Hochschule Karlsruhe
University of
Applied Sciences

Fakultät für
Maschinenbau und
Mechatronik

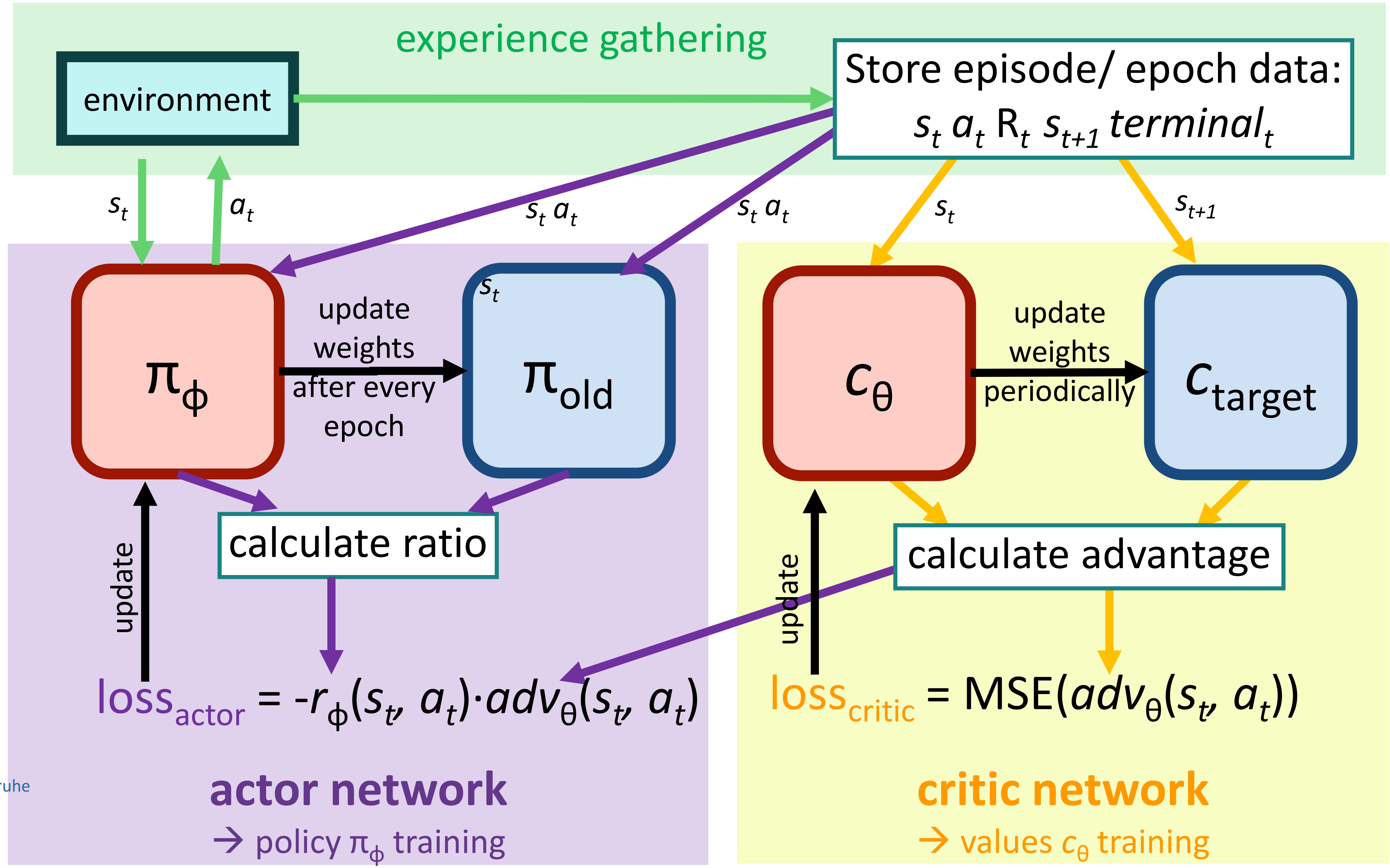
Projektarbeit Roboterprogrammierung

Proximal Policy Optimization (PPO) for Continuous Action Spaces

Robin Wolf, Mathias Fuhrer

16.01.2024

PPO - Proximal Policy Optimization



Discrete vs. continuous actionspace

discrete

- A probability is calculated for each discrete action

action (discrete)	left	0	right
probability	P(left) = 0,6	P(0) = 0,1	P(right) = 0,3

→ Actor-Net has an output for every discrete action

continuous

- An action probability distribution is calculated for each "degree of freedom"

action (continuous)	left ... 0 ... right
probability distribution	

→ Actor-Net has two outputs for each "degree of freedom" (mean and standard deviation)

main changes in the implementation:

- actor network
- act method → how to sample actions to explore the enviroment
- learn method → get_actor_gradients/ sampeling actions to calculate actor loss

Mountain Car Environment

Discrete:

Observation Space (2 dimensional):

- position of the car along the x-axis [-1,2 … 0,6]
- velocity of the car [-0,7 … 0,7]

Action Space (3 dimensional):

- 0: Accelerate to the left
- 1: Don’t accelerate
- 2: Accelerate to the right

Length of one episode is 200 steps

Continuous:

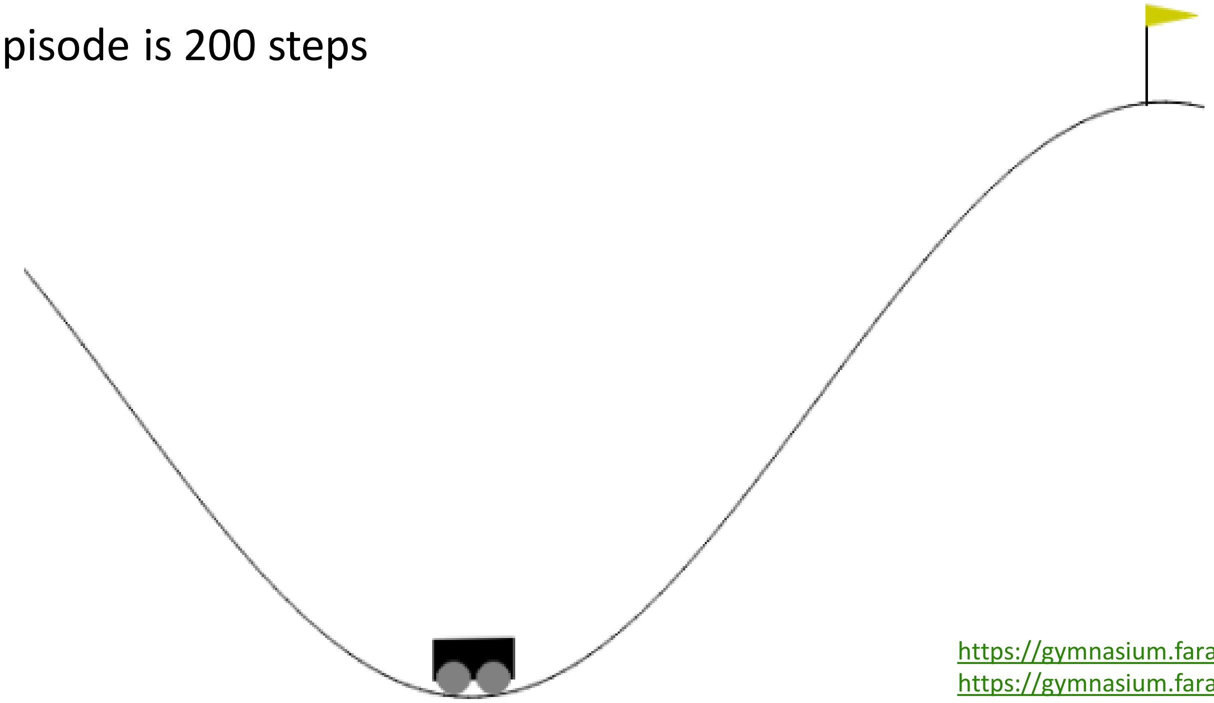
Observation Space (2 dimensional):

- position of the car along the x-axis [-Inf … Inf]
- velocity of the car [-Inf … Inf]

Action Space (1 dimensional):

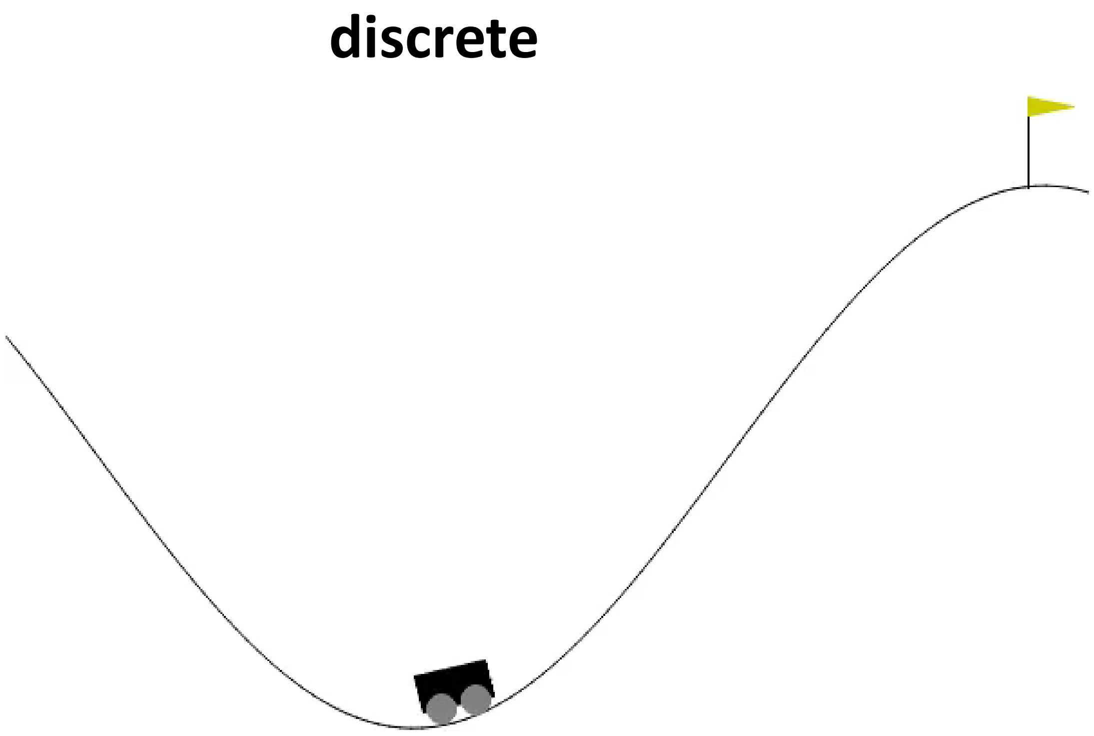
- force applied to the car [-1 … 1]

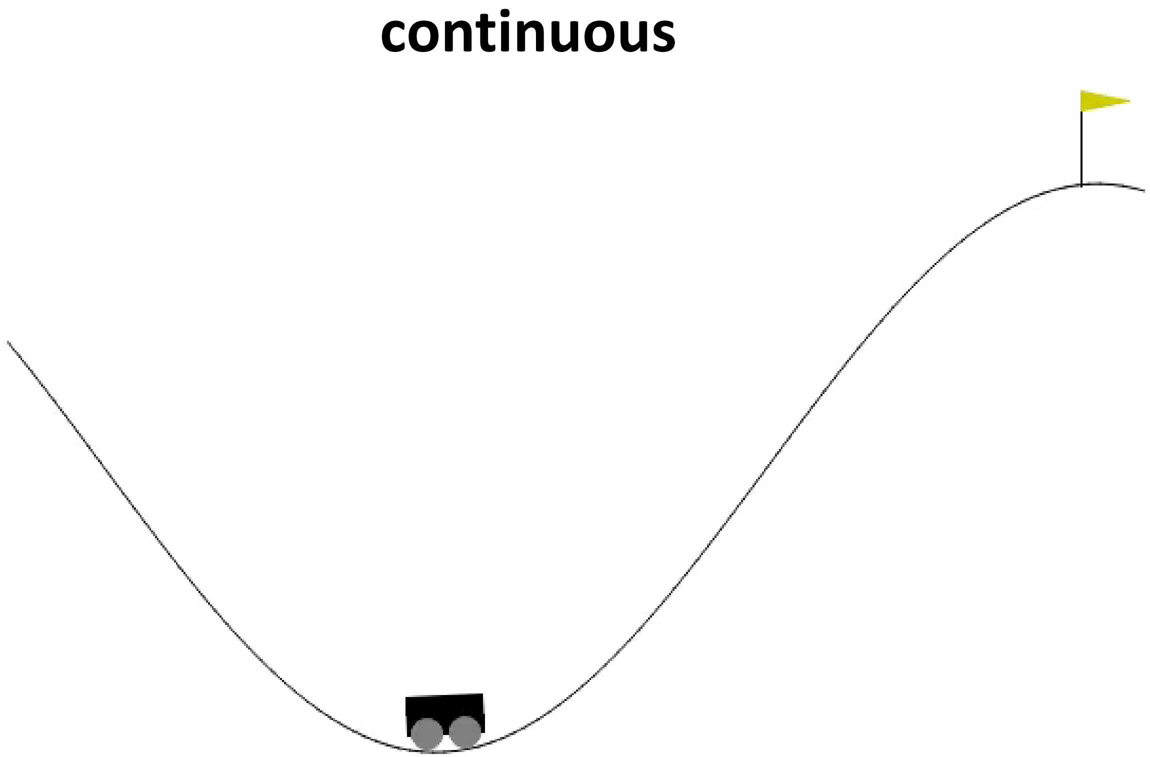
Length of one episode is 999 steps



https://gymnasium.farama.org/environments/classic_control/mountain_car/
https://gymnasium.farama.org/environments/classic_control/mountain_car_continuous/
(Abgerufen am 12.01.2024)

After training in the original environments …





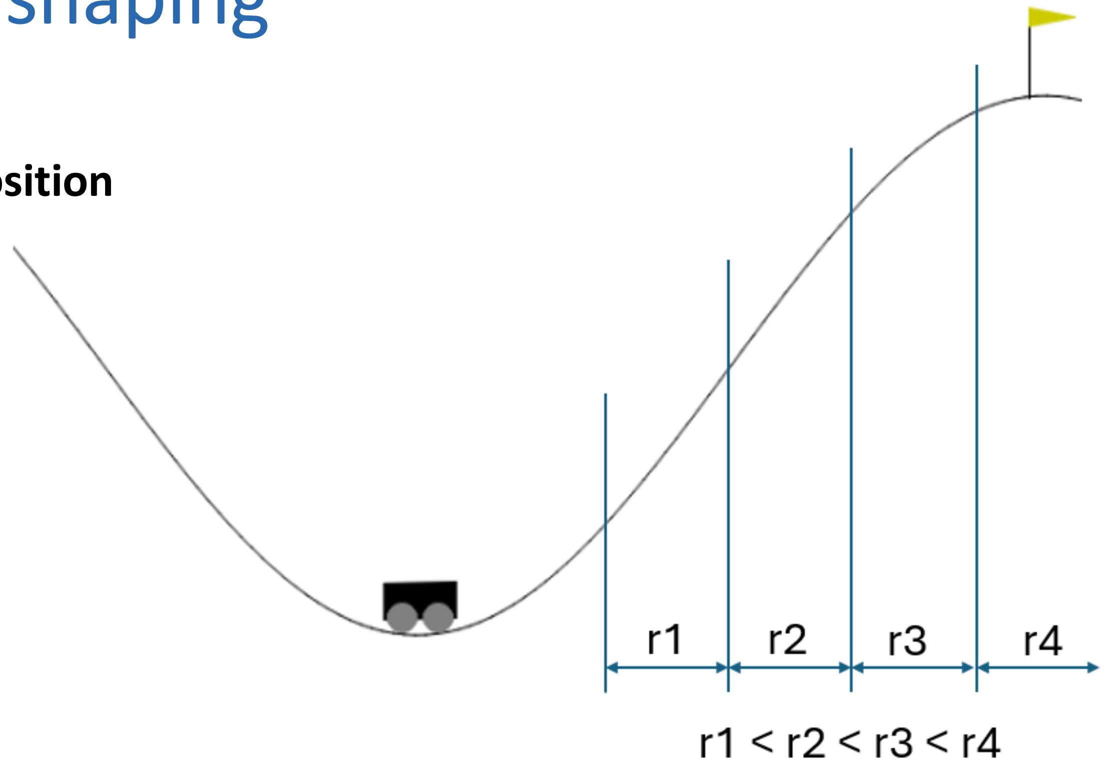
Discrete: negative reward of -1 at each timestep

Continuous: negative reward of -0.1 * $action^2$ at each timestep, positive reward of +100 added if the car reaches the goal

→ Unlikely that car will make it into the goal by chance → reward shaping

Reward shaping

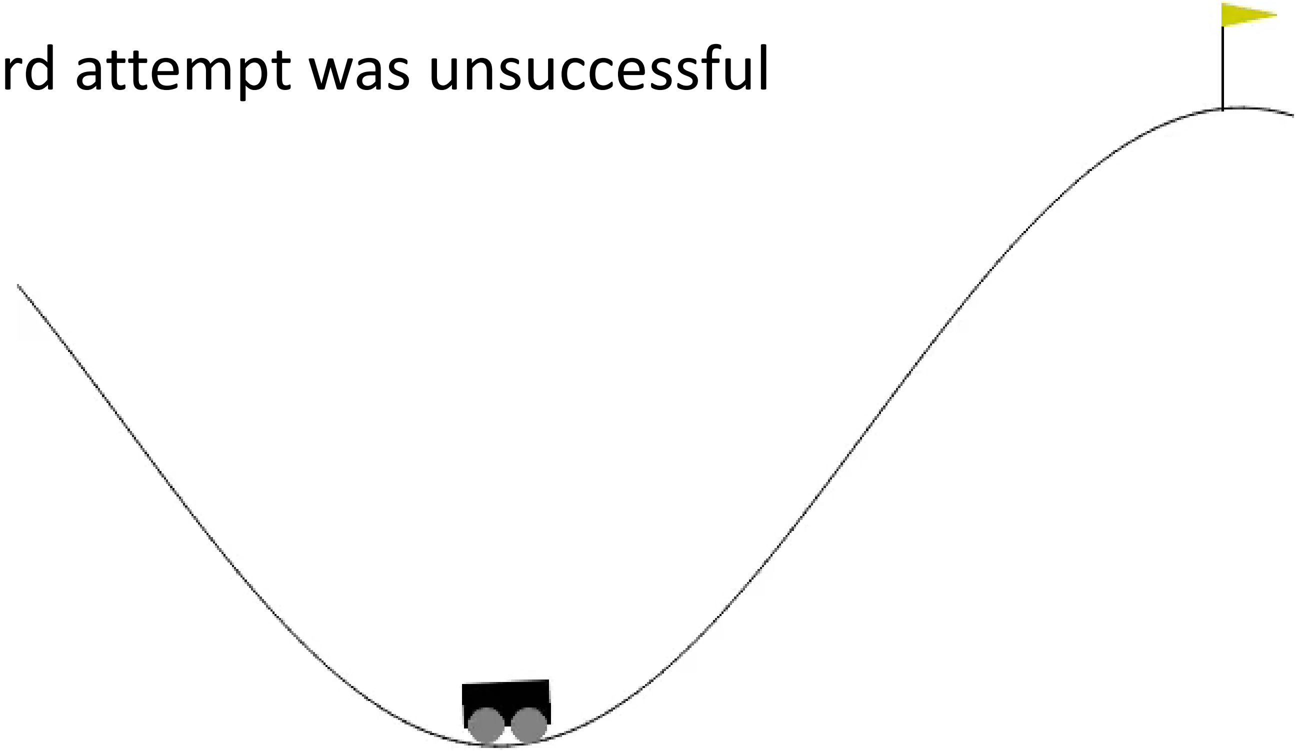
Attempt 1: position



Attempt 2: velocity
→ reward is higher when velocity is higher

After reward shaping

Position-based reward attempt was unsuccessful

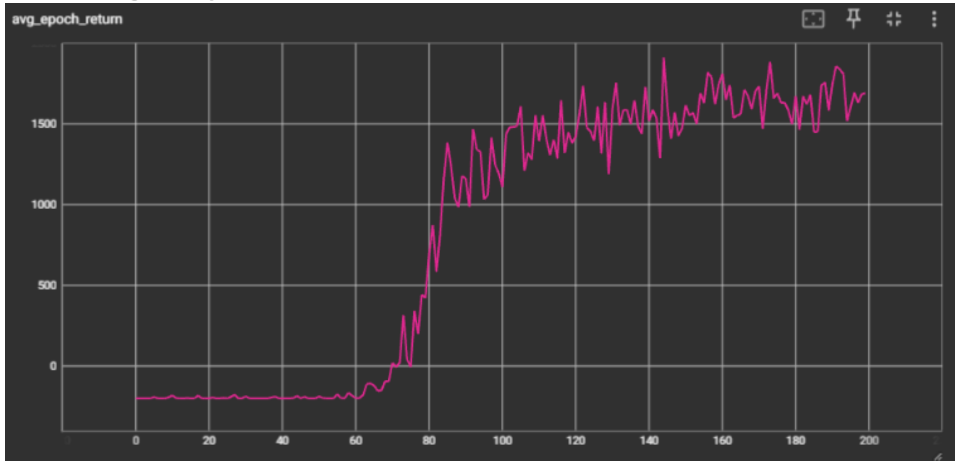


- tries to drive up the right hill without gaining momentum on the left hill
- to get up the mountain, the agent would have to accept a temporarily worse reward
  → remains at a local (reward) maximum

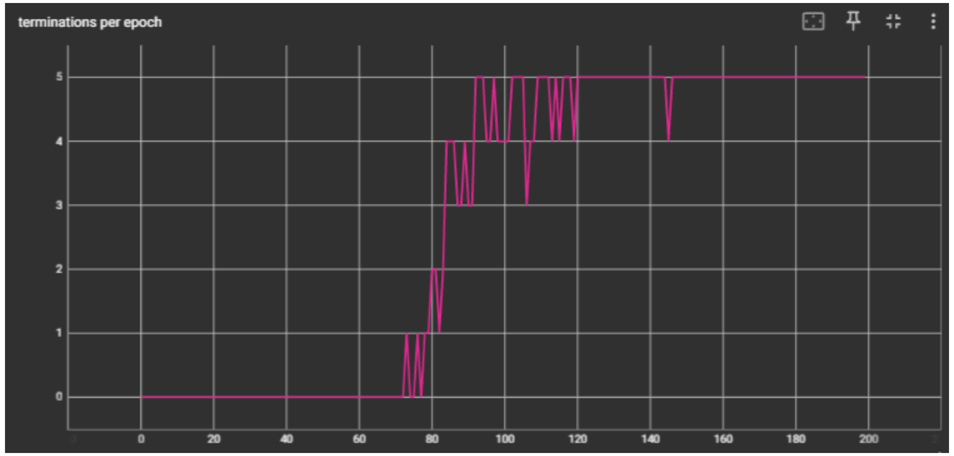
After reward shaping

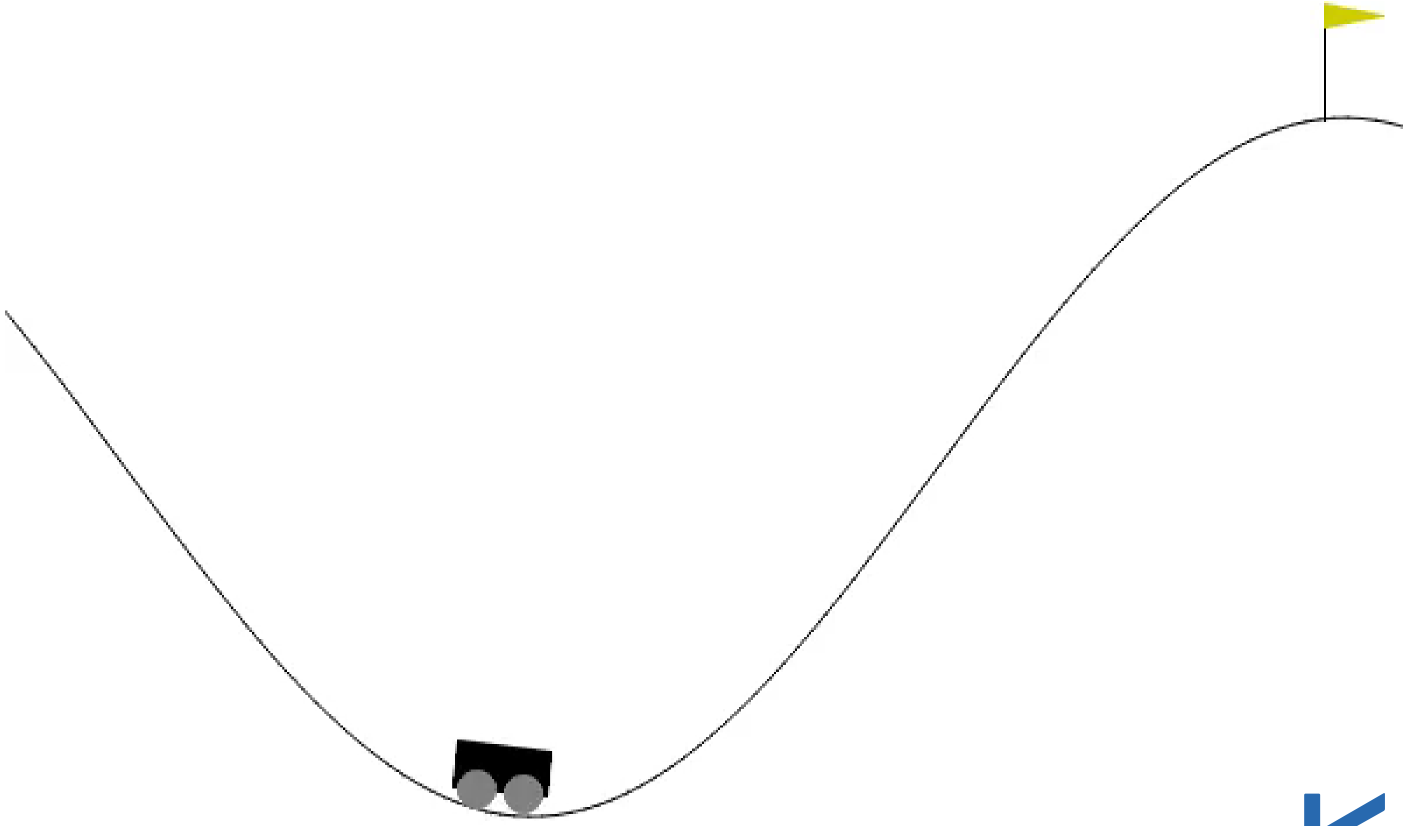
Velocity-based reward attempt was successful (with discrete and continuous action space)

Average epoch return:



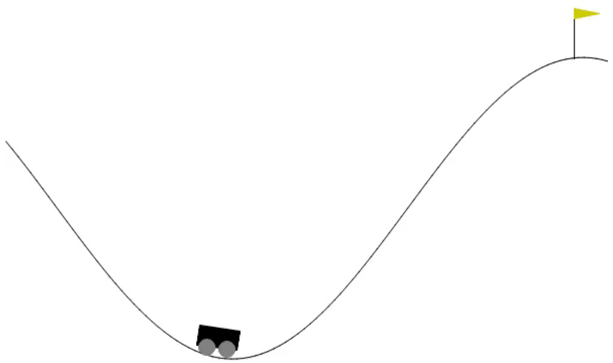
Terminations per epoch (one epoch has 5 episodes):

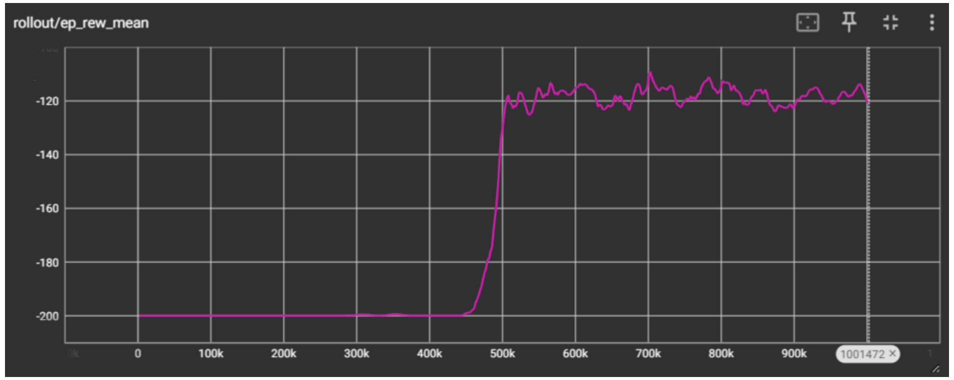




Discrete Mountain Car with stable baselines3

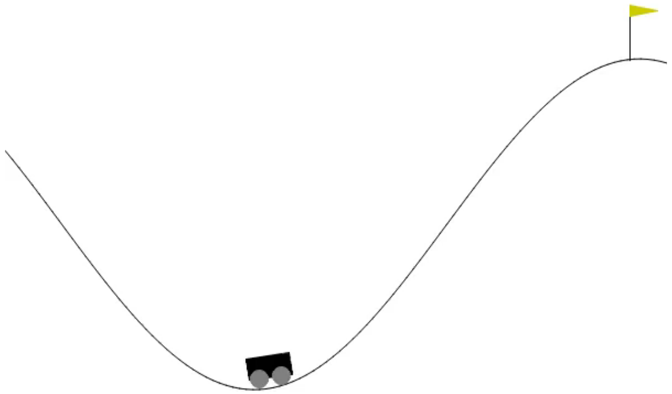
Standard environment without reward shaping

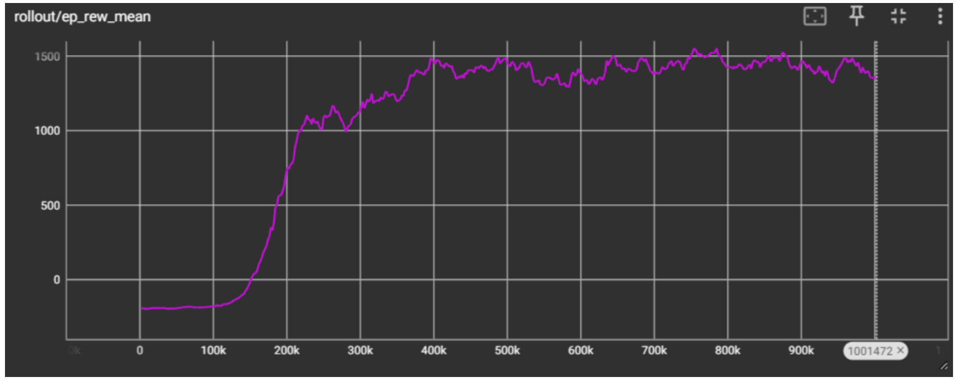




- Works well if goal is reached once, but needs lot of training steps

environment with reward shaping



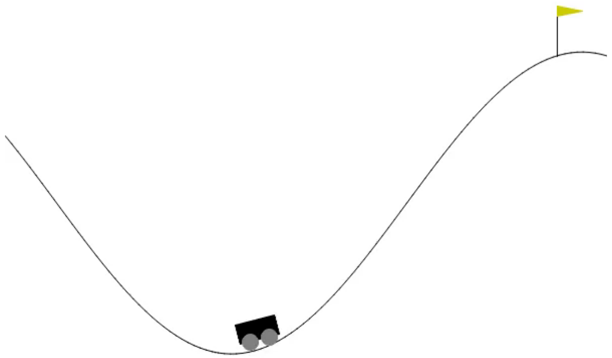


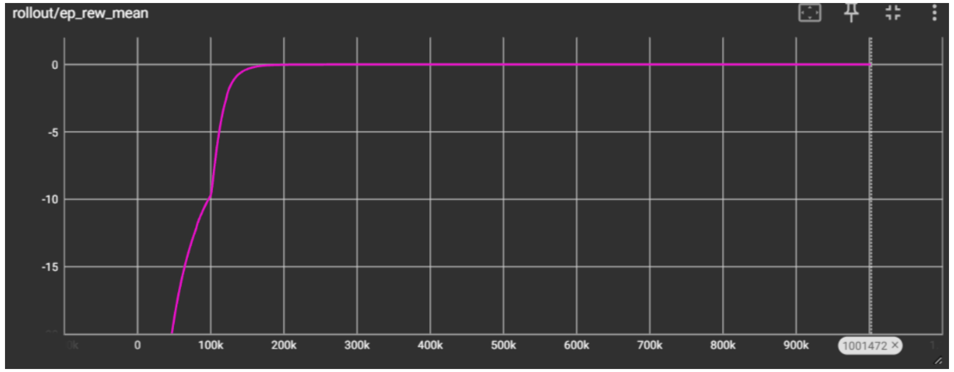
- only velocity attempt terminates
- faster learning at nearly equal performance

default parameters from BaselinesZoo (by DLR) were used

Continuous Mountain Car with stable baselines3

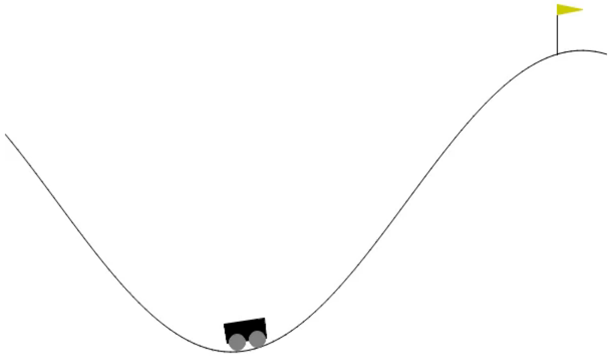
Standard environment without reward shaping

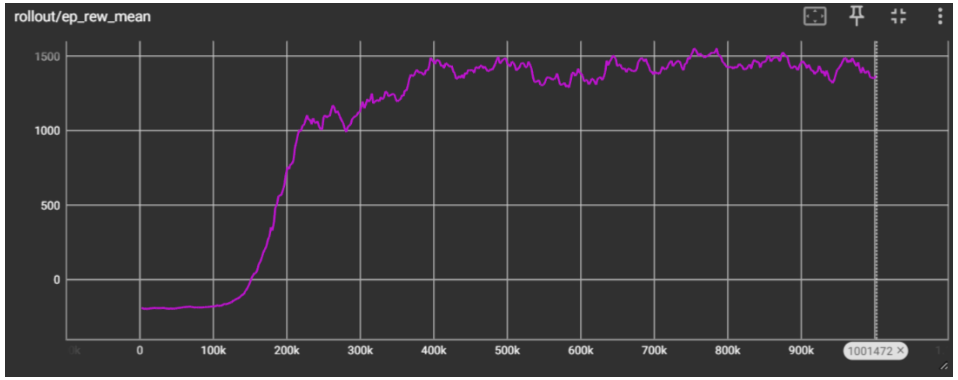




- Agent learns to do nothing
- Doesn‘t explore the positive reward at goal
- Local loss minimum at reward = 0

environment with reward shaping





- Only velocity attempt terminates
- Needs more attempts to reach goal in comparison to the discrete agent
- Balance between positive and negative rewards

default parameters from BaselinesZoo (by DLR) were used

Hopper Environment

Part of the MUJOCO environments → physics engine for multi joint control in robotics

Observation Space (11 dimensional):

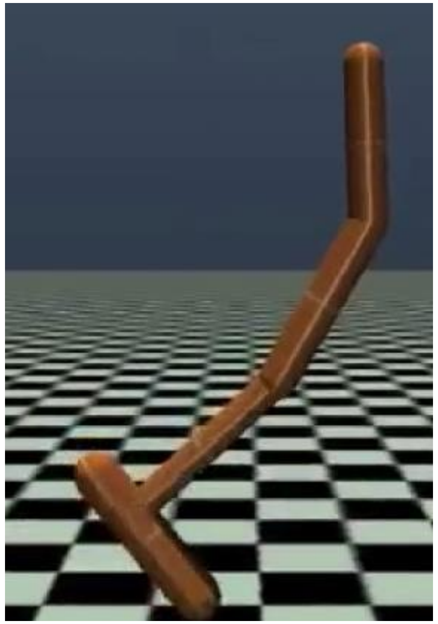
- Height of the hopper [-Inf … Inf]
- Angle of all joints and the top [-Inf … Inf]
- Angular velocity of all joints and the top [-Inf … Inf]
- Velocity of the top in X and Z of the world [-Inf … Inf]

Rewards = sum of:

- Healthy reward (not terminated)
- Forward_reward: positive if hopper hops to the right
  - *(forward_reward_weight * (x before action – x after action)/dt*
- Ctrl_cost: penalizing big actions
  - *Ctrl_cost_weight * sum (action²)*

Action Space (3 dimensional):

- torque applied to the top joint [-1 … 1]
- torque applied to the leg joint [-1 … 1]
- torque applied to the foot joint [-1 … 1]



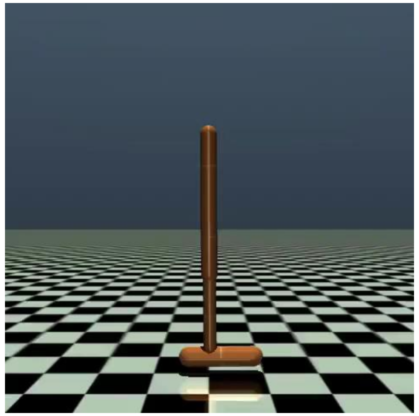
Episode End:

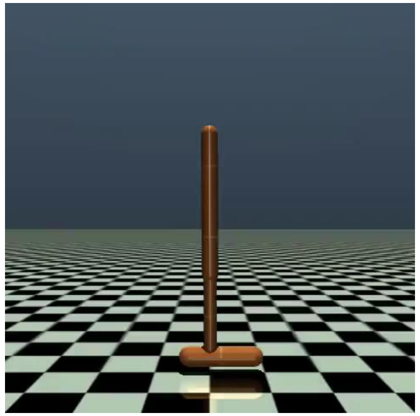
- Termination if hopper is unhealthy:
  - hopper has fallen (healthy z range)
  - Angle of the tigh joint is to big (healthy angle range)
  - All other observations are out of range e.g. hopper leaves the enviroment (healthy_state_range)
- Truncation if episode step >= 1000

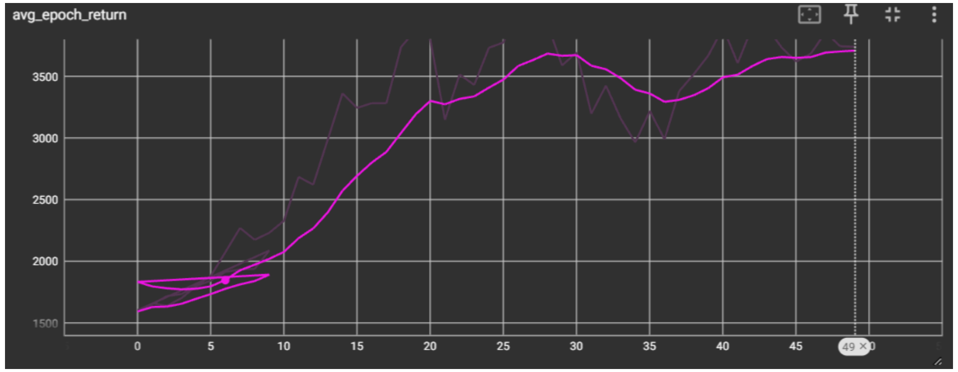
https://www.gymlibrary.dev/environments/mujoco/hopper/ (Abgerufen am 13.01.2024)
https://mujoco.org/ (Abgerufen am 113.01.2024)

Hopper agents

Our implementation

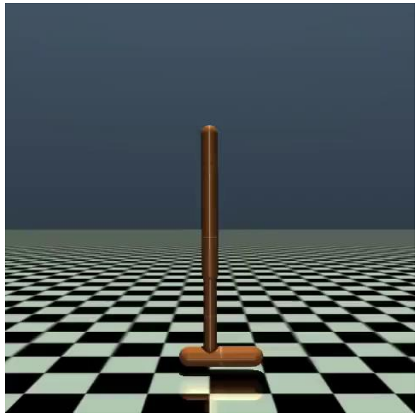


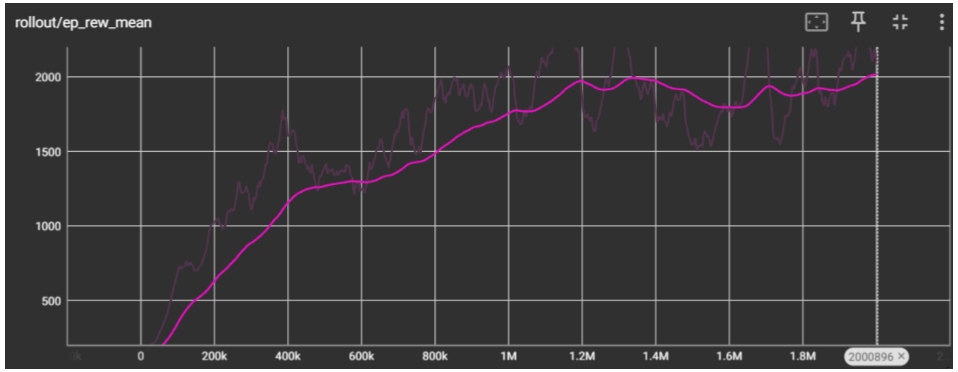




- Standard Implementation doesn‘t work at all
- Changing hyperparameters → no improvement
- Implementing some details from → improvement

Stable baselines 3





- Successful training and good performance in the complex enviroment
- Slightly improvements with more training expected to be possible

default parameters from BaselinesZoo (by DLR) were used

Main differences in the implementations

Recognized a huge difference in performance of the used algorithms, although they follow the same concept → WHY?

Our Implementation:

- Use of frozen networks as targets/ baselines
- Train the Actor and Critic with different losses
  - Both networks are independent
  - Critic: Value Estimation with TD-Learning
  - Actor: Policy Estimation with Policy-Gradient
- Policy (mean and std) are both outputted by the Actor network
- Only standardized advantage

Stable Baselines Implementation:

- Use of saved variables from the last rollout as baselines instead of frozen networks
- Train Actor and Critic with the same combined losses
  - Both networks are separated, but dependent
  - Use of entropy and weight factors to weight the different parts of the combined loss
- Only the mean of the policy is outputted by the Actor network, std is implemented as a trainable variable
- Observations, reward and advantage are standardized

We implemented some features from stable baselines and archieved some improvements on the hopper enviroment.

Combining networks and discarding the frozen nets not implemented because of limited time and computing power.

https://github.com/DLR-RM/rl-baselines3-zoo/tree/master (Abgerufen am 13.01.2024)
https://github.com/openai/baselines/tree/master/baselines (Abgerufen am 13.01.2024)

Hochschule Karlsruhe
University of
Applied Sciences

Fakultät für
**Maschinenbau und
Mechatronik**

Overall Conclusions/ Lessons Learned

- Conversion of a discrete algorithm to a continuous possibly by changing the policy structure (actor network) and some sampeling methods
- Performance depends on small implementation details, not only on the algorithm itself (see hopper)
  - There are a lot possibilities to implement the same algorithm → huge variablity
- Reward Shaping helps to solve difficult environments (with destructive reward)
- NaN Issue - for complex environments (e.g. hopper) some of the actor weights became 0
  - Units get "deactivated", Adam Optimizer fails and outputs a NaN – value
  - "Solved" by using leaky_relu instead of relu as activation of the hidden layers (weight = 0 is much less possible)
- Avoid calculating a fraction, use log difference instead → better numerical stability
- Standardize observations in experience gathering can be crucial, if actions get out of bounds [-1 … 1] to often
- Tuning hyperparameters is only for fine-tuning → using defaults worked out to be always the best

Trainingsparameter Mountain Car

```
# Parameter for the actor and critic networks
actor_learning_rate = 0.00025   # learning rate for the actor
critic_learning_rate = 0.001    # learning rate for the critic

# Parameter for the agent
gamma = 0.99                    # discount factor
epsilon = 0.1                   # clip range for the actor loss function

# Parameter for training
epochs = 50                     # number of learning iterations
n_rollouts = 5                  # number of episodes/ rollouts to collect experience
batch_size = 8                  # number of samples per learning step
learn_steps = 16                # number of learning steps per epoch
```