Five - Policy Gradients

Motivations for using policies, how to use a policy, the score function, Actor-Critic, DPG, PPO.

Contrast with value function methods

Previously we generated a policy from a value function.

$$a = \underset{a}{argmax} Q(s, a)$$

In policy gradients we **parameterize a policy directly**. This policy is a probability distribution over actions.

$$a \sim \pi(a_t | s_t; \theta)$$

Motivations for policy gradients

Stochastic policies

Figure 1: A determinstic policy (i.e. always rock) is easily exploited

A stochastic policy means exploration is built into the policy -> exploration can be controlled by the agent by changing parameters. A common example of this is for the agent to be able to learn the standard deviation of an action.

High dimensional action spaces

Q-Learning requires a discrete action space to *argmax* across

Lets imagine controlling a robot arm in three dimensions in the range [0, 90] degrees

This corresponds to approx. 750,000 actions a Q-Learner would need to argmax across

We also lose shape of the action space by discretization. By this I mean that the agent now has an action space of discrete actions with no understanding of how they relate to each other

Discretizing continuous action spaces

In [8]:
```
#  a robot arm operating in three dimensions with a 90 degree range
single_dimension = np.arange(91)
single_dimension
```

Out[8]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
       51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
       68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
       85, 86, 87, 88, 89, 90])
```

In [32]:
```
#  we can use the combinations tool from the Python standard library
from itertools import product
all_dims = [single_dimension.tolist() for _ in range(3)]
all_actions = list(product(*all_dims))
print('num actions are {}'.format(len(all_actions)))
print('expected_num_actions are {}'.format(len(single_dimension)**3))

#  we can look at the first few combinations of actions
all_actions[0:10]
```

```
num actions are 753571
expected_num_actions are 753571
```

Out[32]:
```
[(0, 0, 0),
 (0, 0, 1),
 (0, 0, 2),
 (0, 0, 3),
 (0, 0, 4),
 (0, 0, 5),
 (0, 0, 6),
 (0, 0, 7),
 (0, 0, 8),
 (0, 0, 9)]
```

In [33]:
```
#  and the last few
all_actions[-10:]
```

Out[33]:
```
[(90, 90, 81),
 (90, 90, 82),
 (90, 90, 83),
 (90, 90, 84),
 (90, 90, 85),
 (90, 90, 86),
 (90, 90, 87),
 (90, 90, 88),
 (90, 90, 89),
 (90, 90, 90)]
```

Figure 2: An example of the exponential blowup in action space complexity - aka the Curse of Dimensionality

Optimize return directly

When learning value functions our optimizer is working towards improving the predictive accuracy of the value function - our gradients point in the direction of predicting return

This isn't what we really care about - we care about maximizing return

Policy methods optimize return directly - changing weights according to the gradient that maximizes future reward - aligning gradients with our objective (and hopefully a business objective)

Simplicity

Learning a value function and deriving a policy from it is more complex than simply parameteterizing a policy. Some states don't require exact quantification of return for each action - it's very obvious what is the correct action. An example of this is TODO

Policy gradients are more general and versatile

More compatible with recurrent neural networks. Policy gradient methods are often trained using sequences of experience.

Policy gradients versus value functions

Policy gradients

- optimize return directly
- work in continuous and discrete action spaces
- works better in high-dimensional action spaces
- usually on-policy - meaning it is hard to escape the bias of a bad initial policy
-

Value functions

- optimize value function accuracy
- off policy learning
- exploration
- better sample efficiency

Figure 3: Sometimes it's just eaiser to go the shorter route (than to estimate times for all possible actions)

Parameterizing policies

The type of policy you parameterize depends on the **action space**

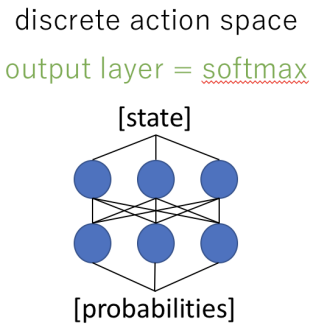


Figure 4: Parameterizing a discrete policy

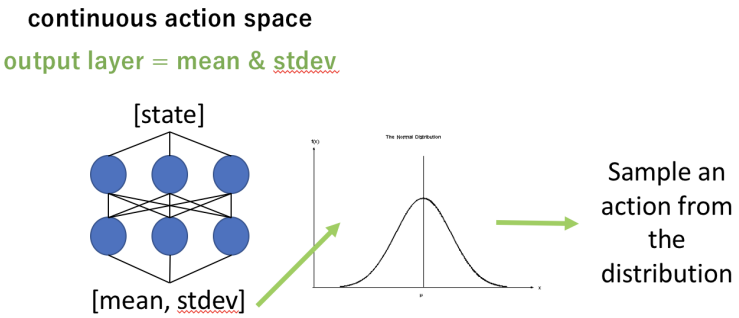


Figure 5: Parameterizing a continous policy

Policy gradients without equations

We have a parameterized policy

- a neural network that outputs a distribution over actions

How do we improve it - how do we learn?

- change parameters to take actions that get more reward
- change parameters to favour probable actions

Reward function is not known

- but we can calculate the *gradient the expected reward*

Policy gradients with a few equations

Our policy $\pi(a_t|s_t;\theta)$ is a **probability distribution over actions**

How do we improve it?

- change parameters to take actions that get more reward
- change parameters to favour probable actions

Reward function is not known - but we can calculate the *gradient of the expectation of reward*

$$\nabla_\theta \mathbb{E}[G_t] = \mathbb{E}[\nabla_\theta \log \pi(a|s) \cdot G_t]$$

We can figure out how to change our parameters without actually knowing the reward function itself

The score function in statistics

The **score function** comes from using the log-likelihood ratio trick

The score function allows us to get the gradient of a function by **taking an expectation**

Expectataions are averages - use sample based methods to approximate them

$$\nabla_\theta \mathbb{E}[f(x)] = \mathbb{E}[\nabla_\theta \log P(x) \cdot f(x)]$$

$$\begin{aligned}
\nabla_\theta E_x[f(x)] &= \nabla_\theta \sum_x p(x) f(x) && \text{definition of expectation} \\
&= \sum_x \nabla_\theta p(x) f(x) && \text{swap sum and gradient} \\
&= \sum_x p(x) \frac{\nabla_\theta p(x)}{p(x)} f(x) && \text{both multiply and divide by } p(x) \\
&= \sum_x p(x) \nabla_\theta \log p(x) f(x) && \text{use the fact that } \nabla_\theta \log(z) = \frac{1}{z} \nabla_\theta z \\
&= E_x[f(x) \nabla_\theta \log p(x)] && \text{definition of expectation}
\end{aligned}$$

Figure 6: Derivation of the score function

The score function in reinforcement learning

$$\nabla_\theta \mathbb{E}[G_t] = \mathbb{E}[\nabla_\theta \log \pi(a|s) \cdot G\_t]$$

```
gradient of return = expectation of the gradient of the policy * return
```

The RHS is an expectation - we can estimate it by sampling

The expectation is made up of things we can sample from

- we can sample from our policy
- we can sample the return (from experience)

Training a policy

We use the score function to get the gradient, then follow the gradient

```
gradient = log(probability of action) * return
```

```
gradient = log(policy) * return
```

The score function limits us to on-policy learning - we need to calculate the log probability of the action taken by the policy

Policy gradient intuition

$$\nabla\_\theta \mathbb{E}[G\_t] = \mathbb{E}[\nabla\_\theta \log \pi(a|s) \cdot G\_t]$$

$\log \pi(a_t|s_t;\theta)$ - how probable was the action we picked - we want to reinforce actions we thought were good

$ G_t $ - how good was that action - we want to reinforce actions that were actually good

Different methods to approximate the return G_t

We can use a Monte Carlo estimate - this is known as REINFORCE

Using a Monte Carlo approach comes with all the problems we saw earlier - high variance - no online learning - requires episodic environment

How can we get some the advantages of Temporal Difference methods?

Baseline

We can introduce a baseline function

- this reduces **variance without introducing bias**
- a natural baseline is the value function (weights w)
- baseline should not be dependent on our policy parameters θ

$$\log \pi(a_t|s_t;\theta) \cdot (G_t - B(s_t;w))$$

This also gives rise to the concept of **advantage** - how much better this action is than the average action (policy & env dependent)

$$A_\pi(s_t, a_t) = Q_\pi(s_t, a_t) - V_\pi(s_t)$$

Actor-Critic

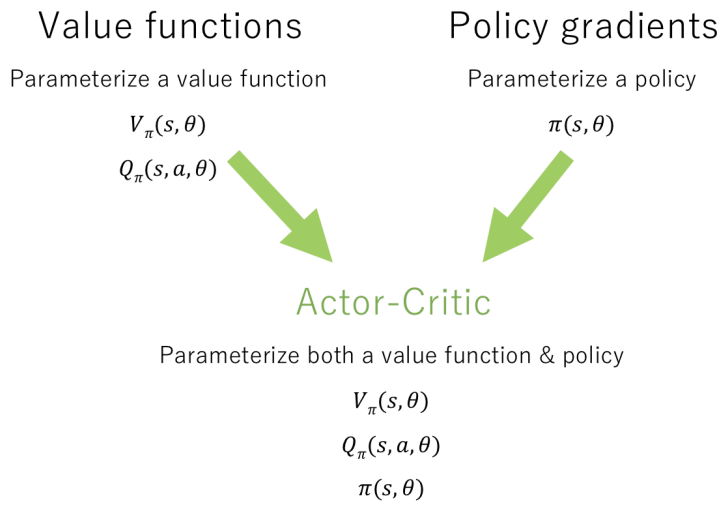


Figure 7: fig

Actor-Critic brings together value functions and policy gradients

We parameterize two functions

- **actor** = policy
- **critic** = value function

We update our actor (i.e. the behaviour policy) in the direction suggested by the critic

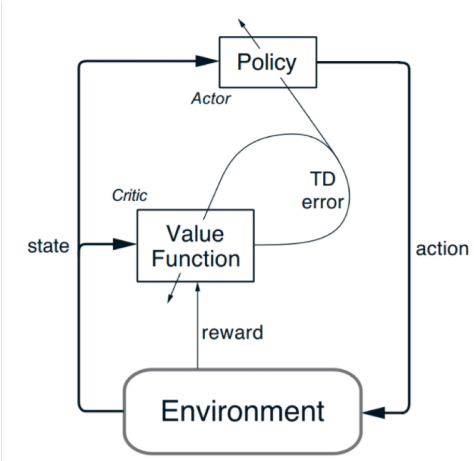


Figure 11.1: The actor–critic architecture.

Figure 8: Actor-Critic acrchitecture - Sutton & Barto

Input: policy $\pi(a|s, \boldsymbol{\theta})$, $\hat{v}(s, \boldsymbol{w})$
Parameters: step sizes, $\alpha > 0$, $\beta > 0$
Output: policy $\pi(a|s, \boldsymbol{\theta})$
initialize policy parameter $\boldsymbol{\theta}$ and state-value weights $\boldsymbol{w}$
for *true* **do**
  initialize s, the first state of the episode
  $I \leftarrow 1$
  for s *is not terminal* **do**
    $a \sim \pi(\cdot|s, \boldsymbol{\theta})$
    take action a, observe s', r
    $\delta \leftarrow r + \gamma \hat{v}(s', \boldsymbol{w}) - \hat{v}(s, \boldsymbol{w})$ (if s' is terminal, $\hat{v}(s', \boldsymbol{w}) \doteq 0$)
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \beta \delta \nabla_{\boldsymbol{w}} \hat{v}(s_t, \boldsymbol{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha I \delta \nabla_{\boldsymbol{\theta}} log \pi(a_t|s_t, \boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $s \leftarrow s'$
  end
end

within episode updating
TD error
update value function (critic)
update policy (actor)

Algorithm 6: Actor-Critic (episodic), adapted from Sutton and Barto (2017)

Figure 9: Actor-Critic algorithm - Sutton & Barto

Deterministic Policy Gradient

Actor Critic

Determinstic policy - more efficient than stochastic

Continuous action spaces

Off-policy learning

Uses experience replay

Uses target networks

Stochastic vs determinstic policies

Stochastic policy is a probability distribution over actions

Actions are selected by sampling from this distribution

$$\pi_\theta(a|s) = P[a|s;\theta]$$

$$a \sim \pi_\theta(a|s)$$

DPG parameterizes a deterministic policy

$$a = \mu_\theta(s)$$

DPG components

Actor - off policy - function that maps state to action - exploratory

Critic - on-policy - critic of the current policy - estimates $Q(s, a)$

Gradients

Stochastic integrates over both the state & action spaces

Deterministic integrates over only the state space -> leading to better sample efficiency

Updating policy weights

DPG results - the difference between stochastic (green) and deterministic (red) increases with the dimensionality of the action space

Stochastic

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim p^\pi, a\sim\pi_\theta}\ \nabla_\theta \log \pi_\theta(a|s) \cdot Q^\pi(s,a)$$

Deterministic

On policy

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim p^\pi}\ \nabla_\theta \mu_\theta(s)\, \nabla_a Q^\mu(s,a)\Big|_{a=\mu_\theta(s)}$$

Off policy

$$\nabla_\theta J_\beta(\pi_\theta) = \mathbb{E}_{s\sim p^\beta}\ \nabla_\theta \mu_\theta(s)\, \nabla_a Q^\mu(s,a)\Big|_{a=\mu_\theta(s)}$$

Figure 10: fig

The gradient

$$\nabla_\theta J_\beta(\pi_\theta) = \mathbb{E}_{s\sim p^\beta}\ \nabla_\theta \mu_\theta(s)\, \nabla_a Q^\mu(s,a)\Big|_{a=\mu_\theta(s)}$$

The update function

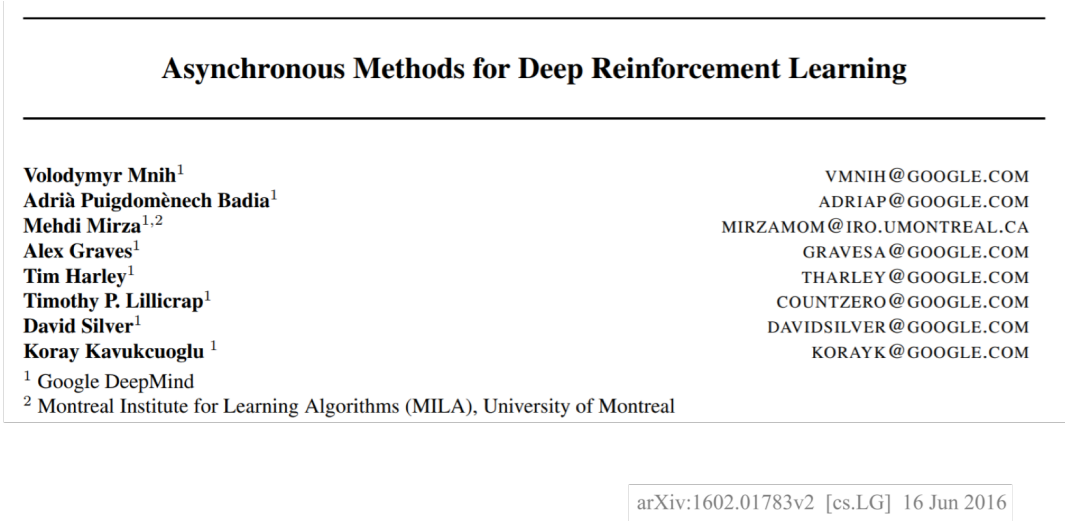
$$\theta_{t+1} = \theta_t + \alpha\, \nabla_\theta \mu_\theta(s_t)\, \nabla_a Q^w(s_t,a_t)\Big|_{a=\mu_\theta(s)}$$

α learning rate

Q^w action value function parameterized by weights w

Figure 11: fig

A3C



Asynchronous Methods for Deep Reinforcement Learning

Volodymyr Mnih[1] VMNIH@GOOGLE.COM
Adrià Puigdomènech Badia[1] ADRIAP@GOOGLE.COM
Mehdi Mirza[1,2] MIRZAMOM@IRO.UMONTREAL.CA
Alex Graves[1] GRAVESA@GOOGLE.COM
Tim Harley[1] THARLEY@GOOGLE.COM
Timothy P. Lillicrap[1] COUNTZERO@GOOGLE.COM
David Silver[1] DAVIDSILVER@GOOGLE.COM
Koray Kavukcuoglu [1] KORAYK@GOOGLE.COM
[1] Google DeepMind
[2] Montreal Institute for Learning Algorithms (MILA), University of Montreal

arXiv:1602.01783v2 [cs.LG] 16 Jun 2016

Figure 12: fig

Asynchronous Advantage Actor-Critic

We saw earlier that experience replay is used to make learning more stable & decorrelate updates - but can only be used with off-policy learners

Asynchronous

- multiple agents learning separately
- experience of each agent is independent of other agents
- learning in parallel stabilizes training
- allows use of on-policy learners
- runs on single multi-core CPU
- learns faster than many GPU methods

Advantage

- the advantage function

$$A_\pi(s_t, a_t) = Q_\pi(s_t, a_t) - V_\pi(s_t)$$

How much better an action is than the average action followed by the policy

A3C algorithm

https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-8-asynchronous-actor-critic-agents-a3c-c88f72a5e9f2

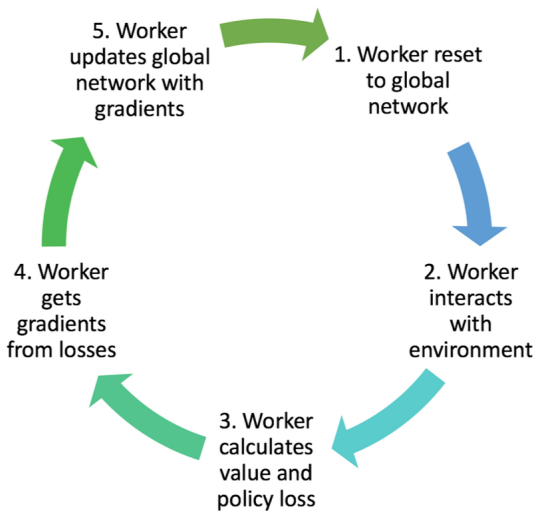


Figure 13: fig

Natural Policy Gradients, TRPO and PPO

These three pieces of work can be understood in order. The maths can get pretty heavy - a high level summary is given below

- natural policy gradients - rely on a compuationally intense second order derivative method to find a better gradient than normal gradient descent
- TRPO -
- PPO -

Natural Policy Gradient

- Kakde (2002) A Natural Policy Gradient - paper
- Natural Policy Gradient Explained - Jonathan Hui - Medium blog post

Standard gradient descent will follow the direction of steepest descent. This is a metric that is defined based on the choice of co-ordinates of the parameters - it is non-covariant.

The exact gradient of the average reward of a policy parameterized by θ:

$$\nabla\eta(\theta) = \sum \rho^{\pi}(s)\nabla\pi(a; s, \theta)Q^{\pi}(s, a)$$

Regular policy gradients can point towards plateaus. The natural gradient instead points towards the optimal solution - and finds the greedy action that would be chosen under one step of policy iteration. **The natural policy gradient helps with convergence**.

The natural gradient is an alternative gradient which is based on the structure of the parameter space. The natural gradient makes use of the Fisher infomation matrix, which is a matrix to measure the distance between distributions, regardless of the co-ordinates.

Calculating the gradient requires us to take the inverse of the Fisher infomation matrix $F(\theta)$, which is computationally expensive (second order derivative):

$$\tilde{\nabla} \equiv F(\theta)^{-1}\nabla\eta(\theta)$$

The update rule for the natural policy gradient is then:

$$f \leftarrow f + \nabla log\pi(a_t; s_t, \theta) \nabla log\pi(a_t; s_t, \theta)^T$$

Trust Region Policy Optimization (TRPO)

- Schulman et. al (2015) Trust Region Policy Optimization - paper

Trust region = finding an optimal solution within a certain distance (contrast with line search algorithms which follow graidents wherever they go).

In TRPO we optimize the objective function with a hard constraint

$$maximize \mathbb{E} \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_old}(a_t|s_t)} A_t]$$

$$\text{subject to} \beta \mathbb{E}[KL[\pi_{\theta_old}(\cdot|s_t), \pi_\theta(\cdot|s_t)]] \leq \delta$$

This problem is solved by making a linear approximation of the objective, and a quadratic approximation of the constraint.

TODO

Proximal Policy Optimization (PPO)

- Schulman et. al (2017) Proximal Policy Optimization Algorithms - paper
- Proximal Policy Optimization (PPO) Explained - blog post

Context - used in Open AI DOTA work (single 1024 node LSTM layer).

> Q-learning (with function approximation) fails on many simple problems and is poorly understood, vanilla policy gradient methods have poor data efficiency and robustness; and trust region policy optimization (TRPO) is relatively complicated, and is not compatible with architectures that include noise (such as dropout) or parameter sharing (between the policy and value function, or with auxiliary tasks).

PPO

- shares some of the benefits of TRPO, but is simpler (only first order derivatives) & more sample efficient
- multiple epochs of batch updates (rather than a single gradient update per sample)

Policy optimization is alternating between sampling data from the policy (i.e. acting) and performing several epochs of optimization.

Natural policy gradient (TODO) addresses the convergence problem of policy gradient methods. The natural policy gradient requires an unscalable calculation of a second-order derivative and its inverse. There are two solutions to this problem

1. approximate calculations of the second order derivative & its inverse (computationally expensive) - this is what TRPO & ACKTR do
2. use a first order solution with soft constraints to approximate the second order solution - this is what PPO does

PPO imposes a constraint as a penalty in the objective function. This soft constraint attempts to make the first order solution closer to the second order solution. Sometimes this constraint won't work, but the benefit of simplicity outweighs the occasional bad updates.

PPO limits how much we change our policy using the **Kullback–Leibler divergence (KLD)**. The KLD measures the distance between two distributions - we use it to penalize the distance between policy updates.

We can adjust the size of the trust region dynamically, based on the divergence of the policy (measured using the KLD). This is controlled using a hyperparamter β.

The objective function for PPO with an adaptive KLD penalty:

TODO

We can dynamically adjust β based on the value of the KLD. For low KLD we can expand the size of the trust region (by increasing β).

```
algorithm TODO
```