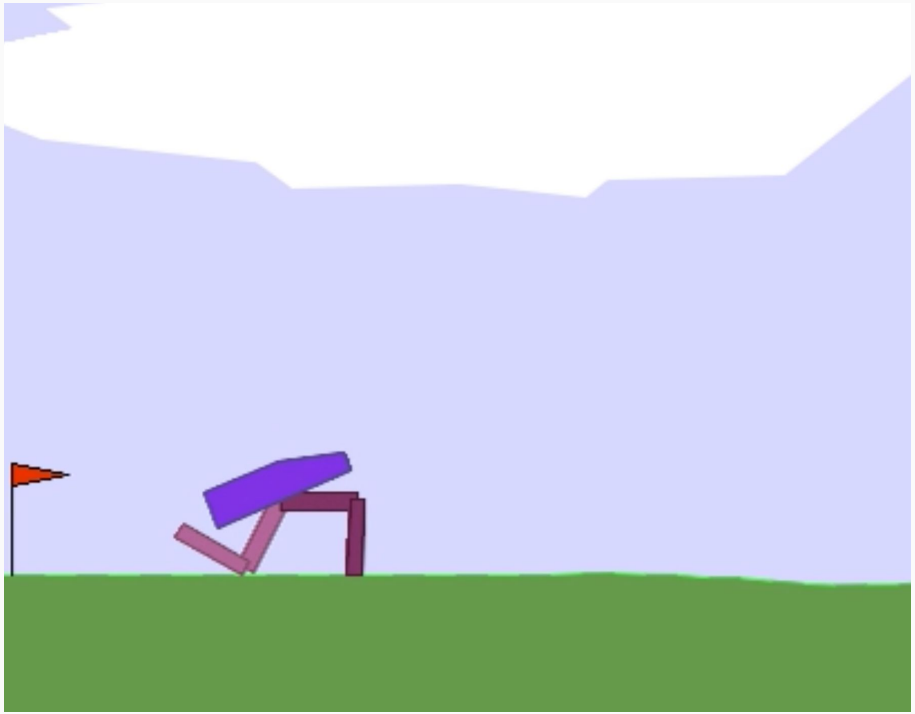
Deep Deterministic Policy Gradients

Teaching a robot how to walk

Jules Kreuer, Roman Machacek, Jonas Müller
08.05.2023



Continuous Control Problem

- Agent interacts with environment in discrete time steps, at each step agent observes c. state and performs action, receiving reward from the environment

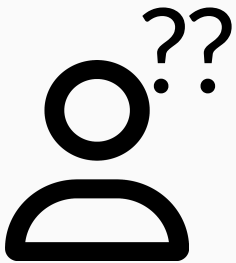
- MDP <**S**, **A**, **R**, **P**>. State, Action space, Reward function and state transition function P

- Return is cumulative reward discounted by gamma
- Policy function determines behavior of the agent
- Value functions **V**, **Q** represent expected returns by following the policy

$$R = \sum_{t=0}^{\infty} \gamma^t r_t,\ \gamma \in [0, 1)$$
$$\pi : S \longrightarrow A.$$
$$V_\pi : S \longrightarrow \mathbb{R},\ Q_\pi : S \times A \longrightarrow \mathbb{R}$$

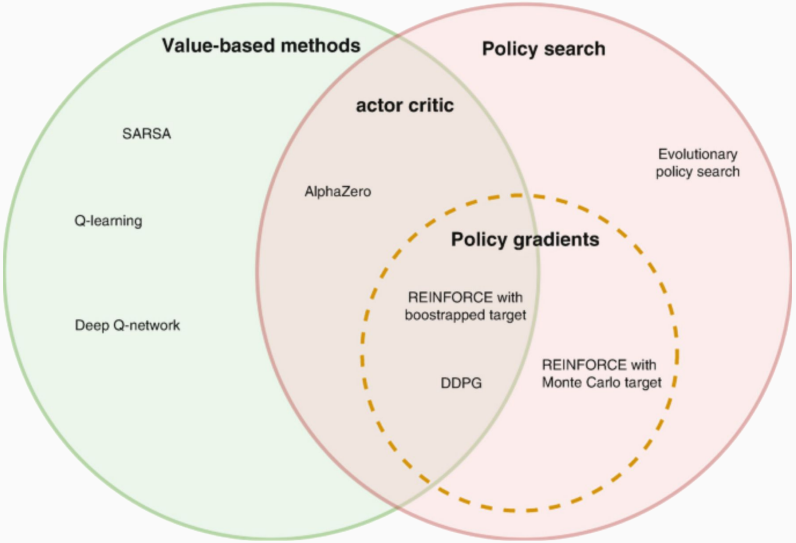
Finding an optimal policy

- How do we find a policy that would maximize the expected return
- Two approaches
  - Value based methods
  - Policy based methods
  - Actor-Critic



Finding an optimal policy

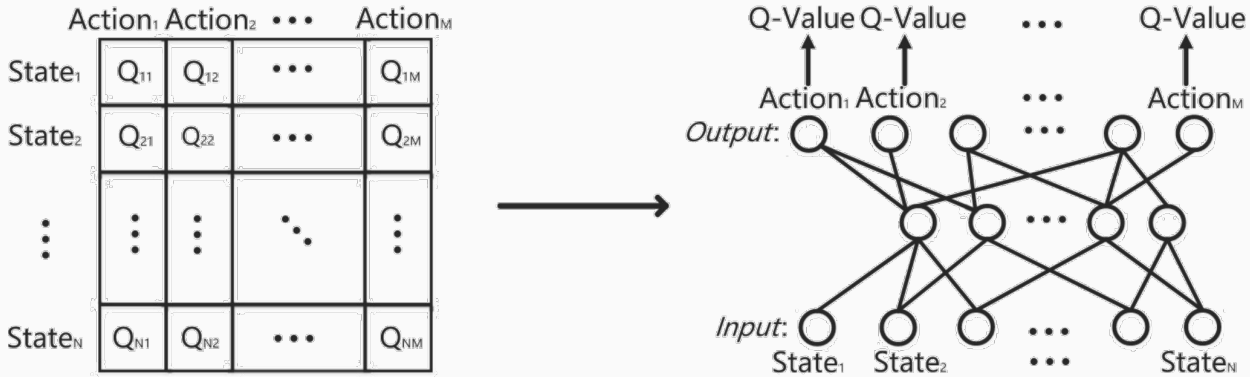
- How do we find a policy that would maximize the expected return
- Two approaches
  - Value based methods
  - Policy based methods
  - Actor-Critic



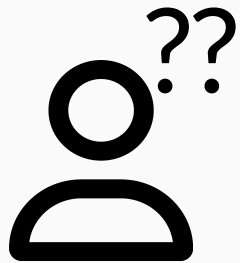
Aske Plaat (2022) , Policy-Based Reinforcement Learning

Value based methods

- Try to approximate optimal value functions **V***, **Q***
- Evaluation of states is possible through these functions
- Policy is implicit (take an action leading to the highest value)
- Q-learning, DQN



Ji Zhang et al. (2021). CDBTune: An efficient deep reinforcement learning-based automatic cloud database tuning system. The VLDB Journal. 30. 10.1007/s00778-021-00670-9.



Value based methods

- Try to approximate optimal value functions **V***, **Q***
- Evaluation of states is possible through these functions
- Policy is implicit (take an action leading to the highest value)
- Q-learning, DQN

- **A**: Not feasible due to large amount of states, actions („continuous space”)

Policy based methods

- Approximate optimal policy directly
- Model our policy, ie. NN(θ)
- Optimize our network as to maximize the expected return
- Policy gradient method

$$J(\theta) = v_\pi(s_0)$$

$$J(\theta) = \sum_{s \in \mathcal{S}} d^\pi(s) V^\pi(s) = \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}} \pi_\theta(a|s) Q^\pi(s,a)$$

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[Q^\pi(s,a) \nabla_\theta \ln \pi_\theta(a|s)]$$

Policy based methods

- Approximate optimal policy directly
- Model our policy, ie. NN(θ)
- Optimize our network as to maximize the expected return
- Policy gradient method

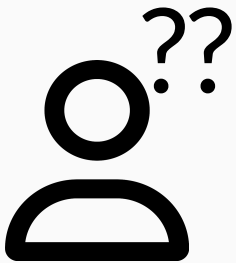
$$Policy\ gradient: E_{\pi}[\nabla_{\theta}(log\pi(s,a,\theta))R(\tau)]$$

Policy function — Score function

$$Update\ rule: \Delta\theta = \alpha * \nabla_{\theta}(log\pi(s,a,\theta))R(\tau)$$

Change in parameters — Learning rate

Thomas Simonini (2018). An introduction to Policy Gradients with Cartpole and Doom.



Policy based methods

- Approximate optimal policy directly
- Model our policy, ie. NN(θ)
- Optimize our network as to maximize the expected return
- Policy gradient method

- **A**: Complex environment, sampling from multiple trajectories (at start uniform distribution over actions) -> Highly variable behaviors. During training we reshape densities, for that we need to observe many outcomes of actions, states

Policy gradients - Advantage

- Many improvements for reducing variance - **Advantage** function

$$A(s,a) = Q(s,a) - V(s)$$

$$\nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] = \mathbb{E}_{\tau\sim\pi_\theta}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^\pi(s_t,a_t)]$$

- The function of the baseline is to make sure that only the actions that are better than average receive a positive nudge
- Subtract baseline (constant - V(s)) from Q in order to reduce variance

$$V^\pi(s) = \sum_{a\in A} \pi(a|s) * Q^\pi(s,a)$$

Policy gradients - Temporal difference

- Temporal difference error (Can be used to update V)

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \qquad V(S_t) = V(S_t) + \alpha\delta_t$$

- Unbiased estimate of Advantage function
- We can approximate only V with NN, and not worry about Q

$$\nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] = \mathbb{E}_{\tau\sim\pi_\theta}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\delta_t]$$

Comparison

- Value based methods
- **Advantages**: Lower variance
- **Disadvantages**: Doesn't work well for large, continuous spaces

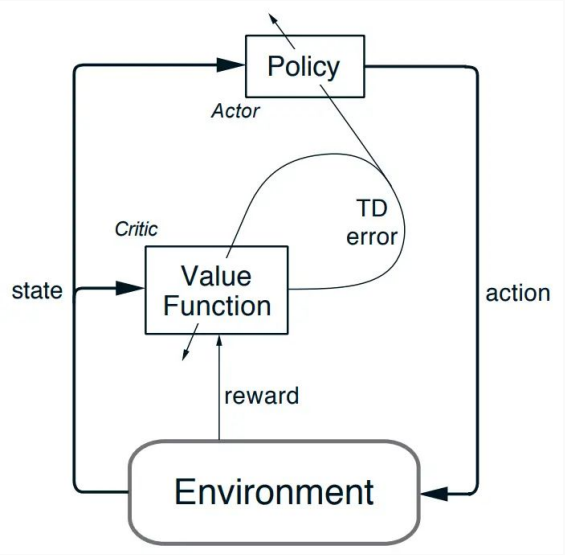
- Policy based methods
- **Advantages**: Works well for continuous spaces, Distributions over actions
- **Disadvantages**: High variance of the policy-gradients, slow convergence and efficiency

Is there a way to combine both of the methods and utilize their advantages?

Actor-Critic Models
The Basics

Introid

- Actor
  - Policy proposing actions given a state
- Critic
  - Evaluates actions taken by the actor based on the given policy



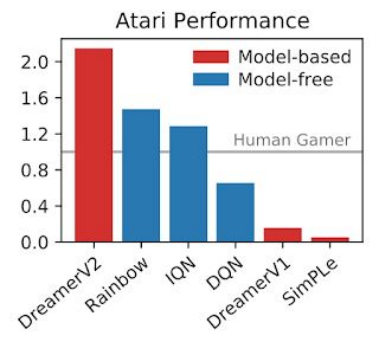
Reinforcement Learning, second edition: An Introduction, p. 258, Fig 11.1

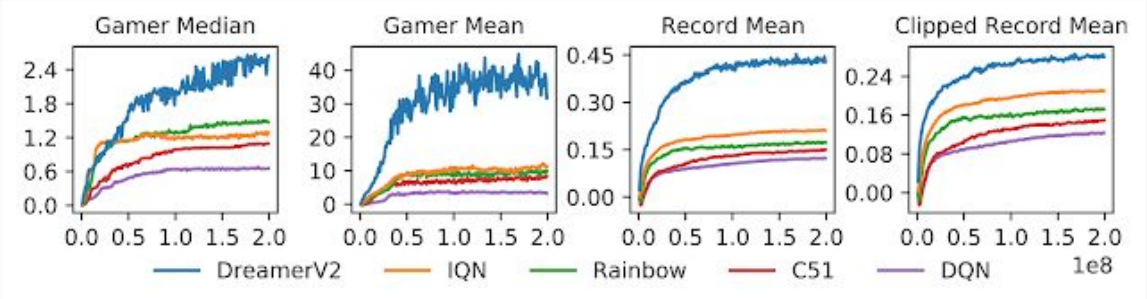
Examples

Solving Atari games with latent world models, e.g. Dreamer model Hafner, D., Lillicrap, T., Ba, J., & Norouzi, M. (2019). Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*.

Smart home energy management, e.g. Yu, L., Xie, W., Xie, D., Zou, Y., Zhang, D., Sun, Z., ... & Jiang, T. (2019). Deep reinforcement learning for smart home energy management. *IEEE Internet of Things Journal*, *7*(4), 2751-2762.

Mobile Robotics, e.g. de Jesus, J. C., Kich, V. A., Kolling, A. H., Grando, R. B., Cuadros, M. A. D. S. L., & Gamarra, D. F. T. (2021). Soft actor-critic for navigation of mobile robots. *Journal of Intelligent & Robotic Systems*, *102*(2), 31.





The Actor: Policy network (π)

- Definition: A function (often a neural network) that maps states to actions
- Parametrized by $\boldsymbol{\theta}$ (e.g., neural network weights)

- Goal: Learn a policy (π) that maximizes expected cumulative reward
- Updates: Based on gradients from the critic

The Critic: Action-Value network (Q)

- Definition: A function (often a neural network) that estimates the expected return given a state-action pair
- Parametrized by **Θ** (e.g., neural network weights)

- Goal: Learn an accurate estimate of action-value function (Q)
- Updates: Based on Temporal Difference (TD) error

Update

The Actor updates the policy distribution in the direction suggested by the Critic

- Critic computes the gradient of the action-value function (Q) w.r.t. the policy parameters (**θ**)
- Actor updates the policy parameters (**Θ**) using the gradients from the critic

- Critic provides a low-variance, data-driven signal for updating the actor

TD learning: Actor-Critic

- Recall the TD error

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

- Using Q-values with TD error we define loss function for actor and critic NNs

$$L(Q', Q, \mu', \mu) = \sum_i (y_i - Q(s_i, a_i)|\Theta^Q))^2,$$

$$y_i = r_t + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$$

- Where we use separate target actor, critic networks to define the target values y_i

$$\mu', Q'$$

Target Networks

- Without target networks we use the same network for prediction of current and next timesteps -> prone to instability and divergence

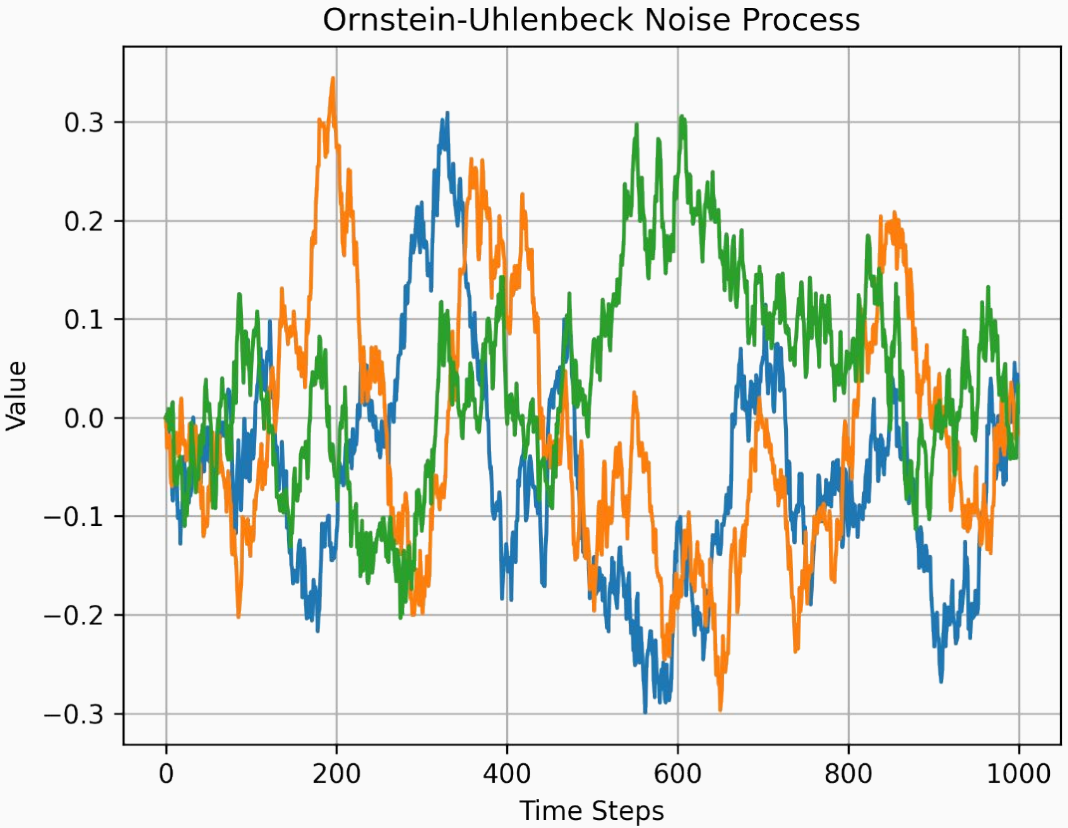
- **DQN:** Use target Q-function network that is switched after a fixed number of steps (Stabilization of learning) with Q-function network

- **DDPG:** Soft update of target network for actor and critic that updates target network after each timestep with a hyperparameter regulating the magnitude of the update.

Balancing Exploration and Exploitation

- Exploration: Trying new actions to discover potentially better policies
- Exploitation: Using the current best policy to maximize immediate reward
- Balancing methods:
  - ϵ-greedy
  - Upper Confidence Bounds (UCB)
  - Noisy Networks: Add noise to the policy network weights to encourage exploration

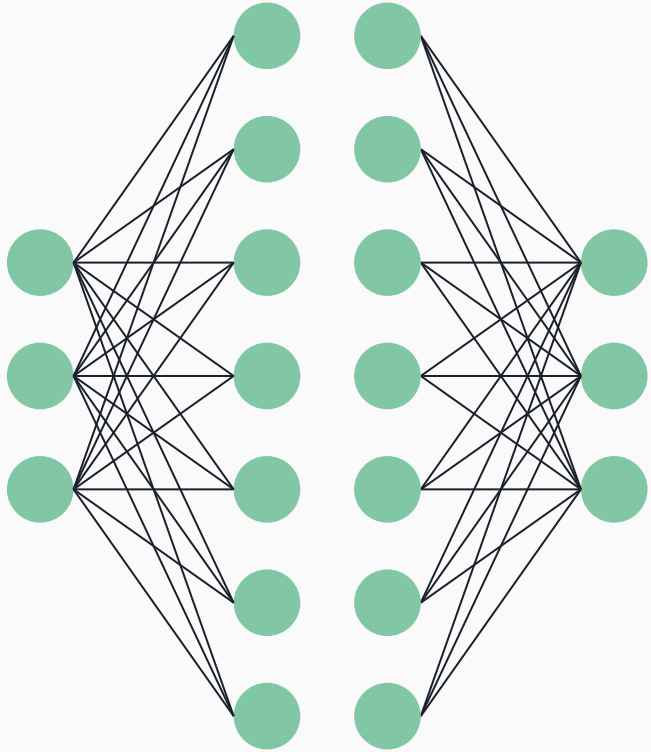
- Ornstein-Uhlenbeck noise process
  - stochastic differential equation models noise



- initial value: 0
- mean reversal rate: 0.15
- mean: 0
- sigma: 0.2
- scaling: 0.1

Actor Network

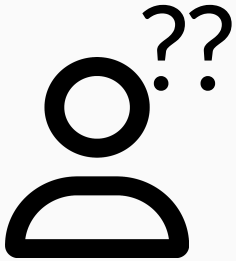
```
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim=256):
        super(Actor, self).__init__()
        self.fc1 = nn.Linear(state_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, action_dim)

    def forward(self, state):
        x = torch.relu(self.fc1(state))
        x = torch.relu(self.fc2(x))
        return torch.tanh(self.fc3(x))
```



State | 2x Hidden | Action

Critic Network

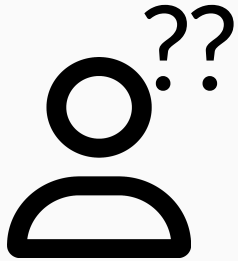
```
class Critic(nn.Module):
    def __init__(self, ??????????           , hidden_dim=256):
        super(Critic, self).__init__()
        self.fc1 = nn.Linear(??????????, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, ??????????)
```



Critic Network

A function that estimates the expected return given a state-action pair

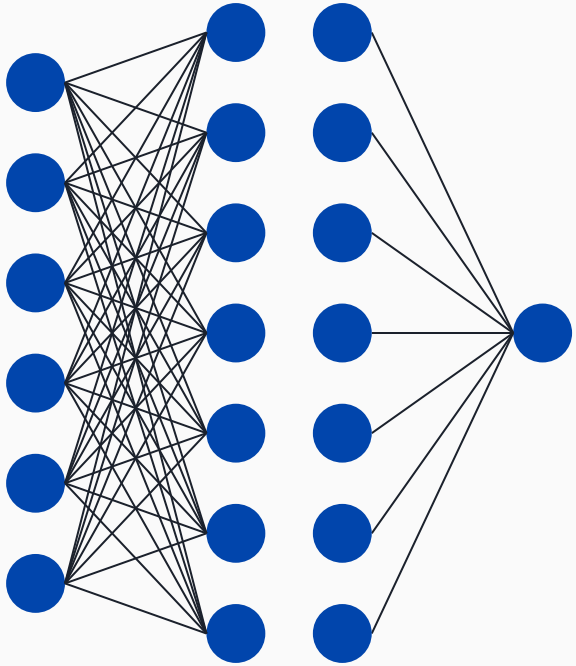
```
class Critic(nn.Module):
    def __init__(self, ??????????           , hidden_dim=256):
        super(Critic, self).__init__()
        self.fc1 = nn.Linear(??????????, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, ??????????)
```



Critic Network

A function hat estimates the expected return given a state-action pair

```
class Critic(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim=256):
        super(Critic, self).__init__()
        self.fc1 = nn.Linear(state_dim + action_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, 1)

    def forward(self, state, action):
        x = torch.relu(self.fc1(torch.cat([state, action], dim=1)))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)
```



State + Action | 2x Hidden | E. Return

Deep Deterministic Policy Gradients

bases on Lillicrap, Hunt et al. (2019), “Continuous control with deep reinforcement learning”, https://arxiv.org/abs/1509.02971

What is a replay buffer?

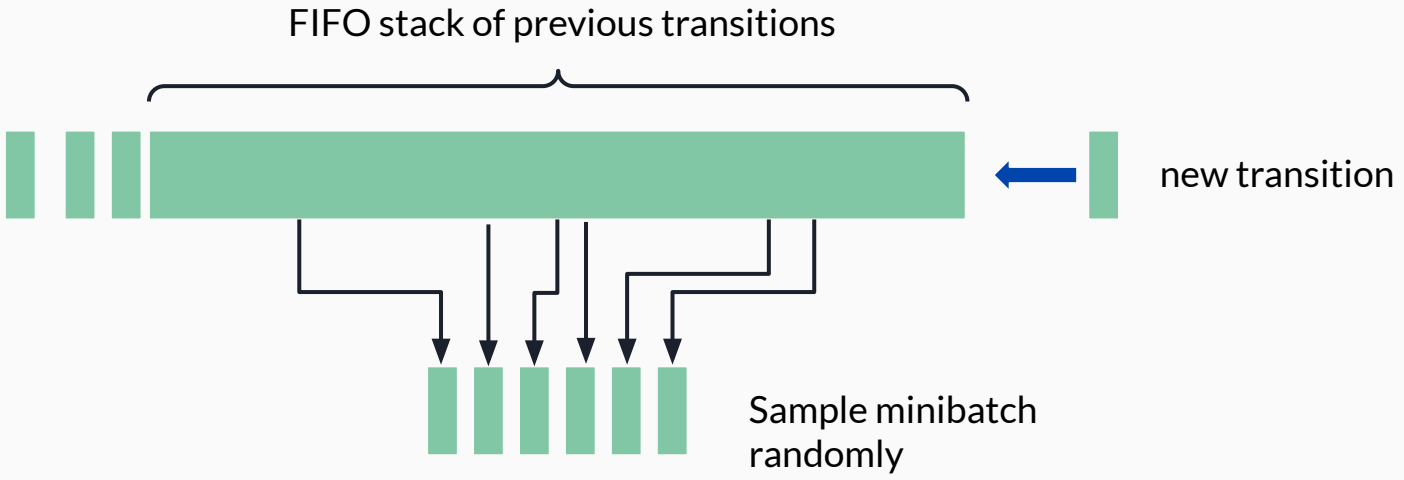
- Elisabeth

What is the advantage?

- Marta

Replay Buffer

- Store previous transitions
- Minimize correlations between samples



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:

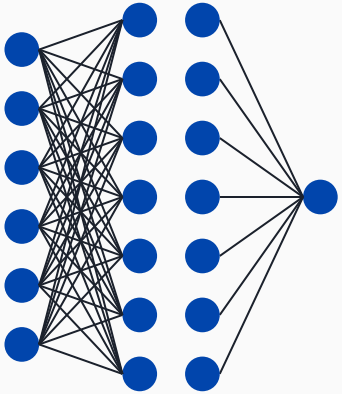
$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

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

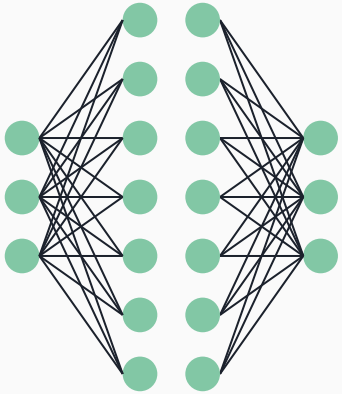
end for
end for

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

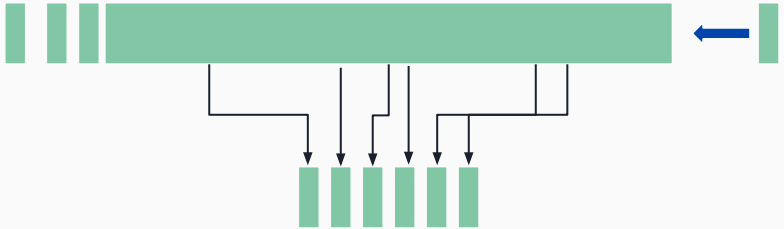
2x Q:



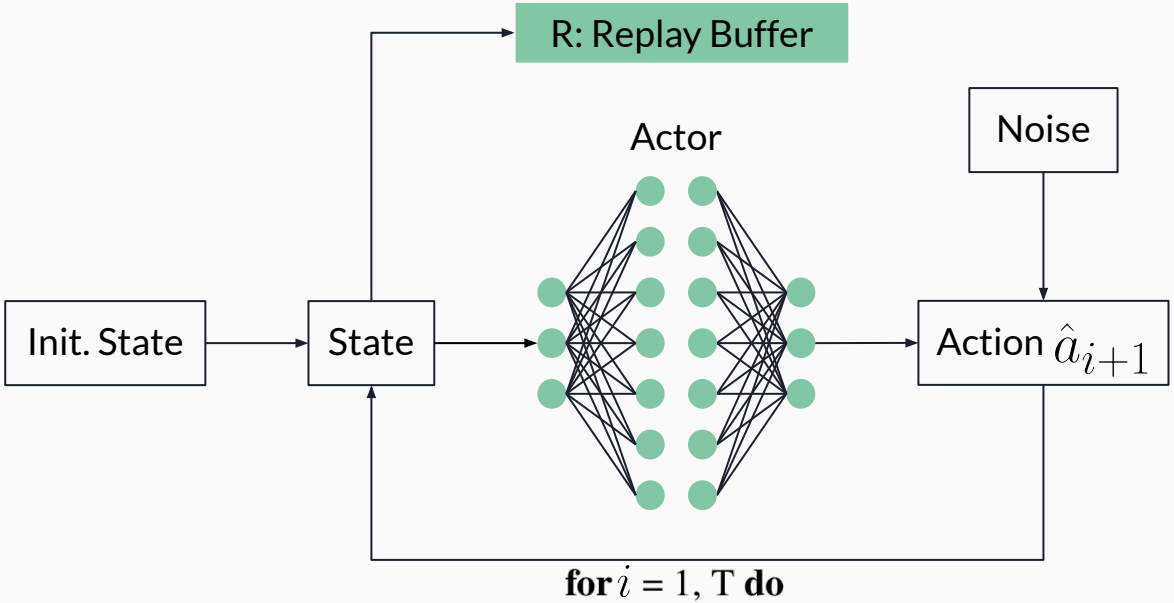
2x μ:



R:

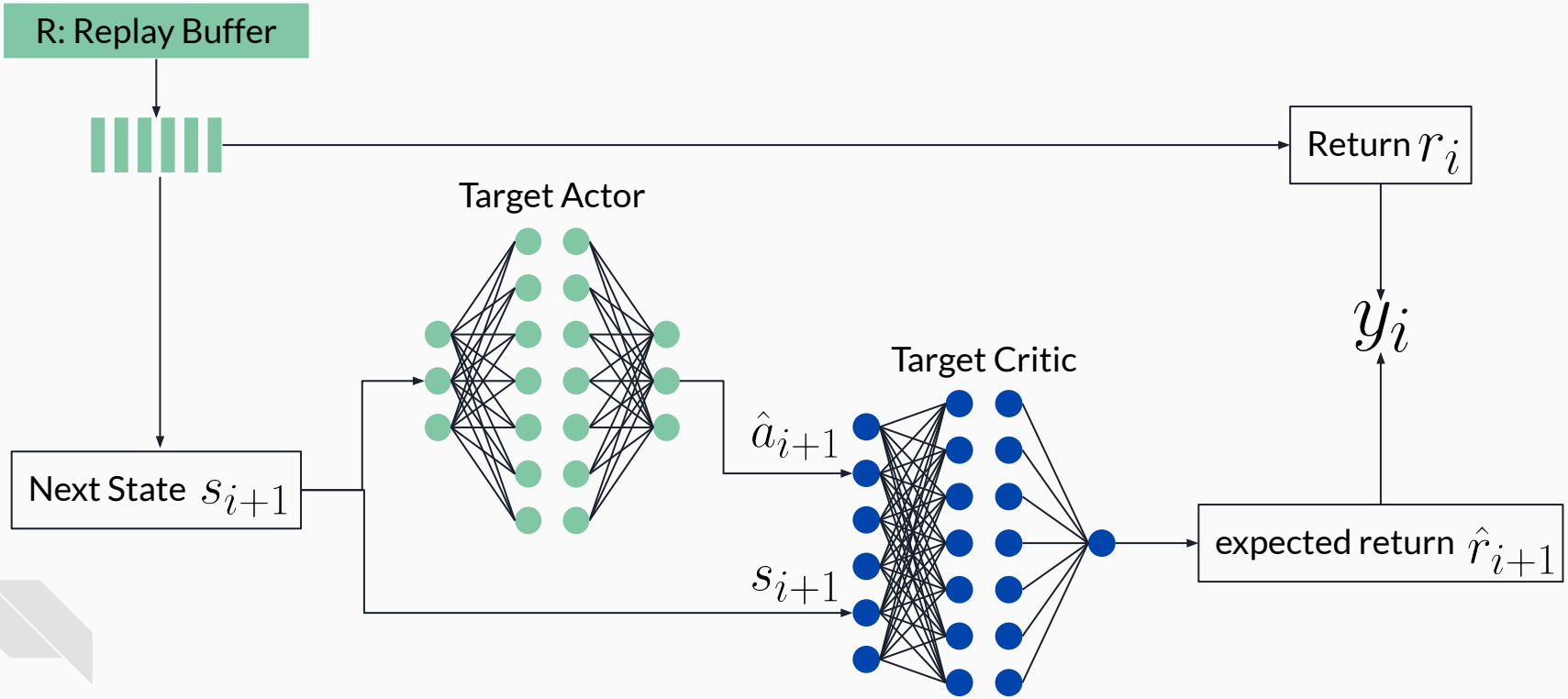


for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R

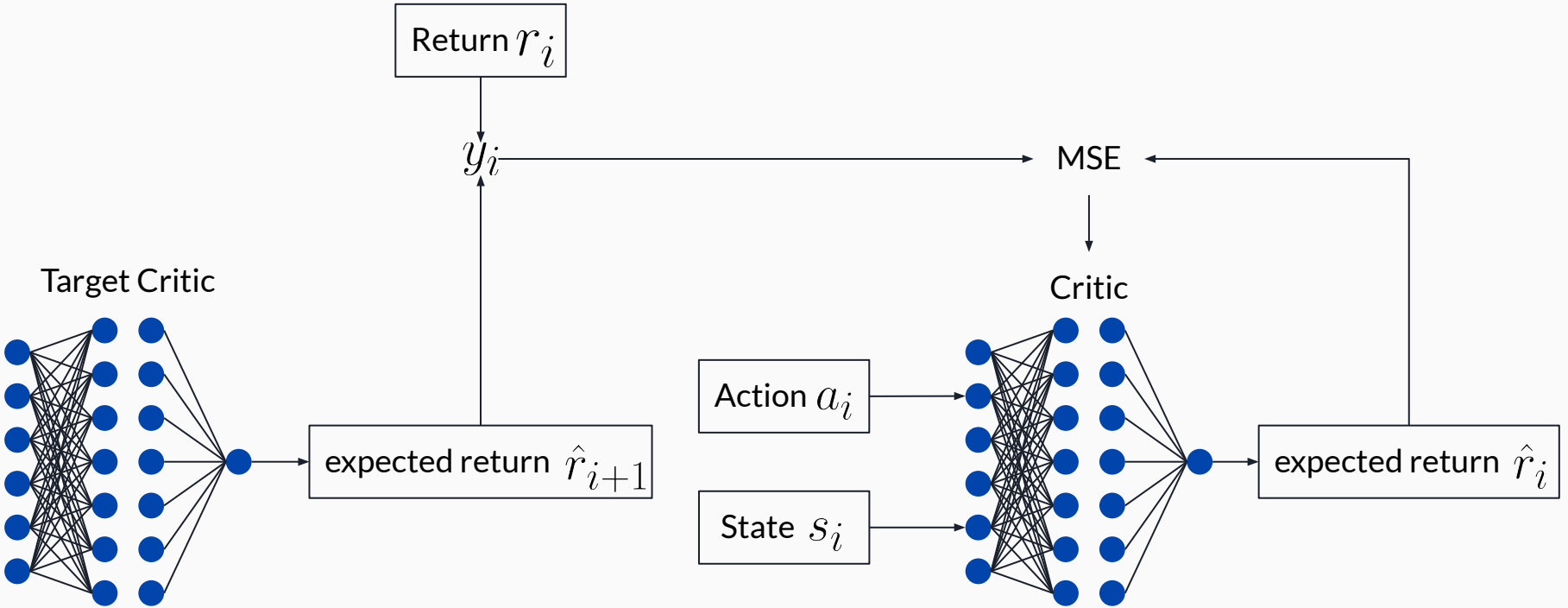


Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R

Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$

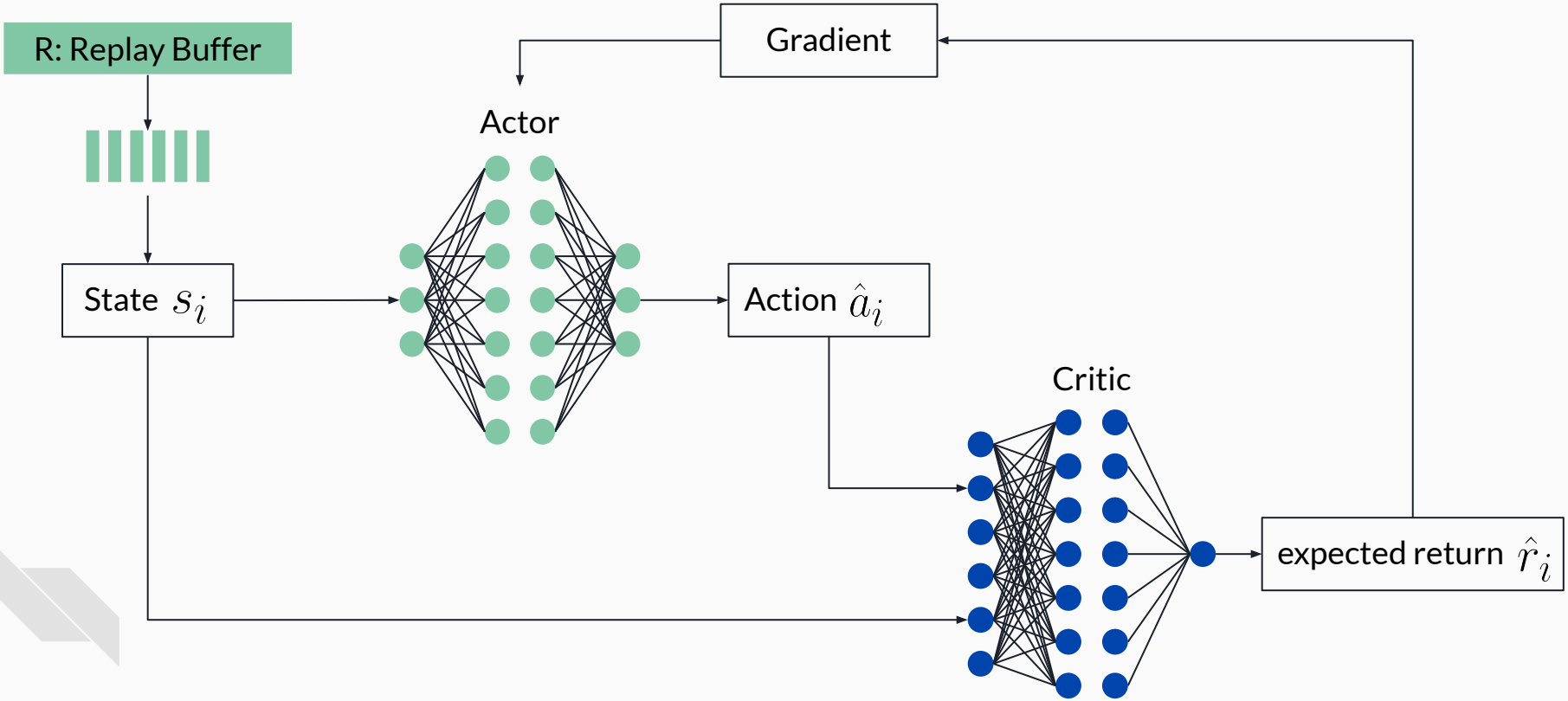


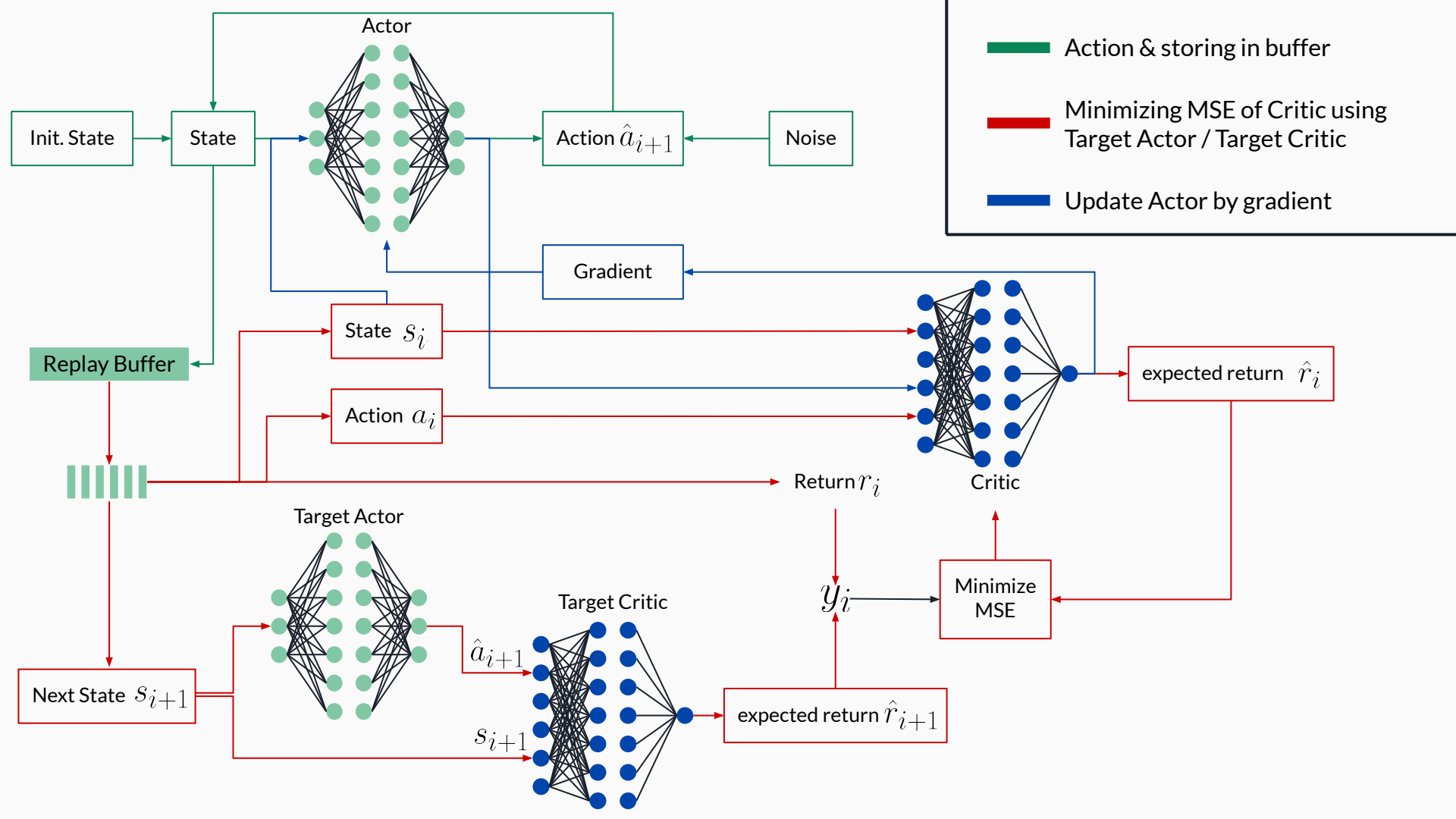
Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$



Update the actor policy using the sampled policy gradient:

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




Action & storing in buffer
Minimizing MSE of Critic using Target Actor / Target Critic
Update Actor by gradient
Actor
Init. State
State
Action $\hat{a}_{i+1}$
Noise
Gradient
Replay Buffer
State s_i
Action a_i
expected return $\hat{r}_i$
Return r_i
Critic
Target Actor
Target Critic
$\hat{y}_i$
Minimize MSE
Next State s_{i+1}
$\hat{a}_{i+1}$
s_{i+1}
expected return $\hat{r}_{i+1}$

How do you update the target networks?

- Elisabeth

Soft-Update

- Since Critic Q is updated by Target Critic Q', setting Q' ← Q is prone to divergence.

Update the target networks:

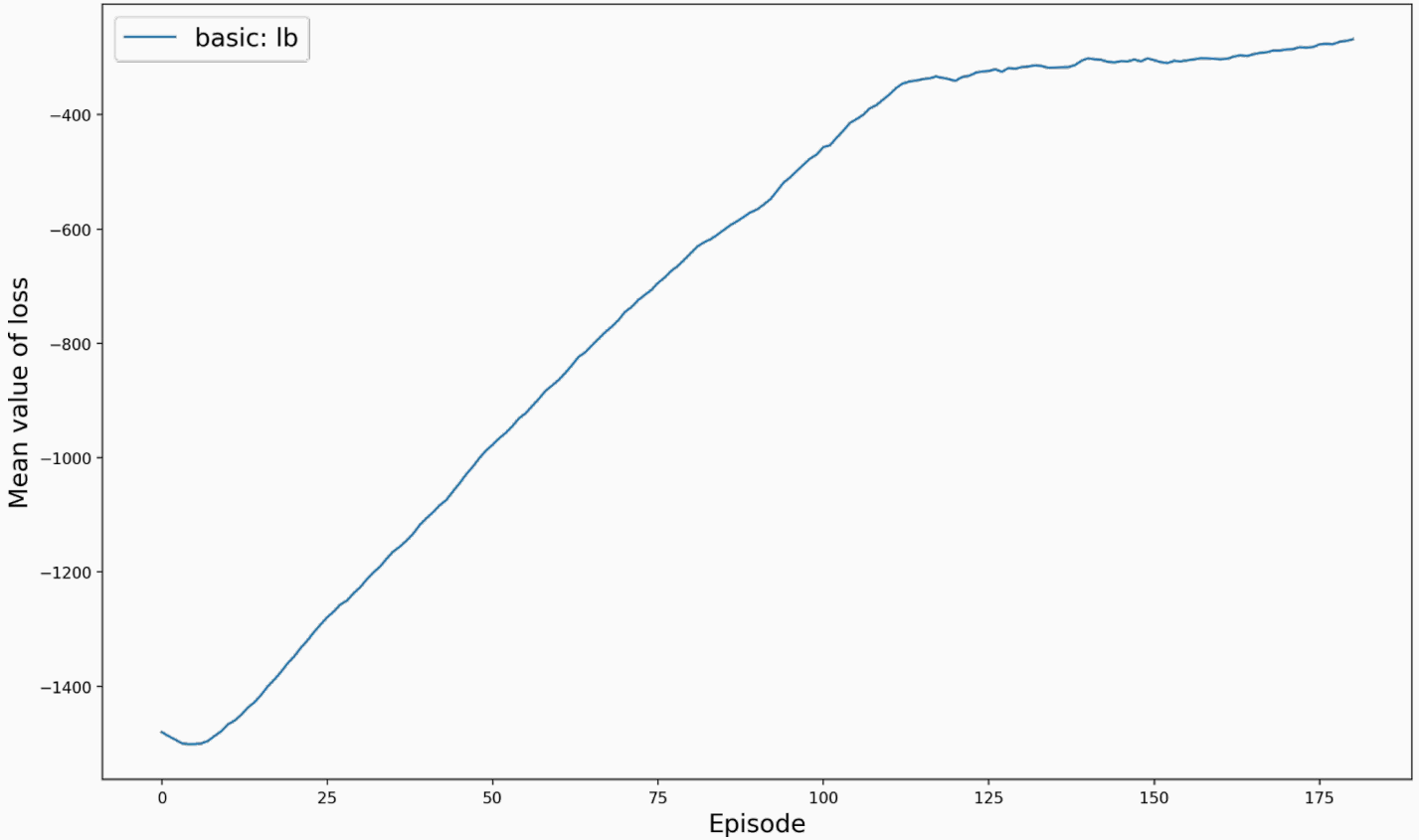
$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

- Slowly track the learned networks
- Tau to regulate learning

Performance

State Dimension: 3
Action Dimension: 1

State Dimension: 24
Action Dimension: 4



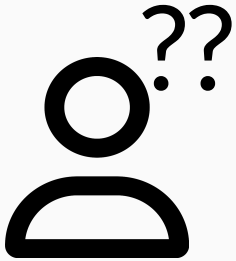
Average loss of basic DDPG implementation (30 runs, rolling average 10) on Pendulum V1

Possible Improvements

Stratification of Replay Buffer

- Differentiate between “good” and “bad” episodes
- Sample more “good” episodes

→ Two assumptions:

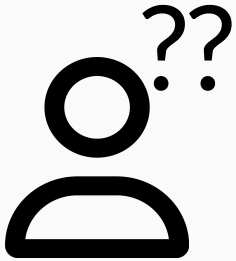


Stratification of Replay Buffer

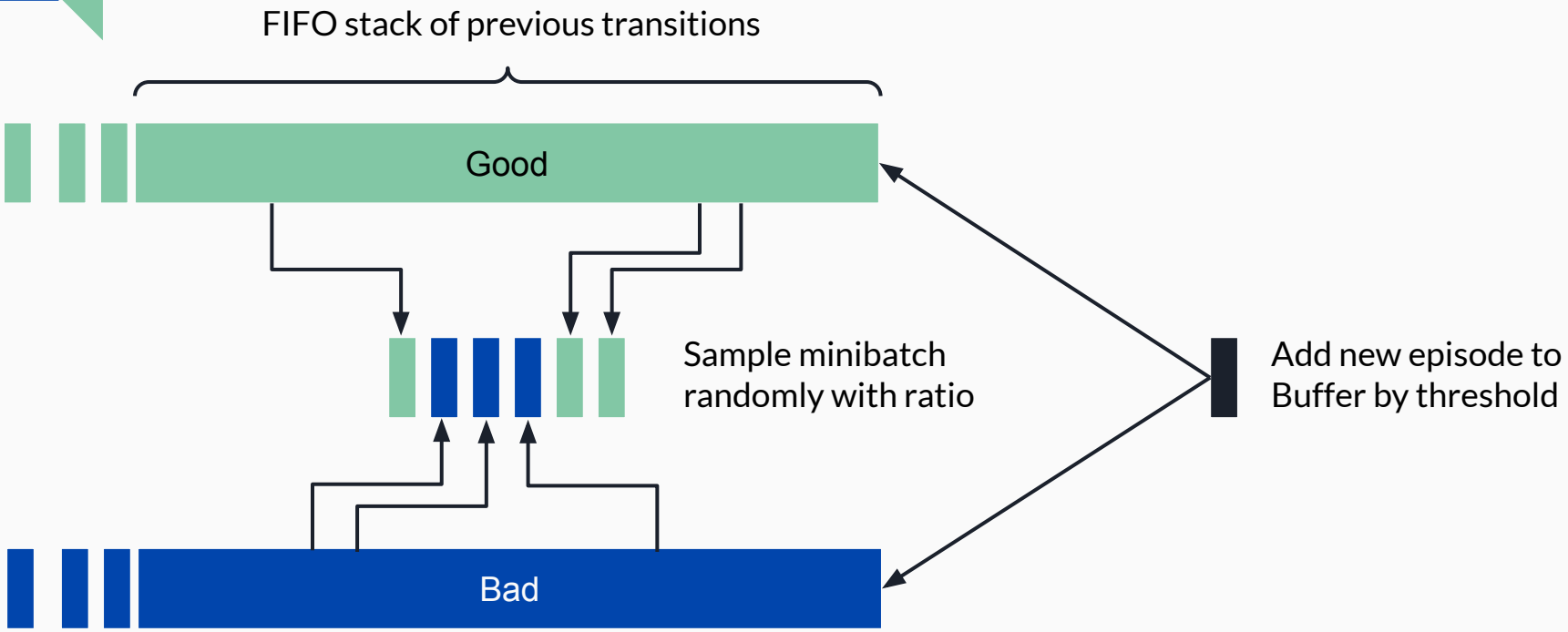
- Differentiate between “good” and “bad” episodes
- Sample more “good” episodes

→ Two assumptions:

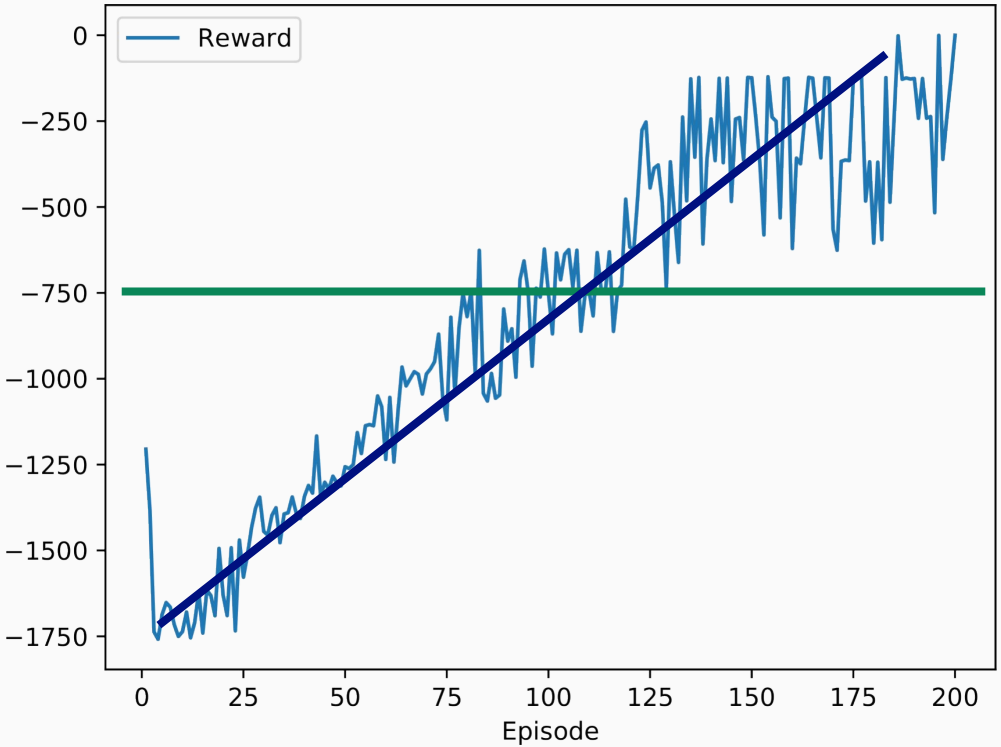
- Learning from “good” episodes is better
- No strong local maxima



Stratification of Replay Buffer



Threshold

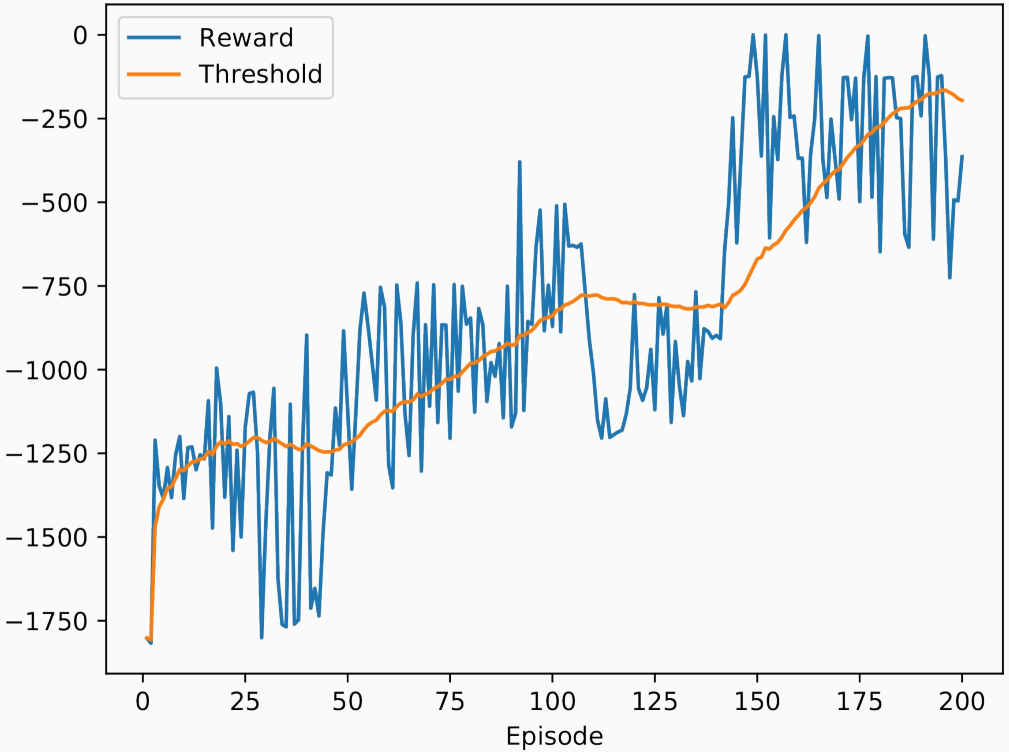


What is a good transition?

Constant?
Linear?

How to determine?

Dynamic Threshold



$$\mu_{\text{roll}} = \frac{1}{w} \sum_{j=i-w}^{i} l_j$$

$$\sigma_{\text{roll}} = \sqrt{\frac{1}{w} \sum_{j=i-w}^{i} (l_j - \mu_{\text{roll}})^2}$$

$$t_i = \mu_{\text{roll}} + \lambda \sigma_{\text{roll}}$$

Softmax of Replay Buffer

- Weighted sampling of trajectories based on absolute deviance from mean of rewards
- Sampling weights estimated through softmax function
- Temperature parameter regulates magnitude of sampling bias

$$p_i = softmax(|r_i - \mu_R|)$$

$$softmax(z_i) = \frac{e^{z_i/\beta}}{\sum_{j=1}^{n} e^{z_j/\beta}}$$

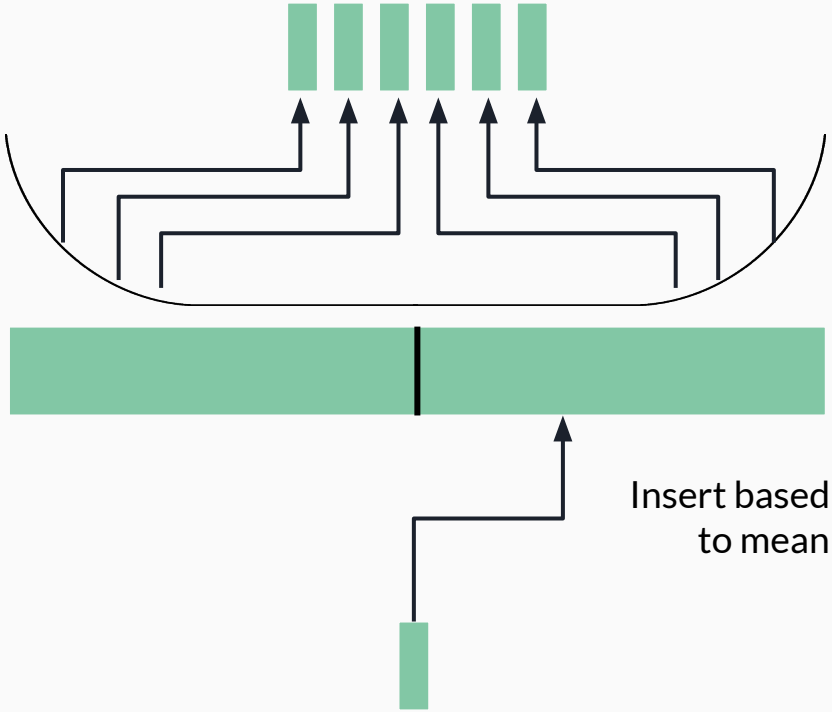
→ One assumption:

- “Extreme” episodes have more information, more valuable gradients
- Beta = 1 for experiments

Why is the deviance from the mean a good measure of importance?

- Bendik

Softmax of Replay Buffer



Sampling based on distribution
regulated by beta

Insert based on distance
to mean reward

Variable Buffer Size

- Decrease Buffer Size
- Sample from more recent (and better) episodes

→ Two assumptions:

- Learning from “newer / better” episodes is better
- Buffer is large enough so that autocorrelation remains low

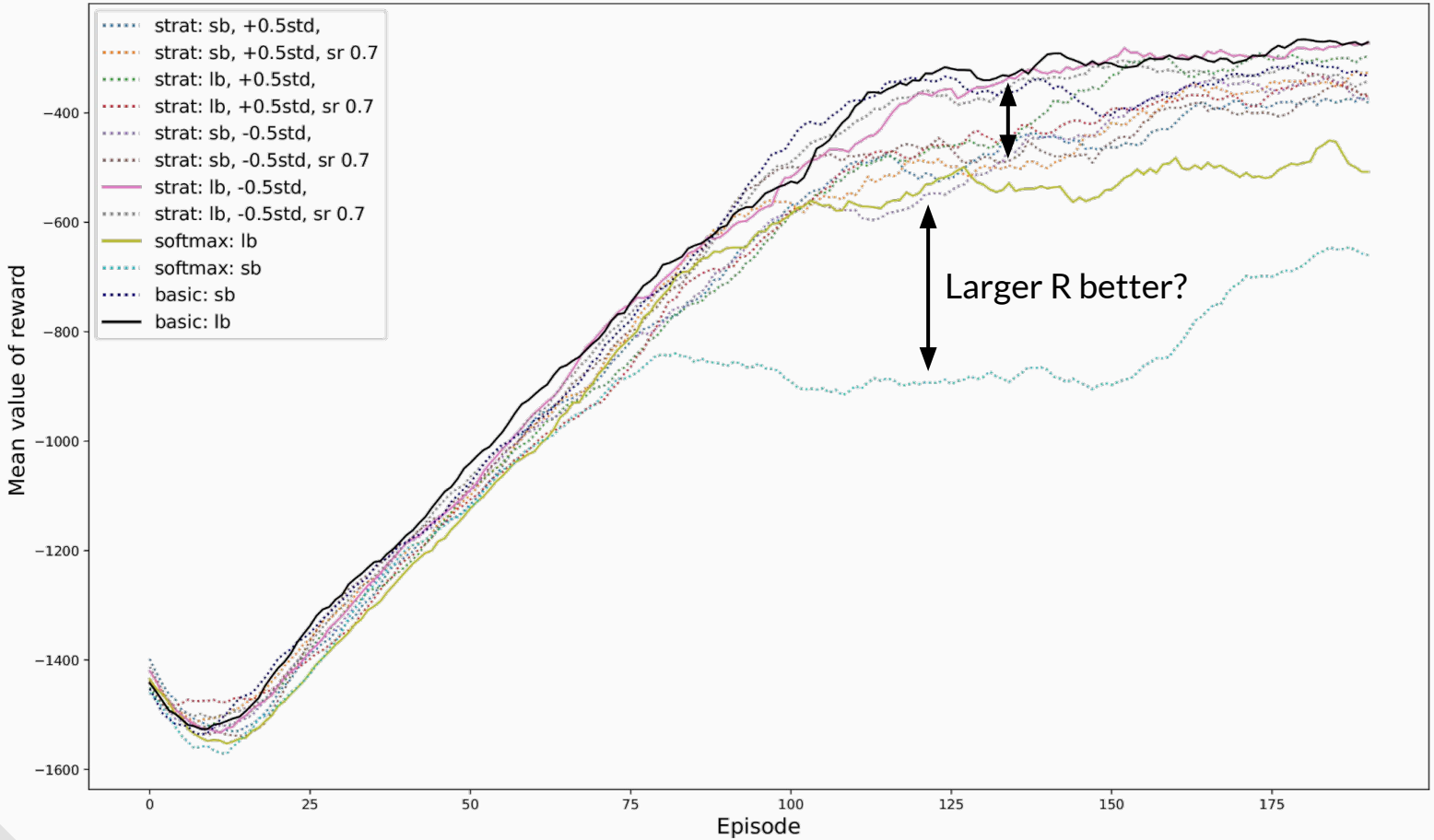
Results and Discussion

Results

- Pendulum V1
  - 200 Episodes
  - Stratification:
    {small 10^4, large 10^6} buffer ×
    {low -0.5, high +0.5} threshold ×
    {0.5/0.5, 0.3/0.7} sampling rate
  - Softmax:
    {small 10^4, large 10^6} buffer
  - Basic:
    {small 10^4, large 10^6} buffer
- BipedalWalker V3
  - 1000 Episodes
  - Stratification:
    {large 10^6} buffer ×
    {low -0.5, high +0.5} threshold ×
    {0.5/0.5} sampling rate
  - Softmax:
    {large 10^6} buffer
  - Basic:
    {large 10^6} buffer

General Results

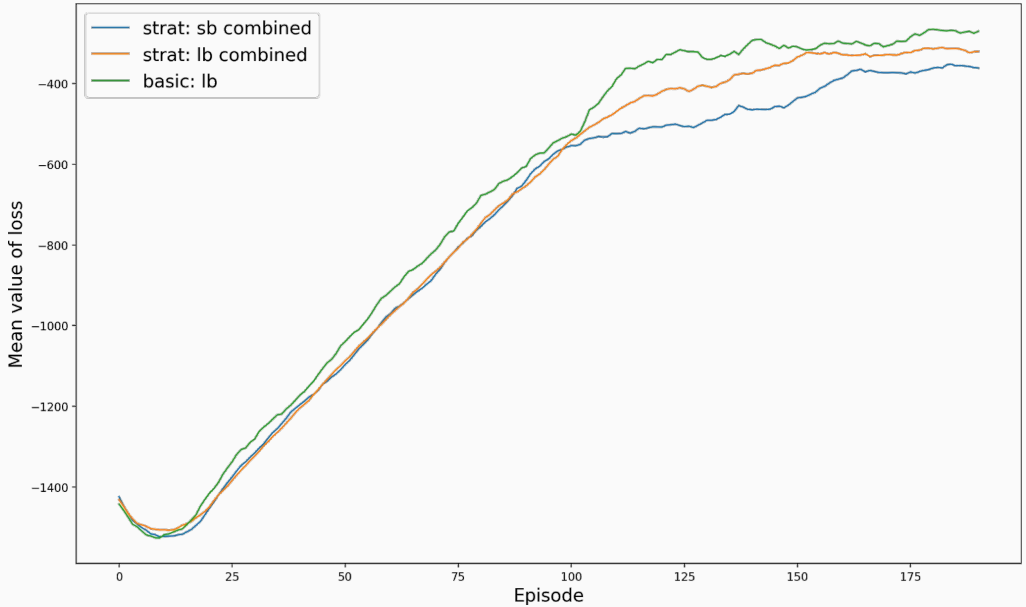
- Softmax is ~ 2x slower than Stratification / Basic.
- BipedalWalker V3 is a hard environment with strong local maxima.

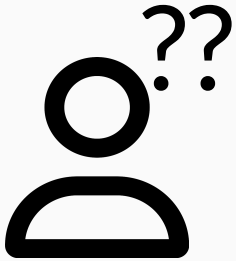


Average loss (30 runs, rolling average 10) on Pendulum V1, all 12 configurations.

Larger Replay Buffer better?

More transitions → higher chance for …

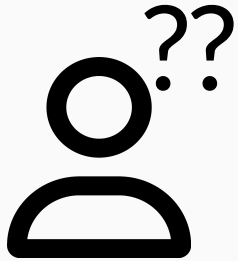


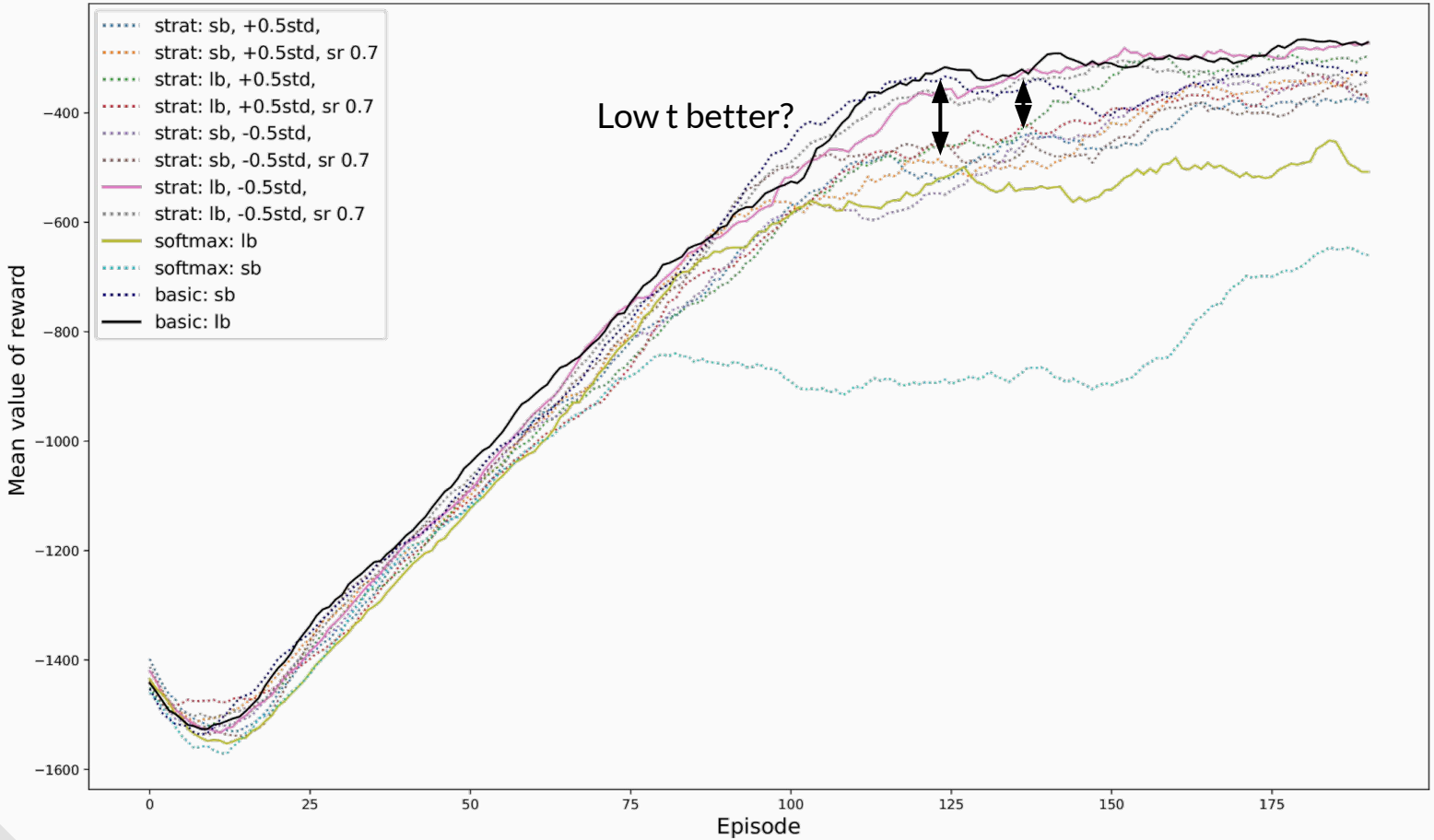


Larger Replay Buffer better?

More transitions → higher chance

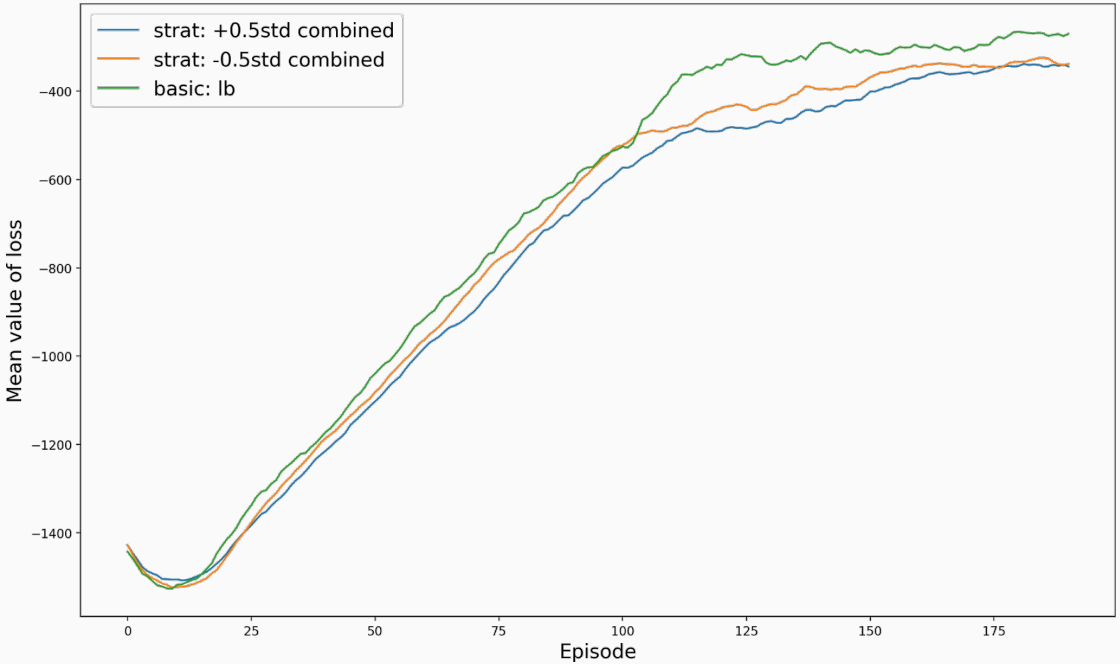
- for uncorrelated samples
- for larger variance
- for identical distributed minibatches
- getting out of local maxima

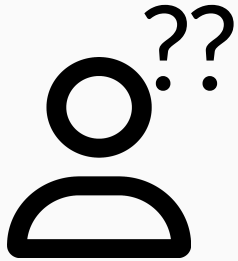




Average loss (30 runs, rolling average 10) on Pendulum V1, all 12 configurations.

Low threshold better?



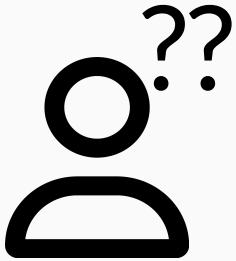


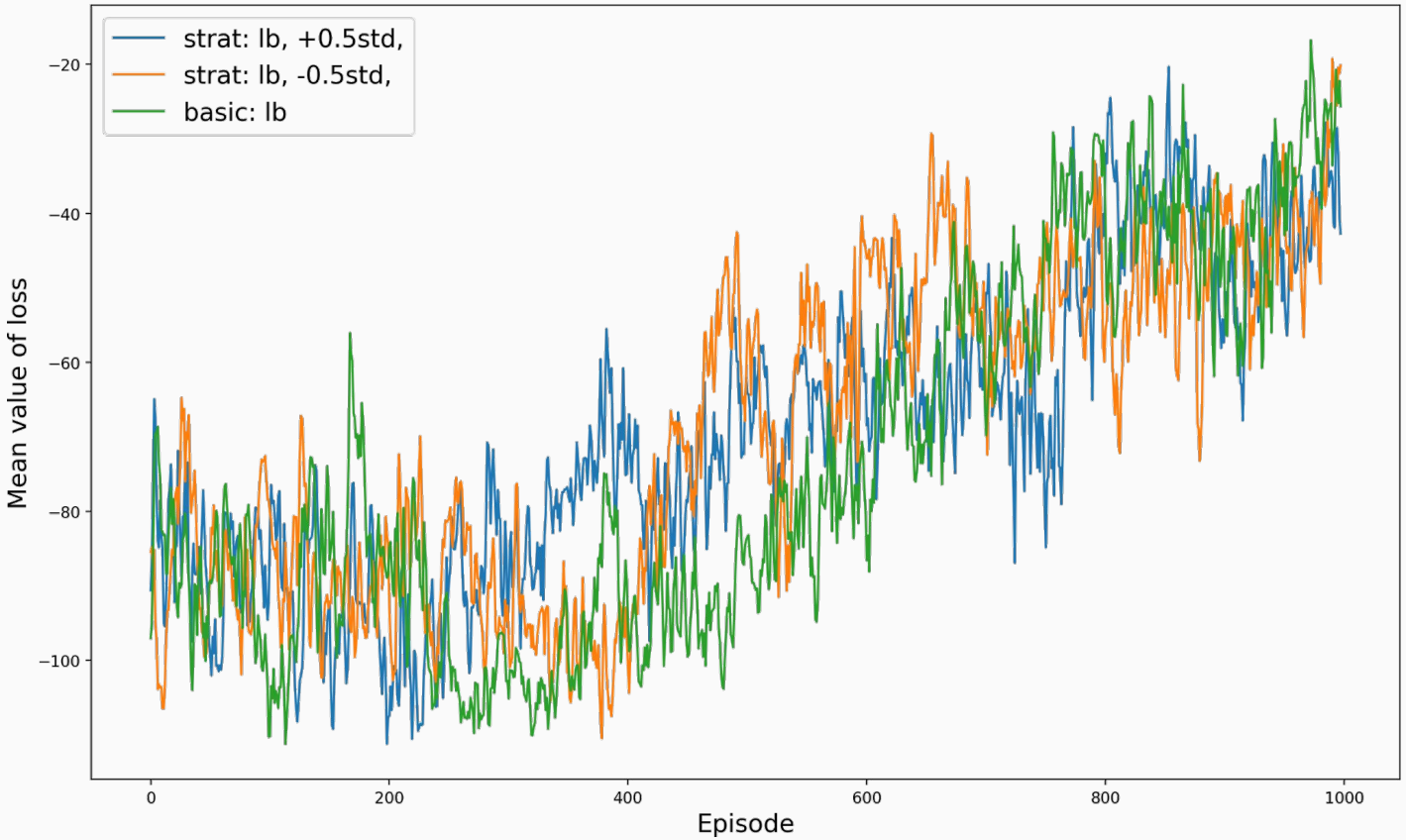
Low threshold better?

- Slight improvement over high threshold.
- Prioritize better transitions.
- Optimization of threshold hyperparameters could lead to improved performance

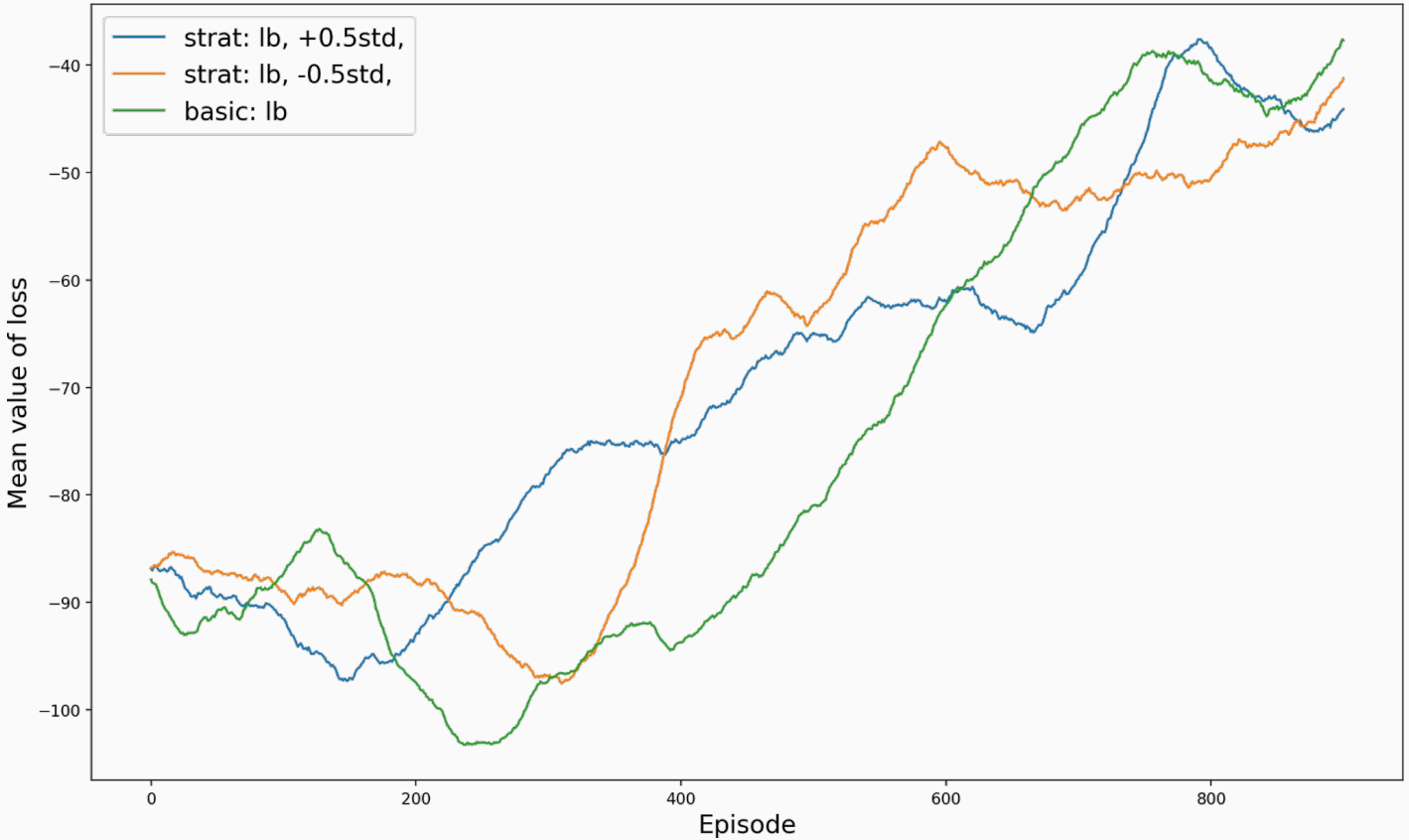
- “good” buffer gets updated more often because of low threshold
  -> more recent episodes get mixed with less recent episodes from “bad” buffer

- sampling of batches consisting of good correlated trajectories mixed with less recent ones from “bad” buffer
  -> reduced autocorrelation in sampled minibatches, while still using successful trajectories

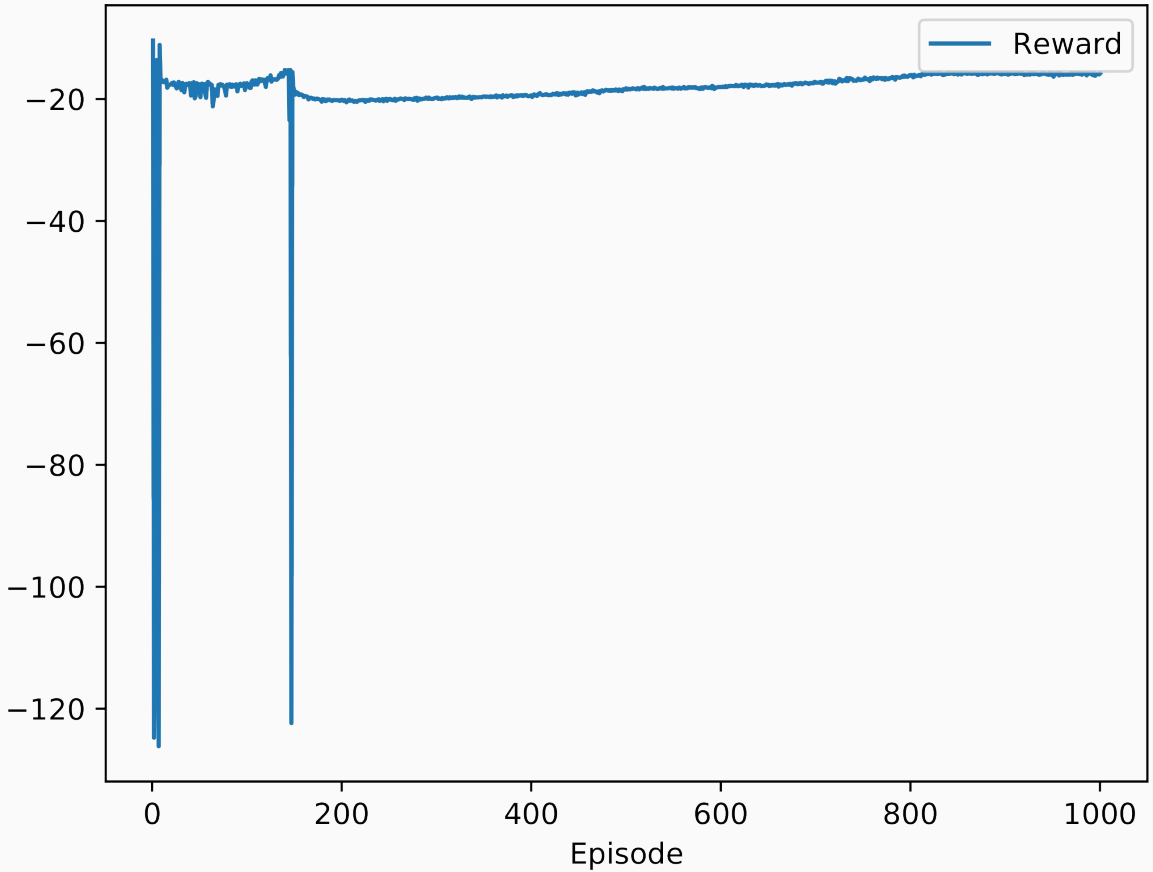




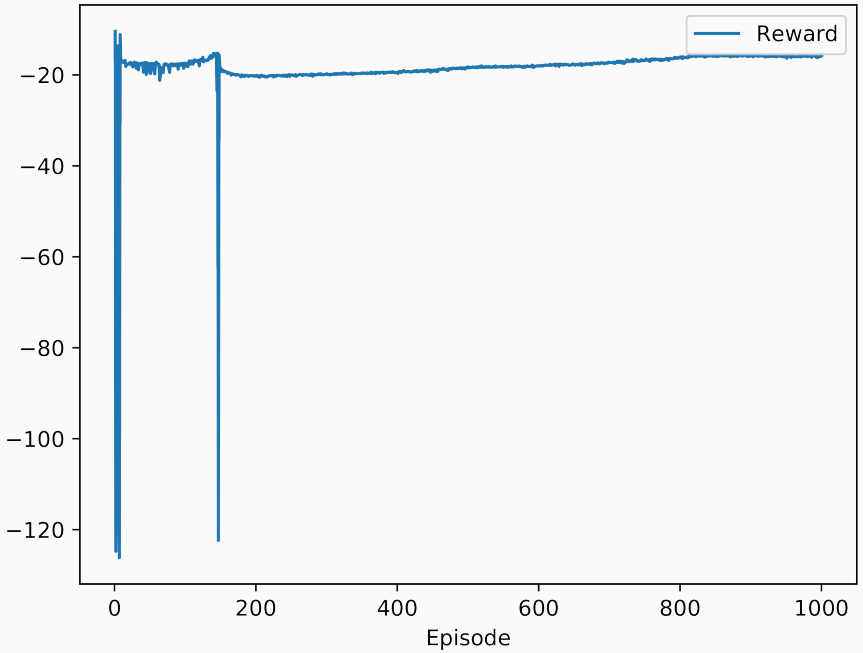
Average loss (10 runs, rolling average 3) on BipedalWalker V3

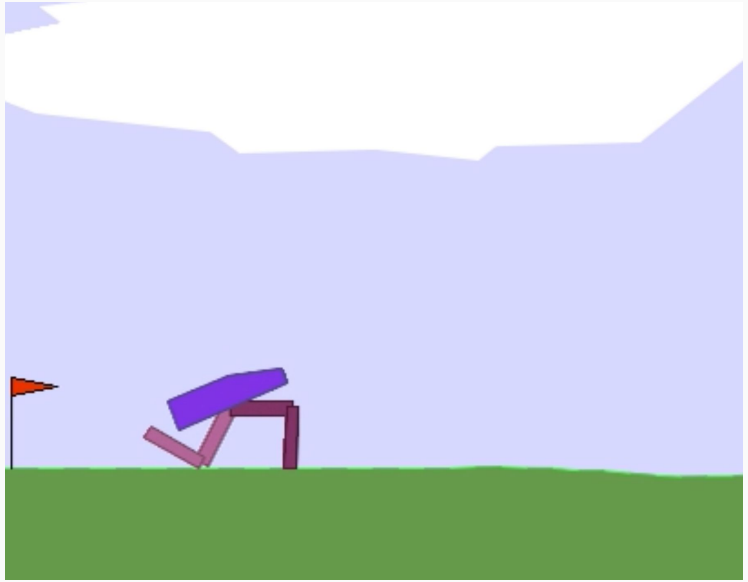


Average loss (10 runs, rolling average 100) on BipedalWalker V3

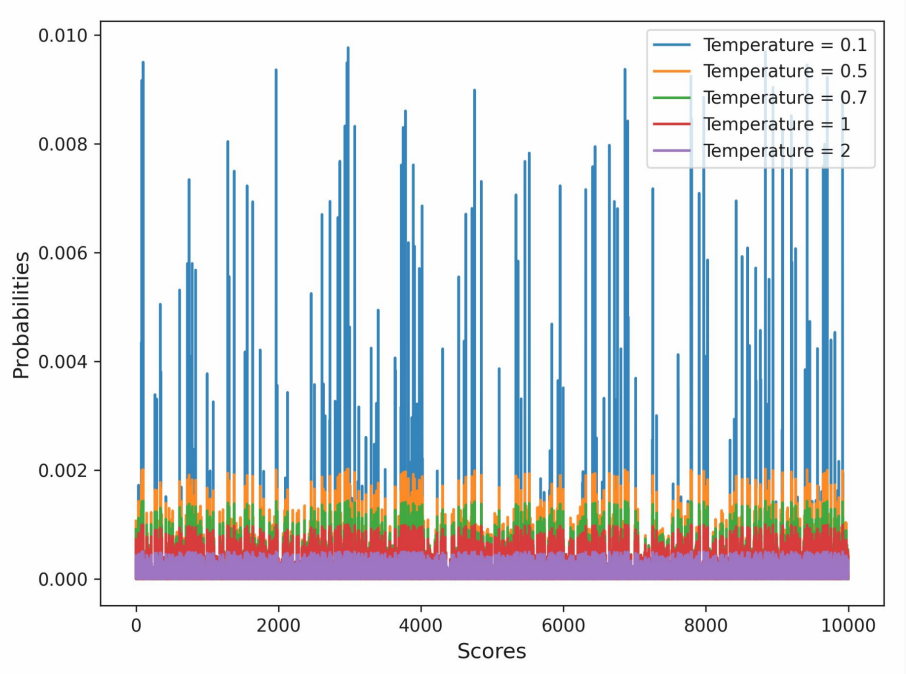


Hyperparameter tuning is needed!
Strong convergence to local maximum. Softmax beta=1 BipedalWalker V3





Hyperparameter tuning is needed!
Strong convergence to local maximum. Softmax beta=1 BipedalWalker V3



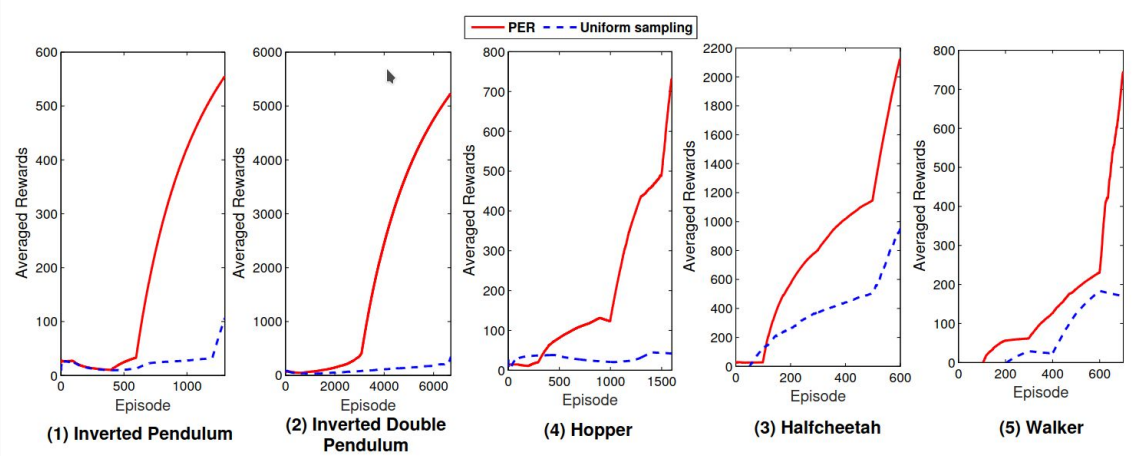
- Effects of Softmax parameter on randomly sampled values
- Hyperparameter search with evolutionary optimization to find optimal exploration exploitation trade-off could be used to increase success of method
- 1. Define prior distribution of beta (e.g. log-gaussian)
- 2. Use evolutionary optimization to get optimized parameter distribution (e.g. Tournament Sampling, parallelization necessary)

Conclusion

- DDPG is a robust policy gradient method.
- Stratification and Softmax need hyperparameter tuning.
- Larger buffer sizes are better.
- Stability of gradients crucial for success.

Existing Improvements

- Improving DDPG via Prioritized Experience Replay
- Similar idea: Select important samples from buffer for training
- Different: Update loss function based on importance sampling



https://ieeexplore.ieee.org/document/8122622

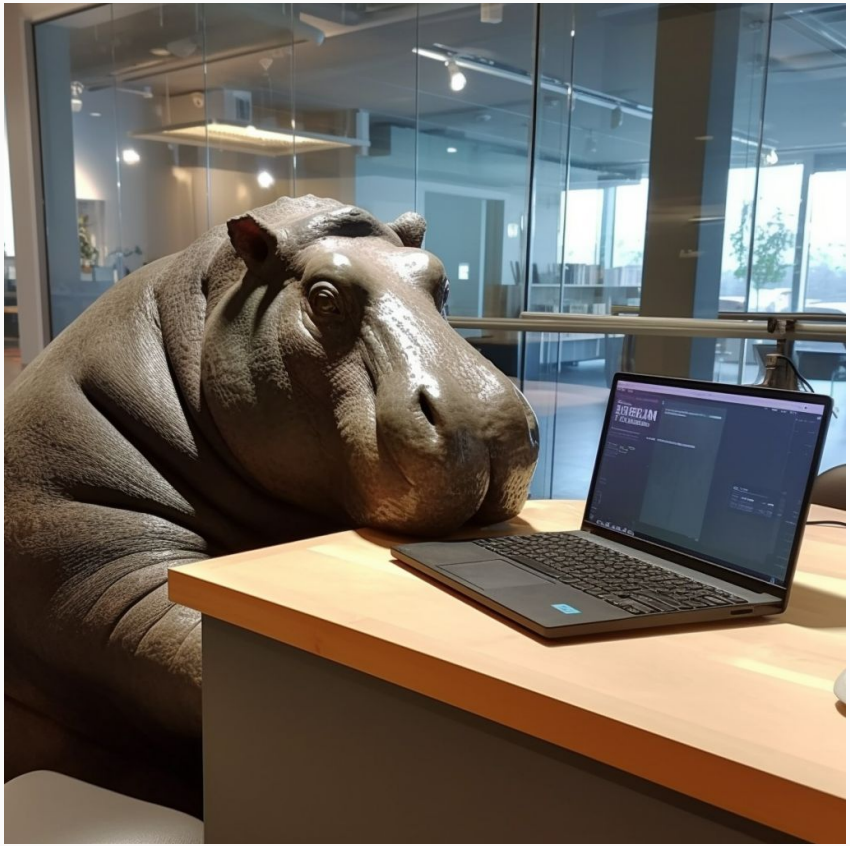
Existing Improvements

- A3C: Asynchronous Advantage Actor-Critic
  - Multiple actors
  - Critics learn the value function, get synced from time to time
  - Parallel training
- A2C
- D4PG: Distributed Distributional DDPG
  - Use softmax to prioritize the experiences
  - Multiple steps to calculate TD error
  - Use multiple actors (common buffer)

Further Directions

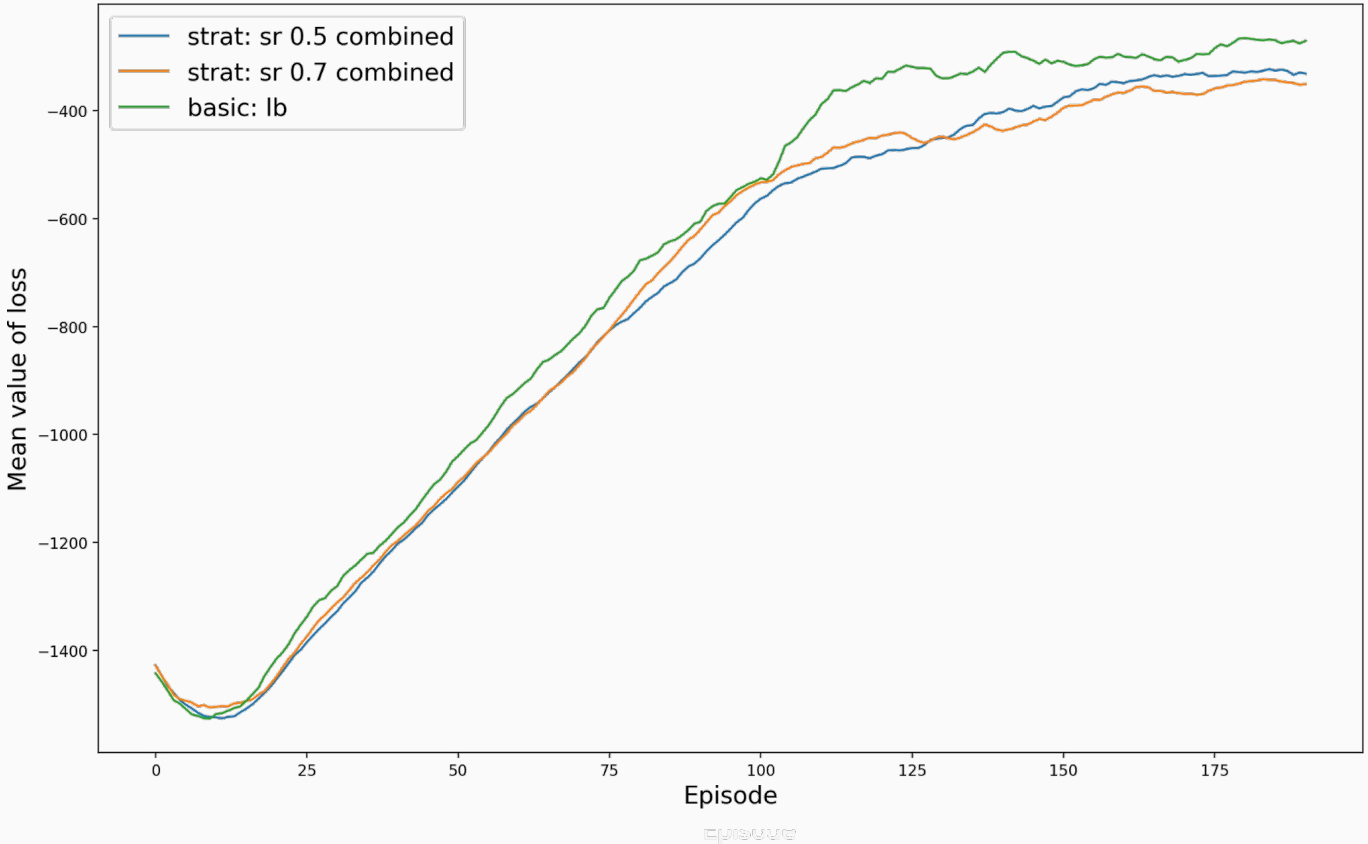
- Play with hyperparameters
- Experiment with complex environments
- Effect of neural network architectures

- Using bayesian methods (more heavy, but more robust),
  - circumvent unstable gradient and local optima problem (e.g. because of prioritization of extreme trajectories in stratified and softmax buffers), e.g. bayesian neural nets or gaussian process for actor and critic networks
  - keep previous information in the loop through bayesian inference of parameter updates



Midjourney 5.1: A hippo sitting in front of a laptop programming at IFI in Oslo

Supplements



Sampling rate 0.5 / 0.7 combined, Pendulum V1, mov. average 10

ibminode01:~> lscpu

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
Address sizes:	48 bits physical, 48 bits virtual
CPU(s):	64
On-line CPU(s) list:	0-63
Thread(s) per core:	2
Core(s) per socket:	8
Socket(s):	4
NUMA node(s):	8
Vendor ID:	AuthenticAMD
CPU family:	21
Model:	2
Model name:	AMD Opteron(tm) Processor 6376
Stepping:	0
CPU MHz:	2581.071
BogoMIPS:	4600.28
Virtualization:	AMD-V
L1d cache:	512 KiB
L1i cache:	2 MiB
L2 cache:	64 MiB
L3 cache:	48 MiB

System Info

active environment : deepq
conda version : 4.12.0
conda-build version : not installed
python version : 3.9.12.final.0
virtual packages : __linux=5.4.0=0
__glibc=2.31=0
__unix=0=0
__archspec=1=x86_64

packages in environment at /nfs/wsi/mm/projects/kreuerju/miniconda3/envs/deepq:
#
Name Version Build Channel
gymnasium 0.27.1 py38h6f1a3b6_0 conda-forge
imageio 2.27.0 pyh24c5eb1_0 conda-forge
imageio-ffmpeg 0.4.8 pypi_0 pypi
matplotlib 3.2.2 1 conda-forge
natsort 8.3.1 pyhd8ed1ab_0 conda-forge
numpy 1.24.2 py38h10c12cc_0 conda-forge
openjpeg 2.5.0 hfec8fc6_2 conda-forge
pandas 1.5.3 py38hdc8b05c_1 conda-forge
pillow 9.4.0 py38h961100d_2 conda-forge
pip 23.0.1 pyhd8ed1ab_0 conda-forge
pygame 2.1.3.dev8 pypi_0 pypi
pyparsing 3.0.9 pyhd8ed1ab_0 conda-forge
python 3.8.16 he550d4f_1_cpython conda-forge
pytorch 1.7.1 cpu_py38h36eccb8_2 conda-forge

Environment Info

Proof of the Policy Gradient Theorem (episodic case)

With just elementary calculus and re-arranging terms we can prove the policy gradient theorem from first principles. To keep the notation simple, we leave it implicit in all cases that π is a function of $\boldsymbol{\theta}$, and all gradients are also implicitly with respect to $\boldsymbol{\theta}$. First note that the gradient of the state-value function can be written in terms of the action-value function as

$$
\begin{aligned}
\nabla v_\pi(s) &= \nabla \left[\sum_a \pi(a|s) q_\pi(s,a) \right], \quad \text{for all } s \in \mathcal{S} && \text{(Exercise 3.15)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla q_\pi(s,a) \Big] && \text{(product rule)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla \sum_{s',r} p(s',r|s,a) \big(r + v_\pi(s')\big) \Big] \\
& && \text{(Exercise 3.16 and Equation 3.2)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a) \nabla v_\pi(s') \Big] && \text{(Eq. 3.4)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a) && \text{(unrolling)} \\
&\qquad \sum_{a'} \big[\nabla \pi(a'|s') q_\pi(s',a') + \pi(a'|s') \sum_{s''} p(s''|s',a') \nabla v_\pi(s'') \big] \Big] \\
&= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \Pr(s \to x, k, \pi) \sum_a \nabla \pi(a|x) q_\pi(x,a),
\end{aligned}
$$

after repeated unrolling, where $\Pr(s \to x, k, \pi)$ is the probability of transitioning from state s to state x in k steps under policy π. It is then immediate that

$$
\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= \nabla v_\pi(s_0) \\
&= \sum_s \left(\sum_{k=0}^{\infty} \Pr(s_0 \to s, k, \pi) \right) \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&= \sum_s \eta(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) && \text{(box page 163)} \\
&= \left(\sum_s \eta(s) \right) \sum_s \frac{\eta(s)}{\sum_s \eta(s)} \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&\propto \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a). \qquad \text{Q.E.D.} && \text{(Eq. 9.3)}
\end{aligned}
$$