Reinforcement Learning
Policy Search: Actor-Critic and Gradient Policy search

Mario Martin

CS-UPC

May 7, 2020
Goal of this lecture

- So far we approximated the value or action-value function using parameters $\theta$ (e.g. neural networks)

  $$V_\theta \approx V^\pi$$

  $$Q_\theta(s, a) \approx V^\pi(s)$$

- A policy was generated directly from the value function e.g. using $\epsilon$-greedy

- In this lecture we will directly parameterize the policy in a stochastic setting

  $$\pi_\theta(a|s) = P_\theta(a|s)$$

- and do a direct **Policy search**

- Again on model-free setting
Three approaches to RL

**Value based** learning: **Implicit policy**

- Learn value function \( Q_\theta(s, a) \) and from there infer policy
  \[ \pi(s) = \arg \max_a Q(s, a) \]

**Policy based** learning: **No value function**

- Explicitly learn policy \( \pi_\theta(a|s) \) that implicitly maximize
  reward over all policies

**Actor-Critic** learning: **Learn both Value Function and Policy**
Advantages of Policy over Value approach

- **Advantages:**
  - In some cases, computing Q-values is harder than picking optimal actions
  - Better convergence properties
  - Effective in high dimensional or continuous action spaces
  - Exploration can be directly controlled
  - Can learn stochastic policies

- **Disadvantages:**
  - Typically converge to a local optimum rather than a global optimum
  - Evaluating a policy is typically data inefficient and high variance
Stochastic Policies

- In general, two kinds of policies:
  - Deterministic policy
    \[ a = \pi_\theta(s) \]
  - Stochastic policy
    \[ P(a|s) = \pi_\theta(a|s) \]

- Nice thing is that they are smoother than greedy policies, and so, we can compute gradients!

- Not new: \( \epsilon \)-greedy is stochastic...
In general, two kinds of policies:

- Deterministic policy
  \[ a = \pi_\theta(s) \]

- Stochastic policy
  \[ P(a|s) = \pi_\theta(a|s) \]

Nice thing is that they are **smoother** than greedy policies, and so, we can compute **gradients**!

Not new: $\epsilon$-greedy is stochastic... but different idea. Stochastic policy is good on its own, not because it is an approx. of a greedy policy

Any example where an stochastic policy could be better than a deterministic one?
Two–player game of rock–paper–scissors:
- Scissors beats paper
- Rock beats scissors
- Paper beats rock

Consider policies for iterated rock–paper–scissors
- A deterministic policy is easily exploited
- A uniform random policy is optimal (i.e., Nash equilibrium)
The agent cannot differentiate the grey states
Consider features of the following form:
\[ \phi_d(s) = 1(\text{wall to } d) \quad \forall d \in \{N, E, S, W\} \]

Compare value-based RL, using an approximate value function
\[ Q_\theta(s, a) = f_\theta(\phi(s, a)) \]

To policy-based RL, using a parametrized policy
\[ \pi_\theta(a|s) = g_\theta(\phi(s, a)) \]
Stochastic Policies when aliased states (POMDPs)

- Under aliasing, an optimal deterministic policy will either
  - move W in both gray states
  - move E in both gray states
- Either way, it can get stuck and never reach the money
- So it will be stuck in the corridor for a long time
- Value–based RL learns a deterministic policy (or near deterministic when it explores)
Stochastic Policies when aliased states (POMDPs)

- An optimal stochastic policy will randomly move E or W in gray states
  - $\pi_\theta(\text{move E} | \text{wall to N and S}) = 0.5$
  - $\pi_\theta(\text{move W} | \text{wall to N and S}) = 0.5$

- It will reach the goal state in a few steps with high probability

- Policy-based RL can learn the optimal stochastic policy
Policy optimization
Goal: given policy $\pi_\theta(a|s)$ with parameters $\theta$, find best $\theta$

... but how do we measure the quality of a policy $\pi_\theta$?

In episodic environments we can use the start value

$$J_1(\theta) = V^{\pi_\theta}(s_1)$$
In continuing environments we can use the average value

\[ J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s)V^{\pi_\theta}(s) \]

where \( d^{\pi_\theta}(s) \) is stationary distribution of Markov chain for \( \pi_\theta \) (can be estimated as the expected number of time steps on \( s \) in a randomly generated episode following \( \pi_\theta \) divided by time steps of trial)

Or the average reward per time-step

\[ J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(a|s)r(s, a) \]

For simplicity, we will mostly discuss the episodic case, but can easily extend to the continuing / infinite horizon case
Goal: given policy $\pi_\theta(a|s)$ with parameters $\theta$, find best $\theta$

Policy based reinforcement learning is an optimization problem

Find policy parameters $\theta$ that maximize $J(\theta)$

Two approaches for solving the optimization problem
  - Gradient-free
  - Policy-gradient
Subsection 1

Gradient Free Policy Optimization
Goal: given parametrized method (with parameters $\theta$) to approximate policy $\pi_\theta(a|s)$, find best values for $\theta$

Policy based reinforcement learning is an optimization problem

Find policy parameters $\theta$ that maximize $J(\theta)$

Some approaches do not use gradient

- Hill climbing
- Simplex / amoeba / Nelder Mead
- Genetic algorithms
- Cross-Entropy method (CEM)
- Covariance Matrix Adaptation (CMA)
Goal: given parametrized method (with parameters $\theta$) to approximate policy $\pi_\theta(a|s)$, find best values for $\theta$

Policy based reinforcement learning is an optimization problem

Find policy parameters $\theta$ that maximize $J(\theta)$

Some approaches do not use gradient

- Hill climbing
- Simplex / amoeba / Nelder Mead
- Genetic algorithms
- Cross-Entropy method (CEM)
- Covariance Matrix Adaptation (CMA)
Cross-Entropy Method (CEM)

- A simplified version of Evolutionary algorithm
- Works *embarrassingly* well in some problems, f.i.
  - Playing Tetris (Szita et al., 2006), (Gabillon et al., 2013)
  - A variant of CEM called Covariance Matrix Adaptation has become standard in graphics (Wampler et al., 2009)

- Very simple idea:
  1. From current policy, sample $N$ trials (large)
  2. Take the $M$ trials with larger long-term return (we call the elite)
  3. Fit new policy to behave as in M best sessions
  4. Repeat until satisfied

- Policy improves gradually
# Tabular Cross-Entropy

## Tabular Cross-Entropy Algorithm

Given $M$ (f.i, 20), $N$ (f.i. 200)

Initialize matrix policy $\pi(a|s) = A_{s,a}$ randomly

repeat

Sample $N$ roll-outs of the policy and collect for each $R_t$

elite = $M$ best samples

\[
\pi(a|s) = \frac{[\text{times in } M \text{ samples took } a \text{ in } s] + \lambda}{[\text{times in } M \text{ samples was at } s] + \lambda|A|}
\]

until convergence

return $\pi$

Notice! No value functions!
Some possible problems and solutions:

- If you were in an state only once, you only took one action and probabilities become 0/1
- Solution: Introduction of $\lambda$, a parameter to smooth probabilities
Some possible problems and solutions:

- If you were in an state only once, you only took one action and probabilities become 0/1
- Solution: Introduction of $\lambda$, a parameter to smooth probabilities
- Due to randomness, algorithm will prefer “lucky” sessions (training on lucky sessions is no good)
- Solution: run several simulations with these state-action pairs and average the results.
Approximated Cross-Entropy Method (CEM)

Given $M$ (f.i., 20), $N$ (f.i., 200) and function approximation (f.i., NN) depending on $\theta$
Initialize $\theta$ randomly
repeat
  Sample $N$ roll-outs of the policy and collect for each $R_t$
  elite = $M$ best samples
  $\theta = \theta + \alpha \nabla \left[ \sum_{s,a \in \text{elite}} \log \pi_{\theta}(a|s) \right]$
until convergence
return $\pi_{\theta}$
Approximated Cross-Entropy Method (CEM)

- No Value function involved
- Notice that best policy is:

\[ \arg\max_{\pi_\theta} \sum_{s,a \in \text{elite}} \log \pi_\theta(a|s) = \arg\max_{\pi_\theta} \prod_{s,a \in \text{elite}} \pi_\theta(a|s) \]

so gradient goes in that direction (some theory about Entropy behind)

- Intuitively, is the policy that maximizes similarity with behavior of successful samples
- Tabular case is a particular case of this algorithm
- I promised no gradient, but notice that gradient is for the approximation, not for the rewards of the policy
- Can easily be extended to continuous action spaces (f.i. robotics)
Gradient-Free methods

- Often a great simple baseline to try

- Benefits
  - Can work with any policy parameterizations, including non-differentiable
  - Frequently very easy to parallelize (faster wall-clock *training* time)

- Limitations
  - Typically not very *sample* efficient because it ignores temporal structure
Subsection 2

Policy gradient
Policy gradient methods

- Policy based reinforcement learning is an **optimization** problem
- Find policy parameters $\theta$ that maximize $V^{\pi_\theta}$
- We have seen gradient-free methods, but greater efficiency often possible using gradient in the optimization
- Pletora of methods:
  - Gradient descent
  - Conjugate gradient
  - Quasi-newton
- We focus on *gradient ascent*, many extensions possible
- And on methods that exploit sequential structure
Policy gradient differences wrt Value methods

- With Value functions we use Greedy updates:

\[ \theta_{\pi'} = \arg \max_\theta \mathbb{E}_{\pi_\theta}[Q^{\pi}(s, a)] \]

- Potentially unstable learning process with large policy jumps because arg max is not differentiable

- On the other hand, Policy Gradient updates are:

\[ \theta_{\pi'} = \theta_{\pi'} + \alpha \frac{\partial J(\theta)}{\partial \theta} \]

- Stable learning process with smooth policy improvement
Policy gradient method

- Define $J(\theta) = J^{\pi\theta}$ to make explicit the dependence of the evaluation policy on the policy parameters.
- Assume episodic MDPs.
- Policy gradient algorithms search for a local maximum in $J(\theta)$ by ascending the gradient of the policy, w.r.t parameters $\theta$

$$\nabla \theta = \alpha \nabla_\theta J(\theta)$$

- Where $\nabla_\theta J(\theta)$ is the policy gradient and $\alpha$ is a step-size parameter.
Computing the gradient analytically

- We now compute the policy gradient analytically
- **Assume policy is differentiable whenever it is non-zero**
We now compute the policy gradient analytically.

Assume policy is differentiable whenever it is non-zero and that we know the gradient $\nabla_\theta \pi_\theta(a|s)$.

Denote a state-action trajectory (or trial) $\tau$ as

$$\tau = (s_0, a_0, r_1, s_1, a_1, r_2, \ldots s_{T-1}, a_{T-1}, r_T, s_T)$$

Define long-term-reward to be the sum of rewards for the trajectory $R(\tau)$

$$R(\tau) = \sum_{t=1}^{T} r(s_t)$$

It can be discounted or not. Now not important because we will not use Bellman equations.
Computing the gradient analytically

- The value of the policy $J(\theta)$ is:

$$J(\theta) = \mathbb{E}_{\pi_\theta} [R(\tau)] = \sum_{\tau} P(\tau|\theta) R(\tau)$$

where $P(\tau|\theta)$ denotes the probability of trajectory $\tau$ when following policy $\pi_\theta$

- Notice that sum is for all possible trajectories

- In this new notation, our goal is to find the policy parameters $\theta$ that:

$$\arg \max_{\theta} J(\theta) = \arg \max_{\theta} \sum_{\tau} P(\tau|\theta) R(\tau)$$
In general, assume we want to compute $\nabla \log f(x)$:

\[
\nabla \log f(x) = \frac{1}{f(x)} \nabla f(x)
\]

\[
f(x) \nabla \log f(x) = \nabla f(x)
\]

It can be applied to any function and we can use the equality in any direction.

The term $\frac{\nabla f(x)}{f(x)}$ is called *likelihood ratio* and is used to analytically compute the gradients.

Btw. Notice the caveat... *Assume policy is differentiable whenever it is non-zero.*
Computing the gradient analytically

- In this new notation, our goal is to find the policy parameters $\theta$ that:

$$\arg \max_{\theta} J(\theta) = \arg \max_{\theta} \sum_{\tau} P(\tau | \theta) R(\tau)$$

- So, taken the gradient wrt $\theta$

$$\nabla_{\theta} J(\theta) = \nabla_{\theta} \sum_{\tau} P(\tau | \theta) R(\tau)$$

$$= \sum_{\tau} \nabla_{\theta} P(\tau | \theta) R(\tau)$$

$$= \sum_{\tau} \frac{P(\tau | \theta)}{P(\tau | \theta)} \nabla_{\theta} P(\tau | \theta) R(\tau)$$

$$= \sum_{\tau} P(\tau | \theta) R(\tau) \frac{\nabla_{\theta} P(\tau | \theta)}{P(\tau | \theta)}$$

$$= \sum_{\tau} P(\tau | \theta) R(\tau) \nabla_{\theta} \log P(\tau | \theta)$$
Computing the gradient analytically

- Goal is to find the policy parameters $\theta$ that:
  \[
  \arg \max_{\theta} J(\theta) = \arg \max_{\theta} \sum_{\tau} P(\tau | \theta) R(\tau)
  \]

- So, taken the gradient wrt $\theta$
  \[
  \nabla_{\theta} J(\theta) = \sum_{\tau} P(\tau | \theta) R(\tau) \nabla_{\theta} \log P(\tau | \theta)
  \]

- Of course we cannot compute all trajectories...
Computing the gradient analytically

- Goal is to find the policy parameters $\theta$ that:

$$\arg\max_{\theta} J(\theta) = \arg\max_{\theta} \sum \tau P(\tau|\theta) R(\tau)$$

- So, taken the gradient wrt $\theta$

$$\nabla_{\theta} J(\theta) = \sum_{\tau} P(\tau|\theta) R(\tau) \nabla_{\theta} \log P(\tau|\theta)$$

- Of course we cannot compute all trajectories...but we can sample $m$ trajectories because of the form of the equation

$$\nabla_{\theta} J(\theta) \approx \frac{1}{m} \sum_{i=1}^{m} R(\tau_i) \nabla_{\theta} \log P(\tau_i|\theta)$$
Computing the gradient analytically: at last!

- Sample $m$ trajectories:

\[
\nabla_\theta J(\theta) \approx (1/m) \sum_{i=1}^{m} R(\tau_i) \nabla_\theta \log P(\tau_i|\theta)
\]

- However, we still have a problem, we don’t know the how to compute $\nabla_\theta \log P(\tau|\theta)$

- Fortunately, we can derive it from the stochastic policy

\[
\nabla_\theta \log P(\tau|\theta) = \nabla_\theta \log \left[ \mu(s_0) \prod_{i=0}^{T-1} \pi_\theta(a_i|s_i)P(s_{i+1}|s_i, a_i) \right]
\]

\[
= \nabla_\theta \left[ \log \mu(s_0) + \sum_{i=0}^{T-1} \log \pi_\theta(a_i|s_i) + \log P(s_{i+1}|s_i, a_i) \right]
\]

\[
= \sum_{i=0}^{T-1} \nabla_\theta \log \pi_\theta(a_i|s_i)
\]

No dynamics model required!
We assumed at the beginning that policy is differentiable and that we now the derivative wrt parameters $\theta$

So, we have the desired solution:

$$\nabla_\theta J(\theta) \approx \left(\frac{1}{m}\right) \sum_{i=1}^{m} R(\tau_i) \sum_{i=0}^{T-1} \nabla_\theta \log \pi_\theta(a_i|s_i)$$
Differentiable policies? Soft-max

- One popular way to do action selection instead of using $\epsilon$-greedy is to assign probabilities to actions according to values:

$$\pi(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a'} e^{Q(s,a')/\tau}} \propto e^{Q(s,a)/\tau}$$

where $\tau$ is parameter that controls exploration. Let’s assume $\tau = 1$

- Let’s consider the case where $Q(s, a) = \phi^T(s, a)\theta$ is approximated by a linear function

$$\nabla_\theta \log \pi_\theta(a_i|s_i) = \nabla_\theta \log \frac{e^{\phi^T(s,a)\theta}}{\sum_{a'} e^{\phi^T(s,a')\theta}}$$

$$= \nabla_\theta \phi^T(s, a)\theta - \nabla_\theta \sum_{a'} \phi^T(s, a')\theta$$

$$= \phi(s, a) - \mathbb{E}_{\pi_\theta} [\phi(s, \cdot)]$$
Differentiable policies? Gaussian Policy

- **In continuous spaces of actions**, action is generated by a random distribution with parameters (f.i. Gaussian distribution).
- Parameter of the Gaussian is a linear combination of feature vector \( \mu = \phi^T(s) \theta \). Variance \( \sigma \) can be fixed or approximated.
- This approach allows to consider actions vectors of continuous values (two actions same time!).
- Policy select actions following Gaussian distribution:

\[
\begin{align*}
a & \sim \mathcal{N}(\mu_\theta(s), \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(a-\phi^T(s)\theta)^2}{2\sigma^2}} \\
\n\n\end{align*}
\]

- In this case,

\[
\nabla_\theta \log \pi_\theta(a|s) = \nabla_\theta \left[ -\frac{(a-\phi^T(s)\theta)^2}{2\sigma^2} \right] = \frac{(a-\phi^T(s)\theta)}{\sigma^2} \phi(s)
\]
Differentiable policies? Deep Neural Network

- A very popular way to approximate the policy is to use a Deep NN with soft-max last layer with so many neurons as actions.
- In this case, use *autodiff* of the neural network package you use! In tensorflow:

  ```
  loss = - tf.reduce_mean(tf.log(prob_outputs) * reward)
  ```

  where `prob_outputs` is the output layer of the DNN
- Backpropagation implemented will do the work for you.
- Common approaches:
  - Last *softmax* layer in discrete case
  - Last layer with $\mu$ and log $\sigma$ in continuous case
Vanilla Policy Gradient

Given architecture with parameters $\theta$ to implement $\pi_\theta$
Initialize $\theta$ randomly
repeat
  Generate episode $\{s_1, a_1, r_2, \ldots s_{T-1}, a_{T-1}, r_T, s_T\} \sim \pi_\theta$
  Get $R \leftarrow$ long-term return for episode
  for all time steps $t = 1$ to $T - 1$ do
    $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t)R$
  end for
until convergence

Substitute $\nabla_\theta \log \pi_\theta(a_t|s_t)$ with appropriate equation.
Btw, notice no explicit exploration mechanism needed when policies are stochastic (all on policy)!
Vanilla Policy Gradient

- Remember:

\[
\nabla_\theta J(\theta) \approx \frac{1}{m} \sum_{i=1}^{m} R(\tau_i) \sum_{i=0}^{T-1} \nabla_\theta \log \pi_\theta(a_i|s_i)
\]

- Unbiased but very noisy

- Fixes that can make it practical
  - Temporal structure
  - Baseline
Subsection 3

Reduce variance using temporal structure: Reinforce and Actor-Critic architectures
Policy Gradient using Temporal structure

• Instead on focusing on reward of trajectories,

\[ J(\theta) = \mathbb{E}_{\pi_{\theta}} [R(\tau)] = \sum_{\tau} P(\tau|\theta)R(\tau) \]

• We want to optimize the expected return

\[ J_{avV}(\theta) = \sum_{s} d^{\pi_{\theta}}(s)V(s) = \sum_{s} d^{\pi_{\theta}} \sum_{a} \pi_{\theta}(a|s)Q(s, a) \]

where \( d^{\pi_{\theta}}(s) \) is the expected number of time steps on \( s \) in a randomly generated episode following \( \pi_{\theta} \) divided by time steps of trial

• Let’s start with an MDP with one single step.

\[ J_{avR}(\theta) = \sum_{s} d^{\pi_{\theta}}(s)V(s) = \sum_{s} d^{\pi_{\theta}} \sum_{a} \pi_{\theta}(a|s)r(s, a) \]
Policy Gradient using Temporal structure

\[ J_{avR}(\theta) = \sum_s d^{\pi_\theta} \sum_a \pi_\theta(a|s)r(s, a) \]

\[ \nabla_\theta J_{avR}(\theta) = \nabla_\theta \sum_s d^{\pi_\theta} \sum_a \pi_\theta(a|s)r(s, a) \]

\[ = \sum_s d^{\pi_\theta} \sum_a \nabla_\theta (\pi_\theta(a|s)r(s, a)) \]

\[ = \sum_s d^{\pi_\theta} \sum_a \nabla_\theta \pi_\theta(a|s) \log \pi_\theta(a|s)r(s, a) \]

\[ = E_{\pi_\theta} [\nabla_\theta \log \pi_\theta(a|s)r(s, a)] \]

- And this expectation can be sampled
The policy gradient theorem generalize the likelihood ratio approach to multi–step MDPs.

- Replaces instantaneous reward $r$ with long–term value $Q(s, a)$.
- Policy gradient theorem applies to all objective functions we have seen.

**Policy gradient theorem**

For any differentiable policy $\pi_\theta(s, a)$, for any of the policy objective functions $J = J_1, J_{avR}$ or $J_{avV}$, the policy gradient is:

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

Simple proof in pag. 325 of (Sutton 2018)
REINFORCE algorithm

- REINFORCE algorithm (also called Monte–Carlo Policy Gradient) uses reward $R$ as an unbiased sample of $Q^{\pi_{\theta}}(s, a)$.

### REINFORCE algorithm

Given architecture with parameters $\theta$ to implement $\pi_{\theta}$

Initialize $\theta$ randomly

repeat

Generate episode $\{s_1, a_1, r_2, \ldots s_{T-1}, a_{T-1}, r_T, s_T\} \sim \pi_{\theta}$

for all time steps $t = 1$ to $T - 1$ do

Get $R_t \leftarrow$ long-term return from step $t$ to $T^a$

$\theta \leftarrow \theta + \alpha \nabla_{\theta} \log \pi_{\theta}(a_t|s_t)R_t$

end for

until convergence

---

$^a$See proof from Don’t Let the Past Distract You if you are not convinced.
REINFORCE algorithm

Let’s analyze the update:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t) R_t$$

Let’s us rewrite is as follows

$$\theta \leftarrow \theta + \alpha \frac{\nabla_\theta \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)} R_t$$

Update is proportional to:

- the product of a return $R_t$ and
- the gradient of the probability of taking the action actually taken,
- divided by the probability of taking that action.
REINFORCE algorithm

- Update:

\[ \theta \leftarrow \theta + \alpha \frac{\nabla_{\theta} \pi_{\theta}(a_t|s_t)}{\pi_{\theta}(a_t|s_t)} R_t \]

- ...move most in the directions that favor actions that yield the highest return
- ...is inversely proportional to the action probability (actions that are selected frequently are at an advantage (the updates will be more often in their direction))

- Is it necessary to change something in the algorithm for continuous actions?

Mario Martin (CS-UPC) Reinforcement Learning May 7, 2020 43 / 72
REINFORCE algorithm

- Update:

\[ \theta \leftarrow \theta + \alpha \frac{\nabla_{\theta} \pi_{\theta}(a_t|s_t)}{\pi_{\theta}(a_t|s_t)} R_t \]

- ...move most in the directions that favor actions that yield the highest return
- ...is inversely proportional to the action probability (actions that are selected frequently are at an advantage (the updates will be more often in their direction))

- Is it necessary to change something in the algorithm for continuous actions?
- No! Just uses a continuous action policy mechanism and everything is the same!
Monte-Carlo policy gradient still has high variance because $R_t$ has a lot of variance.

We can reduce variance subtracting a baseline to the estimator

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t)(R_t - b(s_t))$$

without introducing any bias when baseline does not depend on actions taken.

A good baseline is $b(s_t) = V^{\pi_\theta}(s_t)$ so we will use that.
REINFORCE algorithm with baseline

- Monte-Carlo policy gradient still has high variance because $R_t$ has a lot of variance.
- We can reduce variance subtracting a baseline to the estimator:
  \[
  \theta \leftarrow \theta + \alpha \nabla_{\theta} \log \pi_{\theta}(a_t|s_t)(R_t - b(s_t))
  \]

  - without introducing any bias when baseline does not depend on actions taken.
  - A good baseline is $b(s_t) = V^{\pi_{\theta}}(s_t)$ so we will use that.

How to estimate $V^{\pi_{\theta}}$?

We'll use another set of parameters $w$ to approximate.
REINFORCE algorithm with baseline

Given architecture with parameters $\theta$ to implement $\pi_\theta$ and parameters $w$ to approximate $V$

Initialize $\theta$ randomly

repeat

Generate episode $\{s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T, s_T\} \sim \pi_\theta$

for all time steps $t = 1$ to $T - 1$ do

Get $R_t \leftarrow$ long-term return from step $t$ to $T$

$\delta \leftarrow R_t - V_w(s_t)$

$w \leftarrow w + \beta \delta \nabla_w V_w(s_t)$

$\theta \leftarrow \theta + \alpha \delta \nabla_\theta \log \pi_\theta(a_t|s_t)$

end for

until convergence
• Monte-Carlo policy gradient has high variance
• So we used a baseline to reduce the variance \( R_t - V(s_t) \)
• Can we do something to speed up learning like we did with MC using TD?
Monte-Carlo policy gradient has high variance
So we used a baseline to reduce the variance $R_t - V(s_t)$
Can we do something to speed up learning like we did with MC using TD?
Yes, **use different estimators of** $R_t$ **that do bootstrapping** f.i. TD(0), n-steps, etc.
These algorithms are called **Actor Critic**
Actor-Critic Architectures

- The **Critic**, *evaluates the current policy* and the result is used in the policy training.
- The **Actor** *implements the policy* and is trained using Policy Gradient with estimations from the critic.
Actor-Critic Architectures

- Actor-critic algorithms maintain two sets of parameters (like in REINFORCE with baseline):
  - **Critic** parameters: approximation parameters $w$ for action-value function under current policy
  - **Actor** parameters: policy parameters $\theta$

- Actor-critic algorithms follow an approximate policy gradient:
  - **Critic**: Updates action-value function parameters $w$ like in *policy evaluation* updates (you can apply everything we saw in FA for prediction)
  - **Actor**: Updates policy gradient $\theta$, in direction suggested by critic
Actor-Critic Architectures

- Actor updates are always in the same way:

\[ \theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t) G_t \]

where \( G_t \) is the evaluation of long-term returned by the critic for \( s_t \)

- Critic updates are done to evaluate the current policy

\[ w \leftarrow w + \alpha \delta \nabla_\theta V_w(a_t|s_t) \]

where \( \delta \) is the estimated error in evaluating the \( s \) state and that implements the kind of bootstrapping done.
One step Actor Critic (QAC)

One step actor-critic: \( \delta \leftarrow r + Q_w(s', a') - Q_w(s, a) \)

One step Actor Critic

Given architecture with parameters \( \theta \) to implement \( \pi_\theta \) and parameters \( w \) to approximate \( Q \)

Initialize \( \theta \) randomly

repeat
  Set \( s \) to initial state
  Get \( a \) from \( \pi_\theta \)
  repeat
    Take action \( a \) and observe reward \( r \) and new state \( s' \)
    Get \( a' \) from \( \pi_\theta \)
    \( \delta \leftarrow r + Q_w(s', a') - Q_w(s, a) \) // TD-error (Bellman equation)
    \( w \leftarrow w + \beta \delta \nabla_w Q_w(s, a) \) // critic update
    \( \theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a|s) Q_w(s, a) \) // Actor update
    \( s \leftarrow s' \)
  until \( s \) is terminal
until convergence
Advantage Actor Critic (AAC or A2C)

In this critic Advantage value function is used:

\[
A^\pi_\theta(s, a) = Q^\pi_\theta(s, a) - V^\pi_\theta(s)
\]

The advantage function can significantly reduce variance of policy gradient

So the critic should really estimate the advantage function, for instance, estimating both \(V(s)\) and \(Q\) using two function approximators and two parameter vectors:

\[
V^\pi_\theta(s) \approx V_v(s) \quad \text{(1)}
\]
\[
Q^\pi_\theta(s, a) \approx Q_w(s, a) \quad \text{(2)}
\]
\[
A(s, a) = Q_w(s, a) - V_v(s) \quad \text{(3)}
\]

And updating both value functions by e.g. TD learning

Nice thing, you only punish policy when not optimal (why?)
From A2C to REINFORCE with baseline

- One way to implement A2C method without two different networks to estimate $Q_w(s, a)$ and $V_v(s)$ is the following.
- For the true value function $V^\pi_{\theta}(s)$, the TD error $\delta^\pi_{\theta}(s)$

$$\delta^\pi_{\theta}(s) = r + \gamma V^\pi_{\theta}(s') - V^\pi_{\theta}(s)$$

...that it is an unbiased estimate of the advantage function:

$$\mathbb{E}^\pi_{\theta} [\delta^\pi_{\theta}|s, a] = \mathbb{E}^\pi_{\theta} [r + \gamma V^\pi_{\theta}(s')|s, a] - V^\pi_{\theta}(s) = Q^\pi_{\theta}(s, a) - V^\pi_{\theta}(s) = A^\pi_{\theta}(s, a)$$

- So we can use the TD error to compute the policy gradient

$$\nabla_{\theta} J(\theta) = \mathbb{E}^\pi_{\theta} [\nabla_{\theta} \log \pi_{\theta}(a|s) \delta^\pi_{\theta}]$$

- In practice this approach only requires one set of critic parameters $v$ to approximate TD error

$$\delta_v = r + \gamma V_v(s') - V_v(s)$$

- Notice this algorithm resemblance with REINFORCE with baseline
Asynchronous Advantage Actor Critic (A3C)

- **A3C** (Mnih et al. 2016) idea: Sample for data can be parallelized using several copies of the same agent
  - use $N$ copies of the agents (workers) working in parallel collecting samples and computing gradients for policy and value function
  - After some time, pass gradients to a main network that updates actor and critic using the gradients of all
  - After some time the worker copy the weights of the global network

- This parallelism decorrelates the agents’ data, so **no Experience Replay Buffer needed**

- Even one can explicitly use different exploration policies in each actor-learner to maximize diversity

- Asynchronism can be extended to other update mechanisms (Sarsa, Q-learning...) but it works better in Advantage Actor critic setting
Generalized Advantage Estimator (GAE)

- Generalized Advantage Estimator (Schulman et al. 2016). [nice review]
- Use a version of Advantage that consider weighted average of n-steps estimators of advantage like in TD($\lambda$):

$$A_G^{\pi} = \sum_{t'=t}^{\infty} (\lambda \gamma)^{t'-t} \left[ r_{t'+1} + \gamma V_\theta^{\pi}(s_{t'+1}) - V_\theta^{\pi}(s_t) \right]$$

- Used in continuous setting for locomotion tasks
Subsection 4

Conclusions and other approaches
The policy gradient has many equivalent forms

\[
\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(a|s) R_t] \quad \text{REINFORCE}
\]

\[
= \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(a|s) Q_w(s, a)] \quad \text{Actor-Critic}
\]

\[
= \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(a|s) A_w(s, a)] \quad \text{Advantage Actor-Critic}
\]

\[
= \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(a|s) \delta] \quad \text{TD Actor-Critic}
\]

\[
= \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(a|s) \delta e] \quad \text{TD(\lambda) Actor-Critic}
\]

Each leads a stochastic gradient ascent algorithm

Critic uses \textbf{policy evaluation} (e.g. MC or TD learning) to estimate \(Q^\pi(s, a), A^\pi(s, a)\) or \(V^\pi(s)\)
Approximating the policy gradient with critic can introduce bias. A biased policy gradient may not find the right solution. Luckily, if we choose value function approximation carefully, then we can avoid bias.

If the following two conditions are satisfied:

1. Value function approximator is compatible to the policy
   \[ \nabla_w Q_w(s, a) = \nabla_\theta \log \pi_\theta(a|s) \]

2. Value function parameters \(w\) minimize the mean-squared error
   \[ \nabla_w E_{\pi_\theta} [(Q^{\pi_\theta}(s, a) - Q_w(s, a))^2] = 0 \]

Then the policy gradient is **without bias**
Problems with Policy Gradient Directions

- **Goal:** Each step of policy gradient yields an updated policy $\pi'$ whose value is greater than or equal to the prior policy $\pi$: $V^{\pi'} \geq V^{\pi}$
- **Several inefficiencies:**
  - Gradient ascent approaches update the weights a **small step** in direction of gradient
  - Gradient ascent algorithms can follow *any* ascent direction (a good ascent direction can significantly **speed** convergence)
  - Gradient is First order / linear approximation of the value function’s dependence on the **policy parameterization** instead of **actual policy**

---

1A policy can often be re–parameterized without changing action probabilities (f.i., increasing score of all actions in a softmax policy). Vanilla gradient is sensitive to these re–parameterizations.
About step size

- Step size is important in any problem involving finding the optima of a function.
- Supervised learning: Step too far $\rightarrow$ next updates will fix it.
- But in Reinforcement learning:
  - Step too far $\rightarrow$ bad policy.
  - Next batch: collected under bad policy.
  - **Policy is determining data collect!** Essentially controlling exploration and exploitation trade-off due to particular policy parameters and the stochasticity of the policy.
  - May not be able to recover from a bad choice, collapse in performance!
A more efficient gradient in learning problems is the **natural gradient**

It corresponds to **steepest ascent in policy space and not in the parameter space** with **right step size**

Also, the natural policy gradient is **parametrization independent**

Convergence to a local minimum is guaranteed

It finds ascent direction that is closest to vanilla gradient, when changing policy by a small, fixed amount

\[
\nabla_{\theta}^{\text{nat}} \pi_{\theta}(a|s) = G_{\theta}^{-1} \nabla_{\theta} \pi_{\theta}(a|s)
\]

Where \( G_{\theta} \) is the Fisher information matrix

\[
G_{\theta} = \mathbb{E}_{\pi_{\theta}} \left[ \nabla_{\theta} \log \pi_{\theta}(a|s) \nabla_{\theta} \log \pi_{\theta}(a|s)^T \right]
\]
Natural Actor Critic (Peters et al 2005)

- Under linear model modelization of critic:
  \[ A^{\pi_\theta}(s, a) = \phi(s, a)^T w \]

- Using compatible function approximation,
  \[ \nabla_w A_w(s, a) = \nabla_\theta \log \pi_\theta(a|s) \]

- The natural policy gradient nicely simplifies,
  \[ \nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \left[ \nabla_\theta \log \pi_\theta(a|s) A^{\pi_\theta}(s, a) \right] \]
  \[ = \mathbb{E}_{\pi_\theta} \left[ \nabla_\theta \log \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s)^T w \right] \]
  \[ = G_\theta w \]
  \[ \nabla^{nat}_\theta J(\theta) = w \]

- i.e. update actor parameters in direction of critic parameters
**Trust Region Policy Optimization (TRPO)** maximize parameters that change the policy increasing advantage in action over wrt. old policy in proximal spaces to avoid too large step size.

\[
\arg \max_{\theta} L_{\theta_{old}}(\theta) = \arg \max_{\theta} \mathbb{E}_{s_0:\infty} \left[ \sum_{t=0}^{T-1} \mathbb{E}_{a \sim \theta} \left[ \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} A_\theta(s_t, a_t) \right] \right]
\]

- Under penalizing constraint (using KL divergence of \(\theta\) and \(\theta_{old}\)) that ensures improvement of the policy in the proximity (small step size)
- Solves using Natural Gradient
Trust Region Policy Optimization (TRPO) maximize parameters that change the policy increasing advantage in action over wrt. old policy in proximal spaces to avoid too large step size.

\[
\arg \max_{\theta} L_{\theta_{old}}(\theta) = \arg \max_{\theta} \mathbb{E}_{s_0:\infty} \left[ \sum_{t=0}^{T-1} \mathbb{E}_{a \sim \theta} \left[ \frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} A_{\theta}(s_t, a_t) \right] \right]
\]

Under penalizing constraint (using KL divergence of \(\theta\) and \(\theta_{old}\)) that ensures improvement of the policy in the proximity (small step size)

Solves using Natural Gradient

Some TRPO videos [here](#).

Proximal Policy Optimization (PPO) inspired in TRPO simplifies computation
Subsection 5

New off-policy AC methods
DDPG is an extension of Q-learning for **continuous action spaces**.

- Therefore, it is an **off-policy algorithm** (we can use ER!)

- It is also an **actor-critic** algorithm (has networks $Q_\phi$ and $\pi_\theta$.)

- Uses $Q$ and $\pi$ **target** networks for stability.

- Differently from other critic algorithms, **policy is deterministic**, noise added for exploration:
  
  $$a_t = \pi_\theta(s_t) + \epsilon \text{ (where } \epsilon \sim \mathcal{N})$$
• \( Q_\phi \) network is trained using standard loss function:

\[
L(\phi, \mathcal{D}) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}} \left[ \left( Q_\phi(s, a) - (r + \gamma Q_{\phi_{\text{targ}}}(s', \pi_{\theta_{\text{targ}}}(s'))) \right)^2 \right]
\]

• As action is \textit{deterministic} and \textit{continuous} (NN), we can easily follow the gradient in policy network to increase future reward:

\[
\max_{\theta} \mathbb{E}_{s \sim \mathcal{D}} [Q_\phi(s, \pi_\theta(s))] \rightarrow \nabla_\theta \mathbb{E}_{s \sim \mathcal{D}} [Q_\phi(s, \pi_\theta(s))] \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_a Q_\phi(s, a) \nabla_\theta \pi_\theta(s)
\]
Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$

for episode = 1, M do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $s_1$
  for $t = 1, T$ do
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
    Set $y_i = r_i + \gamma Q'(s_{i+1}; \mu'(s_{i+1}|\theta^{\mu'}))|\theta^{Q'}$
    Update critic by minimizing the loss: $L = \frac{1}{N} \sum_{i} (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:
    \[
    \nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_{i} \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}
    \]
  end for
end for

Update the target networks:
\[
\theta^{Q'} \leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'}
\]
\[
\theta^{\mu'} \leftarrow \tau \theta^\mu + (1 - \tau) \theta^{\mu'}
\]
TD3: Twin Delayed DDPG (Fujimoto et al, 2018)

- Similar to DDPG but with the following changes:
  1. **Clipped action exploration**: noise added like DDPG but noise bounded to fixed range.

  \[
  a'(s') = \text{clip} \left( \pi_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{\text{Low}}, a_{\text{High}} \right), \quad \epsilon \sim \mathcal{N}(0, \sigma)
  \]

  2. **Pessimistic Double-Q Learning**: It uses two (twin) Q networks and uses the "pessimistic" one for current state for updating the networks

  \[
  L(\phi_i, D) = \mathbb{E}_{(s,a,r,s') \sim D} \left( Q_{\phi_i}(s, a) - \min_{i=1,2} Q_{\phi_i,\text{targ}}(s', a'(s')) \right)^2
  \]

  3. **Delayed Policy Updates**: Updates of Critic are more frequent than of policy (fi. 2 or 3 times)
SAC: Soft Actor Critic (Haarnoja et al, 2018)

- Policy Entropy-regularized: we will look for maximum entropy policies with given data (in SAC we go back to stochastic \( \pi \)).

\[
\mathcal{H}(\pi(\cdot|s)) = \mathbb{E}_{a \sim \pi(s)} [- \log \pi(a|s)]
\]

- So we search for policy:

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

where \( \alpha \) is the trade-off between reward and entropy.

- Entropy enforces exploration, so no need to add noise to actions.

- Usually \( \alpha \) decreases during learning and is disabled to test performance.
Let’s define value functions in this case:

\[ V^\pi(s) = \mathbb{E}_{\tau \sim \pi} \left[ \sum_{t=0}^{\infty} \gamma^t \left( R(s_{t+1}) + \alpha \mathcal{H}(\pi(\cdot|s_t)) \right) \right| s_0 = s \]

\[ Q^\pi(s, a) = \mathbb{E}_{\tau \sim \pi} \left[ \sum_{t=0}^{\infty} \gamma^t R(s_{t+1}) + \alpha \sum_{t=1}^{\infty} \gamma^t \mathcal{H}(\pi(\cdot|s_t)) \right| s_0 = s, a_0 = a \]

So Bellman equations can be written as:

\[ V^\pi(s) = \mathbb{E}_{\tau \sim \pi} \left[ Q^\pi(s, a) + \alpha \mathcal{H}(\pi(\cdot|s)) \right] \]

\[ Q^\pi(s, a) = \mathbb{E}_{s' \sim P, a' \sim \pi} \left[ R(s') + \gamma \left( Q^\pi(s', a') + \alpha \mathcal{H}(\pi(\cdot|s')) \right) \right] \]

\[ = \mathbb{E}_{s' \sim P} \left[ R(s, a, s') + \gamma V^\pi(s') \right] \]
SAC: Soft Actor Critic (Haarnoja et al, 2018)

- Architecture: Networks and loss functions for each one:
  - Q-value functions: $Q_{\theta_1}(s, a), Q_{\theta_2}(s, a)$ (twin like TD3)
    
    $$L(\theta_i, \mathcal{D}) = \mathbb{E}_{(s,a,r,s',d) \sim \mathcal{D}} \left[ \left( Q_{\theta_i}(s, a) - (r + \gamma V_{\psi_{\text{targ}}}(s')) \right)^2 \right]$$

  - Value functions $V_{\psi}(s), V_{\psi_{\text{targ}}}(s)$:
    
    $$L(\psi, \mathcal{D}) = \mathbb{E}_{s \sim \mathcal{D}, a \sim \pi_{\phi}} \left( V_{\psi}(s) - \left( \min_{i=1,2} Q_{\theta_i}(s, a) - \alpha \log \pi_{\phi}(a|s) \right) \right)^2$$

  - Policy $\pi_{\phi}(a|s)$. Maximize:
    
    $$\mathbb{E}_{a \sim \pi} \left( Q^\pi(s, a) - \alpha \log \pi(a|s) \right)$$

    which maximize $V$ value function... but how to compute gradients?
Reparameterization trick see here or here

- Problematic because in $\nabla_\phi$, expectation follow stochastic $\pi_\phi$.
  $$E_{a \sim \pi_\phi} [Q^{\pi_\phi}(s, a) - \alpha \log \pi_\phi(a|s)]$$

- It can be done using the log-trick like REINFORCE... but high variance.

- Authors use a reparameterization trick. It can be done when we define the stochastic $\pi_\phi$ as Gaussian by adding noise to the action:
  $$\tilde{a}_\phi(s, \xi) = \tanh(\mu_\phi(s) + \sigma_\phi(s) \odot \xi), \quad \xi \sim \mathcal{N}(0, I)$$

- Now we can rewrite the term as:
  $$E_{a \sim \pi_\phi} [Q^{\pi_\phi}(s, a) - \alpha \log \pi_\phi(a|s)] = E_{\xi \sim \mathcal{N}} [Q^{\pi_\phi}(s, \tilde{a}_\phi(s, \xi)) - \alpha \log \pi_\phi(\tilde{a}_\phi(s, \xi)|s)]$$

- Now we can optimize the policy according to
  $$\max_{\phi} E_{s \sim \mathcal{D}, \xi \sim \mathcal{N}} [Q_{\theta_1}(s, \tilde{a}_\phi(s, \xi)) - \alpha \log \pi_\phi(\tilde{a}_\phi(s, \xi)|s)]$$
Recommended resources

- Nice review of Policy Gradient Algorithms in Lil’Log blog
- Good description of algorithms in Spinning Up with implementation in Pytorch and Tensorflow
- Understable implementations of Actor Critic methods in RL-Adventure-2