Sim-To-Sim Gap des Cartpole Environments

Max Bauer

Hochschule für Angewandte Wissenschaften, Hamburg, Germany

Zusammenfassung. Das Training eines auf Reinforcement Learning basierenden Agenten gestaltet sich auf physischer Hardware ressourcen-, personal- und zeitaufwändig, weshalb häufig auf das Trainieren innerhalb von Simulationen zurückgegriffen wird. Diese können die Realität jedoch nicht gänzlich akkurat abbilden, weshalb in Simulationen erfolgreich trainierte Agenten meist daran scheitern, die vorgegebene Aufgabe in physischen Umgebungen zu lösen. Als Sim-To-Real Gap wird das Problem des Policy Transfers in der Literatur bezeichnet. Um diese in Bezug auf OpenAIs Cartpole Environment zu untersuchen, wurde aufgrund eines fehlenden physischen Cartpole Modells die Sim-to-Sim Gap untersucht. Zunächst wurden drei auf verschiedenen Q-Learning Algorithmen basierende Agenten erfolgreich auf das Cartpole Environment trainiert. Anschließend wurden die Agenten und das Environment mit den Rauschmethoden Observation Noise, Domain Randomisation und Random Force Injection versetzt. Experimente zeigen, dass die Agenten, denen das Rauschverhalten während des Trainings verwehrt blieb, nur bis zu einem gewissen Grad dem Rauschen standhalten konnten. Daraufhin wurden die Rauschmethoden in das Training integriert, um Agenten widerstandsfähiger zu machen. Weitere Experimente zeigen allerdings, dass das Hinzufügen von Observation Noise und Domain Randomisation in den Trainingsvorgang die Agenten sogar anfälliger machte. Random Force Injection hatte wiederum wenig Einfluss auf rauschfreie Agenten.

Schlüsselwörter: Q-Learning · Cartpole · Policy Transfer · Sim-To-Sim Gap · Observation Noise · Domain Randomisation · Random Force Injection

1 Einleitung

Das Training von Agenten anhand physischer Systeme erfordert oft Expertenwissen, Zeit und Geld, um für den Versuchsaufbau notwendiges Material zu kaufen [1] [2]. Werden in einer Simulation trainierte Agenten anschließend in einer physischen Umgebung eingesetzt, sinkt ihre Performance dort meist drastisch [1] [2] [3] [4] [5] [6]. Das liegt vor allem daran, dass eine Simulation nicht vollständig alle physischen Parameter akkurat abbilden kann [2]. In anderen Worten nutzt ein Agent Modellierungsfehler der Simulation aus, wenn diese zur Lösung der vorgegebenen Aufgabe beitragen [1]. Ein Policy Transfer aus einer Simulation auf die physische Welt ist also ohne Weiteres unmöglich. Mit dem Sim-To-Sim

Transfer werden in einer Simulation trainierte Agenten in einer weiteren Simulation eingesetzt, welche mit verrauschten Umgebungsparametern versetzt wurde [2]. Diese Art des Transfers wird auch in dieser Arbeit betrachtet.

1.1 Forschungsstand

In [1] wenden die Autoren Domain Randomisation an, um den Policy Transfer unter anderem bei einem realen Cartpole Environment zu bewerkstelligen. Das Ziel war, dass die Realität dem lernenden Agenten nur noch als eine bekannte Variation der Simulation erscheint, in welcher er die Stange ebenso balancieren kann. Vorangegangene Methoden zur Überbrückung des Policy Transfers behandelten das Hinzufügen von Observation Noise [7] oder das Lernen einer Transferfunktion, welche eine aus einer Simulation erlernten Policy mit einem Score versieht, der angibt, inwiefern die Policy einer optimalen Policy der realen Welt entspräche. Des Weiteren wird in [8] Domain Adaption erwähnt, was eine realistischere Parameterkonfiguration der Simulation nahelegt [9]. Tobin et al. fokussieren sich in [4] rein auf visuelle Domain Randomisation. Ein Robotergreifarm wurde darauf trainiert, aus einem unübersichtlichen Pool an geometrischen Objekten ein Bestimmtes herauszunehmen. Domain Randomisation fand in der Form statt, als dass Objekte in der Simulation verschiedene Formen und Farben annahmen, die Hintergrund- sowie die Roboterarmfarbe zufällig gewählt und das zu nehmende Objekt von anderen Objekten teilweise verdeckt wurde. Slaoui et al. [6] ersetzen Domain Randomisation durch Domain Regularisation, da Domain Randomisation unter verschiedenen Aspekten leidet, wie einen hohen Konfigurationsaufwand als auch domainspezifisches Expertenwissen vorauszusetzen sowie zu einer hohen Policyvarianz zu führen, sodass möglicherweise keine erlernte Policy passabel unter realen Bedingungen funktionieren kann. Valassakis et al. machen sich ebenfalls Domain Randomisation zunutze [3], um einem Robotergreifarm verschiedene Fähigkeiten wie das Berühren eines bestimmten Objektes beizubringen. Wegen der genannten Probleme, die Domain Randomisation mit sich bringt, wurden neben Domain Randomisation die erwähnte Methode Observation Noise sowie eine weitere Sim-To-Real Methode namens Random Force Injection angewandt [10]. Letztere verändert die physikalischen Parameter der Simulation nur noch insofern, als dass Kräfte, die auf einen Agenten einwirken, aus einer uniformen Verteilung entnommen werden. Im Gegensatz zu Domain Randomisation werden andere Simulationsparameter wie beispielsweise Masse oder Reibung nicht einer Verteilung entnommen und sind daher statisch. Interessanterweise wurde mit Random Force Injection meist eine bessere Performance in der physischen Welt erzielt als mit Domain Randomisation, obwohl die Parameteranzahl nur noch einen Bruchteil der Anzahl von Domain Randomisation ausmachte. Zhao et al. [11] betrachten eine Reihe weiterer Sim-To-Real Policy Transfer Methoden und kommen zu dem Schluss, dass Domain Randomisation die aktuell am häufigsten gewählte Transferlösung ist, während jedoch beispielsweise Policy Distillation effizienter und ressourcenschonender ist.

1.2 Cartpole Environment

Ziel eines Agenten im Cartpole Environment[1] ist es, eine auf einem Wagen senkrecht befestigte Stange zu balancieren. Der Wagen kann sich nur eindimensional nach links oder rechts bewegen, wofür pro Zeitschritt eine konstante Kraft auf ihn ausgeübt wird (siehe Abbildung 1). Für jeden Zeitschritt erhält der Agent einen Reward von +1. Beobachtet wird die aktuelle Position des Wagens, dessen horizontale Bewegungsgeschwindigkeit, der Winkel der Stange zur Vertikalen sowie deren Winkelgeschwindigkeit. Daraus ergeben sich Aktions- und Beobachtungsvektoren der Größe zwei beziehungsweise vier. Zu Beginn jeder Epoche werden alle Observationen der uniformen Verteilung $\mathcal{U}$(-0.05, 0.05) entnommen. Eine Epoche endet, wenn das Zeitlimit von 200 Zeiteinheiten erreicht wird, der Wagen sich mehr als 2.4 Distanzeinheiten vom Startpunkt entfernt oder die Stange einen Winkel von mehr als 15° zur Vertikalen aufweist. Die Aufgabe gilt als gelöst, wenn über 100 konsekutive Epoche hinweg ein durchschnittlicher Reward von mindestens 195 erreicht wurde.

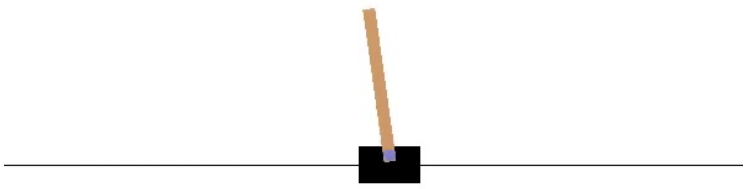


Abb. 1. Cartpole Environment

1.3 Q-Learning

Bei Q-Learning handelt es sich um ein Off-Policy Lernverfahren, da der Agent anhand der Expected Returns beziehungsweise Q-Werte der darauffolgenden State-Action Paare lernt [12]. Der Q-Learning Algorithmus wurde 1989 von Watkins et al. [13] folgendermaßen definiert:

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

[1] https://gym.openai.com/envs/CartPole-v0/

wobei $Q(S_t, A_t)$ für den Q-Wert für die Aktion $\boldsymbol{a} \in \boldsymbol{A}$ im State $\boldsymbol{s} \in \boldsymbol{S}$ zum Zeitpunkt t steht. α stellt die Lernrate, R_{t+1} den Reward für die gewählte Aktion und γ den Discount Faktor dar. $\max_a Q(S_{t+1}, a)$ ist der Expected Return im State S_{t+1} für die Aktion a, für welche aus S_{t+1} heraus der Agent den größten Expected Return ziehen könnte. $\max_a Q(S_{t+1}, a) - Q(S_t, A_t)$ ist daher der Loss beziehungsweise die Differenz aus dem Q-Wert der optimalen Aktion a im State S_{t+1} mit dem Q-Wert der aktuell gewählten Aktion A_t im State S_t. Damit wird ermittelt, ob die aktuell gewählte Aktion besser im Sinne eines größeren Expected Returns ist als der bisherige vorhandene Expected Return für das State-Action-Paar (S_t, A_t). Der Discount Faktor γ wird wie üblich dafür eingesetzt, den Agenten zu motivieren, unmittelbare Rewards stärker zu berücksichtigen, um zu verhindern, dass er einen unendlichen Expected Return zu erreichen versucht [14]. Schließlich wird mit der Lernrate α geregelt, wie stark der neu berechnete Expected Return, oder Q-Wert, in den bisherigen Q-Wert einfließen soll. In der Regel wird die Lernrate im Laufe eines Trainings herabgesetzt, um Trainingskonvergenz zu erreichen [15].

Allgemein gibt die Funktion $Q_\pi(S_t, A_t)$ an, welchen Q-Wert ein Agent erwarten könnte, wenn er im State $\boldsymbol{s} \in \boldsymbol{S}$ die Aktion $\boldsymbol{a} \in \boldsymbol{A}$ ausführen und anschließend der Policy π folgen würde:

$$\begin{aligned} Q_\pi(S_t, A_t) &= E_\pi[G_t \mid S_t = s, A_t = a] \\ Q_\pi(S_t, A_t) &= E[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a] \end{aligned} \tag{2}$$

Die optimale Q-Funktion $Q_*(S_t, A_t)$ gibt demzufolge über alle Policies hinweg an, welche Aktion A_t eines States S_t den größten Q-Wert liefern würde [12]. Ein Agent wüsste also in jedem State S_t anhand $Q_*(S_t, A_t)$, mit welcher Aktion A_t der Expected Return maximiert würde. Dies geht mit dem Finden einer optimalen Policy einher, womit das Reinforcement Learning Problem gelöst wäre. Wie aus (1) hervorgeht, erfüllt $Q_*(S_t, A_t)$ die Bellman Optimality Equation [16]:

$$Q_*(S_t, A_t) = E[R_{t+1} + \gamma \max_a Q_*(S_{t+1}, a)] \tag{3}$$

Intuitiv betrachtet kann die Bellman Optimality Gleichung so verstanden werden, dass eine optimale Policy einen Wert für jeden State liefert, der mit dem Expected Return der optimalen Aktion aus demselben State heraus übereinstimmt.

Q-Learning leidet unter der Tatsache, dass die States eines Trainingsvorganges stark miteinander korrelieren. Jeder State basiert auf seinem Vorgänger, was zu einer hohen Statevarianz führt [17]. In [18] wurde eine mögliche Lösung für dieses Problem vorgestellt, die Experience Replay genannt wird. Anstatt anhand von States zu lernen, die mit vorangegangenen States korrelieren, wird zunächst ein Buffer mit einer bestimmten Größe N initialisiert. Bevor der eigentliche

Trainingsprozess beginnt, versucht der untrainierte Agent, die Aufgabe eines Environments zu lösen, wobei sein Erfolg dabei nebensächlich ist. Wichtig ist, dass der Agent für die N-Male seinen State, seine gewählte Aktion, den dazugehörigen Reward sowie den darauffolgenden State als Tupel $(S_t, A_t, R_{t+1}, S_{t+1})$ dem Experience Replay Buffer hinzufügt. Während des Trainingsvorganges wird dann anhand einer uniformen Verteilung ein MiniBatch an Samples aus dem Experience Replay Buffer verwendet [19]. Dadurch wird sichergestellt, dass die States, mit denen gelernt wird, unabhängig voneinander sind. Das heißt, dass der State, der im vorherigen Trainingsdurchlauf verwendet wurde, nicht der vorangegangene State eines aktuell benutzten States ist. So wird das Training stabilisiert im Sinne einer verringerten Trainingsperformancevarianz.

1.3.1 Deep Q-Learning Q-Werte werden in einer Tabelle gespeichert, wobei Spalten alle möglichen States und Reihen jeweils eine mögliche Aktion eines States repräsentieren. Alle Q-Werte werden zunächst mit dem Wert 0 initialisiert. Ein Agent muss für jede seiner Aktionen lediglich in der Tabelle nachschauen, welche Aktion für seinen derzeitigen State den höchsten Expected Return liefert. Je größer die State- und Action-Spaces werden, desto aufwendiger wird das Ermitteln sowie das Nachschauen aller Q-Werte für den Agenten [19]. Aus diesem Grund wird heutzutage anstelle einer Lookup-Tabelle ein Funktionsapproximator basierend auf neuronalen Netzen gewählt, welcher $Q_*(S_t, A_t)$ annähern soll [20]. Dieses als Policy Netzwerk oder Q_ϕ genannte neuronale Netz nimmt als Input States entgegen, sodass seine Outputs die Q-Werte der in diesem State möglichen Aktionen darstellen. In diesem Fall wird von Deep-Q-Networks (DQN) gesprochen, die die Konzepte hinter Reinforcement Learning und Machine Learning miteinander verknüpfen [19]. Das grundlegende Lernverfahren bleibt hierbei gleich, wobei auf einen signifikanten Unterschied eingegangen werden muss. Die Aktualisierung der Q-Werte beruht in (1) auf der Differenz aus dem optimalen Q-Wert $Q_*(S_t, A_t)$, der mithilfe der rechten Seite in (3) ermittelt wird, mit dem aktuell berechneten Q-Wert $Q(S_t, A_t)$ aus (2):

$$\begin{aligned} Q_*(S_t, A_t) - Q(S_t, A_t) &= loss \\ E[R_{t+1} + \gamma \max_a Q_*(S_{t+1}, a)] - E[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}] &= loss \end{aligned} \tag{4}$$

Da DQNs keine Lookup-Tabellen anlegen, in denen Q-Werte abgespeichert werden, können Agenten die optimalen Q-Werte nicht einfach nachschlagen. Daher wird stattdessen der State S_{t+1}, in dem ein Agent nach Durchführen der gewählten Aktion A_t landet, erneut in Q_ϕ übergeben. Dessen Q-Wert der optimalen Aktion a wird anschließend in (3) eingesetzt, um $Q_*(S_t, A_t)$ zu ermitteln und den Loss anhand von (4) zu berechnen. Anstelle eines Lookups wird also der darauffolgende State S_{t+1} erneut durch das neuronale Netz propagiert, um anhand des Q-Wertes seiner optimalen Aktion die Lossberechnung durchführen zu können.

1.3.2 Target Q-Network Aktuelle Q-Werte $Q(S_t, A_t)$ sowie deren optimale Q-Werte $Q_*(S_t, A_t)$ werden beide mit Q_ϕ mit denselben Gewichtungen ermittelt. Werden die Gewichte aktualisiert, damit $Q(S_t, A_t)$ sich $Q_*(S_t, A_t)$ annähert, so ist der Output von $Q_*(S_t, A_t)$ davon gleichermaßen betroffen, sodass die optimalen Q-Werte sich ebenso weit wieder von den aktuellen Q-Werten entfernen. Dies kann zu Trainingsinstabilitäten führen [21]. Anschaulich ausgedrückt verhält sich Q_ϕ beim Lernen wie ein Hund, der seinen eigenen Schwanz jagt. Als Lösung hierfür wird ein zweites Target Q-Netzwerk oder T_ϕ eingeführt, dass als Klon von Q_ϕ initialisiert wird. Anstatt nun $Q_*(S_t, A_t)$ anhand Q_ϕ zu berechnen, werden die optimalen Q-Werte anhand T_ϕ ermittelt:

$$Q_\phi(S_t, A_t) = E[R_{t+1} + \gamma \max_a T_\phi(S_{t+1}, a)] \tag{5}$$

Alle j Trainingsepochen werden die sich ständig aktualisierenden Gewichte von Q_ϕ auf T_ϕ übertragen. Das bedeutet, dass die Gewichte in Q_ϕ und T_ϕ sich bei der Lossberechnung voneinander unterscheiden beziehungsweise die Gewichte in T_ϕ zeitweise eingefroren werden, sodass dem obigen Problem entgegengewirkt wird [21].

1.3.3 Deep Double-Q-Learning DQNs neigen dazu, Q-Werte überschätzen zu wollen. In einem DQN werden Q-Werte approximiert, weshalb immer von einer gewissen Abweichung vom realen Q-Wert ausgegangen werden kann [22]. Zu Überschätzung der Q-Werte kommt es nun deshalb, weil in $Q_*(S_t, A_t)$ die optimale Aktion mit dem $\max_a$ Operator ermittelt wird. Die approximierten Q-Werte liegen also in den meisten Fällen über den realen Q-Werten. Abhilfe hierfür schafft ein zweites Q-Netzwerk Q_{ϕ_2}, das mit disjunkten Experience Replay MiniBatches trainiert wird verglichen mit dem bestehenden Q_{ϕ_1}. Daher rührt der Name Double Deep-Q-Networks oder DDQNs. Der Lernprozess von Q_{ϕ_1} verändert sich dadurch insofern, als dass mit Q_{ϕ_1} zwar die optimale Aktion, mit Q_{ϕ_2} jedoch deren Q-Wert ermittelt wird [22]:

$$Q_{\phi_1}(S_t, A_t) = E[R_{t+1} + \gamma Q_{\phi_2}(S_{t+1}, \operatorname*{argmax}_a Q_{\phi_1}(S_{t+1}, a))] \tag{6}$$

Analog wird Q_{ϕ_2} trainiert, indem mit Q_{ϕ_2} die Aktion gewählt, aber mit Q_{ϕ_1} deren Evaluation ausgeführt wird.

1.4 Policy Transfer

Nicht akkurat abgebildete Simulationsvariablen wie Beobachtungs-, Umgebungs- oder Kinematiksparameter können zum Erlernen einer Policy führen, die sich nicht dafür eignet, eine Aufgabe in der physischen Welt zu lösen. Steht kein physisches Modell zur Verfügung, kann immerhin der sogenannte Sim-To-Sim Transfer untersucht werden. Dabei wird ein in einer Simulation trainierter Agent in einer weiteren Simulation eingesetzt, deren angesprochene Variablen nicht mehr der ursprünglichen Simulation entsprechen. Damit kann abgeschätzt werden, inwiefern

sich ein simuliert trainierter Agent in der Realität verhalten würde. Um Policy Transfers zu ermöglichen, wird meist eine Bayessche Sichtweise angenommen, bei der verschiedene Simulationsparameter Verteilungen entzogen werden, anstatt diese fest zu definieren. Drei Policy Transfer Verfahren werden im Folgenden näher betrachtet.

1.4.1 Observation Noise Hiermit wird das Rauschen der Umgebungsobservierungen der Agenten in die Simulation integriert [1] [3] [7]. Denn Messdaten aus Sensoren jeglicher Art beinhalten immer eine gewisse Abweichung von der Wirklichkeit [23]. Anstatt nun die Observierungen der Agenten akkurat numerisch abzubilden (wie es eine digitale Simulation ermöglicht), werden sie einer Normalverteilung entnommen, wie sich auch bei einem Sensor in der physischen Welt zu finden wäre. Dabei ist meist in der Sensordokumentation die Standardabweichung σ vom durchschnittlich gemessenen Wert μ angegeben. Dieselbe Verteilung kann in die Simulation integriert werden, um die Agenten an die verrauschten realen Observierungen anzupassen. Jedes Mal, wenn die Umgebung pro Trainingsschritt observiert wird, wird auf die Messdaten Observation Noise angewandt.

1.4.2 Domain Randomisation Als Domain Randomisation werden Verfahren bezeichnet, welche normalverteiltes Rauschen physikalischen Systemparametern wie Masse oder Reibung zuweisen [1] [2] [3] [4]. Auch hier kann in der Realität festgestellt werden, welches Rauschverhalten vorliegt. Beispielsweise kann bei einem physischen Cartpole Modell die Standardabweichung der Masse des Wagens anhand der Waage abgeschätzt werden, mithilfe derer der Wagen gewogen wurde. Denn in der Regel ist in ihrer Dokumentation angegeben, wie verrauscht die Messdaten sind. Im Gegensatz zu Observation Noise werden die verrauschten Parameter nur einmal pro Trainingsepoche einer Verteilung entnommen.

1.4.3 Random Force Injection Im Falle von Random Force Injection werden die Kräfte, die auf einen Agenten einwirken, aus uniformen Verteilungen entnommen [3] [10]. Dies hat den erheblichen Vorteil gegenüber Domain Randomisation, dass weitaus weniger Parameter verrauscht werden müssen. Problematisch werden große Parametervektoren dadurch, dass für jede einzelne Variable eine Verteilung ermittelt werden muss, welche wiederum meistens Expertenwissen voraussetzt. Wie bei Domain Randomisation werden hier die Kinematikvariablen ebenfalls pro Trainingsepoche der Verteilung gezogen.

2 Experimente

Im Folgenden werden Experimente beschrieben, welche die Sim-To-Sim Gap des Cartpole Environments von OpenAI untersuchen sollen. Dafür werden drei Agenten basierend auf den aus Kapitel 1.3 bekannten Q-Learning Algorithmen zunächst auf das standardmäßige Environment trainiert, sodass alle die Fähigkeit erlernen, die Stange auf dem Wagen balancieren zu lassen. Anschließend wird

betrachtet, wie die Agenten in einem Environment zurechtkommen, das mit den Rauschverfahren aus 1.4 versetzt worden ist. Dabei wird auf die Kombination aus mehreren Rauschverfahren verzichtet. Insgesamt werden vier Evaluationsiterationen pro Rauschverfahren durchgeführt, wobei pro Iteration der Grad des Rauschens stets erhöht wird. Schließlich werden die Agenten erneut trainiert, wobei dieses Mal die Rauschverfahren in den Trainingsprozess integriert werden, um zu überprüfen, ob dies die Performance in der verrauschten Simulation verbessert. Erneut wird in vier Evaluationsschritten untersucht, wie die verrauschten Agenten mit steigendem Rauschgrad zurechtkommen.

2.1 Versuchsaufbau

Rauschfreie Agenten werden 100, verrauschte Agenten 200 beziehungsweise 250 Epochen lang trainiert. Da die verrauschten Agenten anhand von Verteilungen lernen müssen, wurde die Epochenanzahl hierfür erhöht. Der Discount Faktor wird mit dem Wert 0.99 initialisiert. Das Input Layer sowie das Hidden Layer der DQNs werden mit der Größe 32 definiert. In der Ausgabeschicht kommt eine lineare Aktivierungsfunktion zum Einsatz, der Loss wird anhand des Mean Square Errors berechnet und für den eingesetzten Optimizer wird auf Adam zurückgegriffen. Die Schichten sind alle voll miteinander verbunden. Der Experience Replay Buffer verfügt über eine Größe von 1000 Tupeln. Die Größe der daraus entnommenen MiniBatches beträgt 32. Aktionen werden mit der ϵ-Greedy Methode gewählt, wobei ϵ anfangs auf den Wert 0.9 gesetzt wird und den Wert 0.01 nicht unterschreiten wird. Technisch wird auf das Python Framework TensorFlow mit dessen AI Erweiterung Keras zurückgegriffen.

Bei der Wahl der Rauschparameter wird folgendermaßen vorgegangen. Es werden jeweils Durchschnittswerte und deren Standardabweichungen für die einzelnen Rauschvariablen definiert, wobei die Durchschnittswerte konstant bleiben und die Standardabweichungen stetig erhöht werden, sodass das simulierte Rauschen pro Evaluation steigt. Alle angegebenen Werte sind als einheitslos zu verstehen, die in Relation zu den im Cartpole Environment angegebenen Simulationsparametern stehen. Die Standardabweichungen wurden so gewählt, dass jeweils der niedrigste Wert mindestens einen rauschfreien Agenten zum Scheitern an der Aufgabe bringen soll. Analog wird der oberste Wert so gewählt, dass ein signifikanter Performanceabfall ersichtlich wird. Die zwei Werte innerhalb des Bereiches interpolieren die Unter- und Obergrenzen.

2.2 Rauschfreie Agenten in rauschfreiem Environment

In 1 ist der Trainingsverlauf aller Agenten dargestellt. Nachdem die Agenten 100 Epochen lang trainiert worden sind, wurde ihre Performance 30 Epochen lang gemessen, wie in 2 zu sehen ist.

Tabelle 1. Training rauschfreier Agenten in rauschfreiem Environment

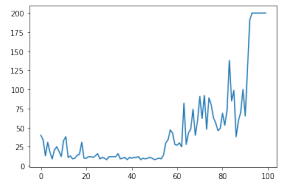
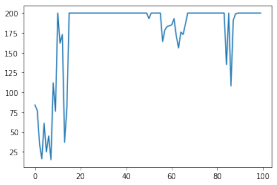
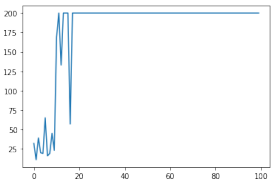
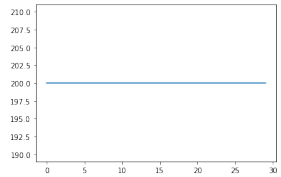
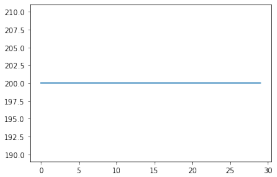
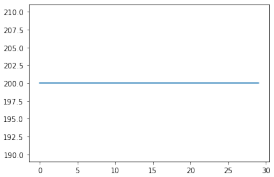
Q_ϕ	Q_ϕ-T_ϕ	Q_{ϕ_1}-Q_{ϕ_2}
		

Tabelle 2. Performance rauschfreier Agenten in rauschfreiem Environment

Q_ϕ	Q_ϕ-T_ϕ	Q_{ϕ_1}-Q_{ϕ_2}
		

2.3 Rauschfreie Agenten in verrauschtem Environment

Inwiefern die rauschfreien Agenten mit verrauschten Environments zurechtkommen, wird im folgenden Experiment gezeigt. Dabei werden die Rauschverfahren aus Kapitel 1.4 nachfolgend jeweils unabhängig voneinander betrachtet.

2.3.1 Observation Noise Im Fall von Observation Noise wird der durchschnittliche Offset des Sensorrauschens auf den Wert 0.0 gesetzt. Angelehnt an das initiale Rauschen $\mathcal{U}$(-0.05, 0.05) der simulierten Sensoren beträgt die Standardabweichung jeweils 0.015, 0.02, 0.025 und 0.03 Einheiten pro Evaluation. Anhand einer Normalverteilung basierend auf Durchschnitt und Standardabweichung werden Agentenobservationen verrauscht. In Tabelle 3 sind die Rewards der rauschfreien Agenten im Cartpole Environment abgebildet, wobei die Standardabweichung der Observation Noise stets erhöht wurde.

2.3.2 Domain Randomisation Die Wahl der Domain Randomisation Rauschparameter orientiert sich an den angegebenen Werten des Cartpole Environments und berücksichtigt jeweils die Masse des Cartpole Wagens (1.0 Einheiten) und der zu balancierenden Stange (0.1 Einheiten) sowie deren Länge (0.5 Einheiten) als Durchschnitt der Verteilung. Standardabweichungen werden als prozentuale Werte von 20%, 25%, 30% und 40% der Durchschnittswerte definiert. Das Rauschen wird hierbei auch einer Normalverteilung entnommen. In Tabelle 4 sind die Rewards der rauschfreien Agenten im Cartpole Environment abgebildet, wobei die Standardabweichung der Domain Randomisation stets erhöht wurde.

2.3.3 Random Force Injection Die einzige Kraft, die auf die Stange einwirkt, resultiert aus dem sich bewegenden Wagen. Deshalb wird der einzige Random Force Injection Parameter auf den aus dem Environment bekannten Kraftstärkewert von 10.0 als Durchschnitt der Verteilung gesetzt. Das Verrauschen der simulierten Schwerkraft von 9.81 ergäbe keinen Sinn, da diese als weltweit nahezu konstant angenommen wird. Wie in [3] wird hierbei das Rauschverhalten einer uniformen Verteilung entzogen. Daher wird jeweils der Offset vom Durchschnitt pro Evaluation erhöht. Dafür wurden die Werte 1.5, 2.0, 2.5 und 10.0 definiert. In Tabelle 5 sind die Rewards der rauschfreien Agenten im Cartpole Environment abgebildet, wobei die uniforme Verteilung der Random Force Injection stets breiter wurde. Der Median der Verteilung entspricht dem im Cartpole Environment angegebenen Kraftstärkewert von 10.

Tabelle 3. Performance rauschfreier Agenten in Observation Noise Environment

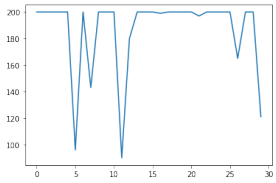
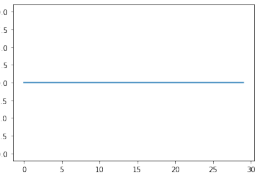
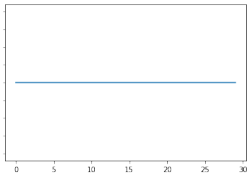
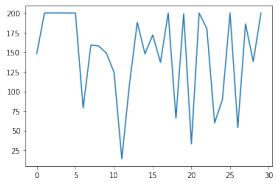
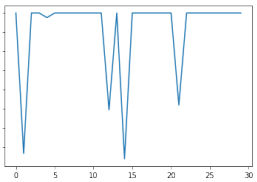
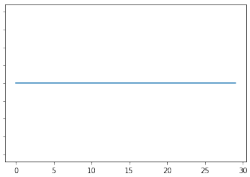
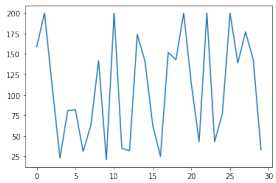
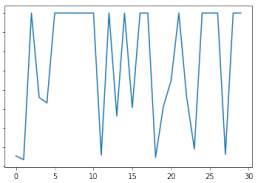
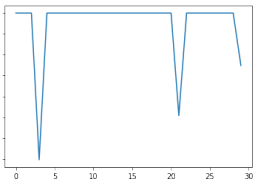
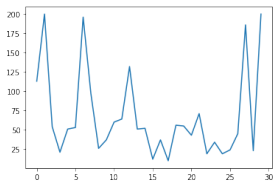
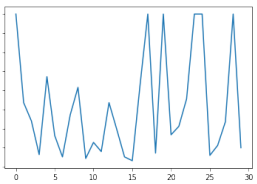
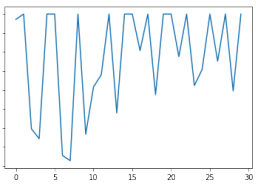
σ	Q_ϕ	Q_ϕ-T_ϕ	Q_{ϕ_1}-Q_{ϕ_2}
0.015			
0.020			
0.025			
0.030			

Tabelle 4. Performance rauschfreier Agenten in Domain Randomisation Environment

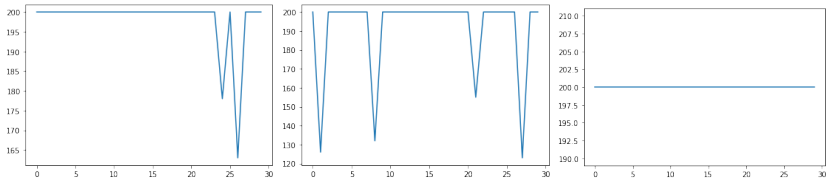
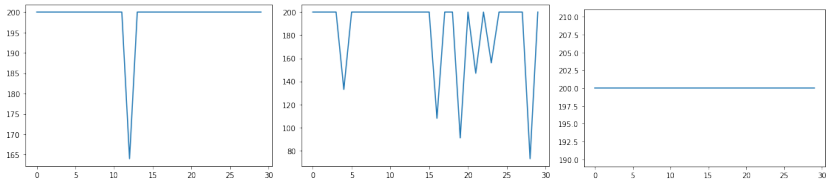
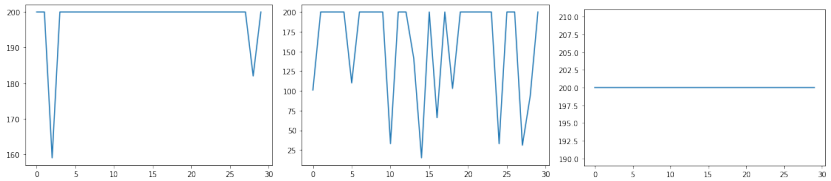
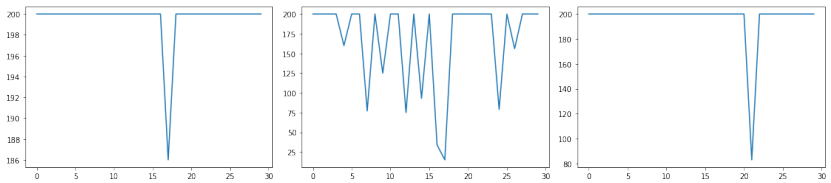
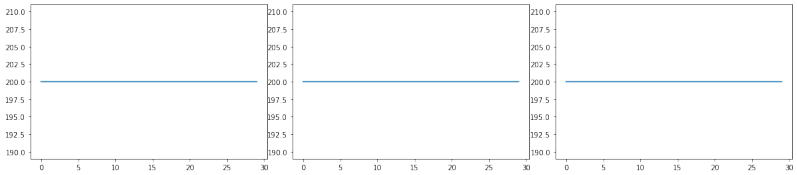
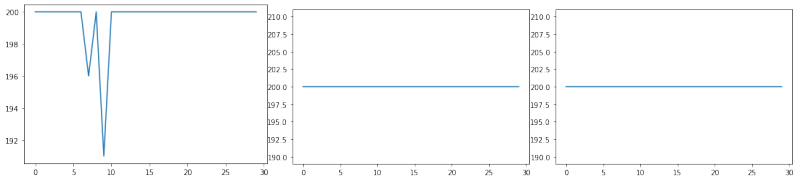
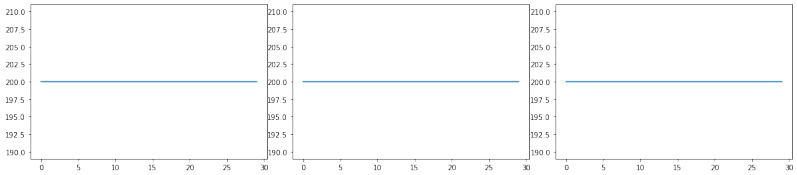
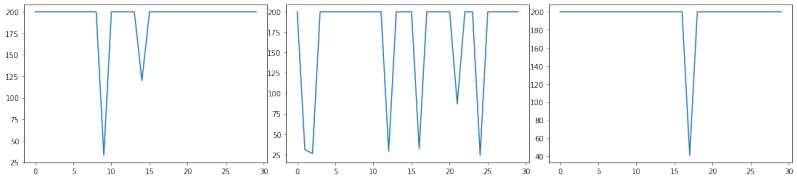
σ	Q_ϕ	Q_ϕ-T_ϕ	Q_{ϕ_1}-Q_{ϕ_2}
20%			
25%			
30%			
40%			

Tabelle 5. Performance rauschfreier Agenten in Random Force Injection Environment

$\mathcal{U}$	Q_ϕ	Q_ϕ-T_ϕ	Q_{ϕ_1}-Q_{ϕ_2}
(8.5, 11.5)			
(8.0, 12.0)			
(7.5, 12.5)			
(0.0, 20.0)			

2.4 Verrauschte Agenten in verrauschtem Environment

Anschließend wird ermittelt, inwiefern die verrauschten Agenten mit verrauschten Environments zurechtkommen. Dabei werden erneut die Rauschverfahren aus Kapitel 1.4 nachfolgend jeweils unabhängig voneinander betrachtet. Im Unterschied zu den rauschfreien Agenten werden die Agenten 200 beziehungsweise 250 Epochen lang trainiert, um aufgrund der Verteilung besser lernen zu können. Die integrierten Rauschparameter stimmen mit denen aus dem rauschfreien Experiment überein. In diesem Experiment wird jedoch nur noch die Performance in den Fällen mit der höchsten Standardabweichung betrachtet, wenn die rauschfreien Agenten signifikant damit fehlschlagen, die Aufgabe zu lösen. Denn Ziel dieses Experiments ist, das Fehlschlagen der Agenten in Situationen mit hoher Standardabweichung zu verhindern.

2.4.1 Observation Noise Um die Agenten besser auf die breite Observationsdomäne einzustellen, wurden die Agenten in diesem Experiment 200 Epochen lang trainiert. In Tabelle 6 ist der Trainingsverlauf der verrauschten Agenten in einem mit Observation Noise versetztem Cartpole Environment dargestellt. In Tabelle 7 ist die Performance der verrauschten Agenten in einem mit Observation Noise versetztem Cartpole Environment dargestellt.

2.4.2 Domain Randomisation Der Trainingsverlauf der verrauschten Agenten in einem mit Domain Randomisation versetztem Cartpole Environment ist in Tabelle 8 dargestellt. Es wurde hierbei 250 Epochen lang trainiert, in der Hoffnung, ein stabileres Lernergebnis zu erzielen. Weiterhin ist in Tabelle 9 die Performance der verrauschten Agenten in einem mit Domain Randomisation versetztem Cartpole Environment dargestellt.

2.4.3 Random Force Injection Da alle drei Agenten selbst mit unrealistisch hoher Random Force Injection gut zurechtkamen, wurde auf das Training der Agenten in einem damit verrauschtem Environment verzichtet.

Tabelle 6. Training verrauschter Agenten in Observation Noise Environment

Q_ϕ	Q_ϕ-T_ϕ	Q_{ϕ_1}-Q_{ϕ_2}

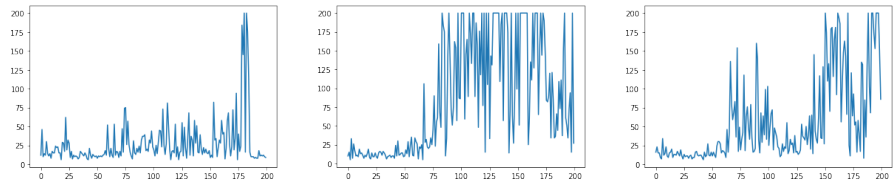


Tabelle 7. Performance verrauschter Agenten in Observation Noise Environment

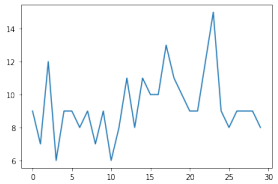
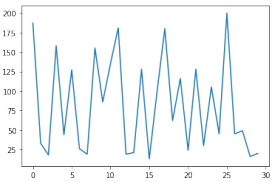
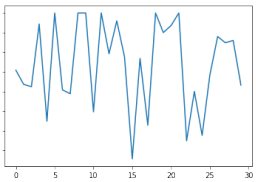
σ	Q_ϕ	Q_ϕ-T_ϕ	Q_{ϕ_1}-Q_{ϕ_2}
0.030			

Tabelle 8. Training verrauschter Agenten in Domain Randomisation Environment

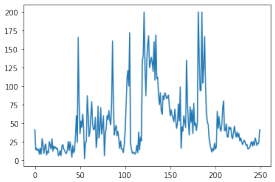
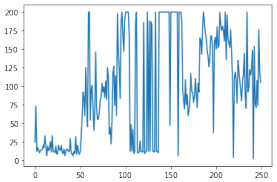
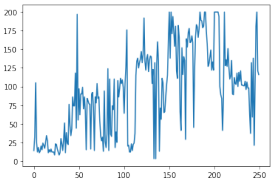
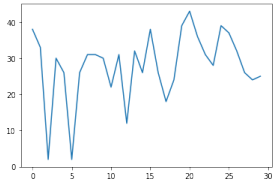
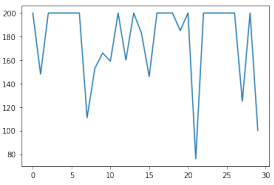
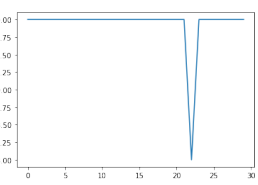
Q_ϕ	Q_ϕ-T_ϕ	Q_{ϕ_1}-Q_{ϕ_2}
		

Tabelle 9. Performance verrauschter Agenten in Domain Randomisation Environment

σ	Q_ϕ	Q_ϕ-T_ϕ	Q_{ϕ_1}-Q_{ϕ_2}
40%			

3 Ergebnisse

Generell lässt sich feststellen, dass die Sim-To-Sim Gap in den Experimenten deutlich erkennbar wurde. Die rauschfreien Agenten spürten den Einfluss bereits von wenig Rauschen, wohingegen die Integration des Rauschens in das Training nicht den erhofften Effekt hatte. Random Force Injection zeigte überraschenderweise keinen Einfluss auf die Performance der Agenten.

3.1 Rauschfreie Agenten in rauschfreiem Environment

Speziell das grundlegende DQN Q_ϕ brauchte viele Epochen, um $Q_*(S_t, A_t)$ zu erlernen, hat es letztlich aber dennoch geschafft. Die anderen zwei neuronalen Netze lernten um einiges rapider. Wie zu erwarten war, erreichten alle trainierten Agenten anschließend mindestens einen durchschnittlichen Reward von 195.

3.2 Rauschfreie Agenten in verrauschtem Environment

Anschließend werden die rauschfreien Agenten betrachtet, denen das Rauschen während des Trainingsvorganges verwehrt blieb.

3.2.1 Observation Noise Wie erkennbar ist, kam das grundlegende DQN Q_ϕ schon mit wenig Observation Noise schlecht zurecht, während die Performance des DQNs mit Target Netzwerk Q_ϕ-T_ϕ erst ab einer Standardabweichung von 0.02 zu sinken begann. Das DDQN Q_{ϕ_1}-Q_{ϕ_2} schien am widerstandsfähigsten zu sein und schlug erst signifikant bei einer Standardabweichung von 0.03 fehl.

3.2.2 Domain Randomisation Zweierlei Erkenntnisse lassen sich ableiten. Zunächst performte das DDQN Q_{ϕ_1}-Q_{ϕ_2} fast jedes Mal am besten. Interessanterweise lag das grundlegende DQN Q_ϕ bei einer Standardabweichung von 40% jedoch im Hinblick auf die Rewards ganz vorne. Auch lässt sich beobachten, dass das DQN mit Target Netzwerk Q_ϕ-T_ϕ am anfälligsten für Domain Randomisation war.

3.2.3 Random Force Injection Die ersten drei Trainings zeigten, dass das Rauschen kaum einen Einfluss auf die Agenten hatte. Daher wurde in der letzten Evaluation absichtlich die unrealistisch breite uniforme Verteilung $\mathcal{U}(0.0, 20.0)$ gewählt, welche aber nach wie vor keinen starken Einfluss auf die Agenten nahm.

3.3 Verrauschte Agenten in verrauschtem Environment

Nun werden die Agenten betrachtet, denen das Rauschverhalten der Simulation bereits während des Trainings bekannt war. Das Training mit Random Force Injection verrauschten Agenten wurde nicht ausgeführt, da kein signifikanter Performanceverlust bei den rauschfreien Agenten festgestellt werden konnte.

Die mit Observation Noise und Domain Randomisation verrauschten Agenten zeigten beide eine spürbar schlechtere Performance. Bereits während des Trainings wurde aufgrund des Lernprozesses erkenntlich, dass die verrauschten Agenten instabiler lernten als die Rauschfreien. Überraschend wurde auch die Anfälligkeit gegenüber dem Rauschen stärker als bei den rauschfreien Experimenten sichtbar. Insbesondere litt das Training von Q_ϕ unter dem Einfluss des Rauschens.

3.3.1 Observation Noise Q_ϕ-T_ϕ lernte schnell und wies einen stabileren Lernprozess als Q_ϕ auf. Q_{ϕ_1}-Q_{ϕ_2} lernte im Gegensatz dazu wiederum etwas langsamer und instabiler. Das grundlegende DQN Q_ϕ erreichte bei einer Standardabweichung von 0.03 nie das Ziel von 200 Zeiteinheiten. Die zwei anderen Agenten performten wesentlich besser, wobei diese ungefähr gleich häufig fehlschlugen. Im Vergleich zu 3 sticht unter anderem die schlechte Performance von Q_ϕ hervor, wobei auch die der anderen beiden Agenten sank.

3.3.2 Domain Randomisation Der Trainingsverlauf von Q_ϕ zeigte, dass etwas schneller und stabiler gelernt wurde als im Fall von Observation Noise. Q_ϕ-T_ϕ lernte zügig und weist erneut den stabilsten Lernprozess auf. Schließlich lernte Q_{ϕ_1}-Q_{ϕ_2} über etwa 200 Epochen recht linear dazu, während die restlichen 50 Epochen der Lernerfolg wieder abnahm. Das grundlegende DQN Q_ϕ performte etwas besser als das im Observation Noise Environment, was anhand des Lernprozesses auch zu erwarten war. Überraschenderweise hatte Q_ϕ-T_ϕ größere Probleme mit dem verrauschten Environment als Q_{ϕ_1}-Q_{ϕ_2}. Dennoch scheint es besser mit Domain Randomisation zurechtzukommen als mit Observation Noise. Dies trifft auch auf Q_{ϕ_1}-Q_{ϕ_2}, das lediglich einmal daran scheiterte, einen Reward von 200 zu erreichen. Im Vergleich zur Perfomance aus 4 scheint sich hier die Integration von Domain Randomisation in das Training nicht gelohnt zu haben, da die rauschfreien Q_ϕ sowie Q_ϕ-T_ϕ höhere Rewards erreichten und die Performance von Q_{ϕ_1}-Q_{ϕ_2} konstant blieb.

4 Diskussion

Drei Besonderheiten kamen in den Resultaten zum Vorschein, welche im Folgenden diskutiert werden. Dabei dreht es sich um die unerwartet hohe Widerstandsfähigkeit von Q_{ϕ_1}-Q_{ϕ_2} gegenüber dem Rauschen, Random Force Injection als ineffektives Mittel zur Bewältigung des Policy Transfers sowie die deutlich abfallende Performance der verrauschten Agenten im Hinblick auf ihre rauschfreien Äquivalente.

4.1 Beste Performance des DDQNs

Sowohl die rauschfreien als auch die verrauschten Ergebnisse offenbaren, dass Q_{ϕ_1}-Q_{ϕ_2} verglichen mit den anderen zwei Agenten oft die größten Rewards generiert. Lediglich der rauschfreie Q_ϕ Agent performte im Domain Randomisation

Environment im Experiment mit der höchsten Standardabweichung von 40% besser als das DDQN. Erklären ließe sich dieser Umstand mit dem stabilsten Trainingsvorgang des rauschfreien DDQNs. Allerdings widersprechen dieser Theorie Tabelle 6 und 8 augenscheinlich, da hier jeweils Q_ϕ-T_ϕ stabiler lernte. Eine Erklärung hierfür könnte in den dem Training zugrundeliegenden Verteilungen liegen, da möglicherweise das DDQN jeweils besser auf die Verteilungen trainiert wurde, welche in der Evaluation letztlich angewandt wurden.

4.2 Kaum Einfluss durch Random Force Injection

Die Ergebnisse aus Tabelle 5 stellen eine weitere Auffälligkeit zur Schau. Selbst eine unrealistisch breite uniforme Verteilung, anhand derer der Kraftstärkewert entnommen wurde, hatte nahezu keinen Einfluss auf die rauschfreien Agenten. Offenbar scheinen diese kaum abhängig davon zu sein, wie stark eine Kraft auf den Wagen einwirkt. Ein Blick in den Code offenbart, dass durch die physikalischen Berechnungen, die anhand des Kraftstärkewertes durchgeführt werden, eine weitaus kleinere Kraft letztlich auf den Wagen angewandt wird, als von der Verteilung entnommen. Anders formuliert hätte die Verteilung vermutlich stark verbreitert werden müssen, um den gewünschten Effekt hervorzurufen.

4.3 Schlechtere Performance der verrauschten Agenten

Schlussendlich fällt der Performanceabfall aller verrauschten Agenten ins Auge. Während es das Ziel der Rauschintegration in das Training war, die Widerstandsfähigkeit gegen das Rauschen zu verbessern, konnte kein verrauschter Agent in keinem verrauschten Environment davon profitieren. Im Gegenteil nahmen die Rewards nach dem verrauschten Training sogar ab. Grundlegend ist die Integration von Rauschen in den Trainingsprozess ein effektives Mittel beim Durchführen eines Policy Transfers, wie in der angegebenen Literatur auch einschlägig bewiesen worden ist. Womöglich liegt es an der Epochenanzahl, weshalb die Widerstandsfähigkeit der Agenten abnahm. In den Experimenten konnten die Agenten vermutlich nicht auf alle Verteilungen eingestellt werden, sodass einige Observationen oder Systemparameter ihnen so fremd waren, dass es zum Fehlschlag der Aufgabe kam.

5 Zusammenfassung

In dieser Arbeit wurde die Sim-To-Sim Gap des OpenAI Cartpole Environments untersucht. Dafür wurden zunächst drei Agenten jeweils basierend auf einem Deep-Q-Network, einem Target-Q-Network und einem Double Deep-Q-Network anhand des standardmäßigen Environments erfolgreich trainiert. Dann wurde eine weitere Simulation definiert, welche über andere Systemparameter verfügte als die Simulation, die für das Training benutzt worden ist. Dafür wurde auf die drei aus der Literatur bekannten Rauschmethoden Observation Noise, Domain

Randomisation und Random Force Injection zurückgegriffen. Der Effekt von Observation Noise und Domain Randomisation auf die rauschfreien Agenten wurde in Experimenten nachgewiesen. Andererseits änderte Random Force Injection wenig an der Performance, da der dafür modifizierte Parameter im Cartpole Environment nach Anwendung des physikalischen Bewegungsmodells sehr klein wird. Anschließend sollten die Agenten widerstandsfähiger gegenüber Observation Noise und Domain Randomisation gemacht werden, indem das Rauschen in den Trainingsprozess integriert wurde. Dieser Versuch schlug jedoch vermutlich deshalb fehl, da das Verhältnis zwischen Streuung der Rauschparameterverteilung und Trainingsepochenanzahl unausgewogen war. Eine weitere Erkenntnis ist, dass DDQNs im Vergleich zu DQNs und Target-DQNs generell am robustesten in verrauschten Environments sind.

Literaturverzeichnis

[1] F. Muratore, M. Gienger, and J. Peters, "Assessing transferability from simulation to reality for reinforcement learning," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 4, pp. 1172–1183, 2021.

[2] E. Salvato, G. Fenu, E. Medvet, and F. A. Pellegrino, "Crossing the reality gap: A survey on sim-to-real transferability of robot controllers in reinforcement learning," *IEEE Access*, vol. 9, pp. 153 171–153 187, 2021.

[3] E. Valassakis, Z. Ding, and E. Johns, "Crossing the gap: A deep dive into zero-shot sim-to-real transfer for dynamics," *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5372–5379, 2020.

[4] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 23–30.

[5] Q. H. Vuong, S. Vikram, H. Su, S. Gao, and H. I. Christensen, "How to pick the domain randomization parameters for sim-to-real transfer of reinforcement learning policies?" *ArXiv*, vol. abs/1903.11774, 2019.

[6] R. B. Slaoui, W. R. Clements, J. N. Foerster, and S. Toth, "Robust domain randomization for reinforcement learning," 2019.

[7] N. Jakobi, P. Husbands, and I. Harvey, "Noise and the reality gap: The use of simulation in evolutionary robotics," in *Advances in Artificial Life*, F. Morán, A. Moreno, J. J. Merelo, and P. Chacón, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 704–720.

[8] S. Koos, J.-B. Mouret, and S. Doncieux, "The transferability approach: Crossing the reality gap in evolutionary robotics," *IEEE Transactions on Evolutionary Computation*, vol. 17, no. 1, pp. 122–145, 2013.

[9] Y. Chebotar, A. Handa, V. Makoviychuk, M. Macklin, J. Issac, N. Ratliff, and D. Fox, "Closing the sim-to-real loop: Adapting simulation randomization with real world experience," in *2019 International Conference on Robotics and Automation (ICRA)*, 2019, pp. 8973–8979.

[10] R. Jeong, J. Kay, F. Romano, T. Lampe, T. Rothörl, A. Abdolmaleki, T. Erez, Y. Tassa, and F. Nori, "Modelling generalized forces with reinforcement learning for sim-to-real transfer," *ArXiv*, vol. abs/1910.09471, 2019.

[11] W. Zhao, J. P. Queralta, and T. Westerlund, "Sim-to-real transfer in deep reinforcement learning for robotics: a survey," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2020, pp. 737–744.

[12] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*, 1998.

[13] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3, pp. 279–292, 1992.

[14] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*, 1998.

[15] R. A. Jacobs, "Increased rates of convergence through learning rate adaptation," *Neural Networks*, vol. 1, no. 4, pp. 295–307, 1988.

[Online]. Available: https://www.sciencedirect.com/science/article/pii/0893608088900032
[16] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*, 1998.
[17] ——, *Reinforcement Learning: An Introduction*, 1998.
[18] L.-J. Lin, *Reinforcement learning for robots using neural networks*. Carnegie Mellon University, 1992.
[19] S. S. Mousavi, M. Schukat, and E. Howley, "Deep reinforcement learning: an overview," in *Proceedings of SAI Intelligent Systems Conference*. Springer, 2016, pp. 426–440.
[20] F. Tan, P. Yan, and X. Guan, "Deep reinforcement learning: from q-learning to deep q-learning," in *International Conference on Neural Information Processing*. Springer, 2017, pp. 475–483.
[21] S. Kim, K. Asadi, M. L. Littman, and G. D. Konidaris, "Removing the target network from deep q-networks with the mellowmax operator." in *AAMAS*, 2019, pp. 2060–2062.
[22] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.
[23] F. Mohd-Yasin, D. J. Nagel, and C. E. Korman, "Noise in mems," *Measurement Science and Technology*, vol. 21, no. 1, p. 012001, 2009.