Reinforcement Learning with Discrete and Continuous State Spaces

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Thursday 28th April, 2022 10:25

Preamble

Reinforcement Learning as a research subject owes its origins to the study of behaviorism in psychology. The behaviorists believe that, generally speaking, our minds are shaped by the reward structures in a society. B. F. Skinner, a famous American psychologist of the last century, is considered to be the high priest of behaviorism.

Unfortunately for the behaviorists, their research was dealt a serious blow by Noam Chomsky. Through his investigations in languages, Chomsky came to the conclusion that the most fundamental structures in our minds are shaped by our biology regardless of where we are born (implying that regardless of the reward structures in a society).

In machine learning, reinforcement learning as a topic of investigation owes its origins to the work of Andrew Barto and Richard Sutton at the University of Massachusetts. Their 1998 book "Reinforcement Learning: An Introduction" (for which a much more recent second-edition is now available on the web) is still a go-to source document for this area.

Preamble (contd.)

The goal of this lecture is to introduce you to some of the basic concepts in reinforcement learning (RL).

Traditional reinforcement learning has dealt with discrete state spaces. Consider, for example, learning to play the game of tic-tac-toe. We can refer to each legal arrangement of X's and O's in a 3×3 grid as defining a state. One can show that there is a maximum of 765 states in this case. (See the Wikipedia page on "Game Complexity".)

More recently it's been shown that neural-network based formulations of RL can be used to solve the learning problems when the state spaces are continuous and when a forced discretization of the state space results in unacceptable loss in learning efficiency.

The primary focus of this lecture is on what is known as Q-Learning in RL. I'll illustrate Q-Learning with a couple of implementations and show how this type of learning can be carried out for discrete state spaces and how, through a neural network, for continuous state spaces.

Outline

Outline

Reinforcement Learning — Some Basic Terminology

- Central to the vocabulary of reinforcement learning (RI) are: *agent*, *environment*, *goal*, *state*, *action*, *state transition*, *reward*, *penalty*, and *episode*
- At any given time, the *environment* is in a particular *state*.
- In general, when the *agent* takes an *action*, that can change the state of the *environment* and elicit a *reward* or a *penalty* from the environment.
- A sequence of state transitions caused by the agent's actions constitutes an episode if there does not exist a state transition from the terminal state in the sequence.

Basic Terminology (contd.)

- As mentioned on the previous slide, the agent receives a reward from the environment for each action taken. The reward may be positive or negative.
- The goal of reinforcement learning is for the agent to learn to maximize the rewards received from the environment during each episode.
- At all times, the *agent* is aware of the *state* of the *environment*. The agent also knows what *actions* it has at its disposal for changing that state. But what the agent does not know in advance is what action to deploy in each state in order to maximize its rewards from the environment. That it must learn on its own.
- During training, the environment *rewards* or *penalizes* the agent based on the state of the environment resulting from the action taken by the agent.

The Notation

- Initially, the agent chooses the actions randomly. Subsequently, based on the rewards received, it learns the best choice for an action in each state of the environment.
- It is common to represent all possible states of the environment by the set $S = \{s_1, \ldots, s_n\}$ and the set of actions that agent can execute in order for the environment to transition from one state to another by $A = \{a_1, a_2, \ldots, a_m\}$.
- I'll use the symbol σ_t to denote the state of the environment at time t. Obviously, $\sigma_t \in S$. The action taken by the agent at time t would be denoted α_t. Obviously, again, $\alpha_t \in A$.
- The action α_t taken by the agent at time step t will cause the state of the environment to change to $\sigma_{t+1} \in S$ for the time step $t + 1$.
- In response to this state transition, the environment will grant the agent a reward r_{t+1}. A reward is typically a small scalar value.

The State, Action, Reward Sequence

- When a reward is a negative number, you can construe that to be a penalty. It is the environment's job to watch how the system is behaving and to provide the agent with positive and negative rewards in response to the consequences of each action.
- After receiving the reward r_{t+1} while the environment is in state σ_{t+1}, the agent takes the action $\alpha_{t+1} \in A$. This action will cause the state of the environment to change to $\sigma_{t+2} \in S$. And that will be followed by the environment granting the agent a reward r_{t+2}, and so on.
- Assuming that the agent starts learning at time $t = 0$, we can think of the following sequence of states, actions, and rewards:

$$\sigma_0, \alpha_0, r_1; \quad \sigma_1, \alpha_1, r_2; \quad \sigma_2, \alpha_2, r_3; \quad .....$$

Visualizing the Learning Processing

- An agent learning in this manner in response to the rewards received from the environment may be visualized in the following manner:

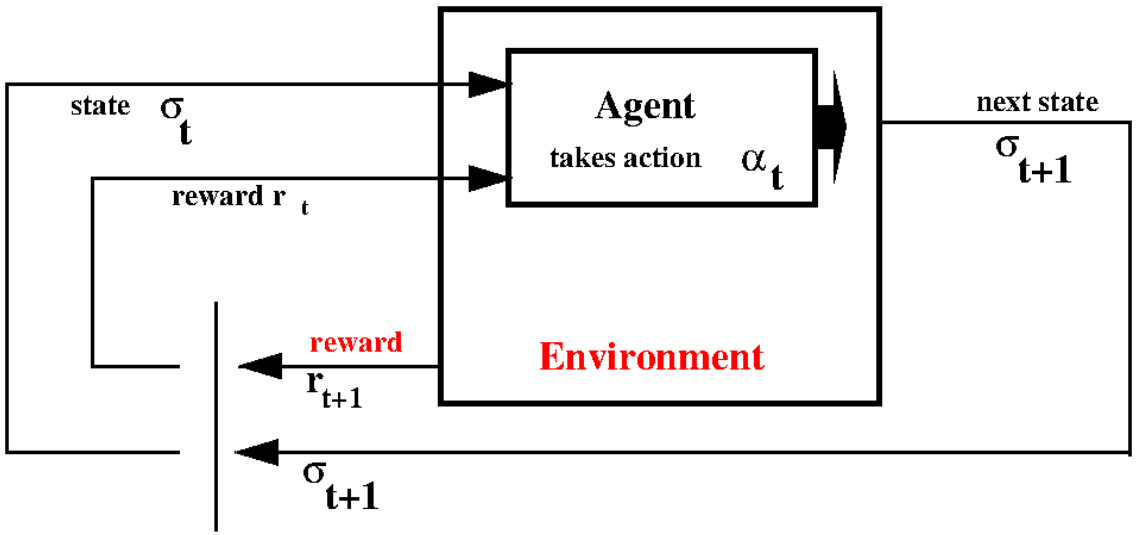

Figure: The Agent's action $\alpha_t \in A$ when the environment is in state $\sigma_t \in S$ results in the state transitioning to σ_{t+1} and the environment granting the agent a reward r_{t+1} (which can be negative).

Outline

A Stochastic RL Agent

- The notation of Reinforcement Learning (RL) I presented in the previous section was sterile — in the sense that it might have created the impression that the relationships between the states, the actions, and the rewards were deterministic and designed to guarantee success.
- Such determinism cannot model the real-life scenarios in which one would want to use RL.
- You see, the environment, while being a source of rewards for the agent, has intractable aspects to it that can cause random perturbations in the behavior of whatever it is that the agent is trying to master through reinforcement learning.
- How to best model the resulting uncertainties is an area of research unto itself. The models that have become the most popular are based on the usual Markovian assumption as shown on the next slide.

The Markovian Assumption

- With the Markovian assumption, you have a stochastic RL agent for which the rewards received from the environment and also the state transition achieved by the environment at time $t+1$ depend probabilistically on just on the state/action pair at time t.
- To be more precise, the state/action pair at time t completely dictates a probability distribution for the next-state/reward pair at time $t+1$. We denote this probability as follows:

$$prob(\sigma_{t+1}, r_{t+1} \mid \sigma_t, \alpha_t)$$

- Such an agent along with the environment is known as a Markov Decision Process (MDP).
- Such a probability distribution would allow us to estimate the conditional probabilities related to the state transitions:

$$prob(\sigma_{t+1} \mid \sigma_t, \alpha_t) = \sum_r prob(\sigma_{t+1}, r \mid \sigma_t, \alpha_t)$$

Future Expected Reward

- In a similar manner, we can also compute the expected reward at time $t+1$:

$$E(r_{t+1} \mid \sigma_t, \alpha_t) = \sum_r r \sum_{\sigma_{t+1}} prob(\sigma_{t+1}, r \mid \sigma_t, \alpha_t)$$

  where the outer summation on the right is with respect to all possible reward values when the environment is in state σ_t and the agent is invoking action α_t. The inner summation is with respect to all possible target states that the system can get to from the state σ_t.

- With respect to time t, the estimate $E(r_{t+1} \mid \sigma_t, \alpha_t)$ is a future expected reward for time $t+1$.

- In general, the agent's policy should be such that the action taken in each state maximizes the **future expected reward** from the environment.

The Quality Index Q

- Toward that end, we can associate with each (s_i, a_j) pair a quality index, denoted $Q(s_i, a_j)$, that is a measure of the maximum possible value for the future expected reward from the environment for that pair. That is, by definition,

$$Q(s_i, a_j) \quad = \quad \max E\Big(r_{t+1} \;\; | \;\; \sigma_t = s_i, \alpha_t = a_j\Big)$$

- The best policy for the agent, naturally, would be to choose that action $a_j \in A$ in state $s_i \in S$ which maximizes the quality index $Q(s_i, a_j)$.

- The conditioning shown on the right in the equation above means that assuming that σ_t, which is the state of the environment at time t, was s_i and assuming that α_t, the action chosen by the agent at the same time, was a_j, then if we *could* estimate the maximum possible value of the expected future reward at time $t+1$, that *would* be the value of the quality index $Q(s_i, a_j)$ at time t.

The Quality Index Q (contd.)

- The goal of RL should be to estimate the values for quality index $Q(s_i, a_j)$ for all possible state/action pairs (s_i, a_j).
- We can conceive of Q as a table with $|S|$ rows and $|A|$ columns, where $|.|$ denotes the cardinality of a set. We could place in each cell of the table the quality-index value we may conjure up for $Q(s_i, a_j)$.

Outline

What is Q-Learning?

- The probabilistic notions of the previous section serve a useful purpose in helping us articulate a principled approach to reinforcement learning. However, in practice, it is rather unlikely that we will have access to the conditional probability distribution mentioned on Slide 13 for a real life problem, not the least because of the intractable nature of the uncertainties in the interaction between the agent and the environment.

- Q-Learning is a method for implementing an RL solution for a learning agent that attempts to approximate the maximization of the future expected rewards in the absence of explicit probabilistic modeling.

- In Q-Learning, we attempt to place in the cells of the Q table the estimated values for the best expected cumulative rewards for the state-action pairs corresponding to those cells.

Maximizing the Sum of the Discounted Rewards

- Fundamentally, the entries in the cells of a Q table are the expected future rewards. For practical reasons, we must incorporate the future into these estimates on a discounted basis.
- You can think of discounting as reflecting our lack of knowledge of the conditional probability distributions mentioned in the last section. That is, we cannot be too sure about how the choice of an action in the current state will influence the future outcomes as the agent makes state-to-state transitions. With this uncertainty about the future, a safe strategy would be to deemphasize the expected rewards that are more distant into the future.
- Discounted or not, there is a troubling question here: Without any probabilistic modeling of the sort mentioned in the previous section, how can any agent look into the future and make any reasonable sense of what should go into the Q table? As to how practical implementations of Q-Learning get around this dilemma, we will get to that later.

Updating the Q Table (contd.)

- We will ignore for a moment the last bullet on the previous slide, and acknowledge the recursive nature of the definition of $Q(s_i, a_j)$ on Slide 15 when that definition is considered as a function of time. That definition is recursive because the future expected reward at t depends on its value at $t+1$ in the same manner as the future expected reward at $t+1$ depends on its value at $t+2$, and so on.
- By unfolding the recursion and incorporating the discounting mentioned on the previous slide, $Q(s_i, a_j)$ can be set to:

$$\begin{aligned} Q(s_i, a_j)\Big|_t &= \max\left(r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \ldots\right) \\ &= \max\left(\sum_{k=0}^{\infty} \gamma^k \cdot r_{t+1+k}\right) \end{aligned}$$

where γ is the discounting factor and where we intentionally do not start discounting the rewards until the time step $t+2$ (assuming that the current time is t) as a reflection of our desire to base the value of Q primarily on the next time step and of our wariness about the more distant future.

Updating the Q Table

- For the purpose of implementation, we express the update equation shown on the previous slide in the following form:

$$Q(s_i, a_j)\Big|_t \quad \Leftarrow \quad (1-\beta) * Q(s_i, a_j)\Big|_t \quad + \quad \beta * \Big[r_{t+1} + \gamma \cdot \max_{a_j \in A} \Big\{ Q(s_i, a_j)\big|_{t+1} \Big\}\Big]$$

  where the β is the learning rate. Note the assignment operator '$\Leftarrow$', which is what makes it an update formula.

- Calling on the RL Oracle to provide the agent with what would otherwise be unknowable, we can reexpress the above update formula as:

$$Q(s_i, a_j)\Big|_t \quad \Leftarrow \quad (1-\beta) * Q(s_i, a_j)\Big|_t \quad + \quad \beta * \Big[r_{t+1} + \{\text{Consult the RL Oracle for the best estimate of the future rewards}\}\Big]$$

- In the absence of consultation with an oracle, we could adopt the following strategy: The agent would start with random entries in the Q table. For obvious reasons, at the beginning the agent will very quickly arrive at the terminal states that indicate failure.

Updating the Q Table (contd.)

- Each failure will result in updating the Q table with negative rewards for the cells that cause transitions to such terminal states. This would eventually — at least one would so hope — result in a Q table that would reflect positive learning.

- To elaborate further, for any state transition to failure, the value of the reward r_{t+1} in an update formula of the type shown on the previous slide would be a large negative number. That would push down the entry in the $Q(s_i, a_j)$ table that led to such a transition, making it less likely that the transition would be taken the next time.

- In algorithm development, we talk about "time-memory tradeoff". What we have here is a "time-foresight tradeoff". [About what's meant by "time-memory" tradeoff: What that means is that you can solve a problem by having memorized or theoretically generated all of your solution paths and then, when confronted with a new instance of the problem, you simply look up the memorized strategies. That would be referred to as a purely memory based solution. An alternative approach would consist of only remembering the decision points in seeking out a solution to the problem. Then, when confronted with a new instance of the problem, you would step through the solution space and test which way you should turn at each decision point. In this case, you would be using lesser memory, but it would take longer to generate a solution.]

Updating the Q Table (contd.)

- I say "time-foresight tradeoff" because we do not have access to an oracle. We therefore choose to stumble into the future anyway through the state-transition table and learn from our mistakes as recorded in the evolving Q table. It may take time, but, hopefully, we'd still be able to solve the problem.
- For stumbling forward in the learning process, we rewrite the formula on Slide 21 as follows in which we estimate the future rewards by stepping through the state transition table and accumulating the future rewards by looking up the Q table:

$$Q(s_i, a_j)\Big|_t \Leftarrow (1-\beta)*Q(s_i, a_j)\Big|_t + \beta*\{\text{Estimate discounted future reward from Q-table and state-transition table}\}$$

- This update formula lends itself to an actual implementation in which you start with random entries in the Q table and, through repeated trials, you converge to the table that reflects positive learning. I have demonstrated this with the two implementations that follow in this lecture, one based on a discrete state space and the other on a continuous state space.

Outline

The Cart-Pole Example

- To explain Q-Learning with an example, consider what's known as the Cart Pole problem. The Cart-Pole problem is to RL what fruit-fly is to genetics research.
- In the Cart Pole problem, you have a cart that can move back and forth on a linear track. Mounted on the cart is a hinged pole. The hinge allows the pole to turn in the same plane that corresponds to the translational motions of the cart.
- The goal is to keep the pole standing upright and, should the pole lean one way or the other, the cart should move in such a way that counters the lean of the pole.
- Considering how unstable the pole would be standing upright, it would be difficult to construct a probabilistic framework for this problem along the lines described earlier in this lecture.

The Cart-Pole Example (contd.)

- The state-space for the Cart Pole problem is obviously continuous since the pole can lean to any degree.
- The question we pursue in this section is whether it is possible to discretize the state space and to then use the Q-Learning based RL as described in the previous section.
- Shown below for illustration is a 5-state discretization of the state space. Each entry represents the lean of the pole one way or the other, but within a range appropriate to the entry:

```
S  =  {  hard_lean_left,
         soft_lean_left,
         soft_lean_right,
         hard_lean_right,
         fallen
      }
```

The Cart-Pole Example (contd.)

- Here is a possible mapping between the next-state achieved and the ensuing reward as required by the update formula shown on Slide 21:

```
fallen              =>   -1
soft_lean_left      =>   +1
soft_lean_right     =>   +1
hard_lean_left      =>    0
hard_lean_right     =>    0
```

- And here are the two actions the agent could invoke on the cart in order to balance the pole:

```
A  = {  push_left,
        push_right
     }
```

- Now that you can appreciate what I mean by the discretization of the Cart-Pole's state space, it's time to actually look at some code that implements this type of Q-Learning.

A Python Implementation of Q-Learning for the Cart-Pole Problem

- Starting with Slide 35, what you see is Rohan Sarkar's Python implementation of Q-Learning for solving a discretized version of the Cart-Pole problem. Rohan is working on his Ph.D. in RVL.
- In line (A), the code starts by initializing the various parameters and the environment variables.
- The function whose definition starts in line (B) creates a discretized set of states for the Cart Pole. The discretization is a much finer version of what I illustrated on the previous slide.
- The purpose of the function `optimal_action()` defined in line (C) is to return the best choice of action when the agent is in a given state s. This function implements what is known as the *epsilon-greedy* policy for choosing the best action in any given state. [This policy says that the earlier phases of training should make a choice between choosing an action randomly from all available actions in a given state and the action offered by the Q table. Furthermore, the policy says that as the training progresses we want to increasingly depend on just the Q table for choosing the best action. The parameter `eps_rate` in the header of the function plays a key role in implementing the epsilon-greedy policy. What is commonly referred to as 'training' in general machine learning is called 'exploration' in RL. And, what is commonly referred to as 'testing' in general machine learning is referred to as 'exploitation' in RL. During exploration, the value of `eps_rate` changes with each episode as follows: (1) we first multiply the current value of `eps_rate` by `EXPLORATION_DECAY`; and then (2) we assign to `eps_rate` the larger of `EXPLORATION_MIN` and the current value of `eps_rate`. When you see how `eps_rate` is used in the logic of `optimal_action()`, you will realize that that function is indeed implementing the epsilon-greedy policy. As `eps_rate` becomes ever smaller, it will become increasingly unlikely that the randomly generated value for p will be less than the current value for `eps_rate`.]

Python Implementation of Q-Learning (contd.)

- The function `updateQValues()` defined starting in line labeled (D) is called by the exploration code that is between the lines labeled (F) and (G). All of this code is for what's known as Double Q Learning (DQL) that was first proposed in the following famous paper:

  `https://arxiv.org/abs/1509.06461`

- DQL was created to counter the over-estimation bias in what's returned for the future expected reward by the Q table. [As you already know, each row of the Q table designates the state of the agent and the different columns of the table correspond to the different available actions in each state. The recommended action returned for each state is from that cell in the corresponding row which contains the highest future expected reward. This entails carrying out a maximization operation on the values stored in the different cells. In the presence of noise in the estimated rewards, this maximization is biased in the sense that it is an overestimate of the future expected reward for the chosen action.]

- DQL calls for using two Q tables, denoted $Q1$ and $Q2$ in the code between the lines labeled E and F, that are trained simultaneously with the idea that since the successive random training inputs are likely to span the permissible range, updating $Q1$ and $Q2$ alternately with the inputs will be such that the average of the rewards returned by the two tables is likely to possess reduced bias.

Python Implementation of Q-Learning (contd.)

- The testing part of the code (the exploitation part) comes after line G.
- I still have not said anything about how the agent interacts with the "environment." Actually, the role of the "environment" is a bit more complicated than what was depicted on Slide 10. That depiction only applies after an agent has been trained.
- In the code that follows, you will see the following calls that stand for the agent interacting with the environment:

```
env = gym.make('CartPole-v0')                          # at the beginning of the code file
action = env.action_space.sample()                     # in function optimal_action()
state = env.reset()                                    # in __main__
next_state, reward, done, info = env.step(action)      # in __main__
```

- In order to convey what is accomplished by the above statements, you need to understand how the Cart-Pole environment is actually implemented in code:

https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py

Python Implementation of Q-Learning (contd.)

- The code at the link at the bottom of the previous slide actually carries out a rudimentary physics based simulation of a Cart-Pole. It makes certain assumptions about the mass of the cart vis-a-vis that of the pole, about the dimensions of the cart and the pole, about the movement of the cart being frictionless, etc. The state of the cart-pole is a list of the following four elements:

```
[cart_position, cart_velocity, pole_angle, pole_angular_velocity]
```

- In terms of these elements and also in terms of the number of timesteps in any single episode during training or testing, the cart-pole is considered to have reached the terminal state (that is, when the pole is considered to have "fallen") when any of the following conditions is satisfied:
  - the pole angle is greater than 12 degrees
  - the cart position is greater than 2.4 units
  - the episode length is greater than 200 timesteps

Python Implementation of Q-Learning (contd.)

- An agent can invoke only two actions on the cart-pole: move left or move right. In each action, 10 units of force is applied to the cart in the desired direction of the movement.
- The environment also assumes that the time interval between successive interactions with it is 0.02 seconds regardless of the actual clock time involved.
- But what about the reward that is supposed to be issued by the environment? The environment issues a reward of 1 for all interactions with it. However, when the terminal condition is reached (meaning when we may think of the pole as having fallen), the environment returns that condition as "done". It is for the user program to translate "done" into a negative reward.
- In the sense indicated above, the environment itself does not judge the consequences of the actions called by the agent.

Python Implementation of Q-Learning (contd.)

- Regarding the rewards, it is for the RL logic being used by the agent to decide what rewards to associate with the actions, including the terminal action.
- Going back to the statements that are used in the code shown next in order to interact with the environment, the call `env.action_space.sample()` returns either 0 (which is the action for "move left") or 1 (which is the action for "move right") by choosing randomly between the two.
- The call `env.reset()`, which is typically how you start an episode, returns four random numbers for the four elements of the state, with numbers being uniformly distributed between -0.05 and +0.05.
- Finally, it is the call `next_state, reward, done, info = env.step(action)` that defines the interactions between the agent and the environment on an on-going basis.

Python Implementation of Q-Learning (contd.)

- With regard to the call `env.step(action)` mentioned at the bottom of the previous slide, at each time step during an episode, the agent calls on the environment with an action, which in our case is either 0 or 1, and the environment returns the 4-tuple `[next_state, reward, done, info]` where `done` would be set to the Boolean `True` if the cart-pole has reached the terminal condition.
- In the 4-tuple that is returned by `env.step(action)`, the `next_state` is again a 4-tuple of numbers standing for the 4 real numbers in the list `[cart_position, cart_velocity, pole_angle, pole_angular_velocity]`.

Python Implementation of Q-Learning (contd.)

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

##  Cartpole_DQL.v2.py

##  Balancing a cartpole using Double Q-learning (without a neural network) and using
##  a discretized state space. The algorithm is trained using epsilon-greedy approach
##  with Replay Memory
##
##  @author: Rohan Sarkar (sarkarr@purdue.edu)

import gym
import numpy as np
import time
import matplotlib.pyplot as plt
import sys
import random

seed = 0
random.seed(seed)
np.random.seed(seed)

## Initialize parameters and environment variables:
env = gym.make('CartPole-v0')                                                    ## (A)
GAMMA = 0.85
#MAX_ITER = 500
MAX_ITER = 1000
EXPLORATION_MAX = 1.0
EXPLORATION_MIN = 0.01
EXPLORATION_DECAY = 1+np.log(EXPLORATION_MIN)/MAX_ITER
REP_MEM_SIZE = 100000
MINIBATCH_SIZE = 64
N_states = 162
N_actions = 2
Q1 = np.zeros((N_states, N_actions))
Q2 = np.zeros((N_states, N_actions))
alpha = 0.001
eps_rate = EXPLORATION_MAX
cum_reward = 0
train_reward_history = []

## Map the four dimensional continuous state-space to discretized state-space:
## Credit: http://www.derongliu.org/adp/adp-cdrom/Barto1983.pdf.
def map_discrete_state(cs):                                                      ## (B)
    ds_vector = np.zeros(4);
    ds = -1
    # Discretize x (position)
    if abs(cs[0]) <= 0.8:
        ds_vector[0] = 1
    elif cs[0] < -0.8:
        ds_vector[0] = 0
    elif cs[0] > 0.8:
        ds_vector[0] = 2
    # Discretize x' (velocity)
    if abs(cs[1]) <= 0.5:
        ds_vector[1] = 1
    elif cs[1] < -0.5:
        ds_vector[1] = 0
    elif cs[1] > 0.5:
        ds_vector[1] = 2
```

(Continued on the next slide .....)

Python Implementation of Q-Learning (contd.)

(..... continued from the previous slide)

```
    # Discretize theta (angle)
    angle = 180/3.1428*cs[2]
    if -12 < angle <= -6:
        ds_vector[2] = 0
    elif -6 < angle <= -1:
        ds_vector[2] = 1
    elif -1 < angle <= 0:
        ds_vector[2] = 2
    elif 0 < angle <= 1:
        ds_vector[2] = 3
    elif 1 < angle <= 6:
        ds_vector[2] = 4
    elif 6 < angle <= 12:
        ds_vector[2] = 5
    # Discretize theta' (angular velocity)
    if abs(cs[3]) <= 50:
        ds_vector[3] = 1
    elif cs[3] < -50:
        ds_vector[3] = 0
    elif cs[3] > 50:
        ds_vector[3] = 2
    ds = int(ds_vector[0]*54+ds_vector[1]*18+ds_vector[2]*3+ds_vector[3])
    return ds

## Return the most optimal action and the corresponding Q value:
def optimal_action(Q, state, eps_rate, env):                              ## (C)
    p = np.random.random()
    # Choose random action if in 'exploration' mode
    if p < eps_rate:
        action = env.action_space.sample()       # choose randomly between 0 and 1
    else:
        # Choose optimal action based on learned weights if in 'exploitation' mode
        action = np.argmax(Q[state,:])
    return action

##  Update Qvalues based on the logic of the Double Q-learning:             ## (D)
def updateQValues(state, action, reward, next_state, alpha, eps_rate, env):
    p = np.random.random()
    if (p < .5):
    # Update Qvalues for Table Q1
        next_action = optimal_action(Q1, next_state, eps_rate, env)
        Q1[state][action] = Q1[state][action] + alpha * \
                (reward + GAMMA * Q2[next_state][next_action] - Q1[state][action])
    else:
    # Update Qvalues for Table Q2
        next_action = optimal_action(Q2, next_state, eps_rate, env)
        Q2[state][action] = Q2[state][action] + alpha * \
                  (reward + GAMMA * Q1[next_state][next_action] - Q2[state][action])
    return next_action

class ReplayMemory:                                                       ## (E)
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
    def push(self, s, a , r, ns):
        self.memory.append([s, a, r, ns])
        if len(self.memory) > self.capacity:
            del self.memory[0]
    def sample(self, MINIBATCH_SIZE):
        return random.sample(self.memory, MINIBATCH_SIZE)
    def __len__(self):
        return len(self.memory)
```

(Continued on the next slide .....)

Python Implementation of Q-Learning (contd.)

(..... continued from the previous slide)

```
if __name__ == "__main__":                                          ## (F)
    ## Learn the weights for Double Q learning:
    max_t = 0
    duration_history = []
    history = []
    epsilon_history = []
    repmemory = ReplayMemory(REP_MEM_SIZE)
    for i_episode in range(MAX_ITER):
        ## state is a 4-tuple: [cart_pos, cart_vel, pole_angle, pole_ang_vel]
        state = env.reset()         ###  state set to uniformly dist random bet -0.05 and 0.05
        done = False
        # Initial state and action selection when environment restarts
        state = map_discrete_state(state)
        action = optimal_action(0.5*(Q1+Q2), state, 0, env)
        t = 0
        # Decay the exploration parameter in epsilon-greedy approach.
        eps_rate *= EXPLORATION_DECAY
        eps_rate = max(EXPLORATION_MIN, eps_rate)
        while True:
            #env.render()
            next_state, reward, done, info = env.step(action)
            if done:
                reward = -10
            next_state = map_discrete_state(next_state)
            repmemory.push(state, action, reward, next_state)
            # Update Q table using Double Q learning and get the next optimal action.
            next_action = updateQValues(state, action, reward, next_state, alpha, eps_rate, env)
            # Update Q values by randomly sampling experiences in Replay Memory
            if len(repmemory)> MINIBATCH_SIZE:
                experiences = repmemory.sample(MINIBATCH_SIZE)
                for experience in experiences:
                    ts, ta, tr, tns = experience
                    updateQValues(ts, ta, tr, tns, alpha, eps_rate, env)
            state = next_state
            action = next_action
            t += 1
            if done:
                break
        history.append(t)
        if i_episode > 50:
            latest_duration = history[-50:]
        else:
            latest_duration = history
        #print(latest_duration)
        duration_run_avg = np.mean(latest_duration)
        #print(duration_run_avg)
        duration_history.append([t, duration_run_avg])
        epsilon_history.append(eps_rate)
        cum_reward += t
        if(t>max_t):
            max_t = t
        train_reward_history.append([cum_reward/(i_episode+1), max_t])
        print("\nEpisode: %d   Episode duration: %d timesteps | epsilon %f" % (i_episode+1, t+1, eps_rate))

    np.save('Q1.npy', Q1)
    np.save('Q2.npy', Q2)
    fig = plt.figure(1)
    fig.canvas.set_window_title("DQL Training Statistics")
    plt.clf()
    plt.subplot(1, 2, 1)
    plt.title("Training History")
    plt.plot(np.asarray(duration_history))
    plt.xlabel('Episodes')
    plt.ylabel('Episode Duration')
```

(Continued on the next slide .....)

Python Implementation of Q-Learning (contd.)

(..... continued from the previous slide)

```
    plt.subplot(1, 2, 2)
    plt.title("Epsilon for Episode")
    plt.plot(np.asarray(epsilon_history))
    plt.xlabel('Episodes')
    plt.ylabel('Epsilon Value')
    plt.savefig('Cartpole_DoubleQ_Learning.png')
    plt.show()                                                    ## (G)

    ##  Finished exploration. Start testing now.
    import pymsgbox
    response = pymsgbox.confirm("Finished learning the weights for the Double-Q Algorithm.  Start testing?")
    if response == "OK":
        pass
    else:
        sys.exit("Exiting")

    print("\n\n\nThe Testing Phase:\n\n")
    ##  Control the cartpole using the learned weights using Double Q learning
    play_reward =0
    for i_episode in range(100):
        observation = env.reset()
        done = False
        # Initial state and action selection when environment restarts
        state = map_discrete_state(observation)
        action = optimal_action(0.5*(Q1+Q2), state, 0, env)
        t = 0
        eps_rate = 0
#        time.sleep(0.25)
        while not done:
            env.render()
            time.sleep(0.1)
            next_state, reward, done, info = env.step(action)
            next_state = map_discrete_state( next_state )
            next_action = optimal_action(Q1, state, 0, env)
            state = next_state
            action = next_action
            t += 1
        play_reward += t
        print("Episode ",i_episode," Episode duration: {} timesteps".format(t+1))
#        print("Episode ",i_episode," : Exploration rate = ", eps_rate, " Cumulative Average Reward = ", play_reward/(i_episode+1))

    env.close()
```

Visualizing the Training Phase for DoubleQ Learning

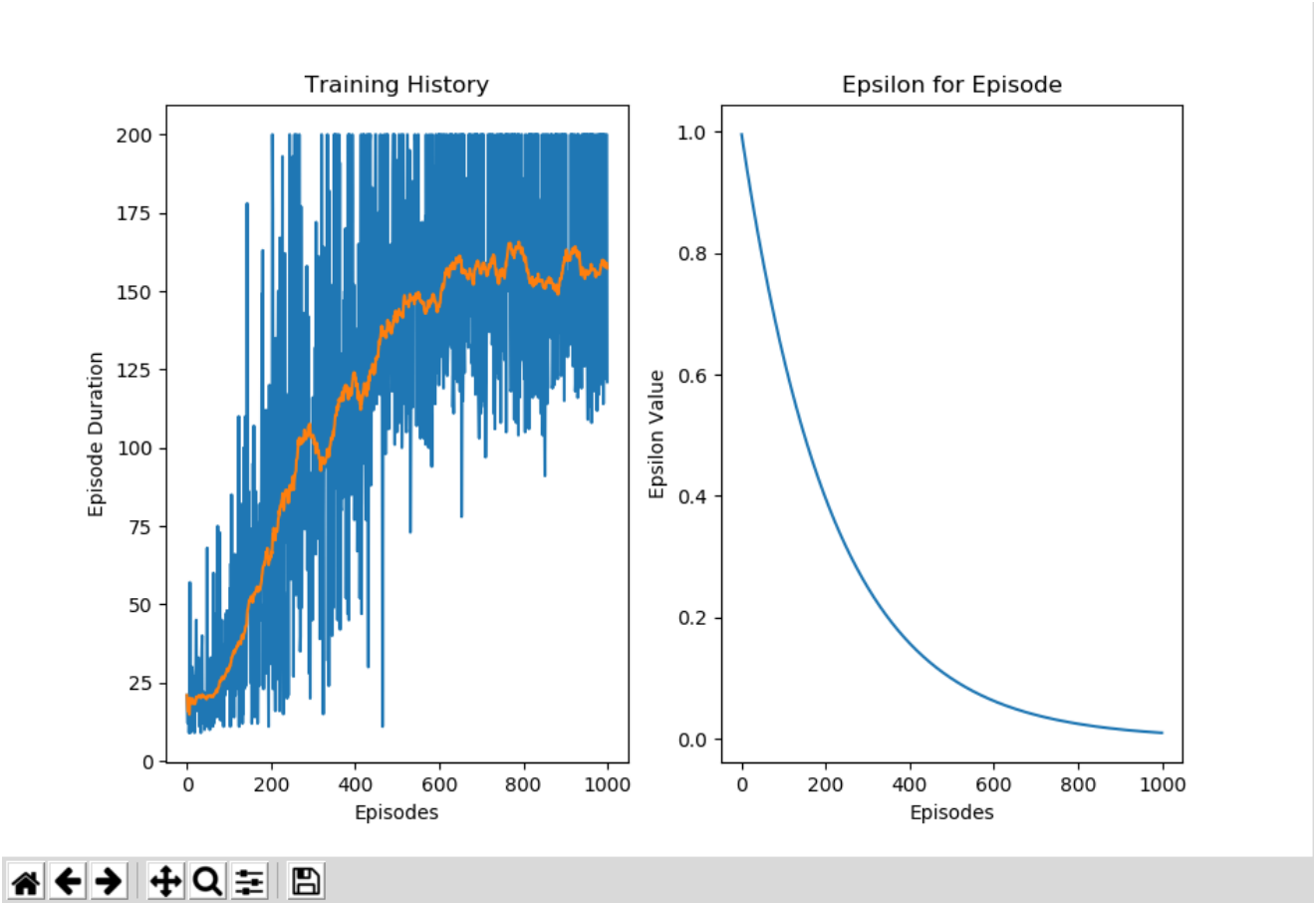

Figure: The orange plot at left shows how the average duration of an episode (in terms of the timesteps until the pole falls) increases as training proceeds. The value of epsilon as shown in the right plot controls the extent to which the agent chooses an action randomly vis-a-vis choosing an action based on Double Q learning. Initially, the value of epsilon is 1, implying that the actions will be chosen completely randomly. In the code, epsilon is represented by the variable `eps_rate`.

Outline

Why a Neural-Network Implementation of Q-Learning

- The Q-Learning based solution shown earlier explicitly constructs a Q table in which each row stands for a state and each column for an action. When in reality the state space is continuous — as it is for the cart-pole problem — the discretization will often introduce an approximation that degrades the quality of learning.
- There is an excellent way to evaluate the quality of learning for the Cart-Pole problem: the duration of each episode, which is the number of time-steps from the start of the episode ($t = 0$) until the time the state `fallen` is reached during each episode.
- If you execute the code shown in the previous section, the average duration you will get with that implementation of Q-Learning is likely to be around 100 time steps.
- To improve the performance beyond this, you'd need to go to a still finer discretization of the state space, or, better yet, to ditch the discretization altogether by using a neural-network based solution.

Q-Learning with a Neural Network

- What is shown next is a neural-network based implementation of Q-Learning that works with a continuous state space. This implementation is also by Rohan Sarkar.
- The neural-network is trained to take the continuous state value s at its input and to output the values of $Q(s, a_i)$ for each action a_i available to the agent in the input state. For its next action, the agent would obviously choose the action that has the largest $Q(s, a_i)$ value.
- Should the action chosen as described above result in getting further away from the desired goal, you would backpropagate a negative reward to reflect the undesirability of what was predicted by the neural network.

A Python Implementation of Q-Learning with a Neural Network

- The code shown next starts by initializing the hyperparameters in the lines that start with the one labeled A on Slide 49.
- By initializing `EPISODES=1000`, we are declaring that we only want to run the code for 1000 episodes. And the setting `EXPLORE_EPI_END = int(0.1*EPISODES)`, we are telling the system to use 100 of the episodes in the exploration phase. What does that mean?
- As it turns out, in its common usage, the phase "Exploration Phase" in neural Q-learning has a different meaning in relation to how I used it in the previous section. The "Exploration Phase" here means for the agent to figure out the next state and the reward just by interacting with the environment.

Q-Learning with a Neural Network (contd.)

- As to what we mean by the agent interacting with the environment, let's examine the following lines of code from the method `run_episode()` of `QNetAgent`. That method is defined at line G on Slide 50.

```
while True:
    environment.render()
    # Select action based on epsilon-greedy approach
    action = self.select_action(FloatTensor([state]))                    ## (a)
    c_action = action.data.cpu().numpy()[0,0]                            ## (b)
    # Get next state and reward from environment based on current action
    next_state, reward, done, _ = environment.step(c_action)             ## (c)
    # negative reward (punishment) if agent is in a terminal state
    if done:                                                             ## (d)
        reward = -10 # negative reward for failing                       ## (e)
    # push experience into replay memory
    self.memory.push(state,action, reward, next_state)                   ## (f)
    # if initial exploration is finished train the agent
    if EXPLORE_EPI_END <= e <= TEST_EPI_START:                           ## (g)
        self.learn()                                                     ## (h)
    state = next_state                                                   ## (i)
    steps += 1                                                           ## (j)
```

- The call in line (c) above is what's meant by interacting with the environment. The variable `environment` is set in the first line under `__main__` on Slide 51 to `gym.make('CartPole-v0')` where `gym` is the very popular OpenAI platform for research in RL.

Q-Learning with a Neural Network (contd.)

- For the Cart-Pole problem, the `gym` library provides a *very rudimentary* physics-based modeling of the Cart-Pole. The call in line (c) on the previous slide asks this model as to what the next state and the next reward should be if a given action is invoked in the current state. Suppose the currently randomly chosen state of the pole is `soft_lean_right` and the current randomly chosen action is `move left` (which is represented by the integer 1 in the code), the environment is likely to return `hard_lean_right` for the next state and a reward of 0 or -1.

- It is such interactions that are carried out in the exploration phase. In each episode, the agent makes state-to-state transitions with the help of the environment until it reaches the terminal state `fallen`.

- Going back to the beginning of this section, the second statement in the hyperparameter initializations that start in line A on Slide 49 means that we want to use the first 100 episodes for exploration prior to starting neural learning.

Q-Learning with a Neural Network (contd.)

- The main purpose of the "Exploration Phase" described above is to fill the `ReplayMemory`, defined in line labeled D, with the state transitions generated during that phase. What is actually stored in the ReplayMemory are 4-tuples like `[state, action, next_state, reward]` that were generated during the exploration. These are stored in the instance variable `self.memory` defined for the `ReplayMemory` class.

- After the storage in `ReplayMemory` is initialized in the manner indicated above, the agent starts up the `Training Phase` for training the neural network that is defined in the section of the code that begins in the line labeled C.

- The function whose job is to train the neural network is called `learn()` and it is defined starting with line H. That function creates batches of randomly selected training samples from what is stored in `ReplayMemory` and feeds those into the neural network.

Q-Learning with a Neural Network (contd.)

- To continue with the description of the `learn()` function, from each 4-tuple `[state, action, next_state, reward]` chosen from the replay memory, the first two, meaning `state` and `action`, are used as input to the network and the target at the output consists of `next_state` and `reward`.

- The variable `e` in the code stands for the episode number. In the following statements taken from the beginning portion of the `run_episode()` function defined starting at line G, note how the episode number `e` is used to decide what phase the agent is in:

```
def run_episode(self, e, environment):
    state = environment.reset() # reset the environment at the beginning
    done = False
    steps = 0
    # Set the epsilon value for the episode
    if e < EXPLORE_EPI_END:
        self.epsilon = EPS_START
        self.mode = "Exploring"
    elif EXPLORE_EPI_END <= e <= TEST_EPI_START:
        self.epsilon = self.epsilon*EPS_DECAY
        self.mode = "Training"
    elif e > TEST_EPI_START:
        self.epsilon = 0.0
        self.mode = "Testing"
```

Q-Learning with a Neural Network (contd.)

- For the parameter initializations shown in the code, the value of `EXPLORE_EPI_END` in the code fragment shown on the previous slide will be set to 100. Therefore, for episode number less than 100, the "if" block shown above would set `self.epsilon` to `EPS_START`, which is equal to 1. And `self.mode` would be set to "Exploring" at the same time.
- Setting `self.epsilon` to 1 causes the `select_action()` method defined at line F to return a purely randomly chosen action in whatever mode the agent happens to be in at the moment. This is in keeping with the expected behavior of the agent in the Exploration phase according to the epsilon-greedy policy for agents.

Q-Learning with a Neural Network (contd.)

```
##  Solving the Cart-Pole problem with PyTorch
##  Code adapted from https://gist.github.com/Pocuston/13f1a7786648e1e2ff95bfad02a51521
##  Modified by Rohan Sarkar (sarkarr@purdue.edu)

##  QNet_pytorch_v5.py

import gym                                                                          ## (A)
import random
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import matplotlib.pyplot as plt
import numpy as np
##  Set the hyperparameters for training:
EPISODES = 1000           # total number of episodes
EXPLORE_EPI_END = int(0.1*EPISODES)     # initial exploration when agent will explore and no training
TEST_EPI_START = int(0.7*EPISODES )    # agent will be tested from this episode
EPS_START = 1.0           # e-greedy threshold start value
EPS_END = 0.05            # e-greedy threshold end value
EPS_DECAY = 1+np.log(EPS_END)/(0.6*EPISODES)         # e-greedy threshold decay
GAMMA = 0.8               # Q-learning discount factor
LR = 0.001                # NN optimizer learning rate
MINIBATCH_SIZE = 64       # Q-learning batch size
ITERATIONS = 40           # Number of iterations for training
REP_MEM_SIZE = 10000      # Replay Memory size

use_cuda = torch.cuda.is_available()                                                ## (B)
FloatTensor = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
LongTensor = torch.cuda.LongTensor if use_cuda else torch.LongTensor
ByteTensor = torch.cuda.ByteTensor if use_cuda else torch.ByteTensor
Tensor = FloatTensor

class QNet(nn.Module):                                                              ## (C)
    """
    Input to the network is a 4-dimensional state vector and the output a
    2-dimensional vector of two possible actions: move-left or move-right
    """
    def __init__(self, state_space_dim, action_space_dim):
        nn.Module.__init__(self)
        self.l1 = nn.Linear(state_space_dim, 24)
        self.l2 = nn.Linear(24, 24)
        self.l3 = nn.Linear(24, action_space_dim)
    def forward(self, x):
        x = F.relu(self.l1(x))
        x = F.relu(self.l2(x))
        x = self.l3(x)
        return x

class ReplayMemory:                                                                 ## (D)
    """
    This class is used to store a large number, possibly say 10000, of the
    4-tuples '[State, Action, Next State, Reward]' at the output, meaning even
    before the nueral-network based learning kicks in.  Subsequently, batches
    are constructed from this storage for training. The dynamically updated
    as each new 4-tuple becomes available.
    """
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
    def push(self, s, a , r, ns):
        self.memory.append((FloatTensor([s]),
                    a,  # action is already a tensor
                    FloatTensor([ns]),
                    FloatTensor([r])))
        if len(self.memory) > self.capacity:
            del self.memory[0]
    def sample(self, MINIBATCH_SIZE):
        return random.sample(self.memory, MINIBATCH_SIZE)
    def __len__(self):
        return len(self.memory)
```

(Continued on the next slide .....)

Q-Learning with a Neural Network (contd.)

(..... continued from the previous slide)

```
class QNetAgent:                                                                    ## (E)
    def __init__(self, stateDim, actionDim):
        self.sDim = stateDim
        self.aDim = actionDim
        self.model = QNet(self.sDim, self.aDim)    # Instantiate the NN model, loss and optimizer for training the agent
        if use_cuda:
            self.model.cuda()
        self.optimizer = optim.Adam(self.model.parameters(), LR)
        self.lossCriterion = torch.nn.MSELoss()
        self.memory = ReplayMemory(REP_MEM_SIZE)  # Instantiate the Replay Memory for storing agent's experiences
        # Initialize internal variables
        self.steps_done = 0
        self.episode_durations = []
        self.avg_episode_duration = []
        self.epsilon = EPS_START
        self.epsilon_history = []
        self.mode = ""

    def select_action(self, state):                                                 ## (F)
        """ Select action based on epsilon-greedy approach """
        p = random.random() # generate a random number between 0 and 1
        self.steps_done += 1
        if p > self.epsilon:
            # if the agent is in 'exploitation mode' select optimal action
            # based on the highest Q value returned by the trained NN
            with torch.no_grad():
                return self.model(FloatTensor(state)).data.max(1)[1].view(1, 1)
        else:
            # if the agent is in the 'exploration mode' select a random action
            return LongTensor([[random.randrange(2)]])

    def run_episode(self, e, environment):                                          ## (G)
        state = environment.reset() # reset the environment at the beginning
        done = False
        steps = 0
        # Set the epsilon value for the episode
        if e < EXPLORE_EPI_END:
            self.epsilon = EPS_START
            self.mode = "Exploring"
        elif EXPLORE_EPI_END <= e <= TEST_EPI_START:
            self.epsilon = self.epsilon*EPS_DECAY
            self.mode = "Training"
        elif e > TEST_EPI_START:
            self.epsilon = 0.0
            self.mode = "Testing"
        self.epsilon_history.append(self.epsilon)
        while True:                                 # Iterate until episode ends (i.e. a terminal state is reached)
            environment.render()
            action = self.select_action(FloatTensor([state]))    # Select action based on epsilon-greedy approach
            c_action = action.data.cpu().numpy()[0,0]
            # Get next state and reward from environment based on current action
            next_state, reward, done, _ = environment.step(c_action)
            if done:                                # negative reward (punishment) if agent is in a terminal state
                reward = -10 # negative reward for failing
            # push experience into replay memory
            self.memory.push(state,action, reward, next_state)
            # if initial exploration is finished train the agent
            if EXPLORE_EPI_END <= e <= TEST_EPI_START:
                self.learn()
            state = next_state
            steps += 1
            if done:                                        # Print information after every episode
                print("{2} Mode: {4} | Episode {0} Duration {1} steps | epsilon {3}"
                      .format(e, steps, '\033[92m' if steps >= 195 else '\033[99m', self.epsilon, self.mode))
                self.episode_durations.append(steps)
                self. plot_durations(e)
                break
```


(Continued on the next slide .....)

Q-Learning with a Neural Network (contd.)

(..... continued from the previous slide)

```
    def learn(self):                                                                    ## (H)
        """
        Train the neural newtwork using the randomly selected 4-tuples
        '[State, Action, Next State, Reward]' from the ReplayStore storage.
        """
        if len(self.memory) < MINIBATCH_SIZE:
            return
        for i in range(ITERATIONS):
            # minibatch is generated by random sampling from experience replay memory
            experiences = self.memory.sample(MINIBATCH_SIZE)
            batch_state, batch_action, batch_next_state, batch_reward = zip(*experiences)
            # extract experience information for the entire minibatch
            batch_state = torch.cat(batch_state)
            batch_action = torch.cat(batch_action)
            batch_reward = torch.cat(batch_reward)
            batch_next_state = torch.cat(batch_next_state)
            # current Q values are estimated by NN for all actions
            current_q_values = self.model(batch_state).gather(1, batch_action)
            # expected Q values are estimated from actions which gives maximum Q value
            max_next_q_values = self.model(batch_next_state).detach().max(1)[0]
            expected_q_values = batch_reward + (GAMMA * max_next_q_values)
            # loss is measured from error between current and newly expected Q values
            loss = self.lossCriterion(current_q_values, expected_q_values.unsqueeze(1))
            # backpropagation of loss for NN training
            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()

    def plot_durations(self,epi):        #  Update the plot at the end of each episode   ## (I)
        fig = plt.figure(1)
        fig.canvas.set_window_title("DQN Training Statistics")
        plt.clf()
        durations_t = torch.FloatTensor(self.episode_durations)
        plt.subplot(1,2,1)
        if epi<EXPLORE_EPI_END:
            plt.title('Agent Exploring Environment')
        elif EXPLORE_EPI_END <= e <= TEST_EPI_START:
            plt.title('Training Agent')
        else:
            plt.title('Testing Agent')
        plt.xlabel('Episode')
        plt.ylabel('Duration')
        plt.plot(self.episode_durations)
        # Plot cumulative mean
        if len(durations_t) >= EXPLORE_EPI_END:
            means = durations_t.unfold(0, EXPLORE_EPI_END, 1).mean(1).view(-1)
            means = torch.cat((torch.zeros(EXPLORE_EPI_END-1), means))
            plt.plot(means.numpy())
        plt.subplot(1,2,2)
        plt.title("Epsilon per Episode")
        plt.xlabel('Episode')
        plt.ylabel('Epsilon')
        plt.plot(self.epsilon_history)
        plt.show(block=False)
        plt.draw()
        plt.pause(0.0001)

if __name__ == "__main__":                                                              ## (J)
    environment = gym.make('CartPole-v0')               # creating the OpenAI Gym Cartpole environment
    state_size = environment.observation_space.shape[0]
    action_size = environment.action_space.n
    # Instantiate the RL Agent
    agent = QNetAgent(state_size, action_size)
    for e in range(EPISODES):                           # Train the agent
        agent.run_episode(e, environment)
    print('Complete')
    test_epi_duration = agent.episode_durations[TEST_EPI_START-EPISODES+1:]
    print("Average Test Episode Duration",np.mean(test_epi_duration))
    environment.close()
    plt.ioff()
    plt.show()
```

Visualizing the Training Phase

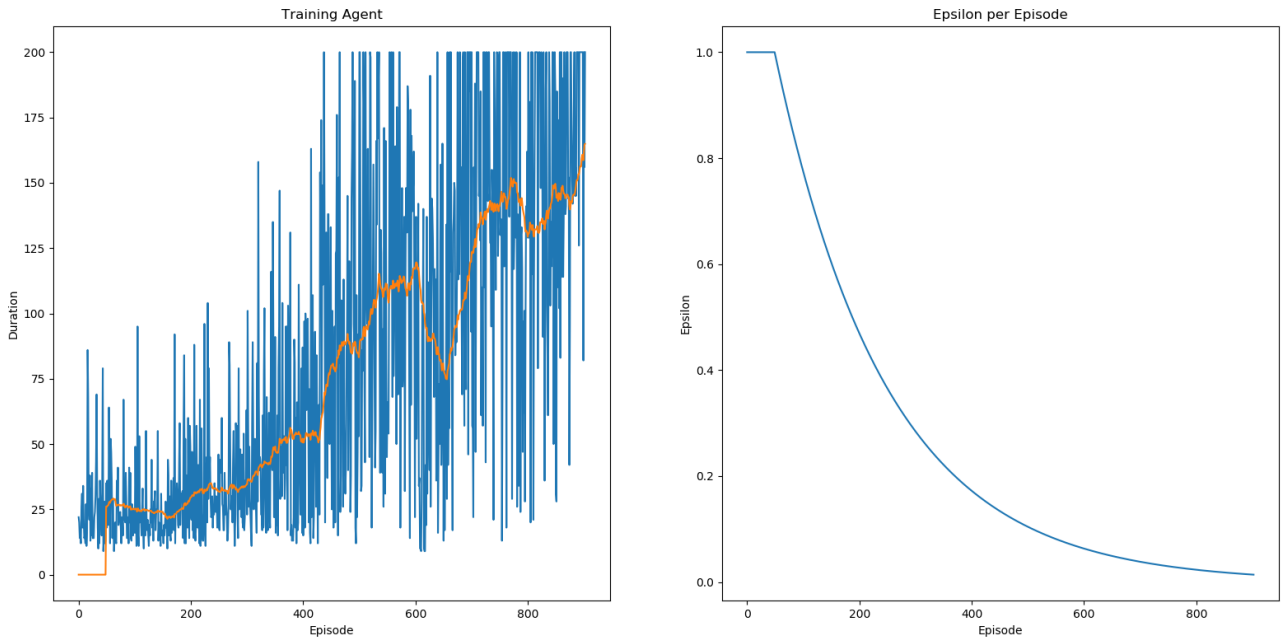

Figure: See how the average duration of an episode, shown in orange in the plot at left, increases as training proceeds. The value of epsilon as shown in the right plot controls the extent to which the agent chooses an action randomly vis-a-vis choosing an action based on neural learning. Initially, the value of epsilon is 1, implying that the actions will be chosen completely randomly. In the code, epsilon is represented by the variable 'self.epsilon'. The small flat part of the curve at the beginning of the plot at right is for the episodes when the replay memory is initialized by interacting with the environment prior to the start of neural learning.