Cartpole - Deep Q Learning

1 - Main goal

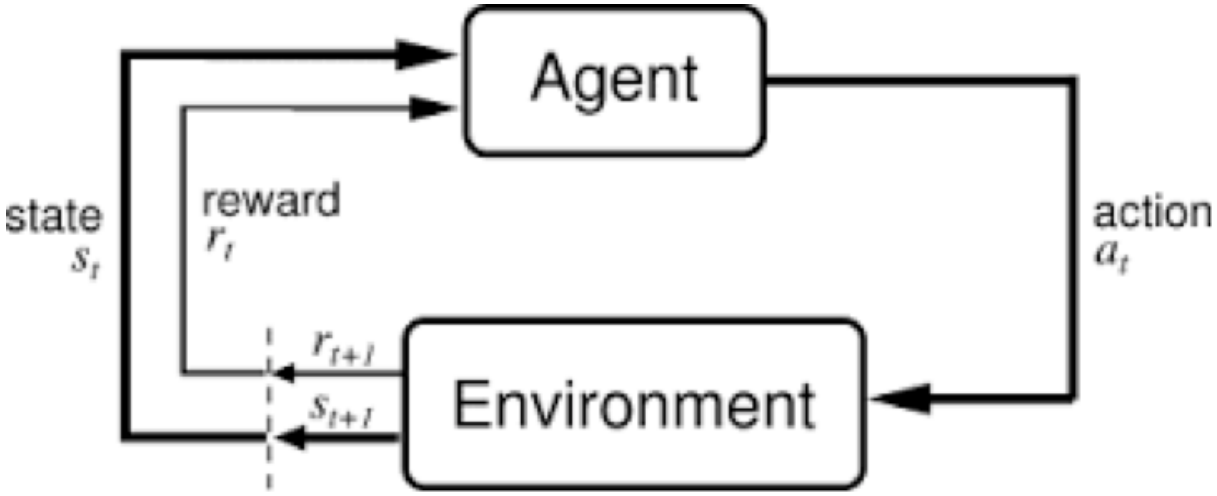
The main goal of the following snippet is to solve the cartpole(inverted pendulum) problem using Deep Q learning

2 - Related Literature

https://medium.freecodecamp.org/an-introduction-to-deep-q-learning-lets-play-doom-54d02d8017d8
https://towardsdatascience.com/cartpole-introduction-to-reinforcement-learning-ed0eb5b58288

3 - Approach and Experimentation

DQN Algorithm -



The main idea behind Q-learning is that if we had a function of the form -

Q*: State x Action -> R

that could tell us what our return would be, if we were to take an action in a given state, then we could easily construct a policy that maximizes our rewards:

$$\pi*(s)=\text{argmax } Q*(s,a) \text{ wrt } a$$

We now need to find Q*.

$$Q^{\pi} = r + \square Q^{\pi}(s', \pi(s'))$$

This is our update function, as every Q function of each policy follows Bellman's equation
And the temporal difference error

$$\delta = Q(s,a) - (r + \square \max Q(s',a))$$

And for this we use the Huber Loss.

L = $0.5a^2$ if a ≤ δ else $\delta(|a| - 0.5\delta)$

The Huber loss acts like the mean squared error when the error is small, but like the mean absolute error when the error is large - this makes it more robust to outliers when the estimates of Q are very noisy.

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

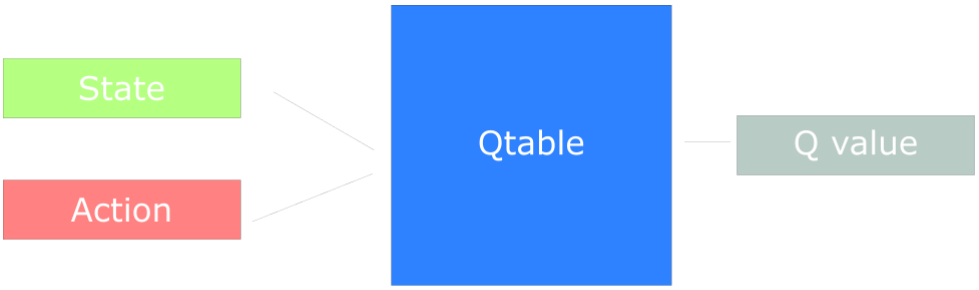
Q value for that state given that action

Expected discounted cumulative reward ...

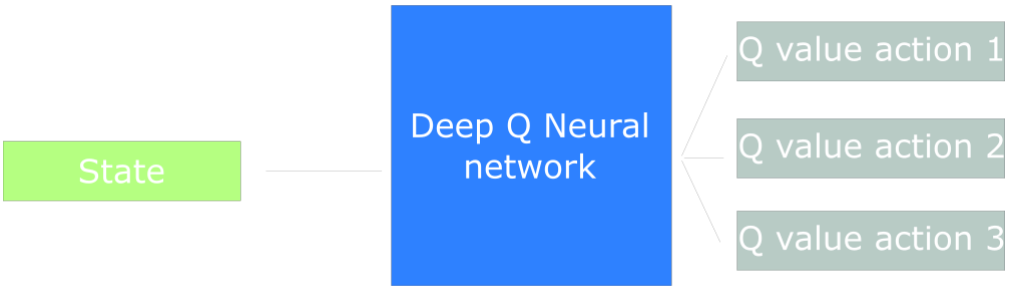
given that state and that action

Some terms -

a) Agent: An **agent** takes actions; for example, a drone making a delivery, or Super Mario navigating a video game. The algorithm is the agent.
b) Action (A): A is the set of all possible moves the agent can make. An **action** is almost self-explanatory, but it should be noted that agents choose among a list of possible actions.
c) Discount factor: The **discount factor** is multiplied by future rewards as discovered by the agent in order to dampen these rewards' effect on the agent's choice of action.
d) Environment: The world through which the agent moves. The environment takes the agent's current state and action as input, and returns as output the agent's reward and its next state.
e) Reward (R): A **reward** is the feedback by which we measure the success or failure of an agent's actions. For example, in a video game, when Mario touches a coin, he wins points.
f) Policy (π): The **policy** is the strategy that the agent employs to determine the next action based on the current state. It maps states to actions, the actions that promise the highest reward.
g) Value (V): The expected long-term return with discount, as opposed to the short-term reward R. Vπ(s) is defined as the expected long-term return of the current state under policy π.



Q* learning



Deep Q* learning

Replay Buffer - At each time step, we receive a tuple (state, action, reward, new_state). We learn from it (we feed the tuple in our neural network), and then throw this experience. As a consequence, it can be more efficient to make use of previous experience, by learning with it multiple times. Hence we create a “replay buffer.” This stores experience tuples while interacting with the environment, and then we sample a small batch of tuple to feed our neural network.

Pseudo Code -

1. Initialize Q-values ($Q(s, a)$) arbitrarily for all state-action pairs.
2. For life or until learning is stopped...
3. Choose an action (a) in the current world state (s) based on current Q-value estimates ($Q(s, \cdot)$).
4. Take the action (a) and observe the the outcome state (s') and reward (r).
5. Update $Q(s, a) := Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$

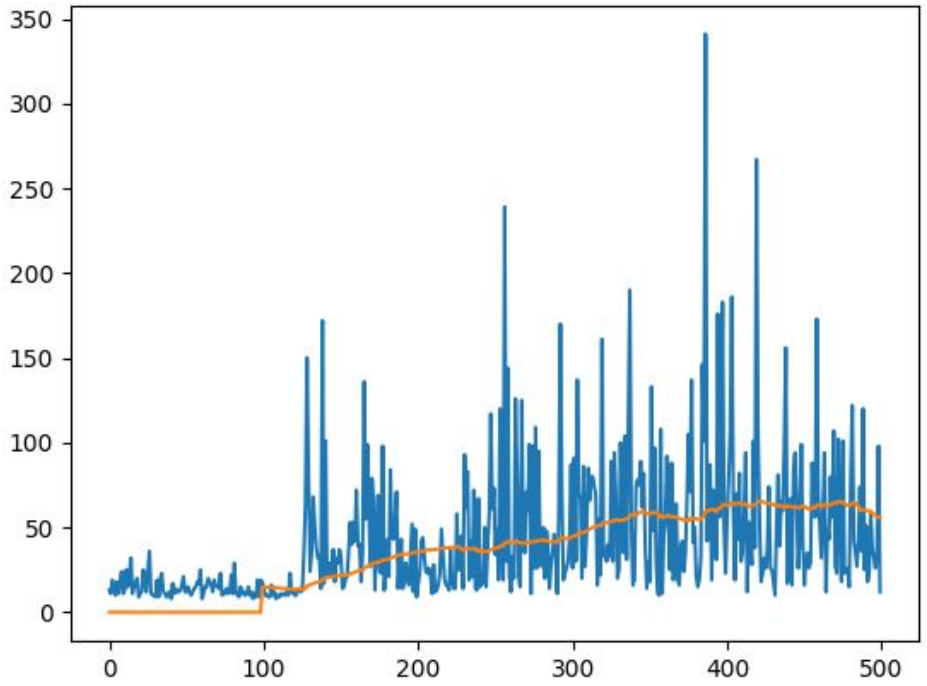
Optimization - First samples a batch, then concentrates all tensors, computes Q(s,a), computes V-Q, and then finds the loss. We also use target network to compute $v(s_{t+1})$ for stability. The target network has its weights kept frozen most of the time, but is updated with the policy network’s weights every so often. This is usually a set number of steps but we shall use episodes for simplicity.

Torchvision -

We have used torchvision.transform. This composes several transforms together, and make it easy to compose image transforms.
We have used these functions -

a) ToPIL - Convert a tensor or an ndarray to PIL Image.
b) Resize - Used for resizing
c) ToTensor - Converts PIL,ndarray into a tensor

Training -



The graph is Time vs Episodes, and the yellow line shows mean for the 100 previous episodes.We see that we hit a mean reward of 50, with a maximum time of 343.

Exact code details -

Run on python 3.5.2 with gym environment. A 500 episode takes nearly 17 minutes to run

The most difficult part - Optimization, and getting the best convolution parameters as well as starting parameters. Another issue face was installing of all packages on Windows

** The code is very similar to one on the pytorch tutorial, as it was the initial start point, but the code was built from scratch, with the understanding of each concept, and iterations. Only the initial starting values were taken from there