Homework 3: Q-Learning, REINFORCE and Advantage-Actor-Critic

CMU 10-403: Deep Reinforcement Learning (Spring 2020)

OUT: Mar. 5, 2020
DUE: Mar. 28, 2020 by 11:59pm

Instructions: START HERE

- **Collaboration Policy:** You may work in groups of up to two people for this assignment. It is also OK to get clarification (but not solutions) from books or online resources after you have thought about the problems on your own. You are expected to comply with the University Policy on Academic Integrity and Plagiarism[1].
- **Late Submission Policy:** You are allowed a total of 10 grace days for your homeworks. However, no more than 3 grace days may be applied to a single assignment. Any assignment submitted after 3 days will not receive any credit. Grace days do not need to be requested or mentioned in emails; we will automatically apply them to students who submit late. We will not give any further extensions so make sure you only use them when you are absolutely sure you need them. See the Assignments and Grading Policy here for more information about grace days and late submissions: `https://cmudeeprl.github.io/Spring202010403website/logistics/`
- **Submitting Your Work:**
  - **Gradescope:** Please write your answers and copy your plots into the provided LaTeX template, and upload a PDF to the GradeScope assignment titled "Homework 3." Additionally, upload your code to the GradeScope assignment titled "Homework 3: Code." Each team should only upload one copy of each part. Regrade requests can be made within one week of the assignment being graded.
  - **Autolab:** Autolab is not used for this assignment.
- **Kind Reminder:** We strongly suggest you **START EARLY**.

[1] `https://www.cmu.edu/policies/`

Problem 0: Collaborators

Please list your name and Andrew ID, as well as those of your collaborators.

Problem 1: DQN (47+20 pts)

In this problem you will implement Q-learning, using tabular and learned representations for the Q-function. This question will be graded out of 47 points, but you can earn up to 67 points by completing the extra credit problem (1.3c).

Problem 1.1: Relations among Q & V & C (13 pts)

The objective of this question is to understand different Bellman Optimality Equations, their strengths and limitations. Consider the Bellman Optimality Equation for the Value function,

$$V(s_1) = \max_{a_1}\left(R(s_1,a_1) + \gamma \sum_{s_2} T(s_1,a_1,s_2)V(s_2)\right).$$

If we continue expanding the value function $V(s_2)$ using its own Bellman Optimality Equation, then we obtain a repeating structure:

$$V(s_1) = \max_{a_1}\left(R(s_1,a_1) + \gamma \sum_{s_2} T(s_1,a_1,s_2)\max_{a_2}\left(R(s_2,a_2) + \gamma \sum_{s_3} T(s_2,a_2,s_3)V(s_3)\right)\right).$$

There are a few more ways in which we can group this repeating sequence. First, we can capture the sequence starting at $R(s,a)$ and ending at max, and observe that it too has a repeating substructure property:

$$V(s_1) = \max_{a_1}\underbrace{\left(R(s_1,a_1) + \gamma \sum_{s_2} T(s_1,a_1,s_2)\max_{a_2}\underbrace{\left(R(s_2,a_2) + \gamma \sum_{s_3} T(s_2,a_2,s_3)V(s_3)\right)}_{Q(s_2,a_2)}\right)}_{Q(s_1,a_1)}.$$

We'll call this repeating expression the state-value function $Q(s,a)$ and use it to rewrite the Bellman Optimality equation as:

$$Q(s_1,a_1) = R(s_1,a_1) + \gamma \sum_{s_2} T(s_1,a_1,s_2)\max_{a_2} Q(s_2,a_2).$$

Next, we can capture another pattern by grouping the expression beginning at γ and ending at $R(s, a)$:

$$V(s_1) = \max_{a_1}\left(R(s_1, a_1) + \underbrace{\gamma \sum_{s_2} T(s_1, a_1, s_2) \max_{a_2}\Big(R(s_2, a_2) + \underbrace{\gamma \sum_{s_3} T(s_2, a_2, s_3) V(s_3)\Big)}_{C(s_2,a_2)}}_{C(s_1,a_1)} \right).$$

We'll call this repeating expression the continuation function $C(s, a)$, which can be written in terms of the value function:

$$C(s_1, a_1) = \gamma \sum_{s_2} T(s_1, a_1, s_2) V(s_2).$$

1. (3 pts) Derive the recurrence relation (Bellman Optimality Equation) for $C(s, a)$.
2. (4 pts) Fill the following table to express the three functions in terms of each other.

	V(s)	Q(s,a)	C(s,a)
V(s)	V(s) = V(s)	V(s) = $\max_a Q(s, a)$	(a)
Q(s,a)	(b)	Q(s,a) = Q(s,a)	(c)
C(s,a)	$C(s, a) = \gamma \sum_{s'} T(s, a, s') V(s')$	(d)	C(s,a) = C(s,a)

Use the relation between the functions and your understanding of MDPs to answer the following True/False questions. Please include a 1-2 sentence explanation for each. Consider the scenario when we want to compute the optimal action without the knowledge of transition function $T(s, a, s')$.

3. (2 pts) Can you derive the optimal policy given only $Q(s, a)$?
4. (2 pts) Can you derive the optimal policy given only $V(s)$ and $R(s, a)$?
5. (2 pts) Can you derive the optimal policy given only $C(s, a)$ and $R(s, a)$?

Problem 1.2: Temporal Difference & Monte Carlo (4 pts)

Answer the true/false questions below, providing one or two sentences for **explanation**.

1. (2 pts) TD methods can't learn in an online manner since they require full trajectories.
2. (2 pts) MC can be applied even with non-terminating episodes.

Problem 1.3: DQN Implementation (30 + 20 pts)

You will implement DQN and use it to solve two problems in OpenAI Gym: `Cartpole-v0` and `MountainCar-v0`. While there are many (fantastic) implementations of DQN on Github, the goal of this question is for you to implement DQN from scratch *without* looking up code online.[2] Please write your code in the `DQN_implementation.py`. You are free to change/delete the template code if you want.

Code Submission: Your code should be reasonably well-commented in key places of your implementation. Make sure your code also has a README file.

How to measure if I "solved" the environment? You should achieve the reward of 200 (`Cartpole-v0`) and around -110 or higher (`MountainCar-v0`) in consecutive 50 trials. *i.e.* evaluate your policy on 50 episodes.

Runtime Estimation: To help you better manage your schedule, we provide you with a reference runtime of DQN on a MacBook Pro 2018. For Cartpole-v0, it takes 5 minutes to first reach a reward of 200 and 68 minutes to finish 5000 episodes. For MountainCar-v0, it takes 40 $\sim$ 50 minutes to reach a reward around -110 and 200 minutes to finish 10000 episodes.

1 **(30 pts)** Implement a deep Q-network with experience replay. While the original DQN paper [5] uses a convolutional architecture, a neural network with 3 fully-connected layers should suffice for the low-dimensional environments that we are working with. For the deep Q-network, look at the `QNetwork` and `DQN_Agent` classes in the code. You will have to implement the following:

- Create an instance of the Q Network class.
- Create a function that constructs a greedy policy and an exploration policy (ϵ-greedy) from the Q values predicted by the Q Network.
- Create a function to train the Q Network, by interacting with the environment.
- Create a function to test the Q Network's performance on the environment.

For the replay buffer, you should use the experimental setup of [5] to the extent possible. Starting from the `Replay_Memory` class, implement the following functions:

- Append a new transition from the memory.
- Sample a batch of transitions from the memory to train your network.
- Collect an initial number of transitions using a random policy.
- Modify your training function of your network to learn from experience sampled *from the memory*, rather than learning online from the agent.

Train your network on both the `CartPole-v0` environment and the `MountainCar-v0` environment (separately) until convergence, *i.e.* train a different network for each environment. We recommend that you periodically checkpoint your network to ensure no work is lost if your program crashes. Answer following questions in your report:

[2] After this assignment, we highly recommend that you look at DQN implementations on Github to see how others have structured their code.

(a) (17 pts) Describe your implementation, including the optimizer, the neural network architecture and any hyperparameters you used.

(b) (5 pts) For each environment, plot the average cumulative test reward throughout training.[3] You are required to plot at least 2000 more episodes after you solve CartPole-v0, and at least 1000 more episodes after you solve MountainCar-v0. To do this, every 100 episodes, evaluate the current policy for 20 episodes and average the total reward achieved. Note that in this case we are interested in total reward without discounting or truncation.

(c) (5 pts) For each environment, plot the TD error throughout training. Does the TD error decrease when the reward increases? Suggest a reason why this may or may not be the case.

(d) (3 pts) We want you to generate a *video capture* of an episode played by your trained Q-network at different points of the training process (0/3, 1/3, 2/3, and 3/3 through the training process) of both environments. We provide you with a helper function to create the required video captures in `test_video()`.

2 **(20 pts, optional)** Implement any of the modifications below. Describe what you implemented, and run some experiments to determine if the modifications yield a better RL algorithm. You may implement multiple of the modifications, but you will not receive more than 20 points of extra credit.

(a) (20 pts) Double DQN, as described in [9].

(b) (20 pts) Dueling DQN, as described in [10].

(c) (20 pts) Residual DQN, as described in [1].

Guidelines on References

We recommend you to read all the papers mentioned in the references. There is a significant overlap between different papers, so in reality you should only need certain sections to implement what we ask of you. We provide pointers for relevant sections for this assignment for your convenience.

The work in [4] contains the description of the experimental setup. Algorithm 1 describes the main algorithm. Section 3 (paragraph 3) describes the replay memory. Section 4 explains preprocessing (paragraph 1) and the model architecture (paragraph 3). Section 5 describes experimental details, including reward truncation, the optimization algorithm, the exploration schedule, and other hyperparameters). The methods section in [5], may clarify a few details so it may be worth to read selectively if questions remain after reading [4].

[3]You can use the `Monitor` wrapper to generate both the performance curves and the video captures.

Guidelines on Hyperparameters

In this assignment you will implement improvements to the simple update Q-learning formula that make learning more stable and the trained model more performant. We briefly comment on the meaning of each hyperparameter and some reasonable values for them.

- Discount factor γ: 1.0 for MountainCar, and 0.99 for CartPole.
- Learning rate α: 0.001 for Cartpole and 0.0001 for Mountaincar.
- Exploration probability ϵ in ϵ-greedy: While training, we suggest you start from a high epsilon value, and anneal this epsilon to a small value (0.05 or 0.1) during training. We have found decaying epsilon linearly from 0.5 to 0.05 over 100000 iterations works well. During test time, you may use a greedy policy, or an epsilon greedy policy with small epsilon (0.05).
- Number of training episodes: For MountainCar-v0, you should see improvements within 2000 (or even 1000) episodes. For CartPole-v0, you should see improvements starting around 2000 episodes.

  Look at the average reward achieved in the last few episodes to test if performance has plateaued; it is usually a good idea to consider reducing the learning rate or the exploration probability if performance plateaus.
- Replay buffer size: 50000; this hyperparameter is used only for experience replay. It determines how many of the last transitions experienced you will keep in the replay buffer before you start rewriting this experience with more recent transitions.
- Batch size: 32; typically, rather doing the update as in (2), we use a small batch of sampled experiences from the replay buffer; this provides better hardware utilization.

In addition to the hyperparameters:

- Optimizer: You may want to use Adam as the optimizer. Think of Adam like a fancier SGD with momentum, it will automatically adjust the learning rate based on the statistics of the gradients its observing.
- Loss function: you can use Mean Squared Error.

The implementations of the methods in this homework have multiple hyperparameters. These hyperparameters (and others) are part of the experimental setup described in [4, 5]. For the most part, we strongly suggest you to follow the experimental setup described in each of the papers. [4, 5] was published first; your choice of hyperparameters and the experimental setup should follow closely their setup. We recommend you to read all these papers. We have given pointers for the most relevant portions for you to read in a previous section.

REINFORCE/A2C Installation instructions (Linux)

In the next part, you will implement 2 policy gradient algorithms and evaluate them on the OpenAI Gym `LunarLander-v2` environment. This environment is considered solved if the agent can achieve an average score of at least 200. We've provided Python packages that you may need in `requirements.txt`. To install these packages using pip and virtualenv, run the following commands:

```
apt-get install swig
virtualenv env
source env/bin/activate
pip install -U -r requirements.txt
```

If your installation is successful, then you should be able to run the provided template code:

```
python reinforce.py
python a2c.py
```

Note: You will need to install `swig` and `box2d` in order to install `gym[box2d]`, which contains the `LunarLander-v2` environment. You can install `box2d` by running

```
pip install git+https://github.com/pybox2d/pybox2d
```

If you simply do `pip install box2d`, you can sometimes get an error because the pip package for `box2d` depends on an older version of `swig`.[4] For additional installation instructions, see https://github.com/openai/gym.

[4] https://github.com/openai/gym/issues/100

Problem 2: REINFORCE (30 pts)

In this section, you will implement episodic REINFORCE [11], a policy-gradient learning algorithm. Please write your code in `reinforce.py`; the template code provided inside is there to give you an idea on how you can structure your code, but is not mandatory to use.

Policy gradient methods directly optimize the policy $\pi(A \mid S, \theta)$, which is parameterized by θ. The REINFORCE algorithm proceeds as follows. We generate an episode by following policy π. After each episode ends, for each time step t during that episode, we update the policy parameters θ with the REINFORCE update. This update is proportional to the product of the return G_t experienced from time step t until the end of the episode and the gradient of $\log \pi(A_t \mid S_t, \theta)$. See Algorithm 1 for details.

Algorithm 1 REINFORCE

```
1: procedure REINFORCE
2:     Start with policy model $\pi_\theta$
3:     repeat:
4:         Generate an episode $S_0, A_0, r_0, \ldots, S_{T-1}, A_{T-1}, r_{T-1}$ following $\pi_\theta(\cdot)$
5:         for $t$ from $T-1$ to 0:
6:             $G_t = \sum_{k=t}^{T-1} \gamma^{k-t} r_k$
7:         $L(\theta) = \frac{1}{T} \sum_{t=0}^{T-1} G_t \log \pi_\theta(A_t|S_t)$
8:         Optimize $\pi_\theta$ using $\nabla L(\theta)$
9: end procedure
```

For the policy model $\pi(A \mid S, \theta)$, we recommend starting with a model that has:

- three fully connected layers with 16 units each, each followed by ReLU activations
- another fully connected layer with 4 units (the number of actions)
- a softmax activation (so the output is a proper distribution)

Initialize bias for each layer to zero. We recommend using a variance scaling kernel initializer that draws samples from a uniform distribution over $[-\alpha, \alpha]$ for $\alpha = \sqrt{(3 * \text{scale}/n)}$ where scale $= 1.0$ and n is the average of the input and output units. HINT: Read the Keras documentation.

You can use the `model.summary()` and `model.get_config()` calls to inspect the model architecture.

You can choose which optimizer and hyperparameters to use, so long as they work for learning on `LunarLander-v2`. We recommend using Adam as the optimizer. It will automatically adjust the learning rate based on the statistics of the gradients it's observing. You can think of it like a fancier SGD with momentum. Keras provides a version of Adam `https://keras.io/optimizers/`.

Train your implementation on the `LunarLander-v2` environment until convergence[5]. Be sure to keep training your policy for at least 1000 more episodes after it reaches 200 reward so that you are sure it consistently achieves 200 reward and so that this convergence is reflected in your graphs. Then, answer the following questions.

1. [10 pts] Describe your implementation, including the optimizer and any hyperparameters you used (learning rate, γ, etc.). Your description should be detailed enough that someone could reproduce your results.

2. [20 pts] Plot the learning curve: Every k episodes, freeze the current cloned policy and run 100 test episodes, recording the mean and standard deviation of the cumulative reward. Plot the mean cumulative reward on the y-axis with the standard deviation as error-bars against the number of training episodes on the x-axis. Write a paragraph or two describing your graph(s) and the learning behavior you observed. Be sure to address the following questions:

   - What trends did you see in training?
   - How does the final policy perform?
   - The REINFORCE algorithm may be unstable. If you observe such instability in your implementation, what could be the reason?

   Hint: You can use matplotlib's `plt.errorbar()` function. https://matplotlib.org/api/_as_gen/matplotlib.pyplot.errorbar.html

Problem 3: Advantage-Actor Critic (40 pts)

In this section, you will implement N-step Advantage Actor Critic (A2C) [2]. Please write your code in `a2c.py`; the template code provided inside is there to give you an idea on how you can structure your code, but is not mandatory to use.

Algorithm 2 N-step Advantage Actor-Critic

```
1: procedure N-Step Advantage Actor-Critic
2:     Start with policy model $\pi_\theta$ and value model $V_\omega$
3:     repeat:
4:         Generate an episode $S_0, A_0, r_0, \ldots, S_{T-1}, A_{T-1}, r_{T-1}$ following $\pi_\theta(\cdot)$
5:         for $t$ from $T-1$ to 0:
6:             $V_{end} = 0$ if $(t + N \geq T)$ else $V_\omega(s_{t+N})$
7:             $R_t = \gamma^N V_{end} + \sum_{k=0}^{N-1} \gamma^k \left(r_{t+k} \text{ if } (t+k < T) \text{ else } 0\right)$
8:         $L(\theta) = \frac{1}{T}\sum_{t=0}^{T-1}(R_t - V_\omega(S_t)) \log \pi_\theta(A_t|S_t)$
9:         $L(\omega) = \frac{1}{T}\sum_{t=0}^{T-1}(R_t - V_\omega(S_t))^2$
10:        Optimize $\pi_\theta$ using $\nabla L(\theta)$
11:        Optimize $V_\omega$ using $\nabla L(\omega)$
12: end procedure
```

[5] `LunarLander-v2` is considered solved if your implementation can attain an average score of at least 200.

N-step A2C provides a balance between bootstraping using the value function and using the full Monte-Carlo return, using an N-step trace as the learning signal. See Algorithm 2 for details. N-step A2C includes both REINFORCE with baseline ($N = \infty$) and the 1-step A2C covered in lecture ($N = 1$) as special cases and is therefore a more general algorithm.

The critic updates the state-value parameters ω, and the actor updates the policy parameters θ in the direction suggested by the N-step trace.

Start off with the same policy architecture described in Problem 1 for both the actor and the critic. Play around with the network architecture of the critic's state-value approximator to find one that works for `LunarLander-v2`. Once again, you can choose which optimizer and hyperparameters to use, so long as they work for learning on `LunarLander-v2`.

Answer the following questions:

1. [10 pts] Describe your implementation, including the optimizer, the critic's network architecture, and any hyperparameters you used (learning rate, γ, etc.).
2. [20 pts] Train your implementation on the `LunarLander-v2` environment several times with N varying as [1, 20, 50, 100] (it's alright if the N=1 case is hard to get working). Plot the learning curves for each setting of N in the same fashion as Problem 1. You may find that your plots with error bars will be too busy to plot all values of N on the same graph. If this is the case, make a different plot for each value of N. Once again, write a paragraph or two describing your graph(s) and the learning behavior you observed. Be sure to address the following questions:
    - What trends did you observe in training?
    - How does the final policy perform?
    - If you found A2C to be unstable or otherwise difficult to train, why might this be the case? What about the algorithm formulation could cause training instability, and what improvements might be made to improve it?
3. [10 pts] Discuss how the performance of your implementation of A2C compares with REINFORCE and how A2C's performance varies with N. Which algorithm and N setting learns faster, and why do you think this is the case?

Extra credit (up to 15 pts)

A major bottleneck in training policy gradient algorithms is that only one episode (or batch of states) is generated at a time. However, once the policy has been updated once, the training data is no longer drawn from the current policy distribution, becoming "invalid" in a sense. A similar challenge occurs when parallelizing training, since once a parameter update is performed by one worker, the policy distribution changes and invalidates the data gathered and gradients computed by the other workers. Mnih *et al.* argue that the exploration noise from asynchronous policy updates can be beneficial to learning [3].

First, let's introduce a more complex environment. Many deep reinforcement learning papers (at least in the past few years) have used Atari games as performance benchmarks due to their greater complexity. Apply your implementation of A2C to any of the OpenAI gym Breakout environments. We recommend either `Breakout-v0` or `BreakoutNoFrameskip-v4` environments. You will need to use a larger, more complex policy network than the one you used in Problem 1 and 2, as well as some tricks like learning rate decay. Think carefully about your hyperparameters, particularly N. You should be able to reach at least 200 average reward after 10-20 hours of training on AWS; note that running these large networks on a laptop may take up to two weeks.

Then, implement multi-threaded synchronous Advantage Actor-Critic by gathering episode rollouts in parallel and performing a single gradient update. What speedup can you achieve? How might you measure this? Then, implement Asynchronous Advantage Actor-Critic (A3C) with multiple threads, using your multi-threaded synchronous Advantage Actor-Critic as a starting point. Do you see a learning speedup or increased stability compared to a synchronous implementation?

Up to 15 points extra credit will be awarded total, contingent on implementation, results, and analysis. Describe how you implemented the task and provide metrics and graphs showing improvement as well as explanations as to why that might be the case. You may also wish to include links to videos of your trained policies. If nothing else, it is entertaining and rewarding to see an agent you trained play Breakout at a superhuman level.

Guidelines on implementation

This homework requires a significant implementation effort. It is hard to read through the papers once and know immediately what you will need to be implement. We suggest you to think about the different components (e.g., model definition, model updater, model runner, ...) that you will need to implement for each of the different methods that we ask you about, and then read through the papers having these components in mind. By this we mean that you should try to divide and implement small components with well-defined functionalities rather than try to implement everything at once. Much of the code and experimental setup is shared between the different methods so identifying well-defined reusable components will save you trouble.

Some hyperparameter and implementation tips and tricks:

- For efficiency, you should try to vectorize your code as much as possible and use **as few loops as you can** in your code. In particular, in lines 5 and 6 of Algorithm 1 (REINFORCE) and lines 5 to 7 of Algorithm 2 (A2C) you should not use two nested loops. How can you formulate a single loop to calculate the cumulative discounted rewards? Hint: Think backwards!
- Moreover, it is likely that it will take between 10K and 50K episodes for your model to converge, though you should see improvements within 5K episodes (about 30 minutes to one hour). On a NVIDIA GeForce GTX 1080 Ti GPU, it takes about five hours to run 50K training episodes with our REINFORCE implementation.

- For A2C, downscale the rewards by a factor of 1e-2 (i.e., divide by 100) when training (but not when plotting the learning curve) This will help with the optimization since the initial weights of the critic are far away from being able to predict a large range such as $[-200, 200]$. You are welcome to try downscaling the rewards of REINFORCE as well.
- Normalizing the returns G_t over each episode by subtracting the mean and dividing by the standard deviation may improve the performance of REINFORCE.
- Likewise, batch normalization between layers can improve stability and convergence rate of both REINFORCE and A2C. Keras has a built-in batch normalization layer `https://keras.io/layers/normalization/`.
- Feel free to experiment with different policy architectures. Increasing the number of hidden units in earlier layers may improve performance.
- We recommend using a discount factor of $\gamma = 0.99$.
- Try out different learning rates. A good place to start is in the range [`1e-5`, `1e-3`]. Also, you may find that varying the actor and critic learning rates for A2C can help performance. There is no reason that the actor and critic must have the same learning rate.
- Policy gradient algorithms can be fairly noisy. You may have to run your code for several tens of thousand training episodes to see a consistent improvement for REINFORCE and A2C.
- Instead of training one episode at a time, you can try generating a fixed number of steps in the environment, possibly encompassing several episodes, and training on such a batch instead.

References

[1] Leemon Baird. Residual algorithms: Reinforcement learning with function approximation. In *Machine Learning Proceedings 1995*, pages 30–37. Elsevier, 1995.

[2] Vijay R. Konda and John N. Tsitsiklis. Actor-critic algorithms. In S. A. Solla, T. K. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems 12*, pages 1008–1014. MIT Press, 2000.

[3] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[6] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[7] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation. *arXiv e-prints*, page arXiv:1506.02438, Jun 2015.

[8] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[9] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. 2016.

[10] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

[11] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Machine Learning*, pages 229–256, 1992.