INF8250AE(Fall 2024) : Reinforcement Learning — Given on: Oct 28

Assignment #3 — Due on : Nov 13, 10 pm

- The primary objective of this assignment is to evaluate your comprehension and application of Deep Q-learning. Through this task, you will demonstrate your ability to understand the core concepts, implement the algorithms, and analyze their performance.
- This is an individual assignment. You are not allowed to discuss the problems with other students.
- Make sure to write your code only in the `.py` files provided. Avoid creating new files. Do not rename the files or functions, as it will interfere with the autograder's ability to evaluate your work correctly. Also, do not change the input or output structure of the functions.
- When Submitting to GradeScope, be sure to submit:
  1. A `'.zip'` file containing all your Python codes (in `.py` format) to the `'Assignment 3 - Code'` section on Gradescope.
  2. A `'pdf'` file that contains your answers to the questions and generated plots to the `'Assignment 3 - Report'` entry.
- Part of this assignment will be auto-graded by Gradescope. You can use it as immediate feedback to improve your answers. You can resubmit as many times as you want. We provide some tests that you can use to check that your code will be executed properly on Gradescope. These are not meant to check the correctness of your answers. We encourage you to write your own tests for this.
- You cannot use ChatGPT or any other code assistance tool for the programming part; however, if you use ChatGPT to edit the grammar in your report, you have to explicitly state it in the report.
- If you have questions regarding the assignment, you can ask for clarifications in Piazza. You should use the corresponding tag for this assignment.

DQN

Deep Q-Network (DQN) is a value-based deep reinforcement learning algorithm that learns the Q-function of the optimal policy directly via the Bellman optimality equation. Deep Q-Network parameterizes Q-function via a deep neural network. Specifically, denote $Q_\theta(s, a)$ to be the Q-function in an MDP implemented as a fully-connected neural network with parameters θ, i.e. optimization of this Q-function should (in practice) converge to the optimal agent. More specifically, Q_θ is a neural network that accepts s as an input vector and outputs a vector of size equal to the number of possible actions. Each value of such vector will be the corresponding Q-function. We will write $Q_\theta(s)$ to define such a vector of Q-values (each associated with each action in MDP) and $Q_\theta(s)[a]$ to refer to a concrete Q-value.

DQN relies on two key components on top of standard Q-learning:

1. **Experience replay.** Initialize the First-In First-Out (FIFO) replay buffer $\mathcal{D}$ of a fixed size N. The idea is to store experience tuples $(s_t, a_t, r_t, d_t, s_{t+1})$, where s_t, a_t, r_t, d_t are the state, action, reward, and episode termination flag in MDP at timestep t, respectively. s_{t+1} is the state of MDP at timestep $t+1$. To simplify the implementation, all `jax`-based environments implement automatic resetting, therefore, whenever d_t is one, s_{t+1} becomes the initial state of a new episode. Note that every state the agent visits gets into the replay buffer two times: first being the "next" state s_{t+1}, and next being the "current" state s_t.

   Each FIFO replay buffer implements two main functions:

   - `buffer.add_transition` this function takes one tuple $(s_t, a_t, r_t, d_t, s_{t+1})$, where each tuple element is a `jax.numpy` array and inserts that into the buffer storage. Note that since this buffer is FIFO, once its size reaches N, each new addition erases the current earliest added element.
   - `buffer.sample_transition` this function randomly selects one transition of experience from the replay buffer. Since this is not a deterministic function, our `jax` implementation will also accept the random numbers generator seed.

2. **Target network.** The authors of DQN found that Q-learning with function approximation is not stable since the Bellman target value is updated too often to produce convergent Q-learning in practice. Therefore, they proposed to use the second Deep Q-Network, which they call "target network" (or "DQN with target parameters"). The idea is to use the target network only to define the Bellman target with the DQN is trained towards and to update target DQN periodically with the current DQN. At the beginning of training, target network is simply a copy of DQN but once DQN starts to optimize its parameters, they become different. Over the optimization, we simply periodically set the weights of the target network to be the same as the current DQN. Note that the target network should be updated neither too rare not too often. If it is updated too often, the training becomes unstable (Due to the reason mentioned above). If it is too rare the network will not converge to the optimal Q-function because the Bellman target will not be an optimal Q-value. Due to this reason, in practice we have to empirically find the value that works well.

Deep Q-Network training is defined in the following way. For simplicity we do not specify all hyperparameters (such as update frequencies), as they can be found in the starting code we provide:

1. Initialize DQN, DQN parameters θ, DQN target parameters $\hat{\theta}$ (being equal to θ), and an empty FIFO replay buffer $\mathcal{D}$. Reset the environment to obtain s_t where $t = 0$.

2. Sample n consecutive environment steps following ε-greedy strategy, i.e. collect a set $\{\tau_{t:t+n}\}$ such that $\tau_t = (s_t, a_t, r_t, d_t, s'_t)$ where $a_t \sim \texttt{epsilon\_greedy}(Q_\theta(s_t), \varepsilon_t)$; and $s'_t, r_t, d_t = \texttt{MDP}(s_t, a_t)$. Note that ε depends on the timestep, i.e. we change the amount of exploration over time.

3. Put $\{\tau_{t:t+n}\}$ into $\mathcal{D}$.

4. Sample a training batch of transitions (of size B) $\{\tau_j\}_{j=1}^{B}, \tau_j \sim \mathcal{D}$.

5. Perform one training step of DQN parameters using the following loss function:

$$L(\theta) = \sum_{j=1}^{B} \left(Q_\theta(s_t)[a_t] - \left[r_j + \gamma \times (1 - d_j) \times \max_a Q_{\hat{\theta}}(s'_j) \right] \right)^2$$

   here $\tau_j = (s_j, a_j, r_j, d_j, s'_j)$, γ is a discount factor used in training.

6. Repeat steps (2-5) m times, then perform the update of target parameters (i.e. set $\hat{\theta} \leftarrow \theta$).

7. Repeat (6) until a total budget of T environment steps ever put into $\mathcal{D}$ is reached.

This algorithm definition is slightly different to the one of Q-learning but it the most closely resembles to real implementations of DQN in practice.

In this task, we provide the main training loop for you that is already implemeted in `jax`. Your goal in this assignment is to implement the key functions of the FIFO replay buffer and the DQN agent in `jax` and make sure your training converges to the expected return, close to an optimal agent. We guarantee every task in this assignment can be done in a few (less than 6) short lines of code in `jax`. Also we guarantee that all `jax`-specific functions you need to use in this assignment were discussed in the `jax` tutorial of this course and its accompanying colab notebook.

1 Experience replay buffer (FIFO)

1. Implement Experience replay buffer (FIFO) (10 points)
   (a) Insert experience (5 points). You need to implement a function in `jax` that inserts one new transition into the replay buffer. Specifically, the function `add_transition` in `buffer.py`. Please keep in mind corner cases e.g. when the buffer is overfilled with new experience.

(b) Sample Experience (5 points). You need to implement a function in `jax` that samples experience tuple. Specifically, the function `sample_transition` in `buffer.py`. Please keep in mind that the buffer might not be full at sample time.

2 Vanilla DQN

2. Vanilla DQN Implementation (50 points). For every function we request you to implement, all the input types, shapes, and data types are described in the documentation of the corresponding function. Each function you implement should be able to be decorated in `jax.jit` and `jax.vmap` transformations and produce valid outputs when decorated properly. Each function you implement should be pure. Note that you can easily test these properties yourself by simply running the training of your agent (see the next section).

   (a) *Neural network definition (5 points).* You need to implement the `__call__` function in the `DQN` class of `model.py` to define an MLP neural network. You cannot use `jax.nn.MLP`, `eqx.nn.MLP` or equivalent pre-existing MLP. Also you can only use submodules of `flax.linen` in this function. You are free to choose non-linearities of this neural net, the number of hidden layers and their sizes but we recommend to use the network with the number of trainable parameters between 2000 and 10000.

   (b) *jax-epsilon greedy (15 points).* Implement the `select_action` function in `model.py` to perform action selection via epsilon-greedy strategy.

   (c) *DQN loss with target value (15 points).* Implement the `compute_loss` function in `model.py`. This function should compute the loss function of Deep Q-Network as defined in step 5 of the algorithm in Section 1 of this document.

   (d) *target network update (5 points).* Implement the `update_target` function in `model.py`. This function should update the target parameters with current parameters. Note that the implementation should work with any architecture of the neural network, not just the one you defined in previous task. If you are not sure, please refer to the jax tutorial, do not overthink it.

   (e) initialization (10 points). Implement the `initialize_agent_state` function in `model.py`. This function should initialize the training state of the DQN agent. That is, it should create the neural network inference function, initialize parameters for the DQN agent and its target network, and create the optimizer. For the optimizer, please use the `optax` library considered in the tutorial.

3. Vanilla DQN Training (10 points).

Environment The Cartpole-v1 environment is a simple control task where the agent is a cart on which an inverted pendulum has been attached. Its goal is to balance the pendulum so that it doesn't fall down. The agent can only move left or right to balance the pendulum. The observation space of the agent is a four-dimension tuple defined

as $(x, \dot{x}, \theta, \dot{\theta})$ where x represents the pendulum's position, $\dot{x}$ represents its velocity, θ represents the angle between the pendulum and the vertical axis and $\dot{\theta}$ represents the pendulum's angular velocity. The action space is a discrete space with two possible values (0 and 1), respectively to push the cart to the left and right.

It is important to note that unlike the regular `gymnasium` library, `gymnax` automatically resets the state of the environment when a transition finishes or is truncated. Therefore, you only need to use `env.reset()` when you explicitly need it.

(a) Train DQN (10 points). Use the notebook `model_playground.ipynb` to train your DQN agent in the CartPole-v1 environment. To pass this task, you need to obtain the score above 400 after 250000 environment steps (the score is averaged over > 10 random seeds for training). Note that valid implementation of all function in the previous two section should achieve that since we provide the hyperparameters that are close to the optimal ones. The only aspect affecting the performance is the network architecture so we recommend you to iterate with a few experiments on that. When running in colab, one training run takes around one minute including the time for compilation.

Please include the plot produced by `model_playground.ipynb` into the report. And describe any changes you made to the algorithm or hyperparameters (if you did any) and upload them (your code) alongside your submission (we will run your code to make sure it produces the performance you report).

(b) Improve DQN (10 BONUS points). Note that the points for this task are counted ABOVE the maximum points you can get for the whole assignment. Therefore, we recommend you to work on it only once you finished the rest of the assignment.

Your goal is to change any part of the code (i.e. anything in `buffer.py`, `model.py`, `trainer.py`) to get the score above 490 after 250000 environment steps (averaged over > 10 seeds chosen arbitrarily not by you). The only part of the code you cannot modify for this task is the function `eval_agent` in `trainer.py`.

3 Double DQN

4. Double DQN Implementation (20 points). The same requirement as in section 2 to the functions you implement applies here (`jax.jit`/`jax.vmap` ability) and purity.

(a) *Double DQN implementation (10 points).* Implement the `compute_loss_double_dqn` function in `model.py`. This function has the same signature as in `compute_loss` but it should perform the double DQN loss.

The idea behind Double DQN is to reduce the overestimation bias introduced by maximization of Q-values in DQN loss. Double-DQN implements that by separately estimating the target action (from current parameters) and target value from target parameters. That is, to implement Double-DQN you need to change $\max_a Q_{\hat{\theta}}(s'_j)$ to $Q_{\hat{\theta}}(s'_j)[\arg\max_a Q_\theta(s'_j)]$ in the notation of the algorithm above.

(b) *Double DQN training (10 points).* Train Double DQN using the code in `model_playground.ipynb` and compare it with vanilla DQN. Show how double target estimation improves performance (empirically). Any, even a small improvement of the average return after 250000 steps will work as a valid solution. Include plots into your report and any modifications you made to the algorithm and upload them (your code) alongside your submission (we will run your code to make sure it produces the performance you report).