Asynchronous Methods for Deep Reinforcement Learning

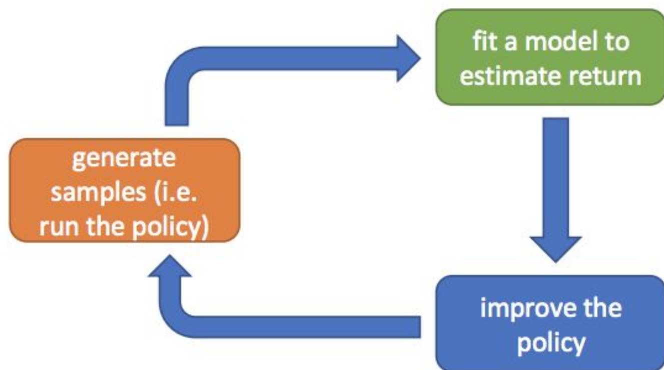
Ashwinee Panda, 6 Feb 2019

Reinforcement Learning Background

Value-based Methods

- Don’t learn policy explicitly
- Learn Q-function
    - Deep RL: Train neural network to approximate Q-function

$$Q_\phi(\mathbf{s}, \mathbf{a}) \leftarrow r(\mathbf{s}, \mathbf{a}) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}', \mathbf{a}')$$

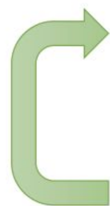


$$\mathbf{a} = \arg\max_{\mathbf{a}} Q_\phi(\mathbf{s}, \mathbf{a})$$

Policy-based Methods

- Parametrize policy with theta and update theta with gradient descent

REINFORCE algorithm:

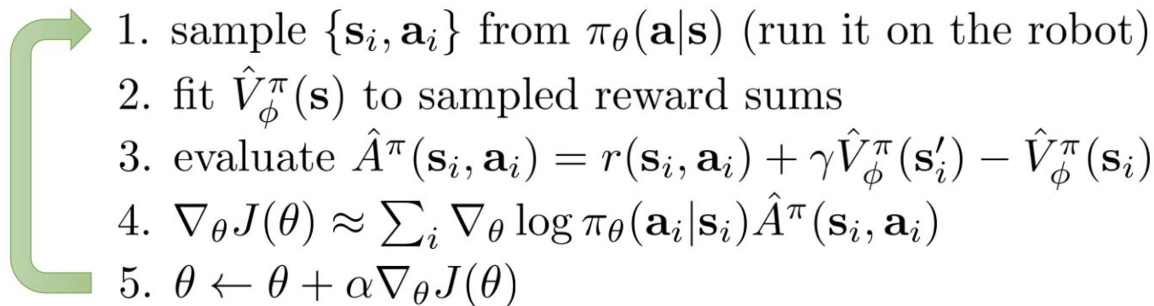


1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ (run it on the robot)
2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\right)\left(\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i)\right)$
3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

- Reduce variance by subtracting an NN baseline
- Use learned estimate of value function as baseline
- This baseline is a “critic” => “actor-critic”

Policy-based Methods

- Reduce variance by subtracting an NN baseline
- Use learned estimate of value function as baseline
- This baseline is a “critic” => “actor-critic”

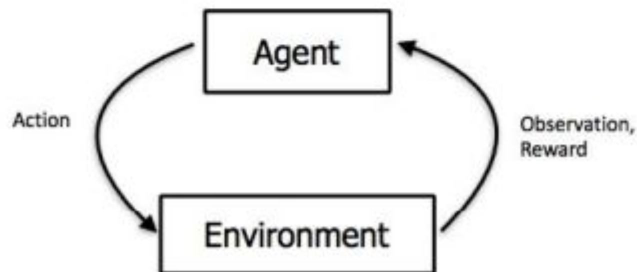

batch actor-critic algorithm:

1. sample $\{\mathbf{s}_i, \mathbf{a}_i\}$ from $\pi_\theta(\mathbf{a}|\mathbf{s})$ (run it on the robot)
2. fit $\hat{V}^\pi_\phi(\mathbf{s})$ to sampled reward sums
3. evaluate $\hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i) = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \hat{V}^\pi_\phi(\mathbf{s}'_i) - \hat{V}^\pi_\phi(\mathbf{s}_i)$
4. $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(\mathbf{a}_i|\mathbf{s}_i) \hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i)$
5. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

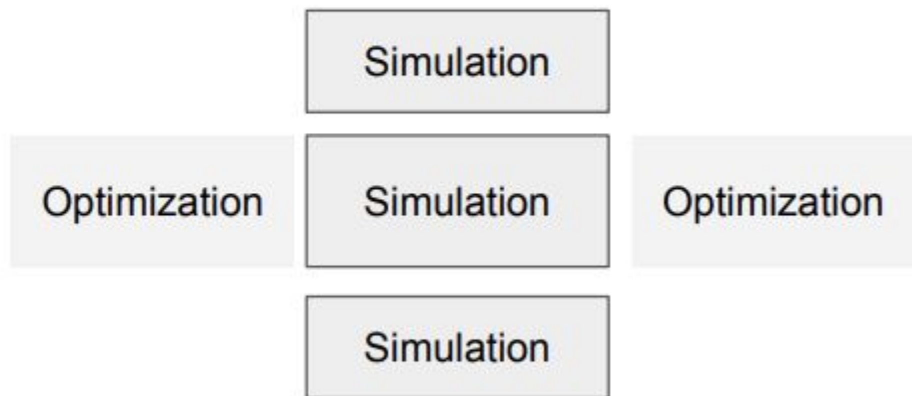
Motivations

General RL Computation

Original

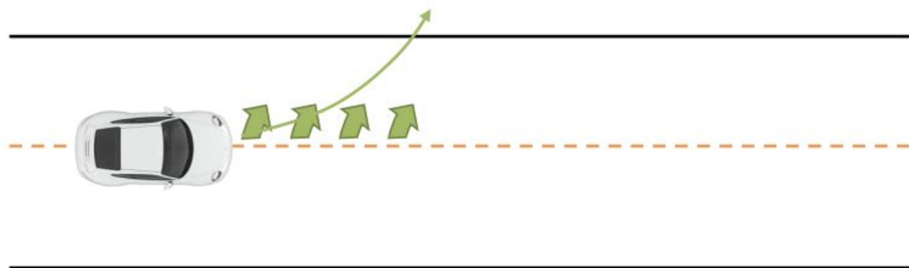


Batch Optimization



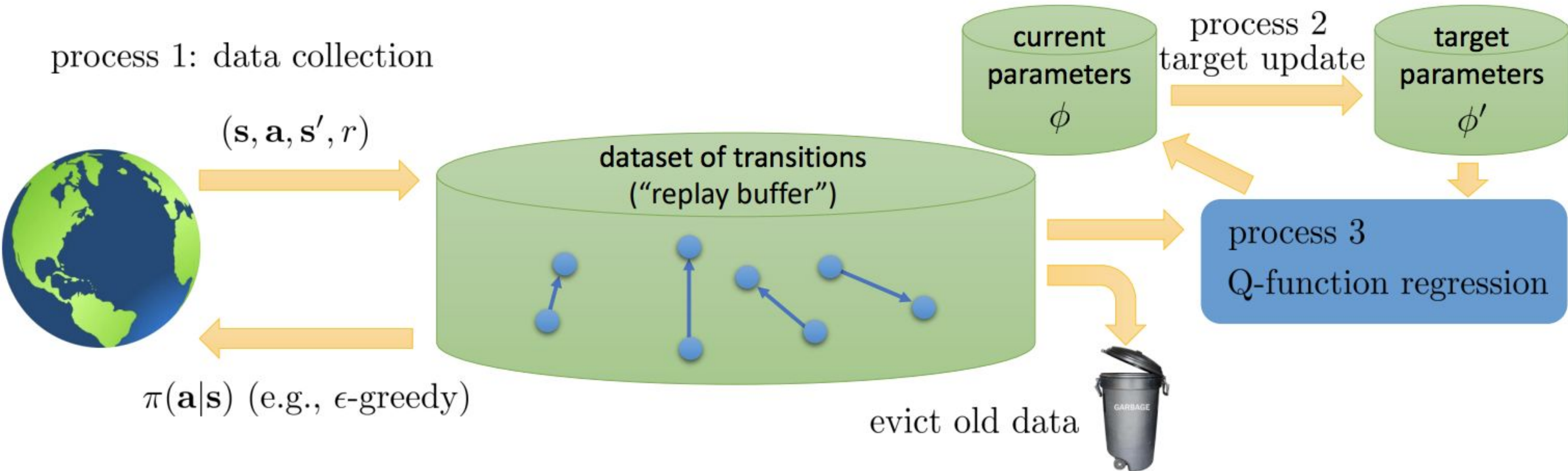
Deep Reinforcement Learning

- Problem: Instability
    - Cause: Correlation between samples
    - Cause: Incremental updates to Q change the policy => distribution
    - Cause: Correlation between Q-values and target values
- Solution: Experience Replay
    - Randomize over data distribution, removing correlations
- Problem: Limits us to off-policy RL



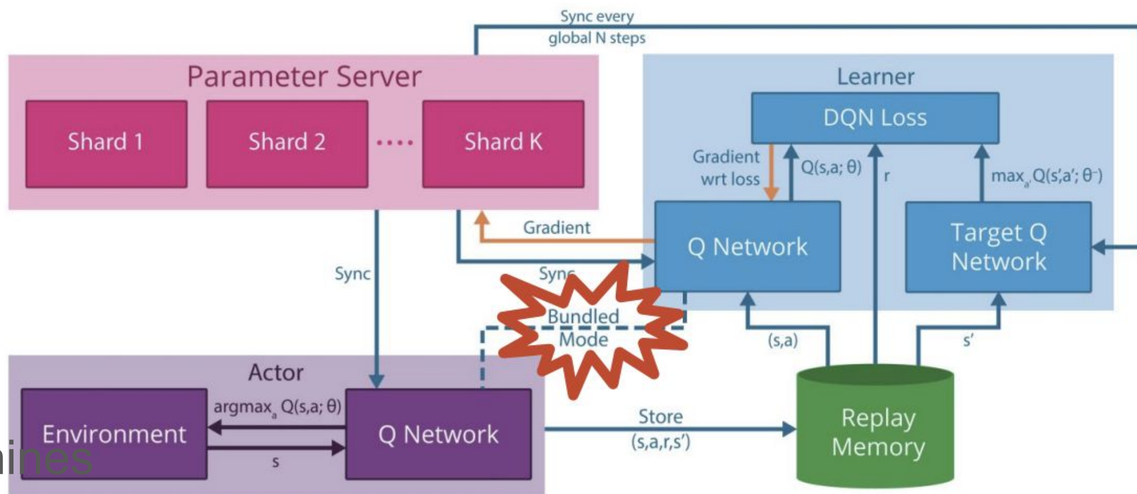
General RL Computation

When possible, reuse data



Ideas in Asynchronous RL

History of Dist. RL

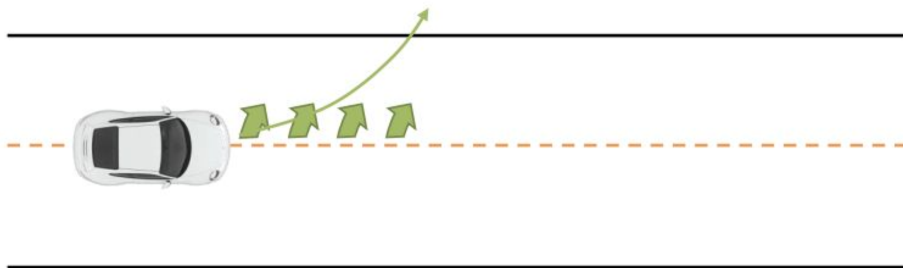


- DQN (already saw this)
- Gorilla - param. Server
  - Replay buffer
  - Enables multiple machines
- Problems w parameter servers?
  - Communication cost
  - Unavoidable when using multiple machines

Asynchronous Actor-Learners

- Rather than separate machines coordinated by a parameter server…
- Multiple CPU threads on single machine coordinated by OS
- Removes communication costs (so what?)
- Actors walk through environment and send updates to learners
- Learners use observations to compute gradients

Parallel Exploration



- Multiple actors running in parallel
- Divergence => Divergence
- Exploring different parts of environment decorrelates observations
- Can also use diff. Exploration policies for each learner
- = = > Can avoid instability due to data correlation w/o using replay buffer!
  - Allows us to use on-policy methods
- Almost-linear reduction in training time w/more actor-learners

Asynchronous RL Algorithms

Asynchronous 1-step Q Learning

- Each thread interacts w copy of env
- Computes gradient of Q-loss
- Problem: Actor-learners may overwrite!
    - Fix: Accumulate updates over several steps
- Optimization: Separate explorations

Algorithm 1 Asynchronous one-step Q-learning - pseudocode for each actor-learner thread.

// Assume global shared θ, θ^-, and counter $T = 0$.
Initialize thread step counter $t \leftarrow 0$
Initialize target network weights $\theta^- \leftarrow \theta$
Initialize network gradients $d\theta \leftarrow 0$
Get initial state s
repeat
  Take action a with ϵ-greedy policy based on $Q(s, a; \theta)$
  Receive new state s' and reward r
  $$y = \begin{cases} r & \text{for terminal } s' \\ r + \gamma \max_{a'} Q(s', a'; \theta^-) & \text{for non-terminal } s' \end{cases}$$
  Accumulate gradients wrt θ: $d\theta \leftarrow d\theta + \frac{\partial (y - Q(s,a;\theta))^2}{\partial \theta}$
  $s = s'$
  $T \leftarrow T + 1$ and $t \leftarrow t + 1$
  if $T \mod I_{target} == 0$ **then**
    Update the target network $\theta^- \leftarrow \theta$
  end if
  if $t \mod I_{AsyncUpdate} == 0$ or s is terminal **then**
    Perform asynchronous update of θ using $d\theta$.
    Clear gradients $d\theta \leftarrow 0$.
  end if
until $T > T_{max}$

Asynchronous Sarsa

- Use $r + \gamma Q(s', a'; \theta^-)$ as target with a’, s’

Asynchronous n-step Q Learning

- To compute one update, the algo:
- Selects n <= t_max / terminal actions
- Receives n <= t_max rewards
- Computes gradients for each s/a pair
- Each n-step update has n updates
- Accumulated updates applied at once

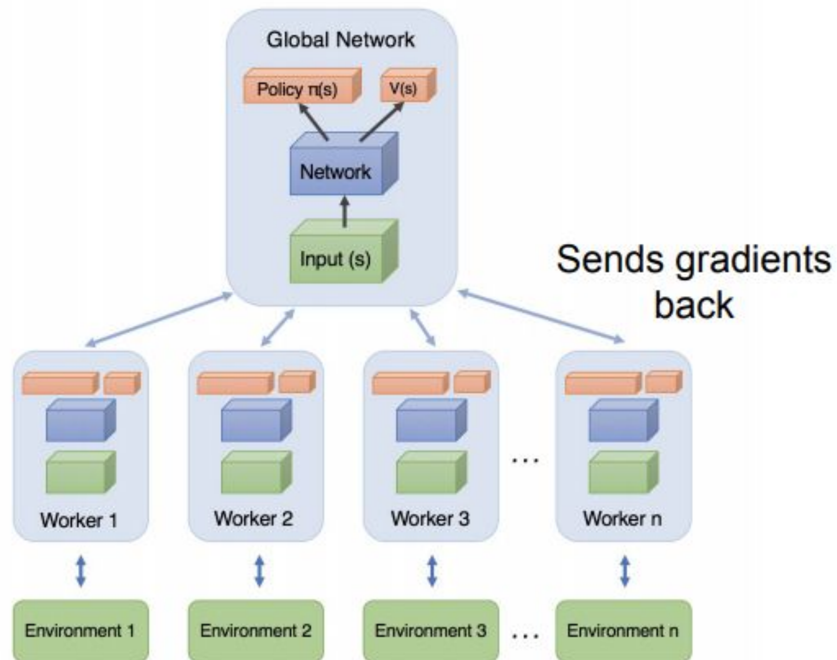
Algorithm S2 Asynchronous n-step Q-learning - pseudocode for each actor-

// Assume global shared parameter vector θ.
// Assume global shared target parameter vector θ^-.
// Assume global shared counter $T = 0$.
Initialize thread step counter $t \leftarrow 1$
Initialize target network parameters $\theta^- \leftarrow \theta$
Initialize thread-specific parameters $\theta' = \theta$
Initialize network gradients $d\theta \leftarrow 0$
repeat
    Clear gradients $d\theta \leftarrow 0$
    Synchronize thread-specific parameters $\theta' = \theta$
    $t_{start} = t$
    Get state s_t
    repeat
        Take action a_t according to the ϵ-greedy policy based on $Q(s_t, a; \theta')$
        Receive reward r_t and new state s_{t+1}
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    until terminal s_t **or** $t - t_{start} == t_{max}$
    $R = \begin{cases} 0 & \text{for terminal } s_t \\ \max_a Q(s_t, a; \theta^-) & \text{for non-terminal } s_t \end{cases}$
    for $i \in \{t-1, \ldots, t_{start}\}$ **do**
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \frac{\partial \left(R - Q(s_i, a_i; \theta')\right)^2}{\partial \theta'}$
    end for
    Perform asynchronous update of θ using $d\theta$.
    if $T \mod I_{target} == 0$ **then**
        $\theta^- \leftarrow \theta$
    end if
until $T > T_{max}$

A3C: Async. Advantage Actor-Critic

A3C (simplified)

- Each worker regularly syncs weights
- Collects sample from env
- Computes grad
- Async. Sends grad to global net



A3C

- Maintain policy
- Maintain value estimate
- Update both after t_max actions
- Grad of log policy
  - Scaled by advantage
- Advantage: diff between future

  and current value functions

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-l

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial\left(R - V(s_i;\theta'_v)\right)^2 / \partial\theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

A3C

- Share params b/t policy and value
- Policy: CNN (shared) w/softmax
- Value: CNN (shared) w/linear
- Entropy regularization
- “Critic” is value-function baseline
- Reduces variance
- Unbiased estimator

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-l

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial \left(R - V(s_i;\theta'_v)\right)^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Key Metrics and Results

Method	Training Time	Mean	Median
DQN	8 days on GPU	121.9%	47.5%
Gorila	4 days, 100 machines	215.2%	71.3%
D-DQN	8 days on GPU	332.9%	110.9%
Dueling D-DQN	8 days on GPU	343.8%	117.1%
Prioritized DQN	8 days on GPU	463.6%	127.6%
A3C, FF	1 day on CPU	344.1%	68.2%
A3C, FF	4 days on CPU	496.8%	116.6%
A3C, LSTM	4 days on CPU	623.0%	112.6%

Table 1. Mean and median human-normalized scores on 57 Atari games using the human starts evaluation metric. Supplementary Table SS3 shows the raw scores for all games.

	Number of threads				
Method	1	2	4	8	16
1-step Q	1.0	**3.0**	**6.3**	**13.3**	**24.1**
1-step SARSA	1.0	**2.8**	**5.9**	**13.1**	**22.1**
n-step Q	1.0	**2.7**	**5.9**	**10.7**	**17.2**
A3C	1.0	2.1	3.7	6.9	12.5

Table 2. The average training speedup for each method and number of threads averaged over seven Atari games. To compute the training speed-up on a single game we measured the time to required reach a fixed reference score using each method and number of threads. The speedup from using n threads on a game was defined as the time required to reach a fixed reference score using one thread divided the time required to reach the reference score using n threads. The table shows the speedups averaged over seven Atari games (Beamrider, Breakout, Enduro, Pong, Q*bert, Seaquest, and Space Invaders).

Limitations and Conclusions

Limitations and Improvements

- A3C doesn’t scale and can’t take advantage of prioritization
  - Ape-X uses priorities from Prioritized-DQN adapted for distributed setting
- Asynchrony => actors working with outdated models
  - IMPALA further improves w importance weighting (fixes policy lag)
- Actors working w diff models => aggregated update is schizophrenic
  - Fixed in A2C by removing asynchrony -turns out the benefit outweighs the costs, A2C>A3C
- Authors themselves admit they should try to use replay
  - Ape-X reintroduces replay buffer
- Too many small changes => instability
- Unsurprisingly, too many large changes => instability
  - Some “bells and whistles” can help

Conclusions

- A3C made Dist. Deep RL possible w/o relying on replay buffer
- “Model-agnostic” DDRL
- Good performance without bells and whistles made it widely used
- Though suffering from problems, still a major step forward in DDRL

DQN -> Gorilla -> A3C -> A2C/Ape-X -> IMPALA -> ?