Schriftliche Ausarbeitung

Anwendungen von Reinforcement Learning

Seminar: Ist künstliche Intelligenz gefährlich?

Dozent

Dr. Ullrich Köthe

Autor

Florian Fallenbüchel

Matrikelnummer: 3144974
E-Mail: ffallenbuechel@aol.com

1 Einführung

Reinforcement Learning wurde bereits in den Neunzigern von Gerald Tesauro und seiner Forschungsgrupe bei IBM dazu benutzt eine künstliche Intelligenz für Backgammon zu erschaffen. Seitdem wurde Reinforcement Learning für die Lösung einer Vielzahl von Problemen benutzt und besonders mit dem Aufkommen von Neuronalen Netzen in den letzten Jahren ergeben sich ständig neue Einsatzmöglichkeiten. Im Folgenden möchte ich drei Anwendungsbeispiele näher beleuchten. Dabei handelt es sich um ein Paper von einem Team von Google DeepMind, in dem sie beschreiben wie sie mit Deep Reinforcement Learning alte Atari Spiele mit nichts weiter als Screenshots als Input spielen. Um zu zeigen wie einfach diese Techniken sind, zeige ich daraufhin wie ein einzelner Entwickler bei sich zuhause auf einer kleinen Maschine die im Paper gezeigten Methoden anwendet, um damit einen Agent Super Mario World spielen zu lassen. Des weiteren werde ich zeigen wie eine Gruppe der Stanford University mit Hilfe von Reinforcement Learning einen autonomen Helikopter komplizierte Flugmanöver ausführen lässt. Zu guter letzt erkläre ich wie erneut ein Team von Google DeepMind eine künstliche Intelligenz für das äußerst komplexe Spiel “Go” entwickelt hat. Auch hierbei hat Reinforcement Learning eine entscheidende Rolle gespielt.

2 Playing Atari with Deep Reinforcement Learning

Die meisten erfolgreichen Reinforcement Learning Anwendungen verließen sich auf handgemachte Features zusammen mit linearen Value-Functions oder Policies. Das Problem an dieser Herangehensweise ist, dass die Performanz dieser Anwendungen klar von der Qualität der Features abhängt. Durch die Fortschritte im Bereich des Deep Learning in den letzten Jahren wurde es nun aber möglich hochqualitative Features aus rohen Sensordaten zu extrahieren. Dies wollte sich ein Team von Google DeepMind zu nutze machen und somit erschufen sie das sogenannte Deep Reinforcement Learning, bei dem sie ein Convolutional Neural Network an einen klassischen Reinforcement Learning Algorithmus angebunden haben. Ihr Ziel war es, alte Atari Spiele mit nichts weiter als Pixeln als Input von einem durch Reinforcement Learning trainierten Agent spielen zu lassen. Der einzige Reward mit dem dieser Agent trainiert werden sollte, war der Score, den er im Spiel erhält. Sie wollten einen einzelnen Agent, der so viele verschiedene Spiele wie möglich spielen konnte, ohne spezifische Informationen zu den einzelnen Spielen zu erhalten. Im nächsten Abschnitt werde ich erklären, wie sie das realisiert haben.

2.1 Experience Replay, Preprocessing und Netzwerkarchitektur

Für das Generieren des Datensatzes, mit denen das Netzwerk trainiert werden sollte, entschieden sie sich für eine Technik namens Experience Replay, bei der aus einem großen Speicher zufällig Samples für das Training ausgewählt werden. Dies hat zwei entscheidende Vorteile: Zum einen werden für Reinforcement Learning aufeinanderfolgende Inputs benötigt, da ein einzelner Screenshot nicht genügend Informationen über die Auswirkungen von Aktionen und über das was auf dem Bildschirm wirklich passiert bietet. Durch Experience Replay kann eine Sequenz von Screenshots an das Netzwerk übergeben werden. Zum anderen kann durch die zufällige Auswahl aus dem Speicher schon mit einem relativ kleinen Datensatz ein komplexeres Netzwerk trainiert werden, da mit einzelnen Samples mehrfach trainiert werden kann. Weiterhin wäre das Lernen mit kontinuierlichen Screenshots ineffizient, da diese eine starke Korrelation untereinander aufweisen würden.

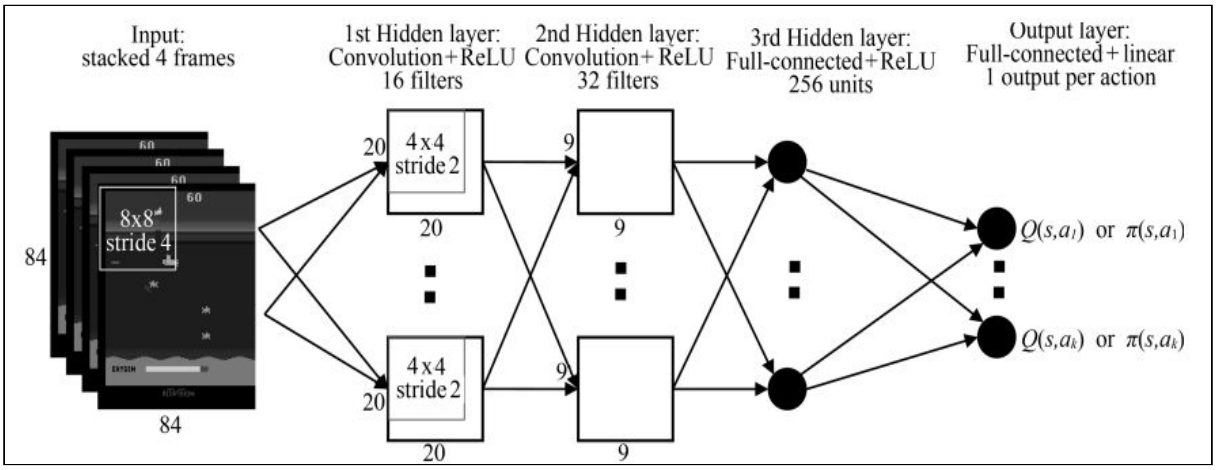

Bild 1: Architektur des Convolutional Neural Networks (CNN), welches direkt die Q-Values für die einzelnen möglichen Aktionen ausgibt. [2]

Bevor die Screenshots an das Netzwerk gefüttert werden können, müssen diese aufbereitet werden. Die rohen Screenshots haben 210x160 Pixel und 128 Farben, entsprechend aufwändig kann die Verarbeitung dieser Screenshots für das Netzwerk sein. Um die benötigte Rechenleistung zu reduzieren übergeben sie die letzten vier Frames einer Sequenz daher an eine Preprocessing Funktion $\phi(s)$. Diese macht aus den Screenshots Graustufenbilder und rechnet sie auf 110x84 Pixel herunter. Anschließend wird ein 84x84 Pixel großer Bereich, der grob den Spielbereich abdeckt, aus den Bildern herausgeschnitten und diese werden gestapelt. Der letzte Schritt des Zuschneidens wird nur benötigt, da die Architektur die sie verwendet haben nur mit quadratischen Inputs funktioniert.

Bild 1 zeigt eine grafische Darstellung des Netzwerks, welches sie für alle sieben getesteten Atari Spiele verwendet haben. Das Netzwerk bekommt einen 84x84x4 Pixel großen Input, welcher durch die Preprocessing Funktion produziert wurde. Die erste Schicht des Netzwerks wendet 16 8x8 Filter mit Stride 4 auf den Input an, gefolgt von einer Rectified Linear Unit (ReLU). Die nächste Schicht verwendet 32 4x4 Filter mit Stride 2, erneut gefolgt von einer ReLU. Die letzte Schicht ist fully-connected und besteht aus 256 ReLUs. Der Output-Layer ist erneut vollkommen zusammenhängend und hat für jede Aktion eine einzelne Ausgabe, welche zwischen 4 und 18 Aktionen für die einzelnen Spiele schwankt. Der Vorteil dieser Architektur ist, dass mit einem einzelnen forward pass durch das Netzwerk die einzelnen Q-Values berechnet werden können.

2.2 Deep Q-learning

Der Algorithmus mit dem sie das Netzwerk trainieren nennen sie Deep Q-learning, da er Q-learning des Reinforcement Learnings mit Deep Learning verknüpft. Der Algorithmus funktioniert wie folgt: Zunächst wird die Sequenz s_1 mit dem ersten Frame x_1 initialisiert und danach wird s_1 an die Preprocessing Funktion übergeben. Diese liefert damit die erste vorverarbeitete Sequenz $\boldsymbol{\phi}_1$. Nun werden bei jedem k-ten Frame (mit dieser Methode ließen sich k-fach mehr Spiele spielen, ohne die Laufzeit großartig zu vergrößern, die Aktion des letzten Schrittes wird weiter ausgeführt für die übersprungenen Frames) folgende Aktionen durchgeführt: Mit Wahrscheinlichkeit ε wird eine zufällige Aktion a_t ausgewählt, andernfalls wird die Aktion a_t ausgewählt, welche für die aktuelle vorverarbeitete Sequenz $\boldsymbol{\phi}_t$ die maximale Q-Value liefert. ε beginnt bei 1, sodass am Anfang alle Aktionen zufällig sind, und verringert sich über die ersten Millionen Frames auf 0.1. Danach wird a_t im Emulator ausgeführt und der erhaltene Reward r_t sowie der aus der Aktion resultierende Frame x_{t+1} werden gespeichert. Nun wird eine neue Sequenz erzeugt mit $s_{t+1} = s_t , a_t , x_{t+1}$. Auch diese Sequenz wird an $\boldsymbol{\phi}$ übergeben um $\boldsymbol{\phi}_{t+1}$ zu erzeugen. Der gesamte Übergang $(\boldsymbol{\phi}_t,a_t,r_t,\boldsymbol{\phi}_{t+1})$ wird im Replay Speicher gespeichert, um später damit das Netzwerk zu trainieren. Nun wird ein zufälliger Übergang $(\boldsymbol{\phi}_j,a_j,r_j,\boldsymbol{\phi}_{j+1})$ aus dem Replay Speicher ausgewählt und y_j wird bestimmt mit

$$y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$$

Mit Hilfe des Gradientenverfahrens werden nun aus dem Unterschied zwischen erwartetem Q-Value und tatsächlichem Q-Value die Gewichte des Netzwerkes aktualisiert. In der Praxis haben sie das Netzwerk mit 10 Millionen Frames trainiert, während sich immer die aktuellsten 1 Millionen Frames im Replay Speicher befanden.

2.3 Playing Super Mario World

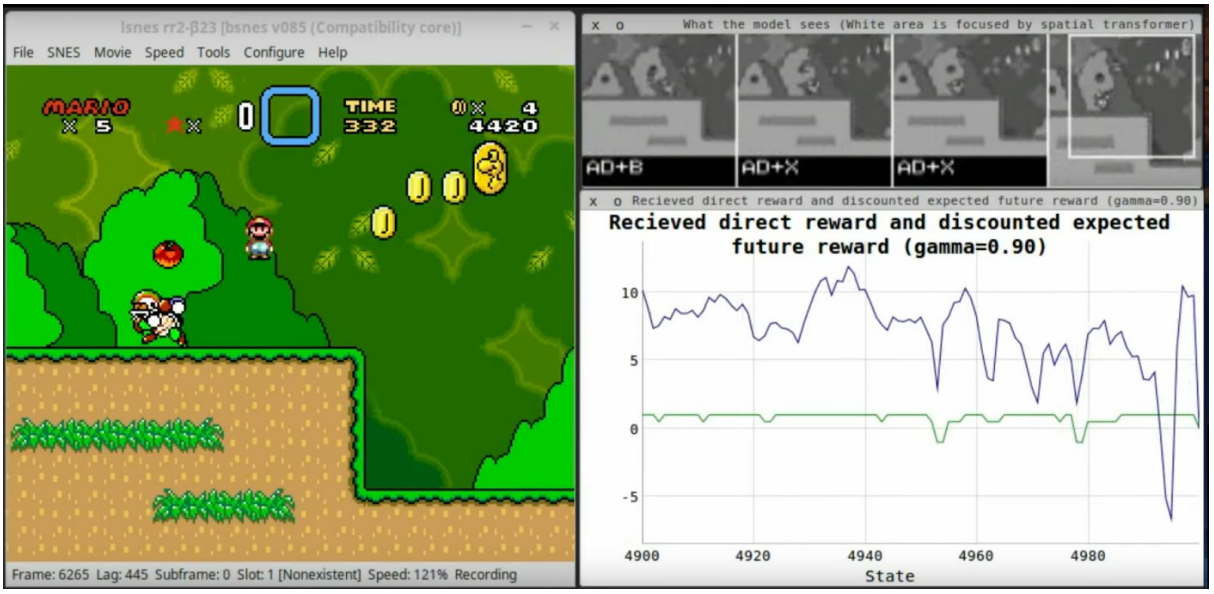

Bild 2: Neuronales Netz spielt Super Mario World. Rechts unten befinden sich der Graph für den erhaltenen Reward (grün) und dem erwarteten zukünftigen Reward (blau). Oben rechts befinden sich kleine Graustufenbilder, die zeigen was das Netzwerk sieht. Der weiß umrandete Bereich wird durch ein Spatial Transformer Network fokussiert. [3]

Der Entwickler Alexander Jung hat sich die Methoden des von Google DeepMind vorgestellten Papers zu eigen gemacht, um selbst ein kleines Projekt damit zu programmieren. Sein Ziel war es, das erste Level von Super Mario World im Super Nintendo Emulator “lsnes” von einer künstlichen Intelligenz spielen zu lassen. Er nutzte nicht nur Experience Replay, sondern ließ Mario auch nach einem beendeten Versuch an verschiedenen Stellen des Levels starten, um Überanpassung des Netzwerks zu verhindern. Als Input für sein Netzwerk benutzte er die letzten vier Screenshots, heruntergerechnet auf 32x32, Graustufe, die entsprechenden letzten vier Aktionen als two-hot-Vektor (bei Super Mario ist es öfters notwendig zwei Aktionen gleichzeitig auszuführen, wie Springen sowie eine Richtungstaste), sowie den aktuellen Screenshot in Größe 64x64, ebenfalls als Graustufenbild. Der State wird bei jedem fünften Frame erfasst, also 12 mal pro Sekunde. Sein Replay Speicher hatte eine Größe von 250.000 Einträgen. Als Ausgabe erhielt er ebenfalls die Q-Values für jede Aktion als 8-dimensionaler Vektor. Aus diesem wurden die höchsten Q-Values für eine Pfeil- sowie eine Aktionstaste ausgewählt. Als Reward nahm er nicht den Spielscore, da dieser bei Super Mario World nicht hilfreich ist, um das Level zu bestehen.

Stattdessen gibt er dem Agent:

- +0.5, hat sich nach rechts bewegt
- +1.0, hat sich schnell nach rechts bewegt (>8 Pixel)
- −1.0, hat sich nach links bewegt
- −1.5, hat sich schnell nach links bewegt (>8 Pixel)
- +2.0, während der Level-Abschluss-Animation
- −3.0, während der Todesanimation

Wie weit sich Mario in eine entsprechende Richtung bewegt hat, liest er direkt aus dem Spielspeicher aus. Weiterhin reduziert er zukünftige Rewards mit $\gamma = 0.9$. Er randomisiert seine Aktionen ebenfalls mit zu Beginn $\varepsilon = 0.8$, welches über die ersten 400.000 Aktionen auf 0.1 sinkt. Wenn eine zufällige Aktionen ausgeführt werden soll, wird eine virtuelle Münze geworfen. Entweder wird nur eine der beiden Aktionen randomisiert, oder beide.
In Bild 2 ist ein Screenshot des Youtube-Videos von Alexander Jung zu sehen, in welchem er sein Projekt vorstellt. Interessant zu sehen ist der Einbruch des Graphen des erwarteten Rewards, kurz bevor Mario über den Gegner gesprungen ist (links im Bild). Dies zeigt, dass das Netzwerk bei gleichbleibendem Status (Kollision mit dem Gegner) einen stark negativen Reward erwartet hat. Darum wurde ein Sprung ausgelöst, um den Client aus der “Schussbahn” zu holen. Kurz darauf besteht keine Gefahr mehr, sodass sich der Graph des erwarteten Rewards wieder in den positiven Bereich erhöht.
Da man in dem Video den Client nur während des Trainings sieht, sind viele Aktionen zufällig. Dies führt dazu, dass viele Aktionen sehr unkontrolliert und die Technik nicht zuverlässig wirken. Dennoch schafft es der Client öfters zum Ende des Levels zu gelangen. Das zeigt, dass Deep Reinforcement Learning auch auf kompliziertere Spiele als Atari anwendbar ist und viel Potential birgt. Noch befindet sich die Technik quasi in den Kinderschuhen, doch bestimmt schon bald werden deutlich anspruchsvollere Probleme, durch die viele Forschung die in diesem Bereich betrieben wird, lösbar sein. Im nächsten Abschnitt werde ich über den autonomen Helikopter der Stanford University reden.

3 Stanford Autonomous Helicopter

Das selbstständige Fliegen von Helikoptern gilt weithin als eines der herausforderndsten Probleme der autonomen Luftfahrt. Es ist nicht nur deutlich komplexer als das Fliegen von Flugzeugen mit starren Flügeln, es gibt auch viele schwierig zu berechnende Faktoren, die das Flugverhalten des Helikopters beeinflussen können. Zu diesen Faktoren zählen unter anderem Luftverwirbelungen der Rotoren, Wind aus unterschiedlichsten Richtungen und sogar das Momentum der Motoren hat einen Einfluss auf den Helikopter. Erst im Jahre 2001 gelang es Bagnell und Schneider einen Helikopter konstant autonom Schweben zu lassen. Doch von richtigem Fliegen, geschweige denn komplexen Kunstflugmanövern war man damals noch weit entfernt. Pieter Abbeel und sein Team von der Stanford University wollten sich dieses Problems annehmen und setzten dabei auf Techniken des Reinforcement Learnings. Doch da ein steuernder Agent bei klassischem Reinforcement Learning zunächst sehr viel ausprobieren muss, um die Konsequenzen einzelner Aktionen abschätzen zu können, würde dies bei einem autonomen Helikopter zu Problemen führen. Konkret würde der Helikopter zu Beginn sehr oft abstürzen, was schlicht zu teuer wäre. Darum entwickelten sie das sogenannte Apprenticeship Learning. Hierbei lassen sie einen Experten das Manöver mit dem Helikopter fliegen und ein Controller lernt daraus, wie er den Helikopter steuern kann. In den folgenden Abschnitten werde ich dies im Detail erläutern.

3.1 Was wird für den autonomen Flug benötigt?

Bevor ich darauf eingehen kann, wie der Helikopter komplexe Flugmanöver aus Flugdaten lernt, muss erst einmal klar gestellt werden, was zum autonomen Fliegen benötigt wird. Zunächst benötigt der Helikopter eine Flugbahn der er folgen soll. Diese haben sie zunächst per Hand festgelegt. Weiterhin benötigen sie ein Dynamics Model. Dieses sollte aus den Flugdaten des Experten erlernt werden und es soll mit dem aktuellen State und den Controls des Helikopters abschätzen, wo sich der Helikopter im nächsten Zeitschritt befindet. Der aktuelle State des Helikopters besteht aus seiner Position im 3-dimensionalen Raum, seiner Orientierung, seiner Geschwindigkeit in eine bestimmte Richtung und der Geschwindigkeit der Rotation um seine Achsen. Das Dynamics Model entspricht der Transitionfunction im klassischen Reinforcement Learning. Zu guter Letzt benötigen wir noch einen Controller, der dem Helikopter die entsprechenden Eingaben gibt, mit dem er die gewünschte Flugbahn fliegen kann. Dies entspricht der Policy im klassischen Reinforcement Learning.

3.2 Erster Algorithmus

In werde nun den Algorithmus erläutern, mit dem sie den Helikopter haben lernen lassen. Zuallererst lassen sie einen Experten einen Beispielflug fliegen. Bei diesem Flug vollführt er die Manöver, die der Helikopter später fliegen soll, aber auch einfachste Grundbewegungen. Im Schnitt benötigten sie circa fünf Minuten an Flugdaten des Experten. Mit diesen Daten berechneten sie dann ein erstes Dynamics Model für den Helikopter. Weiterhin berechneten sie aus den Daten, der handgecodeten Flugbahn und einer Funktion, die Abweichen von dieser Flugbahn bestraft eine Reward Funktion. Im nächsten Schritt bestimmten sie mit Hilfe von Simulationen einen Controller, der in der Lage war diese Flugbahn zu fliegen. Nun wurde der Helikopter mit diesem Controller geflogen und die Daten, die dabei anfielen, wurden zu den Flugdaten hinzugefügt. Wenn der Helikopter die vorgegebene Flugbahn geflogen ist, waren sie fertig. Ansonsten wurde aus den neuen Flugdaten ein neues Dynamics Model errechnet und der Prozess beginnt von vorne. Meistens benötigten sie nur drei Iterationen dieses Algorithmus um das gewünschte Manöver autonom fliegen zu können.

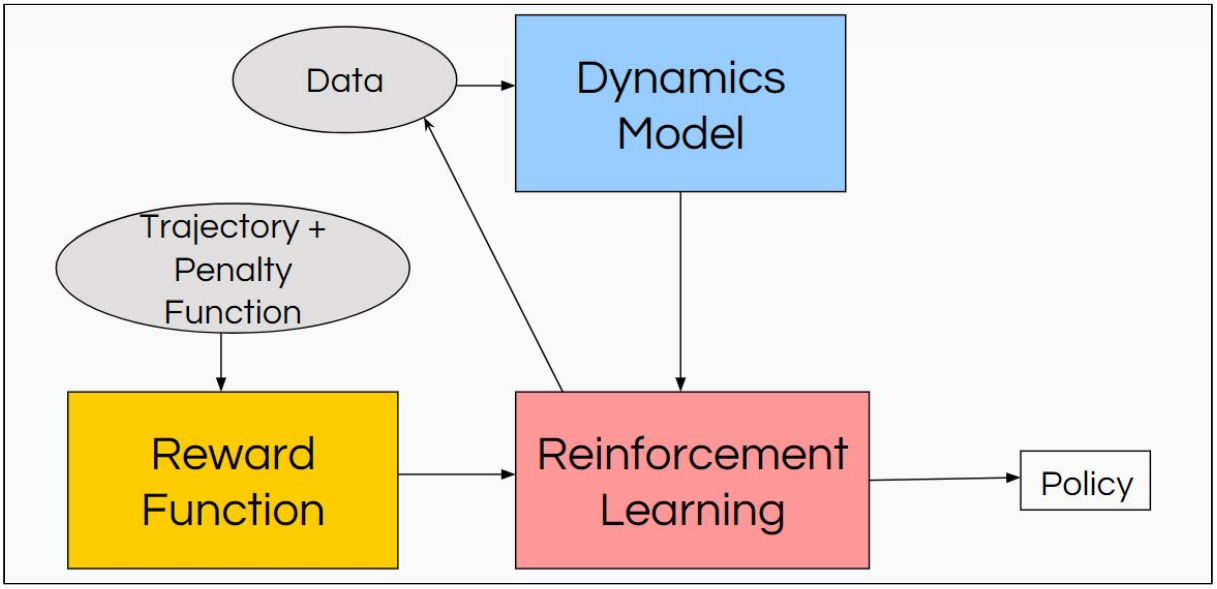

Bild 3: Übersicht der Funktionsweise des Algorithmus, der mit Hilfe von Beispielflügen lernt, komplexe Flugmanöver auszuführen.

Nun stellt sich mit dieser Herangehensweise aber ein entscheidendes Problem. Zwar lernt der Helikopter unglaublich schnell die entsprechenden Manöver auszuführen, aber dennoch handelt es sich dabei um recht einfache Kunststücke, verglichen mit dem was ein menschlicher Pilot kann. Der Grund dafür ist, dass man die Flugbahn von komplexen Manövern nicht per Hand programmieren kann. Einerseits wissen selbst Experten nicht genau, was im Detail bei solchen Kunststücken passiert. Andererseits muss die Flugbahn auch überhaupt im Bereich

des Möglichen für den Helikopter liegen. Dies ist bei einer von Hand erstellten Flugbahn nicht zwangsläufig gegeben. Darum entschieden sie sich ihr System zu verbessern, indem sie den Helikopter nun auch die Flugbahn per Apprenticeship Learning lernen lassen. Im folgenden Abschnitt werde ich dies kurz erläutern, auch wenn sie dabei den Bereich des Reinforcement Learnings verlassen.

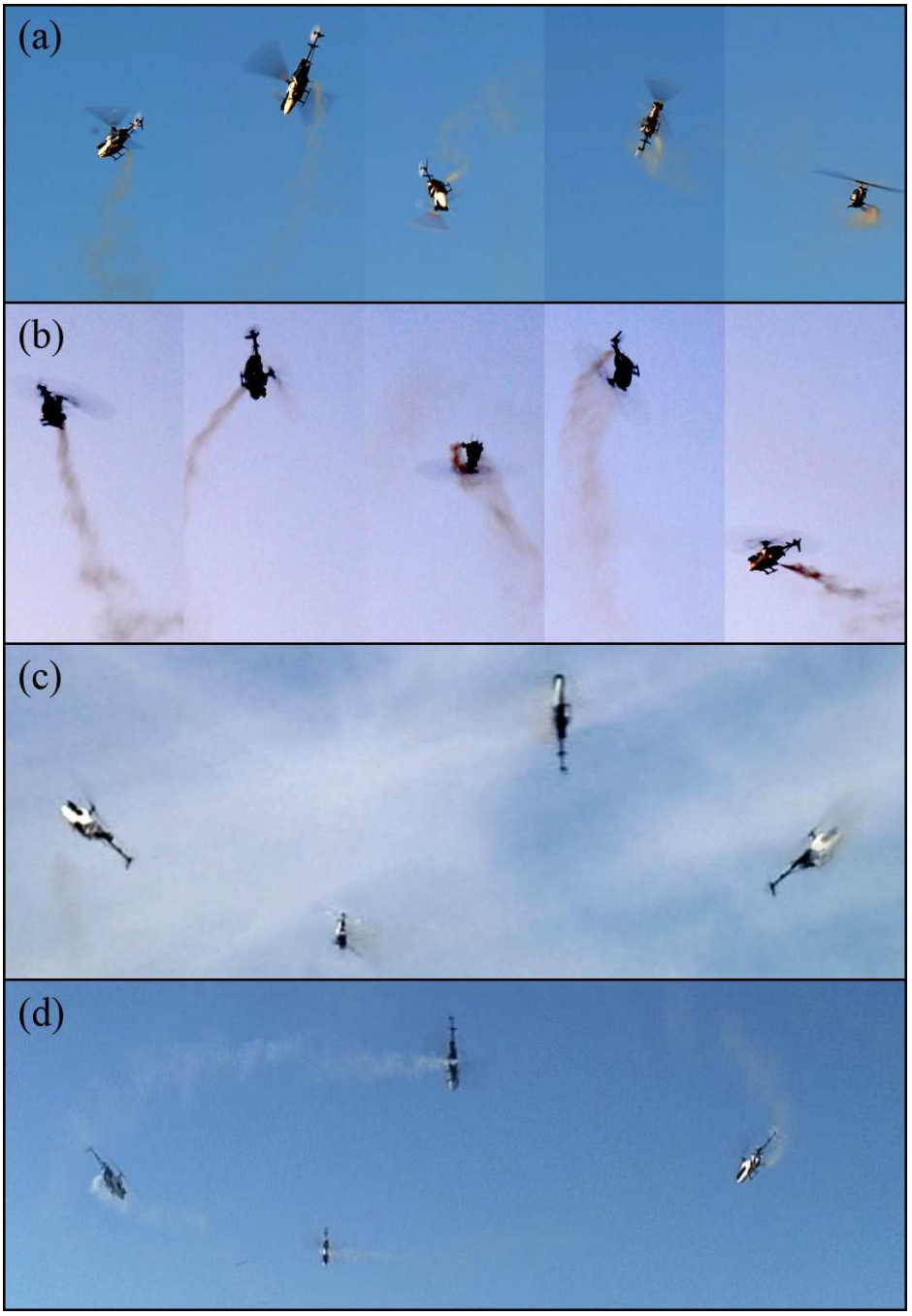

Bild 4: Reihe von Fotos der mit dem ersten Ansatz möglichen Flugmanöver. Diese sind Forward Flip (a), Low Speed Sideways Roll (b), Tail-in Funnel (c) und Nose-in Funnel (d). [4]

3.3 Apprenticeship Learning der Flugbahn

Um dem Helikopter komplexere Flugmanöver, als die, die sie per Hand programmieren können beizubringen, entschieden sie sich, den Helikopter die Flugbahn dieser ebenfalls per Apprenticeship Learning beizubringen. Hierbei stellt sich jedoch das Problem, dass selbst die erfahrensten Experten ein Kunststück nicht fehlerfrei vorführen können. Dadurch ist es schwierig für eine Maschine zu erkennen, ob eine gewisse Bewegung beabsichtigt war, oder nicht. Darum ließen sie den Experten das entsprechende Manöver mehrmals ausführen und speicherten die jeweiligen States s des Helikopters und die dazugehörigen Steuereingaben u währenddessen in einem Statevektor der Form:

$$y_j^k = \begin{bmatrix} s_j^k \\ u_j^k \end{bmatrix}, \text{ for } j = 0..N^k - 1, k = 0..M - 1$$

Hierdurch haben sie Zugriff auf den State und die dazugehörigen Eingaben zu jedem Zeitpunkt j von jeder Vorführung k des Manövers. Das Ziel war es nun, die "versteckte", beabsichtigte Flugbahn dieser Demonstrationen zu berechnen. Diese hat die Länge T und die Form:

$$z_t = \begin{bmatrix} s_t^\star \\ u_t^\star \end{bmatrix}, \text{ for } t = 0..T - 1$$

Sie nehmen nun an, dass die beabsichtigte Flugbahn einem nonlinearen Dynamical Model folgt, auf welches ein gaußsches Rauschen addiert wird. Somit erhalten wir den Folgezustand z_{t+1} mit folgender Formel

$$z_{t+1} = f(z_t) + \omega_t$$

Weiterhin modellieren sie die Vorführungen des Experten als unabhängige Beobachtungen eines "versteckten" States. Auf Bild 5 (a) ist diese Modellierung dargestellt. Leider wissen sie nicht, welcher versteckte State zu welcher Demonstration gehört. Zum Beispiel kann die Demonstration y_1 zu den versteckten States z_1, z_2 und z_3 gehören. Daher nehmen sie weiter an, dass es ein unbekanntes Set von Zeitindizes τ gibt, welche die Beobachtungen den versteckten States zuordnen. Die Flugbahn, die sie damit erhalten, erfüllt auf jeden Fall ein Dynamics Model, welches der Helikopter fliegen kann, aber noch wissen sie nicht wie τ aussieht.

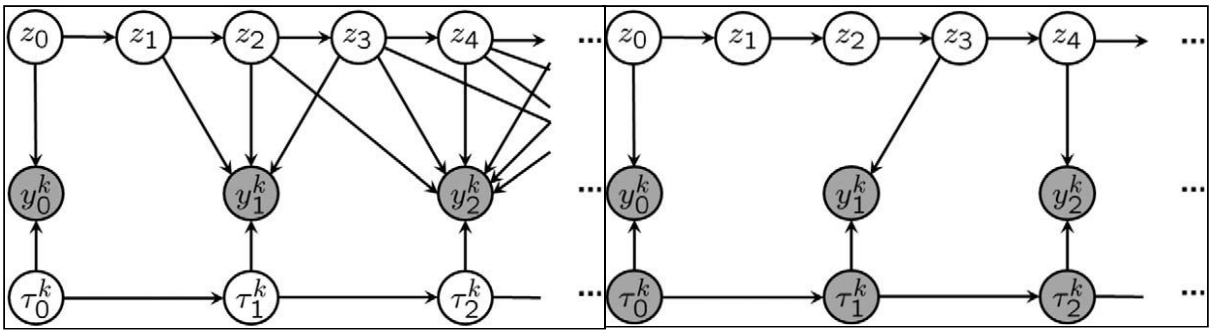

Bild 5: Modellierung der Annahmen zur Bestimmung der beabsichtigten Flugbahn (a) und Wechsel zu Hidden Markov Model bei fixem τ (b). [5]

Solange τ unbekannt ist, ist es sehr schwierig auf z zu schließen. Doch wenn τ bekannt wäre, ergibt sich ein normales Hidden Markov Model, für das es Algorithmen gibt, die diese lösen. Somit bedienen sie sich eines einfachen Tricks um eine Lösung für ihr Problem zu finden. Sie bestimmen eine anfängliche Schätzung für τ und fixieren das Modell darauf. Nun lassen sie den Baum-Welch Algorithmus auf das daraus resultierende Hidden Markov Model laufen. Daraufhin benutzen sie Dynamic Time Warping, um mit diesen Lösungen ein neues τ zu bestimmen. Zwischen diesen beiden Methoden alternieren sie, bis die Werte konvergieren und sie die beabsichtigte Flugbahn des Experten erhalten.

3.4 Weitere Anpassungen und Fazit

Um die Präzision der Ausführung weiter zu verbessern, erweiterten sie ihr System um noch zwei weitere Punkte. Einerseits passten sie das Dynamics Model an, sodass dieses Zeitabhängig war. Dadurch hatten Datenpunkte, die zeitlich nah an am aktuellen Punkt der Ausführung waren (und wodurch der Helikopter beim Sammeln der Daten vermutlich in einer ähnlichen Position im Raum war) ein höheres Gewicht auf das Dynamics Model, wodurch dieses die Bewegungen des Helikopters deutlich präziser vorhersagen konnte. Weiterhin machten sie es möglich, vorheriges Wissen über die Flugbahn in den Algorithmus einfließen zu lassen. Dazu gehört zum Beispiel, dass der Helikopter sich bei einem In-Place-Flip möglichst wenig im Raum bewegt, oder Loops auf einer festen Ebene im Raum stattfinden sollten.
All diese Anpassungen bei ihrem zweiten Versuch führten dazu, dass der Helikopter nun in erstaunlich kurzer Zeit allerhand Flugmanöver lernen und diese ohne Pause hintereinander ausführen kann. Die Videos der Flugshows sind sehr beeindruckend und die Bewegungen des Helikopters zu beschreiben übersteigt meine Fähigkeiten. Daher empfehle ich dieses Video selbst zu schauen, ich habe es in meinen Quellen verlinkt. Im letzten Kapitel werde ich erklären wie DeepMind eine künstliche Intelligenz für das Spiel Go entwickelt hat.

4 Alpha Go

Das Spiel Go gilt seit langer Zeit als eines der herausforderndsten, klassischen Spiele für künstliche Intelligenzen. Bei einer maximalen Spielfeldgröße von 19x19 Feldern ergeben sich $2{,}08*10^{170}$ gültige Spielpositionen, ein Spieler hat circa 250 mögliche Züge pro Position und ein Spiel dauert circa 150 Züge an. Diese Komplexität übersteigt die von Schach bei weitem, bei dem ein Spieler nur 35 mögliche Züge hat bei einer durchschnittlich Spiellänge von 80 Zügen. Eine künstliche Intelligenz, die den kompletten Spielbaum zur Verfügung hat, ist mit unserer heutigen Technologie daher absolut unmöglich und bisherige Go Programme, welche mit Monte Carlo Tree Search (MCTS) Algorithmen und mehreren tausend Simulationen pro Runde arbeiten, bewegten sich maximal auf dem Level von Amateuren. Einem Team unter David Silver und Demis Hassabis bei Google DeepMind ist es gelungen, mit Hilfe von Neuronalen Netzen, kombiniert mit unterschiedlichen Lernmethoden, eine AI zu entwickeln, welche den europäischen Go Meister 5 zu 0, und bisherige AIs mit einer Rate von 99,8% besiegt. Hier werde ich die einzelnen Entwicklungsschritte von AlphaGo erklären.

4.1 Trainingspipeline

Das neuronale Netz, welches in der künstlichen Intelligenz steckt, wurde in drei Stufen trainiert. In der ersten Stufe trainierten sie das sogenannte Supervised Learning (SL) Policy Network p_σ. Dieses Netzwerk alterniert zwischen Convolutional Layern mit Gewichten σ und Rectified Linear Units. Den Abschluss macht ein Softmax Layer der eine Wahrscheinlichkeitsverteilung über alle gültigen Züge ausgibt. Das Netzwerk hat insgesamt 13 Layer und bekommt den Boardstate als einfaches 19x19 Bild als Input. Das Netzwerk wird mit zufälligen State-Action-Paaren aus einem Datensatz von 30 Millionen Positionen des KGS Go Servers trainiert. Dieses Netzwerk erreichte eine Genauigkeit von 57% im Vorhersagen von Zügen eines Experten und benötigte 3 ms für eine Schätzung. Im gleichen Schritt trainierten sie auch ein kleineres Fast Rollout Policy Network $p_{\square}$, welches nur eine Genauigkeit von 24% erreichte, aber dafür auch nur 2 µs für das Auswählen einer Aktion benötigt.
In der zweiten Stufe trainierten sie das sogenannte Reinforcement Learning Policy Network p_ρ, welches die gleiche Struktur wie p_σ hat und mit den selben Gewichten initialisiert wird. Dieses Netzwerk spielt gegen zufällige vorheriger Iterationen von sich selbst, eine Taktik die so ähnlich bereits bei TD-Gammon, einer AI für Backgammon zum Einsatz kam. Durch die zufällige Gegnerwahl wird das Training stabilisiert und Overfitting verhindert. Das Netzwerk bekam als Reward +1, falls es ein Spiel gewann und −1 falls es ein Spiel verloren hat. Für jeden Zug dazwischen erhielt es keinen Reward. Das hiermit trainierte Netzwerk erreichte bereits eine

Siegrate von 85% gegen Pachi, einer Go AI welche mit MCTS 100.000 Simulationen pro Runde ausführt.
In der dritten Stufe trainierten sie das sogenannte Value Network v_θ. Dieses sollte, anders als die vorherigen, nicht eine Wahrscheinlichkeitsverteilung über die möglichen Spielzüge ausgeben, sondern aus dem aktuellen Boardstate den Ausgang des Spieles abschätzen. Auch dieses Netzwerk hat eine ähnliche Struktur wie die Policy Netzwerke. Für die Abschätzung der Züge der einzelnen Spieler, die dafür nötig ist, benutzen sie die Policy, die sie durch das RL Netzwerk p_ρ erhalten haben. Dies kommt einer Value-Funktion für perfekte Spielzüge beider Spieler am nächsten. Als Trainingsdatensatz verwendeten sie 30 Millionen Spiele von p_ρ gegen sich selbst, da Training auf dem Datensatz des KSG Go Servers zu Overfitting führte.

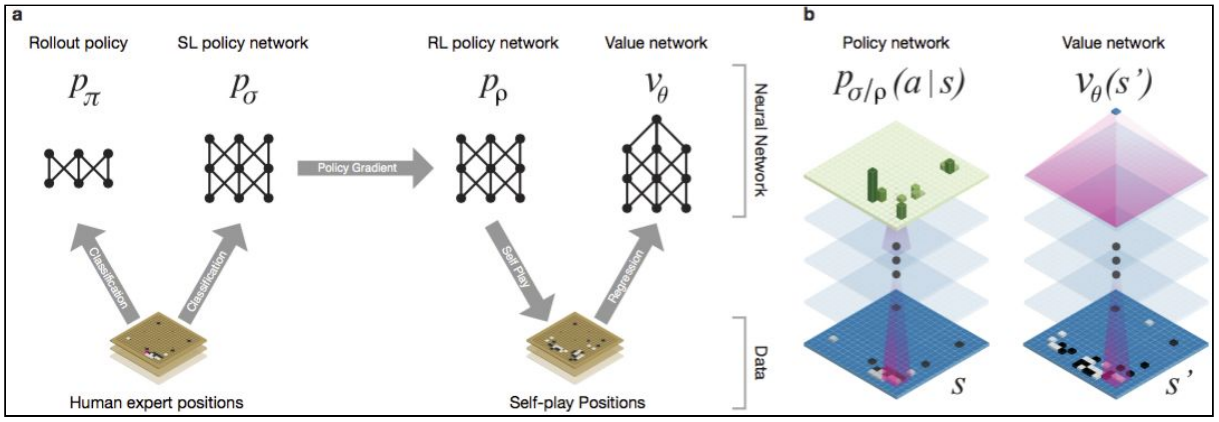

Bild 6: Schematische Darstellung der Trainingspipeline (a) und Architektur der verwendeten Netzwerke (b). [7]

4.2 Zusammensetzen der Einzelteile

Um nun die finale Version von AlphaGo zu erhalten, kombinieren sie die einzelnen Netzwerke mit einem Monte Carlo Tree Search Algorithmus. Jede Kante (State s, Action a) des Suchbaumes speichert eine Action Value Q(s,a), wie oft die Kante besucht wurde N(s,a) und eine anfängliche Wahrscheinlichkeit P(s,a). Bei jedem Schritt der Simulation wird die Aktion mit der höchsten Action Value plus einem Bonus ausgewählt, welcher zunächst der anfänglichen Wahrscheinlichkeit entspricht und mit fortlaufender Benutzung der Kante abnimmt. Dies ist in Bild 7 (a) illustriert. Damit soll der Algorithmus zum Erkunden neuer Spielwege angeregt werden.
Wenn beim Durchlaufen des Suchbaumes ein Blattknoten s_L erreicht wird und es sich nicht um einen Endknoten handelt, wird dieser expandiert. Der entsprechende State wird dann einmal vom SL Policy Network p_σ verarbeitet und die ausgegebenen Wahrscheinlichkeiten werden in P(s,a) für jede Aktion gespeichert, wie auf Bild 7 (b) dargestellt.

Der Blattknoten wird auf zwei verschiedene Arten bewertet. Einerseits wird er vom Value Network v_θ verarbeitet und andererseits wird eine Simulation von diesem Knoten mit der Policy des Fast Rollout Policy Networks bis zum Ende des Spiels gestartet. Diese beiden Bewertungen werden mit Parameter λ zur letztendlichen Blattbewertung $V(s_L)$ gemischt

$$V(s_L) = (1 - \lambda)v_\theta(s_L) + \lambda z_L$$

Mit einem Wert von 0,5 für λ erhielten sie die besten Ergebnisse. Das Interessante an dieser Bewertung ist, dass es sich hierbei, übertragen auf menschliches Denken, um eine Mischung aus intuitiver Bewertung durch das Value Network und strategischem Handeln durch die Simulation handelt. Diese Form der Bewertung kommt menschlichem Denken daher näher als eine reine einseitige Bewertung.
Die Action Values Q(s,a) jeder Kante verbinden die durchschnittlichen Bewertungen jeder Simulation, die durch diese Kante ging. Nach n Simulationen werden die Q-Values und die Anzahl der Simulationen, die durch die Kante gingen aktualisiert. Hierfür wird einfach der Durchschnitt der Bewertungen genommen, wobei **1**(s,a,i) angibt, ob eine Kante während der i-ten Simulation benutzt wurde.

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{n} \mathbf{1}(s,a,i) V(s_L^i)$$

Wenn die Suche vollständig abgeschlossen ist, wird die Aktion am Wurzelknoten ausgewählt, welche während den Simulationen am häufigsten benutzt wurde.

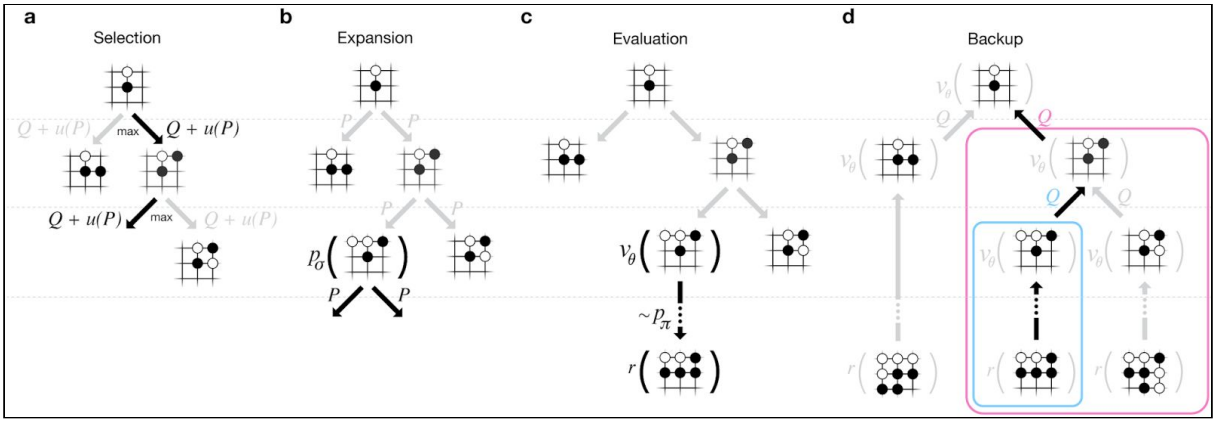

Bild 7: Schematische Darstellung der einzelnen Abläufe während des Monte Carlo Tree Search Algorithmus. [7]

4.3 Performance und Fazit

DeepMind ließ AlphaGo in einem Turnier gegen andere Goprogramme antreten. Hierbei haben sie eine einzelne Maschine benutzt, welche 48 CPUs und 8 GPUs verwendete, und eine verteilte Version mit 1202 CPUs und 176 GPUs. Bei einer Rundenzeit von 5 Sekunden für jede AI, gewann AlphaGo 99,8% (494 von 495 Matches) der Spiele. Selbst mit einem Handicap von 4 Steinen für die anderen AIs gewann AlphaGo immer noch 77% gegen Crazy Stone, 86% gegen Zen und 99% der Spiele gegen Pachi und benutzt dabei nur einen Bruchteil der Rechenleistung. Die verteilte Version von AlphaGo gewann 77% gegen die einfache Version und 100% der Spiele gegen andere AIs. Insgesamt ist dies ein bemerkenswertes Ergebnis. Auch gegen menschliche Spieler schlug sich AlphaGo hervorragend. Der europäische Go Meister Fan Hui wurde von AlphaGo 5 zu 0 besiegt und Lee Sedol, der als der stärkste Go Spieler der Welt gilt, wurde 4 zu 1 besiegt.

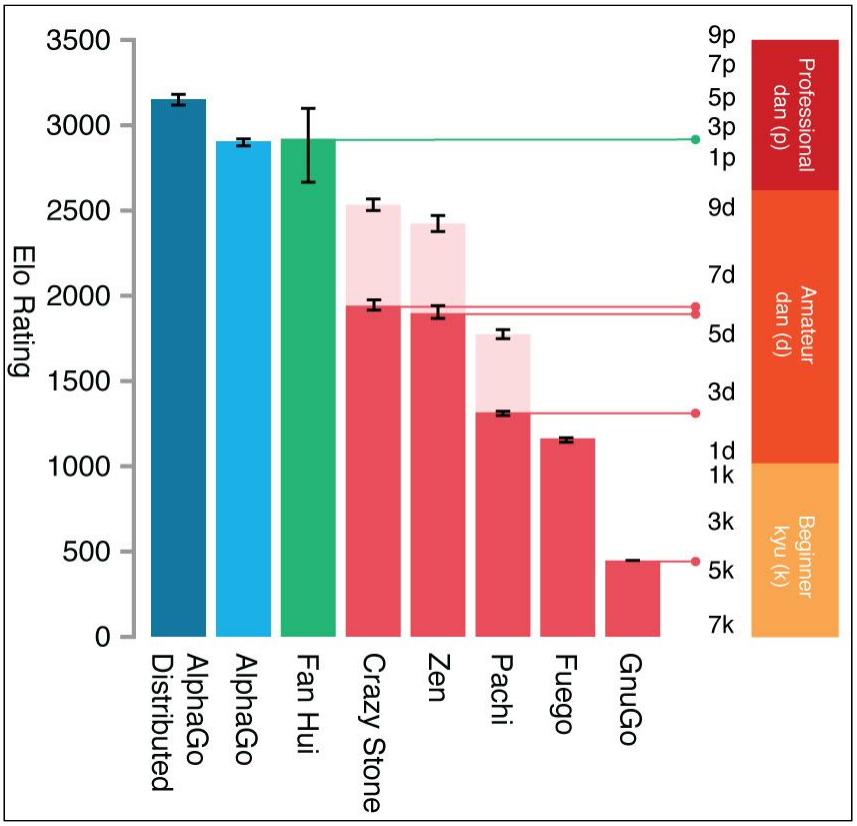

Bild 8: Elo Ratings der einzelnen Goprogramme sowieso von Fan Hui. Die Elozahl beschreibt die Spielstärke von Gospielern. [7]

All dies zeigt erneut, wie viel Potential in Deep Learning, kombiniert mit unterschiedlichsten Lerntechniken, darunter auch Reinforcement Learning, steckt. Durch geschicktes Lernen mit Datensätzen von echten Spielern und Self-play kann schon mit vergleichsweise wenig Rechenpower eine starke künstliche Intelligenz erschaffen werden, welche menschliche Fähigkeiten übersteigt.

5 Fazit

Reinforcement Learning bietet eine unglaubliche Vielzahl an Anwendungsmöglichkeiten. Auch wenn die hier gezeigten Beispiele eher nur einen spielerischen Nutzen vermuten lassen, gibt es auch für die Industrie interessante Anwendungen. Zum Beispiel gibt es bereits Roboter, welche mit Reinforcement Learning innerhalb von 8 Stunden lernen, unsortierte Objekte aus einer Kiste aufzusammeln und auf ein Fließband zu legen. Dadurch erübrigen sich aufwendige Maschinen, die die Objekte in eine gewisse Position bringen. Dies spart Zeit, Geld und die Maschinen sind unglaublich flexibel. Auch wenn klassisches Reinforcement Learning heutzutage kaum noch Anwendung findet, ist es in Verbindung mit dem in den letzten Jahren erst richtig aufgekommenen Deep Learning von großer Relevanz und ich vermute, dass es in den nächsten Jahren noch für einige großartige Neuerungen verantwortlich sein wird.

6 Quellen

[1] Playing Atari with Deep Reinforcement Learning Paper
https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf
[2] Convolutional Neural Network for Game
https://wiki.tum.de/display/lfdv/Convolutional+Neural+Network+for+Game
[3] Playing Super Mario World
https://youtu.be/L4KBBAwF_bE
https://github.com/aleju/mario-ai
[4] Stanford Autonomous Helicopter First Attempt
http://cs.stanford.edu/groups/helicopter/papers/nips06-aerobatichelicopter.pdf
[5] Stanford Autonomous Helicopter Second Attempt
https://people.eecs.berkeley.edu/~pabbeel/papers/AbbeelCoatesNg_IJRR2010.pdf
[6] Stanford Helicopter Air Show
https://youtu.be/VCdxqn0fcnE
[7] DeepMind mastering Go (Originaler Link inzwischen Offline)
https://vk.com/doc-44016343_437229031?dl=56ce06e325d42fbc72
https://www.tastehit.com/blog/google-deepmind-alphago-how-it-works/