Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Stefan Sylvius Wagner

Entwicklung eines Reinforcement Learning basierten Flugzeugautopiloten unter der Verwendung von Deterministic Policy Gradients

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Stefan Sylvius Wagner

Entwicklung eines Reinforcement Learning basierten Flugzeugautopiloten unter der Verwendung von Deterministic Policy Gradients

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Andreas Meisel
Zweitgutachter: Prof. Dr. Wolfgang Fohl

Eingereicht am: 28. Februar 2018

Stefan Sylvius Wagner

Thema der Arbeit

Entwicklung eines Reinforcement Learning basierten Flugzeugautopiloten unter der Verwendung von Deterministic Policy Gradients

Stichworte

Deterministic Policy Gradients, DDPG, DQN, Neuronale Netze, Reinforcement Learning, kontinuierliche Kontrolle, autonomer Flugzeugautopilot

Kurzzusammenfassung

Einer der schwierigsten Aufgaben im Reinforcement Learning ist die Regelung von Systemen in einem kontinuierlichen Zustandsraum und die anschließende Steuerung in einem kontinuierlichen Aktionsraum. In dieser Arbeit wird ein Reinforcment Learning basierter Flugzeugautopilot konzipiert und implementiert, der einen kontinuierlichen Zustandsraum approximiert und ein Flugzeug mit Aktionen in einem kontinuierlichen Wertebereich steuert. Deterministic Policy Gradients bieten ein spezialisiertes Framework in Form einer Actor-Critic Architektur, die in der Lage ist aus einem kontinuierlichen Zustandsraum, kontinuierliche Aktionswerte zu ermitteln. Dieses Framework wird im Zusammenhang mit einer Belohnungsfunktion, die Feedback über das Verhalten des Autopiloten liefert implementiert. Um die Realisierbarkeit und Robustheit des Reinforcement Learning basierten Flugzeugautopiloten zu überprüfen werden unterschiedliche Szenarien erstellt, die anhand eines komerziellen Flugsimulators ausgeführt und anschließend statistisch analysiert werden.

Title of the paper

Development of a reinforcement learning based airplane autopilot using Deterministic Policy Gradients

Keywords

Deterministic Policy Gradients, DDPG, DQN, neural networks, reinforcement learning, continuous control, autonomous autopilot

Abstract

One of the most difficult challenges in reinforcement learning is the continuous control of systems in a continuous state and action space. This papers goal is to design and implement a reinforcement learning based airplane autopilot that controls an aircraft in continuous state and action space. Deterministic Policy Gradients define a framework for this purpose in the form of an actor-critic architecture that approximates a continuous action space and outputs a

continuous action vector. The framework is accompanied by the implementation of a reward function that provides the autopilot with behavioral feedback. Finally, the feasibility and robustness of the implemented autopilot is tested inside a commercial flight simulator. For this purpose multiple scenarios are defined and the resulting data evaluated through statistical methods.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1. Einleitung

In den letzten Jahren hat sich der Bereich des maschinellen Lernens rasant weiterentwickelt. Durch immer schneller werdende Hardware und der daraus resultierenden Software-Architekturen ist es mittlerweile möglich künstliche Intelligenzen zu entwerfen, die hoch komplexe Aufgaben lösen können. Reinforcement Learning ist ein Bereich des maschinellen Lernens, der in der Regelungstechnik und bei der Steuerung von dynamischen Systemen zunehmend an Bedeutung gewinnt. Dadurch öffnen sich Türen für die Bewältigung komplexer Regelungsaufgaben wie das Umsetzen einer Steuerung für autonome Fahrzeuge und Fluggeräte.

1.1. Motivation

Bisherige Erfolge im Reinforcement Learning wurden vor allem in der Steuerung von diskreten Aktionsräumen erzielt. Das in **Human-level control through Deep Reinforcement Learning** Mnih u. a. (2015) behandelte Deep-Q-Network von DeepMind war die erste Architektur, die eine dem Menschen ebenbürtige Steuerung ermöglichte. Es handelte sich hier, um die Steuerung eines Agenten in einem Atarispiel. Eine Problemstellung die eine Steuerung über einen diskreten Aktionsraum erfordert. Für die Steuerung über kontinuierliche Aktionsräume sind hingegen noch nicht viele Architekturen entwickelt worden. Abbildung 1.1 liefert einen grafischen Vergleich diskreter und kontinuierlicher Umgebungen. Kontinuierliche Aktionsräume schöpfen aus einem potenziell unendlichen Wertebereich, wodurch die Komplexität im Vergleich zu diskreten Aktionsräumen erheblich steigt. Mechanismen, die für diskrete Umgebungen Wirkung zeigen scheitern in kontinuierlichen Umgebungen kläglich. Erst durch die Einführung von **Deterministic Policy Gradients** in **Deterministic Policy Gradient Algorithms** Silver u. a. (2014) ist es gelungen erste hochdimensionierte Steuerungsprobleme zu lösen.

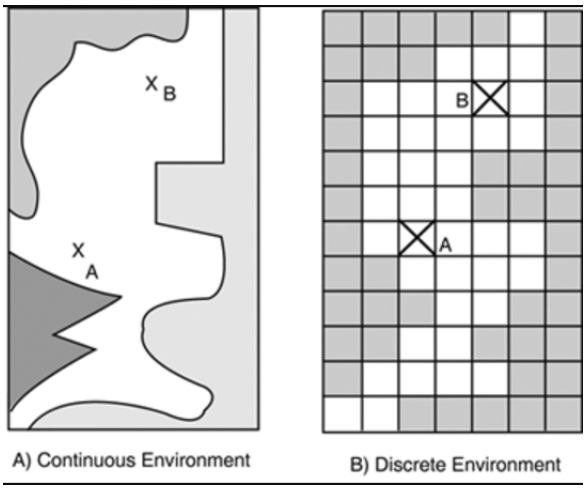


Abbildung 1.1.: **a)** kontinuierliche Umgebung **b)** diskrete Umgebung Champandard (2003)

Unter der Verwendung von Deterministic Policy Gradients ist es mittlerweile möglich Fahrsimulationen über einen kontinuierlichen Wertebereich zu steuern. Die Luftfahrt hatte bisher noch nicht allzu prominente Erfolge. In **Inverted Autonoumous Helicopter Flight via Reinforcement Learning** Ng u. a. (2006) wurden bereits erste Schritte in der autonomen Flugfahrt durchgeführt. In diesem Fall wurde anhand eines Reinforcement Learning Frameworks ein Autopilot für einen Helikopter entworfen, der versucht einen invertierten Flug aufrecht zu erhalten. Ein Autopilot gehört zu den komplexesten Aneinanderreihungen von Regelungssystemen überhaupt. Unter extremen Bedingungen kann dieser allerdings schnell auf seine Grenzen stoßen und ist in bestimmten Phasen, wie beim Abheben eines Flugzeuges nicht handlungsfähig. Reinforcement Learning basierte Systeme könnten hier Abhilfe schaffen. Durch immer realistischere Simulationen bietet sich nun die Möglichkeit auch für die Steuerung von Flugzeugen Reinforcement Learning anzuwenden. Architekturen wie Deterministic Policy Gradients sind zunehmend in der Lage Regelungssysteme zu modellieren, die die Steuerung eines Agenten in stochastisch stark schwankenden Umgebungen wie die eines Flugzeugs ermöglichen.

1.2. Aufgaben und Ziel

In dieser Arbeit soll mit Hilfe von Deterministic Policy Gradients ein Autopilotsystem entwickelt werden, das in der Lage sein soll ein Flugzeug in 3 verschiedenen Phasen des Fluges zu übernehmen:

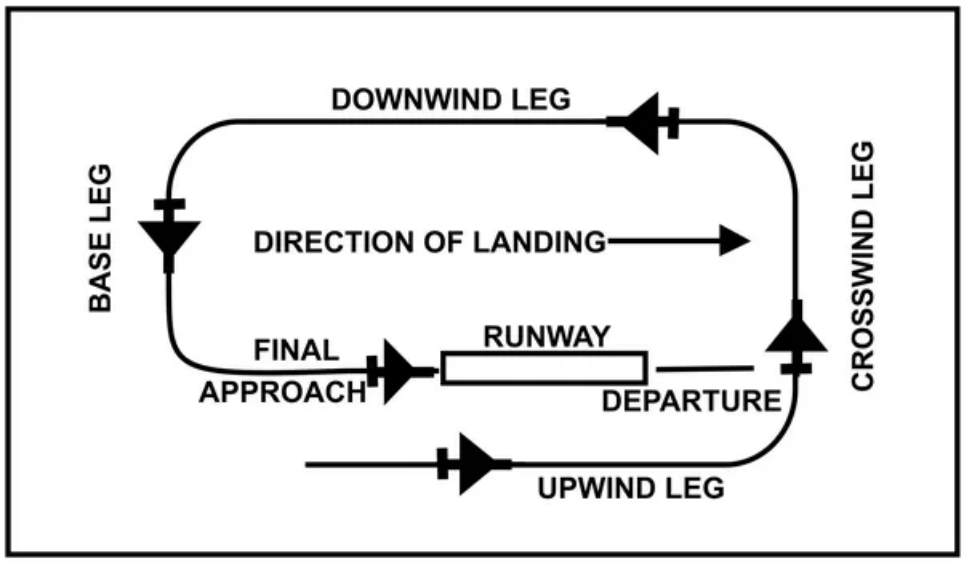


Abbildung 1.2.: Zu testende Szenarien (FAA)

1. **Start:** Besonderer Fokus wird auf die Startprozedur gelegt, da bis dato kein Autopilotsystem eine völlig automatische Startprozedur anbietet.
2. **Landung:** Der Autopilot soll unter extremen Wetterbedingunen in der Lage sein das Flugzeug zu landen. Hier soll der Autopilot die Steuerung im letzten Drittel des Landeanfluges übernehmen und das Flugzeug sicher landen.
3. **Extreme Bedingungen:** Es soll ein Katastrophenszenario, die die Flugfähigkeit des Flugzeugs beeinträchtig simuliert und dabei die Handlungsfähigkeit des Autopiloten analysiert werden.

Konkret soll der DDPG-Algorithmus in **Continuous control with deep reinforcement learning** Lillicrap u. a. (2015) umgesetzt werden, der für die Fahrsimulation TORCS bereits erfolgreich eingesetzt wurde.

Die Implementierung der Architektur muss in einem Machine Learning Framework erfolgen. Für diese Arbeit wurde Tensorflow als Framework ausgewählt.

Wichtiger Bestandteil der Arbeit ist das Erfassen relevanter Daten aus der Umgebung. Hieraus muss ebenfalls eine Belohnungsfunktion modelliert werden, die dem Agenten aus der Umgebung Feedback zum Lernen liefert.

Es ist unter Umständen notwendig den Zustandsraum intensiv zu sondieren, da sonst der lernende Agent in lokale Optima fällt. Um den lokalen Optima vorzubeugen kann eine gezielte Erforschungstrategie angewendet werden, um die Umgebung ausreichend zu erkunden und so lokale Optima zu vermeiden.

Letztendlich soll für jedes Szenario ein Autopilot trainiert werden. Diese sollen dann für die unterschiedlichen Szenarien eingesetzt werden können.

1.3. Gliederung der Arbeit

In Kapitel 2 werden die Grundlagen für der in dieser Arbeit behandelten Themen gelegt. Kapitel 2.1 führt die iterative Parameterbestimmung und neuronale Netze im Zusammenhang mit Supervised Learning ein. Kapitel 2.2 erläutert dann zunächst Grundlagen des Reinforcement Learnings, die für das spätere Verständnis der Deterministic Policy Gradients wichtig sind. Kapitel 2.3 behandelt schließlich die Deterministic Policy Gradients und den daraus entwickelten DDPG-Algorithmus. Im letzten Abschnitt der Grundlagen wird das Reward Shaping zur Modellierung einer Belohnungsfunktion zunächst auf einer konzeptuellen Ebene behandelt. Anschließend wird ein konkretes Konzept für die Modellierung einer Belohnungsfunktion vorgestellt.

In Kapitel 3 werden alle wesentlichen Themen zur Implementierung des Autopiloten behandelt. Als erstes werden die wichtigsten Frameworks und Tools, die für die Implementierung verwendet worden sind vorgestellt und das Konzept eines Flugzeugautopiloten genauer analysiert (Kapitel 3.1). Im nachfolgenden Kapitel 3.2 wird die Implementierung des autonomen Flugzeugautopiloten behandelt. Die Kommunikation mit dem Flugsimulator bzw. das Erfassen und Senden von Steuerdaten wird in Kapitel 3.2.1 behandelt. Danach widmet sich Kapitel 3.2.2 der konkreten Implementierung des DDPG-Algorithmus. Dabei wird die implementierte Softwarearchitektur des DDPG-Algorithmus in Tensorflow präsentiert. Anschließend folgt die konkrete Implementierung des DDPG-Algorithmus in Tensorflow. Das letzte Unterkapitel (Kapitel 3.2.3) beschäftigt sich mit der Modellierung und Implementierung der Belohnungsfunktion für den Flugzeugautopiloten.

In Kapitel 4 werden Versuchsdurchführung und Ergebnisse präsentiert. Vor der Vorstellung der Ergebnisse wird in Kapitel 4.1 zunächst die angewendete Trainingsmethodik für jedes

Szenario erläutert. In diesem Zusammenhang wird auch die Erforschungsstrategie des Agenten erläutert. In Kapitel 4.2 werden dann die unterschiedlichen Testszenarien erläutert und deren jeweiligen Bewertungskriterien. Als letztes werden die Ergebnisse präsentiert und interpretiert (Kapitel 4.3).

Eine Zusammenfassung der Arbeit wird in Kapitel 5 präsentiert. Dabei werden die einzelnen Teile der Arbeit nochmals mit den vorliegenden Ergebnissen analysiert und die gewonnenen Kentnisse zusammengefasst. Es wird versucht ein Fazit zu ziehen und Verbesserungen werden vorgeschlagen. Zu guter Letzt wird in Kapitel 6 ein Ausblick über weitere Optimierungen der arbeit und in die Zukunft des Reinforcement und Deep Learnings gegeben.

2. Grundlagen

2.1. Supervised Learning

Bevor die Grundlagen des Reinforcement Learnings erläutert werden ist es wichtig sich zuerst mit der bekanntesten Form des maschinellen Lernens zu befassen, dem Supervised Learning. Zwar ist das Ziel beider Lernformen recht unterschiedlich, dennoch ist deren Kernstruktur die das Lernen ermöglicht sehr ähnlich.

Im Supervised Learning versucht ein Agent anhand von vorgegebenen Musterbeispielen und durch die Anpassung diverser Parameter eine Heuristik herzuleiten, die sich an die Musterbeispiele anpasst. Dabei wird die Güte der Heuristik durch eine Kostenfunktion gemessen. Die Grundidee besteht darin diese Kostenfunktion zu minimieren, indem die oben genannten Parameter dementsprechend justiert werden. Die Information über die Anpassung der Parameter liefert der Gradient der Kostenfunktion. Durch das iterative Anpassen der Parameter in Richtung des negativen Gradienten passt sich der Agent immer besser an die vorliegenden Daten an. Der Vorgang, der dieses iterative Bestimmen der Parameter ermöglicht nennt sich iterative Parameterbestimmung. Dieses Verfahren wird im folgenden durch ein Anwendungsbeispiel näher erläutert.

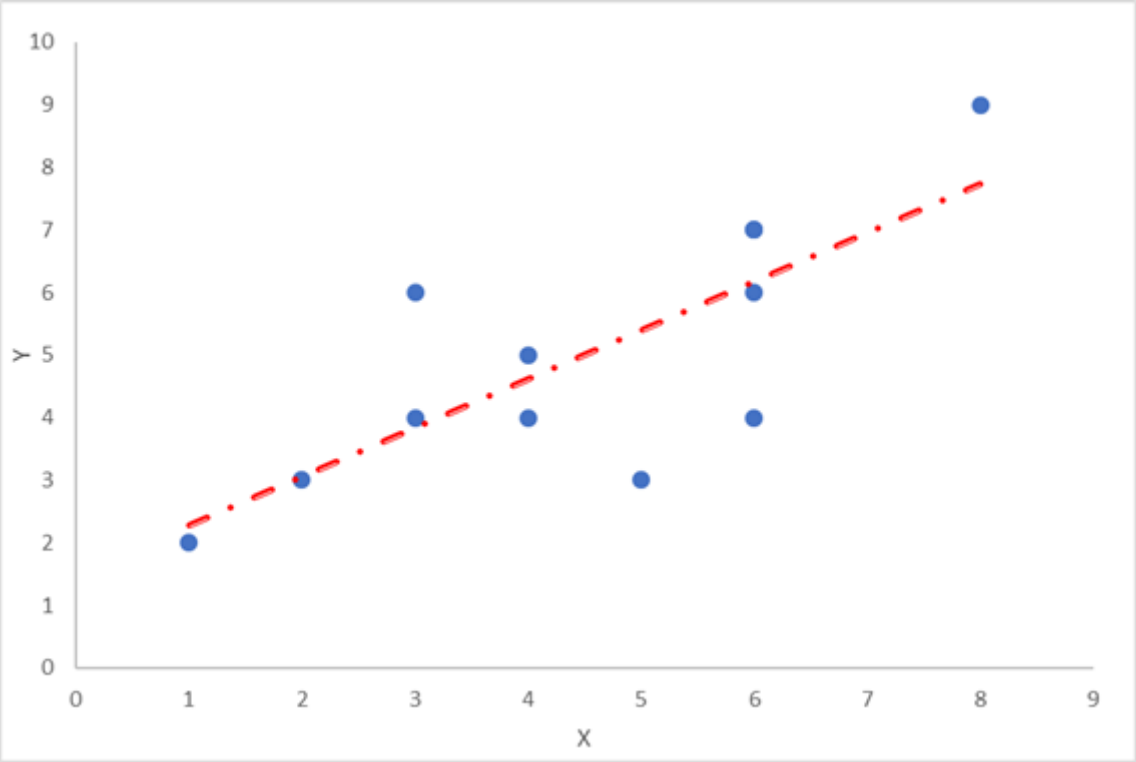


Abbildung 2.1.: Punktemenge P (Eigene Erstellung)

Für die Erläuterung der iterativen Parameterbestimmung bietet sich ein klassiches Regressionsproblem als Beispiel an: Das Fitting einer Gerade auf eine Punktemenge. Dieses Problem kann auch durch eine einfachere Ausgleichsrechnung bestimmt werden. Die iterative Parameterbestimmung wird eigentlich für das Approximieren von Funktionen mit einer hohen Parameteranzahl wie ein neuronales Netz verwendet. Nichtsdestotrotz, erlaubt uns dieses einfache Beispiel die einzelnen Komponenten der iterativen Parameterbestimmung unkompliziert zu erläutern. Anschließend werden anhand der erläuterten Grundlagen die einzelnen Komponenten der iterativen Parameterbestimmung nochmals für neuronale Netze vorgeführt.

Vorgegeben ist eine Menge von Punkten P an der eine Gerade der Form $y = mx + b$ angepasst werden soll (Abbildung 2.1). Eine Gerade der Form $y = mx + b$ wird durch die Parameter m und b bestimmt. Ziel ist eine Gerade mit Parametern m und b zu finden, die sich am besten an die Daten anpasst.

Die iterative Parameterbestimmung kann in 3 wesentliche Komponenten zusammengefasst werden:

- **Kostenfunktion**
- **Gradientenbestimmung**
- **Parameteranpassung**

Im Folgenden werden diese 3 Komponenten anhand des Beispiels näher erläutert.

Kostenfunktion

Eine Kostenfunktion definiert das Ziel nach dem die zu bestimmenden Parameter in einem Gleichungssystem angepasst werden sollen. So kann eine erzeugte Gerade nach jeder Parameteranpassung durch eine Kostenfunktion bewertet werden. Dazu wird in der Regel die mittlere quadratische Abweichung als Kostenfunktion verwendet:

$$\textbf{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - (mx_i + b))^2 \tag{2.1}$$

Der Mean Squared Error ermittelt die Abweichung zwischen einem Musterwert und einem Schätzwert. Dabei ist y_i der jeweilige Musterwert und $mx_i + b$ der jeweilige Schätzwert mit N-Anzahl an vorhandenen Werten. Der Mean Squared Error vergleicht die Distanz zwischen den Punkten der Punktemenge P und den Punkten, die durch die Gerade mit m und b erzeugt werden. Es wird also ein Maß dafür gegeben wie viel die durch m und b erzeugten Punkte von der vorhandenen Punktemenge P abweichen. Wenn Parameter m und b so gewählt worden sind, dass $MSE \approx 0$, dann liegen Schätzwert und Musterwert nah bei einander. Mit anderen Worten je niedriger der MSE desto besser passt sich die durch m und b definierte Gerade an die Punktemenge P an.

Gradientenbestimmung und Parameteranpassung

Der nächste Schritt besteht darin zu ermitteln wie die Parameter m und b modifiziert werden müssen, damit die Kostenfunktion einen möglichst niedrigen MSE-Wert liefert. Formal betrachtet sollen die Parameter m und b so modifiziert werden, dass die Kostenfunktion von **(2.2)** minimiert wird. Eine Möglichkeit dies zu realisieren ist das Bestimmen der Parameter durch **Gradient Descent**. Bei Gradient Descent geht es darum dem negativen Gradienten der Kostenfunktion zu folgen. Der Gradient kann in der einfachsten Form durch die partielle

Ableitung der Kostenfunktion mit den justierbaren Parametern ermittelt werden. In diesem Fall die Steigung m und y-Achsenabschnitt b der Geraden.

$$\vec{\nabla}\mathbf{MSE}(m,b) = \left(\frac{\partial \mathbf{MSE}}{\partial m}, \frac{\partial \mathbf{MSE}}{\partial b}\right)^T \tag{2.2}$$

Aus der partiellen Ableitung der jeweligen Parameter ergeben sich folgende Geradengleichungen, die den Gradienten beschreiben:

$$\frac{\partial \mathbf{MSE}}{\partial m} = \frac{2}{N}\sum_{i=1}^{N} -x_i(y_i - (mx_i + b)) \tag{2.3}$$

$$\frac{\partial \mathbf{MSE}}{\partial b} = \frac{2}{N}\sum_{i=1}^{N} -(y_i - (mx_i + b)) \tag{2.4}$$

Für beide Parameter m und b ergeben sich also für die Ermittlung der nächsten Parameterwerte folgende Aktualisierungsregeln:

$$m_{i+1} = m_i - \eta \cdot \left(\frac{\partial \mathbf{MSE}}{\partial m}\right) \tag{2.5}$$

$$b_{i+1} = b_i - \eta \cdot \left(\frac{\partial \mathbf{MSE}}{\partial b}\right) \tag{2.6}$$

Mit $\vec{\theta} = (\theta_1, \theta_2)$ und $\theta_1 = m, \theta_2 = b$:

$$\vec{\theta}_{n+1} = \vec{\theta}_n - \eta \cdot \vec{\nabla}\mathbf{MSE} \tag{2.7}$$

Abbildung 2.2 stellt die Kostenfunktion für unterschiedliche m und b Parameter graphisch dar. Die Idee besteht darin Parameter m und b so zu wählen, dass die Kosten auf ein Minimum kommen.

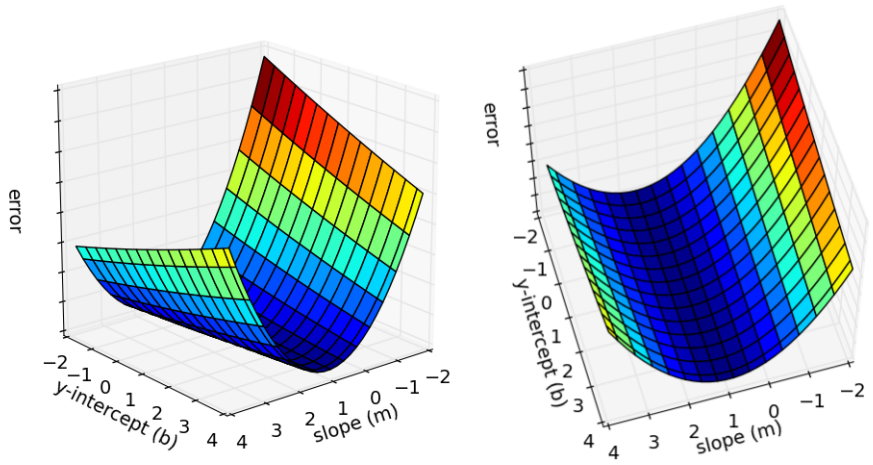


Abbildung 2.2.: Kostenfunktion für unterschiedliche m und b (Angepasst von: Hussain (2015))

Anhand der schwarzen Pfade ist zu sehen, dass Gradient Descent die Werte der Parameter in Richtung der niedriegsten Kosten wählt, also die Kostenfunktion minimiert.

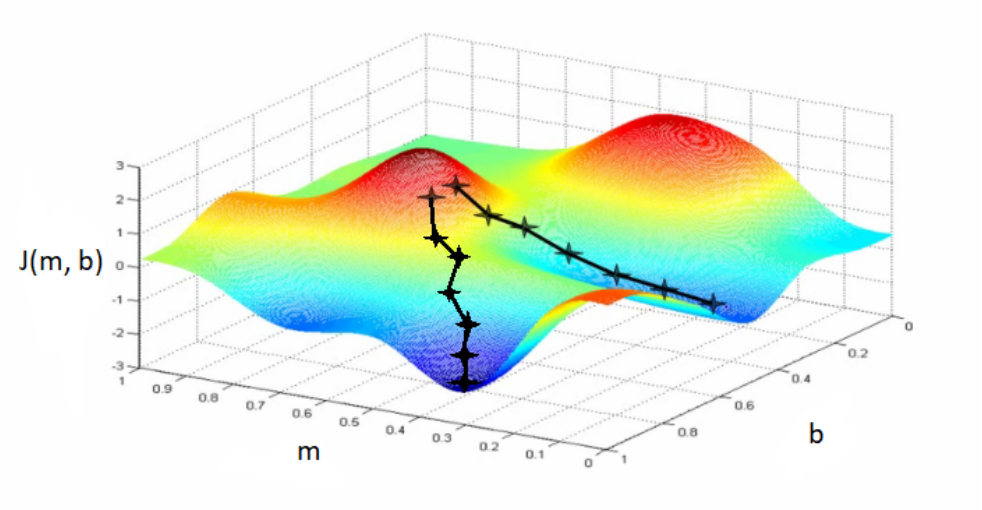


Abbildung 2.3.: Kostenfunktion für unterschiedliche m und b Roy (2015)

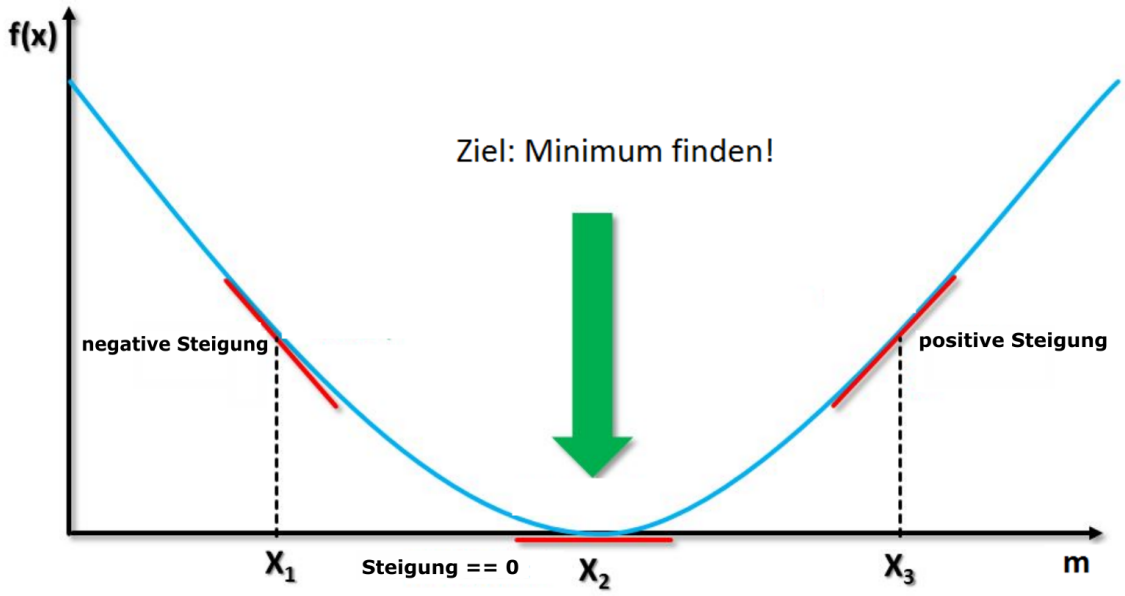


Abbildung 2.4.: Kurvendiskussion des Parameters m (Eigene Erstellung)

Die Richtung in der die Werte, der Parameter angepasst werden müssen, also der Gradient um Minima zu finden liefert wie erwähnt die erste partielle Ableitung der Kostenfunktion mit den jeweiligen Parametern. Abbildung 2.4 zeigt zunächst die Auswertung der partiellen Ableitung mit dem Parameter m des Parameterfeldes.

- Die Ableitungen an den Stellen x_1, x_2, x_3 liefern die Steigungen für die jeweiligen Stellen.
- Eine negative Steigung erfordert eine Anpassung des Parameters nach rechts(positiv), so dass die Kostenfunktion minimiert wird.
- Eine positive Steigung erfordert eine Anpassung der Parameter nach links(negativ), um die Kostenfunktion minimieren.
- Wenn die Ableitung an der Stelle $x == 0$ dann ist ein Minimum erreicht worden.

Wird dies für jeden einzelnen Parameter einer Funktion durchgeführt so ergibt dies den Gradienten einer Funktion welcher die partiellen Ableitungen einer Funktion mit dessen Parametern darstellt. Abbildungen 2.2 und 2.3 bieten eine grafische Darstellung des Gradientenfelds für Parameter m und b. Dabei repräsentieren die unterschiedlichen Farben die Größe des Gradienten für unterschiedliche Parameterwerte. In der Regel werden diese Gradientenfelder auch mit Pfeilen anstatt mit Farben dargestellt, um zusätzlich die Richtung des Gradienten

zu repräsentieren. Außer der Richtung repräsentiert die Größe der Pfeile das Ausmaß der Änderungsrate, wie in Abbildung 2.5 dargestellt.

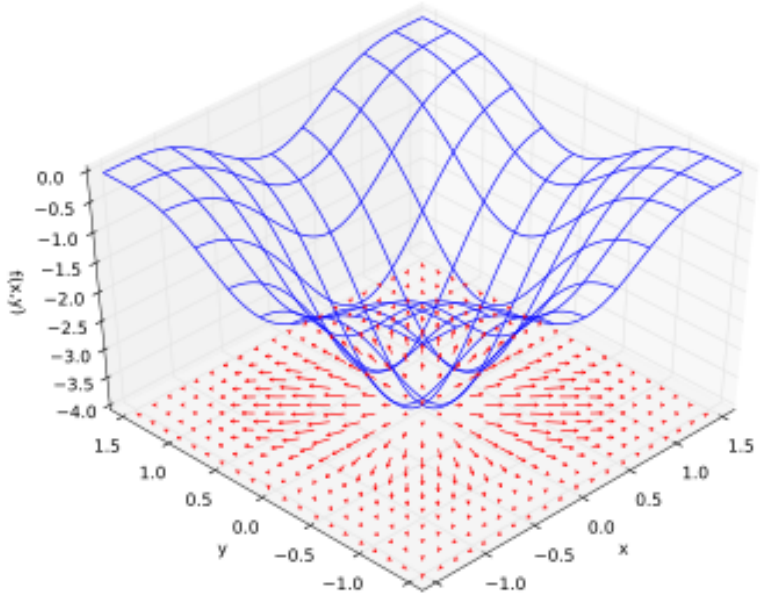


Abbildung 2.5.: Gradientenfeld (IkamusumeFan unter CC BY-SA 4.0 Lizenz)

Letztendlich werden die Parameter in die entgegengesetze Richtung der ermittelten Steigung für die jeweiligen Parameter angepasst. **Mit anderen Worten, die Parameter werden in die entgegengesetzte Richtung des Gradienten angepasst, um Minima der Kostenfunktion zu finden.**

Gradientenbestimmung und Parameteranpassung: Lernrate

In den meisten Fällen treten für eine Kostenfunktion mehrere Minima auf. Ziel ist es allerdings nicht nur ein lokales sondern auch ein globales Minimum zu finden. Das ausschließliche Anpassen der Parameter um den Gradienten kann allerdings dazu führen, dass nur ein lokales Minimum erreicht wird.

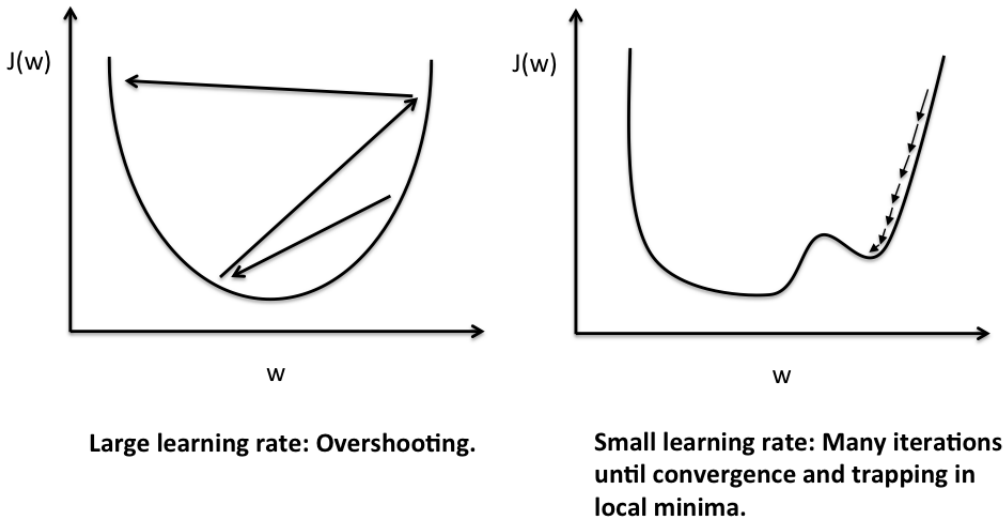


Abbildung 2.6.: Lernrate: lokale und globale Minima Ved (2016)

Um die Anpassung der Parameter zu steuern wird ein Schrittweitenfaktor η einfgeführt. Dieser wird mit dem Gradienten multipliziert und ermöglicht es so wie in Abbildung 2.6 dargestellt, die Anpassung der Parameter um den Gradienten zu steuern. Nichtsdestotrotz, kann selbst mit einem Schrittweitenfaktor kein globales Minimum garantiert werden. Wird die Schrittweite zu groß gewählt kann es zu Oszillationen kommen. Eine zu kleine Schrittweite erhöht hingegen das Risiko in einem lokalen Minimum stecken zu bleiben. Das Bestimmen der optimalen Schrittweite ist datenabhängig und muss daher experimentell ermittelt werden.

Supervised Learning mit neuronalen Netzen

Wie erwähnt ist die Anpassung einer Geraden an eine Datenmenge eine Aufgabe, die auch mit einfacheren Methoden gelöst werden kann. Neuronale Netze hingegen können aufgrund der hohen Parameteranzahl ausschließlich durch eine iterative Parameterbestimmung angepasst werden. Im Folgenden werden der Aufbau eines neuronalen Netzes und die Durchführung der iterativen Parameterbestimmung für neuronale Netze erläutert.

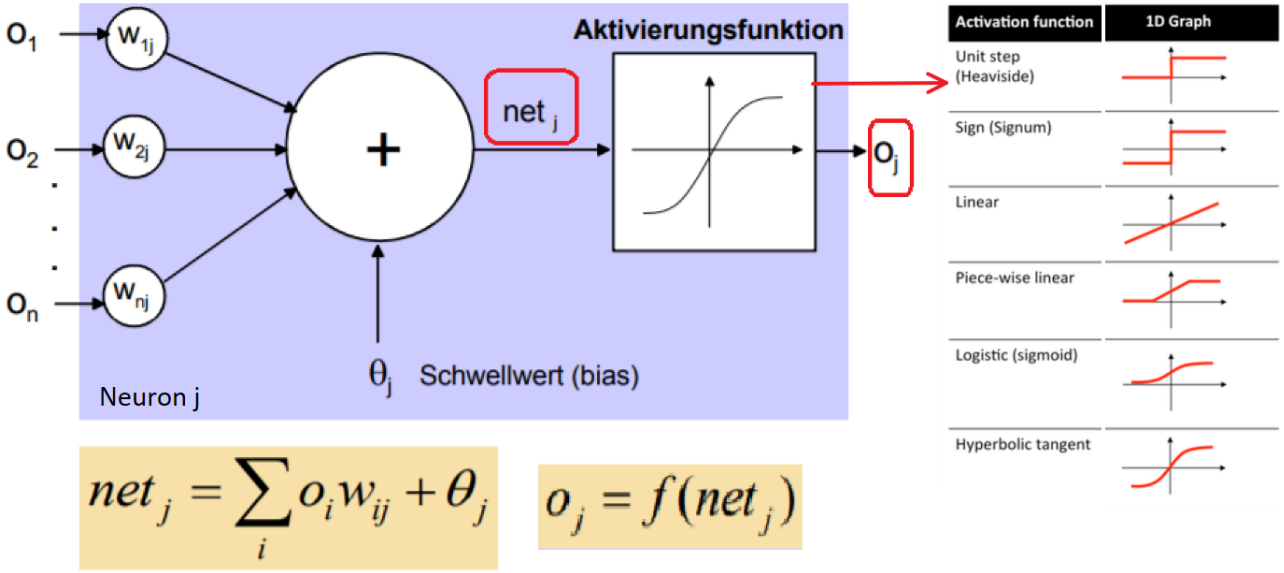


Abbildung 2.7.: Aufbau eines Neurons (Angepasst von: Meisel)

Abbildung 2.7 zeigt den grundlegenden Aufbau eines einzelnen Neurons. Neuronale Netze bestehen aus der Verknüpfung mehrer dieser Neurons. Die Eingänge eines Neurons werden durch die Parameter w_{ij} gewichtet. Dabei ist i die Nummer des Inputs und j die Nummer des Neurons. Diese Parameter werden in diesem Kontext auch Gewichte gennant. Zusätzlich wird ein Schwellwert(bias) θ_j definiert. Die Gewichte w_{ij} und Bias θ_j sind sogennante trainierbare Parameter. Diese werden anhand der iterativen Parameterbestimmung angepasst. Durch $\sum o_i w_{ij} + \theta_i$ resultiert ein Zwischenergebnis net_j. Die Nichtlinearität in neuronalen Netzen wird durch das Anwenden einer Aktivierungsfunktion $f(net_j)$ eingeführt. In der Regel sind Aktivierungsfunktionen nichtlineare Funktionen, die das Zwischenergebnis net_j nichtlinear transformieren.

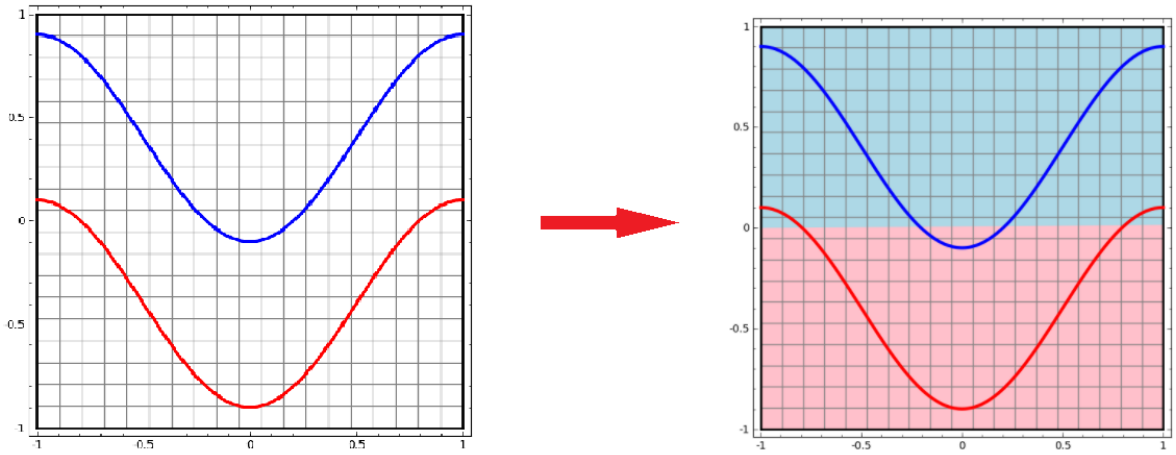


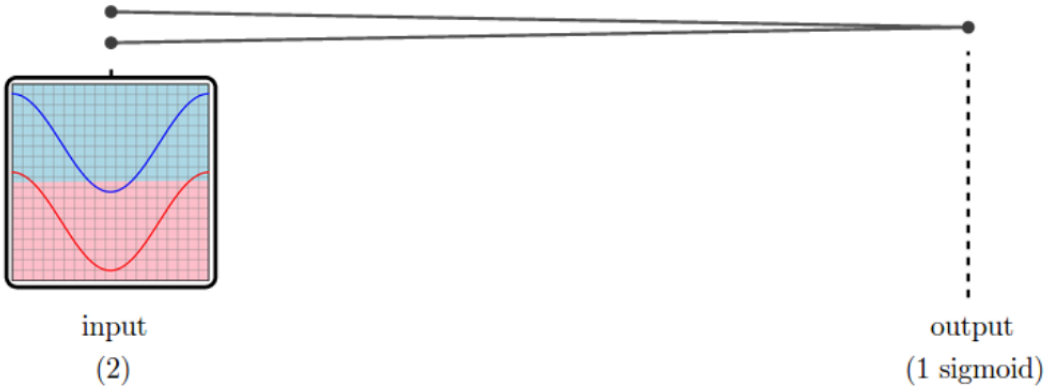
Abbildung 2.8.: Nicht linear separierbare Daten (Angepasst von: Olah (2014))

Die Fähigkeit neuronaler Netze nichtlineare Funktionen zu approximieren ist zwei Aspekten zuzuschreiben:

1. Das Anwenden einer nicht linearen Aktivierungsfunktion auf das Zwischenergebnis net_j.

2. Das Hinzufügen von Neuronschichten(verdeckte Schichten).

Abbildung 2.8 zeigt das Klassifizierungsverhalten eines einzelnen Neurons für zwei Kurven auf einer Ebene die nicht linear separierbar sind. Das heißt, es gibt keine lineare Funktion die beide Mengen eindeutig klassifizieren kann. Die gefärbten Räume sind die Ergebnisse der Ausgabeschicht für jeden Punkt des Gitters. Ein einzelnes Neuron mit n-Eingängen kann lediglich linear separierbare Mengen klassifizieren, indem es den Raum in 2 zwei Teile trennt.



Ohne verdeckte Schicht

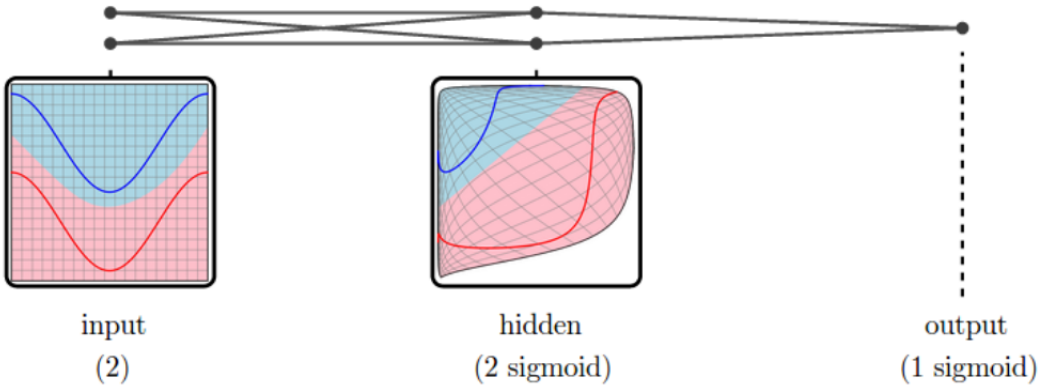


Abbildung 2.9.: Mit verdeckter Schicht Olah (2014)

Das Hinzufügen einer verdeckten Schicht erlaubt eine zusätzliche Komplexitätsebene. In jeder verdeckten Schicht werden die Daten der vorherigen Schicht zuerst linear und anschließend nichtlinear transformiert. Jede verdeckte Schicht hat so eine eigene Repräsentation der Daten. Ziel ist es die Daten so zu transformieren, dass diese linear trennbar werden und dadurch von der Ausgabeschicht klassifiziert werden können (Abbildung 2.1).

Grundsätzlich werden bei jeder Schicht folgende Operationen angewendet:

1. Lineare transformation mit Gewichten w.
2. Eine Translation um Bias θ.

3. Punktweise Anwendung der Aktivierungsfunktion.

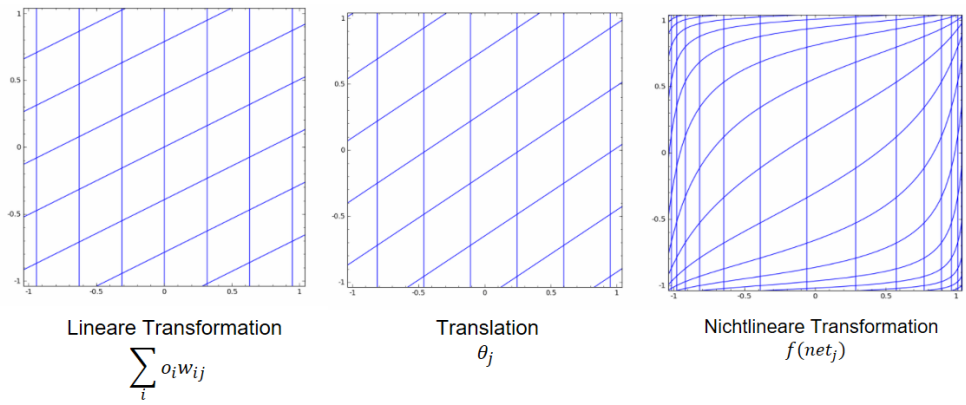


Abbildung 2.10.: Nicht linear separierbare Daten (Angepasst von: Olah (2014))

Der lineare Teil des Netzwerks wird durch das Anpassen der Gewichte und Bias bestimmt, wobei die Nichtlinearität durch die Aktivierungsfunktion eingeführt wird. Letztendlich kann laut dem **Universal approximation theorem** Cybenko (1989), jedes neuronale Netz mit einer verdeckten Schicht, einer endlichen Anzahl an Neurons und ausreichenden trainierbaren Parametern jede Funktion approximieren.

Kostenfunktion bei neuronalen Netzen

Mit dem Wissen wie neuronale Netze aufgebaut sind, wird nun erläutert wie diese trainiert werden. Die Parameter eines neuronalen Netzes können wie das vorherige Beispiel der linearen Regression anhand einer Kostenfunktion trainiert werden.

$$MSE(w,b) = \frac{1}{2N}\sum_x (y_x - o_x)^2 \tag{2.8}$$

	Lineare Regression	**Neuronales Netz**
Kostenfunktion	$\frac{1}{N}\sum_i (y_i - (mx_i + b))^2$	$\frac{1}{2N}\sum_x (y_x - o_x)^2$

Tabelle 2.1.: Supervised Learning im Vergleich

Die Kostenfunktion ist auch bei neuronalen Netzen der Mean Squared Error. Beim neuronalen Netz wird die quadratische Abweichung zwischen dem Musterbeispiel y_x und dem Output o_x für das Trainingsbeispiel x des neuronalen Netzes ermittelt. Die Variable o_x ist der Outputvektor des neuronalen Netzes in Bezug auf die trainierbaren Parameter w, b und des

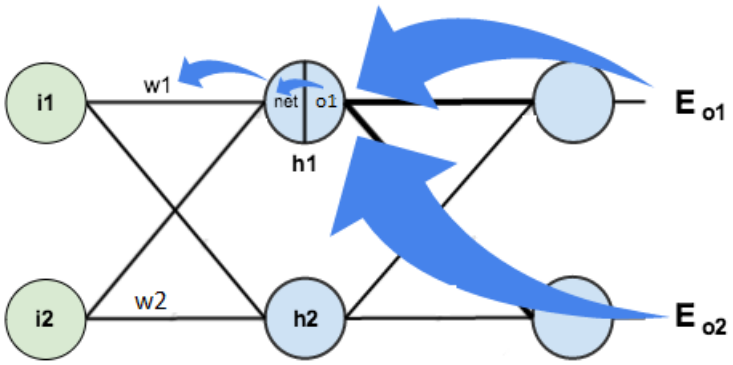
Trainingbeispiels x. Ein Musterbeispiel kann in diesem Fall der korrekte Klassifizierungswert eines Punktes, der zu klassifizieren gilt sein. Die Kostenfunktion des neuronalen Netzes besitzt zudem einen Faktor von $\frac{1}{2}$. Dies macht lediglich die Mathematik einfacher wenn bei der Gradientenbestimmung die Kostenfunktion abgeleitet wird. Hier kürzt sich dann die 2 des Exponenten der Kostenfunktion mit dem Faktor $\frac{1}{2}$.

Gradientenbestimmung bei neuronalen Netzen: Backpropagation

	Lineare Regression	**Neuronales Netz**
Bestimmung des Gradienten	partielle Ableitung	Backpropagation: Kettenregel mit Komponenten des Gradienten

Tabelle 2.2.: Supervised Learning im Vergleich

Neuronale Netze verwenden den Backpropagation Algorithmus, um den Gradienten für die Parameter des Netzwerks zu bestimmen. Dabei verwendet Backpropagation die Kettenregel, um den Gradienten der Parameter im Netzwerk zu bestimmen. Das Bestimmen des Gradienten für die Gewichte eines neuronalen Netzes ist komplizierter als bei der linearen Regression. Zum Beispiel hängt der Gradient der Gewichte in Schicht 1 von den Gradienten der Gewichte in Schicht 2 und Schicht 3 ab. Das heißt, da die Gewichte in den ersteren Schichten von Gewichten der letzteren abhängen, müssen zuerst die Gradienten der Gewichte im hinteren Teil des Netzes bestimmt werden und dann sukzessive für den Rest des Netzwerks. Dies gibt den Ursprung für den Namen Backpropagation, da die Ergebnisse von den hinteren Schichten zu den Vorderen durchgereicht werden.

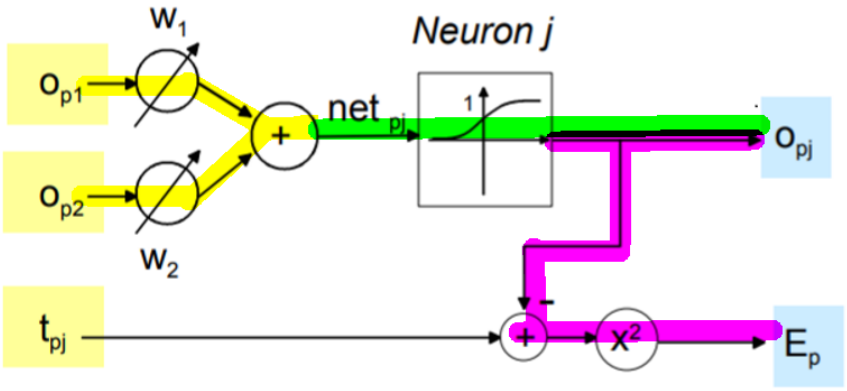


$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} \cdot \frac{\partial out_{h1}}{\partial net_{h1}} \cdot \frac{\partial net_{h1}}{\partial w_1}$$

$$\Downarrow$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} \cdot \frac{\partial E_{o2}}{\partial out_{h1}}$$

Abbildung 2.11.: Backpropagation Visualisierung (Eigene Erstellung)



$$E_p = E_p(o_{pj}) = (t_{pj} - o_{pj})^2 \qquad o_{pj} = o_{pj}(net_{pj}) = f_{act}(net_{pj}) \qquad net_{pj} = o_{p1}w_1 + o_{p2}w_2$$

$$\frac{\partial E_p}{\partial w_i} = \frac{\partial E_p}{\partial o_{pj}} \cdot \frac{\partial o_{pj}}{\partial net_{pj}} \cdot \frac{\partial net_{pj}}{\partial w_i} = -2(t_{pj} - o_{pj}) \cdot f'(net_{pj}) \cdot o_{pi}$$

Abbildung 2.12.: Ermittlung des Gradienten für w_i: Kettenregel (Angepasst von: Meisel)

Der Gradient für einen Gewichtsvektor $\vec{w}$ kann durch die Kettenregel bestimmt werden. Abbildung 2.12 visualisiert die Anwendung der Kettenregel für das Ermitteln des Gradienten für zwei Gewichte w_1 und w_2. Für ein Gewicht w_i wird die Komposition der Funktionen vom Gewicht w_i bis zur Kostenfunktion E_p aufgestellt. Anhand der Kettenregel kann die Ableitung dieser Komposition von Funktionen und somit der Gradient für ein Gewicht w_i ermittelt werden.

Parameteranpassung: Stochastic Gradient Descent

	Lineare Regression	**Neuronales Netz**
Anpassung des Gradienten	Gradient Descent	Stochastic Gradient Descent

Tabelle 2.3.: Supervised Learning im Vergleich

Wie das Beispiel der linearen Regression verwenden auch neuronale Netze Gradient Descent, um die trainierbaren Parameter anzupassen. Allerdings treten bei einer hohen Anzahl an Trainingsbeispielen Performanceprobleme auf, da für jedes Trainingsbeispiel x der Gradient der Kostenfunktion ∇MSE_x berechnet werden muss. Anschließend wird über alle ∇MSE_x gemittelt, um den Gesamtgradienten der Kostenfunktion ∇MSE (2.9) zu ermitteln. Bei einer hohen Anzahl an Trainingsbeispielen kann dies einige Zeit in Anspruch nehmen und verlangsamt den Lernprozess erheblich.

$$\nabla MSE = \frac{1}{N} \sum_{x} \nabla MSE_x \tag{2.9}$$

Um dieses Performanceproblem zu beheben verwenden neuronale Netze eine Variante des Gradient Descent, den Stochastic Gradient Descent. Die Idee besteht darin ∇MSE mit einer geringeren Anzahl an beliebig ausgewählten Samples aus den verfügbaren Trainingsbeispielen zu berechnen. Mehrere Samples j werden in einem **Mini-Batch** aufgefasst. Trotz der reduzierten Anzahl an Samples m des Mini-Batches ist zu erwarten, dass nach mehreren episodeen der echte Gradient ∇MSE ermittelt wird.

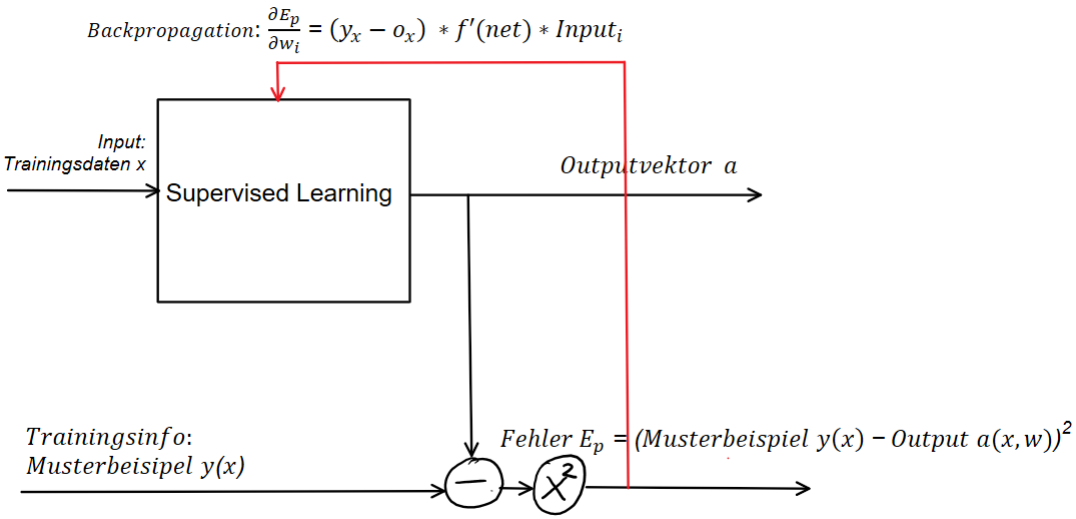
$$\nabla MSE \approx \frac{1}{m} \sum_{j=1}^{m} \nabla MSE_j \tag{2.10}$$

Stochastic Gradient Descent wählt von der gesamten Trainingsmenge N nach und nach beliebige Samples j aus und trainiert das Netzwerk anhand der Mini-Batches. Dies wird durchgeführt bis die gesamte Menge N an Trainingsbeispielen erschöpft wurde. Stochastic Gradient Descent

beschleunigt die Ermittlung des Gradienten und somit das Trainieren des neuronalen Netzes erheblich.

Nichtsdestotrotz, umso kleiner die Mini-Batches desto ungenauer wird die Ermittlung des Gradienten. In der Regel ist dennoch die schnelle Bestimmung wichtiger als die genauere Bestimmung des Gradienten, da die allgemeine Richtung des Gradienten, um die Kostenfunktion zu minimieren für ein erfolgreiches Lernen von größerer Bedeutung ist.

Supervised und Reinforcement Learning



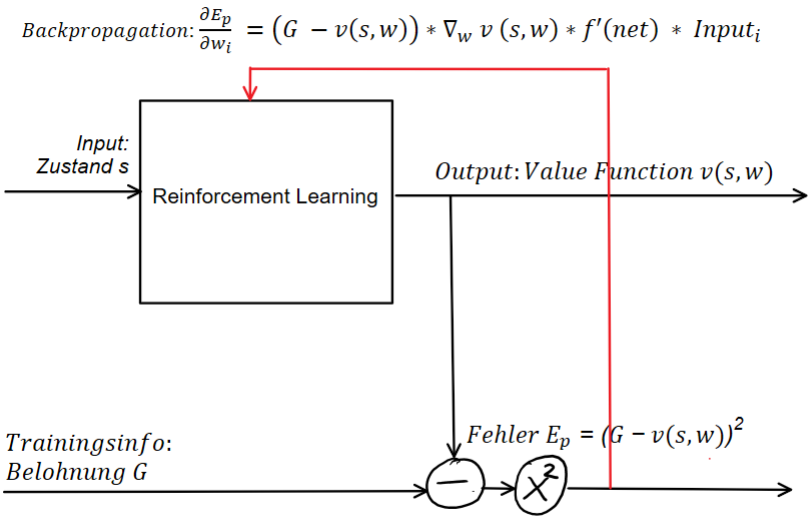


Abbildung 2.13.: Vergleich Supervised und Reinforcement Learning (Eigene Erstellung)

Grundsätzlich wird Supervised Learning für Klassifikations- und Regressionsprobleme verwendet. Reinforcement Learning hingegen zur Interaktion mit dynamischen Umgebungen. Nichtsdestotrotz, verwenden Reinforcement Learning Architekturen neuronale Netze als Funktionsapproximatoren und verwenden dadurch das Verfahren der iterativen Parameterbestimmung. Lediglich das Ziel der Kostenfunktion ändert sich. Im Reinforcement Learning fungiert ein Feedback oder auch Belohnung als Label oder Musterbeispiel und das Ziel ist die Netzwerkparameter so anzupassen, dass diese Belohnungen stets maximiert werden.

2.2. Reinforcement Learning

2.2.1. Grundlegende Konzepte

Das Ziel des Reinforcement Learnings ist aus Interaktionen mit der Umgebung Strategien zu erlernen, die das Erreichen eines bestimmten Zieles ermöglichen. Dabei werden im Reinforcement Learning zwei hauptsächliche Komponenten definiert: Der **Agent** und die **Umgebung**. Die lernende und mit der Umgebung interagierende Komponente wird als Agent bezeichnet, wohingegen die Umgebung alles weitere außerhalb des Agenten definiert.

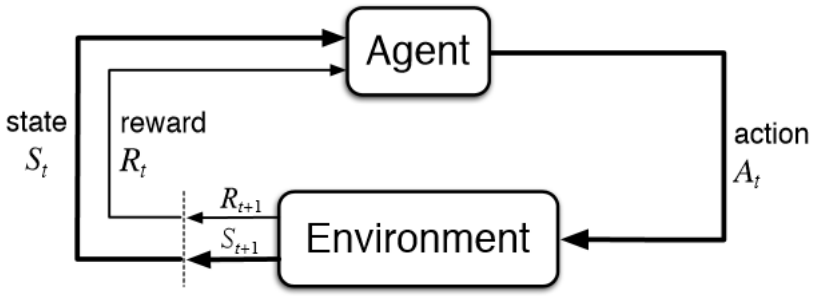


Abbildung 2.14.: Agent und Umgebung Sutton und Barto (2017)

State, Reward und Policy

Die Umgebung liefert Signale, die dem Agenten das Lernen ermöglichen. Eines dieser Signale ist der **Zustand (State)** und bezeichnet den aktuellen Zustand der Umgebung. Ein Zustandsvektor besteht in der Regel aus Informationen, die die aktuelle Lage des Agenten in der Umgebung definieren. Die **Belohnung (Reward)** versucht der Agent zu maximieren. Die Belohnung resultiert aus einer Aktion, die der Agent im sich befindenden Zustand ausgewählt hat. Diese Belohnung kann beispielsweise in einem Modell fest vorgeschrieben sein. In anderen Fällen muss die Belohnung für eine Umgebung modelliert werden, um ein bestimmtes Verhalten zu fördern. Grundsätzlich interagiert der Agent mit der Umgebung in diskreten Zeitschritten $t(z.Bt = 0, 1, 2, 3)$, bekommt nach jeder Interaktion mit der Umgebung ein Belohnungssignal und geht in einen neuen Zustand über.

Anhand von Abbildung 2.14 wird die Interaktion zwischen Agent, Umgebung und den daraus resultierenden Signalen formeller beschrieben:

- S_t : Der Zustand der Umgebung wird durch $S_t \in S$ beschrieben, wo S die Menge aller möglichen Zustände in der Umgebung ist.

- A_t : Von einem Zustand S_t selektiert der Agent eine bestimmte Aktion beschrieben durch $A_t \in A(S_t)$, wo $A(S_t)$ die Menge aller möglichen Aktionen von dem Zustand S_t aus beschreibt.
- R_t : Nach Selektieren einer Aktion A_t folgt aus der Umgebung eine Belohnung $R_{t+1} \in R$. Bei dieser Belohnung handelt es sich in der Regel, um einen reelen skalaren Wert und dient dem Agenten als Feedback.

Anhand der Signale ist es nun möglich das Verhalten eines Agenten formal zu definieren. Für jeden Zeitschritt t besitzt der Agent eine Wahrscheinlichkeitmatrix, die darstellt mit welcher Wahrscheinlichkeit jede mögliche Aktion innerhalb eines Zustands selektiert wird. Dieses Mapping wird als **Policy** bezeichnet. Die Policy wird als π_t definiert, wobei $\pi_t(a|s)$ die Wahrscheinlichkeit darstellt, dass Aktion $A_t = a$ im Zustand $S_t = s$ ausgewählt wird. Die verwendete Methoden im Reinforcement Learning haben als auschließliches Ziel die Policy, also die Aktionen des Agenten so auszuwählen, dass die aus der Umgebung stammenden Belohnungen maximiert werden.

Markov Decision Process

Bisher sind grundlegende Konzepte, die die Interaktion zwischen Agent und Umgebung def nieren präsentiert worden. Wie ein Agent eine Umgebung wahrnimmt und diese navigiert, kann formal durch Markov Decision Processes beschrieben werden.

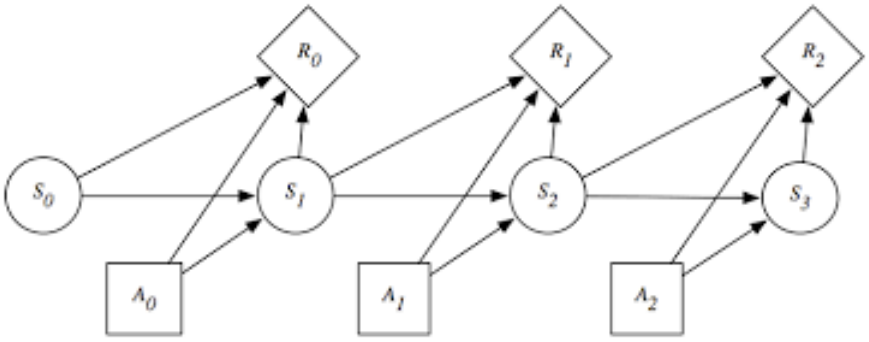


Abbildung 2.15.: Markov Decision Process Poole und Mackwort (2010)

Eine wichtige Eigenschaft um eine Umgebung als Markov Decision Process zu modellieren, ist die Markov Property. Die Markov Property behandelt die Notwendigkeit alte Zustandstransitionen kennen zu müssen. In der Regel wird davon ausgegangen, dass eine Transition zum

Zeitpunkt $t+1$, durch eine Aktion zum Zeitpunkt t verursacht wurde von allen vorherigen Ereignissen abhängt:

$$Pr\{R_{t+1} = r, S_{t+1} = s' | S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\} \tag{2.11}$$

Die Markov Property beschreibt, dass die Transition zum Zeitpunkt $t+1$ lediglich von der aktuellen Aktion und von dem aktuellen Zustand zum Zeitpunkt t abhängen:

$$Pr\{R_{t+1} = r, S_{t+1} = s' | S_t, A_t\} \tag{2.12}$$

Dies ermöglicht es die aktuelle Umgebung des Agenten lediglich durch den aktuellen Zustand zu beschreiben ohne vorherige Ereignisse berücksichtigen zu müssen und reduziert die benötigte Komplexität um eine Umgebung zu beschreiben erheblich.

Eine Umgebung, die die Markov Property besitzt wird als Markov Decision Process kurz MDP bezeichnet. Ein MDP besteht aus den Zustands- und Aktionsmengen, die eine Umgebung beschreiben. In einem MDP wird angenommen das diese beiden Mengen endlich sind.

Die Wahrscheinlichkit, dass von einem Zustand s durch Aktion a ein Folgezustand s' erreicht wird kann folgendermaßen beschrieben werden:

$$p(s'|s,a) = PrS_{t+1} = s' | S_t = s, A_t = a \tag{2.13}$$

Diese Wahrscheinlichkeitswerte werden Übergangswarscheinlihckeiten (Transition Probabilities) genannt und beschreiben die Wahrscheinlichkeit, dass im Zustand s nach Ausführen der Aktion a der Folgezustand s' eintritt.

Aus dem Übergang von Zustand s zu s' durch Aktion a ergibt sich eine unmittelbare Belohnung:

$$r(s,a,s') = \mathbb{E}[R_{t+1} | S_t = s, A_t = a, S_{t+1} = s'] \tag{2.14}$$

$p(s'|s,a)$ und $r(s,a,s')$ beschreiben die Wechselwirkungen innerhalb eines MDP's und somit die Interaktion eines Agenten mit dessen Umgebung.

Ziele und Belohnungen

Die Festlegung eines bestimmten Zieles durch die Modellierung von Belohnungen ist der Kernpunkt im Reinforcement Learning. Aufgrund dessen wird untersucht was unter einer Be-

lohnung tatsächlich zu verstehen ist und wie diese Formal definiert werden kann. Wie erwähnt ist eine Belohnung ein reeler skalarer Wert $R \in \mathbb{R}$, der je nach Situation unterschiedliche Werte annehmen kann.

Abbildung 2.16 zeigt ein einfaches Labyrinth, dass der Agent erfolgreich durchqueren soll. Das Erreichen des Zieles wird mit einem positiven Feedback $R_t = +1$ belohnt. Um den Agenten einen Ansporn zu geben wird nach jedem Zeitschritt ein negatives Feedback $R_t = -1$ erzeugt, damit sich der Agent bemüht das Labyrinth zu verlassen. Die situationsabhängigen Belohnungen formen dadurch das Verhalten des Agenten. Grundsätzlich gilt, dass für einen unerwünschten Zustand eine negative Belohnung und für einen erwünschten Zustand eine positive Belohnung festgelegt wird.

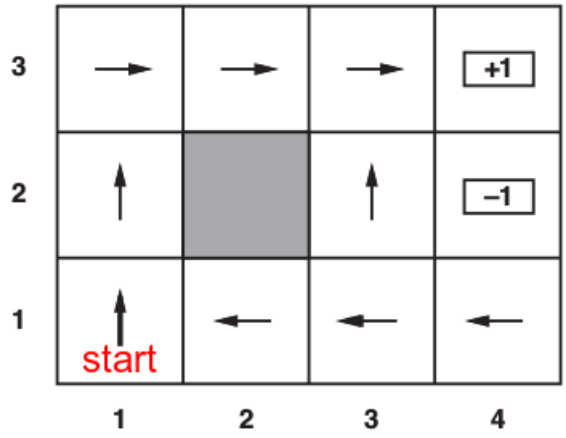


Abbildung 2.16.: Labyrinth: -1 für jeden Zeitschritt, +1 bei Verlassen des Labyrinths cnblogs (2015)

Das grundlegende Konzept einer Belohnung erscheint im ersten Augenblick recht einfach. Dennoch können Belohnungen beliebig komplex werden, weshalb es nötig ist ein formales Grundgerüst zu definieren.

Das Ziel im Reinforcement Learning ist eine Policy zu finden, die die erhaltenen Belohnungen maximiert. Genau genommen können die erhaltenen Belohnungen als Erwartungswert definiert werden, also als Summe aller erhaltenen Belohnungen in den Zeitschritten von $t = 0$ bis $t = T$, wo T den letzten Zeitschritt repräsentiert.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{2.15}$$

Diese Definition ist für Interaktionen gültig in denen eine konkrete Aufgabe tatsächlich terminiert, also ein terminierender Zustand definiert ist. Zum Beispiel das Auftreffen auf eine Wand im Labyrinth könnte als terminierender Zustand definiert werden. Handelt es sich jedoch um Umgebungen, die keinen Terminierungszustand aufweisen, also unendlich weitergehen, ist die Definition des Erwartungswerts in 2.15 nicht mehr korrekt. Für diesen Fall könnte G_t potenziell $G_t = \infty$ werden, da die Zeitschrittfolge unendlich weiter geht. Die Einführung eines Discount-Faktors löst dieses Problem für kontinuierliche Aufgaben. Es wird ein Parameter γ, $0 \leq \gamma \leq 1$ auch Discountrate genannt eingeführt. Die aktualisierte Formel für den diskontierten Erwartungswert sieht dadurch folgendermaßen aus:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.16}$$

Der Discountfaktor bestimmt welchen Anteil der Zukunft der Erwartungswert enthalten soll. Zum Beispiel bedeutet $\gamma = 0$, dass nur die unmittelbare Belohnung R_{t+1} den Erwartungswert ausmacht und daher der Agent "kurzsichtig"wird. Bei $\gamma \approx 1$ hingegen nimmt der Agent immer mehr Werte aus der Zukunft auf und wird dadurch "weitsichtiger". Grundsätzlich garantiert $\gamma < 1$, dass die Summe des Erwartungswertes endlich ist und somit für kontinuierliche nicht terminierende Aufgaben definiert bleibt. In zusammengefasster Schreibweise kann der Erwartungswert für beide Fälle folgendermaßen definiert werden:

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \tag{2.17}$$

Hier ist k der k-te Zeitschritt in der Zukunft. Dabei dürfen $T = \infty$ und $\gamma = 1$ nicht gleichzeitig auftreten, da sonst der Erwartungswert wie oben beschrieben aufgrund der Unendlichkeit nicht definiert werden kann. In der Regel wird beispielsweise für das Zählen von Karten (Monte Carlo Methode) $\gamma = 1$ gesetzt, da hier von einem Zustand aus jeweils die vollständige Belohnung bis zum terminierenden Zustand ermittelt wird. Das Ende eines Spielzuges definiert gleichzeitig auch den terminierenden Zustand. In dieser Arbeit wird eine dynamische Umgebung behandelt, die keinen Terminierungszustand aufweist. Es erscheint als logisch, dass der Erwartungswert diskontiert werden muss, da ansonsten die erwartete Belohung nur als unendlich definiert werden kann. Zudem besitzen dynamische Umgebungen eine temporäre Komponente. Es kann in einer dynamischen Umgebung ausschlaggebend sein wie stark zukünftige Belohnungen berücksichtigt werden. Daher ist für solche Umgebungen $T = \infty$ und $\gamma = 0.99$. Grundsätzlich ist es auch möglich, dass für kontinuierliche Umgebungen ein terminierender Zustand definiert

wird, wenn dies so gewünscht ist.

Value Function

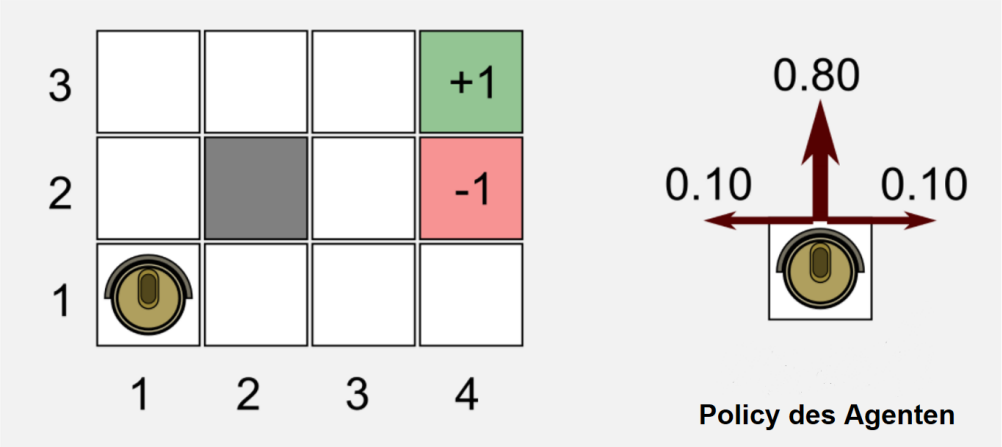


Abbildung 2.17.: Policy eines Agenten in einer Umgebung cnblogs (2015)

Wie bereits erwähnt definiert eine Policy, die Übergangswarscheinlichkeiten aller Aktionen $a \in A(s)$, die in einem Zustand s stattfinden können, wobei $\pi(a|s)$ die Wahrscheinlichkeit ist, dass eine Aktion a im Zustand s ausgewählt wird. Abbildung 2.17 veranschaulicht anhand des vorherigen Beispiels des Labyrinths das Verhalten einer Policy. Für jeden Zeitschrit t sind nun 3 Aktionen fest definiert, die mit jeweils unterschiedlichen Wahrscheinlichkeiten ausgeführt werden können. So wird sich der Agent in 80% der Fälle nach vorne Bewegen, wobei die restlichen 20% eine seitliche Bewegung bedeuten würden.

Ziel ist es nachwievor diese Policy so anzupassen, dass der Agent die erwarteten Belohnungen maximiert. Im Fall des Labyrinths soll dieses so durchquert werden, dass die maximal mögliche Belohnung erwirtschaftet wird. Hierzu muss die definierte Anfangspolicy optimiert werden. Bevor eine verbesserte Policy bestimmt werden kann, müssen Kriterien und Metriken festgelegt werden nach denen das Verhalten einer Policy bestimmt werden kann. Dies geschieht mit sogenannten **Value Functions**. Die Idee besteht darin jeden Zustand eines MDP's durch einen Wert zu modellieren, der eine Auskunft darüber gibt wie gut eine Policy für einen bestimmten Zustand Belohnungen maximiert. Man unterscheidet zwischen zwei Arten von Value Functions: Der **State-Value Function** und der **Action-Value Function**.

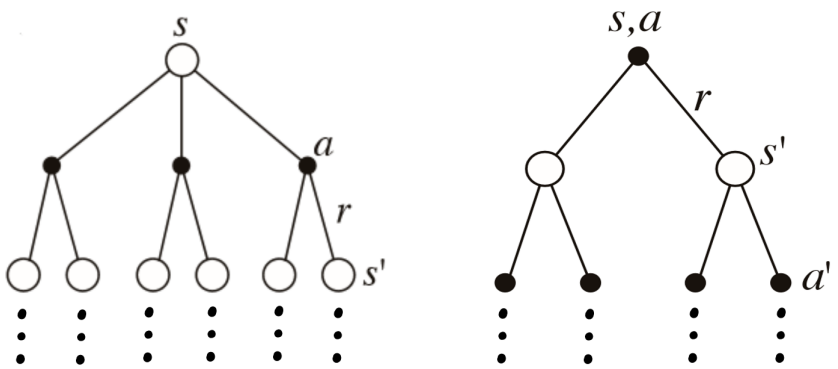


Abbildung 2.18.: Value Functions für v_π und q_π Sutton und Barto (2017)

Die State-Value Function v_π definiert für einen Zustand $s \in S$ die zu erwartende Belohnung, wenn von dem Zustand s aus die Policy π ausgeführt wird. Das heißt, wenn alle die durch Policy π definierten Übergangswahrscheinlichkeiten berücksichtigt werden. Allgemein kann dieser Erwartungswert folgendermaßen formuliert werden:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum \gamma^k R_{t+k+1}|S_t = s\right] \tag{2.18}$$

Formel 2.18 beschreibt die State-Value Function. Ausgehend von einem Zustand s, ist der State-Value $v_\pi(s)$ die zu erwartende diskontierte Belohnung für Zustand s wenn von diesem Zustand aus bis in die Zukunft der Policy π gefolgt wird.

Analog kann auch eine Action-Value Function q_π für eine Aktion a im Zustand s unter der Policy π definiert werden. Die Action-Value Function beschreibt dann die erwartete Belohnung wenn von einem Zustand s aus eine konkrete Aktion a gewählt wird und dann die Übergangswahrscheinlichkeiten der Policy π berücksichtigt werden.

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum \gamma^k R_{t+k+1}|S_t = s, A_t = a\right] \tag{2.19}$$

Formel 2.19 beschreibt die Action-Value Function, die bis auf die Abhängigkeit von der Aktion a identisch mit der State-Value Function ist. Der Unterschied zur State-Value Function besteht darin, dass die Action-Value Function zusätzlich von einer festen Aktion a als Startpunkt ausgeht, also die erwartete Belohnung für eine ausgewählte Aktion a von einem Zustand s aus liefert. Die State-Value Function liefert hingegen die erwartete Belohnung für den Zustand s unter der Berücksichtigung aller Übergangswahrscheinlichkeiten für verschiedene Aktionen

a, die durch die Policy π definiert werden.

Ein Weg v_π und q_π zu bestimmen sind die Monte-Carlo Methoden. Monte-Carlo schätzt v_π über mehrere Episoden. Es wird die erwartete Belohnung für einen Zustand s ermittelt indem die bis zum Ende der Episode erhaltenen Belohnungen gemittelt werden. Dieser Mittelwert konvergiert nach einiger Zeit zum echten $v_\pi(s)$ bzw. $q_\pi(s, a)$. Hierbei ist zu bemerken, dass Monte-Carlo Methoden voraussetzen, dass ein terminierender Zustand definiert ist.

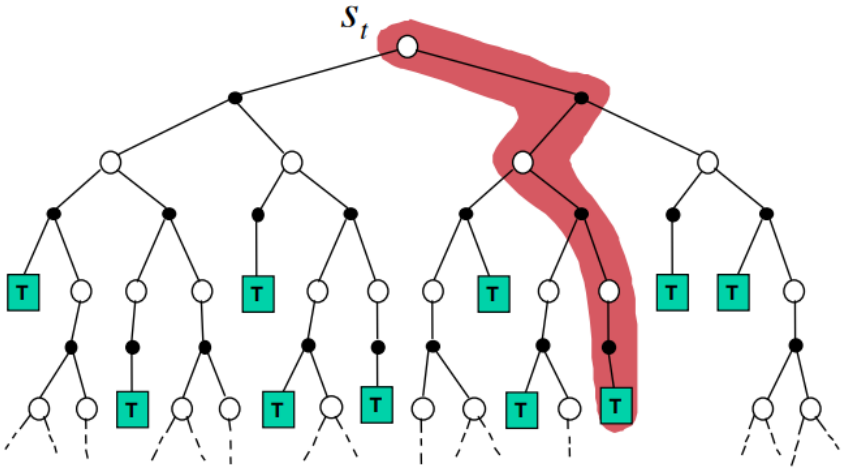


Abbildung 2.19.: Backupdiagramm für Monte-Carlo Silver (2015)

Monte-Carlo Methoden sind allerdings nur auf episodische Aufgaben ausgelegt, da um die erwartete Belohnung G_t ermitteln zu können ein terminierender Zustand definiert sein muss. Es muss also ein Weg gefunden werden, um eine Value Function in kleineren Schritten zu approximieren. Dies kann durch die sogenannte Bellman Equation formuliert werden:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi\left[\sum \gamma^k R_{t+k+1}|S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma \sum \gamma^k R_{t+k+2}|S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s'} p(s'|s, a)\left[r(s, a, s') + \gamma \mathbb{E}_\pi\left[\sum \gamma^k R_{t+k+2}|S_{t+1} = s'\right]\right] \\ &= \sum_a \pi(a|s) \sum_{s'} p(s'|s, a)\left[r(s, a, s') + \gamma v_\pi(s')\right] \end{aligned} \tag{2.20}$$

Die Bellman Equation entsteht durch das Entfalten des Erwartungswerts in 2.18 und stellt einen Zusammenhang zwischen dem State-Value eines Zustands s und dessen Folgezustand s' her.

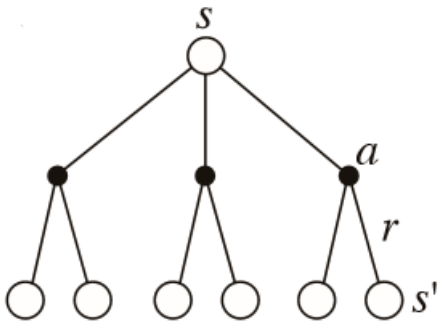


Abbildung 2.20.: Backupdiagramm für v_π Sutton und Barto (2017)

Abbildung 2.20 erläutert die Bellman Equation grafisch. Ausgehend vom Zustand s mittelt der Agent über alle Übergangswahrscheinlichkeiten der Policy π (im Diagramm 3 Aktionen). Von jeder dieser Aktionen kann die Umgebung mehrere Folgezustände s' und Belohnungssignale r liefern. Die Bellman Equation definiert für einen Zeitschritt die **unmittelbare** Belohnung unter Berücksichtigung aller Wahrscheinlichkeiten, der vefügbaren Aktionen $a \in A(s)$ und der danach potenziell eintretenden Folgezustände s'. Die Bellman Equation ist eine Rekursion, die für beliebig viele Zeitschritte gelöst werden kann. Konkret definiert die Bellman Equation, dass der State-Value eines Zustands s den diskontierten State-Values der eintretenden Folgezustände s' plus den unmittelbaren Belohnungen multipliziert mit den jeweligen Übergangswahrscheinlichkeiten $\sum_a \pi(a|s) \sum_{s'} p(s'|s,a)$ entspricht.

Optimal Value Function

Eine Value Function bietet die Möglichkeit das Verhalten einer Policy zu messen. Im Folgenden wird erläutert wie die Definition der Value Function für den optimalen Fall angepasst werden kann, so dass diese optimale Werte liefert und daher auch eine optimale Policy beschreibt.

Eine Policy π' ist besser oder gleich gut als eine Policy π wenn der State-Value Wert jedes Zustands für Policy π' größer oder gleich als der State-Value Wert jedes Zustands für π ist. Mit anderen Worten, ist $\pi' > \pi$, wenn $v_{\pi'}(s) > v_\pi(s)$ für alle Zustände $s \in S$. Das Bestimmen einer optimalen Value Function bedeutet zwangsläufig, dass diese auch eine optimale

Policy beschreibt. Eine optimale Policy π_* besitzt also eine State-Value Function, die optimale State-Value Function, die als v_* bezeichnet wird:

$$v_*(s) = \max_\pi v_\pi(s), s \in S \tag{2.21}$$

Analog kann die optimale Action-Value Function definiert werden:

$$q_*(s,a) = \max_\pi q_\pi(s,a), s \in S, a \in A(s) \tag{2.22}$$

Die optimale State-Value Function beschreibt die erwartete Belohnung wenn von einem Zustand s aus die **optimale** Policy gefolgt wird. Die optimale Policy ist in diesem Fall, eine Policy π, die den State-Value $v_\pi(s)$ bzw. Action-Value $q_\pi(s,a)$ maximiert.

Wie für v_π kann auch für v_* die Bellman Equation formuliert werden. Diese wird im optimalen Fall **Bellman Optimality Equation** genannt. Das Betrachten eines einzelnen Zeitschrittes in der Bellman Equation erlaubt es die Referenz auf π entfallen zu lassen. Stattdessen wird im Aktionsraum $A(s)$ die Aktion a ausgewählt, die den maximalen State-Value liefert.

$$v_*(s) = \max_{a \in A(s)} \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma v_*(s')] \tag{2.23}$$

$$q_*(s,a) = \sum_{s'} p(s'|s,a) \left[r(s,a,s') + \gamma \max_{a'} q_*(s',a') \right] \tag{2.24}$$

Für $q_*(s,a)$ ergibt sich ebenfalls, dass die Referenz zur Policy π entfällt. Hier wird nach Ausführen der Aktion a im Zustand s über den gesamten Aktionsraum des eintretenden Folgezustands s' der Action-Value Wert $q_*(s',a')$ maximiert. Abbildung 2.21 veranschaulicht beide Bellman Optimality Equations. Dabei stellt Diagramm a) v_*(s) und Diagramm b) $q_*(s,a)$ dar.

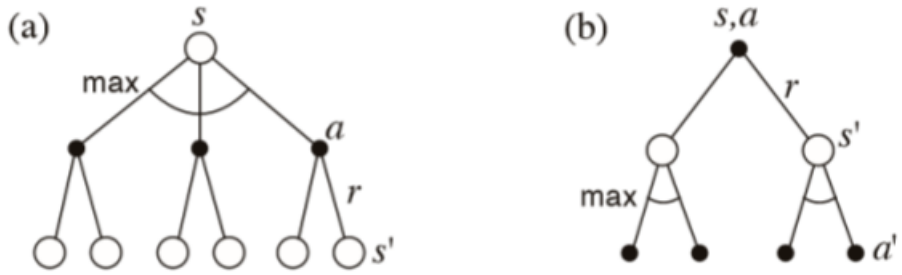


Abbildung 2.21.: Backupdiagramm für Bellman Optimality Equations Sutton und Barto (2017)

Die Bellman Optimality Equations v_* und q_* können für endliche Markov Decision Processes und unter der Annahme, dass alle Übergangswahrscheinlichkeiten bekannt sind, durch das Lösen von N nichtlinearen Gleichungen wo N die Anzahl der Zustände ist, bestimmt werden. Wenn v_* oder q_* bekannt sind, dann kann eine optimale Policy relativ einfach bestimmt werden.

Für v_* muss für jede mögliche Aktion a in einem Zustand s, die Aktion ermittelt werden, die den größten State-Value liefert(vergleiche Gleichung 2.18). Mit anderen Worten, jede Policy, die sich greedy gegenüber v_* verhält ist dann eine optimale Policy. Für q_* ist dies noch einfacher. Es genügt für jeden Zustand s die Aktion zu finden, die $q(s, a)$ maximiert. Der Action-Value beinhaltet bereits, die Information über die Aktion, die die erwartete Belohnung maximiert. Es ist also kein zusätzlicher Schritt(look-ahead) wie bei v_* notwendig.

Diese Erkentnis ist von großer Bedeutung, denn dies erlaubt es eine optimale Policy zu bestimmen ohne Übergangswahrscheinlichkeiten der Umgebung berücksichtigen zu müssen. Dies bildet die Basis für das sogenannte modellfreie Lernen, das unabhängig von der Dynamik der Umgebung in der Lage ist eine Value Function zu bestimmen.

2.2.2. Dynamische Programmierung: Policy Evaluation und Policy Improvement

Im letzen Kapitel wurde das Grundgerüst des Reinforcement Learnings eingeführt. In diesem Kapitel wird ein Framework zum Bestimmen von Value Functions und zum Finden optimaler Policies definiert.

Nach der Definition einer Policy und einer optimalen Policy müssen nun Methoden geschaffen

werden um eine optimale Policy bestimmen zu können. Genau dies ist die Aufgabe der dynamischen Programmierung. Die präsentierten Algorithmen in der dynamischen Programmierung sind in der Realität nur beschränkt einsetzbar, da diese ein vollständiges Modell der Umgebung vorsehen und hohe Kosten aufweisen. Nichtsdestotrotz, ist es das Grundgerüst für diverse Algorithmen, die Value Functions approximieren und daraus neue Policies bestimmen.

Die Kernaufgabe der dynamischen Programmierung ist die Definition von Methoden zur Bestimmung einer Value Function und der daraus resultierenden Suche nach einer besseren Policy. Um dies zu erreichen transformiert die dynamische Programmierung die Bellman Optimality Equations zu Aktualisierungsregeln, die es erlauben v_* und q_* zu approximieren. Anschließend wird die neue Value Function ausgewertet, um eine neue bessere Policy zu bestimmen. Der Vorgang der Ermittlung der Value Function wird als **Policy Evaluation** bezeichnet. Das Bestimmen einer neuen besseren Policy ist Aufgabe des **Policy Improvements**.

Policy Evaluation

Der erste Schritt besteht darin eine Value Function zu ermitteln. Dieser Vorgang wird in der dynamischen Programmierung als Policy Evaluation und als Problem der **Vorhersage (Prediction Problem)** bezeichnet. Zur Erinnerung: Die Bellman Equation für die State-Value Function für alle $s \in S$:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) \left[r(s,a,s') + \gamma v_\pi(s') \right] \tag{2.25}$$

Wenn die Übergangswahrscheinlichkeiten ($\pi(a|s)$ und $p(s'|s,a)$) der Umgebung bekannt sind dann ist wie bei der Bellman Optimality Equation für v_*, v_π ein lineares Gleichungssystem mit $|S|$ Anzahl an Gleichungen mit $|S|$ unbekannten. Dabei ist S die Menge aller Zustände der Umgebung. Obwohl dieses Gleichungssystem grundsätzlich gelöst werden kann ist der Rechenaufwand für eine große Anzahl an Zuständen enorm. Wie bei der iterativen Parameterbestimmung in Kapitel 2.1 helfen iterative Lösungsansätze wenn eine Lösung nicht direkt bestimmt werden kann. Policy Evaluation definiert eine iterative Methode um eine Value Function zu approximieren: Die **iterative Policy Evaluation**.

Für eine Anfangsapproximation v_0, die mit einem beliebigen Wert initialisiert wird, kann

jede folgende Approximation durch das Verwenden der Bellman Equation in Gleichung 2.20 als Aktualisierungsregel bestimmt werden:

$$\begin{aligned} v_{k+1}(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) \left[r(s,a,s') + \gamma v_k(s') \right] \end{aligned} \tag{2.26}$$

Dabei kennzeichnet $\boldsymbol{kt}$ den Iterationsschritt. Für alle Zustände $s \in S$ wendet die iterative Policy Evaluation dann folgenden Algorithmus an, um v_π zu bestimmen: Der alte State-Value des Zustands s wird durch einen neuen Wert $v_{k+1}(s)$ ersetzt, indem der noch alte Wert $v_k(s')$ des Folgezustands für das Update verwendet wird (Abbildung 2.22). Hinzu kommen die unmittelbare Belohnung $r(s,a,s')$ und die Übergangswahrscheinlichkeiten der Umgebung. Für $k \to \infty$ konvergiert v_k zu v_π. Analog kann auch die iterative Policy Evaluation für die Action-Value Function definiert werden:

$$q_{k+1}(s,a) = \sum_{s'} p(s'|s,a) \left[r(s,a,s') + \gamma q_k(s',a') \right] \tag{2.27}$$

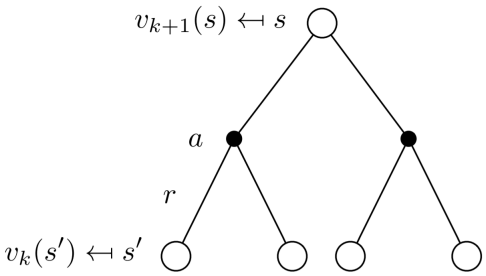


Abbildung 2.22.: Backup für iterative Policy Evaluation Sutton und Barto (2017)

Der Vorgang der iterativen Bestimmung des neuen State-Values v_{k+1}(s) durch Verwendung von $v_k(s')$ des Folgezustands wird als **Bootstrapping** bezeichnet. Dabei wird ein sogenannter **Full-Backup** realisiert, der alle möglichen Folgezustände s' berücksichtigt: $\sum_{s'} p(s'|s,a)$. Andere Value Function bestimmende Algorithmen wie **Monte-Carlo Evaluation** oder **Temporal Difference Prediction** samplen stattdessen den Folgezustand s' aus $\sum_{s'} p(s'|s,a)$. Dies erspart enormen Rechenaufwand, weshalb Algorithmen der dynamischen Programmie-

rung in realen Anwendungsfällen selten Verwendung finden. Abbildung 2.23 beschreibt die Arbeitsweise des Full-Backups grafisch.

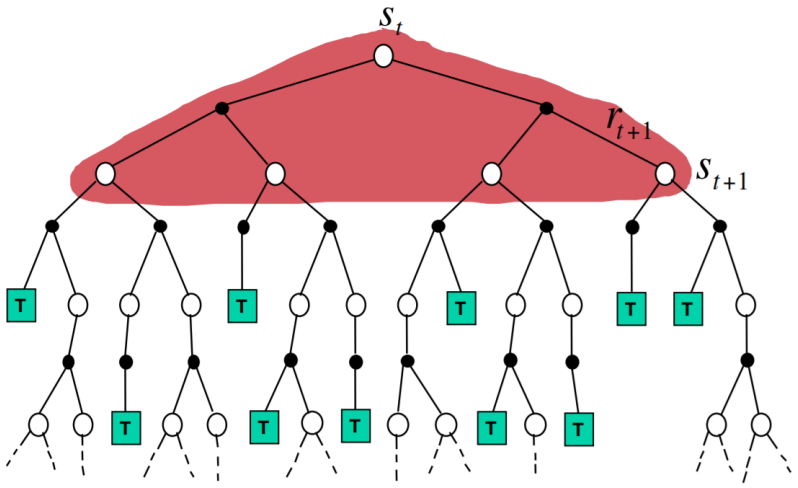


Abbildung 2.23.: Backupdiagramm für Bellman Optimality Equations Silver (2015)

Policy Improvement

Policy Evaluation erlaubt es die Value Function einer Policy zu ermitteln. Dies ist von großer Bedeutung, denn wenn die Value Function einer Policy bestimmt worden ist, kann analysiert werden wie anhand dieser Value Function v_π, eine bessere Policy π' ermittelt werden kann. Policy Improvement versucht für dieses Problem eine Lösung zu finden.

Als Beispiel wird angenommen, dass die Value Function v_π für eine Policy π bereits bestimmt worden ist. Für einen bestimmten Zustand s soll nun bestimmt werden, ob es sinvoll ist die Policy π durch das deterministische Wählen einer Aktion $a \neq \pi(s)$ zu verändern. Ein Weg dies zu beantworten ist deterministisch eine Aktion $a \in A(s)$ zu wählen und danach der Policy π zu folgen. Dies entspricht dem Verhalten der Action-Value Function:

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= \sum_{s'} p(s'|s,a) \left[r(s,a,s') + \gamma v_\pi(s') \right] \end{aligned} \tag{2.28}$$

Es ist festzustellen, ob der Action-Value $q_\pi(s,a)$ größer als der State-Value $v_\pi(s)$ ist. Wenn der Action-Value $q_\pi(s,a)$ tatsächlich größer als der alte State-Value $v_\pi(s)$ ist, dann kann daraus

geschlossen werden, dass das Ausführen der Aktion a im Zustand s allgemein eine bessere Policy formuliert. Diese Erkentnis wird formal durch das **Policy Improvement Theorem** beschrieben.

Wenn π und π' zwei beliebige deterministisch agierende Policies nach dem obigen Beispiel sind und es gilt für alle $s \in S$:

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \tag{2.29}$$

Dann muss Policy π' genau so gut oder besser als Policy π sein, unter der Annahme dass gilt:

$$v_{\pi'}(s) \geq v_\pi(s) \tag{2.30}$$

Gleichung 2.29 formalisiert den Zusammenhang, der vorher bereits grob formuliert wurde. Wenn die Action Value Function $q_\pi(s, \pi'(s))$ für jeden Zustand s anhand der veränderten deterministischen Policy π' größere Werte liefert als die ursprügliche State-Value Function, dann muss die modifizierte Policy π' auch eine bessere Policy als Policy π sein.

Durch das Extrapolieren dieser Erkenntnis auf alle Zustände und alle möglichen Aktionen in den jeweiligen Zuständen kann angenommen werden, dass eine bessere Policy ermittelt werden kann indem für jeden Zustand $s \in S$ die Aktion a gewählt wird, die den höchsten Action-Value $q_\pi(s, a)$ liefert. Eine solche Policy wird als eine neue Policy π' definiert, die sich greedy gegenüber der Action-Value Function q_π verhält:

$$\begin{aligned}
\pi'(s) &= arg\max_a q_\pi(s, a) \\
&= arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}|S_t = s, A_t = a] \\
&= arg\max_a \sum_{s'} p(s'|s, a) \left[r(s, a, s') + \gamma v_\pi(s')\right]
\end{aligned} \tag{2.31}$$

Für jeden Zustand s der neuen Policy π' wird die Aktion ausgeführt, die $q_\pi(s, a)$ maximiert. In diesem Fall bezeichnet arg $\max_a$ die spezifische Aktion a, die die neue Policy π' ausführt, um die Action-Value Function zu maximieren. Eine Policy, die immer die beste Aktion auswählt wird als greedy Policy bezeichnet. **Policy Improvement ist also das Bestimmen einer Policy π', die sich greedy gegenüber der Value-Function der ursprünglichen Policy π verhält und somit besser als die ursprüngliche Policy π ist.**

Policy Iteration

Policy Evaluation und Policy Improvement gehen Hand in Hand. Mit Policy Evaluation wurde die State-Value Function v_π ermittelt und anhand von Policy Improvement konnte die ermittelte State-Value Function dazu verwendet werden, um eine bessere Policy π' zu ermitteln. Dieser Prozess kann beliebig in die Länge gezogen werden. So könnte von π' die State-Value Function $v_{\pi'}$ bestimmt werden, um anschließend eine noch bessere Policy π'' zu finden. Durch die abwechselnde Anwendung von Policy Evaluation und Policy Improvement kann eine Sequenz von sich monoton verbessernden Policies und Value Functions ermittelt werden.

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*$$

Abbildung 2.24.: Policy Iteration: E = Policy Evaluation und I = Policy Improvement Sutton und Barto (2017)

Jede neu gefundene Policy ist eine garantierte Verbesserung gegenüber der vorherigen Policy mit der Ausnahme dass diese bereits optimal ist. Ein Markov Decision Process besitzt nur eine endliche Anzahl an Policies weshalb der Vorgang der Policy Iteration in einer endlichen Anzahl von Iterationen zur optimalen Policy und optimalen Value Function konvergieren muss. Die Methoden der dynamischen Programmierung Policy Improvement und Policy Iteration sind also erste Ansätze, die es ermöglichen mit einer endlichen Anzahl an Iterationen eine Value Function zu bestimmen und eine Policy zu verbessern.

Generalized Policy Iteration (GPI)

Die Policy Iteration definiert strikt, dass nach einer Policy Evaluation unmittelbar danach ein Policy Improvement Schritt folgen muss. Dies ist allerdings nicht immer notwendig, da verschiedene Algorithmen die Granularität zwischen Policy Evaluation und Policy Improvement Schritten variieren. Ausschlaggebend ist lediglich, dass beide Schritte immer den gesammten Zustandsraum vollständig aktualisieren.

Zur Verallgemeinerung wird der Begriff **Generalized Policy Iteration (GPI)** eingeführt, um Algorithmen zu beschreiben, die in irgendeiner Form Policy Evaluation und Policy Improvement anwenden. Unabhängig von der Granularität oder der konkreten Implementierung. Mit anderen Worten, wenn eine Methode eine Value Function und eine Policy identifizieren, die Value Function der Policy bestimmt und anschließend eine neue Policy aus der Value Function bestimmt wird, fällt diese Methode in die Kategorie der Generalized Policy Iteration.

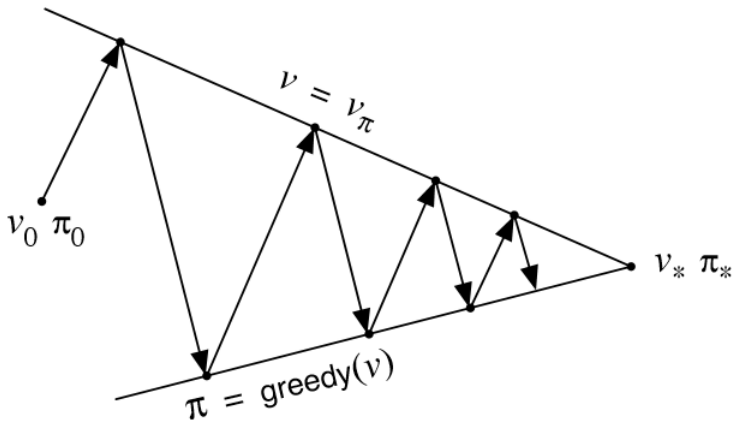


Abbildung 2.25.: Generalized Policy Iteration Silver (2015)

Abbildung 2.25 fasst die Idee der Generalized Policy Iteration und der dynamischen Programmierung zusammen. Die Policy Evaluation und Policy Improvement Schritte können sich unterschiedlich abwechseln. Beide Methoden haben einzeln betrachtet zwei entgegengesetze Ziele. Denn wird eine neue Value Function bestimmt, so ist die aktuelle Policy für die neue Value Function nicht mehr die Optimale. Im anderen Fall wo die Policy sich greedy gegenüber der Value Function verhält, wird die alte Value Function obsolet und muss für die neu ermittelte Policy neu bestimmt werden. Dieses gegenseitige Ausspielen ermöglicht das Finden einer optimalen Value Function und einer optimalen Policy.

2.2.3. Temporal Difference Learning: Q-Learning

Anhand der Grundlagen der dynamischen Programmierung sind die zentralen Schritte für die Bestimmung einer Value Function und der Bestimmung einer verbesserten Policy definiert worden. Allerdings definiert die dynamische Programmierung lediglich ein theoretisches Framework, da die Bestimmung der Value Function und das Anpassen einer Policy für Umgebungen mit vielen Zuständen und großen Aktionsräumen sehr ineffizient ist.

Die **Monte-Carlo** und **Temporal Difference** Methoden sind konkrete Implementierungen der Policy Evaluation und Policy Improvement Methoden basierend auf den Grundlagen der dynamischen Programmierung. In Monte-Carlo werden diese als Monte-Carlo Evaluation und Monte-Carlo Control bezeichnet und im Temporal Difference Learning als **TD-Prediction** und **TD-Control**. Diese können im Gegensatz zur dynamischen Programmierung mit einem

akzeptablen Ressourcenverbrauch und mit einer akzeptablen Laufzeit eine Value Function V bestimmen und eine Policyverbesserung durchführen. Im Folgenden werden größtenteils Temporal Difference Learning Methoden behandelt, da die Monte-Carlo Methoden für diese Arbeit nicht relevant sind. Die Monte-Carlo Methoden werden lediglich im Zusammenhang mit Temporal Difference Learning behandelt.

Temporal Difference Learning besteht aus der Kombination der Ideen der Monte-Carlo Methoden und der dynamischen Programmierung. Wie Monte-Carlo Methoden können für Temporal Difference Methoden die Übergangswahrscheinlichkeiten der Umgebung vernachlässigt werden, da beide Methoden direkt aus gesammelter Erfahrung lernen. Temporal Difference Methoden sind allerdings nicht ausschließlich auf episodische Aufgaben beschränkt wie die Monte-Carlo Methoden. Stattdessen können TD-Methoden genau wie die dynamische Programmierung Schätzungen aus bereits bestehenden Schätzungen ermitteln (Bootstrapping).

Im Folgenden werden Policy Evaluation und Policy Improvement für Temporal Difference Learning erläutert. In Wirklichkeit unterscheiden sich die dynamische Programmierung, Monte-Carlo Methoden und Temporal Difference Learning Methoden lediglich im Policy Evaluation Teil, also für das Prediction Problem. Für den Policy Improvement Teil verhalten sich alle drei Algorithmen greedy gegenüber der Value Function.

Sample Backups und Inkrementelle Updates

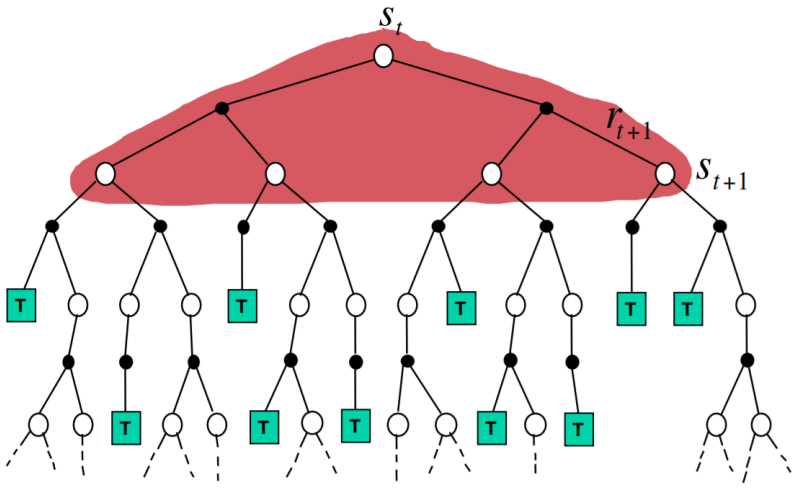


Abbildung 2.26.: Backupdiagramm für Bellman Optimality Equations Silver (2015)

Durch die Betrachtung der Aktualisierungsregel für die dynamische Programmierung fällt auf, dass bei einem Backup die Übergangswahrscheinlichkeiten zu allen Folgezuständen s' berücksichtigt werden müssen.

$$V_{t+1} = \sum_{a} \pi(a|s) \left[r(s, \pi(s)) + \gamma \sum_{s'} Pr(s'|s, \pi(s)) \, V_t(s') \right] \tag{2.32}$$

Wie in Abbildung 2.26 zu sehen bedeutet dies, dass die dynamische Programmierung für ein Backup eines Zustands s für jede mögliche Aktion a auch alle Folgezustände s' berücksichtigen muss. Dies macht die dynamische Programmierung ein sehr ineffizientes Verfahren. Monte-Carlo und Temporal Difference Methoden samplen hingegen lediglich eine Transition (s, a, r, s') und führen dann ein Update aus:

$$\begin{aligned} V(s_t) &= \mathbf{(1 - \alpha)} \, V(s_t) + \boldsymbol{\alpha} \, (r_{t+1} + V(s_{t+1})) \\ &= V(s_t) + \alpha \, [r_{t+1} + V(s_{t+1}) - V(s_t)] \end{aligned} \tag{2.33}$$

Die Updateregel für die TD-Prediction(Formel 2.33) ist eine exponentielle Glättung. Die Aktualisierungsregel aktualisiert die Value Function einen kleinen Schritt α in Richtung des TD-Errors $[r_{t+1} + V(s_{t+1}) - V(s_t)]$. Die Idee ist, dass durch das Besuchen vieler Zustände die Aktualisierungregel zwangsläufig die optimale Value Function V_* approximiert, obwohl im Vergleich zur dynamischen Programmierung nicht alle Folgezustände berücksichtigt werden. So ergibt sich für alle samplebasierten Vorhersagemethoden wie zum Beispiel Monte-Carlo Evaluation und TD-Prediction folgende allgemeine Aktualisierungsregel:

$$V_{t+1} = V_t + \alpha \, [Target - V_t] \tag{2.34}$$

Hier ist das sogenannte **Target** oder auch Ziel die Form in der, der ausgewählte Algorithmus die Samples aus der Umgebung bezieht. Für das obere Beispiel mit Temporal Difference Learning ist das Target: $r_t + \gamma V_t(s_{t+1})$. Abbildungen 2.27 und 2.28 zeigt die Backupdiagramme für die Monte-Carlo Evaluation und die TD-Prediction, dabei ist das Target jeweils rot markiert.

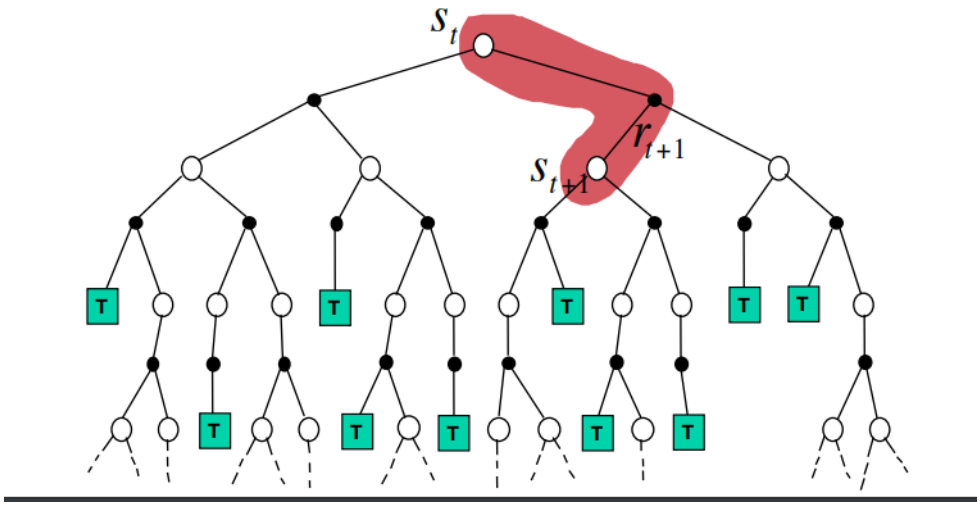


Abbildung 2.27.: Backupdiagramm für TD-Prediction Silver (2015)

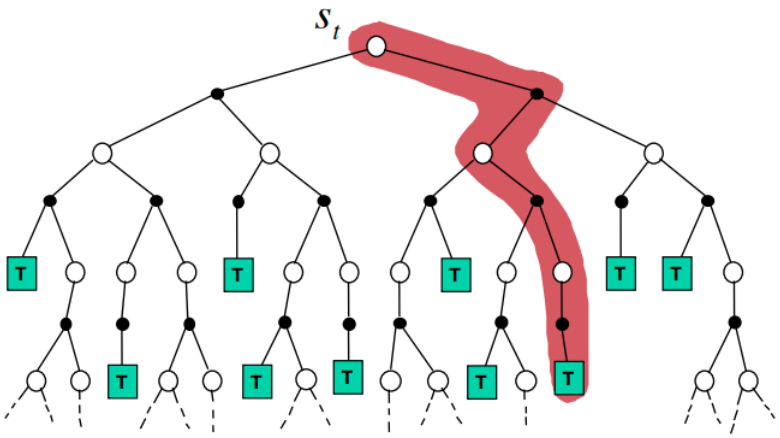


Abbildung 2.28.: Backupdiagramm für Monte-Carlo Evaluation Silver (2015)

TD Prediction

Anhand eines direkten Vergleichs mit der Monte-Carlo Evaluation kann die TD-Prediction erläutert werden. Beide Algorithmen lernen die Value Function V_π direkt aus Erfahrung, die nach dem Samplen nach einer Policy π gesammelt wurde. Monte-Carlo Methoden warten bis die gesammte Belohnung einer Episode bekannt ist, um ein Update der Value Function V_π durchzuführen:

$$V(s_t) = V(s_t) + \alpha \left[G_t - V(s_t)\right] \tag{2.35}$$

Hier ist G_t die gesammte Belohnung über eine Episode für einen Zustand s_t. Mit anderen Worten, wartet Monte-Carlo bis zum Ende der Episode damit G_t bekannt ist, um so das Update der Value Function $v(s_t)$ durchzuführen (Abbildung 2.28). Die TD-Prediction muss hingegen lediglich bis zum nächsten Zeitschritt warten um ein Update durchführen zu können.

$$V(s_t) = V(s_t) + \alpha \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right] \tag{2.36}$$

Die TD-Prediction betrachtet hingegen immer nur ein Zeitschritt pro Update. Die TD-Prediction ermittelt wie die dynamische Programmierung die Schätzung zur Aktualisierung der Value Function aus einer bereits bestehenden Schätzung $V(s_{t+1})$.

Die TD-Prediction ist tatsächlich eine Kombination der Schätzungen der Monte-Carlo Evaluation und der dynamischen Programmierung:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] && (2.37) \\ &= \mathbb{E}_\pi \left[\sum \gamma^k R_{t+k+1} | S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma \sum \gamma^k R_{t+k+2} | S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s \right] && (2.38) \\ & && (2.39) \end{aligned}$$

Die obige Gleichung ist die Bellman Equation aus Kapitel 2.2.1. Anhand der Bellman Equation kann der Zusammenhang zwischen der dynamischen Programierung, der Monte-Carlo und TD Methoden analysiert werden.

Auf den ersten Blick fällt auf, dass Monte-Carlo Methoden eine Schätzung der Gleichung 2.37 als Target benutzen, wohingegen die dynamische Programmierung eine Schätzung der Gleichung 2.38 als Target verwendet. Das Monte-Carlo Target G_t ist eine Schätzung, da der Erwartungswert in 2.37 unbekannt ist. Stattdessen wird G_t als Sample, also als Schätzung des Erwartungswerts verwendet. Analog ist das Target der dynamischen Programmierung eine Schätzung weil die echte Value Function v_π und somit der Wert $v_\pi(S_{t+1})$ unbekannt sind. Hier wird stattdessen die geschätzte Value Function v verwendet. TD-Methoden kombinieren beide Schätzungen und somit beide Methoden. Zum einen ist R_{t+1} eine Schätzung von 2.37 und zum anderen ist $v(s_{t+1})$ wie in der dynamischen Programmierung eine Schätzung von $v_\pi(S_{t+1})$. TD-Methoden kombinieren also das Sampling von Monte-Carlo und das Bootstrapping der dynamischen Programmierung. Tatsächlich kann die TD-Prediction um beliebige Sampling

und Bootstrapping Schritte erweitert werden. Dies wird als TD(λ) bezeichnet, wobei λ in diesem Fall die Anzahl der Sampling und Bootstrapping-Schritten bestimmt. Ein großes λ lässt die TD-Prediction wieder zu einer Monte Carlo Evaluation werden. Ein kleines λ entspricht hingegen wieder der dynamischen Programmierung.

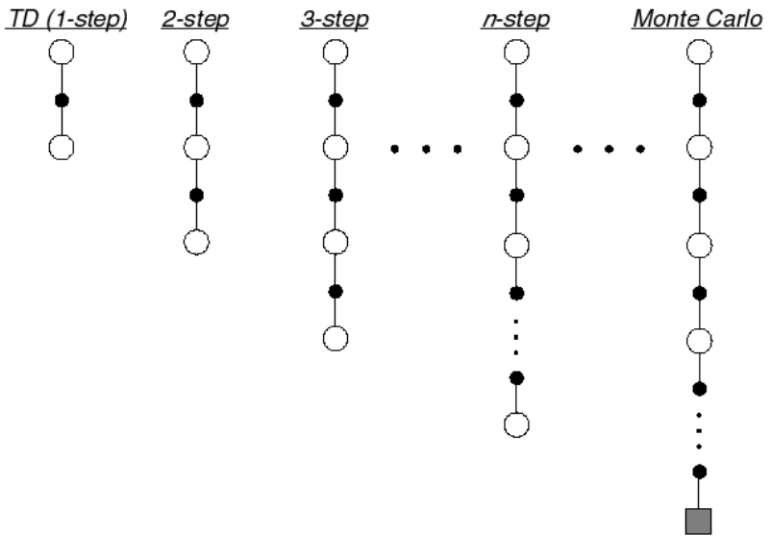


Abbildung 2.29.: TD(λ) Silver (2015)

Vorteile der TD Prediction

Die Eigenschaften der TD-Prediction machen diese besonders attraktiv für kontinuierliche Aufgaben, die in jedem Zeitschritt neue Informationen über die Umgebung brauchen:

- Kann nach jedem Zeitschritt eine neue Schätzung der Value Function abgeben.
- Anwendbar für nicht episodischen Aufgaben.
- Niedrige Varianz des Targets im Vergleich zu Monte-Carlo.
  - Monte-Carlo Target G_t hängt von vielen beliebigen Aktionen, Transitionen und Belohnungen ab.
  - TD Target hängt nur von einer Aktion, Transition und Belohnung ab.

Aus der Erfahrung von vielen Anwendungsfällen scheinen TD-Methoden für stochastische Aufgaben schneller als Monte-Carlo Methoden zu konvergieren.

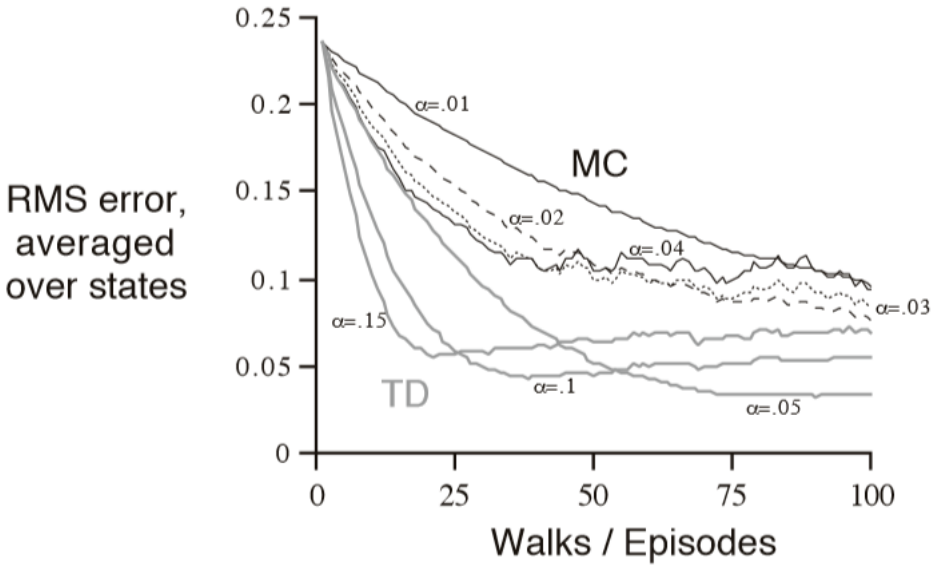


Abbildung 2.30.: Random Walk Beispiel Silver (2015)

Das obige Beispiel beschreibt einen Random Walk, wo der Agent das Ziel auf der rechten Seite finden soll. Die Aufgabe ist nicht diskontiert und episodisch. In diesem Fall ist die echte State-Value Function v_π die Wahrscheinlichkeit, dass das Ziel erreicht wird wenn von einem der Zustände gestartet wird. So ist $v_\pi(C_t) = 0.5$. Für Zustand A bis E sind $v_\pi = \frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}, \frac{5}{6}$. Abbildung 2.30 zeigt den Fehler zwischen der echten State-Value Function v_π und der approximierten State-Value Function v in Abhängigkeit der Anzahl der Episoden. Die TD-Methoden sind für verschiedene α durchgängig besser als die Monte-Carlo Methoden.

Obwohl bisher noch nicht konkret bewiesen worden ist, dass eine Methode schneller als die andere konvergiert, liegt die Vermutung bei der Art und Weise wie die jeweiligen Schätzungen der Value Function ermittelt werden. So generieren die Monte-Carlo Methoden Schätzungen die stets den Mean Squared Error minimieren, wohingegen TD-Methoden, den Maximum Likelihood Operator für das bestehende Modell finden, also die Daten die sich am besten an das

vorliegende Modell der Umgebung anpassen. Deswegen ist die Performance von TD-Methoden, bei Umgebungen, die stark von der Markov Property beschrieben werden, wie beispielsweise der Random Walk, der Performance der Monte-Carlo Methoden überlegen.

Q-Learning

Nachdem die TD-Prediction erläutert wurde wird das TD-Control in Form des Q-Learnings eingeführt. Q-Learning ist ein Algorithmus der unter der Klassifizierung der Generalized Policy Iteration fällt. So definiert Q-Learning die Approximation einer Action-Value Function Q und realisiert zudem den greedy Policy Improvement Schritt auf die Action-Value Function Q. Dabei ist das Q-Learning ein sogenannter Off-Policy Algorithmus. Dabei wird der Policy Improvement Schritt unabhängig vom Policy Evaluation Teil durchgeführt. Mit anderen Worten, die unmittelbare Aktion hat keinen direkten Einfluss auf das Bestimmen der neuen Policy. Der DDPG-Algorithmus besitzt ebenfalls diese Off-Policy Eigenschaft wie in den folgenden Kapitel erläutert wird.

Wie erwähnt zeichnen sich Off-Policy Algorithmen dadurch aus, dass deren Policy Evaluation Part unabhängig vom Policy Improvement Teil geschieht. Im Policy Evaluation Teil approximiert Q-Learning eine Action-Value Function Q:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{2.40}$$

Wie in der dynamischen Programmierung benötigt der Policy Improvement Schritt nach der Policy Evaluation eine Aussage darüber welche Aktion a die langfristige Belohnung maximiert. Die Action-Value Function liefert genau diese Information.

Während Q-Learning eine Policy π auf Basis der Action-Value Function Q_π ausführt, wird die neue Action-Value Function $Q_{\pi'}$ unabhängig von der aktuellen Policy approximiert. Die Policy Evaluation des Q-Learnings maximiert stattdessen über den gesammten Aktionsraum: $\max_a Q_\pi(S_{t+1}, a)$ (Abbildung 2.31 und Formel 2.40). Das heißt, während Q-Learning eine neue Action-Value Function bestimmt, übt die aktuelle Policy π keinen direkten Einfluss auf das Bestimmen der neuen Action-Value Function $q_{\pi'}$ aus. Die Policy π bestimmt lediglich welche Zustände besucht werden.

Abbildung 2.31.: Backupdiagramm für Q-Learning Sutton und Barto (2017)

Der Pseudocode des Q-Learning Algorithmus veranschaulicht die unabhängigen Policy Evaluation und Policy Improvement Schritte:

Algorithmus 1: Pseudocode für eine Q-Learning Implementierung

Initialisiere $Q(s, a)$, alle $s \in S$, $a \in A(s)$ beliebig
für *für jede Episode* **tue**
    Initialisiere S
    für *für jeden Zeitschritt in einer Episode* **tue**
        Aktion a aus S wählen anhand von Q
        `/* Policy Improvement */`
        Aktion a ausführen, R und S' beobachten
        `/* Policy Evaluation */`
        $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$
        $S \leftarrow S'$
    Ende
Ende

Aus dem Backupdiagramm in Abbildung 2.31 fällt auf, dass Q-Learning eine sample basierte Variante der Bellman Optimality Equation für q_* ist. Dies bedeutet, dass Q-Learning direkt die optimale Action-Value Function q_* approximiert und somit auch eine optimale Policy π_*. Diese Eigenschaft macht Q-Learning für stochastische Umgebungen nutzvoll, denn unabhängig von der verfolgten Policy kann Q-Learning unter der Annahme, dass genügende Zustände besucht werden immer noch die optimale Action-Value Function bestimmen und somit auch eine optimale Policy.

2.2.4. Value Function Approximation

In den letzten beiden Kapiteln wurden Methoden wie die dynamische Programmierung und das Temporal Difference Learning präsentiert, die es erlauben eine Value Function zu approximieren

und anhand dieser Value Function eine neue Policy Policy zu verbessern. Eine wichtige Frage in Bezug auf das Approximieren der Value Function bleibt allerdings noch offen. Mit welchen Modellen kann eine Value Function tatsächlich implementiert werden?

		Aktionen					
Zustände		0	1	2	3	4	5
	0	0	0	0	0	80	0
	1	0	0	0	64	0	100
$Q=$	2	0	0	0	64	0	0
	3	0	80	51	0	80	0
	4	64	0	0	64	0	100
	5	0	80	0	0	80	100

Abbildung 2.32.: Q-Table McCullock (2012)

Implizit wurde bisher die einfachste Art eine Value Function darzustellen präsentiert. Eine Lookup-Table wo jeder Zustand s einen Eintrag $V(s)$ besitzt oder für den Action-Value Fall ein Eintrag $Q(s, a)$ pro Zustands-Aktionspaar s, a (Abbildung 2.32). Für große Zustands und Aktionsräume ist die Lookup-Table nicht realisierbar denn:

- Die Anzahl der Zustände und Aktionen ist zu groß, um diese im Speicher zu haben.
- Die explizite Bestimmung der Value Function für alle Zustände ist zu langsam.

Stattdessen sollte die Value Function durch einen Funktionsapproximator approximiert werden:

$$\begin{aligned} \hat{v}(s, w) &\approx v_\pi(s) \\ \hat{q}(s, a, w) &\approx q_\pi(s, a) \end{aligned} \tag{2.41}$$

Mit Funktionsapproximatoren kann aus einer reduzierten Menge an Zuständen für alle anderen nicht besuchten Zustände generalisiert werden. Der Parametervektor w des Funktionsapproximators wird unter der Verwendung von Policy Evaluation Methoden modiziert, um eine State-Value Function $\hat{v}(s, w)$ bzw. eine Action-Value Function $\hat{q}(s, a, w)$ zu approximieren. Die Parameter $w_1 \dots w_n$ modellieren dabei die Value Function.

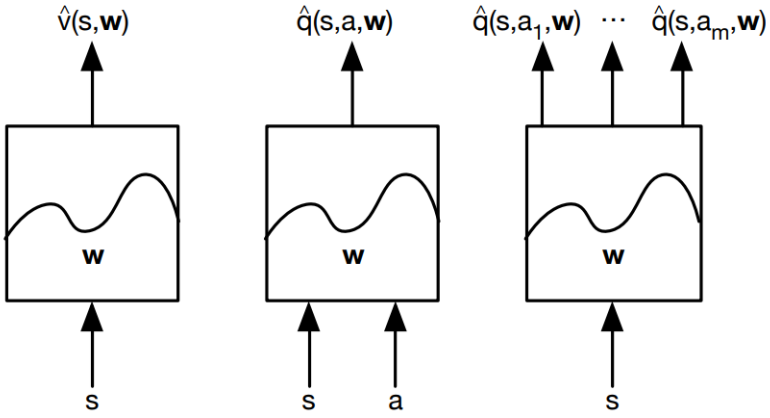


Abbildung 2.33.: Value Function Approximation Silver (2015)

Neuronale Netze als Funktionsapproximatoren

Welcher Funktionsapproximator verwendet wird ist vom zu lösenden Problem abhängig. So kommen lineare Funktionsapproximatoren wie beispielsweise radiale Basisfunktion in Frage. Nicht lineare Funktionsapproximatoren wie neuronale Netze werden seit dem Durchbruch des Deep Learnings zunehmend für große Zustands und Aktionsräume bevorzugt, da diese in der Lage sind Systeme mit einer großen Anzahl an Parametern zu approximieren. Das Trainieren des neuronalen Netzes kann wie für das Supervised Learning durch Stochastic Gradient Descent erfolgen. Mit einer Kostenfunktion $J(w)$ mit Parametervektor w, die eine Value Function approximiert und differenzierbar ist, kann der Gradient dieser Kostenfunktion folgendermaßen definiert werden:

$$\nabla_w J(w) = \begin{bmatrix} \frac{\partial J(w)}{\partial w_1} \\ \vdots \\ \frac{\partial J(w)}{\partial w_n} \end{bmatrix} \tag{2.42}$$

Der Parametervektor w wird dann in entgegengesetzer Richtung des Gradienten aktualisiert, um ein Minimum zu finden:

$$\triangle w = -\frac{1}{2}\alpha \nabla_w J(w) \tag{2.43}$$

Dies ist genau die Vorgehensweise der iterativen Parameterbestimmung, die in Kapitel 2.1 erläutert wurde. Für den konkreten Fall der Approximierung einer Value Function sehen die einzelnen Komponenten der iterativen Parameterbestimmung wie folgt aus:

- **Ziel:** Einen Parametervektor w finden der den Mean Squared Error zwischen der approximierten Value Function $\hat{V}(s, w)$ und der echten Value Function $V_\pi(s)$ minimiert:

$$J(w) = \mathbb{E}_\pi \left[(V_\pi(s) - \hat{V}(s, w))^2 \right] \tag{2.44}$$

- **Bestimmung des Gradienten** (Anwendung der Kettenregel):

$$\begin{aligned} \triangle w &= -\frac{1}{2} \alpha \nabla_w J(w) \\ &= \alpha \mathbb{E}_\pi \left[(V_\pi(s) - \hat{V}(s, w)) \nabla_w \hat{V}(s, w) \right] \end{aligned} \tag{2.45}$$

- **Stochastic Gradient Descent** sampled den Gradienten wodurch der Erwartungswert für die Policy entfällt und aktualisiert den Parametervektor w:

$$\triangle w = \alpha \left[V_\pi(s) - \hat{V}(s, w) \right] \nabla_w \hat{V}(s, w) \tag{2.46}$$

Bisher wurde stets angenommen, dass eine echte Value Function $v_\pi(s)$ als Musterbeispiel existiert. Im Fall der Value Function Approximierung ist das Musterbeispiel das jeweilige Target der verwendeten Approximierungsmethode. Für Temporal Difference Learning würde sich für TD(0) mit dem Target $r_{t+1} + \gamma \hat{v}(s_{t+1}, w)$ folgende Updateregel für den Parametervektor w ergeben:

$$\begin{aligned} \triangle w &= \alpha \left[R_{t+1} + \gamma \hat{V}(S_{t+1}, w) - \hat{V}(s, w) \right] \nabla_w \hat{V}(s, w) \\ &= \alpha \left[\text{TD-Error} \right] \nabla_w \hat{V}(s, w) \end{aligned} \tag{2.47}$$

Formel 2.47 ist in Wirklichkeit der TD-Error. Der Parametervektor w des neuronalen Netzes wird mit den ermittelten Gradienten der approximierten Value Function $\nabla_w \hat{V}(s, w)$ und dem TD-Error aktualisiert. Dabei bestimmt der TD-Error den Einfluss der Gradienten auf die Aktualisierung des Parametervektors w und somit auch wie aus der TD-Prediction bekannt die Richtung in der die approximierte Value Function angepasst werden soll. Gemäß der TD-Prediction sollte nach Besuchen einer ausreichenden Anzahl an Zuständen die approximierte Value Function gegen V_* konvergieren. Allerdings ist dies für neuronale Netze nicht der Fall:

Algorithm	Table Lookup	Linear	Non-Linear
Monte-Carlo Control	✓	(✓)	✗
Sarsa	✓	(✓)	✗
Q-learning	✓	✗	✗

Abbildung 2.34.: Backupdiagramm für Q-Learning (Angepasst von: Silver (2015))

In realen Anwendungsfällen konvergieren Control-Algorithmen wie Q-Learning mit neuronalen Netzen nicht. Neuronale Netze lernen am Besten wenn Musterbeispiele und Trainingsdaten nicht miteinander korrellieren. Für Supervised Learning ist dies einfach realisierbar, da die einzelnen Musterbeispiele vor dem Training ausgewählt werden und dies dadurch tatsächlich gewährleistet werden kann. Im Reinforcement Learning sind die ausgewählten Samples stets zeitlich korreliert was zu einer sehr hohen Varianz des TD-Errors führt. In Kapitel 2.3 wird die Verwendung eines Experience Replay Buffers erläutert. Das Experience Replay Buffer speichert alte Transitionen aus denen dann beliebig gesampled wird. So wird die Korrelierung von aufeinanderfolgenden Samples aufgehoben.

2.2.5. Policy Gradients und Actor-Critic Architekturen

Im letzten Kapitel wurde die Implementierung einer Value Function mit einem neuronalen Netz als Funktionsapproximator behandelt. Dabei wurden die State-Value oder die Action-Value Function anhand eines Parametervektors w modelliert.

$$\begin{aligned} \hat{v}(s, w) &\approx v_\pi(s) \\ \hat{q}(s, a, w) &\approx q_\pi(s, a) \end{aligned} \tag{2.48}$$

Eine neue Policy wird dann wie üblich über einen Policy Improvement Schritt generiert. Zum Beispiel mit einem greedy Verhalten gegenüber der Value Function.

In diesem Kapitel werden **Policy Gradients** und deren Implementierung in der Form einer Actor-Critic Architektur vorgestellt. **Policy Gradients approximieren eine Policy ohne Value Functions**. Stattdessen wird die Policy direkt mit einem Parametervektor θ parametrisiert.

$$\pi_\theta(s, a) = P[a|s, \theta] \tag{2.49}$$

Die Parameter θ bestimmen also direkt die Policy indem die Wahrscheinlichkeitsverteilung P für das Wählen einer Aktion a in Zustand s durch den Parametervektor $\vec{\theta}$ parametrisiert wird.

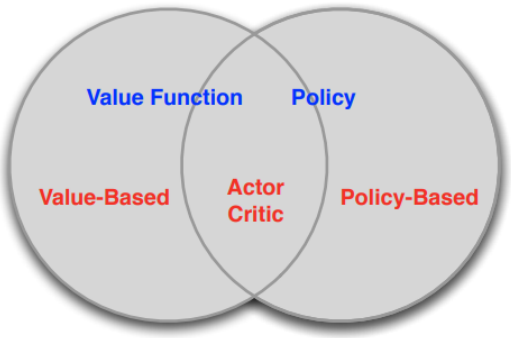


Abbildung 2.35.: Backupdiagramm für Q-Learning Silver (2015)

Zusammengefasst wird von Value-Based und Policy-Based Reinforcement Learning unterschieden:

- **Value Based**
  - Value Function wird gelernt.
  - Implizite Policyverbesserung durch Policy Improvement.
- **Policy Based**
  - Es wird keine Value Function gelernt.
  - Policy wird explizit gelernt und durch Gradienten direkt angepasst.

Policy Gradients werden zur Approximation des Actors in der Actor-Critic Architektur der Deep Deterministic Policy Gradients verwendet. Vor allem in hoch-dimensionalen und kontinuierlichen Aktionsräumen ist die Bestimmung der Policy durch Policy Gradients einer Value-Based Methode überlegen. Hauptgrund dafür ist, dass Algorithmen wie Q-Learning über den ganzen Aktionsraum das globale Maximum finden müssen. Policy Gradients können hingegen durch die Bestimmung des Policy Gradient die Policy direkt anpassen. In diesem Abschnitt werden zunächst stochastische Policies behandelt wodurch natürlich gilt, dass zum Anpassen der Policyparameter θ der **Stochastic Policy Gradient** ermittelt werden muss. Der Stochastic Policy Gradient dient dem Deterministic Policy Gradient als Grundlage, der ein spezieller Fall des Stochastic Policy Gradient ist wenn dieser deterministisch wird.

Kostenfunktion einer stochastischen Policy

Der Policy Gradient soll das Feedback liefern mit dem die Parameter θ einer Policy angepasst werden sollen. Dabei bleiben zwei Fragen offen. Wie wird festgestellt, dass eine Policy besser als eine andere ist? Und mit welchem Feedback soll der Parametervektor θ aktualisiert werden?

Policy Gradient Algorithmen definieren eine Kostenfunktion $J(\theta)$, die Policy Objective Function gennant wird. Diese Kostenfunktion definiert die Performance einer parametrisierten Policy und wird folgendermaßen definiert:

$$J(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s,a) R_s^a \tag{2.50}$$

Dabei ist d^{π_θ} die Verteilung aller Zustände für die Umgebung, $\pi_\theta(s,a)$ die Policy, die mit dem Parametervektor $\vec{\theta}$ parametrisiert wird und R_s^a die daraus resultierende Belohnung. Die Metrik der Policy Objective Function ist somit die durchschnittliche Belohnung pro Zeitschritt unter der Policy π_θ. Die Parameter θ sollen so modifiziert werden, dass die durchschnittliche Belohnung pro Zeitschritt $J(\theta)$ maximiert wird. Ziel ist es ein lokales Maximum für $J(\theta)$ durch Gradient Ascent zu finden:

$$\triangle\theta = \alpha \nabla_\theta J(\theta) \tag{2.51}$$

Wobei $\nabla_\theta J(\theta)$ der Policy Gradient ist.

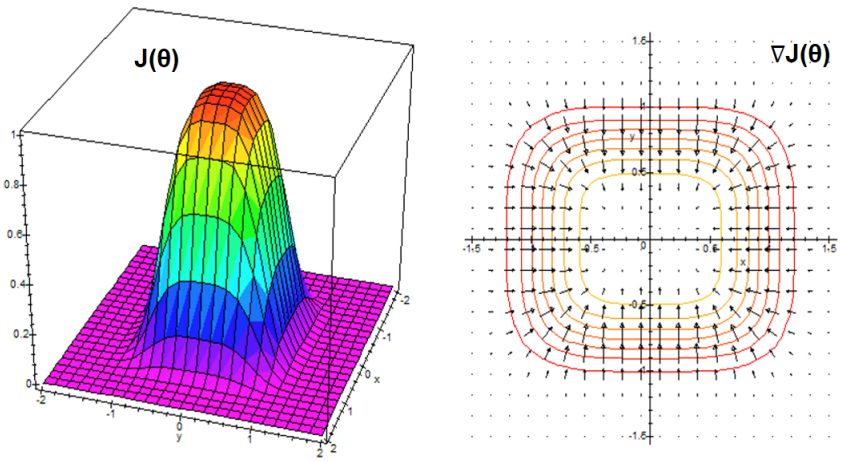


Abbildung 2.36.: Gradient Ascent

Der Gradient der Policy Objective Function kann nicht direkt für die Parameter θ bestimmt werden, da die Belohnung R_s^a nicht von den Parametern θ abhängt und somit auch kein Gradient ermittelt werden kann. Durch Umformen des Gradienten der Policy Objective Function kann ein Zusammenhang zu den Parametern θ hergestellt werden:

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_s d(s) \sum_a \pi_\theta(s|a) R(s,a) \tag{2.52}$$

$$= \sum_s d(s) \sum_a \nabla_\theta \pi_\theta(s|a) R(s,a) \tag{2.53}$$

$$= \sum_s d(s) \sum_a \pi_\theta(s|a) \frac{\nabla_\theta \pi_\theta(s|a)}{\pi_\theta(s|a)} R(s,a) \tag{2.54}$$

$$= \sum_s d(s) \sum_a \pi_\theta(s|a) \log \nabla_\theta \pi_\theta(s|a) R(s,a) \tag{2.55}$$

$$= \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(s|a) R(s,a) \right] \tag{2.56}$$

Durch Ausnutzen der Eigenschaft $\frac{\nabla_\theta \pi_\theta(s|a)}{\pi_\theta(s|a)} = \nabla_\theta \log \pi_\theta(s|a)$ ergibt sich der Gradient der Likelihood-Function auch Score-Function genannt. Der Gradient der Likelihood-Function kann ermittelt werden. Für eine parametrisierte Policy beschreibt die Likelihood-Function welche Parameterwerte am besten eine vorliegende Policy beschreiben.

Intuitiv stellt der Gradient der Likelihood-Function den Einfluss eines Parameters auf die bestehende Verteilung der Policy dar. Die ermittelten Gradienten werden zusätzlich mit der unmittelbaren Belohnung gewichtet, so dass je nachdem wie gut eine bestimmte Aktion für die Maximierung der Belohnung ist die jeweiligen Aktionen auch dementsprechend gewichtet werden.

Der Stochastic Policy Gradient kann auch für beliebige MDP's definiert werden. Dies wird durch das **Policy Gradient Theorem** definiert:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s|a) V^\pi(s,a)] \tag{2.57}$$

Das Policy Gradient Theorem ersetzt, die unmittelbare Belohnung $R(s,a)$ mit einem langfristigeren Wert einer Value Function $V^\pi(s,a)$, der die langfristige Belohnung in einem Markov Decision Process beschreibt. Der Value V^π nimmt wie die unmittelbare Belohnung $R(s,a)$ Einfluss auf das Ausmaß des Parameterupdates.

Actor-Critic Policy Gradient

Das direkte Bestimmen des Stochastic Policy Gradient anhand der unmittelbaren Belohnung hat allerdings einen großen Nachteil: Dies ist mit einer hohen Varianz verbunden. Eine Möglichkeit dies zu lösen ist die Einführung eines sogenannten Critics. Der Critic approximiert den Value $V\pi$ anhand einer separaten Value Function und bewertet wie erwähnt den Gradienten der Policy mit diesem skalaren Value V_π.

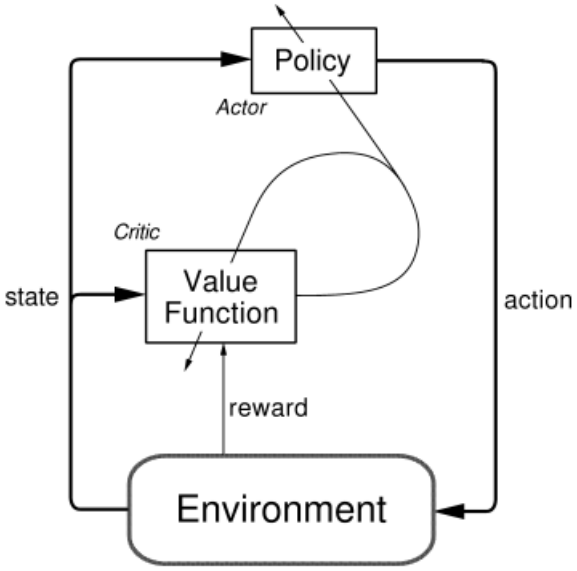


Abbildung 2.37.: Architektur des Actor-Critic Sutton und Barto (2017)

Die Value Function V^π wird direkt durch den Critic approxmiert:

$$V_w(s) \approx V^{\pi_\theta}(s) \tag{2.58}$$

Actor-Critic Algorithmen arbeiten wie folgt:

- **Actor:** Aktualisiert die Parameter θ der Policy mit dem Policy Gradient.
- **Critic:** Approximiert eine Action-Value Function mit Parametervektor w durch bekannte GPI-Methoden wie z.B Temporal Difference Learning.

Der Action-Value wird für den Actor als Verstärkungsfaktor des Gradienten verwendet.

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s|a)\ V_w(s)] \tag{2.59}$$

$$\triangle\theta = \alpha \nabla_\theta \log \pi_\theta(s|a)\ V_w(s) \tag{2.60}$$

Die approximierte Value Function des Critics liefert einen stabileren Belohnungswert als die unmittelbare Belohnung. Dadurch wird auch die Varianz bei der Anpassung der Parameter reduziert.

Mit dem Ende dieses Kapitels sind alle Grundlagen des Reinforcement Learnings präsentiert worden, die für das Verständnis der Deterministic Policy Gradients erforderlich sind. Die Actor-Critic Architektur ist ein Hauptbestandteil des in den Deterministic Policy Gradients beschriebenen Algorithmus zur Bewältigung von kontinuierlichen Zustands und Aktionsräumen. Im nächsten Kapitel werden die Deterministic Policy Gradients eingeführt, die auf alle bisherigen erläuterten Konzepte zurückgreifen. Zum Schluss wird die konkrete Umsetzung der Deterministic Policy Gradients in einem Algorithmus vorgestellt. Dies ist der Deep Deterministic Policy Gradient Algorithmus (DDPG), der in dieser Arbeit zum modellieren des Regelungssystems für die Steuerung eines Flugzeugs verwendet wird.

2.3. Steuerung in kontinuierlichen Aktionsräumen

Einer der schwierigsten Aufgaben im Reinforcement Learning ist das Lösen von Problemen in hoch-dimensionierten und stochastischen Umgebungen. Das Deep-Q-Network in **Human-level control through deep reinforcement learning** Mnih u. a. (2015) hat es bisher geschafft mit Convolutional Neural Networks aus Krizhevsky u. a. (2012) eine menschenähnliche Performance zur Steuerung von Atari-Videospielen anhand von Pixel als Inputs zu erlangen. Um dies zu erreichen wurden Convolutional Neural Networks als Funktionsapproximatoren im Zusammenhang mit Q-Learning verwendet.

Obwohl DQN für Aufgaben mit hoch-dimensionierten Zustandsräumen geeignet ist, kann DQN lediglich diskrete Aktionsräume behandeln. Bei vielen Regelungsaufgaben kann der Aktionsraum nicht ohne weiteres diskretisiert werden. Hoch-dimensionale und kontinuierliche Aktionsräume werden DQN zum Verhängnis, weil dieses Q-Learning als Approximierungsmethode verwendet. Das heißt, für jeden Zeitschritt muss der Algorithmus die Aktion auswählen, die die Action-Value Function global maximiert, was in einem hoch-dimensionierten Aktionsraum eine ressourcenintensive Aufgabe und meistens nicht vollständig realisierbar ist.

Im Folgenden werden der Deterministic Policy Gradient und eine dazugehörige Off-Policy Actor-Critic Architektur eingeführt, die anhand von zwei neuronalen Netzen als Funktions-

approximatoren in der Lage sind Policies für hochdimensionale Aktionsräume mit einem kontinuierlichen Wertebereich zu lernen.

2.3.1. Deterministic Policy Gradients

Der Deterministic Policy Gradient knüpft an dem im vorherigen Abschnitt erläuterten Stochastic Policy Gradient an:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s|a)\ V^\pi(s)] \tag{2.61}$$

Der Stochastic Policy Gradient ist der Gradient einer stochastischen Policy, die aus einer Wahrscheinlichkeitsverteilung $\pi_\theta(s,a) = P[a|s,\theta]$ mit Parametern θ eine Aktion a selektiert. Durch die Betrachtung der Kostenfunktion der stochastischen Policy wird klar, dass die stochastische Policy für den kontinuierlichen Fall über den gesamten Aktionsraum integrieren muss wodurch wie für das Q-Learning das Ermitteln des Gradienten ressourcenintensiv ist und viele Samples erfordert:

$$\begin{aligned} J(\theta) &= \sum_s d^{\pi\theta}(s) \sum_a \pi_\theta(s,a) R_s^a \quad \text{(disk. Wertebereich)} \\ J(\theta) &= \int_S d^{\pi\theta}(s) \int_A \pi_\theta(s,a) R_s^a \, da ds \quad \text{(kont. Wertebereich)} \end{aligned} \tag{2.62}$$

Stattdessen wird der Deterministic Policy Gradient für eine deterministische Policy $\mu_\theta : S \to A$ ermittelt, die Aktionen A direkt auf Zustände S mappt und somit nicht über den gesammten Aktionsraum integrieren muss. Dabei wird festgestellt, dass der Deterministic Policy Gradient der zu erwartende Gradient einer Action-Value Function Q ist. Darüber hinaus ist der Deterministic Policy Gradient ein spezieller Fall des Stochastic Policy Gradient und besitzt somit auch alle Eigenschaften des Stochastic Policy Gradient, wie das Verwenden einer Actor-Critic Architektur. Anschließend wird ein Off-Policy Actor-Critic Algorithmus vorgestellt, dessen Actor eine deterministische Policy besitzt und mit dem Gradient des Deterministic Policy Gradient angepasst wird, der mit Hilfe des Critics bestimmt wird.

Gradient der Action-Value Function

Die Verwendung von Q-Learning oder die Bestimmung des Stochastic Policy Gradient erweist sich in kontinuierlichen Aktionsräumen als problematisch. Q-Learning muss wie erwähnt global über den gesammten Aktionsraum maximieren. Das Bestimmen des Policy Gradient für

eine direkt parametrisierte Policy hat hingegen eine hohe Varianz und das Integrieren über den gesamten Aktionsraum erschweren die Konvergenz der Policy.

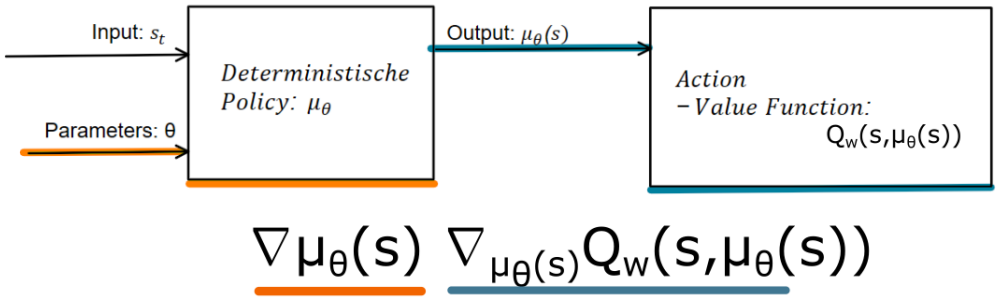


Abbildung 2.38.: Kettenregel für Gradienten der Action-Value Function mit Parameter θ für 1 Aktionsdimension (Eigene Erstellung)

Die Kombinierung beider Ansätze kann diese Problematik lösen. Zur Bestimmung des Policy Gradient für eine deterministische Policy werden dessen Parameter θ in die allgemeine Richtung des Gradienten der Action-Value Function $\nabla_\theta Q_{\mu_k}(s, \mu_\theta(s))$ angepasst, die eine Aktion $\mu_\theta(s)$ der deterministischen Policy entgegennimmt. Dafür wird für jedes Parameterupdate der Erwartungswert des Gradienten der Action-Value Function mit den Parametern θ der Policy genommen:

$$\theta_{k+1} = \theta_k + \alpha \mathbb{E}_{d^{\pi\mu}} \left[\nabla_\theta Q_{w_k}(s, \mu_\theta(s))\right] \tag{2.63}$$

Mit der Kettenregel:

$$\theta_{k+1} = \theta_k + \alpha \mathbb{E}_{d^{\pi\mu}} \left[\nabla_\theta \mu_\theta(s) \quad \nabla_{\mu_\theta(s)} Q_{w_k}(s, \mu_\theta(s))\right] \tag{2.64}$$

$$\theta_{k+1} = \theta_k + \alpha \mathbb{E}_{d^{\pi\mu}} \begin{bmatrix} \nabla_{\theta_1}\mu_\theta(s)_1 & \cdots & \cdots & \nabla_{\theta_1}\mu_\theta(s)_4 \\ \nabla_{\theta_2}\mu_\theta(s)_1 & \cdots & \cdots & \nabla_{\theta_2}\mu_\theta(s)_4 \\ \nabla_{\theta_3}\mu_\theta(s)_1 & \cdots & \cdots & \nabla_{\theta_3}\mu_\theta(s)_4 \\ \nabla_{\theta_4}\mu_\theta(s)_1 & \cdots & \cdots & \nabla_{\theta_4}\mu_\theta(s)_4 \end{bmatrix} \begin{bmatrix} \nabla_{\mu_\theta(s)_1} Q_{w_k}(s, \mu_\theta(s)) \\ \nabla_{\mu_\theta(s)_2} Q_{w_k}(s, \mu_\theta(s)) \\ \nabla_{\mu_\theta(s)_3} Q_{w_k}(s, \mu_\theta(s)) \\ \nabla_{\mu_\theta(s)_4} Q_{w_k}(s, \mu_\theta(s)) \end{bmatrix}^T$$

Der Gradient der Action-Value Function mit den Parametern θ ist also der Gradient der deterministischen Policy für jede Aktionsdimension $\nabla_\theta \mu_\theta(s)_d$ mit dessen Parametern θ multipliziert mit dem Gradienten der Action-Value Function mit den jeweiligen Aktionen $\mu_\theta(s)_d$ der

deterministischen Policy. $\nabla_\theta \mu_\theta(s)$ ist dann eine Jacobi Matrix wo jede Spalte der Gradient für eine Aktion $\mu_\theta(s)_d$ mit den Parametern θ ist, wobei d die Aktionsdimension ist. Im folgenden Schritt wird bewiesen, dass der Action-Value Gradient mit den Parametern θ tatsächlich dem Deterministic Policy Gradient entspricht.

Deterministic Policy Gradient Theorem

Als nächstes wird das Deterministic Policy Gradient Theorem definiert. Es soll ein Zusammenhang zwischen dem Gradienten der Kostenfunktion einer deterministischen Policy $\mu_\theta : S \to A$ und dem Gradienten der Action-Value Function mit den Parametern θ der Policy gefunden werden. Das letztendliche Ziel ist es anhand der Action-Value Function den Determinstic Policy Gradient ermitteln zu können. Zunächst wird die Kostenfunktion für eine deterministische Policy definiert:

$$\begin{aligned} J(\mu_\theta) &= \int_S d^{\pi\theta}(s) r(s, (\mu_\theta(s)) ds \\ &= \mathbb{E}_{d^{\pi\mu}}[r(s, \mu_\theta(s))] \end{aligned} \tag{2.65}$$

Da eine deterministische Policy direkt eine Aktion a auf einen Zustand s abbildet, fällt das Integral über den Aktionsraum weg. (Silver et al.) haben bewiesen, dass der Gradient der Kostenfunktion einer deterministischen Policy dem Gradient der Action-Value Function von 2.64 entspricht. Der vollständige Beweis kann dem Anhang entnommen werden:

$$\begin{aligned} \nabla_\theta J(\mu_\theta) &= \int_S d^{\pi\theta}(s) \nabla_\theta \mu_\theta(s) \quad \nabla_{\mu_\theta(s)} Q_w(s, \mu_\theta(s)) ds \\ &= \mathbb{E}_{d^{\pi\mu}} \left[\nabla_\theta \mu_\theta(s) \quad \nabla_{\mu_\theta(s)} Q_w(s, \mu_\theta(s)) \right] \end{aligned} \tag{2.66}$$

Dies ist genau der Gradient der Action-Value Function, der Aktionen einer deterministischen Policy entgegennimmt. Mit anderen Worten, kann der Deterministic Policy Gradient mit Hilfe einer Action-Value Function bestimmt werden.

Stochastic Policy Gradient und Deterministic Policy Gradient

Im letzten Schritt wird festgestellt, dass der Deterministic Policy Gradient ein Grenzfall des Stochastic Policy Gradient ist und somit alle Werkzeuge des Stochastic Policy Gradient wie zum Beispiel die Actor-Critic Architektur auch für den Deterministic Policy Gradient verwendet werden können.

Gegeben sei eine stochastische Policy $\pi_{\mu_\theta, \sigma}$, die durch eine deterministische Policy $\mu_\theta : S \to A$

und einen Varianzparameter σ definiert wird. Der Varianzparameter bestimmt die Streuung der Policy $\pi_{\mu_\theta}(s, a)$. Für $\sigma = 0$ wird die stochastische Policy zu einer deterministischen Policy. Dann gilt:

$$\lim_{\sigma \downarrow 0} \nabla_\theta J(\pi_{\mu_\theta}, \sigma) = \nabla_\theta J(\mu_\theta) \tag{2.67}$$

Silver u. a. (2014) hat bewiesen, wenn $\sigma \to 0$ dann ist der Stochastic Policy Gradient gleich dem Deterministic Policy Gradient. Der Deterministic Policy Gradient ist also eine spezieller Fall genauer gesagt ein Grenzfall des Stochastic Policy Gradient wodurch alle Eigenschaften des Stochastic Policy Gradient auch für den Deterministic Policy Gradient gelten.

Off-Policy Deterministic Actor-Critic

Mit der Erkenntnis, dass der Deterministic Policy Gradient durch das Bestimmen des Gradienten einer Action-Value Function mit den Parametern θ der Policy ermittelt werden kann, wird nun ein Off-Policy Actor-Critic definiert.

Der Actor verfolgt eine stochastische Policy $\pi(s, a)$, wertet aber eine deterministische Policy μ aus, dessen Gradienten nachwievor von einer Action-Value Function $Q_\mu(s, \mu_\theta(s))$ bestimmt wird. Zum Beispiel wird die Ausgabe des Actors durch einen Rauschterm ergänzt, der Stochastizität beim Sondieren des Zustandsraums einführt. Die Parameter der deterministischen Policy μ_θ werden allerdings nach dem Deterministic Policy Gradient angepasst:

$$\begin{aligned} \nabla_\theta J_\pi(\mu_\theta) &\approx \int_S d^\pi(s)\, \nabla_\theta \mu_\theta(s)\, Q_w(s, \mu_\theta(s))\, ds \\ &= \mathbb{E}_{d^\pi}\left[\nabla_\theta \mu_\theta(s)\, \nabla_{\mu_\theta} Q_w(s, \mu_\theta(s))\right] \end{aligned} \tag{2.68}$$

Für den Off-Policy Actor-Critic ergeben sich dann folgende Updateregeln:

$$\delta_t = r_t + \gamma Q_w(s_{t+1}, \mu_\theta(s_{t+1})) - Q_w(s_t, a_t) \tag{2.69}$$

$$w_{t+1} = w_t + \alpha_w\, \delta_t\, \nabla_w Q_w(s_t, a_t) \quad \textbf{(Update Critic)} \tag{2.70}$$

$$\theta_{t+1} = \theta_t + \alpha_\theta\, \nabla_\theta \mu_\theta(s)\, \nabla_{\mu_\theta} Q_\mu(s, \mu_\theta(s)) \quad \textbf{(Update Actor)} \tag{2.71}$$

Der Off-Policy Actor-Critic ist die Basis für den im nächsten Kapitel behandelten Algorithmus, den Deep Deterministic Policy Gradients. Hier werden die deterministische Policy des Actors und die Value Function des Critics durch zwei neuronale Netze approximiert. Deep Deterministic Policy Gradients ist ein Verfahren des Deep Learnings, da die verwendeten neuronalen Netze mehrere Schichten besitzen. Zudem wurden zwei wichtige Komponenten aus dem Deep-

Q-Network Paper übertragen. Diese Erweiterungen ermöglichen es, dass neuronale Netze als Funktionsapproximatoren für direkt parametrisierte Policies und Action-Value Functions tatsächlich konvergieren.

2.3.2. Deep Deterministic Policy Gradients (DDPG)

Im letzten Kapitel wird der DDPG-Algorithmus behandelt. Der DDPG-Algorithmus verwendet die dargelegte Theorie des Deterministischen Policy Gradient, um einen Agenten in einem kontinuierlichen Zustands- und Aktionsraum zu steuern. Dabei wird konkret der Off-Policy Actor-Critic mit Hilfe von zwei neuronalen Netzen als Funktionsapproximatoren implementiert.

Der Actor approximiert eine deterministische Policy μ_θ, die mit einem Zufallsprozess N Stochastizität zur Erforschung der Umgebung einführt. Der Critic approximiert eine Action-Value Function Q_w, die den Gradienten zur Anpassung von μ_θ liefert. Die Actor-Critic Architektur wird Off-Policy trainiert. Dabei werden die Transitionen der deterministischen Policy mit stochastischem Störfaktor zunächst in einem Buffer gespeichert aus dem anschließend Mini-Batches gesampled werden, um die Actor und Critic Netze zu trainieren.

Der Actor wird nachwievor mit dem Deterministic Policy Gradient aktualisiert:

$$\nabla_\theta J(\theta) = \mathbb{E}_{d^\pi} \left[\nabla_\theta \mu_\theta(s) \, \nabla_{\mu_\theta} Q_w(s, \mu_\theta(s)) \right] \tag{2.72}$$

Der Gradient zur Anpassung der Action-Value Function Q_w wird durch die Kostenfunktion des Critics ermittelt, wie in Kapitel 2.2.4 vorgestellt:

$$J(w) = \left[(Target - Q_w(s_t, a_t))^2 \right] \tag{2.73}$$

Das Target für die Kostenfunktion der Action-Value Function ist dann das Temporal Difference Target:

$$Target = r(s_t, a_t) + \gamma Q_w(s_{t+1}, \mu_\theta(s_{t+1})) \tag{2.74}$$

In Kapitel 2.2.4 wurde festgestellt, dass die Approximation einer Value Function mit nichlinearen Funktionsapproximatoren wie neuronale Netze nicht konvergiert. Das Problem ist zwei Phänomenen geschuldet: **Zum einen korrelieren aufeinanderfolgende ermittelte Samples in einer kontinuierlichen Umgebung stark miteinander.** Sind die gesammelten Daten untereinander nicht unkorreliert genug steigt die Gefahr, dass der Gradient in lokalen Minima stecken bleibt. **Zum anderen führt die hohe Varianz des TD-Errors dazu, dass**

die Parameter des neuronalen Netzes mit der gleichen hohen Varianz aktualisiert werden, wodurch die Lernfähigkeit stark beeinträchtigt wird.

Der DDPG-Algorithmus erweitert den Off-Policy Actor-Critic, um zwei Komponenten, die die oben beschriebenen Phänomene mitigieren. Die erste Komponente ist der bereits angedeutete **Experience Replay Buffer**, der alte Transitionen speichert aus denen die Actor-Critic Architektur dann beliebige Samples zu einem Mini-Batch zusammenfühhrt. Die zweite Komponente besteht aus zwei separaten **Target-Networks**, die eine Kopie der Actor und Critic Netze sind. Deren Parameter werden während des Trainings nur verzögert aktualisiert, wodurch die Varianz des TD-Errors reduziert wird. Beide Komponenten werden im Folgenden näher erläutert.

Experience Replay Buffer

Der Experience Replay Buffer ist ein Buffer mit einer fest definierten Größe N, der alte Transitionen der Form (s_t, a_t, r_t, s_{t+1}) des Actors speichert. Der Buffer ist ein Ringbuffer. Wenn dieser voll ist werden die ältesten Samples verworfen und das neueste Sample an vorderster Stelle hinzugefügt.

Abbildung 2.39.: Experience Replay Buffer (Eigene Erstellung)

In jedem Zeitschritt wird der Buffer mit einer neuen Transition gefüllt. Nach jedem Speichern einer Transition werden die Actor und Critic Netze Off-Policy trainiert. Dazu wird ein Mini-Batch mit einer fest definierten Größe m aus dem Experience Replay Buffer entnommen, der aus beliebigen Samples besteht. Dadurch wird die Korrelation zwischen den Samples größtenteils aufgehoben, da die neuronalen Netze nicht mit aufeinanderfolgenden Transitionen

trainiert werden. Es ist wünschenswert einen sehr großen Buffer zu definieren, da so große Mengen an zeitlich unkorrelierten Samples gespeichert werden können.

Target-Networks

Um die hohe Varianz des TD-Errors zu stabilisieren werden zwei separate Target-Networks für den Actor und Critic definiert (μ'_θ und Q'_w). Diese Netze sind Kopien der originalen Actor und Critic Netze und deren Parameter werden nur verzögert aktualisiert. Alle Ausgaben zur Berechnung der Gradienten werden aus den Target-networks entnommen. Dafür wird der Kopiervorgang derParameter der echten Funktionsapproximatoren mit einer exponentiellen Glättung verzögert zu den Parametern der Target-Networks kopiert:

$$\begin{aligned} \theta' &\leftarrow \tau\theta + (1-\tau)\theta' \\ w' &\leftarrow \tau w + (1-\tau)w' \end{aligned} \tag{2.75}$$

Dies bietet während des Lernens stabile Parameterwerte. Die Verzögerung der Aktualisierung der jeweiligen Netzparameter verlangsamt allerdings das Lernen, da ermittelte Schätzungen langsamer propagiert werden. Nichtsdestotrotz, ist dieser Schritt ausschlaggebend, um irgendeine Form von Konvergenz mit neuronalen Netzen in einer Reinforcement Learning Umgebung zu erzielen.

Implementierung des DDPG-Algorithmus

Algorithmus 2: Pseudocode für DDPG-Algorithmus

Initialisiere Critic $Q_w(s, a)$ und Actor $\mu_\theta(s)$ mit beliebigen Gewichten w und θ
Initialisiere Target-Networks Q' und μ' mit Gewichten $w' \leftarrow w, \theta' \leftarrow \theta$
Initialisiere Experience Replay Buffer R
für *für jede Episode* **tue**
    Initialisiere N für Erforschung des Agenten
    Ersten Zustand s_1 samplen
    für *jeden Zeitschritt* t *in Episode* **tue**
        Wähle Aktion $a_t = \mu_\theta(s_t) + N_t$
        Führe a_t aus und speichere Belohnung r_t und Folgezustand s_{t+1}
        Speichere Transition (s_t, a_t, r_t, s_{t+1}) in Replay Buffer R
        Ein beliebiges Mini-Batch aus N Transitionen (s_i, a_i, r_i, s_{i+1}) aus R samplen
        Setze $Target = r_i + \gamma Q'_{w'}(s_{i+1}, \mu'_{\theta'}(s_{i+1}))$
        Aktualisiere Critic durch Minimierung der Kostenfunktion:

$$J(w) = \frac{1}{N} \sum_i (Target_i - Q_w(s_i, a_i))^2$$

        Aktualisiere Aktorpolicy anhand des gesampleten Deterministic Policy Gradient:

$$\nabla_\theta J \approx \frac{1}{N} \sum_i \nabla_\theta \mu_\theta(s_i) \quad \nabla_{\mu_\theta(s_i)} Q_w(s_i, \mu_\theta(s_i))$$

        Aktualisiere Parameter der Target Networks:

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$$
$$w' \leftarrow \tau w + (1 - \tau)w'$$

    Ende
Ende

Anhand des Pseudocodes ist die Inkorporierung des Experience Replay Buffers und der Target Networks zu sehen. Es werden Mini-Batches von Transitionen aus dem Experience Replay Buffer gesampled, die aus zeitlich unkorellierten Transitionen bestehen **(Zeile 8 - 10)**. Der Actor und Critic trainieren dabei anhand der stabilen Parameter der Target-Networks μ'_θ und Q'_w **(Zeile 11 - 12)**. Letztendlich werden die Gradienten der Actor und Critic Netze berechnet und deren Parameter aktualisiert **(Zeile 13 - 15)**.

2.4. Reward Shaping: Modellierung einer Belohnungsfunktion

Das letzte Kapitel der Grundlagen widmet sich der Modellierung einer Belohnungsfunktion, die anhand der erhaltenen Zustände aus der Umgebung Belohnungen formen soll, die den Agenten das Erreichen eines gewünschten Zieles ermöglichen.

Grundsätzlich ist das Modellieren einer Belohnungsfunktion kein formalisierter Vorgang. Im Gegensatz zum tabula rasa Ansatz des Reinforcement Learnings benötigt das Modellieren einer Belohnungsfunktion in der Regel Expertenwissen der Umgebung. Das heißt, die Umgebung muss genau analysiert werden, um aus dieser die optimalen Belohnungssignale zu gewinnen.

Obwohl die Definition einer Belohnungsfunktion keine exakte Wissenschaft ist, können anhand verschiedener Konzepte und Richtlinien die richtigen Schlussfolgerungen getroffen werden, um eine für die Aufgabe effektive Belohnungsfunktion zu modellieren. Ein Konzept das die Basis für die Modellierung von Belohnungsfunktionen bildet ist das Reward Shaping, welches ürsprunglich aus der Psychologie stammt.

Shaping und Reward Shaping

Das Shaping wurde als erstes vom Psychologen Burrhus Frederic Skinner eingeführt. **Dabei definiert das Shaping, dass zunächst alle Teilaufgaben, die das zu lösende Problem ausmachen identifiziert werden müssen. Durch das spezifische Belohnen dieser Teilaufgaben kann das Gesammtverhalten des Agenten modelliert werden.**

Im Reinforcement Learning wird das Shaping als ein separater Skalar F behandelt (Abbildung 2.40), der zu der eigentlichen Belohnung aus der Umgebung addiert wird. Dabei soll F die Belohnung so ergänzen, dass das Verhalten des Agenten akzentuiert belohnt wird:

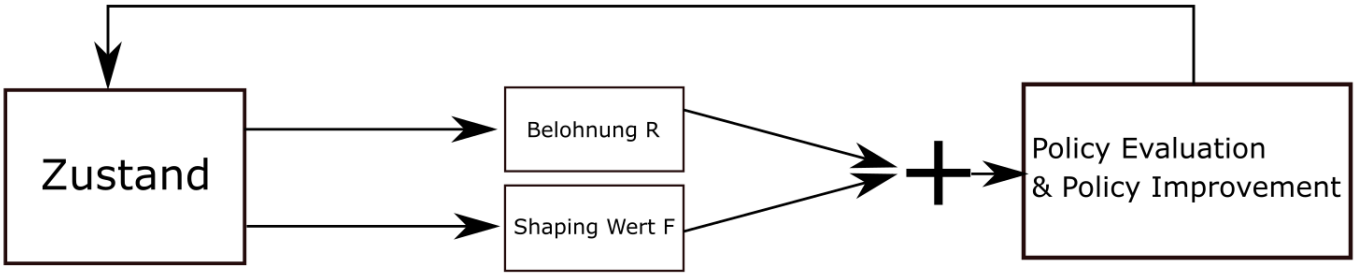


Abbildung 2.40.: Belohnungssignale für die Bewegung eines Roboters (Eigene Erstellung)

Wird dieser Shapingwert so eingesetzt, dass der Agent dadurch diese Teilaufgaben werden, dann kann ein Reinforcement Learning Algortihmus auf größere und komplexere Probleme skaliert werden. Da das Definieren der Teilaufgaben allerdings von Expertenwissen der Umgebung abhängig ist steigt die Gefahr, dass ein Problem überspezifiziert wird. Dies kann dazu führen, dass eine Lösung des Problems gefunden wird, die nur lokal optimal ist.

Im Laufe der Jahre haben sich 3 grundlegende Formen eine Belohnungsfunktion zu modellieren herauskristalisiert:

1. Das Zusammenfügen von mehreren kleinen Aktionen (Mikroaktionen) zu größeren Aktionen (Makroaktionen).
2. Ein mathematisches Modell, dass das gewünschte Verhalten des Agenten approximiert.
3. Eine mehrstufige Architektur die Stück für Stück trainiert wird.

In der Regel treten diese Formen kombiniert auf und nicht als einzelne Phänomene. In **Shaping Robot Behavior Using Principles From Instrumental Conditioning** Saksida u. a. (1998) werden beispielsweise Mikroaktionen definiert, die nacheinander ausgeführt werden und unterschiedlich belohnt werden.

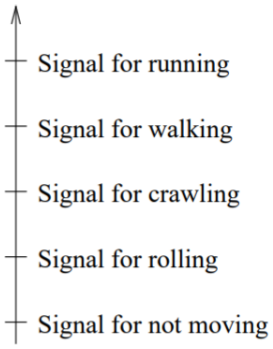


Abbildung 2.41.: Belohnungssignale für die Bewegung eines Roboters Randlov und Alstrom (1998)

Ein Beispiel für eine mehrstufige Architektur präsentiert Abblidung 2.41. Hier werden die erforderlichen Schritte gezeigt, damit ein Roboter das Laufen erlernen kann. In diesem Fall wird der Lernprozess als eine mehrstufige Architektur definiert. Dabei ändert sich das Feedbacksignal im Laufe der Zeit. Grundsätzlich werden die für die zu gerade lernende Stufe relevante Parameter besonders positiv eingestuft, während ein Rückfall in eine niedrigere Stufe bestraft wird. Wenn der Roboter beispielsweise während des Kriechens wieder ins Rollen verfällt so werden die Signale, die das Rollen bestärken als negatives Feedback zurückgegeben, obwohl diese am Anfang für das Erlernen des Rollens als positives Feedback dem Agenten zugeführt wurden.

Dorigo und Colombetti (1997) definieren in **Robot Shaping: An Experiment in Behavior Engineering** sogar ein komplett eigenes Forschungsfeld des Behavioral Engineering, dass sich mit dem Formen des Verhalten von Robotern beschäftigt.

Reward Shaping mit der Differenz von Potentialfunktionen

Reward Shaping ist ein Vorgang, der wie erwähnt nur schwer formalisiert werden kann. Die im vorherigen Abschnitt vorgestellten Konzepte sind lediglich Richtlinien nach denen eine Belohnungsfunktion umgesetzt werden kann, die allerdings keinen problemübergreifenden Erfolg garantieren können.

Ng u. a. (1999) haben in **Policy invariance under reward transformations: Theory and application to reward shaping** dennoch ein einfaches Framework auf Basis von Potentialfunktionen entworfen, wo das Bilden der Differenz solcher Potentialfunktionen $F(s, a, s') =$

$\gamma\phi(s') - \phi(s)$ zwischen Zustand und Folgezustand das Approximieren einer optimalen Value Function V_* garantieren soll. Eine Potentialfunktion ϕ ist hierbei eine Funktion die ein skalares Potential birgt. In diesem Fall ist das Potential die Belohnung, die durch eine modellierte Belohnungsfunktion erzeugt werden kann. Dieses Potential wird dabei als Skalarfeld dargestellt in Abhängigkeit des Zustands s. Dies ist mit der Auswertung einer Kostenfunktion in Kapitel 2.1 zu vergleichen, wie in Abbildung 2.42 dargestellt:

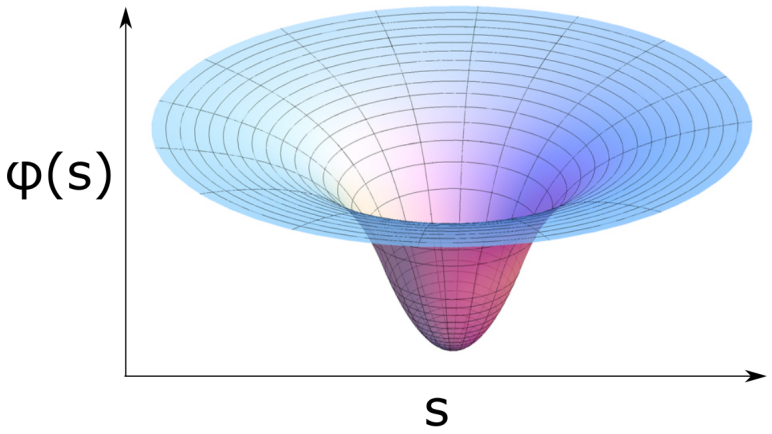


Abbildung 2.42.: Skalarfeld (AllenMcC unter CC BY-SA 3.0 Lizenz)

Dabei ist $F(s, a, s')$ die gesammte Belohnung, die von der Differenz der Potentialfunktionen $\phi(s)$ und $\phi(s')$ erzeugt wird, wobei s der aktuelle Zustand und s' der Folgezustand ist. Einen vollständigen Beweis, dass die potentialbasierte Belohnungsfunktion $F(s, a, s')$ eine optimale Value-Function approximiert kann im entsprechenden Paper nachgelesen werden. Hier wird lediglich anhand eines Beispiels die Modellierung einer potentialbasierten Belohnungsfunktion vorgestellt.

Ein Problem das durch das Verwenden einer potentialbasierten Belohnungsfunktion gelöst werden kann ist das Problem der positiven Belohnungsschleifen. Einer der berühmtesten Reinforcement Learning Benchmarks ist das Steuern eines Fahrrads zu einem bestimmten Ziel. Randlov und Alstrom (1998) haben sich als erstes mit dieser Aufgabe befasst. Dabei trat vor allem das Problem der positiven Belohnungsschleifen auf.

Als eine positive Belohnungsschleife wird ein Verhalten charakterisiert, welches dem Agenten positive Belohnungen bzw. positives Feedback zuführt, das jedoch nicht zielführend ist. In

Randlovs und Altroms Fall war ein Belohnungssignal die Distanz zwischen Fahrrad und Ziel. Da der Agent für das Rückwärtsfahren nicht bestraft wurde drehte sich das Fahrrad im Kreis, da es dadurch immer wieder kleine Schritte nach vorne machen konnte. **Allgemeiner, wenn es eine Zustandssequenz gibt, die der Agent zyklenmäßig durchqueren kann und eine positive Belohnung ergibt, so befindet sich der Agent in einer positiven Belohnungsschleife.**

Wenn s_t der aktuelle Zustand und $F(s_t, a, s_{t+1})$ die Belohnung des aktuellen Zustands zum Folgezustand ist. Wenn dann für eine Zustandschleife: $F(s_1, a_1, s_2) + ... + F(s_{t-1}, a_{t-1}, s_t) + F(s_t, a_t, s_1) \geq 0$, so befindet sich der Agent in einer positiven Belohnungsschleife. Wenn für diesen Fall eine Differenz von Potentialen der Form $F(s, a, s') = \phi(s') - \phi(s)$ definiert wird und $F(s_1, a_1, s_2) + ... + F(s_{t-1}, a_{t-1}, s_t) + F(s_t, a_t, s_1) = 0$, dann ist die positive Belohnungsschleife somit aufgehoben.

Bezogen auf den Anwendungsfall bedeutet dies, dass die Belohnungsfunktion so geformt werden muss, dass die Summe der in der Zustandsschleife erhaltenen Belohungen $= 0$ ergibt. Wird die Belohnungsfunktion als eine Differenz von Potentialen zwischen dem aktuellen Zustand und dem Folgezustand definiert, so reicht es eine Funktion ϕ zu definieren, die bei einem Rückwärtsfahren des Fahrrads für den Folgenzustand s' eine negative entgengesetzte Belohnung liefert, so dass $F(s, a, s') = \phi(s') - \phi(s) = 0$ wird.

Abschließende Worte

Mit Ende diese Kapitels sind alle für diese Arbeit relevante Grundlagen erläutert worden. In Kapitel 2.2.1 wurden zunächst die Grundkonzepte des Reinforcement Learnings erläutert. Der Zusammenhang zwischen Agent und Umgebung bis hin zur Definition und Bestimmung einer Policy. In Kapitel 2.2.2 wurden anhand der dynamischen Programmierung erste Methoden präsentiert, um Value Functions zu approximieren und aus diesen neue Policies zu bestimmen. Diese Methoden waren die Policy Evaluation und das Policy Improvement. Im folgenden Kapitel 2.2.3 wurde eine konkrete Implementierung der Policy Evaluation und Policy Improvement Schritte in der Form von Temporal Difference Learning und Q-Learning eingeführt. Dabei wurde mit Q-Learning ein sogenannter Off-Policy Algorithmus definiert dessen Policy Evaluation und Policy Improvement Schritte unabhängig von einander geschehen. Kapitel 2.2.4 hat versucht Value Functions in einer realisierbaren Form zu implementieren. Dabei wurden neuronale Netze als Funktionsapproximatoren für Value Functions vorgestellt. Im letzten Kapitel der Reinforcement Learning Grundlagen 2.2.5 wurden direkt parametrisierte

Policies vorgestellt und die Anpassung der Parameter solcher Policies anhand des Stochastic Policy Gradient.

In Kapitel 2.3.1 und 2.3.2 wurde der Deterministic Policy Gradient vorgestellt und wie dieser aus kontinuierlichen Zustands- und Aktionsräumen lernen kann. Dabei werden die Parameter einer parametrisierten Policy, um den Gradienten einer Action-Value Function angepasst. Der DDPG-Algorithmus ist dabei eine Implementierung des Deterministic Policy Gradient, wo ein Off-Policy Actor-Critic vorgestellt wurde, der die Theorie des Deterministic Policy Gradient implementiert. Damit die Konvergenz von DDPG gewährleistet werden kann wurden zwei zusätzliche Komponenten in Form eines Experience Replay Buffers und Target-Networks eingeführt.

Letztendlich präsentierte Kapitel 2.4 das Konzept des Reward Shapings, um Belohnungsfunktionen zu modellieren. Dabei wurden verschiedene Ansätze präsentiert, um anhand des Shapings eine möglichst passende Belohnungsfunktion zu modellieren. Anschließend wurde ein konkretes Framework zur Umsetzung einer Belohnungsfunktion in Form von der Differenz von Potentialfunktionen präsentiert. In der Theorie ist so eine Approximation einer optimalen Value Function möglich.

Das folgende Kapitel beschäftigt sich mit der Umsetzung des DDPG-Algorithmus und dessen Integrierung in den Flugzeugsimulator. Dabei werden die konkrete Umsetzung in Tensorflow und die Kommunikation mit dem Flugzeugsimulator über UDP Pakete erläutert. Darüber hinaus wird die Modellierung einer Belohnungsfunktion behandelt, die das Verhalten des Flugzeuges innerhalb dessen Umgebung formen soll.

3. Versuchsaufbau

Im diesem Kapitel wird die Implementierung des Flugzeugautopiloten erläutert. Dabei wird zunächst die Architektur eines herkömmlichen Autopilots betrachtet und wie diese auf eine Reinforcement Learning Architektur übertragen werden kann. Nach Vorstellung der verwendeten Software und Hardware wird die konkrete Architektur zur Implementierung des Reinforcement Learning basierten Flugzeugautopiloten erläutert. Dabei werden einzelne Komponenten wie die Kommunikation mit dem Flugzeugsimulator, die Implementierung des DDPG-Algorithmus in Tensorflow und die Modellierung der Belohnungsfunktion behandelt.

3.1. Vorbereitende Schritte

3.1.1. Konzept eines Flugzeugautopiloten

Autopilot mit Regelkreis

Flugzeugautopiloten werden in der Regel als Regelungssysteme umgesetzt. Dabei kann ein Regelkreis die Funktionsweise eines Flugzeugautopiloten beschreiben:

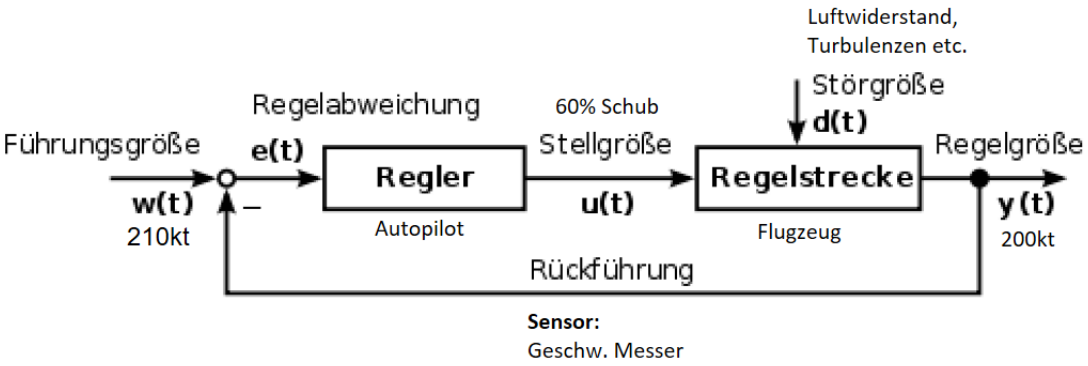


Abbildung 3.1.: Flugzeugautopilot anhand eines Regelkreises (Eigene Erstellung)

Abbildung 3.1 veranschaulicht einen Regelkreis für die Regulierung der Geschwindigkeit eines Flugzeugs:

1. Eine **Führungsgröße** $w(t)$ auch Sollwert gennant, wird als Vorgabe in den Regelkreis gegeben.
2. Der Regelkreis bekommt die akutelle Geschwindigkeit des Sensors und bildet aus dieser und dem Sollwert eine Differenz. Diese Differenz ist die **Regelabweichung** $e(t)$.
3. Anhand dieses Fehlers ermittelt der **Regler** in diesem Fall der Autopilot eine **Stellgröße** $u(t)$, die das Flugzeug steuern soll. Für das Bestimmen der Stellgröße ist ein Modell notwendig, das die Flugdynamik des Flugzeugs beschreibt.
4. Die bestimmte Stellgröße wird zum Steuern der **Regelstrecke** verwendet, die in diesem Fall das Flugzeug ist. Die Regelstrecke kann durch externe Faktoren auch **Störgrößen** $d(t)$ gennant beinflusst werden.
5. Dieser Prozess wiederholt sich ständig, um die **Regelgröße** $y(t)$ so nah wie möglich an der Führungsgröße zu halten. Mit anderen Worten soll die Regelabweichung $e(t)$ minimiert werden.

Regelkreis mit Reinforcement Learning

Ein Regelkreis für den DDPG-Algorithmus kann in ähnlicher Weise definiert werden. Die äquivalenten Komponenten des herkömmlichen Regelkreises sind in Klammern dargestellt:

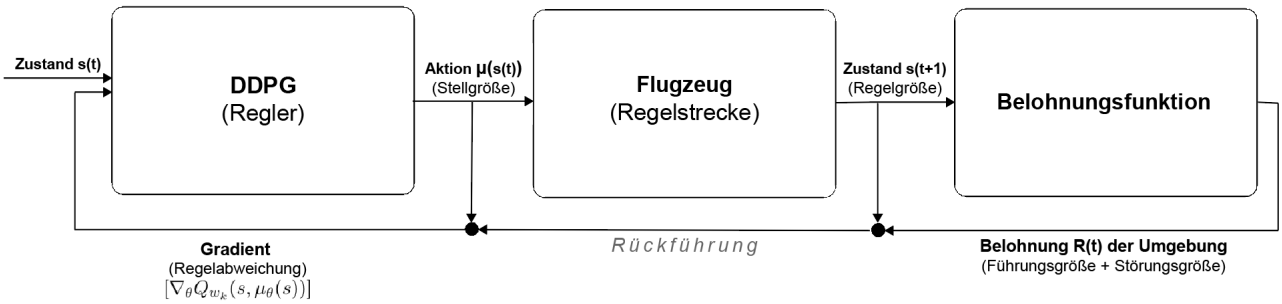


Abbildung 3.2.: Reinforcement Learning basierter Flugregler (Eigene Erstellung)

1. Der **DDPG-Algorithmus** dient in diesem Fall als Regler und bestimmt die Stellgröße zur Steuerung des Flugzeugs.
2. Die Umgebung des Flugzeugs liefert einen **Zustandsvektor**, der **Sensordaten** aus der Umgebung beinhaltet und beschreibt die Regelgröße, die durch Anwenden der Stellgröße resultiert. Zum Beispiel kann dieser Zustandsvektor die aktuelle Geschwindigkeit des Flugzeugs enthalten.

3. Eine **Belohnungsfunktion** erzeugt ein Feedback als Resultat einer getätigten Aktion a_t in einem bestimmten Zustand s_t. Dieses Feedback dient als Führungsgröße, die zusammen mit der Regelgröße und Stellgröße eine Regelabweichung bestimmen mit der, der DDPG-Algorithmus angepasst wird. Im Fall des DDPG-Algorithmus sind die Regelabweichungen die Gradienten für den Actor und den Critic die anhand dieser 3 Komponenten bestimmt werden.

4. Die Modellierung der Störgröße erfolg durch die Modellierung der Belohnungsfunktion, die die Bedingungen der Umgebung berücksichtigt.

Grundsätzlich unterscheidet sich der Reinforcement Learning basierte Autopilot vom herkömmlichen Autopilot mit Regelkreis in der Tatsache, dass dieser durch gesammelte Erfahrung unabhängig von der Flugdynamik des Flugzeuges und der Dynamik der Umgebung ein Regelungssystem approximiert. Das Bestimmen der korrekten Stellgrößen hängt bei einem herkömmlichen Autopilot von der korrekten Modellierung der Flugdynamik und der Umgebungsdynamik ab. Für jedes Fluggerät muss also im schlechtesten Fall eine eigene Flugdynamik bestimmt werden. Der Reinforcement Learning basierte Autopilot kann hingegen die Flugdynamik des Fluggerätes und die Dynamik der Umgebung vernachlässigen, da dieser ausschließlich aus Erfahrung lernt. Durch die Modellierung einer Belohnungsfunktion ist es also möglich ein bestimmtes Steuerungsziel unabhängig von der Flugdynamik des Fluggerätes und der Umgebungsdynamik zu definieren.

Theorethisch wäre es für einen RL-Autopiloten möglich mit gleichbleibenden Zustands- und Aktionsvektoren ausschließlich anhand der gesammelten Erfahrung unterschiedliche Fluggeräte mit dem selben Modell zu steuern. Nichtsdestotrotz, wird in folgenden Kapiteln festgestellt, dass die Modellierung einer Belohnungsfunktion, um ein bestimmtes Ziel zu erreichen keine triviale Angelegenheit ist und die Generalisierung eines Flugmodells nur in bestimmten Fällen funktioniert.

3.1.2. Frameworks und Tools

Verwendete Software

Im Folgenden wird die verwendete Software zur Implementierung des Flugzeugautopiloten vorgestellt. Dabei wird auch ein kurzer Überblick darüber gegeben welche Rolle, die einzelnen Softwarekomponenten in der Implementierung spielen:

- **Ubuntu 16.04 und Limetech Unraid:** Der Flugzeugautopilot wurde unter Ubuntu 16.04 umgesetzt. Dabei wurde Unraid als Virtualisierungssoftware verwendet, die es

ermöglicht virtuelle Instanzen des Betriebssystems zu erzeugen, die direkten Zugriff auf die Hardware des Rechners haben. Somit ist es möglich gewesen zwei parallel laufende Trainingsvorgänge zur realisieren.

- **Google Tensorflow:** Zur Implementierung des DDPG-Algorithmus wurde das Tensorflow Framework verwendet. Tensorflow bietet ein in Python programmierbares Framework für maschinelles Lernen an. Die jeweiligen neuronalen Netze bestehen aus einzelnen modularisierten Komponenten, so dass sich die programmiertechnische Umsetzung eines neuronalen Netzes in Tensorflow stark an die Theorie anlehnt.

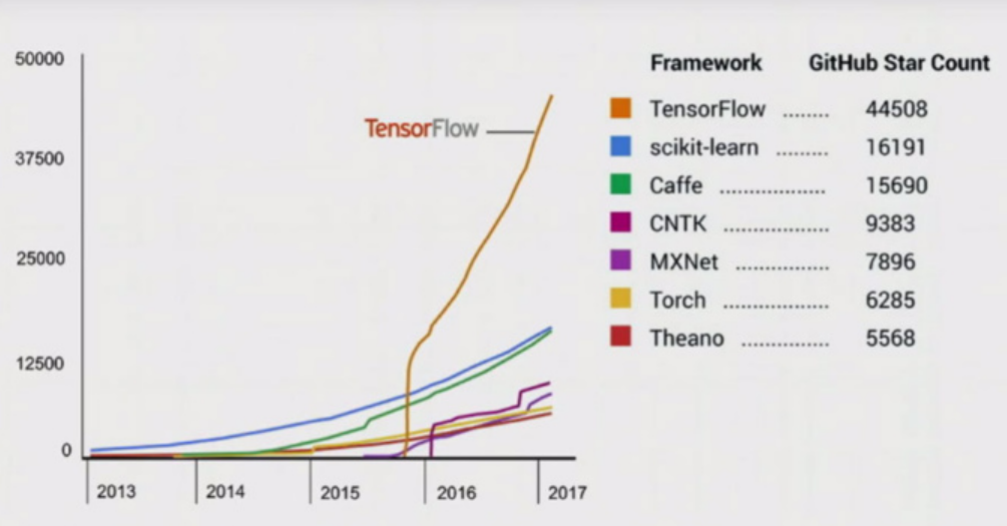


Abbildung 3.3.: Beliebtheit des Tensorflow Frameworks

Konkret können dabei Layer als einzelne Komponenten mit unterschiedlich bestimmbaren Eigenschaften zu einem neuronalen Netz zusammengefügt werden. Darüber hinaus bietet Tensorflow ein Framework zum Trainieren von neuronalen Netzen. Dabei sind beispielsweise die Verwendung von unterschiedlichen Kostenfunktionen und Parameteranpassungsverfahren möglich, die wiederum unterschiedliche Modalitäten des Backpropagation-Algorithmus wie Stochastic Gradient Descent oder Adam anwenden können.

Tensorflow sticht durch gut dokumentierte und ein relativ einfach zu verwendendes Framework heraus. Zudem genießt Tensorflow unter allen ML-Frameworks die größte Popularität, weshalb beispielsweise viele Reinforcement Learning Architekturen von der Community bereits implementiert und getestet worden sind.

- **CUDA Toolkit:** Tensorflow verfügt über die Möglichkeit Nvidias CUDA-Framework zu verwenden. Dieses Framework verwendet, die in den Nvidia Grafikkarten in großer Anzahl enthaltenen CUDA-Kerne, um hoch parallelisierbare Berechnungen ein vielfaches schneller als ein herkömmlicher Prozessor zu realisieren. Zusätzlich zu Tensorflow muss das NVIDIA CUDA Toolkit separat installiert werden. Folgende Versionen und Zusatzpakete sind zum Einsatzt gekommen:
  - **CUDA Toolkit 8.0 GA1:** Die grundlegende CUDA Library, die es ermöglicht mit CUDA-Kernen Operationen durchzuführen.
  - **CuDNN Library Version 6.0:** Eine spezialisierte Library für CUDA, die grundlegende Operationen wie Faltungen, Normalisierungen und Anwenden von verschiedenen Aktivierungsfunktionen im CUDA-Framework implementiert.
- **Flugsimulator:** Als Flugsimulator wurde X-Plane 11 verwendet. Dieser ist unter den Flugsimulatoren im Consumerbereich, der Simulator mit der anspruchsvollsten und detailliertesten Physikmodellierung.

Abbildung 3.4.: X-Plane 11

X-Plane 11 verwendet die sogenannte **Blade Element Simulation**, die ein Fluggerät in dessen einzelne Teile zerlegt und von diesen Einzelteilen aus simuliert. Andere Flugsimulatoren berechnen sogenannte Stabilitätsderivativen, die für eine getätigte Aktion lediglich eine entsprechende Gegenreaktion simulieren. Die Blade Element Simulation berechnet hingegen für jedes Einzelteil eines Fluggerätes eine eigene Physik und addiert diese für das Gesamtsystem zusammen. Die Blade Element Simulation ist in der Lage ausschließlich durch die geometrische Form und der Masse eines Flugzeugs eine eigene Flugdynamik für dieses zu bestimmen.

X-Plane 11 bietet zudem eine offene Netzwerkschnitstelle mit der Daten über UDP gesendet und empfangen werden können. So ist das Auslesen mehrerer Parameter und das Steuern des Flugzeuges über das Netzwerk möglich.

Darüber hinaus kamen einzelne Softwarepakete zum Einsatz, die jetzt vereinzelt präsentiert werden:

- **Tensorboard:** Im Tensorflow-Paket enthalten. Software zur Visualisierung der verschiedener Metriken des neuronalen Netzes.
- **xdotool:** Tool zum Ausführen von Makros. Für das Zurücksetzen des Flugzeugs und das Lösen der Parkbremse notwendig.

- **numpy:** Python Framework zur vereinfachten Manipulation von Datenstrukturen. Speziell multidimensionale Datenstrukturen.
- **socket:** Eine Python native Socket-Library, die Netzwerkkommunikation über Sockets ermöglicht.
- **tflearn**: Ein Tensorflow übergeordnetes Framework welches die Schnitstellen von Tensorflow kapselt.

Verwendete Hardware

Zum Trainieren der neuronalen Netze wurde ein System mit folgenden Spezifikationen verwendet:

- **CPU**: Intel Core i7 5820k 6 Kerne, 12 Threads
- **Arbeitsspeicher:** 64 Gigabyte
- **GPU 1:** Nvidia Geforce GTX 1080
- **GPU 2:** Nvidia Gerforce GTX 970

Jede Grafikkarte wurde in einer virtuellen Ubuntu Instanz mit jeweils 32 Gigabyte Arbeitspeicher und 6 Prozessorkernen verwendet, so dass diese parallel trainieren konnten. Grundsätzlich wurden beide Grafikkarten maximal zu 30% ausgelastet. Das direkte Training im Simulator erzeugt recht niedriege Datenmengen, die für die Grafikkarten leicht zu verarbeiten sind. Zudem arbeitet der DDPG-Algorithmus ausschließlich mit numerischen Daten. In der Regel ist die Verarbeitung von Bildern wie es bei der Anwendung von Convolutional Networks der Fall ist eine GPU intensivere Aufgabe, die vergleichsweise viel Grafikspeicher und Rechenleistung erfordert.

3.2. Implementierung des Flugzeugautopiloten

Der Flugzeugautopilot besteht aus drei hauptsächlichen Komponenten, die miteinander agieren. Abbildung 3.5 veranschaulicht die 3 Hauptbestandteile des Flugzeugautopiloten:

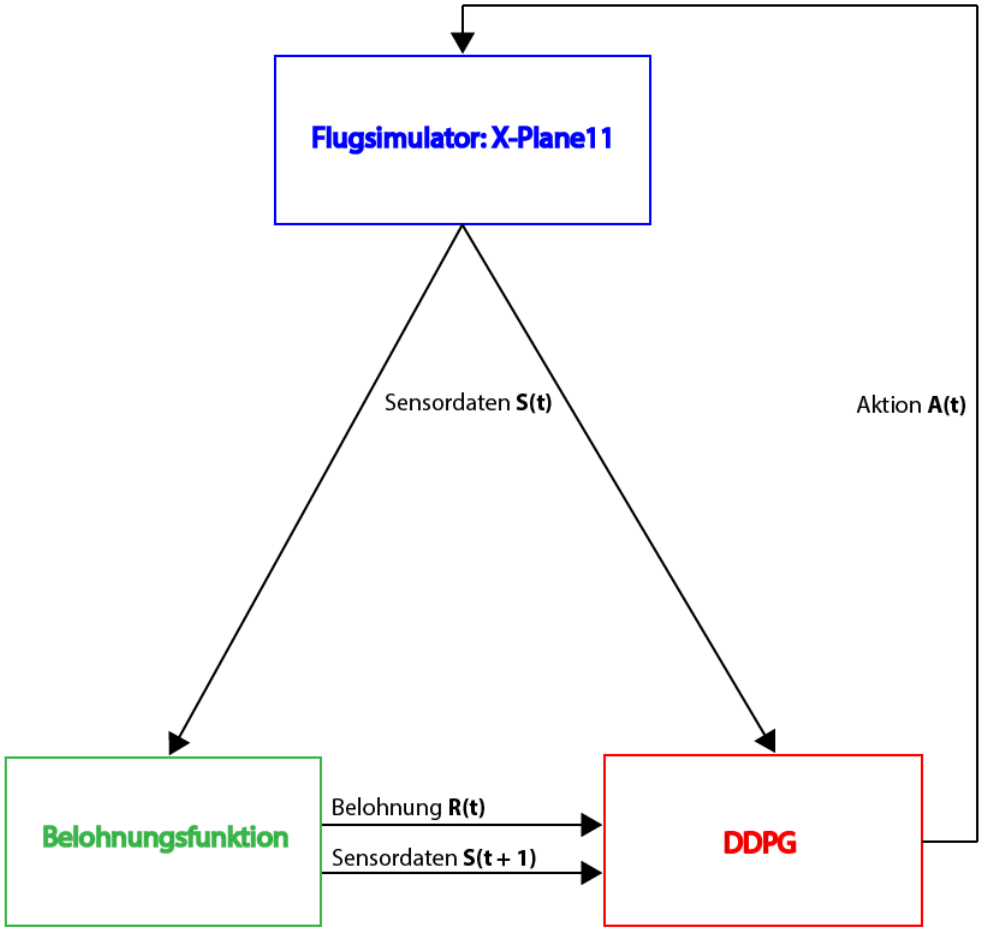


Abbildung 3.5.: Architektur des Autopiloten (Eigene Erstellung)

1. Der **Flugsimulator** sendet Sensordaten in Form von UDP-Paketen über das Netzwerk. Die Sensordaten werden von der Belohnungsfunktion und dem DDPG-Algorithmus empfangen

2. Die **Belohnungsfunktion** verwendet die empfangenen Sensordaten, um die Belohnung zu modellieren. Je nachdem in welchem Zustand sich das Flugzeug befindet liefern verschiedene Sensordaten unterschiedliche Belohnungen R_t. Zudem liefert die Belohnungsfunktion neue Sensordaten aus der Umgebung, die vom DDPG-Algorithmus als Folgezustand S_{t+1} verarbeitet werden.

3. Der **DDPG-Algorithmus** trainiert die Actor-Critic Architektur anhand der erhaltenen Belohnung R_t und des Folgezustands S_{t+1}, die von der Belohnungsfunktion empfangen wurden. Der Actor liefert für jeden Zeitschritt einen Aktionsvektor A_t, der Steuerwerte für das Flugzeug im Flugsimulator enthält und das Flugzeug somit steuert.

Im Folgenden wird jede einzelne Komponente näher erläutert. Zunächst wird die Kommunikation mit dem Flugsimulator näher erläutert, der über UDP-Packete den Aktionsvektor des DDPG-Algorithmus empfängt und die Sensordaten an die anderen Komponenten sendet. Danach wird die Implementierung des DDPG-Algorithmus in Tensorflow Schritt für Schritt anhand des Algorithmus in Kapitel 2.3.2 veranschaulicht. Als letztes wird die Modellierung der Belohnungsfunktion erläutert, die aus den Sensordaten des Flugzeugs Belohnungssignale an den DDPG-Algorithmus sendet.

3.2.1. Kommunikation mit dem Flugzeugsimulator

Austausch von Daten

Das Auslesen von Sensordaten und das Steuern des Flugzeugs sind über das Netzwerk durch das Senden von speziell kodierten UDP-Paketen möglich.

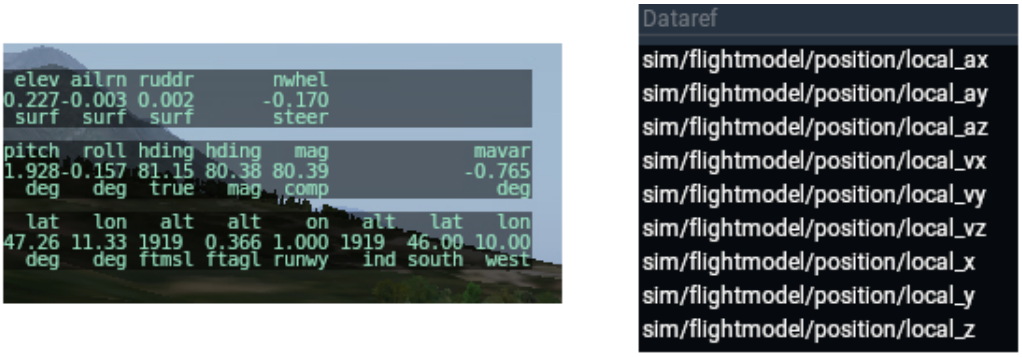


Abbildung 3.6.: **Data Elements**(links) aus der Data Input & Output Tabelle und **Data Refs**(rechts) (Eigene Erstellung)

X-Plane bietet unterschiedliche Nachrichtentypen, die unterschiedliche Informationen über den Flugsimulator liefern. Insgesamt gibt es 17 unterschiedliche Nachrichtentypen. In dieser Arbeit werden nur 2 dieser Nachrichtentypen verwendet, die in Abbildung 3.6 dargestellt sind:

1. **Data Elements:** Enthalten Daten der Data Input & Output Tabelle. Beinhalten alle im Flugzeug verfügbaren Sensordaten.

2. **Data Refs:** Beschreiben Simulatorvariablen. Erlauben Modifikation bestimmter Parameter des Simulators.

Die Data Element Pakete ermöglichen den Zugriff auf die Data Elements der Data Input & Output Tabelle. Jedes Data Element beinhaltet Sensordaten aus verschiedenen Kategorien. So gibt es beispielsweise ein Data Element für den Motorschub oder ein Data Element für die Steuerflächen des Flugzeugs.

Die Data Ref Pakete ermöglichen die Modifizierung von Variablen, die den Flugsimulator beinflussen. Beispielsweise können so Parameter der Flugphysik oder die globale Position des Flugzeugs (siehe Abbildung 3.6) modifiziert werden.

Data Element Paket

Ein Data Element Paket welches Parameter aus der Data Input & Output Tabelle überträgt besteht aus mindestens 41 Bytes, wenn ein einzelnes Data Element übertragen wird. Die genaue Zuteilung der 41 Bytes wird in Abbildung 3.7 beschrieben:

Abbildung 3.7.: Aufbau eines Data Element Pakets (Eigene Erstellung)

- **Byte 0 - 4:** Die ersten 5 Bytes bestehen aus dem Header, der den Nachrichtentypen identifiziert. Dabei ist das 5. Byte ein internes Byte, das nicht gesetzt wird.
- **Byte 5 - 8:** Die nächsten 4 Bytes beinhalten die Nummer des Data Element für welches die nachfolgenden Nutzdaten bzw. Sensordaten empfangen werden sollen.
- **Byte 9 - 40:** Die restlichen 32 Byte sind 8 Single-Precision Floating Point Werte, die aus jeweils 4 Byte bestehen. Dies sind die jeweilegen Nutzdaten des aktuellen Data Elements. Jedes Data Element kann maximal 8 Floating Point Werte besitzen.

Werden beispielsweise mehrere Data Elements angefordert, so werden diese zu einem einzelnen Paket zusammengefasst. Dabei wird der Header nur einmal am Anfang gesendet. Die Data Element Nummer und die Nutzdaten werden für jedes Data Element sukzessive an das Ende des Pakets angefügt, wie in Abbildung 3.8 dargestellt.

Abbildung 3.8.: Paket für mehrere Data Elements

Data Ref Paket

Data Ref Pakete besitzen im Vergleich zu Data Element Pakete immer eine feste Größe von 509 Bytes. Abbildung 3.9 stellt den Aufbau eines Data Ref Paket dar:

Abbildung 3.9.: Aufbau eines Data Ref Pakets

- **Byte 0 - 4:** Wiederum bestehen die ersten 5 Bytes aus dem Header. Das 5. Byte wird nicht gesetzt.
- **Byte 5 - 8:** Die nächsten 4 Bytes beinhalten in diesem Fall den Wert, auf den die Data Ref Variable gesetzt werden soll
- **Byte 9 - 509:** Die restlichen Bytes bestehen aus dem Pfad der zur Data Ref Variable führt. Nicht benutzte Bytes müssen auf 509 Bytes aufgefüllt werden.

Data Ref Pakete können nicht konkateniert werden. Für jede Data Ref Variable muss ein separates Paket gesendet werden.

Implementierung der Server Client Struktur

UDP-Pakete werden über die Python Socket Standardbibliothek empfangen und versandt. Dabei bietet X-Plane 11 einen UDP-Server auf dem separate Ports zum Senden und Empfangen von Daten konfiguriert werden können. Dies hat den Vorteil, dass Daten gleichzeitig gesendet und empfangen werden können, was für die Implementierung des Flugzeugsimulators von hoher Wichtigkeit ist. Die Ports aus Sicht des UDP-Servers sind folgendermaßen konfiguriert:

- **Sendeport:** Daten vom UDP-Server zum Client → 50001
- **Empfangsport:** Daten vom Client zum UDP-Server → 49001

Abbildung 3.10 veranschaulicht die Kommunikation zwischen Client und UDP-Server. Der Client teilt sich in Sender und Empfänger auf. Dabei ist der Empfänger für das Empfangen der Pakete und die Paketdekodierung zuständig, während der Sender die Data Element Pakete und Data Ref Pakete zusammenstellt und auf den Empfangsport des UDP-Servers sendet:

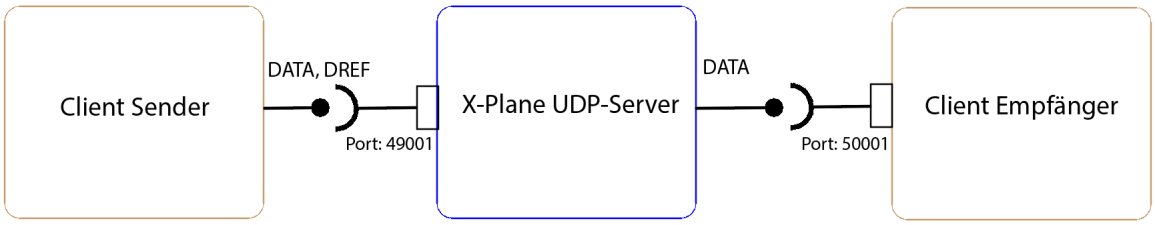


Abbildung 3.10.: Kommunikation zwischen UDP-Server und Client (Eigene Erstellung)

Der Sender und Empfänger sind in separaten Klassen implementiert, da diese sich von der Funktionalität grundsätzlich unterscheiden. Im Folgenden wird die Implementierung beider Komponenten anhand deren Klassendiagramme erläutert.

Sender

Der Sender besteht aus zwei Klassen. Dabei kapselt die Klasse **DataPacket** die Zusamenstellung eines Datenpakets und die Klasse **PackageSender** kümmert sich, um das Senden des aufgebauten DataPacket:

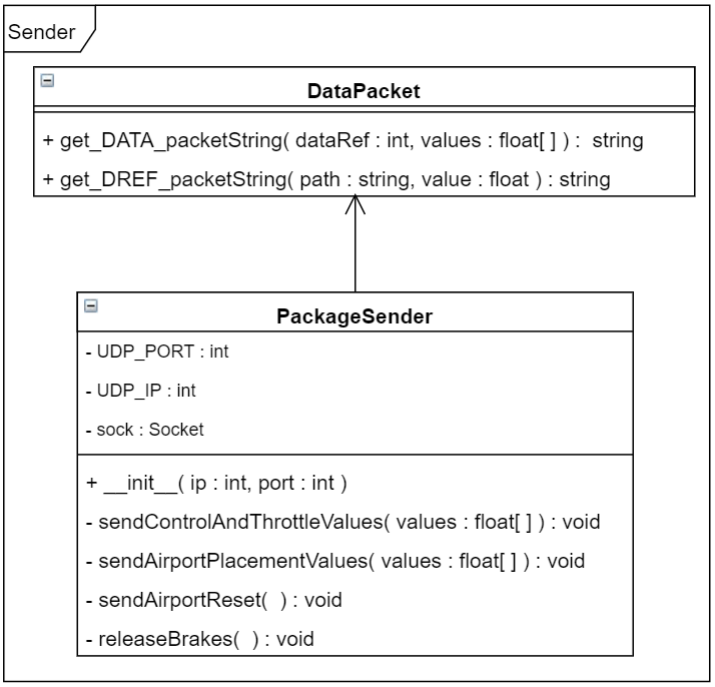


Abbildung 3.11.: Klassendiagramm des Senders (Eigene Erstellung)

- **DataPacket**
  - **get_DATA_packetString (dataElement, values) :** Bildet ein Data Element Paket für die gewünschte Data Element Nummer und die übergebenen Nutzdaten. Die Nutzdaten werden als Floating Point Werte übergeben. Das Konvertieren in die 4-Byte Repräsentation wird über die Python Bibliotheksfunktion **Struct** realisiert. Rückgabewert ist ein Vector mit dem Format eines Data Element Pakets.
  - **get_DREF_packetString (path, value) :** Bildet ein Data Ref Paket zur Modifizierung einer bestimmten Data Ref Variable des Simulators. Der Vorgang der Kodierung erfolgt wie oben beschrieben. Rückgabewert ist ein Vector mit dem Format eines Data Ref Pakets.
- **PackageSender**
  - **sendConrolAndThrottleValues (values) :** Sendet für zwei Data Elements zwei Data Element Pakete aus. Das erste Data Element bestimmt die Steurung des Flugzeugs, wobei das zweite Data Element die Regelung des Motorschubs ermöglicht.

- **sendAirportPlacementValues (values) :** Sendet drei Data Ref Pakete aus, um das Flugzeug mit jeweils X, Y, Z Werten in der Umgebung zu positionieren.
- **sendAirportReset ( ), releaseBrakes ( ) :** Dies sind zwei Hilfsfunktionen, die jeweils das Drücken einer Taste simulieren. Es ist in X-Plane 11 nicht möglich, das Flugzeug anhand von UDP-Paketen in den Anfangszustand zurückzuversetzen. Allerdings, kann dies auf einfache Weise durch das Drücken der Taste **r** erreicht werden. **Xdotool** simuliert dabei die Betätigung der Taste.

Beispiel

Folgendes Beispiel für die Funktion **sendControlAndThrottleValues** veranschaulicht die Verwendung der Klassen innerhalb des Senders, um ein Data Element Paket zu erstellen und dieses über das Netzwerk zu senden:

```
3       packetControl = DataPacket()
4       packetThrottle = DataPacket()
5
6       controlValues = (values[0], values[1], -999, -999, -999, -999, -999, -999)
7       throttleValues = (values[2], values[2], -999, -999, -999, -999, -999, -999)
8
9       packetControlString = packetControl.get_DATA_packetString(8, controlValues)
10      packetThrottleString = packetThrottle.get_DATA_packetString(25, throttleValues)
11
12      self.sock.sendto(packetControlString, (self.UDP_IP, self.UDP_PORT))
13      self.sock.sendto(packetThrottleString, (self.UDP_IP, self.UDP_PORT))
```

Abbildung 3.12.: Senden von Steuer- und Schubwerten

- **Zeilen 3 - 4:** Es wird jeweils ein Paket für ein Data Element erstellt.
- **Zeilen 6 - 7:** Die Nutzdatenvektoren werden definiert. In diesem Fall bekommt die Funktion **sendControlAndThrottleValues** einen Vektor mit zu übertragenden Werten als Parameter. Alle nicht verwendete Nutzdaten werden auf -999 gesetzt, damit der Flugsimulator diese ignoriert.
- **Zeilen 9 - 10:** Das eigentliche Data Element Paket wird mit der Funktion **get_DATA_packetString** aufgebaut. Als Parameter werden die Data Element Nummer und die Nutzdaten übergeben.
- **Zeilen 10 - 11:** Im letzten Schritt wird der Empfangsport des UDP-Servers verwendet, um den als Data Element Paket kodierten Vektor and den UDP-Server des Flugsimulators zu schicken.

Empfänger

Der Empfänger besteht ebenfalls aus zwei Klassen. Die Klasse **DataElement** kapselt alle nötigen Data Elements in einer Enum. Dabei muss die Reihenfolge der Data Elements in der Enum mit der Reihenfolge der Data Elements im Simulator übereinstimmen. Die **PackageReceiver** Klasse kümmert sich, um das Dekodieren der Pakete, die vom UDP-Server des Flugsimulators empfangen werden:

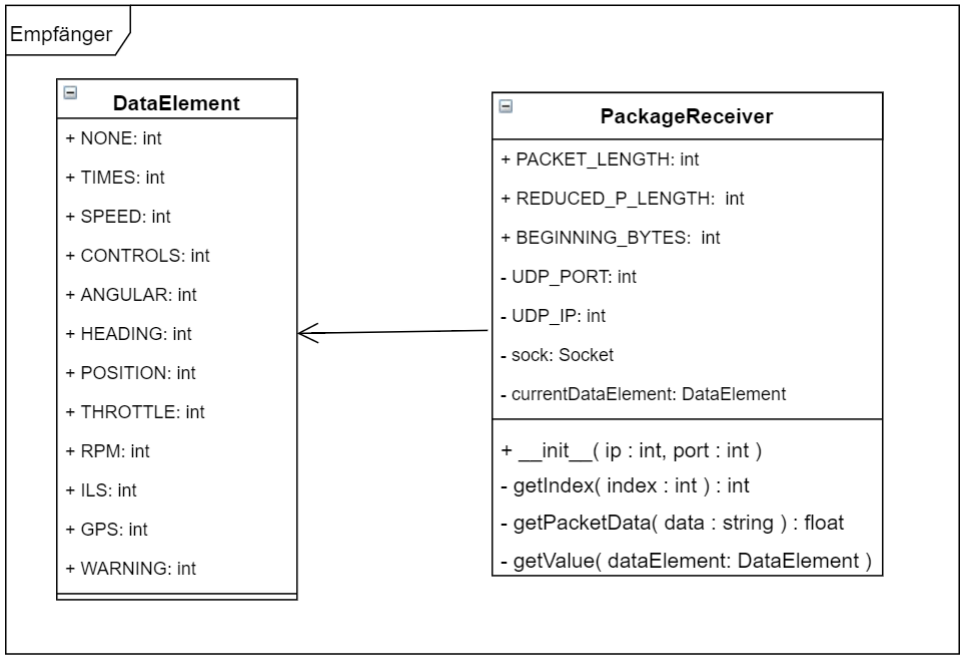


Abbildung 3.13.: Klassendiagramme für Empfänger

- **PackageReceiver**
  - **getIndex (index) :** Diese Funktion berechnet den echten Index, um die richtigen Nutzdaten für das gewünschte Data Element aus dem Nutzdatenvektor zu entehmen. Der Index wird folgendermaßen berechnet:

$$
\begin{aligned}
&= (currentDataElement * Paketlaenge\ ohne\ Header) \\
&+ Headerlaenge \\
&- Nutzdatenlaenge \\
&+ index
\end{aligned}
\tag{3.1}
$$

Angenommen es wird mehr als ein DataElement über das Netzwerk gesendet, dann muss innerhalb des empfangenen UDP-Pakets der korrekte Index ermittelt werden, um auf das gewünschte DataElement zugreifen zu können:

1. Als erstes wird die übergebene Data Element Nummer mit der Paketlänge ohne Header multipliziert, so kann innerhalb des UDP-Pakets auf das richtige Data Element zugegriffen werden.
2. Im nächsten Schritt wird die Headerlänge addiert, die nur einmal am Anfang auftritt.
3. Um an den Anfang der Nutzdaten zu gelangen wird die gesammte Nutzdatenlänge subtrahiert.
4. Letztendlich liefert der übergebene Index mit Werten von 0 - 32, das jeweilige Byte aus den Nutzdaten.

– **getPacketData (data) :** Hier werden die jeweiligen 4-Byte Floating Point Werte der Nutzdaten in dezimal dargestellte Floating Point Werte umgewandelt. Dafür wird das **Struct** Paket der Python Standardbibliothek verwendet.

– **getValue (dataElement) :** Empfängt ein UDP-Paket des UDP-Servers und dekodiert dieses anhand der Hilfsfunktionen **getIndex** und **getPacketData**. Anschließend wird ein Vektor mit 8 in dezimal dargestellten Floating Point Werten zurückgegeben, die den Nutzdaten des gewünschten Data Element entsprechen.

Beispiel

Auch für die Anwendung des PackageReceivers wird ein Beispiel präsentiert:

- **Zeilen 2 - 3:** Ein Vektor mit 8 Feldern wird zu null initialisiert und das gewünschte Data Element wird für die Bestimmung des Index global als Klassenvariable gesetzt.
- **Zeile 5:** Über den Sendeport wird ein Paket vom UDP-Server empfangen.
- **Zeilen 7 - 14:** Die jeweiligen 4-Byte Floating Point Werte werden aus dem UDP-Paket entnommen und zu dezimal Floating Point Werte dekodiert. Insgesamt werden immer 8 Floating Point Werte dekodiert und im Vektor gespeichert.

```
def getValue(self, dataRef):
    useData = [0] * 8
    self.currentDataElement = dataElement

    data, addr = self.sock.recvfrom(1024)

    useData[0] = self.getPacketData(data[self.getIndex(0):self.getIndex(4)])
    useData[1] = self.getPacketData(data[self.getIndex(4):self.getIndex(8)])
    useData[2] = self.getPacketData(data[self.getIndex(8):self.getIndex(12)])
    useData[3] = self.getPacketData(data[self.getIndex(12):self.getIndex(16)])
    useData[4] = self.getPacketData(data[self.getIndex(16):self.getIndex(20)])
    useData[5] = self.getPacketData(data[self.getIndex(20):self.getIndex(24)])
    useData[6] = self.getPacketData(data[self.getIndex(24):self.getIndex(28)])
    useData[7] = self.getPacketData(data[self.getIndex(28):self.getIndex(32)])

    return (
    useData[0],
    useData[1],
    useData[2],
    useData[3],
    useData[4],
    useData[5],
    useData[6],
    useData[7])
```

Abbildung 3.14.: Empfangen eines UDP-Pakets

Das Dekodieren der Nutzdaten eines UDP-Pakets geschieht also in der folgenden Reihenfolge:

$$\textbf{Sensordaten} \leftarrow \textbf{getValue} \leftarrow \textbf{getPacketData} \leftarrow \textbf{getIndex} \tag{3.2}$$

Dabei wird zuerst der Index bestimmt anhand dessen in einem Paket auf die Nutzdaten verschiedener Data Elements zugegriffen werden kann. Die daraus resultierenden binär kodierten 4-Byte Daten werden durch getPacketData in dezimale Floating Point Werte umgewandelt. Die Funktion getValue liefert schließlich die insgesamt 8 Sensorwerte für ein Data Element.

3.2.2. DDPG Implementierung

Der DDPG-Algorithmus empfängt Sensorwerte des Flugsimulators, die zu einem Zustandsvektor s_t zusammengefasst werden. Anhand dieses Zustandvektors und der empfangenen Belohnung R_t aus der Belohnungsfunktion bestimmt der DDPG-Algorithmus eine neue Aktion a_t. Die Grundlagen des DDPG-Algorithmus wurden bereits in Kapitel 2.3.2 erläutert. Im Folgenden wird die Implementierung der einzelnen Komponenten des DDPG-Algorithmus erläutert. Der Actor und Critic, die Target Networks und der Experience Replay Buffer.

Abbildung 3.15 zeigt das konkrete Zusammenspiel zwischen dem DDPG-Algorithmus und den restlichen Komponenten des Flugzeugautopiloten:

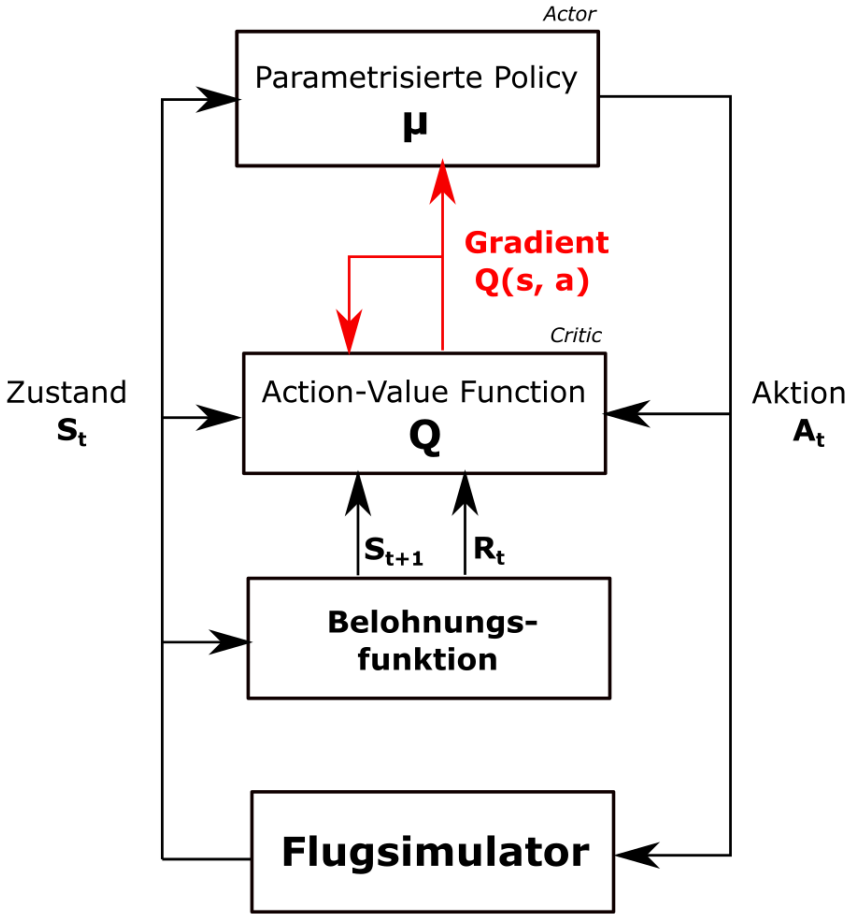


Abbildung 3.15.: DDPG-Algorithmus im Zusammenspiel mit den anderen Komponenten (Eigene Erstellung)

Dabei wird ein Einblick in die Architektur des DDPG-Algorithmus gewährt:

- Der **Actor** bekommt lediglich den aktuellen Zustand s_t der Umgebung des Flugzeugsimulators. Daraus resultiert eine Aktion a_t, die das Flugzeug steuert und für die Approximation der Action-Value Function im Critic verwendet wird. Der Gradient des Actors wird durch den Deterministic Policy Gradient bestimmt, der den Action-Value $Q(s, a)$ zur Bestimmung des Deterministic Policy Gradient benötigt.
- Der **Critic** approximiert anhand der Belohnung R_t aus der Belohnungsfunktion und dem Folgezustand S_{t+1} den Action-Value $Q(s, a)$. Daraus werden zum einen die Gradienten zur Aktualisierung der Parameter des Critics ermittelt und zum anderen der Gradient des Action-Values Q(s, a) mit dessen Aktionen an den Actor übergeben. Dazu wird für ein Mini-Batch aus dem Experience Replay Buffer die entsprechende Kostenfunktion des Critics ausgewertet und gemittelt.

Im Folgenden werden der Aufbau und der oben beschriebene Trainingsvorgang zur Aktualisierung der Parameter für beide Komponenten erläutert. Dabei wird zuerst der Aufbau der einzelnen Komponenten des DDPG-Algorithmus erläutert und anschließend wird die Implementierung des DDPG-Algorithmus Schritt für Schritt anhand des Algorithmus in Kapitel 2.3.2 aufgeführt.

Implementierung eines neuronalen Netzes in Tensorflow

Sowohl der Actor als auch der Critic werden durch neuronale Netze approximiert. Im Folgenden wird der Aufbau beider Komponenten und deren Umsetzung in Tensorflow erläutert. Um die Umsetzung beider Komponenten besser zu verstehen werden bestimmte Tensorflow-Funktionen zum Aufbau von neuronalen Netzen im Voraus erklärt:

- **tensorflow.Graph:** Ein Berechnungsvorgang wird in Tensorflow in Form eines Graphen repräsentiert. Dabei ist jeder Knoten entweder ein Datum (z.B. ein Tensor) oder eine Operation, die ausgeführt werden soll.
- **tensorflow.Variable (<initial-value>):** Tensorflow-Variablen können mehrere Tensoren speichern und repräsentieren trainierbare Einheiten eines Tensorflow-Graphen. Konkret kann eine Variable einen Tensor speichern, der anschließend ein Bestandteil eines neuronalen Netzes wird. Beispielswiese kann eine Variable die gesammte Gewichtsmatrix für eine definierte Anzahl an Neurons eines neuronalen Netzes enthalten.
- **tensorflow.placeholder (dtype, shape=None):** Ein Placeholder ermöglicht das Einführen externer Daten in einen Tensorflow-Graphen. Dabei muss der Datentyp und die

Dimensionalität der Daten im Voraus angegeben werden. Placeholder können direkt wie Variablen eingesetzt werden, die später zur Laufzeit mit externen Daten gefüllt werden.

- **tensorflow.random_uniform (shape, minval, maxval):** Instanziiert einen Tensor der Form **shape** und initialisiert diesen mit den Werten einer Normalverteilung im Wertebereich zwischen **minval** und **maxval**.
- **tensorflow.nn.<relu, tanh> (Tensor):** Wendet eine ausgewählte Aktivierungsfunktion auf einen bestimmten Tensor an. In diesem Fall kann der Tensor die Gewichte mehrerer Neurons eines Layers enthalten auf die eine bestimmte Aktivierungsfunktion angewendet werden soll.

Anhand dieser Tensorflowfunktionen kann nun ein einfaches neuronales Netz aufgebaut werden. Um den Aufbau des Actor und Critics besser verstehen zu können wird zunächst ein einfaches neuronales Netz mit einer verdeckten Schicht und einer Ausgabeschicht aufgebaut:

```
2  def create_simple_nn(input_dim, output_dim):
3      layer1_size = 100
4
5      input = tf.placeholder("float", input_dim)
6
7      W1 = tf.Variable(tf.random_uniform([input_dim, layer1_size],-1e-3, 1e-3))
8      b1 = tf.Variable(tf.random_uniform([layer1_size],-1e-3, 1e-3))
9      W2 = tf.Variable(tf.random_uniform([layer1_size, output_dim],-1e-3,1e-3))
10     b2 = tf.Variable(tf.random_uniform([output_dim],-1e-3,1e-3))
11
12     layer1 = tf.nn.relu(tf.matmul(input, W1) + b1)
13     output = tf.tanh(tf.matmul(layer1,W2) + b2)
```

Abbildung 3.16.: Tensorflowcode für neuronales Netz

- **Zeile 5:** Es wird zunächst ein Placeholder instanziiert, der es ermöglicht dem neuronalen Netz externe Daten zuzuführen.
- **Zeile 7 - 10:** Die einzelnen Bestandteile des neuronalen Netzes werden in Variablen gespeichert. Dabei werden die jeweiligen Parameter (Gewichte und Bias) der verdeckten Schicht und der Ausgabeschicht instanziiert und initialisiert.
- **Zeile 12:** Hier werden die einzelnen Bestandteile des neuronalen Netzes zusammengesetzt. Abbildung 3.17 veranschaulicht die Matrizenoperationen zur Erzeugung der verdeckten Schicht $layer1$:

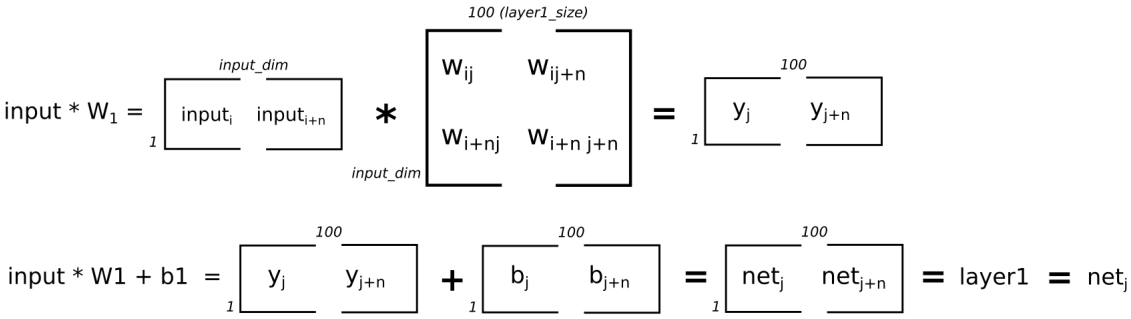


Abbildung 3.17.: Matrizenoperationen: Gewichtindex: i, Neuronindex: j (Eigene Erstellung)

In diesem Fall besteht die verdeckte Schicht aus 100 Neurons. Zunächst wird die Gewichtsmatrix W_1 mit der Eingangsmatrix $input$ multipliziert. Die Gewichtsmatrix ist eine $input\_dim \times layer\_size$ Matrix, die ein Gewicht für jeden Eingang der jeweligen Neurons beinhaltet. Anschließend werden die Biase b_1 für alle Neuronen addiert. Hieraus ergibt sich die verdeckte Neuronenschicht $layer1$.

- **Zeile 13:** Die Ausgabeschicht ergibt sich dann durch die Multiplikation der verdeckten Schicht $layer1$ mit der Gewichtsmatrix W_2 der Ausgabeschicht und der anschließenden Addition der Biase b_2. Sowohl auf die verdeckte Schicht als auch auf die Ausgabeschicht wird eine Aktivierungsfunktion angewendet. Im Fall der verdeckten Schicht eine **Relu** Aktivierungsfunktion und im Fall der Ausgabeschicht eine **Tangens Hyperbolicus** Aktivierungsfunktion.

Abbildung 3.18 zeigt den schematischen Aufbau des Beispielnetzes. Hier sind jeweils die resultierenden Schichten aus dem Anwenden der Matrixoperationen und der Aktivierungsfunktionen dargestellt. Dabei wird eine Schicht, die alle Parameter der vorherigen Schicht berücksichtig als Fully Connected bezeichnet. Diese Darstellungsform wird auch für die nachfolgende Darstellung des Aufbaus der Actor und Critic Netzwerke verwendet.

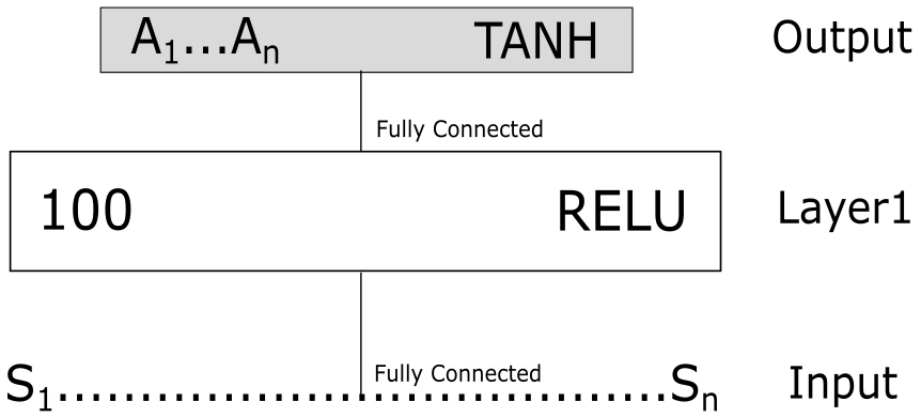


Abbildung 3.18.: Aufbau des Beispielnetzes (Eigene Erstellung)

Implementierung des Actor und Critic

Mit dem Wissen wie ein einfaches neuronales Netz in Tensorflow umgesetzt wird kann nun der etwas komplexere Aufbau der Actor und Critic Netze erläutert werden.

Actor

Abbildung 3.19 zeigt die Tensorfow-Implementierung und einen schematischen Aufbau des Actors:

```
2   def create_actor_network(self,state_dim,action_dim):
3       layer1_size = LAYER1_SIZE
4       layer2_size = LAYER2_SIZE
5
6       state_input = tf.placeholder("float",[None,state_dim])
7
8       W1 = self.variable([state_dim,layer1_size],state_dim)
9       b1 = self.variable([layer1_size],state_dim)
10      W2 = self.variable([layer1_size,layer2_size],layer1_size)
11      b2 = self.variable([layer2_size],layer1_size)
12      W3 = tf.Variable(tf.random_uniform([layer2_size,action_dim],-3e-3,3e-3))
13      b3 = tf.Variable(tf.random_uniform([action_dim],-3e-3,3e-3))
14
15      layer1 = tf.nn.relu(tf.matmul(state_input,W1) + b1)
16      layer2 = tf.nn.relu(tf.matmul(layer1,W2) + b2)
17      action_output = tf.tanh(tf.matmul(layer2,W3) + b3
```

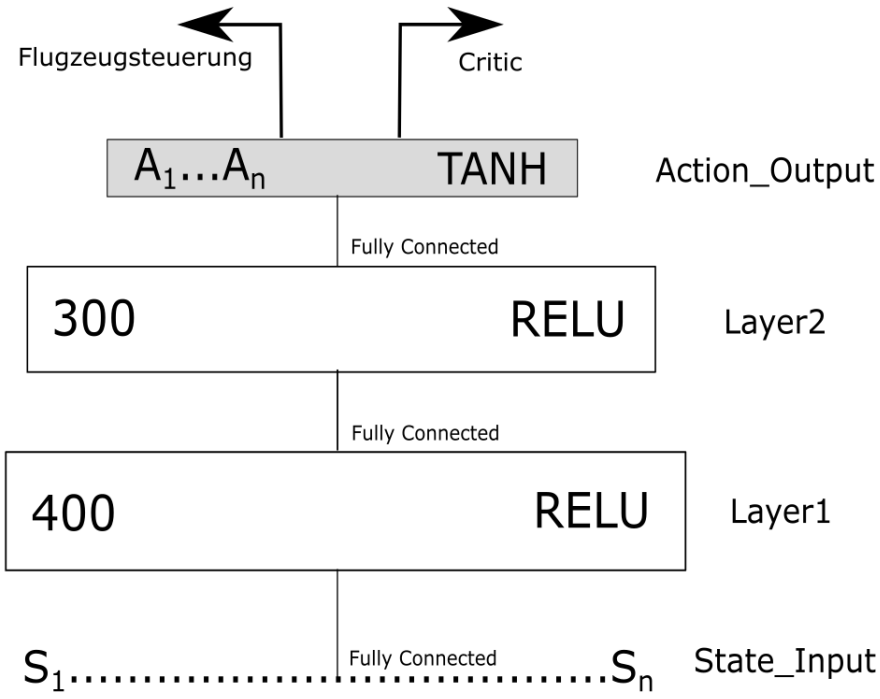


Abbildung 3.19.: Aufbau des Actors

Der Actor ist mit dem vorher präsentierten neuronalen Netz vergleichbar. Der Actor hat allerdings zwei verdeckte Schichten, die aus jeweils 400 und 300 Neurons bestehen. Die Ausgabeschicht (Action_Output) besteht in diesem Fall aus der Anzahl an Aktionen, mit denen der Flugzeugautopilot das Flugzeug steuern soll. Auf beide verdeckte Schichten wird wiederum eine Relu Aktivierungsfunktion angewendet, wohingegen auf die Ausgabeschicht eine Tangens Hyperbolicus Aktivierungsfunktion angewendet wird, um die Ausgänge auf einen Wertebereich zwischen -1 und 1 zu bringen. Der Action_Output dient der direkten Steuerung des Flugzeugs und wird als Input für das neuronale Netz des Critics verwendet.

Critic

Die Umsetzung des Critics ist im Vergleich zu der des Actors etwas komplexer. Der Critic approximiert die Action-Value Function Q anhand des erhaltenen Aktionsvektors des Actors, wie in Abbildung 3.20 zu sehen ist:

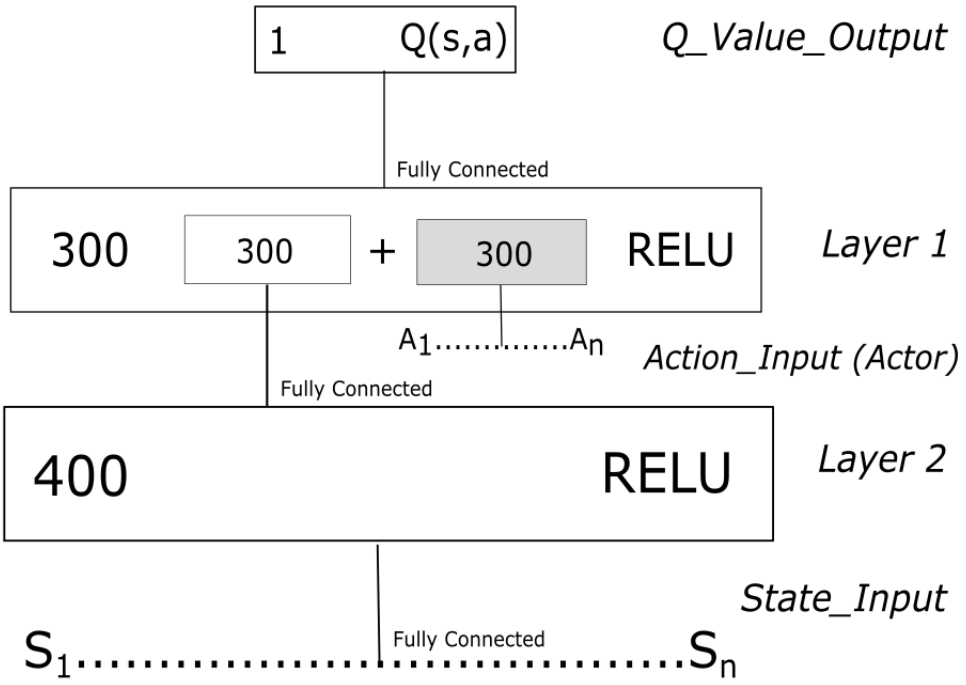


Abbildung 3.20.: Aufbau des Critics

Experimentell wurde von Lillicrap u. a. (2015) festgestellt, dass die Einführung des Aktionsvektors des Actors in der 2. verdeckten Schicht ein stabileres Lernen ermöglicht. Um die Ergebnisse der 1. verdeckten Schicht mit dem Action_Input zu vereinen werden die Neuroneinheiten der **1. verdeckten Schicht** und des **Actor_Inputs** auf eine gemeinsame Anzahl gebracht und anschließend addiert. Wie für den Actor bestehen beide verdeckte Schichten aus jeweils 400 und 300 Neurons auf die ebenfalls eine Relu Aktivierungsfunktion angewendet wird. Die Ausgabe des Critics ist der approximierte Action-Value $Q(s, a)$, der zur Berechnung des Gradienten für den Actor und den Critic verwendet wird.

Target Networks

Die Target Networks sind Kopien der Actor und Critic Netze. Dabei sollen die Gewichte der Target Networks nur verzögert aktualisiert werden. Wie in Kapitel 2.3.2 beschrieben bleiben so die Gewichte für einen bestimmten Zeitraum relativ stabil, wodurch stabilere Gradienten ermittelt werden können.

```
2  def create_target_network(self,state_dim,action_dim,net):
3          state_input = tf.placeholder("float",[None,state_dim])
4          ema = tf.train.ExponentialMovingAverage(decay=1-TAU)
5          target_update = ema.apply(net)
6          target_net = [ema.average(x) for x in net]
7
8          layer1 = tf.nn.relu(tf.matmul(state_input,target_net[0]) + target_net[1])
9          layer2 = tf.nn.relu(tf.matmul(layer1,target_net[2]) + target_net[3])
10         action_output = tf.tanh(tf.matmul(layer2,target_net[4]) + target_net[5])
11
12         return state_input,action_output,target_update,target_net
```

Abbildung 3.21.: Implementierung der Target-Netze

- **Zeile 4:** Das verzögerte Aktualisieren der Gewichte wird durch eine exponentielle Glättung durchgeführt. Eine Instanz des Tensorflow-Objekts **train.ExponentialMoving Average** wird erzeugt, die eine exponentielle Glättung der Form

$$\mu'_t = \tau \cdot \mu_t + (1 - \tau) \cdot \mu'_{t-1} \tag{3.3}$$

  anwendet. Dabei wird der neue Wert des Target Networks durch die Kombination aus den tatsächlichen Gewichten μ_t mit den verzögerten Gewichten μ'_{t-1} des vorherigen Schritts berechnet. Der Kombinationsfaktor wird in diesem Fall durch den Parameter τ bestimmt. Um eine besonders agressive Glättung bzw. ein zu schnelles Kopieren der Gewichte zu vermeiden sollte ein sehr kleines $\tau \sim 0$ gewählt werden.

- **Zeile 5:** Die Funktion **apply (Tensor)** des ExponentialMovingAverage-Objekts erzeugt Kopien der Variablen der echten Gewichte, die mit dem Parameter net übergeben werden. Dabei wird ein Funktionshandler zurückgeliefert, der die in Zeile 4 beschriebene Operation der exponentiellen Glättung für alle übergebenen Gewichte durchführt. Durch das Aufrufen dieses Funktionshandlers können die Target Networks nach jeder Iteration aktualisiert werden.

- **Zeile 6 - 10:** Hier wird das Target Network wie die Actor und Critic Netze aufgebaut. Dabei wird auf das übergebene Netz net ein Schritt der exponentiellen Glättung mit der Funktion **average(Tensor)** durchgeführt. Anhand einer for-each Schleife wird davor die exponentielle Glättung auf alle Variablen des übergebenen neuronalen Netzes angewendet.

Experience Replay Buffer

Die letzte Komponente des DDPG-Algorithmus ist der Experience Replay Buffer. Der Expe-

rience Replay Buffer speichert alte Transitionen der Form s_t, a_t, r_t, s_{t+1}. Diese Transitionen werden anschließend aus dem Experience Replay Buffer beliebig gesampled und anhand einer vordefinierten Mini-Batch Größe für das Training der Actor und Critic Netze verwendet.

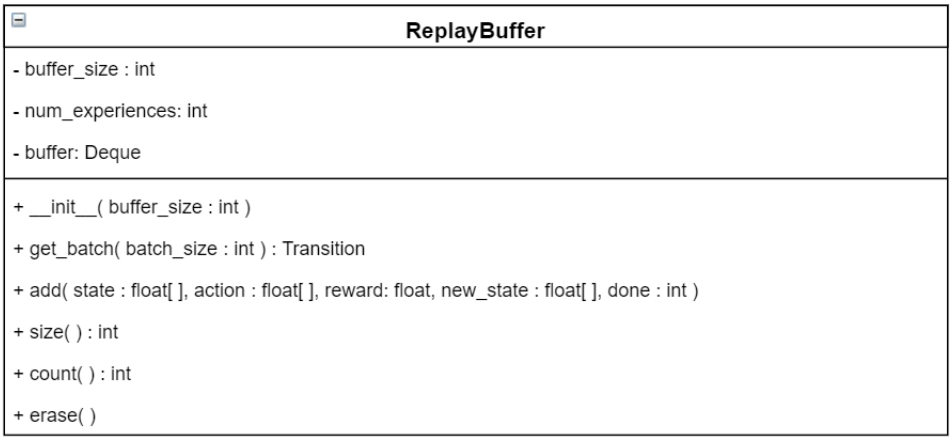


Abbildung 3.22.: Klassendiagramm des Experience Replay Buffer

Der Experience Replay Buffer wird als eine **Deque (Double-Ended Queue)** implementiert, die es ermöglicht Elemente am Anfang und am Ende hinzuzufügen und zu entfernen:

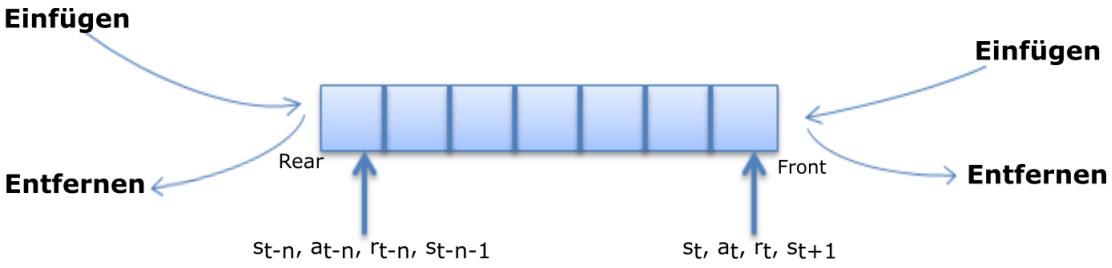


Abbildung 3.23.: Experience Replay Buffer als Deque (Eigene Erstellung)

Bei der Initialisierung des Buffers wird eine feste Größe **buffer_size** definiert. Der Experience Replay Buffer verhält sich wie eine Queue. Abbildung 3.24 zeigt das Hinzufügen einer neuen Transition wenn die Deque voll ist. Wenn die definierte Größe der Deque überschritten wird und eine neue Transition gespeichert werden soll, so wird die älteste Transition in der Deque entfernt und die neue Transition an den Anfang angehängt. Da die Deque als doppelte verkettete Liste implementiert ist muss lediglich die Referenz des vordersten Elements der Deque aktualisiert werden, um die neue Transition an den Anfang hinzuzufügen.

Abbildung 3.24.: Hinzufügen einer neuen Transition bei vollem Buffer (Eigene Erstellung)

Konkret kapselt die Funktion **add** das Hinzufügen einer neuen Transition, wie im Codeauschnitt in Abbildung 3.25 zu sehen ist. Das Entnehmen eines Mini-Batches für das Training wird durch die Python-Bibliotheksfunktion **random.sample (buffer, batch_size)** in der Funktion **get_batch** realisiert. Für einen bestimmten Buffer liefert diese Bibliotheksfunktion eine Liste an zufällig ausgewählter Transitionen aus dem Buffer zurück.

```
2       def get_batch(self, batch_size):
3           # Randomly sample batch_size examples
4           return random.sample(self.buffer, batch_size)
5
6       def add(self, state, action, reward, new_state, done):
7           experience = (state, action, reward, new_state, done)
8
9           if self.num_experiences < self.buffer_size:
10              self.buffer.append(experience)
11              self.num_experiences += 1
12          else:
13              self.buffer.popleft()
14              self.buffer.append(experience)
```

Abbildung 3.25.: Hinzufügen und Entnehmen von Transitionen

Trainingsvorgang anhand des DDPG-Algorithmus

Im Folgenden wird der Trainingsvorgang des Actor und Critic mit Hilfe des präsentierten DDPG-Algorithmus in Kapitel 2.3.2 erläutert. Dabei werden die Berechnung der Gradienten und die Aktualisierung der Parameter für beide Netze Schritt für Schritt dargestellt.

Die neuronalen Netze des Actor und Critics sind in Klassen gekapselt, die ein einfaches Verwenden und Trainieren der Netze ermöglichen:

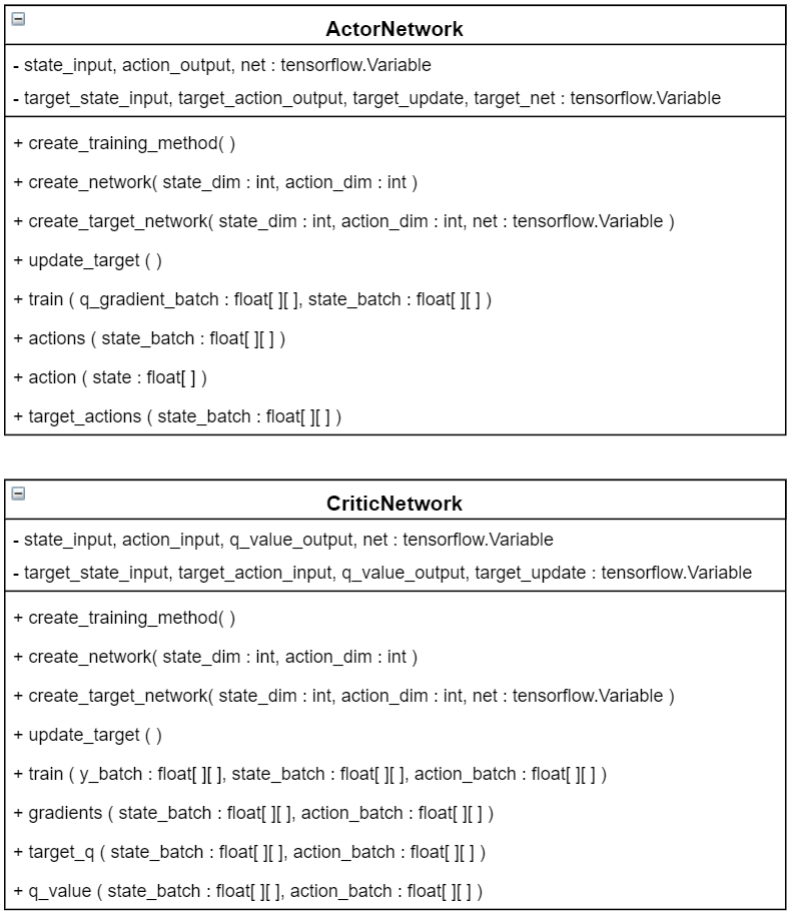


Abbildung 3.26.: Aufbau des Critics

Die **create_network** und **update_target** Funktionen wurden bereits beim Aufbau der Actor und Critic Netze erläutert. Bevor die einzelnen Schritte des DDPG-Algorithmus durchgegangen werden, wird die **create_training_method** Funktion näher analysiert. Diese definiert die Operationen zur Bestimmung des Gradienten und der Aktualisierung der Parameter der beiden neuronalen Netze.

Gradientenbestimmung und Parameteranpassung des Actors

Anhand folgender Funktionen wird der Gradient des Actors bestimmt und die Netzparameter anhand dieses Gradienten angepasst. Zu Erinnerung ergibt sich der Gradient des Actors, also

der Deterministic Policy Gradient durch die Ableitung der Action-Value Function des Critics mit den Parametern θ des Actors:

$$\begin{aligned}\nabla J(\theta) &= \nabla_\theta \, Q_w(s, \mu_\theta(s)) \\ &= \nabla_\theta \mu_\theta(s) \, \nabla_{\mu_\theta(s)} Q_w(s, \mu_\theta(s)) \quad \textbf{(Anwend. der Kettenregel)}\end{aligned} \tag{3.4}$$

Folgender Codeabschnitt in Abblidung 3.27 realisiert die Berechnung des Gradienten und die Aktualisierung der Parameter θ mit dem Adam Optimizer:

```
2  def create_training_method(self):
3      self.q_gradient_input = tf.placeholder("float", [None,self.action_dim])
4      self.parameters_gradients = tf.gradients(self.action_output, self.net, -self.q_gradient_input)
5      self.optimizer = tf.train.AdamOptimizer(LEARNING_RATE) \
6      .apply_gradients(zip(self.parameters_gradients, self.net))
```

Abbildung 3.27.: Gradientenberechung und Parameteranpassung

- **Zeile 3:** Ein Placeholder wird erstellt, um den Gradienten der Action-Value Function $\nabla_{\mu_\theta(s)} Q_w(s, \mu_\theta(s))$ mit dessen Aktionen $\mu_\theta(s)$ zwischenspeichern zu können. Dieser Gradient wird vom Critic geliefert und muss daher extern eingeführt werden.
- **Zeile 4:** Die Funktion **tf.gradients(ys, xs, grad_ys)** berechnet den Gradienten des Vektors ys mit den im Vektor xs enthaltenen Parametern. Dabei können die Gradienten mit dem Vektor grad_ys vorinitialisiert werden. Es kann also ein bereits berechneter Gradient mit dem zu berechnenden Gradienten kombiniert werden.

  Folgende Teile des Gradienten entsprechen den Parametern der Funktion **tensorflow.gradients**:

  - **ys:** Der Output des Actornetzes: $\mu_\theta(s)$.
  - **xs:** Die Parameter θ des Actors.
  - **grad_ys:** Gradient der Action-Value Function mit dessen Aktionen: $\nabla_{\mu_\theta(s)} Q_w(s, \mu_\theta(s))$.
- **Zeile 5 - 6:** Der berechnete Gradient wird auf die Parameter θ anhand des **Adam Optimizer (Adaptive Moments Optimizer)** (siehe Anhang) eine verbesserte Variante des Stochastic Gradient Descent angewendet. Die Funktion **apply_gradients** des Adam Optimizer wendet die im vorherigen Schritt berechneten Gradienten auf die einzelnen Gewichtsmatrizen des neuronalen Netzes, die in **self.net** gespeichert sind an.

Gradientenbestimmung und Parameteranpassung des Critics

Der Gradient des Critics ist der Gradient der Action-Value Function Q. Wie in Kapitel 2.3.1

erläutert ist der Gradient des Critics die Ableitung der Kostenfunktion der Action-Value Function mit dessen Parametern w:

$$J(w) = \mathbb{E}_\pi \left[(R_{t+1} + \gamma \hat{Q}(s_{t+1}, \mu(s_{t+1}), w) - \hat{Q}(s, \mu(s_t), w))^2 \right] \tag{3.5}$$

$$\nabla J(w) = \left[R_{t+1} + \gamma \hat{Q}(s_{t+1}, \mu(s_{t+1}), w) - \hat{Q}(s, \mu(s_t), w) \right] \nabla_w \hat{Q}(s, \mu(s_t), w) \tag{3.6}$$

Wie für den Actor werden nun auch für den Critic die Funktionen zur Bestimmung des Gradienten und Anpassung der Parameter betrachtet. Abbildung 3.28 zeigt die Implementierung in Tensorflow:

```
2  def create_training_method(self):
3      # Define training optimizer
4      self.y_input = tf.placeholder("float",[None,1])
5      #Behandeln von Ausreißern
6      weight_decay = tf.add_n([L2 * tf.nn.l2_loss(var) for var in self.net])
7      self.cost = tf.reduce_mean(tf.square(self.y_input - self.q_value_output)) + weight_decay
8      self.optimizer = tf.train.AdamOptimizer(LEARNING_RATE).minimize(self.cost)
9      self.action_gradients = tf.gradients(self.q_value_output,self.action_input)
```

Abbildung 3.28.: Gradientenberechung und Parameteranpassung

- **Zeile 4:** Es wird ein Placeholder instanziiert, um den aus dem Mini-Batch ermittelten Schätzwert einzuführen. Der Schätzwert ist im Kontext der Action-Value Function der sogenannte Target: $R_{t+1} + \gamma \hat{V}(S_{t+1}, w)$. Der Placeholder muss dabei eine Matrix sein, die alle berechneten Schätzwerte des Mini-Batches beinhaltet.
- **Zeile 6 - 7:** Ein wichtiger Optimierungsschritt, um einem Overfitting des Models vorzubeugen ist die Bestimmung eines zusätzlichen Regularisierungsterms, der zur Kostenfunktion addiert wird:

$$\frac{1}{N} \sum_i \left[(R_i + \gamma \hat{Q}(s_{i+1}, s_{i+1}, w) - \hat{Q}(s_i, a_i, w))^2 \right] + \lambda \sum_w w^2 \tag{3.7}$$

  Dieser Term wird als L2-Regularisierungsterm bezeichnet. Dabei wird die L2-Norm aller Gewichte des neuronalen Netzes des Critics bestimmt und aufsummiert. Durch das Addieren des Regularisierungsterms wird verhindert, dass Gewichte einen zu großen skalaren Wert erlangen. Sind die Gewichte groß so steigt auch die L2-Norm und somit auch der Wert der Kostenfuktion. Im Umkehrschluss bedeutet dies, dass die Anpassung der Gewichte in einen großen skalaren Wertebereich unterbunden wird. Der Parameter λ beinflusst dabei wie stark dabei die Bestrafung ausfällt.

Es ist wichtig einen Anstieg der skalaren Werte der Gewichte zu verhindern, da sonst das Lernen des Agenten beeinträchtigt wird. Dies liegt daran, dass sich die ermittelten Gradienten in der Regel in einem kleinen Wertebereich bewegen. Sind die Gewichte, die angepasst werden müssen zu groß, so haben die Gradienten zur Anpassung einen geringen Einfluss auf die Gewichte des neuronalen Netzes.

- **Zeile 8:** Wie beim Actor wird anhand des Adam Optimizer der Gradient der Kostenfunktion ermittelt. Dabei versucht die Funktion **minimize** des Adam Optimizer, die Parameter so anzupassen, dass die Kostenfunktion minimiert wird.
- **Zeile 9:** Hier wird der Gradient der Action-Value Function Q mit dessen Aktionen: $\nabla_{\mu_\theta(s)} Q_w(s, \mu_\theta(s))$ berechnet. Die Variable **action_gradients** wird dem Actor dann zur Verfügung gestellt, um den Deterministic Policy Gradient zu bestimmen, wie in der Beschreibung von Abbildung 3.27 erläutert wurde.

Training anhand des DDPG-Algorithmus

Im letzten Abschnitt wird die Implementierung des DDPG-Algorithmus, der in Kapitel 2.3.2 vorgestellt wurde präsentiert. Die Erstellung der neuronalen Netze und des Experience Replay Buffers wurden bereits behandelt, sowie die Funktionen zur Gradientenbestimmung und zur Parameteranpassung. Hier wird letztendlich das Zusammenspiel zwischen Actor, Critic und den restlichen Komponenten in Tensorflow dargestellt.

- Als erstes werden die neuronalen Netze und der Experience Replay Buffer instanziiert:

1 Initialisiere Critic $Q_w(s, a)$ und Actor $\mu_\theta(s)$ mit beliebigen Gewichten w und θ
2 Initialisiere Target-Networks Q' und μ' mit Gewichten $w' \leftarrow w, \theta' \leftarrow \theta$
3 Initialisiere Experience Replay Buffer R

```
# initialize neural networks
self.actor_network = ActorNetwork(self.sess,self.state_dim,self.action_dim)
self.critic_network = CriticNetwork(self.sess,self.state_dim,self.action_dim)

# initialize replay buffer
self.replay_buffer = ReplayBuffer(REPLAY_BUFFER_SIZE)
```

- Als nächstes wird der Zufallsprozess **N** initialisiert und der erste Zustand s_1 aus der Umgebung gesampled:

4 **für** *für jede Episode* **tue**
5 | Initialisiere N für Erforschung des Agenten
6 | Ersten Zustand s_1 samplen

```
# Initialize a random process the Ornstein-Uhlenbeck process for action exploration
self.exploration_noise = OUNoise(self.action_dim)

# Get state s_1
state = env.reset()
```

- Nun wird ein Auswertungsschritt des Actors ausgeführt, der dabei eine Aktion a_t liefert. Um die Belohnung für den Zustand s_t und Aktion a_t zu bekommen wird diese ausgeführt. Die Belohnungsfunktion liefert dabei den Folgezustand s_{t+1}, die Belohnung r_t und die Information darüber, ob die Episode zu Ende ist:

7 | **für** *jeden Zeitschritt* t *in Episode* **tue**
8 | | Wähle Aktion $a_t = \mu_\theta(s_t) + N_t$
9 | | Führe a_t aus und speichere Belohnung r_t und Folgezustand s_{t+1}

```
noise = self.exploration_noise.noise()
action = self.actor_network.action(state) + noise
next_state, reward, done = env.step(action)
```

- Nachdem eine komplette Transition (s_t, a_t, r_t, s_{t+1}) erfasst wurde kann diese nun in den Experience Replay Buffer gespeichert werden. In den nachfolgenden Schritten kann nun die Actor-Critic Architektur trainiert werden. Dazu muss das Experience Replay Buffer allerdings eine bestimmte Minimalgröße erreicht haben, um ein Mini-Batch mit Größe **N** an Transitionen samplen zu können.

10 | | Speichere Transition (s_t, a_t, r_t, s_{t+1}) in Replay Buffer R
11 | | Ein beliebiges Mini-Batch aus N Transitionen (s_i, a_i, r_i, s_{i+1}) aus R samplen

```
# Store transition (s_t,a_t,r_t,s_{t+1}) in replay buffer
self.replay_buffer.add(state, action, reward, next_state, done)

# Sample a random minibatch of N transitions from replay buffer
minibatch = self.replay_buffer.get_batch(BATCH_SIZE)
state_batch = np.asarray([data[0] for data in minibatch])
action_batch = np.asarray([data[1] for data in minibatch])
reward_batch = np.asarray([data[2] for data in minibatch])
next_state_batch = np.asarray([data[3] for data in minibatch])
done_batch = np.asarray([data[4] for data in minibatch])
```

- Für den Critic werden nun alle Targets: $R_{t+1} + \gamma \hat{Q}(S_{t+1}, w)$ des Mini-Batches berechnet. Dabei werden zunächst die Parameter der Action-Value Function in separate Arrays gespeichert. Anschließend werden in einer Schleife für die jeweiligen Transitionen die Targets ermittelt:

12 | | | Setze $Target = r_i + \gamma Q'_{w'}(s_{i+1}, \mu'_{\theta'}(s_{i+1}))$

```
# Calculate Target and Target_i
next_action_batch = self.actor_network.target_actions(next_state_batch)
q_value_batch = self.critic_network.target_q(next_state_batch,next_action_batch)

y_batch = []
for i in range(len(minibatch)):
    if done_batch[i]:
        y_batch.append(reward_batch[i])
    else :
        y_batch.append(reward_batch[i] + GAMMA * q_value_batch[i])
```

- Anhand der vorher erläuterten Funktionen in Abbildung 3.28 wird nun für den Critic der Gradient der Kostenfunktion bestimmt und die Parameter so angepasst, dass die Kostenfunktion minimiert wird:

13 Aktualisiere Critic durch Minimierung der Kostenfunktion:

$$J(w) = \frac{1}{N}\sum_i (Target_i - Q_w(s_i, a_i))^2$$

```
# Update critic by minimizing the loss L
predicted_q,_ = self.critic_network.train(y_batch,state_batch,action_batch)
```

- Für den Actor wird der Deterministic Policy Gradient bestimmt. Dabei wird der Gradient der Action-Value Function mit dessen Aktionen im Critic berechnet: $\nabla_{\mu_\theta(s)} Q_w(s, \mu_\theta(s))$ und dem Actor übergeben:

14 Aktualisiere Aktorpolicy anhand des gesampleten Deterministic Policy Gradient:

$$\nabla_\theta J \approx \frac{1}{N}\sum_i \nabla_\theta \mu_\theta(s_i) \quad \nabla_{\mu_\theta(s_i)} Q_w(s_i, \mu_\theta(s_i))$$

```
# Update the actor policy using the sampled gradient:
action_batch_for_gradients = self.actor_network.actions(state_batch)
q_gradient_batch = self.critic_network.gradients(state_batch,action_batch_for_gradients)

self.actor_network.train(q_gradient_batch,state_batch)
```

- Als letztes werden die Target-Netze anhand der Funktion **update_target**, die die Gewichte der Target-Networks durch Anwenden der exponentiellen Glättung aktualisiert:

15 Aktualisiere Parameter der Target Networks:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta'$$
$$w' \leftarrow \tau w + (1-\tau)w'$$

```
# Update the target networks
self.actor_network.update_target()
self.critic_network.update_target()
```

3.2.3. Modellierung und Umsetzung der Belohnungsfunktion

Das letzte Kapitel des Versuchsaufbaus beschäftigt sich mit der Modellierung und Umsetzung einer Belohnungsfunktion. In Kapitel 2.4 wurden bereits einige Konzepte vorgestellt, um eine Belohnungsfunktion zu modellieren. Diese Konzepte wurden auch für die Umsetzung der Belohnungsfunktion des Flugzeugautopiloten zur Hilfe genommen. Teilweise sind einige Modelleriungsentscheidungen ausschließlich aus ausgiebigem Testen getroffen worden, weshalb es nicht immer möglich ist ein Bezug zu den theoretischen Konzepten herzustellen.

Die Belohnungsfunktion kann als eine Black Box betrachtet werden, deren Implementierung je nach Anwendungsfall angepasst wird. Abblidung 3.29 zeigt die Eingänge und Ausgänge der Belohnungsfunktion:

Abbildung 3.29.: Black Box einer Belohnungsfunktion (Eigene Erstellung)

Die Belohnungsfunktion bekommt als Eingang den aktuellen Zustandsvektor S_t und gibt dabei eine skalare Belohnung R_t aus und einen Folgezustand S_{t+1} aus. Dabei ist die skalare

Belohnung R_t von der Implementierung der Belohnungsfunktion abhängig. In diesem Fall gibt es für jede Flugsituation eine eigene Implementierung der Belohnungsfunktion, die anhand des Zustandsvektors ein Belohnungssignal für den entsprechenden Fall formt.

Softwarearchitektur der Belohnungsfunktion

Die Belohnungsfunktion ist in der einfachsten Form ein Interface, dass für jeden Anwendungsfall konkret implementiert wird. In diesem Fall beschreibt das Interface **Environment** das Implementierungtemplate. Dabei wurde das Design des Interfaces größtenteils aus dem Interface **core.py** von **OpenAI Gym** entnommen. Abbildung 3.30 zeigt das Klassendiagramm für das Interface Environment:

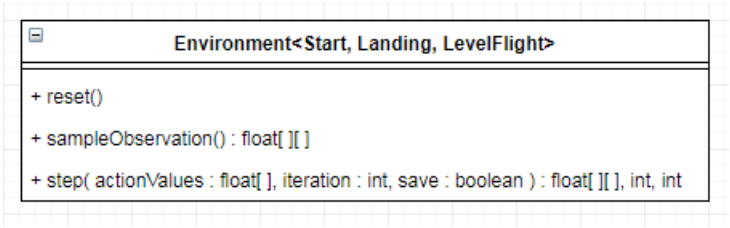


Abbildung 3.30.: Environment Interface (Eigene Erstellung)

- **reset( ):** Diese Funktion setzt die Umgebung auf den Anfangszustand zurück. Dies geschieht wenn eine neue Lernepisode angefangen wird. So wird zum Beispiel beim Start des Flugzeugs, das Flugzeug wieder an den Anfang der Landebahn gesetzt relevante Parameter wie die aktuelle Belohnung oder das Terminierungsflag zurückgesetzt.
- **sampleObservation( ):** Hier wird der Zustandsvektor aus der Umgebung gesampled. Dabei kann anhand des in Kapitel 3.2.1 beschriebenen UDP-Frameworks der PackageReceiver zum Abrufen aller Flugzeugsensoren verwendet werden, um anschließend anhand der Sensordaten den Zustandsvektor zusammenzusetzen.
- **step(actionValues, episode, save):** In dieser Funktion wird die ermittelte Aktion des Actors in **actionValues** an den Flugsimulator gesendet und ausgeführt. Daraufhin wird unmittelbar die Belohnung berechnet. Anschließend werden der neue Zustandsvektor, also der Folgezustand, die Belohnung und ein Terminierungsflag zurückgegeben.

  Zusätzlich helfen die Funktionsparameter **episode** und **save** zur Trainingssteuerung. Durch das Übergeben der Iterationsanzahl können die Constraints zur Berechnung der Belohnung angepasst werden. Außerhalb des Trainings ist es auch wünschenswert, dass

der Agent nicht zurückgesetzt wird, weshalb der Parameter save in diesem Fall eine Fallunterscheidung ermöglicht.

Implementierung der Belohnungsfunktionen für Start, Kreuzfahrtflug und Landung

Die jeweligen Implementierungen der Belohnungsfunktionen implementieren das Interface **Environment**. Darüber hinaus definieren folgende 3 Komponenten die wichtigsten Bestandteile der jeweiligen Belohnungsfunktionen:

1. Erfasste Parameter zur Erstellung des Zustandsvektors.
2. Constraints zur Einschränkung des Suchraumes und Verhaltensmodellierung des Agenten.
3. Formung der Belohnung anhand des erfassten Zustandsvektors und der formulierten Constraints.

(1) Zunächst müssen passende Parameter anhand des UDP-Frameworks erfasst werden. Dies wird in der Funktion **sampleObservation** realisiert. Diese liefert einen Vektor, der pro Zeitschritt die relevanten Sensordaten erfasst. Im Folgenden werden zunächst alle verwendeten Parameter erläutert, um bei den konkreten Implementierungen aus dem Parameterpool schöpfen zu können:

- **normedSpeed:** Die auf 1 genormte aktuelle Geschwindigkeit des Flugzeugs.
- **crsDrift:** Die Abweichung vom vorgegebenen Referenzkurs. Dabei wird der Absolutwert der Differenz zwischen dem Kurs der Mitellinie und dem aktuellen Kurs $abs(refHeading - heading)$.
- **rollDrift:** Die Abweichung von der zentralen Rolllage des Flugzeugs. Es wird der Absolutwert der Differenz zwischen der zentralen Rolllage (0°) und der aktuellen Rolllage des Flugzeugs ermittelt $abs(refRoll - roll)$.
- **deviation:** Die Abweichung von der Mittellinie der Landebahn. Die Berechnung wird im Abschnitt der Landung näher erläutert.
- **relBearing:** Absolute Kursabweichung zur gewünschten Mittellinie der Landebahn.
- **onRunway:** Zeigt an, ob sich das Flugzeug auf der Landebahn befindet.

- **noseWheel:** Lenkwinkel des vorderen Fahrwerks des Flugzeugs.
- **distance:** Auf 1 genormte Distanz zum Flughafen.
- **normedHeight:** Die auf 1 genormte aktuelle Höhe des Flugzeugs.
- **normedPitch:** Die auf 1 genormte aktuelle Neigung des Flugzeugs.
- **angularQ, angularP, angularR:** Neigungsrate, Rollrate und Gierrate des Flugzeugs.
- **throttle:** Die auf 1 genormte Schubreglerstellung.
- **rpm:** Die auf 1 genormte Drehzahl des Flugzeugmotors.

(2) Um das Lernen zu beschleunigen wird vorab der Suchraum des Agenten eingeschränkt. Zum Beispiel wird der Agent beim Start zurückgesetzt wenn dieser die Landebahn verlässt oder zu stark zur Seite giert. Grundsätzlich wird dabei ein zusätzliches Belohnungssignal ermittelt, dass das Betreten des verbotenen Bereiches bestraft. In diesem Sinne soll auch das Verhalten des Agenten geformt werden.

(3) Die Belohnung setzt sich aus einer Differenz zusammen. Dabei ist dies eine Differenz zwischen den Belohnungssignalen, die die Belohnung maximieren und minimieren:

$$\boldsymbol{Reward} = rewardSignalsToMaximize - rewardSignalsToMinimze$$

Beim Start des Flugzeugs ist beispielsweise die Geschwindigkeit ein Wert der maximiert werden sollte damit das Flugzeug abheben kann. Auf der anderen Seite sollte das Flugzeug aber auch möglichst im Zentrum der Landebahn bleiben, weshalb die Abweichung zum Zentrum als minimierendes Belohnungssignal interpretiert werden kann.

Im Folgenden wird die konkrete Umsetzung für die jeweligen Anwendungsfälle erläutert, wo diese 3 Aspekte jeweils behandelt werden.

Start

- **Zustandsvektor:** (normedSpeed, crsDrift, rollDrift, onRunway, noseWheel, distance, normedHeight, normedPitch, angularQ, angularP, angularR)

- **Constraints:** Die dem Agenten aufgesetzten Constraints sind in Abbildung 3.31 zu sehen. Der Agent wird zurückgesetzt wenn dieser die Landebahn verlässt. Dies geschieht wenn der Parameter **onRunway** == 0. In diesem Fall wird ein zusätzliches Belohnungssignal **leaveRunwayReward** eingeführt. Dabei wird der Agent nicht sofort zurückgesetzt sondern erst nach einer festlegbaren Anzahl an Sekunden. So sammelt der Agent genügend negative Erfahrungen, um dieses Verhalten zu unterbinden. Wird der Agent sofort zurückgesetzt so verfällt dieser leichter in ein lokales Optimum. Dieser Zusammenhang wird in der Versuchsdurchführung näher erläutert.

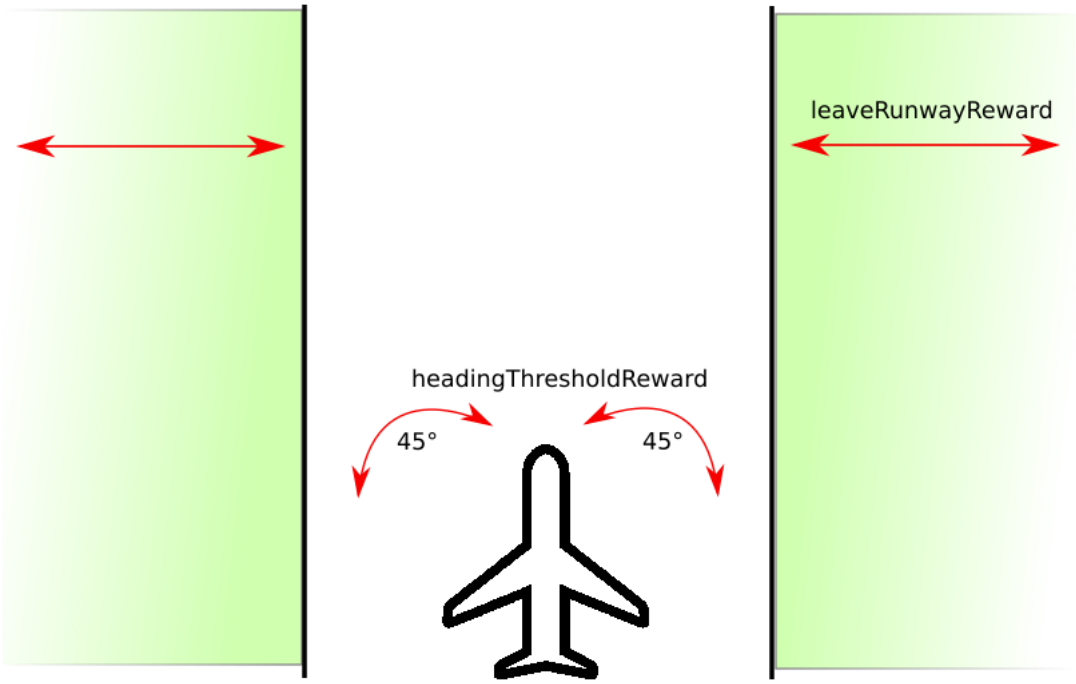


Abbildung 3.31.: Constraints für den Start (Eigene Erstellung)

Der Agent wird ebenfalls zurückgesetzt wenn dieser zu stark giert. In diesem Fall wird der Agent bei einer Kursabweichung von ± 45 zurückgesetzt. Dieser wird ebenfalls nicht sofort zurückgesetzt sondern nach einer festlegbaren Anzahl an Sekunden und sammelt dabei ein negatives Belohnungssignal **headingThresholdReward**.

- **Belohnung:** Wie erwähnt wird die Belohnung durch eine Differenz der zu maximierenden Belohnung und zu minimierender Belohnung geformt. Folgende Signale werden maximiert und minimiert:

$$\boldsymbol{Reward} = rewardSignalsToMaximize - rewardSignalsToMinimze$$

$$\boldsymbol{rewardSignalsToMaximize} = \text{distance} + \text{onRunway} + \text{normedHeight} + \text{normedPitch}$$

$$\boldsymbol{rewardSignalsToMinimze} = \text{crsDrift} + \text{rollDrift} + \text{leaveRunwayReward} + \text{headingThresholdReward}$$

  Die Belohnung des Agenten setzt sich aus den oben dargestellten Signalen zusammen. Zu maximieren gilt für den Agenten die Distanz zum Flughafen, die Höhe des Flugzeuges und der Neigungswinkel, so dass das Flugzeug von der Landebahn abhebt. Die Geschwindigkeit des Flugzeugs kann in diesem Fall als Belohnungssignal vernachlässigt werden, da die Triebwerke bei Start auf vollem Schub geschaltet sind. Für die Auswertung der Lage des Flugzeugs ist die Geschwindigkeit allerdings ausschlaggebend weshalb diese im Zustandsvektor vorkommt.

  Zu minimieren gilt die Abweichung von der Mittellinie der Landebahn und die Rollrate des Flugzeuges damit dieses möglich stabil beim Abheben bleibt. Die Signale **leaveRunwayReward** und **headingThresholdReward** werden wie erwähnt durch die aufgezwungenen Constraints erzeugt.

Kreuzfahrtflug

- **Zustandsvektor:** (normedPitch, normedSpeed, normedHeight, rollDrift, angularQ, angularP, angularR, throttle, rpm)
- **Constraints:** Die Constraints für einen stabilen Kreuzfahrtflug sind im Vergleich zu denen des Start etwas aufgelockert worden. Abbildung 3.32 stellt die Constraints grafisch dar:

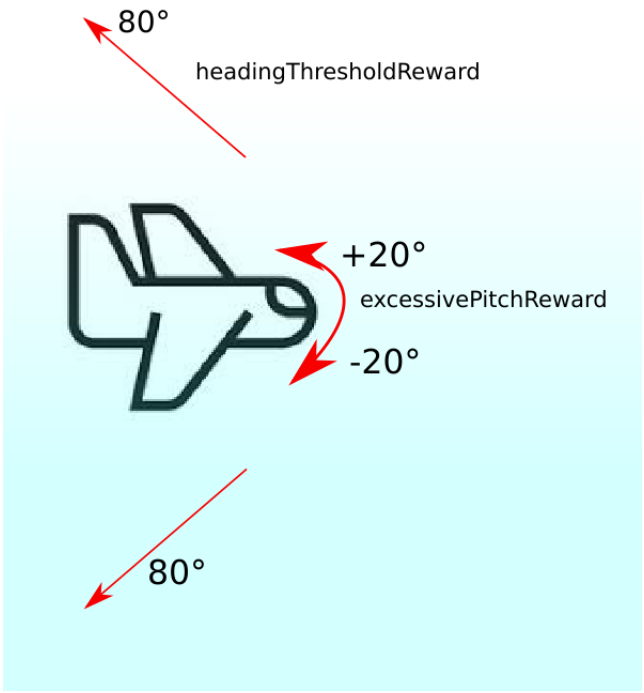


Abbildung 3.32.: Constraints für Kreuzfahrtflug (Eigene Erstellung)

Zunächst wird der Neigungswinkel, also die Attitüde des Flugzeugs auf maximal $\pm 20°$ beschränkt. Dadurch wird verhindert, dass das Flugzeug in eine zu steile Fluglage gerät und somit eine verheerende Situation einleitet. Wie beim Start wird die Kursabweichung eingeschränkt. In diesem Fall auf maximal $\pm 80°$ anstatt von ± 45. Da der Agent am Anfang Schwierigkeiten hat eine zentrierte Rolllage zu erlangen ist es wichtig dem Agenten Zeit zu geben, um diese Rolllage erreichen zu können. Bei einer strengeren Einschränkung verfällt der Lernprozess in ein lokales Optimum.

Zusätzlich wird ermittelt, ob die Triebwerke des Flugzeugs noch in Betrieb sind. Dadurch kann ermittelt werden, ob das Flugzeug abgestürzt ist oder ob die Triebwerke durch exzessive Fliehkräfte beschädigt worden sind.

- **Belohnung:** Die Belohnung setzt sich aus folgenden Signalen zusammen:

$$\boldsymbol{rewardSignalsToMaximize} = 5 * (1 - \text{distance})$$

$$\boldsymbol{rewardSignalsToMinimze} = 2 * abs(\text{normedHeight} - 1) + \text{centerDrift} + 4 * \text{rollDrift} + \text{excessivePitchReward} + \text{damageReward} + 2 * abs(\text{normedPitch})$$

  Die Belohnungssignale werden durch zusätzliche Faktoren verstärkt oder geschwächt. Beispielsweise ist in diesem Fall die abgelaufene Distanz das einzige Signal, das es zu maximieren gilt. Damit dieses einzelne Signal nicht zwischen den negativen Signalen an relevanz verliert wird dieses verstärkt.

  Besonders zu minimieren sind in diesem Fall ein Verlust oder Anstieg an Höhe, ein zu starker Neigungswinkel und die Rolllage des Flugzeugs die konstant zentral bleiben soll, weshalb eine Seitenlage des Flugzeugs besonders bestraft wird. Von geringerer Bedeutung ist, dass das Flugzeug nicht vom Kurs abweicht. Zudem werden die aus den Constraints stammenden Signale hinzugefügt.

Landung

- **Zustandsvektor:** (normedPitch, onRunway, normedSpeed, normedHeight, centerDrift, rollDrift, angularQ, angularP, angularR, distance, elevator, aileron, rudder, deviation, relBearing).
- **Constraints:** Wie beim Kreuzfahrtflug wird auch bei der Landung der Neigungswinkel des Flugzeugs eingeschränkt. Vor allem wird so vor dem Landen ein Stromabriss verhindert, der letzendlich zum Absturz des Flugzeugs führt. Der wichtigste Constraint bei der Landung ist allerdings die Abweichung von der Landebahn. Im Gegensatz zum Start kann diese nicht einfach mit der Abweichung des Kurses beschrieben werden, da während des Fliegens selbst die kleinste Kursabweichung eine Abweichung zur realen Mitellinie der Landebahn bedeutet.

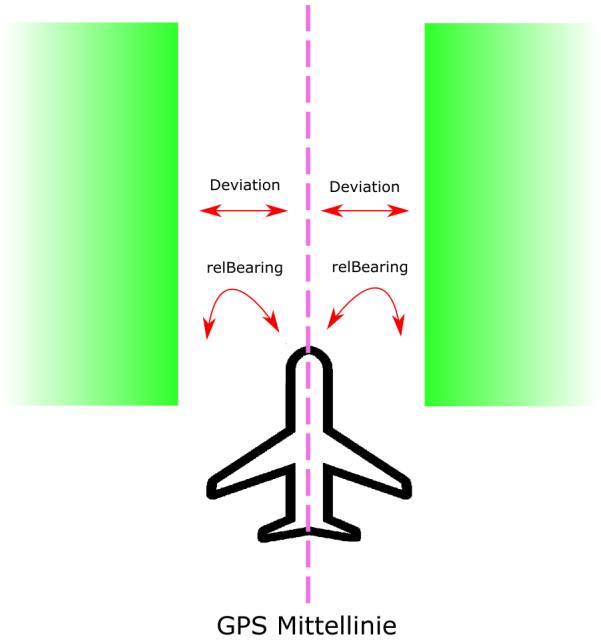


Abbildung 3.33.: Definierte Constraints für die Landung (Eigene Erstellung)

Es muss also die wahre Position des Flugzeugs in Bezug auf die Landebahn ermittelt werden. Abbildung 3.33 veranschaulicht die Komponenten, die für diese Berechnung relevant sind.

- **Deviation:** Es wird zunächst eine GPS Route zwischen zwei Wegpunkten geplottet, die über die Mittellinie der Landebahn geht. Der Course Deviation Indicator (CDI) des GPS Systems berechnet die Distanz zwischen der aktuellen Position des Flugzeugs und der gewünschten Mitellinie.
- **Richtung:** Das GPS liefert zudem auch die absolute Kursabweichung zum Kurs der Mitellinie. Hiermit wird auch die aktuelle Ausrichtung des Flugzeugs in Bezug auf die Mittellinie miteinbezogen.

- **Belohnung:** Grundsätzlich werden bei der Landung die gleichen Signale wie beim Start im Maximierungsterm verwendet. Die eigentliche Belohnung wird allerdings anhand dieser Signale auf eine andere Art und Weise berechnet:

$$\boldsymbol{rewardToMaximize} = onRunway \; + \; 2 * (1 - distance) \; + \; ((1 - distance) * (1 - normedHeight))$$

  - **onRunway:** Beschreibt wie beim Start, ob sich das Flugzeug auf der Fluglinie der Landebahn befindet. Dieser Parameter ist allerdings nicht immer zuverlässig, da der Flugsimulator auch die Rollbahnen als Landebahn bewertet.
  - **2 * (1 - distance):** Liefert eine Belohnung für die Entfernung zur Landebahn. Dabei wird als Referenzwert das Funksignal der Landebahn genommen. Die Entfernung ist auf 1 normiert wodurch diese am Anfang 1 beträgt und möglichst gering werden sollte. Um die niedriger werdende Distanz im Maximierungsterm verwenden zu können müssen die Funktionswerte mit (1 - distance) invertiert werden.
  - **(1 - distance) * (1 - normedHeight):** Hier wird die Belohnung für die Sinkrate des Flugzeugs modelliert. Ist das Flugzeug noch weit von der Landebahn entfernt so wird die Höhe niedriger gewichtet, wobei bei der Höhe die Funktionswerte wie bei der Distanz invertiert sind. Das Belohnungssignal wird maximal wenn sich das Flugzeug nah an der Landebahn und zudem auf einer niedrigen Höhe befindet. Allgemein kann so ein sanfter Abstieg modelliert werden.

Die Belohnungssignale im Minimierungsterm weisen hingegen eine geringere Komplexität auf:

$$\begin{aligned}\boldsymbol{rewardToMinimize} = {} & abs(deviation) + abs(relBearing) + leaveCenterLineReward \\ & + rollDrift + excessivePitchReward + damageReward \end{aligned} \tag{3.8}$$

Wie bei den anderen Fällen wird versucht alle Contraints minimal zu halten. Dazu zählt das Signal für die zu starke Abweichung von der Mitellinie **leaveCenterLineReward**, der zu steile Neigungswinkel **excessivePitchReward** und das Signal für einen Motorausfall **damageReward**.

Zudem wird auch wieder die Abweichung von der zentrierten Rolllage eingeführt, um exzessives Rollen zu vermeiden und die für die Constraints berechnete Mitellinienabweichung **deviation**. Da die Mitellinienabweichung nur sehr träge seine Tendenz ändert wird zusätzlich die normierte Kursabweichung **relBearing** hinzugefügt, die für

das präzise Landen ausschlaggebend ist. Ansonsten oszilliert das Flugzueg über die Landebahn, da es die sich ändernde Kursrichtung nicht vollständig wahrnimmt.

Abschließende Worte

In diesem Kapitel wurde die Implementierung der Reinforcement Learning Architektur erläutert, die das Flugzeug für 3 Anwendungsfälle steuern soll. Zunächst wurde ein Flugzeugautopilot auf einer konzeptuellen Ebene betrachtet. Dazu wurde ein herkömmlicher Flugzeugautopilot aus der Literatur entnommen, der in der Regel als ein Regelkreis beschrieben werden kann. Es wurden parallelen zur Reinforcement Learning Architektur gezogen, indem ein ähnlicher Regelkreis für den Reinforcement Learning Fall aufgestellt wurde. Dabei wurde festgestellt, dass der herkömmliche Flugzeugautopilot, die Flugdynamik und Umgebungsdynamik durch ein mathematisches Modell approximieren muss, um eine Steuerung zu ermöglichen. Die Reinforcement Learning Architektur besitzt hingegen die Fähigkeit genau diese Dynamiken durch Erfahrung zu approximieren und sollte von daher ein allgemeines Modell approximieren können.

Die Implementierung der Reinforcement Learning Architektur wurde in 3 Teilen präsentiert. Es wurde zuerst die Kommunikation mit dem Flugsimulator behandelt, der über speziell formatierte UDP-Pakete mit einem Client kommunizieren kann. Dabei kann der Client sowohl UDP-Pakete empfangen als auch UDP-Pakete an den UDP-Server des Flugsimulators senden, um beispielsweise das Flugzeug zu steuern. Als nächstes wurde die Implementierung des DDPG-Algorithmus erläutert. Dabei wurden zunächst grundlegende Konzepte Tensorflows erläutert, die für das Verständnis der konkreten Implementierung notwendig sind. So wurde zum Beispiel der Aufbau eines einfachen neuronalen Netzes Schritt für Schritt erläutert und nachvollzogen. Es wurden dann der Aufbau der Actor und Critic Netze des DDPG-Algorithmus präsentiert. Zuerst auf konzeptueller Ebene und dann in Tensorflow. Im gleichen Zusammenhang wurden dann die Trainingsfunktionen Tensorflows zum Ermitteln der Gradienten und der Parameteranpassung für die jeweligen Netze erläutert. Zum Schluss wurde die Implementation des DDPG-Algorithmus Schritt für Schritt durchgegangen anhand des Pseudocodes in Kapitel 2.3.2.

Als letztes wurde die Umsetzung der Belohnungsfunktion behandelt. Die konzeptuelle Auffassung einer Belohnungsfunktion wurde bereits in Kapitel 2.4 behandelt. Das Ziel hier war es die konkrete Umsetzung einer Belohnungsfunktion zu behandeln. Das heißt, welche Sensorwerte

gewählt wurden und wie diese bei der Modellierung der Belohnungsfunktion zum Einsatz gekommen sind, um dem Agenten das gewünschte Verhalten beizubringen.

4. Versuchsdurchführung und Ergebnisse

Das letzte Kapitel dieser Arbeit widmet sich den erzielten Ergebnissen des DDPG-Algorithmus zur Steuerung eines Flugzeugs in einem Flugsimulator. Wie im Einführungskapitel erwähnt soll der Flugzeugautopilot in 3 unterschiedlichen Szenarien eingesetzt werden: Beim Start, Kreuzfahrtflug und Landung. Dabei sollen auch unterschiedliche Bedingungen getestet werden wie zum Beispiel adverse Wetterbedingunen oder mechanische Schäden. Die Testmethodik, also die unterschiedlichen Szenarien und Bedingungen in denen sich der Agent beweisen soll werden im Folgenden erläutert.

Ebenso wird versucht ein Überblick über die Trainingsmethodik zu verschaffen. Das Trainieren des Flugzeugautopiloten ist von mehreren Faktoren abhängig, die sorgfältig analysiert werden müssen. Allgemein fällt das Reinforcement Learning Problem aufgrund des Experience Replay Buffers zum Teil zu einem Supervised Learning Problem zurück. Dies bedeutet, dass außer der Parametrisierung der Belohnungsfunktion und der Actor-Critic Architektur, auch das Sammeln relevanter Trainingsdaten ein wichtiger Bestandteil für den Lernerfolg sind.

Grundsätzlich muss die Trainingsmethodik mit der Testmethodik abgestimmt werden. Das heißt, die Testszenarien müssen zu den zum Training verwendeten Daten passen.

4.1. Trainingsmethodik

Am Anfang soll der Agent die Umgebung erforschen. Die Belohnungsfunktion und die Erforschungsstrategie des Agenten müssen dabei aufeinander abgestimmt werden. So müssen die Constraints der Belohnungsfunktion am Anfang für die Erforschungsstrategie aufgelockert werden, um dem Agenten eine optimale Erforschung zu ermöglichen. Mit fortschreitendem Trainingsfortschritt wird die Stochastizität der Erforschungsstrategie verringert und die Constraints der Belohnungsfunktion verengt, so dass das definierte Lernziel erreicht werden kann. Dieses Zusammenspiel zwischen Erforschung und nachfolgender Bestimmung des Ziels wird Erforschung und Ausbeutung (**Exploration and Exploitation**) genannt. Das Training des Agenten teilt sich somit in 2 Phasen auf:

1. Sammeln von Erfahrung anhand eines stochastischen Prozesses, der den Agenten die Umgebung erforschen lässt.
2. Bei fortschreitendem Trainingsvorschritt den Agenten zu einem bestimmten Ziel konvergieren lassen.

Die Erforschungsstrategie sollte für jedes Szenario individuell bestimmt werden. Vor allem bei der Steuerung eines Flugzeugs ist das Sammeln von relevanten Transitionen keine triviale Aufgabe. So muss die Erforschungsstrategie das Sammeln von Transitionen ermöglichen, die dem gewünschten Verhalten des Agenten entsprechen. Zudem müssen die Transitionen untereinander eine gewisse Stochastizität aufweisen. da ansonsten im nachhinein ein Overfitting des Modells droht. Im Folgenden wird die Modellierung eines parametrisierbaren stochastischen Prozesses beschrieben, der die Spezifizierung einer Erforschungsstrategie ermöglicht.

4.1.1. Erforschungsstrategie: Ornstein-Uhlenbeck-Prozess

Die Erforschungsstrategie für den Agenten wird durch eine stochastische Differentialgleichung beschrieben. Im Vergleich zur mathematischen Modellierung mit einer einfachen deterministischen Differentialgleichung besitzen stochastische Differentialgleichungen die Eigenschaft einem Zusätzlich Störfaktor ausgesetzt zu sein. Eine stochastische Gleichung wird durch zwei Funktionen $a, b\colon \mathbb{R} \times \mathbb{R}_+ \to \mathbb{R}$ und einem Zufallsprozess W_t beschrieben:

$$dX_t = a(X_t, t)dt + b(X_t, t)dW_t \tag{4.1}$$

Zu den Funktionen a, b und dem Zufallsprozess W_t wird eine Lösung zur oben beschriebenen Differenzialgleichung gesucht. Die Funktionen a und b werden dabei als Drift und Diffusionskoeffizienten bezeichnet. Der sogenannte Ornstein-Uhlenbeck-Prozess wird durch eine solche stochastische Differentialgleichung modelliert.

Der Ornstein-Uhlenbeck-Prozess wird durch die Konstanten $a, \mu \in \mathbb{R}$ und $\theta, \sigma > 0$ definiert:

$$dX_t = \theta\dot{(}\mu - X_t)dt + \sigma dW_t, \quad X_0 = a \tag{4.2}$$

Dabei ist W_t ein Zufallsprozess der zufällige Werte aus einer Normalverteilung generiert. In diesem konkreten Fall wird hierzu die Numpy Bibliotheksfunktion **randn** verwendet, die Zufallswerte aus einer Normalverteilung liefert. Der Ornstein-Uhlenbeck-Prozess wird iterativ für $t \geq 0$ ausgewertet. Dabei ist der aktuelle Wert X_t der Wert X_{t-1} des vorherigen Auswer-

tungsschritts der Differentialgleichung. Die Konstanten wirken hierbei als Stellschrauben, um den stochastischen Prozess zu modellieren:

- μ**:** Beschreibt den Mittelwert des stochastischen Prozesses, das heißt der stochastische Prozess sollte ungefähr zu diesem Mittelwert zurückfallen. So liefert der Term $(\mu - X_t)$ die Abweichung zu diesem Mittelwert (Driftterm). Liegt X_t über dem Mittelwert μ, so wird der Zufallsprozess W_t nach unten gezogen, da der Term negativ wird. Umgekehrt wird der Term positiv und zieht somit den Zufallsprozess nach oben.
- θ**:** Agiert als Verstärkungsfaktor für den Driftterm. Je höher dieser Wert, umso stärker verfällt der Ornstein-Uhlenbeck-Prozess zum definierten Mittelwert μ.
- σ**:** Bestimmt den Einfluss des Zufallsprozesses W_t (Diffussionskoeffizient). Ist $\sigma = 0$ so konvergiert X_t exponentiell gegen μ. Ansonsten wird W_t gestört und somit mehr stochastizität in den Prozess $X s_t$ gebracht.

Abbildung 4.1 zeigt 3 verschiedene Ornstein-Uhlenbeck Prozesse:

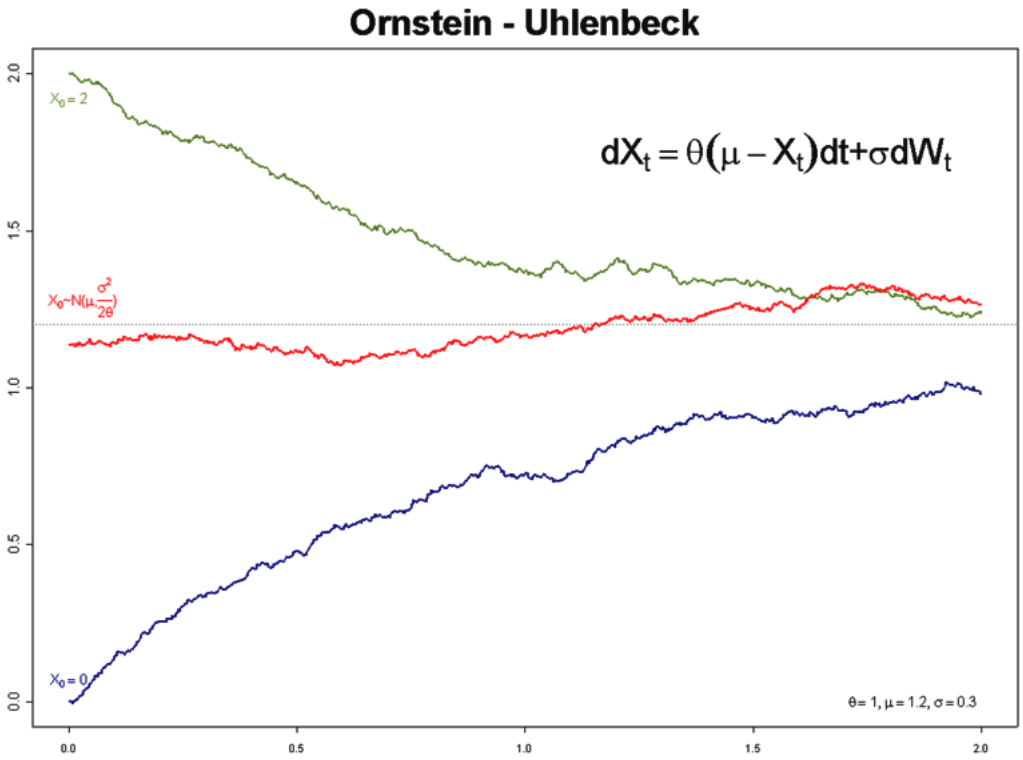


Abbildung 4.1.: Ornstein-Uhlenbeck-Prozess für unterschiedliche Anfangswerte (Thomas Steiner unter CC BY-SA 3.0 Lizenz)

Mit den Parametern $\theta = 1, \mu = 1.2, \sigma = 0.3$ wird für 3 unterschiedliche Anfangswerte $X_{0_n} = (0, \sim \mu, 2)$ der Ornstein-Uhlenbeck-Prozess geplottet. Es ist zu sehen, dass alle 3 Prozesse ungefähr auf den Mittelwert $\mu = 1.2$ zurückfallen. Das kleine σ führt dabei eine leichte Stochastizität ein. Die Bestimmung eines Anfangswertes bestimmt auch den Verlauf des stochastischen Prozesses. Dies wird als Anfangswertproblem bezeichnet, wo die Lösung zur Differentialgleichung zusätzlich die definierte Anfangsbedinung einhalten muss. Unterschiedliche Anfangswerte führen so zu unterschiedlichen Lösungen für die stochastiche Differentialgleichung des Ornstein-Uhlenbeck-Prozesses.

4.1.2. Trainingsmethodik im DDPG-Algorithmus

Zusammengefasst kann der Trainingsvorgang des DDPG-Algorithmus wie in Abbildung 4.2 abstrakt dargestellt werden:

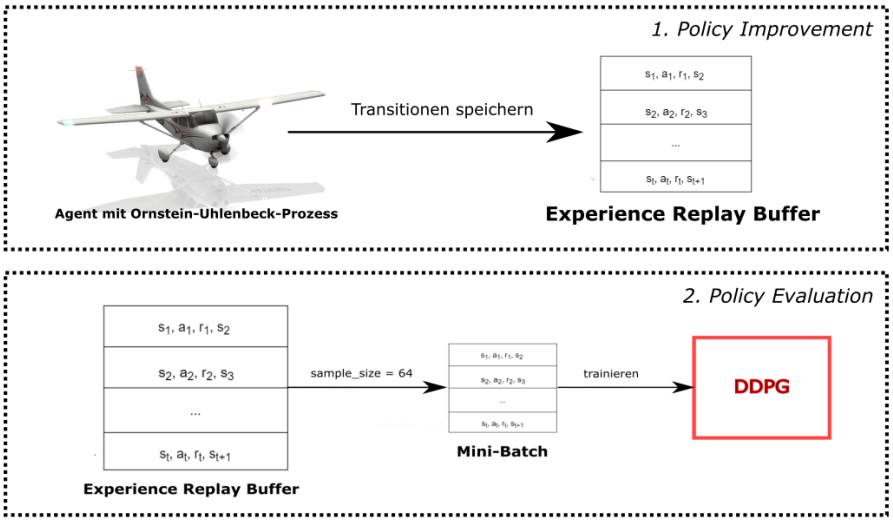


Abbildung 4.2.: Zu testende Szenarien (Eigene Erstellung)

1. Der Agent führt die durch den Ornstein-Uhlenbeck-Prozess definierte Erforschungsstrategie aus.

2. Der Experience Replay Buffer speichert dabei zunächst die generierten Transitionen ohne die Actor-Critic Architektur zu trainieren.

3. Nach einer fest definierten Anzahl an Transitionen wird aus dem Experience Replay Buffer ein Mini-Batch gesampled anhand dessen die Actor-Critic Architektur trainiert wird.

4. Während des Trainings werden weiterhin Transitionen im Experience Replay Buffer gespeichert. Das Mini-Batch zum Training wird hingegen unabhängig von der aktuellen Aktion des Agenten aus dem Experience Replay Buffer entnommen (in Kapitel 2.3.1 wurde der DDPG-Algorithmus deshalb als eine Off-Policy Actor-Critic Architektur bezeichnet).

5. Mit fortschreitendem Trainingsfortschritt wird der Ornstein-Uhlenbeck-Prozess abgestellt und die Constraints des Agenten verschärft.

Die Trennung zwischen dem Ausführen der Policy und der Evaluierung der Policy wird hier offensichtlich. Im Policy Improvement Schritt führt der Agent, die aus der Actor-Critic Architektur gerade angepasste Policy aus und speichert die Transitionen der neuen Policy ab. Im

Policy Evaluation Teil trainiert der DDPG-Algorithmus mit diesen gespeicherten Transitionen. **Genau dies ist die Off-Policy Eigenschaft des DDPG-Algorithmus. Das Netzwerk wird mit Transitionen trainiert, die unterschiedlich zu den geraden erfassten Transitionen sind.**

4.2. Erläuterung der Flugszenarien und der Auswertungsmethodik

Im Folgenden werden die einzelnen Testszenarien näher erläutert. Dabei wird die konkrete Durchführung und Parameterisierung des Trainingsvorgangs aufgeführt, die Auswertungsmethodik und die jeweiligen Bewertungskriterien für die unterschiedlichen Szenarien präsentiert. Zunächst werden allerdings noch die zum Einsatz gekommenen Fluggeräte und der Flugort vorgestellt.

4.2.1. Fluggeräte und Flugort

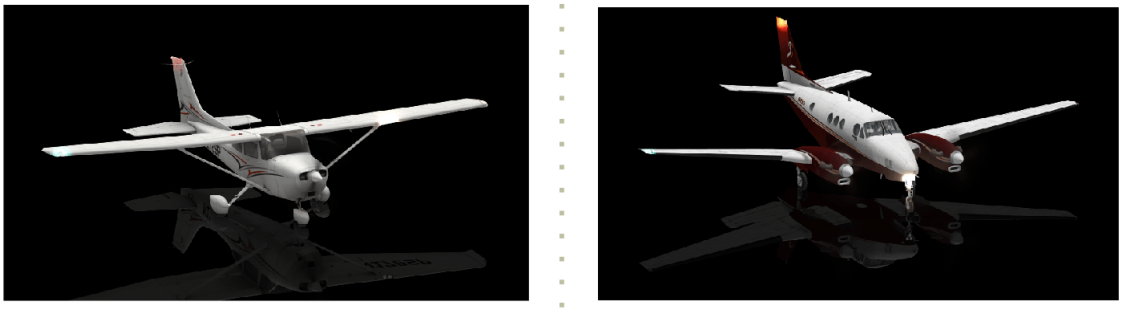


Abbildung 4.3.: Cessna 172P (links) und KingAir C-90 (rechts) (Eigene Erstellung)

Für den Start und den Kreuzfahrtflug wurde die Beechcraft King Air C90 verwendet. Die King Air C90 besitzt zwei Turbopropeller-Triebwerke und ist einer der meistverkauften und bekanntesten Kleinpassagierflugzeuge. Vor allem wird diese auch oft für Pilotenschulungen eingesetzt. Für die Landung hingegen wurde eine Cessna 172P eingesetzt. Das Landen der KingAir C90

erweist sich als recht komplex, da während der Landung die Flaps und das Fahrwerk zum richtigen Zeitpunkt ausgefahren werden müssen. Die Cessna 172P kann hingegen ohne das Aktivieren der Flaps gelandet werden. Da das Landen das ohnehin das komplexeste Szenario ist, dass zu bewältigen gilt wurde hierfür die Cessna 172P statt der KingAir C90 verwendet.

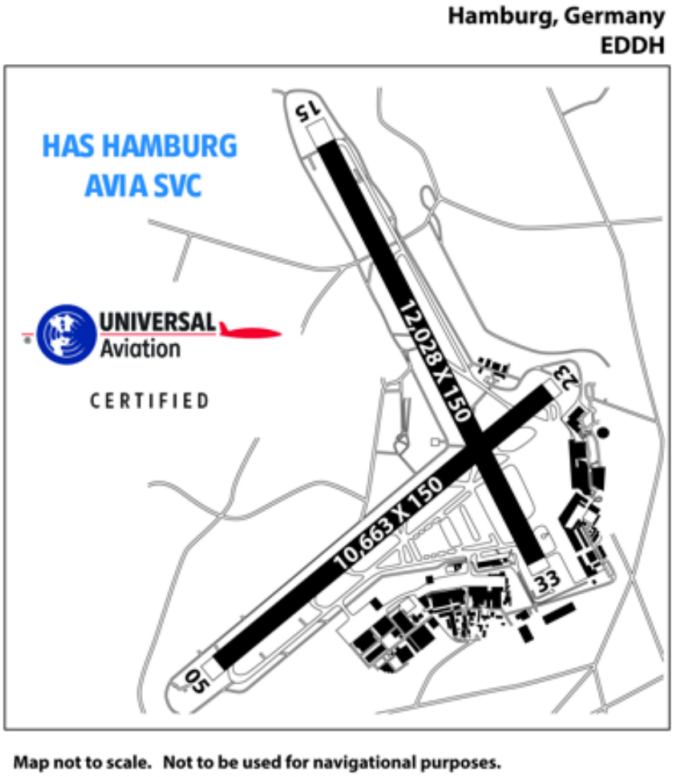


Abbildung 4.4.: Plan des Hamburg Airport EDDH

Da die Hochschule für Angewandte Wissenschaften der Standort der Arbeit ist wurde als Flughafen der Hamburg Airport identifiziert durch EDDH als Flugort gewählt. Dabei erweist sich als praktisch, dass der Hamburg Airport einen überschaubaren Lageplan hat, wie in Abbildung 4.4 zu sehen ist. Als Landebahn wurden durchgängig die Bahn 23 bzw. 05 verwendet, da diese direkt auf das Radiosignal ALF (Alster) gerichtet ist.

4.2.2. Erläuterung der Flugszenarien und Bewertungskriterien

Der Versuch besteht aus 3 Hauptszenarien: **Start, Kreuzfahrtflug und Landung**. Darüber hinaus werden für jedes Hauptszenario 3 Unterszenarien definiert, die unterschiedliche Hindernisse an den zu trainierenden Agenten darstellen sollen. Dabei wird unter **normalen**

Bedingungen, adversen Wetterbedingungen und bei Auftreten eines technischen Defekts getestet. Abbildung 4.5 erläutert den Zusammenhang grafisch:

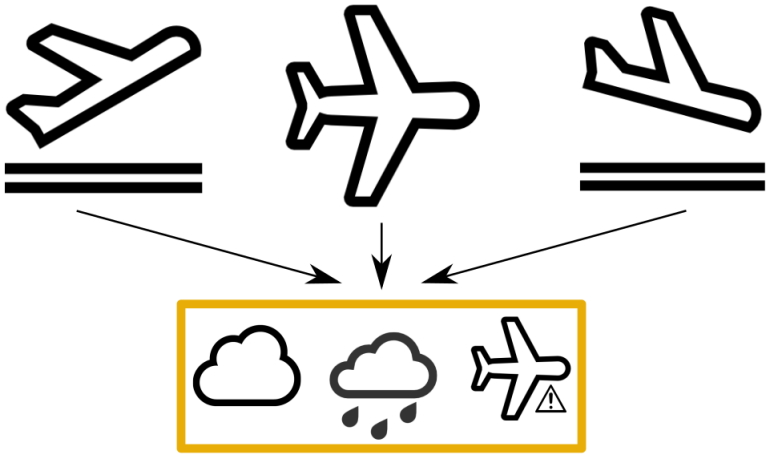


Abbildung 4.5.: Zu testende Szenarien (Eigene Erstellung)

Im Folgenden werden für alle 3 Flugszenarien folgende 3 Aspekte behandelt:

1. Aufbau des Szenarios.
2. Trainings und Testdurchführung.
3. Auswertungsmethodik: Bewertungsmetriken und Bewertungskriterien.

(1) Zunächst wird der Aufbau des Szenarios erläutert. Dabei werden die verwendeten Fluggeräte, Steuerflächen und definierten Randbedingungen der jeweiligen Unterszenarien analysiert.

(2) Die Bewältigung der schwierigeren Szenarien erweist sich nicht immer als selbstverständlich. Teilweise brauchen einige Szenarien Vorkentnisse in Form eines bereits trainierten Modells, um das gewünschte Verhalten zu erreichen. In anderen Fällen sind die Modelle schlecht übertragbar und lediglich ein Training von Anfang an ermöglicht das Bewältigen des Szenarios. In diesem Abschnitt wird die konkrete Trainingsform für das jeweilige Szenario und Unterszenarien näher erläutert.

(3) Für jedes Szenario müssen zudem eigene Kriterien definiert werden nach denen das Verhalten des Agenten bewertet werden soll. Dabei soll definiert werden mit welchen Metriken der Erfolg des Agenten gemessen werden soll. Im gleichen Zusammenhang wird für jedes Szenario

anhand dieser Metriken bestimmt ab wann ein Szenario als erfolgreich bestanden gilt.

(1) Aufbau des Szenarios

Start

Für den Start wird die Beechcraft King Air C90 verwendet. Diese startet von der Landebahn 23, also in Kursrichtung 230°. Die Triebwerke des Flugzeugs werden bei Anfang automatisch mit vollem Schub versorgt und gleichzeitig nach einer kleinen Verzögerung die Parkbremsen gelöst. Als Steuerflächen werden das Höhenruder **(elevator)**, Querruder **(aileron)** und Seitenruder **(rudder)** verwendet.

Parametrisierung der Unterszenarien

Die 3 Unterszenarien werden folgendermaßen parametrisiert:

- **Normale Bedinungen:** Der Test unter normalen Bedingungen erfolgt bei wolkenslosem Himmel und keinem Wind.
- **Adverse Wetterbedingungen:** In Abbildung 4.6 sind die eingestellten Wetterbedingungen für das Szenario der extremen Wetterbedingungen dargestellt.

Abbildung 4.6.: Adverse Wetterbedingungen bei Start

Dabei wird ein Kreuzwind mit Richtung 135° mit einer Geschwindigkeit von 12 Knoten und einer Böengeschwindigkeit von 8 Knoten eingestellt. Der Parameter **Turbulenz** simuliert zudem belieibig auftretende Turbulenzen während des Fluges. Für dieses Szenario wurden diese auf die maximale Stufe **Severe** eingestellt.

- **Defekt:** In diesem Szenario wird während des Starts ein Defekt simuliert. In diesem Fall wird ab einem bestimmten Zeitpunkt des Fluges die Flugfähigkeit des linken Flügels beeinträchtigt und der Agent dazu gezwungen sich an die neue Fluglage anzupassen. X-Plane teilt bei der King-Air den Flügel in 4 Teile auf. Dabei werden Teile 1 bis 3 zum Ausfall programmiert, wodurch das Flugzeug in eine dauerhafte Seitenlage gerät.

Im Folgenden wird die Parametrisierung des Trainings für das Szenario aufgeführt. Dabei wird die Parametrisierung des Ornstein-Uhlenbeck Zufallsprozesses und die Parametrisierung der Constraints, die Episodenabhängig justiert werden.

Weitere Parametrisierungen

Der Ornstein-Uhlenbeck-Prozess besitzt die Parameter θ, σ und μ die wie folgt parametrisiert werden:

Parameter	θ	μ	σ
Werte	0.15	0.0	0.08

Tabelle 4.1.: Parametrisierung Ornstein-Uhlenbeck-Prozess für Start

Der Prozess verläuft mit dieser Parametrisierung dann wie in Abbildung 4.7 dargestellt:

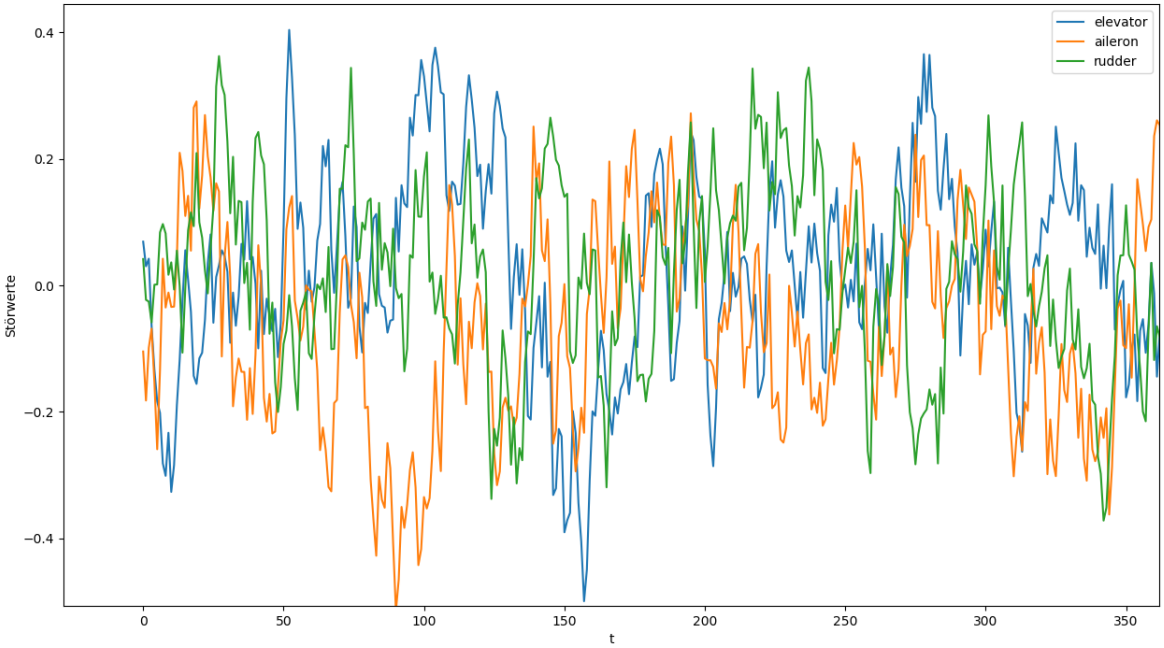


Abbildung 4.7.: Ornstein-Uhlenbeck Parametrisierung (Eigene Erstellung)

Die Störwerte für die 3 Steuergrößen **elevator, aileron und rudder** schwingen in einem Bereich zwischen -0.4 und 0.4. Dabei ist die Stochastizität sehr hoch aber der Verfall zum Mittelwert eher langasm. So bekommt der Agent im Laufe der Erforschung stabile Störwerte des Prozesses ohne ständig in die Extremwerte zu verfallen.

Die Parametrisierung der Contraints wird in der nachfolgenden Tabelle präsentiert:

Episode	**headingThreshold**
≤ 800	75°
> 800	45°

Tabelle 4.2.: Parametrisierung Ornstein-Uhlenbeck-Prozess für Start

Der Constraint für die Abweichung des Kurses **headingThreshold** wird nach 800 Episoden nachjustiert. Dabei wird der Threshold um 30° reduziert, um das Verhalten des Agenten weiter einzuschränken. Alle weitern Constraints wurden im Laufe des Trainings nicht modifiziert.

Kreuzfahrtflug

Für den Kreuzfahrtflug wird ebenfalls die Beechcraft KingAir C90 verwendet. Der Kreuzfahrtflug startet beim Anflug auf die Landebahn 23 des Hamburg Airports. Das Flugzeug befindet sich dabei 10 nautische Meilen vom Flughafen entfernt auf einer Höhe von 2100 Fuß. Als Steuerflächen sind das Höhenruder **(elevator)**, Querruder **(aileron)** und der Motorschub **(throttle)** vorgesehen.

Parametrisierung der Unterszenarien

Die 3 Unterszenarien werden folgendermaßen spezifiziert:

- **Normale Bedinungen:** Der Test unter normalen Bedingungen erfolgt wiederum bei wolkenslosem Himmel und keinem Wind.
- **Adverse Wetterbedingungen:** Die adversen Wetterbedingungen in Abbildung 4.8 unterscheiden sich von den eingestellten Wetterbedingungen des Starts:

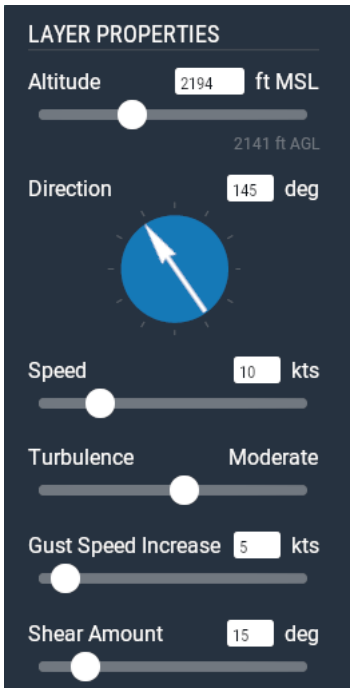


Abbildung 4.8.: Adverse Wetterbedingungen bei Kreuzfahrtflug (Eigene Erstellung)

Allgemein ist der beim Start eingestellte Kreuzwind etwas aufgelockert worden, um das erfolgreiche Training des Agenten zu ermöglichen.

- **Defekt:** Beim Kreuzfahrtflug wird ebenfalls die Tragfähigkeit der linken Tragfläche wie beim Start eingeschränkt. Nach 150 Zeitschritten wird die Flugfähigkeit des linken Flügels beeinträchtigt.

Weitere Parameterisierungen

Die Parametrisierung des Ornstein-Uhlenbeck-Prozesses ist mit der Parametrisierung des Starts identisch und kann Abbildung 4.7 entnommen werden. Obwohl der Motorschub keine negativen Werte annimmt, wertet der Flugsimulator unzulässige Steuerwerte je nachdem mit dem Minimal oder Maximalausschlag aus.

Beim Kreuzfahrtflug wird der gleiche Constraint **headingThreshold** mit fortschreitendem Trainingsfortschritt verändert:

Episode	headingThreshold
≤ 1000	80°
> 1000	60°

Tabelle 4.3.: Parametrisierung Ornstein-Uhlenbeck-Prozess für Start

Landung

Für die Landung wird eine Cessna 172P eingesetzt. Die Cessna 172P besitzt ein fixiertes Fahrwerk und kann ohne Ausfahren der Flaps gelandet werden. Dies reduziert die Komplexität der Landung enorm, da die KingAir C90 ausschließlich durch das Betätigen der Flaps zur richtigen Zeit gelandet werden kann. Die Landung wird ebenfalls auf Landebahn 23 ausgeführt, wobei das Flugzeug sich 1.5 nautische Meilen von der Landebahn auf 1000 Fuß Höhe befindet. Als Steuerflächen sind wiederum das Höhenruder **(elevator)**, Querruder **(aileron)** und der Motorschub **(throttle)** vorgesehen.

Parametrisierung der Unterszenarien

Die 3 Unterszenarien werden folgendermaßen spezifiziert:

- **Normale Bedinungen:** Der Test unter normalen Bedingungen erfolgt wiederum bei wolkenslosem Himmel und keinem Wind.
- **Adverse Wetterbedingungen:**

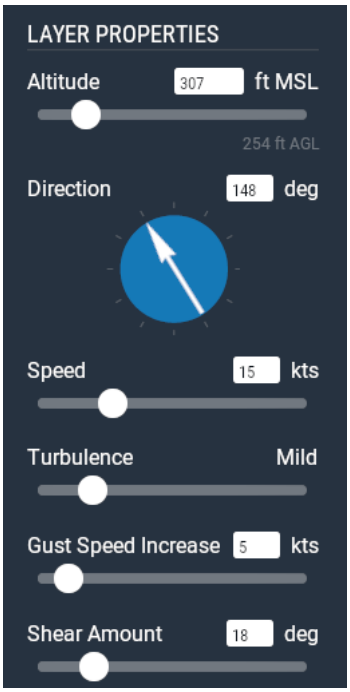


Abbildung 4.9.: Adverse Wetterbedingungen bei Landung (Eigene Erstellung)

Für die Landung mussten die Turbulenzen weiterhin aufgelockert werden. Statt **Moderate** sind diese auf **Mild** zurückgestuft worden.

- **Defekt:** Wie bei den 2 anderen Szenarien wird auch hier die linke Tragfläche beeinträchtigt. Dies geschieht unmittelbar nach Anfang der Episode.

Weitere Parametrisierungen

Die Parametrisierung des Ornstein-Uhlenbeck-Prozesses wird vom Kreuzfahrtflug übernommen. Der anzupassende Constraint ist dabei die **Deviation**, also die zentrale Abweichung zur Landebahn:

Episode	Deviation
≤ 1000	0.5
> 1000	0.3

Tabelle 4.4.: Parametrisierung des Constraints Deviation

(2) Trainings und Testdurchführung

Das Training und Testen ist für alle 3 Szenarien größtenteils gleich strukturiert. Zunächst

wird der Autopilot für alle 3 Szenarien und 3 Unterszenarien trainiert wie in Abbildung 4.10 dargestellt. Nach dem Training werden jeweils alle 3 Unterszenarien getestet.

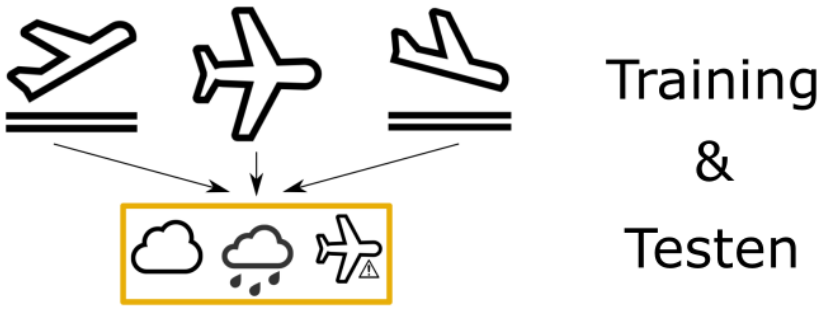


Abbildung 4.10.: Trainings und Testaufbau (Eigene Erstellung)

Um die Generalisierungsfähigkeit der Modelle zu testen wird zudem für jedes Szenario, das unter normale Bedingungen trainierte Modell gegen die 2 anderen Szenarien getestet (Abbildung 4.11).

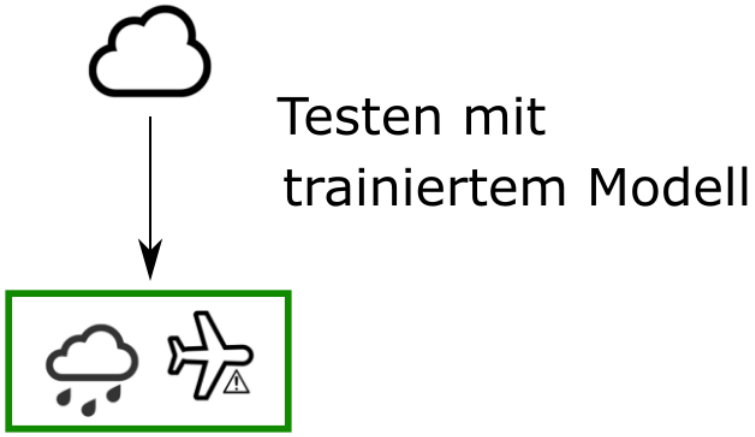


Abbildung 4.11.: Trainings und Testaufbau (Eigene Erstellung)

Letztendlich werden einige spezifische Phänomene untersucht. Abbildung 4.12 veranschaulicht die zusätzlich durchzuführenden Tests:

- **Erforschungsstrategie:** Es wird die Auswirkung der Erforschung zu Anfang des Trainings analysiert, in dem ein Modell mit Erforschungsstrategie und ein Modell ohen Erforschungsstrategie trainiert wird.t

- **Experience Replay Buffer:** Als zweites wird der Einfluss des Experience Replay Buffers auf die Trainingsperfomance analysiert, in dem zwei Modelle entweder mit dem Experience Replay Buffer oder lediglich mit der letzten Transition trainiert werden.
- **Constraints:** Letztendlich wird analysiert welche Auswirkung das Verändern der Constraints im Laufe des Trainings auf die Trainingsperformance haben, in dem diese aufgeschwächt und während des Trainings nicht modifiziert werden.

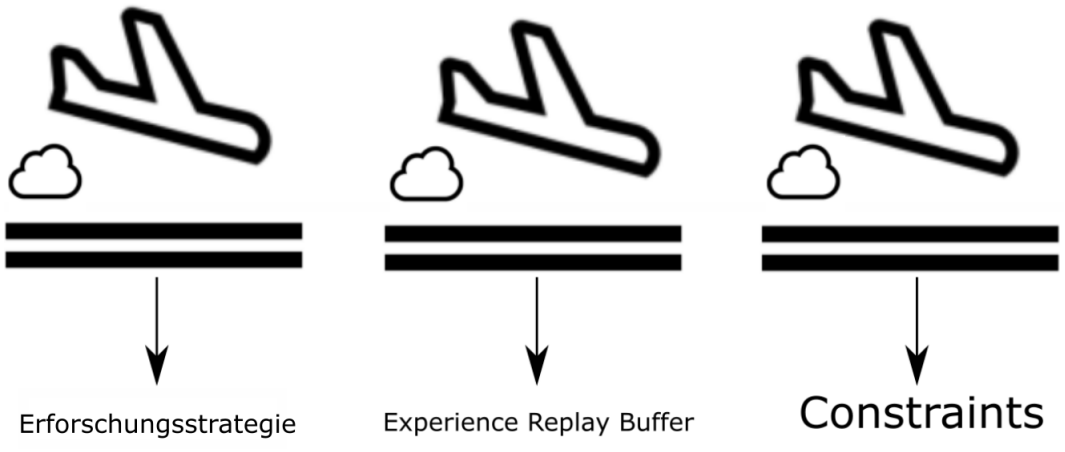


Abbildung 4.12.: Zusätzliche Tests (Eigene Erstellung)

Der nachfolgende Abschnitt beschäftigt sich unter anderem mit der Präsentierung der jeweiligen Testergebnisse. Dabei werden die Auswertungsmethoden erläutert und die jeweiligen Bewertungskriterien für die verschiedenen Flugszenarios aufgeführt.

(3) Präsentation der Ergebnisse, Auswertungsmethodik und Bewertungskriterien

Die Präsentation der Ergebnisse erfolgt in 2 Teilen. Zum einen werden die rohen Daten gegen die Bewertungskriterien geprüft und mit einfachen statistischen Methoden analysiert. Zum anderen wird auch das Flugverhalten des Flugzeugautopiloten analysiert in dem die aufgezeichneten Flugtrajektorien analysiert werden.

Die rohen Daten werden zunächst in roher Form und anschließend mit einem gleitenden Mittelwert präsentiert, um die Veranschaulichung zu erleichtern (Abbildung 4.13). Besonders beim Vergleich der Trainingsmethoden ensteht hierdurch eine klarere Übersicht.

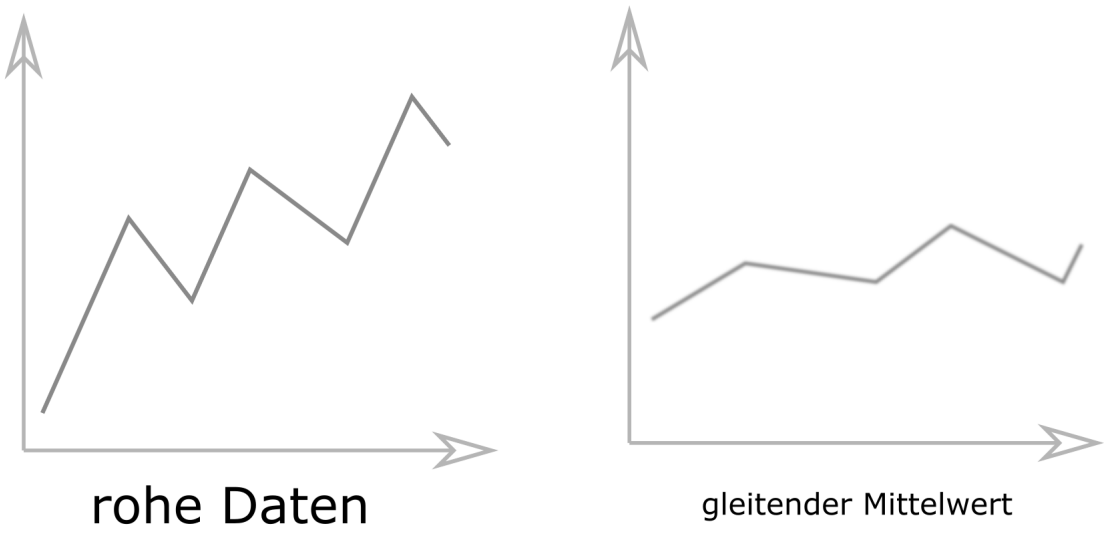


Abbildung 4.13.: Rohe Daten und gefilterte Daten (Eigene Erstellung)

Außer das Prüfen gegen die Bewertungskriterien sollen auch einfache statistische Methoden angewendet, um den Vergleich zweier Testdurchläufe zu erleichtern. In diesem Fall werden folgende Methoden verwendet, um die verschiedenen Testdurchläufe miteinander zu vergleichen:

- **Durchschnitt:** Beispielsweise kann die durchschnittliche Höhe für einen oder mehrere Testdurchläufe des Kreuzfahrtflugs ermittelt werden.
- **Standardabweichung:** In diesem Zusammenhang ist die Ermittlung der Standardabweichung ausschlaggebend, um ein besseres Verständnis über den Kontext verschaffen zu können. Zwar kann die durchschnittliche Höhe eines Durchlaufs höher als die eines anderen Durchlaufs sein, so kann aber eine hohe Standardabweichung auf einen unstabileren Flug hinweisen.

Selbst wenn die geforderten Erfolgskriterien erfüllt werden sagen diese eher wenig über das Flugverhalten des Flugzeugautopiloten aus. Deshalb sind außer den rohen Daten für jedes Szenario auch Trajektorien aufgezeichnet worden (Abbildung 4.14), um so das Flugverhalten des Flugzeugautopiloten qualitativ bewerten zu können.

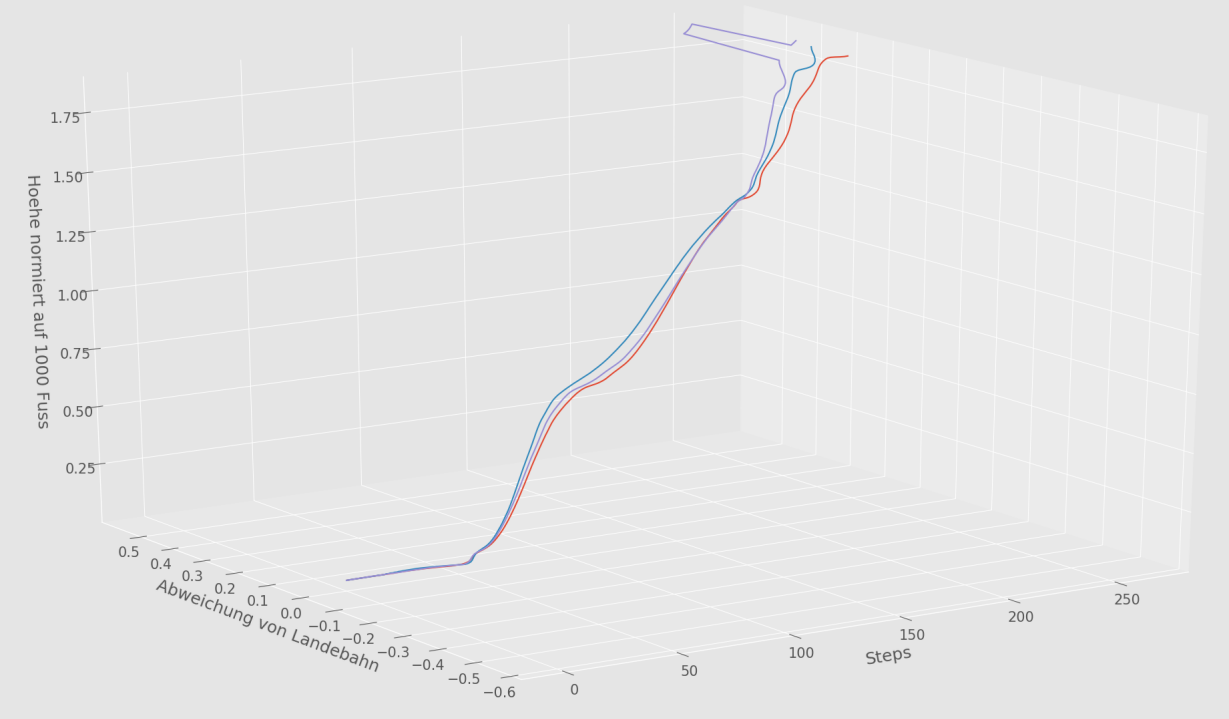


Abbildung 4.14.: Flugverhalten (Eigene Erstellung)

Bewertungskriterien

Als letztes werden nun die Bewertungskriterien für jedes Szenario definiert nach welchen der Erfolg des Flugzeugautopiloten gemessen werden soll. Dabei werden sowohl notwendige als auch hinreichende Bedingungen in Tabelle 4.27 formuliert:

	Start	**Kreuzfahrtflug**	**Landung**
Notwendige Bedingungen	• Durchschnittliche Flughöhe > 150 Ft. • Kein Absturz	• Duchrschnittliche normierte Flughöhe > 0.9 • Kein Absturz	Erfolgreiche Landung
Hinreichende Bedingungen	Durchschnittliche Kursabweichung $\pm$ 50	Durchschnittliche Kursabweichung $\pm$50	Landebahnabweichung $\leq$ 0.3

Tabelle 4.5.: Bewertungskriterien

4.3. Ergebnisse und Auswertung

Im Folgenden werden die Ergebnisse in der im vorherigen Kapitel vorgestellten Form ausgewertet. Dabei werden zunächst die Ergebnisse der von Grund auf trainierten Modelle präsentiert und interpretiert. Um die Generalisierungsfähigkeit eines trainierten Modells zu analysieren werden die Ergebnisse des unter normalen Bedinungen trainierten Modells in anderen Bedingungen analysiert. Zudem werden auch Trainingsergebnisse analysiert und überpüft, ob eine Art Konvergenz besteht. Als letztes werden die im vorherigen Kapitel spezifizierten Spezialfälle behandelt.

4.3.1. Testergebnisse der einzelnen Szenarien

Start

Für den Start werden die Testergebnisse für alle 3 Unterszenarien präsentiert. Es werden zunächst die Bewertungskriterien behandelt, wobei auch die Übertragungsfähigkeit des unter normalen Bedingungen trainierten Modells untersucht wird. Anschließend wird eine qualitative Einschätzung des Flugverhaltens abgegeben.

Auswertung der notwendigen und hinreichenden Bedingungen

Es werden im Folgenden die durchschnittliche erreichte Höhe (Abbildung 4.15a) und der durchschnittliche Kurs (Abbildung 4.15b) während des Starts auf die notwendigen und hinreichenden Bedingungen überprüft. Die jeweiligen Mittelwerte und Standardabweichungen können den Tabellen 4.6 und 4.7 entnommen werden. Die durchschnittliche Höhe ist auf 1 von 1000 Fuß Höhe normiert. Der durchschnittliche Kurs wird unverändert von 0 bis 359 Grad angegeben. Zudem werden die Anzahl der Durchläufe ohne Absturz in Tabelle 4.8 aufgelistet. Dabei werden die Ergebnisse aller 3 Unterszenarien in den Folgenden Abschnitten kommentiert:

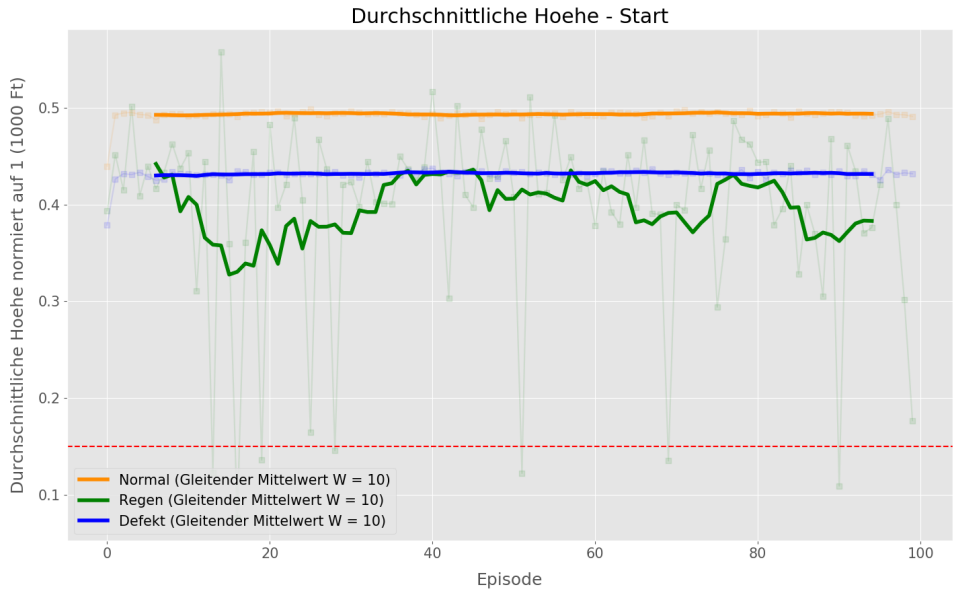
- Unter **normalen Bedingungen** beträgt die normierte durschnittliche Höhe **0.493**. Dies liegt weit über die definierte Bedingung von 0.150. Dabei ist die Standardabweichung mit 0.0057 (1.15 %) sehr gering. Eine Erfolgsrate von **100%** bedeutet ebenfalls, dass der Agent in 100 Testdurchläufen kein einziges mal abgestürzt ist. Die notwendigen Bedingungen sind somit erfüllt. Der Grund für die Robustheit liegt darin, dass in diesem Fall Trainings und Testumgebung fast identisch sind.Da unter normalen Bedingungen keine Störfaktoren präsent sind ist die Testumgebung der Trainingsumgebung sehr ähnlich. Daher ist der Agent auf diese Bedingungen sehr genau spezialisiert wodurch die niedrige Standardabweichung und die stabilen Ergebnisse zu erklären sind.

Betrachtet man den durchschnittlichen Kurs von **234.78** mit einer Standardabweichung von **3.16 (1.34 %)** so fällt auf, dass der Flugzeugautopilot das Flugzeug beim Start sehr stabil in Richtung der Landebahn halten konnte.

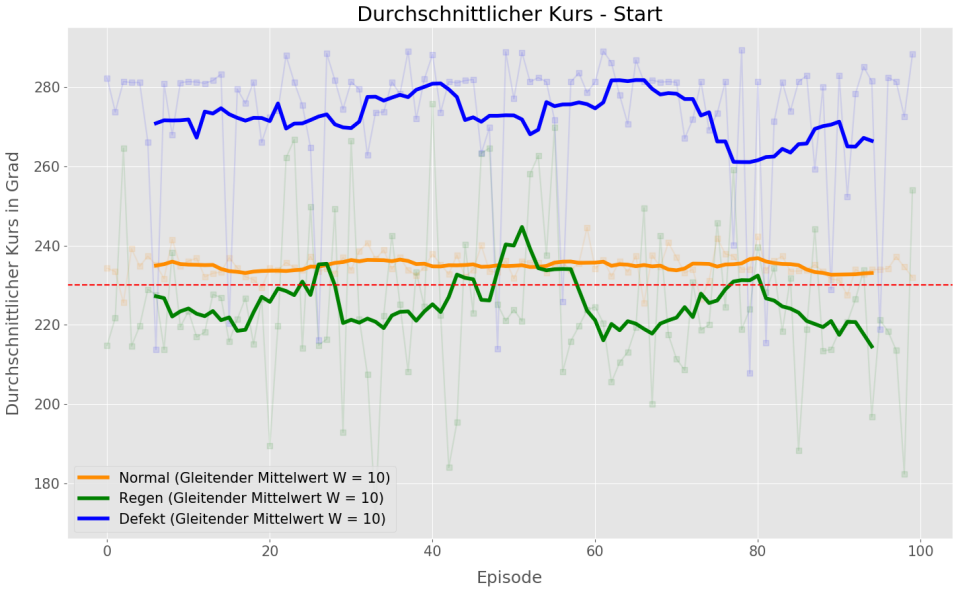
- Unter **adversen Bedingungen** fällt auf, dass das Signal stark streut. Mit einer durschnittlichen Höhe von **0.396** und einer Standardabweichung von **0.0962 (24.29 %)** hat der Agent größere Schwierigkeiten das Flugzeug abzuheben. In 7 von 100 Testdurchläufen **(7 %)** stürzt der Agent sogar ab. Nichtsdestotrotz ist dies für die extremen Wetterbedingungen immer noch ein beachtliches Ergebnis.

  Beim durschnittlichen Kurs von **225.17** Grad fällt ebenfalls auf, dass der Agent im Schnitt zwar nah an den Referenzwert von 230 Grad kommt, letztendlich aber doch eine wesentlich höhere Streuung von **19.92 (8.84 %)** Grad als unter normalen Bedingungen aufweist.

- Bei einem aufretenden **Defekt** erzielt der Flugzeugautopilot eine fast identische durchschnittliche Höhe **0.431** wie unter normalen Bedingungen. Ebenfalls stürzt der Agent kein einziges mal ab. Interessant ist allerdings der erhöhte durchschnittliche Kurs von **272.70** Grad. Da der Flugzeugautopilot den ausfallenden linken Flügel kompensieren muss versucht dieser das Flugzeug möglichst weit nach rechts zu steuern. Deshalb liegt der durchschnittliche Kurs bei 272.70 Grad rechts vom Referenzkurs 230.0 Grad.



(a)



(b)

Abbildung 4.15.: Durschnittlicher Kurs und Höhe - Start (Eigene Erstellung)

Höhe - Start	**Durchschnitt (1000 Ft normiert)**	**Standardabweichung**
Normale Bedingungen	0.493	0.0057 (1.15 %)
Regen	0.396	0.0962 (24.29 %)
Defekt (Linker Aileron)	0.431	0.0058 (1.34 %)

Tabelle 4.6.: Auswertung der Höhe - Start

Kurs - Start	**Durchschnitt (Grad)**	**Standardabweichung**
Normale Bedingungen	234.78	3.16 (1.34%)
Regen	225.17	19.92 (8.84 %)
Defekt (Linker Aileron)	272.70	19.06 (6.99 %)

Tabelle 4.7.: Auswertung des Kurses - Start

Absturz - Start	**Erfolgsrate (über 100 Episoden)**
Normale Bedingungen	100%
Regen	93%
Defekt (Linker Aileron)	100%

Tabelle 4.8.: Absturzrate - Start

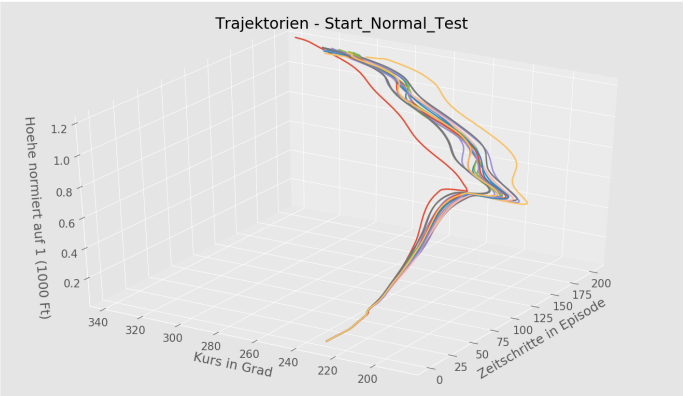
Analyse der Trajektorien

Im Folgenden werden die Flugtrajektorien jedes Unterszenarios analysiert. In den Abbildungen 4.16a, 4.16b und 4.16c sind die Aufzeichnungen der Trajektorien für jeweils 10 beliebige ausgewählte Episoden aus den 100 durchgeführten Testepisoden. Dabei soll ein Eindruck über das Flugverhalten des Flugzeugautopilots entstehen.

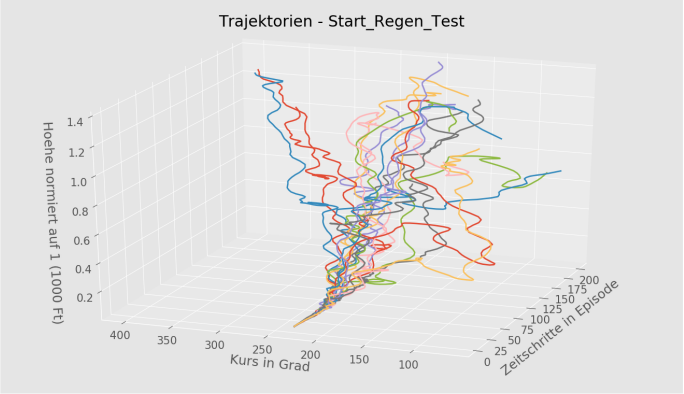
- Abbildung 4.16a zeigt einen nahezu perfekten Start bis zum 125. Zeitschritt. Bis zum Ende gewinnt der Agent weiter an Höhe kommt jedoch vom Referenzkurs von 230 Grad ab.
- In Abbildung 4.16b ist der turbulente Aufstieg bei adversen Wetterbedingungen zu sehen. Allgemein bleibt das Flugzeug in diesem Fall in den meisten Fällen näher am Referenzkurs, wobei die eingetliche Flugbahn sehr turbulent ist.

- Beim Auftreten des Defekts ab Höhe 0.150 ist in Abbildung 4.16c zu sehen, dass wie beim Start der Aufstieg sehr kontrolliert vorgeht. Der Flugzeugautopilot scheint allerdings beim Korrigieren des ausfallenden linken Flügels etwas überzukompensieren.

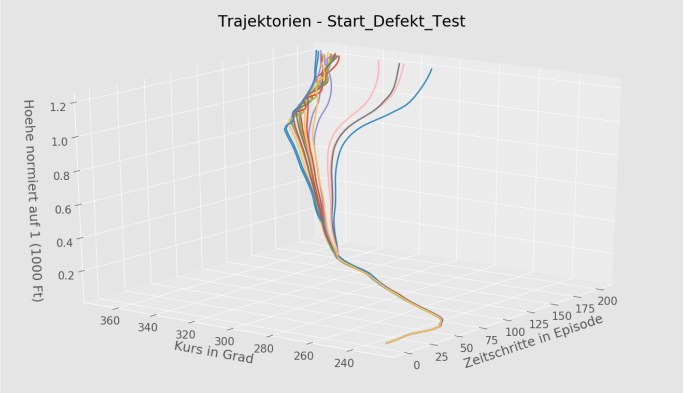
Grundsätzlich ist der Autopilot in der Lage das Flugzeug in allen 3 Szenarien erfolgreich abzuheben.



(a)



(b)



(c)

Abbildung 4.16.: Trajektorienaufzeichnung für die 3 Unterszenarien - Start (Eigene Erstellung)

Kreuzfahrtflug

Für den Kreuzfahrtflug werden ebenfalls die Testergebnisse für alle 3 Unterszenarien präsentiert. Wie beim Start werden zunächst die Bewertungskriterien behandelt und anschließend eine qualitative Einschätzung des Flugverhaltens abgegeben.

Auswertung der notwendigen und hinreichenden Bedingungen

Es werden die **durschnittlich erreichte Höhe** in Abbildung 4.17a und der **durschnittliche Kurs** in Abbildung 4.17b betrachtet. Die dazugehörigen Mittelwerte und Standardabweichungen sind jeweils in den Tabellen 4.9 und 4.10 zu sehen. Die durchschnittliche Höhe ist auf 1 von einer Höhe von 2100 Fuß normiert. Die Darstellung des Kurses und der Erfolgsrate bleiben vom Start unverändert. Auch hier werden alle 3 Unterszenarien miteinander verglichen:

- Unter **normalen Bedingungen** hält der Flugzeugautopilot die ursprüngliche Höhe in allen 100 Testfällen. Dabei beträgt die durchschnittliche normierte Höhe **1.009** mit einer Standardabweichung von **0.0013**. Dabei kann aus Tabelle 4.27 eine Erfolgsrate von **100%** entnommen werden. Das Flugzeug ist also während keines der Episoden abgestürzt. Die notwendigen Bedingungen sind somit auch beim Kreuzfahrtflug unter normalen Bedingungen erfüllt.

  Betrachtet man den durchschnittlichen Kurs, so fällt auf, dass dieser ebenfalls eine sehr niedrige Standardabweichung von **0.27%** besitzt, allerdings der durchschnittliche Kurs 263.81 Grad beträgt. Der Flugzeugautopilot versuchte das Flugzeug gerade zu halten hatte dabei aber einen leichten Bias nach rechts. Nichtsdestotrotz wird die hinreichende Bedingung einer Kursabweichung von ± 50 Grad erfüllt.

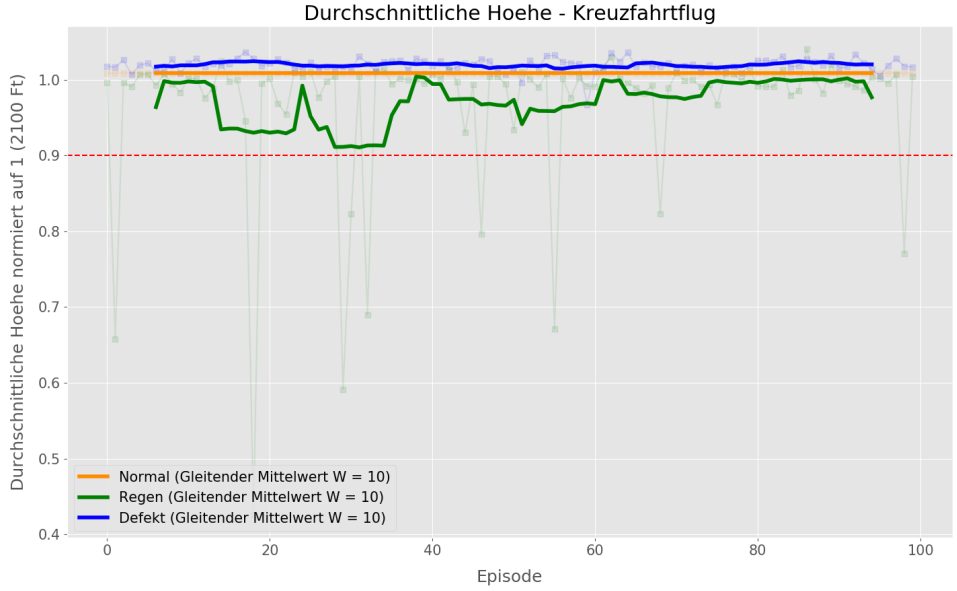
- **Adverse Wetterbedingungen** beeinträchtigen die Performance beim Kreuzfahrtflug ebenfalls. Zwar ist die durchschnittliche Höhe mit **0.97** nur geringfügig niedriger als unter normalen Bedingungen, allerdings ist die Standardabweichung mit **0.095 (9.80%)**, um einiges höher. In einigen Episoden fällt der Agent auch unter die 0.9 Marke, wodurch dieser nur eine Erfolgsrate von **91%** verzeichnen kann. Nichtsdestotrotz, bedeutet dies, dass von 100 Testflügen lediglich 9 abgestürzt sind. Wie beim Start führt die Stochastizität, die durch die auftretenden Turbulenzen erzeugt wird für die stärker streuenden Daten und die somit reduzierte Performance.

  Bei Betrachtung des durchschnittlichen Kurses fällt auf, dass dieser viel näher am Referenzwert von 230 Grad liegt. Nichtsdestotrotz ist die Standardabweichung mit **28.19**

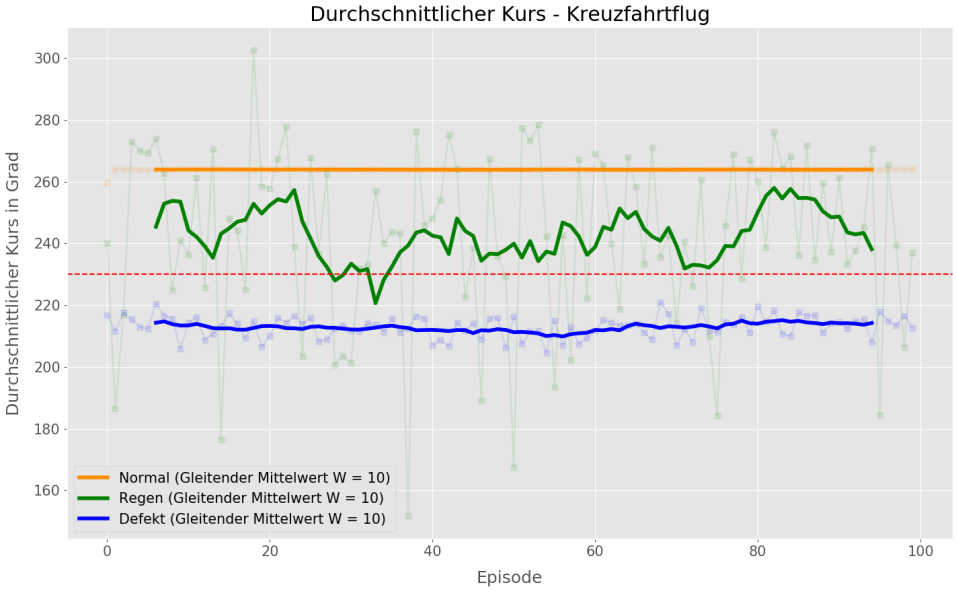
(11.64%) sehr hoch. Man kann annehmen, dass in einigen Fällen der Agent durch die Turbulenzen näher an den Referenzwert kommt aber in anderen sich auch sehr weit von diesem entfernt.

- Ein auftretender **Defekt** kompensiert das trainierte Modell mit Erfolg. Die durchschnittliche Höhe ist mit **1.01**, der durschnittlichen Höhe unter normalen Bedingungen identisch. Lediglich die Standardabweichung ist mit **0.008 (0.79%)** etwas höher. Dies kann dem Kompensieren des ausfallenden linken Flügels zugeschrieben werden. Auch wie unter normalen Bedingungen stürzt das Flugzeug bei keiner der 100 Episoden ab.

  Die durchschnittliche Kursabweichung lässt auch auf den auftretenden Defekt schließen. Diese liegt bei **213.01 Grad** links vom Referenzwert von 230 Grad was offensichtlich durch den beeinträchtigten linken Flügel verursacht wird. Dabei hält der Autopilot diesen Kurs mit einer Standardabweichung von **3.53 (1.65%)** stabil.



(a)



(b)

Abbildung 4.17.: Durschnittlicher Kurs und Höhe - Kreuzfahrtfug (Eigene Erstellung)

Höhe - Kreuzfahrtflug	**Durchschnitt (1000 Ft normiert)**	**Standardabweichung**
Normale Bedingungen	1.009	0.0013 (0.12%)
Regen	0.97	0.095 (9.80%)
Defekt (Linker Aileron)	1.01	0.008 (0.79%)

Tabelle 4.9.: Auswertung der Höhe - Kreuzfahrtflug

Kurs - Kreuzfahrtflug	**Durchschnitt (Grad)**	**Standardabweichung**
Normale Bedingungen	263.81	0.72 (0.27%)
Regen	242.14	28.19 (11.64%)
Defekt (Linker Aileron)	213.01	3.53 (1.65%)

Tabelle 4.10.: Auswertung des Kurses - Kreuzfahrtflug

Absturz - Kreuzfahrtflug	**Erfolgsrate (über 100 Episoden)**
Normale Bedingungen	100%
Regen	91%
Defekt (Linker Aileron)	100%

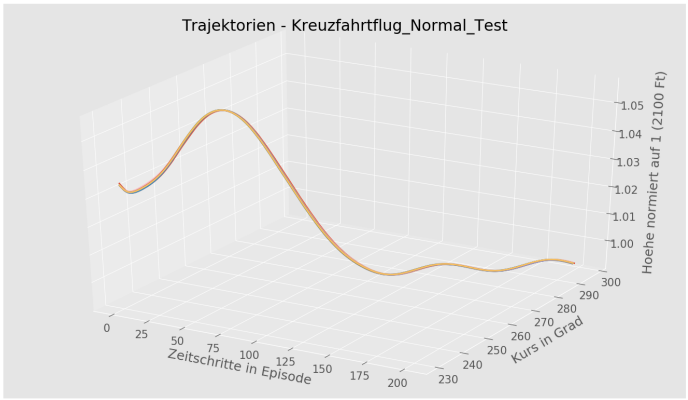
Tabelle 4.11.: Absturzrate - Kreuzfahrtflug

Analyse der Trajektorien

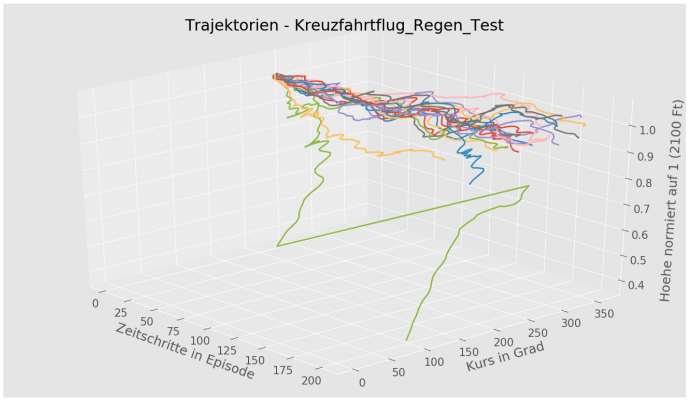
In den Abbildungen 4.18a, 4.18b und 4.18c sind die Aufzeichnungen für jeweils 10 beliebige ausgewählte Trajektorien aus den 100 durchgeführten Testepisoden. In Abbildung 4.18a ist die niedrige Varianz des unter normalen Bedingungen trainierten Modells zu sehen. Alle 10 Trajektorien decken sich nahezu vollständig ab. Ebenso kann die Tendenz des Flugzeuges nach rechts zu schwenken beobachtet werden.

Abbildung 4.18b zeigt die hohe Varianz, die durch die adversen Wetterbedingungen erzeugt wird. Grundsätzlich wird das Flugzeug durch die Turbulenzen hin und her bewegt was zu recht unterschiedlichen Trajektorien führt.

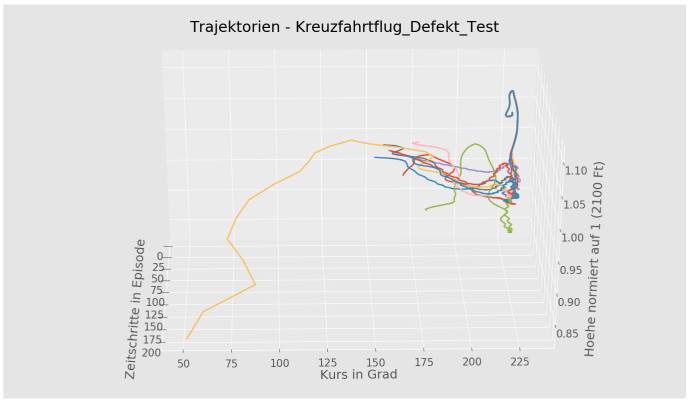
In Abblidung 4.18c wir der Einfluss des Defekts offensichtlicht. Dieser tritt ab dem 150. Zeitschritt der Episode auf. Es ist zu beobachten, dass davor die Trajektorien sehr dem unter normalen Bedingungen trainierten Modell ähneln. Bei Auftreten des Defekts schwenkt das Flugzeug nach links was vom Agenten versucht wird zu kompensieren.



(a)



(b)



(c)

Abbildung 4.18.: Trajektorienaufzeichnung für die 3 Unterszenarien (Eigene Erstellung)

Landung

Bei der Landung werden im Vergleich zu den beiden anderen Szenarien andere Werte untersucht. Vor allem ist von Interesse, ob der Flugzeugautopilot in der Lage gewesen ist das Flugzeug präzise in die Nähe der Landebahn zu steuern. Hierzu werden für die Landung direkt die aufgezeichneten Trajektorien betrachtet, da Durschnittswerte keinen zuverlässigen Indiz auf die Lage des Flugzeugs auf der Landebahn geben. Zudem wird die Erfolgsrate des Flugzeugautopiloten untersucht. Die einzelnen Trajektorien sind farbkodiert. Dabei sind blau markierte Trajektorien erfolgreich ohne Absturz verlaufen und rote Trajektorien Verläufe in denen der Agent abgestürzt ist. Abschließend werden die Anzahl der erfolgreichen Landungen aufgezählt die sich inneralb des definierten Bereiches von $\pm$ **0.12** Mittellinienabweichung befinden.

Unter normalen Bedingungen in Abbildung 4.19 gelingt das Landen ausgeprochen gut. Alle Landungen befinden sich im Bereich von **0.10** bis **0.14** Mittellinienabweichung. Die Landebahn wird bis zu $\pm$ **0.12** getroffen, wodurch der Agent durchgehend am linken Rand der Landebahn gelandet ist und somit nur in **50%** direkt auf die Landebahn gekommen ist. Nichtsdestotrotz, befinden sich alle Landungen sehr nah an der Landebahn. Beachtlich ist in diesem Fall auch die Erfolgsrate ohne Absturz in Tabelle 4.13 von **100%**.

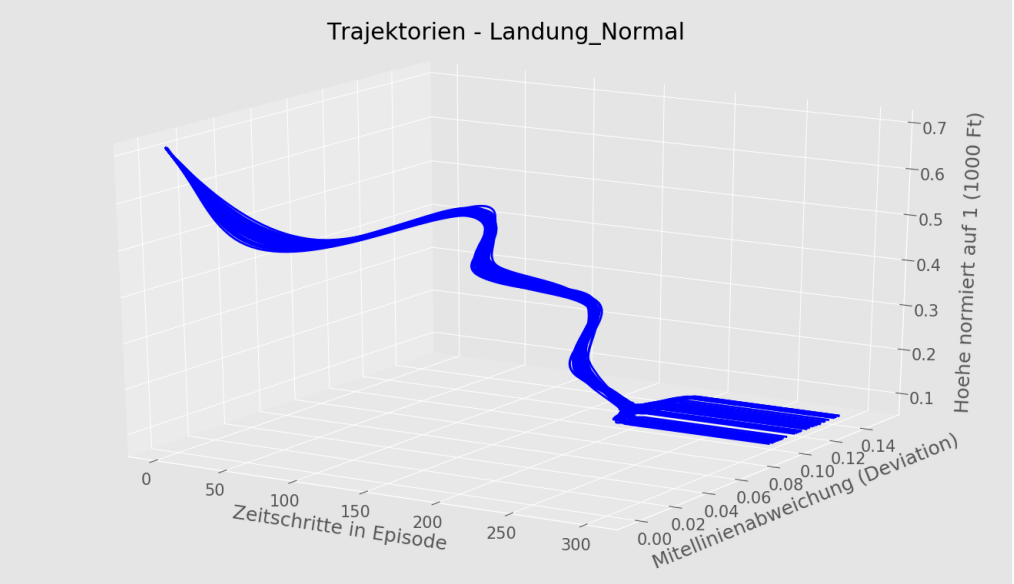


Abbildung 4.19.: Trajektorien der Landung unter normalen Bedingungen

Selbst die eher milden Wetterveränderungen unter **adversen Wetterbedingungen** in Abbildung 4.20 führen bei der Landung bereits zu einer starken Beeinträchtigung der Performance. Zum einen beträgt die Erfolgsrate nur noch **43%** und zum anderen landet der Agent bei einigen erfolgreichen Trajektorien weit neben der Landebahn.

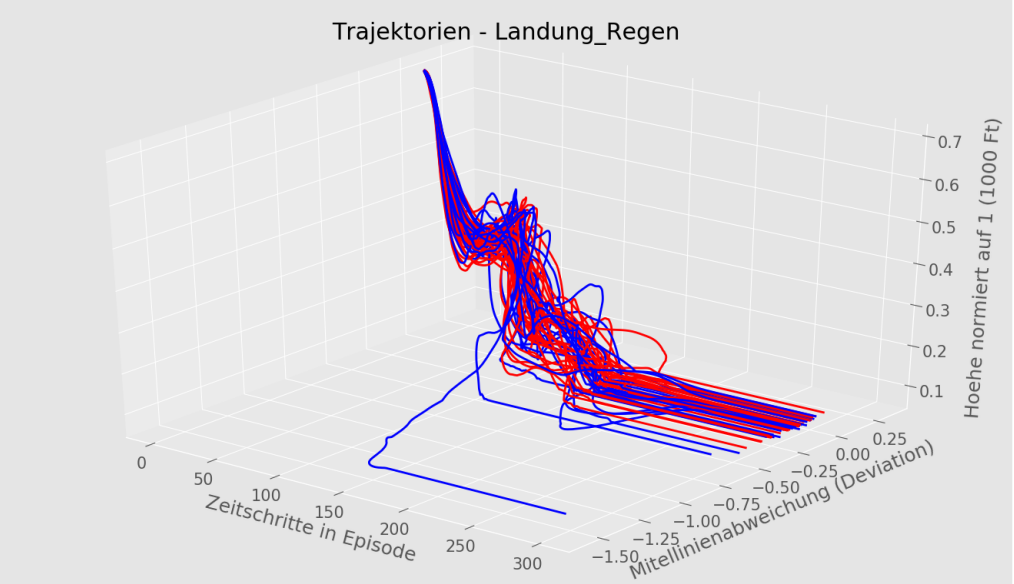


Abbildung 4.20.: Trajektorien der Landung unter adversen Wetterbedingungen

Wesentlich besser kann der Flugzeugautopilot den Ausfall des linken Flügels in Abbildung 4.21 kompensieren. Wie auch bei den anderen Szenarien fällt auf, dass der Agent versucht das Flugzeug weit nach rechts zu steuern, um den Ausfall zu kompensieren. In diesem Fall ist das landen auf der Landebahn sogar präziser als unter normalen Bedingungen. Nichtsdestotrotz schafft der Agent dies nicht immer fehlerfrei wodurch nur **72%** der Testfälle erfolgreich sind.

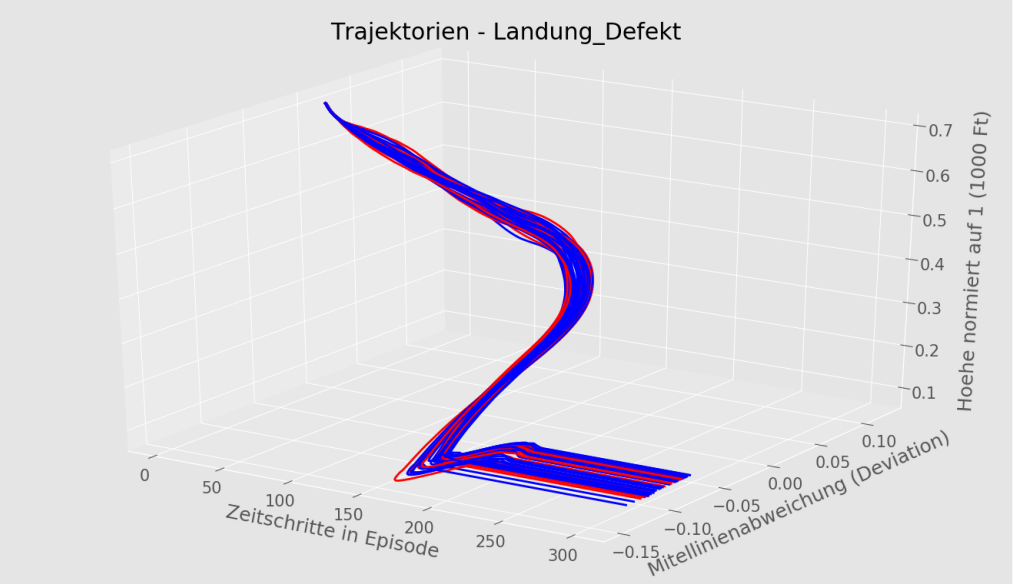


Abbildung 4.21.: Trajektorien der Landung bei defektem linken Flügel

Absturz - Landung	**Erfolgsrate (über 100 Episoden)**
Normale Bedingungen	100%
Regen	43%
Defekt (Linker Aileron)	72%

Tabelle 4.12.: Erfolgsrate ohne Absturz

Auf Landebahn - Landung	**Erfolgsrate (über 100 Episoden)**
Normale Bedingungen	50%
Regen	27%
Defekt (Linker Aileron)	39%

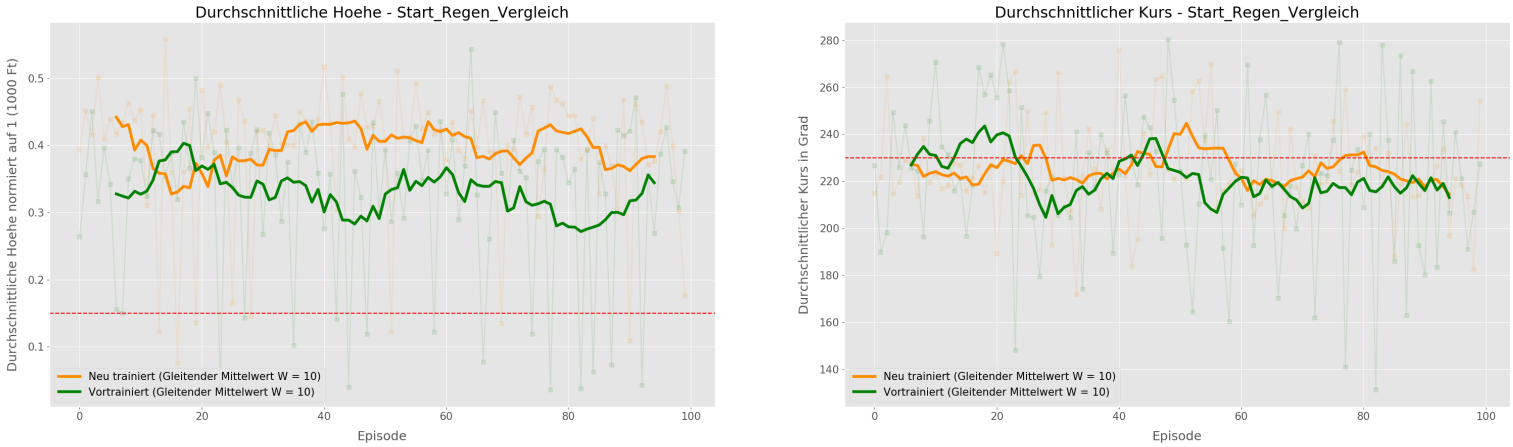
Tabelle 4.13.: Landen auf Landebahn ohne Absturz

4.3.2. Analyse der Generalisierungsfähigkeit

Im Folgenden wird die Übertragungsfähigkeit der unter normalen Bedingungen trainierten Modelle auf die 2 anderen Unterszenarien untersucht. Dabei werden die Ergebnisse mit den Ergebnissen der bereits präsentierten neu trainierten Modelle präsentiert. Wie im vorherigen Kapitel werden die Ergebnisse für alle 3 Flugszenarien aufgeführt.

Start

Die Abbildungen 4.22a, 4.22b und Tabellen 4.14, 4.15 zeigen eine schlechtere Performance des vortainierten Modells unter **adversen Wetterbedingungen**. Die durschnittliche Höhe ist um **-0.067** im Vergleich zum neu trainiertem Modell gesunken. Die Standardabweichung ist mit **+0.0213** ebenfalls größer geworden. Der durschnittliche Kurs zeigt zudem, dass der Flugzeugautopilot mit dem vortrainierten Modell instabiler als mit dem neu trainierten Modell agiert. Die Standardabweichung ist hier um **+12.14** Grad gestiegen.



(a) (b)

Abbildung 4.22.: Durschnittlicher Kurs und Höhe - Regen - Start -Neu trainiert und Vortrainiert (Eigene Erstellung)

Durchschnittliche Höhe - Regen - Start	**Durchschnitt (1000 Ft normiert)**	**Standardabweichung (in Prozent)**
Neu trainiert	0.396	0.0962 (24.29 %)
Vortrainiert	0.329	0.1175 (35.71 %)
Differenz	**−0.067**	**+0.0213**

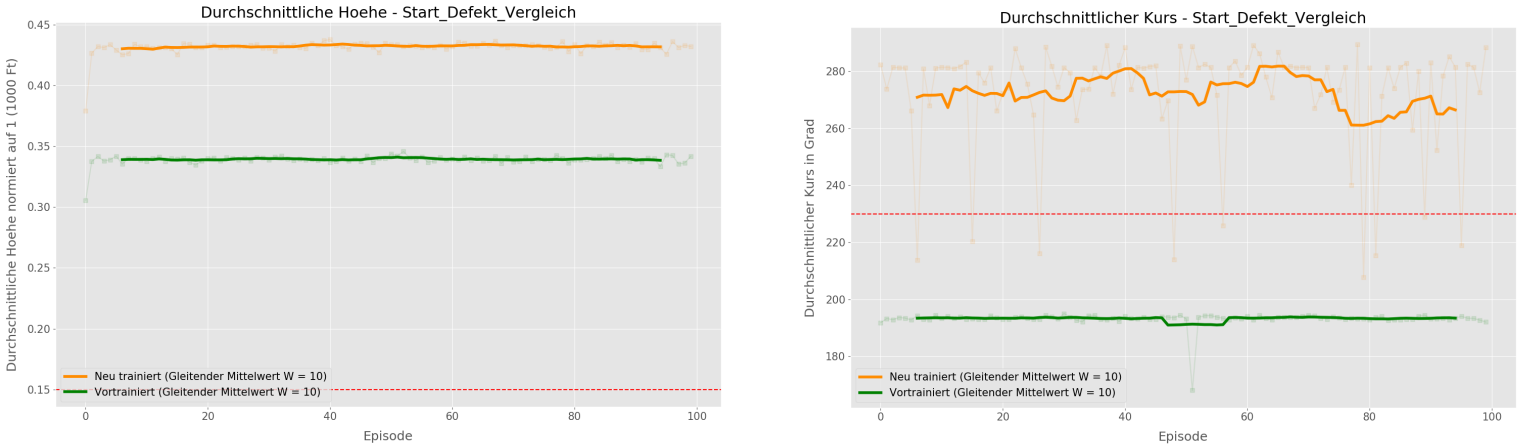
Tabelle 4.14.: Auswertung der Höhe - Regen_Vortrainiert

Durchschnittlicher Kurs - Regen - Start	**Durchschnitt (Grad)**	**Standardabweichung (in Prozent)**
Neu trainiert	225.17	19.92 (8.84 %)
Vortrainiert	221.47	32.06 (14,47 %)
Differenz	**−3.7**	**+12.14**

Tabelle 4.15.: Auswertung des Kurses - Regen_Vortrainiert

Die Unterschiede beider Modelle sind bei einem auftretenden **Defekt** noch größer in Abbildungen **??** und 4.23b. Mit **-0.092** ist die durschnittliche Höhe beim vortrainerten Modell um

fast 100 Meter geringer. Die Standardabweichung ist mit einer Differenz von **-0.0019** lediglich geringfügig niedriger. Betrachtet man den durchschnittlichen Kurs so fällt direkt auf, dass das vortrainierte Modell mit einem durschnittlichen Kurs von **193.17** Grad kaum in der Lage ist den ausfallenden linken Flügel zu kompensieren. Dies erklärt zum Teil auch die deutlich niedrigere erreichte durchschnittliche Höhe. Eine konstante ausgeprägte Schräglage führt in diesem Fall zu einer wesentlich geringeren Standardabweichung von **2.61** Grad.



(a) (b)

Abbildung 4.23.: Durschnittlicher Kurs und Höhe - Defekt - Start - Neu trainiert und Vortrainiert (Eigene Erstellung)

Durchschnittliche Höhe - Defekt - Start	**Durchschnitt (1000 Ft normiert)**	**Standardabweichung**
Neu trainiert	0.431	0.0058 (1.34 %)
Vortrainiert	0.339	0.0039 (1.15 %)
Differenz	**−0.092**	**−0.0019**

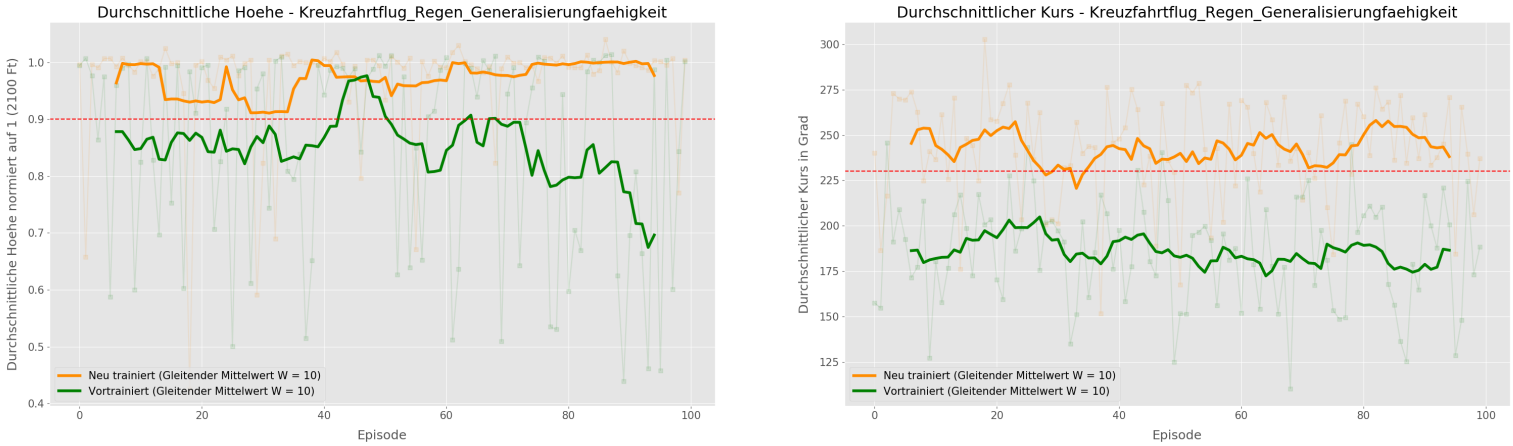
Tabelle 4.16.: Auswertung der Höhe - Regen_Vortrainiert

Durchschnittlicher Kurs - Defekt - Start	**Durchschnitt (Grad)**	**Standardabweichung**
Neu trainiert	272.70	19.06 (6.99 %)
Vortrainiert	193.17	2.61 (1.35 %)
Differenz	**−79.53**	**−16.45**

Tabelle 4.17.: Auswertung des Kurses - Regen_Vortrainiert

Kreuzfahrtflug

Bei **adversen Wetterbedingungen** kann das vortrainierte Modell die Mindestmarke der durchschnittlichen Höhe von 0.9 nicht mehr erreichen. Zudem ist die Standardabweichung um **+0.0815** wesentlich gestiegen. Allgemein ist das Flugzeug mit einem durchschnittlichen Kurs von **185.82** Grad auch wesentlich weiter vom Referenzkurs entfernt.



(a) (b)

Abbildung 4.24.: Durschnittlicher Kurs und Höhe - Regen - Kreuzfahrtflug - Neu trainiert und Vortrainiert (Eigene Erstellung)

Durchschnittliche Höhe - Regen	**Durchschnitt (2100 Ft normiert)**	**Standardabweichung**
Neu trainiert	0.97	0.095 (9.80%)
Vortrainiert	0.85	0.1765 (20.8%)
Differenz	**−0.12**	**+0,0815**

Tabelle 4.18.: Auswertung der Höhe - Kreuzfahrtflug_Regen_Vortrainiert

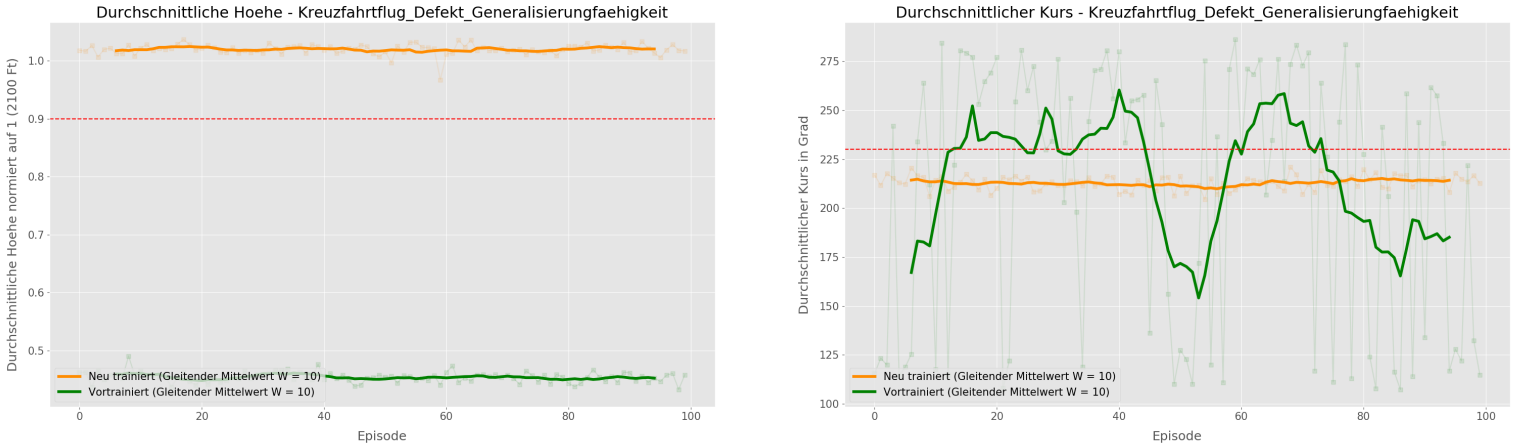
Durchschnittlicher Kurs - Regen	**Durchschnitt (Grad)**	**Standardabweichung**
Neu trainiert	242.14	28.19 (11.64%)
Vortrainiert	185.82	29.37 (15.8%)
Differenz	**−56.32**	**+1.18**

Tabelle 4.19.: Auswertung des Kurses - Kreuzfahrtflug_Regen_Vortrainiert

Völlig unbrauchbar ist hingegen das vortrainerte Modell bei Auftreten eines **Defekts** in Abbildungen 4.25a und 4.25b. Hier stürzt das Flugzeug in 100% der Testfälle ab, weshalb die niedrige durschnittliche Höhe von **0.45** und die hohe Standardabweichung beim durchschnittlichen Kurs nicht verwunderlich sind. Der Agent stürzt hier vermutlich abwechselnd rechts oder links vom Referenzkurs ab.

Durchschnittliche Höhe - Defekt	**Durchschnitt (2100 Ft normiert)**	**Standardabweichung**
Neu trainiert	1.01	0.008 (0.79%)
Vortrainiert	0.45	0.009 (2%)
Differenz	**−0.56**	**+0.001**

Tabelle 4.20.: Auswertung der Höhe - Kreuzfahrtflug_Defekt_Vortrainiert



(a) (b)

Abbildung 4.25.: Durschnittlicher Kurs und Höhe - Defekt - Kreuzfahrtflug - Neu trainiert und Vortrainiert (Eigene Erstellung)

Durchschnittliche Kurs - Defekt	**Durchschnitt (Grad)**	**Standardabweichung**
Neu trainiert	213.01	3.53 (1.65%)
Vortrainiert	208.98	66.52 (31.83%)
Differenz	**−4.03**	**+62.99**

Tabelle 4.21.: Auswertung des Kurses - Kreuzfahrtflug_Defekt_Vortrainiert

Landung

Da das Landen mit den neu trainierten Modellen unter adversen Wetterbedingungen schon schwierig war ist das Landen mit einem vortrainertem Model in Abbildung 4.26 , wie aus den vorherigen Szenarien zu erwarten war deutlich schlechter. Der Agent schafft lediglich in **21%** der Testfällen eine erfolgreiche Landung. Dabei nur in **7%** der Testdurchläufe eine Landung auf der Landebahn im Bereich ± 0.12 der Mittellinienabweichung.

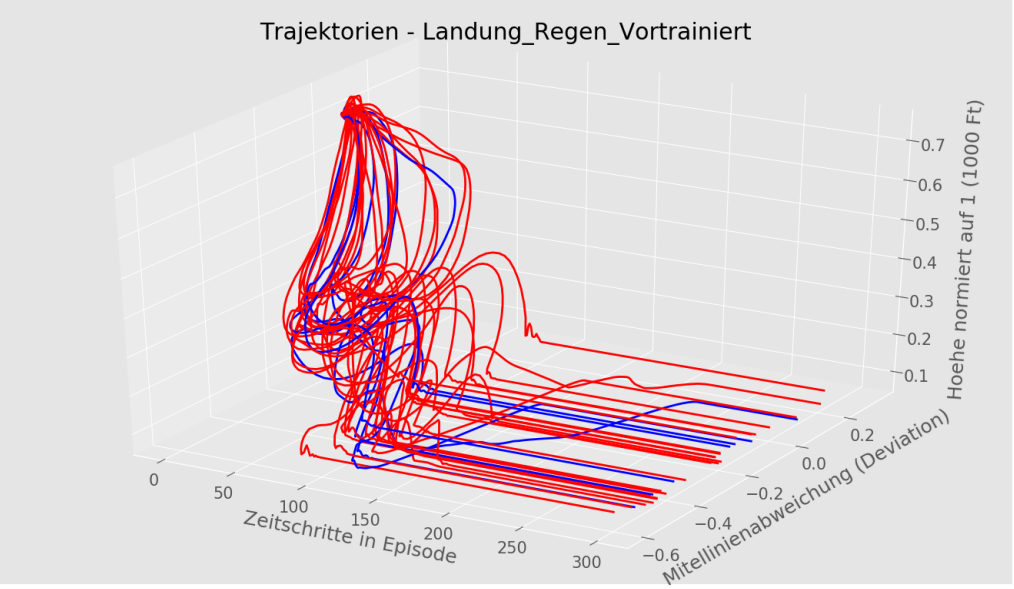


Abbildung 4.26.: Trajektorien der Landung bei adversen Wetterbedingungen mit vortrainiertem Modell (Eigene Erstellung)

Absturz - Landung - Regen	**Erfolgsrate (über 100 Episoden)**
Neu trainiert	43%
Vortrainiert	21%

Tabelle 4.22.: Absturzrate - Landung - Vortrainiert

Landen auf Landebahn - Landung - Regen	**Erfolgsrate (über 100 Episoden)**
Neu trainiert	27%
Vortrainiert	4%

Tabelle 4.23.: Erfolgreiches Landen auf Landebahn - Landung - Vortrainiert

Wiederum wie beim Kreuzfahrtflug ist das vortrainierte Modell bei einem **auftretendem Defekt** völlig unbrauchbar. So schafft es der Flugzeugautopilot mit dem vortrainierten Modell das Flugzeug kein einziges mal zu Landen.

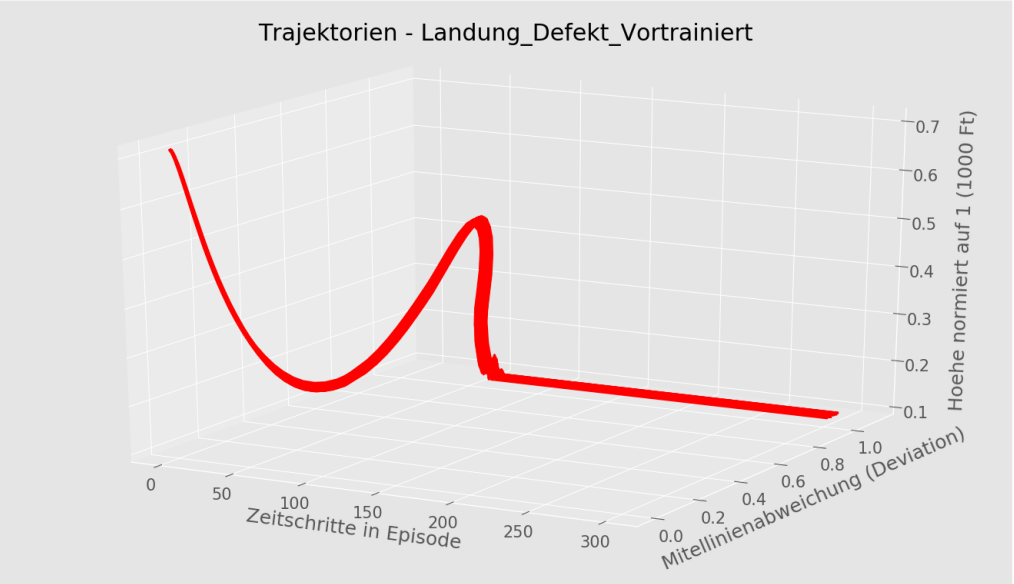


Abbildung 4.27.: Trajektorien der Landung bei defektem linken Flügel mit vortrainiertem Modell (Eigene Erstellung)

Absturz - Landung - Defekt	**Erfolgsrate (über 100 Episoden)**
Neu trainiert	73%
Vortrainiert	0%

Tabelle 4.24.: Absturzrate - Landung - Vortrainiert

Landen auf Landebahn - Landung - Defekt	**Erfolgsrate (über 100 Episoden)**
Neu trainiert	39%
Vortrainiert	0%

Tabelle 4.25.: Erfolgreiches Landen auf Landebahn - Landung - Vortrainiert

4.3.3. Analyse der Trainingskonvergenz

In diesem Abschnitt wird die Konvergenz des Trainingsfortschritts untersucht. Dabei werden folgende Metriken verwendet:

- **Gesamtbelohnung pro Episode**.
- **Durchschnittlicher Action-Value $Q(s, a)$ pro Episode**.

Dies wird für alle 3 Szenarien des Starts, Kreufahrtflugs und Landung untersucht.

Start

Anhand des Trainingsverlaufs in Abbildung **??** ist eine deutliche Konvergenz des Action-Values $Q(s, a)$ festzustellen. Bis zur 2000. Episode steig dieser weiter. Interessant ist allerdings zu beobachten, dass die Gesamtbelohnung viel früher bei rund 1500 Episoden ihren Höchstwert erreicht.

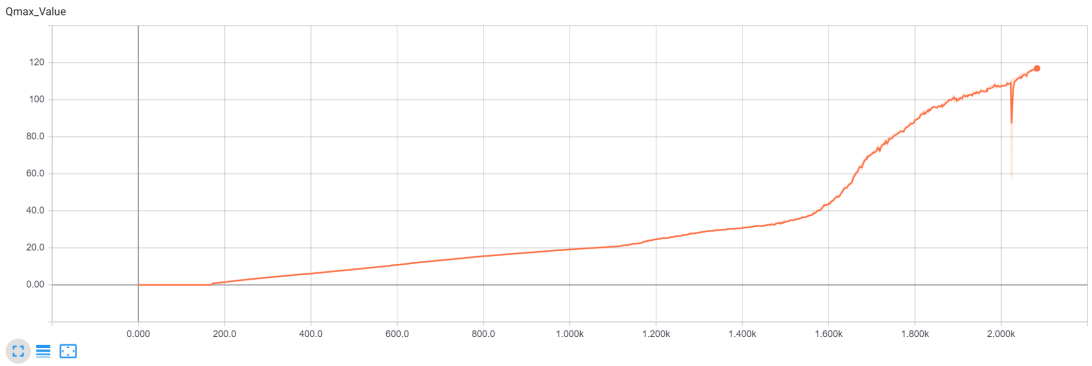


Abbildung 4.28.: Durchschnittlicher Action-Value $Q(s, a)$ pro Episode

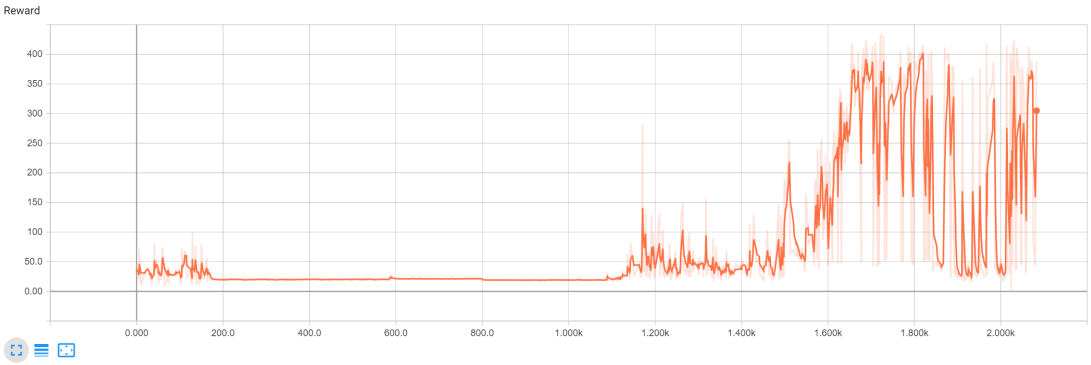


Abbildung 4.29.: Gesamtbelohnung pro Episode

Kreuzfahrtflug

Auch beim Kreufahrtflug ist in Abbildung 4.30 eine Konvergenz zu erkennen. Ebenfalls scheint die Gesamtbelohnung pro Episode positiv mit dem Wert des Action-Values $Q(s, a)$ zu korrellieren. Die zwischenzeitliche Lücken können hierbei ignoriert werden. Hier handelt es sich lediglich, um das Abspeichern der Modelle in Tensorflow, welches eine gewisse Zeit in Anspruch nimmt und somit Episoden im Leerlauf erzeugt.

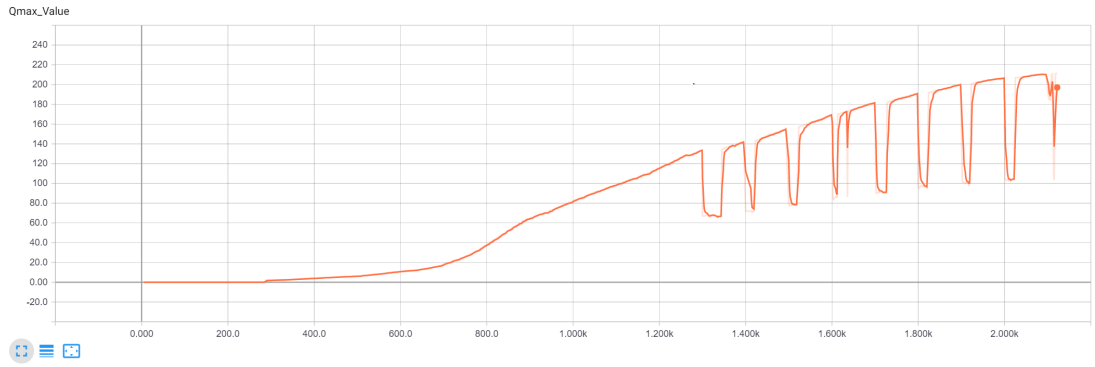


Abbildung 4.30.: Durchschnittlicher Action-Value $Q(s, a)$ pro Episode

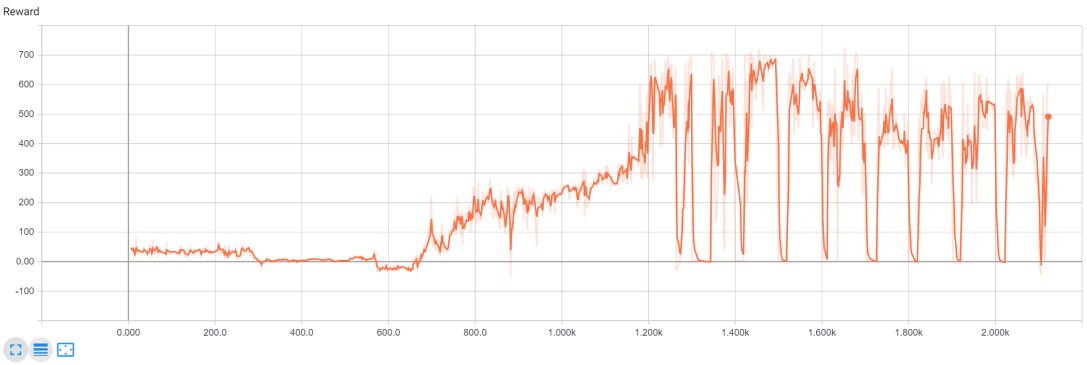


Abbildung 4.31.: Gesamtbelohnung pro Episode

Landung

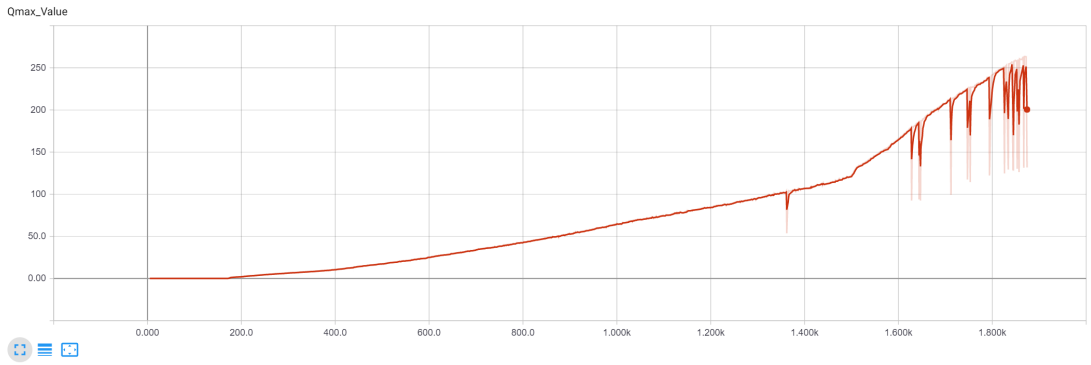


Abbildung 4.32.: Durchschnittlicher Action-Value $Q(s, a)$ pro Episode

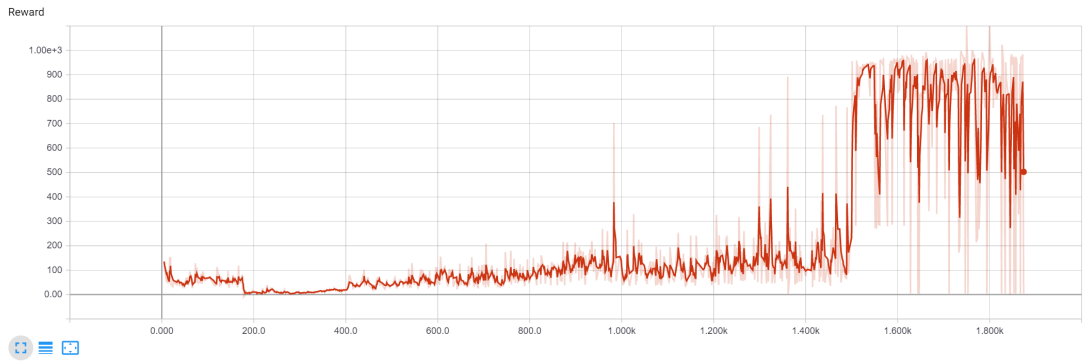


Abbildung 4.33.: Gesamtbelohnung pro Episode

Grundsätzlich kann aus den untersuchten Trainingsverläufen geschlossen werden, dass eine Form von Konvergenz in allen 3 Szenarien existiert. Darüber hinaus scheinen der Wert des Action-Values $Q(s, a)$ und der Gesamtbelohnung pro Episode positiv zu korrelieren. Im Allgemeinen ist dies wenig überraschend, da ein hoher Action-Value $Q(s, a)$ eine Approximierung der Belohnung in einem Markov Decision Process ist und somit auch zwangsläufig eine hohe Gesamtbelohnung resultieren muss.

4.3.4. Spezialfälle

Im Folgenden werden die in Kapitel 4.2.2 aufgeführten Testfälle analysiert. Es wurden bei der Landung unter normalen Bedingungen 3 Hauptbestandteile des Trainings untersucht:

1. Die Performance des Flugzeugautopiloten wird untersucht wenn keine Erforschung der Umgebung am Anfang des Trainings durchgeführt wird.
2. Der Experience Replay Buffer wird durch das Trainieren mit der zuletzt ausgeführten Aktionen ersetzt.
3. Der Deviation Contstraint wird bei der Landung erheblich aufgeschwächt.

Es werden 3 Testdurchläufe mit 100 Episoden durchgeführt. Dabei kommen wenn nicht wie oben anders angegeben die Erforschungsstrategie mit dem Ornstein-Uhlenbeck-Prozess, der Experience Replay Buffer und die Constraints zum Einsatz.

(1) Training ohne Erforschung der Umgebung

Abbildung 4.34 offenbart die Lernschwierigkeiten des Flugzeugautopiloten wenn dieser die Umgebung vorher nicht ausreichend erforscht hat. Wobei der Agent mit ausreichender Erforschung der Umgebung in allen 100 Testfällen erfolgreich landet ergeben sich beim Pendant ohne Erforschung lediglich 6 erfolgreiche Landungen. Von den 6 erfolgreichen Landungen liegt lediglich eine Landung im Bereich von $\pm$ 0.12 Mittellinienabweichung.

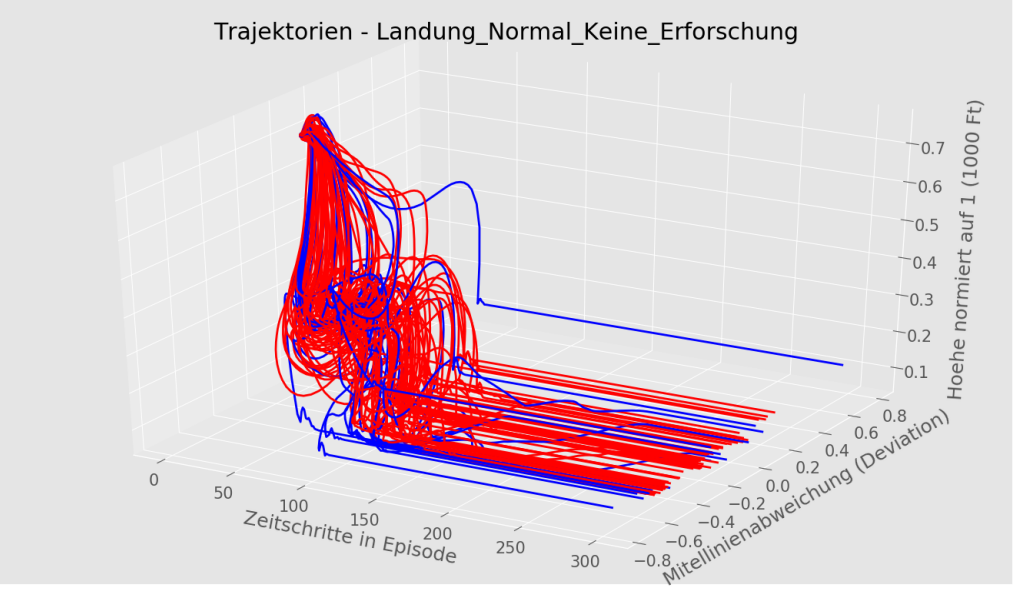


Abbildung 4.34.: Trajektorien der Landung ohne Erforschung (Eigene Erstellung)

Absturz - Landung - Keine Erforschung	**Erfolgsrate (über 100 Episoden)**
Mit Erforschung	100%
Ohne Erforschung	6%

Tabelle 4.26.: Erfolgreiches Landen auf Landebahn - Landung - Keine Erforschung

Landen auf Landebahn - Landung - Keine Erforschung	**Erfolgsrate (über 100 Episoden)**
Mit Erforschung	50%
Mit Erforschung	1%

Tabelle 4.27.: Erfolgreiches Landen auf Landebahn - Landung - Keine Erforschung

(2) Training ohne Experience Replay Buffer

Abbildung 4.35 zeigt die Performance des Agenten ohne Experience Replay Buffer. Wie zu erwarten war ist der Agent nicht in der Lage eine koherente Policy zu erlernen. Alle 100 Testfälle resultieren in einen Absturz:

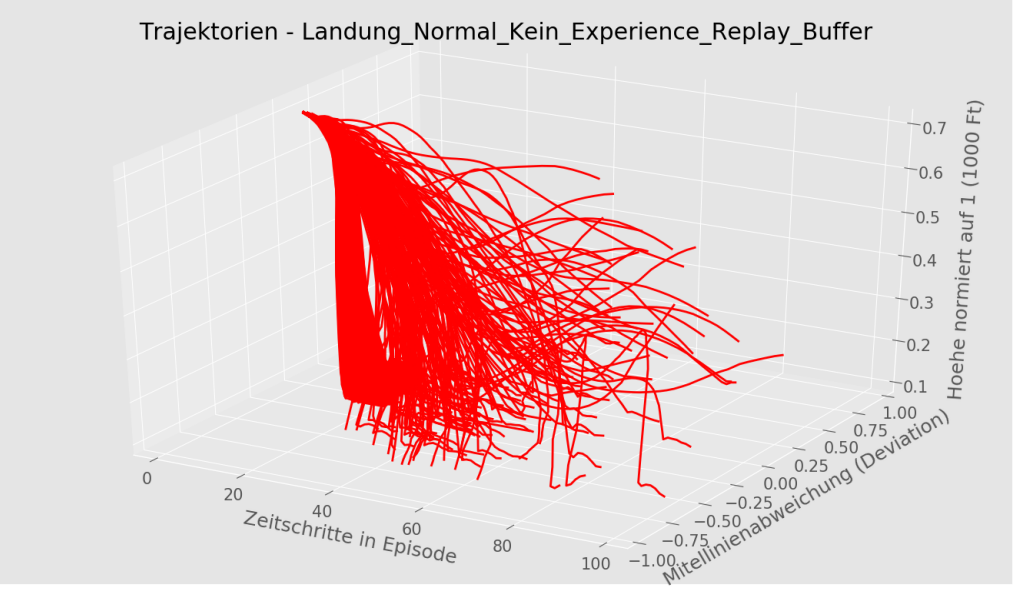


Abbildung 4.35.: Trajektorien der Landung ohne Experience Replay Buffer (Eigene Erstellung)

Betrachtet man den durchschnittlichen Action-Value $Q(s, a)$ in Abbildung 4.36 pro Episode so wird offensichtlich, dass im Vergleich zum Training mit Experience Replay Buffer der Agent einen sehr niedrigen Action-Value $Q(s, a)$ approximiert. Dies ist ein Indiz dafür, dass der Agent nicht in der Lage ist eine zielführende Policy anhand de Action-Value Function Q zu erlernen.

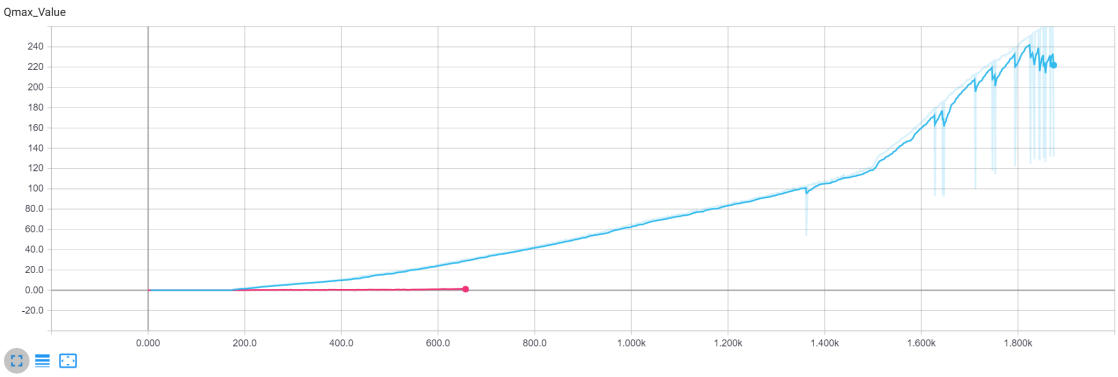


Abbildung 4.36.: Durchschnittlicher Action-Value $Q(s, a)$ pro Episode

(3) Training mit geschwächten Constraints

Beim Training mit aufgelockerten Contraints wurde bei der Landung unter normalen Bedingungen der Constraint **Deviation** (Mittellinienabweichung) aufgelockert. Die Parametrisierung in der Tabelle 4.4 in Kapitel 4.2.2 wurde durch einen einzelnen Constraint mit der Bedingung **Deviation** < 1.0 ersetzt.

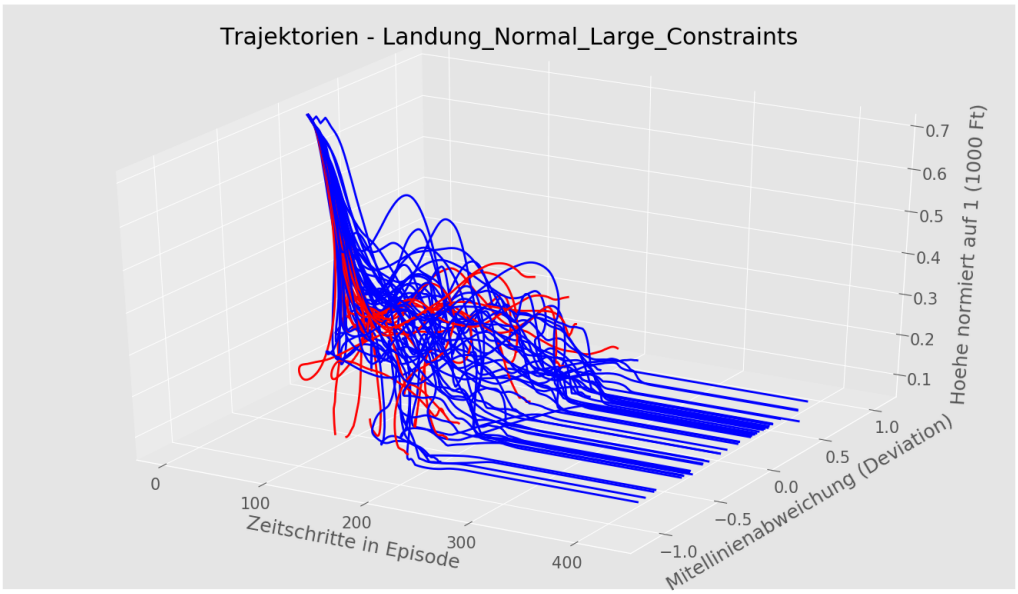


Abbildung 4.37.: Trajektorien der Landung mit aufgelockertem Constraint (Eigene Erstellung)

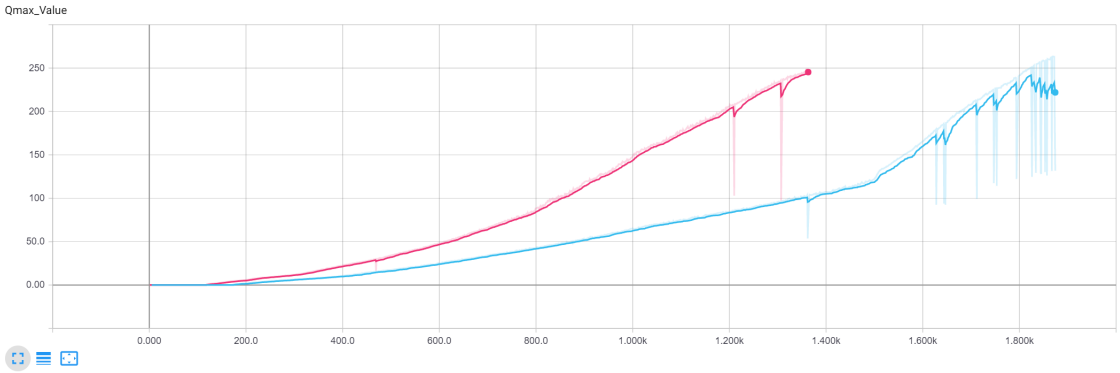


Abbildung 4.38.: Durchschnittlicher Action-Value $Q(s, a)$ pro Episode

Absturz - Landung - Large Constraints	**Erfolgsrate (über 100 Episoden)**
Deviation < 0.5 und Deviation < 0.3	100%
Deviation < 1.0	72%

Tabelle 4.28.: Erfolgreiches Landen auf Landebahn - Landung - Large Constraints

Landen auf Landebahn - Landung - Large Constraints	**Erfolgsrate (über 100 Episoden)**
Deviation < 0.5 und Deviation < 0.3	50%
Deviation < 1.0	20%

Tabelle 4.29.: Erfolgreiches Landen auf Landebahn - Landung - Large Constraints

Das trainierte Modell mit dem schwacheren Constraint erreicht einen höheren Action-Value $Q(s, a)$ als das ursprüngliche Modell (Abbildung 4.38). Es ist ebenfalls zu sehen, dass selbst mit dem aufgeschwächten Constraint, eine funktionierende Policy gelernt wird. Werden allerdings die Trajektorien in 4.38 betrachtet so fällt auf, dass die gelernte Policy allgemein größere Schwierigkeiten hat das Flugzeug auf die Landebahn zu bringen, wie in 4.29 dargestellt.

Allgemein kann anhand dieser 3 Testfälle die Wichtigkeit dieser Komponenten für das Trainieren des Flugzeugautopiloten festgestellt werden. Am wichtigsten für den Trainingserfolg ist der Experience Replay Buffer. Ohne diesen ist es nicht möglich aus aufeinanderfolgenden stark korrelierenden Transitionen zu lernen. **Das Experience Replay Buffer wirkt in diesem Fall wie eine Art Gedächtnis, welches dem Agenten auch alte Transitionen zu Verfügung stellt, die für das Erlernen der Policy ausschlaggebend sind.** Eventuell wäre das Training ohne Experience Replay Buffer mit Rekurrenten Netzwerken möglich, die auch vergangene zeitliche Transitionen erfassen können.

Das Erforschen der Umgebung während des Trainings bewirkt keine allzu große Veränderung in der Trainingsperformance wie der Experience Replay Buffer. **Im durchgeführten Testfall konnte der Agent ohne Erforschung zwar eine Policy erlernen, diese war aber nicht so optimal wie die Policy mit Erforschungsstrategie. Möglicherweise würde hier der Unterschied noch größer ausfallen wenn eine optimierte Erforschungsstrategie definiert wird.**

Letztendlich hat das Auflockern der Constraints für den durchgeführten Testfall den geringsten Unterschied ausgemacht. **Der Agent konnte durch das aufschwächen der Constraints schneller lernen aber die Policy schafft es nicht so überzeugend die Landebahn zu treffen wie die ürsprüngliche Policy.**

Abschließende Worte

In diesem Kapitel wurde die Ergebnisse für die 3 Szenarien des Starts, des Kreuzfahrtflugs und der Landung präsentiert. Grundsätzlich kann unter normalen Bedingungen für alle 3 Szenarien eine funktionierende Policy bestimmt werden. Unter adversen Wetterbedingungen wird die Performance stark beeinträchtigt. Beim Start und Kreuzfahrtflug sind noch funktionierende Policies lernbar, wobei bei die Landung unter adversen Wetterbedingungen sich mit einer Erfolgsrate von 50% schwierig ergibt. Das Auftreten eines Defekts kann der Agent in allen 3 Fällen gut kompensieren. Allerdings fällt auf, dass der Agent aufgrund der gesammelten Transitionen, den auftretenden Defekt in Erinnerung behält und somit teilweise etwas zu stark kompensiert.

Eher unerfolgreich hat sich die Generalisierungsfähigkeit der unter normalen Bedingungen trainierten Modelle erwiesen. Teilweise sinkt die Erfolgrate über 50% oder die Bewältigung des Szenarios stellt sich als nicht möglich heraus.

Durch die Analyse der Trainingsdaten konnte bei den trainierten Modellen eine Konvergenz festgestellt werden. Zudem wird auch eine positive Korrelation zwischen dem Anstieg des Action-Values und der gesammelten Belohnung offensichtlich. Schließlich wurden die Auswirkungen vermeintlich wichtiger Komponenten des DDPG-Algorithmus untersucht. Es konnte unter anderem festgestellt werden, dass der Experience Replay Buffer von hoher wichtiger für den Trainingserfolg ist.

5. Zusammenfassung

In dieser Arbeit sollte ein Reinforcement Learning basierter autonomer Flugzeugautopilot entwickelt werden. Dafür wurde der Deep Deterministic Policy Gradients Algorithm verwendet, der unter der Verwendung von Deterministic Policy Gradients in der Lage ist in einem kontinuierlichen Zustandsraum einen Aktionsvektor mit kontinuierlichen Aktionen zu approximieren.

Dabei sollte der Autopilot unter verschiedenen Bedingungen in der Lage sein zu Starten, einen Kreuzfahrtflug aufrecht zu erhalten und zu Landen. Die Grundlagen hierfür wurden in Kapitel 2 gelegt beginnend mit einer Einführung in das Supervised Learning und in neuronale Netze, die ein Hauptbestandteil des Reinforcement Learnings sind. Im folgenden Kapitel des Reinforcement Learnings wurden alle notwendigen Grundlagen zum Verständnis des DDPG-Algorithmus erläutert und anschließend die Grundlagen des Deterministic Policy Gradient und des DDPG-Algorithmus präsentiert. Letztendlich wurden Konzepte und Methodiken vorgestellt, um eine Belohnungsfunktion für ein komplexes System zu modellieren.

Die Implementierung des Flugzeugautopiloten gliedert sich in 3 Teile. Zunächst wird die Kommunikation mit dem Flugsimulator über UDP-Pakete beschrieben. Die folgenden 2 Teile sind eine Fortsetzung der Grundlagen in denen der DDPG-Algorithmus und die Belohnungsfunktion implementiert wird.

Diese Arbeit besitzt einen ausführlichen theoretischen Anteil, der alle notwendigen Grundlagen zum Verständnis und Implementierung des DDPG-Algorithmus erläutert. Die Implementierung des DDPG-Algorithmus wurde dabei aus bereits bestehenden Repositories zusammengetragen. Die Implementierung der Belohnungsfunktion für den Autopiloten wurde hingegen völlig neu konzipiert.

Zu den größten Schwierigkeiten dieser Arbeit zählen zum einen der hohe theoretische Anteil und zum anderen die investierte Zeit, um jedes Szenario zu modellieren und zu trainieren. Besonders die Spezifikation der Constraints und der notwendigen Belohnungssignale sind im

Laufe mehrerer Trainingsiterationen entstanden. Dabei war es ausschlaggebend die Rolle aller Komponenten des DDPG-Algorithmus zu verstehen, um so das resultierende Verhalten und die resultierenden Daten korrekt interpretieren zu können.

Die trainierten Modelle sind tatsächlich in der Lage unter bestimmten Bedingungen Start, Kreuzfahrtflug und Landung zu bewältigen. Dabei können die trainierten Modelle auch mit Einschränkungen je nach Szenario auf Änderungen der Anfangsbedingungen reagieren. Beispielsweise konnte beim Start ein unter normalen Bedingnungen trainiertes Modell auch unter adversen Wetterbedingungen eingesetzt werden. In den beiden andere Szenarien scheiterte das unter normalen Bedingungen trainierte Modell kläglich. Vor allem schlecht reagierte das vortrainierte Modell auf einen auftretenden Defekt. Hier kann das Modell die notwendigen Aktionen zum Kompensieren des Defekts nicht ermitteln, wodurch das Trainieren eines neuen Modells erforderlich wird.

Als einfachstes zu lösendes Szenario hat sich der Start herausgestellt. Hier kann der Autopilot selbst unter kritischen Wetterbedingungen das Flugzeug starten. Sowohl mit dem von Grund auf trainierten und dem vortainierten Modell. Der Kreuzfahrtflug kann ebenfalls unter normalen Bedingungen problemlos realisiert werden, allerdings bewirken bereits moderate Änderungen der Wetterbedingungen eine Beeinträchtigung der Performance. In diesem Szenario hat sich die Spezifikation der Belohnungsfunktion als besonders schwer erwiesen, da im Vergleich zu Start und Landung wenige eindeutige Belohnungssignale ermittelt werden konnten. Als schwerstes Szenario hat sich die Landung erwiesen. Diese ist auch unter normalen Bedingungen gut realisierbar. Allerdings sinkt bei adversen Wetterbedingungen die Effektivität des trainierten Modells bemerkbar, so dass ein Landen bei adversen Wetterbedingungen nur unter aufgelockerten Wetterbedingungen möglich ist. Vor allem die Generalisierungsfähigkeit ist unter diesem Szenario praktisch nicht vorhanden. Das Landen erfordert ein präzises Ansteuern der Landebahn und eine präzise Regulierung der Flughöhe, wodurch die trainierten Modelle entsprechend empfindlich auf veränderte Bedingungen reagieren.

Grundsätzlich hängt das Trainieren eines erfolgreichen Modells im Wesentlichen von der Definition einer passenden Belohnungsfunktion und vom Sammeln aussagekräftiger Daten mittels einer passenden Erforschungsstrategie ab. Ebenfalls hat sich herausgestellt, dass der Experience Replay Buffer als eine Art Gedächtniss fungiert in dem dieses alte Transitionen zum Trainieren des Agenten sammelt. Das nicht Vorhandensein des Experience Replay Buffers führt deshalb zu einem nicht lernenden Agenten. Das Sammeln von relevanten Daten ist ein

im Supervised Learning bereits bekanntes Problem, wobei die Definierung einer Belohnungsfunktion im Reinforcement Learning eine zusätzliche Herausforderung darstellt. Desweiteren hängt zurzeit die Implementierung eines trainierten Modells in ein echtes Fluggerät von einer akkuraten Simulation des zu steuernden Gerätes ab, da ein Modell lediglich über mehrere Trainingsepisoden trainiert werden kann.

Trotz der Einschränkungen schafft es das DDPG-Framework eine koherente und recht zuverlässige Steuerung für den Flugsimulator zu approximieren. Dabei können die oben gennante Stellschrauben weiter optimiert werden, um noch bessere Ergebnisse zu erzielen. Es bleibt spannend die Entwicklung der Verfahren zur Bewältigung von kontinuierlichen Zustands und Aktionsräumen zu beobachten und ob bereits bestehende Verfahren wie das präsentierte Framework der Deterministic Policy Gradients weiterhin optimiert werden können.

Außer der Entwicklung des steuernden Agenten ist während der Entwicklung dieser Arbeit das Problem der Definierung einer passenden Belohnungsfunktion viel größer in den Vordergrund gedrungen als vorher erwartet. Grundsätzlich war der Lernerfolg an der Definierung einer passenden Belohnungsfunktion gebunden. Dies ist ein großes Forschungsgebiet, dass unabhängig vom Deep Learning seit mehreren Jahren im Bereich des Reinforcement Learnings stattfindet. In weiterführenden Arbeiten sollte dieser Aspekt weitaus tiefer behandelt werden, um möglicherweise ein zuverlässigeres Verhalten erreichen zu können.

Zusätzliche Optimierungen und die mögliche Bewältigung der Limitierungen des DDPG-Algorithmus werden im folgenden Ausblick behandelt. Dabei werden zunächst Implementierungsvarianten des DDPG-Algorithmus präsentiert, die den Zustandsraum durch Erfassen von zeitlichen Zusammenhängen zwischen den Transitionen besser approximieren können. Darüber hinaus wird in Anbetracht der erfahrenen Limitationen in dieser Arbeit auch das Datenverhalten und die Fragilität der im Deep Learning, spezifisch im Reinforcement Learning trainierten Modelle diskutiert.

6. Ausblick

Abschließend sollen Einblicke auf die weitere Entwicklung eines autonomen Flugsystems anhand von Methoden des maschinellen Lernenes gewährt werden und eine Übersicht über die derzeitige Ausrichtung des Reinforcement Learnings und des Deep Learnings gegeben werden.

Auf unmittelbarer Ebene kann der in dieser Arbeit angewendete DDPG-Algorithmus durch Variationen in dessen Implementierung optimiert werden. Diese Variationen werden im Folgenden kurz aufgeführt als Anregung für eine weitere Entwicklung des hier implementierten Flugzeugautopiloten. Beispielsweise kann das konventionelle neuronale Netzes durch ein Convolutional Network ersetzt werden, der den Zustandsraum durch das Einführen von Bildern optimaler approximieren kann. Eine andere Möglichkeit bieten rekurrente Netzwerke wie in **"Memory-based control with recurrent neural networks"(Hees, Hunt, 2015)** präsentiert. Hier wird der DDPG-Algorithmus mit einfachen rekurrenten neuronalen Netzen implementiert, die in der Lage sind zeitliche Zusammenhänge zwischen den Transitionen zu erfassen.

6.1. DDPG mit Convolutional Networks

In **Continuous control with deep reinforcement learning** Lillicrap u. a. (2015) wurde die DDPG-Architektur außer mit einfachen Fully Connected Networks auch bereits mit Convolutional Networks implementiert. Dabei wird ein gesammtes Bild als Zustandsvektor behandelt, wo jeder Pixel einen einzelnen Zustandsparameter repräsentiert. Um den Convolutional Networks zusätzlich den Eindruck von sich bewegenden Frames zu übermitteln, wurden für die in einer Sekunde erfassten Frames jeweils alternierende Frames zu einem Tensor zusammengefasst und zum Training übergeben. Abbildung 6.1 veranschaulicht diesen Vorgang für das Atari Deep-Q-Network welcher mit dem angewendeten Verfahren in **Continuous control with deep reinforcement learning** identisch ist:

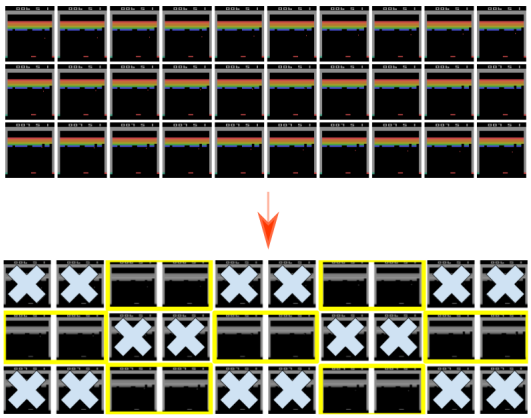


Abbildung 6.1.: Samplen des Zustandsvektor aus Bildern Seita (2016)

Ein zum Flugzeugautopiloten vergleichbares Problem ist der TORCS Rennsimulator, der gerne als Testumgebung zur kontinuierlichen Steuerung von Fahrzeugen genommen wird. Hier erzielt die DDPG Variante mit Convolutional Networks tatsächlich im Durchschnitt einer höhere Belohnung (Abbildung 6.2).

environment	$R_{av,lowd}$	$R_{best,lowd}$	$R_{av,pix}$	$R_{best,pix}$	$R_{av,cntrl}$	$R_{best,cntrl}$
torcs	-393.385	1840.036	-401.911	1876.284	-911.034	1961.600

Abbildung 6.2.: Performance mit Bildern als Zustandsvektor: rot - Pixel, blau - ohne Pixel Lillicrap u. a. (2015)

6.2. RDP

Eine weitere Implementierungsmöglichkeit wird in **Memory-based control with recurrent neural networks** Heess u. a. (2015) behandelt. Hier wird angenommen, dass das zu lösende Problem durch die auftretenden Zustände nicht vollständig beschrieben wird. Formal betrachtet ist das Problem kein voll observierbares Markov Decision Process sondern nur noch ein partiell observierbares Markov Decision Process. Da der zugrundeliegende Markov Process das Verhalten des Agenten nicht vollständig beschreibt wird es notwendig zusätzliche Informationen zu verschaffen, speziell in der Form von zeitlichen Zusammenhängen zwischen den Transitionen. Die vorher beschriebene Variante mit Convolutional Networks bringt diese Komponente be-

reits hinein in dem verschiedene Samples innerhalb einer Aktion zusammengestellt werden. Optimaler kann dies allerdings durch rekurrente neuronale Netze beschrieben werden.

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim p^\mu}\left[\nabla_\theta \mu_\theta(s)\, \nabla_{\mu_\theta(s)} Q_w(s, \mu_\theta(s))\right] \quad \text{(DDPG)}$$

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau\left[\Sigma_t \gamma^{t-1}\, \nabla_\theta \mu_\theta(h_t)\, \nabla_{\mu_\theta(h_t)} Q_w(h_t, \mu_\theta(h_t))\right] \quad \text{(RDPG)}$$
$$\tau = (s_1, o_1, a_1, s_2, o_2, a_2, \ldots)$$
$$h_t = (o_1, a_1, \ldots, o_{t-1}, a_{t-1}, o_t)$$

Konkret wird der DDPG-Algorithmus mit rekurrenten neuronalen Netzen implementiert. Dabei wird der Zustandsvektor durch einen Vektor h_t ersetzt, der aus einer Verteilung τ mehrerer Trajektorien Transitionen in Form des partiell observierten Zustands o_t und der zugehörigen Aktion a_t speichert. Die DDPG-Architektur bleibt dabei gleich, wobei die Gradienten durch Backpropagation Through Time aufgrund der rekurrenten neuronalen Netze ermittelt werden.

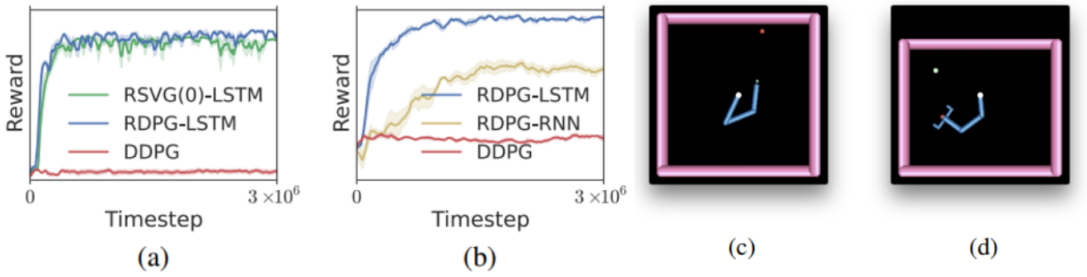


Abbildung 6.3.: Ergebnisse des RDPGs Heess u. a. (2015)

Die TORCS-Umgebung wurde unter dem RDPG Algorithmus nicht getestet. Die komplexeste getestete Aufgabe ist das Greifen eines roten Markers eines Greifers, dessen Ergebnisse in Abbildung 6.3 dargestellt sind. Dabei sieht der Agent den roten Marker nur wenn dieser stationär ist. In diesem Fall ist die RDPG Implementierung der DDPG Implementierung komplett überlegen. Die herkömmliche Implementierung mit Fully Connected Networks kann in diesem Fall keine richtige Policy erlernen. Letztendlich bleibt zu überprüfen, ob die Annahme eines partiell observierbaren Markov Decision Process für die Implementierung des Flugzeugautopiloten zutreffend ist.

6.3. A Critical Appraisal of Deep Learning

Diese alternativen Implementierungen beschleunigen in fast allen Fällen die Lerngeschwindigkeit und Stabilität. Nichtsdestotrotz, garantieren diese Verbesserungen nicht zwangsläufig das Erlernen von komplexeren Aufgaben. Das Lösen von komplexeren Aufgaben hängt vielmehr von zwei Aspekten ab: Zum einen an dem Sammeln von vielfältigen und aussagekräftigen Daten, die das System optimal beschreiben. Zum anderen müssen die zugrundeliegenden Verfahren mächtiger werden, in dem diese einkommende Daten effizienter verarbeiten, um dadurch komplexere Systeme approximieren zu können.

Seit der Veröffentlichung des Papers **Classification with Deep Convolutional Neural Networks** Krizhevsky u. a. (2012) werden Jahr für Jahr stets effizientere und mächtigere Verfahren im Bereich des Deep Learnings entwickelt. Dabei hat Deep Learning den Bereich des Supervised Learnings komplett revolutioniert und komplexe Reinforcement Learning Aufgaben im Bereich des Machbaren katapultiert. Nichtsdestotrotz, wurden im letzten Jahr bereits erste Bedenken über die große Erwartungshaltung an das Deep Learning geäußert. Vor allem wird befürchtet, dass Deep Learning als einziger Lösungsweg für Systeme gesehen wird, die eventuell auf eine effizientere Art und Weise gelöst werden können. Guy Marcus verfasste im letzten Jahr eine Kritik in **Deep Learning: A Critical Appraisal** Marcus (2018), die darauf besteht nicht alle Ressourcen in das Deep Learning zu investieren sondern dieses stattdessen als ein mächtiges Werkzeug unter vielen anderen Werkzeugen im maschinellen Lernen zu betrachten. Vor allem sieht Marcus zwei Aspekte des Deep Learnings kritisch:

- **Aktuelle Verfahren sind datenhungrig.**
- **Die Übertragungsfähigkeit der trainierten Modelle ist zurzeit noch sehr limitiert.**
- **Die fehlende Integration mit bereits bestehenden Intelligenten Systemen.**

Trainierte Modelle sind in der Regel sehr spezialisiert. Auch in dieser Arbeit ist zu sehen, dass die Performance eines trainierten Modells bereits bei kleinen Änderungen der Anfangsbedingungen stark beeinträchtigt wird. Darüber hinaus beklagt Marcus, dass aktuelle Modelle keine allgemeine Intelligenz beweisen. Besonders offensichtlich wird dies bei sprachbasierten Aufgaben, wo die Interpretation des Inhaltes gefragt ist.

Weitere Punkte weisen auf die Intransparenz der neuronalen Netze hin, die trotz bestehender Visualisierungstechniken eher ausschließlich durch Trial and Error parametrisiert werden

können. In diesem Zusammenhang ist ein wichtiger Punkt die Implementierungsfähigkeit dieser Modelle im Zusammenhang mit bereits bestehenden Verfahren im maschinellen Lernen. Zu oft werden Verfahren des Deep Learnings als eigenständige Systeme betrachtet, die ein Problem vollständig lösen sollen anstatt diese mit bereits bestehenden Systemen zu integrieren.

Letzendlich ist es von hoher Wichtigkeit zu identifizieren in welchen Bereichen Deep Learning tatsächlich ein starkes Werkzeug ist. Grundsätzlich schafft es Deep Learning mit einer zuvor beispielslosen Genauigkeit komplexe Klassifizierungsaufgaben zu bewältigen, die in der Vergangenheit als nur schwer zu bewältigen galten. Dabei wird vorrausgesetzt, dass ein großer Datensatz vorhanden ist, der die zu klassifizierenden Mengen gut beschreibt.

Die Effektivität des Deep Learnings sinkt wenn ein Problem nicht mehr als reines Klassifizierungsproblem formuliert werden kann. In diesem Fall ist es erforderlich das zu lösende Problem so gut wie möglich als ein Klassifizierungsproblem zu beschreiben. Siehe in dieser Arbeit den Experience Replay Buffer und die Target Networks. Vor allem hier besteht die Gefahr, dass suboptimale Lösungen anhand des Deep Learnings gesucht werden anstatt passendere Verfahren zu entwickeln. Marcus appeliert dabei als Beispiel Ressourcen im Bereich des Unsupervised Learnings zu investieren, so dass die Notwendigkeit für vorklassifizierte Musterdaten reduziert wird.

Trotz der Kritik ist es noch nicht absehbar wie lang die rapide Entwicklung des Deep Learnings anhalten wird. Im Reinforcement Learning Problem ist nicht nur die Entwicklung der Intelligenz ausschlaggebend, die in diesem Paper durch den DDPG-Algorithmus vorwiegend behandelt wurde sondern ebenfalls die Definierung einer entsprechenden Belohnungsfunktion. Vor allem bei komplexeren Problemen ist die Entwicklung einer passenden Belohnungsfunktion ein fast genau so großer Faktor wie die Entwicklung des Deep Learning Agenten.

A. Adam: Adaptive Moments Optimizier

Adam (Adaptive Moments Optimizer) ist eine Optimierung des Stochastic Gradient Descent Algorithmus. Dabei werden die Parameter nicht direkt mit dem unmittelbar ermittelten Gradient aktualisiert sondern mit sogennanten Momenten, die durch ein exponentielle Glättung der ermittelten Gradienten und der quadrierten Gradienten entstehen:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Das erste Moment m_t approximiert dabei den Mittelwert des Gradienten, wobei das zweite Moment die Varianz approximiert. Letztendlich werden die Parameter mit dem Quotienten beider Momente aktualisiert:

$$\theta_t = \theta_{t-1} - \alpha \frac{m_t}{(\sqrt{v_t} + \epsilon)}$$

Das Paper **Adam: A Method for Stochastic Optimization** Kingma und Lei Ba (2014) behandelt die Entstehung des Adam Optimizers im Detail.

B. Beweis 1: Deterministic Policy Gradient

In Kapitel 2.3.1 wurde das Deterministic Policy Gradient Theorem vorgestellt. Das Deterministic Policy Gradient Theorem beweist, dass aus der Kostenfunktion einer deterministischen Policy der Deterministic Policy Gradient hergeleitet werden kann, der wiederum der Gradient einer Action-Value Function entspricht. Der vollständige Beweis ist im Anhang B des **Determinstic Policy Gradient Algorithms** Silver u. a. (2014) Papers unter: http://proceedings.mlr.press/v32/silver14-supp.pdf zu finden.

C. Beweis 2: Deterministic Policy Gradient

Im zweiten Schritt wurde in Kapitel 2.3.1 aufgeführt, dass der Deterministic Policy Gradient ein limitierender Fall des Stochastic Policy Gradient ist. Dies ist eine Vorraussetzung, um annehmen zu können, dass die Actor-Critic Architektur auch auf eine deterministiche Policy angewendet werden kann. Der Beweis des zweiten Theorems ist unter Anhang C des **Determinstic Policy Gradient Algorithms** Silver u. a. (2014) Papers unter: http://proceedings.mlr.press/v32/silver14-supp.pdf zu finden.

D. DDPG-Netzwerkparameter

Der DDPG-Algorithmus besitzt mehere justierbare Parameter. Im Folgenden werden diese Parameter und deren jeweiligen Werte aufgelistet:

- **Actor-Netz**
  - Layer1: 400 Neurons.
  - Layer2: 300 Neurons.
  - Lernrate: 1e-4.
  - tau (Target-Network Aktualisierungsverzögerung): 0.001.
- **Critic-Netz**
  - Layer1: 400 Neurons.
  - Layer2: 300 Neurons.
  - Lernrate: 1e-3.
  - tau: 0.001
  - L2 (Regularisierungsfaktor λ): 0.01
- **Größe des Experience Replay Buffers:** 1e+6 .
- **Füllungsgröße des Experience Replay Buffers vor Trainingsanfang:** 1e+4.
- **Mini-Batch Größe:** 64.
- **GAMMA:** 0.99.

Literaturverzeichnis

[Champandard 2003] Champandard, Alex J.: *AI Game Development: Synthetic Creatures with Learning and Reative Behaviors*. New Riders, 2003

[cnblogs 2015] cnblogs: *cnblogs.com*. 2015. – URL https://www.cnblogs.com/geniferology/p/what_is_reinforcement_learning.html. – Zugriffsdatum: 2018-02-26

[Cybenko 1989] Cybenko, George: Approximation by Superpositions of a Sigmoidal Function. (1989)

[Dorigo und Colombetti 1997] Dorigo, Marco ; Colombetti, Marco: Robot Shaping: An Experiment in Behavior Engineering. (1997)

[Heess u. a. 2015] Heess, Nicolas ; Hunt, Jonathan J. ; Lillicrap, Timothy P. ; Silver, David: Memory-based control with recurrent neural networks. (2015)

[Hussain 2015] Hussain, Mahboob: *A Program for Linear Regression With Gradient Descent*. 2015. – URL https://dzone.com/articles/program-for-linear-regression-with-gradient-descen. – Zugriffsdatum: 2018-02-26

[Kingma und Lei Ba 2014] Kingma, Diederik P. ; Lei Ba, Jimmy: Adam: A method for stochastic optimization. (2014)

[Krizhevsky u. a. 2012] Krizhevsky, Alex ; Sutskever, Ilya ; Hinton, Geoffrey E.: ImageNet Classification with Deep Convolutional Neural Networks. (2012)

[Lillicrap u. a. 2015] Lillicrap, Timothy P. ; Hunt, Jonathan J. ; Pritzel, Alexander ; Heess, Nicolas ; Erez, Tom ; Tassa, Yuval ; Silver, David ; Wierstra, Daan: Continuous control with deep reinforcement learning. (2015)

[Marcus 2018] Marcus, Gary: Deep Learning: A Critical Appraisal. (2018)

[McCullock 2012] McCullock, John: *Q-Learning Tutorial*. 2012. – URL http://mnemstudio.org/path-finding-q-learning-tutorial.htm. – Zugriffsdatum: 2018-02-26

[Meisel] Meisel, Andreas: *Neuronale Netze*. Vorlesung Robotvision

[Mnih u. a. 2015] Mnih, Vlodymyr ; Silver, David ; Kavukcuoglu, Koray: Human-level control through deep reinforcement learning. (2015)

[Ng u. a. 2006] Ng, Andrew Y. ; Coates, Adam ; Diel, Mark ; Ganapathi, Varun ; Schule, Jamie ; Tse, Ben ; Berger, Eric ; Liang, Eric: Autonomous inverted helicopter flight via reinforcement learning. (2006)

[Ng u. a. 1999] Ng, Andrew Y. ; Harada, Daishi ; Russel, Stuart: Policy invariance under reward transformations: Theory and application to reward shaping. (1999)

[Olah 2014] Olah, Christopher: *Neural Networks, Manifolds, and Topology*. 2014. – URL http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/. – Zugriffsdatum: 2018-02-26

[Poole und Mackwort 2010] Poole, David ; Mackwort, Alan: *Artificial Intelligence*. 2010. – URL http://artint.info/html/ArtInt_224.html. – Zugriffsdatum: 2018-02-26

[Randlov und Alstrom 1998] Randlov, Jette ; Alstrom, Preben: Learning to Drive a Bicycle using Reinforcement Learning and Shaping. (1998)

[Roy 2015] Roy, Souman: *Linear Regression by using Gradient Descent Algorithm: Your first step towards Machine Learning*. 2015. – URL https://medium.com/meta-design-ideas/linear-regression-by-using-gradient-descent-algorithm-your-first-ste – Zugriffsdatum: 2018-02-26

[Saksida u. a. 1998] Saksida, Lisa M. ; Raymond, Scott M. ; Touretzky, David S.: Shaping Robot Behavior Using Principles from Instrumental Conditioning. (1998)

[Seita 2016] Seita, Daniel: *Frame Skipping and Pre-Processing for Deep Q-Networks on Atari 2600 Games*. 2016. – URL https://danieltakeshi.github.io/2016/11/25/frame-skipping-and-preprocessing-for-deep-q-networks-on-atari-2600-g – Zugriffsdatum: 2018-02-26

[Silver 2015] Silver, David: UCL Course on RL: Lecture 3: Planning by Dynamic Programming. (2015)

[Silver u. a. 2014] Silver, David ; Lever, Guy ; Heess, Nicolas ; Degris, Thomas ; Wierstra, Daan ; Riedmiller, Martin: Deterministic Policy Gradient Algorithms. (2014)

[Sutton und Barto 2017] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement Learning: An Introduction*. Bradford Book, 2017

[Ved 2016] Ved: *How to improve performance of neural networks*. 2016. – URL https://d4datascience.wordpress.com/2016/09/29/fbf/. – Zugriffsdatum: 2018-02-26

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 28. Februar 2018 Stefan Sylvius Wagner