Optimierter DDPG für die HalfCheetah-Umgebung mittels Hyperparameter-Tuning

Anna-Maria von Spreckelsen
Fakultät Technik und Informatik
Department Informatik
Hochschule für Angewandte Wissenschaften
Hamburg, DE

Kurzzusammenfassung

Deep Deterministic Policy Gradient (DDPG) ist eine Methodik zur Kontrolle kontinuierlicher Simulationsumgebungen des Reinforcement Learnings. Im Folgenden wird evaluiert inwiefern DDPG für die physikalische HalfCheetah-Umgebung mittels Hyperparameter-Tuning optimiert werden kann. Für die Evaluation werden drei Experimente mit den Hyperparametern Batch-Größe, Standardabweichung des Action-Noise und Lernraten von Actor- und Critic-Optimierer durchgeführt. Der höchste durchschnittliche Reward von über 5000 wurde mit den klassischen Lernraten (0.0001, 0.001) erreicht. Für weitere Optimierungen des DDPG könnte zukünftig auf die Methodik des Parameter-Noise zurückgegriffen werden, da diese vielversprechende Ergebnisse liefert.

Schlüsselworte: Deep Deterministic Policy Gradient, HalfCheetah, Reinforcement Learning, Acor-Critic, Zustandsraum, Aktionsraum, Policy, Reward

1. Einleitung

Reinforcement Learning ist neben dem „Überwachtem“ und „Unüberwachtem Lernen“ ein Teilgebiet des Machine Learnings. Dieses Teilgebiet verfolgt auf Basis einer Policy die Maximierung des Rewards eines Agenten, der mit seiner Umwelt, z.B. innerhalb der Simulationsumgebung, interagiert. Die Agenten verfügen über diskrete oder/und kontinuierliche Zustands- und Aktionsräume.

Da die Kontrolle kontinuierlicher Zustands- und Aktionsräume komplexer als die diskreter ist wurden hierfür sogenannte Policy Gradient-Algorithmen entwickelt. Einer dieser Algorithmen ist Deep Deterministic Policy Gradient (DDPG). Im Folgenden wird evaluiert inwieweit sich dieser Algorithmus für die HalfCheetah-v2-Umgebung mittels Hyperparameter-Tuning optimieren lässt.

2. Grundlagen

2.1 Actor-Critic-Methode in DDPG

Das Actor-Critic-Verfahren ist eine Temporal-Difference-Methode, siehe [1], und beschreibt ein allgemeines Konzept zum Lernen einer Policy π_θ (Actor) und einer geschätzten Bewertungsfunktion V^θ (Critic), [2]. Diese können unter anderem durch Neuronale Netze

abgebildet werden. Actor-Critic findet auch beim DDPG-Algorithmus Anwendung, siehe Abbildung 1. Zunächst nimmt der Actor μ den Zustand s entgegen und bestimmt die Aktion a. Anschließend berechnet der Critic Q auf Basis des Zustands s und der Aktion a den Q-Wert für diese Kombination.

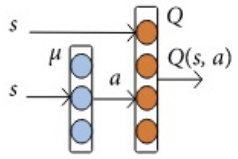


Abbildung 1: Zusammenspiel von Actor- und Critic-Netzwerk im DDPG-Algorithmus aus [3]. Der Actor ist μ und der Critic ist Q.

3. Verwandte Arbeiten

Im folgenden werden zwei verwandte Arbeiten erläutert, die sich mit der Kontrolle kontinuierlicher Simulationsumgebungen insbesondere physikalischen Umgebungen auseinander gesetzt haben.

In [4] wird der Algorithmus aus Abbildung 3 zur Kontrolle kontinuierlicher physikalischer Umgebungen evaluiert. Er ist ein modellfreier Actor-Critic Algorithmus auf Basis des Deterministic Policy Gradient (DPG)[5] Algorithmus. Der Algorithmus ist bei konstant gleichbleibenden Parametern dazu in der Lage über 20 verschiedene Aufgaben physikalischer Umgebung zu lösen.

[6] befasst sich ebenfalls mit der Kontrolle kontinuierlicher Umgebungen, genauer genommen der Bedeutung von Hyperparametern in Policy Gradienten, genereller Varianz in Algorithmen und der Reproduzierbarkeit anderer Veröffentlichungen. Für die Experimente werden die Algorithmen DDPG und Trust Region Policy Optimization (TRPO), einschließlich der beiden physikalischen Umgebungen „Hopper-v1“ und „HalfCheetah-v2“ aus [7] verwendet. Die Untersuchungen zeigen, dass die HalfCheetah-Umgebung anfälliger für Leistungsschwankungen durch das Hyperparameter-Tuning ist als der Hopper. Weiterhin wird die Reproduzierbarkeit von Veröffentlichungen durch externe Zufälligkeiten erschwert, sodass mehrere Veröffentlichungen über unterschiedliche Ergebnisse verfügen. Die durchgeführten Experimente grenzen diese Problematik ein und können als Basis für zukünftige Arbeiten dienen.

4. Methodiken

4.1 Umgebung

Die Umgebung der „HalfCheetah-v2“ ist Bestandteil von [7]. In dieser Bibliothek werden verschiedene physikalische Umgebungen für Reinforcement Learning-Aufgaben gekapselt. Die HalfCheetah verfügt sowohl über einen kontinuierlichen Zustands- als auch Aktionsraum. Tabelle 1 beinhaltet die Größen der beiden Räume. Der Zustandsraum ist ein eindimensionaler Vektor der Länge 17 und der Aktionsraum ein eindimensionaler Vektor der Länge 6.

Zustandsraum	Aktionsraum
17	6

Tabelle 1: Größe von Zustands- und Aktionsraum

Der Aktionsraum bezieht sich auf die physikalischen Gelenke der HalfCheetah. Abbildung 2 zeigt ein Modell der HalfCheetah mit den einzelnen Gelenken und ihrem Vorkommen im Aktionsraum.

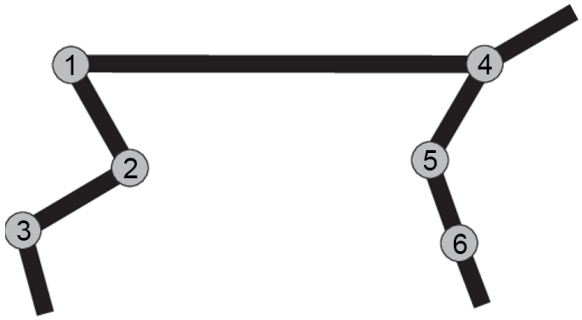


Abbildung 2: Gelenke der HalfCheetah. Sie werden auf den Aktionsraum abgebildet.

4.2 Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) ist ein modellfreier off-policy Algorithmus des Reinforcement Learnings. Er eignet sich zur Kontrolle von Umgebungen mit kontinuierlichen Zustands- und Aktionsräumen und somit auch für die HalfCheetah-Umgebung. [4] hat sich mit der Kontrolle kontinuierlicher Aktionsräume mittels Deep Reinforcement Learning auseinander gesetzt und den DDPG-Algorithmus aus Abbildung 3 evaluiert. DDPG basiert auf der Actor-Critic-Methode, siehe Unterabschnitt 2.1, und verfügt dementsprechend über die zwei Netzwerke Actor und Critic. Außerdem nutzt es einen Experience Replay um nicht nur aus den jüngsten Erfahrungen zu lernen, sondern aus den Stichproben aller bisher gesammelten Erfahrungen. Weiterhin nutzt es zwei Kopien dieser Netzwerke als Target-Netzwerke um den Trainingsprozess zu stabilisieren.

Actor-Netzwerk $\mu(s|\theta^\mu)$ und Critic-Netzwerk $Q(s, a|\theta^Q)$ werden zunächst mit den Gewichten θ^μ und θ^Q zufällig initialisiert. Die Target-Netzwerke μ' und Q' werden mit den Gewichten $\theta^{\mu'}$ und $\theta^{Q'}$ initialisiert, wobei $\theta^{\mu'} \leftarrow \theta^\mu$ und $\theta^{Q'} \leftarrow \theta^Q$. Außerdem nutzt DDPG wie der Deep Q Network (DQN) Algorithmus einen Replay-Buffer, der als ein Cache finiter Größe R dient.

Im nächsten Schritt folgt eine geschachtelte Schleife von 1 bis M Durchläufen. In jedem dieser Durchläufe wird ein zufälliger Prozess N für die Aktionsexploration initialisiert und der initiale Beobachtungszustand s_1 der Umgebung empfangen.

Danach folgt eine weitere Schleife mit 1 bis T Durchläufen. Hier wird zunächst die zu selektierende Aktion a_t auf Basis der aktuellen Policy-Funktion und einem explorierenden Action-Noise berechnet. In der Policy-Funktion wird auf Basis des Beobachtungszustand s_t und dem Actor-Netzwerk $\mu(s|\theta^\mu)$ eine Aktion berechnet, welche wiederum mit

dem explorierenden Action-Noise, Ornstein-Uhlenbeck [8], addiert wird und daraus die zu selektierende Aktion resultiert. Anschließend wird Aktion a_t ausgeführt und die daraus stammenden Parameter Belohnung r_t und neuer bzw. künftiger Beobachtungszustand s_{t+1} gespeichert. Danach wird die gesamte Transition (s_t, a_t, r_t, s_{t+1}) im Replay-Buffer hinterlegt und gesichert. Daraufhin wird stichprobenartig ein zufälliger Minibatch aus N Transitionen (s_i, a_i, r_i, s_{i+1}) aus dem Replay-Buffer R entnommen. Als nächstes wird ein Target y_i durch Addition der Belohnung r_i und dem diskontierten Q-Wert des nächsten Zustands mit $\gamma \cdot Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$ bestimmt, für dessen Berechnungen die beiden Target-Netzwerke in geschachtelter Form, beginnend mit dem Actor, gefolgt vom Critic, angewendet werden. Im nächsten Schritt wird der Critic-Loss L durch Minimierung nach Gleichung 1

$$L = \frac{1}{N} \cdot \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2 \tag{1}$$

upgedatet. Außerdem wird die Actor-Policy nach dem Policy-Gradient-Theorem in Gleichung 2 upgedatet.

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i} \tag{2}$$

Hierbei wird die Aktion für den Critic aus dem Actor generiert, mit $a = \mu(s_i)$.

Abschließend werden die Gewichte der Target-Netzwerke wie folgt upgedatet. Das Critic-Netzwerk Q' mit Gleichung 3

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1 - \tau) \cdot \theta^{Q'} \tag{3}$$

und das Actor-Netzwerk μ' mit Gleichung 4.

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1 - \tau) \cdot \theta^{\mu'} \tag{4}$$

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Abbildung 3: Pseudocode für Deep Deterministic Policy Gradient aus[4]

5. Experimente und Ergebnisse

5.1 Setup

Das Setup beinhaltet die Half-Cheetah-Umgebung aus Unterabschnitt 4.1 und eine Implementation des DDPG-Algorithmus aus Unterabschnitt 4.2. Im Folgenden werden zunächst die Netzwerk-Architekturen für Actor und Critic erläutert.

Actor- und Critic-Netzwerk werden nach den Implementationen aus [9] konfiguriert. Als erstes wird die Architektur des Actors aus Abbildung 4 beschrieben. Diese verfügt zunächst über eine Eingangsschicht mit dem Zustandsraum bzw. der Anzahl Zustände als Eingabegröße. Anschließend wird dessen Ausgang in eine Folge zweier Dense-Schichten mit jeweils 256 Einheiten und einer ReLU-Aktivierungsfunktion eingespeist und der daraus stammende Ausgang in die letzte Dense-Schicht mit 6 Einheiten, einer tanh-Aktivierungsfunktion und einem zufällig initialisierten Kernel im Intervall $[-0.003, 0.003]$ eingegeben und der Aktionsraum berechnet. Das Intervall des zufällig initialisierten Kernels dient präventiv dazu die Ausgabewerte 1 und -1 nicht direkt zu Beginn zu erhalten, weil diese die Gradienten auf 0 reduzieren würden, da die tanh-Aktivierungsfunktion verwendet wird.

```
# Initialize weights between -3e-3 and 3-e3
last_init = tf.random_uniform_initializer(minval=-0.003, maxval=0.003)

inputs = layers.Input(shape=(num_states,))
out = layers.Dense(256, activation="relu")(inputs)
out = layers.Dense(256, activation="relu")(out)
outputs = layers.Dense(num_actions, activation="tanh", kernel_initializer=last_init)(out)

# Our upper bound is 1.0 for HalfCheetah.
outputs = outputs * upper_bound
model = tf.keras.Model(inputs, outputs)
```

Abbildung 4: Actor-Netzwerkarchitektur

Die Architektur des Critic unterscheidet sich zum Actor dahingehend, dass er über zwei separate Eingangsschichten mit separaten Streams verfügt, deren Ausgänge anschließend konkateniert und in eine Folge zweier Dense-Schichten mit ReLU-Aktivierungsfunktion eingegeben werden. Die separaten Eingangsschichten verarbeiten zum einen den Zustandsraum und zum anderen den hierfür parallel berechneten Aktionsraum, siehe auch Unterabschnitt 2.1. Abschließend werden die Q-Werte bestimmt.

```
# State as input
state_input = layers.Input(shape=(num_states))
state_out = layers.Dense(16, activation="relu")(state_input)
state_out = layers.Dense(32, activation="relu")(state_out)

# Action as input
action_input = layers.Input(shape=(num_actions))
action_out = layers.Dense(32, activation="relu")(action_input)

# Both are passed through seperate layer before concatenating
concat = layers.Concatenate()([state_out, action_out])
out = layers.Dense(256, activation="relu")(concat)
out = layers.Dense(256, activation="relu")(out)
outputs = layers.Dense(1)(out)

# Outputs single value for give state-action
model = tf.keras.Model([state_input, action_input], outputs)
```

Abbildung 5: Critic-Netzwerkarchitektur

Weiterhin wird der Adam-Optimierer, genau wie in [4], für Actor und Critic genutzt. Diese und weitere Parameter werden in Tabelle 2 mit ihren Standard-Werten zusammengefasst.

Parameter	**Wert**
Actor-Lernrate	0,001
Critic-Lernrate	0,002
Batch-Größe	64
Trainingsepisoden	2000
Standardabweichung Action Noise	0,2
diskontierter Faktor γ	0,99
Soft-Update-Faktor τ für Target-Netzwerke	0,005
Replay-Buffer Größe	1.000.000

Tabelle 2: Parameter

5.2 Hyperparameter-Tuning

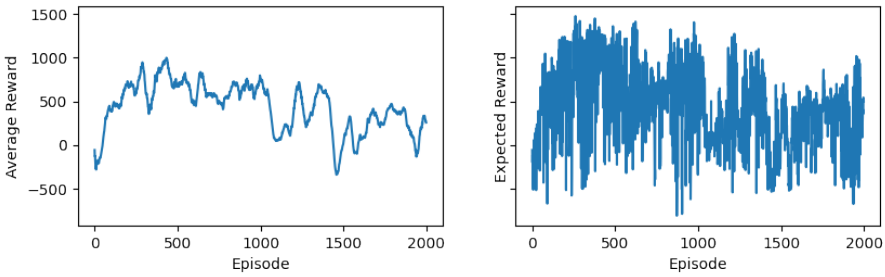
Auf Basis des Setups aus Unterabschnitt 5.1 werden die Parameter Actor- und Critic-Lernrate, Batch-Größe und Standardabweichung des Action Noise als neue Menge von Parametern, auch Hyperparameter genannt, zusammengefasst und in verschiedenen Experimenten untersucht. Die folgenden Experimente werden jeweils mit der Standardkonfiguration verglichen und diskutiert.

5.2.1 Batch-Größe

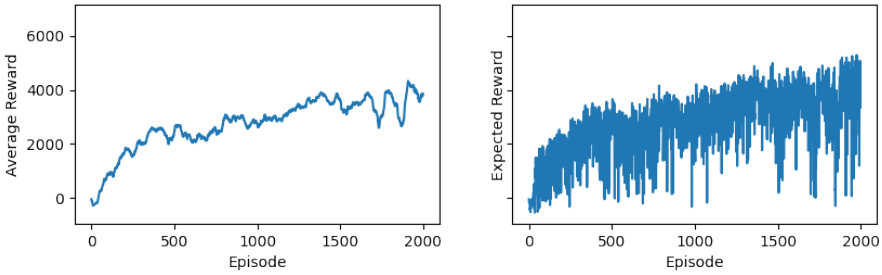
Im ersten Experiment wurden verschiedenen Batch-Größen (16, 32, 64) getestet und die Auswirkung dieser auf den durchschnittlichen und den gesamten Reward festgehalten, siehe Abbildung 6.

Zunächst wird der gesamte Reward über alle Versuchsgrößen hinweg betrachtet. Die Graphen verfügen über stark schwankende Amplituden, welches die Nachvollziehbarkeit dieser erschwert. Dies kommt unter anderem durch die 2000 Episoden mit jeweils 1000 Iterationen zustande. Um dem vorzubeugen werden die Graphen durch das arithmetische Mittel über die letzten 40 Reward-Werte einer Episode geglättet. Die Auswirkung der Glättung ist in den jeweils linken Graphen zu sehen. Die Graphen verfügen über eine im Verlauf gleichmäßigere Amplitude als zuvor, welches die Nachvollziehbarkeit verbessert.

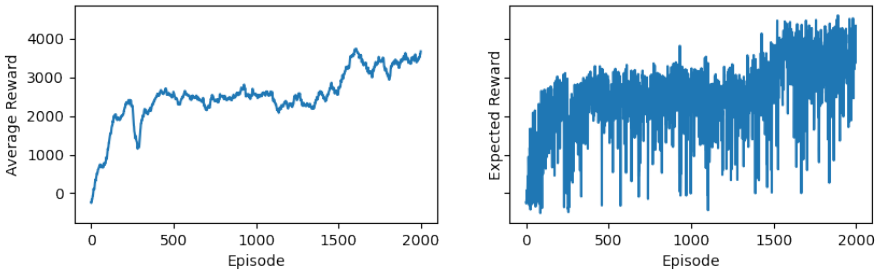
Nach der Glättung werden die durchschnittlichen Rewards der Versuchsgrößen miteinander verglichen. Der höchste durchschnittliche Reward wird bei mittlerer Batch-Größe mit über 4000 erreicht, gefolgt von der Standard-Batch-Größe mit einem Reward von fast 4000, gefolgt von der kleinen Batch-Größe mit einem Reward von knapp 1000. Im Gegensatz zur mittleren Batch-Größe verfügt die Standard Batch-Größe über eine steigende Amplitude und weniger starke Spikes zum Verlaufsende. Ähnliche Beobachtungen wurden in [6] gemach. Demnach sind die Batch-Größen 32 und 64 nahezu gleichwertig im Amplitudenverlauf.



(a) Kleine Batch-Größe 16



(b) Mittlere Batch-Größe 32



(c) Standard Batch-Größe 64

Abbildung 6: Batch-Größen im Vergleich. Durchschnittlicher Reward jeweils links und gesamter Reward jeweils rechts.

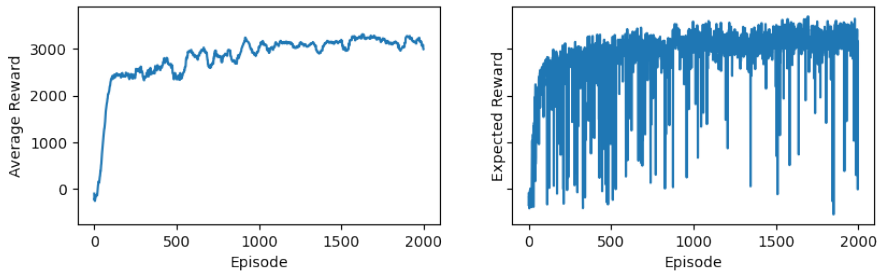
5.2.2 Standardabweichung Action Noise

Im zweiten Experiment wurden verschiedenen Standardabweichungen (0.05, 0.2, 0.6) im Action Noise getestet und die Auswirkung dieser auf den durchschnittlichen und den gesamten Reward festgehalten, siehe Abbildung 7.

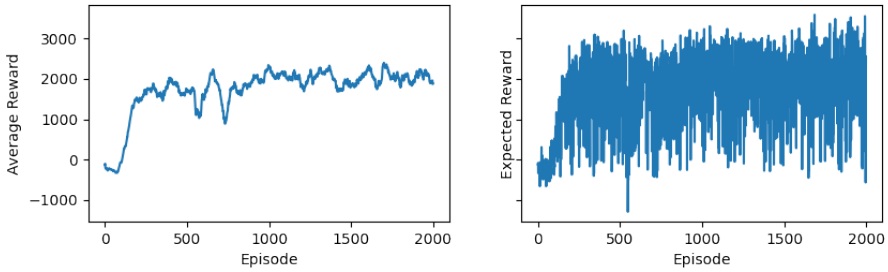
Ebenfalls wie in Unterunterabschnitt 5.2.2 wird auch der gesamte Reward über alle Versuchsgrößen hinweg betrachtet. Die Graphen verfügen genauso über stark schwankende Amplituden, welches die Nachvollziehbarkeit dieser erschwert. Außerdem wurde dieselbe Methode zur Glättung der Graphen angewandt und deren Auswirkung ist in den jeweils linken Graphen ersichtlich.

Der höchste durchschnittliche Reward wird bei niedrigster Standardabweichung mit über 3000 erreicht, gefolgt von der mittleren Standardabweichung mit einem Reward von über

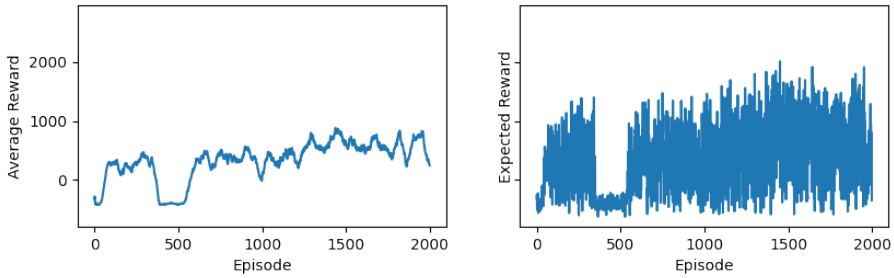
2000, gefolgt von der hohen Standardabweichung mit einem Reward von unter 1000. Im Gegensatz zur mittleren Standardabweichung verfügt die niedrige Standardabweichung über eine schneller ansteigende Amplitude zu Beginn und weniger starke Spikes im Verlauf. Zur weiteren Untersuchung werden die beiden gesamten Reward-Verläufe betrachtet. Hier besitzt die Mittlere Standardabweichung einen gleichmäßigeren Ausschlag der Amplitude als die niedrigere. Die Spikes bei niedriger Standardabweichung können unter anderem Indiz für Instabilität darstellen und sind dahingehen zu vermeiden.



(a) Niedrige Standardabweichung mit 0,05



(b) Standard bzw. Mittlere Standardabweichung mit 0,2



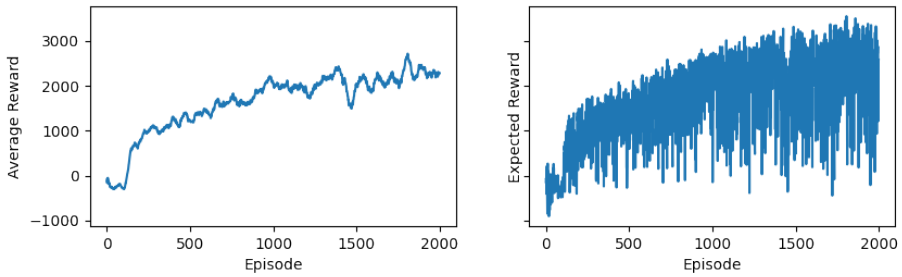
(c) Hohe Standardabweichung mit 0,6

Abbildung 7: Standardabweichungen vom Action Noise im Vergleich. Durchschnittlicher Reward jeweils links und gesamter Reward jeweils rechts.

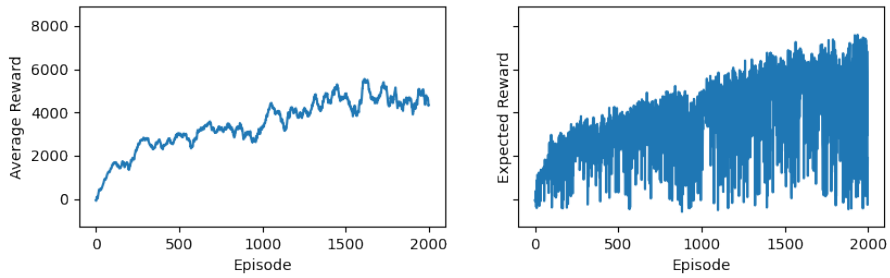
5.2.3 Lernrate

Im dritten Experiment wurden verschiedenen Lernraten bei Actor- und Critic-Optimierer ((0.00001, 0.0001), (0.0001,0.001), (0.001, 0.002)) getestet und die Auswirkung dieser auf den durchschnittlichen und den gesamten Reward festgehalten, siehe Abbildung 8. Die Lernraten stammen aus den Experimente von [6] und haben hier vielversprechende Ergebnisse geliefert.

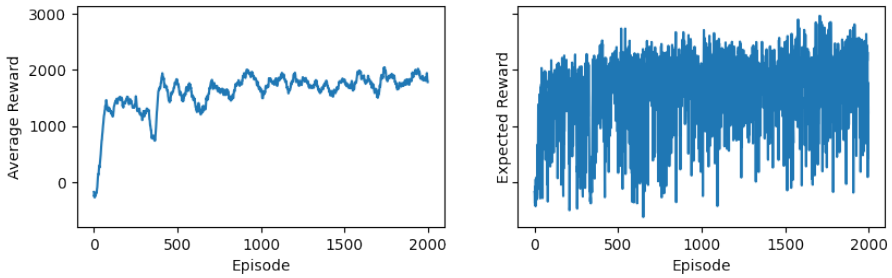
Auch in diesem Abschnitt wird der gesamte Reward über alle Versuchsgrößen hinweg betrachtet. Da auch hier die Graphen über stark schwankende Amplituden verfügen, wird dieselbe Methode zur Glättung der Graphen angewandt und deren Auswirkung in den jeweils linken Graphen zu ersichtlich.



(a) Niedrige Lernraten mit (0.00001, 0.0001)



(b) Klassische Lernraten mit (0.0001, 0.001)



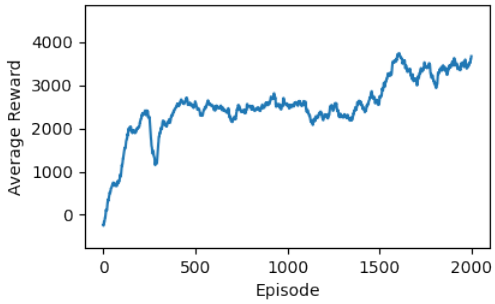
(c) Standard Lernraten mit (0.001, 0.002)

Abbildung 8: Lernraten von Actor- und Critic-Optimierer im Vergleich. Durchschnittlicher Reward jeweils links und gesamter Reward jeweils rechts.

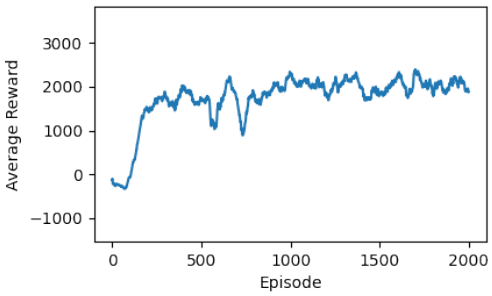
Der höchste durchschnittliche Reward wird durch die klassischen Lernraten mit über 5000 erreicht, gefolgt von den niedrigen Lernraten mit einem Reward von über 2000, gefolgt von den Standard Lernraten mit einem Reward von knapp über 2000. Im Gegensatz zu den Standard Lenraten verfügen klassische und niedrige Lernraten über einen gleichmäßigen Anstieg der Amplitude, während die Standard Lernraten schnell ansteigen und anschließend über starke Spikes im Verlauf verfügen. Selbiges zeigen die Verläufe der gesamten Rewards.

5.3 Zufallsinitialisierer

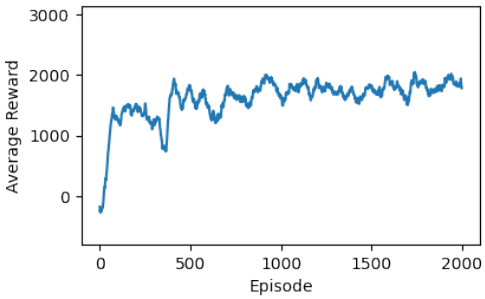
Ein weiterer Aspekt, abseits des Hyperparameter-Tunings, ist der Zufallsinitialisierer für die Gewichte der Actor-Critic-Netzwerke. Wie bereits in [6] benannt haben externe Zufälligkeiten Einfluss auf die Reproduzierbarkeit von Versuchsreihen. Selbiges Szenario wird im Folgenden durch einen Vergleich der Standardkonfigurationen über die vorherig durchgeführten Experimente gezeigt, siehe Abbildung 9. Die Amplituden unterscheiden sich nicht nur im Verlauf sondern auch der Stärke. So erreicht sie in Abbildung 9a nahezu die 4000, während sie bei Abbildung 9b und Abbildung 9c um die 2000 liegt. Weiterhin führt der Zufallsinitialisierer zu einem unterschiedlich schnellen Anstieg der Amplitude. So steigt sie in Abbildung 9c schneller an als in Abbildung 9b.



(a) Standardkonfiguration aus Unterunterabschnitt 5.2.1



(b) Standardkonfiguration aus Unterunterabschnitt 5.2.2



(c) Standardkonfiguration aus Unterunterabschnitt 5.2.3

Abbildung 9: Standardkonfigurationen aus den Experimenten von Unterabschnitt 5.2

6. Fazit

Die vorliegende Arbeit verfolgt das Ziel zu erforschen inwieweit sich der DDPG-Algorithmus für die HalfCheetah-Umgebung mittels Hyperparameter-Tuning optimieren lässt.

Es wurden drei Experimente zu den Hyperparametern Batch-Größe, Standardabweichung des Action Noise und den Lernraten von Actor- und Critic-Optimierer durchgeführt und analysiert. Der jeweils höchste durchschnittliche Reward konnte durch eine 32er Batch-Größe, eine 0,05er Standardabweichung im Action Noise und den klassischen Lernraten für Actor- und Critic-Optimierer mit (0.0001, 0.001) erreicht werden. Über alle Experimente hinweg wird mit den klassischen Lernraten der höchste durchschnittliche Reward von über 5000 erreicht. Außerdem konnte beobachtet werden, dass der Zufallsinitalisierer der Gewichte die Reproduzierbarkeit beeinflusst und eine allgemein gültige Verifizierung der Experimente erschwert. Um die vorherigen Ergebnisse der als am Besten eingestuften Hyperparameter zu verifizieren, wäre es notwendig, jedes der Experimente mehrmals durchzuführen.

Resultierend ist festzustellen, dass die Optimierung des DDPG für die HalfCheetah-Umgebung mittels der Standardkonfiguration und dem Lernraten-Tuning realisiert werden kann. In Abschnitt 7 wird ein Ausblick darüber gegeben welche Methodik dies zukünftig optimieren kann.

7. Ausblick

Angeknüpft an den vorherigen Abschnitt wird im Folgenden eine Methodik zur besseren Exploration diskutiert [10]. Sie nutzt statt dem Action-Noise das Parameter-Noise. Das Konzept besteht darin den Parametern der Neuronalen Netzwerk-Policy ein adaptives Noise hinzuzufügen, statt wie üblich dem Aktionsraum. Dadurch wird den Parametern des Agenten eine Zufälligkeit hinzugefügt, welche die Art der Entscheidungen verändert, sodass diese immer vollständig davon abhängen, was der Agent gerader wahrnimmt. [11] hat gezeigt, dass die Anwendung des Parameter-Noise auf die HalfCheetah-Umgebung in Verbindung mit dem DDPG-Algorihtmus zu einer verbesserten Effizienz des Algorithmus führt und somit auch die Instabilität verringert wird.

Literaturverzeichnis

[1] G. Tesauro *et al.*, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[2] A. Folkers, *Steuerung eines autonomen Fahrzeugs durch Deep Reinforcement Learning*. Springer, 2019.

[3] J. Wu and H. Li, "Deep ensemble reinforcement learning with multiple deep deterministic policy gradient algorithm," *Mathematical Problems in Engineering*, vol. 2020, 2020.

[4] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2019.

[5] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[6] R. Islam, P. Henderson, M. Gomrokchi, and D. Precup, "Reproducibility of benchmarked deep reinforcement learning tasks for continuous control," 08 2017.

[7] OpenAI Gym, "Mujoco." Online zu finden unter `https://github.com/openai/gym/tree/master/gym/envs/mujoco`; abgerufen am 01. März 2022.

[8] G. E. Uhlenbeck and L. S. Ornstein, "On the theory of the brownian motion," *Phys. Rev.*, vol. 36, pp. 823–841, Sep 1930.

[9] amifunny, "Deep deterministic policy gradient (ddpg)." Online zu finden unter `https://keras.io/examples/rl/ddpg_pendulum/`; abgerufen am 01. Januar 2022.

[10] M. Plappert, R. Houthooft, P. Dhariwal, S. Sidor, R. Y. Chen, X. Chen, T. Asfour, P. Abbeel, and M. Andrychowicz, "Parameter space noise for exploration," *CoRR*, vol. abs/1706.01905, 2017.

[11] M. Plappert, R. Houthooft, P. Dhariwal, S. Sidor, P. Abbeel, M. Andrychowicz, R. Chen, X. Chen, and T. Asfour, "Better exploration with parameter noise." Online zu finden unter `https://openai.com/blog/better-exploration-with-parameter-noise/`; abgerufen am 01. Februar 2022.