Tetris Spielen mit Deep Reinforcement Learning

Daniel Bogacz
Hochschule Mannheim
Paul-Wittsack-Str. 10
68163 Mannheim
Email: 1727758@stud.hs-mannheim.de

Simon Schneider
Hochschule Mannheim
Paul-Wittsack-Str. 10
68163 Mannheim
Email: 1725441@stud.hs-mannheim.de

***Abstract*—Deep Reinforcement Learning, eine Kombination von Reinforcement Learning und Deep Learning, ist besonders effektiv im spielerischen Erlernen von Regeln, um seinen Erfolg zu maximieren. In diesem Paper wird eine Deep Reinforcement Implementierung vorgestellt, dessen künstlicher Agent eine modifizierte Version des Spiels Tetris autonom zu spielen und Punkte zu erzielen lernt. Dabei sollen Reihen vervollständigt und das Erreichen eines Endzustands, eines Game Overs, vermieden werden. Beim verwendeten Neuronalen Netz handelt es sich um ein Convolutional Neural Network (CNN), welches für eine Observierung des Spielfeldes eine Aktion ausgibt, für welche es die höchste zukünftigen Belohnungen voraussagt. Diese Belohnungen werden nicht nur nach dem Erreichen des Endzustands oder dem Platzieren eines Steines, sondern auch nach jedem Zeitschritt an den Agenten zurückgegeben. Im Laufe des Trainings zeigt der Agent eine stetige Verbesserung, welche noch weiteres Potential über die bisher durchgeführte Trainingsdauer andeutet.**

I. Einleitung

In den letzten Jahren konnten kontinuierlich beeindruckende Leistungen und Erfolge auf dem Gebiet des maschinellen Lernens erzielt werden. Besonders die zwei Bereiche Deep Learning (DL) und Reinforcement Learning (RL) haben neue Durchbrüche erreicht. Obwohl normales RL relativ erfolgreich ist, muss der Zustand manuell erstellt werden, und der Algorithmus hat Schwierigkeiten, mit komplexen Szenarien umzugehen. Dies führt oft zu unbefriedigenden Ergebnissen. DL stößt ebenfalls an Grenzen, wenn ein kontinuierlicher Zustand und ein hochdimensionaler Aktionsraum vorliegen. Es hat hauptsächlich die Fähigkeit, hochdimensionale Daten zu verarbeiten und automatisch zu lernen. [1]

Deep Reinforcement Learning (DRL) ist die Kombination von RL und DL. Dieses Forschungsgebiet konnte mehrere Aufgaben mit komplexen Entscheidungen lösen, die bisher für eine Maschine unerreichbar waren. DRL wurde aufgrund seiner Effizienz und Erfolgs bei der Bewältigung anspruchsvoller Probleme mit sequentieller Entscheidungsfindung immer beliebter. Diese Kombination aus DL und RL, auch DRL genannt, ist am nützlichsten in Situationen und Problemen, wo ein hochdimensionaler Zustandsraum existiert. DRL ist dank seiner Fähigkeit, verschiedene Abstraktionen aus Daten zu lernen, bei komplizierten Aufgaben mit geringeren Vorkenntnissen erfolgreich. Zum Beispiel kann ein DRL Agent mithilfe von visuellen Wahrnehmungseingaben lernen, die aus tausenden Pixel bestehen [2]. Damit können menschliche Problemlösungsfähigkeiten selbst im hochdimensionalen Raum nachgeahmt werden. [3]

In diesem Paper wird ein DRL Algorithmus verwendet, um einen Agenten zu trainieren, welcher das Videospiel Tetris erlernt. Das Spiel ist selbst implementiert und wird mithilfe der Zahlen 0 und 1 im Terminal bzw. in der Kommandozeile angezeigt. Im folgenden Kapitel werden zunächst die Grundlagen vorgestellt, die benötigt werden, um DRL und infolgedessen die Implementierung zu verstehen. Anschließend wird die Theorie und die Architektur von DRL behandelt. Darauf folgt die modifizierte Implementierung, sodass der Agent das Spielen von Tetris erlernen kann, und dessen Ergebnisse. Abschließend wird ein kurzer Ausblick für weitere Arbeiten an dieser Implementierung gegeben.

II. Erforderliche Vorkenntnisse

Dieses Kapitel gibt einen Einstieg in die Grundlagen von DL und RL, den beiden zugrunde liegenden Bereichen des maschinellen Lernens für DRL und die in diesem Paper vorgestellte Implementierung. Für diese Implementierung sind im Bereich DL vor allem CNN relevant, daher wird speziell auf diese Variante der Neuronalen Netze eingegangen. Abschließend folgt eine kurze Übersicht über die Grundlagen des Spiels "Tetris", um die Herausforderungen für den Agenten zu veranschaulichen, und die vereinfachte Implementierung, welche zum Trainieren des Agenten verwendet wird.

A. Tiefe Neuronale Netzwerke

Tiefe Neuronale Netzwerke haben sich im Laufe den vergangenen Jahre zum State-of-the-Art im Bereich maschinelles Lernen entwickelt. In vielen Bereichen der Klassifizierung, unter anderem Sprach-, Schrift-, Objekt- und Bilderkennung, sind Neuronale Netze in der Lage, komplexe Strukturen und hochdimensionale Daten zu erkennen und entsprechend Zuordnungen zu treffen [4]. Bei einfachen Klassifizierungsaufgaben wie der Handschrifterkennung aus der MNIST Datenbank, können tiefe Neuronale Netze Zuordnungen sogar mit einer menschenähnlichen Genauigkeit treffen [5]. Ein Neuronales Netz besteht aus mehreren Schichten von verbundenen Neuronen. Von der ersten Schicht, der Eingabe-Schicht, werden Signale der Neuronen durch die folgenden versteckten Schichten an die Ausgabe-Schicht weitergegeben. Beim Anlegen von Werten an den Neuronen der Eingabe-Schicht ergeben sich nach der sogenannten Aktivierung der Neuronen aller Schichten die Werte der Neuronen der Ausgabe-Schicht. In der Regel ist jedes Neuron einer Schicht mit allen Neuronen

der vorherigen Schicht verbunden. In diesem Fall spricht man von einem vollvernetzten Netzwerk. Die Aktivierung eines Neurons ergibt sich aus der Aktivierung der Neuronen in der vorherigen Schicht. Hierzu werden die Aktivierungen der Neuronen der vorherigen Schicht mit sogenannten Gewichten multipliziert und addiert. Die Ausgabe einer Aktivierungsfunktion für diese Summe bestimmt die Aktivierung des Neurons. Die Charakteristik eines klassischen Neuronalen Netzes zeichnet sich primär durch die Anzahl der Schichten, die Anzahl der Neuronen pro Schicht und die verwendeten Aktivierungsfunktionen der Neuronen aus. Das konkrete Verhalten des jeweiligen Netzes hängt von dessen Gewichten ab. Je nach Änderung der Gewichte verändern sich bei der Eingabe der selben Werte an der Eingabe-Schicht die Aktivierung der Neuronen in der Ausgabe-Schicht und damit die letztendliche Ausgabe des Netzes. Die am weitesten verbreitete Methode zum Trainieren Neuronaler Netze ist überwachtes Lernen [4]. Dabei wird ein Neuronales Netz mit Eingaben und der jeweiligen erwarteten Ausgabe versorgt. Um nach der Eingabe das Netz zu trainieren, wird eine sogenannte Backpropagation durchgeführt. Hierbei vergleicht das Netz seine Ausgabe mit der erwarteten Ausgabe und erhält so die Abweichung von der erwarteten Ausgabe, den ”Fehler”. Dieser Fehler wird durch eine hochdimensionale Funktion beschrieben, die alle Gewichte des Netzes als Parameter enthält. Mit der Ableitung dieser Funktion wird ein Gradientenabstieg durchgeführt und dabei die Gewichte des Netzes angepasst, sodass die Fehlerfunktion minimiert wird. Wenn ein Neuronales Netz ”lernt”, ändert es also seine Gewichte, um seine Ausgabe an die erwartete Ausgabe anzunähern [6].

Zur Objekterkennung im Bereich der Bildverarbeitung haben sich CNN zum State-of-the-Art entwickelt [5]. Diese zeichnen sich im Vergleich zu einem klassischen Neuronalen Netz durch spezielle Schichten aus. Dabei handelt es sich um Convolutional- und Pooling-Schichten. In einer Convolutional-Schicht werden Filter, sogenannte Kernel, über das Bild geschoben. Pro Filter entsteht ein neues, gefiltertes Bild, das in die nächste Schicht weitergegeben wird. In einer weiteren Convolutional-Schicht wird dann wiederum pro Filter aus allen Eingabe-Bildern ein neues Bild erzeugt. Das Eingangsbild des Netzes wird also vervielfacht und durch die unterschiedlichen Filter verändert. Oft steht direkt nach einer Convolutional-Schicht eine Pooling-Schicht. Diese Schicht schiebt Kernel über das Bild, um die Auflösung des Bildes zu verringern. Dazu können verschiedene Methoden für die Reduzierung der Auflösung angewandt werden. Am weitesten verbreitet ist das Max-Pooling. Dabei wird der Pixel mit dem höchsten Wert im aktuellen Bereich des Kernels in das neue Bild übernommen. Um nach den Convolutional- und Pooling-Layern eine Klassifizierung des Bildes zu ermöglichen, werden die entstandenen Bilder in ein vollvernetztes Neuronales Netz überführt.

B. Reinforcement Learning

Ein weiterer Grundbaustein des DRL ist das bereits im Namen enthaltene RL. Bei RL soll ein Agent durch das Interagieren mit seiner Umgebung lernen, in dieser optimal zu handeln [3]. Der Agent wird trainiert, indem er nach dem Ausführen einer Aktion eine Belohnung erhält, das heißt eine Bewertung des durch die gerade ausgeführte Aktion entstandenen Zustands der Umgebung. Der Agent selbst weiß dabei nichts über das konkrete Ziel, welches er verfolgen soll. Das optimale Handeln besteht für den Agenten in der Maximierung der erhaltenen Belohnungen. Dem Agenten wird also nicht wie beim überwachten Lernen für einen Zustand der Umgebung eine korrekte Aktion vorgegeben, sondern er muss durch kontinuierliches Trial-and-Error lernen [3] und so selbst herausfinden, durch welche Aktionen in welchen Zuständen der Umgebung er die größte Belohnung erhält [7]. Der Agent startet also in einem Zustand der Umgebung und hat eine Auswahl an verfügbaren Aktionen vorgegeben. Der Agent wählt anhand seiner Erfahrungen eine Aktion und führt diese aus. Anschließend erhält er den neuen Zustand der Umgebung und eine Belohnung für die ausgeführte Aktion (siehe Abbildung 1). Dies wiederholt sich solange, bis eine Abbruchbedingung erfüllt ist. Das kann zum Beispiel ein terminaler Zustand der Umgebung oder eine feste Anzahl an Iterationen sein.

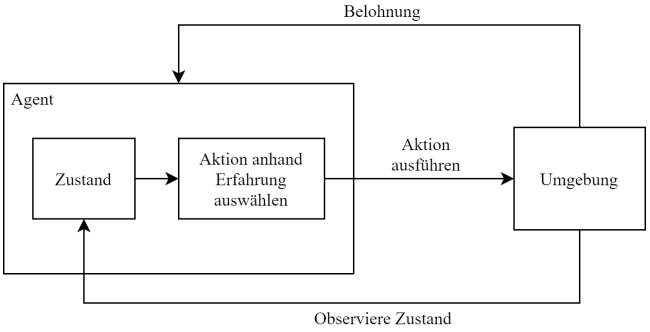

Abbildung 1. Architektur von Reinforcement Learning

C. Spiel und Implementierung

Tetris ist ein 2D Puzzle-Spiel aus der Sowjet-Union [8]. Im Spiel geht es darum, die auf ein rechteckiges Spielfeld herabfallenden Steine so anzuordnen, dass diese sich nicht bis zur Oberkante des Spielfelds stapeln. Die Spielsteine bestehen immer aus vier Quadraten, welche in den 7 möglichen Kombinationen zusammengesetzt sind [8]. Überschreitet ein Spielstein nach dem Landen auf einem anderen Stein die Obergrenze des Spielfelds, ist das Spiel vorbei und der Spieler hat verloren. Der Spieler kann das zu hohe Stapeln der Spielsteine verhindern, indem er eine Reihe, also das Spielfeld vom linken bis zum rechten Rand, lückenlos mit Steinen befüllt. In diesem Fall verschwindet diese Reihe und alle darüberliegenden Steine rutschen nach unten. Um solche Reihen zu puzzeln, kann der Spieler den gerade herabfallenden Stein sowohl drehen, als auch nach links oder rechts verschieben. Das Spiel bietet noch weitere Features, wie zum Beispiel ein Punktesystem oder die Möglichkeit, einen Stein zur Seite zu legen. Diese sind jedoch in unserer Implemen-

tierung des Spiels nicht vorhanden und daher für das Paper irrelevant.

Um das zu lösende Problem für den Agenten zu vereinfachen und das Training effizienter zu gestalten, wurde eine eigene Version von Tetris implementiert. Dazu wurde das Framework PyTorch in Python verwendet. In dieser Version existieren lediglich zwei verschiedene Spielsteine, ein 1x1 und ein 3x1 Spielstein. Die reduzierte Anzahl an Formen vereinfacht das Problem für den Agenten erheblich. Zudem wird der Anzahl der möglichen Aktionen des Agenten verringert, da für die Drehung der Spielsteine aufgrund deren Symmetrie nicht zwischen Links- und Rechtsdrehungen unterschieden werden muss. Das Spielfeld mit den entsprechenden Spielsteinen wurde als Tensor Matrix mit 20 Reihen und 8 Spalten umgesetzt. Als Observation der Umgebung erhält der Agent direkt diese Matrix. Dies hat zum Vorteil, dass alle nötigen Informationen über den Zustand des Spiels in 160 Feldern enthalten sind. Das ermöglicht die Größe des Inputs für das später folgende Neuronale Netz gering zu halten und damit Rechenleistung zu sparen.

III. Methoden und Theorie von Deep Reinforcement Learning

RL hat bemerkenswerte Erfolge in Ingenieur-, Sozial-, Naturwissenschaften und anderen Bereichen erzielt, jedoch ist das traditionelle RL ein sehr einfaches Modell. Realistische Probleme sind meist deutlich komplexer, welche sich durch die großen Dimensionen des Zustandsraums und des Aktionsraums auszeichnen. Der Einsatz von DRL ist eine Lösung für diese Probleme. Der Algorithmus ist in Bereichen wie Computer Vision, Spracherkennung und Verarbeitung von natürlicher Sprache bereits weit verbreitet. Eine Auswahl von Verwendungen werden im Kapitel *Verwandte Arbeiten* genauer beschrieben. Es vereint die starke Wahrnehmungs- und Repräsentationsfähigkeit von DL und die Entscheidungsfähigkeit von RL. [1]

Im folgenden Abschnitt werden der Algorithmus und die Grundlagen von DRL erklärt, der Ablauf des Trainings behandelt und abschließend die Architektur vorgestellt.

A. DRL Grundlagen und Algorithmus

Es werden Aufgaben betrachtet, bei denen ein Agent mit einer Umgebung interagiert. Dabei findet eine Abfolge von Aktionen, Beobachtungen und Belohnungen statt. In jedem Schritt wählt der Agent eine Aktion a_t von einer Auswahl an möglichen Aktionen, wobei t ein Zeitschritt ist. Das Spiel erhält diese Aktion und führt sie aus. Dies verändert den internen Zustand und den Spielstand. Dabei wird der interne Zustand des Spiels nicht vom Agenten beobachtet. Stattdessen nimmt der Agent ein Bild x_t vom Spiel wahr, eine Beobachtung, welche aus einem Vektor von Pixelwerten besteht und den derzeitigen Spielzustand repräsentiert. Der Agent erhält ebenfalls eine Belohnung r_t. Der Spielzustand hängt dabei von mehreren vorherigen Aktionen und Beobachtungen ab und das Feedback einer Aktion kann nur erhalten werden, nachdem zahlreiche Zeitschritte ausgeführt wurden. Daher sind Abfolgen von Aktionen und Beobachtungen $s_t = x_1, a_1, x_2, \ldots, a_{t-1}, x_t$ Input des Algorithmus, welcher aus diesen Abfolgen eine Strategie lernt.

Jede Abfolge von Aktionen terminiert nach einer Anzahl von Zeitschritten. Dadurch erhält man einen großen, aber endlichen Markow Entscheidungs Prozess (MDP). Für diese Entscheidungsprozesse können dann reguläre RL Methoden eingesetzt werden, wenn s_t als Zustand zum Zeitschritt t gewählt wird.

Der Agent möchte Aktionen wählen, welche die zukünftige Belohnung maximieren. Man definiert die optimale Aktionswert-Funktion (AF) $Q^*(s, a)$ als die maximal erhältliche Belohnung bei der einer Regel gefolgt wird, nachdem ein Zustand s wahrgenommen und eine Aktion a ausgeführt wurde. Diese AF folgt dem Optimalitätsprinzip von Bellman. Wenn der optimale Wert $Q^*(s', a')$ bei der Abfolge s' beim nächsten Zeitschritt für jede mögliche Aktion bekannt ist, dann ist die beste Strategie die nächste Aktion a' zu wählen, welche den Erwartungswert $r + \gamma * Q^*(s', a')$ maximiert, wobei die zukünftige Belohnung um den Faktor γ reduziert wird . Diesen nennt man auch den Discountfaktor. Dadurch erhält man folgende Gleichung:

$$Q^*(s, a) = r + \gamma \max_{a'} Q^*(s', a')$$

Damit kann, iterativ aktualisiert, die AF geschätzt werden, um schließlich zur optimalen AF zu konvergieren. Man verwendet dabei eine nicht lineare Funktion, um die AF zu approximieren. Diese nicht lineare Funktion ist ein Neuronales Netzwerk. Man nennt ein Netzwerk mit den Gewichten θ ein Q-Netzwerk. Die Gewichte θ_i in der Iteration i können trainiert werden, um den quadratischen Fehler in der Bellman Gleichung zu reduzieren. Es werden die Target Werte mit $y = r + \gamma \max_{a'} Q(s', a'; \theta_i^-)$ geschätzt, welche die Gewichte θ_i^- von einer vorherigen Iteration verwenden. Die Target Werte sind dabei nicht statisch festgelegt, wie es beim überwachten Lernen der Fall ist, sondern hängen dabei von den Gewichten des Q-Netzwerkes ab. Daraus resultiert eine Loss Funktion $L_i(\theta_i)$, welche sich bei jeder Iteration i verändert:

$$L_i(\theta_i) = (y - Q(s, a; \theta_i))^2$$

Durch Ableitung der Loss Funktion nach den Gewichten erhält man folgenden Gradienten:

$$\nabla_{\theta_i} L_i(\theta_i) = r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)$$

Mithilfe stochastischen Gradienten Abstiegs kann nun die Loss Funktion optimiert werden. Dies erfolgt durch die Aktualisierung der Gewichte nach jedem Zeitschritt und die Ersetzung $\theta_i^- = \theta_{i-1}$. [2]

B. DRL Training

Der im vorherigen Unterkapitel vorgestellte Algorithmus wird beim Training in zwei Arten verändert, damit das Lernen mit großen Neuronalen Netzwerken ohne Divergierungen stattfinden kann.

Eine wesentliche Verbesserung zur Stabilisierung des Algorithmus und zum schnelleren Lernen ist die Verwendung von Experience Replay. Im Experience Replay werden die Zustände, die Aktionen, die Belohnungen und die resultierenden Zustände des Agenten in jedem Zeitschritt gespeichert. Man erhält also ein Tupel $replay_t = (s_t, a_t, r_t, s_{t+1})$, welcher in einer Datenstruktur $M_t = \{replay_1, \ldots, replay_t\}$ für spätere Verwendung abgelegt wird. Während dem Training wird alle D Zeitschritte, eine minibatch Aktualisierung durchgeführt, welche aus einer zufälligen Auswahl von Erfahrungen aus dem Experience Replay M bestehen. Diese Art von Aktualisierungen hat mehrere Vorteile. Jede Erfahrung wird potenziell in vielen Gewichtsaktualisierungen verwendet, sodass eine größere Dateneffizienz ermöglicht wird. Unter anderem ist das Lernen aus aufeinanderfolgenden Erfahrungen aufgrund der starken Korrelationen zwischen den Erfahrungen sehr ineffizient. Indem die Erfahrungen zufällig ausgewählt werden, wird die Korrelation vermieden und reduziert [2]. Darüber hinaus weist eine minibatch Aktualisierung im Vergleich zu einer einzelnen Tupel Aktualisierung weniger Varianz auf. Es finden größere Aktualisierungen statt und ermöglichen zudem eine effiziente Parallelisierung des Algorithmus [3]. Durch die Verwendung von Experience Replay wird ebenfalls die Verteilung des Verhaltens des Agenten über viele der vorherigen Zustände gemittelt, wodurch das Lernen geglättet und Schwankungen oder Divergenzen in den Gewichten verhindert werden können.

Eine weitere Verbesserung ist die Verwendung eines zweiten Neuronalen Netzes, um die Target Werte y zu berechnen. Jede C Aktualisierungen werden die Gewichte des Netzwerkes Q in das sogenannte Zielnetzwerk $\hat{Q}$ kopiert. Anschließend wird $\hat{Q}$ für die nächsten C Aktualisierungen verwendet, um die Target Werte zu berechnen. Das zusätzliche Netzwerk stabilisiert dadurch das Lernen. Ohne dieses Zielnetzwerk würde eine Aktualisierung, die $Q(s_t, a_t)$ erhöht, ebenfalls $Q(s_{t+1}, a)$ für alle a und dadurch den Fehler y erhöhen. Dies kann zu Schwankungen und Divergierung führen. Die Berechnung des Target Wertes mithilfe von älteren Gewichten führt eine Verzögerung ein, die zwischen der Aktualisierung von Q und dem Zeitpunkt, ab dem diese Aktualisierung y beeinflusst, stattfindet. [2]

Der angepasste vollständige Algorithmus lautet wie folgt:

Deep Reinforcement Learning Algorithmus [2]

Initialisiere Vorhersagenetzwerk Q mit zufälligen Gewichten θ
Initialisiere Zielnetzwerk $\hat{Q}$ mit Gewichten $\theta^- = \theta$
Initialisiere Experience Replay M mit einer Größe N
$T =$ gesamte Anzahl von Iterationen
for t = 1 bis T **do**
  Wähle zufällige Aktion a_t mit Wahrscheinlichkeit ϵ
  Andernfalls, wähle $a_t = \underset{a}{argmax}\ Q(s_t, a; \theta)$
  Führe a_t aus und erhalte r_t und s_{t+1}
  Speichere das Tupel (s_t, a_t, r_t, s_{t+1}) in M
  if D Schritte erreicht **then**
    Generiere zufälligen minibatch aus den Tupeln (s_j, a_j, r_j, s_{j+1}) von M
    Berechne y_j aus

$$y_j = \begin{cases} r_j & \text{wenn Endzustand} \\ r_j + \gamma \underset{a'}{max}\ \hat{Q}(s_{j+1}, a'; \theta^-) & \text{andernfalls} \end{cases}$$

    Berechne Gradientenabstieg mit $(y_j - Q(s_j, a_j; \theta))^2$ und passe die Gewichten θ an
  end if
  Alle C Zeitschritte $\hat{Q} = Q$
end for

In Spielen, in denen der Agent einen Endzustand erreicht, also ein Spiel verloren hat, wird der Discountfaktor auf 0 gesetzt. Daher wird der Fehler y_j nur auf die Belohnung r_j gesetzt. Damit vermeidet der Agent diese Endzustände. Unter anderem wird eine spezielle Variante von stochastischem Gradientenabstieg verwendet. Diese Variante nennt sich RMSprop [9]. Zusätzlich werden in DRL ebenfalls Strategien von DL übernommen. Ganz besonders wird wie im typischen DL der Input normalisiert und, in diesem Fall, zwischen den Werten $[-1, 1]$ skaliert. [3]

C. Architektur von Deep Q-Netzwerken

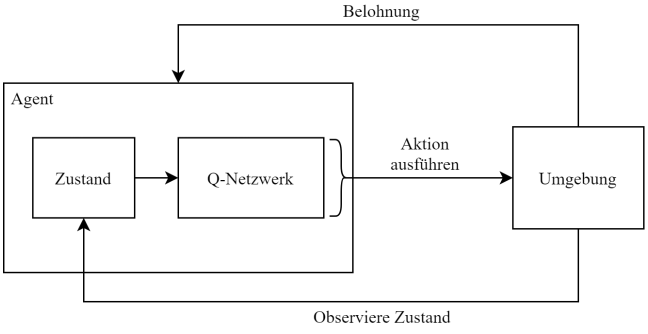

Abbildung 2. Architektur eines Deep Q-Netzwerkes

Ein Deep Q-Netzwerk (DQN) ist ein CNN, welches RL anstelle eines überwachten Trainings verwendet. Wie bereits im vorherigen Unterkapitel erklärt, werden die Q-Werte mit der Ausgabe des Q-Netzwerkes geschätzt, anstatt eine Q-Tabelle zu verwenden. Q-Werte für Zustand-Aktions-Paare können unter Verwendung von Q-Netzwerken gespeichert werden. Um die Q-Werte für unterschiedliche Aktionen in einem Zustand zu finden, wird einfach eine Vorwärtsaktivierung mit dem Zustand als Input durchgeführt. Die Ausgabe enthält anschließend für jede mögliche Aktion einen Q-Wert. Der größte Q-Wert hat die größte Wahrscheinlichkeit den Gewinn zu maximieren. Daher wird die Aktion mit dem größten Q-Wert gewählt. [10]

Der Vorgang und die allgemeine Architektur eines DQN wird in Abbildung 2 dargestellt. Der Agent nimmt den Zustand aus der Umgebung wahr, was zum Beispiel ein Videospiel sein kann. Der Zustand ist dann ein Bild aus dem Spiel, dass als Input zum Q-Netzwerk übergeben wird. Die Aktion

mit dem größten Q-Wert wird anschließend in der Umgebung ausgeführt und eine Belohnung wird generiert. Diese Belohnung erhält der Agent, um seine Wahl der Aktionen und die Berechnung der Q-Werte abhängig vom Zustand zu verbessern.

IV. Verwandte Arbeiten

DRL hat, gerade in den vergangenen Jahren vermehrt, Erfolge in verschiedenen Bereichen erzielen können. Dazu zählen Routenplanungen für Touristen [11], Mention-Ranking im Bereich der Sprachverarbeitung [12], das Wiedererkennen von Personen im Bereich der Bilderkennung [13] oder die Bewertung von Wertpapieren, um Handelstrategien zu entwickeln [14]. Wie auch in diesem Paper wird das Problem als endlicher MDP formuliert, um RL anzuwenden. In [11] und [13] wurden Convolutional Neural Networks als DQN eingesetzt.

DRL hat auch zum Entwickeln künstlicher Intelligenzen, welche das Spielen von Videospielen erlernen, verbreitet Anwendung gefunden. Vor allem für klassische Atari Arcade Games konnten Agenten entwickelt werden, welche die menschliche Leistung nachahmen oder sogar übertreffen können [15] [16] [17] [2]. Speziell [2] hat als Grundlage für dieses Paper gedient. Neben des Trainierens eines Agenten wurde DRL auch für künstliche Intelligenz von Nicht-Spieler-Charakteren eingesetzt, beispielsweise für intelligente Navigation dieser [18].

Auch auf Tetris im speziellen wurde bereits maschinelles Lernen angewandt. Von Interesse für dieses Paper sind zunächst Anwendungen von klassischem RL, jüngst [19]. Dabei haben die Hochdimensionalität der Kombinationen aus Zuständen und Aktionen und die dadurch entstehenden Anforderungen an Speicherkapazität jedoch dazu geführt, dass die Observierung der Umgebung eingeschränkt werden musste. Trotz dieser Einschränkung konnten Erfolge erzielt werden [19].

Bereits 2016 wurde mithilfe von DRL ein Agent trainiert, der Tetris spielen lernt [20]. Dort wurde als Belohnung eine Kombination aus der spielinternen Punktzahl und der in [21] vorgestellten Heuristik eingesetzt. Hierbei wurde als Problem formuliert, dass die Belohnungen zu viele Zeitschritte von den verantwortlichen Aktionen entfernt liegen und somit das Lernen negativ beeinflussen. Stattdessen wurden als Aktionen die möglichen Endpositionen des aktuellen Spielsteins verwendet. Dieses Problem der verzögerten Belohnungen konnte nun durch die im Kapitel *Belohnungsregeln* vorgestellten Belohnungsregeln gelöst werden, wodurch der Agent nach jedem Zeitschritt eine Belohnung erhält. Ein direkter Vergleich der Ansätze ist aufgrund der für dieses Paper vereinfachten Implementierung des Spiels leider nicht möglich.

V. DQN Implementierung zum Tetris Spielen

Das DQN wurde mithilfe des Frameworks PyTorch in Python implementiert. Die Implementierung des DQN definiert sich vorallem durch zwei Bereiche: Die Architektur des Q-Netzwerks und die Belohnungsregeln, welche den Kern des RL darstellen. Beide Bereiche werden im Folgenden erläutert, abschließend werden die verwendeten Hyperparameter aufgelistet.

A. Gewählte Architektur

Das Q-Netzwerk ist durch eine Kombination von Convolutional-Schichten, Pooling-Schichten und vollvernetzten Schichten aufgebaut und ist an die in [2] vorgestellte Architektur angelehnt. Die konkrete Architektur ist in Abbildung 3 dargestellt. Der Agent erhält das Bild des Spielfeldes als Zustand der Umgebung, welches als Eingabe des Q-Netzwerkes dient. In den Convolutional-Schichten werden 2D-Convolutionen mit Kernels der Größe 3x3 Pixel ausgeführt. Die eingesetzten Max-Pooling-Schichten verwenden Kernels mit einer Größe von 2x2 Pixel. Der vollvernetzte Teil des Netzwerkes bildet die Eingabe auf 4 Neuronen ab. Diese stellen die 4 möglichen Aktionen des Agenten dar: Keine Aktion ausführen, den Stein nach Links bewegen, den Stein nach Rechts bewegen oder den Stein drehen. Da der Zustand, also das Bild des Spielfeldes, vor der Eingabe nicht normalisiert wird, ist die Verwendung einer Aktivierungsfunktion notwendig, um optimales Lernen zu ermöglichen [3]. Als Aktivierungsfunktion wurde sowohl für die Convolutional-Schichten, als auch für die vollvernetzten Schichten die tanh-Funktion verwendet, wodurch die Eingaben auf Werte zwischen -1 und 1 abgebildet werden. Im Vergleich zur Sigmoid Aktivierungsfunktion, welche eine Ausgabe zwischen 0 und 1 besitzt, ist das Training bei Verwendung der tanh-Aktivierungsfunktion flexibler, da die Gewichte innerhalb eines Backpropagation Durchlaufs unabhängig voneinander geändert werden können [22] . Die sonst häufig eingesetzte Rectified Linear Aktivierungsfunktion hat beim Trainieren dieses Q-Netzwerkes zum explodieren der Gradienten und des Losses geführt.

B. Belohnungs Regeln

Die Verteilung von Belohnungen ist ein essentieller Bestandteil eines DQN. Abhängig davon, welches Verhalten vom Agenten erwartet wird, müssen entsprechend Belohnungen festgelegt werden. Dabei ist wichtig, wie groß oder klein die Belohnung ist, da der Agent aufgrund solcher Regeln Prioritäten setzen kann, die den weiteren Spielverlauf beeinflussen. Es wurden mehrere Experimente mit unterschiedlichen Regeln zur Verteilung von Belohnung durchgeführt. Die besten Ergebnisse liefert eine Kombination aus vier Regeln.

Zwei dieser Regeln finden nach dem Auftreten bestimmter Ereignisse statt. Der Agent erhält eine große Belohnung, wenn eine Reihe vervollständigt wird. Dieses Ereignis findet nur statt, nachdem ein Spielstein gelandet ist. Wenn ein Stein in einer zuvor festgelegten Reihe landet, die nahe am oberen Ende des Spielfeldes liegt, dann hat der Agent ein Endzustand erreicht. Daraufhin erhält der Agent einen große negative Belohnung.

Die anderen beiden zusätzlichen Regeln werden nach jeder Aktion des Agenten ausgeführt. Dadurch wird der derzeitige Zustand des Spielsteines bewertet. Zum einen wird dem Agenten eine kleine positive Belohnung übergeben, abhängig

davon, in welcher Reihe sich der gerade zu spielende Stein befindet. Je tiefer der Stein ist, desto größer ist die Belohnung. Diese Regeln motivieren den Agenten einen Stein möglichst tief zu legen, um z.B. einen Endzustand zu vermeiden. Zum anderen wird dem Agenten eine mittlere negative Belohnung übergeben, abhängig davon, wie viele Steine sich bereits in der Spalte befinden, in der sich der gerade zu spielende Stein befindet. Je größer die Anzahl der bereits in der Spalte abgelegten Spielsteine, desto größer die negative Belohnung. Dies motiviert ebenfalls den Agenten die Spielsteine möglichst tief zu legen und den Spielstein nicht auf bereits hohe Ablagen von Steinen zu legen.

C. Gewählte Hyperparameter

In Tabelle V-C werden die gewählten Hyperparameter dargestellt und kurz erläutert.

VI. Ergebnisse

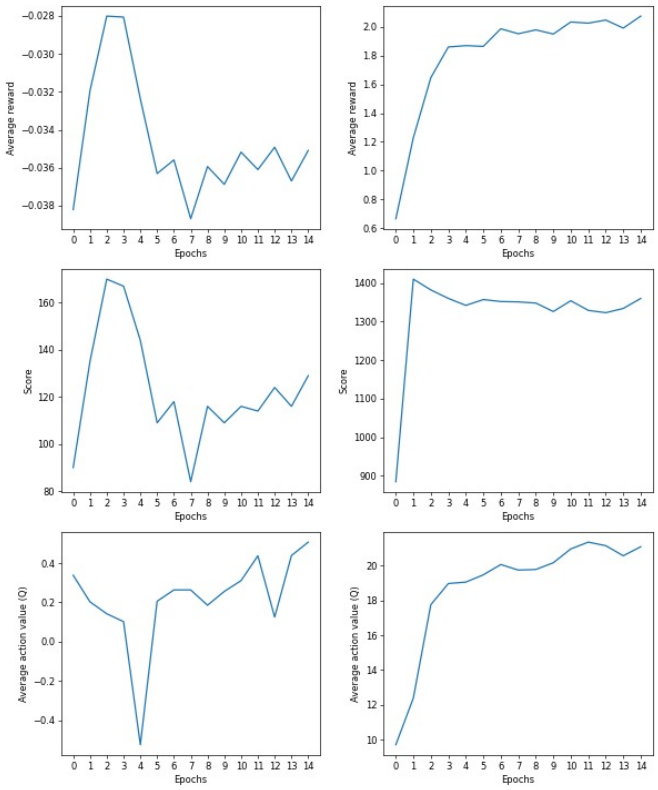

Abbildung 4. Linke Spalte die Ergebnisse vor der Implementierung der zusätzlichen Belohnungs Regeln. Rechte Spalte mit Implementierung.

Es wurden mehrere Durchläufe mit unterschiedlichen Netzwerkparametern und Parametern zum Sammeln von Informationen bezüglich der Performance ausgeführt. Um genaue Daten zu gewinnen, muss der Algorithmus sehr lange laufen. Dies war nur begrenzt möglich, da keine GPU zum Ausführen vorhanden war. Daher musste stattdessen Google Colaboratory [23] verwendet werden. Google Colaboratory erlaubt die Ausführung von Python Code im Browser und dabei kostenlosen Zugriff auf von Google bereitgestellte GPUs. Mithilfe Colab konnte der Algorithmus deutlich schneller und viel mehr Epochen mit mehreren Iterationen als geplant durchgeführt werden.

In Abbildung 4 werden die Ergebnisse der Testläufe dargestellt. Die Testläufe werden mit 15 Epochen jeweils 100000 Iterationen durchgeführt. In der linken Spalte der Abbildung sind die Ergebnisse vor der Implementierung derjenigen Belohnungs Regeln, die zusätzlich bei jeder Aktion ausgeführt werden (siehe V-B) und rechts mit der Implementierung. Für jeden Durchlauf wird die durchschnittliche Belohnung, der Spielstand und der durchschnittliche Aktionswert aller möglichen Entscheidungen für jede Epoche visualisiert. Je größer die Werte der Abbildungen, desto besser verhält sich der Agent und desto mehr hat der Agent gelernt.

Die Durchführung verläuft deutlich besser mithilfe der zusätzlich implementierten Regeln. Damit erhält der Agent eine Evaluation seines Zustandes nach jeder Aktion. Ohne diese zusätzlichen Regeln kann der Agent nicht bewerten, ob eine Aktion gut oder schlecht ist, bevor der Agent verliert oder eine Reihe vervollständigt.

Die Funktionsgraphen zeigen, dass das Training einen stetig steigenden Durchschnitt von Belohnung, Spielstand und Aktionswert bis zum Ende der Epochen erzielt. Das deutet darauf hin, dass der Agent weiterhin lernt und eine größere Anzahl von Epochen ein besseres Ergebnis erzeugen könnte. Doch konnten mehr Trainingsepochen durch Beschränkungen in der Verwendungsdauer von Google Colab nicht durchgeführt werden.

VII. Ausblick

Um die Komplexität des Problems für den Agenten zu reduzieren und somit zunächst mit geringem Training in kurzer Zeit Erfolge erzielen zu können, wurde das Spiel Tetris stark vereinfacht. In einer folgenden Arbeit könnte auf der bestehenden Implementierung aufgebaut und verschiedene Aspekte des Spiels aufgenommen werden. Zunächst würde das bedeuten, alle sieben verschiedenen Spielsteine des Spiels umzusetzen. Dementsprechend müsste der Agent auch die Möglichkeit besitzen, zwischen Links- und nach Rechtsdrehungen unterscheiden zu können. Dies war aufgrund der Symmetrie der bisher verwendeten Formen der Spielsteine noch nicht notwendig. Neben den verschiedenen Formen der Spielsteine bietet Tetris zudem weitere Funktionen. So kann der Spieler, bzw. der Agent, einen Spielstein zur Seite legen, um diesen später mit einem aktuellen Stein austauschen und diesen platzieren zu können. Des Weiteren gibt es die Möglichkeit, die Reihenfolge der nächsten drei zu spielenden Steine einzusehen, wodurch eine gewisse Planung ermöglicht und vorausschauendes Spielen belohnt wird. Neben dem Umgang mit der erhöhten Komplexität des Spiels wäre das Verhalten des Agenten gerade mit dieser letzten Funktion spannend zu verfolgen. Je nach dem wie gut der bereits implementierte DRL Agent mit dieser neuer Umgebung und möglichen Aktionen zurechtkommt, wären natürlich weitere Änderungen an der Implementierung zu erörtern, um die komplexeren Formen des Problems zu lösen. Neben Veränderungen an der Architektur des DQN könnten

das auch verschiedene bereits in [20] verwendete Techniken wie Dropout oder Prioritized Sweeping sein, welches Tupel aus dem Experiance Replay mit einem hohen Fehler priorisiert [20]. Abschließend wird vor allem ein ausgiebigeres Trainieren angestrebt, da die sich stetig verbessernden Ergebnisse noch vorhandenes Potential des Agenten andeuten, welches noch nicht ausgeschöpft scheint.

VIII. Zusammenfassung

In diesem Paper wurde die Implementierung eines DRL Agenten zum Spielen des 2D-Puzzle Spiels Tetris vorgestellt. Dabei wurde mithilfe des Frameworks PyTorch in Python programmiert und auch eine eigene Implementierung des Spiels umgesetzt, wobei Vereinfachungen am Spiel zur Reduzierung der Komplexität des Problems vorgenommen wurden. Das verwendete DQN besteht aus Convolutional-Schichten, Max-Pooling-Schichten und vollvernetzten Schichten, wobei nach ersteren und letzteren tanh-Aktivierungsfunktionen angewandt werden. Um eine zu lange Verzögerung der Belohnung nach den verantwortlichen Aktionen zu vermeiden, wird eine Heuristik aus vier Belohnungsregeln verwendet, wovon zwei der Regeln nach jedem Zeitschritt eine Belohnung an den Agenten zurückgeben. Nach der Einführung dieser beiden Regeln war eine deutliche Verbesserung erkennbar. Aufgrund der fehlenden Verfügbarkeit einer privaten GPU wurde das Trainieren des Agenten über Google Colab ausgeführt. Das Training zeigt Erfolge und einen stetig steigenden Durchschnitt an Belohnung, Punkte und Aktionswert pro Epoche. Doch aufgrund von Google Colab konnte das Training nicht im gewünschten Ausmaß stattfinden.

Abkürzungen

AF Aktionswert-Funktion
DL Deep Learning
DRL Deep Reinforcement Learning
DQN Deep Q-Netzwerk
RL Reinforcement Learning
MDP Markow Entscheidungs Prozess
CNN Convolutional Neural Network

Literatur

[1] L. Cao and ZhiMin, "An overview of deep reinforcement learning," in *Proceedings of the 2019 4th International Conference on Automation, Control and Robotics Engineering*, ser. CACRE2019. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3351917.3351989

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–33, 02 2015.

[3] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An introduction to deep reinforcement learning," *CoRR*, vol. abs/1811.12560, 2018. [Online]. Available: http://arxiv.org/abs/1811.12560

[4] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–44, 05 2015.

[5] D. Ciregan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012, pp. 3642–3649.

[6] M. Nielsen, *Neural networks and deep learning*. Determination Press, 2015.

[7] R. Sutton and A. Barto, *Reinforcement Learning, second edition: An Introduction*, ser. Adaptive Computation and Machine Learning series. MIT Press, 2018. [Online]. Available: https://books.google.de/books?id=uWV0DwAAQBAJ

[8] Nintendo, "Tetris instruction booklet," https://www.gamesdatabase.org/Media/SYSTEM/Nintendo_Game_Boy/manual/Formated/Tetris_-_1989_-_Nintendo.pdf, 1989, zugriff am 01.07.2021.

[9] H. Tieleman, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural Networks for Machine Learning*, 2012.

[10] M.-J. Li, A.-H. Li, Y.-J. Huang, and S.-I. Chu, "Implementation of deep reinforcement learning," in *Proceedings of the 2019 2nd International Conference on Information Science and Systems*, ser. ICISS 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 232–236. [Online]. Available: https://doi.org/10.1145/3322645.3322693

[11] S. Chen, B.-H. Chen, Z. Chen, and Y. Wu, "Itinerary planning via deep reinforcement learning," in *Proceedings of the 2020 International Conference on Multimedia Retrieval*, ser. ICMR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 286–290. [Online]. Available: https://doi.org/10.1145/3372278.3390727

[12] K. Clark and C. D. Manning, "Deep reinforcement learning for mention-ranking coreference models," *CoRR*, vol. abs/1609.08667, 2016. [Online]. Available: http://arxiv.org/abs/1609.08667

[13] X. Lan, H. Wang, S. Gong, and X. Zhu, "Deep reinforcement learning attention selection for person re-identification," 01 2017.

[14] Z. Zhang, S. Zohren, and S. Roberts, "Deep reinforcement learning for trading," *The Journal of Financial Data Science*, vol. 2, pp. 25–40, 04 2020.

[15] K. Shao, Z. Tang, Y. Zhu, N. Li, and D. Zhao, "A survey of deep reinforcement learning in video games," 2019.

[16] I.-A. Hosu and T. Rebedea, "Playing atari games with deep reinforcement learning and human checkpoint replay," 2016.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[18] E. Alonso, M. Peter, D. Goumard, and J. Romoff, "Deep reinforcement learning for navigation in aaa video games," 2020.

[19] A. Groß, J. Friedland, and F. Schwenker, "Learning to play tetris applying reinforcement learning methods," 01 2008, pp. 131–136.

[20] M. Stevens, "Playing tetris with deep reinforcement learning," 2016.

[21] Y. Lee, "Tetris ai - the (near) perfect bot," https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/, 2015, zugriff am 11.07.2021.

[22] T. Stöttner, "Why data should be normalized before training a neural network," https://towardsdatascience.com/why-data-should-be-normalized-before-training-a-neural-network-c626b7f66c7d, 2019, zugriff am 03.07.2021.

[23] Google, "What is colaboratory?" https://colab.research.google.com/notebooks/intro.ipynb?utm_source=scs-index, zugriff am 10.07.2021.

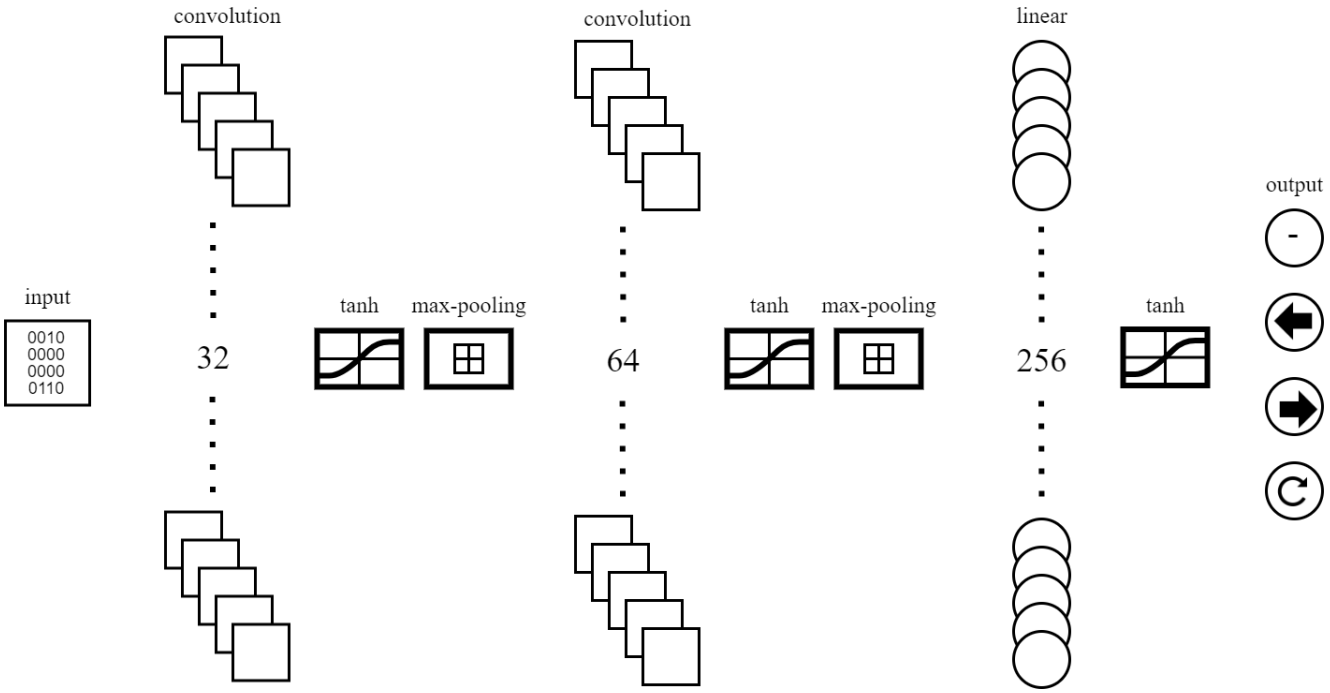

Abbildung 3. Architektur des Q-Netzwerkes

Tabelle I
HYPERPARAMETER

Hyperparameter	**Wert**	**Beschreibung**
`game_rows`	20	Höhe/Anzahl der Reihen des Spielfeldes
`game_colums`	8	Breite/Anzahl der Spalten des Spielfeldes
`experiance replay size`	100.000	Anzahl der vergangenen Züge, die gespeichert werden
`epsilon`	1	Startwert von Epsilon
`epsilon_end`	0.05	Ist dieser Wert erreicht, wird Epsilon nicht mehr verringert
`decay`	0.999	Um diesen Faktor wird Epsilon verringert
`discount`	0.9	Verwendeter Faktor γ zur Reduzierung der zukünftigen Belohnung
`batch-size`	256	Anzahl an Replays, die zum Trainieren aus dem Replay-Memory entnommen werden.
`update_step`	16	Anzahl an Iterationen, nach denen Backpropagation durchgeführt wird
`copy_weights_step`	160	Anzahl an Iterationen, nach denen die Gewichte des Q-Netzwerks zum Zielnetzwerk kopiert werden