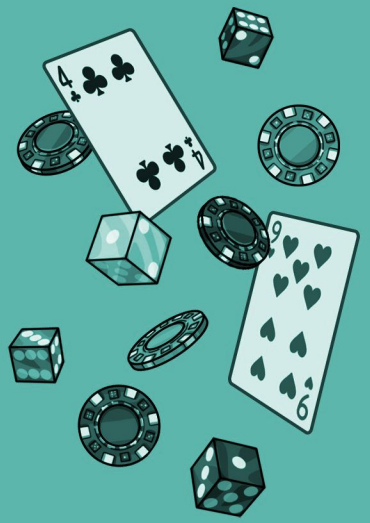
Student Presentations

On RL Algorithms

Assignment 2 Part A

CSE 574

1	MCTS	**6**	HER (Hindsight Expereience Repla	**11**	RecurrentPPO
2	Vanila Policy Gradien	**7**	REINFORCE	**12**	Dueling-DQN
3	SAC	**8**	QR-DQN	**13**	A2C
4	DDPG	**9**	TQC	**14**	A3C
5	SARSA	**10**	ARS (Augmented F	**15**	GAIL

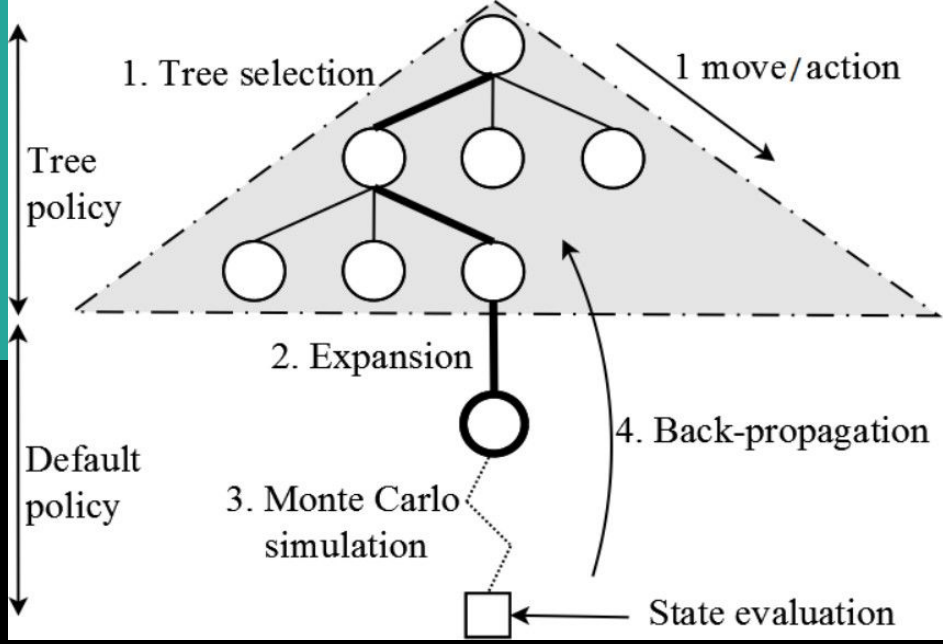


Monte Carlo Simulation

[män-tē ‘kär-‘lō sim-yə-‘lā-shən]

A model used to predict the probability of a variety of outcomes when the potential for random variables is present.

Investopedia



Monte Carlo Tree Search

-Group 1

Introduction

- The Monte Carlo Tree Search (MCTS) Algorithm is a popular tree-based Reinforcement Learning Approach [1]
- States are represented as nodes and actions are edges
- Consists of four repeated steps:
  - Selection
  - Expansion
  - Simulation
  - Backpropagation

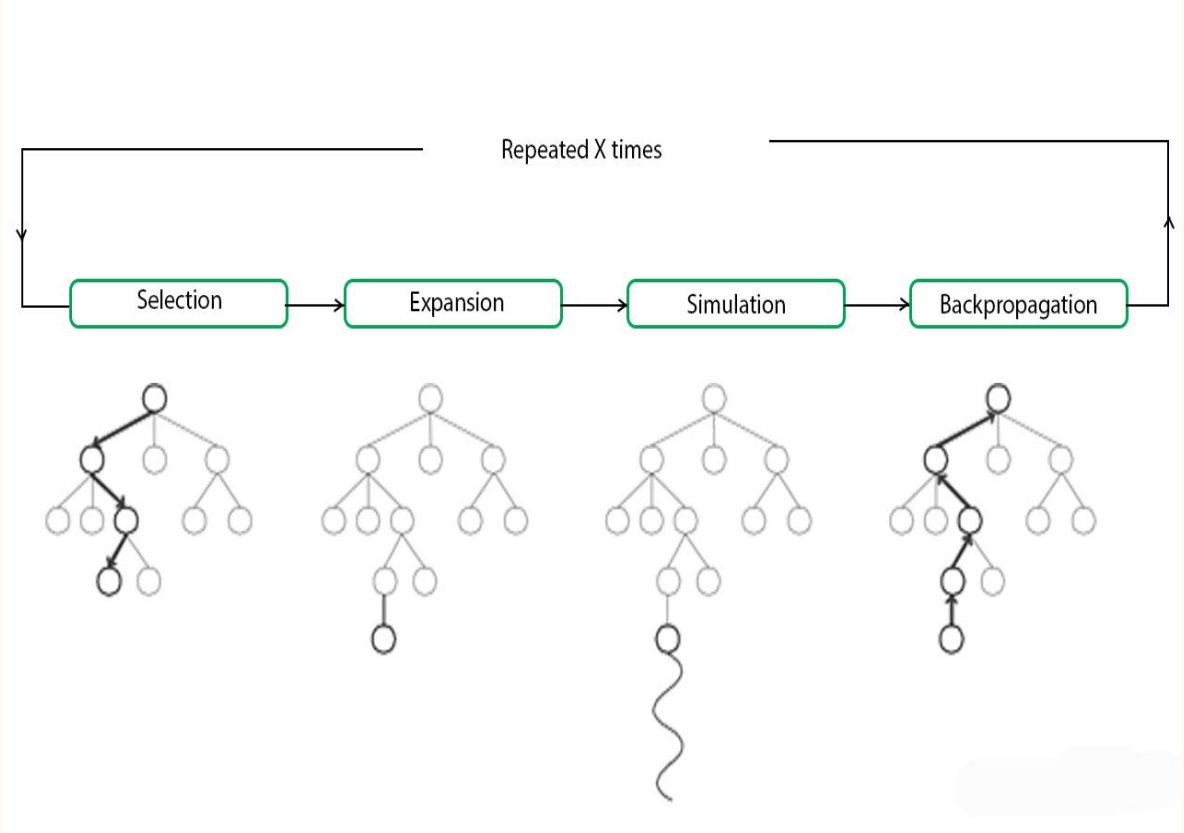


Image from [3]

Selection

- Start from a given node
- Select child node based on tree node policy
- Repeat until leaf node is reached
- Leaf node : Node that hasn't been expanded/explored yet
- Tree Policy : function used to assign value for node selection
    - UCB is the most popular

$$\frac{w_i}{s_i} + c\sqrt{\frac{\ln s_p}{s_i}}$$

- w_i : this node's number of simulations that resulted in a win
- s_i : this node's total number of simulations
- s_p : parent node's total number of simulations
- c : exploration parameter

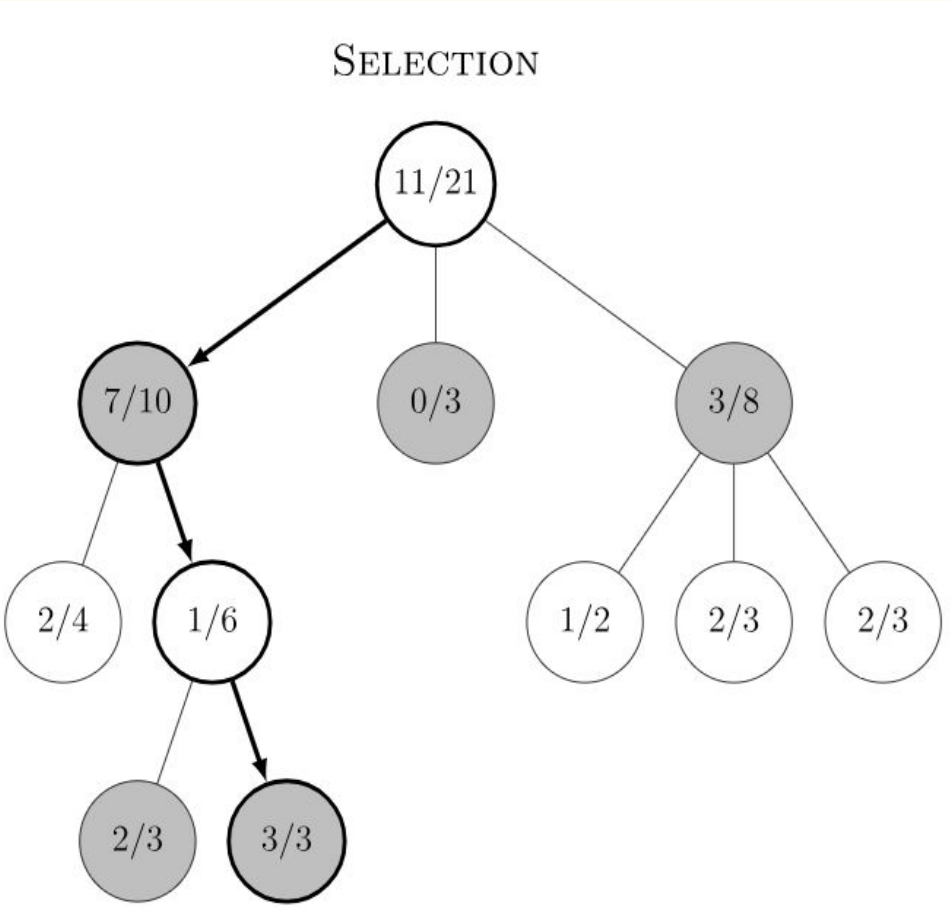


Image from [3]

Expansion

- Expand leaf node by randomly applying one of the possible actions and generating a new node

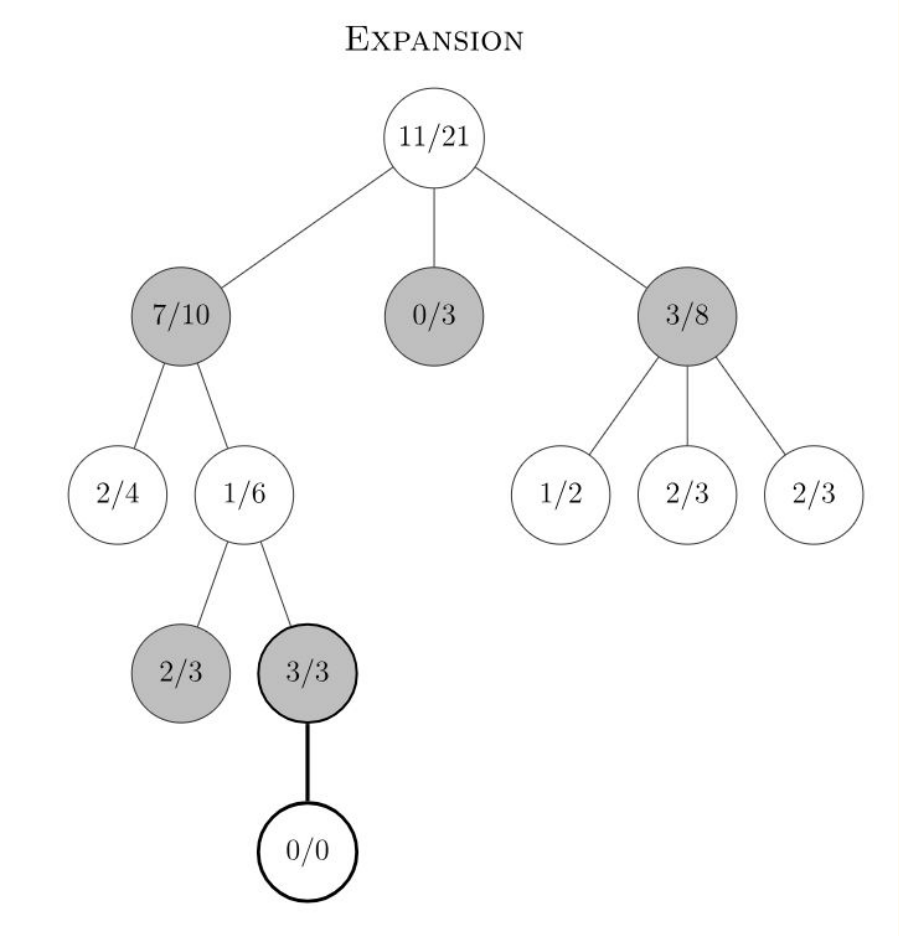


Image from [3]

Simulation

- Check if generated node is terminal state
- If yes, stop
- If no, repeat by choosing another random action and generating another node
- Simulation also called rollout policy
- Rollout policy can be applied multiple times to explore more space at the cost of processing speed

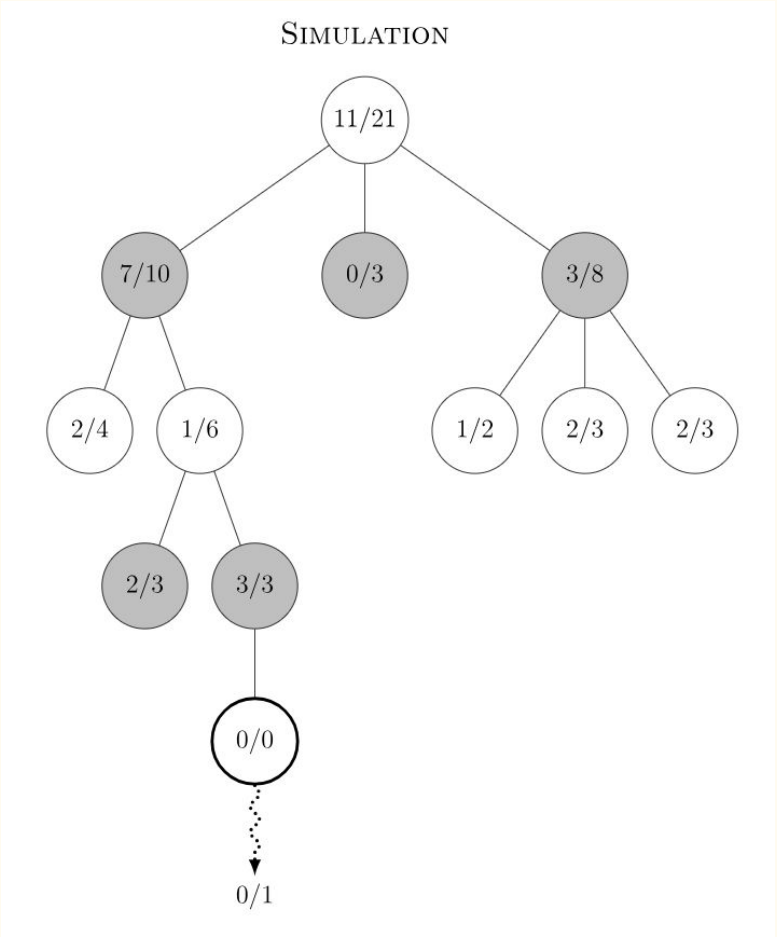


Image from [3]

Backpropagation

- Reward at terminal state is passed up along the same path
- Nodes are updated with new w_i, si, and s_p values
- Repeated for given root node for a certain number of simulations or certain amount of time
- Action chosen based on best score of child node of given root node

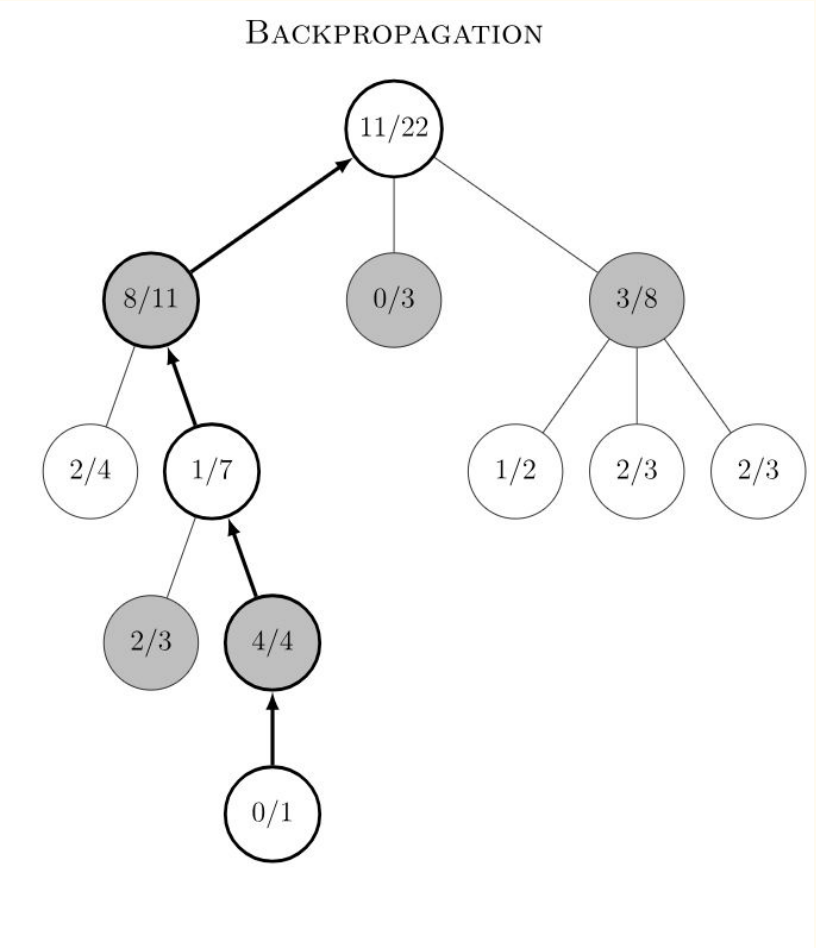


Image from [3]

Pros

- Random selection enables quick navigation of large state spaces
- Tree policy supports balanced approach to exploration and exploitation
- Does not require prior knowledge of environment (model free learning)

Cons

- Can require a lot of iterations to converge to a good solution
- Performance strongly dependent on number of simulations/time allotted

Applications

- Works well in board games, most popularly known to be used in AlphaGo
- Works well for path finding and decision making in simple robots
- Has been used to make AI-powered opponents in video games [2]
- Pros mentioned before explain why it is good in these scenarios

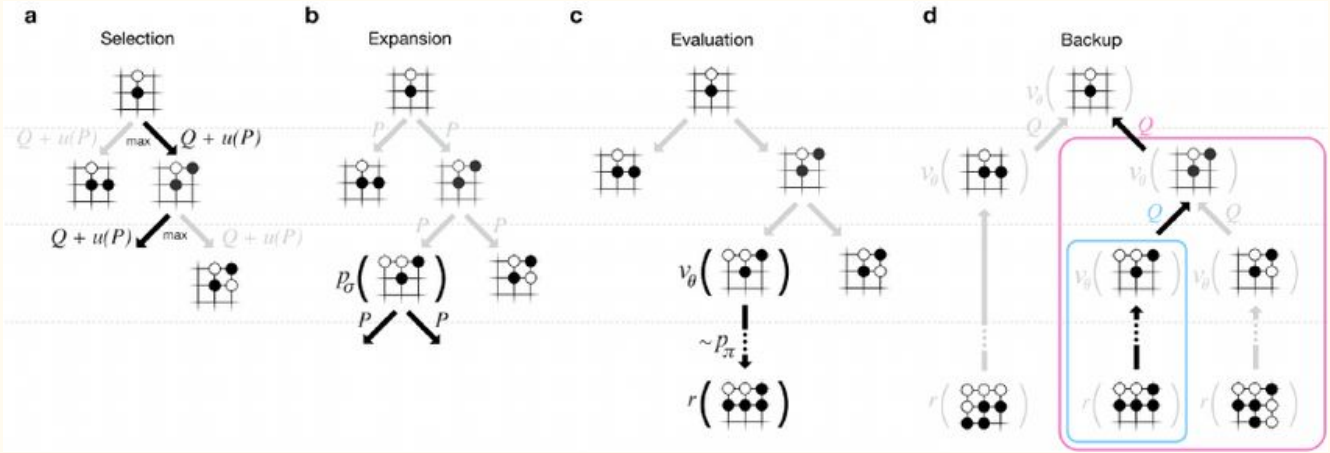


Image from [4]

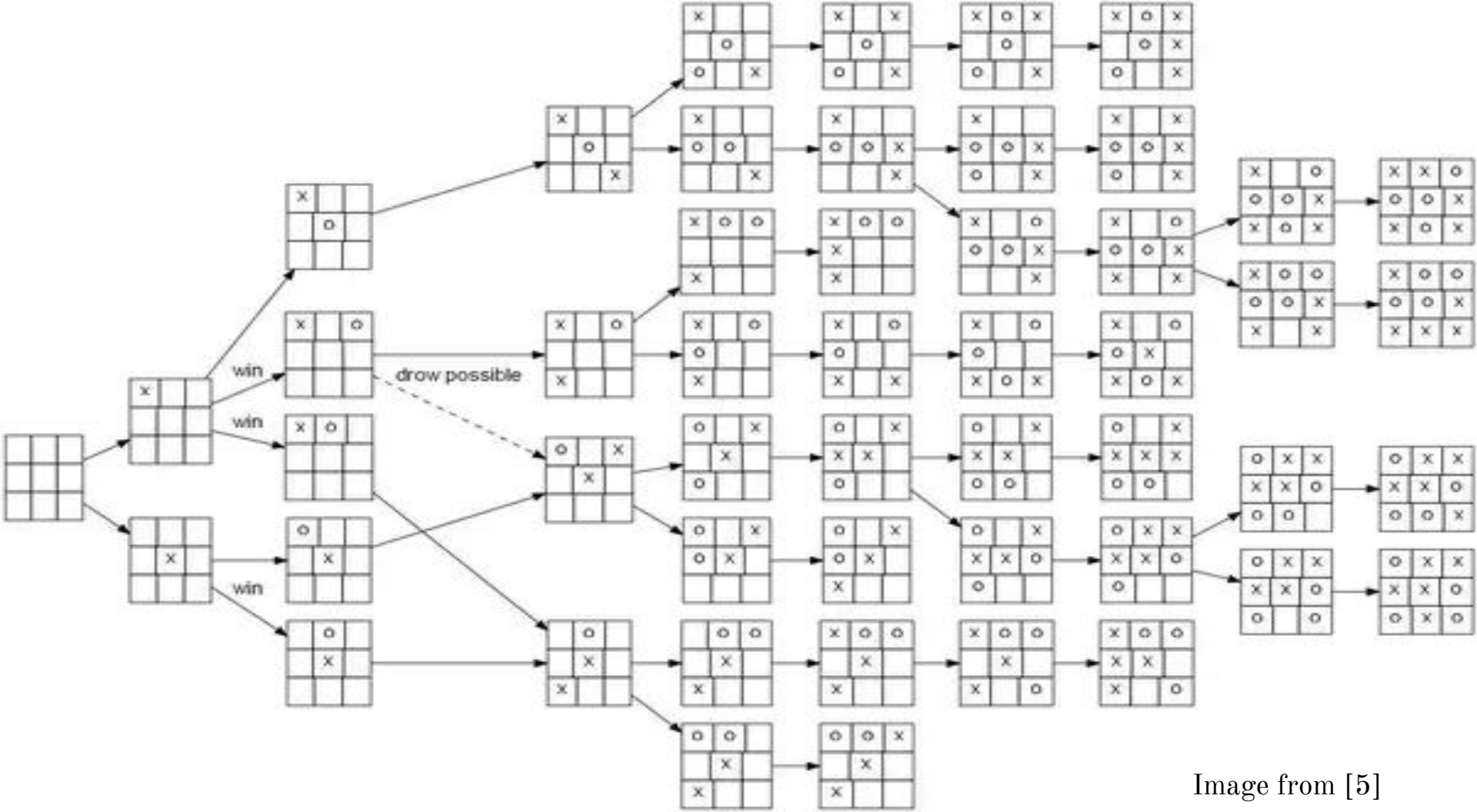


Image from [5]

References

[1] C. B. Browne et al., "A Survey of Monte Carlo Tree Search Methods," in IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, no. 1, pp. 1-43, March 2012, doi: 10.1109/TCIAIG.2012.2186810.

[2] Schrittwieser, Julian, et al. "Mastering atari, go, chess and shogi by planning with a learned model." *Nature* 588.7839 (2020): 604-609.

[3] Chaslot, Guillaume M. Jb, et al. "Progressive strategies for Monte-Carlo tree search." *New Mathematics and Natural Computation* 4.03 (2008): 343-357.

[4] Silver, David & Huang, Aja & Maddison, Christopher & Guez, Arthur & Sifre, Laurent & Driessche, George & Schrittwieser, Julian & Antonoglou, Ioannis & Panneershelvam, Veda & Lanctot, Marc & Dieleman, Sander & Grewe, Dominik & Nham, John & Kalchbrenner, Nal & Sutskever, Ilya & Lillicrap, Timothy & Leach, Madeleine & Kavukcuoglu, Koray & Graepel, Thore & Hassabis, Demis. (2016). Mastering the game of Go with deep neural networks and tree search. Nature. 529. 484-489. 10.1038/nature16961.

[5] De, Ridip & Mahapatra, Rajendra & Chakraborty, Partha Sarathi. (2016). Exploring and Expanding the World of Artificial Intelligence. International Journal of Computer Applications. 134. 1-4. 10.5120/ijca2016907802.

Thank You!

Vanilla Policy Gradient

Group 2

Introduction

- Vanilla Policy Gradient (VPG) is a fundamental reinforcement learning algorithm used to train agents in a trial-and-error manner.
- At a high level, VPG focuses on improving an agent's behavior over time to maximize the total rewards it can earn in an environment.
- It achieves this by directly adjusting the agent's policy, which is like its strategy, to increase the likelihood of taking actions that lead to higher rewards.
- VPG is a simple and intuitive approach to reinforcement learning that forms the basis for more advanced algorithms, making it an important concept in the field.

Working

- VPG trains a stochastic policy in an on-policy way. This means that it explores by sampling actions according to the latest version of its stochastic policy. The amount of randomness in action selection depends on both initial conditions and the training procedure. Over the course of training, the policy typically becomes progressively less random, as the update rule encourages it to exploit rewards that it has already found. This may cause the policy to get trapped in local optima.
- PPO is an advancement over VPG and addresses its high variance issue by using a surrogate objective. PPO generally outperforms VPG in terms of stability, sample efficiency, and convergence speed.

Implementation

Algorithm 1 Vanilla Policy Gradient Algorithm

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for $k = 0, 1, 2, \ldots$ **do**
  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
  Compute rewards-to-go $\hat{R}_t$.
  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
  Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

  Compute policy update, either using standard gradient ascent,

$$\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k,$$

  or via another gradient ascent algorithm like Adam.
  Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

  typically via some gradient descent algorithm.
end for

Working

Initialization (Lines 1-2):

Initialize the policy parameters (θ) and the value function parameters (ϕ).

Main Loop (Lines 2-9):

The algorithm iterates over episodes or iterations, indexed by k.

Data Collection (Line 3):

Collect a set of trajectories (D) by running the current policy (θ) in the environment. Trajectories consist of state-action pairs and rewards.

Working

Compute Rewards-to-Go (Line 4):

Compute the rewards-to-go (R) for each time step t in the trajectories. The rewards-to-go is the total discounted reward from time step t onwards till termination for that state.

Compute Advantage Estimates (Line 5):

Using any method of advantage estimation (e.g., subtracting a value function estimate from the Q Value), compute advantage estimates (A_t) based on the current value function ($V = \theta$).

Compute Policy Gradient (Line 6):

Estimate the policy gradient ($\nabla\theta J(\theta)$) using the advantage estimates (A_t) and the log probabilities of actions ($\log\pi\theta (a|s)$) according to the current policy. This step calculates how policy parameters should be adjusted to maximize expected return.

Implementation

Algorithm 1 Vanilla Policy Gradient Algorithm

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for $k = 0, 1, 2, \ldots$ **do**
  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
  Compute rewards-to-go $\hat{R}_t$.
  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
  Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

  Compute policy update, either using standard gradient ascent,

$$\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k,$$

  or via another gradient ascent algorithm like Adam.
  Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

  typically via some gradient descent algorithm.
end for

Working

Policy Update (Line 7):

Update the policy parameters using gradient ascent, typically following the standard gradient ascent rule ($\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\theta)$), where α is the learning rate. Alternatively, another gradient ascent algorithm like Adam can be used for the update.

Value Function Update (Line 8):

Fit value function by regression on mean-squared error: typically via some gradient descent algorithm

Iteration (Line 9):

The algorithm proceeds by iteratively collecting data, computing rewards-to-go and advantage estimates, estimating policy gradients, updating the policy parameters, and updating the value function. This process continues until convergence or a predefined stopping criterion is met, improving the policy to maximize the expected return.

Pros and cons

Pros

- simple to understand and implement, making it a good starting point for those new to RL.
- VPG is an on-policy algorithm that can be used for environments with either discrete or continuous action spaces
- VPG directly optimizes the expected return by maximizing the objective function, making it suitable for applications where optimizing the policy is the primary goal.

Cons

- VPG often suffers from high variance in the policy gradients, which can lead to slow convergence and instability during training.
- Can be stuck in local optima
- Unlike value-based methods, VPG does not explicitly learn a value function, which can limit its ability to generalize and estimate the value of states.


0
30
60
300
NE
E
W
MS
CHENNAI
Mamallapuram
PRADESH
Nellore
Chennel
PORT

Applications

- VPG is an on-policy algorithm.
- VPG can be used for environments with either discrete or continuous action spaces.
- The Spinning Up implementation of VPG supports parallelization with MPI.
- Vanilla Policy Gradient (VPG) and its variants find applications across a wide range of domains due to their capability to directly learn policy parameterizations and handle continuous action spaces. Here are some notable applications of VPG in various fields:

a. Robotics
b. Game playing
c. NLPs
d. Healthcare
e. Finance
f. Autonomous vehicles

References

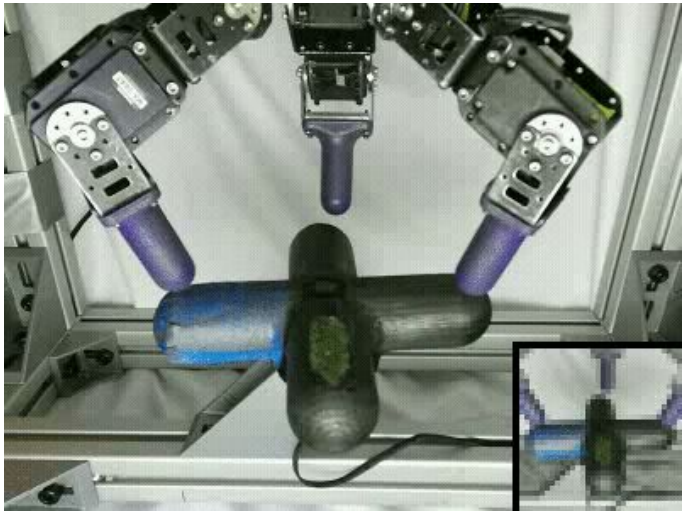
- https://spinningup.openai.com/en/latest/algorithms/vpg.html
- https://medium.com/analytics-vidhya/a-deep-dive-into-vanilla-policy-gradients-3a79a95f3334
- https://medium.com/@alancooney/vanilla-policy-gradient-from-scratch-3c9ebb4de441

Thank you!

CSE574 - Planning & Learning Methods in AI

Soft Actor-Critic

Group-3



Introduction

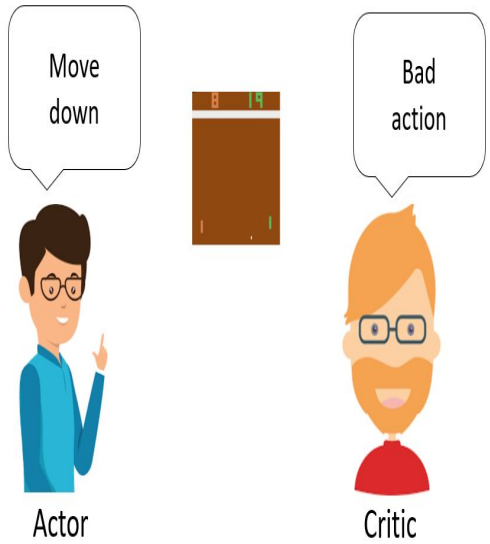
- Soft Actor-Critic (SAC) is an Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor, jointly developed by UC Berkeley and Google.

- It is considered as one of the most efficient RL algorithms to apply to real-world robotics.

- SAC is model-free, online, off-policy, actor-critic RL algorithm. Computes optimal policy that maximizes both the long-term expected reward and entropy of the policy.

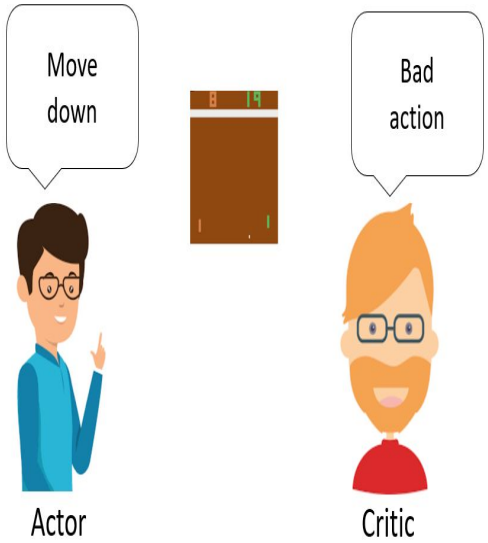
Basic Components

- ACTOR : The actor is responsible for learning the policy. In SAC, the policy is represented as a stochastic policy, meaning it outputs a probability distribution over actions for a given state.
- CRITIC : The critic estimates the state-action value function (Q-function). It evaluates how good it is to be in a particular state and take a specific action.



Basic Components

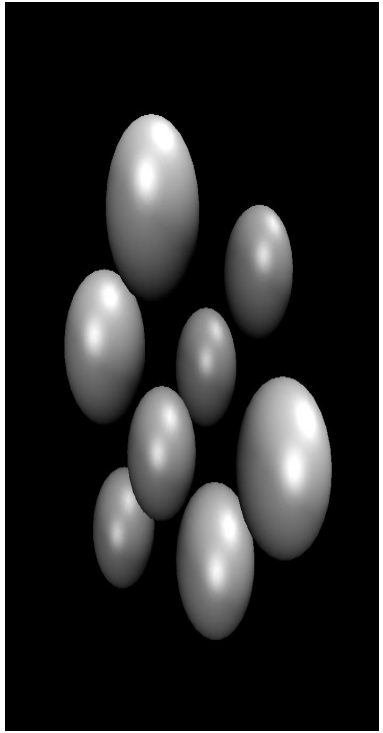
Actor-Critic methods address the limitations of Policy Gradient (lack of foresight) and Q-learning (inability to scale to continuous action spaces) by combining both ideas into one uniform framework.



Key Mechanisms

1. Entropy Regularization

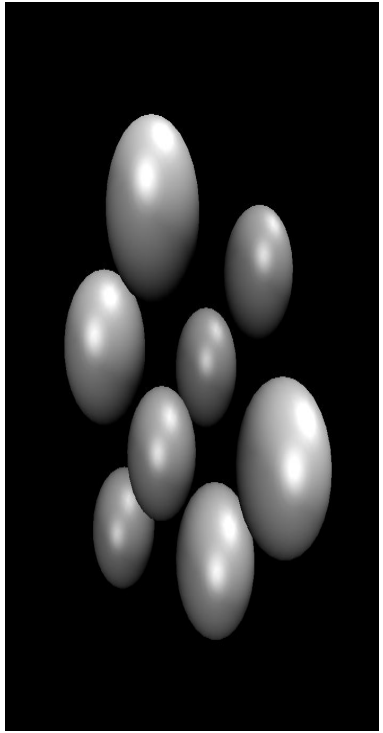
- One of the distinctive features of SAC is its use of entropy regularization.
- Entropy encourages the policy to explore more by adding uncertainty to the action distribution. This prevents the agent from becoming overly deterministic, allowing for more robust learning.
- In entropy-regularized reinforcement learning, the agent gets a bonus reward at each time step proportional to the entropy the policy at that time step.



Key Mechanisms

1. Entropy Regularization

$$\pi^* = \arg\max_{\pi} \mathop{\mathrm{E}}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t\Big(R(s_t,a_t,s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right)\Big)\right]$$



Key Mechanisms

2. Soft Q-Network

- To stabilize learning, SAC uses a modification of the Q-network known as the "soft Q-network." The soft Q-network penalizes overestimation errors and promotes more accurate value function estimation.

3. Clipped Double-Q Learning

- Like the TD3 algorithm, SAC learns two Q-functions instead of one, and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.

Key Mechanisms

4. Exploration vs Exploitation

- SAC trains a stochastic policy with entropy regularization, and explores in an on-policy way.
- The entropy regularization coefficient α explicitly controls the explore-exploit tradeoff, with higher α corresponding to more exploration, and lower α corresponding to more exploitation.
- At test time, to see how well the policy exploits what it has learned, we remove stochasticity and use the mean action instead of a sample from the distribution.

Algorithm Workflow

1. Learning Q

- SAC concurrently learns a policy π_θ and two Q-functions $Q_{\phi 1}$, $Q_{\phi 2}$.
- Both Q-functions are learned with MSBE minimization, by regressing to a single shared target.
- There is no explicit target policy smoothing. SAC trains a stochastic policy, and so the noise from that stochasticity is sufficient to get a similar effect.

Algorithm Workflow

1. Learning Q

- Before we go to the final form of Q-loss, let's take a moment to discuss how the contribution from entropy regularization comes in. Here is the recursive Bellman equation for the entropy-regularized Q^π -

$$\begin{aligned} Q^\pi(s,a) &= \mathop{\mathrm{E}}_{\substack{s'\sim P\\ a'\sim\pi}}\left[R(s,a,s') + \gamma\left(Q^\pi(s',a') + \alpha H\left(\pi(\cdot|s')\right)\right)\right] \\ &= \mathop{\mathrm{E}}_{\substack{s'\sim P\\ a'\sim\pi}}\left[R(s,a,s') + \gamma\left(Q^\pi(s',a') - \alpha \log \pi(a'|s')\right)\right] \end{aligned}$$

- Approximating the above equation with samples (since it's an expectation) :

$$Q^\pi(s,a) \approx r + \gamma\left(Q^\pi(s',\tilde{a}') - \alpha \log \pi(\tilde{a}'|s')\right), \qquad \tilde{a}' \sim \pi(\cdot|s').$$

Algorithm Workflow

1. Learning Q

- Putting it all together, the loss functions for the Q-networks in SAC are :

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

where the target is given by,

$$y(r,s',d) = r + \gamma(1-d)\left(\min_{j=1,2} Q_{\phi_{\text{targ},j}}(s',\tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s') \right), \qquad \tilde{a}' \sim \pi_\theta(\cdot|s').$$

Algorithm Workflow

2. Learning the Policy

- The policy should, in each state, act to maximize the expected future return plus expected future entropy.
- The policy makes use of the reparameterization trick, in which a sample from $\pi_\theta(.|s)$ is drawn by computing a deterministic function of state, policy parameters, and independent noise.
- The reparameterization trick allows us to rewrite the expectation over actions into an expectation over noise.

Algorithm Workflow

2. Learning the Policy

- The policy is optimized according to :

$$\mathop{\mathrm{E}}_{a\sim\pi_\theta}[Q^{\pi_\theta}(s,a) - \alpha \log \pi_\theta(a|s)] = \mathop{\mathrm{E}}_{\xi\sim\mathcal{N}}[Q^{\pi_\theta}(s,\tilde{a}_\theta(s,\xi)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s,\xi)|s)]$$

- Here, squashed Gaussian distribution is used :

$$\tilde{a}_\theta(s,\xi) = \tanh\left(\mu_\theta(s) + \sigma_\theta(s) \odot \xi\right), \qquad \xi \sim \mathcal{N}(0, I).$$

The *tanh* in the SAC policy ensures that actions are bounded to a finite range.

Pseudocode

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      Update policy by one step of gradient ascent using

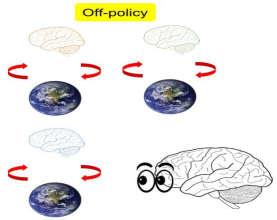
$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \left(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right)\right),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
      Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$

    end for
  end if
until convergence

Pros and Cons

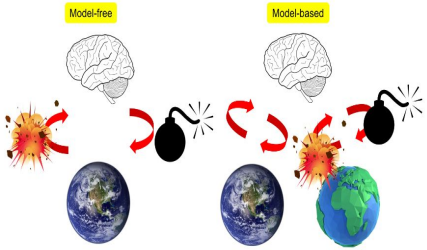


Pros:

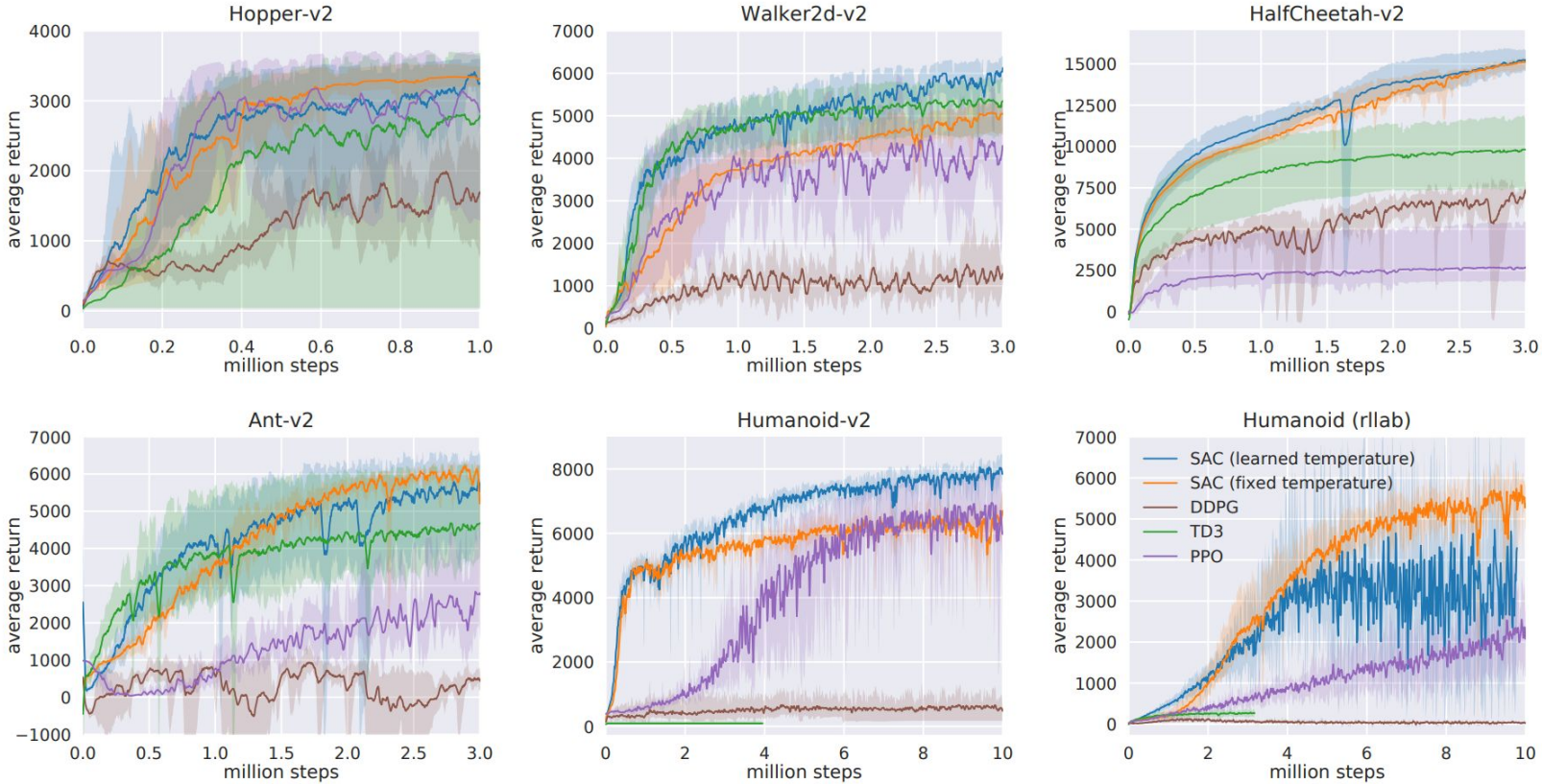
- **Stability** : By using a soft value function and a stochastic policy, which helps reduce variance in the updates and stabilize the learning process.
- **Continuous Action Spaces** : can handle both deterministic and stochastic policies in such settings.
- **Exploration** : incorporates an entropy term in the objective function, which encourages exploration.
- **Off-policy learning** : It can make efficient use of previously collected experience replay data, improving sample efficiency.

Cons:

- **Computationally Intensive** : Especially when using large neural networks or dealing with high-dimensional observations.
- **Lack of Determinism**: SAC policy is stochastic, which may not be suitable for tasks where deterministic behavior is required.



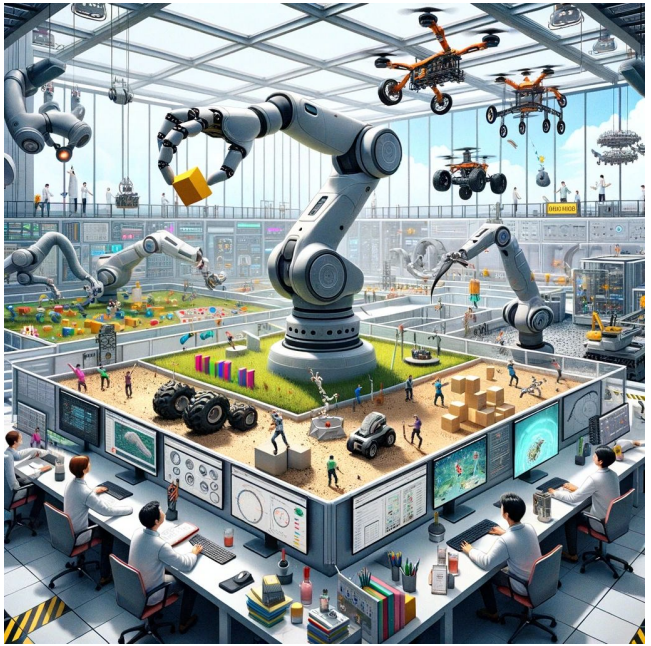
Comparison


Hopper-v2
Walker2d-v2
HalfCheetah-v2
Ant-v2
Humanoid-v2
Humanoid (rllab)
average return
million steps
SAC (learned temperature)
SAC (fixed temperature)
DDPG
TD3
PPO

Applications

Robotics, Autonomous Systems, Industrial Automation.

1. Manipulation with Robotic Arms/Hands:
    a. Tasks: grasping, stacking, inserting objects.
    b. Resilience: external perturbations and partial observations.
2. Locomotion with Robots:
    a. Types: bipedal (walking, running, jumping) and quadrupedal (terrain adaptability).
3. Navigation using Vehicles:
    a. Environments: complex terrains for aerial and ground vehicles.
4. Gaming Applications:
    a. SAC applied to various game scenarios and mechanics.



References

- Haarnoja, T., Zhou, A., Abbeel, P. and Levine, S., 2018, July. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning* (pp. 1861-1870). PMLR.
- Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P. and Levine, S., 2018. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*.
- https://bair.berkeley.edu/blog/2018/12/14/sac/
- https://sites.google.com/view/sac-and-applications

Deep Deterministic Policy Gradient

Group 4:

DDPG: Summary and Key features

DDPG, which stands for Deep Deterministic Policy Gradient, is a model-free, off-policy actor-critic algorithm used for reinforcement learning, particularly in continuous action spaces. It is used for applications like gaming, autonomous vehicle and robotics.

Actor-Critic Architecture: DDPG uses an actor-critic architecture, where the actor is used to approximate the optimal policy directly and the critic is used to approximate the value function. The critic's value estimate is used to guide the policy update.

Deterministic Policy Gradient: Unlike other algorithms that output a probability distribution over actions, DDPG is designed for situations where the action space is continuous. It uses a deterministic policy gradient to improve the policy. The gradient of the expected return with respect to the policy parameters is computed, which guides the updates.

Experience Replay: Like Q-learning-based methods such as DQN, DDPG also utilizes an experience replay buffer. Past experiences are stored in this buffer and random mini-batches from this buffer are used to train the networks. This breaks the correlation between consecutive samples and stabilizes training.

DDPG: Summary and Key features

Target Networks: DDPG employs target networks, both for the actor and the critic, to further stabilize training. This idea was borrowed from DQN. Target networks are essentially copies of the actor and critic networks but with their weights frozen. Periodically, these weights are softly updated to the main actor and critic networks' weights.

Exploration: Exploration in DDPG is tackled by adding noise to the action output. The most common noise added is Ornstein-Uhlenbeck noise, which introduces temporal correlation, making it suited for problems in continuous action spaces.

Batch Normalization: In some implementations of DDPG, batch normalization is applied to different layers of the neural networks to ensure that the various inputs are in a consistent range and have zero mean and unit variance. This can help in accelerating learning.

Pros

- can handle continuous action spaces.
- actor critic architecture stabilizes training.
- off policy training helps stabilize and increase sample efficiency.

Cons

- ddpg is sample inefficient compared to model based.
- it is highly sensitive to hyperparamter.
- It is not good at exploration.

DDPG Algorithm Initialisation

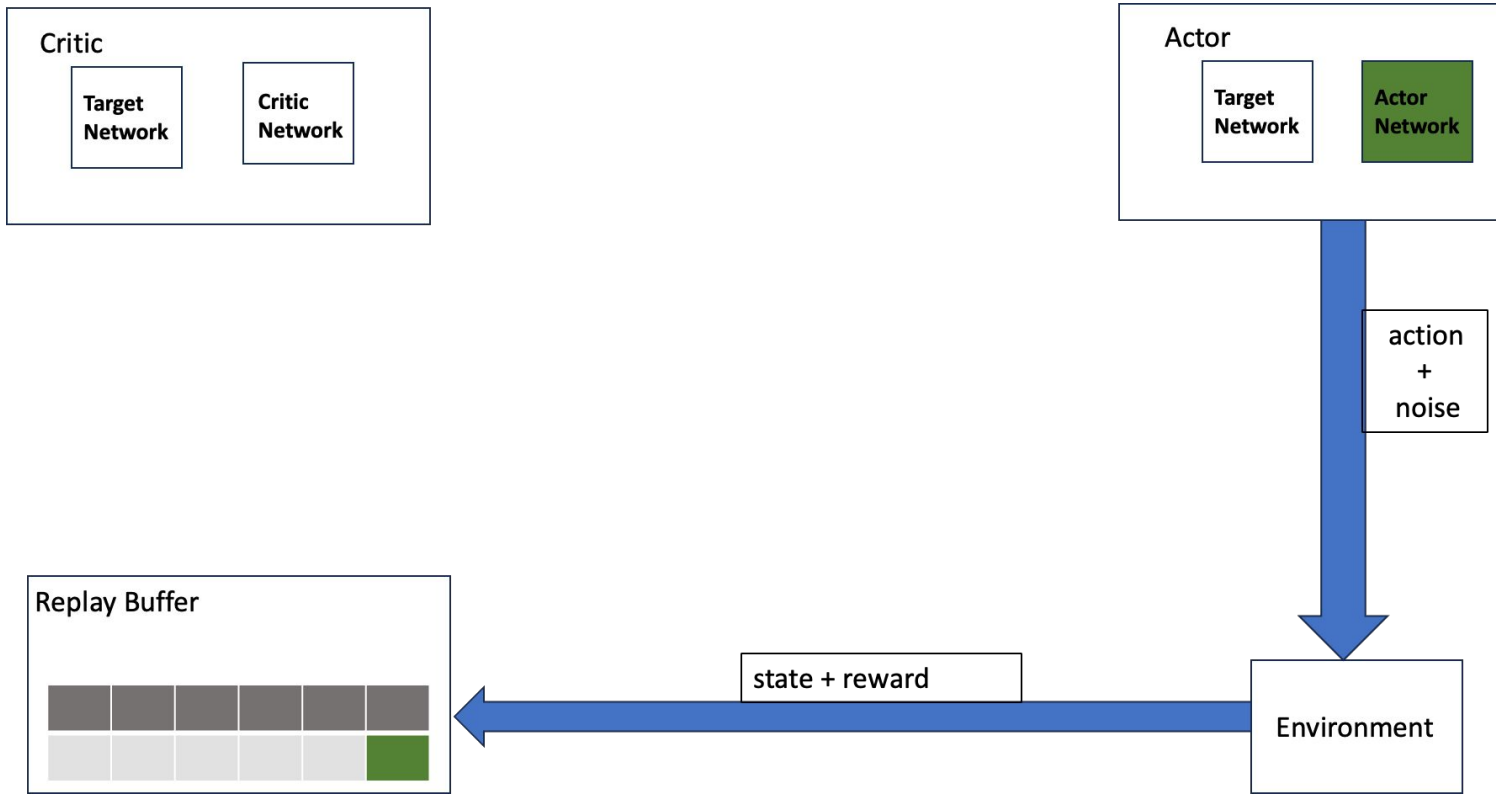
We need to perform the following steps to initialise DDPG:

1. Initialize the actor network and critic network with random weights.
2. Initialize the target networks with the same weights as the main networks.
3. Initialize experience replay buffer.

For each time step:

a. Select an action using the current policy (actor network) and add noise for exploration.

b. Execute the chosen action in the environment and observe the reward and next state.

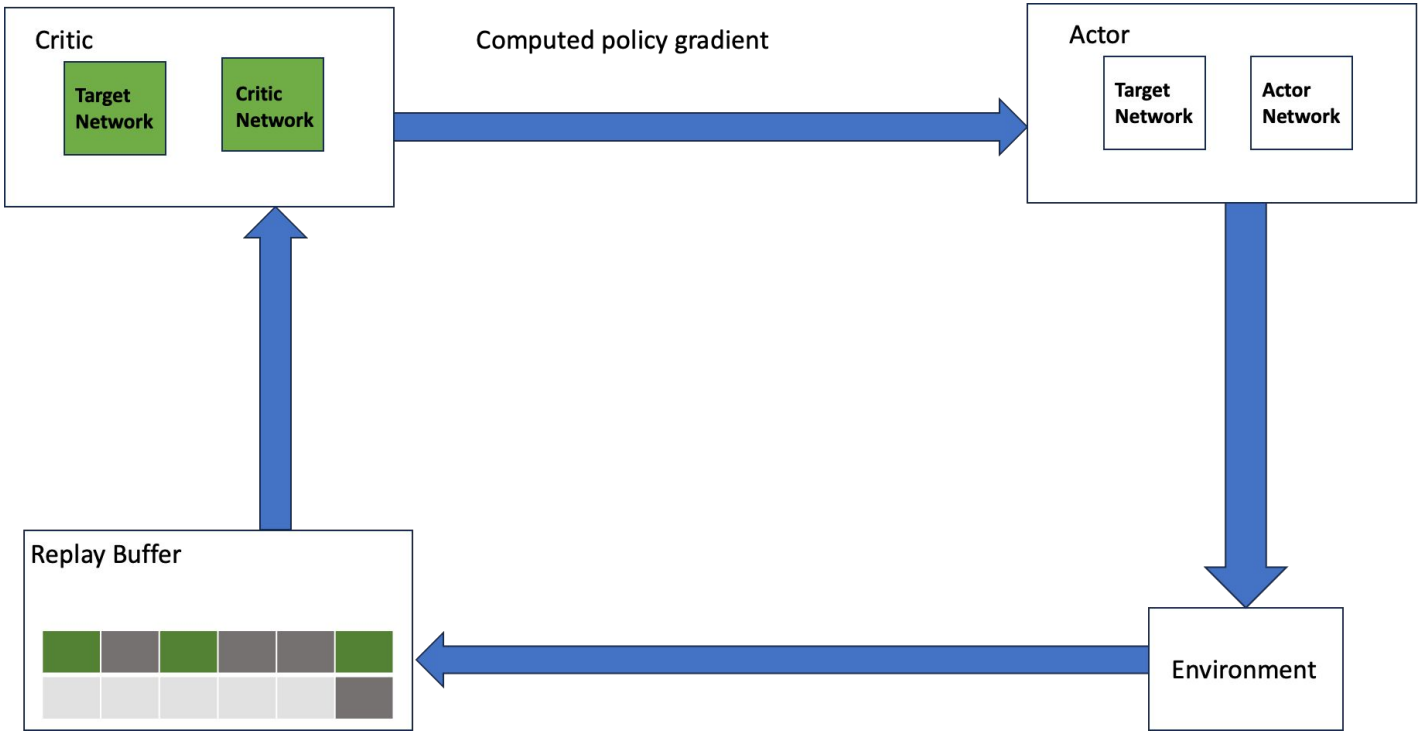


c. Store the transition in the experience replay buffer.

d. Sample a random mini-batch of transitions from the replay buffer.

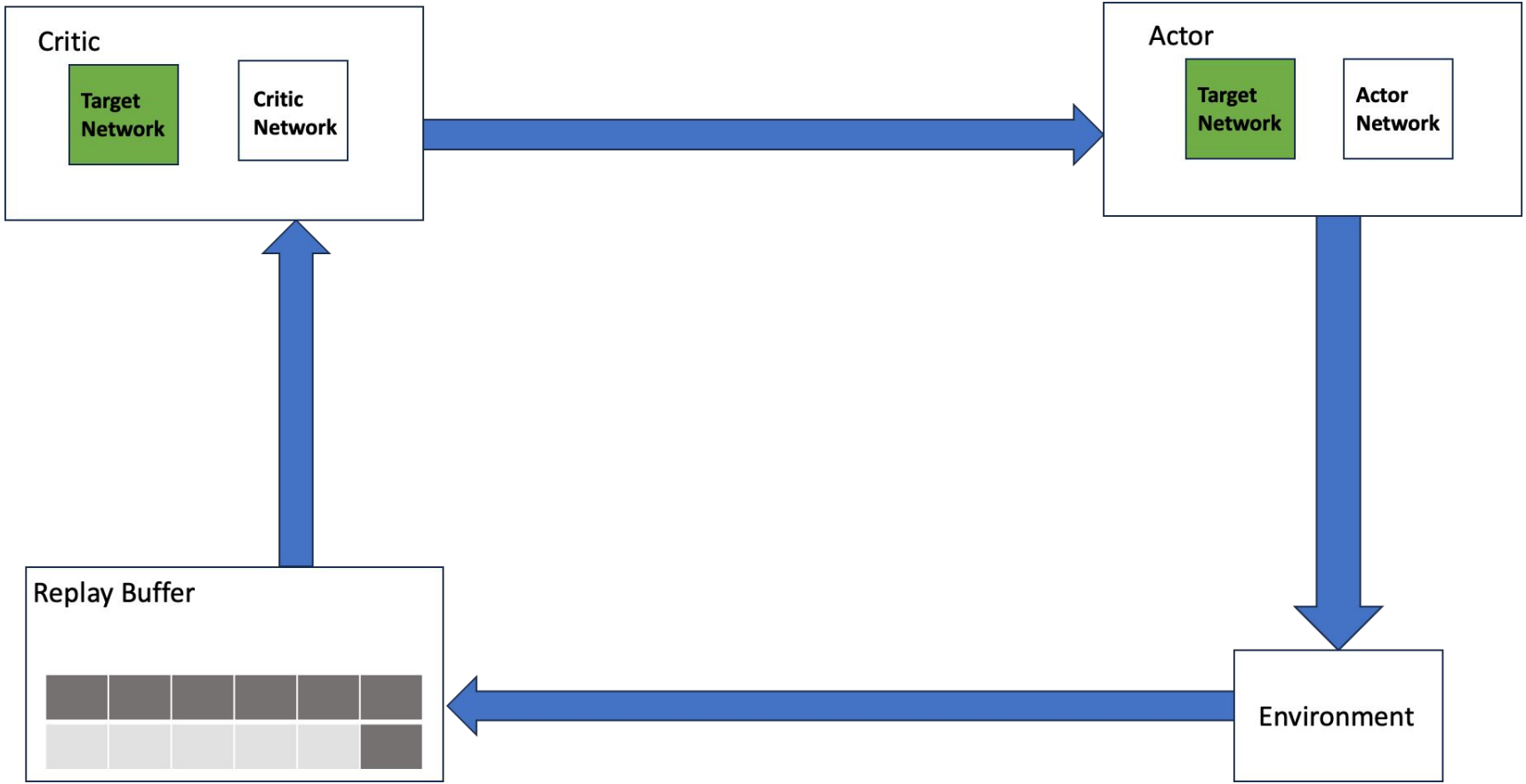
e. Compute the target Q-value using the critic's target network and the reward from the transition.

f. Update the critic by minimizing the Mean Squared Error between the target Q-value and the predicted Q-value.



g. Update the actor using the sampled policy gradient.

h. Softly update the target networks.



DDPG Algorithm Termination and Conclusion

Repeat until convergence or a stopping criterion is met.

DDPG has shown great performance in a variety of tasks, especially those with continuous action spaces like robotic control tasks. It combines ideas from DQN and policy gradient methods, allowing for effective learning in such environments.

Q/A Session

SARSA Reinforcement Learning Algorithm

Group 5

Q Learning

- *Q-Function (Action-Value Function):* The Q-value of a state-action pair represents the expected cumulative reward an agent can achieve.
- Q-Learning follows an off-policy approach, updating Q-values at the current time step using the maximum estimated Q-value at the next, regardless of the current policy.
- Update based on temporal difference:

$$Q(\mathbf{x}_t, a_t) \leftarrow r_t + \gamma \max_{a \in A} Q(\mathbf{x}_{t+1}, a)$$

- Backpropagation for function approximation using NNs:

$$\Delta \mathbf{w}_t = \eta \left[r_t + \gamma \max_{a \in A} Q_{t+1} - Q_t \right] \nabla_w Q_t$$

Temporal Difference Learning

Supervised learning:

$$\Delta w_t = \alpha \left(z - V(s_t)\right) \nabla_w V(s_t),$$

$$w \leftarrow w + \sum_{t=1}^{m} \Delta w_t$$

Where w is a vector of the parameters of our prediction function at time step t, α is a learning rate constant, z is our target value, V (s) is our prediction for input state s

$$z - V_t = \sum_{k=t}^{m} (V_{k+1} - V_k)$$

$$\Delta w_t = \alpha \left(V_{t+1} - V_t\right) \sum_{k=1}^{t} \nabla_w V_k.$$

$$\Delta \mathbf{w}_t = \eta \left[r_t + \gamma \max_{a \in A} Q_{t+1} - Q_t \right] \nabla_w Q_t$$

Q-learning update

Each temporal difference will affect all previous predictions -> How do we use this for faster convergence?

SARSA

$$\Delta \mathbf{w}_t = \eta \left[r_t + \gamma \max_{a \in A} Q_{t+1} - Q_t \right] \sum_{k=0}^{t} (\lambda\gamma)^{t-k} \nabla_w Q_k$$

Q-learning with TD

Is maximum of the Q function the best representation of the next state?

No- incorrect initially, overestimates later

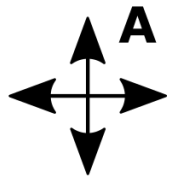
$$\Delta \mathbf{w}_t = \eta \left[r_t + \gamma Q_{t+1} - Q_t \right] \sum_{k=0}^{t} (\gamma\lambda)^{t-k} \nabla_w Q_k$$

SARSA

On policy algorithm

The SARSA Algorithm

1. Initialize the start state S
2. Choose an Action A by using some policy (ex. greedy)
3. Apply Action A and observe the next State S' and the reward R obtained by reaching that state
4. Choose Action A' based on S' by using the policy on current value of Q(S', A')
5. Repeat Steps 2-4 until Terminal State is Reached



$$Q(6, A) = \begin{bmatrix} 0.25 & 0.6 & 0 & 0.25 \end{bmatrix}$$

$$Q(S, A) \leftarrow Q(S, A) + \alpha \cdot \Big(R + \gamma Q(S', A') - Q(S, A) \Big)$$

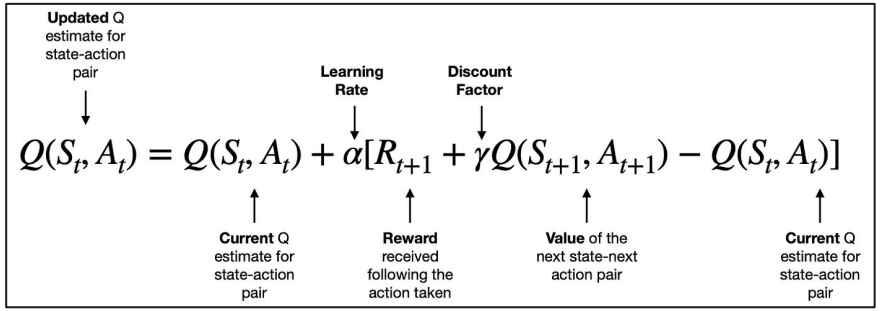
SARSA vs Q-Learning

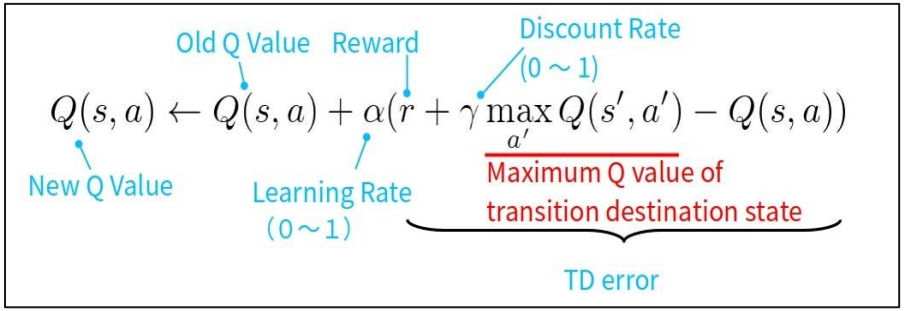
SARSA

On Policy: Updates Q-Function Values based on the current policy

Q-Learning

Off Policy: Updates Q-Function Values based on the maximum expected Q-Value Regardless of the current policy





References

- https://web.stanford.edu/group/pdplab/pdphandbook/handbookch10.html
- https://aleksandarhaber.com/explanation-and-python-implementation-of-on-policy-sarsa-temporal-difference-learning-reinforcement-learning-tutorial/
- http://mi.eng.cam.ac.uk/reports/svr-ftp/auto-pdf/rummery_tr166.pdf

Hindsight Experience Replay

Group 6

Hindsight is 20/20

What is Hindsight Experience Replay?

Imagine you're playing cornhole where the goal is to get your bags on to the board or into the hole (for maximum points).

You're practicing getting the bags in the hole on every shot. Your bags keep landing a little to the right of the hole, so you adjust your stance, aim, and/or velocity after each turn. You eventually make it in the hole, and you try to replicate those stance/aim/velocity choices each time so you consistently make the hole without error.

This is an example of the human ability to learn from mistakes.

What is Hindsight Experience Replay?

Using a standard RL algorithm, there would be no reward for throws that do not result in bags that land outside of the hole.

How can we learn from failure? Move the goal post!

Hindsight Experience Replay (HER) is a replay technique that considers failed attempts to be successful subgoals. The neural network is trained based on completion of these subgoals for broader training and progress towards the primary goal.

Experience Replay*

During training, an agent stores transition tuples (s_t, a_t, r_t, s_t+1) in a **replay buffer**.

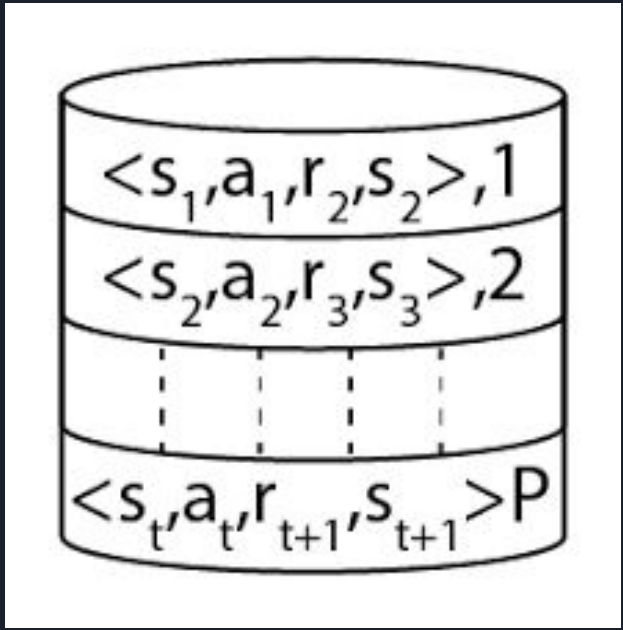
s_t = state

a_t = action

r_t = reward

S_t+1 = next state

Batches of experiences are selected from the replay buffer at random to train the neural network.



*As seen in Deep Q-Networks (DQN)

Hindsight Experience Replay

In HER, Universal Value Function Approximators (UVFA) is applied so states *and* goals are taken into consideration, training an agent to perform multiple tasks instead of just one.

When the agent fails at the primary goal, it considers that experience to be a successful attempt at a subgoal.

During the experience replay, the original goal is replaced by successful subgoals, which are used to train the neural network.

This leads to faster learning based on the “failed” experiences.

Algorithm 1 Hindsight Experience Replay (HER)

Given:

- an off-policy RL algorithm $\mathbb{A}$, ▷ e.g. DQN, DDPG, NAF, SDQN
- a strategy $\mathbb{S}$ for sampling goals for replay, ▷ e.g. $\mathbb{S}(s_0, \ldots, s_T) = m(s_T)$
- a reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{G} \rightarrow \mathbb{R}$. ▷ e.g. $r(s, a, g) = -[f_g(s) = 0]$

Initialize $\mathbb{A}$ ▷ e.g. initialize neural networks
Initialize replay buffer R
for episode = 1, M **do**
  Sample a goal g and an initial state s_0.
  for $t = 0, T-1$ **do**
    Sample an action a_t using the behavioral policy from $\mathbb{A}$:
      $a_t \leftarrow \pi_b(s_t||g)$ ▷ $||$ denotes concatenation
    Execute the action a_t and observe a new state s_{t+1}
  end for
  for $t = 0, T-1$ **do**
    $r_t := r(s_t, a_t, g)$
    Store the transition $(s_t||g,\ a_t,\ r_t,\ s_{t+1}||g)$ in R ▷ standard experience replay
    Sample a set of additional goals for replay $G := \mathbb{S}$(**current episode**)
    for $g' \in G$ **do**
      $r' := r(s_t, a_t, g')$
      Store the transition $(s_t||g',\ a_t,\ r',\ s_{t+1}||g')$ in R ▷ HER
    end for
  end for
  for $t = 1, N$ **do**
    Sample a minibatch B from the replay buffer R
    Perform one step of optimization using $\mathbb{A}$ and minibatch B
  end for
end for

Pros & Cons of HER

Pros

- Allows for learning when rewards are sparse & binary.
- Avoids the need for complicated reward engineering & domain knowledge.
- Can be combined with any off-policy RL algorithm such as DQN, DDPG, SSC etc.
- Doesn't require control over distribution of initial environmental states, as with explicit curriculum.

Cons

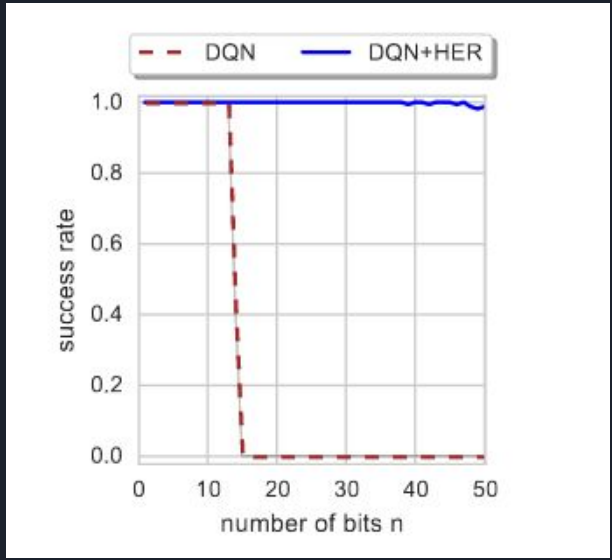
- Limited to goal-oriented tasks.
- Misleading samples can introduce bias to the learning process (i.e., inaction could be considered success).
- Can focus on failed trajectories instead of the goal in some cases.
- Increased computational complexity.

Deep Q-Networks (DQN) with and without HER

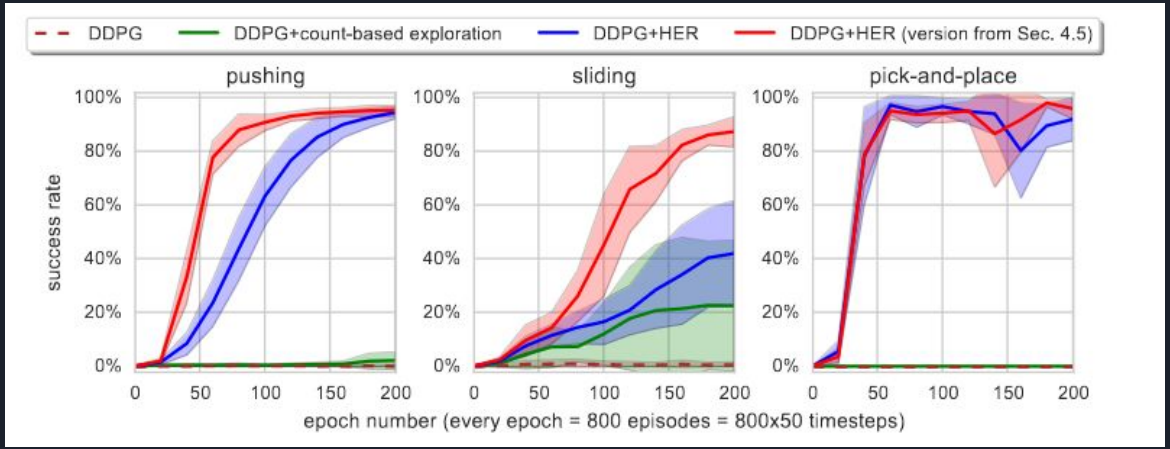
RL algorithms are significantly limited with sparse/binary rewards.

In a bit flipping environment:

- DQN alone can only solve $n < 13$
- DQN with HER solves up to $n = 50$



Deep Deterministic Policy Gradients (DDPG*) with and without HER



Given pushing, sliding, and pick-and-place tasks:

- DDPG alone can't complete the tasks.
- DDPG with count-based exploration makes some progress on the sliding task.
- DDPG with HER completes all tasks.

**DDPG covered by Group 4*

Applications of HER

- Primary application is scenarios with sparse or binary rewards.
- Used with model-free, off-policy algorithms since they are typically used to handle environments with sparse or binary rewards and often use general experience replay.
- Can be used in both discrete & continuous action spaces.

References

- Hindsight Experience Replay: https://arxiv.org/pdf/1707.01495.pdf
- Bias-Reduced Hindsight Experience Replay with Virtual Goal Prioritization: https://arxiv.org/pdf/1905.05498.pdf

Understanding the REINFORCE Algorithm

A Deep Dive into Policy Gradient Methods

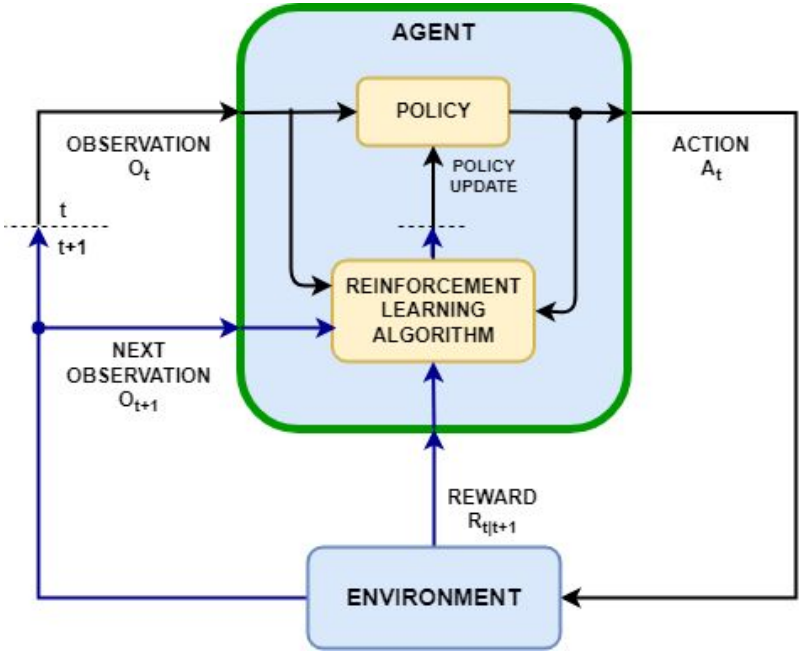
(Group 7)

What is REINFORCE and how does it work?

- REINFORCE stands for "REward Increment = Nonnegative Factor * Offset Reinforcement * Characteristic Eligibility," and it was introduced by Ronald J. Williams in 1992.
- A machine learning technique that focuses on training a policy in order to maximize the expected cumulative reward in an environment.
    - It uses a parametric policy (typically a neural network).
    - Generates a trajectory of states, actions, and rewards.
    - Updates the policy parameters based on the expected return.
- To put it concisely, this is a type of machine learning where an agent interacts with an environment and learns to make a sequence of decisions (actions) to maximize a cumulative reward. The agent explores different actions and learns from the consequences of its actions.

Training Process

- Collect trajectories by interacting with the environment.
- Compute returns and rewards.
- Update the policy parameters to increase the probability of actions that lead to higher rewards.



REINFORCE Pseudocode

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
    Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

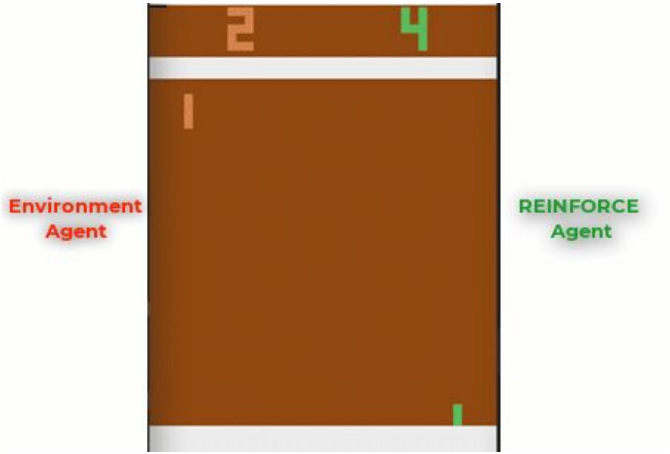
Pros and Cons of REINFORCE

- **Advantages**:
  - Suitable for high-dimensional action spaces.
  - Handles stochastic policies.
  - Converges to a local optimum.
- **Disadvantages**:
  - High variance in training.
  - Inefficient for long trajectories.
  - Struggles with sparse rewards.

- **Comparison to Other RL Algorithms:** REINFORCE stands out as a policy gradient method, emphasizing direct policy optimization. It is versatile and can work in various environments unlike other algorithms.
  - *Q-learning*: better suited for discrete action spaces
  - *DDPG (Deep Deterministic Policy Gradient)*: better suited for precise control in continuous action spaces

Applications of REINFORCE

- Model-free.
- Suitable for discrete and continuous state/action spaces.
- Used in on-policy setting
- Real World Examples:
  - Training agents for playing games.
  - Robot control.
  - Natural language processing tasks.



References

- Guest_Blog. (2020). REINFORCE Algorithm: Taking baby steps in reinforcement learning. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2020/11/reinforce-algorithm-taking-baby-steps-in-reinforcement-learning/
- Papers with Code - REINFORCE Explained. (n.d.). https://paperswithcode.com/method/reinforce
- Reinforcement Learning Agents - MATLAB & Simulink. (n.d.). https://www.mathworks.com/help/reinforcement-learning/ug/create-agents-for-reinforcement-learning.html
- Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press.

Thank You!

Any Questions?

QR-DQN Learning

Group 8

Deep Q-Network

Immediate Reward

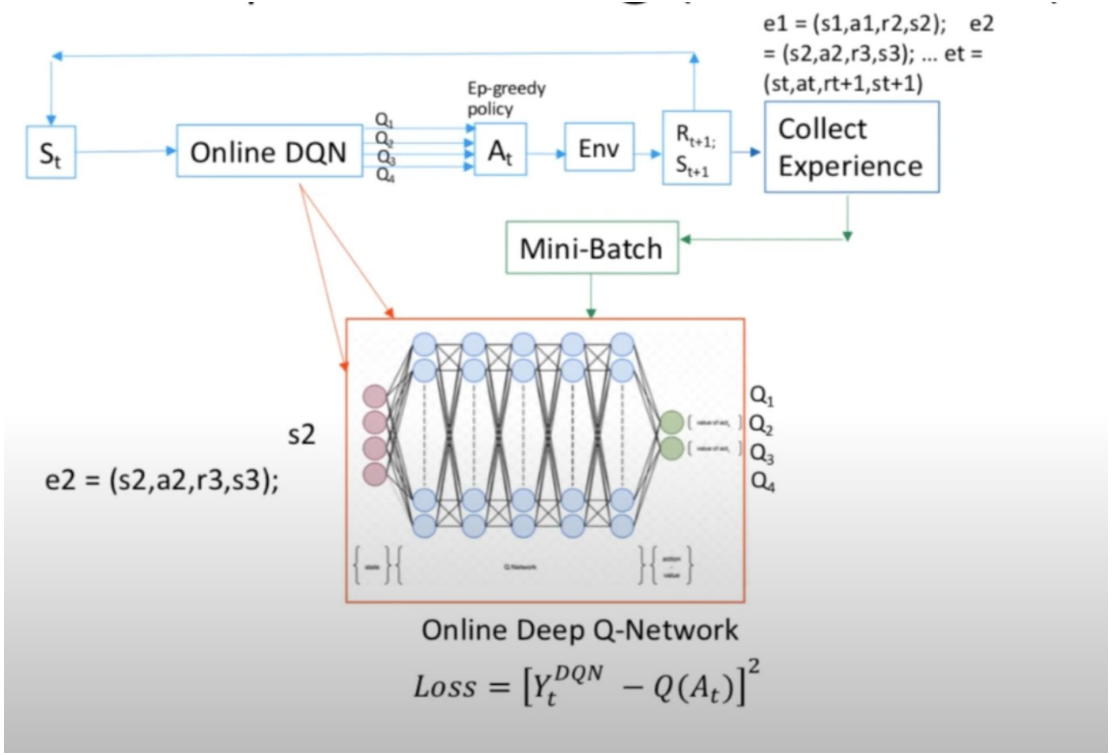
$(S_t, A_t) \rightarrow Env \rightarrow R_{t+1}, S_{t+1}$

Total Future Reward

$Q_t = R_{t+1} + R_{t+2} + \ldots\ldots + R_{t+n}$

Discounted Future Reward

$Q_t = R_{t+1} + \gamma Q_{t+1}$



Why Perform Quantile Regression?

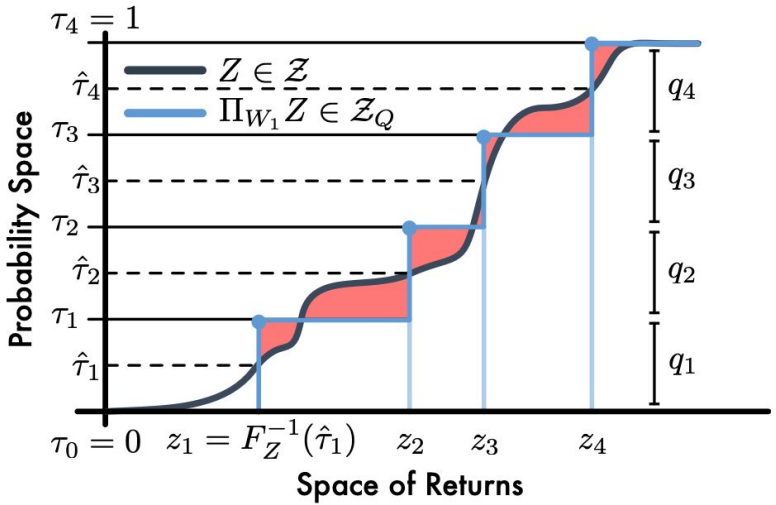
- Traditional Deep Q-Networks (DQNs) learn to estimate the expected value of taking a given action in a given state. This expected value is often referred to as the Q-value.

- Quantile Regression DQN (QR-DQN) is a variant of DQN that learns to estimate the quantiles of the Q-value distribution.

- This means that QR-DQN learns to estimate the probability that the Q-value will be greater than a certain threshold.

QR-DQN Working

- QR-DQN uses the same architecture as traditional DQN, but with a few modifications.
- Output layer of the QR-DQN is a quantile regression network. This network learns to estimate the quantiles of the Q-value distribution.
- QR-DQN is trained using the same experience replay procedure as traditional DQN. However, the loss function used to train the QR-DQN network is different.
- The QR-DQN loss function is designed to minimize the difference between the predicted quantiles and the actual quantiles of the Q-value distribution.



QR-DQN Flow

Step 1: Agent interacts with the environment and takes an action

Step 2: Agent observes the new state and reward

Step 3: Agent stores its experience in a memory buffer

Step 4: Agent samples from the memory buffer and trains the QR-DQN network

Step 5: QR-DQN network learns to estimate the quantiles of the Q-value distribution

Step 6: Agent uses the QR-DQN network to select actions in the environment

Advantages and disadvantages

- QR-DQN addresses the uncertainty in Q value estimates - even in uncertain environments this can result in strong or certain model.
- With help of different quantiles in the distribution, exploration can also be better included even in less rewarding regions.
- Due to which even if outliers occur our model become less sensitive to it - more robust.
- It is more adaptable because quantile values are learnt from the data not pre fixed
- As the data is updated, it also updates the distributions and fine tunes them, which makes it more adaptable.

- It needs more computational power as it is dynamically updating the quantile number and values
- The higher the quantiles - the more accurate the picture of complex data ; but it also results in the higher the number of quantiles - higher the computational power - that gives slower convergence
- It requires proper and large number if samples to learn well and perform
- It is very sensitive to number of quantiles and due to its distribution nature, if quantiles are high - it is most likely going to overfit

Applications

As we have already learnt any environment with good uncertainty is a good fit for QR-DQN:

- **Stock markets and finance trading:** Because QR-DQN represents the entire distribution using quantiles, it captures the environment along with its uncertainties and gives out a state-action pair. Considering the amount of uncertainty in trading, this fine tunes itself and becomes robust when trained properly.
- **Autonomous vehicles:** QR-DQN can shine in this region as uncertainty on the roads is definitely certain and we need the model to be more robust when faced with situations like adverse weather conditions or traffic.
- **Datacenter facility Failure:** As we know that it adapts it changing environment, if a facility had to break downit dynamically allots the resources to mitigate the impacts of the failure which results in efficient and risk aware decision making.
- **Price optimization in retail:** Given proper historically relevant data, QR-DQN can estimate the distribution function; through which one can redefine their inventory and also mark down or up the prices looking at the estimated demand and supply.

The algorithm is hard to set up and needs strong understanding in distributional statistics. QR-DQN can over engineer in certain environments and using a traditional RL algorithm can do better in such scenarios. It is essential to first evaluate the need of the algorithm in given task and use.

CSE 574 Planning and Learning in AI

TRUNCATED QUANTILE CRITICS (TQC)

GROUP 9:

TQC AND ITS WORKING

WHAT IS TQC?

- The overestimation control is coarse – impossible to take the minimum over a fractional number of approximators
- Aggregation with *min* ignores all other estimates except for the minimal one, diminishing the power of the approximator.
- Builds on SAC, TD3 and QR-DQN.
- Quantile regression to predict a distribution for the value function.
- Truncates the quantiles predicted by different networks.

IDEALOGY

- TQC blends 3 ideas:
  - Distributional representations of the critic
  - Truncation of overestimation of critics
  - Ensembling of multiple critics without wastage

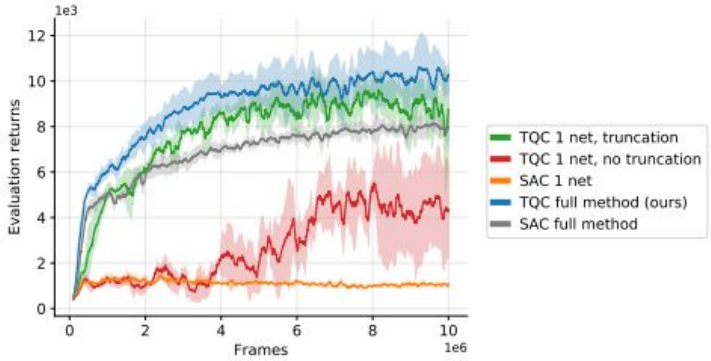
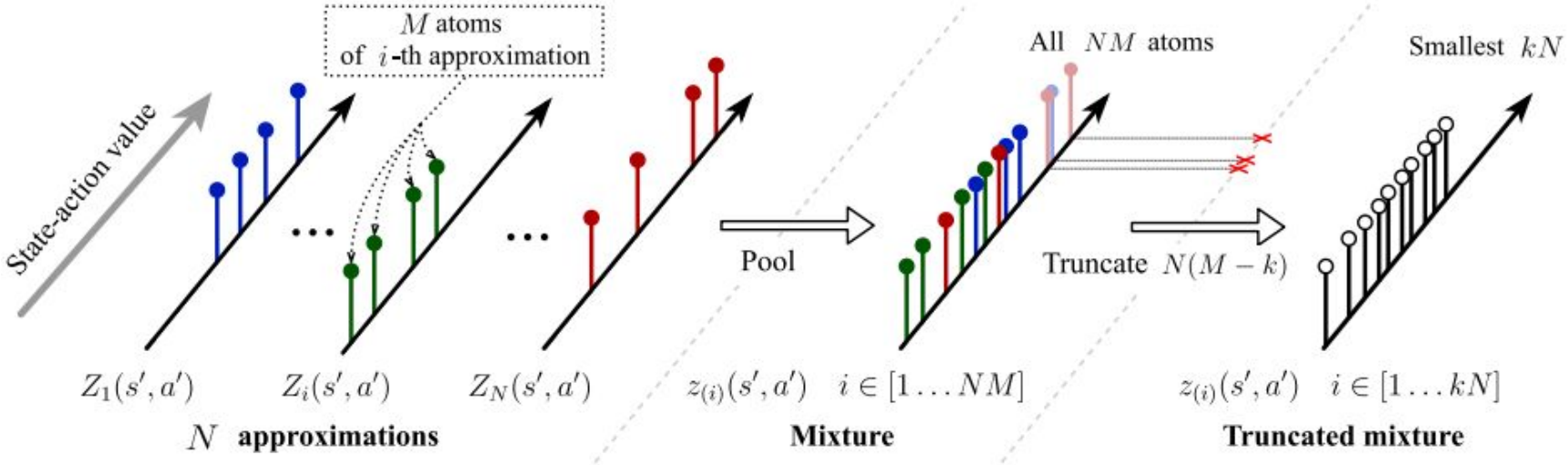


Figure 1. Evaluation on the Humanoid environment. Results are averaged over 4 seeds, ± std is shaded.

WORKING

- “Decompose” the expected return into atoms of distributional return.
- Truncate the approximation of the return distribution to control overestimation.
- Ensemble multiple distributional approximators to improve Q-value estimation.
- Non-truncated critics’ approximations for policy optimization and truncate target return distribution at value learning stage – prevents errors from propagating to other states and eases policy optimization.

WORKING (contd.)



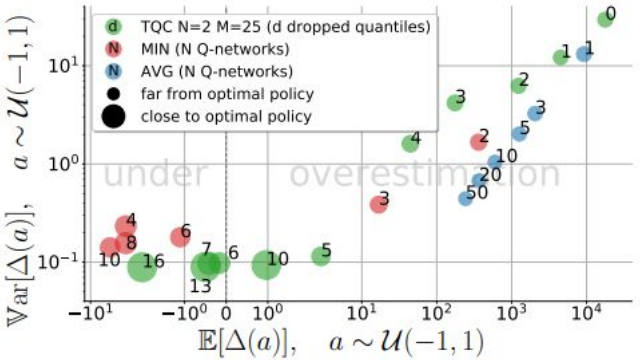
COMPARISION WITH OTHER ALGORITHMS

SINGLE STATE MDP

- Evaluate bias correction techniques in a single state continuous action infinite horizon MDP.

Table 1. Bias correction methods. For simplicity, we omit the state s_0 from all arguments.

METHOD	CRITIC TARGET $r(a) + \gamma\langle \widehat{Q} \text{ OR } \widehat{Z} \rangle(a')$	POLICY OBJECTIVE
AVG	$\widehat{Q}(\cdot) = \frac{1}{N}\sum_{i=1}^{N} Q_i(\cdot)$	$\frac{1}{N}\sum_{i=1}^{N} Q_i(a)$
MIN	$\widehat{Q}(\cdot) = \min_i Q_i(\cdot)$	$\min_i Q_i(a)$
TQC	$\widehat{Z}(\cdot) = \frac{1}{kN}\sum_{i=1}^{kN} \delta\left(z_{(i)}(\cdot)\right)$	$\frac{1}{NM}\sum_{i=1}^{NM} z_{(i)}(a)$



- TQC can achieve the lowest variance and the smallest bias of Q-function approximation among all the competitors.

COMPARATIVE EVALUATION

- Comparing with original implementations of SAC, TrulyPPO and TD3.

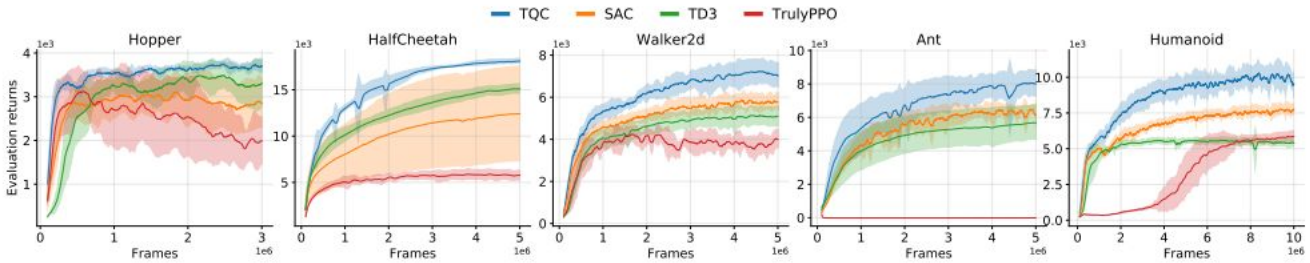
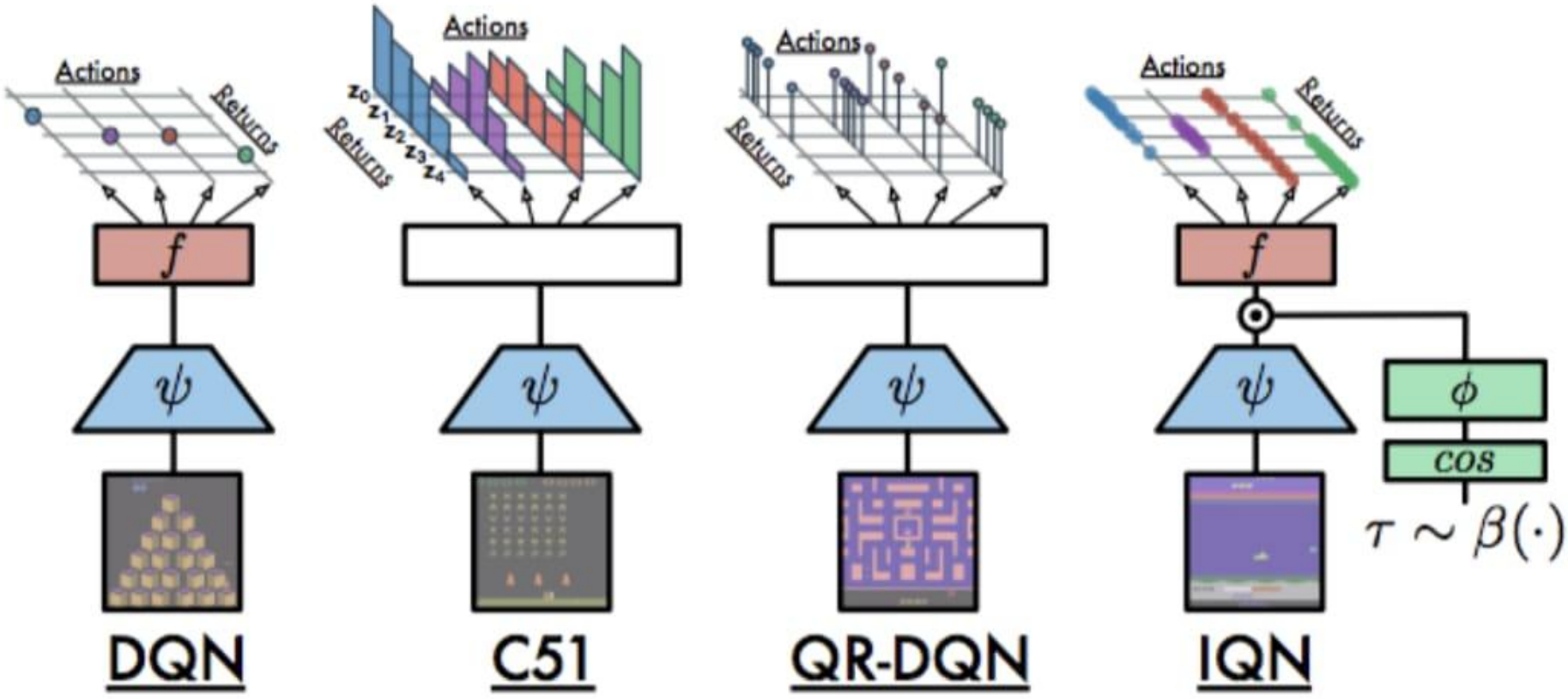


Figure 5. Average performances of methods on MuJoCo Gym Environments with ± std shaded. Smoothed with a window of 100.

- Evaluate the performance every 1000 frames as an average of 10 deterministic rollouts.
- TQC performs consistently better than any of the competitors.
- Also improves upon the maximal published score on four out of five environments

TQC – Pros and Cons

- Improvement over existing off policy algorithms like DQN, SAC, QRDQN and TD3.
- Existing work only focused on Discrete action space; Recent work focus on distributional networks like QRDQN, Implicit Quantile Network and Fully Quantile Network but they focused on their architecture alone and couldn't be used in tandem or as a direct improvement on existing off policy algorithms by improving their Q functions.
- The biggest advantage of TQC is its modularity which allows it to work with algorithms which work on both discrete and continuous action spaces.
- The most prominent disadvantage is it adds to the computational complexity of already complex algorithms.



Applications

Applications of TQC

- Applied in both model-free settings where the agent learns directly from interactions with the environment.
- Can handle both continuous and discrete state and action spaces.
- Used in off-policy settings, allowing it to leverage past experiences efficiently for learning.
- Used in effectively handling stochastic environments by modeling conditional quantiles.
- Balances exploration and exploitation effectively, adapting to uncertainty levels.
- Applied in robotics for control tasks, where continuous action spaces and real-world uncertainties are common.

Applications of TQC

- Enhancement of performance of agents in games, especially in complex and strategic game environments.
- Used in finance for risk assessment, portfolio management, and derivative pricing due to its ability to model non-Gaussian returns.
- Improvement of performance of chatbots and dialogue systems in natural language processing tasks.
- Employed in anomaly detection for identifying unusual patterns and events in various domains like network security and fraud detection.
- Helping optimize resource allocation, sensor placement, and decision-making in environmental monitoring and climate modeling by managing uncertainties.

References

References

- https://sb3-contrib.readthedocs.io/en/master/modules/tqc.html
- https://arxiv.org/abs/2005.04269
- https://openreview.net/forum? id=Bkg0u3Etwr.
- https://proceedings.mlr.press/v119/kuznetsov20a.html
- Bunea, Cornel, Theodore Charitos, Roger M. Cooke, and Günter Becker. "Two-stage Bayesian models—application to ZEDB project." Reliability Engineering & System Safety 90, no. 2-3 (2005): 123-130.
- Kuznetsov, Arsenii, Pavel Shvechikov, Alexander Grishin, and Dmitry Vetrov. "Controlling overestimation bias with truncated mixture of continuous distributional quantile critics." In International Conference on Machine Learning, pp. 5556-5566. PMLR, 2020.

ARS

AUGMENTED RANDOM SEARCH

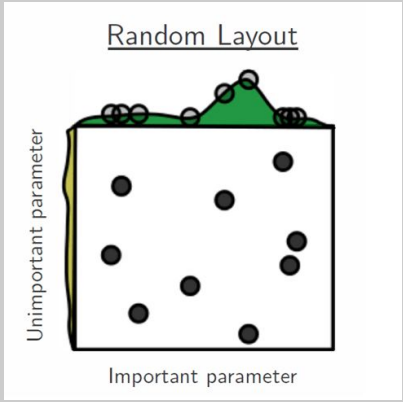
By Group 10

What is ARS?

- Augmented Random Search (ARS) is a reinforcement learning algorithm used for solving continuous control tasks.
- It is a model-free, black-box optimization technique that does not require access to the gradient of the objective function, making it suitable for problems with non-differentiable or unknown dynamics.

Basic Random Search

- Random Search is one of the simplest and oldest optimization methods for derivative-free optimization
- Random search chooses a direction uniformly at random on the sphere in parameter space, and then optimizes the function along that direction
- Basic random search simply computes a finite difference approximation along the random direction and then takes a step along this direction without using a line search



Hyperparameters: step-size α, number of directions sampled per iteration N, standard deviation of the exploration noise ν, number of top-performing directions to use b ($b < N$ is allowed only for **V1-t** and **V2-t**)

Initialize: $M_0 = \mathbf{0} \in \mathbb{R}^{p \times n}$, $\mu_0 = \mathbf{0} \in \mathbb{R}^n$, and $\Sigma_0 = \mathbf{I}_n \in \mathbb{R}^{n \times n}$, $j = 0$.

while ending condition not satisfied **do**

Sample $\delta_1, \delta_2, \ldots, \delta_N$ in $\mathbb{R}^{p \times n}$ with i.i.d. standard normal entries.

Collect $2N$ rollouts of horizon H and their corresponding rewards using the $2N$ policies

$$\textbf{V1:} \begin{cases} \pi_{j,k,+}(x) = (M_j + \nu\delta_k)x \\ \pi_{j,k,-}(x) = (M_j - \nu\delta_k)x \end{cases}$$

$$\textbf{V2:} \begin{cases} \pi_{j,k,+}(x) = (M_j + \nu\delta_k)\,\mathrm{diag}\,(\Sigma_j)^{-1/2}\,(x - \mu_j) \\ \pi_{j,k,-}(x) = (M_j - \nu\delta_k)\,\mathrm{diag}(\Sigma_j)^{-1/2}(x - \mu_j) \end{cases}$$

for $k \in \{1, 2, \ldots, N\}$.

Sort the directions δ_k by $\max\{r(\pi_{j,k,+}), r(\pi_{j,k,-})\}$, denote by $\delta_{(k)}$ the k-th largest direction, and by $\pi_{j,(k),+}$ and $\pi_{j,(k),-}$ the corresponding policies.

Make the update step:

$$M_{j+1} = M_j + \tfrac{\alpha}{b\sigma_R} \sum_{k=1}^{b} \left[r(\pi_{j,(k),+}) - r(\pi_{j,(k),-})\right] \delta_{(k)},$$

where σ_R is the standard deviation of the $2b$ rewards used in the update step.

V2 : Set μ_{j+1}, Σ_{j+1} to be the mean and covariance of the $2NH(j+1)$ states encountered from the start of training.[2]

$j \leftarrow j + 1$

end while

How does ARS work?

- Start with a Random Policy: Begin with a random way of doing tasks.
- Experiment with Actions: Try out various random actions (like movements of a robot's limbs).
- Measure Performance: See how well each action performs the task (like measuring how far the robot moves).
- Make Small Changes: Adjust the random actions slightly (perturbations).
- Repeat Exploration: Try the task again with these slightly changed actions.
- Learn from Results: Figure out which changes led to better performance.
- Update Actions: Modify future actions based on what worked well.
- Repeat and Refine: Keep repeating these steps, gradually improving the task performance through trial and error.

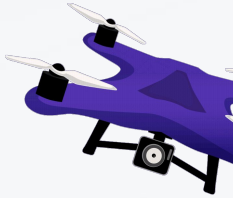
Pros of ARS (Augmented Random Search)

- Efficient Exploration of Policy Space: ARS utilizes a random search strategy to efficiently explore the policy space, allowing it to quickly sample a diverse set of policies.
- Low Data Requirements: ARS often requires fewer samples or interactions with the environment compared to gradient-based algorithms, making it more sample-efficient. This can be crucial in domains where collecting data is expensive or time-consuming.
- No Gradient Calculations: Unlike gradient-based methods, ARS does not require gradient calculations. This can be advantageous in scenarios where gradients are hard to compute or are noisy, making it a suitable choice for non-differentiable or discontinuous optimization problems.
- Potential for Discovering Novel Solutions: The random search strategy employed by ARS may lead to the discovery of unconventional or innovative policies that might not be apparent through more traditional optimization approaches

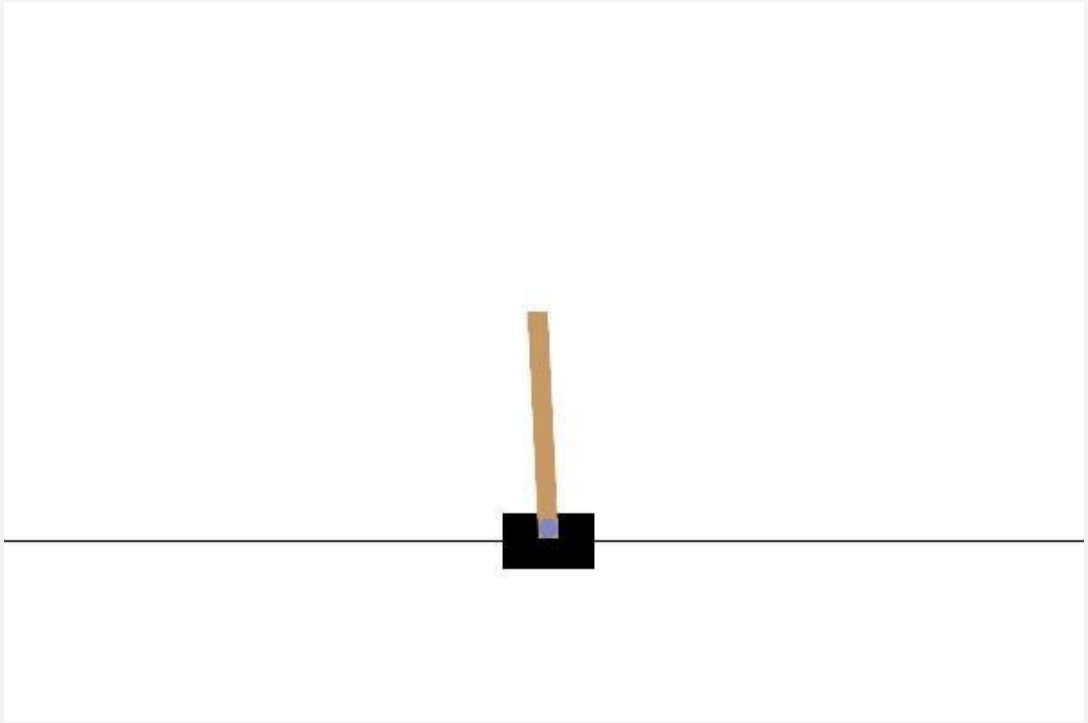
Cons of ARS (Augmented Random Search)

- Sensitivity to Hyperparameters - ARS agent to navigate a maze -> learning rate too high or too low will fail to provide results
- Limited to Policy Optimization - ARS might struggle to find a good strategy in chess whereas value-based methods like Q-learning or AlphaZero excel
- Lack of Theoretical Guarantees - In robotics, where precise control is essential, the lack of theoretical guarantees can be a drawback
- Limited Memory for Exploration - Consider an agent learning to play a complex video game where information from several time steps is essential. ARS's limited memory could hinder its ability
- Not Suitable for All Environments - In a robotics task involving stacking objects in a specific order, ARS might struggle because it lacks the capacity for learning hierarchical representations, which is better suited to hierarchical reinforcement learning methods.

Applications



- Environment monitoring in drones.
- Decision-making in autonomous vehicles in dynamic and complex environments.
- Recommendation systems to generate personalised recommendations.
- Training agents in video games.
- Optimizing the control of manufacturing and other automation processes in the industries.



References

- Simple random search provides a competitive approach to reinforcement learning
- https://github.com/AnshumaanDash/Augmented-Random-Search
- https://towardsdatascience.com/introduction-to-augmented-random-search-d8d7b55309bd
- https://www.sciencedirect.com/science/article/abs/pii/S0959652622042482
- https://iq.opengenus.org/augmented-random-search/

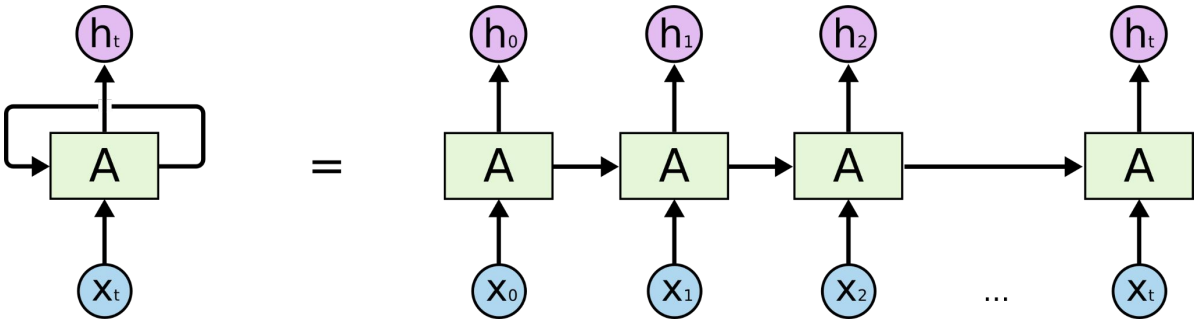
THANK YOU

Recurrent PPO

Team - 11

Recurrent PPO: LSTM

- Agent model now has LSTM module that feeds its hidden state to the (actor/critic) modules
- Meant to incorporate memory of state information



Recurrent PPO: Training Modifications

- Careful calculation of advantage estimator
- We can only use policy gradient were samples are sequentially sampled Not random

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right]$$

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

Advantage Estimate for Recurrent Networks

$$\hat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T)$$

Key Advantages of Recurrent PPO

Sample Efficiency:

1. While on-policy methods may require more samples, in robotics, ensuring safety and effectiveness outweighs sample efficiency concerns.

Overcoming Model Mismatch:

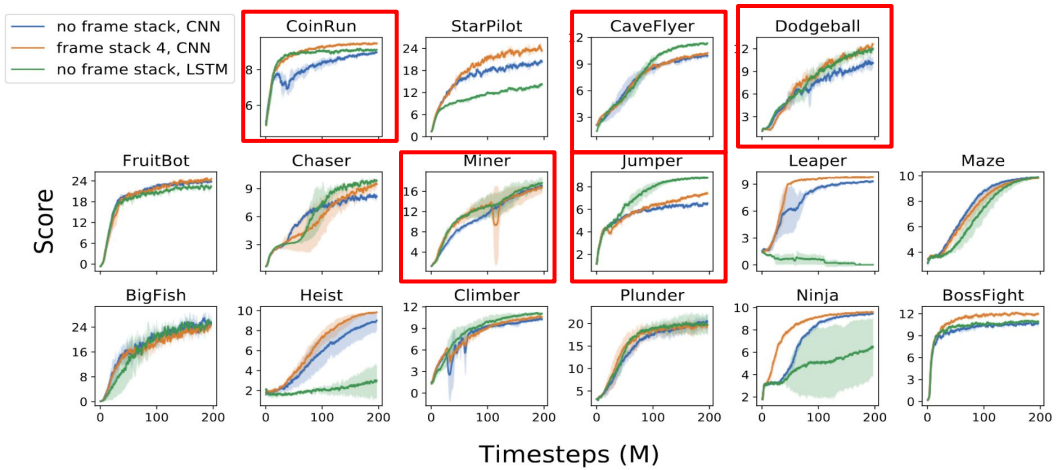
1. Real-world environments often deviate from simulations. Recurrent PPO excels in handling these discrepancies, learning directly from real-world experiences.

Considerations:

1. Continuous State/Action Spaces: Well-suited for tasks with continuous state and action spaces.
2. Safety-Focused Environments: Ideal for environments where safety and stability are paramount.

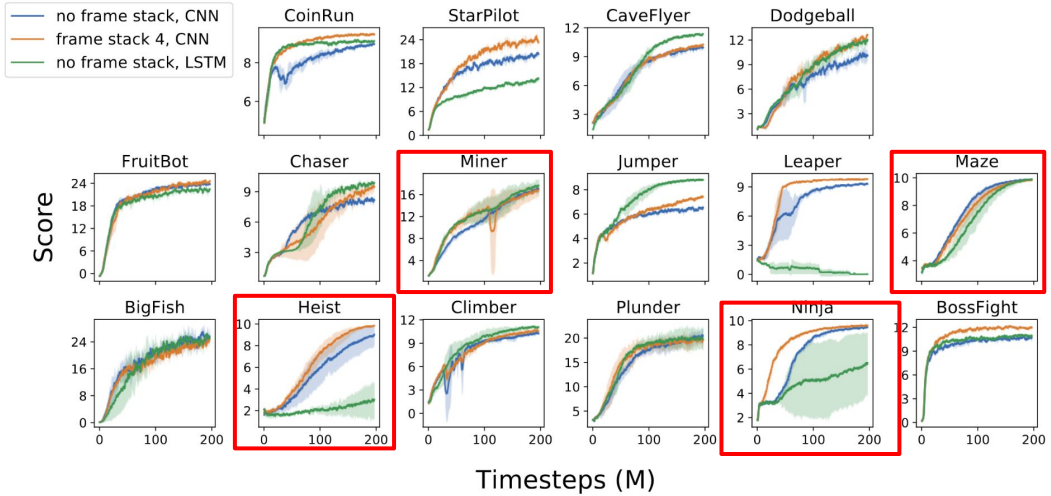
Recurrent PPO: Pros

- The agent may maintain and update an internal memory or context of previous.
- Enhances learning in scenarios where states are only partially observable or rewards are delayed.
- Recurrent PPO is well-suited for large-scale training because, when implemented effectively, it can be parallelized and deployed across multiple processors or distributed computing systems to take advantage of their combined processing power.



Recurrent PPO: Cons

- It can take a lot of time and computing power to train PPO agents for difficult tasks or in high-dimensional state spaces.
- Sensitive to the selection of hyperparameters.
- The use of LSTM in the baseline is notably unstable.



Applications

- Extension of PPO for tasks with sequential or time-dependent data.
- Ideal for scenarios where current actions effectiveness is influenced by past actions and observations.

Few Applications :

- Robotics control tasks involving sequences (e.g., robotic hand handling items)
- Games demanding memory from past actions (e.g., card games, DOTA 2)

Why Recurrent PPO in Robotics?

- Safety and Stability: Reduces the risk of dangerous or costly mistakes, crucial in safety-critical applications.
- Adaptability to Dynamics: Well-suited for complex and unpredictable physical environments, eliminating the need for precise modeling.
- Continuous Learning: Allows for seamless adaptation to varying task requirements.

References

1. Huang, et al., "The 37 Implementation Details of Proximal Policy Optimization", ICLR Blog Track, 2022.
2. Cobbe, Karl, et al. "Leveraging Procedural Generation to Benchmark Reinforcement Learning." *ArXiv*, 2019, /abs/1912.01588. Accessed 13 Oct. 2023.

Q & A

Dueling Deep Q Network Algorithm

Group 12

Introduction:
Dueling DQN

- Dueling DQN (Deep Q-Network) is an advanced variant of the traditional DQN in Reinforcement Learning (RL).

- **Key Components**
  Dueling DQN introduces a novel architecture, splitting the Q-value into two components:

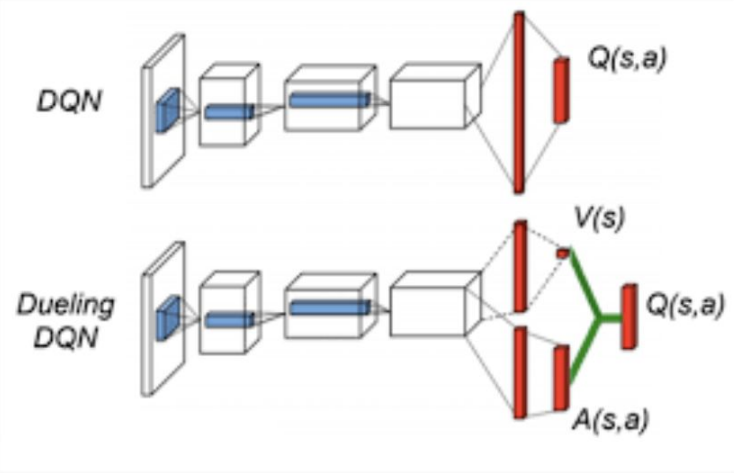
  - *Value Function (V(s)):* Represents the expected cumulative future rewards from a given state, irrespective of the chosen action.

  - *Advantage Function (A(s,a)):* Captures the advantage of taking a specific action in a given state compared to the average action in that state.

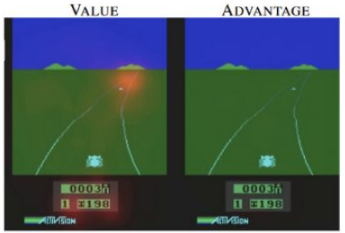
- **Key Innovation**
  - Dueling DQN decouples value and advantage functions, enabling more efficient action-value estimation in RL.

Dueling DQN Architecture: Motivation

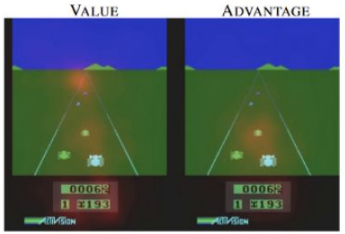
- **Motivation**
  - For many states, estimation of state value is more important, comparing with state-action value.
  - Better approximate state value, and leverage power of advantage function



Focus on 2 things:
- The horizon where new cars appear
- On the score



No car in front, **does not pay much attention because action choice making is not relevant**



Pays attention to the front car, in this case **choice making is crucial to survive**

Working of Dueling DQN

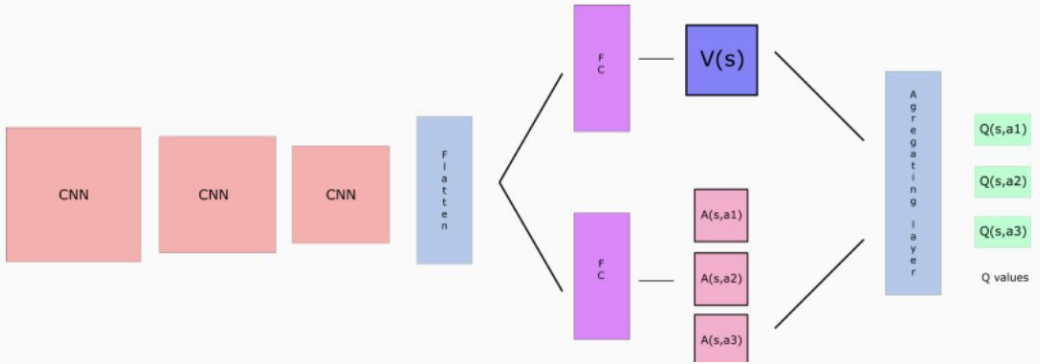
- Dueling DQN works by decomposing the Q-function into two separate functions: the state-value function V(s) and the action-advantage function A(s,a).
- The state-value function estimates the expected reward for being in a given state, regardless of the action taken.
- The action-advantage function estimates how much better it is to take a given action over the average action.
- The following mathematical equation describes the working of Dueling DQN:

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

where,

  - *Q(s,a)*: Q-value for state *s* and action *a*
  - *V(s)*: state-value function for state s
  - *A(s,a)*: action-advantage function

- The V(s) is estimated using CNN with single output, whereas the A(s,a) is estimated using CNN with multiple outputs, one for each action.



Working of Dueling DQN

- **Value Aggregation**
  - The state-value V(s) is not directly used to make action decisions; Instead it is aggregated to help compute the Q-values. Typically, the average of advantage values for all actions is added to V(s). Hence,

$$Q(s,a) = V(s) + A(a,s) - Avg[A(a',s)]$$

- **Decision-Making**
  - To make decisions, the agent selects actions based on the computed Q-values. The action with the highest Q-value in a given state is chosen as the optimal action.

- **Training**
  - Dueling DQN uses Experience Replay and Target Network to train the neural network efficiently.
  - *Loss function:* The mean squared error between the predicted Q-values and the target Q-values.

$$Loss = E[(Q(s, a) - (r + \gamma * \max(Q(s', a'))))^2]$$

where,

r = reward received for taking action a

γ = discount factor

$Q'(s', a')$ = Predicted Q-value for the next state s' and action a'

Dueling DQN Algorithm

1. Initialize two neural networks, one for the state-value function and one for the action-advantage function.
2. Initialize a replay memory.
3. At each time step:
    - Take an action a in the current state s.
    - Observe the next state s' and the reward r.
    - Store the transition (s, a, r, s') in the replay memory.
    - Sample a batch of transitions from the replay memory.
    - Calculate the target Q-values for the sampled transitions using the Bellman equation.
    - Update the state-value network and the action-advantage network using the loss function.
4. Repeat step 3 until the agent reaches convergence.

Pros & Cons

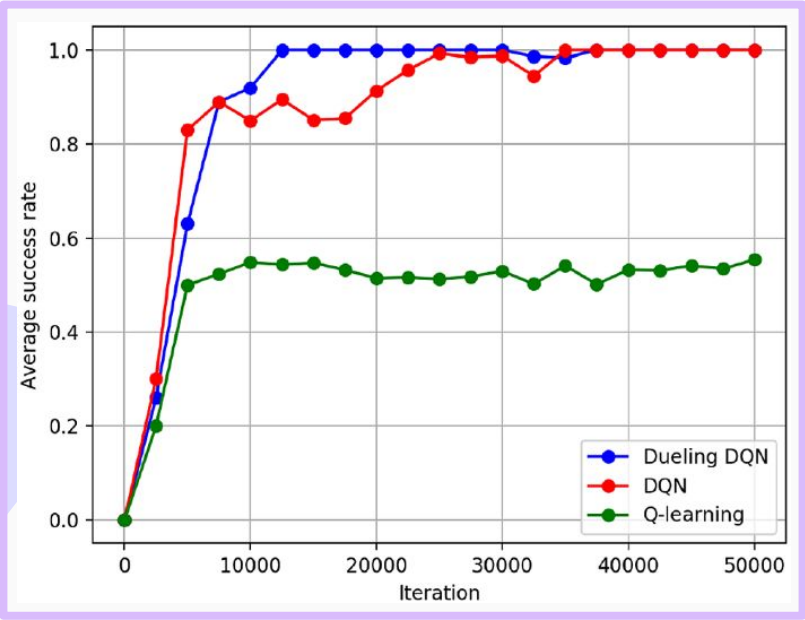
Pros

- Improved performance
- Efficient value estimation (prevents overestimation)
- Better handling of large action spaces
- Improved learning stability
- More robust in complex environments

Cons

- Computationally intensive
- Hyperparameter tuning
- Not as well suited for continuous action spaces
- Data efficiency
- Risk of overfitting

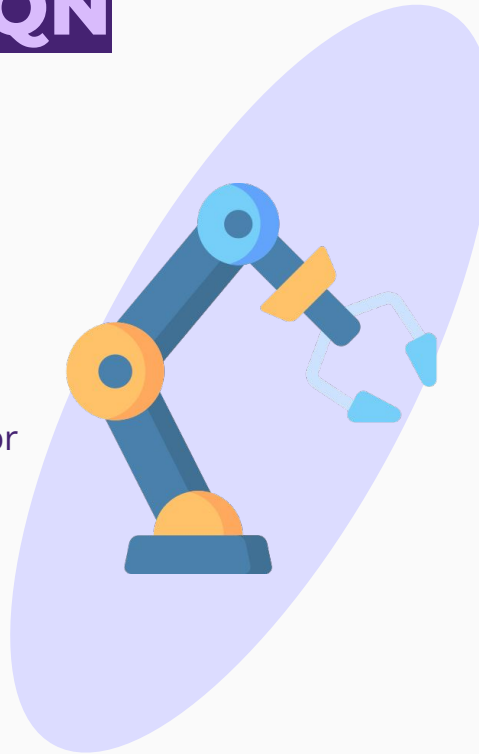
Properties & Requirements of Dueling DQN



- **Properties**
  - Model-free learning
  - Off-policy learning
  - Discrete state/action spaces
- **Requirements**
  - Deep neural network
  - Experience replay
  - Target network

Applications of Dueling DQN

- Training robots to navigate in complex environments
- Controlling robotic arms or grippers to manipulate objects with precision
- Managing Investment portfolios by selecting best actions (buy, sell, hold)
- Personalized treatment planning for patients by optimizing drug dosages or therapy schedules
- Optimizing ad placement for maximum user engagement
- Atari games & board games such as chess, go, or other strategic games where the algorithm can learn optimal strategies



References

- Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., & de Freitas, N. (2015). Dueling network architectures for deep reinforcement learning.
- Van Hasselt, H., Guez, A., & Silver, D. (2015). Deep reinforcement learning with double Q-learning.
- Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Graves, A., Riedmiller, M., Heess, N., Ostrovski, G., Dahl, G. E., & others (2015). Human-level control through deep reinforcement learning.

Thank You!

CSE 574 : Planning & Learning Methods in AI (Fall 2023)

Advantage Actor Critic (A2C)

Group13

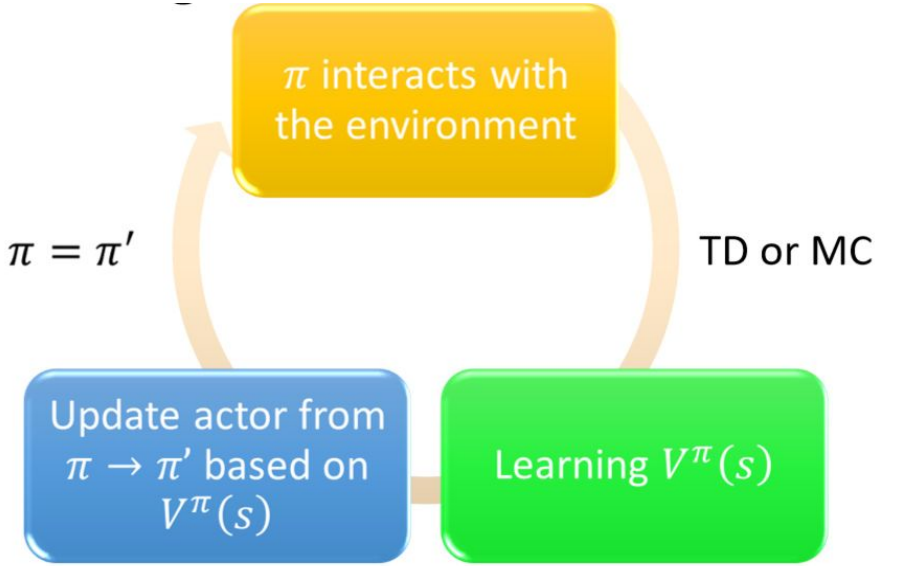
Supervisor : Dr. Ransalu Senanayake

Introduction

- Advantage Actor Critic (A2C) is a policy gradient RL algorithm which uses the combination of policy-based and value-based methods to help agent to learn an optimal policy based on the environment.
- It comprises of three key components:
  - **Actor -** Controls the behaviour of the agent i.e. helps agent in deciding next action based on current state.
  - **Critic -** Judges the actor's decisions and identifies how good the action taken by actor is.
  - **Advantage -** Helps in quantifying the correctness of the decision taken by actor based on critic's prediction.

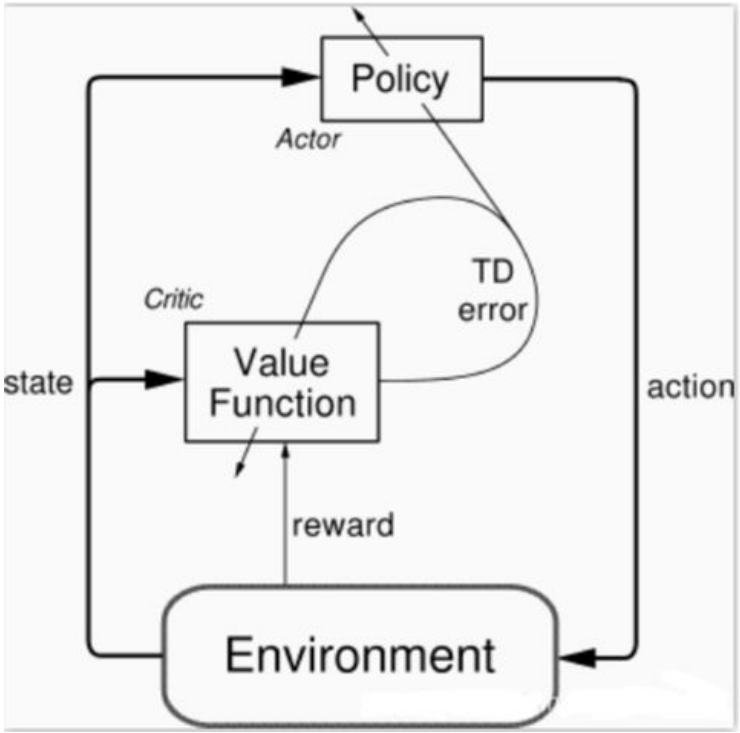
How it works ?

A2C Architecture



$$\nabla \bar{R}_{\theta} \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T_n} \left(r_t^n + V^{\pi}(s_{t+1}^n) - V^{\pi}(s_t^n) \right) \nabla log p_{\theta}(a_t^n | s_t^n)$$

Advantage A (s_t, a_t)



Advantage Function

- This function helps in quantifying how much better or worse the action (a_t) is when compared to critic's estimate for a given state (s_t).

$$Q^{\pi}(s_t^n, a_t^n) - V^{\pi}(s_t^n)$$

$$r_t^n + V^{\pi}(s_{t+1}^n) - V^{\pi}(s_t^n)$$

Estimate two networks? We can only estimate one.

Only estimate state value

A little bit variance

$$Q^{\pi}(s_t^n, a_t^n) = E[r_t^n + V^{\pi}(s_{t+1}^n)]$$

$$Q^{\pi}(s_t^n, a_t^n) = r_t^n + V^{\pi}(s_{t+1}^n)$$

Actor and Critic Network

Actor Network:

- It is a policy network which helps agent decide next action based on current state.

$$a_t = \pi_\Theta (s_t)$$

- The actor network is updated using the policy gradient method.

$$\nabla_\Theta J(\Theta) = E[\nabla_\Theta[\log \pi_\Theta (a_t|s_t) A(s_t, a_t)]]$$

- To favor exploration, A2C also uses entropy term to the gradient calculation.

$$\nabla_\Theta J(\Theta) = E[\nabla_\Theta[\log \pi_\Theta (a_t|s_t) A(s_t, a_t) - \beta H(\pi_\Theta (s_t))]]$$

Critic Network:

- It is a value network which helps in judging the correctness of the action taken by the actor network based on current state.

$$\text{Expected return} = Q (s_t, a_t)$$

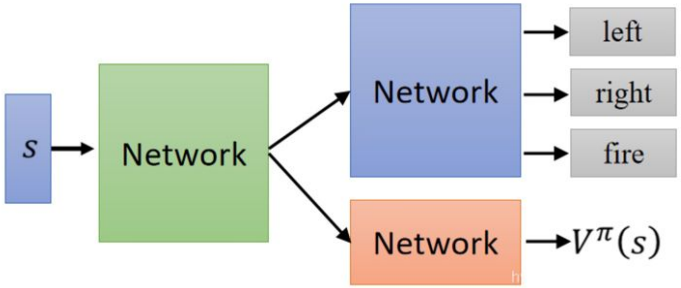
- The critic network is updated using critic loss to improve the expected return for the given state.

$$\Delta\omega = \alpha\nabla_\omega(r_{t+1} + \gamma V (s_{t+1}) - V (s_t))$$

Pros and Cons

Pros:

- Only need to estimate V function, don't need to estimate Q function.
- The parameters of actor $\pi(s)$ and critic $V^{\pi}(s)$ can be shared.



Cons:

- A2C might require a large number of samples to learn effectively. This makes it slow to converge and sensitive to exploration strategies.
- A2C is based on gradient descent. The direction of the steepest improvement is prone to rapid and unpredictable changes depending on the actor, the critic, and the environment which can lead to oscillations, divergence, or poor performance.
- A2C can struggle to generalize and transfer its knowledge to new situations.
- A2C is a black-box algorithm meaning it does not provide any explicit or intuitive rationale for its actions or values.
- A2C is a goal-oriented algorithm, which means it tries to maximize its reward, regardless of the potential side effects or trade-offs. This can raise ethical and social concerns.

Properties and Use Cases

Properties:

- It uses stochastic policies and can't use recurrent policies.
- It can work with both discrete and continuous action spaces.
- It uses multiple workers to avoid the use of replay buffer.
- It is designed to work primarily on CPU.
- It is a on-policy, model-free RL algorithm.
- It uses N-step update.

Use Cases:

- Robotics : Used to train Robots to perform complex tasks, such as grasping and manipulation.
- NLP : Used to train chatbots to interact with humans in a more natural and intuitive way.
- Gaming: Used to train game agents (eg: chess, go).
- Finance: Used to develop trading algos that can learn from market data and adjust their strategies based on changing market conditions.

References

References

1. Mnih, Volodymyr, et al. "Asynchronous methods for deep reinforcement learning." International conference on machine learning. PMLR, 2016.
2. Haarnoja, Tuomas, et al. "Soft actor-critic algorithms and applications." arXiv preprint arXiv:1812.05905 (2018).

Any Questions?

Group 14

Asynchronous Advantage Actor Critic(A3C)

Background

Reinforcement Learning Methods

Value Based methods: map each state-action pair to a value, which represents the expected cumulative reward the agent can obtain by following a specific action-selection strategy.

Policy Based methods: directly optimize the policy without a value function. The goal is to maximize the performance of the parameterized policy using gradient ascent.

A2C (Advantage Actor-Critic)

It a synchronous RL algorithm in which multiple agents run in parallel to collect experiences from the environment.

Drawbacks:

- High variance due to reliance on policy gradients which can lead to slow and unstable learning.
- Sensitivity to hyperparameters.

DQN (Deep Q-Network)

It is a deep reinforcement learning algorithm that focuses on estimating the action-value function (Q-function).

Drawbacks:

- Lack of handling continuous action spaces.
- Sample inefficiency.

How does A3C work?

A3C stands for **Asynchronous Advantage Actor Critic**

Actor - Critic

It stands for two neural networks — Actor and Critic

The goal of the Actor is in optimizing the policy (***"How to act?"***), and the Critic aims at optimizing the value ***("How good action is?")***

- The actor takes as input the state and outputs the best action. It essentially controls how the agent behaves by **learning the optimal policy** (policy-based).

- The critic, on the other hand, **evaluates the action by computing the value function** (value based).

What is Advantage?

Q values can, in fact, be decomposed into two pieces: **the state Value function V(s)** and the **Advantage value A(s, a)**:

$$Q(s,a)=V(s)+A(s,a) \quad \Rightarrow \quad \mathbf{A(s,a)=Q(s,a)-V(s)} \quad \Rightarrow \quad A(s,a)=r+\gamma V(s')-V(s)$$

Advantage function captures **how better an action is** compared to the others at a given state, while as we know the value function captures **how good it is** to be at this state.

“Asynchronous”

The key difference from A2C and DQN is the Asynchronous part.

In A3C we have a global network with multiple agents having their own set of parameters.

It consists of **multiple independent agents**(networks) with their own weights, who interact with a different copy of the environment in parallel.

It relies on parallel actor-learners and accumulated updates for improving training stability.

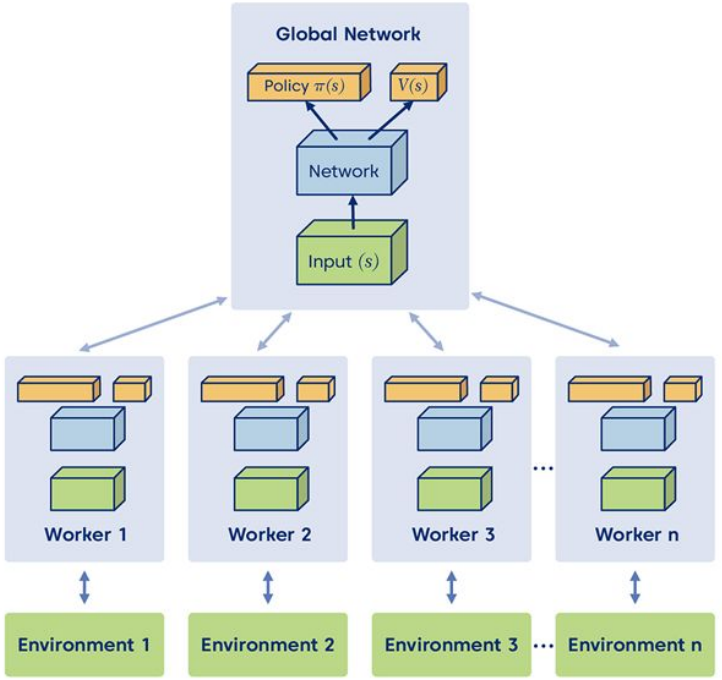


Diagram of A3C high-level architecture.

Important to note!

The updates are **not happening simultaneously** and that's where the asynchronous part comes from.

The policy and the value function are **updated after every tmax actions** or when a terminal state is reached.

After each update, the **agents resets their parameters** to those of the global network and continue their **independent exploration** and training for n steps until they update themselves again. Therefore, the flow of information exists between the agents themselves through the global network.

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ *and* θ_v *and global shared counter* $T = 0$
// Assume thread-specific parameter vectors θ' *and* θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t+1$
    $T \leftarrow T+1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$ $\beta \nabla_{\theta'} H(\pi(s_t;\theta'))$,
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i;\theta'_v))^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Method	Training Time	Mean	Median
DQN	8 days on GPU	121.9%	47.5%
Gorila	4 days, 100 machines	215.2%	71.3%
D-DQN	8 days on GPU	332.9%	110.9%
Dueling D-DQN	8 days on GPU	343.8%	117.1%
Prioritized DQN	8 days on GPU	463.6%	127.6%
A3C, FF	1 day on CPU	344.1%	68.2%
A3C, FF	4 days on CPU	496.8%	116.6%
A3C, LSTM	4 days on CPU	623.0%	112.6%

Table 1. Mean and median human-normalized scores on 57 Atari games using the human starts evaluation metric. Supplementary Table SS3 shows the raw scores for all games.

Properties and Requirements

- **Environment**: Needs an interactive model.
- **Multiple Agents**: A3C uses parallel agents in multiple environment instances.
- **Policy and Value Function**: A3C keeps a policy and value function estimate, needing function approximators like neural networks.
- **Computation Resources**: A3C is suitable for parallel computation on multi-core CPUs and can be GPU-implemented.
- **Reward Signal**: A3C needs a clear reward signal from the environment for learning.

Pros and Cons of A3C

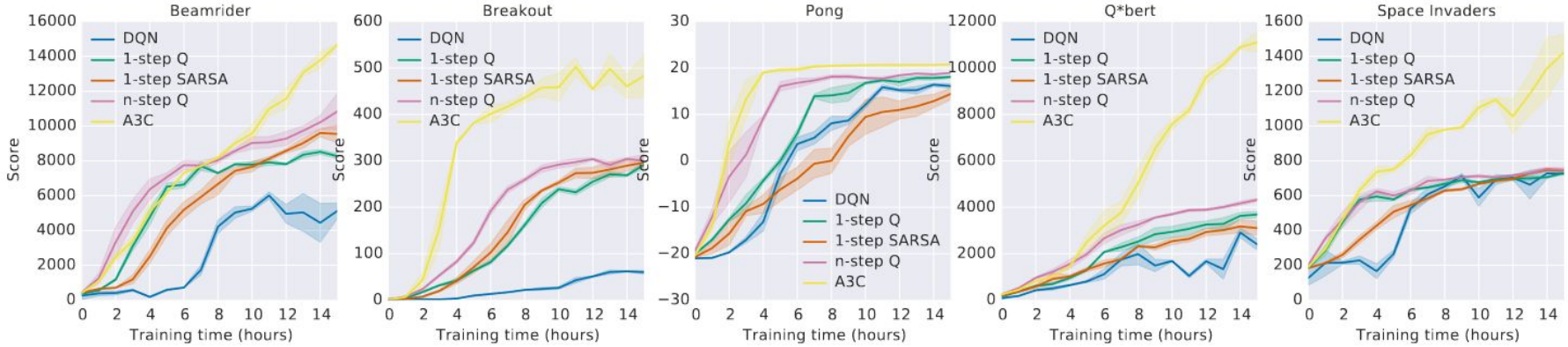


Figure 1. Learning speed comparison for DQN and the new asynchronous algorithms on five Atari 2600 games. DQN was trained on a single Nvidia K40 GPU while the asynchronous methods were trained using 16 CPU cores. The plots are averaged over 5 runs. In the case of DQN the runs were for different seeds with fixed hyperparameters. For asynchronous methods we average over the best 5 models from 50 experiments with learning rates sampled from $LogUniform(10^{-4}, 10^{-2})$ and all other hyperparameters fixed.

Pros and Cons of A3C

PROS:

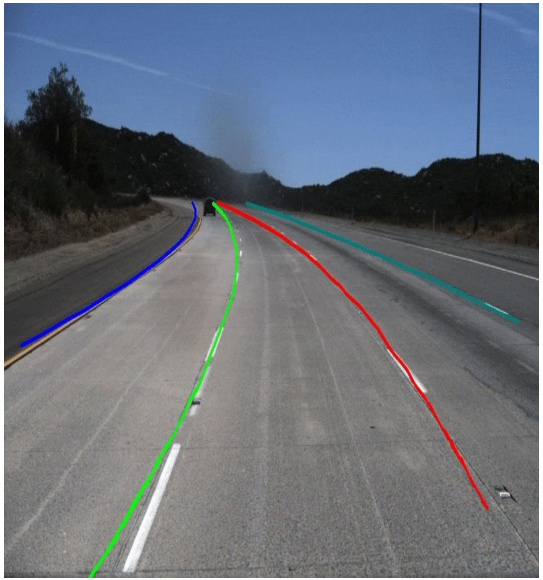
- Faster, Simpler and more robust in standard RL tasks as compared to policy gradients and DQN
- The A3C agent learns the achievement of higher scores, making learning process better.
- A3C consists of multiple independent agents(networks) with their own weights, who interact with a different copy of the environment in parallel. Thus, they can explore a bigger part of the state-action space in much less time than A2C.

CONS:

- The main drawback of asynchrony is that some agents will be playing with an older version of the parameters.

Applications of A3C

Game Playing
Achieved Superhuman performance



Real world Autonomous Driving, lane following, decision-making at intersections

Robotic Controls
Especially in complex and dynamic environments.

Applications of A3C

Being well-suited for environments with a large state or action space, A3C can be used to perform the following tasks:

- **Recommendation systems:** Optimize content recommendations to users. It can help personalize recommendations based on user behavior and preferences.

- **Natural language processing:** Train NLP models to generate text, translate languages, or perform dialogue generation tasks.

- **Resource Management**: Applications involving resource allocation and management, such as data center optimization, dynamic pricing, or energy management.

- **Healthcare:** Used for medical image analysis, drug discovery, and optimizing treatment plans. It can help in making predictions and recommendations in healthcare settings.

References

- Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D. and Kavukcuoglu, K., 2016, June. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning* (pp. 1928-1937). PMLR.
- https://theaisummer.com/Actor_critics/
- https://paperswithcode.com/method/a3c
- https://medium.com/sciforce/reinforcement-learning-and-asynchronous-actor-critic-agent-a3c-algorithm-explained-f0f3146a14ab

Thank You

GENERATIVE ADVERSARIAL IMITATION LEARNING (GAIL)

Presented by **Group-15**:

Ho, Jonathan, and Stefano Ermon. "Generative adversarial imitation learning." Advances in neural information processing systems 29 (2016)- 2770 citations

Reinforcement Learning

Reinforcement Learning

Reward

RL:

Reward Function: A function could provide feedbacks on action taking (resource of improvement)

RL needs Reward functions, hard to get in realistic scenarios

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad 0 \leq \gamma \leq 1$$

F(s) ⇒ R

RL:

Reward Function: A function could provide feedbacks on action taking (resource of improvement)

RL needs Reward functions, hard to get in realistic scenarios

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad 0 \leq \gamma \leq 1$$

IRL(Inverse):

How to overcome the "headache" of not able to explicitly define the reward function ?

⇒ Let's **infer the reward function** from **expert data**, and then based on that reward function, optimize policy

RL:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right], \text{ for all } s \in \mathcal{S}$$

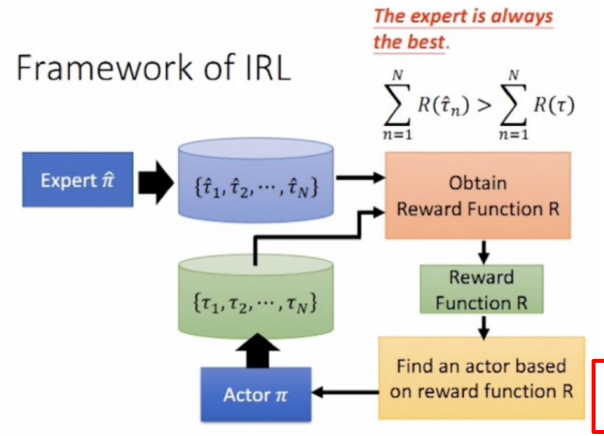
$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right]$$

Model based/Model free

Value Iteration / Policy Gradient

IRL(Inverse):

Let's infer the reward function from **expert data**, and then based on that reward function, optimize policy



MaxEnt IRL $\underset{c \in \mathcal{C}}{\text{maximize}} \left(\min_{\pi \in \Pi} -H(\pi) + \mathbb{E}_\pi[c(s,a)] \right) - \mathbb{E}_{\pi_E}[c(s,a)]$

$c(s,a) = -r(s,a)$

Linear IRL $r(s) = w \cdot \phi(s)$

Learning a reward function is already Challenging, then learn RL on it: **Expensive**

IRL(Inverse):

Let's infer the reward function from **expert data,** and then based on that reward function, optimize policy

MaxEnt IRL

$$\underset{c\in\mathcal{C}}{\text{maximize}} \left(\min_{\pi\in\Pi} -H(\pi) + \mathbb{E}_{\pi}[c(s,a)] \right) - \mathbb{E}_{\pi_E}[c(s,a)] \quad \rightarrow \quad c(s,a) = -r(s,a)$$

$H(\pi) \triangleq \mathbb{E}_{\pi}[-\log \pi(a|s)]$ is the γ-discounted causal entropy

Linear IRL

reward function is assumed to be a linear combination of known features

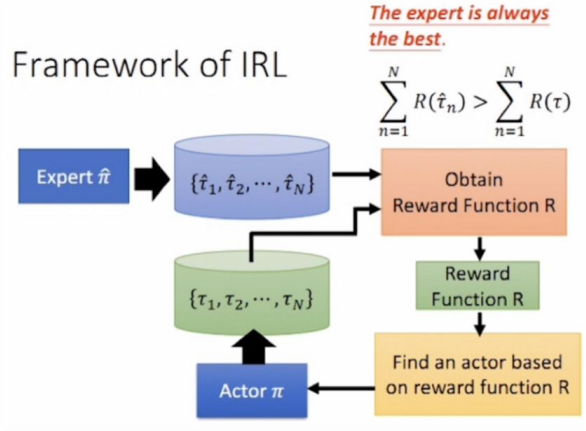
$$r(s) = w \cdot \phi(s)$$

w : weight vector to be learned

$\phi(s)$: feature vector represented from state s

IRL(Inverse):

Let's infer the reward function from **expert data**, and then based on that reward function, optimize policy



MaxEnt IRL $\underset{c \in \mathcal{C}}{\text{maximize}} \left(\min_{\pi \in \Pi} -H(\pi) + \mathbb{E}_{\pi}[c(s,a)] \right) - \mathbb{E}_{\pi_E}[c(s,a)]$

Linear IRL $r(s) = w \cdot \phi(s)$

Learning a reward function is already Challenging, then learn RL on it: **Expensive**

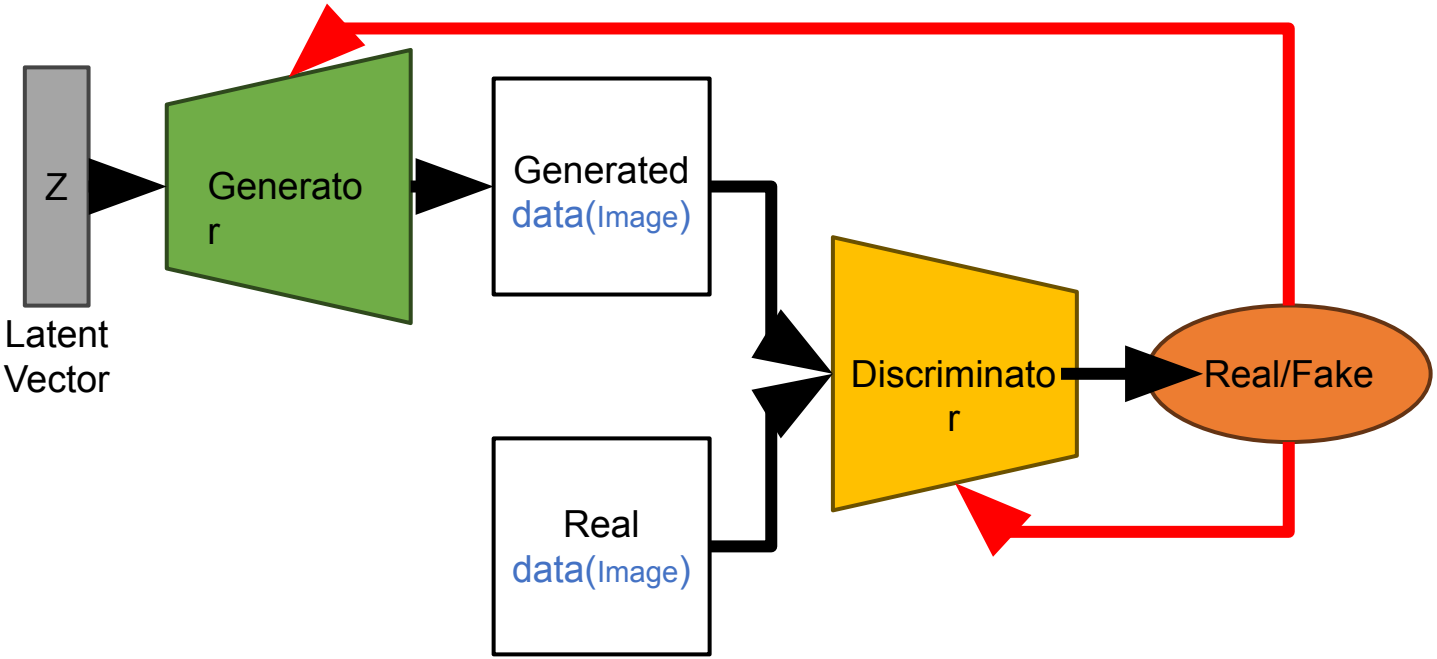
GAIL:

Instead of explicitly recovering the reward function:

A **discriminator** differentiates between the expert's trajectories and the trajectories produced by the current policy (**Generator**)

GAIL From Generative Adversarial Network

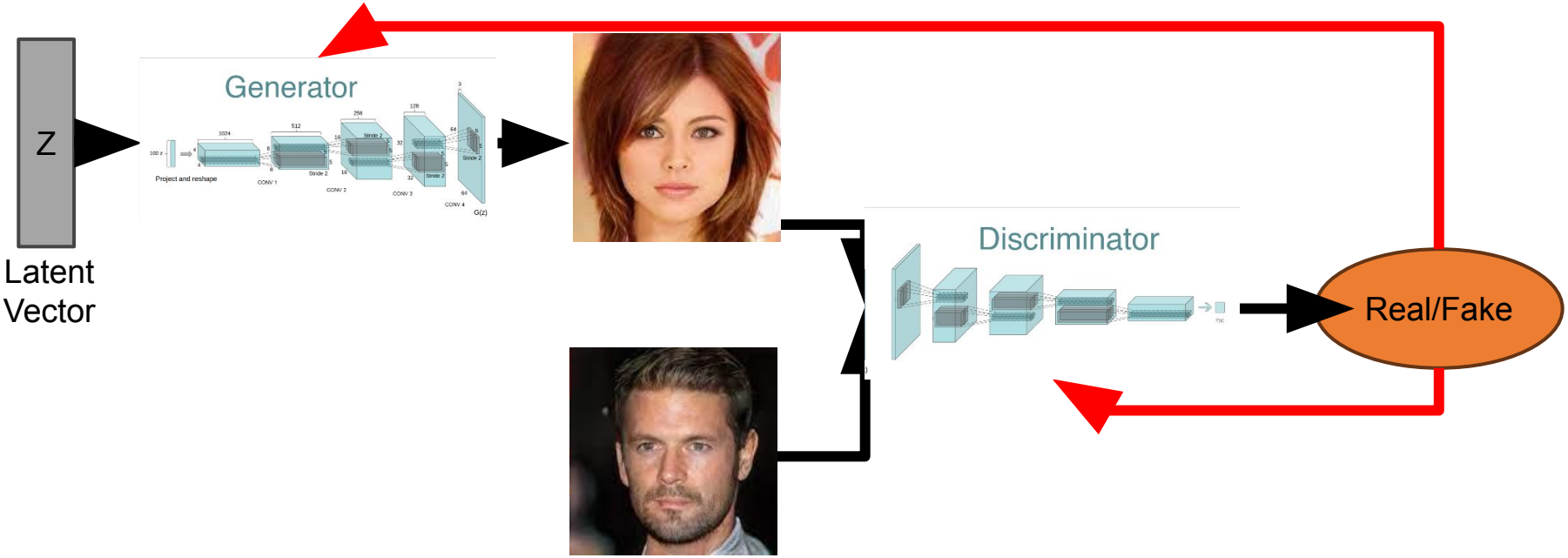
Generative Adversarial Networks (GAN):



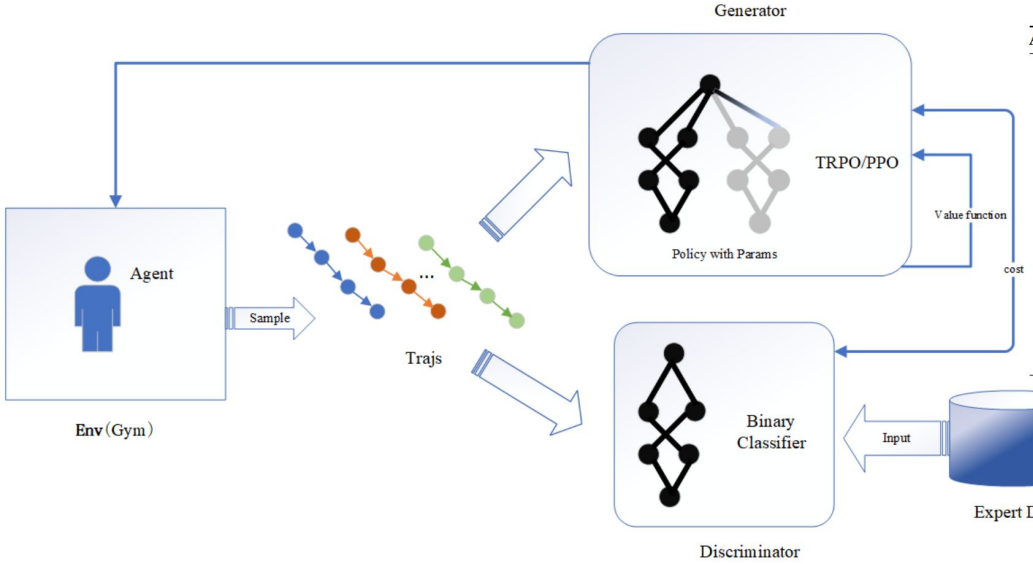
$$J(G,D) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

GAIL From Generative Adversarial Network

Generative Adversarial Networks (GAN):



GAIL: Architecture



Algorithm 1 Generative adversarial imitation learning

1: **Input:** Expert trajectories $\tau_E \sim \pi_E$, initial policy and discriminator parameters θ_0, w_0
2: **for** $i = 0, 1, 2, \ldots$ **do**
3: Sample trajectories $\tau_i \sim \pi_{\theta_i}$
4: Update the discriminator parameters from w_i to w_{i+1} with the gradient

$$\hat{\mathbb{E}}_{\tau_i}[\nabla_w \log(D_w(s,a))] + \hat{\mathbb{E}}_{\tau_E}[\nabla_w \log(1 - D_w(s,a))] \quad (17)$$

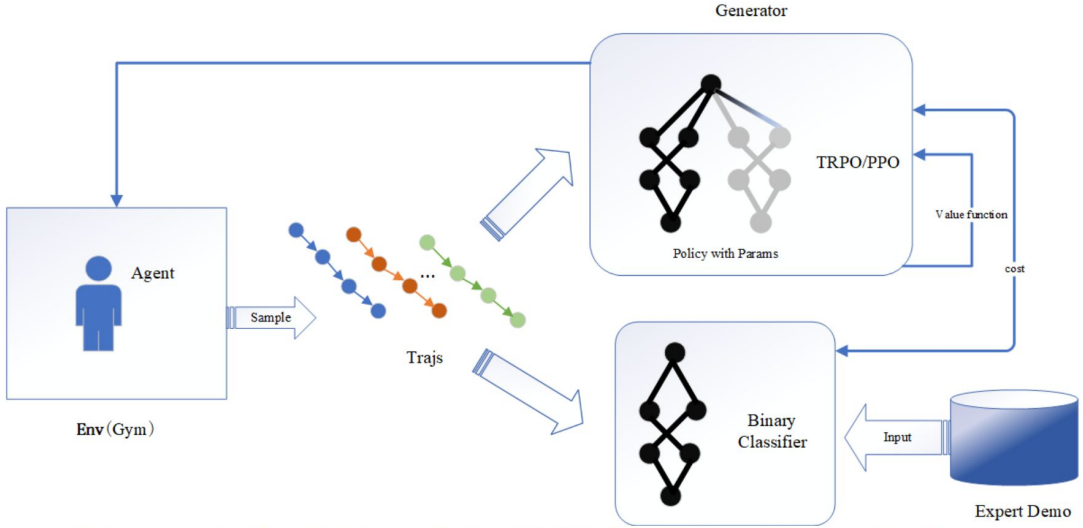
5: Take a policy step from θ_i to θ_{i+1}, using the TRPO rule with cost function $\log(D_{w_{i+1}}(s,a))$. Specifically, take a KL-constrained natural gradient step with

$$\hat{\mathbb{E}}_{\tau_i}[\nabla_\theta \log \pi_\theta(a|s) Q(s,a)] - \lambda \nabla_\theta H(\pi_\theta), \quad (18)$$
$$\text{where } Q(\bar{s},\bar{a}) = \hat{\mathbb{E}}_{\tau_i}[\log(D_{w_{i+1}}(s,a)) \mid s_0 = \bar{s}, a_0 = \bar{a}]$$

6: **end for**

- GAIL learns a policy by simultaneously training it with a discriminator that aims to distinguish expert trajectories against trajectories from the learned policy.

GAIL: Without Reward How does it use TRPO/PPO?



Take a policy step from θ_i to θ_{i+1}, using the TRPO rule with cost function $\log(D_{w_{i+1}}(s,a))$. Specifically, take a KL-constrained natural gradient step with

$$\hat{\mathbb{E}}_{\tau_i}\left[\nabla_\theta \log \pi_\theta(a|s) Q(s,a)\right] - \lambda \nabla_\theta H(\pi_\theta),$$
$$\text{where } Q(\bar{s},\bar{a}) = \hat{\mathbb{E}}_{\tau_i}\left[\log(D_{w_{i+1}}(s,a)) \,|\, s_0 = \bar{s}, a_0 = \bar{a}\right] \tag{18}$$

1. It take the Discriminator score as the collected reward
2. Then apply Policy Gradient method to optimize Policy (Generator)

GAIL: Pros and Cons

- Pros:
  - •Get rid of explicitly designing reward function
  - •Directly extract a policy from data
  - •Model-free imitation learning algorithm
  - •Sample efficient in terms of expert data
- Cons:
  - •Sample inefficient in terms of environment interaction – on par with TRPO
  - •Assuming similar problems of GANs:
    - Mode collapse
    - Hard to converge
  - •Require Expert Dataset

GAIL: Requirements

- Requirements:

  - Problem Modeling: MDP

$$M =< S, A, P, R, \gamma > , \quad s_t \in S, a_t \in A, P^a_{ss'} = P(s'|s, a)$$

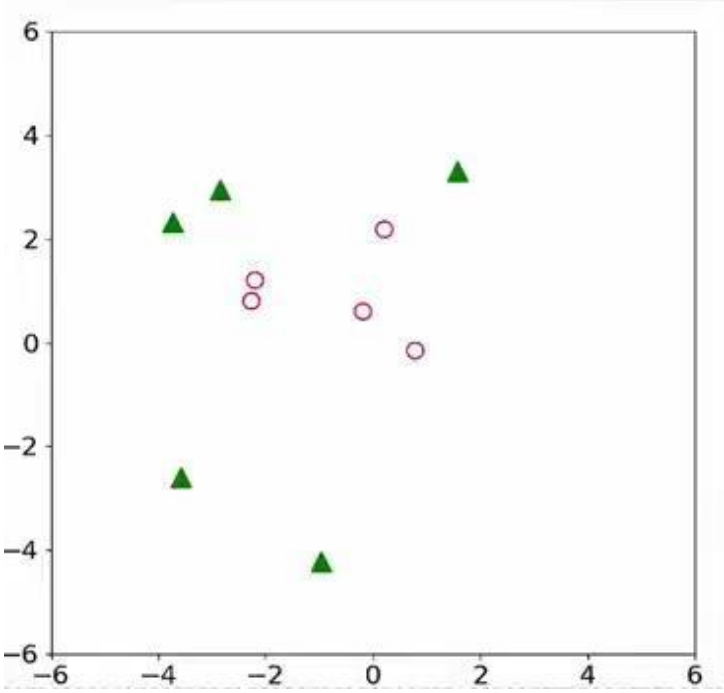
  - Main requirements being:

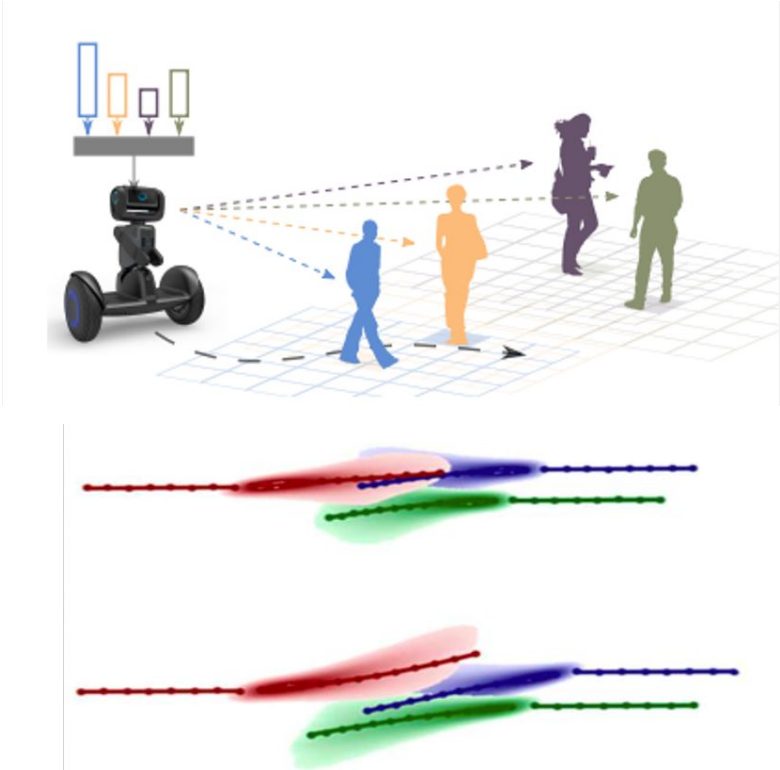
    Large sample of Expert trajectories/demos (high cost)

    Discriminator model

    Policy Network (Generator)

GAIL: Examples





Da L, Wei H. CrowdGAIL: A spatiotemporal aware method for agent navigation[J]. Electronic Research Archive, 2022, 31(2).

QUESTIONS?

Ho, Jonathan, and Stefano Ermon. "Generative adversarial imitation learning." Advances in neural information processing systems 29 (2016)- 2770 citations