Report: Deep Reinforcement Learning Nanodegree Project 1: Bananas

Michaela G. Lawrence

25 November 2020

1 Introduction

This project uses deep reinforcement learning to play a video game effectively. The agent or player is placed into a space containing blue and yellow bananas, surrounded by walls. See Figure 1 for an example of what can be seen from the perspective of a player. The player or agent will score points by collecting yellow bananas and loose points if it collects blue bananas.

This project utilised the architecture for solving the DQN coding exercise as outlined in `solution/Deep_Q_Network_Solution.ipynd`. The details of parameter changes can be found below in Sections 4 and 5.

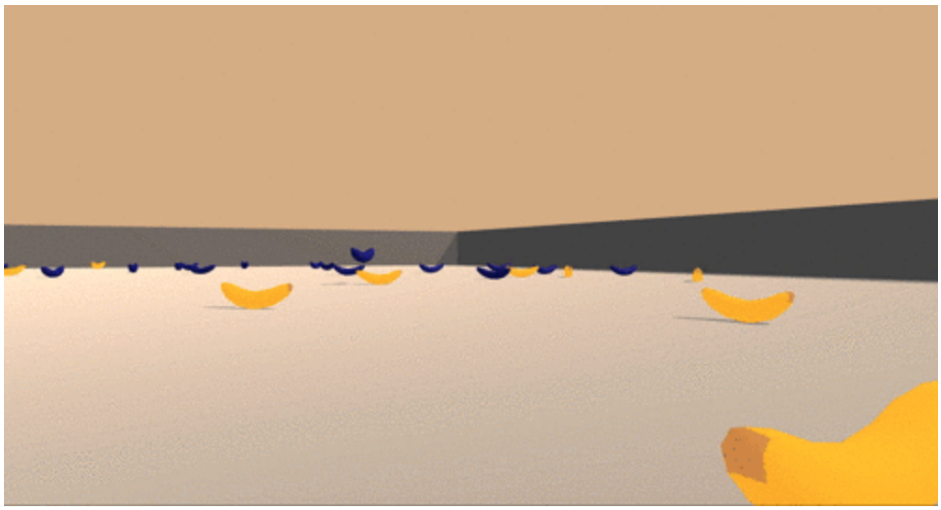

Figure 1: An example of what a player might see when starting the banana game.

2 State and action space

The simulation contains a single agent that navigates a large environment. At each time step, it has four actions at its disposal:

- `0` - walk forward
- `1` - walk backward
- `2` - turn left
- `3` - turn right

The state space has `37` dimensions and contains the agent's velocity, along with ray-based perception of objects around agent's forward direction. More information on the `37` dimensions can be found in Appendix A.

3 Learning algorithm

The agent training utilised the `dqn` function in the navigation notebook.

It continues episodic training via a DQN agent until `n_episodes` is reached or until the environment is solved. The environment is considered solved when the average reward (over the last 100 episodes) is at least +13.

Each episode continues until `max_t` time-steps is reached or until the environment says it's done.

A reward of `+1` is provided for collecting a yellow banana, and a reward of `-1` is provided for collecting a blue banana.

The DQN agent is contained in `dqn_agent.py`.

For each time step the `dqn_state` acts on the current state and epsilon-greedy values. The `dqn_agent` utilises a replay buffer of experiences.

4 DQN hyper parameters

- `n_episodes` (int): maximum number of training episodes
- `max_t` (int): maximum number of timesteps per episode
- `eps_start` (float): starting value of epsilon, for epsilon-greedy action selection

- `eps_end` (float): minimum value of epsilon
- `eps_decay` (float): multiplicative factor (per episode) for decreasing epsilon

where: `n_episodes=400`, `max_t=5000`, `eps_start=0.55`, `eps_end=0.01` and `eps_decay=0.97`.

The epsilon-greedy values were found considering the following, knowing `eps_start`, `eps_end` and `eps_decay`, one can find the total number of episodes to reach `eps_end` from `eps_start`, we call this n,

$$n = \frac{\ln(\texttt{eps\_end}) - \ln(\texttt{eps\_start})}{\ln(\texttt{eps\_decay})}. \tag{1}$$

Considering that one will want `eps_end` to be small but non-zero so that there is always a small chance that instead of taking the "best so far" action the agent will take a random action, this ensures the action space is well explored. `eps_start` is chosen to be higher than `eps_end`, in the beginning the agent doesn't have much experience so it will need to take some random actions to explore the space. it was found that initial random actions could often lead to positive average rewards for early episodes.

It is possible for `eps` to never reach `eps_end`, this occurs when n is larger than the total number of episodes needed to solve the system. This is something I considered when choosing the three `eps` parameters. For the final choice of `eps` parameters, listed at the beginning of this section $n \sim 132$.

Considering the above the three `eps` parameters were adjusted by trial and error, beginning with the values given in `solution/Deep_Q_Network_Solution.ipynd` of `eps_start=1.0`, `eps_end=0.01`, `eps_decay=0.995`.

Choosing a high number of time steps per episode is intuitively beneficial. The agent has more time to collect yellow bananas and therefore reach a higher score. Of course at the beginning of the learning process it will likely also collect more blue bananas, reducing the score, but this is valuable for training. With this in mind I initially chose `max_t=10000` (ten times the default). The other parameters, were adjusted with this setting. When I eventually settled on choosing `eps_start=0.55`, `eps_end=0.01` and `eps_decay=0.97`, solving the environment in 211 episodes, I noticed that towards the end of an episode when there were not many yellow bananas left but there were still a lot of blue ones the agent would just sort of walk into the wall. It was obvious at this point (from around episode 25) that the agent had, to some extent, learned that collecting blue bananas was disadvantageous so at the end of each episode when the agent was just walking into a wall it wasn't learning much from this experience. I then reduced `max_t` to `5000` and managed to achieve solving the environment in the same number of episodes.

At first a high number of episodes was chosen but as I adjusted the other parameters and the number of total episodes taken to solve the environment came down this number became arbitrary.

5 DQN agent hyper parameters

- `BUFFER_SIZE` (int): replay buffer size

- BATCH_SIZE (int): mini batch size
- GAMMA (float): discount factor (the closer to 1 the more "importance" the agent places on expected future. A value of 1 means that the agent considers expected return at all future time steps equally important)
- TAU (float): for soft update of target parameters
- LR (float): learning rate for optimizer
- UPDATE_EVERY (int): how often to update the network

where: BUFFER_SIZE = int(1e6), BATCH_SIZE = 128, GAMMA = 0.99, TAU = 1e-3, LR = 5e-4 and UPDATE_EVERY = 4

GAMMA, TAU, UPDATE_EVERY and LR remained at the default values adjusting them individually up and down was not beneficial.

It seemed that increasing BUFFER_SIZE also increased performance. With this in mind we increased it from the defult value by a factor of 10. Increasing BATCH_SIZE also seemed to benefit training by being increased, so it was doubled from the default value.

6 Neural network

The QNetwork model utilise 2 x 64 Fully Connected Layers with Relu activation followed by a final Fully Connected layer with the same number of units as the action size. The network has an initial dimension the same as the state size.

7 Plot of rewards

The best results can be found in Figure 2.

```
Episode 100 Average Score:  4.19
Episode 200 Average Score:  10.03
Episode 300 Average Score:  12.39
Episode 311 Average Score:  13.03

Environment solved in 211 episodes!  Average Score:  13.03
```

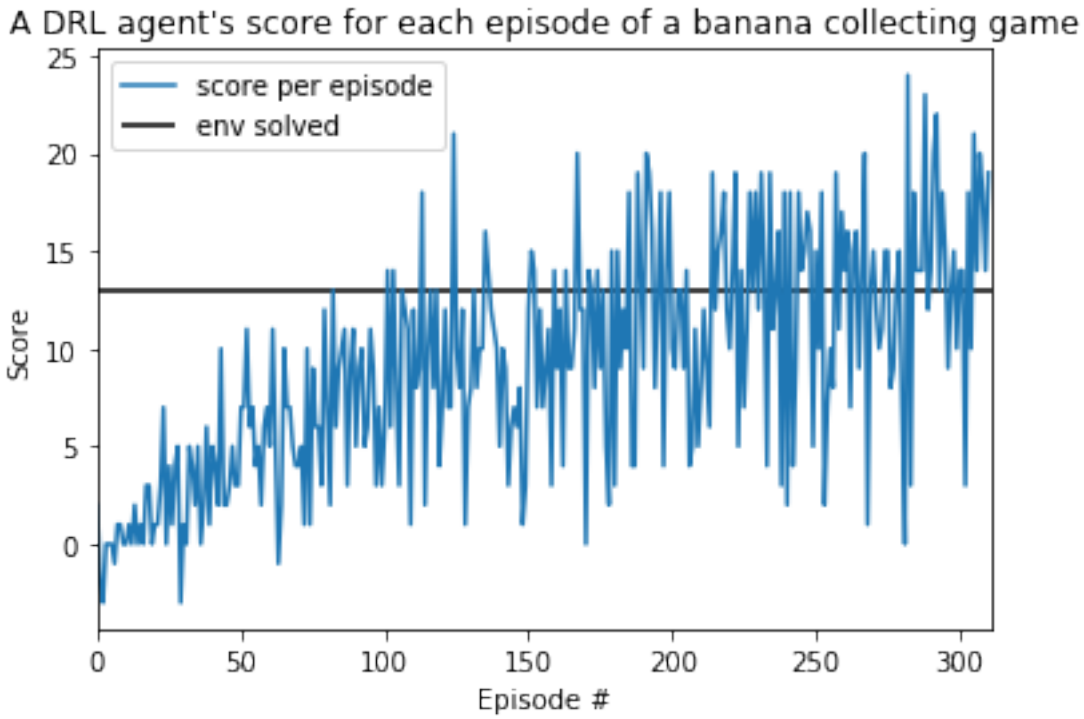

Figure 2: The best results.

8 Ideas for future work

Automated hyper parameter tuning:
The initial random generation of actions would often generate relatively high average scores. Time was spent twiddling the epsilon-greedy values manually over multiple runs. Thus with some automated hyper parameter tuning process, this process could be optimised to find the best settings in an automated fashion.

Common ways to improve the performance of DQNs:
Table 8 summarises six common methods of improving DQNs.

But what should be considered is what would be practical and useful to impose on the DQN for this particular problem? Considering this criteria, I think three methods stand out: double DQN, dueling DQN and prioritized experience replay. Double DQN seems like it would be relatively easy to implement and so would at least be worth trying. Dueling DQN would be a bit more work but could train the agent using less episodes. Prioritized experience replay would again like double DQN would be fairly easy to implement however given the small size of the state space and action space I am not sure if we would see much benefit from implementing this method on this particular problem.

Method name	Problem addressed	Solution method
Double DQN	Overestimation of action values (can happen at the beginning of learning due to not yet having gathered enough information)	Select action using one set of parameters, but evaluate it using a different set w' (we can use w^- if we have already incorporated fixed Q-targets). This prevents the algorithm from propagating incidental high rewards obtained by chance.
Prioritized experience replay	Some experiences collected during experience replay may be more important for learning than others. These important experiences might occur infrequently.	Instead of sampling from the batch of experiences uniformly where these rare but important experiences have a small chance of being selected, select according to a probability. This sampling probability is proportional to the TD error plus some small number. Why proportional TD error? The bigger the TD error the more we expect to learn from that experience. Why the addition of a small number? No experience has zero probability of being selected. NOTE: This reduces the number of batch updates needed to learn a value function.
Duelling DQN	Slow speed of identifying the correct action during policy evaluation.	Use two separate streams (estimators), one that estimates the state value function and another that estimates the advantage for each action. Final Q-values are obtained by combining the state and advantage values. The duelling DQN can learn which states are (or are not) valuable without having to learn the effect of each action at each state. NOTE: This requires more significant changes to the DQN architecture.
Multi-step bootstrap targets	Bias-variance tradeoff (appears when we compare Monte Carlo with TD learning).	If Monte Carlo method over-fits (high variance) and TD method under-fits (high bias), then it is natural to consider the middle ground. We do not want to only consider the last action, but we also do not want to consider all actions made. Hence the name "multi-step" bootstrap targets.
Distributional DQN	Unaccounted for intrinsic randomness of the rewards and transitions of the agent within its environment	The approaches described so far all directly approximate the expected return in a value function, instead we could approximate the distribution of possible cumulative returns (value distribution) . This value distribution provides more complete information of the intrinsic randomness of the rewards and transitions of the agent within its environment
Noisy DQN	Insufficient or inefficient exploration.	A noisy DQN is a DQN with parametric noise added to its weights, the induced stochasticity of the agent's policy can be used to aid efficient exploration. The parameters of the noise are learned with gradient descent along with the remaining network weights. NOTE: Noisy DQN is straightforward to implement and adds little computational overhead.

Table 1: Table showing possible improvements to DQN. Information from this table was gathered from training videos found in the Udacity deep reinforcement learning nano degree and from "An Introduction to Deep Reinforcement Learning" by Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare and Joelle Pineau (2018), Foundations and Trends in Machine Learning: Vol. 11, No. 3-4. DOI: 10.1561/2200000071.

A 37 dimensions

It is not important to really understand what is the meaning of the `37` dimensions, the agent will after all learn from the `37` parameters how to solve the environment. However, we are told that the dimensions are "the agent's velocity, along with ray-based perception of objects around agent's forward direction" that is to say that the input is not the colour of each pixel on the screen - if this were the case this would change the nature of the problem.

So if the dimensions are not the colours of the pixels the agent can see what are they?

Velocity has two dimensions, a magnitude (speed) and direction.

The remaining 35 are "ray-based perception of objects around agent's forward direction". These dimensions are explained well an ml-agents issue we show the relevant text below:

7 rays projecting from the agent at the following angles (and returned in this order):
`[20, 90, 160, 45, 135, 70, 110]` # 90 is directly in front of the agent.

Ray (5):
Each ray is projected into the scene. If it encounters one of four detectable objects the value at that position in the array is set to 1. Finally there is a distance measure which is a fraction of the ray length. `[Banana, Wall, BadBanana, Agent, Distance]`

example:
`[0, 1, 1, 0, 0.2]`
There is a BadBanana detected 20% of the way along the ray and a wall behind it.