

Vergleich von Evolutionsstrategien und Deep-Q-Learning auf OpenAIs Bipedal-Walker-Problem

Tobias Döring und Philip Maas

Hochschule Bochum, Heiligenhaus

philip.maas@stud.hs-bochum.de, tobias.doering@stud.hs-bochum.de

Abstract—Diese Arbeit beschäftigt sich mit der Entwicklung zweier verschiedener Evolutionsalgorithmen und der Anwendung dieser auf dem Bipedal-Walker-Problem. Dieser Ansatz soll mit einem Deep-Q-Learning inklusive Credit Assignment in den Punkten Zeit, Erfolg, Anzahl der Hyperparameter und Komplexität der Implementation verglichen werden.

I. EINLEITUNG

Der Bipedal Walker ist ein bekanntes Problem aus dem OpenAI-Gym und dient als Benchmark für diverse Algorithmen aus dem Gebiet des maschinellen Lernens. Der Bipedal Walker ist mit zwei Beinen, vier Gelenken und einem Rumpf ausgestattet, wie in Abbildung 1 zu sehen ist. In diesem Projekt sollen verschiedene Algorithmen aus dem Bereich des bestärkenden Lernens und des Evolutionslernens auf diesem Problem getestet werden.

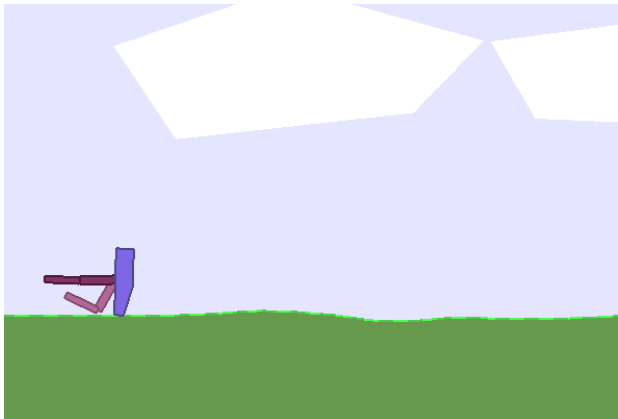


Fig. 1. Ungelernter Bipedal Walker in der Umgebung BipedalWalker-v3

A. Intention

Das bestärkende Lernen ist ein modellfreier Ansatz bei dem ein Agent selbstständig eine, für ihn optimale, Strategie erlernt. Dies geschieht durch eine Voraussage der Belohnung pro Aktion und einer einhergehenden Maximierung dieser.

Erstellt im Rahmen des Moduls *Angewandte künstliche Intelligenz* von Prof. Dr. rer. nat. Frochte

Klassisches Q-Learning baut auf einer Tabelle auf und kann somit nicht generalisieren. Das Deep-Q-Learning hingegen verwendet für die Vorhersage der besten Aktion ein neuronales Netz. Die Größe der Output-Layer ist äquivalent zu der Anzahl der verfügbaren Aktionen, während der Input-Layer an die Anzahl der Informationen, welche die Umgebung zur Verfügung stellt, angepasst ist. Der Agent erhält ausschließlich eine Belohnung für das Zurücklegen von Strecke. Da aber nicht alle notwendigen Bewegungen des Laufens zu einer direkten Vorwärtsbewegung führen, wird ebenfalls die Methodik des Credit Assignment verwendet.

Dieser Ansatz soll mit zwei Algorithmen aus der Familie der Evolutionstechniken verglichen werden. Die Aufgabe dieses Projektes ist es geeignete Algorithmen zu wählen, zu implementieren und sinnvolle Parametrierungen zu ermitteln. Im Anschluss soll der gewählte Ansatz mit dem Deep-Q-Learning verglichen werden. Dabei spielt nicht nur der Grad der Zielerreichung eine Rolle. Es soll ebenfalls verglichen werden, welche Rechenzeit für die Zielerreichung benötigt wurde, wie komplex die Implementierung ist und wie viele Hyperparameter gewählt werden müssen.

B. Aktueller Stand

Zu Beginn dieses Projektes existiert bereits ein Double-Q-Learning Agent mit Credit Assignment für das Mountain-Car-Problem, sowie ein Deep-Q-Agent für die Atari Brick-Breaker Umgebung. Diese sollen zu einem Deep-Q-Agent mit Credit Assignment für den Bipedal Walker zusammengeführt werden. So entsteht voraussichtlich ein Ansatz, welcher generalisieren kann und nicht divergiert. Im Gegensatz zu den oben genannten Problemen besitzt der Bipedal Walker einen kontinuierlichen Aktionsraum, da jeweils die Geschwindigkeit der vier Gelenke in dem Intervall $[-1; 1]$ pro Spielzug eingestellt werden können. Mathematisch dargestellt ist der Aktionsraum \mathcal{A} wie folgt definiert: $\mathcal{A} = [-1; 1]^4$. Die vorhandene Implementation muss auf diesen Umstand angepasst werden.

C. Inhalt

In dieser Arbeit werden die technischen Grundlagen der einzelnen Lösungsstrategien vorgestellt und erarbeitet. Da-

rauffin werden zwei Evolutionsalgorithmen ausgewählt. Es folgt ein theoretischer Vergleich zwischen den Evolutionsalgorithmen und dem Deep-Q-Learning, eine Implementierung der insgesamt drei Ansätze, sowie eine Auswertung und mögliche Validierung der vorausgesagten Ergebnisse. Im Abschluss wird evaluiert, welcher Algorithmus sich besser für den Bipedal Walker eignet.

II. TECHNISCHE GRUNDLAGEN

In diesem Abschnitt sollen die Grundlagen des Q-Learning aufgegriffen werden. Dabei wird eine Herleitung vom einfachen Q-Learning auf einer Tabelle bis hin zum Deep-Q-Learning geschehen. Dieser Abschnitt orientiert sich inhaltlich an den Kapiteln 14 und 15 aus dem Buch *Maschinelles Lernen* von Prof. Dr. rer. nat. Jörg Frochte [1].

Im Anschluss werden die Evolutionsalgorithmen historisch hergeleitet. Dann werden drei verschiedene populäre Strategien vorgestellt und zwei davon zur Implementation gewählt.

A. Q-Learning

Eine Software die auf Basis von definierten Informationen eigenständige Entscheidungen treffen kann wird als Agent bezeichnet. Es gibt unterschiedliche Verfahren um einen solchen Agenten zu entwerfen und zu trainieren. Eines dieser Verfahren ist das bestärkende Lernen. Hierbei erhält der Agent ein Feedback für seine Aktionen in Form von Belohnungen oder Bestrafungen. Das Lernen des Agenten besteht darin, seine Strategie, auch Policy π genannt, hinsichtlich der erhaltenen Belohnung r zu maximieren. Die optimale Strategie ist definiert durch:

$$\pi^* = \operatorname{argmax}_{\pi} V^{\pi}(s) \text{ für alle } s \in S \quad (1)$$

Dabei ist V die Value Funktion, die wiederum durch die Belohnung und den discount factor γ definiert ist [2]:

$$V^{\pi}(s_t) = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \quad (2)$$

1) *Q-Learning mit Tabelle*: Das Q-Learning fällt unter die modellfreien Ansätze. Das bedeutet, dass der Agent keine Informationen darüber hat, wie sich die Umgebung durch seine Aktionen verändert. Das Q-Learning gibt die erwartete Belohnung beim Durchführen einer Aktion a im Zustand s zurück. Dabei wird davon ausgegangen, dass nach dem Durchführen der Aktion eine optimale Strategie verfolgt wird. Daraus ergibt sich die folgende Funktion für das Q-Learning [3] [4]:

$$Q(s, a) = r(s, a) + \gamma V^*(\delta(s, a)) \quad (3)$$

Das δ stellt die Übergangsfunktion der Umgebung dar und drückt hier den Folgezustand aus, der auf den Zustand s durch die Aktion a folgt.

Das Q-Learning bedient sich des *Optimalitätsprinzips von Bellman* [5] und besagt, dass eine optimale Lösung eines Problems aus der Abfolge von optimalen Teillösungen besteht. Für Probleme mit einem kleinen Aktions- und Zustandsraum ist es möglich die Q-Funktion in Form einer Tabelle

darzustellen. Für das Erlernen der Q-Funktion führt ein Agent Aktionen aus und speichert jeweils den Zustand s , die Aktion a , den daraus resultierenden Folgezustand s' und die erhaltene Belohnung r ab. In jedem Schritt nähert der Agent seine Q-Funktion Q' nach folgender Regel weiter an die tatsächliche Q-Funktion an:

$$Q'(s, a) = r + \gamma \max_{a'} Q'(s', a') \quad (4)$$

2) *Q-Learning mit neuronalen Netzen*: Das Verfahren des Q-Learnings auf Basis einer Tabelle aus Abschnitt II-A1 stößt an seine Grenzen sobald der Zustand s oder die Aktion a nicht mehr aus diskreten Werten besteht. Durch das Verwenden von einem neuronalen Netz zum Approximieren der Q-Funktion ist es möglich auch kontinuierliche Werte zu verarbeiten [6]. Die bisherige Vorgehensweise bezieht immer nur die Informationen, die der Agent im letzten Schritt gesammelt hat, in den Lernprozess ein. Allerdings können aus den bereits gesammelten Erfahrungen durch ein wiederholtes Einspeisen in das neuronale Netz Gewichte gefestigt werden. Diese Methodik nennt sich *Experience Replay*. Ein Bündel solcher Informationen wird als *Batch* bezeichnet und das Lernen auf diesem als *Batch-Learning*. Dazu speichert der Agent in jedem Schritt seinen Zustand, die ausgeführte Aktion und die dafür erhaltene Belohnung. In der dadurch entstehenden Tabelle I sind alle Informationen enthalten, die für die Annäherung der Q-Funktion nach Formel 4 notwendig sind [7].

TABLE I
TABELLE ZUR SPEICHERUNG DER GESAMMELTEN INFORMATIONEN DES AGENTEN

Zeitstempel	s	a	r
t_0	s_0	a_0	r_0
t_1	s_1	a_1	r_1
\vdots	\vdots	\vdots	\vdots
t_n	s_n	a_n	r_n

Es ist nicht notwendig in jeder Iteration des Lernens alle vorhandenen Daten zu verwenden. Durch das Bilden von *Mini-Batches* ist es möglich nur Anteile der Daten aus den verfügbaren Informationen zu verwenden.

Ein Problem bei der Verwendung von neuronalen Netzen ist die Wahl der Freiheitsgrade, so wie die Größe der Batches zum Lernen. Verfügt das Netz über zu wenig Freiheitsgrade und sind die Batches zu klein, so verändert sich das Netz beim Lernen auch an Stellen, die nicht der Batch entsprechen. Zu viele Freiheitsgrade des Netzes führen zu einem Overfitting. Das bedeutet, dass der Agent die Fähigkeit zur Verallgemeinerung verliert und nur die gegebenen Informationen im Netz abbildet. Anstatt einer Generalisierung bildet sich beim Overfitting lediglich eine rechenintensive Tabelle. Es ist also sinnvoll eher ein zu kleines, als ein zu großes Netz zu wählen. Zudem kann immensen Veränderungen mit der Einführung einer Lernrate entgegengewirkt werden.

$$Q_{\text{neu}} = (1 - \alpha) Q_{\text{alt}}(s, a) + \alpha(r + \gamma \max_{a'} Q'(s', a')) \quad (5)$$

Diese Lernrate $\alpha \in [0; 1]$ führt dazu, dass die bestehende Approximation der Q-Funktion Q_{alt} für die gegebenen Zustände und Aktionen nicht komplett überschrieben wird.

3) *Deep Q-Learning*: Das Deep-Q-Learning wurde erstmals in einem Paper von Deep-Mind 2015 erwähnt [8]. Dabei geht es jedoch nicht um den Einsatz tiefer Netze für das Q-Learning, da diese bereits in Abschnitt II-A2 verwendet werden können. Es geht viel mehr um die Veränderung der Struktur der Q-Funktion. Bisher bestand der Input immer aus der Aktion plus dem Zustand. Der Output stellt die erwartete Belohnung dar. Das Deep-Q-Learning benötigt lediglich einen Zustand als Input und liefert die erwarteten Belohnungen für alle Aktionen in diesem Zustand als Output.

Das Verfahren benötigt für eine gute Annäherung viele Iterationsschritte. Daher ist es ratsam *Mini-Batches* zum Lernen zu verwenden. Um ein Overfitting auf diese Mini-Batches zu vermeiden ist eine geringe Lernrate sinnvoll.

Ein Problem, das sich durch die Verwendung von Batches bildet ist jedoch, dass für einen Zustand selten Daten zu allen möglichen Aktionen vorliegen. Daher wird das Netz nur für vorliegende Aktionen in dem jeweiligen Zustand angepasst. In Algorithmus 1 ist dies durch das Optimieren hinsichtlich $(y - Q(s_i, a_i))^2$ umgesetzt.

Algorithm 1 Deep Q Learning

- 1: Initialisiere Q-Funktion mit zufälligen Gewichten θ
 - 2: **for** $t \leftarrow 0, 1, 2, \dots$ **do**
 - 3: Bilden einer Mini-Batch aus Tupeln (s_t, a_t, r_t, s_{t+1})
 - 4: $y = \begin{cases} r_j & \text{wenn done=true im Schritt } t+1 \\ r_j + \gamma \max_{a'} Q(s_{t+1}, a') & \text{sonst} \end{cases}$
 - 5: Optimierung von θ bzgl. des Fehlers $(y - Q(s_t, a_t))^2$
 - 6: **end for**
-

4) *Credit Assignment*: Das Problem des Credit Assignments tritt auf, wenn eine große Zeitdifferenz zwischen dem Ausführen einer Aktion und der dafür erhaltenen Belohnung liegt. Der Agent hat in diesem Fall Schwierigkeiten die Belohnung der entsprechenden Aktion zuzuordnen. Um dieses Problem zu umgehen ist es sinnvoll Belohnungen so schnell wie möglich, bestenfalls nach dem Ausführen einer Aktion zu vergeben. Dies ist jedoch nur dann möglich wenn bereits eine generelle Lösungsstrategie bekannt ist. Ist diese nicht bekannt, so erhält der Agent lediglich für das Lösen der gesamten Problemstellung eine Belohnung. Für ein schnelleres Konvergieren der Q-Werte ist es sinnvoll die Gleichung 4 des klassischen Q-Learnings zu erweitern:

$$Q'(s, a) = r + (1 - done)\gamma \max_{a'} Q'(s', a') \quad (6)$$

Das *done* gibt mit einem Wert von null oder eins an, ob das Problem gelöst ist. Durch diese Erweiterung konvergiert das Verfahren deutlich schneller gegen einen Grenzwert [9].

B. Evolutionäre Algorithmen

Evolutionäre Algorithmen sind eine Klasse von Optimierungsverfahren, die an die Evolution natürlicher Lebensformen sowie dem Prinzip *Survival of the fittest* angelehnt sind.

Diese sind stochastisch und metaheuristisch. Dies bedeutet, dass per Zufall ein Problemraum schrittweise erkundet wird, während man sich nach jedem Schritt in die Richtung des zu findenden Extremums bewegt, vergleichsweise zum Bergsteigeralgorithmus [10]. Evolutionsalgorithmen basieren demnach auf dem *Prinzip der lokalen Suche* aus Algorithmus 2.

Algorithm 2 Ablauf der lokalen Suche

- 1: Bestimme eine Startlösung.
 - 2: **while** Startlösung $> \varepsilon$ **do**
 - 3: Definiere eine Nachbarschaft zur Startlösung.
 - 4: Suche diese Nachbarschaft möglichst vollständig ab und bestimme die beste Lösung.
 - 5: **end while**
-

In einem zweidimensionalen Raum ist diese lokale Suche als Evolutionsansatz in der Abbildung 2 von Nicolas Hanssen et al. grafisch dargestellt. Hier ist durch die leichte Streuung der einzelnen Punkte der heuristische Ansatz der Methode gut zu erkennen, es wird also nicht immer die optimale Lösung gefunden. Tatsächlich wird nach einer Lösung gesucht, welche ein Problem hinreichend gut löst. Auf diesen Punkt wird in Kapitel V nochmals eingegangen.

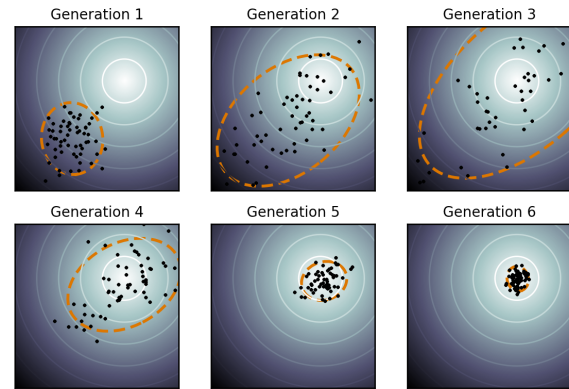


Fig. 2. Lokale Suche eines Evolutionsansatzes über mehrere Generationen [11]

Es ist ebenfalls zu beachten, dass die Evolutionsalgorithmen keinen Gradienten errechnen, sondern Extremstellen durch ein *Trial and Error*-Prinzip finden. Ähnlich zu Gradienten- und weiteren Optimierungsverfahren droht auch hier das Problem der lokalen Extremstellen, welchem man mit sinnvoller Mutation und Rekombination entgegenwirken kann [12]. Auch wird der gesamte Raum zwar möglichst vollständig, aber nicht komplett erkundet. Die meisten Evolutionsalgorithmen haben ihre grundlegende fünf-schrittige Struktur gemeinsam, welche in Algorithmus 3 dargestellt ist. Sie besteht aus einer Initialisierung und einer Generationsschleife, die nach dem *Prinzip der lokalen Suche* so lange durchlaufen wird, bis ein Abbruchkriterium erfüllt ist.

Algorithm 3 Ablauf der Aktionsmutation

```
1: Zufällige Initialisierung einer Population
2: while  $\neg$  Abbruch do
3:   Evaluation
4:   Selektion
5:   Rekombination oder Mutation
6: end while
```

Bei der Evaluation werden die einzelnen Individuen aufgrund einer Funktion bewertet. In den Evolutionsalgorithmen wird hier aufgrund des Prinzips *Survival of the fittest* von einer Fitness-Funktion gesprochen. Tatsächlich unterscheidet sich diese nicht von dem Reward aus dem Reinforcement-Learning weshalb letzterer Begriff projektübergreifend genutzt wird. Anhand der Evaluation werden eine oder mehrere Eltern für die Vererbung ausgewählt. Dabei kann es passieren, dass manche Individuen aufgrund schlechter Leistung ohne Fortpflanzung aussterben. Die nachkommenden Lösungsstrategien werden dadurch verändert, dass Teilstrategien der Eltern miteinander kombiniert werden. Diesen Umstand nennt man Crossover oder Rekombination. Wenn Teile der Strategien zufällig und unabhängig von den Eltern verändert werden, spricht man von Mutation. Beides sind die Suchoperatoren der evolutionären Algorithmen, mit denen der Problemraum erschlossen wird. Durch die Selektion wird einem erfolgreichen Suchprozess die Richtung zum globalen Optimum gegeben. Mithilfe der Mutation können neue Bereiche des Suchraums erschlossen werden, während durch die Rekombination eine Zusammenführung erfolgreicher Teillösungen passiert. Vorteilhaft ist somit die Kombination beider Operatoren. Der Erfolg eines Rekombinationsoperators hängt von der Beschaffenheit der Fitnesslandschaft ab. Mit der Rekombination kann dem Problem der lokalen Extremstellen vorgebeugt werden, indem aus zwei Individuen, die sich auf einem lokalen Optimum befinden, ein Nachfahren im Tal dazwischen entsteht [12].

Historisch gesehen teilen sich die Evolutionsalgorithmen in vier klassische Arten auf, welche im Folgenden vorgestellt werden:

- **Genetische Algorithmen (GA)**

Genetische Algorithmen wurden hauptsächlich durch John H. Hollands [13] bekannt. Sie basieren auf einem Genotyp-Phänotyp-Mapping, mit welchem das Problem durch Rekombination und Mutation der Genome gelöst werden soll. Die genetischen Algorithmen sind damit dem biologischen Vorbild am nächsten. Die Selektion der sich fortpflanzenden Individuen erfolgt über eine Fitness-Funktion, die Fortpflanzung selbst durch *n-Punkt-Crossover*. Die Mutation bei den Genetischen Algorithmen ist anschaulich, da die einzelnen Bits oder Sequenzen der Genome, auch Gene genannt, invertiert, dupliziert oder gelöscht werden können.

- **Evolutionsstrategien (ES)**

In den Evolutionsstrategien wird eine direkte Problemrepräsentationen, zum Beispiel durch reelle Zahlen, verwendet. Der Problem- und der Suchraum sind damit

äquivalent und es findet keine Codierung wie bei den genetischen Algorithmen statt. Somit kann es passieren, dass die ES, wie in Abbildung 2 zu sehen, nicht den gesamten Problemraum erkunden können. Sie nutzen vorrangig die Mutation als Suchoperator, während die Rekombination nur selten verwendet wird. Die Mutation stellt eine Addition eines normalverteilten Wertes dar, wobei die Varianz, die Stärke der Mutation, nicht konstant ist. Da der Algorithmus mit zunehmender Lösungsqualität konvergiert [13] [14], ist es vorteilhaft, die Varianz anzupassen. Dies kann über die Verwendung einer geeigneten und abschwächenden Lernrate passieren. So wird das optimale Ergebnis über die Zeit in immer feineren Schritten gesucht. Für die Anpassung der Varianz hat sich die *Adaptive Anpassung* [15] etabliert. Sie wird auch *1/5-Erfolgsregel* genannt und besagt, dass der Quotient aller Mutationen die eine Verbesserung bei der Problemlösung erzielen ungefähr $\frac{1}{5}$ betragen sollte. Ist der Quotient größer, soll die Varianz erhöht werden, ist sie kleiner soll sie verringert werden. Somit bleibt gewährleistet, dass der Lösungsraum exploriert wird, während eine stetige Verbesserung stattfindet.

- **Genetische Programmierung (GP) und Evolutionäre Programmierung (EP)**

Bei diesen Strategien soll eine symbolische Regression durch die automatisierte Erstellung von Baumstrukturen oder endlichen Automaten durchgeführt werden.

Heutzutage vermischen sich die vier originalen Ansätze zunehmend miteinander. Dennoch sollen die Evolutionsstrategien nach klassischem Beispiel implementiert und an dem BipedalWalker-Problem getestet werden.

1) *Aktionen mutieren*: Ein einfacher Evolutionsansatz in einem Problem mit diskreten Aktionsraum ist es die Abfolge der Aktionen zu mutieren. Dies wird an einem einfachen Labyrinth-Problem veranschaulicht. In dem Problem aus Abbildung 3 beherrscht der Roboter-Hund aus Feld 1 die mögliche Aktionsmenge $\{\text{oben, unten, links, rechts}\}$ der Größe 4. Das Ziel ist es die Zeitung auf Feld 5 möglichst effizient zu erreichen. Das Laufen gegen eine Wand, sowie das Laufen in die Falle in Feld 3 ergeben eine hohe Bestrafung. Für eine Bewegung auf ein leeres Feld erfolgt eine kleine Bestrafung. In einem Evolutionsansatz spielt eine Generation von Roboter-Hunden, beispielsweise 50, mit einer Anzahl von zufällig initialisierten Aktionen das Labyrinth. Im Anschluss werden die gesammelten Rewards miteinander verglichen. Umso besser der Reward, desto wahrscheinlicher ist es, dass die Aktionen des Roboter-Hundes an die nächste Generation weiter vererbt werden. Die Vererbung ist in Abbildung 4 sichtbar.

Hierbei ist zu beachten, dass Kinder der Folgegeneration nicht von zwei Eltern erben, sondern nur von einem Roboter-Hund. Es findet also kein *Crossover* statt. Stattdessen werden die Aktionen des besten Roboter-Hundes direkt vererbt, während alle anderen Aktionsarrays durch einen Mutationsfaktor zufällig variiert werden.

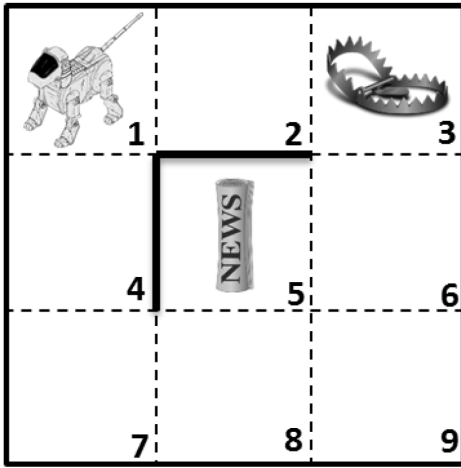


Fig. 3. Roboter-Labyrinth [1, p. 485]

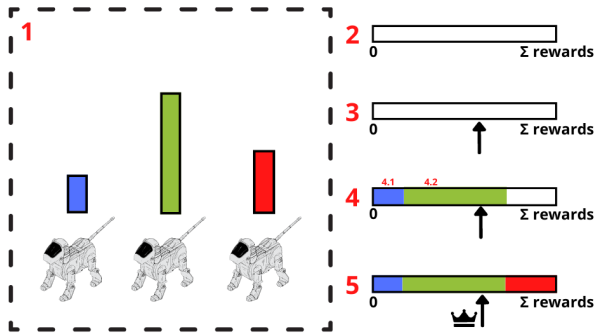


Fig. 4. Vererbung der Aktionen basierend auf dem erzieltm Reward

Die roten Zahlen in der Abbildung 4 verdeutlichen die Reihenfolge der durchgeführten Schritte aus Algorithmus 4.

Dabei kann dieser Algorithmus noch um einen inkrementellen Ansatz erweitert werden, um Rechenzeit zu einzusparen. Um den Effekt zu verdeutlichen wird ein weiteres Mal die Abbildung 3 betrachtet. Ebenso wird davon ausgegangen, dass die Roboter-Hunde gleichzeitig das Labyrinth mit 10.000 zufällig initialisierten Aktionen und einer Zeitspanne von einer Sekunde pro Aktion erkunden. Im Worst-Case-Szenario dauert die Simulation einer Generation nun über zwei Stunden, ohne Garantie, dass das Ziel gefunden wird. Wählt man nun einen iterativen Ansatz, werden die Roboter-Hunde zu Anfang mit einer einzigen zufälligen Aktion initialisiert. Somit ist es zwar unmöglich auf das Feld mit der Zeitung oder der Falle zu treten, allerdings wird in einer Sekunde gelernt, dass die Menge der Aktionen $\{\text{Rechts, Unten}\}$ besser ist als die Menge $\{\text{Links, Oben}\}$. Im Anschluss der Vererbung wird nun der Aktions-Array um eine zufällige Aktion erweitert. Somit ist es relativ wahrscheinlich, dass nach der vierten bis achten Generationen oder 10 bis 36 Sekunden das Labyrinth gelöst wird.

Algorithm 4 Vererbung bei der Aktions-Mutation

```

1: Errechne Rewards  $r_1, \dots, r_n$  der Mutationen  $m_1, \dots, m_n$ 
2: Erstelle Intervall  $i \leftarrow [0, \sum_{i=1}^n r_i]$ .
3: Ermittle zufällige Zahl  $v \in i$ 
4: for  $j \leftarrow 0, 1, 2, \dots, n$  do
5:    $R \leftarrow R + r_j$ 
6:   if  $R > v$  then
7:     Return  $m_j$ 
8:   end if
9: end for

```

Dieser vergleichsweise simple Ansatz funktioniert am besten mit diskreten Aktionsräumen und einer statischen Umgebung. Dies ist beim Bipedal Walker nicht der Fall. Hier wird durch leichte Terrainveränderungen eine dynamische Umgebung simuliert, außerdem können die vier Gelenke des Walkers kontinuierlich im Intervall $[-1; 1]$ eingestellt werden. Dennoch soll dieser inkrementelle Aktions-Evolutions-Ansatz implementiert und getestet werden.

2) *Evolutions-Strategien*: Evolutionsstrategien (ES) sind, wie bereits beschrieben, historisch auf die Evolution aus der Biologie zurückzuführen. Mittlerweile wurden die mathematischen Grundlagen jedoch so weit abstrahiert, dass es besser ist von einer stochastischen Optimierungsmethode als *Black-Box* [16] auszugehen. In jeder Iteration wird eine Generation aus mehreren mutierten Individuen erstellt, welche nach einer Lösung suchen. Anschließend wird für alle Individuen die Güte der Mutation berechnet und die Besten bilden die Grundlage für die Erstellung der nächsten Generation. Die *Black-Box-Optimization*-Algorithmen optimieren Probleme ohne Kenntnisse über die Abläufe in der Umgebung zu besitzen. Die einzigen bekannten Werte sind die gewählten Parameter und das damit erzielte Ergebnis. Das Ziel ist es das Ergebnis bezüglich der gewählten Parameter zu optimieren, ohne einen Zusammenhang zwischen diesen beiden Komponenten zu kennen. Somit sind diese Algorithmen abstrahiert vom eigentlichen Problem und können gut auf andere Probleme transferiert werden.

3) *Natural Evolution Strategies (NES)*: Die Version der Evolutionsstrategien, welche in dieser Arbeit benutzt wird, gehört zu der Klasse der natürlichen Evolutionsstrategien. Sei F die Funktion, die das Verhalten eines Agenten und seine Güte mit den Parametern θ angibt. Die Verteilung der Parameter über eine Population bei den ES ist gegeben durch $p_\psi(\theta)$. Die Verteilung ist abhängig von ψ . Ziel des Verfahrens ist es ein ψ zu finden welches die durchschnittliche Güte der Population maximiert. Um dies zu erreichen wird der stochastische Gradienten Abstieg in der folgenden Form verwendet [16]:

$$\nabla_\psi \mathbb{E}_{\theta \sim p_\psi} F(\theta) = \mathbb{E}_{\theta \sim p_\psi} \{F(\theta) \nabla_\psi \log p_\psi(\theta)\} \quad (7)$$

4) *Natural Evolution Strategies im Reinforcement Learning*: Bei Problemen aus dem Reinforcement Learning ist die Funktion F seitens der Umgebung gegeben. Die Parameter θ sind in diesem Fall Parameter für die Strategie

des Agenten π , vergleichbar zu Abschnitt II. Um die oben angesprochene Optimierung anwenden zu können muss die Umgebung und die Strategie differenzierbar sein. Salimans et al. lösen dieses Problem indem die Verteilung $p_{\psi}(\theta)$ als eine isotrope multivariate Gaussverteilung mit gemitteltem ψ und einer Kovarianz von $\sigma^2 I$ dargestellt wird. Dadurch lässt sich $\mathbb{E}_{\theta \sim p_{\psi}} F(\theta)$ zu $\mathbb{E}_{\epsilon \sim N(0, I)} F(\theta + \sigma \epsilon)$ umformulieren. Durch eine Optimierung im Bezug auf σ ergibt sich aus Gleichung 7 der folgende Zusammenhang:

$$\nabla_{\theta} \mathbb{E}_{\epsilon \sim N(0, I)} F(\theta + \sigma \epsilon) = \frac{1}{\sigma} \mathbb{E}_{\epsilon \sim N(0, I)} \{F(\theta + \sigma \epsilon) \epsilon\} \quad (8)$$

Daraus ergibt sich nach [16] der folgende Ablauf für die Optimierung des θ :

Algorithm 5 Evolution Strategies [16]

- 1: **Input:** Learning rate α noise standard deviation σ , initial policy parameters θ_0
 - 2: **for** $t \leftarrow 0, 1, 2, \dots$ **do**
 - 3: Sample $\epsilon_1, \dots, \epsilon_n \sim \mathcal{N}(0, I)$
 - 4: Compute returns $F_i \leftarrow F(\theta_t + \sigma \epsilon_i)$ for $i \leftarrow 1, \dots, n$
 - 5: Set $\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n\sigma} \sum_{i=1}^n F_i \epsilon_i$
 - 6: **end for**
-

5) *NeuroEvolution of Augmenting Topologies*: NeuroEvolution of Augmenting Topologies (NEAT) [17] ist ein weiterer genetischer Evolutionsalgorithmus welcher eine hohe Popularität [18] genießt. Bei NEAT soll neben optimalen Gewichten auch die Netztopologie alteriert werden. Die Topologie wird in den ES händisch gewählt und ist eine häufige Quelle für suboptimale Ergebnisse.

Vergleichbar zum inkrementellen Ansatz aus Abschnitt II-B1 wird bei NEAT zum Start ein möglichst kleines neuronales Netz verwendet, mit dem die Lösung der Aufgabe wahrscheinlich nicht möglich ist. In den Folgegenerationen wird dieses Netz sukzessiv erweitert.

Durch die sich stetig verändernden Topologien ist es nicht möglich einfache Mutationen oder Crossover-Vererbungen der Aktionen oder Gewichtungen durchzuführen. Stattdessen werden die Netztopologien der Eltern in Form von Genomen codiert. Im Anschluss wird ein *Crossover* durchgeführt. Im Folgenden soll die Vererbung in NEAT anhand von Abbildung 5 beschrieben werden.

Die Architektur eines neuronalen Netzes wird über die Verbindungen der einzelnen Neuronen abgespeichert. Diese werden Nodes genannt und durchnummeriert. Dabei sind die ersten Nodes in den Input- und Output-Layer vorhanden, erst dann werden die Nodes der Hidden-Layer aufgezählt. Tatsächlich werden im Genom die einzelnen Nodes nur implizit abgespeichert, da sich diese vollständig durch ihre jeweiligen Verbindungen und Gewichte beschreiben lassen. Die einzelnen Zellen im Genom, auch genannt Gene, enthalten also die Verbindungen und damit folgende Daten:

- Eine globale Innovationsnummer (engl. innovation number) welche in allen Netzen und in allen Generationen eindeutig ist. Beispielsweise ist in Abbildung 5 die

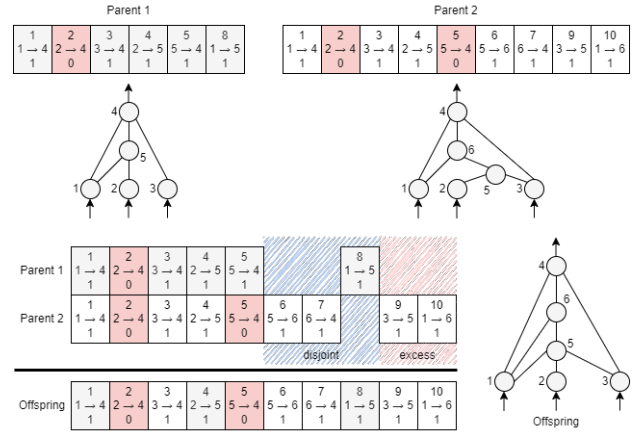


Fig. 5. Crossover in NEAT [17]

Verbindung von Node 1 zu 4 mit der Innovationsnummer 1 versehen und in beiden Elterngenomen gleich. Diese Verbindung wird in allen zukünftigen Generationen ebenfalls die Innovationsnummer 1 besitzen.

- Die Verbindung selbst.
- Die Gewichtung der Verbindung. Diese ist je nach Topologie und Generation unterschiedlich.
- Einen Booleschen Wert, der angibt ob die Verbindung aktiv ist. In Parent 1 ist die Verbindung 2 zwischen den Nodes 2 und 4 deaktiviert, da die tatsächliche Verbindung von Node 2 zu Node 4 über Node 5 stattfindet.

Um das Crossover durchzuführen werden die Verbindungen anhand ihrer Innovationsnummer sortiert und zu einem neuen Genom zusammengeführt. Je nach Kategorie des Gens passiert bei der Vererbung folgendes:

- **Gen ist in beiden Elternteilen vorhanden**
Es wird zufällig eins der beiden Gene ausgewählt.
- **Das Gen ist disjoint**
Ein Gen ist *disjoint*, wenn es nur in einem der beiden Elternteile vorhanden ist, aber danach noch ein separates Gen des anderen Elternteils folgt. In diesem Fall wird das Gen übernommen.
- **Das Gen ist excess**
Wenn das Gen *disjoint* ist, aber kein separates Gen des anderen Elternteils folgt, fällt es unter die Kategorie *excess*. Es kann immer nur ein Elternteil Verbindungen besitzen, welche in diese Kategorie fallen. Die *excess*-Verbindungen werden nur übernommen, wenn der Parent einen höheren Reward erzielt hat. In der Abbildung 5 ist zu erkennen, dass *Parent 2* einen höheren Reward in der Generation hatte, da die *excess*-Verbindungen vererbt wurden.

Nun ist das Crossover durchgeführt. Im nächsten Schritt kann der Offspring mutiert werden. Dafür gibt es fünf verschiedene Mutationsmöglichkeiten.

1) Neue Verbindung generieren

Es wird eine neue Verbindung zwischen zwei Neuronen

mit zufälliger Gewichtung initialisiert. Dabei muss einer bereits bekannten Verbindung zwingend eine bestehende Innovationsnummer zugeordnet werden. Ansonsten wird eine Neue erstellt.

2) **Neuen Node hinzufügen**

Eine bestehende Verbindung wird durch ein neues Neuron aufgeteilt. Dabei wird das die Gewichtung der alten Verbindung übernommen und für die Verbindung vom neuen Neuron zur Folgeschicht eingesetzt. Die Gewichtung der vorherigen Schicht zum Neuron wird zufällig initialisiert. Hier muss eine neue Innovationsnummer erstellt werden.

3) **Verbindungsaktivierung umschalten**

Eine Verbindung wird deaktiviert, wenn sie vorher aktiv war und vice versa.

4) **Verbindung shiften**

Eine Verbindung wird mit einem Mutationsfaktor multipliziert, sodass die Gewichtung leicht verändert wird.

5) **Verbindung re-initialisieren**

Eine bestehende Verbindung wird mit einer komplett neuen Gewichtung initialisiert.

Zuletzt wird bei NEAT die sogenannte Speziation (engl. speciation) zwischen zwei Generationen betrieben. Dabei werden alle Netze anhand ihrer Genome in verschiedene Spezies eingeteilt. Bei der Einteilung wird ein Genom mit dem jeweiligen ersten Genom der Spezies verglichen. Sind diese ähnlich genug, wird es der Spezies hinzugefügt. Kann das Genom keiner Spezies zugeordnet werden, dann wird eine neue Spezies erstellt. Innerhalb einer Spezies werden die einzelnen Individuen nach erhaltenem Reward sortiert. Am Ende jeder Generation wird ein bestimmter Prozentsatz jeder Spezies eliminiert. Dabei kann es sein, dass eine gesamte Spezies ausstirbt. Dennoch wird durch die Speziation sichergestellt, dass verschiedene Lösungswege und -strategien erforscht werden.

6) *Abgrenzung von NEAT zu den ES:* In den Evolution Strategies werden die Gewichte eines neuronalen Netzes durch die einzelnen Mutationen heuristisch optimiert, während alle Mutationen einer Generation in die Gewichtung des Hauptagenten einfließen. Es findet am Ende jeder Generation also eine Verjüngung und kein Crossover zwischen zwei Eltern statt. Weiterhin ist es möglich, dass durch eine Null-Gewichtung einzelne Verbindungen im neuronalen Netz inaktiv sind, dies ist allerdings normalerweise nicht der Fall. Schließlich findet in den ES keine Codierung in Form von Genomen statt.

NEAT besteht durch eine Einteilung diverser Lösungsstrategien in Spezies, welche durch ein Crossover von zwei Parents evolviert werden. Dabei werden die Verbindungen der verschiedenen Netztopologien in Form von Genen in einem Genom abgespeichert und durch klar definierte Regeln vererbt. Auch die Formen der Mutation sind mannigfaltig, und beschränken sich nicht nur auf ein re-initialisieren von einem bestimmten Prozentsatz der Gewichte.

Obwohl NEAT im Vergleich zu den ES wahrscheinlich bessere Ergebnisse erzielen würde sind die Komplexität der

Implementierung und die erwartete Rechenzeit hoch. Auch die Wahl einer geeigneten Konfiguration in NEAT ist vergleichbar mit dem Finden sinnvoller Hyperparameter im Deep-Q-Learning. Diese Arbeit soll jedoch die theoretischen Unterschiede zwischen klassischem Reinforcement Learning und Evolutionsansätzen, welche im Kapitel III erarbeitet werden sollen, validieren. Da diese Unterschiede bei den ES und bei der Aktionsmutation eindeutiger sind, werden diese in Kapitel IV implementiert und in Kapitel V verglichen.

III. THEORETISCHE UNTERSCHIEDE

In dieser Arbeit sollen drei verschiedene Ansätze miteinander verglichen werden: Das Double-Q-Learning, die Evolution Strategies und die inkrementelle Aktionsmutation aus Abschnitt II-B1. Zwischen dem Q-Learning und Evolutionsalgorithmen gibt es theoretische Unterschiede, welche im Kapitel V fundiert werden sollen. Dieses Kapitel ist an [16] und [19] angelehnt.

A. Optimierung der Gewichte

In den Evolutionary Strategies wird, wie beim Q-Learning, die Gewichtung eines neuronalen Netzes optimiert. Beim Q-Learning wird für diese Optimierung ein Gradientenverfahren genutzt. Da die Gewichte der einzelnen Schichten abhängig von ihren Folgeschichten sind, wird für die Gradientenbildung die sogenannte Backpropagation [20] durchgeführt. Diese ist rechenintensiv. In ES findet die Informationsverarbeitung und -weitergabe lediglich vom Input- zur Output-Layer statt. Dadurch sollten circa zwei Drittel der Rechenzeit eingespart werden können. [16] Mit dem Auslassen des Gradientenverfahrens fallen ebenso die Probleme der Wahl einer geeigneten Aktivierungsfunktion [21] und die Gefahr der explodierenden oder verschwindenden Gradienten [22] [23] weg. Ebenso können Aktivierungs- und Lösungsfunktionen genutzt bzw. gefunden werden, welche nicht differenzierbar sind. Dies sorgt für einen schnelleren und weniger komplexen Code und ein robusteres Lernen. Zudem kann die gesamte Implementierung von ES rein in numpy durchgeführt werden. Die Abstraktion der neuronalen Netze durch Bibliotheken wie PyTorch oder Tensorflow fällt weg.

B. Speicherbedarf

Beim Q-Learning werden Erfahrungen gesammelt und dann in Form von Batches in einer Tabelle gespeichert. So können die gesammelten Erfahrungen zum Lernen der Q-Funktion in Form von unsortierten Mini-Batches wiederverwendet werden. Damit die Erfahrungen schnellstmöglich abgerufen werden können, werden sie im Arbeitsspeicher abgelegt. Umso länger das Lernen, desto höher ist der Speicherverbrauch. So kann bei der Wahl einer hohen Gedächtnisgröße beliebig viel Arbeitsspeicher in Anspruch genommen werden. Im Gegensatz dazu werden in den ES die Mutationen des Walkers im Arbeitsspeicher abgelegt. Die Spielzüge der einzelnen Walker müssen nicht abgelegt werden, ein generationenübergreifendes Speichern von Daten ist lediglich wichtig für Auswertungen. Diese können allerdings auch in den permanenten Speicher

geschrieben werden. So ist der Arbeitsspeicherbedarf lediglich von der Populationsgröße anhängig. Durch den geringeren Rechen- und Speicherbedarf ist ES dementsprechend besser für leistungärmere Maschinen geeignet.

C. Robustheit und Reproduzierbarkeit

Q-Learning ist aufgrund der Q-Funktion, sowie der Auswahl von geeigneten Netzwerken, extrem abhängig von der Qualität der ausgewählten Hyperparameter. Jeder Parameter in der Tabelle II kann darüber entscheiden, ob und wie schnell der Agent die Zielerreichung lernt. Im Gegensatz dazu beeinflusst bei den ES hauptsächlich die Lernrate, sowie der Mutationsfaktor den Lernerfolg. Die restlichen Parameter aus Tabelle IV beeinflussen das Lernen nicht so, dass kleine Veränderungen zu gravierenden Unterschieden führen können. Hier kann lediglich die Lernzeit durch eine geschickte Parameterwahl bei gleichbleibendem Erfolg erheblich beeinflusst werden. Durch das, in III-A erwähnte Wegfallen von Tensorflow und PyTorch sollte mit einem festgelegten Seed das Experiment mit gleichem Ausgang wiederholt werden können. Die Erfahrung mit Q-Learning auf anderen Problemen, wie zum Beispiel dem Mountain-Car-Problem, hat gezeigt, dass Experimente mit gleichem Seed und Hyperparametern zu beachtlich unterschiedlichen Ergebnissen geführt haben.

D. Kontinuierliche Aktionsräume

Im Q-Learning wird auf Basis einer Umgebung mit kontinuierlichen (und damit unendlich vielen) Zuständen eine einzige optimale Aktion vorhergesagt. Beispielsweise kann das Auto im Mountain-Car-Problem mit den Aktionen aus der Menge $\{\text{Vorwärts}, \text{Rückwärts}, \text{Nichts tun}\}$ gesteuert werden. Bei dem Bipedal-Walker-Problem werden hingegen alle vier Gelenke des Walkers mit einer Geschwindigkeit im Intervall $[-1; 1]$ gleichzeitig bewegt. Das Q-Learning ist eigentlich nicht für kontinuierliche Probleme vorgesehen. Hier muss getestet werden, ob sinnvolle Aktionen vorausgesagt werden können oder der Aktionsraum mit Präzisionsverlust diskretisiert werden muss. Die Evolutionary Strategies können hingegen kontinuierliche Werte voraussagen.

E. Strukturierte Exploration

In den ES wird in jeder Generation ein zufälliges Rauschen im Aktionsraum erzeugt. Dies macht den Agenten zum einen robuster gegen sogenannte *Frameskips*, zum anderen wird der Aktionsraum zufällig und dadurch gleichmäßig erforscht. Im Q-Learning wird durch den Parameter ϵ -greedy entschieden, ob der Agent eher erkunden oder seine bisher gelernte Strategie einsetzen soll. ϵ -greedy ist damit vergleichbar mit dem Mutationsfaktor, letzterer wird allerdings nicht über die Zeit verkleinert, um ausschließlich die beste Strategie durchzuführen. So können die ES einen Aktionsraum strukturierter und genauer erforschen.

F. Credit Assignment

Beim Q-Learning wird durch die Maximierung der Q-Funktion eine Langzeitstrategie erlernt. Es ist, wie bereits

in Abschnitt II-A4 erwähnt, ein bekanntes Problem, dass extrem späte positive Erfahrungen einen geringen Einfluss auf die Langzeitstrategie haben. Ein bekanntes Problem ist das Mountain-Car-Problem [24], welche ohne eine Anpassung der Reward-Funktion nur schwer lösbar ist. Bei den ES besteht das Problem des Credit Assignment nicht, da kein Markov Decision Process stattfindet.

G. Parallelisierungsmöglichkeiten

Wie bereits in Abschnitt III-A erwähnt wird lediglich numpy für die Implementierung genutzt. Jede mutierte Variante kann durch einen eigenen Worker durchgeführt werden, um den Lösungsraum strategisch und parallel zu erkunden. Da keine Erfahrungen gespeichert werden, besteht in den ES nur am Ende einer Generation Kommunikationsbedarf, um die Leistung der einzelnen Mutationen in Form eines skalaren Wertes zu vergleichen. So ist theoretisch eine bessere Parallelisierung als beim Q-Learning möglich. Experimente haben bewiesen, dass die Lösungszeit umgekehrt linear abhängig zu den eingesetzten Rechenkernen ist [16]. In diesem Experiment werden allerdings keine Parallelisierungsmöglichkeiten implementiert, da der Aufwand der Programmierung voraussichtlich höher ist als die ersparte Zeit.

IV. IMPLEMENTIERUNG

Sowohl der Ansatz des Q-Learnings, als auch der Ansatz auf Basis von ES sind in Python implementiert. Für den Source Code sei hier auf den GitLab-Link <https://gitlab.cvh-server.de/pmaas/bipedal-walker-evo> hingewiesen.

A. Q-Learning

Das Q-Learning ist als Deep-Q-Learning aus Abschnitt II-A3 inklusive dem Batch-Learning aus Abschnitt II-A) implementiert. Der Ablauf des Lernprozesses ist im Algorithmus 6 dargestellt. Ein Problem der Umgebung *BipedalWalker-v3* ist, dass sie lediglich durch ein Umfallen des Walkers vorzeitig beendet wird. Wenn sich der Walker aber in einen Spagat begibt, so sammelt er keine sinnvollen weiteren Informationen mehr aber verharrt in dieser Position bis die maximale Anzahl an Schritten erreicht ist und die Umgebung terminiert wird. Um dieses Problem zu lösen wird in der Schleife der Mittelwert der letzten erhaltenen Rewards berechnet und geprüft ob dieser unter einem gewissen Schwellwert liegt. In diesem Fall ist davon auszugehen, dass sich der Walker in einer stehenden Position befindet und die Umgebung wird vorzeitig terminiert.

Algorithm 6 Grundlegender Ablauf des Q-Learnings

```

1: for trainCycle, maxCycles, ... do
2:   while  $\neg$  done do
3:     get action from walker
4:     execute action in environment
5:     store collected data in memory
6:     learn on batch from memory
7:     if avg of last rewards < threshold then
8:       done  $\leftarrow$  true
9:   end while
10: end for

```

1) *Aufbau des Netzes:* Das Netz besteht neben der Input- und Output-Layer über zwei Hidden-Layers mit 24 Neuronen. Die Aktivierungsfunktion der Hidden-Layers ist die *ReLU*-Funktion. Die Output-Layer hat eine lineare Aktivierungsfunktion. Als Optimizer dient der Adam-Optimizer über den mittleren quadrierten Fehler.

2) *Wahl der Aktion:* Für die Wahl der Aktion wird dem neuronalen Netz der aktuelle Zustand des Walkers übergeben. Das Ergebnis ist ein Vektor mit vier Einträgen, der die Gelenkstellung der vier Gelenke angibt. Zu einer vorgegebenen Wahrscheinlichkeit ε -greedy wählt der Walker statt des Netzergebnisses vier zufällige Gelenkgeschwindigkeiten, um neue Informationen zu sammeln.

Es gibt darüber hinaus die Möglichkeit die vier Werte der Gelenke zu diskretisieren um den Lernvorgang zu beschleunigen.

3) *Bilden der Batches:* Wie in Algorithmus 6 dargestellt speichert der Agent nach jedem Schritt die gesammelten Informationen. Die Informationen sind Tupel der Form $(state_i, action_i, reward_i, done_i, state_{i+1})$ und werden in einem Ringspeicher abgelegt. Somit soll sichergestellt werden, dass das Lernen performant bleibt und Tupel, die dem Lernvorgang hinderlich sind, aussortiert werden.

4) *Optimieren der Q-Funktion:* Der Einsatz des Deep-Q-Learnings bei dem Bipedal-Walker-Problem ist nicht genau nach dem Prinzip aus Algorithmus 1 umzusetzen. Zu Beginn jedes Optimierungsschritts stellt der Walker, vergleichbar zu Abschnitt IV-A3, eine Batch aus seinem Gedächtnis zusammen. Für jeden $state_{i+1}$ in der Batch wird das Maximum des Outputs des neuronalen Netzes Q_{max} bestimmt. Daraufhin wird der Zielvektor der Q-Funktion entsprechend Formel 6 berechnet:

$$y = r + (1 - done) \cdot \gamma \cdot Q_{max} \quad (9)$$

Für das Ziel des neuronalen Netzwerks wird eine Matrix der Größe $Batchsize \times 4$ gebildet. Jede Zeile wird nun mit dem entsprechenden Wert aus dem Zielvektor y gefüllt. Auf diesem Zielvektor führt der Walker genau einen Optimierungsschritt aus.

5) *Wiederherstellung der besten Gewichte:* Für eine genauere Dokumentation werden alle zehn Episoden zehn Testdurchläufe mit dem aktuellen Walker durchgeführt und ein arithmetisches Mittel der erhaltenen Rewards gebildet. Hier wird gleichzeitig geprüft, ob der Walker der Beste des Lernens ist. Am Ende können somit nicht nur die letzten Gewichtungen des Walkers, sondern auch die Besten abgerufen werden.

B. Aktionen evolutionieren

In diesem Abschnitt wird die Implementation des Evolutionssalgorithmus zur Mutation von Aktionen aus Abschnitt II-B1 beschrieben. Dafür wird der Klassenaufbau, ihre jeweilige Funktion und der Ablauf des Lernens veranschaulicht. Der Algorithmus orientiert sich stark an dem Tutorial des Entwicklers Code-Bullet [25] und kann mit leichten Abänderungen auf diverse Probleme angewendet werden. Er eignet sich am besten für diskrete Aktionsräume mit Umgebungen, in denen die Belohnung ein klares Maß der Zielerreichung widerspiegelt. Beim Starten des Lernens wird eine Population in der

gewählten Größe erstellt. Jeder Walker in der Population besitzt beim Initialisieren eine festgelegte Anzahl von Aktionen. Am Ende jeder Generation findet die Vererbung der Aktionen basierend auf dem Reward und ein Inkrement der verfügbaren Aktionen statt. Der Ablauf des Programms ist in Algorithmus 7 zu sehen.

Algorithm 7 Ablauf der Aktionsmutation

```

1: init population with POP_SIZE and BRAIN_SIZE
2: for gen in MAX_GENS do
3:   if all walkers finished then
4:     calculate rewards
5:     create new generation based on rewards
6:     mutate new generation
7:     increment actions
8:   else
9:     update population
10:  end if
11: end for

```

Dabei greift die Main-Funktion nur auf die Klasse Population zu, welche als Verwaltungsklasse für die einzelnen Walker dient. Hier werden nacheinander die Generationen, Walker und Kind-Walker erstellt. Außerdem gibt es Wrapper-Funktionen, welche die selbe Funktion für alle Walker einer Generation durchführen. Auch die Selektion des Parent-Walkers, sowie die Vererbung findet in der Population-Klasse statt.

Die Klasse Walker enthält jeweils eine eigene OpenAI-Environment und Funktionen, um mit dieser zu interagieren. Zuletzt besitzt jeder Walker ein Objekt der Klasse Brain, in welchem sich die durchzuführenden Aktionen sowie die Funktionalität zum Klonen, Erweitern, Mutieren, Laden und Speichern des Aktions-Arrays befinden.

Die Klassen Population, Walker und Brain sind in Abbildung 6 dargestellt. Die Logik der Funktion `select_parent` ist in Abschnitt II-B1 in der Abbildung 4 beschrieben.

C. Evolution Strategies

Der generelle Ablauf der ES ist entsprechend II-B2 umgesetzt. Eine komplette Iteration besteht aus dem Mutieren aller *mutants*, beschrieben in Abschnitt IV-C2, dem Ermitteln der Güte der Mutationen entsprechend Algorithmus 8 und dem anschließenden Optimieren der Gewichte nach Algorithmus 9. Die Logik der Evolution Strategies ist in zwei Klassen unterteilt. Die *Walker* Klasse stellt den Agenten dar, der auf Basis eines neuronalen Netzes und seinem aktuellen Status Aktionen ausführen kann. Die *Population* Klasse entspricht der Population und verwaltet die unterschiedlichen Agenten innerhalb dieser. Auch in diesem Lernverfahren wurde die Möglichkeit zur Wiederherstellung der besten Gewichte aus Abschnitt IV-A5 eingebaut.

1) *Wahl der Aktionen:* Wie bereits erwähnt entscheidet der Agent, welcher durch die Walker Klasse dargestellt wird, auf Basis eines neuronalen Netzwerks. Aktionen können mit Hilfe der Funktion `get_action(state)` erzeugt werden. Das neuronale Netz besteht aus drei Schichten und einem Tangens

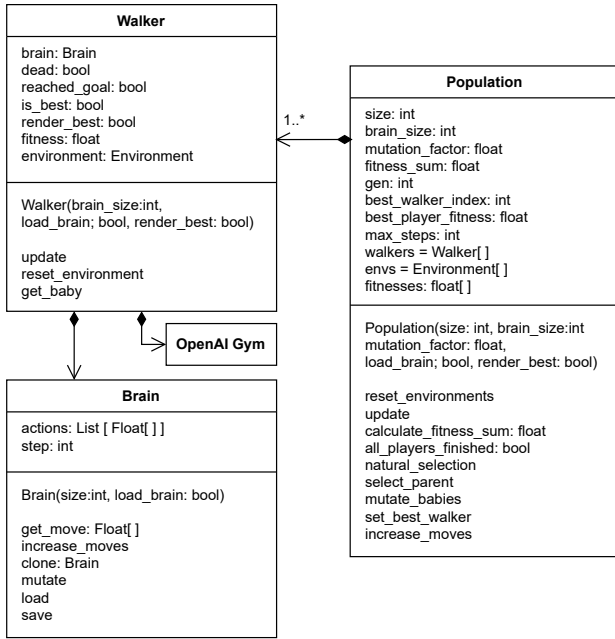


Fig. 6. Klassendiagramm des Evolutionsalgorithmus zur Aktionsmutation

Hyperbolicus mit der Funktionsvorschrift aus Gleichung 10 als Aktivierungsfunktion.

$$[hbt] \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1} \quad (10)$$

Die erste Schicht entspricht den Inputs seitens der Umgebung und die dritte Schicht dem zur Verfügung stehendem Aktionsraum. Die Hidden-Layer ist in ihrer Größe variabel gehalten. Darüber hinaus besteht die Möglichkeit ein Bias Neuron in die Input-Schicht hinzuzufügen.

2) *Mutation*: Die Walker Klasse stellt außerdem eine Funktion zum Mutieren des jeweiligen Walkers zur Verfügung. Dabei werden die Gewichte des Netzes durch den Faktor σ zufällig manipuliert. Die Zufallswerte entsprechen dabei einer Normalverteilung mit Erwartungswert $\mu = 0$ und einer Varianz $\sigma^2 = 1$, also $\mathcal{N}(0, 1)$.

3) *Auswertung der Mutationen*: Um die Walker auswerten zu können, stellt die Klasse eine Funktion zur Verfügung, die einen Durchlauf mit einer definierten Anzahl von Schritten in der Umgebung ausführt. Wie in Algorithmus 8 zu sehen, wird die erhaltene Belohnung in jedem Schritt aufsummiert und schließlich zurückgegeben.

4) *Entwicklung der Population*: Die Population-Klasse verwaltet die einzelnen Walker Objekte. Ein Walker Objekt *walker* entspricht dem aktuellen Stand der Parameter θ . In diesem Fall sind es die Gewichte des neuronalen Netzes. Alle weiteren Walker Objekte *mutants* speichern in jeder Iteration die Mutationen des *walkers*. Die Essenz der Populations-Klasse ist die *evolve*-Funktion aus Algorithmus 9, die bei jedem Aufruf eine Optimierung der Gewichte des *walker*-Objekts vornimmt. Der Kern der Optimierung ist analog zu

Algorithm 8 Ermittlung der Güte eines Agenten

```

1: for  $t \leftarrow 0, 1, \dots, \text{steps}$  do
2:   get action from walker
3:   observation, reward, done  $\leftarrow$  execute action
4:   Set  $r_{t+1} \leftarrow r_t + \text{reward}$ 
5:   if done then
6:     end for
7:   end if
8: end for
9: return  $r_{t+1}$ 

```

Algorithmus 5. Die Funktion setzt voraus, dass bereits alle Mutanten entsprechend Algorithmus 8 ausgewertet wurden.

Algorithm 9 Optimierung der Population

```

1: Input: Vector  $R$  with the total reward of each mutant
2:  $A \leftarrow (R - \bar{R}) / \sigma_R$ 
3:  $\theta_{new} \leftarrow \theta_{walker}$ 
4: for  $i \leftarrow 0, 1, \dots, n \leftarrow \text{population size}$  do
5:    $\theta_{new} \leftarrow \theta_{new} + \frac{\alpha}{\sigma \cdot n} \cdot \theta_{mutant\ i} \cdot A_i$ 
6: end for
7:  $\theta_{mutants, walker} \leftarrow \theta_{new}$ 

```

5) *Visualisierung der neuronalen Netze*: Für die Evolution Strategies wurde eine eigene Visualisierung der neuronalen Netze implementiert. Dem *MLP-Visualizer* werden in der Klasse *Walker* die Größen der einzelnen Layer, sowie die Gewichte übergeben. Diese werden dann in ein Diagramm umgewandelt. So lässt sich beobachten, wie das Netz über die Generationen entwickelt und welche Inputs einen großen Einfluss auf den Walker haben. Zudem kann auch gesehen werden, wenn der Walker nicht lernt, da dann die Verbindungen im Netz wenig ausgeprägt sind.

V. EXPERIMENT

Im Folgenden werden die drei vorgestellten Ansätze auf das Bipedal-Walker-Problem angewendet und hinsichtlich des erzielten Ergebnis unter Berücksichtigung des Aufwands verglichen.

A. Deep-Q-Learning

Für das Deep-Q-Learning wurden die Parameter aus Tabelle II angewendet.

Dabei wurde für die Aktivierungsfunktion die *Rectifier Linear Unit (ReLU)* gewählt. Diese scheint auf den ersten Blick unpassend für dieses Projekt zu sein, da sich für den Aktionsraum $\mathcal{A} = [-1; 1]^4$ die Nutzung eines Tangens Hyperbolicus aus Gleichung 10 anbietet. Der Tangens Hyperbolicus hat jedoch für das bekannte Problem der explodierenden Gradienten [22] gesorgt.

TABLE II
ANGEWENDETE HYPERPARAMETER IM DQN

Parameter	Wert	Intervall
γ	0.99	[0; 1]
α	0.1	[0; 1]
ε_{init}	1	[0; 1]
ε_{low}	0.05	[0; 1]
ε decrease factor	0.999	[0; 1]
bins	7	[3; ∞]
episodes	1	[1; ∞]
epochs	10000	[1; ∞]
batchsize	16	[1; ∞]
memorysize	25000	[1; ∞]
network	[24,24]	[1; ∞]
activation function	ReLU	
activation function output	linear	
optimizer	Adam	
learning rate	0.001	[0; 1]
loss	mean squared error	

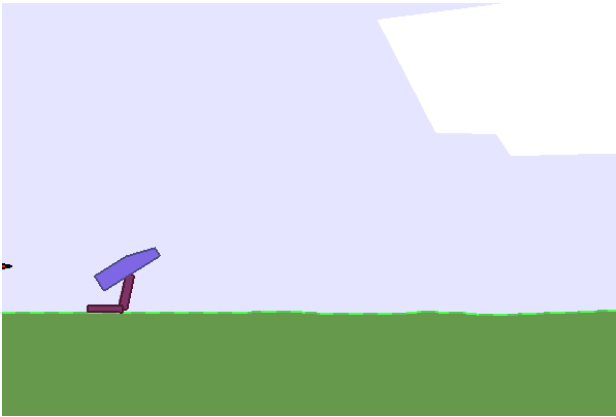


Fig. 7. Walker mit der Kniefall-Strategie

Die Parameter γ , α , ε_{init} , ε_{low} und der ε -decrease-factor haben sich aus Erfahrungen mit diesem und anderen Problemen wie dem LunarLander, dem MountainCar und Atari's Brick Breaker ergeben. Die Batch- und Memorysize wurden hauptsächlich aus Performancegründen gewählt. Erste Lernversuche mit diesen anderen, leicht abgeänderten Parametrierungen waren erfolglos. Zuletzt wurde der Hyperparameter bins eingeführt, mit welchem der kontinuierliche Aktionsraum in sieben feste Aktionen pro Gelenk umgewandelt wird. Dadurch ergeben sich $7^4 = 2401$ verschiedene Aktionskombinationen, die mit einem Zustand, einer Belohnung und einem Folgezustand verbunden und gelernt werden müssen. Durch die Diskretisierung werden die Aktionen des Walkers weniger präzise. Auch hier wäre bins = 11 die intuitivere Wahl, allerdings resultiert diese in $11^4 = 14,641$ verschiedenen Aktionskombinationen, was das Lernen deutlich verlangsamt. Durch die Diskretisierung konnte eine leichte Verbesserung des Lernerfolgs festgestellt werden. Dieser ist in Abbildung 8 zu erkennen. Nach circa 8000 Episoden hat der Walker einen durchschnittlichen Reward von -60 erreicht. Im Gegensatz zur Aktionsmutation aus Abschnitt V-B versucht der Walker keinen Spagat, sondern resultiert in einer Art Kniefall, bei dem

er langsam auf den Boden kippt und schlussendlich mit dem Rumpf auf dem Boden aufschlägt. Dieser ist in Abbildung 7 zu sehen.

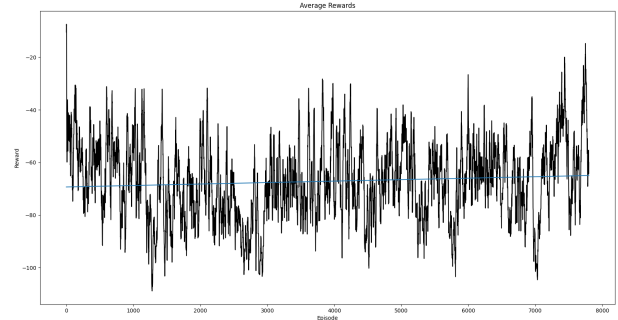


Fig. 8. Leistung des Deep Q Learnings

B. Aktionen Mutieren

TABLE III
ANGEWENDETE HYPERPARAMETER IN DER AKTIONSMUTATION

Parameter	Beschreibung	Interval	Start
INCREASE_BY	Inkrement Schritte / Gen	[1; ∞]	5
BRAIN_SIZE	Schritte in Gen 0	[1; 1600]	50
n	Größe der Population	[0; ∞]	50
σ	Mutationsfaktor %	[0; 1]	0.2
GENS	#Episoden	[0; ∞]	2k - 7k

Das Mutieren der Aktionen benötigt die geringste händische Parametereinstellung. Hier wurden bereits in vergangenen Projekten diverse Möglichkeiten der Parametrierung untersucht. Die Konfiguration aus Tabelle III stellt demnach einen guten Kompromiss aus Lernerfolg und Rechenzeit dar.

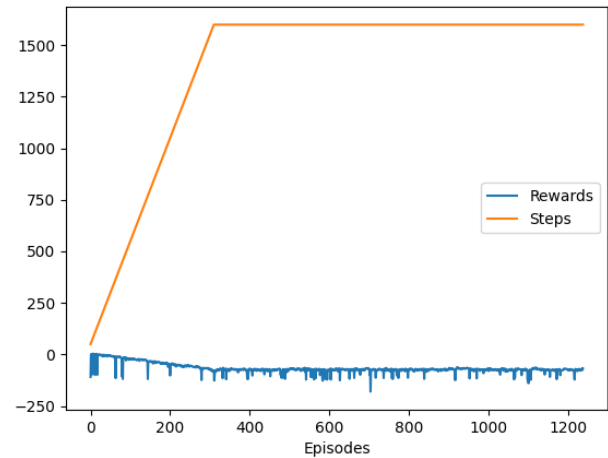


Fig. 9. Leistung des Algorithmus zur Mutation der Aktionen

Wie bereits in Abschnitt II-B1 vorausgesagt ist dieser Algorithmus nicht zum Lösen des Bipedal-Walker-Problem geeignet. Innerhalb von wenigen Episoden werden jeweils zwei Zustände gelernt: Das direkte Hinfallen für einen Reward

von -100 oder das Warten im Spagat für 0 Punkte. Diese Zustände sind abhängig von der Startposition und dem Terrain. Der erspielte Reward sinkt proportional zu den durchgeführten Bewegungen. Dies liegt daran, dass der Walker seine Gelenke leicht zitternd bewegt, um marginal Distanz zurückzulegen. Dabei ist die Bestrafung durch die Bewegung höher als die Belohnung für die zurückgelegte Distanz. Bei dem Erreichen der maximal verfügbaren Schritte stabilisiert sich der durchschnittliche Reward bei circa -60.

Obwohl dieser Algorithmus das Problem nicht lösen konnte ist es beachtlich, dass die Aktionen der Spagat aus Abbildung 10 in wenigen Episoden und damit Sekunden gelernt wurden. So erreicht er die selbe Leistung wie das Deep-Q-Learning in einem Bruchteil der eingesetzten Zeit, Ressourcen und Komplexität.

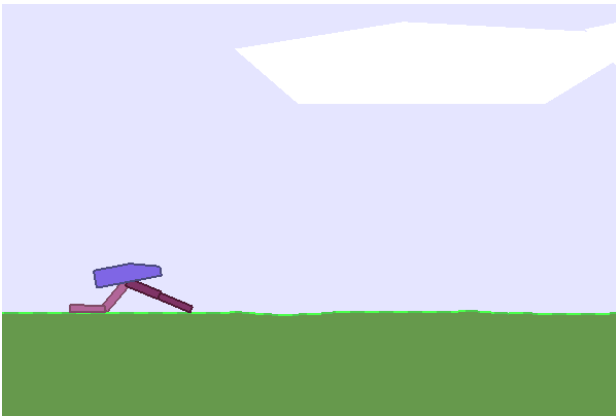


Fig. 10. Verhalten des Bipedal Walkers nach wenigen Episoden

C. Evolution Strategies

In den Evolution Strategies müssen die Hyperparameter aus Tabelle IV händisch zu Beginn des Experiments erzeugt werden.

TABLE IV
ANGEWENDETE HYPERPARAMETER IN ES

Parameter	Beschreibung	Interval	Start
HL	Größe der Hidden Layer	$[1; \infty[$	12
BIAS	Verwendung im Input-Layer	0, 1	0
n	Größe der Popluation	$[0; \infty[$	50
σ	Mutationsfaktor	$[0; 1]$	0.1
α	Lernrate	$[0; 1]$	0.1
GENS	# Episoden	$[0; \infty[$	2k - 7k
MAX_STEPS	# Schritte / Gen	$[0; 1600]$	300

Es wurde herausgefunden, dass das (De-)Aktivieren des Bias-Neurons in diesem Experiment keinen Einfluss auf den Lernerfolg hatte. Der Parameter MAX_STEPS wurde nach wenigen Versuchen auf von 1600 auf 300 gesenkt. So wird also nur circa $\frac{1}{5}$ einer gesamten Episode gespielt, was proportional viel Rechenzeit einspart. Dies ist möglich, da der Agent lediglich eine Belohnung für eine effektive Vorwärtsbewegung erhält. So wurden nur 18.75% der Rechenzeit für vergleichbare Lernerfolge verwendet. Im gleichen Zuge wurde eine

Populationsgröße von 50 identifiziert, welche einen guten Kompromiss aus Lernerfolg und Rechenzeit darstellt.

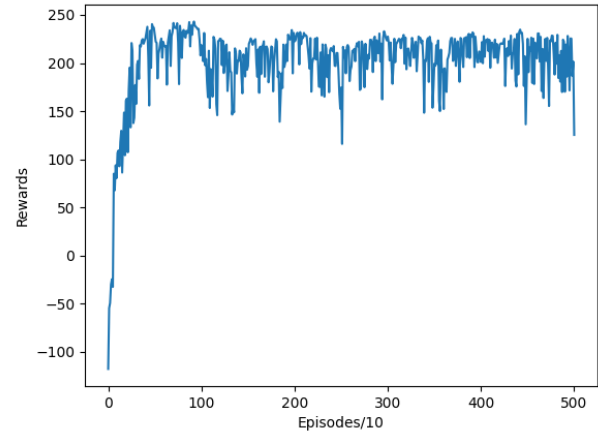


Fig. 11. Parameter zum Start des Experimentes nach Tabelle IV

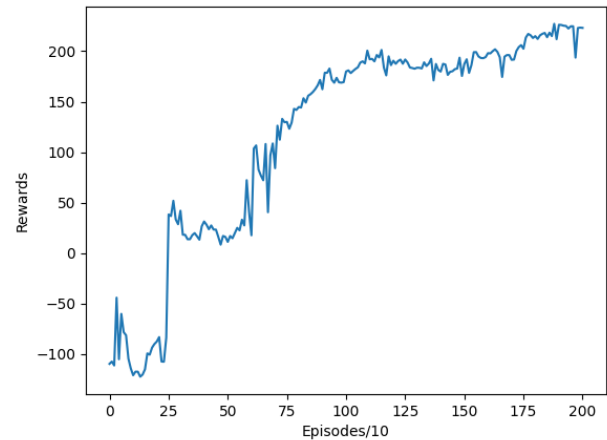


Fig. 12. Experiment ES mit einer Mutationsrate σ von 0.3

Für die Größe der Hidden Layer wurde die Faustregel angewendet, dass die Anzahl der Neuronen die Hälfte der vorherigen Schicht oder das Doppelte der Folgeschicht betragen sollen. Da es nur eine Hidden Layer gibt, ist laut diesem heuristischen Ansatz eine Größe aus dem Intervall $[8, 12]$ sinnvoll. Tatsächlich haben die durchgeführten Experimente dies bestätigt. Bei einer Hidden Layer > 12 ist es teilweise zu einem Overfitting gekommen, bei denen der Walker auf die leichten Bodenveränderungen bei neuen Episoden nicht mehr reagieren konnte. Bei einer Hidden Layer < 8 hingegen sind zu viele Informationen der Observation verloren gegangen, sodass keine Laufbewegung entstanden ist. Somit fehlt nur noch eine Optimierung der Mutations- und Lernrate. Basierend auf den Parametern in Abbildung 11 mit einem Vergleichswert von > 250 wurde eine feinere Auswertung angefangen.

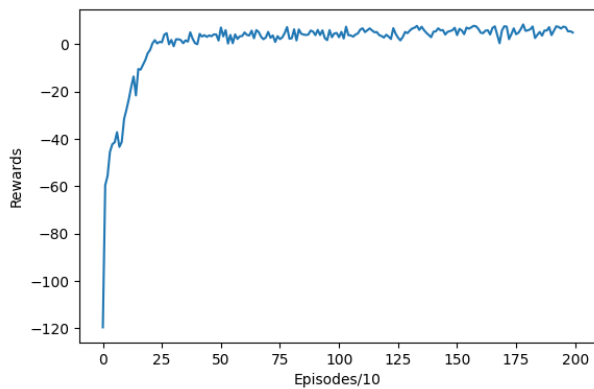


Fig. 13. Experiment ES mit einer Mutationsrate σ von 0.05

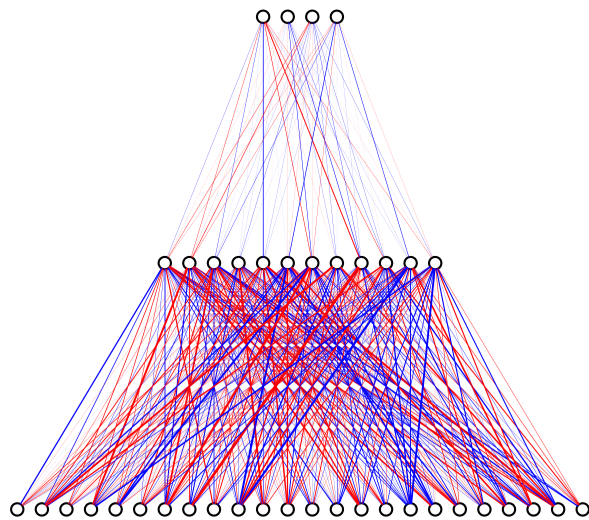


Fig. 14. Neuronales Netz bei einer Mutationsrate σ von 0.05

Als nächstes wurde, wie in Abbildung 12 sichtbar, die Mutationsrate σ vergrößert. Es ist über den Verlauf von 2000 Episoden eine Verbesserung zu erkennen. Allerdings wird die Grenze der Rewards > 200 erst nach 1000 Episoden teilweise erreicht und nach 1750 Episoden konstant überstiegen. Der Verlauf ist allgemein inkonsistent und die Belohnungen sind schlechter als der Initiale Durchlauf aus Abbildung 11. In Abbildung 13 wurde die Mutationsrate verkleinert. Hier konnte der Walker nicht lernen das lokale Extremum bei einem Reward von 0 zu überwinden, allerdings sind die Schwankungen geringer. Dass das lokale Maximum nicht überwunden werden kann lässt sich auch gut in Abbildung 14 erkennen. Hier sieht man starke Verbindungen zwischen Input- und Hiddenlayer, allerdings sind die Gewichte zwischen Hidden- und Output-Layer im Vergleich extrem schwach. Dies bedeutet, dass der Input so gut wie keinen Einfluss auf den Output hat und der Walker nur versucht den Spagat durchzuführen.

Auch eine automatische Mutationsrate nach der $1/5$ -

Erfolgsregel [15] aus Abschnitt II-B hat keine Verbesserung der Ergebnisse erzielt.

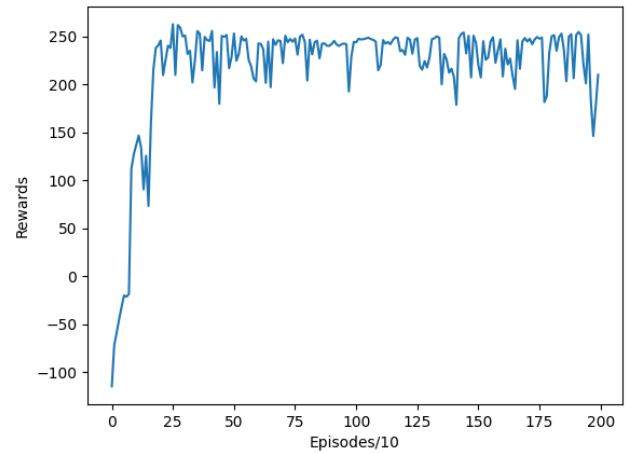


Fig. 15. Experiment ES mit einer Lernrate α von 0.2

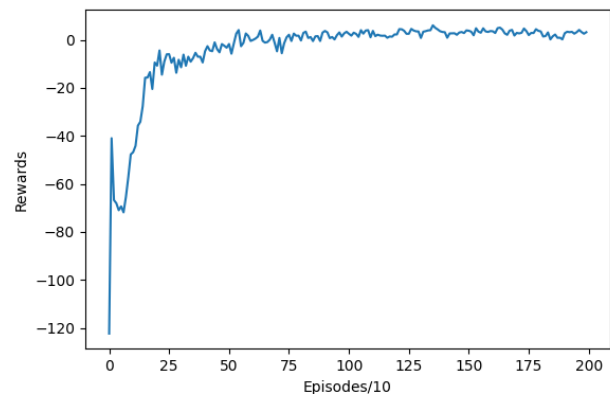


Fig. 16. Experiment ES mit einer Lernrate α von 0.05

Im Anschluss wurde in den Abbildungen 15 und 16 die Lernrate variiert. Bei einer konstanten Lernrate von einem halben Prozent hat der Walker, wie auch bei einer geringen Mutationsrate, das lokale Extremum des Spagats nicht verlassen können. Es ist allerdings davon auszugehen, dass im späteren Verlauf eine geringe Lernrate zu einer leichten, aber stetigen Verbesserung des Walkers führen kann, sobald das lokale Extremum überwunden wurde. Eine Lernrate von 20% hat zu einer schnellen Überwindung des lokalen Maximums geführt. Im Anschluss zeichnet sich das Lernen durch hohe Schwankungen aus. Somit war es nicht möglich deutlich mehr als 250 Punkte zu erreichen.

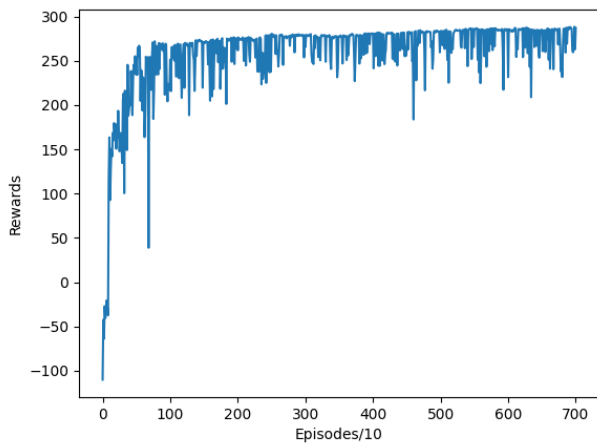


Fig. 17. Experiment ES mit einer fallenden Lernrate α über 7000 Generationen

Die schnelle Überwindung des lokalen Maximums sollte mit einem kontinuierlichen Lernerfolg und den Parametern aus dem Startexperiment verbunden werden. Somit wurde ein fallendes α eingeführt, welches bei 0.1 beginnt und nach 1000 Episoden den Wert 0.05 erreicht. Die Folge ist ein beinahe logarithmischer Lernerfolg aus Abbildung 17. Hier konnte nach 7000 Generationen ein durchschnittlicher Reward über 290 Punkten erreicht werden und eine Betrachtung der Netzvisualisierung zeigt, dass die Informationen aller Inputs gleichmäßig an der Bewegung des Walkers beteiligt sind.

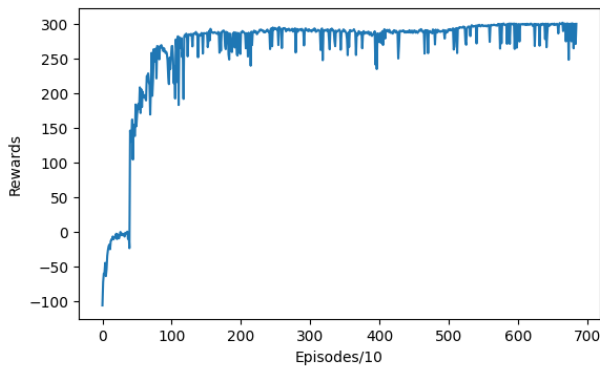


Fig. 18. Experiment ES mit einer fallenden Lernrate α und σ unter 7000 Generationen

Letztendlich wurde neben dem fallenden α auch ein fallendes σ eingeführt. Beide Parameter starten bei 0.1 und werden nach 1000 Episoden auf 0.05 gesetzt. Zudem werden nach 5000 Episoden beide Parameter auf 0.01 reduziert. Mit diesen und den restlichen Parametern aus Tabelle IV, ist es dem Walker unter 7000 Episoden gelungen einen durchschnittlichen Reward von über 300 Punkten zu erreichen. Die Lernkurve kann in Abbildung 18 gesehen werden. Hier sieht man zu Anfang ein längeres Verweilen am lokalen Extremum bei einem Reward von 0. Im späteren Verlauf wird stetig dazugelernt. Dabei entstehen durch das reduzierte σ

im Gegensatz zu Abbildung 17 weniger Schwankungen. Der Walker hat entsprechend der Heuristik der ES eine Bewegung erlernt, welche nicht dem typischen Bild des Laufens auf zwei Beinen entspricht. Das hintere Bein wurde für eine verbesserte Stabilität abgewinkelt. Während das vordere Bein für einen großen Schritt wie in Abbildung 19 angehoben wurde, hat sich das hintere in schneller Folge zweifach vom Boden abgestoßen. Erst dann hat das vordere Bein wieder Bodenkontakt.

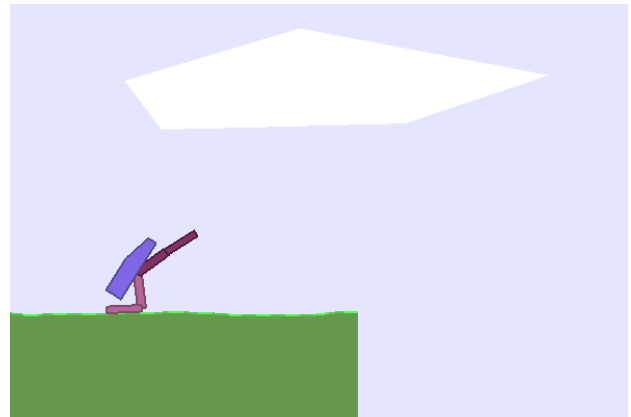


Fig. 19. Walker am Ende der Episode

Auch mit den Evolution Strategies wurde das Bipedal-Walker-Problem nicht offiziell gelöst. In Abbildung 20 kann gesehen werden, dass bei 50 durchgeführten Episoden mehrmals die 300-Punkte-Marke unterschritten wird.

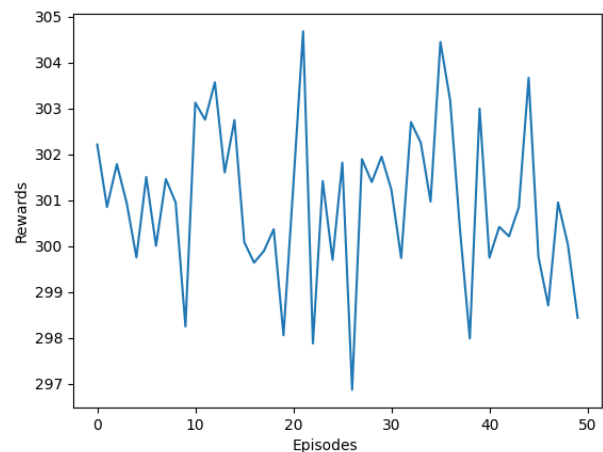


Fig. 20. Leistungs des ausgelernten Walkers über 50 Episoden

Dennoch ist das Ergebnis deutlich besser als beim Deep-Q-Learning. Auch in dem Punkt Leistung sind die ES effektiver. Hier wurde für die Durchführung einer Generation ungefähr 4 Sekunden Rechenzeit benötigt. So konnten 1000 Generation mit einem erreichten Reward 250 in circa einer Stunde simuliert werden. Beim Deep-Q-Learning wird für eine Epoche, was einer Generation entspricht, 3 Sekunden Rechenzeit benötigt. Diese geringere Rechenzeit ist darin begründet,

dass die Batch lediglich eine Größe von 16 hat und der Walker nur weniger Schritte durchführt, da er oft mit dem Rumpf den Boden berührt. Außerdem wird pro Epoche nur eine einzige Annäherung der Q-Funktion vorgenommen. Die CPU-Last, der Speicherverbrauch, die Anzahl der wählbaren Parameter, sowie der erhaltene Reward der ausgewählten Walker sind in Tabelle V dargestellt. Die Rechenzeit für das Erreichen der aufgeführten Rewards sind bei dem Deep-Q-Learning ca. acht Stunden, bei der Aktionsmutation nur wenige Minuten und bei den Evolution Strategies ca. sieben Stunden.

TABLE V
VERGLEICH DER VERFAHREN

Verfahren	CPU-Last	Speicher	Parameter	Reward
Deep-Q-Learning	22,5 %	217,3 MB	16	~ -60
Aktionsmutation	21,8 %	74,6 MB	5	~ -60
Evolution Strategies	19,1 %	52,6 MB	7	~ 300

Schließlich ist zu beachten, dass in der Implementierung der ES ausschließlich numpy genutzt wird. Es findet also keine Abstrahierung der zugrunde liegenden Mathematik statt, durch beispielsweise die Verwendung von Keras für die Abbildung des neuronalen Netzes. Damit ist in den Augen der Autoren der Code verständlicher, was einen subjektiven Vorteil ergibt.

VI. FAZIT

Q-Learning-Ansätze sind für das Bipedal-Walker-Problem nicht gut geeignet. Dies liegt vor Allem daran, dass der Aktionsraum kontinuierlich ist. Somit hat der Q-Agent durch einen Kniefall gelernt seinen Misserfolg hinauszuzögern und einen Reward von ca. -60 erzielt. Die Lernzeit betrug dabei mehrere Stunden. Im Gegensatz dazu wurden zwei Evolutionsansätze ausprobiert. Der Erste mutiert inkrementell die durchgeführten Aktionen des Walkers, um somit die optimale Abfolge an Aktionen zu erlernen. Auch hier besteht das Problem des kontinuierlichen Aktionsraums. Zudem sorgen leichte Terrainveränderungen dafür, dass der Walker keine perfekte Aktionsfolge lernen kann, sondern generalisieren muss. Dennoch war die Leistung dieses Prototypen vergleichbar mit dem Deep-Q-Learning, ohne eine komplexe Implementierung, hohe eingesetzte Rechenzeit oder der Wahl vieler Hyperparameter.

Darauf aufbauend wurde der von OpenAI vorgestellte Algorithmus aus dem Gebiet der Evolution Strategies adaptiert, in welchem die Gewichtung eines neuronalen Netzes durch Mutation erlernt wird. Durch die Nutzung eines neuronalen Netzes und der Vorhersage der Aktionen durch ein Output-Neuron pro Aktion wird den oben genannten Problematiken vorgebeugt.

Obwohl die ES die offizielle, strenge Sieges-Bedingung des Bipedal Walkers nicht kontinuierlich erfüllt hat, sind die Vorteile gegenüber des Deep-Q-Learning klar ersichtlich. Mit einem durchschnittlichen Reward von 250 Punkten nach nur einer Stunde Lernzeit, bringt ES eine bessere Leistung bei weniger eingesetzter Rechenzeit. Zudem ist die Implementierung sowohl in der Komplexität des Codes, als auch in der zugrunde liegenden Mathematik einfacher. Gleichzeitig

müssen weniger Hyperparameter gewählt werden, was für eine höhere Robustheit des Ansatzes sorgt. Für den Bipedal Walker sind die ES auf Basis des Experiments die bessere Wahl.

VII. AUSBLICK

Sowohl Q-Learning-Ansätze, als auch ES sind übertragbar auf andere Probleme. Hier ist ein Vergleich in weiteren OpenAI-Umgebungen interessant. Der Implementierte ES-Ansatz lässt sich bereits ohne Änderungen auf andere kontinuierliche Probleme anwenden. So wurde das Lunar-Lander-Problem bereits mit leichter Re-parametrierung zuverlässig und offiziell gelöst. Für diskrete Aktionsräume ist ein weiterer Implementationsaufwand erforderlich. Zudem wurde keine Parallelisierung für das Bipedal-Walker-Problem eingebaut. Untersuchungen zeigen, dass die Anzahl der eingesetzten Rechenkerne umgekehrt linear zur Lösungszeit steht [19]. Ergo bedeutet eine höhere Parallelisierung eine geringere Zeit zur Zielerreichung.

Zuletzt wäre ein Vergleich in den selben Aspekten mit weiteren gängigen Verfahren wie das Actor-Critic- oder Policy-Gradient-Methoden interessant. Hier ist anzunehmen, dass die Methoden eine bessere Zielerreichung als ES haben, aber sowohl in der Lösungszeit als auch in Komplexitätspunkten deutlich schlechter abschneiden.

REFERENCES

- [1] Jörg Frochte. *Maschinelles Lernen: Grundlagen und Algorithmen in Python*. ger. 3., überarbeitete und erweiterte Auflage. Plus.Hanser-Fachbuch. München: Hanser, 2021. ISBN: 9783446461444.
- [2] Csaba Szepesvári, Jozsef Attila, and Michael Littman. "Generalized Markov Decision Processes: Dynamic programming and Reinforcement-learning Algorithms". In: (Dec. 1996).
- [3] Christopher Watkins. "Learning From Delayed Rewards". In: (Jan. 1989).
- [4] Christopher J. C. H. Watkins and Peter Dayan. "Q-learning". en. In: *Machine Learning* 8.3-4 (May 1992), pp. 279–292. ISSN: 0885-6125, 1573-0565. DOI: 10.1007/BF00992698. URL: <http://link.springer.com/10.1007/BF00992698> (visited on 03/01/2022).
- [5] Richard Bellman. *Dynamic programming*. Princeton, NJ: Princeton Univ. Pr, 1984. ISBN: 9780691079516.
- [6] Martin Riedmiller. "Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method". In: *Machine Learning: ECML 2005*. Vol. 3720. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 317–328. ISBN: 9783540292432 9783540316923. DOI: 10.1007/11564096_32. URL: http://link.springer.com/10.1007/11564096_32 (visited on 03/01/2022).
- [7] Marco Wiering and Martijn van Otterlo, eds. *Reinforcement learning: state-of-the-art*. Adaptation, learning, and optimization v.12. OCLC: ocn768170254. Heidelberg ; New York: Springer, 2012. ISBN: 9783642276446 9783642276453.

- [8] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". en. In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 1476-4687. DOI: 10.1038/nature14236. URL: <https://www.nature.com/articles/nature14236> (visited on 03/01/2022).
- [9] Vyacheslav Alipov et al. "Towards Practical Credit Assignment for Deep Reinforcement Learning". In: *arXiv:2106.04499 [cs]* (Feb. 2022). arXiv: 2106.04499. URL: <http://arxiv.org/abs/2106.04499> (visited on 03/01/2022).
- [10] Stuart J. Russell, Peter Norvig, and Ernest Davis. *Artificial intelligence: a modern approach*. 3rd ed. Prentice Hall series in artificial intelligence. Upper Saddle River: Prentice Hall, 2010. ISBN: 9780136042594.
- [11] Nikolaus Hansen. "The CMA Evolution Strategy: A Comparing Review". en. In: *Towards a New Evolutionary Computation*. Vol. 192. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 75–102. ISBN: 9783540290063 9783540324942. DOI: 10.1007/3-540-32494-1_4. URL: http://link.springer.com/10.1007/3-540-32494-1_4 (visited on 03/01/2022).
- [12] William M Spears. *Evolutionary Algorithms The Role of Mutation and Recombination*. German. OCLC: 863879995. 2000. ISBN: 9783662041994. URL: <https://doi.org/10.1007/978-3-662-04199-4> (visited on 03/01/2022).
- [13] John H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. 1st MIT Press ed. Complex adaptive systems. Cambridge, Mass: MIT Press, 1992. ISBN: 9780262082136 9780262581110.
- [14] Clayton L Bridges and David E. Goldberg. "An analysis of reproduction and crossover in a binary-coded genetic algorithm". In: *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*. USA: L. Erlbaum Associates Inc., Oct. 1987, pp. 9–13. ISBN: 9780805801583. (Visited on 03/01/2022).
- [15] Ingo Rechenberg. *Evolutionssstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. ger. Problemata, 15. Stuttgart-Bad Cannstatt: Frommann-Holzboog, 1973. ISBN: 9783772803734.
- [16] Tim Salimans et al. "Evolution Strategies as a Scalable Alternative to Reinforcement Learning". In: *arXiv:1703.03864 [cs, stat]* (Sept. 2017). arXiv: 1703.03864. URL: <http://arxiv.org/abs/1703.03864> (visited on 03/01/2022).
- [17] Kenneth O. Stanley and Risto Miikkulainen. "Evolving Neural Networks through Augmenting Topologies". en. In: *Evolutionary Computation* 10.2 (June 2002), pp. 99–127. ISSN: 1063-6560, 1530-9304. DOI: 10.1162/106365602320169811. URL: <https://direct.mit.edu/evco/article/10/2/99-127/1123> (visited on 03/01/2022).
- [18] Hunter Heidenreich. *NEAT: An Awesome Approach to NeuroEvolution*. en. Jan. 2019. URL: <https://towardsdatascience.com/neat-an-awesome-approach-to-neuroevolution-3eca5cc7930f> (visited on 03/01/2022).
- [19] *Evolution Strategies as a Scalable Alternative to Reinforcement Learning*. en. Mar. 2017. URL: <https://openai.com/blog/evolution-strategies/> (visited on 03/01/2022).
- [20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. Adaptive computation and machine learning. Cambridge, Massachusetts: The MIT Press, 2016. ISBN: 9780262035613.
- [21] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. "Searching for Activation Functions". In: *arXiv:1710.05941 [cs]* (Oct. 2017). arXiv: 1710.05941. URL: <http://arxiv.org/abs/1710.05941> (visited on 03/01/2022).
- [22] Y. Bengio, P. Simard, and P. Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* 5.2 (Mar. 1994), pp. 157–166. ISSN: 1045-9227, 1941-0093. DOI: 10.1109/72.279181. URL: <https://ieeexplore.ieee.org/document/279181/> (visited on 03/01/2022).
- [23] Sepp Hochreiter. "Untersuchungen zu dynamischen neuronalen Netzen". In: (Apr. 1991).
- [24] Andrew W. Moore. "Efficient memory-based learning for robot control". In: 1990.
- [25] Code Bullet. *How AIs learn part 2 — Coded example*. May 2018. URL: https://www.youtube.com/watch?v=BOZfhUcNiqk&ab_channel=CodeBullet.