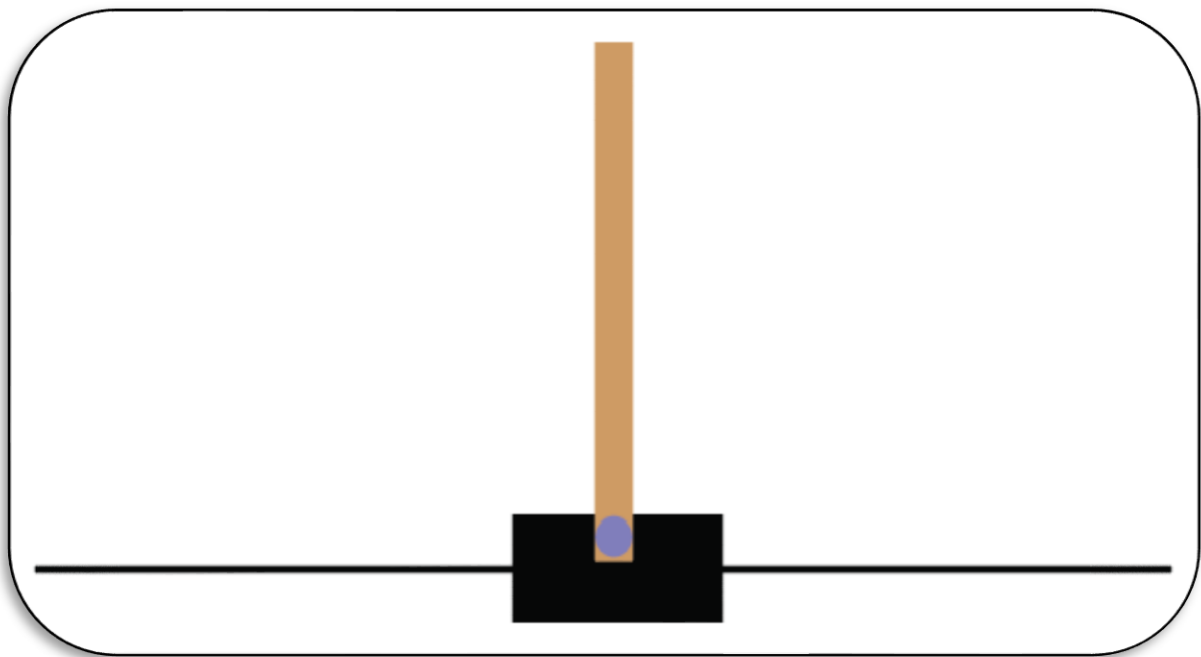


Balancing Cart Pole in Open AI Gym using Deep Reinforcement Learning

The Ultimate Guide for Implementing a Cart Pole Game using Python, Deep Q Network (DQN), Keras and Open AI Gym.



Keras



OpenAI

Blog By: Chaithanya Vamshi Sai

Student Id: 21152797

Introduction

Reinforcement Learning is a type of machine learning that allows us to create AI agents that learn from their mistakes and improves their performance in the environment by interacting to maximize their cumulative reward.

AI agents will learn it by trial and error and agents are incentivized with punishments for wrong actions and rewards for good ones.

In this blog, I will demonstrate and show how we can harness the power of how Deep Reinforcement learning (Deep Q-learning) can be implemented and applied to play a Cart Pole game using Keras, DQN Algorithm and Open AI Gym.

Steps to Implement Cart Pole Game using Keras, Deep Q Network (DQN) Algorithm and Open AI Gym

1. Problem Statement
2. Importing Libraries
3. Setting up the Environment with Open AI Gym
4. Implementing DQN Algorithm by applying LinearAnnealed - EpsGreedyQPolicy
5. Building DQN Agent with Keras-RL
6. Testing the DQN Agent for 20 Consecutive Episodes
7. Saving the Best DQN Model Weights

1. Problem Statement

The main objective of this task is to apply Deep Reinforcement learning to replace the human element in the CartPole-V0 environment in Open AI Gym environment using the Deep Q Network (DQN) algorithm.

Conditions for Cart Pole Game

- The goal is to balance the pole by moving the cart from side to side to keep the pole balanced upright.
- If the pole angle is more than 12 degrees or the cart moves by more than 2.4 units from the centre, then the game will end and if the pole remains standing for 200 steps, then the game is successful.
- Apply Linear annealed policy with the EpsGreedyQPolicy as the inner policy.
- Achieve a DQN model that trains in the least possible number of episodes.
- Balance pole on the cart for 200 steps (Maximum Reward) for 20 consecutive episodes while testing.

2. Importing Libraries

- **Keras - rl2**: Integrates with the Open AI Gym to evaluate and play around with DQN Algorithm
- **Matplotlib**: For displaying images and plotting model results.
- **Gym**: Open AI Gym for setting up the Cart Pole Environment to develop and test Reinforcement learning algorithms.
- **Keras**: High-level API to build and train deep learning models in TensorFlow.

```
#Importing Libraries
!pip install keras-rl2
!pip install gym
import gym
import rl
import matplotlib.pyplot as plt
from keras import Sequential
from keras.layers import Input, Flatten, Dense
from tensorflow.keras.optimizers import Adam
from rl.memory import SequentialMemory
from rl.agents.dqn import DQNAgent
from rl.policy import LinearAnnealedPolicy, EpsGreedyQPolicy
```

3. Setting up the Cart Pole Environment with Open AI Gym

Cart Pole is one of the simplest environments in the Open AI gym which is a collection of environments to develop and test Reinforcement learning algorithms.

The goal of the Cart Pole is to balance a pole connected with one joint on top of a moving cart. An agent can move the cart by performing a series of 0 or 1 actions, pushing it left or right.

1. Observation is an array of 4 floats that contains
 - Angular Position and Velocity of the Cart
 - Angular Position and Velocity of the Pole
2. Reward is a scalar float value
3. Action is a scalar integer with only two possible values
 - 0 — "move left"
 - 1 — "move right"

```
#Load the CartPole environment from the OpenAI Gym suite
ENV_NAME = 'CartPole-v0'
env = gym.make(ENV_NAME)
```

4. Implementing DQN Algorithm by applying Linear Annealed - EpsGreedyQPolicy

A policy defines the way an agent acts in an environment. Typically, the goal of reinforcement learning is to train the underlying model until the policy produces the desired outcome.

In this task, we will set our policy as Linear Annealed with the EpsGreedyQPolicy as the inner policy, memory as Sequential Memory because we want to store the result of actions we performed and the rewards we get for each action.

Epsilon-Greedy Policy

Epsilon-Greedy means choosing the best (greedy) option now, but sometimes choosing a random option that is unlikely (epsilon).

The idea is that we specify an exploration rate - epsilon, which is initially set to 1.

In the beginning, this rate should be the highest value because we know nothing about the importance of the Q table.

In simple terms, we need to have a big epsilon value at the beginning of Q function training and then gradually reduce it as the agent has more confidence in the Q values.

```
import rl
from rl.memory import SequentialMemory
from rl.agents.dqn import DQNAgent
from rl.policy import LinearAnnealedPolicy, EpsGreedyQPolicy

# setup experience replay buffer
memory = SequentialMemory(limit=50000, window_length=1)

# setup the Linear annealed policy with the EpsGreedyQPolicy as the inner policy
# policy used to select actions
# attribute in the inner policy to vary
# maximum value of attribute that is varying
# minimum value of attribute that is varying
# test if the value selected is < 0.05
# the number of steps between value_max and value_min
policy = LinearAnnealedPolicy(inner_policy= EpsGreedyQPolicy(), attr='eps',
                              value_max=1.0, value_min=0.1,value_test=0.05,nb_steps=10000)
```

5. Building DQN Agent with Keras-RL

a) Feed-Forward Neural Network Model for Deep Q Learning (DQN)

```
#Neural Network model for Deep Q Learning
model = Sequential()
#Input is 1 observation vector, and the number of observations in that vector
model.add(Input(shape=(1,env.observation_space.shape[0])))
model.add(Flatten())
#Hidden layers with 24 nodes each
model.add(Dense(24, activation='relu'))
model.add(Dense(24, activation='relu'))
#Output is the number of actions in the action space
model.add(Dense(env.action_space.n, activation='linear'))
#Neural Network Architecture Summary
print(model.summary())
```

b) Feed-Forward Neural Network Architecture Summary

```
#Feed Forward Neural Network Architecture Summary
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 4)	0
dense (Dense)	(None, 24)	120
dense_1 (Dense)	(None, 24)	600
dense_2 (Dense)	(None, 2)	50

```
Total params: 770
```

```
Trainable params: 770
```

```
Non-trainable params: 0
```

c) Defining DQN Agent for DQN Model

The DQN agent can be used in any environment which has a discrete action space.

The heart of a DQN Agent is a Q - Network, a neural network model that can learn to predict Q-Values (expected returns) for all actions, given an observation from the environment.

```
#Defining DQN agent
dqn = DQNAgent(model=model,          # Q-Network model
               nb_actions=env.action_space.n, # number of actions
               memory=memory,          # experience replay memory
               nb_steps_warmup=25,      # how many steps are waited before starting experience replay
               target_model_update=1e-2, # how often the target network is updated
               policy=policy)          # the action selection policy
```

d) Compiling the DQN Model

Once the layers are added to the model, we need to compile the DQN agent by using Adam Optimizer, Learning rate and evaluation metrics like MAE and Accuracy.

```
#Compile DQN Agent.
#We use Adam optimizer, learning rate and evaluation metrics

from tensorflow.keras.optimizers import Adam
dqn.compile(Adam(learning_rate=1e-3), metrics=['mae', 'accuracy'])
```

e) DQN Model Training

To start training the DQM model, we will use the “model.fit” method to train the data and parameters with nb_steps=50000.

```
#Finally fit and train the agent
history = dqn.fit(env, nb_steps=50000, visualize=False, verbose=2)
```

f) Summary of Training episode steps and the total episodes of the DQN Model

One iteration of the Cartpole-v0 environment consists of 200-time steps.

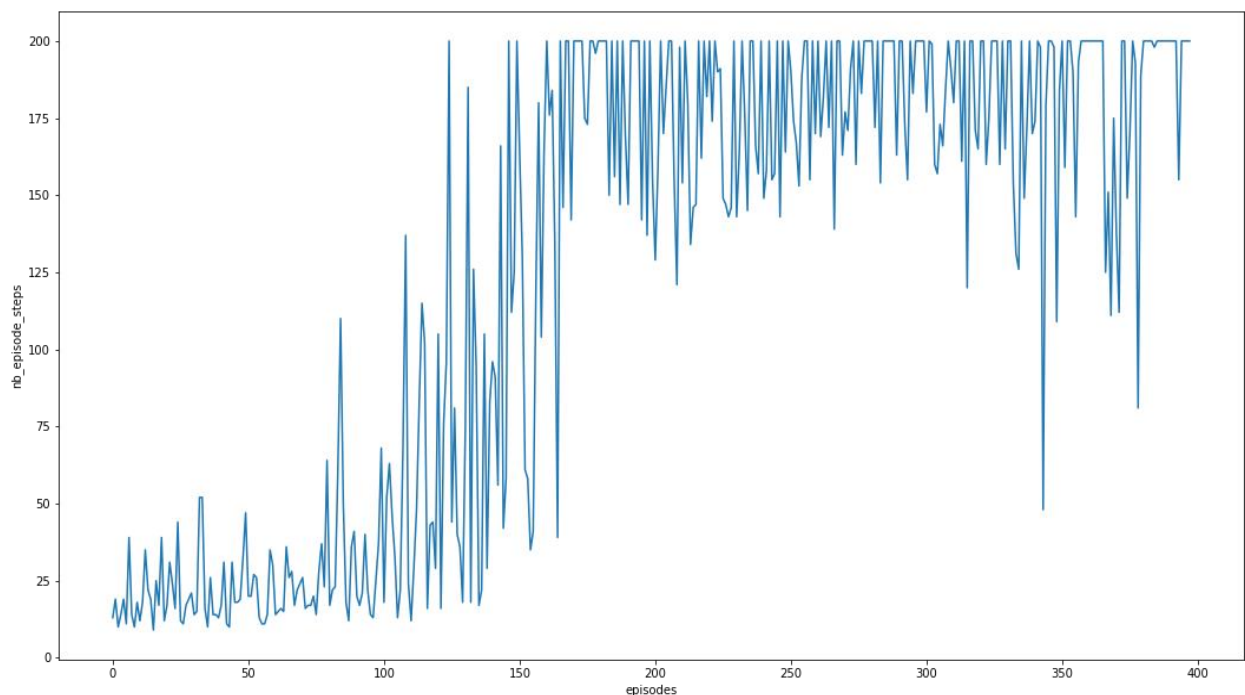
The environment gives a reward of +1 for each step the pole stays up, so the maximum return for one episode is 200.

During its training, this instance of the DQN agent was able to achieve a **maximum reward of 200 at 125 episodes**, meaning it reached 200 steps without dropping the pole with the least number of possible episodes while training.

From the graph, we can observe how in the first episodes, the rewards stay low as the agent is still exploring the state-space and learning the values for each state-action pair. However, as we complete more episodes, the agent's performance keeps improving and more episodes are completed.

Python Implementation of Balancing Cart Pole using Keras, DQN and Open AI Gym is available on [GitHub](#)

```
47505/50000: episode: 386, duration: 1.598s, episode steps: 200, steps per second: 125, episode reward: 200.000, mean reward: 1.000
47705/50000: episode: 387, duration: 1.609s, episode steps: 200, steps per second: 124, episode reward: 200.000, mean reward: 1.000
47905/50000: episode: 388, duration: 1.610s, episode steps: 200, steps per second: 124, episode reward: 200.000, mean reward: 1.000
48105/50000: episode: 389, duration: 1.645s, episode steps: 200, steps per second: 122, episode reward: 200.000, mean reward: 1.000
48305/50000: episode: 390, duration: 1.600s, episode steps: 200, steps per second: 125, episode reward: 200.000, mean reward: 1.000
48505/50000: episode: 391, duration: 1.670s, episode steps: 200, steps per second: 120, episode reward: 200.000, mean reward: 1.000
48705/50000: episode: 392, duration: 1.647s, episode steps: 200, steps per second: 121, episode reward: 200.000, mean reward: 1.000
48905/50000: episode: 393, duration: 1.577s, episode steps: 200, steps per second: 127, episode reward: 200.000, mean reward: 1.000
49060/50000: episode: 394, duration: 1.229s, episode steps: 155, steps per second: 126, episode reward: 155.000, mean reward: 1.000
49260/50000: episode: 395, duration: 1.632s, episode steps: 200, steps per second: 123, episode reward: 200.000, mean reward: 1.000
49460/50000: episode: 396, duration: 1.652s, episode steps: 200, steps per second: 121, episode reward: 200.000, mean reward: 1.000
49660/50000: episode: 397, duration: 1.644s, episode steps: 200, steps per second: 122, episode reward: 200.000, mean reward: 1.000
49860/50000: episode: 398, duration: 1.663s, episode steps: 200, steps per second: 120, episode reward: 200.000, mean reward: 1.000
```

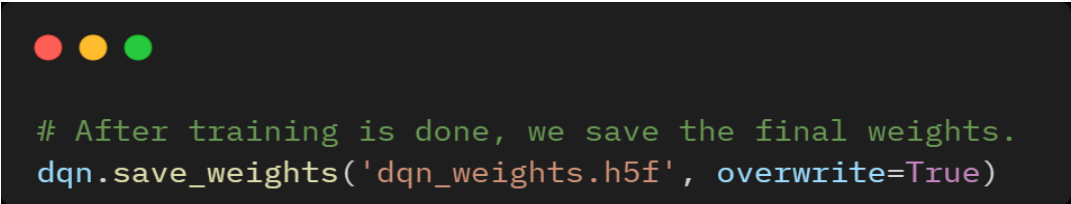


6. Testing the DQN Agent Model for 20 Consecutive Episodes

Finally, after testing 20 consecutive episode steps, DQN Model was able to achieve a maximum reward of 200 at each step without dropping the pole.

```
Testing for 20 episodes ...
Episode 1: reward: 200.000, steps: 200
Episode 2: reward: 200.000, steps: 200
Episode 3: reward: 200.000, steps: 200
Episode 4: reward: 200.000, steps: 200
Episode 5: reward: 200.000, steps: 200
Episode 6: reward: 200.000, steps: 200
Episode 7: reward: 200.000, steps: 200
Episode 8: reward: 200.000, steps: 200
Episode 9: reward: 200.000, steps: 200
Episode 10: reward: 200.000, steps: 200
Episode 11: reward: 200.000, steps: 200
Episode 12: reward: 200.000, steps: 200
Episode 13: reward: 200.000, steps: 200
Episode 14: reward: 200.000, steps: 200
Episode 15: reward: 200.000, steps: 200
Episode 16: reward: 200.000, steps: 200
Episode 17: reward: 200.000, steps: 200
Episode 18: reward: 200.000, steps: 200
Episode 19: reward: 200.000, steps: 200
Episode 20: reward: 200.000, steps: 200
<keras.callbacks.History at 0x7f9e648146d0>
```

7. Saving the Best DQN Model Weights



```
# After training is done, we save the final weights.
dqn.save_weights('dqn_weights.h5f', overwrite=True)
```

8. Conclusion

In this blog, we discussed how to implement balancing Cart Pole Game using Deep Q Network (DQN), Keras and Open AI Gym.

While this DQN agent seems to be performing well already, its performance can be further improved by applying advanced Deep Q learning algorithms like Double DQN Networks, Dueling DQN and Prioritized Experience replay which can further improve the learning process and give us better scores using an even lesser number of episodes.

9. References

- [1] <https://gym.openai.com/docs/>
- [2] <https://gym.openai.com/envs/CartPole-v0/>
- [3] https://www.tensorflow.org/agents/tutorials/1_dqn_tutorial