Deep Q Network in a Stochastic Grid Environment: Navigating Turtles to Food

Luca Delcarmine

UNIVERSITÀ DEGLI STUDI DI MILANO

Overview and environment

- **Agent** (turtle) collects yellow dots (food)
- Avoids red dots (**enemies**) moving randomly
- **Objective**: Train the agent to play efficiently using DQN

Q-Learning and Bellman's Equation

$$\boxed{\mathrm{Q(s,a)} = \mathrm{R(s,a)} + \gamma \mathrm{max}_{a'} Q(s', a')}$$

$$\boxed{\pi(s) = \mathrm{max}_a \mathrm{Q}(s, a)}$$

$R(s, a)$: **reward** for taking action a in the state s

γ **: discount factor**

$\mathrm{Q}(s, a)$ **: quality** function

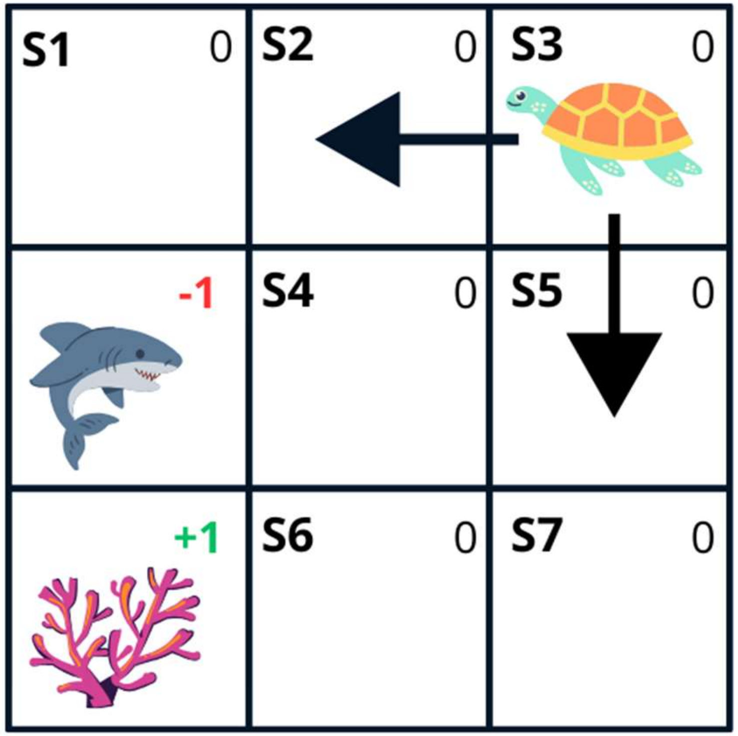
Q-Learning is a model-free reinforcement learning algorithm that aims to find the optimal policy for a Markov Decision Process by learning the value of action-state pairs through iterative updates based on observed rewards and transitions.

Why deep neural network? (1)

Simplified environment:

- 3x3 grid
- One enemy (shark)
- One food (coral)
- The enemy cannot move: **deterministic environment**

$$\mathcal{P}(s'|s,a) = \delta(\tilde{a} - a)$$



Why deep neural network? (2)

Q-table

$$Q(s_6, L) = 1$$

$$Q(s_7, L) = 0 + \gamma \times 1$$

$$Q(s_5, D) = 0 + \gamma \times \gamma$$

$$Q(s_3, D) = 0 + \gamma \times \gamma \times \gamma$$

$$\cdots$$

$$\gamma = 0,90$$

	R	L	U	D
S1	$0,73$	×	×	-1
S2	$0,66$	$0,66$	×	$0,81$
S3	×	$0,73$	×	$0,73$
S4	$0,73$	-1	$0,73$	$0,90$
S5	×	$0,81$	$0,66$	$0,81$
S6	$0,81$	$+1$	$0,81$	×
S7	×	$0,90$	$0,73$	×

Why deep neural network? (3)

- k undistiguishable enemies on a n-grid $k = 25 \quad n \times n = 49 \times 49$

$$\binom{n^2+k-1}{k} = \frac{(n^2+k-1)!}{k!(n^2-1)!} \sim 2,4 \cdot 10^{59}$$

- 2376 remainig spots for the turtle, for a total of $N_{\text{states}} \sim 5,6 \times 10^{62}$

- The **stochasticity** in the movement of the enemies would require also an estimation of the transition probability distributions, making it way harder to address the problem of finding the optimal Q-function

$$\Rightarrow \quad \hat{\mathrm{Q}}(s, a|\vec{\theta}) \approx \mathrm{Q}(s, a)$$

Deep Neural Network as **estimator** for the Q-function

DQN class: Model architerchture

Input State: The input state can be adjusted as needed, typically including distances from enemies and food.

```python
def build_model(self):
    model = Sequential()
    model.add(Dense(64, input_shape=(self.state_space,), activation='tanh'))
    model.add(Dense(128, activation='tanh'))
    model.add(Dense(128, activation='tanh'))
    model.add(Dense(128, activation='tanh'))
    model.add(Dense(64, activation='tanh'))
    model.add(Dense(self.action_space, activation='linear'))
    model.compile(loss='huber_loss', optimizer=Adam(learning_rate=self.learning_rate))
    return model
```

- Fully connected layers
- **Activation Function**: Hyperbolic tangent (tanh) is used for the inner layers due to its symmetry around zero, aligning with the distribution of state values
- **Huber Loss**: This loss function effectively combines Mean Squared Error (MSE) and Absolute Error (MAE), reducing the impact of outliers.

$$L_\delta(\hat{Q}(s,a), Q(s,a)) = \begin{cases} \frac{1}{2}(\hat{Q}(s,a) - Q(s,a))^2 & \text{for } |\hat{Q}(s,a) - Q(s,a)| \leq \delta, \\ \delta\left(|\hat{Q}(s,a) - Q(s,a)| - \frac{1}{2}\delta\right) & \text{for } |\hat{Q}(s,a) - Q(s,a)| > \delta \end{cases}$$

DQN class: Exploration vs Exploitation

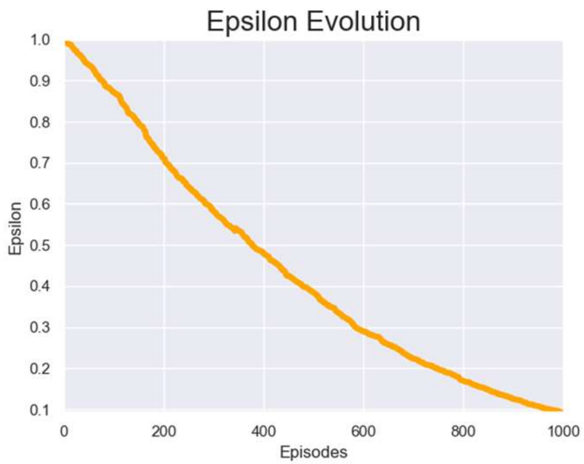
Epsilon-Greedy Policy:

- **Process**:
  - With probability ϵ, select a random action (**exploration**).
  - With probability $1-\epsilon$, select the action with the highest estimated reward (**exploitation**).
- **Decay of Epsilon**:
  - Start with $\epsilon = 1$.
  - Gradually decrease ϵ over time to shift towards exploitation as the agent learns more about the environment.
- **Avoiding Local Optima**: exploration helps the agent avoid getting stuck in suboptimal strategies by discovering new and potentially better actions.
- **Balancing Act**: too much exploration can lead to poor performance due to random actions, while too much exploitation can lead to missing out on better strategies not yet discovered.

```
self.epsilon = 1
```

```
self.epsilon_min = 0.01
self.epsilon_decay = 0.99999
```

```
if self.epsilon > self.epsilon_min:
    self.epsilon *= self.epsilon_decay
```



DQN class: temporal correlations (1)

- **Temporal Correlation:**

In this kind of Q-learning problems, consecutive data are highly temporally correlated. The model could then **overfit recent data** and drive the learning process towards substandard policies.

- **Replay Method:**
  - **Replay Buffer**: A memory buffer that stores past experiences as tuples (state, action, reward, next state, done).
  - **Random Sampling**: Instead of using consecutive experiences for training, the agent samples a random minibatch from the buffer. This disrupts the temporal correlations and ensures more stable and efficient training.
- **Advantages of Experience Replay:**

•**Breaks Temporal Correlation**: By randomizing the order of experiences, the temporal correlations are broken, leading to better learning.

•**Efficient Use of Data**: Experiences can be reused multiple times, improving data efficiency.

DQN class: temporal correlations (2)

Defined two double-ended queues, one for **generic experiences**,one for the **deaths**.

```
self.memory = deque(maxlen=1000)
self.deaths = deque(maxlen=100)
```

Created functions to populate the queues after the steps

```
def remember(self, state, action, reward, next_state, done):
    self.memory.append((state, action, reward, next_state, done))

def remember_prioritized(self, state, action, reward, next_state, done):
    self.memory.append((state, action, reward, next_state, done))
    if done:
        self.deaths.append((state, action, reward, next_state, done))
```

Used the **replay** or **replay_prioritized** functions to randomly sample **minibatches** of experiences from the queues.

```
def replay(self):
    if len(self.memory) < self.batch_size:
        return

    minibatch = random.sample(self.memory, self.batch_size)
    states = np.array([i[0] for i in minibatch])
    actions = np.array([i[1] for i in minibatch])
    rewards = np.array([i[2] for i in minibatch])
    next_states = np.array([i[3] for i in minibatch])
    dones = np.array([i[4] for i in minibatch])

    states = np.squeeze(states)
    next_states = np.squeeze(next_states)

    target_q_values = rewards +
    self.gamma * np.amax(self.target_model.predict_on_batch(next_states), axis=1) * (1 - dones)
    q_values = self.model.predict_on_batch(states)

    ind = np.array([i for i in range(self.batch_size)])
    q_values[[ind], [actions]] = target_q_values

    Hist = self.model.fit(states, q_values, epochs=1, verbose=0)
    self.losses = Hist.history['loss'][0]
    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay
```

DQN class: stabilization

➢ **Instability problems:**
As the primary network's weights change, the target network ones keep shifting, making the learning process unstable and prone to get stuck into **loops**.

➢ **Target network method:**

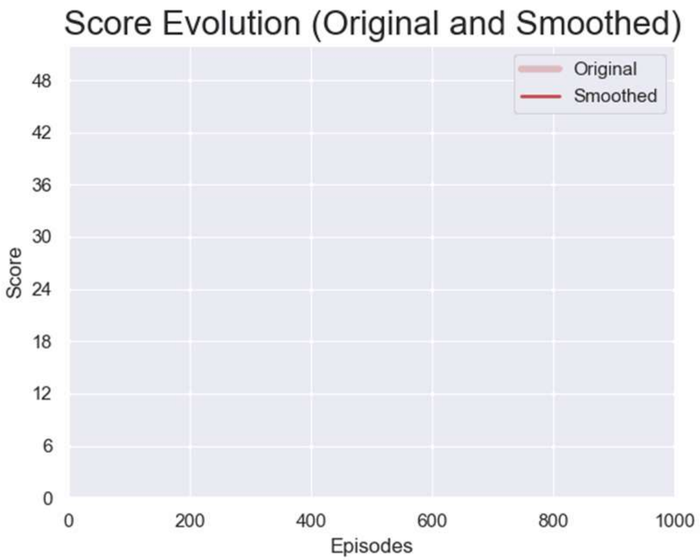
- A neural network with the same architecture is used to provide stable Q-value targets during training.
- The second network is updated less frequently providing a clearer target for certain number of episodes.
- When updated the primary network weights are copied into the target network.

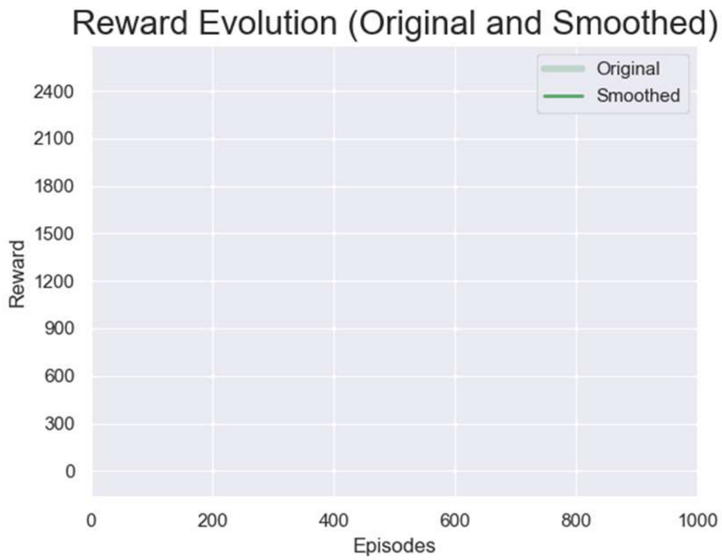
```
self.target_model = self.build_model()
```

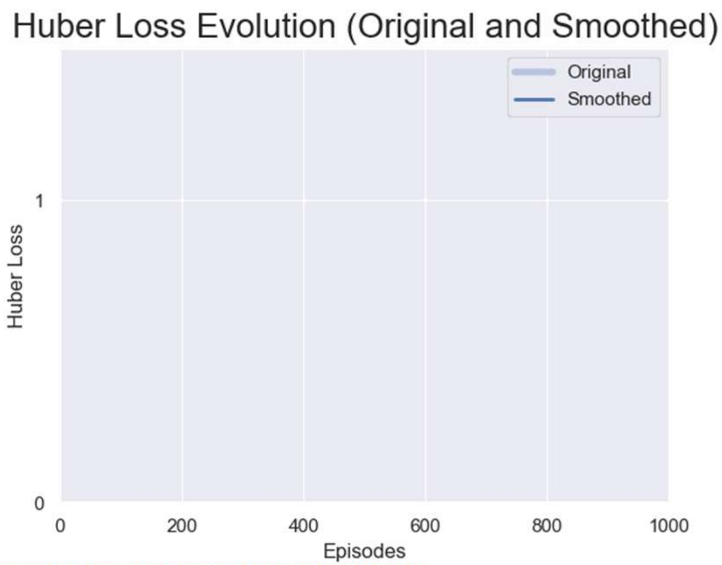
```
def update_target_network(self):
    self.target_model.set_weights(self.model.get_weights())
```

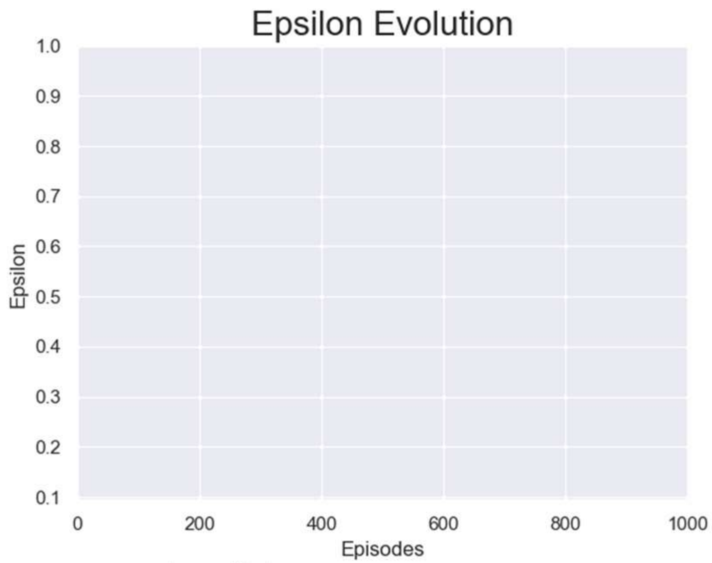
```
if e % target_update_freq == 0:
    agent.update_target_network()
    target_updates.append(e)
    print("Updated target network at episode {}".format(e))
```

$$Q(s,a) = R(s,a) + \max_{a'} Q_{target}(s',a')$$





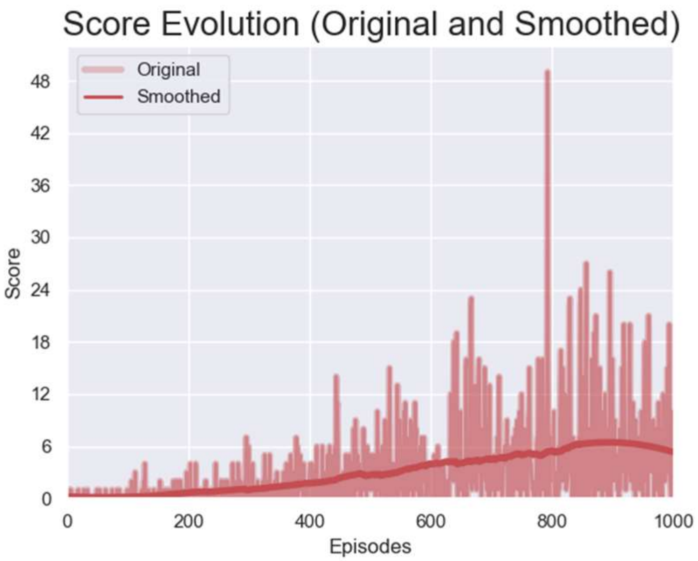


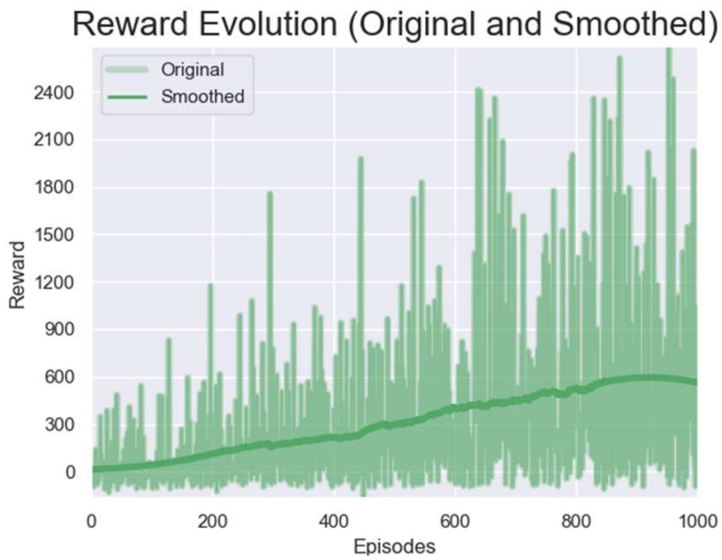


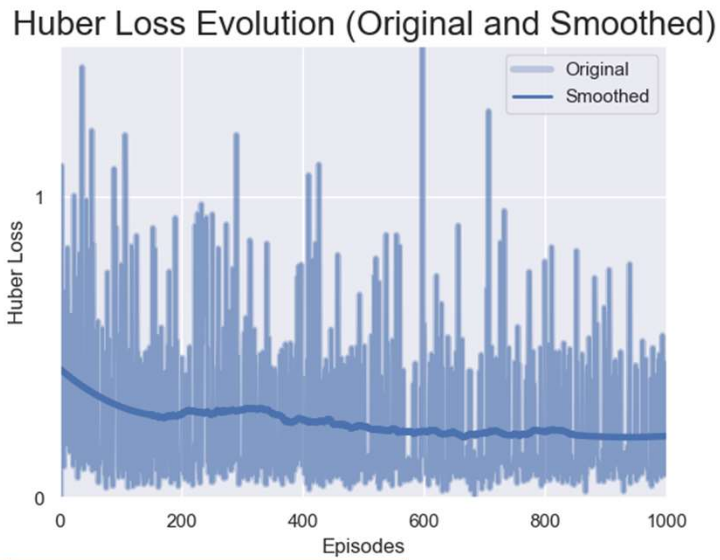
Results (1): Small state space agent

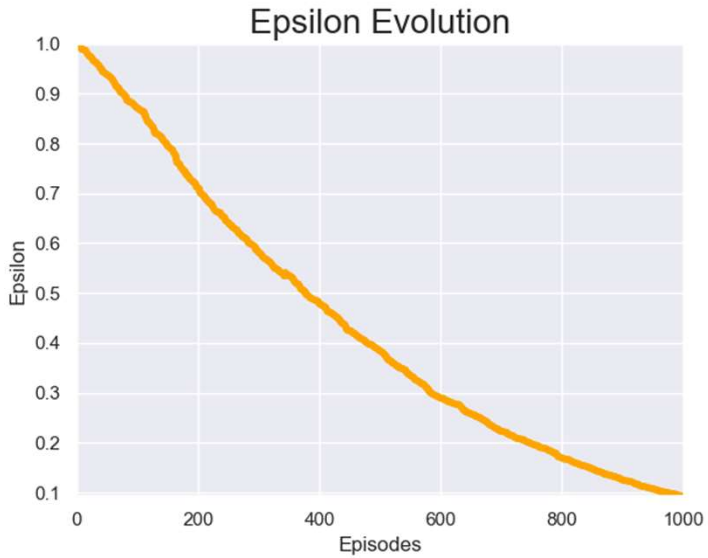
Agent features:

- State dimension = 12
- LR =0.001
- Minibatch size = 128
- S-Network update frequency = 1
- Epsilon decay = 0.99999
- Memory queue = 10000
- Reward function: (see code)





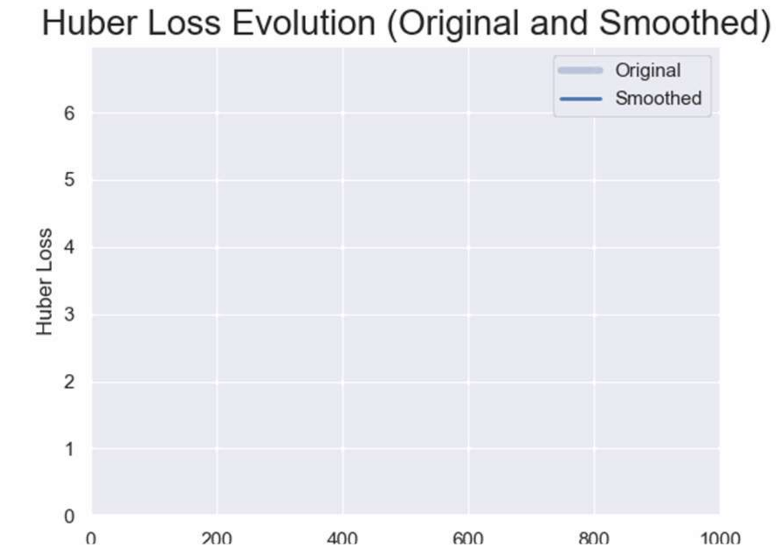


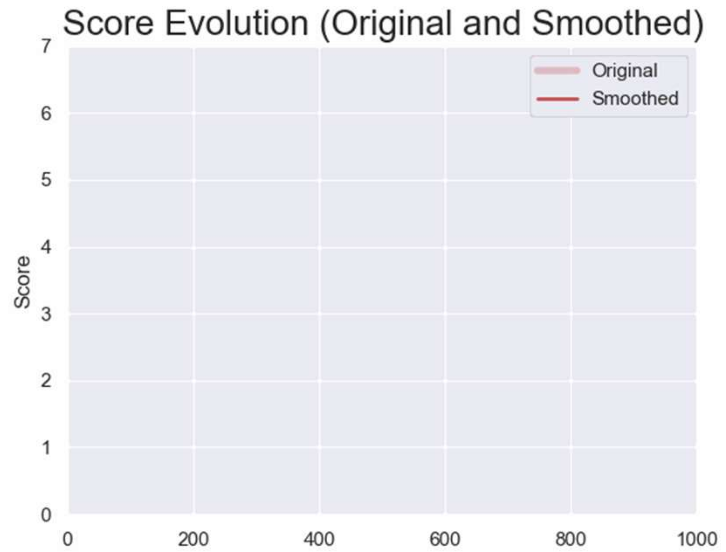


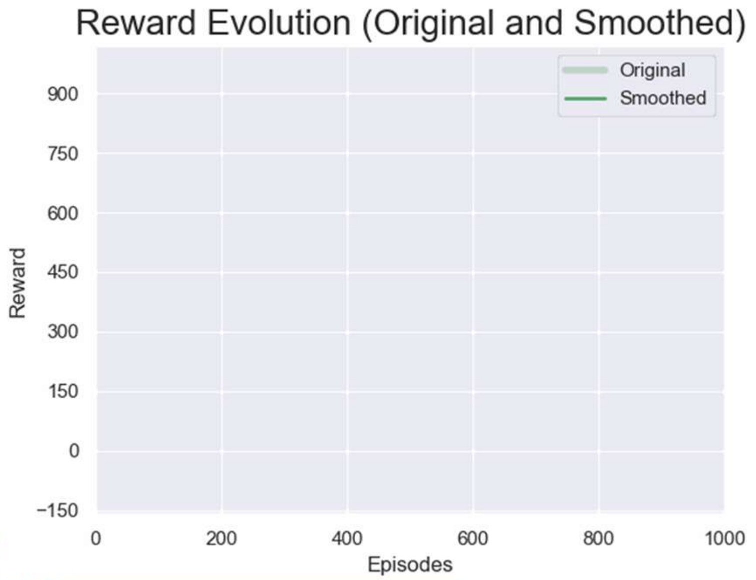
<u>Results (1): Small state space agent</u>

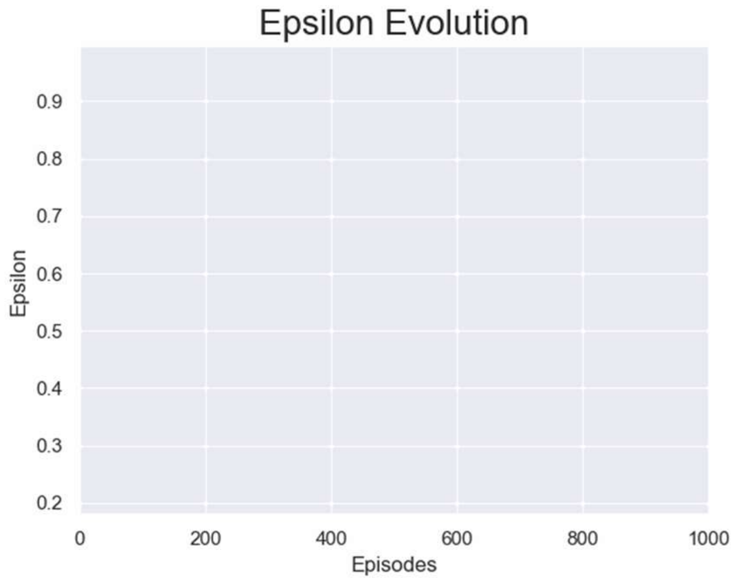
Agent features:

- State dimension = 12
- LR =0.001
- Minibatch size = 128
- S-Network update frequency = 1
- Epsilon decay = 0.99999
- Memory queue = 10000
- Reward function: (see code)





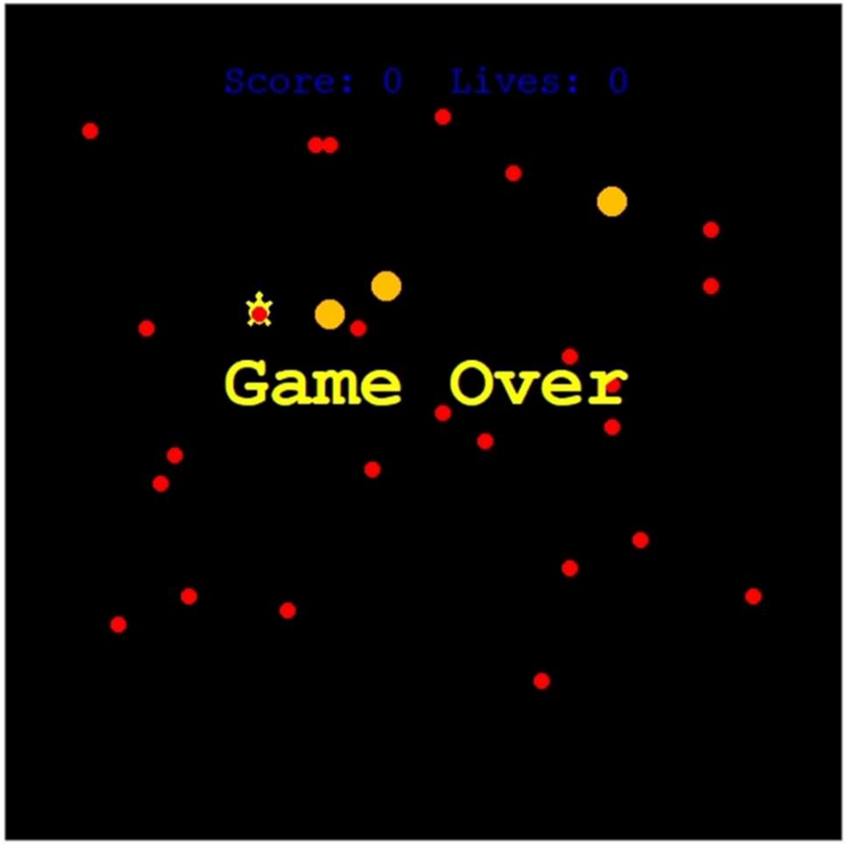




Results (2): Training failed

Agent features:

- State dimension = 20
- LR =0.001
- Minibatch size = 128
- S-Network update frequency = 15
- Epsilon decay = 0.99999
- Memory queue = 30000
- Reward function: (see code)


Score: 0 Lives: 0
Game Over