-function and q-function explained. *towards data science*, <https://towardsdatascience.com/the-bellman-equation-59258a0d3fa7>, 2019.
- [13] *Python*, 2020. Available at <https://www.python.org>.
- [14] *TensorFlow*, 2020. Available at <https://www.tensorflow.org>.
- [15] Wikipedia contributors. Tensorflow — Wikipedia, the free encyclopedia, 2020. <https://en.wikipedia.org/wiki/TensorFlow>.
- [16] *numpy*, 2020. Available at <https://numpy.org>.
- [17] *matplotlib*, 2020. Available at <https://matplotlib.org>.
- [18] *PyGame*, 2020. Available at <https://www.pygame.org>.
- [19] tf.gradienttape - tensorflow, 25.09.2020. https://www.tensorflow.org/api_docs/python/tf/GradientTape.
- [20] tf.math.reduce_sum - tensorflow, 25.09.2020. https://www.tensorflow.org/api_docs/python/tf/math/reduce_sum.
- [21] tf.keras.optimizers.optimizer - tensorflow, 24.09.2020. https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Optimizer.
- [22] Hennie de Harder. Snake played by a deep reinforcement learning agent, 2020. <https://towardsdatascience.com/snake-played-by-a-deep-reinforcement-learning-agent-53f2c4331d36>.
- [23] Math department misouri contributors. Solution to problem #10 - misouri math department. http://people.missouristate.edu/lesreid/sol10_04.html.
- [24] Wikipedia contributors. Kombination - wikipedia, 2020. [https://de.wikipedia.org/wiki/Kombination_\(Kombinatorik\)](https://de.wikipedia.org/wiki/Kombination_(Kombinatorik)).
- [25] Backpropagation - wikipedia, 06.10.2020. <https://en.wikipedia.org/wiki/Backpropagation>.

List of Figures

1	The relation between the different layers.	6
2	A visualization of a neuron.	6
3	The difference between forward and back propagation.	10
4	The structure of the neuronal network. It contains three input layer nodes, one hidden layer node and one output layer node.	12
5	The weights and error with a being 1	18
6	The weights and error with a being 2	18
7	The weights and error with a being 3	18
8	The decay of epsilon over time, example of game snake multi- plying epsilon by 0.999 every episode. Stopping at a minimum epsilon of 0.05.	25
9	Title screen of Game TARS.	33
10	TicTacToe interface.	35
11	Snake interface.	38
12	20×20 Grid of Snake, the arrow representing the direction the snake's head.	41
13	Space invaders interface.	43
14	A comparison between the achieve winrate (left) and a purely random agent (right).	50
15	The learning rate of the final tic-tac-toe model.	51
16	Method 1 of a 20×20 grid.	52
17	Method 2 of a (10x10)grid.	52
18	Space invaders, 500 episodes, first training.	54
19	Space invaders, 500 episodes, second training, first training as basis.	54

Note - The source code and the project can be downloaded via this link:

<https://github.com/Silvan-M/Game-TARS>

Selbständigkeitserklärung

Betreuungsperson: Patric Rousselot

Ich erkläre hiermit, dass ich

- diese Arbeit ohne unerlaubte fremde Hilfe angefertigt habe.
- keine anderen als die angegebenen Quellen benutzt habe.
- sämtliche Stellen, die wörtlich aus fremden Quellen entnommen oder mit eigenen Worten wiedergegeben wurden, als solche gekennzeichnet und die Urheberschaft angegeben habe.
- die Meinungen und Ideen anderer explizit ausgewiesen habe.
- die Arbeit in eigenen Worten formuliert habe.

Ich bin damit einverstanden, dass meine Arbeit als pdf-Datei in der Mediothek archiviert wird und Schulangehörige diese einsehen dürfen.


Name: Funk Vorname: Brian Abt.: G4C

Name: Metzker Vorname: Silvan Abt.: G4C

Name: _____ Vorname: _____ Abt.: _____

Ort, Datum: Jonen, 22. Oktober

Unterschrift: 

Unterschrift: 

Unterschrift: _____