Assignment 3: Q-Learning

In this assignment you will implement deep Q-networks (DQN) [1], a Q-learning algorithm that leverages deep neural networks, to play Atari games. The template code is available at:

https://github.com/xbpeng/rl_assignments

Installation instructions are provided in `README.md`. All of the files that need to be changed in this assignment are located in the `a3/` directory. No files outside of this directory should be modified. Locations where code needs to be modified are labeled `TODO`.

1 Q-Learning

Q-learning is a family of reinforcement learning algorithms that solves an MDP by learning a Q-function $Q(\mathbf{s}, \mathbf{a})$, instead of directly learning a policy $\pi(\mathbf{a}|\mathbf{s})$. The Q-function Q^π provides an estimate of the expected return of performing an action $\mathbf{a}$ in state $\mathbf{s}$, and following a policy π for all future timesteps,

$$Q^\pi(\mathbf{s}, \mathbf{a}) = \mathbb{E}_{\tau \sim p(\tau|\pi, \mathbf{s}_0 = \mathbf{s}, \mathbf{a}_0 = \mathbf{a})} \left[\sum_t \gamma^t r_t \right]. \tag{1}$$

The optimal Q-function for a given MDP can be learned using an iterative fixed-point method, where at each iteration k, a new Q-function Q^{k+1} is constructed via the Bellman update by bootstrapping from the current Q-function Q^k:

$$Q^{k+1}(\mathbf{s}, \mathbf{a}) = \mathbb{E}_{\mathbf{s}' \sim p(\mathbf{s}'|\mathbf{s}, \mathbf{a})} \left[r(\mathbf{s}, \mathbf{a}, \mathbf{s}') + \gamma \max_{\mathbf{a}'} Q^k(\mathbf{s}', \mathbf{a}') \right], \tag{2}$$

where $r(\mathbf{s}, \mathbf{a}, \mathbf{s}')$ is the reward function, $\gamma \in [0, 1]$ is a discount factor, and $p(\mathbf{s}'|\mathbf{s}, \mathbf{a})$ represents the dynamics of the environment.

A policy can be recovered from a given Q-function by selecting the action with the maximum predicted Q-value at a given state:

$$\pi^k(\mathbf{a}|\mathbf{s}) = \begin{cases} 1 & \text{if } \mathbf{a} = \arg\max_{\mathbf{a}'} Q^k(\mathbf{s}, \mathbf{a}') \\ 0 & \text{otherwise} \end{cases}. \tag{3}$$

This arg-max procedure returns a deterministic policy that selects actions greedily according to the Q-function. If Q-function is the optimal Q-function Q^*, then the resulting arg-max policy will be an optimal policy π^*. However, if the Q-function is not optimal, then a deterministic greedy policy will often lead to insufficient exploration during the learning process. ϵ-greedy exploration is a simple strategy that mitigate this exploration issue by constructing a stochastic policy

$$\pi^k(\mathbf{a}|\mathbf{s}) = \begin{cases} 1 - \epsilon & \text{if } \mathbf{a} = \arg\max_{\mathbf{a}'} Q^k(\mathbf{s}, \mathbf{a}') \\ \epsilon & \text{otherwise} \end{cases}, \tag{4}$$

which greedily select the best action according to the current Q-function with probability $1-\epsilon$, and selects a random action with probability ϵ. This methods helps to ensure that the agent can explore new actions and potentially discover more optimal strategies during the training process. Pseudo-code for Q-learning is available in Algorithm 1.

ALGORITHM 1: Q-Learning

$Q^0 \leftarrow$ initialize Q-function
$\mathcal{D} \leftarrow \{\emptyset\}$ initialize dataset

for iteration $k = 0, ..., n-1$ **do**
  Sample trajectory τ according to $Q^k(\mathbf{s}, \mathbf{a})$
  Add transitions to dataset $\mathcal{D} = \mathcal{D} \cup \{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i)\}$

  Calculate target values for each sample i:
  $y_i = r_i + \gamma \max_{\mathbf{a}'} Q^k(\mathbf{s}'_i, \mathbf{a}')$

  Update Q-function:
  $Q^{k+1} = \arg\min_Q \ \mathbb{E}_{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{r}_i, \mathbf{s}'_i) \sim \mathcal{D}} \left[(y_i - Q(\mathbf{s}_i, \mathbf{a}_i))^2 \right]$
end for

return Q^n

In this assignment, you will be implementing the deep Q-networks (DQN) algorithm to play Atari games [1]. To run DQN on the `Pong` task, use the following command:

```
python run.py --mode train \
--env_config data/envs/atari_pong.yaml \
--agent_config a3/atari_pong_dqn_agent.yaml \
--log_file output/log.txt \
--out_model_file output/model.pt \
--max_samples 3000000 \
--visualize
```

`--visualize` should be disabled for faster training. If you have a GPU that supports Cuda, then `--device cuda:0` can be used to enable Cuda during training, which can be much faster. Once a model has been trained, you can load a checkpoint and test the model with the following command:

```
python run.py --mode test \
--env_config data/envs/atari_pong.yaml \
--agent_config a3/atari_pong_dqn_agent.yaml \
--model_file output/model.pt \
--test_episodes 20 \
--visualize
```

1.1 Epsilon-Greedy Exploration

In `a3/dqn_agent.py`, implement the `_get_exp_prob()` method, which calculates the probability ϵ of selecting a random action in ϵ-greedy exploration. Since the Q-function is randomly initialized at the start of training, the exploration probability typically starts with a high value (e.g. $\epsilon = 1$), and then decreases to a low value (e.g. $\epsilon = 0.1$) over the course of training as the Q-function improves. The exploration probability starts with a value of ϵ_{beg}, and then linearly annealed to ϵ_{end} over the course of $n_{\max}$ samples,

$$\epsilon(n) = (1 - l)\epsilon_{\text{beg}} + l\epsilon_{\text{end}}, \qquad l = \text{clip}\left(\frac{n}{n_{\max}}, 0, 1\right), \tag{5}$$

where n denotes the total number of samples collected for training so far. The initial exploration probability ϵ_{beg} is given by `self._exp_prob_beg`, and the final probability ϵ_{end} is given by `self._exp_prob_end`. The total number of samples $n_{\max}$ for annealing the exploration probability is specified by `self._exp_anneal_samples`, and the number of samples n collected so far is recorded in `self._sample_count`. The output should be a scalar value corresponding to the probability of selecting a random action.

1.2 Action Sampling

In `a3/dqn_agent.py`, implement the `_sample_action(qs)` method, which samples actions according to the Q-values of each action. The input consists of a tensor of `qs`, which contains the predicted Q-values of each action. Implement ϵ-greedy exploration (Equation 4), where the probability of sampling a random action ϵ is specified by self._get_exp_prob(). With probability $1 - \epsilon$, greedily select the action with the highest Q-value. With probability ϵ, select a random action uniformly from the set of possible actions. The output `a` should be a tensor containing the index of the selected action.

1.3 Target Values

In `a3/dqn_agent.py`, implement the `_compute_tar_vals(r, norm_next_obs, done)` method, which calculates target values for updating the Q-function. The inputs consist of a tensor of rewards `r`, normalized observations at the next timestep `norm_next_obs`, and done flags `done` indicating if a timestep is the last timestep of an episode. The target value y_i is then calculated acoording to

$$y_i = r_i + \gamma(1 - \text{done}_i) \max_{\mathbf{a}'_i} Q^{\text{tar}}(\mathbf{s}'_i, \mathbf{a}'_i). \tag{6}$$

If a sample corresponds to the last timestep of an episode (i.e. $\text{done}_i = 1$), then the Q-value of the next timestep should be set to 0.

One of the innovations of DQN is the use of a target model Q^{tar} to calculate the target values [1], instead of directly calculating target values using the current Q-function. The target model is a delayed version of the Q-function, which is a copy of the parameters of the main Q-function from a number of iterations ago. Q^{tar} is kept fixed for a number of iteration, before being updated with a copy of the parameters from the latest Q-function. By keeping Q^{tar} fixed for a number of iterations, the target model provides more stable target values for updating the main Q-function. The target model is given by `self._tar_model`, and the main model is given by `self._model`. The Q-function can be queried by using the method `eval_q(norm_obs)`. The output `tar_vals` should be a tensor containing the target values for each sample.

1.4 Loss Calculation

In `a3/dqn_agent.py`, implement the `_compute_q_loss(norm_obs, a, tar_vals)` method, which compute the loss for updating the Q-function. The input consists of a tensor of normalized observations `norm_obs`, a tensor containing the indices of discrete actions selected at each timestep `a`, and target values for each timestep `tar_vals`. The output `loss` should be a scalar tensor containing the loss for updating the Q-function. The loss should be calculated as the mean squared-error between the target values and the outputs of the Q-function

$$l(Q) = \mathbb{E}_{\mathbf{s}_i,\mathbf{a}_i,y_i \sim \mathcal{D}} \left[\left(y_i - Q(\mathbf{s}_i, \mathbf{a}_i) \right)^2 \right]. \tag{7}$$

Note, the loss should only be applied to the main model `self._model`, and should not be applied to the target model `self._tar_model`.

1.5 Target Model Update

In `a3/dqn_agent.py`, implement the `_sync_tar_model()` method, which updates the target model by copying the parameters from the main model. The main model is given by `self._model`, and the target model is given by `self._tar_model`. This method is used to periodically synchronize the parameters of the target model and the main model. `self._model.parameters()` can be used to retrieve a list of tensors containing the parameters of a model.

1.6 Tasks

Train DQN policies to play two Atari games, `Pong` and `Breakout`. The two games can be specified using the environment config files `data/envs/atari_pong.yaml` and `data/envs/atari_breakout.yaml`. The correspnoding agent config files are `a3/atari_pong_dqn_agent.yaml` and `a3/atari_breakout_dqn_agent.yaml`. Train each model for at least 3 million timesteps. The default hyperparameters in the agent config files should already be fairly effective. However, depending on your implementation details, some additional tuning might be required. Tune the hyperparameters in the agent config files so that the policies reach a test return of at least 19 for `Pong` and 40 for `Breakout`. Plot a learning curve of the test return of each model using the plotting script `tools/plot_log/plot_log.py`.

2 Bonus

1 bonus point will be award to the submission that achieves the best performance on the `Breakout` tasks using DQN. To improve the performance of DQN, you can do more extensive tuning of the hyperparameters, as well as make any changes to the algorithm in `dqn_agent.py`. Submit any modifications made to the DQN algorithm in a separate file `bonus_dqn_agent.py`, hyperparameters `bonus_dqn_agent.yaml`, a checkpoint of the trained model `bonus_dqn_breakout_model.pt`, a training log `bonus_dqn_breakout_log.txt`, a learning curve `bonus_dqn_breakout_curve.png`, and a text file `bonus.txt` detailing the modifications you made.

Submission

Your submission should contain the following files:

- `dqn_agent.py`: code changes.
- `atari_dqn_agent.yaml`: tuned hyperparameters.
- `atari_pong_dqn_model.pt`: trained model for the `Pong` task.
- `atari_breakout_dqn_model.pt`: trained model for the `Breakout` task.
- `atari_pong_dqn_log.txt`: training log for the `Pong` task.
- `atari_breakout_dqn_log.txt`: training log for the `Breakout` task.
- `atari_pong_dqn_curve.png`: image of the learning curve for the `Pong` task.
- `atari_breakout_dqn_curve.png`: image of the learning curve for the `Breakout` task.

- `bonus_dqn_agent.py`: modified version of the DQN agent for the bonus component.
- `bonus_dqn_agent.yaml`: hyperparameters for the bonus component.
- `bonus_dqn_breakout_model.pt`: trained `Breakout` model for the bonus component.
- `bonus_dqn_breakout_log.txt`: training log for the bonus component.
- `bonus_dqn_breakout_curve.png`: image of the learning curve for the bonus component.
- `bonus.txt`: a text file detailing any modifications you made in `bonus_dqn_agent.py` for improving the performance of the algorithm.

All files should be stored in a directory named `a3`, and then zip the directory for submission. Do not add any additional subdirectories.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015.