Model-Free Value Methods in Deep RL

Presenter: Jake Grigsby

University of Virginia
https://qdata.github.io/deep2Read/

202008

Markov Decision Process

Definition

A Markov Decision Process **(MDP)** consists of:

- $\mathcal{S}$, a set of states
- $\mathcal{A}$, a set of actions
- $\mathcal{R} \subseteq \mathbb{R}$, a set of rewards
- a dynamics function $p : \mathcal{S} x \mathcal{R} x \mathcal{S} x \mathcal{A} \rightarrow [0, 1]$

$$p(s', r|s, a) = Pr\{\mathcal{S}_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$$

It's common to break the dynamics function p up into a **Transition Function** $T(s, a, s') = \sum_{r \in \mathcal{R}} p(s', r|s, a)$, and a **Reward Function**

$$R(s, a) = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a)$$

The RL Problem

The goal of RL agents is to find a **policy**[1] $\pi^* : \mathcal{S} \rightarrow \mathcal{A}$ that maximizes the *expected discounted return*

$$\pi^* = \underset{\pi}{argmax} \underset{\tau \sim \pi}{\mathbb{E}} \left[\sum_{t=0}^{t=\infty} \gamma^t R_t \right]$$

where $\gamma \in [0, 1)$ is the *discount factor* that lets us deal with non-episodic tasks and τ is a *trajectory* (a sequence of states and actions that describe the agent's experience)

[1]Policies can also be stochastic, in which case they're written $\pi(a|s) : \mathcal{S}x\mathcal{A} \rightarrow [0, 1]$

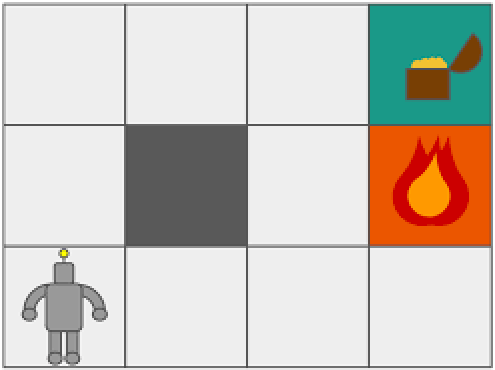
Simplifying Assumptions

We begin by making some assumptions about the task we are trying to solve:

1. The dynamics of the model ($p(s', r|s, a)$) are known
2. $|\mathcal{S}| \ll \infty$
3. $|\mathcal{A}| \ll \infty$

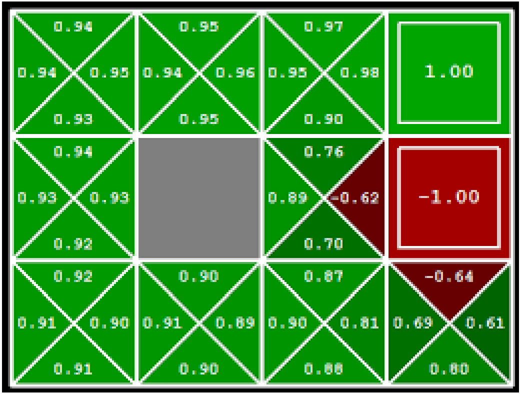
Simplifying Assumptions

Example: Gridworlds



Generalized Policy Iteration

Solution: Policy Iteration Dynamic Programming



We'll skip these details because knowledge of dynamics is such a limiting assumption in our case. More info can be found in [9]

Simplifying Assumptions

1. ~~The dynamics of the model ($p(s', r|s, a)$) are known~~
2. $|\mathcal{S}| \ll \infty$
3. $|\mathcal{A}| \ll \infty$

What if the environment dynamics are unknown?

Value Methods

Definition

Value methods attempt to learn the optimal *Q Function*

$$Q^*(s,a) = \underset{\pi}{max}\, \underset{\pi}{\mathbb{E}} \left[\sum_{k=0}^{k=\infty} \gamma^k R_{t+k+1} \mid \mathcal{S}_t = s, \mathcal{A}_t = a \right]$$

Why? Because given $Q^*(s,a)$, the optimal policy can easily[2] be computed by

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

[2] Well at least for now. The max operation is going to be a problem later...

Value Methods: Monte Carlo

- Play out entire episodes and keep track of the average return we experience from every (s, a) pair.
- Pros
  - Easy to implement
- Cons
  - Learning can only happen at the end of each episode. What if episodes are long (or never end)?

Value Methods: Monte Carlo

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:

$Q(s,a) \leftarrow$ arbitrary
$\pi(s) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list

Fixed point is optimal policy π^*

Proof is open question

Repeat forever:

(a) Generate an episode using exploring starts and π
(b) For each pair s, a appearing in the episode:
  $R \leftarrow$ return following the first occurrence of s, a
  Append R to $Returns(s,a)$
  $Q(s,a) \leftarrow \text{average}(Returns(s,a))$
(c) For each s in the episode:
  $\pi(s) \leftarrow \arg\max_a Q(s,a)$

Figure: Monte Carlo Action Value Control [9]

A Quick Note on Exploration vs. Exploitation

- At each time step, the agent must choose between "exploiting" the action it currently thinks has the best return and "exploring" alternatives to learn more about them.
- Most convergence guarantees assume state *coverage*
  - Every state will be visited an infinite number of times in an infinite number of timesteps.
  - This can be acheived by enforcing:

$$\pi(a|s) > 0, \forall s \in \mathcal{S}$$

A Quick Note on Exploration vs. Exploitation

- The simplest way to do this is to make an existing policy ϵ-greedy:

$$\pi'(s) = \begin{cases} \pi(s) & \text{with probability } (1-\epsilon); \\ \pi_{random}(s) & \text{with probability } \epsilon; \end{cases}$$

A Quick Note on Exploration vs. Exploitation

- The simplest way to do this is to make an existing policy ϵ-greedy:

$$\pi'(s) = \begin{cases} \pi(s) & \text{with probability } (1-\epsilon); \\ \pi_{random}(s) & \text{with probability } \epsilon; \end{cases}$$

- This can be thought of as injecting noise into the action space
- All of the agent's we'll be talking about use this general approach, but there is a lot of interesting work on motivating agents to explore efficiently. [5] [11]

Value Methods: Temporal Difference

- Randomly initialize Q(s, a) and use interactions with the environment as a sample to update this 'bootstrap'
- Updates based on the Bellman Equation:

$$Q^{\pi}(s,a) = \mathop{\mathbb{E}}_{s'}\left[r(s,a) + \gamma \mathop{\mathbb{E}}_{a' \sim \pi}\left[Q^{\pi}(s',a')\right]\right]$$

Value Methods: Temporal Difference

- Randomly initialize Q(s, a) and use interactions with the environment as a sample to update this 'bootstrap'
- Updates based on the Bellman Equation:

$$Q^\pi(s, a) = \mathop{\mathbb{E}}_{s'}\left[r(s, a) + \gamma \mathop{\mathbb{E}}_{a' \sim \pi}\left[Q^\pi(s', a')\right]\right]$$

- Pros
  - Online learning, no need to wait for the end of an episode.
- Cons
  - Generally less stable when used with function approximation methods (more on this soon...)

Value Methods: Temporal Difference

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
    Initialize S
    Loop for each step of episode:
        Choose A from S using policy derived from Q (e.g., ε-greedy)
        Take action A, observe R, S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
        $S \leftarrow S'$
    until S is terminal

Figure: Q-Learning Pseudo-code [9]

Simplifying Assumptions

1. ~~The dynamics of the model ($p(s', r|s, a)$) are known~~
2. $\cancel{|\mathcal{S}| \ll \infty}$
3. $|\mathcal{A}| \ll \infty$

What if the state space is too large for dynamic programming?

Tasks with Large State Spaces

Example: Video Games

- Pixel input makes $|S| = \mathbb{Z}_{256}^{HxWxC}$
- Atari 2600 games make up one of the most popular benchmarks in modern RL.

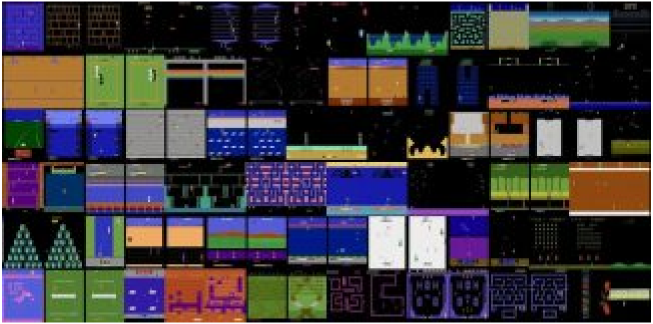


Figure: Games in the Arcade Learning Environment [1] benchmark

Deep Q Networks (DQN)

- Paramaterize Q with a neural network that can learn to recognize patterns between similar states.
- Train this network to minimize the *Mean Squared Bellman Error* (MSBE)

$$BE(s, a, r, s', d) = (r + \gamma(1 - d) \max_{a'} Q_{\theta'}(s', a')) - Q_\theta(s, a)$$

Deep Q Networks (DQN)

- Paramaterize Q with a neural network that can learn to recognize patterns between similar states.
- Train this network to minimize the *Mean Squared Bellman Error* (MSBE)

$$BE(s, a, r, s', d) = (r + \gamma(1 - d) \max_{a'} Q_{\theta'}(s', a')) - Q_{\theta}(s, a)$$

- Kind of like supervised deep learning!
  - One important difference:
    - The data distribution depends on the parameters (far from i.i.d)

Deep Q Networks (DQN)

DQNs [3] use a couple tricks to make this more like supervised learning:

1. Create a *replay buffer* $\mathcal{R}$ to store transitions (s, a, r, s', d)
   - Randomly sample from this buffer at each training step.
2. Create a *target network* to generate the the bellman error targets.
   - This is a duplicate of the original network that is not trained but is updated with fresh params every ~ 10000 steps.

The original DQN was able to learn superhuman policies on many games with dense reward signals!

Deep Q Networks (DQN)

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode $= 1, M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure: [3]

Simplifying Assumptions

1. ~~The dynamics of the model ($p(s', r|s, a)$) are known~~
2. $\cancel{|\mathcal{S}| \lll \infty}$
3. $\cancel{|\mathcal{A}| \lll \infty}$

Continuous Control Tasks

MDPs where the actions are vectors (e.g. torque on a robot's motors, acceleration of a car, degrees to turn...)

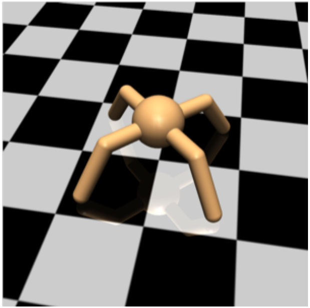


Figure: Example MuJoCo control task

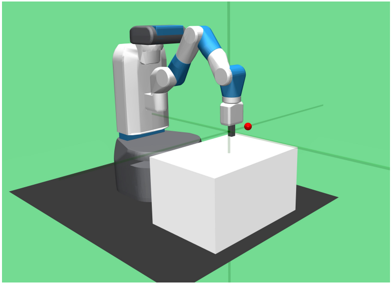


Figure: Simulated robotics task

Continuous Control Tasks

Q: Why won't DQN work?
A: It's too difficult to compute the Bellman Error, because we can't max over such a large set of actions

$$BE(s, a, r, s', d) = (r + \gamma(1 - d) \max_{a'} Q_{\theta'}(s', a')) - Q_\theta(s, a)$$

Deep Deterministic Policy Gradient (DDPG)

"Deep Q Learning for Continuous Action Spaces" [2]

- DDPG is an *Actor-Critic* method
  - Actor network $\mu_\theta(s)$
  - Critic network $Q_\phi(s, a)$
- We can get around the max operation issue by having the network learn this for us:

$$\mu_\theta(s) = \underset{a}{argmax}\, Q_\phi(s, a)$$

- How do we train this?
  - At each step, we optimize the critic network based on standard MSBE, and we optimize the actor network with gradient ascent using

$$\nabla_\theta \frac{1}{|\mathcal{B}|} \sum_{s \in \mathcal{B}} Q_\phi(s, \mu_\theta(s))$$

Deep Deterministic Policy Gradient (DDPG)

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$

Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$

repeat

  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$

  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 however many updates **do**

      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$

      Compute targets

$$y(r, s', d) = r + \gamma(1-d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} \left(Q_\phi(s,a) - y(r,s',d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s\in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta$$

    end for

  end if

until convergence

Figure: DDPG Pseudocode [4]

Twin Delayed DDPG (TD3)

- Actor-critic methods suffer from *overestimation bias*
  - Actor network learns to exploit inaccuracies in the approximation of the Q function
- Three ticks help reduce this effect:
  1. Delayed Policy Updates
     - Update the critic more often than the actor
  2. Smoothing the Q function by adding noise to the target actions

$$\mu_{\theta'}(s') \rightarrow \mu_{\theta'}(s') + \epsilon, \epsilon \sim \mathcal{N}(0, \sigma)$$

     - Force the bellman targets to be the same in the neighborhood of each action.
  3. "Clipped Double-Q Learning"
     - Train two critics and use the smallest of the two Q values.
     - Explicitly prefer underestimates of the Q function to overestimates.

Twin Delayed DDPG (TD3)

Algorithm 1 Twin Delayed DDPG

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  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 target actions

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

      Compute targets

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', a'(s'))$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      if j mod `policy_delay` $= 0$ **then**
        Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_{\phi_1}(s, \mu_\theta(s))$$

        Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$
$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$$

      end if
    end for
  end if
until convergence

Twin Delayed DDPG (TD3)

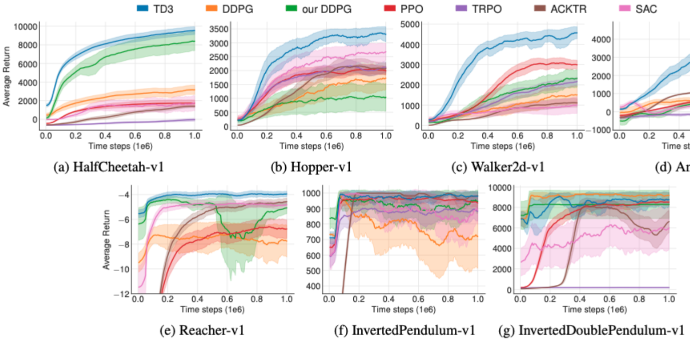
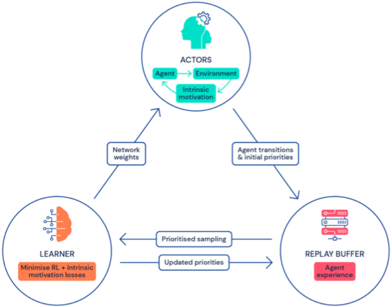


Figure 5. Learning curves for the OpenAI gym continuous control tasks. The shaded region represents half a standard deviation of the average evaluation over 10 trials. Curves are smoothed uniformly for visual clarity.

Figure: [6]

Distributed Methods

- Like many other areas of Deep Learning, Model-Free Deep RL benefits from more computation, large training sets and high quality data.
- In RL, we can't make the training set larger, but we can collect more experience
- Distributed methods run multiple actor agents in parallel, and store the transitions in a distributed replay buffer. A learner samples from the buffer to improve its parameters.



Distributed DQNs

1. Ape-X [7]
   - Distributed actors that feed to a central replay buffer. High performance at the cost of sample efficiency.

Distributed DQNs

1. Ape-X [7]
    - Distributed actors that feed to a central replay buffer. High performance at the cost of sample efficiency.
2. R2D2 [8]
    - Ape-X + RNN architecture that helps with partially observable tasks.

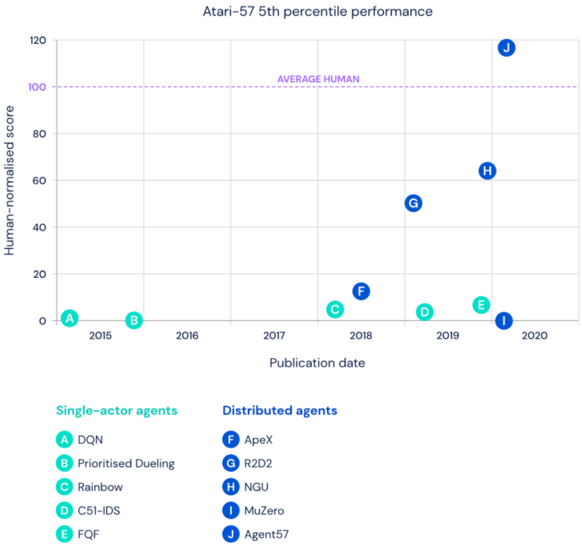
Distributed DQNs

1. Ape-X [7]
   - Distributed actors that feed to a central replay buffer. High performance at the cost of sample efficiency.
2. R2D2 [8]
   - Ape-X + RNN architecture that helps with partially observable tasks.
3. NGU [11]
   - R2D2 + a family of policies with different levels of *intrinsic motivation*
   - Great results on sparse reward games that are hardest to explore

Distributed DQNs

1. Ape-X [7]
   - Distributed actors that feed to a central replay buffer. High performance at the cost of sample efficiency.
2. R2D2 [8]
   - Ape-X + RNN architecture that helps with partially observable tasks.
3. NGU [11]
   - R2D2 + a family of policies with different levels of *intrinsic motivation*
   - Great results on sparse reward games that are hardest to explore
4. Agent57 [10]
   - NGU + a multiarmed bandit for policy hyperparameter selection.
   - Superhuman performance on all 57 ALE Games!

Distributed DQNs



Distributed DQNs

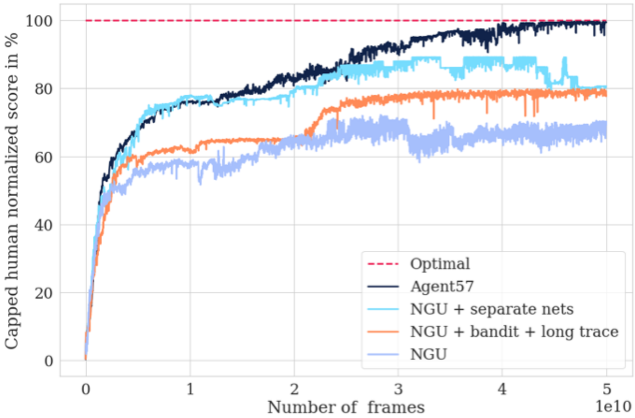


Figure: Performance of Distributed DQN variants on the 10 most challenging Atari games. Note the incredible 50B frames required to find the optimal policy![10]

Distributed DDPG

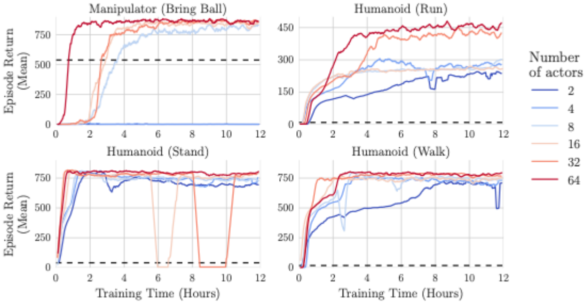


Figure 3: Performance of Ape-X DPG on four continuous control tasks, as a function of wall clock time. Performance improves as we increase the numbers of actors. The black dashed line indicates the maximum performance reached by a standard DDPG baseline over 5 days of training.

Figure: Ape-X DPG [7]

The Problem with Model-Free

The best model-free methods require millions if not billions of environment interactions to train. This creates several problems:

1. It would be difficult to apply them to problems where experience is hard to come by (e.g. real-world robots)
2. They are incredibly expensive to train and experiment with
   - It would take at least 17 *Trillion* steps to do a full comparison sweep vs Agent57 on the ALE. (5.7 Trillion per random seed...)
   - Each new training run takes roughly *17 days* even on Google's hardware.
3. They are extremely complicated to implement, and are not all open source.

References I

- Marc G Bellemare et al. "The arcade learning environment: An evaluation platform for general agents". In: *Journal of Artificial Intelligence Research* 47 (2013), pp. 253–279.
- Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2015. arXiv: 1509.02971 [cs.LG].
- Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.
- Joshua Achiam. "Spinning Up in Deep Reinforcement Learning". In: (2018).
- Yuri Burda et al. "Large-scale study of curiosity-driven learning". In: *arXiv preprint arXiv:1808.04355* (2018).
- Scott Fujimoto, Herke Van Hoof, and David Meger. "Addressing function approximation error in actor-critic methods". In: *arXiv preprint arXiv:1802.09477* (2018).

References II

- Dan Horgan et al. "Distributed prioritized experience replay". In: *arXiv preprint arXiv:1803.00933* (2018).
- Steven Kapturowski et al. "Recurrent experience replay in distributed reinforcement learning". In: (2018).
- Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
- Adrià Puigdomènech Badia et al. "Agent57: Outperforming the Atari Human Benchmark". In: *arXiv preprint arXiv:2003.13350* (2020).
- Adrià Puigdomènech Badia et al. "Never Give Up: Learning Directed Exploration Strategies". In: *arXiv preprint arXiv:2002.06038* (2020).