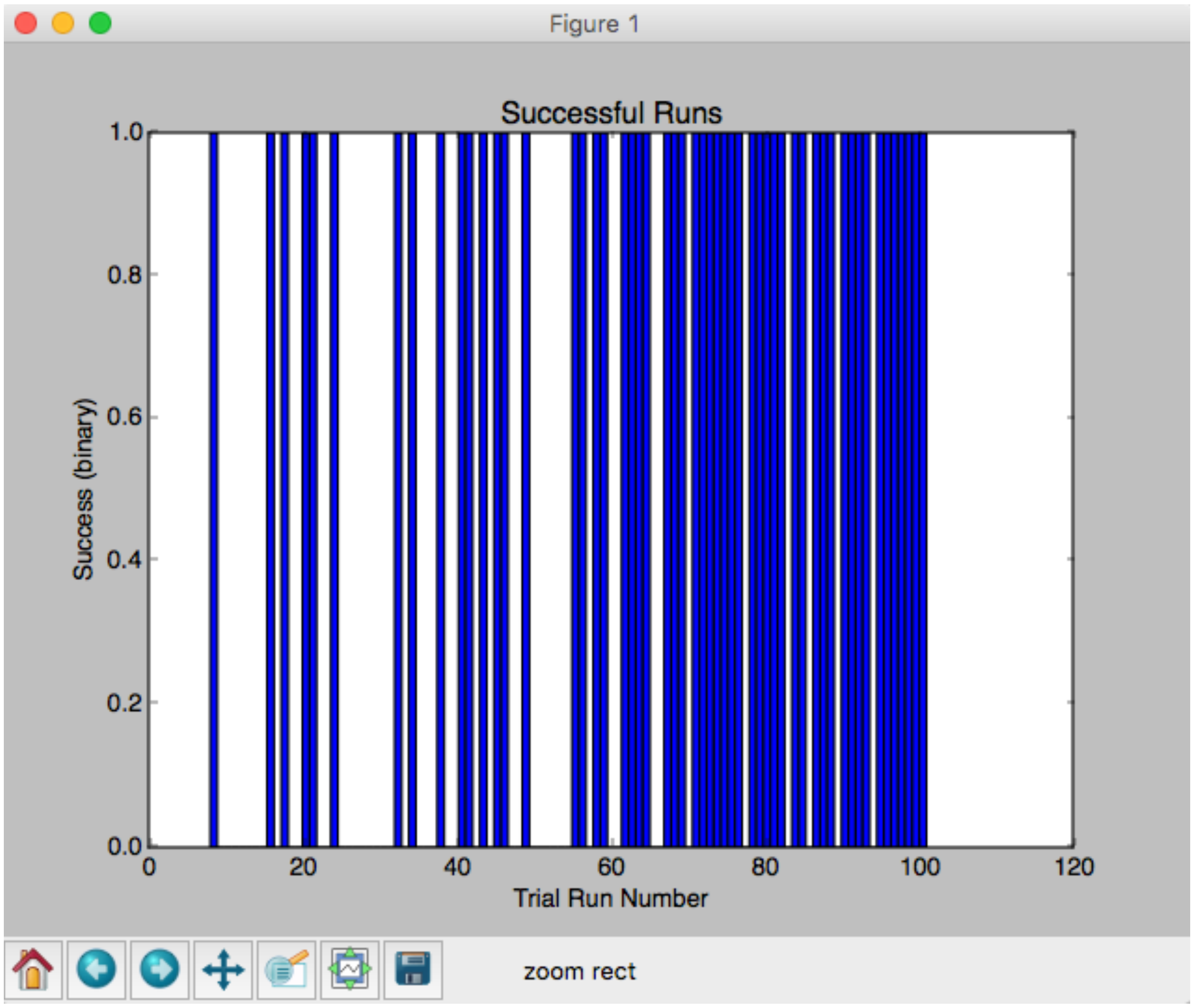
Choosing the State:
After testing different combinations of states, I boiled down the factors into two essential parts: the next waypoint (in order to reach the destination) and the state of the traffic light at the current intersection. If the agent were to receive a reward, it should know that a positive or negative reward was based on whether it followed the next waypoint and the action it followed during the state of the traffic light. I also wanted to include whether it sensed the presence of other cars in the intersection, but the former two state factors were enough.

Choosing the Action:
An action was chosen at random epsilon percent of the time. (1 - epsilon) percent of the time the action was determined based on the current state's q-value.

Changes in Behavior:
These might not technically be the reported changes in behavior, but these are the changes I've noticed on macro scale.
I included some analyses to observe the car's progress of being able to successfully reaching its destination within the time limit. Here are the desired results:

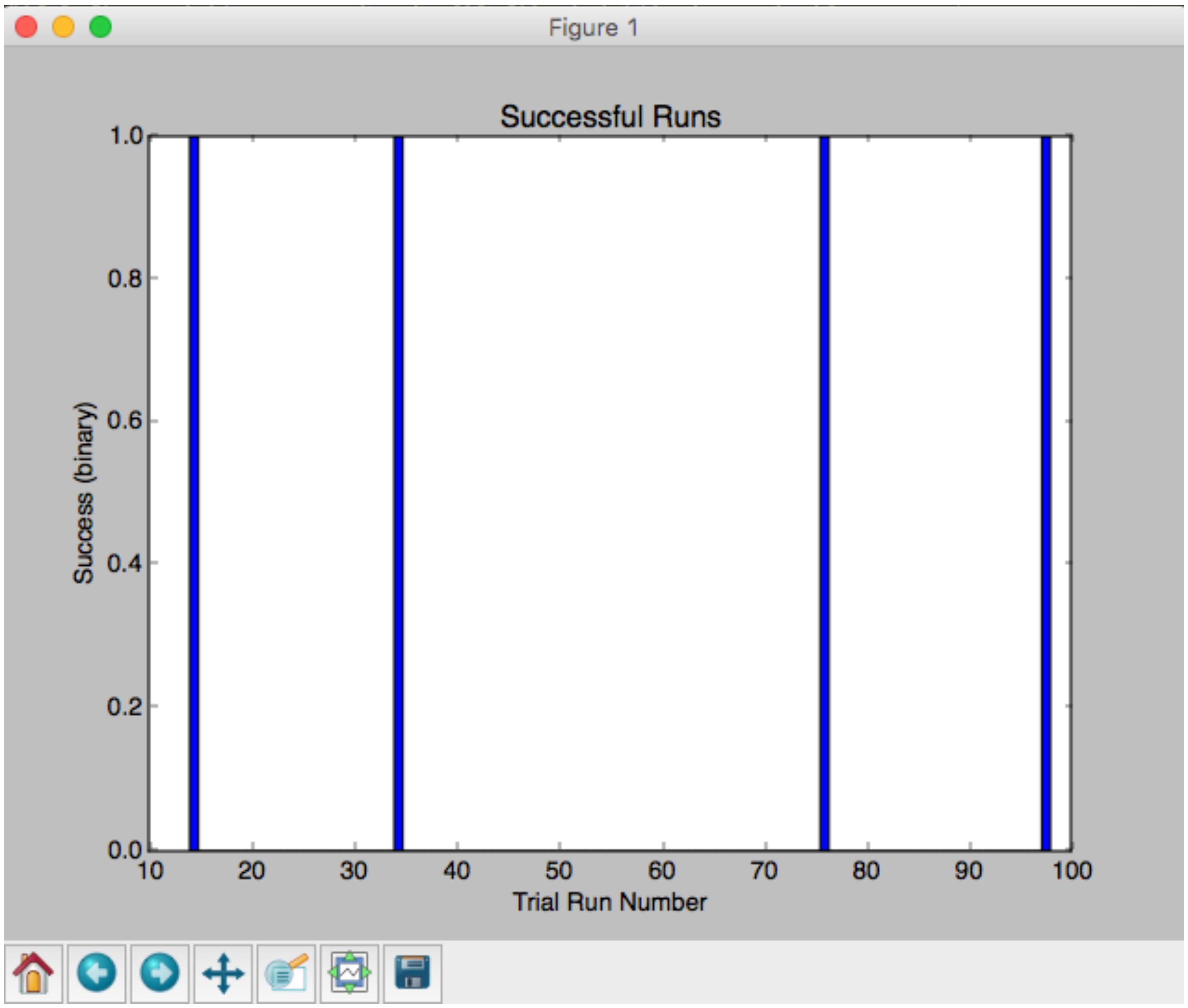


List of trial numbers, where the agent successfully reached the destination:
[8, 16, 18, 20, 21, 24, 32, 34, 38, 41, 42, 43, 45, 46, 49, 55, 56, 58, 59, 62, 63, 64, 65, 67, 68, 69, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 84, 85, 87, 88, 89, 90, 91, 92, 93, 95, 96, 97, 98, 99, 100, 101]

Number of successful runs:
54

In the figure above, the blue bars indicate each time that the agent had successfully reached its destination within the time limit. At first, it rarely reached its destination. By trial number 60, it is able to consistently reach its destination with a high success rate. By trial number 90, it reaches its destination 90% of the time. In total, it reached the destination 54 times.

Bad Trial: (not required for report but thought it was necessary)
Once in a while, the smartcar doesn't learn well.



List of net rewards per trial run: # all still positive
[0, 13.0, 6.5, 26.0, 18.5, 14.5, 10.5, 21.0, 22.5, 18.5, 19.0, 37.0, 26.0, 20.5, 21.0, 21.0, 19.5, 14.0, 25.5, 43.0, 18.5, 30.0, 18.5, 23.0, 24.5, 22.5, 33.0, 27.0, 21.0, 17, 29.0, 29.5, 40.5, 40.0, 34.0, 30.0, 26.5, 47.0, 21.0, 21.5, 20.0, 30.5, 20.5, 36.0, 31.5, 20.0, 44, 16.5, 29.5, 19, 30.5, 19, 32.0, 26.0, 25.5, 24.5, 21, 25.5, 26, 21.0, 30.5, 21.5, 25.0, 30.5, 36, 25.5, 41.5, 16.0, 36, 40.5, 20.5, 21, 36.5, 29.5, 52.0, 10.5, 26, 25.5, 31, 19.5, 36, 20.5, 47.5, 31, 28.5, 31, 35.0, 20.0, 21, 40.0, 27, 20.5, 21, 39, 27.5, 31, 19, 23, 21.0, 30.5]

List of trial numbers, where the agent successfully reached the destination:
[14, 34, 76, 98]

Number of successful runs:
4

```
for state, q_value in q_table.iteritems():
    print state
    print q_value
>>>
('forward', 'red') # when the light is red, the highest q-value is None
{'forward': 1.1466596419156958, 'right': 1.1914127540437718, None: 1.2191785778893167,
'left': 1.1946234442209993}
('right', 'green') # when the light is green, the highest q-value is left, when it should be going right
{'forward': 0.9598251821415558, 'right': 0.9414288777917754, None: 1.0034001080812558,
'left': 0.9816484486098959}
('right', 'red') # when the light is red, it should be taking a right
{'forward': 0.8006064771417455, 'right': 0.8425441070131333, None: 0.9827737089995868,
'left': 0.8434968030262442}
('left', 'green') # when the light is green it turns right rather than left
{'forward': 2.0775710667589298, 'right': 2.093214389091346, None: 2.090274967037783, 'left':
2.0637407276062945}
('left', 'red') # when the light is red it turns left rather than waiting at the light or turning right
{'forward': 1.730464360998142, 'right': 1.7426226679818924, None: 1.7353246677345884,
'left': 1.7464998957446924}
('forward', 'green') # when the light is green, it goes forward, as it should
{'forward': 1.8025492770255966, 'right': 1.7246577639518421, None: 1.7225693952756436,
'left': 1.7244592902812288}
```

I hypothesize that the reason for this is that the agent must have been negatively rewarded early in the trials. It may have been “traumatized” early on by, for example, turning left at a green light, when an oncoming car was going forward, resulting in a negative reward. The presence of cars isn’t factored into the policy.

Agent Learns Feasible Policy:
Going back to the first (ideal) example, here are the net rewards per trial run:

List of net rewards per trial run: # all positive
[0, 18.0, 19.0, 7.0, 24.0, 13.5, 6.0, 17.0, 6.5, 31.5, 22.5, 20.5, 19.5, 13.0, 21.5, 25.5, 19.0, 37.0, 38.5, 56.0, 43.5, 23.5, 24.0, 46.0, 37.5, 31.0, 34.0, 23.0, 31.5, 27.5, 17.5, 18.0, 18.5, 23.5, 24.0, 26.0, 38.5, 43.5, 19.0, 23.5, 21.5, 32.0, 24.5, 34.5, 17.5, 11.0, 21.0, 20.5, 29.5, 38.5, 36.0, 20.0, 34.0, 21.0, 33.0, 29, 44.5, 30.5, 30, 33.0, 59.5, 30.5, 36, 15.0, 30.5, 56.0, 41.0, 46, 31, 26.5, 31, 26, 36.5, 30, 24, 31, 22, 22.5, 35, 26, 35.5, 35, 46.0, 24, 38.5, 41.0, 22, 30, 40, 24, 35, 20, 23, 43.5, 30.5, 38.0, 18, 27, 38, 10.5]

Improvements Reported:
I tuned three parameters: alpha (learning rate), gamma (discount rate), and epsilon (randomness). Alpha and epsilon were both initially set to 1. They decayed at the same rate also. At each trial, they were decayed at a rate of 1 over the trial number (1 / trial). I set the variable, gamma = 0.7 ** trial. The 0.7 was chosen after testing gamma at 0.9, 0.8, and 0.7. The decay rate equations for alpha and epsilon I learned from the lectures. The equation for gamma was also learned from the lectures.

Final Performance:
Here is the optimal policy for each of the six states:

```
for state, q_values in q_table.iteritems():
    print state
    print q_values
>>>
('forward', ‘red') # 1
{'forward': 0.916403088077901, 'right': 0.9097482777530781, None: 1.8647011985087083,
'left': 0.9219947419002325}
('right', 'green') # 2
{'forward': 1.4857296178104615, 'right': 1.2443045890748672, None: 1.2436912189536018,
'left': 1.2725610574244341}
('right', 'red') # 3
{'forward': 0.9668227717651976, 'right': 1.6226575258038178, None: 0.9450434347890134,
'left': 0.9596962384734086}
('left', 'green') # 4
{'forward': 1.7348453909923336, 'right': 1.743091304005982, None: 1.7563510086151892,
'left': 2.0727582044612527}
('left', 'red') # 5
{'forward': 0.9189742412832963, 'right': 1.200381330402584, None: 0.9151762059763113,
'left': 0.9077261550719182}
('forward', 'green') # 6
{'forward': 2.358558612352207, 'right': 1.534619239448997, None: 1.5176265388727608, 'left':
1.5208372526409422}
```

For the first state, its intuitive to take no action when the light is red and the next waypoint is forward. The highest q-value supports this.
For the second state, the immediate reasonable action would be to turn right when the light is green and the next waypoint is right. However, the policy shows that the optimal action to take in that state in order to maximize long term reward is to turn left instead.

In state three, when the next waypoint is right and the light is red, the optimal action is to turn right.
In state four, the optimal policy aligns with the next waypoint.
In state five, the optimal policy says to turn right when the light is red and the next waypoint is left.
In state six, the agent should go forward when the light is green and the next waypoint is forward.
The highest q-values in each state is the action that that the agent should follow according to the policy.

The q-hat values are calculated in code:

```
max_q = max(self.q_table[state].values())
v = self.q_table[self.previous_state][self.previous_action]
x = reward + gamma * max_q
q_hat = ((1 - alpha) * v) + (alpha * x)
```