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

Answer: The agent is just aimlessly wandering the grid city.

(1) It does not comply with the traffic rule.

(2) It can not always reach the destination within deadline, though sometimes it did.

(3) Ignoring the reward.

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

Answer: The primary agent driving in the city grid, at any specific time step, it must be at one of 48 intersections. So it's states should be a combination of these factors:

* time

* location

* next intersection

At every crossing, the agent should 'sense' the environment:

How much time do I have?

Is the light red or green?

Is there any other car around at the same place?

If there are, what are their moves?

* What's the next intersection? (added latter)

I use a tuple to hold all these information and construct my agent's state at first:

state = (time, light, oncoming, left, right)

The rational of the state factors:

* time: Generally speaking, time always has something to do with the agent, at different time step, the agent may or may not in different state. At first, it does in my agent's states, but this way make the states space too large, it's not efficient for the agent to learn, there are too many states. More importantly, we don't want the agent get reckless with the deadline gets closer, after all, traffic rules must always come first and we want to make sure the agent learns that.

* location: Location here really means the traffic, that is: light, oncoming, left, right, it's pretty obvious.

* next waypoint: I fought for this one a long time! At the very first, I have thought hard whether I should put it in the state tuple. I decided to not add it in finally, for the sake of a more lean set of states. I have read many posts in the course's support forum, people are all talking about avoiding the too big states space. And honestly, I'm not so sure about it then. I did think about the heading, and when I read the source code of the planner I found there is a 'heading' in the environment which keeps track all the agents. Actually, I was a little confused that moment. All in all, my first time state is:

state=(light, oncoming, left, right)

I use this one about 4 days to run the simulation. During that time, I went over the lecture videos again, read all kinds of materials in and out the forum, and google of course. But the agent performance was not getting much better. Then I asked myself a question, the agent must show preference over the direction that the route planner picked. If it's state doesn't have a heading, how is it gonna know that? Then here comes my final state:

state = (light, oncoming, left, right, next_waypoint)

After I changed the content of my agent's state, the performance showed a big jump. Below is the comparison:

	Before (light, oncoming, left, right)	After (light, oncoming, left, right, next)
Successful delivery	13	88
	26	85
	8	88
	24	89
	22	89
	19	87
	15	98

3. *What changes do you notice in the agent’s behavior?*

Answer: During several early trials, the agent went somewhat randomly, because the policy was not quite learned yet. With time goes by, it catches up pretty quickly, although it sometimes get negative reward as penalty, that's because the 'exploring' setting. Part of the agent's behavior should be 'exploring' and the other part 'exploiting', that is choosing action by Q-value. I read a post in the forum that one of udacity's mentors said how often the agent should 'exploring' and 'exploiting' is still an open research field, and I set 80% of time the agent choose action by Q-value, other 20% act randomly. No specific reason, it just worked.

After around 20 trials or so, the agent can generally move toward the destination, comply with the traffic rules most of the time, and almost always pick the action that yields the biggest reward. It sometimes doesn't choose action according to the Q-table, because there is 0.2 probability to randomly choose an action, besides this learning behavior, the agent can also use it to break cycle routines.

4. *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?*

Answer: The first time Q-learning implementation, my parameters are: (written on July. 1th)

alpha, gamma = 0.1, 0.1

All the initial Q-values are uniformly distributed between 0 and 0.1, and when in simulation, the agent sit still most of the time. My speculations are:

* alpha, gamma too small

alpha is the learning rate, smaller alpha means we incline to not update the Q-value too much at each time step, which I should not use at the beginning. I do want the agent learning fast at first so I should use a bigger alpha; gamma is discount, smaller gamma means we incline to focus on the immediately reward which is not what I want.

* initial Q-values are too small

I initialized all the Q-values to 1.

(update on July. 5th)

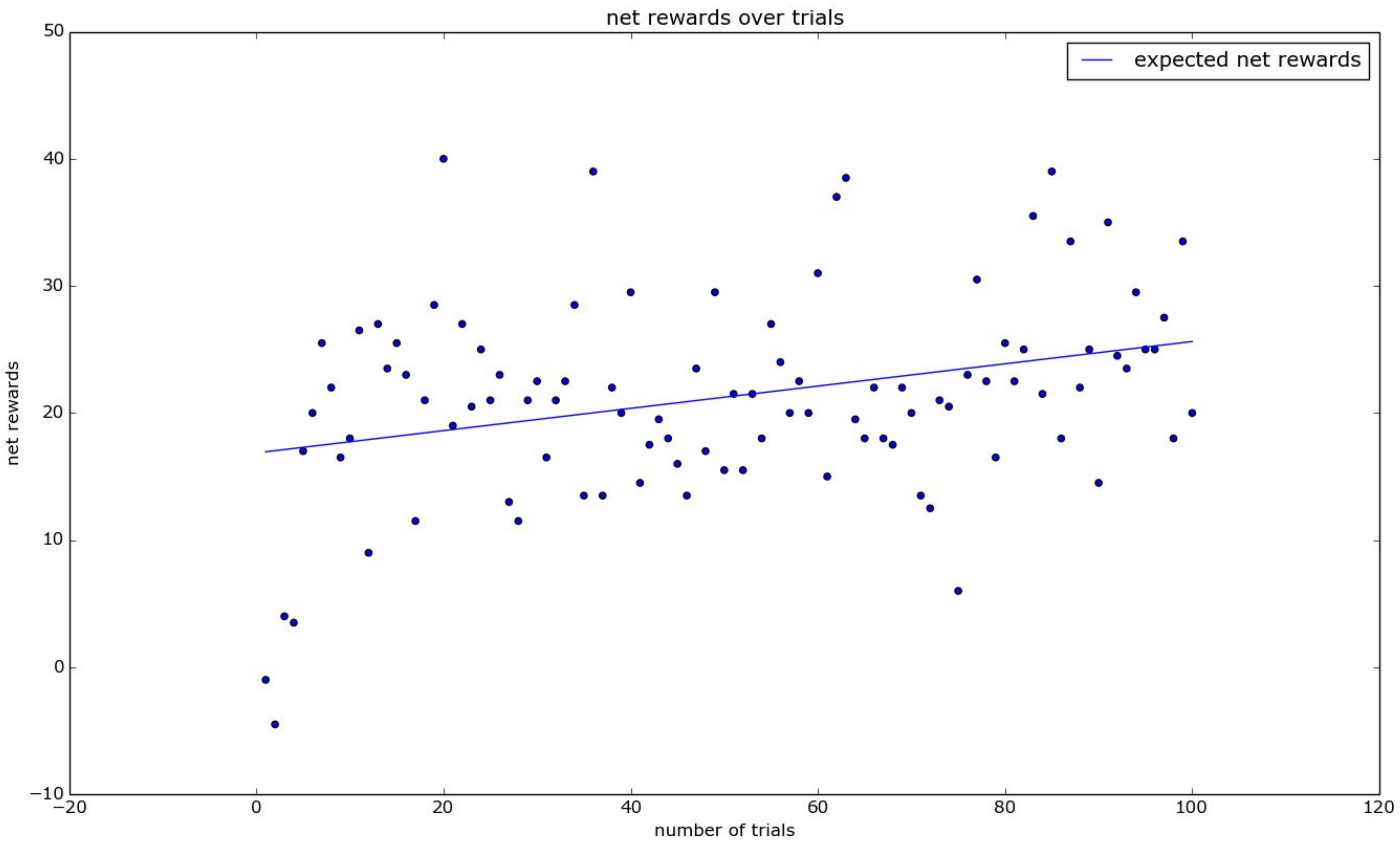
It turns out the problem has nothing to do with alpha, gamma and Q-values. **The true problems are** :

* I hard coded the traffic rules into the get_action func and
* there was no randomness when chose an action.
* Plus, the agent's state didn't keep track of the next waypoint.

All those things together make the agent sit still. **Then what I have done**:

* After I reimplemented the get_action function, that is put randomness in it;
* used the new state construction, that is put next waypoint in the state tuple;
* generated the Q-table on the fly instead of constructing it at the very beginning;

The agent works pretty well. Below is a picture shows the net reward goes up over 100 trials. I fit a line into the scatter plot to show the trend of net rewards over trials.



5. *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?*

Answer: The I think the agent get pretty close to finding an optimal policy. My parameters are:

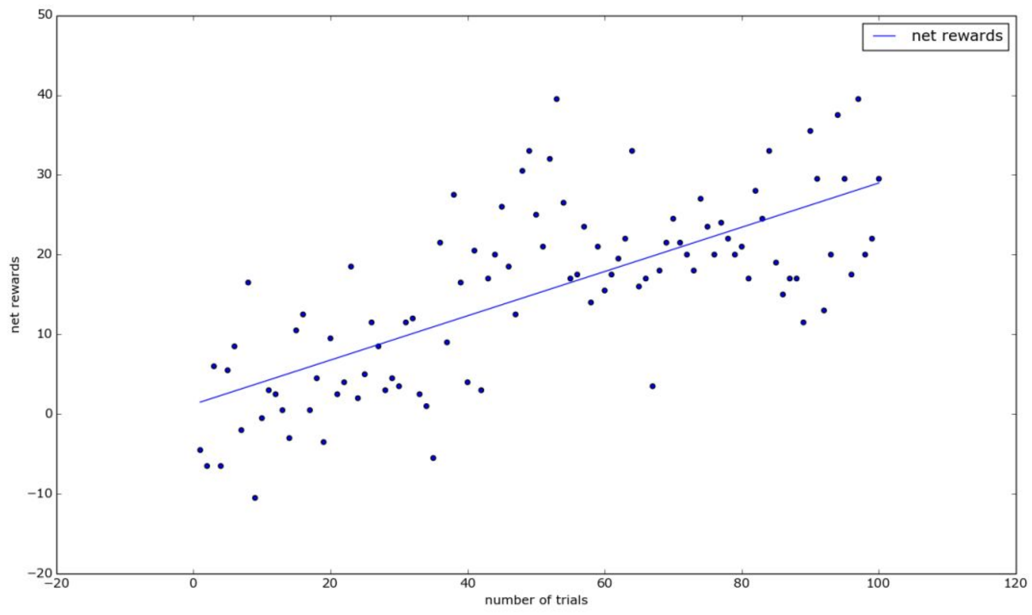
alpha: The learning rate. I have read a post where people talk about decreasing the learning rate over all the trials. So that at the beginning, the agent may pay more attentions to try all kinds of different actions and **update the Q-values heavily**, hence the **high learning rate**. Later on, most of the (state, action) pairs have been learned, and the respective Q-values have been updated towards the true ones, then the agent can rely on the Q-table to choose action and **not update the Q-values heavily**, hence the **lower learning rate** and the learned policy that is the emerged Q-table. I think this is a brilliant idea and I implemented it this way:

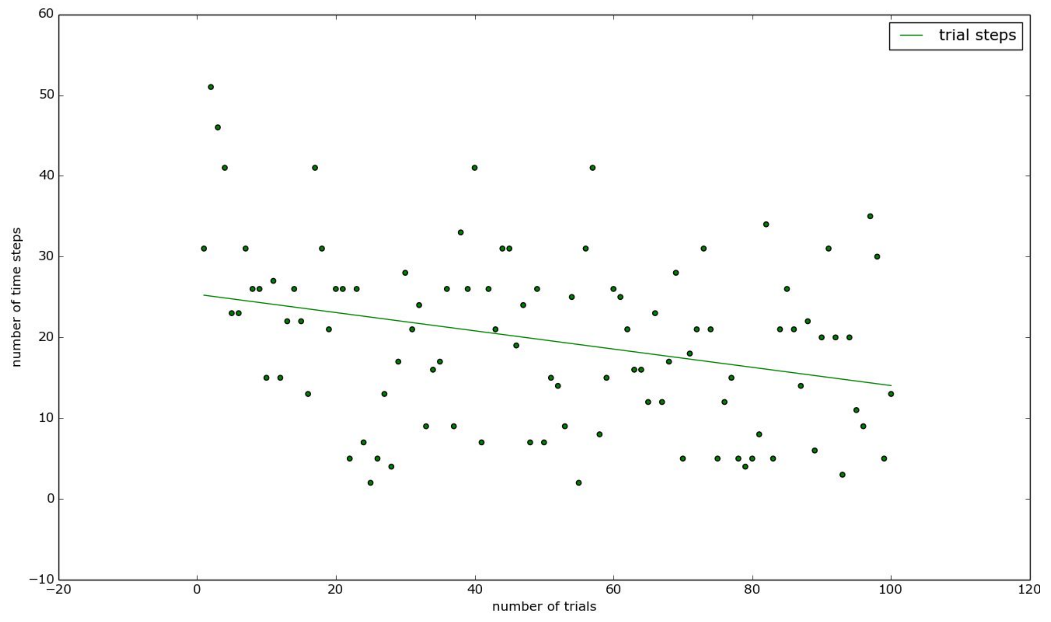
alpha = 1 - num_trial / 100.0 number of trial goes from 0 to 99, so alpha decreases with it

gamma: The discount of utilities. Higher gamma means the agent should pay more attention to the long term reward, lower the opposite. In the hope of finding the optimal policy, I have tuned this one many times, below are some of the results:

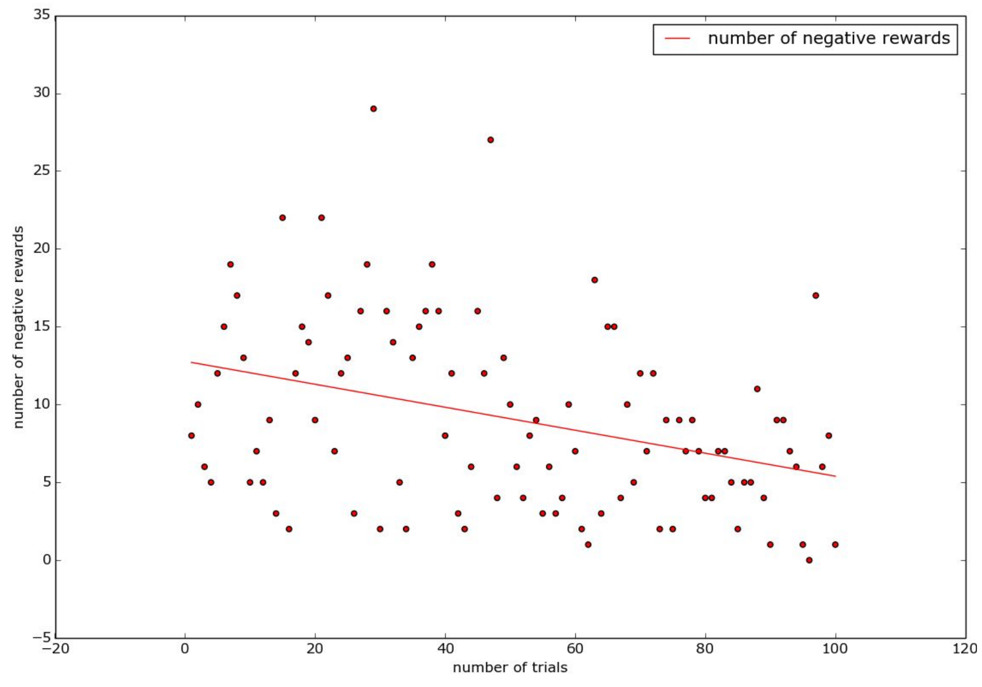
gamma = 0.9

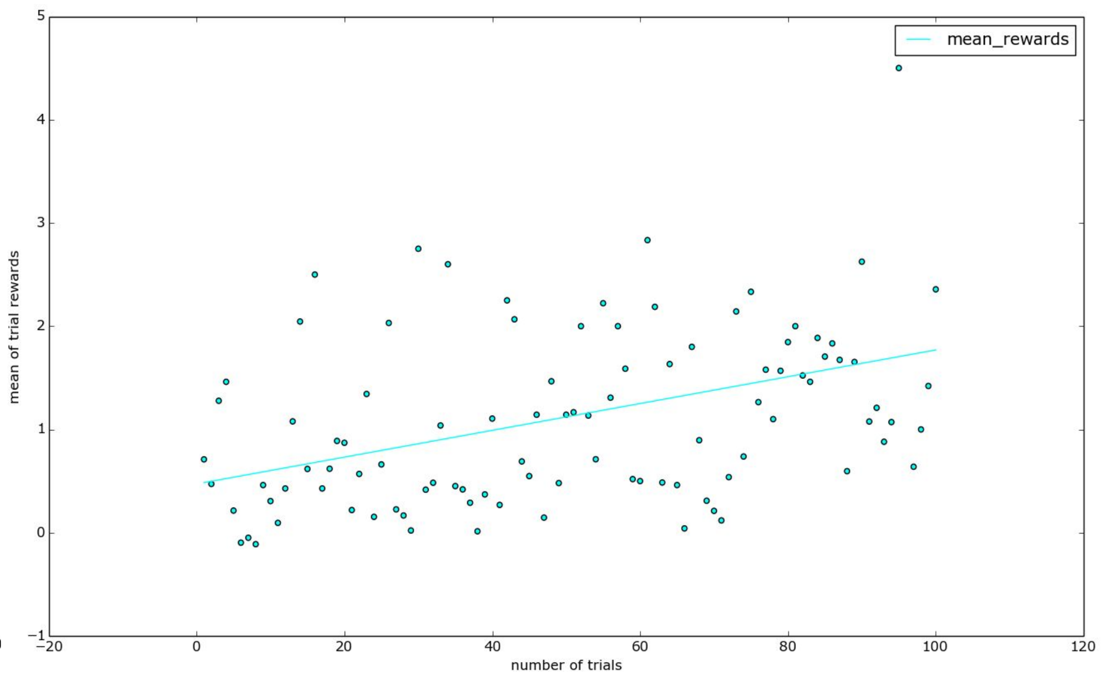
I fitted two lines into the respective scatter plots to show the trend of the data points.





The left figure shows net rewards increase over trials, the right one shows number of steps in one trial decreases over trials. The optimal policy should be "reach the destination in the minimum possible time, and not incur any penalties". This gives us two indicators of **optimal policy, net rewards and trial steps**. When the trial goes from 80 to 100,and gamma is 0.9 or 0.3,we can see either way the net rewards are more or less the same. In terms of trial steps, however, when gamma is 0.9, it is more stable than when it is 0.3. **Put together, I choose gamma**=0.9 **as my final value**. From the above two figures we can see that the agent get more rewards with fewer time steps. As the "not incur any penalties"part, we can see from the figures below that show the number of negative rewards not quite reach zero. And that's



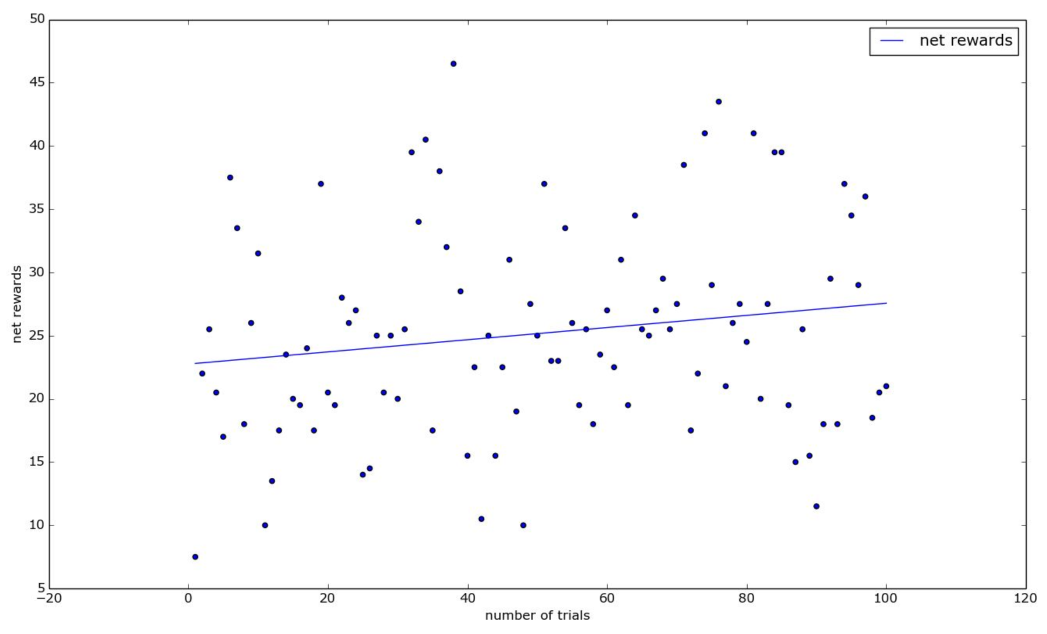


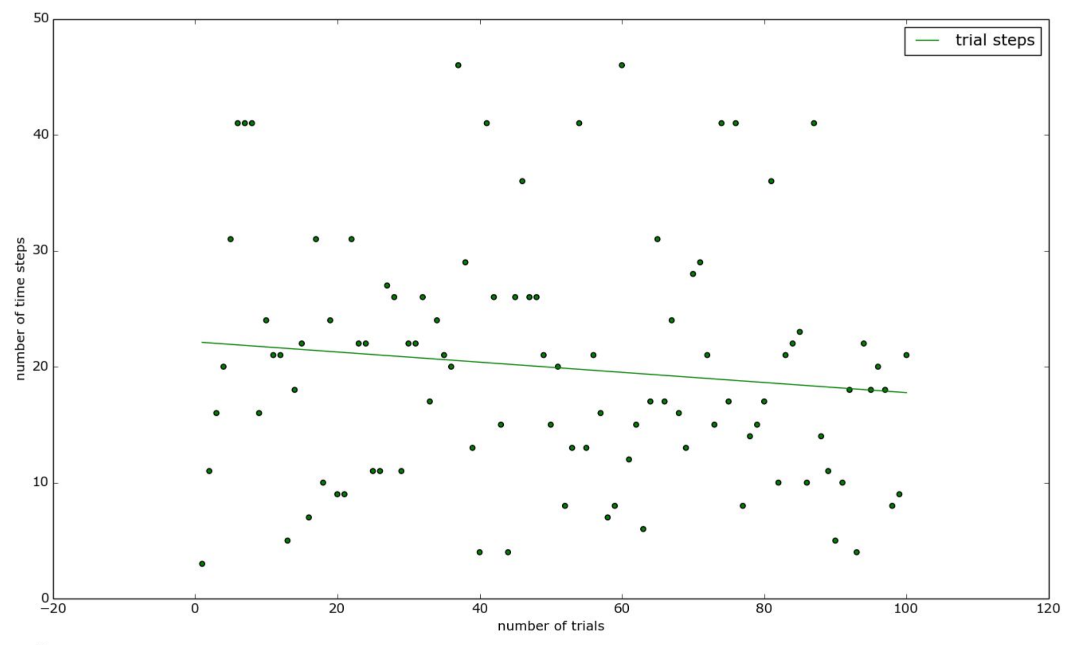
because 20% of the time it pick an action randomly, which is the learning behavior.

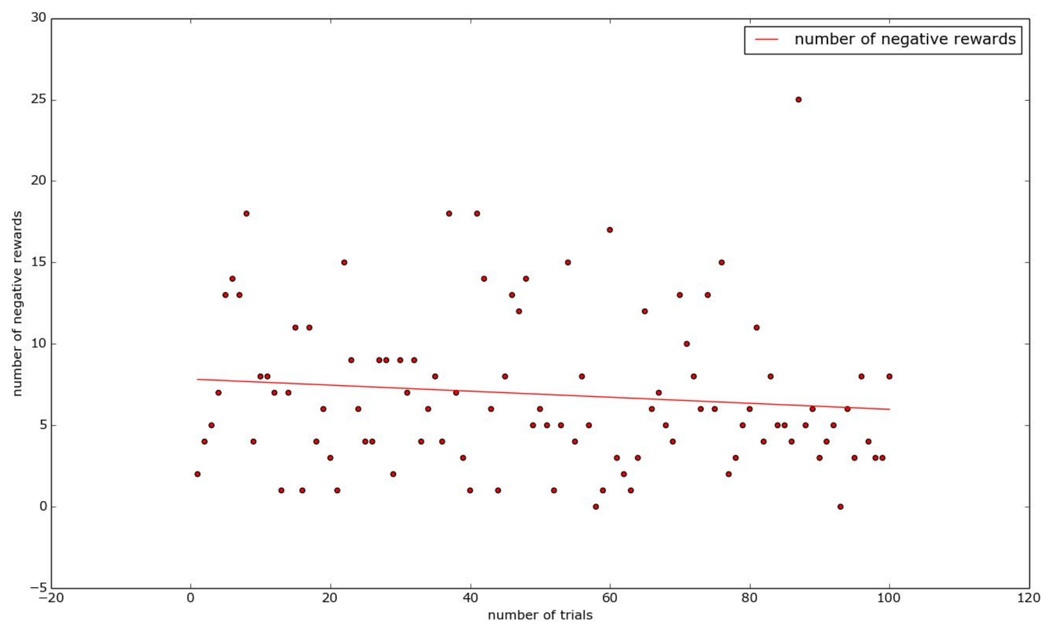
Please check out the next pages for more parameter tuning.

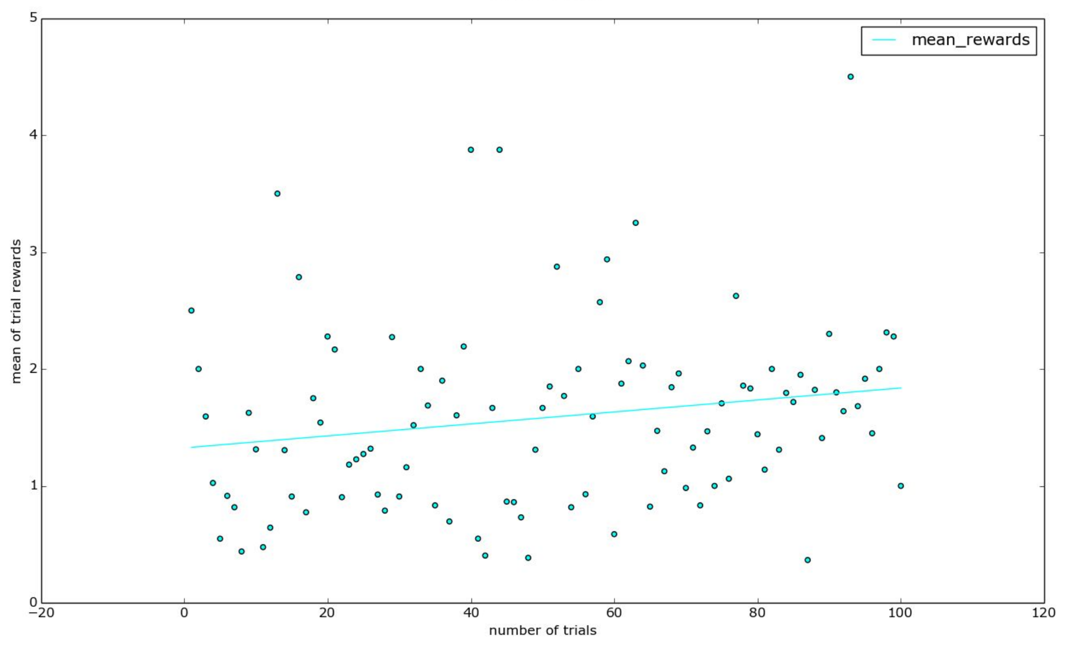
(more content in next pages)

When gamma = 0.7: Net rewards sparsely distributed around the blue line, it’ s not very stable when compared to 0.9 in terms of net rewards.

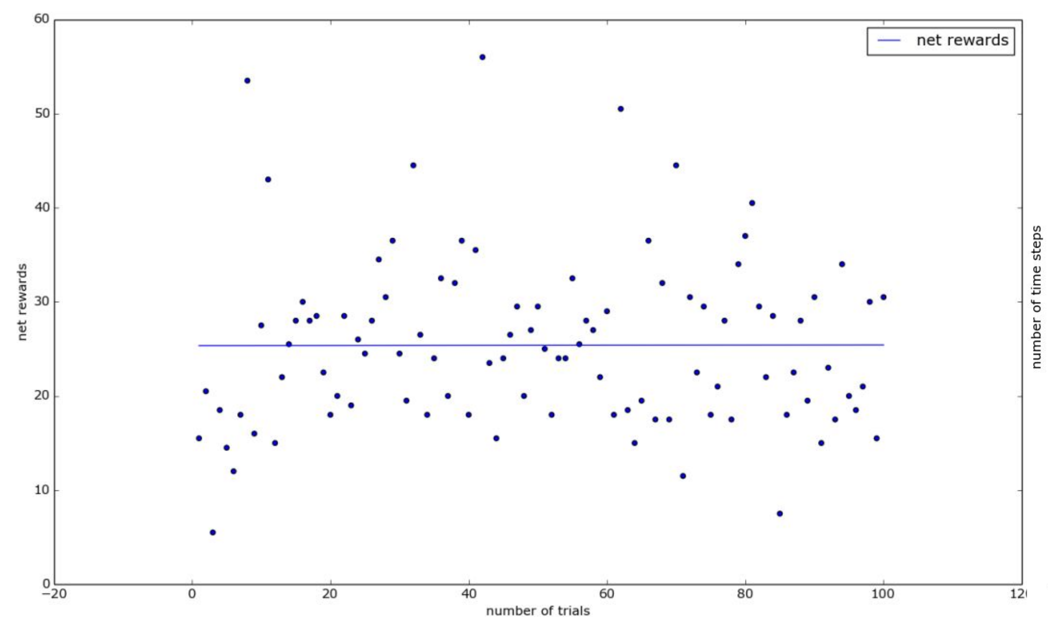


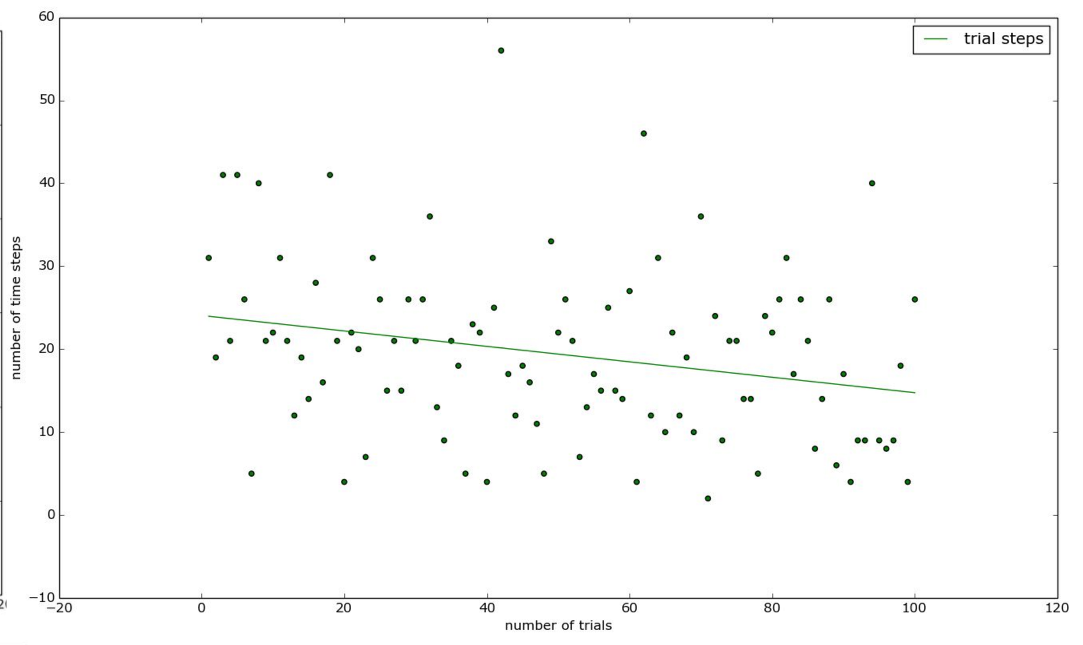


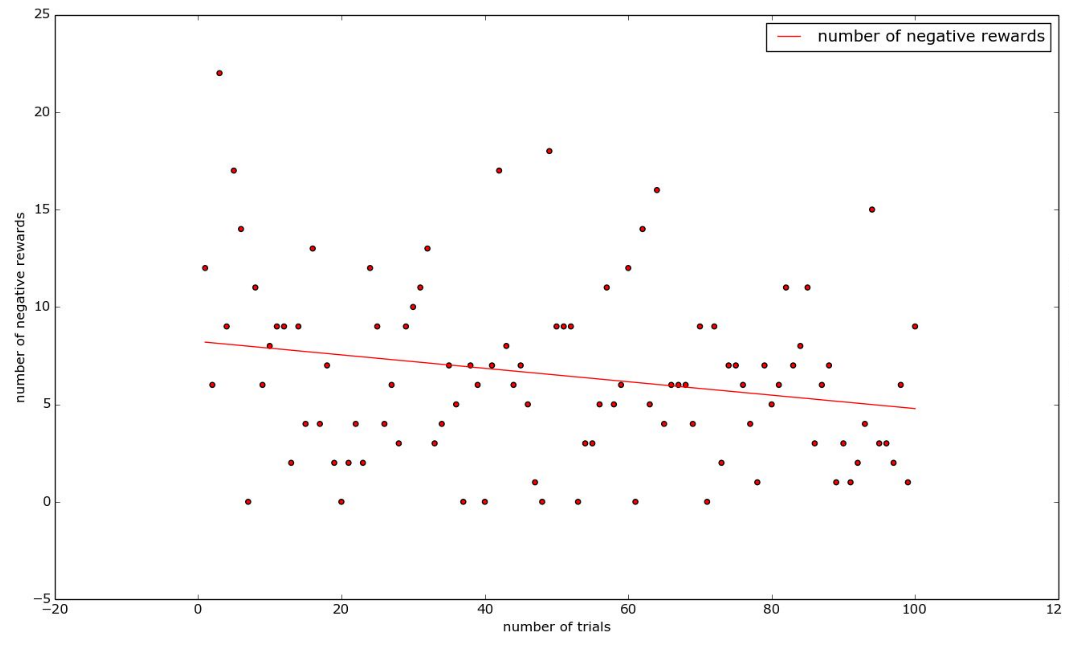


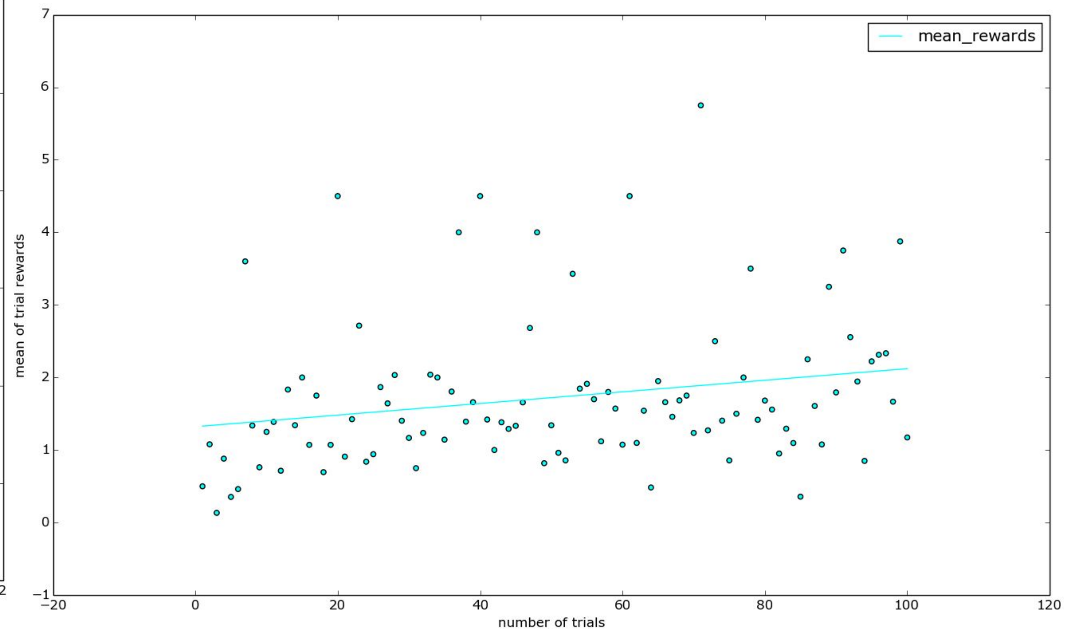


Below is when gamma=0.5 We can see the same problem as gamma=0.7

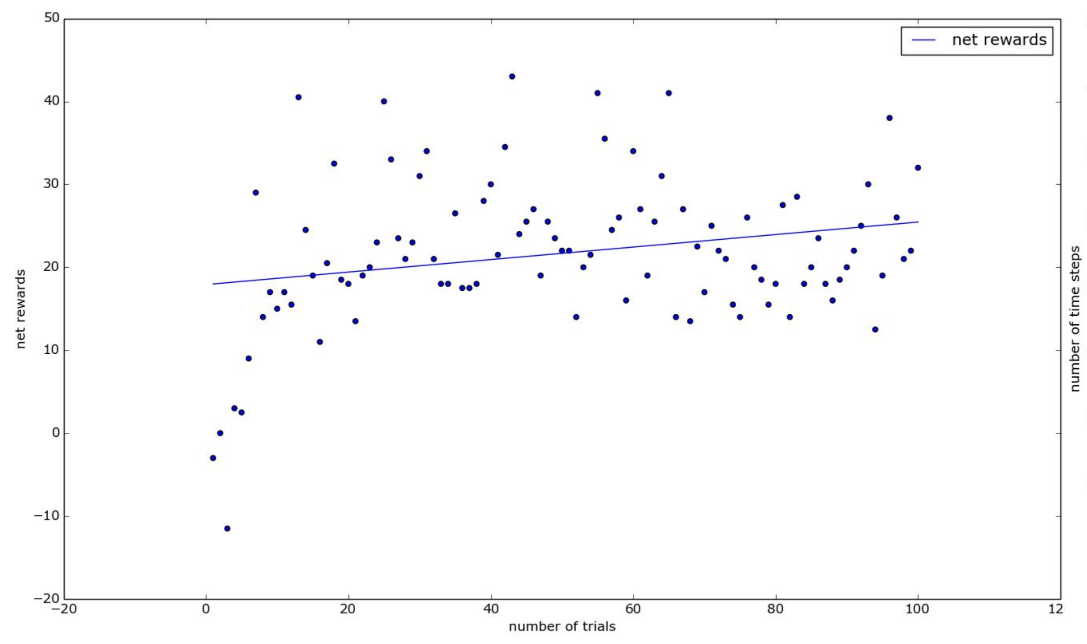


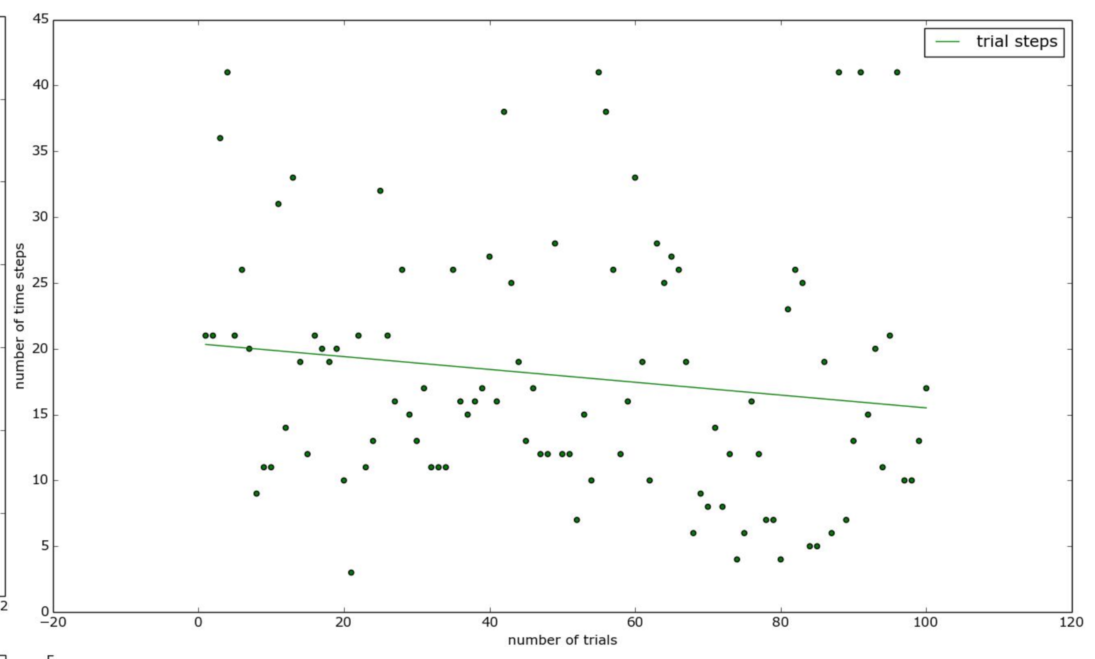


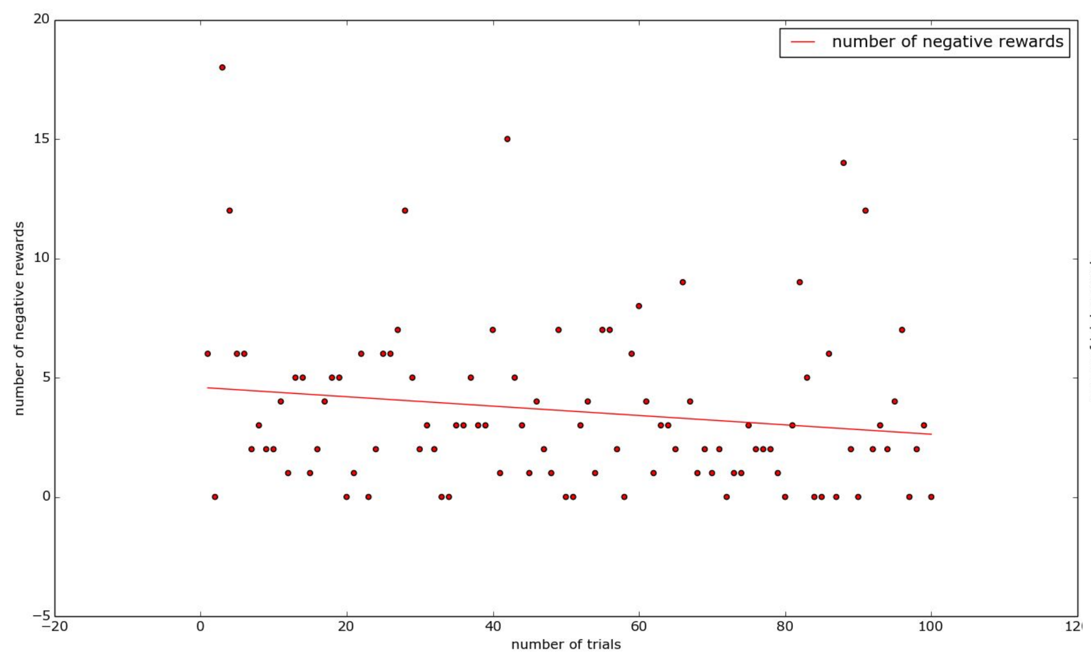


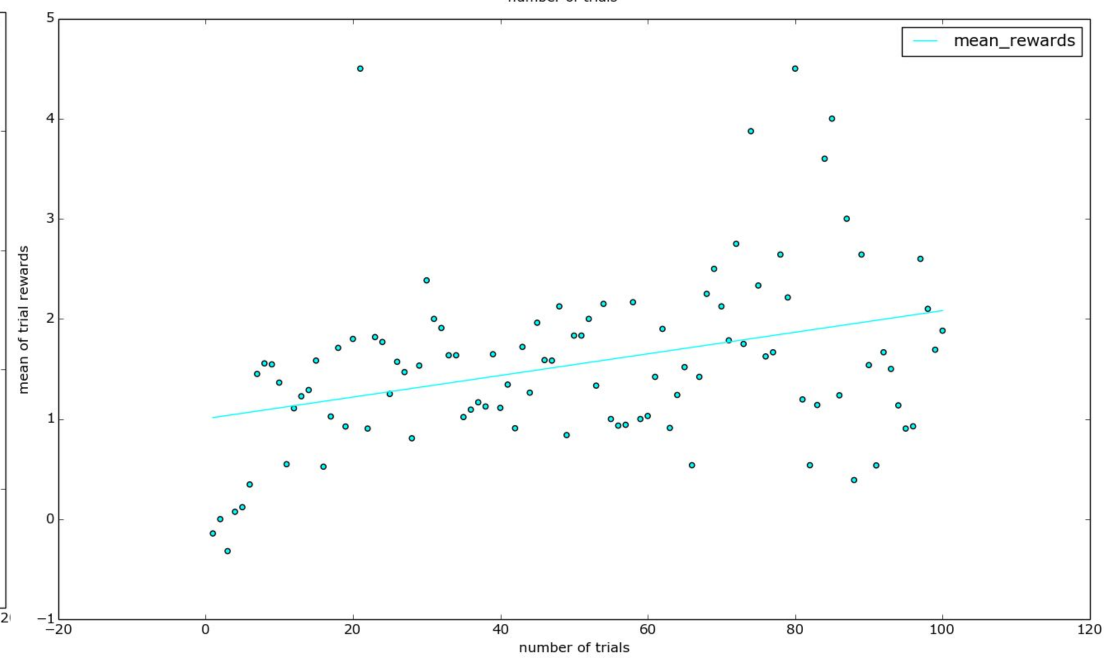


When gamma=0.3 Here, we can see the points more densely distributed around the blue line than before, and a steadily growing net rewards. It’ s better than when gamma is 0.5 or 0.7. The number of points over net_reward=30 is a lot more than the number of points in the same interval as gamma=0.1. In fact, if we only want to maximize the total net rewards, then gamma=0.3 is the best choice. Compare to when gamma=0.9, however, the trial steps are less stable. That’ s why I chose gamma=0.9 as the final value.

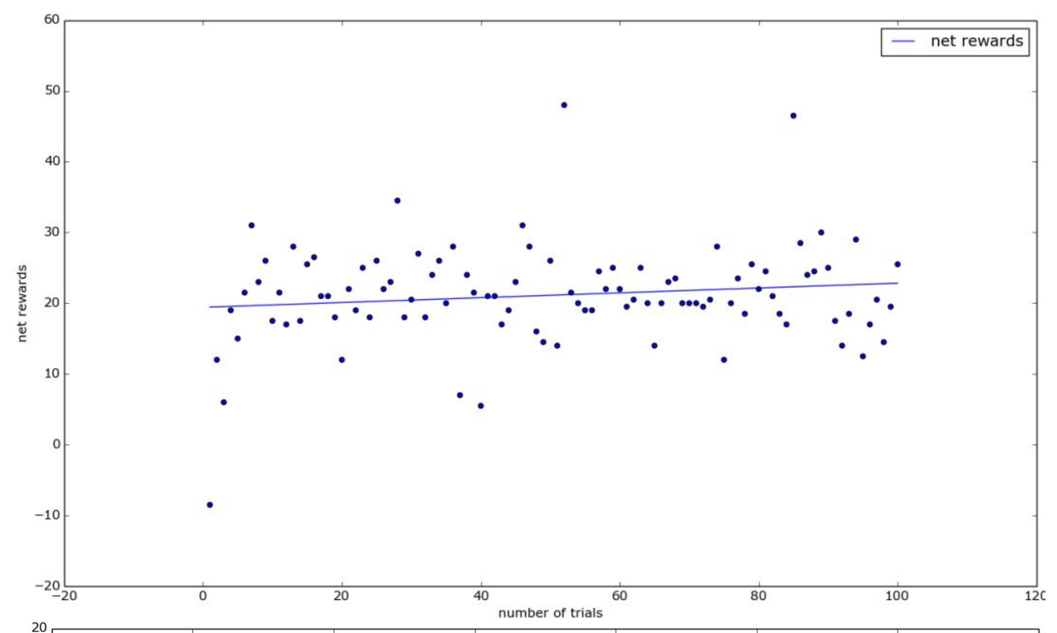


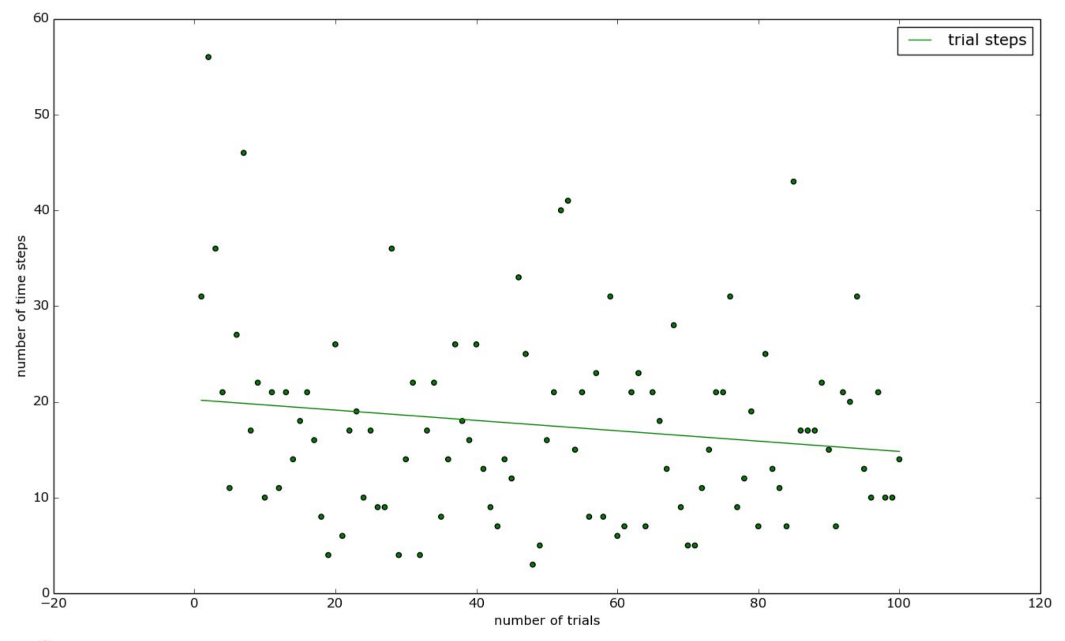


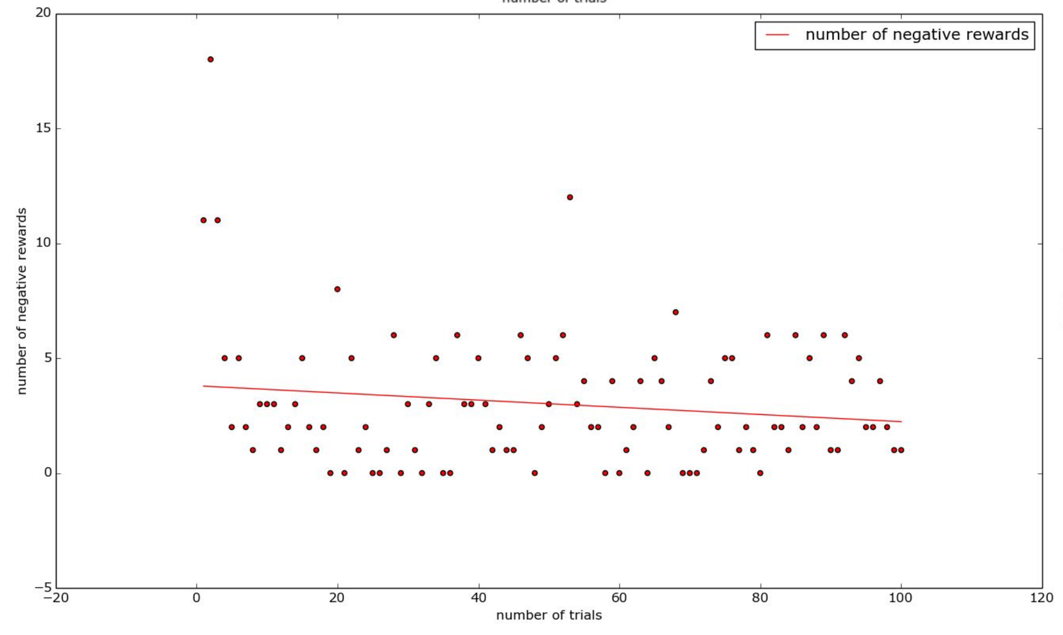


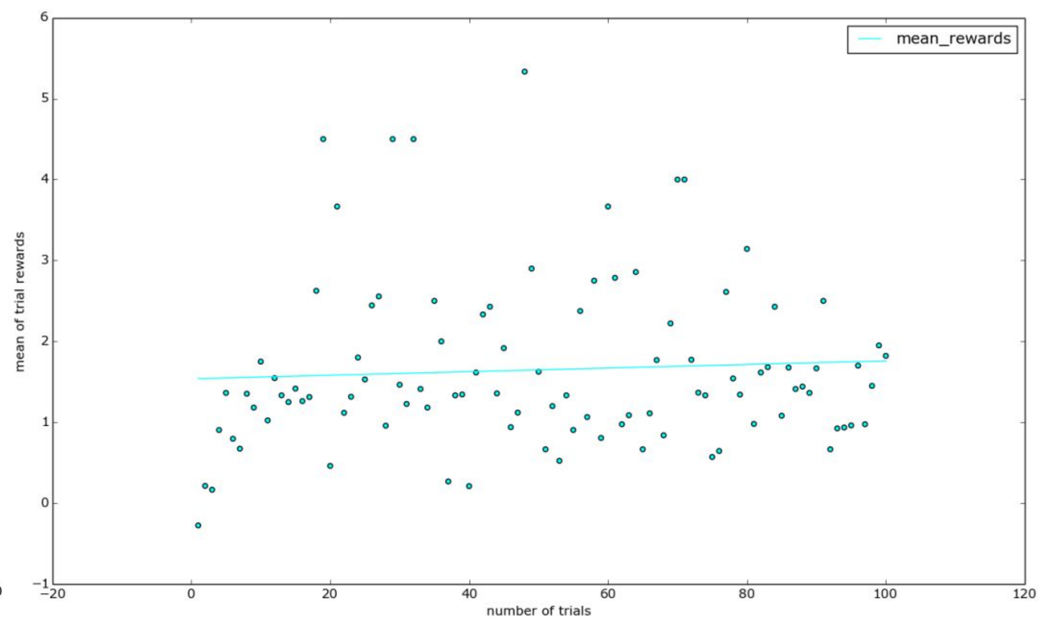


gamma=0.1 It’ s more stable than when gamma is 0.5 or 0.7









Update on July 12:

1. The optimal policy.

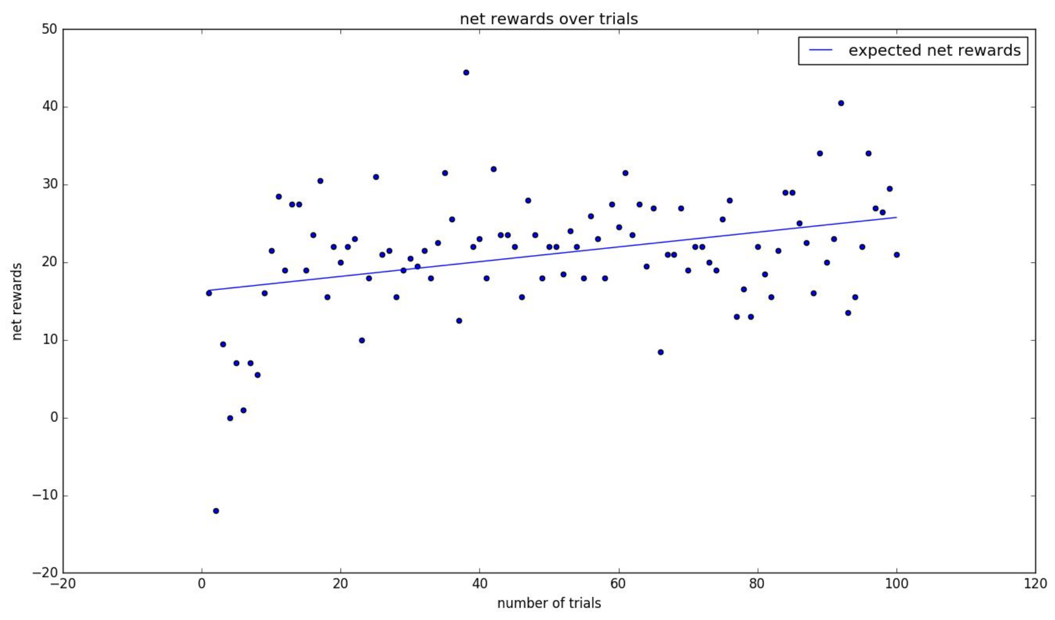
The optimal policy should be reach the destination in the minimum possible time, and not incur any penalties. This gives us two indicators of optimal policy:

* trial steps
* net rewards

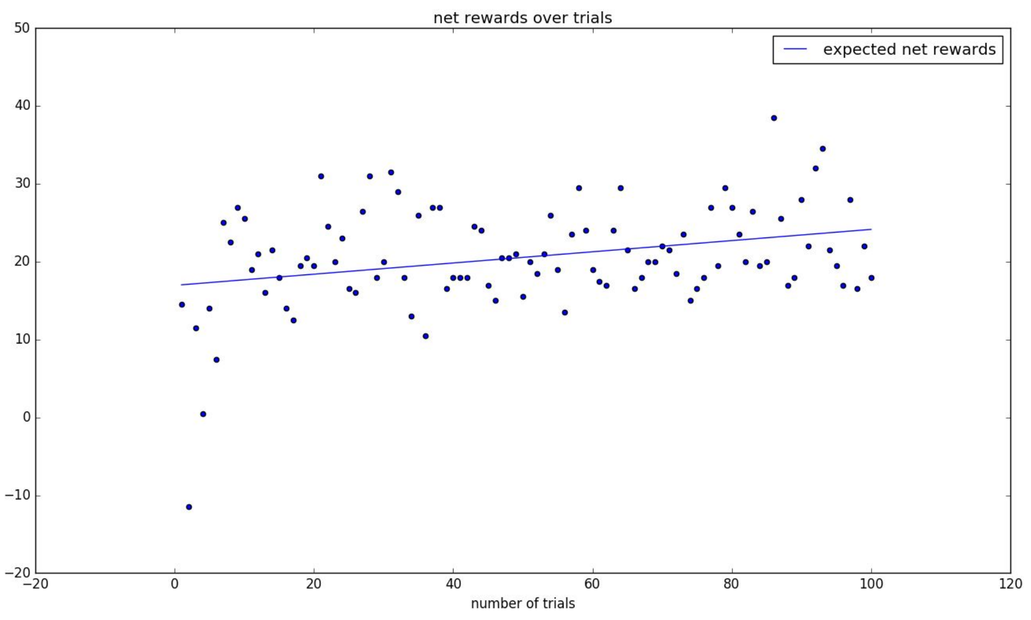
Let's say I'm a passenger on the smartcab, it's obvious that I want to get to my destination as soon as possible. This means the learning agent should safely arrive at my intentional point in minimal trial steps. At the same time, the agent must comply with the traffic rules, and get the maximum rewards. These two requirements inherently contradict each other, so there are a lot of trade offs when we tune the hyperparameters. When it comes to the penalty, while the ideal perfect agent can get to the destination without penalty, my agent may get to the destination most of the time, it inherently get the 0.2 probability to act randomly even with the true Q-table. That means we can't totally keep the agent from incurring penalty, though it may be at a very low level. (See number of negative rewards figures) In summary, hopefully, I could say my learning agent is pretty close to find the optimal policy.

2. Tuning alpha.

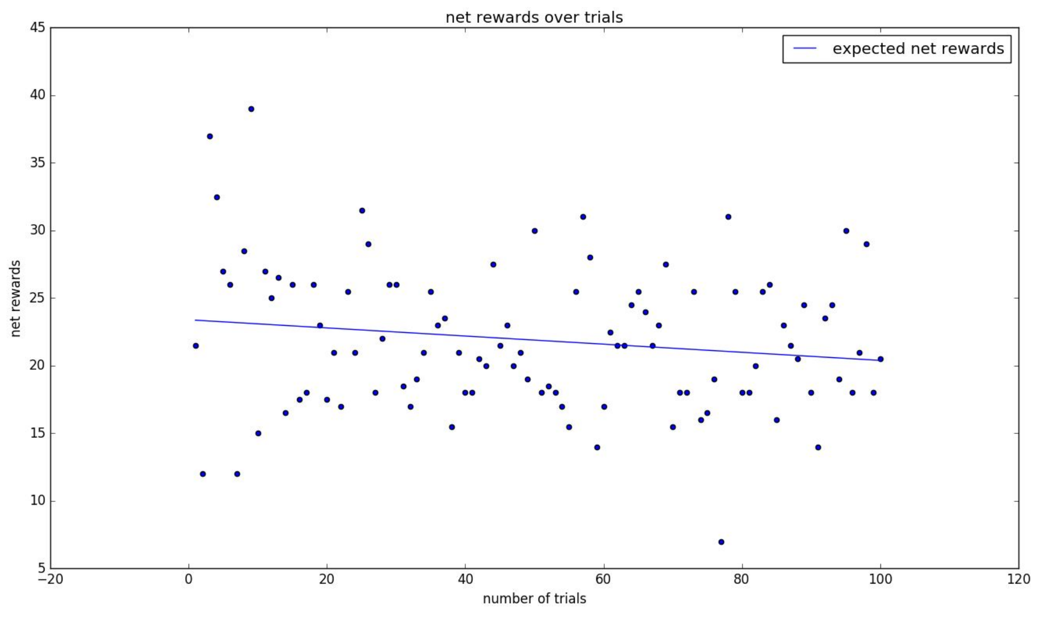
(1) Below are net rewards respect to different alpha values. We can see when alpha=0.1 and when decaying alpha, they both have similar net rewards. When decaying alpha, however, we have more stable trial steps than when alpha=0.1. So, I choose decaying alpha as my final alpha value.



Decaying alpha (above)

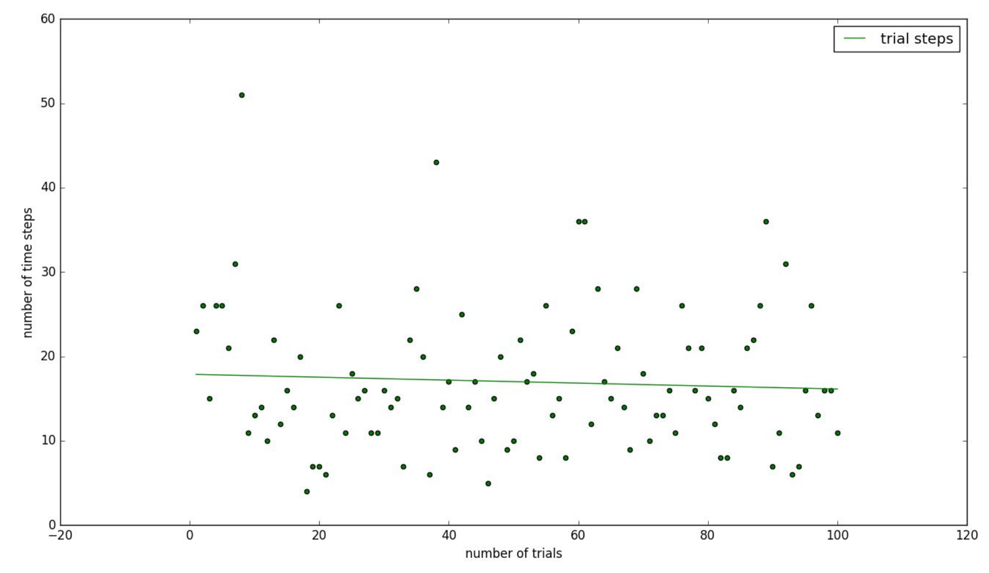


alpha=0.1(above)

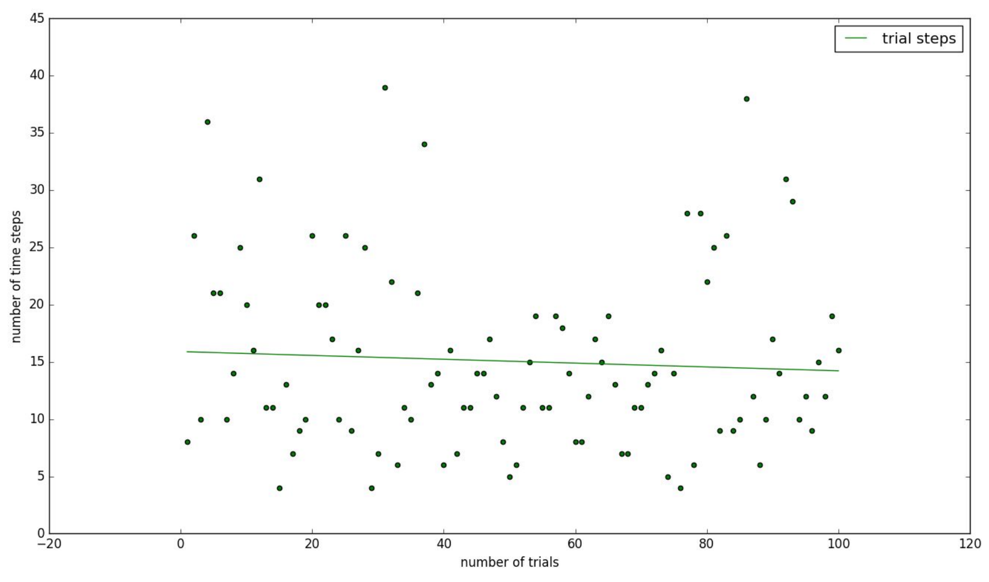


alpha=0.7(above)

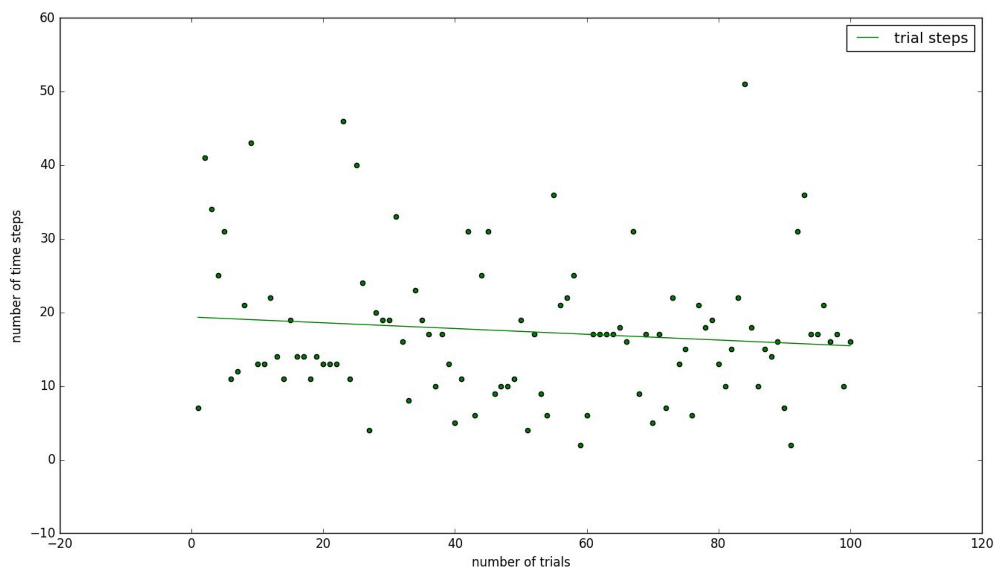
(2) Trial steps in different alpha values. We can see when decaying alpha, it has more stable trial steps than when alpha=0.1



Decaying alpha(above)

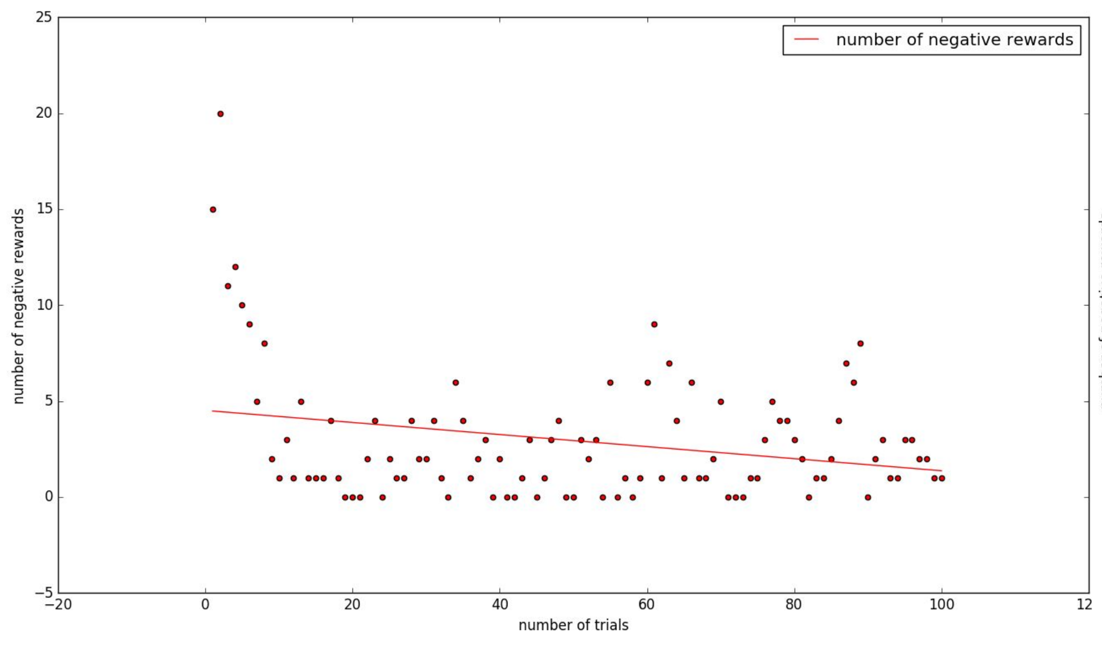


alpha=0.1 (above)

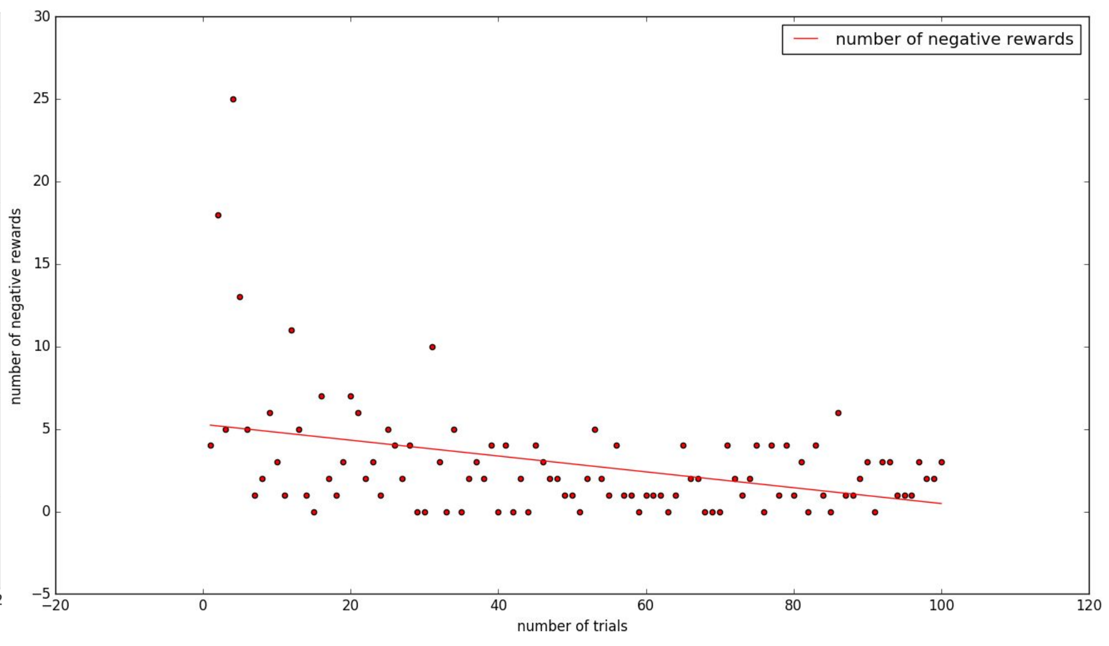


alpha=0.7 (above)

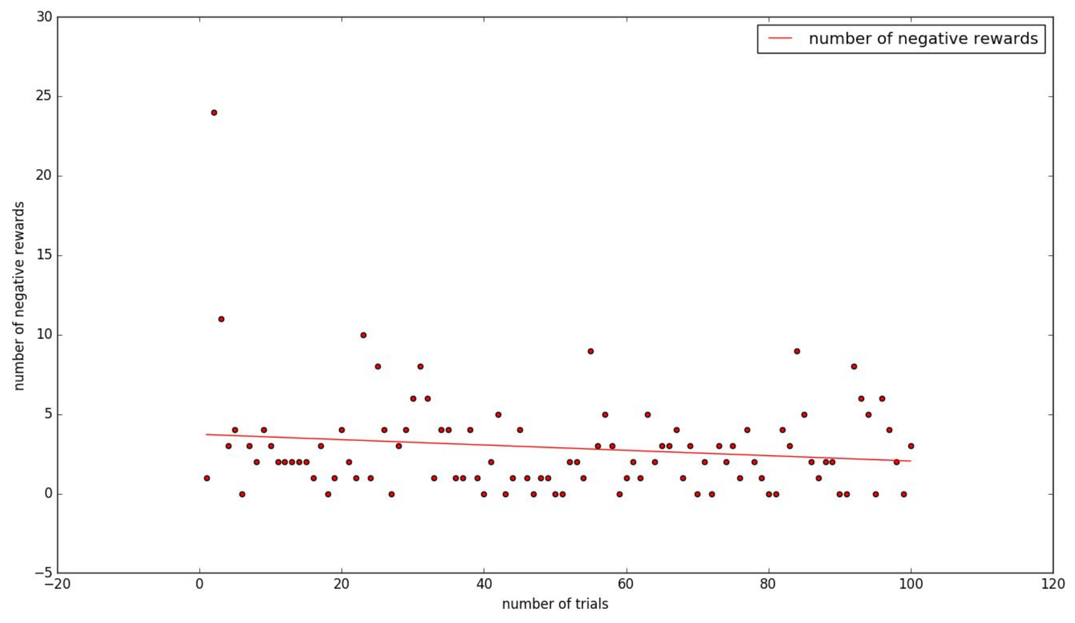
(3) Number of negative rewards in every trial.



Decaying alpha (above)

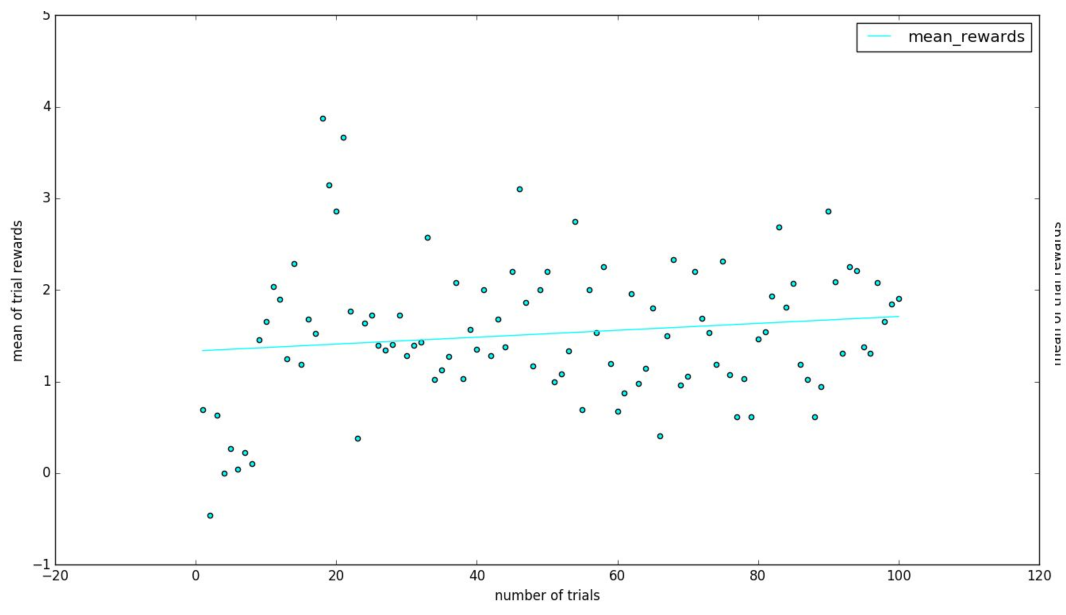


alpha=0.1 (above)

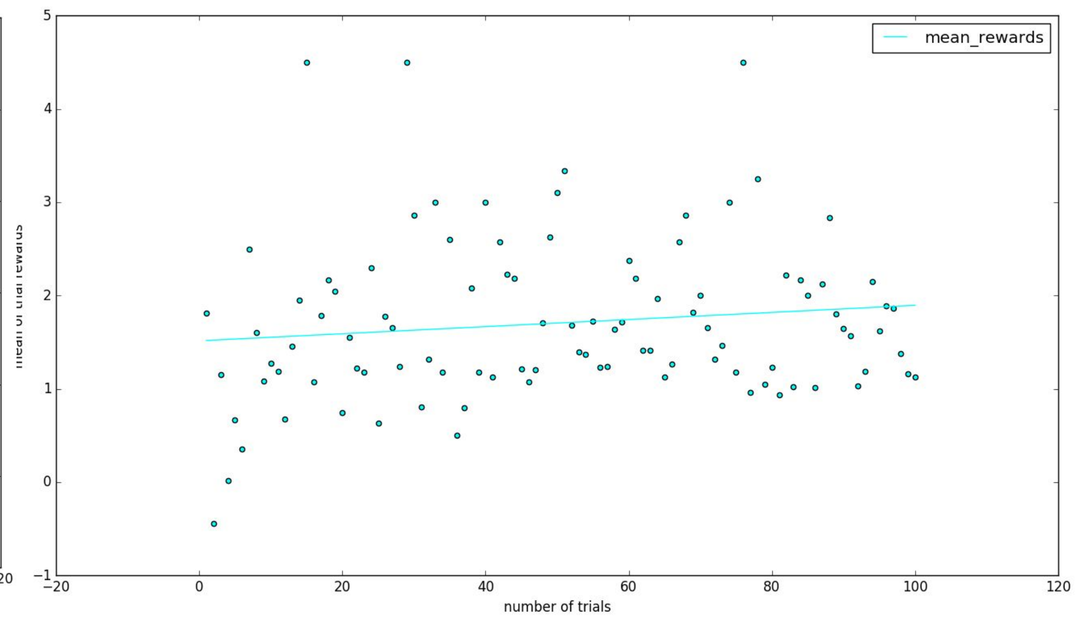


alpha=0.7 (above)

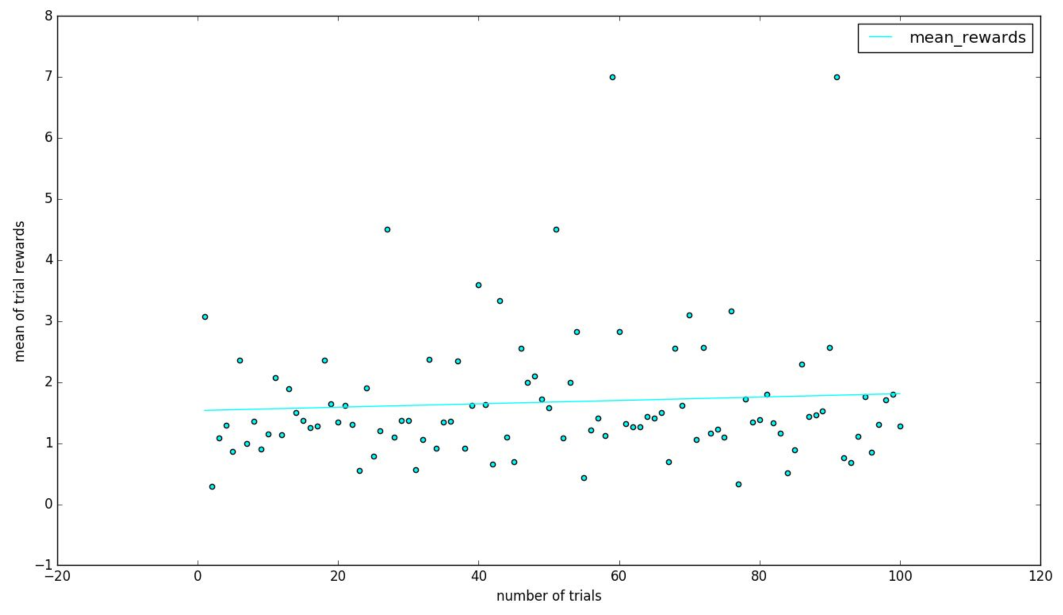
(4) Mean rewards of different alpha values.



Decaying alpha (above)



alpha=0.1 (above)



alpha=0.7 (above)