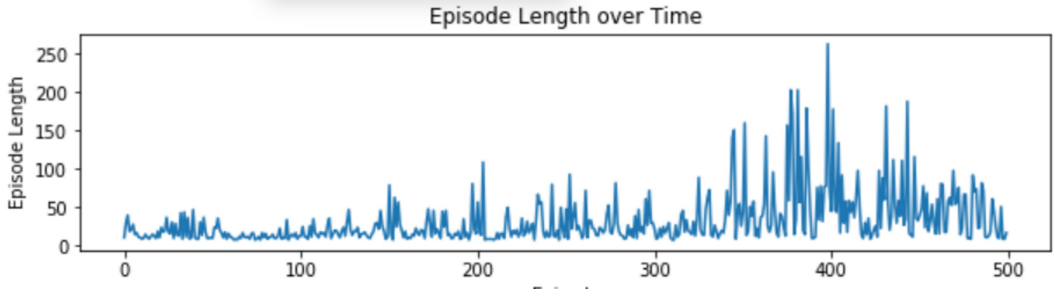
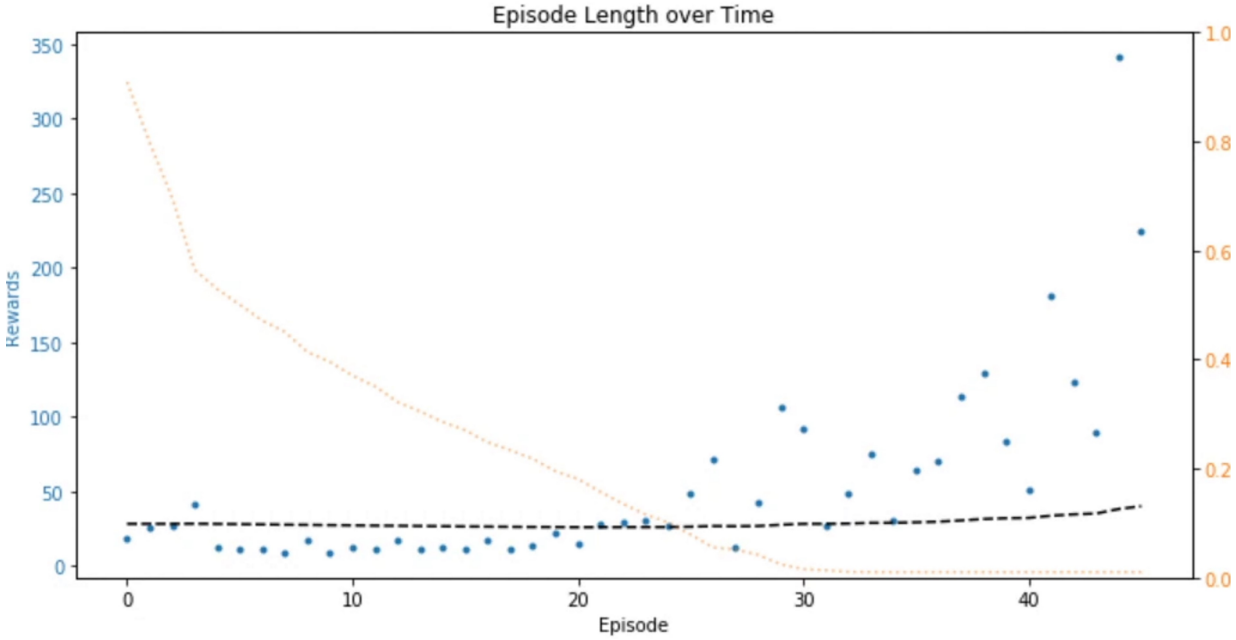
Cartpole Log

Saturday, November 23, 2019 8:20 AM

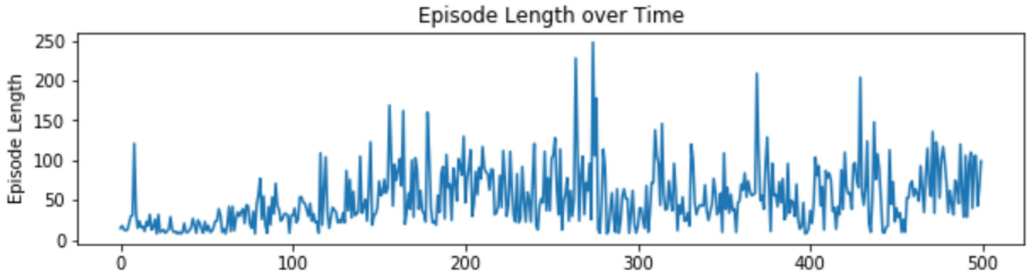
11/22/19 After working with this for a long time and having it not learn, I got it to do this:



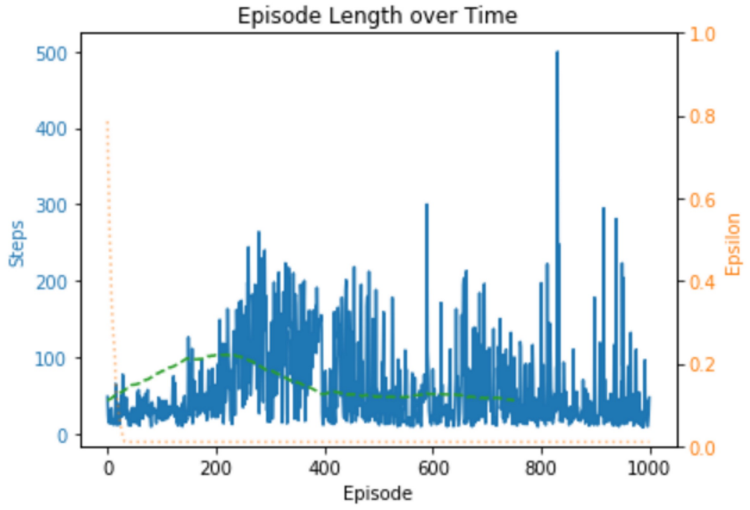
[Fast forward to a month or so later of working on this, and the results look like this: After ~40 episodes it's doing better than the other one did after 500 episodes, and it's holding on to what it learned. Plus, my chart is way better.]

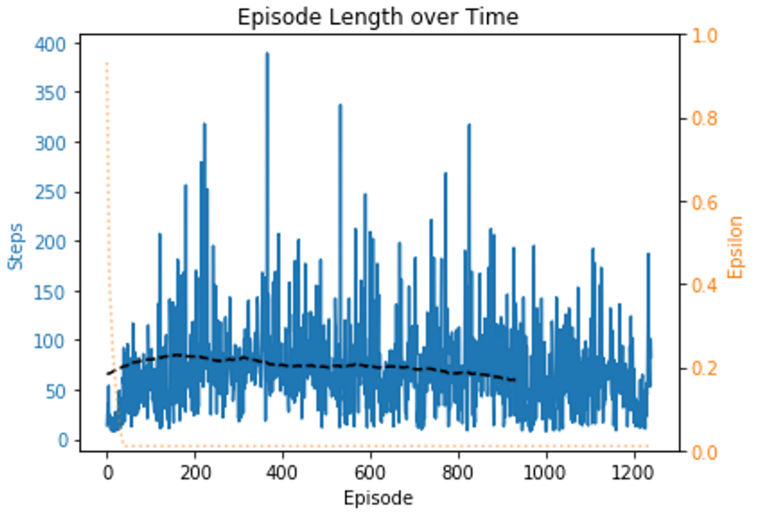


It's learning, but not holding onto its knowledge confidently.
Here's another run:

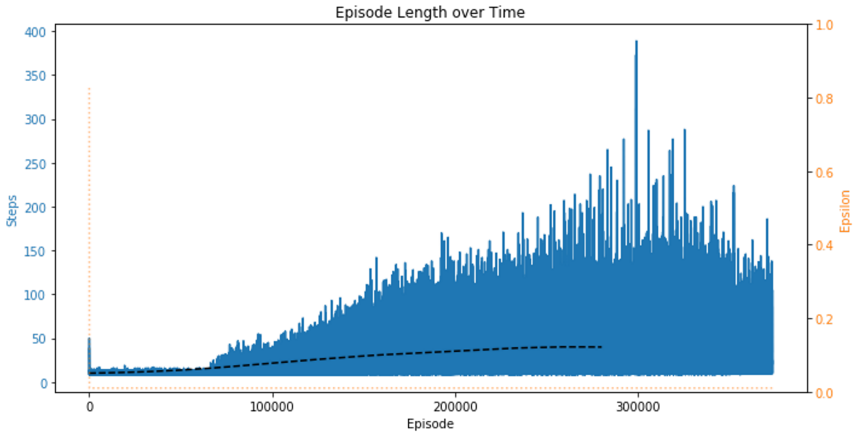


It's learning something the first 300 steps, then forgetting it:

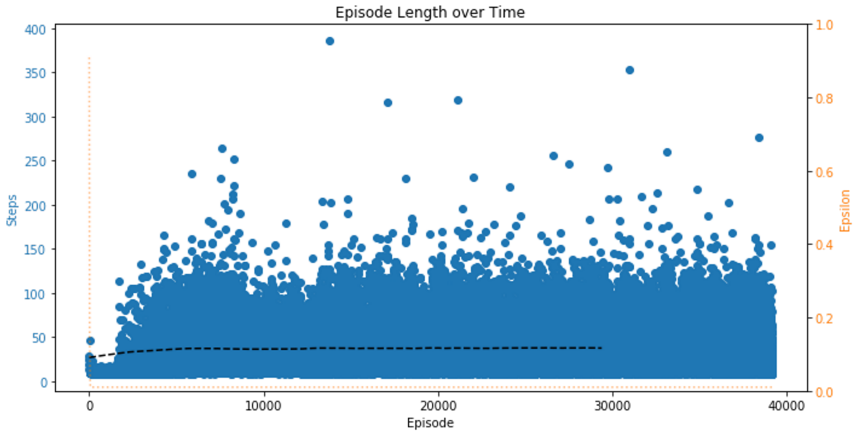




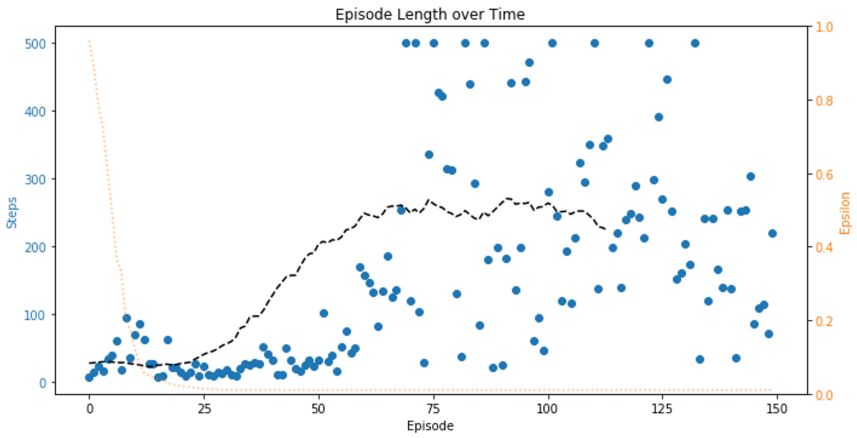
Tabular sarsa first attempt with no experience replay
It's like thousands of times faster, but learns very slowly



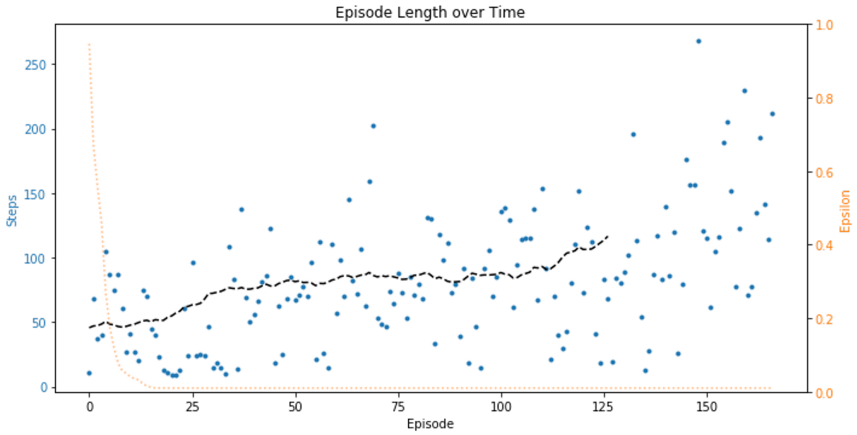
Tabular sarsa with experience replay



11/24 Re-ran the deep q again for comparison



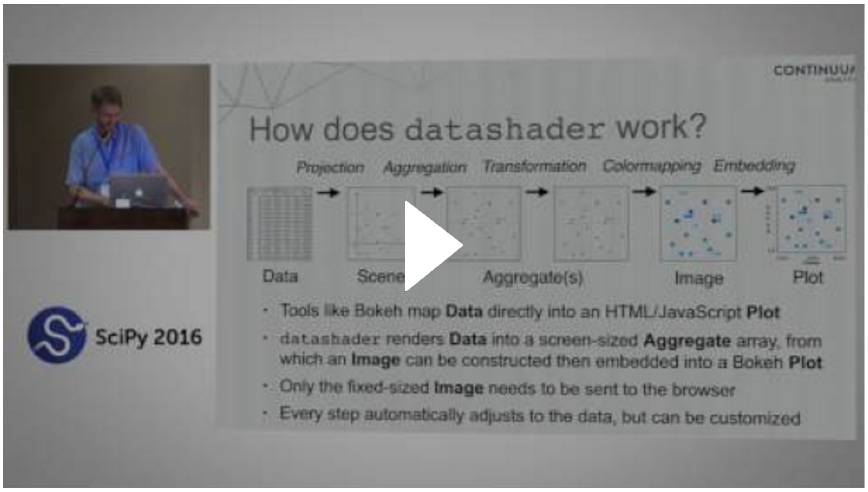
After changing future rewards to be 0 (instead of -1) for terminal state:



It's definitely worse in terms of raw performance. But it's still trending upward, which is definitely better…

★ Things sure look more clear with the updated graphing. Lesson learned: it's worth the investment to make more clear metrics

- [ ] Consider looking into bokeh and Datashader for visualization of genuinely big data
  https://www.reddit.com/r/Python/comments/4zj4ml/scatter_plot_with_large_data/
  Datashader_Revealing the Structure of Genuinely Big Data | SciPy 2016 | James A_Bednar



There's this thing that happens around 300 episodes through where it starts to get hazy on the right way to proceed with the early steps.

It's raising the high end fine, but it's having trouble with keeping the low end up.

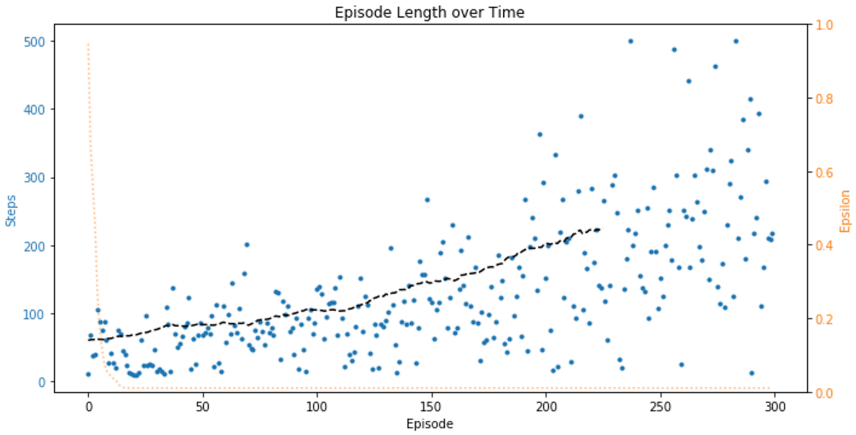
- [x] Take another look at your handling of terminal conditions. I think that I need to **not** calculate future rewards for the terminal condition.

- [ ] One problem I'm having is how to work with larger code, and accidentally not using the latest version of code
    - I want the graphical output of notebooks
    - I want the ability to see clear diffs of what I did that I get from having code in plain files
    - While I can use importlib.reload, it's all too easy to not really get the latest code

    I've taken to using regular code, and sometimes command line, sometimes restarting the kernel

- ★ Lesson learned: in terminal state, it's definitely better to do future_rewards=0 than it is to do future_rewards=-1.

    It makes a clearly visible difference in the learning pattern



Notice the continuous upward trend of the average here, as compared to the tapering in earlier runs. The difference between these runs is

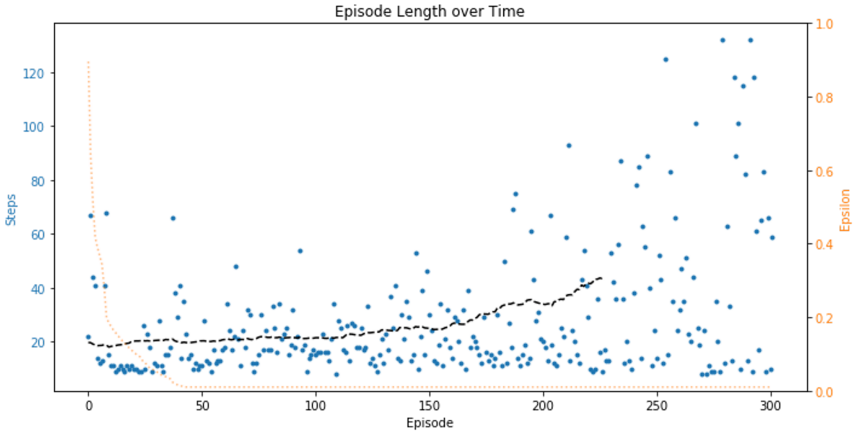
```
If done:
        Future_rewards = 0  # was -1
Else
        …
```

After Each Step Update

- [x] Need to update every single step as you experience it
    - [x] Right now you're only updating in random batches, so it's going to take longer to converge

I changed it to train the net on every single step, in addition to the experience replay batches

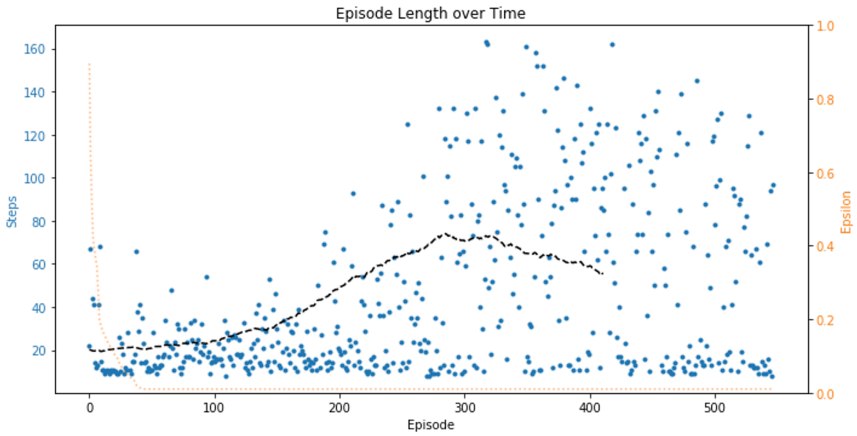
This makes it look like this



Which is definitely worse-- notice that it's hit a peak of only ~120 after 300 steps, as opposed to several ~500 before.

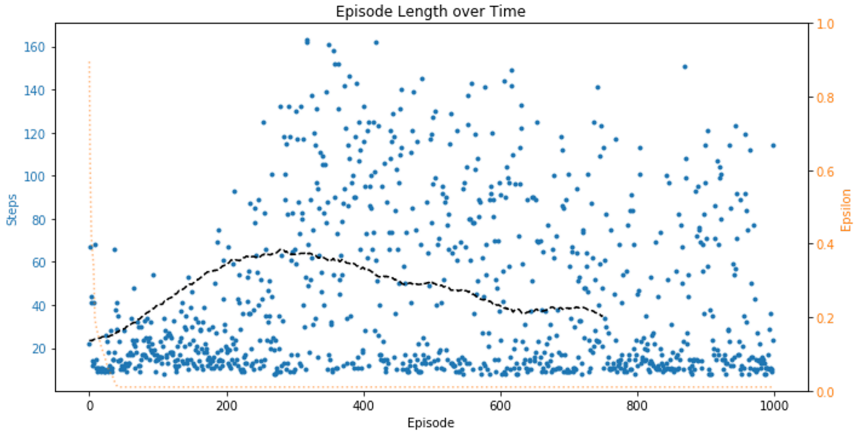
Update 11/25: I theorize that this is because updating every step makes it think too much about states that are very similar to each other, and messes up its ability to generalize across the state space.

But on the other hand, it's got a pretty solid upward trend going. Let's let it go for a while… it also has a lot more low-end dots than the previous version…

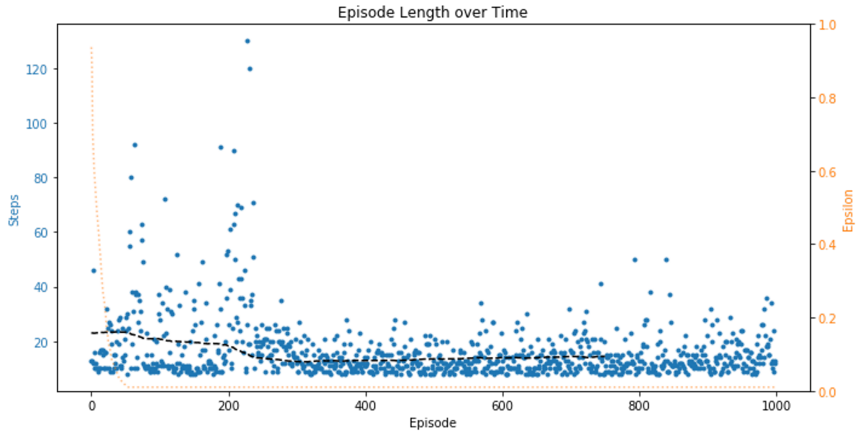


Again it hits a peak at 300 steps and starts to crash. Why?

It's struggling with lots of failure episodes that are under 20 steps. This seems to be a key weakness. But perhaps that's a non-insight: that the problem it's having is it doesn't know what to do in the early phases of an episode. That's tantamount to saying it doesn't know what to do at all. Here's the same training session after 1000 runs:

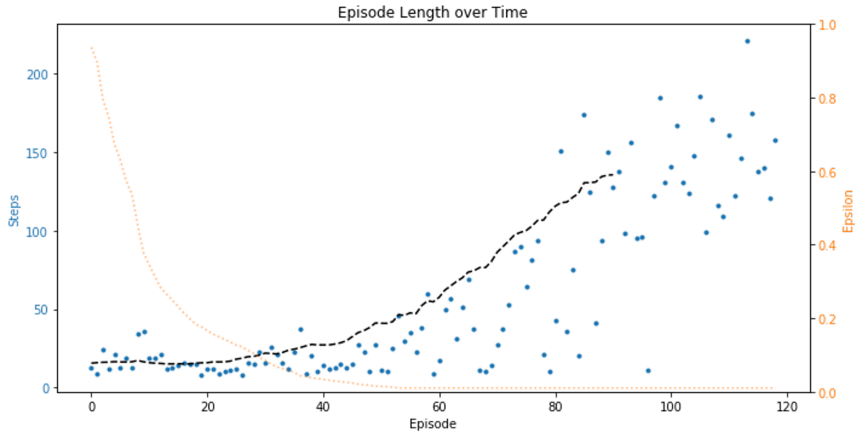


- [x] Now I'm going to remove the after-each-step update, and go back to batch updates only



To riff off David Bowie in "Modern Love": "There's no sign of life. There's just the power to chart". Notice that once again, the collapse happens around episode 300. A little before that this time, plus it never ramped up enough to call that a collapse.

- [ ] Try a reset with the same code but a new random number seed.



This is working much, much better. And I didn't change anything; it's just a different random number seed for the initial network weights.

Is it possible that some starting states are just not solvable? How do I compare to state of the art results?

https://github.com/openai/gym/wiki/Leaderboard

CartPole-v0 defines "solving" as getting average reward of 195.0 over 100 consecutive trials.

- [ ] Look into the writeups others have done
  - [ ] https://github.com/Ben-C-Harris/reinforcement-learning-pole-balance

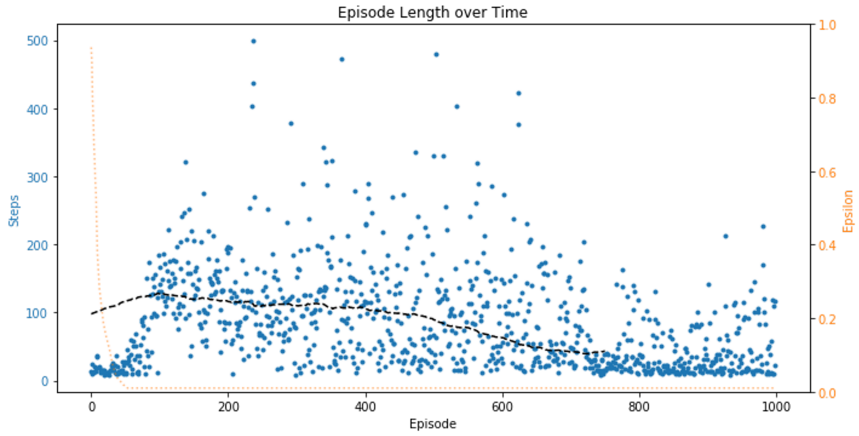
    Differences:

Issue	Theirs	Mine
Batch size	10	20
Model	24, 24, 2 outputs	64, 64, 2 outputs
Q update		Maybe different?
Reward at terminal	-1	0
~~Memory size~~ No this is the same	1,000,000	1,000,000
~~Each step too?~~ No this is the same	No	No
Future reward	Np.amax(…)	Np.max(…) I think this is the same…

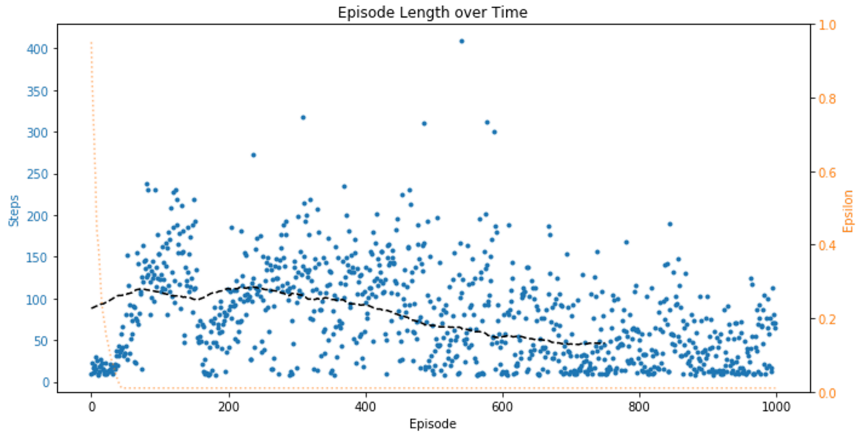
  - [ ] Check into your _learn_step function. It may be different from Ben Harris

- [ ] I've heard that Double Deep Q learning helps deal with this issue where sometimes the model just doesn't converge. Maybe I'll try that at some point.
- [x] Try increasing the memory size beyond 10,000. *Actually it's been at 1,000,000 for a while now.*
- [ ] Quantify just how bad this a randomness problem is. Run several runs up to 1000 and record the results

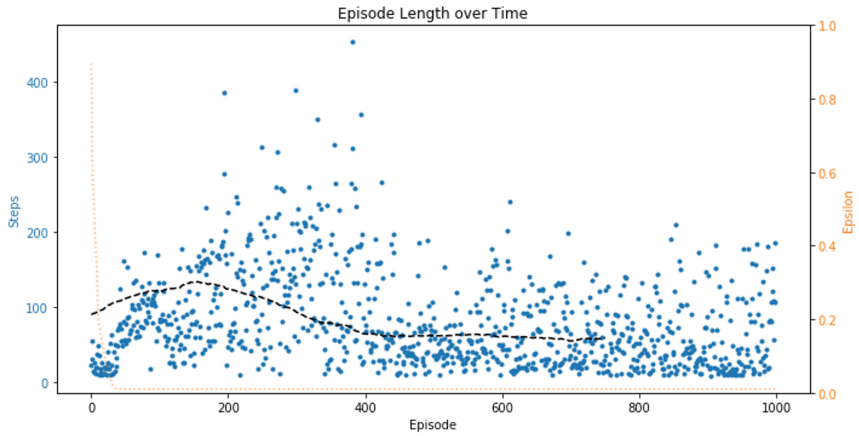
Run 1



Run 2

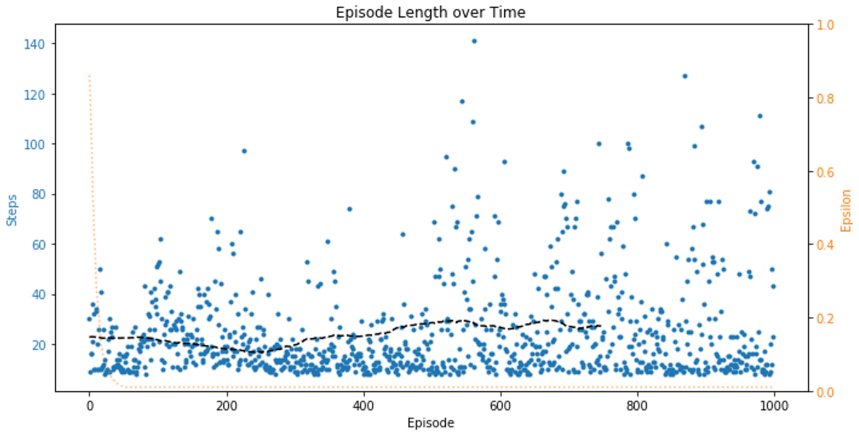


Run 3



Run 4

Try upping setting batch size down to 10 from 20. *Didn't help.*



I've run it 3 times now and haven't been able to replicate the performance I got this morning. Now I'm beginning to wonder about it.

- [x] Compare update steps: is mine the same as Ben's if your algebraically transform it?

Ben

```
Q_update = reward # Updated quality score
If not terminal:
    Q_update = reward + gamma * np.amax(self.model.predict(state_next)[0])
Q_values = self.model.predict(state)
Q_values[0][action] = q_update
Self.model.fit(state, q_values)
```

They look mathematically identical to me.

- [ ] Try model size
- [ ] Try negative reward *on the final state*. Currently I'm doing full reward for final state but zero for future rewards
- [ ] Change your rolling average to work like this

```
mean_step = np.mean(all_steps[-100:])
```

From <https://github.com/JamesUnicomb/ReinforcementLearning/blob/master/DynaQLearning/CartPole.py>

This way you'll get your average number to go all the way to the right side.

A side note 11/24 4:42pm
I was able to run the same carpole_lab deep q learning agent **totally without modification** against this lunar lander environment:

And it immediately started learning… without any code changes at all! Which impressed me because this other environment has **8** state features, and **4** actions, where the cartpole environment has 4 state features and only 2 actions. This kinda *does* feel like intelligence…

11/25/2019
I finally found the problem that's been plaguing my agents for almost 2 weeks!

The correcting PR

9 cartpole_lab/agent.py

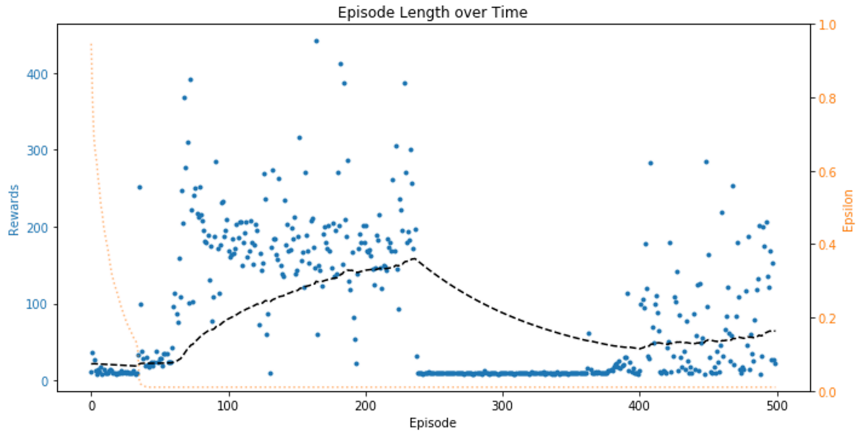
```
@@ -13,12 +13,11 @@ def run_episode(self, render=False):
            state = self.env.reset()
            action = None
            while True:
-               prev_state = state
-               prev_action = action
                action = self.policy.suggest_action(state)
-               state, reward, done, _ = self.env.step(action)
-               episode.append((prev_state, prev_action, reward, state, done))
-               self.policy.step_completed(prev_state, prev_action, reward, state,
done)

                if render:
                    self.env.render()
                if done:
```

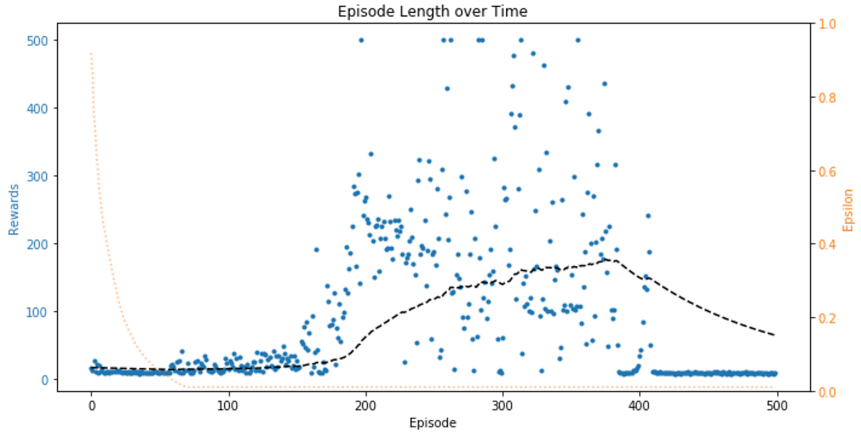
```
            state = self.env.reset()
            action = None
            while True:

                action = self.policy.suggest_action(state)
+               state_next, reward, done, _ = self.env.step(action)
+               episode.append((state, action, reward, state_next, done))
+               self.policy.step_completed(state, action, reward, state_next, done)

+               state = state_next
                if render:
                    self.env.render()
                if done:
```

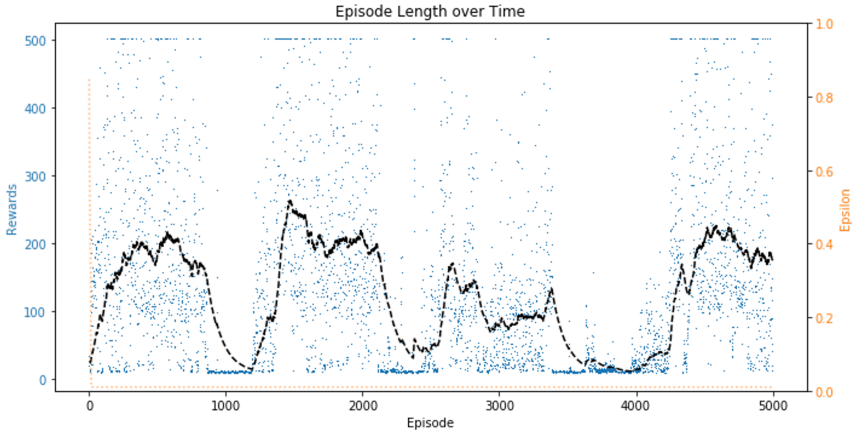
The problem was I was sending update(prev_state, prev_action, reward, state) and the prev_action went with the action before! So I was doing

update(state[t], action[t-1], reward, state[t+1]) # Not what it should be!
Update(state[t], action[t], reward, state[t+1]) # This is what it should have been

Collapse: I've seen this happen a couple of times now-- a complete collapse followed by a recovery

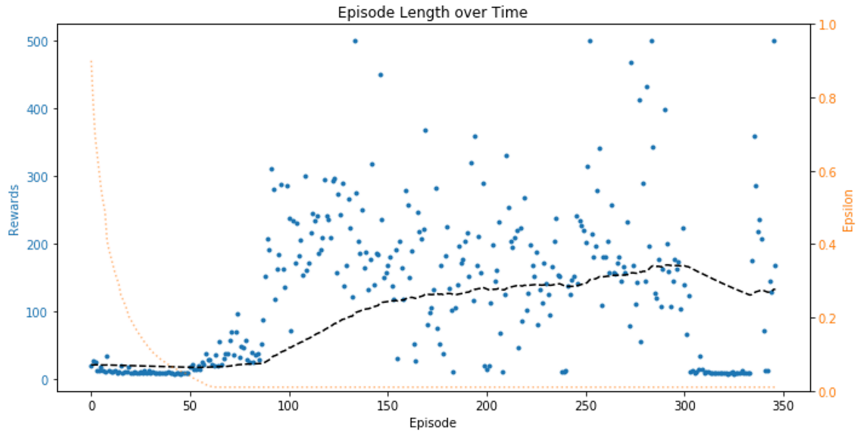






What's with the instability and collapse? I'm guessing that prioritized replay will help, because it's probably forgotten a significant chunk of the policy that keeps it up.

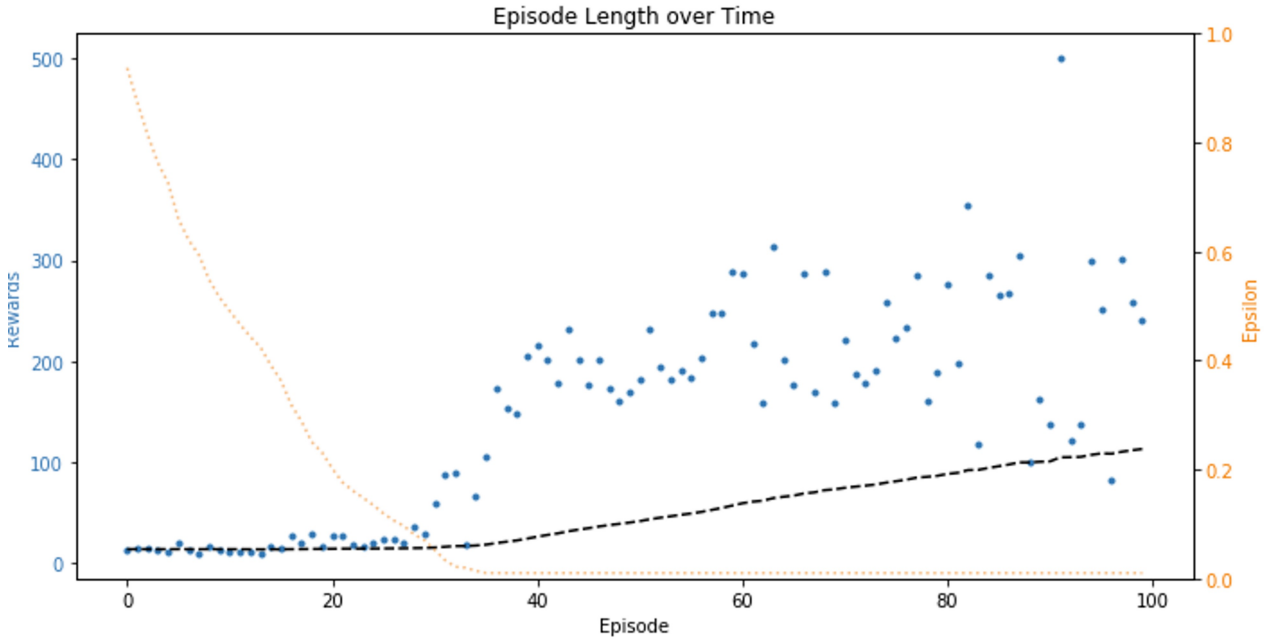
After removing "if done: rewards=-rewards" we still get collapse:



- [ ] See if passing in the step number as part of the state helps? Maybe it can't tell if balancing it in the middle is bad because it sees that as being end of the world so often. Try it again now that you've gotten rid of that negative rewards line.

- [ ] See if prioritized replay helps with this
- [ ] I bet Double Deep Q N would help too. I've heard DDQN helps "stabilize"
- [ ] Actually I think that the time step number needs to be part of the state. Reason: staying hovering forever isn't a good strategy… but then the continuous negative reward stream for fuel tells it the same thing. But in the cartpole environment, timestep could be significant, because if you're at step 480 your expected rewards are only going to be +20 at best.

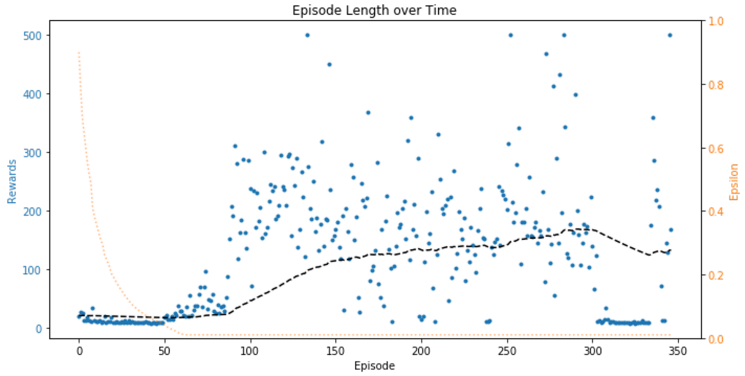
11/26 I implemented Prioritized Experience Replay and it worked!



I'm stoked about this because it was an original idea I had, and then I discovered someone else already had the idea, but the point is I thought about how it wasn't properly absorbing the lessons of its experience, and about how to fix that. Then I implemented the idea and it worked! The first "200" that yesterday's model was hitting was at episode 90, and this one hit that at about episode 40 (compare to below)

Comparison to yesterday.

- Note that yesterday I never hit 200 until episode 90
- More importantly, note that before I was getting below average a lot, but now it never once dropped below average! Except for that little foible around episode 90, but it's seriously doing a better job of absoring lessons! Compare that to below from yesterday, where it's forgetting the basics of how to control itself constantly past turn 150. Then it has an utter collapse at 300-350.



- [ ] Instead of returning the surprises in at near zero weight, try reducing them exponentially
- [ ] See how lunar lander does now.
  - [ ] Be sure to try it with different gamma levels too
- [ ] See if this thing has a collapse when training it for 1000 or 5000 episodes like it did above with the mountains and collapses
- [ ] See what the official version of prioritized experience replay does about sticking things back into the queue. I'm not sure what weight to put on things once I've replayed them once. Maybe what I did is the right thing, where I put it in at a lacklustre weight.