Kantonsschule Wohlen

Matura-project

Creating an AI which can play games

Brian Funk und Silvan Metzker

directed by
Patric Rousselot und Mark Heinz

10. September, 2020

Contents

1 Introduction

The goal of this project is to create an artificial intelligence (AI) which can play games. This is achieved by utilizing a practice called *machine learning*. Machine learning is defined as a technique which can learn by using computational power. It's a generic term mainly used to describe AIs which learn by analysing 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. In this process we strive to better understand the complexity and functioning of an artificial human-like brain.

2 Basics of Artificial Intelligence

2.1 What is an AI

An Artificial Intelligence, in short 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 a *Narrow AI*. [1]
Artificial Narrow AI (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 that detects brain tumors way more accurately than a experienced neurosurgeon is expected to. It's revolutionary and does very well at one certain task, regardless it cannot tell the difference between a cat and a dog. This is considered as an Artifcial Narrow Intelligence. An Artificial General Intelligence (AGI) on the other hand is an AI that can perform all tasks a human can fulfil. This type of AI is not yet discovered. The step after that would be a Artificial Super Intelligence (ASI), this is the kind of AI that is superior in every way a human. It is also considered as the type of AI that could possibly lead to the extinction of the human race. The closest approach to one of those higher levels of a AI was achieved by one of the so called artificial neuronal networks (ANNs). [2]

2.2 Mathematics behind a neuronal network

2.2.1 Neuronal network

A model which is used for many self-learning applications is called *neuronal network*. This model mimics the human brain and tries to describe the learning-behaviour as accurate 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. As in the brain a *neuron* takes multiple inputs and passes forward an output, which is somehow dependant on the input. The human brain uses electric current to transfer information. The AI does this numerically. The inputs 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 have more or less impact. The neuron takes all the inputs and adds them together. This sum gets forwarded to the *activation function*, which calculates the respective value. This transformed value gets weighted and gets passed to the output or to a new neuron in the next *layer*. Two neurons can make a connection, which is used to pass the numerical outputs of the function into the next neuron. These neurons build up *layers* and determine its size with the amount of neurons. One of the properties of the layer is, that all neurons from one layer don't connect to each other. But they do connect to the previous and subsequent layer. The layer can be categorized in three different kinds: input, output and hidden layers. The hidden layer describes all layers, except the in- and output layers. [3]

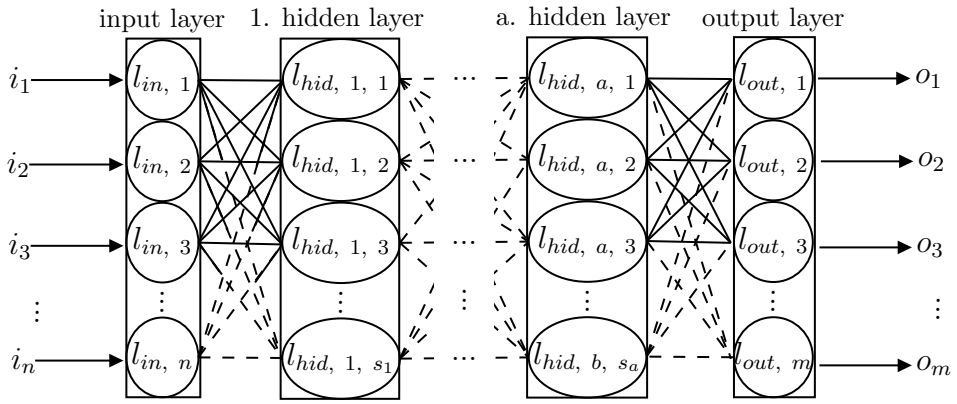


Figure 1: The relation between the different layers.

The neuronal network consists of all parts mentioned before, as it can be seen in figure 1. The first layer is the input layer and has as many nodes, as the input size n. Which means if you want to forward binary information of four variables, the input size and therefore the input layer would be 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. Every circle or ellipse represents one neuron.

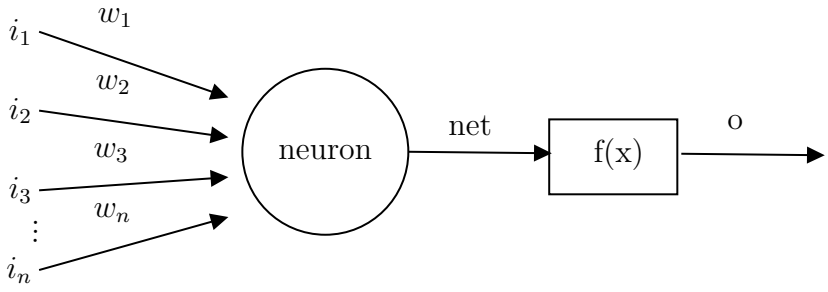


Figure 2: A visualization of a neuron.

In figure 2 the structure 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 $\vec{w}$.

$$\vec{i} = \begin{pmatrix} i_1 \\ i_2 \\ \dots \\ i_n \end{pmatrix} \qquad \vec{w} = \begin{pmatrix} w_1 \\ w_2 \\ \dots \\ w_n \end{pmatrix} \tag{1}$$

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

$$\sum_{x=1}^{n} i_x \cdot w_x \tag{2}$$

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

$$net = \vec{i} \cdot \vec{w} \tag{3}$$

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:

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

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

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

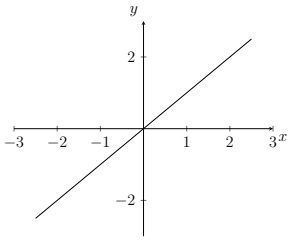
As we have seen in formula 3 the net value can also be described in a vector form. By expanding the stated definition it can be formulated, such that multiple values are defined in one vector. Although it yields the same result, the output vector $\vec{o}$ can be calculated easier when using net values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

$$\begin{pmatrix} net_1 \\ net_2 \\ ... \\ net_m \end{pmatrix} = \begin{pmatrix} w_{1,1} & w_{2,1} & ... & w_{n,1} \\ w_{1,2} & w_{2,2} & ... & w_{n,2} \\ ... & ... & ... & ... \\ w_{1,s} & w_{2,s} & ... & w_{n,s} \end{pmatrix} \cdot \begin{pmatrix} i_1 \\ i_2 \\ ... \\ i_n \end{pmatrix} \tag{6}$$

The 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 size of the layer 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

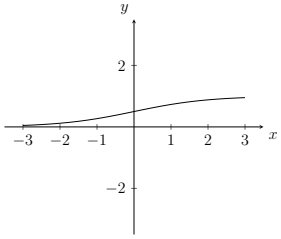
$$f(x) = x \tag{7}$$



Mostly used in input layers.

Sigmoid

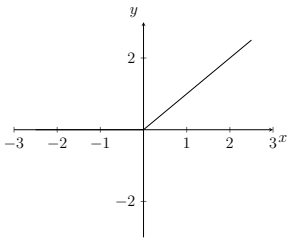
$$f(x) = \frac{1}{1 + e^{-x}} \tag{8}$$



Mostly used in hidden layers.

Rectified linear unit (ReLU)

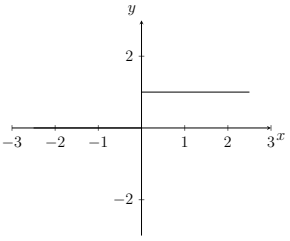
$$f(x) \begin{cases} 0 \text{ for } x \leq 0 \\ x \text{ for } x > 0 \end{cases} \tag{9}$$



Mostly used in hidden layers.

Binary step

$$f(x) \begin{cases} 0 \text{ for } x < 0 \\ 1 \text{ for } x \geq 0 \end{cases} \tag{10}$$



Mostly used in output layers.

The derivative of these functions are important in order to make a learning process possible. It is used for the *back propagation*. Therefore the linear and binary step functions are not really used in the hidden layers. Although the *sigmoid neuron* is biological more plausible, the ReLU functions has shown to be more efficient than hidden layers.[4]

2.2.2 Learning

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

$$f(x_1, x_2, x_3, ..., x_n) \rightarrow (y_1, y_2, y_3, ..., y_m) \tag{11}$$

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.

$$E = \frac{1}{m} \sum_{i=1}^{m} (y_i - t_i)^2 \tag{12}$$

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 $\vec{t}$ is 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:

$$E_{\vec{i}\,\vec{t}}(\vec{w}) \tag{13}$$

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 $\vec{w}$ we can achieve an convergence or divergence with the target values. The function E is multidimensional, that means in order to be optimized the *gradient* descent has to be used. This function is denoted by ∇. The gradient of a more-dimensional function describes its steepest ascent. [5]
In this context the gradient always points in the direction where the error increases the most. This would lead to an divergence of the outputs and

targets. So the negative gradient can be used to find the direction, in which the error decreases the fastest. Adding a multiple of the negative gradient 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, which the gradient gets multiplied with, is not to large. In the worst case scenario the new $\vec{w}$ does not perform better, because it overshoots the optimal point. An other unlucky case would be if the greatest descent of the function would lead to a local minimum which is achieving worse results than other minima. This can be prevented by fully optimizing the function multiple times with random weights. In order to complete the learning process, the optimizing has to be executed repeatedly until the outputs approximate the targets sufficiently. Mathematically the step can be defined as follows:

$$w_{new} = w_{old} - \alpha \cdot \nabla E(w_{old}) \tag{14}$$

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 own dis- and advantages. An AI which looks for patterns in pictures and has to state if the pictures 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 accordingly to which object is seen on 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. And should output dog or not a dog. By converting this label to a 0 for not a dog and 1 a dog the numerical value can be used as target value. By repeatedly showing pictures and making the optimization process stated in equation 14 the network recognises a dog increasingly better. 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. Although sometimes a machine learning approach can be unnecessary or inefficient. On the other hand, if the best possible non-machine learning algorithm is very hardware intensive, a machine learning approach might even be more efficient. Therefore it is an important point to evaluate, whether or not machine learning is more efficient or if it is necessary in the first place.

3 Reinforcement learning

3.1 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 [6]
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 you 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 cross. 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 of in which field is to be placed the next symbol.
But the AI is missing one crucial part to learn successfully, 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 looses. 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]. [7]
This strategy can only work well if the actor is able to take action that affect the state. In addition, 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. [6]
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 they are finite. These methods try to approach the

$$NewQ(s,a) = Q(s,a) + \alpha * [R(s,a) + \gamma * maxQ'(s',a') - Q(s,a)]$$

Figure 3: Bellman Equation

policy. In this case, a deep learning algorithm was used to approximate the policy. For more details continue to the next chapter. [8]

3.2 Deep Q-networks (DQNs)

3.2.1 Introduction to Q-learning

To first understand how Deep Q-learning works, one needs to understand the principles of *Q-learning*.
Q-learning is a RL algorithm which is operating *off-policy*. This means that the RL algorithm learns from actions which are outside of it's policy. The 'Q' stands for 'quality', which in this context means how beneficial a 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 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 most beneficial action is called a *greedy action*. [9]
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 a expected future reward will be listed. Each of those values can be calculated using a bellman equation (fig. 3). 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. [10]

3.2.2 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 (fig. 3). 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 with gamma [γ] the variable for future reward. And subtracts it from the Reward. [11]

$$Q(s, a) = R(s, a) - \gamma \cdot Q(s', a')$$

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 [α]. Gamma and alpha as well as all the other are important variables for deep Q-learning are described more comprehensively in section 4.3.

4 Implementation of machine learning

4.1 Python

The first decision when approaching such a project, is which programming language to use. In this case Python was the favorable choice, because of the familiarity to 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 language, yielding many results for a wide variety of problems. That's why Python was used for this project. [12]

4.2 TensorFlow

The second decision one has to take for such a project, is which machine learning framework to use. TensorFlow 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. [13][14]
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 Siwei Xu as basis of the TensorFlow implementation. Subsequently this example was adapted to our own use. [15]

4.3 Variables

4.3.1 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 in order to make it numerically more gaining, to plan for the future.

4.3.2 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 consideres as optimal, the so-called *greedy choice*, will be chosen. This value gets continuously decreased over time by a constant decay-factor and thus making the chance smaller to take random actions increasing with time. The randomness is used to ensure diversity of the first few decisions. This does not guarantee that the network will find the most efficient strategy but it decreases the likelyhood of repeatedly choosing a relatively good tactic, which is in comparison with other efficient ways worse.

4.3.3 Number of Episodes [N]

N determines the amount of games which are used to train the AI.

4.3.4 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 in order to minimize the loss function.

4.3.5 Experience and Copy Step

The experience of a DQN model is like in a human brain the memory of events. 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 while adding a new one. This DQN actually consists of two seperate 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 of strategy.

4.3.6 Batch Size

The batch size determines how many states should be processed at once, in order to detect movement.

5 User interface

5.1 Pygame

This project contains games that are played by ANNs. Each of those games have 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 possibilities to train the ANNs on 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 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 ever fraction of a second, for example 30 times/frames per second (30 fps). 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. 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. [16]

6 Basic structure of code

Readability is an important factor of coding too. Therefore this project went with a object oriented layout of the code. This basically 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 will be explained in more detail in section 7. In the file `games.py` every game has it's own 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 into a graph. This type of structure holds the advantage of having the possibility generalize code and easily switch out a game, or change the model. Additionally, it further improves readability.

7 The model

7.1 The class MyModel

The class `MyModel()` creates a callable neuronal network using `keras`. Keras is a library makes the process of setting 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 exactly the mathematical model discussed before. The second part makes it possible to call this network and calculated

numerical values, with inputs given.

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 required to interfere with it and use kera-specific functions. The model uses the same three kinds of 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 with `input_shape`. It could also be multidimensional but for the projects 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 are 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 the section 2.2.2.

```
8     self.output_layer = tf.keras.layers.Dense(
9         num_actions, activation='linear',
              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, inputs):
2     z = self.input_layer(inputs)
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` to a tensor, which is a specific kind of vector. The input layers passes the values on to the first hidden units. By iterating over 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 line three and four. In line five the last hidden layer passes the value on to the output layer and saves this value in `output`. The variable gets 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 discussed before it has an initialisation part, in which all variable get set. MyModel gets called for the first time in this step, in order to create a neuronal network. The most important part is the memory, 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 it 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 in order to train the model. To get a brief overview the following functions are defined in the class:

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 information 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 4.3.5.

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(
          self.predict(states) * tf.one_hot(actions,
          self.num_actions), axis=1)
3
4     loss = tf.math.reduce_mean(tf.square(actual_values -
          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` in order to compute the according gradient. [17] To better understand line 2, let's take a closer look at every part of it. So firstly `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 1).

Table 1: visualization of one hot encoding

Label encoding

Color
green (1)
red (2)
blue (4)
green (1)
yellow (3)

$\rightarrow$

One hot encoding

green (1)	red (2)	yellow (3)	blue (4)
1	0	0	0
0	1	0	0
0	0	0	1
1	0	0	0
0	0	1	0

But why should one do this extra effort? Let's think of following situation: One wants to input four colors into a neuronal network. Now you can only input numerical values, not strings. So one may refer to each color a identical number. 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 1). 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`. [18] This complex procedure is a strategy to get the re-evaluated Q-value of the action taken. See following example to understand more closely:

$$\begin{aligned}
\text{Nr. of possible actions (}\texttt{self.num_actions}\text{)} &= 4 \\
\text{Action previously taken} &= 2 \\
\text{One hot encoded action} &= [0,0,1,0] && (1) \\
\text{Q-Values calculated by the AI} &= [1150, 1535, 1220, 773] && (2) \\
(1) \odot (2) &= [0,0,1220,0] \\
\text{tf.math.reduce_sum([0,0,1220,0], axis=1)} &= 1220
\end{aligned}$$

On line 4 the program calculates the loss of the action taken. In order to understand this line, there needs to be a definition for 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_values variable will be just the reward of that memory. So concluding this is the way how the actual_value gets calculated:

$$actual\_value = R(s, a) + \gamma * maxQ'(s', a')$$

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:

$$loss = [R(s, a) + \gamma * maxQ'(s', a') - Q(s, a)]^2$$

As seen in line 4 `tf.math.reduce_mean` is taken, this because we are processing multiple values (amount of batch_size) 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 in order 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 it can be seen in line eight.[19]

8 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.

8.1 Tic-tac-toe

8.1.1 Rules

Tic-tac-toe is a board game, in which two players can make a move in reciprocal succession. Moving in this game means to place a cross or naught in one of the nine fields. 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 denotation can't be used anymore. In order 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.

8.1.2 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 is 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. Then the step function will output winner, if there is any, if it's done and what the current state is (array of nine fields).

8.1.3 Interaction with the AI

In order 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, which it will use to make a decision. This decision will be transferred back to the game itself, which will evaluate the new state according to the rule given. Additionally an input, either from a player or the game itself,gets generated and gives this stated yielded back to the AI. This happens as long as the game is not done. There are four opponents, which the game can provide for the network in order to learn. The first is a a real-life player. Although it is possible to train it that way, it is far slower than any computer equivalence. Depending on the agent, the game setting and the computational power, a computer needs approximately $\frac{1}{15}$ of a second to complete a game. This can be done continuously without a break. Which makes it possible to simulate over $50'000$ games within an hour. The human counterpart is much slower and needs breaks. This is why in most cases the AI won't be trained with this first option. The second option is an agent which takes purely random choices. This yields the advantages of being multiple times faster than any other computer generated opponent due to its simplicity. A bit more complex and therefore slower, is the `mini max`

`algorithm`, which is also known as the perfect strategy for Tic-tac-toe. The algorithm calculates from the current state all possible outcomes and chooses the option which minimizes the opponents chance to win the most and tries to maximize the own score, hence the name. In this training procedure the network can't win. The best score it can get is by achieving a draw. The last training method involves an 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 8.1.2 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.

8.1.4 Rules

Snake is a single-player game, in which the player pilots a snake in a field with four walls. The field size in this case can be varied accordingly, but is usually set to a 20x20 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 which allow the transform this game in a multiplayer one. The same rules apply but one can also lose by colliding with the other snake.

8.1.5 Implementation

The implementation is quite simple. The game consists of a 20x20 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 it's index in a 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 afterwards. This is getting handled by the `updateFieldVariable()` function:

Listing 5: 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 the 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 snakes 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 snakes 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 snake). Additionally, the index of the field where the snake is heading, is added to the list. This process runs until the player dies.

8.1.6 Interaction with the AI

Unlike Tic-tac-toe the game snake has not several options to train, due to its structure which allows just one player. Additionally the data flow differs. The data has four states. One stands for above, two for right, three for below and four for the left direction. Multiple of the four numbers get allocated to the position of the apple, positioned obstacles and the direction the snake is heading. This gets one hot encoded and transmitted to the AI. The data does not reveal how far away the apple is. For the obstacle this is indirectly known, due to the mechanism which only gets triggered when an obstacle is one unit away. So if an obstruction gets allocated to a direction, then its clear that this must be one unit away.

8.2 Space Invaders

8.2.1 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

which try to shoot the players ship. They have the same limitation, which means they can just move right, move left and fire downwards. There are many different versions, though in the following wording the implemented version is described. This adaptation is slightly simplified in order 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 players 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.

8.2.2 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 object by calling it and passing on the position and the object. The center of every matrix is set to permanently to nine. This marks the center. In every processing step all the elements of

the matrix are scanned for the 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 get 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.This 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 which 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 infinitely many as long as the player has more than zero lives.

8.2.3 Interaction with the AI

9 Graphical User Interface

The GUI was implemented using pygame, when was structured object oriented. The most important variable for the functioning is *self.currentScreenfunction*. It gets called multiple times per second and if the content gets changed the screen gets updated to the specified page. One way to change the screen functions is by using a button which has to be defined. The most screen functions are similar, thus the button example is representative for all the other ones. The following extract is a shortened version of the definition. All purely visual additions are excluded. `updateFieldVariable()` function:

Listing 6: Example - Definition of the button function

```
1    def addButton(self,x_center,y_center,w,h,action=None):
```

```
        mouse = pygame.mouse.get_pos()
        x = x_center - 0.5*w
        y = y_center - 0.5*h

        # Detect mouse hover
        if y < mouse[1] < y+h and x < mouse[0] < x+w:
            # Detect mouse press
            if self.mouseDidPress and action != None:
                if message == 'Previous':
                    self.page -=1
                elif  message == 'Next':
                    self.page +=1
                elif message != 'Previous' and message != 'Next':
                    self.currentScreenFunction = action
                self.mouseDidPress = False
```

In the first line the name of the function is defined. The variables in the brackets have to plugged in in order 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 the function is called with just one variable less, than suggested in the definition, than the value of this variable will be None.In the variable *mouse* the position of the cursor is saved. In the next step the x and y-positions get calculated in order 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 this is the case,the mouse gets pressed and the action is not None, it looks what the content of the variable *message* is. If the message is previous the *self.page* variable gets subtracted by one. If its 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.

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 for python.

10.1 Tic-tac-toe

10.1.1 Expected random win rate

Win rate describes the amount of the games won by the AI in percent. In order to distinguish random choices from a well-planned actions the probability of winning has to be determined. Thus yielding the expected win rate from an agent which takes purely random actions. The solution shown was discovered by *Marcello Cammarata* [20].
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.[21]

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

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 yield not a definite win for a certain player. This amount are 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 which player completes first the triad, the amount of wins for both players can be determined in this '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.

$$Win_x = C_2^{n-1} \cdot \left(\sum_{i=0}^{5-n} C_{4-i}^2 \right) \tag{15}$$

$$Win_y = C_2^{n-1} \cdot \left(\sum_{i=0}^{6-n} C_{5-i}^2 \right) \tag{16}$$

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:

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

In order 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 combinatorical value.

10.1.2 Achieved winrate

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

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

11 Conclusion

12 Glossary

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)*.

[6] Richard S Sutton, Andrew G Barto, et al. *Reinforcement Learning an introduction*. MIT press Cambridge, 2nd edition, 2018.

[7] Jeremy Zhang. Reinforcement learning — implement tictactoe. *towards data science, https://towardsdatascience.com/reinforcement-learning-implement-tictactoe-189582bea542*, 2019.

[8] Robert Moni. Reinforcement learning algorithms — an intuitive overview. *Medium, https://medium.com/@SmartLabAI/reinforcement-learning-algorithms-an-intuitive-overview-904e2dff5bbc*, 2019.

[9] Andre Violante. Simple reinforcement learning: Q-learning. *Medium, https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56*, 2019.

[10] Ayush Singh. An introduction to q-learning: reinforcement learning. *Medium, https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/*, 2019.

[11] Jordi TORRES.AI. The bellman equation v-function and q-function explained. *towards data science, https://towardsdatascience.com/the-bellman-equation-59258a0d3fa7*, 2019.

[12] *Python*, 2020. Available at *https://www.python.org*.

[13] *TensorFlow*, 2020. Available at *https://www.tensorflow.org*.

[14] Wikipedia contributors. Tensorflow — Wikipedia, the free encyclopedia, 2020. *https://en.wikipedia.org/wiki/TensorFlow*.

[15] Siwei Xu. Deep reinforcement learning: Build a deep q-network(dqn) with tensorflow 2 and gym to play cartpole. *towards data science, https://towardsdatascience.com/deep-reinforcement-learning-build-a-deep-q-network-dqn-to-play-cartpole-with-tensorflow-2-ar* 2019.

[16] *PyGame*, 2020. Available at *https://www.pygame.org*.

[17] tf.gradienttape - tensorflow, 25.09.2020. *https://www.tensorflow.org/api_docs/python/tf/GradientTape*.

[18] tf.math.reduce_sum - tensorflow, 25.09.2020. *https://www.tensorflow.org/api_docs/python/tf/math/reduce_sum*.

[19] tf.keras.optimizers.optimizer - tensorflow, 24.09.2020. *https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Optimizer*.

[20] Math department misouri contributors. Solution to problem #10 - misouri math department. *http://people.missouristate.edu/lesreid/sol10_04.html*.

[21] Wikipedia contributors. Kombination - wikipedia, 2020. *https://de.wikipedia.org/wiki/Kombination_(Kombinatorik)*.

List of Figures