

BITS PILANI, K K BIRLA GOA CAMPUS

Machine Learning(BITS F464) : Major Project

Urvil Nileshbhai Jivani(2017A7PS0943G)

Anupreet Singh(2017A7PS0955G)

Adithya Samavedhi(2017A7PS0071G)

Abstract

This document is for partial fulfillment of the Major Project of the course Machine Learning(BITS F464) in BITS Pilani, Goa campus. We were given three variants of standard Cartpole-v1(Open AI). We tried different deep RL(Reinforcement Learning) algorithms and got fantastic results in our research project. After training the model on that task, we got **500** reward for all three tasks over **100 episodes**. We found that **DQN** was giving the best results on all three tasks.

Keywords Reinforcement Learning, Deep Learning, Cartpole problem, DQN, Policy Gradient

Acknowledgement

We express our sincere gratitude to Ashwin Sir, Tirtharaj Sir, and TAs for allowing us to work on this significant lab project on the Cartpole environment from OpenAI. Over the course of this major lab project, we came across opportunities to learn the implementation of neural networks, explore reinforcement learning agents' policies, and apply them to the Cartpole environment. We are also thankful for the BITS curriculum that provides us with similar opportunities to put theory and skills to practical use.

CONTENTS

1	Introduction	5
1.1	Reinforcement Learning	5
1.1.1	Environment and Agent	5
1.1.2	Inside An RL Agent	5
1.1.3	Classification of RL Agents	5
1.2	Cartpole-v1	6
1.3	Objective	6
1.3.1	Task1	6
1.3.2	Task2	7
1.3.3	Task3	7
2	Methodology	8
2.1	Introduction to Q-Learning	8
2.2	DQN:Our Approach	10
2.2.1	Models	12
2.3	Policy Gradient:Our Approach	17
3	Results	21
4	References other than Research Papers	22

1 INTRODUCTION

1.1 REINFORCEMENT LEARNING

Reinforcement Learning(RL) is the third stream of Machine Learning after supervised and unsupervised learning, but closest to the learning animals and humans do.

1.1.1 ENVIRONMENT AND AGENT

It is the physical world in which Agent operates. We are supposed to create an Agent in response to an RL problem that can make decisions and reach the goal state. The Agent is provided with reward and observation by the Environment. The Agent takes action with the help of previous rewards and observations.

1.1.2 INSIDE AN RL AGENT

An RL Agent may contain one or more of these components:

- **Policy** : Agent's Behaviour function.
- **Value Function** : Gives us how good is each state and action.
- **Model** : Agent's representation of Environment.

1.1.3 CLASSIFICATION OF RL AGENTS

- **Model Free** : Only Policy and/or value function
- **Model Based** : Policy and/or value function + Model

In our approach to the problem assigned to us, we have explored the **Model Free** approach.

- **Policy Based**: policy
- **Value Based**: value function
- **Actor Critic**: policy + value function

There are two types of learning methods in RL :

- **On Policy Learning:** $Q(s, a)$ function is learned from actions we took using our current policy.
- **Off Policy Learning:** $Q(s, a)$ function is learned from different actions (for example, random actions). We even don't need a policy at all.

1.2 CARPOLE-V1

The cart-pole Environment consists of a pole attached by a 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 pole starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every time step that the bar remains upright. The episode ends when the pole is more than 15 degrees vertical or the cart moves more than 2.4 units from the center. The maximum number of no steps in each episode of the Caertpole-v1 problem is 500. The project aims to train a model to achieve the highest average reward over a 100-episode period(i.e., 500). For basic techniques of RL for the cart-pole problem and introduction to the cart-pole problem, refer to this [1].

1.3 OBJECTIVE

We were tasked to build a Reinforcement learning model that'll balance the pole on a cart running on a flat terrain with a noisy set of parameters like action and sensors. The values for gravity and friction randomly vary at each step for the same episode. We were given the following three tasks:

1.3.1 TASK1

The cart-pole problem with random variation in gravity and friction.

1.3.2 **TASK2**

The cart-pole problem with the previous set's noise and noisy controls. This means that the cart's force in the desired direction may be less/more than expected.

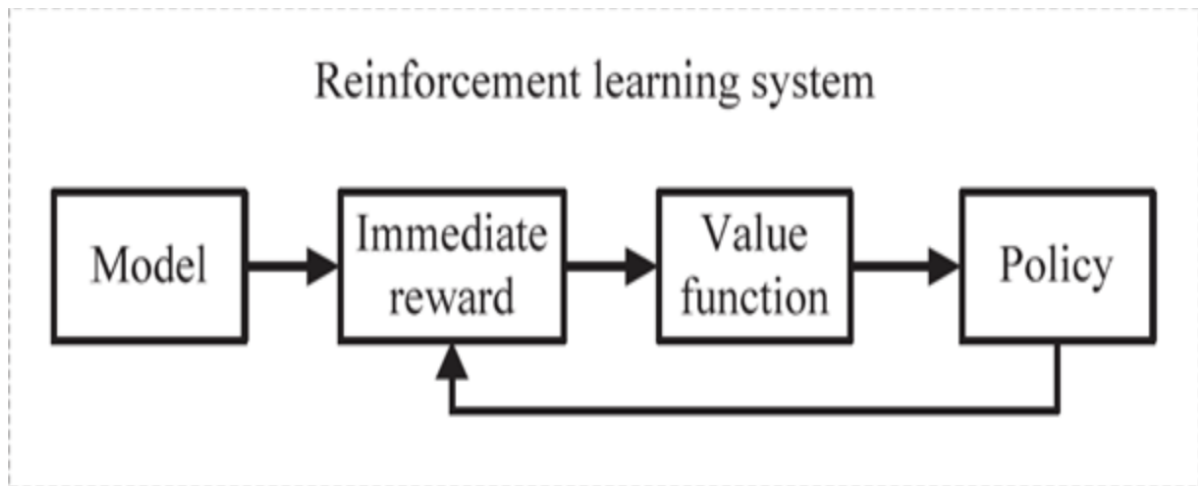
1.3.3 **TASK3**

The cart-pole problem with the previous modifications and noisy sensors/sensor observations of the pole angle at any moment.

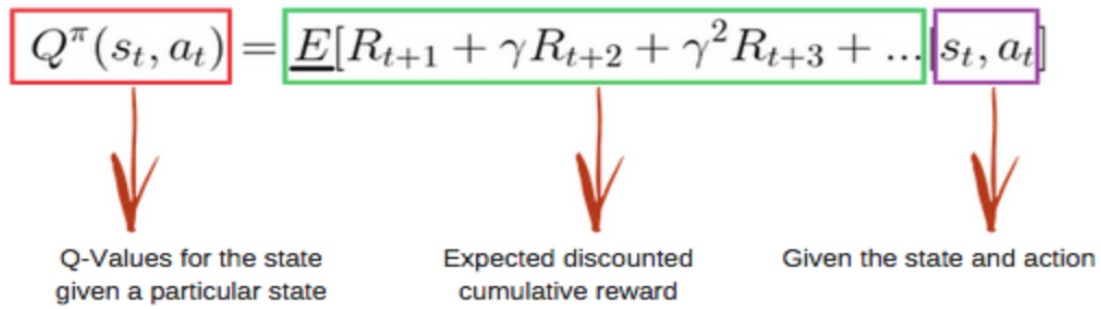
2 METHODOLOGY

2.1 INTRODUCTION TO Q-LEARNING

Q-learning [2] is a simple way for agents to learn how to act optimally in controlled Markovian domains. Q-Learning comes under the category of off-Policy reinforcement learning algorithms. It comes under off-policy because of the randomness in the function used to select an action, hence not requiring a policy. The main aim of Q-learning is to maximize the total rewards. Q learning seeks to learn from actions outside its current policy by choosing random actions with a certain probability. This probability of selecting an arbitrary action helps the agent explore new states.



The 'Q' in Q-learning represents quality. In our case, quality stands for how beneficial a specific action is in obtaining future rewards. In Q-Learning, we maintain a Q table consisting of [state, action] pairs, which make up our model's memory.

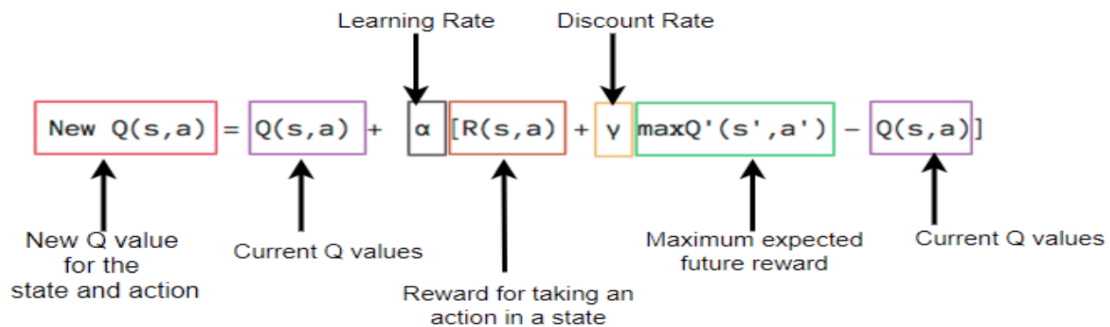
$$Q^{\pi}(s_t, a_t) = \underline{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | s_t, a_t]$$


Q-Values for the state given a particular state

Expected discounted cumulative reward

Given the state and action

Our agent interacts with the environment in two ways: Exploration and Exploitation. The first is when our agent takes random actions with a certain probability. These actions are outside the agent's current policy and hence act as a method of exploration of the environment by our agent. The second is Exploitation; here, our agent takes an option while viewing the possible actions from our q-table for the given state. It will choose the action with the maximum reward.

$$\text{New } Q(s, a) = Q(s, a) + \alpha [R(s, a) + \gamma \max_{a'} Q'(s', a') - Q(s, a)]$$


Learning Rate

Discount Rate

New Q value for the state and action

Current Q values

Reward for taking an action in a state

Maximum expected future reward

Current Q values

Here the $Q(\text{state}, \text{action})$ returns the expected future reward for choosing a particular action at that particular state.

The exploration is an important field since it is responsible for the times our model explores new states which have not previously been visited. The rate of choosing Exploitation vs. exploration is done by the parameter (epsilon).

Updating the Q table: The updates occur after each step or action and continue until the

episode is done or finished. Done in this case, signifies reaching an endpoint/ terminal point by the agent. In our Cartpole environment, done occurs when the model moves more than 2.4 units from the center or when the pole tilts more than 15 degrees with respect to the vertical.

Here are the three basic steps:

- The agent is present in an (S') state upon taking an (a') action and receives an (r')reward.
- The agent selects an action by referring to the Q-table and will choose the action with the highest value/maximum OR choose at random (epsilon)
- Updating the Q-values

Important terms:

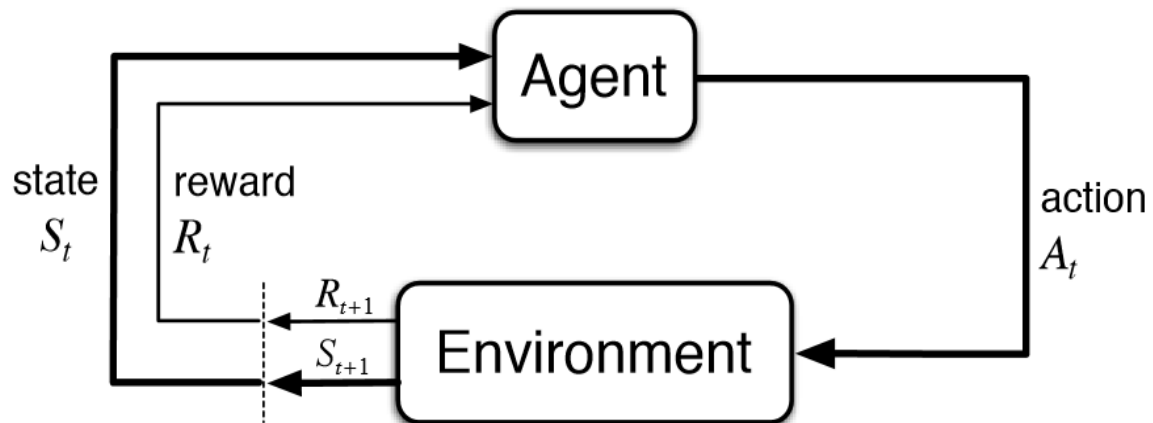
- **Gamma:** gamma is a discount factor. It's used to create a balance between immediate and future rewards. During our update, we apply the discount to the future reward.
- **Learning Rate:** lr or learning rate, often denoted by alpha. We take the difference between new and old and then multiply that value by the learning rate. This value then gets added to our previous q-value, which moves it toward our latest update.
- **Reward:** reward is the value received after completing a certain action at a given state. A reward can happen at any time step or terminal time step.

2.2 DQN:OUR APPROACH

We dropped the idea of using Q-Learning because we need large Q tables for learning complex models, and learning them will take more time. So we started working on DQN models. DQN uses Q-Learning ideas at its root level. We use a neural net to make predictions based on the model's current state to produce an action. This is Deep Q Learning.

Let's call our neural network model.

Our environmental behavior is something like this:



We use the next state to predict what would've been an optimized action.

We try to optimize our Q values. This can be done by minimizing the prediction and target gap. We use a square loss function to make the network learn more if it makes a wrong move. Also, a discount rate is used to give less weight to future predictions.

$$loss = (r + \gamma \max_{d'} Q'(s, d') - Q(s, a))^2$$

Here we have acted 'a' and observed a reward 'r'. We wanted to get an effort 'a' on a state 's'. Hence, we update our loss this way and then backprop through the net.

A replay function helps in doing this.

We define a memory for a network (2000 in our case). This memory holds previous actions, states, rewards, next_states, and the variable done. After each action of the environment, all these values are appended onto the memory, and a random sample (of size = batch size) of this is chosen. From the chosen sample, states and next_states are passed through the network to get targets and nex_targets, respectively. Now, these targets are used to find the optimal targets for particular actions using:

Target= reward + gamma*np.amax(target_max)

Here gamma is the discount rate.

After the targets are updated, we call model.fit(states, targets).

Thus, this is how our model gets trained.

Our model also uses an exploration rate (e) or (epsilon) to take random steps randomly. This helps in not training the model unless we have at least 1000 samples in memory. This makes sure that after a few random episodes (around 80 in our case), the model trains fast and has a good number of samples to get trained on.

This exploration is decayed over time using an epsilon decay.

An epsilon minimum is used so that we can explore once in a while with a very less probability.

The criteria we used to stop training our model were unconventional but gave nice results. The max score for cartpole problem is 500. I made sure that the model stops only when it sees 10 back-to-back 500's. This way, the model with a score of 500 would be saved, and it would not be justified by "Luck" that our model hit 500.

2.2.1 MODELS

We tried a number of models. Here is a summary of them.

- **Model 1:**

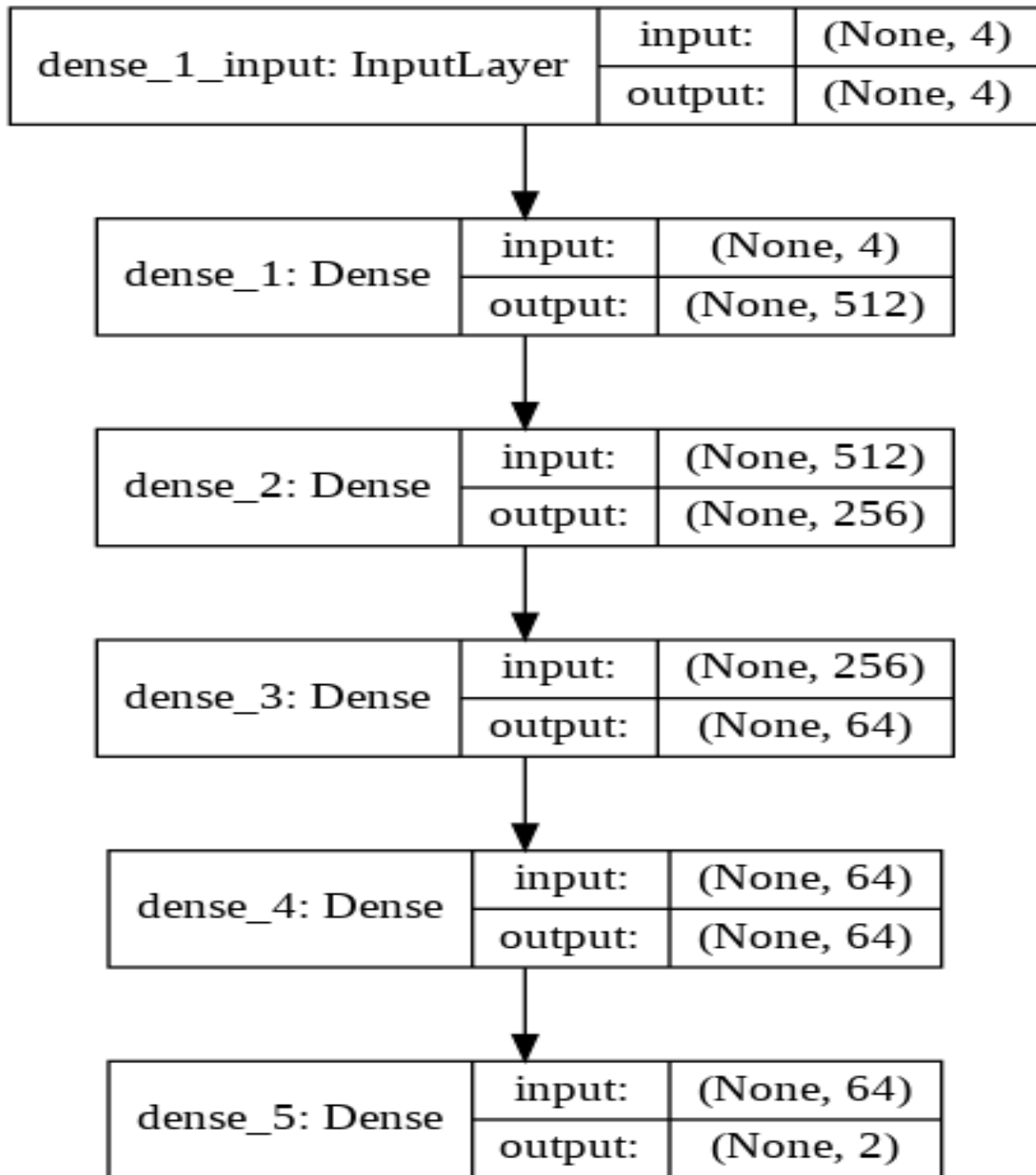


Figure 2.1: Model1

With a huge number of parameters (135000), this model was bound to give great results. A brief summary is given below.

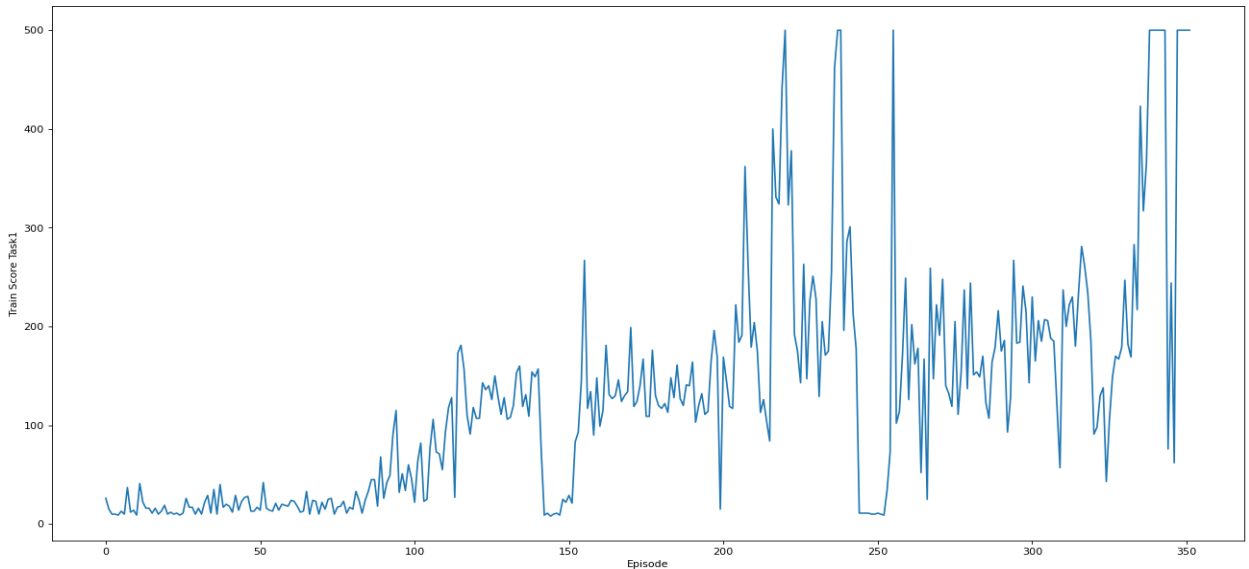


Figure 2.2: Training graph for Task1

The training and testing graph was similar to task 1.

To summarize: Testing Average Score

- Task1 500.0
- Task2 500.0
- Task3 500.0

A strange thing that happened was the model trained for task1 worked for task2 and task3 giving an average of 500. On rendering the environment, We learned that the model had learned to stay still and in one position. Hence it was this good. We have included the weights for this model in the file “t1model.h5”.

While this model will run for all three tasks, We have also included the weights for the other two models trained using this method in “t2model.h5” and “t3model.h5”.

Now the next task was to reduce the model's complexity.

- **Model 2:**

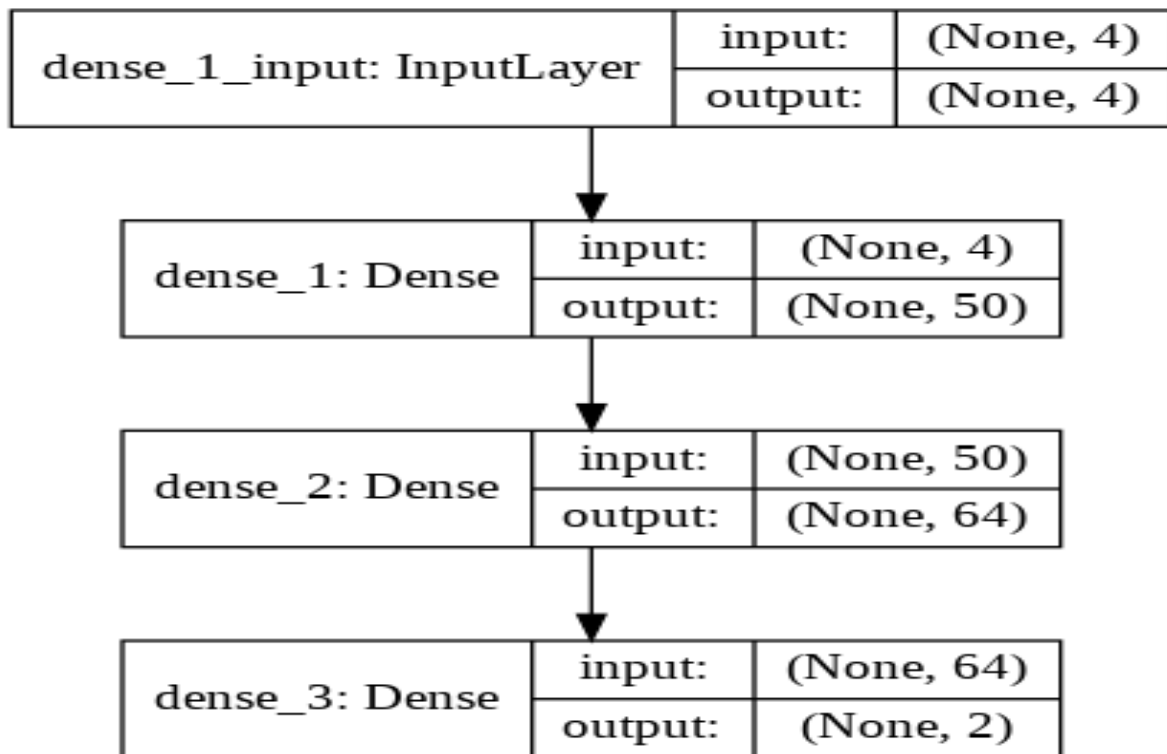


Figure 2.3: Model 2

This brings down the number of parameters to 3.5k from 135k. A great reduction. The summary is given below.

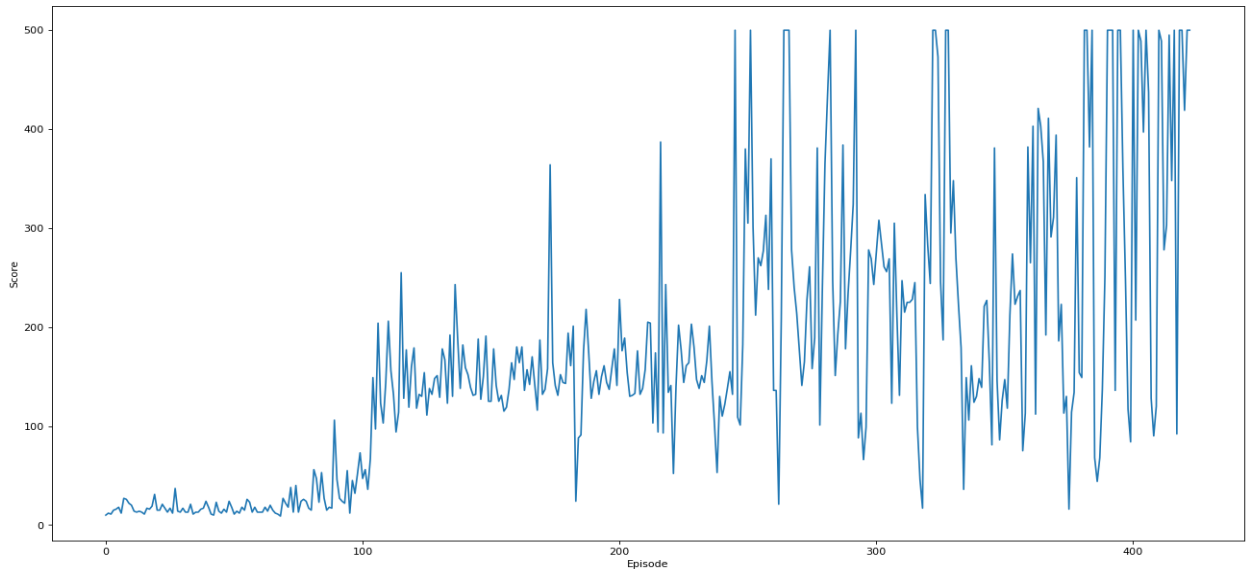


Figure 2.4: Training graph for Task1

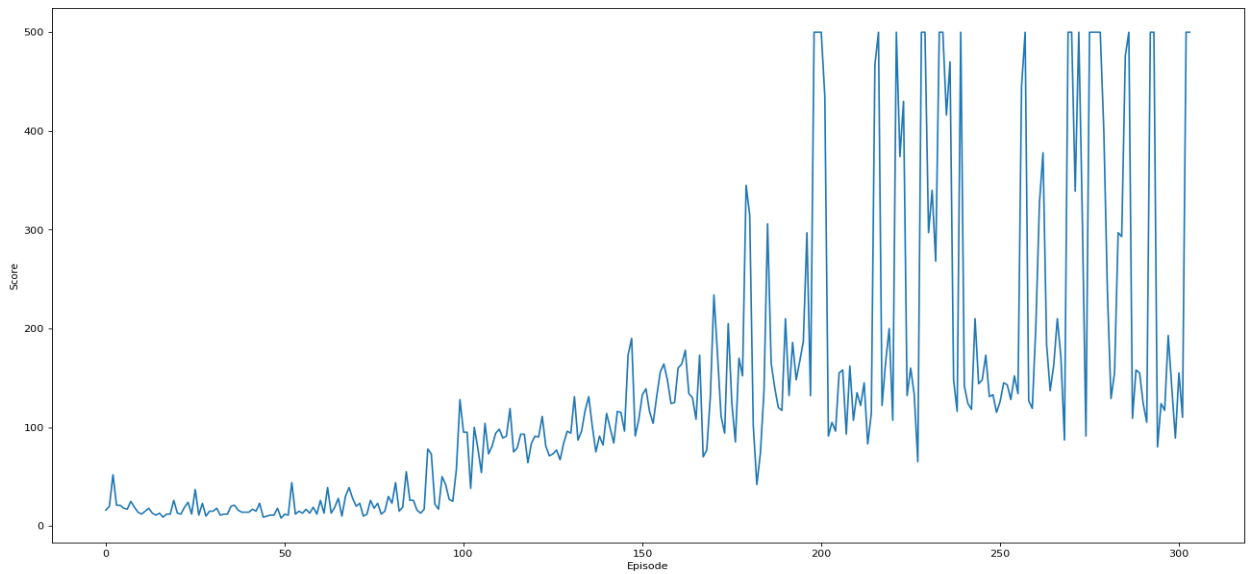


Figure 2.5: Training graph for Task2

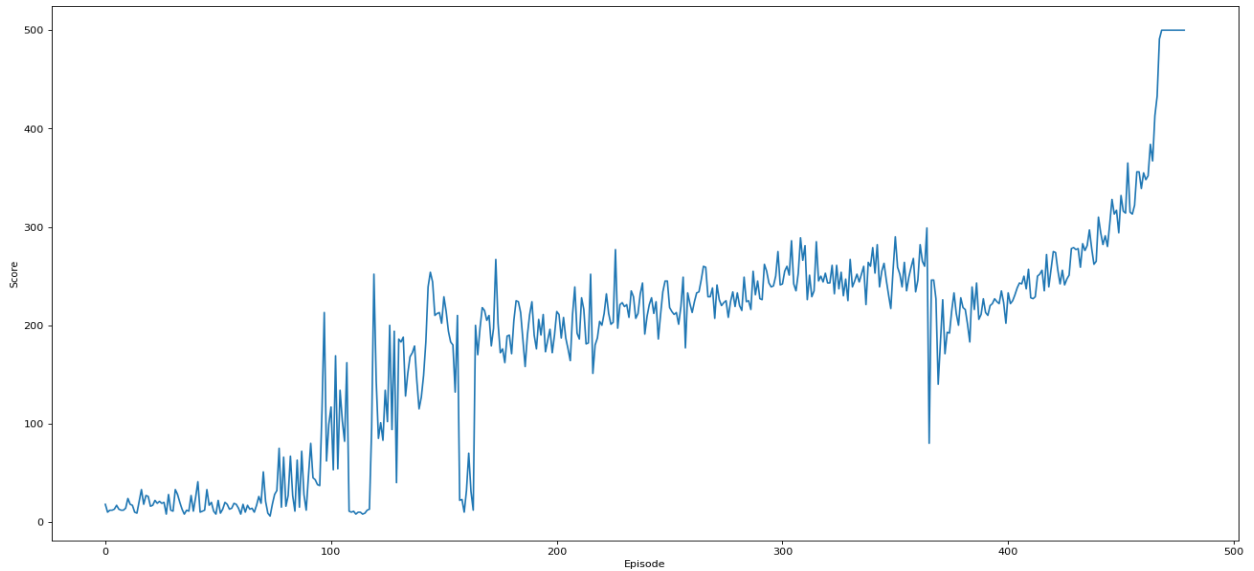


Figure 2.6: Training graph for Task3

For Task3, the first hidden layer was of size 512. Making this a model of 30k parameters.

All three models gave an average score of 500 on testing. I've included the models in the files "cartpole-dqnt1short.h5", "cartpole-dqnt2short.h5" and "cartpole-dqnt3short.h5".

The model for task 1 ie "cartpole-dqnt1short.h5" gave 500 average score on all three tasks. This was also because the model learned to stay still.

Final Model for Submission:

As the model trained for task 1, "cartpole-dqnt1short.h5" gave an average score of 500 for all three tasks, we will use that model for the final submission.

2.3 POLICY GRADIENT:OUR APPROACH

This is a Policy based method for RL Agents. In policy gradients, we parameterize a policy directly. This Policy is a probability distribution over actions. So our actions follow Policy:

$$a \sim \pi(a_t|s_t;\theta)$$

Policy Gradient's features

- optimize return directly
- work in continuous and discrete action spaces
- works better in high-dimensional action spaces
- usually on-policy - meaning it is hard to escape the bias of a bad initial policy

How does Policy Gradient Methods work ?

- We have a parameterized policy
 - a neural network that outputs a distribution over actions
- How do we improve it - how do we learn?
 - change parameters to take actions that get more reward
 - change parameters to favor probable actions
- Reward function is not known
 - but we can calculate the gradient of the expected reward
 - We can use different ways to estimate G_t . In our approach, we have used a simple Monte Carlo method to estimate the expected reward's gradient.

$$\nabla_{\theta} \mathbb{E}[G_t] = \mathbb{E}[\nabla_{\theta} \log \pi(a|s) \cdot G_t]$$

Figure 2.7: Equation for learning gradients of expected reward

Models

We tried different FNNs for our Policy, but the best results were given by the models described here 3.1. As we increase the number of layers in the model for policy moving, the average of the reward is not increasing and gets stuck near 20-30. This is essential because a large network needs large data to be trained on. But one hidden layer network gave great results on task1 and task2. And for task 3, as there is more randomness, we need a larger model to learn those things, and we used two layered models for that. We used the ideas of

[3, 1] these papers as well to build our model.

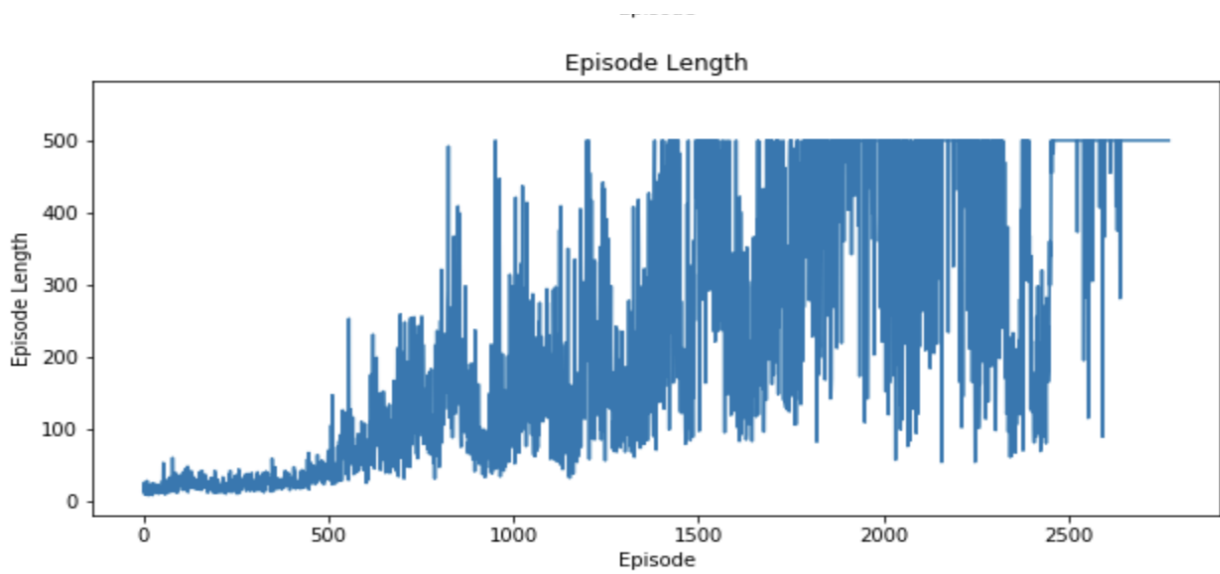


Figure 2.8: training graph for Task1

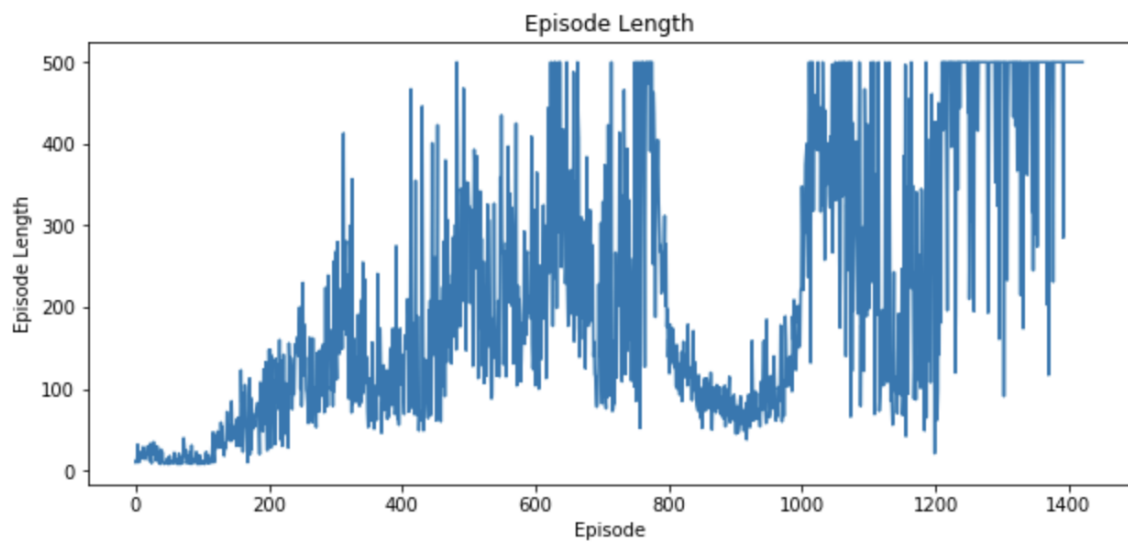


Figure 2.9: training graph for Task2

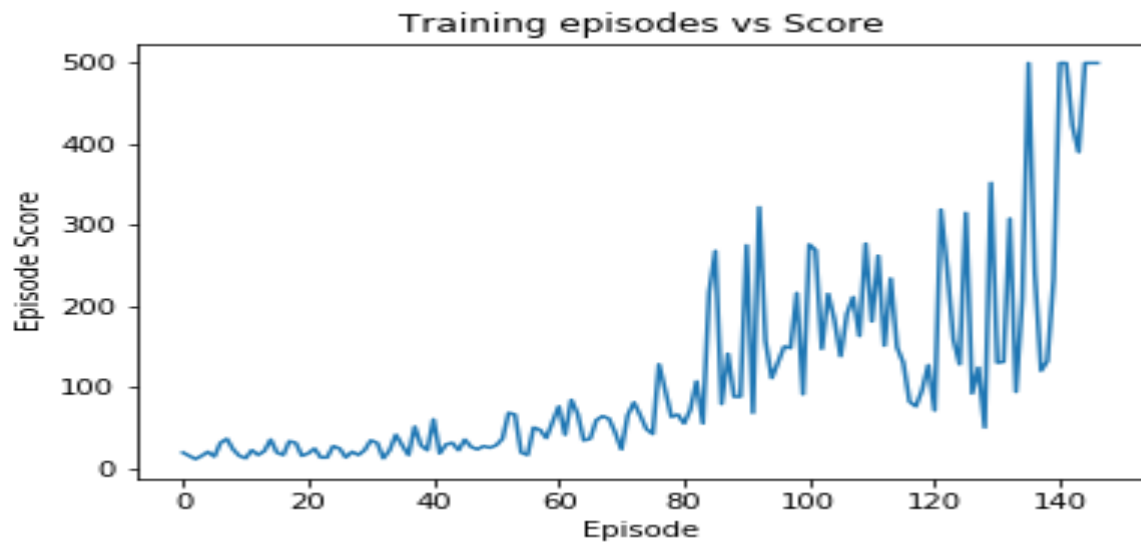


Figure 2.10: training graph for Task3

3 RESULTS

This table gives the combined results of the discussion in the previous section.

Techniques ->	DQN	Policy Gradient
Task1 Performance	500	495
Task1 Model Complexity	(50,64)	(128)
Task2 Performance	500	495
Task2 Model Complexity	(50,64)	(256)
Task3 Performance	500	495
Task3 Model Complexity	(50,64)	(128,32)

Table 3.1: Final result table: Task x performance gives an average score of 100 episodes while training phase, Task x Model Complexity gives hidden layer size in the final model for that task

4 REFERENCES OTHER THAN RESEARCH PAPERS

- Course by David Silver(DeepMinds)
- Video series on RL(DQN) by sendex(youtube channel)
- Andrej Karpathy blog(Deep Reinforcement Learning: Pong from Pixels)
- pythonprogramming.net(tutorials for RL)
- Sutton & Barto(Holy Book for RL)

REFERENCES

- [1] Savinay Nagendra, Nikhil Podila, Rashmi Ugarakhod, and Koshy George. Comparison of reinforcement learning algorithms applied to the cart-pole problem. In *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 26–32. IEEE, 2017.
- [2] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.
- [3] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. 2014.