Q Learning using Neural Network as Function Approximator

Environment – MountainCar-v0

The MountainCar-v0 environment is an openAI Gym environment where the goal is to move the cart to the goal position. The car must swing right and left before reaching the goal position.

State Space:

The state space for this problem is continuous with two attributes

1. "Position" with range [-1.2 to 0.6]
2. "Velocity" with range [-0.07 to 0.07]

Action space:

Action space is discrete with 3 actions

1. Push Left
2. No Push
3. Push Right

Since action space is continuous, vanilla version of Q learning which uses Q table implementation will not be sufficient since both state space and action space are to be discretized. If discretization is coarse, model will not learn the environment properly and on the other hand if discretization is fine, there will be a memory problem to hold the huge Q table. Hence Function approximation will be a better option. In this version we are using Neural Network as function approximator. Since the action space is discrete, value-based technique can be used were we learn Q values and use a greedy policy to estimate the action to be taken.

Design Process

Designing Neural Network

Layers

If we think of the problem intuitively, the problem can be solved with a simple if and else statement where if the velocity is positive, push right and if velocity is negative push left. To represent such a binary function, simple network with 4 neurons with 2 layers should be ideally enough (provided the velocity space is binary). Hence the network need not be deep or big. Also, simple feedforward network is enough. The problem is not complex enough to use other type of architecture such as resnet, convolution or recurrent networks

Layer	Number of Neurons	Activation	Comments
Input	2	-	Since there are 2 components in state space of environment
Hidden 1	64	RELU	RELU activation practically results in stable network with faster convergence
Hidden 2	64	RELU	
Output	3	Linear	The size of action space is 3

Other Parameters

Parameter	Type
Optimizer	ADAM
Learning Rate	0.002 - this can be considered as learning rate of Q value updates
Loss Function	Mean Squared Error – Since the problem is modelled as regression problem
Activation Functions	RELU and Linear
Dropout	Not used since network is shallow

The results of different NN architectures being tried are discussed below in Results section.

Value Based Deep Q Learning

In value-based learning the policy we use is greedy policy where we choose the action corresponding to largest Q value

$$V(s) = \max_{a \,\epsilon\, ActionSpae} Q(s,a)$$

The function Approximator learns $Q(s,a)$ for all states and actions. To do so we start with random values are and do Q updates by following equation

$$Q^{t+1}(s_t,a_t) = \begin{Bmatrix} R(s_t,a_t) & For\ terminal\ state \\ R(s_t,a_t) + \gamma \max_{a'} Q^t(s_{t+1},a\,;\theta) & For\ non\ terminal\ state \end{Bmatrix}$$

Where,

γ is the discount factor which weighs the future actions

Here we used a discount factor of 0.95 stating that future actions have high weightage

Reward Shaping

-1 for each time step, until the goal position of 0.5 is reached and there is no penalty for climbing left hill. This reward is sparse and only place where positive reward can be achieved is when goal position is reached. With this reward structure, large number of iterations are needed since the probability of reaching goal position with random action is very very low.

Instead if incremental reward is given, the agent learns effectively to collect immediate rewards and a large reward for reaching the goal position. Thus, below reward structure with some domain knowledge is used

1. If car is on left side of hill and the action results in climbing left hill, reward of +15 is given.
2. If car is on right side of hill and the action results in climbing right hill, reward of +15 is given
3. If Goal state is reached reward of +1000 is given
4. All other cases -10 is given

With the above reward structure, if the car climbs down left form right side of hill, continues climbing left side of hill and comes back to the same position, collectively a positive reward will be obtained.

Epsilon policy

To choose an action during training phase, epsilon greedy policy is used.

$$action = \begin{cases} \max_a Q(s,a) & with\ probability\ \epsilon \\ random\ action & witn\ probability\ (1-\epsilon) \end{cases}$$

This exploration vs exploitation is tackled.

Also ϵ value is decayed with a factor of 0.997 form initial value of 1.0 to minimum value of 0.01 is done on every step taken

$$\epsilon_{t+1} = \begin{cases} \beta\epsilon_t & if\ greter\ than\ 0.01 \\ \epsilon_t & if\ less\ than\ 0.001 \end{cases}$$

Where,

β is the decay factor. If decay factor of 1.0 is used, the exploration will be constant and will take more time to learn the policy.

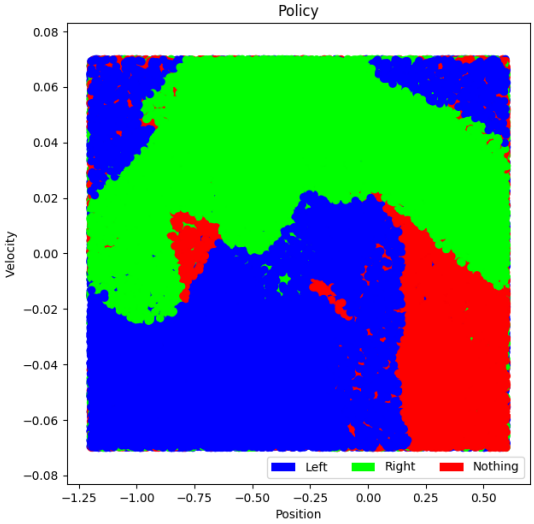
Training and action replay

As the actions are performed $< s_t, a_t, r, s_{t+1} >$ are captured as replay memory. Mini-batch of 128 random experiences is used for training the neural network.

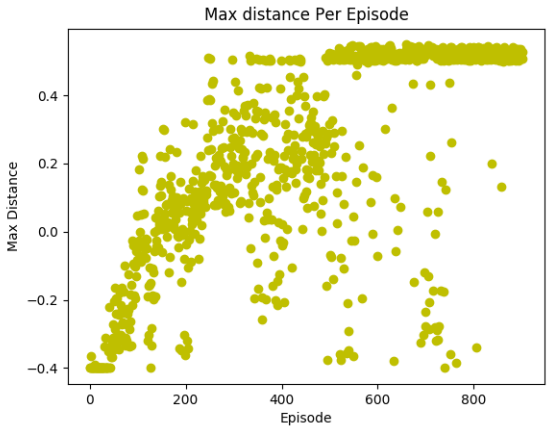
Inference

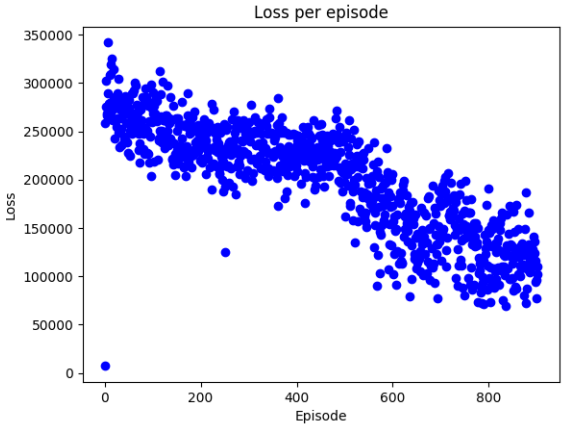
- When Large neural network is used, loss becomes small and the network tends to over fit due to which reward per episode doesn't increase.
- As expected, network learns the relation between position and velocity to take an action.
- When the agent is trained based on action and velocity alone instead of position, the reward decreased when goal state is reached.
- Even when reward is based on velocity, there is no segregation in terms of velocity but there is segregation in terms of position.
- Convergence of neural network depends on random initialization.
- If replay memory is cleared for every episode, the network doesn't learn the policy.
- By accumulating replay memory from start of training the network converges. This might be due to the fact that there is significant noise in data of singe episode and NN is not able to converge in presence of high noise.
- If epsilon is lesser policy is learnt in shorter time.
- Initially the loss is large even though NN has learnt the policy. But as we train the network with more and more iterations, the loss decreases.
- If Loss is constant and less, but result is not good, then this shows that NN is not able to learn with current reward structure or needs a lot more iterations.
- As reward function has more and more domain knowledge, the policy is learnt faster.

Training Result (Corresponding to best performance)



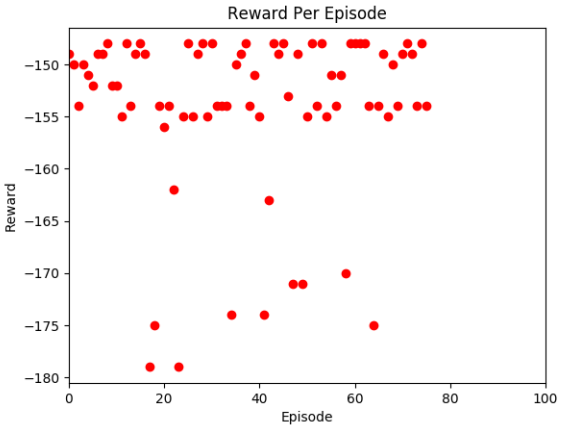
It can be seen that the learnt policy is segregated based on velocity than position.

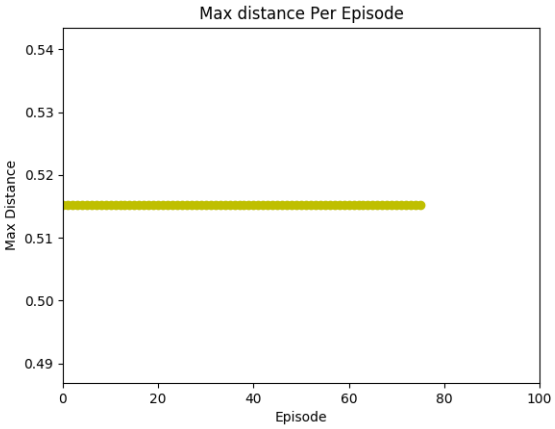


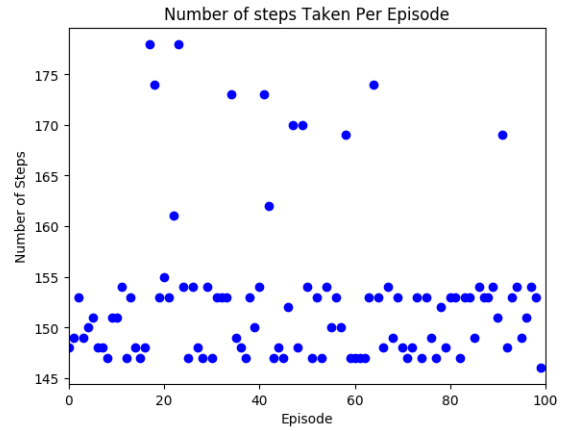


The reward increases initially and once the agent learns to goal the reward stays constant. Also, the loss per episode decreases showing that the policy is learnt but has variance. If more training is performed, then policy becomes robust.

Game Play Result (Exploitation Only)







Number of Episodes = 100

Success Rate = 100 %

Average number of steps taken = 152.86

The agent always reaches the goal position. In the reward structure we do not have higher reward of the car reaches goal with lesser number of steps. If this factor is included in reward structure, the number of steps taken to reach the goal can be decreased.

Environment – MountainCarContinous-v0

The MountainCarContinous-v0 environment is an openAI Gym environment where the goal is to move the cart to the goal position. The car must swing right and left before reaching the goal position.

State Space:

The state space for this problem is continuous with two attributes

1. "Position" with range [-1.2 to 0.6]
2. "Velocity" with range [-0.07 to 0.07]

Action space:

Action space is continuous "Velocity" with range [-1, 1]

Since action space is also continuous, value-based learning is not the ideal candidate and policy-based methods can be used. But we can discretize the action space and use value-based learning. This continuous version can be tackled the same way discrete version is solved.

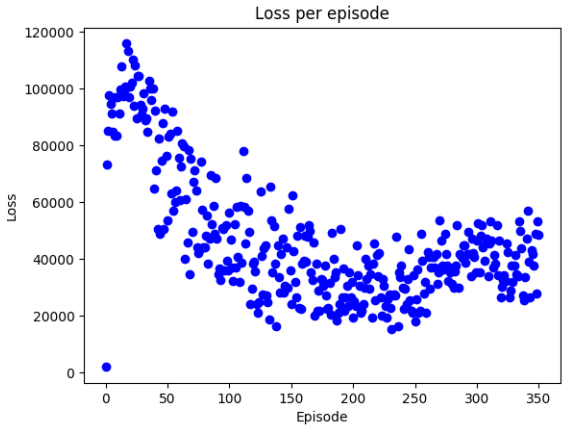
Action space is discretized into 10 divisions of equal range between -1 and 1.

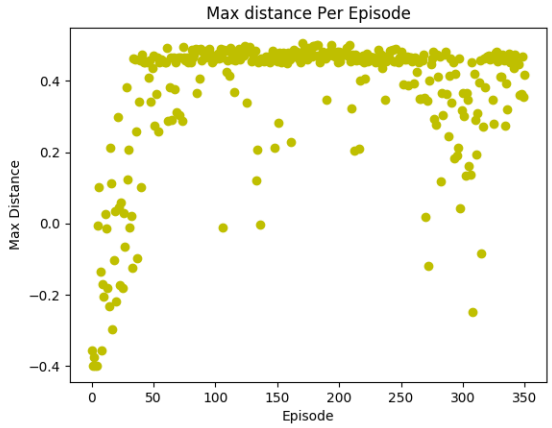
Neural Network Design

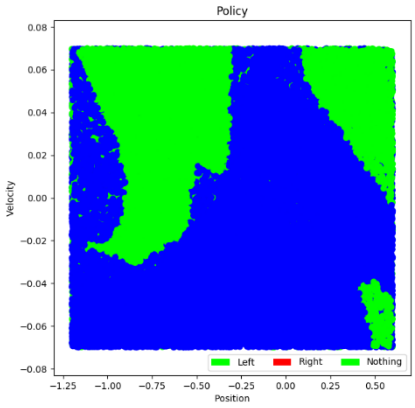
Layer	Number of Neurons	Activation	Comments
Input	2	-	Since there are 2 components in state space of environment
Hidden 1	64	RELU	RELU activation practically results in stable network with faster convergence
Hidden 2	64	RELU	
Output	10	Linear	The action space is discretized to 10 units

Other than number of units in output layer, same setting and same hyper parameters used in discrete version is retained.

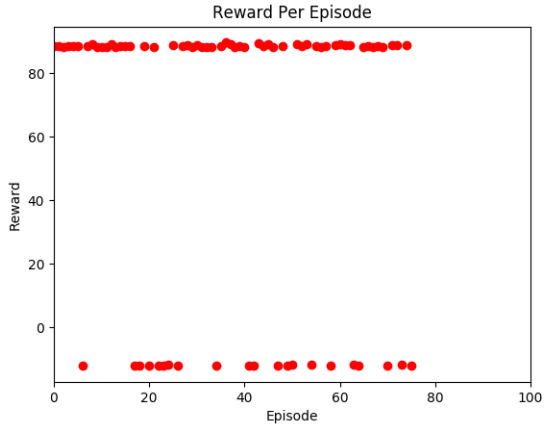
Training Result

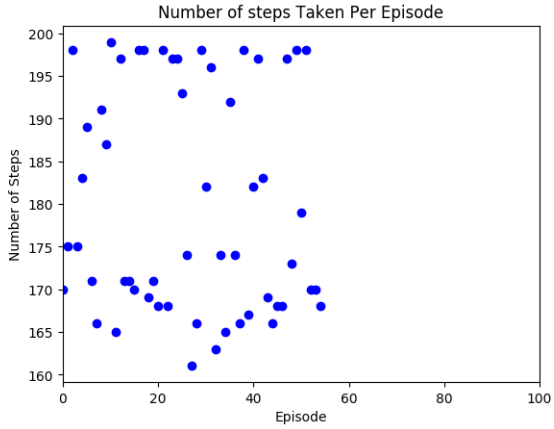


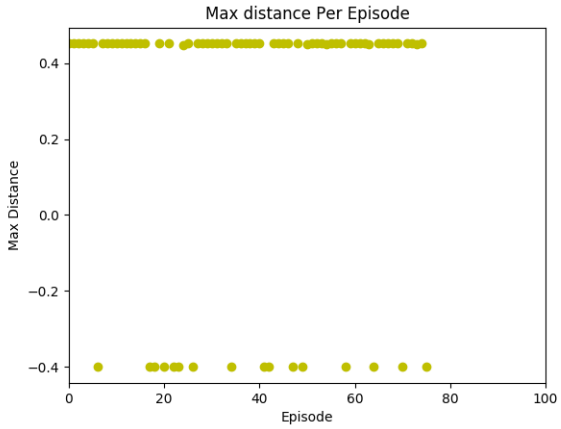




Game Play Result (Exploitation Only)







Number of Episodes = 100

Success Rate = 75 %

Average number of steps taken = 181.28

It can be observed from training loss that it decreases. So the policy is being learnt and with more training success rate can be increased.

References:

1. https://medium.com/coinmonks/solving-curious-case-of-mountaincar-reward-problem-using-openai-gym-keras-tensorflow-in-python-d031c471b346
2. https://towardsdatascience.com/getting-started-with-reinforcement-learning-and-open-ai-gym-c289aca874f
3. https://medium.com/@ts1829/solving-mountain-car-with-q-learning-b77bf71b1de2

Appendix

Results for different training methods used (MountainCar-v0)

Reward 1

#==

 # # Reward 1

 # if nextState[0] >= 0.5 or nextState[0] > episodeMaxDist:

 # reward += 5

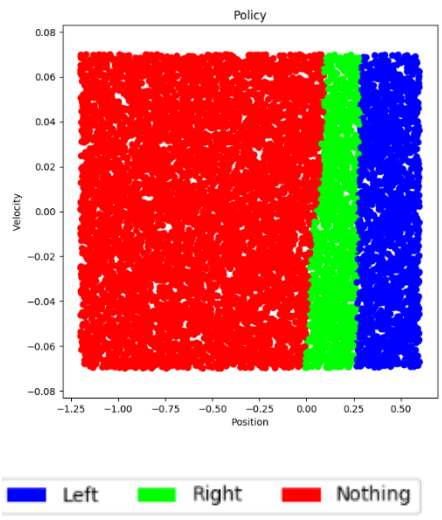
 # else:

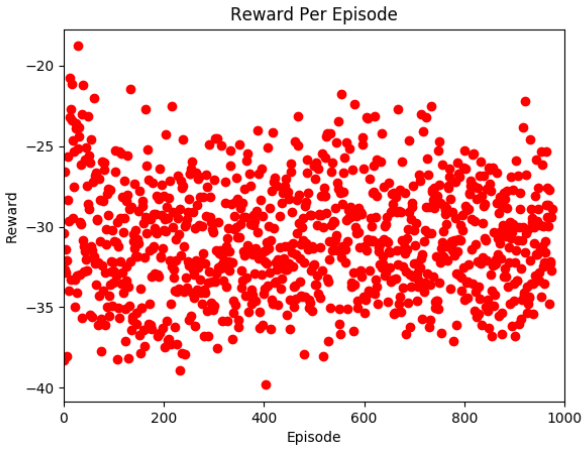
 # reward = nextState[0] + 0.5

 # ==

NN Architecture 1 with Reward 1

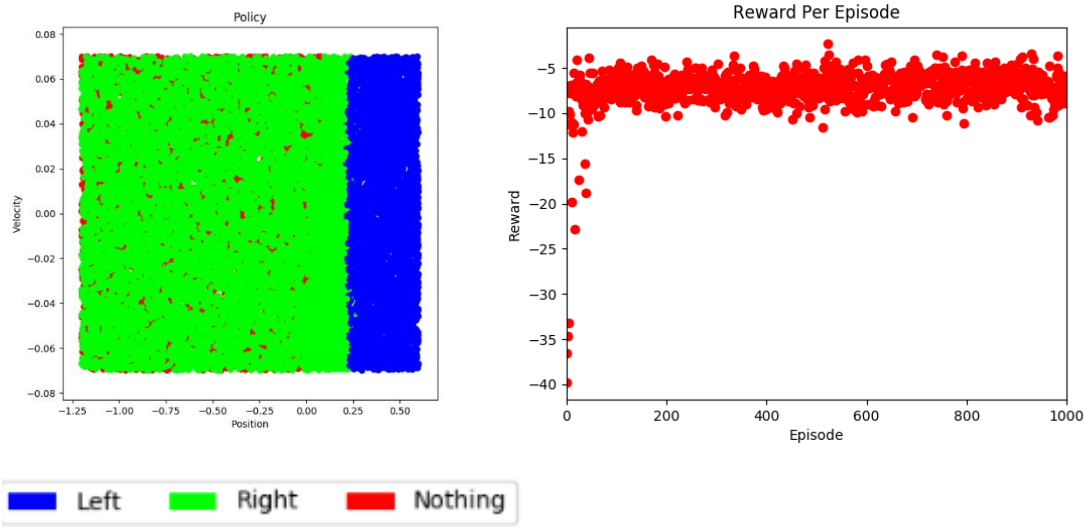
2 * 50 * 3





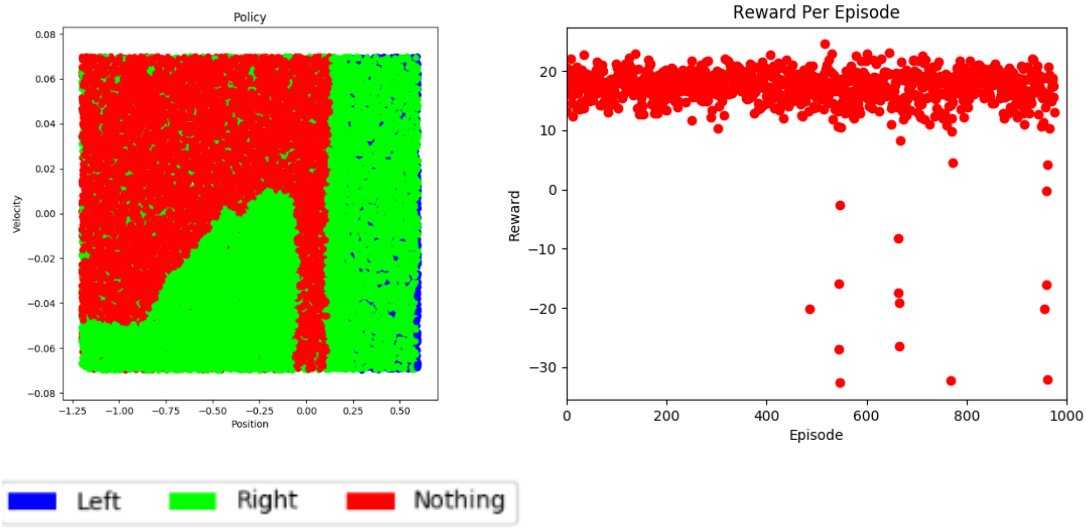
NN Architecture 2 with Reward 1

2 * 128 * 52 * 3



NN Architecture 3 with Reward 1

2 * 34 * 31 * 21 * 19 * 10 * 4 * 3



Reward 2 with NN Architecture 1

 # ==========

 # # Reward 4

 # sign = np.array([-1.0,0.0,1.0])

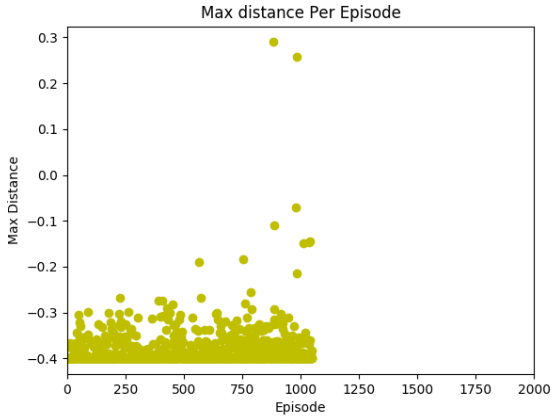
 # if nextState[1]*sign[action] >= 0:

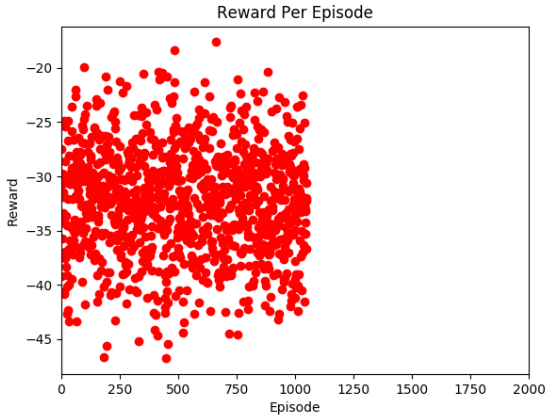
 # reward = nextState[0] + 0.5

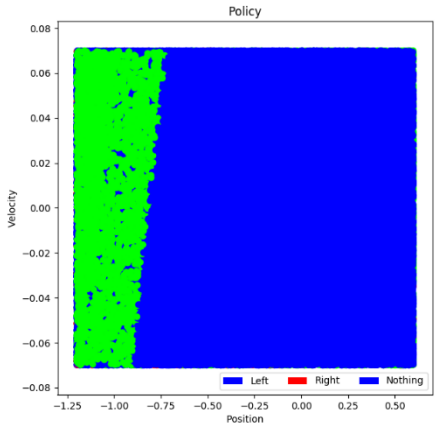
 # else:

 # reward = nextState[0] - 0.5

 # ==========







Reward 3 with NN Architecture 1

 # ==========

 # # Reward 5

 # sign = np.array([-1.0,0.0,1.0])

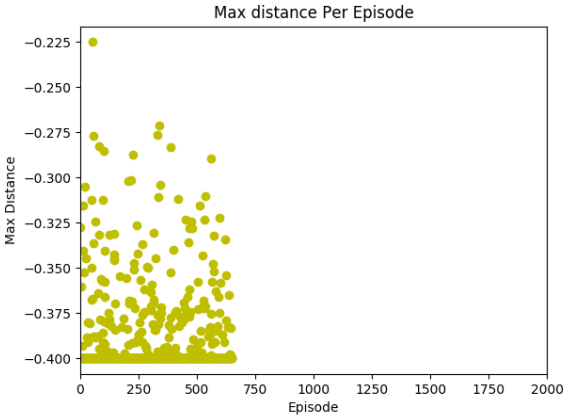
 # if currState[1]*sign[action] >= 0:

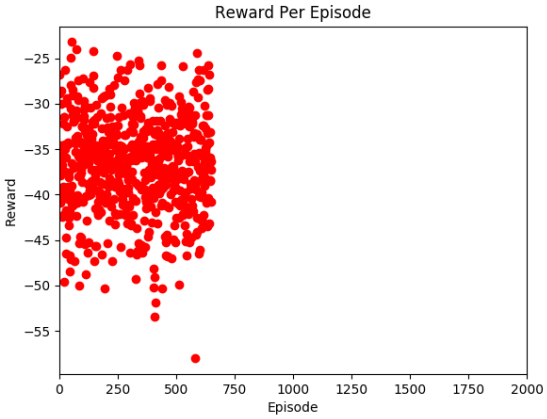
 # reward = nextState[0] + 0.5

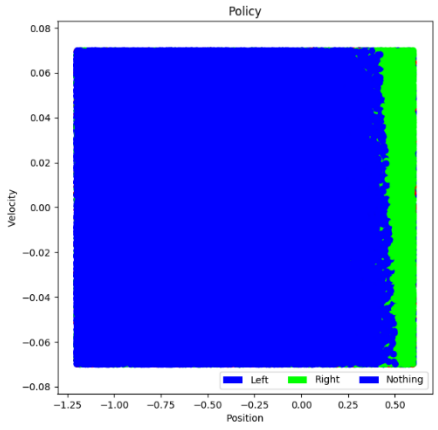
 # else:

 # reward = nextState[0] - 0.5

 # ==========







Reward 4 with NN Architecture 1

 # ==========

 # # Reward 6

 # sign = np.array([-1.0,0.0,1.0])

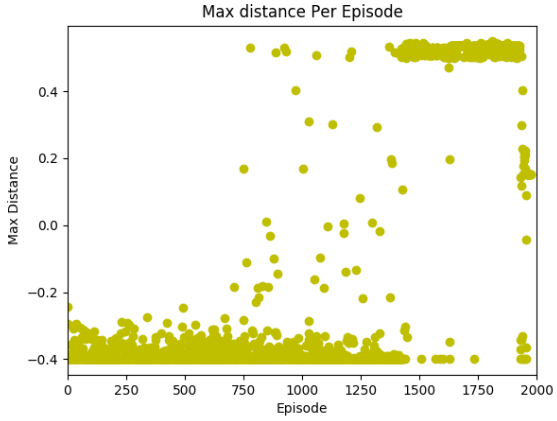
 # if currState[1]*sign[action] >= 0:

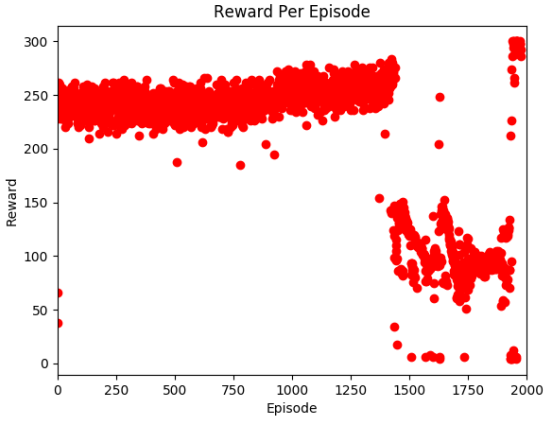
 # reward = 1

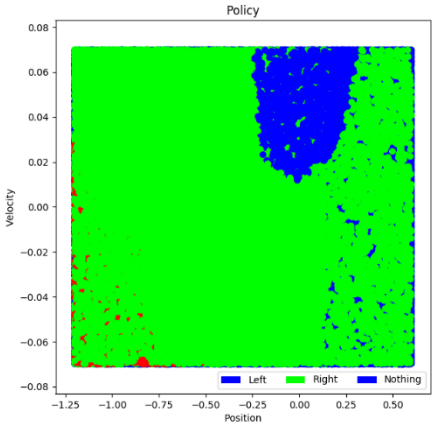
 # else:

 # reward = -1

 # ==========







Reward 5 with NN Architecture 1

 # ==========

 # # Reward 7

 # sign = np.array([-1.0,0.0,1.0])

 # if currState[1]*sign[action] >= 0:

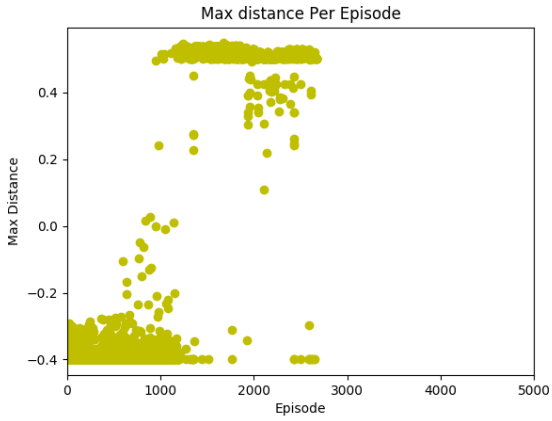
 # reward = 1

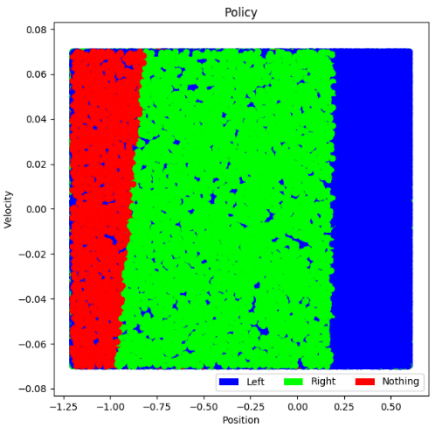
 # else:

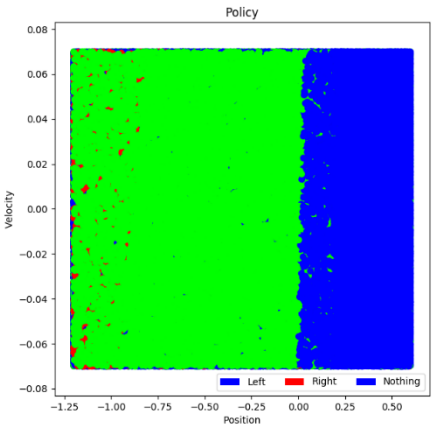
 # reward = -1

 # reward = (0.999step) * reward**

 # ==========







Reward 5 with NN Architecture 3

NN Architecture – 2*64*64*3

 # ==

 # # Reward 10

 # sign = np.array([-1.0,0.0,1.0])

 # if currState[1]*sign[action] >= 0:

 # reward = 1

 # else:

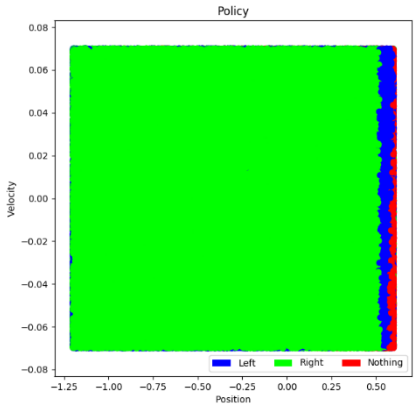
 # reward = -1

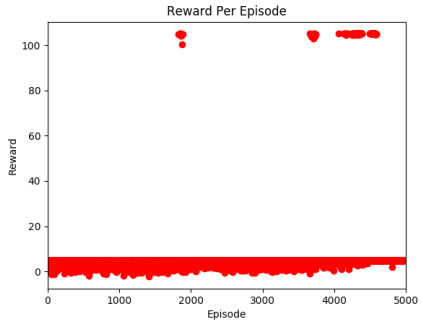
 # reward = (0.8step) * reward**

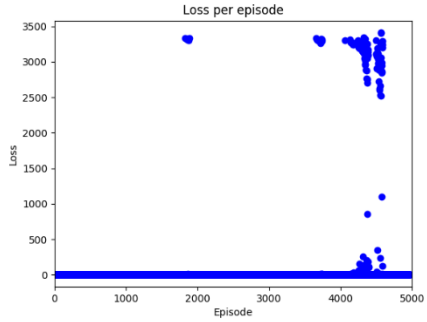
 # if nextState[0] >=0.5:

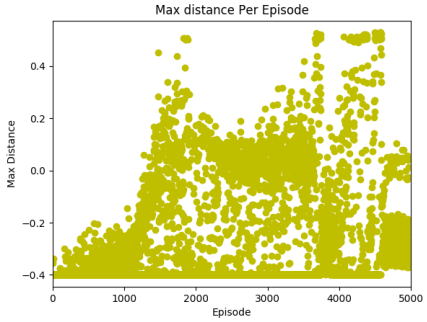
 # reward+= 100

 # ==









Reward 6 with NN Architecture 3

NN Architecture – 2*64*64*3

 # ==

 # # Reward 11

 # if nextState[1] > currState[1] and nextState[1]>0 and currState[1]>0:

 # reward += 15

 # elif nextState[1] < currState[1] and nextState[1]<=0 and currState[1]<=0:

 # reward +=15

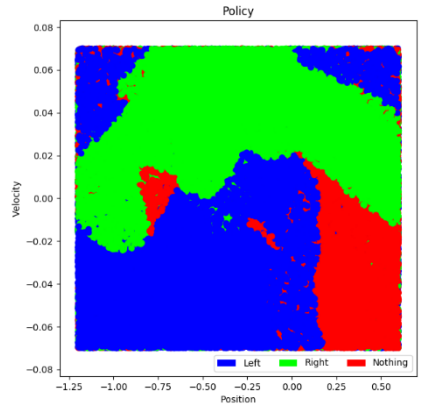
 # if done:

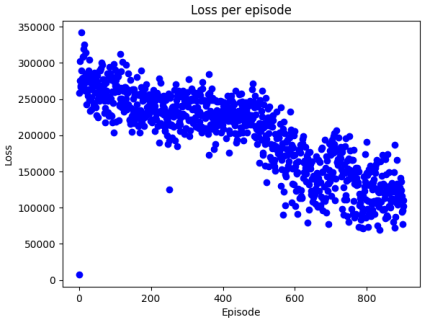
 # reward = reward + 1000

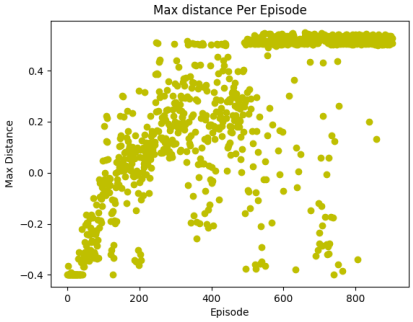
 # else:

 # reward=reward-10

 # ==







Reward 7 with NN Architecture 3

NN Architecture – 2*64*64*3

 # ==

 # # Reward 15

 # if nextState[1] > currState[1] and nextState[1]>0 and currState[1]>0:

 # reward += 15

 # elif nextState[1] < currState[1] and nextState[1]<=0 and currState[1]<=0:

 # reward +=15

 # if done:

 # reward = reward + 1000

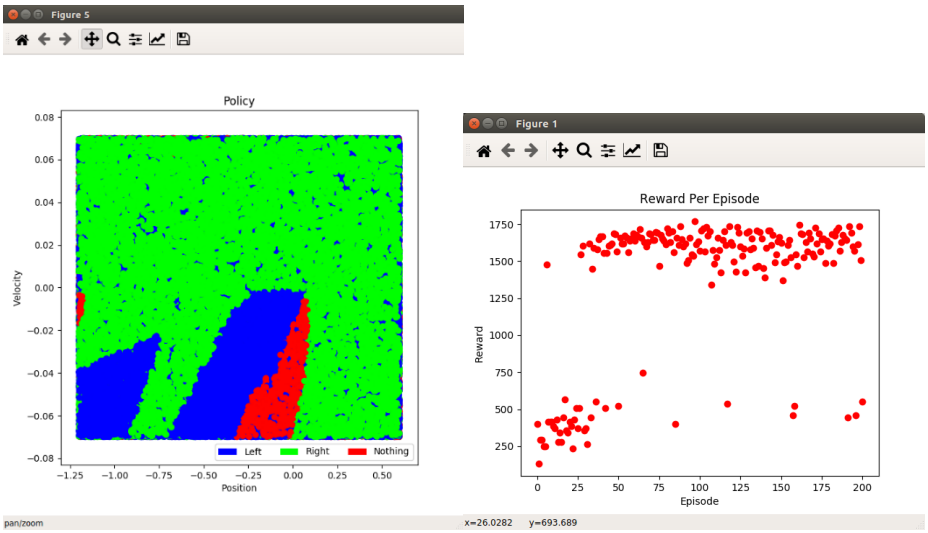
 # else:

 # reward=reward-10

 # if nextState[0]>= 0.5:

 # reward += 1000

 # ==



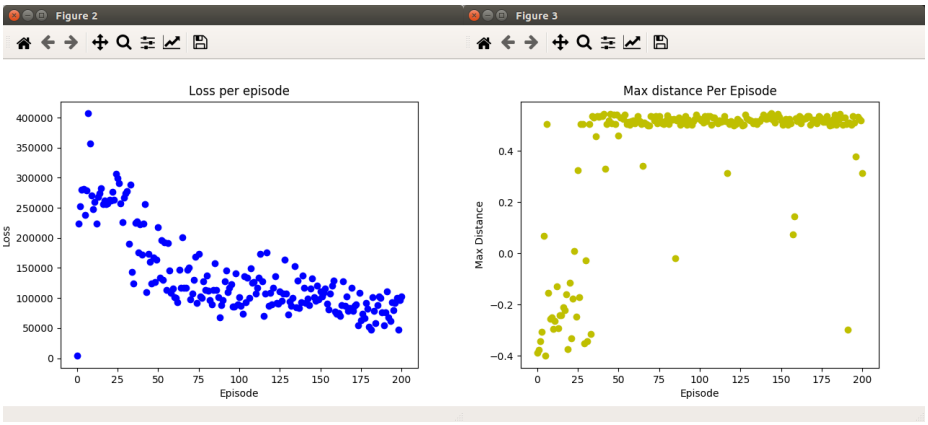


Figure 2
Loss per episode
Loss
Episode
0
50000
100000
150000
200000
250000
300000
350000
400000
0
25
50
75
100
125
150
175
200
Figure 3
Max distance Per Episode
Max Distance
Episode
−0.4
−0.2
0.0
0.2
0.4
0
25
50
75
100
125
150
175
200