Autonom lernende Agenten in Computerspielen

Anwendung von Reinforcement-Lernverfahren im Star Ships Learning Framework

Bachelorarbeit

von

Christian Lutz

Freiburg, 13. September 2010

Lehrstuhl für maschinelles Lernen und natürlichsprachliche Systeme
Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Betreuer:

Dr. Thomas Gabel

Prof. Dr. Martin Riedmiller

Inhaltsverzeichnis

1 Einleitung

Künstliche Intelligenz (KI) ist aus heutigen Computerspielen kaum mehr wegzudenken. Sie ist meist in Form von Computergegnern anzutreffen, gegen die der menschliche Spieler antreten kann. Dabei finden viele Bereiche der KI Anwendung, wie zum Beispiel Planung und Pfadfindung. Auf regelmäßig tagenden, internationalen Konferenzen wie dem *IEEE Symposium on Computational Intelligence and Games* (`www.ieee-cig.org`) wird durch verschiedene Veröffentlichungen und Wettbewerbe die Entwicklung von Künstlicher Intelligenz in Computerspielen vorangetrieben. Immer häufiger kommen dabei auch maschinelle Lernverfahren zum Einsatz.

Reinforcement Learning[1] (RL) ist ein Teilgebiet des maschinellen Lernens, bei dem ein Agent mithilfe von Belohnungs- und Bestrafungssignalen erlernen soll, ein vorgegebenes Ziel auf möglichst optimalem Weg zu erreichen. Neben klassischen Benchmark-Problemen wie dem Stabbalancierer hat man die Leistungsfähigkeit von RL-Methoden bereits an einigen beliebten Brett- und Kartenspielen unter Beweis gestellt. Einen Meilenstein stellte 1995 Tesauros TD-Gammon dar – ein gelernter Backgammon-Computergegner, der es mit den weltbesten menschlichen Spielern aufnehmen kann [17]. Weitere Erfolge sind in Blackjack, Othello, Schach und der Dame-Variante Checkers zu verbuchen. Mit diesem Wissen hat man sich auch schon an einigen Videospielen wie Tetris, Arkanoid und PacMan versucht. Moderne Computerspiele bestechen neben immer besserer Grafik auch durch zunehmende Komplexität. Das erschwert jedoch gleichzeitig die Anwendung von RL oder KI allgemein. Insbesondere in umfangreichen Strategiespielen ist es kaum mehr möglich einen Computergegner zu implementieren, der (ohne zu tricksen) einen erfahrenen menschlichen Spieler besiegen kann. Dennoch gibt es auch in diesem Bereich einige Ansätze, unter anderem in Civilization IV [19, 2], Wargus [20] und BosWars [9].[2]
Eine weitere Plattform zur Weiterentwicklung und praktischen Umsetzung von Lernalgorithmen stellt der RoboCup dar. Auch hier hat Reinforcement Learning insbesondere in der Fußball-Simulations-Liga maßgeblich zum Erfolg von siegreichen Mannschaften wie den Brainstormers beigetragen [15]. Dort gewonnene Erkenntnisse lassen sich auch auf andere Computerspiele anwenden, wie in dieser Arbeit demonstriert werden soll.

Am Beispiel eines Weltraum-Shooters wollen wir zeigen, wie Reinforcement Learning eingesetzt werden kann, um bestehende von Hand programmierte Verhalten durch bessere, erlernte Verhalten zu ersetzen. Hierzu identifizieren wir zwei Teilaufgaben, die einen zentralen Bestandteil der Strategie eines Agenten in diesem Spiel ausmachen. Die möglichst optimale Bewältigung beider Teilaufgaben werden wir in jeweils verschiedenen Abstufungen der Kom-

[1] Die deutschen Übersetzungen „Optimierendes Lernen“ oder „Bestärkendes Lernen“ haben sich kaum durchgesetzt.

[2] In den hier genannten Arbeiten wurden diese Spiele nachträglich als Testumgebung für akademische Forschungen verwendet. Über die Anwendung von RL in (kommerziellen) Originalspielen ist dagegen wenig zu finden.

plexität mithilfe von RL-Methoden den Agenten erlernen lassen. Dabei verwenden wir den NFQ-Algorithmus mit neuronalen Netzen als Funktionsapproximator und einer speziellen Anpassung zur dynamischen Skalierung von Ziel- und Ausgabewerten. Sämtliche einzustellende Parameter der zugrundeliegenden Lernaufgaben und des verwendeten Lernverfahrens werden erläutert und ihre jeweilige Einstellung motiviert. Anschließend werden die gelernten Teilverhalten zu einem Gesamtverhalten zusammengefügt und die Verbesserung der Leistung gegenüber der ursprünglichen, aufwändig handkodierten Version evaluiert.

In Kapitel 2 werden theoretische Grundlagen des Reinforcement Learning mit dem hier verwendeten NFQ-Algorithmus erläutert und die Anwendungsdomäne dieser Arbeit, das Spiel Star Ships, vorgestellt. Kapitel 3 beinhaltet die praktische Anwendung in Form von zwei gelernten Teilverhalten sowie die Einbettung dieser Teilverhalten in ein Gesamtverhalten. Abschließend werden im vierten Kapitel eine Zusammenfassung sowie ein Ausblick auf mögliche zukünftige Arbeiten gegeben.

2 Theoretische Grundlagen

In diesem Kapitel werden wir Reinforcement Learning mit einigen Lösungsmethoden sowie das Spiel Star Ships kennenlernen.

2.1 Reinforcement Learning

Reinforcement Learning ist eines der drei Typen des maschinellen Lernens neben dem überwachten Lernen (Generalisierung über einer Beispielmenge von Ein-/Ausgabe-Paaren) und dem unüberwachten Lernen (Strukturierung und Erfassung von Zusammenhängen) [16]. Hierbei soll durch Belohnungen und Bestrafungen ein zuvor spezifiziertes Ziel erreicht werden. Anders als bei den beiden anderen Typen wird dabei zunächst keinerlei Information über eine Lösungsstrategie vorausgesetzt. Eine optimale Strategie wird durch geschicktes Ausprobieren von Lösungen direkt am System oder an einer Simulation selbständig erlernt.

2.1.1 Markovsche Entscheidungsprozesse

Probleme, die beim RL betrachtet werden, sind in der Regel als Markovsche Entscheidungsprozesse (*Markov Decision Process*, kurz: MDP) [11] modelliert. Dabei handelt es sich um ein 4-Tupel $\langle S, A, p, c\rangle$ bestehend aus:

- einer Menge S von Zuständen, dem Zustandsraum,
- einer Menge A von Aktionen,
- einer Wahrscheinlichkeitsverteilung für Zustandsübergänge, $p(s, a, s') = P(s_{t+1} = s' | s_t = s, a_t = a)$ und
- einer Kosten- bzw. Belohnungsfunktion $c(s, a)$. Diese kann auch zusätzlich von dem eintretenden Folgezustand oder beispielsweise nur von der Aktion abhängen (entsprechend $c(s, a, s')$ bzw. $c(a)$).

Ein MDP heißt endlich, wenn die Zustands- und die Aktionsmenge endlich sind.

Für ein MDP wird die sogenannte **Markov-Eigenschaft** vorausgesetzt, welche besagt, dass die Wahrscheinlichkeitsverteilung eines Folgezustands s_{t+1} allein vom aktuellen Zustand s_t und der Aktion a_t abhängt und nicht von der „Geschichte“ des Systems:

$$P(s_{t+1} = j | s_t, a_t) = P(s_{t+1} = j | s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0).$$

Ziel ist es nun, eine optimale Strategie $\pi^* : S \to A$ zu finden, die – ausgehend von jedem beliebigen Zustand – die erwarteten angehäuften Kosten minimiert bzw. die Belohnung maximiert.

2.1.2 Dynamische Programmierung

Ein grundlegender Ansatz zur Lösung von endlichen MDPs ist es, eine Wertfunktion $V(s)$ über den Zuständen zu berechnen, anhand derer die beste Aktion bestimmt werden kann. Für die optimale Wertfunktion V^* gilt das Bellmansche Optimalitätsprinzip:

$$V^*(s) = \min_{a \in A(s)} \sum_{s' \in S} p(s,a,s')(c(s,a) + V^*(s')) \qquad \forall s \in S$$

Ist diese Funktion bekannt so ergibt sich eine stationäre optimale Strategie π^* wie folgt:

$$\pi^*(s) = \arg\min_{a \in A(s)} \sum_{s' \in S} p(s,a,s')(c(s,a) + V^*(s'))$$

Eine Möglichkeit, diese Funktion zu ermitteln ist das Wertiterationsverfahren (*Value Iteration*). Dabei wird die aktuelle Schätzung V_k stetig verbessert, indem der Wert für jeden Zustand gemäß der letzten Schätzung aktualisiert wird:

$$\forall s \in S: \qquad V_k(s) := \min_{a \in A(s)} \sum_{s' \in S} p(s,a,s')(c(s,a) + \alpha V_{k-1}(s'))$$

Der Diskontierungsfaktor $0 \leq \alpha < 1$ sorgt dafür, dass spätere Kosten weniger gewichtet werden als unmittelbar auftretende. Betrachtet man Probleme mit unendlichem Horizont, so ist dieser Faktor unverzichtbar, da sonst Kosten bis ins Unendliche aufsummiert werden können.
Das Verfahren konvergiert für jeden beliebigen Startvektor V_0, es gilt also $\lim_{k\to\infty} V_k = V^*$. Ein Beweis hierfür ist in [3] zu finden.

2.1.3 Q-Learning

Oft ist das zugrundeliegende Modell (die Wahrscheinlichkeitsverteilung der Zustandsübergänge und die Kostenstruktur) eines Prozesses nicht bekannt und kann nur am realen System oder an einer Simulation beobachtet werden. Doch auch in diesem Fall ist es möglich, eine optimale Strategie zu erlernen. Hierzu werden nicht die Werte für Zustände, sondern für Zustands-Aktions-Paare ermittelt. Die Funktion $Q(s,a)$ bildet die erwarteten Kosten ab, wenn im Zustand s die Aktion a gewählt wird. Für die optimale Q-Funktion gilt wiederum die Bellman-Gleichung:

$$\begin{aligned} Q^*(s,a) &= \sum_{s' \in S} p(s,a,s')(c(s,a) + V^*(s')) \\ &= \sum_{s' \in S} p(s,a,s')(c(s,a) + \min_{a' \in A(s')} Q(s',a')) \end{aligned}$$

Daraus lässt sich die zugehörige optimale Strategie ermitteln:

$$\pi^*(s) = \arg\min_{a \in A(s)} Q^*(s,a)$$

Auch hier lässt sich eine an Value Iteration angelehnte Methode anwenden, welche als Q-Lernen (*Q-Learning*) [18] bekannt ist. Um den Erwartungswert über allen möglichen Zustandsübergängen zu bestimmen geht der jeweils neue Wert nur mit einer im Laufe des

Lernvorgangs abnehmenden Gewichtung, der Lernrate γ, gegenüber der alten Schätzung ein (stochastische Approximation):

$$Q_{k+1}(s,a) := (1-\gamma)Q_k(s,a) + \gamma(c(s,a) + \alpha \min_{a' \in A(s')} Q_k(s',a'))$$

Dabei ist s' der beobachtete Folgezustand und $c(s,a)$ die beobachteten Kosten bei Ausführung von a in Zustand s. Das Verfahren konvergiert, wenn jedes Zustands-Aktions-Paar (s,a) unendlich oft besucht wird und die zugehörige Lernrate $\gamma_t(s,a)$ gemäß den Bedingungen der stochastischen Approximation abnimmt: $\sum_{t=0}^{\infty} \gamma_t(s,a) = \infty$ und $\sum_{t=0}^{\infty} (\gamma_t(s,a))^2 < \infty$.

2.2 Funktionsapproximatoren

Die bisher vorgestellten Algorithmen funktionieren mit endlichen MDPs, wenn die einzelnen Werte in Tabellen gespeichert und geändert werden können. In den meisten realitätsnahen Fällen haben wir es jedoch mit unendlichen (kontinuierlichen) Zustandsräumen zu tun. Mit zunehmender Anzahl an Zustandsdimensionen kommt man auch mit Diskretisierung nicht weit, da die Tabellengrößen bis ins Unhandhabbare wachsen würden (der sog. Fluch der Dimensionalität). Auch ist es dann nahezu unmöglich jeden Zustand bzw. jedes Zustands-Aktions-Paar während des Lernens zu besuchen. An dieser Stelle kommen Funktionsapproximatoren ins Spiel. Von ihnen verspricht man sich, dass sie mit kontinuierlichen Zustandsräumen umgehen und im Idealfall sogar über nicht besuchte Zustände generalisieren können.

2.2.1 Lineares Modell

Eine einfache Methode ist es, die Wertfunktion durch eine lineare Funktion zu approximieren:

$$y(\mathrm{x}, \mathrm{w}) = w_0 + \sum_{i=1}^{n} w_i, x_i = \mathrm{w}^T \mathrm{x}$$

Dabei ist $\mathrm{x} = (1, x_1, x_2, \ldots, x_n)^T$ der Eingabevektor und $\mathrm{w} = (w_0, w_1, w_2, \ldots, w_n)^T$ der Parameter- oder Gewichtsvektor. Gesucht ist ein optimaler Parametervektor w^*, sodass die Eingabedaten möglichst genau durch die Funktion approximiert werden. Als Fehlerfunktion dient die Summe der quadratischen Differenzen zum tatsächlichen Wert t über alle Muster:

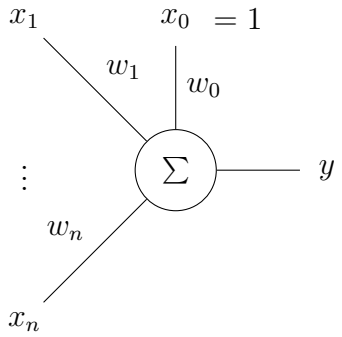

Abbildung 2.1: *lineares Modell*

$$E(\mathrm{w}) := \frac{1}{2} \sum_{p=1}^{P} e^p(\mathrm{w}) = \frac{1}{2} \sum_{p=1}^{P} (y(\mathrm{x}^p, \mathrm{w}) - t^p)^2$$

Entsprechend ist $\mathrm{w}^* = \arg\min_{\mathrm{W}} E(\mathrm{w})$. In diesem Fall kann eine eindeutige Lösung durch Nullsetzen der Ableitung berechnet werden:

$$\mathrm{w}^* = (\mathrm{X}^T\mathrm{X})^{-1}\mathrm{X}^T\vec{t} \qquad \text{mit } \mathrm{X} = \begin{pmatrix} (\mathrm{x}^1)^T \\ \vdots \\ (\mathrm{x}^p)^T \end{pmatrix} \text{ und } \vec{t} = \begin{pmatrix} t^{(1)} \\ \vdots \\ t^{(p)} \end{pmatrix}$$

Das Modell lässt sich erweitern, indem man den Eingabe-Vektor x durch eine Reihe von Basisfunktionen ϕ_i, dem sog. Feature-Vektor $\Phi(\mathrm{x}) = (\phi_1(\mathrm{x}), \phi_2(\mathrm{x}), \ldots, \phi_m(\mathrm{x}))$ transformiert. Die Gesamtfunktion setzt sich dann zusammen aus $y(\mathrm{x}, \mathrm{w}) = \sum_{i=1}^{m} \phi_i(\mathrm{x})\mathrm{w}_i = \Phi(\mathrm{x}) \cdot \mathrm{w}$. Auf diese Weise lassen sich auch nichtlineare Zusammenhänge darstellen und durch geeignete Wahl von Basisfunktionen Vorwissen über die zu approximierende Funktion einbringen. Dieser Ansatz wird unter anderem von CMACs [1] und RBF-Netzwerken [4] aufgegriffen.

2.2.2 Multi-Layer-Perzeptron (Neuronale Netze)

Auch in neuronalen Netzen findet sich das lineare Modell als Grundbaustein wieder. Die Idee ist es, mehrere solcher Bausteine hintereinander zu schalten.

Ein einzelnes Neuron (Perzeptron) j besteht dabei aus einer internen Aktivierung $net_j = \sum_{i=0}^{n} w_{ij} x_i$, in der die gewichtete Summe der eingehenden Werte berechnet wird und einer Aktivierungsfunktion f_{sig}, die diesen Wert für die Ausgabe transformiert (siehe Abbildung 2.2). Als Aktivierungsfunktion kann man eine Schwellwertfunktion (Stufenfunktion) verwenden, die entweder 0 oder 1 ausgibt. In den meisten Fällen wählt man jedoch eine differenzierbare, monoton wachsende Funktion wie die logistische Sigmoidfunktion $f_{sig}(z) = \dfrac{1}{1 + e^{-az}}$.

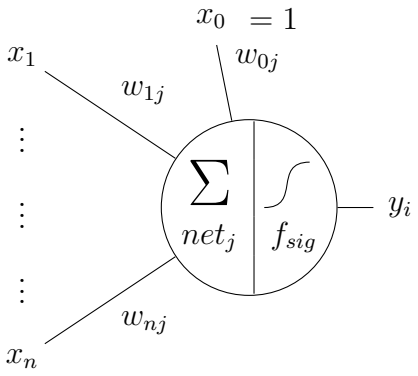

Abbildung 2.2: *Aufbau eines Neurons*

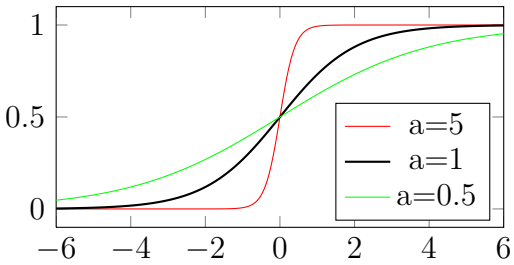

Abbildung 2.3: *Sigmoidfunktion*

Für große a nähert sich diese Kurve der Schwellwertfunktion an, für kleine a ist sie im mittleren Bereich fast linear (siehe Abbildung 2.3). In der Praxis wird dieser Parameter aber nicht explizit verwendet, da er sich auch über den Gewichtsvektor w ausdrücken lässt.

Die Neuronen werden in Schichten (Layer) angeordnet, sodass jedes Neuron einer Schicht mit jedem Neuron aus den benachbarten Schichten verbunden ist, wie in Abbildung 2.4. Der Vektor x repräsentiert die Eingabe-Schicht, wobei es sich hierbei genau genommen nicht um Neuronen handelt. Ganz rechts befindet sich die Ausgabeschicht, die häufig nur aus einem Ausgabe-Neuron besteht. Dazwischen liegen eine oder mehrere verdeckte Schichten

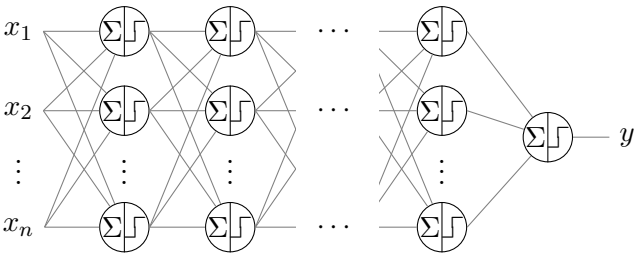

Abbildung 2.4: *Multi-Layer-Perzeptron*

(*hidden layer*). Auch hier wird als Fehlerfunktion üblicherweise die Summe der quadratischen Abweichungen verwendet. Allerdings gibt es nun deutlich mehr Freiheitsgrade durch die einzelnen Kantengewichte, sodass keine eindeutige analytische Lösung mehr gefunden werden kann. Stattdessen werden die Gewichte numerisch angepasst durch $w_{ij}^{neu} := w_{ij}^{alt} - \epsilon \frac{\partial E(\mathrm{w})}{\partial w_{ij}}$ mit einer Lernrate $\epsilon > 0$, in vektorieller Schreibweise: $\mathrm{w}^{neu} := \mathrm{w}^{alt} - \epsilon \nabla E(\mathrm{w})$. Der Gradient $\nabla E(\mathrm{w}) = (\frac{\partial E(\mathrm{w})}{\partial w_1}, \frac{\partial E(\mathrm{w})}{\partial w_2}, \ldots, \frac{\partial E(\mathrm{w})}{\partial w_n})^T$ kann durch ein Gradientenabstiegsverfahren wie *Backpropagation* berechnet werden. Dabei wird der Fehler an der Ausgabeschicht angelegt und von Schicht zu Schicht zurückpropagiert.

Im Allgemeinen ist diese Lösung jedoch nicht optimal, da man auch in einem lokalen Minimum oder auf einem Plateau der Fehlerfunktion landen kann. Ein weiteres Problem birgt die Wahl der Lernrate ϵ. Wählt man sie zu groß, so kann das Verfahren um ein Optimum herum oszillieren, ein zu kleiner Wert dagegen würde die Konvergenz unnötig verlangsamen. Es existieren daher verschiedene Methoden, um diese Lernrate dynamisch anzupassen, unter anderem das von Riedmiller und Braun entwickelte RPROP-Verfahren [14].

Um eine Wertfunktion beim modellbasierten Lernen einzutrainieren legt man die skalierten Werte der Zustandsdimensionen an der Eingabeschicht und den jeweils erwarteten Zielwert am Ausgabeneuron an. Beim Approximieren einer Q-Funktion gibt es dagegen mehrere Möglichkeiten, die Netztopologie aufzubauen, da hier neben den Zuständen auch die Aktionen berücksichtigt werden müssen:

1. Haben die Aktionen ähnliche oder gegensätzliche Auswirkungen wie zum Beispiel unterschiedlich starkes Beschleunigen und Bremsen, bietet es sich an, die Aktionen wie eine weitere Dimension in der Eingabeschicht zu behandeln.

2. Es besteht auch die Möglichkeit, für jede Aktion ein Ausgabeneuron zu reservieren. Bei einem angelegten Zustand kann an einem Ausgabeneuron der Wert für die zugehörige Aktion bzw. für das entsprechende Zustands-Aktions-Paar abgelesen werden. Hierzu müssen jedoch beim Training des Netzes für jeden betrachteten Zustand immer Werte für alle Aktionen vorliegen, um diese simultan an das Netz anlegen zu können.

3. Verbreiteter ist dagegen die Variante, für jede Aktion ein eigenes neuronales Netz zu verwenden, das die Zustandswerte bezüglich dieser Aktion approximiert.

2.3 Neural Fitted Q Iteration

Viele Funktionsapproximatoren – insbesondere neuronale Netze – haben die Eigenschaft, dass im Vergleich zu Tabellen die Anpassung eines Wertes an einer Stelle auch Werte an ganz anderen Stellen beeinflussen kann. Dadurch sind die Bedingungen für den Nachweis der Konvergenz nicht mehr gegeben, sodass diese im Allgemeinen nicht mehr gewährleistet werden kann. Tatsächlich ist auch in der Praxis beobachtet worden, dass beim Online-Lernen mit neuronalen Netzen auch nach langer Zeit keine Konvergenz eintritt, wenn das Netz immer nur mit einzelnen oder wenigen Zustands-Werte-Paaren aktualisiert wird [13].
Hier haben sich die sogenannten Offline- bzw. Batch-Lernverfahren bewährt, bei denen zuerst

online eine Menge von Trainingsmustern gesammelt wird, um diese dann in einem Block in das Netz einzutrainieren.

Hinzu kommt, dass vor allem in realen und realitätsnahen Szenarien Trainingsdaten oft rar und aufwändig zu beschaffen sind. Es wäre daher verschwenderisch, diese nur einmal zu nutzen und dann zu verwerfen. Um dies zu vermeiden wurde für modellfreies Lernen mit Funktionsapproximatoren das *Fitted Q Iteration*-Verfahren [5] entwickelt, welches an *Fitted Value Iteration* [8] für den modellbasierten Fall angelehnt ist. Die Effektivität dieses Verfahrens wurde bereits für verschiedene Regressionsmethoden bestätigt. Wir wollen hier die Variante mit neuronalen Netzen – genannt *Neural Fitted Q* (NFQ) [13] – näher betrachten, wie sie im folgenden Pseudokode ausgeführt ist:

Algorithmus 1: NFQ(D)

input : a set of transition samples D
output : Q-value function Q_N

```
1  init_MLP()→ Q_0
2  for k := 0 to N − 1 do
       // generate Pattern set P
3      P := ∅
4      for l := 1 to #D do
5          input^l := (s^l, a^l)
6          target^l := c(s^l, a^l, s'^l) + α min_b Q_k(s'^l, b)
7          P := P ∪ {(input^l, target^l)}
8      end
       // RPROP-Training → Q_{k+1}:
9      for i := 1 to #RpropEpochs do
10         for l := 1 to #P do
11             net_in[ ] :=scale(input^l[ ])
12             forward_pass() → net_out
13             error := net_out − target^l
14             net_out := error
15             backward_pass()
16         end
17         update_weights()
18     end
19 end
```

Das Verfahren greift die Grundidee des dynamischen Programmierens auf. Geht man von einer zu 0 initialisierten Funktion Q_0 aus, so entsprechen die im ersten Durchlauf berechneten Zielwerte $target^l$ den unmittelbaren Kosten $c(s^l, a^l, s'^l)$ bzw. den Terminalkosten C^+ oder C^-, falls s'^l aus der Menge der positiven oder negativen Terminalzustände $\mathcal{S}^+$ bzw. $\mathcal{S}^-$ ist. Im nächsten Durchlauf werden die Kosten für das Problem mit 2-stufigem Horizont berechnet, dann für den 3-stufigen Horizont und so weiter. N sollte daher mindestens so groß wie die Dauer einer Sequenz gewählt werden.

Die Trainingsdaten (Zustandsübergänge) können natürlich durch rein zufällig ausgewählte Aktionen gewonnen und anschließend eintrainiert werden. Noch effektiver ist es jedoch, die beiden Phasen „Sammeln von Trainingsdaten“ und „Trainieren des neuronalen Netzes“ abzuwechseln, wie in Abbildung 2.5 dargestellt ist. Beim Sammeln der Zustandsübergänge kann dann das zuletzt trainierte Netz evaluiert werden, um die momentan beste Aktion zu ermitteln und diese dann auch auszuführen. Dabei kann eine Explorationswahrscheinlichkeit für die Wahl von zufälligen Aktionen eingeführt werden, sodass weiterhin nach möglicherweise besseren Alternativwegen gesucht wird. Auf diese Weise verbessert sich die Strategie von Iteration zu Iteration.
Alte Trainingsdaten müssen nicht verworfen werden, da sie ja weiterhin gültig sind. Die Trainingsmenge wird daher stetig wachsen.

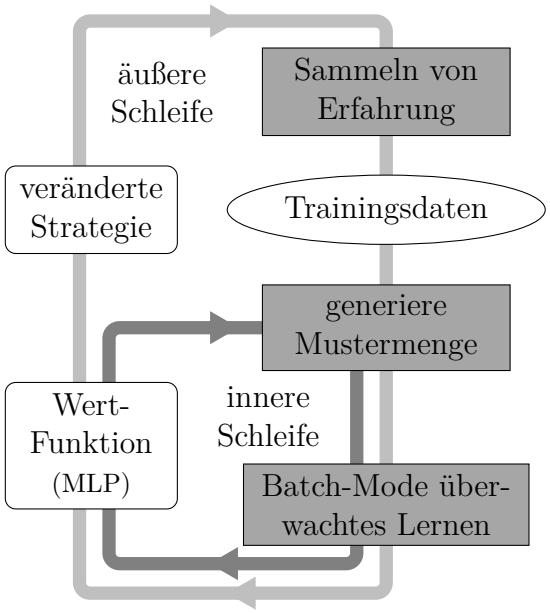

Abbildung 2.5: *Das Batch-RL-Framework*

2.4 Star Ships

In diesem Kapitel werden wir die Anwendungsdomäne dieser Arbeit kennenlernen – das Spiel Star Ships und die zur Interaktion mit dem Spiel geschaffene Schnittstelle.

2.4.1 Das Spiel

Star Ships [6] ist ein an die TV-Serie „Star Trek Enterprise“ angelehntes 2D-Action-Spiel, bei dem zwei Raumschiffe gegeneinander kämpfen. Ziel des Spiels ist es, mithilfe von Offensivwaffen (Phaser und Photonentorpedos) die Schilde des Gegners zu schwächen und letztendlich das Gegnerschiff zu zerstören.

Aufbau und Spielaktionen

Das Universum ist torusförmig aufgebaut, d.h. verschwindet ein Objekt am Rand des rechteckigen Spielfelds, so taucht es am gegenüberliegenden Rand wieder auf.

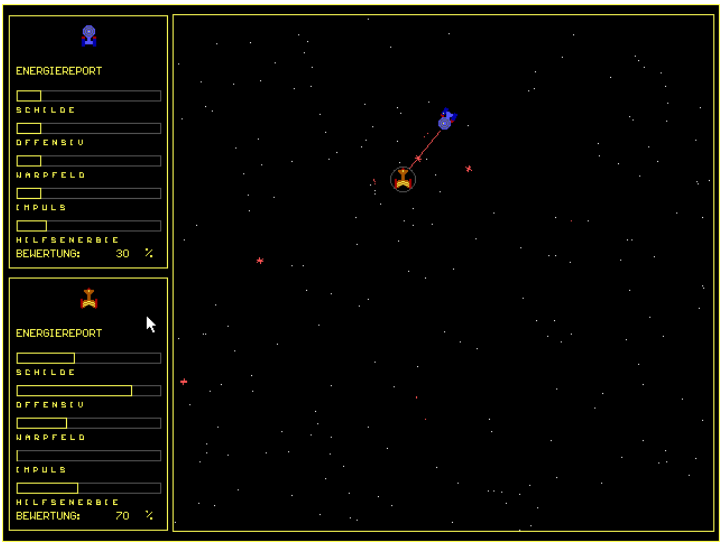

Abbildung 2.6: *Screenshot aus dem Spiel*

Jeder Spieler hat folgende Aktionen zur Auswahl, um sein eigenes Raumschiff zu steuern:

- **Rotation** nach links oder rechts um 15°
- **Impuls** in die Richtung, in die das Schiff ausgerichtet ist
- **Phaser** abfeuern
- **Photonentorpedo** abfeuern
- **Warpsprung** (dabei wird man auf eine zufällige Position im Spielfeld mit Geschwindigkeit $v = 0$ versetzt)

Wird keine Aktion ausgeführt, so fliegt das Schiff mit der aktuellen Geschwindigkeit konstant weiter. Nur über die Impuls-Aktion kann die momentane Geschwindigkeit geändert werden. Um das Schiff zu bremsen muss dieses beispielsweise entgegen der Flugrichtung ausgerichtet werden, um mit dem Impuls dem Geschwindigkeitsvektor entgegenzuwirken.

Energiesystem

Jedes Schiff besitzt fünf Energiesysteme, die zwischen 0 und 100 Energieeinheiten umfassen. Zu Beginn sind alle Systeme bis auf die Hilfsenergie voll aufgeladen, was insgesamt 400 Einheiten entspricht.

1. **Schilde**: Ist diese Energie auf 0% gesunken, so ist das Schiff ungeschützt und explodiert bei einem weiteren Treffer.
2. **Offensiv**: Beide Waffensysteme (Phaser und Torpedos) verbrauchen je 2 Energieeinheiten pro Schuss. Einem getroffenen Schiff werden 6 Einheiten vom Schutzschild, sowie 5 bis 10 Einheiten von einem zufällig gewählten Energiesystem abgezogen. Dabei hängt die Reichweite des Phasers von der Offensivenergie ab. Von jedem Spieler dürfen zur selben Zeit maximal 5 Torpedos unterwegs sein. Sie verschwinden spätestens nach 100 Zeitschritten (etwa 10 Sekunden).
3. **Warpfeld**: Jeder Warpsprung benötigt 10 Energieeinheiten.
4. **Impuls**: Jede Beschleunigung in Form von Impulsen verbraucht eine Energieeinheit. Rotationen verursachen keine Kosten.
5. **Hilfsenergie**: Diese lädt sich allmählich auf. Sie ist außerdem dazu da, um Energieeinheiten zwischen den Systemen zu transferieren. Darauf soll hier jedoch nicht näher eingegangen werden, da der Ausgleich zwischen den Energiesystemen später automatisch durchgeführt werden wird.

Bei einem Zusammenprall beider Schiffe endet die Runde unentschieden.

Das Spiel bietet von Grund auf folgende Spielmodi: Mensch gegen Mensch, Mensch gegen Computergegner (im Folgenden als Spiel-KI bezeichnet) und – zu Demonstrationszwecken – einen Modus, in dem beide Schiffe von der Spiel-KI gesteuert werden. Neu hinzugekommen ist die Möglichkeit, eigene (intelligente) Programme, insbesondere lernende Agenten als Gegner einzusetzen, wie im folgenden Kapitel erläutert wird.

2.4.2 Die Schnittstelle – Das Star Ships Learning Framework (SSLF)

In der „Academic Version“ des Spiels wird eine dateibasierte Schnittstelle bereitgestellt, die es externen Programmen ermöglicht, die Kontrolle über ein oder beide Schiffe zu übernehmen. In jedem Zeitschritt wird der aktuelle Zustand in eine Datei geschrieben, die von einem externen Programm ausgelesen werden kann. In dem überlieferten Zustand sind alle Informationen enthalten, die die aktuelle Spielsituation vollständig beschreiben:

- Anzahl Zeitschritte seit Beginn des Spiels,
- aktueller Spielzustand (Spiel läuft, Spieler 1 gewonnen, Spieler 2 gewonnen oder unentschieden),
- x/y-Position und -Geschwindigkeit[1], Ausrichtung und Energiezustand beider Schiffe,
- Information, ob ein Schiff soeben von einem Phaser und/oder von wie vielen Torpedos es getroffen wurde,
- Anzahl Torpedos und
- für jeden Torpedo: Position, Geschwindigkeit, Alter und von welchem Spieler er stammt.

Abbildung 2.7 zeigt ein Beispiel für einen Systemzustand.

```
1  80,-1
2   257.99138, 794.81247,330, 3.12132, -3.85337,2,0,0
3  54,54,54,53,55
4   499.44496, 293.91657,150, -8.82843, -6.14966,4,0,0
5  53,53,53,52,54
6  6
7   608.08764,  98.28167, -11.50000, -19.91858,62,2
8   218.97259, 201.74761,  -9.08819, -20.91759,51,2
9   486.08764, 189.48090,  -6.50000, -21.25833,49,2
10  734.35221, 264.50037,  -3.91181, -20.91759,47,2
11  456.30989, 264.69205,  -9.65926,   2.58819,36,1
12  299.77229, 716.78702,   7.12132, -10.78157,3,1
```

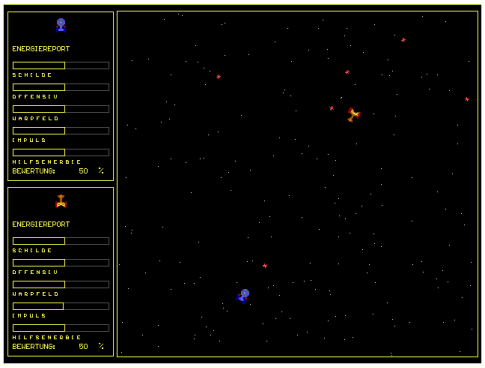

Abbildung 2.7: *Beispielinhalt einer Zustandsdatei mit zugehörigem Screenshot. Zeile 2 und 4 enthalten u.a. Position und Geschwindigkeit der Schiffe, Zeile 3 und 5 die Energiezustände. Es befinden sich 6 Torpedos auf dem Spielfeld (Zeilen 7-12)*

Das externe Programm wiederum hat die Möglichkeit, über Dateien dem Spiel mitzuteilen, welche Aktionen die Schiffe ausführen sollen. Außerdem gibt es eine spezielle Aktion, mit der beliebige Zustände (etwa Startsituationen für Trainingsvorgänge) gesetzt werden können. Für die in den folgenden Kapiteln betrachteten Szenarien werden wir jeweils nur einen Teil dieser vollen Zustandsinformation verwenden. Unter anderem werden wir den Energiezustand und den damit verbundenen Transfer von Energieeinheiten ausblenden. Dieser wird

[1] Genau genommen sind die Zeitschritte in zwei Halbzeitschritte unterteilt, in denen abwechselnd die Aktionen der beiden Spieler ausgeführt werden. Die Geschwindigkeitswerte geben daher an, um wie viele Einheiten sich ein Objekt in einem Halbzeitschritt bewegt.

automatisch vom System übernommen, welches permanent für einen ausgewogenen Energiehaushalt sorgt.

Mit dem Star Ships Learning Framework (SSLF) [7] steht bereits ein Grundgerüst für einen externen Agenten mit den zur Kommunikation mit dem Spiel notwendigen Routinen bereit. Des Weiteren ist hier exemplarisch die spielinterne KI nachimplementiert worden.

Anmerkungen zum SSLF-Systemzustand

Das Spiel ist bereits 1996/97 in der Programmiersprache Turbo Pascal entwickelt worden. Damals ging man intern von einem Bildschirm mit 1000×1000 Einheiten aus. Von diesen Einheiten stehen (abzüglich Ränder und Energie-Report) in x-Richtung 75% und in y-Richtung 96% für das eigentliche Spielfeld zur Verfügung. Es hat demnach einen rechteckigen Wertebereich von 750×960 Einheiten, in denen sich auch die über die Dateischnittstelle übermittelten Koordinaten bewegen. Betrachtet man das Spiel (wie auch in unserem Fall) in einem DOS-Fenster mit einer VGA-Auflösung von 640×480 Pixeln, so wirkt der Bereich für das Spielfeld mit 480×460.8 Pixeln nahezu quadratisch. Das Spielfeld erscheint also gegenüber dem internen Koordinatensystem leicht verzerrt.
Nicht verzerrt sind dagegen die Objekte selbst. Dies hat Einfluss auf die Kollisionsberechnung, da zwei Objekte kollidieren, wenn sich ihre Radien auf dem Bildschirm berühren. Dabei haben Schiffe einen Radius von 10 Pixeln und Torpedos einen Radius von 2.5 Pixeln. Das bedeutet wiederum, dass die Objekte nach dem internen Koordinatensystem unterschiedliche Radien in x- und y-Richtung haben. Weiterhin betrifft dies die Torusförmigkeit des Universums: Objekte werden stets vollständig auf dem Bildschirm angezeigt. Demnach erscheinen sie bereits am gegenüberliegenden Rand, sobald sie den Rand des Spielfelds berühren. Das hat zur Auswirkung, dass Raumschiffe früher verschwinden und an einer anderen Position wieder auftauchen als Torpedos.

3 Praktische Anwendung

Dem Grundgedanken des Reinforcement Learning nach sollte ein Agent allein durch Belohnung und Bestrafung lernen können, ein definiertes Ziel optimal zu erreichen. Demnach müsste es also möglich sein, ein komplettes Spielverhalten zu erlernen, das in der Lage ist, gegen andere Computergegner oder menschliche Spieler zu gewinnen. Hierzu ist der Zustandsraum (Position, Geschwindigkeit und Ausrichtung der Objekte) geeignet zu modellieren. Die Aktionsmenge – bestehend aus sechs Aktionen – ist bereits bekannt. Als Belohnungssignal kann die Information dienen, ob ein Spiel gewonnen oder verloren worden ist.
In der praktischen Umsetzung ergeben sich jedoch eine Reihe von Problemen: Ganze Spiele dauern viel zu lange, sodass das ausschlaggebende Bewertungssignal stark verzögert ist und sich kaum in Bezug zu der Wahl einzelner Aktionen setzen lässt. Ein weiteres Problem ist die Größe des Zustandsraums. Würde man alle Informationen des Systemzustands verwenden und die kontinuierlichen Koordinaten geeignet diskretisieren, so käme man auf eine Größenordnung von etwa 10^{142} unterschiedlichen Zuständen[1].
Daher soll in dieser Arbeit ein anderer Ansatz verfolgt werden: Wir identifizieren spielrelevante Teilverhalten, die sich besonders für die Anwendung von Reinforcement Learning eignen und sich später leicht in ein handkodiertes Gesamtverhalten einbetten lassen.

Anwendungsszenarien

Neben den Phasern ist der Einsatz von Torpedos für den Spielverlauf von entscheidender Bedeutung. Für die Betrachtung innerhalb eines Lernverfahrens sind sie vor allem dadurch interessant, dass sie nicht unmittelbar treffen, sondern zunächst durch den Raum fliegen. Dadurch ergeben sich für den Spieler im Wesentlichen zwei zu bewältigende Aufgaben:

1. Ausweichmanöver: Da Torpedos beim Auftreffen einen nicht unerheblichen Schaden verursachen, gilt es, diesen mithilfe von kostengünstigen Aktionen (Impulsen und Rotationen) auszuweichen.

2. Offensivmanöver: In der Regel sind sowohl das eigene Schiff als auch der Gegner in Bewegung. Die Schwierigkeit besteht nun darin, einen oder mehrere Torpedos genau in den Lauf des Gegners zu schießen, sodass sie ihn möglichst sicher treffen. Dabei sollte berücksichtigt werden, dass der Gegner seine Geschwindigkeit durch Impulse jederzeit ändern kann.

Beiden Szenarien werden wir uns im Folgenden widmen. Zunächst aber werden wir eine Anpassung des in den theoretischen Grundlagen vorgestellten NFQ-Algorithmus hinsichtlich

[1] Es wird ausgegangen von 2 Schiffen und der Maximalzahl von 10 Torpedos, Positionen (x und y) diskretisiert in 2er-Schritten, Geschwindigkeit (x,y) in 1er-Schritten, alle anderen Angaben sind bereits diskret. Berücksichtigt werden sämtliche Positionen, Geschwindigkeiten, Energielevel, Ausrichtungen und Anzahl und Alter von Torpedos: $((960/2)^{2+10} \cdot (750/2)^{2+10}) \cdot (21^{2\cdot 2} \cdot 31^{10\cdot 2}) \cdot 100^{5\cdot 2} \cdot 24^2 \cdot 11 \cdot 100^{10} \approx 10^{142}$

der Skalierung der Ausgabewerte vornehmen, die in all unseren Lernexperimenten verwendet wird.

3.1 NFQ mit dynamischer Zielwertskalierung

Für die von uns verwendeten neuronalen Netze nutzen wir die von Riedmiller entwickelte n++-Bibliothek [12]. Als Aktivierungsfunktion ist die logistische Sigmoidfunktion mit Parameter $a = 1$ eingestellt. Sie hat die Eigenschaft, dass sie für sehr kleine bzw. große x-Werte gegen 0 bzw. 1 konvergiert und die Ausgabewerte dort sehr nah beieinander liegen. Nur für den Eingabebereich von etwa -3 bis $+3$ liefert sie deutlich unterscheidbare Ausgabewerte. Daher ist es sinnvoll, die Eingabedimensionen so zu skalieren, dass sie genau in diesem Bereich liegen.

In diesem Kapitel konzentrieren wir uns allerdings auf die Skalierung der Ausgabewerte, welche eine zentrale Rolle spielt. Wir schlagen hier eine dynamische Skalierungstechnik für die Zielwerte vor, die wir beim überwachten Trainieren des neuronalen Netzes verwenden werden.
Um aus dem Ausgabewert des neuronalen Netzes einen Q-Wert interpretieren zu können und umgekehrt, muss der Wertebereich $(0, 1)$ der Sigmoidfunktion skaliert werden.
Dabei ist zu beachten, dass 0 und 1 Konvergenzwerte der Funktion sind, die praktisch nicht erreicht werden. Es macht demnach wenig Sinn, diese als Zielwerte zu propagieren oder als Ausgabe zu erwarten. Wir nutzen daher effektiv nur das Intervall $[netMin, netMax] = [0.1, 0.9]$.
Weiterhin ist von vornherein nicht bekannt, welchen Wertebereich die Q-Funktion einnehmen wird. Es ist außerdem davon auszugehen, dass dieser sich von einer NFQ-Iteration zur nächsten ändern wird. Um dieses Problem zu umgehen verwenden wir eine Form von **dynamischer Skalierung**:

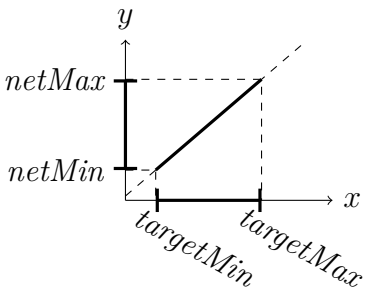

Abbildung 3.1: *Skalierung*

Sei hierfür x der tatsächliche Q-Wert und y der Zielwert für das neuronale Netz. Wir stellen eine bijektive lineare Funktion $f_{scale}\colon [targetMin, targetMax] \to [netMin, netMax], x \mapsto y$ auf, wie sie auch in Abbildung 3.1 dargestellt ist:

$$\begin{aligned} y &= \frac{(x - targetMin)}{(targetMax - targetMin)} \cdot (netMax - netMin) + netMin \\ &= \frac{x \cdot (netMax - netMin)}{targetMax - targetMin} - \frac{targetMin \cdot (netMax - netMin)}{targetMax - targetMin} + netMin \\ &= A \cdot x + B \end{aligned}$$

Dabei ist $A = \frac{netMax - netMin}{targetMax - targetMin}$ die Steigung und $B = -\frac{targetMin \cdot (netMax - netMin)}{targetMax - targetMin} + netMin$ der

Achsenabschnitt. Entsprechend gestaltet sich die Umkehrfunktion f_{scale}^{-1}:

$$\begin{aligned} x &= \frac{y - netMin}{netMax - netMin} \cdot (targetMax - targetMin) + targetMin \\ &= \frac{y - B}{A} \end{aligned}$$

Von entscheidender Bedeutung ist nun, dass die Parameter A und B in jeder NFQ-Iteration dynamisch angepasst werden. Die Rückskalierung f_{scale}^{-1} wird bei der Berechnung des minimalen Q-Werts eines Folgezustands benötigt, um die Zielwerte der Mustermenge P zu bestimmen. Aus den minimalen und maximalen Zielwerten ergibt sich dann unmittelbar das neue Zielintervall, woraus die Parameter A und B berechnet werden können. Vor dem anschließenden Training des neuronalen Netzes werden die Zielwerte anhand der aktuellen Parameter auf das effektive Netzintervall [*netMin*,*netMax*] skaliert.
Wo die Skalierung genau Anwendung findet, wird an nachfolgendem Pseudokode (Algorithmus 2) deutlich.
Die Vorteile dieser dynamischen Skalierungsmethode sind:

- Der Zielwertbereich des Netzes von [0.1,0.9] wird stets voll ausgeschöpft.
- Das Problem der vorab unbekannten Q-Werte wird umgangen.
- Das Verfahren ist unabhängig von der Anzahl N der NFQ-Iterationen.
- Die lineare Skalierung ist ohne großen Rechenaufwand durchführbar.
- Es sind Erweiterungen denkbar, die eine nicht-lineare Skalierung verwenden, um eine möglicherweise ungleichmäßige Verteilung der Ausgabewerte zu berücksichtigen.

Algorithmus 2: learn_behaviour()

```
1  init_net()
2  init_scaling_params()            // e.g. assuming [targetMin, targetMax] = [0, 1]
3  netMin := 0.1, netMax := 0.9
4  repeat
5    repeat
6      set_startsituation()
7      repeat
8        chose_best_action()        // exploit current net: a = arg min_b Q(s, b)
9        or chose_random_action()   // with exploration probability ϵ
10     until s ∈ S+ or s ∈ S−
11   until enough transitions collected ↪ D
12   reinit_net()                   // optional, may be omitted
13   reinit_scaling_params()
     // NFQ-Iterations:
14   for k := 0 to N − 1 do
15     for all l ∈ D do
16       input^l := (s^l, a^l)
```

17 $$target_Q^l := \begin{cases} C^+ & \text{if } s'^l \in \mathcal{S}^+ \\ C^- & \text{if } s'^l \in \mathcal{S}^- \\ c(s^l, a^l, s'^l) + \alpha \min_b Q_k(s'^l, b) & \text{else} \end{cases}$$
`// with` $Q_k(s'^l, b) = f_{scale}^{-1}($`get_current_net_value`$(s'^l, b))$

18 **end for**

`// apply scaling:`

19 $targetMax := \max_{l \in D} target_Q^l$

20 $targetMin := \min_{l \in D} target_Q^l$

21 $A := \dfrac{netMax - netMin}{targetMax - targetMin}$

22 $B := -\dfrac{targetMin \cdot (netMax - netMin)}{targetMax - targetMin} + netMin$

23 **for all** $l \in D$ **do**

24 $target^l := f_{scale}(target_Q^l)$

25 **end for**

`// supervised batch-mode learning using` $\{(input^l, target^l) | l \in D\}$`:`

26 reinit_net()

27 RPROP_training() $\rightarrow Q_{k+1}$

28 **end for**

29 evaluate_current_net() `// #EvalSeqs start situations without exploration`

30 **until** behavior successfully learned

3.2 Ausweichmanöver

Zunächst wollen wir erlernen, einem Torpedo auszuweichen, der das eigene Schiff bedroht (einfaches Ausweichmanöver). Da in einem Spiel jedoch oft mehrere Torpedos gleichzeitig unterwegs sind, deren Laufbahnen sich mit der des Raumschiffs kreuzen können, werden in einem zweiten Schritt auch Überlegungen angestellt, wie man mehrere Torpedos berücksichtigen kann (komplexes Ausweichmanöver).

3.2.1 Einfaches Ausweichmanöver

Das Erlernen eines Teilverhaltens gliedert sich in folgende Abschnitte:

- Zunächst werden Kriterien festgelegt, nach denen die Startsituationen konstruiert werden. Diese werden später zum einen für das Training und zum anderen für die Evaluation der Leistung des Agenten benötigt.
- Dann wird das Problem als MDP formuliert. Dazu gehört in erster Linie eine möglichst kompakte Repräsentation des Zustandsraums, die Definition der Aktionen und des Kostenmodells.
- Anschließend werden die Parameter des verwendeten NFQ-Verfahrens sowie des Funktionsapproximators (in unserem Fall des neuronalen Netzes) festgelegt.

Generierung der Startsituationen

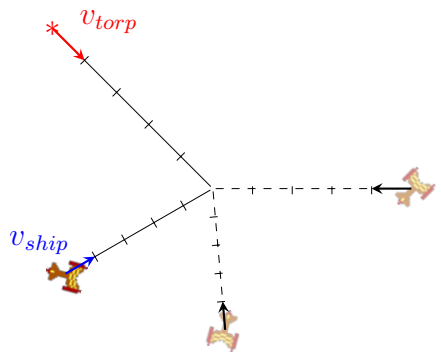

Abbildung 3.2: *Startsituationen*

Die Startsituationen sind so ausgelegt, dass Torpedo und Raumschiff unweigerlich zusammenstoßen werden, sofern das Schiff keine Impulsaktionen ausführt. Sowohl die Geschwindigkeit des Torpedos als auch des Raumschiffs ist variabel, die x- und y-Komponenten der beiden Geschwindigkeiten bewegen sich in einem Intervall von $[-8, +8]$ bzw. $[-4, +4]$. Dadurch ergeben sich automatisch verschiedene Winkel, in denen Torpedo und Raumschiff aufeinandertreffen, wie in Abbildung 3.2 dargestellt ist. Auch die initiale Ausrichtung des Schiffes ist zufällig. Zwischen Startzustand und möglicher Kollision liegen mindestens sechs und bis zu 12 Zeitschritte, in denen der Agent reagieren kann.

Eine Episode ist beendet, sobald das Schiff getroffen ist (negativer Ausgang) oder mit momentaner Geschwindigkeit ohne weitere Aktion dem Torpedo entkommen kann (positiver Ausgang).

Formulierung als MDP

Der hier betrachtete MDP (siehe Kapitel 2.1.1) besteht aus folgenden Komponenten:

Zustandsraum S**:** In der Regel gibt es viele Möglichkeiten, anhand von Rohdaten einen abstrakten Zustandsraum zu definieren. Dieser kann die Effektivität des Lernverfahrens und die Leistung des gelernten Verhaltens maßgeblich beeinflussen. Es liegt daher in der Hand des Programmierers, einen geeigneten Zustandsraum zu finden. Ein Kriterium ist dabei die Anzahl der Dimensionen. Oft kann es sinnvoll sein, wenig relevante Informationen wegzulassen, um Dimensionen einzusparen. Zwar ist damit die Markov-Eigenschaft verletzt, diese geringfügige Verletzung kann aber durch ein robustes Lernverfahren toleriert werden. Rein formal bewegen wir uns hier vom MDP weg zu einem partiell observierbaren MDP (POMDP). Doch auch hierfür gibt es in der Literatur Beispiele, in denen POMDPs erfolgreich mit MDP-Methoden (approximativ) gelöst wurden [10]. Wir werden hier ebenso den Ansatz verfolgen, den POMDP wie einen MDP zu behandeln und eine deterministische Strategie nur in Abhängigkeit vom aktuellen Zustand zu erlernen.
Für dieses Szenario werden wir zwei verschiedene Varianten der Zustandsraummodellierung betrachten und vergleichen:

(a) In dieser Variante beschränken wir uns zu Testzwecken auf drei Dimensionen: die Ausrichtung des Schiffes relativ zum Torpedo α und relativ zur eigenen Geschwindigkeit β sowie die Anzahl n der Zeitschritte bis zur Kollision.

(b) Diese Variante konstruiert ein schiffszentrisches Koordinatensystem, sodass die x-Achse durch das Schiff und den Torpedo verläuft. Die vier Dimensionen sind hier die Ausrichtung des Schiffes α, der Abstand d vom Torpedo zum Schiff und die relative Geschwindigkeit $v_{torp} - v_{ship}$ (x- und y-Wert).

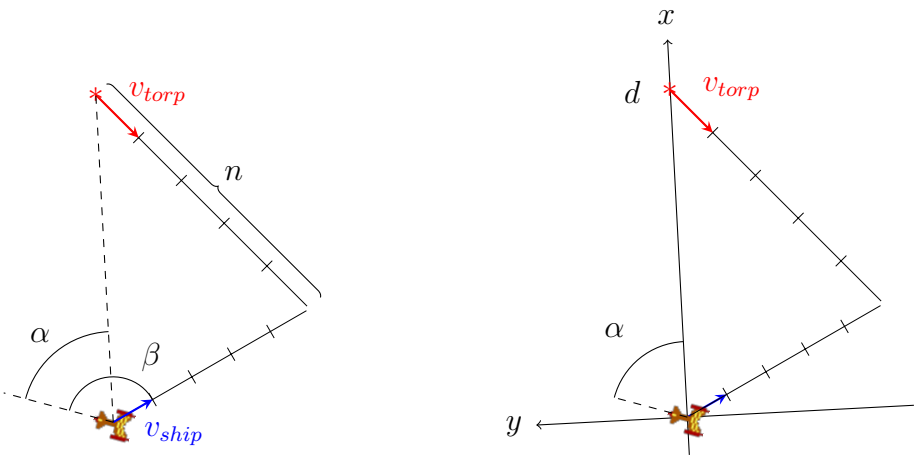

Abbildung 3.3: *Varianten (a) und (b) für die Modellierung des Zustandsraums*

Aktionen A**:** Für das Ausweichmanöver sind die beiden Offensiv-Aktionen nicht relevant. Auch den Warpsprung, der uns mit großer Wahrscheinlichkeit aber mit hohen Kosten vor dem Torpedo retten würde, wollen wir für das Lernen ignorieren, denn unser Ziel ist es, dem Torpedo unter Einsatz von möglichst wenig Schiffsenergie auszuweichen. Wir beschränken uns also auf die Aktionen $A = \{\text{Linksrotation, Impuls, Rechtsrotation}\}$.

Modell p**:** Das durch die Spiel-Engine realisierte Transitionsmodell ist für das aktuell betrachtete Szenario deterministisch, da der Gegner hier keinen Einfluss hat. Der lernende Agent kennt das Modell jedoch nicht, weshalb wir das modellfreie Erlernen einer Q-Funktion gewählt haben.

Kosten c**:** Als Maß für die unmittelbaren Kosten der Aktionen eignet sich in diesem Spiel der Energieverbrauch. Unter diesem Aspekt kostet ein Impuls eine Energie-Einheit, während die Rotationen kostenfrei sind. Um auch den Zeitfaktor mit einzubringen (der Agent soll möglichst schnell ausweichen), weisen wir dennoch einer Rotation Kosten von 0.25 zu. Schafft es der Agent nicht, dem Torpedo auszuweichen (Folgezustand s' ist die Kollision), so entstehen dem Agenten zusätzlich Terminalkosten von durchschnittlich[2] 13.5 in Form von abgezogener Energie.

Parameter des Lernalgorithmus

Der verwendete Algorithmus wurde bereits im vorigen Kapitel vorgestellt. Hier müssen nun lediglich noch einige Parameter festgelegt werden.

Netztopologie: Wir verwenden ein neuronales Netz mit vier bzw. fünf Eingabedimensionen, einer verdeckten Schicht mit 10 Neuronen und einem Ausgabeneuron. Die Eingabeschicht beinhaltet neben den Dimensionen des Zustandsraums eine Eingabe für die Aktion. Dies bietet sich an, da die Rotation nach links oder rechts gegensätzliche Wirkung haben während der Impuls bezüglich der Rotation neutral ist. Für die Aktionen

[2]Die tatsächlich abgezogene Energie liegt zufällig zwischen 6+5 und 6+10 (siehe Kapitel 2.4.1). Daraus ergibt sich der Erwartungswert 13.5. Zwar wäre es möglich, den jeweiligen tatsächlichen Wert zu verwenden und damit gleichzeitig das stochastische Energiemodell zu lernen, auf die dadurch entstehende Varianz kann hier aber durch Kenntnis des Mittelwertes verzichtet werden.

Linksrotation, Impuls und Rechtsrotation werden entsprechend die Eingabewerte -3, 0 und $+3$ gesetzt.

Überwachtes Lernen: In den von uns gewählten 500 RPROP-Lernepochen werden die anfangs zufällig initialisierten Gewichte des neuronalen Netzes iterativ korrigiert, sodass die Netzwerte möglichst genau mit den zuvor berechneten Zielwerten übereinstimmen. Gemäß dem RPROP-Verfahren wird dabei die Lernrate des Netzes (innerhalb eines vorgegebenen Intervalls von [0.05,5]) stets angepasst.

Diskontierung: Zwar sind alle hier betrachteten Sequenzen endlich, aber dennoch können zyklenartige Übergänge auftreten. Beispielsweise durch wiederholte Hin- und Herbewegung können gleiche oder sehr ähnliche Zustände mehrfach besucht werden, auch über mehrere ähnliche Sequenzen hinweg. Durch die generalisierende Eigenschaft des neuronalen Netzes werden ähnliche Zustände auch mit ähnlichen Werten belegt. Da jeder Zustandsübergang mit positiven Übergangskosten behaftet ist, würden sich die entsprechenden Q-Werte innerhalb der NFQ-Iterationen stets vergrößern.
Dieses Problem kann durch einen Diskontierungsfaktor $\alpha < 1$ unterbunden werden. In unserem Fall hat sich ein Wert von 0.95 bewährt.

Exploration: Wir verwenden während des gesamten Lernverlaufs eine Explorationsrate von $\epsilon = 10\%$.

NFQ-Iterationen: Auch wenn die meisten Sequenzen weniger als zehn Zeitschritte dauern, schadet es nicht, mit einem größeren Wert von $N = 20$ Iterationen zu arbeiten. Die erlernte Wertfunktion wird dadurch robuster.

Trainingsdaten: Einen Durchlauf der äußersten Schleife des Lernalgorithmus bezeichnen wir im Folgenden als **Trainingseinheit** (TE). Sie umfasst in diesem Fall das Sammeln von 500 weiteren Zustandsübergängen (vgl. Kriterium „*enough transitions collected*“) und das anschließende Training eines neuen neuronalen Netzes. Alle bisher gesammelten Übergänge verbleiben in der Menge D und werden in jeder weiteren Trainingseinheit verwendet. Dies wird auch als *growing batch*-Lernverfahren bezeichnet.

Ergebnisse

Für einen fairen Vergleich wurden beide Varianten auf der gleichen Menge von 5000 Startsituationen trainiert. Nach jeder Trainingseinheit wurde das neu trainierte Netz anhand von 50 immer gleichen Startsituationen evaluiert. Die Erfolgsrate in Abbildung 3.4 gibt an, in wie vielen Fällen dem Torpedo ausgewichen werden konnte.

Mit beiden Varianten ist ein recht schneller Anstieg der Leistungsfähigkeit zu beobachten. Die Variante (b) mit dem vollen Zustandsraum erzielte mit bis zu 86% ein besseres Ergebnis als die Variante (a) mit bis zu 82%. Der Informationsgehalt der weiteren Dimension ist demnach nicht vernachlässigbar.
In beiden Fällen wurde die Erfolgsrate im weiteren Verlauf nahezu konstant gehalten, jedoch nicht mehr verbessert. Leichte Schwankungen sind durch das jeweils zufällig neu initialisierte neuronale Netz zu erklären.

In den erfolglosen Fällen wird außerdem durch entsprechend hohe (zweistellige) Q-Werte signalisiert, dass dem Torpedo allein mit Rotationen und Impulsen nicht ausgewichen werden

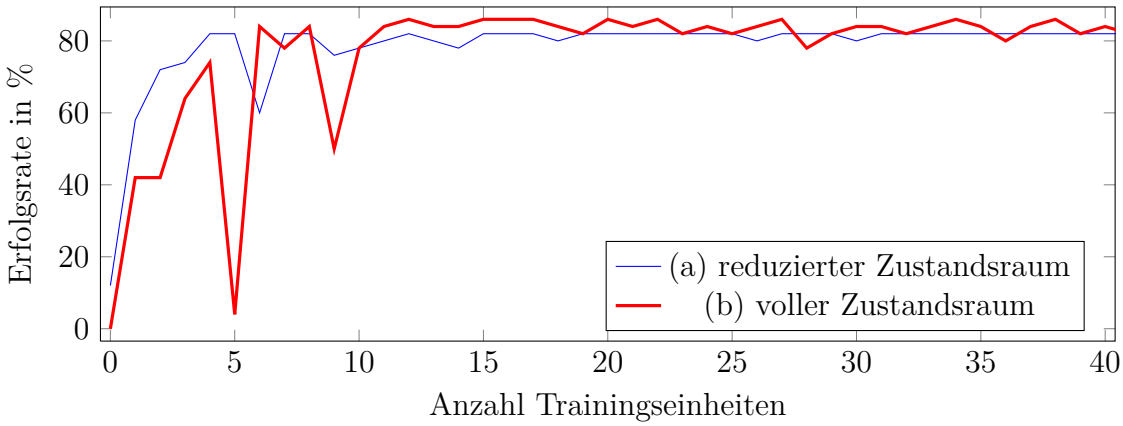

Abbildung 3.4: *Ergebnisse für Varianten (a) und (b) des Zustandsraums. Eine Trainingseinheit umfasst 500 gesammelte Zustandsübergänge, was etwa 100 Episoden entspricht.*

kann. Eingebettet in ein handkodiertes Gesamtverhalten könnte man anhand dieser Werte einen Warpsprung erwägen, der mit Kosten von 10 erwartungsgemäß günstiger wäre als die durch einen Torpedo-Treffer durchschnittlich verlorenen 13.5 Einheiten.[3]

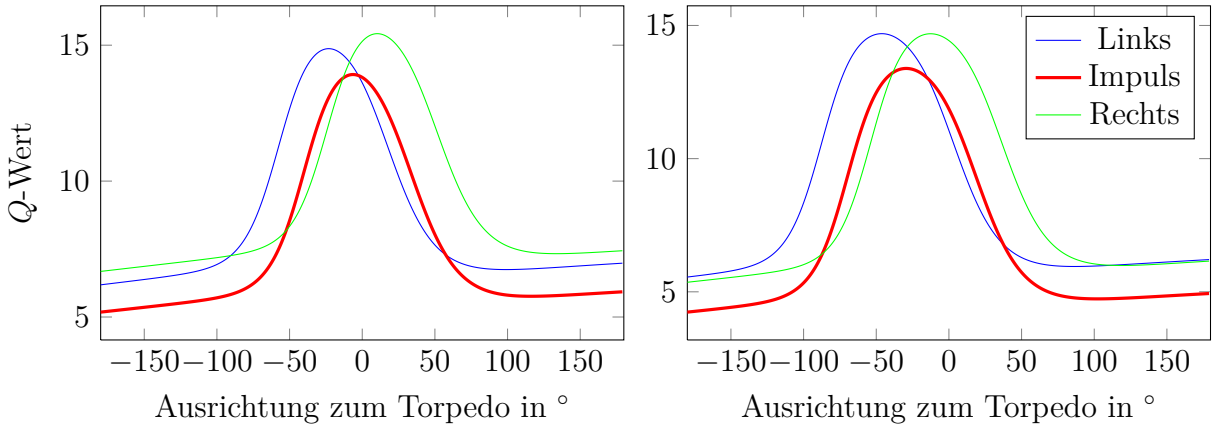

Abbildung 3.5: *Ausschnitte der erlernten Q-Funktion mit Variante (b) der Zustandsraummodellierung nach TE 38. Die Parameter sind: Abstand $d = 120$, Differenzgeschwindigkeit $v_x = -10$ und $v_y = 0$ (links) bzw. $d = 90$, $v_x = -10$ und $v_y = -1.5$ (rechts).*

Die in dem Netz gespeicherte mehrdimensionale Wertfunktion lässt sich ausschnittweise plotten, wenn man einzelne Parameter festsetzt, wie in Abbildung 3.5. Im linken Diagramm würden Schiff und Torpedo geradlinig aufeinander zufliegen. Gemäß der Wertfunktion ist der Impuls für dem Torpedo abgewandte Ausrichtungen die sinnvollste Aktion. Ist das Schiff dem Torpedo zugewandt, so muss es sich erst von diesem wegdrehen. Da die Wahrscheinlichkeit für ein erfolgreiches Ausweichen in diesem Fall gering ist, nimmt die Wertfunktion entsprechend größere Werte an. Steuern Schiff und Torpedo in einem anderen Winkel aufeinander zu (rechtes Diagramm), so ist dieses lokale Maximum der Wertfunktion verschoben. Es fällt auf, dass das Maximum der Impuls-Kurve in einem kleinen Bereich in der Mitte

[3]Hierbei werden jedoch weitere Aspekte vernachlässigt. Möglicherweise wird dadurch eine strategisch günstige Position zum Abschießen des Gegners aufgegeben oder man landet zufällig unmittelbar in der Schussbahn weiterer Torpedos.

unterhalb der Rotationen liegt, was vermutlich in einer nicht optimalen Aktion resultiert. Dies könnte daran liegen, dass sich die Kosten bei Hin- und Herrotationen gegenseitig hochschaukeln (siehe Argumentation zur Wahl des Diskontierungsfaktors), während Impulse keine zyklenartigen Zustandsübergänge verursachen und deren Werte von diesem Phänomen daher unberührt bleiben. Allerdings tritt dieser Fehler nur in einem Bereich des Zustandsraums auf, in dem den hohen Werten nach zu urteilen ohnehin kein Ausweichen mehr möglich wäre.

3.2.2 Komplexes Ausweichmanöver

Es gibt verschiedene Möglichkeiten, sich dem Problem mit mehreren Torpedos anzunähern:

1. Die schnellste Lösung wäre es, einfach immer nur den zuerst eintreffenden Torpedo zu betrachten und das beste zuvor eintrainierte Netz für einen Torpedo heranzuziehen. Das Vernachlässigen der anderen Torpedos kann jedoch dazu führen, dass man aufgrund des Ausweichmanövers unmittelbar und möglicherweise unausweichlich in die Flugbahn eines anderen Torpedos gerät, selbst wenn es vielleicht möglich gewesen wäre, mit einem geschickteren Manöver beiden Torpedos auszuweichen.

2. Man könnte die Q-Werte für jeden relevanten Torpedo einzeln ermitteln und für jede Aktion aufsummieren. Anschließend würde man die Aktion wählen, deren Summe den kleinsten Wert hat. Dabei muss jedoch betont werden, dass es sich bei den einzelnen Werten nur um Schätzungen handelt. Oft entscheiden bereits für einen Torpedo nur kleine Wertdifferenzen darüber, welche Aktion als beste ausgewählt wird. Die Summe über mehrere Schätzungen an verschiedenen Stellen des Zustandsraums wird also selten dem tatsächlichen Wert des Verbund-Zustands entsprechen.

3. Eine Variation davon besteht darin, für jeden Torpedo einzeln die beste Aktion zu ermitteln. Anschließend wird die Aktion ausgeführt, die am häufigsten als beste Aktion auftritt. Bei Gleichstand könnte man Torpedos, die näher sind, stärker gewichten.

4. Wir werden hier den Weg einschlagen, den Agenten jeweils ein neuronales Netz für eine bestimmte Anzahl Torpedos (bis zu drei) trainieren zu lassen.

Zunächst werden wir also neuronale Netze für zwei und drei Torpedos trainieren. Anschließend vergleichen wir die Leistungsfähigkeit dieser vier Ansätze im Fall von drei Torpedos.

Die Startsituationen werden ähnlich wie im vorigen Kapitel generiert, mit dem Unterschied, dass nun zunächst zwei und später drei Torpedos gleichzeitig aus unterschiedlichen Richtungen auf das Raumschiff zusteuern. Sie haben ebenfalls verschiedene zufällige Geschwindigkeiten und können auch versetzt auftreffen (siehe Abbildung 3.6). Damit soll ein möglichst großes Spektrum an denkbaren Konstellationen abgedeckt werden.

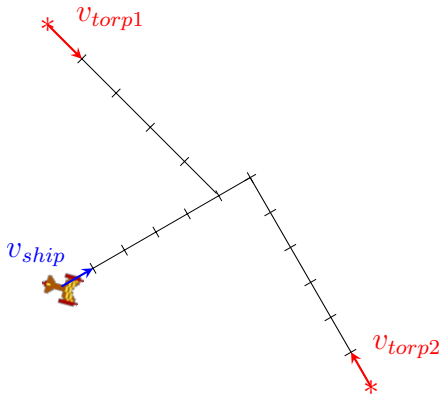

Abbildung 3.6: *Startsituation mit zwei Torpedos*

Da wir jeweils nur ein Netz für eine bestimmte Anzahl Torpedos trainieren wollen, enden Situationen bereits, sobald nur einer der Torpedos trifft (wie bisher mit Terminalkosten von $C^- = 13.5$) bzw. erst, wenn das Schiff allen Torpedos ausgewichen ist ($C^+ = 0$).

Aufgrund der besseren Ergebnisse verwenden wir nun ausschließlich die Variante (b) des Zustandsraums und konstruieren entsprechend für jeden Torpedo ein Koordinatensystem. Somit kommen für jeden weiteren Torpedo vier Eingabedimensionen hinzu. Um eine einheitliche Reihenfolge festzulegen werden die Torpedos nach (zeitlichem) Abstand (Zeitschritte bis zur Kollision, bei Nicht-Kollision entscheidet die Nähe des Vorbeifliegens) sortiert.
Die Anzahl der verdeckten Neuronen wird auf 15 bzw. 20 erhöht, ansonsten verwenden wir die gleichen Einstellungen und Lernparameter wie in Kapitel 3.2.1.

Ergebnisse für zwei Torpedos

In Abbildung 3.7 ist der Lernverlauf für das Training mit zwei Torpedos dargestellt. Der Anstieg der Leistung ist im Vergleich zum einfachen Ausweichmanöver zunächst langsamer, da hier mehr Zustandsübergänge gesammelt werden müssen, um den Zustandsraum mit vergleichbarer Dichte abzudecken. Dennoch wurde in diesem Szenario eine Erfolgsrate von bis zu 76% erzielt.

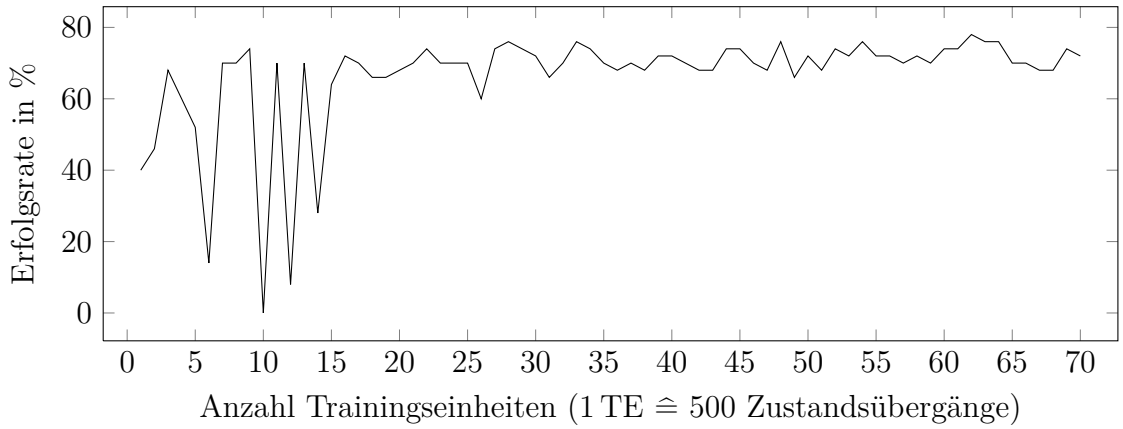

Abbildung 3.7: *Lernverlauf für zwei Torpedos, evaluiert auf 50 Startsituationen*

Ergebnisse für drei Torpedos

Auch in diesem komplexen Szenario mit drei Torpedos und entsprechend 12 Zustandsdimensionen konnte ein beachtlicher Lernerfolg von bis zu 68% erzielt werden, wie in Abbildung 3.8 zu sehen ist. Es ist davon auszugehen, dass unter den Startsituationen mit drei Torpedos aus zufälligen Richtungen einige Situationen sind, in denen nicht allen drei Torpedos gleichzeitig ausgewichen werden kann. Wie bereits beim einfachen Ausweichmanöver kann die Ausweglosigkeit dieser Zustände den entsprechend hohen Q-Werten entnommen werden. Der Agent bevorzugt in diesen Fällen die kostengünstigeren Rotationen, da er davon ausgeht, dass mit Impulsen nicht allen Torpedos ausgewichen werden kann.

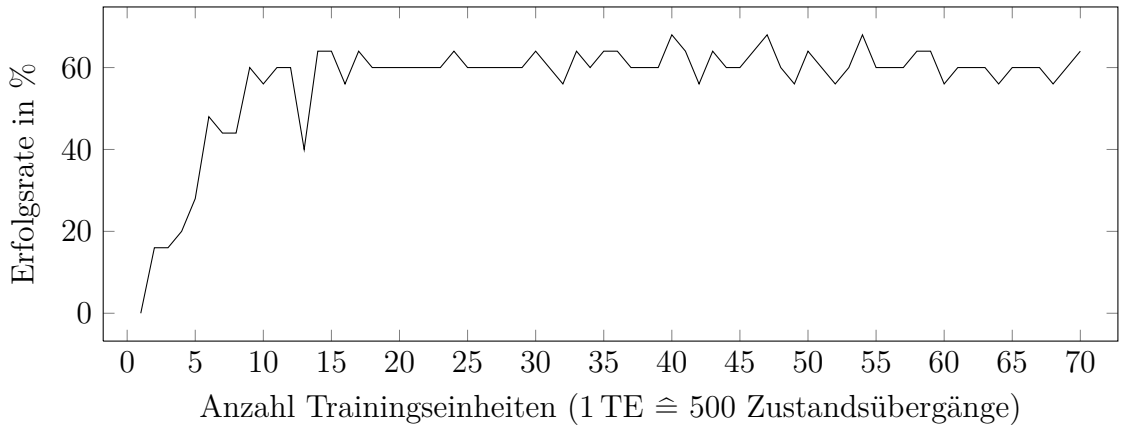

Abbildung 3.8: *Lernverlauf für drei Torpedos*

Empirischer Vergleich der betrachteten Ansätze

In einem weiteren Versuch wurden die zu Beginn dieses Kapitels vorgestellten Ansätze zur Handhabung mehrerer Torpedos empirisch verglichen. Die Startsituationen wurden nach den selben Kriterien wie für das Training gegen drei Torpedos generiert.
Es wurde jeweils eines der besten Netze aus dem Training mit einem bzw. drei Torpedos verwendet. Für die Varianten 2 und 3 wurden nur die Torpedos betrachtet, die noch treffen würden. In Variante 3 erhielt die jeweils beste Aktion einen Punkt multipliziert mit dem Kehrwert des zeitlichen Abstands. Die Ergebnisse sind in Abbildung 3.9 zu sehen. Das gelernte Verhalten schneidet mit 59% besser ab als die anderen Ansätze.

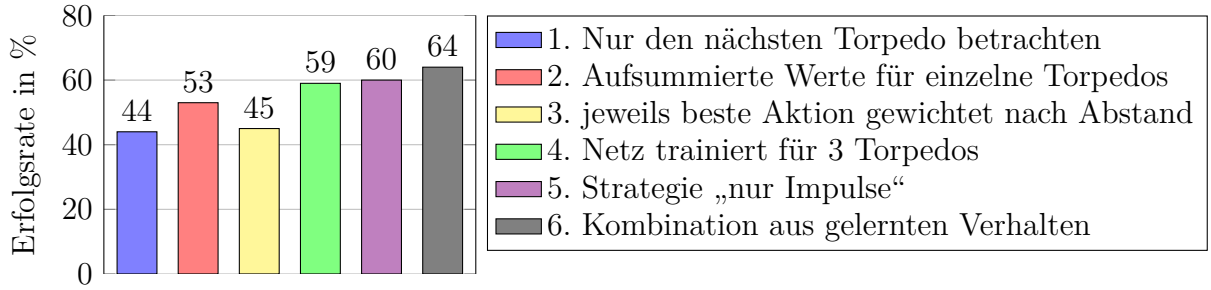

Abbildung 3.9: *Vergleich der Ansätze für mehrere Torpedos auf einer Menge von 100 Startsituationen*

Aus der Beobachtung heraus, dass in den erfolgreichen Sequenzen überwiegend die Impuls-Aktion ausgeführt wird, entstand die Motivation für ein primitives handkodiertes Verhalten, das ausschließlich Impulse ausführt. Es überrascht zunächst, dass dieses Verhalten mit 60% sogar erfolgreicher ist als das gelernte Verhalten. Allerdings unterscheiden sich die beiden Varianten in den zum Ausweichen benötigten Kosten. Während das Verhalten 5 in erfolgreichen Sequenzen durchschnittlich etwa 4.18 Impulsaktionen ausführen muss, hat das Verhalten 4 gelernt, energieeffiziente Aktionen einzusetzen. Es benötigt neben den kostenfreien Rotationen im Durchschnitt nur etwa 3.7 Impulse (12% weniger).
Ferner sind die Mengen der erfolgreich bewältigten Situationen nicht identisch. Es gibt also sowohl Startsituationen, in denen das reine Impuls-Verhalten scheitert und das gelernte Verhalten erfolgreich ist, als auch den umgekehrten Fall.

Hin und wieder sind Situationen zu beobachten, in denen das gelernte Verhalten bereits zwei Torpedos erfolgreich ausgewichen ist, aber mit dem Flüchten vor dem letzten Torpedo in Bereiche des Zustandsraums eintritt, für die das neuronale Netz nicht ausreichend trainiert ist und daher suboptimale Aktionen ausführt. In diesem Fall wäre es sinnvoller, die nicht mehr relevanten Torpedos auszublenden und das neuronale Netz für entsprechend einen oder zwei Torpedos heranzuziehen. Dieser Ansatz wurde in der Variante 6 realisiert, welche mit einer Erfolgsrate von 64% tatsächlich noch bessere Ergebnisse erzielt. Zudem liegen die Kosten pro erfolgreicher Sequenz hiermit bei 3.5 (17% weniger als beim reinen Impulsverhalten).

3.3 Offensivmanöver

Während es beim Ausweichmanöver darum ging, sich grob vom Torpedo abzuwenden und mithilfe von Impulsen die Flugbahn zu ändern, ist beim Offensivmanöver eine präzise Ausrichtung gefordert, um einen Torpedo gezielt auf den Gegner abzufeuern.
Bei diesem Szenario bietet sich ein inkrementelles Vorgehen an: Zunächst soll der stehende Agent versuchen, ein stehendes Raumschiff zu treffen (Voruntersuchungen). Anschließend werden wir die Raumschiffe in konstante Bewegung versetzen (einfaches Offensivmanöver). Schließlich soll auch der Gegner seine Geschwindigkeit anhand einer stationären Strategie ändern können (komplexes Offensivmanöver).

3.3.1 Voruntersuchungen

Beide Raumschiffe werden mit zufälliger Position und Ausrichtung platziert. Der Agent hat die Aktionen Links- und Rechtsrotation sowie das Abfeuern eines Torpedos zur Verfügung. Mit dem Abschuss des Torpedos endet die Sequenz (nach einem Timeout von 15 Schritten wird der Abschuss erzwungen). Für den Agenten ergeben sich Terminalkosten in Abhängigkeit davon, ob der Torpedo getroffen hat (Belohnung bzw. negative Kosten von $C^+ = -11.5$) oder nicht ($C^- = +2$). Sie errechnen sich aus den zwei Energieeinheiten für das Abfeuern und den erwarteten 13.5 Einheiten, die dem Gegner bei einem Treffer an Schaden zugefügt werden.
Um das Schiff zu einer möglichst schnellen Ausrichtung zu veranlassen (etwa 9 Linksdrehungen statt 15 Rechtsdrehungen), belegen wir auch hier die Rotationen abweichend vom Energiemodell mit Kosten von 0.5.
Da sich die Raumschiffe nicht bewegen, treten durch abwechselnde Rechts- und Linksdrehungen Zyklen im MDP auf. Ein Diskontierungsfaktor ist daher unverzichtbar. Wir wählen hier einen Wert von 0.9.

Der Zustandsraum umfasst zunächst den Abstand d zwischen den beiden Schiffen und die Ausrichtung α des eigenen Schiffes relativ zum Gegner.
In Kapitel 2.4.2 wurde bereits angesprochen, dass sich die Seitenverhältnisse der Darstellung auf dem Bildschirm und der systeminternen Koordinaten unterscheiden. Konkret bedeutet das für unseren Fall, dass der Schiffsradius und damit der Winkelbereich, in dem ein Torpedo treffen würde, in y-Richtung größer ist als in x-Richtung (siehe Abbildung 3.10). Um dem lernenden Agenten die Möglichkeit zu geben, diese Besonderheit autonom

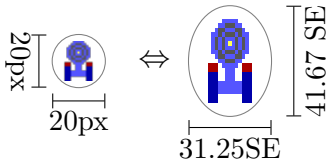

Abbildung 3.10: *Pixel versus Systemeinheiten*

zu erlernen, wird eine weitere Zustandsdimension eingeführt, die angibt, ob das eigene Schiff absolut eher in x- oder in y-Richtung ausgerichtet ist.

Es wäre denkbar, ähnlich wie beim Ausweichmanöver für die Aktionen eine Eingabedimension des neuronalen Netzes bereitzustellen, mit den Werten -3, 0 und $+3$ für Linksrotation, Torpedo abfeuern und Rechtsrotation. Von der Q-Funktion bezüglich der Aktion Torpedo abfeuern würden wir jedoch erwarten, dass sie nahezu ausschließlich die Werte -11.5 oder $+2$ annimmt und sich zwischen diesen beiden Werten scharfe „Kanten“ bilden, während die Funktion der Rotationen, aufgetragen auf die Ausrichtung zum Gegner, eher V-förmig ausfallen wird. Wir ziehen es daher vor, für jede Aktion ein eigenes neuronales Netz zu verwenden. Die Anpassungen an den Lernalgorithmus sind minimal: Es muss lediglich anhand der gerade betrachteten Aktion das zugehörige Netz ausgewählt werden, alle drei Netze werden gleichzeitig trainiert und aktualisiert.
Jedes der neuronalen Netze besteht aus drei Eingabedimensionen, sieben Neuronen in der verdeckten Schicht und einem Ausgabeneuron.

Ergebnisse

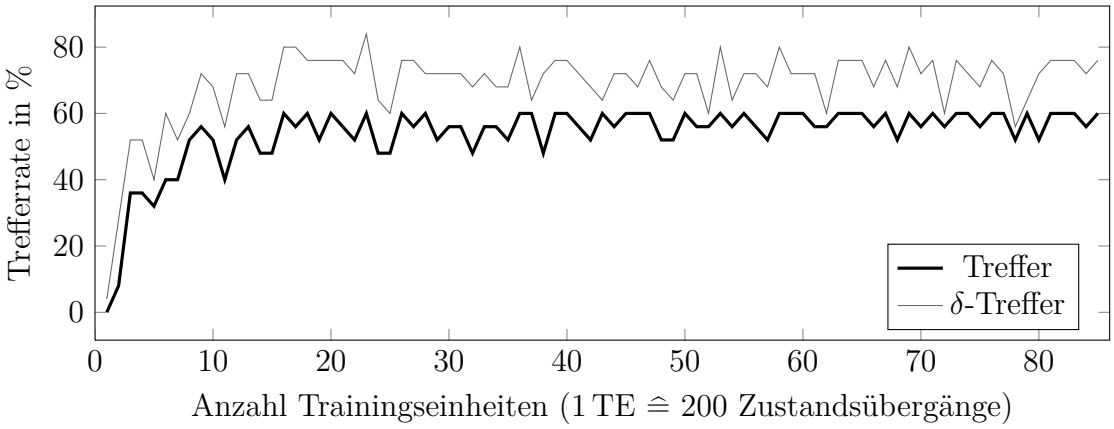

Abbildung 3.11: *Lernverlauf für stationäre Raumschiffe.*

Der Lernverlauf in Abbildung 3.11 zeigt, dass in bestenfalls 60% der Startsituationen ein Treffer erzielt wird. Bei genauerem Betrachten lässt sich jedoch feststellen, dass der Gegner in vielen Situationen gar nicht getroffen werden kann. Dies hängt mit der Rasterung der Rotationen in 15°-Schritte zusammen. Für das Raumschiff ergeben sich dadurch nur $\frac{360°}{15°} = 24$ mögliche absolute Ausrichtungen. Mit zunehmendem Abstand der beiden Schiffe häufen sich die Situationen, in denen sich der Gegner in einem toten Winkel zwischen zwei solchen Raster-Schritten befindet (siehe Abbildung 3.12).

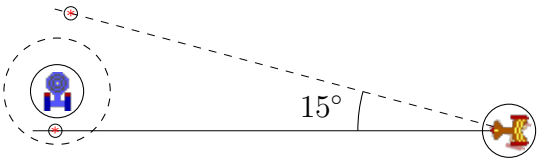

Abbildung 3.12: *Problem der 15°-Rasterung*

Dennoch lernt der Agent wie erwartet, sich in Richtung des Gegners zu orientieren. Er schießt dann entweder knapp daran vorbei oder verharrt mit Hin- und Herbewegungen in dieser Position, bis er durch das Timeout zum Abschuss gezwungen wird. Doch auch Torpedos, die nicht direkt, sondern nur fast treffen, können von spielerischem Wert sein. Zum Beispiel

können sie den Gegner dazu veranlassen, ein Ausweichmanöver einzuleiten, das ihn mehr Energieeinheiten kostet als das Abfeuern des Torpedos. Dies trifft insbesondere auf menschliche Gegner zu, die nicht in Sekundenschnelle exakt beurteilen können, ob ein Torpedo das eigene Schiff treffen würde. Auch im Zusammenspiel mit weiteren abgefeuerten Torpedos kann ein nur fast treffender Torpedo den Gegner unausweichbar bedrängen. Selbst wenn nur einer von fünf Torpedos trifft, wird dem Gegner mehr Schaden zugefügt als dem eigenen Schiff an Energiekosten entstanden ist.
Um auch diese knappen Treffer in der Statistik berücksichtigen zu können, führen wir sogenannte δ**-Treffer** ein. Darunter fallen alle Torpedos, die innerhalb des doppelten Schiffsradius am Gegner vorbeifliegen, wie beispielsweise der untere Torpedo in Abbildung 3.12.[4]

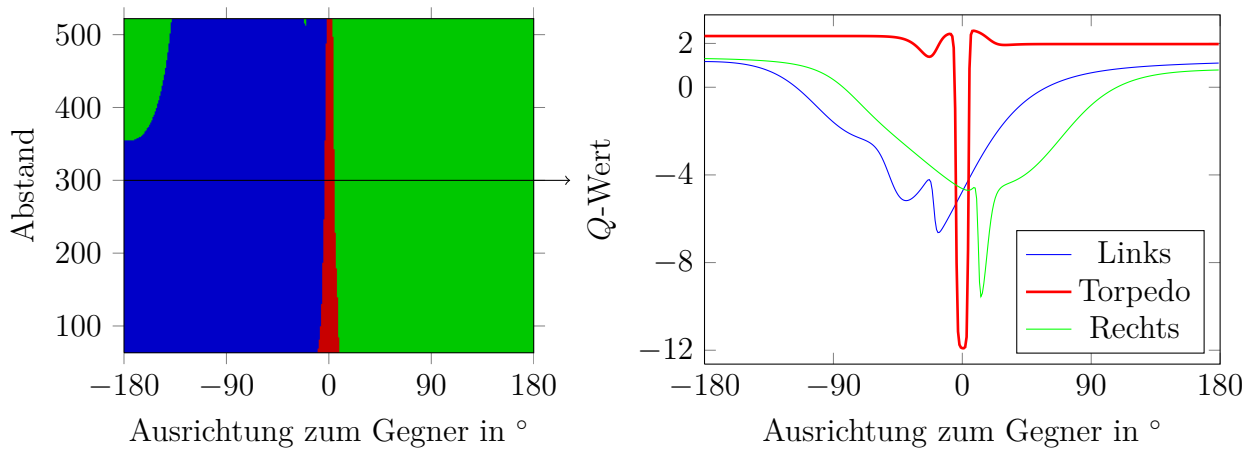

Abbildung 3.13: *Die gelernte Wertfunktion nach Trainingseinheit 82. Im linken Diagramm ist die Farbe für die Aktion mit jeweils minimalem Q-Wert über der Ausrichtung und dem Abstand zum Gegner aufgetragen. Das rechte Diagramm zeigt einen Ausschnitt der Wertfunktionen für einen mittleren Abstand von 300 Einheiten. Der Parameter für die Absolutausrichtung wurde in beiden Fällen auf einen Wert festgesetzt, der eine diagonale Ausrichtung repräsentiert.*

Auch in diesem Szenario bietet es sich an, die gelernte Wertfunktion zu plotten. In Abbildung 3.13 (links) ist zu erkennen, dass der Winkelbereich, in dem die Torpedo-Aktion als beste Aktion gilt, mit zunehmendem Abstand zum Schiff kleiner wird. Auch die Dimension des Zustandsraums für die Absolutausrichtung (siehe Anfang von Kapitel 3.3.1) wirkt sich auf die Breite des Winkelbereichs aus, wenn auch nur geringfügig (hier nicht dargestellt).
Außerhalb dieses Winkelbereichs ist jeweils die Aktion die beste, die das Schiff näher in Richtung Gegner ausrichtet. Es gibt jedoch einen kleinen Teil des Zustandsraums für Abstände größer als 350 Einheiten, in dem die Q-Funktion nicht zur optimalen Aktion führt (das wäre in diesen Fällen die Linksrotation). Möglicherweise waren hierfür nicht ausreichend Trainingssituationen vorhanden oder die zugehörigen Zielwerte wurden von dem Netz schlecht approximiert. Wie das rechte Diagramm zeigt, liegen für (vom Absolutbetrag her) große Winkel die Kurven für Links- und Rechtsrotation ohnehin nah beieinander, sodass sie sich leicht überschneiden können. Dieser kleine Fehler hätte jedoch lediglich zur Auswirkung, dass bis zu 3 Rotationen mehr als nötig durchgeführt würden.

[4] Natürlich gibt es in diesem stationären Szenario auch Fälle, in denen kein δ-Treffer möglich ist.

3.3.2 Einfaches Offensivmanöver

Aufbauend auf den Voruntersuchungen erweitern wir das Szenario, indem wir die Raumschiffe in konstante, geradlinige Bewegung versetzen. Der Zustandsraum erweitert sich damit um zwei Dimensionen für die relative Geschwindigkeit der Raumschiffe zueinander. Diese werden (in Analogie zur Variante (b) beim einfachen Ausweichmanöver) in einem schiffszentrischen Koordinatensystem angegeben, dessen x-Achse durch das gegnerische Schiff verläuft. Entsprechend verwenden wir drei neuronale Netze mit der Topologie „5-7-1“.

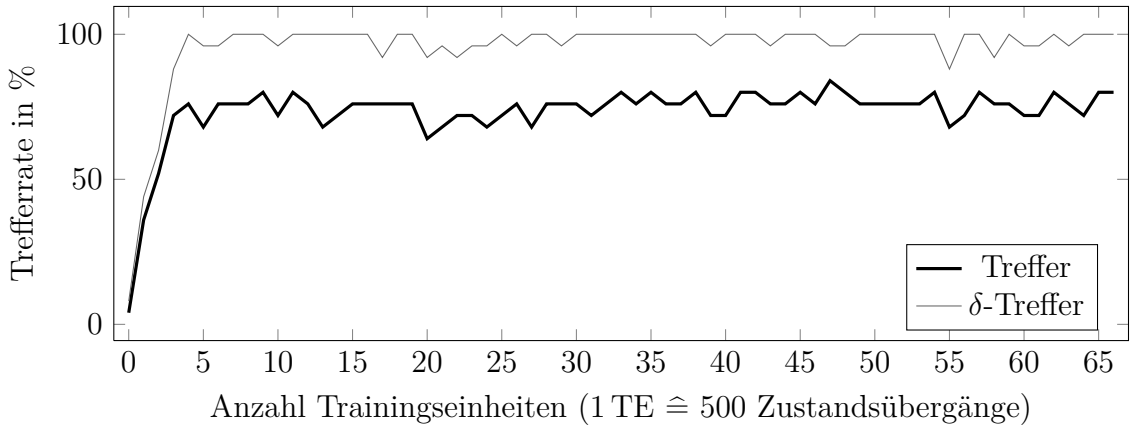

Abbildung 3.14: *Lernverlauf für konstant fliegende Raumschiffe*

Abbildung 3.14 zeigt den Lernverlauf dieses Trainings. Es wurde eine Trefferrate von bis zu 80% erzielt. Dass dieses Ergebnis besser ist als beim vermeintlich leichteren Szenario mit stationären Schiffen, hängt damit zusammen, dass der Gegner sich in diesem Fall wieder aus einem toten Winkel herausbewegen kann und das eigene Schiff die Möglichkeit hat, diesen Augenblick abzuwarten. Somit ist es fast von jedem Startzustand aus möglich, den Gegner zu treffen oder zumindest sehr knapp vorbei zu schießen. Nach der letzten Trainingseinheit konnte beobachtet werden, dass alle Torpedos selbständig innerhalb des Zeitlimits abgeschossen wurden und selbst die δ-Treffer den Gegner nur um weniger als 3 Pixel verfehlten.

3.3.3 Komplexes Offensivmanöver

Nach den guten Ergebnissen gegen einen konstant fliegenden, untätigen Gegner wollen wir nun überprüfen, wie es sich mit Gegnern verhält, die eine bestimmte Strategie verfolgen. Wir nutzen hier erstmals die Möglichkeit des SSLF, auch den Gegner zu steuern. Zunächst werden wir einen Gegner wählen, der das handkodierte Verhalten der Spiel-KI imitiert. In einem zweiten Experiment werden wir den Gegner mit einem der zuvor gelernten Ausweichmanöver ausstatten.

Lernen gegen Spiel-KI

Wir verwenden hier die im SSLF nachimplementierte Spiel-KI. Da uns für dieses Szenario in erster Linie das Bewegungsprofil des Gegners interessiert, blenden wir die Offensivaktionen (Phaser und Torpedo) des gegnerischen Schiffes aus, um das Lernverhalten nicht zu stören.

Auch den unvorhersehbaren Warpsprung wollen wir hierfür ignorieren. Diese drei Aktionen werden entsprechend durch „keine Aktion“ ersetzt.
Die Startsituationen werden nach den Kriterien für einen konstant fliegenden Gegner gesetzt. Allerdings wird nach dem Abschuss noch bis zu 10 Zeitschritte (bzw. bis zum Treffer) abgewartet, in denen der Gegner sich weiter nach seiner Strategie verhält. Für den lernenden Agenten endet eine Sequenz weiterhin mit dem Abschuss, lediglich die Bestimmung des Bewertungssignals (Treffer oder nicht) wird verzögert.
Als weitere Zustandsdimension kommt die Ausrichtung des Gegners bezüglich des eigenen Schiffs hinzu, sodass der Agent die Chance hat, Änderungen der Flugbahn des Gegners vorherzusehen. Im Gegenzug verzichten wir auf die Dimension für die Absolutausrichtung, da diese nur eine geringe Auswirkung auf die Wertfunktion zeigt. Dennoch erhöhen wir die Anzahl verdeckter Neuronen auf 9.

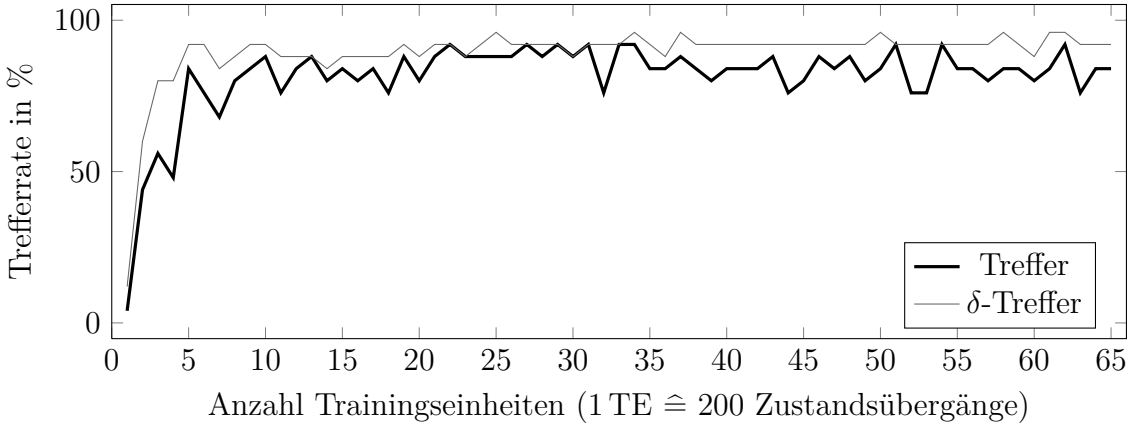

Abbildung 3.15: *Lernverlauf mit modifizierter Spiel-KI als Gegner*

Die erstaunlich guten Trefferraten (siehe Abbildung 3.15) überraschen nicht, wenn man das Bewegungsverhalten des Gegners genauer analysiert: Die Strategie des Gegners setzt sich aus handkodierten Wahrscheinlichkeitsverteilungen der verfügbaren Aktionen zusammen, die im Wesentlichen über den Abstand und die Ausrichtung zum Gegner definiert sind. Der Kern der Strategie besteht darin, sich zum Gegner hin auszurichten und in dessen Richtung Impulse auszuführen. Mehr oder weniger zufällig werden dabei auch Torpedos abgefeuert und bei hinreichend kleinem Abstand und passender Ausrichtung Phaser eingesetzt, gelegentlich auch Warpsprünge. Für unser Szenario bedeutet dies, dass sich der Computergegner – nicht auf kürzestem Weg aber dennoch stetig – auf den lernenden Agenten zubewegt und meistens zu ihm hin ausgerichtet ist. Dadurch wird es unserem lernenden Agenten sogar erleichtert, den Gegner zu treffen. Offensichtlich gelingt es ihm auch, diesen Vorteil zu nutzen und sich auf die stationäre Strategie des Gegners einzustellen.

Lernen gegen ausweichenden Gegner

Ganz anders sieht es jedoch gegen einen ausweichenden Gegner aus. In diesem Versuch wurde das gegen einen Torpedo gelernte Ausweichmanöver aus Kapitel 3.2.1 eingesetzt. Dieses wird aktiv, sobald der Gegner unmittelbar von dem abgefeuerten Torpedo bedroht wird, andernfalls verhält er sich passiv, wie im Szenario mit konstant fliegenden Raumschiffen.
Der Lernverlauf dieses Szenarios ist in Abbildung 3.16 dargestellt. Nur bei kleinen Abstän-

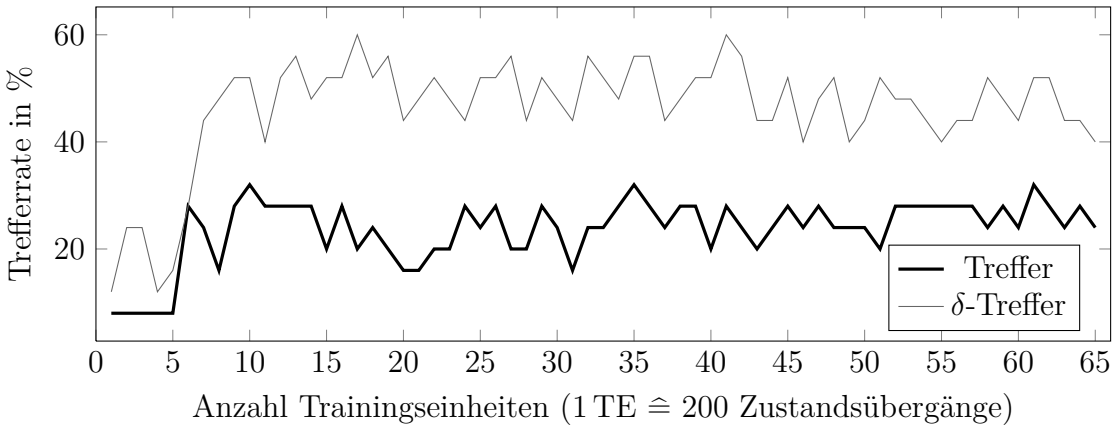

Abbildung 3.16: *Lernverlauf mit ausweichendem Gegner*

den ist es dem lernenden Agenten möglich, den Gegner zu treffen. Bei größeren Abständen gelingt es dem Gegner nahezu immer, dem Torpedo auszuweichen, da dessen Verhalten genau für diese Art von Szenario trainiert wurde. Entsprechend erhält der Agent während des Lernens viel negative aber nur wenig positive Rückmeldung.
Dieser Versuch demonstriert zwar an einem weiteren Beispiel die Leistungsfähigkeit des gelernten Ausweichmanövers, offenbart aber gleichzeitig, dass eine dem lernenden Agenten gestellte Aufgabe natürlich auch lösbar sein muss, um gute Ergebnisse erwarten zu können.

3.3.4 Erweitertes Offensivmanöver

In Anlehnung an das komplexe Ausweichmanöver mit ausweichendem Gegner wollen wir hier eine Erweiterung vornehmen, bei der der lernende Agent zwei Torpedos abfeuert. Der Abschuss des zweiten Torpedos kann unmittelbar auf den ersten folgen oder um eine Rotation nach rechts oder links versetzt sein.
Der Fokus soll hierbei jedoch nicht allein auf der Trefferrate sondern allgemein auf den dem Gegner verursachten Kosten liegen, denn auch das Ausweichmanöver benötigt Energie für die Impulse, im Idealfall sogar mehr als das Abfeuern des Torpedos.
Entsprechend soll der lernende Agent nicht nur eine Belohnung erhalten, wenn mindestens ein Torpedo trifft ($C^+ = -13.5$), sondern auch für Impulsaktionen des Gegners. Um auch dieses positive Signal zu verstärken entspricht die Belohnung nicht der tatsächlichen Anzahl von Impulsen, sondern dem konstanten Wert $\tilde{C}^+ = 10$, sofern der Gegner zum Ausweichen (mit Impulsen) veranlasst wurde. Bei der Evaluation werden dagegen nur die ausgeführten Impulse gezählt.
Der Zustandsraum wird um eine Dimension erweitert, die angibt, ob der erste Torpedo bereits abgeschossen wurde (Wert $+3$) oder nicht (Wert -3). Die Terminalkosten gelten direkt für die unmittelbar auf den ersten Abschuss folgende Aktion (Torpedo oder Rotation), da diese bereits darüber entscheidet, in welche Richtung der zweite Torpedo abgefeuert wird.

Abbildung 3.17 zeigt den Lernverlauf für dieses Szenario. Zwar hält sich die Trefferrate auch bei zwei abgefeuerten Torpedos in Grenzen, dafür lässt sich aber eine positive Kostenbilanz ziehen: Im Durchschnitt liegen die dem Gegner – durch vereinzelte Treffer und durch Ausweichmanöver – entstandenen Kosten von etwa sechs Energieeinheiten pro Sequenz über den jeweils nur vier benötigten Einheiten zum Abfeuern der beiden Torpedos.

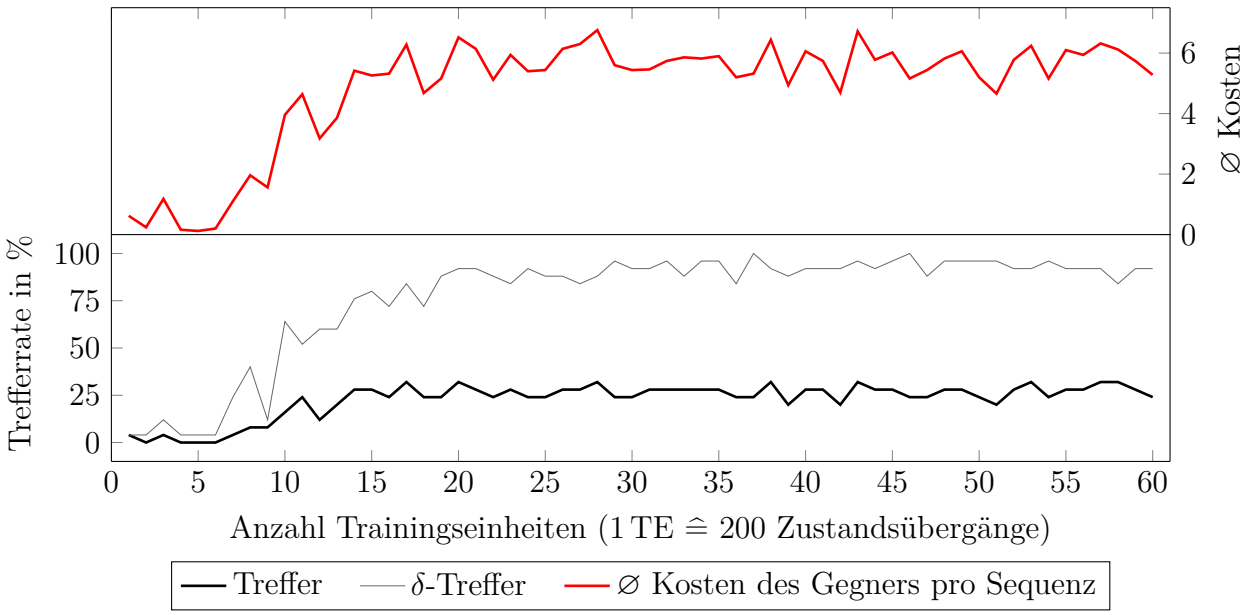

Abbildung 3.17: *Lernverlauf für zwei Torpedos gegen ausweichenden Gegner. Die hier aufgetragenen Kosten berechnen sich wie folgt: Für eine erfolgreiche Sequenz (mindestens ein Treffer) werden pauschal 13.5 Einheiten angerechnet, hinzu kommt eine Energieeinheit für jeden vom Gegner ausgeführten Impuls. Anschließend wird der Durchschnitt über alle 25 Evaluationssequenzen gebildet.*

3.4 Einbettung in die Spiel-KI

Nachdem wir einige Teilverhalten gelernt haben, geht es im Folgenden darum, diese zu einem Gesamtverhalten zusammenzufügen bzw. in ein bereits vorhandenes handkodiertes Verhalten zu integrieren. Wir greifen dabei auf das spielinterne Verhalten (die Spiel-KI) zurück, welche im SSLF nachimplementiert wurde und daher von uns angepasst werden kann. Wie bereits festgestellt wurde, ist dort keinerlei Ausweichverhalten vorhanden und Torpedos werden nahezu willkürlich abgefeuert. Dies wollen wir durch unsere zuvor gelernten Verhalten ändern. Das Nahkampfverhalten der Spiel-KI mit dem Einsatz von Phasern und Torpedos auf kurzer Distanz ist dagegen sehr leistungsfähig und wird daher von uns weitestgehend beibehalten.

3.4.1 Integration des Ausweichmanövers

Nach den Ergebnissen in Kapitel 3.2.2 bietet es sich am ehesten an, das hybride Verhalten einzusetzen, welches immer das neuronale Netz für die gerade relevante Anzahl Torpedos verwendet (Variante 6). Damit können bis zu drei Torpedos berücksichtigt werden, die nach momentaner Geschwindigkeit das Schiff treffen würden.
Das Ausweichmanöver wird aktiv, sobald das eigene Schiff Gefahr läuft, mit mindestens einem Torpedo zu kollidieren. Zeigt das jeweilige neuronale Netz für den aktuellen Zustand einen minimalen Q-Wert größer als 10 an, so wird ein Warpsprung ausgeführt.

Bei allen Trainingsvorgängen wurde bisher darauf geachtet, dass die betrachteten Objekte innerhalb einer Sequenz nicht über den Spielfeldrand hinaus fliegen, um das Lernen nicht durch sprunghafte Zustandsübergänge zu stören. Innerhalb eines richtigen Spiels passiert dies jedoch sehr häufig. So kann ein sich am Rand befindendes Raumschiff schon im nächsten Zeitschritt von einem vermeintlich weit entfernten, gegenüberliegenden Torpedo getroffen werden. Die Torusförmigkeit des Universums sollte daher bei der Modellierung des Zustandsraums insbesondere beim Ausweichmanöver berücksichtigt werden. Hierzu betrachten wir jeweils alle neun Fälle, in denen eine fiktive Kopie des Spielfelds am Spielfeldrand anliegen kann (siehe Abbildung 3.18). Letztendlich wird für die Beziehung zwischen zwei Objekten (dem eigenen Schiff und dem Torpedo) der Fall angenommen, bei dem der Abstand am kleinsten ist.[5]

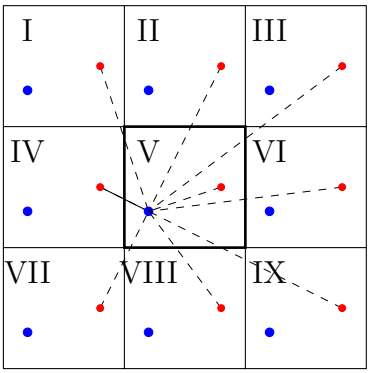

Abbildung 3.18: *Berücksichtigung der Torusförmigkeit. In diesem Beispiel wird Fall IV angenommen.*

Auswertung

Um die Verbesserung der Spiel-KI durch das integrierte Ausweichmanöver empirisch nachzuweisen, wurden die angepasste und die originale Spiel-KI in einer Reihe von ganzen Spielen gegeneinander getestet. Diese sowie die noch folgenden Ergebnisse sind am Ende des Kapitels in Tabelle 3.1 zu finden.
Obwohl das Ausweichmanöver nur in 9.4% der Zustände zum Einsatz kommt, wurde die Erfolgsrate der Spiel-KI gegen sich selbst im Verhältnis 2.4:1 verbessert[6].

3.4.2 Integration des Offensivmanövers

In ähnlicher Weise statten wir die Spiel-KI nun mit einem gelernten Offensivmanöver aus. Die folgenden beiden Varianten kommen dabei für uns in Frage:

1. Das gegen einen konstant fliegenden Gegner gelernte Offensivmanöver verhält sich unabhängig von der Strategie des Gegners. Es trifft in vielen Fällen, sofern der Gegner keine Impulse ausführt oder veranlasst ihn zum Ausweichen. Analog zum Ausweichmanöver kann dabei die Torusförmigkeit berücksichtigt werden, um den Zustandsraum jeweils über den Fall zu definieren, bei dem die beiden Schiffe den kürzesten Abstand haben. Torpedos würden somit auf möglichst kurzer (und erfolgversprechender) Distanz in den Lauf des Gegners abgefeuert. Wir unterscheiden für unsere Betrachtungen daher zwischen zwei Varianten:
   (a) Zustandsberechnung innerhalb des Spielfelds

[5] Da Objekte stets vollständig auf dem Bildschirm dargestellt werden, muss vom eigentlichen Spielfeldrand die Zone abgeschnitten werden, die vom passierenden Objekt übersprungen wird. In der Regel sind Torpedos schneller als Schiffe, sodass wir pauschal von allen Rändern jeweils den Radius eines Torpedos subtrahieren.
[6] gegenüber der Referenz 1:1 wenn die Spiel-KI gegen sich selbst spielt

   (b) Zustandsberechnung über die Spielfeldgrenzen hinweg unter Berücksichtigung der Torusförmigkeit

2. Da wir für unsere Testzwecke die Spiel-KI als Gegner verwenden, bietet es sich an, das gegen dessen Bewegungsprofil gelernte Offensivmanöver einzusetzen. Hier sind bessere Ergebnisse zu erwarten, wenn die Torusförmigkeit bei der Zustandsmodellierung ignoriert wird, da auch die Spiel-KI intern den Abstand und die Ausrichtung ausschließlich innerhalb des Spielfelds berechnet und die Strategie davon abhängig ist. Zum besseren Vergleich betrachten wir dennoch beide Varianten (a) und (b).

In beiden Fällen wird das Offensivmanöver aktiv, wenn die Maximalzahl 5 der eigenen abgefeuerten Torpedos nicht erschöpft ist und die Schiffe mindestens in einem Abstand etwas größer als die maximale Phaserlänge zueinander stehen. Bei kleineren Abständen wird auf die original Spiel-KI zurückgegriffen, die dann mit hoher Wahrscheinlichkeit Phaser verwendet. Auch für Abstände größer als 600 Einheiten wird auf das Verhalten der Spiel-KI umgeschalten, welches sich tendenziell auf den Gegner zubewegt, da die Offensivmanöver für diese Abstände nicht trainiert sind und die Wahrscheinlichkeit für einen Treffer gering ist.
Um Deadlock-Situationen zu vermeiden, in denen beide Schiffe für längere Zeit stehen und erfolglos versuchen, sich gegenseitig anzuvisieren, wird nach ununterbrochener Ausführung des Offensivmanövers über 25 Zeitschritte hinweg ein Impuls erzwungen.

Auswertung

Auch durch gelernten Offensivmanöver konnte die Erfolgsrate der Spiel-KI von 50% auf 76% bzw. 78% verbessert werden. Für beide Offensivmanöver ist das Ergebnis mit Variante (b) zur Zustandsberechnung etwas schlechter ausgefallen als mit Variante (a). Das könnte damit zusammenhängen, dass die im Nahkampf eingesetzte Spiel-KI das Schiff immer innerhalb des Spielfelds zum Gegner ausrichtet, da die Phaser nicht über den Rand hinweg feuern können. Der häufige Wechsel zwischen gelerntem Offensivmanöver und Nahkampfverhalten verursacht daher oft zusätzliche Rotationen. Dagegen tritt nur selten der erhoffte Effekt ein, dass auf kurzer Distanz über den Rand hinweg abgefeuerte Torpedos den Gegner treffen.

3.4.3 Integration des Ausweich- und des Offensivmanövers

Abschließend wollen wir beide gelernten Teilverhalten gemeinsam in die Spiel-KI einbinden. Wie bereits in Kapitel 3.4.2 wird das Ausweichmanöver aktiv, wenn das Schiff von mindestens einem Torpedo bedroht wird (dabei werden ebenfalls Torpedos an den gegenüberliegenden Rändern berücksichtigt). Andernfalls wird das jeweilige Offensivmanöver für Abstände zwischen 300 und 600 Einheiten aktiv und im Nahkampf das Verhalten der Spiel-KI. Für die beiden Offensivmanöver beschränken wir uns jedoch aufgrund der vorherigen Beobachtungen auf die Variante (a).

Durch die gemeinsame Einbettung beider gelernten Teilverhalten konnte das Gewinnverhältnis noch einmal deutlich gesteigert werden auf 9:1 bzw. 10:1 mit dem gegen die Spiel-KI gelernten Offensivmanöver, wie die folgende Tabelle 3.1 zeigt.

Variante	Spiel-KI gegen Spiel-KI	mit Ausweichmanöver	mit Offensivmanöver gelernt: 1. gegen konstant fliegenden Gegner / 2. gegen Spiel-KI				mit Ausweich- und Offensivmanöver	
	Referenz	1	2.1(a)	2.1(b)	2.2(a)	2.2(b)	3.1	3.2
gewonnen	129	176	192	150	193	167	253	249
verloren	128	73	59	85	54	74	28	24
unentschieden (Kollision)	43	51	49	65	53	59	19	27
Gewinnverhältnis	1.01:1 50.2%	2.41:1 70.7%	3.25:1 76.5%	1.76:1 63.8%	3.57:1 78.1%	2.26:1 69.3%	9.04:1 90.0%	10.38:1 91.2%
Ausweichen gegen 1/2/3 Torpedos	-	7.3%/ 1.3%/ 0.7%	-	-	-	-	6.4%/ 1.3%/ 0.7%	6.2%/ 1.4%/ 0.7%
Ausweichen gesamt	-	9.4%	-	-	-	-	8.4%	8.4%
Offensivmanöver	-	-	22.6%	20.4%	23.7%	30.5%	21.8%	23.3%
∅ Spieldauer (Zeitschritte)	164	197	131	131	119	131	179	179

Tabelle 3.1: *Ergebnisse nach je 300 Spielen gegen die originale Spiel-KI – jeweils aus Sicht des angepassten Verhaltens – und Statistiken, zu welchen Anteilen die jeweiligen Teilverhalten zum Einsatz kommen. Die Nummerierung der Varianten entspricht den Unterkapiteln bzw. Aufzählungen von Kapitel 3.4.*

4 Zusammenfassung und Ausblick

4.1 Zusammenfassung

In dieser Arbeit haben wir uns das Ziel gesetzt, die bereits vorhandene, handkodierte Spiel-KI des Weltraum-Spiels Star Ships durch die Anwendung von Reinforcement-Lernverfahren zu verbessern. Um dies zu realisieren, haben wir zwei besonders spielrelevante Teilverhalten – das Ausweichmanöver und das Abfeuern von Torpedos – identifiziert und diese jeweils in unterschiedlich komplexen Ausführungen als Lernaufgaben (MDPs) formuliert.
Mithilfe des NFQ-Verfahrens und neuronalen Netzen als Funktionsapproximatoren konnten diese Teilverhalten erfolgreich erlernt werden.
So ist es gelungen, ein hybrides Ausweichmanöver basierend auf drei einzelnen gelernten Teilverhalten zu entwickeln, welches zuverlässig bis zu drei Torpedos gleichzeitig ausweichen kann oder frühzeitig signalisiert, wenn den Torpedos nicht ausgewichen werden kann, sodass alternative Maßnahmen (zum Beispiel ein Warpsprung) ergriffen werden können. Zudem setzt dieses Verhalten – im Vergleich zu einer handkodierten Variante – die zur Verfügung stehenden Aktionen (Rotationen und Impulse) energieeffizient ein. Allein durch dieses Ausweichmanöver konnte die Leistung der vorhandenen Spiel-KI mehr als verdoppelt werden.
Ferner wurde gelernt, einen Torpedo in den Lauf eines konstant fliegenden Raumschiffes abzufeuern und damit den Gegner in bis zu 80% der Fälle zu treffen. In einem weiteren Experiment wurde gezeigt, dass der lernende Agent in der Lage ist, das stochastische Bewegungsverhalten der Spiel-KI zu erfassen und zu seinem Vorteil zu nutzen. Selbst gegen einen intelligenten, ausweichenden Gegner konnte ein Lernerfolg erzielt werden: mithilfe von zwei versetzt abgefeuerten Torpedos wird der Gegner zu einem Ausweichmanöver gezwungen, das ihn mehr Energieeinheiten kostet als das Abfeuern der Torpedos.
Die Einbettung von Offensiv- und Ausweichmanöver in die Spiel-KI brachte letztendlich ein Gesamtverhalten hervor, das die ursprüngliche Version der Spiel-KI mit einem Gewinnverhältnis von 10:1 schlagen kann. Das Anfangs gesetzte Ziel wurde damit also definitiv erreicht.

4.2 Ausblick

Sowohl die Klassen der gelernten Teilverhalten als auch die zusammengesetzten Komplettverhalten aus Kapitel 3.4 werden innerhalb des SSLF zur Verfügung gestellt. Sie können somit als Benchmark zum Vergleich oder als Lerngegner in anderen zukünftigen Arbeiten verwendet werden.

Wie in Kapitel 3.2.1 geschildert wurde, muss bereits für eine einfache Lernaufgabe eine Vielzahl von Faktoren und Parametern festgelegt werden. Bei einigen davon handelt es sich um Empfehlungen, Erfahrungswerte oder manuell optimierte Einstellungen. Für jede Kombination von Parametern empirische Vergleiche anzustellen, würde den Rahmen dieser Arbeit

sprengen. Für den einen oder anderen Faktor könnte es sich aber lohnen, verschiedene Einstellungen zu testen, zum Beispiel verschiedene Netztopologien mit unterschiedlicher Anzahl verdeckter Neuronen oder mehreren verdeckten Schichten.

In Kapitel 3.1 wurde eine dynamische Skalierungstechnik für Zielwerte vorgestellt, die wir während der gesamten Arbeit verwendet haben. Die Entwicklung des Zielintervalls durch die dynamische Skalierung wurde hier nicht genauer analysiert. Man könnte abervermuten, dass sich die Intervallgrenzen kontinuierlich von den minimalen bzw. maximalen Übergangskosten hin zu den minimalen bzw. maximalen Q-Werten hin entwickeln. Dies zu beobachten und die Auswirkungen mit einer statischen Skalierung zu vergleichen könnte Bestandteil zukünftiger Arbeiten sein.

Wir haben uns in dieser Arbeit im Wesentlichen auf zwei Low-Level-Verhalten beschränkt. Es lassen sich aber durchaus noch weitere Teilverhalten finden, die mithilfe von RL-Methoden gelernt oder optimiert werden können, beispielsweise verschiedene Nahkampfverhalten oder gar eine Art Anschleichmanöver, das die Torusförmigkeit ausnutzt, um sich von hinten dem Gegner zu nähern.
Für eine erfolgreiche Gesamtstrategie spielt jedoch auch entscheidende Rolle, wie diese Teilverhalten auf höherer Ebene kombiniert werden. Wir haben hier nur einen handkodierten Ansatz verfolgt, doch auch auf dieser Ebene ist der Einsatz verschiedener Lernmethoden denkbar. So könnte man versuchen, eine Wertfunktion für Gesamtzustände zu lernen, die bei der Entscheidung für oder gegen einen Warpsprung zu Rate gezogen werden kann. Damit könnten Fragen beantwortet werden wie „In welchen Situationen lohnt es sich, einen Torpedotreffer in Kauf zu nehmen, um meine momentane Position nicht aufzugeben?“.
Wenn man mehrere verschiedene Offensivverhalten zur Auswahl hat (sowohl gelernte als auch handkodierte), stellt sich auch die Frage, wann welches Teilverhalten am besten geeignet ist. Diese Fragestellung könnte mit evolutionären Algorithmen angegangen werden. Dazu müssten Merkmale aufgestellt werden, anhand derer sich Zustände kompakt kodieren lassen und diese mit den Metaaktionen (Teilverhalten) kombiniert werden. Die Fitnessfunktion könnte darin bestehen, eine Reihe von Testspielen durchzuführen, in denen die Strategien gegeneinander evaluiert werden. Durch Selektion, Rekombination und Mutation würden neue Strategien gefunden, die dann wiederum durch die Fitnessfunktion bewertet werden. Bei geeigneter Implementierung kann dieses Verfahren eine leistungsstarke Optimierungsmethode darstellen.

Alle bisherigen Ansätze zielen darauf ab, die Spiel-KI weiter zu verbessern und zu einem möglichst optimalen Verhalten auszubauen. Für akademische Zwecke mag dies ein erstrebenswertes Ziel sein. Kehrt man jedoch zum eigentlichen Ziel von Computerspielen zurück, menschliche Spieler möglichst gut und ausdauernd zu unterhalten, so wird man früher oder später feststellen, dass die Optimalität eines Computergegners nicht das ausschlaggebende Kriterium für den Erfolg eines Spiels ist. Um einen menschlichen Spieler längerfristig an ein Spiel zu fesseln, darf der Computergegner nicht zu schwach aber auch nicht zu stark sein. Er muss stets eine neue, aber auch zu bewältigende Herausforderung darstellen. Die Aufgabe der Spielentwickler ist es daher, das richtige Maß zwischen Langeweile und Frustration zu finden. Dabei ist zu beachten, dass Menschen in Bezug auf ein Spiel unterschiedlich stark sind und sich durch die Auseinandersetzung mit dem Spiel kontinuierlich verbessern.

Viele Spiele bieten daher an, die Schwierigkeit des Gegners selbst zu wählen, zum Beispiel „Leicht“, „Normal“ oder „Schwer“. Eine Alternative wäre, den Schwierigkeitsgrad dynamisch an die momentane Leistung des Spielers anzupassen. Dies kann auf manueller Basis realisiert werden, indem der Computergegner zum Beispiel durch zufällige Aktionen mit stufenlos regelbarer Wahrscheinlichkeit geschwächt wird. Interessanter aber natürlich auch riskanter wären Online-Lernansätze. Dabei würde die Spiel-KI versuchen, Erfahrungen und Beobachtungen aus bereits absolvierten Spielen zu nutzen, um ihr Verhalten – speziell auf einen menschlichen Gegner bezogen – zu verbessern. In unserem Fall könnte beispielsweise die Kenntnis der menschlichen Strategie helfen, Torpedos gezielter abzufeuern. Man könnte sogar in Erwägung ziehen, die Strategie des Spielers zu imitieren. Sollte dies gelingen, so wäre durchweg ein der Stärke des menschlichen Spielers angemessener Computergegner vorhanden, der sich kontinuierlich mit der Leistung des Spielers verbessert. Der Mensch müsste sich damit stets der Herausforderung stellen, eine neue Strategie zu finden, um seine eigene zuvor gespielte Strategie zu schlagen.
Für das jeweilige RL-Verfahren kommt erschwerend hinzu, dass es mit sehr wenigen Trainingsdaten auskommen muss, die allein aus den Spielen gegen den menschlichen Spieler gesammelt werden können.

Angesichts des zu erwartenden Aufwands für die Umsetzung eines Lernverhaltens bleibt abzuwarten, ob sich Reinforcement Learning in naher Zukunft in kommerziellen Computerspielen etablieren wird. In jedem Fall ist individuell abzuschätzen, ob sich der Aufwand für den Einsatz von maschinellen Lernverfahren gegenüber der händischen Optimierung lohnt, denn Tatsache ist: Auch autonom lernende Agenten müssen das autonome Lernen erst einmal beigebracht bekommen.

Literaturverzeichnis

[1] J. S. Albus. *A New Approach to Manipulator Control: the Cerebellar Model Articulation Controller (CMAC).* Journal of Dynamic Systems, Measurement, and Control, 97:220–227, 1975.

[2] C. Amato und G. Shani. *High-level Reinforcement Learning in Strategy Games.* In *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010).* 2010.

[3] D. P. Bertsekas und J. N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, 1996.

[4] D. S. Broomhead und D. Lowe. *Multivariable Functional Interpolation and Adaptive Networks.* Complex Systems 2, Seiten 321–355, 1988.

[5] D. Ernst, P. Geurts, L. Wehenkel und L. Littman. *Tree-based batch mode reinforcement learning.* Journal of Machine Learning Research, 6:503–556, 2005.

[6] T. Gabel. *Anleitung zum Action-Spiel Star Ships*, 1997.

[7] T. Gabel. *Star Ships Learning Framework Manual (SSLF V1.0)*, `http://sslf.sourceforge.net`, 1996–2010.

[8] G. J. Gordon. *Approximate Solutions to Markov Decision Processes.* Technischer Bericht, Carnegie Mellon University, 1999.

[9] E. Kok. *Adaptive reinforcement learning agents in RTS games.* Diplomarbeit, 2008, University Utrecht, The Netherlands.

[10] M. Littman, A. R. Cassandra und L. P. Kaelbling. *Learning policies for partially observable environments: Scaling up.* In *Proceedings of the Twelfth International Conference on Machine Learning*, Seiten 362–370. Morgan Kaufmann, 1995.

[11] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* Wiley-Interscience, 1994.

[12] M. Riedmiller. *Dokumentation zu n++*, 1997.

[13] M. Riedmiller. *Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method.* In *16th European Conference on Machine Learning*, Seiten 317–328. Springer, 2005.

[14] M. Riedmiller und H. Braun. *A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm.* In *IEEE International Conference on Neural Networks*, Seiten 586–591. 1993.

[15] M. Riedmiller, T. Gabel, R. Hafner und S. Lange. *Reinforcement learning for robot soccer.* Auton. Robots, 27(1):55–73, 2009.

[16] R. S. Sutton und A. G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning).* The MIT Press, 1998.

[17] G. Tesauro. *Temporal difference learning and TD-Gammon.* Commun. ACM, 38(3):58–68, 1995.

[18] C. Watkins und P. Dayan. *Technical Note: Q-Learning.* Machine Learning, 8(3-4):279–292, 1992.

[19] S. Wender. *Integrating Reinforcement Learning into Strategy Games.* Diplomarbeit, The University of Auckland, New Zealand, 2009.

[20] A. Wilson, A. Fern, S. Ray und P. Tadepalli. *Learning and Transferring Roles in Multi-Agent MDPs.* Association for the Advancement of Artificial Intelligence (AAAI), 2008.

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Freiburg, 13. September 2010