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**

Abstract

We propose a conceptually simple and lightweight framework for deep reinforcement learning that uses asynchronous gradient descent for optimization of deep neural network controllers. We present asynchronous variants of four standard reinforcement learning algorithms and show that parallel actor-learners have a stabilizing effect on training allowing all four methods to successfully train neural network controllers. The best performing method, an asynchronous variant of actor-critic, surpasses the current state-of-the-art on the Atari domain while training for half the time on a single multi-core CPU instead of a GPU. Furthermore, we show that asynchronous actor-critic succeeds on a wide variety of continuous motor control problems as well as on a new task involving finding rewards in random 3D mazes using a visual input.

1 Introduction

Deep neural networks provide rich representations that can enable reinforcement learning (RL) algorithms to perform effectively. However, it was previously thought that the combination of simple online RL algorithms with deep neural networks was fundamentally unstable. Instead, a variety of solutions have been proposed to stabilize the algorithm [Riedmiller, 2005, Mnih et al., 2013, 2015, Van Hasselt et al., 2015, Schulman et al., 2015a]. These approaches share a common idea: the sequence of observed data encountered by an online RL agent is non-stationary, and online RL updates are strongly correlated. By storing the agent's data in an experience replay memory, the data can be batched [Riedmiller, 2005, Schulman et al., 2015a] or randomly sampled [Mnih et al., 2013, 2015, Van Hasselt et al., 2015] from different time-steps. Aggregating over memory in this way reduces non-stationarity and decorrelates updates, but at the same time limits the methods to off-policy reinforcement learning algorithms.

Deep RL algorithms based on experience replay have achieved unprecedented success in challenging domains such as Atari 2600. However, experience replay has several drawbacks:

it uses more memory and more computation per real interaction; and it requires off-policy learning algorithms that can update from data generated by an older policy.

In this paper we provide a very different paradigm for deep reinforcement learning. Instead of experience replay, we asynchronously execute multiple agents in parallel, on multiple instances of the environment. This parallelism also decorrelates the agents' data into a more stationary process, since at any given time-step the parallel agents will be experiencing a variety of different states. This simple idea enables a much larger spectrum of fundamental on-policy RL algorithms, such as Sarsa, n-step methods, and actor-critic methods, as well as off-policy RL algorithms such as Q-learning, to be applied robustly and effectively using deep neural networks.

The asynchronous reinforcement learning paradigm also offers practical benefits. Whereas previous approaches to deep reinforcement learning rely heavily on specialized hardware such as GPUs [Mnih et al., 2015, Van Hasselt et al., 2015, Schaul et al., 2015] or massively distributed architectures [Nair et al., 2015], our experiments run on a single machine with a standard multi-core CPU. When applied to a variety of Atari 2600 domains, on many games asynchronous reinforcement learning achieves better results, in far less time than previous GPU-based algorithms, using far less resource than massively distributed approaches. Furthermore, the best of the proposed methods, asynchronous advantage actor-critic (A3C), was also able to master a variety of continuous motor control tasks as well as learn general strategies for exploring 3D mazes purely from visual inputs. We believe that the success of A3C on both 2D and 3D games, discrete and continuous action spaces, as well as its ability to train feedforward and recurrent agents makes it the most general and successful reinforcement learning agent to date.

2 Related Work

The General Reinforcement Learning Architecture (Gorila) of Nair et al. [2015] performs asynchronous training of reinforcement learning agents in a distributed setting. In Gorila, each process contains an actor that acts in its own copy of the environment, a separate replay memory, and a learner that samples data from the replay memory and computes gradients of the DQN loss [Mnih et al., 2015] with respect to the policy parameters. The gradients are asynchronously sent to a central parameter server which updates a central copy of the model. The updated policy parameters are sent to the actor-learners at fixed intervals. By using 100 separate actor-learner processes and 30 parameter server instances, for a total of 130 CPU cores, Gorila was able to significantly outperform DQN over 49 Atari games. On many games Gorila reached the score achieved by DQN over 20 times faster than DQN. We also note that a similar way of parallelizing DQN was proposed by Chavez et al. [2015].

In earlier work, Li and Schuurmans [2011] applied the Map Reduce framework to parallelizing batch reinforcement learning methods with linear function approximation. Parallelism was used to speed up large matrix operations but not to parallelize the collection of experience or stabilize learning. Grounds and Kudenko [2008] proposed a parallel version of the Sarsa algorithm that uses multiple separate actor-learners to accelerate training. Each actor-learner learns separately and periodically sends updates to weights that have changed significantly to the other learners using peer-to-peer communication.

Tsitsiklis [1994] studied convergence properties of Q-learning in the asynchronous optimization setting. These results show that Q-learning is still guaranteed to converge when some of the information is outdated as long as outdated information is always eventually discarded and several other technical assumptions are satisfied. Even earlier, Bertsekas [1982] studied the related problem of distributed dynamic programming.

Another related area of work is in evolutionary methods, which are often straightforward to parallelize by distributing fitness evaluations over multiple machines or threads [Tomassini, 1999]. Such parallel evolutionary approaches have recently been applied to some visual reinforcement learning tasks. In one example, Koutník et al. [2014] evolved convolutional neural network controllers for the TORCS driving simulator by performing fitness evaluations on 8 CPU cores in parallel.

3 Background

3.1 Reinforcement Learning

We consider the standard reinforcement learning setting where an agent interacts with an environment $\mathcal{E}$ over a number of discrete time steps. At each time step t, the agent receives a state s_t and selects an action a_t from some set of possible actions $\mathcal{A}$ according to its policy π, where π is a mapping from states s_t to actions a_t. In return, the agent receives the next state s_{t+1} and receives a scalar reward r_t. The process continues until the agent reaches a terminal state after which the process restarts. The return $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ is the total accumulated return from time step t with discount factor $\gamma \in (0, 1]$. The goal of the agent is to maximize the expected return from each state s_t.

The action value $Q^\pi(s, a) = \mathbb{E}\left[R_t | s_t = s, a\right]$ is the expected return for selecting action a in state s and following policy π. The optimal value function $Q^*(s, a) = \max_\pi Q^\pi(s, a)$ gives the maximum action value for state s and action a achievable by any policy. Similarly, the value of state s under policy π is defined as $V^\pi(s) = \mathbb{E}\left[R_t | s_t = s\right]$ and is simply the expected return for following policy π from state s.

In value-based model-free reinforcement learning methods, the action value function is represented using a function approximator, such as a neural network. Let $Q(s, a; \theta)$ be an approximate action-value function with parameters θ. The updates to θ can be derived from a variety of reinforcement learning algorithms. One example of such an algorithm is Q-learning, which aims to directly approximate the optimal action value function: $Q^*(s, a) \approx Q(s, a; \theta)$. In one-step Q-learning, the parameters θ of the action value function $Q(s, a; \theta)$ are learned by iteratively minimizing a sequence of loss functions, where the ith loss function defined as

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

where s' is the state encountered after state s.

Alternatively, Sarsa [Rummery and Niranjan, 1994, Sutton and Barto, 1998] is a widely used on-policy algorithm where the approximate action value function Q is updated by minimizing the following loss function during the ith iteration

$$L_i(\theta_i) = \mathbb{E}\left(r + \gamma Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\right)^2, \tag{2}$$

where a' is the action taken by the agent in state s'. In tabular environments, where $Q(s, a; \theta)$ is a lookup table, both Q-learning and Sarsa are known to converge to the optimal value function Q^* under certain conditions [Jaakkola et al., 1994, Tsitsiklis, 1994, Singh et al., 2000].

We refer to the above methods as one-step Q-learning and one-step Sarsa because they update the action value $Q(s, a)$ toward one-step returns $r + \gamma \max_{a'} Q(s', a'; \theta)$ and $r + \gamma Q(s', a'; \theta)$ respectively. One drawback of using one-step methods is that obtaining a reward r only directly affects the value of the state action pair s, a that led to the reward. The values of other state action pairs are affected only indirectly through the updated value $Q(s, a)$. This can make the learning process slow since many updates are required the propagate a reward to the relevant preceding states and actions.

One way of propagating rewards faster is by using n-step returns [Watkins, 1989, Peng and Williams, 1996]. In n-step Q-learning, $Q(s, a)$ is updated toward the n-step return defined as

$$r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n} + \max_a \gamma^n Q(s_{t+n+1}, a). \tag{3}$$

This results in a single reward r directly affecting the values of n preceding state action pairs. This makes the process of propagating rewards to relevant state-action pairs potentially much more efficient.

In contrast to value-based methods, policy-based model-free methods directly parameterize the policy $\pi(a|s; \theta)$ and update the parameters θ by performing, typically approximate, gradient ascent on $\mathbb{E}[R_t]$. One example of such a method is the REINFORCE family of algorithms due to Williams [1992]. Standard REINFORCE updates the policy parameters θ in the direction $\nabla_\theta \log \pi(a_t|s_t; \theta) R_t$, which is an unbiased estimate of $\nabla_\theta \mathbb{E}[R_t]$. It is possible to reduce the variance of this estimate while keeping it unbiased by subtracting a learned function of the state $b_t(s_t)$, known as a baseline [Williams, 1992], from the return

$$\nabla_\theta \log \pi(a_t|s_t; \theta) \left(R_t - b_t(s_t)\right). \tag{4}$$

A learned estimate of the value function is commonly used as the baseline $b_t(s_t) \approx V^\pi(s_t)$ leading to a much lower variance estimate of the policy gradient. When an approximate value function is used as the baseline, the quantity $R_t - b_t$ used to scale the policy gradient can be seen as an estimate of the *advantage* of action a_t in state s_t, or $A(a_t, s_t) = Q(a_t, s_t) - V(s_t)$, because R_t is an estimate of $Q^\pi(a_t, s_t)$ and b_t is an estimate of $V^\pi(s_t)$. This approach can be viewed as an actor-critic architecture where the policy π is the actor and the baseline b_t is the critic [Sutton and Barto, 1998, Degris et al., 2012].

3.2 Deep Q Networks

Temporal difference learning methods, such as Q-learning, have been known to diverge when used with nonlinear function approximators [Tsitsiklis and Roy, 1997]. The recently introduced variant of Q-learning for training Deep Q Networks [Mnih et al., 2015] made use of two techniques for avoiding such divergences in practice. First, an experience replay memory mechanism due to Lin [1993] was used to perform Q-learning updates on random samples of past experience instead on the most recent samples of experience. Experience replay reduces the correlations between successive updates applied to the network thereby

making the training data less non-stationary. Second, the network used for computing Q-learning targets was held fixed for intervals of several thousand updates, after which it would be updated with the current weights of $Q(s, a; \theta)$. This technique of employing a target network reduces the correlations between the target and the predicted Q-values, again making the training problem less non-stationary. The loss function minimized by DQN then takes the form

$$L(\theta) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}} \left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta) \right)^2, \tag{5}$$

where $\mathcal{D}$ is the experience replay memory and θ^- are the parameters of the target network. Both experience replay and the target network were empirically shown to be important for obtaining the best policies on a number of Atari games, but as discussed earlier, the replay memory can have substantial memory requirements.

4 Asynchronous Lock-Free Reinforcement Learning

We now present multi-threaded asynchronous variants of one-step Sarsa, one-step Q-learning, n-step Q-learning, and advantage actor-critic. The aim in designing these methods was to find RL algorithms that can train deep neural network policies reliably and without large resource requirements. While the underlying RL methods are quite different, with actor-critic being an on-policy policy search method and Q-learning being an off-policy value-based method, we use two main ideas to make all four algorithms practical given our design goal.

First, we use asynchronous actor-learners as proposed in the Gorila framework [Nair et al., 2015], but instead of using separate machines and a parameter server, we use multiple threads on a single machine. Keeping the learners on a single machine removes the communication costs incurred by sending gradients and parameters and enables us to use Hogwild! [Recht et al., 2011] style updates for training the controllers.

Second, we make the observation that multiple actors-learners running in parallel are likely to be exploring different parts of the environment. Moreover, one can explicitly use different exploration policies in each actor-learner to maximize this diversity. By running different exploration policies in different threads, the overall changes being made to the parameters by multiple actor-learners applying online updates in parallel are likely to be less correlated in time than a single agent applying online updates. Hence, we do not use a replay memory and rely on parallel actors employing different exploration policies to perform the stabilizing role undertaken by experience replay in the DQN training algorithm.

In addition to stabilizing learning, using multiple parallel actor-learners has multiple practical benefits. First, we obtain a reduction in training time that is roughly linear in the number of parallel actor-learners. Second, since we no longer rely on experience replay for stabilizing learning we are able to use on-policy reinforcement learning methods such as Sarsa and actor-critic to train neural networks in a stable way. We now describe our variants of one-step Q-learning, one-step Sarsa, n-step Q-learning and advantage actor-critic, discussing design choices specific to each algorithm.

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

// Assume global shared parameter vector θ.
// Assume global shared target parameter vector θ^-.
// Assume global shared counter $T = 0$.
Initialize thread step counter $t \leftarrow 0$
Initialize target network weights $\theta^- \leftarrow \theta$
Initialize network gradients $d\theta \leftarrow 0$
Get initial state s
repeat
  Take action a according to the 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: $d\theta \leftarrow d\theta + \frac{\partial (y - Q(s,a;\theta))^2}{\partial \theta}$
  $s = s'$
  $T \leftarrow T + 1$
  $t \leftarrow t + 1$
  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 $T > T_{max}$

4.1 Asynchronous one-step Q-learning

Pseudocode for our variant of Q-learning, which we call Asynchronous one-step Q-learning, is shown in Algorithm 1. Each thread interacts with its own copy of the environment and at each step computes a gradient of the Q-learning loss. We use a shared and slowly changing target network in computing the Q-learning loss, as was proposed in the DQN training method. We also accumulate gradients over multiple timesteps before they are applied, which is similar to using minibatches. This reduces the chances of multiple actors learners overwriting each other's updates in the Hogwild! setting. Accumulating updates over several steps also provides some ability to trade off computational efficiency for data efficiency.

Finally, we found that giving each thread a different exploration policy helps improve robustness. Adding diversity to exploration in this manner also generally improves performance through better exploration. While there are many possible ways of making the exploration policies differ we experiment with using ϵ-greedy exploration with ϵ that is periodically sampled from some distribution of values by each thread.

While Algorithm 1 gives the pseudocode for the method used in our experiments, we also experimented with a number of variants. For example, we experimented with using thread-

specific target networks instead of using a single shared target network as in Algorithm 1. Another choice is which network is used for selecting actions, the model network with parameters θ or the target network with parameters θ^-. However, we found that these modification led to slightly worse results on a subset of games on the Atari domain.

4.2 Asynchronous one-step Sarsa

The asynchronous one-step Sarsa algorithm is the same as asynchronous one-step Q-learning as given in Algorithm 1 except that it uses a different target value for $Q(s, a)$. The target value used by one-step Sarsa is

$$y = \begin{cases} r & \text{for terminal } s' \\ r + \gamma Q(s', a'; \theta^-) & \text{for non-terminal } s' \end{cases} \tag{6}$$

where a' is the action taken in state s' [Sutton and Barto, 1998]. We again use a target network and updates accumulated over multiple timesteps to stabilize learning.

4.3 Asynchronous n-step Q-learning

Pseudocode for our variant of multi-step Q-learning is shown in Algorithm 2. The algorithm is somewhat unusual because it operates in the forward view by explicitly computing n-step returns, as opposed to the more common backward view used by techniques like eligibility traces [Sutton and Barto, 1998]. We found that using the forward view is easier when training neural networks with momentum-based methods and backpropagation through time. In order to compute a single update, the algorithm first selects actions using its exploration policy for up to t_{max} steps or until a terminal state is reached. This process results in the agent receiving up to t_{max} rewards from the environment since its last update. The algorithm then computes gradients for n-step Q-learning updates for each of the state-action pairs encountered since the last update. Each n-step update uses the longest possible n-step return resulting in a one-step update for the last state, a two-step update for the second last state, and so on for a total of up to t_{max} updates. The accumulated updates are then applied in a single gradient step.

4.4 Asynchronous advantage actor-critic

Our asynchronous variant of actor-critic is presented in Algorithm 3. The algorithm, which we call asynchronous advantage actor-critic (A3C), maintains a policy $\pi(a_t|s_t;\theta)$ and an estimate of the value function $V(s_t;\theta_v)$. Like our variant of n-step Q-learning, our variant of actor-critic also operates in the forward view and uses the same mix of n-step returns to update both the policy and the value-function. The policy and the value function are updated after every t_{max} actions or when a terminal state is reached. The update performed by the algorithm can be seen as $\nabla_{\theta'} \log \pi(a_t|s_t;\theta') A(s_t, a_t; \theta, \theta_v)$ where $A(s_t, a_t; \theta, \theta_v)$ is an estimate of the advantage function given by $\sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta_v) - V(s_t; \theta_v)$, where k varies from state to state and is upper-bounded by t_{max}.

As with the value-based methods we rely on parallel actor-learners and accumulated updates for improving training stability. Note that while the parameters θ of the policy and θ_v of the value function are shown as being separate for generality, we always share some

Algorithm 2 Asynchronous n-step Q-learning - pseudocode for each actor-learner thread.

// 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': $d\theta \leftarrow d\theta + \frac{\partial (R - Q(s_i, a_i; \theta'))^2}{\partial \theta'}$
  end for
  Perform asynchronous update of θ using $d\theta$.
  if $T \mod I_{target} == 0$ **then**
    $\theta^- \leftarrow \theta$
  end if
until $T > T_{max}$

of the parameters in practice. We typically use a convolutional neural network that has one softmax output for the policy $\pi(a_t|s_t;\theta)$ and one linear output for the value function $V(s_t;\theta_v)$, with all non-output layers shared.

We also found that adding the entropy of the policy π to the objective function improved exploration by discouraging premature convergence to suboptimal deterministic policies. This technique was originally proposed by [Williams and Peng, 1991], who found that it was particularly helpful on tasks requiring hierarchical behavior. The gradient of the full objective function including the entropy regularization term with respect to the policy parameters takes the form

$$\nabla_{\theta'} \log \pi(a_t|s_t;\theta')(R_t - V(s_t;\theta_v) + \beta \nabla_{\theta'} H(\pi(s_t;\theta')) \tag{7}$$

where H is the entropy. The hyperparameter β controls the strength of the entropy regularization term.

Algorithm 3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// 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 $T > T_{max}$

4.5 Optimization

We investigated two different optimization algorithms with our asynchronous framework – stochastic gradient descent and RMSProp. Our implementations of these algorithms do not use any locking in order to maximize throughput when using a large number of threads.

Momentum SGD: The implementation of SGD in an asynchronous setting is relatively straightforward and well studied [Recht et al., 2011]. Let θ be the parameter vector that is shared across all threads and let $\Delta\theta_i$ be the accumulated gradients of the loss with respect to parameters θ computed by thread number i. Each thread i independently applies the standard momentum SGD update $m_i = \alpha m_i + (1-\alpha)\Delta\theta_i$ followed by $\theta \leftarrow \theta - \eta m_i$ with learning rate η, momentum α and without any locks. Note that in this setting, each thread maintains its own separate gradient and momentum vector.

RMSProp: While RMSProp [Tieleman and Hinton, 2012] has been widely used in the deep learning literature, it has not been extensively studied in the asynchronous optimization setting. The standard non-centered RMSProp update is given by

$$g = \alpha g + (1-\alpha)\Delta\theta^2 \tag{8}$$

$$\theta \leftarrow \theta - \eta \frac{\Delta\theta}{\sqrt{g + \epsilon}}, \tag{9}$$

where all operations are performed elementwise. In order to apply RMSProp in the asynchronous optimization setting one must decide whether the moving average of elementwise squared gradients g is shared or per-thread. We experimented with two versions of the algorithm. In one version, which we refer to as RMSProp, each thread maintains its own g shown in Equation 8. In the other version, which we call Shared RMSProp, the vector g is shared among threads and is updated asynchronously and without locking. We will show that this way of sharing the statistics greatly improves the stability of the method. Additionally, sharing statistics among threads reduces memory requirements by using one fewer copy of the parameter vector per thread.

5 Experiments

We use four different platforms for assessing the properties of the proposed framework. First, the Arcade Learning Environment [Bellemare et al., 2012] that provides a simulator for Atari 2600 games. This is one of the most commonly used benchmark environments for RL algorithms. We compare against state of the art results on this environment as reported by Van Hasselt et al. [2015], Wang et al. [2015], Schaul et al. [2015], Nair et al. [2015] and Mnih et al. [2015]. The second environment we use is the TORCS car racing simulator [Wymann et al., 2013]. TORCS is a 3D simulator where the graphics are more realistic compared to Atari and, additionally, understanding the physics of the car is an important component. The third environment we use to report results is the MuJoCo [Todorov, 2015] physics simulator for evaluating agents on continuous motor control tasks with contact dynamics. The last domain, which was only used to evaluate our best-performing agent, is a new 3D environment called Labyrinth where the agent must learn to find rewards in randomly generated mazes from a visual input. Finally, we have carried out a detailed stability and scalability analysis of the proposed methods.

5.1 Experimental Setup

The experiments performed on a subset of Atari games (Figures 1, 6, 7 and Table 2) as well as the TORCS experiments (Figure 2) used the following setup. Each experiment used 16 actor-learner threads running on a single machine and no GPUs. All methods performed updates after every 5 actions ($t_{max} = 5$ and $I_{Update} = 5$) and shared RMSProp was used for optimization. The three asynchronous value-based methods used a shared target network that was updated every 40000 frames. The Atari experiments used the same input preprocessing as Mnih et al. [2015] and an action repeat of 4. The agents used the network architecture from Mnih et al. [2013]. The network used a convolutional layer with 16 filters of size 8×8 with stride 4, followed by a convolutional layer with with 32 filters of size 4×4 with stride 2, followed by a fully connected layer with 256 hidden units. All three hidden layers were followed by a rectifier nonlinearity. The value-based methods had a single linear output unit for each action representing the action-value. The model used by actor-critic agents had two set of outputs – a softmax output with one entry per action representing the probability of selecting the action, and a single linear output representing the value function.

The value based methods sampled the exploration rate ϵ from a distribution taking

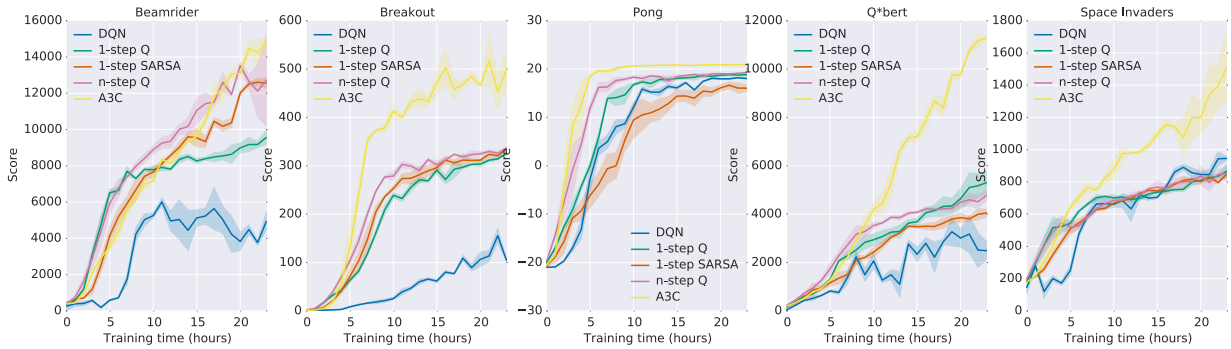

Figure 1: Learning speed comparison for DQN and the new asynchronous algorithms on five Atari 2600 games. DQN was trained on a single Nvidia K40 GPU while the asynchronous methods were trained using 16 CPU cores. The plots are averaged over 5 runs. In the case of DQN the runs were for different seeds with fixed hyperparameters. For asynchronous methods we average over the best 5 models from 50 experiments with learning rates sampled from $LogUniform(10^{-4}, 10^{-2})$ and all other hyperparameters fixed.

three values $\epsilon_1, \epsilon_2, \epsilon_3$ with probabilities $0.4, 0.3, 0.3$. The values of $\epsilon_1, \epsilon_2, \epsilon_3$ were annealed from 1 to $0.1, 0.01, 0.5$ respectively over the first four million frames. Advantage actor-critic used entropy regularization with a weight $\beta = 0.01$ for all Atari and TORCS experiments. We performed a set of 50 experiments for five Atari games and every TORCS level, each using a different random initialization and initial learning rate. The initial learning rate was sampled from a $LogUniform(10^{-4}, 10^{-2})$ distribution and annealed to 0 over the course of training. We analyze the sensitivity of the methods to the choice of learning rate in Section 5.3.2. Note that in comparisons to prior work (Tables 1 and 3) we followed standard evaluation protocol and used fixed hyperparameters.

5.2 Results

5.2.1 Atari 2600 Games

We first present results on a subset of Atari 2600 games to demonstrate the training speed of the new methods. Figure 1 compares the learning speed of the DQN algorithm trained on an Nvidia K40 GPU with the asynchronous methods trained using 16 CPU cores on five Atari 2600 games. The results show that all four asynchronous methods we presented can successfully train neural network controllers on the Atari domain. The asynchronous methods tend to learn faster than DQN, with significantly faster learning on some games, while training on only 16 CPU cores. Additionally, the results suggest that n-step methods do indeed learn faster than one-step methods. Overall, the policy-based advantage actor-critic method significantly outperforms all three value-based methods.

We then evaluated asynchronous advantage actor-critic on 57 Atari games. In order to compare with the state of the art in Atari game playing, we largely followed the training and evaluation protocol of Van Hasselt et al. [2015]. Specifically, we tuned hyperparameters

Method	Training Time	Mean	Median
DQN (from [Nair et al., 2015])	8 days on GPU	121.9%	47.5%
Gorila [Nair et al., 2015]	4 days, 100 machines	215.2%	71.3%
Double DQN [Van Hasselt et al., 2015]	8 days on GPU	332.9%	110.9%
Dueling Double DQN [Wang et al., 2015]	8 days on GPU	343.8%	117.1%
Prioritized DQN [Schaul et al., 2015]	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. Table 3 shows the raw scores for all games.

(learning rate and amount of gradient norm clipping) using a search on six Atari games (Beamrider, Breakout, Pong, Q*bert, Seaquest and Space Invaders) and used the best hyperparameters for all 57 games. We trained both a feedforward agent with the same architecture as [Mnih et al., 2015, Nair et al., 2015, Van Hasselt et al., 2015] as well as a recurrent agent with an additional 256 LSTM [Hochreiter and Schmidhuber, 1997] cells after the final hidden layer. We additionally used the final network weights for evaluation to make the results more comparable to the original results from Bellemare et al. [2012]. We trained our agents for four days using 16 CPU cores, while the other agents were trained for 8 to 10 days on Nvidia K40 GPUs. Table 1 shows the average and median human-normalized scores obtained by our agents trained by asynchronous advantage actor-critic (A3C) as well as the current state-of-the art while Table 3 shows the scores on all games. A3C significantly improves on state-of-the-art the average score over 57 games in half the training time of the other methods while using only 16 CPU cores and no GPU. Furthermore, after just one day of training, A3C matches the average human normalized score of Dueling Double DQN as well as the median human normalized score of DQN. We note that many of the improvements that are presented in Double DQN [Van Hasselt et al., 2015] and Dueling Double DQN [Wang et al., 2015] can be incorporated to 1-step Q and n-step Q methods presented in this work with similar potential improvements.

5.2.2 TORCS Car Racing Simulator

We also compared the four asynchronous methods on the TORCS 3D car racing game [Wymann et al., 2013]. TORCS not only has more realistic graphics than Atari 2600 games, but also requires the agent to learn the dynamics of the car it is controlling. At each step, an agent received only a visual input in the form of an RGB image of the current frame as well as a reward proportional to the agent's velocity along the center of the track at the agent's current position. This reward structure differs considerably from most Atari games, where the rewards are usually very sparse. We used the same neural network architecture as the one used in the Atari experiments specified in Section 5.1. We performed experiments using four different settings – the agent controlling a slow car with and without opponent bots, and the agent controlling a fast car with and without opponent bots. The

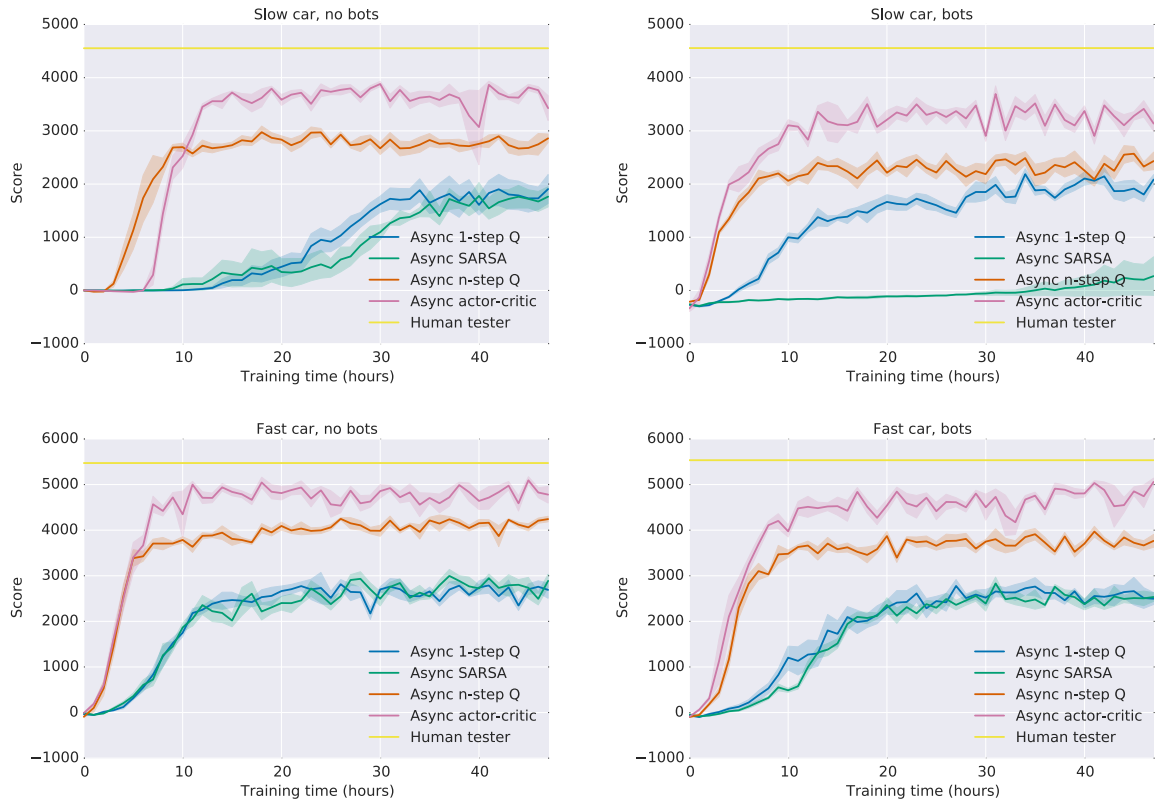

Figure 2: Comparison of algorithms on the TORCS car racing simulator. Four different configurations of car speed and opponent presence or absence are shown. In each plot, all four algorithms (one-step Q, one-step Sarsa, n-step Q and Advantage Actor-Critic) are compared on score vs training time in wall clock hours. Multi-step algorithms achieve better policies much faster than one-step algorithms on all four levels. The curves show averages over the 5 best runs from 50 experiments with learning rates sampled from $LogUniform(10^{-4}, 10^{-2})$ and all other hyperparameters fixed.

results for the different game configurations comparing all four algorithms are shown in Figure 2. Multi-step algorithms learn much faster and reach better policies on all four configurations. Moreover, the best method, Async Advantage Actor-Critic approached its best performance after roughly 12 hours of training. Its performance reached between roughly 75% and 90% of the score obtained by a human tester on all four game configurations. A video showing the learned driving behavior of the best performing agent can be found at `https://youtu.be/0xo1Ldx3L5Q`.

5.2.3 Continuous Action Control Using the MuJoCo Physics Simulator

Finally, we also examined a set of tasks where the action space is continuous. In particular, we look at a set of rigid body physics domains with contact dynamics where the tasks

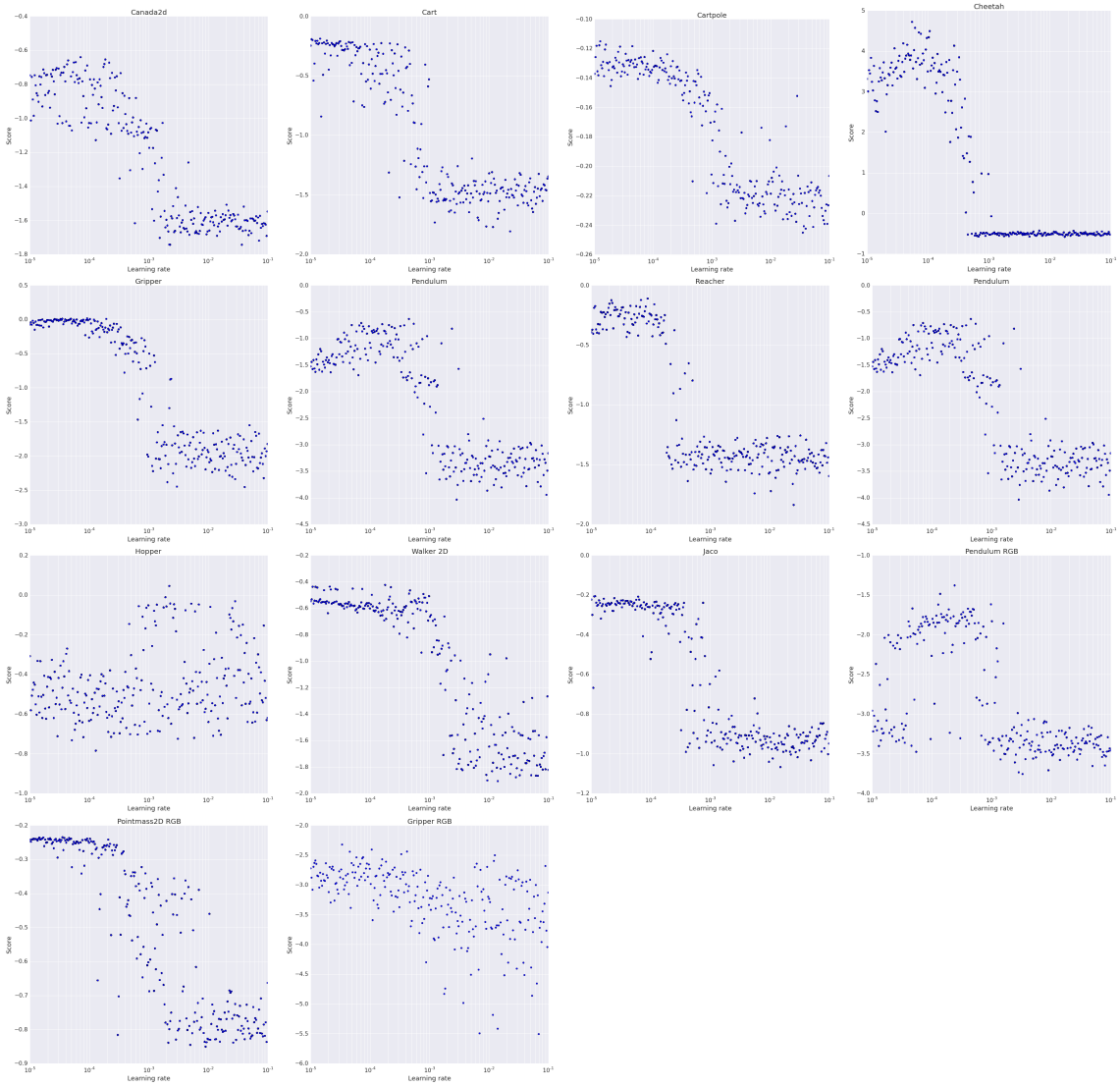

Figure 3: Performance for the Mujoco continuous action domains. Scatter plot of the best score obtained against learning rates sampled from $LogUniform(10^{-5}, 10^{-1})$. For nearly all of the tasks there is a wide range of learning rates that lead to good performance on the task.

include many examples of manipulation and locomotion. These tasks were simulated in the Mujoco physics engine. The action space for the Atari domains is naturally discrete and for TORCS a small discretization of the action space is straightforward and was found to be successful. However, there are many problems for which discretization of the action space is unlikely to be a good strategy. If, for example, a problem requires controlling a system with 10 independently controlled joint torques, then even very coarse discretization of the action space into 5 values for each joint leads to 5^{10} discrete actions. Because of this fact, the DQN algorithm (or any algorithm that relies on a max operator over actions) cannot

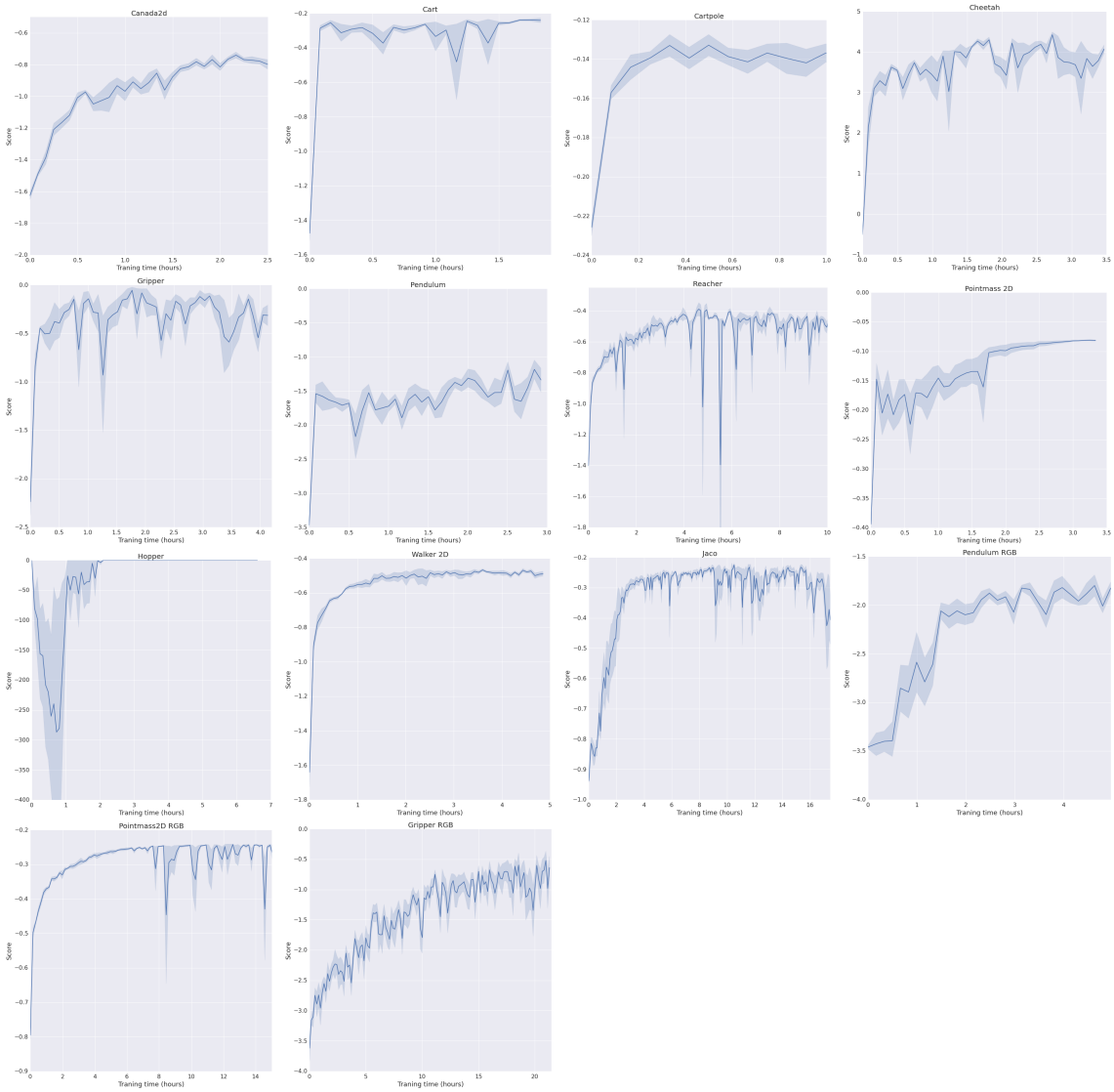

Figure 4: Score per episode vs wall-clock time plots for the Mujoco domains. Each plot shows error bars for the top 5 experiments.

easily be applied to continuous control problems with even moderately sized action spaces.

However, one of the algorithms examined here, the asynchronous advantage actor-critic, is straightforward to apply in continuous action spaces. Since this algorithm does not rely on the max operator over actions, all that is required to apply it to the Mujoco domains is to ensure that the actor network outputs a vector sampled from a continuous distribution in the appropriately sized space. Thus, in the context of the continuous action control problems we examined only the asynchronous advantage actor-critic algorithm. Since most of the design choices for the algorithm were made with discrete control problems in mind, these results serve as a proof-of-concept application and could likely be improved by further adapting the method to continuous control tasks.

To apply the asynchronous advantage actor-critic algorithm to the Mujoco tasks the necessary setup is nearly identical to that used in the discrete action domains, so here we enumerate only the differences required for the continuous action domains. The essential elements for many of the tasks (i.e. the physics models and task objectives) are near identical to the tasks examined in [Lillicrap et al., 2015]. However, the rewards and thus performance are not comparable for most of the tasks due to changes made by the developers of Mujoco which altered the contact model.

For all the domains we attempted to learn the task using the physical state as input. The physical state consisted of the joint positions and velocities as well as the target position if the task required a target. In addition, for three of the tasks (pendulum, pointmass2D, and gripper) we also examined training directly from RGB pixel inputs. In the low dimensional physical state case, the inputs are mapped to a hidden state using one hidden layer with 200 ReLU units. In the cases where we used pixels, the input was passed through two layers of spatial convolutions without any non-linearity or pooling. In either case, the output of the encoder layers were fed to a single layer of 128 LSTM cells. The most important difference in the architecture is in the the output layer of the policy network. Unlike the discrete action domain where the action output is a Softmax, here the two outputs of the policy network are two real number vectors which we treat as the mean vector μ and scalar variance σ^2 of a multidimensional normal distribution with a spherical covariance. To act, the input is passed through the model to the output layer where we sample from the normal distribution determined by μ and σ^2. In practice, μ is modeled by a linear layer and σ^2 by a SoftPlus operation, $\log(1 + \exp(x))$, as the activation computed as a function of the output of a linear layer. In our experiments with continuous control problems the networks for policy network and value network do not share any parameters, though this detail is unlikely to be crucial. Finally, since the episodes were typically at most several hundred time steps long, we did not use any bootstrapping in the policy or value function updates and batched each episode into a single update.

As in the discrete action case, we included an entropy cost which encouraged exploration. In the continuous case the we used a cost on the differential entropy of the normal distribution defined by the output of the actor network, $-\frac{1}{2}(\log(2\pi\sigma^2) + 1)$, we used a constant multiplier of 10^{-4} for this cost across all of the tasks examined. The asynchronous advantage actor-critic algorithm finds solutions for all the domains. Figure 4 shows learning curves against wall-clock time, and demonstrates that most of the domains from states can be solved within a few hours. All of the experiments, including those done from pixel based observations, were run on CPU. Even in the case of solving the domains directly from pixel inputs we found that it was possible to reliably discover solutions within 24 hours. Figure 3 shows scatter plots of the top scores against the sampled learning rates. In most of the domains there is large range of learning rates that consistently achieve good performance on the task.

Some of the successful policies learned by our agent can be seen in the following video `https://youtu.be/Ajjc08-iPx8`.

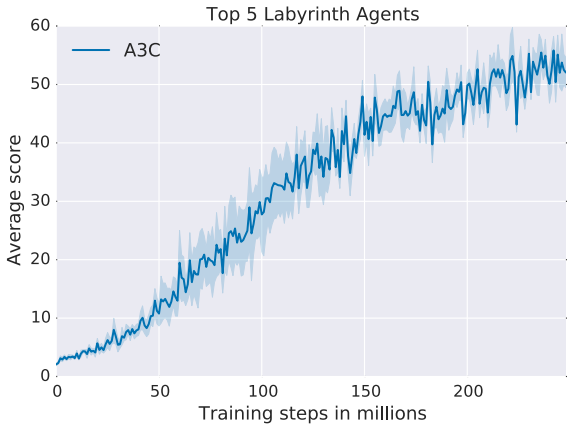

Figure 5: Training curves for the best 5 Labyrinth agents selected from a search over 50 random learning rates and entropy penalties. Training took approximately 3 days.

5.2.4 Labyrinth

We performed an additional set of experiments with A3C on a new 3D environment called Labyrinth. The specific task we considered involved the agent learning to find rewards in randomly generated mazes. At the beginning of each episode the agent was placed in a new randomly generated maze consisting of rooms and corridors. Each maze contained two types of objects that the agent was rewarded for finding – apples and portals. Picking up the agent lead to a reward of 1. Entering a portal lead to a reward of 10 after which the agent was respawned in a new random location in the maze and all previously collected apples were regenerated. An episode terminated after 60 seconds after which a new episode would begin. The aim of the agent is to collect as many points as possible in the time limit and the optimal strategy involves first finding the portal and then repeatedly going back to it after each respawn. This task is much more challenging than the TORCS driving domain because the agent is faced with a new maze in each episode and must learn a general strategy for exploring random mazes.

We trained an A3C LSTM agent on this task using only 84×84 RGB images as input. Figure 5 shows an averaged training curve for the best 5 agents we trained. The final average score of around 50 indicates that the agent learned a reasonable strategy for exploring random 3D maxes using only a visual input. A video showing one of the agents exploring previously unseen mazes is included at `https://youtu.be/nMR5mjCFZCw`.

5.3 Analysis

5.3.1 Scalability and Data Efficiency

We now analyze the effectiveness of our proposed framework by looking at how the training time and data efficiency changes with the number of parallel actor-learners. When using multiple workers in parallel and updating a shared model, one would expect that in an ideal case, for a given task and algorithm, the total number of training steps to achieve a certain

	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).

score would remain the same with varying numbers of workers. Therefore, the advantage would be solely due to the ability of the system to consume more data in the same amount of wall clock time and possibly improved exploration. Table 2 shows the training speed-up achieved by using increasing numbers of parallel actor-learners averaged over seven Atari games. These results show that all four methods achieve substantial speedups from using multiple worker threads, with 16 threads leading to at least an order of magnitude speedup. This confirms that our proposed framework scales well with the number of parallel workers, making efficient use of resources.

Somewhat surprisingly, asynchronous one-step Q-learning and Sarsa algorithms exhibit superlinear speedups that cannot be explained by purely computational gains. These effects are shown more clearly in Figure 6, which shows plots of the average score against the total number of training frames for different numbers of actor-learners and training methods on five Atari games, and Figure 7, which shows plots of the average score against wall-clock time. Figure 6 shows that one-step methods (one-step Q and one-step Sarsa) often require less data to achieve a particular score when using more parallel actor-learners. While a similar effect exists for n-step Q-learning it is less dramatic. When these gains in data efficiency are combined with a sublinear computational speedup, n-step Q-learning achieves a linear speedup from using multiple actor-learners while one-step Q-learning and Sarsa achieve superlinear gains shown in Table 2. The data efficiency of asynchronous advantage actor-critic seems to be largely unaffected by the number of parallel actor-learners. Nevertheless, asynchronous actor-critic still exhibits a substantial speedup, training over 12 times faster using 16 actor-learners.

5.3.2 Robustness and Stability

We first analyze three asynchronous optimization algorithms by inspecting their sensitivity to different learning rates and random network initializations. Stochastic gradient descent is still widely used for training neural networks due to its simplicity and computational efficiency. Although there are many extensions such as ADAGRAD [Duchi et al., 2011],

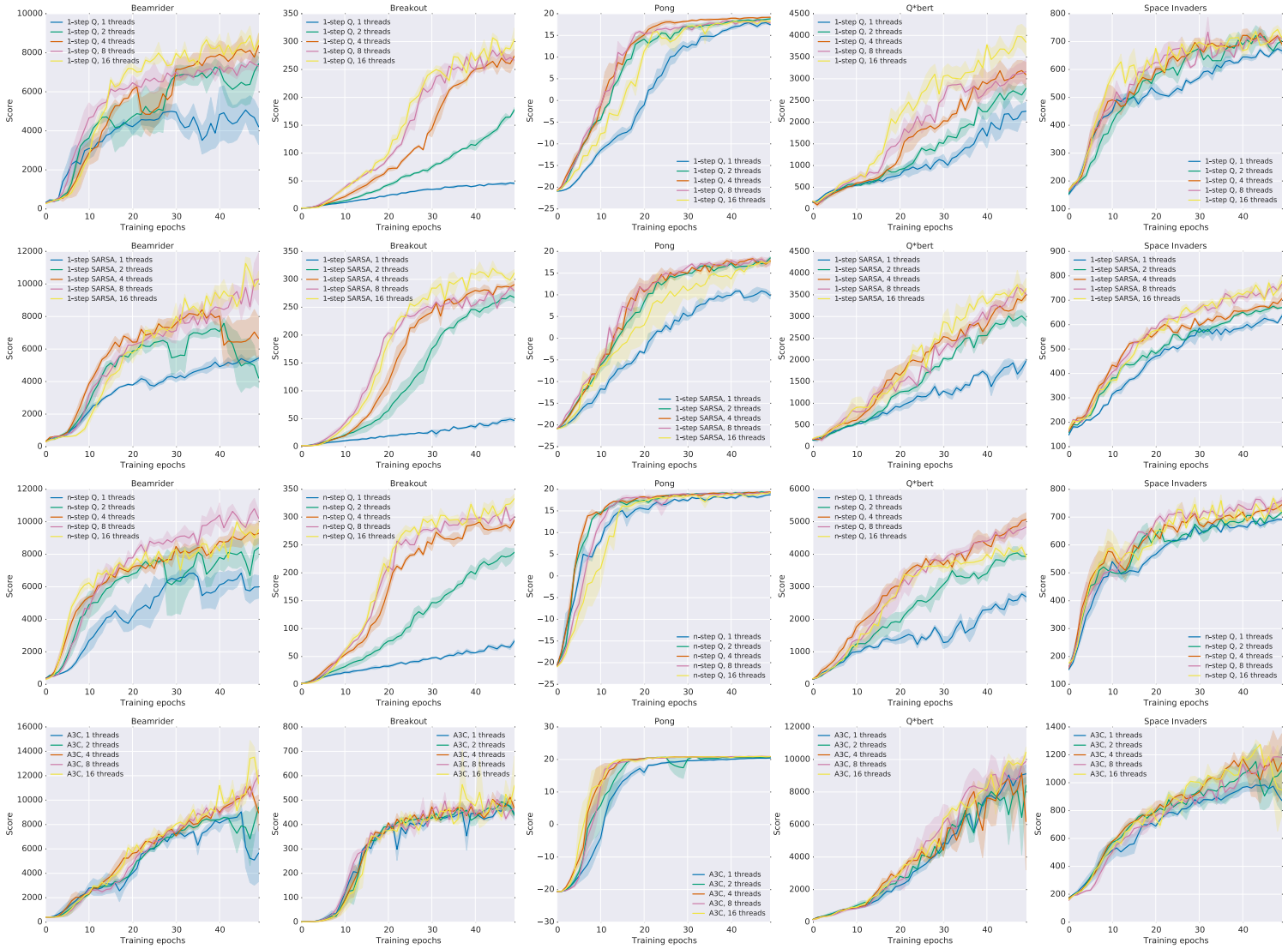

Figure 6: Data efficiency comparison of different numbers of actor-learners for all four asynchronous methods on five Atari games. The x-axis shows the total number of training epochs where an epoch corresponds to four million frames (across all threads). The y-axis shows the average score. Each curve shows the average of the three best performing agents from a search over 50 random learning rates. Single step methods show increased data efficiency with increased numbers of parallel workers.

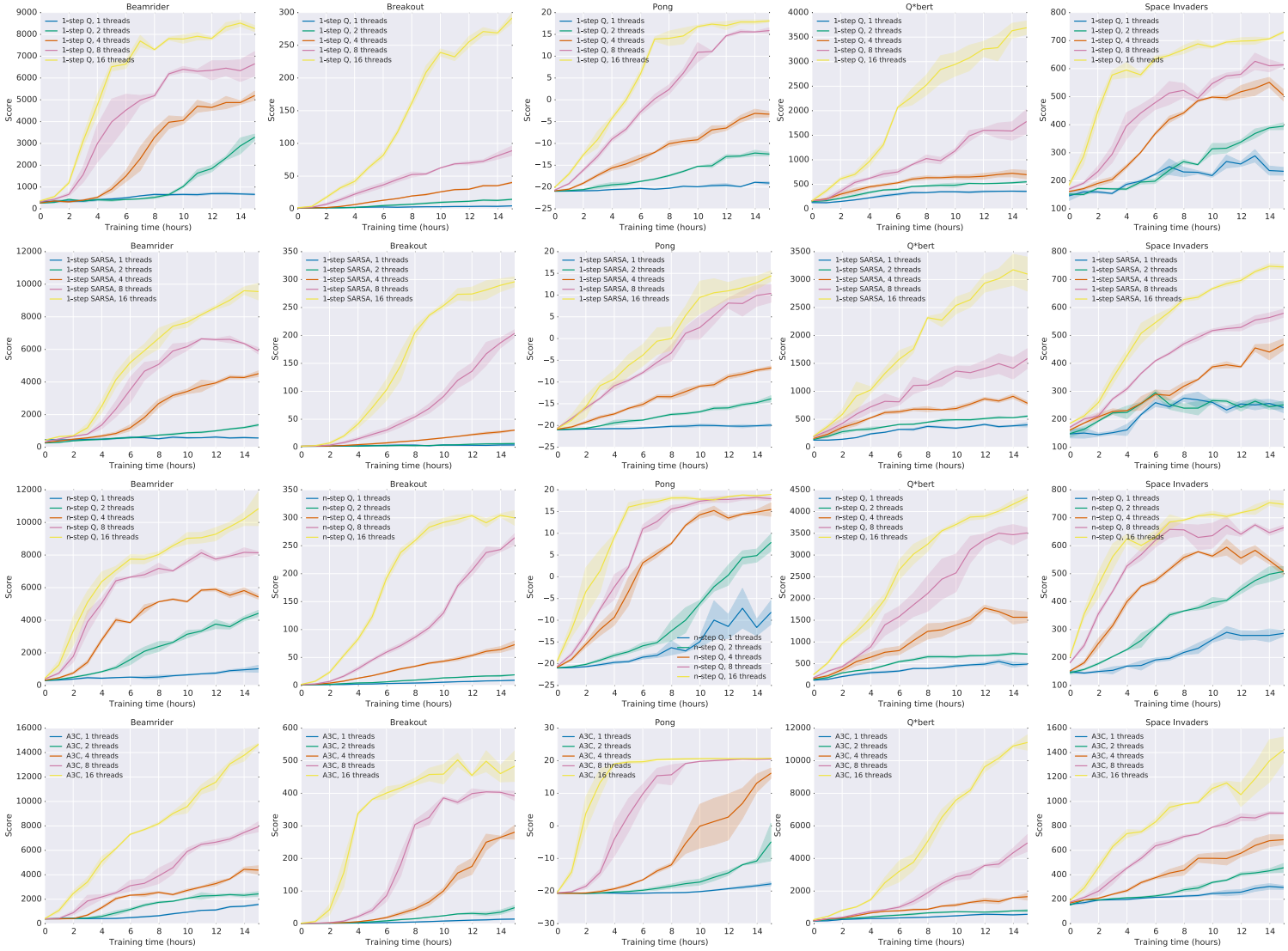

Figure 7: Training speed comparison of different numbers of actor-learners for all four asynchronous methods on five Atari games. The x-axis shows training time in hours while the y-axis shows the average score. Each curve shows the average of the three best performing agents from a search over 50 random learning rates. All asynchronous methods show significant speedups from using greater numbers of parallel actor-learners.

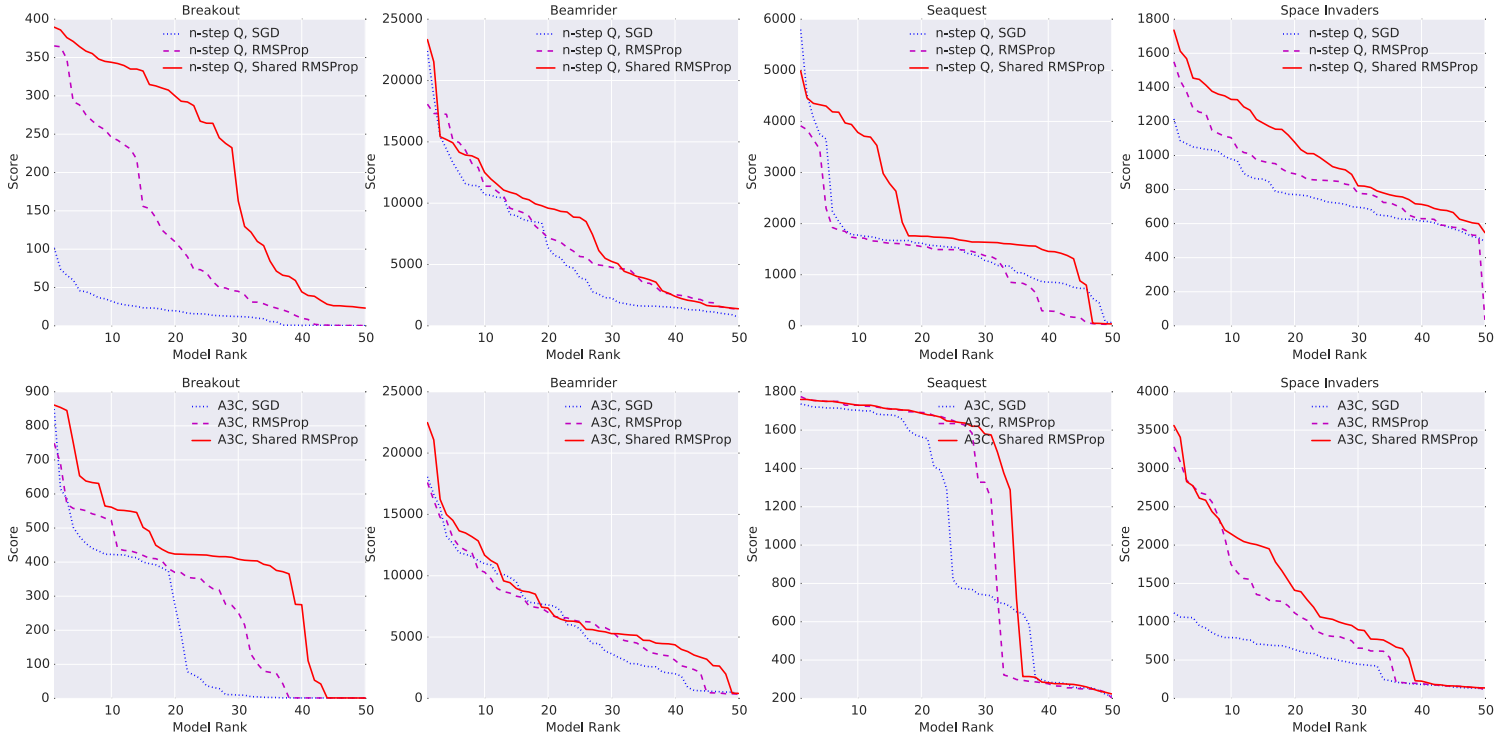

Figure 8: Comparison of three different optimization methods (Momentum SGD, RMSProp, Shared RMSProp) tested using two different algorithms (Async n-step Q and Async Advantage Actor-Critic) on four different Atari games (Breakout, Beamrider, Seaquest and Space Invaders). Each curve shows the final scores for 50 experiments sorted in descending order that covers a search over 50 random initializations and learning rates. The top row shows results using Async n-step Q algorithm and bottom row shows results with Async Advantage Actor-Critic. Each individual graph shows results for one of the four games and three different optimization methods. Shared RMSProp tends to be more robust to different learning rates and random initializations than Momentum SGD and RMSProp without sharing.

ADADELTA [Zeiler, 2012], RMSProp [Tieleman and Hinton, 2012] and ADAM [Kingma and Ba, 2014], there is no consensus as to which method is the best. In Figure 8 we compare three different asynchronous optimization algorithms (Momentum SGD, RMSProp, Shared RMSProp) combined with two different reinforcement learning methods (Async n-step Q and Async Advantage Actor-Critic) on four different tasks (Breakout, Beamrider, Seaquest and Space Invaders). Each curve shows the scores for 50 experiments that correspond to 50 different random learning rates and initializations. The x-axis shows the rank of the model after sorting in descending order by final average score and the y-axis shows the final average score achieved by the corresponding model. In this representation, the algorithm that performs better would achieve higher maximum rewards on the y-axis and the algorithm that is most robust would have its slope closest to horizontal, thus maximizing the area under the curve. RMSProp with shared statistics tends to be more robust than

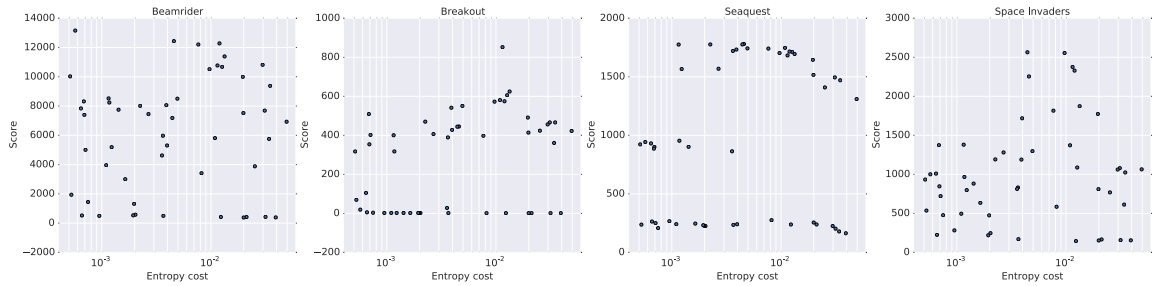

Figure 9: Scatter plots of final scores achieved by Advantage Actor-Critic on four games (Breakout, Beamrider, Seaquest, Space Invaders) for 50 different entropy regularization penalty coefficients, learning rates, and random initializations. On some games using entropy regularization improves performance.

RMSProp with per-thread statistics, which is in turn more robust than Momentum SGD.

Next, we look at the stability and robustness of the asynchronous algorithms. We trained models on five games (Breakout, Beamrider, Pong, Q*bert, Space Invaders) using four different algorithms (one-step Q, one-step Sarsa, n-step Q and Advantage Actor-Critic) using 50 different learning rates and random initializations. Scatter plots of scores are shown for all algorithms and tasks in Figure 10. There is usually a range of learning rates for each method and game combination that leads to a high score, indicating that all methods are quite robust to the choice of learning rate. The fact that there are virtually no points with scores of 0 in regions with good learning rates indicates that the methods are stable and do not collapse or diverge once they are learning. Similarly, in Figure 9 we show scatter plots of scores obtained by training Advantage Actor-Critic for 50 combinations of random initialization, learning rate and entropy cost on four games. These results show that using entropy regularization with advantage actor-critic does lead to better scores on some games.

6 Conclusions and Discussion

We have presented asynchronous versions of four standard reinforcement learning algorithms and showed that they are able to train neural network controllers on a variety of domains in a stable manner. Our results show that in our proposed framework stable training of neural networks through reinforcement learning is possible with both value-based and policy-based methods, off-policy as well as on-policy methods, and in discrete as well as continuous domains. When trained on the Atari domain using 16 CPU cores, the proposed asynchronous algorithms train faster than DQN trained on an Nvidia K40 GPU, with A3C surpassing the current state-of-the-art in half the training time.

One of our main findings is that using parallel actor-learners to update a shared model had a stabilizing effect on the learning process of the three value-based methods we considered. While this shows that stable online Q-learning is possible without experience replay, which was used for this purpose in DQN, it does not mean that experience replay is not

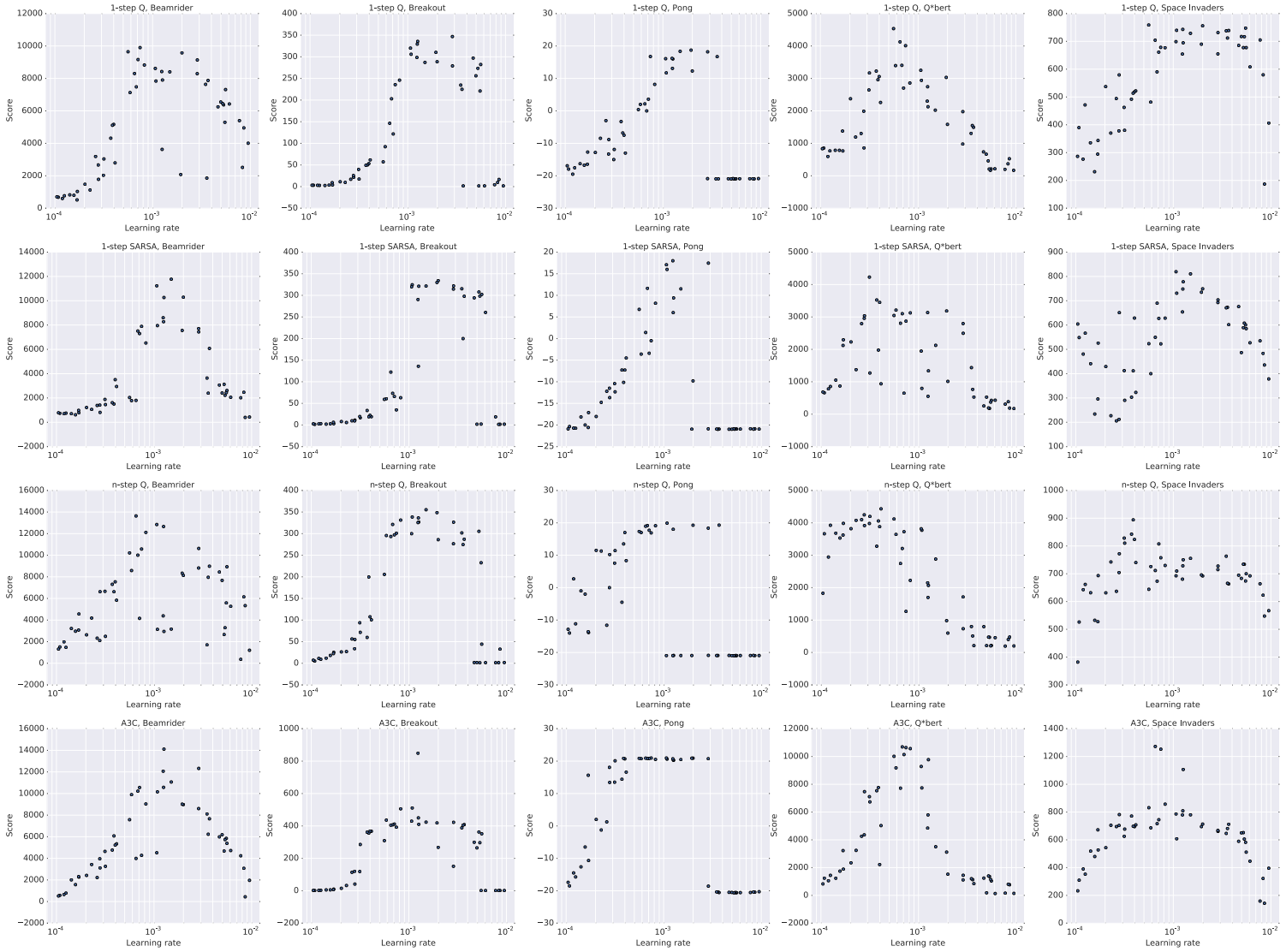

Figure 10: Scatter plots of scores obtained by four different algorithms (one-step Q, one-step Sarsa, n-step Q and Advantage Actor-Critic) on five games (Beamrider, Breakout, Pong, Q*bert, Space Invaders) for 50 different learning rates and random initializations. All algorithms exhibit some level of robustness to the choice of learning rate.

useful. Incorporating experience replay into the asynchronous reinforcement learning framework could substantially improve the data efficiency of these methods by reusing old data. This could in turn lead to much faster training times in domains like TORCS where interacting with the environment is more expensive than updating the model for the architecture we used.

Combining other existing reinforcement learning methods or recent advances in deep reinforcement learning with our asynchronous framework presents many possibilities for immediate improvements to the methods we presented. While our n-step methods operate

in the *forward view* [Sutton and Barto, 1998] by using corrected n-step returns directly as targets, it has been more common to use the *backward view* to implicitly combine different returns through eligibility traces [Watkins, 1989, Sutton and Barto, 1998, Peng and Williams, 1996]. The asynchronous advantage actor-critic method could be potentially improved by using other ways of estimating the advantage function, such as generalized advantage estimation of Schulman et al. [2015b]. All of the value-based methods we investigated could benefit from different ways of reducing over-estimation bias of Q-values [Van Hasselt et al., 2015, Bellemare et al., 2016]. Yet another, more speculative, direction is to try and combine the recent work on true online temporal difference methods [van Seijen et al., 2015] with nonlinear function approximation.

In addition to these algorithmic improvements, a number of complementary improvements to the neural network architecture are possible. The dueling architecture of Wang et al. [2015] has been shown to produce more accurate estimates of Q-values by including separate streams for the state value and advantage in the network. The spatial softmax proposed by Levine et al. [2015] could improve both value-based and policy-based methods by making it easier for the network to represent feature coordinates.

Acknowledgments

We thank Thomas Degris, Remi Munos, Marc Lanctot, Sasha Vezhnevets and Joseph Modayil for many helpful discussions, suggestions and comments on the paper. We also thank the DeepMind evaluation team for setting up the environments used to evaluate the agents in the paper.

Game	DQN	Gorila	Double	Dueling	Prioritized	A3C FF*	A3C FF	A3C LSTM
Alien	570.2	813.5	1033.4	**1486.5**	900.5	182.1	518.4	945.3
Amidar	133.4	189.2	169.1	172.7	218.4	**283.9**	263.9	173.0
Assault	3332.3	1195.8	6060.8	3994.8	7748.5	3746.1	5474.9	**14497.9**
Asterix	124.5	3324.7	16837.0	15840.0	**31907.5**	6723.0	22140.5	17244.5
Asteroids	697.1	933.6	1193.2	2035.4	1654.0	3009.4	4474.5	**5093.1**
Atlantis	76108.0	629166.5	319688.0	445360.0	593642.0	772392.0	**911091.0**	875822.0
Bank Heist	176.3	399.4	886.0	**1129.3**	816.8	946.0	970.1	932.8
Battle Zone	17560.0	19938.0	24740.0	**31320.0**	29100.0	11340.0	12950.0	20760.0
Beam Rider	8672.4	3822.1	17417.2	14591.3	**26172.7**	13235.9	22707.9	24622.2
Berzerk			1011.1	910.6	1165.6	**1433.4**	817.9	862.2
Bowling	41.2	54.0	**69.6**	65.7	65.8	36.2	35.1	41.8
Boxing	25.8	74.2	73.5	**77.3**	68.6	33.7	59.8	37.3
Breakout	303.9	313.0	368.9	411.6	371.6	551.6	681.9	**766.8**
Centipede	3773.1	**6296.9**	3853.5	4881.0	3421.9	3306.5	3755.8	1997.0
Chopper Comman	3046.0	3191.8	3495.0	3784.0	6604.0	4669.0	7021.0	**10150.0**
Crazy Climber	50992.0	65451.0	113782.0	124566.0	131086.0	101624.0	112646.0	**138518.0**
Defender			27510.0	33996.0	21093.5	36242.5	56533.0	**233021.5**
Demon Attack	12835.2	14880.1	69803.4	56322.8	73185.8	84997.5	113308.4	**115201.9**
Double Dunk	-21.6	-11.3	-0.3	-0.8	**2.7**	0.1	-0.1	0.1
Enduro	475.6	71.0	1216.6	**2077.4**	1884.4	-82.2	-82.5	-82.5
Fishing Derby	-2.3	4.6	3.2	-4.1	9.2	13.6	18.8	**22.6**
Freeway	25.8	10.2	**28.8**	0.2	27.9	0.1	0.1	0.1
Frostbite	157.4	426.6	1448.1	2332.4	**2930.2**	180.1	190.5	197.6
Gopher	2731.8	4373.0	15253.0	20051.4	**57783.8**	8442.8	10022.8	17106.8
Gravitar	216.5	**538.4**	200.5	297.0	218.0	269.5	303.5	320.0
H.E.R.O.	12952.5	8963.4	14892.5	15207.9	20506.4	28765.8	**32464.1**	28889.5
Ice Hockey	-3.8	-1.7	-2.5	-1.3	**-1.0**	-4.7	-2.8	-1.7
James Bond	348.5	444.0	573.0	835.5	**3511.5**	351.5	541.0	613.0
Kangaroo	2696.0	1431.0	**11204.0**	10334.0	10241.0	106.0	94.0	125.0
Krull	3864.0	6363.1	6796.1	8051.6	7406.5	**8066.6**	5560.0	5911.4
Kung-Fu Master	11875.0	20620.0	30207.0	24288.0	31244.0	3046.0	28819.0	**40835.0**
Montezuma's Revenge	50.0	**84.0**	42.0	22.0	13.0	53.0	67.0	41.0
Ms. Pacman	763.5	1263.0	1241.3	**2250.6**	1824.6	594.4	653.7	850.7
Name This Game	5439.9	9238.5	8960.3	11185.1	11836.1	5614.0	10476.1	**12093.7**
Phoenix			12366.5	20410.5	27430.1	28181.8	52894.1	**74786.7**
Pit Fall			-186.7	-46.9	**-14.8**	-123.0	-78.5	-135.7
Pong	16.2	16.7	**19.1**	18.8	18.9	11.4	5.6	10.7
Private Eye	298.2	**2598.6**	-575.5	292.6	179.0	194.4	206.9	421.1
Q*Bert	4589.8	7089.8	11020.8	14175.8	11277.0	13752.3	15148.8	**21307.5**
River Raid	4065.3	5310.3	10838.4	16569.4	**18184.4**	10001.2	12201.8	6591.9
Road Runner	9264.0	43079.8	43156.0	58549.0	56990.0	31769.0	34216.0	**73949.0**
Robotank	58.5	61.8	59.1	**62.0**	55.4	2.3	32.8	2.6
Seaquest	2793.9	10145.9	14498.0	37361.6	**39096.7**	2300.2	2355.4	1326.1
Skiing			-11490.4	-11928.0	**-10852.8**	-13700.0	-10911.1	-14863.8
Solaris			810.0	1768.4	**2238.2**	1884.8	1956.0	1936.4
Space Invaders	1449.7	1183.3	2628.7	5993.1	9063.0	2214.7	15730.5	**23846.0**
Star Gunner	34081.0	14919.2	58365.0	90804.0	51959.0	64393.0	138218.0	**164766.0**
Surround			1.9	**4.0**	-0.9	-9.6	-9.7	-8.3
Tennis	-2.3	-0.7	-7.8	**4.4**	-2.0	-10.2	-6.3	-6.4
Time Pilot	5640.0	8267.8	6608.0	6601.0	7448.0	5825.0	12679.0	**27202.0**
Tutankham	32.4	118.5	92.2	48.0	33.6	26.1	**156.3**	144.2
Up and Down	3311.3	8747.7	19086.9	24759.2	29443.7	54525.4	74705.7	**105728.7**
Venture	54.0	**523.4**	21.0	200.0	244.0	19.0	23.0	25.0
Video Pinball	20228.1	112093.4	367823.7	110976.2	374886.9	185852.6	331628.1	**470310.5**
Wizard of Wor	246.0	10431.0	6201.0	7054.0	7451.0	5278.0	17244.0	**18082.0**
Yars Revenge			6270.6	**25976.5**	5965.1	7270.8	7157.5	5615.5
Zaxxon	831.0	6159.4	8593.0	10164.0	9501.0	2659.0	**24622.0**	23519.0

Table 3: Raw scores for the human start condition (30 minutes emulator time). DQN scores taken from Nair et al. [2015]. Double DQN scores taken from Van Hasselt et al. [2015], Dueling scores from Wang et al. [2015] and Prioritized scores taken from Schaul et al. [2015]

References

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 2012.

Marc G. Bellemare, Georg Ostrovski, Arthur Guez, Philip S. Thomas, and Rémi Munos. Increasing the action gap: New operators for reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2016.

Dimitri P Bertsekas. Distributed dynamic programming. *Automatic Control, IEEE Transactions on*, 27(3):610–616, 1982.

Kevin Chavez, Hao Yi Ong, and Augustus Hong. Distributed deep q-learning. Technical report, Stanford University, June 2015.

Thomas Degris, Patrick M Pilarski, and Richard S Sutton. Model-free reinforcement learning with continuous action in practice. In *American Control Conference (ACC), 2012*, pages 2177–2182. IEEE, 2012.

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

Matthew Grounds and Daniel Kudenko. 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*, pages 60–74. Springer-Verlag, 2008.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Tommi Jaakkola, Michael I Jordan, and Satinder P Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural computation*, 6(6):1185–1201, 1994.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Jan Koutník, Jürgen Schmidhuber, and Faustino Gomez. Evolving deep unsupervised convolutional networks for vision-based reinforcement learning. In *Proceedings of the 2014 conference on Genetic and evolutionary computation*, pages 541–548. ACM, 2014.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *arXiv preprint arXiv:1504.00702*, 2015.

Yuxi Li and Dale Schuurmans. 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*, pages 309–320, 2011.

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

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

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

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

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, and David Silver. Massively parallel methods for deep reinforcement learning. In *ICML Deep Learning Workshop*. 2015.

Jing Peng and Ronald J Williams. Incremental multi-step q-learning. *Machine Learning*, 22(1-3):283–290, 1996.

Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *Machine Learning: ECML 2005*, pages 317–328. Springer Berlin Heidelberg, 2005.

Gavin A Rummery and Mahesan Niranjan. On-line q-learning using connectionist systems. 1994.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

John Schulman, Sergey Levine, Philipp Moritz, Michael I Jordan, and Pieter Abbeel. Trust region policy optimization. In *International Conference on Machine Learning (ICML)*, 2015a.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015b.

Satinder Singh, Tommi Jaakkola, Michael L Littman, and Csaba Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 38 (3):287–308, 2000.

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

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4, 2012.

E Todorov. *MuJoCo: Modeling, Simulation and Visualization of Multi-Joint Dynamics with Contact (ed 1.0)*. Roboti Publishing, 2015.

Marco Tomassini. Parallel and distributed evolutionary algorithms: A review. Technical report, 1999.

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

John N Tsitsiklis. Asynchronous stochastic approximation and q-learning. *Machine Learning*, 16(3):185–202, 1994.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *arXiv preprint arXiv:1509.06461*, 2015.

H. van Seijen, A. Rupam Mahmood, P. M. Pilarski, M. C. Machado, and R. S. Sutton. True Online Temporal-Difference Learning. *ArXiv e-prints*, December 2015.

Z. Wang, N. de Freitas, and M. Lanctot. Dueling Network Architectures for Deep Reinforcement Learning. *ArXiv e-prints*, November 2015.

Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge England, 1989.

R.J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, 1992.

Ronald J Williams and Jing Peng. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3(3):241–268, 1991.

B. Wymann, E. Espi, C. Guionneau, C. Dimitrakakis, R. Coulom, and A. Sumner. Torcs: The open racing car simulator, v1.3.5, 2013.

Matthew D Zeiler. Adadelta: An adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.