assignment

June 16, 2025

1 Assignment 3: Function Approximation and Control

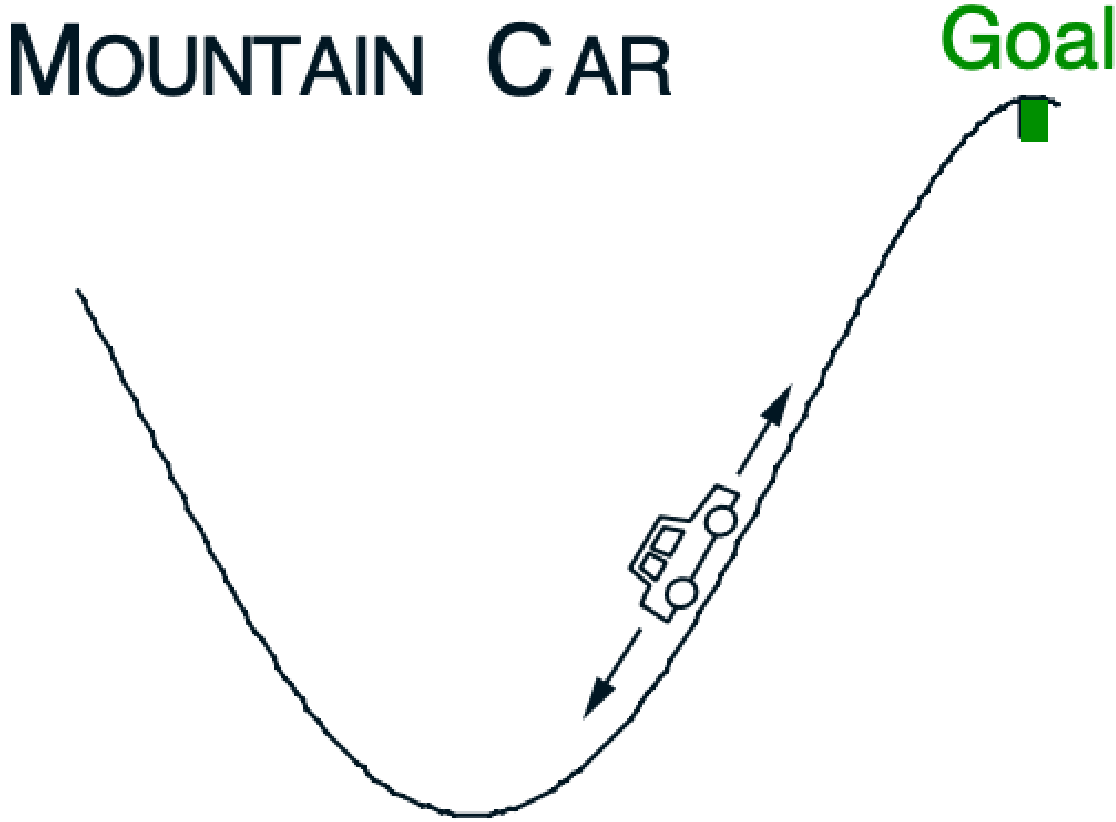
Welcome to Assignment 3. In this notebook you will learn how to: - Use function approximation in the control setting - Implement the Sarsa algorithm using tile coding - Compare three settings for tile coding to see their effect on our agent

As with the rest of the notebooks do not import additional libraries or adjust grading cells as this will break the grader.

MAKE SURE TO RUN ALL OF THE CELLS SO THE GRADER GETS THE OUTPUT IT NEEDS

[31]:
```
# Import Necessary Libraries
import numpy as np
import itertools
import matplotlib.pyplot as plt
import tiles3 as tc
from rl_glue import RLGlue
from agent import BaseAgent
from utils import argmax
import mountaincar_env
import time
```

In the above cell, we import the libraries we need for this assignment. You may have noticed that we import mountaincar_env. This is the **Mountain Car Task** introduced in Section 10.1 of the textbook. The task is for an under powered car to make it to the top of a



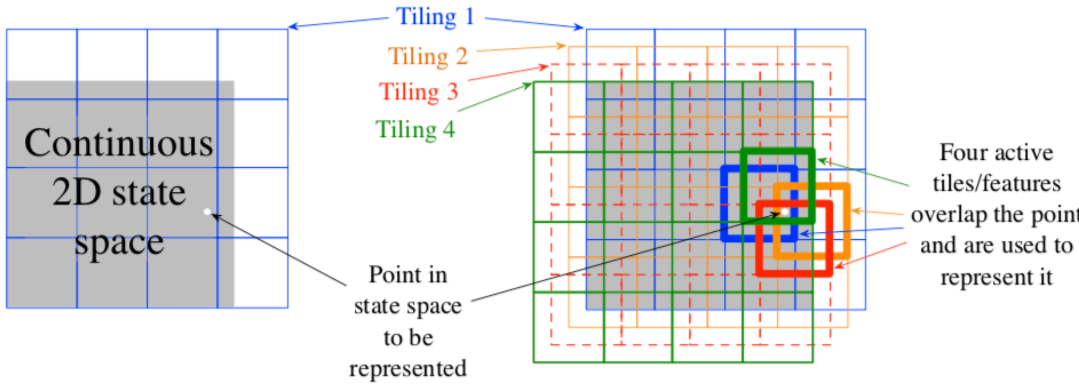
hill: The car is under-powered so the agent needs to learn to rock back and forth to get enough momentum to reach the goal. At each time step the agent receives from the environment its current velocity (a float between -0.07 and 0.07), and it's current position (a float between -1.2 and 0.5). Because our state is continuous there are a potentially infinite number of states that our agent could be in. We need a function approximation method to help the agent deal with this. In this notebook we will use tile coding. We provide a tile coding implementation for you to use, imported above with tiles3.

1.1 Section 0: Tile Coding Helper Function

To begin we are going to build a tile coding class for our Sarsa agent that will make it easier to make calls to our tile coder.

1.1.1 Tile Coding Function

Tile coding is introduced in Section 9.5.4 of the textbook of the textbook as a way to create features that can both provide good generalization and discrimination. It consists of multiple overlapping tilings, where each tiling is a partitioning of the space into tiles.



To help keep our agent code clean we are going to make a function specific for tile coding for our Mountain Car environment. To help we are going to use the Tiles3 library. This is a Python 3 implementation of the tile coder. To start take a look at the documentation: Tiles3 documentation To get the tile coder working we need to implement a few pieces: - First: create an index hash table - this is done for you in the init function using tc.IHT. - Second is to scale the inputs for the tile coder based on the number of tiles and the range of values each input could take. The tile coder needs to take in a number in range [0, 1], or scaled to be [0, 1] * num_tiles. For more on this refer to the Tiles3 documentation. - Finally we call tc.tiles to get the active tiles back.

[32]:
```
# -----------
# Graded Cell
# -----------
class MountainCarTileCoder:
    def __init__(self, iht_size=4096, num_tilings=8, num_tiles=8):
        """
        Initializes the MountainCar Tile Coder
        Initializers:
        iht_size -- int, the size of the index hash table, typically a power of␣
 ↪2
        num_tilings -- int, the number of tilings
        num_tiles -- int, the number of tiles. Here both the width and height␣
 ↪of the
                    tile coder are the same
        Class Variables:
        self.iht -- tc.IHT, the index hash table that the tile coder will use
        self.num_tilings -- int, the number of tilings the tile coder will use
        self.num_tiles -- int, the number of tiles the tile coder will use
        """
        self.iht = tc.IHT(iht_size)
        self.num_tilings = num_tilings
        self.num_tiles = num_tiles

    def get_tiles(self, position, velocity):
        """
        Takes in a position and velocity from the mountaincar environment
```

```
        and returns a numpy array of active tiles.

        Arguments:
        position -- float, the position of the agent between -1.2 and 0.5
        velocity -- float, the velocity of the agent between -0.07 and 0.07
        returns:
        tiles - np.array, active tiles
        """
        # Use the ranges above and self.num_tiles to scale position and␣
↪velocity to the range [0, 1]
        # then multiply that range with self.num_tiles so it scales from [0,␣
↪num_tiles]

        position_scaled = 0
        velocity_scaled = 0

        # ----------------
        # your code here
        pos_min, pos_max = -1.2, 0.5
        vel_min, vel_max = -0.07, 0.07

        # linearly scale into [0, num_tiles]
        position_scaled = (position - pos_min) / (pos_max - pos_min) * self.
↪num_tiles
        velocity_scaled = (velocity - vel_min) / (vel_max - vel_min) * self.
↪num_tiles

        # ----------------

        # get the tiles using tc.tiles, with self.iht, self.num_tilings and␣
↪[scaled position, scaled velocity]
        # nothing to implment here
        tiles = tc.tiles(self.iht, self.num_tilings, [position_scaled,␣
↪velocity_scaled])

        return np.array(tiles)
```

[33]:
```
# -----------
# Tested Cell
# -----------
# The contents of the cell will be tested by the autograder.
# If they do not pass here, they will not pass there.

# create a range of positions and velocities to test
# then test every element in the cross-product between these lists
pos_tests = np.linspace(-1.2, 0.5, num=5)
```

```python
vel_tests = np.linspace(-0.07, 0.07, num=5)
tests = list(itertools.product(pos_tests, vel_tests))

mctc = MountainCarTileCoder(iht_size=1024, num_tilings=8, num_tiles=2)

t = []
for test in tests:
    position, velocity = test
    tiles = mctc.get_tiles(position=position, velocity=velocity)
    t.append(tiles)

expected = [
    [0, 1, 2, 3, 4, 5, 6, 7],
    [0, 1, 8, 3, 9, 10, 6, 11],
    [12, 13, 8, 14, 9, 10, 15, 11],
    [12, 13, 16, 14, 17, 18, 15, 19],
    [20, 21, 16, 22, 17, 18, 23, 19],
    [0, 1, 2, 3, 24, 25, 26, 27],
    [0, 1, 8, 3, 28, 29, 26, 30],
    [12, 13, 8, 14, 28, 29, 31, 30],
    [12, 13, 16, 14, 32, 33, 31, 34],
    [20, 21, 16, 22, 32, 33, 35, 34],
    [36, 37, 38, 39, 24, 25, 26, 27],
    [36, 37, 40, 39, 28, 29, 26, 30],
    [41, 42, 40, 43, 28, 29, 31, 30],
    [41, 42, 44, 43, 32, 33, 31, 34],
    [45, 46, 44, 47, 32, 33, 35, 34],
    [36, 37, 38, 39, 48, 49, 50, 51],
    [36, 37, 40, 39, 52, 53, 50, 54],
    [41, 42, 40, 43, 52, 53, 55, 54],
    [41, 42, 44, 43, 56, 57, 55, 58],
    [45, 46, 44, 47, 56, 57, 59, 58],
    [60, 61, 62, 63, 48, 49, 50, 51],
    [60, 61, 64, 63, 52, 53, 50, 54],
    [65, 66, 64, 67, 52, 53, 55, 54],
    [65, 66, 68, 67, 56, 57, 55, 58],
    [69, 70, 68, 71, 56, 57, 59, 58],
]
assert np.all(expected == np.array(t))
```

1.2 Section 1: Sarsa Agent

We are now going to use the functions that we just created to implement the Sarsa algorithm. Recall from class that Sarsa stands for State, Action, Reward, State, Action.

For this case we have given you an argmax function similar to what you wrote back in Course 1 Assignment 1. Recall, this is different than the argmax function that is used by numpy, which returns the first index of a maximum value. We want our argmax function to arbitrarily break ties, which is what the imported argmax function does. The given argmax function takes in an array of values and returns an int of the chosen action: argmax(action values)

There are multiple ways that we can deal with actions for the tile coder. Here we are going to use one simple method - make the size of the weight vector equal to (iht_size, num_actions). This will give us one weight vector for each action and one weight for each tile.

Use the above function to help fill in select_action, agent_start, agent_step, and agent_end.

Hints:

1) The tile coder returns a list of active indexes (e.g. [1, 12, 22]). You can index a numpy array using an array of values - this will return an array of the values at each of those indices. So in order to get the value of a state we can index our weight vector using the action and the array of tiles that the tile coder returns:

`self.w[action][active_tiles]`

This will give us an array of values, one for each active tile, and we sum the result to get the value of that state-action pair.

2) In the case of a binary feature vector (such as the tile coder), the derivative is 1 at each of the active tiles, and zero otherwise.

[34]:
```
# -----------
# Graded Cell
# -----------
class SarsaAgent(BaseAgent):
    def __init__(self):
        # khởi tạo trước, được set trong agent_init
        self.last_action = None
        self.previous_tiles = None
        self.epsilon = None
        self.gamma = None
        self.alpha = None
        self.iht_size = None
        self.num_tilings = None
        self.num_tiles = None
        self.num_actions = None
        self.initial_weights = None
        self.w = None
        self.tc = None

    def agent_init(self, agent_info={}):
        """Setup for the agent called when the experiment first starts."""
        self.num_tilings    = agent_info.get("num_tilings", 8)
        self.num_tiles      = agent_info.get("num_tiles", 8)
        self.iht_size       = agent_info.get("iht_size", 4096)
        self.epsilon        = agent_info.get("epsilon", 0.0)
        self.gamma          = agent_info.get("gamma", 1.0)
```

```
        # note: chia alpha cho num_tilings nếu muốn
        self.alpha          = agent_info.get("alpha", 0.5) / self.num_tilings
        self.initial_weights= agent_info.get("initial_weights", 0.0)
        self.num_actions    = agent_info.get("num_actions", 3)

        # khởi tạo weight matrix cho mỗi action
        self.w = np.ones((self.num_actions, self.iht_size)) * self.
↪initial_weights

        # khởi tạo tile coder
        self.tc = MountainCarTileCoder(self.iht_size, self.num_tilings, self.
↪num_tiles)

    def select_action(self, tiles):
        """
        Selects an action using epsilon-greedy.
        Returns (chosen_action, action_value).
        """
        # tính giá trị cho mỗi action
        action_values = np.zeros(self.num_actions)
        for action in range(self.num_actions):
            action_values[action] = np.sum(self.w[action][tiles])

        # epsilon-greedy
        if np.random.random() < self.epsilon:
            chosen_action = np.random.randint(self.num_actions)
        else:
            chosen_action = argmax(action_values)

        return chosen_action, action_values[chosen_action]

    def agent_start(self, state):
        """Called at the start of an episode, returns first action."""
        position, velocity = state
        active_tiles = self.tc.get_tiles(position, velocity)
        current_action, _ = self.select_action(active_tiles)

        self.last_action    = current_action
        self.previous_tiles = np.copy(active_tiles)
        return self.last_action

    def agent_step(self, reward, state):
        """A step taken by the agent, returns next action."""
        position, velocity = state

        # 1) compute next tiles & next action
        active_tiles = self.tc.get_tiles(position, velocity)
```

```
        current_action, action_value = self.select_action(active_tiles)

        # 2) tính giá trị cũ
        last_value = np.sum(self.w[self.last_action][self.previous_tiles])

        # 3) TD-error và cập nhật w
        td_error = reward + self.gamma * action_value - last_value
        self.w[self.last_action][self.previous_tiles] += self.alpha * td_error

        # 4) lưu lại cho bước sau
        self.last_action    = current_action
        self.previous_tiles = np.copy(active_tiles)
        return self.last_action

    def agent_end(self, reward):
        """Called when the episode terminates (no next state)."""
        # chỉ dùng reward vì không có next state
        last_value = np.sum(self.w[self.last_action][self.previous_tiles])
        td_error = reward - last_value
        self.w[self.last_action][self.previous_tiles] += self.alpha * td_error

    def agent_cleanup(self):
        """Cleanup done after the agent ends."""
        pass

    def agent_message(self, message):
        """A function used to pass information from the agent to the experiment.
 ↪"""
        pass
```

[35]:
```
# -----------
# Tested Cell
# -----------
# The contents of the cell will be tested by the autograder.
# If they do not pass here, they will not pass there.

np.random.seed(0)

agent = SarsaAgent()
agent.agent_init({"epsilon": 0.1})
agent.w = np.array([np.array([1, 2, 3]), np.array([4, 5, 6]), np.array([7, 8,␣
 ↪9])])

action_distribution = np.zeros(3)
for i in range(1000):
    chosen_action, action_value = agent.select_action(np.array([0,1]))
    action_distribution[chosen_action] += 1
```

```
print("action distribution:", action_distribution)
# notice that the two non-greedy actions are roughly uniformly distributed
assert np.all(action_distribution == [29, 35, 936])

agent = SarsaAgent()
agent.agent_init({"epsilon": 0.0})
agent.w = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

chosen_action, action_value = agent.select_action([0, 1])
assert chosen_action == 2
assert action_value == 15

# -----------
# test update
# -----------
agent = SarsaAgent()
agent.agent_init({"epsilon": 0.1})

agent.agent_start((0.1, 0.3))
agent.agent_step(1, (0.02, 0.1))

assert np.all(agent.w[0,0:8] == 0.0625)
assert np.all(agent.w[1:] == 0)
```

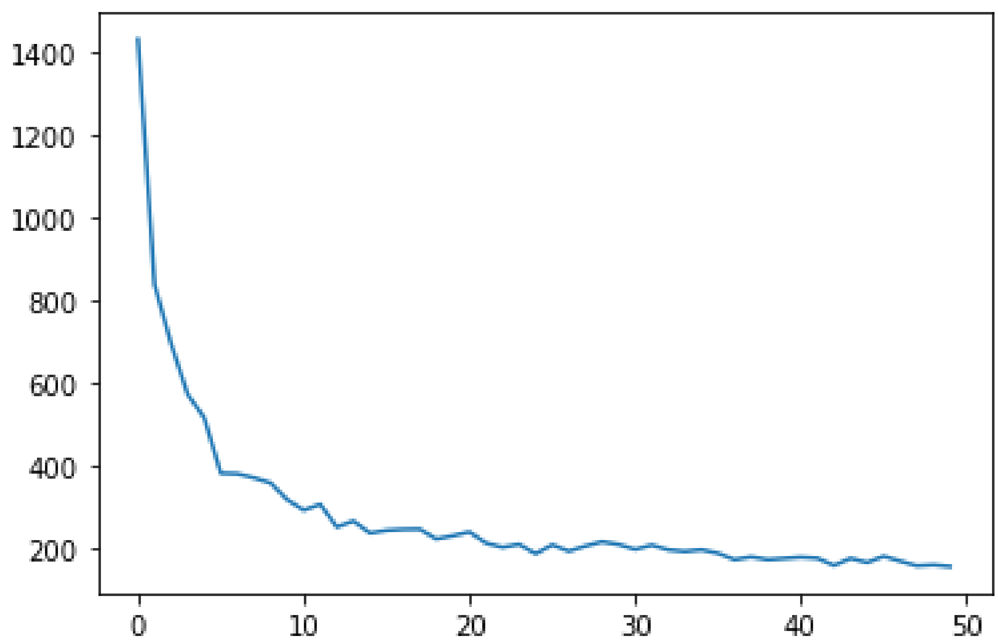
```
action distribution: [ 29.  35. 936.]
```

[36]:

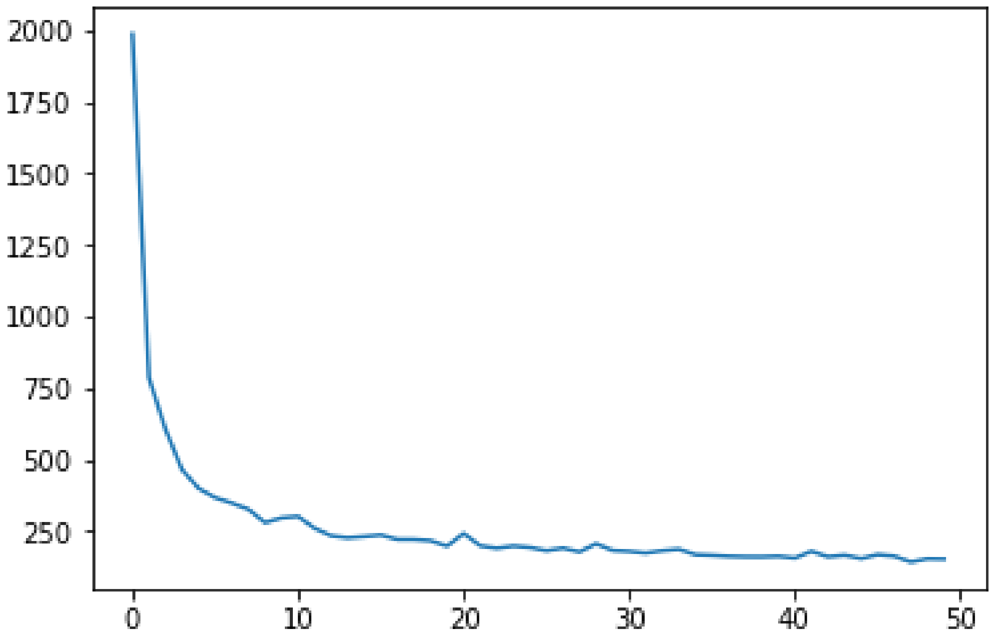
```
# -----------
# Tested Cell
# -----------
# The contents of the cell will be tested by the autograder.
# If they do not pass here, they will not pass there.

np.random.seed(0)

num_runs = 10
num_episodes = 50
env_info = {"num_tiles": 8, "num_tilings": 8}
agent_info = {}
all_steps = []

agent = SarsaAgent
env = mountaincar_env.Environment
start = time.time()

for run in range(num_runs):
    if run % 5 == 0:
```

```
        print("RUN: {}".format(run))

    rl_glue = RLGlue(env, agent)
    rl_glue.rl_init(agent_info, env_info)
    steps_per_episode = []

    for episode in range(num_episodes):
        rl_glue.rl_episode(15000)
        steps_per_episode.append(rl_glue.num_steps)

    all_steps.append(np.array(steps_per_episode))

print("Run time: {}".format(time.time() - start))

mean = np.mean(all_steps, axis=0)
plt.plot(mean)

# because we set the random seed, these values should be *exactly* the same
assert np.allclose(mean, [1432.5, 837.9, 694.4, 571.4, 515.2, 380.6, 379.4, 369.
 ↪6, 357.2, 316.5, 291.1, 305.3, 250.1, 264.9, 235.4, 242.1, 244.4, 245., 221.
 ↪2, 229., 238.3, 211.2, 201.1, 208.3, 185.3, 207.1, 191.6, 204., 214.5, 207.
 ↪9, 195.9, 206.4, 194.9, 191.1, 195., 186.6, 171., 177.8, 171.1, 174., 177.1,␣
 ↪174.5, 156.9, 174.3, 164.1, 179.3, 167.4, 156.1, 158.4, 154.4])
```

```
RUN: 0
RUN: 5
Run time: 11.211244583129883
```



The learning rate of your agent should look similar to ours, though it will not look exactly the same.If there are some spikey points that is okay. Due to stochasticity, a few episodes may have taken much longer, causing some spikes in the plot. The trend of the line should be similar, though, generally decreasing to about 200 steps per run.



This result was using 8 tilings with 8x8 tiles on each. Let's see if we can do better, and what different tilings look like. We will also text 2 tilings of 16x16 and 4 tilings of 32x32. These three choices produce the same number of features (512), but distributed quite differently.

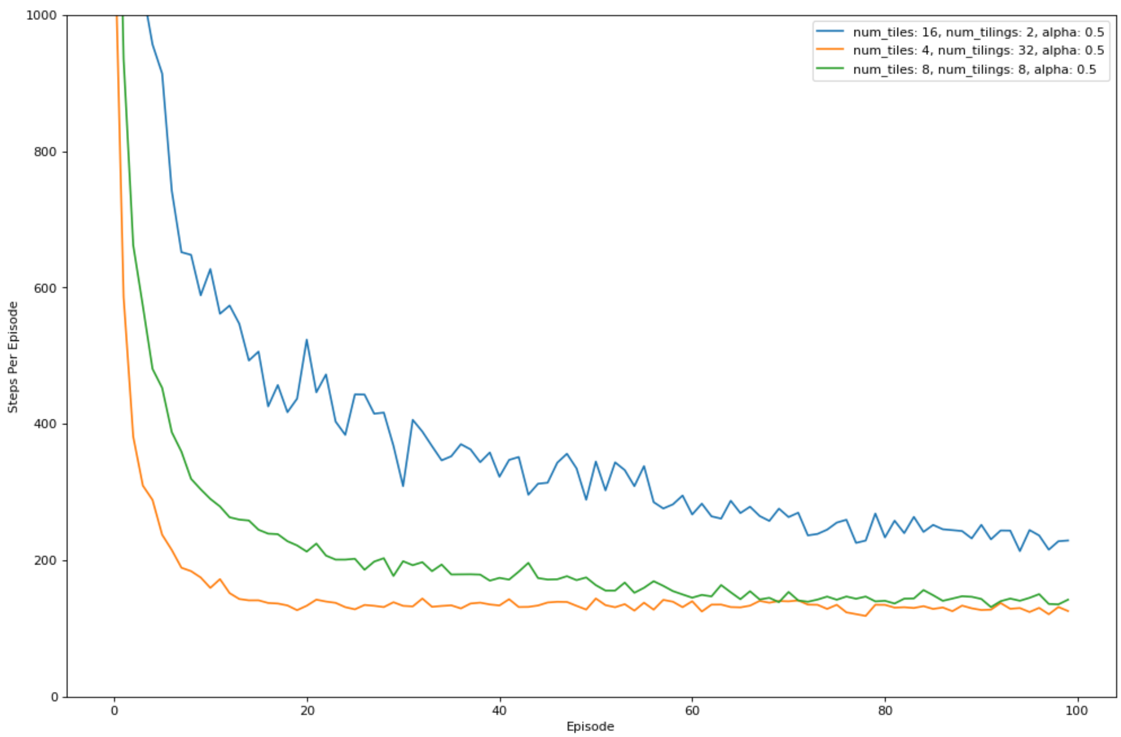
[37]:
```
# ---------------
# Discussion Cell
# ---------------

np.random.seed(0)

# Compare the three
num_runs = 20
num_episodes = 100
env_info = {}

agent_runs = []
# alphas = [0.2, 0.4, 0.5, 1.0]
alphas = [0.5]
agent_info_options = [{"num_tiles": 16, "num_tilings": 2, "alpha": 0.5},
                      {"num_tiles": 4, "num_tilings": 32, "alpha": 0.5},
                      {"num_tiles": 8, "num_tilings": 8, "alpha": 0.5}]
agent_info_options = [{"num_tiles" : agent["num_tiles"],
                       "num_tilings": agent["num_tilings"],
```

```
                            "alpha" : alpha} for agent in agent_info_options for␣
↪alpha in alphas]

agent = SarsaAgent
env = mountaincar_env.Environment
for agent_info in agent_info_options:
    all_steps = []
    start = time.time()
    for run in range(num_runs):
        if run % 5 == 0:
            print("RUN: {}".format(run))
        env = mountaincar_env.Environment

        rl_glue = RLGlue(env, agent)
        rl_glue.rl_init(agent_info, env_info)
        steps_per_episode = []

        for episode in range(num_episodes):
            rl_glue.rl_episode(15000)
            steps_per_episode.append(rl_glue.num_steps)
        all_steps.append(np.array(steps_per_episode))

    agent_runs.append(np.mean(np.array(all_steps), axis=0))
    print("stepsize:", rl_glue.agent.alpha)
    print("Run Time: {}".format(time.time() - start))

plt.figure(figsize=(15, 10), dpi= 80, facecolor='w', edgecolor='k')
plt.plot(np.array(agent_runs).T)
plt.xlabel("Episode")
plt.ylabel("Steps Per Episode")
plt.yscale("linear")
plt.ylim(0, 1000)
plt.legend(["num_tiles: {}, num_tilings: {}, alpha: {}".
↪format(agent_info["num_tiles"],
                                                                           ␣
↪agent_info["num_tilings"],
                                                                           ␣
↪agent_info["alpha"])
            for agent_info in agent_info_options])
```

```
RUN: 0
RUN: 5
RUN: 10
RUN: 15
stepsize: 0.25
Run Time: 60.07236647605896
RUN: 0
```

```
RUN: 5
RUN: 10
RUN: 15
stepsize: 0.015625
Run Time: 34.47454524040222
RUN: 0
RUN: 5
RUN: 10
RUN: 15
stepsize: 0.0625
Run Time: 34.97170352935791
```

[37]: <matplotlib.legend.Legend at 0x70a9523f7410>



Here we can see that using 32 tilings and 4 x 4 tiles does a little better than 8 tilings with 8x8 tiles. Both seem to do much better than using 2 tilings, with 16 x 16 tiles.

1.3 Section 3: Conclusion

Congratulations! You have learned how to implement a control agent using function approximation. In this notebook you learned how to:

- Use function approximation in the control setting
- Implement the Sarsa algorithm using tile coding

- Compare three settings for tile coding to see their effect on our agent