Zürcher Hochschule
für Angewandte Wissenschaften

School of Engineering

IDP Institut für Datenanalyse und Prozessdesign

Zürcher Hochschule
für Angewandte Wissenschaften

School of Engineering

ICP Institute of Computational Physics

Projektarbeit (Informatik / Wirtschaftsingenieurwesen)

Pinball Wizard - Intelligentes Flippern mit Deep Neural Networks

Autoren	Cédric Bürke Sascha Stutz
Betreuung	Dr. Oliver Dürr Dr. Thilo Stadelmann
Datum	22.12.2016

Zusammenfassung

Diese wissenschaftliche Arbeit befasst sich mit der Durchführbarkeit des Pinball Wizards. Dies ist ein realer Flipperkasten, der sich selber intelligentes Flippern beibringen soll. Der Kasten steht momentan bereits an der ZHAW, wobei der Ball mit einer Eventkamera getrackt wird und die Flipper per Computer gesteuert werden können. In dieser ersten Arbeit versteht sich die umfängliche Recherche und Erkenntniserbringung über die Machbarkeit des Pinball Wizards als Hauptaufgabe. Zudem wird ein vielversprechender Ansatz von Google DeepMind genausten untersucht, der mittels Deep Reinforcement Learning ein Neural Network trainiert, welches anschliessend selber spielen kann. Deren Algorithmus kann mit den gleichen Parametern 48 verschiedene Atarispiele erlernen und erzielt meist eine bessere Punktzahl als ein menschlicher Spieler. Das Spiel, das im Vergleich zu einem Random Player und einem Menschen am besten abschnitt, ist VideoPinball. Wie es der Name bereits verrät, ist es an einen Flipperkasten angelehnt. Da der originale in Luascript geschriebene Code wurde, ist eine Referenzimplementierung in Python in Kombination mit TensorFlow auf deren Performance untersucht worden. Trotz umfangreichen Debuggings des Codes konnte kein intelligentes Flippern erzielt werden, das im Bezug auf einen Menschen oder einen Random Player bedeutend besser abgeschnitten hätte. Zudem wurde getestet, ob sich die Aktivität der Flipper reduzieren lässt. Während des Projektverlaufs wurden zahlreiche Erkenntnisse gewonnen, welche erklären, weshalb sich dieses Spiel nicht als Grundlage für den Pinball Wizard eignet. Dabei spielen physikalische Eigenschaften einer Flipperumgebung eine Schlüsselrolle. Der Austausch der Eventkamera am Flipperkasten mit einer normalen Kamera wird für das Erkennen der Hindernisse im Flipperkasten empfohlen. Die Recherche ergab, dass der Google DeepMind Algorithmus durch die lange Trainingszeit von 38 Tagen nicht empfiehlt, um den Pinball Wizard komplett auf dem realen Flipperkasten zu trainieren. Seit der Erscheinung des DeepMind Papers wurden zudem neue Netzarchitekturen wie z. B. das Double Q-Network und Duelling Network entwickelt, welche ihre Agents besser trainieren. Es Bedarf weiterer Informationsbeschaffung in Bezug auf die Architekturen. Eine weitere Idee ist das Prioritized Experience Replay, dass gegenüber dem gewöhnlichen Replay Memory für eine kürzere Trainingszeit sorgt. Nach weiterer Recherche über die genannten Themen scheint die Durchführung des Pinball Wizards plausibel.

Vorwort

Wir möchten Dr. Oliver Dürr für seine Fehlersuche und wertvolle Unterstützung danken. Ebenfalls danken möchten wir Dr. Thilo Stadelmann für das Bereitstellen von Vorlagen und lehrreichen Schreib- und Arbeitstipps. Speziell die zur Verfügung stellen von Grundlagen in verschiedenen Bereichen des Machine Learnings ermöglichte einen schnellen Einstieg in diese Arbeit. Auch möchten wir Siraj Raval Dank aussprechen. Sein Code diente als Grundlage für diese Arbeit und er stand uns stets für Fragen zur Verfügung. Zu guter Letzt möchten wir Diego Browarnik für das Bereitstellen von Beispielbildern, der am Flipperkasten angebrachten Eventkamera, danken. Zudem wurde uns von ihm der momentane Aufbau umfangreich erklärt.

Erklärung betreffend das selbständige Verfassen einer Projektarbeit an der School of Engineering

Mit der Abgabe dieser Projektarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat. (Bei Gruppenarbeiten gelten die Leistungen der übrigen Gruppen-mitglieder nicht als fremde Hilfe.)

Der/die unterzeichnende Studierende erklärt, dass alle zitierten Quellen (auch Internetseiten) im Text oder Anhang korrekt nachgewiesen sind, d.h. dass die Projektarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Bei Verfehlungen aller Art treten die Paragraphen 39 und 40 (Unredlichkeit und Verfahren bei Unredlichkeit) der ZHAW Prüfungsordnung sowie die Bestimmungen der Disziplinarmassnahmen der Hochschulordnung in Kraft.

Ort, Datum:	Unterschriften:
………………………………………………	…………………………………………………………
	………………………………………………………...
	…………………………………………………………

Das Original dieses Formulars ist bei der ZHAW-Version aller abgegebenen Projektarbeiten zu Beginn der Dokumentation nach dem Abstract bzw. dem Management Summary mit Original-Unterschriften und -Datum (keine Kopie) einzufügen.

Inhaltsverzeichnis

1. Einleitung **6**
1.1. Ausgangslage 6
1.2. Zielsetzung 7
1.3. Übersicht der Arbeit 8

2. Grundlagen **9**
2.1. Machine Learning 9
2.1.1. Unsupervised und Supervised Learning 9
2.2. Reinforcement Learning 11
2.2.1. Notation und Terminologie 12
2.2.2. Elemente im Reinforcement Learning 12
2.2.3. Value Function 13
2.2.4. Exploration 15
2.2.5. Target Network 16
2.3. Neural Networks 16
2.3.1. Convolutional Neural Networks 19
2.4. Deep Reinforcement Learning 21
2.4.1. Replay Memory 21
2.5. OpenAI Gym 21
2.6. TensorFlow 21

3. Vorgehen **23**
3.1. Referenzimplementierung 23
3.1.1. Ablauf des Hauptprogramms 24
3.1.2. Trainieren des Netzwerks 25
3.1.3. Erstellen eines Eintrages im Memory 26
3.1.4. Gewichtanpassung im Main Network 27
3.2. Versuchsaufbau 27
3.2.1. Menschlicher Spieler 28
3.2.2. Random Player 28
3.2.3. Trainierte Netzwerke 28
3.3. Verwendete Software 30

4. Resultate **32**
4.1. Menschlicher Spieler 32
4.2. Random Player 33
4.3. Trainierter Agent 33
4.3.1. Spezifikation der Aktionen 33
4.3.2. Ermittlung des optimalen Action Penalties 34
4.3.3. Profiling 36

5. Diskussion der Resultate **38**

6. Ausblick **40**

7. Verzeichnisse **42**
(Abkürzungsverzeichnis) 48

A. Anhang **I**
A.1. Installationsanleitung I
A.2. Trainierte Netzwerke II
A.3. Graustufenbild III

1. Einleitung

Künstliche Intelligenz, im Folgenden auch als KI abgekürzt, ist ein Teilgebiet der Informatik und befasst sich mit intelligentem Verhalten und der Automatisierung von Maschinen. Ein vielversprechender Lösungsansatz von KI ist Machine Learning, womit sich diese Arbeit beschäftigt [1]. Der Schlüssel zu einer allgemeinen künstlichen Intelligenz ist zurzeit noch nicht gefunden. Die Menschheit ist jedoch im Begriff, immer mehr Aufgaben zu automatisieren und den Menschen obsolet zu machen. Wenn es eine allgemeingültige KI gäbe, die in jedem Bereich besser als der beste Mensch wäre, kämen Menschen lediglich dann zum Einsatz, wenn menschliche Intuition und menschliches Entscheidungsvermögen notwendig sind. Es wird von [2] behauptet, dass die Menschheit im Bezug auf den menschlichen Fortschritt kurz vor der grössten Veränderung der Geschichte steht. Siehe dazu Abbildung 1.1.

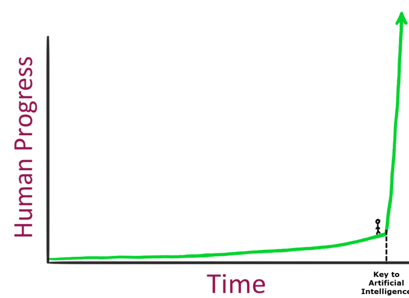

Abbildung 1.1.: Menschlicher Fortschritt mit dem Schlüssel zur künstlichen Intelligenz (http://waitbutwhy.com/2015/01/artificial-intelligence-revolution-1.html)

Wissenschaftler aus dem Bereich der KI beschäftigen sich statt mit industriellen oder wirtschaftlichen Problemen oft mit Spielen. Dies liegt daran, dass Spiele eine vereinfachte und abstrakte Version von realen Problemen sind, die einfacher simuliert werden können und man nicht mit voller Komplexität von Echtweltproblemen konfrontiert ist. Andererseits sind Spiele attraktiv und generieren dadurch schnell Popularität. Ein Beispiel aus der näheren Vergangenheit ist die 4-1 Niederlage des Weltmeisters Lee Se-Dol im Spiel Go gegen Googles AlphaGo Algorithmus im März 2016 [3]. Dies war das erste Mal, dass ein Computer einen Weltklasse Spieler in diesem für Computer hochkomplexes Spiel geschlagen hat. Dies führte dazu, dass die breite Öffentlichkeit darauf aufmerksam wurde [4]. Auch diese wissenschaftliche Arbeit beschäftigt sich mit einem Spiel, nämlich dem Flippern. Ein Maschine soll sich selber das Flippern beibringen. In dieser Grundlagenarbeit wird untersucht, wie eine Software selbständig lernen kann zu Flippern und dieses Verhalten anschliessend auf einen realen Flipperkasten übertragen werden kann.

1.1. Ausgangslage

Ein Programm, das auf einem realen Flipperkasten intelligent spielen lernt, wurde noch nicht publiziert. Dennoch ist von mehreren Quellen, wie [5], bekannt, dass das von Google für mehr als 500 Mio. US Dollar aufgekaufte Unternehmen DeepMind einen vielversprechenden Deep Q-Learning Algorithmus entwickelt hat. Dieser ist laut offiziellen Papers [6] fähig, 48 Atarispiele zu spielen. Darunter ist auch das Spiel VideoPinball, welches einen Flipperkasten darstellt. Der Ansatz, eine Ausführung dieses Algorithmus zu verwenden, scheint vielversprechend. Im offiziellen Dokument wird deklariert, dass der Deep Q-Learning Algorithmus von allen Atarispielen in VideoPinball im Vergleich zwischen einem Menschen und einem

Zufallsspieler das beste Ergebnis erzielt. Dies ist ausschlaggebend, dass sich diese wissenschaftliche Arbeit ausschliesslich auf DeepMinds Paper [6] fokussiert und deren Ansatz überprüft. Ob dies zu einem späteren Zeitpunkt auch auf den realen Flipperkasten übertragbar ist, ist unklar und zu überprüfen.

Ein Flipperkasten an der Zürcher Hochschule für Angewandte Wissenschaften existiert bereits. Dieser ist mit einer Eventkamera sowie einer Verkabelung zur maschinellen Ansteuerung ausgerüstet. Die Eventkamera liefert mit einer sehr hohen Framerate Bilder, welche Änderungen zwischen aufeinanderfolgenden Bildern (Events) darstellt. In einer weiteren Arbeit soll zudem der Score mittels Kamera ausgelesen werden. Der Aufbau ist in Abbildung 1.2 dargestellt.

Abbildung 1.2.: Momentaner Aufbau des Flipperkastens an der ZHAW

1.2. Zielsetzung

Das übergeordnete Ziel dieser wissenschaftlichen Arbeit ist es eine Grundlage zu schaffen, um einem Flipperkasten das Spielen beizubringen. Diese erste Arbeit konzentriert sich auf einige Unterthemen.

Es soll auf einem Emulator Netzwerke mit DeepMinds Deep Q-Network Algorithmus trainiert und analysiert werden. Hierbei stellt sich die Frage, ob es möglich ist, menschenähnlich zu spielen. Die Maschine soll zudem mit möglichst wenig unnötige Flipperschläge spielen lernen, da sich ein physischer Flipperkasten unter Dauerbelastung abnutzt. Dazu werden in den Experimenten unterschiedlich hohe Strafpunkte für Flipperschläge eingebaut und deren Auswirkungen analysiert.

Eine Abschätzung aufgrund von Literaturrecherchen und erworbenem Wissen, wie lange auf einem realen Flipperkasten trainiert werden muss, ist ein weiteres Ziel. Dabei soll der Flipperkasten mindestens auf menschlichem Niveau spielen, also zumindest denselben Score wie ein Durchschnittsspieler erreichen.

Die Informationsbeschaffung zur Durchführung und Machbarkeit eines intelligenten Flipperkasten versteht sich allerdings als Hauptaufgabe dieser Arbeit. Daraus soll eine erste Empfehlung zur Durchführung des übergeordneten Zieles entstehen und ein Ausblick in einen möglichen Versuchsaufbau mit dem realen Flipperkasten gewagt werden. Nützliche Informationen diesbezüglich sollen gesammelt und im Kapitel 6 aufgearbeitet werden.

1.3. Übersicht der Arbeit

Im Kapitel 2 sind die Grundlagen erörtert. Diese sind essenziell für das Verständnis des Versuchsaufbaus in Kapitel 3, in dem auch die Vorgehensweise erklärt ist. Des Weiteren ist in 3.1 der Ablauf einer Implementierung zum intelligenten Flippern abstrakt dargestellt und die unterschiedlichen Experimente erläutert. In Kapitel 4 werden die Daten der durchgeführten Versuche abgebildet und beschrieben. Die Diskussion und der Ausblick dienen dazu, gewonnene Informationen der Recherchen und durchgeführter Experimente zu interpretieren und zu analysieren. Zudem werden Empfehlungen zur Machbarkeit des Pinball Wizards abgegeben. Danach kommt das Literatur-, Abbildungs- und Tabellenverzeichnis, gefolgt vom Glossar, in welchem gewählte Abkürzungen in ausgeschriebener Form verzeichnet sind. Die Arbeit wird mit einem Anhang abgeschlossen, welcher weitere Daten und Informationen beinhaltet, die nicht direkt mit der Beantwortung der Fragestellungen zusammenhängen.

2. Grundlagen

Im Folgenden werden die Grundlagen erklärt, welche für das Verständnis des Versuchsaufbaus und der Resultate essenziell sind.

2.1. Machine Learning

Lernen wird laut [8] als «to gain knowledge, or understanding of, or skill in, by study, instruction, or experience» und «modification of a behavioral tendency by experience» definiert. Beim maschinellen Lernen soll der Computer so programmiert werden, dass aus eingespeisten Daten bestimmte Probleme gelöst werden können [9]. Die Maschine lernt, wenn die Struktur, die Daten oder das Programm automatisch aufgrund des Inputs bzw. der Umwelteinflüsse verändert wird, um die Genauigkeit des Outputs zu verbessern [8] [10]. Die fundamentale Idee dahinter ist, dass die Maschine selbst lernen soll, wie das Problem am besten gelöst wird, anstatt den Lösungsweg in einem expliziten Algorithmus festzuhalten [11].

Machine Learning ist kein neues Thema in der Wissenschaft. Der Gedanke, dass sich eine Maschine automatisch mit mathematischen Formeln iterativ anhand Inputs und Einflüsse anpassen soll, ist schon länger bekannt [12].

Das zunehmende Interesse am Machine Learning hat zahlreiche Gründe [14]. ML ist ein Kernbereich der künstlichen Intelligenz. Diese Technik ermöglicht den Computern das Generieren von Wissen aus Erfahrung. Ein Beispiel für Machine Learning ist das Klassifizieren von Internetseiten [16] oder Recommendation Engines, die Internetseiten aufgrund persönlichen Surfverhaltens und gesammelten Cookies vorschlagen [17]. Zum Beispiel wird bei einem Buchkauf auf Amazon überaus erfolgreich berechnet, welche weiteren Bücher für eine bestimmte Person interessant sein könnten. [14].

Mit steigender Rechenleistung der Computer [15] steigerte sich auch das Interesse an ML. Da sich die Leistung ungefähr alle zwei Jahre verdoppelt, auch bekannt als Moore's Law [18], wurde es möglich mit grossen Datenmengen umzugehen. Die hohe Parallelisierbarkeit von Rechenoperationen auf einer Grafikkarte wird auch beim Pinball Wizard zum Trainieren der Neural Networks benötigt.

2.1.1. Unsupervised und Supervised Learning

Bei Machine Learning wird unter anderem zwischen Supervised und Unsupervised Learning unterschieden.

Supervised Learning ist eine Technik, bei der eine Funktion aufgrund von Inputs und dem zugehörigen Output erlernt wird. Diese bestehen typischerweise aus einem Input $\mathbf{x}$ und einem zugehörigen Output y. Die Aufgabe der Maschine ist das Finden eines Modells, mit welchem aus dem Trainingsinput der Trainingsoutput berechnet wird. Das trainierte Modell wird dann verwendet, um mit Testdaten $\mathbf{x}$ die Vorhersage $\mathbf{y}$ zu generieren [20]. Bekannte Beispiele für Supervised Learning sind (multiple) lineare Regressionen [21] oder K-Nearest-Neighbours-Classification [19].

Unsupervised Learning bezeichnet laut [19] die Technik, dass die Maschine ausschliesslich Inputbeispiele ohne zugehörigen Output erhält und daraus ein Modell erstellt. Clustering ist eine dieser Unsupervised-Learning-Methoden.

Ein Vorhersagemodell kann zu generell sein, was unter dem Begriff Underfitting bekannt ist. Die Trainingsdaten haben einen geringfügigen Einfluss auf das Modell oder die Modellkomplexität ist gering.

Dies tritt auf, wenn beispielsweise zu wenig Parameter für die Vorhersage verwendet werden. Die Vorhersage aufgrund des Trainingsdateninputs weicht von den echten Outputs stark ab. Der sogenannte In-Sample-Error ist gross. Durch das Modell werden auch die Outputs der Testdaten falsch geschätzt, was dazu führt, dass der Out-Of-Sample-Error gross ist. Die vorhergesagten Daten weisen zwar einen kleine Varianz auf, haben aber einen grossen Bias.[22]

Das Vorhersagemodell kann sich andererseits auch zu stark an die Trainingsdaten anpassen. Dieses Problem nennt sich Overfitting. Das Modell ist zu spezifisch an die Trainingsdaten angepasst oder zu komplex. Dadurch passt das Modell sehr gut zu den zum Trainieren verwendeten Daten, aber weicht bei Vorhersagen von neuen Daten stärker ab als ein allgemeineres Modell. Der In-Sample-Error ist klein. Durch die starke Anpassung des Modells an die Trainingsdaten ist das Modell unflexibel und berechnet die Vorhersage der Testdaten mit einem hohen Out-Of-Sample-Error. Die Outputs der Testdaten haben eine hohe Varianz, dafür aber einen kleinen Bias.[23]

Over- und Underfitting, wie auch die beiden Errors in Abhängigkeit der Modellkomplexität, sind in der Abbildung 2.1 grafisch dargestellt.

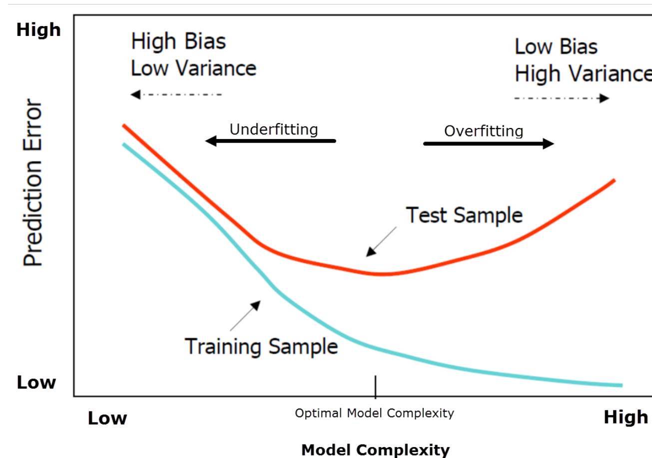

Abbildung 2.1.: Vorhersagefehler in Abhängigkeit der Modellkomplexität (http://oduerr.github.io/teaching/stdm/woche6/slides6.pdf)

Ziel ist also eine Balance zu finden, wie stark ein Modell an die Trainingsdaten angepasst werden soll. Simple Methoden zur Findung eines Optima gibt es meist nicht[24]. Eine Möglichkeit der Überprüfung der Modellgüte ist das Unterteilen der Daten mit Label in Trainigs- und Validationsset, Cross Validation genannt. Das Modell wird aufgrund der Trainingsdaten erstellt und mit den Validationset wird die Modellgüte überprüft[19].

2.2. Reinforcement Learning

Reinforcement Learning ist ein Bereich des Machine Learnings[25] und gehört weder in die Kategorie Unsupervised noch in die Kategorie Supervised Learning, sondern stellt eine neue Methode dar. Es gibt einen Agent, der seine Environment beobachtet. Dieser führt in der Environment Aktionen aus, die einen Reward auslösen können. Der Agent wird dabei so optimiert, dass er weiss, welche Aktion auf eine bestimmte Beobachtung ausführen soll 2.2. Dabei wird einem gewissen Input $\mathbf{x}$ eine spezifische Aktion zugewiesen. Dadurch soll ein skalarer Wert $\mathbf{y}$, Reward genannt, als Güte der ausgeführten Aktion im bestimmten Zustand maximiert werden. Allerdings wird nicht mitgeteilt, welche Aktionen in welchen Zuständen den Reward maximieren, dies soll vom Computer selber durch Wiederholung der Aktionen in Zuständen erkannt werden.

In RL werden nicht Lernmethoden charakterisiert, sondern das Lernen des Problems wird bezeichnet. Daher werden alle zugelassenen Methoden zur Lösung des Problems als Reinforcement Learning Methode akzeptiert. Agents, menschliche oder maschinelle Spieler, können auch verschiedenste ausgeklügelte Taktiken zur Lösung des Problems und der Maximierung des Rewards auffinden, wie dies in[26] ersichtlich ist. Gerade die Trial-And-Error-Suche nach der besten Aktion in einem bestimmten Zustand macht das Reinforcement Learning aus.

Die folgende Notation und Terminologie wird von[25] übernommen. Eine ausgeführte Aktion des Agents a_t zur Zeit t hat nicht nur einen Einfluss auf den unmittelbaren Reward r_t (immediate Reward), sondern auch auf zukünftige Zustände, sogenannte Observations o_t werden durch die Wahl und das Ausführen der Aktion beeinflusst. Eine Aktion wirkt sich zudem auf alle zu einem späteren Zeitpunkt möglichen Rewards aus. Bei jedem Zeitschritt t wird also eine Aktion a_t ausgeführt und eine Observation o_t, wie auch Reward r_t, dem Agent zurückgegeben. Das Environment erhält also eine Aktion a_t und durchläuft eine Veränderung. Dadurch werden Observation o_{t+1} und Reward r_{t+1} ausgegeben, wie in der Abbildung 2.2 ersichtlich.

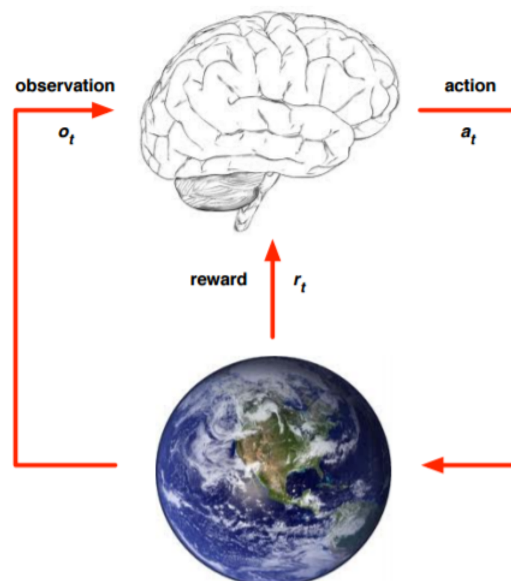

Abbildung 2.2.: Einfluss der gewählten Aktion a_t auf den vom Environment ausgegebenen Reward r_t und die Observations o_t
(http://hunch.net/ beygel/deep_rl_tutorial.pdf)

2.2.1. Notation und Terminologie

Ähnlich wie im RL werden ein Agent und ein Environment benötigt, wie in [25] beschrieben. Jede getätigte Aktion führt dazu, dass das die Environment Beobachtungen (Reward bspw.), wie auch der nächste Zustand, dem Agent zurückgibt. Normalerweise werden die Interaktionen mit dem Environment in verschiedene Serien von Episoden unterteilt. In jeder Serie wird der initiale Zustand s_0 von einer Verteilung $\mu(s_0)$ gezogen. Bei jedem Schritt wird eine Aktion ausgeführt, die entweder deterministisch oder zufällig gewählt wurde. Dies wird folgendermassen notiert: $a_t \sim \pi(a_t|s_0)$. Dabei wird beschrieben, dass die Aktionen zufällig gewählt oder nach der Wahrscheinlichkeitsverteilung π gezogen wird. π, die Wahrscheinlichkeitsverteilung, ist eine sogenannte Policy. Der nächste Zustand s_{t+1} und Reward r_t zu einer gewissen Zeit t werden gemäss der sogenannten Transition-Wahrscheinlichkeitsverteilung $P(s_{t+1}, r_t|s_t, a_t)$ bestimmt. Dieser Prozess der Beobachtung und Ausführung einer Aktion des Agents wird so lange repetiert, bis der Agent an einem Zeitpunkt T angelangt ist, an dem die Episode terminiert ist und der Endzustand s_T erreicht wird. Der gesamte Prozess läuft wie folgt ab:

$$
\begin{aligned}
s_0 &\sim \mu(s_0) \\
a_0 &\sim \pi(a_0|s_0)) \\
s_1, r_0 &\sim P(s_1, r_0|s_0, a_0) \\
a_1 &\sim \pi(a_1|s_1) \\
s_2, r_1 &\sim P(s_2, r_1|s_1, a_1) \\
&\ldots \\
a_{T-1} &\sim \pi(a_{T-1}|s_{T-1}) \\
s_T, r_{T-1} &\sim P(s_T|s_{T-1}, a_{T-1})
\end{aligned}
$$

2.2.2. Elemente im Reinforcement Learning

Neben dem Environment und dem Agent werden drei weitere essenzielle Elemente definiert: die Policy, eine Reward Function und eine Value Function, wie bei [25] beschrieben.

Zu jeder bestimmten Zeit beschreibt die Policy das Verhalten des Agents, sprich wird durch die Policy bestimmt, in welchem State welche Aktion auszuführen ist. Eine Policy ist also eine Abfolge von States und deren zugewiesenen Aktionen. Meist sind Policys stochastisch.

Die Reward Function definiert das Ziel des RL Problems. Dabei wird jedem State ein numerischer Wert zugeteilt. Dieser soll die Güte der momentanen Aktion quantifizieren. Meist werden erwünschte States mit einem positiven Wert versehen, unerwünschten States werden eine negative Zahl zugeordnet. Grundlegend ist das einzige Ziel des Reinforcement Learning Agent das Maximieren des Total Rewards über eine längere Zeitspanne. Dies ist die Summe aller immediate Rewards bis zum Zeitpunkt T. Dadurch, dass unerwünschte States negativ bewertet werden können, sollte der Agent Aktionen ausführen, um möglichst nicht in diese States zu gelangen. Wenn beispielsweise auf eine ausgeführte Aktion nach einer Policy ein tiefer oder negativer Reward folgt, wird die Policy geändert, um dies künftig zu verhindern. Auch Reward Functions sind generell stochastisch.

Value Functions beziehen sich im Gegensatz zu den Reward Functions nicht auf den unmittelbaren Reward in einem bestimmten State, sondern auf einen längeren Zeithorizont. Der Value eines States wird durch den zu erwartenden akkumulierten immediate Reward vom momentanen State s_t aus bis zum Ende s_T definiert. Der Value eines beschreibt die Güte, wenn einer bestimmten Policy gefolgt wird, in Miteinbezug der Wahrscheinlichkeiten der nächsten States und dessen immediate Rewards. So kann ein Status einerseits über einen kleinen immediate Reward verfügen und dennoch einen hohen Value haben, da dieser State oftmals durch States mit hohen immediate Rewards gefolgt wird. Der Value in einem terminierten Zustand ist dementsprechend 0.

Bei unbeschränkter Lebenszeit würde die Aufsummierung der zukünftigen Rewards

$$R = \sum_{k=t+1}^{T} r_k \tag{2.1}$$

gegen unendlich streben[27]. Selbst wenn in einem oder mehreren States die Lebenszeit terminiert ist, ist ungewiss, ob diese States jemals erreicht werden. Eine Entscheidung für State mit grösstem Value ist dadurch nicht mehr eindeutig gegeben. Naheliegend ist der Einsatz eines Diskontierungsfaktors γ. Dieser Parameter liegt im Wertebereich $\gamma = [0,1]$. Damit soll erreicht werden, dass weiter in der Zukunft liegende und daher auch unsicherere Rewards weniger gewichtet werden als diejenigen, die mit grosser Wahrscheinlichkeit in näherer Zukunft eintreffen werden. Rewards erhalten also eine Gewichtung je nach Unsicherheit des Eintreffens. Der diskontierte und aufsummierte zukünftige Reward (Success) ist

$$R = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k \ . \tag{2.2}$$

Values sind dementsprechend essenziell, um derjenigen Policy zu folgen, welche den maximalen erwarteten Success hat. Immediate Rewards werden durch das Environment erhalten, während Values eine Schätzung darstellen. Dies geschieht meist durch die vom Agent über die Lebensdauer getätigten Beobachtungen. Effiziente Abschätzungen der Value Functions sind eine der wichtigsten Komponenten im RL.

2.2.3. Value Function

Die Bestimmung der Value Function wird laut[25] wie folgt bestimmt. Eine Policy π weist dem State s $\in S$ und eine Aktion $a \in A(s)$ eine Wahrscheinlichkeit $\pi(a|s)$ zu. Diese beschreibt, wie wahrscheinlich es ist, die Aktion a auszuführen bei gegebenem State s. Der Value vom State s unter der Policy π wird als $v_\pi(s)$ notiert und beschreibt den erwarteten Success, sofern man in s startet und π folgt. $v_\pi(s)$ wird definiert, als

$$v_\pi(s) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s\right], \tag{2.3}$$

wobei $\mathbb{E}_\pi[\ \cdot\ ]$ den erwarteten Value beschreibt, der durch dem Folgen der Policy π entsteht und t ein Zeitschritt beliebiger Grösse ist. Die Funktion $v_\pi(s)$ wird als State-Value-Function für Policy π bezeichnet.

Den Value beim Ausführen der Aktion a im State s unter der Policy π wird als $q_\pi(s,a)$ notiert und bezeichnet den erwarteten Success, wenn in s gestartet, a ausgeführt und der Policy π gefolgt wird.

$$q_\pi(s,a) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s, A_t = a\right], \tag{2.4}$$

$q_\pi(s,a)$ wird als Action-Value-Function für Policy π definiert.

Die Value Functions $q_\pi(s,a)$ und $v_\pi(s)$ können einzig mittels Erfahrung geschätzt werden. Folgt der Agent n-mal der Policy π und die n immediate-Rewards werden gemittelt, so nähert sich dieser Schnitt dem Value $v_\pi(s)$ vom State s an. Mit gegen unendlich strebendem n wird die Approximation genauer. Durch das Mitteln über mehrere zufällige Stichproben von wahren Successes wird diese Methode als

Monte-Carlo-Methode bezeichnet. Oftmals ist es unmöglich und zeitlich inpraktikabel, für alle States alle Actions mehrmals durchzuspielen, um eine genaue Schätzung zu erhalten. Der Agent behält $v_\pi(s)$ und $q_\pi(s,a)$ bei als parametrisierte Funktionen und passt die Parameter an, um besser beobachtete Successes zuzuordnen. Allerdings können auch genaue Schätzungen entstehen, dies hängt von der parametrisierten Funktion ab.

Eine Eigenschaft der Value Function im RL ist, dass rekursive Beziehungen eingehalten werden. Für eine beliebige Policy π und einen beliebigen State s gilt für den möglichen nachfolgenden State s' folgendes:

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1} \,\middle|\, S_t = s\right] &(2.5)\\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k R_{t+k+2} \,\middle|\, S_t = s\right] &(2.6)\\
&= \sum_a \pi(a \mid s)\sum_{s'} p(s' \mid s,a)\left[r(s,a,s') + \gamma\mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+2} \,\middle|\, S_{t+1} = s'\right]\right] &(2.7)\\
&= \sum_a \pi(a \mid s)\sum_{s'} p(s' \mid s,a)\left[r(s,a,s') + \gamma v_\pi(s')\right] &(2.8)
\end{aligned}
$$

Die Gleichung 2.8 wird als Bellman Gleichung für v_π bezeichnet. Dabei wird die Beziehung vom Value eines States zum Value des nachfolgenden States beschrieben. Es wird also einen Schritt nach vorne in die möglichen nächsten States geblickt, wie in Abbildung 2.3 links beschrieben. Auf der rechten Seite der Grafik wird q_π grafisch dargestellt.

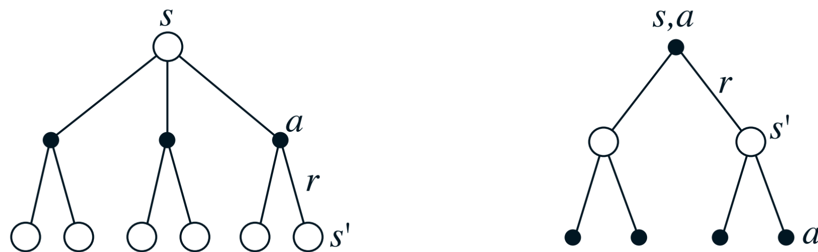

Abbildung 2.3.: Darstellungsdiagramm für v_π und q_π
(http://people.inf.elte.hu/lorincz/Files/RL_2006/SuttonBook.pdf)

Jeder weisse Kreis symbolisiert einen State und jeder schwarze Kreis stellt ein State-Action-Paar dar. Vom State s (linke Grafik) kann der Agent zwischen drei unterschiedlichen Aktionen auswählen. Das Environment könnte darauf verschiedene nachfolgende States s' zurückgeben, wie auch einen Reward r. Die Bellman Gleichung mittelt über alle Möglichkeiten und gewichtet jede mit der Wahrscheinlichkeit des Eintreffens. Der Value des States s muss gleich dem diskontierten Value des nächsten States s' zuzüglich dem immediate Reward r sein.

Generell sollte der Policy π gefolgt werden, sofern diese einen grösseren Value verspricht als Policy π', sprich $\pi \geq \pi' \Leftarrow v_\pi(s) \geq v_{\pi'}(s)$ für alle $s \in S$. Es gibt stets eine Policy, welche besser oder gleich gut ist. Diese wird/werden als optimale Policy definiert und als π^* geschrieben, der optimale Value ist v^* und wird berechnet als

$$v^*(s) = \max_\pi \; v_\pi(s) \tag{2.9}$$

für alle $s \in S$.

Optimale Policys haben also dieselben optimalen Action-Value-Functions, geschrieben als

$$q^*(s,a) = \max_{\pi} \; q_\pi(s,a), \tag{2.10}$$

für alle $s \in S$ und $a \in A(s)$. Für das State-Action-Paar (s,a) gibt die Gleichung 2.10 den erwarteten Success aus, wenn im State s die Aktion a ausgeführt wird und dadurch der optimalen Policy gefolgt wird. Dabei kann q^* auch in Bezug auf v^* wie folgt geschrieben werden:

$$q^*(s,a) = \mathbb{E}[R_{t+1} + \gamma v^*(S_{t+1}) \mid S_t = s, A_t = a]. \tag{2.11}$$

Aufgrund dessen, dass v^* eine Value Function für eine Policy darstellt, müssen die Bedingungen der Bellman Gleichung gelten. Da v^* jedoch optimal ist, kann die Gleichung in einer speziellen Form ohne Referenz auf eine spezifische Policy aufgestellt werden. Dies wird als Bellman Optimality Equation bezeichnet. Diese beschreibt, dass der Wert eines States unter der optimalen Policy gleich dem erwarteten Success für die beste Aktion von diesem State ist.

$$
\begin{aligned}
v^*(s) &= \max_a \mathbb{E}_{\pi*}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s, A_t = a\right] && (2.12)\\
&= \max_a \mathbb{E}_{\pi*}\left[R_{t+1} \; + \; \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \;\middle|\; S_t = s, A_t = a\right] && (2.13)\\
&= \max_a \mathbb{E}\big[R_{t+1} \; + \; \gamma v^*(S_{t+1}) \mid S_t = s, A_t = a\big] && (2.14)\\
&= \max_{a \in A(s)} \sum_{s'} p(s'|s,a)\big[r(s,a,s') \; + \; \gamma v^*(s')\big] && (2.15)
\end{aligned}
$$

Die Gleichungen 2.14 und 2.15 sind die oben bereits genannten Bellman Optimality Equations. Für q^* sehen diese wie folgt aus:

$$
\begin{aligned}
q^*(s,a) &= \mathbb{E}\big[R_{t+1} \; + \; \gamma \max_{a'} q^*(S_{t+1}, a') \big| S_t = s, A_t = a\big] && (2.16)\\
&= \sum_{s'} p(s'|s,a)\big[r(s,a,s') \; + \; \gamma \max_{a'} q^*(s', a')\big] && (2.17)
\end{aligned}
$$

Wurde v^* berechnet, so bekommt man die optimale Policy einfach. Für jeden State s wird es mindestens eine Aktion geben, bei welcher das Maximum in der der Bellman Optimality Equation erreicht wird. Mit q^* wird es noch einfacher, die optimale Aktion zu wählen. Für beliebige States s kann simpel die Aktion a gefunden werden, welche $q^*(s,a)$ maximiert. Die optimale Action-Value-Function erlaubt es optimale Aktionen zu finden ohne Vorwissen über den nachfolgenden State und dessen Values.

2.2.4. Exploration

Eine weitere Herausforderung im RL ist der Kompromiss zwischen dem Ausprobieren von neuen Aktionen und dem Ausführen von bereits erlernten Aktionen [28]. Um möglichst viel Reward zu erhalten sollte der Agent eine derjenigen Aktion wählen, die sich in der Vergangenheit bewährt hat und bereits durchgeführt wurde. Andererseits kann der Agent zu Beginn nicht wissen, welche Aktion den grössten Success erbringt. Der Agent muss dies also selber lernen. Der Agent führt neue Aktionen aus und favorisiert diejenige, die sich in den jeweiligen Zuständen als am besten kristallisiert, sprich den Success

maximiert. Um die korrekte Präferenz für die beste Aktion zu erhalten, müssen die unterschiedlichen Aktionen mehrmals ausprobiert werden. Nur so kann eine verlässliche Abschätzung des erwarteten Successes berechnet werden.

Es gibt jedoch auch weitere Gründe für das Explorieren von Aktionen, die der optimalen Policy nach nicht ausgeführt werden sollten [29]. Values von States werden gemittelt und darauf basierend die Entscheidung für eine Aktion a getroffen. So kann es vorkommen, dass in einem bestimmten State s eine Aktion ausgeführt wird, die zu einem gewissen Success führt. Diese Aktion wird als momentan beste Aktion in s bestärkt, denn ist der Agent ein weiteres Mal in s, so wird erneut die Aktion a ausgeführt. Allerdings ist es auch durchaus denkbar, dass eine andere Aktion maximalen Success erbringen würde, diese jedoch nie ausgeführt und später nicht bestärkt wurde. Die Einbringung von Stochastik in das Auswahlverfahren von Aktionen in bestimmten States ist daher essenziell. Speziell zu Beginn ist es sinnvoll, teils zufällige Aktionen zu wählen. Eine zuverlässige Mittlung des zu erwarteten Successes aller möglichen Aktionen erfolgt daraus. Optimal ist ein fliessender Übergang vom Explorieren neuer Aktionen und dem Bestärken von bereits Erlerntem. Durch diese Methode wird unter anderem Overfittig verhindert.

2.2.5. Target Network

Teils wird ein zweites Netzwerk benutzt, das sogenannte Target Network. Neben dem Main Network dient das Target Network zur Berechnung der Loss Funktion jeder Aktion während des Trainings. Der Grund für dessen Benutzung sind die sich stets verändernden Q Values (Kapitel 2.4.1) im Netzwerk. Das Netzwerk kann durch das Fallen in Feedback Loops zwischen dem Target und den geschätzten Q Values instabil werden. Um dieses Risiko abzuschwächen, werden die Gewichte im Target Network fixiert und nur periodisch und langsam angepasst. Daher ist der Einsatz eines Target Networks zusätzlich zum Main Network zu empfehlen. [30]

2.3. Neural Networks

Wie aus dem Namen Neural Networks vielleicht schon ersichtlich, stammt dieses Konzept aus der Neuroinformatik. Folgender Inhalt ist aus [31] und [32] erstellt worden. In den 50er und 60er Jahren definierte Frank Rosenblatt den Begriff Perzeptron, ein künstliches Neuron, als logisches Schwellenwertelement. Dieses besitzt einen oder mehrere binäre Eingänge x_i und gibt 0 oder 1 aus. Um die Wichtigkeit der Eingänge x_i miteinzubeziehen, fügte er den Eingängen Gewichte $w_1, w_2, ...$ hinzu. Sobald $\sum_i w_i * x_i$ den Schwellwert überschreitet, sendet das Perzeptron einen Impuls. Diese Idee bildet die Grundlage heutiger Neural Networks, im folgenden auch NN genannt. Formal geschrieben liest sich das folgendermassen:

$$\text{Output} = \begin{cases} 0 & \text{falls } \sum_i w_i x_i < \text{ Schwellwert} \\ 1 & \text{falls } \sum_j w_i x_i \geq \text{ Schwellwert} \end{cases} \tag{2.18}$$

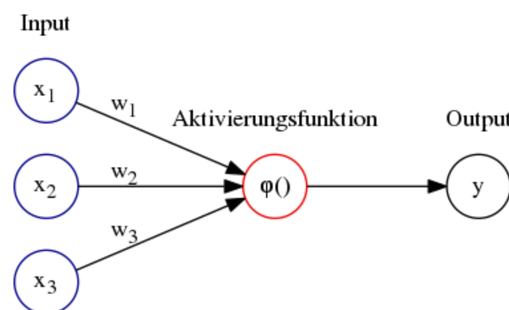

Abbildung 2.4.: Grafische Darstellung eines Neurons

Ein kurzes Beispiel zur Verständlichkeit: Ein Perzeptron, das entscheidet, ob ein Regenschirm mitgeführt werden soll, funktioniert wie folgt: Der Eingang x_1 gibt an, ob es zur Zeit regnet. x_2, ob der Wetterbericht Regen vorausgesagt hat. x_3 beschreibt die Zeitdauer, in welcher die Person im Freien ist.

x_1 wird mit $w_1 = 7$ gewichtet,x_2 mit $w_2 = 4$ und x_3 mit $w_3 = 2$.

Der Schwellwert wird auf 5 gesetzt. Es werden folgende Aussagen getroffen: Der Regen fliesst in Strömen trotzdem der Wetterbericht keinen vorausgesagt hat und die Person möchte kurze Zeit ausser Haus. Das Perzeptron entscheidet in diesem Fall einen Schirm mitzunehmen, da

$$\sum_i w_i * x_i = 1*7+0*4+0*4 = 7 \tag{2.19}$$

den Schwellwert übersteigt. Trifft jedoch keine dieser oben genannten Aussagen zu, ergibt die Summe 0 und somit ist auch die Ausgabe 0.

In einem NN besitzt ein Perzeptron/Neutron eine Aktivierungsfunktion $\varphi()$. In unserem obigen Beispiel war $\varphi()$ die in Formel 2.18 beschriebene Funktion. Der Output y eines Neurons wird somit als $\varphi(\sum_i w_i * x_i)$ berechnet. Eine Aktivierungsfunktion wird benötigt, um Nichtlinearität zu modellieren, welche oft in realen Daten vorkommt. Meist werden folgende Funktionen $\varphi(x)$ angetroffen:

- Sigmoid: $(\frac{e^x}{1+e^x})$, bildet auf den Bereich [0,1] ab
- TanH: $(\frac{\mathrm{e}^x - \mathrm{e}^{-x}}{\mathrm{e}^x + \mathrm{e}^{-x}}) = 2 * Sigmoid(2x) - 1$, bildet auf den Bereich [-1,1] ab
- ReLU (Rectified Linear Unit): $\max(0, x)$, bildet auf den Bereich [0,∞) ab

Früher waren vor allem die Sigmoid- und TanH-Funktion populär, da diese differenzierbar sind[33]. Heute werden oft ReLus verwendet (siehe Abbildung 2.5), weil diese Funktion sich schnell berechnen lassen und der Gradient nicht verschwindet (Verschwindender Gradient: $\varphi'(\varphi'(\varphi'(...))) \approx 0$ in grösseren Netzen). Ein Nachteil ist jedoch, dass die Funktion im Punkt 0 nicht differenzierbar ist.

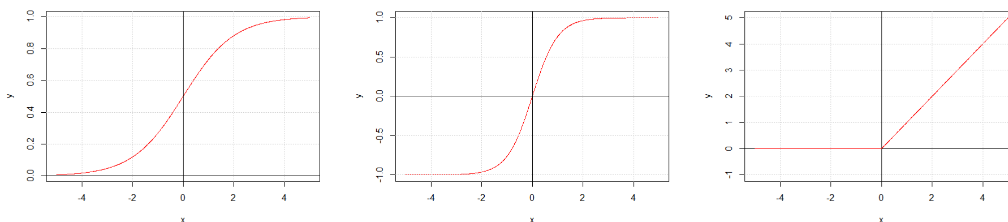

Abbildung 2.5.: Sigmoid-, ReLu- und Tangenshyperbolicus-Aktivierungsfunktion (v.l.n.r.)

Der folgende Abschnitt stützt sich auf die Vorlesung Learning and Intelligent Systems[33]. Die Grundidee eines NN ist es, Features direkt von den rohen Daten zu lernen. Um komplexere Funktionen zu approximieren, lassen sich wie in Abbildung 2.6 eine Vielzahl von Neuronen in Schichten und mehrere Schichten hintereinander zu einem Netzwerk verschalten. Diejenigen einzelnen Schichten, welche weder Eingang noch Ausgang sind, werden Hidden Layers genannt.

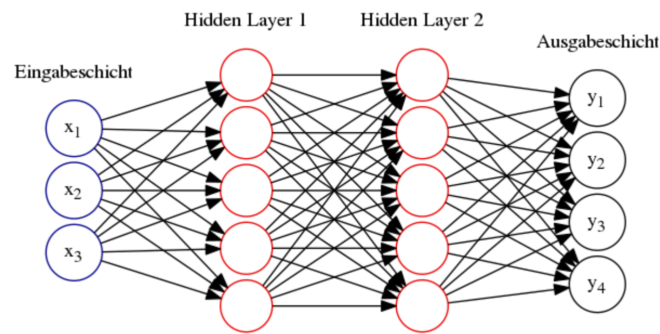

Abbildung 2.6.: Graphische Darstellung eines Neural Networks

Die verschiedenen Eingänge x_i werden hier mit jedem Knoten j der ersten Schicht (Hidden Layer 1) verbunden. Diese Kante besitzt das Gewicht $w_{i,j}$. Der Knoten j wiederum ist mit allen Neuronen der 2. Schicht verbunden (Hidden Layer 2). Repräsentiert werden alle Kantengewichte zwischen zwei Schichten durch die Matrix $\mathbf{W}^l$.

Der Outputvektor $\mathbf{y}$ eines Netzes wird mit der Feedforeward Methode berechnet. Folgender Algorithmus zeigt den Ablauf:

Eingabeschicht: $\mathbf{v}^{(0)} = \mathbf{x}$

Für jedes Hidden Layer:

$l = 1 : L-1$
$\mathbf{v}^{(l)} = \varphi(\mathbf{W}^{(l)}\mathbf{v}^{(l-1)})$

Ausgabeschicht: $\mathbf{f} = \mathbf{W}^{(L)}\mathbf{v}^{(L-1)}$
Vorhersage $\mathbf{y} = \mathbf{f}$ (Regression), $\mathbf{y} = sign(\mathbf{f})$ (Klassifikation)

Hierbei werden oftmals Matrixmultiplikationen ausgeführt. Diese können stark parallelisiert werden, was das effiziente Auswerten eines Netzes auf einer GPU begünstigt.

Das Finden einer Lösung für die optimalen Gewichte in einem NN ist ein nicht-konvexes Optimierungsproblem, man kann jedoch versuchen, ein lokales Optimum zu finden. Dies bedeutet, dass die Gewichte iterativ angepasst werden, bis sich der Loss $\ell()$, z.B $\delta_{total} = \frac{1}{2}\sum_i(output_i - y_i)^2$, nicht mehr gross ändert. Dies wird auch Trainieren eines Netzes genannt. Mit einem Optimierer wie z.B SGD (Stochastic Gradient Descent) wird versucht, möglichst optimale Gewichte $\mathbf{W}^* = argmin_{\mathbf{W}} \sum_{i=1}^{n} \ell(\mathbf{W}; \mathbf{y}_i, \mathbf{x}_i)$ zu finden.
$\eta_t =$ Learning Rate zur Zeit t $D =$ Trainingsdaten

Gewichte $\mathbf{W}$ initialisieren
For t = 1,2,...

Ziehe einen Datenpunkt $(\mathbf{x}, \mathbf{y}) \in D$ i.i.d
Mache einen Schritt in die negative Gradientenrichtung
$\mathbf{W} \leftarrow \mathbf{W} - \eta_t \bigtriangledown_{\mathbf{W}} \ell(\mathbf{W}; \mathbf{y}, \mathbf{x})$

Um die Gefahr zu vermindern, in einem lokalen Optimum stecken zu bleiben, gibt es für SGD eine Erweiterung namens Momentum. Die Idee dahinter ist, dass man sich mit jedem Gewichtsupdate nicht nur in die Richtung des Gradienten bewegt, sondern auch in die des letzten Gewichtupdates.
$m =$ Momentum

$a \leftarrow m * a + \eta_t \bigtriangledown_{\mathbf{W}} \ell(\mathbf{W}; \mathbf{y}, \mathbf{x})$
$\mathbf{W} \leftarrow \mathbf{W} - a$

Um jedoch SDG anwenden zu können, muss $\bigtriangledown_{\mathbf{W}} \ell(\mathbf{W}, \mathbf{y}, \mathbf{x})$ berechnet werden. D.h., für jedes Gewicht zwischen zwei verbundenen Einheiten i und j muss $\frac{\partial}{\partial w_{i,j}} \ell(\mathbf{W}; \mathbf{y}, \mathbf{x})$ berechnet werden. Mittels Backpropagation werden die Fehlersignale von der Ausgabeschicht zur Eingabeschicht zurückgerechnet.

Für jeden Knoten j der Ausgabeschicht

Berechne Fehlersignal $\delta_j = \ell'_j(f_i)$
Für jeden Knoten i der Schicht L, $\frac{\partial}{\partial w_{j,i}} = \delta_j v_i$

Für jede Einheit j auf dem Hidden Layer $\ell = L-1 : -1 : 1$

Berechne den Fehler $\delta_j = \phi'(v_j) \sum_{i \in Layer_{\ell+1}} w_{i,j}\delta_i$
Für jede Einheit i der Schicht $\ell - 1$, $\frac{\partial}{\partial w_{j,i}} = \delta_j v_i$

Da NN eine nicht-konvexe Funktion darstellen, spielt die Initialisierung der Gewichte keine Rolle. Meist liefert eine zufällige Initialisierung zufriedenstellende Ergebnisse:

$$w_{i,j} \sim \mathcal{N}(0, 0.1)$$

$$w_{i,j} \sim \mathcal{N}(0, \sqrt{1/Layer_{l+1}})$$

2.3.1. Convolutional Neural Networks

Folgender Abschnitt basiert auf dem Data Science Blog[34]. Convolutional Neural Networks sind vor allem in der Bilderkennung und -klassifikation, wie auch für Klassifikationsaufgaben im Natural-Language-Processing-Umfeld relevant.

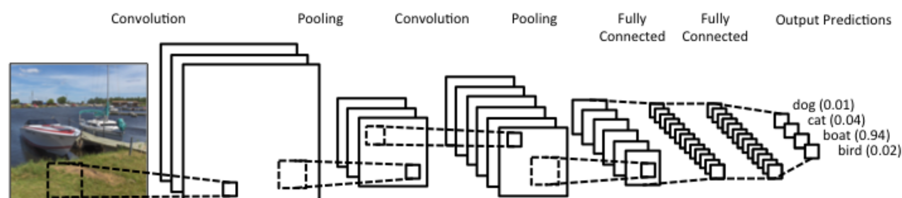

Abbildung 2.7.: Convolutional Neural Network LeNet
(https://www.clarifai.com/technology)

In einem CNN gibt es vier grundlegende Operationen:

- Convolution
- Aktivierungsfunktion, siehe Kapitel 2.3
- Pooling oder Sub Sampling
- Klassifikation

Die Operation, welche die Bilddaten auf eine andere Ebene abbildet, nennt sich Convolution. Ein Bild besteht aus einer 2-dimensionalen Matrix, wobei jeder Pixel/Eintrag einen Wert von 0 bis 255 besitzt. Dies ist eine vereinfachte Annahme und bezieht sich auf Bilder in Graustufen, wodurch je nach Graustufe der Wert zustande kommt. Bei einem Convolutional Step wird eine kleinere Matrix, auch Filter genannt, über die Bildmatrix gelegt und damit ein neuer Wert in der Featurematrix berechnet. Der Filter wird bei jedem Schritt um eine Einheit verschoben, wobei im Anschluss die Werte der beiden Matrizen Elementweise multipliziert werden. Wie in Abbildung 2.8 dargestellt, wurden hier die ersten beiden Schritte bereits ausgeführt und in die Feature Map übertragen.

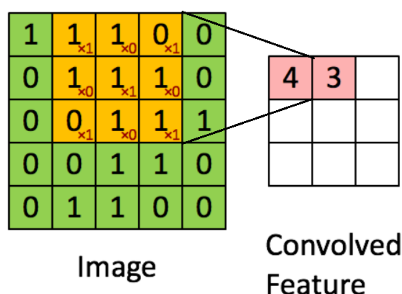

Abbildung 2.8.: Filter in Aktion
(http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution)

Mit einer Convolution werden Features aus einem Bild extrahiert. Der Filter, welcher hierbei verschoben wird, erhält die Zusammengehörigkeit von Pixeln in der selben Umgebung. Die Feature Map enthät 3 wichtige Parameter:

- Tiefe: Die Anzahl an Convolved Features, für die jeweils ein separater Filter verwendet worden ist. Siehe die gestapelten Convolved Features in Abbilding 2.7.
- Stride: Dieser Wert gibt an, um wie viel der Filter bei jedem Schritt verschoben wird.
- Padding: Hierbei wird definiert, wie sich der Filter, beziehungsweise die Bildmatrix, verhaltet, wenn er am Rande ankommt. Der Rand kann beispielsweise mit Nullen aufgefüllt werden, s.d. der Filter am Rande weitergeschoben und zum Rechnen verwendet werden kann, obwohl er nur ein Teil des eigentlichen Bildes abdeckt.

Spactial Pooling reduziert die Dimensionalität der Feature Maps, erhält aber die wichtigsten Daten. Beispiele dafür sind die Funktionen $\max, Durchschnitt$ und $Summe$. Beim Max Pooling wird beispielsweise jeweils der grösste Wert innerhalb des Fensters auf der Bildmatrix oder Feature Map in die neue Feature Map übertragen, siehe Abbildung 2.9.

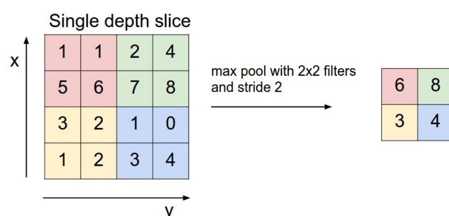

Abbildung 2.9.: Darsetellung von Max Pooling
(http://cs231n.github.io/convolutional-networks/)

Schlussendlich wird die Klassifizierung mittels Fully Connected Schichten durchgeführt. Da CNNs eine Unterart von NNs sind, lassen sich auch diese mittels Backpropagation trainieren.

2.4. Deep Reinforcement Learning

Deep Reinforcement Learning setzt sich aus Reinforcement Learning und Deep Neural Networks zusammen. DNN bezeichnet NNs mit mehreren Hidden Layers. Mittels eines solchen Netzwerkes wird die Q-Funktion des Reinforcement-Learning-Vorgangs approximiert.

2.4.1. Replay Memory

Reinforcement Learning mittels eines nicht linearen Funktionsapproximators, wie es Q-Funktion repräsentierende sind NNs, ist bekannt dafür, instabil zu sein [6]. Grund dafür sind:

- Korrelation von aufeinanderfolgenden Beobachtungen
- Kleine Updates der Q-Funktion ändern die Policy und somit die Datenverteilung signifikant
- Korrelation von Action Values (Q) und Targetvalues $(r + \gamma \max_{a'} Q(s', a'))$

Um diese Ursachen zu neutralisieren, wurde das Replay Memory eingeführt. Unter Replay Memory versteht man einen Speicher, der die zuletzt ausgeführten Aktionen und deren zugehörige Bilder speichert. Während des Lernprozesses werden immer wieder zufällige Einheiten (Minibatches) aus dem Speicher gezogen, mit denen die Q-Funktion trainiert wird.

2.5. OpenAI Gym

OpenAI Gym [35] ist ein Open Source Tool für die Entwicklung und den Vergleich von ML Algorithmen. 2016 wurde dieses Tool von Elon Musk und Sam Altman mit dem Hintergrundgedanken, dass Recherchen und Forschung reproduzier- und erweiterbar sein sollten, erstellt. Zuerst wurde die Umgebung nur für interne Nutzung freigegeben, Elon Musk ist aber bekannt dafür, ein Verfechter des freien Wissensaustausches zu sein. Eine Betaversion ist seit April 2016 massenzugänglich und verfügt über eine wachsende Anzahl an unterschiedlichen Umgebungen.

In dieser Trainingsumgebung können Agenten unter anderem das Spielen von Atari Spielen wie Video-Pinball lernen. Für den Austausch von Programmierern untereinander stellt OpenAI Gym ein Chatroom und neuerdings (seit 5.12.2016) auch ein Forum zur Verfügung. Auf der Webseite gibt es zudem ein Leaderboard für jede Trainingsumgebung, in der man sich mit anderen Personen an der Effizienz seines Algorithmus messen kann.

2.6. TensorFlow

TensorFlow [36] ist eine Open Source Bibliothek, die mit sogenannten Data Flow Graphs mathematische Berechnungen ausführen kann. Dieser Graph besteht aus Knoten und Kanten. Ein Knoten repräsentiert eine mathematische Operation oder ein Eingang, eine Kante die Übertragung eines Tensors (Datenmatrix) an die nächste Operation. Das ganze Framework ist ausgelegt, um auf einer oder mehreren CPUs oder GPUs ausgeführt zu werden. Es gibt Möglichkeiten Flowgraphs in Android zu laden und auszuführen. Diese Bibliothek die zu ML- und Deeplearningzwecken entworfen wurde entstammt aus Googles ML-Forschungsgruppe Brain Team.

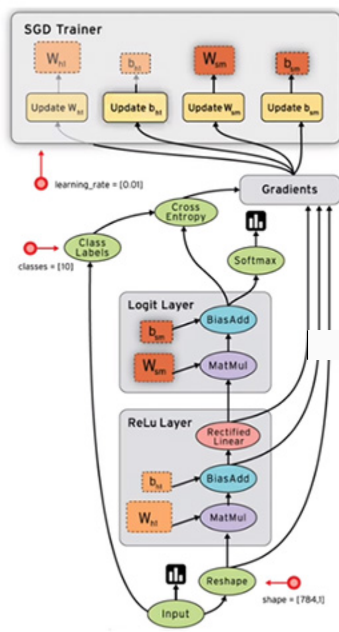

Abbildung 2.10.: Data Flow Graph (http://nlpx.net/wp/wp-content/uploads/2015/11/TensorFlow-graph1.jpg)

Um eine simplen Data Flow Graph für eine Multiplikation von 2 Zahlen a und b in Python zu erstellen gehen wir wie folgt vor:

```
import tensorflow as tf

mat1 = tf.placeholder(tf.float32)
mat2 = tf.placeholder(tf.float32)
product = tf.matmul(mat1,mat2)

with tf.Session() as sess:
    print sess.run([product], feed_dict
={mat1:[[1., 2.]], mat2:[[3.], [5.]]})
```

Eine Berechnung wird als Graph angesehen. Dieser Graph wird innerhalb einer Session ausgeführt. Mittels eines feeds werden die Daten in den Graphen gefüttert und mit der Fetchoperation wieder herausgeholt.

3. Vorgehen

Durch das Wiederaufleben von ML[14] bestand die Chance, dass das erfolgreiche Trainieren eines Flipperkastens in dieser oder einer ähnlichen Form bereits durchgeführt wurde. Dies würde es ermöglichen, dass ein zur Verfügung stehender Algorithmus auf die genaue Problemstellung dieser wissenschaftlichen Arbeit angepasst werden könnte. Die Recherche über bereits bestehende und geeignete Implementierungen war daher essenziell. Nach dem Aneignen der dazugehörigen Theorie zur Implementierung ist auch das penible Verständnis des Codeablaufs wichtig.

Danach folgt eine Anpassung, beziehungsweise eine Erweiterung, der bestehenden Implementierung. Unterschiedlich hohe Penaltisierungen der Flipperschläge sollen eingebaut werden, damit wird nervöses Schlagen der Flipper vermindert. Die Maschine soll allerdings nicht nur in Sachen Punktzahl einem Menschen ähneln oder diesen übertreffen, sondern auch in Bezug auf die Anzahl ausgelöster Flipperschläge pro Zeit einem Menschen gleichen. Andererseits würde der Agent zu jeder Möglichkeit Flippern, wodurch intelligentes Spielen nicht identifizierbar wäre. Die trainierten Netze werden untereinander, wie aber auch mit der Performance eines Menschen und einem Agent, der zufällige Aktionen auslöst, verglichen.

3.1. Referenzimplementierung

Leider steht das offizielle Paper von Google DeepMind der Öffentlichkeit nicht zur Verfügung, welches am erfolgversprechendsten wirkte. Allerdings wurde bei der Recherche auf eine Referenzimplementierung von Siraj Raval gestossen. Siraj bezeichnet sich selbst als Lehrer von ML, der sich mit offen zugänglichen Repositories[37] und dem YouTube Kanal[38] im Internet hohen Renommees erfreut. Ein Repository, Game-AI, beschäftigt sich mit dem Erlernen von Ataripielen, darunter auch VideoPinball und stützt sich auf Google DeepMind ab. Mehrere Anpassungen mussten allerdings vorgenommen werden, damit der Algorithmus sinngemäss ablief.

Der Algorithmus ist für das Verständnis in mehrere Stücke unterteilt.

3.1.1. Ablauf des Hauptprogramms

In einem ersten Schritt werden die Parameter initialisiert. Danach werden das Environment und das Netz initialisiert. Darin wird bspw. abgefragt, welches Atarispiel aufgerufen wird und welches Format die Inputpixel haben. Dann wird das Netzwerk trainiert 3.1.2. Am Ende werden mit dem trainierten Netz Episoden durchgespielt und diese aufgezeichnet. Dieser Ablauf ist in Abbildung 3.1 dargestellt.

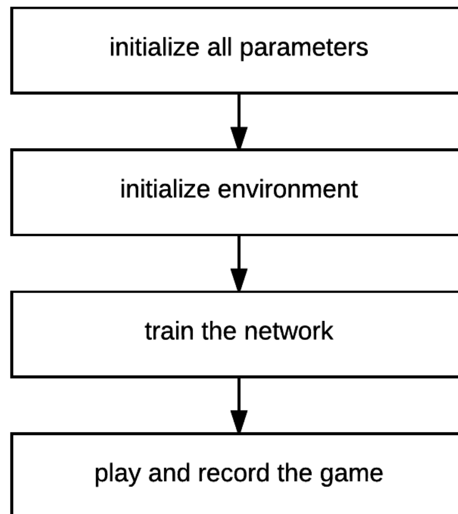

Abbildung 3.1.: Ablaufdiagramm des Hauptprogramms

3.1.2. Trainieren des Netzwerks

Das Netz wird wie folgt trainiert: Das Spiel startet und der Agent führt stets zufällige Aktionen aus. Ein Tupel wird in das Memory gespeichert (siehe Kapitel 3.1.3). Es folgt eine neue Situation, in der erneut ein Tupel erstellt und ins Memory gespeichert wird. Diese Füllung des Replay Memories wird bis zu einer angegebenen Anzahl an Tupels durchgeführt. Danach findet der zweite Teil des Trainings statt. Der Training Step wird auf 0 gesetzt und der ϵ-greedy Value aufgrund des aktuellen Schrittes berechnet. Dieser wird mit zunehmenden Training Steps um einen konstanten Wert linear reduziert. Der ϵ-greedy Value, oder auch Explorationsrate genannt, bezeichnet, wie viel Mal im Schnitt eine zufällige Aktion statt der in der Situation besten Aktion ausgeführt werden soll. Ein neuer Eintrag im Replay Memory wird erstellt (vlg. Kapitel 3.1.3). Alle 4 Schritte werden die Gewichte im Main Network angepasst (vgl. Kapitel 3.1.4). Jeder 100'000. Schritt wird das Main Network als Target Network abgespeichert. Falls noch nicht alle Trainingsschritte durchgeführt sind, wird der aktuelle ϵ-greedy Value berechnet, der Training Step um 1 erhöht und weiter trainiert. Anderenfalls ist as Training abgeschlossen.

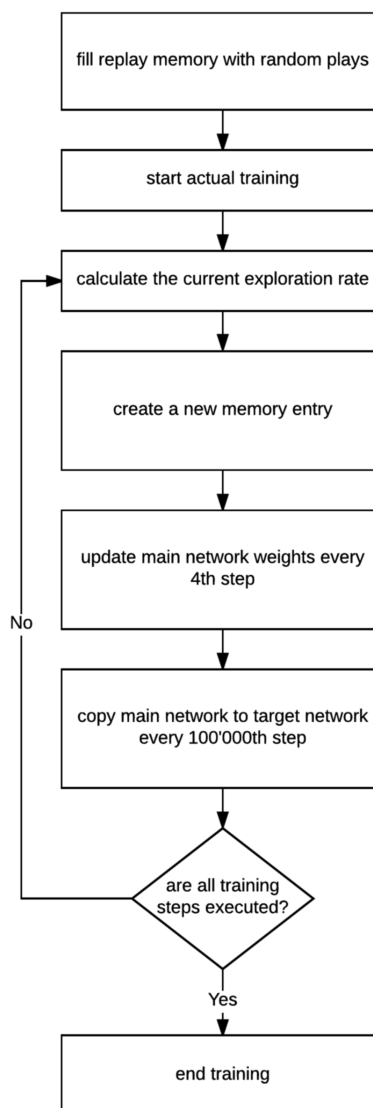

Abbildung 3.2.: Ablaufdiagramm des Netzwerktrainings

3.1.3. Erstellen eines Eintrages im Memory

Das Erstellen eines neuen Memoryeintrages erfolgt folgendermassen: Die Explorationsrate wird verwendet, um zu entscheiden, ob eine zufällige Aktion ausgeführt wird oder nicht. Je grösser die Explorationsrate ϵ, desto wahrscheinlicher fällt die Wahl auf eine zufällige Aktion. Fällt die Wahl nicht auf eine zufällige Aktion passiert folgendes: Die letzten 4 Frames in Form eines Arrays werden abgefragt. Aufgrund dieses States von 4 Frames werden vom Target Network die Action Values für alle Aktionen berechnet. Die Aktion mit dem grössten Action Value wird ausgeführt, dadurch entsteht ein neuer State, der nachfolgende State. Zudem wird überprüft, ob das Spiel fertig ist. In einem nächsten Schritt wird der Reward auf -1 bis 1 geclippt. Dies bedeutet, falls der Reward kleiner als -1 beziehungsweise grösser als 1 ist, dass der Reward als -1 oder 1 definiert wird. Werte innerhalb dieses Intervalls bleiben in ihrer Grösse bestehen. Der momentane, der nachfolgende State, der Reward, die gespielte Aktion und ob das Spiel fertig ist wird als Tupel bezeichnet. Es wird überprüft, ob das Replay Memory bereits voll ist. Falls dies der Fall ist, wird das älteste Tupel aus dem Memory gelöscht, um für das Neue Platz zu machen. Dies besteht aus den Einträgen der momentanen und nächsten State, dem Reward, der ausgeführten Aktion und ob das Spiel zu Ende ist oder nicht. Falls das Replay Memory noch nicht voll ist, wird das neue Tupel hinzugefügt.

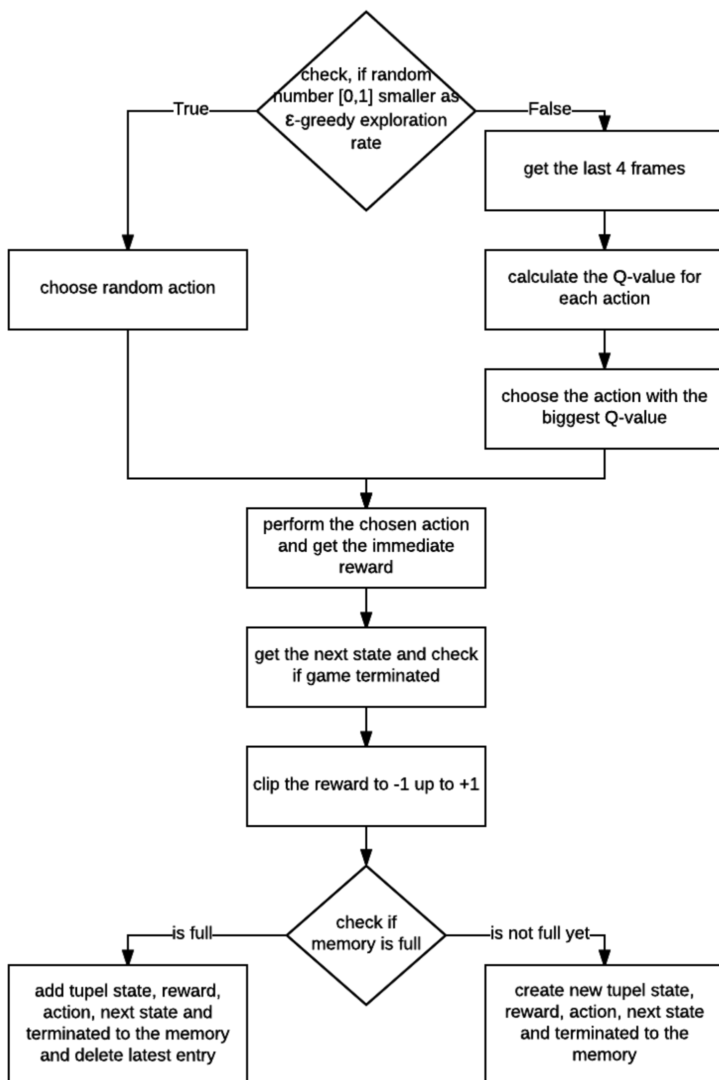

Abbildung 3.3.: Ablaufdiagramm des Erstellens eines Memoryeintrages

3.1.4. Gewichtanpassung im Main Network

Die Anpassung der Gewichte des Main Networks läuft wie folgt ab: 32 zufällige Tupel werden aus dem Memory gezogen, dabei bezeichnet 32 die gewählte Batch Size (vgl. Tabelle 3.1). Die Rewards der Tupel werden addiert und als Success bezeichnet. Analog wird zusammengezählt, wie viel mal das Spiel terminiert war und als Failures abgespeichert. Der absolute Q-Target Wert aus dem Array der Q-Targets aller Aktionen wird berechnet als

$$Q_{target,abs} = r_t + ((1.0 - terminal) * (\gamma * \max\ Q_{target,A})). \tag{3.1}$$

Dabei ist die Variable terminal 1.0, falls das Spiel fertig ist und 0.0 sonst. Mit dem definierten Q-Target wird nun der Loss ermittelt (vgl. Kaptel 2.3). Nun wird nach Gewichten abgeleitet und diese so angepasst, dass der Loss minimal wird. Sprich wird eine Backpropagation durchgeführt.

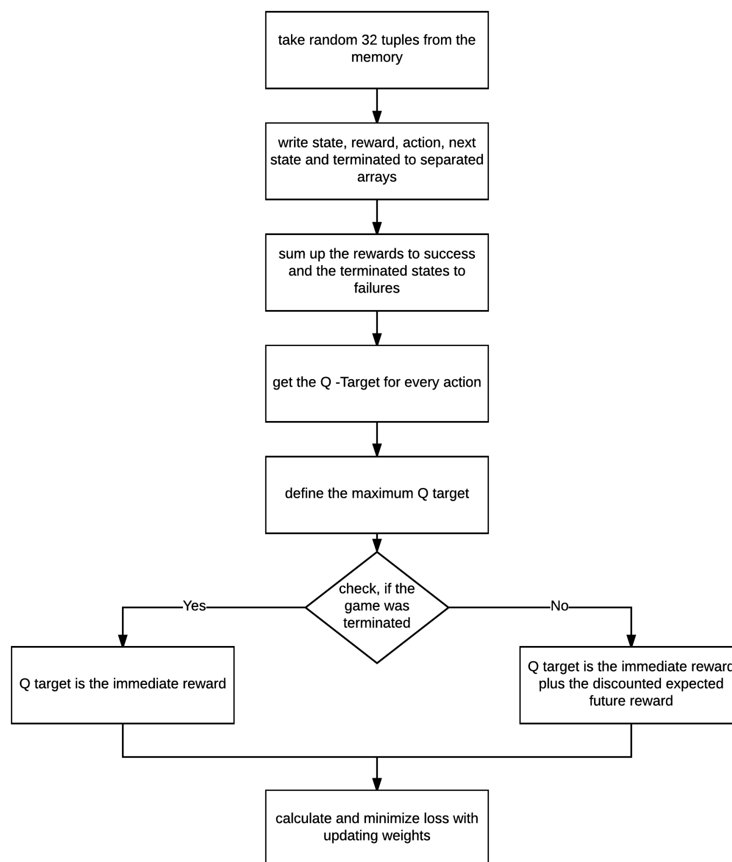

Abbildung 3.4.: Ablaufdiagramm der Gewichtanpassung im Main Network

3.2. Versuchsaufbau

Der Vergleich eines trainierten Agents mit einem Menschen ist essenziell, um einen Referenzwert in der Performance zu erhalten. Zudem stellt der Random Player einen weiteren Benchmark für die trainierten Netze. Dabei sind die Wahrscheinlichkeiten für alle Aktionen $a \in A$ uniformverteilt, es werden zufällig ohne Beachtung des Environments Aktionen ausgeführt. Der Vergleich der Agents mit einem Random Player und einem menschlichen Spieler sollen Anzeichen intelligenten Spielens deutlich machen. Zudem

wird ersichtlich, ob der Mensch, der Random Player oder das trainierte Netzwerk besser spielt in Bezug auf die Punktzahl pro Spiel. Auch die Anzahl Schläge pro Spiel ist von Interesse.

Ein Spiel gilt als beendet, falls der blaue Ball vertikal unter die Höhe beider Flipper fällt und auf der rechten Seite zum erneuten Abschuss bereit steht, also herunterfällt. Diese Situation ist in der Abbildung 3.5 dargestellt. Die Zeit zwischen zwei End-, bzw. Anfangszuständen, wird als ein Spiel definiert.

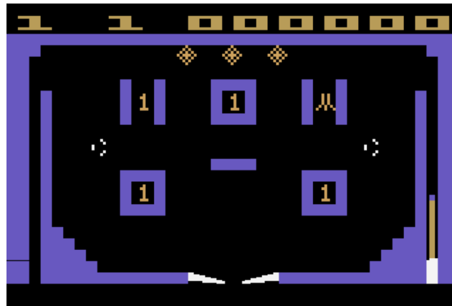

Abbildung 3.5.: Start- und Endzustand beim VideoPinball

3.2.1. Menschlicher Spieler

Bei der Testperson handelt es sich um den Autor Sascha Stutz, der zuvor noch nie VideoPinball gespielt hat. Dadurch kann von einem ungeübten Spieler mit ungefähr menschlicher Intelligenz ausgegangen werden. Erst wurden 10 Spiele auf dem VideoPinball-Emulatoren [40] ohne Speicherung jeglicher Daten durchgeführt, um sich mit dem Spiel vertraut zu machen. Danach wurden von der Versuchsperson 50 Spiele durchgeführt, sprich 50-mal wurde versucht, eine möglichst hohe Punktzahl zu erreichen. Die Auswahl der Aktionen erfolgte nach Ermessen des Spielers, wurde also durch menschliche Intelligenz und menschliche Intuition bestimmt. Eine zweiten Person, Autor Cédric Bürke, notierte die Anzahl Flipperschläge und die erreichte Punktzahl. Diese wurde gleich der Referenzimplementierung (siehe Kapitel 3.1.3) gezählt: Eine Erhöhung des echten Spielstandes wurde als $+1$ gewertet, uninteressant, ob der Score im Spiel um 100 oder 1 stieg. Dies ermöglicht einen direkten Vergleich der erreichten Punktzahl der Agents.

3.2.2. Random Player

Ein weiterer Referenzwert stellt der Random Player dar. Dabei wird ein zufällig initialisiertes Netz verwendet. Durch den konstanten ϵ-greedy Value von 1.0 werden beim Spielen nie Aktionen aus dem Netz ausgeführt, sondern zufällig für eine Aktion entschieden. Das Erstellen eines Memoryeintrages ist daher unwesentlich, da das Memory nie verwendet wird. Erneut wurde die Punktzahl um 1 erhöht, falls der Spielstand sich verändert. Dies ist durch den Emulator nur in positive Richtung möglich. Auch die Anzahl Flipperschläge pro Spiel wurde festgehalten.

3.2.3. Trainierte Netzwerke

Vor dem Trainieren der Agents wurde [6] konsultiert. Der gewählte Versuchsaufbau sollte bezüglich der gewählten Parameter, wie auch dem Ablauf, so ähnlich als möglich an die Experimente von Google DeepMind anknüpfen. Allerdings konnten leider nicht alle Parameter übernommen werden. So wurden für den Versuchsaufbau statt 50 Mio. Training Steps lediglich 10 Mio. verwendet. Dies war durch die limitierte zur Verfügung stehenden Rechenpower und die Zeitbegrenzung der Arbeit bedingt. Zudem wurde der ϵ-greedy Value linear von 1.0 nach 9'500'000 Training Steps auf 0.0 reduziert. Die Wahrscheinlichkeit des Ausführens einer zufälligen Aktion nach 9'500'000 Schritten wurde damit gänzlich unterbunden. Sprich wurde nach 9'500'000 Training Steps die Aktion aus dem Netz gewählt, welche nach der optimalen Policy ausgeführt werden soll. Eine Simulation des Spielens sollte damit erzielt

werden. Des Weiteren wurde ein Momentum vernachlässigt. Diese werden meist verwendet, um lokale Optima zu umgehen und globale aufzufinden[41]. Allerdings sind lokale Optima laut[42] selten enorm schlechter als globale, was der Beweggrund zur Vernachlässigung darstellt.

In der folgenden Tabelle 3.1 sind die gewählten Parameter aufgelistet:

Parameter	Wert
Screen width	84
Screen height	84
Memory size	1'000'000
Number of most recent frames experiences by the agent	4
Discount factor	0.99
Learning rate	0.00025
Batch size	32
Decay rate for RMSPropagation	0.95
Initial value of e-greedy exploration	1.0
Final value of e in e-greedy exploration	0.0
The number of steps over which the initial value of e is linearly annealed to its final	9'500'000
Batch accumulator	mean
Copy main network to target network after this many steps	10'000
Number of training steps	10'000'000
Clip error term in update between this number and its negative	1.0
The number of actions selected between successive SGD updates	4
A uniform random policy is run for this number of frames before learning starts and the resulting experience is used to populate the replay memory	200'000
Action Penalty	differs in its value

Tabelle 3.1.: Auflistung der verwendeten Parameter in den Experimenten

Eine grundlegende Änderung zur Referenzimplementierung nebst umfangreichen Debuggings stellt der Einbau eines Bestrafungsparameters für das Flippern dar. Anderenfalls kann der Agent zu jedem Zeitpunkt einen Flipperschlag auslösen, wodurch intelligentes Spielen nicht mehr ersichtlich sein würde. Dafür muss in einem ersten Schritt ermittelt werden, welche Aktionen einen Flipperschlag bewirken. Dies wird erreicht, indem dem Agent mitgeteilt wird, dass in jeder Situation immer dieselbe Aktion $a \in A$ ausgeführt werden soll. Die Aufzeichnung des Spiels (siehe Kapitel 3.1.1) zeigt auf, was die gewählte Aktion ausführt.

Beim Trainieren des Netzwerks soll nach dem Erhalt des Rewards aus dem Environment beim Schlagen eines Flippers ein Penalty vom Reward abgezogen werden. Dieser aktualisierte Reward wird ins Memory gespeichert (siehe Kapitel 3.1.3). Durch die Gewichtanpassung (Kapitel 3.1.4) wird dem Agent der reduzierte Action State Value mitgeteilt. Die Aufgabe der Successmaximierung führt dazu, dass diese Aktion nach dem Übertragen des Main auf das Target Network bei einem hohen Penalty nicht mehr ausgeführt wird. Dabei soll die Höhe des Penaltys die Performance möglichst nicht beeinträchtigen. Die Findung eines optimalen Bestrafungsparameters wird deshalb einzig durch das Trainieren mehrerer Netzwerke und deren Vergleich erreicht. Diese unterschiedlichen Netze sollen unter anderem auf durchschnittlich erreichte Punktzahl und mittlere Anzahl Flipperschläge pro Spiel untersucht werden.

Dafür sollen erst zwei Netzwerke trainiert werden, eines mit einem Action Penalty von 0.0 und ein weiteres mit einem Bestrafungsparemeter von 1.0. Es wird angenommen, dass ein Action Penalty von 0.0 keine Beeinträchtigung in der Anzahl Schläge festzustellen ist, ungehindertes Flippern und Auslösen möglichst vieler Aktionen, welche eine Einfluss auf das Spiel haben, wird die Folge sein. Bei einem Penalty von 1.0 sollte das Gegenteil resultieren. Der Agent wird merken, dass auf jede Ausführung

eines Flipperschlages eine im Vergleich zum erreichten Reward hohe Bestrafung folgt. Dadurch sollte der Agent die Aktionen zum Betätigen eines Flippers unterlassen. Falls diese beiden Annahmen sich bestätigen lassen und die Trainingsanalyse vielversprechend aussieht, ist verifiziert, dass der optimale Action Penalty auf dem Intervall $[1, 0]$ liegt. Weitere Netzwerke werden trainiert und der Action Penalty iterativ angepasst, um einen optimalen Bestrafungsparameter aufzufinden.

3.3. Verwendete Software

Für diese Arbeit wurden die unten aufgeführten Programme eingesetzt.

Arbeitsumgebung

- Windows 10
- Ubuntu 16.04 LTS
- Ubuntu 14.04 LTS

Entwicklungsumgebung

- Sublime 3
- Eclipse Neon
- PyDev 5.2
- PyCharm

Libraries

- virtualenv
- Python 2.7
- Tensorflow r0.11
- OpenAI Gym
  - python-numpy
  - python-dev
  - cmake
  - zlib1g-dev
  - libjpeg-dev
  - xvfb
  - libav-tools
  - xorg-dev
  - python-opengl
  - libboost-all-dev
  - libsdl2-dev
  - swig

- openCV2
- CUDA 8.0
- cuDNN 5.1

Auswertung

- R

Dokumentation

- MiKTeX 2.9.6161 mit TeXstudio 2.11.2
- Sumatra PDF

4. Resultate

Im folgenden werden die Resultate des Versuchsaufbaus aus Kapitel 3.2 beschrieben. Die Werte werden jeweils auf 2 Dezimalstellen genau gerundet.

4.1. Menschlicher Spieler

Im Schnitt über 50 Spiele wurden 47.30 Punkte erzielt, bei durchschnittlichen 5.60 Flipperschläge pro Spiel. In einem realen Flipperkasten zieht die Schwerkraft den Ball in Richtung der beiden Flipper, dies ist in diesem Spiel nur geringfügig der Fall. Physikalisch korrekte Flugbahnen vermisst die Anwendung auch. Da die Kugel so nur selten in die Nähe der Flipper kommt, lässt sich die tiefe durchschnittliche Anzahl Flipperschläge erklären.

Bei einem hohen Punktstand werden intuitiv relativ viele Aktionen ausgeführt. Diese Annahme wird mit einem hohen positiven Korrelationsfaktor zwischen ausgelösten Aktionen und erspieltem Punktestand von 0.90 impliziert. Das in Abbildung 4.1 dargestellte lineare Regressionsmodell ist mit einem p-Wert von $2.0 * 10^{-16}$ hochsignifikant. Pro Flipperschlag werden durchschnittlich 8.01 Punkte erzielt. Der lineare Trend ist mit roter Farbe gekennzeichnet.

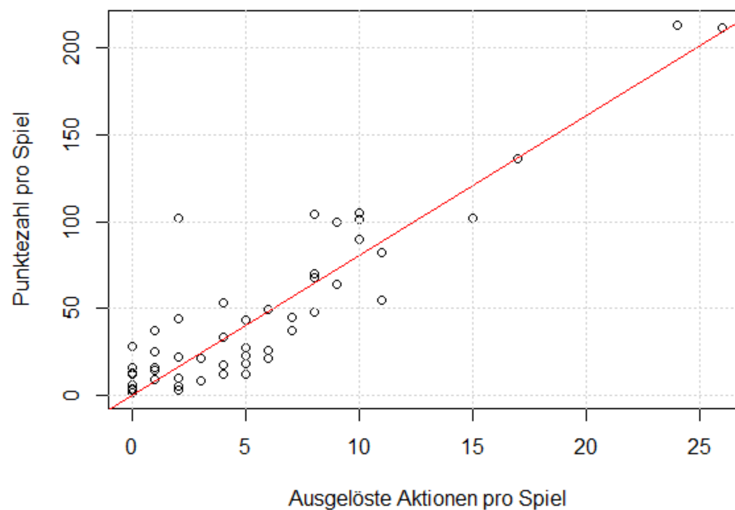

Abbildung 4.1.: Menschlicher Spieler

4.2. Random Player

Im Schnitt wurden beim Random Player 43.80 Punkte erspielt, was lediglich geringfügig weniger ist, als ein menschlicher Spieler erreicht. Dies erklärt auch die hohe Performance in DeepMinds Paper [6], welche mit der Formel

$$Performance = 100 * (DQNscore - randomscore)/(humanscore - randomscore) \tag{4.1}$$

berechnet wurde. Die Behauptung, dass nur minim Einfluss auf das Spiel genommen werden kann, wurde somit bestätigt. Absehbar war allerdings, dass der Random Player mit einem Durchschnitt von 61.46 Schlägen deutlich mehr flippert als ein Mensch. Pro Flipperaktion werden nur 0.68 Punkte erreicht. Der Random Player analysiert, wie bereits erwähnt, nicht das Spielgeschehen, sondern führt zufällig Aktionen aus. Nicht ganz überraschend weist der Datensatz einen Korrelationsfaktor von 0.89 zwischen der erreichten Punktzahl und der Anzahl ausgeführter Schläge pro Spiel auf. Eine höhere Punktzahl bedeutet meist eine längere Spielzeit, wodurch der Random Player die Möglichkeit hat, mehr Schläge auszuführen. Im aufgezeichneten Video ist unüberraschend kein intelligentes Spielen auszumachen.

Durch mehrmaliges Trainieren des Netzes hat sich herausgestellt, dass ein Spiel im Mittel 1'099.2 Training Steps dauert. Das sind die Anzahl Frames zwischen dem Abspielen der Kugeln, bis sie einmal herunterfällt, entstehen. Mit einer Batchgrösse von 32 werden pro Minibatch $\frac{32}{1'099.20} \approx 3\%$ der Frames eines Spiels wiederverwendet. Also gelten $61.46 * 0.03 = 1.79$ Schläge pro Batch im Folgenden als Benchmark für nervöses Spielen. Nicht nervöses Spielen wird anhand vom Mittelwert der Anzahl Flipperschläge pro Batchgrösse zwischen 0 und 1.79 definiert.

4.3. Trainierter Agent

Im Folgenden werden die Resultate des Versuchsaufbaus von Kapitel 3.2.3 erläutert.

4.3.1. Spezifikation der Aktionen

Mit der Implementierung einer konstanten Aktion für jeden Actionwert wurde ersichtlich, welche Aktionen was bewirken. Dies ist in der Tabelle 4.1 dargestellt.

a	ausgeführte Aktion
0	keine Aktion
1	keine Aktion
2	beide Flipper werden gedrückt
3	rechter Flipper wird gedrückt
4	linker Flipper wird gedrückt
5	Der Hebel wird gezogen und das Spiel gestartet
6	zusätzliche Löcher werden geöffnet
7	Ball bleibt hängen
8	keine Aktion

Tabelle 4.1.: Auflistung der verschiedenen Aktionen

Die grosse Anzahl an Aktionen, welche keine Auswirkung auf das Spiel haben, sind auffallend. Dies liegt daran, dass jede Aktion eine Kombination der Auslösung des Atarijoysticks und Firebuttons ist. Die Aktionen 2, 3 und 4 lösen einen Flipperschlag aus und werden im weiteren Verlauf der Arbeit als Flipperaktionen deklariert.

Der Nutzen der Aktion 7 sollte wohle das Anheben des Kastens simulieren. Durch die Ausführung wird die horizontale Bewegungskomponente des Balles verlangsamt und der Ball bewegt sich nach einiger Zeit nur noch vertikal. Selbst durch die geringe Schwerkraft im Spiel bleibt der Ball nach einer gewissen Zeit stehen und fällt nicht zwischen den beiden Flippern durch, da keine Schrägen, sondern lediglich horizontale und vertikale Spielfeldumrandungen auszumachen sind. Um am realen Flipperkasten zu spielen, wird diese Aktion nicht benötigt. Die Aktionen 5 und 6 sind in den Abbildungen 4.2 und 4.3 mittels roter Kennzeichnung grafisch dargestellt. Die Aktion 5 kann lediglich ausgeführt werden, wenn der Ball sich im Startzustand befindet. Ein Hebel wird gezogen und je nach Dauer der Ausführung wird der Ball mit einer gewissen Geschwindigkeit in das Spiel befördert. Die Aktion 6 öffnet weitere Möglichkeiten des Herunterfallens des Balles. Der Sinn dieser Aktion ist nicht bekannt.

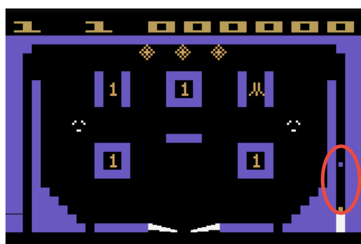

Abbildung 4.2.: Ausführung der Aktion 5

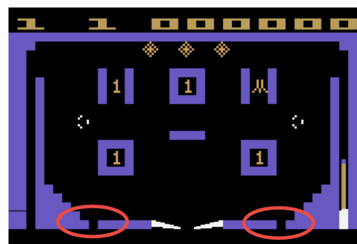

Abbildung 4.3.: Ausführung der Aktion 6

4.3.2. Ermittlung des optimalen Action Penalties

In den nachfolgenden Diagrammen sind aus Übersichtlichkeitsgründen jeweils 100 Datenpunkte gemittelt. Erst wurden Netzwerke mit einem Action Penalty von 0.0 und 1.0 trainiert. Das Trainieren eines Netzwerk dauerte mit Unterstützung der GPU ungefähr 14 Stunden, wobei jeweils 10'000'000 Training Steps durchgeführt wurden. Dies zeigte einen ersten Trend an. Im Paper Human-level control through deep reinforcement learning[6] ist jedoch eine Anzahl Training Steps von 50'000'000 angegeben. Der Average Batch Loss sinkt mit zunehmendem Training Steps und approximiert den Wert 0. Dies lässt laut[43] auf einen gut gewählte Learningrate schliessen. Zudem zeigt sich, dass die Minimierung des Average Batch Loss erfüllt wird. Wie in Kapitel 2.3 beschrieben, impliziert der Loss, wie schlecht ein bestimmtes Modell sich nach einer Iteration der Optimierung verhält. Der Average Batch Loss mit einem Action Penalty von 0.0 bzw. 1.0 ist in der Abbildungen 4.4 dargestellt.

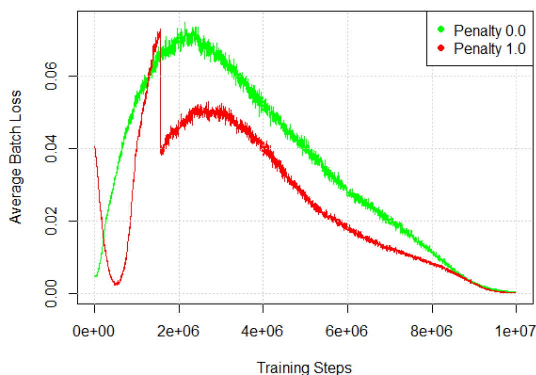

Abbildung 4.4.: Average Batch Loss bei einem Action Penalty von 0.0 und 1.0

Danach wurden die Anzahl ausgeführte Aktionen untersucht, welche einen Flipperschlag auslösen (siehe Abbildung 4.6). Durch die Batchgrösse waren 32 Aktionen pro Minibatch möglich. Theoretisch werden bei einer zufälligen Aktionsauswahl (Action Penalty $= 0.0$) der 9 unterschiedlichen Aktionen durchschnittlich $\frac{3}{9} * 32 = 10.67$ Flipperaktionen ausgeführt. Dies stimmt in etwa mit der Mittlung über 10'000'000 Training Steps überein (11.4). Mit zunehmender Anzahl Training Steps und einem abnehmendem ϵ-greedy Value streuen die Anzahl Flipperaktionen immer mehr um den wahren Wert. Dies lässt sich dadurch erklären, dass in Zeitpunkten, in denen der Ball näher bei den Flippern ist, die Aktionen 2, 3 und 4 öfters ausgelöst werden und umgekehrt. Intelligentes Flippern könnte vorhanden sein, wegen der vielen Aktionen herrscht allerdings noch grosse Nervosität. Gegensätzlich dazu werden mit einem Action Penalty von 1.0 jegliche Flipperaktionen unterbunden. Die Abnahme der Anzahl Flipperschläge wird durch die Abnahme des ϵ-greedy Values verursacht. Mit zunehmender Anzahl Training Steps wird die Randomness vermindert und vermehrt die beste Aktion aus dem Netz gewählt, welche nicht 2, 3 oder 4 ist. Nach 9'500'000 Training Steps flippert der Agent gar nicht mehr. Dies, weil bei jeder Ausführung der zu bestrafenden Aktionen ein möglicher maximaler Reward von 1.0 durch die Action Penalty auf 0.0 reduziert und ins Memory gespeichert wird. Durch die Gewichtanpassung merkt der Agent, dass auf die Ausführung der Aktionen 2, 3 und 4 nie ein positiver Reward folgt, also der Action State Value zu tief ist. Da der ϵ-greedy Value stets bei 1.0 startet, lässt sich vermuten, dass beide Netzwerke bei der gleichen Anzahl Flipperaktionen starten. Dies durch in der Abbildung 4.6 bestätigt. Zu beachten ist, dass durch die zunehmende Abnahme der Randomness über die Zeit die Gefahr von Overfitting besteht, da weiterhin eine Gewichtanpassung des Netzes erfolgt.

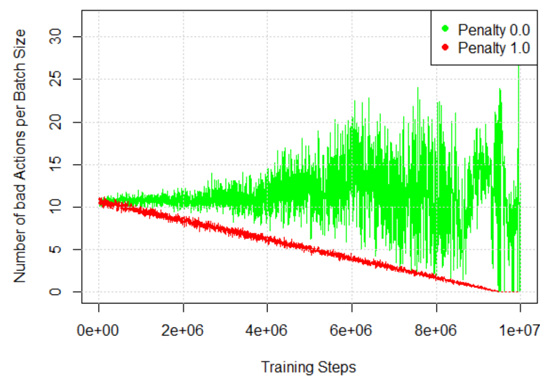

Abbildung 4.5.: Anzahl der Flipperaktionen pro Batch Size bei einem Action Penalty von 0.0 und 1.0

Der Average Batch Loss weist bei beiden trainierten Netzwerken auf erfolgreiches Training hin. Durch die unterschiedliche Bestrafung schlägt der Agent entweder beinahe immer oder nie. Dadurch wird ein optimaler Action Penalty auf dem Intervall $(1,0)$ impliziert. In einem nächsten Schritt werden daher Netzwerke mit einem Action Penalty auf genanntem Intervall ceteris paribus analysiert.

Es stellte sich heraus, dass ein optimaler Action Penalty zwischen 0.01 und 0.1 liegen muss. Beide Netzwerke zeigen eine Reduktion des Average Batch Loss über die Zeit an. Bei einem Penalty von 0.1 werden nach 9'500'000 Training Steps keine Flipperaktionen mehr ausgeführt, wie in Abbildung 4.6 ersichtlich ist. Der Penalty von 0.01 bewirkt ein durchschnittliches Flippern von 4.18 Schlägen pro Batch Size. Dies ist noch immer als nervös zu bezeichnen. Eine erste Eingrenzung des optimalen Parameters wurde allerdings erreicht. Pro Spiel sind dies zusätzliche $\frac{4.18}{0.03} = 139.3$ Flipperaktionen mehr als beim Random Player.

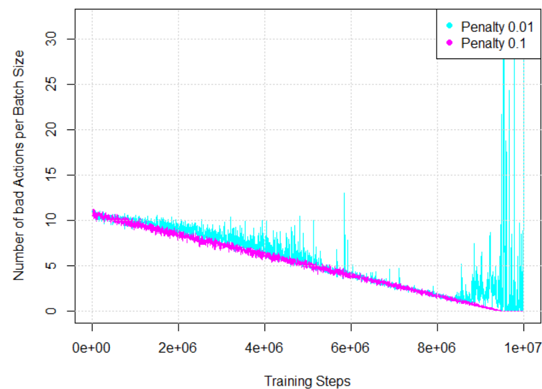

Abbildung 4.6.: Anzahl der Flipperaktionen pro Batch Size bei einem Action Penalty von 0.01 und 0.1

Die Analyse der beiden Neural Networks zeigt eine schlechte Performance. Im Schnitt wird mit einem Action Penalty von 0.1 8.4 Punkte pro Spiel erzielt, bei einem Bestrafungsparameter von 0.01 werden im Schnitt 18.2 Punkte erreicht. Die Punktzahl pro Schlag lässt sich bei einem Penalty von 0.1 nicht berechnen, da der Nenner 0 und der Wert daher nicht definiert ist. Bei einem Bestrafungsparameter von 0.01 werden im Mittel mit jeder ausgelösten Flipperaktion pro Spiel lediglich $\frac{18.2}{143.6} = 0.13$ Punkte erreicht. Der Random Agent erreicht bei der Punktzahl pro Flipperaktion einen um 5.23 besseren Wert. Der Mensch ist gar 61.62-mal besser.

4.3.3. Profiling

Um Engpässe in der Applikation zu eruieren, wurde das Programm an verschieden Orten vermessen. Eine Funktion, die für Optimierungen vielversprechend aussah, war das Umwandeln des Bildes in Graustufen und reduzieren auf 84x84 Pixel, wie dies in DeepMinds Paper vorgeschlagen ist [6]. In mehreren Repositories die Algorithmen für OpenAI Gym implementieren wurden vorwiegend folgende 3 Funktionen zur Bildumwandlung gefunden:

OpenCV2

```
cv2.resize(cv2.cvtColor(observation, cv2.COLOR_RGB2GRAY), (84, 84))
```

SciPy

```
y = 0.2126 * observation[:, :, 0] + 0.7152 * observation[:, :, 1] +
    0.0722 * observation[:, :, 2]
y = y.astype(np.uint8)
y_screen = imresize(y, [84, 84])
```

scikit-image

```
resize(rgb2gray(observation), (84, 84))
```

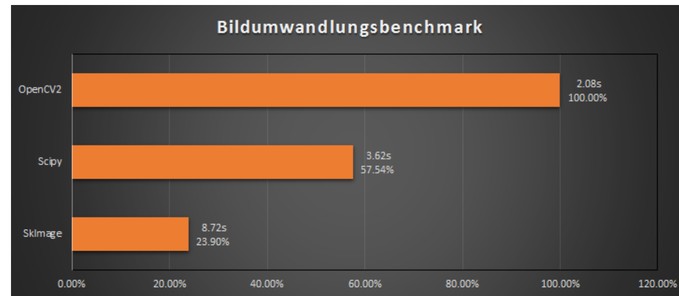

Abbildung 4.7.: Bildumwandlungsbenchmark

Für die oben genannten Funktionen wurde die Zeit gemessen, die für die Umwandlung 10'000 Frames aus der OpenAI Gym Environment 2.5 benötigt wurden. Die Resultate lassen sich untereinander vergleichen, sind aber nicht aussagekräftig für Laufzeiten in der realen Umgebung. Das Testbild wird gecached, währenddessen im realen Szenario ein Stream von Bildern verarbeitet wird. Des Weiteren wurde gemessen, wie lange OpenAI Gym braucht, um eine Aktion im Atariemulator, der das Bild mit 60 Hz [6] ausgibt, auszuwerten und den neuen Bildschirminhalt anzuzeigen. Im Durchschnitt über 20 Stichproben wir pro Frame 0.733 ms benötigt. Dies entspricht einer beschleunigten Simulation um den Faktor 22.7. Wenn die Zeit für das Trainieren und Auswerten des Netzwerkes dazu verglichen wird, kommt man dabei lediglich noch auf einen Beschleunigungsfaktor von 3.3. Durchschnittlich lief der Rechner mit laut top mit 100 % CPU und belegt dabei 1200 Mb Arbeitsspeicher. Nvidia-smi rapportierte dazu eine 35-%-GPU-Auslastung.

5. Diskussion der Resultate

Das Atarispiel VideoPinball ist zur Simulation eines realen Flipperkastens nicht geeignet. Im Spiel ist die natürliche Schwerkraft, die auf die Kugel wirkt, zu gering. Die Kugel bewegt sich auf dem Schirm mehr oder weniger nach dem Einfallswinkel gleich Ausfallswinkelprinzip. Leider fällt der Ball selten in Richtung der beiden Flipper und springt im Spielfeld herum. Zudem bewegt er sich mit nahezu konstanter Geschwindigkeit und wird teilweise unbegründet beim Aufschlag am Spielfeldrand beschleunigt bzw. verlangsamt. Physikalische Gesetze, die auf der Erde existieren, kennt das Spiel nicht wirklich. Mit den Flippertasten kann nur wenig Einfluss auf das Spielgeschehen genommen werden, weshalb der Random Agent ähnlich gut abschneidet wie ein menschlicher Spieler.

Der Score verändert sich je nach getroffenem Objekt um +100 oder +1. Dieser Reward wird auf die den Bereich -1 bis 1 zugeschnitten, was zur Folge hat, das die unterschiedlich bewerteten Objekte als gleich gut eingestuft werden. Ursprüngliche Idee des Rewardclipping ist, dass unterschiedliche Atarispiele mit demselben Parametern trainiert werden können. Das Clipping führt daher zu einer Generalisierung der Funktion für die 48 Atarispiele. Dies verhindert die Entstehung einer Spieltaktik, bei der bestimmte Objekte bevorzugt getroffen werden. Der Agent soll für unnötige Schläge mit einem absoluten Wert bestraft werden. Ein Action Penalty über 1 würde dem Agent mitteilen, dass eine Flipperaktion lediglich beim Treffen eines 100er-Objektes in näherer Zukunft sinnvoll ist. Die 1er-Belohnung würde vernachlässigt. Bei einer Bestrafung von unter 1 wird der Agent die Penaltisierung beinahe vernachlässigen bei der Chance auf eine zukünftige 100er-Belohnung. Nervöses Flippern wird nicht verhindert. Zudem ist das Abschneiden des Rewards Grund dafür, dass keine Spieltaktik ersichtlich ist. Es besteht die Möglichkeit, dass durch das Rewardclippung viele Aktionen einen Action Value von 1.0 aufweisen und dadurch eine eindeutige Entscheidung für eine Aktion nicht möglich ist. In einer nächsten Arbeit muss untersucht werden, wie die Nervosität mit den unterschiedlich hohen Belohnungen zu vereinbaren ist und wie sich ein ungeclippter Reward insgesamt auswirkt.

Oftmals bleibt der Ball beim Flippern lange im Spielfeld und Punkte werden gesammelt. Speziell beim VideoPinball von Atari fällt der Ball selten runter und beendet das Spiel. Des Weiteren gilt das Atarispiel als beendet, wenn 3 Bälle gespielt wurden. Nur jedes dritte Mal wird dem Agent mitgeteilt, wenn der Ball fertig gespielt ist. Es wird empfohlen, im Quellcode einen negativen Reward bei jedem Runterfallen des Balles zu implementieren. Es wird vermutet, dass durch eine unmittelbare Bestrafung jeden Balles ein schnelleres Trainieren ermöglicht wird.

Eine optimale Bestrafung der Flipperschläge wurde nicht gefunden, dies würde weitere Versuche benötigen. Dennoch wird gezeigt, dass nervöses Flippern mithilfe eines Action Penalty unterbunden werden könnte. Die schlechten Ergebnisse der Performance resultieren möglicherweise aufgrund Overfittings. Nach 9'500'000 Trainingsschritten wurde die Randomness gänzlich beseitigt, wodurch Overfitting eine starke Gefahr darstellte und mit grosser Wahrscheinlichkeit auch eintrat. Allerdings flippert der Agent trotz Penalty mehr als der Random Agent. Dies impliziert, dass der Agent gemerkt hat, dass die Flipperaktionen die einzigen Aktionen darstellen, die einen Einfluss auf das Spiel und positive Auswirkung auf den Score haben. Ansonsten würde der Agent aufgrund der Bestrafung eine ähnliche oder tiefere Anzahl Flipperaktionen ausführen wie der Random Player. Eine weitere Fehlerquelle stellt allerdings die vergleichsweise tiefe Anzahl Trainingsschritte dar. Die von Google DeepMind gewählten 50 Mio. Trainingsschritte waren in dieser Arbeit trotz Berechnung auf der Grafikkarte zeitlich unmöglich [6]. Die 10 Mio. Training Steps versprachen aufgrund abnehmendem Loss gute Ergebnisse, hätten im Nachhinein jedoch länger trainiert werden müssen. Dadurch könnte eine ähnliche Performance des Agents wie der Mensch möglich sein.

Die Trainingszeit wird laut dem offiziellen Google DeepMind Paper [6] auf 38 Tage geschätzt bei Verwendung derselben Parameter. Diese Trainingszeit ist extrem hoch und wird mit einem realen Flipperkasten

nicht realisierbar sein. Es wird empfohlen, andere Implementierungen oder gar gänzlich andere Algorithmen und andere Techniken für das Training zu verwenden. Ein anderer Emulator scheint zudem sinnvoll. Auch könnte untersucht werden, wie des Replay Memories benutzt werden kann, um die Trainingszeit zu verringern. Anderenfalls scheint der erfolgreiche Aufbau eines Pinball Wizards nicht realisierbar.

Der von Siraj Raval zur Verfügung stehende Code weist viele Fehler auf und weicht in vieler Hinsicht vom offiziellen Google DeepMind Paper [6] ab. Einerseits wird beim Ausführen einer Gewichtsanpassung bei der Implementierung von Siraj Raval das maximale Q-Target falsch berechnet. Durch einen subtilen und dennoch entscheidenden Fehler wird das maximale Argument als Q-Target genommen und nicht das Argument mit dem maximalen Wert. Auch wird bei Siraj im letzten Convolution Layer eine Filtersize von 4 gewählt, bei Google 3. Der Einbau eines Momentums wurde nicht verwendet, könnte die Performance in einem weiteren Versuch steigern, indem lokale Optima umgangen werden. Des Weiteren wurden Fehler in der ϵ-greedy Value Funktion gefunden. Dieser Wert nahm korrekterweise linear ab, wurde aber nach dem Erreichen von 1'000'000 Training Steps wird der ϵ-greedy Value gleich dem aktuellen Training Step gesetzt. Google DeepMind setzt korrekterweise einen finalen ϵ-greedy Value ein, welcher 0.1 ist. Ansonsten wird die Randomness mit zunehmendem Training vermindert und ab 1'000'000 Steps wieder auf 100 % gesetzt. Des Weiteren wird die nach dem Training die angepasste Session verworfen. D. h., nach dem Trainieren wurde das gesamte Netz verworfen und mit einem untrainierten Netz gespielt. Dieses untrainierte Netzwerk kam allerdings nie zum Einsatz, da im Quellcode von Siraj Raval, wo der Agent eigentlich mit seinem trainierten Netz spielen sollte, der ϵ-greedy Value konstant den Wert 1 statt 0 besitzt. So würde, auch wenn die Session beibehalten würde, die beste Aktion des Netzes nie ausgeführt, da stets eine zufällige Aktion ausgewählt werden wird. Zudem werden bei Google ausgewählte Aktionen über 6 Frames gespielt, bei Siraj wird in jedem Schritt das Netz ausgewertet und eine Aktion ausgeführt, was die Trainingszeit verlängert. Laut Google wird die Performance beim Ausführen der Aktion über 6 Frames statt einem um lediglich 5 % verschlechtert. Diese 10 Hz sind ungefähr die obere Grenze, in welcher ein Mensch eine Aktion ausführen kann.

Zusammenfassend wird nicht empfohlen, den Pinball Wizard auf [6] aufzubauen, da die Trainingszeit auf einem realen Flipperkasten zu lange dauern würde. Zudem wurden seit der Publikation des Papers neue Techniken und Netzwerkarchitekturen, welche vielversprechende Ansätze bereithalten, veröffentlicht.

6. Ausblick

Der momentane Aufbau des Flipperkastens ist nicht für den Pinball Wizard geeignet. Die Eventkamera spezialisiert sich auf das Tracking des Balles. Bewegungen werden auf einem schwarz-weiss Video aufgezeichnet. Der gesamte Aufbau der Hindernisse auf dem Spielfeld wird dabei allerdings vernachlässigt. Falls der Spielaufbau sich nicht bewegt bzw. keine Leuchteffekte von sich gibt, kann die Eventkamera diesen nicht identifizieren. Dadurch wird dem Agent nicht das genaue Abbild des Flipperkastens mitgeteilt und das Lernen erschwert. Es wird geraten, eine reguläre Kamera zu verwenden. Diese sollte trotz 30 oder 60 Frames/s ausreichen. Wie in DeepMinds Paper[6] beschrieben verschlechtert sich die Performance des DQN-Algorithmus nur minimal, wenn nur jedes 6. Frame einer 60 Hz Bildschirmausgabe verwendet wird. Eine klare Kennzeichnung der Hindernisse im Gegensatz zum Rollfeld des Balles würde das Lernen bei sich ändernden Lichtverhältnissen begünstigen. Dies könnte umgesetzt werden, indem alle Hindernisse eine schwarze Oberfläche aufweisen (beispielsweise mit zugeschnittenem schwarzem Klebeband) und diejenigen Orte, an denen der Ball nicht anstösst, sondern sich bewegen kann, mit Weiss kennzeichnen. Einerseits wird das Bild beim in der Arbeit gewählten Algorithmus auf Greyscale gebrochen, dies könnte durch ähnliche Farbtöne der Hindernisse und dem Rollfeld des Balles zum Problem werden. Eine klare Unterscheidung durch starke Kontraste könnte dies verhindern.

Seit der Erscheinung des Papers Human-level control through deep reinforcement learning von Google DeepMind[6] sind weitere Artikel zum Thema RL in Kombination mit Atarispielen veröffentlicht worden. Es wurden neue Netzarchitekturen wie z. B. das Double Q-Network[44] und Duelling Network Architekturen[45] entwickelt, welche ihre Agents besser trainieren. Es wird geraten, weitere Recherche über diese Architekturen durchzuführen.

Da das lange Flippern auf dem realen Kasten am Zustand der Maschine nagt, ist ein Training mit möglichst wenig Trainingszeit wünschenswert. Das Paper Prioritized Experience Replay[46] beschreibt hierzu, wie das Replay Memory effizienter genutzt werden kann, um Netze schneller zu trainieren. Die in diesem Paper beschriebenen Methoden mit proportional und rank-based Priorization sind jedoch nicht in jedem Fall gewinnbringend. In Video Pinball hat die rank-based Methode schlechter abgeschnitten als das Standard DQN. In 41 von 49 Atarispielen wirkten beide Methoden verbessernd.

Eine weitere Möglichkeit zum schnelleren Trainieren des Netzes ist die Asynchronous Advanced Actor-critic (A3C) Methode[47]. Hierbei werden mehrere Instanzen des Environments parallel ausgeführt. Dies ermöglicht den Verzicht des Experiance Replay Memories und lässt sich auf einer Standard Multicore CPU effizienter trainieren als ein DQN auf der GPU. Diese Methode ist jedoch nur auf Simulation auf dem Computer anwendbar, da mehrere Instanzen parallel ausgeführt werden müssen. Die Trainingszeit verringert sich hierbei von 8 Tagen auf der GPU 4 Tagen auf der CPU, während zusätzlich in den Spielen ein höherer Score erspielt wurde.

Für weitere Tests wird empfohlen, auf eine andere Codebasis umzusteigen. Das Framework von Kim Tae-Hoon[48] scheint hierbei vielversprechender, wobei ein Trainingsversuch eines DQN auf der CPU 14 Tage dauerte. Dafür stehen verschiedene Agents zur Verfügung, welche die Ausblick zitierten Papers implementieren.

Da das Atarispiel VideoPinball keine optimale Lösung zur Simulation des richtigen Flipperkastens ist, wird das Programmieren einer eigenen Simulation empfohlen, die dem Flipperkasten 1.2 im Aufbau und physikalischen Verhalten möglichst ähnlich ist. Optimalerweise ist diese Simulation beschleunigbar und kann schneller als in Echtzeit abgespielt werden. Das schont den richtigen Flipperkasten und die Trainingsdauer fällt kürzer aus. Des Weiteren wäre eine Einbindung in das OpenAI Gym[35] vorteilhaft. Dies würde die Möglichkeit schaffen, dass sich andere Personen aus dieser Community dem Lernproblem einen effizienten Algorithmus zu entwickeln, der für den realen Kasten verwendet werden kann, annehmen.

Zusammenfassend wird weitere Recherche über genannte Themen empfohlen, damit der Pinball Wizard erfolgreich sich selber das Flippern beibringt.

7. Verzeichnisse

Im folgenden werden die Verzeichnisse der wissenschaftlichen Arbeit aufgelistet.

Literaturverzeichnis

[1] A. J. Champandard, “Reinforcement learning introduction,” 2001. [Online]. Available: http://reinforcementlearning.ai-depot.com/. Accessed: Dec. 12, 2016.

[2] T. Urban, “The AI Revolution: The Road to Superintelligence,” in Wait But Why, 2015. [Online]. Available: http://waitbutwhy.com/2015/01/artificial-intelligence-revolution-1.html. Accessed: Dec. 12, 2016.

[3] J. Tapson, “Google’s go victory shows AI thinking can be unpredictable, and that’s a concern,” in phsy.org, 2016. [Online]. Available: http://phys.org/news/2016-03-google-victory-ai-unpredictable.html. Accessed: Dec. 12, 2016.

[4] “DeepMind AlphaGo vs Lee Sedol,” in Go Game Guru, 2016. [Online]. Available: https://gogameguru.com/tag/deepmind-alphago-lee-sedol/. Accessed: Dec. 12, 2016.

[5] C. Shu, “Google acquires artificial intelligence startup DeepMind for more than USD 500M,” in techcrunch, TechCrunch, 2014. [Online]. Available: https://techcrunch.com/2014/01/26/google-deepmind/. Accessed: Dec. 12, 2016.

[6] V. Mnih et al., “Human-level control through deep reinforcement learning,” Nature, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[7] T. Stadelmann, “Thilo on data,” 2016. [Online]. Available: https://dublin.zhaw.ch/“stdm/?p=327. Accessed: Dec. 12, 2016.

[8] N. J. Nilsson et al. , ”Introduction to Machine Learning,“ Stanford Artificial Intelligence Laboratory, Stanford University, Nov. 3, 2016. [Online Document]. Available: http://ai.stanford.edu/”nilsson/MLBOOK.pdf. Accessed: Dec. 12, 2016.

[9] E. Alpaydin, Introduction to machine learning, 2nd ed. Cambridge, MA: MIT Press, 2010.

[10] R. S. Michalski, J. G. Carbonell, T. M. Mitchell, and P. IVONNE, Machine learning an artificial intelligence approach. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983.

[11] T. van der Laan, “Machine learning: The bigger picture, part I - DZone big data,” in DZone / Big Data Zone, dzone.com, 2016. [Online]. Available: https://dzone.com/articles/machine-learning-the-bigger-picture-part-i. Accessed: Dec. 12, 2016.

[12] B. Marr, “A short history of machine learning every manager should read,” in Forbes, Forbes, 2016. [Online]. Available: http://www.forbes.com/sites/bernardmarr/2016/02/19/a-short-history-of-machine-learning-every-manager-should-read/#5cb0717e323f. Accessed: Dec. 12, 2016.

[13] A. Lycett, “History - enigma (pictures, video, facts and news),” in BBC, 2010. [Online]. Available: http://www.bbc.co.uk/history/topics/enigma. Accessed: Dec. 12, 2016.

[14] Priyadharshini, “Machine learning: What it is and why it matters?,” in simplilearn, Simplilearn.com, 2015. [Online]. Available: https://www.simplilearn.com/what-is-machine-learning-and-why-it-matters-article. Accessed: Dec. 12, 2016.

[15] J. Davies, “What is the promise of big data? Computer will be better than humans,” in businesscloudnews, 2016. [Online]. Available: http://www.businesscloudnews.com/2016/06/06/what-is-the-promise-of-big-data-computer-will-be-better-than-humans/. Accessed: Dec. 12, 2016.

[16] X. Qi and B. D. Davison, “Web Page Classification: Features and Algorithms,” Lehigh University, Feb. 12, 2009. [Online]. Available: https://www.cs.ucf.edu/“dcm/Teaching/COT4810-Fall%202012/Literature/WebPageClassification.pdf. Accessed: Dec. 12, 2016.

[17] B. Ifrach, "How Airbnb uses machine learning to detect host preferences," in airbnb, 2015. [Online]. Available: http://nerds.airbnb.com/host-preferences/. Accessed: Dec. 12, 2016.

[18] R. Kochar, "Is this the revival of artificial technology?," in Entrepreneut India, 2016. [Online]. Available: https://www.entrepreneur.com/article/272174. Accessed: Dec. 12, 2016.

[19] O. Dürr, "Statistisches Datamining," 2016. [Online]. Available: http://oduerr.github.io/teaching/stdm/aufgaben.html. Accessed: Dec. 12, 2016.

[20] T. O. Ayodele, "Machine Learning Overview," University of Portsmouth, Feb. 1, 2010. [Online]. Available: http://cdn.intechopen.com/pdfs/10683/InTech-Machine_learning_overview.pdf. Accessed: Dec. 12, 2016.

[21] J. Higgins, "Introduction to Multiple Regression," 2005. [Online]. Available: http://www.biddle.com/documents/bcg_comp_chapter4.pdf. Accessed: Dec. 12, 2016.

[22] StatSoft, "Overfitting (Überanpassung)," in StatSoft, 2002. [Online]. Available: https://www.statsoft.de/glossary/O/Overfitting.htm. Accessed: Dec. 12, 2016.

[23] K. Borne, "Are Your Predictive Models like Broken Clocks?," in http://rocketdatascience.org/, 2016. [Online]. Available: http://rocketdatascience.org/?tag=machine-learning. Accessed: Dec. 12, 2016.

[24] J. M. Girard, "What are ways to prevent over fitting your training set data?," in Quora, 2013. [Online]. Available: https://www.quora.com/What-are-ways-to-prevent-over-fitting-your-training-set-data. Accessed: Dec. 12, 2016.

[25] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction," MIT Press, 2012. [Online]. Available: http://people.inf.elte.hu/lorincz/Files/RL_2006/SuttonBook.pdf. Accessed: Dec. 12, 2016.

[26] Two Minute Papers, "Google DeepMind's deep Q-learning playing Atari Breakout," in YouTube, YouTube, 2015. [Online]. Available: https://www.youtube.com/watch?v=V1eYniJ0Rnk. Accessed: Dec. 13, 2016.

[27] M. Martin, "Mario Martin's home page," in Departament de Ciències de la Computació, 2011. [Online]. Available: http://www.cs.upc.edu/"mmartin/Ag3-Goal%20definition%20and%20examples-4x.pdf. Accessed: Dec. 13, 2016.

[28] T. Rückstieß and et al., "Exploring Parameter Space in Reinforcement Learning," 2016.

[29] R. Füss, "materialien-statistik-2," in Empirische Wirtschaftsforschung und Ökonometrie, 2008. [Online]. Available: https://www.empiwifo.uni-freiburg.de/lehre-teaching-1/summer-term-08/materialien-statistik-2/Stat%20II%20kap%201%20Grenzwert. Accessed: Dec. 13, 2016.

[30] A. Juliani, "Simple reinforcement learning with Tensorflow part 4: Deep Q-Networks and beyond," 2016. [Online]. Available: https://medium.com/@awjuliani/simple-reinforcement-learning-with-tensorflow-part-4-deep-q-networks-and-beyond-8438a3e2b8df#.9rhz2yi8v. Accessed: Dec. 20, 2016.

[31] M. A. Nielsen, "Neural Networks and Deep Learning," 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com/chap1.html. Accessed: Dec. 18, 2016.

[32] U. Karn, "A quick introduction to neural networks," the data science blog, 2016. [Online]. Available: https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/. Accessed: Dec. 18, 2016.

[33] A. Krause, "Neural Networks," in Learning and Intelligent Systems, 2014. [Online]. Available: las.ethz.ch. Accessed: 2014

[34] U. Karn, "An intuitive explanation of Convolutional neural networks," the data science blog, 2016. [Online]. Available: https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/. Accessed: Dec. 19, 2016.

[35] "OpenAI gym: A toolkit for developing and comparing reinforcement learning algorithms,". [Online]. Available: https://gym.openai.com/. Accessed: Dec. 18, 2016

[36] Google Inc., "TensorFlow," 2016. [Online]. Available: https://www.tensorflow.org/. Accessed: Dec. 18, 2016.

[37] S. Raval, "llSourcell," in GitHub, GitHub, 2016. [Online]. Available: https://github.com/llSourcell. Accessed: Dec. 13, 2016.

[38] "Sirajology," in YouTube, YouTube. [Online]. Available: https://www.youtube.com/channel/UCWN3xxRkmTPmbKwht9FuE5A. Accessed: Dec. 13, 2016.

[39] S. Raval, "llSourcell/Game-AI," in GitHub, GitHub, 2016. [Online]. Available: https://github.com/llSourcell/Game-AI. Accessed: Dec. 13, 2016.

[40] RetroGames.cz, "Video pinball (Atari 2600) - online game," RetroGames.cz. [Online]. Available: http://www.retrogames.cz/play_047-Atari2600.php. Accessed: Dec. 13, 2016.

[41] S. Ruder, "An overview of gradient descent optimization algorithms," Sebastian Ruder, 2016. [Online]. Available: http://sebastianruder.com/optimizing-gradient-descent/index.html#momentum. Accessed: Dec. 14, 2016.

[42] D. Silver, "Deep Reinforcement Learning," 2015. [Online]. Available: http://videolectures.net/rldm2015_silver_reinforcement_learning/. Accessed: Dec. 14, 2016.

[43] "CS231n Convolutional neural networks for visual recognition,". [Online]. Available: http://cs231n.github.io/neural-networks-3/. Accessed: Dec. 17, 2016.

[44] Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," 2015. [Online]. Available: https://arxiv.org/abs/1509.06461. Accessed: Dec. 18, 2016.

[45] Z. Wang, T. Schaul, M. Hessel, Hasselt, and M. Lanctot, "Dueling network architectures for deep reinforcement learning," 2015. [Online]. Available: https://arxiv.org/abs/1511.06581. Accessed: Dec. 18, 2016.

[46] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2015. [Online]. Available: https://arxiv.org/abs/1511.05952v4. Accessed: Dec. 18, 2016.

[47] V. Mnih et al., "Asynchronous methods for deep reinforcement learning," 2016. [Online]. Available: https://arxiv.org/abs/1602.01783. Accessed: Dec. 18, 2016.

[48] K. Tae-Hoon, "Deep Reinforcement Learning in TensorFlow," GitHub, 2016. [Online]. Available: https://github.com/carpedm20/deep-rl-tensorflow. Accessed: Dec. 18, 2016.

Abbildungsverzeichnis

1.1. Menschlicher Fortschritt mit dem Schlüssel zur künstlichen Intelligenz 6
1.2. Momentaner Aufbau des Flipperkastens an der ZHAW 7

2.1. Vorhersagefehler in Abhängigkeit der Modellkomplexität 10
2.2. Einfluss der gewühlten Aktion a_t auf den vom Environment ausgegebenen Reward r_t und die Observations o_t 11
2.3. Darstellungsdiagramm für v_π und q_π 14
2.4. Grafische Darstellung eines Neurons 16
2.5. Sigmoid-, ReLu- und Tangenshyperbolicus-Aktivierungsfunktion (v.l.n.r.) 17
2.6. Graphische Darstellung eines Neural Networks 18
2.7. Convolutional Neural Network LeNet 19
2.8. Filter in Aktion 20
2.9. Darsetellung von Max Pooling 20
2.10. Data Flow Graph 22

3.1. Ablaufdiagramm des Hauptprogramms 24
3.2. Ablaufdiagramm des Netzwerktrainings 25
3.3. Ablaufdiagramm des Erstellens eines Memoryeintrages 26
3.4. Ablaufdiagramm der Gewichtanpassung im Main Network 27
3.5. Start- und Endzustand beim VideoPinball 28

4.1. Menschlicher Spieler 32
4.2. Ausführung der Aktion 5 34
4.3. Ausführung der Aktion 6 34
4.4. Average Batch Loss bei einem Action Penalty von 0.0 und 1.0 34
4.5. Anzahl der Flipperaktionen pro Batch Size bei einem Action Penalty von 0.0 und 1.0 . 35
4.6. Anzahl der Flipperaktionen pro Batch Size bei einem Action Penalty von 0.01 und 0.1 36
4.7. Bildumwandlungsbenchmark 37

A.1. Average Batch Loss bei unterschiedlicher Penaltisierung II
A.2. Graustufenbild III

Tabellenverzeichnis

3.1. Auflistung der verwendeten Parameter in den Experimenten 29
4.1. Auflistung der verschiedenen Aktionen 33
7.1. Abkürzungsverzeichnis 48

Glossar

In diesem Abschnitt werden Abkürzungen und Begriffe kurz erklärt.

bzw.	beziehungsweise
Backpropagation	Backward propagation of errors
CNN	Convolutional Neural Networks
d.h.	das heisst
i.i.d	independant identical distributed
KI	Künstliche Intelligenz
Mio.	Millionen
NN	Neural Networks
ML	Machine Learning
o.ä.	oder ähnliches
DQN	Deep Q-Network
RL	Reinforcement Learning
s.d.	so dass
vgl.	vergleiche
v.l.n.r.	von links nach rechts
ZHAW	Zürcher Hochschule für Angewandte Wissenschaften
z. B.	zum Beispiel

Tabelle 7.1.: Abkürzungsverzeichnis

A. Anhang

A.1. Installationsanleitung

Die Hardware, auf der die NN trainiert worden sind, besteht aus einem 3.2 Ghz Intel i5-6500 Prozessor, 16 Gb RAM und einer Geforce GTX 1070 Grafikkarte. Da diese nur von CUDA 8 unterstützt wird, musste TensorFlow selber kompiliert und installiert werden, weil das zur Verfügung gestellte Binary r0.11 nur CUDA 7 unterstützt hat.

```
1. JDK 8 installieren
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer

2.Bazel URI als package source hizufügen
$ echo "deb [arch=amd64] http://storage.googleapis.com/bazel-apt stable jdk1.8" |
sudo tee /etc/apt/sources.list.d/bazel.list
$ curl https://bazel.build/bazel-release.pub.gpg | sudo apt-key add -
$ sudo apt-get update && sudo apt-get install bazel

3. Auf neuere Version von Bazel umsteigen
$ sudo apt-get upgrade bazel

4. CUDA herunterladen und nach Anleitung der Seite installieren
https://developer.nvidia.com/cuda-downloads

5. CuDNN herunterladen und nahc Anlleitung der Seite installieren
https://developer.nvidia.com/cudnn

6.PATH variable setzen
$export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH

7. benötigte packete installieren und tensorflow repository klonen
$sudo apt-get install python-numpy swig python-dev python-wheel
$git clone https://github.com/tensorflow/tensorflow.git

8. in das Verzeichniss tensorflow navigieren und ./configure ausführen und folgende
Konfiguration vornehemen

$./configure
Please specify the location of python. [Default is /usr/bin/python]: [enter]
Do you wish to build TensorFlow with Google Cloud Platform support? [y/N] n
No Google Cloud Platform support will be enabled for TensorFlow
Do you wish to build TensorFlow with GPU support? [y/N] y
GPU support will be enabled for TensorFlow
Please specify which gcc nvcc should use as the host compiler.
[Default is /usr/bin/gcc]: [enter]
Please specify the Cuda SDK version you want to use, e.g. 7.0.
[Leave empty to use system default]: 8.0
```

```
Please specify the location where CUDA 8.0 toolkit is installed.
Refer to README.md for more details. [Default is /usr/local/cuda]: [enter]
Please specify the Cudnn version you want to use. [Leave empty to use system default]: 5
Please specify the location where cuDNN 5 library is installed.
Refer to README.md for more details. [Default is /usr/local/cuda]: [enter]
Please specify a list of comma-separated Cuda compute capabilities you want to build with.
You can find the compute capability of your device at: https://developer.nvidia.com/cuda-gpus.
Please note that each additional compute capability significantly increases your build time
and binary size.
[Default is: "3.5,5.2"]: 6.1
Setting up Cuda include
Setting up Cuda lib64
Setting up Cuda bin
Setting up Cuda nvvm
Setting up CUPTI include
Setting up CUPTI lib64
Configuration finished


9. Tensorflow packages kompilieren
bazel build -c opt --config=cuda //tensorflow/tools/pip_package:build_pip_package
bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/tensorflow_pkg


10. package installieren
$ sudo pip install --upgrade /tmp/tensorflow_pkg/tensorflow-0.9.0-*.whl
```

A.2. Trainierte Netzwerke

In Abbildung A.1 sind die Average Batch Loss einiger trainierter Netzwerke abgebildet.

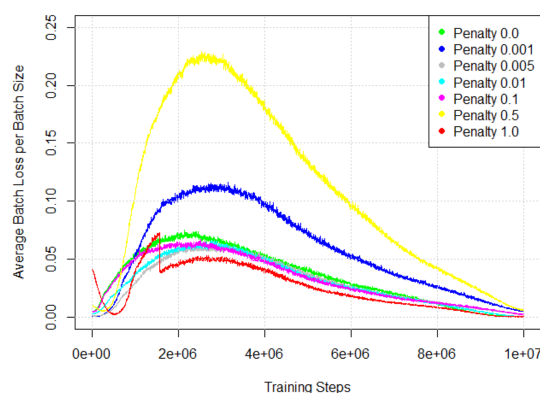

Abbildung A.1.: Average Batch Loss bei unterschiedlicher Penaltisierung

A.3. Graustufenbild

In der Abbildung A.2 ist abgebildet, wie das auf 84x84 reduzierte und auf Greyscale reduzierte Inputbild aussieht.

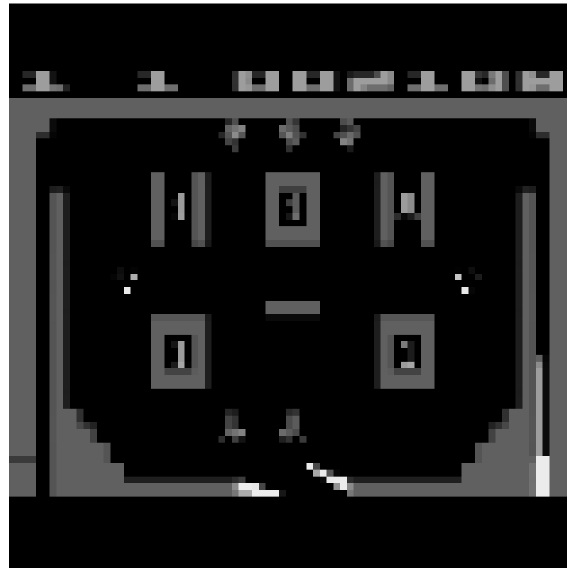

Abbildung A.2.: Graustufenbild