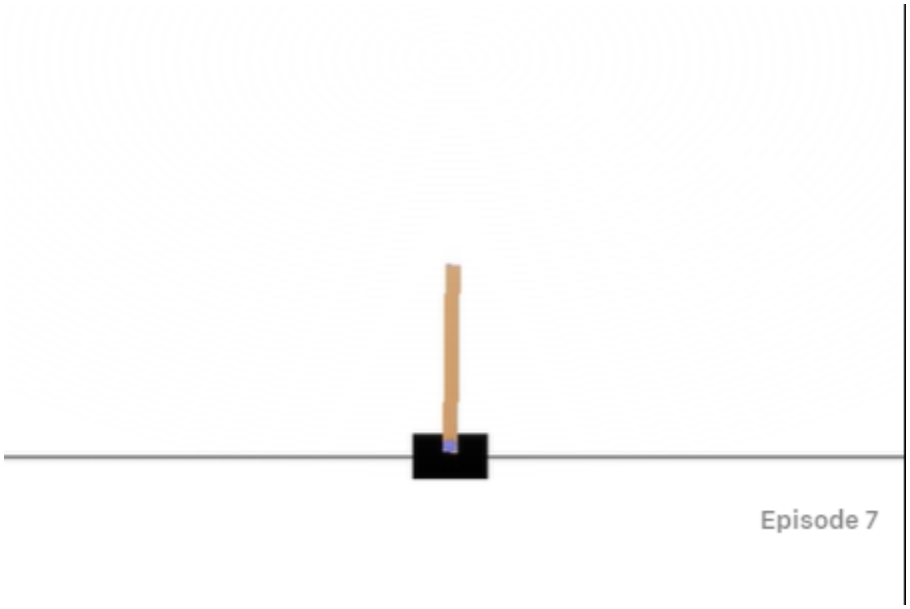
Report

Environment

CartPole-v1



Description:

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum starts upright, and the goal is to prevent it from falling over by increasing and reducing the cart's velocity.

Observation:

Type: Box(4)

Num	Observation	Min	Max
0	Cart Position	-2.4	2.4
1	Cart Velocity	-Inf	Inf
2	Pole Angle	-0.209 rad (-12 deg)	0.209 rad (12 deg)
3	Pole Angular Velocity	-Inf	Inf

Actions:

Type: Discrete(2)

Num	Action
0	Push cart to the left
1	Push cart to the right

Note: The amount the velocity that is reduced or increased is not fixed; it depends on the angle the pole is pointing. This is because the center of gravity of the pole increases the amount of energy needed to move the cart underneath it

Reward:

Reward is 1 for every step taken, including the termination step

Starting State:

All observations are assigned a uniform random value in [-0.05..0.05]

Episode Termination:

Pole Angle is more than 12 degrees.

Cart Position is more than 2.4 (center of the cart reaches the edge of the display).

Episode length is greater than 200.

Solved Requirements:

Considered solved when the average return is greater than or equal to 195.0 over 100 consecutive trials.

Learning Algorithms

DQN:

Initialize network Q
Initialize target network $\hat{Q}$
Initialize experience replay memory D
Initialize the *Agent* to interact with the Environment
while *not converged* **do**

    /* Sample phase
    $\epsilon \leftarrow$ setting new epsilon with ϵ-decay
    Choose an action a from state s using policy ϵ-greedy(Q)
    Agent takes action a, observe reward r, and next state s'
    Store transition $(s, a, r, s', done)$ in the experience replay memory D

    if *enough experiences in* D **then**
        /* Learn phase
        Sample a random *minibatch* of N transitions from D
        for *every transition* $(s_i, a_i, r_i, s'_i, done_i)$ ***in*** *minibatch* **do**
            if $done_i$ **then**
                $y_i = r_i$
            else
                $y_i = r_i + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s'_i, a')$
            end
        end
        Calculate the loss $\mathcal{L} = 1/\mathrm{N} \sum_{i=0}^{N-1} (Q(s_i, a_i) - y_i)^2$
        Update Q using the SGD algorithm by minimizing the loss $\mathcal{L}$
        Every C steps, copy weights from Q to $\hat{Q}$
    end
end

Double DQN:

Algorithm 1 : Double Q-learning (Hasselt et al., 2015)

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma\ Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

Hyperparameters

```
GAMMA=0.99 #Discount rate
LEARNING_RATE=5e-4 #learning rate
BATCH_SIZE=32 #Number of samples from repaly buffer
BUFFER_SIZE=50000 #Maximum size before overwritting
MIN_REPLAY_SIZE=1000 #Number of transiotions in replay buffer before computing gradients
EPSILON_START=1.0 #Starting value of eps
EPSILON_END=0.002 #Ending value of eps
EPSILON_DECAY=10000 #Decay value of eps
TARGET_UPDATE_FREQ=1000 #Number of steps of target update
```

Model Architecture

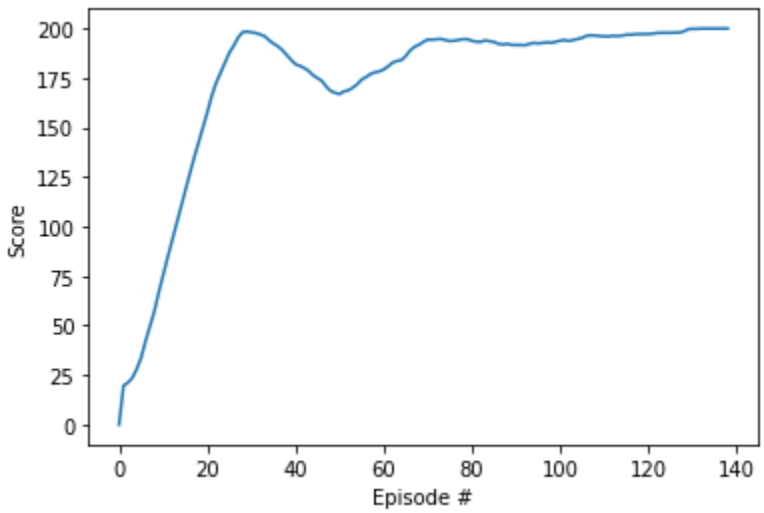
Input : State shape (4)
Hidden layers : 64
Output : Number of actions (2)

```python
class Network(nn.Module):
    def __init__(self,env):
        super().__init__()

        in_features=int(np.prod(env.observation_space.shape))
        self.net=nn.Sequential(
           nn.Linear(in_features,64),
           nn.Tanh(),
           nn.Linear(64,env.action_space.n))
    def forward(self,x):
        return self.net(x)
    def act(self,obs):
        obs_t=torch.as_tensor(obs,dtype=torch.float32)
        q_values=self(obs_t.unsqueeze(0))
        max_q_index=torch.argmax(q_values,dim=1)[0]
        action=max_q_index.detach().item()

        return action
```

Results



Test on 5 episodes :

```
episode:1 score:200.0
episode:2 score:200.0
episode:3 score:200.0
episode:4 score:200.0
episode:5 score:200.0
```