# ATCI: Reinforcement Learning

## Policy Search: Actor-Critic and Gradient Policy search

Mario Martin

CS-UPC

March 13, 2024

# Goal of this lecture

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

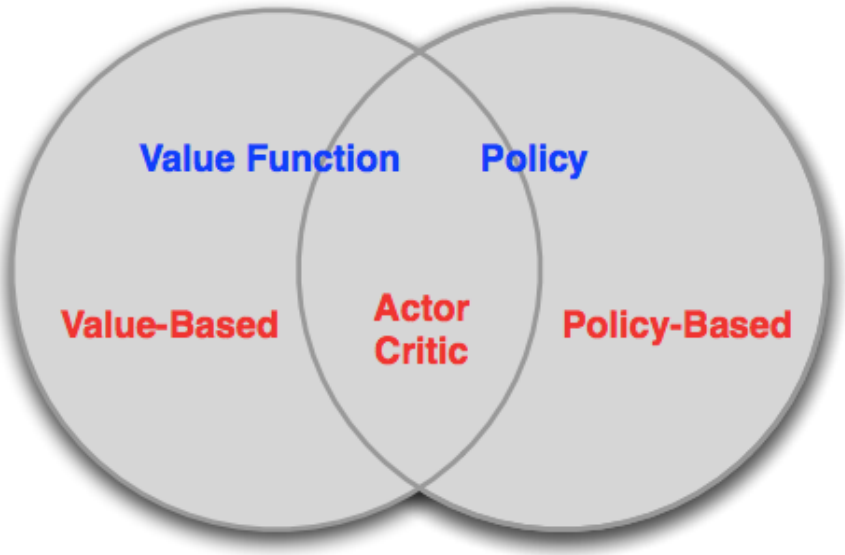
$$\begin{aligned} V_\theta &\approx V^\pi \\ Q_\theta(s,a) &\approx V^\pi(s) \end{aligned}$$

- 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**

# Advantges 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**

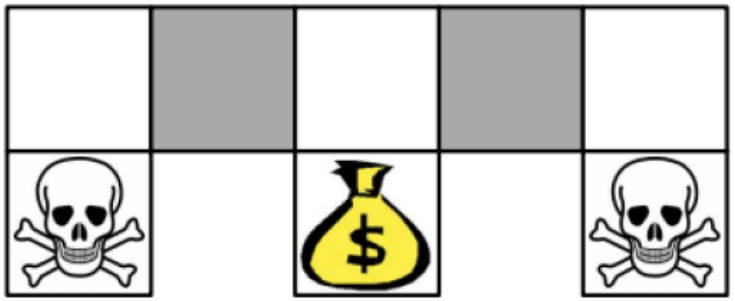
# Stochastic Policies

- 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...

# 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... 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?

# Stochastic Policies when *aliased* states (POMDPs)



- The agent *cannot differentiate* the grey states
- Consider features of the following form:

$$\phi_d(s) = 1(\text{wall to } d) \quad \forall d \ \in \{N, E, S, W\}$$

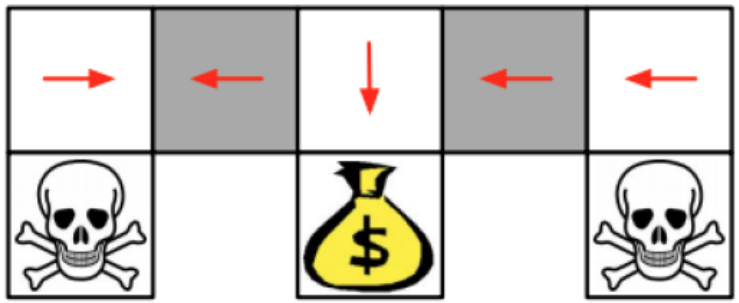
- Compare value-based RL, using an approximate value function

$$Q_\theta(s, a) = f_\theta(\phi(s, a))$$

- To policy-based RL, using a parametrized policy

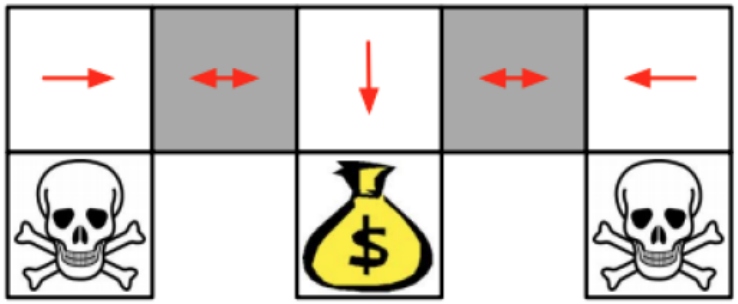
$$\pi_\theta(a|s) = g_\theta(\phi(s, a))$$

# 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
  - $\pi_\theta(\text{move E} \mid \text{wall to N and S}) = 0.5$
  - $\pi_\theta(\text{move W} \mid \text{wall to N and S}) = 0.5$
- It will reach the goal state in a few steps with high probability
- Policy–based RL can learn the optimal stochastic policy

# Policy optimization

# Policy Objective Functions

- 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**

$$J_1(\theta) = V^{\pi_\theta}(s_1)$$

# Policy Objective Functions

- 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

# Policy optimization

- 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

# 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)

# 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)

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

Notice! No value functions!

# Tabular Cross-Entropy

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

# Tabular Cross-Entropy

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)

**Approximated Cross-Entropy Method**

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
  $\theta = \theta + \alpha \nabla \left[ \sum_{s,a \in elite} \log \pi_\theta(a|s) \right]$
**until** convergence
**return** $\pi_\theta$

# Approximated Cross-Entropy Method (CEM)

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

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

  so gradient goes in that direction
- Intuitively, is the policy that maximizes similarity with behavior of successful samples [Notice this is Cross-Entropy loss of output of NN and actions of the *elite*.]
- I promised no gradient, but notice that gradient is for the approximation, not for the rewards of the policy

# Approximated Cross-Entropy Method (CEM)

- Another popular implementation of CE methods consist is approximating the policy using a Neural Network where weights have a value and a standard deviation. It is initialized randomly with large standard deviation.
- Repeat until convergence:
  1. A population of $N$ neural networks are a created from the policy by **sampling** the weights from the stochastic neural network
  2. Each of the $N$ networks is evaluated and a elite of best $M$ members is selected.
  3. Weights of the stochastic neural network are recomputed from statistics from the Elite
- Instead of approximating probabilities of action of the elite, we approximate the weights
- Simple to implement and effective

# 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:

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

- $V^{\pi_0} \xrightarrow{\text{small change}} \pi_1 \xrightarrow{\text{large change}} V^{\pi_1} \xrightarrow{\text{small change}} \pi_2 \xrightarrow{\text{large change}} V^{\pi_2}$
- 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**

# Computing the gradient analytically

- 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 works also for discounted returns.

# Computing the gradient analytically

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

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

  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:

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

# [Log-trick: a convenient equality]

- In general, assume we want to compute $\nabla \log f(x)$ :

$$\begin{aligned} \nabla \log f(x) &= \frac{1}{f(x)} \nabla f(x) \\ f(x) \nabla \log f(x) &= \nabla f(x) \end{aligned}$$

- 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 $\theta$

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \sum_{\tau} P(\tau|\theta) R(\tau) \\
&= \sum_{\tau} \nabla_\theta P(\tau|\theta) R(\tau) \\
&= \sum_{\tau} \frac{P(\tau|\theta)}{P(\tau|\theta)} \nabla_\theta P(\tau|\theta) R(\tau) \\
&= \sum_{\tau} P(\tau|\theta) R(\tau) \frac{\nabla_\theta P(\tau|\theta)}{P(\tau|\theta)} \\
&= \sum_{\tau} P(\tau|\theta) R(\tau) \nabla_\theta \log P(\tau|\theta)
\end{aligned}$$

# 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

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

# 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

$$\begin{aligned}
\nabla_\theta \log P(\tau|\theta) &= \nabla_\theta \log \left[ \mu(s_0) \prod_{i=0}^{T-1} \pi_\theta(a_i|s_i) P(s_{i+1}|s_i, a_i) \right] \\
&= \nabla_\theta \left[ \log \mu(s_0) + \sum_{i=0}^{T-1} \log \pi_\theta(a_i|s_i) + \log P(s_{i+1}|s_i, a_i) \right] \\
&= \sum_{i=0}^{T-1} \underbrace{\nabla_\theta \log \pi_\theta(a_i|s_i)}_{\text{No dynamics model required!}}
\end{aligned}$$

# Computing the gradient analytically

- We assumed at the beginning that policy is differentiable and that we now the derivative wrt parameters $\theta$
- So, we have the desired solution:

$$\nabla_\theta J(\theta) \approx (1/m) \sum_{i=1}^{m} \left( R(\tau_i) \sum_{(s_j,a_j)\in\tau_i} \nabla_\theta \log \pi_\theta(a_j|s_j) \right)$$

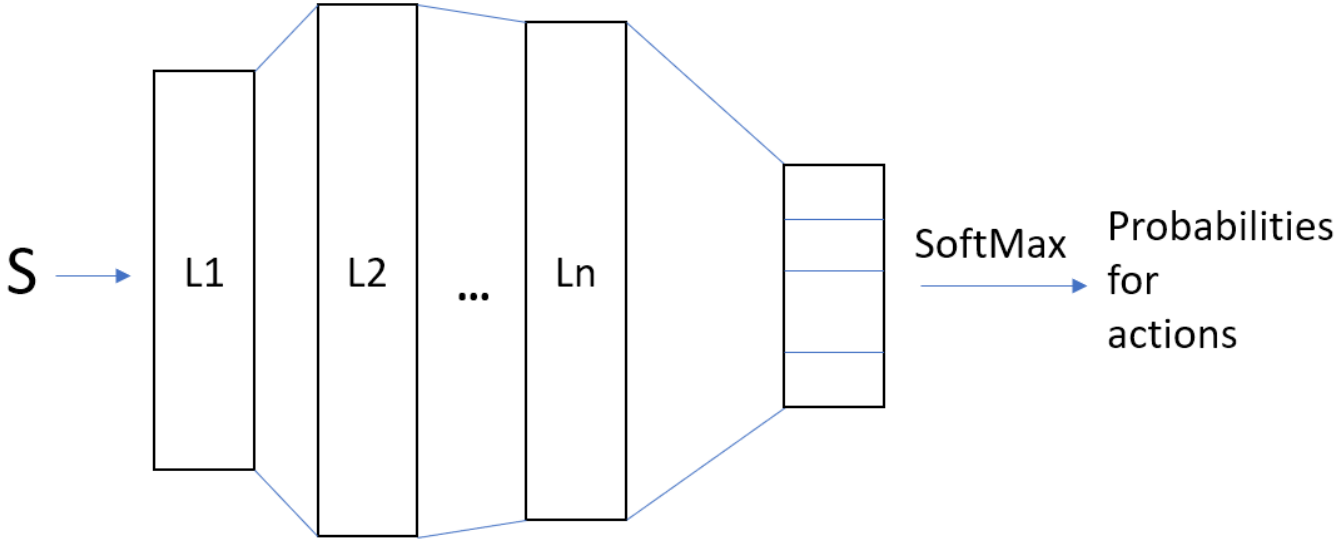
# Differentiable policies? Deep Neural Network

- 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 `pytorch`:

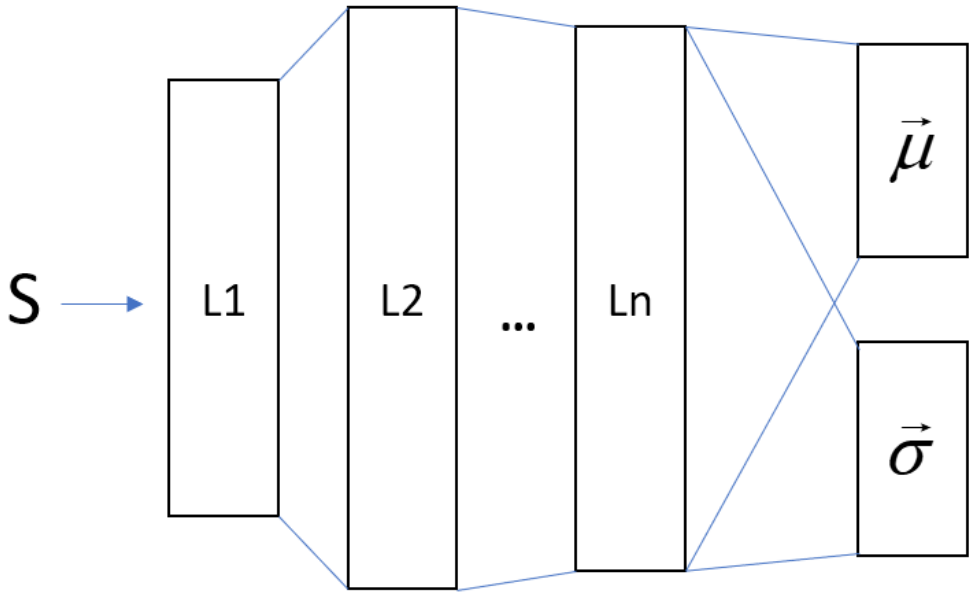
```
loss = - torch.mean(log_outputs * R)
```

  where `prob_outputs` is the output layer of the DNN and `R` the long term reward.
- Backpropagation is implemented in `pytorch` and will do the work for you!
- Common approaches for stochastic policies:
  - Last *softmax* layer in discrete case
  - Last layer with $\mu$ and $\log \sigma$ in continuous case

# Discrete action space

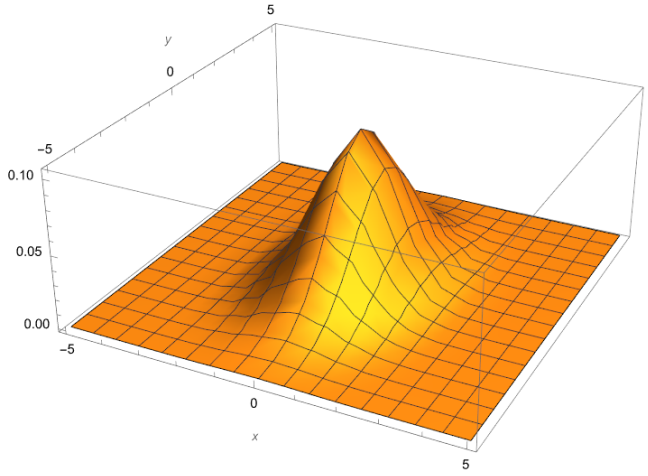


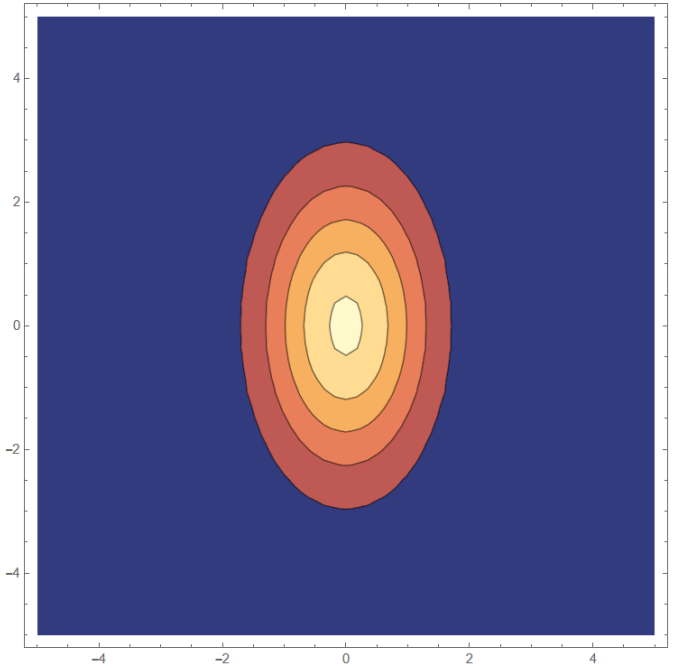
# Continuous action space



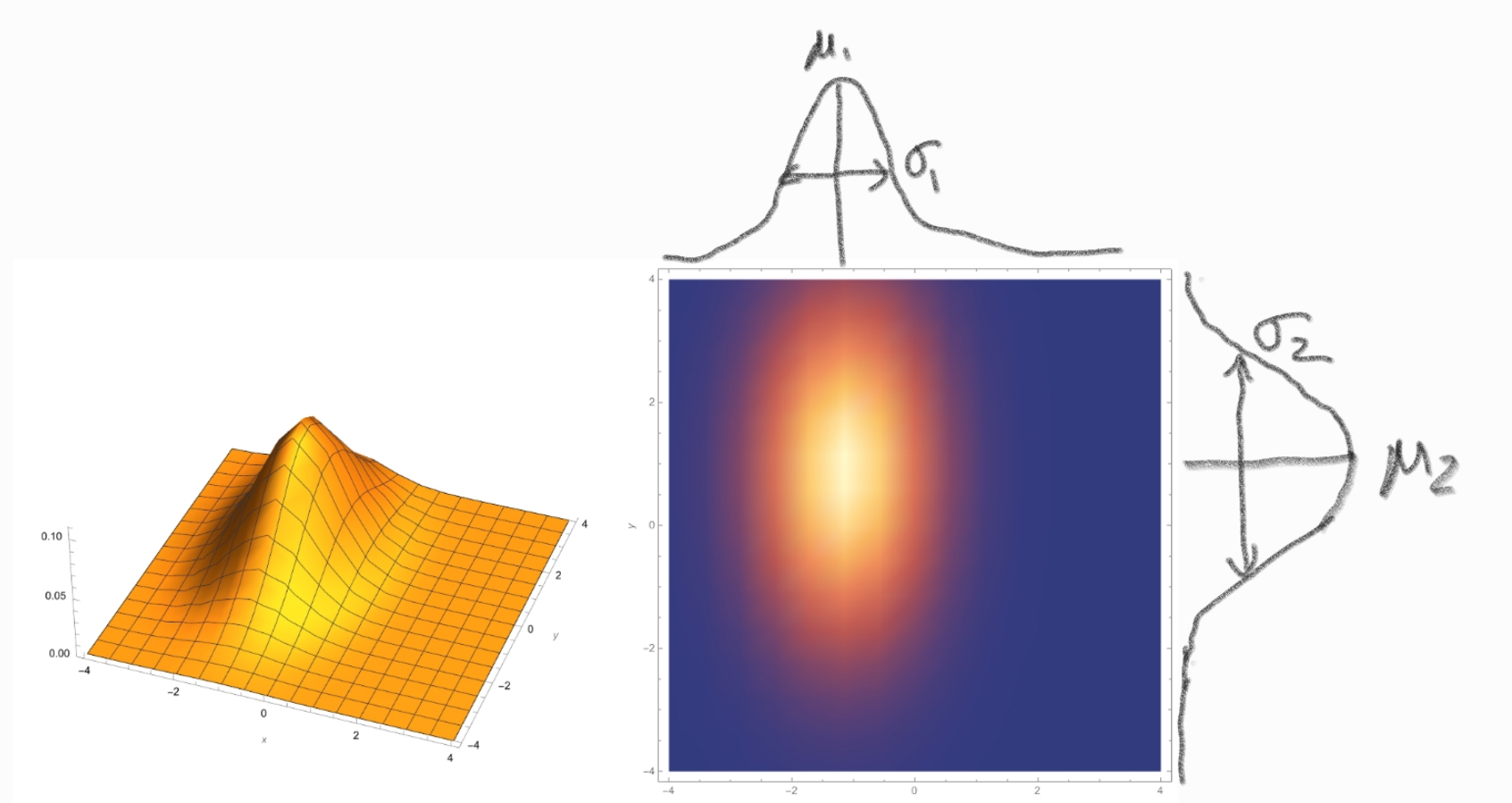

Define diagonal multivariate Gaussian probability distribution

# Continuous action space

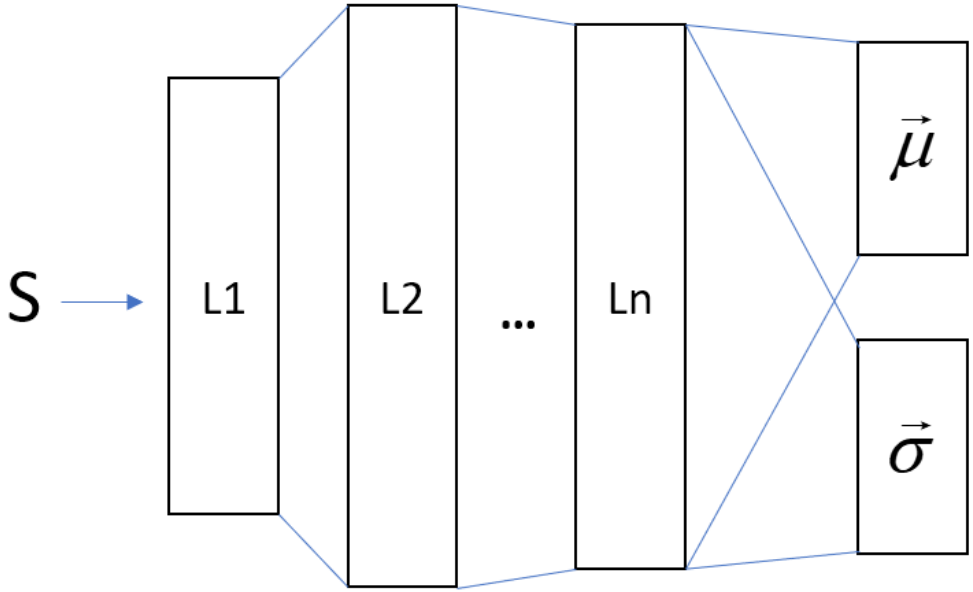


# Continuous action space

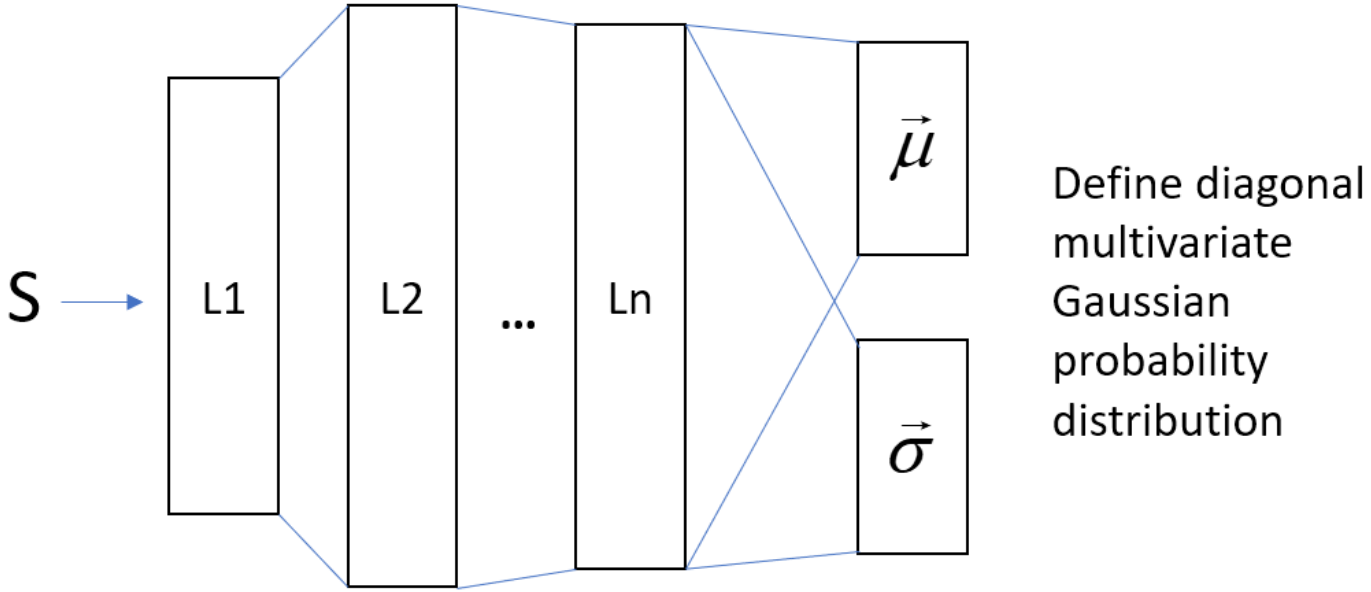


# Continuous action space




Define diagonal multivariate Gaussian probability distribution

# Continuous action space




Yes!!!! **Continuous actions**! Big improvement in applicability of RL!

# Vanilla Policy Gradient

**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:

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

- 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

# REINFORCE algorithm

- An deeper analysis shows we can also consider rewards-to-go for states instead of rewards of whole trajectory, adding temporal information to the algorithm and improving learning.

**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 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

# 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

**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

# Actor-Critic Architectures

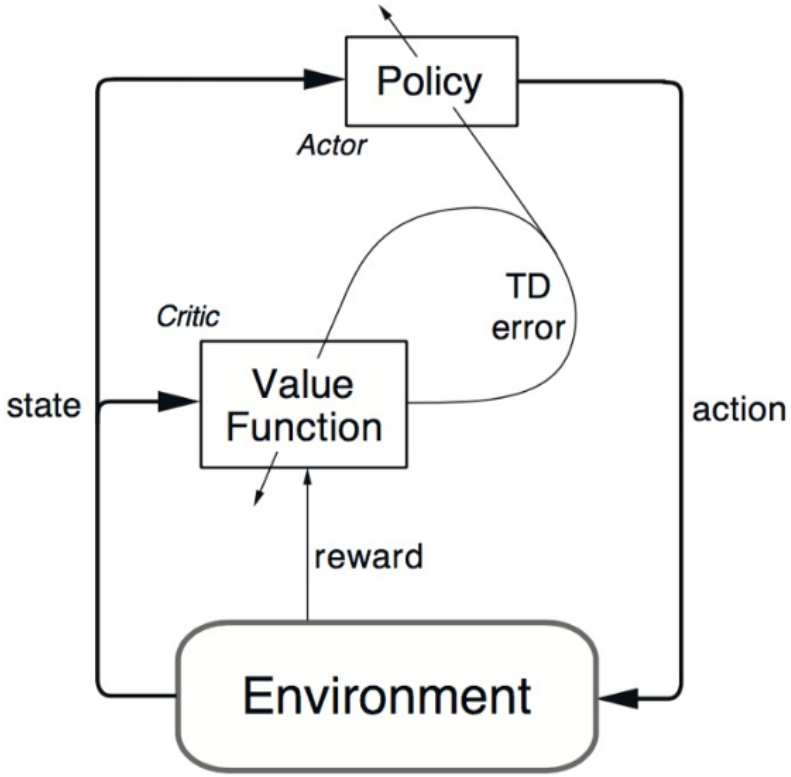
- Monte-Carlo policy gradient has high variance
- So we used a baseline to reduce the variance $R_t - V(s_t)$
- Can we do something to speed up learning like we did with MC using TD?

# Actor-Critic Architectures

- 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.

# Choices

- The policy gradient has many equivalent forms

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

$$\begin{aligned}
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) Q_w(s,a)\right] && \text{Actor-Critic} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) A_w(s,a)]\right] && \text{Advantage Actor-Critic} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) \delta\right] && \text{TD Actor-Critic} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) A_{GAE}\right] && \text{Generalized Actor Critic}
\end{aligned}$$

- Each leads a stochastic gradient ascent algorithm
- Critic uses **policy evaluation** (e.g. MC or TD learning) to estimate $Q^\pi(s,a), A^\pi(s,a)$ or $V^\pi(s)$

# 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:

$$\begin{aligned} V^{\pi_\theta}(s) &\approx V_v(s) && (1) \\ Q^{\pi_\theta}(s,a) &\approx Q_w(s,a) && (2) \\ A(s,a) &= Q_w(s,a) - V_v(s) && (3) \end{aligned}$$

- And updating both value functions by e.g. TD learning
- Nice thing, you only punish policy when not optimal (why?) Do you see resemblance with REINFORCE with baseline?

# Other versions of A2C

- One way to implement A2C method without two different networks to estimate $Q_w(s,a)$ and $V_v(s)$ is to use estimators of $Q_w(s,a)$.
- For instance, TD Advantage estimator:

$$\begin{aligned} A^{\pi_\theta}(s,a) &= Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s) \\ &= \mathbb{E}_{\pi_\theta}\left[r + \gamma V^{\pi_\theta}(s')|s,a\right] - V^{\pi_\theta}(s) \end{aligned}$$

- or MonteCarlo Advantage estimator:

$$\begin{aligned} A^{\pi_\theta}(s,a) &= Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s) \\ &= \mathbb{E}_{\pi_\theta}\left[R|s,a\right] - V^{\pi_\theta}(s) \end{aligned}$$

- In practice these approaches only require one set of critic parameters $v$ to approximate TD error

# 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($\lambda$):

$$A^{\pi}_{GAE} = \sum_{t'=t}^{\infty} (\lambda\gamma)^{t'-t} \underbrace{[r_{t'+1} + \gamma V^{\pi}_{\theta}(s_{t'+1}) - V^{\pi}_{\theta}(s_{t'})]}_{\text{t'-step advantage}}$$

- Used in continuous setting for locomotion tasks

# Asyncrhonous 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

# Asyncrhonous Advantage Actor Critic (A3C)

- What about exploration in Policy Gradient methods?

# Asyncrhonous Advantage Actor Critic (A3C)

- What about exploration in Policy Gradient methods?
- Policy is stochastic, so naturally it explores
- But degree of exploration usually converges too fast
- Usually, in the loss function, a term is added that encourages exploration
- This is done computing the *Entropy* of the policy:

$$H\left(\pi\left(\cdot \mid s_t\right)\right) = -\sum_{a \in A} \pi\left(a \mid s_t\right) \log \pi\left(a \mid s_t\right)$$

Subsection 4

## Conclusions and other approaches

# Summary

- The policy gradient has many equivalent forms

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) R_t\right] && \text{REINFORCE} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) Q_w(s,a)\right] && \text{Actor-Critic} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) A_w(s,a)]\right] && \text{Advantage Actor-Critic} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) \delta\right] && \text{TD Actor-Critic} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) A_{GAE}\right] && \text{Generalized Actor Critic}
\end{aligned}
$$

- Each leads a stochastic gradient ascent algorithm
- Critic uses **policy evaluation** (e.g. MC or TD learning) to estimate $Q^\pi(s,a), A^\pi(s,a)$ or $V^\pi(s)$

# Compatible Function Approximation: Bias in AC

- 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

$$\nabla_w \mathbb{E}_{\pi_\theta}\left[(Q^{\pi_\theta}(s,a) - Q_w(s,a))^2\right] = 0$$

- 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 is First order / linear approximation of the value function's dependence on the **policy parameterization** instead of **actual policy**[1]

[1] A 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 o due to particular policy parameters and the stochasticity of the policy
  - May not be able to recover from a bad choice, collapse in performance!

# 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 o due to particular policy parameters and the stochasticity of the policy
  - May not be able to recover from a bad choice, collapse in performance!
  - *Small learning rates do not solve the problem because small changes in weights can change a lot the policy* (distances in weight spaces not necessarily mean small distances in policies)

# Better Policy Gradient Directions: Natural Gradient

- 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

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

- Where $G_\theta$ is the Fisher information matrix

$$G_\theta = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s)^T\right]$$

# Better Policy Gradient Directions: Natural Gradient

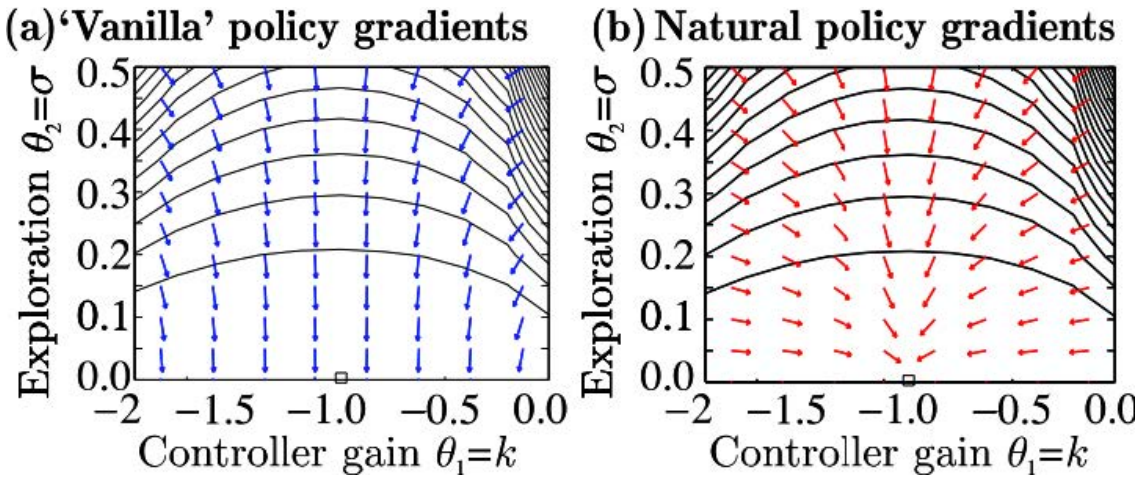



**Figure:** From (Peters et al. 2005)

# 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,

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) A^{\pi_\theta}(s,a)\right] \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s)^T w\right] \\
&= G_\theta w \\
\nabla_\theta^{nat} J(\theta) &= w
\end{aligned}$$

- i.e. update actor parameters in direction of critic parameters

# Problems with Policy gradient methods

- Step size and Policy gradient directions
- Data inefficiency:
  - We don't have data replay because action should be the one selected by the current policy
  - And policy changes after learning
  - (notice the difference with off-policy learning)
  - We cannot reuse data which lead to policy inefficiency
- We don't have anymore the experience Replay. Can we reuse data?
- Yes! Let's go back to Importance Sampling

# [Importance Sampling (IS) technique]

- Estimate the expectation of a different distribution w.r.t. the distribution used to draw samples

$$
\begin{aligned}
\mathbb{E}_{x\sim p}\left[f(x)\right] &= \sum p(x)f(x) \\
&= \sum q(x)\frac{p(x)}{q(x)}f(x) \\
&= \mathbb{E}_{x\sim q}\left[\frac{p(x)}{q(x)}f(x)\right] \\
&\approx \frac{1}{T}\sum_{t=1}^{T}\frac{p(x^t)}{q(x^t)}f(x^t)
\end{aligned}
$$

*where data is sampled using $q$ distribution*. That means, we can estimate $\mathbb{E}_{x\sim p}\left[f(x)\right]$ using distribution $q$ instead of $p$

# [Importance Sampling (IS) technique]

- Caution:
  - Cannot use if $q$ is zero where $p$ is non–zero
  - Importance sampling can dramatically increase variance (choose $q$ wisely, as close to $p$ as possible)

# Problems with Policy gradient methods

- Let's use old policy to collect data

$$\begin{aligned}\nabla J(\theta) &= \mathbb{E}_{(s_t,a_t)\sim\pi_\theta}\left[\nabla \log \pi_\theta\left(a_t \mid s_t\right) A\left(s_t, a_t\right)\right] \\ &= \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta_{old}}}\left[\frac{\pi_\theta\left(s_t, a_t\right)}{\pi_{\theta_{old}}\left(s_t, a_t\right)} \nabla \log \pi_\theta\left(a_t \mid s_t\right) A\left(s_t, a_t\right)\right]\end{aligned}$$

# Problems with Policy gradient methods

- Let's use old policy to collect data

$$\begin{aligned}\nabla J(\theta) &= \mathbb{E}_{(s_t,a_t)\sim\pi_\theta}\left[\nabla \log \pi_\theta\left(a_t \mid s_t\right) A\left(s_t, a_t\right)\right] \\ &= \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta_{old}}}\left[\frac{\pi_\theta\left(s_t, a_t\right)}{\pi_{\theta_{old}}\left(s_t, a_t\right)} \nabla \log \pi_\theta\left(a_t \mid s_t\right) A\left(s_t, a_t\right)\right]\end{aligned}$$

- Surrogate function to optimize:

$$J(\theta) = \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta_{old}}}\left[\frac{\pi_\theta\left(s_t, a_t\right)}{\pi_{\theta_{old}}\left(s_t, a_t\right)} A\left(s_t, a_t\right)\right]$$

# Problems with Policy gradient methods

- Cool. We can use now old data!

# Problems with Policy gradient methods

- Cool. We can use now old data!
- However, we have a problem with Importance Sampling.
- The expectations are them same, but we are using sampling method to estimate them and **variance** is different.
- That means that we may need to sample more data, if ratio is far away from 1 (old policy is far from current policy)

# TRPO (Schulman et al 2017)

- *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.

$$\begin{aligned} \underset{\theta}{\text{maximize}} \quad & \hat{\mathbb{E}}_t \left[ \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}(s_t, a_t) \right] \\ \text{subject to} \quad & \hat{\mathbb{E}}_t \left[ \mathrm{KL} \left[ \pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t) \right] \right] \leq \delta \end{aligned}$$

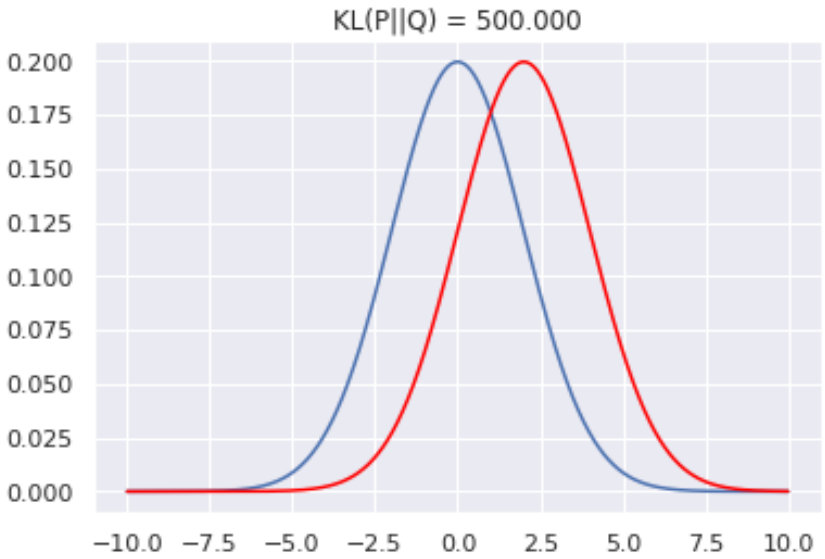
- Under penalizing constraint (using KL divergence of $\theta$ and $\theta_{old}$) that ensures improvement of the policy in the proximity (small step size)

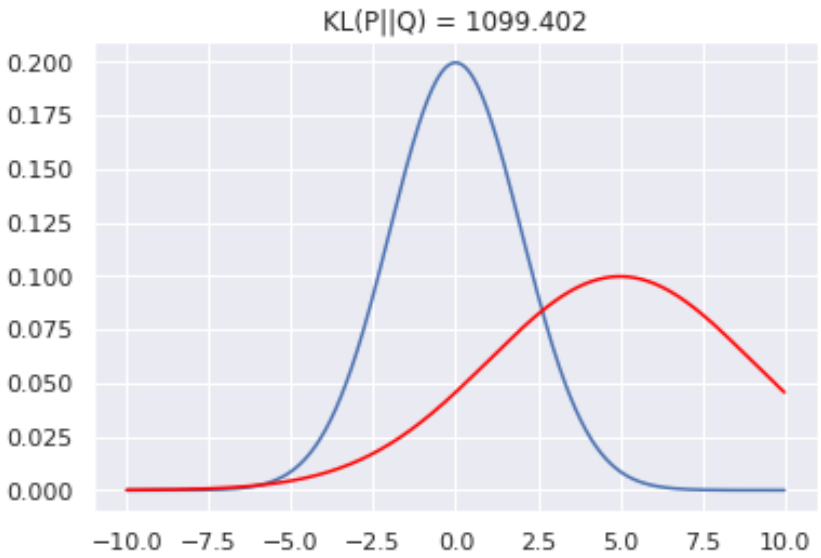
# [ Kullback–Leibler divergence (KL Divergence) ]

- Used to compute differences between distributions

$$D_{KL}(P\|Q) = \int p(x) \log\left(\frac{p(x)}{q(x)}\right) dx$$

- Examples:



# TRPO (Schulman et al 2017)

- In policies $D_{KL}$:

$$D_{KL}\left(\pi_1 || \pi_2\right)[s] = \sum_{a \in A} \pi_1(a \mid s) \log \frac{\pi_1(a \mid s)}{\pi_2(a \mid s)}$$

- So:

$$\begin{array}{ll} \underset{\theta}{\text{maximize}} & \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}(s_t, a_t)\right] \\ \text{subject to} & \hat{\mathbb{E}}_t\left[\mathrm{KL}\left[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)\right]\right] \leq \delta \end{array}$$

- Equivalent to improve the maximum with minimum change in parameters under the KL divergence measure.
- It is solved using Natural Gradient (see here for a nice explanation).

# TRPO (Schulman et al 2017)

- In policies $D_{KL}$:

$$D_{KL}\left(\pi_1 || \pi_2\right)[s] = \sum_{a \in A} \pi_1(a \mid s) \log \frac{\pi_1(a \mid s)}{\pi_2(a \mid s)}$$

- So:

$$\begin{array}{ll} \underset{\theta}{\text{maximize}} & \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}(s_t, a_t)\right] \\ \text{subject to} & \hat{\mathbb{E}}_t\left[\mathrm{KL}\left[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)\right]\right] \leq \delta \end{array}$$

- Equivalent to improve the maximum with minimum change in parameters under the KL divergence measure.
- It is solved using Natural Gradient (see here for a nice explanation).
- A lot of other details. See paper for details

# Proximal Policy Optimization (Schulman et al 2017)

- *Proximal Policy Optimization* (PPO) inspired in TRPO but simplifies computation.
- New goal *surrogate* function is objective function clipped to limit changes around the current solution:

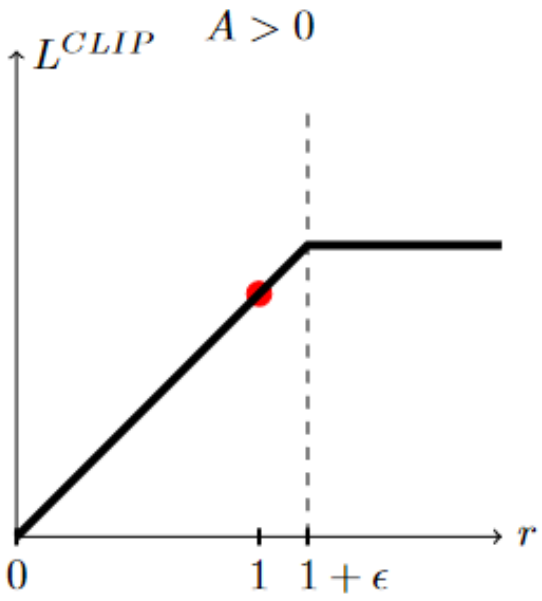
$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[ \min \left( r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right]$$

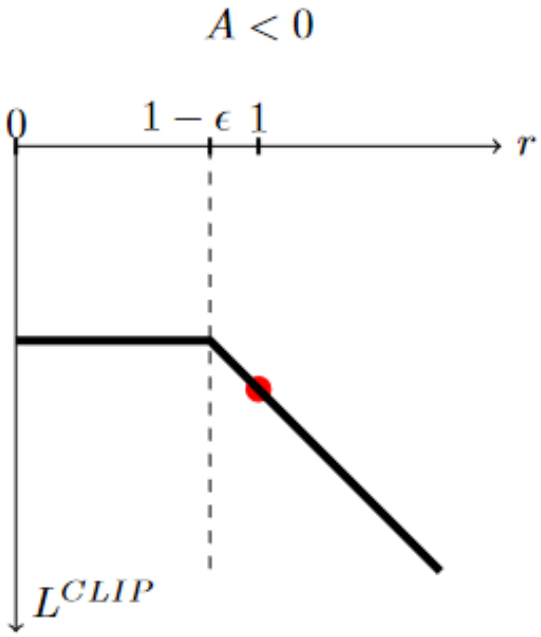
where

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

# Proximal Policy Optimization (Schulman et al 2017)

- How clipping works:



# Proximal Policy Optimization (Schulman et al 2017)

- Simple algorithm:

**Algorithm 1** PPO, Actor-Critic Style

```
for iteration=1, 2, ... do
    for actor=1, 2, ..., N do
        Run policy π_θ_old in environment for T timesteps
        Compute advantage estimates Â_1, ..., Â_T
    end for
    Optimize surrogate L wrt θ, with K epochs and minibatch size M ≤ NT
    θ_old ← θ
end for
```

- N actors (in parallel) run in order to get data from old policy (from few hundred to a few thousand samples). [ Notice iid and amount of data collected ]
- Optimization is done for K (3-10) batches reusing data (notice that at each iteration r changes!)

# PPO conclusions

- The clipped objective function prevent the policy from diverging or becoming unstable. This allows PPO to learn from smaller amounts of data without overfitting or becoming overly sensitive to noisy samples.
- Still no use of Experience Replay, so not so sample efficient like value-based methods.
- A lot of implementation details to be aware (Engstrom et al 2020) and The 32 Implementation Details of PPO
- In recent versions some terms added in the Loss function (entropy and Bellman Error)
- Some videos: Learning to walk in minutes from (Rudder et alt 22)

# PPO conclusions

- The clipped objective function prevent the policy from diverging or becoming unstable. This allows PPO to learn from smaller amounts of data without overfitting or becoming overly sensitive to noisy samples.
- Still no use of Experience Replay, so not so sample efficient like value-based methods.
- A lot of implementation details to be aware (Engstrom et al 2020) and The 32 Implementation Details of PPO
- In recent versions some terms added in the Loss function (entropy and Bellman Error)
- Some videos: Learning to walk in minutes from (Rudder et alt 22)
- ... Most popular on-policy method and famous nowadays because it has been used to train *ChatGPT*!

Subsection 5

# Off-policy AC methods

# DDPG: Deep Determ. PG (Lillicrap et al. 2016)

- 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: $a_t = \pi_\theta(s_t) + \epsilon$ (where $\epsilon \sim \mathcal{N}$)

# DDPG: Deep Determ. PG (Lillicrap et al. 2016)

- $Q_\phi$ network is trained using standard loss function:

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

- As action is *deterministic* and *continuous* (NN), we can easily follow the gradient in policy network to increase future reward:

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

# DDPG: Deep Determ. PG (Lillicrap et al. 2016)

**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:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  **end for**
**end for**

# TD3: Twin Delayed DDPG (Fujimoto et al, 2018)

- Similar to DDPG but with the following changes:
  1. *Pessimistic Double-Q Learning*: It uses two (twin) Q networks and uses the "pessimistic" one for current state for updating the networks

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

  2. *Clipped action regularization in loss*: noise added like DDPG but noise bounded to fixed range.

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

  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)

- DDPG and TD3 are deterministic methods that add noise for exploration. In SAC, policies are stochastic according to Soft-max:

$$\pi(a|s) = \frac{e^{\left(\frac{Q(s,a)}{\alpha}\right)}}{Z(s)}$$

- Solution to this criteria are Entropy-regularized policies: we will look for *maximum entropy* policies with given data,

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

where $\alpha$ is the trade-off between reward and entropy and the entropy of a policy is defined as:

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

# SAC: Soft Actor Critic (Haarnoja et al, 2018)

- However, we cannot apply the soft-max operator in the continuous space! We need an actor that tries to guess the maximum. So the goal is, given a Q-value function Q, find the policy that:

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}} \left[ \mathrm{D}_{\mathrm{KL}} \left( \pi_\phi \left( \cdot \mid \mathbf{s}_t \right) \middle\| \frac{e^{Q_\theta(\mathbf{s}_t, \cdot)/\alpha}}{Z_\theta \left( \mathbf{s}_t \right)} \right) \right]$$

- With some rearrangement applying the $D_{KL}$ definition, we have the loss for the Actor.

$$J_\pi(\phi) = \mathbb{E}_{s_t \sim D} \left[ \mathbb{E}_{a_t \sim \pi_\phi} \left[ \alpha \log \pi_\phi \left( a_t \mid s_t \right) - Q_\theta \left( s_t, a_t \right) \right] \right]$$

# SAC: Soft Actor Critic (Haarnoja et al, 2018)

- Let's define value functions in this case:

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

- So Bellman equations can be written as:

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

# SAC: Critic loss

Architecture: Networks and loss functions for actor and critic:

- Q-value functions: $Q_{\theta_1}(s,a), Q_{\theta_2}(s,a)$ (*twin* like TD3) with Q-target counterpart
  - Let's define the *target* (Bellman eq.) where $a'$ is sampled from the policy:

$$y(s,a,r,s') = r + \gamma \left( \min_{i=1,2} Q_{\theta'_i}(s',a') - \alpha \log \pi_\phi(a'|s') \right)$$

  - Then Loss for the Q-value networks is:

$$L(\theta_i, \mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s',d)\sim\mathcal{D}} \left[ \left( Q_{\theta_i}(s,a) - y(s,a,r,s') \right)^2 \right]$$

# SAC: Actor loss and Reparametrization trick

- Policy $\pi_\phi(a|s)$. Maximize:

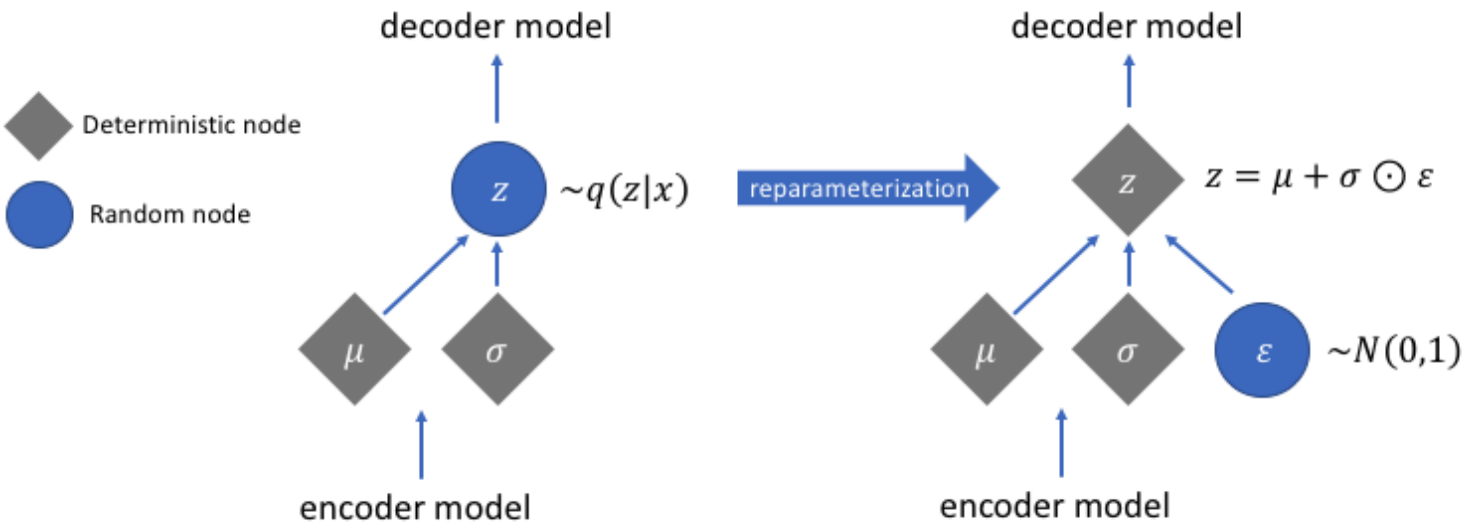
$$\mathop{\mathbb{E}}_{a\sim\pi_\phi}\left[Q^{\pi_\phi}(s,a) - \alpha \log \pi_\phi(a|s)\right]$$

- But problematic! because in gradient $\nabla_\phi$, expectation follow **stochastic** $\pi_\phi$.
- Authors use a reparametrizarion trick (see here or here). It can be done **when we define the stochastic** $\pi_\phi$ **as Gaussian** by adding noise to the action:

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

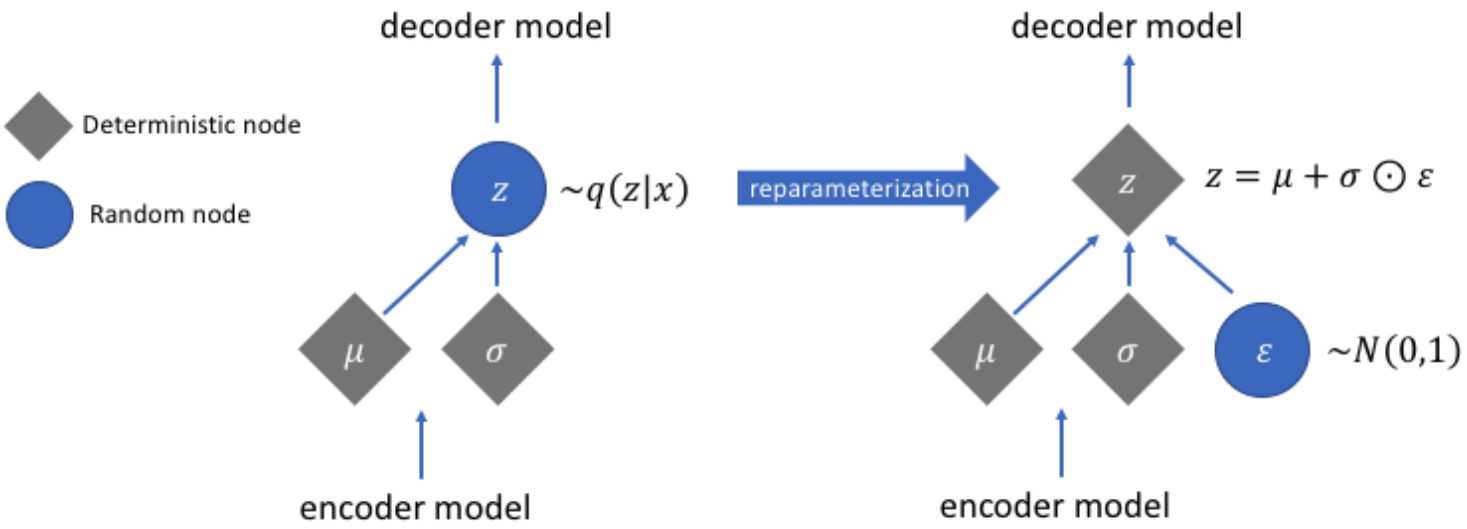
# SAC: Reparametrization Trick and Tahn

- Reparametrization trick solves the problem of applying the gradients:



# SAC: Reparametrization Trick and Tahn

- Reparametrization trick solves the problem of applying the gradients:




- Finally, `tanh is to set a limit to the actions while having exact values for` $\pi$ `(notice problem with Normal distribution and boundaries)`

# SAC: Some comments

- Now we can rewrite the term as:

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

- Now we can optimize the policy according to

$$\max_\phi \mathop{\mathbb{E}}_{s\sim\mathcal{D},\xi\sim\mathcal{N}}[Q_{\theta_1}(s,\tilde{a}_\phi(s,\xi)) - \alpha \log \pi_\phi(\tilde{a}_\phi(s,\xi)|s)]$$

and we can compute now the gradients:

# SAC: Algorithm

**Algorithm 1** Soft Actor-Critic

Input: initial policy parameters $\theta$, Q-function parameters $\phi_1$, $\phi_2$, 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 $\theta$ 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

# SAC: Some final comments

- Entropy enforces exploration (see why?), so no need to add noise to actions.
- Usually $\alpha$ is fixed as a hyper-parameter or decreases during learning and is disabled to test performance. Also some heuristic methods to automatically adjust it (Haarnoja et al, 2018)
- State of the art during a lot of time
- Very popular in robotics
- Very robust in stochastic domains
- An extra hyperparameter $\alpha$

# SAC: Some final comments

- Entropy enforces exploration (see why?), so no need to add noise to actions.
- Usually $\alpha$ is fixed as a hyper-parameter or decreases during learning and is disabled to test performance. Also some heuristic methods to automatically adjust it (Haarnoja et al, 2018)
- State of the art during a lot of time
- Very popular in robotics
- Very robust in stochastic domains
- An extra hyperparameter $\alpha$
- Some videos of the paper: Mujoco and Robot

# SAC: Some final comments

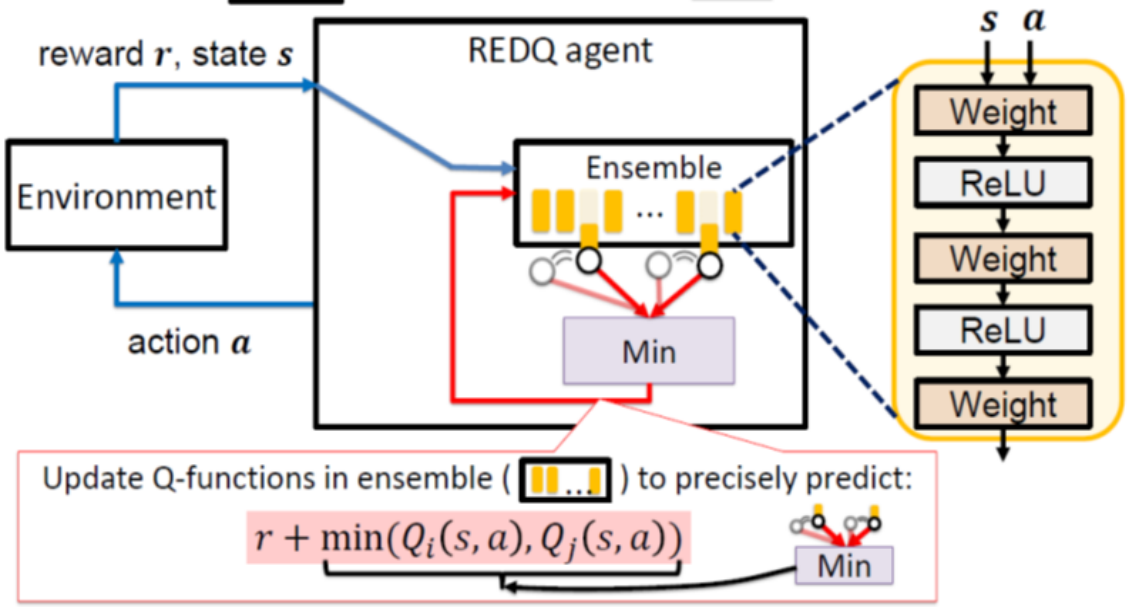
- Entropy enforces exploration (see why?), so no need to add noise to actions.
- Usually $\alpha$ is fixed as a hyper-parameter or decreases during learning and is disabled to test performance. Also some heuristic methods to automatically adjust it (Haarnoja et al, 2018)
- State of the art during a lot of time
- Very popular in robotics
- Very robust in stochastic domains
- An extra hyperparameter $\alpha$
- Some videos of the paper: Mujoco and Robot
  ( Today we can do it better (Wu et al 2022) using advanced techniques that we will see in the course )

# Latest methods in the family

- TQC in (Kuznetsov, 2020) extends SAC to the Distributional approach to approximate returns and, recently, (Farebrother et al., 2024) to Distributional losses
- REDQ in (Chen 2021) extends SAC to an ensemble of Q-value networks and doing several updates of the networks for each sample from the environment.
- DroQ in (Hiraoka 2022) modifies REDQ to have dropout Q-functions

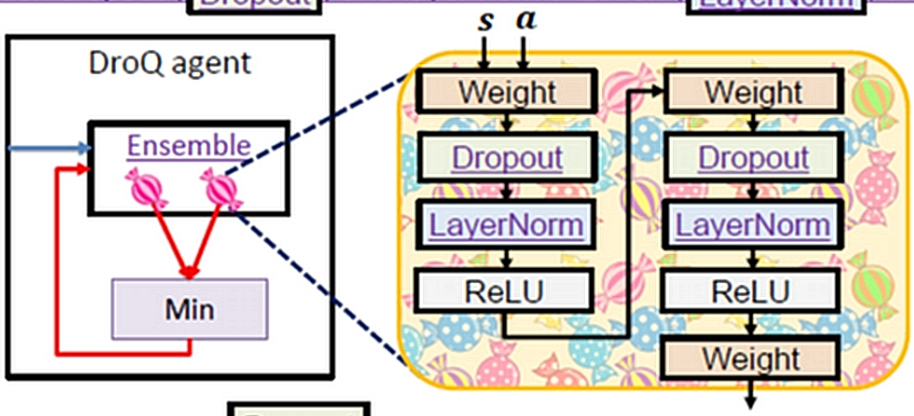
# REDQ (Chen 2021)

- REDQ (Chen, 2021) is a sample-efficient RL method equipped with **high update-to-data (UTD) ratio** and **randomized ensemble**.
- **High UTD ratio:** number of Q updates (→) per environment interaction (→) is high (e.g., 20 updates per interaction).
- **Randomized ensemble:** a randomly selected subset ( ) of ensemble ( ) is used at the target ( Min ) in the Q update (→).

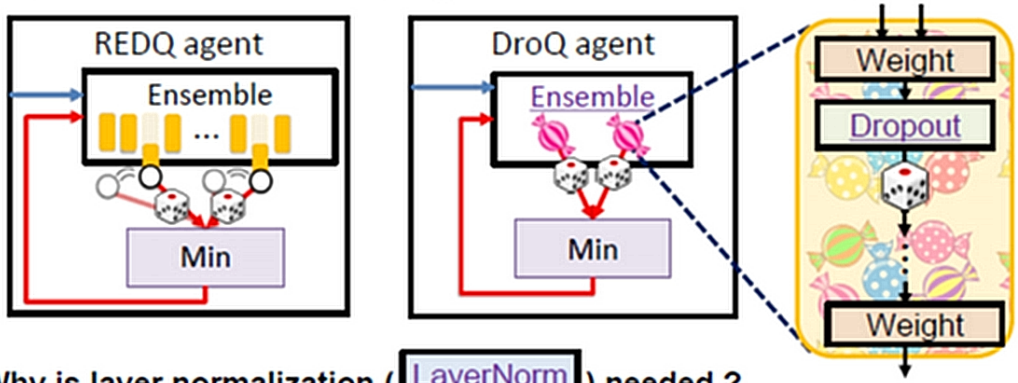


# DroQ (Hiraoka 2022)

- DroQ is a REDQ variant using a small ensemble of dropout Q-functions ( ) in which dropout ( Dropout ) and layer normalization ( LayerNorm ) are used.




- **Q. Why is dropout ( Dropout ) needed ?**
  A. To injecting Q-function uncertainty ( ) to the target ( Min ), similarly to REDQ.




- **Q. Why is layer normalization ( LayerNorm ) needed ?**
  A. To suppress ( ) the learning instability caused by dropout.

# Latest methods in the family

- TQC in (Kuznetsov, 2020) extends SAC to the Distributional approach to approximate returns and, recently, Farebrother et al. - 2024 to Distributional losses
- REDQ in (Chen 2021) extends SAC to an ensemble of Q-value networks and doing several updates of the networks for each sample from the environment.
- DroQ in (Hiraoka 2022) modifies REDQ to have dropout Q-functions
- Not so popular neither widely used as SAC.

# Latest methods in the family

- TQC in (Kuznetsov, 2020) extends SAC to the Distributional approach to approximate returns and, recently, Farebrother et al. - 2024 to Distributional losses
- REDQ in (Chen 2021) extends SAC to an ensemble of Q-value networks and doing several updates of the networks for each sample from the environment.
- DroQ in (Hiraoka 2022) modifies REDQ to have dropout Q-functions
- Not so popular neither widely used as SAC.
- We have found state of the art generic model-free algorithms for RL

# 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