Reinforcement Learning

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2026-06 Dozent: Karsten Flügge

Deep Reinforcement Learning

Es lassen sich alle Hilfskonzepte und Hilfsfunktionen welche wir bis jetzt kennen gelernt haben mit **neuronalen Netz trainieren**:

- **Value** Funktionen,
- **Quality** Funktionen oder
- **Policy** direkt

Themen des Tages

Tag 8

Actor-Critic-Methoden

- **AC Grundalgorithmus** (TD)
  - **REINFORCE with baseline** (MC)
  - V/Q Advantage
- Deep Deterministic Policy Gradients (DDPG)

Deep Reinforcement Learning

Es lassen sich alle Hilfskonzepte und Hilfsfunktionen welche wir bis jetzt kennen gelernt haben mit **neuronalen Netz trainieren**:

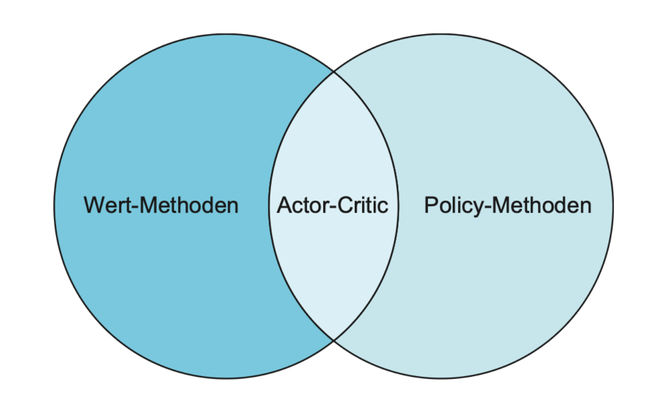
- **Value** Funktionen,
- **Quality** Funktionen oder
- **Policy** direkt!
- => kombinieren!

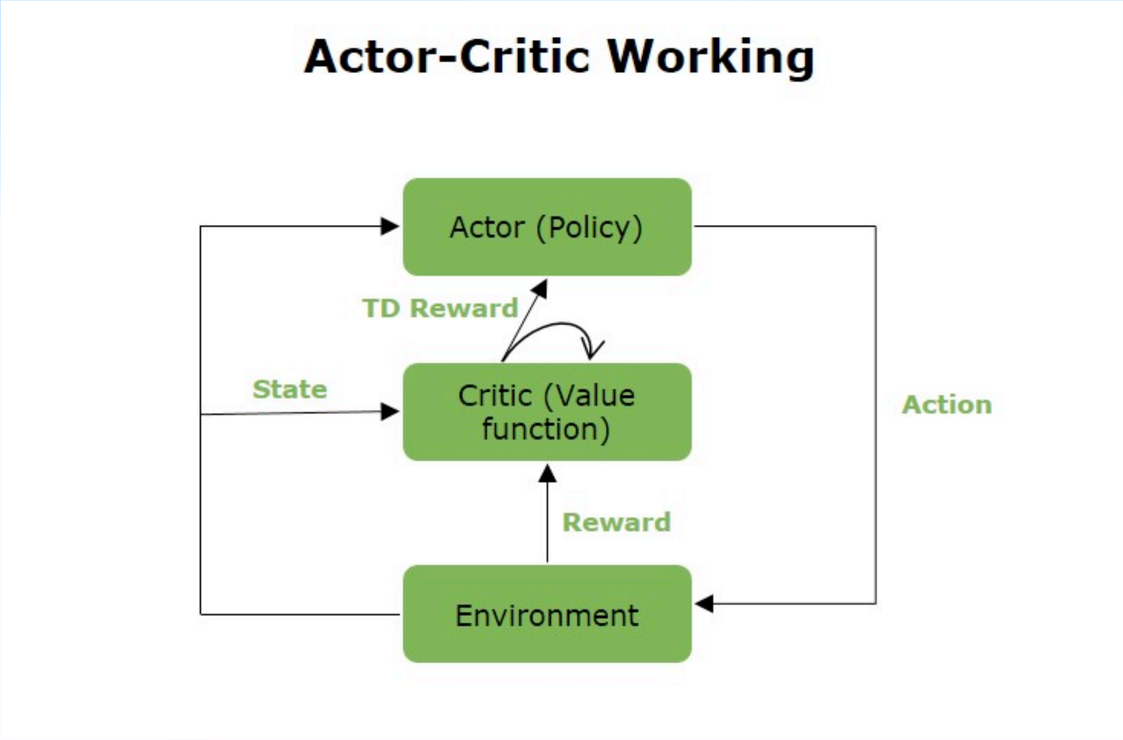
Actor-Critic-Methoden

Eine **Klasse von Methoden** mit folgenden Gemeinsamkeiten

Eigenes Netz für **Policy(Actor)** und eines für **Value(Critic).**

Critic ist on-policy und bewertet den Actor, anhand von **TD**-Fehlern:





Actor-Critic-Methoden

Nach jeder Aktion bewertet der Kritiker den neuen Zustand, um zu schätzen, ob sich die Dinge **besser oder schlechter** entwickelt haben als erwartet:

$\underline{V}_t$ = true value = $\mathbf{r} + \gamma V_t(S_{t+1})$ **Wahrer Wert** ≈ geschätzter Gewinn, berechnet aus **rewards**

V_t = estimated value = Schätzung / Voraussage

$\delta_t = \underline{V}_t(S_t) - V_t(S_t)$ "Temporal Difference of Target V to real V"

ausformuliert:
$\delta_t = R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t)$ "**TD error** of estimate"

Actor-Critic-Methoden

Als Kritiker kann man statt V Netz auch **Q** Netz nehmen, dass nennt sich dann **Advantage**.

Nach jeder Aktion bewertet der Kritiker den neuen Zustand, um zu schätzen, ob sich die Dinge **besser oder schlechter** entwickelt haben als erwartet:

$V=Q_t$ = true value = $\mathbf{r} + \gamma \max Q_{tt}(S_{t+1})$ **Wahrer Wert** ≈ geschätzter Gewinn

V_t = estimated value = Schätzung / Voraussage

$\delta_t = Q_t(S_t) - Q_t(a|S_t)$ **"Temporal Difference of Target V to real V"**

ausformuliert:

$\delta_t = R_{t+1} + \gamma \max Q_t(a \mid S_{t+1}) - Q_t(S_t)$ **"TD error** of estimate"

besser A statt δ

Actor-Critic-Methoden

Nach jeder Aktions bewertet der Kritiker den neuen Zustand, um zu schätzen, ob sich die Dinge **besser oder schlechter** entwickelt haben als erwartet:

$\underline{V}_t$ = true value = $\sum$**Gains** = **r** + $\gamma V_t(S_{t+1})$ **Wahrer Wert (berechnet)**
V = estimated value = Schätzung / Voraussage vom NN

$$\delta_t = \underline{V}_t(S_t) - V_t(S_t) \quad \text{"Temporal Difference (delta) of } \underline{\text{Target}} \ \underline{V} \text{ to real V"}$$

$\delta_t > 0$: Die Belohnung war **besser als erwartet** → verstärke die *Tendenz*, Aktionen **a_t** zu wählen die zu s führen
$\delta_t < 0$: Die Belohnung war **schlechter als erwartet** → schwäche die *Tendenz*, Aktion a_t zu wählen

TD-Fehler als Lernsignal für beide Komponenten:

- Der **Critic** nutzt **δ_t** zur Verbesserung der Schätzung **V(s)**, direct als (MSE) **loss**!
- Der **Actor** passt die Policy **π(a | s)** so an, dass Aktionen mit positivem **δ_t** wahrscheinlicher werden:

Der **policy loss** verwendet **δ_t** wie vorher den Gewinn G_t:

$$\pi\text{-loss} = -\sum \log \pi(a_t | s_t; \theta) * \delta_t \qquad \text{“surrogate } \Phi\text{”}$$

Actor-Critic-Methoden

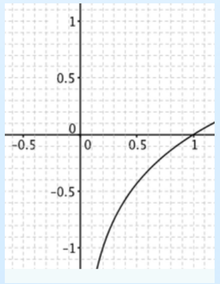
$\delta_t = \underline{V_t}(S_t) - V_t(S_t)$ "Temporal Difference (delta) of Target V to real V"

$\delta_t > 0$: Die Belohnung war **besser als erwartet** → verstärke die *Tendenz*, Aktionen a_t zu wählen die zu s führen
$\delta_t < 0$: Die Belohnung war **schlechter als erwartet** → schwäche die *Tendenz*, Aktion a_t zu wählen

Der **policy loss** verwendet δ_t wie vorher den Gewinn G_t:

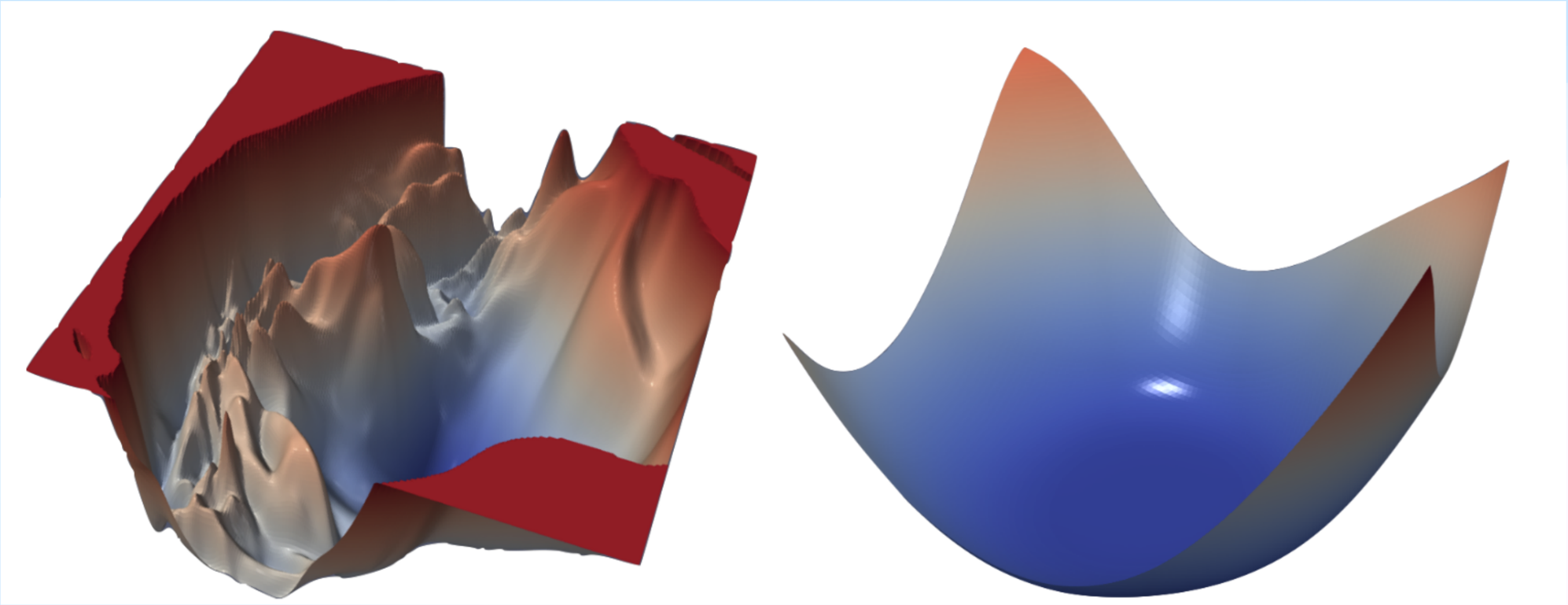
$$\pi\text{-loss} = -\sum \log \pi(a_t|s_t; \theta) * \delta_t$$

$\delta_t = 0$ => π-loss immer 0 => nicht nachbessern
$\pi(a) \approx 1$ hoch δ_t hoch => loss klein? ≈ 0 alles gut
$\pi(a) \approx 1$ hoch δ_t negativ => 0 loss klein!?!
$\pi(a) \approx 0$ niedrig δ_t hoch => - -hoch * hoch => hoher Fehler nachbessern!
$\pi(a) \approx 0$ niedrig δ_t negativ => - -hoch * -hoch => niedriger Fehler < 0 nicht nachbessern



Negative Losses

Im Reinforcement-Learning können die Verluste nicht nur **negativ** sein. Sie können sich sogar so ändern, dass sie im Laufe des Trainings **größer** werden statt kleiner. Das ist aber kein Problem, solange es ein Tal im Gebirge gibt, den der Gradient abwandern kann.
Schwankende, negative, wachsende Verluste sind normal!



Actor-Critic-Methoden

Nach jeder Aktions bewertet der Kritiker den neuen Zustand, um zu schätzen, ob sich die Dinge **besser oder schlechter** entwickelt haben als erwartet:

$\underline{V}_t$ = true value = $\sum$**Gains** = $r + \gamma V_t(S_{t+1})$ **Wahrer Wert**
V = estimated value = Schätzung / Voraussage vom NN

$\delta_t = \underline{V}_t(S_t) - V_t(S_t)$ **"Temporal Difference of Target V to real V"**
ausformuliert:
$\delta_t = R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t)$ **"TD error** of value estimate"

R_{t+1} reward kommt von der Umgebung
$V_t(S_t)$ geschätzter Wert des alten Zustand
$V_t(S_{t+1})$ geschätzter Wert des neuen Zustandes
γ discount

$\delta_t > 0$: Die Belohnung war **besser als erwartet** → verstärke die Tendenz, Aktion a_t zu wählen
$\delta_t < 0$: Die Belohnung war **schlechter als erwartet** → schwäche die Tendenz, Aktion a_t zu wählen

TD-Fehler als Lernsignal für beide Komponenten:

- Der **Critic** nutzt δ_t zur Verbesserung der Schätzung **V(s)**, direct als (MSE) loss!
- Der **Actor** passt die Policy **π(a | s)** so an, dass Aktionen mit positivem δ_t wahrscheinlicher werden:

Der **policy loss** verwendet δ_t wie vorher den Gewinn G_t:

π**-loss** = $-\sum$ **log** $\pi(a_t|s_t; \theta) * \delta_t$ **"surrogate Φ"**

Actor-Critic-Methoden

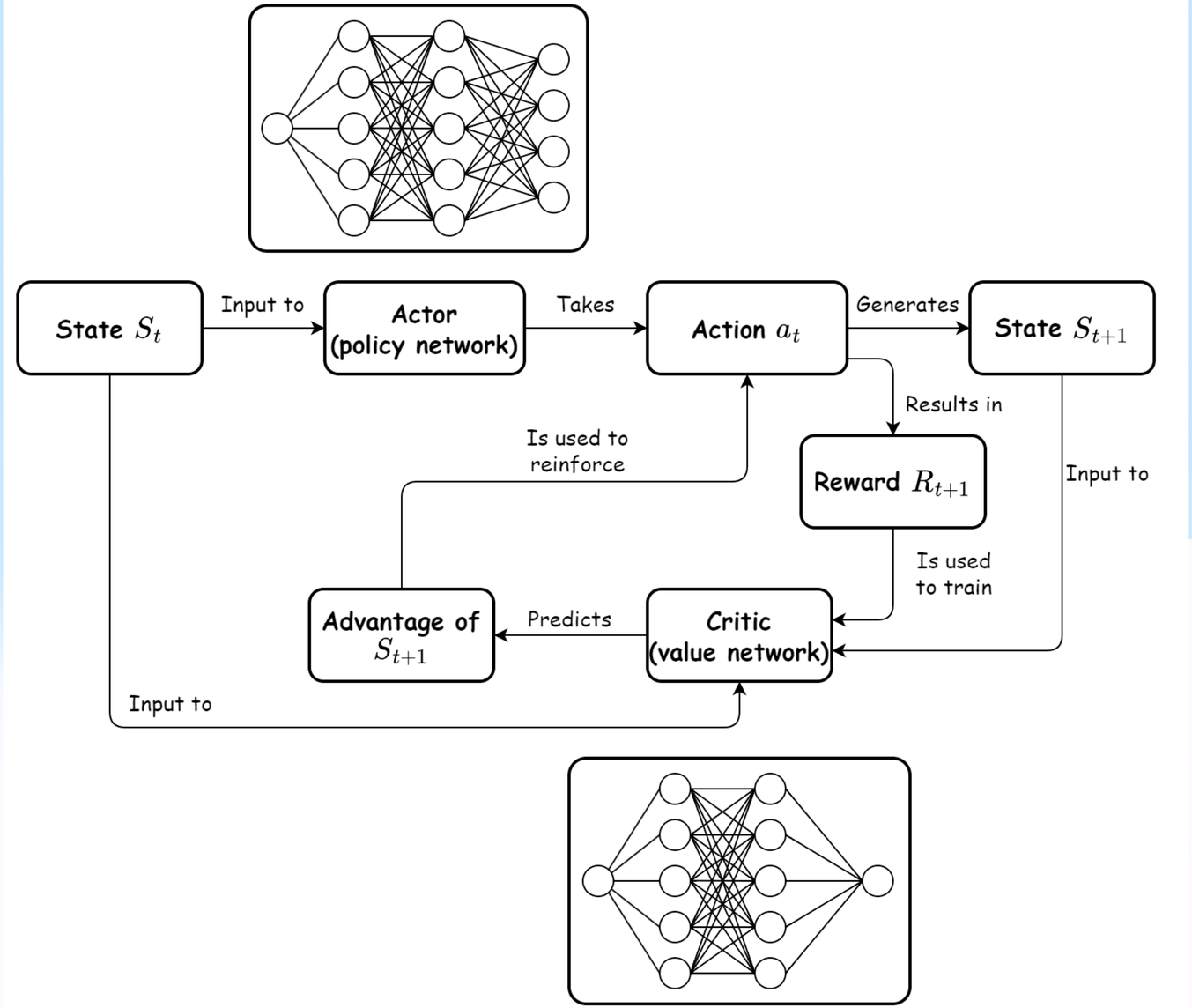
Actor/Action Network wie bisher **Policy Network**
Critic Network ähnlich wie **V/Q Netzwerk** nur **mit Anzahl der States** (ohne Aktionen), 1 Wert

Critic:
input = #states
schichten = beliebig ML
output = Wert (Bewertung des States *unter policy π*) ≈ reward + Zukunfts-Gewinn
(im Gegensatz zum Q-Netz, wo wir alle Qualities gleichzeitig berechneten)

Actor (policy):
input = #states
schichten = beliebig ML
output = Wahrscheinlichkeiten der Aktionen

Actor-Critic-Methoden

Sutton



Actor-Critic-Methoden

Algorithmus

```
a_t = actor(s_t)  # wählt Aktion
r_t, s_t+1 = environment(s_t, a_t)
# z.B. Schritt direkt beim Spielen wie Sarsa :

def ActorCriticTrainingStep(s_t, a_t, r_t, s_t+1):    # Update value approximation for v(s_t)!
    V_s = critic(s_t)  # estimate
    V_s+1 = critic(s_t+1).detach()

    V_target = r_t + gamma * V_s+1  # mini Bellman Gleichung  Gewinnschätzung
    delta = V_target - V_s   # TD-Fehler wie gut war Schätzung

    # Critic Loss: MSE für V(s) = L2
    critic_loss = delta^2  # mean square error: V_estimate und V_target sollen möglichst gleich sein

    # Actor Loss: policy gradient mit TD-Fehler
    prob = policy(s_t)[a_t]
    log_pi =  log π * δ_t
    actor_losses += -log_pi * delta.detach()  # Spezialfall von loss = -∑ log π * δ_t

    # Update Networks (Adam)
    criticθ ← criticθ - α · ∇ critic_loss
    actorθ ← actorθ - α · ∇ actor_losses    # später, summiert über alle
```

Actor-Critic-Methoden

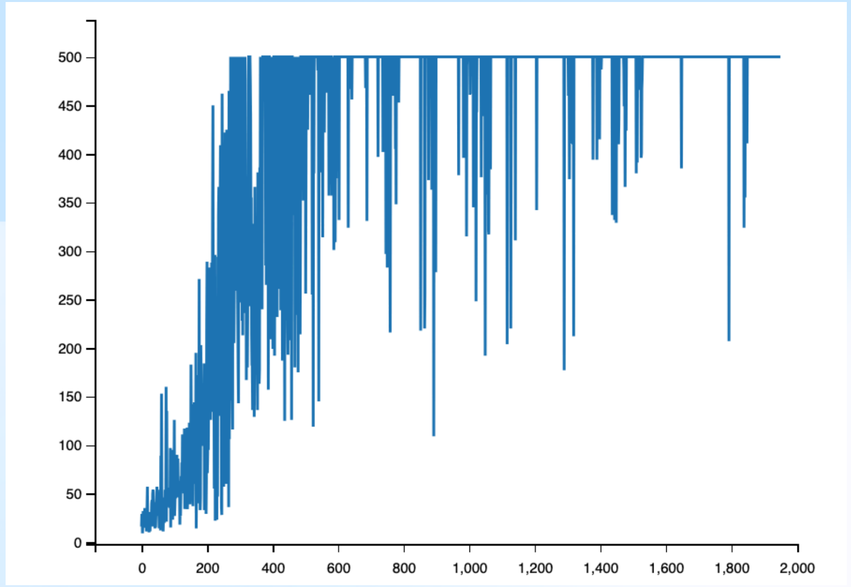
TD scrittweise:

$V_{target} = r_t + gamma * V_{s+1}$ # mini Bellman Gleichung, Schätzung via VORHER V (bootstrap)

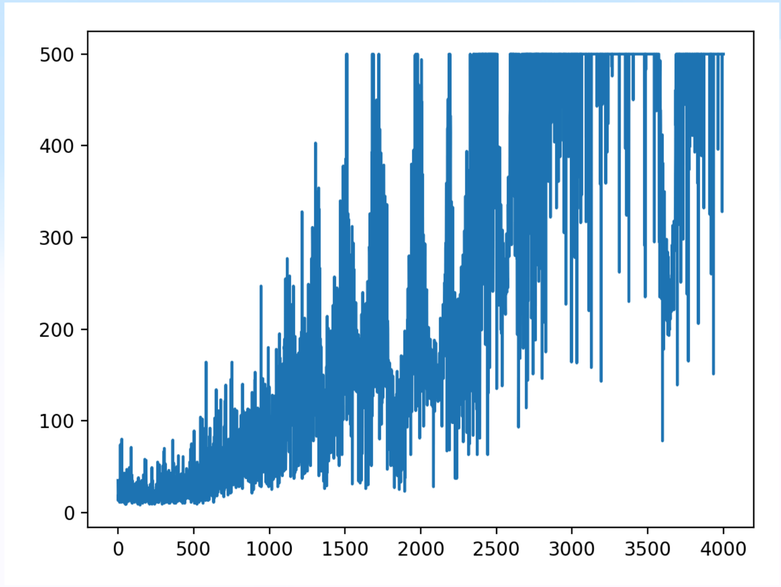
MC episodic:

$V_{target} = r_t + gamma * G$ # Gewinnschätzung via **Messung**

MC:



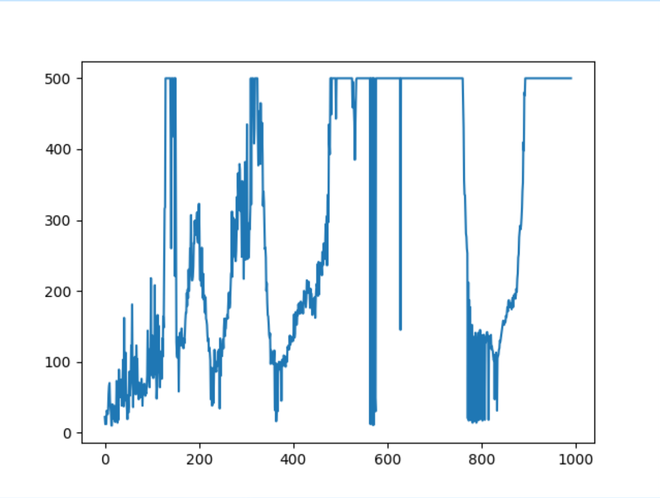
TD:



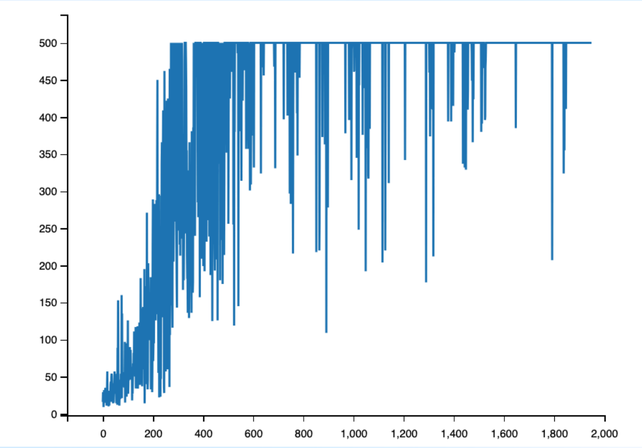
Actor-Critic-Methoden

Helfen gegen **katastrophales Vergessen**

Vorher



Nachher:



Actor-Critic-Methoden

Actor-Critic-Methoden kombinieren Policy Gradient mit einem **Value** Netzwerk

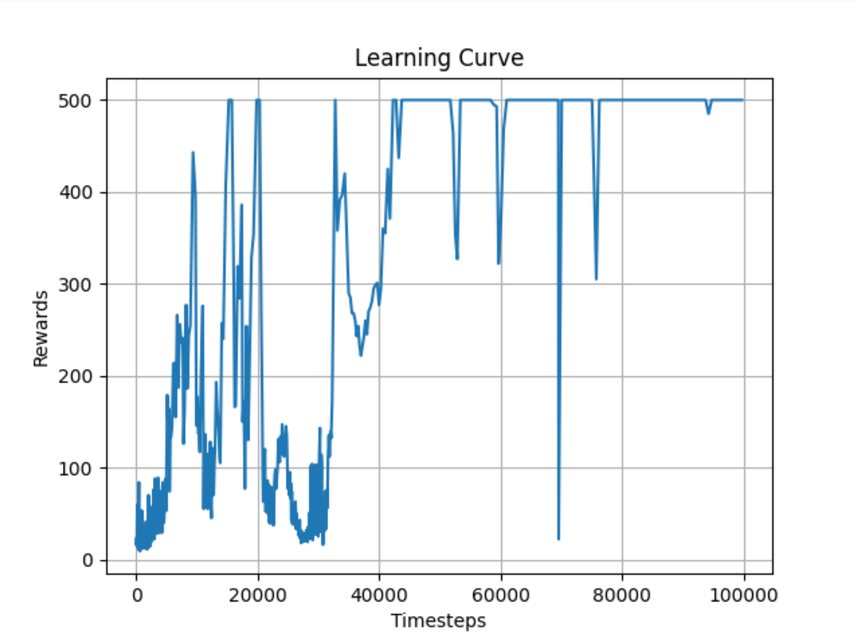
State-Value vs Action-Value "Quality" :

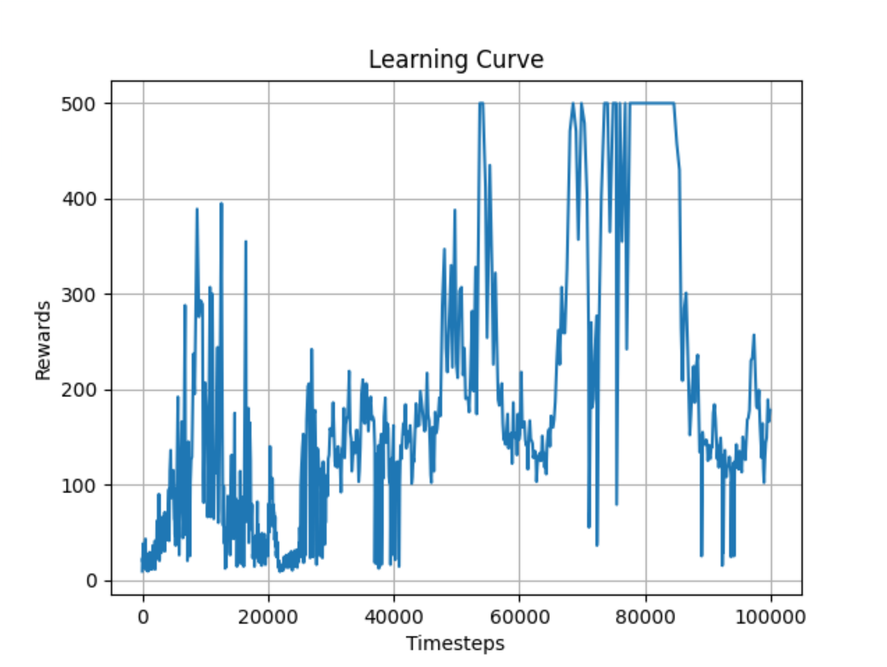
- **A2C / PPO**: Kritiker = **Value**-Netzwerk V(s)
- **DDPG, TD3, SAC**: Kritiker = **Q**-Netzwerk Q(s)

Actor-Critic-Methoden

A2C in baselines3 fertiger Algorithmus

```
from stable_baselines3 import A2C # WORKS GREAT! sometimes!?

env = Monitor(gym.make("CartPole-v1", render_mode=render_mode), log_dir)

algo = A2C("MlpPolicy", env, verbose=1)
algo.learn(total_timesteps=100000) # 1 step = one action! (not episode!)
```





Regularisierung für bessere Konvergenz

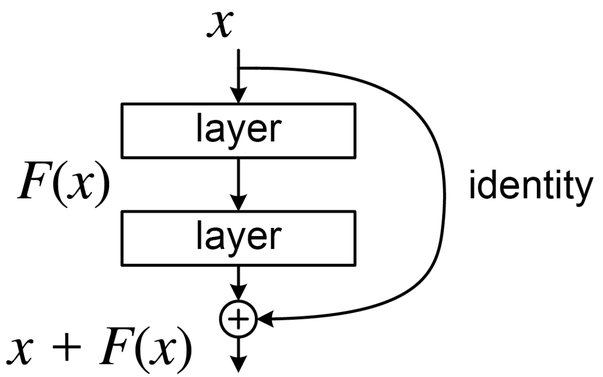
- Wie können wir unser **Netzwerk besser trainieren** lassen
- **(**schnellere/**zuverlässigere Konvergenz)?**

- **<u>alle</u> deep learning Regularisierungs Techniken**
  - (dropout, batch/layer-norm, **skip/residual**s, noise, initialization)
- **batch / replay**-buffer!
- **model**
- **Target** Network
- polyak **moving average**

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi,$$

- **clipping/clamp** (große/kleine Werte abschneiden, immer wenn ein Wert den 'guten' Bereich verlässt) besser:
- **bestrafen / belohnen über loss error (z.B. L1/L2)**
- **custom reward**
- **Actor / Critic**
- **"distributions" beobachten** (tensorboard) **=>**
- **Normierung** besser reversibel! (batch norm auf gain = mean / std)
- data curation

- **Double**-Deep-Q-Learning? **Double A/C**



Techniken / Features / Aspekte

Zoo taxonomy von Algorithmen alle in MDP Umgebung?

Aspekte

on/off policy
value / quality / policy -> actor-critic / advantage based (mixed)
batch / replay-buffer
episodisch / single step (Monte Carlo / TD)
mathematisch / tabular / deep
with(out) target network
noise/greed (exploration)

(non)supervised (real/simulation teacher data)
single/multi agent
model based / model free (interne simulation)

single / complete sampling "sweep?"

Kompatibel/Mischformen? Ausprobieren **Stable-Baselines3**

Techniken / Features / Aspekte

Die meisten Algorithmen sind implementiert in standard Bibliotheken wie
Stable-Baselines3
RLlib
Ray?

pole-**actor-critic-mc.py** endlich ein Algo der gut funktioniert (nicht nur für pole;)

Algo zoo

- **SARSA** – On-policy TD control.
- **DQN** – Deep **Q**-Network. DQL
- **PG** – Policy Gradient
- **REINFORCE** – Monte Carlo policy gradient.
- **DDPG** – Deep Deterministic Policy Gradient.
- **TD3** – Twin Delayed DDPG!
- **A2C** – Advantage Actor-Critic (synchronous version).
- **A3C** – Asynchronous Advantage Actor-Critic.
- **TRPO** – Trust Region Policy Optimization.
- ***PPO*** – better and simpler than TRPO <<
- FLOW / diffusion & **Transformer**
- **GAIL** – Generative Adversarial Imitation Learning.

...

Algo zoo 2

Modern Policy Gradient Variants

- **PPO** – Proximal Policy Optimization.
- **DPO** – Direct Preference Optimization.
- **GRPO** – **Group Relative Policy Optimization.**
- **DAPO** – Decoupled Clip and Dynamic Sampling Policy Optimization.
- **AMPO** – Active Multi-Preference Optimization.
- **SPPO** – Self-Play Preference Optimization.
- **RSPO** – Regularized Self-Play Policy Optimization.

—

Human Feedback / Preference-Based RL

- **RLHF** – Reinforcement Learning from Human Feedback.
- **RLAIF** – RL from AI Feedback.
- **P3O** – Pairwise Proximal Policy Optimization (variant of PPO using pairwise preferences).
- **T-REX** – Trajectory-ranked Reward Extrapolation.
- **IQ-Learn** – Imitation with Q-functions and offline reward inference.

—

Self-Play / Multi-Agent Learning

- **SPIN** – Self-Play Fine-Tuning.
- **PSRO** – Policy Space Response Oracles.
- **NFSP** – Neural Fictitious Self-Play.
- **MAPPO** – Multi-Agent PPO.
- **MADDPG** – Multi-Agent DDPG.
- **ROMMEO** – Regularized Opponent Modeling.

Algo zoo 3

Exploration & Intrinsic Motivation

- **RND** – Random Network Distillation.
- **ICM** – Intrinsic Curiosity Module.
- **NGU** – Never Give Up (combining multiple exploration bonuses).
- **BEBold** – Exploration via prediction error without collapse.
- **VIME** – Variational Information Maximizing Exploration.

Tree Search & Planning

- **MCTS** – Monte Carlo Tree Search.
- **MuZero** – Model-based RL with latent dynamics and MCTS.
- **AlphaZero** – Self-play + MCTS + deep networks.

Offline / Batch RL

- **CQL** – Conservative Q-Learning.
- **BCQ** – Batch-Constrained Q-Learning.
- **AWAC** – Advantage Weighted Actor-Critic.
- **IQL** – Implicit Q-Learning.
- **OPAL** – Offline Preference-based Active Learning.

Algo zoo 4

Imitation / Behavior Cloning

- **BC** – Behavioral Cloning.
- **DAgger** – Dataset Aggregation.
- **IQ-Learn** – Offline imitation without environment rewards.

—

Large-Scale / LLM-RL Specific

- **SFT** – Supervised Fine-Tuning.
- **RLHF** – Reinforcement Learning from Human Feedback.
- **DPO** – Direct Preference Optimization (LLM fine-tuning).
- **ORPO** – Optimal Reward Preference Optimization.
- **SLiC** – Score-based Learning with CLIP.
- **RRHF** – Rank Rewarded HF (heuristic post-RL reward shaping).

REINFORCE mit baseline

REINFORCE mit baseline =
anderer Name für Actor-Critic mit Monte Carlo

pole-actor-critic-mc.py

Baseline / Advantage

Der **Advantage** beschreibt, wie viel besser (oder schlechter) eine bestimmte **Aktion a** im Zustand s im **Vergleich** zum **durchschnittlichen Verhalten** gemäß der aktuellen Policy ist.

Formal:

delta = V'(s) - V(s) # eins gemessen, das andere geschätzt

A(s,a) = Q(s,a) - "baseline"

A(s,a) = Q(s,a) – V(s)

Baseline / Advantage

Man **vergleicht** beim **Advantage** einen konkreten **Quality**-Wert Q **gegen** den durchschnittlichen **Wert V.** Unabhängig wie dieser geschätzt wird nennt man letzteren **Baseline**.

d(s) = V'(s) – V(s)

```
TD scrittweise: V' = r_t + gamma * V_s+1  # mini Bellman Gleichung, Schätzung via VORHER V
MC episodic:    Q' = r_t + gamma * G  # Gewinnschätzung via Messung
```

A(s,a) = Q(s,a) – V(s) **Advantage von meiner *Action a* im Vergleich zum Durchschnitt (gemäß policy)**

A(s,a) ≈ Q(s,a) – V(s) = V-Netz Baseline V(s)

A(s,a) ≈ Q(s,a) – V(s) = Q-Netz Baseline E[Q(s',a')]

Baseline / Advantage

Der **Advantage** (Vorteil) beschreibt, wie viel besser (oder schlechter) eine bestimmte **Aktion a** im Zustand s im **Vergleich** zum **durchschnittlichen Verhalten** gemäß der aktuellen Policy ist.

Formal:

A(s,a) = Q(s,a) – V(s)

Je nachdem ob der Kritiker ein V oder Q Netz ist, schätzt man den jeweils **anderen Teil**:

A(s,a) = Q(s,a) – V(s) = (r + ɣ*V(s')) - V(s) **V-Netz Advantage** $V(s) = E[G]$ $G \approx r + \gamma * E[G]$

A(s,a) = Q(s,a) – V(s) = Q(s,a) **- ∑π(a|s)*Q(a|s)** **Q-Netz Advantage** $E[Q(s,a')] = \sum \pi(a) * Q(a|s)$

V(s') := $E[Q(s,a')] = \sum \pi(a) * Q(a|s)$ summiere alle Qualities gemäß W-keit dass man die Aktion auswählt

Baseline / Advantage

V-Netz Advantage G = r + g*E[G]

$A(s,a) \approx Q(s,a) - V(s) = (r + g*V(s')) - V(s)$

- **A2C / A3C (*Advantage* Actor–Critic):** Policy-Gradient-Methoden, die $V(s)$ als Baseline schätzen.
- **GAE (Generalized Advantage Estimation):** Erweiterung, die bias–variance trade-off reguliert durch Exponentielles Glätten der TD-Reste.
- ***PPO* (Proximal Policy Optimization):** Verwendet $V(s)$ als Kritiker und GAE zur Vorteilsschätzung.
- **TRPO (Trust Region Policy Optimization):** ebenfalls mit Value-baseline.

Q-Netz Advantage

$A(s,a) \approx Q(s,a) - V(s) = Q(s,a) - E[Q(s',a')]$

- **Dueling DQN:** Trennt Schätzung in $V(s)$ und $A(s,a)$, kombiniert sie zu $Q(s,a)$.
- **ACER (Actor–Critic with Experience Replay):** nutzt Q-basierte Advantage-Schätzung.
- **Q-Prop:** Policy-Gradient-Verfahren, das Q-Funktionen als Baseline integriert.
- **Soft Actor–Critic (*SAC*):** in der Entropie-maximierenden Form nutzt ein Q-Netz mit Advantage-Konstruktion gegenüber der log-Policy-Baseline.

Advantage

a) Monte Carlo **Advantage <u>replay</u>** "episodisches PG"

Wir spielen erst eine **<u>episode</u>** komplett durch und schauen uns *dann* unsere V Schätzungen an:

Idee: vergleiche beim standard REINFORCE, wie viel besser unsere Position als der Durchschnitt ist.

$$\mathbf{G_t} = \sum_i r_i + \gamma^{t+1} G_{\mathbf{t}-1}$$

$\mathbf{A_t} = G_{\mathbf{t}} - \mathbf{V}(s_{\mathbf{t}})$ "baseline" "Advantage"

- **REINFORCE mit Baseline** episodisch

$$\theta = \theta + \alpha * \nabla \log \pi(state_{\mathbf{t}}, action_{\mathbf{t}} | \theta) * \mathbf{A_t} \quad // \text{ statt G oder } \boldsymbol{\delta_t}$$

REINFORCE mit Baseline

pole-actor-critic-mc.py identisch zu
pole-REINFORCE-baseline.py

Hier kombinieren wir REINFORCE Algorithmus mit actor critic.

Advantage A2C

A2C – **<u>Advantage</u> Actor-Critic** (historische uminterpretierung)

Initialize policy π(a | s), value function V(s)

for iteration = 1, 2, ... do:

Collect N **trajectories** of length T using π_θ

for each **trajectory** τ = (s_0, a_0, r_0, ..., s_T):

for t = 0 to T–1:

A_t = **Q**(s_t, a_t) - **V**(s_t) , Q(s_t, a_t)≈ r_t + γ * V(s_{t+1})

A_t = r_t + γ * V(s_{t+1}) – V(s_t) # V Netz, estimate Q for Advantage estimate (💡≈ delta!)

Loss_π = – E_t [**log π(a_t | s_t) * A_t**] # policy loss E_t => **.mean() !**

Loss_V = E_t [**(V(s_t) – (r_t + γ * V(s_{t+1}))²**] # critic/value MSE loss

// **Loss_V** = E_t [**(V(s_t) – Q**(s_t, a_t))²] # when using DDQN dueling DQN

H = E_t [–log π_θ(a_t | s_t) * 1] # Compute *entropy* bonus (optional, week 3):

Update θ using **∇θ** (Loss_π – β * **H**)

Update ϕ using **∇ϕ** Loss_V

A3C – Asynchronous Advantage Actor-Critic. (Always Absolute Apsilon A8C)

Baseline RL

Baseline bei papers ≠ Baseline in Advantage (RL algos)

Baseline bei papers = Standard Algorithmus, gegen den verglichen wird oder einfach random() **Vergleichskurven**

Was kann man als **Baseline** in **papers** nehmen?

a) **random** policy
b) mit steigender Komplexität:
c) tabulatorisch
d) einfaches PG
e) Foundation model? Dreamer?
f) `from stable_baselines3 import A2C` (selber machen >> libraries)

stable baseline

stable_baselines3 library

⚠ pip install box2d

```
from stable_baselines3 import A2C # …
env = Monitor(gym.make("BipedalWalker-v3", render_mode=render_mode), log_dir)
algo = A2C("MlpPolicy", env, verbose=1)
algo.learn(total_timesteps=15000) # step = one action!!
```

```
['CartPole-v0', 'CartPole-v1', 'MountainCar-v0', 'MountainCarContinuous-v0', 'Pendulum-v1', 'Acrobot-v1', 'phys2d/
CartPole-v0', 'phys2d/CartPole-v1', 'phys2d/Pendulum-v0', 'LunarLander-v2', 'LunarLanderContinuous-v2', 'BipedalWalker-
v3', 'BipedalWalkerHardcore-v3', 'CarRacing-v2', 'Blackjack-v1', 'FrozenLake-v1', 'FrozenLake8x8-v1', 'CliffWalking-
v0', 'Taxi-v3', 'tabular/Blackjack-v0', 'tabular/CliffWalking-v0', 'Reacher-v2', 'Reacher-v4', 'Pusher-v2', 'Pusher-
v4', 'InvertedPendulum-v2', 'InvertedPendulum-v4', 'InvertedDoublePendulum-v2', 'InvertedDoublePendulum-v4',
'HalfCheetah-v2', 'HalfCheetah-v3', 'HalfCheetah-v4', 'Hopper-v2', 'Hopper-v3', 'Hopper-v4', 'Swimmer-v2', 'Swimmer-
v3', 'Swimmer-v4', 'Walker2d-v2', 'Walker2d-v3', 'Walker2d-v4', 'Ant-v2', 'Ant-v3', 'Ant-v4', 'Humanoid-v2', 'Humanoid-
v3', 'Humanoid-v4', 'HumanoidStandup-v2', 'HumanoidStandup-v4', 'GymV21Environment-v0', 'GymV26Environment-v0']
```

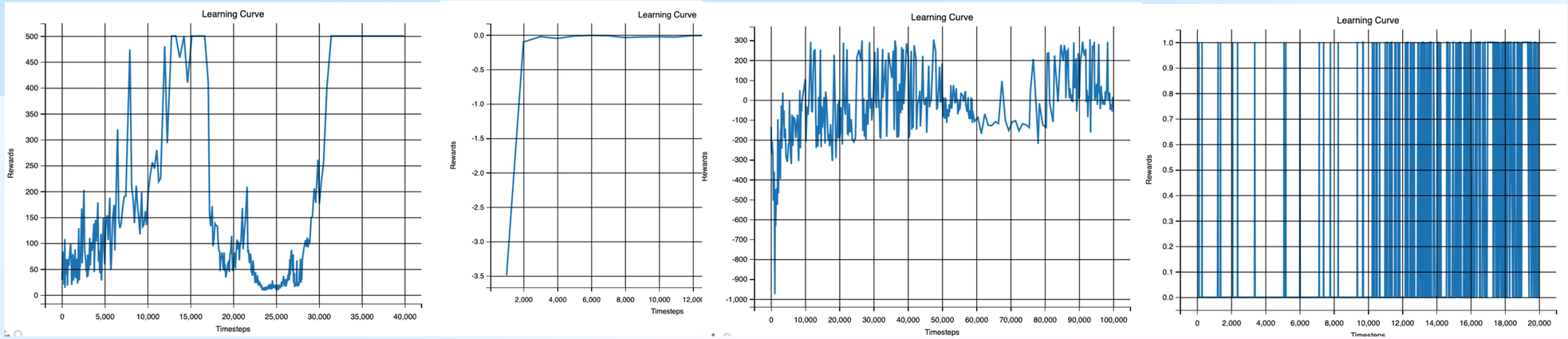
⚠ **DO *NOT* use mujoco examples (unstable, hard install)**
installiert eigenes python, alte versionen Abhängigkeit

stable baseline

stable_baselines3 library

⚠️ pip install box2d
from stable_baselines3 import A2C # …
⚠️ einige algos funktionieren nur für continuous, aber:
💡 **A2C, PPO schlucken alles**

für pole, mountaincar (c), lunar-lander(??), frozenlake … :



import gymnasium as gym; print([env for env in gym.registry])

selber machen >> libraries für Erkenntnisgewinn und Kontrolle.
Aber stable baseline sehr schnell! Aufgabe: wie schnell im **vergleich?**

Baseline RL

Baseline in RL

Motivation:
negative Q's in DPG sorgen für Verschlechterung beim Gradientenabstieg.

Ziehe baseline/bias **b** von Q ab falls Q und b negativ sind

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\left(\sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t)\right)\right]$$

Meistens **b=V(s)** => Advantage

---

replay ≠ buffer ≠ batch

replay bei Monte Carlo: Spiel speichern
replay buffer : mehrere Spiele speichern
batch : mehrere Daten speichern (kann unsortiert sein)

bei MC ist Ordnung wichtig, weil wir successive gain zählen:
gain = reward + gamma * gain

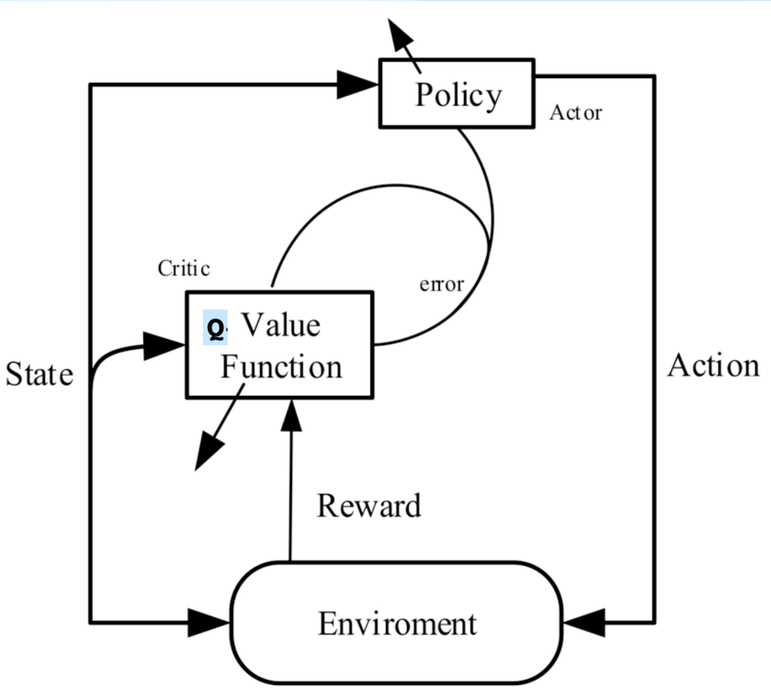
DPG

Deterministic Policy Gradient Methode
Silver et al., 2014

Actor Critic der die **Quality** bewertet!

deterministisch $\pi(s)=a$ statt $\pi(s|a)$ (softmax)Verteilung!
direct **continuous** action e.g. a in [-1 1]

```
def DPGStep(s_t, a_t, r_t, s_t+1, done):
    # Q-Wert für a_t im aktuellen Zustand
    Q_sa = critic(s_t, a_t)

    a_t+1 = actor(s_t+1).detach()
    Q_t+1 = critic(s_t+1, a_t+1).detach()
    target_t = r_t + γ * Q_t+1  if not done else r_t  # one-step Bellman backu

    a_pred = actor(s_t)
    q_value = critic(s_t, a_pred)
    actor_loss = -q_value  # 💡 maximize Q, so minimize -Q !
    critic_loss = (Q_sa - target_t)² # (MSE)

    # Update Schritt
    actorθ ← actorθ - α_actor · ∇ actor_loss
    criticθ ← criticθ - α_critic · ∇ critic_loss
```



DDPG

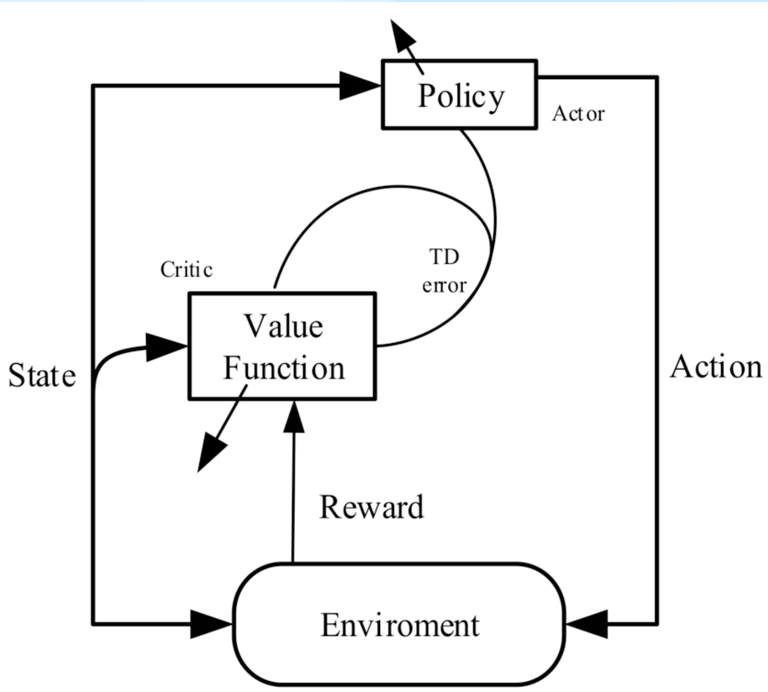
***Deep* Deterministic Policy Gradient** Methode

Lillicrap et al., 2016

Deep NNs (actor + critic)
Stabilisation tricks: **Replay buffer** + **target networks**
Exploration mechanism: Built-in action **noise** (OU) !

Scales to high-dimensional continuous control

Foundation for modern algos such as **SAC, TD3**



DDPG

Deep Deterministic Policy Gradient as **Foundation** for modern algos:

2018 **TD3** (Twin-Delayed DDPG) ① Twin critics with min-value target, ② actor updated every 2-3 critic steps, ③ Gaussian smoothing on target actions, Cures Q-overestimation, stabilizes training

2018 **D4PG** (Distributed **Di**stributional DDPG) Categorical critic (Z-distribution), N-step returns, distributed actors, PER; Higher throughput and better value estimates

2018 **SAC** (**Soft Actor-Critic**) Stochastic policy with entropy term, double critics; retains off-policy replay Robust exploration, state-of-the-art sample efficiency

2017 **DDPGfD** (DDPG from Demonstrations) **Dual replay** buffers (demo + agent), PER weighting Solves sparse-reward robotics tasks

2017 → MADDPG Centralized critic with all agents' actions, decentralized deterministic actors Mixed cooperative-competitive multi-agent control

2021 TD3 + BC **Behavior-cloning** loss weighted by advantage, feature normalization Strong minimalist baseline for offline RL

2021 DrQ-v2 Heavy image augmentation, n-step returns, DDPG backbone Pixel-based continuous control on one GPU

2020–24 TQC / VI-TD3 / VI-DDPG Quantile truncation or value-improved targets applied to TD3/DDPG critics Sharper value distributions, faster convergence

2024 **P-DDPG** (Prioritized + Param-noise) PER on transitions, parameter-space exploration noise More robust continuous-space agents

2025 DATD3 Depth-wise-attention encoder for observation histories built on TD3 Handles partial observability without RNNs

2025 DBOP-DDPG Swarm-style dung-beetle optimizer;) for actor pre-updates, tri-level priority replay Escapes local optima, accelerates convergence

NICHT SoTa: Dreamer (Hafner et al. 2019, 2020) is built around a world model + latent-space policy optimization

Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) is an **algorithm** which uses off-policy data and the Bellman equation to learn the Q-function, and uses the **Q-function to learn the policy**.

policy π via best action $a^*(s) = \arg\max_a Q^*(s,a).$

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}} \left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d) \max_{a'} Q_\phi(s',a') \right) \right)^2 \right]$$

d: done 1/0

DDPG can be thought of as being *Deep Q-learning* for *continuous* action spaces.

https://spinningup.openai.com/en/latest/algorithms/ddpg.html

DDPG

Deep Deterministic Policy Gradient.

```
Initialisiere Actor-Netzwerk μ(s|θ^μ) und Critic-Netzwerk Q(s,a|θ^Q)
Initialisiere Zielnetzwerke μ' und Q' mit gleichen Parametern
Initialisiere Replay-Buffer R

für jede Episode:
    Setze Umgebung zurück, initialisiere Startzustand s
    für jeden Zeitschritt:
        Wähle Aktion a = μ(s|θ^μ) + Noise (z.B. Ornstein-Uhlenbeck)
        Führe a aus, beobachte Belohnung r und neuen Zustand s'
        Speichere (s,a,r,s') in R
        Ziehe zufälliges Mini-Batch aus R
        Berechne Zielwert y = r + γ * Q'(s', μ'(s'|θ^μ') | θ^Q')
        Update Critic durch Minimierung von (Q(s,a) - y)²
        Update Actor mit Policy Gradient: ∇θ^μ J ≈ ∇a Q(s,a|θ^Q) ∇θ^μ μ(s|θ^μ)
        Soft Update der Zielnetzwerke:
            θ' ← τθ + (1−τ)θ'
```

DDPG

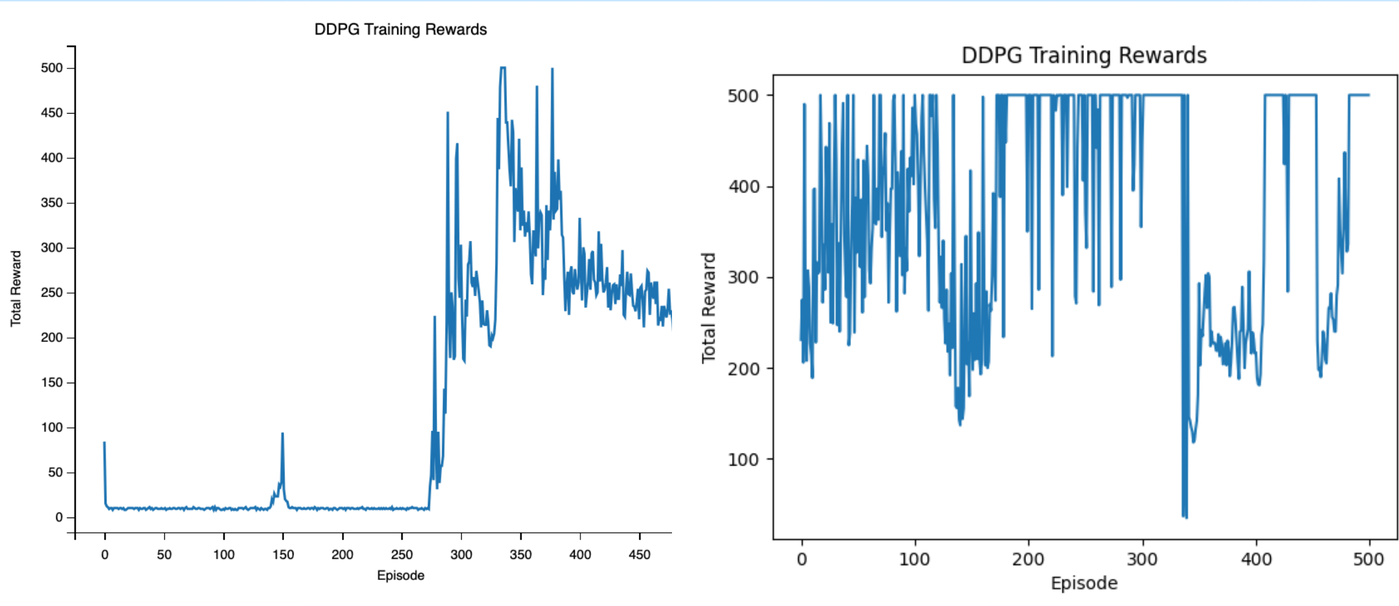
Deep Deterministic Policy Gradient.

Off-Policy, model-free Algorithmus zur Lösung von kontinuierlichen Steuerungsproblemen.

- Kombination aus:
  - Actor-Critic Architektur
  - Deterministic Policy Gradient Methode
- Verwendung von Replay Buffer und Target Networks
- Ursprünglich von **DeepMind** eingeführt (Lillicrap et al., 2015).

DDPG

```
def DDPGStep(s_t, a_t, r_t, s_t+1, done):
    a_t+1 = target_actor(s_t+1).detach()

    # Q-Wert der nächsten State-Action-Paarung aus target critic
    Q_t+1 = target_critic(s_t+1, a_t+1).detach()

    # Zielwert für critic (Bellman-Backup)
    y_t = r_t if done else r_t + γ * Q_t+1

    # Q(s_t, a_t) aus aktuellem critic
    Q_sa = critic(s_t, a_t)

    # Critic Loss (MSE)
    critic_loss = (Q_sa - y_t)²

    # Actor Loss: ∇a Q(s, a) |_{a=μ(s)}
    a_pred = actor(s_t)
    actor_loss = -critic(s_t, a_pred)

    # Gradient descent Schritte
    actorθ ← actorθ - α_actor · ∇ actor_loss
    criticθ ← criticθ - α_critic · ∇ critic_loss

    # Soft Update der Target-Netze
    for θ, θ_target in zip([actorθ, criticθ], [actorθ_target, criticθ_target]):
        θ_target ← τ · θ + (1 - τ) · θ_target
```



REINFORCE mit Baseline

- **AC REINFORCE mit Baseline** ist **episodisch**, lässt sich leicht in **REINFORCE** einbauen!

```
initialize actor
initialize critic  # for V(s), the baseline

for each episode:
    collect trajectory τ = [(s₀, a₀, r₁), (s₁, a₁, r₂), ..., (s_T, -, -)]

    compute returns:
        Gₜ = ∑ᵢ γ^{i-t-1} * rᵢ   for all t

    for each step t in trajectory:
        # Advantage estimate
        Aₜ = Gₜ - V(sₜ)  # critic 'baseline'

        # Actor & Critic update
        θ ← θ - α * ∇ -log π(aₜ | sₜ) * Aₜ
        φ ← φ - β * ∇ [V(sₜ) - Gₜ]²
```

REINFORCE continuous & episodic

initialize θ (actor), φ (critic)

for each **episode:**
collect **trajectory** τ = [(s_0, r_1), (s_1, r_2), ..., (s_T, -)]
compute deterministic actions: a_t = policy(s_t) / **all** = actor(states)

compute returns:
$G_t = \sum_i \gamma^{i-t-1} * r_i$ for all t

for each step t in trajectory:
$A_t = G_t - V(s_t)$ # advantage
actor_loss = - (all * advantages).mean()

Actor gradient (deterministic policy gradient)
$\theta \leftarrow \theta - \alpha * \nabla \pi(s_t)$ * **actor_loss**

Critic update
$\varphi \leftarrow \varphi - \beta * \nabla [V(s_t) - G_t]^2$

Aufgaben

a) adaptiere bisherige PG Lösungen (pole / NICHT: mountain-car) (vorher kopieren;)

   a) Achtung: PG ist episodisch, **AC** Grundversion ist step basiert, muss etwas umgeschrieben werden:

b) Nutze Advantage in REINFORCE (haben wir schon: pole-REINFORCE-baseline.py)

c) Implementiere Actor Critic für gym Problem **LunarLander**:

   a) erst nur PG (wie gestern),

   b) wenn es läuft **AC** hinzufügen: per Hand oder copy paste *NICHT* GPT (nur zum Erklären!)

   c) Nutze **Regularisierung** z.B. **replay / batch** (mehrere Schritte/Spiele)

   d) Lösung: *lunar-actor-critic-mc.py*

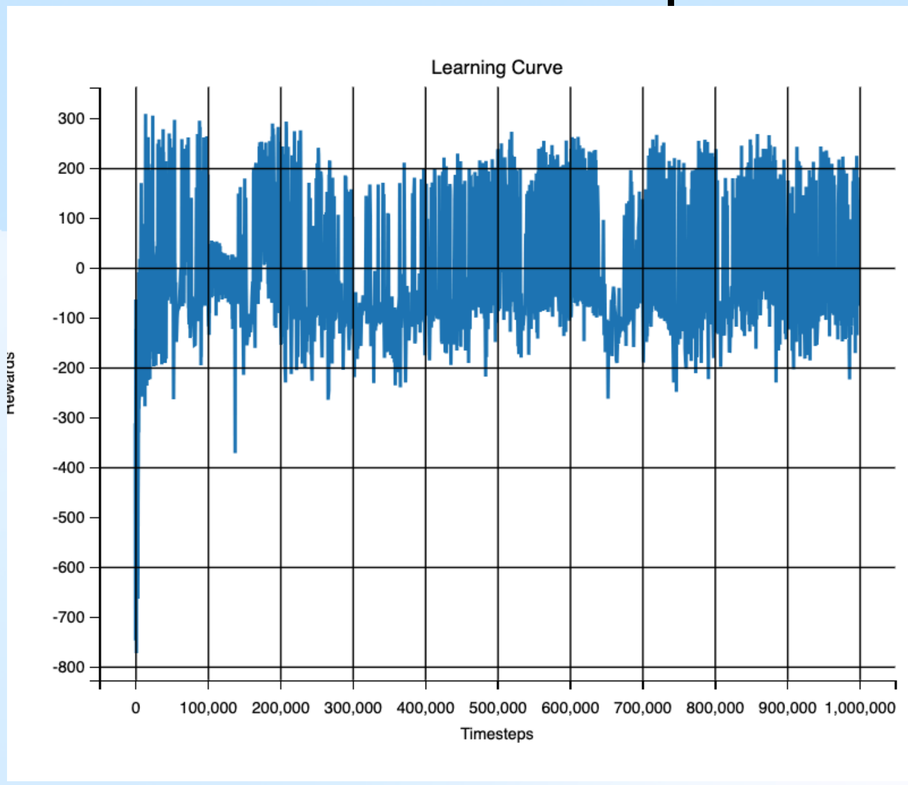
d) Vergleiche stable baseline algo für obiges und gym env Eurer Wahl!

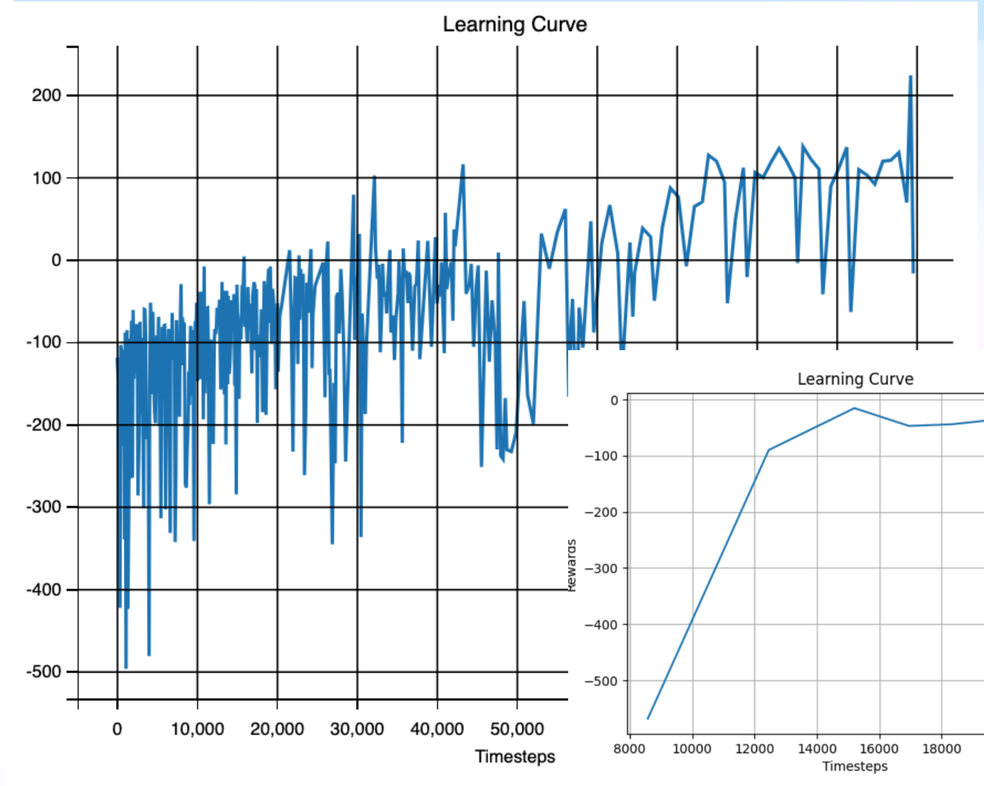
e) Aufgabe: wie schnell sind stable baseline im Vergleich zu eigenem algo?

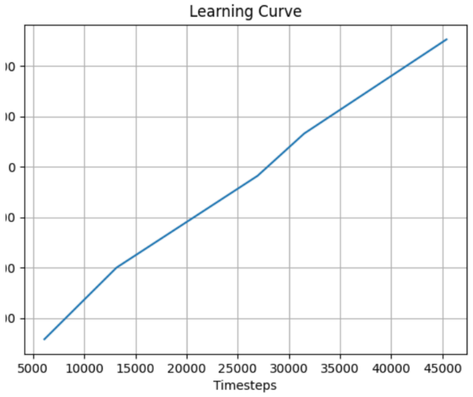
Aufgaben

Aufgaben Tag 8

lunar lander nur halb gelernt mit stable baseline A2C, PPO ok'ish
custom reward helps in carracer-v3 !







Probleme beim Trainieren

Tensor mehrfach beim loss benutzen:
"**double** usage Fehler"
RuntimeError: Trying to backward through the graph **a second time**

Tritt auf wenn man Tensor in batch/replay steckt und mehrfach raus holt.

Ausweg: **with torch.no_grad**(): Gar kein Gradient, z.B. evaluierung
oder tensor**.detach()**
löst tensor von Gradienten an der richtigen Stelle: nach der ersten Benutzung.
Tensor geht dann nur als **unveränderliche Konstante** ein

Kein tensor für Gradient vorhanden. "zu viel detach;)"
"element 0 of tensors does not require grad and does not have a grad_fn"

Drauf achten, welchen Wert man mit backprop ändern möchte und welchen nicht.

Problemen beim Trainieren

Exploding gradients

RuntimeError: probability tensor contains either `**inf**`, `**nan**` or element < 0

Gewichte haben sich aufgeschaukelt.

Ausweg: Alten stand löschen.

Ausweg: **Regularisierungstechniken** aus ML (dropout, skip, layer norms, …)

Ausweg: **Loss kontrollieren, gegebenenfalls clip/clamp abschneiden z.B. um -1,1**
(breakpoint / print / tensorboard)

Problemen beim Trainieren

Dimension passt nicht: einfach breakpoint und tensoren angucken.

Fehlermendungen etwas kryptisch aber **0D**, 1D, **2D** heisst: Dimension passt nicht!
RuntimeError: both arguments to linear need to be at least 1**D**, but they are 0D and 2D

Besonders bei batch Zusatzdimension
[batch, höhe, breite]
[batch, höhe * breite]

Ideale Welt wie in Physik: jede Dimension erhält einen Namen.
https://docs.pytorch.org/docs/stable/named_tensor.html

damit kann man sich dann manuelles squeeze, unsqueeze, stack, unstack, reshape, ... sparen

Gegenthese zu hässlichem torch.einsum('bij,bjk->bik', As, Bs)

Problemen beim Trainieren

Im Zweifel einfach Rollback und mit funktionierendem Beispiel neu anfangen.

git revert oder (pycharm) history

Extra Techniken

OU - **Noise** (in DDPG)

Over engineering smell! **Macht man heute so nicht mehr!**

```
theta, sigma = 0.15, 0.2
x = 0  # internal state
def ou_noise(x, mu=0, dt=1e-2):
    return x + theta * (mu - x) * dt + sigma * np.sqrt(dt) * normal()
```

```
# Gaussian noise example
with torch.no_grad():
    action = actor(state_tensor).item()
    noise = np.random.normal(0, 0.1)  # mean 0, std 0.1
    action += ou_noise(x)
    action = np.clip(action + noise, -1, 1)
```

Extra Techniken

Exploration in PG:
Policy Gradient Methoden haben schon Aktions*wahrscheinlichkeiten*, aber eventuell können diese zu nahe 0 liegen.
Auswege:

a) Softmax with temperature

```
DivideByTemperature(), # optional: for more exploration
nn.Softmax(dim=-1)
```

```
class Divide(nn.Module):
   def __init__(self):
      super(Divide, self).__init__()

   def forward(self, x):
      return x / temperature
```

b) uniform random epsilon?

```
def epsilon_policy(state):
    if random() < epsilon:
        action_probs = torch.ones(n_actions) / n_actions # uniform random
    else:
        action_probs = policy(state)
    return action_probs
```

RuntimeError: element 0 of tensors does not require grad and does not have a grad_fn TODO? nein, gleich richtig:

c) Entropie loss += - log π *1

Vokabular

PG

AC

value ≈ moral ("böser actor") source: Torsten

delta (TD error / reward)

advantage

💡 **Delta is a one-step estimate of the advantage!**

baselines

trajectory ≈ episode

single replay : one trajectory **τ** (so wie bisher)

replay buffer : batch of trajectories **D**aten $\mathbf{D_i = (\tau_0, \ldots, \tau_n)}$

episodic replay buffer : - " -

step replay buffer : batch of (s,a,r,s') single independent steps

DDPG (V-net-critic algo von 2015 muss man den kennen?)