Policy Gradient techniques

- Sarsa
- Actor-critic methods
- A3C and A2C

A problem of Q-learning

Q-learning is a 1-step method since it updates the action value $Q(s, a)$ toward the one- step return $r + \gamma max_{a'} Q^i(s', a')$.

This only directly affects the value of the state action pair (s, a) that led to the reward. The values of other state action pairs are affected only indirectly through the updated value $Q(s, a)$.

This can make the learning process slow since many updates are required to propagate a reward to all relevant preceding states and action.

Can we learn a policy directly, e.g. finding the best policy from a collection of policies?

On-policy vs off policy techniques

Q-learning is an off-policy technique: it does not rely on any policy, and only needs local transitions.

- can take advantage of experience replay

On policy techniques try to improve the current policy.

- it requires sampling long trajectories according to the current strategy
- need many diversified trajectories (e.g. parallel agents)

SARSA

SARSA

State-Action-Reward-State-Action (SARSA) is a learning algorithm very similar to Q-learning.

- update for Q-learning:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

- update for SARSA:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Instead of considering the best action at time $t+1$ (greedy choice) we consider the actual action a_{t+1} under the current policy.

SARSA vs. Q-learning

Q-learning is based on single step transisitons

$$(s_i, a_i, r_i, s_{i+1})$$

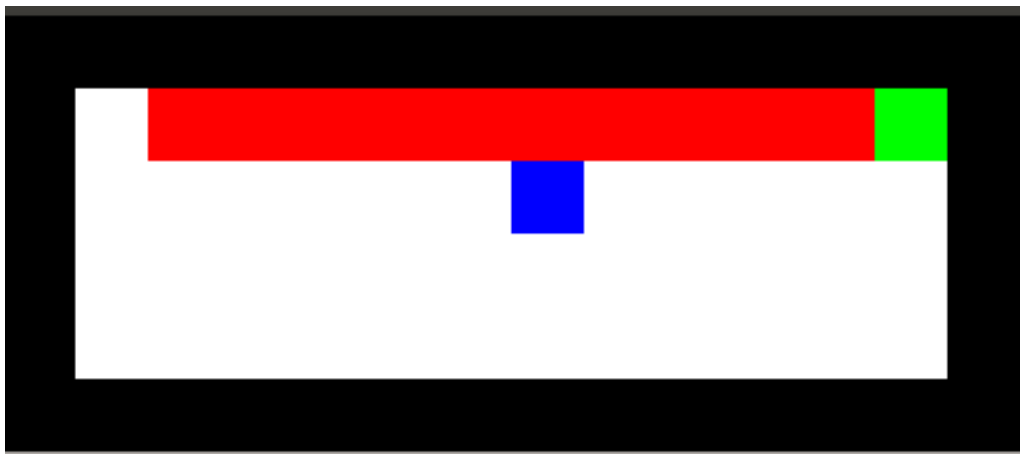
SARSA is based on mini-trajectories (two steps):

$$(s_i, a_i, r_i, s_{i+1}, a_{i+1})$$

For this reason, SARSA is traditionaly considered "on policy" (more later).

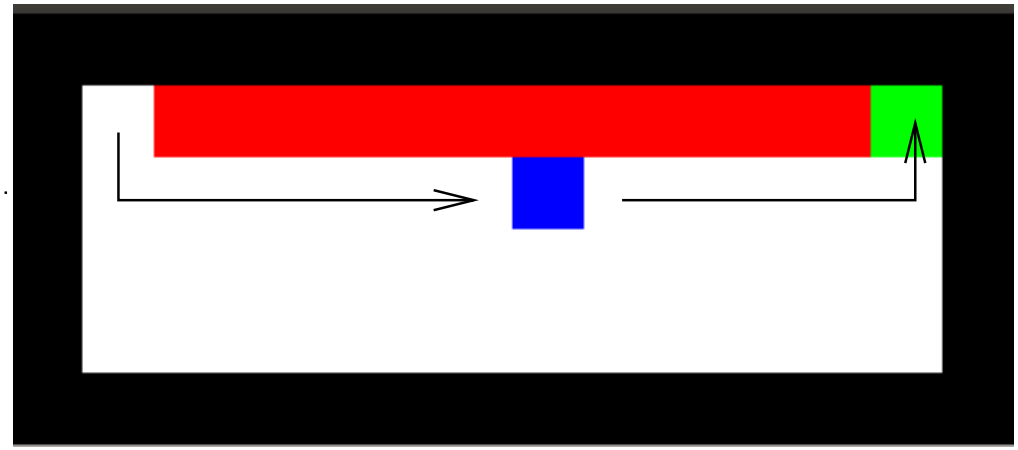
The mouse and cliff scenario

A mouse (blue) is trying to get to a piece of cheese (green). Additionally, there is a cliff in the map (red) that must be avoided, or the mouse falls and dies.



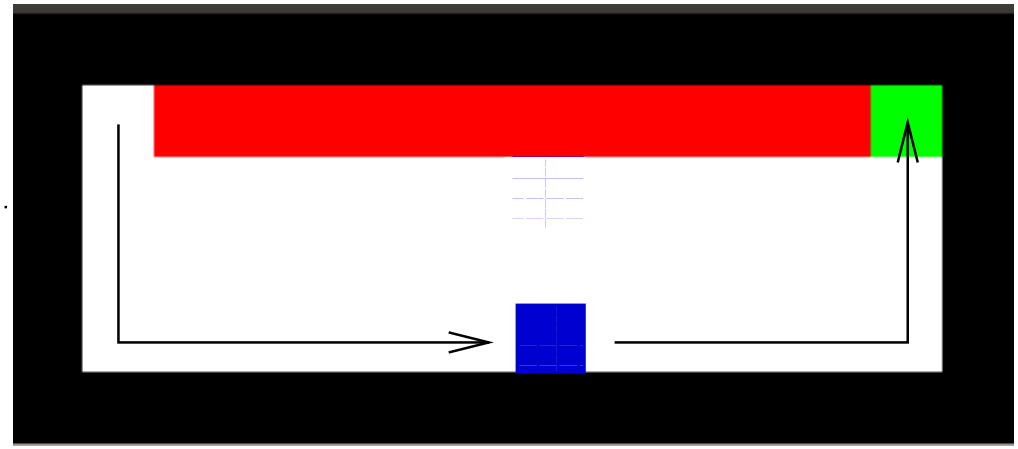
Example from SARSA vs Q-learning by Travis DeWolf.

Via Q-learning



With Q-learning, the mouse ends up running along the edge of the cliff, but occasionally jumping off and plummeting to its death.

Via SARSA



With SARSA, the mouse learns that from time to time it commits errors. The best path is not to run straight to the cheese along the edge of the cliff but taking a safer root, far away from it.

Even if a random action is chosen there is little chance of it resulting in death.

Policy gradients

Policy gradients

How can we improve a strategy?
Suppose to have a class of parametrized policies $\Pi = \{\pi_\theta\}$.
For each policy π_θ, define its value:

$$J(\theta) = \mathbb{E} \sum_{t \geq 0} \gamma^t r_t$$

We want to find the optimal policy

$$\theta^* = argmax_\theta J(\theta)$$

As usual, we address the problem by gradient ascent on policy parameters.

Policy gradients key idea

The main idea underlying policy gradients is **reinforcing good actions**: to push up the probabilities of actions that lead to higher return, and push down the probabilities of actions that lead to a lower return, until you arrive at the optimal policy.

The policy gradient method will iteratively amend the policy network weights (with smooth updates) to make state-action pairs that resulted in positive return more likely, and make state-action pairs that resulted in negative return less likely.

REINFORCE approach

There are several different approaches to policy gradient ascent.

For a given, sampled trajectory, standard REINFORCE updates the policy parameters θ in the direction

$$\nabla_\theta log \pi(a_t|s_t, \theta) R_t$$

(nice theory but no time to investigate)

Intuition:

- if R_t is high push up the probability of actions
- if R_t is low push down the probability of actions

Adding a baseline

Problem:
the raw value of a trajectory is not so meaningful. E.g. if rewards are all positive, we keep pushing up probabilities of all actions.

What matters is whether a reward is better or worse than expected.

Idea:
Introduce a baseline dependent on the state.

The new estimator becomes:

$$\nabla_\theta log \pi(a_t|s_t, \theta)(R_t - b(s_t))$$

The Actor-critic architecture

An excellent choice for the baseline is the value function of the state

$$b(s_t) = V^{\pi}(s_t)$$

This approach can be viewed as an **actor-critic** architecture where the policy π is the actor and the value function (baseline) is the critic.

A3C pseudo-code

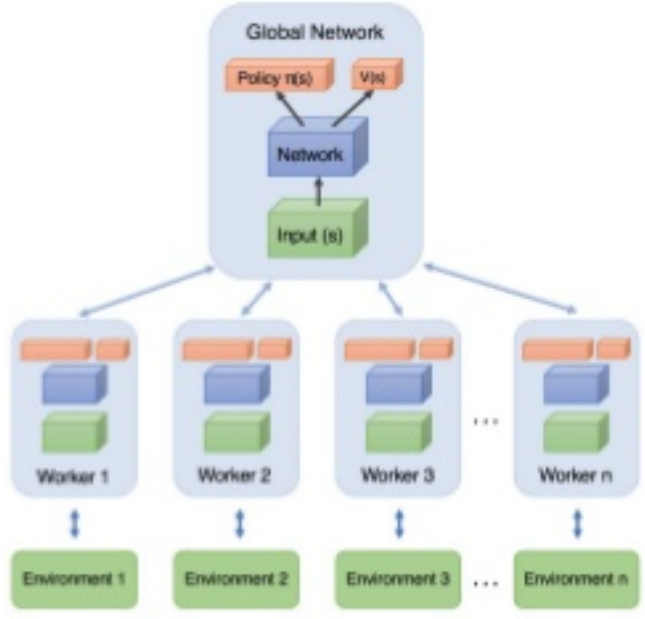
Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

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

A3C architecture

Asynchronous Advantage Actor-Critic (A3C)

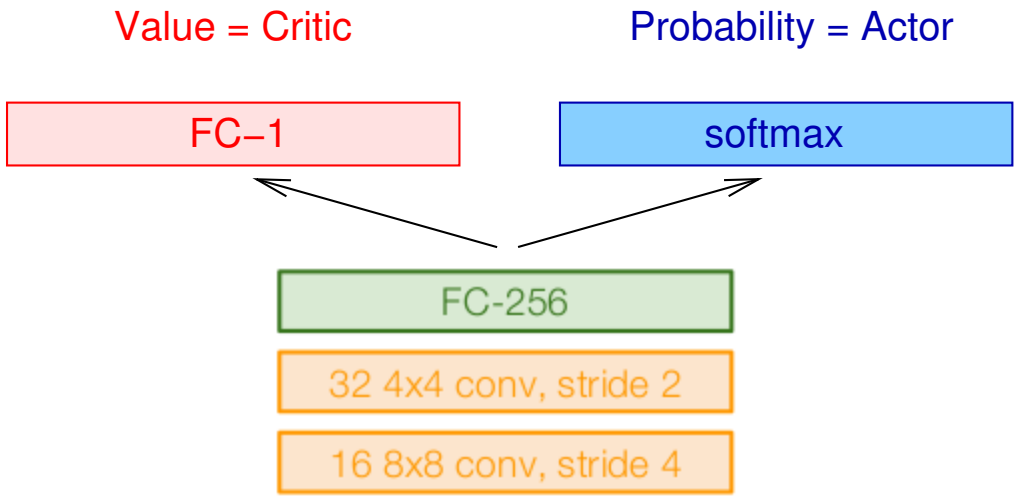
- A3C utilizes multiple Worker agents
- Speedup & Diverse Experience
- Combines benefits of Value & Policy Iteration
- Continuous & Discrete action spaces



Images (L-R): A3C: Training workflow of each worker agent (L) and High-level architecture (R)

A3C architecture for Atari

Most parameters of actor and critic are shared



$t_{max} = 5$ (short!), $\gamma = 0.99$

A3C vs. A2C

A2C is similar to A3C but without asynchronous agents; essentially, it is a single-worker variant of A3C. Empirically, A2C produces comparable performance with A3C while being more efficient.

Quoting the authors:

> *After reading the paper, AI researchers wondered whether the asynchrony led to improved performance (e.g. "perhaps the added noise would provide some regularization or exploration?"), or if it was just an implementation detail that allowed for faster training with a CPU-based implementation [. . .] Our synchronous A2C implementation performs better than our asynchronous implementations - we have not seen any evidence that the noise introduced by asynchrony provides any performance benefit. This A2C implementation is more cost-effective than A3C when using single-GPU machines, and is faster than a CPU-only A3C implementation when using larger policies.*

Proximal Policy Optimization

Problems with policy gradient methods

- **Sample inefficiency** Samples are only used once. When policy is updated, the new policy is used to sample another trajectory. As sampling is often expensive, this can be prohibitive. However, after a large policy update, the old samples are no longer representative.
- **Inconsistent policy updates** Policy updates tend to overshoot and miss the reward peak, or stall prematurely. Vanishing and exploding gradients are severe problems. The algorithm may not recover from a poor update.
- **High reward variance** Policy gradient is a Monte Carlo learning approach, taking into account the full reward trajectory. Such trajectories often suffer from high variance, hampering convergence (partially addressed by adding a critic)

Trusted region

When passing from a given policy π_k (e.g. a randomized version of the current policy) to a new policy π, small modifications can easily results in large fluctuations in behaviours and performances.

How can we take the biggest possible improvement step on a policy, still reamining inside a **trusted region**, i.e. without stepping so far that we accidentally cause performance collapse?

TRPO: key equations

Let π_θ denote a policy with parameters θ.
In theory, the TRPO update is:

$$\theta_{k+1} = \arg\max_{\theta} \mathcal{L}(\theta_k, \theta) \text{ s.t. } \overline{KL}(\theta||\theta_k) \leq \delta$$

where $\mathcal{L}(\theta_k, \theta)$ is the surrogate advantage, a measure of how policy π_θ performs relative to the old policy π_{θ_k} using data from the old policy ($A^{\pi_{\theta_k}}(s,a)$ is the advantage of a in s):

$$\mathcal{L}(\theta_k, \theta) = \mathbb{E}_{s,a\sim\pi_{\theta_k}} \frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a)$$

and $\bar{D}_{KL}(\theta||\theta_k)$ is an average KL-divergence between policies across states visited by the old policy:

$$\overline{KL}(\theta||\theta_k) = \mathbb{E}_{s\sim\pi_{\theta_k}} KL\left(\pi_\theta(\cdot|s)||\pi_{\theta_k}(\cdot|s)\right)$$

TRPO implementation

The previous theoretical update is hard to implement.

TRPO makes approximations based on Taylor expansions, to improve efficiency.

The theory is quite complex and beyond the scope of this introductory lessons.
See TRPO for additional information.

PPO: key equations

PPO achieves a similar objective of TRPO by updating policies via

$$\theta_{k+1} = \arg\max_{\theta} \mathbb{E}_{s,a\sim\pi_{\theta_k}} [L(s, a, \theta_k, \theta)]$$

where

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \quad clip(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon) A^{\pi_{\theta_k}}(s, a) \right)$$

The probability ratio is clipped in a range $[1-\epsilon, 1+\epsilon]$, but only if the resulting loss is larger than the unclipped value; in this way, the final objective is a lower bound (i.e., a pessimistic bound) on the unclipped objective.

summary of possible behavious

ratio	Advantage	Loss	clip	grad. $\neq 0$
$\frac{\pi_\theta(a\|s)}{\pi_{\theta_k}(a\|s)} \in [1-\epsilon, 1+\epsilon]$	$A^{\pi_{\theta_k}}(s,a) > 0$	$\frac{\pi_\theta(a\|s)}{\pi_{\theta_k}(a\|s)} A^{\pi_{\theta_k}}(s,a)$	*no*	*yes*
$\frac{\pi_\theta(a\|s)}{\pi_{\theta_k}(a\|s)} \in [1-\epsilon, 1+\epsilon]$	$A^{\pi_{\theta_k}}(s,a) < 0$	$\frac{\pi_\theta(a\|s)}{\pi_{\theta_k}(a\|s)} A^{\pi_{\theta_k}}(s,a)$	*no*	*yes*
$\frac{\pi_\theta(a\|s)}{\pi_{\theta_k}(a\|s)} < 1-\epsilon$	$A^{\pi_{\theta_k}}(s,a) > 0$	$\frac{\pi_\theta(a\|s)}{\pi_{\theta_k}(a\|s)} A^{\pi_{\theta_k}}(s,a)$	*no*	*yes*
$\frac{\pi_\theta(a\|s)}{\pi_{\theta_k}(a\|s)} < 1-\epsilon$	$A^{\pi_{\theta_k}}(s,a) < 0$	$(1-\epsilon) A^{\pi_{\theta_k}}(s,a)$	*yes*	*no*
$\frac{\pi_\theta(a\|s)}{\pi_{\theta_k}(a\|s)} > 1+\epsilon$	$A^{\pi_{\theta_k}}(s,a) > 0$	$(1+\epsilon) A^{\pi_{\theta_k}}(s,a)$	*yes*	*no*
$\frac{\pi_\theta(a\|s)}{\pi_{\theta_k}(a\|s)} > 1+\epsilon$	$A^{\pi_{\theta_k}}(s,a) < 0$	$\frac{\pi_\theta(a\|s)}{\pi_{\theta_k}(a\|s)} A^{\pi_{\theta_k}}(s,a)$	*no*	*yes*

PPO pseudocode

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

What future for DRL?

In comparison with other fields, the impact of Deep Learning techniques in Reinforcement Learning has remained quite superficial

- time to go deeper?
- too algorithmic?
- need learning to learn?
- ...

Recommended reading:
What is the future of Reinforcement Learning