DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Reinforcement Learning

Hausarbeit

im Studiengang Angewandte Informatik

an der Dualen Hochschule Baden-Württemberg Mannheim

von

Luisa Fuss und Lea Pellikan

10.06.2021

Vorlesung	Wissensbasierte Systeme
Semester	6
Matrikelnummer, Kurs	5136331 und 3062311, TINF18AI2
Ausbildungsfirma	Roche Diagnostics GmbH
	DB Systel GmbH

Inhaltsverzeichnis

1 Einleitung

Die vorliegende Arbeit behandelt das Thema „Reinforcement Learning“ und wurde im Rahmen einer Prüfungsleistung für das Fach „Wissensbasierte Systeme“ erstellt. Dieses Dokument sowie die dazugehörige Implementierung sind ebenfalls auf GitHub unter luisafuss/reinforcement_learning zu finden.

1.1 Motivation

Stellen Sie sich vor, Sie haben einen Roboter, der geradeaus laufen soll. Zum Bewegen besitzt er einige Motoren als Gelenke. Ihm genau zu sagen, wie er sich bewegen soll, um nach vorne zu kommen, bedeutet einen großen Programmieraufwand und erfordert auch das Wissen, wie die optimale Bewegung aussehen muss. Einfacher ist es, den Roboter dazu zu befähigen, sich die Bewegung selbst beizubringen. Dazu erhält der Roboter ähnlich wie beim Training mit Tieren Belohnungen für gute Bewegungen. Als gute Bewegung wird hier ein Schritt verstanden, der den Roboter nach vorne bewegt. Geht er zurück, kann er analog eine Strafe erhalten. So ist es möglich, das Ziel zu kennen, aber den Weg dorthin offen zu lassen. Das Konzept der Informatik, dass solche Problemlösungen ermöglicht, nennt sich Reinforcement Learning.

Reinforcement Learning (RL), zu Deutsch auch bestärkendes oder verstärkendes Lernen, ist eine Methode des Maschinellen Lernens, bei der ein Agent die Strategie zum Erreichen eines Zieles selber erlernen muss. Der Agent versucht die Belohnungen, die er durch die Belohnungsfunktion erhält, zu maximieren. Der Vorteil dabei ist, dass auch wenn keine Daten vorhanden sind, ein Modell erstellt werden kann. Künstliche Intelligenz (KI) kann in vielen Bereichen eingesetzt werden. Doch das Problem, das jede KI hat: Sie braucht Daten, um trainiert zu werden. Diese Daten gilt es zu sammeln. In der Theorie kann eine KI nie besser werden, als die Daten, die sie bekommt. Beim Reinforcement Learning ist das anders: Hierbei wird die Strategie der KI eigenständig erlernt. Stuart Russel und Peter Norvig fassen dies treffend zusammen: ‚Imagine playing a new game whose rules you don’t know; after a hundred or so moves, your opponent announces, „You lose.“ This is reinforcement learning in a nutshell.‘ (Russell und Norvig 2010)

1.2 Aufgabenstellung

Innerhalb dieser Arbeit soll erläutert werden, was Reinforcement Learning ist, worauf es basiert, welche Algorithmen verwendet werden können und wie bei der Entwicklung vorzugehen ist. Als Grundlage dient die Vortragsreihe „CS 188: Artificial Intelligence“ von Pieter Abbeel und Dan Klein an der University of California, Berkeley. Zuerst wird auf den Markov-Decision-Process eingegangen, der die grundlegende Definition eines RL-Problems beschreibt.

Im Anschluss werden verschiedene Verfahren vorgestellt, die das Vorgehen der KI beschrieben. Mit Hilfe von Q-Learning wird dann eine Policy erstellt, mit deren Hilfe das vorher definierte Problem gelöst werden kann. Zum Schluss wird eine Evaluierung der unterschiedlichen Verfahren ausgeführt. Die Arbeit wird durchgehend von einem praktischen Beispiel begleitet, in dem die Theorie implementiert wird.

2 Markov Decision Process

Die Grundlage für Reinforcement Learning bilden Markow-Entscheidungsprozesse. Aus diesem Grund sollen jene nun genauer betrachtet werden. Ein Markov Decision Process (MDP) oder zu Deutsch auch Markow-Entscheidungsprozess ist ein Fünf-Tupel der Form (S, A, T, R, s_0) wobei gilt:

- S = eine Menge von Zuständen (states)
- A = eine Menge von Aktionen (actions)
- $T(s, a, s')$ = eine transition function mit $s \in S$
- R = Belohnungsfunktion $R(s, a, s')$
- s_0 = Startzustand

Ein MDP ist immer von seinem Umfeld, dem Environment abhängig und kann optional einen Endzustand besitzen. Die Lösung von einem solchen Problem ist eine Funktion der Form $\pi\colon S \rightarrow A$. Diese wird auch Policy genannt. Die Policy gibt an, welche Aktion in welchem Zustand ausgeführt wird. Ziel ist es, eine (möglichst) optimale Policy zu finden. (vgl. Abbeel und Klein 2014 a)

States sind eine Menge an Token, die angeben, in welchem Zustand sich jemand in der Umgebung befindet. Dies kann zum Beispiel die Position sein, in der sich jemand, bzw. genauer gesagt der Agent, in der Umgebung befindet. Ein *Agent* kann in einer Umgebung (*Environment e*) eine fest definierte Menge an *Aktionen* ausführen, die ihn in einen neuen *State s* bringen.

Diese Theorie soll nun praktisch an einem Bespiel von einem Lieferwagen eingesetzt und erläutert werden, bevor näher auf die Transition- und Reward-Function eingegangen wird.

2.1 Implementierung 1

2.1.1 Das Problem

Wir sind Betreiber einer Spedition und sind zuständig für die Belieferung von Supermärkten. Dafür muss Ware zwischen Lagern und den Märkten innerhalb unserer Stadt transportiert werden. Unser innovativer LKW ist selbstfahrend und bekommt lediglich den Auftrag an einer Station (Lager oder Supermarkt) Ware einzusammeln und an einer anderen Station wieder abzuliefern.

2.1.2 Definitionen

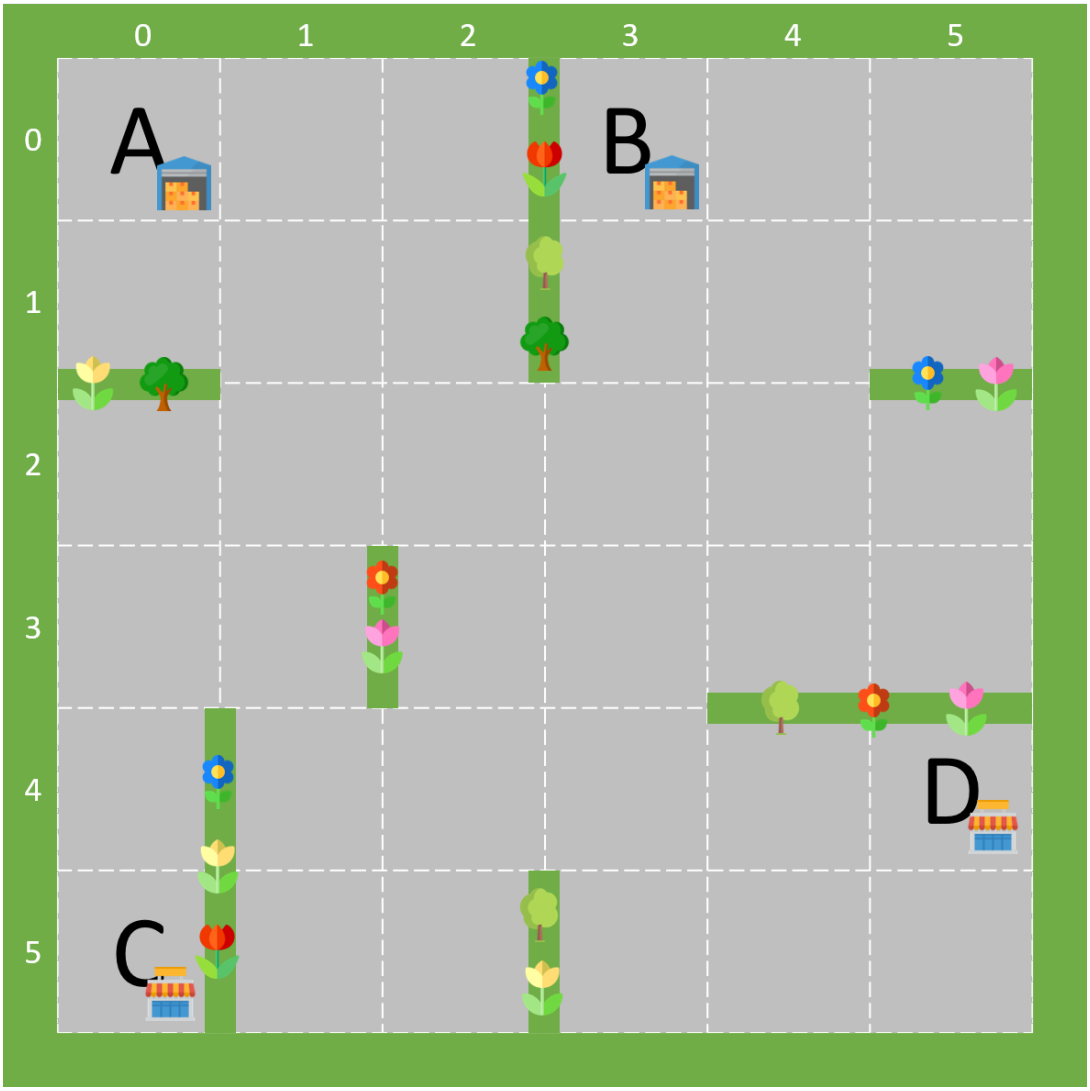
Wir bedienen zwei Lager und zwei Supermärkte.

Lager:
0. A-Lager (A)

1. B-Lager (B)

Supermarkt:
2. C-Markt (C)
3. D-Markt (D)

Diese verteilen sich folgendermaßen in unserer Stadt:



Die Stadt ist eine 6x6-Quadratestadt und mit 36 Positionen, die die Koordinaten $(0, 0)$ bis $(5, 5)$ haben, versehen. Der LKW kann sich frei in der Stadt bewegen, aber nicht durch die Grünstreifen fahren.

Die Anzahl der Zustände (*States*) ergibt sich folgendermaßen:

- 6 x 6 Positionen
- 4 Orte zu denen die Ware gebracht werden kann (A bis D bzw. 0 bis 3)

- 5 Orte, an denen sich die Ware befindet (A bis D bzw. 0 bis 3 und im LKW (Position Nr.4))

$$6 \cdot 6 \cdot 4 \cdot 5 = 720 \texttt{ mögliche Zustände}$$

Die Aktionen (*Actions*), die der LKW ausführen kann sind: 0. nach Norden fahren 1. nach Osten fahren 2. nach Süden fahren 3. nach Westen fahren 4. Ware einsammeln 5. Ware abladen

Dabei kann er folgende Belohnungen (und Abzüge) (*Rewards*) erhalten:

- Ware korrekt abliefern: +20
- Ware falsch einsammeln/abliefern: -10
- Pro Schritt: -1

2.1.3 Definitionen implementieren

Zunächst werden alle Libraries, die jetzt oder im Verlaufe der gesamten Entwicklung benötigt werden, importiert. `copy` dient dazu eine Deepcopy von einem Objekt anzufertigen, um nicht nur die Referenz darauf zu übergeben. `random` bietet verschiedene Methoden, um Zufallszahlen zu generieren oder eine zufällige Auswahl zu treffen. Die Library `time` kann später verwendet werden, um zwischen der Visualisierung der einzelnen Schritte eine kleine Pause einzulegen, sodass sich die einzelnen Zustände leichter erkennen lassen. Mit Hilfe von `tqdm` kann eine Fortrittsleiste bei Iterationen angezeigt werden. `numpy` bietet zahlreiche Funktionen unter anderem für den Umgang mit Arrays, Matritzen oder mathematischen Berechnungen.

```
import copy
import random
import time
from tqdm import tqdm
import numpy as np
```

Im Folgenden werden die zuvor getroffenen Spezifikationen in ein für den Rechner verständliches Format überführt. Es gibt jeweils sechs Spalten und Reihen (0 bis 5) und eine Menge von Aktionen (ebenfalls 0 bis 5). Für die Ausgabe wird noch eine Überführung in eine Beschreibung vorgenommen. Außerdem werden die Koordinaten der Lager und Supermärkte festgehalten. Die Koordinaten haben dabei die Form `(Reihe, Spalte)`.

Die Grünstreifen werden in einer Menge gespeichert. Ein Grünstreifen liegt immer zwischen zwei Feldern. Diese werden in einem Tupel in der Reihenfolge `(Links, Rechts)` bzw. `(Oben, Unten)` angegeben. Daraus ergibt sich für jedes Stück eines Grünstreifens ein Tupel der Form `((Reihe Zelle links, Spalte Zelle links), (Reihe Zelle rechts, Spalte Zelle rechts))` bzw. mit „oben“ und „unten“, falls es sich um einen horizontalen Streifen handelt.

```
rows = [row for row in range(0, 6)]
cols = [col for col in range(0, 6)]
actions = {action for action in range(0, 6)}
actions_description = ["Drive north", "Drive east", "Drive south", "Drive␣
 ↪west", "Pickup goods", "Dropoff goods"]
stations = [(0,0), (0,3), (5,0), (4,5)]
stations_descriptions = ["Warehouse A", "Warehouse B", "Supermarket C",␣
 ↪"Supermarket D"]
position_goods_descriptions = stations_descriptions + ["In the truck"]
walls = {
    ((0,2), (0,3)), #vertikal
    ((1,2), (1,3)),
    ((3,1), (3,2)),
    ((4,0), (4,1)),
    ((5,0), (5,1)),
    ((5,2), (5,3)),
    ((1,0), (2,0)), # horizontal
    ((1,5), (2,5)),
    ((3,4), (4,4)),
    ((3,5), (4,5))
}
```

Als nächstes soll für jedes Feld im Stadtplan festgehalten werden, welche Aktionen von diesem Feld ausgehend möglich sind. Zunächst wird allen Feldern alle Aktionen zugewiesen. Den am Rand liegenden Feldern wird die Aktion aberkannt, die aus der Stadt raus führen würde. Einsammel- und Ablieferaktionen sind nur an den zuvor definierten Stationen möglich, weshalb den anderen Feldern diese Aktion ebenfalls entzogen wird. Als nächstes werden Felder betrachtet, die in der direkten Nachbarschaft eines Grünstreifens liegen und dort die Aktionen entfernt, die den LKW dazu veranlassen würden, die Grünanlage zu zerstören. Daraus ergibt sich dann das gewünschte Dictionary mit dem Feld als Key und einer Menge möglicher Aktionen als Value.

```
possible_actions = dict()

for row in rows:
    for col in cols:
        possible_actions[(row, col)] = copy.deepcopy(actions)
for key in possible_actions:
    (row, col) = key
    # Ränder
    if row == 0:
        possible_actions[key].remove(0)
```

```
    elif row == 5:
        possible_actions[key].remove(2)
    if col == 0:
        possible_actions[key].remove(3)
    elif col == 5:
        possible_actions[key].remove(1)
    # Einsammeln, Abliefern
    if (row, col) not in stations:
        possible_actions[key].remove(4)
        possible_actions[key].remove(5)
    # Grünstreifen
    if ((row, col), (row, col + 1)) in walls:
        possible_actions[key].remove(1)
    if ((row, col - 1), (row, col)) in walls:
        possible_actions[key].remove(3)
    if ((row, col), (row + 1, col)) in walls:
        possible_actions[key].remove(2)
    if ((row - 1, col), (row, col)) in walls:
        possible_actions[key].remove(0)
```

2.1.4 Die Transport-Klasse

Die Transportklasse speichert den aktuellen Zustand (`state`), die grafische Darstellung (`canvas`) von diesem sowie den aktuell erreichten Wert (`current_value`). Außerdem speichert sie, wie viele Schritte bereits ausgeführt wurden (`steps`). Ebenfalls ist eine Methode zur Ermittlung der nächsten Aktion, die `action_method`, zu übergeben. Dies kann eine Methode sein, die eine zufällige Aktion zurückgibt (`random_action`) oder eine Methode, die die durch Q-Learning erlernten Werte ausnutzt (`q_learning_action`). Außerdem werden alle besuchten Zustände in `visited_states` aufgelistet. Ist die Variable `stepwise` auf `True` gesetzt, wird zur besseren Sichtbarkeit der Aktionen in der Darstellung nach jeder Aktion eine kurze Pause durchgeführt. Die graphische Darstellung erfolgt nur, wenn `visualize` auch `True` ist. Die Variable `done` markiert, ob der Transport der Waren erfolgreich abgeschlossen wurde. Für eine spätere Berechnung wird die Vorgängerposition der Waren benötigt, weshalb diese hier ebenfalls abgespeichert wird.

```
class Transport():
    def __init__(self, state, action_method, stepwise = False, visualize =␣
 ↪False):
        self.state = state
        self.canvas = self.init_canvas()
        self.action_method = action_method
        self.current_value = 0
```

```
        self.steps = 0
        self.visited_states = [self.state]
        self.stepwise = stepwise
        self.visualize = visualize
        self.started = False
        self.done = False
        self.position_goods_old = self.state[1]

        self.canvas[3].on_mouse_down(self.handle_mouse_down)
```

2.2 Transition-Funktion, Probability und Reward-Funktion

Alle Schritte, die bis jetzt beschrieben wurden, befassen sich mit dem Problem. Um eine Lösung zu finden, wird eine Policy-Funktion $\pi(s)$ aufgestellt, die für einen State s eine Aktion a zurückgibt, die die vermutlich höchste Belohnung für den Agenten bereithält. Die Policy-Funktion kennt Tripel der Form $< s, a, r >$ wobei s der aktuelle State ist, a die Aktion die im State s ausgeführt wird und r die Belohnung, die der Agent für die Aktion a im State s bekommen würde. Die Policy-Funktion hat viele von diesen Tripeln vorliegen und gibt die Aktion mit der höchsten Belohnung zurück. Eine optimale Policy $\pi*$ gibt immer die Aktion an, die langfristig die höchste Belohnung ergibt.

Der Prozess, also der Weg vom Start- zum Endzustand, hat dabei die Markov-Eigenschaft. Das bedeutet, dass der Folgezustand nur vom aktuellen Zustand abhängig ist und nicht auf den vorangegangenen basiert. Es gilt also:

$$P(S_{t+1} = s'_t | s_t, a_t, s_{t-1}, a_{t-1}) = P(S_{t+1} = s'_t | s_t, a_t) \forall s \in S$$

Das Markovsche Prinzip besagt, dass die vergangenen States oder Aktionen keinen Einfluss auf die aktuellen oder zukünftigen Entscheidungen haben. Die Wahrscheinlichkeit, die die Transition-Funktion berechnet ist unabhängig von den vorherigen States.

Die $Transition - Funktion$ gibt an, mit welcher Wahrscheinlichkeit man von einem Zustand in den nächsten gelangt. Es handelt sich also um eine Funktion der Art $P_{ss'}(s'|s, a)$. Dabei ist P die Wahrscheinlichkeit, dass Aktion a vom Zustand s zu s' führt. Das Ergebnis der Funktion kann mit Hilfe einer Matrix dargestellt werden:

$$P = \begin{bmatrix} P_{11} & \dots & P_{1n} \\ \dots & & \\ P_{n1} & \dots & P_{nn} \end{bmatrix}$$ wobei 1 bis n alle möglichen Zustände bezeichnen

Die Summe der Werte aus einer Reihe der Matrix muss für alle Reihen 1 ergeben.

Für jeden Zustandswechsel kann eine Belohnung r („Reward“) vergeben werden. Der Höhe der Belohnung ist durch die $Reward - Funktion$ festgelegt. In vielen MDPs wird die Belohnung mit einem Faktor γ, dem „Discount“ multipliziert. Dabei ist $\gamma \in [0, 1]$ und wird mit der Anzahl

der bisher erfolgten Schritte t potenziert. Der Discount-Faktor wird oftmals genutzt, um zu erreichen, das früher erhaltene Belohnungen stärker gewichtet werden als spätere. Der aktuelle Wert des Zustands ergibt sich aus der Summe der erhaltenen Belohnungen mal des Discounts. Ziel ist es, den Wert zu maximieren. Es gilt:

$$\texttt{Wert} = \sum_{k=0}^{t} \gamma^k \cdot r_{k+1}$$

(vgl. Abbeel und Klein 2014 a, Silver 2015) Das Ganze soll nun wieder an unserem Beispiel erläutert werden.

2.3 Implementierung 2

Die Transition function gibt für eine Aktion den Zustand zurück der folgt, wenn man auf den aktuellen Zustand die gewünschte Aktion anwendet. Dies ist möglich, da man aus der Kombination Zustand s_t und Aktion a in diesem Beispiel eindeutig den Folgezustand s_{t+1} bestimmen kann, die Wahrscheinlichkeit für diesen Folgezustand also bei 100% liegt. Falls die Aktion nicht möglich ist, wird der alte Zustand zurückgegeben. Es gilt dann $s_t = s_{t+1}$. Ein Zustand ist dabei folgendermaßen aufgebaut:

`(Position LKW, Position Ware, Position Ziel).`

Wobei gilt:

- Position LKW = (Reihe LKW, Spalte LKW)
- Position Ware $\in [0, 4] \rightarrow$ [Lager A, Lager B, Supermarkt C, Supermarkt D, im LKW]
- Position Ziel $\in [0, 3] \rightarrow$ [Lager A, Lager B, Supermarkt C, Supermarkt D]

Bei der Funktion handelt es sich um eine Funktion der Klasse `Transport`.

```
def transition_function(self, action):
    position_truck, position_goods, position_goal = self.state
    if action not in possible_actions[position_truck]:
        return self.state

    row, col = position_truck
    if action == 0:
        position_truck = (row - 1, col)
    elif action == 1:
        position_truck = (row, col + 1)
    elif action == 2:
        position_truck = (row + 1, col)
    elif action == 3:
```

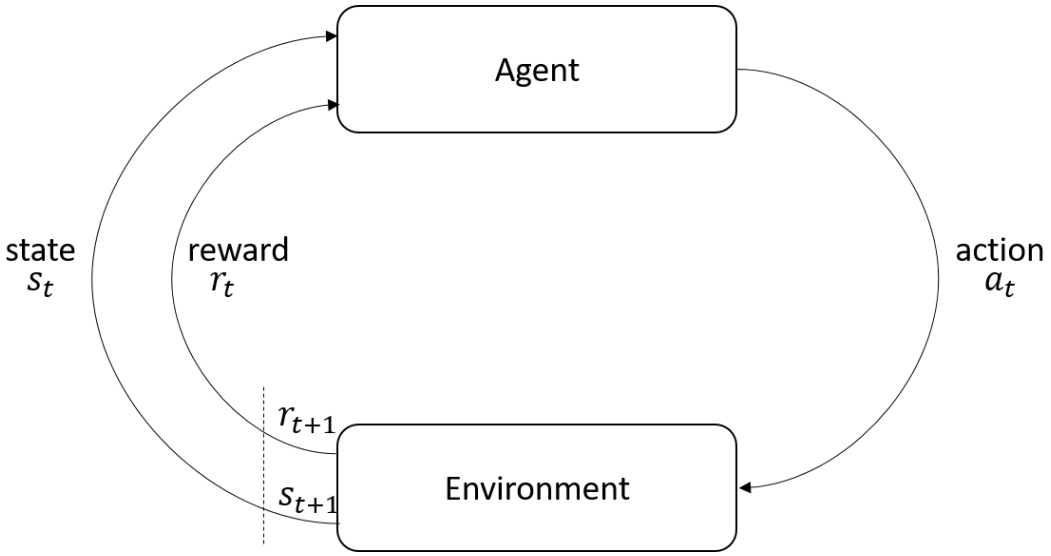
```
        position_truck = (row, col - 1)
    elif action == 4:
        if position_goods == stations.index(position_truck):
            position_goods = 4
    elif action == 5:
        if position_goods != 4:
            return self.state
        position_goods = stations.index(position_truck)

    return (position_truck, position_goods, position_goal)
Transport.transition_function = transition_function
del transition_function
```

`reward_function()` gibt die Belohnung zurück, die für die Aktion im aktuellen Zustand erzielt wird. Bei der Funktion handelt es sich ebenfalls um eine Funktion der Klasse `Transport`.

```
def reward_function(self, action):
    reward = -1 # Pro Schritt
    position_truck, position_goods, position_goal = self.state

    if action not in possible_actions[position_truck]: return reward

    if action == 4:
        station_truck = stations.index(position_truck)
        if position_goods != station_truck: # Ware falsch einsammeln
            reward -= 10
    elif action == 5:
        station_truck = stations.index(position_truck)
        if position_goal != station_truck and position_goods == 4: # Ware␣
 ↪falsch abliefern
            reward -= 10
        elif position_goal == station_truck and position_goods == 4: # Ware␣
 ↪korrekt abliefern
            reward += 20
        else: # Abliefern ohne geladene Ware
            reward -= 10

    return reward
Transport.reward_function = reward_function
del reward_function
```

3 Reinforcement learning

Beim Reinforcement Learning geht es darum, dass sich die KI ihr Modell innerhalb eines Lernprozesses selber entwickelt. Während des Lernprozesses wird mit Hilfe der `reward_function()` eigenständig eine Strategie entwickelt, mit deren Hilfe ein Ziel erreicht werden kann. Ein Ziel zeichnet sich im Algorithmus dadurch aus, dass der Agent eine Belohnung erhält, wenn er den Ziel-Zustand erreicht. Zum Trainieren werden zufällige Aktionen ausgeführt.



Die Abbildung 3.1 zeigt, den Zyklus aus Aktionen, die das Environment beeinflussen und den Erkenntnissen, die der Agent dadurch entwickelt. Im nächsten Schritt nutzt er dieser Erkenntnisse aus dem Folgezustand und dem erhaltenen Reward zur Auswahl der nächsten Aktion.

Mithilfe von verschiedenen Verfahren, die im Laufe dieser Arbeit erläutert werden, kann eine Strategie erstellt werden, die den Agenten zum Ziel führt. Diese Strategie (engl. Policy) wird mit π bezeichnet. Mit den bis jetzt definierten States, Aktionen, Rewards und der Umgebung können nun die Strategien für den Lernprozess angewendet werden.

3.1 Implementierung 3

Um die verschiedenen Algorithmen des Reinforcement Learnings wieder auf unser Beispiel anwenden zu können, wird zunächst eine allgemeine Funktion definiert.

`transport_goods()` führt so oft eine Aktion, die von der gewünschten `action_method()` ausgewählt wird, aus, bis die Ware an ihrem Zielort eingetroffen ist. Nach dem Ausführen wird die Darstellung falls gewünscht aktualisiert. Die Konstante `MAX_VISITS_STATE` gibt an, wie oft ein Zustand besucht werden kann, bevor ausgehend von diesem Zustand statt der eigentlich

gewünschten Aktion eine zufällige Aktion gestartet wird. Diese Randomisierung ist notwendig, um zu verhindern, dass sich Aktions-Schleifen bilden, in denen der Agent festhängt. Im idealen Fall besucht der Agenten einen State nur einmal (Position zweimal, einmal mit Paket, einmal ohne). Der Wert wird trotzdem höher angesetzt, um keine zu starke Einschränkung zu bieten.

Bei der Funktion handelt es sich auch um eine Funktion der Klasse `Transport`.

```
MAX_VISITS_STATE = 10
```

```
def transport_goods(self):
    position_truck, position_goods, position_goal = self.state
    while not self.done:
        self.steps += 1
        state_num = state_to_state_num(self.state)
        action = self.action_method(state_num)
        if action not in possible_actions[position_truck]:
            action = random.sample(possible_actions[position_truck], 1)[0]

        value = self.reward_function(action)
        new_state = self.transition_function(action)
        if self.visited_states.count(new_state) > MAX_VISITS_STATE:
            action = random.sample(possible_actions[position_truck], 1)[0]
            value  = self.reward_function(action)
            new_state = self.transition_function(action)

        self.current_value += value
        self.state = new_state
        self.visited_states.append((self.state))
        position_truck, position_goods, position_goal = self.state

        if position_goods == position_goal: self.done = True
        if self. visualize:
            action_label.value = 'Action: ' + actions_description[action]
            self.update_canvas()
        if self.stepwise: time.sleep(0.1)
Transport.transport_goods = transport_goods
del transport_goods
```

Im Folgenden werden alle möglichen Zustände in einer Liste gespeichert, sodass sie nummerierbar und damit eindeutig identifizierbar sind. Die nachfolgenden Funktionen `state_to_state_num()` und `state_num_to_state()` dienen dazu, zwischen den beiden Darstellungsweisen zu wechseln.

```
states = []
for row in range (0, 6):
    for col in range (0, 6):
        for position_goods in range (0, 5):
            for position_goal in range (0, 4):
                states.append(((row, col), position_goods, position_goal))
```

```
def state_to_state_num(state):
    return states.index(state)

def state_num_to_state(num):
    return states[num]
```

Um einen Vergleich zu haben, wie der Transport ohne Reinforcement Learning aussehen würde, wird die folgende Methode genutzt. Die Funktion `random_action()` gibt eine zufällige Aktion zurück. Sie nutzt also keinen Lerneffekt oder Trainingsdaten. Im weiteren Verlauf der Arbeit werden verschiedene Algorithmen zur Auswahl der Aktionen implementiert, die genau dies tun. In der abschließenden Evaluation wird dann zu sehen sein, wie sich die Nutzung dieser Daten auf die Effizienz des Transports auswirkt.

```
def random_action(state_num):
    return random.randrange(0, 6, 1)
```

3.2 Modell basiert vs. Modell frei

Im Reinforcement Learning lassen sich die Probleme und ihre Lösungsalgorithmen in zwei Ansätze aufgliedern. Zum einen gibt es den Modell basierten Ansatz, häufig als modell-based bezeichnet. Die andere Gruppe heißt Modell frei bzw. modell-free. Verwendet der Agent während dem Lernvorgang Vorhersagen darüber, welche Reaktion zu erwarten ist, so handelt es sich um einen Modell basierten Ansatz. Dies kann den Umfang einer Stichprobe haben, oder sogar die komplette Information über die Verteilung des nächsten Zustands inklusive aller Belohnungen beinhalten. Modell basiert bedeutet jedoch nicht, dass kein Modell der Umgebung implementiert sein darf, oder das Modelle für Maschinelles Lernen vorhanden sein müssen. Typische Beispiele für Modell basierte Algorithmen sind Policy oder Value Iteration die dem Bereich Dynamic Programming angehören. Basiert ein Algorithmus auf Erfahrungswerten, wie zum Beispiel Monte-Carlo und Q-learning, gehören entsprechend der Gruppe der Modell freien Methoden an. Sie verwenden lediglich Proben aus den bereits erlernten Werten, aber keine Vorhersagen über zukünftige Belohnungen. (vgl. Russell und Norvig 2010)

3.3 Passives lernen vs. aktives lernen

Reinforcement Learning Methoden lassen sich in zwei weitere Kategorien aufteilen. Beim passiven Lernen steht eine optimale Policy für den Agenten bereits und er muss bewerten, wie gut die ausführbaren Aktionen sind. Muss der Agent noch lernen, was er tun soll, so spricht man vom aktiven Lernen.

Zunächst werfen wir ein Blick auf passives Lernen. Kennt der Agent die optimale Strategie bereits, um ein Problem zu lösen bedeutet das, dass die Policy π bekannt ist. Somit führt er für einen Zustand s immer die Aktion $\pi(s)$ aus. Es gilt herauszufinden, wie sinnvoll und nützlich dies ist. Die Funktion die dies ermittelt nennt man Utility-Funktion $U^{\pi}(s)$. Diese Funktion U gilt es zu finden. Algorithmen, die dies ermöglichen sind zum Beispiel Adaptive dynamic programming (ADP) und Temporal difference learning (TD-learning).

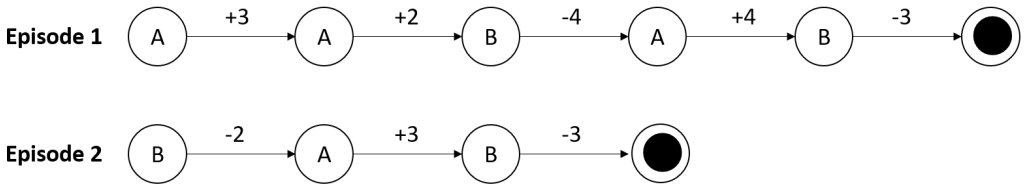
Die zweite Kategorie ist die Gruppe der Probleme, bei denen der Agent zunächst eine Strategie für die gewünschte Aktion ermitteln muss. Er kennt die optimale Policy π nicht. Er muss also in jedem Zustand selbst entscheiden, was die beste nächste Aktion ist. So entsteht ein Modell, dass er immer weiter verbessert, um in Zukunft größere Belohnungen zu erhalten. Dabei kann es passieren, dass er annimmt einen optimalen Weg gefunden zu haben und diesen immer wieder geht. Andere vielleicht genauso gute oder sogar bessere Zustände und Optionen werden nicht mehr betrachtet. Dieses Problem bezeichnet man auch als „exploitation vs. exploration“. Bei der exploitation (dt. „Ausbeutung“) macht der Agent das zuvor beschriebene: er nutzt Wege, von denen er gelernt hat, dass sie gut sind und optimiert sie gegebenenfalls weiter. Exploration hingegen bezeichnet die Erkundung von neuen Zuständen. Versucht der Agent nur neue Zustände zu besuchen und neue Wege zu analysieren, ist dies auch nicht zielführend. Denn was nützt es, wenn man einen guten Weg kennt, aber ihn die nutzt? Der Agent muss also einen Kompromiss zwischen exploitation und exploration eingehen, um sowohl die kurzfristige als auch langfristige Belohnung zu maximieren. Später wird dafür der Wert ϵ eingeführt werden. In ϵ-Prozent der Fälle geht der Agent dann neue Wege. In allen anderen Fällen versucht er die bekannten Wege zu gehen und zu lernen. Algorithmen mit denen dies möglich ist sind zum Beispiel ADP mit einer zusätzlichen exploration-Funktion oder der gleich vorgestellte Q-learning-Algorithmus. (vgl. Russell und Norvig 2010)

Bei unserem Speditions-Beispiel handelt es sich gemäß der zuvor erläuterten Theorie um aktives lernen. Denn unser Agent, der LKW, hat von einem Feld ausgehend häufig viele Optionen in welche Richtung er fährt oder ob er Ware abliefert oder einsammelt. Er weiß, welche Aktionen in welchem Zustand möglich sind, kennt aber ohne Lernvorgänge nicht die Aktion, die am schnellsten zum Ziel führt.

3.4 Monte-Carlo-Methode

Die Monte-Carlo-Methode ist eine Methode zur Ermittlung der zu erwartenden Belohnung beim Reinforcement Learning. Das besondere bei Monte-Carlo ist, dass es sich um einen randomisierten Algorithmus handelt. Als Voraussetzung werden Erfahrungswerte, also Trainings-

daten bestehend aus einer Folge von Zuständen, Aktionen und Belohnungen benötigt. Die Idee ist, aus diesen Daten den durchschnittlich erreichten Wert nach besuchen eines Zustands s zu ermitteln. Dazu gibt es zwei Ansätze: „Every-Visit“ und „First-Visit“. Diese sollen mit Hilfe der beiden folgenden Episoden erklärt werden (Quelle: Eigene Darstellung nach Gupta 2020).



Bei „First-Visit“ werden alle Belohnungen, die zwischen dem ersten Besuch des betrachteten Zustands s und dem Endzustand auftreten, aufsummiert. Die so berechneten Werte für alle Episoden werden aufaddiert und durch die Anzahl der betrachteten Episoden geteilt.

Für unsere Beispiel-Episoden bedeutet dies:

Für den Wert von Zustand A:

$$\begin{aligned} V_1(A) &= 3+2+(-4)+4+(-3) = 2 \\ V_2(A) &= 3+(-3) = 0 \\ V(A) &= \frac{V_1(A)+V_2(A)}{2} = \frac{2+0}{2} = 1 \end{aligned} \tag{3.1}$$

Für den Wert von Zustand B:

$$\begin{aligned} V_1(B) &= (-4)+4+(-3) = -3 \\ V_2(B) &= (-2)+3+(-3) = -2 \\ V(B) &= \frac{V_1(B)+V_2(B)}{2} = \frac{(-3)+(-2)}{2} = -2,5 \end{aligned} \tag{3.2}$$

Bei „Every-Visit“ wird ein neuer Summations-Term für jedes Auftreten des Zustands s erstellt. Das bedeutet man summiert zunächst alle Belohnung zwischen dem ersten Auftreten und dem Endzustand, addiert dazu die Summe der Belohnungen zwischen dem zweiten Auftreten und dem Endzustand und so weiter bis zur Addition der Summe der Belohnungen zwischen dem letzten Auftreten und dem Endzustand. Dies wird für alle Episoden wiederholt, aufaddiert und durch die Anzahl der Summations-Terme, also die Anzahl des Auftretens des Zustands s, dividiert.

Für unsere Beispiel-Episoden bedeutet dies:

Für den Wert von Zustand A:

$$\begin{aligned} V_1(A) &= (3+2+(-4)+4+(-3))+(2+(-4)+4+(-3))+(4+(-3)) = 2+(-1)+1 = 2 \\ V_2(A) &= 3+(-3) = 0 \\ V(A) &= \frac{V_1(A)+V_2(A)}{3+1} = \frac{2+0}{4} = 0,5 \end{aligned} \tag{3.3}$$

Für den Wert von Zustand B:

$$\begin{aligned} V_1(B) &= ((-4)+4+(-3))+(-3) = -3+(-3) = -6 \\ V_2(B) &= ((-2)+3+(-3))+(-3) = -2+(-3) = -5 \\ V(B) &= \frac{V_1(B)+V_2(B)}{2+2} = \frac{(-6)+(-5)}{4} = -2,75 \end{aligned} \tag{3.4}$$

(vgl. Gupta 2020, Sutton 2015)

In der Anwendung von Monte-Carlo wird eine Episode von einem zufälligen Zustand bis zum Endzustand mit zufälligen Aktionen durchgespielt und dabei Zustände, Aktionen und Belohnungen aufgezeichnet. Anschließend werden die Werte der Zustände nach einer der oben erläuterten Methode berechnet. Anschließend wird ein neuer zufälliger Zustand bestimmt und unter Ausnutzung der zuvor berechneten Werte bis zum Ende geführt. Die Werte werden nach dem Durchgang neu berechnet und gemittelt. Dies wird viele Male wiederholt. So ergibt sich die gewünschte Werte-Matrix Q.

Bei diesem Algorithmus ist nicht sichergestellt, dass jeder Zustand nach dem Training besucht wurde. Ein weiterer Nachteil ist, dass das Problem den Endzustand erreichen muss, bevor die Werte berechnet werden können. Es muss sich also zwingend um terminierende Probleme handeln.

3.5 Implementierung 4

Für die Monte-Carlo-Methode wird neben der Werte-Tabelle auch noch eine Tabelle benötigt, die zählt, welche Aktion in welchem Zustand wie oft aufgerufen wurde.

```
mc_table = np.zeros([len(states), len(actions)])
count_state_action = np.zeros([len(states), len(actions)])
```

Die Funktion `mc_function` passt nach abschließen einer Episode die Werte in der Tabelle an. Dafür wird das eben vorgestellte „First-Visit“-Prinzip genutzt.

```
def mc_function(state_list):
    states = [s for (s, _,_) in state_list]
    state_action_pairs = {(s, a) for (s, a, _) in state_list}
```

```
    for (state, action) in state_action_pairs:
        idx = states.index(state)
        value = 0
        for i in range (idx, len(state_list)):
            value += state_list[i][2]
        state_num = state_to_state_num(state)
        count_state_action[state_num][action] += 1
        mc_table[state_num][action] = (mc_table[state_num][action] + value)/
↪count_state_action[state_num][action]
```

`mc_learning` berechnet die Werte-Tabelle mit einer zu übergebenden Anzahl an Episoden-Durchläufen. Dabei wird das Exploitation und Exploration-Prinzip verwendet. Bei der Exploitation wird der beste Action-Wert für den Zustand aus der Taabelle genommen. Da am Anfang alle Werte gleich sind, würde immer Aktion 0 gewählt werden. Um dies auszugleichen wurde eine Funktion `best_actions` geschrieben, die alle besten Aktionen für einen Zustand zurückgibt. Aus diesen wird anschließend zufällig eine Aktion ausgewählt.

```
def best_actions(table, state_num):
    return [i[0] for i in np.argwhere(table[state_num] == np.
↪amax(table[state_num]))]
```

```
def mc_learning(episodes):
    epsilon = 0.3

    for episode in tqdm(range(episodes)):
        state = states[random.randrange(0, len(states), 1)]
        done  = False
        state_list = []
        transport  = Transport(state, None, stepwise = False, visualize =␣
↪False)

        while not done:
            state_num = state_to_state_num(transport.state)
            if random.uniform(0, 1) < epsilon:
                action = random.sample(possible_actions[transport.state[0]],␣
↪1)[0] # Aktionsbereich erkunden
            else:
                b = best_actions(mc_table, state_num)
                action = random.sample(b, 1)[0] # Gelernte Werte ausnutzen
            reward     = transport.reward_function(action)
            transport.state = transport.transition_function(action)
            state_list.append((transport.state, action, reward))
```

```
            _, position_goods, position_goal = transport.state
            if position_goods == position_goal: done = True
        mc_function(state_list)
```

Achtung, die Ausführung des nächsten Befehls dauert länger.

Zum Trainieren muss die Funktion `mc_learning()` mit einem relativ hohen Wert aufgerufen werden. Damit dies nicht nach jedem Neustart des Kernels notwendig ist, kann die Tabelle abgespeichert und beim nächsten Mal einfach geladen werden.

```
#mc_learning(1000) # Trainieren
#np.savetxt('mc_table.txt', mc_table) # Speichern der Tabelle
mc_table = np.loadtxt('mc_table.txt') # Laden der Tabelle
```

Die folgende Methode kann dann nach dem Trainieren in der tatsächlichen Anwendung genutzt werden, um für jeden Zustand die beste Aktion auf Basis der zuvor berechneten Werte zu ermitteln. Dazu muss sie der Klasse `Transport` bei der Initialisierung übergeben werden und wird später dann von `transport_goods()` aufgerufen.

```
def mc_action(state_num):
    return np.argmax(mc_table[state_num])
```

Die Erkenntnis aus dieser Implementierung ist, dass die Monte-Carlo-Methode für das hier vorliegende Problem nicht sehr gut geeignet ist. Der Grund dafür ist, dass gerade die Durchläufe am Anfang sehr lange dauern, da die in der Tabelle gespeicherten Werte für jede Aktion identisch sind. Der Agent nimmt suboptimale Aktionen und versucht diese zu nutzen, gelangt jedoch nicht zum Ziel. Da sich die Werte erst verbessern, wenn die Episode terminiert, kann der Agent innerhalb einer Episode seinen Weg nicht optimieren und wieder schlechte und nicht zielführende Aktionen. Für eine gute Lösung des hier vorliegenden Problems eignen sich entsprechen Algorithmen besser, die eine Optimierung nach jedem Schritt ermöglichen.

3.6 Temporal-difference learning (TD-Learning)

Ähnlich wie bei Monte Carlo, wird Temporal-Difference-Learning zur Berechnung der Q-Matrix verwendet. Bei jedem Durchlauf entscheidet sich der Agent für die Aktion, dessen Folgezustand die höchsten Belohnung bereitstellt. Im Anschluss wird der Wert innerhalb der Q-Matrix mit Hilfe der Value-Funktion nach folgender Formel angepasst:

$V(s) \leftarrow V(s) + \alpha(R + \gamma V(s') - V(s))$

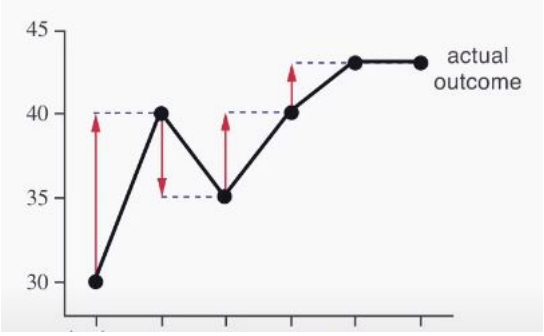
wobei gilt:

- $V(s)$ ist der Wert des aktuellen States
- $V(s')$ ist der Wert des nächsten States

- R ist der Reward, der im nächstem Schritt erhalten wird
- γ ist die Gewichtung, wie hoch der Einfluss des Wertes in den Gesamtwert eingeht (engl. Discount-Factor)
- α ist die Lernrate

Der Teil $R + \gamma V(s') - V(s)$ wir auch als TargetError bezeichnet.

Nach jeder Aktion des Agenten wird die Q-Matrix optimiert und V nähert sich V^{π} an. Im Gegensatz zu Monte Carlo wird die Matrix also nicht erst angepasst, wenn ein Reward erhalten wurde. Zudem werden die Rewards, mit einem Gewicht γ versehen. Dadurch wird aktuelleren Werten eine höhere Wertigkeit im Endergebnis gegeben. Der Vorteil dabei ist, dass die Varianz minimiert wird. Ausreißer haben also kleinere Auswirkungen auf das Endergebnis.



Auf der Abbildung (Quelle: Willcocks 2021) ist zu sehen, wie sich der Wert (gestrichelte Linie) nach jeder ausgeführten Aktion optimiert, wobei auf der x-Achse der zeitliche Ablauf und auf der y-Achse der Wert abgebildet ist.

(vgl. Kumar 2019, Willcocks 2021)

3.7 Q-learning

Q-learning ist ein modellfreier Reinforcement Learning Algorithmus, mit dem der Wert einer Aktion berechnet werden kann. Er ist eine Variante von TD-Learning und benötigt keine Anpassung des Environments, lediglich Rewards und stochastische Transitionen. Für einen endlichen Markov-Decision-Prozess ist Q-learning in der Lage eine Policy zu erstellen. Eine optimale Policy $\pi*$ auf Basis von zufälligen Aktionen zu erstellen, wäre mit Q-Learning möglich, falls unendlich viel Zeit zur Verfügung stünde.

In eine Matrix der Form $len(States) \times len(Aktionen)$ werden die Rewards eingetragen, die für eine Aktion erhalten werden, die in einem State ausgeführt wird. Diese Matrix wird im Code als `q_table` bezeichnet und ist initial mit Nullen oder niedrigen zufälligen Werten gefüllt. Die Q-Funktion: $Q : SxA \rightarrow \mathbb{R}$ mappt eine Aktion und einen State auf eine rationale Zahl, die in der Matrix an der Position `q_table[state][action]` eingetragen wird. Neue Werte werden mit einer höheren Gewichtung in die Matrix einberechnet. Die Gewichtung gibt der

Discount-Faktor γ an. In den Werten der Matrix sind die Werte der letzten States aufsummiert eingetragen, also die Rewards des ganzen Weges.

3.8 Implementierung 5

Zunächst wird eine Tabelle mit einer Zeile für jeden Zustand und einer Spalte für jede Aktion angelegt. In jede Zelle wird zu Beginn der Wert Null geschrieben.

```
q_table = np.zeros([len(states), len(actions)])
```

Die `q_function()` nimmt einen Zustand und eine Aktion sowie die Lernrate α und den Discount-Faktor γ. Darauf basierend berechnet sie mit Hilfe der `q_table` den neuen Wert für den Zustand und die Aktion und trägt diesen an der entsprechenden Stelle in der Tabelle ein. Zurückgegeben wird außerdem der nächste Zustand. Die Formel für die Berechnung des neuen Wertes lautet folgendermaßen:

$$Q_{\texttt{new}}(s_t, a_t) \leftarrow Q_{\texttt{old}}(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot max_q(st+1, a) - Q_{\texttt{old}}(s_t, a_t))$$

mit:

- $s_t, s_{t+1} \in$ len(states)
- a $\in$ actions
- $Q_{\texttt{old}}(s_t, a_t)$ = Tabellenwert in Zeile s_t und Spalte a_t
- α = Lernrate
- γ = Discount-Faktor
- r_t = Reward
- max_q = Schätzung des optimalen Zukunftswertes

Der Wert in der Klammer wird dabei auch als „temporal difference“ bezeichnet.

```
def q_function(state, action, alpha, gamma):
    transport  = Transport(state, stepwise = False, visualize = False)

    reward     = transport.reward_function(action)
    next_state = transport.transition_function(action)

    state_num      = state_to_state_num(state)
    next_state_num = state_to_state_num(next_state)

    old_value  = q_table[state_num][action]
```

```
    max_q      = np.max(q_table[next_state_num])
    new_value  = old_value + alpha * (reward + gamma * max_q - old_value)

    q_table[state_num, action] = new_value
    return next_state
```

Die Funktion `q_learning()` führt für eine zu übergebende Anzahl an Startzuständen (`episodes`) den Transport der Waren zum Ziel durch. Der Startzustand ist dabei ein zufälliger. Dies dient dem Training und somit der Verbesserung der `q_table`. Denn in jedem Schritt wird `q_function()` aufgerufen. In rund 10% (= `epsilon`) der Schritte ist auch die Aktion zufällig ausgewählt. In den anderen Fällen wird die am besten bewertete Aktion für den aktuellen Zustand genutzt.

```
def q_learning(episodes):
    alpha   = 0.1
    gamma   = 0.6
    epsilon = 0.1

    for episode in tqdm(range(episodes)):
        state = states[random.randrange(0, len(states), 1)]
        done  = False

        while not done:
            state_num = state_to_state_num(state)
            if random.uniform(0, 1) < epsilon:
                action = random.sample(possible_actions[state[0]], 1)[0] #␣
 ↪Aktionsbereich erkunden
            else:
                action = np.argmax(q_table[state_num]) # Gelernte Werte␣
 ↪ausnutzen

            state = q_function(state, action, alpha, gamma)
            _, position_goods, position_goal = state
            if position_goods == position_goal: done = True
```

Zum Trainieren muss die Funktion `q_learning()` mit einem relativ hohen Wert aufgerufen werden. Damit dies nicht nach jedem Neustart des Kernels notwendig ist, kann die Tabelle abgespeichert und beim nächsten Mal einfach geladen werden.

```
#q_learning(10000) # Trainieren
#np.savetxt('q_table.txt', q_table) # Speichern der Tabelle
q_table = np.loadtxt('q_table.txt') # Laden der Tabelle
```

Die folgende Methode kann dann nach dem Trainieren in der tatsächlichen Anwendung genutzt werden, um für jeden Zustand die beste Aktion zu ermitteln. Dazu muss sie der Klasse `Transport` bei der Initialisierung übergeben werden und wird später dann von `transport_goods()` aufgerufen.

```
def q_learning_action(state_num):
    return np.argmax(q_table[state_num])
```

4 Fazit

Ziel der Arbeit war es, die benötigten Grundlagen für Reinforcement Learning zu erklären und an Hand eines Fallbeispieles die Implementierung zu erläutern. Zu Beginn der Arbeit wurde im Rahmen des Markov-Decision-Prozess erläutert, was für Reinforcement Learning benötigt wird. Des Weiteren wurden verschiedene Arten und Vorgehensweisen des Reinforcement Learning aufgegliedert und erklärt. Die verschiedenen Algorithmen zum Lernen wurden theoretisch erläutert und teilweise am Beispiel angewandt. Nun soll eine Visualisierung des Problems, des Lernverfahrens und der Lösung zum besseren Verständnis implementiert werden. Abschließend werden die verschiedenen Lernmodelle evaluiert.

4.1 Visualisierung

Zunächst werden die bereits vorgenommenen Implementierungen geladen. Das Notebook, das im als letztes geladen wird, beinhaltet Funktionen für die grafische Darstellung des aktuellen Zustands.

```
%run ./02_markov_decision_process.ipynb
%run ./03_reinforcement_learning.ipynb
%run ./implementierung_gui.ipynb
```

Folgendermaßen kann dann eine Episode ausgeführt und visualisiert werden. Als Methoden für die Auswahl der Aktionen sind `random_action`, `mc_action` und `q_learning_action` möglich.

```
state = (((3, 0), 0, 2))
transport = Transport(state, action_method = q_learning_action, stepwise =␣
↪True, visualize = True)
transport.update_canvas()
```

4.2 Evaluation

Um die Performance des trainierten Agenten beurteilen zu können, wird für eine große Menge an zufälligen Startzuständen zur Auswahl der Aktionen einmal der zufällige Algorithmus genutzt und einmal jener, der auf den erlernten Werten basiert:

```
def evaluation(number_of_tests, action_method):
    total_value = 0
    total_steps = 0
```

```
    for i in tqdm(range (number_of_tests)):
        state = states[random.randrange(0, len(states), 1)]
        transport = Transport(state, action_method = action_method, stepwise␣
↪= False, visualize = False)
        transport.transport_goods()

        total_value += transport.current_value
        total_steps += transport.steps

    print('Result after ' + str(number_of_tests) + ' tests:')
    print('Average value: ' + str(total_value/number_of_tests))
    print('Average number of steps: ' + str(total_steps/number_of_tests))
    print('Average reward per step: ' + str((total_value/number_of_tests)/
↪(total_steps/number_of_tests)))
```

```
#evaluation(2000, random_action)
```

Ergebnis nach 2000 Durchläufen ohne Reinforcement Learning:

- Average value: -1404.6395
- Average number of steps: 890.9995
- Average reward per step: -1.5764761933087503

```
#evaluation(2000, mc_action)
```

Ergebnis nach 2000 Durchläufen mit Reinforcement Learning und 1.000 Trainingsdurchläufen mit der Monte-Carlo-Methode:

- Average value: -2685.706
- Average number of steps: 1797.876
- Average reward per step: -1.4938215983749714

```
#evaluation(2000, q_learning_action)
```

Ergebnis nach 2000 Durchläufen mit Reinforcement Learning und 10.000 Trainingsdurchläufen für das Q-Learning:

- Average value: 1.2275
- Average number of steps: 12.8825
- Average reward per step: 0.0952843004075296

Vergleicht man die hier ermittelten Werte der Evaluation, insbesondere die durchschnittliche

Belohnung pro Schritt, lässt sich eine Verbesserung von der zufälligen Auswahl der Aktionen, über die Monte-Carlo-Methode bis zur Nutzung der mittels Q-Learning berechneten Werte feststellen. Zu Beachten beim Vergleich von Monte-Carlo und Q-Learning ist, dass bei Q-Learning 10.000 Trainingsdurchgänge genutzt wurden, während es bei Monte-Carlo aus Zeitgründen nur 1.000 Iterationen gab. Das Ergebnis zeigt jedoch deutlich, dass sich das Trainieren gelohnt hat und sich die Anzahl der benötigten Schritte pro Episode bei der Verwendung von Q-Learning deutlich reduziert hat. Entsprechend ist die im Durchschnitt pro Schritt erhaltene Belohnung deutlich höher, wenn die Werte, die durch Q-Learning ermittelt wurden, genutzt werden. Die durchschnittliche Belohnung bei zufälligen Aktionen und bei Monte-Carlo liegt hingegen im negativen Bereich. Schaut man sich die Episoden in der Visualisierung an, sieht man auch, dass der mit Q-Learning trainierte Agent meistens den optimalen, also kürzesten, Weg nimmt. Eine weitere Optimierung des Algorithmus ist perspektivisch durch eine Anpassung der Werte α, γ und ϵ möglich. Außerdem wurde während der Implementierung bereits das Fazit gezogen, dass Monte Carlo keine effiziente Lösung des Problems zu Verfügung stellt, da Lernvorgänge auf Grund von suboptimalen Prozessen zu lange dauern. Von den präsentierten Algorithmen ist Q-Learning für diesen Anwendungsfall zu präferieren.

5 Funktionen für die Visualisierung

Zur Darstellung der Stadt wird ipycanvas verwendet. Die Label für die informativen Angaben werden von ipywidgets genutzt.

```
import ipywidgets
from ipywidgets import Image
import ipycanvas
from ipycanvas import MultiCanvas
```

Für die Größenverhältnisse in der Darstellung werden einige Konstanten definiert:

- CANVAS_WIDTH = Breite und Höhe der Karte
- BORDER_WIDTH = Breite des Grünstreifens am Rand der Karte
- CELL_WIDTH = Breite und Höhe der einzelnen Kartenfelder

Anschließend werden die Bilder die die Stadt, den LKW, die Waren und das Ziel symbolisieren geladen

```
# Konstanten
CANVAS_WIDTH = 400
BORDER_WIDTH = 20
CELL_WIDTH   = (CANVAS_WIDTH - 2 * BORDER_WIDTH) / 6

# Bilder laden
city  = Image.from_file('img/city.png')
truck = Image.from_file('img/truck.png')
goods = Image.from_file('img/goods.png')
goal  = Image.from_file('img/goal.png')
```

Die Funktion `init_canvas()` wird zu Beginn aufgerufen und nimmt einen Zustand und stellt diesen Startzustand grafisch dar. Dazu wird eine Leinwand mir vier Ebenen erstellt. Eine für die Karte im Hintergrund, eine für die Zielflagge, eine für den LKW und eine für das Warenpaket. Die Aufteilung auf verschiedene Ebenen hat den Vorteil, das bei der Veränderung von einer Ebene die anderen nicht neu geladen werden müssen. Da sich die Karte und das Ziel nicht verändern, werden die Ebenen 0 und 1 nur am Anfang einmal generiert. So wirkt das Bild stabiler.

Die Positionsnummern von Waren und Ziel werden in Koordinaten und diese schließlich in absolute Positionen auf der Leinwand umgerechnet um danach gezeichnet zu werden.

```
def init_canvas(self):
    #canvas[Stadt, Ziel, LKW, Waren]
    canvas = MultiCanvas(4, width = CANVAS_WIDTH, height = CANVAS_WIDTH)
    _, position_goods, position_goal = self.state

    position_goods_label.value = 'Position goods: ' +␣
 ↪position_goods_descriptions[position_goods]
    goal_label.value           = 'Goal: ' +␣
 ↪stations_descriptions[position_goal]

    goal_x_coordinate  = stations[position_goal][1]
    goal_y_coordinate  = stations[position_goal][0]
    goal_x_position  = BORDER_WIDTH + CELL_WIDTH * goal_x_coordinate
    goal_y_position  = BORDER_WIDTH + CELL_WIDTH * goal_y_coordinate

    if position_goods != 4:
        goods_x_coordinate = stations[position_goods][1]
        goods_y_coordinate = stations[position_goods][0]
        goods_x_position = BORDER_WIDTH + CELL_WIDTH * goods_x_coordinate
        goods_y_position = BORDER_WIDTH + CELL_WIDTH * goods_y_coordinate

    canvas[0].draw_image(city, 0, 0, CANVAS_WIDTH, CANVAS_WIDTH)
    canvas[1].draw_image(goal,  goal_x_position,  goal_y_position, ␣
 ↪CELL_WIDTH, CELL_WIDTH)
    if position_goods != 4:
        canvas[3].draw_image(goods, goods_x_position, goods_y_position,␣
 ↪CELL_WIDTH, CELL_WIDTH)

    return canvas
Transport.init_canvas = init_canvas
del init_canvas
```

Die Funktion `show_start_button()` zeichnet auf dem Stadtplan ein Rechteck, dass das Wort „Start“ enthält und optisch einem Button ähnelt.

```
def show_start_button(self):
    with ipycanvas.hold_canvas(self.canvas):
        # Hintergrund
        self.canvas[2].clear()
        self.canvas[2].fill_style = 'black'
        self.canvas[2].global_alpha = 0.5
        self.canvas[2].fill_rect(0, 0, CANVAS_WIDTH, CANVAS_WIDTH)
```

```
        # Button
        self.canvas[2].global_alpha = 1
        self.canvas[2].shadow_color = 'black'
        self.canvas[2].shadow_offset_x = 5
        self.canvas[2].shadow_offset_y = 5
        self.canvas[2].shadow_blur = 5
        self.canvas[2].fill_style = 'silver'
        self.canvas[2].fill_rect(CANVAS_WIDTH/3, (CANVAS_WIDTH/5)*2,␣
↪CANVAS_WIDTH/3, CANVAS_WIDTH/5)

        self.canvas[2].shadow_offset_x = 0
        self.canvas[2].shadow_offset_y = 0
        self.canvas[2].shadow_blur = 0

        self.canvas[2].stroke_style = 'black'
        self.canvas[2].stroke_rect(CANVAS_WIDTH/3, (CANVAS_WIDTH/5)*2,␣
↪CANVAS_WIDTH/3, CANVAS_WIDTH/5)

        # Text
        self.canvas[2].fill_style = 'black'
        self.canvas[2].font = '30px Arial'
        self.canvas[2].fill_text('Start', CANVAS_WIDTH/2 *0.83, CANVAS_WIDTH/
↪2*1.05)
Transport.show_start_button = show_start_button
del show_start_button
```

`visualize_state` zeichnet die Ebenen, die die Waren und den LKW enthalten neu. Wobei die Ebene 3 der Waren nur erneuert wird, falls die Aktion eine Fahrt darstellt, da nur in diesem Fall das Paket tatsächlich bewegt wird.

```
def visualize_state(self):
    # calculating coordinates
    redraw_goods = True
    position_truck, position_goods, position_goal = self.state
    truck_x_coordinate, truck_y_coordinate = position_truck[1],␣
↪position_truck[0]
    if position_goods != 4:
        if position_goods == self.position_goods_old:
            redraw_goods = False
        self.position_goods_old = position_goods
```

```
        goods_x_coordinate, goods_y_coordinate =␣
↪stations[position_goods][1], stations[position_goods][0]
    else:
        goods_x_coordinate, goods_y_coordinate = truck_x_coordinate,␣
↪truck_y_coordinate

    # calculating positions
    truck_x_position = BORDER_WIDTH + CELL_WIDTH * truck_x_coordinate
    truck_y_position = BORDER_WIDTH + CELL_WIDTH * truck_y_coordinate
    goods_x_position = BORDER_WIDTH + CELL_WIDTH * goods_x_coordinate
    goods_y_position = BORDER_WIDTH + CELL_WIDTH * goods_y_coordinate

    # drawing images
    with ipycanvas.hold_canvas(self.canvas):
        self.canvas[2].clear()
        self.canvas[2].draw_image(truck, truck_x_position, truck_y_position,␣
↪CELL_WIDTH, CELL_WIDTH)
        if redraw_goods:
            position_goods_label.value = 'Position goods: ' +␣
↪position_goods_descriptions[position_goods]
            self.canvas[3].clear()
            self.canvas[3].draw_image(goods, goods_x_position,␣
↪goods_y_position, CELL_WIDTH, CELL_WIDTH)
Transport.visualize_state = visualize_state
del visualize_state
```

Im Folgenden werden Label für die Anzeige der aktuellen Schrittanzahl, des aktuellen Werts, der Position der Waren, des Zielorts, der erfolgten Aktion und die Anzahl der falschen bzw. unmöglichen Aktionen initialisiert.

```
step_label           = ipywidgets.Label(value = 'Step: ')
value_label          = ipywidgets.Label(value = 'Current value: ')
position_goods_label = ipywidgets.Label(value = 'Position goods: ')
goal_label           = ipywidgets.Label(value = 'Goal: ')
action_label         = ipywidgets.Label(value = 'Action: ')
```

Die Funktion `update_canvas()` wird nach jeder Aktion aufgerufen und aktualisiert die Statusanzeigen so wie die grafische Darstellung des aktuellen Zustands.

```
def update_canvas(self):
    self.canvas[2].clear()
    step_label.value = 'Step: ' + str(self.steps)
```

```
    value_label.value = 'Current value: ' + str(self.current_value)
    one_below_other = ipywidgets.VBox([step_label, value_label,␣
↪position_goods_label, goal_label, action_label])
    side_by_side = ipywidgets.HBox([self.canvas, one_below_other])
    self.visualize_state()
    display(side_by_side)

    if not self.started:
        self.show_start_button()
Transport.update_canvas = update_canvas
del update_canvas
```

`handle_mouse_down()` wird durch Klicken auf die Karte aufgerufen. Die Funktion ruft lediglich `transport_goods()` auf.

```
def handle_mouse_down(self, x, y):
    if not self.started:
        self.started = True
        self.transport_goods()
Transport.handle_mouse_down = handle_mouse_down
del handle_mouse_down
```

6 Literaturverzeichnis

Abbeel, Pieter; Klein, Dan (2014 a): Markov Decision Processes. CS 188: Artificial Intelligence. University of California, Berkeley, 20.08.2014. Online verfügbar unter http://ai.berkeley.edu/slides/Lecture%208%20–%20MDPs%20I/SP14%20CS188%20Lecture%208%20–%20MDPs%20I.pptx, zuletzt geprüft am 27.05.2021.

Abbeel, Pieter; Klein, Dan (2014 b): Reinforcement Learning. CS 188: Artificial Intelligence. University of California, Berkeley, 20.08.2014. Online verfügbar unter http://ai.berkeley.edu/slides/Lecture%2010%20–%20Reinforcement%20Learning%20I/SP14%20CS188%20Lecture%2010%20–%20Reinforcement%20Learning%20I.pptx, zuletzt geprüft am 27.05.2021.

Gupta, Mehul (2020): Monte Carlo for Reinforcement Learning with example. In: Data Science in your pocket, 04.03.2020. Online verfügbar unter https://medium.com/data-science-in-your-pocket/monte-carlo-for-reinforcement-learning-with-example-1754439dd628, zuletzt geprüft am 06.06.2021.

Kumar, Vaibhav (2019): Reinforcement learning: Temporal-Difference, SARSA, Q-Learning & Expected SARSA in python. In: Towards Data Science, 20.03.2019. Online verfügbar unter https://towardsdatascience.com/reinforcement-learning-temporal-difference-sarsa-q-learning-expected-sarsa-on-python-9fecfda7467e, zuletzt geprüft am 08.06.2021.

Russell, Stuart J.; Norvig, Peter (2010): Artificial intelligence. A modern approach. 3. ed. Upper Saddle River, NJ: Prentice-Hall (Prentice-Hall series in artificial intelligence).

Silver, David (2015): RL Course. Lecture 2: Markov Decision Process. Online verfügbar unter https://www.youtube.com/watch?v=lfHX2hHRMVQ&list=PLqYmG7hTraZDM-OYHWgPebj2MfCFzFObQ&index=3, zuletzt aktualisiert am 11.06.2021.000Z, zuletzt geprüft am 11.06.2021.125Z.

Sutton, R. S.; Barto, A. G. (2015): RL Lecture 5. Monte Carlo Methods. Online verfügbar unter http://www-edlab.cs.umass.edu/cs689/lectures/RL%20Lecture%205.pdf, zuletzt geprüft am 06.06.2021.

Willcocks, Chris G. (2021): Reinforcement Learning. 6: Temporal-difference methods. Reinforcement Learning. Durham University. Online verfügbar unter https://www.youtube.com/watch?v=phgI_880uSw, zuletzt geprüft am 09.06.2021.582Z.