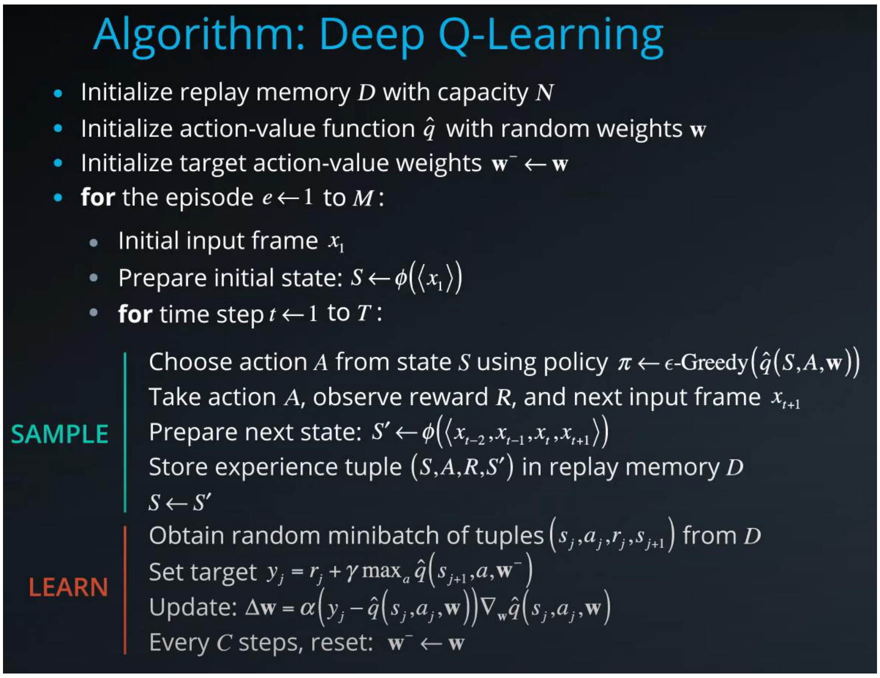
1. Learning algorithm

1.1. Overview of the technique

The agent is implemented using the Deep Q-Learning method (along with Replay Buffer and Fixed Q-Target technique). At each time step, the agent will take an action, store the transition in its replay buffer (Sample) and then sample a minibatch from the buffer to update the primary Q-network with the target Q-network. The target network is updated either after a number of steps by copying the primary network's weights or slowly update by τ (a hyperparameter):

target = τ * primary + (1- τ) * target

The algorithm in detail:



Besides, in order to help the agent learn better, I have implemented the algorithm with 2 improvements: Duelling DQN and Double DQN to avoid overestimating problem.

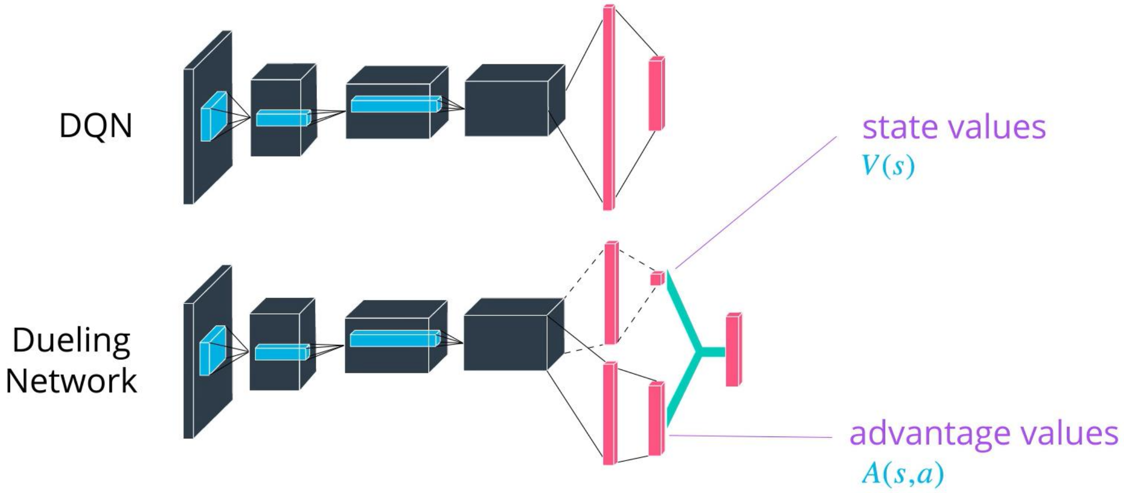
*) Double DQN:

When learning:

- Vanilla DQN estimate the true value for computing loss by the maximum action-value at the next state, which return from the target network. This can lead to overestimation of action-value.
  Formula for this true value estimation can be written as:
  $$R + \gamma\hat{q}(S', \underset{a}{argmax}\,\hat{q}(S', a, w), w)$$
  (Select best action from the target network and evaluate that action on the target network)
- Double DQN solve this by select the best action from the primary network and evaluate that action on the target network.
- Formula for this estimation
  $$R + \gamma\hat{q}(S', \underset{a}{argmax}\,\hat{q}(S', a, w), w^-)$$

*) Dueling DQN

Its network architecture is different from vanilla DQN. It uses 2 streams, 1 to compute the value of state and 1 to compute the advantage of each action in that state.

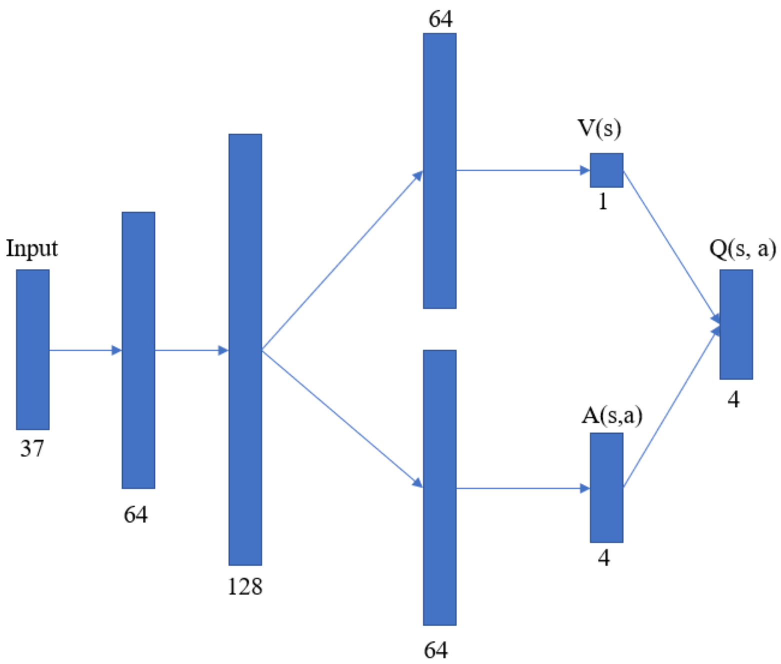


The output of the dueling network is computed as follow (forcing the advantage function estimator to have 0 advantage at the chosen action):

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{Na}\sum_{a'} A(s,a')\right) \qquad (Na: \#\ actions)$$

1.2. Network architecture

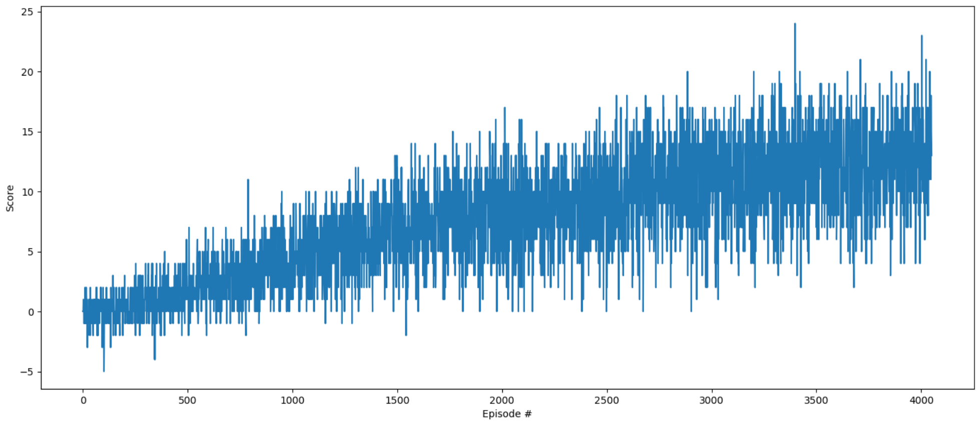
The network architecture for both primary and target Q-network is described below (the number is # neurons at each hidden layers)



1.3. Hyperparameters select

```
class Config:
    # seed for randomness
    SEED = 3737

    # name of the unity environment
    UNITY_ENV = "Banana_Windows_x86_64\Banana.exe"

    # training hyperparams
    BATCH_SIZE = 64                         # batch size for learning
    BUFFER_SIZE = 100000                    # buffer max capacity
    TAU = 1e-3                              # hyperparam update the target network
    NETWORK_ARCHITECTURE = [64, 128, 64]    # hidden layers in the network
    LEARN_EVERY = 4                         # agent will sample and learn after LEARN_EVERY steps
    EPSILON_START = 1.0                     # epsilon start at first episode
    EPSILON_END = 0.05                      # epsilon min
    EPSILON_DECAY = 0.9995                  # epsilon decay after each episode
    EPSILON_FOR_INFER = 0.05                # epsilon for inferencing
    GAMMA = 0.99                            # discount factor
    LR = 0.001                              # learning rate
    DEVICE = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    NUM_EPISODE = 5000                      # num of trAaining episodes
    NUM_EPiSODE_INFER = 100                 # num of inferencing episodes
    MAX_TIMESTEP = 300                      # max timestep for each episode

    CHECKPOINT = "checkpoint.pth"           # checkpoint path of trained weight
```

2. Plot of rewards



3. Ideas for future works

The result I have shown is not too good, which solve the problem in about 4k episodes ☹ I think some modifications will make the result better:

- Tuning hyperparameters such as epsilon (start, end, decay) and the architecture of the Q-network probably enhance the learning ability of the agent
- Moreover, many other techniques can be used in order to help the agent learn better such as Prioritized Replay Buffer, Distributing DQN, … or even combine all – Rainbow.