

Supplementary Material for "Asynchronous Methods for Deep Reinforcement Learning"

May 25, 2016

1 Optimization Details

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

$$g = \alpha g + (1 - \alpha)\Delta\theta^2 \tag{S1}$$

$$\theta \leftarrow \theta - \eta \frac{\Delta\theta}{\sqrt{g + \epsilon}}, \tag{S2}$$

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 S1. In the other version, which we call Shared RMSProp, the vector g is shared among threads and is updated asynchronously and without locking. Sharing statistics among threads also reduces memory requirements by using one fewer copy of the parameter vector per thread.

We compared these three asynchronous optimization algorithms in terms of their sensitivity to different learning rates and random network initializations. Figure S1 shows a comparison of the methods for two different reinforcement learning methods (Async n -step Q and Async Advantage Actor-Critic) on four different games (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 RMSProp with per-thread statistics, which is in turn more robust than Momentum SGD.

2 Experimental Setup

The experiments performed on a subset of Atari games (Figures 1, 3, 4 and Table 2) as well as the TORCS experiments (Figure S2) 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 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. All experiments used a discount of $\gamma = 0.99$ and an RMSProp decay factor of $\alpha = 0.99$.

The value based methods sampled the exploration rate ϵ from a distribution taking 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. Note that in comparisons to prior work (Tables 1 and S1) we followed standard evaluation protocol and used fixed hyperparameters.

3 Continuous Action Control Using the MuJoCo Physics Simulator

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 continu-

ous 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 S4 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 S3 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.

Algorithm S1 Asynchronous n-step Q-learning - pseudocode for each actor-learner thread.

```

// Assume global shared parameter vector  $\theta$ .
// Assume global shared target parameter vector  $\theta^-$ .
// 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  $\epsilon$ -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  $i \in \{t - 1, \dots, t_{start}\}$  do
     $R \leftarrow r_i + \gamma R$ 
    Accumulate gradients wrt  $\theta'$ :  $d\theta \leftarrow d\theta + \frac{\partial(R - Q(s_i, a_i; \theta'))^2}{\partial \theta'}$ 
  end for
  Perform asynchronous update of  $\theta$  using  $d\theta$ .
  if  $T \bmod I_{target} == 0$  then
     $\theta^- \leftarrow \theta$ 
  end if
until  $T > T_{max}$ 

```

Algorithm S2 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 $i \in \{t - 1, \dots, t_{start}\}$ **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 : $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i; \theta'_v))^2 / \partial \theta'_v$

end for

Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.

until $T > T_{max}$

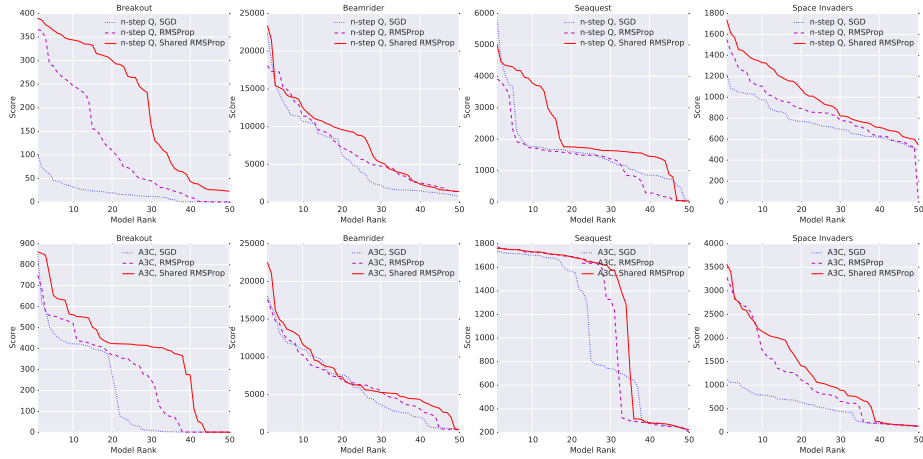


Figure S1: Comparison of three different optimization methods (Momentum SGD, RMSProp, Shared RMSProp) tested using two different algorithms (Async n -step Q and Async Advantage Actor-Critic) on four different Atari games (Breakout, Beamrider, Seaquest and Space Invaders). Each curve shows the final scores 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.

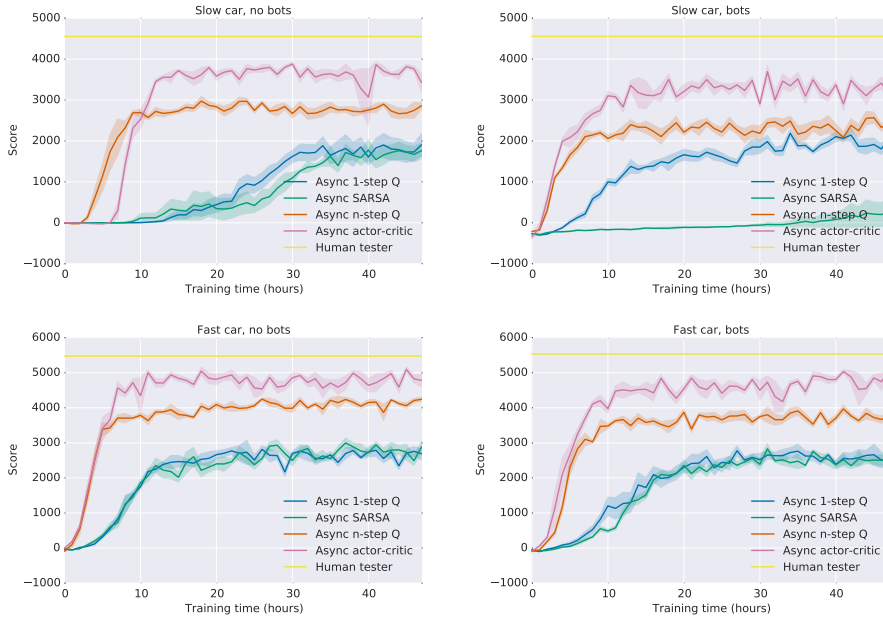


Figure S2: Comparison of algorithms on the TORCS car racing simulator. Four different configurations of car speed and opponent presence or absence are shown. In each plot, all four algorithms (one-step Q, one-step Sarsa, n -step Q and Advantage Actor-Critic) are compared on score vs training time in 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 $\text{LogUniform}(10^{-4}, 10^{-2})$ and all other hyperparameters fixed.

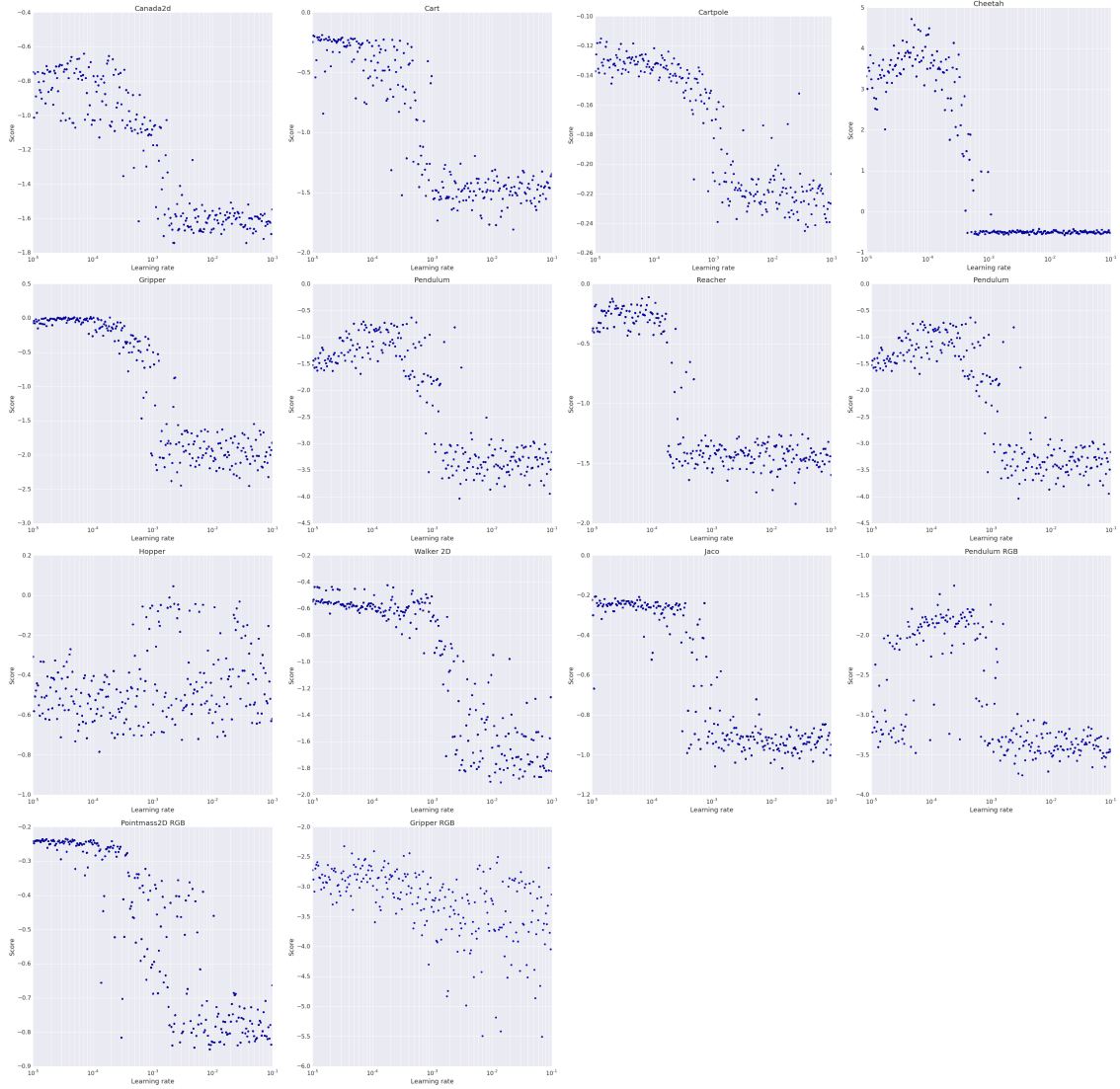


Figure S3: Performance for the Mujoco continuous action domains. Scatter plot of the best score obtained against learning rates sampled from $LogUniform(10^{-5}, 10^{-1})$. For nearly all of the tasks there is a wide range of learning rates that lead to good performance on the task.

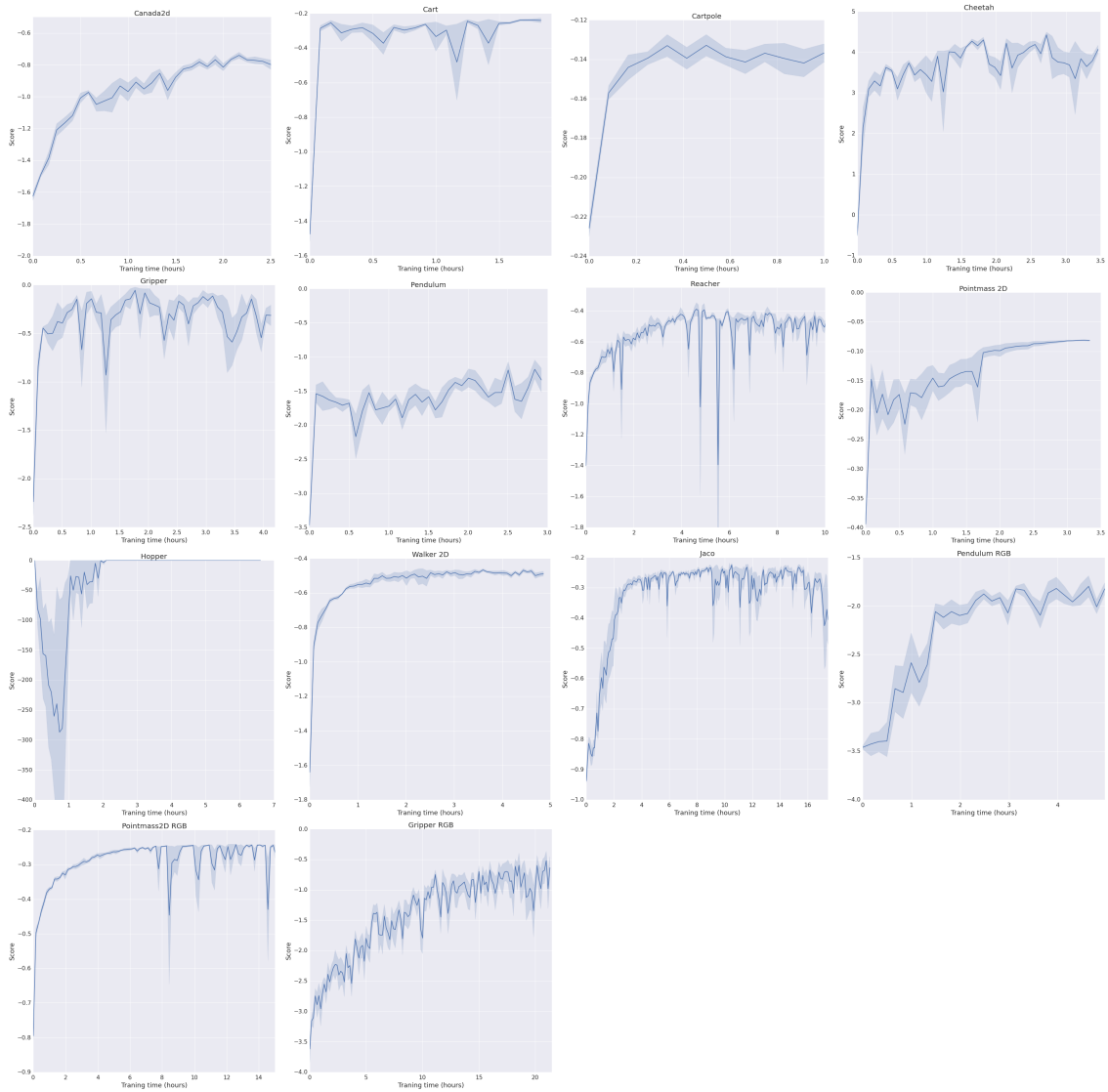


Figure S4: Score per episode vs wall-clock time plots for the Mujoco domains. Each plot shows error bars for the top 5 experiments.

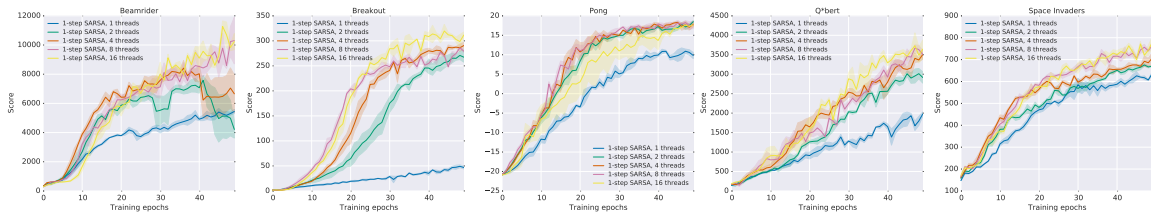


Figure S5: Data efficiency comparison of different numbers of actor-learners one-step Sarsa on five Atari games. The x-axis shows the total number of training epochs where an epoch corresponds to four million frames (across all threads). The y-axis shows the average score. Each curve shows the average of the three best performing agents from a search over 50 random learning rates. Sarsa shows increased data efficiency with increased numbers of parallel workers.

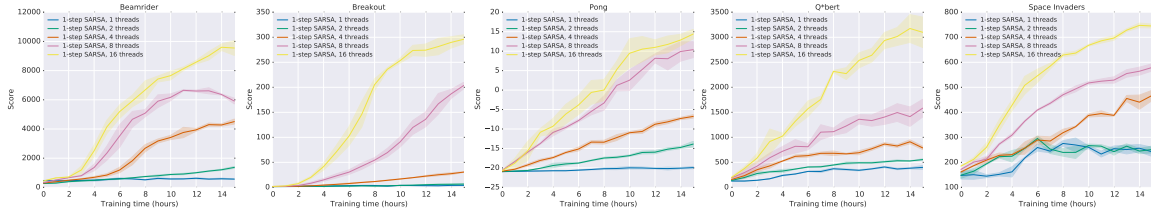


Figure S6: Training speed comparison of different numbers of actor-learners for all one-step Sarsa on five Atari games. The x-axis shows training time in hours while the y-axis shows the average score. Each curve shows the average of the three best performing agents from a search over 50 random learning rates. Sarsa shows significant speedups from using greater numbers of parallel actor-learners.

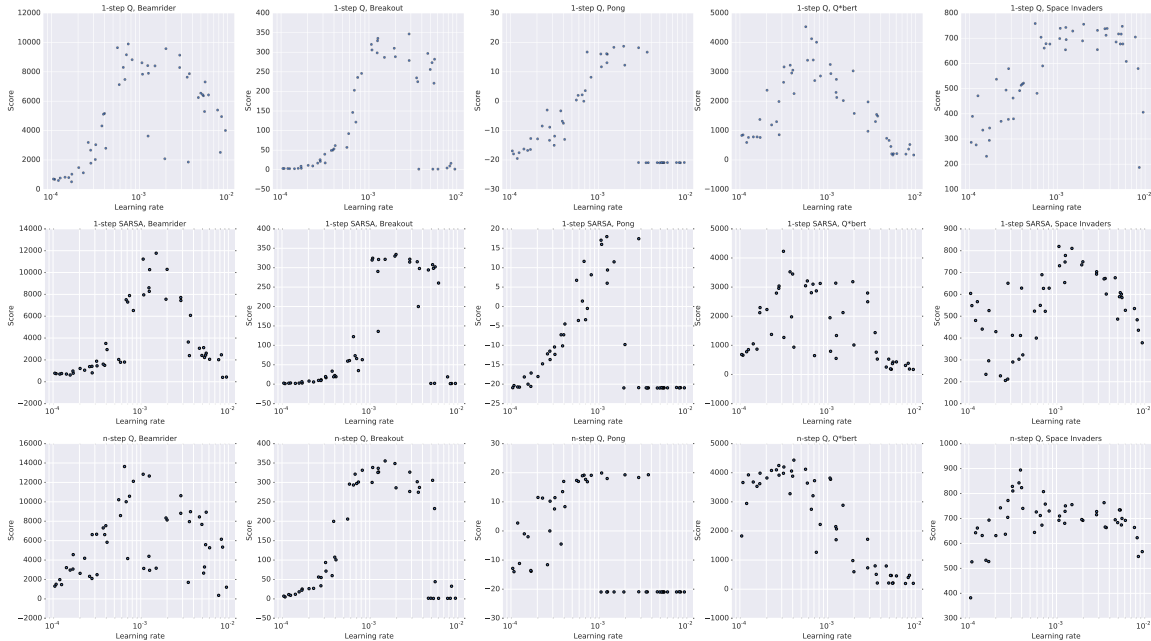


Figure S7: Scatter plots of scores obtained by one-step Q, one-step Sarsa, and n -step Q on five games (Beamrider, Breakout, Pong, Q*bert, Space Invaders) for 50 different learning rates and random initializations. All algorithms exhibit some level of robustness to the choice of learning rate.

Game	DQN	Gorila	Double	Dueling	Prioritized	A3C FF, 1 day	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 S1: 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

- 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.
- 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, Dhharshan 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.
- 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.
- Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.
- 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.
- Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *arXiv preprint arXiv:1509.06461*, 2015.
- Z. Wang, N. de Freitas, and M. Lanctot. Dueling Network Architectures for Deep Reinforcement Learning. *ArXiv e-prints*, November 2015.