Deep Reinforcement Learning for the Unity "Banana collector" Environment

1. Introduction

The current report aims at solving the **Unity "Banana collector" environment** as described in the accompanying README file.

This is a **single-agent** environment that is also **stationary** (underlying dynamics do not change with time).

The current solution implements the Double Deep Q Network (**Double DQN**) agent with learning based on the **Huber Loss**.

2. Methodology

1) Double DQN agent

The agent trains a **double Q-network** because the action space is **discrete**.

This solution is preferred over the **vanilla Q-network** as the latter is noisy. In the **classical DQN**, the same function $Q_{\theta\_}$ is used to select the action $\arg\max_{a'}$, and evaluate that action's value. This introduces a **systematic bias**, because noisy estimates, when passed through a max operator, tend to be overestimated. So vanilla DQN slowly inflates Q-values and learning collapses.

$$y_t^{\text{DQN}} = r_t + \gamma(1 - d_t) \max_{a'} Q_{\theta^-}(s_{t+1}, a')$$

In the double DQN, the action a^* is calculated with the online Q_θ before being evaluated by $Q_{\theta\_}$.

$$a^* = \arg\max_{a'} Q_\theta(s_{t+1}, a')$$

$$y_t = r_t + \gamma\,(1 - d_t)\, Q_{\theta^-}(s_{t+1}, a^*)$$

In this setup, the agent uses the online Q_θ to contribute experience to the replay buffer, while the learning is offline through the target $Q_{\theta\_}$ and the online Q_θ.

2) Learning from the Environment Reward

At each time step t, the agent receives a reward from the environment as follows :

$$r_t = \begin{cases} +1 & \text{if a yellow banana is collected at step } t \\ -1 & \text{if a blue banana is collected at step } t \\ 0 & \text{otherwise} \end{cases}$$

The learning process will use it as is.

The objective is to learn a **navigation policy** that maximises the **expected cumulative reward** by efficiently collecting yellow bananas and avoiding blue bananas.

Because this reward is **sparse, discrete**, the Q-network will infer **long-term value** through **temporal credit assignment (**using **bootstrapping** in the **TD target**). This is a major difference with **continuous control** that benefits from continuous **geometric hints** to give guidance, making the learning a smooth **hill-climbing problem**.

3) Replay buffer

The solution uses a uniform **experience replay buffer** (fixed-size ring buffer) that stores each transition from the online data collection:

$$(s_t, a_t, r_t, s_{t+1}, d_t)$$

Where $s_t$is the state, a_t is the action, r_t is the reward, s_{t+1} is the next state, and d_t is the done status (terminal state or not).

$$\begin{aligned} s_t &\in \mathbb{R}^{37} \\ a_t &\in \{0, 1, 2, 3\} \\ r_t &\in \mathbb{R} \\ s_{t+1} &\in \mathbb{R}^{37} \\ d_t &\in \{0, 1\} \end{aligned}$$

The size of the buffer "**buffer_size**' is a default hyperparameter, which value is set to 50000.

The replay buffer makes the training **off-policy**, **decorrelated** (random mini batches from the replay buffer almost make the training data **independent and identically distributed (i.i.d.)**), stable, and sample efficient. Without experience replay, the

learning target would be strongly **correlated** with the network updates leading to severe instability and divergence.

4) Network architectures

a) Online Q_θ

For each state *s*, the Q-network estimates the **expected return per discrete action**:

$$Q_\theta : \mathbb{R}^{37} \longrightarrow \mathbb{R}^4$$

$$Q_\theta(s) = \begin{bmatrix} Q_\theta(s,0) \\ Q_\theta(s,1) \\ Q_\theta(s,2) \\ Q_\theta(s,3) \end{bmatrix}$$

Architecture:

- MLP with **2 hidden layers** (128, 128).
- ReLU activations on both hidden layers
- Linear output layer with 4 units

$$\begin{aligned} z^{(1)} &= W_1 s + b_1 \\ h^{(1)} &= \mathrm{ReLU}(z^{(1)}) \\ z^{(2)} &= W_2 h^{(1)} + b_2 \\ h^{(2)} &= \mathrm{ReLU}(z^{(2)}) \\ z^{(3)} &= W_3 h^{(2)} + b_3 \\ Q_\theta(s) &= z^{(3)} \in \mathbb{R}^4 \end{aligned}$$

Where :

$$\begin{aligned} W_1 &\in \mathbb{R}^{128\times 37} \\ W_2 &\in \mathbb{R}^{128\times 128} \\ W_3 &\in \mathbb{R}^{4\times 128} \end{aligned}$$

b) Target network Q_{θ^-} initialization and update

The **Temporal Difference (TD) Q-target** Q_{θ^-} is a **hard full copy** (parameters are not learned and the copy is not slow moving like with Polyak averaging) of the online Q-network, performed every "**target_update_every**" environment steps (default hyperparameter, which value is set to 100).

5) Training process

a) $\varepsilon - greedy$ actions

The exploration policy is $\epsilon - greedy$.

$$a_t = \begin{cases} \arg\max_a Q_\theta(s_t, a) & \text{with probability } 1 - \varepsilon_t \\ \text{Uniform}\{0, 1, 2, 3\} & \text{with probability } \varepsilon_t \end{cases}$$

ϵ controls the **exploitation-exploration trade-off** and is set to decrease from $\boldsymbol{\epsilon_0}$ (initial epsilon) to $\boldsymbol{\epsilon_{min}}$ (min epsilon), according to a **schedule** (linear or exponential) **after warm-up**. The latter variables are default hyperparameters, which values are respectively 1, 0.05 and '**exp'** (for exponential).

Linear decay schedule:

$$\varepsilon_t = \max\left(\varepsilon_{\min},\ \varepsilon_0 - (\varepsilon_0 - \varepsilon_{\min}) \cdot \frac{t}{T}\right)$$

Exponential decay schedule (decay is faster in the beginning, and much slower later on):

$$\varepsilon_t = \varepsilon_{\min} + (\varepsilon_0 - \varepsilon_{\min}) \exp\left(-\frac{t}{\tau}\right)$$

Where ***t*** is the number of environment steps taken so far and ***T*** is the total number of steps over which ε decays ('**decay_steps'**). The latter is a default hyperparameter set to 300000.

Because **the schedule is exponential** in this experiment, the parameter τ is used.

It is defined as **decay_steps** x **tau_ratio**. '**Tau_ratio'** is a default hyperparameter set to 0.3, hence τ is 90000.

Early in training, actions are mostly **uniformly random** and later in training, they are mostly greedy. Without exploration from the start, the agent would repeatedly choose the same bad action, and large parts of the environment would never be explored.

The exploration policy is used to generate the training data. The noisy transitions are then fed into the replay buffer. This avoids on-policy greedy trajectories and prevents the buffer from collapsing into local optima very quickly.

The exploration action is not used in the optimisation process to avoid polluting gradients.

In the log file, the ratio "**mean_exploration_ratio_over_100_episodes_from_training**" tells how much exploration is injected in each episode. It is calculated as follows:

- During **training** (not evaluation), every time the agent chooses an action, it falls into one of the two categories (greedy action or exploration action)

$$a_t = \arg\max_a Q_\theta(s_t, a)$$

$$a_t \sim \text{Uniform}\{0, 1, 2, 3\}$$

- The script keeps two counters per episode: num_random_actions and num_total_actions. The **episode exploration ratio** is then calculated per episode:

$$\rho_{\text{episode}} = \frac{\text{num\_random\_actions}}{\text{num\_total\_actions}}$$

- The ratio "**mean_exploration_ratio_over_100_episodes_from_training**" is computed then as the moving average (MA) of $\rho_{episode}$ over the last 100 episodes. In the experiment, it decreases from 0.76 to 0.09 when the performance goal is reached.

b) greedy actions

A greedy action is directly output by Q_θ as the best and is used only in optimization and evaluation.

$$a^* = \arg\max_{a'} Q_\theta(s_{t+1}, a')$$

c) *Online data collection*

Transitions are collected online with noisy actions during training, into the replay buffer.

$$(s_t,\ a_t,\ r_t,\ s_{t+1},\ d_t)$$

d) *Optimisation*

Optimisation starts after the size of the replay buffer D has reached the below threshold (**warm-up**):

$$|\mathcal{D}| > (\text{batch\_size}) \times (\text{n\_warmup\_batches})$$
$$\mathcal{D} = \{(s_t,\ a_t,\ r_t,\ s_{t+1},\ d_t)\}$$

"**n_warmup_batches**" is a default hyperparameter set to 5. Hence, the warmup threshold is 320 transitions.

During optimization, **sampling** from the replay will be based on a **minibatch**, the size of which is the '**batch_size**' default hyperparameter and is set to 64. The sampling is **uniform random without replacement within a batch**.

Note that optimisation takes place in sync with online data collection in training, with the $\boldsymbol{Q_\theta}$ **network** update performed **at every environment step**.

Target networks are updated according to hyperparameter "**target_update_every**".

Because the optimisation is based on the replay buffer, it is called **off-policy pattern** (**collect online, optimise offline**).

e) *Online Q_θ update with Huber Loss*

For each transition from the minibatch, the next action *a** is computed directly with the online greedy Q-network:

$$a^* = \arg\max_{a'} Q_\theta(s_{t+1}, a')$$

Next, the action a^* is evaluated with the target network:

$$Q_{\theta^-}(s_{t+1}, a^*)$$

Then **TD target** is computed:

$$y_t \;=\; r_t \;+\; \gamma\,(1-d_t)\,Q_{\theta^-}(s_{t+1},a^*)$$

Gamma is set to 0.99. Note that if the same rapidly-changing network Q_θ were used to build the TD target instead of $Q_{\theta-}$, the TD target would move every gradient step, leading to divergence. In this scenario, we would have :

$$r_t + \gamma \max_{a'} Q_\theta(s_{t+1}, a')$$

This scenario would create correlated data, leading to unstable updates, as gradient descent assumes i.i.d. samples.

Next, the predicted value for the action effectively taken (from the replay buffer's transition) is calculated with the online Q-network.

$$\hat{q}_t \;=\; Q_\theta(s_t,a_t)$$

Then, the Huber loss is output:

$$\mathcal{L}_{\text{Huber}}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{if } |\delta| \le \kappa \\ \kappa\left(|\delta| - \frac{1}{2}\kappa\right) & \text{if } |\delta| > \kappa \end{cases}$$

With δ, the **TD error** being :

$$\delta_t \;=\; \hat{q}_t - y_t$$

And threshold parameter k = 1 by default.

The derivative of the loss is then :

$$\frac{\partial \mathcal{L}}{\partial \delta} = \begin{cases} \delta & |\delta| \le \kappa \\ \kappa\,\text{sign}(\delta) & |\delta| > \kappa \end{cases}$$

As a result :

- In the case of small errors $|\delta| \le k$, the loss behaves like MSE, and the gradients grow linearly.
- In the case of large errors $|\delta| > k$, it behaves like MAE and the gradient is bounded at $\pm\kappa$ (no explosion).
- Once learning stabilises and TD errors shrink, the loss naturally transitions into precise MSE-like learning.

The old gradients of the loss are cleared from memory before the new ones are computed with respect to the parameters via backpropagation. The gradients are accumulated across all transitions from the batch and averaged.

After backpropagation, we clip the **total gradient norm**, as a global safety brake on the final update magnitude. This clipping rescales every parameter/weight gradient by the same factor.

$$\|\nabla_\theta \mathcal{L}\| \leftarrow \min(\|\nabla_\theta \mathcal{L}\|, c)$$

Where c is a hyperparameter and is set to 10. As a reminder, the **total gradient norm** is simply the **Euclidean length** of all the parameter gradients.

$$\|\nabla_\theta \mathcal{L}\| = \sqrt{\sum_k \sum_{i,j} \left(\nabla_{\theta_k}^{(i,j)}\right)^2}$$

Where θ_k is the *k-th* parameter tensor of the network, and (*i,j*) are the row and column indexes of the tensor. It measures the **overall strength of the update signal** produced by the current batch.

Then, the parameters are updated via Adam update rule.

$$\theta \leftarrow \theta - \alpha \, \text{Adam}(\nabla_\theta \mathcal{L}(\theta))$$

Alpha α is the **learning rate**. It is a hyperparameter. Its best value is found to be 0.0004.

f) Early stopping

Training stops when any of these triggers:

- The goal performance is achieved (the environment is solved!) when the 100-episode MA of the episodic return score exceeds +13.
- The wall-clock elapsed time exceeds 2000 minutes.
- The total number of episodes exceeds 2000.

g) Evaluation

i. Online evaluation during training

After **every training episode**, exactly **1 greedy episode** (no noise) is run using the online Q-network. Then, the agent accumulates its **undiscounted episodic return**:

$$R = \sum_{t=0}^{T-1} r_t$$

Where T is the episode length and r_t is the raw environment reward at time step t. The **evaluation score** for that episode is simply this return. Then a **100-episode MA** of the evaluation returns is computed.

This MA is used for early stopping if it reaches or exceeds +13 and if there are at least 100 values in the MA calculation.

This MA is used in the plot together with the episodic return score.

ii. Final evaluation

After training finishes, the final Q-network is evaluated over 100 episodes without noise. The **mean and standard deviation** of these 100 episode scores are then calculated, providing a low-variance and reliable estimate of the final Q-network's performance.

h) Training sequence

Per training episode:

- Run one episode in Unity environment using noisy actions from the online Q-network.
- Store each step into replay.
- Once warm-up is passed, do minibatch updates every step.
- Update targets every "target_update_every" steps.
- End episode when Unity environment signals done.
- Evaluate immediately**:** run 1 greedy episode (no noise), calculate the episodic return, and record that evaluation score.
- Save an checkpoint model.<episode>.tar.

After training ends:

- Run final evaluation of 100 greedy episodes with the final model, return mean ± std.

i) Network Usage and Reward Flow Across Training Phases

The below table highlights how explorative and greedy actions are used by the data collection, optimisation and evaluation phases.

Phase	Online Q-network	Target Q-network	Minibatch from replay
Data collection into replay buffer	X (explorative strategy)		
Optimisation	X (greedy strategy)		X (explorative strategy)
Evaluation	X (greedy strategy)		

3. Results

1) Best model over seeds

The run with the default hyperparameters found that :

- **785** episodes were required to exceed the goal score of +13, with a 100-episode MA of evaluation score of +13.04 at stop during training.
- However, the final evaluation performance (post training) exhibits a poorer score of **7.47.** To get a better sense of the real performance of the models created with the default hyperparameters, it would be necessary to run multiple seeds, but it would be costly in terms of running time as a seed run takes 6-8 hours.

Based on the assignment requirements, it can be concluded that the current experiment surpasses Udacity's reference (785 < 1800 episodes).

The weights of the best online policy model are then extracted using the script extract_weights_n_load_into_policy.py.

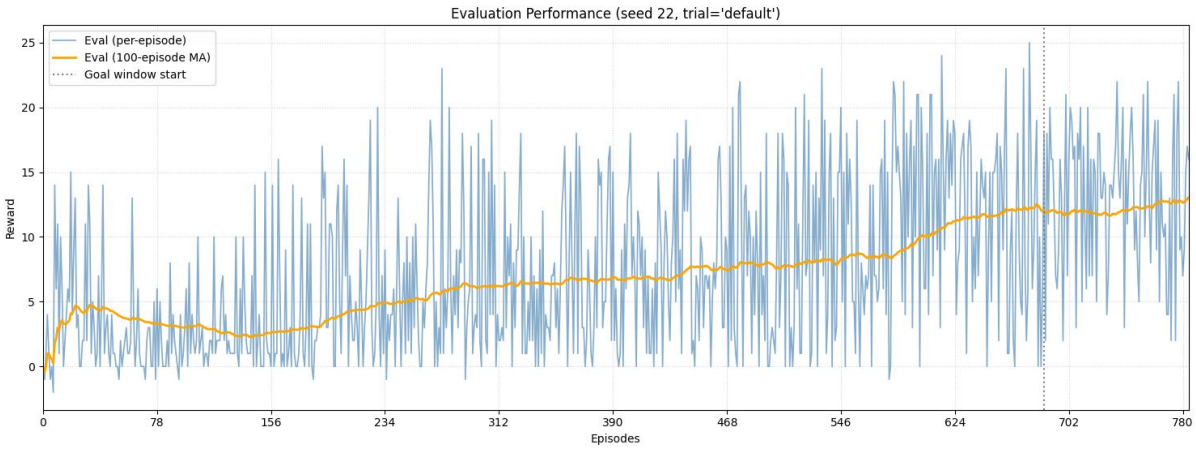
The extracted weights and plot can be found in ./ best_model_and_weights_and_plot/ folder.

./ best_model_and_weights_and_plot/

model.784.tar
model.784.weights.npz
model.784.weights.pt

2) Plot of performance of during training

The below plot shows that the 100-episode MA score (orange curve) has almost a steady linear progress while the episodic score exhibits a very high variance around the orange curve.



4. Suggestions for future work

1) Incorporate Dueling Network Architecture

The current Q-network directly estimates $Q(s, a)$. A **duelling architecture** decomposes this into:

$$Q(s, a) = V(s) + A(s, a)$$

where A is **the Advantage function network**.

This helps the agent learn which states are valuable even when actions do not differ significantly, often leading to faster learning and better generalisation in navigation tasks.

2) Use Multi-Step Returns

The TD target currently uses **1-step bootstrapping**. Using n**-step returns** would propagate reward information faster across time and reduce **credit assignment delay**.

$$y_t = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n \max_a Q(s_{t+n}, a)$$