Quiz 11

Deep Reinforcement Learning Algorithms Comparison

DQN - Deep Q-Network

Characteristics

- Value-based method -> Estimate of the optimal action-value function.
- Off-policy & model-free.
- Neural Network (NN) is used as a function approximator.

Loss Function

$$L = E\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_k) - Q(s, a; \theta_k)\right)^2\right]$$

Update function

$$\theta_{k+1} = \theta_k + \alpha\left(r + \gamma \max_{a'} Q(s', a'; \theta_k) - Q(s, a; \theta_k)\right) \nabla_{\theta_k} Q(s, a; \theta_k)$$

Cons

- Maximization bias - Selecting the maximum estimated value over and over again causes this bias. This produces low-quality policy & unstable training.

DDQN - Double Deep Q-Network

Characteristics

- Value-based method
- Uses 2 NNs to select & evaluate action.
- We will reduce the *Root Mean Squared* (RMS) Error between the estimated Q & the target Q.

Loss Function

$$L = E\left[\left(r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right)^2\right]$$

Update Function

$$Q^A_{t+1} = (1-\alpha)Q^A_t(s_t, a_t) + \alpha\Big(R_t + \gamma Q^B_t\big(s_{t+1}, \arg\max_a Q^A_t(s_{t+1}, a)\big)\Big)$$

$$Q^B_{t+1} = (1-\alpha)Q^A_t(s_t, a_t) + \alpha\Big(R_t + \gamma Q^B_t\big(s_{t+1}, \arg\max_a Q^A_t(s_{t+1}, a)\big)\Big)$$

Algorithm

Algorithm 1 : Double Q-learning (Hasselt et al., 2015)

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma\, Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

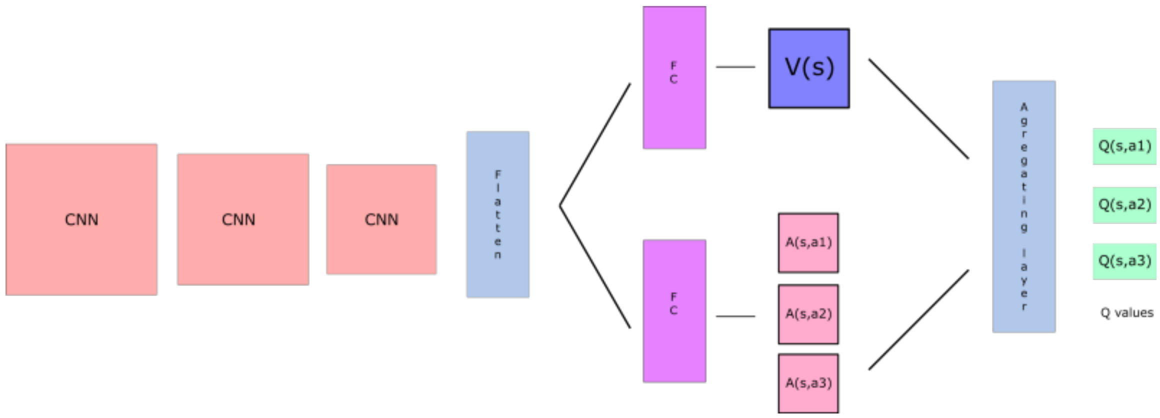
Pros

- Avoids maximization bias.
- More stable & reliable than the DQN.

Dueling DQN

Characteristics

- Value-based method
- Has 2 estimators: One for state-value function & other for state-dependent action advantage function.



Aggregate computation

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + (A(s, a; \theta, \alpha) - \frac{1}{A}\sum_{a'} A(s, a'; \theta, \alpha))$$

Pros

- Learns which states are valuable & which are not.
- Fast training.

REINFORCE

Characteristics

- Monte Carlo Policy-gradient method -> Estimate best weight by gradient ascent.
- On-policy
- Updates parameters by stochastic gradient ascent.

Algorithm

REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G \leftarrow$ return from step t
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Pros

1. Works in environments with discrete or continuous action spaces.

Cons

1. Has high variance.

A2C - Advantage Actor Critic

Characteristics

- Actor-critic (AC) method
  - *Critic:* Updates action-value function.

- *Actor:* Updates policy parameters based on Critic.

- On-policy.

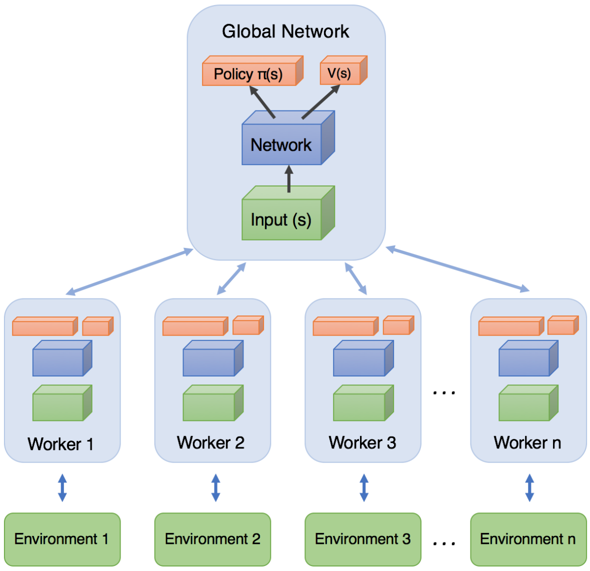
Advantage Function
$A(s, a) = Q(s, a) - V(s)$

Pros
1. Advantage function reduces variance.

Cons
1. Not suited for continuous control.

A3C - Asynchronous Advantage Actor Critic

Characteristics
- Uses multiple agents which has its own set of parameters & their own copy of the environment.
- Combines the strength of both *Value-iteration* & *Policy-gradient* methods, and predicts *value function* $V(s)$ & the *optimal policy function* $\pi(s)$.



Algorithm

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 $\theta'_v = \theta_v$
  $t_{start} = t$
  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 $i \in \{t-1, \dots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i; \theta')(R - V(s_i; \theta'_v))$
    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 $T > T_{max}$

Pros

1. Copies of agent in environment de-correlates the data.
2. No Experience Replay is needed.

Cons

1. Not suited for continuous control.
2. Data inefficient.

TRPO - Trust Region Policy Optimization

Characteristics

- Trust Region method
- On-policy [Source] & model-free.
- Adds KL (Kullback-Leibler) divergence for optimization.

Objective Function

$J(\theta) = E_{s \sim \rho^{\pi_{\theta_{\text{old}}}}, a \sim \pi_{\theta_{\text{old}}}} \left[\frac{\pi_\theta(a\,|\,s)}{\pi_{\theta_{\text{old}}}(a|s)} \hat{A}_{\theta_{\text{old}}}(s,a) \right]$ **or** $J^{\text{TRPO}}(\theta) = E[r(\theta)\hat{A}_{\theta_{\text{old}}}(s,a)]$

Where, $r(\theta)$ is the probability ratio between old & new policies.

Algorithm

Algorithm 1 Trust Region Policy Optimization

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: Hyperparameters: KL-divergence limit δ, backtracking coefficient α, maximum number of backtracking steps K
3: **for** $k = 0, 1, 2, \ldots$ **do**
4: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
5: Compute rewards-to-go $\hat{R}_t$.
6: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
7: Estimate policy gradient as

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

8: Use the conjugate gradient algorithm to compute

$$\hat{x}_k \approx \hat{H}_k^{-1} \hat{g}_k,$$

where $\hat{H}_k$ is the Hessian of the sample average KL-divergence.
9: Update the policy by backtracking line search with

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k,$$

where $j \in \{0, 1, 2, \ldots K\}$ is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraint.
10: 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.
11: **end for**

Pros

- Suitable for environments with both continuous & discrete action space.

Cons

- Data inefficient

PPO - Proximal Policy Optimization

Characteristics

- Trust Region method
- On-policy [Source 1 & Source 2] & model-free.
- Learns policy & value function at the same time.
- Improves/simplifies on TRPO by adding a *clipped surrogate objective*.
- Has entropy component which is the measure of uncertainty in the policy (i.e.) lower the entropy, the policy is more confident in choosing an action.
- Prefers exploration & avoids bad local optimum by rewarding for choosing actions with high entropy.

Objective Function

$$J^{\text{CLIP}}(\theta) = \hat{E}_t\big[\min\big(r(\theta)\hat{A}_{\theta_{\text{old}}}(s,a),\ \text{clip}(r(\theta),1-\epsilon,1+\epsilon)\hat{A}_{\theta_{\text{old}}}(s,a)\big)\big]$$

The clip component clips the ratio within $[1-\epsilon,\ 1+\epsilon]$.

Algorithm

Algorithm 5 PPO with Clipped Objective

Input: initial policy parameters θ_0, clipping threshold ϵ

for $k = 0, 1, 2, \ldots$ **do**

Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$

Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm

Compute policy update

$$\theta_{k+1} = \arg\max_\theta \mathcal{L}_{\theta_k}^{CLIP}(\theta)$$

by taking K steps of minibatch SGD (via Adam), where

$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_k}\left[\sum_{t=0}^{T}\left[\min(r_t(\theta)\hat{A}_t^{\pi_k}, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t^{\pi_k})\right]\right]$$

end for

Pros

1. Simpler compared to TRPO, but same performance.
2. Works in both discrete & continuous environments.
3. Faster & stable training.
4. Entropy regularization

Cons

1. Data inefficient

SAC - Soft Actor Critic

Characteristics

- Off-policy
- Avoids convergence to bad local optimum by rewarding actions with high entropy.
- Learns value function & the policy at the same time.

Objective function

$$J(\pi) = \sum_{t=0}^{T} E_{(s_t,a_t)\sim\rho_\pi}\big[(s_t, a_t) + \alpha \mathrm{H}(\pi(\,.\,|\,s_t))\big]$$

Algorithm

Algorithm 1 Soft Actor-Critic

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

1. Sample efficient - Useful when environment is expensive to sample from.
2. Entropy maximization - Data efficient when compared to ER of PPO & balances exploitation-exploration.

Cons

1. Suitable only for environment with continuous action space.
2. Unstable when compared to PPO.