RL- models evaluation on Snake

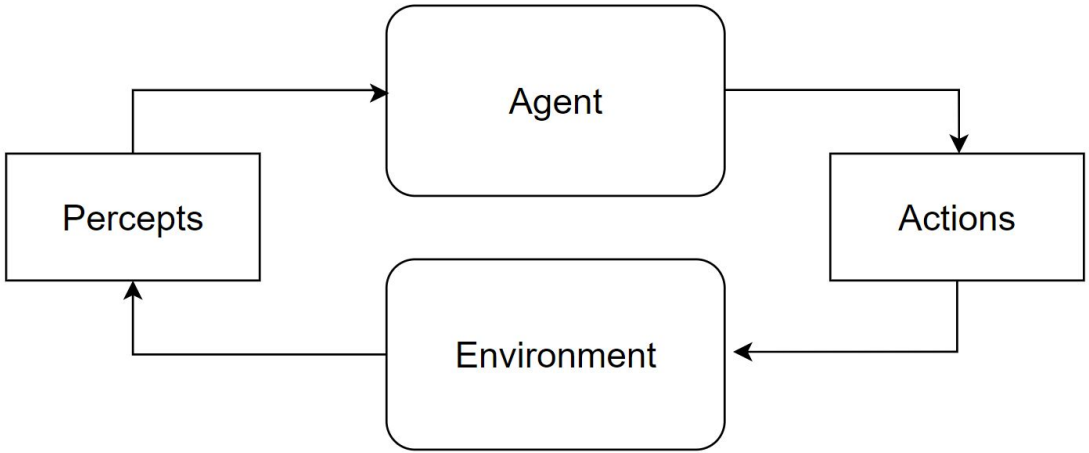
Assesment of DQN, DDQN algorithms

Problem statement

- Environment:
  - 2 dimensional space, bounded. A snake moves around this space looking for food to eat. When the snake eats it grows in length. Do not hit the barrier or the snake itself!
- Task:
  - Train an agent with DQ Learning to achieve the best possible score
- Assess improvements:
  - Use DQN extension
  - Inject some experiences before the training process

Tools

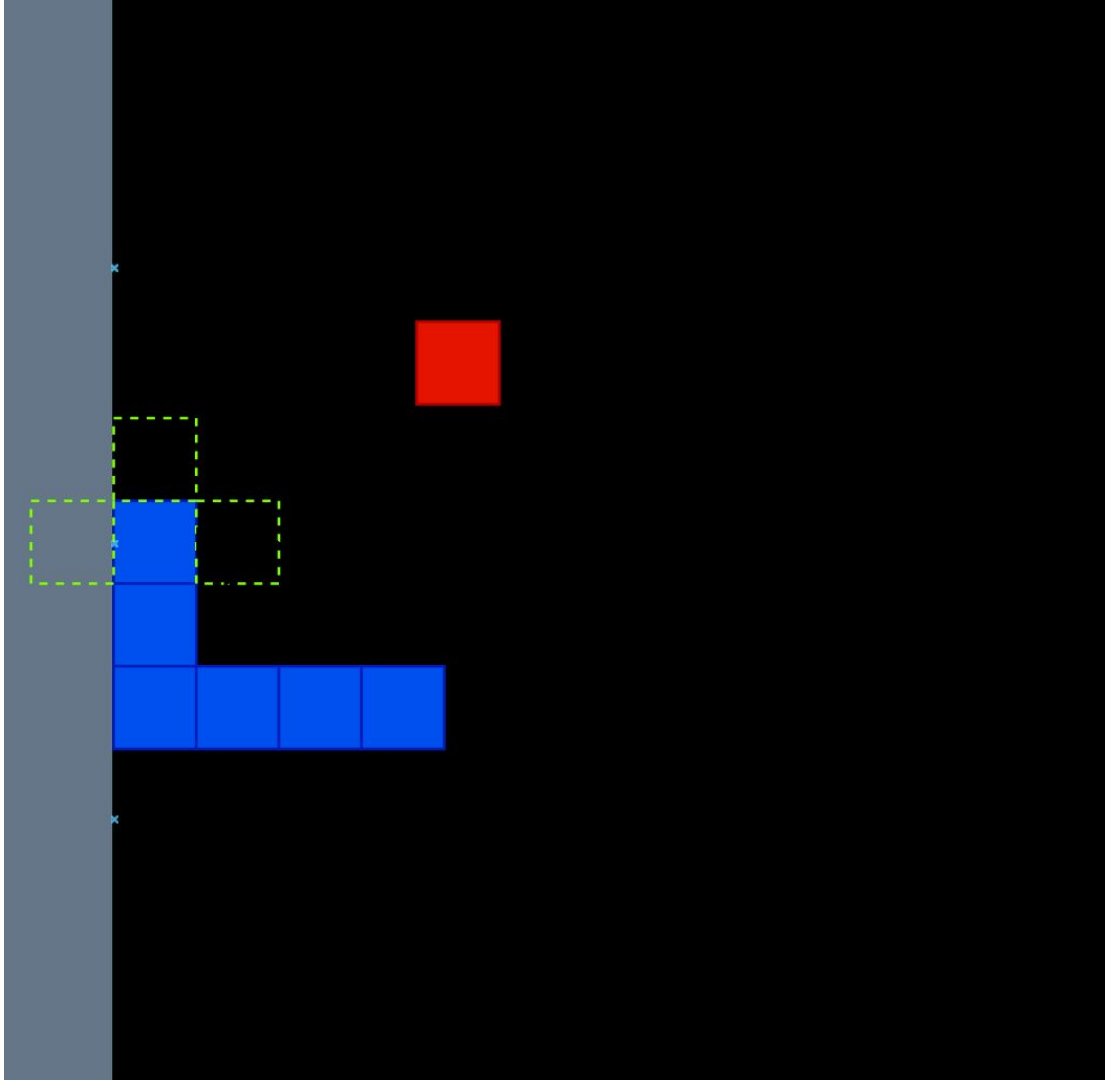
Agent interactions



Environment

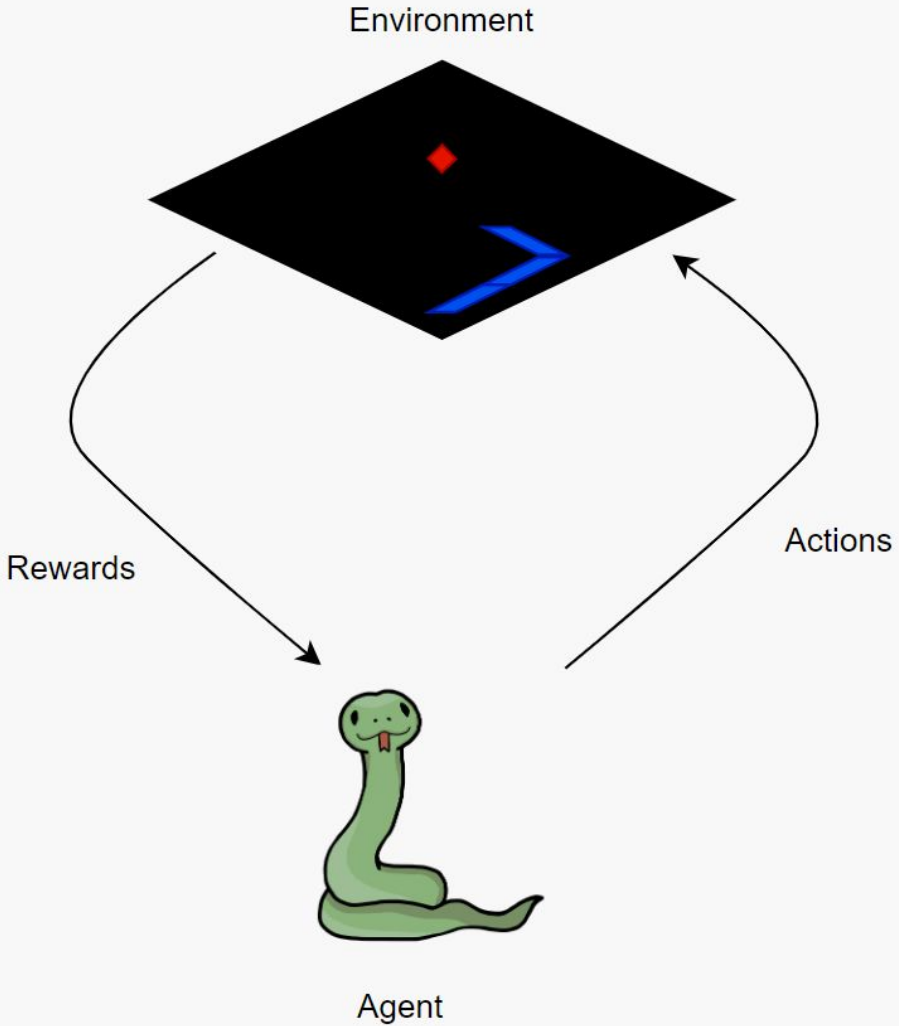
- Game state vector:
  - Danger
    - Straight, left, right
  - Current moving directions
    - up, down, left, right
  - Current food direction
    - up, down, left, right

example : [0 1 0 1 0 0 0 1 0 0 1]



Percepts

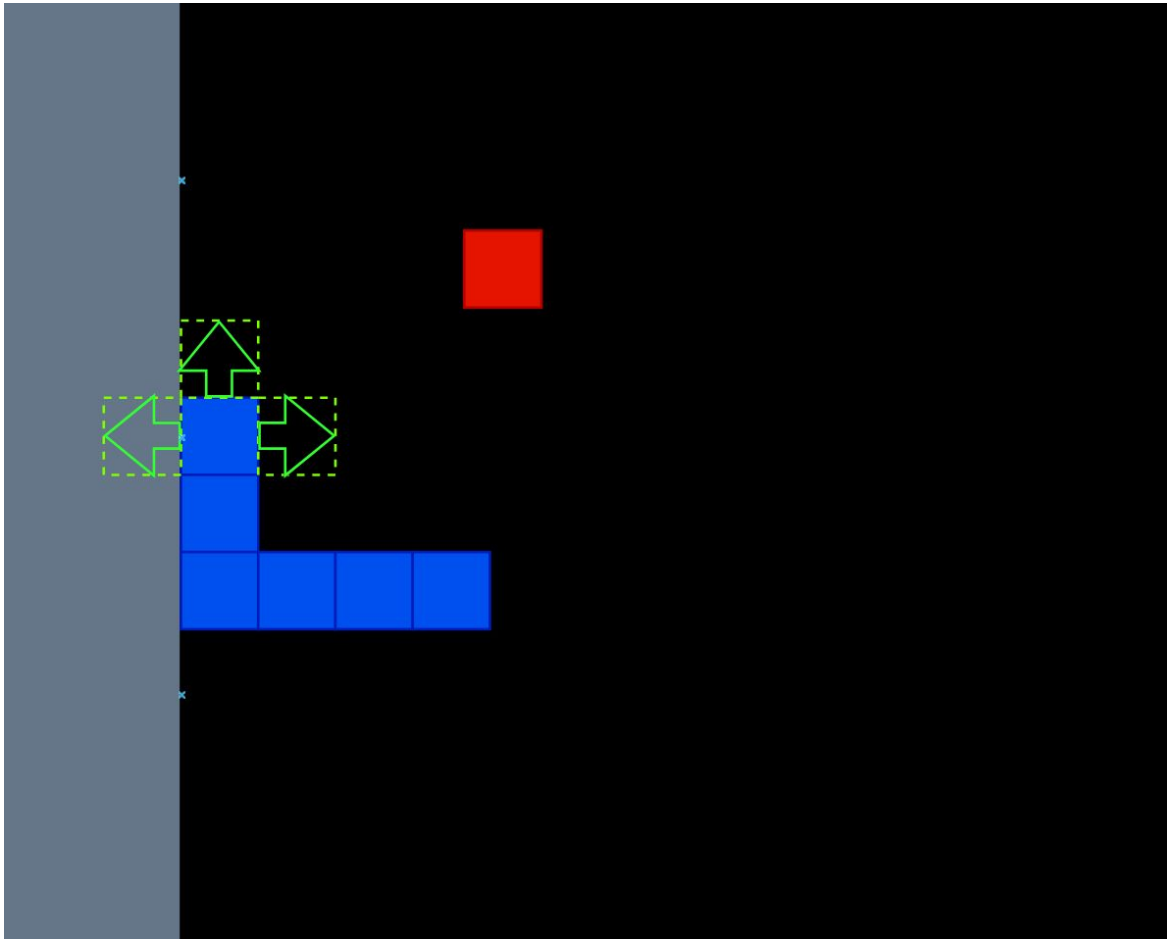
- Reward
  - If collision -> -10
  - If chop food -> +10
  - else 0
- Done:
  - Wheter the game is finished or not



Actions

- 3 possible actions
  - Forward, turn-left, turn-right

example : [0 0 1]



Training process - Optimal Bellman equation

$$Q(s,a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t \mid s, a\right]$$

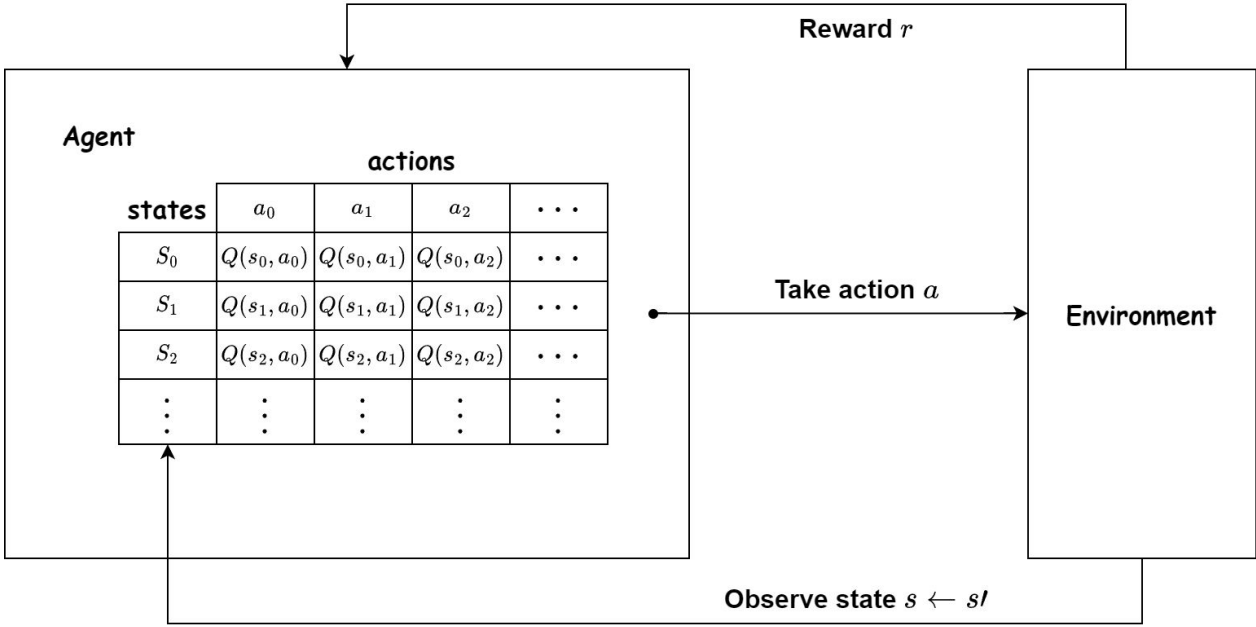
$$Q(s,a) = \sum_{s',r} p(s', r|s, a)\left[r + \gamma \max_{a'} Q(s', a')\right]$$

Q-learning - Base approach

$$\text{New } Q(s,a) = Q(s,a) + \alpha\,[R(s,a) + \gamma\, \max Q'(s',a') - Q(s,a)]$$

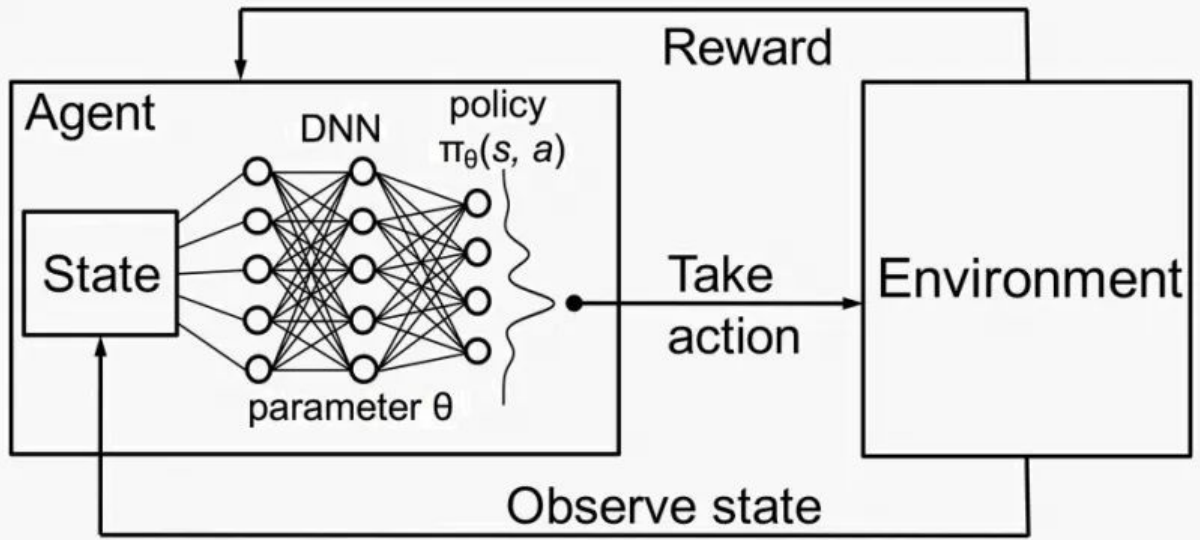
TD-error

- New Q Value for that state and the action
- Learning Rate
- Reward for taking that action at that state
- Current Q Values
- Maximum expected future reward given the new state (s') and all possible actions at that new state.
- Discount Rate



Deep learning approach- Deep Q-Learning

- Deep Q-Network
  - In DQN, the Q-function is represented implicitly by a neural network



Model - DQN

Algorithm 1 Deep Q-Network (DQN)

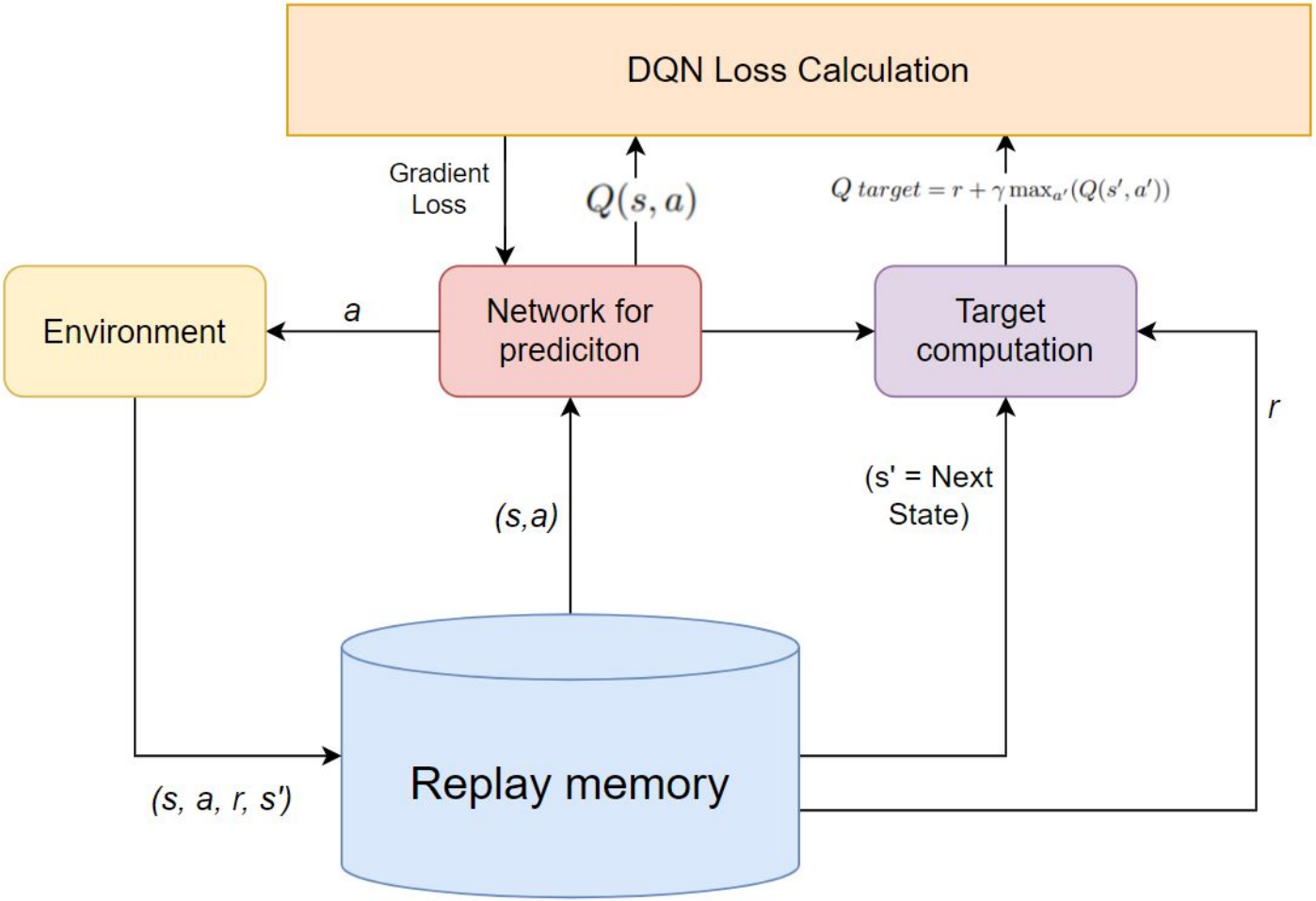
Initialize replay memory D with capacity N
Initialize Q-network with random weights

for episode in range(total_episodes) **do**
  Initialize state s
  for step in range(max_steps_per_episode) **do**
    Choose action a using epsilon-greedy policy based on Q-network
    Take action a, observe reward r and next state s'
    Store (s, a, r, s') in replay memory D
    Sample a random minibatch of experiences (s, a, r, s') from D

    Calculate target Q-values:
    if s' is terminal state **then**
      $target = r$
    else
      $target = r + \gamma \max_{a'}(Q(s', a'))$
    end if

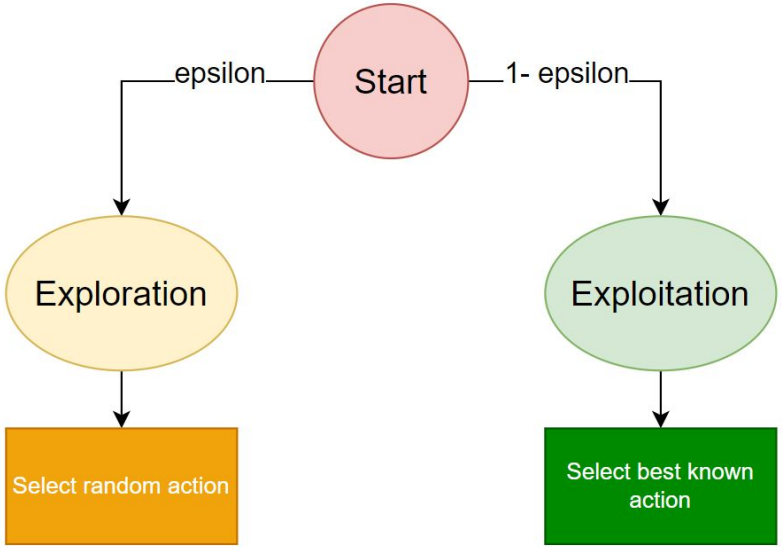
    Update the Q-network weights using gradient descent:
    $L(\theta) = \text{MSE}(Q(s, a), target)$
    $\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta)$
    Backpropagate the loss and update the Q-network weights
    Update state $s = s'$
  end for
  Reduce epsilon (exploration rate) over time (e.g., $\epsilon* = \epsilon$_decay)
end for

Model - DQN


DQN Loss Calculation
Gradient Loss
$Q(s,a)$
$Q\ target = r + \gamma \max_{a'}(Q(s',a'))$
Environment
a
Network for prediciton
Target computation
r
(s,a)
(s' = Next State)
Replay memory
(s, a, r, s')

Models - Policy and action selection strategies

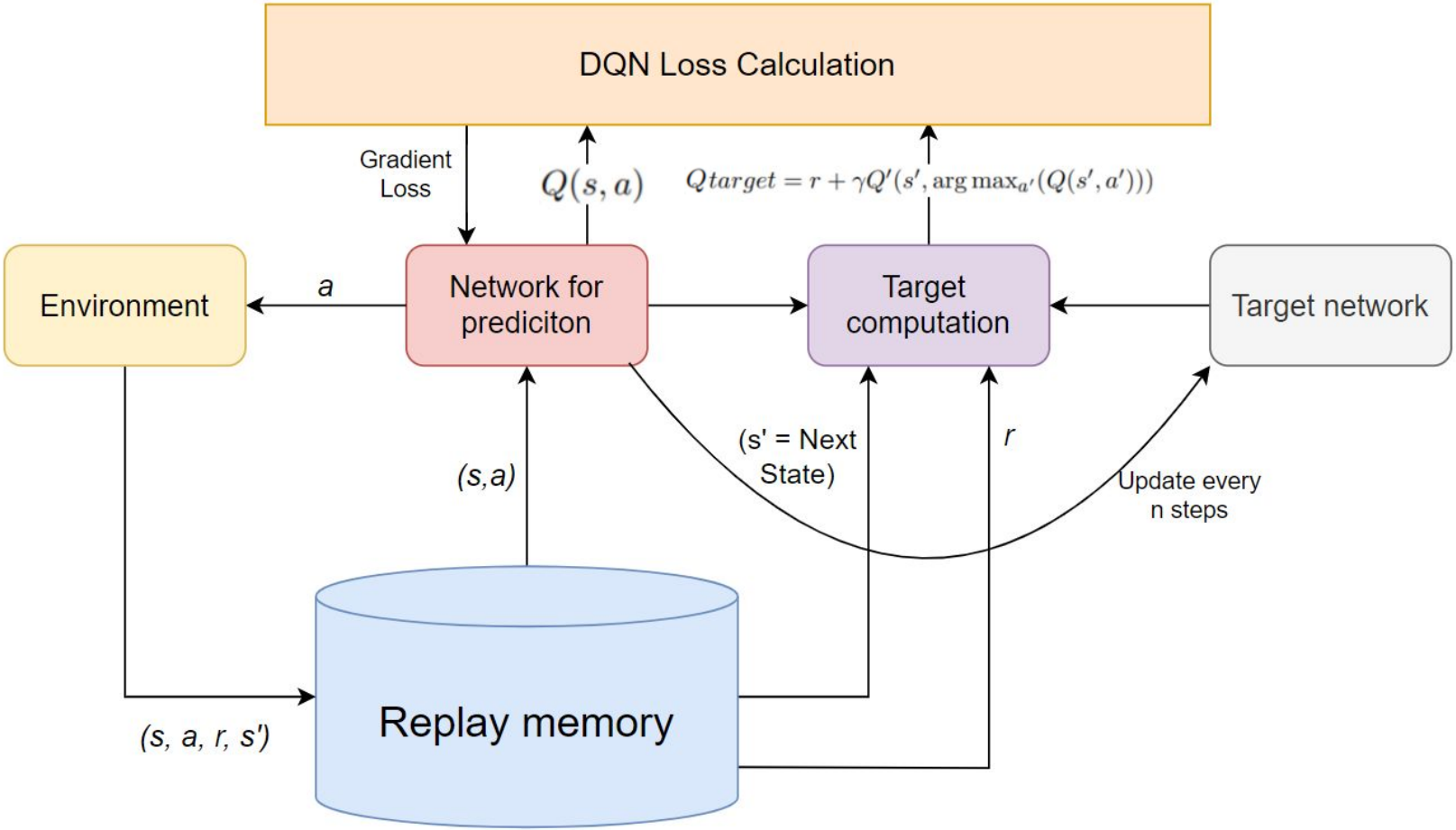
- Like QL, this is an off-policy algorithm
  - The policy we try to learn is not the same as the policy used for action selection during training.
    Agent learns from its past experiences, regardless of the policy used to generate those experiences.
- Epsilon-greedy
  - Trade-off between exploration and exploitation
  - epsilon is reduced while the training goes on



Models

- DQN extension: Double Deep Q-Network
    - This method handles the problem of the overestimation of Q-values.
      During the training, especially at the beginning, the Q network estimates which is the best action to take, but possibly the actual Q value is overestimated
    - When we compute the Q target, we use two networks to decouple the action selection (online network) from the target Q value generation (target network).

Models – DDQN


DQN Loss Calculation
Gradient Loss
$Q(s,a)$
$Qtarget = r + \gamma Q'(s', \arg\max_{a'}(Q(s',a')))$
Environment
a
Network for prediciton
Target computation
Target network
(s' = Next State)
r
(s,a)
Update every n steps
Replay memory
(s, a, r, s')

Models – DDQN

Algorithm 2 Double Deep Q-Network (Double DQN)

Initialize replay memory D with capacity N
Initialize Q-network Q and target Q-network Q' with random weights

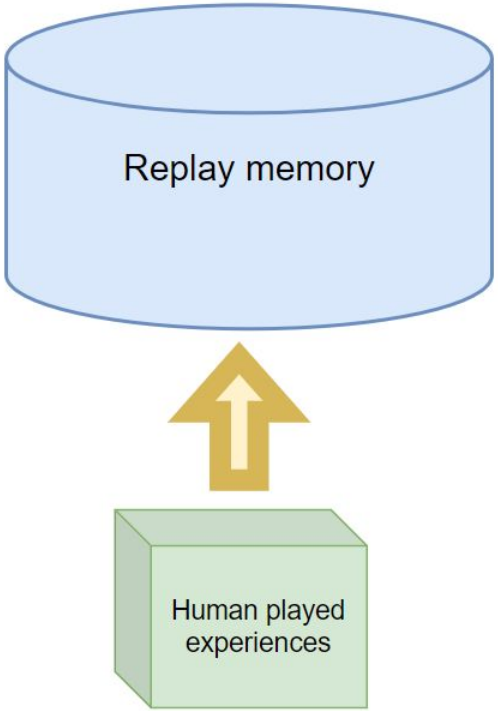
for episode in range(total_episodes) **do**
  Initialize state s
  for step in range(max_steps_per_episode) **do**
    Choose action a using epsilon-greedy policy based on Q-network Q
    Take action a, observe reward r and next state s'
    Store (s, a, r, s') in replay memory D
    Sample a random minibatch of experiences (s, a, r, s') from D

    Calculate target Q-values using target network Q':
    if s' is terminal state **then**
      $target = r$
    else
      $target = r + \gamma Q'(s', \arg\max_{a'}(Q(s', a')))$
    end if

    Update the Q-network weights using gradient descent:
    $L(\theta) = \text{MSE}(Q(s, a), target)$
    $\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta)$
    Backpropagate the loss and update the Q-network weights
    Update state $s = s'$
  end for
  Reduce epsilon (exploration rate) over time (e.g., $\epsilon* = \epsilon$_decay)
  if episode%target_update_frequency == 0 **then**
    Update the target Q-network Q' with the current Q-network Q weights
  end if
  if episode%save_model_frequency == 0 **then**
    Save the Q-network Q model
  end if
end for

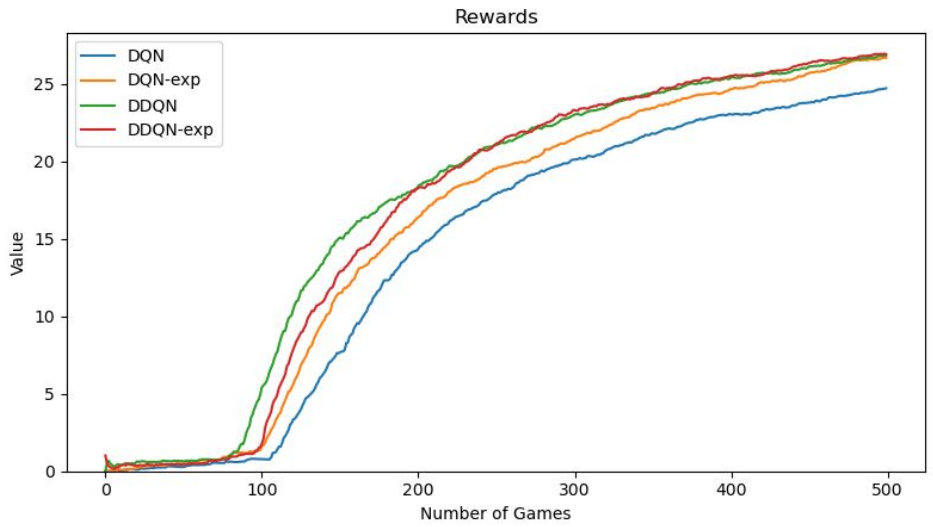
Improvements - Injecting experience

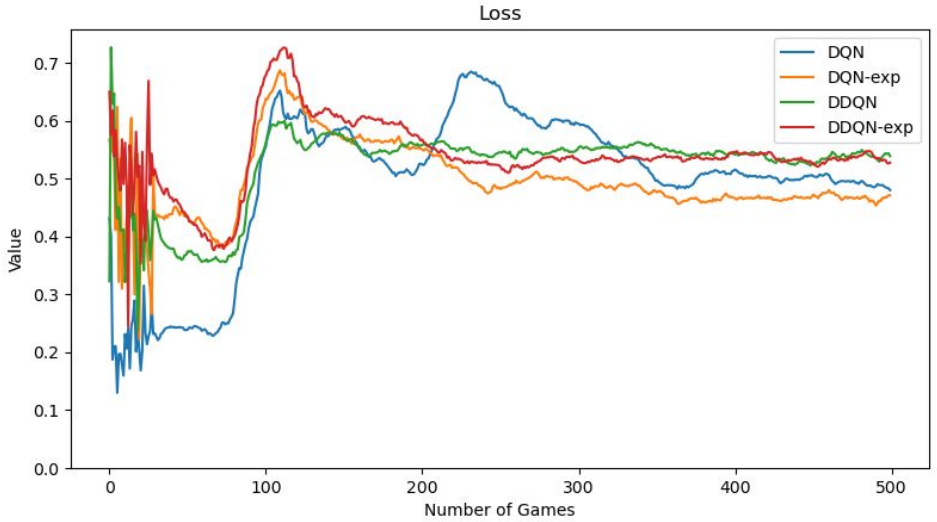
Before start: inject inside the agent's replay memory buffer some experiences derived from some episodes played by an human agent



Results and metrics

- Results reflects more or less what we could expect:
    - DDQN variants achieved highest scores
    - Variants with injected experience seem to perform better





Thank you for your attention!