Homework 3: VI/PI, DQN, REINFORCE

CMU 10-703: Deep Reinforcement Learning (Fall 2020)

OUT: Wednesday, Oct. 14, 2020
DUE: Friday, Oct. 30, 2020 by 11:59pm ET

Instructions: START HERE

- **Collaboration policy:** You may work in groups of up to three 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/703website/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, zip all the code folders into a directory titled `<andrew_id>.zip` and upload it 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.

Introduction

In this assignment, you will implement value/policy iteration, DQN and REINFORCE and evaluate these algorithms on a variety of OpenAI Gym environments.

You may need additional compute resources for this assignment. Time estimates for training are given for problems 2 and 3. An AWS setup guide can be found at https://aws.amazon.com/ec2/getting-started/. We recommend using `p2-*` or `p3-*` GPU instances with the Deep Learning AMI that AWS provides with most dependencies pre-installed. Be sure to

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

monitor the usage cost, and always try to use Spot Instances over Dedicated machines to save on compute cost.

This is a challenging assignment. **Please start early!**

Installation instructions (Linux)

For this assignment, we recommend using **Python 3.6 and above**. 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
```

On **AWS Deep Learning instances**, the conda environments will have most of the dependencies and you may directly install `swig` and the requirements without having to create a virtualenv.

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

```
pip install box2d box2d-py
```

For additional installation instructions, see https://github.com/openai/gym.

Problem 0: Collaborators

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

Problem 1: Value Iteration & Policy Iteration (30 pts)

Problem 1.1: Contraction Mapping (3 pts)

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

1. If $f : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ has a fixed point, then it is a contraction mapping w.r.t. the euclidean norm.
2. Let S, ρ be a complete metric space, where S is the set and ρ is the metric. If a function $f : S \longrightarrow S$ is a contraction mapping then there exists a unique x^* such that $f(x^*) = x^*$.
3. Let $\{\pi_k\}$ be the sequence of of policies generated by the policy iteration algorithm. Then $F^{\pi_{k+1}} = F^{\pi_k}$ if and only if $F^{\pi_k} = F^{\pi_*}$, where F is the Bellman expectation backup operator.

VI/PI implementation (27 pts)

In this problem, you will implement value iteration and policy iteration. Throughout this problem, initialize the value functions as zero for all states and break ties in order of state numbering (the numbering is described further below).

We will be working with a different version of the OpenAI Gym environment `Deterministic-*-FrozenLake-v0`[2], defined in `hw3q1/lake_envs.py`. You can check `README.md` for specific coding instructions. Starter code is provided in `hw3q1/pi_vi.py`, with useful helper functions at the end of the file! It is recommended that you use Python 3.6+.

We have provided two different maps, a 4×4 map and a 8×8 map:

```
FHSF
FGHF
FHHF
FFFF
```

```
FFFFFSFF
FFFFFFFF
HHHHHHFF
FFFFFFFF
FFFFFFFF
FHFFFHHF
FHFFHFHH
FGFFFFFF
```

There are four different tile types: Start (S), Frozen (F), Hole (H), and Goal (G).

[2] https://gym.openai.com/envs/FrozenLake-v0

- The agent starts in the Start tile at the beginning of each episode.
- When the agent lands on a Frozen or Start tile, it receives 0 reward.
- When the agent lands on a Hole tile, it receives 0 reward and the episode ends.
- When the agent lands on the Goal tile, it receives +1 reward and the episode ends.

States are represented as integers numbered from left to right, top to bottom starting at zero. For example in a 4×4 map, the upper-left corner is state 0 and the bottom-right corner is state 15:

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{array}$$

Note: Be careful when implementing value iteration and policy evaluation. Keep in mind to make sure the reward function ($r(s, a, s')$) is correctly considered. Also, terminal states are slightly different. Think about the backup diagram for terminal states and how that will affect the Bellman equation.

In this section, we will use the **deterministic** versions of the FrozenLake environment. Answer the following questions for the maps `Deterministic-4x4-FrozenLake-v0` and `Deterministic-8x8-FrozenLake-v0`.

Problem 1.2: Synchronous Policy Iteration

1. (4 pts) For each domain, find the optimal policy using **synchronous policy iteration** (see Fig. 3). Specifically, you will implement `policy_iteration_sync()` in `hw3q1/pi_vi.py`, writing the policy evaluation steps in `evaluate_policy_sync()` and policy improvement steps in `improve_policy()`. Record (1) the number of policy improvement steps and (2) the total number of policy evaluation steps. Use a discount factor of $\gamma = 0.9$. Use a stopping tolerance of $\theta = 10^{-3}$ for the policy evaluation step.

Environment	**# Policy Improvement Steps**	**Total # Policy Evaluation Steps**
Deterministic-4x4		
Deterministic-8x8		

2. (2 pts) Show the optimal policy for the Deterministic-4x4 and 8x8 maps as grids of letters with "U", "D", "L", "R" representing the actions up, down, left, right respectively. See Figure 1 for an example of the 4x4 map. Helper: `display_policy_letters()`.

3. (2 pts) Find the value functions of the policies for these two domains. Plot each as a color image, where each square shows its value as a color. See Figure 2 for an example for the 4x4 domain. Helper function: `value_func_heatmap()`.

```
LLLL
RRRR
UUUU
DDDD
```

Figure 1: An example (deterministic) policy for a 4×4 map of the `FrozenLake-v0` environment. L, D, U, R represent the actions up, down, left, right respectively.

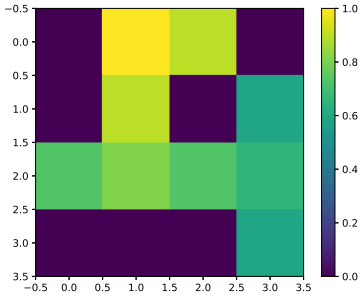

Figure 2: Example of value function color plot for a 4×4 map of the `FrozenLake-v0` environment. Make sure you include the color bar or some kind of key.

Problem 1.3: Synchronous Value Iteration

1. (3 pts) For both domains, find the optimal value function directly using **synchronous value iteration** (see Fig. 4). Specifically, you will implement `value_iteration_sync()` in `hw3q1/pi_vi.py`. Record the number of iterations it took to converge. Use $\gamma = 0.9$ Use a stopping tolerance of 10^{-3}.

Environment	# Iterations
Deterministic-4x4	
Deterministic-8x8	

2. (2 pts) Plot these two value functions as color images, where each square shows its value as a color. See Figure 2 for an example for the 4x4 domain.

3. (2 pts) Convert both optimal value functions to the optimal policies. Show each policy as a grid of letters with "U", "D", "L", "R" representing the actions up, down, left, right respectively. See Figure 1 for an example of the expected output for the 4x4 domain.

Notes on **Asynchronous v.s. Synchronous** (value iteration & policy iteration): The main difference between sync and async versions is that: whether all the updates are performed in-place (async) or not (sync). Take the synchronous value iteration for example, at some time step k, you would maintain two separate vectors V_k and V_{k+1} and perform updates of the following form inside the loop as described in Figure 4:

$$V_{k+1}(s) \longleftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V_k(s')].$$

For asynchronous updates, you don't need two copies of the vector as above.

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s)) \big[r + \gamma V(s')\big]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma V(s')\big]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Figure 3: Policy iteration, taken from Section 4.3 of Sutton & Barto's RL book (2018).

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\textit{terminal}) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma V(s')\big]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma V(s')\big]$

Figure 4: Value iteration, taken from Section 4.4 of Sutton & Barto's RL book (2018).

Problem 1.4: Asynchronous Policy Iteration

1. (4 pts) Implement **asynchronous policy iteration** using two heuristics:

   (a) The first heuristic is to sweep through the states in the order they are defined in the gym environment. Specifically, you will implement `policy_iteration_async_ordered()` in `hw3q1/pi_vi.py`, writing the policy evaluation step in `evaluate_policy_async_ordered()`.

(b) The second heuristic is to choose a random permutation of the states at each iteration and sweep through all of them. Specifically, you will implement `policy_iteration_async_randperm()` in `hw3q1/pi_vi.py`, writing the policy evaluation step in `evaluate_policy_async_randperm()`.

Fill in the table below with the results for Deterministic-8x8-FrozenLake-v0. Run one trial for the **first** ("async_ordered") heuristic. Run **ten trials** for the **second** ("async_randperm") heuristic and report the **average**. Use $\gamma = 0.9$. Use a stopping tolerance of 10^{-3}.

Heuristic	**Policy Improvement Steps**	**Total Policy Evaluation Steps**
Ordered		
Randperm		

Problem 1.5: Asynchronous Value Iteration

1. (4 pts) Implement **asynchronous value iteration** using two heuristics:

   (a) The first heuristic is to sweep through the states in the order they are defined in the gym environment. Specifically, you will implement `value_iteration_async_ordered()` in `hw3q1/pi_vi.py`.

   (b) The second heuristic is to choose a random permutation of the states at each iteration and sweep through all of them. Specifically, you will implement `value_iteration_async_randperm()` in `hw3q1/pi_vi.py`.

   Fill in the table below with the results for Deterministic-8x8-FrozenLake-v0. Run one trial for the **first** ("async_ordered") heuristic. Run **ten trials** for the **second** ("async_randperm") heuristic and report the **average**. Use $\gamma = 0.9$. Use a stopping tolerance of 10^{-3}.

Heuristic	**# Iterations**
Ordered	
Randperm	

2. (4 pts) Now, you can use a domain-specific heuristic for asynchronous value iteration (`value_iteration_async_custom()`) to beat the heuristics defined in Q1.5.1. Specifically, you will sweep through the entire state space ordered by Manhattan distance to goal.

   (a) Fill in the table below (use a stopping tolerance of 10^{-3}).

Env	**# Iterations**
Deterministic-4x4	
Deterministic-8x8	

   (b) In what cases would you expect this "goal distance" heuristic to perform best? Briefly explain why.

Problem 2: REINFORCE (30 pts)

In this section, you will implement episodic REINFORCE [7], a policy-gradient learning algorithm. Please write your code in `hw3q2/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

procedure REINFORCE
    Start with policy model π_θ
    repeat:
        Generate an episode $S_0, A_0, r_0, \ldots, S_{T-1}, A_{T-1}, r_{T-1}$ *following* $\pi_\theta(\cdot)$
        for t *from* $T-1$ *to* 0:
            $G_t = \sum_{k=t}^{T-1} \gamma^{k-t} r_k$
        $L(\theta) = -\frac{1}{T} \sum_{t=0}^{T-1} G_t \log \pi_\theta(A_t|S_t)$
        Optimize π_θ *using* $\nabla L(\theta)$
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 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/.
Note: Be sure to check out the "Guidelines on Implementation" section, especially for **episode generation**.

Toy environment: First test your implementation on a simple toy environment `BanditEnv` defined in `hw3q2/reinforce.py`. You will not be graded for this, testing on this environment serves as a sanity check for your implementation since the optimal policy will be learnt extremely fast. As it is a bandit set up, each 'episode' (pull of an arm/action) is of length one and the state is fixed. At convergence, your policy would get a reward of around 0 (will be close to 0 but not equal) consistently for 100 consecutive pulls/actions/episodes, which can be expected in around 15k episodes. Because the bandit arms have no effect on each other, setting $\gamma = 0$ would work well in this setting. On confirming that your implementation works as expected on the toy environment:

Train your implementation on the `LunarLander-v2` environment until convergence[3]. 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. Sometimes the agent learn to 'hover' in this environment, leading to long episodes causing a bottleneck in training. In case this happens, train up till the agent consistently reaches a reward of at least 160. Answer the following questions.

1. (8 pts) Describe your implementation, including the optimizer and any hyperparameters you used (learning rate, γ, layer sizes, activations and anything else that's relevant). Your description should be detailed enough that someone could reproduce your results.

2. (10 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 (return) on the y-axis with the standard deviation as error-bars against the number of training episodes on the x-axis. Describe 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?

   Hint: You can use matplotlib's `plt.errorbar()` or `plt.fill_between()` functions.

3. (12 pts) Implement REINFORCE with a baseline (Section 13.4, S&B). You can choose whatever baseline you like. Plot the learning curve in a manner similar to the previous sub-part. Describe your graph(s) and the learning behavior you observed, while making sure to answer these points:

   - According to S&B, can the baseline depend on the action? Can the baseline depend on the state?
   - What is your choice of a baseline, and why?
   - What differences do you notice between the training and plots for REINFORCE with and without a baseline. What could the reason be?

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

Problem 3: DQN (33 pts)

In this problem you will implement Q-learning, using tabular and learned representations for the Q-function.

Problem 3.1: 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 3.2: DQN Implementation (29 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.[4] Please write your code in the `hw3q3/dqn.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.

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

Runtime Estimation: To help you better manage your schedule, we provide you with reference runtime of DQN on MacBook Pro 2018. For Cartpole-v0, it takes 5 minutes to first reach a reward of 200 and around 70 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.

Implement a deep Q-network with experience replay. While the original DQN paper [3] 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.

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

Decaying ϵ may help with faster learning, though it is not necessary. For the replay buffer, you should use the experimental setup of [3] 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. As with Problem 2, you may first test your implementation on the `BanditEnv` for a quick check. Answer the following questions in your report:

1. (8 pts) Describe your implementation, including the optimizer, the neural network architecture and any hyperparameters (learning rate, γ, activations and anything else that's relevant) you used.
2. (11 pts) For each environment, plot the average cumulative test reward (return) throughout training.[5] 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.
3. (10 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.
4. (0 pts) Note: *This question* ***will not be graded****. However, visualizing a working policy can be instructive as well as fun.*
   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()`.

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

Quiz Questions (5 pt)

Propose five great quiz questions that cover the topics from this homework. Please submit your questions using this form:

https://forms.gle/RufYcPBUa88xBdDNA

Please submit this form five times, once per question. Try to avoid proposing questions that have already been proposed:

https://docs.google.com/spreadsheets/d/1L4v_pfX9eqa_H4ryuctisdC4W3D-bLHERdRqmOr9kq0/edit?usp=sharing

Extra (2 pt)

Feedback (1 pt): You can help the course staff improve the course by providing feedback. You will receive a point if you provide actionable feedback. What was the most confusing part of this homework, and what would have made it less confusing?

Time Spent (1 pt): How many hours did you spend working on this assignment? Your answer will not affect your grade.

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.

It's 2020, life's already a mess. A couple of points which may make it easier:

- **Episode generation**: In keras, model.predict() is considerably slower than `__call__` for single batch execution. Make sure you use the latter for episode generation. (https://www.tensorflow.org/api_docs/python/tf/keras/Model?hl=en#predict)
- (optional) Training progress logging: Training for Problems 2 and 3 will take a long time. An easy way to keep track of training progress is to use TensorBoard. TensorBoard can be used with Tensorflow, keras as well as PyTorch. This tutorial https://www.tensorflow.org/tensorboard/scalars_and_keras is on using TensorBoard with keras. We encourage you to use this extremely useful tool for Problems 2 and 3.

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. For example, in lines 5 and 6 of Algorithm 1 (REINFORCE) 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 50 minutes to one hour). On a NVIDIA GeForce GTX 1080 Ti GPU, it takes about eight hours to run 50K training episodes with our REINFORCE implementation.
- Normalizing the returns G_t over each episode by subtracting the mean and dividing by the standard deviation may improve the performance of REINFORCE. (Think why?)
- Likewise, if needed, batch normalization between layers can improve stability and convergence rate of both REINFORCE. 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 $\gamma = 0.99$, unless specified otherwise int the question.
- Try out different learning rates. A good place to start is in the range [`1e-5`, `1e-3`]
- 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.
- 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] 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.

[2] 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.

[3] 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.

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

[5] 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.

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

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