MOUNTAINCAR V0-OPEANAI GYM

A COURSE PROJECT REPORT SUBMITTED
TO

UNIVERSITY OF MARYLAND, COLLEGE PARK

Professional Masters in Robotics Engineering
Course: Behavioral planning of Autonomous Robots
(CMSC818B)

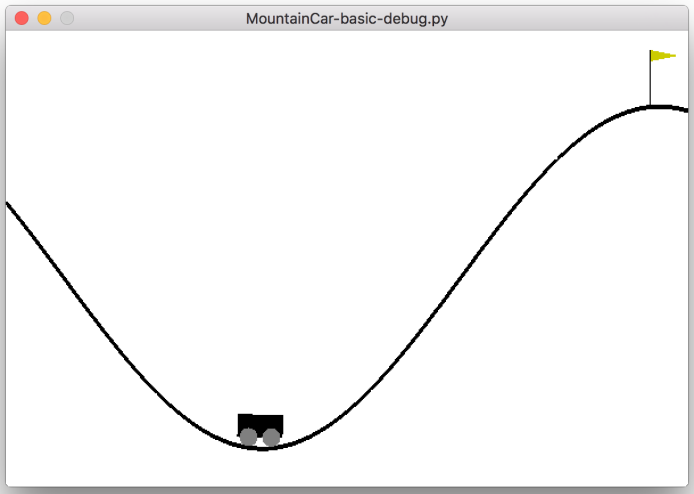
SUBMITTED BY:

PIYUSH BHUVA **UID-116327473**

Explanation of mountain car openai gym problem:

OpenAI Gym:

OpenAI gym is a wonderful tool to discover the world of reinforcement learning and it has some model/environments that can be directly experimented on. One of those tools is mountaincar v0 problem in discrete action space. Again the environment looks something like this:



Basically here your agent is a cart which has to reach to the yellow flag and we have to learn the policy about the action which acts on environment which is in this case the sag between mountains. Mountain car has to learn the appropriate actions to take at each state (position and velocity) which action will take us to right yellow flag.

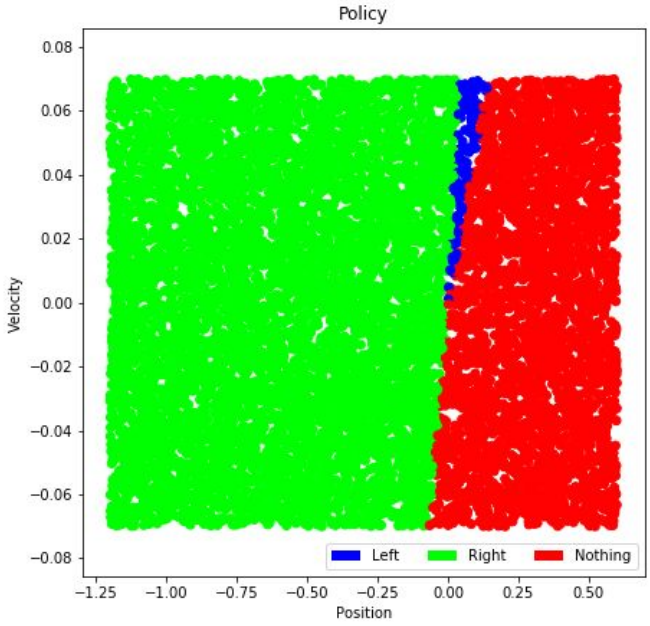
Here basic human instinct says that after for the left most part up the hill push should be right hand side. But giving always that push will result in the failure sometimes. As thrust isn't always sufficient to reach to top of mountains.So, we have to design a policy to reach uptop here.

Mechanics and rule of the game:

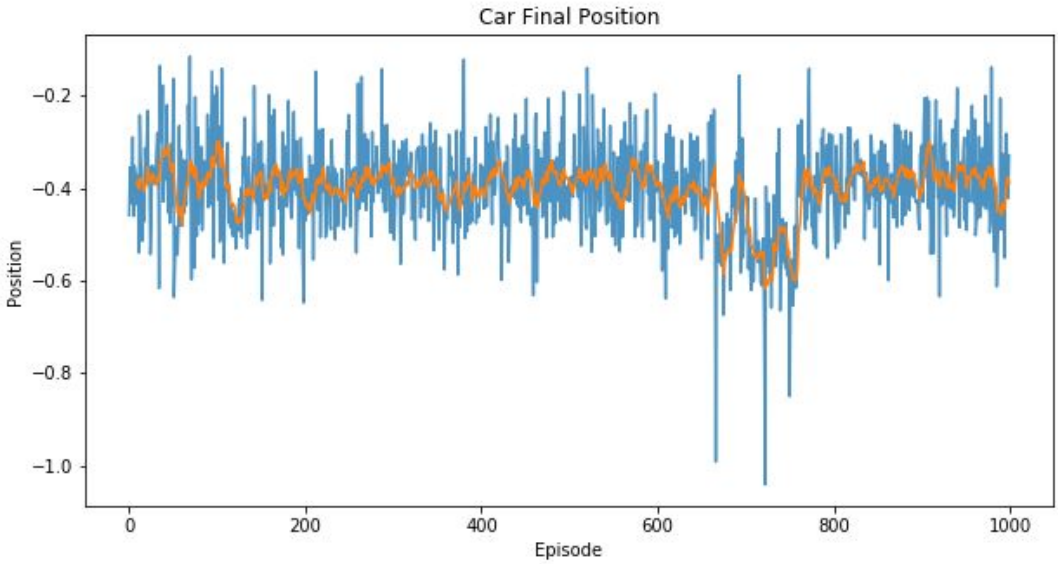
1. The episode ends either after reaching at goal or after 200 time stamps.
2. For each time stamps we get reward of -1. So, game ends at -200 or lesser to that value.
3. Three actions are available 0-Go right,1-Neutral and 2-Go left.

Neural network architecture:

I have used pytorch to build neural network with sequential model with the use of 100 hidden layers to update the weights. I wanted to select the best tradeoffs to runtime vs quality of the neural network. So, I have started with 4 layers in between and observed the policy. Which looked like the image given below.

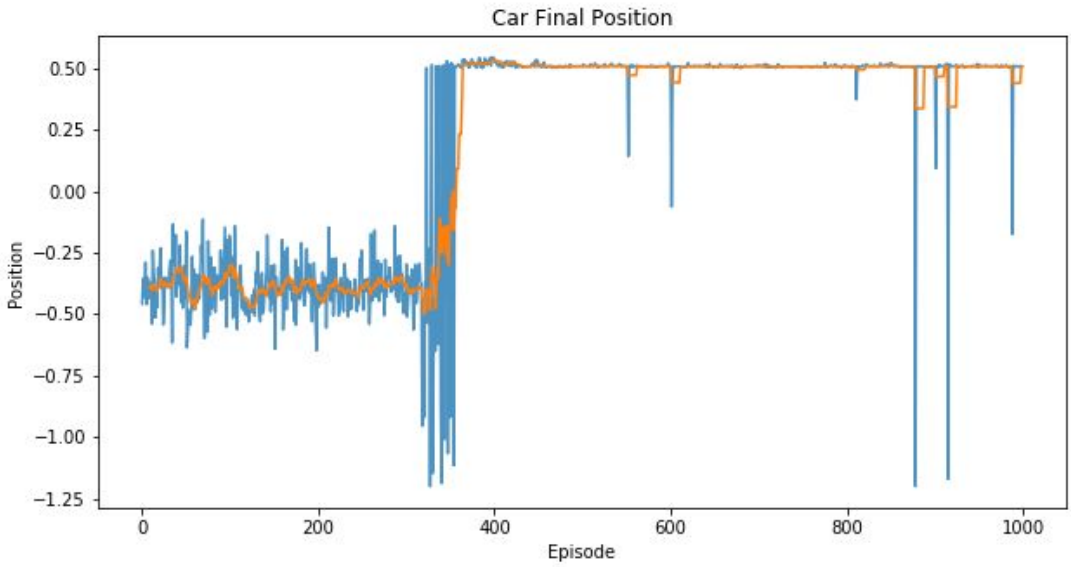


This policy learned is really a bad policy and the reason for that is for the major part it says for the most position go to right and which is a bad move to converge policy. This policy took 2 minutes to run on my computer.The reward vs policy can be also looked up like:



This shows that our policy never actually converged and bounced back and fourth in very unlikely region(0.5) is final position but it didn't even touch nearby that value.

On the other I have bumped up the number of layers to 100 and it has given a stable policy which eventually learns that thrust to right is not the best way to reach top and this image shows that bumping up hidden layers we reach to the nice policy.And the positions looks like,



So, we after 400 iteration policy learns how to achieve goal frequently as we can see.Again the time taken here was almost 5 minutes to run(a double time but affordable.)

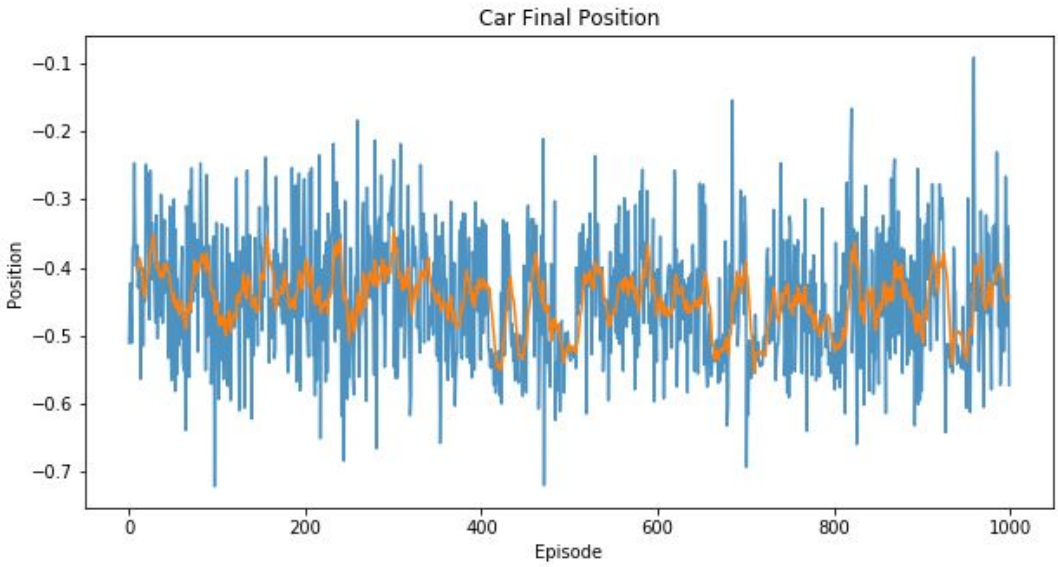
The activation function has not been changed in comparison to keep case straight. I have kept bias term in the program to zero in both the cases.
I have selected sequential model for the application.Basically the updation of weights will be sequential.The reason I have choose this architecture and number of hidden layers is a little different.

I have considered Gaussian Kernel to be selected but from several online articles the kernel design was rigorous and the change in loss type impacts hugely on final product. So, I decided to take NN approach to solved this with standard parameters and it gave me decent results. All the little modification has been described thoroughly below. But, selection of number of hidden layers is described above.

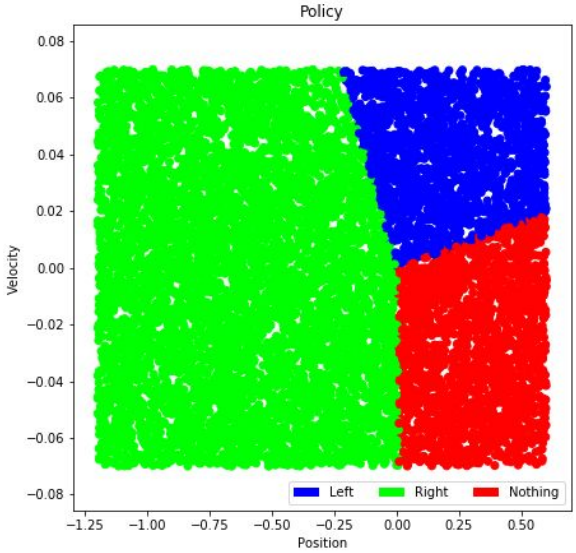
The initial weights are set to -1 in the beginning and after every episode we are going to update this weights.Some actions will be given more weights (As thy are producing decent results and other one are not).At the end of 1000 episodes we will have final weights. I have used pytorch for this application and to create neural network. The runtime will be displayed while running and from my calculation it takes 5-6 minutes to run.

Exploration vs Exploitation Strategy:

This policy determines how much you want to explore(Find new ways) or exploit(Trusting old Q-values) . This option is always available and determined by epsilon parameter.
If epsilon=0 means your policy will explore more.As in the code if that random number falls below epsilon you want to take a new action and on the other hand you will trust your old q-values epsilon=1. Default value is set to 0.3 and decreased every single time by a factor of 0.95 if we successfully complete the episode.
This changes means that you are trusting you older values more now. Ideally, this also means that after several iterations you want your policy to learn and assuming that has happened you want to trust updated q values more.
Now, here are practical show of changing the epsilon values.
I kept the value of epsilon to be 0.5 which means it equally exploit and explore and the results are as below:



This shows that we have way too much randomness in this final position and actually it didn't reach the final position. And the image above this image shows how it converge setting epsilon to 0.3.
So, setting appropriate value of epsilon helped to balance between results of strategy. The policy with parameter to be 0.5 is looking like below.



MSE Loss after iteration:

Ideally MSE describes the difference between the q values and after several iteration it should decrease (again ideally!). If you want try out yourself in the main code please uncomment print(loss) line.It will print out the loss between Q-values after 100 iteration. But the difference between actual and ideal Q-values are our MSE loss.Again we can see that when our position is near to the goal(y=0.5) our losses decreases dramatically which is a good thing as it is a sign they want to converge.

tensor(1.1439e-07, grad_fn=<MseLossBackward>)
tensor(2.8349e-07, grad_fn=<MseLossBackward>)
tensor(1.1337e-06, grad_fn=<MseLossBackward>)
tensor(3.7276e-06, grad_fn=<MseLossBackward>)
tensor(9.7602e-06, grad_fn=<MseLossBackward>)
tensor(2.1488e-05, grad_fn=<MseLossBackward>)
tensor(4.1651e-05, grad_fn=<MseLossBackward>)
tensor(7.3360e-05, grad_fn=<MseLossBackward>)

They look like this near to the goal and the loss will increase drastically when we reset our condition means we are back at the bottom and here loss is increased again.
For an example, this are the loss directly after this steps and observe the difference.

tensor(0.0100, grad_fn=<MseLossBackward>)
tensor(0.0003, grad_fn=<MseLossBackward>)
tensor(0.0004, grad_fn=<MseLossBackward>)
tensor(0.0116, grad_fn=<MseLossBackward>)
tensor(0.0127, grad_fn=<MseLossBackward>)

tensor(0.0011, grad_fn=<MseLossBackward>)
tensor(0.0144, grad_fn=<MseLossBackward>)
tensor(0.0018, grad_fn=<MseLossBackward>)
tensor(0.0018, grad_fn=<MseLossBackward>)
tensor(0.0019, grad_fn=<MseLossBackward>)
tensor(0.0165, grad_fn=<MseLossBackward>)

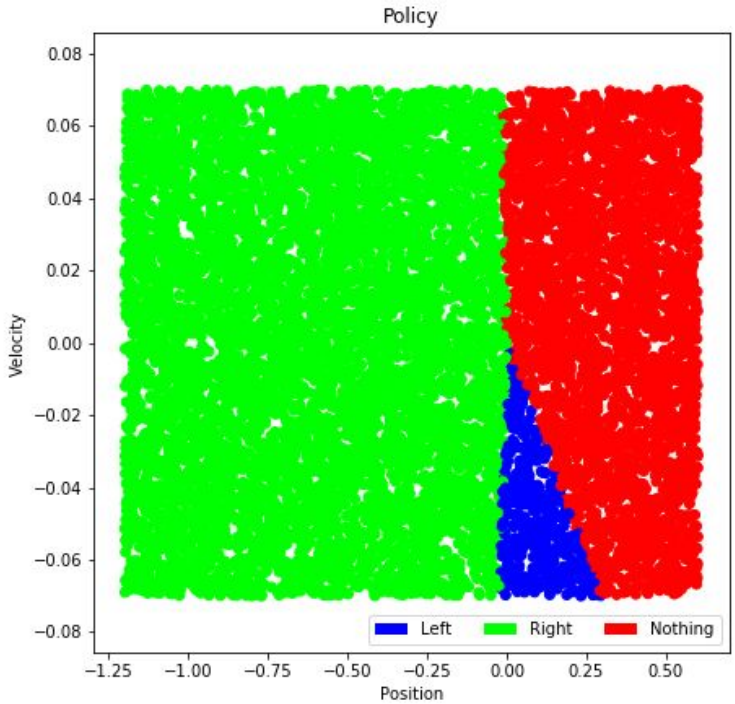
We can see an exponential jump over this, which is understandable. There is no need that consecutive losses should always decrease.(After all the action are random selections). But the overall trend should be decreasing and as shown above they will try to converge to minimum value and after reset you are again repeating the same process.(You can observe the effect by uncommenting the lines in code.)

The function approximator is actually converging nicely and they follow this acceptable curve above. And which is a good thing!

Effect of hyperparameters:

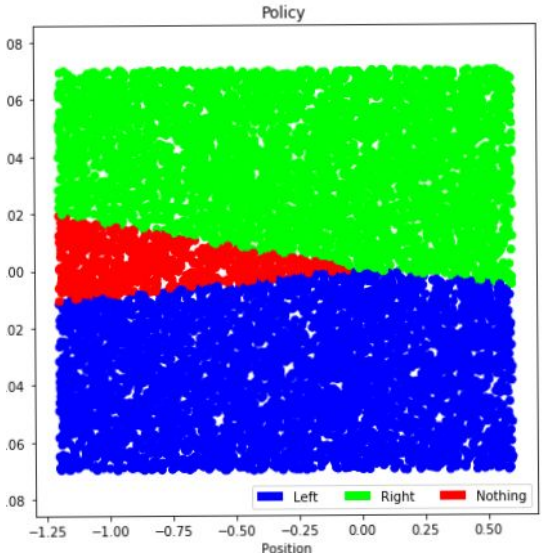
Learning rate:alpha

Changing the learning rate also affects the performance of the model. I have increased the learning rate in order to increase performance. I have set learning rate to be 0.003 in the beginning of the iteration and the policy learned is again giving bad results in which it tells us to



push right.

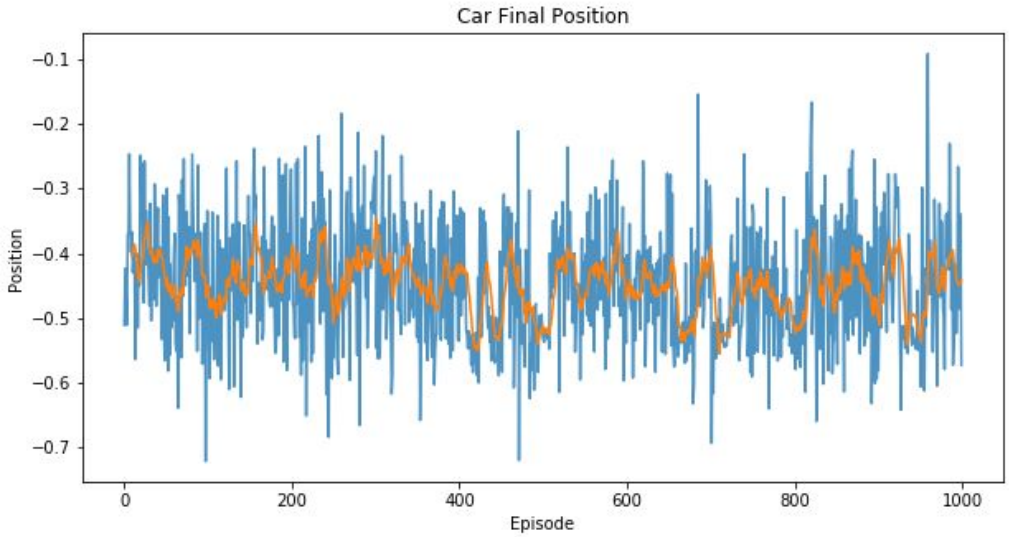
But that too is also ok but the problem is even at the higher position it wants to do nothing(Look at the red region). This shows the lack of convergence as the learning rate is set to be higher at the beginning. And the success here is actually 0% and that means it never reaches to the top. Again this implies convergence issue.Hence this has been kept around 0.001 and this way we can converge to optimal value. After setting it to 0.001 we get policy like:



This should be the ideal policy here. Which leads to convergence after around 400 iterations.

Discount factor:

This factor is basically determines how to decrease your epsilon. Again this parameter helps us to reach to convergent values. Setting it to 0.95 gave us result like exactly image above and now I will set it to 0.5 - meaning decrease epison drastically.



First of all it does not converge. Unlike the image shown where gamma is set to be 0.95.
The gamma is always set to be 0.95 to 0.99 and it helps to diminish the value of epsilon.

Loss Function:

I have choose Mean square error here basically (a standard version) which gives me decent results. The results are shown as above in the MSE LOSS after iteration section. The loss minimize itself near the values near to y=0.5 as shown above and helps to converge after 400 iteration considering other parameters left unchanged.

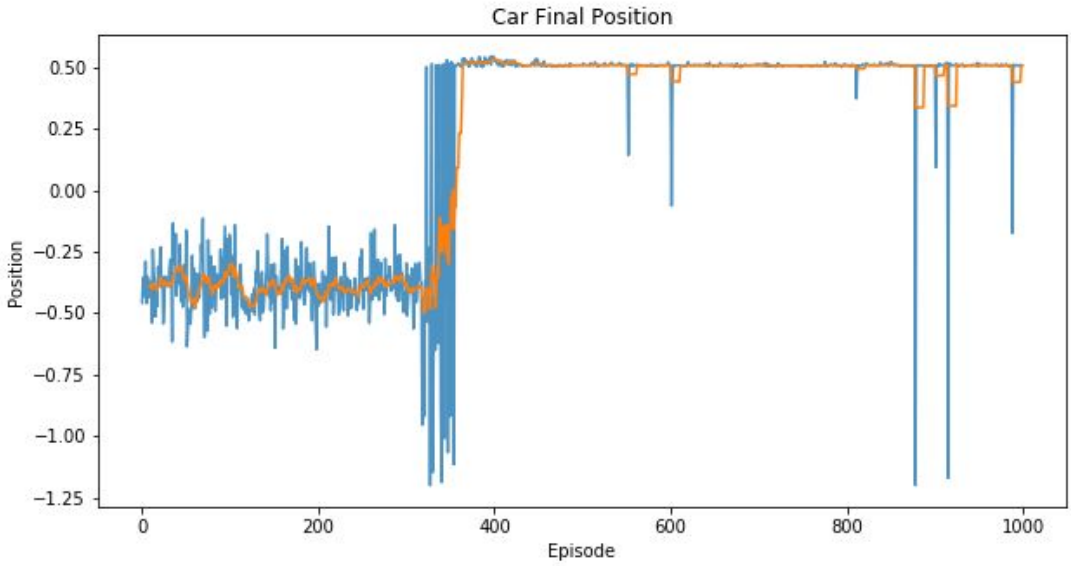
Termination Condition:

The basic termination condition is set to be the 200 iteration either or the goal state reached. On the other hand the no. of episode set is kept constant.(1000 to be exact). Meaning that a good policy will converge nicely at the end of last iterations.

At the termination we will have updated q values and this values will be trusted more as at this stage you exploit more and explore less.

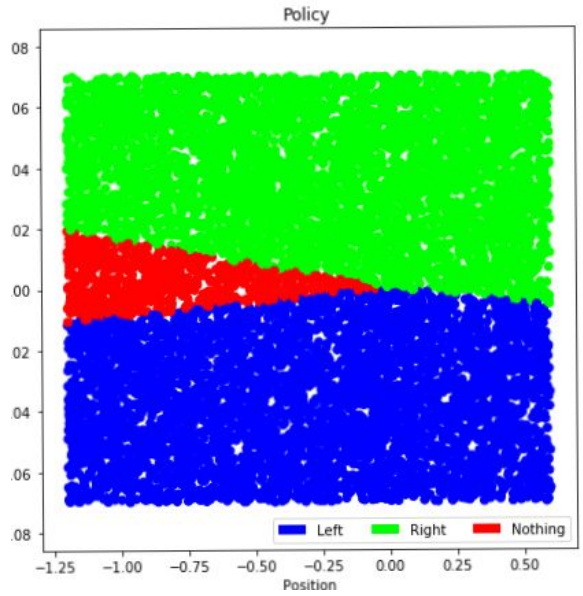
Evaluation of my Policy:

After setting hyper parameters as shown above my reward looks like the image given below:



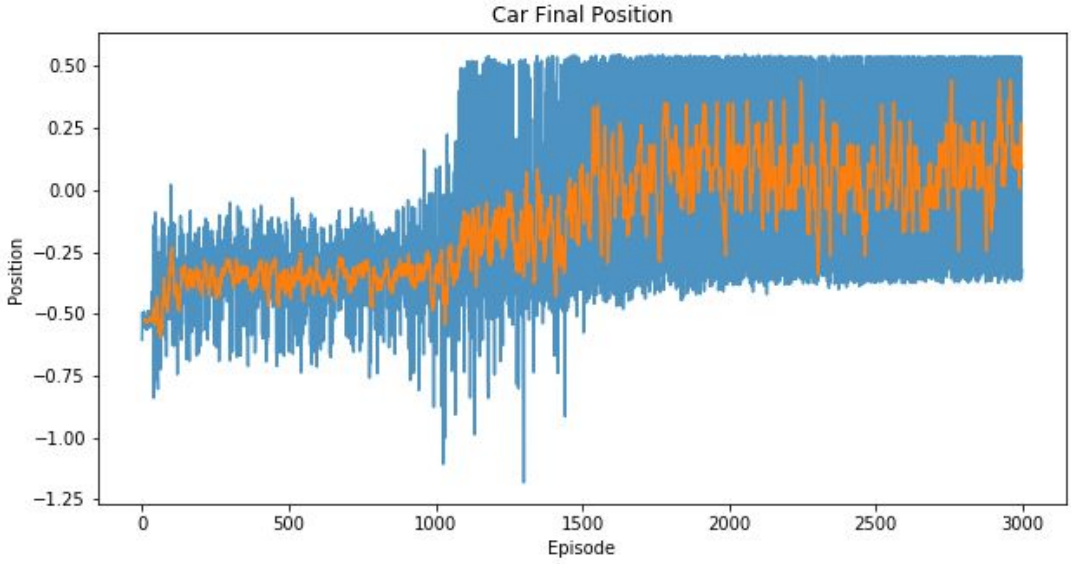
The orange line is actually the mean around that episode. (The average reward). As shown above uptill 300 episodes my policy juggle and struggle to actually reach to top and success rate until that point is 0 but after that updated q value will be trusted. And hopefully if this is good my policy will get to 0.5 position very frequently. The orange line after 400 is pretty consistent and almost reach the goal every time.

In the video submitted you can see the effect at last 5 animations. Where my cart escalates quickly and reach to the top. This is the result of getting trusted q values. The policy after this looks like this:

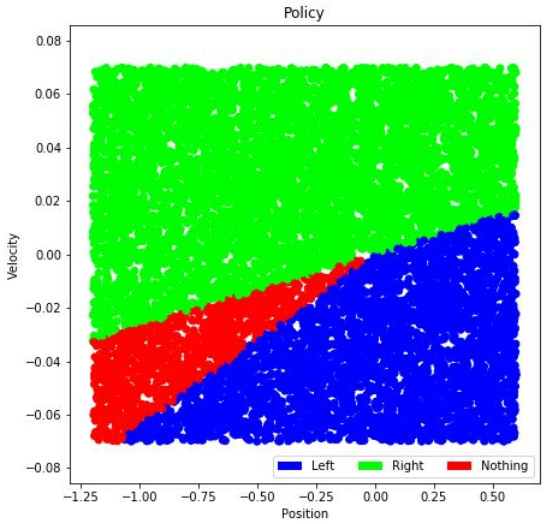


And this is the right balance, above cases where the changes have been frequent. The policy turns out to be always right push here its a balanced version and advice us which action to choose corresponding to state space. This is the correct balance between actions and optimal to reach to the yellow flag.

Now, if I turn off exploration, my policy only exploits and trust Q-table values in that case it looks like:



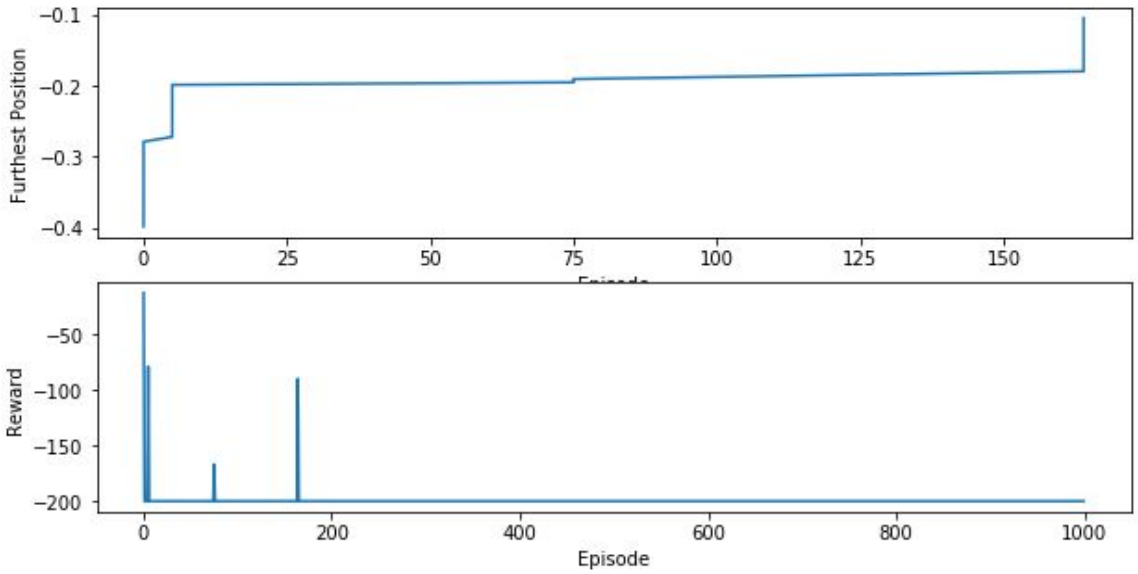
Again not as good as the best one. But it succeed in some of the episodes after all and the



policy it gives looks like:

Basically there are instances when reward fell below -50 which is fairly low(just a single spike as shown in figure below). This number is less than when reward fell under -100 and at last -175 the number is fairly large. But most of the time it is -200(meaning termination after 200 time stamps)
No. of times reward fell below -100:around 17
No. of times reward fell below -150:around 23
No. of times reward fell below -175:around 43

The reward given is like:



Conclusion:

Here, I have learnt to implement neural network to solve mountaincar v0 problem and used pytorch library to create neural net and evaluate it.

The high rendering video can be found in this link:

https://drive.google.com/drive/folders/1SsOowmvztXhsHN11PabFDWk6MCsVp21s?usp=sharing