Homework 2: Value Iteration, Policy Iteration, and Q-Learning

CMU 10-703: Deep Reinforcement Learning (Fall 2019)

OUT: Sept. 20, 2019

DUE: Oct. 2, 2019 by 11:59pm

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 2." Additionally, upload your code the GradeScope assignment titled "Homework 2: 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:** Though this writeup looks long, some parts are quick questions that designed to give you a better picture of what we have learnt so far. We strongly suggest you **START EARLY** on problem 3.3 (DQN Implementation), which might takes a few hours to train your model.

[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: Basics & MDPs (27 pts)

For this problem, assume the MDP has finite state and action spaces. Let $V^\pi(s)$ be the value of state s under a policy π, which is the expected return when starting in state s at time t and following the policy π thereafter:

$$V^\pi(s) := \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \;\middle|\; s_t = s \right]$$

1. (6 pts) Assume a discount factor less than one: $\gamma < 1$. Show that for all policies π and states $s \in S$, the value function $V^\pi(s)$ is well-defined, *i.e.*. That is, show that (i) the infinite sum $\sum_{k=0}^{\infty} \gamma^k r_{t+1+k}$ converges even if the episode never terminates; and (ii) the outer expectation $\mathbb{E}_\pi[\cdot]$ is bounded. Hint: $\sum_{k=0}^{k=\infty} \gamma^k = \frac{1}{1-\gamma}$.

   Solution

2. (6 pts) For each of the following three statements, answer "True" or "False." In addition, explain why in 1-2 sentences. As a reminder, we are considering MDPs with finite state and action spaces.

   (a) For every MDP, there exists a unique optimal policy.

   (b) If an MDP has multiple optimal policies, then each has the same value function.

   (c) There are some fully-observed MDPs on which stochastic policies can obtain higher reward than deterministic policies.

   Solution

3. (9 pts) Consider three MDPs with two states ($\{s_1, s_2\}$) and two actions ($\{a_1, a_2\}$).

   `MDP 1:`

   Transition function:

   - $P(s_1|a_1, s_1) = 1$
   - $P(s_2|a_2, s_1) = 1$
   - $P(s_2|a_1, s_2) = 1$
   - $P(s_2|a_2, s_2) = 1$
   - $P(s'|a, s) = 0$ for all other s, a, s' triples

Reward function:

- $R(s_1, a_2, s_2) = 1$
- $R(s, a, s') = 0$ for all other s, a, s' triples

`MDP 2:`

Transition function is the same as MDP 1.

Reward function:

- $R(s_2, a_1, s_2) = 1$
- $R(s_2, a_2, s_2) = 1$
- $R(s, a, s') = 0$ for all other s, a, s' triples

`MDP 3:`

Transition function:

- $P(s_1|a_1, s_1) = 1$
- $P(s_2|a_2, s_1) = 1$
- $P(s_1|a_1, s_2) = 1$
- $P(s_2|a_2, s_2) = 1$
- $P(s'|a, s) = 0$ for all other s, a, s' triples

Reward function:

- $R(s_1, a_1, s_1) = 1$
- $R(s_2, a_2, s_2) = 1$
- $R(s, a, s') = 0$ for all other s, a, s' triples

(a) (6 pts) Assume a discount factor $\gamma = 1$. For each of three MDPs above, does there exists a policy whose value function is infinite for s_1. If the answer is yes, please describe the set of all such policies; otherwise, please describe the set of all policies whose value function is maximum for s_1 across all policies.

Solution

(b) (3 pts) Now assume a discount factor of $1 - \epsilon$, where ϵ is some small positive value. For each of three MDPs above, describe the set of optimal policies.

Solution

4. (6 pts) The discount factor γ can change the optimal policy. Describe an MDP and two policies, each of which is optimal for a different value of γ.

Solution

Problem 2: Value Iteration & Policy Iteration (26+4)

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 (as described further below).

We will be working with a different version of the OpenAI Gym environment `Deterministic-*-FrozenLake-v0`[2], defined in `deeprl_hw2q2/lake_envs.py`. You can check `README.md` for specific coding instructions. Starter code is provided in `deeprl_hw2q2/rl.py`, with useful HELPER functions at the end of the file! You may use either Python 2.7 or 3.

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).

- 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:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Note: Be careful when implementing value iteration and policy evaluation. Keep in mind that in this environment, the reward function depends on the current state, the current action, and the **next state**. 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`.

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

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 `deeprl_hw2q2/rl.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()`.

Solution

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()`.

Solution

```
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.

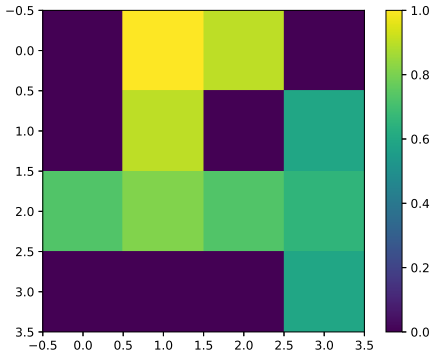

Figure 2: Example of value function color plot for a 4×4 map of the `FrozenLake-v0` environment. Make sure you include the color bar or some kind of key.

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 $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r \,|\, s, \pi(s)) \big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   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 $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \,|\, s, a) \big[r + \gamma V(s')\big]$
      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: Synchronous 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 $V(\mathit{terminal}) = 0$

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

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

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

- Notes on **Asynchronous v.s. Synchronous** (value iteration & polocy 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.

4. (4 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 `deeprl_hw2q2/rl.py`. Record the number of iterations it took to converge. Use $\gamma = 0.9$ Use a stopping tolerance of 10^{-3}.

Solution

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

5. (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.

Solution

6. (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.

Solution

7. (2 pts) For both of the two domains, measure the average run-time for your two algorithms (do not include as part of your answer). Which algorithm was faster in your case (policy iteration or value iteration)? Which would you expect to be faster? Would you generally expect any differences in the value function?

Solution

8. (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 `deeprl_hw2q2/rl.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 `deeprl_hw2q2/rl.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}.

Solution

Heuristic	Policy Improvement Steps	Total Policy Evaluation Steps
Ordered		
Randperm		

9. (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 `deeprl_hw2q2/rl.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 `deeprl_hw2q2/rl.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}.

Solution

Heuristic	# Iterations
Ordered	
Randperm	

10. (**4 pts, ptional**) Now, you can use a domain-specific heuristic for asynchronous value iteration (`value_iteration_async_custom()`) to beat the heuristics defined in Q2.9. 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}).

    Solution

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

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

    Solution

Problem 3: 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 (3.3c).

Problem 3.1: Relations among Q & V & C (13 pts)

The objective of this question is to understand different Bellman equations, their strengths and limitations. Consider the Bellman Equation for 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 equation, then we obtain a repeating structure:

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

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:

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

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

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

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 equation) for $C(s, a)$.

Solution

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 $R(s, a)$?

Solution

Problem 3.2: 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.

Solution

Problem 3.3: 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.[3] 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 reference runtime of DQN on MacBook Pro 2018. For Cartploe-v0, it takes 5 minuts 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 [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.

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. Answer following questions in your report:

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

(a) (20 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.[4] 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) 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 [4].

(b) (20 pts) Dueling DQN, as described in [5].

(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 [2] 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 [3], may clarify a few details so it may be worth to read selectively if questions remain after reading [2].

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

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

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 [2, 3]. For the most part, we strongly suggest you to follow the experimental setup described in each of the papers. [2, 3] 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.

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., replay buffer, Tensorflow or Keras model definition, model updater, model runner, exploration schedule, learning rate schedule, ...)

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.

We provide some code templates that you can use if you wish. Contrary to the previous assignment, abiding to the function signatures defined in **these templates is not mandatory you can write your code from scratch if you wish**.

References

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

[2] 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.

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

[5] 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.