Effectiveness of Combining Deep Reinforcement Learning Algorithms

Kyu-Young Kim

Abstract

Deep learning models have been successfully employed in many reinforcement learning (RL) algorithms as effective function approximators. Deep Q-Network (DQN) is a notable example of such an application of deep learning to RL, and since DQN many improvements have been independently made to address different limitations of the algorithm. One natural extension would be to combine these improvements to potentially achieve performance better than that from applying them individually. This paper empirically examines the effectiveness of combining some of these deep RL algorithms in terms of data efficiency and performance in both fully-observable and partially-observable settings. Our experiments show that some of the extensions can indeed be combined to achieve better convergence and are often even necessary to achieve good performance in certain domains.

1. Introduction

Deep learning has recently been successfully applied to reinforcement learning (RL) to advance the state-of-the-art in a wide range of complex sequential decision making tasks. With the ability to automatically learn high-level features from raw input, deep learning models allowed to scale RL algorithms even to highly complex environments.

Deep Q-Network (DQN), for example, combined the classic Q-learning algorithm with a function approximator based on convolutional neural networks (CNN) to train agents that can learn human-level control policies from raw pixel input. DQN agents were able to achieve human-level performance on many of the Atari 2600 games (Mnih et al., 2013). Following the DQN's success, many enhancements have been proposed to improve different aspects of the algorithm such as data efficiency and training stability.

Conventional Q-learning uses the maximum action value to estimate the maximum expected value during updates. This can lead to large overestimation of action values, resulting in poor performance (Hasselt, 2010). Double Q-learning addresses this inherent overestimation bias by decomposing the max operation into selection and evaluation of the action (Hasselt et al., 2016). Using rewards accumulated over multiple steps instead of a single step as the bootstrap target can give better bias-variance tradeoff and lead to faster convergence (Sutton & Barto, 2018). Actor-critic architectures seek to reap the benefits of policy gradient method while reducing variance to stabilize training using a baseline. Various algorithms have been proposed that use deep learning models to represent the policy and baseline (e.g., Q actor-critic). Finally, recurrent neural networks (RNN) which are well suited for learning temporal dependencies have been used with deep RL models in order to handle partial-observability of state information. Given that the aforementioned techniques are used to address different shortcomings of the original DQN, it is natural to try combining them to train an integrated agent for potential further improvement in performance.

In this paper, we study different combinations of these ingredients and their effectiveness in terms of data efficiency and performance. This work resembles and is motivated by (Hessel et al., 2017), but we experiment with both fully-observable and partially-observable environments to better understand the characteristics of the algorithms in more diverse setups.

2. Background

2.1. Reinforcement Learning

Reinforcement learning studies the problem of training an *agent* that learns to act within an *environment* in order to maximize the expected return. In contrast to standard supervised learning, no direct supervision as to what the optimal action is is given to the agent. The challenge is for the agent to learn a good policy based on a series of reward signals received from its interaction with the environment.

Given this setup, there are several common issues in RL that are worth noting. First, the agent needs to handle temporal dependencies among a potentially long sequence of actions so to learn which actions led to a certain outcome. This is especially important in an environment with sparse rewards, where the agent needs to experience a long series of transitions before receiving a reward signal. This problem is often referred to as credit assignment problem. Second,

the agent needs to employ a suitable exploration strategy in order to learn a good policy. In a standard supervised learning setup, the set of training examples is often fixed, and the model is trained to optimize a certain loss on these examples. In RL, it is common for the agent to be trained in an online fashion, where the set of observations used to train the agent is directly dependent on the agent's policy that is being optimized. This inherent feedback loop makes it extremely challenging to train a good agent in certain RL domains. Third, the agent needs to learn to generalize to states that it has not previously encountered. For environments where the state space is discrete and small, the agent may be able to explore all possible states and learn a good policy. In many RL problems, however, the state space is too large or even continuous for this to be feasible, so that the agent would need to learn to generalize from its limited set of interactions with the environment.

To formalize, at each discrete time step t, an agent chooses an *action* a_t from state s_t and transitions to a new state s_{t+1} according to an unknown dynamics model of the environment. For each transition, the agent is given a scalar *reward* r_{t+1} as a feedback which the agent uses to update its policy. The discounted return $R = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$ with $\gamma \in [0, 1]$ represents the discounted sum of future rewards accumulated by the agent. The objective of the agent is then to learn a good policy (or even an optimal policy) in order to maximize the expected discounted return (Arulkumaran et al., 2017). That is:

$$\pi^* = \arg\max_{\pi} \mathbb{E}[R \mid \pi]$$

This interaction can be seen as a Markov decision process (MDP) with the Markov assumption which states that the future state is conditionally independent of the past given the present state and action. An instance of MDP is defined by a set of states S, a set of actions A, a transition function $T(s' \mid s, a)$ representing the probability distribution over the possible next states following action a from state s, and a reward function $R(s, a)$ representing the immediate scalar reward given for executing a from s. In RL problems, states are often assumed to be *fully observable* by the agent. However, in many real world applications, states are more often than not only *partially observable* in which case the agent would need to learn to somehow account for this limited information. This generalization of MDP is called partially observable MDP or POMDP.

An agent can either learn a policy directly or construct one based on some other learned quantities. In value-based RL methods, an agent learns an estimate of the expected discounted return, represented as the value function $V^\pi(s)$ or the state-action value function $Q^\pi(s, a)$. Given such an estimate, a policy can be constructed by selecting an action that results in the highest estimated expected return from a particular s. Q-learning is one such value-based RL method that using the recursive relation

$$Q(s_t, a_t) = \mathbb{E}_{s_{t+1}}[r_t + \gamma Q^\pi(s_{t+1}, \pi(s_{t+1})]$$

iteratively improves the estimate with bootstrap target as

$$Q(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a))$$

where α is the learning rate. The algorithm converges to an optimal Q-function in the tabular case: $Q_i \to Q^*$ as $i \to \infty$.

2.2. Deep Q-Network

Estimating the Q-value for each state and action combination becomes impractical when the state and/or action spaces are large or even continuous. Instead, a function approximator parameterized by θ is often used to estimate the Q-function: $Q(s, a; \theta) \approx Q^*(s, a)$. A simple linear function can be used as the approximator or a more complex non-linear function such as a deep neural network can be used. Using deep networks as function approximator led to the Deep Q-network (DQN) algorithm (Mnih et al., 2013).

In DQN, a convolutional neural network was used to approximate the action values for a given state to train agents that learned human-level control policies from raw pixel image input. DQN employed two techniques to successfully train the agents. A target network, which is a periodic copy of the online Q-network, was used to estimate the bootstrap target. This was important in making training more stable. Use of a replay buffer, which contains a series of transitions, was also introduced to improve data efficiency and training stability. By storing the transitions that the agent experiences into a buffer and sampling from it, the same transition could be used for multiple updates and the inherent correlation among consecutive transitions could be avoided. With these techniques, DQN demonstrated human-level performance on Atari games.

3. Approach

DQN is one of early notable achievements in deep RL that demonstrated using the power of deep learning techniques to scale RL to problems previously considered intractable. However, subsequent work has revealed several limitations of the original algorithm and proposed several extensions to address them. We study a few of these extensions and consider several combinations to train integrated agents.

3.1. Deep RL

Double Q-Learning. In standard Q-learning, the bootstrap target is computed based on the reward received and the current estimate of the maximum action value. In case of

DQN, the Q-network is trained using gradient descent to minimize the loss

$$(r_{t+1} + \gamma_{t+1} Q(s_{t+1}, \arg\max_a Q(s_{t+1}, a; \theta); \boldsymbol{\theta}) - Q(s_t, a_t; \theta))^2$$

The max operator above selects and evaluates an action using the same values from the Q-network. This leads to selecting overestimated values which in turns leads to overly optimistic value estimates. Double Q-learning seeks to reduce this bias by untangling the selection from the evaluation (Hasselt et al., 2016). Since DQN uses a target network for updates, this can simply be implemented by using the target network for evaluation as

$$(r_{t+1} + \gamma_{t+1} Q(s_{t+1}, \arg\max_a Q(s_{t+1}, a; \theta); \boldsymbol{\theta^-}) - Q(s_t, a_t; \theta))^2$$

where θ^- represents the weights of the target network. Note that this incurs no additional memory compared to DQN. Double Q-learning has been shown to improve convergence speed as well as the quality of the solution it converges to on a range of environments.

Multi-Step Target. Conventional Q-learning uses reward from a single step and the greedy action at the next step to bootstrap. In an environment with sparse rewards, it can take many steps before reaching a non-zero reward and propagating it to the states visited earlier. Instead, we can accumulate rewards over multiple steps, and use the accumulated return as the bootstrap target. More formally, define the truncated n-step return as

$$r_t^{(n)} = \sum_{k=0}^{n-1} \gamma_t^{(k)} r_{t+k+1}$$

then the alternative loss becomes

$$(r_t^{(n)} + \gamma_t^{(n)} Q(s_{t+n}, \arg\max_a Q(s_{t+n}, a; \theta); \theta) - Q(s_t, a_t; \theta))^2$$

It has been shown that with a suitable choice of n, this can lead to faster convergence (Sutton & Barto, 2018).

Actor-Critic. In actor-critic methods, both an action function $\mu(s \mid \theta^\mu)$ that represents the current policy and a critic that is used as baseline in gradient computation are utilized. In case the critic is the value function $Q(s, a \mid \theta)$, it becomes Q actor-critic. While DQN is able to handle problems with high-dimensional observation spaces, it handles only discrete and low-dimensional action spaces. Deep deterministic policy gradient (DDPG) (Lillicrap et al., 2016) is an actor-critic, model-free algorithm that extends DQN to specifically handle this limitation. In DDPG, the critic function is optimized by minimizing the loss

$$\frac{1}{N} \sum_i (y_i - Q(s_i, a_i \mid \theta^Q))^2$$

and the actor policy with the policy gradient

$$\frac{1}{N} \sum_i \nabla_q Q(s, a \mid \theta^Q) \mid_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s \mid \theta^\mu) \mid_{s_i}$$

DDPG has shown promising results on a number of challenging physical control problems that involve high-dimensional action spaces. Twin delayed deep deterministic policy gradient (TD3) is yet another improvement on DDPG that better handles overestimation bias present in actor-critic architectures (Fujimoto et al., 2018). It does this by borrowing the idea from the original double Q-learning of using two separate networks to make value predictions (Hasselt, 2010). We consider this particular actor-critic method in this work.

Recurrent Neural Networks. The DQN architecture used for Atari 2600 games in (Mnih et al., 2013) took four consecutive frames as input to the Q-network. This was necessary because information such as velocity cannot be inferred from a single frame. However, if the same agent is used for a game in which a longer sequence of frames needs to be processed, the environment would no longer be Markovian and the agent would likely perform badly. To handle such partial-observability of state information, which is common in many real-world environments, recurrent neural networks (RNN) have been used with algorithms such as DQN. In (Hausknecht & Stone, 2015), LSTM was combined with DQN to train agents that performed comparably on modified Atari environments where some state information was intentionally dropped.

3.2. Combined Agent

The algorithms discussed above address different shortcomings of the original DQN algorithm. Some of these techniques can be naturally combined to train integrated agents. Specifically, in this work, we consider the following in fully-observable settings

- DQN and multi-step target
- Double Q-learning and multi-step target
- TD3

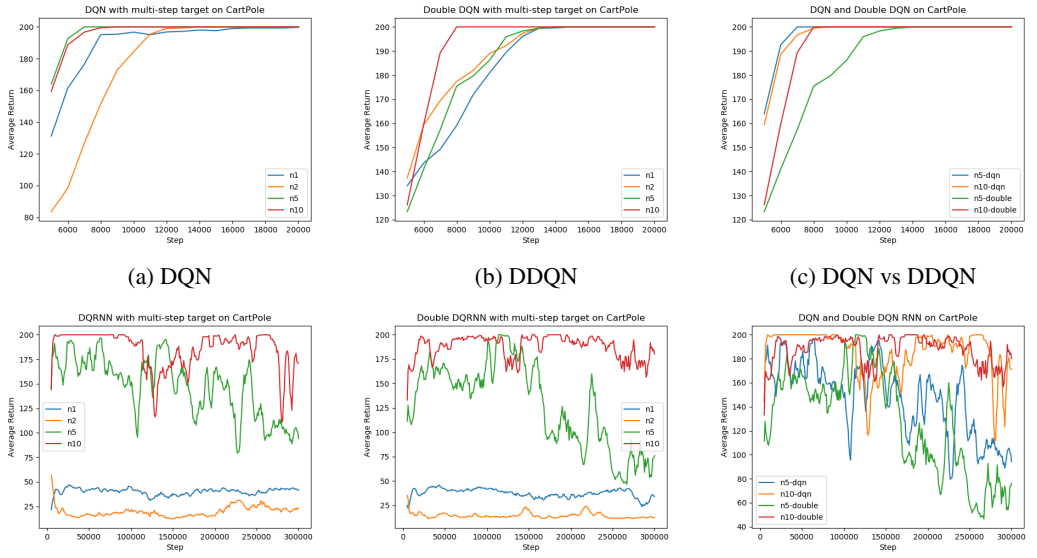
and the following in partially-observable settings

- DQN with RNN and multi-step target
- Double Q-learning with RNN and multi-step target
- TD3 with RNNs and multi-step target

For each of these agents, we followed the high-level flow as outlined below for training and eval.

4. Experiments

We have performed experiments on classic control environments including CartPole, MountainCar, and MountainCar-Continuous. To simulate partial-observability, we created a



(a) DQN (b) DDQN (c) DQN vs DDQN

(d) RNN-DQN (e) RNN-DDQN (f) RNN-DQN vs RNN-DDQN

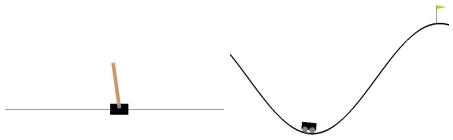
Figure 1. Learning curves for fully-observable and partially-observable CartPole environments.

Algorithm 1 Deep RL Training

Initialize replay buffer with random exploration strategy
Initialize function approximator networks
for episode $\leftarrow 1$ to M **do**
  for $t \leftarrow 1$ to T **do**
    Select a_t based on a chosen exploration strategy
    Execute a_t and observe reward r_t and new state s_{t+1}
    Sample a random batch b of transitions from the replay buffer
    Compute predictions for the batch (e.g. $\hat{Q}(b;\theta)$)
    Compute gradients and update network weights (e.g. $(y - \hat{Q}(b;\theta))^2$)
  end for
end for

new set of environments where we removed a subset of the state components to experiment with the RNN-based agents. All experiments were conducted using the TensorFlow library (Abadi et al., 2015).

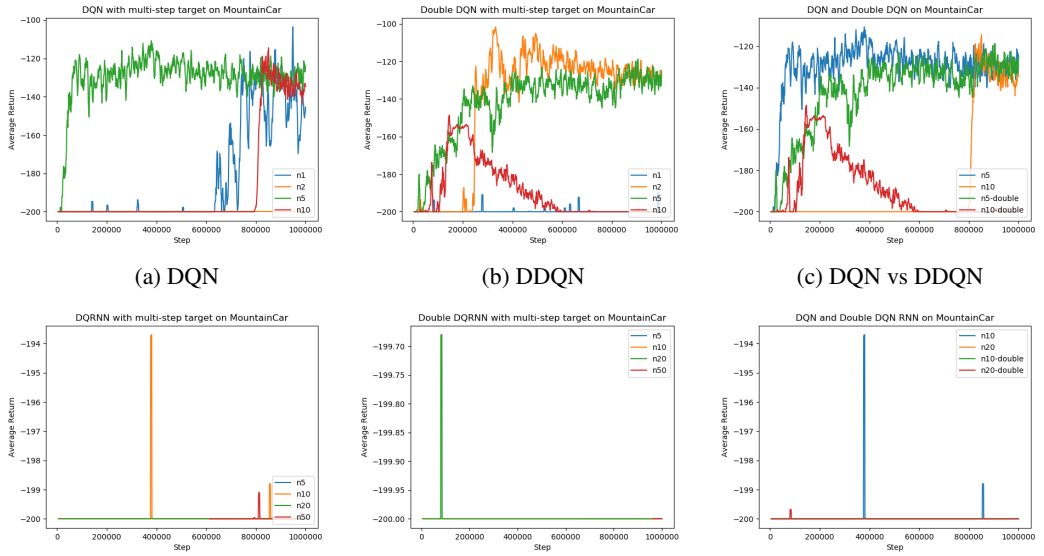
In CartPole, a pole is attached to a cart that is free to move along a friction-less, horizontal track, and the objective is to apply a left or a right force of a fixed magnitude at each time step to prevent the pole from falling over (Barto et al., 1990). For the partial environment, we removed both the vertical angle of the pole and the angular velocity so that the agent would need to infer them based only on the position and velocity of the cart. In MountainCar, a car is situated between two uphills, and the objective is to drive the car over to the right hill. The agent needs to drive the car back and forth to build up enough momentum to climb up the hill in the least amount of time as possible (Moore, 1990). For the partial environment, we dropped the velocity component to provide only the positional information to the agent. MountinCarContinuous is the continuous version of the MountainCar environment where the agent needs to provide a real number representing the magnitude of force to apply at each step. The actor-critic method TD3 was tested on this environment.



(a) CartPole (b) MountainCar

Figure 2. Classic control gym environments.

The overall experiment setup is as follows. We start by initializing the replay buffer with 100k transitions collected from a random exploration. Once it is full, we enter into the training loop to train the agent using batches of transitions sampled from the buffer. We make appropriate predictions for the batch (e.g. $\hat{Q}(b;\theta)$ for DQN), compute the gradients, and update the network weights. Weight update was done using the Adam optimizer (Kingma & Ba, 2015). For exploration, we used (a) ϵ-greedy where we choose an optimal action with probability $(1-\epsilon)$ or a random action with probability ϵ, (b) Boltzmann sampling where actions are chosen with probabilities proportional to the current value estimates, and (c) exploration noise drawn from the Gaussian for TD3 used for the continuous action space case.



(a) DQN (b) DDQN (c) DQN vs DDQN

(d) RNN-DQN (e) RNN-DDQN (f) RNN-DQN vs RNN-DDQN

Figure 3. Learning curves for fully-observable and partially-observable MountainCar environments.

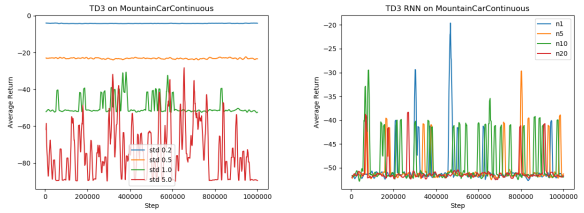
Multi-Step Target. Figure 1 shows learning curves of the agents trained with multi-step target on both the original CartPole and the partially-observable counterpart. In the fully-observable case, with $n = 5$ and $n = 10$ the convergence speed is clearly faster than the single-step bootstrap target case, and even with $n = 2$ the maximum average return was attained in a fewer number of steps. The effect of using multi-step target is more dramatic in the partially-observable case. The RNN-based DQN (and DDQN) agent struggled to achieve average return of above 50 when trained with a single-step target or $n = 2$ even after 300k training steps. Only when trained with $n = 5$ or $n = 10$, the agent was able to achieve good performance. Between $n = 5$ and $n = 10$, the latter showed noticeably more stable learning curve, and the former in fact showed signs of degrading performance as training continued.

Figure 3 shows a similar set of graphs for the MountainCar environment. For this environment also, combining multi-step target with DQN or DDQN had noticeably better convergence than the single-step target case when a suitable value of n was used. On the other hand, in contrast to CartPole, too large of a value for n led to worse convergence. For DQN, as an example, both $n = 1$ and $n = 10$ struggled for the initial 500k or so steps, and $n = 1$ started to improve before $n = 10$. This demonstrates that the value of n in multi-step target controls the bias-variance trade-off and hence need to be chosen carefully for optimal performance.

Double Q-Learning. In Figure 1, plots (c) and (f) show comparison between vanilla DQN and Double Q-learning each combined with multi-step target on CartPole. For a certain configuration such as $n = 10$ on partial CartPole, combining it with Double Q-learning shows more stable learning curve than the vanilla DQN. However, this behavior was not consistent across configurations; for $n = 5$ on partial CartPole, we observe the opposite. In certain cases, Double Q-learning even led to slightly slower convergence than DQN, as can be seen from the fully-observable CartPole case. In Figure 3, plots (c) and (f) show equivalent graphs for MountainCar, and we observe similar behaviors. That is, when combined with multi-step target with a suitable n, Double Q-learning shows more stable convergence, but for other cases the additional value it brings over vanilla DQN is less clear.

We suspect that this might be due that both multi-step target and Double Q-learning are used to address some form of bias in the RL training. The former is used when a better trade-off between bias and variance for the target reward can be attained by considering a longer sequence of reward signals. Similarly, the latter is used to reduce the maximization bias present in the max operator of Q-learning when computing the bootstrap target. Since the two types of bias are closely related, it is possible that addressing one type of bias can potentially help address the other type as well. If this is the case, employing only one of the two techniques may be sufficient.

Partial-Observability. On the partial CartPole environment, where we dropped two of the four state components, RNN was highly effective in handling partial-observability when combined with a suitable multi-step target. Despite that half of the components were removed, RNN-based



(a) TD3 (b) RNN-TD3

Figure 4. TD3 on MountainCarContinuous.

DQN and DDQN agents both managed to achieve the maximum average return of 200 when $n = 10$ multi-step target was used. The learning curves, however, were still noticeably unstable compared to the fully-observable counterparts. In contrast, RNN combined with multi-step target was not sufficient to adequately handle partial-observability in MountainCar and MountainCarContinuous which are more complex than CartPole. It might be that the capacity of the RNN model used in the experiments was insufficient to handle the complexity of the environment. Or, a more sophisticated strategy for sampling transitions from the reply buffer was needed to speed up convergence. A more principled way of using RNN-based function approximators to deal with partially-observable environments is something worth pursuing as future work.

Continuous Action Space. MountainCarContinuous is the continuous version of MountainCar in which the agent provides a real number force to apply to the car. If the car reaches the destination, a reward of 100 is given. Otherwise, a reward equal to the square of the predicted force multiplied by -0.1 is given to the agent. Hence, applying a greater force may help reach the destination more quickly but also results in greater negative immediate rewards.

This environment was much more challenging than the other environments, and the main difficulty was in appropriately configuring the exploration setting to backpropgate sufficient gradients to update the network weights. In case of TD3, this is the standard deviation of the Gaussian from which exploration noise is drawn and added to the prediction. Figure 4(a) shows learning curves of agents trained with different values of the Gaussian standard deviation, and it depicts well the tension between exploration and exploitation. Since larger negative rewards are given for larger forces, the agent myopically optimized its return by applying as little force as possible, which is a losing strategy in the long run as it would not help the agent obtain the reward of 100 given when the car reaches the destination. In one experiment, we tried standard deviation of 5.0 to dramatically increase the magnitude of the exploration noise, and this did in some iterations reached higher average return than the one with standard deviation of 1.0. But finding an appropriate level of exploration that would keep immediate rewards from being too low and concurrently be sufficient to help the agent obtain the max reward was a challenge even after 1M training steps. In the partial environment without the velocity component, we experimented with several RNN-based TD3 agents trained with multi-step target. The performance was on par with some of the low-performing agents on the fully-observable environment but was still less than ideal. It was clear from the experiments that (a) a more sophisticated exploration strategy for environments with continuous action spaces and (b) a more principled approach to combining RNN models with agents designed for such environments would both be interesting directions to pursue.

5. Discussion

Deep Q-network is a notable early achievement that successfully employed deep learning techniques to dramatically scale conventional reinforcement learning algorithms. However, DQN revealed limitations in several respects, and subsequent extensions were independently proposed to improve different aspects of the algorithm. In this work, we considered a set of these extensions and studied the effectiveness of combining them in terms of convergence speed and performance. In particular, we studied (a) Double DQN used to reduce the overestimation bias in DQN, (b) multi-step bootstrap target that can better balance bias and variance, (c) TD3 which is an actor-critic architecture widely used to handle continuous action space, and (d) RNN-based agents used to handle partial-observability of the environment state.

We were able to make several interesting observations from our experimental study. First, using multi-step target of a suitable length can indeed result in both faster convergence and better performance. However, as the length of the target decides the bias-variance trade-off, choosing an appropriate length was crucial. Second, DDQN can often lead to more stable convergence but when also combined with multi-step target, its additional contribution appeared to be small. We suspect that this has to do with the fact that both techniques are used to address some forms of biases that are closely related, so that using one can often be sufficient. Third, RNN-based agents can handle partial-observability quite well especially when combined with using multi-step target. This was especially evident in the partial CartPole environment. Last, actor-critic methods can be used handle continuous action space, but finding an appropriate level of exploration can be challenging especially when more exploration can directly lead to less immediate rewards, as in the MountainCarContinuous environment.

As future work, it would be worth exploring more principled approach to using RNN-based agents for partially-observable environments and finding a suitable exploration

strategy in the context of continuous action space environments.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Arulkumaran, K., Deisenroth, M. P., Brundage, M., and Bharath, A. A. A brief survey of deep reinforcement learning. *CoRR*, abs/1708.05866, 2017. URL http://arxiv.org/abs/1708.05866.

Barto, A. G., Sutton, R. S., and Anderson, C. W. Artificial neural networks. chapter Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems, pp. 81–93. IEEE Press, Piscataway, NJ, USA, 1990. ISBN 0-8186-2015-3. URL http://dl.acm.org/citation.cfm?id=104134.104143.

Fujimoto, S., Hoof, H., and Meger, D. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pp. 1582–1591, 2018.

Hasselt, H. V. Double q-learning. In Lafferty, J. D., Williams, C. K. I., Shawe-Taylor, J., Zemel, R. S., and Culotta, A. (eds.), *Advances in Neural Information Processing Systems 23*, pp. 2613–2621. Curran Associates, Inc., 2010. URL http://papers.nips.cc/paper/3964-double-q-learning.pdf.

Hasselt, H. v., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI16, pp. 20942100. AAAI Press, 2016.

Hausknecht, M. J. and Stone, P. Deep recurrent q-learning for partially observable mdps. *CoRR*, abs/1507.06527, 2015. URL http://arxiv.org/abs/1507.06527.

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. G., and Silver, D. Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017. URL http://arxiv.org/abs/1710.02298.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. In *ICLR*, 2016. URL http://dblp.uni-trier.de/db/conf/iclr/iclr2016.html#LillicrapHPHETS15.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

Moore, A. W. Efficient memory-based learning for robot control. Technical report, 1990.

Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.