Untersuchung des Pfannkuchen-Sortierproblems mit Hilfe von reinforcement learning

Lennart Obermüller

Besondere Lernleistung
Johannes-Kepler-Gymnasium Chemnitz

schulischer Betreuer: Steffen Polster

externer Betreuer: Dr. habil. Julien Vitay
(Technische Universität Chemnitz, Fakultät für Informatik,
Professur Künstliche Intelligenz)

2019/2020

1 Motivation

Aufgrund meiner Interessen in den Bereichen Mathematik und Informatik, beschäftigte ich mich schon länger über die Schulinhalte hinaus mit diesen Wissenschaften. Deshalb stand für mich auf jeden Fall fest, in diesen Gebieten meine BeLL anzusiedeln. Den Anstoß für die genauere Themenfindung brachte mir mein Praktikum an der TU Chemnitz in der Fakultät für Informatik, Professur Künstliche Intelligenz, welches ich vom 29.1.2018 bis zum 9.2.2018 durchführte. Aufgrund von diesem Praktikum entstand die Idee, mich in Form meiner BeLL genauer mit dem Thema der künstlichen Intelligenz, beziehungsweise besonders mit dem maschinellen Lernen auseinanderzusetzen. Ich finde dieses Thema besonders interessant und es ist unter anderem Gegenstand der aktuellen Forschung im Bereich Informatik. Jedoch ist der Fachbereich des maschinellen Lernens zu weitläufig, um ihn komplett in einer BeLL mit befriedigender Genauigkeit zu bearbeiten. Deshalb befasste ich mich zuerst mit dem Wesen des maschinellen Lernens und arbeitete einige Grundarten heraus. Dabei sprach mich eine Unterart des maschinellen Lernens besonders an – das reinforcement learning. Es basiert auf dem Prinzip des trial-and-error-learnings. Umgangssprachlich heißt das, dass der Computer verschiedene Möglichkeiten, ein Problem zu lösen, durchprobiert. Zufällig findet er dabei eine gute Möglichkeit und merkt sich diese und verbessert so seine gesamte Strategie. Besonders faszinierend finde ich zum Ersten, dass der Computer somit selbstständig eine gute Lösung für ein ihm vorher unbekanntes Problem findet und zum Zweiten, dass auch die meisten Lernvorgänge bei Tieren (und somit natürlich auch bei Menschen), wie z.B. das Laufen lernen, nach diesem Prinzip ablaufen. So entstand die Idee ein Problem in einem Programm zu simulieren und ein Programm zu schreiben, welches dieses Problem mit Hilfe von reinforcement learning löst. Einige Zeit später stieß ich zufällig auf das Pfannkuchen-Sortierproblem. Ich erkannte, dass reinforcement learning eine sehr gut geeignete Lösungsmethode für dieses Problem ist. Aufgrund dessen entschied ich mich, in meiner BeLL dieses Problem zu bearbeiten. Zusätzlich ist dieses Problem relativ einfach in einem Programm simulierbar. Außerdem stellte die Tatsache, dass die genaue minimale Zuganzahl zum Lösen des Problems noch nicht genau bekannt ist, sondern dass bisher dafür nur Abschätzungen existieren, einen weiteren Anreiz dar, da möglicherweise ein Programm entstehen könnte, welches besser als die bisher optimale Lösung ist.

Im Rahmen dieser Arbeit möchte ich mich mit dem Thema reinforcement learning beschäftigen, um ein Programm zu entwickeln, welches das Pfannkuchen-Sortierproblem möglichst effizient lösen kann.

Inhaltsverzeichnis

1 Motivation **1**

2 Aufbau der Arbeit **5**

3 Pfannkuchen-Sortierproblem **5**

3.1 Problembeschreibung 5

3.2 Abschätzungen 5

I Reinforcement learning - tabellarische Methoden **5**

4 Eingrenzung des Themas **6**

4.1 Künstliche Intelligenz 6

4.2 Maschinelles Lernen 6

4.3 Unsupervised learning 6

4.4 Supervised learning 6

4.5 Reinforcement learning 7

5 Einleitung **7**

6 Markow-Entscheidungsprozess **8**

7 Policy **10**

8 Value functions **11**

9 Bellman equations **11**

10 Dynamic programming **12**

11 Monte-Carlo Methoden **13**

12 Temporal-Difference learning **16**

13 Eligibility traces **17**

14 Reward scheduling **20**

II Reinforcement learning mit künstlichen neuronalen Netzen **20**

15 Funktionsabschätzung **20**

15.1 value-basierte Funktionsabschätzung 21

15.2 policy-basierte Funktionsabschätzung 21

16 Deep reinforcement learning **22**

17 Deep Q-learning **24**
17.1 Experience replay memory 24
17.2 Target network 26

III Anwendung auf das Pfannkuchen-Sortierproblem **26**

18 Environment **27**

19 Grundprogramm **28**

20 Agent **30**
20.1 Q-learning 31
20.2 SARSA 31
20.3 ϵ-greedy Algorithmus 31
20.4 Softmax Algorithmus 33

21 Eligibility traces **34**
21.1 Vergleich dictionaries - numpy arrays 34

22 Reward scheduling **35**

23 Variables ϵ **35**

24 Deep Q-learning **36**

IV Auswertung **41**

25 Vier Grundmethoden **41**
25.1 Parameter 41
25.2 Vergleich 41

26 Eligibility traces **45**

27 Reward scheduling **47**

28 Variables ϵ **47**

29 Deep Q-learning **50**

30 Zusammenfassung **50**
30.1 Fazit: optimale policy? 51
30.2 Ausblick: Pfannkuchen bei Mutationen 52

31 Literaturverzeichnis **54**

32 Ehrenwörtliche Erklärung **54**

2 Aufbau der Arbeit

Zuerst wird das Pfannkuchen-Sortierproblem erklärt und Ergebnisse der aktuellsten Arbeiten zu diesem Thema vorgestellt. Im zweiten Teil der Arbeit werden zuerst die Grundlagen von reinforcement learning erläutert und anschließend wird genauer auf die theoretischen Hintergründe zu den Methoden, welche zur Problemlösung benutzt wurden, eingegangen. Dabei wird in tabellarische Methoden und in Methoden mit künstlichen neuronalen Netzen unterteilt. Im dritten Teil der Arbeit werden besagte Methoden beim Pfannkuchen-Sortierproblem angewendet, und diese in einem Programm umgesetzt. Dies geschieht in der Programmiersprache Python. Zum Schluss werden verschiedene Methoden ausgewertet und verglichen.

3 Pfannkuchen-Sortierproblem

Das Problem wurde erstmals 1975 von Jacob E. Goodman unter dem Pseudonym „Harry Dweighter“ in der Zeitschrift American Mathematical Monthly veröffentlicht. Es beschreibt eine Sortiermethode, welche als „sorting by prefix reversal“ bekannt wurde.

3.1 Problembeschreibung

Gegeben sei ein Stapel von n Pfannkuchen mit $n \in \mathbb{N}$. Alle Pfannkuchen sind paarweise unterschiedlich groß, ihnen können also je nach Größe die Zahlen 1 bis n zugeordnet werden. Dabei bekommt der kleinste Pfannkuchen die 1 und der größte Pfannkuchen n. Der Stapel liegt in einer zufälligen Permutation vor. Nun sollen die Pfannkuchen nach Größe sortiert werden, sodass der größte Pfannkuchen ganz unten liegt. Ein Zug besteht darin, einen Teilstapel vom gesamten Stapel oben abzuheben, die Reihenfolge der Pfannkuchen in dem abgehobenen Teilstapel zu invertieren und ihn dann wieder dem verbliebenen Teilstapel oben hinzuzufügen.

3.2 Abschätzungen

Die Pfannkuchenzahlen P_n geben an, wie viele Züge man bei n Pfannkuchen maximal (d.h. im worst-case) benötigt. Die bisher beste Abschätzung ist $\frac{15}{14}n < P_n < \frac{18}{11}n$ (siehe [4]). Das Ziel ist ein, Programm zu entwickeln, welches möglichst schnell oft Ergebnisse in diesem Bereich liefert.

Teil I

Reinforcement learning - tabellarische Methoden

4 Eingrenzung des Themas

4.1 Künstliche Intelligenz

Die Künstliche Intelligenz ist ein Teilgebiet der Informatik, wobei menschliche Vorgehensweisen der Problemlösung und kognitive Fähigkeiten auf Computer angewendet werden sollen, um so Probleme zu lösen, welche Intelligenzleistungen voraussetzen. Ein großes Teilgebiet der künstlichen Intelligenz ist das maschinelle Lernen.

4.2 Maschinelles Lernen

Das maschinelle Lernen ist ein Teilgebiet der künstlichen Intelligenz und befasst sich mit der künstlichen Generierung von Wissen aus Erfahrung. Es gibt viele verschiedene Ansätze für Algorithmen für das maschinelle Lernen, welche sich in ihrer Funktionsweise unterscheiden. Jeder dieser Ansätze ist darauf ausgelegt einen bestimmten Problemtyp zu lösen. Man unterschiedet zwischen drei verschiedenen grundlegenden Arten des maschinellen Lernens. Diese sind unsupervised learning, supervised learning und reinforcement learning. Weiterhin wird ein Modell benötigt, welches mit den Daten trainiert wird. In dieser Arbeit werden Tabellen und künstliche neuronale Netze als Modelle in Kombination mit verschiedenen Algorithmen benutzt. Daher entsteht die grundlegende Unterscheidung in tabellarische Methoden und in Methoden mit neuronalen Netzen. Wird ein künstliches neuronales Netz verwendet bezeichnet man dies auch als deep learning.

4.3 Unsupervised learning

Beim unsupervised learning ist dem Computer ein großer Datensatz gegeben. Der Computer soll nun selbstständig Muster erkennen und Kategorien oder Zusammenhänge erstellen. Dadurch können Vorhersagen von weiteren Eingaben gemacht werden.

4.4 Supervised learning

Beim supervised learning soll der Computer eine Funktion lernen, welche zu bestimmten Eingaben gewünschte Ausgaben liefert. Dazu bekommt der Computer ein

Set von Trainingsbeispielen, wobei jedes Beispiel aus einer Eingabe und einer dazugehörigen gewünschten Ausgabe besteht. Nach Analyse dieses Trainingssets soll der Computer auch die gewünschten Ausgaben zu Eingaben finden, die nicht in den Trainingsdaten enthalten waren. Die äquivalente Art zu lernen, welche Tiere verwenden heißt concept learning. Eine weitere Art des maschinellen Lernens ist das semi-supervised learning. Es kombiniert Methoden vom supervised und vom unsupervised learning. Hier sind nur bei einem kleinen Teil der Trainingsdaten zu den Eingaben die gewünschten Ausgaben bekannt.

4.5 Reinforcement learning

Beim reinforcement learning interagiert der Computer mit einer Umgebung und erhält basierend auf seinen Eingaben Belohnungen. Ziel ist es, diese Belohnungen zu maximieren. Ähnliche Methoden werden auch beim active learning benutzt, bei dem der Computer ein großen Datensatz an Eingaben bekommt. Er versucht durch Interaktionen mit einer Umgebung die gewünschten Ausgaben zu bestimmten Eingaben zu erfragen und mithilfe dieser Trainingspaare die gewünschten Ausgaben zu den restlichen Eingaben abzuschätzen. Das Ziel beim active learning ist es, eine Strategie zu finden, bei der mit möglichst geringem Aufwand, ausgehend von bestimmten Eingaben, die gewünschten Ausgaben erfragt werden.

5 Einleitung

Reinforcement learning basiert auf dem Prinzip des trail-and-error learning. Der Lernende versucht, in einer Umwelt ein bestimmtes Ziel zu erreichen und sucht nach geeigneten Aktionen, um diesem Ziel näherzukommen. Wenn ein Baby laufen lernt, macht es viele verschiedene Bewegungen, um im Stand das Gleichgewicht zu halten, jedoch fällt es Anfangs immer wieder hin. Nach jedem Versuch merkt es, dass die Bewegungen, die es gerade gemacht hat, auf eine bestimmte Art und Weise dazu geführt haben, dass der Versuch zu Laufen scheiterte. Das Baby analysiert diesen Sachverhalt und beim nächsten Versuch wird das Baby sich in seinen Bewegungen anders verhalten, um der Art und Weise wie es gerade scheiterte entgegenzuwirken. Über viele Versuche hinweg werden so immer mehr Erfahrungen gesammelt und die Bewegungsabläufe mit jedem mal optimiert. An diesem einfachen Beispiel kann man das Prinzip des reinforcement learnings sehr gut verdeutlichen. Ein *agent* (dt. Agent = das Baby) interagiert mit einem *environment* (dt. Umwelt, Umgebung = der Boden und die Gravitation, die das Baby in seinen Bewegungen beeinflussen), um ein bestimmtes Ziel (=Laufen) zu erreichen. Hierbei bekommt der *agent* von *environment* einen *reward* (dt. Belohnung = Zeit des aufrechten Stehens; gelaufene Strecke), welcher dem *agent* angibt, wie nützlich seine *action* (dt. Aktion = Kombination aus Bewegungen) im aktuellen *state* (dt. Status = Position des Kör-

pers des Babys und die aktuellen Bewegungsvektoren) war. Der *agent* passt aus diesen Informationen seine aktuelle *policy* (dt. Strategie) an. Die Grundlage einer *policy* ist eine *value function*, welche die erwarteten *rewards* in der Zukunft angibt. Diese Berechnung wird andauernd neu durchgeführt und so beurteilt, welche *states* und *actions* gut und welche schlecht sind. Mit diesen Werten wird die *policy* angepasst. Es gibt jedoch auch Methoden, welche sich dazu eignen solche Probleme zu lösen, aber keine *value functions* berechnen. Beispielsweise interagieren bei evolutionären Methoden mehrere statische *policies* mit separaten Instanzen des *environment* über eine bestimmte Zeitspanne. Am Ende dieser Zeitspanne werden die *policies*, welche die höchsten *rewards* bekommen haben in einen nächste Generation von *policies* übertragen und noch einige zufällige Variationen dieser *policies* hinzugefügt. Dann wird der gesamte Vorgang wiederholt, um so eine optimale *policy* zu ermitteln. In dieser Arbeit werden jedoch nur reinforcement learning Methoden benutzt, die auf der Berechnung einer *value function* basieren. Reinforcement learning benutzt das Modell des Markow-Entscheidungsprozesses, um die Interaktion zwischen *agent* und *environment* zu definieren. Daher lassen sich Probleme, welche als ein Markow-Entscheidungsprozess formuliert werden können, meistens gut mit reinforcement learning lösen. Dies ist auch beim Pfannkuchen-Sortierproblem der Fall.

6 Markow-Entscheidungsprozess

Markow-Entscheidungsprozesse (engl. markov decision process; Abkürzung: MDP) bilden die Grundlage des environments in dem sich der agent befindet. Der agent interagiert in diskreten Zeitschritten $t = 0, 1, 2, ...$ mit dem environment. Zu jedem Zeitpunkt t bekommt der agent vom environment die Information seines aktuellen state $s_t \in S$. S ist hierbei die Menge aller möglichen states. Basierend auf dieser Information, wählt der agent durch Anwendung einer bestimmten policy eine action $a_t \in A$ aus. A ist die Menge aller möglichen actions. Diese wird hier unabhängig vom state definiert, da auch beim Pfannkuchen-Sortierproblem diese Menge in jedem state gleich ist. Durch die Durchführung einer action im environment wechselt der das environment in den nächsten state und geht gleichzeitig einen Zeitschritt weiter. Der agent erhält nun vom environment einen reward $r_{t+1} \in R$ mit $R \subset \mathbb{R}$ und seinen neuen state $s_{t+1} \in S$. Diese Abfolge von state-action-reward-state nenne ich einen Übergang. Hierbei ist R die Menge aller möglichen rewards.

Bei finiten MDPs sind die Mengen S, A und R finit. Da dies beim Pfannkuchen-Sortierproblem auch der Fall ist, werden im Folgenden auch nur finite MDPs betrachtet. Weiterhin muss eine Startverteilung $p_0 : S \mapsto \mathbb{R}$ gegeben werden, welche angibt wie wahrscheinlich es iste in einem state zu starten.

Um das Modell komplett zu definieren, muss man noch die Dynamiken des

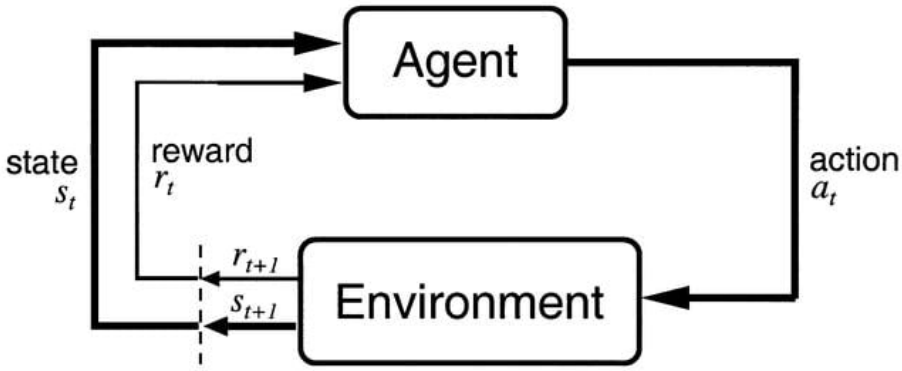

Abbildung 1: Interaktion von agent und environment. Abb. aus [1]

Modells beschreiben. Dies geschieht indem man die Wahrscheinlichkeit q dafür beschreibt, dass ein bestimmter state s' und ein bestimmter reward r zur Zeit t auftreten, wenn zur Zeit $t-1$ im state s die action a durchgeführt wurde.

$$q(s', r \mid s, a) \doteq P(s_t = s', r_t = r \mid s_{t-1} = s, a_{t-1} = a) \, \forall s, s' \in S, r \in R, a \in A$$
$$mit \, q : S \times R \times S \times A \mapsto [0, 1]$$

Die state-transition probability (dt. Übergangswahrscheinlichkeit) ist die Wahrscheinlichkeit von einem state s nach Durchführung einer action a in einen anderen state s' zu kommen.

$$p(s' \mid s, a) \doteq P(s_t = s' \mid s_{t-1} = s, a_{t-1} = a) = \sum_{r \in R} q(s', r \mid s, a)$$
$$mit \, p : S \times S \times A \mapsto [0, 1]$$

Damit kann man auch eine reward function (dt. Belohnungsfunktion) definieren. Diese gibt den erwarteten reward bei einem Übergang, d.h. unter gegebenen s,a und s' an.

$$r(s, a, s') \doteq \mathbb{E}(s_{t-1} = s, a_{t-1} = a, s_t = s') = \sum_{r \in R} r \frac{q(s', r \mid s, a)}{p(s' \mid s, a)}$$
$$mit \, r : S \times A \times S \mapsto \mathbb{R}$$

Erfüllt ein Problem besagte Bedingungen, beziehungsweise lässt es sich so definieren, dann besitzt dieses Problem die Markow-Eigenschaft. Dies bedeutet, dass die Wahrscheinlichkeit einen state s' zu erreichen nur von dem vorherigen state s und der in diesem state durchgeführten action a und nicht von deren Vorgängern abhängt. Ist diese Bedingung erfüllt, dann handelt es sich bei einem Problem um einen MDP.

Das Pfannkuchen-Sortierproblem ist ein finiter MDP. S ist die Menge aller Permutationen von Pfannkuchen und A die Menge aller möglichen Größen von Teilsta-

peln, die man abheben kann. Da die Anzahl an Pfannkuchen endlich ist, sind auch $|A|$ und $|S|$ endlich. Für die Startverteilung ist für jeden state die Wahrscheinlichkeit gleich groß der Start-state zu sein, da als Start-state eine zufällige Permutation der Pfannkuchen ausgewählt wird. Außerdem ist es, von einem state ausgehend und nach Invertierung eines Teilstapels, eindeutig wie die Pfannkuchen danach angeordnet sind, d.h. es ist eindeutig in welchen state sich das System danach befindet. Dies hängt nicht davon ab in welchen states sich das System vor dem state vor der durchgeführten action befand. In der Umsetzung im Programm kann man auch jedem Übergang eindeutig einen reward zuweisen.

7 Policy

Der agent wählt actions mit einer bestimmten policy aus. Diese policy ist eine Funktion, welche die Wahrscheinlichkeiten dafür angibt, dass der agent zu einer bestimmten Zeit t in Abhängigkeit des aktuellen states s eine action a auswählt.

$$\pi(s, a) = P(a_t = a \mid s_t = s)$$

Das Ziel des agent ist es, die optimale policy π^* zu lernen. Dafür wird versucht, die Summe der erwarteten rewards zu maximieren. R_t ist die Summe aller zukünftigen rewards ab einem Zeitpunkt t.

$$R_t = \sum_{k=0}^{T} r_{t+k+1}$$

Hierbei wird zwischen episodic und continuing tasks unterschieden. Episodic tasks kommen nach einer endlichen Anzahl Zeitschritten zum Ende ($T \in \mathbb{N}$). T ist der Zeitschritt in dem der agent in einen terminal state kommt und die Episode endet. Danach beginnt eine neue Episode von einem neuen Start-state aus, welcher unabhängig davon ist, wie die letzte Episode endete.

Continuing tasks können unendlich lang laufen, d.h. $T = \infty$. R_t kann dabei unendlich groß werden und somit ist es schwierig R_t zu maximieren. Deswegen verwendet man für continuing tasks einen Diskontinuierungsfaktor $\gamma \in (0, 1]$.

$$R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1}$$

Somit werden rewards nur noch bis zu einer bestimmten Anzahl an Zeitschritten im Voraus beachtet, da $\lim_{k\to\infty} \gamma^k r_{t+k+1} = 0$ für $\gamma < 1$. Für $\gamma = 1$ ergibt sich die Formel für einen episodic task. Für continuing tasks wird der Parameter γ häufig nah bei 1 gewählt. Wird γ zu klein, dann werden nur rewards von wenigen zukünftigen Schritten beachtet und nicht die rewards in fernerer Zukunft. Dies ist ungünstig,

wenn der agent zum Beispiel nach Durchführung einer bestimmten action direkt einen hohen reward bekommen würde, aber in den zukünftigen states immer einen sehr kleinen reward, während er nach Durchführung einer anderen action in den nächsten states kleine rewards bekommen würde, aber nach kurzer Zeit sehr hohe rewards. Bei einem γ nahe 0 würde der agent die erste action wählen und auf lange Sicht einen kleineren reward bekommen als bei Wahl der zweiten action. Diese würde er wählen wenn γ nah bei 1 liegt.

Beim Pfannkuchen-Sortierproblem gibt es einen terminal state, in welchem die Pfannkuchen vollständig sortiert sind. Dann wäre die Episode zu Ende. Dieser state muss aber nicht erreicht werden, da es beim Pfannkuchen-Sortierproblem Kreise im Übergangsgraphen gibt. Beispielsweise könnte immer die selbe Aktion ausgewählt werden und immer der gleiche Teilstapel würde unendlich oft umgedreht werden. Deshalb muss man für das Pfannkuchen-Sortierproblem die Berechnung für einen continuing task anwenden.

8 Value functions

Value functions werden berechnet, um die policy zu optimieren. Dabei gibt es zwei Arten von value functions. State-value functions berechnen die sogenannten V-values. Diese ergeben sich aus der Summe aller erwarteten rewards, wenn nach aktueller policy weiter verfahren würde.

$$V^{\pi}(s) = \mathbb{E}_{\pi}(R_t \mid s_t = s)$$

Action-value functions weisen jeder action in jedem state (jedem sogenannten state-action pair) die Summe der erwarteten rewards zu, wenn nach aktueller policy weiter verfahren würde.

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}(R_t \mid s_t = s, a_t = a)$$

9 Bellman equations

Die V-values und Q-values sind voneinander abhängig. Ihr Zusammenhang kann mit Hilfe der Wahrscheinlichkeit $\pi(s, a)$, eine action a in einem state s zu wählen, beschrieben werden.

$$V^{\pi}(s) = \sum_{a \in A} \pi(s, a) \cdot Q^{\pi}(s, a)$$

Die Q-values sind Abhängig vom state s' nach der durchgeführten action, vom V-value $V^{\pi}(s')$ dieses states und von dem erhaltenen reward r'. So können die Q-Values

auch durch die V-values beschrieben werden.

$$Q^\pi(s,a) = \sum_{s' \in S} p(s' \mid s,a) \cdot [r(s,a,s') + \gamma V^\pi(s')]$$

Wenn man diese beiden Gleichungen ineinander einsetzt erhält man die Bellman equations für die V-values und Q-values.

$$V^\pi(s) = \sum_{a \in A} \pi(s,a) \cdot \sum_{s' \in S} p(s' \mid s,a) \cdot [r(s,a,s') + \gamma V^\pi(s')]$$
$$Q^\pi(s,a) = \sum_{s' \in S} p(s' \mid s,a) \cdot [r(s,a,s') + \gamma \sum_{a' \in A} \pi(s',a') \cdot Q^\pi(s',a')]$$

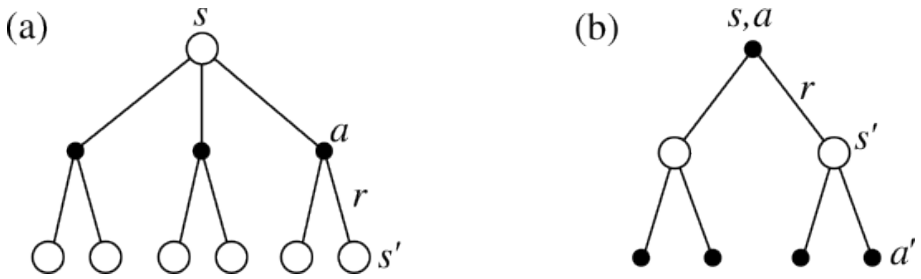

Abbildung 2: „backup diagrams“ zu den jeweiligen Bellman equations. (a): V-values (b): Q-values. Abb. aus [1]

10 Dynamic programming

Wenn ein Problem die Markow-Eigenschaft besitzt, dann gibt es für die Bellman equations nur eine Lösung. Es ist also möglich, für eine bestimmte policy alle V-values und Q-values auszurechnen, indem man das Gleichungssystem aller Bellman equations löst (eine Gleichung pro state). Diesen Vorgang nennt man policy evaluation. Das Gleichungssystem kann mit folgender Vorschrift iterativ gelöst werden.

$$V_{k+1}(s) \leftarrow \sum_{a \in A} \pi(s,a) \cdot \sum_{s' \in S} p(s' \mid s,a) \cdot [r(s,a,s') + \gamma V_k(s')] \; \forall s \in S$$

Für $k \to \infty$ konvergiert V_k sicher zu V^*. Nun kann es sein, dass die aktuelle policy nicht immer die bestmögliche action $a^* = argmax_a Q(a,s)$ in einem state wählt (greedy action). Es wird eine neue policy π' erzeugt, welche in jedem state die greedy action wählt. Dieser Vorgang heißt policy improvement.

Der Vorgang von policy evaluation und policy improvement wird wiederholt, bis beim policy improvement $\pi' = \pi$ gilt. Dieser Algorithmus heißt policy iteration. Besitzt das Problem die Markow-Eigenschaft gilt, dass dieser Vorgang von policy evaluation und improvement bis zu einer optimalen policy π^* konvergiert. Demnach gilt beim Ende des Algorithmus $\pi' = \pi = \pi^*$. Dynamic programming kann immer die

optimale policy finden. Für die optimale policy π^* gelten folgende Bellman equations.

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} p(s' \mid s, a) \cdot [r(s, a, s') + \gamma V^*(s')]$$

$$Q^*(s, a) = \sum_{s' \in S} p(s' \mid s, a) \cdot [r(s, a, s') + \gamma \max_{a' \in A} Q^*(s', a')]$$

Da bei policy iteration vor jedem policy improvement eine komplette policy evaluation durchgeführt wird, ist dieser Algorithmus extrem rechenaufwändig. Eine Verbesserung zu policy iteration bietet value iteration. Hier wird nach jedem Iterationsschritt während der policy evaluation policy improvement durchgeführt. Die Regel zum iterativen Lösen des Gleichungssystems kann wie folgt vereinfacht werden.

$$V_{k+1}(s) \leftarrow \max_{a \in A} \sum_{s' \in S} p(s' \mid s, a) \cdot [r(s, a, s') + \gamma V_k(s')] \, \forall s \in S$$

Dieser Algorithmus konvergiert schneller zu π^* als policy iteration. Trotzdem benötigt value iteration auch noch einen vergleichsweise hohen Rechenaufwand. Aufgrund dieser Tatsache ist dynamic programming keine gute Lösungsvariante für das Pfannkuchen-Sortierproblem, da nur kleine Problemgrößen praktikabel sind. Bei n Pfannkuchen gibt es $n!$ states (alle verschiedenen Permutationen). Dynamic programming wäre für maximal 10 Pfannkuchen mit $10! = 3628800$ states geeignet, da in der Praxis normalerweise wenige Millionen states das Limit für dynamic programming sind.

11 Monte-Carlo Methoden

Im Gegensatz zu dynamic programming funktionieren Monte-Carlo Methoden auch, wenn nicht die kompletten Dynamiken des environment bekannt sind. Man führt zuerst verschiedene Episoden nach der aktuellen policy von verschiedenen states aus durch, bis man einen terminal state erreicht (die sogenannten sample episodes). Nach jeder Episode werden alle R_t für alle verschiedenen t, welche während der Episode durchlaufen wurden, berechnet. Danach kann man für einen state s den V-value wie folgt mitteln.

$$V^\pi(s) = \mathbb{E}_\pi(R_t \mid s_t = s) = \sum_{k=1}^{K} R_t^{(k)}$$

K gibt hierbei an, wie oft der state s in den vorher durchgeführten sample episodes vorkam. In der Praxis werden jedoch die V-values von states nach jeder Episode folgendermaßen abgeschätzt.

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha \cdot (R_t - V^\pi(s))$$

Nach selbigem Vorgehen können auch die Q-values abgeschätzt werden.

$$Q^\pi(s,a) \leftarrow Q^\pi(s,a) + \alpha \cdot (R_t - Q^\pi(s,a))$$

α ist hierbei der step-size Parameter. Er gibt an, wie stark die neuen Werte nach den alten Werten angepasst werden. Ist α zu klein, dann wird nur sehr langsam gelernt. Ist α zu groß, dann können die Ergebnisse einzelner Episoden, welche nicht nach optimaler policy durchgeführt wurden, bereits gelernte und eventuell bessere V-values beziehungsweise Q-values überschreiben und somit den Lernfortschritt zerstören. Es kann also sein dass gar nicht gelernt wird wenn α zu groß ist.

Nach Berechnung neuer V-values oder Q-values muss noch ähnlich wie bei dynamic programming die policy verbessert werden. Es kann wieder policy improvement durchgeführt werden, sodass die neue policy immer die greedy action wählt. Man schreibt zur Vereinfachung die action, welche von einer policy π im state s gewählt wird als $\pi(s)$. Beim policy improvement wird also für alle s $\pi_{t+1}(s) = a$ gesetzt. Nun kann wieder policy evaluation und policy improvement abwechselt durchgeführt werden bis die optimale policy gefunden wurde.

Ein Problem bei Monte-Carlo Methoden ist das exploration-exploitation dilemma. Wählt der agent immer die greedy action, dann nennt man das exploitation der bisherigen Abschätzungen. Da zu Beginn noch kein Wissen über V-values oder Q-values besteht, wählt der agent manchmal zufällige actions, welche jedoch im jeweiligen state nicht die optimalen actions sind. Diese gewählten actions werden nach erstmaliger policy evaluation automatisch zu den greedy actions und ab dort werden immer diese actions durchgeführt. Dadurch kann es sein, dass manche optimalen actions niemals durchgeführt werden, d.h. die optimale policy kann nicht gefunden werden. Um dies zu verhindern muss der agent auch manchmal non-greedy actions durchführen. Dies nennt man exploration, da neue actions erkundet werden, welche besser oder schlechter als die bisherige beste action sein können. Handelt der agent nur nach exploration, kann niemals eine gute policy gefunden werden, da immer zufällig gute oder schlechte actions durchgeführt werden. Die Lösung dieses Problems ist es, ein Mittel zwischen exploration und exploitation zu finden.

Für dieses Problem gibt es verschiedene Ansätze. Bei On-policy Methoden muss eine policy gegeben sein, welche ϵ-soft ist. Dies bedeutet dass alle actions eine Wahrscheinlichkeit von mindestens $\frac{\epsilon}{|A|}$ haben müssen ausgewählt zu werden. Dies sichert das Vorhandensein von exploration und dass die policy gegen die optimale policy konvergiert.

Zwei einfache Methoden für ϵ-soft policies sind der ϵ-greedy Algorithmus und der softmax Algorithmus. Beim ϵ-greedy Algorithmus wird die greedy action mit einer Wahrscheinlichkeit von $1-\epsilon$ mit $\epsilon \in (0,1)$ ausgewählt. Mit einer Wahrscheinlichkeit von ϵ wird eine zufällige non-greedy action ausgewählt. ϵ wird häufig nahe null gewählt, damit die greedy action die größte Wahrscheinlichkeit hat gewählt zu

werden. Der Nachteil beim ϵ-greedy Algorithmus ist, dass die zweitbeste action mit der gleichen Wahrscheinlichkeit gewählt wird wie die schlechteste.

Beim softmax Algorithmus hingegen werden die Wahrscheinlichkeiten für die einzelnen actions durch die normierten Q-values der actions in einem bestimmten state festgelegt.

$$\pi(s,a) = \frac{\exp(\frac{Q(s,a)-Q(s,a^*)}{\tau})}{\sum_{a\in A}\exp(\frac{Q(s,a)-Q(s,a^*)}{\tau})} \tag{1}$$

Da die Funktionswerte der Exponentialfunktion schnell sehr groß werden können, wird erst der Q-value von der greedy action von allen Q-values subtrahiert, d.h. alle diese Differenzen sind ≤ 0 und somit sind die Funktionswerte der Exponentialfunktion > 0 und ≤ 1. Der Parameter τ hat eine ähnliche Funktion wie ϵ im ϵ-greedy Algorithmus. τ ist eine reelle positive Zahl und wird temperature genannt. Ist τ sehr groß, haben alle actions ähnliche Wahrscheinlichkeiten gewählt zu werden. Ist τ sehr klein, dann wird fast immer die greedy action gewählt. Ist $\tau < 1$, dann können die Funktionswerte der Exponentialfunktion natürlich wieder > 1, aber nicht extrem groß werden.

Bei off-policy Methoden wird neben der estimation policy π noch eine weitere policy benutzt, die behavior policy π'. Bei off-policy Monte Carlo kann man eine deterministische (greedy) policy π lernen, wobei das exploration-exploitation Problem durch die policy π' gelöst wird. Diese kann z.B. mit einem ϵ-greedy Algorithmus arbeiten und ist dazu da, sample episodes zu erstellen. π' lernt nach on-policy Methoden und nach jeder Episode werden die erwarteten rewards nach der relativen Wahrscheinlichkeit des Auftretens nach π und π' gewichtet und so die Q-values für π berechnet.

$$Q^\pi(s,a) = \frac{\sum_{i=1}^{N}\frac{p_i}{p_i'}\cdot R_i(s,a)}{\sum_{i=1}^{N}\frac{p_i}{p_i'}} \; mit \; \frac{p_i}{p_i'} = \prod_{k=t}^{T-1}\frac{\pi(s_k,a_k)}{\pi'(s_k,a_k)}$$

t ist hierbei der Zeitpunkt, ab dem π und π' verschiedene actions wählen. Der Vorteil von off-policy Methoden ist, dass man seine policy π mit Episoden trainieren kann, welche von einer anderen policy durchgeführt wurden. So können z.B. beim Schach Spiele von Experten als sample episodes genutzt werden. Der Vorteil ist, dass nur gute actions gelernt werden, weil sehr schlechte actions von Experten niemals gewählt werden und somit auch gar nicht gelernt werden. Dies fördert einen schnellen Lernprozess. Der Nachteil dabei ist, dass wenn die behavior policy niemals die optimale Lösung findet, dass dann der agent auch keine Möglichkeit diese zu lernen.

Eine Einschränkung von Monte-Carlo Methoden ist, dass sie nur auf episodic tasks angewandt werden können und immer nur am Ende einer Episode gelernt wird. Da beim Pfannkuchen-Sortierproblem das Erreichen eines terminal states in endlicher Zeit nicht gegeben ist, können die Episoden sehr lang werden, ohne dass etwas gelernt wird. Dadurch kann der Lernprozess nur sehr langsam vorangehen. Aufgrund

dessen sind Monte-Carlo Methoden zur Lösung des Pfannkuchen-Sortierproblems auch eher ungeeignet.

12 Temporal-Difference learning

Bei Monte-Carlo Methoden werden immer ganze Episoden benötigt, da zur Berechnung der V-values und Q-values die Summe der gesamten erhaltenen rewards R_t verwendet wird. Temporal-Difference learning umgeht dieses Problem, indem die gleiche Formel zur Berechnung von V-values und Q-values verwendet wird, aber R_t durch eine Näherung aus dem erhaltenen reward und dem V-value beziehungsweise dem Q-value des nächsten state beziehungsweise der nächsten action. Die neuen Update-Regeln für die V-values und Q-values ergeben sich entsprechend wie folgt.

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha \cdot (r(s,a,s') + \gamma V^\pi(s') - V^\pi(s)) \tag{2}$$

$$Q^\pi(s,a) \leftarrow Q^\pi(s,a) + \alpha \cdot (r(s,a,s') + \gamma Q^\pi(s',a') - Q^\pi(s,a)) \tag{3}$$

$\delta = r(s,a,s') + \gamma V^\pi(s') - V^\pi(s)$ beziehungsweise $\delta = r(s,a,s') + \gamma Q^\pi(s',a') - Q^\pi(s,a)$ wird als reward-prediction error oder auch als TD error bezeichnet. Der TD error gibt an, wie gut die aktuelle Vorhersage der V-values beziehungsweise Q-values mit den tatsächlich erhaltenen Werten nach Durchführung einer action übereinstimmt. Ist $\delta > 0$, dann war der Wert des letzten state beziehungsweise der letzten action zu niedrig und er wird erhöht. Ist $\delta < 0$, dann war der Wert des letzten state beziehungsweise der letzten action zu hoch und er wird verringert. Ist $\delta = 0$, dann war der Wert des letzten state beziehungsweise der letzten action genau so hoch wie er vorhergesagt wurde und muss daher nicht verändert werden. Wenn α klein genug gewählt ist, dann konvergieren Temporal-Difference Methoden immer gegen V^π.

Ein Vorteil beim Temporal-Difference learning ist, dass schon während dem Durchlaufen einer Episode gelernt werden kann. Dies nennt man online learning. Temporal-Difference Methoden können daher auch auf continuing tasks angewendet werden.

Abbildung 3: „backup diagram“ für Temporal-Difference learning nach Gleichung 2. Abb. aus [1]

Der Nachteil beim Temporal-Difference learning ist, dass die V-values und Q-values nur mit Abschätzungen von R_t aktualisiert werden. Es kann dadurch einige

Zeit dauern bis die realen Werte erreicht sind. Trotzdem konvergieren Temporal-Difference Methoden meistens schneller als Monte-Carlo Methoden.

Es ist meist praktischer direkt die Q-values zu lernen, da man dann von einem state aus nicht erst berechnen muss welche action die beste ist, ausgehend von den V-values aller möglichen nächsten states, sondern man rechnet direkt mit den Q-values. In der Update-Regel für die Q-values wird aber der Wert der action gebraucht, welche im nächsten state gewählt wird. Um diese action zu bestimmen gibt es zwei verschiedene Möglichkeiten.

On-policy Temporal-Difference learning wird SARSA genannt. Dabei wird die action, welche im nächsten state gewählt wird, nach der aktuellen policy bestimmt. Es wird also die action $\pi(s')$ gewählt.

$$\delta = r(s, a, s') + \gamma Q^{\pi}(s', \pi(s')) - Q^{\pi}(s, a) \tag{4}$$

Die gelernte policy muss ϵ-soft sein. Wenn α klein genug gewählt ist und ϵ nicht konstant gewählt wird, sondern über viele Episoden hinweg zu 0 konvergiert, dann konvergiert SARSA zur optimalen policy.

Off-policy Temporal-Difference learning wird Q-learning genannt. Dabei ist die action, welche im nächsten state gewählt wird, immer die greedy action. Dies bedeutet nicht, dass nach dem Übergang vom aktuellen state in den nächsten state wirklich die greedy action von der policy gewählt wird. Sie wird nur zur Berechnung der Q-values der actions im aktuellen state verwendet.

$$\delta = r(s, a, s') + \gamma \max_{a'} Q^{\pi}(s', a') - Q^{\pi}(s, a) \tag{5}$$

Genau wie bei off-policy Monte-Carlo Methoden gibt es bei Q-learning das Risiko, dass die optimale policy niemals erreicht wird.

Auch bei Temporal-Difference Methoden gibt es das exploration-exploitation Problem. Häufig werden einfach ϵ-soft policies verwendet. Die richtige Mitte zwischen exploration und exploitation muss durch Variation gewisser parameter (z.B. ϵ bei einem ϵ-greedy Algorithmus oder τ bei einem softmax Algorithmus) gefunden werden.

13 Eligibility traces

Temporal-Difference Methoden können sehr lange Zeit benötigen bis sie eine gute policy erreichen, da in jedem Übergang immer nur die Q-values oder V-values im aktuellen Übergang aktualisiert werden. Dies ist besonders schwerwiegend, wenn im Problem nur sparse rewards gegeben sind. Dass heißt der agent bekommt nur sehr selten einen reward. Dies ist beim Pfannkuchen-Sortierproblem der Fall, da nur ein reward von 1 gegeben wird wenn alle Pfannkuchen vollständig sortiert sind, also am

Ende der Episode. Alle anderen actions erhalten einen reward von 0. Sollten alle Q-values mit 0 initialisiert worden sein, dann kann der agent in der ersten Episode nur die letzte durchgeführte action updaten und in der zweiten Episode nur die letzte und die vorletzte action usw.

Für das Erreichen eines bestimmten states ist nicht nur die davor getätigte action, sondern alle vorher durchgeführten actions verantwortlich. Eligibility traces nutzen diese Idee indem sie den TD error zur Zeit t (δ_t) nicht nur auf die durchgeführte action a_t übertragen, sondern auch auf alle actions, die vorher in der Episode gewählt wurden. Hierbei wird der decaying factor $\lambda \in (0, 1)$ verwendet, welcher angibt wie sehr sich der aktuelle TD error auf die vorherigen actions auswirkt. Eligibility traces verbinden damit die Konzpte von Monte-Carlo Methoden ($\lambda = 1$) und Temporal-Difference Methoden ($\lambda = 0$).

Für die Umsetzung gibt es zwei verschiedene Konzepte. Bei der forward view werden alle TD errors, welche in Zukunft gemacht werden, benutzt und mit λ verringert, um den Q-value der action a_t zur Zeit t zu berechnen.

$$Q^\pi(s_t, a_t) \leftarrow Q^\pi(s_t, a_t) + \alpha \sum_{t'=t}^{T} (\gamma\lambda)^{t'-t} \cdot \delta_t$$

Der Nachteil dieser Methode ist allerdings wieder, dass eine endliche Episode vorausgesetzt wird. Diese Methode verhindert also online learning.

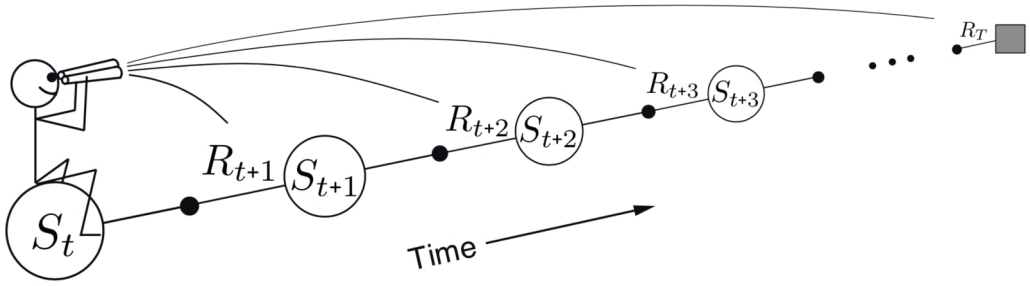

Abbildung 4: Prinzip der forward view. Abb. aus [1]

Bei der backward view beeinflusst der TD error zu einer bestimmten Zeit alle vorherigen durchgeführten actions. Je länger es her ist, dass der agent zu einem bestimmten Zeitpunkt t zuletzt eine bestimmte action in einem state durchgeführt hat, desto weniger beeinflusst δ_t diese action, d.h. umsto mehr muss der TD error verringert werden. Dies wird mit einem eligibility trace umgesetzt, welcher in jedem Zeitschritt für jedes state-action pair aktualisiert wird. Dabei wird der Wert der aktuell gewählten action um 1 erhöht und die Werte aller anderen state-action pairs

mit λ verringert.

$$e(s,a) = \begin{cases} e(s,a) + 1 & \text{falls } s = s_t \text{ und } a = a_t \\ \lambda \cdot e(s,a) & \text{sonst} \end{cases} \tag{6}$$

Danach werden die Q-values aller state-action pairs proportional zum korrespondierenden eligibility trace aktualisiert.

$$Q^\pi(s,a) \leftarrow Q^\pi(s,a) + \alpha \cdot e(s,a) \cdot \delta_t \; \forall s \in S, a \in A \tag{7}$$

Bei der backward view ist online learning möglich.

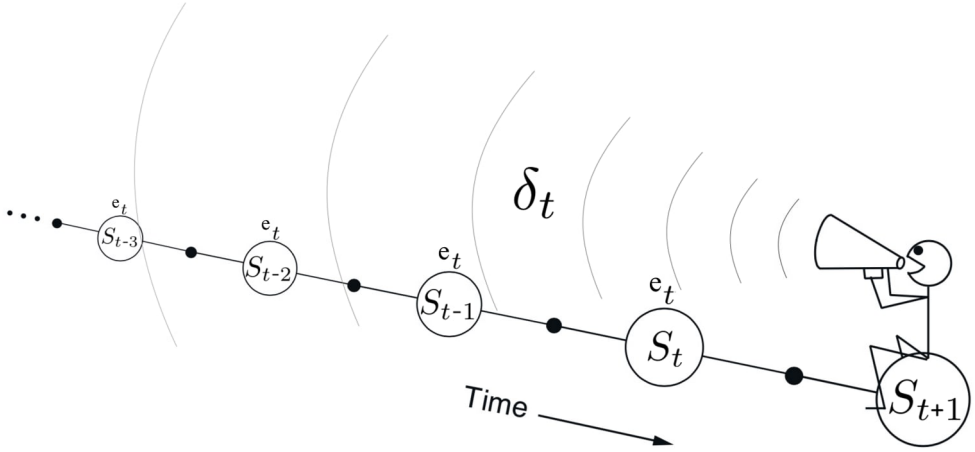

Abbildung 5: Prinzip der backward view. Abb. aus [1]

Die Algorithmen Temporal-Difference learning, SARSA und Q-learning können alle mit eligibility traces erweitert werden. Häufig wird die backward view verwendet. Diese Variationen werden dann als TD(λ), SARSA(λ) und Q(λ) bezeichnet. Durch eligibility traces kann viel schneller gelernt werden, da selbst bei sparse rewards immer alle Q-values von actions, welche in der episode gewählt wurden mit einem Mal aktualisiert werden können. Schneller lernen trifft jedoch nur zu, wenn man die Anzahl der Episoden betrachtet, die benötigt werden um eine gute policy zu erreichen. Der Nachteil bei eligibility traces ist, dass ein großer Rechenaufwand betrieben wird, da in jedem Zeitschritt alle Q-values aktualisiert werden. Wenn man die reale Zeit betrachtet sind Algorithmen mit eligibility traces also nicht unbedingt schneller als Algorithmen ohne diese. Ein weiterer Nachteil ist, dass für die Speicherung von eligibility traces ein genau so großer Speicherplatz benötigt wird, wie für die Q-values.

14 Reward scheduling

Sind bei einem Problem nur sparse rewards gegeben, versucht man im Problem eine andere Art von reward zu konstruieren, welche man auch zwischen den sparse rewards vergeben kann. Ist dies bei einem Problem mit sparse rewards nicht möglich, dann ist eine weitere Möglichkeit, den Lernprozess zu fördern, reward scheduling. Dabei bekommt der agent für alle actions, für die er sonst keine rewards bekommen würde, einen kleinen negativen reward, wie z.B. -1. Die positiven sparse rewards werden sehr groß gesetzt, wie z.B. 50 oder 100. Je länger der agent benötigt, um große sparse rewards zu bekommen, desto mehr negative rewards erhält er. Dadurch versucht er andere Wege zu finden, welche möglicherweise schneller sind.

Teil II

Reinforcement learning mit künstlichen neuronalen Netzen

15 Funktionsabschätzung

Die bisher vorgestellten Methoden des reinforcement learning waren alle tabellarische Methoden, d.h. es werden alle V-values oder Q-values in einer Tabelle abgespeichert. Wenn man beispielsweise die Q-values in einer Tabelle speichert muss man pro state-action pair einen Wert speichern. Der benötigte Speicherplatz kann aus diese Weise sehr schnell extrem groß werden. Beim Pfannkuchen-Sortierproblem mit n Pfannkuchen gibt es $n! \cdot n$ state-action pairs. Bei 10 Pfannkuchen muss man also schon 36288000 Werte speichern. Zusätzlich muss man, um die optimale policy zu erreichen, in jedem state jede action schon mal durchgeführt haben. Darum dauert es oft extrem lange, die optimale policy zu erreichen.

Ein anderer Ansatz für dieses Problem ist die Generalisierung oder Verallgemeinerung. Das Grundprinzip ist es, bereits erlangtes Wissen auf ähnliche Situationen zu übertragen. Dabei muss man nur die Werte von wenigen state-action pairs kennen und kann mit diesem Wissen, die Werte neuer state-action pairs abschätzen, in Abhängigkeit davon, wie ähnlich die neuen state-action pairs den bereits bekannten sind. Dies wird durch Funktionsabschätzung erreicht. Anstatt alle Q-values einzeln in einer Tabelle zu speichern, lernt ein function approximator die Q-values oder eine policy.

15.1 value-basierte Funktionsabschätzung

Bei value-basierten Methoden soll der function approximator die Q-values $Q^{\pi}(s, a)$ direkt ausgeben. Entweder kann der function approximator ein state-action pair (s, a) als Eingabe haben und einen Q-value ausgeben oder er kann einen state s als Eingabe haben und die Q-values von allen möglichen actions in diesen state ausgeben. Für letztere Variante muss die Menge aller möglichen actions diskret sein. Dafür hat sie den Vorteil, dass oft bei ähnlichen states die Q-values besser abgeschätzt werden können. Beim Pfannkuchen-Sortierproblem ist immer $|A| \in \mathbb{N}$. Da diese Bedingung für die zweite Variante erfüllt ist, wird auch nur diese zweite Variante in dieser Arbeit verwendet werden.

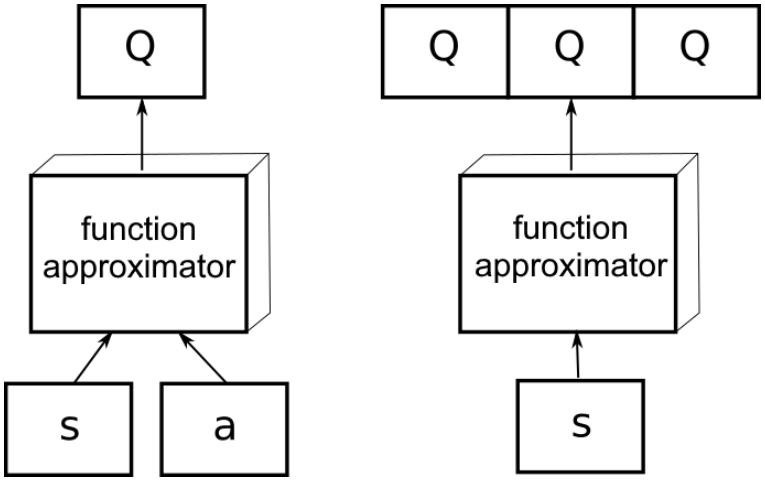

Abbildung 6: Links wird ein state-action pair als Eingabe verwendet und rechts nur ein state. Abb. aus [2]

Der function approximator beinhaltet einen Satz Parameter, welcher als θ bezeichnet wird. Diese Parameter werden während des Lernvorgangs angepasst. Die Q-values, die von diesem function approximator berechnet wurden sind also auch von θ abhängig und werden als $Q_{\theta}(s, a)$ geschrieben. Die policy verwendet direkt die Q-values um zu entscheiden, welche actions gewählt werden. Sie hängt also auch von θ ab und wird mit π_{θ} bezeichnet.

Der function approximator versucht, eine loss function $\mathcal{L}(\theta)$ (dt. Verlustfunktion oder Kostenfunktion) zu minimieren, welche so konstruiert ist, dass die geschätzten Q-values zu ihren optimalen Werten konvergieren.

15.2 policy-basierte Funktionsabschätzung

Bei policy-basierten Methoden lernt der function approximator direkt eine policy $\pi_{\theta}(s, a)$, welche die Summe der erwarteten rewards für alle Übergänge maximiert, d.h. diese actions wählt, für die diese Summe maximal wird. Es können zwar value functions benutzt werden, um die policy zu trainieren, d.h. um ihre Parameter

anzupassen, aber die Auswahl der actions geschieht völlig unabhängig von solchen value functions. Die policy soll Wege (Folge von states und actions von Anfang bis Ende einer Episode) finden welche nur hohe rewards bringen. Um die policy zu berechen müsste man über alle möglichen Wege integrieren, was vom Rechenaufwand her nicht möglich ist. Es gibt zwar Methoden mit denen Abschätzungen gemacht werden können, aber diese werden in dieser Arbeit nicht weiter betrachtet, da zu Lösung des Pfannkuchen-Sortierproblems nur value-basierte Methoden benutzt werden.

16 Deep reinforcement learning

Wird als function approximator ein deep neural network (Abkürzung: DNN) verwendet, spricht man von deep reinforcement learning. Ein deep neural network ist eine Art von künstlichen neuronalen Netz, welches aus einem input, einem oder mehreren hidden layers und einem output layer besteht. Jeder layer besteht aus mehreren künstlichen Neuronen und aus Gewichten, welche die künstlichen Neuronen mit den künstlichen Neuronen des vorhergehenden layers verbinden. Eine Besonderheit bei den hier genutzten deep neural networks ist, dass jedes Neuron in einem layer mit jedem Neuron des vorherigen layer verbunden ist. Die layers werden daher als fully-connected bezeichnet. Es gibt andere Arten von neuronalen Netzen wie z.B. convolutional networks, bei denen das nicht so ist. Diese eignen sich jedoch nicht für die Lösung des Pfannkuchen-Sortierproblems und werden hier nicht weiter betrachtet. Jedes Neuron bekommt als Eingabe die Ausgaben aller anderen Neuronen des vorherigen layers, jeweils gewichtet mit den Gewichten. Es wird ein Schwellwert addiert und eine activation function (dt. Aktivierungsfunktion) angewendet. Der Funktionswert dieser activation function wird ausgegeben. Für einen ganzen layer k kann man also sagen, dass der output ein Vektor h_k ist. Der layer k bekommt also den output-Vektor h_{k-1} als Eingabe. Der layer k multipliziert h_{k-1} mit einer Gewichtematrix W_k, addiert einen bias-Vektor b_k und wendet eine nicht lineare activation function f an. Der input-Vektor des DNN wird als x bezeichnet und der output (=output-Vektor des output layer) mit y.

$$h_k = f(W_k \times h_{k-1} + b_k)$$

Das neuronale Netz lernt, indem es versucht eine loss function, welche abhängig von allen Gewichten und biases ist, zu minimieren. Die Gewichte und biases werden dementsprechend angepasst. Alle Gewichte und biases in einem DNN werden als θ zusammengefasst. Um das Netz zu trainiren wird ein Satz von Trainingsdaten $\mathcal{D}$ verwendet.

Was für eine loss function man verwendet hängt davon ab, ob es sich beim Problem um ein regression problem oder ein classification problem handelt. Das Berechnen von Q-values ist ein regression problem. Für regression problems wird versucht

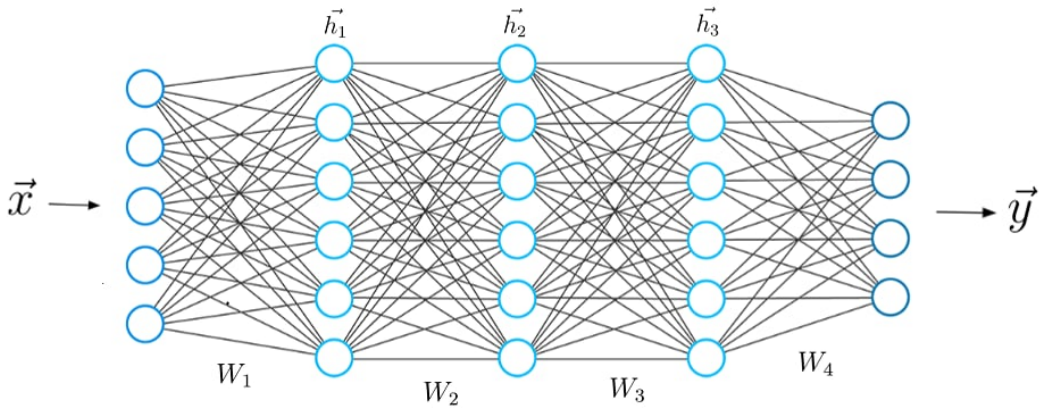

Abbildung 7: Schematische Skizze eines fully-connected deep neural network mit drei hidden layers. Abb. aus [3]

den mean square error (Abkürzung: mse; dt. mittlere quadratische Abweichung) zu minimieren.

$$\mathcal{L}(\theta) = \mathbb{E}_{x,t\in\mathcal{D}}[\| t - y \|^2]$$

x ist der input, y die Vorhersage des DNN für den input x und t der reale output, welcher durch $\mathcal{D}$ festgelegt ist.

Die loss function für SARSA formuliert sich wie folgt.

$$\mathcal{L}(\theta) = \mathbb{E}_{\pi}[(r(s,a,s') + \gamma Q_\theta(s', \pi(s')) - Q_\theta(s,a))^2]$$

Äquivalent dazu ist auch die folgende loss function für Q-learning.

$$\mathcal{L}(\theta) = \mathbb{E}_{\pi}[(r(s,a,s') + \gamma \max_{a'} Q_\theta(s', a') - Q_\theta(s,a))^2] \tag{8}$$

Die Minimierung der loss function geschieht basierend auf dem Gradientenverfahren. Dabei werden alle Parameter in θ einzeln so verändert, dass sich der Funktionswert der loss function etwas verkleinert, so lange bis ein Minimum gefunden wurde.

$$\Delta\theta = -\eta\nabla_\theta\mathcal{L}(\theta) = -\eta\frac{\partial\mathcal{L}(\theta)}{\partial\theta}$$

Die learning rate η hat eine ähnliche Funktion wie der step-size Parameter α und wird meist klein gewählt, um eine genaue Konvergenz gegen das Minimum zu ermöglichen. Die partielle Ableitung der loss function wird durch einen backpropagation Algorithmus berechnet.

$$\frac{\partial\mathcal{L}(\theta)}{\partial W_k} = \frac{\partial\mathcal{L}(\theta)}{\partial y} \times \frac{\partial y}{\partial h_n} \times \frac{\partial h_n}{\partial h_{n-1}} \times ... \times \frac{\partial h_k}{\partial W_k}$$

Von der loss function aus wird also jeder layer in umgekehrter Reihenfolge aktuali-

siert. Die nun benötigten partiellen Ableitungen sind alle gegeben.

$$\frac{\partial h_k}{\partial h_{k-1}} = f'(W_k \times h_{k-1} + b_k)W_k$$
$$\frac{\partial h_k}{\partial W_k} = f'(W_k \times h_{k-1} + b_k)h_{k-1}$$

Die partiellen Ableitungen werden in vielen Bibliotheken automatisch berechnet. Allerdings benötigt man noch einen optimizer. Dies ist eine Funktion, welche in Abhängigkeit der berechneten Steigungen festlegt, wie genau die Werte der Parameter der DNN angepasst werden sollen. In den meisten Bibliotheken sind schon viele gebräuchliche optimizer mit standardisierten Parametern zur direkten Verwendung implementiert. Optimizer benutzen als Trainingsdaten $\mathcal{D}$ sogenannte minibatches. Ein minibatch ist eine zufällige Auswahl von Trainingsbeispielen. Die Anzahl der Trainingsbeispiele, welche in einem minibatch vorhanden sind, nennt man batch size. Ein Trainingsbeispiel wäre beim Temporal-Difference learning ein Übergang.

17 Deep Q-learning

Damit ein neuronales Netz lernt, würde man nun Trainingsbeispiele nach aktueller policy π_θ generieren, bis die benötigte Anzahl für einen minibatch erreicht ist. Dann trainiert man das Netz mit dem minibatch $\mathcal{D}$ wie im letzten Abschnitt beschrieben. Anschließend werden alle Beispiele aus dem minibatch gelöscht und es wird von vorn begonnen. In dieser Arbeit wird Q-learning als Ansatz für das Lernen mit einem Neuronalen Netz benutzt. Das nennt man auch deep Q-learning. Würde deep Q-learning nach eben beschriebenen Prinzip umgesetzt werden, dann tauchen zwei grundlegende Probleme auf, welche ich in den Folgenden Abschnitten genauer erläutere.

17.1 Experience replay memory

Das erste Problem ist, dass die Trainingsbeispiele in einem minibatch zusammenhängend sind, wenn man eine Episode durchführt. D.h. es werden Übergänge benutzt, welche alle hintereinander in einer Episode stattfinden und sich somit relativ ähnlich sind. Dies ist sehr schlecht, wenn das Neuronale Netzwerk verallgemeinern soll, d.h. mit den Trainingsdaten auch die Q-values zu neuen Eingaben vorhersagen soll. Man sagt das Netz overfitted. Dies bedeutet, dass höchstens die richtigen Q-values zu den bekannten Eingaben aus dem Trainingsdaten auswendig gelernt werden, aber es kann nichts verallgemeinert werden kann.

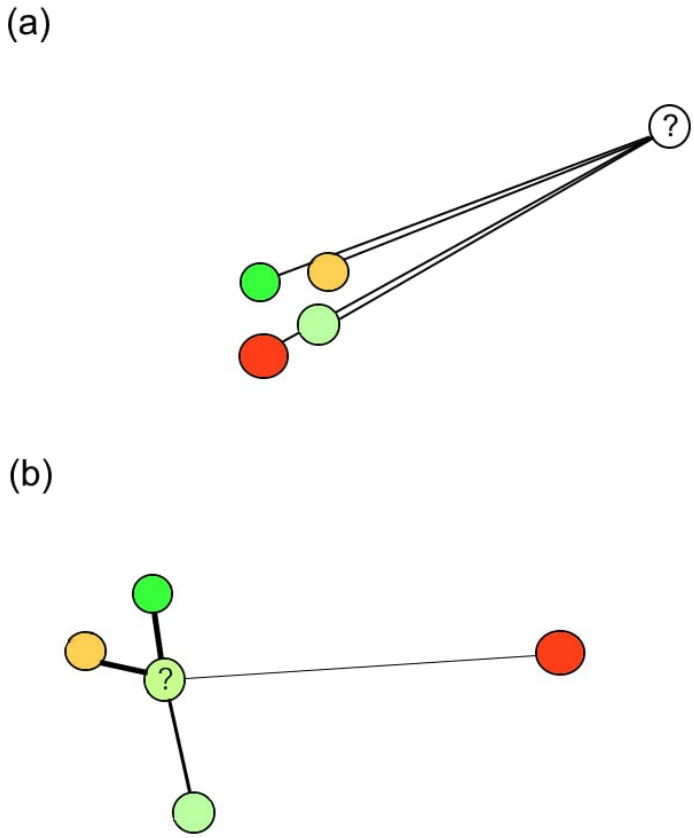

Abbildung 8: Schema zu Abschätzen neuer Beispiele (?-Kreise). Bekannte Beispiele haben hohe Wertigkeit (grün) oder niedrige Wertigkeit (rot). (a) Das Netzwerk overfitted. Es ist sehr schwer, aus ähnlichen Beispielen den Wert von neuen Beispielen zu berechnen. (b) Mit vielen verschiedenen Beispielen ist es einfacher, den Wert neuer Beispiele zu bestimmen.

Dieses Problem der ähnlichen Trainingsdaten kann relativ einfach durch einen experience replay memory (Abkürzung: ERM) gelöst werden. Der ERM ist ein sehr großer Speicher für viele Trainingsbeispiele. In ihm werden alle Übergänge gespeichert. Ist der Speicher voll, werden die ältesten Ergebnisse ersetzt. Nun wählt man in einer bestimmten Frequenz zufällige Einträge aus dem ERM aus, und formt daraus einen minbatch. Damit die Trainingsbeispiele wirklich unterschiedlich sind, ist es wichtig, dass die Kapazität des ERM sehr viel größer als die batch size ist.

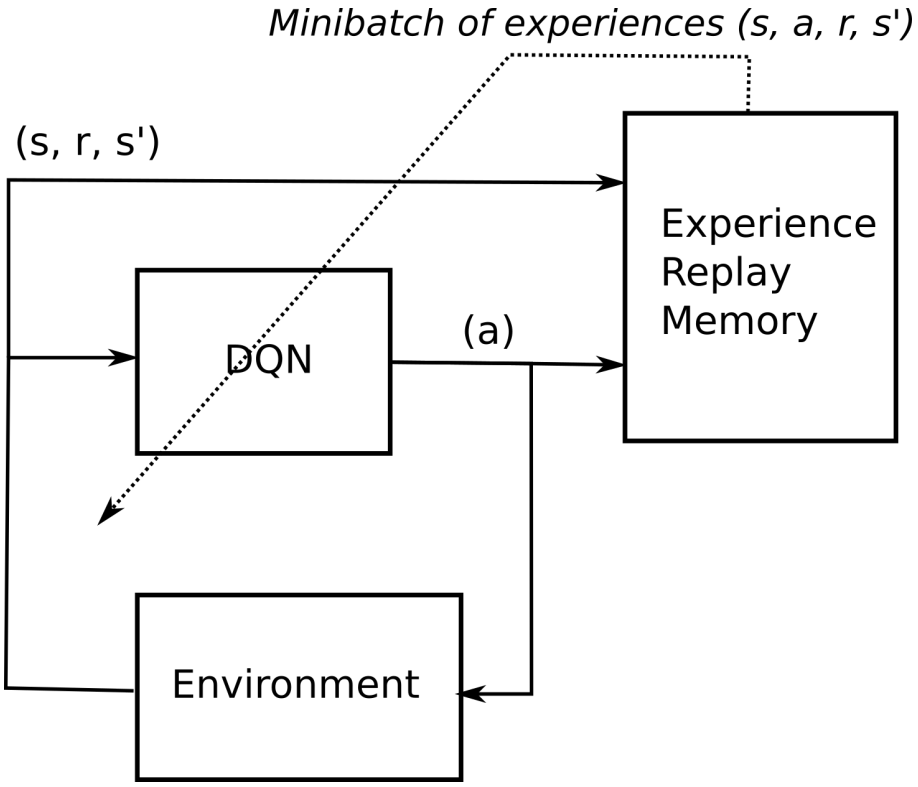

Abbildung 9: Funktionsweise des ERM. Abb. aus [2]

17.2 Target network

Das zweite Problem, welches beim deep Q-learning auftritt ist, dass der reale output beziehungsweise der Zielwert t in der loss function nicht konstant ist.

$$\mathcal{L}(\theta) = \mathbb{E}_{x,t\in\mathcal{D}}[\| t - y \|^2]$$

Für Q-learning ist $t = r(s,a,s') + \gamma \max_{a'} Q_\theta(s',a')$. Da $Q_\theta(s',a')$ von θ abhängt, verändert es sich während des Lernprozessen und somit auch t. Neuronale Netze konvergieren zwar gegen den Zielwert, aber wenn sich der Zielwert bewegt, kann es sein dass gar nicht gelernt wird. Dieses Problem wird mit einem target network gelöst. Dies ist eine ältere Version des Netzes, welches eigentlich lernt und besitzt die Parameter θ'. Das target network wird in einer bestimmten Frequenz aktualisiert ($\theta' \leftarrow \theta$). Das target network wird nur dazu benutzt, um den Zielwert $r(s,a,s') + \gamma \max_{a'} Q'_\theta(s',a')$ zu berechnen. Da es nur nach gewissen Zeitabständen aktualisiert wird, bleiben die Zielwerte, jeweils über bestimmte Intervalle lang gleich. Wird das target network nur relativ selten aktualisiert, reicht dies aus damit das lernende Netz gut lernen kann.

Teil III

Anwendung auf das Pfannkuchen-Sortierproblem

18 Environment

Das environment ist die Simulation des Pfannkuchen-Sortierproblems. In ihm sind alle Dynamiken festgelegt und auch welche rewards vergeben werden. Das Environment ist als Klasse dargestellt und besteht aus verschiedenen Funktionen.

```
import numpy as np

class Env:
    def __init__(self, number):
        self.state = list(range(1, number+1))
        np.random.shuffle(self.state)
        self.next_state = self.state
        self.final_state = list(range(number, 0, -1))
        self.action_space = list(range(number-1))

    def reset(self, number):
        self.state = list(range(1, number+1))
        np.random.shuffle(self.state)
        self.next_state = self.state
        return self.state

    def step(self, action):
        self.state = self.next_state
        down = self.state[:action]
        up = self.state[action:]
        up.reverse()
        self.next_state = down+up
        if self.next_state == self.final_state:
            self.reward = 1
            self.done = True
        else:
            self.reward = 0
            self.done = False
        return self.next_state, self.reward, self.done
```

In der init Funktion wird ein Start-state festgelegt. Dieser wird in Zeile 5 als List der Zahlen 1 bis number (=Anzahl der Pfannkuchen) definiert und in Zeile 6 werden die Zahlen in eine zufällige Reihenfolge gebracht. Jede Zahl steht hierbei für einen Pfannkuchen. Der final state ist der terminal state. Er ist eine Liste von number bis 1 rückwärts aufgelistet, da der größte Pfannkuchen unten liegen soll. Der action_space ist die Menge aller möglichen actions. Dies ist eine Liste von 0 bis number-2, wobei 0 die action ist alle Pfannkuchen umzudrehen, 1 die action ist die obersten n-1 Pfannkuchen umzudrehen und n-2 die action ist die obersten 2 Pfannkuchen umzudrehen. Die action n-1 wird nicht mit in den action_space aufgenommen, da nur der oberste Pfannkuchen umgedreht und somit keine Veränderung im state stattfinden würde. Dies würde den Lernprozess nur behindern.

Die Interaktion mit dem agent ist durch die Funktion step realisiert. Hierbei wird die action auf den aktuellen state angewandt. In den Zeilen 19 und 20 wird der Stapel in einen oberen und einen unteren Teilstapel zerteilt und in Zeile 21 die Reihenfolge des oberen Teilstapels umgekehrt. Dies ergibt nach Durchführung der action einen neuen state, den next state. Als nächstes wird noch in Zeile 23 bis 28 der reward berechnet.

$$r(s, a, s') = \begin{cases} 1 & \text{falls } s_T = s' \\ 0 & \text{sonst} \end{cases}$$

Zuletzt diese Funktion gibt den next state, den reward, und den Wert done, welcher angibt ob sich der agent nach dem Übergang in einem terminal state befindet, an den agent zurück.

Um nach jeder Episode einen zufälligen neuen Start-state zu bestimmen, wird die Funktion reset verwendet. Sie setzt den state auf eine zufällige Permutation der Liste von 1 bis number.

19 Grundprogramm

Das Grundprogramm und der agent, welcher als Klasse definiert ist, sind in einer Programmdatei zusammengefasst, während das environment in einer seperaten Programmdatei 'environment.py' gespeichert wird. Dies hat den Vorteil dass das environment und das Programm mit dem agent, theoretisch mit beliebigen anderen agents beziehungsweise environments kombiniert werden können, indem einfach nur eine andere Programmdatei implementiert wird. Dafür müssen das environment beziehungsweise der agent bestimmte Funktionen mit bestimmten standardisierten Namen enthalten. Die Funktionen init, step und reset im environment sind Beispiele für solche Funktionen. Weiterführend könnte man also verschiedene andere bereits erstellte agents aus dem Internet, nutzen um das Pfannkuchen-Sortierproblem zu lösen oder man könnte den im Rahmen dieser Arbeit geschriebenen agent benutzen,

um seine Leistung auf andere bereits bestehende environments aus dem Internet zu testen. Das wurde in dieser Arbeits jedoch nicht getan.

```
1  from environment import Env
2
3  n = int(input('number of pancakes: '))
4  k = int(input('number of episodes: '))
5
6  psp = Env(n)
7  agent = cAgent(psp.state, psp.action_space)
8
9  for episode in range(k):
10     state = psp.reset(n)
11     step_count = 0
12     while True:
13         step_count += 1
14         action = agent.get_action(state)
15         next_state, reward, done = psp.step(action)
16         agent.learn(state, action, reward, next_state)
17         state = next_state
18         if done or step_count == 10000000:
19             break
20     print(f'episode: {episode} steps: {step_count}')
```

Da, wie bereits erklärt, das environment in einer anderen Programmdatei gespeichert wird, muss man diese erst importieren. Dann kann der Nutzer die Anzahl an Pfannkuchen n, mit denen gerechnet werden soll und die Anzahl an Episoden k, die durchgeführt werden sollen, eingeben. In Zeile 6 wird noch das environment und in Zeile 7 der agent initialisiert.

In Zeile 9 startet eine Schleife, welche für jede Episode einmal durchgeführt wird. In Zeile 10 wird zuerst in jeder Episode ein Start-state festgelegt. Die Variable step count repräsentiert den Zeitschritt t und wird zu Anfang der Episode auf 0 gesetzt.

Die Schleife von Zeile 12 bis 19 wird jeden einzelnen Zeitschritt wiederholt. Es wird zuerst der aktuelle Zeitschritt um 1 erhöht. In Zeile 14 wählt der agent in Abhängigkeit von aktuellen state eine action nach seiner aktuellen policy aus. Diese action wird vom environment verarbeitet und es gibt dem agent seinen neuen state, den erhaltenen reward und die Information, ob er sich in einem terminal state befindet, zurück. Der agent benutzt diesen Übergang s, a, r, s', um seine Q-values upzudaten. Dies geschieht in Zeile 16 in der learn Funktion. Zum Schluss wird noch in Zeile 17 der aktuelle state in dem sich der agent nach dem Übergang befindet aktualisiert. Die Episode wird beendet, wenn sich der agent in einem terminal state befindet, oder wenn er 10 Millionen Schritte gemacht hat. Diese zweite Abbruchbe-

dingung ist praktisch, da sie verhindert, dass sich der agent besonders in den ersten paar Episoden bei größerem n extrem lange in irgendwelchen Schleifen verfängt und gar nichts lernt. Stattdessen wird einfach eine neue Episode gestartet.

20 Agent

Der agent wird als Klasse definiert und besteht aus drei grundlegenden Funktionen. Der init Funktion, um Parameter und die Tabelle mit Q-values zu initialisieren, der learn Funktion, die mit einem Trainingsbeispiel die Q-values updatet und der get_action Funktion, welche basierend auf der aktuellen Tabelle mit Q-values eine eine action in Abängigkeit von einem state auswählt.

```
1  import numpy as np
2  from itertools import permutations
3
4  class cAgent:
5      def __init__(self, init_state, actions):
6          self.i_state = init_state
7          self.actions = actions
8          self.learning_rate = 0.01
9          self.discount_factor = 0.9
10         self.state_index = dict()
11         k = 0
12         for i in list(permutations(self.i_state, len(self.
               i_state))):
13             self.state_index[i] = k
14             k += 1
15         self.q_table = np.zeros((len(self.state_index), len(
               self.actions)))
16
17     def learn(self, state, action, reward, next_state):
18                 ...
19     def get_action(self, state):
20                 ...
21         return take_action
```

In der init Funktion werden die Parameter α und γ initialisiert. Weiterhin wird in Zeile 16 ein numpy-array initialisiert, um alle Q-values zu speichern. Dieses hat so viele Zeilen, wie es states gibt, und so viele Spalten wie es actions gibt. Da für numpy arrays nur integer Zahlen möglich sind um auf eine Zeile zuzugreifen, aber die states listen sind, wird noch ein dictionary verwendet, welches jedem state eine Zahl von 0 bis Anzahl der states -1 zuweist. Dieses wird in den Zeilen 13 bis 15 initialisiert. Die

Speicherung der Q-values in numpy arrays ist sehr wichtig, da andere Methoden, wie z.B. das Nutzen von dictionaries viel langsamer sind. Dies zeigt sich stärker bei höherem Rechenaufwand, weshalb ein Vergleich im Abschnitt 21.1 gemacht wird.

20.1 Q-learning

In der learn Funktion passt der agent die Q-values, basierend auf einem Trainingsbeispiel an.

```
1 def learn(self, state, action, reward, next_state):
2     current_q = self.q_table[self.state_index[tuple(state)],
          action]
3     new_q = reward + self.discount_factor * np.max(self.
          q_table[self.state_index[tuple(next_state)]])
4     self.q_table[self.state_index[tuple(state)], action] +=
          self.learning_rate * (new_q - current_q)
```

Beim Q-learning wird in Zeile 3 der TD error nach Gleichung 5 berechnet. Anschließend werden in Zeile 4 die Q-values wie in Gleichung 3 aktualisiert (natürlich mit dem auf Q-learning angepassten TD error).

20.2 SARSA

Die learn Funktion für den SARSA Algortihmus ist ähnlich wie die, des Q-learning Algorithmus aufgebaut.

```
1 def learn(self, state, action, reward, next_state):
2     current_q = self.q_table[self.state_index[tuple(state)],
          action]
3     new_q = reward + self.discount_factor * self.q_table[
          self.state_index[tuple(next_state)], self.get_action(
          next_state)]
4     self.q_table[self.state_index[tuple(state)], action] +=
          self.learning_rate * (new_q - current_q)
```

Bei SARSA wird in Zeile 3 der TD error nach Gleichung 4 berechnet. Anschließend werden in Zeile 4 die Q-values wie in Gleichung 3 aktualisiert (hier natürlich auch wieder mit dem auf SARSA angepassten TD error).

20.3 ϵ-greedy Algorithmus

Für die action Auswahl kann man einen ϵ-greedy Algorithmus verwenden. Dies wird in der Funktion get_action umgesetzt. Zusätzlich zu folgendem Code muss man noch den Parameter ϵ in der init Funktion des agent definieren.

```
1  def get_action(self, state):
2      if np.random.rand() < self.epsilon:
3          action_a = np.random.choice(self.actions)
4          action_b = np.random.choice(self.actions)
5          while action_a == action_b:
6              action_b = np.random.choice(self.actions)
7          if self.q_table[self.state_index[tuple(state)],
               action_a] <= self.q_table[self.state_index[tuple(
               state)], action_b]:
8              take_action = action_a
9          else:
10             take_action = action_b
11     else:
12         max_q = np.max(self.q_table[self.state_index[tuple(
               state)]])
13         best_actions = list()
14         for i in self.actions:
15             if self.q_table[self.state_index[tuple(state)],
                   self.actions[i]] == max_q:
16                 best_actions.append(self.actions[i])
17         take_action = np.random.choice(best_actions)
18     return take_action
```

Zuerst wird in Zeile 2 mit einer Zufallszahl zwischen 0 und 1 ermittelt, ob eine zufällige non-greedy action oder die greedy action gewählt werden soll. Von Zeile 3 bis 10 wird mit Wahrscheinlichkeit ϵ eine zufällige non-greedy action ausgewählt. Dafür werden zufällig zwei actions a und b ausgewählt. In Zeile 5 und 6 wird sichergestellt, dass wirklich zwei verschiedene actions gewählt werden. In den Zeilen 7 bis 10 wird dann die action ausgewählt, welche den kleineren Q-values hat. Sind die Q-values gleich wird action a ausgewählt. Sollte zufälligerweise eine der actions a oder b die greedy action gewesen sein, dann wird auf jeden Fall eine action mit einem kleineren Q-value gewählt. Gibt es mehrere greedy actions, also actions die alle den gleichen Q-value haben und dieser Q-value der maximale von allen Q-values von state-action pairs im aktuellen state ist, dann ist es trotzdem zulässig wenn eine dieser actions ausgewählt wird. Sollten zum Beispiel alle actions gleiche Q-values haben, dann sind sie alle greedy actions und es wird zufällig eine action ausgewählt. In Zeile 12 bis 17 wird mit der Wahrscheinlichkeit $1 - \epsilon$ eine greedy action ausgewählt. Dabei wird zuerst der maximale Q-value aller state-action pairs im aktuellen state ermittelt. Dann werden in den Zeilen 14 bis 16 alle actions gespeichert, die diesen maximalen Q-value besitzen und in Zeile 17 wird eine dieser greedy actions ausgewählt.

20.4 Softmax Algorithmus

Anstelle eines ϵ-greedy Algorithmus kann man in der get_action Funktion auch einen softmax Algorithmus verwenden. Auch bei softmax muss man zusätzlich zu folgendem Code den Parameter τ in der init Funktion des agent definieren.

```
1  def get_action(self, state):
2      q_val = self.q_table[self.state_index[tuple(state)]]
3      q_val = np.exp((q_val - np.max(q_val)) / self.tau) / np.
           sum(np.exp((q_val - np.max(q_val)) / self.tau), axis
           =0)
4      prob = np.random.rand()
5      q_val_sum = 0
6      count = 0
7      for i in q_val:
8          q_val_sum += i
9          if prob <= q_val_sum:
10             take_action = count
11             break
12         count += 1
13     return take_action
```

Beim softmax Algorithmus werden zuerst alle Q-values nach Gleichung 1 in Zeile 3 in Wahrscheinlichkeiten umgewandelt. Diese Wahrscheinlichkeiten ergeben in ihrer Summe 1. Mit der Zufallszahl prob zwischen 0 und 1 wird eine action ausgewählt. Dazu werden in den Zeilen 7 bis 12 die Wahrscheinlichkeiten hintereinender addiert, sodass sie das Intervall [0,1] vollständig abdecken. Es wird diejenige action gewählt, in deren Bereich im Intervall [0,1] die Zahl prob liegt. Dazu werden nacheinander die Wahrscheinlichkeiten addiert, bis die Summe der Wahrscheinlichkeiten größer oder gleich prob ist.

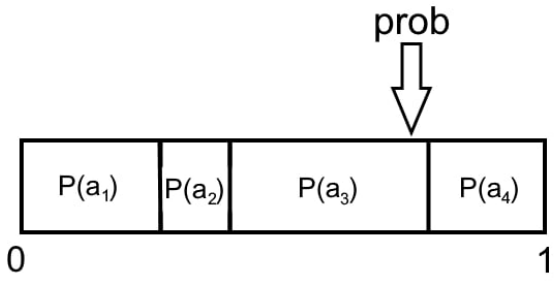

Abbildung 10: Auswahl aus vier verschiedenen actions beim softmax Algorithmus

21 Eligibility traces

Die eligibility traces werden als numpy array genau so wie die Q-values in der init Funktion initialisiert. Zusätzlich muss auch noch der Parameter λ in der init Funktion definiert werden. Die Berechnung der Q-values in der learn Funktion wird durch Implementierung der eligibility traces verändert.

```
1 def learn(self, state, action, reward, next_state):
2         current_q = self.q_table[self.state_index[tuple(
              state)], action]
3     new_q = reward + self.discount_factor * np.max(self.
          q_table[self.state_index[tuple(next_state)]])
4     self.e_table *= self.decaying_factor
5     self.e_table[self.state_index[tuple(state)], action] /=
          self.decaying_factor
6     self.e_table[self.state_index[tuple(state)], action] +=
          1
7     self.q_table[self.state_index[tuple(state)], action] +=
          self.learning_rate * (new_q - current_q) * self.
          e_table[self.state_index[tuple(state)], action]
```

Wieder wird zuerst der TD error berechnet. Hier in diesem Beispiel wurde Q-learning verwendet. In den Zeilen 4 bis 6 wird die Tabelle mit den eligibility traces aktualisiert. Gemäß Gleichung 6 wird für die gewählte action im letzten state 1 addiert und alle anderen Werte werden mit λ multipliziert. In Zeile 7 werden die Q-values gemäß Gleichung 7 aktualisiert. Es ist möglich, vor Beginn jeder Episode die eligibility traces zurück auf 0 zu setzen. Dies ist jedoch nicht notwendig. Nach Vergleich beider Möglichkeiten zeigte sich, dass beide Methoden beim Pfannkuchen-Sortierproblem nahezu gleiche Ergebnisse liefern. Es wird bei der Auswertung und dem Vergleich von Methoden nur die Variante benutzt, bei der die eligibility traces vor jeder Episode auf 0 gesetzt werden.

21.1 Vergleich dictionaries - numpy arrays

Im Rahmen dieser Arbeit wurden verschiedene Möglichkeiten für die Implementierung von tabellarischen Methoden ausprobiert. Bisher wurden alle Tabellen durch numpy arrays umgesetzt. Dies ist auch die schnellere Methode. Eine andere Möglichkeit ist die Darstellung von Tabellen mit dictionaries. Eine Tabelle wird durch die Verkettung zweier dictionaries realisiert.

```
1 self.q_table = defaultdict(dict)
2 for i in list(permutations(self.i_state, len(self.i_state)))
      :
```

```
    for k in self.actions:
        self.q_table[i][k] = 0.0
```

Auf diese Weise würde die Tabelle mit Q-values und auch die eligibility traces initialisiert werden. Bei eligibility traces werden in jeden Zeitschritt alle Q-values aktualisiert. Dies kann mit dictionaries nur über for-Schleifen geschehen, welche viel langsamer sind, als die Operationen mit numpy arrays. Dies liegt daran, dass die numpy Bibliothek in C geschrieben ist.

Für einen Vergleich wird zweimal der gleiche Algorithmus (Q-learning, ϵ-greedy, mit eligibility traces) mit identischen Parametern ($\alpha = 0,01$, $\gamma = 0,9$, $\epsilon = 0,1$, $\lambda = 0,9$) benutzt. Es werden 500 Episoden mit 6 Pfannkuchen einmal mit numpy arrays und einmal mit dictionaries berechnet. Die Umsetzung mit dictionaries hat im Durchschnitt 110,12 s benötigt, während die Umsetzung mit numpy arrays durchschnittlich 0,68 s gebraucht hat

22 Reward scheduling

Beim reward scheduling wird nur ein leicht verändertes environemt benutzt. In folgender Version wird immer am Ende der Episode ein reward von 100 gegeben und für alle anderen actions ein reward von -1.

```
if self.next_state == self.final_state:
    self.reward = 100
    self.done = True
else:
    self.reward = -1
    self.done = False
```

Die rewards werden als entsprechend folgender Regel vergeben.

$$r(s,a,s') = \begin{cases} 100 & \text{falls } s_T = s' \\ -1 & \text{sonst} \end{cases}$$

23 Variables ϵ

Wie bereits in Kapitel 12 erwähnt, ist es möglich den Parameter ϵ bei einem ϵ-greedy Algorithmus nicht konstant zu wählen, sondern ihn erst etwas höher als gewöhnlich zu initialisieren und ihn dann über viele Episoden hinweg langsam zu verringern. Dies führt dazu, dass am Anfang sehr viel exploration stattfindet, dies jedoch mit jeder Episode reduziert wird. Trotz der hohen exploration am Anfang hat man nach einiger Zeit einen stabilen Algorithmus, bei dem viel exploitation stattfindet. In jeder Episode wird folgende Berechnung durchgeführt.

```
agent.epsilon *= 0.995
if agent.epsilon < 0.01:
    agent.epsilon = 0.01
```

Damit die exploration nicht komplett verschwindet, wurde für das ϵ eine untere Grenze von 0,01 festgelegt. In diesem Beispiel wurde ϵ mit dem Wert 0,25 initialisiert.

24 Deep Q-learning

Für den Umgang mit Neuronalen Netzen wird in dieser Arbeit die Bibliothek Keras benutzt. Zuerst wird die separate Funktion build_model, welche nicht in der agent Klasse enthalten ist, benötigt, um ein Neuronales Netz zu modellieren.

```
def build_model(p_num):
    model = keras.models.Sequential()
    model.add(keras.layers.Dense(6, input_dim=p_num*p_num,
        activation='relu'))
    model.add(keras.layers.Dense(6, activation='relu'))
    model.add(keras.layers.Dense(p_num-1, activation='relu')
        )
    model.compile(optimizer='adam', loss='mse')
    return model
```

Das Sequential model ist ein neuronales Netz, welches aus mehreren layers besteht, die hintereinander agieren. Es funktioniert also so, wie in Kapitel 16 beschrieben. Nun kann man dem model einzelne layers hinzufügen. Der Dense layer ist ein einfach fully-connected layer. Jedem layer muss man noch eine Größe (d.h. Anzahl der Neuronen) und eine activation function zuweisen. Die hier verwendete activation function relu funktioniert mit Standardparametern wie folgt.

$$f(x) = max(x, 0)$$

Die Anzahl der Pfannkuchen wird als p_num bezeichnet. Im ersten layer muss man noch die Größe des inputs festlegen. Hier ist der input ein zweidimensionales Array der Größe p_num*p_num. Der letzte layer ist gleichzeitig der output layer. Dieser hat eine Größe von p_num−1, da es in jedem state p_num−1 actions gibt, für welche die Q-values gelernt werden sollen. Zum Schluss muss das model noch kompiliert werden, um es zu verwenden. Dabei legt man einen optimizer und die Art der verwendeten loss function fest. Der optimizer adam ist ein sehr häufig verwendeter optimizer für Probleme ähnlich dem Pfannkuchen-Sortierproblem.

Weiterhin wird die Funktion input_state definiert. Diese ist dazu da, die states zu transformieren. Man könnte auch einfach dem neuronalen Netz den state in sei-

ner bisher verwendeten Form übergeben. Beispielsweise würde bei 3 Pfannkuchen der state [3,1,2] übergeben. Die verschiedenen Pfannkuchen stehen einfach nur an unterschiedlichen Stellen, haben aber keine unterschiedliche Wertigkeit. Da bei einer Eingabe von [3,1,2] mit den Zahlen 3, 1 und 2 gerechnet wird, wird den verschiedenen Pfannkuchen indirekt eine unterschiedliche Wertigkeit zugewiesen. Dies kann den Lernprozess behindern. Alternativ formt man die states mit folgender Methode um.

```
1 def input_state(state):
2     matrix_state = np.zeros((len(state), len(state)), dtype=
          int)
3     for i in range(len(state)):
4         matrix_state[i, state[i]-1] = 1
5     state_od = matrix_state.flatten()
6     return state_od
```

In Zeile 2 wird eine Tabelle mit 3 Zeilen und 3 Spalten erstellt (für den state [3,1,2]). Die Zeilen stehen dabei für die Position und die Spalten für die Zahlen. Dann wird in Zeile 3 und 4 an den Stellen an denen die Zahlen in die richtigen Position stehen eine 1 eingetragen. Die restlichen Felder bleiben 0. Im Beispiel [3,1,2] sähe das wie folgt aus.

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

In Zeile 4 wird der state noch in ein einziges array umgewendelt. Im Beispiel wäre das Endergebnis also [0,0,1,1,0,0,0,1,0]. Jetzt haben alle Pfannkuchen die gleiche Wertigkeit, da sie sich nur durch ihre Position unterscheiden.

Nun kann mit dem eigentlichen Programm begonnen werden. Das Grundprogramm unterscheidet sich nur etwas von dem der tabellarischen Methoden. Es gibt nur einen Unterschied in dem Teil, was in jedem step passiert.

```
1 step_count += 1
2 action = agent.get_action(state)
3 next_state, reward, done = psp.step(action)
4 agent.buffer.append(state, action, reward, next_state, done)
5 if agent.batch_size <= len(agent.buffer.states) and agent.
      times_act % agent.train_period == 0:
6     minibatch = agent.buffer.sample(agent.batch_size)
7     agent.train(minibatch)
8 agent.update_target()
9 state = next_state
```

Genau wie bei tabellarischen Methoden wird der step count um 1 erhöht und der

agent wählt mit seiner aktuellen policy eine action in Abhängigkeit von state aus. Diese wird ans environment übergeben und der agent erhält seinen nächsten state, den reward und die Information ob er sich in einem terminal state befindet zurück. Nun wird jedoch nicht direkt gelernt, sondern der Übergang erstmal im experience replay memory gespeichert. Mit einer bestimmten Frequenz, welche in Zeile 5 berechnet wird, wird ein minibatch aus dem ERM gewählt und der agent mit diesem minibatch trainiert. Anschließend wird noch das target network aktualisiert. Dies geschieht auch mit einer bestimmten Frequenz, welche aber in der Funktion update_target festgelegt ist. Weiterhin wird in Zeile 5 noch bestimmt, dass er agent frühestens anfängt zu lernen, wenn genug Übergänge für einen minibatch im ERM gespeichert sind. Die Phase, während der agent am Anfang noch gar nicht lernt wird warm-up phase genannt.

Der agent selbst ist wieder als separate Klasse definiert.

```
1  class cAgent:
2      def __init__(self, actions, p_num):
3          self.action_space = actions
4          self.discount_factor = 0.9
5          self.epsilon = 0.1
6          self.times_act = 0
7
8          erm_capacity = 25000
9          self.buffer = ExperienceReplayMemory(erm_capacity)
10         self.batch_size = 32
11
12         self.model = build_model(p_num)
13         self.target_model = build_model(p_num)
14         self.target_model.set_weights(self.model.get_weights
               ())
15         self.train_period = 5
16         self.update_target_period = 1000
17
18     def train(self, batch):
19         states, actions, rewards, next_states, dones = batch
20
21         next_q_values = np.max(np.array(self.target_model.
               predict_on_batch(next_states)), axis=1)
22         for i in range(len(dones)):
23             if dones[i]:
24                 next_q_values[i] = 0.0
25
```

```
26          targets = np.array(self.model.predict_on_batch(
                states))
27          for i in range(self.batch_size):
28              targets[i, actions[i]] = rewards[i] + self.
                    discount_factor * next_q_values[i]
29
30          self.model.fit(states, targets, batch_size=self.
                batch_size, verbose=0)
31
32      def update_target(self):
33          if self.times_act % self.update_target_period == 0:
34              self.target_model.set_weights(self.model.
                    get_weights())
35
36      def get_action(self, state):
37          ...
38          return take_action
```

Die get_action Funktion besteht genau wie bei den anderen Methoden aus einem ϵ-greedy oder softmax Algorithmus. Weiterhin besitzt der agent noch eine init, eine train und eine update_target Funktion.

In der init Funktion werden wichtige Parameter wie γ und ϵ (falls ein ϵ-greedy Algorithmus verwendet wird) initialisiert. So auch die train period und die update_target_period, welche angeben alle wie viel Zeitschritte das Netz trainiert beziehungsweise das target network aktualisiert werden soll. In Zeile 9 wird ein ERM mit einer Kapazität von 25000 initialisiert. Dies geschieht durch eine extra Klasse ExperienceReplayMemory. Die batch size von minibatches wird aus 32 festgelegt. In den Zeilen 12 bis 14 wird ein neuronales Netzwerk und das target network mit den gleichen Gewichten initialisiert. Dazu wird die Funktion build_model benutzt in der schon genauer beschrieben wurde, wie die Netze aussehen.

In der update_target Funktion werden nur die Gewichte des lernenden Netzes auf das target network übertragen. Dies geschieht mit einer bestimmten Frequenz.

In der train Funktion werde in Zeile 21 die Q-values der greedy actions im nächsten state vorhergesagt. In den Zeilen 22 bis 24 wird festgelegt, dass terminal state einen Wert von 0 bekommen. In den Zeilen 26 bis 28 werden die Zielwerte berechnet, so wie sie auch in Gleichung 8 benötigt werden. Für das eigentliche Trainieren des Netzes benutzt man in Keras die Funktion fit. Dieser muss man nur noch die Zielwerte und den aktuellen state übergeben und es wird automatisch ein backpropagation Algorithmus durchgeführt. Die verwendete loss function und der optimizer wurden ja schon in der Funktion build_model festelegt.

Zuletzt muss noch die Klasse ExperienceReplayMemory definiert werden.

```
1  class ExperienceReplayMemory:
2      def __init__(self, max_capacity):
3          self.states = deque(maxlen=max_capacity)
4          self.actions = deque(maxlen=max_capacity)
5          self.rewards = deque(maxlen=max_capacity)
6          self.next_states = deque(maxlen=max_capacity)
7          self.dones = deque(maxlen=max_capacity)
8
9      def append(self, state, action, reward, next_state, done
           ):
10         self.states.append(input_state(state))
11         self.actions.append(action)
12         self.rewards.append(reward)
13         self.next_states.append(input_state(next_state))
14         self.dones.append(done)
15
16     def sample(self, batch_size):
17         indices = sorted(np.random.choice(np.arange(len(self
               .states)), batch_size, replace=False))
18         return [np.array([self.states[i] for i in indices]),
19                 np.array([self.actions[i] for i in indices])
                       ,
20                 np.array([self.rewards[i] for i in indices])
                       ,
21                 np.array([self.next_states[i] for i in
                       indices]),
22                 np.array([self.dones[i] for i in indices])]
```

Hierbei wird ein deque aus der Bibliothek collections benutzt. Ein deque ist ein array mit einer maximalen Kapazität. Ist das deque voll und wird ein neues Objekt hinzugefügt, dann wird automatisch das Objekt ersetzt, welches schon am längsten im deque gespeichert ist. Dies ist genau die Vorgehensweise beim ERM. Der ERM im Programm besteht aus fünf einzelnen deques für die einzelnen Variablen. Diese deques sind in der init Funktion initialisiert. Man könnte auch alle Variablen in ein einziges deque speichern, aber dies wäre weitaus unübersichtlicher. In der append Funktion wird einfach ein neuer Übergang in den deques gespeichert. In Zeile 17 werden zufällig Zahlen aus dem Bereich 0 bis Anzahl der gespeicherten Übergange ausgewählt. Dabei werden genau so viele Zahlen ausgewählt, wie es die batch_size für einen minibatch vorschreibt. Es können keine Zahlen doppelt ausgewählt werden. In Zeile 18 bis 22 werden die zu den gewählten Indices passenden Übergänge aus

dem ERM kopiert und in einem minibatch zusammengefasst. Dieser kann einfach zu train Funktion des agent übertragen werden.

Teil IV

Auswertung

25 Vier Grundmethoden

Zuerst ist es nützlich die vier Grundmethoden miteinander zu vergleichen. Diese sind die tabellarischen Varianten von Q-learning mit ϵ-greedy Algorithmus, Q-learning mit softmax Algorithmus, SARSA mit ϵ-greedy Algorithmus und SARSA mit softmax Algorithmus.

25.1 Parameter

Wie gut eine Methode funktioniert ist von Problem zu Problem unterschiedlich. Die Leistung hängt jedoch auch von den gewählten Parametern ab. In der theoretischen Betrachtung wurde bereits angedeutet, wie die Parameter für die beste Funktionsweise ungefähr liegen sollen. Beispielsweise sollte γ nah bei 1 liegen um einen weitsichtigeren Algorithmus zu bekommen, der auch mögliche rewards in viel späteren Zeitschritten mitberechnet, als wenn γ sehr klein ist. Es gibt einige häufig genutzt Standardwerte, welche bei den meisten Problemen am besten funktionieren. Beim Beispiel des Q-learnings mit ϵ-greedy Algorithmus gibt es die Parameter γ, α und ϵ. Standardwerte wären $\gamma = 0,9$, $\alpha = 0,01$ und $\epsilon = 0,1$. In den folgenden Versuchen werden die Parameter, welche verwendet wurden, aber auch immer mit angegeben. Nach einigen Tests, bei denen Parameter variiert wurden, stellte sich heraus, dass die Standardparameter circa 10% von den abgeschätzen optimalen Parametern abweichen. Daher sind diese Standardparameter beim Pfannkuchen-Sortierproblem gut genug geeignet um verschiedene Methoden realistisch vergleichen zu können. Daher werden in den folgenden Auswertungen diese Parameter verwendet.

25.2 Vergleich

Für den Vergleich werden für alle Algorithmen die gleichen Parameter $\gamma = 0,9$, $\alpha = 0,01$, $\tau = 0,000001$ und $\epsilon = 0,1$ gewählt. Es wird eine Problemgröße von 7 Pfannkuchen bearbeitet. Dies ist für eine Differenzierung völlig ausreichend. Es werden immer 200 Episoden durchgeführt.

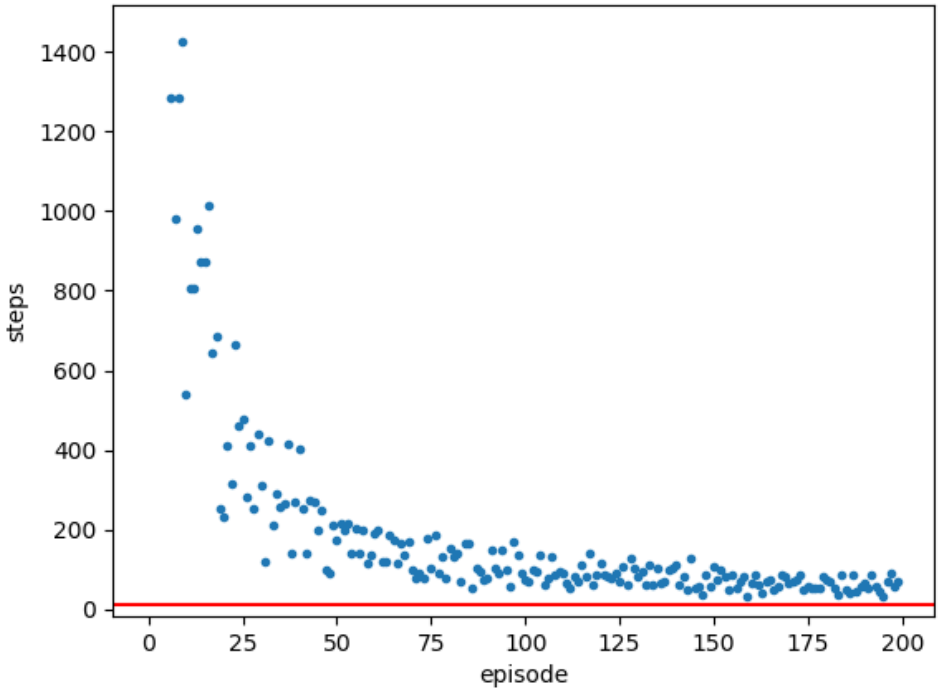

Abbildung 11: Q-learning mit ϵ-greedy Algorithmus

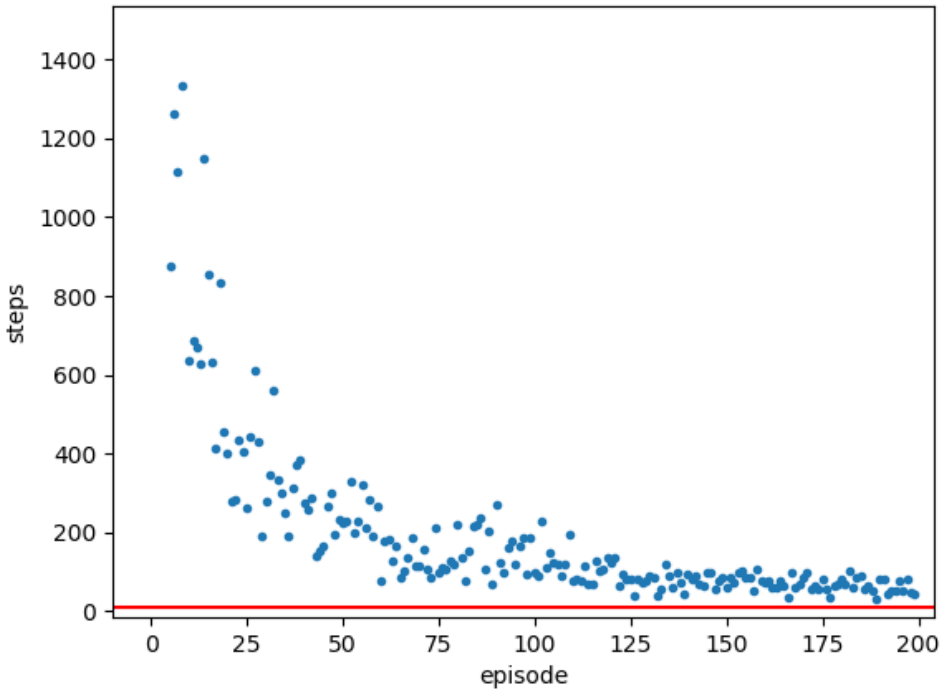

Abbildung 12: SARSA mit ϵ-greedy Algorithmus

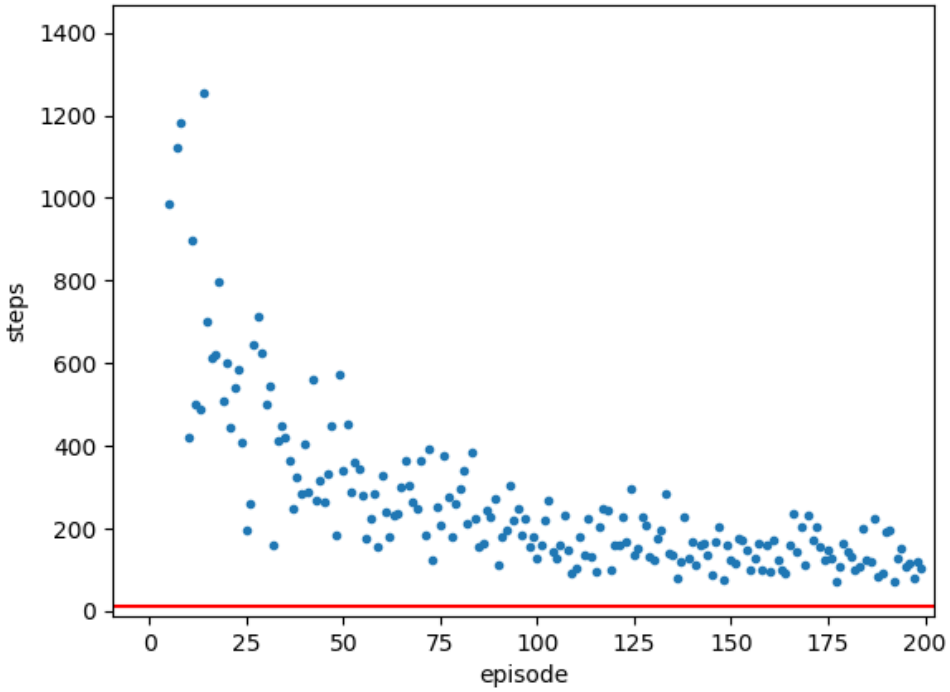

Abbildung 13: Q-learning mit softmax Algorithmus

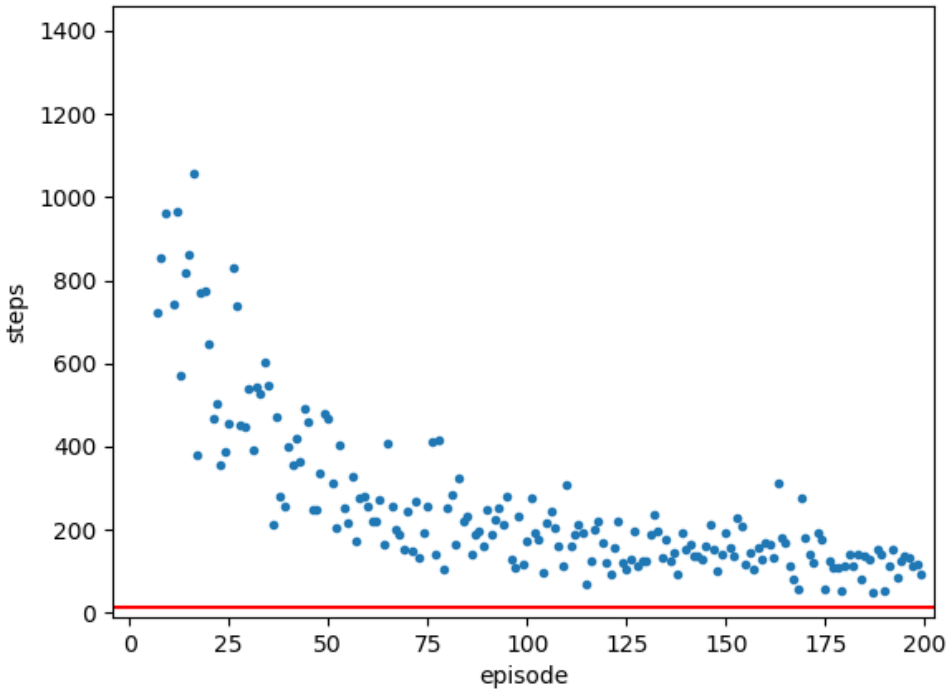

Abbildung 14: SARSA mit softmax Algorithmus

Diese Diagramme wurden mit der Bibliothek matplotlib für Python erstellt. Die rote Linie liegt immer bei $\frac{18}{11}n$, also bei der bisher besten oberen Schranke

für die Pfannkuchenzahlen. Es gibt fast keinen Unterschied zwischen Q-learning und SARSA. Beide Algorithmen zeigen beim Pfannkuchen-Sortierproblem in etwa die gleiche Leistung. Weiterhin ist zu bemerken, dass der softmax Algorithmus beim Pfannkuchen-Sortierproblem etwas langsamer als der ϵ-greedy Algorithmus lernt, obwohl schon ein sehr kleines τ gewählt wurde. Um herauszufinden welcher der beiden Algorithmen über längere Zeit hinweg besser lernt, wurde einmal ein ϵ-greedy Algorithmus und einmal ein softmax Algorithmus über 10200 Episoden mit 7 Pfannkuchen trainiert. Für beide Algorithmen wurde Q-learning verwendet. Es wurde $\epsilon = 0.2$ und $\tau = 0.0001$ gesetzt, um viel exploration zu haben. Um die Leistung der beiden trainierten Algorithmen zu vergleichen, wurde ab Episode 10000 keine exploration mehr durchgeführt. Dafür wurde das ϵ auf 0 und das τ auf die kleinste positive in Python verfügbare Gleitkommazahl gesetzt. Diese kann mit der Bibliothek sys durch sys.float_ info.min bestimmt werden.

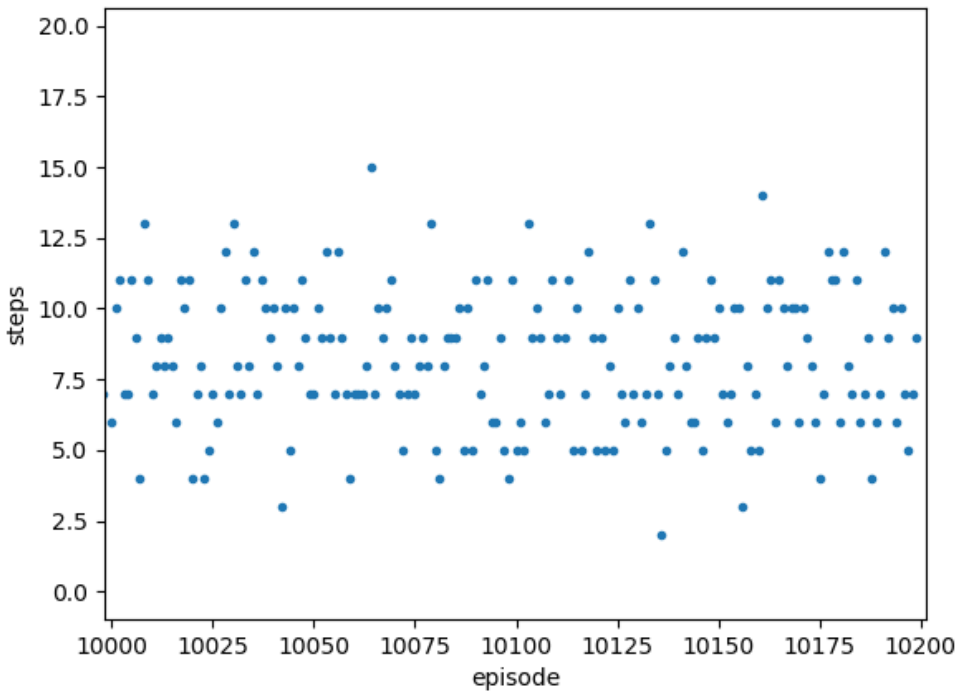

Abbildung 15: Q-learning mit ϵ-greedy Algorithmus

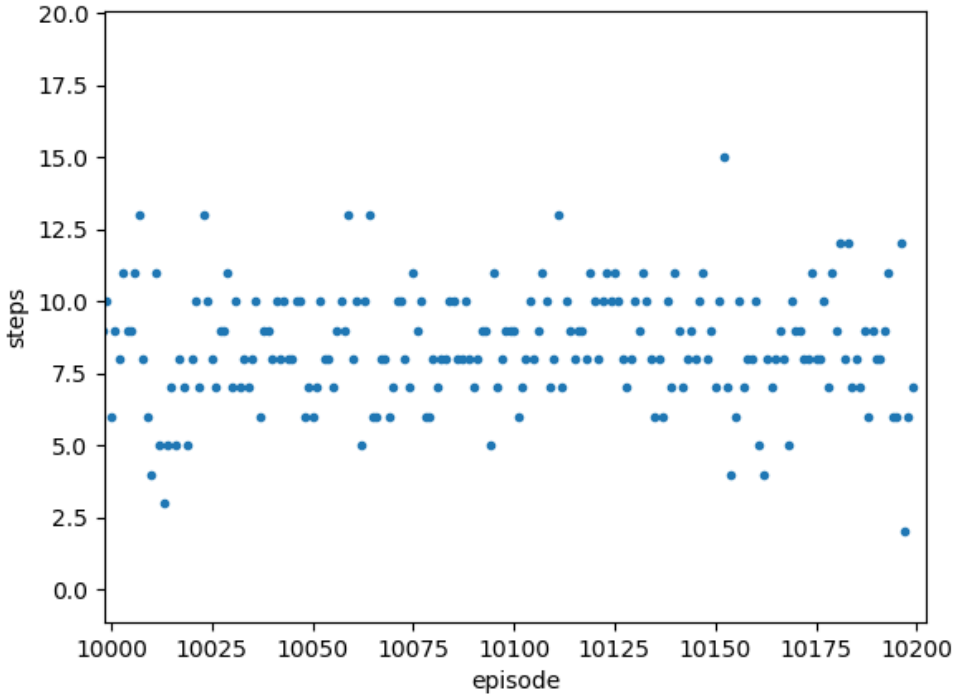

Abbildung 16: Q-learning mit softmax Algorithmus

Man sieht, dass es fast keinen Unterschied mehr zwischen ϵ-greedy und softmax Algorithmus gibt. Der softmax Algorithmus lernte mit den verwendeten Parametern sogar etwas besser als der ϵ-greedy Algorithmus.

26 Eligibility traces

Für den Vergleich von Methoden mit eligibility traces mit Methoden ohne, wurde zweimal ein Q-learning ϵ-greedy Algorithmus gewählt, einmal ohne und einmal mit eligibility traces. Es werden die Standardparameter $\gamma = 0,9$, $\alpha = 0,01$, $\epsilon = 0,1$ und $\lambda = 0,9$ benutzt. Es werden 500 Episoden mit 7 Pfannkuchen berechnet.

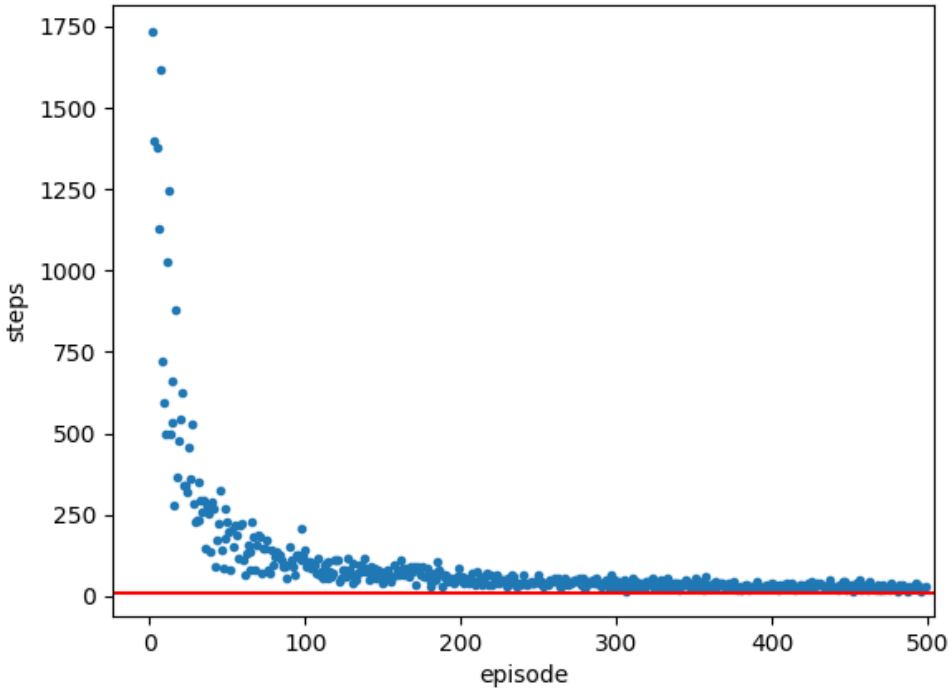

Abbildung 17: ohne eligibility traces

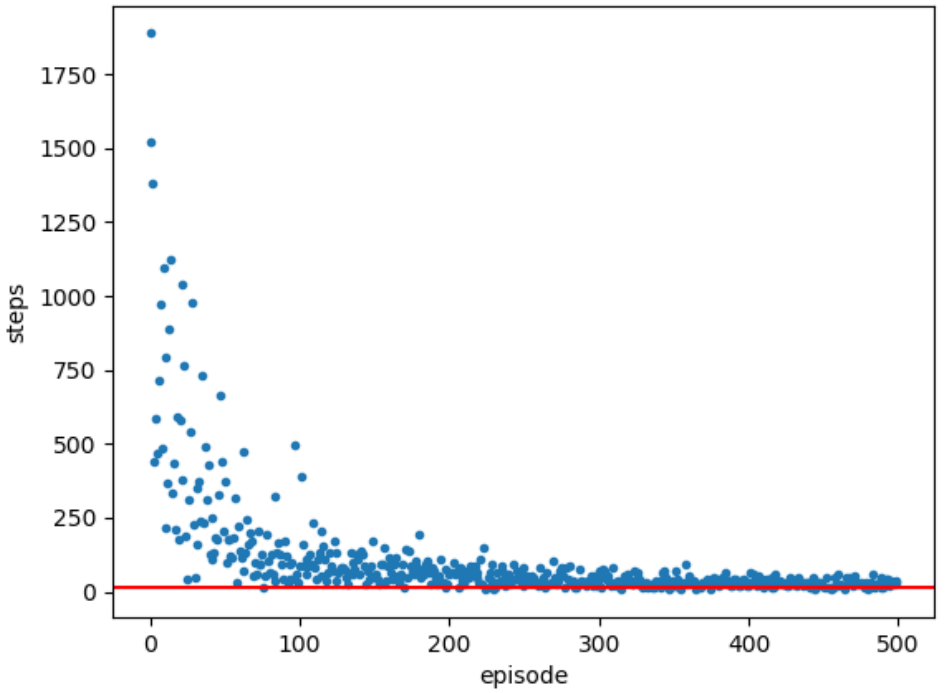

Abbildung 18: mit eligibility traces

Wie man sieht, bieten eligibility traces keine sehr große Verbesserung. Es ist Anzumerken, dass besonders in den frühen Episoden, mit eligibility traces öfters

sehr niedrige Werte erreicht werden. Dies bedeutet, dass eligibility traces schneller eine gute policy finden. Dafür ist die Streuung der Werte größer, d.h. es treten auch mehr höhere Werte auf. In späteren Episoden gibt es keine Unterscheidung mehr zwischen den beiden Methoden.

27 Reward scheduling

Um reward scheduling auszutesten, wurde wieder ein Q-learning ϵ-greedy Algorithmus mit den Standardparametern verwendet. Es wurde am Ende der Episode ein reward von 100 vergeben, während für alle anderen actions ein reward von -1 vergeben wird. Es werden 7 Pfannkuchen verwendet.

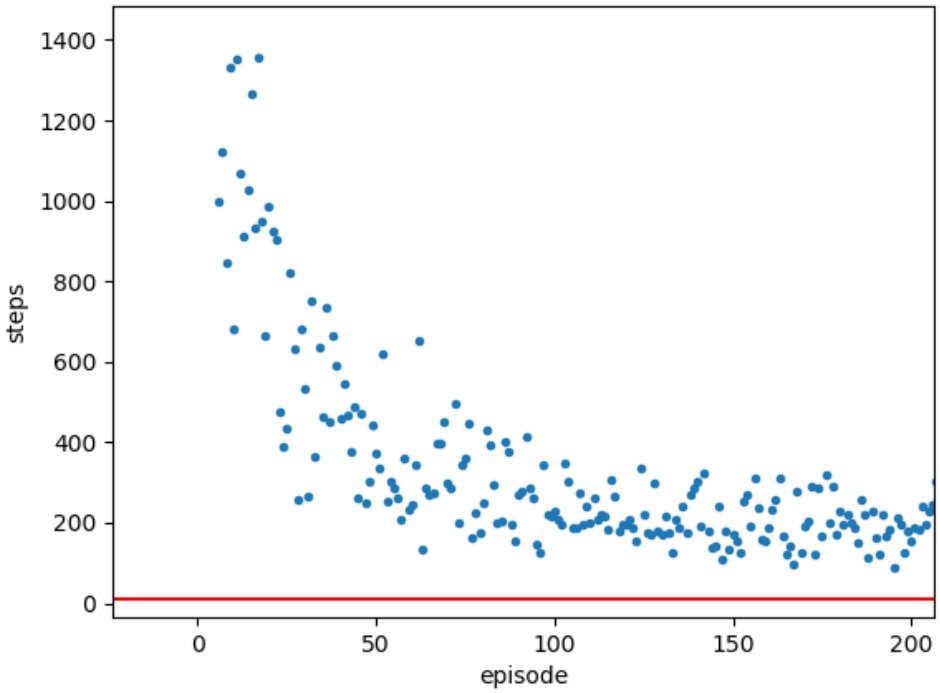

Abbildung 19: Q-learning ϵ-greedy Algorithmus mit reward scheduling

Im Vergleich zu Abbildung 11 lernt in diesem Falle ein Algorithmus mit reward scheduling sogar langsamer als einer ohne reward scheduling.

28 Variables ϵ

Es wird wieder ein Q-learning ϵ-greedy Algorithmus mit den Standardparametern bei 7 Pfannkuchen verwendet. Das ϵ wird mit 0,2 initialisiert und jede episode mit 0,95 multipliziert. Als untere Grenze für das ϵ wird 0,1 gesetzt.

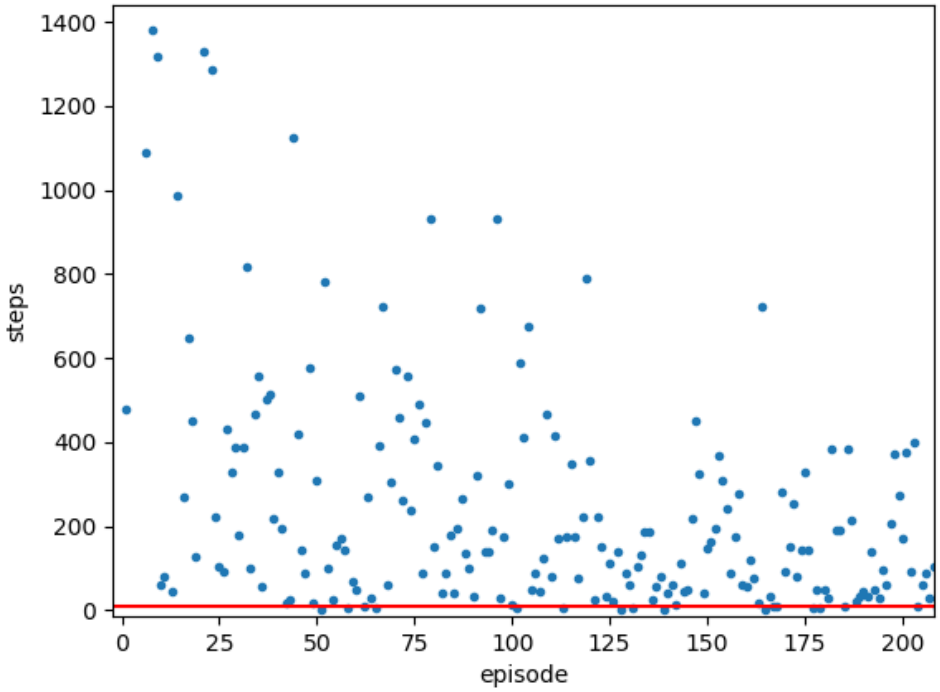

Abbildung 20: Q-learning ϵ-greedy Algorithmus mit decreasing ϵ

Da das ϵ am Anfang noch sehr groß ist, wird viel exploration betrieben. Demnach lernt der Algorithmus anfangs schlechter (vergleiche Abbildung 11). Schaut man jedoch auf spätere Episoden weist der Algorithmus mit decreasing ϵ eine bessere Leistung als der Algorithmus mit konstanten ϵ auf. Dies liegt daran, dass der Algorithmus mit decreasing ϵ schon vorher durch viel exploration bessere Wege gefunden hat.

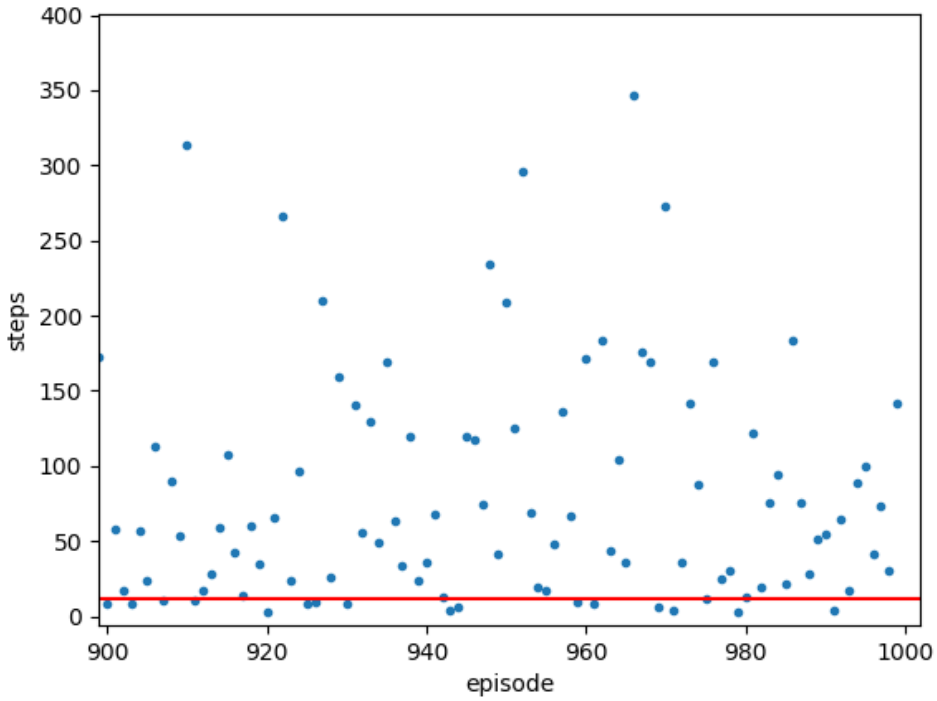

Abbildung 21: mit konstantem ϵ

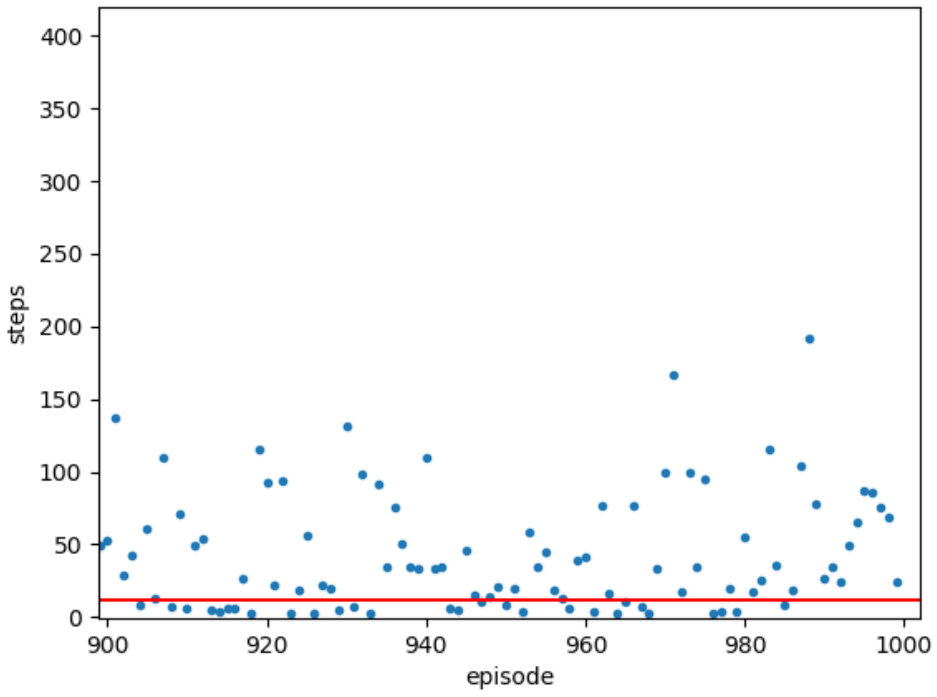

Abbildung 22: mit decreasing ϵ

29 Deep Q-learning

Für das deep Q-learning wurde das neuronale Netz verwendet, welches in Kapitel 24 in der build_model Funktion definiert ist. Weiterhin ist $\gamma = 0,9$, $\alpha = 0,01$, die Kapazität des ERM ist 50000 und die batch size ist 32. Das lernende Netz wird in jedem Zeitschritt trainiert und das target network wird alle 1000 steps aktualisiert. Neuronale Netze lernen viel langsamer als tabellarische Methoden. Das liegt daran, dass die beiden Methoden auf grundlegend anderen Prinzipien basieren. In einer Tabelle können direkt die Erfahrungswerte abgespeichert werden. Dem Lernen des neuronalen Netzes liegt ein Optimierungsvorgang zugrunde. Dabei werden viel mehr Daten benötigt, um eine Zielfunktion zu minimieren. Im Vergleich zu den Experimenten mit den vier Grundarten gibt es beim deep Q-learning keinen Lernfortschritt innerhalb der ersten 200 Episoden bei 7 Pfannkuchen. Um zu demonstrieren dass der Algorithmus trotzdem konvergiert, werden im folgenden Beispiel 200 Episoden mit 5 Pfannkuchen durchgeführt.

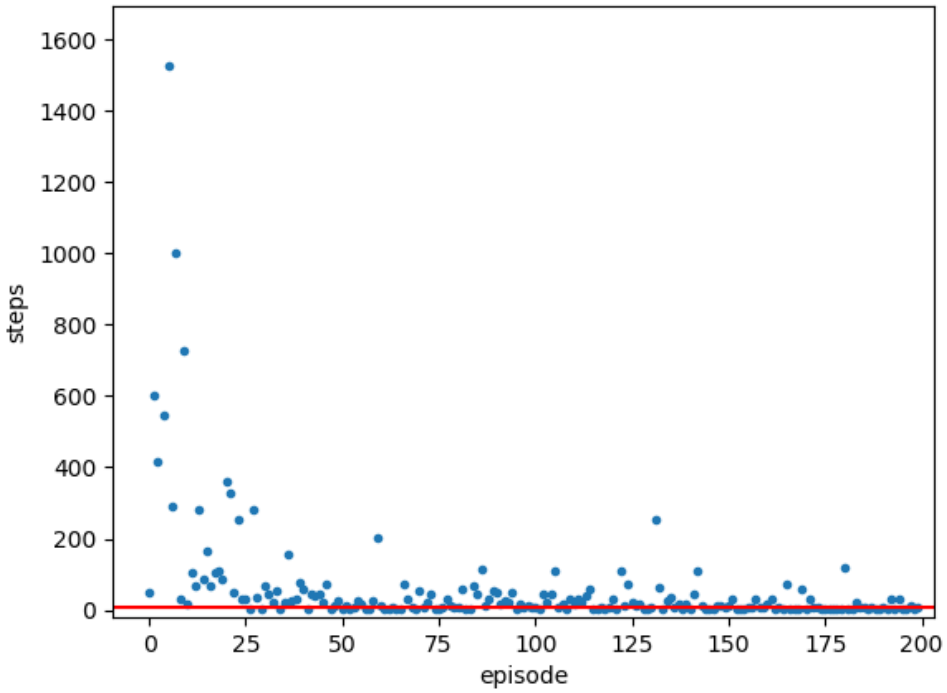

Abbildung 23: deep Q-learning

30 Zusammenfassung

In dieser Arbeit wurde sich mit reinforcement learning in Anwendung auf das Pfannkuchen-Sortierproblem beschäftigt. Reinforcement learning ist eine Art von maschinellem Lernen, die auf dem Prinzip des trail-and-error learning basiert. Die-

ses Prinzip ist auch eine der häufigsten Lernmethoden bei Tieren. Reinforcement learning lässt sich auf Probleme anwenden, die als Markow-Entscheidungsprozess formuliert werden können. Das Pfannkuchen-Sortierproblem ist ein finiter Markow-Entscheidungsprozess. Beim reinforcement learning gibt es verschiedene Methoden, die auf der Berechnung von value functions basieren. Am besten für das Pfannkuchen-Sortierproblem eignet sich das Temporal-Difference learning, da hier kein extrem hoher Rechenaufwand betrieben werden muss und auch keine vollständigen Episoden zum lernen benötigt werden. Für das Temporal-Difference learning gibt es die verschiedenen Algorithmen SARSA und Q-learning. Weiterhin tritt das exploration-exploitation Problem auf. Zu dessen Lösung werden ϵ-soft policies bei der Auswahl der actions eingesetzt. Das Problem besagt, dass man bei der Auswahl der actions die richtige Mitte zwischen dem Erkunden neuer und möglicherweise besserer actions und dem Verstärken der bisher bekannten besten actions finden muss. Zur Verbesserung der Leistung wurden zusätzlich die Methoden eligibility traces und reward scheduling betrachtet. Weiterhin wurde ein ganz anderer Ansatz als tabellarische Methoden vorgestellt. Dabei schätzt ein function approximator eine Kostenfunktion ab, um die Werte von actions zu beurteilen. Als function approximator kann zum Beispiel ein künstliches neuronales Netz verwendet werden. Dies wurde mit dem Algorithmus des deep Q-learning realisiert, welcher auf dem Q-learning basiert und ein neuronales Netz verwendet. Alle vorgestellten Algorithmen wurden in der Programmiersprache Python umgesetzt und deren Leistungen in Bezug auf das Pfannkuchen-Sortierproblem verglichen. Dazu wurden hauptsächlich die Bibliotheken NumPy (für diverse Berechnungen und Operationen mit arrays) und Keras (für das Arbeiten mit neuronalen Netzen) für Python verwendet.

30.1 Fazit: optimale policy?

Die verschiedenen Methoden haben alle etwas unterschiedlich gut funktioniert und haben verschiedene Vor- und Nachteile. Mit genügend Training würden wahrscheinlich alle diese Algorithmen eine nahezu optimale policy entwickeln. Als Beispiel wurde ein Q-learning ϵ-greedy Algorithmus so lange trainiert, bis er in 100 aufeinanderfolgenden Episoden weniger oder genau so viele Schritte benötigt wie es im worst-case optimal wäre. Dafür wurden die Parameter $\gamma = 0,9$, $\alpha = 0,01$ und ein decreasing ϵ verwendet. In der folgenden Tabelle sind die Ergebnisse für 4, 5, 6 und 7 Pfannkuchen zusammengefasst.

n	P_n	benötigte Episoden
4	4	95844
5	5	349606
6	7	3712695
7	8	76371090

Einige Methoden verhielten sich in der Praxis nicht ganz so, wie es theoretisch erwartet wurde. Die eligibility traces sollten, gerade weil nur sparse rewards vorhanden sind, eine deutlich bessere Leistung bringen, als einer der Grundalgorithmen ohne eligibility traces. Es zeigte sich hier nur eine kleine Verbesserung. Auch die Methode des reward scheduling ist darauf ausgelegt, bei sparse rewards bessere Leistungen zu erzielen. Im Experiment schnitt ein Algorithmus mit reward scheduling sogar schlechter ab, als ein Algorithmus ohne reward scheduling. Weiterführend kann man sich bei allen verwendeten Methoden zusätzlich verschiedene Daten anschauen, diese auswerten und umfangreiche Parameterforschung betreiben, um die Ergebnisse der Programme zu verbessern. Dafür könnte man bei den tabellarischen Methoden die Änderung der Q-values oder bei deep Q-learning die Funktionswerte der loss function untersuchen. Weiterhin wäre es möglich, verschiedene activation functions und optimizer zu verwenden und diese auch in ihren Parametern anzupassen.

Zusammenfassend kann man sagen, dass das Ziel dieser Arbeit, ein Programm zu entwickeln, welches das Pfannkuchen-Sortierproblem nahezu perfekt löst, erfüllt ist. Nach Training des Programmes, welches hier beispielsweise für 4, 5, 6 und 7 Pfannkuchen umgesetzt wurde, wurde das ϵ wieder auf 0 gesetzt und einige Testepisoden durchgeführt. In circa 99% der Fälle wurden weniger oder genau so viele Schritte benötigt wie es die der Problemgröße zugehörige Pfannkuchenzahl angibt. Mit entsprechend mehr Rechenaufwand kann können solche Programme auch für höhere Problemgrößen verwendet werden. Hierbei ist anzumerken, dass sich die policy anfangs sehr schnell und später nur extrem langsam verbessert. Daher ist ein Algorithmus, welcher nur für einen kleinen Teil der in der Tabelle angegebenen Episodenanzahlen trainiert wird, schon fast so gut wie der finale Algorithmus, welcher 100 mal in Folge das Optimum erzielt. Solch ein Algorithmus ist in der Praxis meist ausreichend, da er auch schon fast perfekte Werte erzielt.

Besonders hervorzuheben ist auch das deep Q-learning. Da die Q-values nicht in einer riesigen Tabelle gespeichert werden, sondern mit einen künstlichen neuronalen Netzwerk gearbeitet wird, kann deep Q-learning theoretisch auch auf beliebig große Problemgrößen angewandt werden, da nicht so viel Speicherplatz benötigt wird. Dafür benötigt deep Q-learning viel mehr Rechenleistung. Auch deep Q-learning konvergierte. Mit genügend Training wird also auch mit deep Q-learning eine fast optimale policy erreicht werden.

Die Methoden, welche in dieser Arbeit verwendet wurden, eigneten sich gut, um ein Programm zu entwickeln, welches das Pfannkuchen-Sortierproblem sehr gut löst.

30.2 Ausblick: Pfannkuchen bei Mutationen

Im Alltag findet die Sortiermethode des „sorting by prefix reversal“ keine Anwendung, da sie zu umständlich ist und es weitaus effizientere Sortieralgorithmen gibt. Tatsächlich ist das Pfannkuchen-Sortierproblem in der Biologie relevant. Chromo-

somen sind Sequenzen aus verschiedenen Genen. Manchmal kann es passieren, dass Gene durch kleine Mutationen verändert werden. Seltener kommt es vor, dass sich komplette Gene oder Reihen von Genen umdrehen. Wenn man sich bestimmte Äquivalente Chromosomen von Lebewesen anschaut, kann man bestimmen wie eng die Lebewesen evolutionär verwandt sind, wenn man die kleinste Anzahl an Umkehrungen bestimmt, die benötigt werden, um eine Sequenz von Genen in eine andere zu überführen.

Allerdings funktioniert das Umdrehen bei Genen etwas anders als beim klassischen Pfannkuchen-Sortierproblem. Die Umkehrungen passieren nicht nur an einem Ende der Sequenz, sondern können überall in der Sequenz passieren. Eine Sortiermethode nach diesem Prinzip heißt „sorting by reversal“. Das Pfannkuchen-Sortierproblem müsste in diesem Kontext zum Pfannkuchen-Sortierproblem mit extra Teller erweitert werden. Hier hat der Kellner die Möglichkeit einen Teilstapel der Pfannkuchen auf einen extra Teller zu legen, danach mit dem ursprünglichen Stapel einen prefix reversal durchzuführen und anschließend den Teilstapel vom extra Teller wieder oben hinzuzufügen. Ein weiterer Unterschied zum klassischen Pfannkuchensortieren ist, dass die Gene eine Ausrichtung haben. Es macht also einen Unterschied mit welcher Seite die einzelnen Pfannkuchen nach oben und mit welcher Seite nach unten liegen. Diese Modifikation wird durch das verbrannte Pfannkuchen-Sortierproblem umgesetzt. Hierbei ist jeder Pfannkuchen auf genau einer Seite verbrannt. Am Ende sollen die Pfannkuchen nicht nur nach Größe sortiert werden, sondern auch noch alle mit der verbrannten Seite nach unten zeigen.

Durch die eben beschriebenen Modifikationen könnten die in dieser Arbeit entwickelten Programme, auch auf das Problem der Umkehrungen von Genen bei Mutationen angewandt werden. Mit genügend Training könnte solch ein Programm bestimmen, wie eng Lebewesen evolutionär verwandt sind.

31 Literaturverzeichnis

[1] Sutton, Richard S.; Barto, Andrew G.: *Reinforcement learning: An Introduction*. 2. Auflage. Cambridge, MA : The MIT Press, 2018 - ISBN 9780262039246

[2] Vitay, Julien: Deep Reinforcement Learning. URL: https://julien-vitay.net/deeprl/ [1.1.2020].

[3] Oppermann, Artem: Artificial Intelligence vs. Machine Learning vs. Deep Learning. URL: https://www.deeplearning-academy.com/p/ai-wiki-machine-learning-vs-deep-learning [1.1.2020].

[4] Chitturi, B.; Fahle, W.; Meng, Z.; Morales, L.; Shields, C.O.; Sudborough, I.H.; Voit, W.: *An (18/11)n upper bound for sorting by prefix reversals*. In: Theoretical Computer Science Volume 410 (2009). S. 3372-3390

[5] Wikipedia: Pancake sorting. URL: https://en.wikipedia.org/wiki/Pancake_sorting [1.1.2020].

[6] Wikipedia: Artificial Intelligence. URL: https://en.wikipedia.org/ wiki/Artificial_ intelligence [1.1.2020].

[7] Wikipedia: Machine learning. URL: https://en.wikipedia.org/wiki/Machine_learning [1.1.2020].

[8] Roney-Dougal, Colva; Vatter, Vince: Of pancakes, mice and men. URL: https://plus.maths.org/content/pancakes-mice-and-men [1.1.2020].

[9] Mnih, V.; Kavukcuoglu, K.; Silver, D. et al.: *Human-level control through deep reinforcement learning*. In: Nature 518 (2015). S. 529–533

32 Ehrenwörtliche Erklärung

Hiermit erkläre ich, Obermüller, Lennart, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer, als der angegebenen Quellen angefertigt habe. Ich möchte mich besonders bei Herrn Dr. habil. Julien Vitay und bei Herrn Steffen Polster bedanken, die mich bei meiner Arbeit unterstützt haben. Diese Arbeit hat in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum

Lennart Obermüller