Reinforcement Learning V
Introduction to Deep Reinforcement Learning

Antoine SYLVAIN

EPITA

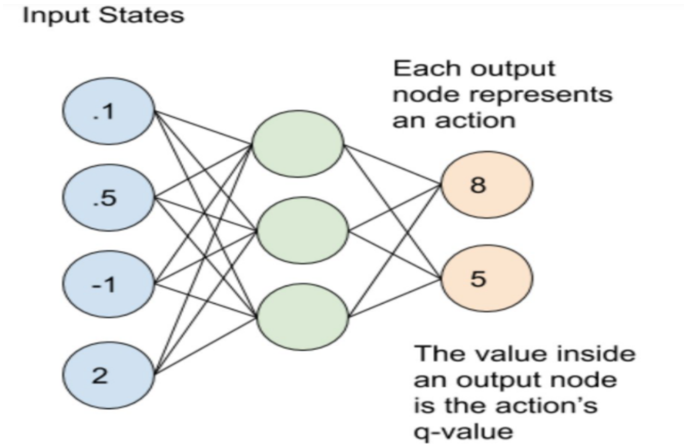
2021

Contents

Deep Q-Network

- Two neural networks, with the same architecture, but different weights
- A **main** network
- A **target** network
- Every N steps, weights from the main network are copied to the target network

Deep Q-Network


Input States
.1
.5
-1
2
Each output node represents an action
8
5
The value inside an output node is the action's q-value

Deep Q-Learning

- For each given state input, the network outputs estimate Q-values for each action that can be taken from that state
- The objective of this network is to approximate the optimal Q-function, that will satisfy the Bellman equation
- $q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(s', a')]$
- Loss from the network is calculated by comparing the outputted Q-values to the target Q-values
- We want to minimize the loss
- After the loss is calculated, the weights within the network are updated via backpropagation

Experience Replay

- Replay memory: dataset where agent's experiences at each time step are stored
- $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$
- We set a maximum size to replay memory table
- Network will be trained by sampling randomly from the replay memory dataset
- The act of gaining experience and sampling from the replay memory that stores these experiences is called experience replay
- A key reason for using replay memory is to break the correlation between consecutive samples
- If the network learned only from consecutive samples of experience as they occurred sequentially in the environment, the samples would be highly correlated and would therefore lead to inefficient learning. Taking random samples from replay memory breaks this correlation

DQN Algorithm

- Initialize replay memory capacity.
- Initialize the network with random weights.
- For each episode:
  - Initialize the starting state.
  - For each time step:
    - Select an action. (via Exploration or Exploitation)
    - Execute selected action in an emulator.
    - Observe reward and next state.
    - Store experience in replay memory.
    - Sample random batch from replay memory.
    - Preprocess states from batch.
    - Pass batch of preprocessed states to policy network.
    - Calculate loss between output Q-values and target Q-values.
    - Gradient descent updates weights in the policy network to minimize loss.

Contents

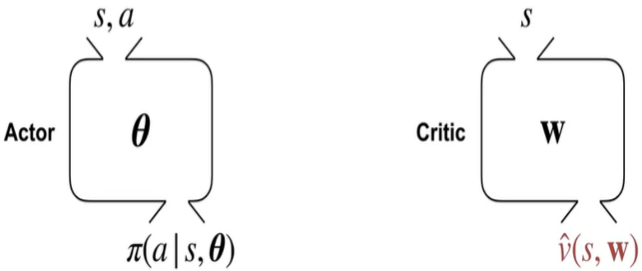
Actor-Critic Methods

- Two main components in policy gradient are the policy model and the value function
- If the value function is learned in addition to the policy, we would get Actor-Critic algorithm
- Two elements:
  - **Critic**: updates value function parameters w and depending on the algorithm it could be action-value $Q_w(a|s)$ or state-value $V_w(s)$
  - **Actor**: updates policy parameters θ, in the direction suggested by the critic, $\pi_\theta(a|s)$
- Convenient for continuous tasks

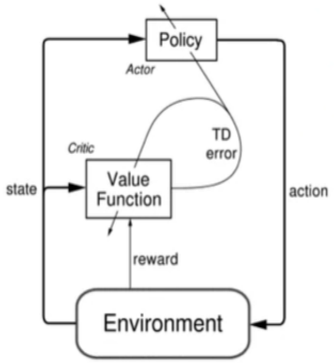
Approximating the Action-Value in the Policy Update

$$\theta_{t+1} \doteq \theta_t + \alpha \nabla \ln \pi(A_t \mid S_t, \theta_t)[R_{t+1} - \bar{R} + \hat{v}(S_{t+1}, \mathbf{w})]$$

Average Reward Semi-Gradient TD(0)



How the Actor and the Critic Interact ?



$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \nabla \ln \pi(A_t \mid S_t, \boldsymbol{\theta}_t)\delta_t$$

Action-Value Actor-Critic Algorithm

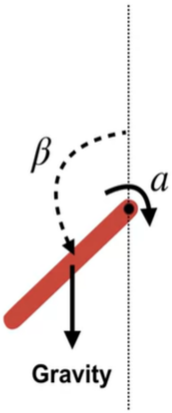
```
Initialisation of $s, \theta, w$, sample for $a \sim \pi_\theta(a|s)$
For $t = 1...\tau$:
  Sample reward $r_t \sim R(s,a)$ and next state $s' \sim P(s'|s,a)$
  Sample next action $a' \sim \pi_\theta(a'|s')$
  Update policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s,a)\nabla_\theta \ln \pi_\theta(a|s)$
  Compute the correction (TD error) for action-value at time $t$:
    $\delta_t = r_t + \gamma Q_w(s',a') - Q_w(s,a)$
  Update the parameters of action-value function:
    $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s,a)$
  $a \leftarrow a'$
  $s \leftarrow s'$
```

An example: Pendulum Swing-up

Pendulum Swing-Up

$$s \doteq \begin{bmatrix} \beta \\ \dot{\beta} \end{bmatrix} \begin{array}{l} \textbf{Angular Position} \\ \textbf{Angular Velocity} \end{array}$$

$$a \in \{-1, 0, 1\}$$



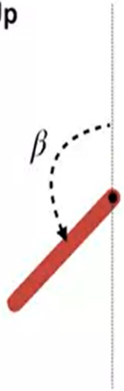
An example: Pendulum Swing-up

The Reward in Pendulum Swing-Up

$$R \doteq -|\beta|$$

$$-2\pi < \dot{\beta} < 2\pi$$

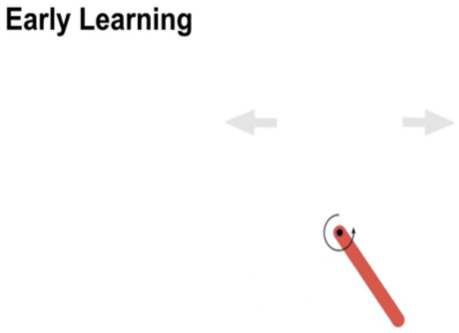
Continuing Task



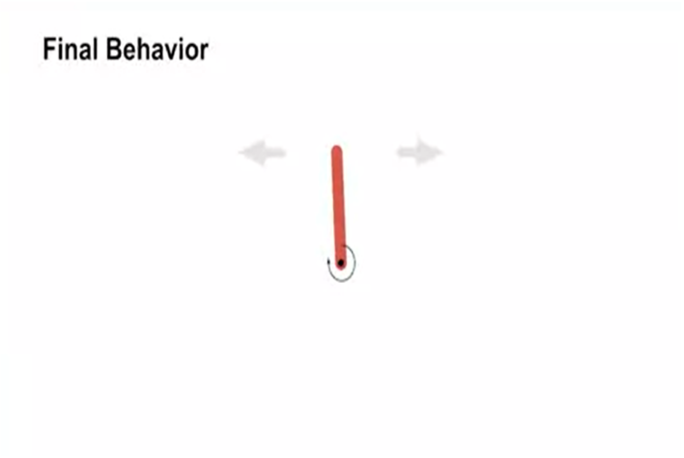
Key Points of the Pendulum Swing-Up

- The actions are not strong enough to move the pendulum directly to the vertical position from the resting position
- A good policy must apply actions that move the pendulum away from the desired position in order to gain enough momentum to swing up
- The vertical position is unstable, so good policy must continually balance the pole

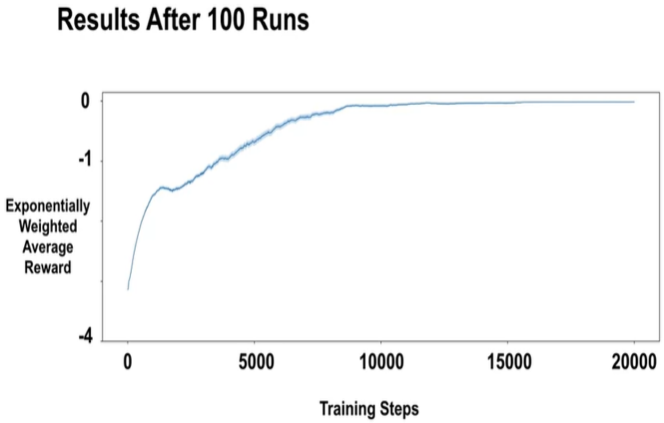
An example: Pendulum Swing-up



An example: Pendulum Swing-up



An example: Pendulum Swing-up



A3C: Asynchronous Advantage Actor-Critic

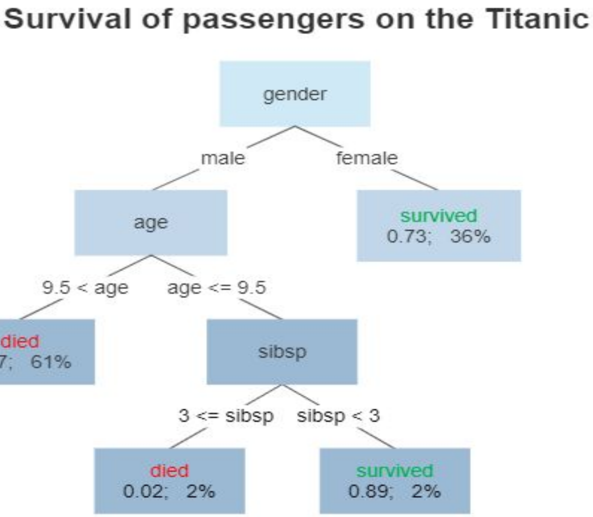
- Classic policy gradient method with a special focus on parallel training
- Critic learn the state-value function, while multiple actors are trained in parallel and get synced with global parameters from time to time
- Good for parallel training on one machine with multi-core CPU
- Enables the parallelism in multiple agent training

A3C Algorithm

```
Initialize global parameters $\theta$ ans $w$, thread-specific parameters $\theta'$ and $w'$
and time-step $t = 1$
While $T \leq \tau$:
  Reset gradient $d\theta = 0$ and $dw = 0$
  Synchronize thread-specific parameters with global ones: $\theta = \theta'$ and
$w = w'$
  While $s_t$ not terminal and $t - t_{start} \leq t - t_{max}$
    Pick the action $a_t \sim \pi_{\theta'}(a_t|s_t$
    $t = t + 1$; $T = T + 1$
  Initialize the variable that holds the return estimation $R = 0$ if $s_t$
terminal, else $R = V(s_t, w'$
  For $i = t - 1, ..., t_{start}$
    $R \leftarrow r_i + \gamma R$
    $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi_{\theta'}(a_i|s_i)(R - V(s_i, w'))$
    $dw \leftarrow dw + \nabla_{w'}(R - V(s_i, w'))^2$
  Update synchronously $\theta$ using $d\theta$, and $w$ using $dw$
```

Contents

Decision Tree



Decision Tree: Motivations

- Well-known class of algorithms
- Relevant as a function approximator, can take a state as input and return an action as output
- Allows the space to be divided with varying levels of resolution
- Requires little data pre-processing
- Easy to understand and interpret

Contents

Learning from Limited Sample

- All training data comes from the real system
- Often means that exploration must be limited
- Gathered data might have low variance (very little of the state space may be covered in the logs)

Ideas

- Use expert demonstration to bootstrap the agent, rather than starting from zero
- Focus on sample-efficiency, drive exploration when possible,
- ...

High-Dimensional Continuous State and Action Space

- Many practical real world problems have large and continuous state and action spaces
- For example, recommender systems action space can increase exponentially
- For example, the number of sensors and actuators to control cooling in a Google data center

Ideas

- Eliminate irrelevant actions
- Focus on related actions
- ...

Safety Constraints

- Almost all physical systems can destroy or degrade themselves and their environment
- Safety is important during system operation, but also during exploratory learning phases as well
- System: limiting system temperatures, limiting contact forces, maintaining minimum battery levels...
- Environment: avoiding dynamic obstacles, limiting end effector velocities...

Ideas

- Hardcode unviolable safety laws
- Try to evaluate the safety of a state
- ...

Partial Observability

- Almost all real systems are partially observable
- Real world systems are often stochastic and noisy compared to most simulated environments
- Sensor and action noise as well as action delays add to the perturbations an agent may experience in the real-world setting

Ideas

- Work with Partially Observable Markov Decision Processes
- ...

Unspecified and Multi-Objective Reward Functions

- Reinforcement learning frames policy learning through the lens of optimizing a global reward function
- Most systems have multi-dimensional costs to be minimized
- In many cases, system or product owners do not have a clear picture of what they want to optimize
- When an agent is trained to optimize one metric, other metrics are discovered that also need to be maintained or improved
- A lot of the work on deploying RL to real systems is in formulating the reward function, which may be multi-dimensional
- It may be desired that the policy performs well for all task instances and not just in expectation

Ideas

- Track the different objectives individually
- Minimize rare catastrophic reward on a given objective
- ...

Explainability

- Real systems is that they are owned and operated by humans, who need to be reassured about the controllers' intentions and require insights regarding failure cases
- Thus policy explainability is important for real-world policies, especially in cases where the policy might find an alternative and unexpected approach to controlling a system
- Understanding the longerterm intent of the policy is important for obtaining stakeholder buy-in
- In the event of policy errors, being able to understand the error's origins *a posteriori* is essential

Ideas

- Translate the policy into a human-readable program
- Distill a learned neural network policy into an explicit program
- ...

Real-Time Inference and System Delays

- Policy inference must be done in real-time at the control frequency of the system
- For example, this may be on the order of milliseconds for a recommender system responding to a user request
- Most real systems have delays in either the sensation of the state, the actuators, or the reward feedback

Ideas

- Use a model-based approach if the data is too slow to collect
- Try to predict the delays of the system
- Keep previous states and actions in memory to recompute their function later, when the delayed reward will be known
- ...