Implementierung von neuroevolutionären KI-Verfahren für das Erlernen von Atari-Spielen und Robotik-Problemen

Dokumentation von

Nico Peter

an der Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation (IPD)

Erstgutachter:	Ralf Reussner
Zweitgutachter:	Daniel Zimmermann
Betreuender Mitarbeiter:	

24. Oktober 2019 – 31. März 2020

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Änderungen entnommen wurde.

PLACE, DATE

. . . . . .

(Nico Peter)

Zusammenfassung

In dieser Arbeit wird ein Ansatz vorgestellt, um Modelle der künstlichen Intelligenz für das Spielen von Atari-Spielen und dem Lösen von Robotik-Problemen zu trainieren. Die Grundlage dafür bildet der Algorithmus Neuroevolution of Augmenting Topologies (NEAT), ein Vertreter der evolutionen Algorithmen. Eine Implementierung des Algorithmus findet sich unter Anderem mit NEAT-Python, welches zusammen mit dem OpenAI Gym Framework die Implementierung eines Trainingsansatzes für Atari-Spiele und Robotik-Probleme ermöglicht. Die Ergebnisse zeigen auf, dass für das Lernen von Atari-Spielen eine Bildvorverarbeitung sinnvoll ist und NEAT für Robotik-Probleme im Rahmen dieser Arbeit als ungeeignet erscheint.

Inhaltsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die Menschheit hat sich im Laufe der Zeit über Generationen hinweg weiterentwickelt. Jede Generation lernt, mit der gegebenen Umwelt besser klarzukommen, als die Vorherige. Nur die Individuen, die in der frühen Vergangenheit nicht gegessen wurden oder andersweitig versagten, sind weitergekommen und gaben ihr gelerntes Wissen an die Nachkommen weiter. Diese Art des Lernens findet auch im Bereich der künstlichen Intelligenz ihre Anwendung. Sogenannte evolutionäre Algorithmen erlauben das Trainieren von ganzen Gruppen aus künstlichen Intelligenzen, die gegeneinander um den Platz des Besten konkurrieren.

Um einen Konkurrenzkampf zu ermöglichen, ist die Definition einer Herausforderung erforderlich. Die Gestalt solcher zu lösender Probleme ist sehr vielfältig. Es wurden bereits Veröffentlichung gemacht, in denen künstliche Intelligenzen versuchen, verschiedene Videospiele zu meistern. Dazu gehören zum Beispiel Atari- [**stanley2002**] oder Sega-Spiele [**gupta2019**]. Weiterhin gibt es Situationen, in denen mithilfe verschiedener Arten von Robotern Versuche unternommen werden, sich möglichst schnell fortzubewegen. Die Schwierigkeit liegt hierbei darin, alle gegebenen Gelenke eines Roboters optimal anzusteuern, um das definierte Ziel zu erreichen.

In dieser Arbeit sollen künstliche Intelligenzen für das Spielen von Atari-Spielen und dem Lösen von Robotik-Problemen trainiert werden. Dafür gestaltet sich der Inhalt dieser Arbeit folgendermaßen. Zu Beginn werden die essentiellen Grundlagen für das Problemverstehen beleuchtet. Dazu gehört das Thema der evolutionären Algorithmen und ein Vertreter dieser Gruppe namens Neuroevolution of Augmenting Topologies (NEAT). Außerdem werden das Framework Gym von OpenAI zur Bereitstellung von Trainingsumgebungen und eine Auswahl an NEAT-Implementierungen vorgestellt. Danach werden die vorgestellten Frameworks in einem Implementierungsansatz kombiniert, um sowohl das Training von Atari-Spielen als auch Robotik-Problemen zu ermöglichen. Im Anschluss daran werden die Trainingserfahrungen und Ergebnisse präsentert, ehe die Arbeit mit einer Zusammenfassung und einem Ausblick endet.

2 Grundlagen

2.1 Evolutionäre Algorithmen

Es gibt viele mögliche Verfahren, um maschinelles Lernen durchzuführen. Das in dieser Arbeit verwendete Verfahren gehört zu der Klasse der evolutionären Algortihmen [**dirk2018**]. Dabei wird nicht nur ein Modell trainiert, sondern eine Menge von Modellen gleichzeitig. Evolutionär ist ein Algorithmus auf Grund der Ähnlichkeit zur Evolution in der Natur. In der Natur versucht eine Spezies möglichst gut mit der Umwelt klarzukommen, um bspw. Nahrung oder einen Partner für die Fortpflanzung zu finden. Einzelne Individuen der Spezies fallen Räubern zum Opfer oder sind in einem bestimmten Sinne nicht gut genug für das andere Geschlecht, sodass keine nachfolgende Generation entstehen kann. Dieser Überlebens- bzw. Konkurrenzkampf findet sich auch in evolutionären Algorithmen wieder. Der konkrete Zyklus evolutionärer Algorithmen wird im Folgenden beleuchtet.

Evolutionären Algorithmen haben einige Zustände oder Operatoren im Zyklus der Evolution gemeinsam. Zu Beginn wird eine Menge von Individuen initialisiert, die als Population bezeichnet wird. Jedes Individuum wird anschließend auf ein zu lernendes Problem angewendet. Dabei wird mit jeder Aktion eines Individuums ein Wert zurückgegeben, der den Erfolg oder Misserfolg einer Aktion darstellt. Dieser reelle Wert wird Belohnung oder Fitness genannt und wird auf Basis einer zu definierenden Fitness-Funktion berechnet. Anhand der jeweils trainierten Fitness wird eine Untermenge der besten Individuen aus der Population gewählt, was als Selektion bezeichnet wird. Danach folgt eine sogenannte Rekombination, bei der die Gene der aktuellen Gewinner kombiniert werden, um somit die Grundlage für die nächste Generation zu bilden. Anschließend werden Eigenschaften eines Anteils der neuen Population verändert. Dieser Vorgang ist die Mutation und sorgt dafür, dass eine Art von Vielgestaltigkeit in der Population existiert. Mit diesem Ansatz wird versucht, lokalen Optima entgegenzuwirken. Nach der Mutation folgt erneut die Selektion und Rekombination. Diese drei Operationen werden durchgeführt bis ein bestimmter Fitness-Schwellwert erreicht wird oder eine Trainingszeit abgelaufen ist.

Es existieren verschiedene konkrete Vertreter der evolutionären Algorithmen, von denen der für diese Arbeit Relevante im folgenden Abschnitt vorgestellt wird.

2.2 Neuroevolution of Augmenting Topologies (NEAT)

Der Algorithmus Neuroevolution of Augmenting Topologies (NEAT) ist ein Beispiel für einen evolutionären Algorithmus und wurde im Paper von [**stanley2002**] vorgestellt. NEAT im Detail zu beschreiben, würde den Rahmen dieser Dokumentation sprengen, weshalb im Folgen-

den lediglich auf Grundaspekte des Algorithmus eingegangen wird. Bei NEAT geht es um künstliche neuronale Netze, die möglichst klein beginnen, aber mit der Zeit an Struktur und damit Komplexität hinzugewinnen. Die Initialisierung des evolutionären Vorgangs geschieht dementsprechend mit der Generierung der Population aus Netzen. Die Gene eines jeden Netzes setzen sich aus je einer Liste an Neuronen und Verbindungen zusammen. Neuronen bestehen in dieser Liste jeweils aus einer Nummer und der Bezeichnung als Eingabe-, Ausgabe- oder Verstecktes-Neuron (engl. hidden). Die Beschreibung einer Verbindung enthält dagegen die Informationen, zwischen welchen Neuronen sie verbindet, welches Gewicht sie hat, ob sie aktiv oder deaktiviert ist und eine sogenannte Innovationszahl. Letzteres wird dazu verwendet, um eine Rekombination homogener Netze zu ermöglichen.

Die Netze erhalten für ihre Ausgabe einen Fitness-Wert einer Fitness-Funktion, mit dem nach einer Trainingsepisode eine Selektion der besten Netze stattfindet. In NEAT konkurrieren jedoch nicht jeweils alle Netze der Population mit allen Anderen, sondern innerhalb von Gruppen, die durch eine Artbildung entstehen. NEAT gruppiert Netze mit ähnlichen Topologien und Gewichten, um dadurch eine Vielgestaltigkeit zu gewährleisten und lokalen Optima möglichst lange zu entfliehen. Verschiedene Topologien entstehen bei NEAT durch den Mutationsvorgang, in welchem ein Netz neue Neuronen oder Verbindungen hinzubekommt oder stattdessen entfernt werden.

Um mithilfe des NEAT-Algorithmus Modelle für Atari- oder Robotik-Umgebungen rechnergestützt zu trainieren, werden die entsprechenden Umgebungen selbst benötigt. Eine mögliche Quelle dafür wird im folgenden Abschnitt erläutert.

2.3 Open AI Gym

Um Modellen des maschinellen Lernens eine einheitliche Lernumgebung zu bieten, wurde die Python-Bibliothek OpenAI Gym im Jahr 2016 von der Forschungseinrichtung Open AI veröffentlicht [**brockman2016**]. Open AI verfolgt das Ziel, die Forschung bezüglich künstlicher Intelligenz global offener und standardisierter zu etablieren. Darum bietet OpenAI Gym die Möglichkeit an, über festgelegte Schnittstellen in gleicher Weise auf unterschiedliche Lernumgebungen zuzugreifen, die entweder bereits in OpenAI Gym implementiert sind oder von Bibliotheken Dritter stammen.

Alle Lernumgebungen in OpenAI Gym bieten die Möglichkeit an, eine Aktion über die Methode ßtep(action)"durchzuführen. Dabei wird ein Parameter äction"mitgegeben, der einen oder mehrere Zahlenwerte enthält, die eine Aktion innerhalb der Umgebung beschreibt. In Atari-Spielen ist äctionëine Ganzzahl, die für Aktionen wie "Nach links fliegenöder SSchießenßteht. Jede Aktion liefert vier Rückgabewerte, die im Folgenden aufgelistet sind.

- Beobachtung (engl. observation) vom Typ „object“: Ein an die Umgebung spezialisiertes Objekt, das dementsprechende Attribute, wie Pixeldaten einer Kamera oder den Zustand eines Brettspieles enthält. Bereits mit dem Befehl „env.reset()“ in Zeile 4 des Listings wird eine erste Beobachtung zurückgegeben, die in diesem Beispiel keine Verwendung findet.

- Belohnung (engl. Reward) vom Typ „float“: Dieser Wert gibt die Größe der Belohnung an, die mittels der letzten Aktion erhalten wurde. Die Skalierung des Wertes variiert dabei zwischen verschiedenen Umgebungen. Die Erhöhung des insgesamt erzielten Belohungswertes ist meistens das Ziel eines in diesem Kontext angewandten Algorithmus.
- Fertig (engl. Done) vom Typ „boolean“: Wenn dieser Wert „wahr“ ist, signalisiert die Umgebung, dass ein Zurücksetzen der Umgebung notwendig ist. Gründe dafür sind beispielsweise das Verlieren aller Lebenspunkte in einem Spiel.
- Information (engl. Info) vom Typ „dictionary“: Hierbei handelt es sich um für das Debugging relevante Informationen, die nicht für das Lernen eines Algorithmus verwendet werden dürfen.

Für das Trainieren einer künstlichen Intelligenz für Robotik-Probleme, stehen verschiedene konkrete Trainingumgebungen zur Auswahl. Zusätzlich sind diese Umgebungen aus unterschiedlichen Programmbibliotheken beziehbar. Zum Einen sind das die Robotik-Umgebungen aus Roboschool, die in der Pybullet-Bibliothek kostenlos verfügbar sind. Zum Anderen liefert die Mujoco-Bibliothek kostenlos für Testzwecke oder gegen Bezahlung ebenfalls Robotik-Umgebungen. In beiden Bibliotheken sind nahezu die gleichen Robotik-Aufgaben zu finden, wobei sie unterschiedlich implementiert sind. Das führt bei Pybullet dazu, dass ein NEAT-Modell hier schwieriger zu trainieren ist, als in Mujoco. Aus diesem Grund wird im Rahmen dieser Arbeit eine kostenlose Mujoco-Lizenz für Studenten erworben, die für ein Jahr und nur auf einem Endgerät gültig ist.

Die für diese Arbeit andere relevante Kategorie ist „Atari“. In jeder Umgebung wird ein Atari-Spiel emuliert, das ein Agent so gut wie möglich abschließen soll. Jedes Spiel ist in jeweils zwei Umgebungen verfügbar. Eine der beiden Umgebungen liefert ein Bild des Spiel-Bildschirms als Eingabe für das Lernen, während die andere Umgebung den Zustand des RAM der emulierten Atari-Maschine als Input anbietet. Somit ergeben sich momentan 59 verfügbare Atari-Spiele. Daher existieren viele verschiedene mögliche Aufgaben-Typen. Zum Beispiel werden in der Umgebung „Assault-v0“ Punkte durch das Zerstören von feindlichen Raumschiffen mittels eines Lasergeschosses erzielt, während in „Bowling-v0“ eine Bowling-Kugel, von links nach rechts geworfen, möglichst viele Pins umwerfen soll.

2.4 NEAT Frameworks

Um eine künstliche Intelligenz für das Spielen von Atari-Spielen oder dem Lösen von Robotik-Problemen zu trainieren, ist die Implementierung des Trainingsquellcodes erforderlich. Für den vorgestellten evolutionären Algorithmus NEAT gibt es bereits fertig implementierte Programmbibliotheken in verschieden Programmiersprachen [**stanley2020**]. Da in dieser Arbeit die Programmiersprache Python verwendet wird, werden im Folgenden nur die für Python vorhandenen Vertreter aufgelistet.

Framework	Lizenz	Veröffentlichung	Eigenschaften
NEAT-Python	BSD-3-Clause	2008	Gute Dokumentation
MultiNEAT	LGPL	2012	Schnell durch C++
PyTorch NEAT	Apache 2.0	2018	GPU-Unterstützung

Tabelle 2.1: Die Frameworks NEAT-Python, MultiNEAT und PyTorch NEAT implementieren den NEAT-Algorithmus in Python

Für die finalen Trainingsdurchläufe dieser Arbeit wird die Bibliothek NEAT-Python verwendet, da sie am bewährtesten und am besten dokumentiert ist. Daher werden im Folgenden Beispiele für Konfigurationsparameter des NEAT-Python-Frameworks vorgestellt.

In der Konfigurationsdatei für NEAT-Python befinden sich verschiedene Sektionen, die jeweils bestimmte Konfigurationsparameter beinhalten. Die Angabe aller Sektionen und deren Parameter ist bis auf ein paar Ausnahmen nicht notwendig, um die Konfigurationsdatei zu verwenden. Alle verfügbaren Parameter hier vorzustellen würde den Rahmen dieser Arbeit sprengen, daher wird stattdessen eine Auswahl der Möglichkeiten beleuchtet.

- Die NEAT-Sektion enthält Parameter, die das Experiment als Ganzes betreffen. Zum Beispiel den Schwellwert für die Populations-Leistung, ab welchem das Training für beendet erklärt wird, oder die Anzahl an Genomen in jeder Population, was die erforderliche Rechenleistung erhöht.

- Die Sektion für die standardweise Stagnation enthält Parameter für die Angabe, ab wann eine Spezies als stagniert gilt oder wieviele der besten Spezies trotz Stagnation behalten werden sollen.

- In der Sektion für die standardweise Reproduktion sind Werte wie die Anzahl der als „Elite“ zu bezeichneten Individuen oder die kleineste Anzahl an Genomen pro Spezies nach jeder Reproduktion einstellbar.

- Die dritte und damit letzte in der Dokumentation von Neat-Python erwähnte Sektion ist die für standardmäßige Genome. Hierbei sind zudem die meisten Parameter einstellbar. Dazu gehört zum Beispiel die Wahrscheinlichkeit für Mutationen, eine neue Verbindung zwischen existierenden Knoten zu erstellen.

Im nachfolgenden Kapitel werden die Implementierungen vorgestellt.

3 Implementierungsansatz

In diesem Kapitel wird beschrieben, wie mithilfe des NEAT-Python-Frameworks und der Bibltiohek Open AI Gym das Training eines Modells für Atari-Spiele und Robotik-Probleme implementiert wird. Das Training eines Modells erfolgt für beide Zieldomänen in ähnlicher folgender Weise.

Zunächst wird die Konfigurationsdatei für NEAT-Python geladen und daraus die Population generiert. Der Population werden anschließend optional sogenannte Reporter hinzugefügt. Das sind Objekte, die den Lernverlauf der Population für verschiedene Zwecke nutzen. Zum Beispiel fasst ein SStatisticsReporterObjekt Statistiken bezüglich des Lernfortschirtts, wie zum Beispiel die in jeder Generation maximal erreichte Leistung, in einem Liniendiagramm zusammen. Anschließend wird für eine parallele Ausführung ein "ParallelEvaluatorObjekt erzeugt, dem die Anzahl an Threads und eine Referenz zur Lernmethode mitgegeben wird. Der Evaluator zusammen mit einer optionalen Angabe für die Anzahl an maximal auszuführenden Generationen, werden dem Start des Trainings durch das Populations-Objekt als Eingangsvariablen mitgegeben. Das Ergebnis des Trainings ist ein Gewinner-Netz, das im Anschluss daran abgespeichert wird. Für die Evaluation der Trainingsergebnisse werden zudem Statistiken ebenfalls generiert und abgespeichert.

Die konkrete Implementierung einer Trainingsmethode hängt von der Zieldomäne ab, weshalb im Folgenden die Implementierungsansätze der Atari- und Robotik-Domäne einzeln vorgestellt werden.

3.1 Implementierung eines Trainings für Atari-Modelle

Für jeden Thread wird zunächst eine konkrete Umgebung aus OpenAI Gym instantiiert und das zu trainierende Netz mittels des Genoms dieses Threads und der Konfiguration generiert. Anschließend befindet sich der Thread in einer Endlosschleife, die endet, sobald die Lernumgebung den Wert "done == Trueßurückgibt oder durch eine selbst gewählte Zeitüberschreitung der Wert "done"manuell auf "True"gesetzt wird. In jedem Schleifendurchlauf geschehen die folgenden Dinge.

Im ersten Schritt wird die aktuelle Beobachtung, also das Bild des Atari-Bildschrims, vorverarbeitet. Im ersten Durchlauf wird das dementsprechend erste Bild mittels einer Beispielaktion erzeugt. Die Vorverarbeitung des Bildes ist wichtig, weil die Anzahl der Eingangsparameter bei einem unverarbeiteten Bild mit drei Farbkanälen bereits bei 100.800 (210 x 160 x 3) liegt, was relativ hoch ist. Stattdessen wird aus dem Farbbild ein Graustufenbild mithilfe der OpenCV-Bibliothek erzeugt, wodurch noch 33.600 (210 x 160 x 1) Parameter bleiben. Zusätzlich wird das

Bild auf 42 x 32 Pixel kleiner skaliert, um die Parameteranzahl weiter zu reduzieren. Dadurch bleiben nach der Vorverarbeitung 1.344 Eingangsparameter.

Nach der Vorverarbeitung wird die Bild-Matrix in einen einzeiligen Vektor mithilfe der Numpy-Bibliothek umgewandelt und dem Netz als Eingabe übergeben. Die resultierende Ausgabe ist ein Vektor, der für jedes der im Atari-Spiel möglichen Aktionen einen Wert enthält. Die Aktion mit dem höchsten Wert wird dem Atari-Spiel als nächste durchzuführende Aktion weitergegeben.

Die aus der Aktion resultierende Fitness wird der Fitness des Threads hinzugefügt und falls die Umgebung "done==Truesßurückgibt oder eine bestimmte Zeitgrenze erreicht wurde, wird in der Konsole unter Anderem die vom Thread erreichte Fitness ausgegeben. Die neue Beobachtung überschreibt die Alte und durchläuft im nächsten Schleifendurchlauf die bereits erwähnte Bildvorverarbeitung.

3.2 Implementierung eines Trainings für Roboter-Modelle

Für das Trainieren eines Robotik-Modells ist im Gegensatz zu Atari-Modellen keine Vorverarbeitung der Beobachtungsdaten notwendig, da die Dimension des Beobachtungsvektors vergleichsweise klein ausfällt. Der Trainingsansatz ist der Folgende.

Wie auch im Training für Atri-Modelle wird für jeden Thread eine eigene Umgebung und ein eigenes Netz erzeugt. Im Anschluss daran folgt eine Endlosschleife, die im Gegensatz zum Atari-Training nicht nach einer bestimmten Zeitspanne enden soll. Der Grund dafür ist, dass ein Netz für Atari-Spiele eine Aktion lernen konnte, bei der ein Spieler im Spiel nichts macht. Um dies in gewisser Weise zu bestrafen, wurde eine Zeitschwelle für Atari-Modelle eingeführt. In den Robotik-Beispielen gibt es eine solche lernbare Aktion explizit nicht.

In der Endlosschleife wird zunächst der aktuelle Beobachtungsvektor ohne weitere Vorverarbeitung dem Netz als Eingabe übergeben. Die resultierende Ausgabe erhält daraufhin direkt die Umgebung aus OpenAI Gym, um eine Aktion auszuführen. Für die SSwimmerUmgebung war es zudem notwendig, vor dem Aktionsschritt eine Zeit von ca. einem Zehntel einer Millisekunde zu warten, da die Mujoco-Umgebung aufgrund der parallelen Verarbeitung ansonsten einen Fehler wirft.

Zum Schluss wird die aktuelle Fitness der Fitness des Threads hinzugefügt und im Falle eines Schleifenendes in der Konsole ausgegeben. Es hat sich zudem als nützlich erwiesen, bei einer Gesamt-Fitness von unter -300 die Endlosschleife sofort abzubrechen, da ein längeres Training für Threads in diesem Bereich keine guten Erfolge erzielt.

4 Ergebnisse

In diesem Kapitel werden die Erfahrungen und Trainingsergebnisse des Modell-Trainings für Atari-Spiele und Robotik-Umgebungen beleuchtet. Da sich das Ziel des Praktikums im späteren Verlauf zum Training von Robotik-Modellen wandelte, sind lediglich für das Trainieren von Robotik-Modellen letzten Endes Statistiken entstanden. Diese sind im öffentlichen Repository des Projekts zu finden.

4.1 Ergebnisse des Atari-Trainings

Nachdem erste Tests mit dem Atari-Spiel „Pitfall“ durchgelaufen sind, ergeben sich die in diesem Abschnitt zu findenden Erkenntnisse.

Potentiell jedes Atari-Spiel hat einen eigenen Aktionsraum, obwohl es die gleiche Konsole ist. In Pitfall sind für das Spiel 18 verschiedene Eingaben möglich, während im Spiel „Space Invaders“ lediglich 6 Eingaben zur Verfügung stehen. Darum sollte zunächst mit dem Befehl ënv.env.get_action_meanings()der mögliche Aktionsraum des jeweiligen Spiels erforscht werden, um anschließend die NEAT-Konfiguration für dieses Spiel anzupassen (Anzahl der Ausgabe-Neuronen).

Während der ersten Trainings-Einheiten für „Pitfall“ sind in den gerenderten Bildszenen unnötige Aktionen aufgefallen. Genauer das Springen auf der Stelle oder das Drücken der Eingabe „Unten“, wenn der Spieler auf der Erde steht. Genauso ist die Aktion „Noop“ (nichts tun) nicht zielführend. Stattdessen ist es für dieses Spiel besser, wenn die Spielfigur möglichst in Bewegung bleibt und somit die Umgebung mehr erkundet. Darum werden die sich aus dem Netz ergebenden Aktionen, die unwichtig erscheinen, mit 0 belegt, bevor die Aktion mit dem höchsten Wert dem Spiel übergeben wird. Dadurch wird sichergestellt, dass bspw. „Noop“ niemals im Spiel ausgeführt wird. Das Ergebnis war, dass die Figur sich wesentlich mehr bewegt. Daher sollte für jedes zu trainierende Spiel untersucht werden, ob es potentielle Aktionen gibt, die für das Erreichen besserer Punktzahlen unterdrückt werden sollten.

Ein weiteres Problem ist die Schwierigkeit des Spiels. Nachdem es trotz mehrerer Anläufe der KI nicht möglich war, einen einzigen Punkt in Pitfall zu erhalten, habe ich zunächst die zur Verfügung stehende Zeit für die KI erhöht, bevor der jeweilige Versuch beendet wird. Nachdem auch das keine Punkte einbrachte, habe ich es selbst probiert. Im Internet gibt es eine Plattform, bei der kostenlos und legal im Browser einige Atari-Spiele gestartet werden können. Dazu zählt auch „Pitfall“. Da ich es selbst auch kaum schaffte, Punkte zu bekommen, ist mir dadurch die tatsächliche Komplexität des Spiels aufgefallen. Logischerweise ist es dadurch für das Netz ebenfalls schwierig, das Erlangen von Punkten zu lernen. Darum sollte vor dem

Trainieren eines Spiels zunächst dessen Schwierigkeit am besten durch eigenes Spielen in Erfahrung gebracht werden.

In manchen Beispielen aus dem Internet sind unterschiedliche Ansätze zur Bildvorverarbeitung zu finden. Zwar wurde nicht immer das NEAT-Verfahren eingesetzt, jedoch ist aus diesen Beispielen ebenfalls eine Erkenntnis im Bereich der Vorverarbeitung hervorgegangen. Zunächst werden aus dem aktuellen Bild des Spiels einige Teile herausgeschnitten. Zum Beispiel gibt es in „Pitfall“ am oberen Ende des Bildschirms eine Punkte- und Zeitanzeige. Wird dieser „Bildstreifen“ unberührt dem Netz übergeben, wird das Netz diese Anzeigen ebenfalls für das Lernen verwenden bzw. das Netz wird denken, dass dies relevante Informationen sind, um bessere Punktzahlen zu gewinnen. Da die beiden Anzeigen keinerlei Relevanz auf die resultierende Aktion des Netzes haben sollten, werden diese „Bildstreifen“ demenstprechend entfernt. Danach wird das Bild häufig in ein Graustufenbild umgewandelt, sodass statt drei Dimensionen (R, G und B) eine Dimension mit Grauwerten übrig bleibt. Das hat den Vorteil, dass das Netz für die Eingabeschicht zwei Drittel weniger Neuronen benötigt, als vor der Umwandlung. Im Anschluss daran, wird das Eingabebild verkleinert, um ebenfalls an Eingabeneuronen zu sparen. Dabei wird entweder gezielt eine Bildgröße in Pixel angegeben oder die ursprüngliche Größe wird durch eine Ganzzahl geteilt und danach gerundet. In bestimmten Fällen wird zum Schluss eine Segmentierung bzw. Binarisierung durchgeführt, um nicht relevante Hintergrundpixel, wie Bäume oder Himmel, für das Netz unsichtbar zu machen. Ob die eben erläuterten Arbeitsschritte zu einem besseren Trainingsergebnis führen, ist auszuprobieren, jedoch erhöhen sie logischerweise die Trainingsgeschwindigkeit des Netzes durch die Verkleinerung des Eingabebildes.

Weiterhin fiel in den ersten Trainingseinheiten mit „Pitfall“ auf, dass es sich als schwierig heraustellt, dem Netz „Hilfestellungen“ zu geben. In einem Beispiel, in dem eine KI für das Spiel „Sonic“ trainiert wird, wird die aus der Aktion resultierende Belohung nicht einmal verwendet. Stattdessen wird bei jeder neu gerenderten Szene die horizontale Position des Spielers aus der Variablen „info“ nach der Aktionsdurchführung ermittelt und wenn die neue Position höher (weiter rechts) ist, als die bisher erreichte, wird die tatsächliche Belohnung von anfangs 0 um 10 erhöht. Diese Zahl ist am Ende einer Episode die Belohnung für das jeweilige Genom. Dadurch wird der KI eher antrainiert, dass sie sich möglichst nach rechts bewegen soll. In „Pitfall“ ist das jedoch nicht möglich, da die „info“-Variable hierbei nur die Anzahl der verbliebenen „Leben“ bereitstellt. Die einzige andere Möglichkeit, bei „Pitfall“ den Lernprozess zu verändern, ist, nach Erhalt einer negativen Belohnung (bspw. Durch das Fallen in Löcher) die aktuelle Lernepisode sofort zu beenden. Dadurch könnte sich das Netz zumindest merken, was es nicht machen darf. Dadurch ergeben sich Netze, die die Anfangspunktzahl von 2000 Punkten besitzen aber nur selten darunter. Es wird zudem in der offiziellen Dokumentation von Open AI Gym darauf hingewiesen, dass für offizielle Evaluationen eines Netzes die Werte innerhalb der „info“-Variable nicht für das Lernen dieser Netze verwendet werden darf.

4.2 Ergebnisse des Roboter-Trainings

Für das Training von Robotik-Modellen werden die Mujoco-Umgebungen ÏnvertedPendulum", SSwimmer", "Hopper", "HalfCheetahünd Änt"verwendet. Die Trainingsergebnisse werden im Folgenden vorgestellt, wobei Statistiken zu den jeweiligen Umgebungen im öffentlichen Repository zu finden sind.

Alle trainierten Robotik-Modelle sind in Anbetracht der Ergebnisse nicht in der Lage, die jeweiligen Umgebungen zu lösen. Stattdessen werden lokale Optima erreicht, die im Anschluss nicht mehr verlassen werden. Hierfür gibt es viele mögliche Ursachen. Den wahrscheinlich größten Einfluss haben die Hyperparameter, die aus der Konfigurationsdatei geladen werden. Die Trainingsdurchläufe fanden mit einer Populationsgröße von 100 Netzen statt. Dies ist der begrenzten Rechenleistung des Trainingscomputers geschuldet, um in absehbarer Zeit Ergebnisse vorweisen zu können. Mit 300, 1000 oder mehr Netzen steigt die Wahrscheinlickeit, bessere Ergebnisse zu erzielen für den Preis der erforderlichen Rechenleistung. Durch eine größere Population könnten bessere Optima entdeckt werden.

Des Weiteren bildet die Wahl der initialen Verbindungsart, zumindest auf die zu Beginn zu beobachtenden Ergebnisse, ebenfalls eine wichtige Stellschraube. In den Hyperparametern ist einstellbar, ob die Neuronen vollvernetzt, teilvernetzt oder überhaupt nicht vernetzt sind. Im dritten Fall wurden sehr häufig nur schlechte Ergebnisse erzielt, weswegen diese Strategie nicht empfehlenswert ist. Es war zu beobachten, dass eher eine teilweise oder volle Vernetzung hilfreicher für das Training ist. Beim inversen Pendel erzielten sowohl eine initiale Vernetzung von 50% als auch von 90% nach ungefähr 200 Generationen die gleichen Werte.

5 Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war das Training von Modellen des maschinellen Lernens für das Lösen von Atarispielen und Robotikproblemen. Der dafür verwendete Algorithmus ist Neuroevolution of Augmenting Topologies (NEAT), der in einem evolutionären Verfahren künstliche neuronale Netze in ihrer Struktur und ihren Gewichten optimiert. Für die Implementierung des Algorithmus standen veschiedene bereits veröffentliche Frameworks zur Auswahl, von denen die NEAT-Python-Bibltiohek aufgrund ihrer Dokumentation und Etabliertheit in der Community gewählt wurde. Um dem Modell eine einheitliche Trainingsumgebung zur Verfügung zu stellen, wurden die Umgebungen aus dem Framework OpenAI Gym genutzt. Anschließend wurde der Code für das Training der Netze implementiert, wobei für das Lernen von Atari-Spielen eine Bildvorverarbeitung nötig war, um die Anzahl der zu lernenden Parameter zu reduzieren. Für das Lernen von Atari-Spielen wurde die Erfahrung gemacht, dass es sinnvoll ist, für ein bestimmtes Spiel den Aktionsraum auf wesentliche oder zielführende Aktionen zu beschränken und sich der Schwierigkeit eines Spiels vor dem Training bewusst zu werden. Die trainierten Robotik-Modelle waren nicht in der Lage, innerhalb ihrer Trainingszeit gute Ergebnisse zu erzielen, was entweder an der Zeitbeschränkung oder der Netzkonfiguration liegt.

Für weitere Arbeiten in Bezug auf das Trainieren von Robotik-Modellen wären längere Trainingszeiten bzw. größere Populationen denkbar, um eine höhere Chance auf bessere Optima zu gewährleisten. Dazu wäre eine tiefergehende Optimierung der Hyperparameter von NEAT sinnvoll, um für bestimmte Umgebungen bessere Ergebnisse zu erzielen. Ebenfalls wäre ein Vergleich von NEAT mit anderen evolutionären Lernverfahren interessant, um für das Lösen von Robotik-Problemen bessere Alternativen zu identifizieren.

Literatur

[gupta2019] Vedant Gupta. *How to use AI to play Sonic the Hedgehog. It's NEAT!.* https://www.freecodecamp.org/news/how-to-use-ai-to-play-sonic-the-hedgehog-its-neat-9d862a2aef98/, 2019.

[stanley2002] Stanley, Kenneth O., Risto Miikkulainen. *Evolving neural networks through augmenting topologies.* Evolutionary computation, 10.2, 99-127, 2002.

[dirk2018] Jan-Dirk. *Was sind evolutionäre Algorithmen?.* https://www.it-talents.de/blog/it-talents/was-sind-evolutionaere-algorithmen, 2018.

[brockman2016] Brockman, Greg u.a. *Openai gym.* arXiv:1606.01540, 2016.

[stanley2020] Stanley, Kenneth O. *Find the Right Version of NEAT for Your Needs.* http://eplex.cs.ucf.edu/neat_software/, 2020.