A Tabular Q-Learning Hyperparameter Sweeps

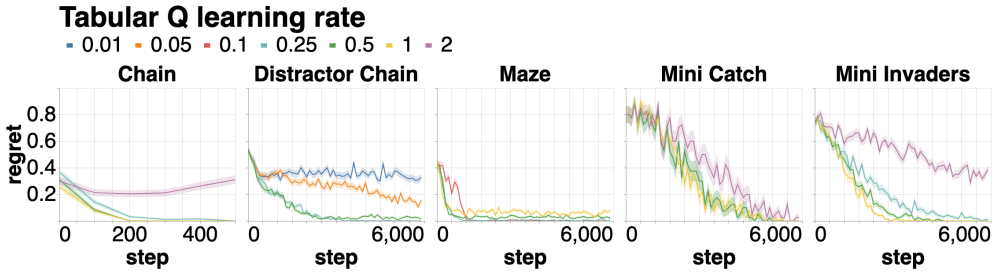

Figure 1: This figure presents learning curves using the standard Tabular Q-Learning algorithm. Given an appropriately chosen learning rate, this algorithm does converge, but this can take several thousand timesteps on all but the simplest domains.

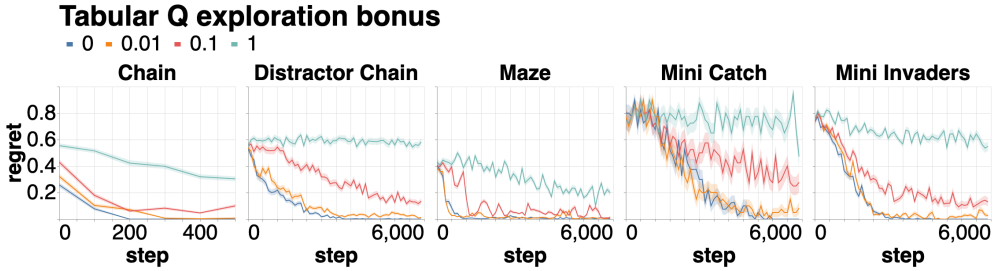

Figure 2: This figure presents learning curves for Tabular Q-Learning with a count-based exploration bonus. Actions are chosen using the following formulation: $a \leftarrow \arg\max_{a'} Q(s,a') + k/(1+N(s,a'))$ where k is the exploration bonus hyperparameter, assuming values $\{0, 0.01, 0.1, 1\}$ in the graph, and $N(s,a')$ is the visitation count for the state-action (s,a').

B Random Prediction Ablation

This ablation studies the contribution that each prediction type makes to ICPI's ability to learn. To this end, for prediction type — observation, termination, action, and reward — we ran an ablation that substitutes a random, valid prediction for the prediction made by the LLM. For example, for actions, we first sample a random action from the action space of the environment and then apply the standard string formatting used in other parts of the algorithm. As figure 3 indicates, on Chain, the simplest of our environments, none of the ablations are able to learn.

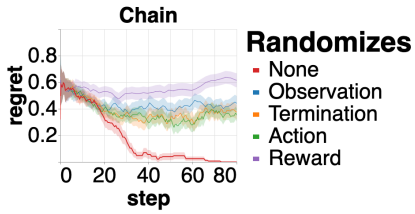

Figure 3: This figure presents learning curves for the random prediction ablation described above on the Chain domain. The Y-axis represents regret and the X-axis represents timesteps in the environment. Each colored line refers to a different prediction which was randomized. The red line is the original ICPI learning curve, included for reference.

C Error Analysis

In this section we present examples of errors made by ICPI on each domain. In each subsection, we will examine one rollout generated by the Codex LLM that causes the agent to choose suboptimal behavior in the actual environment.

Chain

Here, we analyze an example in which the rollouts incorrectly estimate equal values for all three actions. Each line in the code snippet represents actual output from the LLM, excluding comments which are added for clarity. The agent's state is 5 (recall that the goal state is 4). For the `left()` action, the model generates the following rollout:

```
assert state == 5 and state != 4  # the current, starting state
state = left()  # a new state is the result of the left() action
assert not done
assert reward == 0
assert state == 4 and state == 4  # The new state is state 4, the goal state

# Here the rollout policy unexpectedly chooses to move right
# instead of trying the goal.
state = right()
assert not done
assert reward == 0
assert state == 5 and state != 4

# Again the rollout policy outputs a suboptimal action:
reward = try_goal(state)
assert done  # try_goal always terminates an episode with 0 reward
assert reward == 0
```

Note that the model outputs the action `right()` when the agent is in goal

state 4, and then outputs action `try_goal()`. As a result the rollout estimates value 0 for an action which has value γ under the optimal policy (one timestep away from a terminal state with reward 1).

The estimate for action `try_goal()` is accurate in state 5:

```
assert state == 5 and state != 4  # the current, starting state
reward = try_goal(state)
assert done
assert reward == 0
```

However the rollout also underestimates the value of action `right()`:

```
assert state == 5 and state != 4  # the current, starting state
state = right()
assert not done
assert reward == 0
assert state == 6 and state != 4
reward = try_goal(state)  # left() would be the optimal policy, not try_goal().
assert done
assert reward == 0
```

Distracter Chain

In this example the agent starts in state `(2, 3)`. Recall that in this environment, the second digit in the tuple is chosen randomly and does not affect environment transitions or reward.

In the first of the three rollouts, the first action is `left()`. Since the goal is state (4, *), this action is already suboptimal. However, we see that the agent continues to behave sub-optimally throughout the rest of the rollout by continuously choosing action `left()` instead of the optimal action `right()`. This occurs because at this stage of training, the agent has mostly taken `left()` actions and as a result, the rollout policy prompt mostly contains `left()` actions.

```
assert state == [2, 3] and state != [4, 3]
state = left()
assert not done
assert reward == 0
assert state == [1, 2] and state != [4, 2]
state = left()  # The optimal action is right()
assert not done
assert reward == 0
assert state == [0, 0] and state != [4, 0]
state = left()  # The optimal action is right()
assert not done
assert reward == 0
assert state == [0, 0] and state != [4, 0]
state = left()  # The optimal action is right()
```

```
assert not done
assert reward == 0
assert state == [0, 0] and state != [4, 0]
state = left()  # The optimal action is right()
assert not done
assert reward == 0
assert state == [0, 0] and state != [4, 0]
state = left()  # The optimal action is right()
assert not done
assert reward == 0
assert state == [0, 1] and state != [4, 1]
state = left()  # completion
```

The rollout for the `try_goal()` action is correct:

```
assert state == [2, 3] and state != [4, 3]
reward = try_goal(state)
assert done
assert reward == 0  # completion
```

Here the initial action is optimal: `right()`. However, we once again see that the rollout policy is suboptimal because it favors the `left()` action over the optimal `right()` action.

```
assert state == [2, 3] and state != [4, 3]
state = right()
assert not done
assert reward == 0
assert state == [3, 7] and state != [4, 7]
state = left()
assert not done
assert reward == 0
assert state == [2, 4] and state != [4, 4]
state = left()
assert not done
assert reward == 0
assert state == [1, 7] and state != [4, 7]
state = left()
assert not done
assert reward == 0
assert state == [0, 0] and state != [4, 0]
state = left()
assert not done
assert reward == 0
assert state == [0, 0] and state != [4, 0]
state = left()
assert not done
```

```
assert reward == 0
assert state == [0, 0] and state != [4, 0]
state = left()
```

This causes the model to inaccurately produce a value estimate of 0 for the `right()` action.

Maze

Similar to the previous example, we study an example for maze early in training, when the rollout policy is still quite suboptimal. As a result, the rollout policy does not discover the reward, causing the rollouts to estimate equal value for each of the four actions.

The rollouts all start at state (1, 1).

```
assert state == C(i=1, j=1) and state != C(i=1, j=0)
state, reward = left()
assert not done
assert reward == 0
assert state == C(i=1, j=0) and state != C(i=1, j=0)
state, reward = right()
assert not done
assert reward == 0
assert state == C(i=1, j=1) and state != C(i=1, j=0)
state, reward = up()
assert not done
assert reward == 0
assert state == C(i=0, j=1) and state != C(i=1, j=0)
state, reward = down()
assert not done
assert reward == 0
assert state == C(i=1, j=1) and state != C(i=1, j=0)
state, reward = up()
assert not done
assert reward == 0
assert state == C(i=0, j=1) and state != C(i=1, j=0)
state, reward = down()
assert not done
assert reward == 0
assert state == C(i=1, j=1) and state != C(i=1, j=0)
state, reward = up()

assert state == C(i=1, j=1) and state != C(i=1, j=0)
state, reward = down()
assert not done
assert reward == 0
assert state == C(i=2, j=1) and state != C(i=1, j=0)
```

```
state, reward = down()
assert not done
assert reward == 0
assert state == C(i=2, j=1) and state != C(i=1, j=0)
state, reward = left()
assert not done
assert reward == 0
assert state == C(i=2, j=0) and state != C(i=1, j=0)
state, reward = right()
assert not done
assert reward == 0
assert state == C(i=2, j=1) and state != C(i=1, j=0)
state, reward = right()
assert not done
assert reward == 0
assert state == C(i=2, j=2) and state != C(i=1, j=0)
state, reward = right()
assert not done
assert reward == 0
assert state == C(i=2, j=2) and state != C(i=1, j=0)
state, reward = right()

assert state == C(i=1, j=1) and state != C(i=1, j=0)
state, reward = up()
assert not done
assert reward == 0
assert state == C(i=0, j=1) and state != C(i=1, j=0)
state, reward = down()
assert not done
assert reward == 0
assert state == C(i=1, j=1) and state != C(i=1, j=0)
state, reward = up()
assert not done
assert reward == 0
assert state == C(i=0, j=1) and state != C(i=1, j=0)
state, reward = right()
assert not done
assert reward == 0
assert state == C(i=0, j=2) and state != C(i=1, j=0)
state, reward = right()
assert not done
assert reward == 0
assert state == C(i=0, j=2) and state != C(i=1, j=0)
state, reward = right()
assert not done
assert reward == 0
```

```
assert state == C(i=0, j=2) and state != C(i=1, j=0)
state, reward = right()

assert state == C(i=1, j=1) and state != C(i=1, j=0)
state, reward = right()
assert not done
assert reward == 0
assert state == C(i=1, j=2) and state != C(i=1, j=0)
state, reward = right()
assert not done
assert reward == 0
assert state == C(i=1, j=2) and state != C(i=1, j=0)
state, reward = down()
assert not done
assert reward == 0
assert state == C(i=2, j=2) and state != C(i=1, j=0)
state, reward = right()
assert not done
assert reward == 0
assert state == C(i=2, j=2) and state != C(i=1, j=0)
state, reward = right()
assert not done
assert reward == 0
assert state == C(i=2, j=2) and state != C(i=1, j=0)
state, reward = right()
assert not done
assert reward == 0
assert state == C(i=2, j=2) and state != C(i=1, j=0)
state, reward = right()
```