Train a Smartcab How to Drive

Fernando Hernandez

Udacity - Machine Learning Engineer - Project 4

Implement a basic driving agent

Implement the basic driving agent, which processes the following inputs at each time step:

- *Next waypoint location, relative to its current location and heading,*
- *Intersection state (traffic light and presence of cars), and*
- *Current deadline value (time steps remaining),*
  And produces some `random move/action` *(None, 'forward', 'left', 'right'). Don't try to implement the correct strategy! That's exactly what your agent is supposed to learn.*

Run this agent within the simulation environment with enforce_deadline set to False (see run function in `agent.py`*), and observe how it performs. In this mode, the agent is given unlimited time to reach the destination. The current state, action taken by your agent and reward/penalty earned are shown in the simulator.*

In your report, mention what you see in the agent's behavior. Does it eventually make it to the target location?

With random behavior, the agent does eventually reach the target, but very rarely makes it if a deadline is enforced.

Identify and update state

Identify a set of states that you think are appropriate for modeling the driving agent. The main source of state variables are current inputs, but not all of them may be worth representing. Also, you can choose to explicitly define states, or use some combination (vector) of inputs as an implicit state.

At each time step, process the inputs and update the current state. Run it again (and as often as you need) to observe how the reported state changes through the run.

Justify why you picked these set of states, and how they model the agent and its environment.

The states of the system were represented by:

1. The state of the traffic light: ‘green’ or ‘red’
2. The next waypoint direction: A valid direction returned by ‘next_waypoint()’.

With these two variables, we can represent where we are anywhere in the map relative to the target location, and the state of the biggest source of negative scores - running red lights (-1).

If the world were not round (going all the way right teleports us left) and we could access our location (and destination location), then Manhattan distance to destination might be used.

With these representing the states of the system, we should be able to learn to follow the way point direction, while following the current implementation of traffic laws. This should eschew the need to explicitly use deadline in our representation. We should also be able to reasonably cover most instances of state/action pairs enough times within 100 trials (given some stochastic choice encouragement when appropriate.)

- 2 (lights) * 3 (waypoint directions) = 6 different states
- 6 (states) * 4 (actions) = 24 different state/action pairs

Implement Q-Learning

Implement the Q-Learning algorithm by initializing and updating a table/mapping of Q-values at each time step. Now, instead of randomly selecting an action, pick the best action available from the current state based on Q-values, and return that.

Each action generates a corresponding numeric reward or penalty (which may be zero). Your agent should take this into account when updating Q-values. Run it again, and observe the behavior.

What changes do you notice in the agent's behavior?

When Q-Values were initialized to 0, our model had trouble getting past just gobbling up the small rewards for simply making right turns. This is presumably because there was always a positive reward associated since right turns are always legal at red lights for some positive reinforcement. There is currently no state-space representation associated with deadline time left, so as far as the model was concerned, it was happy just collecting guaranteed small rewards ad infinitum.

Rather than try to model each of these possible deadline state values multiple times to learn (and seriously prolong learning) , we can learn to chase the goal state implicitly without the need for an impending timer of doom. (We can also avoid trying to create custom features such as 'deadline < 5', etc.)

To accomplish this, a few conditions were needed in our Q-learning updates.

The sample-based Q-value iteration was based on a running average of samples and is defined as follows:

$$sample = R(s, a, s') + \gamma * max_{a'}\left(Q(s', a')\right)$$

$$Q(s, a) := (1 - \alpha)Q(s, a) + (\alpha)[sample]$$

Through some algebraic manipulation, we can arrive at a more compact:

$$Q(s, a) := Q(s, a) + \alpha[sample - Q(s, a)]$$

Where:

$R(s, a, s')$ is the reward for taking an action a from state s to s'.

$max_{a'}(Q(s', a'))$ is the max Q-value of the actions we can take from the next state s'

α is our learning rate in [0, 1]

γ is our discounted reward rate for future rewards in [0, 1]

This should make some intuitive sense. We want to take our current view of the world $Q(s, a)$ and update it with the difference between our one-time $sample$ and our current view.

But since it's just one sample, we should presumably already have a pretty good view of the world if are acting optimally, we want to discount this new sample's difference by α before updating our current view.

Our critical parameters used were:

1. α - Our learning rate is set to around 0.1 We want to slowly incorporate our new knowledge into our view of the world (our Q-values) without discounting past experiences too heavily.
2. γ - Our discount factor for future rewards is set to around 0.9. In much the same vein, we want our past values to remain fairly vivid, and phase out slowly as we gain knowledge about the way the world works.
3. Optimistic Initial Conditions (Sutton & Barton, 2005) - These values were set to around 10 (at or near our reward for reaching the target destination) and turned out to be critical for learning quickly.

α and γ are kind of two sides of the same supporting our Optimistic Initial Conditions (OIC). The OIC turned out to be useful for overcoming our initial problem of getting stuck chasing the first tiny rewards seen (right turn loops).

With higher OIC, we are encouraged to explore in the beginning since no action's rewards can be as good as the initial Q-values. Then once we start to chase the right actions, our high scores are reinforced and strengthened, while chasing the wrong actions cause those Q-values to continually drop.

As this happens, we start to quickly favor chasing the right actions and the Q-values should, in the limit, converge on the optimal values. But in the meantime, we can work out an

optimal policy long before this convergence in optimal values.

With Q-learning (with OIC) implemented, the agent learned a fair bit in 100 trials. Once we changed to Initial Optimistic Conditions, the agent learned some optimal behaviors within 1-2 trials.

Quick Learning

For instance, upon observing the Q-value table after 100 trials, the agent quickly learned to follow the waypoint on all green lights, and if the light was red, the agent learned to follow the waypoint if it pointed right.

Slower learning

But if the light was red and not the 'right' direction waypoint, the agent (more slowly) learned to either:

- Make right turns anyways (for small safe reward that is always positive) then immediately start following the waypoint direction again.
- Wait for the light to change, then continue following the waypoint straight or left

---

Enhance the driving agent

Apply the reinforcement learning techniques you have learnt, and tweak the parameters (e.g. learning rate, discount factor, action selection method, etc.), to improve the performance of your agent. Your goal is to get it to a point so that within 100 trials, the agent is able to learn a feasible policy - i.e. reach the destination within the allotted time, with net reward remaining positive.

Report what changes you made to your basic implementation of Q-Learning to achieve the final version of the agent. How well does it perform?

As discussed earlier, adjusting the learning rate to 0.1, discount factor to 0.9, and Initial

Optimistic Q-values to 10 allowed our agent to explore at first, and learn quickly.

In our post-mortem, it turns out that our agent learned to mimic the 'next waypoint' action sending it into the rewarding target area. This turned out to not be surprising since, once inspected, our agent is rewarded with +2 for following the 'next waypoint' and only +0.5 otherwise (unless waiting, +1).

Incorporating Dyna-Q Learning

Richard Sutton's Dyna-Q learning saves the most recent [state, action, next state, reward] $< s, a, s', r >$transitions that were observed. (Sutton & Barto 2005) & (Kaelbling, Littman, & Moore, 1996)

These transitions can then be randomly sampled N times after each $Q(s, a)$to update our belief of the world. This essentially let's us replay our memories of the world in simulated experience planning before acting on the world again.

In fact, Google DeepMind has had much success recently building upon these same ideas in deep learning implementations with general purpose agents that learned to play Atari (Mnih, et al, 2013) and even beat professional Go players (Silver, et al)!

Implementation

We only add two steps after our $Q(s, a)$ update. We store the observed $< s, a, s', r >$ transition in a ***replay memory***, then randomly sample N times out of it, updating Q(s,a) as usual with these samples.

1. $sample = R(s, a, s') + \gamma * max_{a'}(Q(s', a'))$
2. $Q(s, a) := Q(s, a) + \alpha[sample - Q(s, a)]$
    1. Replay_memory.insert($< s, a,' s, r >$)
    2. Repeat N times:
        - $\tilde{s}, \tilde{a}, \tilde{s'}, \tilde{r}$ = sample(Replay_memory)
        - $sample^{random} = \tilde{r} + \gamma * max_{\tilde{a'}}(Q(\tilde{s'}, \tilde{a'}))$
        - $Q(\tilde{s}, \tilde{a}) := Q(\tilde{s}, \tilde{a}) + \alpha[sample^{random} - Q(\tilde{s'}, \tilde{a'})]$

Whew! So, what do all of these extra steps lead to? More thinking (computation) but less actions required on the environment before learning optimal strategies!

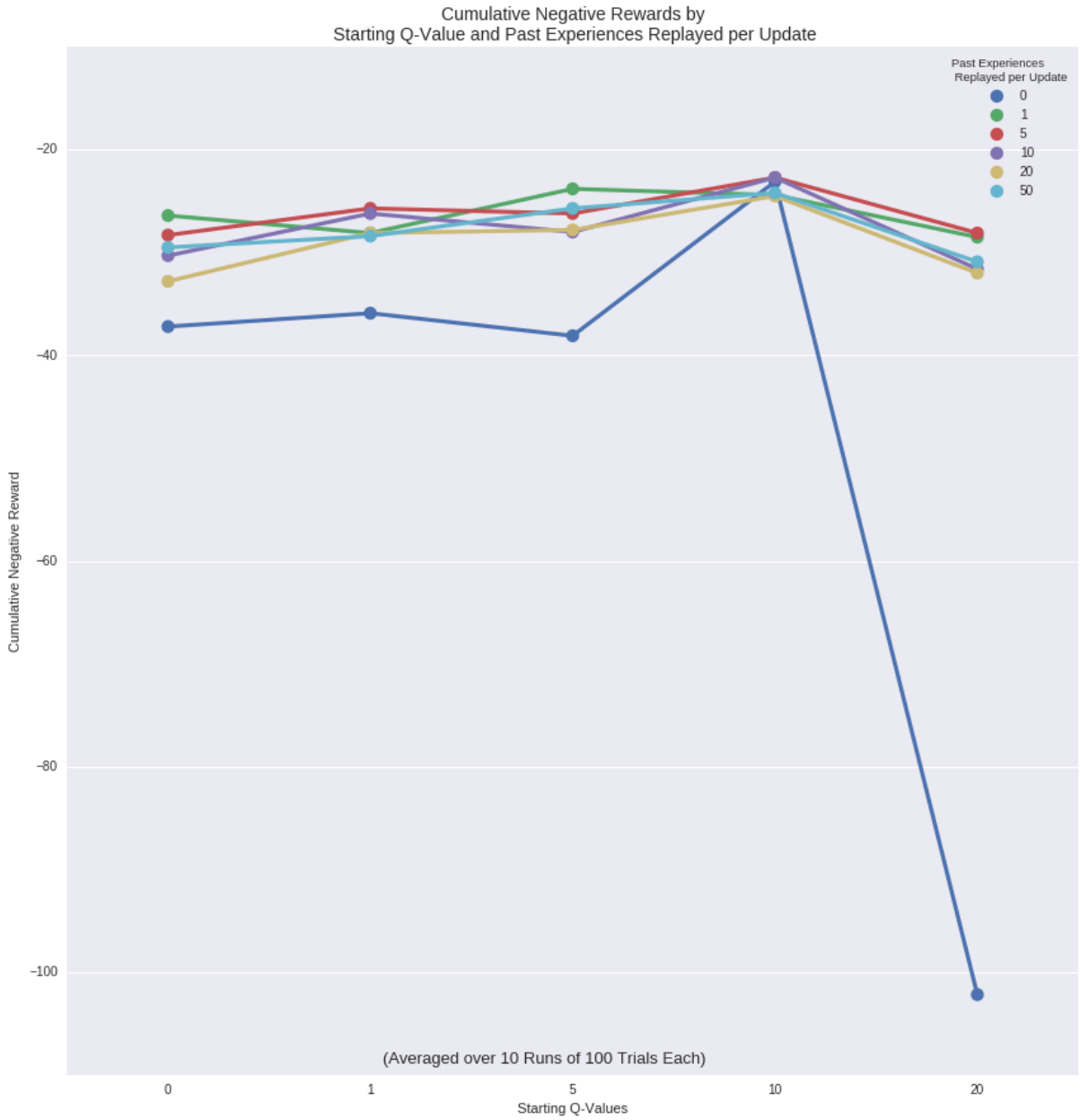
Analysis

Rather than just talk about the potential benefits (and peculiarities!) of this method on our particular problem, let's run a bunch of simulations and plot the agent's average behaviors

over different settings of hyperparameters!

The models were run 10 times at 100 trials of learning each time under different settings of our implemented changes, ***optimistic initial conditions*** and ***number of memory episodes*** to use in experience replay/planning.

First, let's look at the average negative rewards accumulated during each run.

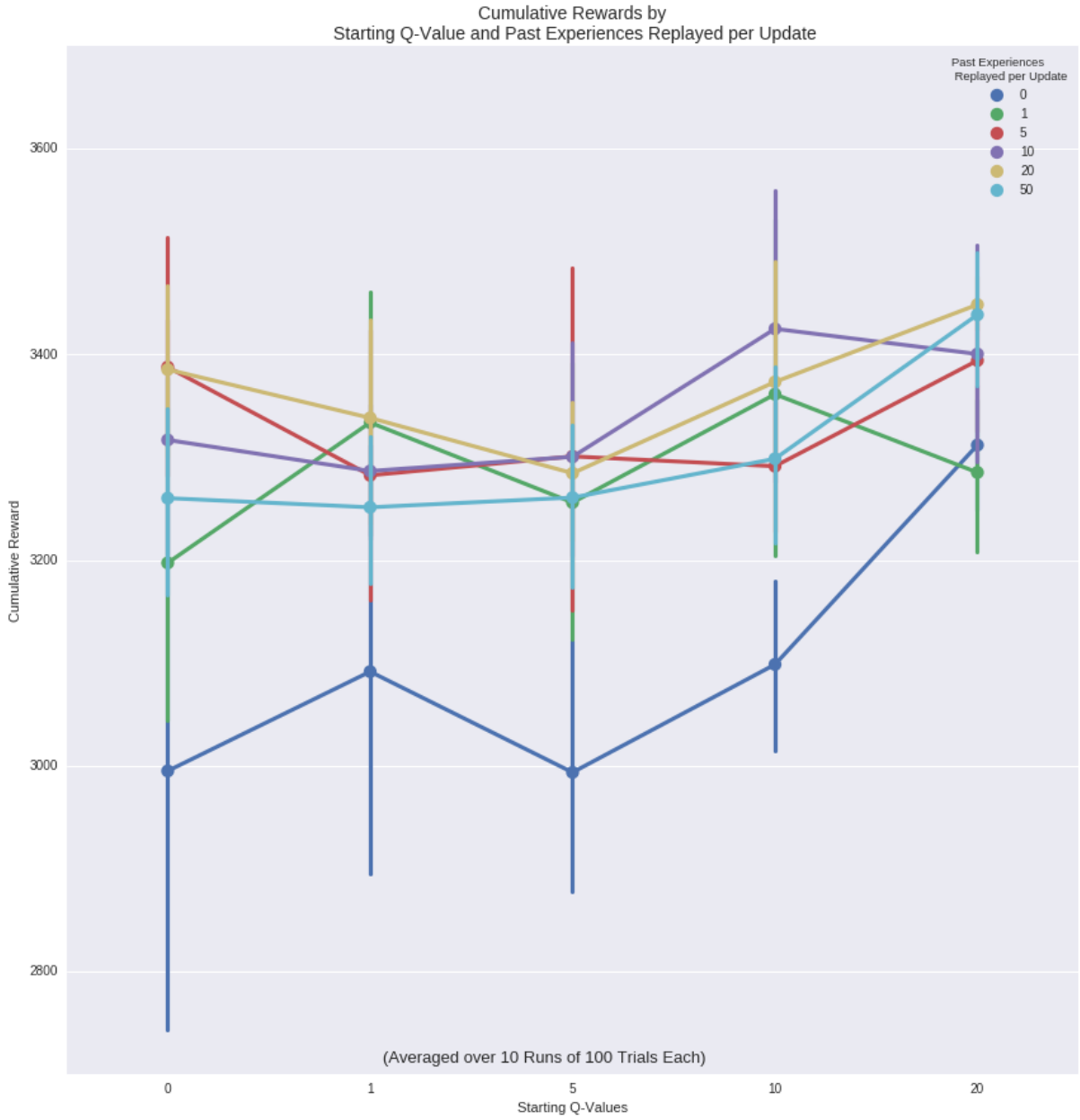


In general, we are penalized by -1 for trying to run a red light, or trying to turn left at a red light.

Here we can see that when we don't replay any memories, we generally make more

mistakes than when replaying memories. The one difference is when we have *optimistic initial Q-values* (OIC) all set to 10 at the onset. The agent seems to learn to quickly avoid mistakes here, while a higher value of 20 causes our agent to go on a reckless rampage since any random action is better than reaching the goal and getting a measly +10 reward!

Next, we'll look at average total accumulated rewards over each 100 trials of learning from scratch.



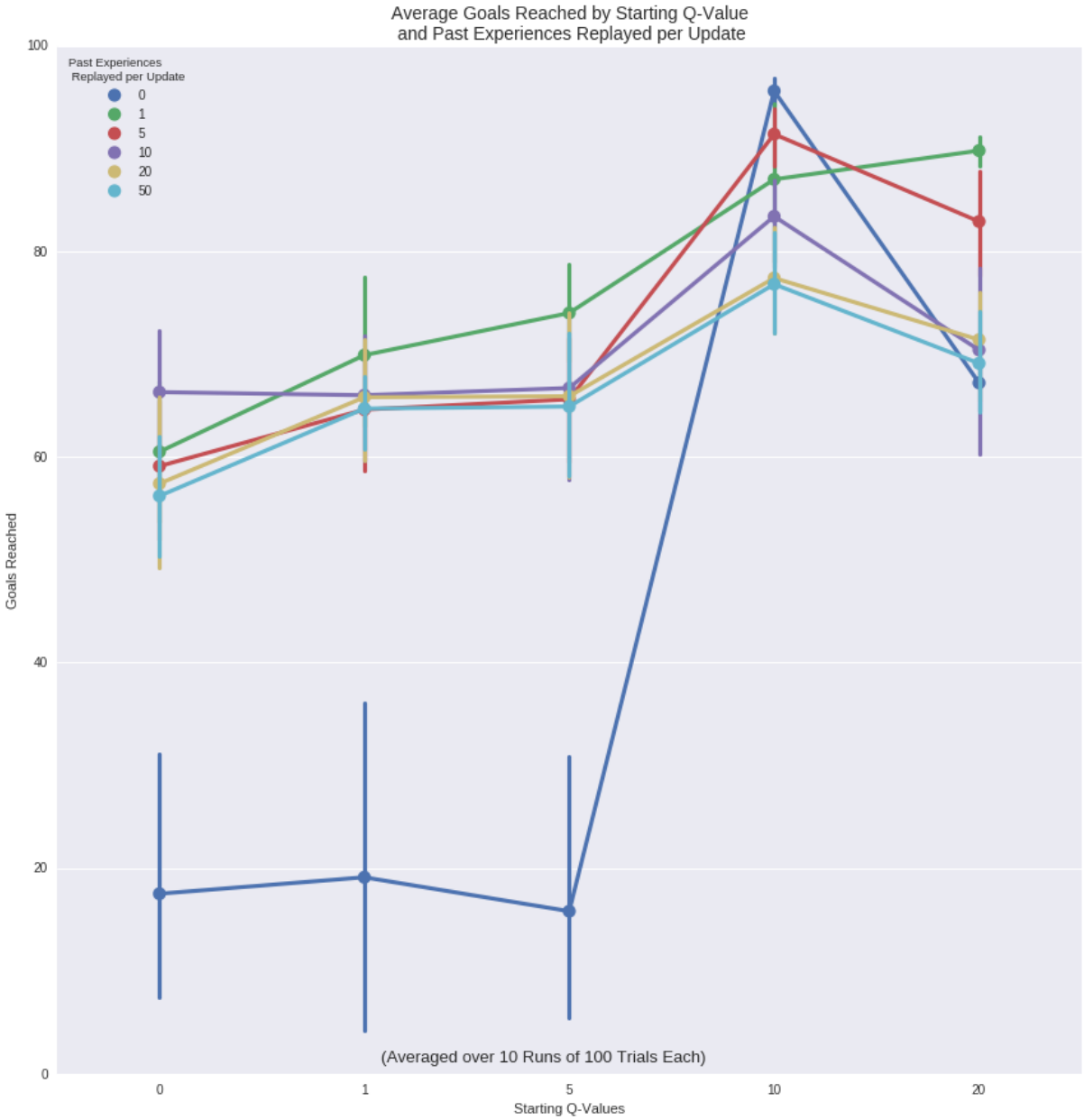
No experience replay definitely seems to accrue less rewards on average for nearly all starting Q-values. Of note are the confidence intervals (based on 100 bootstrap samples); at smaller initial Q-values and no experience replay sometimes our agent can score fairly

high, or sometimes abysmally low.

This makes sense because it's just running around trying to figure out the world and not incorporating knowledge very quickly. So sometimes our agent can luck upon the goal and start learning, while other times our agent just wanders around without randomly stumbling into much positive feedback.

As we up the OIC values, our agent is encouraged to try to new actions rather than just random actions more frequently in the beginning, and finds those that reinforce those higher rewards more often (but still not as often as sitting and pondering a bit about past actions first.)

Next we'll look at the other major goal of our agent, reaching the goal in time.



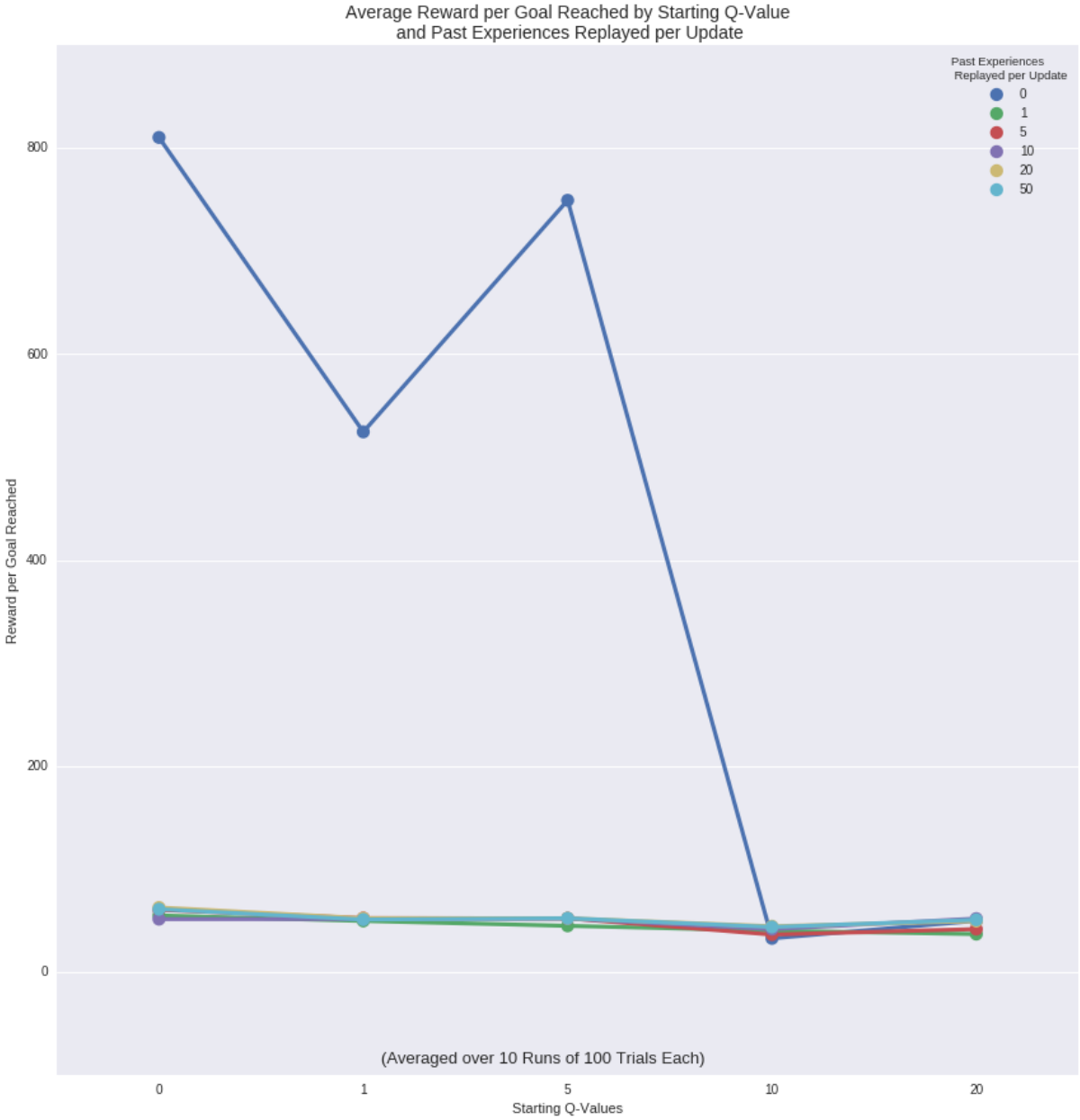
Here we can see replaying experiences pays big dividends regardless of OIC! The big difference is at OIC of 10, here there seems to be just the right about of initial exploration and relative positive feedback learn to get to the goal in time!

The one big contradiction here is our 10-OIC, 0-replay agent. It fairs the best on average getting to the goal, but does the worst with the same parameter on previous cumulative score!

Maybe the replay learning has learned something different to balance total reward maximization and goal attainment? Let's press on.

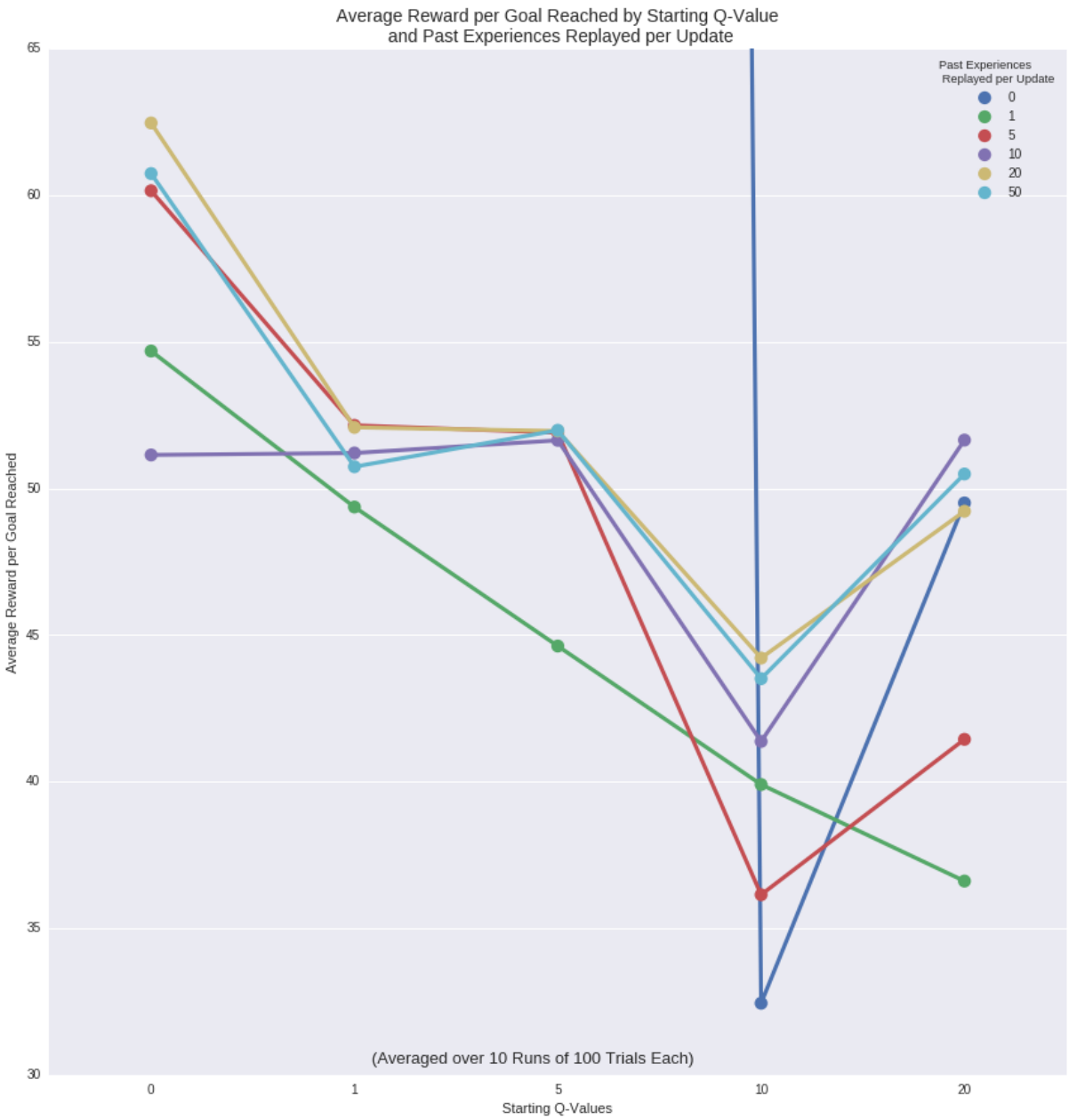
To try to address this conundrum, let's take a look at the average cumulative reward

earned per goal reached.



While this may not seem very illuminating at first, it does show that the agent that doesn't learn much at lower initial OIC still earns a fair bit of cumulative positive reward for just driving around not reaching the goals making the goal/reward ratio huge!

But let's zoom in a bit to look at the more interesting section of agents who reach the goals often.



Here we see that, at the general optimal OIC of 10, our agent with 0 replay has the lowest average reward per goal!

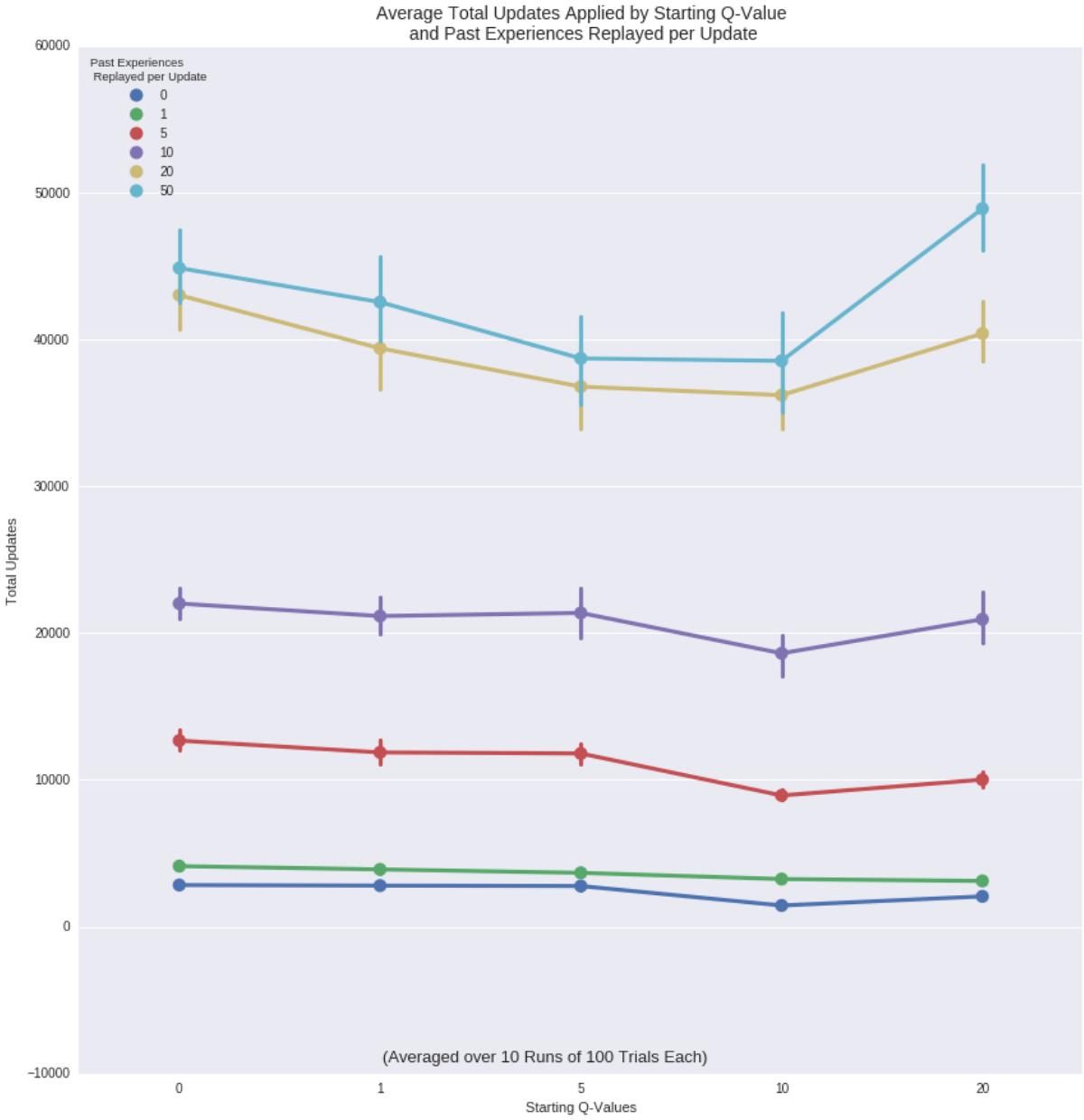
Again, maybe the replay learning is balancing total reward maximization and goal attainment.

Perhaps it could be if we have a goal that is fairly close, it may be more advantageous reward-wise in the long-run to drive around a bit in the right direction while obeying the law and collecting +1's/+2's than to just get there quickly for +10. (Hmm..)

The goal distances are also randomly chosen at each of the 100 trials per run, so maybe we are also at the mercy of stochastic processes and a sample size of only 10 runs per average statistic.

Finally, let's see how many updates (computation) these different approaches are costing.

—



Experience replay planning does come at a cost, typically at about the ***number of replay experiences*** times more updates/computations. But for our problem, these computation differences are negligible.

For a larger problem, it Dyna-Q/experience replay is probably still worth it due to their faster convergence rates in general with less interactions required on the environment.

So which is best?

Optmisitic initial conditions shows to be ideal at 10 for this particular problem. This is almost certainly because it creates exploration at the onset, but quickly reinforces optimal actions and drops sub-optimal actions below 10 quickly making them less appealing.

For ***5 replay experience episodes*** it's hard to say, but I'd put my money on a value of 5, even though the computational cost is higher.

The 0-replay-experience agent surprisingly comes in as a close second choice.

This is because the at these settings, our agent attains:

- Much higher average cumulative reward than 0-replay-agent
- 2nd highest average goals attained (1st being 0-replay/ 10 OIC)
- Higher average reward per goal attained than 0-replay
- Tied for least overall average mistakes made during learning.
- As the environmental search space and complexity grow, experience replay agents would be able to fair much better with disseminating knowledge.

Again, the 0-replay-agent fairing so well (at 10 OIC) was fairly surprising!

This could be due to:

- Relatively small search space of state/action pairs in this problem not being as important when OIC is used
- Reward system not emphasizing goal attainment too heavily in comparison to cruising in the general direction while obeying laws
- Low sample size of runs.

Does your agent get close to finding an optimal policy, i.e. reach the destination in the minimum possible time, and not incur any penalties?

At an OIC of 10, our optimal agents do seem to find the optimal policy very quickly, rarely not making it to the destination. Our agents are also learning that sub-optimal actions Q-values drop below the OIC of 10 quickly, while optimal actions add to the initial OIC. This has the added benefit of our agent learning to avoid incurring penalties quickly.

Our Dyna-Q/experience replay agents also seem to incorporate reward maximization while still reaching the destination in time while incurring minimal penalties.

Finally, our agent seems to learn an optimal ***policy***/actions much more quickly than converging to the actual optimal Q-values, which is a good thing!

Conclusion

This has been a very interesting project. Implementing Q-learning, and Dyna-Q learning really gave me a much more solid grasp on what's going on during reinforcement learning, which can be neglected in a lot of applied machine learning courses. I am also able to now read and understand much better related research like Google Deepmind's Atari learning agent (Mnih, et al, 2013).

Our current Dyna-Q agent might benefit from priority search such as prioritized sweeping instead of simple random samples, as well as possible e random-action annealing to further minimize mistakes after intial learning.

I would also be interested in implementing approximate Q-learning using algorithms with Loss functions and temporal-difference methods to approximate Q-values. Then we can start plugging in linear models and neural networks! Fun stuff!