Reinforcement Learning

Final Project

Submitted in partial fulfillment of the requirements of
BITS F464T Machine Learning

By

Ayush Mall : 2016A3TS0163G
Lavanay Thakral : 2016B5A7PS0566G
Shreyas Patel : 2016B2A7PS0549G

Under the supervision of:

Ashwin Srinivasan
&
Tirthraj Dash

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, GOA CAMPUS
April 2020

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, GOA CAMPUS

Abstract

Reinforcement Learning

by Ayush Mall : 2016A3TS0163G
Lavanay Thakral : 2016B5A7PS0566G
Shreyas Patel : 2016B2A7PS0549G

This project tries to solve the problem of balancing a Cart-Pole in a custom environment, with added noise and friction. We will discuss the various methodologies and their variations tried in order to get the best results. It also tries to explain the reader the basics of Reinforcement Learning, laid down in almost layman's term.

Contents

Chapter 1

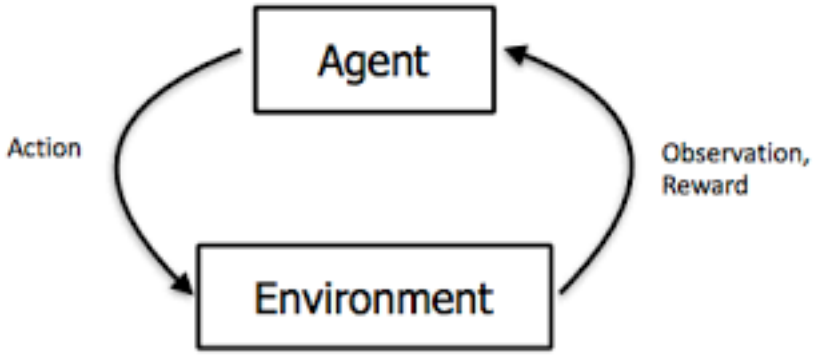
Reinforcement Learning Models

1.1 Basics of Reinforcement Learning

In very simple terms, Reinforcement Learning is learning how to map situations to actions- so as to maximize a numerical reward. The learner isn't told what to do, but instead it learns from the action it takes in different situations. In most cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics : trial-and-error search and delayed reward—are the two most important distinguishing features of reinforcement learning.

A simple example for reinforcement learning would be a person trying to learn how to ride a bicycle. Learning to ride a bicycle cannot be mastered by memorising a particular set of rules from a rule book and implementing them to perfection. It can be only learned if you ride the bicycle multiple times and fall. In doing so, you figure out what action works well in which situation. If your cycle is falling to the left, shifting your center of mass to the right would probably help you ride for longer. The reward system here would simply be how long you are able to ride your bicycle.

All such examples share some very basic features. All involve an interaction between an active decision-making agent and its environment, within which the agent seeks to achieve a goal despite uncertainty about its environment. The agent's actions are permitted to affect the future state of the environment. Also, the effects of actions cannot be fully predicted; thus the agent must monitor its environment frequently and react appropriately[9].



1.2 Elements of Reinforcement Learning

There are four main elements of a learning system: a policy, a reward signal, a value-function and a Q-function. A *policy* is a mapping from the known states of the environment to the actions to be taken when in those states. This defines the general behavior of the learning agent. In the start of a problem, the agent has a random policy and over time it improves its policy to maximize rewards. These could be modelled as simple look-up tables for a finite state environment or a simple function for a continuous state environment.

$$Policy - pi(s) : S \Rightarrow A$$

A *reward signal* defines the goal in a reinforcement learning problem. At each stage, the environment sends to the learning agent, a single number called the reward. The agent's sole objective is to maximize the total reward it receives in the long run. Reward is the metric used to judge a policy. If an taking an action in a particular state leads to a low total reward in long term, then the policy for that state should be changed to taking an action that yields a higher total reward. The total reward is discounted to give more important to recent awards.

$$total\,discounted\,reward = \sum_{i=1}^{\infty} \gamma^{i-1} r_i$$

where γ is the discount factor in [0,1] and r_i is the reward that it gets i-1 steps from the current state.

A value function approximates the total amount of reward an agent can accumulate over the future, starting from that state. Since reward is for immediate step, and doesn't give an idea about how good an action is in a long run, the value function are used while making and evaluating policies. Unfortunately, value functions are a lot more difficult to evaluate than rewards, as rewards as given directly by the environment, but values depend on every state that

it goes to after its present state. This forms the most important component of Reinforcement Learning, to find the most efficient estimate for values. A value function following a given policy as given as:

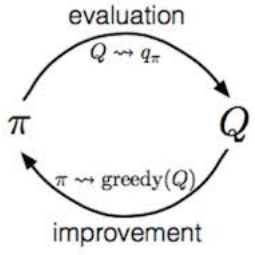
$$V^{\pi}(s) = E[\sum_{i=1}^{\infty} \gamma^{i-1} r_i] \, , \forall s \in S$$

Q-function is a state-action pair, and returns a real value. $Q^{\pi}(s, a)$ would give the total reward expected to be received by the agent starting in s and picking an action$a$. value function and Q-function can be related such:

$$V^{\pi}(s) = \max_{a} Q^{\pi}(s, a) \, , \forall s \in S$$

1.3 Model-free Prediction

Model-free prediction is predicting the value function of a certain policy without a concrete model. This is different from the case of model-based prediction, where all states are known, in which case a Markov Decision Problem(MDP) can be established and can be solved using the recursive equation given by the Bellman equation. Since our problem of a Cart-pole system is that of model-free prediction, we would refrain from discussing the model-based prediction.
Model-free prediction can be broken down into two parts. One is called policy evaluation and the other is called policy improvement.
Policy Evaluation can be explained as choosing the right action to be taken based on the present policy. This decision is based on the estimated Q- function. The biggest challenge in this step is the approximation of the Q- function, which will be discussed later.
The other part is called *Policy Improvement* and here the policy is changed such that in a greedy fashion, the action that gives the highest total reward, is chosen. Both these steps go hand in hand, as shown in the figure below:



1.3.1 Monte-Carlo Learning

Monte-Carlo (MC) method learns directly from episodes of experience. It must run at the end of an episode, and it takes in account actual reward from each state till the end of the episode.

Here lies its drawback, that it can be only implemented for problems where episodes always must terminate. For a policy π , $V^{\pi}(s)$ can be estimated as:

$$Q^{\pi}(s,a) = E_{\pi}[G_t|S_t = s]$$

$$where,\ G_t = R_{t+1} + \gamma R_{t+2} + ...\gamma^{T-1} R_T$$

This expectation of the value function can be approximated as a moving average :

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(G_t - Q(S_t, A_t))$$

1.3.2 Temporal-Difference Learning

Temporal-Difference method learns from direct-interaction with the environment. It can learn from incomplete episodes, unlike the MC method, using the help of *bootstrapping*(guessing the value function for the next state). The total reward and consequently the value function here is calculated differently as shown below:

$$G_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(G_t - Q(S_t, A_t))$$

1.3.3 Approximate Value Function Learning

For finite-state environments, value functions can be stored as a look-up tables, but for environment with continuous states, a look-up table would be impossible to make. Instead, a function is used, which can give out value functions for each state and also learn from the states that it has seen. Usually, a Deep Neural Network is used, although any SVM or Linear Regression can be used too.

1.3.4 Epsilon-Greedy Algorithm

It is possible that, if we blindly follow a policy, we might reach a local maxi-ma and would never move from their. Therefore it is important to add some randomness in the policy-evaluation step. In every step, the an random action is taken with a probability of ϵ. This is process is called *Exploration and Evaluation.*

1.4 Models Used

With the background, it would be easier to explain the models used in this report.

1.4.1 Deep Q-learning

Q-learning is an off-policy TD policy. The off-policy denotes that ,to calculate G_t, A_{t+1} is calculated greedily, instead of using the policy. That is,

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \underset{a}{max} Q(S_{t+1}, a) - Q(S_t, A_t)]$$

The word 'Deep' denotes the use of Neural Networks to approximately estimate the value of Q-function.
Another feature used in the program is that of *experience replay*. Each $< S_t, a_t, r_t, S_{t+1} >$ are stored in a memory; and after each step, randomly a batch of k is selected from the memory and the neural network is trained on it [7].

1.4.2 Policy Gradient Learning

Unlike Q-learning, instead of estimating the Q-function for each state-action pair to make a greedy , here the policy is improved. Let $\pi_{\theta_t}(a|s)$ denote the probability of taking an action a in state s according to the policy π , where θ denotes the weights of the neural network. The goal is to come to a reach optimum π^* through following updates to the weights [8]:

$$\theta_{t+1} = \theta_t + \alpha \Delta \pi_{\theta_t}(a^*|s)$$

1.4.3 Actor Critic Learning

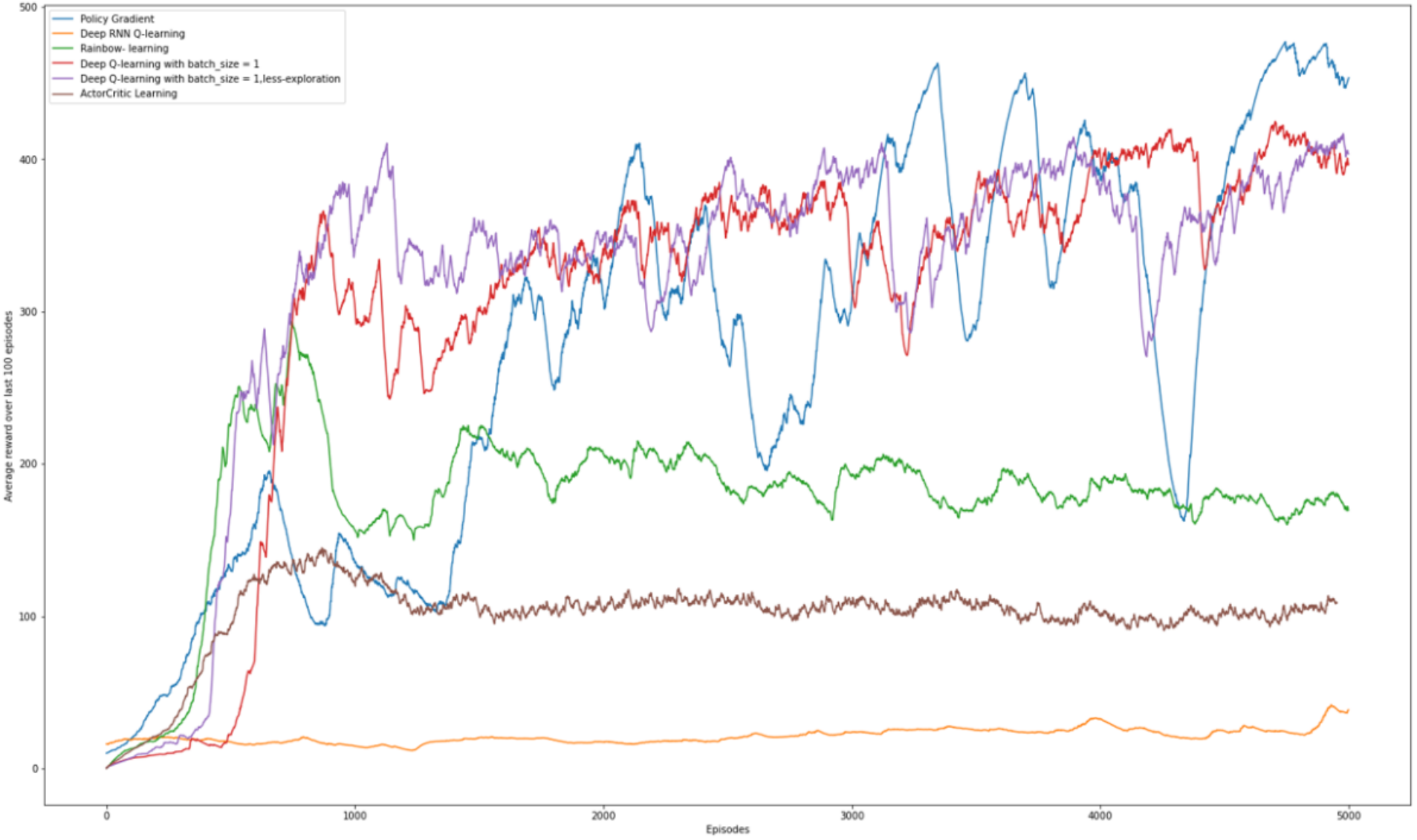
There are two functions *Actor* and *Critic* parameterized with neural networks. The *Critic* estimates the value functions. This could either be the state-value function(V) or the action value function(Q). The *Actor* updates the policy distribution in the direction suggested by the Critic (like in Policy Gradient). It is a mix of Q-learning and Policy Gradient [6].

1.4.4 Rainbow Learning

This state-of-the-art algorithm was developed by leading Reinforcement Learning pioneers in DeepMind for Atari 2600 benchmark. It incorporates a lot of different approaches. The explanation is outside the scope of this report. [5]

1.5 Performance of Models

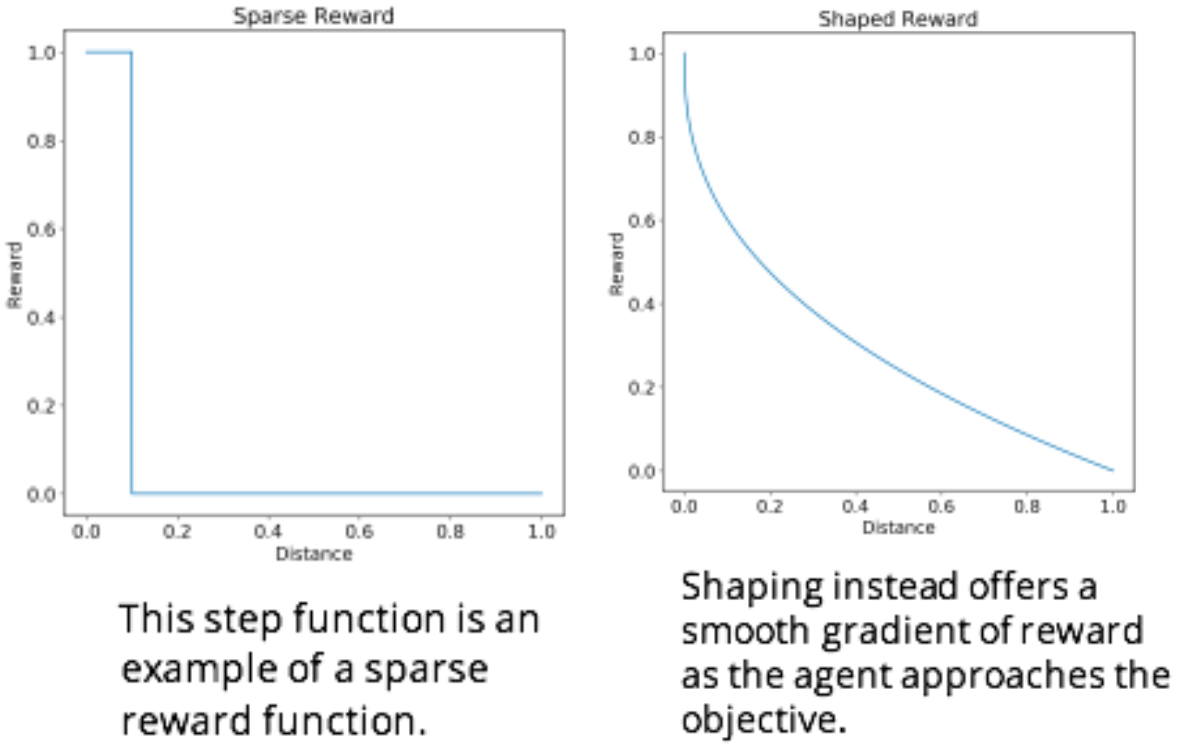
In total, 6 variations of the models were run for 5000 episodes. Three of them were the Deep Q-learning with different batch sizes and minimum epsilon values. As observed from the graph, policy gradient and Deep Q learning have performed the best. Rainbow learning and Actor-critic learning although very complex, don't work so well in our case. Deep RNN Q-learning is Q-learning with a Recurrent neural network. It performed very poorly in this case. Due to Deep Q-learning's smaller variance in reward, we went ahead with it despite Policy-Gradient settling on a slightly higher value.The graph with the moving average over 100 episodes plotted against the episodes for different models is given below:


Policy Gradient
Deep RNN Q-learning
Rainbow- learning
Deep Q-learning with batch_size = 1
Deep Q-learning with batch_size = 1,less-exploration
ActorCritic Learning
Average reward over last 100 episodes
Episodes
500
400
300
200
100
0
0
1000
2000
3000
4000
5000

Chapter 2

Reward Functions

2.1 Shape of the reward functions:



The default reward awarded for cartpole-v1 is of +1 is provided for every time step that the pole remains upright. The primary problem with this reward function is that it gives very little feedback about the degree to which the decision made by the model was right or wrong. We propose two reward function to improve learning [1] :

2.1.1 Linear Reward function

The linear reward function is based on the basis that the pole is more likely to be balanced for longer if the pole is more vertical and the cart is more towards the centre of the environment. The reward function takes both of the factors into consideration. The reward function is given by :

$$r_v = \frac{v_{threshold} - |v|}{v_{threshold}} - i_v$$

$$v : state\,variable$$
$$v_{threshold} : maximum\,value\,of\,state\,variables$$
$$i_v : [0.2, 0.4, 0.6, 0.8]$$
$$reward = r_x + r_\theta$$
$$x : x - coordinate\,of\,cart$$
$$\theta : angle\,of\,pole$$

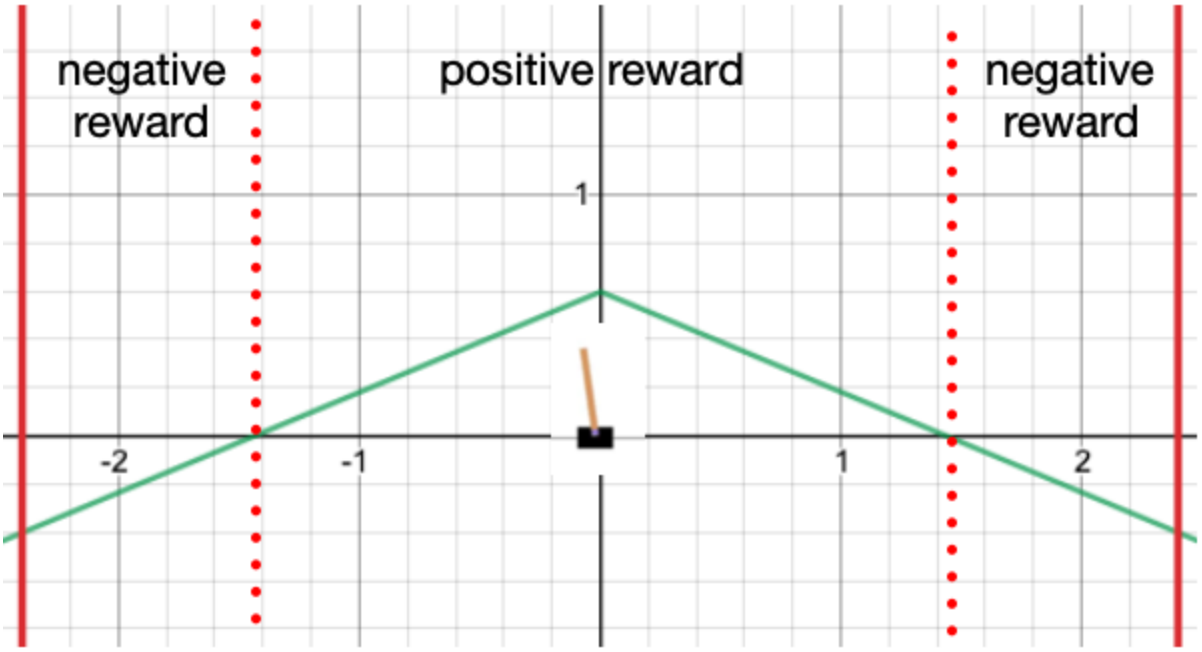


FIGURE 2.2: r_x function with $i_v = 0.4$

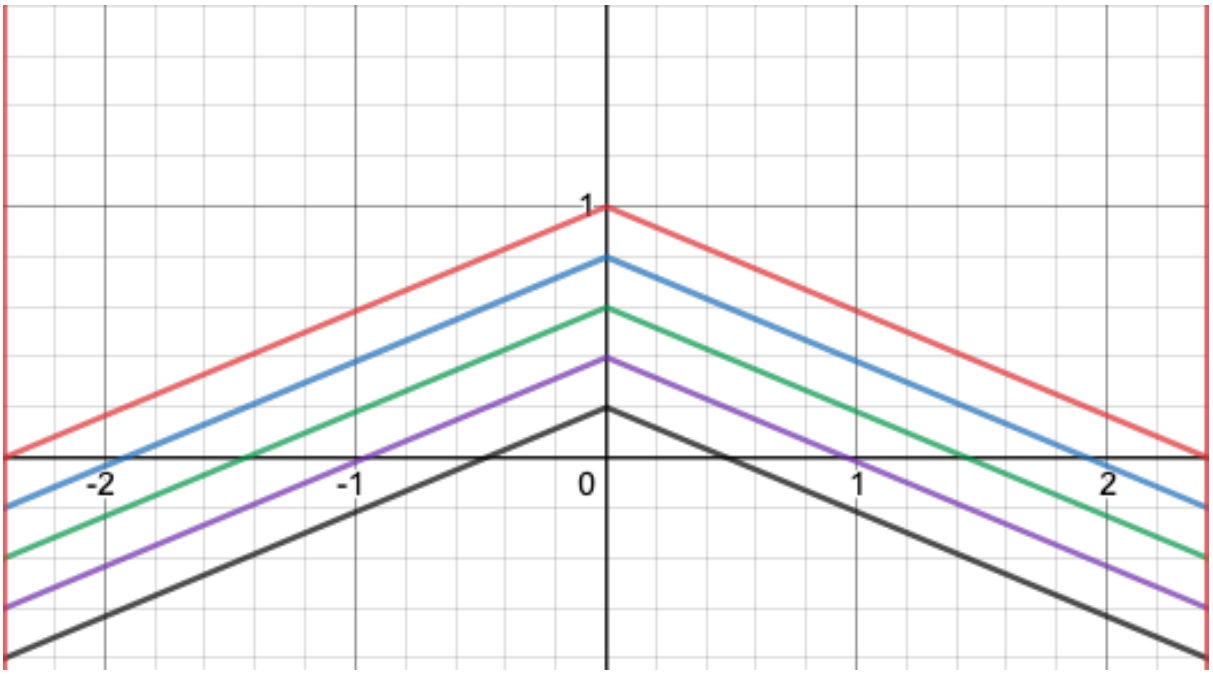


FIGURE 2.3: hyper parameters for r_x (i_v = 0.2, 0.4, 0.6, 0.8)

2.1.2 Quadratic Reward function

The quadratic reward function is made on the same principle of the linear reward function except that it provides a smoother feedback and punishes the extremes of the environment more severely. The reward function is given by :

$$r_v = \frac{-v^2}{v_{threshold}^2} * \frac{1}{i_v^2} + 1$$

$$v : state\,variable$$
$$v_{threshold} : maximum\,value\,of\,state\,variables$$
$$i_v : [0.2, 0.4, 0.6, 0.8]$$
$$reward = r_x + r_\theta$$
$$x : x - coordinate\,of\,cart$$
$$\theta : angle\,of\,pole$$

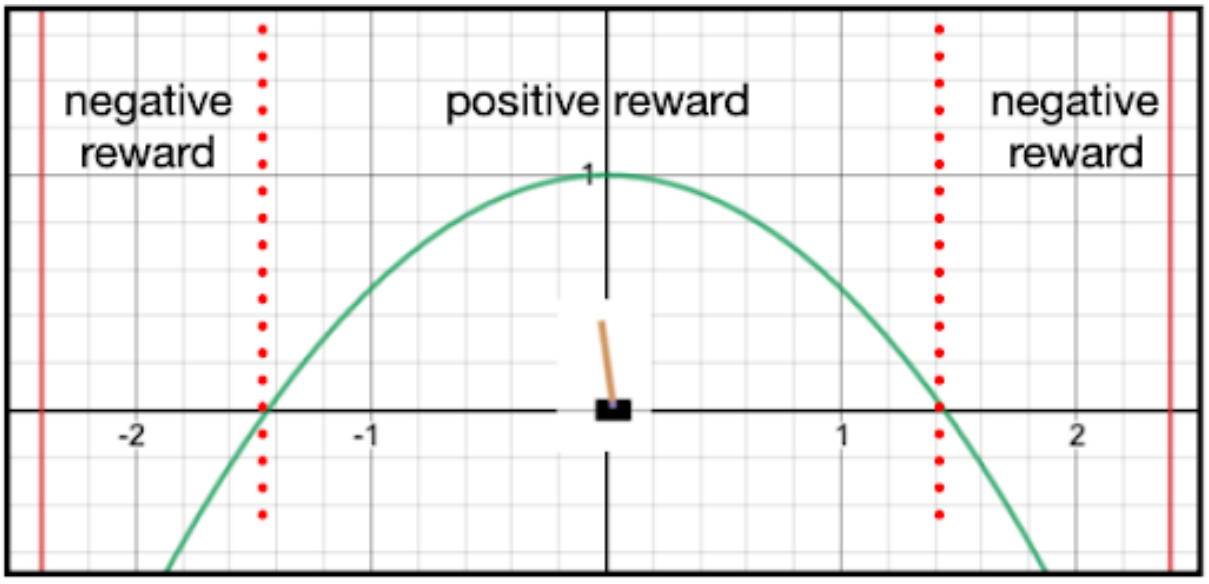


FIGURE 2.4: r_x function with $i_v = 0.4$

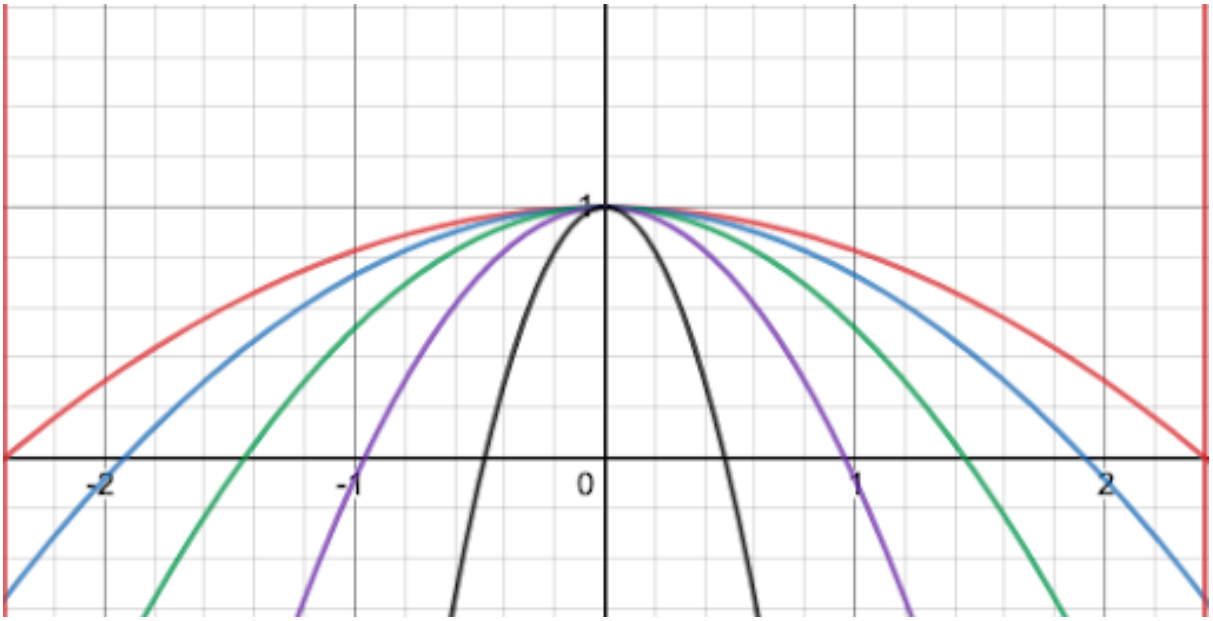
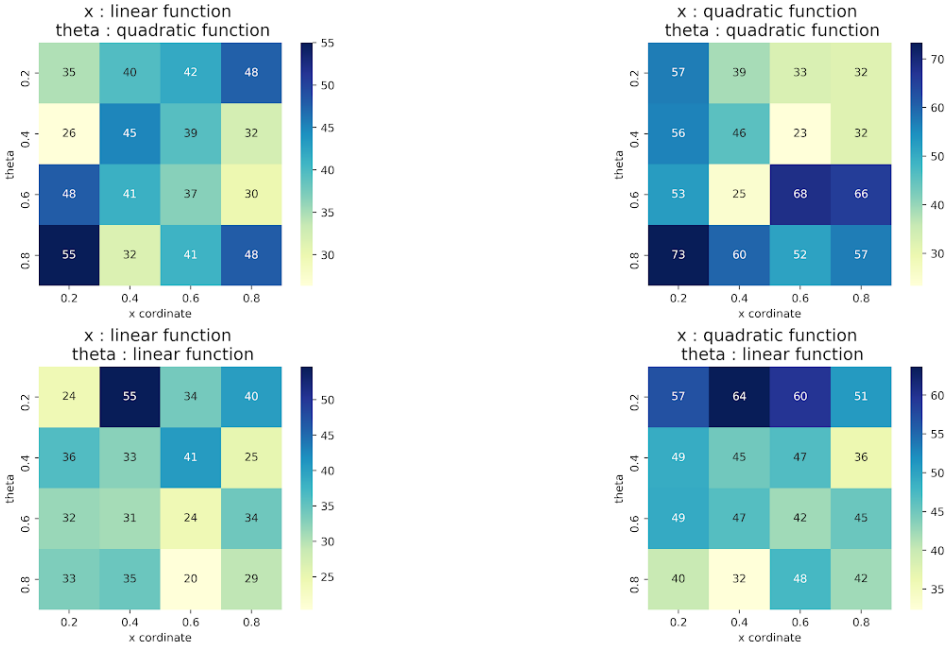


FIGURE 2.5: hyper parameters for r_x ($i_v = 0.2, 0.4, 0.6, 0.8$)

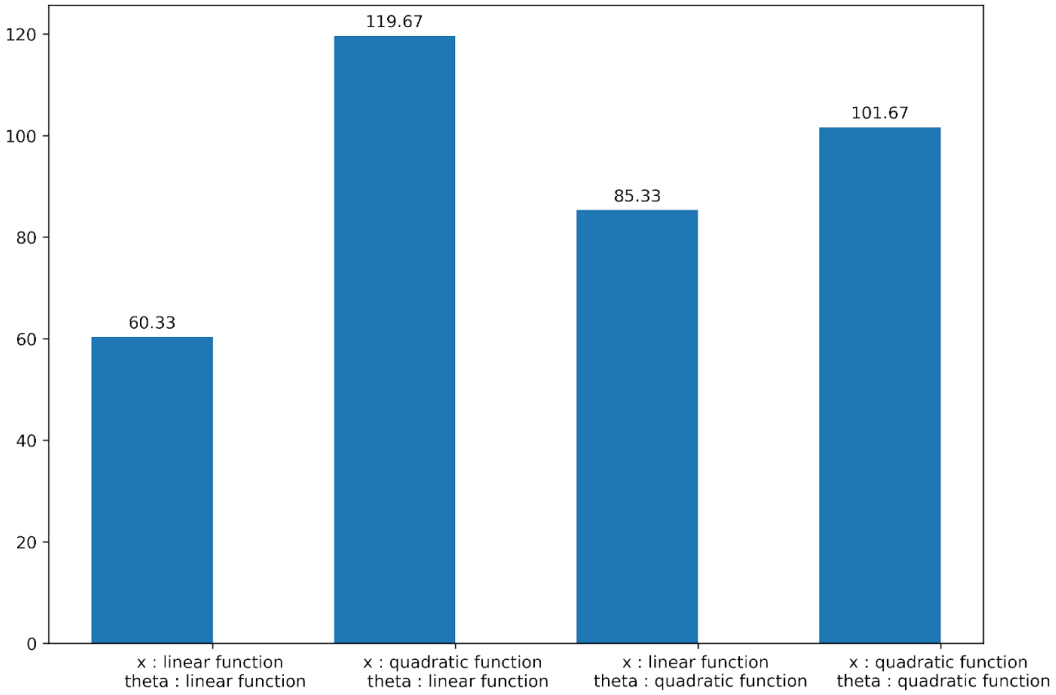
2.2 Models Tested

Four models were tested with both linear and quadratic reward functions for the x coordinate of cart and the angle of the pole with the vertical. The threshold for hyper-parameter testing was a

moving average score above 200 for the previous 10 games. Each combination was run thrice and averaged out to maintain consistency. The average number of tries required to achieve threshold by the various hyper-parameters were as follows :

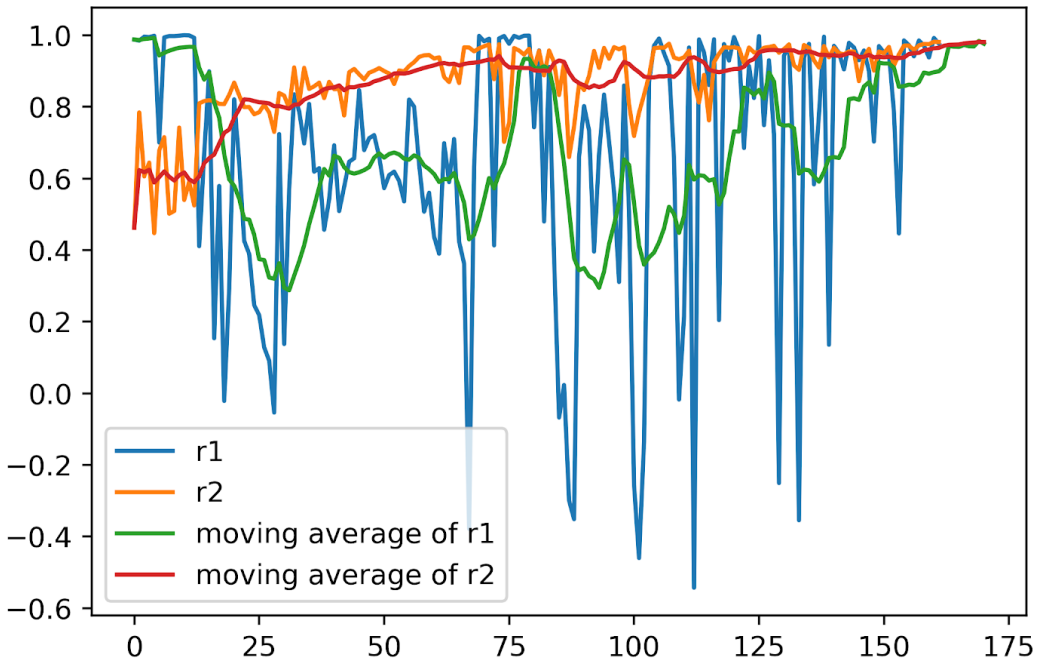


The best hyper parameters were tested against each other with a threshold for a moving average score of more than 470 for the previous 10 games. The results were as follows:



Each model was run thrice till it hit a moving average of 470 over 10 games. The reward function with the linear function for both x coordinate and theta outperformed all the other models.

2.3 Incremental Learning



Incremental learning is based on an incremental learning scheme where the reward functions may change over time instead of being static[4]. The above graph depicts how the model learns to balance using its reward system, where r1 is the reward received from the cart staying towards the centre of the environment and r2 is the reward received from the pole staying vertical. As you can see the model needs a substantial jump in learning r2 to compensate for the reduction in r1 which occurs when the pole starts to drift from the centre as the cart tries to balance the pole. Our hypothesis was that the pole should not be reprimanded to stay in the centre of the environment in the beginning stages as this might hinder the learning process of pole (it might drop the pole just to stay in the centre and it would still be rewarded). The intention with the incremental learning approach was to give less importance to r1 in the beginning and progressively increase its importance over time. This was achieved by making r1 dependent on the moving average of r2 over the last 10 games and increase it as r2 increases. The theory was tested on the best model of the previous experiment (x : linear function , $i_x = 0.6$, θ : linear function , $i_\theta = 0.8$)

$$r_1 = r_x$$

$$r_2 = r_\theta$$

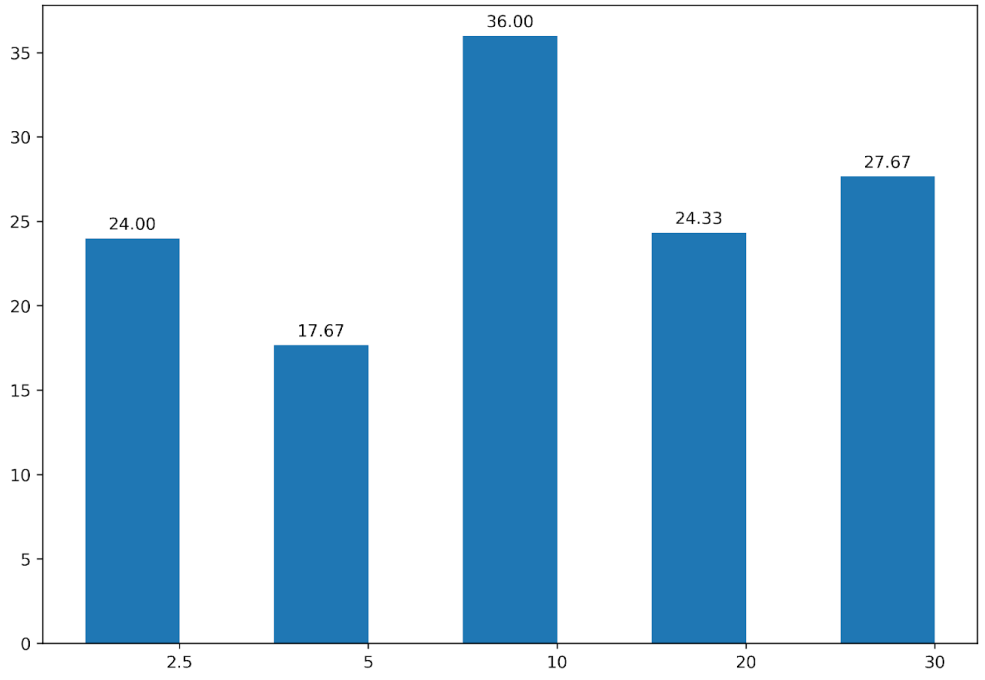
$$r_x = \frac{1}{1 - avg_\theta} * \frac{1}{i_x} * \left(\frac{x_{threshold} - |x|}{x_{threshold}} - 0.6 \right)$$

avg_θ: average value of r2 over the previous 10 games

i_x = 2.5,5,10,20,30

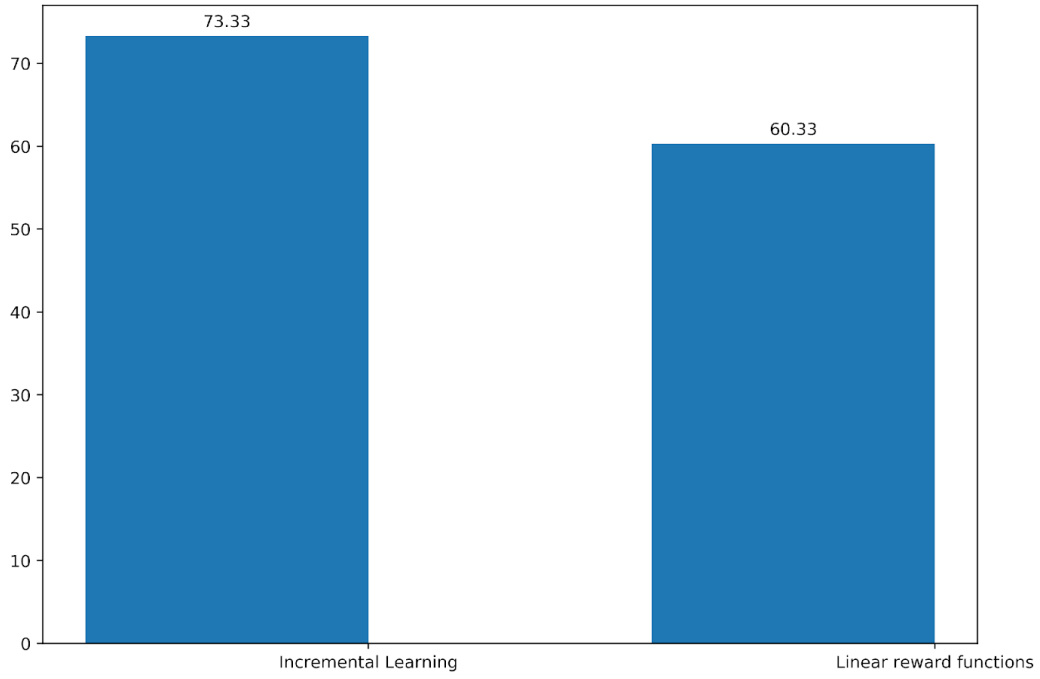
$$r_\theta = \left(\frac{\theta_{threshold} - |\theta|}{\theta_{threshold}} \right)$$

reward = $r_x + r_\theta$



The hyper parameters show the influence of r1 decreasing with an increase in i. Five values of i were tried. As we can we can see r1 increases as the average r2 increases.

The threshold for hyper parameter testing was the same as before. The threshold for hyper parameter testing was a moving average score above 200 for the previous 10 games. Each combination was run thrice and averaged out to maintain consistency.



The best hyper parameter was run against the original linear function with a threshold of 470 over three games. The results were as follows: As incremental learning does not provide significant improvement over linear reward functions we go ahead with using linear reward functions.

Chapter 3

Loss Functions

3.1 Different types of Loss functions

Loss Functions are used to calculate the gradients to update weights. In our project, we tried out four different loss functions and documented the result. The four functions used were[2] [3]:

3.1.1 Mean Squared Error:

This Loss function is defined as the sum of the squared difference between the actual and predicted values.

$$MSE = \frac{\sum_{i=1}^{n}(y_i - y_i^p)^2}{n}$$

3.1.2 Mean Absolute Error

This Loss function is defined as the absolute difference between the actual and predicted values.

$$MAE = \frac{\sum_{i=1}^{n}|y_i - y_i^p|}{n}$$

3.1.3 Huber Loss

This Loss Function is an update to the normal MAE. It becomes smoother and curves as the graph approach minima which ensures a low gradient value. It is less sensitive to outliers as compared to MSE. We used the Huber Loss with a hyperparameter Delta equal to 1 initially.

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 \ , for\, |y - f(x)| \leq \ \delta \\ \delta|y - f(x)| - \frac{1}{2}\delta^2 \, , otherwise \end{cases}$$

3.1.4 Log Cosh Loss

This Loss Function is very similar to Huber Loss functions with hyper-parameter delta close to 1. It is also robust to outliers like Huber Loss.

$$L(y, y^p) = \sum_{i=1}^{n} log(cosh(y_i^p - y_i))$$

3.2 Performance of different Loss functions

The measure we used to check the learning of our model using these Loss functions was the average of the previous 20 scores in the trial.

To get a cursory understanding of these Loss functions, we first set the threshold to 30, and plotted the mean and median of the number of trials to reach the threshold. We realized that MAE worked poorly as compared to the other loss functions and took more trials to reach the same score. MSE and Huber Loss were comparable, while Logcosh performed better than all the functions.

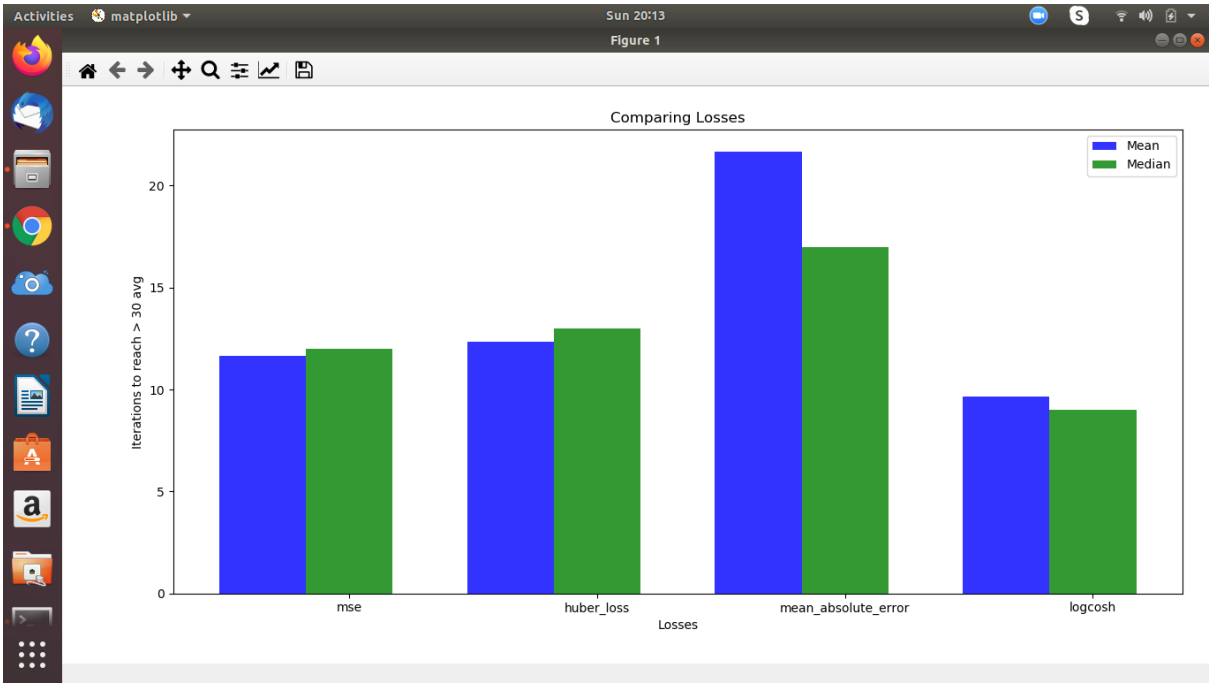


FIGURE 3.1: All the Loss functions

To take this further, we compared the functions excluding MAE, with a threshold of 470, which is more relevant to our learning. Here, Huber Loss performed much better than its counterparts, and we chose to move forward with this.

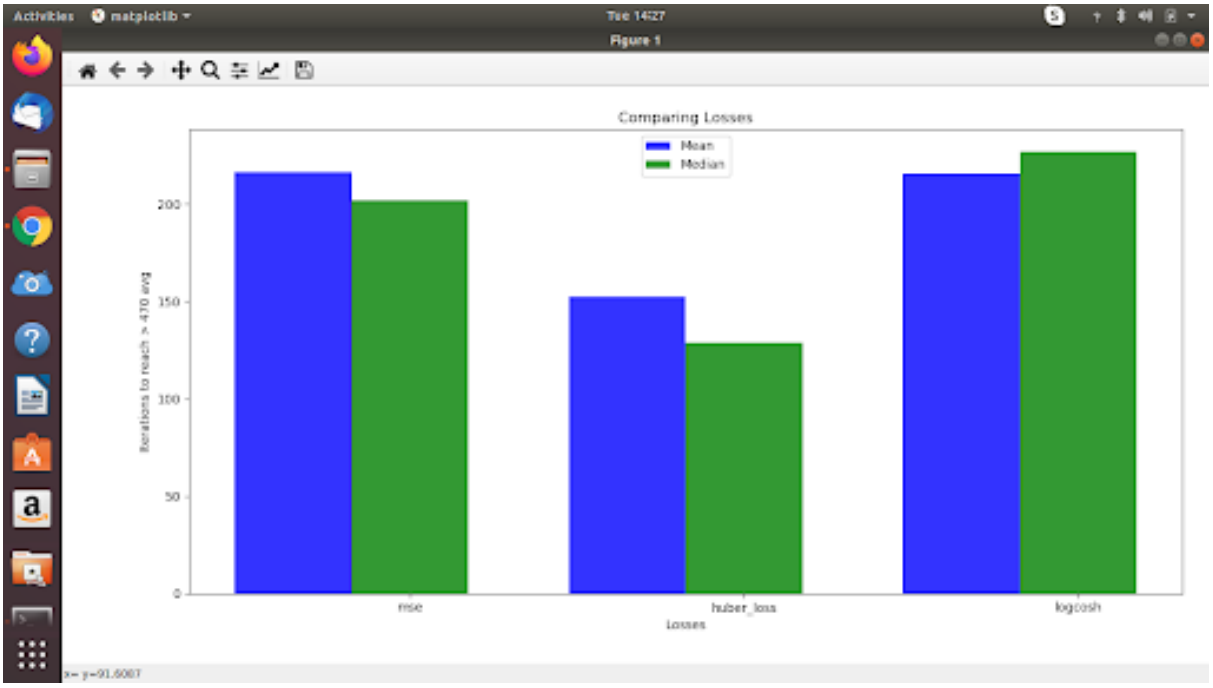


FIGURE 3.2: All the Loss functions

Chapter 4

Final Model

4.1 Final Model Used

We used the same model for all the three variations of the environment. The configuration of the model is:
Method used: Q-learning with value function approximation
Neural Network used: 2 layers(32,16) Neural Network with learning rate of 0.001
Loss function: Huber Loss with clip delta = 1
Batch Size = 30
Minimum Epsilon Value = 0.01
Reward function used: Custom Linear function for both x and theta, with fine tuning derived from grid search

```
def huber_loss( y_true, y_pred, clip_delta=1.0):
    error = y_true - y_pred
    cond  = K.abs(error) <= clip_delta
    squared_loss = 0.5 * K.square(error)
    quadratic_loss = 0.5 * K.square(clip_delta) + clip_delta * (K.abs(error) - clip_delta)
    return K.mean(tf.where(cond, squared_loss, quadratic_loss))

def init(  ):
    model = Sequential()
    model.add(Dense(32, input_dim=states, activation='relu'))
    model.add(Dense(16, activation='relu'))
    model.add(Dense(actions, activation='linear'))
    model.compile(
                loss=huber_loss,
                optimizer=Adam(lr=learn_rate))
    return model
```

FIGURE 4.1: Snippet of Training model and Loss function

```
def retrain( batch ):
    global eps
    minibatch = random.sample(mem, batch)
    for state, action, reward, next_state, done in minibatch:
        target = model.predict(state)
        if done:
            target[0][action] = reward
        else:
            target[0][action] = reward + gam * np.amax(_target_model.predict(next_state)[0]_)
        model.fit(state, target, epochs=1, verbose=0)
    if eps > eps_min:
        eps *= eps_decay
```

FIGURE 4.2: Snippet of learning using Q-learning

```
next_state, reward, done, _ = env.step(action)
x,cart_vel,theta,pole_vel = next_state

r1 = (env.x_threshold - abs(x)) / env.x_threshold - 0.6
r2 = (env.theta_threshold_radians - abs(theta)) / env.theta_threshold_radians - 0.8
reward = r1 + r2
```

FIGURE 4.3: Snippet of the Reward function used

4.2 Performance of Final Model

The model was trained on three different variations of cart-pole environment (with friction,gravity and noise) for 300 episodes. Given below is the graph showing their moving average of the last 100 episodes against the episodes.

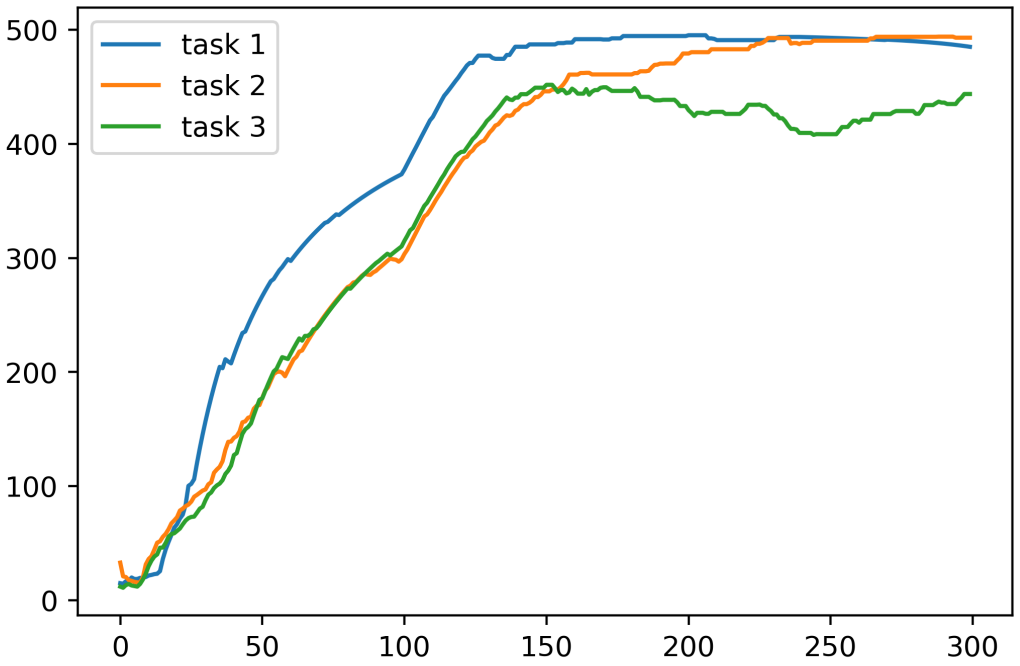


FIGURE 4.4: Performance of final model on all three environments

Bibliography

[1] URL: https://medium.com/@BonsaiAI/deep-reinforcement-learning-models-tips-tricks-for-writing-reward-functions-a84fe525e8e0.

[2] URL: https://towardsdatascience.com/understanding-different-loss-functions-for-neural-networks-dd1ed0274718.

[3] URL: https://heartbeat.fritz.ai/5-regression-loss-functions-all-machine-learners-should-know-4fb140e9d4b0.

[4] Alain Dutech, Olivier Buffet, and François Charpillet. "Multi-agent systems by incremental gradient reinforcement learning". In: *International Joint Conference on Artificial Intelligence*. Vol. 17. 1. Citeseer. 2001, pp. 833–838.

[5] Matteo Hessel et al. "Rainbow: Combining improvements in deep reinforcement learning". In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[6] Vijay R Konda and John N Tsitsiklis. "Actor-critic algorithms". In: *Advances in neural information processing systems*. 2000, pp. 1008–1014.

[7] David Silver, Richard S Sutton, and Martin Müller. "Sample-based learning and search with permanent and transient memories". In: *Proceedings of the 25th international conference on Machine learning*. 2008, pp. 968–975.

[8] David Silver et al. "Deterministic policy gradient algorithms". In: 2014.

[9] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*. Vol. 135. MIT press Cambridge, 1998.