1. Project 1: Navigation

1.1 Goal

The goal of this project is building a deep reinforcement learning (DRL) agent that navigates an environment with the idea of collect as many yellow bananas as possible while avoiding purple bananas.

When the agent collects a yellow banana, the reward is +1, but if the agent collects a purple banana, the reward is -1. To solve this environment, our agent must get a score of +13 over 100 consecutive episodes.

1.2 Environment

The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around the agent's forward direction. Given this information, the agent must learn how to best select actions. Four discrete actions are available, corresponding to:

- 0 : move forward
- 1 : move backward
- 2 : turn left
- 3 : turn right

1.3 RL Algorithm

Reinforcement learning is learning what to do -how to map situations to actions- so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them.

One of the challenges that arise in reinforcement learning, and not in other kinds of learning, is the trade-off between exploration and exploitation. To obtain a lot of reward, a RL agent must prefer actions that it has tried in the past and found to be effective in producing reward. But to discover such actions, it has to try actions that it has not select before. The agent must exploit what it has already experience in order to obtain reward, but it also has to explore in order to make better actions in the future. To solve this, we use Epsilon Greedy Algorithm.

The main objective of RL algorithm is to find an optimal policy. The optimal policy must be discovered by interacting with the environment and the agent learns the policy through a process of trial-and-error. In order to solve this, we use Q-learning.

1.2.1 Q-learning

Q-learning is a model-free reinforcement algorithm. The goal of Q-learning is to learn a policy, which tells an agent what actions to take under what circumstances. The Q-function calculates the expected reward for all possible actions in all possible states.

We can define our optimal policy as the actions that maximizes the Q function for a give state across all possible states.

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

Figure 1. Q-Algorithm

1.2.2 Deep Q-Network (DQN)

The value functions as described in the preceding section are high dimensional objects. To approximate them, we can use a DQN: Q(s,a,w) with parameters w. To estimate this network, we optimize the following sequence of loss function at iteration i.

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}\left[\left(y_i^{DQN} - Q(s,a;\theta_i)\right)^2\right]$$

Figure 2. Loss function

$$y_i^{DQN} = r + \gamma \max_{a'} Q(s', a'; \theta^-)$$

Figure 3. Y – DQN

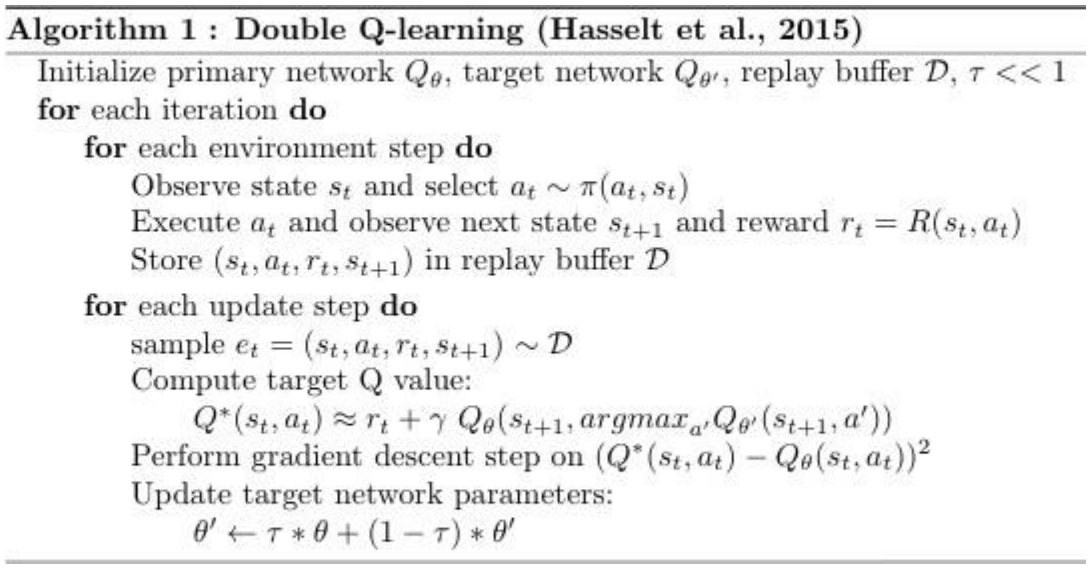
Where the O’ represents the parameters of a fixed and separate target network. The parameters of the target network are freeze for a fixed number of iterations while updating the online network by gradient descend.

Another key is experience replay. During learning, the agent accumulates a dataset of experiences from many episodes. The network is trained by sampling mini batches of experiences from D uniformly at random. Experience replay increases data efficacy through re-use of experience samples in multiple updates and it reduces variance as uniform sampling from the replay buffer reduce the correlation among the samples used in the update.

1.2.3 Double Q-Network (DDQN)

One issue with DQN is they can overestimate Q-values. The accuracy of the Q-value depends on which actions have been tried and which states have been explored. Overestimations occur even when assuming we have samples of the true action value at certain states.

The idea of DDQN is to reduce overestimations by decomposing the max operation in the target into action selection and action evaluation. Although not fully decoupled, the target network in DQN architecture provides a natural candidate for the second value function, without having to introduce additional networks. We therefore propose to evaluate the greedy policy according to the online network but using the target network to estimate its value.



```
Algorithm 1 : Double Q-learning (Hasselt et al., 2015)
Initialize primary network $Q_\theta$, 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:
            $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$
```

Figure 4. DDQN Algorithm

1.2.4 Dueling Q-network

The key insight behind our net architecture is that for many states, it is unnecessary to estimate the value of each action choice.

In some states, it is of paramount importance to know which action to take, but in many other states the choice of actions has no repercussion on what happens.

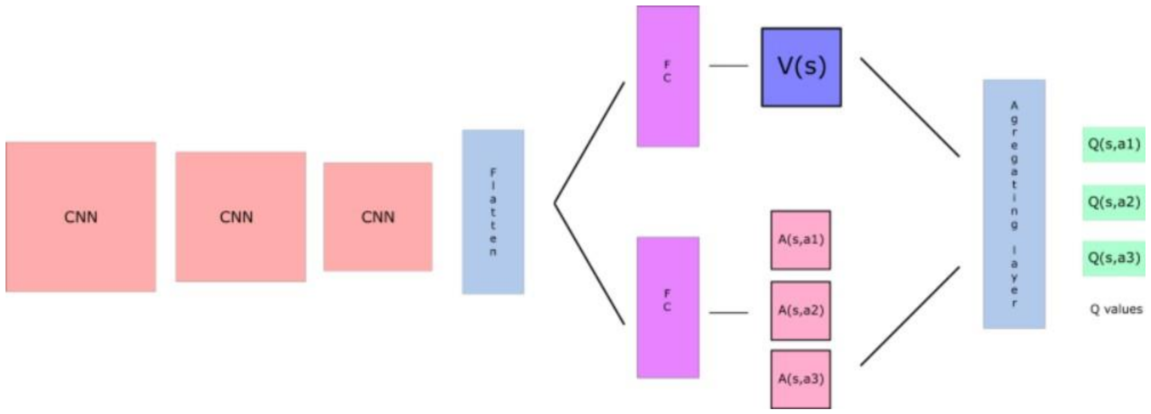


Figure 5. Dueling Architecture

By decoupling the estimation, intuitively our DDQN can learn which states (or are not) valuable without having to learn the effect of each action at each state (since it's also calculating V(s)).

$$Q(s,a;\underset{\text{Common network parameters}}{\boxed{\theta}},\underset{\text{Advantage stream parameters}}{\boxed{\alpha}},\underset{\text{Value stream parameters}}{\boxed{\beta}}) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \underbrace{\frac{1}{\mathcal{A}}\sum_{a'} A(s,a';\theta,\alpha))}_{\text{Average advantage}}$$

Figure 6. Q value Dueling Q network

2. Experiments

2.1 DQN

```
Episode 100	Average Score: 0.53
Episode 200	Average Score: 3.74
Episode 300	Average Score: 6.77
Episode 400	Average Score: 10.62
Episode 500	Average Score: 12.77
Episode 512	Average Score: 13.01
Environment solved in 412 episodes!	Average Score: 13.01
```

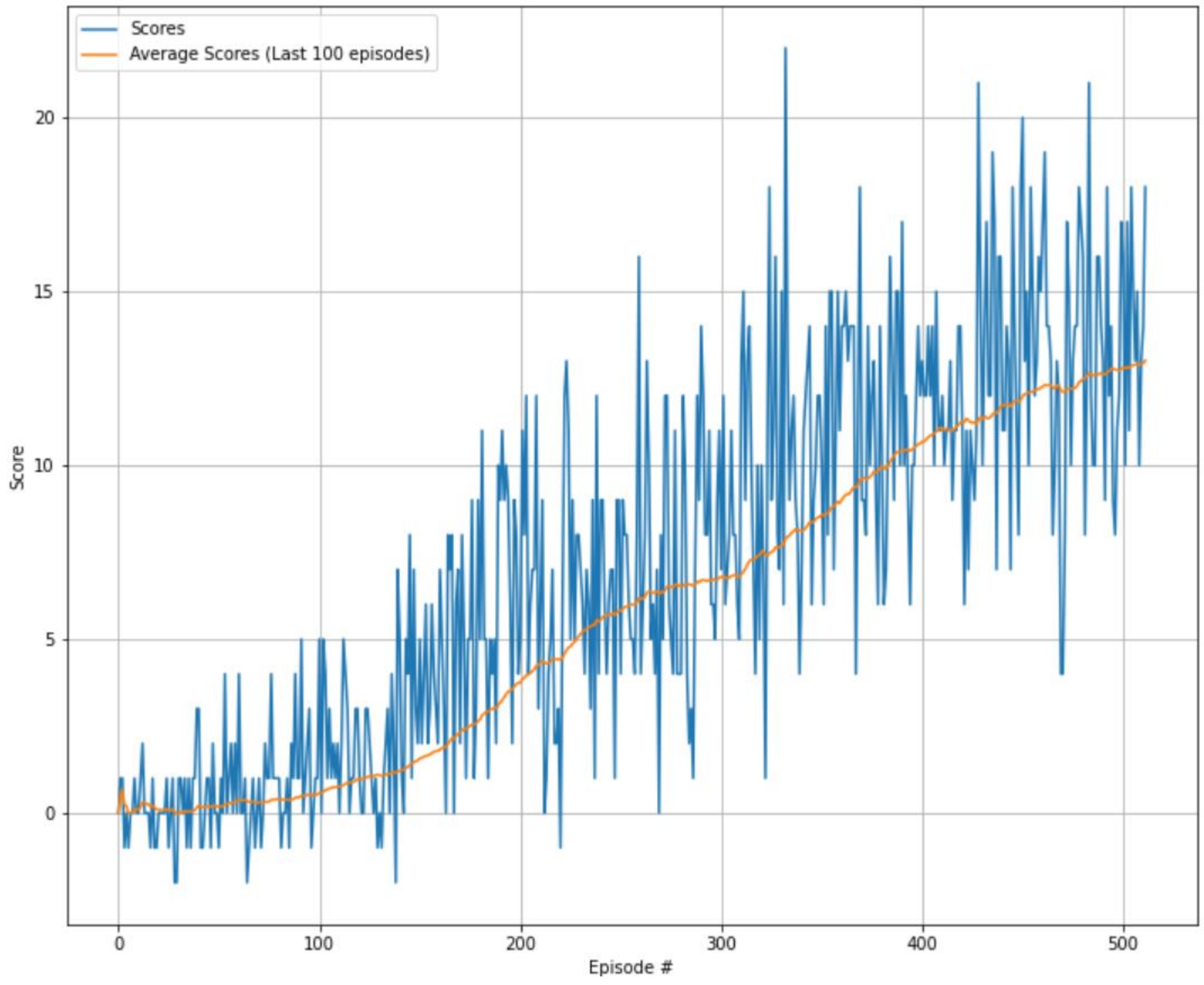
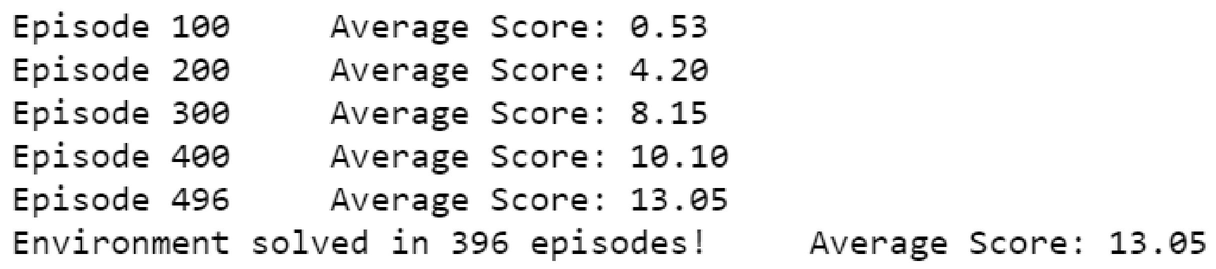


Figure 7. Train DQN Agent

2.2 Double DQN



```
Episode 100     Average Score: 0.53
Episode 200     Average Score: 4.20
Episode 300     Average Score: 8.15
Episode 400     Average Score: 10.10
Episode 496     Average Score: 13.05
Environment solved in 396 episodes!     Average Score: 13.05
```

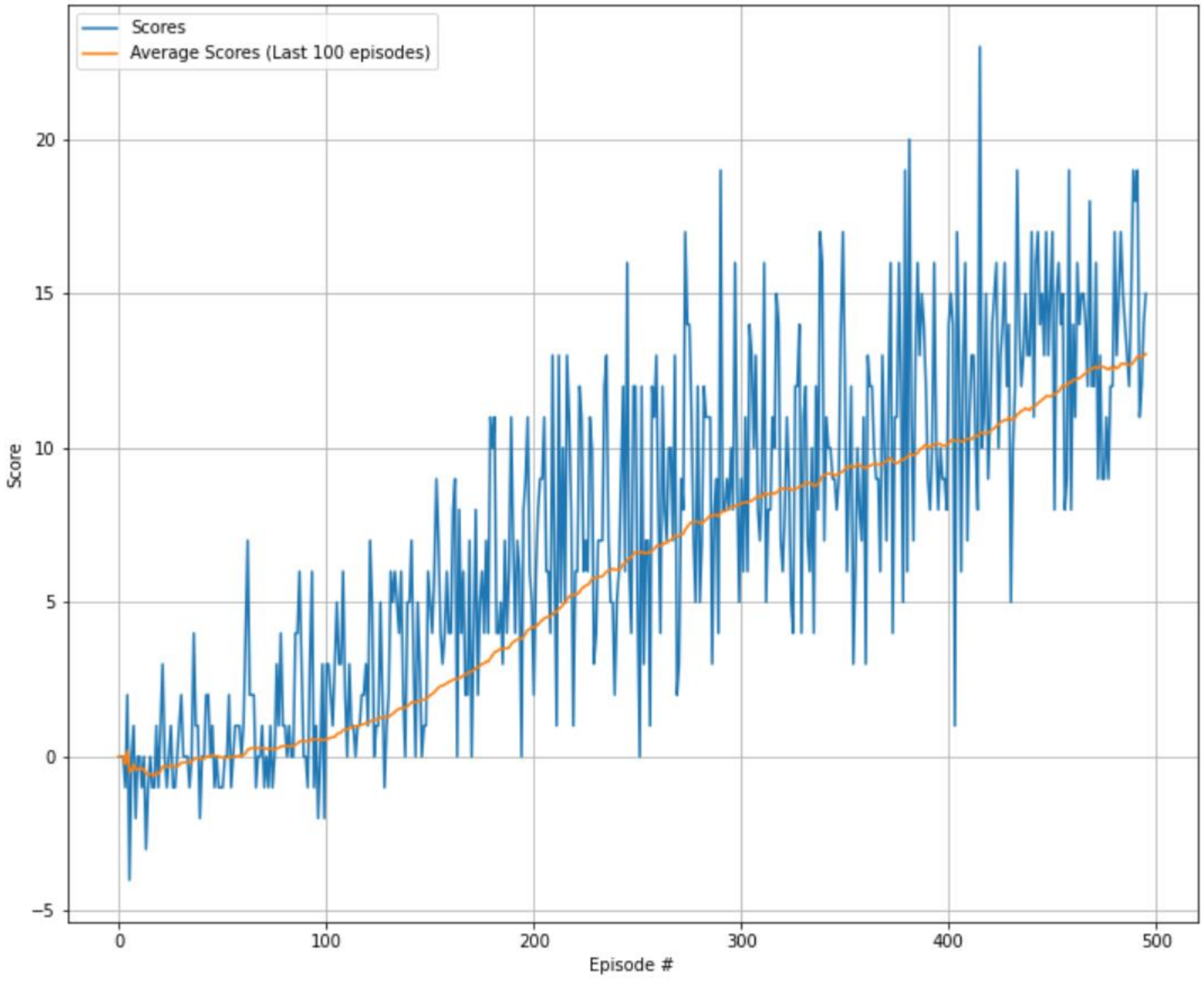


Figure 8. Training Agent using Double DQN

2.3 Dueling DQN

```
Episode 100     Average Score: 0.49
Episode 200     Average Score: 3.34
Episode 300     Average Score: 6.88
Episode 400     Average Score: 10.02
Episode 497     Average Score: 13.08
Environment solved in 397 episodes!     Average Score: 13.08
```

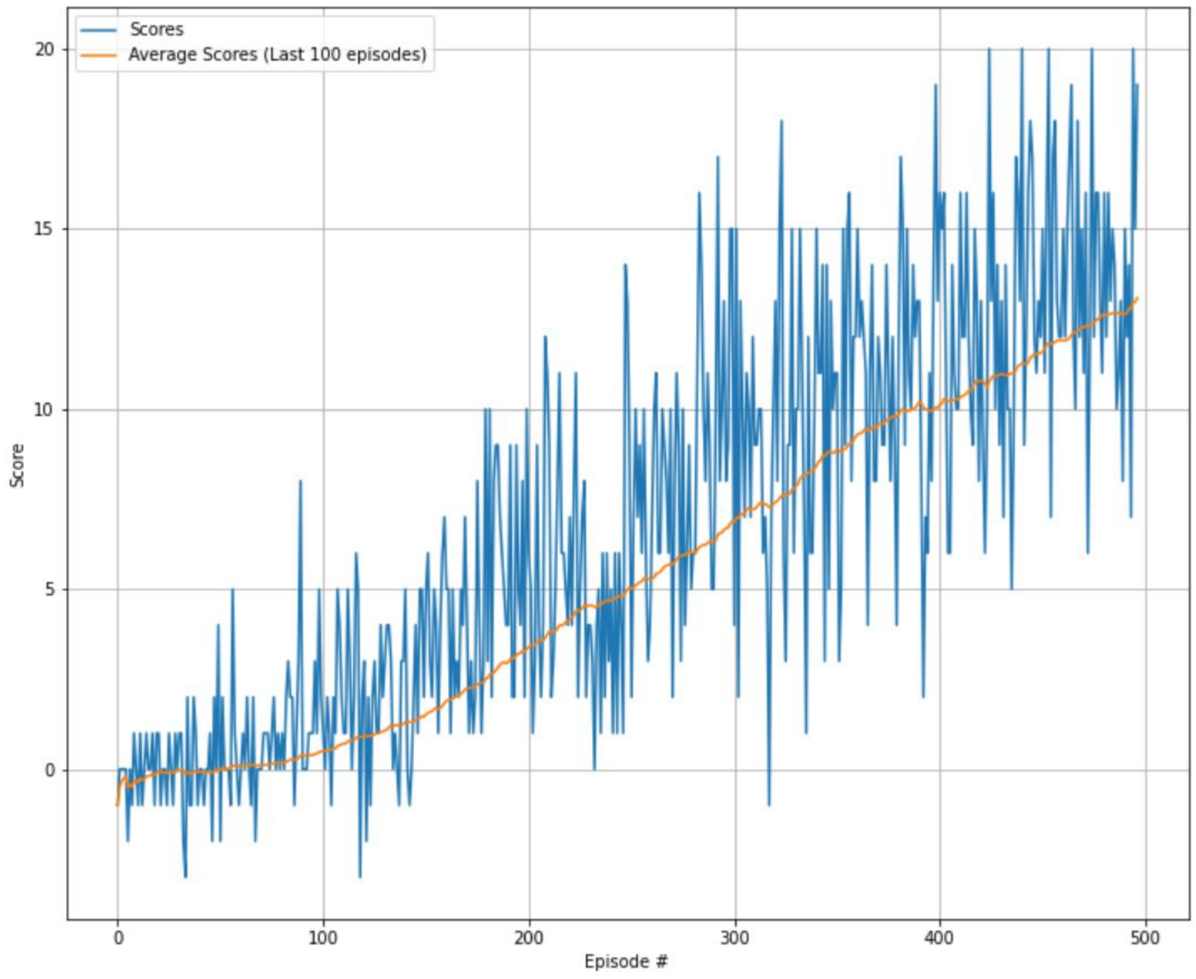
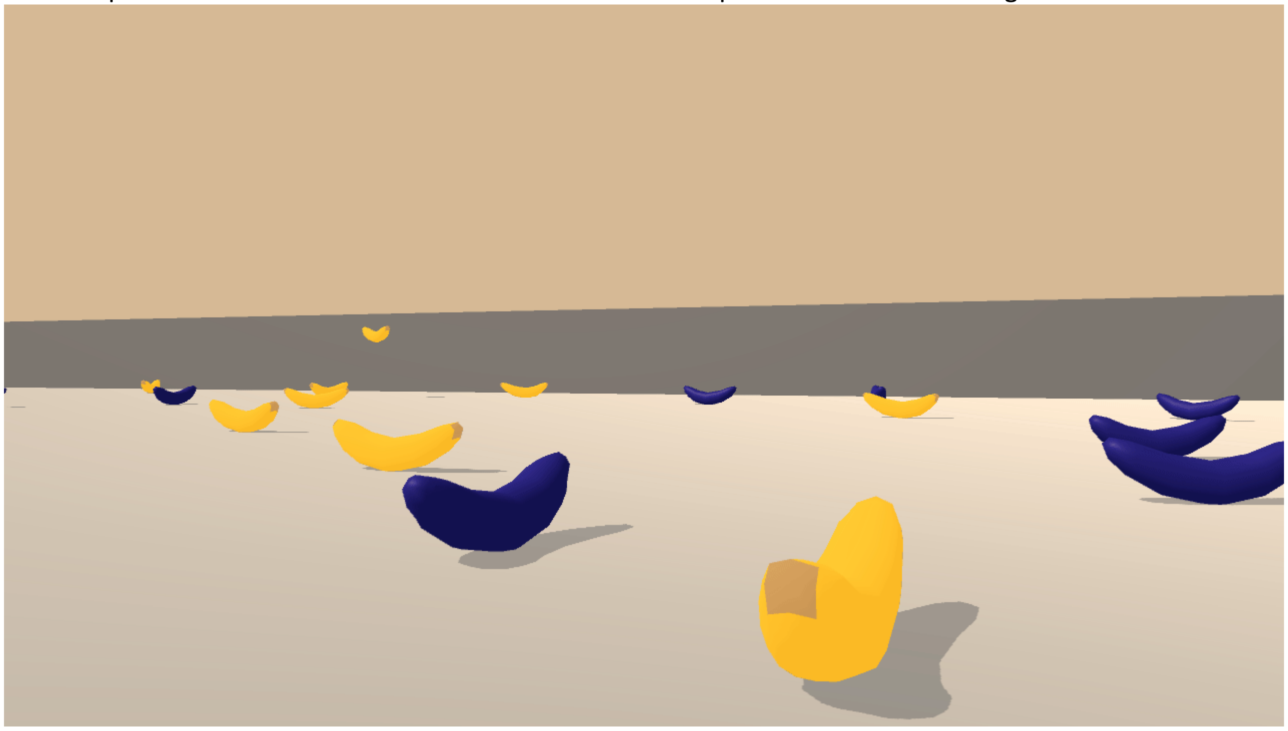


Figure 9. Train Agent Using Dueling DQN

3. Summary of the experiments

It has been observed that the agent is able to learn quite quickly. It is true that it takes several hundred episodes, but this is partly due to the fact that it learns by using epsilon-greedy, since at the beginning our agent takes certain random actions, but for each episode, it reduces these random actions according to the value of eps_decay, and it takes quite a few episodes before the value of epsilon is close to 0 so that the agent performs the most appropriate movements. A summary of the results from the previous experiments is presented in the next table. It can be seen how all three experiments manage to resolve the environment in a similar range of episodes. Even so, the Double DQN algorithm solves it faster than the others, probably due to the improvements offered by this algorithm with respect to the traditional DQN. An example of the behaviour of the model once trained are provided in the following GIF.



ARCHITECTURE	EPISODES TO SOLVE
DQN	412
DOUBLE DQN	396
DUELING DQN	397

4. Next Steps

The algorithm is improvable, and it is possible to make different improvements that can improve the results and speed up the training process. Some of them are:

1. Optimize the hyperparameters of the process and network by using other strategies like random search, hyperband, Bayesian optimization.
2. Change the network architecture replacing the ANN with other architectures (LSTM, CNN, Transformers, ...).
3. Test other algorithms like Rainbow.
4. Add prioritized replay.
5. Contenarise the proposed approach using Docker to improve the portability and scalability of the code. It is possible to create a docker container instead of a conda env to share the code and notebooks without manually installing any dependencies.
6. Once the application is containerized, we can implement it using Kubeflow to put them into production and scale it depending on the needs of the problems. Additionally, we can train our networks using different workers in a distributed way. The amount of workers depends on the time we want to solve the problem, taking into account that while more speed trains the algorithm, more hardware is used, and hence, the cost of training the agent could increase. Using Kubeflow, we can implement Katib to found the best combination of hyperparameters and transform the code focusing on further production application