Offline Reinforcement Learning: Vergleich der Performance von Algorithmen bei Verwendung unterschiedlicher Datensätze am Beispiel eines Roboterarms

Pamela Kunert

HAW Hamburg, Berliner Tor 5, 20099 Hamburg, Germany
Pamela.Kunert@haw-hamburg.de

Zusammenfassung. Offline Reinforcement Learning, also das Erlernen einer Policy auf Basis eines statischen Datensatzes ohne Interaktion mit der Umwelt, verspricht neue Anwendungsfelder zu erschließen und besser zu skalieren als gewöhnliche Reinforcement Algorithmen.
Am Beispiel eines Roboterarms, welcher lernen soll, einen Zielpunkt zu erreichen, werden in dieser Arbeit unterschiedliche Algorithmen sowohl aus dem Bereich des off-Policy Reinforcement Learnings als auch aus dem Offline Reinforcement Learning zunächst erläutert und schließlich im Offline-Modus erprobt und verglichen.
Um ebenso den Einfluss des verwendeten Datensatzes beurteilen zu können, werden fünf unterschiedliche Datensätze zum Erlernen einer Policy verwendet: der erste Datensatz wird mit zufällig gewählten Aktionen erzeugt, der zweite wird während eines Reinforcement Learning Trainingsprozesses erfasst, während der dritte mit einer einheitlichen Policy erzeugt wird. Ergänzend werden zwei weitere Datensätze hinzugezogen: ein um das zehnfache vergrößerter Datensatz, sowie ein Datensatz, welcher mit optimierter Policy erzeugt wird.
Die Ergebnisse zeigen, dass je nach vorhandenem Datensatz unterschiedliche Algorithmen zu bevorzugen sind; sie machen jedoch auch deutlich, dass Offline-Algorithmen denen des off-Policy Reinforcement Learning ohne Interaktion mit der Umwelt signifikant überlegen sind, indem sie bekannte Herausforderungen im jeweiligen Algorithmus adressieren.

1 Einführung

Reinforcement Learning (RL), zu deutsch auch verstärkendes Lernen, ist neben überwachtem und unüberwachtem Lernen ein Teil des maschinellen Lernens (ML). Dabei grenzt RL sich insofern von den beiden anderen Teilgebieten ab, als dass es hier keinen vorbestimmten Soll-Output gibt, der erlernt werden soll, sondern das Lernen wird gefördert, indem Belohnungen für gutes Verhalten ausgesprochen werden. Klassischerweise betrachtet man beim RL einen Agenten, der mit einer Umwelt interagiert, meist in mehreren Schritten (sequenziell), und dabei entweder direkt, nach einer Anzahl von Aktionen oder zum Ende einer

Episode eine positive oder negative Belohnung erhält. Diese zu Maximieren ist das Ziel des Agenten, sodass die Belohnungsfunktion maßgeblich das Verhalten des Agenten beeinflusst. Die Interaktion mit der Umwelt erfolgt dabei durch eine Policy des Agenten, welche den Zuständen bestimmte Aktionen (deterministisch oder stochastisch) zuweist, und damit das Verhalten des Agenten repräsentiert.[18]
Der Agent handelt also entsprechend seiner Policy und erhält für sein Handeln eine Belohnung. Um sein Verhalten zu optimieren, wird die Policy evaluiert und entsprechend optimiert, um die Belohnung zu erhöhen - dies nennt man on-Policy RL, da der Agent entsprechend seiner eigenen Policy mit der Umgebung interagiert und daraus lernt. Im Gegensatz dazu gibt es auch off-Policy Lernverfahren, bei denen der Agent Daten erhält, die mit einer anderen, ggfs. früheren Policy gesammelt wurden. Die mit den früheren Policies gesammelten Daten werden in einem Buffer gespeichert und für jedes Update der Policy können alle Daten (beruhend auf vielen Policies) verwendet werden.
Das durch die Interaktion eines Agenten mit der Umwelt geprägte aktive Lernparadigma des RL kann auch eine Herausforderung darstellen, eben da wo diese Exploration der Umwelt aufwendig, teuer oder gefährlich ist, wie beispielsweise beim autonomen Fahren: einem Auto mithilfe eines RL-Algorithmus das Agieren im Verkehr beizubringen wäre gerade zu Anfang mit vielen Unfällen verbunden und ist dadurch undenkbar.
Im besten Falle gibt es, wie beim autonomen Fahren auch, für solche Anwendungsfälle Simulationsumgebungen, in denen der Algorithmus gefahrenlos erprobt werden kann - oft aber eben auch nicht, oder die Entwicklung einer Simulationsumgebung ist schlicht zu aufwendig.
Die medizinische Behandlung ist ein gutes Beispiel für einen Anwendungsfall, in welchem eine Simulation nicht möglich, und eine Exploration undenkbar wäre: kein Arzt würde seinem Patienten willkürlich Behandlungen oder Medikamente verschreiben, um zu erforschen, wie diese sich bei gegebenen Symptomen oder Beschwerden auswirken könnten. Nichtsdestotrotz gibt es hier und eben auch bei vielen anderen Anwendungsfällen, wie dem autonomen Fahren, viele historische Daten, auf die man prinzipiell zurückgreifen kann. Genau darum geht es beim Offline Reinforcement Learning: Aus einem bestehenden Datensatz, ohne zusätzliche Interaktion mit der Umwelt, eine Policy für den Agenten erlernen - dabei geht es aber nicht um eine Imitation der zur Generierung des Datensatzes verwendeten Policy, sondern darum, die Daten so zu nutzen, um eine noch bessere Policy daraus zu erlernen.[11]
Ziel dieser Arbeit ist es, den Leser mit einer Auswahl an Algorithmen vertraut zu machen, welche sich zur Anwendung bzgl. Offline Reinforcement Learning eignen, und diese auf unterschiedliche Datensätze anzuwenden, um so ihre Stärken und Schwächen analysieren zu können und daraus Erkenntnisse sowohl hinsichtlich der Beschaffenheit der für Offline RL benötigten Daten als auch der Auswahl eines Algorithmus abzuleiten.
Anhand eines konkreten Anwendungsfalls, nämlich dem Erreichen eines Zielpunktes durch einen Roboterarm, sollen im Zuge dieser Arbeit Herausforde-

rungen von eigentlich für den off-Policy Fall entwickelte Algorithmen deutlich gemacht werden und untersucht werden, ob Offline Algorithmen diese Hürden überwinden können. Des weiteren sollen konkrete Algorithmen und ihre Performance verglichen werden, wobei sowohl solche zum Einsatz kommen, die einen diskreten Aktionsraum verwenden, als auch solche, die mit kontinuierlichen Aktionen arbeiten.
Durch die Verwendung von unterschiedlich beschaffenen Datensätzen soll die Bedeutung der Güte des Trainingsdatensatzes untersucht werden, sodass der Leser in die Lage versetzt wird, die vielversprechendste Kombination aus Datensatz und Algorithmus zu wählen.
Im weiteren Verlauf dieser Arbeit wird dazu zunächst eine Einordnung in verwandte Arbeiten und den aktuellen Stand der Forschung vorgenommen. Daraufhin wird in Kapitel 3 eine Einführung in das Thema Offline Reinforcement Learning gegeben. Eine Auswahl an bekannten Algorithmen, sowohl aus dem Bereich des off-Policy als auch des Offline RL, werden gemeinsam mit dem Behavioral Cloning Algorithmus in Kapitel 4 vorgestellt.
Der Anwendungsfall des Roboterarms, der ein Ziel erreichen soll, wird mithilfe einer Simulationsumgebung erprobt, auf welche in Kapitel 5 eingegangen wird. Hier werden auch die verwendeten Datensätze sowie der Programmablauf der Modell-Trainings und -Evaluationen und Details der Implementierung der Experimente näher beschrieben.
In Kapitel 6 werden die Ergebnisse der vorher beschriebenen Experimente dargestellt und diskutiert, sodass abschließend die wichtigsten Erkenntnisse zusammengefasst werden können.

2 Verwandte Arbeiten und Stand der Forschung

Zunächst ist hier die parallel entstehende Arbeit im Zuge der Vorlesung Selbstoptimierende Systeme zu erwähnen: hier findet die Erprobung von Reinforcement Learning Algorithmen am UR5 Roboterarm, sowie der OpenAI Gym Reacher Simulationsumgebung statt. Aus dieser parallelen Arbeit stammen fast alle Datensätze (bis auf den randomisiert erzeugten): sie wurden mithilfe des Deep Deterministic Policy Gradient Algorithmus innerhalb der Simulationsumgebung erzeugt.
Viele nützliche Informationen sowohl zum UR5 als auch zur Gym-Reacher Simulationsumgebung finden sich in der Veröffentlichung von Mahmood et al.[13], in welcher ein ähnliches Ziel wie in der eben beschriebenen parallelen Arbeit verfolgt wird.
Im Bezug auf **Offline Reinforcement Learning** ist besonders die Veröffentlichung von Levine et al.[11] zu nennen, da sie als Tutorial konzipiert einen breiten Überblick über das Thema selbst gibt, inklusive einer Einordnung geeigneter Algorithmen, möglichen Anwendungsfeldern, besonderen Herausforderungen und offenen Fragestellungen in diesem Bereich.
Im Bezug auf den **Vergleich von unterschiedlichen Algorithmen** des off-Policy und Offline RLs ist vor allem die NeurIPS 2019 Workshop Veröffentlichung

von Fujimoto et al.[2] nah an der Zielsetzung dieser Ausarbeitung. Hier werden ebenfalls unterschiedliche Algorithmen in einer einheitlichen Umgebung, nämlich Atari-Spielen, erprobt um ihre Performance vergleichbar zu machen. Der verwendete Trainingsdatensatz umfasst 10 Millionen Transitionen und wurde mit einer teil-trainierten, Deep Q-Network-basierten Policy generiert. Die wichtigste Erkenntnis des Benchmarks ist, dass keiner der erprobtem Algorithmen unter den gegebenen Bedingungen die Performance eines online Agenten oder der datenerzeugenden Policy erreichen konnte.

3 Offline Reinforcement Learning

Von Offline RL, manachmal auch Batch RL genannt, spricht man, wenn man RL-Algorithmen auf einem statischen Datensatz anwendet, statt mit der Umwelt direkt zu interagieren. Levine et al. sprechen in ihrer Veröffentlichung [11] von einer datengetriebenen Formulierung des RL Problems, bei welchem - genau wie bei klassischem (on-Policy) RL - eine Policy gefunden werden soll, welche die zu erwartenden, aufsummierten Belohnungen maximiert, nur dass hierfür eben keine Policy „live“ mit der Umwelt evaluiert wird, sondern auf einen bestehenden, mit einer anderen Policy (π_β) gesammelten Datensatz zurückgegriffen wird. Dieser Datensatz besteht aus Transitionen, welche durch den Zustand s_t, die gewählte Aktion a_t, den Folge-Zustand s_{t+1} und die erhaltene Belohnung r_t beschrieben werden.

3.1 Motivation

Zwei Gründe für diese Herangehensweise wurden bereits in der Einführung erläutert: zum einen erschließen sich hierdurch neue Anwendungsgebiete, in denen eine Exploration vorher nicht möglich war. Zum anderen liegen solche Daten in großen Mengen vor bzw. können leicht beschafft werden, sodass ihre Nutzung sinnvoll scheint.
Ein weiterer Aspekt, der für Offline RL spricht, ist die Skalierbarkeit: im Bereich des Reinforcement Learnings müssen nach jedem Update der Policy neue Daten gesammelt werden (um die Policy zu evaluieren) - dies kostet Zeit, die man bei der Verwertung eines statischen Datensatzes nicht aufbringen muss.
Ein Stück weit scheint Offline RL damit eine Annäherung an die anderen Bereiche des Maschinellen Lernens zu sein, da diese statt Interaktion mit einer Umwelt eben auf einem großen Trainingsdatensatz basieren. Dabei existiert zwischen diesen Lernverfahren ein großer Unterschied: während man in anderen ML-Bereichen (und auch beim Imitation Learning) davon ausgeht, dass der Trainingsdatensatz bereits mithilfe von (fast-)optimalem Verhalten gesammelt wurde und damit das Ziel verfolgt, ein Modell eben auf dieses Verhalten zu trainieren, will man im RL aus dem Trainingsdatensatz ein **besseres Verhalten** extrahieren, also eine andere, bessere Policy finden, als diejenige, die bei der Datensammlung angewandt wurde. Dieser Unterschied birgt einige Herausforderungen, die im Folgenden näher beleuchtet werden sollen.

3.2 Herausforderungen

Beim Offline RL unterliegen also die Trainingsdaten einer anderen Verteilung als die daraus erlernte Policy, man spricht hier von einer Verteilungsverschiebung. Unter der erlernten Policy werden potentiell andere Zustände besucht, der Agent erreicht so Zustandsräume, die er im Training nie gesehen hat und die dadurch kaum kontrollierbar sind. Dies birgt die größte Gefahr beim Offline RL: der Agent kann in solchen Fällen nicht einschätzen, wie gut oder schlecht sein Verhalten ist, weil er eben keinerlei Informationen darüber hat und gerät ggfs. in Zustandsbereiche, die niedrige Belohnungen auszahlen, obwohl der Agent aufgrund der Verteilungsverschiebung dort hohe Belohnungen erwartet.[11] Man spricht hier auch von einem Extrapolationsfehler, da der Agent hier die Zustands- / Aktionsbereiche verlässt die ihm durch die Trainingsdaten bekannt sind und sich so außerhalb des gesicherten Bereichs befindet. Besonders fatal ist daran, dass sich diese Fehleinschätzungen aufgrund von unbekannten Zustandsbereichen fortpflanzen, da sie in weitere Prognosen der Zustands-Aktions-Werte einberechnet werden, sodass es hier unmöglich sein kann, eine gute Policy zu erlernen.[4]
Eine weitere Herausforderung kann die Qualität des Datensatzes darstellen: wenn der Datensatz nur einen kleinen Bereich des Zustandsraums abdeckt oder nur solche Zustandsbereiche, in denen niedrige Belohnungen ausgezahlt werden, kann darauf keine optimale Policy erlernt werden. Wie eigentlich immer im ML kann eben nur erlernt werden, was auch schon in den Daten ist. Daher liegt der Nutzung von Offline RL Algorithmen die Annahme zugrunde, dass der Datensatz die Transitionsbereiche mit hohen Belohnungen angemessen abdeckt.[11]

3.3 Evaluierung

Prinzipiell stellt beim Offline RL die Evaluierung der erlernten Policy ebenfalls eine große Herausforderung dar: Offline RL basiert auf der Idee, keine Interaktion mit der Umwelt zuzulassen - wie soll man also eine, auf Basis des statischen Datensatzes gefundene, Policy bewerten? Gerade in Anwendungsgebieten wie der Medizin, in der die Risikominimierung der treibende Faktor für eine solche Policy-Entwicklung ist, ist eine Erprobung an der Umwelt kaum vertretbar.
Da wo man doch die Möglichkeit hat, durch geeignete Schutzmaßnahmen (wie Kollisionsabfragen vor der Ausführung einer Bewegung eines Roboterarms beispielsweise) oder die Verwendung einer Simulation die Policy an einer Umwelt zu erproben, sollte dies genutzt werden, dies ist im hier gewählten und folgend beschriebenen Anwendungsfall gegeben.
In Anwendungsfällen, wo dies nicht möglich ist, können off-Policy Evaluierungsalgorithmen helfen, die Güte einer Policy zu bewerten.

4 Algorithmen

Da beim Offline RL keine Interaktion mit der Umwelt stattfindet, sondern ausschließlich auf den statischen Datensatz zurückgegriffen wird, lässt sich prinzipiell jeder off-Policy Algorithmus für Offline RL anwenden, indem die sonst übliche

Interaktion mit der Umwelt übergangen wird.[11]
Eine Auswahl an performanten off-Policy Algorithmen wird in diesem Kapitel vorgestellt, darauf folgend wird auf Algorithmen eingegangen, die nur für den Offline-Fall entwickelt wurden.
Bevor jedoch off-Policy und Offline Algorithmen näher erläutert werden sollen, wird mit dem Behavior Cloning-Algorithmus noch ein Verfahren dargestellt, was nicht (wie einleitend beschrieben) dazu dient, aus den mit einer Policy π_β gesammelten Daten eine bessere Policy zu berechnen, sondern darauf abzielt, genau dieses Verhalten zu kopieren. Die Anwendung dieses Verfahrens kann Hinweise darauf geben, ob der Datensatz prinzipiell das Erlernen einer Policy zulässt und kann für die anderen Algorithmen als Baseline gesehen werden, da die im Offline RL angewandten Algorithmen mindestens so gut sein sollten, wie die Policy mit der die Daten gesammelt wurden - sonst ist ihre Anwendung nicht sinnvoll, da bei Offline RL die Komplexität der Berechnungen[1] und damit auch die Trainingszeiten erhöht werden, welche nur durch eine Verbesserung der Performance gerechtfertigt wird.

4.1 Behavior Cloning

Beim Behavior Cloning (BC) soll die Policy π_β gelernt werden, die einem Datensatz aus Transitionen zugrunde liegt; es soll also von jedem Zustand s die zu wählende Aktion a gefunden werden. Um diese Abbildung von Zustand auf Aktion zu lernen, werden Verfahren des überwachten maschinellen Lernens verwendet, wie beispielsweise Support Vektor Maschinen oder Neuronale Netze.[15]

4.2 Deep Q-Network

Der Deep Q-Network Algorithmus (DQN) wurde 2015 von DeepMind-Mitarbeitern entwickelt [14] und basiert auf Q-learning, also dem Erlernen der Action-Value-Funktion Q(s,a), um daraus die optimale Policy abzuleiten. Die optimale Policy ist dann diejenige, bei welcher man hinsichtlich des Q-Werts *greedy* handelt, also immer die Aktion im Zustand s wählt, welche den höchsten Q-Wert aufweist. Für die Berechnung des Returns wird bereits von der Verfolgung dieser greedy Policy ausgegangen (in die Berechnung wird der Q-Wert des Folgezustands mit einbezogen, welcher durch die greedy Aktion erzielt wird), obwohl der Agent dieser greedy Policy (noch) nicht folgt, deshalb handelt es sich bei Q-Learning um off-Policy Algorithmen.
Prinzipiell erweitert DQN Q-Learning einerseits um einen sog. *Replay Buffer*, also einen Datensatz, der (im off-Policy Modus) über online Interaktionen mit der Umwelt gesammelt wird und dann verwendet wird, um daraus Transitionen zu sampeln. Beim Funktionsapproximator handelt es sich dann um ein Neuronales Netz, welches auf Basis eines Zustands die Q-Werte vorhersagen soll. Da hier jedes Output-Neuron für eine mögliche Ausprägung der Aktion steht (für

[1] Im Vergleich zum Behavioral Cloning werden bei den Offline Algorithmen mehrere Neuronale Netze unter Berücksichtigung von Belohnungen trainiert

welche der Q-Wert vorhergesagt werden soll), eignet sich der DQN Algorithmus nicht für kontinuierliche Aktionsräume.
Eine weitere Verbesserung der Performance des Algorithmus kann durch die Erweiterung um ein Target-Netz erfolgen: dieses stabilisiert den Lernerfolg, indem es als Kopie des eigentlichen Q-Netzes (Neuronales Netz zur Abbildung eines Zustands auf Q-Werte der Aktionen) dient und zur Berechnung des Q-Werts für den Folgezustand Q(s', a') verwendet wird.[10]
Bei der Adaption für die offline Anwendung dieses Algorithmus fungiert der Offline-Datensatz als Replay Buffer, sodass keine Interaktion mit der Umwelt mehr stattfindet.

4.3 Deep Deterministic Policy Gradient

Auch der Deep Deterministic Policy Gradient (DDPG) Algorithmus kommt von Mitarbeitern von DeepMind ([12], der Preprint ist jedoch von 2015) und basiert auf dem eben beschriebenen DQN, integriert dieses jedoch in ein Actor-Critic-Verfahren (basierend auf [17]), um es auf einen kontinuierlichen Aktionsraum anwendbar zu machen:
Der Actor ist hier ein Policy Netz ähnlich dem DQN, jedoch wird hier nicht ein Neuron pro Aktionsausprägung im Output-Layer bereitgestellt, sondern direkt die zu wählende Aktion in einem Output-Neuron. Es findet also eine direkte Abbildung vom Zustand auf die zu wählende (Q-maximierende) Aktion statt, der Aktionsraum kann hierdurch auch kontinuierlich sein.
Als Critic wird ein zweites Netz trainiert, welches auf Basis des Zustands s und der vom Policy Netz vorgeschlagenen Aktion a den zugehörigen Q-Wert berechnet.
Neben der Verwendung des Replay Buffers wird auch hier das Training stabilisiert, indem Target-Netze (jeweils eins für Actor und Critic) zur Berechnung für den Folgezustand hinzugezogen werden. Im Gegensatz zum DQN erfolgt im DDPG Algorithmus jedoch ein *soft Update* der Target-Netz-Gewichte, indem jeweils nur ein Teil der Gewichte übertragen wird.

4.4 Twin Delayed Deep Deterministic Policy Gradient

Die Veröffentlichung von Fujimoto et al.[3] adressiert das Problem der Überschätzung von Q-Werten bei Actor-Critic Verfahren, indem der Twin Delayed Deep Deterministic Policy Gradient (TD3) Algorithmus als off-Policy RL-Verfahren vorgeschlagen wird.
Der TD3 basiert im Wesentlichen auf dem DDPG und erweitert ihn um drei Anpassungen, um Stabilität und Performance zu erhöhen:

- Clipped Double Q-Learning: Es wird ein zweiter Critic hinzugefügt, um die Q-Funktion zu schätzen. In die Berechnungen des Returns wird dann der kleinere der beiden Q-Werte einbezogen, womit die Überschätzung reduziert werden soll.

- Verzögerte Policy Updates: Die Q-Funktion bzw. der Critic wird öfter angepasst als die Policy bzw. der Actor. Die Autoren argumentieren, dass die Q-Wert-Schätzungen damit geringerer Varianz unterliegen, sobald die Policy geupdatet wird, sodass diese Policy-Aktualisierungen qualitativ besser sind.
- Regularisierung: um Overfitting (durch die deterministische Policy) zu vermeiden, wird zu der Target-Policy ein Rauschen addiert

4.5 Soft Actor-Critic

Auch beim Soft Actor-Critic (SAC) Algorithmus [5] handelt es sich um ein off-Policy Actor-Critic Verfahren für kontinuierliche Zustands- und Aktionsräume, es basiert jedoch auf einem anderen Ansatz: dem Maximum Entropy Reinforcement Learning. Hierbei wird der Actor so modelliert, dass er sowohl den Return als auch die Entropie maximieren soll - also so zufällig wie möglich agiert. Diese Erweiterung zielt darauf ab, die Policy robust zu machen, indem unterschiedliche Verhalten und Strategien zur Lösung des RL Problems erlernt werden.

4.6 Batch-Constrained Q-Learning

Beim Batch-Contrained Q-Learning (BCQ) Algorithmus [4] handelt es sich um das erste Verfahren, das für Offline RL entworfen wurde und demenstprechend ganz ohne Interaktion mit der Umwelt auskommen soll. Entwickelt wurde es unter anderem von Scott Fujimoto, der auch den TD3 Algorithmus entworfen hat. Dieser argumentiert, dass die meisten off-Policy Algorithmen bei der offline Anwendung vor allem das Problem des Extrapolationsfehlers haben, bei welchem Zustands-Aktions-Paare, die nicht im Datensatz vorhanden sind, falsch bewertet und darauf basierend falsche Entscheidungen getroffen werden.
Um dieses Problem zu adressieren wird beim BCQ (neben der Maximierung des Returns) das Ziel gesetzt, den Unterschied zwischen den besuchten Zustands-Aktions-Paaren und denen im Datensatz so klein wie möglich zu halten, sodass diese Fehleinschätzung möglichst nicht erfolgt.
Statt also die Aktion zu wählen, in welcher der Q-Wert am größten ist, wird nur aus den Aktionen gewählt, die im Datensatz aus diesem Zustand gewählt würden. Dabei ist es unrealistisch, dass der Datensatz wirklich jeden Zustand und damit auch Aktionen aus diesem heraus abbildet, deshalb werden nicht direkt die Zustands-Aktions-Paare aus dem Datensatz berücksichtigt, sondern basierend auf ihnen ein generatives Modell (Variational Autoencoder) trainiert, welcher auf Basis eines Zustands die wahrscheinlichsten Aktionen ausgibt. Ein Perturbationsmodell berechnet auf Basis dieser ausgegebenen, wahrscheinlichen Aktionen die zu wählenden Aktionen, um (beschränkt) eine höhere Diversität an Aktionen zu ermöglichen (und so ggfs. Regionen mit höherer Belohnung zu erschließen).
Kombiniert wird das so entstandene Policy Modell (bestehend aus Aktionsgenerator und Perturbation) als Actor mit einer Modifikation des bereits beschriebenen Clipped Double Q-Learning als Critic, daher handelt es sich auch hier um einen Actor-Critic-basierten Algorithmus.

4.7 Bootstrapping Error Accumulation Reduction

Auch beim Bootstrapping Error Accumulation Reduction (BEAR) Algorithmus von Kumar et al. [7] handelt es sich um ein für Offline RL entwickeltes Verfahren, welches genau wie BCQ darauf abzielt, die Fehler auszumerzen, welche durch die Wahl von Aktionen außerhalb der im Datensatz vorkommenden Zustands-Aktions-Paare resultieren (diese heißen hier *bootstrapping error* und akkumulieren sich über die Bellman Gleichung, daher der Name des Algorithmus).
Während jedoch bei BCQ die Policy insofern beschränkt wird, als dass sie sich nicht zu stark von der Policy π_β (die während der Generierung des Datensatzes verwendet wurde) unterscheiden darf, erfolgt bei BEAR eine Beschränkung nur hinsichtlich des Vorkommens der Aktion im Trainingsdatensatz. Für einen Zustand s würde also bei BCQ eine Aktionsverteilung ähnlich der von π_β gefordert werden, während bei BEAR lediglich gefordert wird, dass die Policy innerhalb der Unterstützung der Verteilung liegt (deshalb spricht der Autor hier von Unterstützungsbeschränkung statt Policybeschränkung).[6] Durch diese Abschwächung der Beschränkung kann auch bei nicht-optimalen Trainingsdaten eine optimale (oder zumindest bessere) Policy gefunden werden, da größere Abweichungen von der ursprünglichen Policy π_β ermöglicht und trotzdem Fehleinschätzungen vermieden werden, indem Aktionen vermieden werden, die im Trainingsdatensatz so nicht vorkommen.
Auch hier wird das Policy Modell als Actor in einen Actor-Critic Algorithmus mit zwei (oder mehr) Q-Funktion-Modellen als Critic eingebunden.

4.8 Conservative Q-Learning

Conservative Q-Learning (CQL) ist ebenfalls von Kumar veröffentlicht worden [8] und kann sowohl als Q-Learning Algorithmus als auch in der Actor-Critic Variante umgesetzt werden. Kern von CQL ist das (offline) Erlernen der Untergrenze der Q-Funktion, sodass keine Fehler durch Überschätzung der Q-Werte entstehen. Diese Untergrenze der Q-Funktion erhält man einerseits durch die Minimierung der Q-Werte für die aktuelle Policy und andererseits durch die Maximierung der Q-Werte für den Trainingsdatensatz.

4.9 Zusammenfassung der Algorithmen

Die wichtigsten Eigenschaften der in diesem Kapitel vorgestellten Algorithmen sind in Tabelle 1 zur Übersicht dargestellt.

5 Anwendungsfall und Versuchsaufbau

Offline RL soll im Zuge dieser Ausarbeitung seine Anwendung an einem UR5 Roboterarm finden. Dieser soll lernen, sich zu einem zufällig generierten Zielpunkt im zweidimensionalen Raum (an einer Wand) zu bewegen, zur Vereinfachung

Tabelle 1: Übersicht Algorithmen

Kürzel	Name	Art	Charakterisierung	Aktionsraum
BC	Behavior Cloning	Imitation Learning	Überwachtes Lernen	kontinuierl.
DQN	Deep Q-Network	Off-Policy	Q-Learning mit Neuronalem Netz	diskret
DQN+T	Deep Q-Network mit Target-Netz	Off-Policy	Q-Learning mit Target-Netz (Neuronale Netze)	diskret
DDPG	Deep Deterministic Policy Gradient	Off-Policy	Actor-Critic mit Target-Netzen	kontinuierl.
TD3	Twin Delayed Deep Deterministic Policy Gradient	Off-Policy	Actor-Critic mit Clipped Double Q-Learning und verzögerten Updates der Policy	kontinuierl.
SAC	Soft Actor-Critic	Off-Policy	Actor-Critic mit maximaler Entropie	kontinuierl.
BCQ	Batch-Constrained Q-Learning	Offline	Actor-Critic mit Policy-Beschränkung und Clipped Double Q-Learning	kontinuierl.
BEAR	Bootstrapping Error Accumulation Reduction	Offline	Actor-Critic mit Unterstützungs-Beschränkung	kontinuierl.
CQL	Conservative Q-Learning	Offline	Q-Learning bzw. Actor-Critic mit Untergrenze der Q-Funktion	kontinuierl.

werden dazu nur zwei der Gelenke verwendet. In einer parallelen Arbeit soll dem Agenten diese Fähigkeit mithilfe von Reinforcement Learning beigebracht werden, während es Zielsetzung dieser Arbeit ist, die dabei generierten Daten mithilfe von Offline RL so zu nutzen, dass eine bessere Policy zur Zielerreichung gefunden werden kann. Dabei stehen sowohl Simulationsumgebungen (OpenAI Gym Reacher und ROS-Gazebo) als auch der Roboterarm selbst zur Verfügung, sodass die Evaluierung der Policys hierüber erfolgen kann.
Die Modellierung erfolgt über einen Markow-Entscheidungsprozess (MDP), der sich wie folgt zusammensetzt:

1. **Zustände:** die Zustände werden definiert über die Winkel der beiden Gelenke, sowie einem Vektor zwischen der Fingerspitze (des Arms) und dem Zielpunkt
2. **Aktionen:** pro Gelenk kann eine (kontinuierliche) Aktion gewählt werden, welche das Gelenk in beide Richtungen (positive und negative Werte) bewegen kann
3. **Transitionswahrscheinlichkeiten:** es handelt sich um eine deterministische Umwelt, bei der aus den Winkelbewegungen die neue Position errechnet wird
4. **Belohnungsfunktion:** die Belohnung setzt sich gleichermaßen aus zwei Komponenten zusammen: der negativen euklidischen Distanz zwischen Fingerspitze und Zielpunkt, sowie der negativen Summe der quadrierten Aktionen (sodass große Aktionen bestraft werden)
5. **Diskontfaktor:** der Diskontfaktor wird auf $\gamma = 0.99$ gesetzt

Um den Einstieg zu erleichtern, wird sowohl der Reinforcement Learning Algorithmus zur Sammlung von Daten der parallelen Arbeit als auch die Evaluierung der Offline RL Ergebnisse mithilfe der OpenAI Gym Reacher Umgebung [1] umgesetzt, da hier schnell und einfach Berechnungen durchgeführt und Daten erzeugt werden können. Die Adaption auf die andere Simulationsumgebung bzw. den Roboterarm wird in der anderen Arbeit behandelt, sodass im Zuge dieser Arbeit nur auf die Gym-Reacher Umgebung zurückgegriffen wird. Die Gym-Reacher Umgebung inklusive der Zustände und Aktionen wird im folgenden Abschnitt näher erläutert. Darauf folgt eine Erläuterung zu den unterschiedlichen Datensätzen, auf denen die Algorithmen für das Offline RL durchgeführt wurden. Im letzten Abschnitt dieses Kapitels wird der Programmablauf zur Durchführung der Experimente erläutert.

5.1 Gym-Reacher: Datengenerierung und Testumgebung

Die Reacher-v2 Umgebung von OpenAI Gym bietet als Umwelt eine Abstraktion der für den Roboterarm gewählten Aufgabe, da hier der Agent aus zwei Gelenken besteht und lernen soll, ein zufällig generiertes Ziel mithilfe von Winkelbewegungen zu erreichen.
Der Beobachtungsraum setzt sich aus elf Beobachtungen zusammen, die von der Umwelt zurückgegeben werden:

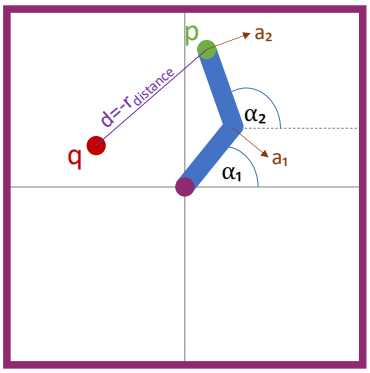


Abb. 1: Schematische Darstellung des Gym-Reachers: die zwei blauen Gelenkstangen bilden mit den horizontalen Hilfslinien die Gelenkwinkel α_1 und α_2. Eine Aktion besteht aus dem Anlegen zweier Kräfte an den Gelenkstangen (hier in die positive Richtung mit a_1 und a_2 dargestellt). Die Fingerspitze des Reachers ist als grüner Punkt p dargestellt, das zu erreichende Ziel ist als roter Punkt q markiert. Die euklidische Distanz d zwischen den zwei Punkten ist ebenfalls eingezeichnet.

- **Beobachtung 1 und 2:** der Kosinus der beiden Gelenkwinkel α_1 und α_2
- **Beobachtung 3 und 4:** der Sinus der beiden Gelenkwinkel α_1 und α_2
- **Beobachtung 5 und 6:** die x- und y-Koordinate der Fingerspitze p
- **Beobachtung 7 und 8:** die Geschwindigkeit der beiden Gelenkwinkel α_1 und α_2
- **Beobachtung 9, 10 und 11:** drei-dimensionaler Vektor von Fingerspitze p zum Zielpunkt q

Da es sich beim Sinus der beiden Gelenkwinkel, sowie der Position der Fingerspitze (bei gegebenen Cosinus-Werten) und auch bei der z-Koordinate des Vektors von Fingerspitze zum Zielpunkt um redundante Informationen handelt, sind diese (grau gefärbten) Beobachtungen nicht Teil des verwendeten Zustandsraums[2], um diesen möglichst klein zu halten und dem MDP in Abschnitt 5 zu entsprechen.
So wie der Zustandsraum ist auch der Aktionsraum in der Reacher-v2 Umgebung kontinuierlich: er besteht aus zwei Werten im Wertebereich von jeweils -1 und 1, wodurch das Drehmoment des Gelenks gesteuert wird. Damit handelt es sich nicht um eine reine Verschiebung der Gelenke um eine bestimmte Grad-Anzahl, sondern um das Anlegen einer Kraft, was wiederum ein „Nachrutschen“ der Positionen bedeutet.
Die von der Umwelt zurückgegebenen Belohnungen erfüllen die eingangs be-

[2] Der hier beschriebene gesamte Beobachtungsraum findet erst in einer Erweiterung der Datensätze Gebrauch, siehe dazu Abschnitt Datensätze und Implementierung

schriebene Belohnungsfunktion, welche sich wie folgt berechnen lässt:

$$\begin{aligned} r &= r_{distance} + r_{control} \\ &= -\sqrt{(q_x - p_x)^2 + (q_y - p_y)^2} - a_1^2 - a_2^2 \end{aligned} \tag{1}$$

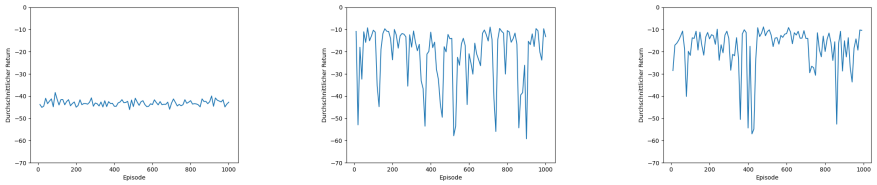
Dabei bezeichnet q_x bzw. q_y die x- bzw. y-Koordinate der Zielposition, p steht für die Position der Fingerspitze des Roboterarms und a_1 und a_2 sind die an den Gelenken angewandten Drehmomente.
Da in der Gym-Reacher Umgebung eine Episode nicht mit Erreichung der Zielposition, sondern nach Ablauf von 50 Schritten endet, wird der Gewinn als Summe der Belohnungen über die Schritte der Episode definiert.
Die in diesem Abschnitt beschriebene Gym-Reacher Umgebung wird sowohl zur Datengenerierung (in der parallelen Arbeit) als auch zur Evaluierung der hier angewandten Offline RL-Algorithmen angewandt. Letzteres geschieht, indem ein trainierter Agent zehn Mal mit der Gym-Reacher Umwelt interagiert und aus seinen erzielten Gewinnen der Mittelwert gebildet wird, um damit den Agent bzw. seine Policy zu bewerten.

5.2 Datensätze

Grundlage für die Anwendung der Offline RL-Algorithmen bilden zunächst drei Datensätze, von denen zwei durch das Trainieren und Agieren eines RL-Agenten mithilfe DDPG-Algorithmus[12] in der Reacher-Umwelt in der parallelen Arbeit entstanden sind: Der sogenannte *ddpg_train*-Datensatz ist dabei während des Trainierens des DDPG-Algorithmus entstanden und weist damit Wechsel in der Policy auf, während der *ddpg_control*-Datensatz mit einer, bereits trainierten, Policy erzeugt wurde. Der dritte initiale Datensatz nennt sich *random* und wurde durch zufällige Auswahl der Aktionen innerhalb der Umwelt erzeugt.
Alle drei Datensätze bestehen aus 1000 Episoden à 50 Schritte, also 50.000 Transitionen, die wiederum ein Tupel aus Zustand, Aktion, Folgezustand und Belohnung darstellen:

$$D = \{(s, a, s', r(s, a))\} \tag{2}$$

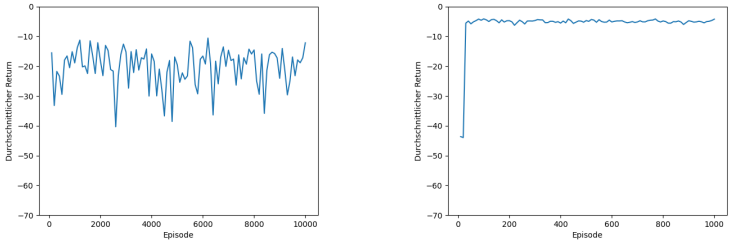
Dabei besteht jeder Zustand aus den in 5.1 beschriebenen fünf Beobachtungen (Kosinus des ersten und des zweiten Gelenkwinkels, Geschwindigkeiten der beiden Gelenke, Vektor zum Ziel), eine Aktion setzt sich aus zwei Werten zur Steuerung des Drehmoments der Gelenke zusammen und die Belohnung errechnet sich aus der Distanz zum Ziel und der gewählten Aktion.
Abbildung 2 zeigt den durchschnittlichen Gewinn, also die Summe der Belohnung innerhalb einer Episode im Episodenverlauf der initialen Datensätze.
Zur Ergänzung der Experimente werden zwei weitere Datensätze herangezogen, dabei handelt es sich einerseits um eine Vergrößerung des ddpg_control-Datensatzes: der *ddpg_control_big*-Datensatz ist mit 10-Tausend Episoden zehn Mal größer als der initiale Control-Datensatz. Andererseits wird mit Einbeziehung des gesamten in 5.1 beschriebenen Zustandsraums (aus elf Beobachtungen) ein optimierter DDPG-Algorithmus verwendet, um mit *ddpg_control_optimal* einen



(a) random (b) ddpg_train (c) ddpg_control

Abb. 2: Durchschnittliche Summe der Belohnungen über den Verlauf der Episoden für die initialen Datensätze (Durchschnitt aus je 10 Episoden)

Datensatz zu erzeugen, in dem durchschnittlich bessere Ergebnisse bzgl. Gewinn erzielt werden.



(a) ddpg_control_big (b) ddpg_conntrol_optimal

Abb. 3: Durchschnittliche Summe der Belohnungen über den Verlauf der Episoden für die erweiternden Datensätze (Durchschnitt aus 100 Episoden im großen Datensatz, bzw. 10 Episoden im optimalen)

Die durchschnittlichen Gewinne dieser erweiternden Datensätze sind in Abbildung 3 dargestellt, wobei für *ddpg_control_big* der Durchschnitt auf Basis von jeweils 100 Episoden gebildet wurde, um den Verlauf besser erkennbar zu machen.
Die wichtigsten Informationen zu den jeweiligen Datensätzen sind in Tabelle 2 zusammengefasst. Zu beachten ist dabei, dass auch der maximale Gewinn einen Durchschnittswert (auf Basis von zehn Episoden) abbildet, um zu vermeiden, dass ggfs. zufällig sehr gute Episoden (beispielsweise mit einem Zielpunkt, welcher sehr nah am Arm gesetzt wird) hier überbewertet werden. Der durchschnittliche Gewinn hingegen ist der Durchschnittswert über den gesamten Datensatz.

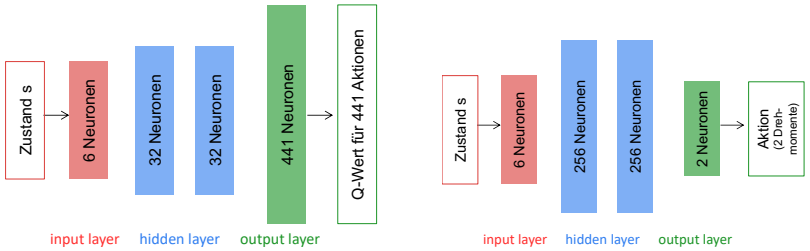
Tabelle 2: Charakterisierung der Datensätze

Datensatz	Anzahl Transitionen	Größe des Zustandsraums	Durchschnittl. Gewinn	Maximaler Gewinn
ddpg_random	50.000	6	-43,08	-38,38
ddpg_train	50.000	6	-21,48	-8,94
ddpg_control	50.000	6	-18,28	-8,85
ddpg_control_big	500.000	6	-20,28	-10,59
ddpg_control_optimal	50.000	11	-5,76	-4,15

5.3 Implementierung

Im Folgenden soll nun auf die Implementierung eingegangen werden. Dabei wird zwischen den Algorithmen bzgl. des Aktionsraums unterschieden, da für den kontinuierlichen Aktionsraum teilweise auf die Implementierung des DQN in der Vorlesung *Selbstoptimierende Systeme* zurückgegriffen wird. Für die Anwendung der anderen Algorithmen (BC, DDPG, TD3, SAC, BCQ, BEAR und CQL) wird das D3RLPY-Framework verwendet, sodass hier andere Vorverarbeitungsschritte und Evaluationsfrequenzen notwendig bzw. sinnvoll sind.

Diskreter Aktionsraum: DQN Für die Anwendung des DQNs muss der Aktionsraum zunächst diskretisiert werden. Dafür werden die Drehmomente der Aktion jeweils auf eine Nachkommastelle gerundet, wodurch sich 21 mögliche Ausprägungen für das Drehmoment ergeben (-1.0, -0.9, ... , 0.9, 1.0). Da sich eine Aktion aus zwei Drehmomenten (Steuerung beider Gelenke) zusammensetzt, entsteht so ein diskreter Aktionsraum aus 21 * 21 = 441 möglichen Aktionen: (-1.0 , -1.0), (-1.0 , -0.9) , (-1.0 , -0.8) , ... , (1.0 , 1.0).
Da es sich bei DQN um einen off-Policy Algorithmus handelt, der für Offline RL verwendet werden soll, erfolgt hier keine Interaktion mit der Umwelt, sondern die Daten werden direkt aus dem jeweiligen Datensatz genommen. Dabei wird durch zufällige Auswahl aus diesem ein Minibatch (bestehend aus 32 Transitionen) zum Trainieren des Q-Netzes erzeugt.
Das Q-Netz soll lernen, zu einem Zustand s den jeweiligen Return für die möglichen Aktionen zu bestimmen, um auf Basis dieser Vorhersage die Aktion wählen zu können, die den maximalen Return verspricht. Es ergibt sich die in Abbildung 4 links dargestellte Netzarchitektur, bestehend aus einer Eingabe-Schicht, welche dem Zustandsraum entspricht (bei den initialen Datensätzen = 6), zwei versteckten Schichten aus jeweils 32 Neuronen und der Ausgabe-Schicht aus 441 Neuronen, welche die Aktionen repräsentieren.
Das Neuronale Netz wird in einer Epoche mit dem (aus zufällig ausgewählten Transitionen) erzeugten Minibatch trainiert, ein Epoche besteht hier also nicht aus der Anwendung des gesamten Datensatzes, sondern nur des Minibatches.
Das Training des Q-Netzes besteht ingesamt aus 1000 Epochen, wobei alle 10 Epochen eine online Evalutaion in der Gym-Reacher-Umgebung erfolgt: hierbei wird das Modell für zehn Episoden als Agent eingesetzt und wählt dabei



(a) diskreter Aktionsraum (b) kontinuierlicher Aktionsraum

Abb. 4: Architektur der Neuronalen Netze

immer die Aktion mit dem höchsten vorhergesagten Q-Wert. Der durchschnittliche Gewinn der zehn Episoden wird gespeichert, um den Lernfortschritt zu dokumentieren. Es erfolgt keine Anpassung der Gewichte, da hier kein Lernerfolg sondern nur die Evaluierung durchgeführt werden soll. Der Ablauf ist in Abbildung 5 dargestellt.

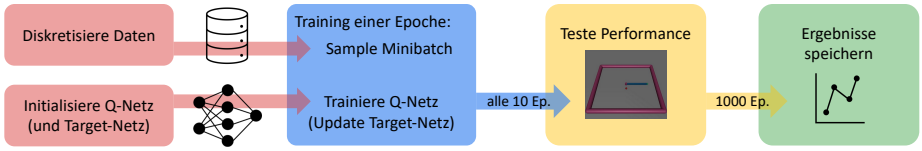


Abb. 5: Ablauf beim diskreten Aktionsraum

Kontinuierlicher Aktionsraum Für das Training der Algorithmen, die sich für die Umsetzung eines kontinuierlichen Aktionsraums eignen, wird auf das Framework D3RLPY von Takuma Seno [16] zurückgegriffen: hierbei handelt es sich um eine Programmbibliothek, die für Offline RL entwickelt wurde und eine Vielzahl an off-Policy und offline Algorithmen implementiert, unter anderem auch die in Kapitel 4 beschriebenen Verfahren BC, DDPG, TD3, SAC, BCQ, BEAR und CQL.
Da es sich, abgesehen vom Behavioral Cloning, bei diesen Verfahren um Actor-Critic Verfahren handelt[3], verfügen sie alle über ein Policy-Netz, welches vom Zustand s die Aktion a direkt ausgibt (und nicht für alle möglichen Aktionsausprägungen den Q-Wert wie beim diskreten Aktionsraum). Das Behavioral Cloning besteht ausschließlich aus diesem Policy Netz und wird darauf trainiert, entsprechend der Trainingsdaten den Zustand auf eine Aktion abzubilden.

[3] Auch bei der CQL Implementierung von D3RLPY handelt es sich um die Actor-Critic-Variante

Im hier betrachteten Anwendungsfall besteht eine Aktion jeweils aus zwei Drehmomenten, sodass sich auf der Ausgabe-Schicht aller Netze zwei Neuronen zur Ausgabe dieser (kontinuierlichen) Drehmoment-Werte befinden. Auf der Eingabe-Schicht findet sich die Größe des Zustandsraums als Anzahl für die Neuronen (6 bzw. 11) und durch die Verwendung der Standard-Konfiguration von D3RLPY befinden sich je 256 Neuronen auf den zwei versteckten Schichten, wodurch sich die in Abbildung 4 rechts dargestellte Netzarchitektur ergibt. Nicht dargestellt ist die Netzarchitektur der Critic-Netze, welche von Zustand und Aktion auf einen zugehörigen Q-Wert abbilden - sie folgt jedoch demselben Schema wie die Actor-Netze mit zwei versteckten Schichten zu je 256 Neuronen. Durch das Anlegen von Zustand und Aktion (konkateniert) ergeben sich acht Neuronen[4] auf der Eingabe-Schicht und ein Neuron auf der Ausgabe-Schicht für den zugehörigen Q-Wert.
Wie auch aus der Abbildung 6 ersichtlich wird, ähnelt der Ablauf für den kontinuierlichen Aktionsraum dem des diskreten prinzipiell, auf die Unterschiede soll folglich kurz eingegangen werden: Auch hier müssen die Daten vorverarbeitet werden, es findet eine Transformation in ein sog. *MDPDataset* statt, sodass die Modelle des Frameworks mit den Daten trainieren können. Die Initialisierung des Modells je nach Algorithmus findet über eine API-Anfrage statt und produziert die bereits beschriebene Netzarchitektur. Darauf folgt das Training einer Epoche, wobei hier Minibatch-weise der gesamte Datensatz zur Gewichtsanpassung der Netze Verwendung findet. Da hierdurch in einer Epoche viel größere Lernfortschritte erreicht werden können (da nicht nur mithilfe eines Minibatchs sondern des gesamten Datensatzes trainiert wurde), erfolgt die Evaluation der Performance jede Epoche. (Auch hier findet die Evaluation zwar in der Gym-Reacher Umgebung statt, jedoch ohne die Gewichte dabei anzupassen.) Um das Training möglichst kurz zu halten, wird der Lernfortschritt über die Epochen beobachtet und bei Stagnation (kein Überschreiten des Maximums in den letzten fünf Epochen) wird das Training beendet und die Ergebnisse gespeichert.

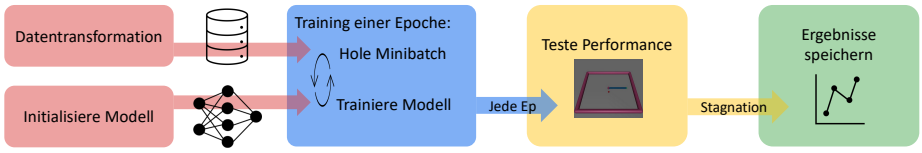


Abb. 6: Ablauf beim kontinuierlichen Aktionsraum unter Verwendung der D3RLPY-Bibliothek

[4] für den reduzierten Zustandsraum, beim ddpg_control_optimal mit erweitertem Zustandsraum sind es dementsprechend 13 Neuronen

6 Ergebnisse und Interpretation

6.1 Anwendung der drei initialen Datensätze

Wie im vorangegangenen Kapitel beschrieben werden die unterschiedlichen (in Kapitel 4 beschriebenen) Algorithmen zunächst auf den drei Datensätzen *random*, *ddpg_train* und *ddpg_control* trainiert und in bestimmten Abständen in der Gym-Reacher Umgebung evaluiert, indem ihr Gewinn als Summe der Belohnungen über zehn Episoden gemittelt wird. Die jeweils besten Ergebnisse der Lernkurve sind in Tabelle 3 dokumentiert, sowie die jeweils höchsten und die Durchschnittswerte, die bei der Datenerzeugung erzielt wurden (in grau). Die Kategorie bezeichnet dabei nur den ursprünglichen Zweck des Algorithmus, die Anwendung war in allen Fällen offline.

Tabelle 3: Durchschnittlicher Gewinn

Kategorie	Algorithmus	random	ddpg_train	ddpg_control
	Datensatz: bester	-38,379	-8,941	-8,854
	Datensatz: Durchschnitt	-43,084	-21,484	-18,282
Baseline	BC	-37,853	-10,999	-9,002
off-Policy	DQN	-8,749	-12,148	-9,512
	DQN+T	-7,737	-13,719	-8,824
	DDPG	-36,293	-38,030	-10,858
	TD3	-10,784	-10,279	-10,255
	SAC	-8,431	-8,684	-11,249
offline	BCQ	-19,457	-9,957	-8,367
	BEAR	-7,635	-8,711	-7,739
	CQL	-8,887	-8,774	-9,818

Aus diesen Ergebnissen lassen sich Erkenntnisse ziehen, welche im Folgenden näher beleuchtet werden.

BC Das BC nähert sich im Wesentlichen den besten Durchschnittswerten der Datensätze an - da hier die Abbildungen vom Zustand auf Aktion aus den Datensätzen gelernt werden sollen, entspricht dieses Ergebnis den Erwartungen. Auch die Tatsache, dass das BC beim *ddpg_train*-Datensatz eher weiter entfernt vom besten Ergebnis des Datensatzes liegt (-10,999 und -8,941) ergibt Sinn, da sich die Policy während des Trainings vielfach ändert und das BC keine konsistente Zustands-Aktions-Abbildung lernen kann, da es keine gibt.

DQN und TQN+T (diskreter Aktionsraum) Zunächst lässt sich feststellen, dass das Target-Netz dem DQN zu leicht besseren Ergebnissen verhilft, wodurch der DQN+T Algorithmus es für den *random*- und den *ddpg_control*-Datensatz über die Baseline des BC schafft. Beim *random*-Datensatz gelingt

dies besonders deutlich (-7,737 im Vergleich zu -37,853 beim BC bzw. -38,379 im Datensatz selbst), was dafür spricht, dass der Algorithmus in der Lage ist, eine bessere Policy als in der Datenerzeugung zu finden. Beachtlich ist dies vor allem dahingehend, dass hier der diskrete Aktionsraum verwendet wurde - diese Annäherung an den Aktionsraum scheint also durchaus zu funktionieren. Interessant wären hier weitere Versuche, in denen die hier als kontinuierlich verwendeten Algorithmen den diskreten Aktionsraum verwenden, um zu untersuchen, ob diese Vereinfachung zu Lernerfolgen beitragen kann.
Während die DQN-Modelle aber von der hohen Zustandsraum-Abdeckung des *random*-Datensatzes profitieren, beeinflussen die Policy-Wechsel im *ddpg_train*-Datensatz die Performance negativ: beide Modelle erreichen (mit -12,148 und -13,719) weder die Baseline (-10,999), noch die Bestwerte des Datensatzes (-8,941). Der Trainingsdatensatz scheint hier keine gute Grundlage zum Trainieren des Modells.

DDPG Der DDPG Algorithmus erzielt, verglichen mit den anderen Verfahren, die schlechtesten Ergebnisse (-36,293 und -38,03), einzig unter Verwendung des austrainierten Datensatzes *ddpg_control* liegt die Performance mit -10,858 nicht ganz so weit unter der der anderen Verfahren.
Eine Begründung für die schlechte Performance beim *random*- und *ddpg_train*-Datensatz könnte das Policy-Netz des DDPG liefern: dieses soll auf Basis der Daten den Zuständen direkt Aktionen zuweisen, das Prinzip ähnelt dem des BC. Zwar soll der Critic aus Zustand und Aktion den Q-Wert vorhersagen können und könnte dieses falsch trainierte Verhalten damit eigentlich eindämmen, ggfs. lernt dieser nicht schnell genug, um den Actor dahingehend zu beeinflussen. Dies könnte in weiteren Experimenten unter Anpassung der Lernraten von Actor und Critic untersucht werden.
Auch der Extrapolationsfehler könnte hier eine Begründung liefern: Zustands-Aktions-Paare die außerhalb des Datensatzes liegen und die damit verbundene Überschätzung von Q-Werten dürften bei den suboptimalen Datensätzen (hier *random*- und *ddpg_train*) viel häufiger vorkommen, da das Modell ja gerade ein anderes Verhalten als in diesen anstrebt. Da der TD3 Algorithmus eben dieses Problem der Überbewertung adressiert, lohnt sich ein Vergleich mit dessen Ergebnissen: tatsächlich erzielt dieser deutlich bessere Ergebnisse in den beiden betrachteten Datensätzen als der DDPG, was darauf hindeutet, dass der DDPG besonders bei suboptimalen Datensätzen einen Extrapolationsfehler aufweist und in der offline Verwendung damit eher ungeeignet ist.
Eine weitere Ursache könnte darin liegen, dass hier mit DDPG der Algorithmus auf die Daten angewandt wurde, welcher auch zur Erzeugung aller Datensätze verwendet wurde. Auch Fujimoto et al. machen diese Erfahrung in [4] und berichten davon, dass der off-Policy Agent viel schlechtere Ergebnisse erzielt als der ursprüngliche Agent, wenn sie mit demselben Algorithmus trainiert werden. Auch hier wird jedoch der DDPG verwendet, sodass zunächst untersucht werden muss, ob das Phänomen auch bei anderen Algorithmen auftritt oder ein DDPG-spezifisches Problem darstellt.

TD3 Wie bereits im vorangegangen Abschnitt angesprochen, erzielt der TD3 als Verbesserung des DDPGs mit Zielsetzung, Fehleinschätzungen der Q-Werte zu vermeiden, bessere Ergebnisse als der DDPG selbst - und das besonders da, wo der DDPG sehr schlecht abschneidet (*random* und *ddpg_train*). Dies spricht deutlich für die Adressierung dieses Problems, auch wenn die Ergebnisse besonders im *ddpg_control*-Datensatz (mit -10,255) darauf schließen lassen, dass der Fehler nicht ganz behoben werden kann, da die Performance hinter der Baseline des BC (mit -9,002) zurückbleibt.

SAC Der SAC Algorithmus erzielt mit -8,431 und -8,684 vergleichsweise gute Ergebnisse für den *random*- und *ddpg_train*-Datensatz, während er auf dem *ddpg_control*-Datensatz die schlechteste Performance aufweist. Hier profitiert der Ansatz der maximalen Entropie von der höheren Varianz der Zustands-Aktions-Paare in den beiden nicht fertig-trainierten Datensätzen.

BCQ Die BCQ Modelle erzielen relativ gute Ergebnisse auf dem *ddpg_train*- und *ddpg_control*-Datensatz, besonders auf letzterem: hier liegt der Bestwert mit -8,367 sowohl über der BC-Baseline (-9,002), als auch leicht über dem Bestwert des Datensatzes (-8,854). Als Verfahren, das für offline Bedingungen konstruiert wurde, indem es Policy-Beschränkung implementiert, um Fehleinschätzungen aufgrund ungesehener Zustands-Aktions-Paare zu verhindern, zeigt es einerseits, dass diese Policy-Beschränkung generell wirksam ist, aber andererseits auch, dass sie damit die Verbesserung einer Policy im Vergleich zur erzeugenden Policy behindert: auf dem *random*-Datensatz verzeichnet sie mit -19,457 einen vergleichsweise niedrigen Gewinn und auch die Policy-Wechsel im *ddpg_train*-Datensatz verhindern eine Annäherung an den Bestwert des Datensatzes.

BEAR Der BEAR Algorithmus verzeichnet insgesamt die besten Ergebnisse (-7,635; -8,711 und -7,739) und überschreitet in allen Fällen sowohl die Baseline als auch die Bestwerte der Datensätze. Die Abschwächung der Policy-Beschränkung zu einer Unterstützungsbeschränkung ist effektiv und profitiert beim *random*-Datensatz sogar von der hohen Varianz, sodass hier eine deutliche Verbesserung der Policy möglich ist.

CQL Die CQL Modelle erzielen mit -8,887 und -8,774 die besseren Ergebnisse auf den nicht aus-trainierten Datensätzen, während das Ergebnis auf Basis des *ddpg_control*-Datensatzes mit -9,818 hinter der Baseline bleibt. Dies deutet darauf hin, dass sich die Untergrenze für die Q-Werte besser aus Datensätzen ableiten lässt, die eine höhere Varianz bzgl. der Zustands-Aktions-Paare aufweisen.

Alle Verfahren Hervorzuheben ist wohl vor allem die Erkenntnis, dass sich auch auf Daten, die mit einer suboptimalen Policy gesammelt wurden, prinzipiell eine bessere Policy erlernen lässt - am eindrucksvollsten wird dies am Beispiel

des zufällig generierten Datensatzes deutlich.
Die Vermutung, dass eigens für die offline Anwendung entwickelte Algorithmen eine höhere Performance aufweisen, hat sich bestätigt.
Nichtsdestotrotz bleibt die Performance (vor allem von den Offline Verfahren) hinter den Erwartungen, vor allem wenn man sie mit Benchmarks zur Gym-Reacher Aufgabe vergleicht: hier [9] bewegen sich die Gewinne je nach verwendetem Verfahren fast ausschließlich im Bereich von -6,71 bis -1,5; Werte also, mit denen sich die hier im Offline-Modus erzielten Ergebnisse kaum messen können.
Da jedoch bereits eine Vielzahl an Algorithmen erprobt wurden, wird die Suche nach Ursachen für die vergleichsweise schlechten Ergebnisse in Richtung der Daten gelenkt. Zwei mögliche Probleme bzgl. der Daten werden identifiziert: zum einen kann eine zu geringe Datenmenge den Lernerfolg behindern - um dies zu untersuchen werden die Algorithmen auf dem *ddpg_control_big* Datensatz (siehe 5.2) trainiert. Zum anderen weisen die Datensätze selbst schon eine vergleichsweise schlechte Performance auf (*ddpg_control* liegt bei einem durchschnittlichen Gewinn von -18,282 und erzielt auch in den durchschnittlich besten Episoden -8,854), was natürlich die Performance der darauf trainierten Modelle beeinflusst. Daher erfolgt ein weiterer Durchlauf der Algorithmen unter Verwendung des *ddpg_control_optimal*-Datensatzes, welcher mit einer besseren Policy generiert wurde.
Diese zwei Datensätze bilden die Erweiterung, deren Ergebnisse im nächsten Abschnitt diskutiert werden.

6.2 Anwendung der erweiternden Datensätze

Die zwei erweiternden Datensätze *ddpg_control_big* und *ddpg_control_optimal* werden analog zum vorgestellten Ablauf mit den ausgewählten Algorithmen angewandt und die jeweils beste durchschnittliche Episode bzgl. ihrer aufsummierten Belohnungen in Tabelle 4 festgehalten. Dabei sind die Gewinne grün markiert, die eine Verbesserung der Performance des Algorithmus im Vergleich zur Verwendung des *ddpg_control*-Datensatzes (vgl. Tabelle 3) darstellen.

Vergrößerung des Datensatzes Die Vergößerung des Datensatzes um das 10-fache an Transitionen scheint für viele Algorithmen hilfreich: DQN, SAC, BEAR und CQL verbessern ihre Performance, jedoch ist diese Verbesserung vor allem bei SAC und BEAR signifikant. Nur bei DDPG kann man bei einem besten durchschnittlichen Gewinn von -73,454 von einer signifikanten Verschlechterung der Performance sprechen - ob dies wirklich durch die Vergößerung des Datensatzes bedingt wurde, müsste in weiteren Experimenten verifiziert werden (denkbar wäre auch eine ungünstige Initialisierung der Start-Gewichte).

Verbesserung der erzeugenden Policy Diese Maßnahme resultiert zwar fast ausschließlich in besseren Performances im Vergleich zum ursprünglichen *ddpg_control*-Datensatz, jedoch schafft es hier kein Modell über die Baseline des

Tabelle 4: Durchschnittlicher Gewinn bei erweiternden Datensätzen (grün: Verbesserung der Algorithmus-Performance)

Kategorie	Algorithmus	ddpg_control_big	ddpg_control_optimal
	Datensatz: bester	-10,589	-4,145
	Datensatz: Durchschnitt	-20,283	-5,761
Baseline	BC	-9,260	-3,109
off-Policy	DQN	-8,873	-7,843
	DQN+T	-10,426	-11,091
	DDPG	-73,454	-6,765
	TD3	-13,942	-9,040
	SAC	-8,353	-7,932
offline	BCQ	-9,115	-9,913
	BEAR	-6,801	-5,117
	CQL	-9,423	-8,766

BC von -3,109. Letztere übertrifft jedoch selbst die Performance im Datensatz (-4,145), sodass es sich hierbei um eine zu hohe Messlatte handeln mag. Doch auch die Performance des Datensatzes erreichen die Modelle nicht, sodass hier weitere Experimente notwendig wären, die ggfs. die Vergrößerung des Datensatzes mit der Verbesserung der Policy kombinieren. Auch eine Verbesserung der Modelle über Hyperparameter-Anpassungen wäre denkbar.

6.3 Zusammenfassung der Erkenntnisse

Die wichtigsten Erkenntnisse dieses Kapitels sind in den folgenden Punkten zusammengefasst:

- Sollte man Zugang zu einem Datensatz haben, der bereits mit einer sehr guten Policy gesammelt wurde, kann Behavior Cloning eine gute Alternative zu Offline RL darstellen.
- Eine signifikante Verbesserung der Policy im Vergleich zu der bei der Datengenerierung verwendeten Policy ist grundsätzlich möglich, wie man an der Anwendung des *random*-Datensatzes erkennen kann.
- Die Diskretisierung des Aktionsraums ist eine valide Option (im hier betrachteten Anwendungsfall).
- Die Anwendung des DDPG Algorithmus auf einem mit DDPG erzeugtem Datensatz erzielt schlechte Ergebnisse und ist daher nicht empfehlenswert.
- Verfahren, die auf dem Ansatz der maximalen Entropie basieren (SAC) profitieren von einer höheren Varianz im Datensatz, sodass fertig-trainierte Datensätze hier schlechtere Ergebnisse erzeugen und ggfs. höhere Datenmengen benötigen.
- Der BEAR Algorithmus erzielt auf allen Datensätzen die besten Ergebnisse.
- Von der Auswahl der initalen Datensätze eignet sich vor allem der *ddpg_control*-Datensatz als Basis für offline Modelle, da hier eine konsistente Policy verfolgt wird. Im Gegensatz dazu kann die Verwendung des *ran-*

dom-Datensatzes, abhängig vom Verfahren, durch eine hohe Zustandsraum-Abdeckung und eine hohe Varianz bzgl. Zustands-Aktions-Paare, gute Ergebnisse produzieren. Eher schlecht geeignet als Trainingsdatensatz ist der *ddpg_train*-Datensatz.
- Eine Vergrößerung des Datensatzes kann bestimmten Modellen helfen, ihre Performance zu verbessern.
- Die Performance der Policy, welche zur Datenerzeugung verwendet wurde, beeinflusst die Performance des Offline-Modells.

7 Schlussbetrachtung

Diese Ausarbeitung dient dem Einstieg in das Thema des Offline Reinforcement Learnings, einem Zweig des RLs, welcher mit dem aktiven Lernparadigma des RL bricht und mithilfe eines statischen Datensatzes eine Policy erlernen soll, welche sich zur Interaktion mit der Umwelt eignet.
Im Zuge dessen wurden unterschiedliche Algorithmen sowohl aus dem Bereich des off-Policy Reinforcement Learnings, als auch aus dem Offline RL vorgestellt und schließlich auf fünf unterschiedlichen Datensätzen erprobt.
Die Ergebnisse zeigen, dass das Erlernen einer Policy unter diesen Bedingungen nicht nur möglich ist, sondern auch eine Verbesserung im Vergleich zum Datensatz erzielt werden kann. Trotzdem sind die Ergebnisse im Verhältnis zu bekannten Benchmarks des Anwendungsfalls weit entfernt von optimalem Verhalten, sodass weitere Experimente hier hilfreich und spannend sein können.

Literatur

1. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: Openai gym (2016)
2. Fujimoto, S., Conti, E., Ghavamzadeh, M., Pineau, J.: Benchmarking batch deep reinforcement learning algorithms (2019)
3. Fujimoto, S., van Hoof, H., Meger, D.: Addressing function approximation error in actor-critic methods (2018)
4. Fujimoto, S., Meger, D., Precup, D.: Off-policy deep reinforcement learning without exploration (2019)
5. Haarnoja, T., Zhou, A., Abbeel, P., Levine, S.: Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor (2018)
6. Kumar, A.: Data-driven deep reinforcement learning (12 2019), https://bair.berkeley.edu/blog/2019/12/05/bear/
7. Kumar, A., Fu, J., Tucker, G., Levine, S.: Stabilizing off-policy q-learning via bootstrapping error reduction (2019)
8. Kumar, A., Zhou, A., Tucker, G., Levine, S.: Conservative q-learning for offline reinforcement learning (2020)
9. Lee, S.R.: Mujoco reacher environment: Performances of rl agents. https://www.endtoend.ai/envs/gym/mujoco/reacher (2020), accessed: 2021-04-02
10. Levine, S.: Deep reinforcement learning: Deep rl with q-functions (November 2020), http://rail.eecs.berkeley.edu/deeprlcourse/static/slides/lec-8.pdf

11. Levine, S., Kumar, A., Tucker, G., Fu, J.: Offline reinforcement learning: Tutorial, review, and perspectives on open problems (2020)
12. Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D.: Continuous control with deep reinforcement learning (2019)
13. Mahmood, A.R., Korenkevych, D., Komer, B.J., Bergstra, J.: Setting up a reinforcement learning task with a real-world robot (2018)
14. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. nature **518**(7540), 529–533 (2015)
15. Sammut, C.: Behavioral Cloning, pp. 93–97. Springer US, Boston, MA (2010). https://doi.org/10.1007/978-0-387-30164-8_69, https://doi.org/10.1007/978-0-387-30164-8_69
16. Seno, T.: d3rlpy: An offline deep reinforcement library. https://github.com/takuseno/d3rlpy (2020)
17. Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., Riedmiller, M.: Deterministic policy gradient algorithms. In: Xing, E.P., Jebara, T. (eds.) Proceedings of the 31st International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 32, pp. 387–395. PMLR, Bejing, China (22–24 Jun 2014), http://proceedings.mlr.press/v32/silver14.html
18. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. The MIT Press, second edn. (2018), http://incompleteideas.net/book/the-book-2nd.html