University of Applied Sciences Berlin

Report

Reinforcement Learning in einem 2D-Environment (Super Mario Bro.)

by

Dennis Lehmann
s0568827

Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Supervisor: **Patrick Baumann, M.Sc.**

03. April, 2021

Abstract

Reinforcement Learning in einem 2D-Environment (Super Mario Bro.)

Das Projekt beschäftigt sich mit dem *Reinforcement Learning*, genauer gesagt mit dem *Double Q-Learing* Algorithmus. Die primäre Frage, die in diesen Versuchen untersucht werden soll ist, ob, ein artifizielles neuronales Netzwerk, dass diesen Algorithmus implementiert, dazu in der Lage ist, das 2D-Environment *Super Mario Bro.* zu erlernen. Darauf aufbauend soll geklärt werden, wie gut ein solches Modell performt. Die Untersuchung ergab, dass das Modell grundsätzlich dafür geeignet scheint, das Environment zu erlernen. Jedoch wurde das sekundäre Ziel - das erste Level abzuschließen - aufgrund unzureichender Ressourcen nicht erreicht.

Contents

Chapter 1

Introduction

1.1 Motivation

Ziel des Projekts ist die Erweiterung des Wissensstands im Bereich des Maschinelles Lernens. Es gilt im Besonderen die bereits erworbenen Kenntnisse über *Convolutional Neural Networks* (CNN) zu vertiefen und mit den Grundlagen des *Reinforcement Learnings* vertraut zu werden. Dieses Wissensgebiet erfährt durch seine manigfaltigen Anwendungmöglichkeiten zunehmend Bedeutung, so beispielsweise in Bereichen wie dem autonomen Fahren oder im der Spieleindustrie (siehe AlphaGo)[**âĂć**].

1.2 Ziel des Projekts

In diesem Projekt wird untersucht, ob der *Double Q-Learning Algorithmus*, der in den Bereich des Reinforcement Learnings fällt, das Spiel-Environment zu erlernen und eigenständig das erste Level dieses Spiels erfolgreich zu absolvieren. Weiterhin soll durch den praktischen Aspekt dieses Projekts Wissen über die einzelnen Techniken erlangt werden.

1.3 Vorgehensweise und Aufbau der Arbeit

Die Plattform *gym.openai.com* stellt vielerlei Lernumgebungen für das maschinelle Lernen bereit, darunter auch zahlreiche 2D-Spiele *Application Programming Interfaces* (API). In Kombination mit dem *Nes-py Emulator* stellen diese beiden Elemente die Grundlage für dieses Projekt dar. Durch das PyTorch-Tutorial zum Thema Reinforcement Learning inspiriert, fiel die Wahl des Environments auf *Super Mario Bro.*. Die Inputdaten für das Reinforcement Learning durchlaufen während des Trainings mehrere

Wrapper und ein internes CNN. Die Arbeit ist wie folgt gegliedert: Zunächst wird das Thema Reinforcement Learning im *Grundlagen* Kapitel theoretisch erörtert, um die Wahl des Algorithmus' nachvollziehbar zu machen. Der Aufbau des Modells und der verwendete Algorithmus werden im nachfolgenden Kapitel *Konzeption* beschrieben. Im nächsten Schritt werden auf die *Implementierung* betreffende Feinheiten im gleichnamigen Kapitel näher eingegangen. Die Ergebnisse der Experimente sind in Kapitel *Auswertung* zu finden und werden abschließend im *Fazit* zusammengefasst.

Chapter 2

Grundlagen

Um einen Algorithmus zu entwickeln, der das Ziel hat, durch Lernen ein Spiel spielen zu können, gibt es zunächst drei grundlegende Techniken, die in Betracht kommen könnten. Supervised Learning, Unsupervised Learning und Reinforcement Learning. Die Techniken sollen hier übersichtshalber vorgestellt werden, um im späteren Verlauf die Wahl der verwandten Technik nachvollziehbar zu machen. Am besten kann man die verschiedenen Techniken anhand der Daten unterscheiden, die man braucht, um sein Netz zu trainieren und die Probleme, die man mit solchen Ansätzen lösen kann.

Das Supervised Learning braucht für seinen Ansatz einen Datensatz, bestehend aus den Rohdaten und einer Zuschreibung (Label). Dieses Label sagt, um was für einen Typ es sich bei dem Datenpunkt handelt. Mathematisch ausgedrückt in der Form (X, y). Das Ziel ist es, beim Supervised Learning eine Funktion zu finden, welche die Daten auf das zu ihnen zugeordnete Label abbildet. Die Technik wird meist zur Lösung von Klassifikationsproblemen gewählt, zu beachten hierbei ist jedoch, dass genügend Daten zur Verfügung stehen müssen, um ein maschinelles Lernen zu ermöglichen.[Zha20](S. 22 ff.)

Beim Unsupervised Learning werden als Datensatz nur die Datenpunkte benötigt. Das Ziel ist es, dann die zugrunde liegende Struktur der Daten zu erlernen.[Zha20](S. 30 ff.)

Reinforcement Learningunterscheidet sich von beiden vorherigen Techniken. Es steht kein Datensatz zur Verfügung, nur eine Umgebung, die Werte wie State, Actions und Reward Paare hat, auf denen das Netz trainieren soll. Diese werden während des Lernens durch erkunden der Umgebung generiert und nicht wie bei den beiden anderen Techniken vorgegeben. Das Ziel ist es, für jede Situationen genau die Aktion zu finden,

welche den Reward maximiert.[Zha20](S. 32 ff.)

In diesem Projekt wird das Reinforcement Learning eingesetzt, die Wahl der Umgebung fiel auf ein Level aus einem 2D-Spiel. siehe unten 2.3. Das Ziel ist es, das Level erfolgreich abzuschließen, dafür müssen die richtigen Aktionen vorhersagen und ausgeführt werden. Da keine gelabelten Daten existieren, ist die Techniken des Reinforcement Learnings die richtige Wahl. In diesem und folgenden Kapiteln wird auf Techniken rundum das Reinforcement Learnings eingegangen.

2.1 Reinforcement Learning

In diesem Abschnitt soll es um die mathematischen Konstrukte hinter der Idee des Reinforcement Learnings gehen. Der Kerngedanke ist, der Umgebung all jene Informationen zu entnehmen, die es braucht um zu entscheiden, was die nächstbeste Aktion ist. Diese Idee wurde das erste mal vom Mathematiker Richard Bellman publiziert, der nannte es "Markov decision process". Dieser "Markov decision process"[BEL57] liegt allen Reinforcement Learning Algorithmen zugrunde. Bevor hier auf den "Markov decision process" eingegangen wird, müssen noch ein paar Begriffe erklärt werden.

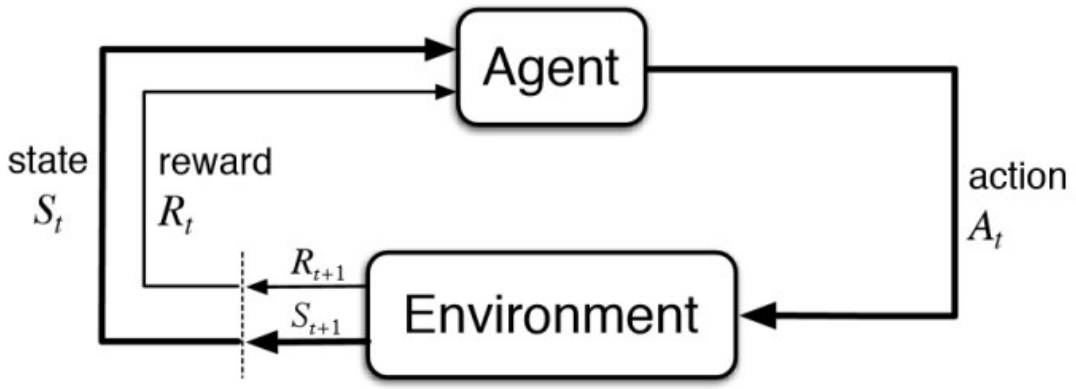

Figure 2.1: Exemplary Reinforcement Learning sequence [Bha18]

In der obigen Abbildung 2.1 ist das Objekt Environment zu sehen. Das Environment ist die Umgebung, in der sich alles abspielt. Zum Beispiel könnte auch die reale Welt als Environment betrachtet werden. Ein Level aus dem Spiel "Super Mario" stellt das Environment dieses Projektes dar. Das Environment gibt vor, welche Regeln herrschen. Es legt auch die Handlungsmöglichkeiten und Erscheinungsbilder der Objekte fest, welche sich in ihr befinden. Um Objekte in der Umgebung zu bewegen,

bekommt das Environmen sogenannte Actions. Action sind Befehle, die in der Umgebung bestimmt Aktionen ausführen, zum Beispiel springen oder nach vorn zu laufen. Welche Actions akzeptiert werden, hängt vom Environment ab. Nach Ausführung dieser Actions befindet sich das Environment in einem neuen Zustand, welchen es der betrachteden Person zurückgibt. Meist in Form bildlicher Darstellung. Außerdem wird ein Reward zurückgegeben. Dieser Reward bewertet die ausgeführte Action, schätzt ein ob sie gut oder schlecht war. Diese wird zum Lernen benötigt und weiter unten genauer erklärt. Wie die Rewards aussehen können, wird vom Environment bestimmt. Ein gutes Beispiel für einen Reward wäre das Ergebnis, wenn man seine Hand auf oder neben die heiße Herdplatte legen würde - eine Verbrennung stellt einen negativen Reward dar, wohingegen ein positiver Reward durch eine unverletzt gebliebene Hand symbolisiert werden könnte.[Gér19](S. 610 ff.)

Ein weiteres Objekt, was oben in der Abbildung dargestellt ist, ist der Agent. Im Reinforcement Learning ist der Agent das Gehirn des Algorithmus' und kann mit dem Gehirn des Menschen verglichen werden. Als Information bekommt er vom Environment die States und Reward Paare, mit diesen Informationen und dem Gelernten entscheidet er, welche Action als nächstes ausgeführt werden soll.[Gér19](S. 610 ff.)

Nach dem nun die wichtigsten Begriffe erklärt wurden, zurück zum Markov decision process. Der Markov decision process kennt alle States- ($S = \{s_1...s_n\}$)und Aktionen ($A = \{a_1...a_n\}$) und auch die Übergangsfunktionen (transition function)$T : S \times A \times S \rightarrow [0, 1]$ und die Belohnungsfunktionen $R : S \times A \times S \rightarrow \mathbb{R}$. Mit diesen Werten lässt sich die folgende Formel aufstellen, mit deren Hilfe die Entscheidung getroffen wird, welche Aktion am sinnvollsten ist.

$$V*(s) = \max_{a \in A} \sum_{s\prime \in S} T(s, a, s\prime)[R(s, a, s\prime) + \gamma * V*(s\prime)] \tag{2.1}$$

Das Gamma ist der Discount-Faktor, der dafür da ist, zu entscheiden, ob kurze Wege bis zum Ziel besser bewertet werden oder nicht. [BEL57] Es gibt zwei verschiedene Arten es einzusetzen. Einmal als Value Iteration [Cho18] oder als Policy Extraction [BPS18]. Da man beim Reinforcement Learning vor dem Problem steht, nicht alle States zu kennen, dies aber Voraussetzung zur Anwendung der Formel ist, muss diese Formel für das Reinforcement Learning angepasst werden. Dafür wurden die Q-Values entwickelt, ihr Konzept wird nachfolgend besprochen 3.1.

2.2 Convolutional Neural Networks

Um Bilder maschinell verarbeiten zu können, werden bevorzugt Convolutional Neural Networks (CNN) benutzt, die Idee wird im folgenden erörtert. Ein **CNN** ist eine spezielle Art von mehrschichtigen neuronalen Netzen, welche visuelle Muster direkt aus Pixelbildern mit minimaler Vorverarbeitung erkennen.[ZF13] CNNs sind von der Natur inspirierte Modelle, welche in den Forschungen von D. H. Hubel und T. N. Wiesel [HW62] beschrieben wurden. Sie schilderte wie Menschen und Tiere ihre Umgebung in Bildern wahrnehmen - mit einer Schichtenarchitektur von Neuronen im Gehirn. Diese Beschrebung nahmen sich Ingenieure und Wissenschaftler als Vorbild um daraus die Idee der Bilderkennung zu konstruieren, welche heute alsCNN bekannt sind.

Folgende Aspekte werden bei CNNs berücksichtigt: Die **Übersetzungsinvarianz** in Bildern impliziert, dass alle Patches eines Bildes auf dieselbe Weise behandelt werden. **Lokalität** bedeutet, dass nur eine kleine Nachbarschaft von Pixeln verwendet wird, um die entsprechenden Darstellungen zu berechnen. Bei der Bildverarbeitung erfordern Faltungsschichten typischerweise **viel weniger Parameter** als vollständig verbundene Schichten. CNNs sind eine spezielle Familie neuronaler Netze, die **Faltungs-Schichten** enthalten. Mit den Ein- und Ausgabekanälen kann ein Modell an **jedem räumlichen Ort** mehrere Aspekte eines Bildes erfassen. ([Zha20] Ch. 6)

Ein CNN lässt sich somit als eine Folge von Schichten beschreiben, wobei jede Schicht mit der Umwandlung der Informationen aus der vorherigen Schicht beauftragt ist. Dabei werden die komplexen Informationen in jeder Schicht weiter abstrahiert und die zu verarbeitende Informationsmenge verringert, bevor sie an die nachfolgend geschaltete Schicht weitergegeben wird. Auf der unteren Abbildung 2.2 ist eine allgemeine Darstellung eines CNNs zu sehen.

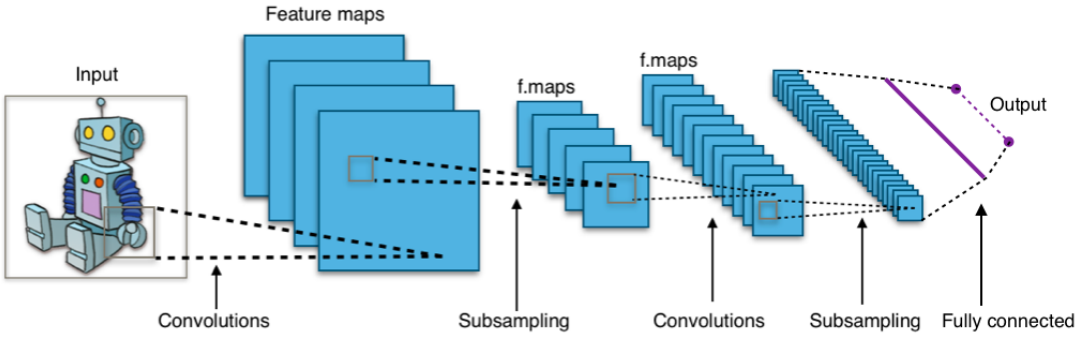

Figure 2.2: Exemplary CNN architecture [Com15]

CNNs lassen sich in zwei grundlegende Blöcke unterteilen: *Convolutional Block* und *Fully Connected Block*. Der Convolutional Block besteht aus dem *Convolutional Layer* und weiteren Layern, beispielsweise einem *Pooling Layer*. Dies Verbindungen mehrerer solcher Layern zu Blöcken ist für den wesentlichen Bestandteil der Informationsextraktion verantwortlich. Der *Fully Connected Block* ist der zweite Bestandteil und besteht aus einer vollständig verbundenen, einfachen, neuronalen Netzwerkarchitektur (fully connected, FC). Basierend auf dem Output des Convolutional Blocks, ist es Aufgabe dieses Teils des Modells die übergebenen Daten zu klassifizieren.

Indem ein Filter in mehreren Schritten auf das gesamte Bild angewendet wird, extrahieren die *Convolutional Layer Feature-Maps* aus dem Input-Bild. Die Feature Maps repräsentieren Wahrscheinlichkeitsverteilungen, die angeben, wie wahrscheinlich es ist, in der untersuchten Region das Feature anzutreffen, welches durch den Filter definiert ist. Aus den Feature-Maps berechnet der fully connected Block die Klassenzugehörigkeitswahrscheinlichkeiten. Filter bzw. *Kernel* sind auch eine Abbildung, welche eine bestimmte Funktion darstellt. In folgender Abbildung 2.3 wird beispielsweise eine Kurve dargestellt. Diese dient als Beispielmerkmal, welches erkannt werden soll - es soll mit Hilfe von diesen Filtern bestimmt werden, ob so eine Kurve in einem Bild vorhanden ist. Die *Pooling* layer verkleinert die Informationsmenge, welche die *Convolutional Layer* für jedes Merkmal generiert, und behält die wichtigsten Informationen bei (der Prozess *Convolutional-* und *Pooling Layer* wird normalerweise mehrmals wiederholt). ([Mis])

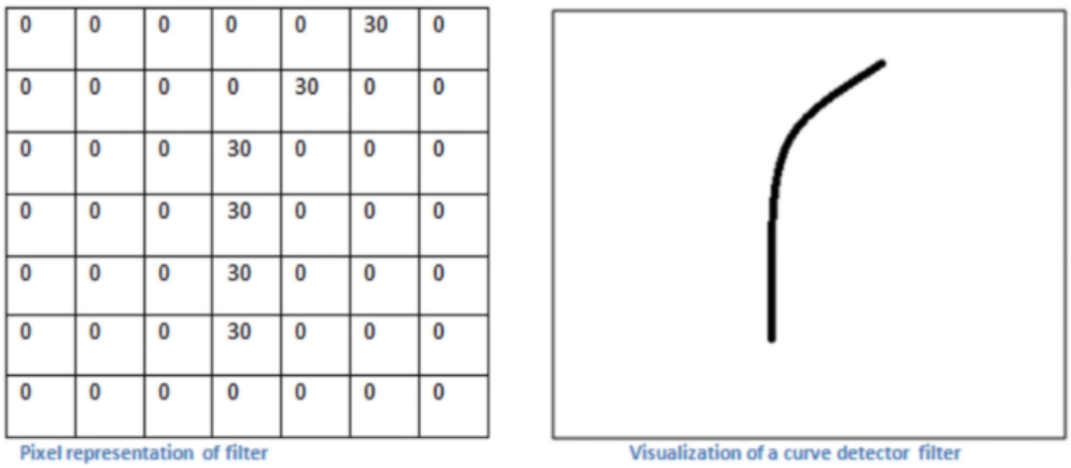

Figure 2.3: Beispiel von einem Filter, welcher eine Kurve in Pixel-Werten wiedergibt [Cha19]

Herkömmliche Convolutional Layer haben n Schichten und $n - 1$ Verbindungen; eine zwischen jeder Schicht und ihrer nachfolgenden Schicht. Für die jeweilige Ebene wird die *feature-map* der vorhergehenden Schicht als Eingabe verwendet und ihre eigene *feature-map* anschließend als Eingabe für die nachfolgende Schicht benutzt.

Anders als bei den Vorgängermodellen der Faltungsnetzwerke - mehrlagige Perceptrons (MLP) - berücksichtigen diese die räumliche Anordnung der untersuchten Pixel. Die Berücksichtigung des Umstands, dass nahegelegene Pixel normalerweise miteinander in Beziehung stehen, ermöglichte es, effizientere Modelle zum maschinellen Lernen aus Bilddaten zu kreieren.

CNN-basierte Architekturen sind im Bereich der Bildverarbeitung mittlerweile allgegenwärtig und dominant geworden. CNNs sind dank mehrerer Schichten bei der Erzielung genauer Modelle **stichproben- und auch rechnerisch effizient**, da sie weniger Parameter als vollständig verbundene Architekturen benötigen und weil Faltungen leicht über Grafikprozessor-Kerne (GPU) parallelisiert werden können. Es erscheint sinnvoll, dass bei jeder Methode, bei der versucht wird Objekte zu erkennen, die genaue Position des Objekts im Bild nicht übermäßig berücksichtigt werden sollte. Wie ein konkretes Objekt aussieht, hängt nicht davon ab, wo es sich in der Umgebung befindet. CNNs systematisieren diese Idee der räumlichen Invarianz und nutzen sie, um nützliche Darstellungen mit weniger Parametern zu lernen.

2.3 Super Mario

Das 2D Jump and Run Spiel Super Mario Bros. erschien im Jahre 1985 (Japan) zwei Jahre später in Deutschland, vertrieben von der Firma Nintendo für das Spielekonsolensystem Nintendo Entertainment System (NES). Das Spielziel ist es ein Level von Links nach Rechts zu durchqueren und dabei allerlei Hindernissen (Gegner, Säulen, Gräben, ...) zu überwinden. Technische gesehen, ist jedes Frame ein Bild der Spielemmgebung und hat folgende Dimension 3x240x256 (Anzahl der Farbkanäle, Höhe, Breite). Ziel ist es vor Ablauf einer festgelegten Zeitdauer von 400 Sekunden das Level komplett zu durchlaufen. Kollisionen der Spielfigur mit bestimmten Objekten der Spielwelt haben zur Folge, dass der aktuelle Lauf abgebrochen wird und neubegonnen werden muss. Ein Spieldurchlauf enthält in der Regel drei solcher Versuche. Durch das Sammeln von bestimmten Objekten (Münzen, Pilze, ...) kann sich die spielende Person gegebenenfalls Vorteile verschaffen. [Kau18b]

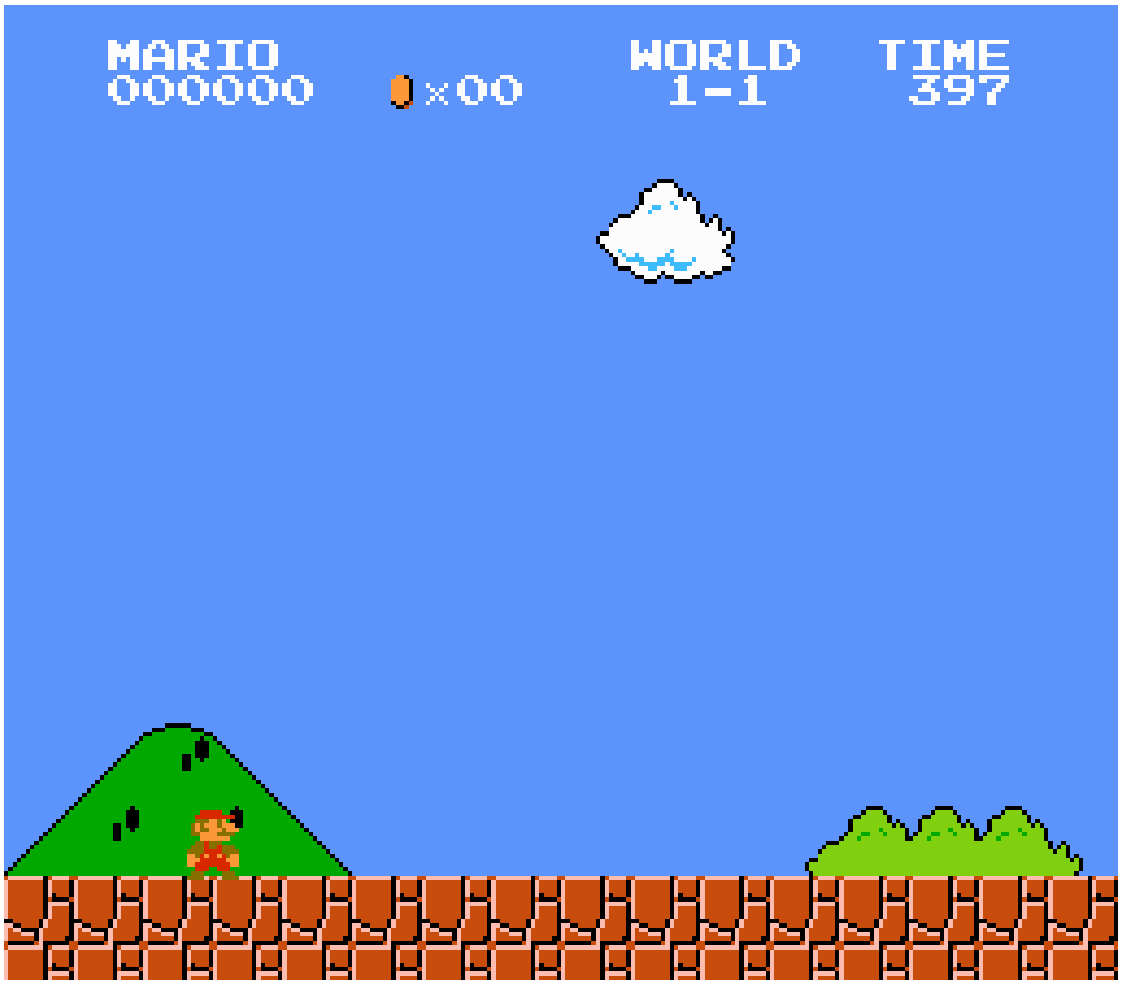

Figure 2.4: Zeigt das Super Mario Bros. in Level 1 [Kau18a]

Chapter 3

Conception

Im Reinforcement Learning kommen verschiedene Techniken (Q-Learning, Double Q-Learning) zum Einsatz, die einen Algorithmus in die Lage versetzen sollen, ein Spiel zu gewinnen. Nachfolgende Textabschnitte beleuchten Vor- und Nachteile verschiedener Konzepte und begründen die Vorgehensweise des Projektes.

Im Gegensatz zu Q-Learning, umgeht die Technik des Double Q-Learning das Problem des Overfittings - der Überanpassung des Algorithmus' an die Trainigsumgebung und lernt zudem noch schneller als erstgenannte Vorgehensweise.[Has10]

3.1 Double Q-Learning

Bevor das Konzept des Double Q-Learnings genauer beschrieben werden kann, muss auf die zuvor genannten Q-Values [WD92b] eingegangen werden. Die Verwendung von Q-Values hat gegenüber des Markov decision process den Vorteil, dass die Policy implizit mitgelernt wird und nicht separat für jede Entscheidung berechnet werden muss.[SB11](Kapitel 6)

Beim Markov decision process wird ein Wert für jeden State ausgerechnet, die Idee der Q-Values hingegen ist es, für jede Aktion einen Wert zu berechnen. Formal bedeutet das:

$$Q*(s,a) = \sum_{s\prime \in S} T(s,a,s\prime)[R(s,a,s\prime) + \gamma * V*(s\prime)] \tag{3.1}$$

Nach der Einführung in die Q-Values kann das Konzept des Double Q-Learnings erläutert werden. Double Q-Learning orientiert sich am Ursprungskonzept Q-Learning von [WD92a]. Modelle die das Q-Learning benutzen, haben das Problem dazu zu tendieren, einzelne Zustand-Aktions-Paaren zu überschätzen. Dieses Phänomen kann sich dadurch manifestieren, dass während der Suche nach den Aktionen die den Reward

maximieren, durch das wiederholte Anwenden der Update-Regel auf den Schätzer (Estimator) kleine Abweichungen akkumulieren.[TS93][Has10]

Double Q-Learning ist eine Verbesserung des Q-Learnings und hat dieses Problem nicht mehr. Um das Problem mit der Verzerrung des Schätzers zu lösen, wird nicht nur eine Q-Value Funktion verwendet, sondern zwei beim Lernen und beim Updaten der Q-Values.[Has10]

Die Idee sind zwei separate Funktionen die dazu dienen, sich gegeneinander zu updaten. Im Weiteren werde diese Funktionen Q^a und Q^b genannt. Im Originalpapier von 2010 [Has10] geschieht dieses Updaten randomisiert. Das Paper wurde 2015 noch ein Mal überarbeitet [HGS15] und der Algorithmus verändert, um die Randomisierung zu umgehen. In dieser Variante finden sich keine zwei unabhängigen Funktionen Q^a und Q^b. Sondern Q^a übernimmt die Aktionsauswahl und Q^b die Aktionsüberwachung. Dies kann man mathematisch wie folgt ausdrücken.

$$Q*(s_t, a_t) \approx r_t + yQ(s_{t+1}, argmax_a\prime Q\prime(s_t, a_t)) \tag{3.2}$$

Damit wird der mittlere quadratische Fehler langsam zwischen Q^a und dem wahren Wert minimiert. Außerdem werden in zyklischen Schritten die Parameter von Q^a zu Q^b kopiert. Die eben erwähnte Anpassung von 2015 findet auch in dieser Untersuchung Anwendung. Am Rande sei hier noch erwähnt, dass Double Q-Learning im Jahr 2018 noch einmal überarbeit wurde (jetzt "Clipped Double Q-learning").[FHM18]
Ferner muss bei SARSA-Algorithmen (kurz für State-Action-Reward-State-Action) zu den auch DQN gehört, auf das Exploration/Exploitation Problem geachtet werden. Das Problem wird auch "dilemma of exploration and exploitation" ([SB11]) bezeichnet. Es besagt, dass übermäßige Exploration die Maximierung der kurzfristigen Belohnung verhindert, da ausgewählte "Explorations"-Aktionen negative Belohnungen bekommen könnten. Zu wenig Exploitation kann bedeuten, dass die Umgebung nicht komplett erkundet wurde. Es ist denkbar, dass Aktionen die im späteren Verlauf des Spiels von Bedeutung wären in der Explorationsphase nicht optimal maximiert wurden, da sie während dieses Zeitraums nicht genügend Reward erbracht haben, um als nützlich abgespeichert zu werden. Um dieses Dilemma zu lösen, wird der e-greedy Wert eingeführt. Er repräsentiert die Wahrscheinlichkeit, ob eine mit Hilfe der Q-Values ausgewählte Aktion ausgeführt wird oder eine randomisierte Funktion Anwendung findet. Der e-greedy Wert wird im Verlauf des Trainings immer weiter herabgesetzt. Ein hoher e-greedy Wert am Anfang des Trainings führt häufig zu einer random Aktion, was der Erkundung dient. Gegen Ende ist der Wert verhältnisweise niedrig, demnach werden die Q-Values häufiger zur Auswahl der nächsten Aktion bemührt, was in vermehrter Exploitation

mündet.[Tok10]

3.2 Architektur des Netzes

Wie bereits erklärt, gibt die Umgebung nach jeder Aktion ein Bild zurück. Um mit dem Bild arbeiten zu können, wird ein CNN benutzt. Da für das Funktionieren des Algorithmus keine Farbbilder oder räumliche Tiefe verarbeitet werden muss, wurde auf ein einfaches CNN zurückgegriffen. Durch die kompaktere Architektur, mussten weniger weights & biases angepasst werden, was zu einer verbesserten Performance hinsichtlich der benötigten Ressourcen während des Trainings führte. Die verwendete Architektur (LeNet [Zha20] (Chapter 6.6)) ist nachfolgend aufgeführt:

	Input	kernel_size	stride	Output
convolutional layers 1	4x84x84	8x8	4	32x32
ReLu				
convolutional layers 2	32x32	4x4	2	64x64
ReLu				
convolutional layers 3	64x64	3x3	1	64x64
ReLu				

Table 3.1: Convolutional Blocks Elements

In der Tabelle sieht man den Aufbau des Convolutional Blocks bestehend aus 6 Elementen. Nach jedem der drei Convolutional layers befindet sich eine Rectified Linear Unit (ReLU). Der zweite Teil des CNNs ist der Fully-connected Block, bestehend aus zwei linearen Layern. Der Input des ersten Layers umfasst 3136 Neurone hat und bildet seinen Output auf 512 Neurone ab. Auch diese beiden Layer sind durch eine ReLU verbunden. Der letzte lineare Layer hat einen Input von 512 Neuronen und umfasst im Output die Anzahl der Actions, die das Environment hat. Auf bau wurde nach dem Pytorch Tutorials.[Fen17]

Chapter 4

Implementation

4.1 Framework

Bevor es mit der Implementierung des Double-Q Learning Algorithmus losgehen kann, muss die Entscheidung getroffen werden, welches Framework verwendet wird. Es gibt zwei weit verbreitete Frameworks: PyTorch und TensorFlow. PyTorch wurde 2016 durch das *AI Research Lab* von Facebook veröffentlicht und verlässt sich, wie man am Namen schon vermutet, auf die Programmiersprache Python, hat aber auch eine Schnittstelle für C++. TensorFlow hat Schnittstellen für viele Programmiersprachen. Diese zwei *frameworks* ähneln einander immer mehr, weil sie Funktionalitäten ihrer Wettbewerber integrieren. Es gibt jedoch immer noch eine Spaltung in zwei Nutzer-Gemeinschaften: PyTorch ist oft das bevorzugte Framework für maschinelles Lernen in der Forschung, TensorFlow ist hingegen in der Produktion weiter verbreitet.([Ind18]) Für dieses Projekt wurde PyTorch verwendet. Der wichtigsten Gründe für diese Entscheidung sind PyTorchs **leichte Benutzung** und die Möglichkeiten zur **feineren Einstellung** des Modells. Sie eignen sich gut für die Nutzung bei schnellerer, kleinerer Modelle.

4.2 Wrapper

Die Informationen die man vom Super Mario Environment bekommt, müssen zunächst an das Inputformat des CNNs angepasst werden. Zu diesem Zweck wurden Wapper geschrieben. Diese sind in `DQN\_Wapper.py` zu finden. Wrapper haben die Aufgabe, die Informationen, die von Super Mario Game kommen, in Informationen umzuwandeln, die der Agend benutzen kann. Es wurden 3 Wrapper geschrieben und eine von der `gym` Bibliothek[Bro16] benutzt.

Der erste selbst geschrieben Wrapper nennt sich `SkipFrame` und fasst eine bestimmte Anzahl von Frames zusammen, in dieser Implementierung sind es 4 Frames. Weil davon ausgegangen wird, dass zwischen einer geringen Anzahl Frames keine große Veränderung stattfindet, kann man, um die zu verarbeitende Datenmenge zu reduzieren, eine geringe Anzahl von Frames zusammenfassen, was das Modell performanter macht.(Quelle)

Es wird so zusammen gefasst, dass der letzte der 4 Frames genommen wird, die Rewards werden alle zusammen addiert.

Der nächste der 3 selbst geschrieben Wrapper ist der `GrayScaleObservation` Wrapper [Bro16]. Er erbte vom `gym.ObservationWrapper` und ist dafür da, den Frame von 3×240×256 auf Grauwerte zu reduzieren, sodass das man als Outputwert ein 1×240×256 Frame bekommt. Dies wird gemacht, weil der Informationsverlust sehr gering ist, man damit jedoch die Datenmenge die das CNN benötigt weiter reduzieren kann. Somit kann das CNN klein gehalten werden, was in einer kürzeren Trainignsphase resultiert.

`ResizeObservation` ist der letzte der selbst geschrieben Wrapper. Er skaliert die Größe des Frames von 240×240 auf 84×84

Außerdem wird noch der `FrameStack` verwendet. Das ist Wrapper der `gym()` Bibliothek[Bro16]. Dieser packt 4 Frames zusammen zu einem Frame. Dies wird benötigt, um Bewegungen wie springen besser Lernen zu können, in einzelnen Frames würden diese Bewegungen aufgrund der statischen Natur von Bildern nicht erkannt werden. Bei 4 Frames zusammen kann das CNN beispielsweise lernen, ob die Spielfigur gerade abspringt oder landet.

Alle diese Wapper wurden aus dem Beispiel von Pytorch übernommen. [Fen17]

4.3 Net

Die Datei `DQN_Net.py` implementiert die beiden neuronalen Netze, die einen Teil der zuvor beschriebenen Q-Funktion darstellen. In der Implementierung ist Q^a das online Model und Q^b das Target Model. Diese beiden Modelle haben die gleiche Netzstruktur. Diese wurde so implementiert, dass die Klasse von `nn.Module` von der Bibliothek torch erbte. Dem Konstruktor werden die Input- und Outputdimension übergeben.

Mit diesen Informationen und mit Hilfe der nn.Sequential Methode wird ein neu-

ronales Trainingsnetz gebaut. Die Struktur wird danach kopiert und aus ihr ein zweites Netz Targetmodell Q^b, geschaffen, dessen Parameter jedoch nicht intern durch Backpropagation angepasst werden können, sondern damit es von Außen geupdatet werden kann, wie oben beim DQN erklärt.

Code snippet 4.1: Parameter einfrieren

```
1 self.target = copy.deepcopy(self.online)
2
3  for p in self.target.parameters():
4   p.requires_grad = False
```

Zuletzt wurde noch eine forward Funktion gebaut, die je nach Model den Input an das online oder target model gibt.

4.4 Agent

In der Datei `DQN_Agend.py` ist die Maindatei und in dieser wird der restliche Teil des Double Q-Lering implementiert, hier findet sich auch die Main-Funktion des Systems.

Dem Konstruktor der Klasse muss das Environment - bestehend aus Wrapper und Spiel - übergeben werden. Zusätzlich wird ein Speicherpfad benötigt, unter welchem die Parameter gespeichert werden können. Weiterhin anzugeben ist die Anzahl der möglichen Actions, sowie die Dimension der Frames, in diesem Fall 4x84x84.

Algorithm 1 : Double Q-learning (Hasselt et al., 2015)

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma\ Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

Figure 4.1: DQL Algorithmus [Yoo19]

Im Konstruktor wird ein Objekt der Model Klasse initialisiert und verschiedene, globale Parameter gesetzt. Einer dieser Parameter ist `xploration_rate_decay` , was den zuvor erklärten epsilon-greedy 3.1 Wert repräsentiert. Außerdem ist in der Abbildung 4.1 zu sehen, dass es ein Puffer gibt, in welchem die bereits getätigten Aktion abgespeichert werden. Dieser Puffer ist ein deque, das bis zu 95000 Schritte speichert - aus diesem Puffer ruft das Netz die Trainingsdaten ab. Zuletzt wird ein Optimierer erstellt (hier Adam, Lernrate = 0.0025) und eine Loss-Funktion definiert (SmoothL1Loss).

Der Agent hat 10 Funktion. Hier werden kurz alle Funktion aufgelistet, eine genaue Beschreibung kann im Code auf github eingesehen werden. Die erste Funktion ist `get_action`. Bei der Funktion wird die Actions aus gewählt, entweder mit einem Random Aktion oder mit Hilfe des Models. Die Entscheidung darüber, welche Aktion ausgewählt wird, hängt vom epsilon-greedy Wert ab.

Die `cache` Funktion ist dafür da, dass der Schritt in den Puffer gespeichert wird.

Eine weitere Funktion ist die `recall` Funktion. Ihre Aufgabe besteht darin, aus den gespeicherten Daten im Puffer zufälig ausgewählte, gleich große Batches zu generieren und diese dann zurückzugeben.

Die beiden Funktionen `td_estimator` und `td_target` berechnen die Q-Values, einmal mit dem Q^a model, das passiert in der `td_estimator` Funktion, für das Q^b Model steht die `td_target` Funktion zur Verfügung.

Die Funktion update`_Q_online` nutzt Backpropagation um die Parameter des online Models anzupassen. Zu dieser Funktion kann man auch die `sync_Q_target` Funktion sehen, in gewissen Abständen updated sie die Parameter des Target Models mithilfe der Parameter des online Models. Die save Funktion speichert den Zustand des Models in einer Datei um es später laden zu können.

Die beiden wichtigsten Funktionen fürs Lernen sind die `lerne` und `run` Funktion. Die Run Funktion startet den Lernprozess. In ihr befindet sich die for-Schleife die über die Episoden iteriert und dabei alle notwendigen Funktionen aufruft. Die Lerne Funktion wird pro Episode einmal aufgerufen und startet alle Funktionen, die für das DQN wichtig ist auf.

Die letzte Funktion ist die `play` Funktion, sie lädt die Parameter eines trainierten Netzes und führt bis zum Game Over Aktionen im Environment aus.

Zum Abschluss alle Programmteile wurden vom Pytorch Tutorial inspiriert und übernommen.[Fen17] Den Sourcecode ist im GitHub Repositorie `https://github.com/deleAIst/Reinforcment_Lerning_for_Atari_Game` einzusehen

Chapter 5

Auswertung

Für die Auswertung werden beim Lernen verschiedene Werte geloggt und ausgewählte Werte als Plot dargestellt. Geloggt wurde unter anderem die Anzahl der Schritte, die seit dem Starten des Lernens im Environment gemacht wurden. Weiterhin wurde der momentane Epsilon Werte, die Mittelwerte des Reward, Length, Loss und Q-Values aufgezeichnet. Die Entscheidung nur die Mittelwerte zu speichern ist dadurch motiviert, dass es übersichtlicher ist nur jede 20. Episode zu loggen, so zeichnen sich Veränderungen deutlicher ab und es werden Ressourcen gespart. Die Logs sind mit Zeitstempeln versehen und enthalten auch die Dauer, die nötig war, 20 Episoden zu durchlaufen. Als Plot gespeichert werden Loss, Q-Values, Reward und Length. Es wurden drei verschiedene Trainingsdurchläufe mit unterschiedlicher Anzahl von Episoden untersucht. Weiteres Hypermeter Tuning konnte nicht vorgenommen werden, siehe Fazit **??**. Im weitern Verlauf werden die drei Durchläufe beschrieben und die gewonnenen Erkenntnisse geschildert.

5.1 Verlauf 1

Im ersten Durchlauf wurden 1360 Episoden trainiert, die Spielfigur hat in dieser Trainingszeit 1226846 Schritte im Environment gemacht. Das Training dauerte ungefähr 3 Stunden und 15 min.

In dem ersten Plot sieht man 5.1, wie viel Schritte durchschnittlich pro Episode gemacht wurden. Das Modell startet mit durchschnittlich 1400 Schritten und endet mit ca. 600 Schritten. Ein solcher Unterschied mag zunächst verwundern - sollte die Anazhl der Schritte nicht größer werden? Das Ziel des Spiels ist schließlich das Level zu durchlaufen und dafür sollte die Spielfigur mehr Schritte machen als am Anfang. Dieser

Gedanke ist prinzipiell richtig, dieses vermeintliiche Missverhältnis ist darin begründet, dass die Spielfigur Hindernisse anfänglich gar nicht oder nur mit überdurchschnittlich vielen Versuchen überwinden konnte. Ein anfänglicher Abfall der Anzahl der Schritte ist also zunächst nichts Schlechtes.

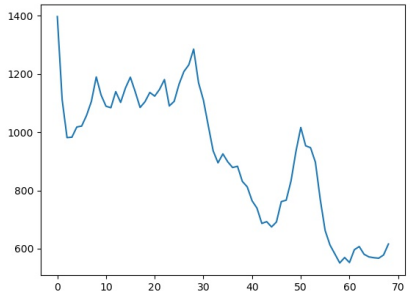

Figure 5.1: Mittelwert der zurückgelegten Länge pro 20 Episoden

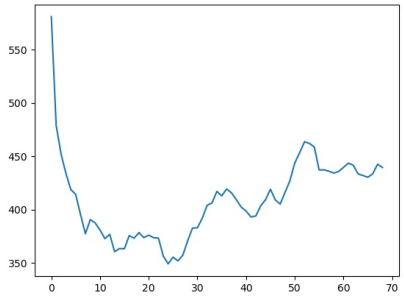

Figure 5.2: Mittelwert der Reward pro 20 Episoden

Der reward Plot 5.2 zeigt, wie viel reward es pro 20 Episoden durchschnittliche gegeben hat. Gestartet hat der Wert bei über 550 und fällt zunächst, steigt dann wieder an und endet wieder einem Wert um 550. Die Begründung, warum die reward Werte erst einmal fallen, ist ähnlich wie bei der Länge - weil es für jede Aktion mehr Punkte gibt und am Anfang mehr Schritte gemacht werden, ist auch der reward so hoch, nähert sich am Ende jedoch wieder an. Das ist darin begründet, dass die vom Modell gesteuerte Spielfigur jetzt schneller weiter kommt und dafür mehr Punkte bekommt, obwohl weniger Schritte getätigt werden. Ein Hinweis darauf, dass das Modell die Umgebung besser kennenlernt.

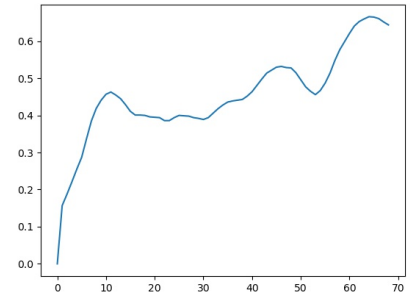

Figure 5.3: Mittelwert des Loss pro 20 Episoden

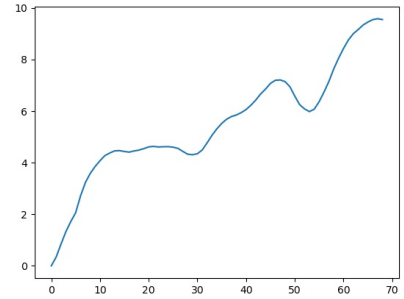

Figure 5.4: Mittelwert der Q-Values pro 20 Episoden

In den beiden Plots sieht man, dass sie bei 0 starten und dann steigen, bis die Q-Values 10 5.4 und der loss 0.6 5.3 erreicht. Beide Verhalten lassen sich daran erklären, dass die Umgebung ohne Vorwissen erlernt werden muss. Deshalb steigt der Loss zuerst und müsste sich dann wieder nach unten anpassen, wenn mehr über die Umgebung gelernt wurde.

Aus allen Kurven kann abgelesen werden, dass die Zeit zum Erlernen anscheinend noch nicht ausgereicht hat.

Versuch 2 werde ich hier nicht anhand von plots erklären, da die Ergebnisse sehr ähnlich zu denen aus Versuch 1 waren (10000 Episoden, 4917529 Schritte im Environment). Es wurde mehr als 12 Stunden trainiert. Weitere Daten zu diesem Durchgange können im github eingesehen werden.

5.2 Verlauf 3

Bei dem dritten Durchlauf wurde mit 32160 Episoden trainiert, der Versuch musste jedoch aus Zeitgründen abgebrochen werden. Geplant war ein 50000 Episoden umfassender Trainingszyklus. Doch bereits das Training der über 32000 Episoden nahm mehr als 2 Tage in Anspruch. Bis zu diesem Punkt wurden 15170814 Schritte getätigt.

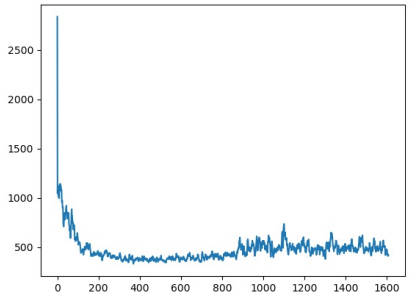

Figure 5.5: Mittelwert der zurückgelegten Länge pro 20 Episoden

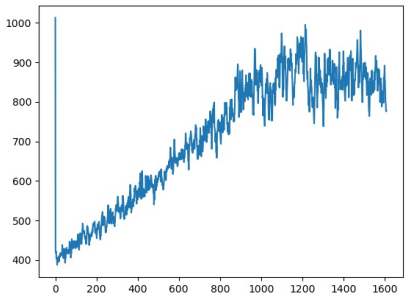

Figure 5.6: Mittelwert der Reward pro 20 Episoden

In den beiden Plots sieht man wieder die gleiche Anpassung, nur zeichnet sich bei den reward Werten ein stetiger Anstieg 5.6 ab, welcher gegen Ende anfängt langsam wieder zu sinken. Bei der Länge scheint sich das Modell leicht zu steigen und erst gegen Ende zu stagnieren, jedoch gibt es vereinzelte Sprünge nach oben und ob die Lernkurve wirklich stagniert, kann nicht mit letzter Gewissheit gesagt werden. Deutlich erkennbar jedoch ist, dass er den Weg am Anfang stark optimiert.

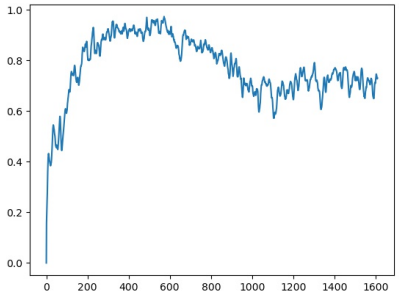

Figure 5.7: Mittelwert des Loss pro 20 Episoden

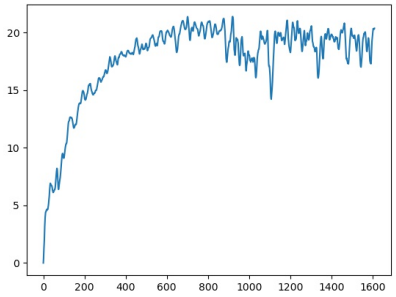

Figure 5.8: Mittelwert der Q-Values pro 20 Episoden

Beim loss sieht man den oben beschriebenen Verlauf am besten 5.7, der loss steigt bis fast zu 1 und wird weiter optimiert und bleibt um die 0.7 stehen. Es sieht so aus, als ob er nicht weiter sinkt, ob es sich um eine Stagnation oder lediglich um ein Dach handelt, müsste in weiteren Untersuchungen gezeigt werden. Beim Q-Value Plot 5.8 beobachtet man einen Anstieg der um den Wert 20 endet. Was man nicht darstellen kann ist, dass wenn man sich einen einzigen Durchlauf des Spiels anschaut, die Spielfigut verschiedene statische Hindernisse überwindet, den ersten Kontakt mit einem beweglichen Objekt (Gegner) jedoch nicht überwinden kann. Es müsste in weiteren Untersuchungen überprüft werden, ob dieses Problem durch weitere Episoden überwunden werden könnte.

Chapter 6

Fazit

6.1 Zusammenfassung

Wie oben gesehen, lässt sich erst einmal sagen, dass es möglich ist die Architektur des DQN, das Spielen des Spiels erlernen zu lassen. Ob ein ganzes Level mit den hier gewählten Parametern absolviert werden kann, muss noch untersucht werden; die für diese Versuche zu Rate gezogenenen Pytorch Tutorials [Fen17] legen jedoch nah, dass es möglich ist. Eine zusätzliche Erkenntnis ist, das RL sehr zeitaufwendig ist oder und mit hohen Kosten verbunden ist. Entsprechend starke Hardware ist also notwendig um das Modell schneller lernen zu lassen. Es ist schwierig zu beurteilen, ob die Hyperparameter richtig beziehungsweise optimal gewählt sind, eine Frage die sich fast nur durch weiteres Experimentieren klären lassen kann. Die Bedeutung der Zeitkomponente wird, zieht man den Aufwand in Betracht, der bereits notwenig war diese verhältnismäßig kleine K.I. zu trainieren, umso deutlicher.

6.2 Future Work

Für weitere Arbeiten in diesme Gebiet bleibt zu klären, welchen Einfluss ein Anheben der Episodenzahl auf das Ergebnis haben würde. Ob das Modell weiter lernt oder stagniert, den bekannten Teil also überoptimiert. Weiterhin wäre es möglich, die Hyperparameter zu tunen und zu untersuchen, welchen Einfluss sie auf das Lernverhalten des Modells haben. Natürlich wäre auch denkbar eine gänzlich andere Architektur aus dem Bereich des Reinforcement Learnings auf das selbe Problem hin zu trainineren und die performance der unterschiedlichen Modelle miteinander zu vergleichen.

List of Figures

List of Tables

Source Code Content

Bibliography

[BEL57] RICHARD BELLMAN. "A Markovian Decision Process". In: *Journal of Mathematics and Mechanics* 6.5 (1957), pp. 679–684. ISSN: 00959057, 19435274. URL: http://www.jstor.org/stable/24900506.

[BPS18] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. "Verifiable Reinforcement Learning via Policy Extraction". In: (May 22, 2018). arXiv: 1805.08328v2 [cs.LG].

[Cho18] Hana Chockler. *Computer aided verification : 30th International Conference, CAV 2018, held as part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*. Ed. by Hana Chockler and Georg Weissenbacher. Cham, Switzerland: Springer Open, 2018, pp. 623–642. ISBN: 9783319961453.

[FHM18] Scott Fujimoto, Herke van Hoof, and David Meger. "Addressing Function Approximation Error in Actor-Critic Methods". In: (Feb. 26, 2018). arXiv: 1802.09477v3 [cs.AI]. URL: https://arxiv.org/pdf/1802.09477.pdf.

[Gér19] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly UK Ltd., Oct. 1, 2019. 819 pp. ISBN: 1492032646. URL: https://www.ebook.de/de/product/33315532/aurelien_geron_hands_on_machine_learning_with_scikit_learn_keras_and_tensorflow.html.

[Has10] Hado Hasselt. "Double Q-learning". In: *Advances in Neural Information Processing Systems*. Ed. by J. Lafferty et al. Vol. 23. Curran Associates, Inc., 2010. URL: https://proceedings.neurips.cc/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf.

[HGS15] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: (Sept. 22, 2015). arXiv: 1509.06461v3 [cs.LG].

[HW62] D. H. Hubel and T. N. Wiesel. "Receptive fields, binocular interaction and functional architecture in the visual cortex". In: *The Journal of Physiology* 160.1 (Jan. 1962), pp. 106–154. DOI: 10.1113/jphysiol.1962.sp006837.

[Tok10] Michel Tokic. "Adaptive ε-greedy exploration in reinforcement learning based on value differences". In: *Annual Conference on Artificial Intelligence*. Springer. 2010, pp. 203–210. URL: https://link.springer.com/chapter/10.1007/978-3-642-16111-7_23.

[TS93] Sebastian Thrun and A. Schwartz. "Issues in Using Function Approximation for Reinforcement Learning". In: *Proceedings of the 1993 Connectionist Models Summer School*. Ed. by M. Mozer et al. Erlbaum Associates, June 1993.

[WD92a] Christopher J. C. H. Watkins and Peter Dayan. "Q-learning". In: *Machine Learning* 8.3-4 (May 1992), pp. 279–292. DOI: 10.1007/bf00992698.

[WD92b] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[ZF13] Matthew D. Zeiler and Rob Fergus. "Visualizing and Understanding Convolutional Networks". In: *CoRR* abs/1311.2901 (Nov. 12, 2013). arXiv: 1311.2901 [cs.CV]. URL: http://arxiv.org/abs/1311.2901.

[Zha20] Aston Zhang et al. *Dive into Deep Learning*. https://d2l.ai. 2020. URL: https://d2l.ai.

Online References

[Bro16] Greg Brockman et al. *OpenAI Gym*. June 5, 2016. arXiv: 1606.01540v1 [cs.LG].

[Fen17] Yuansong Feng et al. *TRAIN A MARIO-PLAYING RL AGENT*. PyTorch. 2017. URL: https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html.

[Ind18] Udacity India. *Tensorflow or PyTorch : The force is strong with which one?* Apr. 2018. URL: https://medium.com/@UdacityINDIA/tensorflow-or-pytorch-the-force-is-strong-with-which-one-68226bb7dab4.

[Kau18b] Christian Kauten. *Super Mario Bros for OpenAI Gym*. GitHub. 2018. URL: https://github.com/Kautenja/gym-super-mario-bros.

[Mis] MissingLink.ai. *Convolutional Neural Network Architecture: Forging Pathways to the Future*. MissingLink.ai. URL: https://missinglink.ai/guides/convolutional-neural-networks/convolutional-neural-network-architecture-forging-pathways-future/.

[SB11] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2011.

Image References

[Bha18] Shweta Bhatt. *Reinforcement Learning sequence*. 2018. URL: https://www.kdnuggets.com/images/reinforcement-learning-fig1-700.jpg (visited on 02/24/2021).

[Cha19] Himadri Sankar Chatterjee. *A Basic Introduction to Convolutional Neural Network*. 2019. URL: https://upload.wikimedia.org/wikipedia/commons/6/63/Typical_cnn.png (visited on 03/24/2021).

[Com15] Wikimedia Commons. *Typical CNN architecture*. 2015. URL: https://upload.wikimedia.org/wikipedia/commons/6/63/Typical_cnn.png (visited on 04/03/2018).

[Kau18a] Christian Kauten. *gym-super-mario-bros*. 2018. URL: https://user-images.githubusercontent.com/2184469/40948820-3d15e5c2-6830-11e8-81d4-ecfaffee0a14.png (visited on 03/24/2021).

[Yoo19] Chris Yoon. *Double Deep Q Networks*. 2019. URL: https://miro.medium.com/max/1068/1*NvvRn59pz-D1iSkBWpuIxA.png (visited on 03/24/2021).

Appendix A

About Appendices

Sourcecode des Projektes: `https://github.com/deleAIst/Reinforcment_Lerning_for_Atari_Game`