Informatik
Hauptcampus
HOCHSCHULE TRIER

Visualisierung von Reinforcement Learning Methoden und Artificial Neural Networks

Jonas Wild

Bachelor-Abschlussarbeit

Betreuer: Prof. Dr. Hans-Peter Beise

Trier, 26. Juni 2020

Kurzfassung

Reinforcement Learning ist ein Methodenkollektiv mit dem künstliche neuronale Netze trainiert werden können. Es bestärkt gute Aktionen und schwächen schlechte. Der Vorteil dieser Methoden ist, dass sie nicht auf gelabelte Datensätze angewiesen sind, sondern zur Laufzeit eigens Lernvektoren generieren. Die Komplexität von Reinforcement Learning mit künstlichen neuronalen Netzwerken ist gewaltig, weshalb eine Anwendung notwendig wird, die den Einstieg in die Thematik vereinfacht. Die Visualisierung neuronaler Netzwerke, eine Parameteradjustierung auf einer graphischen Oberfläche sowie das Betrachten der Lernkurve in Echtzeit werden den Anforderungen an solch eine Anwendung gerecht. In der Anwendung können verschiedene Probleme mit unterschiedlichen Reinforcement Learning Methoden gelöst werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

1

Einleitung

Maschinelles Lernen ist seit einiger Zeit Teil unseres Alltags. Bildanalysetools mit künstlicher Intelligenz werden von den verschiedensten Applikationen implementiert. Jede Eingabe kann eindeutig einer Ausgabe zugeordnet werden. Bilder von Hunden und Katzen können von Menschen als solche identifiziert werden. Deshalb kann Programmen erfolgreich beigebracht werden, diese zu klassifizieren.

Es gibt aber auch Probleme, an denen Menschen eine Eingabe nicht optimal einer Ausgabe zuordnen können. Im Spiel *Go* gibt es 10^{170} verschiedene Ausgangssituationen, mehr als es Atome im Universum gibt [All94]. Für Menschen ist es praktisch unmöglich, jede einzelne zu bewerten. Die optimale Lösung kann also nicht mehr vom Menschen erfasst werden, sondern muss von dem System selbst erfahren werden. Die derzeit vielversprechendste Methode hierfür stellt *Reinforcement Learning* dar. Der erste große Durchbruch auf diesem Feld erfolgte 2017 durch ein Programm von *Deepmind*: *AlphaZero*. Innerhalb von 24 Stunden erreichte die künstliche Intelligenz übermenschliche Fähigkeiten in drei der bekanntesten strategischen Brettspiele (Schach, Shogi und Go) [SHS+18]. Das gleiche Forschungsteam verbesserte sich zwei Jahre später nochmals, als das Programm *AlphaStar* professionelle Spieler in einem der komplexesten Computerspiele unserer Zeit übertraf. Das System entscheidet sich in dem Echtzeitstrategiespiel *Starcraft* 2 in jedem Schritt zwischen 10^{26} Möglichkeiten [VBC+19]. Mit dem Ziel, die Systeme in die reale Welt zu übertragen, haben sich computeranimierte Umgebungen durch eine Vielzahl von Vorteilen, insbesondere aber dem sicheren und kontrolliertem Testen, als präferiertes Forschungsobjekt im Bereich des *maschinellen Lernens* entwickelt.

In den ersten zwei Kapiteln werden Grundlagen künstlicher neuronaler Netze sowie Funktionsweise und Chancen von *ReinforcementLearning* erläutert. Im Hauptteil der Arbeit wird gezeigt, wie *Reinforcement Learning* mit künstlich neuronalen Netzen visualisiert werden kann. Zuletzt werden im experimentellen Teil mit Hilfe der Visualisierung verschiedene *Reinforcement Learning* Erfolgsfaktoren beleuchtet.

2

Künstliche neuronale Netzwerke

Menschliches Lernen ist biologisch gesehen eine Anpassung unseres Gehirns durch aufgenommene Information. Durch wiederholte Aktivierung bestimmter Neuronen intensiviert sich deren Verknüpfung und neuronale Netzwerke entstehen. Frank Rosenblatt stellte 1958 eine mathematische Nachbildung vor, das Perzeptron [Ros58]. Mehrlagige Perzeptrone bilden die Grundlage aller derzeitigen, auf künstlich neuronalen Netzen (engl. artificial intelligent network, ANN) basierenden Systeme. Dieses Kapitel erläutert Grundlagen und Lernen von ANN.[1]

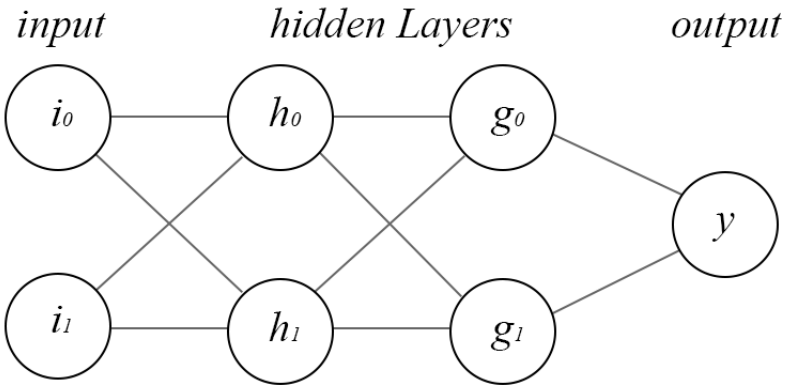

Abb. 2.1. ANN mit zwei hidden layer

Anders wie im menschlichen Gehirn werden bei einem ANN die Neuronen in Schichten eingeteilt. Das kleinste Netzwerk besteht aus der Eingabeschicht und der Ausgabeschicht. Zwischen diesen können sich eine variable Anzahl versteckter Schichten (engl. *hidden layer*) befinden. Abbildung 2.1 veranschaulicht ein solches Modell mit zwei *hidden layer*. Jedes Neuron einer Schicht ist mit jedem Neuron aus der Vor- und Nachgänger-Schicht verbunden (*fully connected layer*). Ausgehend von einer Eingabe in die Eingabeschicht werden die Werte durch die *hidden layers* zu einem Ausgabevektor in der Ausgabeschicht propagiert.

[1] Die Herleitungen dieses Kapitels basieren auf [Haf17], [Kaf17]

2.1 Vorwärtspass

Der Vorwärtspass definiert die schichtweise Werteübertragung von der Eingabe bis zu einer Ausgabe in einem ANN.

Definition 2.1 (Berechnung einer Neuronenausgabe).

1. *Alle eingehenden gewichteten Verbindungen w_{ij} eines Neurons y_j werden mit dem Wert des jeweiligen verknüpften Neurons y_i multipliziert und aufsummiert.*

$$z_j = \sum y_i * w_{ij} \tag{2.1}$$

2. *Ein Bias wird zum Zwischenergebnis summiert, um das Aktivitätslevel des Neurons zu individualisieren.*

$$z_j = z_j + bias_j \tag{2.2}$$

3. *Auf diese Summe wird eine nichtlineare Aktivierungsfunktion angewendet. Diese bildet auf einen bestimmten Wertebereich ab, um eine zu starke Gewichtung des Netzes auf einzelne Neuronen zu vermeiden.*[2]

$$y_j = \sigma(z_j) = \frac{1}{1 + e^{-z_j}} \tag{2.3}$$

Die Definition lässt sich auf Schichten mit beliebig vielen Neuronen skalieren. Formel 2.4 errechnet aus einem n-stelligen Eingabevektor einer Gewichtsmatrix und einem Bias-Vektor einen m-stelligen Ausgabevektor.

$$\begin{pmatrix} y_{10} \\ y_{11} \\ \vdots \\ y_{1m} \end{pmatrix} = \sigma \left(\begin{pmatrix} y_{00,10} & y_{01,10} & \cdots & y_{0n,10} \\ y_{00,11} & y_{01,11} & \cdots & y_{0n,11} \\ \vdots & \vdots & \ddots & \vdots \\ y_{00,1m} & y_{01,1m} & \cdots & y_{0n,1m} \end{pmatrix} * \begin{pmatrix} y_{00} \\ y_{01} \\ \vdots \\ y_{0n} \end{pmatrix} + \begin{pmatrix} bias_{10} \\ bias_{11} \\ \vdots \\ bias_{1m} \end{pmatrix} \right) \tag{2.4}$$

2.2 Rückwärtspass

Im Rückwärtspass (engl. Backpropagation) vollzieht sich das eigentliche Lernen des ANN, verdeutlicht in Abb. 2.2.

Definition 2.2 (Backpropagation).

1. *Voraussetzung ist eine Fehlerfunktion, die die Qualität bzw. den Verlust (engl. loss) des Ausgabevektors numerisch abbildet, indem dieser mit der gewünschten Ausgabe verglichen wird. Formel 2.5 berechnet den Fehler eines Neurons j in der Schicht y.*[3]

$$Fehler(y_j, y_{wanted}) = 0.5 * (y_j - y_{wanted})^2 \tag{2.5}$$

[2] Hier veranschaulicht ist eine Sigmoidfunktion. Es ist jedoch anzumerken, dass sich auch andere Funktionen zur Aktivierung eignen (vgl. `https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/`)

[3] Exemplarisch wird hier der quadratische Fehler vorgestellt. Auch hier können andere Funktionen verwendet werden (vgl. `https://www.analyticsvidhya.com/blog/2019/08/detailed-guide-7-loss-functions-machine-learning-python-code/`)

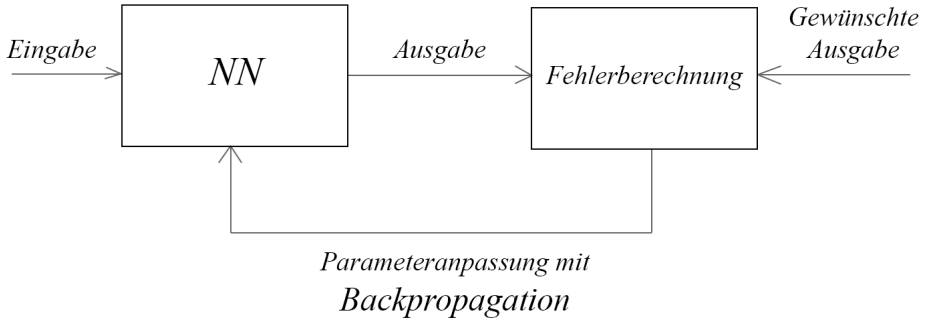

Abb. 2.2. Zusammenspiel des Vorwärts- und Rückwärtspasses eines ANN

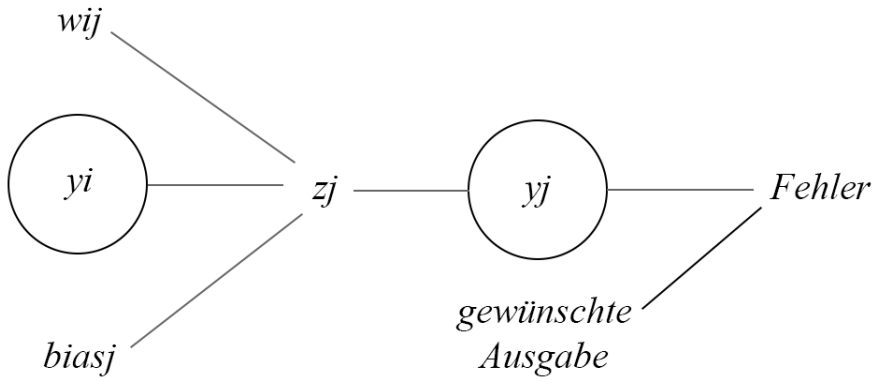

Abb. 2.3. Einfluss der Instanzen eines ANN auf den Fehler

2. *Dieser Fehlerwert kann nun in das Netz eingearbeitet werden, indem alle Instanzen, die einen Einfluss auf diesen hatten, betrachtet werden (siehe Abb. 2.3). So ist der Wert direkt abhängig von der errechneten und der erwünschten Ausgabe. Die errechnete Ausgabe ist abhängig von der Aktivierungsfunktion und dem Zwischenergebnis 2.3. Und dieses wiederum wird von den Neuronenwerten der Vorgängerschicht und deren dazugehörigen Gewichtungen 2.1 sowie dem Bias 2.2 bestimmt.*
3. *Um diese Abhängigkeiten in einen exakten numerischen Wert zu transformieren, wird die Kettenregel (engl. chain rule) verwendet. Bei dieser werden die Abhängigkeiten durch Multiplikation verkettet. Formel 2.6 errechnet den Wert, um den ein Gewicht w_{ij} angepasst werden muss, um den Fehler des Neurons y_j zu verringern. Eine Matrix, die Anpassungswerte für alle Parameter des Netzes enthält, wird auch als Gradient bezeichnet.*

$$w_{ij} = \frac{\partial Error_j}{\partial w_{ij}} = \frac{\partial Error_j}{\partial y_j} \frac{\partial y_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}} = (y_j - y_{wanted}) * \sigma'(z) * y_i \tag{2.6}$$

4. *In Gleichung 2.7 wird das Gewicht dann entsprechend adjustiert.*

$$w_{ij} = w_{ij} - \Delta w_{ij} \tag{2.7}$$

2.3 Softmax und Kreuzentropie

Softmax

Insbesondere bei Klassifizierungsproblemen wird eine spezielle Aktivierungsfunktion auf der Ausgabeschicht $Z = (z_1, z_2, ...)$ angewendet. Die $Softmaxfunktion$ 2.8 transformiert die Ausgabe der Schicht in eine Wahrscheinlichkeitsverteilung.

$$softmax(z)_\mathrm{i} = \frac{exp(z_\mathrm{i})}{\sum_{z \in Z} exp(z)} \tag{2.8}$$

Es gilt Gleichung 2.9.

$$\sum_{z \in Z} softmax(z) = 1 \tag{2.9}$$

Kreuzentropie

Die Kreuzentropie (engl. *cross entropy*) stellt ein geeignetes Werkzeug zum Bewerten eines $softmax$-aktivierten $layers$ dar. Das Ergebnis der Funktion 2.10 ist ein Abstandsmaß der beiden diskreten Wahrscheinlichkeitsverteilungen $H = (h_1, h_2, ...)$ und $Q = (q_1, q_2, ...)$.

$$H(P, Q) = -\sum_{i=0}^{i<\#P} log(q_\mathrm{i}) p_\mathrm{i} \tag{2.10}$$

Sei $\hat{Y}$ die Ausgabe des $softmax$-aktivierten $layers$ zu einem Trainingsbeispiel (X, Y) aus einem gelabeltem Datensatz N, dann ist 2.11 die Abweichung der Ausgabe des Netzes von der gewünschten Ausgabe.

$$\frac{1}{\#N} \sum_{(X,Y) \in N} H(\hat{Y}, Y) \tag{2.11}$$

Die Ableitung der Kreuzentropie Fehlerfunktion eines $softmax$-aktivierten Neurons y_j lässt sich mit Funktion 2.12 errechnen.[4]

$$\frac{\partial Error_j}{\partial z_j} = y_j - y_{wanted} \tag{2.12}$$

[4] https://peterroelants.github.io/posts/cross-entropy-softmax/ beschreibt $softmax$ und *cross entropy* genauer und leitet die Ableitungen der Funktionen her.

2.4 Gradient Descent

Durch die Definition 2.2 kann das ANN durch eine Fehlerberechnung seine Parameter entsprechend anpassen und bei der nächsten Ausgabe einen geringer zu erwartenden Verlust erzielen. Man spricht von einem Gradientenschritt. Die Summe mehrerer Gradientenschritte wird Gradientenabstiegsverfahren (engl. *Gradient Descent*) genannt. Abhängig von den Anforderungen, wie Größe des Datensatzes oder Komplexität des Problems, existieren verschiedene Arten dieses Verfahren durchzuführen. Im Folgenden werden die wichtigsten Algorithmen vorgestellt.[5]

1. **Batch Gradient Descent**
   Gemäß dem Verfahren des Algorithmus 1 wird ein Gradient für den gesamten Datensatz errechnet und erst anschließend an die Parameter des ANN angepasst.

Algorithmus 1: Batch Gradient Descent

```
while epoch < maxEpochs do
    foreach example in trainingData do
        gradient = gradient + ComputeGradient(parameter, example)
    UpdateParameter(parameter, gradient)
```

2. **Stochastic Gradient Descent**
   Bei der Methode von Algorithmus 2 werden Parameterupdates zu jedem Trainingsbeispiel durchgeführt.

Algorithmus 2: Stochastic Gradient Descent

```
while epoch < maxEpochs do
    Shuffle(trainingData)
    foreach example in trainingData do
        gradient = gradient + ComputeGradient(parameter, example)
        UpdateParameter(parameter, gradient)
```

3. **Mini-batch Gradient Descent**
   Der letzte hier veranschaulichte Algorithmus vereint die Vorteile der beiden vorangegangenen. Aus dem Trainingsdatensatz werden n (= Stapelgröße (engl. *batch size*)) zufällige Elemente entnommen, mit deren Hilfe der Gradient berechnet wird. Erst anschließend werden die Parameter angepasst, bevor, abhängig von der Epoche, wieder n neue Elemente untersucht werden.

[5] Alle Verfahren basieren auf [Rud16]

Algorithmus 3: Mini-batch Gradient Descent

```
while epoch < maxEpochs do
    batch = RandomSample(trainingData, batchSize)
    foreach example in batch do
        gradient = gradient + ComputeGradient(parameter, example)
    UpdateParameter(parameter, gradient)
```

3

Reinforcement Learning

Reinforcement Learning (RL, dt. bestärkendes Lernen) wird ein Kollektiv unterschiedlicher Methoden des maschinellen Lernens genannt. Sie einen sich darin, dass Aktionen mit dem Ziel Belohnungen zu maximieren gefördert bzw. gehemmt werden. In der Folge, dass Aktionen in einem Spiel öfter gewählt, wenn diese zum Gewinn verhelfen, umgekehrt solche gemieden, die zu einer Niederlage führen. Auf diese Weise können Annäherungen an eine optimale Strategie, auch in sehr komplexen Umgebungen, erreicht werden. Dieses Kapitel erläutert die Grundlagen von RL und beleuchtet eine Auswahl der wichtigsten Methoden auf diesem Gebiet.

3.1 Theoretische Grundlagen

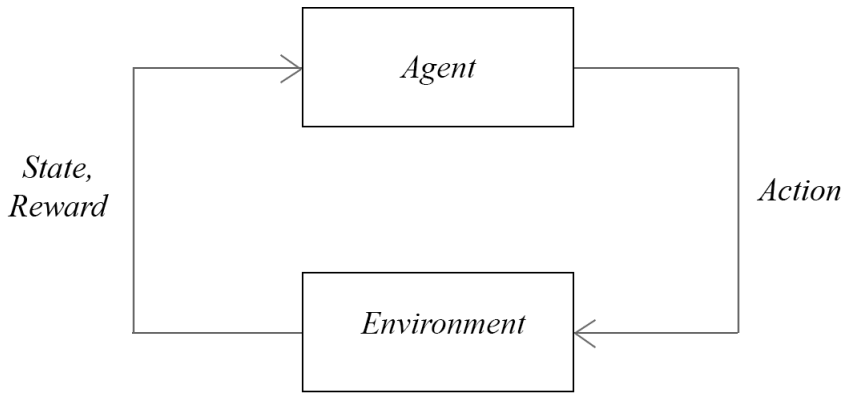

Abb. 3.1. Vereinfachte Darstellung der RL Interaktionsschleife nach [KLM96].

Wie in Abb. 3.1 veranschaulicht, interagiert ein Agent mit seiner Umgebung. Der Agent führt die von den ANN errechneten Aktionen aus. Eine Aktion zeichnet sich dadurch aus, dass sie eine Umgebung von einem Zustand in einen anderen überführt. Der Agent wählt eine Aktion, die von der Umgebung mit einer Beloh-

nung (bzw. Bestrafung) und dem neuen aus der Aktion resultierendem Zustand beantwortet wird.

3.1.1 Belohnungsfunktion

Wie eine Belohnung ausfällt, hängt von der Belohnungsfunktion R 3.1 (engl. reward function) ab. Der Betrag ist dabei abhängig von dem Zustand s_t der Umgebung zum Zeitpunkt t sowie der von dem Agenten gewählten Aktion a_t, die einen Wechsel in den nächsten Zustand s_{t+1} verursacht hat.

$$r_{\mathrm{t}} = R(s_{\mathrm{t}}, a_{\mathrm{t}}, s_{\mathrm{t+1}}) \tag{3.1}$$

Entscheidend ist die Definition der Belohnungsfunktion. Folgende Kriterien müssen dabei berücksichtigt werden:

- *Maß*
  Aktionen müssen gewichtet werden. Deshalb muss eine Wertetabelle für verschiedene Aktions-Zustands-Paare definiert werden. Beispielsweise ist der Sieg in einem Schachspiel gewichtiger als das Schlagen einer einzelnen Figur.
- *Häufigkeit*
  In manchen Spielen ist es praktisch unmöglich mit zufälligen Aktionen einen Sieg zu erringen. Beispielsweise erzielte *Deepmind* 2015 einen weiteren Meilenstein in der AI Entwicklung, als ihr Agent in 29 der 49 *Atari* 2600 Computerspiele überdurchschnittliche Fähigkeiten erlangte. In dem *Jump-And-Run*-Spiel *Montezuma's Revenge* konnte das Programm allerdings keinen einzigen Sieg erzielen[MKS+15]. Zwischenbelohnungen können folglich zwingend notwendig für Lernerfolg sein.
- *Zeitpunkt*
  Der Zeitpunkt, indem ein Spiel entschieden wird, muss nicht mit dem Zeitpunkt zusammenfallen, zu dem der Agent das Spiel gewinnt oder verliert. Das Problem, die richtigen Entscheidungen zu belohnen, wird in *Reinforcement Learning* auch als *credit assignment problem* bezeichnet.
- *Vorzeichen*
  In *Reinforcement Learning* können Belohnungen ein positives oder ein negatives Vorzeichen haben.
- *Ausgeglichenheit*
  Ein Agent kann durch ausschließlich negative oder positive Belohnungen keine nachhaltigen Lernerfolge erzielen. Es sollte also eine Balance zwischen positiven und negativem Feedback hergestellt werden.

3.1.2 Markov Decision Process (MDP)

Reinforcement Learning beschäftigt sich typischerweise mit Problemen, die auf eine Reihe richtiger Entscheidungen angewiesen sind, um Belohnungen zu maximieren. Solche Probleme werden auch als Markow-Entscheidungsprobleme bezeichnet

und können als Tupel $MDP = (S, A, R, T)$ modelliert werden. Dieses Tupel beschreibt alle Variablen, die für einen Agenten notwendig sind, um Entscheidungen innerhalb einer Umgebung zu treffen. Das Tupel setzt sich aus folgenden Bestandteilen zusammen:

- $S = (s_1, s_2, ...)$ - Alle möglichen Zustände der Umgebung
- $A = (a_1, a_2, ...)$ - Alle möglichen Aktionen
- R: $S \times A \to \mathbb{R}$ - Eine Belohnungsfunktion (vgl. 3.1)
- T: $S \times A \to \pi(s)$ - Eine Übergangsfunktion, die die Wahrscheinlichkeit zum Zeitpunkt t von dem Zustand $\mathrm{s_t}$ in $\mathrm{s_{t+1}}$ mit $\mathrm{a_t}$ zu wechseln, abbildet.

Bei einem MDP gilt die Markow-Annahme. Diese besagt, dass eine Zustandsänderung von s_t zu s_{t+1} nicht abhängig von den vorangegangenen Zuständen von s_t ist. Ein MDP modelliert also zeitunabhängige Entscheidungsprobleme.

Die Lösung eines MDP ist die optimale *policy* π. Diese gibt die Aktion aus, die den zukünftigen Gewinn maximiert. *Policies* werden in der nächsten Sektion definiert.

3.1.3 Policy

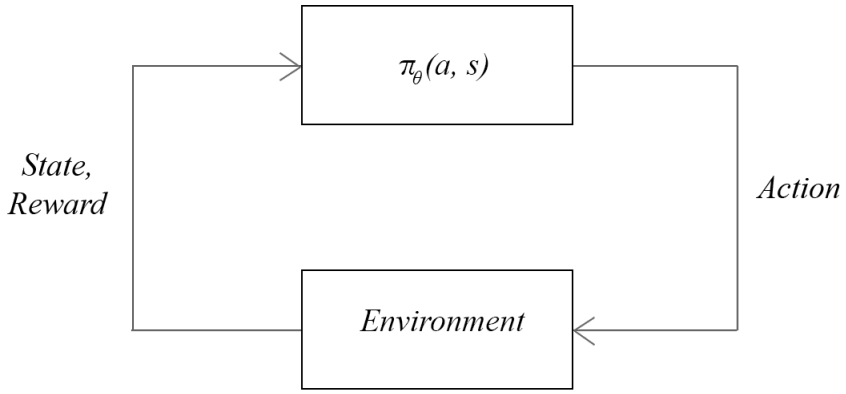

Abb. 3.2. Vereinfachte Darstellung der Interaktionsschleife einer *Policy* nach [SB+98]. θ sind die Parameter des Agenten (z.B ein ANN).

Der Agent handelt nach einer Richtlinie (engl. *policy*) π. Dementsprechend kann die Interaktion des Agenten mit seiner Umwelt auch wie in Abbildung 3.2 modelliert werden. Es gibt zwei unterschiedliche Arten von *policies*. Eine deterministische *policy* äußert sich, wie Gleichung 3.2 verdeutlicht, durch eine eindeutige Aktion als Ausgabe zu einem Zustand der Umgebung.

$$\pi(s) = a \tag{3.2}$$

Eine stochastische *policy* hingegen, wie Gleichung 3.3 zeigt, liefert eine Wahrscheinlichkeitsverteilung P der möglichen Aktionen A zu einem Zustand der Umgebung s.

$$\pi(s, a) = P(s|A) \tag{3.3}$$

Damit eine *policy* konstruiert werden kann, wird definiert, welches Verhalten optimal ist. In einer Umgebung mit festgelegter Zeitspanne wäre dies die Maximierung der Gesamtheit der erwarteten Belohnungen $\mathbb{E}$ innerhalb des Zeitraumes $T = (t_1, t_2, ..., t_H)$. Das Modell des zeitlich begrenzten Erwartungshorizontes 3.4 formuliert dieses Ziel.

$$\mathbb{E}(\sum_{t=0}^{H} r_\mathrm{t}) \tag{3.4}$$

In zeitlich variablen Problemen ist der Term zu modifizieren. Um die Konzentration des Agenten nicht zu stark von weit in der Zukunft liegenden Belohnungen abhängig zu machen, wird eine *discount*-Variable γ eingeführt. Das modifizierte Modell des zeitlich unbegrenzten Erwartungshorizontes wird in 3.5 formuliert. Das optimale Verhalten entspricht also der Maximierung der durch γ diskontierten erwarteten Belohnung. [KLM96]

$$\mathbb{E}(\sum_{t=0}^{\infty} \gamma^t r_\mathrm{t}) \tag{3.5}$$

Mit dem Modell 3.5 lässt sich eine Funktion definieren, die einen Zustand bewertet, wenn der Agent nach der *policy* π handelt. Die Funktion V 3.6 ist optimal, wenn auch die *policy* optimal ist.

$$V_\pi^*(s) = \mathbb{E}_\pi(\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}) \tag{3.6}$$

Die äquivalente Gleichung 3.7 zeigt, dass dies dem Erwartungswert für die beste Aktion a eines Zustand s entspricht.

$$V_\pi^*(s) = max_a(R(s, a) + \gamma \sum_{\hat{s} \in S} T(s, a, \hat{s}) V^*(\hat{s})) \tag{3.7}$$

Ausgehend der Funktion 3.7 kann die optimale *policy* definiert werden. Sie bildet nicht den numerischen Wert der bestmöglichen Aktion ab, sondern liefert genau diese Aktion.

$$\pi^*(s) = argmax_a(R(s,a) + \gamma \sum_{\hat{s} \in S} T(s,a,\hat{s}) V^*(\hat{s})) \tag{3.8}$$

Ein Problem von *policies* ist die Bewertung dieser. Der Agent weiß nicht, wie gut seine *policy* ist oder ob er bereits die optimale erreicht hat. Folgende Sektion beschreibt dieses Problem und Lösungen dazu genauer.

3.1.4 Exploitation versus Exploration

Wie in Abbildung 2.2 veranschaulicht, bedarf es zum Lernen eines ANN immer eines Lernvektors, der als Datenträger dem System zeigt, wie es seine Parameter anzupassen hat, um den Verlust zu verringern. Da bei RL-Problemen typischerweise keine Trainingsdaten vorliegen, müssen diese erst von dem Agenten selbst erfahren werden. Dieses Problem wird *Exploitation versus Exploration* (dt. Erkundung versus Erschließung) genannt.

Ein bekanntes Beispiel in Bezug auf RL ist das *k-armed bandit problem*. Ein Agent trifft dabei die Entscheidung, einen von k Glücksspielmaschinen zu betätigen. In der Ausgangssituation verfügt der Agent bereits über eine auf erster aber nicht vollständiger Erkundung basierende Einschätzung, aber nicht über den tatsächlichen Erwartungswert der Glücksmaschinen. Das Problem lässt sich zusammenfassen in der Fragestellung, ob der Agent seine Entscheidung basierend auf dieser Einschätzung trifft oder neue Maschinen erkundet. In einem pragmatischen Szenario hängt dies von der Länge des Spieles ab. Wenn der Agent nur noch wenige Spielzüge vor sich hat, ist die Wahl der Glücksmaschine mit dem höchsten Erfahrungswert empfehlenswert. Bei vielen weiteren Spielzügen sollte jede Maschine ausreichend erkundet werden, bevor sich auf eine festgelegt wird. [KLM96]

Fortschreitende Erkundung ist ein wichtiger Bestandteil von RL. Es gibt verschiedene Methoden dies zu realisieren. Im Fall von deterministischen *policies* erkundet der Agent, indem Aktionen durch Wahrscheinlichkeiten repräsentiert sind. Dadurch können weniger wahrscheinliche Spielverläufe ebenso durchlaufen werden.

Ist das Ziel eines Systems jedoch die optimale *policy* aus Funktion 3.8, so wird immer die Aktion mit dem höchstem erwarteten Gewinn gewählt, Exploration bleibt aus und es kann fälschlicherweise in eine suboptimale *policy* investiert werden. Wählt der Agent nach diesem Prinzip seine Aktion, wird seine Strategie auch als *greedy* bezeichnet.

Die naheliegendste Möglichkeit das Erkundungsproblem zu lösen, ist mit einer gewissen Wahrscheinlichkeit p eine zufällige Aktion zu wählen. Diese Lösung konnte sich beispielsweise in dem in Kapitel 3.2.1 beschriebenen Algorithmus *Deep* $Q-learning$ durchsetzen. Oft wird dies ergänzt durch eine bestimmte Verfallsrate (engl. *decay*), durch die p im Laufe der Zeit oder mit ansteigendem Lernerfolg vermindert wird. Algorithmus 4 zeigt eine Variante, sich einem Gleichgewicht zwischen Erkundung und Erschließung anzunähern.[1]

[1] Mittlerweile existieren eine Vielzahl verschiedener Methoden *Exploration* zu optimieren. `https://towardsdatascience.com/exploration-in-reinforcement-learning-e59ec7eeaa75` geht mehr ins Detail.

Algorithmus 4: Exploration versus Exploitation Example

```
while learning do
    state = environment.Render()
    p = Max(p, pMin)
    if p < Random(0, 1) then
        Explore(environment)
    else
        Exploit(state, environment)
    p = p * decay
```

3.2 Methoden

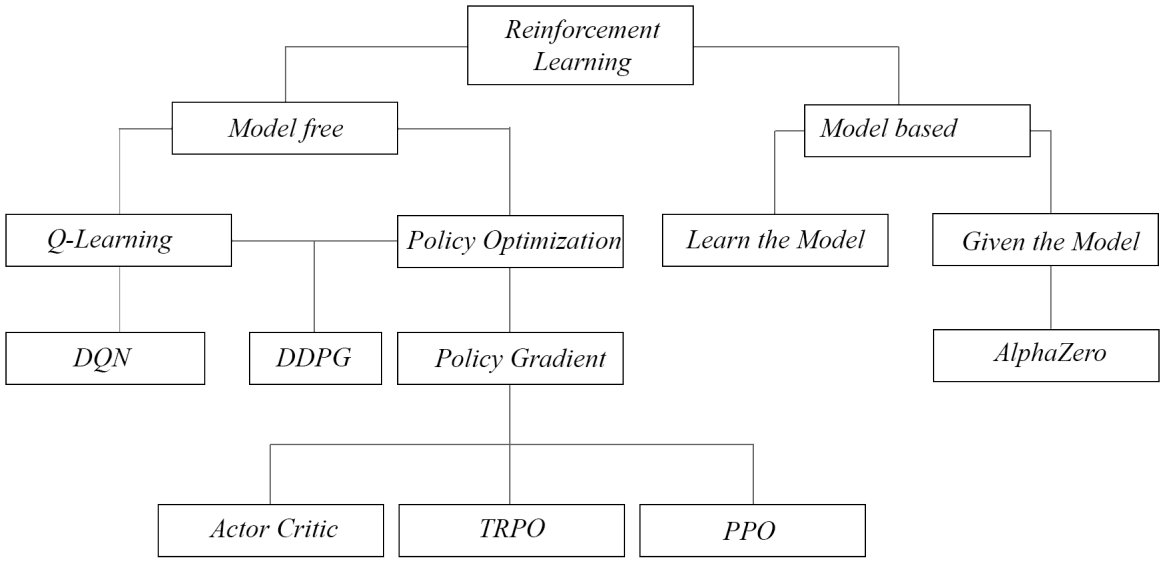

Abb. 3.3. Auswahl RL Algorithmen in Anlehnung an `https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html`

RL Algorithmen sind nach Abbildung 3.3 in zwei Kategorien einzuteilen. In *Model based* ist eine korrekte Repräsentation der Umgebung bekannt. Mit Hilfe von iterativen Suchverfahren, wie der *Monte-Carlo-tree-search* (MCTS), können hier Spielverläufe simuliert und so optimale Lösungen gefunden werden. Da aber bei komplexen Strategiespielen, wie Schach, Shogi und Go, die Anzahl unterschiedlicher Spielverläufe bis ins nahezu Grenzenlose geht, können die Spiele praktisch nicht vollständig durchgerechnet werden. *AlphaZero* nutzt die Suche für die Vorhersage einiger weniger Spielzüge und passt seine Parameter an die Ergebnisse an. Falls die MCTS beispielsweise Spielverläufe findet, die in naher Zukunft zu einem Sieg führen, so werden die zugehörigen Aktionen bestärkt. [SHS+18]

Bei vielen, vor allem realen Problemen, ist die Anzahl der Transitionsfunktionen $\#T$ des MDP jedoch unbekannt bzw. praktisch unbegrenzt. Eine Repräsentation der Umgebung liegt nicht vor. In diesen $Model\ free$ Szenarios werden *Policy*- und/oder $Value$-Funktionen erlernt. Die optimale Definition dieser Funktionen wurde bereits im Kapitel 3.1.3 formuliert. Folgende Kapitel erläutern jeweils eine entscheidende Lernmethode dieser beiden Funktionen.

3.2.1 Deep Q-Learning

$Deep\ Q-learning$ ist ein von *DeepMind* 2015 vorgestellter Algorithmus, der sogenannte Q-Values mit $Deep\ Q-Networks$ (DQN) lernt. Q-Values sind, wie in 3.9 zu sehen ist, eine Funktion, die eine Aktion a und einen Zustand s auf einen numerischen Wert abbilden.

$$Q\colon S \times A \to \mathbb{R} \tag{3.9}$$

Die Q-Funktion bewertet also nicht nur den Zustand s einer Umgebung, sondern die Aktion a, wenn sie im Zustand s gewählt wird und anschließend nach der *policy* π gehandelt wird. Sie wird in Referenz auf die Value-Funktion V aus 3.6 wie in 3.10 formuliert.

$$Q(s,a) = R(s,a) + \gamma \sum_{\hat{s} \in S} T(s,a,\hat{s}) V(\hat{s}) \tag{3.10}$$

Eine bekannte Methode Q-Values zu lernen ist $Q-learning$. Bedingung hierfür ist, dass die Zustandsmenge S und der Aktionsraum A endlich sind. Ist das der Fall, so werden in der Startphase alle Aktions-Zustand-Paare $Q(s,a)$ mit 0 initialisiert. Mit der Updateregel 3.11 für die Q-Funktion können dann neue Q-Werte gelernt werden. Mit α wird die Lerngeschwindigkeit adjustiert.

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha \cdot (r_t + \gamma \cdot max_a(Q(s_{t+1},a)) - Q(s_t,a_t)) \tag{3.11}$$

Algorithmus 5 zeigt, wie ein Agent eine Q-Tabelle ausnutzt und anschließend aktualisiert. 1992 wurde bereits bewiesen, dass $Q-learning$ konvergiert. [WD92]

Algorithmus 5: Q-learning

```
if p < Random(0, 1) then
    Explore(environment)
else
    Execute(Argmax(Q[s]), environment)
newState, reward = environment.Render()
Q[s][a] += alpha * (reward + gamma * Max(Q[newState]) — Q[s][a])
```

Wenn die Zustandsmenge eines MDP-Problems jedoch (praktisch) unbegrenzt ist, ist eine Q-Tabelle auf Grund des nur endlich verfügbaren Speichers keine praktikable Lösung. Dieses Problem löst der Algorithmus *deep* $Q-learning$, indem es die Informationen, die eine Q-Tabelle enthält, in den Parametern eines Neuronalen Netzes speichert. Jedes Ausgabeneuron entspricht dem Q-Value einer möglichen Aktion im derzeitigen Zustand.

Das Verfahren wird in Algorithmus 6 präsentiert. Der Agent agiert nach einer *greedy policy* mit einer Zufallswahrscheinlichkeit p. Um einen stabilen Lernvorgang zu gewährleisten, benutzen sie eine Technik, genannt *experience replay*. Hier werden in einer Datenstruktur eine bestimmte Menge an Erinnerungen $e_t = (s_t, a_t, r_t, s_{t+1})$ abgelegt. Der Erinnerungsspeicher wird dafür genutzt alte Erinnerungen aufzufrischen bzw. nicht zu vergessen, sodass die Q-Values nicht zu stark an die letzten Erfahrungen angepasst sind. In der Praxis werden in jedem Zeitschritt eine gewisse Anzahl an Beispielen zufällig aus den Speicher gewählt und an die Parameter des Netzwerkes trainiert (vgl. 3).

Weiter optimierte *DeepMind* seine Methode, indem es zwei identische Architekturen eines ANN verwendet. Ein Hauptnetzwerk wählt Aktionen und passt stetig seine Parameter an, während das andere *target*-Netz nur auf dem Erinnerungsspeicher arbeitet, wo es den höchsten Q-Value des nächsten Zustandes errechnet. Dieser Wert wird anschließend in den Lernvektor für das Hauptnetzwerk eingespeist. Nach bestimmt vielen Zeitschritten C übernimmt dieses dann alle Parameter des Hauptnetzwerkes. Der Grund für das zweite Netzwerk ist, dass eine Parameteranpassung verursacht durch $Q(s_t, a_t)$ höchstwahrscheinlich auch $Q(s_{t+1}, a)$ ändert, möglicherweise resultierend in einem anderen Q-Value einer anderen Aktion. Durch eine Verzögerung zwischen der Änderung von $Q(s_t, a_t)$ und $Q(s_{t+1}, a)$ werden größere Schwankungen und Divergenz des Lernerfolgs unwahrscheinlicher. [MKS+13][GLSL16]

3.2.2 Policy Gradient

Policy-basierte Systeme lernen keine Q-Values, sondern eine stochastische *policy* 3.3.

Bei *Policy-Gradient* Verfahren werden in der Regel ganze Episoden bzw. Aktions-Zustands-Sequenzen $\tau = (s_0, a_0, ..., s_H, a_H)$ gelernt. Die Belohnungsfunktion R 3.12 kalkuliert die zukünftig erwartete Belohnung einer solchen Sequenz.

$$R(\tau) = \sum_{t=0}^{H} R(s_t, a_t) \tag{3.12}$$

Diese Sequenz hilft eine Funktion J 3.13 zu definieren, die eine *policy* π bewertet. Die Optimierung dieser Funktion wird durch Optimierung der Parameter θ erreicht. J lässt sich auch als Erwartungswert $\mathbb{E}$ der zukünftigen Belohnungen, wenn nach der *policy* π agiert wird, formulieren. Die *policy* π_θ ist hier stochastisch, weshalb sie sich nach Definition 3.3 als Wahrscheinlichkeitsverteilung P ausdrücken lässt.

Algorithmus 6: deep Q-learning

```
/* Wähle zufällige Aktion a mit Wahrscheinlichkeit p, sonst
   Aktion mit höchstem Q-Value                                  */
if p < Random(0, 1) then
    action = Explore(environment)
else
    action = Argmax(mainNet.Predict(state))

/* Führe Aktion aus und beobachte neuen Zustand der Umgebung
   sowie Belohnung der Aktion                                   */
Execute(action, environment)
newState, reward, done = environment.Render()

/* Füge Beispiel dem Erinnerungsspeicher hinzu                  */
experience.Append((state, action, reward, newState, done))

/* Führe Minibatch Update durch                                 */
batch = RandomSample(experience, batchSize)
foreach sample in batch do
    output = mainNet.Predict(sample.state)
    target = Copy(output)
    if sample.done then
        /* Label Lernvektor mit der Belohnung, falls die Episode
           hier geendet ist                                     */
        target[sample.action] = r
    else
        /* Ansonsten addiere zusätzlich den höchsten
           diskontierten Q-Value des nächsten Zustandes         */
        qFuture = Max(targetNet.Predict(sample.newState))
        target[sample.action] = sample.reward + qFuture * gamma

    /* Verlust mit einer Fehlerfunktion bestimmen und
       Gradienten für die Parameter des Netzwerkes errechnen    */
    loss = net.loss.Backward(target)
    gradient = gradient + ComputeGradient(mainNet.parameter, loss)
UpdateParameter(mainNet.parameter, gradient)

/* Synchronisiere die Netzwerke alle C Schritte                 */
targetNet = mainNet
```

$$J(\theta) = \mathbb{E}(\sum_{t=0}^{H} R(s_t, a_t); \pi_\theta) = \sum_{\tau} P(\tau|\theta)R(\tau) \tag{3.13}$$

In moderner Forschung steckt hinter einer *policy* typischerweise ein ANN. θ kann in diesem Kontext mit den Parametern des Netzwerkes gleichgesetzt werden, da die *policy* auf abstrakte Weise in den *weights* und *bias* repräsentiert ist. Folgend wird J maximiert, indem die Parameter des Netzwerkes angepasst werden. Der Gradient für das Gradientenabstiegsverfahren aus 2.4 ergibt sich in folgenden vier Schritten:

1. Gradient und Summe tauschen

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_{\tau} P(\tau|\theta)R(\tau) = \sum_{\tau} \nabla_\theta P(\tau|\theta)R(\tau) \tag{3.14}$$

2. Mit $P(\tau|\theta)$ multiplizieren und dividieren

$$\nabla_\theta J(\theta) = \sum_{\tau} P(\tau|\theta)\frac{\nabla_\theta P(\tau|\theta)}{P(\tau|\theta)}R(\tau) \tag{3.15}$$

3. Log-Ableitungstrick nutzen

$$\nabla_\theta J(\theta) = \sum_{\tau} P(\tau|\theta)\nabla_\theta log(P(\tau|\theta))R(\tau) \tag{3.16}$$

4. In Erwartungswert konvertieren

$$\nabla_\theta J(\theta) = \mathbb{E}(\nabla_\theta log(P(\tau|\theta))R(\tau)) \tag{3.17}$$

Über einen Datensatz N von Aktions-Zustands-Sequenzen lässt sich der Durcschnitt des Gradienten nach Gleichung 3.18 formulieren. [BB01][2]

$$\nabla J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}(\sum_{t=0}^{T} \nabla_\theta log(\pi_\theta(s_{i,t}, a_{i,t})))(\sum_{t=0}^{T} R(s_{i,t}, a_{i,t}) \tag{3.18}$$

$\nabla_\theta log(\pi_\theta(s_{i,t}, a_{i,t}))$ ist dabei die Richtung, in die die *policy* $\pi_\theta(s_{i,t}, a_{i,t})$ verbessert werden muss, damit die logarithmische Wahrscheinlichkeit für Aktion a steigt. Wird diese Richtung multipliziert mit der Belohnung $R(s_{i,t}, a_{i,t})$, ergibt sich ein

[2] Eine detailreiche Erklärung vo dem OpenAI Team lässt sich hier finden: `https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html`

Maß für die tatsächlich gewünschte Änderung. Es resultiert in einer Wahrscheinlichkeitsverteilung, die abhängig von der Stärke und dem Vorzeichen der Belohnung die Aktion $a_{i,t}$ häufiger oder seltener wählt.

Wie Algorithmus 7 zeigt, ist diese Richtung eine Wahrscheinlichkeitsverteilung mit gefälschtem Label. Es wird also ein Nullvektor mit einer 1 am Index der Aktion $a_{i,t}$ initialisiert. Dies entspricht einer Wahrscheinlichkeitsverteilung mit 100% für $a_{i,t}$ und 0% für alle anderen Aktionen.

In einem nächsten Schritt muss nun der Abstand zwischen dieser Wahrscheinlichkeitsverteilung und der *policy* $\pi_\theta(s_{i,t}, a_{i,t})$ errechnet werden. Ein geeignetes Abstandsmaß ist die Kreuzentropie (vgl. Kapitel 2.3). Die Kreuzentropie als Fehlerfunktion hat den Vorteil, dass sie bereits den logarithmischen Fehler minimiert. Das Ergebnis der abgeleiteten Kreuzentropie der beiden Wahrscheinlichkeitsverteilungen ist die Richtung $\nabla_\theta log(\pi_\theta(s_{i,t}, a_{i,t}))$, in die $\pi_\theta(s_{i,t}, a_{i,t})$ verbessert werden muss, damit die Wahrscheinlichkeit für $a_{i,t}$ im Zustand $s_{i,t}$ 100% ist. Die Richtung besteht also aus einem Vektor, der die Wahrscheinlichkeit für $a_{i,t}$ erhöht und für alle anderen Aktionen verringert. Wenn dieser mit der Belohnung multipliziert wird, dann wird $\nabla_\theta log(\pi_\theta(s_{i,t}, a_{i,t}))$ abhängig von Vorzeichen und Wert der Belohnung skaliert.

Algorithmus 7: Simple Policy Gradient

```
/* Aktion abhängig der des ANN generierten
   Wahscheinlichkeitsdistribution wählen, ausführen, Änderung
   beobachten und an Episodenspeicher anhängen                    */
action = RunPolicy(net.Predict(environment))
Execute(action, environment)
state, reward, done = environment.Render()
episode.Add((state, action, reward))

if done then
    /* diskontierte Belohnungen rückwirkend einfügen              */
    reward = 0
    foreach sample in reverse episode do
        sample.reward += reward * discount
        reward = sample.reward

    /* Batch Update                                               */
    foreach sample in episode do
        /* Lernvektor mit 'gefälschtem' Label generieren          */
        target = new Vector(environment.actionSpace)
        target[sample.action] = 1
        /* Fehler der Ausgabe und des Lernvektors errechnen und
           mit der Belohnung multiplizieren                       */
        output = net.Predict(sample.state)
        loss = crossEntropy.Backward(output, target)
        loss *= sample.reward

        /* Gradienten berechnen                                   */
        gradient = gradient + ComputeGradient(net.parameter, loss)

    /* Netzwerk mit manipuliertem Fehler aktualisieren            */
    UpdateParameter(net.parameter, gradient)
```

4

Implementierung einer RL-Umgebung zur Visualisierung

4.1 Motivation

Künstliche neuronale Netzwerke als Schnittstelle zwischen Informatik und den Neurowissenschaften sind ein komplexes Forschungsgebiet. Anders wie bei rein informationstechnischen Problemen können Sachverhalte hier nicht vollständig mit Logik erklärt werden. Eine Konvergenz mancher Algorithmen kann trotz großem Forschungsaufwand nicht bewiesen werden [MKS+13].

Um den Einstieg in die Thematik zu vereinfachen, ist es notwendig, das abstrakte Konzept der ANN greifbarer zu machen. Moderne Programmierbibliotheken versuchen sich bereits daran. *Keras* verfolgt das Ziel, komplizierte *machine learning* Schnittstellen so nutzerfreundlich wie möglich zu machen. Auch wenn die Anwendung dadurch deutlich erleichtert wird, bleibt die Funktionsweise dem Anwender unbekannt. Maschinelles Lernen wird angewandt, aber nicht verstanden.

Das folgend beschriebene Konzept bietet einen leichten Einstieg in die Grundzüge von Maschinellem Lernen mit ANNs und RL.

4.2 Konzept

Menschen sind besonders empfänglich für visuelle Reize. Im Kopf helfen sie, aus theoretischen Modellierungen reale Strukturen zu kreieren.

Eine geeignete Visualisierungsmethode für das abstrakte System der ANN ist eine Applikation. Komponenten wie *weights*, *bias* und Neuronen können mit unterschiedlichen Formen und Farben ihre Werte abbilden. Ergänzt werden soll dies durch ein Baukastensystem, durch das Änderungen an den Netzwerken in Echtzeit beobachtet und nachvollzogen werden können. In diesem Feature soll auch ein Anpassen sämtlicher Hyperparameter möglich sein. Dadurch wird ein leichter Zugriff in den Lernprozess gewährt und die Auswirkungen können direkt beobachtet werden.

Mit maschinellem Lernen wird oft *supervised learning* assoziiert. Nach einer Trainingsphase wird das Netz getestet und bewertet. Eine deutlich prägnantere Methode, Lernfortschritte zu beobachten, ist, einem Agenten in Echtzeit beim Trainieren zu folgen. RL ist ein geeignetes Werkzeug hierfür. Hinzu kommt, dass

keine Trainingsdaten abgespeichert oder über eine Schnittstelle zur Verfügung gestellt werden müssen. Die Daten werden stattdessen während der Laufzeit generiert. Innerhalb von RL soll zwischen unterschiedlichen Lernpraktiken ausgewählt werden können.

Eine häufige Testumgebung im Forschungsbereich künstlicher Intelligenz sind Computerspiele. Aus diesem Grund wird das Programm eine Reihe verschiedener Spiele mit unterschiedlichem Schwierigkeitsgrad enthalten. Bei diesen werden diskriminierende Probleme definiert, die den Anwender zwingen, die Problemlösungen zu variieren.

Die Applikation soll als weiteres Feature paralleles Trainieren enthalten, mit dem Vorteil unterschiedliche Netzwerkarchitekturen, Hyperparameter oder Lernmethoden in Echtzeit zu vergleichen. Dadurch liefert das System stabilere Ergebnisse, indem es die Folgen zufallsgenerierter Parameter des ANN abschwächt.

Die Basisimplementation sollte eine Visualisierung für ANN, verschiedene RL Praktiken, Computerspiele als Lernumgebung und paralleles Trainieren umfassen. Alle weiteren Komponenten stellen Ergänzungen dieser dar. Besonders im *machine learning* können *layer*-Typen, *optimizer* und Regularisationsmethoden beinahe grenzenlos ergänzt werden. Folgendes Kapitel beschreibt eine Möglichkeit der Realisierung.

4.3 Realisierung

Die Umsetzung folgt dem Ziel einfachster Bedienbarkeit. Die größte Herausforderung stellt hier ein gelungener Kompromiss zwischen Nutzerfreundlichkeit und Systemfähigkeiten dar. Mit vereinfachter Bedienungsoberfläche werden den Handlungsoptionen Grenzen gesetzt. In der Anwendung ist die erste Begrenzung das Angebot der Probleme. Auf der Startseite ist somit eine feste Anzahl von Spielen zu sehen, die verschiedene Herausforderungen für RL bieten. Kenngrößen, die diese Auswahl beeinflusst haben, sind die Anzahl der Frames, nach denen Belohnungen ausgegeben werden, die maximale Spiellänge und die Variablen des MDP 3.1.2. Abbildung 4.1 zeigt die Probleme, die nach Diversität vorangegangener Kriterien in die Implementation integriert worden sind[1].

4.3.1 Probleme

Pong

Die hier abgewandelte Version von *Pong* ist ein Einzelspielerspiel mit der Aufgabe, den Ball mit Hilfe eines Klotzes nicht auf den Boden kommen zu lassen. Das Problem zeichnet sich besonders durch kurze Eingabe- und Ausgabelängen aus. Als Eingabe erhält der Agent die jeweiligen x- und y-Koordinaten der Position und des Richtungsvektors des Balles sowie die x-Koordinate des kontrollierten Klotzes.

[1] Es ist anzumerken, dass als *input* keine Pixel gewählt wurden, da die Anwendung kein *image processing* thematisiert

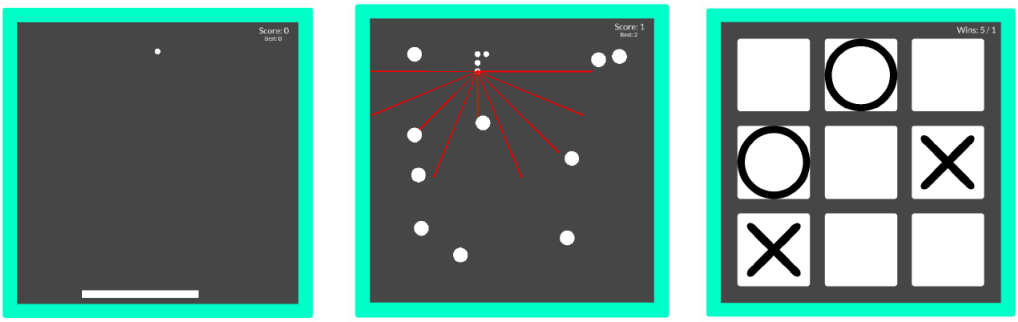

Abb. 4.1. Implementierte Spiele in der aktuellen Version von links nach rechts: Pong, Snake, TicTacToe

Die Ausgabe beschränkt sich auf Rechts-Links-Bewegung, also eine Änderung der x-Koordinate des Klotzes, oder das Aussetzen einer Aktion. Das Spiel kann nicht gewonnen und damit unendlich weitergeführt werden. Die größte Herausforderung in diesem Spiel ist die große zeitliche Differenz zwischen zwei Belohnungen. Der Agent wählt also viele Aktionen, die unbewertet bleiben.

Snake

Bei *Snake* versucht der Spieler hier möglichst viele Früchte zu sammeln, ohne mit sich selbst oder der Wand zu kollidieren. Die hier implementierte Version arbeitet mit einem relativ großen Spielfeld. Die Eingabe beschränkt sich bei diesem Problem auf eine bestimmte Anzahl von Blickwinkeln. Realisiert werden diese durch Strahlen S, die vom Kopf der Schlange aus in eine bestimmte Richtung d ausgestoßen werden (In Abbildung 4.1 sind die Strahlen in rot zu sehen). Wie Funktion 4.1 zeigt, liefern diese als Rückgabe die Differenz der maximalen Ausstrahlungsdistanz $maxD$ und der Distanz von dem Agenten mit Position p zum Kollisionsobjekt obj. Falls es zu keiner Kollision kommt, wird 0 zurückgegeben.

$$S(p, d, maxD) = \begin{cases} maxD - D(p, obj.p), & \text{falls Kollision} \quad (4.1) \\ 0, & \text{sonst} \quad (4.2) \end{cases}$$

In jede Richtung werden drei Strahlen ausgesendet, die jeweils versuchen entweder mit den Begrenzungen des Spielfeldes, mit einer Schlangenkomponente oder einer Frucht zu kollidieren. Der Agent kann jedoch nicht hinter ein Objekt sehen. Falls eine Frucht vor einer Wand liegt, wird nur die Distanz zur Frucht gegeben und die zur Wand als 0 evaluiert. In diesem Verhalten erklärt sich auch die Differenz von der maximalen Ausstrahlungsdistanz aus dem Term 4.1.

Als Standard wird die Anzahl der Richtungen auf 9 festgelegt, die in 22,5°Abständen ausgesandt werden. Der *Input*-Vektor hat damit eine Länge von 27 und der totale Blickwinkel der Schlange ist 180°.

Die Aktionen bei *Snake* beschränken sich auf Rechts-, Links-, Oben- und Untenbewegung. Vereinfacht wurde diese, indem die Schlange sich ausgehend von ihrer aktuellen Richtung entweder nach rechts, links oder vorne bewegen kann. Dies hat den Vorteil, dass erstens die Schlange gar nicht ihre eigene Richtung wissen muss, um entscheiden zu können und sich zweitens die Länge des Ausgabevektors auf drei reduziert.

Grundannahme ist, dass der Agent keine optimale Strategie erlernen kann. Der Agent verfügt nur über eine begrenzte Sichtweite und ihm stehen deshalb nicht alle Informationen zum aktuellen Zustand der Umgebung zur Verfügung.

Die Besonderheit dieses Problems ist die Steigerung des Schwierigkeitsgrades im Spielverlauf, da die Schlange mit Anzahl der kollidierten Früchte wächst.

TicTacToe

Als letztes Problem wurde das Strategiespiel *TicTacToe* gewählt. Alle Variablen des MDP 3.1.2 sind vollständig bekannt und es existieren Verfahren zur iterativen Suche der optimalen Strategie (vgl. Minimax-Algorithmus). Es existieren 255.168 verschiedene Ausgangssituationen. Davon kann 131.184 der erste Spieler für sich entscheiden, und 77.904 der folgende.

TicTacToe bietet für RL interessante Herausforderungen. Erstens wird der Handlungsspielraum abhängig vom Zustand der Umgebung stark eingeschränkt. Der Spieler kann nur freie Felder wählen, während die Netzwerkarchitektur jedoch nur eine festgelegte Ausgabelänge zulässt. Zweitens ist die Spiellänge stark limitiert. Der angestrebte Erwartungswert wird durch die zeitliche Beschränkung von maximal 5 Zügen auf der Seite des Agenten nach 3.4 formuliert. Und drittens existiert ein Gegenspieler. Dieser wird standardmäßig durch einen zufallsbasierten Akteur repräsentiert, kann aber auch durch andere oder neue Agenten ersetzt werden.

Das Spielfeld von *TicTacToe* besteht aus neun Feldern. Unter Berücksichtigung der drei verschiedenen Zustände, die ein Feld einnehmen kann, setzt sich die Eingabeschicht des *TicTacToe*-Agenten aus 27 Neuronen zusammen. Die Ausgabelänge setzt sich gleich mit der Anzahl der Felder. Eine Aktion wählt also ein Feld aus, auf das das entsprechende Zeichen gesetzt werden soll.

4.3.2 Visualisierung

Die Kernintention der Applikation ist, ANN und RL nachvollziehen zu können. Die Visualisierung spielt dabei die entscheidende Rolle. Realisiert ist diese durch ein ein- und ausblendbares Fenster (vgl. Abbildung 4.2), welches in Echtzeit die Werte des ANN des Agenten projiziert. Wie im Konzept bereits beschrieben, kann dies durch verschiedene Farben und Formen umgesetzt werden. Neuronen werden als Kreise dargestellt, die abhängig von ihrem letzten Wert mit einem Grauwert

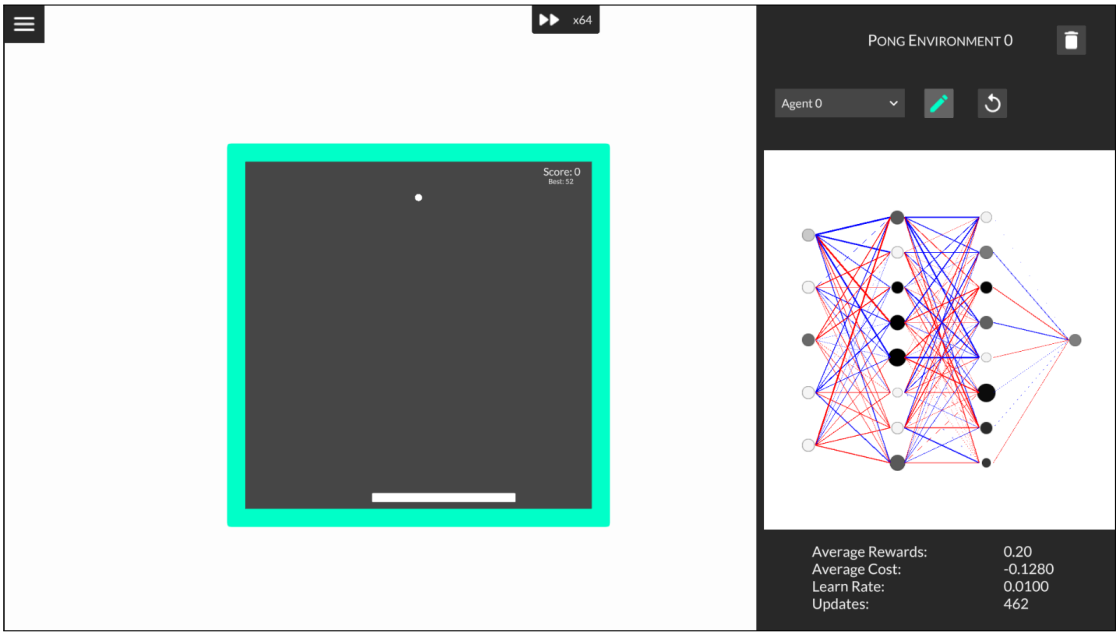

Abb. 4.2. Ein- und ausblendbares Informationsfenster zur ausgewählten Umgebung und den dort agierenden Akteuren. Enthält von oben nach unten folgende Elemente: Namen und Löschfunktion der Umgebung, Dropdown mit allen Akteuren in dieser Umgebung sowie eine Editier- und Rücksetzfunktion für Agenten, Visualisierung des Netzwerkes, Parameter- und Erfolgsinformationen

gefüllt sind. Ihre Größe ist zudem abgeleitet von dem aktuellen *bias*. Zwischen Neuronen sind Gewichtungen in Form von Linien gespannt, die ihre Größe auf ihren Absolutwert anpassen und rot bzw. blau für positive bzw. negative Werte erscheinen.

Wird ein neues Spiel erstellt, so wird das ANN immer mit einem *input-layer* mit Länge des Eingabevektors und einem *softmax-layer* abhängig von der Länge des Ausgabevektors erstellt. In dem ANN visualisierendem Fenster kann in ein Baukastenmenü (vgl. Abbildungen 4.3 und 4.4) gewechselt werden. Hier können *hidden layer* gelöscht oder neue hinzugefügt werden. Beim Hinzufügen kann die *layer*-Art, die Anzahl der Neuronen und die Aktivierungsfunktion ausgewählt werden.

4.3.3 Parallelität

Im Hauptmenü können unter *Add*, wie in Abbildung 4.5 zu sehen ist, beliebig viele Umgebungen hinzugefügt werden. Die Startbedingungen für diese wurden bereits in Kapitel 4.3.1 beschrieben. Das parallele Trainieren bietet viele Vorteile, sowohl für den Trainingserfolg als auch für die Veranschaulichung des Trainingsvorgangs. So können verschiedene Netzwerkarchitekturen, Hyperparameter und Lernmethoden gegeneinander abgeglichen werden. Hinzugefügt wurde eine erleichternde Funktion, die es ermöglicht eine beliebige Anzahl Umgebungen mit Agenten derselben Struktur zu erzeugen. Somit kann entgegen der zufallsbestimmten Pa-

Abb. 4.3. Baukastenmenü eines Agenten. Rechts werden Strukturänderungen an dem ANN des Agenten vorgenommen. Links können verschiedene Informationen eingesehen und Parameter adjustiert werden

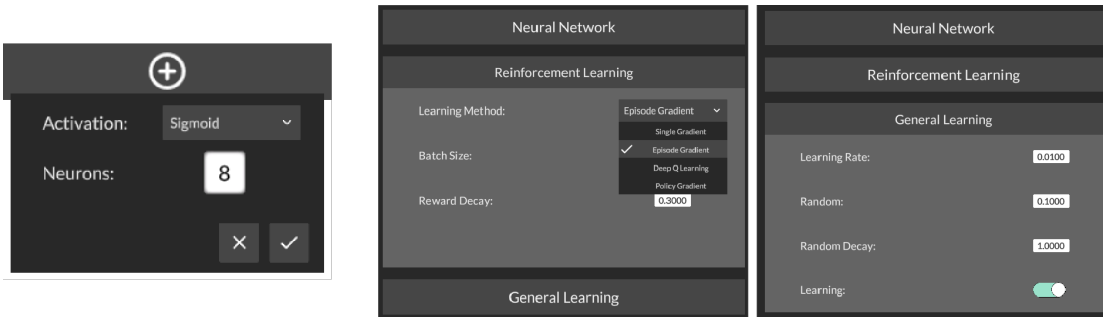

Abb. 4.4. Von links nach rechts: Optionenpanel nach Auswahl einer hinzuzufügenden Netzschicht, Parameterpanel für RL, Parameterpanel für allgemeine Lernparameter

rameter durch eine Vielzahl der Netzwerke ein mittlerer Lernerfolg für bestimmte Hyperparameter mit einer bestimmten Architektur determiniert werden.

Ein weiterer Vorteil parallel laufender Umgebungen ist die Möglichkeit Agenten zu selektieren. Durch Klonen können nach evolutionärem Prinzip erfolgreiche Agenten vervielfältigt bzw. nicht erfolgreiche durch gezieltes Aussortieren entfernt werden.

4.3.4 Validierung

Der Lernerfolg der Agenten muss für den Anwender nachvollziehbar sein. Die effizienteste Möglichkeit sind Graphen, die ein Maß für den Lernerfolg auf der x-Achse auf eine Zeiteinheit auf der y-Achse abbilden. Der Durchschnitt aller Belohnungen ist ein präzises und greifbares Maß für Lernerfolg, da es exakt das Ziel des Agen-

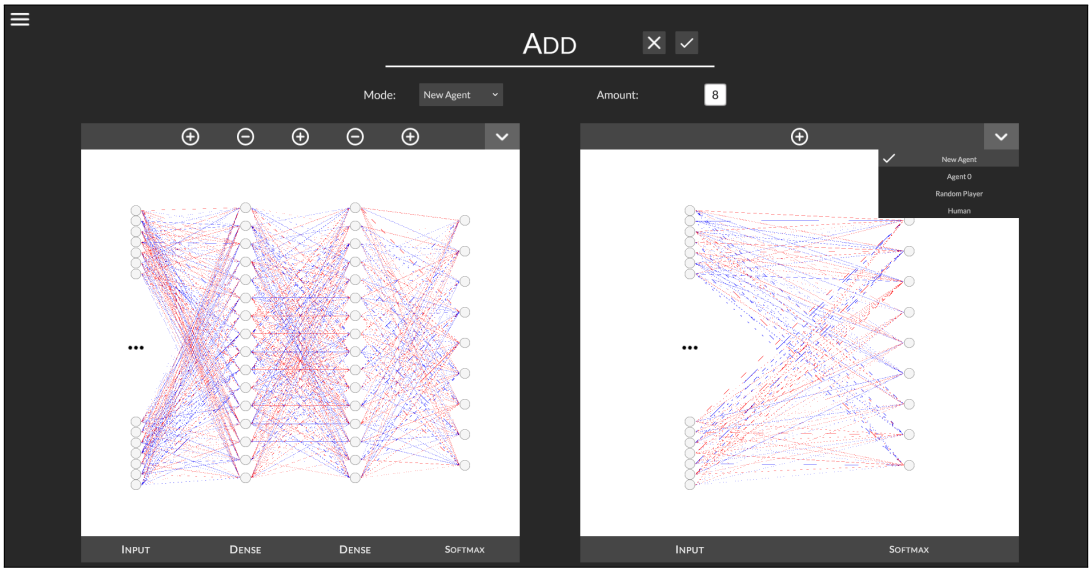

Abb. 4.5. Add-Option für TicTacToe. Oben kann der Modus verändert werden. Unterschieden wird zwischen dem Klonen eines Agenten, dem Kopieren der Netzwerkarchitektur und der Neuinitialisierung. Der rechte Block stellt den ersten Akteur und der linke den zweiten. In der rechten oberen Ecke der Blöcke kann ein Agent zum klonen oder kopieren sowie ein menschlicher Spieler oder ein rein zufälliger Spieler ausgewählt werden.

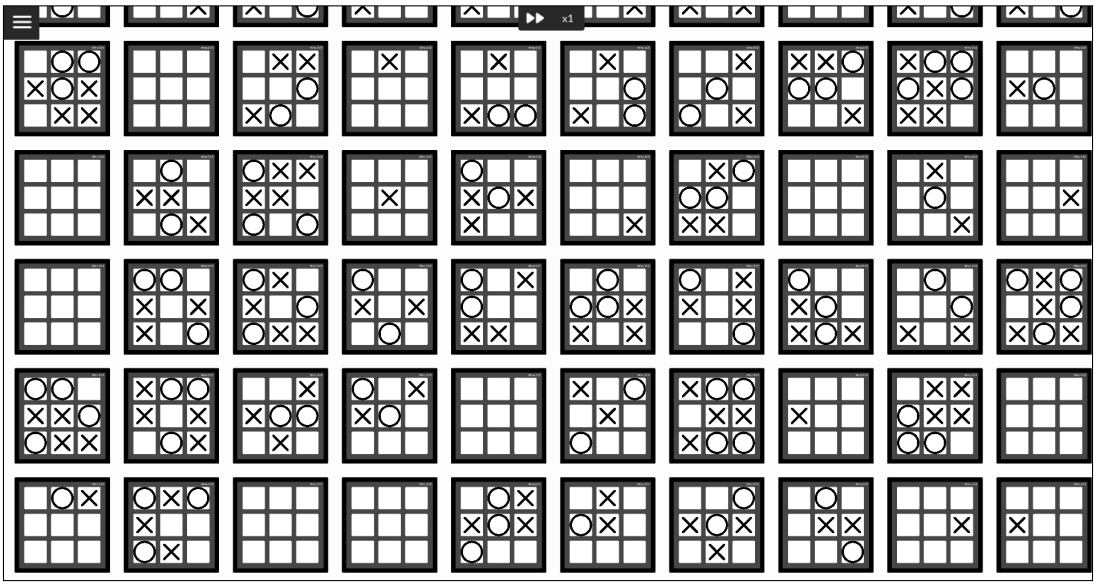

Abb. 4.6. Demonstration mehrerer Agenten und Umgebungen.

ten, Belohnungen zu maximieren, widerspiegelt. Damit zudem die Fehlerfunktion überwacht werden kann, kann der Anwender die Anzeige des Graphen wechseln. In der zweite Anzeige wird der durchschnittliche Fehler des Agenten abgebildet.

Zusätzlich werden spielspezifische Informationen abgebildet. So wird die Siegesrate bei *TicTacToe* bzw. die am meisten gegessenen Früchte bei *Snake* oder die gefangenen Bälle in *Pong* in der rechten oberen Ecke der Spielfelder visualisiert. Auch diese Kennzahlen stellen einen validen Maßstab für Lernerfolg dar. In kürzlich durchgeführten Forschungsprojekten wurden Werte verwendet, wie beispielsweise *elo* in Schach oder Ranglistenwertigkeiten der online Ligen in *Starcraft* 2 und *Dota* 2, um den Lernerfolg ihrer Systeme zu veranschaulichen[VBC+19][BBC+19].

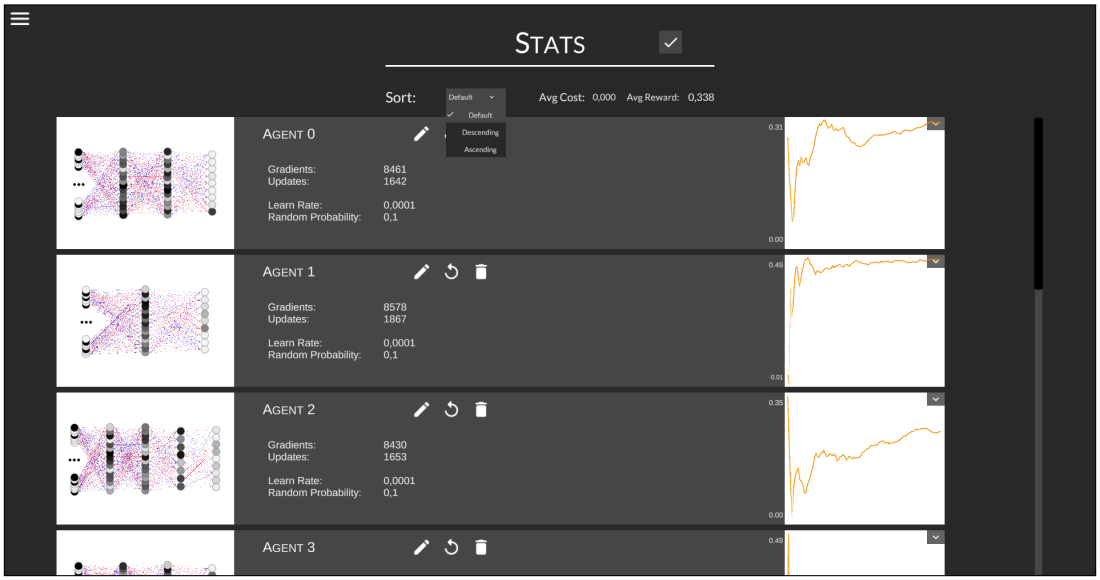

Abb. 4.7. Stats-Option, die es erlaubt verschiedene Agenten direkt zu vergleichen

Da die Applikation auch paralleles Trainieren ermöglicht, muss ein geeignetes Vergleichswerkzeug zwischen unterschiedlichen Agenten bereitgestellt werden. Die Applikation setzt dies mit Hilfe einer Statistikoption um. Hier werden alle trainierenden Agenten mit Netzwerkarchitektur, einer Auswahl der wichtigsten Lern- und Erfolgsparameter und den genannten Graphen abgebildet. Sortiert werden können diese nach auf- bzw absteigendem Lernerfolg. Auf diese Weise können starke bzw. schwache Agenten schnell entdeckt, angepasst oder entfernt werden.

Die parallele Ausführung in Kombination mit einer anschaulichen Visualisierung der Netzwerke und übersichtlicher Methoden zur Validierung stellen die Grundbausteine der Implementation dar. Zusammen bieten sie ausreichend Spielraum mit RL und ANN zu experimentieren. Im folgenden und letztem Kapitel werden Anwendungsmöglichkeiten und Testergebnisse beleuchtet.

4.3.5 Technische Details

Die Anwendung wurde mit Hilfe der *Unity Engine* umgesetzt. Diese Umgebung ist auf die Entwicklung von Computerspielen spezialisiert. Das hat den Vorteil, dass sie die Implementierung flüssig laufender Grafik Applikationen erleichtert. Zusätzlich bietet sie eine übersichtliche Organisationsstruktur. *Unity* verwendet als Skriptsprache *C#*, weshalb ein *machine learning* Framework in dieser Programmiersprache implementiert wurde. Zudem wurden die Standardbibliotheken von *.NET* und *Unity* verwendet.

5

Anwendungen und Experimente

Reinforcement Learning ist ein sehr breit gefächertes Methodenkollektiv und moderne Verfahren basieren auf komplexen mathematischen Theorien. In dieser Sektion wird der Weg zu bekannten Methoden, wie *DQN* 3.2.1 oder *Policy Gradient* 3.2.2, erläutert. Angefangen mit dem naivesten Lösungsweg werden einzelne Anpassungen des RL-Algorithmus untersucht. Hierzu wurde die Applikation durch ein Speichersystem ergänzt, welches den Lernerfolg eines Agenten in eine *.csv* Datei schreibt. Mit Hilfe von *Excel* wurden so Statistiken erfasst, entscheidende RL-Praktiken isoliert betrachtet und verglichen. Als Messwert wurde der Durchschnitt der Belohnungen pro Zeitschritt gewählt. Der höchste bzw. niedrigste zu erreichende Messwert entspricht also der höchsten bzw. niedrigsten zu erreichenden Belohnung. Um eine statistische Wertigkeit trotz begrenzter Rechenleistung zu erhalten, wurde die Anzahl der getesteten Agenten pro Versuch auf 100 festgelegt.

Das Ziel von *Reinforcement Learning* Methoden ist es, eine Lösung zu finden, wie die Parameter des Agenten anzupassen sind, um Belohnungen zu maximieren. Auf ANN bezogen wird dies erreicht, indem aus einer Belohnung ein Lernvektor generiert und in die Parameter des Netzwerkes eingearbeitet wird.

Falls der Aktionsraum nur aus zwei Möglichkeiten, z.B. Rechts- und Linksbewegung, besteht, so kann die Ausgabeschicht und damit der Lernvektor mit einem einstelligen Vektor realisiert werden. Die erste Aktion wird repräsentiert durch eine Neuronenausgabe von 0, die zweite mit einer von 1. Falls 0 die Rechtsbewegung repräsentiert und belohnt werden soll, so wird der Lernvektor zu [0]. Bei einer Bestrafung folglich zu [1]. Eine Bestrafung der einen Aktion ist somit auch immer eine Belohnung der anderen.

Naive Single Gradient

In einem ersten naiven Ansatz passt der Agent seine Parameter nur durch aktuelle Belohnungen an. Wenn sich der Agent also im Falle von *Pong* nach rechts bewegt und in dem Moment den Ball nicht fängt, dann wird genau diese Rechtsbewegung bestraft. Im Umkehrschluss wird sie belohnt, wenn der Agent den Ball fängt.

In diesem Ansatz wird nicht immer die tatsächlich richtige Entscheidung belohnt. Falls beispielsweise ein Agent mit der beschriebenen Rechtsbewegung sich in Ballrichtung bewegt hat, wird fälschlicherweise die richtige Entscheidung bestraft.

Es ist jedoch weniger wahrscheinlich, dass dies eintritt, da insgesamt weniger Szenarien existieren, in denen der Agent sich für eine Rechtsbewegung entscheidet und der Ball trotzdem rechts von ihm an den Boden gelangt, als solche in denen der Agent sich nach rechts bewegt, der Ball aber links von ihm auf den Boden fällt.

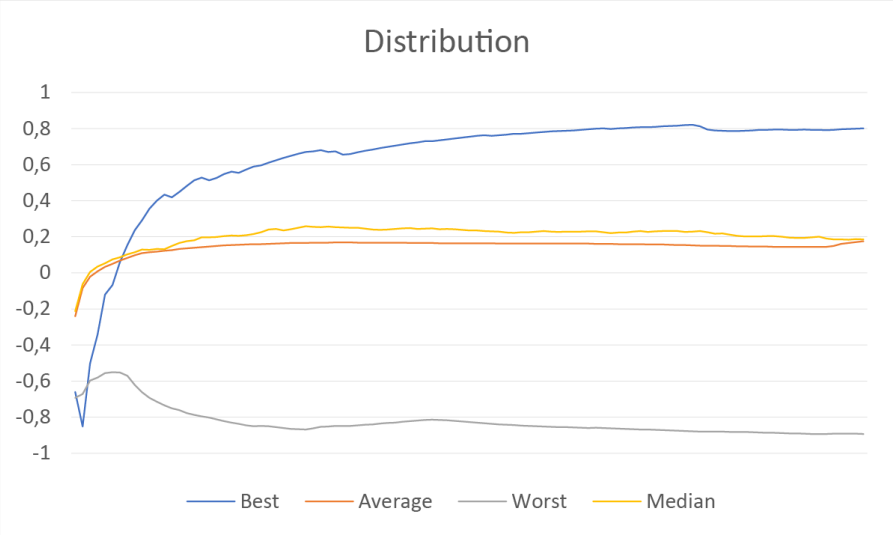

Abb. 5.1. Verteilung des Lernerfolges von 100 Agenten bei naivem bestärkendem Lernen in dem Spiel *Pong*

Wie Abbildung 5.1 zeigt, reicht diese naive Realisierung von *Reinforcement Learning* aus, dem Agenten das Spiel *Pong* zu lehren. Obwohl für den Agenten nur eine von vielen Aktionen einer Episode, und zwar genau die zum Zeitpunkt der Belohnung, bewertet wird, kann ein durchschnittlicher Lernerfolg verzeichnet werden.

Ein Erklärungsansatz besteht darin, dass die Anpassung zum aktualisierendem Zeitpunkt auch eine Auswirkung auf die anderen Zeitschritte hat, da die betroffenen Neuronen schon vor diesem Zeitpunkt feuern. Der Lernvektor kann also auch korrekt für vergangene Zeitschritte der Episode sein. Bei der Eingabe des Spielfeldes als Pixelmatrix ist hingegen kein Lernerfolg zu erwarten, da zu jedem unterschiedlichen Zustand andere Inputneuronen feuern.

Das Ergebnis des Algorithmus ist jedoch sehr instabil, wie Abbildung 5.1 weiterhin zeigt. Während die erfolgreichsten Agenten nahezu alle Bälle fangen, verfehlen andere den Ball zu fast 100%. Bei *Snake* und *TicTacToe* konnte bei deutlich weniger Agenten Erfolg bewiesen werden. Zudem ist bei einem Durchschnitt von 100 Agenten kein Lernprozess zu erkennen.

Episode Gradient

Eine naheliegende Verbesserungsmöglichkeit ist es, vergangene Aktionen ebenfalls mit der letzten Belohnung zu bewerten. Wie in dem *policy gradient* Algorithmus 7

wird eine Belohnung rückwirkend über eine Episode diskontiert vergeben. Da diese Zeitspanne und damit die Anzahl der Gradienten groß werden kann (in *Pong* theoretisch unendlich), ist es notwendig die Belohnungen zu standardisieren. Wie im Codeabschnitt 8 verdeutlicht, wird der Durchschnitt aller Belohnungen einer Episode von jeder einzelnen Belohnung abgezogen, resultierend in einer Gesamtbelohnung von 0. Anschließend wird die Belohnung weiter durch die Varianz aller Belohnungen geteilt, sodass $Varianz = 1$ gilt. Damit wird sichergestellt, dass die Gradienten keine zu großen Änderungen an den Parametern des Netzes verursachen und die Belohnungen gleichzeitig ausgeglichen verteilt sind. Falls der Agent also nicht erfolgreich ist, werden Belohnungen stärker gewichtet bzw. ist er erfolgreich, werden sie abgeschwächt.[1]

Algorithmus 8: Standardize Rewards

```
discountedRewards -= Mean(discountedRewards)
discountedRewards /= StandardDeviation(discountedRewards)
```

Ein Lernvektor kann anschließend nach Algorithmus 9 generiert werden. Durch Addition der Belohnung auf die gewählte Aktion, wird der Wert der Aktion in genau diesem Zustand leicht erhöht. Alle anderen Aktionen bleiben hingegen unbewertet. Der Agent agiert nach einer *greedy policy*, weshalb die Neuronen auch als Q-Values interpretiert werden können.

Algorithmus 9: Episode Gradient

```
foreach sample in episode do
    target = net.Predict(sample.input)
    target[sample.action] += sample.reward
```

In Abbildung 5.2 ist zu sehen, dass der Durchschnitt der erzielten Belohnungen der Agenten in *Pong* deutlich angestiegen ist zur naiven Methode. Hinzu kommt, dass auch deutliche Lernerfolge bei den beiden anderen Problemen zu erkennen sind. Die Kurven weisen zudem einen gleichmäßigen Verlauf auf, was für Stabilität des Algorithmus spricht. Der Ausbruch bei *Pong* am Anfang könnte sich durch die anfänglich hohe Zufallswahrscheinlichkeit erklären lassen. Zufällige Aktionen in Relation zu einem untrainierten Agenten zeigen höheren Erfolg, da untrainierte Agenten oft zu nur genau einer Aktion neigen.

Ein großes Problem dieser Methode ist, dass die Belohnungsfunktion gut ausbalanciert sein muss. Bei zu vielen Belohnungen mit gleichem Vorzeichen konvergieren die Neuronen trotz Standardisierung zu stark gegen 0 oder 1.

[1] Die Methode wird unter anderem in dem *Pong*-Projekt von Andrej Karpathy verwendet (vgl. http://karpathy.github.io/2016/05/31/rl/).

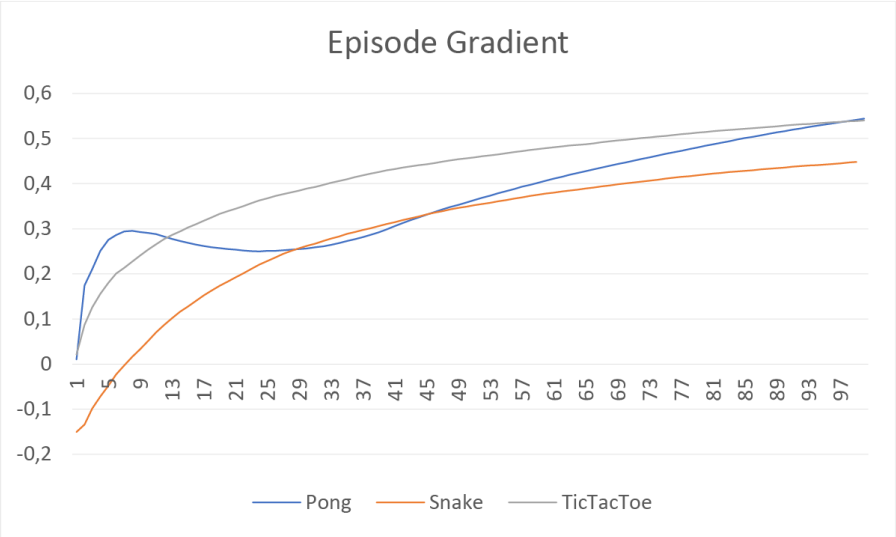

Abb. 5.2. Lernerfolg von jeweils 100 Agenten pro Spiel trainiert mit Gradienten einer ganzen Episode.

Policy Gradient

Wie bereits in Sektion 3.2.2 beschrieben, agiert ein Agent hier nach einer Wahrscheinlichkeitsverteilung. Theoretisch würde eine Belohnung einer Aktion automatisch die anderen bestrafen. Jedoch führt dies zu sehr ungleichmäßigen Gradienten, da nur die Parameter angepasst werden, die unmittelbaren Einfluss auf genau dieses Ausgabeneuron hatten.

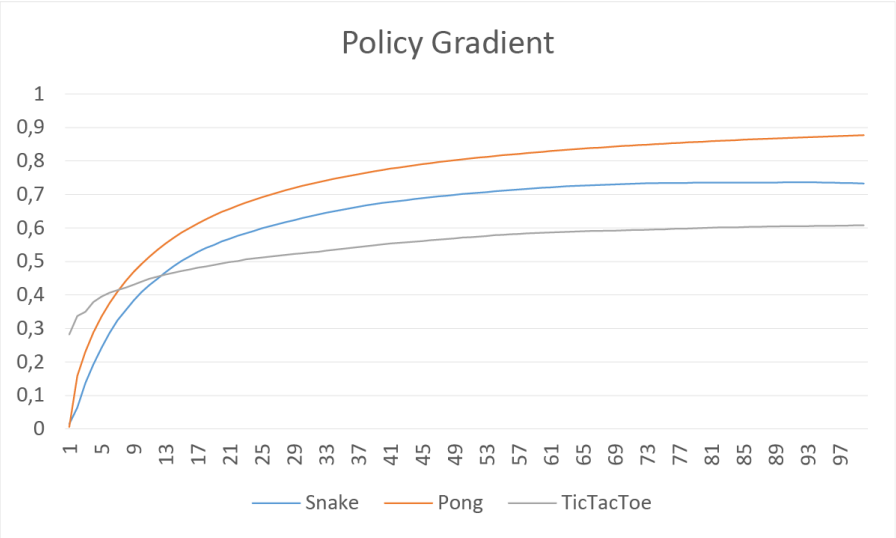

Abb. 5.3. Lernerfolg von jeweils 100 Agenten pro Spiel trainiert mit einem einfachen *policy gradient* Algorithmus.

Der *policy gradient* Algorithmus 7 umgeht dies, indem zuerst die Label Lernvektoren gefälscht werden, dann der Fehler berechnet und erst anschließend die Belohnung multipliziert wird. Damit wird gleichmäßig eine Aktion belohnt, wohingegen alle anderen bestraft werden. Abbildung 5.3 legt dar, dass mit dieser Variante stabile und große Lernfortschritte erzielt werden können.

Policy gradient liefert vielversprechende Ergebnisse für die drei ausgewählten Probleme. Jedoch gibt es auch Probleme, bei denen es nicht die beste Möglichkeit ist, nur die vergangene Episode zu betrachten.

Experience replay

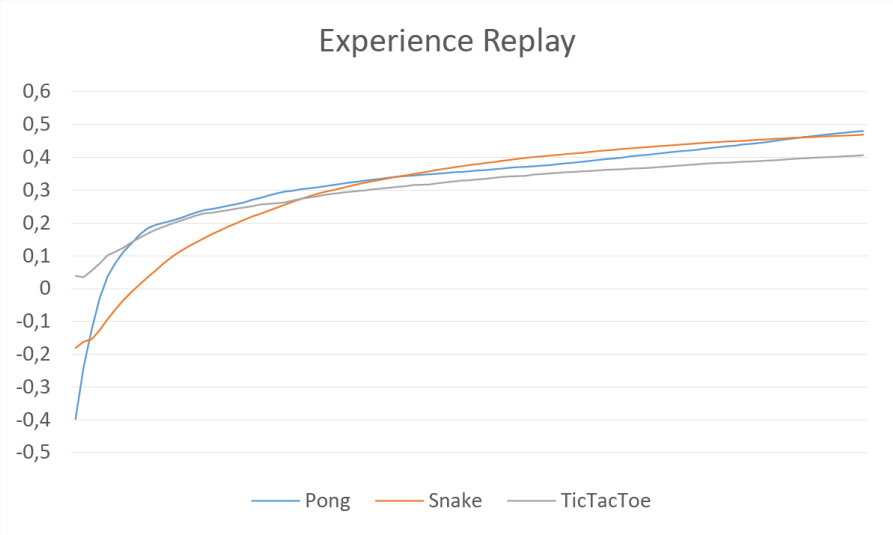

Abb. 5.4. Lernerfolg von jeweils 100 Agenten pro Spiel trainiert mit *experience replay*. Die Agenten agierten nach einer *greedy policy*, rewards wurden diskontiert und standardisiert und Lernvektoren wurden nach Algorithmus 9 generiert.

Experience replay ist eine Variante, die versucht die Allgemeingültigkeit eines Netzwerkes zu stärken. Werden nur die letzten Erfahrungen bearbeitet, so kann das Netzwerk nützliche Lösungswege vergangener Episoden vergessen. Stattdessen werden hier Trainingsbeispiele in einem Erinnerungsspeicher abgelegt und immer wieder in *batches* neu verarbeitet. Abbildung 5.4 zeigt, dass diese Methode ähnliche Ergebnisse erzielt wie bereits in *epsiode gradient* 5.2.

Deep Q-learning

Deep Q-learning kombiniert die Methodik von *experience replay* mit $Q-$*learning*. Wie in Algorithmus 6 veranschaulicht, werden vom Netzwerk generierte Q-Values durch ein *batch*-basiertes Verfahren aktualisiert. *Deep Q-learning* benötigt in Bezug auf die einfachen implementierten Probleme deutlich mehr Zeit bis ein Lernerfolg zu sehen ist. Grund hierfür könnte sein, dass das zweite Netzwerk,

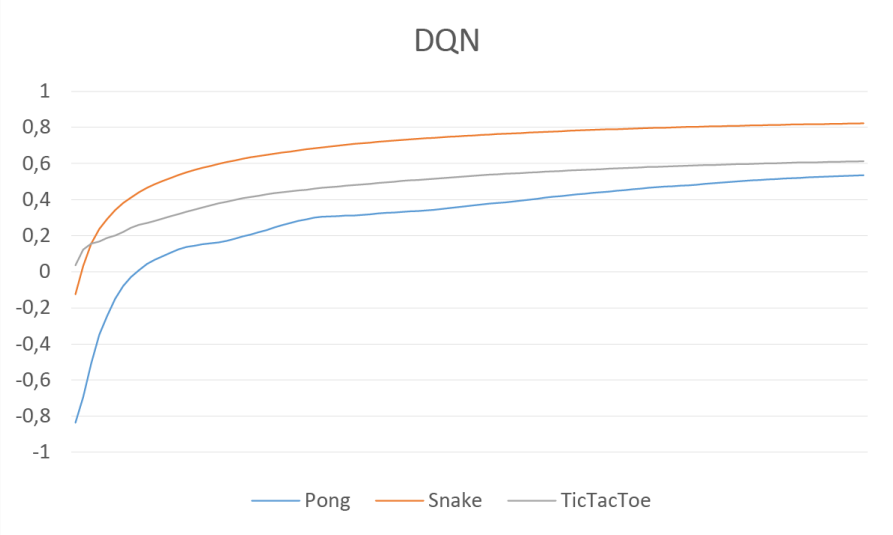

Abb. 5.5. Lernerfolg von jeweils 100 Agenten pro Spiel trainiert mit *DeepQ-learning* nach Algorithmus 6.

welches zukünftige Q-$Values$ vorhersagt, erst nach einer bestimmten Trainingszeit lernfähige Werte ausgibt. Abbildung 5.5 zeigt, dass das Verfahren einen stabilen Lernprozess ermöglicht.

Wie in diesem Kapitel gezeigt wurde, können intuitive Lösungsverfahren gute Ergebnisse liefern. Durch jahrelange Forschung wurden Methoden entwickelt, die aufbauend auf der Idee eines episodenübergreifenden Gradientens oder einer Form des *experience replay* Stabilität und Performanz des Lernprozess steigern konnten. DQN und *policy gradient* sind die Folge dieses Forschungsprozess.

Es ist jedoch anzumerken, dass dieses Kapitel nur einen kleinen Teil der einfließenden RL-Faktoren behandelt. Andere Explorationsverfahren, Belohnungsfunktionen und Standardisierungsverfahren von Belohnungen würden die Lernerfolge möglicherweise stark beeinträchtigen. Außerdem ist zu erwarten, dass weitere Anpassungen der Netzwerkarchitektur, der Lernrate oder des Optimierers des Gradientenabstiegsverfahrens zu besseren Ergebnissen führen würden. Weitere Faktoren, die den Lernerfolg maßgeblich beeinflussten, waren zum einen eine Skalierung der Eingabewerte auf einen Wertebereich zwischen 0 und 1. Zum anderen die Interpretation der Input- und Outputneuronen. Also welche Inputneuronen zwingend notwendige Informationen für den Agenten darstellen und wie die Aktionen repräsentiert werden.

6

Fazit

In dieser Arbeit wurde eine Applikation vorgestellt, die ein Experimentieren mit RL und ANN auch ohne große Vorkenntnisse, theoretische Expertise oder Programmierfähigkeit ermöglicht. Die Visualisierung von Neuronalen Netzwerken hilft die abstrakte Struktur greifbar zu machen und der Transfer der Hyperparameter schafft einen schnellen Überblick über die möglichen Anpassungen der Netzwerke und der Trainingsmethoden. Außerdem veranschaulichen Computerspiele Lernerfolge mit RL nachvollziehbar in Echtzeit.

Das letzte Kapitel wirft ein Licht auf die Komplexität der Thematik und legt nahe, dass verschiedene Probleme auch verschiedene Lösungsansätze präferieren.

Die Applikation hilft der Demonstration neuronaler Netzwerke oder von *Reinforcement Learning* Methoden und lässt diese komplexen Thematiken besser nachvollziehen und verstehen.

7

Ausblick

Die Implementierung der RL-Umgebung wird nur einem kleinen Teil des sehr komplexen Problems gerecht. Eine Erweiterung in viele Richtungen ist denkbar.

Ein interessantes Feature der Applikation ist die evolutionäre Selektierung. In einem nächsten Schritt könnte man diesen Prozess automatisieren und evolutionäre Lernalgorithmen einbinden. Auch *DeepMind* hat mit dem Projekt *AlphaStar* gezeigt, dass die Kombination von evolutionären Verfahren mit RL Methoden ein vielversprechender Trainingsprozess ist [VBC+19].

Ein weiterer Schritt würde zeitabhängige Probleme umfassen, bei denen eine Aktion abhängig von den vorher gewählten Aktionen ist oder die vorherigen Zustände der Umgebung von Bedeutung sind. Die Voraussetzungen, dass die Applikation in einer zukünftigen Version zeitabhängige Probleme unterstützt, sind bereits in der aktuellen Version vorhanden, jedoch nicht notwendig für die implementierten Probleme.

Weitere Erweiterungen in Bezug auf neuronale Netze sind nahezu grenzenlos, unter anderem für Optimizer, *layer*-Typen, Fehlerfunktionen und Regularisationsverfahren.

Mit steigender Komplexität der Anwendung könnte ein Wechsel zu einem der performanten *machine learning* Frameworks empfehlenswert sein. Auch wenn diese die Programmiersprache der populären Grafikengines nicht unterstützen, ist ein Transfer auf eine andere Plattform denkbar. Als Webapplikation beispielsweise könnte die Anwendung leicht zugänglich gemacht werden. Zudem existiert eine Implementierung des Frameworks *TensorFlow* in *JavaScript*. Dies wäre ein weiterer Schritt, die Applikation populärer und marktfähig zu machen.

Literaturverzeichnis

All94. Allis, L. Victor: *Searching for solutions in games and artificial intelligence.* 1994.

BB01. Baxter, Jonathan und Peter L Bartlett: *Infinite-horizon policy-gradient estimation.* Journal of Artificial Intelligence Research, 15:319–350, 2001.

BBC+19. Berner, Christopher, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub W. Pachocki, Michael Petrov, Henrique Pond'e de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski und Susan Zhang: *Dota 2 with Large Scale Deep Reinforcement Learning.* ArXiv, abs/1912.06680, 2019.

GLSL16. Gu, Shixiang, Timothy P. Lillicrap, Ilya Sutskever und Sergey Levine: *Continuous Deep Q-Learning with Model-based Acceleration.* CoRR, abs/1603.00748, 2016.

Haf17. Haffner, Ernst Georg: *Informatik für Dummies. Das Lehrbuch.* WILEY-VCH Verlag, 2017.

Kaf17. Kaffka, Thomas: *Neuronale Netze.* mitp Verlag, 2017.

KLM96. Kaelbling, L P, M L Littman und A W Moore: *Reinforcement Learning: A Survey.* Technischer Bericht cs.AI/9605103, May 1996.

MKS+13. Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra und Martin A. Riedmiller: *Playing Atari with Deep Reinforcement Learning.* CoRR, abs/1312.5602, 2013.

MKS+15. Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg und Demis Hassabis: *Human-level control through deep reinforcement learning.* Nature, 518:529–33, 02 2015.

Ros58. Rosenblatt, Frank F.: *The perceptron: a probabilistic model for information storage and organization in the brain.* Psychological review, 65 6:386–408, 1958.

Rud16. Ruder, Sebastian: *An overview of gradient descent optimization algorithms.* CoRR, abs/1609.04747, 2016.

SB+98. Sutton, Richard S, Andrew G Barto et al.: *Introduction to reinforcement learning*, Band 135. MIT press Cambridge, 1998.

SHS+18. Silver, David, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan und Demis Hassabis: *A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play.* Science, 362(6419):1140–1144, 2018.

VBC+19. Vinyals, Oriol, Igor Babuschkin, Wojciech Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John Agapiou, Max Jaderberg und David Silver: *Grandmaster level in StarCraft II using multi-agent reinforcement learning.* Nature, 575, 11 2019.

WD92. Watkins, Christopher J. C. H. und Peter Dayan: *Q-learning.* In: *Machine Learning*, Seiten 279–292, 1992.

A

Glossar

ANN	ANN (Artificial neuronal Network(dt. Künstliches neuronales Netzwerk)), Softwarearchitektur, die einem biologischen Gehirn nachempfunden ist
AI	AI (Artificial Intelligence (dt. Künstliche Intelligenz)), Teilgebiet der Informatik, das sich mit der Nachahmung intelligentem Verhalten beschäftigt
RL	RL (Reinforcement Learning (dt. Bestärkendes Lernen)), Methoden des maschinellen Lernens
MDP	MDP (Markov Decision Process (dt. markovscher Entscheindungsprozess), mathematische Modellierung eines Entscheidungsprozesses