MACHINE LEARNING

BITS F464

REPORT ON REINFORCEMENT LEARNING PROJECT

By

Soneji Visarg Balkrishna : 2017A7PS0029G

Avil Aneja : 2017A7PS0968G

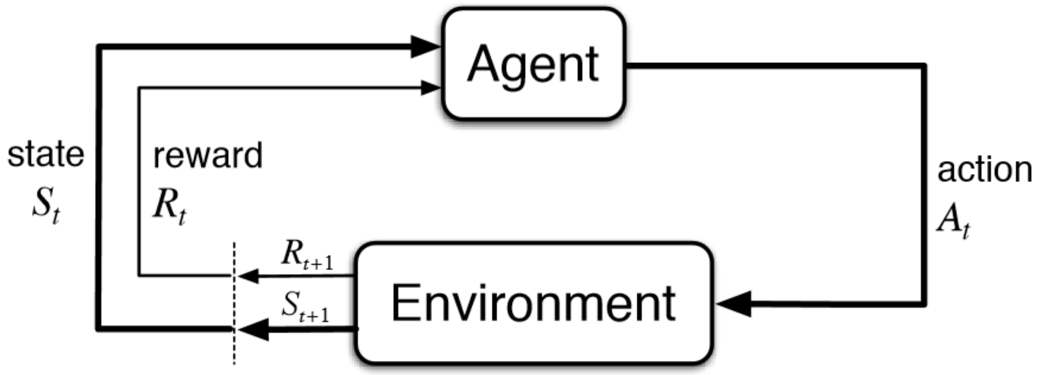
Aryan Kabra : 2017A3PS0333G

INTRODUCTION

Reinforcement learning is an area of Machine Learning. It is about taking suitable action to maximize reward in a particular situation. It is employed by various software and machines to find the best possible behavior or path it should take in a specific situation. Reinforcement learning differs from the supervised learning in a way that in supervised learning the training data has the answer key with it so the model is trained with the correct answer itself whereas in reinforcement learning, there is no answer but the reinforcement agent decides what to do to perform the given task. In the absence of a training dataset, it is bound to learn from its experience.

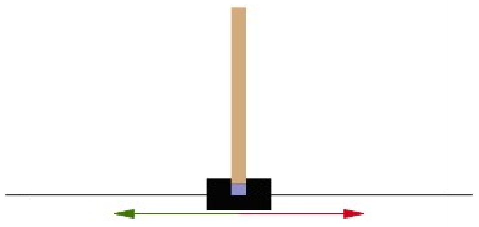
SPECIAL FOCUS ON CART-POLE PROBLEM :

Cartpole is built on a Markov chain model that is illustrated below :



Then for each iteration, an agent takes current state (St), picks best (based on model prediction) action (At) and executes it on an environment. Subsequently, the environment returns a reward (Rt+1) for a given action, a new state (St+1) and an information if the new state is terminal. The process repeats until termination.[7]

Cartpole - known also as an Inverted Pendulum is a pendulum with a center of gravity above its pivot point. It’s unstable, but can be controlled by moving the pivot point under the center of mass. The goal is to keep the cartpole balanced by applying appropriate forces to a pivot point.



Cartpole schematic drawing

- Violet square indicates a pivot point.
- Red and green arrows show possible horizontal forces that can be applied to a pivot point.

A pole is attached by an un-actual joint to a cart, which moves along a frictionless track.The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that pole remains upright.The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

So considering the above problem we have our action space as discrete with +1(push cart to the right) and 0(push cart to the left) values applied to the cart , so that the pole remains stable. However, coming to the observation space we find it to be continuous with following parameters in it :

1. Cart Position with a minimum value of -2.4 and a maximum value of 2.4.
2. Cart velocity with a minimum value of -inf and a maximum value of +inf.
3. Pole angle with a minimum value of -41.8 degrees to a maximum value of 41.8 degrees.
4. Pole velocity at tip with a minimum value of -inf and a maximum value of +inf.

- Reward is 1 for every step taken, including the termination step. The threshold is 475 for v1.
- All observations are assigned a uniform random value between ±0.05 in the starting step.
- Episode Termination
  - Pole Angle is more than ±12°
  - Cart Position is more than ±2.4 (center of the cart reaches the edge of the display)
  - Episode length is 500 (for v1).

CartPole-v1

CartPole-v1 is a challenging environment which requires the agent to balance a pole on a cart for 500 timesteps. The agent solves the environment when it gets an average reward of 450 or more over the course of 100 timesteps. However, again for the sake of consistency, we again measure performance by taking the best 100-episode average reward.

We will be using Cartpole-v1 for this project with different little variations in the noise of the problem in various tasks.

PROBLEM STATEMENT

TASK 1 : Under all the tasks we have to balance the pole running on the flat terrain. There are little variations in the noise of the problem in various tasks. So for the task1 we are given variable gravity. In the basic problem we have fixed gravity as our environment but in this task we have variable gravity for every episode. So our model has to learn how to react to the change of gravity after every episode. Also, here there is a change in the mass of the pole from 0.1 to 0.4. Friction is also added to this problem for both cart and pole and there is random scaling of that feature also. Considering all these changes , we have to get the average reward value greater than or equal to 480 in 100 episodes. Here we have constant force of magnitude (i.e. self.force_mag = 10).

TASK 2 : Compared to task 1 , now we have varying force which is generated randomly from 7 to 13 units. Our goal remains the same, that is to balance the pole on cart so that our average total_reward exceeds 480 in a maximum of 100 episodes.

TASK 3: Till now we have noise parameter as 0. But now in this task we have made noise to be a uniform random number between -0.3 to 0.3. Parameter space noise injects randomness directly into the parameters of the agent, altering the types of decisions it makes such that they always fully depend on what the agent currently senses. Again our goal remains the same , that is to balance the pole on cart so that our average total_reward exceeds 480 in a maximum of 100 episodes.

ALGORITHMS TRIED

Simple Q-learning approach:

Temporal difference (TD) learning is a class of RL algorithms which involves one-step updates of the value function and bootstrapping to estimate the quality of a state. In TD methods, the quality of the state at every step is updated using the reward obtained at that step and an old estimate of quality of the next state. . Q-Learning is a TD algorithm applied to control problems using an off-policy method since two different policies are utilized by the agent. The policy used to select actions using the state-action values is the greedy policy, given by $\max_{a'}$ $Q(S_{t+1},a')$.[5]

Q-Learning is a model-free form of machine learning, in the sense that the AI "agent" does not need to know or have a model of the environment that it will be in. The same algorithm can be used across a variety of environments.For a given environment, everything is broken down into "states" and "actions." The states are observations and samplings that we pull from the environment, and the actions are the choices the agent has made based on the observation.

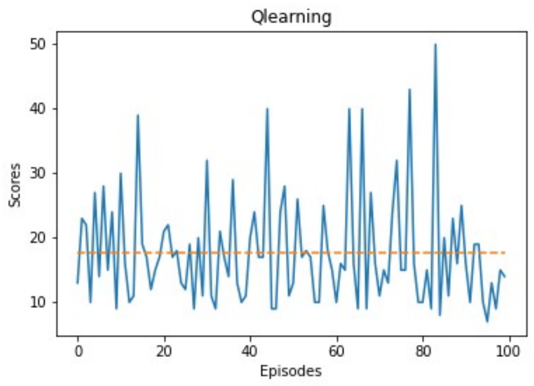
Using this approach , you get Q-values for every action you take at a state. Over time you will update these values in such a way that these will produce good results.For saving these values Q-table is used.It is a table having all combinations of states and actions. We will look into it and select the action corresponding to maximum Q-value.Initially , Q-table is filled by random values. Our model will then explore and update these Q values using the following formula :

Source: https://en.wikipedia.org/wiki/Q-learning

$$Q^{new}(s_t,a_t) \leftarrow (1-\alpha)\cdot \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\bigg(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}} \bigg)}^{\text{learned value}}$$

But the major problem to use Q-learning was discrete observation space. Since Q-table contains all possible combinations of states so observation space can't be continuous but in our case we have continuous observation space (as distance , velocity , angle etc. can be any real number between the given range). Hence to resolve this problem we tried to group the values of different parameters of observation space. So, we made the window size of each observation space parameter as (env.observation_space.high - env.observation_space.low)/DISCRETE_ OS_SIZE where DISCRETE_OS_SIZE = [100,100,100,100]. Hence we made it discrete space and then used a Q-learning algorithm.

So we used this basic approach for solving our task 1. It was just a start to explore which algorithms will be suited better as we all also do reinforcement learning in our life. The result for task 1 was not satisfactory as can be seen in the following graph :



As we can see the above graph between scores and episodes , it doesn’t give the results that we require. It never reached closer to even 100. So we concluded that this model is not going to work, we have to see other models. However , for the above model we get an average of reward as 18.37 which is far lesser than our expectation.

HILL CLIMBING APPROACH :

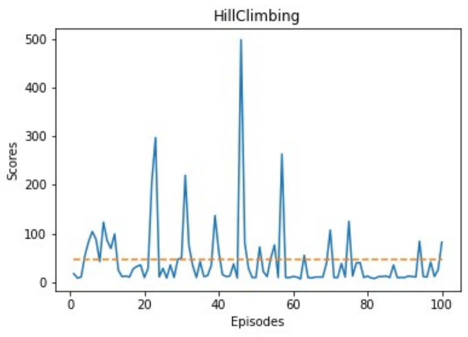
Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.A node of hill climbing algorithm has two components which are state and value.In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

How did we use it in the CartPole environment?
Hill climbing is a technique used for solving optimization problems usually where we're trying to optimize a function to maximize or minimize its output. It starts with a random initial solution and then uses a trial and error approach to tweak the solution in an arbitrary way and then try it with the given inputs and assess how well it did compared to the previous solution. If the modified solution performed better then it becomes our new best solution and we repeat the process again. However if the modified solution performed worse than the previous solution then we describe that one and try different modifications of the best solution.

So now let's think about how we would apply this to the CartPool environment the CarPool environment .At each time step we were given the current state of the carts position and velocity along the track as well as the polls angle and angular velocity which we would need to use to determine whether to push the car left or right to keep the pole balanced so essentially we know Function that takes in the current state and outputs the best action to keep the pole balanced and therefore maximizing the overall award sounds like a task for hill climbing now our state will be given to us as a vector of the carts position velocity and the polls angle and angular velocity so we need our function to output the best action of left or right to take in a particular state.

To generalize for any number of actions we will output a vector with predicted values for taking each action in that state where the action with the highest value is the one we will take in that given state. Then to transform our input state vector of four elements to an output vector of two elements our function can simply matrix multiply the input state with a four by two weight matrix to output two values matching our left or right action options.

Now we will start with a random initial weight matrix and take our function for an initial test run in the environment to see how well it could balance the pole using those initial weights based on the total of the rewards from that episode.We will save these as the initial best observed reward and the best weights that achieved the reward which we will represent as R0 and W0 who then sample a new trial weight matrix as w by adding a random noise matrix and to the best weights and then test the functions performance again where the trials total reward as r will be compared with the best observed of reward for updating our weights. if the trail reward is greater than the best observed reward we will save it as the new best reward and update our best job



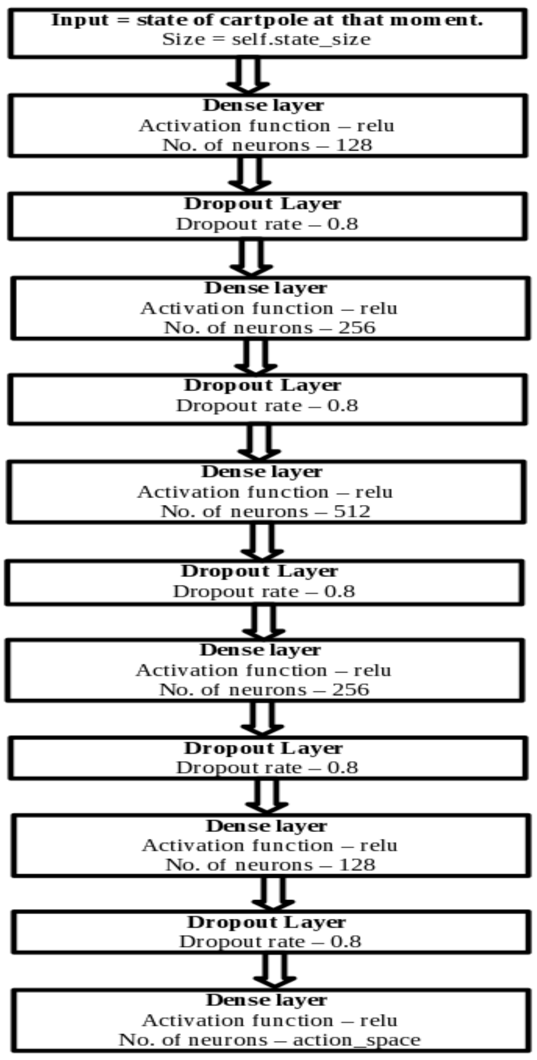
.

It is better than the Q-learning simple algorithm but still not satisfactory for task1 as there are kinks in between i.e. learning is not good.

Training neural network model on random games

To train this model the training data here is the observation and the move made, but the moves will all be random. So we will only append it in the training data if the score in that episode is greater than 50. For each episode we select random actions to select the next move and if that episode gives score >50 we store it in training data and append corresponding scores to an accepted score array.

For using this training data we create a neural network with an input layer, 5 hidden layers and an output layer. The 5 hidden layers will be fully connected layers with activation function as “relu” followed by a dropout layer. For the output layer size is 2 as we need only 2 outputs that are movement towards left or right. The activation for this layer is softmax. The estimator used for this model is regression with an “adam” optimizer.[1] Neural network is shown below diagramatically :



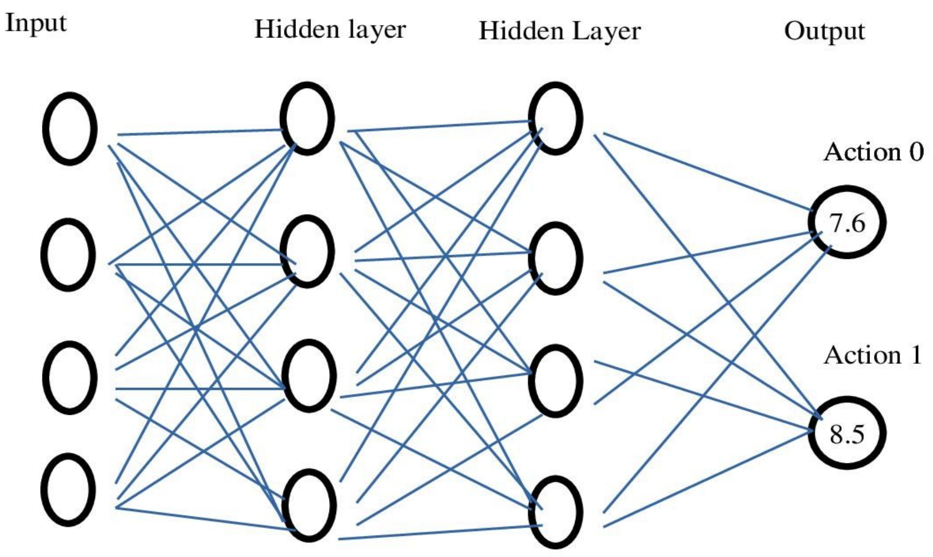
Now to train this model we select X as observation and y as moves. So this model will help us to predict the next move based on current observations. Now we will use data with scores greater than 50 to train this model. When we train this model with task1 settings and test it on 50 episodes we get an average score of 215. So this model was not satisfactory. Hence we now move to Deep Q learning which uses concepts of neural networks with Q-learning.

ONE WITH THE BEST RESULTS

DEEP Q LEARNING :

So this technique gave us the best results compared to the above methods we have applied. In the end of every approach we need to find the q-values and take the best action corresponding to the highest q-value. So in this approach, q-value will be predicted by our Neural network.With DQNs, instead of a Q Table to look up values, you have a model that you inference (make predictions from), and rather than updating the Q table, you fit (train) your model.

Let us consider an example : We feed our current state to our neural network (say any for once)and then it predicts the following values. Consider the index to be the action number too :



Since in the above say neural network we get our q-values to be 7.6 for action0 and 8.5 for action1. So we will take the action corresponding to the maximum q-value i.e. here action 1 will be taken.

As we engage in the environment, we will do a .predict() to figure out our next move (or move randomly). When we do a .predict(), we will get the 2 float values, which are our Q values that map to actions. We will then do an argmax on these, like we would with our Q Table's values. We will then "update" our network by doing a .fit() based on updated Q values. When we do this, we will actually be fitting for all 2 Q values, even though we intend to just "update" one.

The formula for a new Q value changes slightly, as our neural network model itself takes over some parameters and some of the "logic" of choosing a value. Now, we just calculate the "learned value" part:

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \boxed{\overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}}$$

So our loss function become as[8] :

$$loss = (r + \gamma maxQ'(s, a') - Q(s, a))^2$$

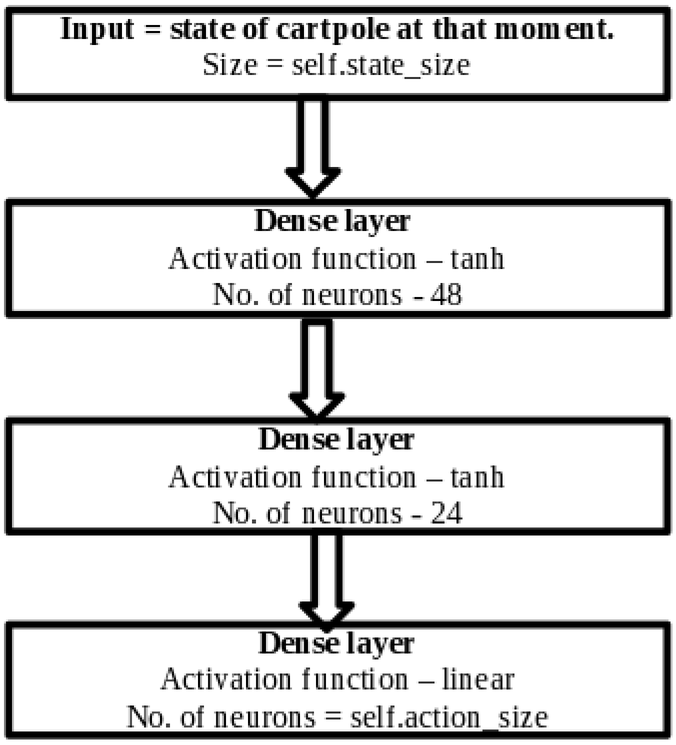
The basic intuition behind the whole concept is we first carry out an action a and observe the reward r and resulting new state s. Based on the result, we calculate the maximum target Q and then discount it so that the future reward is worth less than immediate reward. Lastly, we add the current reward to the discounted future reward to get the target value. Subtracting our current prediction from the target gives the loss. Squaring this value allows us to punish the large loss value more and treat the negative values same as the positive values.

We also use an exploration-exploitation strategy while deciding the action to take in the current state. We use the ϵ-greedy strategy while deciding the action. The value of this ϵ is decreased after every episode ends. It is multiplied by a decay factor of 0.995 after every episode ends. This is needed because initially our function approximator for Q-values is not correct. Hence, we need to explore more actions. As we interact more with the environment and update our weights for the function approximator, it becomes more reliable and gives more stable values. Hence, we reduce exploration and our policy starts to resemble the greedy policy.[4]

However we need not to worry about these things as keras calculate this loss function internally, we just need to mention the updated q-values i.e. r + ƔmaxQ'(s,a').

Let’s take a look at the neural network we used :

TASK1 & TASK2 :


Input = state of cartpole at that moment.
Size = self.state_size
Dense layer
Activation function – tanh
No. of neurons - 48
Dense layer
Activation function – tanh
No. of neurons - 24
Dense layer
Activation function – linear
No. of neurons = self.action_size

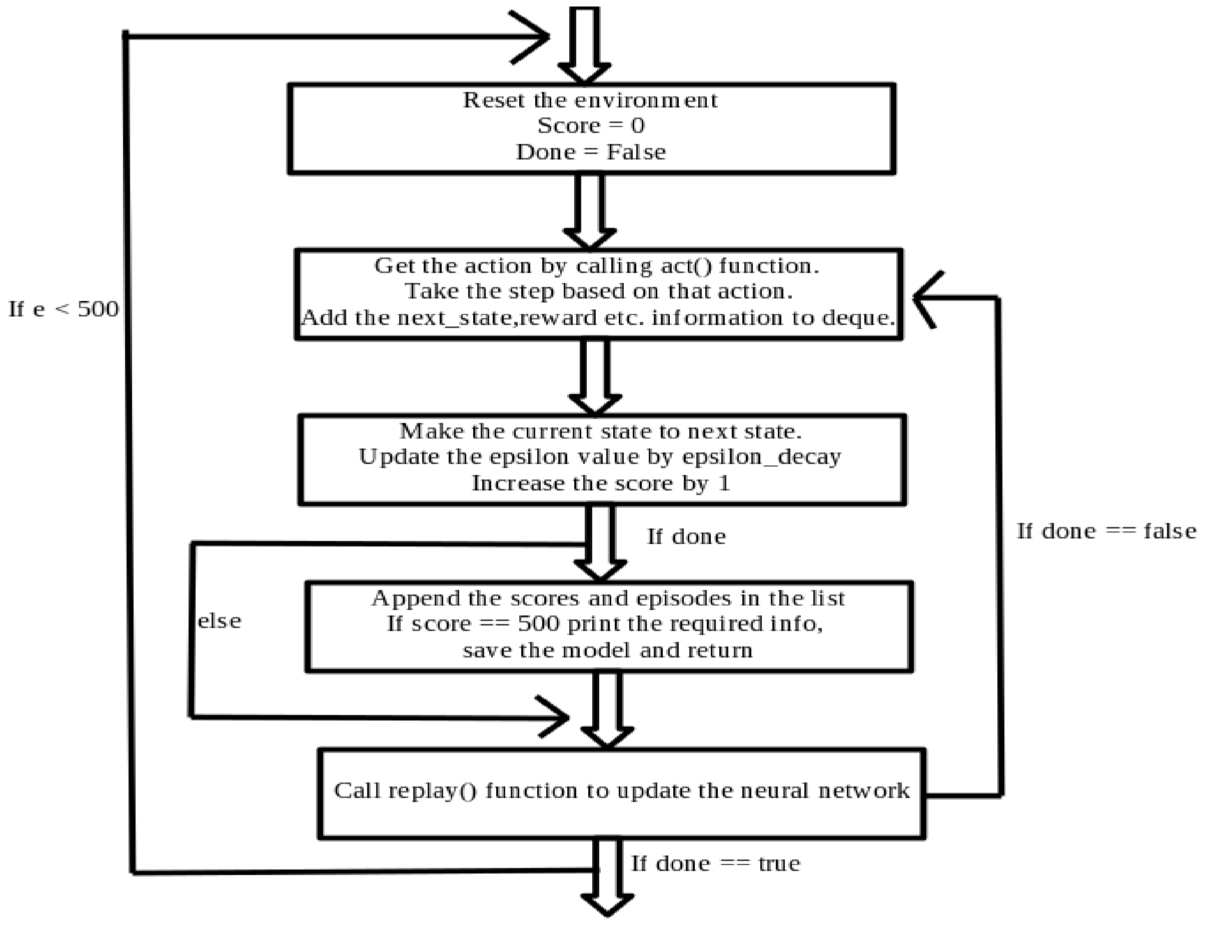
WHAT IS HAPPENING IN THE CODE?

First let us look at some of the important functions:

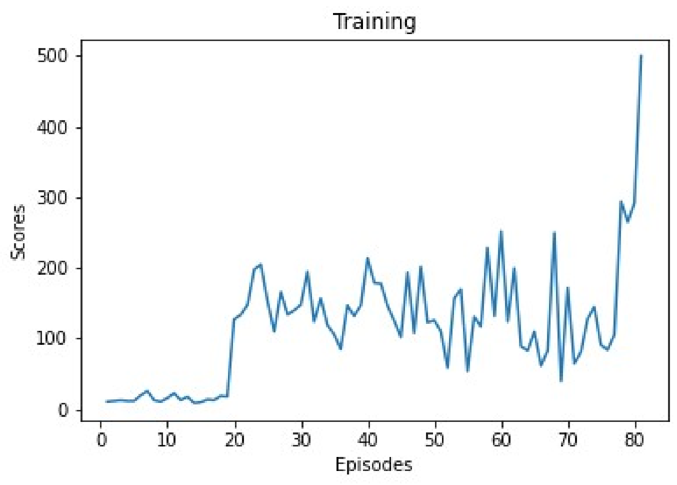
1. **build_model() -** In this function we are just making the model corresponding to the neural network mentioned above.
2. **remember(self,state,action,reward,next_state,done) -** This function just appends the information of the next_state and reward achieved by taking the action on the state. This is stored in the deque which is used for training the model later when it has filled at least equal to batch_size.
3. **act(self,state) -** This function is used for the model to learn either randomly (i.e. take random action) or predict from the weights obtained till now. This is done by epsilon which decays after every iteration. Hence if epsilon is greater than the random value generated then random action is taken else it is predicted from the model.
4. **replay(self) -** So, we have learned how the update happens in the neural network above. This happens in this function that is if the length of deque is greater than batch size then we take a minibatch from it randomly and fit the model using these values. Keras then do backpropagating and update the weights.

5. **exploration(self) -** This function simply updates the value of epsilon by a factor of epsilon_decay.

How is the model getting trained?


Reset the environment
Score = 0
Done = False
Get the action by calling act() function.
Take the step based on that action.
Add the next_state,reward etc. information to deque.
If e < 500
Make the current state to next state.
Update the epsilon value by epsilon_decay
Increase the score by 1
If done
If done == false
Append the scores and episodes in the list
If score == 500 print the required info,
save the model and return
else
Call replay() function to update the neural network
If done == true

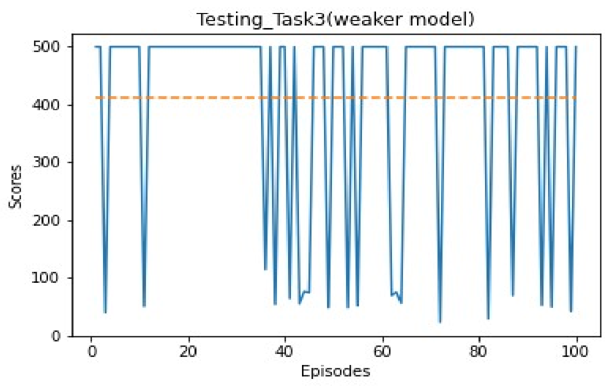
TRAINING PLOT FOR TASK1 & TASK2(trained on task1 basically):



No. of steps for training - 81

TASK3 : To show that it didn’t worked on the model trained earlier , we have our testing plot as follows:

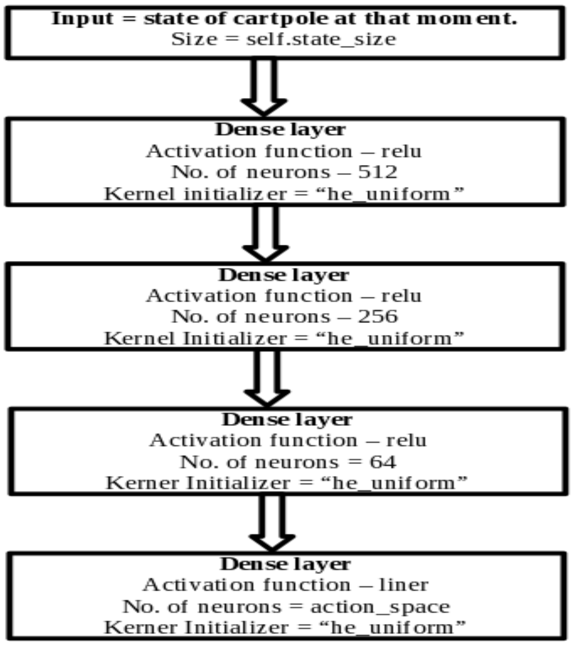
Testing Plot On Smaller Model(TASK 1 Model) :



Average Score : 412

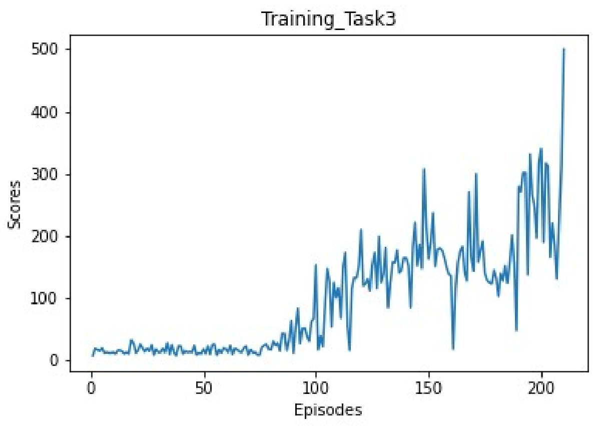
Hence we didn’t rely on this model for task3. We introduced more dense layers and changed some hyperparameters to get the desired results.

Since task3 contains a noise factor and has more randomness than task1 and task2 , so we need a bigger model (in terms of no. of neurons and layers) to train it and get the desired results. However, the basic structure of training and various functions will remain same as for task1 and task2, but we will change the neural network to as shown below :



Input = state of cartpole at that moment.
Size = self.state_size
Dense layer
Activation function – relu
No. of neurons – 512
Kernel initializer = "he_uniform"
Dense layer
Activation function – relu
No. of neurons – 256
Kernel Initializer = "he_uniform"
Dense layer
Activation function – relu
No. of neurons = 64
Kerner Initializer = "he_uniform"
Dense layer
Activation function – liner
No. of neurons = action_space
Kerner Initializer = "he_uniform"

TRAINING PLOT FOR TASK3:

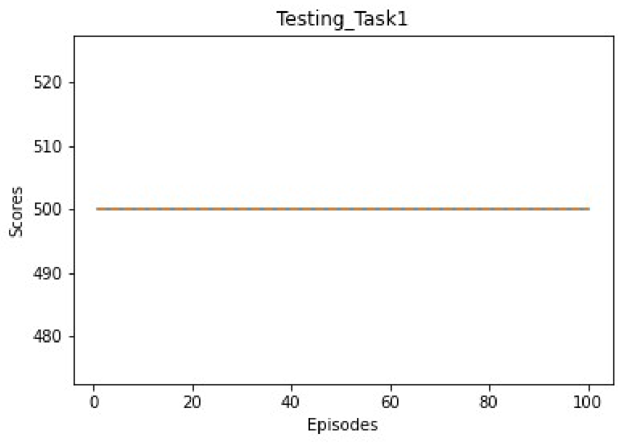


No. of steps for training – 210

RESULTS :

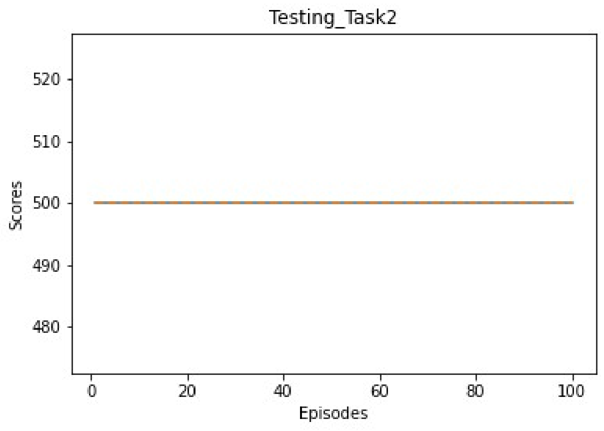
So now we have seen how to train the models for the respective tasks, So now it’s the time to see the performance of our model with the weights decided. For the task1 and task2 model used will be the one trained on task1 only and for task3 we have explicitly trained other models with greater complexity.

TASK1 :
Testing Plot :



Average Score : 500

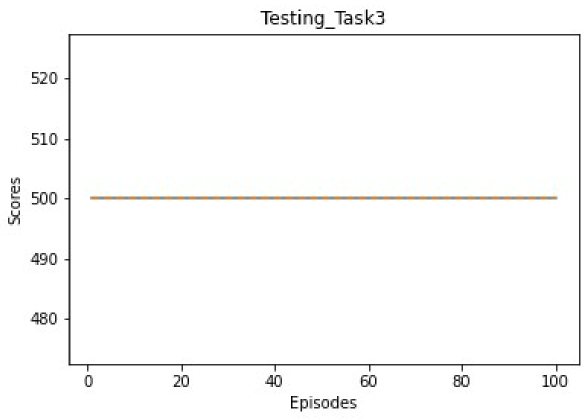
TASK2 :
Testing Plot :



Average Score : 500

TASK 3:

Testing Plot with the new model :



Average Score : 500

CONCLUSION

From the above three tasks we conclude that using neural networks with Q-learning can help us to deal with various complex environments. However using neural nets requires more space and time but can help us to achieve the goals easily.

We were able to get the score greater than 480 for each of the tasks. For task1 and task2 we were able to achieve our goal with training only one model. But for task3 we have to increase the dense layer to reach the score of 480. From this we can also conclude that if noise or randomness increases in the environment then sometimes adding layers or making neural nets can help us to make the computer predict correct values or here keep the pole stable on the cart for more than 480 frames.

ACKNOWLEDGEMENT

We would like to thank our professor Prof. Ashwin Srinivasan and Prof. Tirtharaj Dash for giving us the opportunity to work on this project. We would also like to thank our teaching assistants Adithya M Niranjan, Aditya Ahuja, Naigam Shah, Khyati Jain and Sharan for helping us throughout the duration of this project.

REFERENCES

[1] *Using a neural network to solve OpenAI's CartPole balancing environment*

[2]*A Beginner's Guide to Deep Reinforcement Learning*

[3] E Knight, O Lerner *Natural Gradient Deep Q-learning*

[4]A Kumar, J J Hasenbein. *Enhancing Performance of Reinforcement Learning Models in the Presence of Noisy Rewards*

[5]S Nagendra, N Podila, R Ugarakhod, K George. *Comparison of Reinforcement Learning Algorithms applied to the Cart-Pole Problem*

[6]VF-Lavet, P Henderson, R Islam, MG Bellemare and J Pineau. *An Introduction to Deep Reinforcement Learning*

[7]Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*

[8]V Mnih , K Kavukcuoglu, D Silver, A Rusu. *Human-level control through deep reinforcement learning*

CONTRIBUTION :

The work was divided equally between the three of us. We initiated with basic Q learning and one of us tried with a simple Q-learning algorithm, then one of us moved to a hill climbing approach. We in the end came up to deep Q learning and implemented it together using google meet to discuss. Similarly, we moved to the report part and did it with equal efforts of all of us.