Ludwig-Maximilians-Universität München
Institut für Informatik
Lehrstuhl für Mobile und Verteilte Systeme
Prof. Dr. Claudia Linnhoff-Popien

mobile and distributed systems group

Praktikum Autonome Systeme
Wintersemester 2020/21
Übungsblatt 4 – Value-based Deep Reinforcement Learning

Ziel der heutigen Praxisveranstaltung ist das Tuning und die Erweiterung von DQN zur Lösung von einfachen OpenAI Gym Umgebungen. Für dieses Übungsblatt wird zusätzlich das aktuelle `pytorch`-Package benötigt.

Aufgabe 1: DQN Hyperparameter Tuning

Laden Sie für dieses Übungsblatt das ZIP-Archiv `autonome-systeme-uebung4.zip` runter. In diesem Archiv finden Sie zusätzlich die Datei `dqn.py` und eine für OpenAI Gym angepasste `main.py`.

In der Datei `dqn.py` finden Sie eine vollständige Implementierung des DQN Algorithmus in der Klasse `DQNLearner`. Machen Sie sich mit der Klasse vertraut: an welchen Stellen finden Sie die Mechanismen *Experience Replay* und *Target Network Update*?

Durch das Deep Learning kommen eine Menge neuer Hyperparameter hinzu (siehe `main.py`). Finden Sie ein geeignetes Hyperparameter-Setting, mit denen der `DQNLearner` die OpenAI Gym Domänen `CartPole-v1`, `Acrobot-v1` und `MountainCar-v0` löst (**Vorsicht:** Diese Aufgabe erfordert viel Zeit und Resourcen. Nutzen Sie dazu ggf. die Slurm-Engine im CIP-Pool).

Zusatzaufgabe: Implementieren Sie ein Verfahren (z.B. Random Search, Evolutionary Optimization), das automatisch nach einem geeigneten Hyperparameter-Setting sucht.

Aufgabe 2: DQN Extensions

Testen Sie die folgenden Erweiterungen gegen die Original-Version des DQN (s. Aufgabe 1) in den OpenAI Gym Domänen `CartPole-v1`, `Acrobot-v1` und `MountainCar-v0`.

1. Double DQN

Ändern Sie die ursprüngliche TD-Target Berechnung von (siehe Zeile 109/110 in `dqn.py`):

$$Q_{target} = r_t + \gamma \max_{a_{t+1} \in \mathcal{A}} \hat{Q}_{\theta^-}(s_{t+1}, a_{t+1})$$

zu:

$$Q_{target'} = r_t + \gamma \hat{Q}_{\theta^-}(s_{t+1}, \overline{a_{t+1}})$$

wobei $\overline{a_{t+1}} = argmax_{a_{t+1} \in \mathcal{A}} \hat{Q}_{\theta}(s_{t+1}, a_{t+1})$, $\hat{Q}_{\theta}$ das trainierte Q-Network `self.policy_net` und $\hat{Q}_{\theta^-}$ das Target Network `self.target_net` repräsentieren.

2. Prioritized Experience Replay

Speichern Sie zusätzlich zu dem Experience Tuple $e_t = \langle s_t, a_t, r_t, s_{t+1}, done \rangle$ den absoluten TD-Error $\delta_t = |Q_{target'} - \hat{Q}_\theta(s_t, a_t)|$ (Verwenden Sie als $Q_{target'}$ die Double DQN Variante), sodass Sie in Zeile 108 in `dqn.py` das erweiterte Tupel $e'_t = \langle s_t, a_t, r_t, s_{t+1}, done, \delta_t \rangle$ in `self.memory` ablegen (Vergessen Sie dabei nicht Zeile 115 anzupassen!).

Passen Sie die Methode `sample_batch` in der Klasse `ReplayMemory` an, welche Experience Tuples $e'_t = \langle s_t, a_t, r_t, s_{t+1}, done, \delta_t \rangle$ mit der Wahrscheinlichkeit $P(e_t) = \frac{\delta_t}{\sum_j \delta_j}$ sampled.

Aktualisieren nach dem Aufruf von `self.optimizer.step()` den absoluten TD-Error δ_t aller Tuple e'_t aus dem Minibatch.

3. Soft Target Network Update

Passen Sie die Methode `update_target_network` an, sodass die Gewichte θ^- von `self.target_network` folgendermaßen aktualisiert werden:

$$\theta^- = (1 - \tau)\theta^- + \tau\theta$$

Wobei θ die aktuellen Gewichte von `self.policy_net` sind und τ das Target-Update gewichtet (experimentieren Sie zunächst mit $\tau \in \{0.1, 0.001, 0.0001\}$).

Stellen Sie zudem sicher, dass `update_target_network` nun nach jedem Trainingsschritt aktualisiert wird (und nicht nur alle `self.target_update_interval`-mal).

Zusatzaufgabe: Implementieren Sie einen "Mini Rainbow" DQN, der alle drei Erweiterungen nutzt. Vergleichen Sie die Performance des "Mini Rainbow" DQNs mit der Original-Version. Achten Sie darauf, dass beide Ansätze dasselbe Hyperparameter-Setting nutzen!