Institut für Informatik
Lehrstuhl für Organic Computing
Prof. Dr. rer. nat. Jörg Hähner

Bachelorarbeit

Implementing Rainbow DQN in StarCraft II using Keras-rl

Benjamin Möckl

Erstprüfer:	Prof. Dr. rer. nat. Jörg Hähner
Zweitprüfer:	Prof. Dr. Elisabeth André
Betreuer:	Baron Wenzel Pilar von Pilchau
Matrikelnummer:	1380802
Studiengang:	Ingenieur-Informatik (Bachelor)
Eingereicht am:	11. März 2019

Abstract

Der Rainbow-DQN Algorithmus [HMVH+18], der aus dem Versuch, mehrere unabhängige Verbesserungen des ursprünglichen Deep-Q-Network (DQN) Algorithmus [MKS+13] zu vereinen, entstanden ist, hat 2018 im populären Atari 2600 Benchmark state-of-the-art Ergebnisse erzielen können. Für diese Arbeit war es Ziel, diesen Algorithmus mit Hilfe des Frameworks Keras [C+15], beziehungsweise Keras-rl [Pla16], an das StarCraft II Learning Environment [VEB+17] anzupassen, welches gegenüber den Atari Spielen einige neue Herausforderungen vorweist. Die Experimente verifizieren, dass die Kombination der verschiedenen Rainbow-Erweiterungen [Double DQN, Dueling DQN, Noisy Nets, Multi-Step Learning, Prioritized Experience Replay; ohne Distributional RL] mit der „FullyConv“Architektur [VEB+17], die DeepMind vorschlägt, einen leistungsstärkeren Algorithmus hervorbringt, der auch einen deutlich größeren Action-Space im Vergleich zu Atari bewältigt.

Inhaltsverzeichnis

1 Einleitung

Deep Reinforcement Learning, als Form der Anwendung von neuralen Netzwerken auf Kontrollprobleme, hat spätestens durch die Algorithmen AlphaZero und AlphaGo [SHS+17] große Aufmerksamkeit erlangt, welche in ihren jeweiligen Gebieten [Schach und Go] übermenschliche Leistungen erreichen konnten. Verglichen mit vielen anderen Problemstellungen haben Schach und Go allerdings einige Eigenschaften, die Reinforcement Learning zugute kommen. Insbesondere ist der State der Spiele jederzeit vollständig für den Agenten sichtbar (man sieht das ganze Spielfeld mit allen Figuren jederzeit) und die Gesamtzahl der Spielzüge bis zum Ende eines Spiels ist nur zwei-bis dreistellig - wodurch beinahe jeder Zug strategisch sehr relevant ist. Zusätzlich sind die Spiele einfach simulierbar, unter Aufwand von wenig Rechenleistung.

Der nächste logische Schritt der Forschung war neue Herausforderungen zu finden. Hervorzuheben ist dabei das Arcade Learning Environment [BNVB13], welches eine Schnittstelle für die Observierung und Kontrolle von über 70 Atari Spielen bereitstellt. Neue Herausforderungen sind hier beispielsweise kontinuierliche Kontrollprobleme sowie das Lernen von den rohen RGB Pixeln des (simulierten) Spielbildschirms. Nachdem 2015 in vielen der Atari Games die Performance menschlicher Spieletester mit dem Deep Q-Learning Algorithmus [MKS+15] erreicht oder übertroffen wurde, wurden bis heute viele weitere neue Rekorde aufgestellt sowie Erweiterungen und Verbesserungen zu diesem veröffentlicht. Einer der Spitzenreiter Ende 2017 war dabei der sogenannte Rainbow DQN Algorithmus [HMVH+18], welcher sechs voneinander unabhängige Verbesserungen des ursprünglichen DQN Algorithmus zusammenfasst (die in den folgenden Kapiteln ausführlich beschrieben werden). Ziel dieser Arbeit ist es

nun, die Erfolge dieses Algorithmus in einer neuen herausfordernden Umgebung zu reproduzieren und zu validieren.

1.1 StarCraft II

Dieser Abschnitt ist eine Übersetzung und Paraphrasierung des Abschnitts „1 Introduction“ des bereits mehrfach zitierten Papers [BNVB13]. Das StarCraft II Learning Environment stellt einen weiteren Schritt in Richtung noch komplexerer Herausforderungen dar. StarCraft II ist ein Echtzeit-Strategiespiel (RTS), das viele schnelle Mikro-Interaktionen mit der Notwendigkeit, weit vorauszuplanen, kombiniert. StarCraft I und II sind schon lange im e-Sport verwurzelt, haben mehrere millionen Spieler und viele hochkarätige Pro-Spieler in einer aktiven, kompetitiven Szene. Menschliche Top-Spieler zu besiegen sei deshalb ein bedeutungsvolles und messbares langfristiges Ziel, regt Deepmind an.

StarCraft II ist ein Multiagentenproblem. Zum einen in der Hinsicht, dass man gegen einen oder mehrere Gegner spielt, zum anderen aber, weil man hunderte Einheiten sinnvoll kontrollieren muss, um besagte Gegner zu besiegen. Außerdem ist die Information über den aktuellen Stand des Spiels für den Spieler bzw. den Agenten nicht vollständig einsehbar: Man sieht stets nur einen Teil des Spielfelds und muss dieses Sichtfenster aktiv verschieben, um sich umzuschauen. Außerdem liegt (ähnlich zu anderen RTS- oder MOBA-Spielen) ein sogenannter „Fog of War“ über dem Spielfeld, eine Mechanik, die bewirkt, dass man nur Gebiete sehen kann, in denen eigene Einheiten oder Gebäude stehen. Um also herauszufinden, was der Gegner plant, muss man ihn mit Einheiten geschickt ausspionieren und diese Erfahrungen in seine Strategie einfließen lassen sowie versuchen zu verhindern, dass der Gegner einen zu genau beobachtet. Dies speziell sind schon Konzepte, die klassischem Reinforcement Learning [RL] bereits größere Probleme bereiten.

Besonders im Vergleich zu „klassischen“ RL Problemdomänen wie Schach, aber auch im Vergleich zu dem bereits erwähnten Arcade LE [BNVB13], ist

außerdem ein gigantischer Action-Space zu beachten, also die Anzahl der möglichen Spielzüge zu jedem Zeitpunkt. Die Arcade-Spiele beispielsweise haben aufgrund der einfachen Peripherie der Konsole (einem Joystick für Richtungseingaben und einem einzelnen Button für Aktionen - andere verfügbare Eingabegeräte haben ähnlich wenig Möglichkeiten) nur eine Handvoll möglicher Aktionen. Schach hat pro Spielfigur mehrere Optionen, für den ersten Spielzug sind es summiert 20, und im Spielverlauf kann diese Zahl die Hundert übersteigen. StarCraft II hingegen benutzt als Eingabegeräte Tastatur und Maus, die meisten verfügbaren Aktionen können durch ein Tastaturkürzel und einen Mausklick (mit einigen Einschränkungen) auf jeden Pixel des Bildschirms als Ziel durchgeführt werden, was bei hunderten möglichen Aktionen und nach Schätzung von Deepmind in einer Größenordnung von 10^8 resultiert. Zudem verändert sich die Anzahl der möglichen Aktionen über die Zeit, wenn der Spieler den Technologiebaum des Spiels erforscht. Außerdem ist der Spieler bzw. Agent gezwungen, weit vorauszuplanen, da ihm im späteren Spielverlauf frühere Entscheidungen Probleme bereiten können. Ein Beispiel dafür sei das Ausbilden von einer genügend großen Armee, um, wenn der Gegner 10-20 Minuten später angreift und man dessen Armee das erste Mal sieht, dieser entgegentreten zu können.

StarCraft II ist also eine herausfordernde und sehr spannende Plattform, um den Rainbow-DQN Algorithmus zu testen.

1.2 Aufbau dieser Arbeit

Zu Beginn wird die verwendete Lernumgebung aus einer technischeren Sicht beschrieben, zusammen mit den verwendeten Frameworks. Anschließend werden alle Komponenten des Rainbow-DQN-Algorithmus von Q-Learing und Deep-Q-Learning bishin zum fertig integrierten Agent aus theoretischer und technischer Sicht detailliert erklärt. Im nächsten Kapitel werden alle Testläufe und Ergebnisse vorgestellt, inklusive der verwendeten Methoden. Anschließend folgt eine Interpretation, die sich besonders auf die Aussagen und Ergebnisse

des Rainbow-DQN Papers [HMVH+18] bezieht. Es folgen weiterführende Gedanken und mögliche Erweiterungen bzw. Verbesserungen des Algorithmus.

1.3 Verwandte Arbeiten

Grundlage dieser Arbeit ist sowohl der Rainbow-DQN Algorithmus und das zugehörige Paper [HMVH+18], sowie die darin verwendeten vorausgehenden Arbeiten der einzelnen Teilalgorithmen. Insbesondere hervorzuheben sind die Arbeiten [MKS+13] und [MKS+15], die einen Grundstein für die Verwendung des Q-Learning Algorithmus in Kombination mit neuralen Netzen unter dem Namen Deep Q-Networks [DQN] für Kontrollprobleme (hier Atari Spiele) legten. In vielen dieser Spiele erreichten sie bereits bessere Ergebnisse als menschliche Spieler.

Auf der anderen Seite stehen Arbeiten, die sich mit der gleichen Problemdomäne StarCraft II beschäftigen und damit wichtige Vergleichswerte liefern. Das in Kollaboration von DeepMind und Blizzard erstellte Paper [VEB+17], das mit der Veröffentlichung der Lernumgebung einherging, bietet neben einer StarCraft II Python API „pysc2“, ohne welche diese Arbeit nicht möglich wäre. Außerdem enthält es eine Netzwerkarchitektur namens „FullyConv“, auf deren Grundlage die oben genannten Algorithmen implementiert werden und welche allein die Leistungsfähigkeit des DQN Algorithmus bereits (in StarCraft II) verbessert. Mit Hilfe des A3C Algorithmus und dieser Architektur erstellte Vergleichswerte sind auch bereitgestellt.

In Bezug auf die Implementierung wird Keras-rl [Pla16] benutzt, das eine Sammlung von RL-Algorithmen mit Hilfe des Deep-Learning Frameworks Keras bereitstellt [C+15], insbesondere DQN, Double DQN und Dueling DQN. Für die Implementierung eines Prioritized Replay Memorys wurde eine von OpenAI veröffentlichte Implementierung [DHK+17] verwendet und angepasst.

2 StarCraft II Learning Environment

Das StarCraft II Learning Environment (SC2LE)[VEB+17] ist das Ergebnis der Kollaboration von DeepMind und Blizzard Entertainment, um eine offene Umgebung für Reinforcement Learning Forschung zu schaffen. Während Blizzard das Spiel um eine Schnittstelle (API) erweitert hat [Ent17], die vollständige Kontrolle und Informationsfluss zur Verfügung stellt, sowie Linux-Builds, die erlauben, das Spiel „headless“ (ohne Benutzeroberfläche) und somit effizienter (und auf Linux, da StarCraft II nur auf PC und Mac erschienen ist) ausführen zu können, hat DeepMind in Form der „pysc2“ Python Bibliothek [Dee17] dafür einen Wrapper gebaut, mit welchem man nun programmatischen Zugriff auf das Spiel hat. Des Weiteren wurden von Blizzard eine Reihe verschiedener Maps (Spielfelder), die aktuell und im Laufe der letzten Jahre viel gespielt werden beziehungsweise wurden, zusammen mit sieben Mini-Games veröffentlicht. Zusätzlich wurde eine Vielzahl von anonymisierten Replays menschlicher Spieler zum Download bereitgestellt.

2.1 Observation

Ein Agent sieht das Spiel durch die Schnittstellen der pysc2 API auf zwei mögliche Art und Weisen: Entweder in Form von den RGB Pixeln der normalen Spieloberfläche, die auch ein menschlicher Spieler sehen würde, mit einer Ausnahme: Bedienelemente und rein kosmetische Elemente des Spiels sind ausgeblendet. Oder über so genannte Feature-Layers, die in Form von mehreren

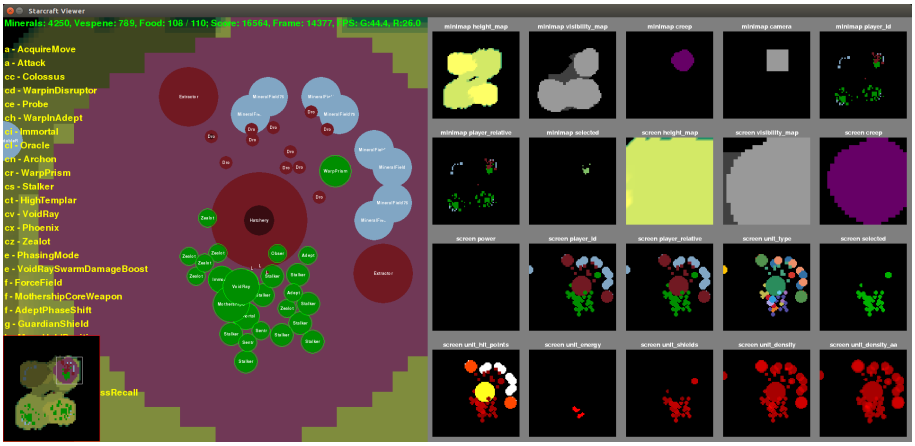

Abbildung 2.1: Darstellung der Feature-Layers, links eine für einen Menschen interpretierbare Version, rechts eingefärbte Versionen der Minimap- und Screen-Feature-Layers, aus [VEB+17]

Bildern erscheinen, auf welchen abstrahiert jeweils einzelne Aspekte des Spiels abgebildet sind, siehe Abbildung 2.1. Ergänzt werden diese beiden Möglichkeiten durch eine Reihe von nicht-positionsspezifischen Daten, auf die ein Spieler durch für die API ausgeblendete Elemente Zugriff hätte, sowie die Möglichkeit, auf weitere Daten zuzugreifen, wie Position und Status aller Einheiten auf dem Spielfeld.

Für die Experimente, die im Rahmen dieser Arbeit an mehreren Mini-Games gemacht wurden, wurden dem Agent nur zwei für die jeweiligen Tasks relevante Feature-Layers mit einer Auflösung von 32∗32 Pixeln gezeigt, um den Aufwand an Rechenleistung gering zu halten.

2.2 Interaktion

Die möglichen Aktionen, die durch die pysc2 API bereitgestellt werden, sind so entworfen, dass sie möglichst genau das normale Spielinterface imitieren. Eine Aktion besteht aus ihrer Identifikationsnummer und einer Liste von Argumenten, die auch leer sein kann. Jede Aktion akzeptiert verschiedene Argumente. Da viele Aktionen durch einen Mausklick an eine bestimmte Position auf dem

Bildschirm abgeschlossen werden (für Menschen), sind häufige Argumente ein diskretes Koordinatenpaar, um das Ziel der Aktion anzugeben. Da nicht in jedem Zeitschritt alle Aktionen verfügbar sind, stellt die API außerdem eine Liste der zum jeweiligen Zeitpunkt verfügbaren Optionen bereit.

Um Rechenleistung zu sparen wurden auch an dieser Stelle für die Experimente nur die Aktionen, die relevant sind um ein Mini-Game zu lösen, für den Agenten benutzt.

Theoretisch hätte ein Agent die Möglichkeit, nach jedem einzelnen Zeitschritt eine Aktion auszuführen. Dies wäre allerdings ein großer Vorteil gegenüber einem menschlichen Spieler. In StarCraft II wird die Metrik „actions-per-minute“ (APM) oft benutzt, die beispielsweise auch bei Wettkämpfen für die Zuschauer mit eingeblendet wird. Bei normalen Spielern kann die APM typischerweise Werte zwischen 30 und 500 annehmen, wobei Anfänger oft eine relativ niedrige APM um die 30 haben, 60-100 APM der Durchschnitt ist, und 200+ APM (mehr als drei Aktionen pro Sekunde) bei professionellen Spielern normal ist. Um dem Agent ähnliche Werte zu geben, darf er nur jeden 8. Frame eine Observation erhalten und eine Aktion durchführen nach Vorbild der Versuche von DeepMind in [VEB+17]. Dies resultiert in einer APM von ungefähr 180.

2.3 Mini-Games

Um einzelne Aspekte des Spiels getrennt zu untersuchen sowie sich inkrementell dem schwierigen Task, ein ganzes StarCraft II Match zu spielen, zu nähern, hat DeepMind eine Reihe von Mini-Games veröffentlicht. Im folgenden werden davon nur die erläutert, die ich für meine Experimente benutzt habe, aufgrund von Zeit und Rechenleistung konnte es nur eine kleine Auswahl sein.

2.3.1 MoveToBeacon

Der Spieler erhält einen einzelnen Marine und eine runde Zielmarkierung auf dem Spielfeld. Erreicht er mit diesem die Markierung, erhält der Spieler einen Reward von +1 und ein neues Ziel erscheint. Der Spieler soll in einer fixen Zeitspanne von 120 Sekunden möglichst viele Punkte sammeln. Dieses Mini-Game ist gedacht worden als Test für das Funktionieren aller Komponenten bzw. des verwendeten Lernalgorithmus, da es ein sehr einfaches Problem mit einer trivialen Strategie ist - die optimale Punktzahl erhält man einfach, wenn man den Laufbefehl für den Marine stets ins Zentrum der runden Markierung setzt. Um die Implementationen in dieser Arbeit zu testen, war dieses Mini-Game sehr hilfreich.

2.3.2 CollectMineralShards

Der Spieler erhält zwei Marines und kann diese selektieren oder deselektieren und bewegen. Auf dem Spielfeld befinden sich 20 Mineraliensplitter (Mineral Shards) an zufälligen Positionen (mit Mindestabstand zu den Marines). Bewegt sich ein Marine über einen dieser Splitter, sammelt er diesen ein und erhält dafür einen Reward von +1. Sind alle eingesammelt, erscheinen 20 neue zufällig verteilte Splitter. Ziel ist es, in einer fixen Zeitspanne möglichst viele Mineraliensplitter zu sammeln. Eine optimale Strategie erfordert, beide Marines unabhängig voneinander zu bewegen und gleichzeitig Kristalle aufzusammeln.

3 Die Algorithmen

Im folgenden Kapitel werden von den Grundlagen des Reinforcement Learning, über Deep Q-Networks und sechs Erweiterungen dessen, bishin zum fertigen, integrierten Rainbow-DQN Algorithmus detaillierte Erklärungen anhand von Pseudo-Code sowie einigen Details über die Implementierung bereitgestellt.

3.1 Reinforcement Learning

Reinforcement Learning (RL, zu Deutsch „Bestärkendes Lernen“) beschäftigt sich damit, in einer für den Algorithmus (Agent) unbekannten Umgebung (Environment) durch Ausprobieren von Aktionen (Actions) eine möglichst hohe Belohnung zu erreichen, die nach dem Ausführen einer Aktion von der Umgebung ausgeschüttet wird und auch negativ sein kann. Formal kann man diese Agent-Environment-Interaktion als teilweise observierbaren Markov Entscheidungsprozess (partally observable) (Markov decision process, MDP, siehe Kapitel 3.1.1) beschreiben. Teilweise observierbar beschreibt dabei die Annahme, dass der Agent nicht den gesamten Zustand des Environments sehen kann, sondern nur einen Teil - wie beispielsweise in StarCraft II man die Spielzüge seines Gegners erst sehen kann, wenn man ihn mit Einheiten ausspioniert. Es gibt allerdings RL-Probleme, für die diese Einschränkung wegfällt. Zu erwähnen seien hier beispielsweise Brettspiele wie Schach, wo man stets das ganze Spielfeld sehen kann. Diese Einschränkung bedeutet, dass der Agent nicht das Ergebnis seiner Aktionen vorher berechnen, sondern nur eine stochastische Aussage darüber treffen kann.

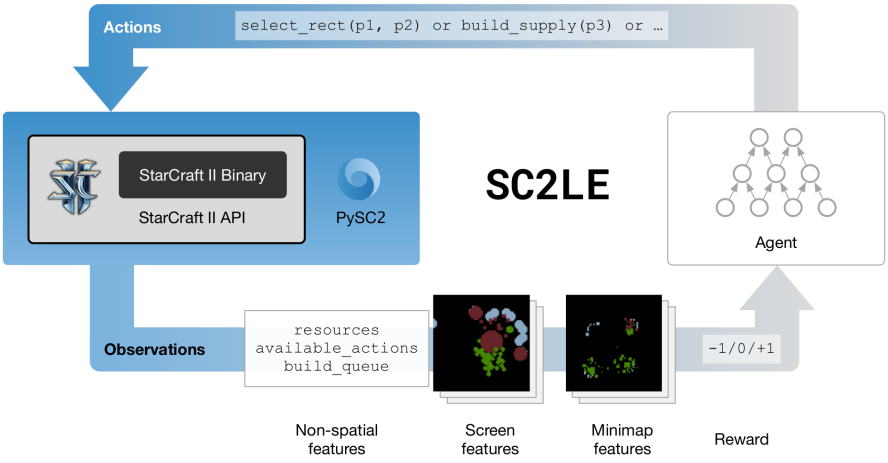

Abbildung 3.1: Interaktion des Agents mit dem StarCraft II Environment via pysc2. [VEB+17]

3.1.1 Markov Decision Process

Ein MDP lässt sich durch ein Tupel (S, A, r, p) beschreiben, wobei S die Menge aller möglichen Zustände (States) und A die Menge aller möglichen Aktionen (Actions) des Agents ist. Die Rewardfunktion $r(s, a)$ beschreibt, welche Belohnung (Reward) der Agent bei ausführen der Aktion $a \in A$ im State $s \in S$ in seiner Umgebung (Environment) erhält. Die Funktion $p(s'|s, a)$ hingegen beschreibt die Wahrscheinlichkeit, im Zustand $s' \in S$ zu landen, nachdem im Zustand s die Aktion a gewählt wurde. Eine wichtige Eigenschaft eines MDP ist die so genannte „Gedächtnislosigkeit“, was bedeutet, dass die Wahrscheinlichkeit, einen bestimmten Folgezustand zu betreten, nur vom aktuellen Zustand abhängt, und nicht von weiter zurückliegenden Vorhergehenden. Das Verhalten eines Agents wird durch seine Policy $\pi(s)$ (zu Deutsch Strategie, Konzept) bestimmt, mit der dieser eine Aktion passend zu einem gegebenen Zustand wählt. Ziel des Reinforcement Learning ist es, eine optimale Policy π^* zu finden, die den größtmöglichen Reward am Ende einer Episode verspricht. [SB18]

Die Interaktion des Agents mit dem Environment lässt sich etwas verständlicher auch durch einen Kontrollkreislauf beschreiben, siehe Abbildung 3.4. Be-

ginnend mit einer initialen Beobachtung (Observation) s_0 ($r_0 = 0$), erhält der Agent eine solche in jedem Zeitschritt, zusammen mit einem Rewardsignal, das sich auf den vorherigen Zeitschritt und die dort gewählte Aktion bezieht. Er antwortet mit einer von ihm gemäß seiner Policy gewählten Aktion $a_t = \pi(s_t)$, und bekommt die Observation des nächsten States und den erhaltenen Reward s_{t+1}, r_{t+1}.

3.1.2 Keras-rl

Keras-rl [Pla16] ist ein kleines Reinforcement Learning Framework, das als Ausgangspunkt für diese Arbeit dient, und den in 3.1.1 beschriebenen Interaktionszyklus von Agent und Environment implementiert. Es ist auf Basis des Frameworks Keras [C+15] geschrieben, welches wiederum eine high-level API für Neurale Netze ist, und als Backend TensorFlow [AAB+15], CNTK [SA16] oder Theano [ARAA+16] benutzen kann. Für dieses Projekt wurde das TensorFlow Backend benutzt. Im folgenden sind die Grundkomponenten von Keras-rl erläutert, da sie für die Implementierung wichtig sind.

Die Klasse Agent dieses Frameworks bildet hierbei den zentralen Baustein. Bei der Initialisierung übergibt man ein Modell des zu verwenden gewünschten Neuralen Netzes (Model), dessen Input-Dimensionen denen einer (vorbearbeiteten) Observation entsprechen müssen, sowie dessen Output-Layer einen Ausgang pro möglicher Aktion des Environments mit linearer Aktivierungsfunktion besitzen soll (in der Standardkonfiguration für einen DQN Agent). Außerdem übergibt man eine Policy - im Normalfall ist dies eine Epsilon-Greedy-Policy. Diese führt mit einer gewissen, im Verlauf des Lernens sinkenden Wahrscheinlichkeit ϵ zufällige Aktionen aus und wählt ansonsten „greedy“ (zu Deutsch gierig) die nach der aktuellen Evaluation des Agents beste Option.

Ein eigener Lernalgorithmus kann in den Agent eingebaut werden, in dem man dessen Methoden „_init_()“ und „backward()“ überschreibt. In „_init_()“ kann das übergebene Model noch modifiziert (siehe Dueling-DQN 3.6) sowie

eine eigene Loss-Funktion definiert werden. „backward()“ hingegen enthält den Backpropagation-Step und ist somit der richtige Platz für die eigentliche Implementierung eines eigenen Algorithmus.

Die Klasse Environment dient als Schnittstelle zur gewünschten Lernumgebung. Sie enthält (neben anderen) die simplen Methoden „reset()“ und „step()“, wobei erstere die Umgebung zurücksetzt und eine initiale Observation zurückgibt, und zweitere als Argument eine Action erhält, diese in sich ausführt und die Observation des Folgezustands sowie den dabei erhaltenen Reward zurückgibt. Zu erwähnen ist hier auch die Klasse Processor, die als Adapter zwischen dem Agent und dem Environment eingesetzt werden kann.

Herzstück des Frameworks ist nun die „fit()“ Methode (siehe Algorithmus 1) der Agent-Klasse, die diesen auf dem gegebenen Environment trainiert. Dieser Algorithmus wird für alle Implementierungen dieser Arbeit verwendet, mit leichten Änderungen, die aber auf Pseudo-Code Ebene keine Rolle spielen. Prinzipiell wird in einer Schleife immer eine neue Episode gestartet, und der Zyklus aus Abbildung 3.4 bis zum Episodenende wiederholt. Dieser besteht aus dem Wählen einer Aktion auf Basis einer Observation durch die „forward()“ Methode, die die Policy enthält, dem Ausführen dieser im Environment mithilfe der „step()“ Methode und schließlich einem Backpropagation Schritt mit den gewonnenen Erfahrungen durch die „backward()“ Funktion.

Algorithmus 1 Agent.fit() Methode

```
1: procedure FIT(env, nbsteps, maxEpisodeSteps, ...)
2:     step, episode ← 0
3:     while step < nbsteps do
4:         if obs_0 ≡ None then                          ▷ Start of a new Episode
5:             obs_0 ← env.reset()
6:             stepInEpisode, episodeReward ← 0
7:         end if
8:
9:         action ← self.forward(observation)            ▷ Choose Action
10:        obs_1, reward, done ← env.step(action)        ▷ Execute Action
11:
12:        if stepInEpisode ≥ maxEpisodeSteps then       ▷ Force End
13:            done ← True
14:        end if
15:
16:        self.backward(obs_0, action, reward, obs_1, done)   ▷ Learning Step
17:
18:        episodeReward ← episodeReward + reward
19:        stepInEpisode ← stepInEpisode + 1
20:        step ← step + 1
21:        obs_0 ← obs_1
22:        if done then                                  ▷ End Episode
23:            episode ← episode + 1
24:            obs_0 ← None
25:        end if
26:    end while
27: end procedure
```

3.2 Q-Learning

Q-Learning [Wat89] ist ein populärer Algorithmus für off-policy Temporal-Diffrence-Learning (TD). Die Idee dahinter ist, eine Action-Value-Funktion, auch Q-Funktion genannt, zu lernen. Diese erhält als Eingabe einen State s_t und eine Action a_t, und trifft eine Aussage über den Wert dieser Aktion in diesem Zustand. Die Ausgabe ist die geschätzte Belohnung, die der Agent am Ende der Episode sammeln konnte, nachdem er diese Aktion im gegebenen Zustand gewählt hat. Die optimale Q-Funktion wird auch Q^* genannt, und ist definiert durch Gleichung 3.1. Die Idee dahinter ist, dass wenn man die genauen Q-Werte für alle State-Action-Paare kennen würde, man in jedem Zustand die Aktion mit dem höchsten Q-Wert wählen könnte, und damit automatisch einer optimalen Policy π^* folgt.

3.1: Definition der optimalen Q^*-Funktion [MKS+13]

$$Q^*(s,a) = \mathbb{E}_{s'}[r + \gamma \max_{a'} Q^*(s',a')|s,a] \tag{3.1}$$

Um diese optimale Q^*-Funktion möglichst genau zu berechnen, ist die Idee hinter Q-Learning, deren Definition als iterative Updateregel zu verstehen, siehe Gleichung 3.2. α ist dabei die Lernrate, die bestimmt, wie stark Erfahrungen den zuvor erarbeiteten Q-Wert beeinflussen dürfen. Ein α von 0.1 wird oft verwendet. S_t und S_{t+1} bezeichnen den State des Environments zu den Zeitpunkten t und t+1. γ beschreibt den so genannten Discountfaktor, der langfristige gegen kurzfristige Rewards abwägt. Ein typischer Wert hierfür ist 0.99 oder gar 0.999. Der Einfluss von weit in der Zukunft liegenden Erfolgen nimmt damit langsam aber stetig ab. Die Konvergenz dieses Algorithmus gegen Q^* ist gewährleistet, solange kontinuierlich alle Q-Werte geupdatet werden [SB18].

3.2: Q-Learning Algorithmus [Wat89]

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t)] \tag{3.2}$$

3.3 Deep Q-Networks

In der Praxis schlägt sich Q-Learning nicht besonders gut. Zum einen ist es für große Anzahlen an möglichen Zuständen und/oder Actions sehr speicherintensiv, da es für jede mögliche Kombination aus beiden einen zugehörigen Q-Wert speichert, sowie um gute Resultate zu erzielen auch jede einzelne dieser Situationen auch mehrfach besuchen muss, um die entsprechenden Werte zu verfeinern. Dies ist beispielsweise im Fall von StarCraft II mit heutiger Rechenleistung nicht praktikabel bei einem Action-Space von ca. 10^8 Aktionen und einem nochmals deutlich größeren Observation-Space [VEB+17]. Außerdem generalisiert Q-Learning nicht - ein Update eines einzelnen Q-Werts verbessert nur das Wissen für dieses eine Zustand/Action Paar - eine deutlich bessere Lösung ist es, die Q-Funktion durch einen Funktionsapproximierer zu schätzen. Eine mögliche Wahl hierfür sind Neurale Netzwerke, die namensgebend für den Deep Q-Networks (DQN) Algorithmus sind. Anstelle der Update-Regel aus 3.2 tritt dann die Loss-Formel 3.4, die von dem Netzwerk minimiert wird. Dabei hängt die neue Definition der Q-Funktion nun auch von den Gewichten des neuralen Netzwerks θ ab. Die Variablen s und a beschreiben dabei einen Zustand und eine darin gewählte Aktion, s' den daraus resultierten Folgezustand und r den dabei erhaltenen Reward. Da es bei komplexen Problemen nicht trivial oder gar möglich ist, alle möglichen Zustände/Aktionen zu besuchen, wird zudem das Konzept des „stochastic gradient descent“ benutzt: Ein Minimierungsschritt wird auf einer zufällig gewählten Teilmenge gesammelter Erfahrungen berechnet [MKS+13].

3.3: DQN Target

$$y^{DQN} = r + \gamma \max_{a'} Q(s', a'; \theta^-) \tag{3.3}$$

3.4: DQN Loss [MKS+13]

$$L(\theta) = \mathbb{E}_{s,a,s'}[(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta^-))^2] \tag{3.4}$$

Der Algorithmus wurde dafür um ein so genanntes „Experience Replay Memory“ erweitert, in welchem alle Erfahrungen gespeichert werden, und aus dem

in jedem Schritt zufällig ein Mini-Batch (meistens 32 Erfahrungen) gezogen wird, auf Basis von welchen dann ein Netzwerkupdate ausgeführt wird - weitere Informationen dazu stehen in Abschnitt 3.7. Zudem wird eine Struktur mit einem zweiten „Target-Network“ benutzt mit Gewichten θ^-, welche eine eingefrorene Kopie der Gewichte des Q-Networks θ sind, und auf Basis dessen nun das Target 3.3 bzw. der Loss 3.4 berechnet wird. Nach einer fixen Anzahl an Schritten τ werden die Gewichte des Q-Networks auf das Target-Network kopiert. Beide Erweiterungen verbessern die Performance des Algorithmus deutlich [MKS+15].

3.4 FullyConv Architektur

In dem bereits zitierten SC2LE Paper [VEB+17] stellt DeepMind eine Netzwerkarchitektur vor, die sie „FullyConv“ nennen. Diese ist entstanden unter dem Vorbild des Agents aus [MKS+13] und [MKS+15], welcher auf diversen Atari-Spielen hervorragende Performance erzielte. Zwei Inputs erhalten jeweils die Screen-Feature-Layers beziehungsweise die Mini-Map-Feature-Layers einer Observation. Diese werden durch jeweils zwei 2D-Convolution-Layers - die erste mit 16 Filtern der Größen $5 * 5$, die zweite mit 32 Filtern der Größe $3 * 3$ - verarbeitet. Alle Convolution-Operationen im Netzwerk haben absichtlich keinen Stride (entspricht Stride 1) und benutzen entsprechendes Padding, um die genaue 2D-Dimension des Inputs zu erhalten. Die beiden Netzwerkzweige werden anschließend aneinandergehängt. Danach teilt sich das Netzwerk wieder: Eine Dense-Layer mit 256 Einheiten gefolgt von einer weiteren Dense-Layer mit einer Einheit pro Aktion - Aktion im Sinne der „pysc2“-API ohne Argumente - stellen einen der Outputs des Netzwerks dar. Den zweiten Zweig bildet eine weitere 2D-Convolution-Layer mit einem einzigen Filter der Größe $1 * 1$. Dies stellt den zweiten Output des Netzwerks dar, welcher die Dimension des Bildschirms (des Inputs) hat, siehe Abbildung 3.2. Um diese Architektur benutzen zu können, mussten diverse Keras-rl Klassen umgeschrieben werden, da das Framework nur Netzwerke mit einem Output unterstützt.

Das Netzwerk berechnet (im Gegensatz zu der theoretischen Darstellung in Kapitel 3.3) damit nicht zu einem State-Action Paar eine zugehörige Q-Value. Um wiederholte Evaluation des Netzwerks zu vermeiden, berechnet es mit der Eingabe eines States die Q-Werte für alle möglichen Aktionen. In diesem speziellen Fall, Aufgrund des großen Action-Spaces, sind es allerdings sogar 2 Q-Values, die sich wie folgt ergeben: Die Q-Values des linearen Outputs geben an, welcher Reward erwartet wird, wenn eine der Actions gewählt wird, unabhängig von deren Argumenten. Der zweidimensionale Output hingegen gibt an, an welcher Position auf dem Bildschirm bei gewählter Aktion (die Position ist als Argument dieser in „pysc2“ zu verstehen) welcher Reward zu erwarten ist. Eine „greedy“ Policy ergibt sich nun, indem man immer die Aktion mit dem höchsten Q-Wert zusammen mit den Koordinaten des höchsten Q-Werts im zweidimensionalen Output kombiniert (falls die Aktion positionelle Argumente unterstützt, ansonsten wird dieser Teil ignoriert). Dadurch ergeben sich zwei neue Loss-Funktionen, deren Summe minimiert wird, siehe Gleichung 3.5. Der Index einer Action a_a bzw. a_b bezieht sich auf die lineare sowie die zweidimensionale Komponente. Da das Netzwerk wie bereits erwähnt eigentlich nicht das Interface $Q(s, a; \theta)$ besitzt sondern $Q(s; \theta)$, wird der Loss gemäß Gleichung 3.5 nur für die gewählte Action berechnet und für alle anderen auf Null gesetzt (maskiert).

3.5: FullyConv Loss

$$
\begin{aligned}
L_a(\theta) &= \mathbb{E}_{s,a_a,r,s'}[(r + \gamma \max_{a'_a} Q(s', a'_a; \theta) - Q(s, a_a; \theta))^2] \\
L_b(\theta) &= \mathbb{E}_{s,a_b,r,s'}[(r + \gamma \max_{a'_b} Q(s', a'_b; \theta) - Q(s, a_b; \theta))^2] \\
L(\theta) &= L_a + L_b
\end{aligned}
\tag{3.5}
$$

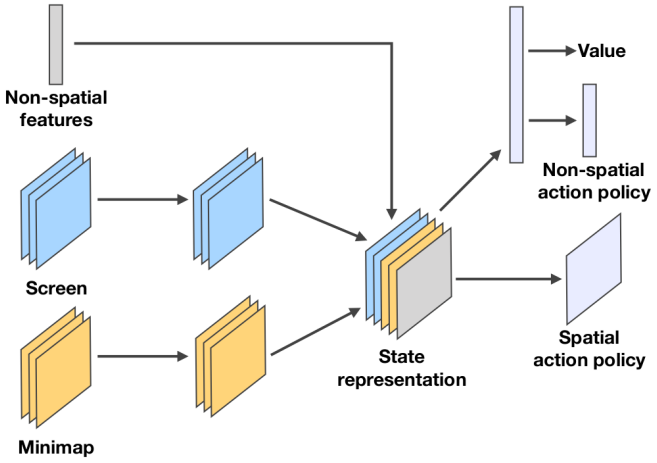

Abbildung 3.2: Darstellung des FullyConv Netzwerks. [VEB+17]

3.5 Double DQN

Ein weiteres Problem, an welchem sowohl Deep Q-Learning als auch klassisches Q-Learning leiden, ist, dass beide Algorithmen dazu neigen, während dem Lernprozess überschätzte Werte eher in einem Update zu verwenden. Dies führt zu einer generellen Überschätzung vieler Q-Werte und leider dadurch auch zu suboptimalen Policys, wie das Paper [VHGS16] deutlich nachweist. Darin schlagen die Autoren die Abwandlung „Double DQN“ des DQN Algorithmus vor, um diesem Problem zu begegnen. Die Idee dahinter ist, nicht mehr das gleiche Netzwerk für die Selektion der Aktion und die Schätzung ihres Wertes zu benutzen. Gleichung 3.6 zeigt das Target des DQN-Algorithmus umgeschrieben, um die Wahl der Aktion (argmax) und die Evaluation des Q-Wertes dieser getrennt darzustellen. Die folgende Gleichung 3.7 benutzt das „online“-Netzwerk (Gewichte θ) für die Wahl von a' und das in Abschnitt 3.3 eingeführte Target-Network (Gewichte θ^-) zur Evaluation dieser.

$$y^Q = r + \gamma Q(s', \underset{a'}{\operatorname{argmax}}\, Q(s', a'; \theta^-); \theta^-) \tag{3.6}$$

$$y^{DoubleQ} = r + \gamma Q(s', \underset{a'}{\operatorname{argmax}}\, Q(s', a'; \theta); \theta^-) \tag{3.7}$$

3.6 Dueling DQN

Dueling Networks [WSH+15] versuchen, durch eine spezielle Netzwerkarchitektur das Lernen einer Value-Funktion und einer Action-Advantage-Funktion zu trennen. Eine Value-Funktion $V(s)$ gibt, der Namensgebung folgend, auf Basis eines States dessen Wert aus, also die Summe aller zukünftigen Rewards, die von diesem aus zu erwarten sind (durch ein γ zeitlich gewichtet). Eine Action-Advantage-Funktion $A(s, a)$ soll ausgeben, wie die übergebene Action den übergebenen State verbessern oder verschlechtern würde. Zusammengefasst ergeben beide wieder exakt die Definition der bisher betrachteten Q-Funktion 3.8.

Einer der Vorteile dieser Trennung ist, das für jeden Lernschritt unabhängig der Action immer die Value-Funktion verfeinert wird, dabei aber die Action-Advantages nicht beeinflusst werden, was eine höhere Lerneffizienz bedeutet. Besonders bei großen Action-Spaces sei dies laut dem Paper sehr deutlich. Um diese Trennung in der Netzwerkarchitektur abzubilden, wird die letzte Dense-Layer durch eine Verzweigung ersetzt. Ein Zweig hat die Dimension des ursprünglichen Outputs, eine Dense-Layer mit einem Node pro möglicher Aktion, die die Action-Advantages für jede dieser ausgeben soll. Der zweite Zweig ist auch eine Dense-Layer, allerdings mit nur einem einzigen Node, deren linearer Output der Value entsprechen soll. Durch eine spezielle Lambda-Layer werden nun beide Zweige zum ursprünglichen Q-Value Output rekombiniert. Diese Layer besitzt allerdings keine Gewichte (repräsentiert also nur eine fixe mathematische Operation).

Die naive Version, die Gleichung 3.8 zu verwenden, ist allerdings wenig praktikabel, da man anhand einer Q-Value des Outputs nicht mehr zurückrechnen kann zu den ursprünglichen Value- und Action-Advantage-Komponenten. Dies ist einfach zu zeigen, indem man zur Value eine beliebige Konstante addiert, und selbige von der Action-Advantage abzieht. Die resultierende Q-Value bleibt unverändert: $Q(s, a) = V(s) + A(s, a) = (V(s) + c) + (A(s, a) - c)$. Selbiges gilt auch für den Backpropagation Schritt des Lernprozess, was zur Folge hat, dass keine separaten Value- und Action-Advantage-Funktionen gelernt werden,

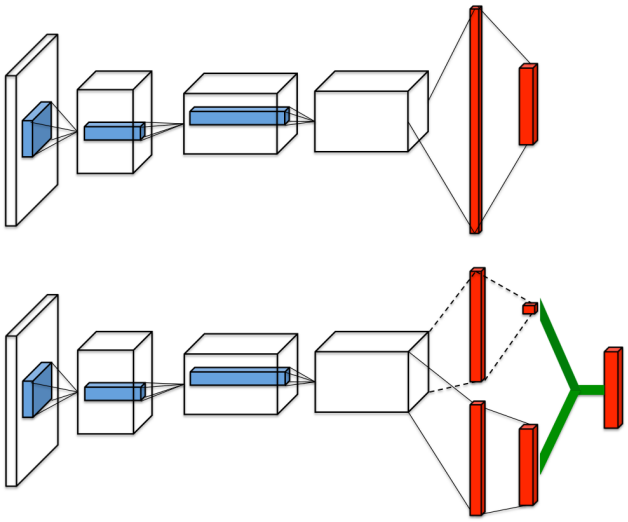

Abbildung 3.3: Darstellung der Dueling-Networks Architektur gegenüber einem normalen ConvNet. [WSH+15]

und die Vorteile der Architektur nicht zum Vorschein kommen (beispielsweise könnte der Value-Branch des Netzwerks stets die gleiche Konstante ausgeben, und das Resultat wäre ein normales DQN, das den Value-Branch ignoriert und dessen Action-Advantage Branch Q-Werte ausgibt (um eine Konstante verschoben in diesem Beispielt)).

Um dies zu vermeiden, wird die letzte Ebene stattdessen mit der folgenden beiden Gleichung ausgestattet: 3.9. Damit zwingt man die Action-Advantage Funktion für die beste Option im gegebenen Zeitschritt einen Wert von Null anzunehmen. Eine Variante, die zusätzliche Stabilität mit sich bringt und daher bevorzugt verwendet wird, ist in Gleichung 3.10 zu sehen, welche anstelle eines Max-Operators den Durchschnitt aller Action-Advantages benutzt. Diese drei Varianten werden in Keras-rl als die Dueling-Modes „naive“, „max“ und „mean“ bezeichnet.

$$Q(s,a) = V(s) + A(s,a) \tag{3.8}$$

$$Q(s,a) = V(s) + (A(s,a) - \max(A(s,*))) \tag{3.9}$$

$$Q(s,a) = V(s) + (A(s,a) - \text{mean}(A(s,*))) \tag{3.10}$$

Da das FullyConv-Network zwei Outputs besitzt, wurde die Dueling Architektur auf beide separat angewandt. Bei dem linearen Output ist die Implementierung wie eben beschrieben. Der zweidimensionale Output benuzt genauso eine Dense-Layer mit einem einzelnen Node für den Value-Teil. Jedoch für die Action-Advantages, die die Dimension beibehalten müssen, wird eine $1 * 1$ Convolutional Layer mit einem einzelnen Filter benutzt. Für alle Experimente wurde „mean“ verwendet. Das Netzwerk kann aufgrund der beibehaltenen Output-Dimension wie gewohnt trainiert werden.

3.7 Prioritized Experience Replay

Der bisher beschriebene Agent benutzt ein in Abschnitt 3.3 eingeführtes Replay Memory, welches alle Erfahrungen speichert, während sich dieser in einer Umgebung bewegt. Aus diesem wird für die jeweiligen Lernschritte eine zufällig gewählte Teilmenge (Mini-Batch) gewählt. Als Ergänzung zum vorherigen Kapitel sei zu erwähnen, dass das Memory eine endliche Größe hat und nach dem „FIFO“-Prinzip - sollte es voll sein - stets die älteste Erfahrung löscht. Die Verwendung eines Replay Memorys verbessert die Stabilität von Q-Learning deutlich. Der Grund dafür ist, dass die zufällige Auswahl der Transitionen eventuelle zeitliche Korrelationen aufbricht (in Bezug auf erhaltenen Reward), da ältere und neuere Erfahrungen gleichermaßen für Updates benutzt werden. Es bietet außerdem die Chance, dass seltene Erfahrungen mehr als einmal wiederholt werden. Dennoch ist dieses Konzept nicht optimal, da beispielsweise Erfahrungen, die viel neue Information enthalten (zum Beispiel das Entdecken einer neuen, besseren Strategie) genauso oft wiederholt werden wie Erfahrungen, die kaum bis keine Informationen enthalten.

Dieses Potential freizusetzen war das Ziel hinter Prioritized Experience Replay [SQAS15]. Die Idee dahinter ist, jeden Speichereintrag mit einer Priorität zu versehen, die in Relation zum jeweiligen TD-Error steht. Die Intuition dahinter ist, Transitionen zu bevorzugen, die für das Netzwerk „überraschend“ waren, also für welche die Vorhersage des Netzwerks am meisten abgewichen ist von

dem tatsächlich Erlebten. Es werden weiter stochastisch Erfahrungen aus dem Memory gezogen, nun jedoch ist die Wahrscheinlichkeit für jede Erfahrung, gezogen zu werden, definiert durch Gleichung 3.11. Dabei ist p_i die Priorität der Transition i, k beschreibt die Größe des Memorys und der Hyperparameter α legt fest, wie viel Priorisierung benutzt wird ($\alpha = 0$ entspricht normalem Experience Replay ohne Priorisierung). p_i kann auf zwei verschiedene Arten berechnet werden, einmal durch direkte Zuweisung des TD-Errors $|\delta|$ plus einer kleinen positiven Konstante ϵ, die sicherstellt, dass eine Transition nicht nie mehr besucht wird (siehe Gleichung 3.12), und einmal durch eine Rang-basierte Priorität (siehe Gleichung 3.13), wobei $rank(i)$ die Position der Erfahrung i im Replay-Memory bei einer Sortierung nach den TD-Errors $|\delta|$ beschreibt.

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{3.11}$$

$$p_i = |\delta_i| + \epsilon \tag{3.12}$$

$$p_i = \frac{1}{rank(i)} \tag{3.13}$$

Für die Integration in den bisherigen FullyConv-Agent wurde eine Implementierung aus den OpenAI Baselines [DHK+17] in Form der Klasse „PrioritizedReplayBuffer“ benutzt und die Agent-Klassen von Keras-rl passenderweise modifiziert. Im Gegensatz zur vorher benutzten Memory Klasse werden nun nicht mehr 3-Tupel der Form (s, a, r) gespeichert, da für einen Lernschritt bisher immer zwei aufeinander folgende Transitionen zusammen gezogen werden mussten, um die dafür benötigten Komponenten s_t, a_t, r_t, s_{t+1} zu kennen. Nun möchte man jede Erfahrung mit einer Priorität versehen, weswegen sich das Speichern von 4-Tupeln der Form (s_t, a_t, r_t, s_{t+1}) anbietet. Spätestens für die MultiStep-Erweiterung in Kapitel 3.8 war diese Designentscheidung sehr wertvoll. Neue Erfahrungen werden mit maximaler Priorität eingefügt, was eine leichte Bevorzugung neuer Transitionen erzeugt, wie in [HMVH+18] vorgeschlagen wurde. In jedem Lernschritt wird außerdem nun in der „backward()“-Funktion nach der Backpropagation (dem eigentlichen Netzwerkupdate) ein weiterer „forward()“ Call gemacht. Mit dessen Ergebnis wird der TD-Error erneut bestimmt und im Memory aktualisiert.

3.8 MultiStep DQN

Q-Learning als Temporal-Diffrence (TD) Methode aktualisiert auf Basis eines einzelnen Schritts im Environment und einer darauf folgenden Vorhersage seine Erfahrung. Dies ist verglichen mit Monte-Carlo Methoden, welche ihre Erfahrung auf Basis von einer ganzen Episode auf einmal aktualisieren, auch eine Art Extremfall. Im Buch [SB18] propagieren die Autoren, dass keine der beiden Methoden immer im Vorteil ist. Sie präsentieren n-Schritt TD Methoden, welche eine beliebige Anzahl an Schritten als Basis für die Erfahrungsupdates benutzen können. Die Autoren des Rainbow-DQN Papers [HMVH$^+$18] definieren einen dem „off-policy n-Step Expected-Sarsa“ ähnlichen n-Step Q-Learning Algorithmus, dessen Target in Gleichung 3.15 definiert ist. Implementierungstechnisch musste dafür nur die Art geändert werden, wie Erfahrungen im Replay-Memory abgespeichert werden. Anstatt dem Tupel (s_t, a_t, r_t, s_{t+1}) (siehe Abschnitt 3.7) wird nun $(s_t, a_t, R_t^{(n)}, s_{t+n})$ gespeichert, wobei der n-Step Return $R_t^{(n)}$ in Gleichung 3.14 als aggregierter, mit γ gewichteter Reward über n Schritte definiert ist. Anschließend musste nur die Target-Formel des Netzwerks gemäß 3.15 abgeändert werden. In allen Experimenten wurde eine Schrittweite von $n = 3$ verwendet, welche in [HMVH$^+$18] vorgeschlagen wurde.

$$R_t^{(n)} \equiv \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1} \tag{3.14}$$

$$y_t^n = R_t^{(n)} + \gamma^n Q(s_{t+n}, \underset{a'}{\operatorname{argmax}}\, Q(s_{t+n}, a'; \theta); \theta^-) \tag{3.15}$$

3.9 Noisy Nets

Effiziente Erkundung der Umgebung sowie das Erforschen neuer Strategien ist eines der Kernprobleme des Reinforcement Learnings. Die weit verbreitete in Kapitel 3.1.2 vorgestellte Methode ist die so genannte Epsilon-Greedy-Policy. Diese führt mit einer Wahrscheinlichkeit ϵ in jedem Zeitschritt statt der für den

Agenten in diesem Moment subjektiv besten Option eine zufällige Aktion aus. Der Wert von Epsilon wird im Laufe des Trainingsprozesses verringert und schließlich bei einem kleinen Wert nahe Null eingefroren (typischerweise 1% o.ä.). Diese Methode hat allerdings auch klare Schwächen, welche beispielsweise in Umgebungen zutage treten, welche viele Aktionen erfordern, bevor ein erster Reward ausgeschüttet wird.

Ein neuer Ansatz hierfür sind „Noisy Nets“ [FAP+17]. Dabei soll das Netzwerk Kontrolle über den eingebrachten Zufall (Noise) haben durch spezielle Netzwerk-Layers. Diese besitzen Gewichte, die den Einfluss von Zufallsvariablen regulieren und können via Gradient-Descent trainiert werden. Der Agent kann damit die Intensität des Zufalls beziehungsweise dessen Einfluss auf spezielle Aspekte des Netzwerks steuern. Außerdem kann er lernen, den Zufall nach und nach zu ignorieren, und keine von außen vorgegebenen Parameter (wie beispielsweise ϵ aus Epsilon-Greedy) sind mehr nötig.

$$y = wx + b \tag{3.16}$$

$$y \overset{def}{=} (\mu^w + \sigma^w \odot \epsilon^w)x + \mu^b + \sigma^b \odot \epsilon^b \tag{3.17}$$

Eine normale Dense-Layer eines neuralen Netzwerks, welche durch Gleichung 3.16 beschreiben wird, multipliziert den Input-Vektor x mit einer Matrix w an trainierbaren Gewichten und addiert danach den trainierbaren Bias-Vektor b. Auf Basis dieser Definition wird nun durch die Gleichung 3.17 die zugehörige Noisy-Dense-Layer definiert („noisy linear layer“ im Original [FAP+17]). Diese ersetzt die Gewichtsmatrix w durch drei gleich große Matrizen μ^w, σ^w und ϵ^w. Letztere ist dabei gefüllt mit Zufallszahlen, welche normalverteilt zwischen Null und Eins liegen. Diese wird elementweise mit der Matrix σ^w multipliziert, welche trainierbare Gewichte enthält, die die Intensität des Zufalls regulieren sollen ($\odot$ beschreibt die elementweise Multiplikation). Anschließend addiert man die entstandene Matrix, die den gewichteten Zufall repräsentiert, zu einer weiteren Gewichtsmatrix μ^w, welche die Stelle der ursprünglichen Matrix w einnimmt (wäre der Zufall Null, würde μ^w exakt diese Rolle erfüllen). Da diese Layer nicht in Keras [C+15] oder Keras-rl [Pla16] enthalten ist, musste sie

selbst implementiert werden. „NoisyDense“ erbt nun von Dense (Keras linear layer) und besitzt die gleichen Schnittstellen, kann also in bestehendem Code einfach ausgetauscht werden. Nach Empfehlung in [FAP+17] wurde dies nur für die letzten Ebenen vor dem Output gemacht.

$$y = conv(x, w) + b \tag{3.18}$$

$$y \stackrel{def}{=} conv(x, \mu^w + \sigma^w \odot \epsilon^w) + \mu^b + \sigma^b \odot \epsilon^b \tag{3.19}$$

Das Prinzip der Noisy Networks musste nun auch auf eine Convolutional-Layer übertragen werden, da das verwendete Netzwerk der FullyConv Architektur folgt. Dies bedeutet, dass ein Teil des Outputs nur durch solche mit dem Input verbunden ist, in diesem Fall speziell 2D-Convolutional-Layers (kurz Conv2D). Eine solche besitzt einen vierdimensionalen Filterkernel an Gewichten mit den Dimensionen (Anzahl Input-Ebenen, Input X-Dim, Input Y-Dim, Anzahl Filter) sowie einen Gewichtsvektor der Länge der Anzahl der Filter, welcher als Bias zu dem Output addiert wird. Notiert man die Convolution Operation analog zur Notation der Dense-Layer, kann man eine NoisyConv2D-Layer definieren durch Gleichung 3.19. „$conv(x, w)$“ beschreibt dabei abstrakt die Convolution-Operation in Abhängigkeit von ihrem Input x und dem Filterkernel w. Wie dort zu sehen ist, ersetze ich den Filterkernel durch drei Stück der gleichen Größe, wovon ϵ^w gefüllt mit normalverteilten Zufallszahlen ist, während μ^w und σ^w trainierbare Gewichte enthalten. $\odot$ und $+$ sind beides elementweise Operationen. Der Bias-Vektor wird wie in der NoisyDense-Layer behandelt. Die von der Conv2D-Klasse (Keras [C+15]) erbende entstandene Klasse „NoisyConv2D“ besitzt die gleichen Schnittstellen wie ihre Elternklasse, und kann damit ebenfalls im Code einfach ausgetauscht werden.

3.10 Rainbow-DQN

Dieser Abschnitt präsentiert den fertigen Rainbow DQN Agent ([HMVH+18] und [VEB+17]), der im Folgenden allerdings „FullyConv V10“ genannt wird, da eine der originalen Rainbow-Komponenten fehlt. Aus zeitlichen Gründen

konnte „Distributional Reinforcement Learning“ nicht mehr implementiert werden. Diese Erweiterung wird dennoch in Kapitel 3.11.2 als mögliche Verbesserung beschrieben.

3.10.1 FullyConv V10

Das Gehirn des fertigen Agents bildet ein Convolutional-Neural-Net, sehr ähnlich zu dem FullyConv-Agent aus Kapitel 3.4. Die Unterschiede sind unter anderem mit einer Beschneidung der Größe des Input-Spaces begründet: es erhält als Input zwei $(32 * 32)$ Pixel große Feature-Layers im Gegensatz zu bis zu dreizehn, und keine Mini-Map Feature-Layers vom Environment, da die betrachteten Mini-Games alle auf den Sichtbereich eines Screens beschränkt sind. Der zugehörige zweite Input des Netzwerks fällt damit weg. Der verbleibende Input wird verarbeitet von einer Conv2D-Layer mit 16 $(5 * 5)$ großen Filtern, gefolgt von einer zweiten mit 32 $3 * 3$ großen Filtern. Beide sind ohne Stride und mit entsprechendem Padding ausgestattet, um die Dimension des Inputs beizubehalten. An dieser Stelle teilt sich das Netzwerk. Ein Zweig besteht aus einer Dense-Layer mit 256 Nodes, gefolgt von einer weiteren Dense-Layer mit drei Nodes (einem für jede Action). Der zweite Zweig behält die Zweidimensionalität bei und besteht aus einer weiteren Conv2D-Layer mit einem einzigen Filter der Größe $(1*1)$. Alle Layers haben „ReLU“-Aktivierungsfunktionen, bis auf die beiden Outputs, welche eine lineare Aktivierungsfunktion besitzen.

Der Agent ist vollständig Modular programmiert, so dass jede der Rainbow-Erweiterungen separat ein- und ausgeschaltet werden kann. Die Erweiterung „Noisy Nets“ (Kapitel 3.9) ändert die Struktur des Netzwerks, indem sie die beiden Dense-Layers durch NoisyDense-Layers sowie die letzte Conv2D-Layer (mit $(1*1)$ Filter) durch eine NoisyConv2D-Layer austauscht. Ebenfalls direkte Änderungen am Netzwerk werden durch das Einschalten von „Dueling DQN“ hervorgerufen, welche ausführlich in Kapitel 3.6 beschrieben werden.

Das Prioritized Replay Memory kann deaktiviert werden, was dieses durch ihre Elternklasse „ReplayBuffer“ ersetzt und den in Kapitel 3.7 beschriebenen zu-

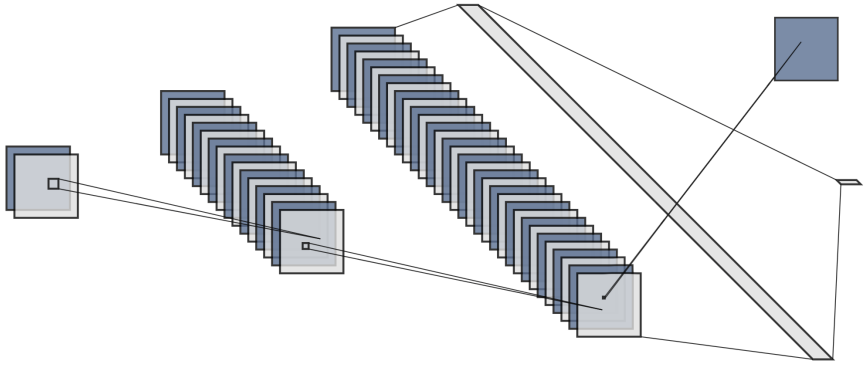

Abbildung 3.4: Neurales Netz des FullyConv V10 Agents ohne Dueling.

sätzlichen „forward()“-Aufruf zur Bestimmung der neuen Prioritäten weglässt. Ansonsten entspricht dessen Implementierung der im Kapitel beschriebenen.

Das Einschalten von Double DQN, wie in Kapitel 3.5 beschrieben, tauscht das Netzwerk zur Bestimmung der Aktion mit dem maximalen Q-Wert. MultiStep DQN 3.8 hingegen benutzt zwei kleine Ring-Buffer der Größe der Schrittweite n, um die nötige Information für einen mehrere Schritte umfassenden Eintrag ins Replay-Memory bereitzustellen. Der Reward der n Schritte wird aggregiert und zeitlich gewichtet gespeichert und das Target wird entsprechend angepasst.

3.11 Mögliche Verbesserungen

Dieser Abschnitt diskutiert mögliche Verbesserungen des „FullyConv V10“ Agent (Kapitel 3.10).

3.11.1 Hyper-Parameter Tuning

Durch die relativ geringe Anzahl an Versuchen, die aufgrund begrenzter Rechenleistung in der gegebenen Zeit möglich waren, mussten viele der Hyperparameter festgelegt werden, ohne zu testen, wie sie sich auf das Ergebnis auswirken. Besonders die Schrittweite der MultiStep Erweiterung, welche auf den Wert 3 nach [HMVH+18] gesetzt wurde, verspricht eventuelle Verbesserungen. Da jede Erweiterung eigene Hyperparameter mitbringt, haben schon die Autoren des eben zitierten Rainbow DQN Papers angemerkt, dass sie nur begrenztes Tuning durchführen konnten, welches allerdings weitere Performance-Vorteile verspricht.

3.11.2 Distributional Reinforcement Learning

Distributional Reinforcement Learning [BDM17] ist eine Modifikation für den DQN-Algorithmus, welche anstelle des erwarteten zukünftigen Rewards für ein State-Action-Paar (definition der Q-Funktion, siehe Gleichung 3.1) eine Wahrscheinlichkeitsverteilung von diesem lernt. Dafür haben die Autoren des Papers einen neuen Algorithmus entworfen, der die Bellmannsche Gleichung 3.20 auf einen verteilten Fall anwendet: Gleichung 3.21. Die Funktion Z gibt dabei eine Wahrscheinlichkeitsverteilung über den zukünftig erwarteten Reward beim Ausführen von Aktion a im State s an, deren Erwartungswert dem Q-Wert dieses State-Action Paars entsprechen soll. Die Verteilung ist charakterisiert durch drei Zufallsvariablen: den Reward $R(s, a)$, das nächste State-Action Paar (S', A') und dessen stochastische Rewardverteilung $Z(X', A')$ [BDM17].

$$Q(s, a) = \mathbb{E}R(s, a) + \gamma \mathbb{E}Q(S', A') \tag{3.20}$$

$$Z(s, a) \stackrel{def}{=} R(s, a) + \gamma Z(S', A') \tag{3.21}$$

In der Praxis hat diese Methode deutliche Verbesserungen gegenüber normalem DQN zeigen können im Atari 2600 Benchmark, und auch der in [HMVH+18] empirisch gemessene Einfluss auf die Gesamtperformance des Rainbow-DQN

Agents (im selben Benchmark) gehört zu den drei stärksten. Dies ist die vielversprechendste Verbesserung, die der „FullyConv V10“ Agent (Kapitel 3.10) erfahren könnte.

3.11.3 „Schlechtes“ Prioritized Experience Replay

Ein Programmierfehler bei der Implementierung des Prioritized Experience Replay aus Kapitel 3.7 blieb einige Zeit unentdeckt, weil die Performance des den Fehler enthaltenden Agents hervorragend war. Nach beheben des Fehlers zeigt sich eine durchweg unstabilere Leistung, welche langsamer gegen einen schlechteren Wert konvergiert. Den Ursprung dieser Verbesserung zu finden ist vielversprechend. Nachdem sechs Testläufe (drei Tage Rechenzeit) mit unterschiedlichen Variationen des fehlerhaften sowie des korrekten Algorithmus einen weiteren Programmierfehler der vermeintlich korrekten Implementierung ans Licht brachten, hob die Korrektur dessen die Leistung nur auf in etwa das gleiche Niveau des fehlerhaften Algorithmus. Da bereits ein Großteil der Versuche mit der fehlerhaften Version durchgeführt worden waren, und die Performance durch die Reparatur nicht verbessert wurde, ist um der Einheitlichkeit willen diese Version weiterverwendet worden. Im Code kann zwischen beiden Versionen durch setzen eines Hyperparameters umgeschaltet werden. Dennoch reicht die Anzahl der Versuche bei weitem nicht aus, um eine statistisch signifikante Aussage treffen zu können. Die folgende Tabelle 3.1 fasst die Versuche hierzu zusammen.

Die genaue Implementierung ist im Quellcode nachzulesen, sie benutzt als Target für die TD-Error-Berechnung aller Erfahrungen im Mini-Batch den selben Wert, welcher das Target der letzten Erfahrung des Batches vor dem Lernupdate ist, da eine Schleifenvariable nicht initialisiert wurde. Warum dies überhaupt funktioniert, ist unklar.

Tabelle 3.1:					
Bad PER - G			Bad PER + G		
MEAN	MAX	σ	MEAN	MAX	σ
75,72	99	10,57	72	99	10,81
79,05	104	12,61			
Good PER - G			Good PER + G		
64,10	92	15,43	65,34	94	10,20
			korr. Good PER + G - Noisy		
			69,17	96	11,58

TABELLE 3.1: MoveToBeacon: Good PER VS Bad PER; (+/-) G beschreibt, ob Gewichte für off-Policy Korrektur verwendet wurden. Die letzte Messung benutzt eine korrigierte (korr.) Version des Good PER, wie in 3.11.3 beschrieben, verwendet aber keine NoisyNets (um Zeit zu sparen).

3.11.4 Noisy Nets: Factorised Gaussian Noise

Bei der Implementierung der NoisyNet-Layers wurde so genannte „independent Gaussian noise“ verwendet. Dies bedeutet, dass jedes Gewicht der Gewichtsmatrix und des Biasvektors einen unabhängig berechneten Zufallswert erhalten. Eine mögliche Variante namens „factorised Gaussian noise“ reduziert die dafür nötige Rechenleistung, indem anstelle einer Matrix und eines Vektors an Zufallswerten nun nur noch zwei Vektoren der Dimension des Inputs und der Dimension des Bias zufällig generiert werden. Durch das Produkt beider wird die benötigte Matrix an Zufallswerten berechnet. Da die NoisyNets Erweiterung aktuell den größten Performanceverlust zur folge hat, ist diese Änderung vielversprechend für schnelleres Testen.

3.11.5 Besseres Multi-Step Q-Learning

Nach längeren Recherchen, gründlichen Lesens des Rainbow DQN Papers (kurz: RB!) [HMVH+18] sowie mehrerer Paper von Richard S. Sutton und Andrew G.

Barto sowie deren Buch „Reinforcement Learning: An Introduction“ [SB18], welche in RB! zitiert wurden in Bezug auf den Multistep-Algorithmus, bin ich zu folgendem Schluss gekommen: Der Multistep DQN Algorithmus, der laut dem RB! verwendet wird, sollte in der formulierten Form nur für einen on-policy Fall gut funktionieren. Da dies durch das Prioritized Experience Replay (Kapitel 3.7) nicht mehr gegeben ist, könnte die Kombination beider den Lernprozess destabilisieren, wie im folgenden Beispiel beschrieben:

Gegeben sei ein 2-Step DQN Algorithmus in einer einfachen Umgebung, in welcher zu einem bestimmten Zeitpunkt eine binäre Entscheidung getroffen werden kann. Dies ähnelt einer Weggabelung, deren jeweilige Folgezustände nicht mehr erreicht werden können, sollte die jeweils andere Option gewählt worden sein (die Wege treffen sich nie mehr). Nun sei einer dieser Pfade sehr lukrativ in Form von vielen Rewards, welche entlang der Folgezustände ausgeschüttet werden. Der andere Pfad verspricht keinen Reward, und auch in den Zuständen direkt vor der Entscheidung wird kein Reward ausgeschüttet.

Gemäß dem 2-Step DQN Algorithmus werden nun 3 aufeinanderfolgende States mit den 2 zugehörigen Actions und dabei erhaltenen Rewards aus dem Prioritized Replay Memory gezogen in Form des Tupels $(s_t, a_t, R_t^{(n)}, s_{t+n})$. Angenommen, diese 3 Schritte entsprechen nun genau den folgenden: Der erste State s_t steht genau einen Schritt vor der Entscheidung den guten oder schlechten Pfad zu nehmen, s_{t+1} ist der Schritt, indem die Entscheidung getroffen werden muss, und s_{t+2} ist der Folgezustand, nachdem der Agent zufällig die schlechtere Entscheidung gewählt hat (beispielsweise durch eine Epsilon-Greedy Policy). Die RB! Updateregel für den Q-wert von $Q(s_t, a_t)$ würde an dieser Stelle diesen nach unten korrigieren, da die Berechnung des n-Step Targets (siehe Gleichung 3.15) Null ergeben würde. $R_t^{(n)}$ ist Null, da für diese Umgebung angenommen wird, dass die Transition zu Zuständen vor der Entscheidung sowie zu allen Zuständen nach der negativ getroffenen Entscheidung Null ist.

Wenn man on-policy lernt, ist dies kein Problem, da dieser Spezialfall nur sehr selten auftritt (kleines Epsilon, Greedy würde immer den besseren Pfad wählen). Dementsprechend wird der Q-Wert nur minimal nach unten korrigiert.

Wenn man allerdings Prioritized Replay Memory (PER) benutzt, würde diese seltene Abzweigung sehr hoch priorisiert werden (da das kleine Target Null im Gegensatz zu dem hohen erwarteten Reward des oberen Pfads steht) und damit sehr oft wieder besucht werden. Das würde den Q-Wert für die Aktion a_t im State s_t deutlich nach unten korrigieren (je nach Hyperparametern des Replay Memorys) und eventuell gegenüber anderen Aktionen im State s_t diesen fälschlich schlechter aussehen lassen. Eine mögliche Gegensteuerung wird in Kapitel 3.4 des PER Papers [SQAS15] durch importance-sampling Gewichte vorgeschlagen, die jedoch in RB! keine Erwähnung finden. Zudem schreibt DeepMind in dem auf PER aufbauenden Paper [HQB+18]: „Furthermore, we are using a multi-step return with no off-policy correction, which in theory could adversely affect the value estimation.“ . Dies belegt zwar nicht, dass auch für den Rainbow-DQN Algorithmus keine off-policy Korrektur verwendet wurde, legt dies aber nahe.

Soweit zur Theorie. Allerdings funktioniert die Variante aus RB! laut deren Experimenten (besonders dem „ohne multistep“ Experiment) sehr gut. Dies lässt nun die logischen Schlüsse zu: Entweder treten Situationen wie die oben beschriebene, in welchen ein solcher Fehler einen Einfluss haben könnte, in den getesteten Umgebungen praktisch nie auf. Oder der Einfluss ist relativ schwach und damit zu vernachlässigen.

$$Q(s_t, a_t) \leftarrow R_{t+1} + \gamma R_{t+2} + \gamma^2 \max_{a'} Q(s_{t+2}, a') \tag{3.22}$$

$$Q(s_t, a_t) \leftarrow R_{t+1} + \gamma \max \begin{cases} R_{t+2} + \gamma^2 \max\limits_{a'} Q(s_{t+2}, a') \\ \max\limits_{a'} Q(s_{t+2}, a' | a' \neq a_{t+1}) \end{cases} \tag{3.23}$$

Dennoch gibt es eine Korrektor zu diesem Algorithmus, die ich vorschlagen und gerne testen würde. Ähnlich zum n-step Tree-Backup-Algorithmus bzw. off-policy n-step Sarsa (nachzulesen in [SB18]) ist eine Möglichkeit, das 1-Step

Q-Update mit dem 2-Step Q-Update zu vergleichen mit einem Max-Operator (entsprechend tiefere Verschachtelung für den n-Step Fall) - Gleichung 3.22 stellt ein 2-Step Q-Update dar, Gleichung 3.23 die vorgeschlagene Änderung.

4 Benchmarks und Ergebnisse

Dieses Kapitel fasst alle durchgeführten Versuche, die verwendeten Parameter und Methoden zusammen und vergleicht sie gegen existierende Baseline-Messungen. Zu erwähnen sei, dass aufgrund begrenzter Hardware und Zeit ein Kompromiss zwischen Anzahl der Testläufe, Anzahl der verschiedenen Experimente und Länge der einzelnen Versuche gemacht werden musste. Ein Testlauf über drei Millionen Schritte (24 Millionen Game-Steps (MGS)) dauert auf zur Verfügung stehender Hardware zwischen sechs und elf Stunden (das Einschalten mancher Rainbow-Erweiterungen beeinflusst dies stark). Die verwendete Hardware ist zu finden in Anhang A.

4.1 Methodik und Metriken

Alle Testläufe wurden über drei Millionen Schritte (24 MGS) evaluiert. Betrachtet wird der über 200 Episoden gemittelte Reward, den ein Agent erbringen konnte, sowie die sich auf diesen Zeitraum beziehende Standardabweichung als Maß für die Zuverlässigkeit der erlernten Policy. Zudem betrachtet wird der maximal erreichte Wert in allen Episoden. Diese Methode eignet sich gut, den Verlauf des Lernprozesses zeitlich darzustellen. Als Metrik für Vergleiche verwende ich davon den besten der so berechneten Mittelwerte mit der ihm zugeordneten Standardabweichung (im folgenden Online-Messung genannt). Da diese Werte alle während des Lernens ausgelesen werden, und sich dabei von Episode zu Episode die Policy mehrmals verändert, wurden zudem Tests mit deaktiviertem Lernen mit den finalen Netzwerkgewichten über 100 Episoden

durchgeführt, und ein mittlerer Reward sowie die Standardabweichung über diese errechnet (im folgenden Offline-Messung genannt).

Zum Vergleich: DeepMind gibt im SC2LE Paper [VEB+17] an, 600 MGS pro Testlauf zu evaluieren, und benutzt eine Best-Mean genannte Metrik, welches die durchschnittliche Performance des besten Agents von mehreren mit verschiedenen Hyperparametern ist. Über wie viele Episoden dieser Mittelwert berechnet ist, ist nicht genau angegeben.

4.2 Hyperparameter

Die folgenden Hyperparameter wurden für alle Testläufe unverändert benutzt. Die Auflösung der Feature-Layers wurde auf 32*32 festgelegt. Der Agent sieht jeden achten Frame des Environments und kann zu diesen Zeitpunkten auch interagieren (entspricht 180 APM). Der Seed wurde für jeden Test zufällig neu generiert. Es standen stets drei Aktionen zur Auswahl: „no-op“, „move_screen“ und „select_point“. Alle Aktionen werden dabei sofort ausgeführt und unterbrechen die vorher selektierte Aktion. „select_point“ war außerdem auf den Modus „toggle“ gestellt, so dass mit wiederholtem Ausführen man eine Einheit selektieren oder deselektieren kann. Falls Multi-Step Learning aktiviert war, wurde eine Schrittweite von drei verwendet, ansonsten eine Schrittweite von 1. Das verwendete Gamma als Discountfaktor zukünftiger Rewards ist bei 0.99 festgelegt. Das Replay-Memory hatte eine Größe von 200.000 Einträgen. Die Learning-Rate des Adam-Optimizers ist auf 0.0001 festgelegt. Es werden stets die ersten 4000 Schritte ohne Lernen durchgeführt, um das ReplayMemory zu füllen. Jeden vierten Schritt wird ein Lernschritt auf einem 32 Erfahrungen großen Mini-Batch ausgeführt. Für das Prioritized Replay Memory wurde der zugehörige Parameter Alpha auf 0.6 festgelegt, der Parameter Beta wächst von 0.5 zu 1.0 innerhalb 200.000 Schritte. Die Variante „Schlechtes“ Prioritized Experience Replay (aus in Kapitel 3.11.3 beschriebenen Gründen) ist eingeschaltet über den Parameter bad_prio_replay = $True$. Für die Epsilon-Greedy Policy wird von einem Startwert von 1.0 Epsilon innerhalb von 100.000

Schritten auf 0.01 verringert. Sollte NoisyNets aktiviert sein, wird Epsilon von 1.0 auf 0.0 innerhalb von nur 4000 Schritten verringert. Die zufallsregulierenden Gewichtsmatrizen und Vektoren der Noisy-Layers werden vollständig mit dem Wert 0.017 vorinitialisiert. Für Testing (ohne Lernen, wie in Abschnitt 4.1 beschrieben) wird stets das kleinste Epsilon (Zielwert) benutzt.

4.3 Ergebnisse und Vergleiche

Das folgende Kapitel präsentiert nun alle Versuchsergebnisse. Zum Vergleich wird unter anderem Abbildung 4.1 verwendet. Zeit wird in verschiedenen Darstellungen in verschiedenen Einheiten gemessen. Eine Episode entspricht 238 observierten Schritten oder 1920 Game-Steps, da jeder achte Schritt vom Agenten observiert wird. Die Einheit „Millionen Game-Steps“ wird im folgenden oft mit „MGS“ abgekürzt.

Agent	Metric	MoveToBeacon	CollectMineralShards	FindAndDefeatZerglings	DefeatRoaches	DefeatZerglingsAndBanelings	CollectMineralsAndGas	BuildMarines
Random policy	mean	1	17	4	1	23	12	< 1
	max	6	35	19	46	118	750	5
Random search	mean	25	32	21	51	55	2318	8
	max	29	57	33	241	159	3940	46
DeepMind human player	mean	26	133	46	41	729	6880	138
	max	28	142	49	81	757	6952	142
StarCraft GrandMaster	mean	28	177	61	215	727	7566	133
	max	28	179	61	363	848	7566	133
Atari-net	best mean	25	96	49	101	81	3356	< 1
	max	33	131	59	351	352	3505	20
FullyConv	best mean	26	103	45	100	62	3978	3
	max	45	134	56	355	251	4130	42
FullyConv LSTM	best mean	26	104	44	98	96	3351	6
	max	35	137	57	373	444	3995	62

Abbildung 4.1: Von DeepMind bereitgestellte Tabelle mit verschiedenen Baseline-Messungen und Evaluationen ihrer eigenen Agents. MEAN steht dabei für die durchschnittliche Performance eines Agents, BEST MEAN für die durchschnittliche Performance des besten Testlaufs eines Agents (ggf. mit anderen Hyperparametern) und MAX dem beobachteten maximalen Scores einer Episode, der jemals erreicht wurde. Alle Agents wurden auf 600 Millionen Game-Steps (MGS) evaluiert. [VEB+17]

4.3.1 MoveToBeacon

Für Details zu diesem Mini-Game siehe Abschnitt 2.3.1. Als Performance-Grundlage wurden zunächst fünf Messungen des DQN-Agents ohne eingeschaltete Erweiterungen vorgenommen, deren Lernkurven in Abbildung 4.2 zu sehen sind. In der Darstellung ist die durchgezogene Linie der über 200 Episoden (100 vor und 100 hinter der Stelle) gemittelte Reward, die jeweilige Standardabweichung ist als halbtransparente Fläche gleicher Farbe eingezeichnet. Nachdem

in allen Testläufen der Agent relativ schnell innerhalb von 2000 Schritten sein höchstes Niveau erreicht, kann man ab da eine leichte Regression erkennen, was auf eine Unstabilität des Lernziels schließen lässt. Dies lässt sich auch durch den Vergleich der Offline-Messungen mit den Online-Messungen zeigen (siehe Tabelle 4.1), da erstere zwar ähnliche Maximalwerte erreichen, aber doppelt so stark Streuen (doppelte Standardabweichung) und einen deutlich schlechteren Durchschnittswert erreichen. Ausnahme dabei ist Testlauf 4, welcher herausragende Performance lieferte, seine Online-Performance sogar übertraf und eine bemerkenswert kleine Streuung aufwies.

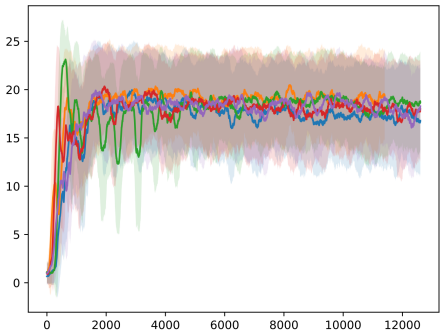

Abbildung 4.2: MoveToBeacon: DQN ohne Erweiterungen. X-Achse: Episodennummer. Eine Episode entspricht 1920 MGS.

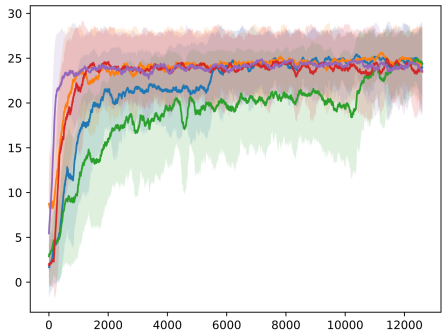

Abbildung 4.3: MoveToBeacon: DQN mit allen Erweiterungen. X-Achse: Episodennummer (je 1920 MGS).

Dem gegenübergestellt ist die Performance des FullyConv V10 Agents mit allen Erweiterungen, von welchem eine deutliche Verbesserung erwartet wird. Diese Erwartung wird auch erfüllt, wie man in Abbildung 4.3 sehen kann: Alle Tests konvergieren zu einem höheren Durchschnittswert, welcher mit 24,9 bis 25,6 auch sehr nah an die Ergebnisse von DeepMind (26, siehe Abbildung 4.1) herankommt, sowie einer deutlich kleineren Standardabweichung. Außerdem zeigt die Offline-Evaluation (siehe Tabelle 4.2) keinen starken Einbruch der Leistung im Gegensatz zu DQN, auch wenn die Performance leicht absinkt und die Streuung der Ergebnisse leicht zunimmt. Ein direkter Vergleich der

Tabelle 4.1:					
Online Evaluation			Offline Evaluation		
MEAN	MAX	σ	MEAN	MAX	σ
19,97	30	4,97	13,74	28	7,84
20,49	30	4,04	12,87	28	7,74
23,13	30	2,52	12,08	27	8,62
20,34	30	4,47	24,77	29	1,72
19,82	31	5,26	13,88	30	9,93

TABELLE 4.1: MoveToBeacon: DQN Baseline Messungen. Eine Zeile gehört jeweils zum selben Testlauf.

Tabelle 4.2:					
Online Evaluation			Offline Evaluation		
MEAN	MAX	σ	MEAN	MAX	σ
25,45	34	1,9	23,7	29	5,26
25,64	33	2,21	24,22	28	3,03
24,95	31	2,88	22,23	29	5,37
24,95	32	2,49	25,63	30	2,22
24,89	32	2,75	24,06	30	4,31

TABELLE 4.2: MoveToBeacon: FullyConv V10 Messungen. Eine Zeile gehört jeweils zum selben Testlauf.

Kurven ist in Abbildung 4.4 abgebildet, wobei die beiden Kurven jeweils der Durchschnitt zu jedem Zeitpunkt von den fünf (bzw. vier) Messkurven aus 4.2 und 4.3 ist. Zudem bleibt die Kurve des FullyConv V10 Agents steigend, was neben der geringeren Streuung für eine verbesserte Stabilität des Algorithmus spricht.

Die Auflösung der Lernkurven zu MoveToBeacon, die DeepMind in [VEB+17] veröffentlicht hat, ist zu gering, um den Anstieg des Lernfortschritts zeitlich abzulesen - der Graph springt sofort auf den Maximalwert. Für CollectMineralShards ist ein genauerer Vergleich möglich, siehe Kapitel 4.3.2.

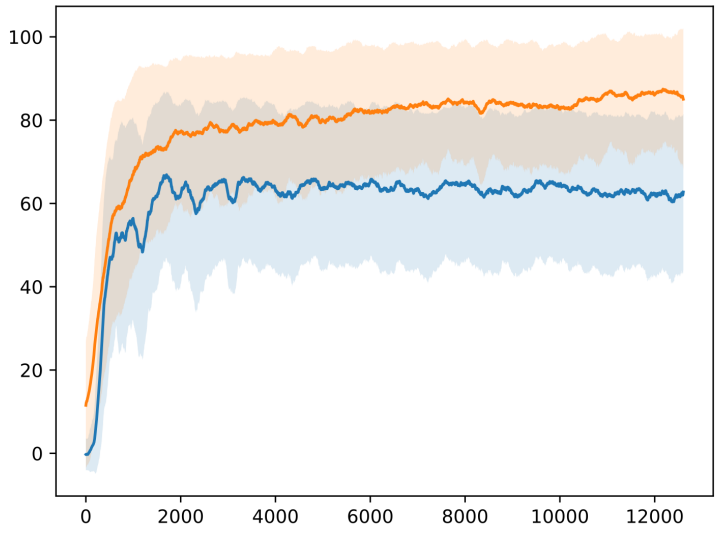

Abbildung 4.4: MoveToBeacon: DQN (blau, unten) vs. FullyConv V10 (orange, oben) normalisiert zu menschlicher Baseline (StarCraft GrandMaster) aus [VEB+17]. X-Achse: Episodennummer. Eine Episode entspricht 1920 MGS.

4.3.2 CollectMineralShards

Für dieses etwas kompliziertere Mini-Game zeichnet sich die Charakteristik der beiden Algorithmen, die für MoveToBeacon beschrieben wurden, nocheinmal

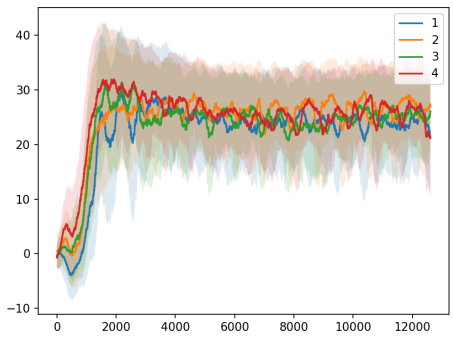

Abbildung 4.5: CollectMineralShards: DQN ohne Erweiterungen. X-Achse: Episodennummer. Eine Episode entspricht 1920 MGS.

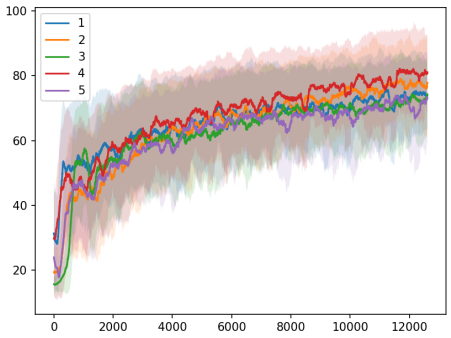

Abbildung 4.6: CollectMineralShards: DQN mit allen Erweiterungen. X-Achse: Episodennummer. Eine Episode entspricht 1920 MGS.

deutlicher ab, insbesondere der Rückgang der Performance des reinen DQN Agents in Abbildung 4.5. Die Streuung der Messwerte war hier allerdings für beide Algorithmen relativ hoch, sowohl im Online- als auch im Offline-Fall. Die Durchschnittswerte des DQN-Algorithmus brachen für die Offline-Messungen allerdings nicht ein im Gegensatz zu MoveToBeacon, siehe Tabelle 4.3 und Tabelle 4.4.

Der FullyConv V10 Agent erreicht innerhalb der begrenzten Schrittzahl von 24 beziehungsweise 80 Millionen Game-Steps (MGS) einen sehr ähnlichen Durchschnittswert wie der beste von DeepMind eingesetzte FullyConv-A3C-Agent [VEB+17] aus 100 verschiedenen Versuchen. Aus dem Graphen 4.13 abzulesen, erreicht deren bester Lauf die 80 Punkte Marke in CollectMineralShards nach 48,7 MGS (Pixelgenau aus der Vektorgrafik des zugehörigen Papers ausgelesen) und eine durchsichtige blaue Kurve bereits nach 27,8 MGS. FullyConv V10 aus Abbildung 4.12 erreicht dies nach circa 48,9 MGS. Der vierte der fünf Testläufe aus Tabelle 4.4 erreicht dieses Niveau allerdings bereits nach 19,7 MGS. Lauf Nummer Zwei erreicht die 79 Punkte in 22,5 MGS. (24 MGS sind 3 Millionen observierte Schritte, oder 12600 Episoden, was der Testlänge der Versuche in dieser Arbeit entspricht).

Tabelle 4.3:					
Online Evaluation			Offline Evaluation		
MEAN	MAX	σ	MEAN	MAX	σ
64,59	98	17,61	66,37	79	8,15
64,60	96	8,99	67,17	80	9,10
67,22	95	16,37	56,54	78	16,43
68,14	97	14,46	69,70	97	17,25

TABELLE 4.3: CollectMineralShards: DQN Baseline Messungen. Eine Zeile gehört jeweils zum selben Testlauf.

Tabelle 4.4:					
Online Evaluation			Offline Evaluation		
MEAN	MAX	σ	MEAN	MAX	σ
75,72	99	10,57	78,93	99	12,14
79,05	104	12,61	77,39	100	19,22
74,30	98	11,56	64,00	93	18,42
82,22	107	11,83	74,85	85	12,75
74,19	97	10,94	75,36	95	12,66

TABELLE 4.4: CollectMineralShards: FullyConv V10 Messungen. Eine Zeile gehört jeweils zum selben Testlauf.

Auch wenn dies Zufallstreffer sein können und diese Ergebnisse keinesfalls langfristige Stabilität über weitere 520 Millionen Game-Steps garantieren, sind diese deutlich besser (im Sinne von erreichen mit weniger Game-Steps ein gewisses Niveau) als der beste Testlauf aus 100 von DeepMind. Der beste FullyConv V10 Lauf brauchte nur 40% der Game-Steps des besten FullyConv-A3C-Agents beziehungsweise 71% der Zeit des transparenten schnellsten Agents, um einen Durchschnitt von 80 Punkten in CollectMineralShards zu erreichen. Der beste Atari-Net Agent erreicht dies erst nach circa 130 Millionen Game-Steps. Dies zeigt, dass der FullyConv V10 Agent im evaluierten Bereich deutlich Dateneffizienter arbeitet, als FullyConv-A3C oder FullyConvLSTM-A3C.

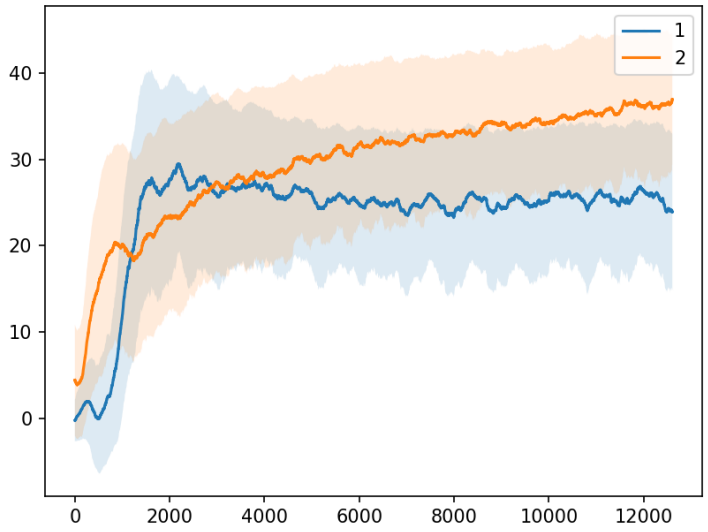

Abbildung 4.7: CollectMineralShards: DQN (blau, unten) vs. FullyConv V10 (orange, oben) normalisiert zu menschlicher Baseline (StarCraft GrandMaster) aus [VEB+17]. X-Achse: Episodennummer. Eine Episode entspricht 238 Steps.

4.3.3 Vergleiche der Erweiterungen

Nachdem die bisherigen Ergebnisse relativ deutlich zeigen, dass der Rainbow-DQN Algorithmus (ohne Distributional RL) eine deutliche Verbesserung gegenüber DQN bedeutet, stellt sich die Frage, welche der fünf Erweiterungen darauf welchen Einfluss hatte. Dem Vorbild des Rainbow-DQN Papers [HMVH+18] folgend, wurden Messungen mit nur einer eingeschalteten Erweiterung durchgeführt, um das einprägsame Bild 4.9 der ersten Seite des Rainbow DQN Papers nachzustellen. Im Gegensatz zu diesem Vorbild zeigen diese leider keine schöne Abgrenzung zwischen den meisten Komponenten an, lediglich Dueling DQN kann sich abheben. Außerdem ist die gute Performance des Rainbow-Agents gegenüber den Einzelalgorithmen gut sichtbar. Dieses Experiment ist allerdings nicht aussagekräftig und wurde nur zu Vergleichszwecken erstellt, da aus Zeitmangel jeweils nur eine einzelne Messung pro Variante ge-

Tabelle 4.5:						
	Online Evaluation			Offline Evaluation		
Name	MEAN	MAX	σ	MEAN	MAX	σ
DQN	20,75	31	4,25	15,47	30	7,17
DDQN	20,31	30	3,82	24,60	28	1,81
Dueling DQN	24,61	32	2,40	24,48	28	2,41
PER DQN	20,17	30	6,26	11,36	26	8,52
Noisy DQN	19,69	28	3,79	14,67	24	5,46
MultiStep DQN	18,73	29	4,27	17,11	27	7,54
FullyConv V10	25,18	34	2,45	23,97	30	4,04

TABELLE 4.5: MoveToBeacon: Messung der Teilkomponenten des FullyConv V10 Agents. Eine Zeile gehört jeweils zum selben Testlauf.

macht werden konnte. Außerdem konnte nur MoveToBeacon getestet werden. Die Ergebnisse sind in Tabelle 4.5 zusammengefasst. Für DQN und FullyConv V10 wurde jeweils der Durchschnitt aller fünf zur Verfügung stehenden Messungen benutzt.

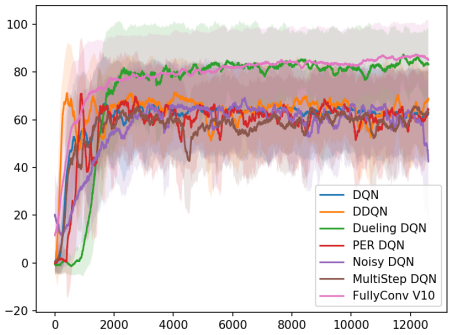

ABBILDUNG 4.8: FullyConv V10 Komponenten in MoveToBeacon. X-Achse: Episodennummer. Eine Episode entspricht 1920 MGS.

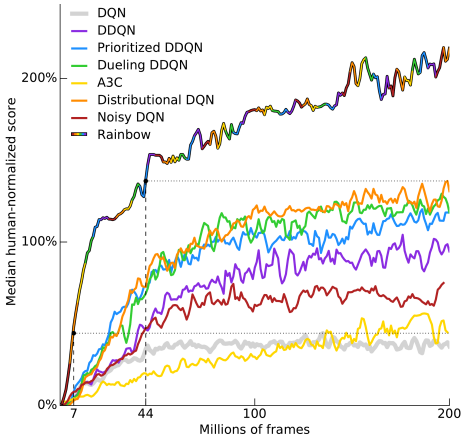

ABBILDUNG 4.9: Rainbow Komponenten [HMVH+18]: Median über 57 Atari-Spiele, normalisiert zu menschlicher Leistung.

Eine interessante Sache, die im Rainbow DQN Paper untersucht wurde, waren Benchmarks, bei welchen jeweils eine der Erweiterungen deaktiviert war. Dies dient dazu, herauszufinden, wie groß der Einfluss einer Erweiterung auf das Gesamtergebnis ist. Für dieses Experiment wurden pro Variante jeweils zwei Messungen gemacht, um eine etwas genauere Aussage treffen zu können. Die Werte in der Tabelle 4.6 sowie in dem Graphen 4.10 sind die Durchschnittswerte (beziehungsweise für den MAX Operator das Maximum) beider Läufe. Für DQN und FullyConv V10 wurden alle fünf zur Verfügung stehenden Messungen benutzt.

Die Varianten mit dem stärksten Performance-Abfall lassen nun auf den Einfluss der jeweiligen Erweiterungen schließen. Dueling Networks stellte sich dabei als relevanteste Erweiterung heraus, zusammen mit NoisyNets und PER. Die Ergebnisse aus Abbildung 4.11 stellen (neben dem nicht getesteten Distributional RL) PER, MultiStep und NoisyNets als relevanteste Erweiterungen dar. Die Unterscheide zu diesen Ergebnissen liegen bei dem Einfluss von MultiStep-Targets - dies ist in der MoveToBeacon Umgebung nicht von großer Bedeutung gewesen - sowie Dueling Networks, welche großen Einfluss hatten.

Überraschenderweise übertraf die Performance des Agents ohne Double DQN sogar die des FullyConv V10 Agents. Dies kann auch in Abbildung 4.11 beobachtet werden, wo die „no double“-Kurve regelmäßig über der des Rainbow-Agents liegt.

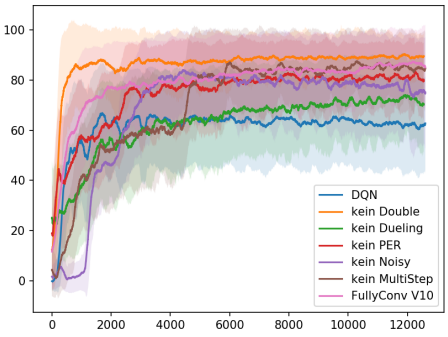

Abbildung 4.10: Deaktivieren verschiedener FullyConv V10 Komponenten in MoveToBeacon. X-Achse: Episodennummer. Eine Episode entspricht 1920 MGS.

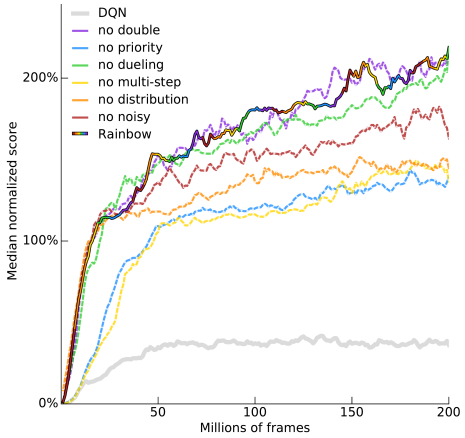

Abbildung 4.11: Deaktivieren verschiedener Rainbow Komponenten [HMVH+18]: Median über 57 Atari-Spiele, normalisiert zu menschlicher Leistung.

Diese Ergebnisse müssen allerdings äußerst kritisch betrachtet werden. Die relativ kurzen Evaluationszeiten sind zu wenig, um den Einfluss der Erweiterungen auf den späteren Lernprozess vorauszusagen (24 Millionen Game-Steps (MGS) gegen 200 Millionen Steps im Atari 2600 Benchmark und 600 MGS in StarCraft II durch DeepMind [VEB+17]). Wie der Verlauf des Graphen 4.11 zeigt, werden die Unterschiede zwischen den Varianten erst nach einer gewissen Zeit klar. Um zu untersuchen, wie sich die Algorithmen nach längerer Lernzeit verhalten, ist deutlich mehr Rechenzeit nötig. Ein Testlauf des FullyConv V10 Agents über 80 MGS, welcher über 32 Stunden dauerte, ist in Abbildung 4.12 zu sehen. Dieser wurde zwar im Gegensatz zu den in Abbildung 4.10 gezeigten Versuchen auf dem anderen Mini-Game (CollectMineralShards) ausgeführt, dennoch zeigt er, dass nach 24 MGS der Lernprozesses noch lange keine Grenze erreicht hat. Zudem ist die geringe Anzahl an Versuchen ein Problem. Bei den von Lauf zu Lauf schwankenden Resultaten kann aus nur zwei Testläufen keine signifikante Aussage abgeleitet werden, sondern lediglich eine vage Tendenz. Die durchsichtig gezeichneten Kurven der Abbildung 4.13 zeigen die

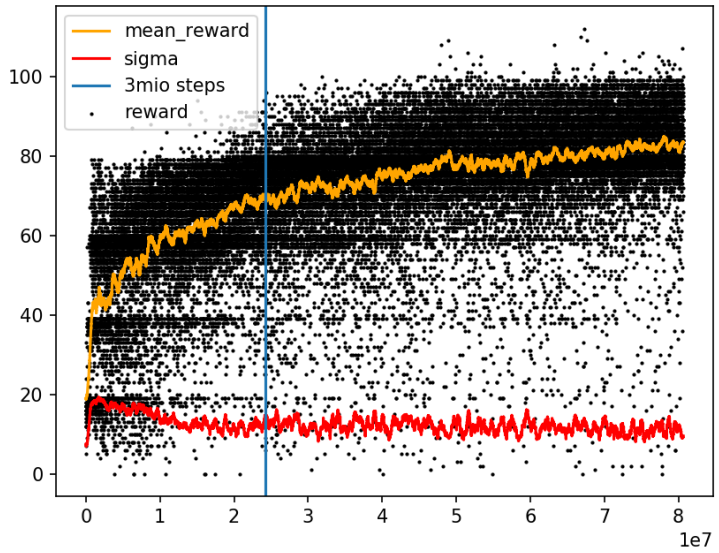

Abbildung 4.12: CollectMineralShards: FullyConv V10 über zehn Millionen Schritte evaluiert. X-Achse: Game-Steps. Ein Schritt entspricht 8 Game-Steps.

gesamten 100 Testläufe von DeepMind. Diese streuen relativ breit gefächert, und verdeutlichen einmal mehr die Notwendigkeit einer größeren Anzahl an Tests.

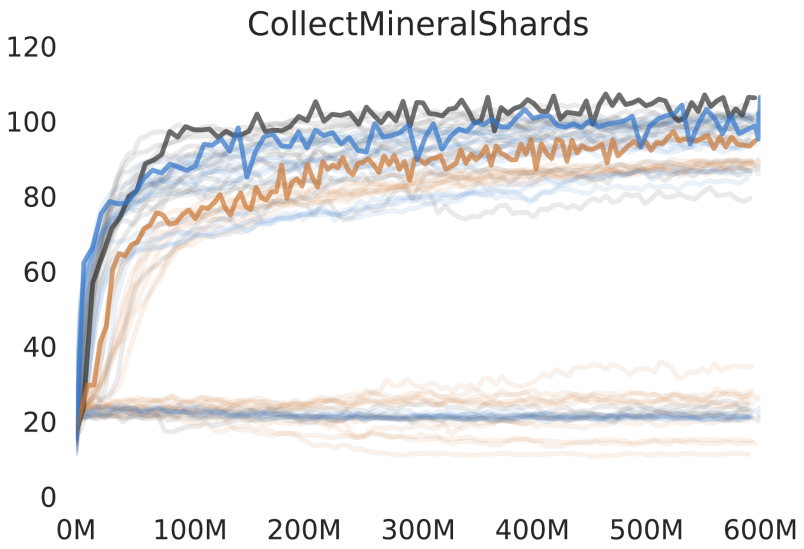

Abbildung 4.13: CollectMineralShards: Blau: FullyConv-A3C-Agent, Grau: FullyConvLSTM-A3C-Agent, Orange: Atari-Net-Agent; aus [VEB+17]. X-Achse: Game-Steps. Durchsichtig gezeichnet sind alle 100 Testläufe.

Tabelle 4.5:

	Online Evaluation			Offline Evaluation		
Name	MEAN	MAX	σ	MEAN	MAX	σ
DQN	20,75	31	4,25	15,47	30	7,17
kein Double	25,70	33	2,24	25,42	32	1,97
kein Dueling	21,29	29	3,86	19,67	27	6,10
kein PER	23,80	33	2,93	23,68	30	2,89
kein Noisy	23,93	33	3,84	20,74	29	6,87
kein MultiStep	24,76	33	2,86	24,56	31	3,72
FullyConv V10	25,18	34	2,45	23,97	30	4,04

Tabelle 4.6: MoveToBeacon: FullyConv V10 Messungen. Eine Zeile gehört jeweils zum selben Testlauf.

5 Zusammenfassung

Der Rainbow-DQN Algorithmus in kombination mit der FullyConv Architektur reproduziert seine verbesserte Leistung gegenüber DQN in der StarCraft II Mini-Games Domäne, jedoch mit geringerem Abstand. Grund dafür ist höchstwahrscheinlich allerdings die zu geringe Länge der Experimente, dennoch zeichnen sich klar die Verbesserungen durch verschiedene Erweiterungen wie Dueling DQN ab. So wie im Rainbow-DQN Paper [HMVH$^+$18] gezeigt wird, ist der Einfluss der verschiedenen Erweiterungen auf verschiedene Problemdomänen (im Paper verschiedene Atari 2600 Spiele) teilweise sehr unterschiedlich, so dass es nicht überrascht, dass in dieser Arbeit für StarCraft II eine andere Verteilung festgestellt werden konnte. Insbesondere die höhere Dateneffizienz gegenüber dem FullyConv-A3C Algorithmus von DeepMind [VEB$^+$17] stellt eine vielversprechende Überraschung dar, und diverse Verbesserungen aus Kapitel 3.11 versprechen weitere Steigerung der Performance des Agents.

Letztendlich drehen sich viele Argumente darum, dass die Rechenzeit für die breite Palette an interessanten Experimenten zu gering war, und die dadurch auferlegten Einschränkungen einige Fragen offenlassen. Reinforcement Learning ist von einem gewissen Zufallsfaktor von Experiment zu Experiment abhängig, was insbesondere eigentlich eine höhere Anzahl an Versuchswiederholungen erfordert. Dennoch stimmen die gesammelten Ergebnisse mit allen Erwartungen überein, was diese zu bestätigen scheint.

Literaturverzeichnis

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[ARAA+16] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre-Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Mélanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu

Germain, Xavier Glorot, Ian Goodfellow, Matt Graham, Caglar Gulcehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrancois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert T. McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[BDM17] Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 449–458. JMLR. org, 2017.

[BNVB13] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, jun 2013.

[C+15] François Chollet et al. Keras. `https://keras.io`, 2015.

[Dee17] DeepMind. Pysc2. `https://github.com/deepmind/pysc2`, 2017.

[DHK+17] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

[Ent17] Blizzard Entertainment. Starcraft ii client. `https://github.com/Blizzard/s2client-proto`, 2017.

[FAP+17] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, et al. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

[HMVH+18] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[HQB+18] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[Mö19] Benjamin Möckl. Rainbow-dqn for keras-rl in sc2. `https://github.com/chucnorrisful/dqn`, 2019.

[Pla16] Matthias Plappert. keras-rl. `https://github.com/keras-rl/keras-rl`, 2016.

[SA16] Frank Seide and Amit Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. pages 2135–2135, 08 2016.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[SHS+17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[SQAS15] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[VEB+17] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

[VHGS16] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[Wat89] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

[WSH+15] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for

deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

Abbildungsverzeichnis

Tabellenverzeichnis

A Verwendete Hardware

Die meisten Testläufe wurden auf PC 1 ausgeführt. Um mehr Messungen in der gegebenen Zeit zu schaffen, wurden die letzten 6 Tage auch auf PC 2 Versuche durchgeführt - siehe Tabelle A. Als Betriebssystem (OS) war Ubuntu notwendig, da die Windows-Version von StarCraft II es nicht erlaubt, das Spiel „headless“, also ohne die 3D-Spieloberfläche, auszuführen. Der Wechsel zu Ubuntu führte damit zu einer circa zweifachen Beschleunigung des Lernprozesses. Desweiteren wird in beiden Systemen mittels der Pakete „tensorflow-gpu“ und „keras-gpu“ die jeweilige Grafikkarte benutzt, was zu einer fünffachen bis zehnfachen Beschleunigung gegenüber dem Berechnen mittels der CPU geführt hat.

Tabelle A: Verwendete Hardware.

PC 1	
CPU	i7 7700K @ 4,8 GHz
RAM	16GB + 8GB Swap
GPU	NVidia Geforce GTX 1080Ti
OS	Ubuntu 18.10
DRIVE	SSD

PC 2	
CPU	i7 7700K @ 4,5 GHz
RAM	16GB + 16GB Swap
GPU	NVidia Geforce GTX 1070
OS	Ubuntu 18.04
DRIVE	HDD

B Stand der Implementierung

Der Quellcode zu dieser Arbeit kann neben der abgegebenen Version auch auf GitHub gefunden werden: [Mö19]. Er enthält neben einer Installationsanleitung in der Datei „README.md“ den vollständig funktionierenden „FullyConv V10“ Agent, sowie alle vorhergehenden Versionen. Dazu gehören eigene Implementierungen einer „NoisyDense“-Layer und einer „NoisyConv2D“-Layer. Ein auf das Keras-rl Framework zugeschnittener Environment-Wrapper für StarCraft II unter Verwendung der pysc2 Bibliothek zusammen mit einer zugehörigen Policy-Klasse ist außerdem vorhanden. Daneben zu finden ist eine leicht modifizierte Version der Prioritized-Replay-Buffer Implementierung von OpenAI/baselines [DHK+17]. Die Hilfsdatei „plot.py“ bietet Hilfsmethoden für Visualisierung, „customCallbacks.py“ enthält einen GPU-Logger in form eines Keras-Callbacks, welcher den Status der Grafikkarte mitschreiben kann. Ausgangspunkt des Codes ist die Datei „exec.py“, welche auch die Agenten enthält. Teile der Implementierung des Agentens befinden sich auch in „sc2DqnAgent.py“ und „agent2.py“. Der Code ist ausführlich auf Deutsch kommentiert. Die Kommentare sind am einfachsten zu verstehen, wenn man in „exec.py“ zu lesen beginnt, sowie Kapitel 3.4, welches die zugrundeliegende Architektur beschreibt, dabei im Hinterkopf hat. Um Missverständnisse zu vermeiden sei gesagt, dass der sporadisch auftauchende Name „fake_rainbow“ die alte Bezeichung für den „FullyConv V10“ Agent war, welcher im Vergleich zu dem richtigen Rainbow-DQN Algorithmus kein Distributional RL implementiert.

Es existiert zudem eine experimentelle Version eines Agents, welcher diese Erweiterung doch implementiert, unter dem Namen „FullyConv V11“. Dieser lernt jedoch aktuell nicht, auch wenn er keine Fehler wirft.

Im Ordner /finalWeights finden sich außerdem zwei Dateien, die trainierte Gewichte für jeweils einen auf CollectMineralShards und einen auf MoveToBeacon trainierten Agent enthalten. Aufgrund der Dateigröße sind nicht mehr Gewichte beigelegt.

Eidesstattliche Erklärung

Ich versichere, dass die Bachelorarbeit mit dem Titel

„Implementing Rainbow DQN in
StarCraft II using Keras-rl“

nicht anderweitig als Prüfungsleistung verwendet wurde und diese Bachelorarbeit noch nicht veröffentlicht worden ist. Die hier vorgelegte Bachelorarbeit habe ich selbstständig und ohne fremde Hilfe abgefasst. Ich habe keine anderen Quellen und Hilfsmittel als die angegebenen benutzt. Diesen Werken wörtlich oder sinngemäß entnommene Stellen habe ich als solche gekennzeichnet.

Augsburg, den 11. März 2019 Unterschrift