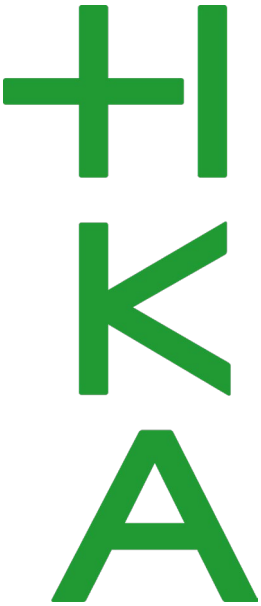
Hochschule Karlsruhe
University of
Applied Sciences

Fakultät für
Elektro- und
Informationstechnik



Studiengang Elektro- und Informationstechnik –
Automatisierungstechnik

Projektarbeit

Entwicklung und Implementierung eines Deep-Q-Learning-Algorithmus am Beispiel eines Mühlespiels

von
Clemens Diener (64581)
Andre Kronsbein (64588)

Betreuer: Prof. Dr.-Ing. Manfred Strohrmann

Inhaltsverzeichnis

1 Einleitung

1.1 Hinführung Thema

Künstliche Intelligenz ist aus unserer heutigen Welt nicht mehr weg zu denken. Diese Projektarbeit hat zum Ziel, den Q-Learning Algorithmus anhand eines Demonstrators zu verstehen. Q-Learning ist aus dem Bereich des Reinforcement Learning. Dieser Bereich steht für eine Reihe von Methoden des maschinellen Lernens, die selbständig zur Laufzeit lernen. Dadurch sind im Gegensatz zu anderen Algorithmen keine gelabelten Datensätze nötig.

Ziel der Projektarbeit ist es, eine laufzeitfähige Anwendung zu haben, über die Mühle gespielt werden kann. Ein Spiel kann Mensch gegen Mensch, Mensch gegen KI und KI gegen KI sein. Das Training der KI, sowie die Validierung erfolgt ebenfalls in der GUI. Trainierte Modelle können einfach gespeichert und später wieder geladen werden. Nachfolgend zu dieser Projektarbeit wird ein Aufbau mit Roboter erstellt, so dass physisch gegen eine trainierte KI gespielt werden kann.

1.2 Q-Learning-Algorithmus

Beim klassischen Q-Learning werden zu jedem Zustand S Q-Werte gelernt, die die Qualität aller in diesem Zustand möglichen Aktionen A beschreiben. Es wird also eine Tabelle oder ähnliche Datenstruktur benötigt, in der für alle möglichen Zustände S und mindestens für jede gültige Aktion A in den jeweiligen Zuständen ein Q-Wert abgespeichert wird, der die entsprechende Aktion A bewertet. Das Lernen erfolgt über die Aktualisierung der Q-Werte mit der Formel

$$Q(S_t, A_t) = (1 - \alpha) \cdot Q(S_t, A_t) + \alpha \cdot (R_t + \lambda \cdot max_a Q(S_{t+1}, a))$$ [1]

mit folgenden Bedeutungen der Symbole:

- S: Zustand, wobei S_t der aktuelle Zustand und S_{t+1}der Folgezustand innerhalb eines Spiels ist.
- A: Aktion
- $Q(S_t, A_t)$: Q-Wert zu der im aktuellen Zustand gewählten Aktion
- $max_a Q(S_{t+1}, a))$: maximal erreichbarer Q-Wert durch eine beliebige Aktion im durch die gewählte Aktion resultierenden Folgezustand
- R_t: Reward, also die Belohnung oder Bestrafung für die gewählte Aktion
- α: Lernrate, die bestimmt, wie stark sich die gespeicherten Q-Werte durch einen Aktualisierungsschritt anpassen sollen.
- λ: Discount-Faktor, der bestimmt, wie stark zukünftige Ereignisse im Verhältnis zum aktuellen Reward gewichtet werden sollen.

Es wird also in jedem Aktualisierungsschritt nur der Q-Wert zu der im aktuellen Zustand ausgewählten Aktion aktualisiert. Dabei werden die Q-Werte des Folgezustands verwendet, um den in Zukunft zu erwartenden Reward abzuschätzen. Um den vorliegenden Zustandsraum in den Q-Werten der Tabelle möglichst vollständig abzudecken, ist es gerade zu Beginn des Trainings notwendig, so viele verschiedene Aktionen wie möglich durchzuführen. Um dies zu erreichen, wird der sogenannte Epsilon-Greedy-Algorithmus verwendet. Zu einer Wahrscheinlichkeit Epsilon wird dabei nicht die durch den

[1] Q-Learning – einfach erklärt. 6.Mai 2023 (https://databasecamp.de/ki/q-learning)

maximalen Q-Wert zum aktuellen Zustand der Tabelle gewählte Aktion, sondern eine zufällige Aktion ausgewählt. Epsilon hat zu Beginn des Trainings den Wert 1 und wird im Verlauf des Trainings immer weiter reduziert, sodass die gelernten Q-Werte verstärkt für die Wahl der Aktionen benutzt werden.

1.3 Bewertung des Q-Learning für das Beispiel Mühlespiel

In Bezug auf das Mühlespiel hat sich herausgestellt, dass die Verwendung des klassischen Q-Learnings aufgrund eines sehr großen Zustandsraums nicht möglich ist. Allein in der Zug- und Springphase des Mühlespiels ergibt sich unter Berücksichtigung aller Symmetrien eine Menge von 7.673.759.269 Zuständen[2]. Die Anzahl der Zustände in der Setzphase ist noch einmal deutlich größer. Unter Berücksichtigung aller Symmetrien, aber unter Einberechnung von ungültigen Zuständen handelt es sich hier um 17.874.891.168 Zustände[3]. Wenn man von insgesamt etwa 25 Milliarden Zuständen und einer konservativen Abschätzung von 10 möglichen Zügen pro Zustand ausgeht, ergibt sich bei einer minimalen Speichergröße von 4 Byte pro Q-Wert ein Gesamtspeicherbedarf von etwa einem Terrabyte nur für die Q-Werte. Da die Abspeicherung und dynamische Verwendung einer so großen Datenstruktur nicht realistisch sind, wird anstatt dem klassischen Q-Learning auf Deep-Q-Learning zurückgegriffen. Dieses Lernverfahren, bei dem die Abspeicherung sämtlicher Q-Werte durch die approximative Berechnung der Q-Werte über ein Neuronales Netz ersetzt wird, wird in dieser Projektarbeit implementiert.

[2] Gasser, Ralph. "Solving nine men's morris." Games of No Chance MSRI Publications Volume 29 (1996).
[3] Loewer, Wesley. (2016). The Effects of Rule Variations on Perfect Play Databases for Nine Men's Morris. 10.13140/RG.2.1.4972.4407.

2 Mühlenlogik

Die Mühlenlogik dient dazu, den Spielablauf zu verwalten und dafür zu sorgen, dass die Spielregeln eingehalten werden. Es wird das in Abbildung 2-1 oben dargestellte standardmäßige Spielfeld mit 24 Feldern verwendet.

2.1 Spielregeln

Folgende grundlegende Regeln gelten:

- Die Spieler sind abwechselnd an der Reihe.
- Eine Mühle besteht aus 3 eigenen Spielsteinen in einer Reihe von verbundenen Feldern.
- Bildet ein Spieler eine Mühle, darf er einen Spielstein des Gegners von einem Feld entfernen und aus dem Spiel nehmen, der nicht Teil einer Mühle ist.
- Zu Beginn besitzt jeder Spieler 9 Spielsteine, die sich außerhalb des Spielfelds befinden.
- In der Setzphase setzen die Spieler abwechselnd ihre Spielsteine auf freie Felder im Spielfeld.
- Sobald alle Spielsteine gesetzt wurden, beginnt die Zugphase, in der ein eigener Spielstein von seinem aktuellen Feld auf ein damit direkt verbundenes leeres Feld gezogen werden darf.
- Wenn ein Spieler nur noch 3 Spielsteine besitzt, darf er mit diesen Steinen springen, also einen Spielstein auf ein beliebiges leeres Feld im gesamten Spielfeld bewegen.
- Ein Spieler verliert, sobald er nur noch 2 Spielsteine besitzt oder keinen gültigen Zug durchführen kann. Der andere Spieler gewinnt dann.

Zusätzlich werden folgende spezielle Regeln verwendet:

- Ein Unentschieden tritt auf, wenn nach einer selbst definierten Anzahl von Spielzügen kein Spieler gewonnen hat.
- Wenn ein Spielstein entfernt werden darf, aber alle Spielsteine des Gegners Teil einer Mühle sind, darf ein beliebiger Spielstein entfernt werden.
- Wenn in der Setzphase gleichzeitig 2 Mühlen gebildet werden, darf trotzdem nur ein Spielstein entfernt werden.

2.2 Darstellung des Spielfelds

Die direkte Abbildung der Geometrie des Spielfelds in der Software würde zu einem Array mit 7 mal 7 Feldern und vielen nicht erlaubten Feldern führen. Stattdessen wird die Darstellung wie in Abbildung 2-1 unten verwendet. Das Spielfeld wird zwischen den Feldern oben links und links in jedem Ring aufgeschnitten. Die Felder jedes Rings werden beginnend von oben links im Uhrzeigersinn nummeriert. Damit kann das Spielfeld als ein Array mit 3 mal 8 Feldern (3 Ringe und 8 Positionen in jedem Ring) implementiert werden. Bei ungeraden Positionen innerhalb des Rings ist ein Ziehen zwischen den verschiedenen Ringen möglich. Außerdem muss beachtet werden, dass die Ringe geschlossen sind. Es ist also möglich zwischen den Positionen 7 und 0 innerhalb eines Rings zu ziehen.

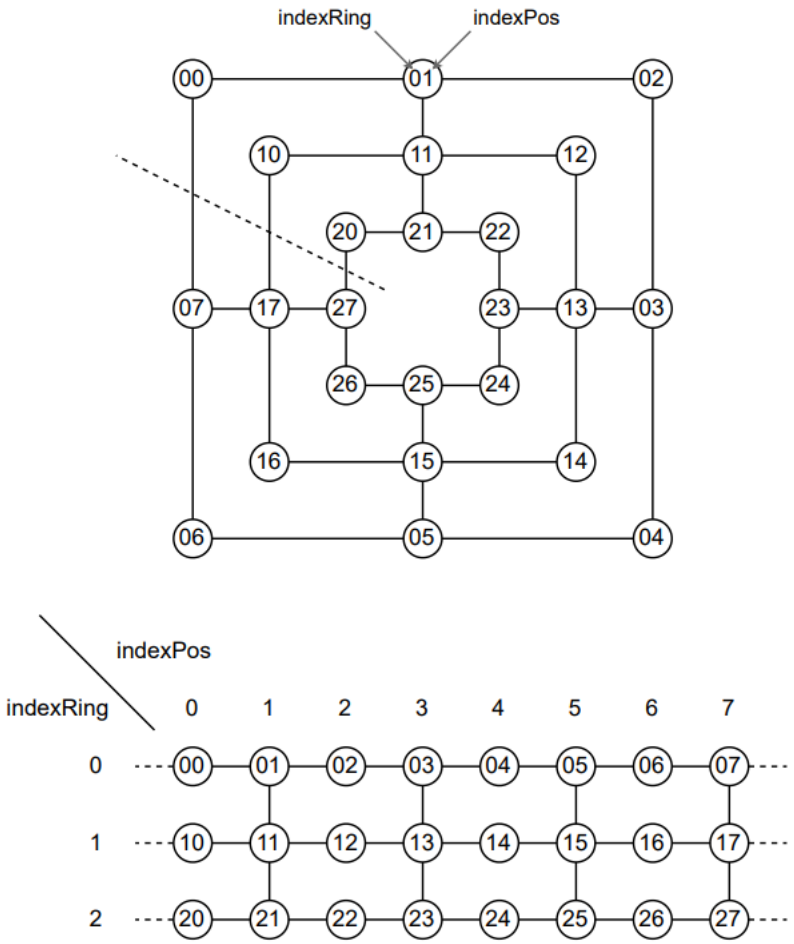


Abbildung 2-1: Spielfeld mit Darstellung in der Software

2.3 Aufbau der Software

Die Mühlenlogik ist als Klasse implementiert. Damit ist es möglich, mehrere Instanzen der Logik zu erstellen und mehrere Spiele parallel durchzuführen.

Mit der ersten Gruppe von Methoden der Klasse läuft die Interaktion ab. Es kann ein neues Spiel gestartet, der aktuelle Zustand des Spiels ausgelesen und ein neuer Spielzug durchgeführt werden. Es werden hier nicht die Änderungen am Spielfeld, sondern immer der gesamte Zustand des Spielfelds betrachtet. Um einen Zug durchzuführen, wird also der Gesamtzustand aller Felder im Spielfeld nach dem Zug benötigt.

Die eigentliche Logik wird durch eine Zustandsmaschine implementiert. Die möglichen Zustände sind in Tabelle 2-1 aufgeführt. Wenn ein neuer gültiger Zug durchgeführt wird, wird eine Methode zum aktuellen Zustand aufgerufen. In dieser werden der Folgezustand, die nächste erwartete Aktion sowie die bei der nächsten Aktion erlaubten Züge berechnet und der neue Zustand des Spielfelds übernommen.

Tabelle 2-1: Zustände der Mühlenlogik

Zustand	Beschreibung
SetPlayer1 (0)	Spieler 1 muss einen Spielstein setzen, Initialzustand
SetPlayer2 (1)	Spieler 2 muss einen Spielstein setzen
SetMillPlayer1 (2)	Spieler 1 hat in der Setzphase eine Mühle geformt und muss einen Spielstein von Spieler 2 entfernen
SetMillPlayer2 (3)	Spieler 2 hat in der Setzphase eine Mühle geformt und muss einen Spielstein von Spieler 1 entfernen
ShiftPlayer1 (4)	Spieler 1 muss einen Spielstein ziehen (Springphase inbegriffen)
ShiftPlayer2 (5)	Spieler 2 muss einen Spielstein ziehen (Springphase inbegriffen)
ShiftMillPlayer1 (6)	Spieler 1 hat in der Zugphase eine Mühle geformt und muss einen Spielstein von Spieler 2 entfernen
ShiftMillPlayer2 (7)	Spieler 2 hat in der Zugphase eine Mühle geformt und muss einen Spielstein von Spieler 1 entfernen
GameFinished (8)	Das Spiel ist beendet, Endzustand

2.4 Interaktion mit der Mühlenlogik

Die Methoden, über die eine Interaktion mit der Mühlenlogik möglich ist, sind in Tabelle 2-2 aufgeführt.

Tabelle 2-2: Methoden für die Interaktion mit der Mühlenlogik

Methode	Beschreibung
__init__(checkMoves=True)	Eine neue Instanz der Mühlenlogik wird initialisiert. Es kann ausgewählt werden, ob die durchgeführten Züge überprüft werden sollen. Diese Funktion sollte verwendet werden, wenn falsche Züge nicht sicher ausgeschlossen werden können.
restartGame( )	Ein neues Spiel wird gestartet.
getState( )	Das aktuelle Spielfeld wird als 3x8 Array mit dem Zustand von allen Feldern (EMPTY/PLAYER1/PLAYER2) zurückgegeben. Außerdem wird die nächste erwartete Aktion zurückgegeben: • SetPlayer1 (0): Spieler 1 setzt einen Spielstein • SetPlayer2 (1): Spieler 2 setzt einen Spielstein • ShiftPlayer1 (2): Spieler 1 zieht einen Spielstein • ShiftPlayer2 (3): Spieler 2 zieht einen Spielstein • RemoveTokenPlayer1 (4): Spieler 1 entfernt einen Spielstein von Spieler 2 • RemoveTokenPlayer2 (5): Spieler 2 entfernt einen Spielstein von Spieler 1 • Player1Wins (6): Spieler 1 hat das Spiel gewonnen

	• Player2Wins (7): Spieler 2 hat das Spiel gewonnen
getPossibleMoves( )	Alle möglichen Folgezustände des Spielfelds werden zurückgegeben. Es handelt sich also um eine Liste aus mehreren 3x8 Arrays.
getInStockTokens( )	Die Anzahl der Spielsteine, die jeder Spieler noch setzen kann wird zurückgegeben.
getRemainingTokens( )	Die Anzahl der Spielsteine, die von jedem Spieler insgesamt noch im Spiel ist, wird zurückgegeben.
getFullState( )	Vereint die Rückgaben der Methoden getState( ), getPossibleMoves( ), getInStockTokens( ) und getRemainingTokens( ).
getMovesFromSelectedToken(indexRing, indexPos)	In der Zug- oder Springphase werden für die gegebene Position im Spielfeld die möglichen Folgepositionen des ausgewählten Spielsteins zurückgegeben.
setMove(newMillField)	Ein Spielzug wird durchgeführt, indem dieser Methode der Zustand des Spielfelds nach dem Zug übergeben wird. Wenn die Funktion aktiviert ist, wird zunächst überprüft, ob der durchgeführte Zug erlaubt ist. Ist dies der Fall, wird die Methode zum aktuellen Zustand der Zustandsmaschine aufgerufen und diesem der neue Zustand des Spielfelds übergeben.
initializeSpecificState(…)	Mit dieser Funktion kann die Mühlenlogik in einen beliebigen Zustand versetzt werden, indem der Zustand des Spielfelds, die nächste erwartete Funktion und die Anzahl der noch zu setzenden Spielsteine definiert werden.

3 Deep-Q-Learning-Algorithmus

3.1 Aufbau der Netzwerke

Im Unterschied zum klassischen Q-Learning wird beim Deep-Q-Learning ein Neuronales Netz für die Berechnung der Q-Werte für die verschiedenen Aktionen zu einem Zustand verwendet. Diese „Deep-Q-Learning-Networks (DQN) berechnen also die Güte der verschiedenen Aktionen, anstatt diese Q-Werte zu jeder Aktion in einer Tabelle abzuspeichern. Damit kann eine sehr große Tabelle durch ein vergleichsweise kleines Neuronales Netz mit deutlich weniger zu lernenden Parametern approximiert werden. Wie in Abbildung 3-1 zu sehen ist, werden beim Deep-Q-Learning die Q-Werte zu allen möglichen Aktionen in einem Zustand gleichzeitig berechnet. Es müssen also nicht alle Q-Werte einzeln berechnet werden. Beim klassischen Q-Learning ist dies unproblematisch, da das Ermitteln des Q-Wertes zu einem Zustand und einer Aktion durch eine Look-Up-Operation schnell erfolgen kann. Der Rechenaufwand bei der einzelnen Berechnung jedes Q-Wertes über ein Neuronales Netz wäre aber unverhältnismäßig hoch.

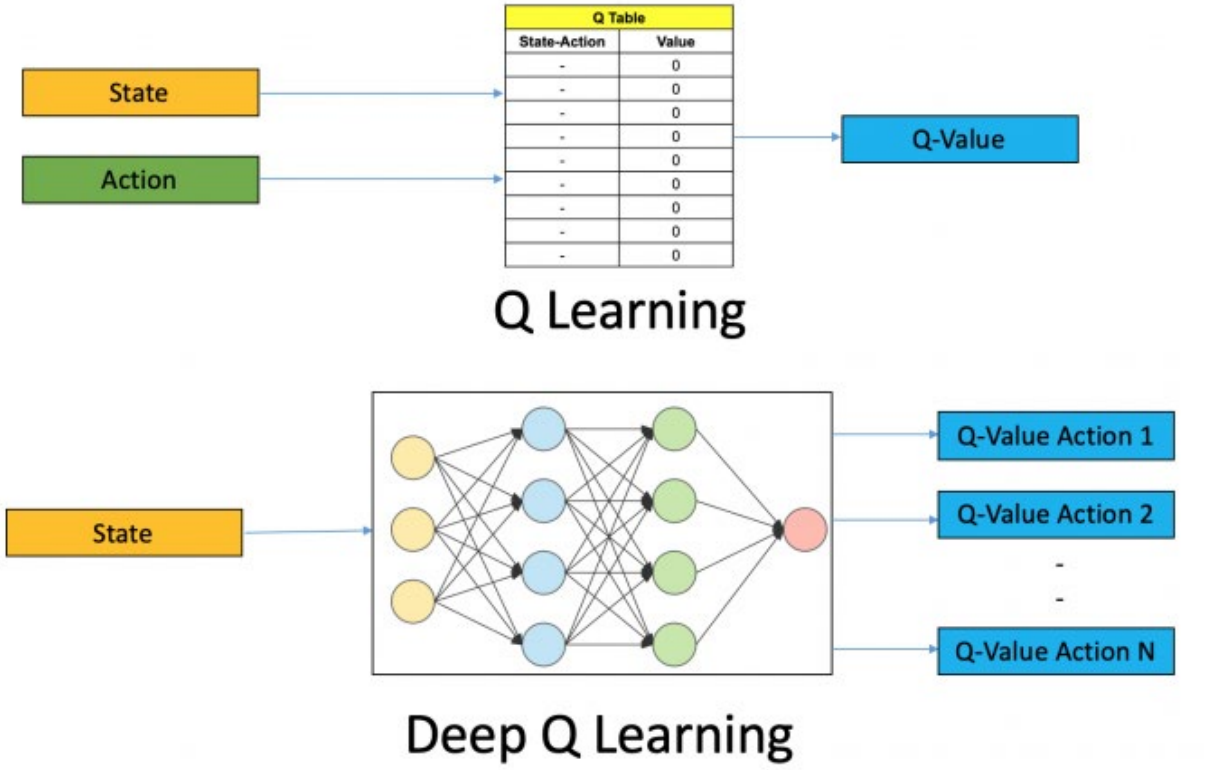


Abbildung 3-1: Vergleich von Q-Learning und Deep-Q-Learning

Für die Bewertung der Aktionen im Mühlespiel werden Netze mit Fully-Connected-Layern verwendet, da der Umfang der Eingangsdaten gering ist und die Geometrie des Mühlespiels nur bedingt mit räumlicher Information verwendet werden können. Es werden drei verschiedene Netze für die drei Spielphasen (Setz-, Zug- und Springphase) verwendet, da sich die möglichen Aktionen und damit die Ausgangsdaten zwischen den Phasen stark unterscheiden.

In Abbildung 3-2 ist die Definition der Input- und Output-Neuronen der drei Netze dargestellt. Bei allen Spielphasen bestehen die Eingangsdaten aus dem Spieler, der an der Reihe ist, der Art der durchzuführenden Aktion und dem Zustand des Spielfeldes. In der Setzphase wird zusätzlich die Anzahl der

noch zu setzenden Spielsteine des Spielers, der an der Reihe ist, übergeben. Die Output-Neuronen beinhalten die Q-Werte, die zu allen in der jeweiligen Spielphase theoretisch möglichen Zügen zugeordnet werden. Die ersten 24 Neuronen sind in allen Phasen den Aktionen zum Entfernen der Spielsteine auf dem jeweiligen Feld zugeordnet. Der Index des Neurons ist dabei direkt einer Position im Spielfeld zugeordnet, wie in Abbildung 3-3 rechts zu sehen ist. Hier wird eine andere Indizierung als in der Mühlenlogik (links) verwendet, da die Schicht der Output-Neuronen eindimensional ist. Die weiteren Neuronen ab Index 24 unterscheiden sich je nach Spielphase in ihrem Umfang und der Zuordnung zu den Aktionen.

Für die Anzahl und Größe der Hidden-Layer für die drei Netze wurde bisher kein Optimum ermittelt, weshalb es sich hier um anpassbare Parameter handelt, die aber nicht über die GUI, sondern direkt in der Software innerhalb der Klasse „DQNAgent“ eingestellt werden müssen.

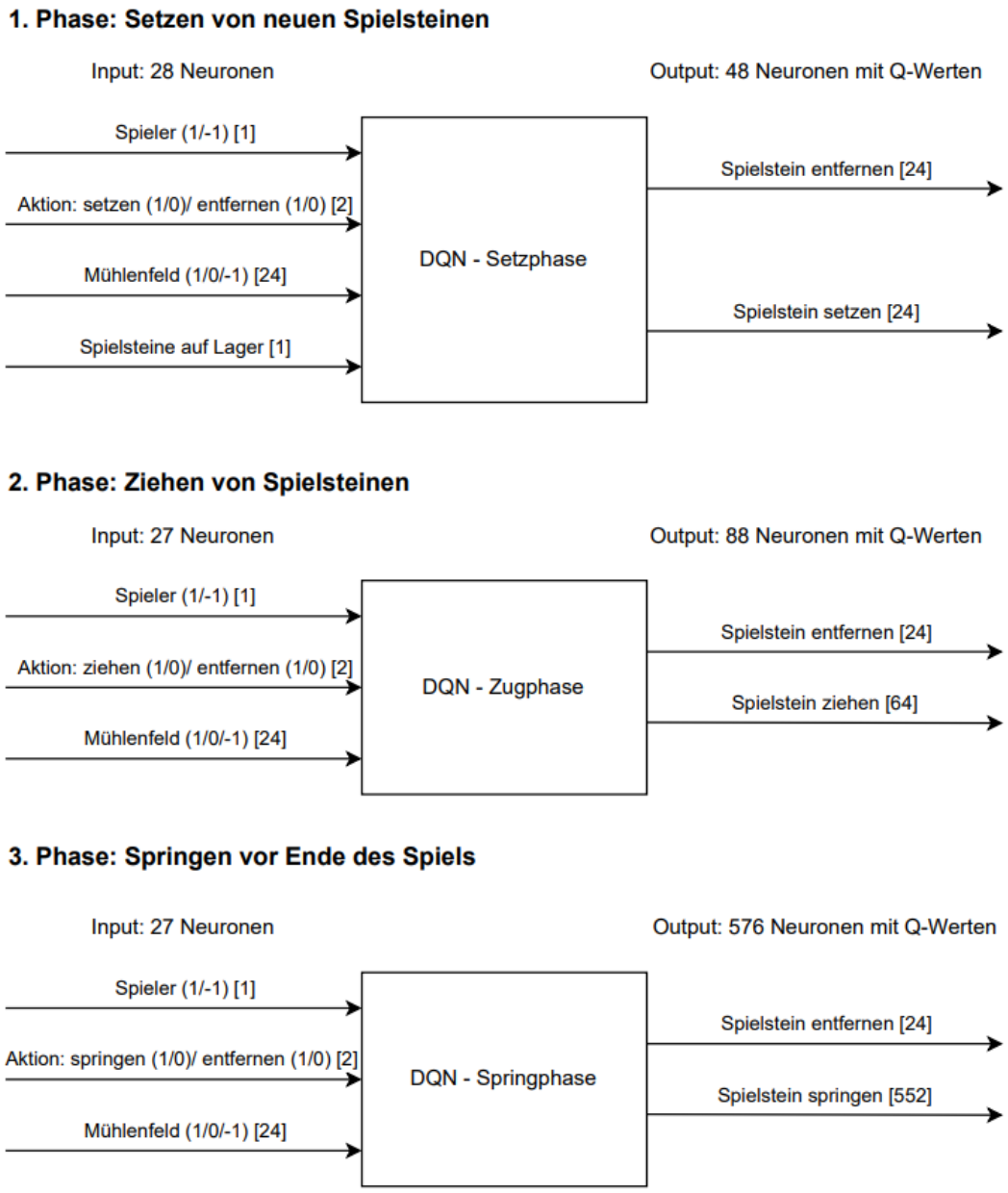


Abbildung 3-2: Ein- und Ausgänge der neuronalen Netze für die drei Spielphasen. Mögliche Werte der Neuronen sind in runden Klammern und die Anzahl der Neuronen in eckigen Klammern angegeben.

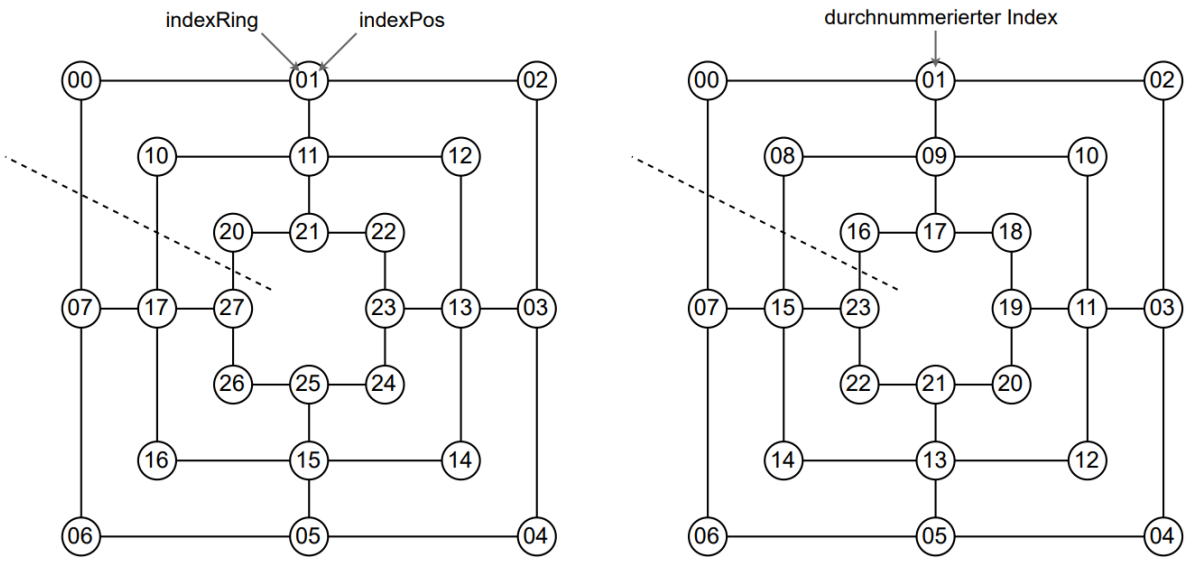


Abbildung 3-3: Vergleich der Indizierung des Mühlenfeldes für Mühlenlogik (links) und Output-Neuronen des DQN (rechts)

Zu beachten ist, dass immer nur ein Teil der Q-Werte aus den Output-Neuronen zu gültigen Aktionen gehört. Um sicherzustellen, dass immer eine gültige Aktion ausgewählt wird, wird der in Abbildung 3-4 dargestellte Ablauf verfolgt. Bei jedem Zustand des Mühlespiels werden die gültigen Züge berechnet. Damit werden aus den Q-Werten nur diejenigen ausgewählt, die zu einer möglichen Aktion gehören. Aus diesen gültigen Q-Werten wird für die Auswahl der Aktion der maximale Wert betrachtet. Der zu diesem Q-Wert gehörende Index wird dann in eine Aktion konvertiert, die der Mühlenlogik übergeben werden kann.

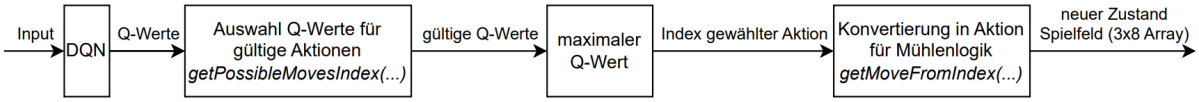


Abbildung 3-4: Ablauf der Vorhersage einer Aktion

3.2 Rewards

Die Bewertung der ausgewählten Aktionen erfolgt neben den Q-Werten für zukünftige Aktionen vor allem durch die in Tabelle 3-1 definierten Rewards, welche in der GUI angepasst werden können. Die Auswahl des Rewards ist von der aktuellen eigenen Aktion und der letzten Aktion des Gegners abhängig. Um ein sinnvolles Training zu ermöglichen, müssen die Rewards für eigene Mühlen und einen Sieg positiv und alle anderen negativ sein. Eine Optimierung der Höhe der Rewards ist bisher nicht erfolgt. Für jeden normalen Zug wird ein geringer negativer Reward veranschlagt, um lange Spiele ohne Ergebnis zu bestrafen. Die Initialwerte wurden so gewählt, dass eine Mühle in der Regel, also bei maximal 400 Zügen, die negativen Rewards durch die Züge überwiegt. Außerdem wird ein Sieg oder eine Niederlage deutlich stärker gewichtet als alle anderen Aktionen, da dies das primäre Ziel ist.

Tabelle 3-1: Rewards

Reward	Initialwert	Beschreibung
move	-1	Die letzten Züge beider Spieler hatten keine besonderen Auswirkungen.
own mill	200	Eine eigene Mühle wurde gebildet und ein Spielstein darf entfernt werden.
enemy mill	-200	Der Gegner hat eine Mühle gebildet und einen Spielstein entfernt.
win	5000	Das Spiel ist beendet und wurde gewonnen.
loss	-5000	Das Spiel ist beendet und wurde verloren.

3.3 DQN-Agent

Die Grundlage für den Deep-Q-Learning-Algorithmus bildet der DQN-Agent. Dieser ist für die Verwaltung der Netze beziehungsweise Modelle für die verschiedenen Spielphasen, die Abspeicherung der Trainingsdaten, das Trainieren der Modelle und die Vorhersage der Q-Werte zuständig. Der DQN-Agent ist als Klasse implementiert, von der mehrere Instanzen erstellt werden können, sodass auch zwei Agents gegeneinander spielen können.

Für jede Spielphase wird ein Main- und ein Target-Network angelegt, welche in der Architektur des Modells identisch sind, sich aber in den Gewichten unterscheiden. Mit dem Main-Network werden die Q-Werte des aktuellen Zustands berechnet. Um diese Q-Werte zu trainieren, werden die Q-Werte des Folgezustands benötigt, welche mit dem Target-Network berechnet werden. Die Verwendung von zwei Netzwerken dient der Stabilität. Da die Aktualisierung der Q-Werte auf der Berechnung der Q-Werte des Folgezustands basiert, werden die Gewichte des Target-Networks deutlich seltener aktualisiert als die Gewichte des Main-Networks, um eine stabile Grundlage für das Training zu schaffen. Beim Training ändern sich die Gewichte des Main-Networks nach jedem Trainingsschritt, was einem bis zu wenigen Spielzügen entspricht. Dagegen werden die gelernten Gewichte des Main-Networks immer nach einer in der GUI festlegbaren Anzahl von Spielen (Parameter updateTargetEvery) in das Target-Network übertragen.

Der DQN-Agent besitzt einen Replay-Memory für jede Spielphase, in dem alle für das Trainieren benötigten Informationen zu einem Spielzug abgespeichert werden (Tabelle 3-2). Die Größe des Speichers kann festgelegt werden. Ist der Replay-Memory voll, wird der älteste Datensatz überschrieben.

Tabelle 3-2: Parameter, die im Replay-Memory abgespeichert werden

Parameter	Beschreibung
current_state	Eingangsvektor für das DQN mit Informationen über aktuellen Zustand der Mühlenlogik
action	Index der ausgewählten Aktion / des gewählten Output-Neurons
reward	Reward für die gewählte Aktion
player	Spieler der die Aktion durchgeführt hat
new_current_state	Eingangsvektor des auf die gewählte Aktion folgenden Zustands, an dem der gleiche Spieler wieder an der Reihe ist
new_field	Zustand des Spielfelds (3x8 Array) im Folgezustand
new_move	True: Spielstein ziehen (set/shift/jump) als folgende Aktion False: Spielstein entfernen als folgende Aktion
new_phase	Spielphase, in der sich das Spiel im Folgezustand befindet
done	True: das Spiel ist nach der durchgeführten Aktion beendet, der Folgezustand ist irrelevant

Beim Training wird eine der in der GUI wählbaren Batchgröße entsprechende Anzahl von Spielzügen zufällig aus dem Replay-Memory ausgewählt. Das Training wird nur durchgeführt, wenn eine Mindestanzahl von Zügen im Speicher zur aktuellen Spielphase vorhanden ist, die mindestens so groß wie die Batchgröße sein muss. Außerdem kann eingestellt werden, dass das Training nur alle x Spielzüge durchgeführt wird (Parameter trainEveryXSteps), um die Batchgröße bei gleichbleibendem Trainingsumfang erhöhen zu können. Damit kann das Training insbesondere bei Verwendung einer GPU beschleunigt werden.

Zu Beginn des Trainings werden die Q-Werte für alle aktuellen Zustände mit dem Main-Network der aktuellen Spielphase und die Q-Werte für alle Folgezustände mit den Target-Networks berechnet. Dabei ist zu beachten, dass die Folgezustände aus verschiedenen Spielphasen stammen können. Wird zum Beispiel die Setzphase trainiert, kann sich der Spieler in einigen Folgezuständen bereits am Anfang der Zugphase befinden. Deshalb muss an dieser Stelle besonders darauf geachtet werden, dass die Q-Werte der Folgezustände jeweils mit dem Target-Network zur korrekten Spielphase berechnet werden.

Im Anschluss werden die Q-Werte zu den ausgewählten Aktionen mit der Formel

$$Q_{neu} = R + \lambda \cdot Q_{future,max}$$

aktualisiert. In der Software ist R als „reward“ und λ als „discount“ bezeichnet. Aus den Q-Werten der Folgezüge wird der maximale Wert verwendet, der zu einem gültigen Zug gehört, also der maximal erreichbare Wert im Folgezug. Der Discount-Faktor gewichtet den Einfluss der Folgezüge auf die Bewertung der aktuellen Züge. Falls der Spielzug zum Ende des Spiels geführt hat, fällt der hintere Teil der Formel weg und der neue Q-Wert wird allein durch den Reward bestimmt. Nach diesem Schritt liegt ein Trainingsdatensatz vor, der die Q-Werte zu allen Aktionen für alle aktuellen Zustände aus dem Batch enthält. Nur die Q-Werte zu den ausgewählten Aktionen wurden verändert. Das Main-Network

wird mit den aktuellen Zuständen als Eingangsvektoren und den Q-Werten als gewünschte Vorhersagen trainiert. Die im verwendeten Modell definierte Learning-Rate bestimmt hier, wie stark sich die vorhergesagten Q-Werte durch den Trainingsschritt verändern.

3.4 DQN-Training

Im DQN-Training werden Spiele durchgeführt, in denen 2 Spieler gegeneinander spielen, um Trainingsdaten zu generieren. Für die Vorhersagen der Spielzüge für beide Spieler wird derselbe DQN-Agent verwendet. Die von beiden Spielern erzeugten Trainingsdaten werden verwendet, um das Training im DQN-Agent durchzuführen. Auf diese Weise lernen die Modelle die Vorhersage der Spielzüge beider Spieler. Zudem werden im Vergleich zu einem Konzept, bei dem zwei verschiedene DQN-Agents gegeneinander spielen, doppelt so viele Trainingsdaten für den DQN-Agent generiert, und es wird kein zweites Modell unnötig trainiert. Zur Erhöhung der Spielgeschwindigkeit werden außerdem mehrere Spiele gleichzeitig gespielt. Damit muss die Vorhersage der Q-Werte nicht für jeden Spielzug einzeln durchgeführt werden, sondern kann für Spielzüge in mehreren Spielen gleichzeitig erfolgen. Da sich die Dauer der Spiele unterscheidet, bedeutet dies aber, dass sich die verschiedenen Spiele in unterschiedlichen Phasen befinden, was bei der Vorhersage der Q-Werte berücksichtigt werden muss. Damit sind beim Training bis zu drei Vorhersagen von Q-Werten mit den drei Modellen zu den Spielphasen notwendig. Um das Training zu beschleunigen, sollte die Anzahl an parallel ablaufenden Spielen deshalb größer als drei, also beispielsweise 20 bis 30 sein. Die beiden Spieler sind auch nicht immer abwechselnd an der Reihe, da auf das Schließen einer Mühle ein Zug desselben Spielers zum Entfernen eines Spielsteins folgt. Deshalb muss in jedem Schritt aus der Mühlenlogik abgefragt werden, welcher Spieler an der Reihe ist.

Ein naheliegender Ansatz zur Implementierung des DQN-Trainings wäre eine Schleife, in der immer der nächste Spielzug durchgeführt wird. Um aber die Kommunikation zu anderen Prozessen und die Anzeige des Fortschritts in der GUI zu ermöglichen, ist die Software so aufgebaut, dass jeder Spielzug, allerdings in allen gleichzeitig ablaufenden Spielen zusammen, im Training durch einen separaten Funktionsaufruf durchgeführt wird. Damit kann das Training auch jederzeit von außen angehalten werden.

Von großer Bedeutung im Training ist auch der Epsilon-Greedy-Algorithmus. Der Parameter $0 < \varepsilon < 1$ bestimmt dabei, wie der nächste Zug ausgewählt wird. Mit einer Wahrscheinlichkeit von ε wird ein zufälliger Zug aus den erlaubten Aktionen ausgewählt. Ansonsten wird der Zug durch den DQN-Agent berechnet. Der Wert von ε startet in der Regel bei 1 und wird dann nach dem Ende jedes Spiels gemäß

$$\varepsilon_{neu} = decay \cdot \varepsilon_{alt}$$

verringert, bis ein eingestellter minimaler Wert erreicht wird. Der Parameter decay sollte so eingestellt werden, dass sich ε kontinuierlich über den gesamten Trainingszeitraum verringert und nicht zu früh seinen Minimalwert erreicht. Durch den Epsilon-Greedy-Algorithmus wird erreicht, dass zu Beginn des Trainings möglichst viele verschiedene Züge ausprobiert werden, um einen umfangreichen Zustandsraum abzudecken. Im späteren Trainingsverlauf werden dann verstärkt die Vorhersagen der Modelle verwendet und bewertet.

Ein Schritt im Training läuft am Beispiel der Instanz für ein Spiel folgendermaßen ab:

1. Wenn es erforderlich ist, wird ein neues Spiel gestartet. Dabei wird der erste Zug des Spiels direkt durchgeführt, ohne Trainingsdaten abzuspeichern. Als Trainingsdaten sind immer zwei aufeinanderfolgende Zustände erforderlich, bei denen der gleiche Spieler an der Reihe ist. Die ersten Trainingsdaten eines Spiels sind demnach verfügbar, wenn der anfangende Spieler zum zweiten Mal an der Reihe ist. Deshalb muss der Start eines Spiels separat behandelt werden.

2. Im Anschluss werden abhängig vom Epsilon-Greedy-Algorithmus die Q-Werte zum aktuellen Zustand berechnet und damit die nächste Aktion bestimmt oder eine zufällige Aktion ausgewählt.
3. Der Eingangsvektor für das DQN zum aktuellen Zustand und die ausgewählte Aktion des Spielers werden für den Durchlauf abgespeichert, in dem derselbe Spieler das nächste Mal an der Reihe ist.
4. Der gewählte Spielzug wird der Mühlenlogik übergeben und damit durchgeführt.
5. Der aktuelle Zustand für den neuen Spieler wird aus der Mühlenlogik bestimmt und der Reward für die letzte Aktion dieses Spielers, die also zu diesem Zustand geführt hat, wird berechnet.
6. Alle benötigten Informationen zum aktuellen Zustand des Spielers und dem vorherigen Zustand des Spielers sowie der Reward werden im Replay-Memory zur Spielphase des vorherigen Zustands abgespeichert.
7. Das Training im DQN-Agent wird aufgerufen. Dieses ist allerdings unabhängig vom aktuellen Spielzug, da zufällige Spielzüge aus dem Replay-Memory verwendet werden. Außerdem entscheidet der DQN-Agent, ob alle notwendigen Bedingungen erfüllt sind, um tatsächlich ein Training durchzuführen.
8. Wenn ein Spieler gewonnen hat, werden zusätzlich Trainingsdaten für den Gegenspieler erstellt, bei denen dessen Niederlage berücksichtigt wird. Ohne diesen Schritt würde der Gegenspieler keine Kenntnis der Niederlage erhalten, da er nicht mehr an die Reihe kommt.
9. Das Spiel wird beendet, wenn ein Spieler gewonnen hat oder die eingestellte maximale Anzahl von Zügen erreicht wurde. Durch die zweite Bedingung werden lange Spiele mit wenig Information über besonders gute oder schlechte Züge vermieden.

Während des Trainings wird eine Statistik erstellt, die die Gesamtrewards der beiden Spieler und die Anzahl der Züge zu jedem Spiel speichert. Bei Bedarf kann ausgewählt werden, dass im Trainingsprozess nur einzelne Spielphasen trainiert werden.

3.5 DQN-Validierung

Die DQN-Validierung wird verwendet, um zu bewerten, wie gut ein DQN-Modell ist. Die Validierung läuft ähnlich zum DQN-Training ab. Vor allem das Durchführen von vielen Spielen gleichzeitig ist hier für eine schnelle Durchführung der Validierung relevant. Es bestehen folgende Unterschiede zum DQN-Training:

- Es werden zwei DQN-Agents verwendet, die gegeneinander spielen und jeweils die Hälfte der Spiele starten. Der eine Agent verwendet das trainierte Modell, das validiert werden soll. Der zweite Agent verwendet ein trainiertes Referenzmodell oder ein zufälliges Modell.
- Für die Vorhersage der Spielzüge werden prinzipiell immer die Modelle verwendet, und es existiert kein Epsilon-Greedy-Algorithmus. Dabei ergibt sich das Problem, dass die zwei Agents immer dasselbe Spiel gegeneinander spielen. Um dies zu verhindern, gibt es zwei Möglichkeiten. Zum einen kann eingestellt werden, dass der erste Zug im Spiel immer zufällig ist, wodurch verschiedene Ausgangssituationen geschaffen werden. Zum anderen kann aktiviert werden, dass die Züge des Referenzmodells komplett zufällig durchgeführt werden.
- Es müssen keine Daten für das Training abgespeichert werden, da kein Training durchgeführt wird.
- Die Gesamtrewards über alle Spiele, die als Statistik dienen, werden nicht den Spielern, sondern den Agents zugeordnet. Dies ist notwendig, da ein Agent in verschieden Spielen auch als unterschiedliche Spieler spielt.

- Es werden zusätzlich die Anzahl der Siege, Niederlagen und Unentschieden des DQN-Agents mit dem zu validierenden Modell bestimmt.

3.6 Verwendung der DQN-Modelle als Spieler

In Form einer Klasse ist der DQN-Player implementiert, der trainierte Modelle verwendet, um die Züge von einem der beiden Spieler auszuwählen. Es können zwei Instanzen der Klasse verwendet werden, um zwei DQN-Player mit unterschiedlichen Modellen gegeneinander spielen zu lassen. Die einzige Aufgabe des DQN-Players besteht darin, aus dem vollständigen Zustand der Mühlenlogik mithilfe des vorliegenden Modells zur passenden Spielphase den nächsten Zug zu bestimmen und zurückzugeben. Die Abfrage des Zustands der Mühlenlogik und die Ausführung des Spielzuges muss außerhalb des DQN-Players durchgeführt werden.

4 Aufbau Software

Damit die in diesem Projekt notwendigen Aufgaben parallel abgearbeitet werden können, besteht die Software aus verschiedenen Prozessen. Dies ist insbesondere notwendig, damit die GUI auch bedient werden kann, während z.B. das Training läuft. Alle Prozesse werden über Prozess 2: Main/Logik miteinander verbunden. Die Kommunikation zwischen den Prozessen erfolgt über sogenannte Warteschlangen (englisch Queue). Die sind einfach FIFO-Puffer, die zur Kommunikation zwischen Prozessen verwendet werden. Jeder Prozess (ausgenommen der Prozess 2: Main/Logik) besitzt einen Input-Queue und einen Output-Queue, jeweils für Ein- und Ausgangsdaten des Prozesses. Der Prozess 2: Main/Logik besitzt dann für jeden weiteren Prozess jeweils zwei Queue (je ein Input- und ein Output). Der Aufbau der Software ist in Abbildung 4-1 skizziert.

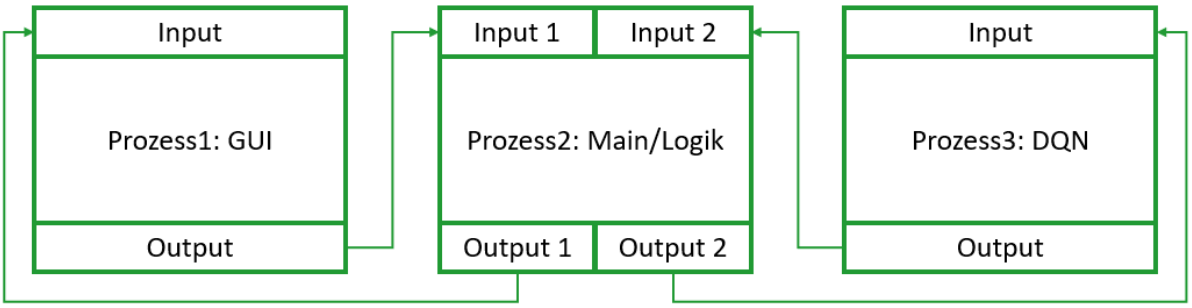


Abbildung 4-1 Skizze Aufbau Software

Anmerkung zu Spyder: Die Entwicklungsumgebung Spyder kommt mit dem Multiprocessing stark an seine Grenzen. Aufgrund des Multiprocessing war Debugging mit V5.4.2 von Spyder nicht möglich. Je nach Zielsetzung und Vorgehen weiterer Gruppen, sollte überlegt werden, ob nicht eine professionellere Entwicklungsumgebung wie z.B. PyCharm mehr Sinn macht.

4.1 Aufbau Nachrichten zwischen Prozessen

Zur Kommunikation zwischen den Prozessen wird ein Array der Länge größer gleich eins verwendet, der folgenden Aufbau besitzt:

[Nachricht, Beliebig lange Payload]

In Prozess 1: GUI werden die Nachrichten in der Funktion *UpdateGUI()* ausgewertet, in allen anderen Prozessen findet die Auswertung in der *run()* Funktion statt. Die Funktion *UpdateGUI()* wird alle 33 ms aufgerufen, während innerhalb der *run()* Funktionen eine Dauerschleife ist. Das ist notwendig, da in der GUI die Logik der GUI außerhalb der *UpdateGUI()* existiert, während in den anderen Prozessen alles in der *run()* Funktion bearbeitet wird. Die existierenden Nachrichten werden in den Kapiteln der jeweiligen Prozesse beschrieben

4.2 Prozess 1: GUI

In diesem Prozess ist die GUI und ihre Logik realisiert. Zur GUI gehören mehrere Dateien, diese sind in Tabelle 4-1 aufgelistet und beschrieben.

Tabelle 4-1: Dateien von Prozess 1: GUI

Datei	Beschreibung
PyQtDesinger_GUI.ui	Datei die mit dem QtDesigner verändert wird. Enthält die Rohform der GUI
PyQtDesinger_GUI.py	Wird aus der PyQtDesinger_GUI.ui erzeugt und enthält ausführbaren Python Code.
Ressourcen.qrc	Enthält die Bilder für die GUI. Wird im QtDesigner erstellt und verändert
Ressourcen_rc.py	Ressourcen.qrc in Python übersetzt. Kann mit dem Befehl *pyrcc5 -o Ressourcen_rc.py Ressourcen.qrc* erzeugt werden
GUI_Klasse.py	Import die PyQtDesinger_GUI.py und integriert in die GUI die notwendige Logik. Auch die Kommunikation zu Prozess 2: Main/Logik ist hier realisiert

Für die Erstellung der GUI wird das Qt-Framework verwendet. Da die Installation des Qt-Designers mühsam war, wird dieser mitsamt dem Quellcode bereitgestellt.

ACHTUNG: Da Spyder ebenfalls Qt nutzt, sollten keine Updates an PyQt vorgenommen werden, da ansonsten Spyder nicht mehr funktioniert.

4.2.1 Input-Nachrichten

Folgenden Nachrichten gibt es als Input. Die Output-Nachrichten sind die Input-Nachrichten von Prozess 2: Main/Logik.

Tabelle 4-2 Input-Nachrichten von Prozess 1: GUI

Nachricht	Payload	Beschreibung
SetState	millField nextAction InStockTokens	Setzt den aktuellen Zustand des Spielfeldes, welche Aktion als nächstes erfolgt und wie viele Spielsteine die Spieler noch setzen können.
StopTraining		Stoppt das Training.
TrainingRunning	numberOfGames AllStepsPerGame	Wenn ein Training läuft, wird die Anzahl aller beendeten Spiele mitgeteilt, sowie die Anzahl der Züge aller bisher beendeten Spiele.
TrainingFinished		Nachricht, dass das Training beendet wurde.

LoadTrainingModel_Success	FilePath	Wenn ein Modell, das trainiert werden soll, erfolgreich geladen werden konnte, wird der Dateipfad des geladenen Modells zurückgegeben.
LoadSettings	Settings	Nachricht, um die übergebenen Einstellungen zu laden.
ShowError	ErrorMessage	Anzeige eines Fehlertextes als Pop-Up.
StopValidation		Stoppt die Validierung.
ValidationRunning	numberOfGames wins losses draws	Wenn eine Validierung läuft, wird die Anzahl der beendeten Spiele, die Siege, Niederlagen und Unentschieden mitgeteilt.
ValidationFinished	wins losses draws	Nachricht, dass die Validierung abgeschlossen ist, mit dem Endstand der Siege, Niederlagen und Unentschieden
LoadValidationModel_Success	FilePath	Wenn ein Modell, das validiert werden soll, erfolgreich geladen werden konnte, wird der Dateipfad des geladenen Modells zurückgegeben.
LoadReferenceModel_Success	FilePath	Wenn das Referenzmodell, erfolgreich geladen werden konnte, wird der Dateipfad des geladenen Modells zurückgegeben.

4.2.2 Änderungen an der GUI umsetzen

Die GUI wird mit dem QtDesigner erstellt und in ihrem Aussehen verändert. Sobald die Änderung erfolgt wurde, kann folgendermaßen der Python Quellcode erzeugt werden:

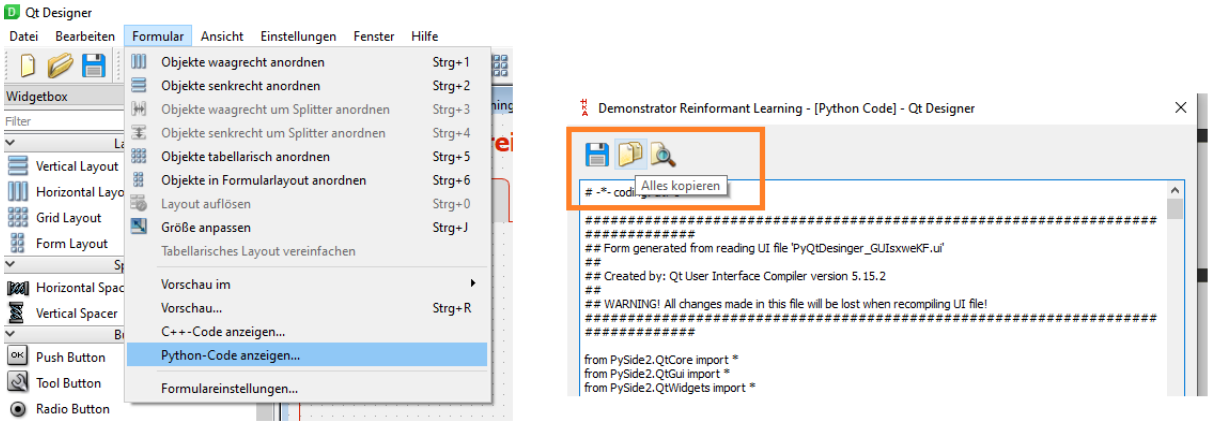


Abbildung 4-2: Python Code im QtDesigner erzeugen

Der erzeugte Quellcode muss dann in die Datei PyQtDesinger_GUI.py kopiert werden. Anschließend muss in den Zeilen 11 bis 13 PySide2 durch PyQt5 ersetzt werden.

Werden neue Bilder bei den Ressourcen hinzugefügt, muss in einem Terminal in den Ordner der Ressourcen.qrc navigiert werden und dann mit dem Befehl *pyrcc5 -o Ressourcen_rc.py Ressourcen.qrc* die Datei übersetzt werden.

4.2.3 Training - Diagramm

Im Fenster *Training* wird ein Diagramm angezeigt. Es werden die mittleren Abweichungen der vorhergesagten Q-Werte zu den neu berechneten Q-Werten aus allen Trainingsschritten jeder Episode angezeigt. Daraus kann abgelesen werden, wie stark sich das Modell während des Trainings verändert.

In Abbildung 4-3 ist das Diagramm beispielhaft gezeigt. Mithilfe des Diagramms kann dann, während dem Training analysiert werden, wie stark sich die Gewichte und damit das Modell verändert.

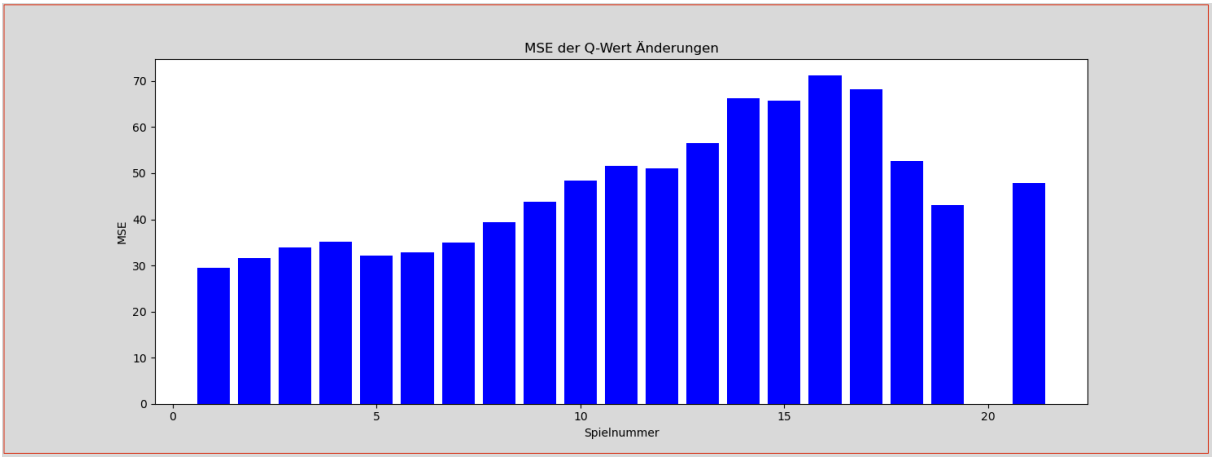


Abbildung 4-3 Training - Diagramm

4.3 Prozess 2: Main/Logik

Dieser Prozess verknüpft alle anderen Prozesse miteinander und stellt außerdem die Logik für das Mühle Spiel der GUI bereit.

4.3.1 Input-Nachrichten von Prozess 1: GUI

In Tabelle 4-3 sind die Input Nachrichten aufgelistet, die der Prozess 2: Main/Logik von Prozess 1: GUI bekommt.

Tabelle 4-3 Input-Nachrichten von Prozess 2: Main/Logik die von Prozess 1: GUI kommen

Nachricht	Payload	Beschreibung
Terminate		Signalisiert, dass die GUI geschlossen wurde. Beendet den Prozess.
GamePosClicked	indexRing indexPos	Position auf dem Spielfeld wurde geklickt.
StartTraining	*data*	Training soll gestartet werden. Wird an Prozess 3: DQN weitergereicht. Payload wird nur durchgereicht.
StopTraining	*data*	Training soll gestoppt werden. Wird an Prozess 3: DQN weitergereicht. Payload wird nur durchgereicht.
SaveSettings	Settings	Speichert die übergebenen Einstellungen in *settings.dat*
LoadTrainingModel	*data*	Modell zum Trainieren soll geladen werden. Wird an Prozess 3: DQN weitergereicht. Payload wird nur durchgereicht.
SaveTrainingModel	*data*	Trainiertes Modell soll gespeichert werden. Wird an Prozess 3: DQN weitergereicht. Payload wird nur durchgereicht.
ResetTrainingModel	*data*	Trainiertes Modell soll zurückgesetzt werden. Wird an Prozess 3: DQN weitergereicht. Payload wird nur durchgereicht.
AIPlayerHandler		Während Spiel zyklische Nachricht, dass der KI-Spieler eventuell einen Zug machen muss. Benötigt, damit Zuggeschwindigkeit des KI-Spielers reduziert wird.
StartValidation	*data*	Validierung soll gestartet werden. Wird an Prozess 3: DQN weitergereicht. Payload wird nur durchgereicht.

StopValidation	*data*	Validierung soll gestoppt werden. Wird an Prozess 3: DQN weitergereicht. Payload wird nur durchgereicht.
LoadValidationModel	*data*	Modell zur Validierung soll geladen werden. Wird an Prozess 3: DQN weitergereicht. Payload wird nur durchgereicht.
LoadReferenceModel	*data*	Referenzmodell soll geladen werden. Wird an Prozess 3: DQN weitergereicht. Payload wird nur durchgereicht.

4.3.2 Input-Nachrichten von Prozess 3: DQN

In Tabelle 4-4 sind die Input Nachrichten aufgelistet, die der Prozess 2: Main/Logik von Prozess 3: DQN bekommt.

Tabelle 4-4 Input-Nachrichten von Prozess 2: Main/Logik die von Prozess 3: DQN kommen

Nachricht	Payload	Beschreibung
TrainingRunning	*data*	Signalisiert, dass Training läuft. Wird an Prozess 1: GUI weitergereicht. Payload wird nur durchgereicht.
TrainingFinished	*data*	Signalisiert, dass Training beendet ist. Wird an Prozess 1: GUI weitergereicht. Payload wird nur durchgereicht.
ShowError	*data*	Signalisiert, dass ein Fehler aufgetreten ist. Wird an Prozess 1: GUI weitergereicht. Payload wird nur durchgereicht.
LoadTrainingModel_Success	*data*	Trainingsmodell erfolgreich geladen. Wird an Prozess 1: GUI weitergereicht. Payload wird nur durchgereicht.
ValidationRunning	*data*	Validierung läuft. Wird an Prozess 1: GUI weitergereicht. Payload wird nur durchgereicht.
ValidationFinished	*data*	Validierung wurde beendet. Wird an Prozess 1: GUI weitergereicht. Payload wird nur durchgereicht.
LoadValidationModel_Success	*data*	Validierungsmodell wurde erfolgreich geladen. Wird an Prozess 1: GUI weitergereicht. Payload wird nur durchgereicht.

LoadReferenceModel_Success	*data*	Referenzmodell wurde erfolgreich geladen. Wird an Prozess 1: GUI weitergereicht. Payload wird nur durchgereicht.

4.3.3 Logik Spielablauf

Welcher Spieler Mensch und welcher Spieler eine KI ist, kann in den Einstellungen verändert werden. Wird dann ein Spiel gestartet, werden die menschlichen Spielzüge anhand von geklickten Spielfeldpositionen ausgewertet und an die Mühlenlogik übergeben (mehr zur Mühlenlogik in Kapitel 2 Mühlenlogik). Danach wird von der Mühlenlogik der aktuelle Zustand wieder abgefragt und an die GUI übergeben. Da die Mühlenlogik keine ungültigen Züge akzeptiert (z.B. ein Spielstein auf ein Feld zu setzen, auf dem bereits ein Spielstein ist), wird so sichergestellt, dass kein ungültiger Zug ausgeführt wird.

Ist in mindestens ein Spieler eine KI, wird einmal direkt nach dem menschlichen Zug ein KI-Zug ausgeführt. Auch bei KI-Zügen wird identisch, wie bei menschlichen Zügen durch die Mühlenlogik sichergestellt, dass keine ungültigen Züge ausgeführt werden.

Zusätzlich wird zyklisch von der GUI eine Nachricht gesendet, um eventuell einen weiteren KI-Zug durchzuführen. Dies ist nötig, damit bei Spielzügen, bei der zwei aufeinander folgende Aktionen nötig sind (z.B. nach dem Bilden einer Mühle muss noch ein zu entfernender Spielstein ausgewählt werden), der KI-Spieler beide Aktionen ausführt. Außerdem wird es benötigt, damit bei einem Spiel KI gegen KI ein menschlicher Beobachter dem Spiel zuschauen kann und es nicht innerhalb von Millisekunden beendet wird. Der Timer für das zyklische Aufrufen ist im Prozess 1: GUI realisiert, da PyQt passende Timer-Funktionen besitzt.

4.4 Prozess 3: DQN

In diesem Prozess finden das Training und die Validierung der KI statt. Wie das Training und die Validierung funktioniert, ist in den Kapiteln 3.4 DQN-Training und 3.5 DQN-Validierung genauer beschrieben.

4.4.1 Input-Nachrichten

Nachricht	Payload	Beschreibung
Terminate		Signalisiert, dass Programm beendet werden soll.
StartTraining	TrainingsParameter	Startet das Training mit den übergebenen Parametern. *TrainingsParameter* ist ein Struct.
StopTraining		Stoppt das Training.
LoadTrainingModel	FilePath	Lädt ein Modell, das Trainiert werden soll, aus dem übergebenen Dateipfad.

SaveTrainingModel	FilePath	Speichert das Trainingsmodell an dem übergebenen Dateipfad.
ResetTrainingModel		Setzt das Trainingsmodell zuück.
StartValidation	ValidationParameter	Startet die Validierung mit den übergebenen Parametern. *ValidationParameter* ist ein Struct.
StopValidation		Stoppt die Validierung
LoadValidationModel	FilePath	Lädt ein Modell, das validiert werden soll, aus dem übergebenen Dateipfad.
LoadReferenceModel	FilePath	Lädt ein Modell, das als Referenzmodell verwendet werden soll, aus dem übergebenen Dateipfad.

4.4.2 Speichern und Laden von trainierten KI-Modellen

Da für jede Spielphase ein eigenes Neuronales Netz existiert, gibt es immer drei Modelle, die gespeichert und geladen werden müssen. Es wurde versucht, alle Modelle in einer einzigen Datei zu speichern, allerdings war das leider erfolglos. Aus diesem Grund werden drei Dateien in einem Ordner gespeichert. Die drei Dateien heißen immer *model1.h5*, *model2.h5*, *model3.h5*.

Jede Datei enthält das Modell aus einer Spielphase (Mehr zu den Modellen siehe Kapitel Deep-Q-Learning-Algorithmus).

4.5 Wie können weitere Prozesse integriert werden?

Um weitere Prozesse zu integrieren, muss eine neue Klasse für diesen Prozess erstellt werden. Als Init-Parameter wird eine Input-Queue und eine Output-Queue benötigt. Außerdem sollte in der __init__ Funktion direkt der Aufruf zu einer self.run() Funktion sein. Dies macht den Start des Prozesses einfacher. In der self.run() Funktion wird dann die Kommunikation zu Prozess 2: Main/Logik realisiert, sowie die sonstige Funktionalität des Prozesses. Als Beispiel/Vorlage siehe *class Process2_Logic()* und *class Process3_DQN()* in *main.py*.

Wichtig: Der Befehl „Terminate“ muss realisiert werden, damit der Prozess am Ende richtig beendet wird.

Außerdem muss der Prozess 2: Main/Logik um die entsprechenden Warteschlangen zur Kommunikation erweitert werden.

Der neue Prozess mit den neuen Warteschlangen muss letztendlich noch in der *main()*-Funktion hinzugefügt werden.

5 Fazit

Im Zuge dieser Projektarbeit konnte Q-Learning und insbesondere das Deep-Q-Learning besser verstanden werden. Es wurde erkannt, dass der klassische Q-Learning Ansatz, mit einer Tabelle, in der Praxis selten anwendbar ist und stattdessen die Deep-Q-Learning Variante deutlich besser skalierbar und anwendbar ist. Allerdings ergeben sich durch den aktuellen Ansatz noch Schwachstellen in Bezug auf die Performance des Trainings. Dadurch, dass im Training und im Spiel sehr viele Vorhersagen benötigt werden, wird ein Großteil der benötigten Zeit nicht für das eigentliche Training, sondern für die Vorhersage der Q-Werte verwendet. Diese Q-Werte sind sowohl beim Erzeugen der Trainingsspielzüge als auch beim Erzeugen der gewünschten Q-Werte als eigentliche Trainingsdaten notwendig.

Mit dieser Projektarbeit wurde eine Grundlage geschaffen, auf die weitere Arbeiten aufbauen können. Die Software wurde bewusst modular aufgebaut, sodass weitere Aspekte, wie z.B. ein Roboter relativ einfach integriert werden können. Auch gibt es noch Verbesserungsmöglichkeiten, wie z.B. das Training des neuronalen Netztes und auch der Aufbau des zugrunde liegenden neuronalen Netzes.

Die besten bisher trainierten Modelle konnten bei der Validierung 87 % Siege, 6% Niederlagen und 6% Unentschieden (Pfad: *...\Modelle\4 Hidden Layers\2000_Spiele_LearningRate_10-5*) beziehungsweisen 73% Siege, 2% Niederlagen und 25% Unentschieden (Pfad: *...\Modelle\4 Hidden Layers\md5_10000Spiele_lr10-5*) im Vergleich zu einem Spieler mit zufälligen Zügen erreichen.