



RL homework 3

Name: Yuan Zhang

SN: 17044633

Start date: 7th March 2018

Due date: 21st March 2018, 11:55 pm

Assignment 2: Analyse Results

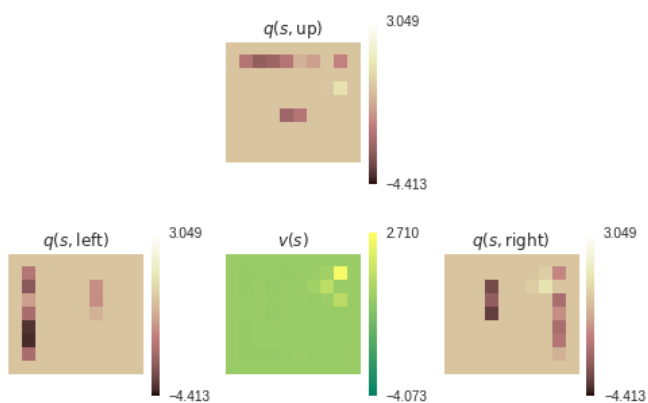
2.1 Tabular Learning

2.1.1 Data Efficiency

Online Q-learning

- number_of_steps = $1e3$ and num_offline_updates = 0

```
1 grid = Grid()
2 agent = ExperienceQ(
3     grid._layout.size, 4, grid.get_obs(),
4     random_policy, num_offline_updates=0, step_size=0.1)
5 run_experiment(grid, agent, int(1e3))
6 q = agent.q_values.reshape(grid._layout.shape + (4,))
7 plot_action_values(q)
```





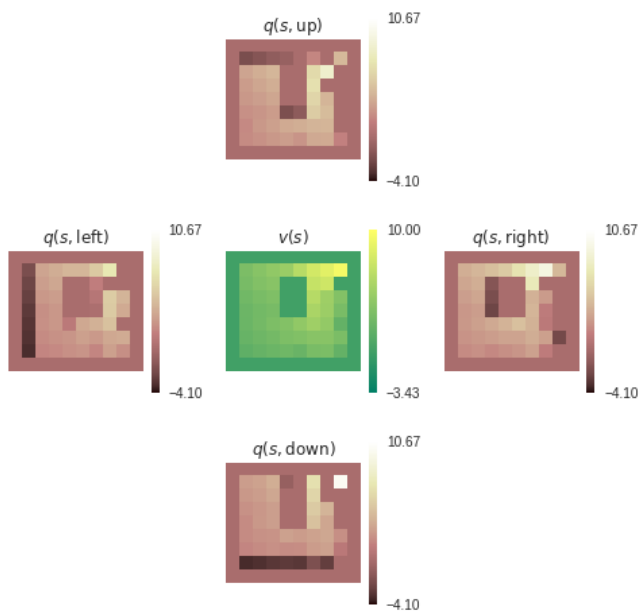
Experience Replay

- number_of_steps = $1e3$ and num_offline_updates = 30

```

1 grid = Grid()
2 agent = ExperienceQ(
3     grid._layout.size, 4, grid.get_obs(),
4     random_policy, num_offline_updates=30, step_size=0.1)
5 run_experiment(grid, agent, int(1e3))
6 q = agent.q_values.reshape(grid._layout.shape + (4,))
7 plot_action_values(q)

```



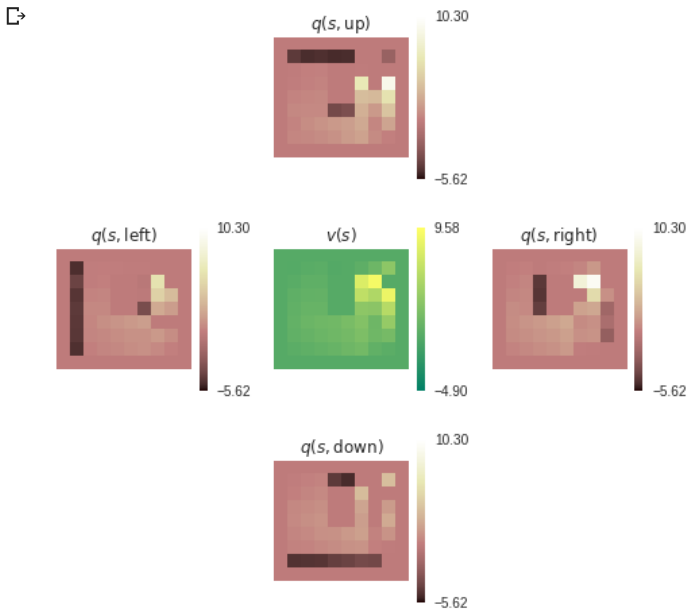
DynaQ

- number_of_steps = $1e3$ and num_offline_updates = 30

```

1 grid = Grid()
2 agent = DynaQ(
3     grid.layout.size, 4, grid.get_obs(),
4     random_policy, num_offline_updates=30, step_size=0.1)
5 run_experiment(grid, agent, int(1e3))
6 q = agent.q_values.reshape(grid.layout.shape + (4,))
7 plot_action_values(q)

```



▼ 2.1.2 Computational Cost

What if sampling from the environment is cheap and I don't care about data efficiency but only care about the number of updates to the model?

How do Online Q-learning, ExperienceReplay and Dyna-Q compare if I apply the same number of total updates?

Online Q-learning

- number_of_steps = $3e4$ and num_offline_updates = 0

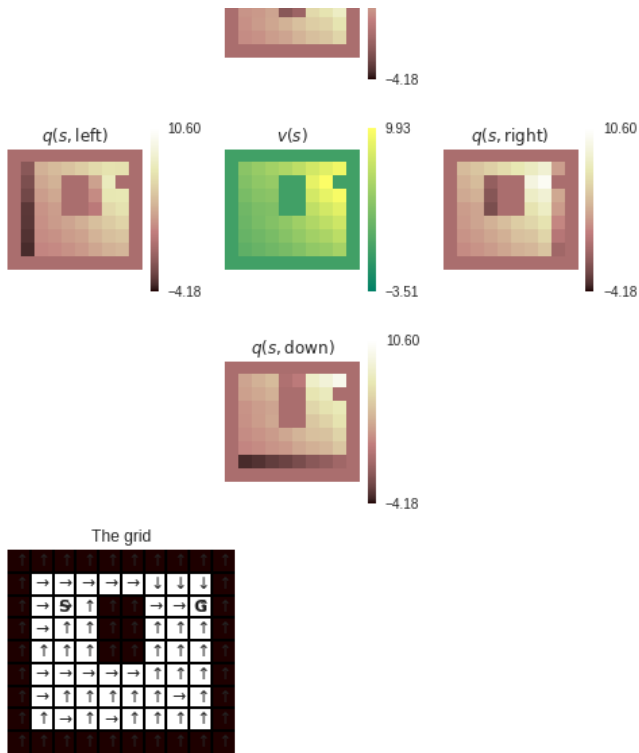
```

1 grid = Grid()
2 agent = ExperienceQ(
3     grid.layout.size, 4, grid.get_obs(),
4     random_policy, num_offline_updates=0, step_size=0.1)
5 run_experiment(grid, agent, int(3e4))
6 q = agent.q_values.reshape(grid.layout.shape + (4,))
7 plot_action_values(q)
8 plot_greedy_policy(grid, q)

```

⚠ /usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:58: MatplotlibDeprecat
Future behavior will be consistent with the long-time default:
plot commands add elements without first clearing the
Axes and/or Figure.





ExperienceReplay

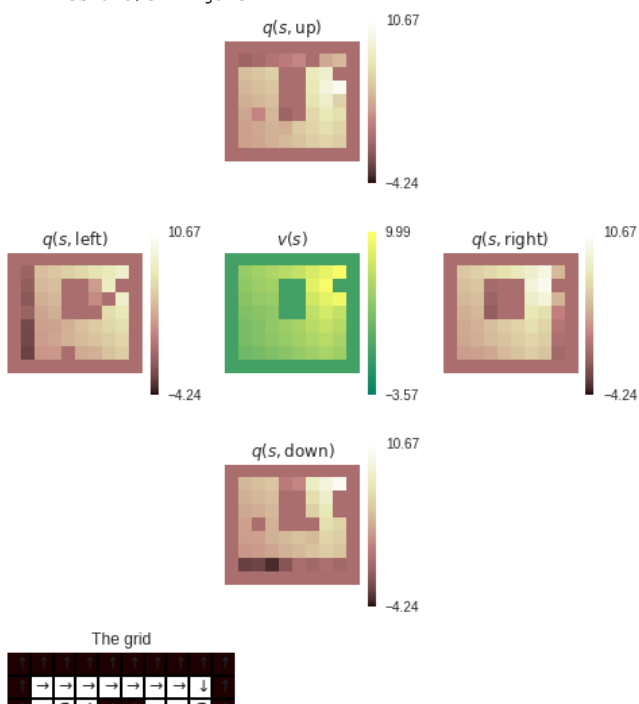
- number_of_steps = 1e3 and num_offline_updates = 30

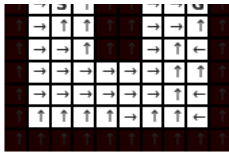
```

1 grid = Grid()
2 agent = ExperienceQ(
3     grid.layout.size, 4, grid.get_obs(),
4     random_policy, num_offline_updates=30, step_size=0.1)
5 run_experiment(grid, agent, int(1e3))
6 q = agent.q_values.reshape(grid.layout.shape + (4,))
7 plot_action_values(q)
8 plot_greedy_policy(grid, q)

```

➤ /usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:58: MatplotlibDeprecat
Future behavior will be consistent with the long-time default:
plot commands add elements without first clearing the
Axes and/or Figure.





DynaQ

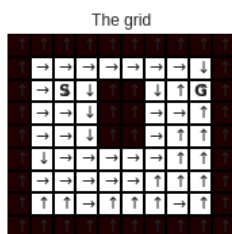
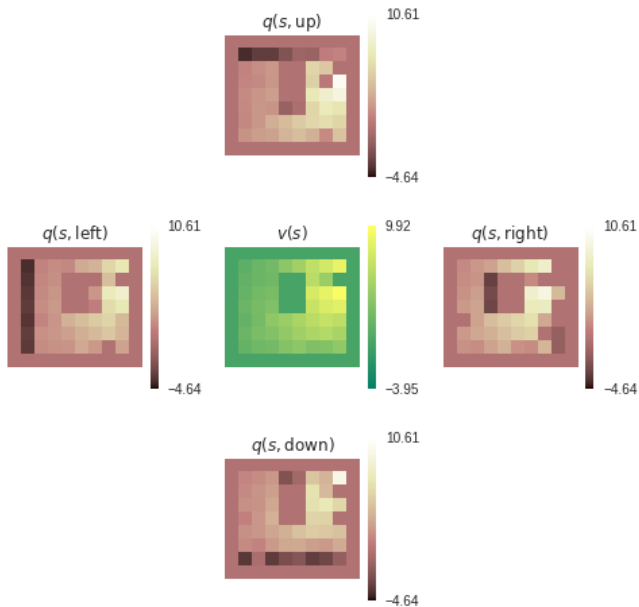
- number_of_steps = 1e3 and num_offline_updates = 30

```

1 grid = Grid()
2 agent = DynaQ(
3     grid._layout.size, 4, grid.get_obs(),
4     random_policy, num_offline_updates=30, step_size=0.1)
5 run_experiment(grid, agent, int(1e3))
6 q = agent.q_values.reshape(grid._layout.shape + (4,))
7 plot_action_values(q)
8 plot_greedy_policy(grid, q)

```

⚠ /usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:58: MatplotlibDeprecat
Future behavior will be consistent with the long-time default:
plot commands add elements without first clearing the
Axes and/or Figure.



2.3 Linear function approximation

We will now consider the FeatureGrid domain.

And evaluate Q-learning, Experience Replay and DynaQ, in the context of linear function approximation.

All experiments are run for number_of_steps = 1e5

Online Q-learning with Linear Function Approximation

```


1 grid = FeatureGrid()
2
3 agent = FeatureExperienceQ(
4     number_of_features=grid.number_of_features, number_of_actions=4,
5     number_of_states=grid._layout.size, initial_state=grid.get_obs(),
6     num_offline_updates=0, step_size=0.01, behaviour_policy=random_policy)

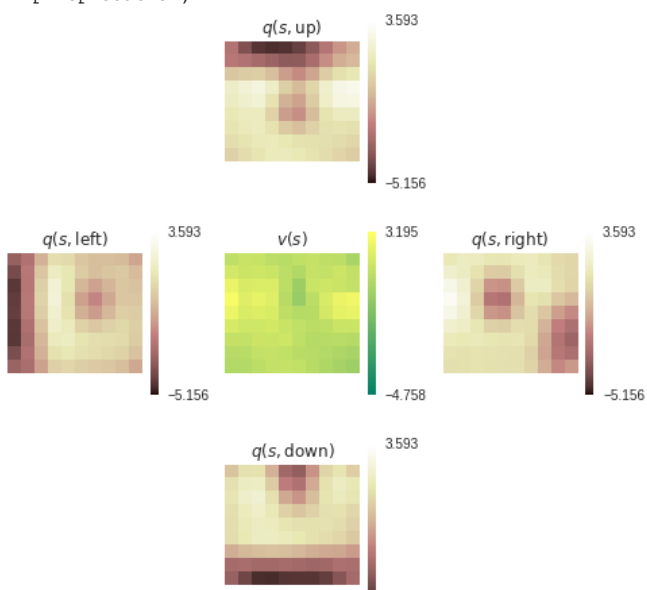
```

```

7 run_experiment(grid, agent, int(1e5))
8 q = np.reshape(
9     np.array([agent.q(grid.int_to_features(i)) for i in xrange(grid.number_of_states)]),
10    [grid._layout.shape[0], grid._layout.shape[1], 4])
11 plot_action_values(q)
12 plot_greedy_policy(grid, q)

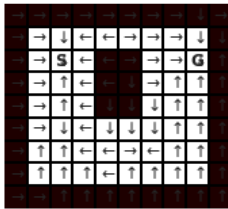
```

 /usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:58: MatplotlibDeprecat
 Future behavior will be consistent with the long-time default:
 plot commands add elements without first clearing the
 Axes and/or Figure.
 /usr/local/lib/python2.7/dist-packages/matplotlib/__init__.py:805: MatplotlibDeprec
 mplDeprecation)
 /usr/local/lib/python2.7/dist-packages/matplotlib/rcsetup.py:155: MatplotlibDepreca
 mplDeprecation)



■ -5.156

The grid



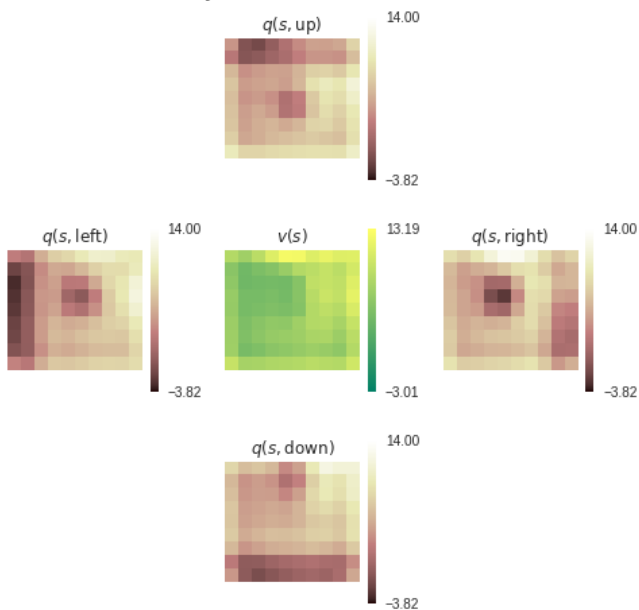
Experience Replay with Linear Function Approximation

```

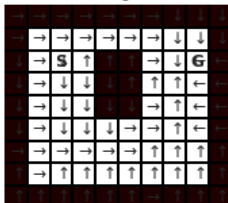
1 grid = FeatureGrid()
2
3 agent = FeatureExperienceQ(
4     number_of_features=grid.number_of_features, number_of_actions=4,
5     number_of_states=grid.layout.size, initial_state=grid.get_obs(),
6     num_offline_updates=10, step_size=0.01, behaviour_policy=random_policy)
7 run_experiment(grid, agent, int(1e5))
8 q = np.reshape(
9     np.array([agent.q(grid.int_to_features(i)) for i in xrange(grid.number_of_states)]),
10    [grid._layout.shape[0], grid._layout.shape[1], 4])
11 plot_action_values(q)
12 plot_greedy_policy(grid, q)

```

⌘ /usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:58: MatplotlibDeprecat
Future behavior will be consistent with the long-time default:
plot commands add elements without first clearing the
Axes and/or Figure.



The grid



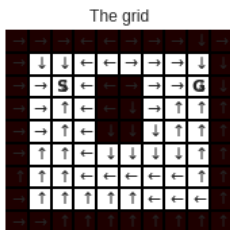
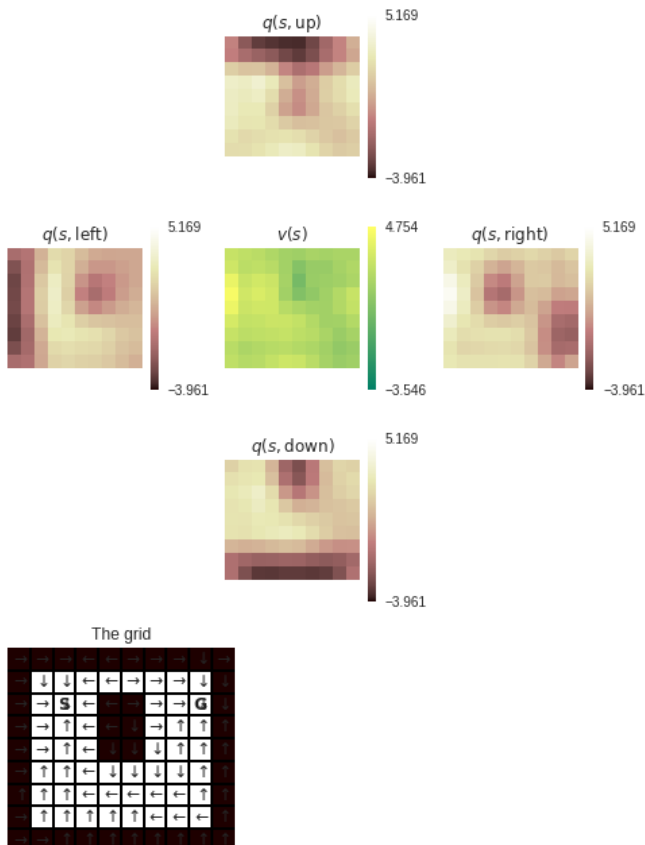
DynaQ with Linear Function Approximation

```

1 grid = FeatureGrid()
2
3 agent = FeatureDynaQ(
4     number_of_features=grid.number_of_features,
5     number_of_actions=4,
6     number_of_states=grid._layout.size,
7     initial_state=grid.get_obs(),
8     num_offline_updates=10,
9     step_size=0.01,
10    behaviour_policy=random_policy)
11
12 run_experiment(grid, agent, int(1e5))
13 q = np.reshape(
14     np.array([agent.q(grid.int_to_features(i)) for i in xrange(grid.number_of_states)]),
15     [grid._layout.shape[0], grid._layout.shape[1], 4])
16 plot_action_values(q)
17 plot_greedy_policy(grid, q)

```

⚠ /usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:58: MatplotlibDeprecat
Future behavior will be consistent with the long-time default:
plot commands add elements without first clearing the
Axes and/or Figure.



2.4 Non stationary Environments

We now consider a non-stationary setting where after `pretrain_steps` in the environment, the goal is moved to a new location (from the top-right of the grid to the bottom-left).

The agent is allowed to continue training for a (shorter) amount of time in this new setting, and then we evaluate the value estimates.

```

1 pretrain_steps = 2e4
2 new_env_steps = pretrain_steps / 30

```

Online Q-learning

```

1 # Train on first environment
2 grid = Grid()
3 agent = ExperienceQ(
4     grid._layout.size, 4, grid.get_obs(),
5     random_policy, num_offline_updates=0, step_size=0.1)
6 run_experiment(grid, agent, int(pretrain_steps))
7 q = agent.q_values.reshape(grid._layout.shape + (4,))
8 # plot_state_value(q)
9
10 # Change goal location
11 alt_grid = AltGrid()

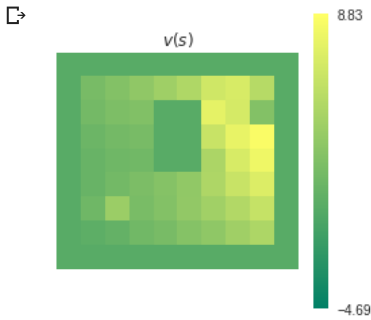
```



```

11 # Run experiment on new environment
12 run_experiment(alt_grid, agent, int(new_env_steps))
13 alt_q = agent.q_values.reshape(alt_grid._layout.shape + (4,))
14 plot_state_value(alt_q)

```

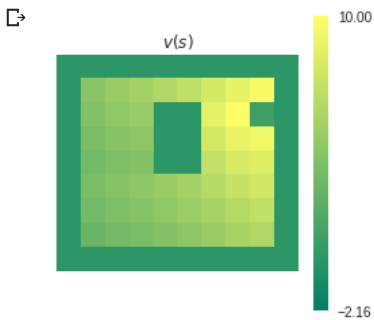


Experience Replay

```

1 # Train on first environment
2 grid = Grid()
3 agent = ExperienceQ(
4     grid._layout.size, 4, grid.get_obs(),
5     random_policy, num_offline_updates=30, step_size=0.1)
6 run_experiment(grid, agent, int(pretrain_steps))
7 q = agent.q_values.reshape(grid._layout.shape + (4,))
8 # plot_state_value(q)
9
10 # Change goal location
11 alt_grid = AltGrid()
12 run_experiment(alt_grid, agent, int(new_env_steps))
13 alt_q = agent.q_values.reshape(alt_grid._layout.shape + (4,))
14 plot_state_value(alt_q)

```

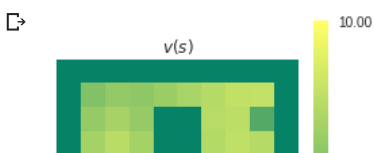


Dyna

```

1 # Train on first environment
2 grid = Grid()
3 agent = DynaQ(
4     grid._layout.size, 4, grid.get_obs(),
5     random_policy, num_offline_updates=30, step_size=0.1)
6 run_experiment(grid, agent, int(pretrain_steps))
7 q = agent.q_values.reshape(grid._layout.shape + (4,))
8 # plot_state_value(q)
9
10 # Change goal location
11 alt_grid = AltGrid()
12 run_experiment(alt_grid, agent, int(new_env_steps))
13 alt_q = agent.q_values.reshape(alt_grid._layout.shape + (4,))
14 plot_state_value(alt_q)

```





▼ Questions

Basic Tabular Learning

[5 pts] Why is the ExperienceReplay agent so much more data efficient than online Q-learning?

Because ExperienceReplay learns with the same data multiple times in offline updates. And multiple passes with the same data can make Q-learning converge faster.

[5 pts] If we run the experiments for the same number of updates, rather than the same number of steps in the environment, which among online Q-learning and Experience Replay performs better? Why?

Online Q-learning performs better than Experience Replay. For the same number of updates, online Q-learning runs more steps, all data comes from the interaction with the real world. However, for the ExperienceReplay, offline updates use the data stored in ReplayBuffer which may be outdated and biased.

[5 pts] Which among online Q-learning and Dyna-Q is more data efficient? why?

Dyna-Q is more data efficient. Because it reuses the data by feeding it into the model and updates q-value in offline updates with the updated model.

[5 pts] If we run the experiments for the same number of updates, rather than the same number of steps in the environment, which among online Q-learning and Dyna-Q performs better? Why?

Online Q-learning performs better. For the same number of updates, online Q-learning runs more steps, all data comes from the interaction with the real world. However, for the Dyna-Q, offline updates use the data generated by approximate model which may have some biases.

Linear function approximation

[5 pts] The value estimates with function approximation are considerably more blurry than in the tabular setting despite more training steps and interactions with the environment, why is this the case?

For tabular setting, one state-action pair only changes its own q-value and doesn't influence others' values in one single update. However, when using a linear function to approximate the q-value, one state-action pair will update the whole parameters, which will change the q-values for all other states. In this way, all states have a similar trend and values are more continuous or we can say blurry. We can see even for the unseen states like the walls, their borders are not so obvious, which also shows such influences by other states.

[5 pts] Inspect the policies derived by training agents with linear function approximation on FeatureGrid (as shown by plot_greedy_policy). How does this compare to the optimal policy? Are there any inconsistencies you can spot? What is the reason of these?

Its policy is worse than the optimal policy. Some adjacent grids point to each other, so the agent moves between these two states, leading to inconsistency of the policy. As said before, all states share a similar trend and it takes more time to differentiate each state. If given enough time, the policy will converge to the optimal one.

Learning in a non stationary environment

Consider now the tabular but non-stationary setting of section 2.4.

After an initial pretraining phase, the goal location is moved to a new location, where the agent is allowed to train for some (shorter) time.

[10 pts] Compare the value estimates of online Q-learning and Experience Replay, after training also on the new goal location, explain what you see.

For the Experience Replay, its value estimates are quite similar to the stationary one (Both higher values on top-right and lower values on bottom-left). It looks like that it doesn't find out the transfer of the goal. However, for the non-stationary setting of online Q-learning, there is an obvious high point on the bottom-left compared to the stationary one, but still low

values generally on bottom-left compared to the top-right. So it begins to find out the change of the goal. The reason for that is after starting training on the new goal, all updates of the q-values in online Q-learning come from the interaction with the new environment, while offline updates in Experience Replay still rely on the previous environment, which is outdated. So it finds the change of the goal much slower.

[10 pts] Compare the value estimates of online Q-learning and Dyna-Q, after training also on the new goal location, explain what you see.

Compared with online Q-learning, Dyna-Q have a larger range of high-valued regions on bottom-left. And values on bottom-left exceed the ones on top-right, which implies that it has already found the change of the goal very well. Dyna-Q utilizes new observations to update the model so that offline updates of q-values can perceive the change of the goal by using the data generated from the updated model. So Dyna-Q is more suitable to the non-stationary environment.