Fundamental Reinforcement Learning Algorithms

Nurrachman Liu
Department of Computer Science
UCLA
`rachliu@cs.ucla.edu`

1 Introduction

This paper explores some basic RL algorithms learned in our class, to gain better understanding and intuition. The original intent was to investigate the paper [1], which is focused on data-efficient (sample-efficient) Hierarchical Reinforcement Learning (HRL). However, due to lack of time and manpower, I decided it would be more important for my understanding to focus on more fundamental concepts.

We explore the following concepts learned in class:

- Basic TD algorithms: Q-Learning and Sarsa.
- Look-ahead TD algorithms: Sarsa-λ.
- Exploration Methods beyond ϵ-greedy: Upper-Confidence Bounds (UCB).
- Gradient Bandit Algorithms (precursor for Gradient Algorithms in RL setting).
- Basic MC Methods in RL.
- Basic function approximation in RL (Linear methods).

In particular, we implemented the following specific algorithms and demos:

- Q-Learning and Sarsa (TD-0).
- Sarsa-λ (TD-1).
- Simple-bandit Algorithm with, without UCB (MAB setting).
- MC-Control using 1st-Visit MC and Every-Visit MC.
- Gradient Bandit Algorithm described in Ch2 of [2].
- Linear Function Approximation with Q-Learning (partially complete; some parts unsuccessful).

We used the following environments in OpenAI Gym [3]:

- MiniGridworld [4] (we had to customize this for classic Gridworld)
- Bandits Environment [5] (for MAB)

The paper is outlined as follows:

- Implementation & Intuitive Discussion
- Results
- Appendices (Algorithm outlines, Code implemented)

Preprint. Under review.

2 Implementation & Intuitive Discussion

This section discusses our understanding and thoughts on the implementations of the different demo experiments.

2.1 Q-Learning and Sarsa

Q-Learning and Sarsa are *temporal difference* techniques applied to learning Q-values. Temporal difference techniques simply means using running averages as estimators for the Q/V-values, updated from time-step (iteration) to time-step, rather than relying on having the *entire* trajectory beforehand to compute the expected rewards G_t at each visited state to use to calculate their estimators for Q/V-values. Thus, temporal difference techniques are online: their estimates for Q/V-values can have maximum granularity (iteration to iteration), rather than MC methods which must be offline (maximum granularity is episode to episode).

Both Q-Learning and Sarsa are Q-value methods; that is, they learn Q-values rather than V-values. V/Q-values are related as $V(s) := max(Q(s, :) \equiv max(Q(s, a') \; \forall a' \in \mathcal{A}$. Practically, V-value methods lose/don't-store policy information because they store a single value at each state, rather than one V-value per-action. Typically, however, a V-value method would have computed the future values dependent on taking some action anyways, in order to make comparisons between V-value estimates to store. Thus, Q and V values are the essentially the same, but Q-values explicitly store the values per action.

The two can be confusing, because sometimes Policy Evaluation (PE) steps that use V-value methods will *implicitly* use a policy-extraction step to find the best next-action to take (ie, when sampling trajectories using an ϵ-greedy approach) in the form of: $a = argmax(V(s'))\forall a' \in \mathcal{A} s.t. step(a') \to s' \equiv argmax(Q(s', :))$. Another point of confusion is that sometimes V-value methods estimate the V-values using an implicit action-search: $V_{t+1}(s) \leftarrow max_{a \in \mathcal{A}}(R(s, a) + \gamma P(\cdot|s, a)^T V^t)$, which is then equivalent to just storing Q-values and taking a max over stored Q-values (which, many V-value methods do): $V_{t+1}(s) \leftarrow max_{a \in \mathcal{A}} Q_t(s, a)$. In this case, it may look like a V-value method is also a 'Q-value method', even though it is just pre-storing action-values using Q-values.

Thus, Q-value methods facilitate policy extraction more easily if only for the simple fact that they pre-store all the V-values partitioned by action.

Q-Learning and Sarsa differ in that the former is off-policy while the latter is on-policy. That is, Q-Learning steps from a current state s to a next-state s', then uses the *best possible* Q-value at this next-state s' (ie, for all possible actions from next-state, $Q(s', :)$) to update the Q-value for the state s that it just stepped from. This is off-policy because it may end up updating the previous state's Q-value ($Q(s, a)$) with a Q-value corresponding to some action $a_2 \neq a$ that also arrives in the same next state: ie, $Q(s, a) \leftarrow Q(s', a_2)$. In contrast, Sarsa steps from a current state s to a next-state s', and then uses the Q-value for the *same* action a it just took to get to next state s' (ie, $Q(s', a)$), to update the state that it just stepped from with (ie, $Q(s, a) \leftarrow Q(s', a)$).

2.2 Sarsa-λ

The Sarsa-λ algorithm uses the weighted average of all n-step Sarsa from 0-step (ie, a TD(0) target) to H-step (ie, the maximum trajectory length, ie, G_t used in the MC methods).

One drawback of using (forward-view) n-step methods is the requirement to have the next n steps of results beforehand; that is, the algorithm cannot be fully online from iteration to iteration. A clever solution to this problem is the backwards-view λ, which accumulates some running counters/averages (called 'Eligibility Traces', tracked per-(s,a)) in the opposite direction to the trajectory direction (ie, the previous steps). It has been shown mathematically that the backwards-view λ and forwards-view λ are equivalent. The proof relies on expanding out the TD error targets recursively in time steps to show that all the telescoping sum terms in between cancel out neatly.

2.3 UCB

We experiment with the Upper-Confidence-Bounds (UCB) approach on the MAB and RL settings. The UCB approach uses the Optimism in the Face of Uncertainty Principle [6], where the more times you pull an arm for an MAB (take an action in RL), the more confident you can be about its value estimate.

2.3.1 MAB

In the MAB setting, the UCB-method adds an estimate of the variance to the Q-value being estimated; this estimated variance term is the upper-bound of the confidence interval for the estimated mean expected reward (Q-value) of that arm; this is tracked per-arm using the number of pulls $N(a)$ to each arm: $\hat{U}_t := c\sqrt{\frac{lnt}{N_t(a)}}$. The '$\alpha$'-confidence level is specified through the c parameter. This is akin to maintaining an optimistic-initialization for each arm, which decreases with the number of pulls to each arm. Thus, the argmax selection for which arm now has a 'bonus' term added: $A_t := argmax_a(\hat{Q_t(a)} + \hat{U}_t)$.

The above bound is derived from using Hoeffding's Inequality to derive the concentration inequalities around the mean expected reward per arm. Concentration inequalities measure how a random variable (usually, a sum of independent random variables) deviates around its mean, and usually take the form of a 2-sided bound (the simplest being Chebyshev's) [7]. The normal distribution offered by applying CLT to measure a sum of iid RVs, however, has an error of approximation that decays too slowly (slower than linear); thus, Hoeffding's is applied which bypasses using the CLT to measure the concentration inequality more directly [7].

2.3.2 RL

The idea behind applying a bound using concentration measures of the sum of iid RVs ('pulls') can be applied in the RL setting to basically do the same modification to an existing RL algorithm which allows sufficient exploration, ie, $\hat{Q}(s,a)' := \hat{Q}(s,a) + \hat{U}(s,a)$. The caveat here, however, is that Hoeffding's inequality can no longer be used to estimate this $\hat{U}(s,a)$, because the estimated probabilities for some state $s'|s,a$ are now dependent on their estimated V-values from the preceding time-step h_{t+1} [6] (ie, because the states are traversed in a dependent fashion).

From [6], we can use a regret decomposition to show that the overall regret across all iterations $T := HK$ is proportional to $H^2 \cdot \sqrt{S^2AT \cdot logT}$ (where H is # of max iterations per episode (ie, max trajectory length), A is size of action space, T is total iterations across all episodes). The interesting part here is the $logT$ factor; since $T := HK$, this term is proportional to $\sqrt{K \cdot logK}$, leading to an average regret proportional to $\sqrt{K \cdot logK}/K \sim \sqrt{logK/K}$, which decays to 0 as iterations increase (ie, it can actually learn).

From [8], and relatedly, some results from [9], a Q-learning algorithm using UCB is shown, called 'Q-Learning with UCB-Hoeffding'. We implement this algorithm in a simple environment for demo.

2.4 Gradient Bandit Algorithms

In gradient bandit algorithms, a different function, $H(a)$, now guides the action selection, rather than $Q(a)$ as before. $H(a)$ represents a numerical preference for action a. Actions are selected by *relative* comparison of their $H(a)$ scores through a softmax.

This normalized score thus becomes a probability of selecting the action, and is defined as $\pi_t(a) := Pr(A_t) := \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}}$, the probability of selecting action a at time t.

This is naturally extended to the full RL setting in policy gradient methods.

In the update equation for $H(a)$, the update delta is proportional to $(R_t - \bar{R}_t)$, where the latter is the accumulated rewards that serves as the 'baseline' for relative comparisons with the received reward R_t. The reward R_t is $q^*(a)$, which also implies that its running mean $\bar{R}_t$ is a running mean of Q-values. Note, however, that $H(a)$ values have no direct interpretation as rewards, since the softmax comparison means that additive offsets cancel out.

It seems then that $H(a)$ is capturing the relative change in Q-values, since Q-values represent future expected rewards.

The equations used for the implementation of gradient bandit algorithms are [2]: $H_{t+1} := H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t))$ (for the selected action A_t) and $H_{t+1} := H_t(A_t) - \alpha(R_t - \bar{R}_t)\pi_t(a)$, for all non-selected actions $a \neq A_t$. Thus, preferences for non-selected actions move opposite to those for the selected action.

2.5 MC Methods in RL

Given an environment, we can start with a (random) policy and iteratively improve it through MC-Control, which is the same idea as Policy Iteration.

We explore first-visit MC and every-visit MC by using them as the evaluation steps in a MC-Control setting.

Since we do not have any explicit model of the environment, we use the MC methods to estimate action values (that is, Q-values for (state,action) pairs) rather than V-values for states). Thus, this enables us to start with any arbitrary policy and iteratively improve it with only estimation of state-action values (Q-values), given enough samples. We thus use an ϵ-greedy MC-Control scheme.

To ensure that all state-action pairs are explored, there are two methods: exploring-starts, where all state-action pairs are guaranteed a non-zero probability of being started at, or simply requiring that the policy function itself is stochastic (all state-action pairs have probability $\in (0, 1)$).

We use an ϵ-greedy stochastic policy. This is the same as was done for ϵ-greedy Q-Learning, except that the argmax is now taken on a stochastic policy at (s,a), that is, the action with maximum probability (rather than the action with maximum reward).

In first-visit MC, we start with some (arbitrary) policy, and sample some number of episodes (trajectories) from it. For each sampled trajectory, we can examine all states in the visited sequence; for each state's *first* visit, we increment the $N(s, a)$ estimator by 1, and the cumulative reward estimator $C(s, a)$ by G_t (thus, MC methods are limited to an update granularity of per-episode, ie, not 'online' / per-step updatable as TD-methods are). The Q-value estimator is then updated for every such (s,a) seen in the trajectory using $Q(s, a) = C(s, a)/N(s, a)$.

2.6 Linear Function Approximation in RL

Linear approximation methods are the simplest to understand as well as giving the most intuition. Thus, we explored these in our experiments.

We examine the random-walk example of [2]. State aggregation is the simplest way of linear feature representation, which can be seen as an approximation of one-hot state encoding by grouping states. It is also equivalent to a tile-coding scheme with overlap of 0. Finally, it is a special case of SGD where the gradient $\nabla\hat{v}(S_t, \vec{w_t})$ "is 1 for S_t's group's components and 0 for the other components[2]", since the feature vectors are aligned exactly with their parametrization.

2.6.1 Tile-coding and Receptive Field

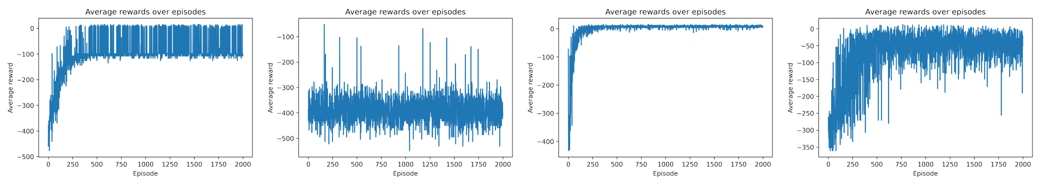
Tile-coding with non-zero overlap increases the receptive field of aggregated features. The idea is that tile-coding provides the *conjunctive* form of coarse generalization [2], where the overlapping tiles form the active feature dimensions for a particular state (observation).

Practically, implementing this into our system required just calculating the different tiles which an observation falls into, and then taking the dot-product with the weights to get the total value.

Other interesting tiling patterns exist for generalization, and good mixtures of all of them (horizontal, vertical, diagonal, and conjunctive) tend to be the best for good generalization [2].

2.6.2 Comparison of State Aggregation to N-step TD methods

State aggregation has a similar effect as n-step TD methods [2], as both n-step TD methods and state aggregation group states together concordant with the order in which they are encountered while traversing the model. Thus, pre-estimating returns over n-steps averages their returns over n subsequent states, which is similar (roughly) to grouping the subsequent n-states. The two are most



(a) (b) (c) (d)

Figure 1: Taxi env. (a) Sarsa, ϵ-decay (b) Sarsa, no decay (c) Q-Learning, ϵ-decay (d) Q-learning, no decay

```
WWWWWWWW
Wo     W
W      W
W      W
W      W
W      W
W     GW
WWWWWWWW
```
(a)

```
WWWWWW
W    W
W    W
Wo  GW
VVVVVV
```
(b)

```
WWWWWWWWWWW
W         W
W         W
W         W
W         W
Wo       GW
VVVVVVVVVVV
```
(c)

Figure 2: Gridworld maps. (a) Empty 8x8. (b) Cliff h=3 w=3. (b) Cliff h=5 w=9.
Legend: W: wall, V: lava, G: goal, o: agent

similar in a left-right random walk, since the state sequence that can be possibly encountered is fixed by design.

2.7 Policy Gradient Methods in RL

3 Results

3.1 Verification of implementation of Q-Learning and Sarsa baseline algorithms, using Taxi environment

The taxi environment served to verify that my baseline implementations (Q-Learning, Sara) are working correctly.

In addition, Figure 1 shows the impact of ϵ-decay on these two baseline algorithms. ϵ-decay cuts down on the noise and allows the algorithm to transition to exploitation (a,c vs b,d). In the finite sample setting, Sarsa appears to learn a suboptimal value (a,b), while Q-learning learns the optimal values (c,d).

3.2 Sarsa vs Sarsa-λ

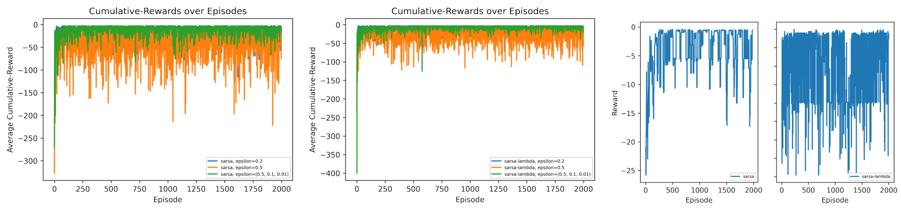
Figure 2 shows the various Gridworld maps used for testing.

3.2.1 Gridworld Empty8x8: Sarsa vs Sarsa-λ

Figure 3. We implemented Sarsa-λ (backwards-view) and compare its performance with regular Sarsa on the Gridworld Empty8x8 room. Sarsa-λ performs much better than Sarsa in all cases (large ϵ of 0.5 (orange) or smaller ϵ of 0.2 (blue) or decaying-ϵ (green)).

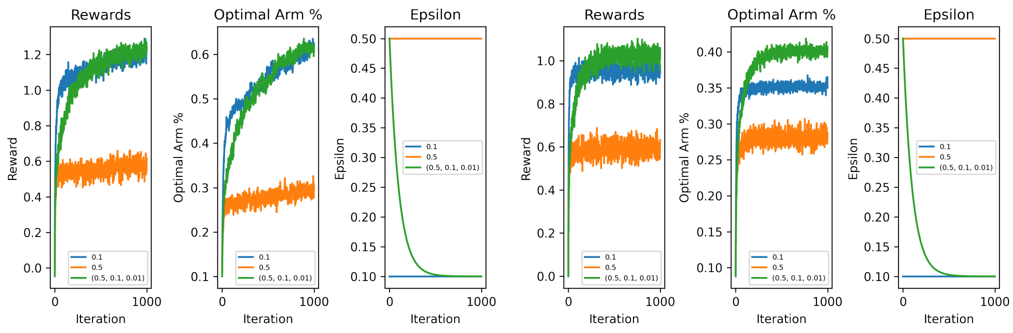
3.2.2 Gridworld Cliff: Sarsa vs Sarsa-λ

Our Cliff grid (Figure 2) set-up is: -100 reward for falling into the cliff, -1 reward on all transitions, and +10 reward for reaching the goal. The +10 reward for goal allows the Q-values of the goal-reaching paths to propagate faster through the Q-table.



(a) (b) (c)

Figure 3: (a,b) Empty8x8. (a) Sarsa (b) Sarsa-λ. ϵ-decay (orange), ϵ=0.2,0.5 (blue, orange). (c) Cliff5x9. Cumulative-reward per episode, Sarsa (left) vs Sarsa-λ (right), Both algorithms run with ϵ-decay.



(a) (b)

Figure 4: ϵ-greedy Simple-bandit algorithm, 10-armed Gaussian MAB. Varying ϵ's: 0.1, 0.5, and exponential-decay starting at 0.5. (a) $\alpha = 1/N(a)$. (b) $\alpha = 1e$-4.

Figure 3. The Sarsa-λ results show definite improvement over regular SARSA; this is highlighted in the cumulative rewards shown in the figure, where we see Sarsa-λ achieves the goal (positive cumulative reward) close to all the time, while Sarsa still struggles.

3.3 UCB-1 in the MAB setting: Simple-bandit algorithm with UCB-1

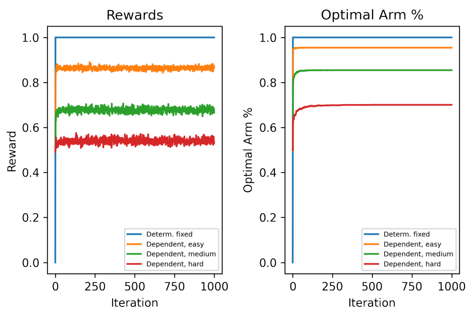
3.3.1 Simple-bandit algorithm on a 10-arm-Gaussian

Figure 4. The stationary-version using 1/N(a) appears to perform better than the non-stationary version that uses the constant α. The latter version may simply require more total episodes to attain a similar performance.

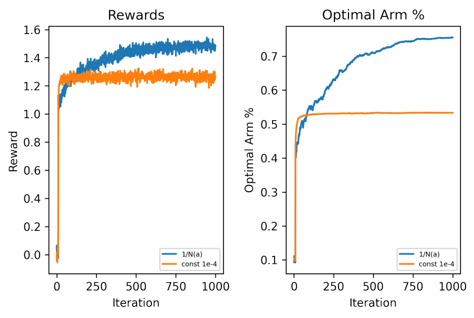
3.3.2 Simple-bandit algorithm with UCB-1 on 2-arm-p,$(1-p)$ and 10-arm-Gaussian

Figure 5. The first problem (a) is a 2-arm MAB with varying difficulties. The easiest instance has arm deterministic arm probabilities 0.2 and 0.8; the other three 2-arm MABs have dependent arm probabilities p and 1-p selected from $[0.1, 0.9]$, $[0, 25, 0.75]$, and $[0.4, 0.6]$, respectively. The average rewards drop lower for the harder case, being weighted towards 0.5, where the hardest instances lie.

For the 10-arm-Gaussian, we see that constant-α is unable to learn past a certain point, while the UCB-1 version continues to improve. This is illustrated directly by percent of time the two algorithms select the optimal action (b, right); UCB-1 (blue) continues to improve up to ≈ 80+%, far surpassing the ≈ 50+% of the non-UCB version.



(a)



(b)

Figure 5: UCB-1 Simple-bandit algorithm [2]. (a) 2-arm, $\alpha = 1/N(a)$ (left). (b) 10-arm Gaussian, $\alpha = 1/N(a)$ (right, blue) or $\alpha = 1e - 4$ (right, orange).

3.4 UCB-1 in the RL setting: Q-Learning with UCB-Hoeffding

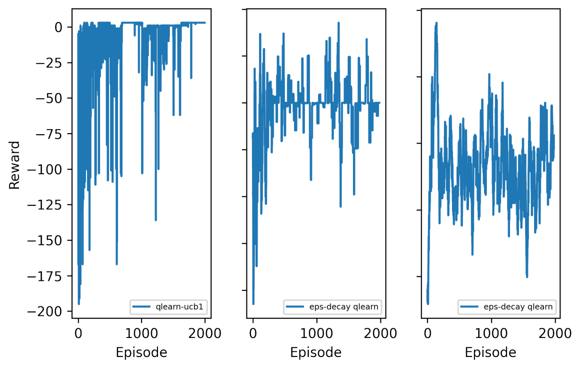


Figure 6: Q-Learning with UCB-1 on Cliff grid H=5,W=9. (left) with UCB-1. (middle) ϵ-decay, optimistic initialization (right) ϵ-decay, zeroes initialization .

Figure 6. The UCB-1 version is able to learn, eventually having trajectories always reach the goal. Regular Q-Learning, in contrast, is unable to get to the goal within a single trajectory. For the optimistically-initialized regular Q-Learner (middle), the optimistic values seem to cause local optima such as 'staying' in the top-left corner or hitting against the walls repeatedly. Given more trajectory length, these will (eventually) converge to the true optimal. The Q-values that the regular Q-Learners learn do lead to an optimal policy when extracted, but the ϵ-greedy is too noisy and causes many 'cliff-deaths'.

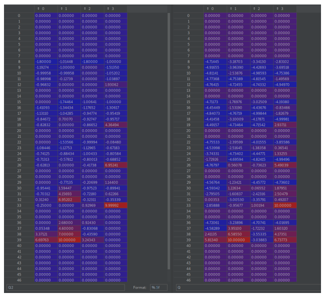
3.5 Basic Monte Carlo Methods in RL

3.5.1 MC-Control with First and Every Visit MC

Figure 7 shows the Q-Values for first-visit and every-visit MC used as the Policy Evaluation steps in an ϵ-greedy MC-Control.

3.6 Gradient Bandit Algorithm

Figure 8 shows the cumulative rewards across iterations, averaged across 2000 independently randomly generated 10-arm-Gaussian MAB instances.



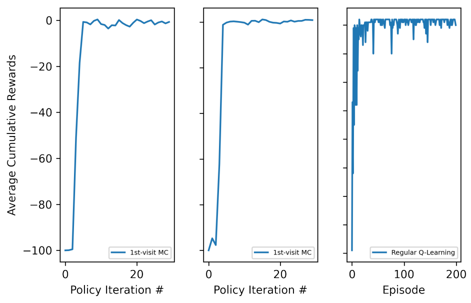
	0	1	2	3
0	0.00000	0.00000	0.00000	0.00000
1	0.00000	0.00000	0.00000	0.00000
2	0.00000	0.00000	0.00000	0.00000
3	0.00000	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00000
5	0.00000	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00000
7	0.00000	0.00000	0.00000	0.00000
8	-1.80000	-1.01448	-1.80000	-1.00000
9	-1.19274	-1.00000	0.00000	-1.51350
10	-0.99958	-0.99958	0.00000	-1.05202
11	-0.98998	-0.12739	0.00000	-1.03897
12	-0.96832	0.00000	0.00000	1.93062
13	0.00000	0.00000	0.00000	0.00000
14	0.00000	0.00000	0.00000	0.00000
15	0.00000	-1.74464	-1.00641	-1.00000
16	-1.41093	-1.34434	-1.17652	-1.30617
17	-1.13110	-1.04285	-0.94774	-0.95419
18	-0.84672	0.70070	-0.92747	-0.95717
19	-0.82832	0.00000	-0.86652	4.36494
20	0.00000	0.00000	0.00000	0.00000
21	0.00000	0.00000	0.00000	0.00000
22	0.00000	-1.53566	-0.99994	-0.08480
23	-1.08446	-1.12753	-1.12965	-0.67183
24	-0.74125	-0.88434	-1.00433	-0.80584
25	-0.71313	-0.57812	-0.80013	-0.68852
26	-0.62813	0.00000	-0.41738	6.95241
27	0.00000	0.00000	0.00000	0.00000
28	0.00000	0.00000	0.00000	0.00000
29	0.00000	-0.77135	-0.20699	1.14400
30	-0.95441	1.59447	-0.97523	-0.89941
31	-0.70312	4.15693	-0.72180	0.62266
32	0.31240	6.95202	-0.32311	-0.35339
33	-0.25000	0.00000	0.92969	9.99992
34	0.00000	0.00000	0.00000	0.00000
35	0.00000	0.00000	0.00000	0.00000
36	0.00000	2.68000	0.92430	0.00000
37	0.05348	4.60000	-0.83068	0.00000
38	3.37121	7.00000	-0.43590	0.00000
39	6.69763	10.00000	3.24343	0.00000
40	0.00000	0.00000	0.00000	0.00000
41	0.00000	0.00000	0.00000	0.00000
42	0.00000	0.00000	0.00000	0.00000
43	0.00000	0.00000	0.00000	0.00000
44	0.00000	0.00000	0.00000	0.00000
45	0.00000	0.00000	0.00000	0.00000
46	0.00000	0.00000	0.00000	0.00000

Q2 Format: %.5f

	0	1	2	3
0	0.00000	0.00000	0.00000	0.00000
1	0.00000	0.00000	0.00000	0.00000
2	0.00000	0.00000	0.00000	0.00000
3	0.00000	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00000
5	0.00000	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00000
7	0.00000	0.00000	0.00000	0.00000
8	-4.71445	-3.18703	-3.34130	-2.83102
9	-4.91655	-3.96390	-4.42693	-3.69518
10	-4.81141	-2.53876	-4.98593	-4.75386
11	-4.77368	-4.75389	-4.81545	-1.49569
12	-4.76415	-4.72455	-4.74332	-4.94970
13	0.00000	0.00000	0.00000	0.00000
14	0.00000	0.00000	0.00000	0.00000
15	-4.71173	-1.76976	-3.02509	-4.19380
16	-4.45449	-1.53190	-4.43676	-0.83466
17	-4.84073	-4.76719	-4.99844	-1.82679
18	-4.41458	-3.10009	-4.17871	-4.99981
19	-4.49157	-4.73464	-4.71542	2.93711
20	0.00000	0.00000	0.00000	0.00000
21	0.00000	0.00000	0.00000	0.00000
22	-4.75533	-2.39599	-4.01555	-3.85586
23	-4.53998	-2.51645	-1.38358	0.36541
24	-3.74331	-4.73402	-4.44771	1.33255
25	-1.72926	-4.69594	-4.82415	-4.99496
26	-4.76797	0.56078	-0.73623	5.49039
27	0.00000	0.00000	0.00000	0.00000
28	0.00000	0.00000	0.00000	0.00000
29	-4.56764	-1.23421	-4.45772	-4.73602
30	-4.59342	1.22634	0.06552	1.87951
31	-2.79505	-1.60837	-2.42316	2.50479
32	0.00353	-3.00530	-3.35791	0.49207
33	-2.85888	-0.95677	1.00194	10.00000
34	0.00000	0.00000	0.00000	0.00000
35	0.00000	0.00000	0.00000	0.00000
36	-4.72061	-3.23896	-4.70741	-4.01695
37	-4.58289	3.95100	-1.72232	1.60320
38	2.41135	6.58550	-3.55335	4.17351
39	5.81340	10.00000	-3.03883	6.73773
40	0.00000	0.00000	0.00000	0.00000
41	0.00000	0.00000	0.00000	0.00000
42	0.00000	0.00000	0.00000	0.00000
43	0.00000	0.00000	0.00000	0.00000
44	0.00000	0.00000	0.00000	0.00000
45	0.00000	0.00000	0.00000	0.00000
46	0.00000	0.00000	0.00000	0.00000

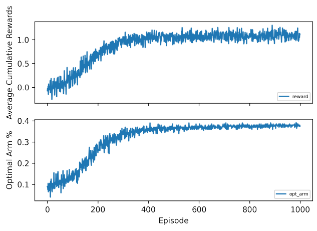
Q

(a)



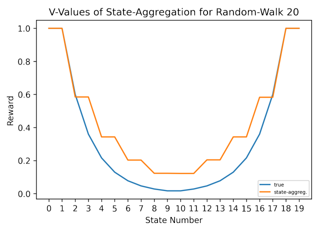
(b)

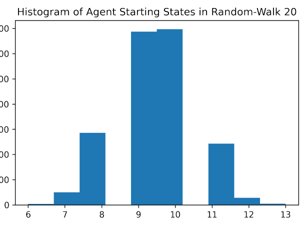
Figure 7: MC-Control Q-value convergence verification. (a) Q-Values after 30 Policy Iterations, (left) 1st-visit, (right) Regular Q-Learning. (b) Cumulative Rewards per Episode (averaged over 200 episodes), (left) 1st-visit (middle) Every-visit, (right) Regular Q-Learning.



(a)

Figure 8: Gradient algorithm for MAB. $\alpha = 0.1$ (top) Cumulative Rewards (bottom) Optimal Arm %.





(a) (b)

Figure 9: State aggregation V-values for Random-Walk 20, 2 states per group. (a) V-Values (b) Starting-State distribution.

3.7 Linear Function Approximation

3.7.1 State Aggregation applied to Random Walk on Gridworld Empty

Figure 9 shows the effect of state aggregation on V-values, for the Randomwalk-20. As expected, state aggregation concordant with the state-sequence of the actual model discretizes the state space on the same 'axis' as the actual state visits, and so results in the staircase pattern seen. The graph is symmetrical because we used a +1 reward on either end of the Randomwalk, instead of ± 1 as in [2].

3.7.2 Q-Learning using Linear Function Approximation

Unfortunately, I was unable to complete this section in time. I implemented the feature parametrization code and weight and gradient updates, but ran into the following issues:

- Weights blow up very quickly.
- Clipping the weights helped a little, but not enough to make the values converge correctly.
- Difficult to debug how the weights converging/diverging behavior.
- Not sure how to represent the action-space in the feature dimension. Representing the discrete action space (integer 4, for example) as a final dimension in the features, seems to contribute to the weight blow-up. Not sure how the action space is conventionally represented in the feature space without blowing up the weights.

The essential code for qlearning with features is snippeted here. It is also attached fully in the code listings in the Appendices.

Listing 1: Q-Learning with Linear Features snippet

```
next_state, reward, done, info = env.step(a)
ns = state_fn(env, next_state)

fvec = feat_fn(ns, a)

qval = np.dot(weights, fvec.T)  # current q-val based on fvec, self.weights@fvec
qmax = np.max(np.dot(weights, feat_fn(ns).T))  # max q-val based on ns,np.max(Q[ns,:])
diff = (reward + discount * qmax - qval)
weights += alpha(step,a) * diff * fvec
```

In Figures 10 and 11, I show the weight matrix for a Classic Gridworld environment, a feature space that is the 2x2 tile-coding with overlap=1, and how the weight values diverge. The first figure shows the weights when they are clipped to H (max trajectory length), and the second when they are unclipped (full divergence allowed).

Acknowledgements

Thanks to Professor L. Yang and the TAs for their kind help.

References

[1] O. Nachum, S. Gu, H. Lee, and S. Levine, "Data-efficient hierarchical reinforcement learning," 2018.

[2] Sutton and Barto, "Reinforcement learning,"

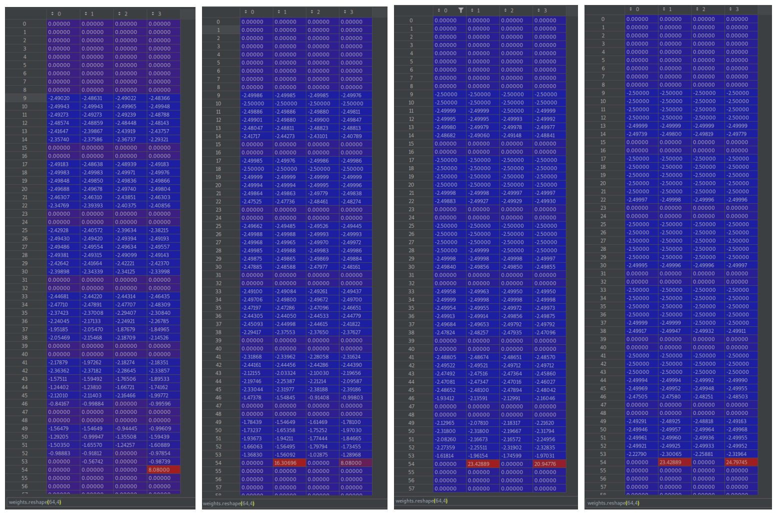
[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[4] M. Chevalier-Boisvert, L. Willems, and S. Pal, "Minimalistic gridworld environment for openai gym." https://github.com/maximecb/gym-minigrid, 2018.

[5] JKCooper2 and ThomasLecat, "Bandit environments." https://github.com/ThomasLecat/gym-bandit-environments.

[6] L. Yang, "Ucla ece239as: Reinforcement learning, lectures,"

[7] R. Vershyn, "High dimensional probability,"



(a)

	0	1	2	3
0	0.00000	0.00000	0.00000	0.00000
1	0.00000	0.00000	0.00000	0.00000
2	0.00000	0.00000	0.00000	0.00000
3	0.00000	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00000
5	0.00000	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00000
7	0.00000	0.00000	0.00000	0.00000
8	0.00000	0.00000	0.00000	0.00000
9	-2.49020	-2.48631	-2.49022	-2.48366
10	-2.49943	-2.49943	-2.49965	-2.49948
11	-2.49273	-2.49273	-2.49239	-2.48788
12	-2.48574	-2.48859	-2.48448	-2.48143
13	-2.41647	-2.39867	-2.43919	-2.43757
14	-2.35740	-2.37586	-2.36737	-2.29321
15	0.00000	0.00000	0.00000	0.00000
16	0.00000	0.00000	0.00000	0.00000
17	-2.49183	-2.48638	-2.48939	-2.49183
18	-2.49983	-2.49983	-2.49971	-2.49976
19	-2.49848	-2.49850	-2.49836	-2.49866
20	-2.49688	-2.49678	-2.49740	-2.49804
21	-2.46307	-2.46310	-2.43851	-2.46303
22	-2.34769	-2.39393	-2.40375	-2.40856
23	0.00000	0.00000	0.00000	0.00000
24	0.00000	0.00000	0.00000	0.00000
25	-2.42928	-2.40572	-2.39634	-2.38215
26	-2.49430	-2.49420	-2.49394	-2.49193
27	-2.49486	-2.49554	-2.49634	-2.49557
28	-2.49381	-2.49315	-2.49099	-2.49143
29	-2.42642	-2.41664	-2.42221	-2.42370
30	-2.39898	-2.34339	-2.34125	-2.33998
31	0.00000	0.00000	0.00000	0.00000
32	0.00000	0.00000	0.00000	0.00000
33	-2.44681	-2.44220	-2.44314	-2.46435
34	-2.47710	-2.47891	-2.47707	-2.48309
35	-2.37423	-2.37008	-2.29407	-2.30840
36	-2.24045	-2.17133	-2.24921	-2.26785
37	-1.95185	-2.05470	-1.87679	-1.84965
38	-2.05469	-2.15468	-2.18709	-2.14526
39	0.00000	0.00000	0.00000	0.00000
40	0.00000	0.00000	0.00000	0.00000
41	-2.17879	-1.97262	-2.18274	-2.18351
42	-2.36362	-2.37182	-2.28645	-2.33857
43	-1.57511	-1.59492	-1.76506	-1.89533
44	-1.24402	-1.23810	-1.66721	-1.74162
45	-2.12010	-2.11403	-2.16466	-1.99772
46	-0.84167	-0.99884	0.00000	-0.99596
47	0.00000	0.00000	0.00000	0.00000
48	0.00000	0.00000	0.00000	0.00000
49	-1.56479	-1.54649	-0.94445	-0.99609
50	-1.29205	-0.99947	-1.35508	-1.59439
51	-1.50350	-1.65570	-1.24257	-1.60889
52	-0.98883	-0.91812	0.00000	-0.97854
53	0.00000	-0.56742	0.00000	-0.98739
54	0.00000	0.00000	0.00000	8.08000
55	0.00000	0.00000	0.00000	0.00000
56	0.00000	0.00000	0.00000	0.00000

weights.reshape(64,4)

(b)

	0	1	2	3
0	0.00000	0.00000	0.00000	0.00000
1	0.00000	0.00000	0.00000	0.00000
2	0.00000	0.00000	0.00000	0.00000
3	0.00000	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00000
5	0.00000	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00000
7	0.00000	0.00000	0.00000	0.00000
8	0.00000	0.00000	0.00000	0.00000
9	-2.49986	-2.49985	-2.49985	-2.49976
10	-2.50000	-2.50000	-2.50000	-2.50000
11	-2.49886	-2.49886	-2.49880	-2.49811
12	-2.49901	-2.49880	-2.49900	-2.49847
13	-2.48047	-2.48811	-2.48823	-2.48813
14	-2.41717	-2.44273	-2.43101	-2.40789
15	0.00000	0.00000	0.00000	0.00000
16	0.00000	0.00000	0.00000	0.00000
17	-2.49985	-2.49976	-2.49986	-2.49986
18	-2.50000	-2.50000	-2.50000	-2.50000
19	-2.49999	-2.49999	-2.49999	-2.49999
20	-2.49994	-2.49994	-2.49995	-2.49996
21	-2.49864	-2.49863	-2.49779	-2.49838
22	-2.47525	-2.47736	-2.48461	-2.48274
23	0.00000	0.00000	0.00000	0.00000
24	0.00000	0.00000	0.00000	0.00000
25	-2.49662	-2.49485	-2.49526	-2.49445
26	-2.49988	-2.49988	-2.49993	-2.49993
27	-2.49968	-2.49965	-2.49970	-2.49972
28	-2.49985	-2.49988	-2.49983	-2.49986
29	-2.49875	-2.49865	-2.49869	-2.49884
30	-2.47885	-2.48588	-2.47977	-2.48161
31	0.00000	0.00000	0.00000	0.00000
32	0.00000	0.00000	0.00000	0.00000
33	-2.49100	-2.49084	-2.49261	-2.49437
34	-2.49706	-2.49800	-2.49672	-2.49700
35	-2.47197	-2.47286	-2.47096	-2.46651
36	-2.44305	-2.44050	-2.44533	-2.44779
37	-2.45093	-2.44998	-2.44615	-2.41822
38	-2.29417	-2.37553	-2.37650	-2.37627
39	0.00000	0.00000	0.00000	0.00000
40	0.00000	0.00000	0.00000	0.00000
41	-2.31868	-2.33962	-2.28058	-2.31624
42	-2.44161	-2.44456	-2.44286	-2.44390
43	-2.12155	-2.03324	-2.10030	-2.19656
44	-2.19746	-2.25387	-2.21214	-2.09587
45	-2.33044	-2.31977	-2.38188	-2.39186
46	-1.47378	-1.54845	-0.91408	-0.99803
47	0.00000	0.00000	0.00000	0.00000
48	0.00000	0.00000	0.00000	0.00000
49	-1.78439	-1.54649	-1.61469	-1.78100
50	-1.73237	-1.65358	-1.75252	-1.97030
51	-1.93673	-1.94211	-1.77444	-1.84665
52	-1.66063	-1.56495	-1.79794	-1.73455
53	-1.36830	-1.56092	-1.02875	-1.28968
54	0.00000	16.30696	0.00000	8.08000
55	0.00000	0.00000	0.00000	0.00000
56	0.00000	0.00000	0.00000	0.00000
57	0.00000	0.00000	0.00000	0.00000

weights.reshape(64,4)

(c)

	0	1	2	3
0	0.00000	0.00000	0.00000	0.00000
1	0.00000	0.00000	0.00000	0.00000
2	0.00000	0.00000	0.00000	0.00000
3	0.00000	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00000
5	0.00000	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00000
7	0.00000	0.00000	0.00000	0.00000
8	0.00000	0.00000	0.00000	0.00000
9	-2.50000	-2.50000	-2.50000	-2.50000
10	-2.50000	-2.50000	-2.50000	-2.50000
11	-2.49999	-2.49999	-2.50000	-2.49999
12	-2.49995	-2.49995	-2.49993	-2.49992
13	-2.49980	-2.49979	-2.49978	-2.49977
14	-2.48682	-2.49060	-2.49148	-2.48841
15	0.00000	0.00000	0.00000	0.00000
16	0.00000	0.00000	0.00000	0.00000
17	-2.50000	-2.50000	-2.50000	-2.50000
18	-2.50000	-2.50000	-2.50000	-2.50000
19	-2.50000	-2.50000	-2.50000	-2.50000
20	-2.50000	-2.50000	-2.50000	-2.50000
21	-2.49998	-2.49998	-2.49997	-2.49997
22	-2.49883	-2.49927	-2.49929	-2.49930
23	0.00000	0.00000	0.00000	0.00000
24	0.00000	0.00000	0.00000	0.00000
25	-2.50000	-2.50000	-2.50000	-2.50000
26	-2.50000	-2.50000	-2.50000	-2.50000
27	-2.50000	-2.50000	-2.50000	-2.50000
28	-2.50000	-2.49999	-2.50000	-2.50000
29	-2.49998	-2.49998	-2.49998	-2.49997
30	-2.49840	-2.49856	-2.49850	-2.49855
31	0.00000	0.00000	0.00000	0.00000
32	0.00000	0.00000	0.00000	0.00000
33	-2.49958	-2.49963	-2.49950	-2.49950
34	-2.49999	-2.49998	-2.49998	-2.49998
35	-2.49954	-2.49955	-2.49972	-2.49973
36	-2.49913	-2.49914	-2.49856	-2.49875
37	-2.49684	-2.49653	-2.49792	-2.49792
38	-2.47824	-2.48257	-2.47935	-2.47096
39	0.00000	0.00000	0.00000	0.00000
40	0.00000	0.00000	0.00000	0.00000
41	-2.48805	-2.48674	-2.48651	-2.48570
42	-2.49522	-2.49521	-2.49712	-2.49712
43	-2.47492	-2.47516	-2.47364	-2.45860
44	-2.47081	-2.47347	-2.47016	-2.46027
45	-2.48652	-2.48100	-2.47894	-2.48042
46	-1.93412	-2.13591	-2.12991	-2.16046
47	0.00000	0.00000	0.00000	0.00000
48	0.00000	0.00000	0.00000	0.00000
49	-2.12965	-2.07810	-2.18317	-2.21620
50	-2.31800	-2.31800	-2.19667	-2.31794
51	-2.08260	-2.16673	-2.16572	-2.24956
52	-2.27359	-2.25511	-2.31902	-2.32835
53	-1.61814	-1.96154	-1.74599	-1.97031
54	0.00000	23.42889	0.00000	20.94776
55	0.00000	0.00000	0.00000	0.00000
56	0.00000	0.00000	0.00000	0.00000
57	0.00000	0.00000	0.00000	0.00000

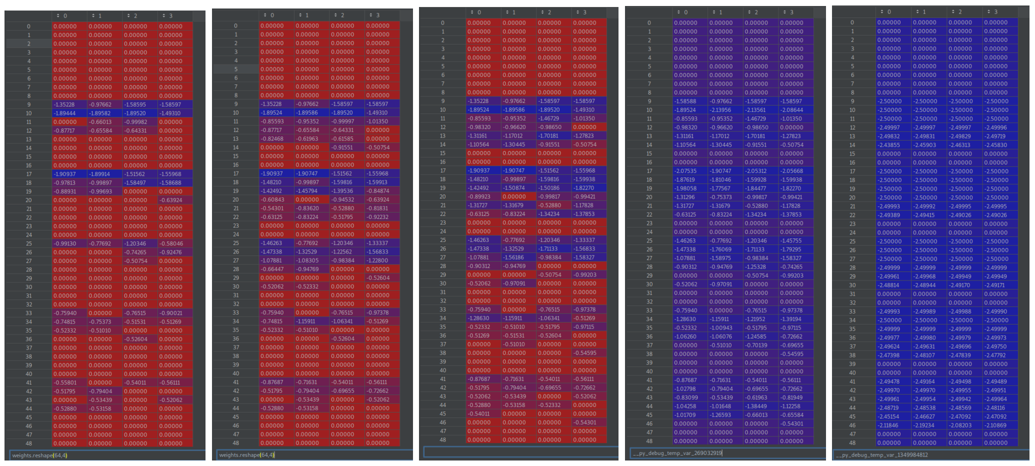
weights.reshape(64,4)

(d)

	0	1	2	3
0	0.00000	0.00000	0.00000	0.00000
1	0.00000	0.00000	0.00000	0.00000
2	0.00000	0.00000	0.00000	0.00000
3	0.00000	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00000
5	0.00000	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00000
7	0.00000	0.00000	0.00000	0.00000
8	0.00000	0.00000	0.00000	0.00000
9	-2.50000	-2.50000	-2.50000	-2.50000
10	-2.50000	-2.50000	-2.50000	-2.50000
11	-2.50000	-2.50000	-2.50000	-2.50000
12	-2.50000	-2.50000	-2.50000	-2.50000
13	-2.49999	-2.49999	-2.49999	-2.49999
14	-2.49739	-2.49800	-2.49819	-2.49779
15	0.00000	0.00000	0.00000	0.00000
16	0.00000	0.00000	0.00000	0.00000
17	-2.50000	-2.50000	-2.50000	-2.50000
18	-2.50000	-2.50000	-2.50000	-2.50000
19	-2.50000	-2.50000	-2.50000	-2.50000
20	-2.50000	-2.50000	-2.50000	-2.50000
21	-2.50000	-2.50000	-2.50000	-2.50000
22	-2.49997	-2.49998	-2.49996	-2.49996
23	0.00000	0.00000	0.00000	0.00000
24	0.00000	0.00000	0.00000	0.00000
25	-2.50000	-2.50000	-2.50000	-2.50000
26	-2.50000	-2.50000	-2.50000	-2.50000
27	-2.50000	-2.50000	-2.50000	-2.50000
28	-2.50000	-2.50000	-2.50000	-2.50000
29	-2.50000	-2.50000	-2.50000	-2.50000
30	-2.49995	-2.49996	-2.49996	-2.49997
31	0.00000	0.00000	0.00000	0.00000
32	0.00000	0.00000	0.00000	0.00000
33	-2.50000	-2.50000	-2.50000	-2.50000
34	-2.50000	-2.50000	-2.50000	-2.50000
35	-2.50000	-2.50000	-2.50000	-2.50000
36	-2.50000	-2.50000	-2.50000	-2.50000
37	-2.49999	-2.49999	-2.50000	-2.50000
38	-2.49917	-2.49947	-2.49932	-2.49911
39	0.00000	0.00000	0.00000	0.00000
40	0.00000	0.00000	0.00000	0.00000
41	-2.50000	-2.50000	-2.50000	-2.50000
42	-2.50000	-2.50000	-2.50000	-2.50000
43	-2.50000	-2.50000	-2.50000	-2.50000
44	-2.49994	-2.49994	-2.49992	-2.49990
45	-2.49969	-2.49952	-2.49948	-2.49955
46	-2.47505	-2.47580	-2.48251	-2.48503
47	0.00000	0.00000	0.00000	0.00000
48	0.00000	0.00000	0.00000	0.00000
49	-2.49291	-2.48925	-2.48818	-2.49163
50	-2.49946	-2.49957	-2.49964	-2.49968
51	-2.49961	-2.49960	-2.49936	-2.49955
52	-2.49921	-2.49925	-2.49933	-2.49952
53	-2.22790	-2.30065	-2.25881	-2.31964
54	0.00000	23.42889	0.00000	24.79745
55	0.00000	0.00000	0.00000	0.00000
56	0.00000	0.00000	0.00000	0.00000
57	0.00000	0.00000	0.00000	0.00000

weights.reshape(64,4)

Figure 10: Weights for Q-Learning, Classic Gridworld Empty Room 6x6. Clipping On. (a) Ep. 30 (b) Ep. 54 (c) Ep. 102 (d) Ep. 1000



(a)

	0	1	2	3
0	0.00000	0.00000	0.00000	0.00000
1	0.00000	0.00000	0.00000	0.00000
2	0.00000	0.00000	0.00000	0.00000
3	0.00000	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00000
5	0.00000	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00000
7	0.00000	0.00000	0.00000	0.00000
8	0.00000	0.00000	0.00000	0.00000
9	-1.35228	-0.97662	-1.58595	-1.58597
10	-1.89444	-1.89582	-1.89520	-1.49310
11	0.00000	-0.66013	-0.99982	0.00000
12	-0.87717	-0.65584	-0.64331	0.00000
13	0.00000	0.00000	0.00000	0.00000
14	0.00000	0.00000	0.00000	0.00000
15	0.00000	0.00000	0.00000	0.00000
16	0.00000	0.00000	0.00000	0.00000
17	-1.90937	-1.89914	-1.51562	-1.55968
18	-0.97813	-0.99897	-1.58497	-1.58688
19	-0.88931	-0.99693	0.00000	0.00000
20	0.00000	0.00000	0.00000	-0.63924
21	0.00000	0.00000	0.00000	0.00000
22	0.00000	0.00000	0.00000	0.00000
23	0.00000	0.00000	0.00000	0.00000
24	0.00000	0.00000	0.00000	0.00000
25	-0.99130	-0.77692	-1.20346	-0.58046
26	0.00000	0.00000	-0.74265	-0.92476
27	0.00000	0.00000	-0.50754	0.00000
28	0.00000	0.00000	0.00000	0.00000
29	0.00000	0.00000	0.00000	0.00000
30	0.00000	0.00000	0.00000	0.00000
31	0.00000	0.00000	0.00000	0.00000
32	0.00000	0.00000	0.00000	0.00000
33	-0.75940	0.00000	-0.76515	-0.90021
34	-0.74815	-0.75373	-0.51531	-0.51269
35	-0.52332	-0.51010	0.00000	0.00000
36	0.00000	0.00000	-0.52604	0.00000
37	0.00000	0.00000	0.00000	0.00000
38	0.00000	0.00000	0.00000	0.00000
39	0.00000	0.00000	0.00000	0.00000
40	0.00000	0.00000	0.00000	0.00000
41	-0.55801	0.00000	-0.54011	-0.56111
42	-0.51795	-0.79404	0.00000	0.00000
43	0.00000	-0.53439	0.00000	-0.52062
44	-0.52880	-0.53158	0.00000	0.00000
45	0.00000	0.00000	0.00000	0.00000
46	0.00000	0.00000	0.00000	0.00000
47	0.00000	0.00000	0.00000	0.00000
48	0.00000	0.00000	0.00000	0.00000

weights.reshape(64,4)

(b)

	0	1	2	3
0	0.00000	0.00000	0.00000	0.00000
1	0.00000	0.00000	0.00000	0.00000
2	0.00000	0.00000	0.00000	0.00000
3	0.00000	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00000
5	0.00000	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00000
7	0.00000	0.00000	0.00000	0.00000
8	0.00000	0.00000	0.00000	0.00000
9	-1.35228	-0.97662	-1.58597	-1.58597
10	-1.89524	-1.89586	-1.89520	-1.49310
11	-0.85593	-0.95352	-0.99997	-1.01350
12	-0.87717	-0.65584	-0.64331	0.00000
13	-0.82468	-0.61963	-0.61585	0.00000
14	0.00000	0.00000	-0.91551	-0.50754
15	0.00000	0.00000	0.00000	0.00000
16	0.00000	0.00000	0.00000	0.00000
17	-1.90937	-1.90747	-1.51562	-1.55968
18	-1.48210	-0.99897	-1.59816	-1.59913
19	-1.42492	-1.45794	-1.39536	-0.84874
20	-0.60843	0.00000	-0.94532	-0.63924
21	-0.54301	-0.83620	-0.52880	-0.81831
22	-0.63125	-0.83224	-0.51795	-0.92232
23	0.00000	0.00000	0.00000	0.00000
24	0.00000	0.00000	0.00000	0.00000
25	-1.46263	-0.77692	-1.20346	-1.33337
26	-1.47338	-1.32529	-1.22562	-1.56833
27	-1.07881	-1.08305	-0.98384	-1.22800
28	-0.66447	-0.94769	0.00000	0.00000
29	0.00000	0.00000	0.00000	-0.52604
30	-0.52062	-0.52332	0.00000	0.00000
31	0.00000	0.00000	0.00000	0.00000
32	0.00000	0.00000	0.00000	0.00000
33	-0.75940	0.00000	-0.76515	-0.97378
34	-0.74815	-1.15911	-1.06341	-0.51269
35	-0.52332	-0.51010	0.00000	0.00000
36	0.00000	0.00000	-0.52604	0.00000
37	0.00000	0.00000	0.00000	0.00000
38	0.00000	0.00000	0.00000	0.00000
39	0.00000	0.00000	0.00000	0.00000
40	0.00000	0.00000	0.00000	0.00000
41	-0.87687	-0.71631	-0.54011	-0.56111
42	-0.51795	-0.79404	-0.69655	-0.72662
43	0.00000	-0.53439	0.00000	-0.52062
44	-0.52880	-0.53158	0.00000	0.00000
45	0.00000	0.00000	0.00000	0.00000
46	0.00000	0.00000	0.00000	0.00000
47	0.00000	0.00000	0.00000	0.00000
48	0.00000	0.00000	0.00000	0.00000

weights.reshape(64,4)

(c)

	0	1	2	3
0	0.00000	0.00000	0.00000	0.00000
1	0.00000	0.00000	0.00000	0.00000
2	0.00000	0.00000	0.00000	0.00000
3	0.00000	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00000
5	0.00000	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00000
7	0.00000	0.00000	0.00000	0.00000
8	0.00000	0.00000	0.00000	0.00000
9	-1.35228	-0.97662	-1.58597	-1.58597
10	-1.89524	-1.89586	-1.89520	-1.49310
11	-0.85593	-0.95352	-1.46729	-1.01350
12	-0.98320	-0.96620	-0.98650	0.00000
13	-1.31161	-1.17012	-1.70181	-1.27823
14	-1.10564	-1.30445	-0.91551	-0.50754
15	0.00000	0.00000	0.00000	0.00000
16	0.00000	0.00000	0.00000	0.00000
17	-1.90937	-1.90747	-1.51562	-1.55968
18	-1.48210	-0.99897	-1.59816	-1.59938
19	-1.42492	-1.50874	-1.50186	-1.82270
20	-0.89923	0.00000	-0.99817	-0.99421
21	-1.31727	-1.31679	-0.52880	-1.17828
22	-0.63125	-0.83224	-1.34234	-1.37853
23	0.00000	0.00000	0.00000	0.00000
24	0.00000	0.00000	0.00000	0.00000
25	-1.46263	-0.77692	-1.20346	-1.33337
26	-1.47338	-1.32529	-1.71133	-1.56833
27	-1.07881	-1.56186	-0.98384	-1.58327
28	-0.90312	-0.94769	0.00000	0.00000
29	0.00000	0.00000	-0.50754	-0.99203
30	-0.52062	-0.97091	0.00000	0.00000
31	0.00000	0.00000	0.00000	0.00000
32	0.00000	0.00000	0.00000	0.00000
33	-0.75940	0.00000	-0.76515	-0.97378
34	-1.28630	-1.15911	-1.06341	-0.51269
35	-0.52332	-0.51010	-0.51795	-0.97115
36	-0.51269	-0.51531	-0.52604	0.00000
37	0.00000	-0.51010	0.00000	0.00000
38	0.00000	0.00000	0.00000	-0.54595
39	0.00000	0.00000	0.00000	0.00000
40	0.00000	0.00000	0.00000	0.00000
41	-0.87687	-0.71631	-0.54011	-0.56111
42	-0.51795	-0.79404	-0.69655	-0.72662
43	-0.52062	-0.53439	0.00000	-0.52062
44	-0.52880	-0.53158	-0.52332	0.00000
45	-0.54011	0.00000	0.00000	0.00000
46	0.00000	0.00000	0.00000	-0.54301
47	0.00000	0.00000	0.00000	0.00000
48	0.00000	0.00000	0.00000	0.00000

(d)

	0	1	2	3
0	0.00000	0.00000	0.00000	0.00000
1	0.00000	0.00000	0.00000	0.00000
2	0.00000	0.00000	0.00000	0.00000
3	0.00000	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00000
5	0.00000	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00000
7	0.00000	0.00000	0.00000	0.00000
8	0.00000	0.00000	0.00000	0.00000
9	-1.58588	-0.97662	-1.58597	-1.58597
10	-1.89524	-2.13956	-2.13561	-2.08644
11	-0.85593	-0.95352	-1.46729	-1.01350
12	-0.98320	-0.96620	-0.98650	0.00000
13	-1.31161	-1.17012	-1.70181	-1.27823
14	-1.10564	-1.30445	-0.91551	-0.50754
15	0.00000	0.00000	0.00000	0.00000
16	0.00000	0.00000	0.00000	0.00000
17	-2.07535	-1.90747	-2.05312	-2.05668
18	-1.87619	-1.81046	-1.59928	-1.59938
19	-1.98058	-1.77567	-1.84477	-1.82270
20	-1.31296	-0.75373	-0.99817	-0.99421
21	-1.31727	-1.31679	-0.52880	-1.17828
22	-0.63125	-0.83224	-1.34234	-1.37853
23	0.00000	0.00000	0.00000	0.00000
24	0.00000	0.00000	0.00000	0.00000
25	-1.46263	-0.77692	-1.20346	-1.45755
26	-1.47338	-1.76069	-1.71133	-1.79295
27	-1.07881	-1.58975	-0.98384	-1.58327
28	-0.90312	-0.94769	-1.25328	-0.74265
29	0.00000	0.00000	-0.50754	-0.99203
30	-0.52062	-0.97091	0.00000	0.00000
31	0.00000	0.00000	0.00000	0.00000
32	0.00000	0.00000	0.00000	0.00000
33	-0.75940	0.00000	-0.76515	-0.97378
34	-1.28630	-1.15911	-1.21952	-1.39194
35	-0.52332	-1.00943	-0.51795	-0.97115
36	-1.06260	-1.06076	-1.24585	-0.72662
37	0.00000	-0.51010	-0.70139	-0.69655
38	0.00000	0.00000	0.00000	-0.54595
39	0.00000	0.00000	0.00000	0.00000
40	0.00000	0.00000	0.00000	0.00000
41	-0.87687	-0.71631	-0.54011	-0.56111
42	-1.02798	-0.79404	-0.69655	-0.72662
43	-0.83099	-0.53439	-0.61963	-0.81949
44	-1.04258	-1.01648	-1.38449	-1.12258
45	-1.01709	-1.26593	-0.66013	-0.65584
46	0.00000	0.00000	0.00000	-0.54301
47	0.00000	0.00000	0.00000	0.00000
48	0.00000	0.00000	0.00000	0.00000

__py_debug_temp_var_269032919

(e)

	0	1	2	3
0	0.00000	0.00000	0.00000	0.00000
1	0.00000	0.00000	0.00000	0.00000
2	0.00000	0.00000	0.00000	0.00000
3	0.00000	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00000
5	0.00000	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00000
7	0.00000	0.00000	0.00000	0.00000
8	0.00000	0.00000	0.00000	0.00000
9	-2.50000	-2.50000	-2.50000	-2.50000
10	-2.50000	-2.50000	-2.50000	-2.50000
11	-2.50000	-2.50000	-2.50000	-2.50000
12	-2.49997	-2.49997	-2.49997	-2.49996
13	-2.49832	-2.49831	-2.49829	-2.49719
14	-2.43855	-2.45903	-2.46313	-2.45830
15	0.00000	0.00000	0.00000	0.00000
16	0.00000	0.00000	0.00000	0.00000
17	-2.50000	-2.50000	-2.50000	-2.50000
18	-2.50000	-2.50000	-2.50000	-2.50000
19	-2.50000	-2.50000	-2.50000	-2.50000
20	-2.50000	-2.50000	-2.50000	-2.50000
21	-2.49993	-2.49992	-2.49995	-2.49995
22	-2.49389	-2.49415	-2.49026	-2.49026
23	0.00000	0.00000	0.00000	0.00000
24	0.00000	0.00000	0.00000	0.00000
25	-2.50000	-2.50000	-2.50000	-2.50000
26	-2.50000	-2.50000	-2.50000	-2.50000
27	-2.50000	-2.50000	-2.50000	-2.50000
28	-2.49999	-2.49999	-2.49999	-2.49999
29	-2.49961	-2.49968	-2.49949	-2.49949
30	-2.48814	-2.48944	-2.49170	-2.49171
31	0.00000	0.00000	0.00000	0.00000
32	0.00000	0.00000	0.00000	0.00000
33	-2.49993	-2.49989	-2.49988	-2.49990
34	-2.50000	-2.50000	-2.50000	-2.50000
35	-2.49999	-2.49999	-2.49999	-2.49999
36	-2.49977	-2.49980	-2.49979	-2.49973
37	-2.49624	-2.49631	-2.49696	-2.49750
38	-2.47398	-2.48107	-2.47839	-2.47792
39	0.00000	0.00000	0.00000	0.00000
40	0.00000	0.00000	0.00000	0.00000
41	-2.49478	-2.49164	-2.49498	-2.49489
42	-2.49970	-2.49970	-2.49955	-2.49951
43	-2.49961	-2.49954	-2.49942	-2.49964
44	-2.48719	-2.48538	-2.48569	-2.48116
45	-2.45154	-2.46627	-2.47092	-2.47092
46	-2.11846	-2.19234	-2.08203	-2.10869
47	0.00000	0.00000	0.00000	0.00000
48	0.00000	0.00000	0.00000	0.00000

__py_debug_temp_var_1349984812

Figure 11: Weights for Q-Learning, Classic Gridworld Empty Room 6x6. Clipping Off. (a) Ep. 2 (b) Ep. 3 (c) Ep. 4 (d) Ep. 5 (e) Ep. 130

[8] C. Jin, Z. Allen-Zhu, S. Bubeck, and M. I. Jordan, “Is q-learning provably efficient?,” 2018.

[9] S. Kakade, M. Wang, and L. F. Yang, “Variance reduction methods for sublinear reinforcement learning,” 2020.

A Appendix

A.1 Algorithms

Algorithms are from [6] and [2].

Algorithm 1: Q-Learning (TD-0 target)

```
**Result:** $Q(s,a)$, the estimated q-values
initialize $Q(s,a),\ \forall s \in \mathcal{S}, a \in \mathcal{A}$ arbitrarily, $Q(\text{terminal-state},\cdot) = 0$
**repeat** for each step of episode
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
    Take action $A$, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma max_a Q(S',a) - Q(S,A)]$
    $S \leftarrow S'$
**until** $S$ *is terminal*;
```

Algorithm 2: Sarsa (TD-0 target)

```
**Result:** $Q(s,a)$, the estimated q-values
initialize $Q(s,a),\ \forall s \in \mathcal{S}, a \in \mathcal{A}$ arbitrarily, $Q(\text{terminal-state},\cdot) = 0$
**repeat** for each episode
    initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
    **repeat** for each step of episode
        Take action $A$, observe R, S'
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
        $S \leftarrow S'$; $A \leftarrow A'$
**until** $S$ *is terminal*;
```

Algorithm 3: Sarsa-λ (Backwards-view)

```
**Result:** $Q(s,a)$, estimated q-values
initialize $Q(s,a),\ \forall s \in \mathcal{S}, a \in \mathcal{A}$ arbitrarily
**repeat** for each episode
    $E(S,A) = 0$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}$
    initialize $S$, $A$
    **repeat** for each step of episode
        Take action $A$, observe R, S'
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        $\delta \leftarrow R + \gamma Q(S',A') - Q(S,A)$
        $E(S,A) \leftarrow E(S,A) + 1$
        **forall** $s \in \mathcal{S}$, $a \in \mathcal{A}$ **do**
            $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
            $E(s,a) \leftarrow \gamma\lambda E(s,a)$
        **end**
        $S \leftarrow S'$; $A \leftarrow A'$
**until** $S$ *is terminal*;
```

Algorithm 4: Simple-Bandit Algorithm

```
Result: $Q(s,a)$, the estimated q-values
initialize $Q(a) = 0,\ \forall a \in \mathcal{A}$
initialize $N(a) = 0,\ \forall a \in \mathcal{A}$
repeat
    $T(n) = \begin{cases} \text{argmax}_a Q(a) & \text{with probability } 1-\epsilon \text{ (breaking ties randomly)} \\ \text{a random action} & \text{with probability } \epsilon \end{cases}$
    $R \leftarrow bandit(A)$
    $N(A) \leftarrow N(A) + 1$
    $Q(A) \leftarrow Q(A) + \frac{1}{N(A)}[R - Q(A)]$
```

Algorithm 5: Simple-Bandit Algorithm with UCB

```
Result: $Q(s,a)$, the estimated q-values
initialize $Q(a) = 0,\ \forall a \in \mathcal{A}$
initialize $N(a) = 0,\ \forall a \in \mathcal{A}$
repeat
    $T(n) = \begin{cases} \text{argmax}_a Q(a) + c\sqrt{\frac{lnt}{N_t(a)}} & \text{with probability } 1-\epsilon \text{ (breaking ties randomly)} \\ \text{a random action} & \text{with probability } \epsilon \end{cases}$
    $R \leftarrow bandit(A)$
    $N(A) \leftarrow N(A) + 1$
    $Q(A) \leftarrow Q(A) + \frac{1}{N(A)}[R - Q(A)]$
(where: c represents the confidence level and can be treated as a hyperparameter)
```

Algorithm 6: Q-Learning with UCB-Hoeffding [8]

```
Result: $Q_h(s,a)$, the estimated q-values
initialize $Q_h(s,a) \leftarrow H$ and $N_h(s,a) \leftarrow 0$ for all $(s,a,h) \in \mathcal{S} \times \mathcal{A} \times H$;
for episode k = 1,...,K do
    receive $s_1$;
    for step h = 1,...,H do
        Take action $a_h \leftarrow argmax_{a'} Q_h(s_h, a')$, and observe $s_{h+1}$
        $t = N_h(s_h, a_h) \leftarrow N_h(s_h, a_h) + 1; b_t \leftarrow c\sqrt{H^3\iota/t}$
        $Q_h(s_h, a_h) \leftarrow (1-\alpha_t)Q_h(s_h, a_h) + \alpha_t[r_h(s_h, a_h) + V_{h+1}(s_{h+1}) + b_t]$
        $V_h(s_h) \leftarrow min(H, max_{a' \in \mathcal{A}} Q_h(s_h, a'))$
    end
end
(where: $H^3$ and $\iota$ can be treated as hyperparameters in practice and $\alpha_t := \frac{H+1}{H+t}$)
```

Algorithm 7: Gradient-Bandit Algorithm [2]

```
Result: $Q(s, a)$, the estimated q-values
initialize $Q(a) = 0, N(a) = 0, H(a) = 0, \forall a \in \mathcal{A}$
initialize $\bar{R} = 0$
repeat
    $\pi(A) \leftarrow \frac{e^H(A)}{\sum e^H(A)}$
    $A \leftarrow argmax_{a' \in \mathcal{A}}(\pi(a'))$
    $R \leftarrow bandit(A)$
    $\bar{R} \leftarrow \bar{R} + R$
    $N(A) \leftarrow N(A) + 1$
    $Q(A) \leftarrow Q(A) + \frac{1}{N(A)}[R - Q(A)]$
    forall $a' \in \mathcal{A}$ do
        $H(a') \leftarrow \begin{cases} \text{H(a') + } \alpha(1 - \pi(a')) & \text{if } a = A \\ \text{H(a') - } \alpha(R - \frac{\bar{R}}{k+1})\pi(a') & \text{if } a \neq A \end{cases}$
    end
(where: c represents the confidence level and can be treated as a hyperparameter)
```

Algorithm 8: MC-Control using $\text{MC}_{\text{first-visit}}$ or $\text{MC}_{\text{every-visit}}$ as PE.

```
Result: $Q(s, a)$, the estimated q-values
initialize $\pi^0$, randomly
repeat for each step of episode
    Play an episode and collect trajectory $S_0, A_0, R_0, S_1, A_1, R_1, ...(S, A, R, S')$
    Update $Q(S, A)$ according to MC-PE: $Q(S, A) \leftarrow C(S, A)/N(S, A)$
    Update policy $\pi$: $\pi \leftarrow \epsilon$-greedy($Q(s, a)$)
```

Algorithm 9: Q-Learning using Linear Function Approximation and TD-0 Target

```
Result: $\vec{w}$, the estimated weights parametrizing $Q_w$
initialize weights $\vec{w} \in \mathbf{R}^{\text{dim}(\phi)}$ arbitrarily
repeat for each episode
    Initialize $S$ (received from env)
    repeat for each step of episode
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        Take action $A$, observe R, S'
        $Q_{w^t} \leftarrow \vec{w} \bullet \phi(s', a)$
        $Q_{\max, w^t} \leftarrow argmax_{a' \in \mathcal{A}}(\vec{w} \bullet \phi(S', a'))$
        $\hat{g}_{\vec{w}^t} \leftarrow (R + \gamma Q_{\max, w^t} - Q_{w^t}) \cdot \nabla_w Q_{w^{t-1}}(s, a)$
        $w_t \leftarrow w_{t-1} - \eta_t \cdot \hat{g}_{w^{t-1}}$
        $S \leftarrow S'$
until $S$ is terminal;
(where: 1. $\phi$ is the chosen feature vector space, 2. in practice, $\nabla_w Q_{w^{t-1}}(s, a) = \phi(s, a)$)
```

A.2 Code

The following algorithmic implementations of the various RL algorithms were written by myself (Rach Liu).

A.2.1 Q-Learning

Author: Rach Liu

Listing 2: Regular Q-Learning for tabular

```
def qlearn(env, num_eps, H, Q, N, state_fn, alpha, epsilon, discount, enable_legal=False):
    """
    Epsilon is a scalar (constant epsilon) or a 3-tuple specifying an exponentially-decaying
    epsilon: (max_eps, min_eps, decay-rate).
    """
    ep_lengths = np.zeros(num_eps)
    ep_rewards = np.zeros((num_eps, H))
    ep_opt_axn = np.zeros((num_eps, H))
    ep_epsilons = np.zeros(num_eps)

    decay = None
    if type(epsilon) is tuple:
        # decaying epsilon
        max_epsilon = epsilon[0]
        min_epsilon = epsilon[1]
        decay = epsilon[2]
        epsilon = max_epsilon

    for ep in range(num_eps):
        state = env.reset()
        s = state_fn(env, state)

        rewards = np.zeros(H)
        opt_axn = np.zeros(H)

        step = 0
        for step in range(H):
            pr = random.uniform(0, 1)

            if enable_legal:
                try:
                    if pr > epsilon:
                        # info is a list of legal actions, argmax is the idx of this list, not in Q
                        a_max_idx = np.argmax(Q[s, info])
                        a = info[a_max_idx]
                    else:
                        a = np.random.choice(info)
                except NameError:
                    if pr > epsilon:
                        a = np.argmax(Q[s, :])
                    else:
                        a = env.action_space.sample()
            else:
                if pr > epsilon:
                    a = np.argmax(Q[s, :])
                else:
                    a = env.action_space.sample()

            next_state, reward, done, info = env.step(a)
            ns = state_fn(env, next_state)

            N[s, a] += 1
            Q[s, a] = Q[s, a] + alpha(step, a) * (
                    reward + discount * np.max(Q[ns, :]) - Q[s, a])

            rewards[step] += reward
            if 'optimal_action' in info:
                opt_axn[step] = a == info['optimal_action']

            if done == True:
                # print ("Total reward for episode {}: {}".format(ep, total_rewards))
                break

            s = ns


        if decay is not None:
            epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-decay * ep)

        ep_lengths[ep] = step+1  # convert to 1-indexed
        ep_rewards[ep] = rewards
        ep_opt_axn[ep] = opt_axn
        ep_epsilons[ep] = epsilon

    return ep_lengths, ep_rewards, ep_opt_axn, ep_epsilons
```

Listing 3: Q-Learning with UCB-Hoeffding for tabular

```
def qlearn_ucb1(env, num_eps, H, Q, N, state_fn, alpha, bonus, discount):
    """
    Bonus is specified as a function that accepts the current state and an action, (s,a).
    """
    ep_lengths = np.zeros(num_eps)
    ep_rewards = np.zeros((num_eps, H))
    ep_opt_axn = np.zeros((num_eps, H))

    for ep in range(num_eps):
        state = env.reset()
        s = state_fn(env, state)

        rewards = np.zeros(H)
        opt_axn = np.zeros(H)

        step = 0
        for step in range(H):
            q_max = None
            a_max = None
            for a in range(0, env.action_space.n):
                if N[s,a] == 0:
                    a_max = a
                    break
                else:
                    q = Q[s, a] + bonus(step, s, a)
                    if q_max is None or q_max < q:
                        q_max = q
                        a_max = a

            next_state, reward, done, info = env.step(a_max)
            ns = state_fn(env, next_state)

            N[s, a_max] += 1
            b = bonus(step, s, a_max)
            Q[s, a_max] = Q[s, a_max] + alpha(step, a_max) * (
                    reward + discount * np.max(Q[ns, :]) - Q[s, a_max] + b)

            Q[s] = np.clip(Q[s], a_min=None, a_max=H)

            rewards[step] += reward
            if 'optimal_action' in info:
                opt_axn[step] = a == info['optimal_action']

            s = ns

            if done == True:
                # print ("Total reward for episode {}: {}".format(ep, total_rewards))
                break

        ep_lengths[ep] = step+1  # convert to 1-indexed
        ep_rewards[ep] = rewards
        ep_opt_axn[ep] = opt_axn

    return ep_lengths, ep_rewards, ep_opt_axn
```

Listing 4: Q-Learning using Linear Function Approximation

```
def qlearn_cont(env, num_eps, H, weights, state_fn, feat_fn, alpha, epsilon, discount):
    """

    """
    ep_lengths = np.zeros(num_eps)
    ep_rewards = np.zeros((num_eps, H))
    ep_opt_axn = np.zeros((num_eps, H))
    ep_epsilons = np.zeros(num_eps)

    decay = None
    if type(epsilon) is tuple:
        # decaying epsilon
        max_epsilon = epsilon[0]
        min_epsilon = epsilon[1]
        decay = epsilon[2]
        epsilon = max_epsilon

    for ep in range(num_eps):
        state = env.reset()
        s = state_fn(env, state)

        rewards = np.zeros(H)
        opt_axn = np.zeros(H)

        step = 0
        for step in range(H):
            pr = random.uniform(0, 1)

            if pr > epsilon:
                a = np.argmax(np.dot(weights, feat_fn(s).T))  # === np.argmax(Q[s,:])
            else:
                a = env.action_space.sample()

            next_state, reward, done, info = env.step(a)
            ns = state_fn(env, next_state)

            """
            gradient update is -2*(Q^pi(s,a) - Q_w(s,a)) * grad_w(Q_w(s,a))
            where Q^pi(s,a) is the ideal Q-value we are approximating; substitute with different
               estimators depending on method. Here, we substitute it with our td0 q-learning estim.
            and Q_w(s,a) is the Q-function parametrized by weights, with value equal to
               dotproduct of weights with feature vector: w_t dot phi(s,a)
            """
            fvec = feat_fn(ns, a)

            clip = True
            if clip:
                clip_to = H
                qval = np.minimum(clip_to, np.dot(weights, fvec.T))  # current q-val based on fvec, self.weights@fvec
                qmax = np.minimum(clip_to,np.max(np.dot(weights, feat_fn(ns).T)))
# max q-val based on ns,np.max(Q[ns,:])
            else:
                qval = np.dot(weights, fvec.T)  # current q-val based on fvec, self.weights@fvec
                qmax = np.max(np.dot(weights, feat_fn(ns).T))  # max q-val based on ns,np.max(Q[ns,:])
            diff = (reward + discount * qmax - qval)
            weights += alpha(step,a) * diff * fvec

            rewards[step] += reward
            if 'optimal_action' in info:
                opt_axn[step] = a == info['optimal_action']

            if done == True:
                # print ("Total reward for episode {}: {}".format(ep, total_rewards))
                break

            s = ns

        if decay is not None:
            epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-decay * ep)

        ep_lengths[ep] = step+1  # convert to 1-indexed
        ep_rewards[ep] = rewards
        ep_opt_axn[ep] = opt_axn
        ep_epsilons[ep] = epsilon

    return ep_lengths, ep_rewards, ep_opt_axn, ep_epsilons
```

A.2.2 Sarsa

Author: Rach Liu

Listing 5: Regular Sarsa for tabular

```
def sarsa(env, num_eps, H, Q, state_fn, alpha, epsilon, discount):
    """
    regular sarsa
    :param Q: must initialize the q-table before hand to size of s, a
    :param state_fn: convert the raw env state to something else; set as identity to ignore
    :param: epsilon is a scalar (constant epsilon) or a 3-tuple specifying an exponentially-decaying
    epsilon: (max_eps, min_eps, decay-rate).
    """
    ep_lengths = np.zeros(num_eps)
    ep_rewards = np.zeros((num_eps, H))
    ep_epsilons = np.zeros(num_eps)

    decay = None
    if type(epsilon) is tuple:
        # decaying epsilon
        max_epsilon = epsilon[0]
        min_epsilon = epsilon[1]
        decay = epsilon[2]
        epsilon = max_epsilon


    for ep in range(num_eps):
        state = env.reset()
        s = state_fn(env, state)

        rewards = np.zeros(H)

        pr = random.uniform(0, 1)
        if pr > epsilon:
            a = np.argmax(Q[s, :])
        else:
            a = env.action_space.sample()

        step = 0
        for step in range(H-1):

            next_state, reward, done, info = env.step(a)
            ns = state_fn(env, next_state)

            pr = random.uniform(0, 1)
            if pr > epsilon:
                na = np.argmax(Q[ns, :])
            else:
                na = env.action_space.sample()

            Q[s, a] = Q[s, a] + alpha(step, a) * (
                    reward + discount * Q[ns, na] - Q[s, a])

            s = ns  # start at s' (ns) on the next iteration
            a = na  # and take action na (a') from ns (s') on next iter, as already chosen by policy

            rewards[step] += reward

            if done:
                # print ("Total reward for ep {}: {}".format(ep, rewards))
                break

        if decay is not None:
            epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-decay * ep)

        ep_lengths[ep] = step+1  # convert to 1-indexed
        ep_rewards[ep] = rewards
        ep_epsilons[ep] = epsilon

    return ep_lengths, ep_rewards, ep_epsilons
```

Listing 6: Sarsa-λ for tabular

```
def sarsa_lambda(env, num_eps, H, Q, E, lmbda, state_fn, alpha, epsilon, discount):
    """
    sarsa-lambda using backwards view via eligibility traces
    :param Q: must initialize the q-table before hand to size of s, a
    :param E: must initialize the eligibility trace before hand to size of s, a (same space as q)
    :param state_fn: convert the raw env state to something else; set as identity to ignore
    :param: epsilon is a scalar (constant epsilon) or a 3-tuple specifying an exponentially-decaying
    epsilon: (max_eps, min_eps, decay-rate).
    """
    ep_lengths = np.zeros(num_eps)
    ep_rewards = np.zeros((num_eps, H))
    ep_epsilons = np.zeros(num_eps)

    decay = None
    if type(epsilon) is tuple:
        # decaying epsilon
        max_epsilon = epsilon[0]
        min_epsilon = epsilon[1]
        decay = epsilon[2]
        epsilon = max_epsilon

    for ep in range(num_eps):
        state = env.reset()
        s = state_fn(env, state)

        rewards = np.zeros(H)

        pr = random.uniform(0, 1)
        if pr > epsilon:
            a = np.argmax(Q[s, :])
        else:
            a = env.action_space.sample()

        step = 0
        for step in range(H-1):

            next_state, reward, done, info = env.step(a)
            ns = state_fn(env, next_state)

            pr = random.uniform(0, 1)
            if pr > epsilon:
                na = np.argmax(Q[ns, :])
            else:
                na = env.action_space.sample()

            td0_delta = reward + discount * Q[ns, na] - Q[s, a]
            E[s, a] = E[s, a] + 1  # incr elig tr of current (s,a) by 1

            # iterate over all s, then all a in each s; if we assume all a are eligible in all s,
            #   then this is covered by entire space of Q; this is an exp decay through all
            #   states,actions
            for s in range(0, Q.shape[0]):
                for a in range(Q.shape[1]):
                    Q[s, a] = Q[s, a] + alpha(step, a) * td0_delta * E[s, a]
                    E[s, a] = discount * lmbda * E[s, a]  # decay elig tr of all states by gamma*lmbda

            s = ns  # start at s' (ns) on the next iteration
            a = na  # and take action na (a') from ns (s') on next iter, as already chosen by policy

            rewards[step] += reward

            if done:
                # print ("Total reward for ep {}: {}".format(ep, rewards))
                break

        if decay is not None:
            epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-decay * ep)

        ep_lengths[ep] = step+1  # convert to 1-indexed
        ep_rewards[ep] = rewards
        ep_epsilons[ep] = epsilon

    return ep_lengths, ep_rewards, ep_epsilons
```

A.2.3 MC-Control with 1st-visit and Every-visit MC

Author: Rach Liu

Listing 7: MC-Control using First-Visit MC

```
def mc_first_visit(env, num_eps, state_fn, H, Q, N, C, epsilon, discount):
    """
    Derives an epsilon-greedy policy from Q, and then estimates updated Q-values using
    first-visit MC on trajectories sampled from the epsilon-greedy derived policy.
    Thus, this performs one step of Policy Iteration: Policy extraction, then Policy evaluation,
    then Policy update (implicitly, through update of Q, which will then cause a new policy to
    be extracted upon the next call).
    Q, N are the q-value and n-value estimators (functions of (s,a)); the Q estimator is just the
    estimated Q-value (expected rewards from (s,a)) and N estimator is the number of visits to
    state-action pair (s,a).
    """

    ep_lengths = np.zeros(num_eps, dtype='int')
    ep_trajectories = np.zeros((num_eps,H,4))

    for ep in range(num_eps):
        state = env.reset()
        s = state_fn(env, state)

        trajectory = np.zeros((H,4))
        step = 0
        for step in range(H):
            pr = random.uniform(0, 1)

            if pr < epsilon:
                a = env.action_space.sample()
            else:
                a = np.argmax(Q[s, :])  # implicit use of Q as the policy function Pi

            next_state, reward, done, info = env.step(a)
            ns = state_fn(env, next_state)

            trajectory[step] = s, a, reward, ns

            s = ns

            if done == True:
                break

        ep_trajectories[ep] = trajectory
        ep_lengths[ep] = step+1  # convert to 1-indexed

    # Precompute discount factors: 1, gamma, gamma^2, gamma^3, ..., gamma^H
    Gammas = np.zeros(H)
    accum = 1
    for i in range(len(Gammas)):
        Gammas[i] = accum
        accum *= discount

    for ep, (trajectory, ep_len) in enumerate(zip(ep_trajectories, ep_lengths)):
        Gt = np.zeros(H)
        Rt = trajectory[:, 2]
        for t in range(ep_len):
            # compute sum of discounted rewards starting at trajectory step t
            Gt[t] = np.dot(Rt[t:ep_len], Gammas[:(ep_len-t)])

        seen = defaultdict(lambda: 0)
        for t,(s,a,r,ns) in enumerate(trajectory):
            s,a,ns = int(s), int(a), int(ns)
            if seen[s] == 0:
                N[s,a] += 1
                C[s,a] += Gt[t]
            seen[s] += 1

    for s in range(Q.shape[0]):
        for a in range(Q.shape[1]):
            if N[s,a] > 0:
                Q[s,a] = C[s,a] / N[s,a]

    return ep_lengths, ep_trajectories
```

Listing 8: MC-Control using Every-Visit MC

```
def mc_every_visit(env, num_eps, state_fn, H, Q, N, C, epsilon, discount):
    """
    Derives an epsilon-greedy policy from Q, and then estimates updated Q-values using
    every-visit MC on trajectories sampled from the epsilon-greedy derived policy. Same set-ups as
    first-visit method above.
    """

    ep_lengths = np.zeros(num_eps, dtype='int')
    ep_trajectories = np.zeros((num_eps,H,4))
    for ep in range(num_eps):
        state = env.reset()
        s = state_fn(env, state)

        trajectory = np.zeros((H,4))
        step = 0
        for step in range(H):
            pr = random.uniform(0, 1)

            if pr < epsilon:
                a = env.action_space.sample()
            else:
                a = np.argmax(Q[s, :])  # implicit use of Q as the policy function Pi

            next_state, reward, done, info = env.step(a)
            ns = state_fn(env, next_state)

            trajectory[step] = s, a, reward, ns

            s = ns

            if done == True:
                break

        ep_trajectories[ep] = trajectory
        ep_lengths[ep] = step+1  # convert to 1-indexed

    # Precompute discount factors: 1, gamma, gamma^2, gamma^3, ..., gamma^H
    Gammas = np.zeros(H)
    accum = 1
    for i in range(len(Gammas)):
        Gammas[i] = accum
        accum *= discount

    for ep, (trajectory, ep_len) in enumerate(zip(ep_trajectories, ep_lengths)):
        Gt = np.zeros(H)
        Rt = trajectory[:, 2]
        for t in range(ep_len):
            # compute sum of discounted rewards starting at trajectory step t
            Gt[t] = np.dot(Rt[t:ep_len], Gammas[:(ep_len-t)])

        for t,(s,a,r,ns) in enumerate(trajectory):
            s,a,ns = int(s), int(a), int(ns)
            N[s,a] += 1
            C[s,a] += Gt[t]

    for s in range(Q.shape[0]):
        for a in range(Q.shape[1]):
            if N[s,a] > 0:
                Q[s,a] = C[s,a] / N[s,a]

    return ep_lengths, ep_trajectories
```

A.2.4 Bandit Algorithms

Author: Rach Liu

Listing 9: Simple Bandit Algorithm

```
def simple_mab(env, num_eps, H, Q, N, alpha, epsilon):
    """
    The eps-greedy simple bandit algorithm (p24 Sutton) with a customizable alpha(t,a) and bonus
    function b(t,a). Set alpha(t,a):=1/N(a) to get simple_mab1, or const for simple_mab2.
    A constant alpha = 1e-4 would be:  lambda t,a: 1e-4.
    A smoothed alpha = 1/N(a) = H/(H+N(a)) would be:  lambda t,a: H/(H+N(a)) where
       the H, N(a) are in the closure of the lambda.
    To decay epsilon, specify a tuple (max_eps, min_eps, decay-rate) for it.
    Note that because the MAB environment is always done after one iteration, the number of
    episodes is thus equivalent to the number of desired iterations.
    """
    ep_rewards = np.zeros((num_eps, H))
    ep_opt_arm = np.zeros((num_eps, H))
    ep_epsilons = np.zeros(num_eps)

    decay = None

    if type(epsilon) is tuple:
        # decaying epsilon
        max_epsilon = epsilon[0]
        min_epsilon = epsilon[1]
        decay = epsilon[2]
        epsilon = max_epsilon

    for ep in range(num_eps):
        env.reset()
        rewards = np.zeros(H)
        opt_arm = np.zeros(H)

        for step in range(0, H):
            pr = random.uniform(0, 1)
            if pr > epsilon:
                a = np.argmax(Q[:])
            else:
                a = env.action_space.sample()

            _, reward, done, info = env.step(a)

            N[a] += 1
            Q[a] = Q[a] + alpha(step, a) * (reward - Q[a])

            rewards[step] = reward
            if 'optimal_arm' in info:
                opt_arm[step] = a == info['optimal_arm']

            if done:
                # print ("Total reward for episode {}: {}".format(episode, rewards))
                break

        if decay is not None:
            # decay the epsilon
            epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-decay * ep)

        ep_rewards[ep] = rewards
        ep_opt_arm[ep] = opt_arm
        ep_epsilons[ep] = epsilon

    return ep_rewards, ep_opt_arm, ep_epsilons

"""
Notes on 1/N(a) vs constant alpha learning rate:
  1/N(a) is a decay on the learning rate as N(a) increases; which satisfies the
  R-M conditions and ensure convergence, though cannot solve *non*-stationary problems
  (rewards(a) that change over time).  for tracking non-stationary problems,
  convergence is actually sacrificed for a constant alpha, which intuitively ensures
  sufficient exploration while rewards(a) change.
  H/(H+N(a)) needed for smoothing, otherwise the 1/0 initial values skew Q
    towards some extremes.
"""
```

Listing 10: Simple Bandit Algorithm with UCB

```
def simple_mab3(env, num_eps, H, Q, N, alpha, bonus):
    """
    The simple bandit algorithm (p24 Sutton) with a customizable alpha(t,a) and bonus function
    b(t,a). Set alpha(t,a):=1/N(a) to get simple_mab1, or const for simple_mab2. Set a bonus
    function for ex to sqrt(logt^2/2N(a)) for UCB-1.
    Note that because the MAB environment is always done after one iteration, the number of
    episodes is thus equivalent to the number of desired iterations.
    """
    ep_rewards = np.zeros((num_eps, H))
    ep_opt_arm = np.zeros((num_eps, H))

    for ep in range(num_eps):
        env.reset()
        rewards = np.zeros(H)
        opt_arm = np.zeros(H)

        for step in range(0, H):

            q_max = None
            a_max = None
            for a in range(0, env.action_space.n):
                if (N[a] == 0):
                    a_max = a
                    break
                else:
                    q = Q[a] + bonus(step, a)
                    if q_max is None or q_max < q:
                        q_max = q
                        a_max = a

            _, reward, done, info = env.step(a_max)

            N[a_max] += 1
            Q[a_max] = Q[a_max] + alpha(step, a_max) * (reward - Q[a_max]) + bonus(step,a)

            rewards[step] = reward
            if 'optimal_arm' in info:
                opt_arm[step] = a_max == info['optimal_arm']

            if done:
                # print ("Total reward for episode {}: {}".format(episode, rewards))
                break

        ep_rewards[ep] = rewards
        ep_opt_arm[ep] = opt_arm

    return ep_rewards, ep_opt_arm
```

Listing 11: Gradient Bandit Algorithm

```
def gradient_mab(env, num_eps, H, Q, N, alpha):
    """
    Gradient-update for the simple-mab 1 (epsilon-greedy action selection, simple-bandit algorithm).
    Note: 'T' is the maximum trajectory length instead of H (as for the other bandit functions),
      since H now represents the H(a) function.
    """
    ep_rewards = np.zeros(num_eps)
    ep_opt_arm = np.zeros(num_eps)
    ep_rbar = np.zeros(num_eps)

    Rbar = 0

    for ep in range(num_eps):
        env.reset()

        reward = None
        opt_arm = None

        for step in range(0, 1):
            pr = random.uniform(0, 1)
#            H = H - np.max(H)  # log overflow prevention trick
            Pi = np.exp(H) / np.sum(np.exp(H))
            Pi_cdf = np.cumsum(Pi)

            a = Pi_cdf.shape[0]-1
            for i in reversed(range(Pi_cdf.shape[0])):
                if pr < Pi_cdf[i]:
                    a = i

            _, reward, done, info = env.step(a)

            N[a] += 1
            Q[a] = Q[a] + (1/N[a]) * (reward - Q[a])
#            Q[a] = Q[a] + (num_eps/(N[a]+num_eps)) * (reward - Q[a])

            Rbar += reward

            for a_ in range(env.action_space.n):
                if a == a_:
                    H[a] = H[a] + alpha * (1 - Pi[a])
                else:
#                    H[a_] = H[a_] - alpha * (reward - Q[a_]) * Pi[a_]
                    H[a_] = H[a_] - alpha * (reward - Rbar/(ep+1)) * Pi[a_]

            if 'optimal_arm' in info:
                opt_arm = a == info['optimal_arm']

            if done:
                # print ("Total reward for episode {}: {}".format(episode, rewards))
                break

        ep_rewards[ep] = reward
        ep_opt_arm[ep] = opt_arm
        ep_rbar[ep] = Rbar

    return ep_rewards, ep_opt_arm, ep_rbar
```

A.2.5 Supporting Code

Author: Rach Liu

Listing 12: Features

```
import numpy as np
import math

def create_feat_fn(env, feat_fn):
    """
    Wrapper to create a feat_fn with closure to env and given feat_fn
    """

    def _feat_fn(state, action=None):
        """
        Returns a list of mapped features
        """
        if action is None:
            actions = list(range(env.unwrapped.action_space.n))
        elif type(action) is list:
            actions = action
        else:
            actions = [action]

        features = []
        for a in actions:
            features.append(feat_fn(state, a))

        return np.array(features)
    return _feat_fn


def create_feat_fn_identity(env, size_s, size_a):
    """
    Returns the input, for testing
    """
    def _feat_fn_identity(state, action):
        fvec = np.zeros((size_s,size_a))
        fvec[state, action] = 1
        return fvec.reshape(-1)

    return create_feat_fn(env, _feat_fn_identity)


def create_feat_fn_2dtile(env, tile_size=(2,2)):
    W,H = env.unwrapped.width, env.unwrapped.height
    w,h = tile_size

    def _feat_fn_2dtile(state, action):
        x, y = state
#         offsets = [(0,0), (1,0)]
        offsets = [(0,0)]
        W2, H2 = math.ceil(W/w), math.ceil(H/h)
        fvec = np.zeros(W2*H2*len(offsets)+1)

        # for each tiling (each different tiling is represented as an offset from 'base'-tiling)
        #   examine which tile the given state (x,y) falls into
        for i,(ox,oy) in enumerate(offsets):
            col = (x-ox) // w
            row = (y-oy) // h
            fvec[(i*W2*H2) + row*W2+col] = 1
            fvec[-1] = action
        return fvec

    return create_feat_fn(env, _feat_fn_2dtile)


def test_feat():
    print("Test feat ..")

    class Object(object):
        pass
    env = Object()
    env.unwrapped = Object()
    env.unwrapped.width = 4
    env.unwrapped.height = 4
    env.unwrapped.action_space = Object()
    env.unwrapped.action_space.n = 5
    fn = create_feat_fn_2dtile(env)

    # test single action '3'
    a = fn((2,1), 3)

    # test fake slice ':' (for all actions..)
    b = fn((2,1))

    # test list of actions [3,2]
    c = fn((2,1), [3,2])

    print(a, "\n", b, "\n", c, "\n")

    pass


if __name__ == "__main__":
    test_feat()
```

Listing 13: Q-Table Inializations and Minigrid Env Utils

```
from gym_minigrid.minigrid import *


def init_q(env):
    """
    Returns a new q(s,a)-table, N(s,a)-table, E(s,a)-table.
    """
    benv = env.unwrapped
    Q = np.zeros((benv.width * benv.height, benv.action_space.n))
    N = np.zeros((benv.width * benv.height, benv.action_space.n))
    E = np.zeros((benv.width * benv.height, benv.action_space.n))
    return Q, N, E

def init_q_optimistic(env, H):
    """
    Returns a new q(s,a)-table, N(s,a)-table, intialized to H (optimistic).
    """
    benv = env.unwrapped
    Q = np.full((benv.width * benv.height, benv.action_space.n), H)
    N = np.zeros((benv.width * benv.height, benv.action_space.n))
    return Q, N

def init_mccontrol(env):
    """
    Returns a policy function table, pi(a|s), that returns the next action to take given a (
    state, action) pair, q(s,a) table, N(s,a) table, and C(s,a) table (stores G_t's of every
    (s,a)).
    """
    benv = env.unwrapped
    Pi = np.zeros((benv.width * benv.height, benv.action_space.n))
    Q = np.zeros((benv.width * benv.height, benv.action_space.n))
    N = np.zeros((benv.width * benv.height, benv.action_space.n))
    C = np.zeros((benv.width * benv.height, benv.action_space.n))
    return Pi, Q, N, C

def state_fn_identity(env, state):
    """
    Identity (pass-through); returns the same state as input.
    """
    return state

def state_fn_agentpos(env, state):
    """
    Converts the agent position for use as a state.
    """
    benv = env.unwrapped
    i,j = benv.agent_pos
    # grid coords are (x,y) ie (col, row)
    return benv.width * j + i

def state_fn_2dtile0(env, state):
    """
    2d state-aggregation (tile-code of 0 overlap) on the physical gridworld grid.
    For use in tabular setting.
    """
    benv = env.unwrapped
    W,H = benv.width, benv.height
    w,h = (2, 2)
    x, y = benv.agent_pos
    col = x // w
    row = y // h
    return row * (math.ceil(W//w)) + col
```

Listing 14: MAB Initialization

```
def init_mab(env):
    """
    :return: Returns a new Q-table, N(a)-table for a MAB environment.
    """
    Q = np.zeros(env.action_space.n)
    N = np.zeros(env.action_space.n)
    return Q, N

def init_gradmab(env):
    """
    :return: Returns a new H(a)-table, Q(a)-table, N(a)-table for a gradient
      algorithms in MAB setting.
    """
    H = np.zeros(env.action_space.n)
    Q = np.zeros(env.action_space.n)
    N = np.zeros(env.action_space.n)
    return H, Q, N
```

A.2.6 Customization Codes for Creating Classic Gridworld from Minigridworld [4]

Author: Rach Liu, based on sample/original code by [4]

Listing 15: envs/simple.py

```
"""
Add custom entities here
"""
OBJECT_TO_IDX['wall2'] = 100
"""
Regenerate IDX_TO_OBJECT map
"""
IDX_TO_OBJECT = dict(zip(OBJECT_TO_IDX.values(), OBJECT_TO_IDX.keys()))

class Wall2(WorldObj):
    """
    When SimpleMiniGridEnv.step sees this wall type, it will exclude it from the returned
    list of legal actions. Thus, this wall type can be used for walls in which the agent can
    exclude from moving towards.
    """
    def __init__(self, color='grey'):
        super().__init__('wall2', color)
    def see_behind(self):
        return False

    def render(self, img):
        fill_coords(img, point_in_rect(0, 1, 0, 1), COLORS[self.color])


class SimpleMiniGridEnv(MiniGridEnv):

    class Actions(IntEnum):
        # move left, move right, move forward, move down
        left = 0
        right = 1
        up = 2
        down = 3

    """
      # Map agent's direction to short string
      AGENT_DIR_TO_STR = {
          0: '>',
          1: 'V',
          2: '<',
          3: '^'
      }
    """

    def __init__(
            self,
            grid_size=None,
            width=None,
            height=None,
            max_steps=100,
            see_through_walls=False,
            seed=1337,
            agent_view_size=7
    ):
        super().__init__(
            grid_size,
            width,
            height,
            max_steps,
            see_through_walls,
            seed,
            agent_view_size
        )

        # Action enumeration for this environment
        self.actions = SimpleMiniGridEnv.Actions

        # Actions are discrete integer values
        self.action_space = spaces.Discrete(len(self.actions))

        # Initialize previous cell
        self.prev_pos = None

        self.agent_pos = np.array(self.agent_pos)


    def _reward(self):
        """
        Compute the reward to be given upon success
        Override for map-specific reward.
        """
        #return 1 - 0.9 * (self.step_count / self.max_steps)
        return 10

    def _livingreward(self):
        """
        The standard cliff grid setup has a -1 reward for all transitions.
        Override for map-specific reward.
        """
        return -1

    def _lavareward(self):
        """
        Override for map-specific reward.
        """
        # a -100 reward allows the cumulative rewards to better express the trajectory
        return -100


    def _legalactions(self):
        """
        Generates a list of legal actions. Returned as the 'info' (note: is an ndarray,
        not a python dict as usual for 'info').
        *** For the case where the agent is able to rotate.
        """
        # for if using agent_direction
        #  # Direction of agent's left:
        #  agent_left_dir = self.agent_dir - 1
        #  if agent_left_dir < 0:
        #      agent_left_dir += 4
        #   # Direction of agent's right:
        #   agent_right_dir = (self.agent_dir + 1) % 4
        #
        #   agent_left_pos = self.agent_pos + DIR_TO_VEC[agent_left_dir]
        #   agent_right_pos = self.agent_pos + DIR_TO_VEC[agent_right_dir]
        #
        #   agent_left_cell = self.grid.get(*agent_left_pos)
        #   agent_right_cell = self.grid.get(*agent_right_pos)
        #   agent_fwd_cell = self.grid.get(*self.front_pos)

        left_pos = self.agent_pos - [1, 0]
        right_pos = self.agent_pos + [1, 0]
        up_pos = self.agent_pos - [0, 1]
        down_pos = self.agent_pos + [0, 1]

        agent_left_cell = self.grid.get(*left_pos)
        agent_right_cell = self.grid.get(*right_pos)
        agent_up_cell = self.grid.get(*up_pos)
        agent_down_cell = self.grid.get(*down_pos)

        legal_actions = []
        if agent_left_cell is None or agent_left_cell.type is not 'wall2':
            legal_actions.append(0)
        if agent_right_cell is None or agent_right_cell.type is not 'wall2':
            legal_actions.append(1)
        if agent_up_cell is None or agent_up_cell.type is not 'wall2':
            legal_actions.append(2)
        if agent_down_cell is None or agent_down_cell.type is not 'wall2':
            legal_actions.append(3)

        return np.array(legal_actions)



    def step(self, action):
        self.step_count += 1

        reward = self._livingreward()  # 0 by default
        done = False

        if action == self.actions.left:
            next_pos = self.agent_pos - [1, 0]
        elif action == self.actions.right:
            next_pos = self.agent_pos + [1, 0]
        elif action == self.actions.down:
            next_pos = self.agent_pos + [0, 1]
        elif action == self.actions.up:
            next_pos = self.agent_pos - [0, 1]
        else:
            raise Exception("SimpleGridWorld: Invalid action")

        next_cell = self.grid.get(*next_pos)

        # Move to next cell:
        if self.prev_pos is None or not(np.array_equal(next_pos, self.prev_pos)):
            if next_cell == None or next_cell.can_overlap():
                self.prev_pos = self.agent_pos
                self.agent_pos = next_pos
            if next_cell != None and next_cell.type == 'goal':
                reward = self._reward()
                done = True
            if next_cell != None and next_cell.type == 'lava':
                reward = self._lavareward()
                done = True

        if self.step_count >= self.max_steps:
            done = True

        obs = self.gen_obs()

        legal_actions = self._legalactions()

        return obs, reward, done, legal_actions

    def __str__(self):
        """
        Produce a pretty string of the environment's grid along with the agent.
        A grid cell is represented by 2-character string, the first one for
        the object and the second one for the color.
        """
        # Map of object types to short string
        OBJECT_TO_STR = {
            'wall'          : 'W',
            'floor'         : 'F',
            'door'          : 'D',
            'key'           : 'K',
            'ball'          : 'A',
            'box'           : 'B',
            'goal'          : 'G',
            'lava'          : 'V',
            #
            'wall2': '-',
        }

        str = ''

        for j in range(self.grid.height):

            for i in range(self.grid.width):
                if i == self.agent_pos[0] and j == self.agent_pos[1]:
                    str += 'o'
                    continue
                c = self.grid.get(i, j)

                if c == None:
                    str += ' '
                    #str += '  '
                    continue

                if c.type == 'door':
                    if c.is_open:
                        str += '_'
                        #str += '__'
                    elif c.is_locked:
                        str += 'L'# + c.color[0].upper()
                    else:
                        str += 'D'# + c.color[0].upper()
                    continue

                str += OBJECT_TO_STR[c.type]# + c.color[0].upper()

            if j < self.grid.height - 1:
                str += '\n'

        return str



class SimpleEmptyEnv(SimpleMiniGridEnv):
    """
    Empty grid environment, no obstacles, sparse reward
    """

    def __init__(
            self,
            size=8,
            agent_start_pos=(1, 1),
            agent_start_dir=0,
    ):
        self.agent_start_pos = agent_start_pos
        self.agent_start_dir = agent_start_dir

        super().__init__(
            grid_size=size,
            max_steps=4 * size * size,
            # Set this to True for maximum speed
            see_through_walls=True
        )

    def _gen_grid(self, width, height):
        # Create an empty grid
        self.grid = Grid(width, height)

        # Generate the surrounding walls
        self.grid.wall_rect(0, 0, width, height)

        # Place a goal square in the bottom-right corner
        self.put_obj(Goal(), width - 2, height - 2)

        # Place the agent
        if self.agent_start_pos is not None:
            self.agent_pos = np.array(self.agent_start_pos)
            self.agent_dir = self.agent_start_dir
        else:
            self.place_agent()

        self.mission = "get to the green goal square"



class SimpleEmptyEnv3x3(SimpleEmptyEnv):
    def __init__(self, **kwargs):
        super().__init__(size=5, **kwargs)

class SimpleEmptyEnv4x4(SimpleEmptyEnv):
    def __init__(self, **kwargs):
        super().__init__(size=6, **kwargs)

class SimpleEmptyEnv5x5(SimpleEmptyEnv):
    def __init__(self, **kwargs):
        super().__init__(size=7, **kwargs)

class SimpleEmptyEnv6x6(SimpleEmptyEnv):
    def __init__(self, **kwargs):
        super().__init__(size=8, **kwargs)

class SimpleEmptyEnv7x7(SimpleEmptyEnv):
    def __init__(self, **kwargs):
        super().__init__(size=9, **kwargs)

class SimpleEmptyEnv8x8(SimpleEmptyEnv):
    def __init__(self, **kwargs):
        super().__init__(size=10, **kwargs)

register(
    id='MiniGrid-Simple-Empty-3x3-v0',
    entry_point='gym_minigrid.envs:SimpleEmptyEnv3x3'
)

register(
    id='MiniGrid-Simple-Empty-4x4-v0',
    entry_point='gym_minigrid.envs:SimpleEmptyEnv4x4'
)

register(
    id='MiniGrid-Simple-Empty-5x5-v0',
    entry_point='gym_minigrid.envs:SimpleEmptyEnv5x5'
)

register(
    id='MiniGrid-Simple-Empty-6x6-v0',
    entry_point='gym_minigrid.envs:SimpleEmptyEnv6x6'
)

register(
    id='MiniGrid-Simple-Empty-7x7-v0',
    entry_point='gym_minigrid.envs:SimpleEmptyEnv7x7'
)

register(
    id='MiniGrid-Simple-Empty-8x8-v0',
    entry_point='gym_minigrid.envs:SimpleEmptyEnv8x8'
)



class SimpleRandomWalkEnv(SimpleMiniGridEnv):
    """
    Random walk environment, no obstacles
    """

    def __init__(
            self,
            size=10,
            agent_start_pos=None,
            agent_start_dir=0,
    ):
        self.agent_start_pos = agent_start_pos
        self.agent_start_dir = agent_start_dir

        super().__init__(
            width=size,
            height=3,
            max_steps=4 * size * size,
            # Set this to True for maximum speed
            see_through_walls=True
        )

    def _reward(self):
        if self.agent_pos[0] > self.width//2:
            return 1
        else:
            return 1

    def _livingreward(self):
        return 0

    def _gen_grid(self, width, height):
        # Create an empty grid
        self.grid = Grid(width, height)

        # Generate the surrounding walls
        self.grid.horz_wall(0, 0, obj_type=Wall2)
        self.grid.horz_wall(0, height - 1, obj_type=Wall2)

        # Place the left and right goals
        self.put_obj(Goal(), 0, 1)
        self.put_obj(Goal(), width-1, 1)

        # Place agent in randomly based on a normal:
        s = np.random.normal(width/2, 0.05*width, 1)
        self.agent_start_pos = (int(s), 1)
#        self.agent_start_pos = (1, 1)

        # Place the agent
        if self.agent_start_pos is not None:
            self.agent_pos = np.array(self.agent_start_pos)
            self.agent_dir = self.agent_start_dir
        else:
            self.place_agent()

        self.mission = "get to the green goal square"

class SimpleRandomWalkEnv10(SimpleRandomWalkEnv):
    def __init__(self, **kwargs):
        super().__init__(size=10, **kwargs)

class SimpleRandomWalkEnv20(SimpleRandomWalkEnv):
    def __init__(self, **kwargs):
        super().__init__(size=20, **kwargs)

class SimpleRandomWalkEnv30(SimpleRandomWalkEnv):
    def __init__(self, **kwargs):
        super().__init__(size=30, **kwargs)

class SimpleRandomWalkEnv100(SimpleRandomWalkEnv):
    def __init__(self, **kwargs):
        super().__init__(size=100, **kwargs)


register(
    id='MiniGrid-Simple-RandomWalk-10-v0',
    entry_point='gym_minigrid.envs:SimpleRandomWalkEnv10'
)

register(
    id='MiniGrid-Simple-RandomWalk-20-v0',
    entry_point='gym_minigrid.envs:SimpleRandomWalkEnv20'
)

register(
    id='MiniGrid-Simple-RandomWalk-30-v0',
    entry_point='gym_minigrid.envs:SimpleRandomWalkEnv30'
)

register(
    id='MiniGrid-Simple-RandomWalk-100-v0',
    entry_point='gym_minigrid.envs:SimpleRandomWalkEnv100'
)



class SimpleRandomWalkGridEnv(SimpleMiniGridEnv):
    """
    Random walk environment, no obstacles
    """

    def __init__(
            self,
            size=10,
            agent_start_pos=None,
            agent_start_dir=0,
    ):
        self.agent_start_pos = agent_start_pos
        self.agent_start_dir = agent_start_dir

        super().__init__(
            grid_size=size,
            max_steps=4 * size * size,
            # Set this to True for maximum speed
            see_through_walls=True
        )

    def _reward(self):
        return 1

    def _livingreward(self):
        return 0

    def _gen_grid(self, width, height):
        # Create an empty grid
        self.grid = Grid(width, height)

        # Generate the surrounding walls
        self.grid.wall_rect(0, 0, width, height)

        # Place a goal alongst inside walls
        for i in range(1,width-1):
            for j in range(1,height-1):
                if i == 1 or i == width-2 or j==1 or j == height-2:
                    self.put_obj(Goal(), i, j)

        # Place agent in middle
        self.agent_start_pos = (width//2, height//2)

        # Place the agent
        if self.agent_start_pos is not None:
            self.agent_pos = np.array(self.agent_start_pos)
            self.agent_dir = self.agent_start_dir
        else:
            self.place_agent()

        self.mission = "get to the green goal square"

class SimpleRandomWalkGridEnv3x3(SimpleRandomWalkGridEnv):
    def __init__(self, **kwargs):
        super().__init__(size=5, **kwargs)

class SimpleRandomWalkGridEnv6x6(SimpleRandomWalkGridEnv):
    def __init__(self, **kwargs):
        super().__init__(size=8, **kwargs)

class SimpleRandomWalkGridEnv10x10(SimpleRandomWalkGridEnv):
    def __init__(self, **kwargs):
        super().__init__(size=12, **kwargs)

register(
    id='MiniGrid-Simple-RandomWalkGrid-3x3-v0',
    entry_point='gym_minigrid.envs:SimpleRandomWalkGridEnv3x3'
)

register(
    id='MiniGrid-Simple-RandomWalkGrid-6x6-v0',
    entry_point='gym_minigrid.envs:SimpleRandomWalkGridEnv6x6'
)

register(
    id='MiniGrid-Simple-RandomWalkGrid-10x10-v0',
    entry_point='gym_minigrid.envs:SimpleRandomWalkGridEnv10x10'
)


class SimpleCliffEnv(SimpleMiniGridEnv):
    """
    Room with a straight cliff (lava) on one side; make the room height 1
    to make a 1-sided bridge.
    """

    def __init__(self, width, height, obstacle_type=Lava, seed=None):
        self.obstacle_type = obstacle_type
        super().__init__(
            grid_size=None,
            width=width,
            height=height,
            max_steps=20*width*height,
            # Set this to True for maximum speed
            see_through_walls=False,
            seed=None
        )


    def _gen_grid(self, width, height):
        assert width >= 3 and height >= 3

        # Create an empty grid
        self.grid = Grid(width, height)

        # Generate the surrounding walls
        self.grid.wall_rect(0, 0, width, height, obj_type=Wall)
#         self.grid.wall_rect(0, 0, width, height, obj_type=Wall2)

        # Overwrite the bottom wall with lava
        self.grid.horz_wall(0, height-1, width, obj_type=Lava)
#         self.grid.horz_wall(0, height-1, width, obj_type=Wall)
#         self.grid.horz_wall(0, height-1, width, obj_type=Wall2)

        # Place the agent in the bot-left corner
        self.agent_pos = np.array((1, height-2))
        self.agent_dir = 0

        # Place a goal square in the bottom-right corner above the lava
        self.goal_pos = np.array((width - 2, height - 2))
        self.put_obj(Goal(), *self.goal_pos)

        self.mission = (
            "avoid the lava and get to the green goal square"
        )

class SimpleCliffH1W3Env(SimpleCliffEnv):
    def __init__(self):
        super().__init__(width=5, height=3)  # +2 to account for walls on left/right/top/bot

class SimpleCliffH1W5Env(SimpleCliffEnv):
    def __init__(self):
        super().__init__(width=7, height=3)  # +2 to account for walls on left/right/top/bot

class SimpleCliffH1W7Env(SimpleCliffEnv):
    def __init__(self):
        super().__init__(width=9, height=3)  # +2 to account for walls on left/right/top/bot
```

```
class SimpleCliffH2W4Env(SimpleCliffEnv):
    def __init__(self):
        super().__init__(width=6, height=4)  # +2 to account for walls on left/right/top/bot

class SimpleCliffH2W5Env(SimpleCliffEnv):
    def __init__(self):
        super().__init__(width=7, height=4)  # +2 to account for walls on left/right/top/bot

class SimpleCliffH3W3Env(SimpleCliffEnv):
    def __init__(self):
        super().__init__(width=5, height=5)  # +2 to account for walls on left/right/top/bot

class SimpleCliffH3W4Env(SimpleCliffEnv):
    def __init__(self):
        super().__init__(width=6, height=5)  # +2 to account for walls on left/right/top/bot

class SimpleCliffH3W5Env(SimpleCliffEnv):
    def __init__(self):
        super().__init__(width=7, height=5)  # +2 to account for walls on left/right/top/bot

class SimpleCliffH3W7Env(SimpleCliffEnv):
    def __init__(self):
        super().__init__(width=9, height=5)  # +2 to account for walls on left/right/top/bot

class SimpleCliffH3W9Env(SimpleCliffEnv):
    def __init__(self):
        super().__init__(width=11, height=5)  # +2 to account for walls on left/right/top/bot

class SimpleCliffH5W9Env(SimpleCliffEnv):
    def __init__(self):
        super().__init__(width=11, height=7)  # +2 to account for walls on left/right/top/bot

register(
    id='MiniGrid-Simple-Cliff-H1W3-v0',
    entry_point='gym_minigrid.envs:SimpleCliffH1W3Env'
)

register(
    id='MiniGrid-Simple-Cliff-H1W5-v0',
    entry_point='gym_minigrid.envs:SimpleCliffH1W5Env'
)

register(
    id='MiniGrid-Simple-Cliff-H1W7-v0',
    entry_point='gym_minigrid.envs:SimpleCliffH1W7Env'
)

register(
    id='MiniGrid-Simple-Cliff-H2W4-v0',
    entry_point='gym_minigrid.envs:SimpleCliffH2W4Env'
)

register(
    id='MiniGrid-Simple-Cliff-H2W5-v0',
    entry_point='gym_minigrid.envs:SimpleCliffH2W5Env'
)

register(
    id='MiniGrid-Simple-Cliff-H3W3-v0',
    entry_point='gym_minigrid.envs:SimpleCliffH3W3Env'
)

register(
    id='MiniGrid-Simple-Cliff-H3W4-v0',
    entry_point='gym_minigrid.envs:SimpleCliffH3W4Env'
)

register(
    id='MiniGrid-Simple-Cliff-H3W5-v0',
    entry_point='gym_minigrid.envs:SimpleCliffH3W5Env'
)

register(
    id='MiniGrid-Simple-Cliff-H3W7-v0',
    entry_point='gym_minigrid.envs:SimpleCliffH3W7Env'
)

register(
    id='MiniGrid-Simple-Cliff-H3W9-v0',
    entry_point='gym_minigrid.envs:SimpleCliffH3W9Env'
)

register(
    id='MiniGrid-Simple-Cliff-H5W9-v0',
    entry_point='gym_minigrid.envs:SimpleCliffH5W9Env'
)
```

A.3 Jupyter Notebooks

Author: Rach Liu

Listing 16: sarsa and qlearning journal

```
#%%

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

#%%

import gym
import matplotlib.pyplot as plt
import numpy as np
import random
import gym_minigrid
from gym_minigrid.wrappers import *
from utils import *

np.random.seed(0)

#%% md

##### Comparing w and w/o Decay for qlearn, sarsa using Taxi env

#%%

# Taxi
env = gym.make("Taxi-v3").env
env.reset()
env.render()
Utils.show_env_space(env)

from minigrid import *
state_fn = MiniGridUtils.state_fn_identity

#%%

import sarsa
num_eps = 2000
H = 100

alpha = 0.7
discount = 0.618
epsilon = 1
max_epsilon = 1
min_epsilon = 0.01
decay = 0.01

Q = np.zeros((env.observation_space.n, env.action_space.n))

ep_rewards, ep_epsilons = \
    sarsa.sarsa(
        env, num_eps, Q, state_fn,
        discount, alpha, epsilon, min_epsilon, max_epsilon, decay)

print("Average reward per episode: " + str(sum(ep_rewards) / num_eps))
Utils.plot_rewards(ep_rewards, fname='sarsa_taxi_rewards_decay')
Utils.plot_epsilons(ep_epsilons, fname='sarsa_taxi_epsilons_decay')

# without decay
decay = None
Q = np.zeros((env.observation_space.n, env.action_space.n))

ep_rewards, ep_epsilons = \
    sarsa.sarsa(
        env, num_eps, Q, state_fn,
        discount, alpha, epsilon, min_epsilon, max_epsilon, decay)

print("Average reward per episode: " + str(sum(ep_rewards) / num_eps))
Utils.plot_rewards(ep_rewards, fname='sarsa_taxi_rewards_nodecay')
Utils.plot_epsilons(ep_epsilons, fname='sarsa_taxi_epsilons_nodecay')

#%%

import qlearn
num_eps = 2000
H = 100

alpha = 0.7
discount = 0.618
epsilon = 0.5
max_epsilon = 1
min_epsilon = 0.01
decay = 0.01

ep_rewards, ep_epsilons = \
    qlearn.qlearn(
        env, num_eps, Q, state_fn,
        discount, alpha, epsilon, min_epsilon, max_epsilon, decay)

print("Average reward per episode: " + str(sum(ep_rewards) / num_eps))
Utils.plot_rewards(ep_rewards, fname='qlearn_taxi_rewards_decay')
Utils.plot_epsilons(ep_epsilons, fname='qlearn_taxi_epsilons_decay')

# without decay
decay = None
Q = np.zeros((env.observation_space.n, env.action_space.n))

ep_rewards, ep_epsilons = \
    qlearn.qlearn(
        env, num_eps, Q, state_fn,
        discount, alpha, epsilon, min_epsilon, max_epsilon, decay)

print("Average reward per episode: " + str(sum(ep_rewards) / num_eps))
Utils.plot_rewards(ep_rewards, fname='qlearn_taxi_rewards_nodecay')
Utils.plot_epsilons(ep_epsilons, fname='qlearn_taxi_epsilons_nodecay')



#%% md

#### Comparing sarsa and sarsa-lambda using Gridworld 8x8

The empty 8x8 shows the learning, but the cliff grid shows it better, because we can set the lava
 reward to -100.


#%%

import sarsa
from minigrid import *
state_fn = MiniGridUtils.state_fn_agentpos

env = gym.make('MiniGrid-Simple-Empty-8x8-v0')
env = FullyObsWrapper(env)
env.reset()
env.render()
Utils.show_env_space(env)

num_eps = 2000
H = 1000

alpha = lambda t,a: (H+1)/(H+t)
#alpha = lambda t,a: 0.5
#alpha = lambda t,a: 0.8
discount = 0.8
epsilons = [0.2, 0.5, (0.5, 0.1, 0.01)]
lmbda = 0.5


def _new_plot():

    def _plot(rewards, lab):
        plt.plot(range(len(rewards)), rewards, label=lab)
        plt.xlabel('Episode')
        plt.ylabel('Average Cumulative-Reward')
        plt.title('Cumulative-Rewards over Episodes')
        #plt.ylim(ymin=ymin)

    def _end(fname):
        plt.legend(loc="lower right", prop={'size':6})
        plt.savefig(fname, dpi=300, bbox_inches='tight')
        plt.show()

    return _plot, _end

_plot, _end = _new_plot()

# regular sarsa
for epsilon in epsilons:
    Q,N,E = init_q(env)
    ep_lens, ep_rewards, ep_epsilons  = \
        sarsa.sarsa(
            env, num_eps, H, Q, state_fn, alpha, epsilon, discount)

    tot_ep_rewards = np.sum(ep_rewards, axis=1)

    print("Average cumulative-reward per episode: " + str(sum(tot_ep_rewards) / num_eps))
    _plot(tot_ep_rewards, f'sarsa, epsilon={epsilon}')

_end('sarsa_empty8x8')

# sarsa-lambda
for epsilon in epsilons:
    Q,N,E = init_q(env)
    ep_lens, ep_rewards, ep_epsilons  = \
        sarsa.sarsa_lambda(
            env, num_eps, H, Q, E, lmbda, state_fn, alpha, epsilon, discount)
    tot_ep_rewards = np.sum(ep_rewards, axis=1)

    print("Average cumulative-reward per episode: " + str(sum(tot_ep_rewards) / num_eps))
    _plot(tot_ep_rewards, f'sarsa-lambda, epsilon={epsilon}')

_end('sarsalambda_empty8x8')


#%% md

#### Cliffworld investigation

#%%

from minigrid import *
state_fn = MiniGridUtils.state_fn_agentpos
import sarsa

#env = gym.make('MiniGrid-Simple-Cliff-H3W4-v0')
#env = gym.make('MiniGrid-Simple-Cliff-H2W4-v0')
env = gym.make('MiniGrid-Simple-Cliff-H5W9-v0')
env = FullyObsWrapper(env)
env.reset()
env.render()
Utils.show_env_space(env)

#alpha = lambda t,a: 0.5
alpha = lambda t,a: (H+1)/(H+t)
discount = 0.9
epsilons = [0.05, (0.5, 0.1, 0.01)]
epsilon = epsilons[1]
lmbda = 0.5

num_eps = 2000
H = 100

def _new_plot():
    fig, axs = plt.subplots(1,2)
    for ax in axs.flat:
        ax.set(xlabel='Episode', ylabel='Reward')
        ax.label_outer()
    fig.tight_layout()

    def _plot(i, rewards, lab):
        axs.flat[i].plot(range(len(rewards)), rewards, label=lab)

    def _end(fname):
        for ax in axs.flat:
            ax.legend(*ax.get_legend_handles_labels(), loc="lower right", prop={'size':6})
        plt.savefig(fname, dpi=300, bbox_inches='tight')
        plt.show()

    return _plot, _end

# regular sarsa
_plot, _end = _new_plot()

Q, N, E = MiniGridUtils.init_q(env)
ep_lens, ep_rewards, ep_epsilons  = \
    sarsa.sarsa(
       env, num_eps, H, Q, state_fn, alpha, epsilon, discount)

tot_ep_rewards = np.sum(ep_rewards, axis=1)/ep_lens
#avg_ep_rewards = np.sum(ep_rewards, axis=1)/ep_lens

smoothed_tot_ep_rewards = np.convolve(tot_ep_rewards, np.ones(20)/20, mode='valid')
print("Average cumulative-reward per episode: " + str(sum(tot_ep_rewards) / num_eps))
_plot(0, smoothed_tot_ep_rewards, 'sarsa')

# sarsa-lambda
Q, N, E = MiniGridUtils.init_q(env)
ep_lens, ep_rewards, ep_epsilons  = \
    sarsa.sarsa_lambda(
       env, num_eps, H, Q, E, lmbda, state_fn, alpha, epsilon, discount)

tot_ep_rewards = np.sum(ep_rewards, axis=1)

print("Average cumulative-reward per episode: " + str(sum(tot_ep_rewards) / num_eps))
_plot(1, tot_ep_rewards, 'sarsa-lambda')

smoothed_tot_ep_rewards = np.convolve(tot_ep_rewards, np.ones(20)/20, mode='valid')
_end('sarsa-vs-sarsalambda__cliffh5w9')


#%% md

#### Q-Learn vs Q-Learn with UCB-1


#%%
```

```
from minigrid import *
from qlearn import *
from utils import *

env_init = lambda: gym.make('MiniGrid-Simple-Cliff-H5W9-v0')
state_fn = MiniGridUtils.state_fn_agentpos

env = env_init()
num_eps = 2000
H = 100
discount = 0.8


def _new_plot():
    fig, axs = plt.subplots(1,3)
    for ax in axs.flat:
        ax.set(xlabel='Episode', ylabel='Reward')
        ax.label_outer()
    fig.tight_layout()

    def _plot(i, rewards, lab):
        axs.flat[i].plot(range(len(rewards)), rewards, label=lab)

    def _end(fname):
        for ax in axs.flat:
            ax.legend(*ax.get_legend_handles_labels(), loc="lower right", prop={'size':6})
        plt.savefig(fname, dpi=300, bbox_inches='tight')
        plt.show()

    return _plot, _end

_plot, _end = _new_plot()

# treat (H^3)iota as a hyperparameter
H_iota = 1000

env = env_init()
Q, N = MiniGridUtils.init_q_optimistic(env, H)
alpha = lambda t,a: (H+1)/(H+t)
#bonus = lambda t,s,a: 2*np.sqrt(H**3/N[s,a])
bonus = lambda t,s,a: 2*np.sqrt(H_iota/N[s,a])

ep_lens, ep_rewards, ep_opt_axn  = \
    qlearn_ucb1(env, num_eps, H, Q, N, state_fn, alpha, bonus, discount)

tot_ep_rewards = np.sum(ep_rewards, axis=1)
print("Average reward per episode: " + str(sum(tot_ep_rewards) / num_eps))
_plot(0, tot_ep_rewards, 'qlearn-ucb1')


#epsilons = [0.1, 0.5, (0.5, 0.1, 0.01)]
#for i, epsilon in enumerate(epsilons):
epsilons = [(0.5, 0.1, 0.01)]
q_inits = ['optim', 'no-optim']
for i, q_init in enumerate(q_inits):
    print(f"*** qlearn, epsilon {epsilon} ***")

    if q_init == 'optim':
        Q,N = MiniGridUtils.init_q_optimistic(env, H)
    else:
        Q,N,E = MiniGridUtils.init_q(env)
#    alpha = lambda t,a: (H+1)/(H+t)
    alpha = lambda t,a: 0.25
    ep_lens, ep_rewards, ep_opt_axn, ep_epsilons  = \
        qlearn(env, num_eps, H, Q, N, state_fn, alpha, epsilon, discount)

    tot_ep_rewards = np.sum(ep_rewards, axis=1)
    smoothed_tot_ep_rewards = np.convolve(tot_ep_rewards, np.ones(20)/20, mode='valid')
    print("Average cumulative-reward per episode: " + str(sum(tot_ep_rewards) / num_eps))
    _plot(1 + i, smoothed_tot_ep_rewards, f'eps-decay qlearn')

_end('qlearn_cliff-h5w9')


#%% md

####
```

Listing 17: Monte Carlo Methods journal

```
#%%

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

#%%

import gym
import matplotlib.pyplot as plt
import numpy as np
import random
import gym_minigrid
from gym_minigrid.wrappers import *
from utils import *

np.random.seed(0)

#%% md

##### Gradient bandit algorithm

#%%

from gym_bandits import *
from bandits import *

#env_init = lambda: gym.make('BanditTenArmedGaussian_mean4-v0')
env_init = lambda: gym.make('BanditTenArmedGaussian-v0')
num_eps = 200
alpha = 0.1
T = 1000 # number of desired iterations (pulls in 1 sequence)

ep_rewards = np.zeros((num_eps, T))
ep_opt_arms = np.zeros((num_eps, T))
ep_rbar = np.zeros((num_eps, T))

def _new_plot():
    fig, axs = plt.subplots(2)
    for ax in axs.flat:
        ax.set(xlabel='Episode', ylabel='Average Cumulative Rewards')
        ax.label_outer()
    axs[1].set(ylabel='Optimal Arm %')
    fig.tight_layout()

    def _plot(i, data, lab):
        axs.flat[i].plot(range(len(data)), data, label=lab)

    def _end(fname):
        for ax in axs.flat:
            ax.legend(*ax.get_legend_handles_labels(), loc="lower right", prop={'size':6})
        plt.savefig(fname, dpi=300, bbox_inches='tight')
        plt.show()

    return _plot, _end

_plot, _end = _new_plot()


for ep in range(num_eps):
    env = env_init()
    Ha,Q,N = init_gradmab(env)
    iter_rewards, iter_opt_arms, iter_rbar = \
        gradient_mab(env, T, Ha, Q, N, alpha)
    ep_rewards[ep] = iter_rewards
    ep_opt_arms[ep] = iter_opt_arms
    ep_rbar[ep] = iter_rbar

avg_rewards = np.average(ep_rewards, axis=0)
avg_opt_arm = np.average(ep_opt_arms, axis=0)

print("Average reward per episode: " + str(sum(avg_rewards) / num_eps))
_plot(0, avg_rewards, 'reward')
_plot(1, avg_opt_arm, 'opt_arm')


_end('gradmab_10arm-gaussian-mean0')



#%%

Pi = np.exp(Ha)/np.sum(np.exp(Ha))
Pi_cdf = np.cumsum(np.exp(Ha)/np.sum(np.exp(Ha)))

rbar = iter_rbar/np.arange(1,T+1)






#%% md

##### 1st-vist, Every-visit MC


#%%

import mc
from minigrid import *
```

```
env = gym.make('MiniGrid-Simple-Empty-5x5-v0')
env = FullyObsWrapper(env)
env.reset()
env.render()
Utils.show_env_space(env)
state_fn = state_fn_agentpos

num_eps = 200
H = 100
pi_iters = 30

discount = 0.8
epsilon = 0.1

tot_ep_rewards_1st = None
tot_ep_rewards_every = None

_,Q_1st,N,C = init_mccontrol(env)
for i in range(pi_iters):
    ep_lengths, ep_trajectories = \
        mc.mc_first_visit(env, num_eps, state_fn, H, Q_1st, N, C, epsilon, discount)
    ep_rewards = ep_trajectories[:,:,2]
    if tot_ep_rewards_1st is None:
        tot_ep_rewards_1st = np.sum(ep_rewards, axis=1)
    else:
        tot_ep_rewards_1st = np.vstack((tot_ep_rewards_1st, np.sum(ep_rewards, axis=1)))

_,Q_every,N,C = init_mccontrol(env)
for i in range(pi_iters):
    ep_lengths, ep_trajectories = \
        mc.mc_every_visit(env, num_eps, state_fn, H, Q_every, N, C, epsilon, discount)
    ep_rewards = ep_trajectories[:,:,2]
    if tot_ep_rewards_every is None:
        tot_ep_rewards_every = np.sum(ep_rewards, axis=1)
    else:
        tot_ep_rewards_every = np.vstack((tot_ep_rewards_every, np.sum(ep_rewards, axis=1)))

#%%

# Generate a Q using a Q-Learner to compare the Q's
#   to be fair, average a few of them

import qlearn

tot_ep_rewards_reg = None

for i in range(5):
    Q_reg,N,E = init_q(env)
    #    alpha = lambda t,a: (H+1)/(H+t)
    alpha = lambda t,a: 0.25
    ep_lens, ep_rewards, ep_opt_axn, ep_epsilons = \
        qlearn.qlearn(env, num_eps, H, Q_reg, N, state_fn, alpha, epsilon, discount)
    if tot_ep_rewards_reg is None:
        tot_ep_rewards_reg = np.sum(ep_rewards, axis=1)
    else:
        tot_ep_rewards_reg = np.vstack((tot_ep_rewards_reg, np.sum(ep_rewards, axis=1)))


#%%

def _new_plot():
    fig, axs = plt.subplots(1,3)
    for ax in axs.flat:
        ax.set(xlabel='Episode', ylabel='Average Cumulative Rewards')
        ax.label_outer()
    axs[0].set(xlabel='Policy Iteration #')
    axs[1].set(xlabel='Policy Iteration #')
    fig.tight_layout()

    def _plot(i, rewards, lab):
        axs.flat[i].plot(range(len(rewards)), rewards, label=lab)

    def _end(fname):
        for ax in axs.flat:
            ax.legend(*ax.get_legend_handles_labels(), loc="lower right", prop={'size':6})
        plt.savefig(fname, dpi=300, bbox_inches='tight')
        plt.show()

    return _plot, _end

_plot, _end = _new_plot()

_plot(0, np.sum(tot_ep_rewards_1st, axis=1)/num_eps, "1st-visit MC")
_plot(1, np.sum(tot_ep_rewards_every, axis=1)/num_eps, "1st-visit MC")
_plot(2, tot_ep_rewards_reg[3], "Regular Q-Learning")

_end("mcpi-q-vs-qlearn-q_rewards.png")
```

Listing 18: bandits journal

```
#%%

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

#%%

import gym
from utils import *
from gym_bandits import *
np.random.seed(0)

env = gym.make('BanditTenArmedGaussian-v0')

#%% md

#### epsilon-greedy simple-bandit

#%%

from bandits import *

env_init = lambda: gym.make('BanditTenArmedGaussian-v0')

num_eps = 2000
H = 1000

fig, (ax1, ax2, ax3) = plt.subplots(1,3)
ax1.set_title("Rewards")
ax1.set_xlabel("Iteration")
ax1.set_ylabel("Reward")
ax2.set_title("Optimal Arm %")
ax2.set_xlabel("Iteration")
ax2.set_ylabel("Optimal Arm %")
ax3.set_title("Epsilon")
ax3.set_xlabel("Iteration")
ax3.set_ylabel("Epsilon")
fig.tight_layout()

def _plot(rewards, opt_arm, eps, lab):
    ax1.plot(range(len(rewards)), rewards, label=lab)
    ax2.plot(range(len(opt_arm)), opt_arm, label=lab)
    ax3.plot(range(len(eps)), eps, label=lab)

alphas = [
    {'name': '1/N(a)', 'f': lambda t,a: H/(H+N[a])},
    {'name': 'const 1e-4', 'f': lambda t,a: 1e-4},
]

epsilons = [0.1, 0.5, (0.5, 0.1, 0.01)]


#####
# alpha = 1/N
alpha = alphas[0]

for epsilon in epsilons:
    print(f"*** epsilon {epsilon} ***")

    ep_rewards = np.zeros((num_eps, H))
    ep_opt_arms = np.zeros((num_eps, H))
    for ep in range(num_eps):
        env = env_init()
        Q,N = init_mab(env)
        iter_rewards, iter_opt_arms, iter_epsilons = \
            simple_mab(env, H, 1, Q, N, alpha['f'], epsilon)
        ep_rewards[ep] = iter_rewards[:,0]
        ep_opt_arms[ep] = iter_opt_arms[:,0]

    avg_rewards = np.average(ep_rewards, axis=0)
    avg_opt_arm = np.average(ep_opt_arms, axis=0)

    print("Average reward per episode: " + str(sum(avg_rewards) / num_eps))
    _plot(avg_rewards, avg_opt_arm, iter_epsilons, epsilon)

ax1.legend(*ax1.get_legend_handles_labels(), loc="lower right", prop={'size':6})
ax2.legend(*ax2.get_legend_handles_labels(), loc="lower right", prop={'size':6})
ax3.legend(*ax3.get_legend_handles_labels(), loc="center right", prop={'size':6})
plt.savefig('mab_10arm-gaussian_alpha-1_n', dpi=300, bbox_inches='tight')
plt.show()


#%% md

##### alpha = constant

#%%

alpha = alphas[1]
```

```
fig, (ax1, ax2, ax3) = plt.subplots(1,3)
ax1.set_title("Rewards")
ax1.set_xlabel("Iteration")
ax1.set_ylabel("Reward")
ax2.set_title("Optimal Arm %")
ax2.set_xlabel("Iteration")
ax2.set_ylabel("Optimal Arm %")
ax3.set_title("Epsilon")
ax3.set_xlabel("Iteration")
ax3.set_ylabel("Epsilon")
fig.tight_layout()

for epsilon in epsilons:
    print(f"*** epsilon {epsilon} ***")

    ep_rewards = np.zeros((num_eps, H))
    ep_opt_arms = np.zeros((num_eps, H))
    for ep in range(num_eps):
        env = env_init()
        Q,N = init_mab(env)
        iter_rewards, iter_opt_arms, iter_epsilons = \
            simple_mab(env, H, 1, Q, N, alpha['f'], epsilon)
        ep_rewards[ep] = iter_rewards[:,0]
        ep_opt_arms[ep] = iter_opt_arms[:,0]

    avg_rewards = np.average(ep_rewards, axis=0)
    avg_opt_arm = np.average(ep_opt_arms, axis=0)

    print("Average reward per episode: " + str(sum(avg_rewards) / num_eps))
    _plot(avg_rewards, avg_opt_arm, iter_epsilons, epsilon)

ax1.legend(*ax1.get_legend_handles_labels(), loc="lower right", prop={'size':6})
ax2.legend(*ax2.get_legend_handles_labels(), loc="lower right", prop={'size':6})
ax3.legend(*ax3.get_legend_handles_labels(), loc="center right", prop={'size':6})
plt.savefig('mab_10arm-gaussian_alpha-constant', dpi=300, bbox_inches='tight')
plt.show()


#%% md

#### UCB-1 MAB

#%%

# these are possible options for alpha
alpha = lambda t,a: H/(H+N[a])
alpha = lambda t,a: 1e-4

# these are equivalent except for a factor of 2 in the bottom of the denom of first one
bonus = lambda t,a: np.sqrt(np.log((t+1)**4)/(2*N[a])) # incr t by 1 to avoid t=0
bonus = lambda t,a: 2*np.sqrt(np.log((t+1))/N[a]) # incr t by 1 to avoid t=0


#%%

#env_reinit = lambda: gym.make('BanditTwoArmedDeterministicFixed-v0')
#env_reinit = lambda: gym.make('BanditTwoArmedDependentEasy-v0')
#env_reinit = lambda: gym.make('BanditTwoArmedDependentMedium-v0')
#env_reinit = lambda: gym.make('BanditTwoArmedDependentHard-v0')

#%%

from bandits import *

num_eps = 2000
H = 100


envs = [
    {'name': 'Determ. fixed', 'f': lambda: gym.make('BanditTwoArmedDeterministicFixed-v0')},
    {'name': 'Dependent, easy', 'f': lambda: gym.make('BanditTwoArmedDependentEasy-v0')},
    {'name': 'Dependent, medium', 'f': lambda: gym.make('BanditTwoArmedDependentMedium-v0')},
    {'name': 'Dependent, hard', 'f': lambda: gym.make('BanditTwoArmedDependentHard-v0')}
]

fig, (ax1, ax2) = plt.subplots(1,2)
ax1.set_title("Rewards")
ax1.set_xlabel("Iteration")
ax1.set_ylabel("Reward")
ax2.set_title("Optimal Arm %")
ax2.set_xlabel("Iteration")
ax2.set_ylabel("Optimal Arm %")
fig.tight_layout()

def _plot(rewards, opt_arm, lab):
    ax1.plot(range(len(rewards)), rewards, label=lab)
    ax2.plot(range(len(opt_arm)), opt_arm, label=lab)

for i in range(0,len(envs)):
    ep_rewards = np.zeros((num_eps, H))
    ep_opt_arms = np.zeros((num_eps, H))
    for ep in range(num_eps):
        env = envs[i]['f']()
        Q,N = init_mab(env)
        alpha = lambda t,a: H/(H+N[a])
        #alpha = lambda t,a: 1e-4
        bonus = lambda t,a: 2*np.sqrt(np.log((t+1))/N[a]) # incr t by 1 to avoid t=0
        iter_rewards, iter_opt_arms = \
            simple_mab3(env, H, 1, Q, N, alpha, bonus)
        ep_rewards[ep] = iter_rewards[:,0]
        ep_opt_arms[ep] = iter_opt_arms[:,0]

    avg_rewards = np.average(ep_rewards, axis=0)
    avg_opt_arm = np.average(ep_opt_arms, axis=0)

    print("Average reward per episode: " + str(sum(avg_rewards) / num_eps))
    _plot(avg_rewards, avg_opt_arm, envs[i]['name'])

ax1.legend(*ax1.get_legend_handles_labels(), loc="lower right", prop={'size':6})
ax2.legend(*ax2.get_legend_handles_labels(), loc="lower right", prop={'size':6})
plt.savefig('mab-ucb1_2arm_alpha1n', dpi=300, bbox_inches='tight')
plt.show()

#%% md

In this setup, the deterministic 2-arm MAB has fixed arm probabilities 0.2 and 0.8 and is solved
easily. The other 2-arm MABs have dependent arm probabilities p and 1-p; these are selected at
the start of the problem-instance; we run 2000 such independent instances. Easy, medium, and hard
 selects p from [01,0.9], [0,25,0.75], and [0.4,0.6]. The average rewards drop lower for the
 harder case, being weighted towards 0.5, where the hardest instances lie.


#%%

env_reinit = lambda: gym.make('BanditTenArmedGaussian-v0')
env = env_reinit()
env.reset()
env.render()
Utils.show_env_space(env)

#%%

from bandits import *

alphas = [
    {'name': '1/N(a)', 'f': lambda t,a: H/(H+N[a])},
    {'name': 'const 1e-4', 'f': lambda t,a: 1e-4},
]

fig, (ax1, ax2) = plt.subplots(1,2)
ax1.set_title("Rewards")
ax1.set_xlabel("Iteration")
ax1.set_ylabel("Reward")
ax2.set_title("Optimal Arm %")
ax2.set_xlabel("Iteration")
ax2.set_ylabel("Optimal Arm %")
fig.tight_layout()

def _plot(rewards, opt_arm, lab):
    ax1.plot(range(len(rewards)), rewards, label=lab)
    ax2.plot(range(len(opt_arm)), opt_arm, label=lab)

# for MAB environment, num_eps becomes num_iters, and H is thus unused. Thus,
#   need to loop over MAB outside for # of eps:
for alpha in alphas:
    print(f"*** alpha {alpha['name']} ***")

    ep_rewards = np.zeros((num_eps, H))
    ep_opt_arms = np.zeros((num_eps, H))
    for ep in range(num_eps):
        env = env_reinit()
        Q,N = init_mab(env)
        #alpha = lambda t,a: H/(H+N[a])
        #alpha = lambda t,a: 1e-4
        #bonus = lambda t,a: np.sqrt(np.log((t+1)**4)/(2*N[a])) # incr t by 1 to avoid t=0
        bonus = lambda t,a: 2*np.sqrt(np.log((t+1))/N[a]) # incr t by 1 to avoid t=0
        iter_rewards, iter_opt_arms = \
            simple_mab3(env, H, 1, Q, N, alpha['f'], bonus)
        ep_rewards[ep] = iter_rewards[:,0]
        ep_opt_arms[ep] = iter_opt_arms[:,0]

    avg_rewards = np.average(ep_rewards, axis=0)
    avg_opt_arm = np.average(ep_opt_arms, axis=0)

    print("Average reward per episode: " + str(sum(avg_rewards) / num_eps))
    _plot(avg_rewards, avg_opt_arm, alpha['name'])

ax1.legend(*ax1.get_legend_handles_labels(), loc="lower right", prop={'size':6})
ax2.legend(*ax2.get_legend_handles_labels(), loc="lower right", prop={'size':6})
plt.savefig('mab-ucb1_10arm-gaussian_alpha-1n-const', dpi=300, bbox_inches='tight')
plt.show()


#%% md
```

```
We see that the UCB-1 algorithm with an adaptive alpha (1/N(a)) does better than the
constant-alpha that works for non-stationary problems. The constant-alpha version's learning
flat-lines very quickly (to $\approx60\%$), demonstrating that a constant alpha sacrifices past
learning experience in favor of new ones, thus limiting a threshold to which it can exploit its
learned values. In contrast, while the alpha=1/N(a) is able to learn up to > 70\%, but has the
drawback of learning collapsing to 0 as 1/N(a) approaches 0 as N(a) approaches infinity, thus not
being able to learn new experiences over time (ie, stationary problems where the arm probabilities
 may change).

#%%

env_reinit = lambda: gym.make('BanditTenArmedGaussian-v0')
env = env_reinit()
env.reset()
env.render()
Utils.show_env_space(env)
```

Listing 19: Linear Function Approximation journal

```
#%%

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

#%%

import gym
import matplotlib.pyplot as plt
import numpy as np
import random
import gym_minigrid
from gym_minigrid.wrappers import *
from utils import *

np.random.seed(0)

#%% md

##### Linear Function Approximation

#%%

from features import *
from qlearn import *
from minigrid import *
import math

#env = gym.make('MiniGrid-Simple-Empty-5x5-v0')
#env = gym.make('MiniGrid-Simple-RandomWalkGrid-10x10-v0')
def env_init():
    env = gym.make('MiniGrid-Simple-RandomWalk-20-v0')
    env = FullyObsWrapper(env)
    return env

env = env_init()
env.reset()
env.render()
Utils.show_env_space(env)

num_eps = 2000
H = 100
discount = 0.6

alpha = lambda t,a: (H+1)/(H+t)
epsilon = (0.5, 0.1, 0.01)
#state_fn = state_fn_2dtile0
state_fn = state_fn_agentpos
Q,N,E = init_q(env)

ep_lens, ep_rewards, ep_opt_axn, ep_epsilons = \
    qlearn(env, num_eps, H, Q, N, state_fn, alpha, epsilon, discount, True)

tot_ep_rewards = np.sum(ep_rewards, axis=1)
print("Average reward per episode: " + str(sum(tot_ep_rewards) / num_eps))

# save the results from above for the estimated true values:
Q_true = np.copy(Q)
# convert to V-values:
V_true = np.max(Q_true[20:40,0:2], axis=1)


#%%

# run with state aggregation
state_fn = state_fn_2dtile0

Q,N,E = init_q(env)
ep_lens, ep_rewards, ep_opt_axn, ep_epsilons = \
    qlearn(env, num_eps, H, Q, N, state_fn, alpha, epsilon, discount, True)
tot_ep_rewards = np.sum(ep_rewards, axis=1)
print("Average reward per episode: " + str(sum(tot_ep_rewards) / num_eps))

V = np.max(Q[0:10,0:2], axis=1)


#%%

# as reward is 1, state-transitions (living reward) are 0,
#   the v-value at a state thus encodes the probability of reaching
#   the goal
V_normed = V/np.max(V)
# by definition, the goal state has a v-value of 0. but, when the
#   states are aggregated, the goal-state gets agregretaed as well, and so
#   it now 'has' a reward of the states with which it has been aggregated
#   so, v_normed_x2[0] has a v-value of 1 instead of 0 as a true goal-state would have.
V_normed_x2 = np.repeat(V_normed,2)
# for our purposes of illustration, however, we can therefore count the
#   goal state for the true v-values as having a v-value of 1, as well
V_true[0] = 1
V_true[19] = 1


#%%

# the true-values over their state distribution:
plt.xticks(range(len(V_true)))
plt.plot(range(len(V_true)), V_true, label='true')
plt.plot(range(len(V_true)), V_normed_x2, label='state-aggreg.')
plt.title('V-Values of State-Aggregation for Random-Walk 20')
plt.xlabel('State Number')
plt.ylabel('Reward')
plt.legend(loc="lower right", prop={'size':6})
plt.savefig('lva--saggr--rw20--vvals', dpi=300, bbox_inches='tight')
plt.show()


#%%

# Verifies the start positions are normally distributed
start_positions = []
for ep in range(num_eps):
    env = env_init()
    start_positions.append(env.unwrapped.agent_pos[0])
plt.hist(start_positions)
plt.title('Histogram of Agent Starting States in Random-Walk 20')
plt.savefig('lva--saggr--rw20--startpos', dpi=300, bbox_inches='tight')
plt.show()


#%% md

##### tile-code (using lfa to represent tile-code state)

#%%
from minigrid import *
from features import *
from qlearn import *


def env_init():
    #env = gym.make('MiniGrid-Simple-RandomWalk-20-v0')
    env = gym.make('MiniGrid-Simple-Empty-6x6-v0')
    env = FullyObsWrapper(env)
    return env

def _new_plot():
    def _plot(rewards, lab):
        plt.plot(range(len(rewards)), rewards, label=lab)
        plt.xlabel('Episode')
        plt.ylabel('Average Cumulative-Reward')
        plt.title('Cumulative-Rewards over Episodes')
        #plt.ylim(ymin=ymin)
    def _end(fname=None):
        plt.legend(loc="lower right", prop={'size':6})
        if fname is not None:
            plt.savefig(fname, dpi=300, bbox_inches='tight')
        plt.show()
    return _plot, _end

_plot, _end = _new_plot()

env = env_init()
env.reset()
env.render()
Utils.show_env_space(env)

num_eps = 2000
H = 100
discount = 0.6
```

```
alpha = lambda t,a: (H+1)/(H+t)
epsilon = (0.5, 0.1, 0.01)

# run with tile-coding on qlearn-continuous
#state_fn = lambda env, state: env.unwrapped.agent_pos
#feat_fn = create_feat_fn_2dtile(env, tile_size=(2,2))

state_fn = state_fn_agentpos
feat_fn = create_feat_fn_identity(
    env,
    env.unwrapped.width*env.unwrapped.height, env.action_space.n)

weights = np.zeros_like(feat_fn(state_fn(env),1))

ep_lens, ep_rewards, ep_opt_axn, ep_epsilons = \
    qlearn_cont(env, num_eps, H, weights, state_fn, feat_fn, alpha, epsilon, discount)

print('weights: ', weights)

tot_ep_rewards = np.sum(ep_rewards, axis=1)
print("Average reward per episode: " + str(sum(tot_ep_rewards) / num_eps))


#%%
_plot(tot_ep_rewards, 'tile2d lfa')

_end()

#%%

weights2 = weights.reshape(64,4)
```