CSCD84: Artificial Intelligence
Assignment 2: Adversarial Games, PGM, and RL
Due Monday 11:59PM, Feb 24, 2025

Changes

Feb. 12: Q 1.3.3, in the finite-horizon case, the summation should go from $t = 0, \ldots, T-1$ rather than $t = 0, \ldots, T$

1 Written Component

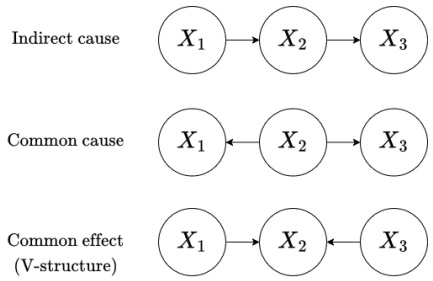
1.1 Adversarial Games

1. Consider an arbitrary game tree. Each leaf node corresponds to the utility for each player (e.g. in a two-player game, $U(A) = -U(B)$). Does the minimax algorithm change its optimal strategy if we only consider the sign of the utility, $U(A)$ (or $U(B)$), rather than the actual utility? If so, provide an example where the optimal strategy changes. Otherwise, explain why it does not change.

2. How would you generalize a two-player game into a three-player game? Give an example game tree involving three players—it must have at least one turn per player. Then, describe how the minimax algorithm changes and show the optimal strategy for your example game tree.

   Hint: The utility function is going to be represented as a 3-tuple, where entry i corresponds to the utility of player i.

1.2 Probabilistic Graphical Model



A Bayes' net can often be decomposed into triplets, which is helpful for determining whether two variables in a Bayes' net is (conditionally) independent. There are three types of triplets: indirect cause, common cause, and common effect. Answer the following questions:

1. Show that in a common-cause structure, X_1 and X_3 are conditionally independent given X_2. In other words, $\mathbb{P}(X_1, X_3|X_2) = \mathbb{P}(X_1|X_2)\mathbb{P}(X_3|X_2)$.

2. Show that in a common-effect structure, X_1 and X_3 are independent. In other words, $\mathbb{P}(X_1, X_3) = \mathbb{P}(X_1)\mathbb{P}(X_3)$.

3. Consider the indirect-cause structure and the query: $\arg\max_{X_1,X_2,X_3} \mathbb{P}(X_1, X_2, X_3)$. Use variable elimination (VE) with ordering X_1, X_2, X_3 to perform inference.

   Hint: The algorithm is similar to computing the posterior—the main difference is that introduced factors will use $\arg\max_{x_i}$ rather than $\sum_{x_i}$.

1.3 Reinforcement Learning

1. Fix a policy $\pi : \mathcal{S} \to \Delta(\mathcal{A})$. Recall that to evaluate the value function of the policy, V^π, we iteratively apply the Bellman expected operator $T^\pi v = r + \gamma P^\pi v$ to any vector $v \in \mathbb{R}^{|\mathcal{S}|}$. We indicated that this results in V^π and this is due to γ-contraction. Prove that T^π is indeed a γ-contraction with respect to the max-norm, where $\gamma \in [0, 1)$. That is, for any $u, v \in \mathbb{R}^{|\mathcal{S}|}$, we have that

   $$\|T^\pi u - T^\pi v\|_\infty \leq \gamma \|u - v\|_\infty.$$

   Consequently, by Banach's fixed-point theorem, we can show that continually applying the Bellman expected operator on a vector $v \in \mathbb{R}^{|\mathcal{S}|}$ will converge to V^π.

   Hint 1: The max-norm is defined to be $\|v\|_\infty = \max_{i \in d} |v_i|$ for any $v \in \mathbb{R}^d$.

   Hint 2: The max-norm satisfies triangle inequality, that is: $\|a + b\|_\infty \leq \|a\|_\infty + \|b\|_\infty$.

2. Let's now consider Markov decision process (MDP) with fixed horizon (i.e. all trajectories are of length T). Here a trajectory is defined as $\tau = (S_0, A_0, R_0, S_1, A_1, R_1, \ldots, S_{T-1}, A_{T-1}, R_{T-1}, S_T)$. We can also write the probability of a trajectory τ given a policy π as

   $$\mathbb{P}^\pi(\tau) = \mu(S_0) \prod_{t=0}^{T-1} P(S_{t+1}|S_t, A_t)\pi(A_t|S_t).$$

   Draw the probabilistic graphical model (PGM) of a fixed-horizon MDP with $T = 3$.

3. In lecture we showed that the policy gradient can be writen as

   $$\nabla_\pi J(\pi) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t G_{t:\infty}(\tau) \nabla_\pi \log \pi(A_t|S_t)\right],$$

   where $G_{a:b}(\tau) = \sum_{h=a}^{b} \gamma^{h-a} r(S_h, A_h)$ is the (random) return from timesteps a to b, with $0 \leq a \leq b$. Show that for a fixed-horizon MDP of length T with $\gamma = 1$, we can similar write the policy gradient as

   $$\nabla_\pi J(\pi) = \mathbb{E}\left[\sum_{t=0}^{T-1} G_{t:T-1}(\tau) \nabla_\pi \log \pi(A_t|S_t)\right].$$

   Note that you will be implementing this in the coding component. In practice, the expectation is approximated using (Monte-Carlo) samples. With N sampled trajectories, we can write:

   $$\nabla_\pi J(\pi) \approx \frac{1}{N} \sum_{n=1}^{N} \left[\sum_{t=0}^{T-1} G^{(n)}_{t:T-1}(\tau) \nabla_\pi \log \pi(A^{(n)}_t|S^{(n)}_t)\right],$$

   where the superscript $(\cdot)^{(n)}$ indicates the n'th sampled trajectory. From this perspective, $G^{(n)}_{t:T-1}$ really is also approximating the Q-function $Q^\pi(S^{(n)}_t, A^{(n)}_t)$. Consequently you "can" replace $G_{t:T-1}$ with (estimated) Q^π and can perform updates every timestep like Q-learning—this is known as actor-critic.

   Hint: You might want to argue that rewards before (and excluding) timestep t is not affected by the policy at timestep t. You will need to mathematically show this—it might be helpful to draw connections with the PGM above.

4. Policy gradient assumes that our policy is differentiable. In this question we will assume that our policy is parameterized as a linear softmax policy. Specifically, a linear softmax policy is defined as

$$\pi_\theta(a|s) := \frac{\exp \theta^\top \phi(s,a)}{\sum_{a' \in \mathcal{A}} \exp \theta^\top \phi(s,a')}, \tag{1}$$

for any state-action pairs $(s,a) \in \mathcal{S} \times \mathcal{A}$. Here $\theta \in \mathbb{R}^d$ is the parameters (with d values) of the policy and $\phi(s,a) \in \mathbb{R}^d$ is the feature of the state-action pair (s,a). Note that, in order for π_θ to be a probability distribution, Equation (1) exponentiates the inner product $\theta^\top \phi(s,a)$ (i.e. linear w.r.t. ϕ) and normalize across all actions—this operation is known as the *softmax function*.

The softmax function is widely used in both reinforcement learning and deep learning—you will see more of this in this course. In general, you may choose other parameterizations, e.g., neural network as features, escort function over softmax function in discrete action space, Gaussian/Beta distributions in continuous action space, diffusion policies, etc. Nevertheless, linear softmax policy is our focus now.

With a linear softmax policy, the gradient $\nabla_\pi \log \pi(a|s)$ can be expressed as $\nabla_\theta \log \pi_\theta(a|s)$. Show that

$$\nabla_\theta \log \pi_\theta(a|s) = \phi(s,a) - \sum_{a' \in \mathcal{A}} \pi_\theta(a'|s)\phi(s,a').$$

Having this result and applying the policy gradient we can now implement the REINFORCE algorithm, which we will do in the programming component!

Hint 1: Recall that for function $f(g(x))$, we can apply chain rule to the gradient $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$.

Hint 2: It might be helpful to start with the identity $\log(\frac{a}{b}) = \log a - \log b$.

2 Programming Component

2.1 Python Packages and Imports

You will need to install:

```
$ pip install numpy
$ pip install gymnasium
$ pip install pygame
```

We have already included all the necessary Python packages for this assignment—you are **not allowed to import any other packages**.

2.2 Task

Remember the Frozen Lake environment from tutorials? Since obtaining the treasure from Lake Ontario and Lake Simcoe, you've learned to space travel efficiently. As an ambitious adventurer you are embarking a new journey to unexplored exoplanets. Before you step outside the space ship you decided to send a robot do some initial exploration. The robot has very limited memory so it cannot really store all possible states that it observes. Fortunately, you have learned about reinforcement learning with linear function approximation and it is perfect for this task! In this assignment, you will need to implement both Q-learning and REINFORCE algorithms with linear function approximation.

Files you will edit:

- `q_learning.py`: Q-learning agents with linear function approximation for discrete action space.
- `reinforce.py`: REINFORCE agents with linear softmax policy for discrete action space.
- `features.py`: The features that both Q-learning and REINFORCE agents to use—we have already included tabular feature and a reference linear feature.

Files you should read but can ignore:

- `env.py`: The frozen lake environment.
- `main.py`: The entrypoint for training RL agents.
- `test_q_learning.py`: The unit test for Q-learning agents.
- `test_reinforce.py`: The unit test for REINFORCE agents.

Training tabular RL agents. To train any RL agents, you may execute `python main.py --feature_type=tabular` to run a random policy using tabular features—note that this will not train anything. To get some understanding on what arguments are expected, please refer to Figure 2.

Unlike algorithms like value iteration, both Q-learning and REINFORCE are algorithms that learn from interactions with the environment. In both algorithms, you will need to implement:

- `get_action`: Computes the action to interact with the world. For Q-learning agents you will have to implement ε-greedy strategy. For REINFORCE agents you simply have to sample from the linear softmax policy.
- `compute_update`: This computes the update to be applied to the parameters. For Q-learning agents you will have to implement the Bellman optimality update. For REINFORCE agents you will have to implement the policy gradient update. Say the update is δ_k, the parameters will be updated through

  $$\theta_{k+1} = \theta_k + \alpha\delta_k,$$

  where θ is the parameters and α is the learning rate (or step size). Notice the difference that Q-learning agents update at every timestep, whereas REINFORCE agents update every trajectory of length T. The latter does perform same number of updates at the end because within a trajectory REINFORCE performs T parameter changes.

First test the implementations with the unit tests `test_q_learning.py` and `test_reinforce.py`. Note that your implementation should not have any randomness as we have already done those for you. Otherwise it is likely that you will fail the test cases even if the idea is correct.[1] Once you have some confidence in the implementation, you can run your algorithms using the tabular features:

```
# For Q-learning agents
$ python main.py --algorithm=q_learning --feature_type=tabular

# For REINFORCE agents
$ python main.py --algorithm=reinforce --feature_type=tabular
```

[1] In general it is way harder to test learning code because of randomness and numerical stability.

Figure 1: Frozen lake examples.

Throughout the run you will see the code reporting the average return every 100 trajectories. To speed the training up you may disable the rendering through `--disable_render` flag. When the code has finished execution, you will see a pickle file with name `params-<algo>-<feature>-<time>.pkl` that stores the parameters of the agent. You may load the parameter by providing `--load_params=<params_path>`.

Evaluation. To evaluate your trained agents, you can run:

```
# For Q-learning agents
$ python main.py --algorithm=q_learning --feature_type=tabular --evaluate --
    load_params=<params_path>

# For REINFORCE agents
$ python main.py --algorithm=reinforce --feature_type=tabular --evaluate --
    load_params=<params_path>
```

This will run the RL agent with the provided parameters without further updates. To further see how the agent generalizes to other tasks, you can change the environment seed `env_seed` and the map size `map_size`.

Function approximation. You might notice that using tabular features will not generalize most of the time. Here "generalize" means that you can use the same policy for different maps and the agent still achieves good performance. The generalization is poor with tabular features because the features implicitly memorize where the holes and treasure are. Using function approximation might alleviate this problem by focusing on general features (e.g. number of holes surrounding a tile, number of flat tiles on a row, direction of the treasure, etc.)

Now you will design your own linear features in `features.py`—we have provided a reference feature that you can try—it should be quite easy to outperform the reference feature on a new environment. To use your designed linear features or the provided linear features, use `feature_type=linear` or `feature_type=linear_ref` respectively. You should evaluate how good your features are by evaluating it on varying maps after training on the original map.

Grading. We will grade put 2 points based on the correctness of `get_action` and `compute_update` for each RL algorithm, with a total of 4 points. To evaluate your designed feature, we will grade it out of 2 (and up to +4 bonus points):

- 1/2 if your agent reaches the goal $\approx 25\%$ of the trajectories
- 2/2 if your agent reaches the goal $\approx 50\%$ of the trajectories
- +1 bonus if your agent is top 10% of the class performance with Q-learning (in average return)
- +1 bonus if your agent is top 10% of the class performance with REINFORCE (in average return)
- +1 bonus if your agent is the best of the class performance with Q-learning (in average return)
- +1 bonus if your agent is the best of the class performance with REINFORCE (in average return)

We will use multiple unused maps to evaluate the robustness of the features.

3 Submission Instructions

For the written component, please submit `A2.pdf`. For the programming component, please submit only `q_learning.py`, `reinforce.py`, and `features.py`.

```
$ python main.py --help
usage: main.py [-h] [--map_size MAP_SIZE] [--horizon HORIZON] [--env_seed
    ENV_SEED]
                [--sample_seed SAMPLE_SEED] [--algorithm {random,q_learning,
                   reinforce}]
                [--feature_type {tabular,linear,linear_ref}] [--num_trajs
                   NUM_TRAJS]
                [--alpha ALPHA] [--evaluate] [--disable_render]
                [--load_params LOAD_PARAMS] [--eps EPS]

options:
    -h, --help            show this help message and exit
    --map_size MAP_SIZE   The size of the map
    --horizon HORIZON     The length of the trajectory (or horizon)
    --env_seed ENV_SEED   The random seed to use for the environment
    --sample_seed SAMPLE_SEED
                          The random seed to use for the sampling, including
                             parameter
                          initialization, action sampling, and trajectory
                             seeding
    --algorithm {random,q_learning,reinforce}
                          The algorithm to use
    --feature_type {tabular,linear,linear_ref}
                          Feature to use
    --num_trajs NUM_TRAJS
                          The number of trajectories
    --alpha ALPHA         The learning rate, alpha, for training
    --evaluate            Whether or not to evaluate agent
    --disable_render      Whether or not to disable rendering---this will
       speed up the code
    --load_params LOAD_PARAMS
                          Whether or not to load a trained agent
    --eps EPS             Epsilon in epsilon-greedy exploration
```

Figure 2: Arguments for running `main.py`