Massively Parallel Methods for Deep Reinforcement Learning

Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, David Silver
{ARUNSNAIR, PRAV, BLACKWELLS, CAGDASALCICEK, RORYF, ADEMARIA, DARTHVEDA, MUSTAFASUL, CBEATTIE, SVP, LEGG, VMNIH, KORAYK, DAVIDSILVER @GOOGLE.COM }
Google DeepMind, London

Abstract

We present the first massively distributed architecture for deep reinforcement learning. This architecture uses four main components: parallel actors that generate new behaviour; parallel learners that are trained from stored experience; a distributed neural network to represent the value function or behaviour policy; and a distributed store of experience. We used our architecture to implement the Deep Q-Network algorithm (DQN) (Mnih et al., 2013). Our distributed algorithm was applied to 49 games from Atari 2600 games from the Arcade Learning Environment, using identical hyperparameters. Our performance surpassed non-distributed DQN in 41 of the 49 games and also reduced the wall-time required to achieve these results by an order of magnitude on most games.

1. Introduction

Deep learning methods have recently achieved state-of-the-art results in vision and speech domains (Krizhevsky et al., 2012; Simonyan & Zisserman, 2014; Szegedy et al., 2014; Graves et al., 2013; Dahl et al., 2012), mainly due to their ability to automatically learn high-level features from a supervised signal. Recent advances in reinforcement learning (RL) have successfully combined deep learning with value function approximation, by using a deep convolutional neural network to represent the action-value (Q) function (Mnih et al., 2013). Specifically, a new method for training such deep Q-networks, known as DQN, has enabled RL to learn control policies in complex environments with high dimensional images as inputs (Mnih et al., 2015). This method outperformed a human professional in many games on the Atari 2600 platform, using the same network architecture and hyper-parameters. However, DQN has only previously been applied to single-machine architectures, in practice leading to long training times. For example, it took 12-14 days on a GPU to train the DQN algorithm on a single Atari game (Mnih et al., 2015). In this work, our goal is to build a distributed architecture that enables us to scale up deep reinforcement learning algorithms such as DQN by exploiting massive computational resources.

One of the main advantages of deep learning is that computation can be easily parallelized. In order to exploit this scalability, deep learning algorithms have made extensive use of hardware advances such as GPUs. However, recent approaches have focused on massively distributed architectures that can learn from more data in parallel and therefore outperform training on a single machine (Coates et al., 2013; Dean et al., 2012). For example, the DistBelief framework (Dean et al., 2012) distributes the neural network parameters across many machines, and parallelizes the training by using asynchronous stochastic gradient descent (ASGD). DistBelief has been used to achieve state-of-the-art results in several domains (Szegedy et al., 2014) and has been shown to be much faster than single GPU training (Dean et al., 2012).

Existing work on distributed deep learning has focused exclusively on supervised and unsupervised learning. In this paper we develop a new architecture for the reinforcement learning paradigm. This architecture consists of four main components: parallel actors that generate new behaviour; parallel learners that are trained from stored experience; a distributed neural network to represent the value function or behaviour policy; and a distributed experience replay memory.

A unique property of RL is that an agent influences the training data distribution by interacting with its environment. In order to generate more data, we deploy multiple agents running in parallel that interact with multiple

Presented at the Deep Learning Workshop, International Conference on Machine Learning, Lille, France, 2015.

instances of the same environment. Each such *actor* can store its own record of past experience, effectively providing a distributed *experience replay memory* with vastly increased capacity compared to a single machine implementation. Alternatively this experience can be explicitly aggregated into a distributed database. In addition to generating more data, distributed actors can explore the state space more effectively, as each actor behaves according to a slightly different policy.

A conceptually distinct set of distributed *learners* reads samples of stored experience from the experience replay memory, and updates the value function or policy according to a given RL algorithm. Specifically, we focus in this paper on a variant of the DQN algorithm, which applies ASGD updates to the parameters of the Q-network. As in DistBelief, the parameters of the Q-network may also be distributed over many machines.

We applied our distributed framework for RL, known as *Gorila* (General Reinforcement Learning Architecture), to create a massively distributed version of the DQN algorithm. We applied Gorila DQN to 49 games on the Atari 2600 platform. We outperformed single GPU DQN on 41 games and outperformed human professional on 25 games. Gorila DQN also trained much faster than the non-distributed version in terms of wall-time, reaching the performance of single GPU DQN roughly ten times faster for most games.

2. Related Work

There have been several previous approaches to parallel or distributed RL. A significant part of this work has focused on distributed multi-agent systems (Weiss, 1995; Lauer & Riedmiller, 2000). In this approach, there are many agents taking actions within a single shared environment, working cooperatively to achieve a common objective. While computation is distributed in the sense of decentralized control, these algorithms focus on effective teamwork and emergent group behaviors. Another paradigm which has been explored is concurrent reinforcement learning (Silver et al., 2013), in which an agent can interact in parallel with an inherently distributed environment, e.g. to optimize interactions with multiple users on the internet. Our goal is quite different to both these distributed and concurrent RL paradigms: we simply seek to solve a single-agent problem more efficiently by exploiting parallel computation.

The MapReduce framework has been applied to standard MDP solution methods such as policy evaluation, policy iteration and value iteration, by distributing the computation involved in large matrix multiplications (Li & Schuurmans, 2011). However, this work is narrowly focused on batch methods for linear function approximation, and is not immediately applicable to non-linear representations using online reinforcement learning in environments with unknown dynamics.

Perhaps the closest prior work to our own is a parallelization of the canonical *Sarsa* algorithm over multiple machines. Each machine has its own instance of the agent and environment (Grounds & Kudenko, 2008), running a simple reinforcement learning algorithm (linear Sarsa, in this case). The changes to the parameters of the linear function approximator are periodically communicated using a peer-to-peer mechanism, focusing especially on those parameters that have changed most. In contrast, our architecture allows for client-server communication and a separation between acting, learning and parameter updates; furthermore we exploit much richer function approximators using a distributed framework for deep learning.

3. Background

3.1. DistBelief

DistBelief (Dean et al., 2012) is a distributed system for training large neural networks on massive amounts of data efficiently by using two types of parallelism. Model parallelism, where different machines are responsible for storing and training different parts of the model, is used to allow efficient training of models much larger than what is feasible on a single machine or GPU. Data parallelism, where multiple copies or replicas of each model are trained on different parts of the data in parallel, allows for more efficient training on massive datasets than a single process. We briefly discuss the two main components of the DistBelief architecture – the central parameter server and the model replicas.

The central parameter server holds the master copy of the model. The job of the parameter server is to apply the incoming gradients from the replicas to the model and, when requested, to send its latest copy of the model to the replicas. The parameter server can be sharded across many machines and different shards apply gradients independently of other shards.

Each replica maintains a copy of the model being trained. This copy could be sharded across multiple machines if, for example, the model is too big to fit on a single machine. The job of the replicas is to calculate the gradients given a mini-batch, send them to the parameter server, and to periodically query the parameter server for an updated version of the model. The replicas send gradients and request updated parameters independently of each other and hence may not be synced to the same parameters at any given time.

3.2. Reinforcement Learning

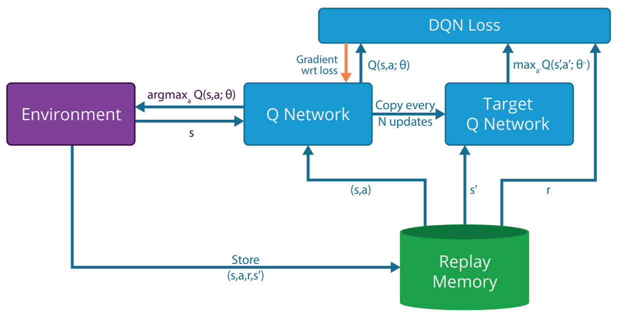

Figure 1. The DQN algorithm is composed of three main components, the Q-network $(Q(s, a; \theta))$ that defines the behavior policy, the target Q-network $(Q(s, a; \theta^-))$ that is used to generate target Q values for the DQN loss term and the replay memory that the agent uses to sample random transitions for training the Q-network.

In the reinforcement learning (RL) paradigm, the agent interacts sequentially with an environment, with the goal of maximising cumulative rewards. At each step t the agent observes state s_t, selects an action a_t, and receives a reward r_t. The agent's *policy* $\pi(a|s)$ maps states to actions and defines its behavior. The goal of an RL agent is to maximize its expected total reward, where the rewards are discounted by a factor $\gamma \in [0, 1]$ per time-step. Specifically, the *return* at time t is $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$ where T is the step when the episode terminates. The *action-value function* $Q^\pi(s, a)$ is the expected return after observing state s_t and taking an action under a policy π, $Q^\pi(s, a) = \mathbb{E}\left[R_t | s_t = s, a_t = a, \pi\right]$, and the *optimal* action-value function is the maximum possible value that can be achieved by any policy, $Q^*(s, a) = \underset{\pi}{\operatorname{argmax}}\ Q^\pi(s, a)$. The action-value function obeys a fundamental recursion known as the Bellman equation, $Q^*(s, a) = \mathbb{E}\left[r + \gamma \max_{a'} Q^*(s', a')\right]$.

One of the core ideas behind reinforcement learning is to represent the action-value function using a function approximator such as a neural network, $Q(s, a) = Q(s, a; \theta)$. The parameters θ of the so-called *Q-network* are optimized so as to approximately solve the Bellman equation. For example, the Q-learning algorithm iteratively updates the action-value function $Q(s, a; \theta)$ towards a sample of the Bellman target, $r + \gamma \max_{a'} Q(s', a'; \theta)$. However, it is well-known that the Q-learning algorithm is highly unstable when combined with non-linear function approximators such as deep neural networks (Tsitsiklis & Roy, 1997).

3.3. Deep Q-Networks

Recently, a new RL algorithm has been developed which is in practice much more stable when combined with deep Q-networks (Mnih et al., 2013; 2015). Like Q-learning, it iteratively solves the Bellman equation by adjusting the parameters of the Q-network towards the Bellman target. However, DQN, as shown in Figure 1 differs from Q-learning in two ways. First, DQN uses experience replay (Lin, 1993). At each time-step t during an agent's interaction with the environment it stores the experience tuple $e_t = (s_t, a_t, r_t, s_{t+1})$ into a replay memory $D_t = \{e_1, ..., e_t\}$.

Second, DQN maintains two separate Q-networks $Q(s, a; \theta)$ and $Q(s, a; \theta^-)$ with current parameters θ and old parameters θ^- respectively. The current parameters θ may be updated many times per time-step, and are copied into the old parameters θ^- after N iterations. At every update iteration i the current parameters θ are updated so as to minimise the mean-squared Bellman error with respect to old parameters θ^-, by optimizing the following loss function (DQN Loss),

$$L_i(\theta_i) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right)^2\right] \quad (1)$$

For each update i, a tuple of experience $(s, a, r, s') \sim U(D)$ (or a minibatch of such samples) is sampled uniformly from the replay memory D. For each sample (or minibatch), the current parameters θ are updated by a stochastic gradient descent algorithm. Specifically, θ is adjusted in the direction of the sample gradient g_i of the loss with respect to θ,

$$g_i = \left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right) \nabla_{\theta_i} Q(s, a; \theta) \quad (2)$$

Finally, actions are selected at each time-step t by an ϵ-greedy behavior with respect to the current Q-network $Q(s, a; \theta)$.

4. Distributed Architecture

We now introduce *Gorila* (General Reinforcement Learning Architecture), a framework for massively distributed reinforcement learning. The Gorila architecture, shown in Figure 2 contains the following components:

Actors. Any reinforcement learning agent must ultimately select actions a_t to apply in its environment. We refer to this process as *acting*. The Gorila architecture contains N_{act} different actor processes, applied to N_{act} corresponding instantiations of the same environment. Each actor i generates its own trajectories of experience $s_1^i, a_1^i, r_1^i, ..., s_T^i, a_T^i, r_T^i$ within the environment, and as a result each actor may visit different parts of the state space. The quantity of experience that is generated by the actors after T time-steps is approximately TN_{act}.

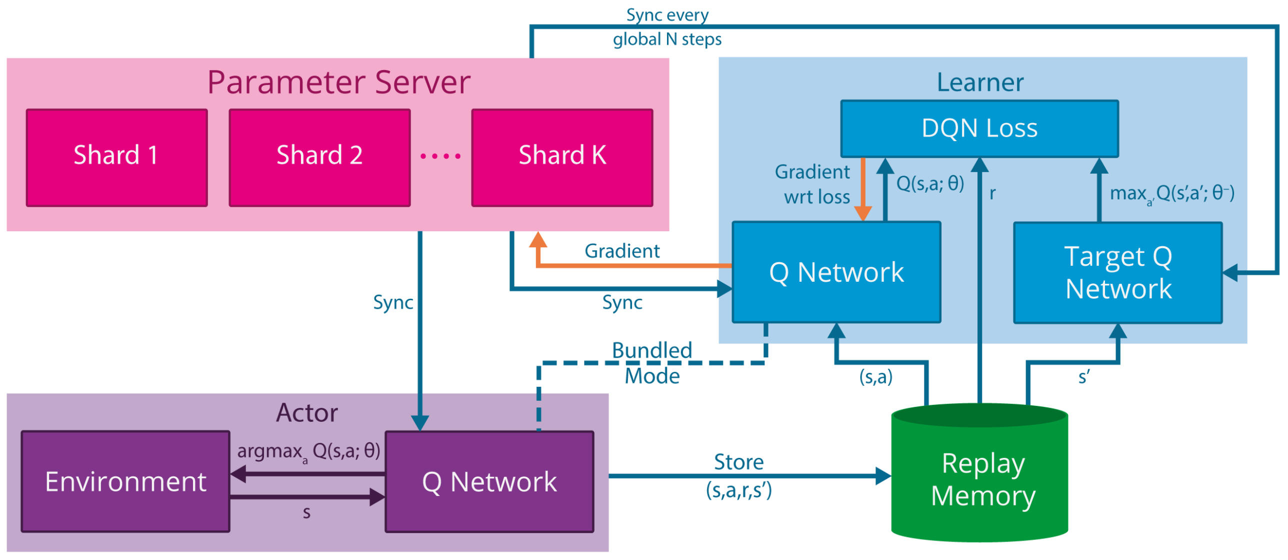

Figure 2. The Gorila agent parallelises the training procedure by separating out learners, actors and parameter server. In a single experiment, several learner processes exist and they continuously send the gradients to parameter server and receive updated parameters. At the same time, independent actors can also in parallel accumulate experience and update their Q-networks from the parameter server.

Each actor contains a replica of the Q-network, which is used to determine behavior, for example using an ϵ-greedy policy. The parameters of the Q-network are synchronized periodically from the 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. We consider two forms of experience replay memory. First, a *local* replay memory stores each actor's experience $D_t^i = \{e_1^i, ..., e_t^i\}$ locally on that actor's machine. If a single machine has sufficient memory to store M experience tuples, then the overall memory capacity becomes MN_{act}. Second, a *global* replay memory aggregates the experience into a distributed database. In this approach the overall memory capacity is independent of N_{act} and may be scaled as desired, at the cost of additional communication overhead.

Learners. Gorila contains N_{learn} learner processes. Each learner contains a replica of the Q-network and its job is to compute desired changes to the parameters of the Q-network. For each learner update k, a minibatch of experience tuples $e = (s, a, r, s')$ is sampled from either a local or global experience replay memory D (see above). The learner applies an off-policy RL algorithm such as DQN (Mnih et al., 2013) to this minibatch of experience, in order to generate a gradient vector g_i.[1] The gradients g_i are communicated to the parameter server; and the parameters of the Q-network are updated periodically from the parameter server.

Parameter server. Like DistBelief, the Gorila architecture uses 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. The parameter server receives gradients from the learners, and applies these gradients to modify the parameter vector θ^+, using an asynchronous stochastic gradient descent algorithm.

The Gorila architecture provides considerable flexibility in the number of ways an RL agent may be parallelized. It is possible to have parallel acting to generate large quantities of data into a global replay database, and then process that data with a single serial learner. In contrast, it is possible to have a single actor generating data into a local replay memory, and then have multiple learners process this data in parallel to learn as effectively as possible from this experience. However, to avoid any individual component from becoming a bottleneck, the Gorila architecture in general allows for arbitrary numbers of actors, learners, and parameter servers to both generate data, learn from that data, and update the model in a scalable and fully distributed fashion.

The simplest overall instantiation of Gorila, which we consider in our subsequent experiments, is the *bundled* mode in which there is a one-to-one correspondence between actors, replay memory, and learners ($N_{act} = N_{learn}$). Each bundle has an actor generating experience, a local replay

[1]The experience in the replay memory is generated by old behavior policies which are most likely different to the current behavior of the agent; therefore all updates must be performed off-policy (Sutton & Barto, 1998).

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 $\theta$ 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 $\theta$ from parameters $\theta^+$ of the parameter server.
  for $t = 1$ to $T$ do
    With probability $\epsilon$ 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 $\theta$ from parameters $\theta^+$ 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 $\theta$ using equation 2.
    Send gradients to the parameter server.
    Every global $N$ steps sync $\theta^-$ with parameters $\theta^+$ from the parameter server.
  end for
end for
```

memory to store that experience, and a learner that updates parameters based on samples of experience from the local replay memory. The only communication between bundles is via parameters: the learners communicate their gradients to the parameter server; and the Q-networks in the actors and learners are periodically synchronized to the parameter server.

4.1. Gorila DQN

We now consider a specific instantiation of the Gorila architecture implementing the DQN algorithm. As described in the previous section, the DQN algorithm utilizes two copies of the Q-network: a current Q-network with parameters θ and a target Q-network with parameters θ^-. The DQN algorithm is extended to the distributed implementation in Gorila as follows. The parameter server maintains the current parameters θ^+ and the actors and learners contain replicas of the current Q-network $Q(s, a; \theta)$ that are synchronized from the parameter server before every acting step. The learner additionally maintains the target Q-network $Q(s, a; \theta^-)$. The learner's target network is updated from the parameter server θ^+ after every N gradient updates in the central parameter server. Note that N is a global parameter that counts the total number of updates to the central parameter server rather than counting the updates from the local learner.

The learners generate gradients using the DQN gradient given in Equation 2. However, the gradients are not applied directly, but instead communicated to the parameter server. The parameter server then applies the updates that are accumulated from many learners.

4.2. Stability

While the DQN training algorithm was designed to ensure stability of training neural networks with reinforcement learning, training using a large cluster of machines running multiple other tasks poses additional challenges. The Gorila DQN implementation uses additional safeguards to ensure stability in the presence of disappearing nodes, slowdowns in network traffic, and slowdowns of individual machines. One such safeguard is a parameter that determines the maximum time delay between the local parameters θ (the gradients g_i are computed using θ) and the parameters θ^+ in the parameter server. All gradients older than the threshold are discarded by the parameter server. Additionally, each actor/learner keeps a running average and standard deviation of the absolute DQN loss for the data it sees and discards gradients with absolute loss higher than the mean plus several standard deviations. Finally, we used the AdaGrad update rule (Duchi et al., 2011).

5. Experiments

5.1. Experimental Set Up

We evaluated Gorila by conducting experiments on 49 Atari 2600 games using the Arcade Learning Environment (Bellemare et al., 2012). Atari games provide a challenging and diverse set of reinforcement learning problems where an agent must learn to play the games directly from 210×160 RGB video input with only the changes in the score provided as rewards. We closely followed the experimental setup of DQN (Mnih et al., 2015) using the same preprocessing and network architecture. We preprocessed the 210×160 RGB images by downsampling them to 84×84 and extracting the luminance channel.

The Q-network $Q(s, a; \theta)$ had 3 convolutional layers followed by a fully-connected hidden layer. The $84 \times 84 \times 4$ input to the network is obtained by concatenating the images from four previous preprocessed frames. The first convolutional layer had 32 filters of size $4 \times 8 \times 8$ and stride 4. The second convolutional layer had 64 filters of size $32 \times 4 \times 4$ with stride 2, while the third had 64 filters with size $64 \times 3 \times 3$ and stride 1. The next layer had 512 fully-connected output units, which is followed by a linear fully-connected output layer with a single output unit for each valid action. Each hidden layer was followed by a rectifier nonlinearity.

We have used the same frame skipping step implemented in (Mnih et al., 2015) by repeating every action a_t over the next 4 frames.

In all experiments, Gorila DQN used: $N_{param} = 31$ and $N_{learn} = N_{act} = 100$. We use the bundled mode. Replay memory size $D = 1$ million frames and used ϵ-greedy as the behaviour policy with ϵ annealed from 1 to 0.1 over the first one million global updates. Each learner syncs the parameters θ^{-} of its target network after every 60K parameter updates performed in the parameter server.

5.2. Evaluation

We used two types of evaluations. The first follows the protocol established by DQN. Each trained agent was evaluated on 30 episodes of the game it was trained on. A random number of frames were skipped by repeatedly taking the null or do nothing action before giving control to the agent in order to ensure variation in the initial conditions. The agents were allowed to play until the end of the game or up to 18000 frames (5 minutes), whichever came first, and the scores were averaged over all 30 episodes. We refer to this evaluation procedure as *null op starts*.

Testing how well an agent generalizes is especially important in the Atari domain because the emulator is completely deterministic. Our second evaluation method, which we call *human starts*, aims to measure how well the agent generalizes to states it may not have trained on. To that end, we have introduced 100 random starting points that were sampled from a human professional's gameplay for each game. To evaluate an agent, we ran it from each of the 100 starting points until the end of the game or until a total of 108000 frames (equivalent to 30 minutes) were played counting the frames the human played to reach the starting point. The total score accumulated only by the agent (not considering any points won by the human player) were averaged to obtain the evaluation score.

In order to make it easier to compare results on 49 games with a greatly varying range of scores we present the results on a scale where 0 is the score obtained by a random agent and 100 is the score obtained by a professional human game player. The random agent selected actions uniformly at random at 10Hz and it was evaluated using the same starting states as the agents for both kinds of evaluations (*null op* starts and *human starts*).

We selected hyperparameter values by performing an informal search on the games of Breakout, Pong and Seaquest which were then fixed for all the games. We have trained Gorila DQN 5 times on each game using the same fixed hyperparameter settings and random network initializations. Following DQN, we periodically evaluated each model during training and kept the best performing network parameters for the final evaluation. We average these final evaluations over the 5 runs, and compare the mean evaluations with DQN and human expert scores.

6. Results

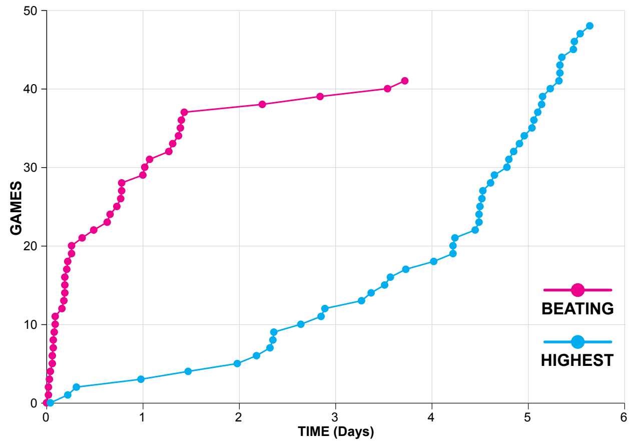

Figure 4. The time required by Gorila DQN to surpass single DQN performance (red curve) and to reach its peak performance (blue curve).

We first compared Gorila DQN agents trained for up to 6 days to single GPU DQN agents trained for 12-14 days. Figure 3 shows the normalized scores under the human starts evaluation. Using human starts Gorila DQN outperformed single GPU DQN on 41 out of 49 games given roughly one half of the training time of single GPU DQN. On 22 of the games Gorila DQN obtained double the score of single GPU DQN, and on 11 games Gorila DQN's score was 5 times higher. Similarly, using the original *null op starts* evaluation Gorila DQN outperformed the single GPU DQN on 31 out of 49 games. These results show that parallel training significantly improved performance in less training time. Also, better results on *human starts* compared to *null op starts* suggest that Gorila DQN is especially good at generalizing to potentially unseen states compared to single GPU DQN. Figure 5 further illustrates these improvements in generalization by showing Gorila DQN scores with human starts normalized with respect to GPU DQN scores with human starts (blue bars) and Gorila DQN scores from null op starts normalized by GPU DQN scores from null op starts (gray bars). In fact, Gorila DQN performs at a level similar or superior to a human professional (75% of the human score or above) in 25 games despite starting from states sampled from human play. One possible reason for the improved generalization is the sig-

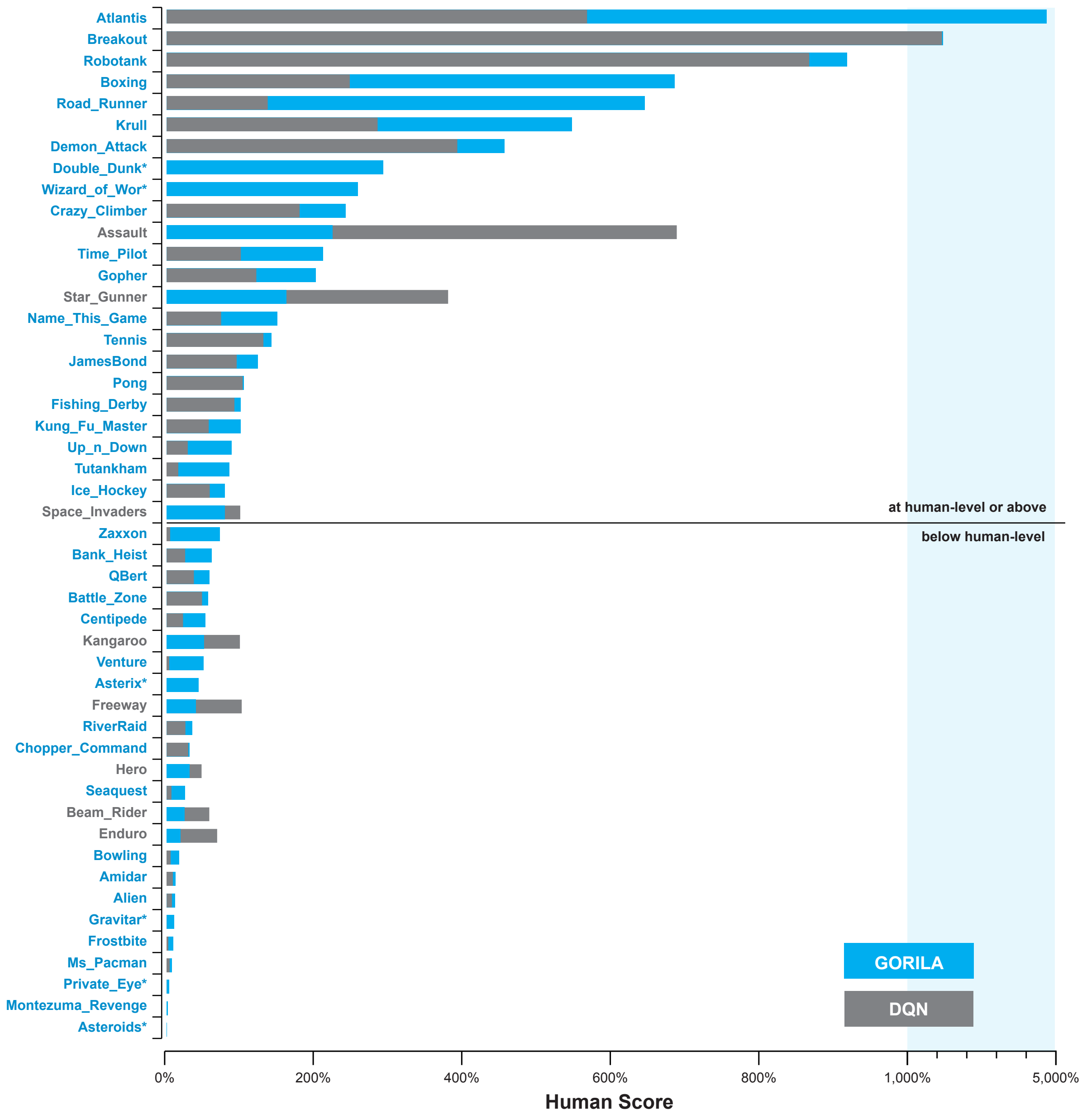

Figure 3. Performance of the Gorila agent on 49 Atari games with human starts evaluation compared with DQN (Mnih et al., 2015) performance with scores normalized to expert human performance. Font color indicates which method has the higher score. *Not showing DQN scores for Asterix, Asteroids, Double Dunk, Private Eye, Wizard Of Wor and Gravitar because the DQN human starts scores are less than the random agent baselines. Also not showing Video Pinball because the human expert scores are less than the random agent scores.

nificant increase in the number of states Gorila DQN sees by using 100 parallel actors.

We next look at how the performance of Gorila DQN improved during training. Figure 4 shows how quickly Gorila DQN reached the performance of single GPU DQN and how quickly Gorila DQN reached its own best score under the human starts evaluation. Gorila DQN surpassed the best single GPU DQN scores on 19 games in 6 hours, 23 games in 12 hours, 30 in 24 hours and 38 games in 36 hours (red curve). This is a roughly an order of magnitude reduction in training time required to reach the single process DQN score. On some games Gorila DQN achieved its best score in under two days but for most of the games the performance keeps improving with longer training time (blue curve).

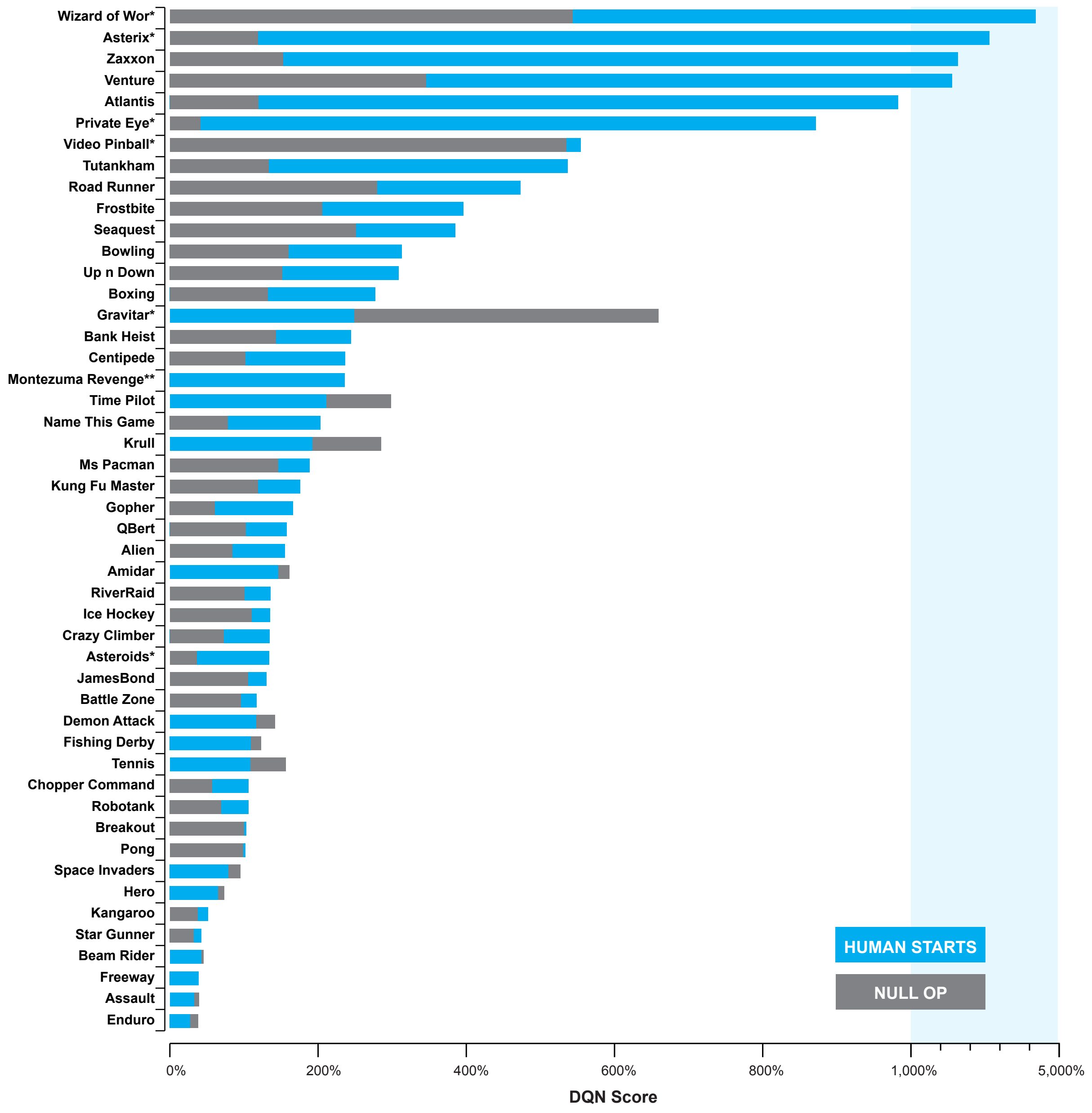

Figure 5. Performance of the Gorila agent on 49 Atari games with human starts and null op evaluations normalized with respect to DQN human start and null op scores respectively. This figure shows the generalization improvements of Gorila compared to DQN. *Using a score of 0 for the human starts random agent score for Asterix, Asteroids, Double Dunk, Private Eye, Wizard Of Wor and Gravitar because the human starts DQN scores are less than the random agent scores. Not showing Double Dunk because both the DQN scores and the random agent scores are negative. **Not showing null op scores for Montezuma Revenge because both the human start scores and random agent scores are 0.

7. Conclusion

In this paper we have introduced 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. We applied Gorila to an asynchronous variant of the state-of-the-art DQN algorithm. A single machine had previously achieved state-of-the-art results in the challenging suite of Atari 2600 games, but it was not previously known whether the good performance of DQN would continue to scale with additional computation. By leveraging massive parallelism, Gorila DQN significantly outperformed single-GPU DQN on 41 out of 49 games; achieving by far the best results in this domain to date. Gorila takes a further step towards fulfilling the promise of deep learning in RL: a scalable architecture that performs better and better with increased computation and memory.

References

Bellemare, Marc G, Naddaf, Yavar, Veness, Joel, and Bowling, Michael. The arcade learning environment: An evaluation platform for general agents. *arXiv preprint arXiv:1207.4708*, 2012.

Coates, Adam, Huval, Brody, Wang, Tao, Wu, David, Catanzaro, Bryan, and Andrew, Ng. Deep learning with cots hpc systems. In *Proceedings of The 30th International Conference on Machine Learning*, pp. 1337–1345, 2013.

Dahl, George E, Yu, Dong, Deng, Li, and Acero, Alex. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *Audio, Speech, and Language Processing, IEEE Transactions on*, 20(1):30–42, 2012.

Dean, Jeffrey, Corrado, Greg, Monga, Rajat, Chen, Kai, Devin, Matthieu, Mao, Mark, Senior, Andrew, Tucker, Paul, Yang, Ke, Le, Quoc V, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pp. 1223–1231, 2012.

Duchi, John, Hazan, Elad, and Singer, Yoram. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

Graves, Alex, Mohamed, A-R, and Hinton, Geoffrey. Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 6645–6649. IEEE, 2013.

Grounds, Matthew and Kudenko, Daniel. Parallel reinforcement learning with linear function approximation. In *Proceedings of the 5th, 6th and 7th European Conference on Adaptive and Learning Agents and Multi-agent Systems: Adaptation and Multi-agent Learning*, pp. 60–74. Springer-Verlag, 2008.

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoff. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pp. 1106–1114, 2012.

Lauer, Martin and Riedmiller, Martin. An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *In Proceedings of the Seventeenth International Conference on Machine Learning*, pp. 535–542. Morgan Kaufmann, 2000.

Li, Yuxi and Schuurmans, Dale. Mapreduce for parallel reinforcement learning. In *Recent Advances in Reinforcement Learning - 9th European Workshop, EWRL 2011, Athens, Greece, September 9-11, 2011, Revised Selected Papers*, pp. 309–320, 2011.

Lin, Long-Ji. Reinforcement learning for robots using neural networks. Technical report, DTIC Document, 1993.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Graves, Alex, Antonoglou, Ioannis, Wierstra, Daan, and Riedmiller, Martin. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*. 2013.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A., Veness, Joel, Bellemare, Marc G., Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K., Ostrovski, Georg, Petersen, Stig, Beattie, Charles, Sadik, Amir, Antonoglou, Ioannis, King, Helen, Kumaran, Dharshan, Wierstra, Daan, Legg, Shane, and Hassabis, Demis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015. URL `http://dx.doi.org/10.1038/nature14236`.

Silver, David, Newnham, Leonard, Barker, David, Weller, Suzanne, and McFall, Jason. Concurrent reinforcement learning from customer interactions. In *Proceedings of the 30th International Conference on Machine Learning*, pp. 924–932, 2013.

Simonyan, Karen and Zisserman, Andrew. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Sutton, R. and Barto, A. *Reinforcement Learning: an Introduction*. MIT Press, 1998.

Szegedy, Christian, Liu, Wei, Jia, Yangqing, Sermanet, Pierre, Reed, Scott, Anguelov, Dragomir, Erhan, Dumitru, Vanhoucke, Vincent, and Rabinovich, Andrew. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

Tsitsiklis, J. and Roy, B. Van. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997.

Weiss, Gerhard. Distributed reinforcement learning. 15: 135–142, 1995.