Christian Tonnesen

ID: 260847409

2A

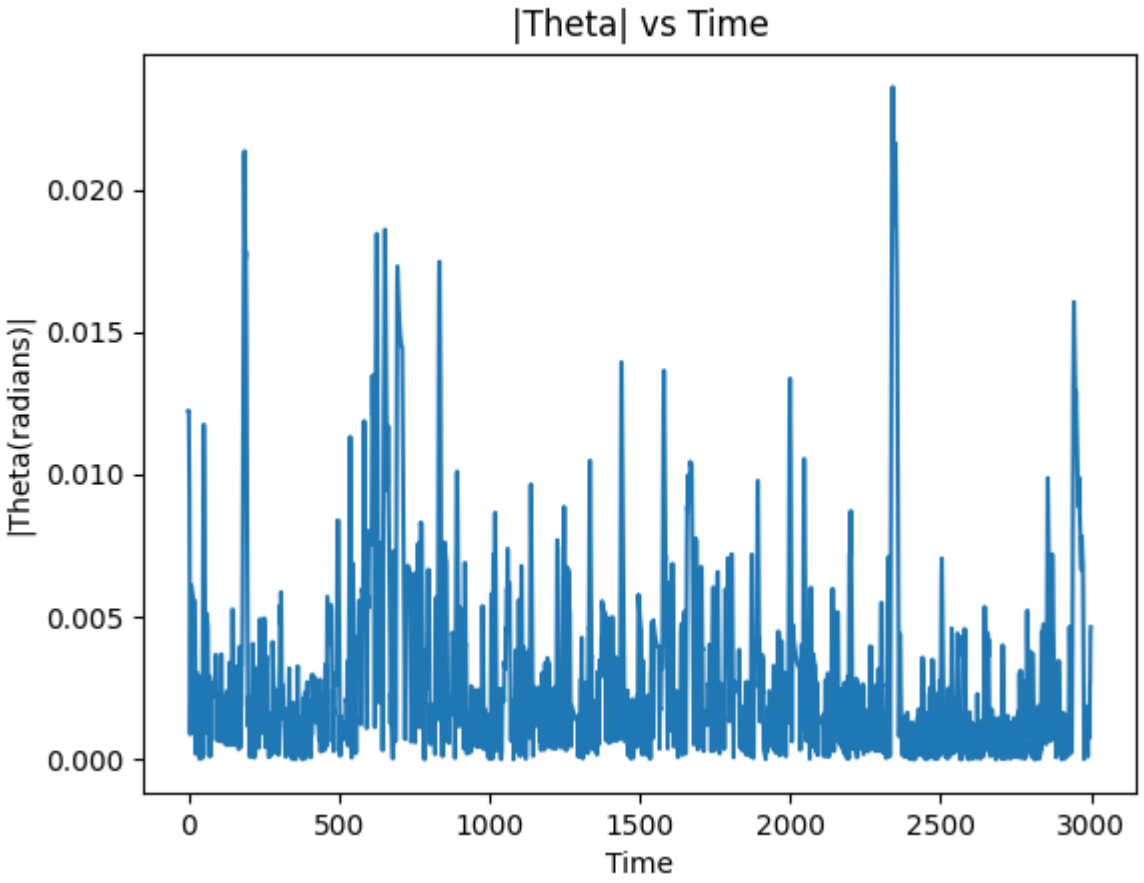
I initially had a lot of difficulty with this assignment in that the model did not appear to learn quickly, meaning the pendulum would fall most of the time. At first I consulted the Bellman Equation I was using to see whether I had ommitted something, but it appeared to be correct:

```
self.Q_value[self.prev_s[0]][self.prev_s[1]][self.prev_a] += self.lr * (reward + self.gamma * max(
            self.Q_value[theta][theta_dot]) - self.Q_value[self.prev_s[0]][self.prev_s[1]][self.prev_a
```

I then tried some initial toying with the learning rate/gamma parameters, but still did not see any noticeable learning during any of the 100 episode testing trials I was running. I also inspected the Q-Value table during these runs and noted that the rewards system was functioning correctly. For instance, with a state value of `[19,22]`, which corresponds to a `theta` around `0 radians` and a `theta_dot` around `0.875` (falling to the right). As such, checking the `Q-Table` for `[19,22]`, we see `[0.16899995859591607, 0.10889024185734208, 2.2042777437738073]` for the Q-values of moving left, nowhere, and right, respectively. The results match what we would expect, as when the pendulum is falling to the right, we should provide an action that moves the cart in the rightwards direction, of which the model predicts with the maximum value of `2.2042777437738073` out of the three options.

After posting on the discussion board about the number of episodes needed to train, I opted to try running the model for an extended period of time, to see if the model was learning.

After running the simulation overnight, I was able to reach `63,242 episodes`, which seemed to be sufficient training for the model. After that many episodes, the pole was able to successfuly stabilize, granted it did often slide off to the side of the screen slightly before the end of the max timestep. A `theta vs time` graph for the 63,242 episodes-trained model can be found below:

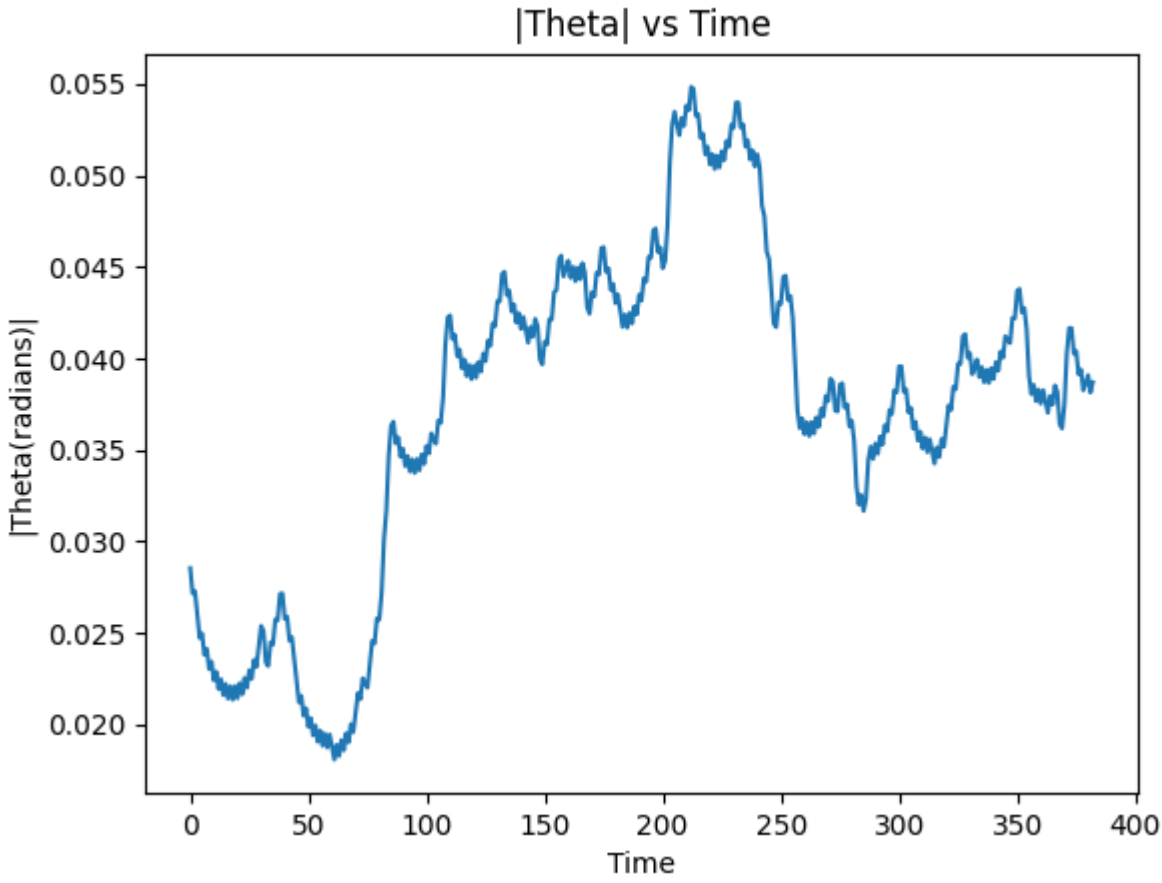


It is not known at what episode the model began self-balancing consistently, however we know that it must happen at some point between `1000` and `63,242`, since I observed that the model was not performing consistently at least up to `1000`.

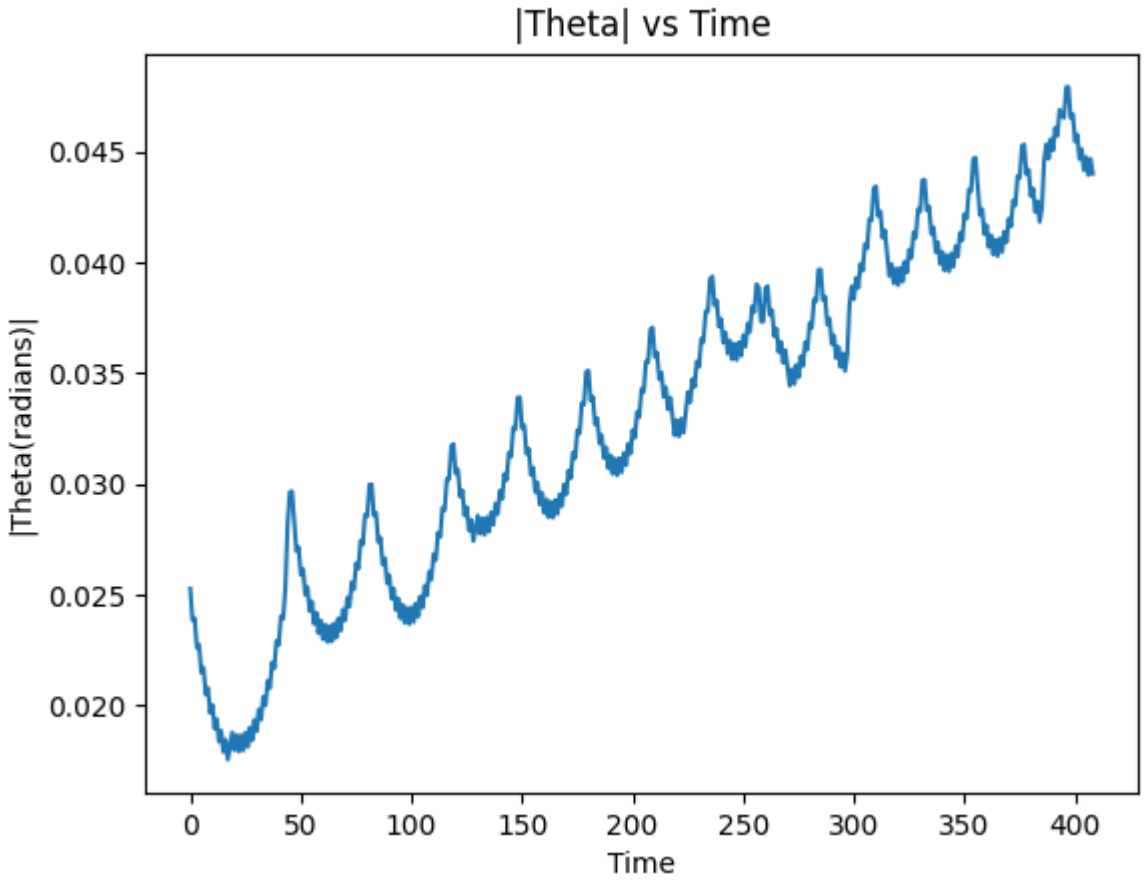
2B

In the model, the random action acts an exploratory factor. For example, in the case of `np.random.rand() > 0.8`, this means that `2` out of every `10` timesteps will result in the model choosing a random direction to move in. This raises the possibility that whatever state that model ends up in is one that it has not encountered before. This allows the model to train on new states, making it more prepared for the next time it possibly arrives there. If there was no exploratory factor, this would mean that there are states that the model would never encounter and thus be unprepared to deal with them in future episodes. By increasing the value from `0.8` to `0.9`, the model is less likely to explore (`1` in every `10` timesteps), meaning it will not be as likely to have encountered a state outside "normal" operating procedure and thus more likely to react incorrectly. This can be seen in the comparison of the following graphs:

`np.random.rand() > 0.8`




```
np.random.rand() > 0.9
```



In the `np.random.rand() > 0.9`, we see that the model is not able to stabilize as well as the `np.random.rand() > 0.8`, which is able to recover easier after the introduction a disturbance. This is because the `np.random.rand() > 0.8` has encountered the "abnormal" states more often (more likely), and thus has more training on what is the correct action to take and recover. In comparison, the `np.random.rand() > 0.9` model is less likely to have encountered these states and has less training on the correct action, resulting in it performing more poorly and falling. In essence, having more random actions means more exploration, and thus better chance of reacting correctly in these uncommon states.

2C

During the later episodes on the trained model, nearly every run would result in the cart sliding off to the side of the screen before the end of the max timestep. While we could simply adjust the bounds of the space such that the cart does not reach it before the final timestep, that would not be a very proactive solution. A better way would be to redefine our states, opting to discretize the carts `x` position on the plane and include it as a variable of the state. From there, we could tune the reward function to respond similarly to the theta difference, where an `x` value closer to the center of the bounds will result in a higher reward. This means that over time, the model would prefer actions that minimize both `theta` and `x` differences, keeping it within the bounds.

2D

The state value matrix can be found by either loading the `VTable_episode63433.npy` into Python and examining the array in the debugger, or alternatively looking at the preformatted `VTable_episode63433.csv` file. The state values consist of the highest Q-value calculated across the 3 possible actions for a given state.

2E

Given that the original model had already learned and become self-stabilizing, I needed to use fresh models to examine the changes in performance when adjusting the `lr` and `gamma` parameters. I decided that running the model for `300` episodes would serve as enough of a microcosm to observe differences. I first attempted to alter the learning rate `lr`, starting it at half the original value of `0.001`, then attempting to double the original value. The learning factor is responsible for how much the model overrides old info with new episodes, meaning that a higher learning rate would result in the model considering more recent knowledge over older runs. This can be witnessed in the trials, where the halving the learning model results in lower Q and State values, as the model takes longer to accumulate rewards (more emphasis is placed on old info). Conversely, when the learning rate is doubled (not linearly scaling, but still a positive correlation), we see higher Q and State values than control, as the model is favoring the new information more than the old. In practice, this "favoring" is the result of the Bellman Equation

```
self.Q_value[self.prev_s[0]][self.prev_s[1]][self.prev_a] += self.lr * (reward + self.gamma * max(self
```

, where the `self.lr` dictates the "weight" of the reward to be added to the previous state. As for the discount factor, or `gamma`, I tested models where the value was halved, and then quartered, since setting it to more than the original `0.99`. The discounting factor is responsible for dictating how much "weight" is assigned to later values, where a high `gamma` means that the model favors a long-term time horizon, factoring in farther along future rewards (future reward are worth more). Conversely, a lower discounting factor means the model favor more "current" rewards, where future rewards have less impact on the overall learning. In the case of our situation, a higher discounting factor resulted in a higher value for a given state (not linearly scaling, but still a positive correlation), while a lower value resulted in a lower Q-Value. This make sense as, if we revisit the Bellman Equation,:

```
self.Q_value[self.prev_s[0]][self.prev_s[1]][self.prev_a] += self.lr * (reward + self.gamma * max(self
```

the `self.gamma` is directly tied to the "future value" of `self.Q_value[theta][theta_dot]`, meaning that the higher `gamma` is, the more weight that future states will have on the current state.