Banana Collector – Project 1

Problem description

The agent is to navigate an enclosed area and collect yellow bananas that are scattered about the floor. The agent must avoid picking up the blue bananas that are also scattered around the environment.

A reward of +1 is given for each yellow banana collected and -1 is given for each blue banana collected. The task is episodic (meaning it has a terminal state, either time (number of steps) bound or based on achieved state) and is considered complete when the agent gets an average score of +13 over 100 consecutive episodes.

The state space of the environment has 37 dimensions. It contains the agent's velocity along with a ray-based perception of objects around the agent's forward direction in a continuous observation space.

For the action space, there are 4 discrete actions which correspond to: 0 (move forwards), 1 (move backwards), 2 (turn left) and 3 (turn right). The agent can choose any of these at each timestep.

Proposed approach

The approach to solving this environment will be to apply Double Deep Q-Learning with Performance Experience Replay, adapting code from the Udacity Lunar Lander task. Some hyperparameters shall be identified and changed to show their effect on the agent's training speed.

The rest of the report talks about Deep Q-Learning, the improvements made through Double Deep Q-Learning and Performance Experience Replay. Following this is a section on what hyperparameters have been changed and how their values affect training. The report is ended with a conclusion and ideas for future work.

Deep Q-Learning

Deep Q-Learning is an algorithm that uses neural networks, called Deep Q Networks (DQN), to represent the optimal action-value function [1]. In Q-Learning a table is used to represent the action-value function, where the table would store the expected value for each action for each state. This is fine for small environments where there are only a handful of discrete states and actions, but the table soon becomes too large for more complex environments or when continuous state spaces are introduced. A neural network can replace the table, taking a state vector as an input and returning a vector of values, one for each action. The policy used during training is the same as in Q-Learning, where an epsilon-greedy policy is used to choose the next action. To enable exploration and exploitation during training, a *decay rate* of epsilon is used during training to change epsilon over time. This *decay rate* is a hyperparameter. The learning algorithm also includes *gamma*, the discount factor that affects the expected future reward. This is another hyperparameter in the learning algorithm.

To help with training experience data is stored in a replay buffer, as this allows the algorithm to train the neural network on many more experience samples than just the previous, most recent, experience and also allows previously visited but rare and costly states to be used in training.

Sampling from this replay buffer and using the Q-Learning algorithm enables the neural network to train. However, as each action affects the next state, there is correlation between consecutive experience samples. To combat this the experiences are retrieved from the replay buffer using random sampling, therefore breaking correlation.

Another aspect to Deep Q-Learning is using fixed targets. In table-based Q-Learning, the algorithm used to update the table uses itself to find the difference between what it received at that step in the episode and what it expected to receive. Using this same method in Deep Q-Learning would mean using the difference in the output of the DQN at two different states to update itself. This correlation is broken by using two neural networks, one to find the old expected value at the current state and one to find the new expected value (found by summing the received reward plus the expected value from the next state). The copy of the network (termed the q target network) is used in finding the new expected value. The weights for the q target network are initialised to be the same as the current network (termed the q local network). During training, the q local network is updated every timestep whilst the q target network kept fixed at each timestep and is only updated every C timesteps, where C is a hyperparameter.

A slightly different way to update the q target network's weights is to move the weights towards the ones of the q local network by a small amount every timestep, rather that performing a larger change in weights every C timesteps. Performing the smaller update is termed a soft update to the weights and the amount the weights move towards those of the q local network is controlled by a hyperparameter *tau*.

Improvements to Deep Q Networks

Double Deep Q Networks

The DQN algorithm has been shown to overestimate the action values either in noisy environments or when the estimate of the action value function is inaccurate [2]. This has been addressed by Double Deep Q-Learning (DDQN). In the standard DQN algorithm, when finding the expected value from the next state, the max operator is used to find the action that returns the largest action value. Therefore, this means the same network is used to select an action and evaluate the action making it more likely to select overestimated values. Double DQN decomposes these by selecting the action using the q local network and evaluating the action using the q target network. Whilst not fully decoupled as the two networks are related, it separates out the choice of the next action and the evaluation of it without the need to introduce a new network and more complexity. Especially towards the beginning of training, the DDQN algorithm will likely result in actions selected to find the temporal difference target that do not fully maximise the action value function of the q target network.

Prioritised Experience Replay

The replay buffer used in DQN allows the DQN algorithm to sample from the buffer and learn from the past experiences, breaking down the correlation of using consecutive experience data when training. However, some experiences may occur infrequently, and these may also be particularly important and when sampling uniformly, it is unlikely these rare experiences are recalled. Prioritised Experience Replay addresses this by attaching a priority to each experience [3]. The priority is calculated from the temporal difference error, where intuitively the larger the error the more the algorithm can learn from this experience data. With more to learn from it, the algorithm would want to see this experience data multiple times. Once an experience sample has been used in training, its priority is updated to be the new temporal difference error, which should decrease over time.

The experience data is stored in the replay buffer with its priority. This priority is converted into a probability by raising the priority to the power of hyperparameter *a* and dividing that by the sum of all priorities to power of *a*. Hyperparameter *a* introduces some randomness back into the sampling. If this was not applied, then sampling by priority would mean that only the highest priorities would be sampled with the training algorithm missing out on all other experiences. For *a*, if set to zero this is the equivalent of uniform sampling as all experience probabilities become identical. If *a* is set to one, then the sampling will heavily favour those with the highest probabilities. From the paper, *a* is set to be 0.6.

Now that sampling experiences is no longer uniform, the network update rule must be adjusted. Non-uniform sampling introduces a bias in the network towards the high priority samples. To counter this, importance sampling weights are used to reduce the effect that the repeatedly seen high priority experience samples have on the network update. The importance sampling weight is raised to the hyperparameter *b*, which controls how much the weights affect learning. As *b* is set to start at 0.4 and is annealed towards 1 as training goes on. This has the effect of increasing the correction (i.e. reducing the effect of) of experiences that are sampled more frequently than others. From the paper [3], *b* is set to be 0.4.

Hyperparameters

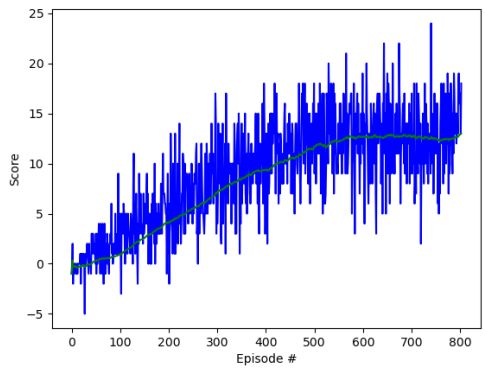
The following section varies some hyperparameters. These are, along with their default values:

- Epsilon decay rate = 0.995
- Gamma = 0.99
- Tau = 0.001
- a = 0.6
- b = 0.4
- b_increment = 0.0001

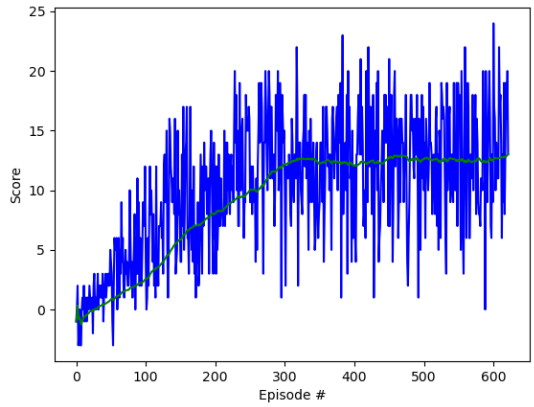
Epsilon

Epsilon is a hyperparameter that is used in the epsilon-greedy policy, a policy used to choose the next action. Where a random value is greater than epsilon, a random action is chosen, else the action that has the largest action value function is chosen. However, and depending on the value of epsilon, always following this means there is either no exploration of the environment or no exploitation of the agent's knowledge. To combat this a decay rate is introduced, so epsilon starts at the value 1 at the beginning of training and decays over time to a minimum value such as 0.01. This forces more environment exploration at the beginning of training and exploitation towards the end.

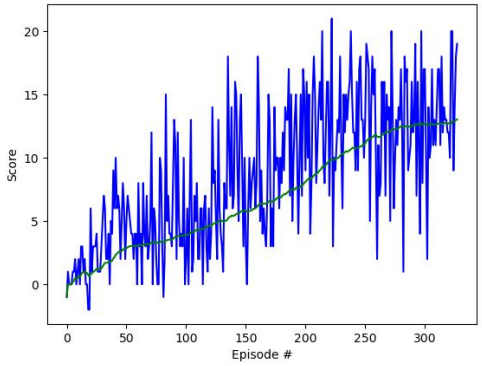
Setting epsilon decay rate to 0.995 solves the environment in 803 episodes.



Setting the epsilon decay rate to 0.975 solves the environment in 622 episodes.



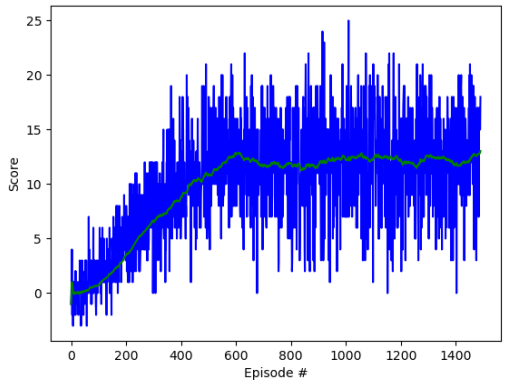
Setting the epsilon decay rate to 0.955 solves the environment in 328 steps.



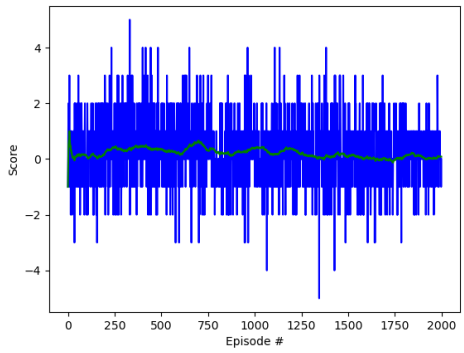
Gamma

The discount factor *gamma* is a hyperparameter that affects how far into the future the agent looks when training and is set to be between 0 and 1. The DQN learning algorithm calculates the temporal difference target using the just received reward and a discounted return from the next state. The discount value therefore influences how much of the expected future return the agent considers. A high value, nearer 1 means the agent looks to the future while a value of 0 means the agent is only concerned about the immediate reward.

A *gamma* value of 1 solves the environment in 1491 episodes.



A *gamma* value of 0.1 did not solve the environment within 2000 episodes.



Setting the discount to 0.1 leads to no improvement to the agent. With the reward being heavily discounted the agent only cares about the most immediate rewards. The agent is not looking far into the future, so it is not looking to maximise reward over many steps. It is interested in learning how to get the maximum reward at the next step, so when training the agent does not look to take decisions that lead it to collect many yellow bananas. This is reflected in the low average score.

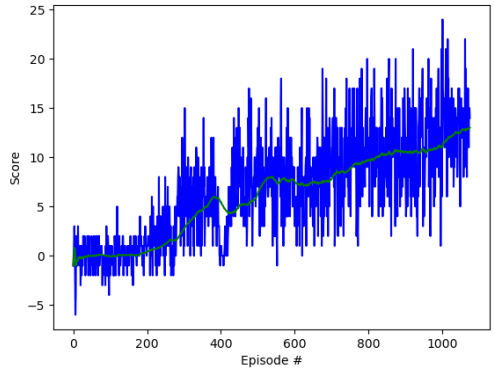
Setting the discount to 1.0 means the agent is looking as far ahead as possible when wanting to maximise the reward. The agent achieves a score of +13 over 100 episodes to solve the problem. It means that future banana collection counts as much as immediate collection and assumes other bananas will be collected and will be worth as much as current ones. It could lead the agent to collect a blue banana if there are many yellow bananas behind it rather than navigate around the blue banana.

Setting gamma is problem specific and in this case future bananas are worth as much and do not disappear during the episode. If this was not the case and bananas disappeared or their value decreased, then investigating a lower gamma value would be useful.

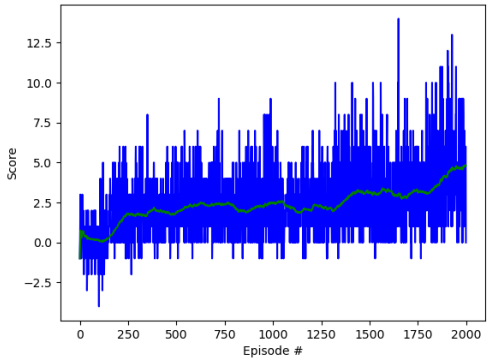
Tau

Tau is a hyperparameter that sets how far away the q target network is from the q local network. To use fixed Q-targets in Deep Q-Learning, a separate network, the q target network, is used in training, which is similar, but not identical to the q local network. This allows the q local network to update its weights using an error that it did not calculate using itself, rather the q target network was used to calculate part of the temporal difference error. Every learning step q target network must be updated to keep its weights near those of the q local network and the *tau* value can be seen as a lag of how far behind the target network is to the local network. It is a value between 0 and 1 where 0 would mean the target network never updates and 1 means it is a copy of the local network.

A *tau* value of 0.9 solves the environment in 1075 episodes.



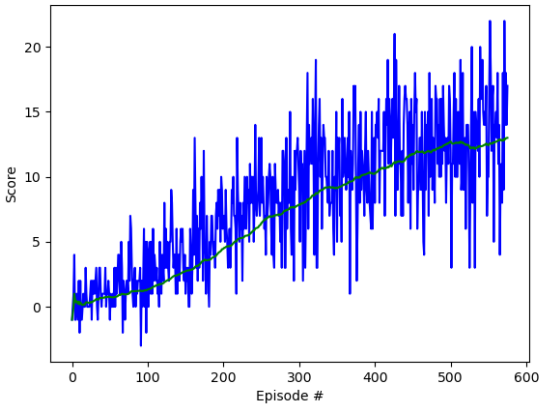
A tau value of 0.00001 did not solve the environment after 2000 episodes.



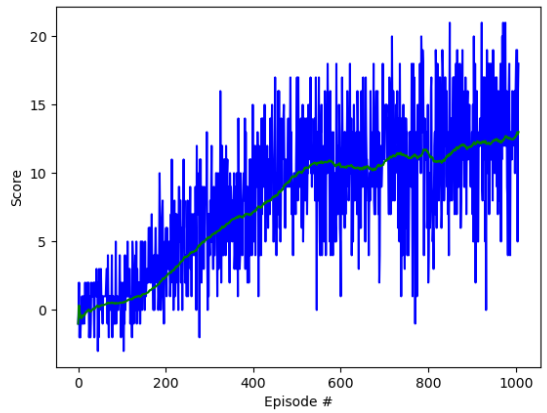
Prioritised Experience Replay – a

The *a* hyperparameter in Prioritised Experience Replay affects how the priorities of experience is converted into a sampling probability. A value of 1 sets the probabilities to their normalised priorities, which means when sampling from the buffer those experiences with the highest priority (and therefore probability) will be chosen a lot more often. Setting the *a* value to 0 equalises all sampling probabilities across experiences, so priority is not taken into account when sampling from the buffer.

An *a* value of 0.1 solves the environment in 576 episodes.



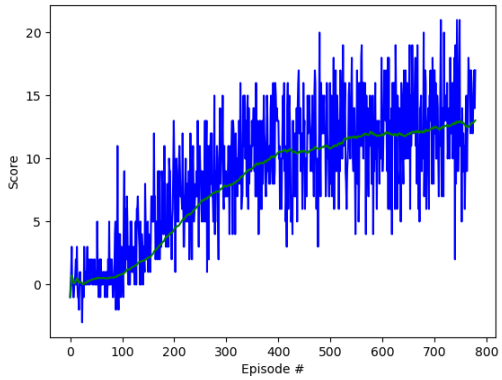
An *a* value of 0.9 solves the environment in 1007 episodes.



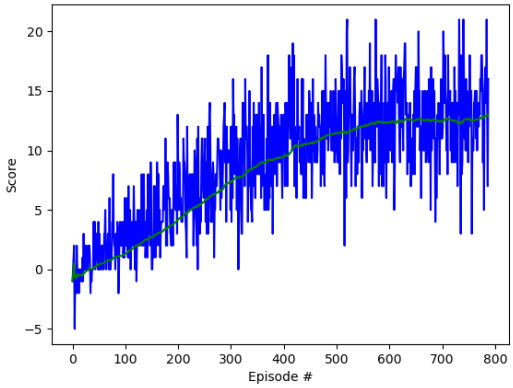
Prioritised Experience Replay – b

The *b* hyperparameter is a control of the importance sampling weights. These importance sampling weights are required to counteract the fact that certain experiences are sampled more frequently, and the weights reduce the effect these samples have on training. The *b* hyperparameter is annealed from its start value, around 0.4, to 1 during training. This increases the correction of the weights, meaning their value is reduced further over time and therefore higher priority weights have less of an effect at the end of training. The *b_increment* value, how quickly b is annealed, is another hyperparameter.

A *b* value of 0.4 with a *b_increment* of 0.001 solved the environment in 779 episodes.



A *b* value of 0.4 with a *b_increment* of 0.00001 solved the environment in 788 episodes.



The results can be summarised as follows:

Hyperparameter	Value	Episodes to solve
Epsilon decay	0.995	803
Epsilon decay	0.975	622
Epsilon decay	0.955	328
Gamma	1.0	1491
Gamma	0.1	Did not solve
Tau	0.9	1075
Tau	0.00001	Did not solve
A	0.1	576
A	0.9	1007
B_increment	0.001	779
B_increment	0.00001	788

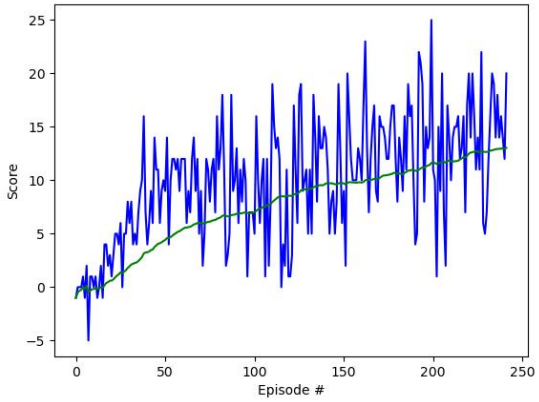
Conclusion

From the preliminary exploration work into how hyperparameters can affect training performance it is shown that the values have quite an effect on how quickly the agent trains. While taking recommended values as a starting point when implementing an algorithm, time should be taken to explore the effect of the hyperparameters and the best choice for the problem at hand.

The epsilon decay rate seemed to have the most effect on training speed, meaning the exploitation phase of the policy came into effect a little earlier than with other training runs. The Prioritised Experience Replay had less of an effect on the number of episodes to solve the environment. This could be that the Banana Collector environment does not have rare and important experiences that need to be used repeatedly in training.

Tailoring the hyperparameters to the following resulted in an agent that trained in 242 episodes:

- Epsilon decay = 0.955
- Gamma = 0.99
- Tau = 0.001
- A = 0
- B = 0
- B_increment = 0



Future Work

Other techniques such as Dueling DQN, Noisy DQN and Distributional DQN could all be used to further improve the agent's performance. All these techniques and more have been implemented in Deepmind's Rainbow algorithm which was shown to achieve what was then the state-of-the-art performance on the Atari 2600 benchmark [4]. Since then Deepmind has shown what is possible with AlphaGo which has since evolved into MuZero.

References

[1] Mnih, Volodymyr & Kavukcuoglu, Koray & Silver, David, et. al. Human-level control through deep-reinforcement learning, https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

[2] Van Hasselt, Hado & Guez, Arthur & Silver, David. Deep Reinforcement Learning with Double Q-learning https://arxiv.org/abs/1509.06461

[3] Schaul, Tom & Quan, John & Antonoglou, Ioannis & Silver, David. Prioritized Experience Replay https://arxiv.org/abs/1511.05952

[4] Hessel, Matteo & Modayil, Joseph & Van Hasselt, Hado, et. al. Rainbow: Combining Improvements in Deep Reinforcement Learning https://arxiv.org/abs/1710.02298