Massively Parallel Methods for Deep Reinforcement Learning

Google DeepMind, London

Presented by liyang.cs@pku.edu.cn

Background: DistBelief

- A distributed system for training large neural networks on massive amounts of data efficiently by using two types of parallelism
- Model parallelism
- Data parallelism
- Two main components: 1) central parameter server, 2) model replicas

Background: Reinforcement Learning

- Agent interacts sequentially with an environment, with the goal of maximizing cumulative rewards
- At each step t the agent observes state s_t, selects an action a_t, and receives a reward r_t
- Discounted factor
- Action-value function $Q^{\pi}(s, a)$ is the expected return after observing state s_t and taking an action a_t under a policy π
- Bellman equation: $Q^{\star}(s, a) = E[\mathrm{r} + \lambda \max_{a'} Q^{\star}(s', a')]$

Background: Reinforcement Learning

- Core idea: represent the action-value function using a function approximator such as a neural network
- Q-network: $Q(s, a) = Q(s, a; \theta)$
- Parameters θ are optimized so as to approximately solve the Bellman Equation
- Q-learning algorithm: 1) value iteration, 2) highly unstable when combined with non-linear function approximators

Deep Q-Networks

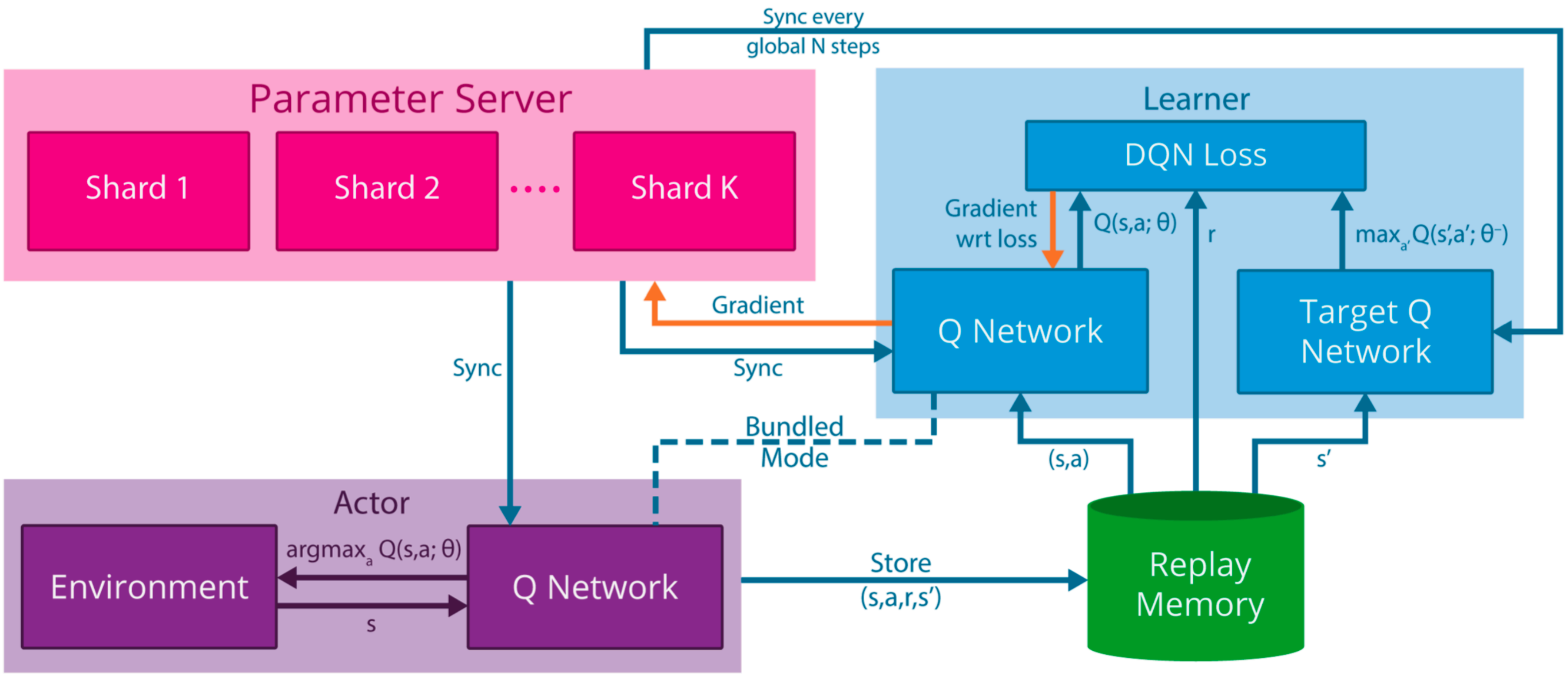
- More stable, like Q-learning, iteratively solve the Bellman equation by adjusting the parameters of the Q-networks towards bellman target
- Difference:
  - 1) using experience replay
  - 2) DQN maintains two separate Q-networks $Q(s, a; \theta)$ and $Q(s, a; \theta^{-})$ with current parameters θ and old parameters θ^{-}
- Parameter θ are updated so as to minimize the mean-squared Bellman error with respect to old parameters θ^{-}, the loss function is:
  - $L_i(\theta_i) = E[\left(r + \lambda \max_{a'} Q(s', a'; \theta_i^{-}) - Q(s, a; \theta_i)\right)^2]$

Deep Q-Networks

- For each update i, a tuple of experience (s, a, r, s') ~ U(D) is sampled uniformly from the replay memory D
- For each mini-batch, the current parameters θ are updated by a stochastic gradient descent algorithm
- The gradient is:
  - $g_i = \left(r + \lambda \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right) \nabla_{\theta_i} \mathrm{Q}(\mathrm{s}, \mathrm{a}; \theta)$
  - Each action is selected at each time-step t by an ϵ-greedy behavior with respect to the current Q-network $Q(s, a; \theta)$

Distributed Architecture

- Gorila: General Reinforcement Learning Architecture



Actors

- Agent must ultimately select action a_t to apply in its environment
- Gorila contains N_{act} different processes (instances of the same env)
- Each actor i generates its own experience $s_1^i, a_1^i, r_1^i, \ldots, s_T^i, a_T^i, r_T^i$ within the env (each actor may visit different parts of the state space)
- The quantity of experience after T time-steps is: TN_{act}
- Each actor contains a replica of the Q-network, which is used to determine the behavior (ϵ-greedy policy); the parameters of the Q-network are synchronized periodically from parameter server

Experience Replay Memory

- The experience tuples $e_t^i = (s_t^i, a_t^i, r_t^i, s_{t+1}^i)$ generated by the actors are stored in a replay memory D
- Two forms of experience replay memory:
  1. **Local replay memory**: store each actors' experience $D_t^i = \{e_1^i, e_2^i, \dots, e_t^i\}$ locally; if a single machine can store M experience tuples, the overall memory capacity becomes MN_{act}
  2. **Global replay memory**: aggregate the experience into a distributed database (overall capacity is independent of N_{act}, at the cost of additional communication overhead)

Learners

- Gorila contains N_{learn} learner processes
- Each learner contains a replica of the Q-network
- Target: compute desired changes to the parameters of the Q-network

1. For each learner update, sample a mini-batch of experience tuples $e = (s, a, r, s')$ from either a local or global experience replay memory D
2. Apply an off-policy RL algorithm such as DQN to this mini-batch of experience, in order to generate a gradient vector g_i; the gradients are communicated to the parameter server
3. The parameters of the Q-network are updated periodically from the parameter server

Parameter Server

- Use a central parameter server to maintain a distributed representation of the Q-network $Q(s, a; \theta^+)$
- The parameter vector θ^+ is split disjointly across N_{param} different machines
- Each machine is responsible for applying gradient updates to a subset of the parameters
- Apply gradients to modify the parameter vector θ^+, using an asynchronous stochastic gradient descent algorithm

Considerable Flexibility

- Avoid any individual component from becoming a bottleneck, the gorila architecture allows for arbitrary numbers of actors, learners, and parameter servers to both generate data, learn from that data, and update the model
- The simplest overall instantiation of gorila: bundled mode
- One-to-one correspondence between actors, replay memory, and learners ($N_{act} = N_{learn}$)
- Each bundle: an actor, a loca replay memory, a learner

Stability

- Challenges: disappearing nodes, slowdowns in network traffic, and slowdowns of individual machines
- Discard the gradients older than the threshold (determine the maximum time delay between local parameter and parameters in the parameter server)
- Discard gradients with absolute loss higher than the mean plus serveral standard deviations
- Use the AdaGrad update rule

Gorila DQN

Algorithm 1 Distributed DQN Algorithm

Initialise replay memory D to size P.
Initialise the training network for the action-value function $Q(s, a; \theta)$ with weights θ and target network $Q(s, a; \theta^-)$ with weights $\theta^- = \theta$.
for $episode = 1$ **to** M **do**
  Initialise the start state to s_1.
  Update θ from parameters θ^+ of the parameter server.
  for $t = 1$ **to** T **do**
    With probability ϵ take a random action a_t or else $a_t = \underset{a}{\operatorname{argmax}}\, Q(s, a; \theta)$.
    Execute the action in the environment and observe the reward r_t and the next state s_{t+1}. Store (s_t, a_t, r_t, s_{t+1}) in D.
    Update θ from parameters θ^+ of the parameter server.
    Sample random mini-batch from D. And for each tuple (s_i, a_i, r_i, s_{i+1}) set target y_t as
    if s_{i+1} is $terminal$ **then**
      $y_t = r_i$
    else
      $y_t = r_i + \gamma \max_{a'} Q(s_{i+1}, a'; \theta^-)$
    end if
    Calculate the loss $L_t = (y_t - Q(s_i, a_i; \theta)^2)$.
    Compute gradients with respect to the network parameters θ using equation 2.
    Send gradients to the parameter server.
    Every global N steps sync θ^- with parameters θ^+ from the parameter server.
  end for
end for

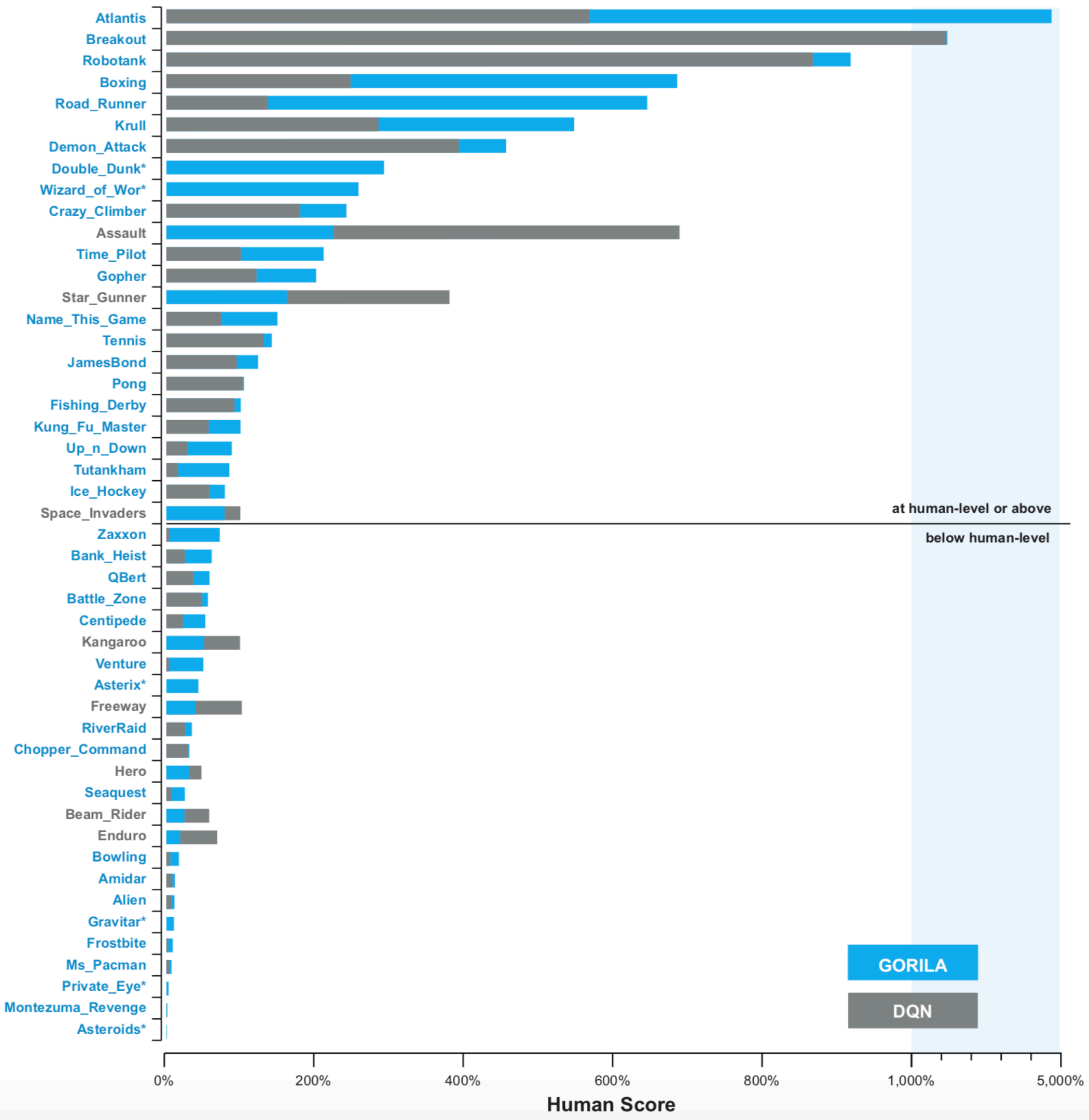
Experiments

- Evaluate Gorila by conducting experiments on 49 atari 2600 games
- 210x160 RGB video as input, the changes in the score provided as rewards
- Use the same preprocessing and network architecture of DQN (2015)
- In all experiments:
    1. $N_{param} = 31$ and $N_{learn} = N_{act} = 100$
    2. Use the bundled mode
    3. Replay memory size D=1 million frames and use ϵ-greedy with annealed from 1 to 0.1 over the first one million global updates
    4. Sync the parameter θ^{-}after every 60K parameter updates

Evaluation

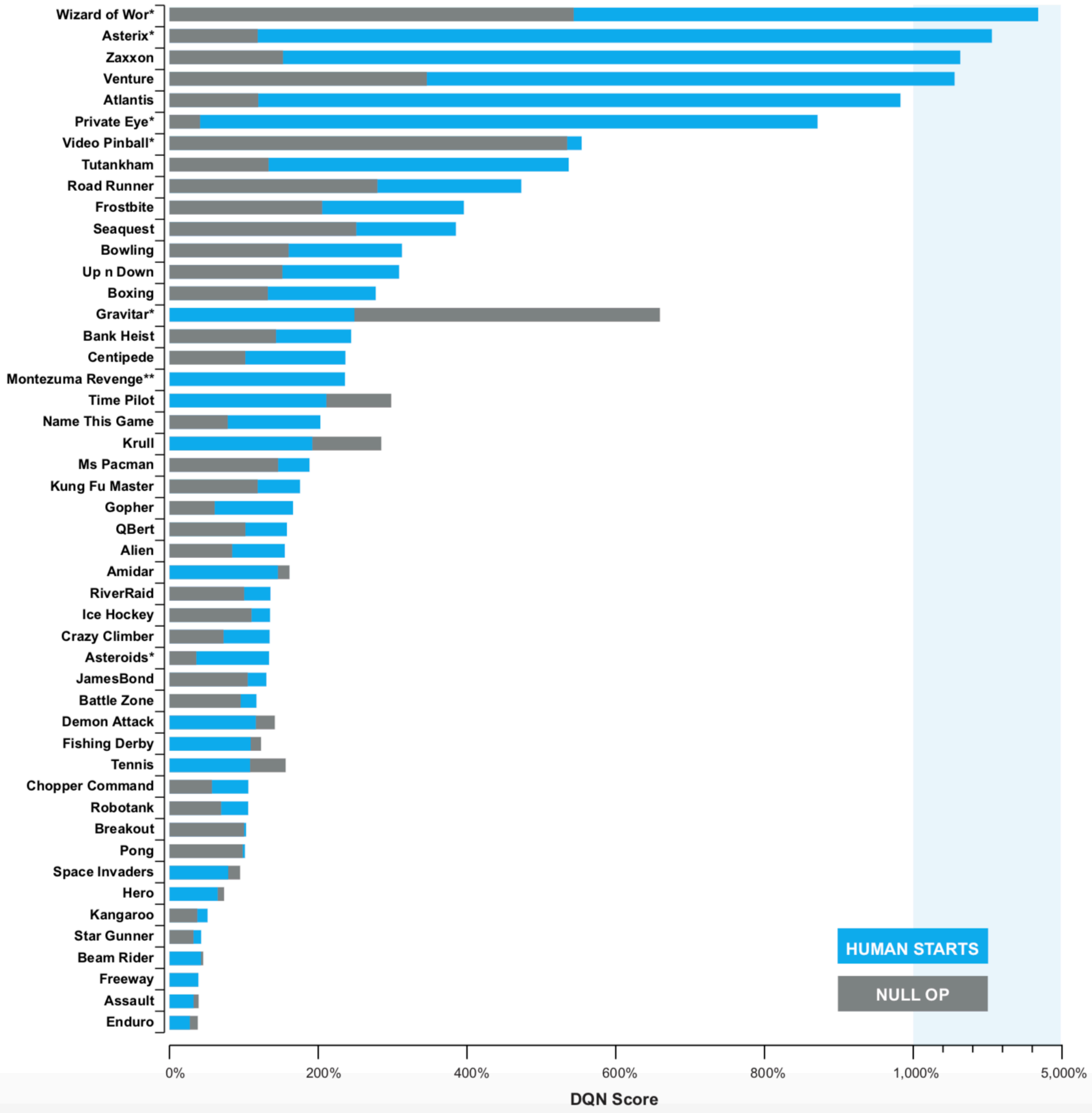
- Two types of evaluations:
  1. Follow the protocol established by DQN, each trained agent was evaluated on 30 episodes of the game it was trained on (avg score, null op starts)
  2. Aim to measure how well the agent generalizes to states it may not have trained on (100 random professional’s gameplay start points, human starts)
- 0 is the score obtained by a random agent and 100 is the score obtained by a professional human game player

Results



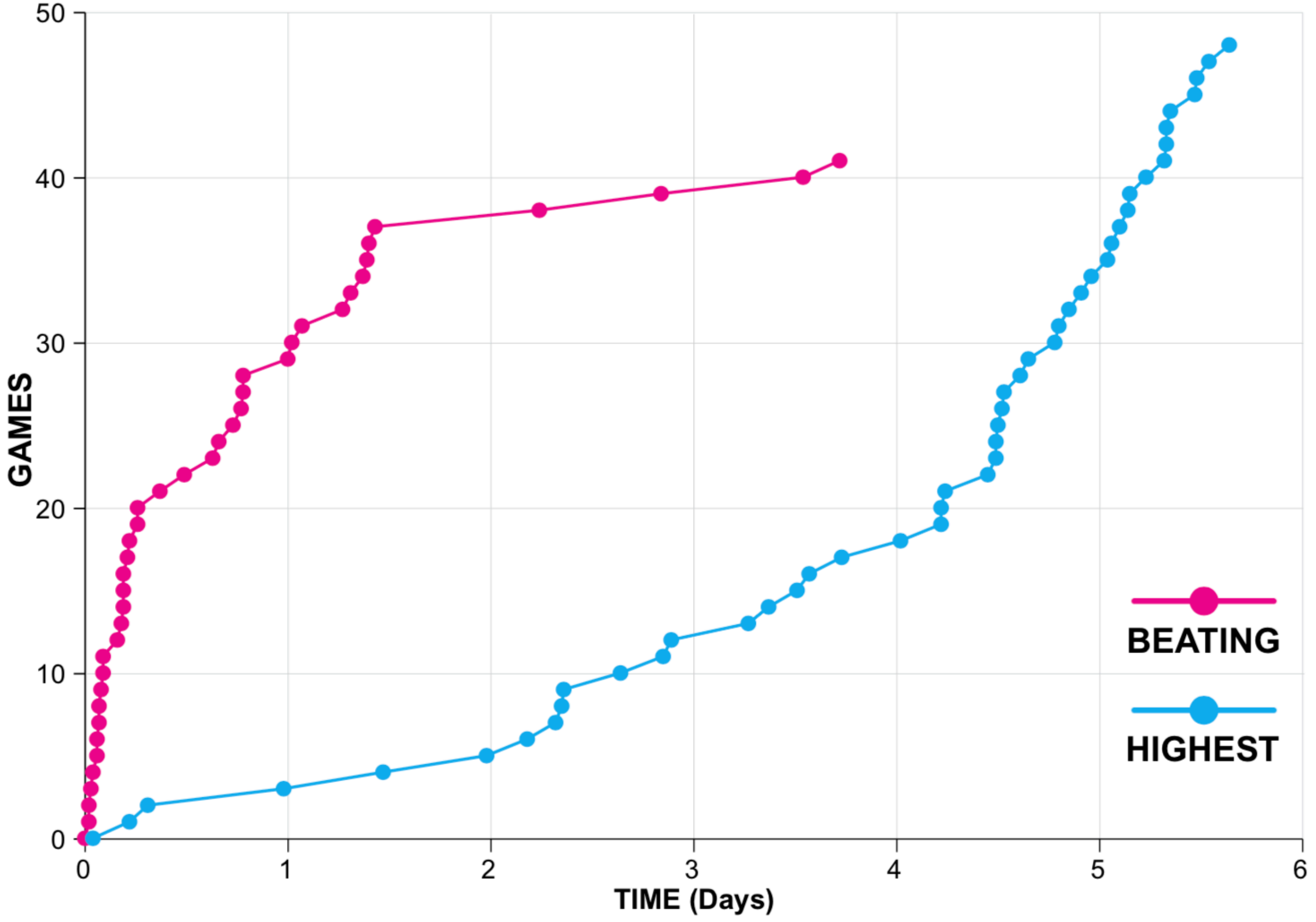
Atlantis
Breakout
Robotank
Boxing
Road_Runner
Krull
Demon_Attack
Double_Dunk*
Wizard_of_Wor*
Crazy_Climber
Assault
Time_Pilot
Gopher
Star_Gunner
Name_This_Game
Tennis
JamesBond
Pong
Fishing_Derby
Kung_Fu_Master
Up_n_Down
Tutankham
Ice_Hockey
Space_Invaders
Zaxxon
Bank_Heist
QBert
Battle_Zone
Centipede
Kangaroo
Venture
Asterix*
Freeway
RiverRaid
Chopper_Command
Hero
Seaquest
Beam_Rider
Enduro
Bowling
Amidar
Alien
Gravitar*
Frostbite
Ms_Pacman
Private_Eye*
Montezuma_Revenge
Asteroids*
at human-level or above
below human-level
GORILA
DQN
0%
200%
400%
600%
800%
1,000%
5,000%
Human Score

Results



Wizard of Wor*
Asterix*
Zaxxon
Venture
Atlantis
Private Eye*
Video Pinball*
Tutankham
Road Runner
Frostbite
Seaquest
Bowling
Up n Down
Boxing
Gravitar*
Bank Heist
Centipede
Montezuma Revenge**
Time Pilot
Name This Game
Krull
Ms Pacman
Kung Fu Master
Gopher
QBert
Alien
Amidar
RiverRaid
Ice Hockey
Crazy Climber
Asteroids*
JamesBond
Battle Zone
Demon Attack
Fishing Derby
Tennis
Chopper Command
Robotank
Breakout
Pong
Space Invaders
Hero
Kangaroo
Star Gunner
Beam Rider
Freeway
Assault
Enduro
HUMAN STARTS
NULL OP
0%
200%
400%
600%
800%
1,000%
5,000%
DQN Score

Results



Conclusion

- The first massively distributed architecture for deep reinforcement learning
- The gorila architecture acts and learns in parallel, using a distributed replay memory and distributed neural network
- Explore whether the good performance of DQN would continue to scale with additional computation
- Outperformed single GPU DQN on 41/49 games and achieved the best results in this domain, reduced the training time by an order of magnitude