# Solving Unity Machine Learning Agents Banana Collector Environment Using a Deep Q Network

Udacity - Project 1 Report: Navigation

Ryan Herchig

## 1 Introduction

This project focuses on using temporal-difference learning to solve the Unity Banana Collector environment, similar but not identical to that found in [1]. The objective is for the agent to learn to collect as many yellow bananas as possible while avoiding blue ones. A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. In order to solve the environment, the agent must get an average score of +13 over 100 consecutive episodes. As a slightly more stringent requirement, for this project the environment was considered solved once a score of +15 was maintained for 200 episodes.

The continuous state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around the agent's forward direction. The action space is discrete with 4 possible actions: move forward, move backward, turn left, and turn right. For environments with continuous state spaces and discrete action spaces, a deep Q-learning approach is suitable. While calculating the state-action value function using Q-learning, or SARSA-max, the maximum over the probabilities of the predicted actions must be calculated. This operation is simple to formulate for discrete action spaces represented by categorical distributions, but is ill-defined and expensive to calculate for continuous action spaces in which calculating the most probable action would itself result in an optimization problem.

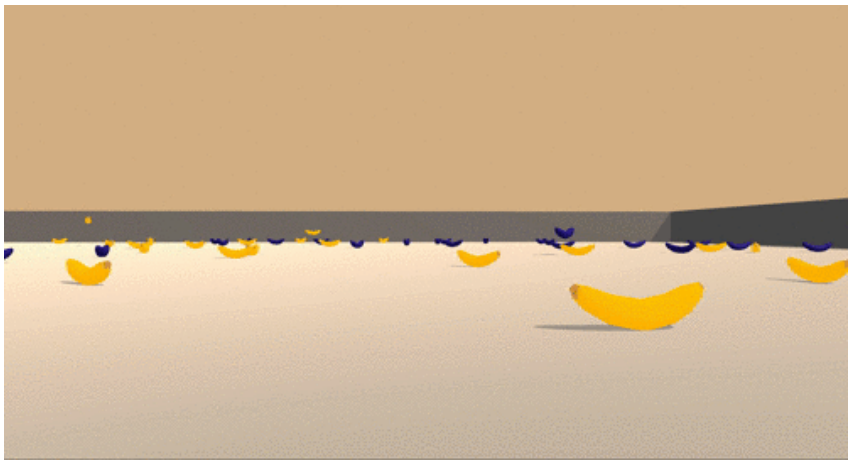

Figure 1: Image of the Unity Banana Collector environment [1].

## 2 Methodology

The deep Q-learning algorithm was originally proposed by [2] to allow a computer to learn to play Atari games with human-level ability. The state-action value function is used to learn a good policy, with the goal of learning an optimal policy. The state-action value function for current state $S_t$ and current action $A_t$ is given by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{1}$$

where $R_{t+1}$ is the reward at time step $t+1$ and $\gamma$ is the discount factor which controls the degree to which the agent values future rewards. A value of 0.99 was used for $\gamma$. However, when a function approximator (like a neural network) is used to estimate the state-action values, the update rule becomes

$$\theta_t \leftarrow arg \min_\theta \mathcal{L} \left[ Q(S_t, A_t; \theta), R_t + \gamma Q(S_{t+1}, A_{t+1}; \theta) \right] \tag{2}$$

where $\theta$ represents the parameters of the function approximator. The parameters are updated using gradient descent with a learning rate $\alpha$ of $5\text{x}10^{-4}$. To help stabilize the learning, a Q-target network was used in addition to the local Q-network. In the original Deep Q-networks paper [2], the authors perform a hard update on the parameters such that every 10000 steps, the parameters from the local Q-network are copied to the target network. In this project, a soft update to the target network was used instead, identical to the approach followed in [3].

The target network soft update equation is given by

$$\theta_{target} = \tau \theta_{local} + (1 - \tau) \theta_{target} \tag{3}$$

where $\tau$ was set to 0.001. Equation 3 shows that the parameters of the local network are slowly blended into the target network at each step. At each step,the new weights are composed of 99.9 % of the target network weights and 0.1 % of the local network weights. This helps ensure that at each learning step the TD target and the prediction are independent as they do not depend on the same neural network parameters.

In order to make the learning algorithm more sample efficient and to stabilize learning, an experience replay buffer was used as in [2]. The experience replay buffer stores a large number of samples which batches can be randomly drawn from for learning. This removes the temporal correlation which is present due to the causal nature of the simulation data. A random batch from the experience replay buffer provides samples from individual time steps, allowing the agent to learn the correct action given its current state without keying in on temporal information. Specifically, the environment was run for batch size number of steps before sampling from the replay buffer for learning. This was done to ensure that enough samples were available for learning. The replay buffer had a maximum capacity of 10,000 samples, and was implemented using a doubly ended queue appended only from the right meaning that the older samples were discarded to make room for newer ones.

The deep Q-learning model architecture is shown in fig. 2. The arrows connected to the experience replay buffer show that it receives input from the environment which it stores as (state, action, reward) tuples. The arrow pointing out of the replay buffer indicates that it can be queried for random batches of these tuples for training the networks. The two blocks labeled "Agent" represent the local (prediction) and target networks. The prediction network uses the current state to predict the best action and the expected Q values. The current state reward and next states are used to compute the Q targets for the current states. The expected Q values and the Q targets are used to compute the mean squared error loss, which is subsequently used to perform gradient

descent on the local (prediction) network parameters only. The optimal action computed from the prediction network is what is used to take the next step in the environment. At each step, a soft update of the target networks parameters is done, indicated by the "Update $\theta$'".

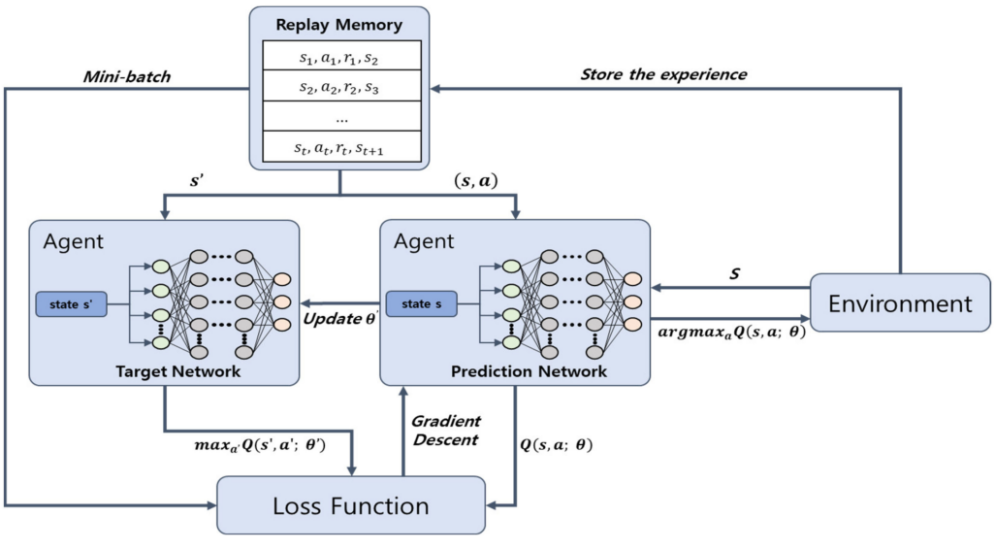


Figure 2: Schematic diagram of the Deep Q-learning with experience replay architecture [4].

The neural networks used for the local and target networks had 3 layers total. The input size for layer 1 was equal to the state size of 37 with 64 neurons in that layer. The hidden layer had 64 neurons as well and was connected to an output layer with 64 input neurons and 4 output neurons corresponding to the 4 possible actions. Rectified linear unit activation functions were used after layers 1 and 2. The Q networks were implemented using the PyTorch deep learning API [5]. The Q-network was trained using the Adam Optimizer [6] with the default PyTorch settings.

## 3 Results

The scores for the deep Q-learning agent are shown in fig. 3. Thought the raw scores are noisy (blue solid lines), the running mean (orange solid lines) which is taken over the previous 20 episodes is much smoother. This helps show the general trend of the scores as a function of episode. The black dashed line denotes the episode at which the environment was considered solved, specifically a score of +15 for 200 episodes. This occurred at episode 1000. The orange dashed line represents the mean score (15.56) taken over the final 1000 episodes. The variance of the mean was 0.050, indicating that the agent maintained, on average, a score close to the mean value with few outliers. Overall the agent learned at a steady rate with a small plateau from about 250 to 300 episodes.

After convergence, around episode 1000, the algorithm was allowed to continue to run for another 1000 episodes. This was done to ensure the stability of the learned policy. The policy learned by the DQN algorithm proves to be relatively stable upon convergence, maintaining a relatively constant average score until the remainder of the code execution. While the returns are noisy, the average score holds in the "solved" region. One technique which could be applied to smooth the noisy returns is outlined in [7], where they used a weight averaging technique to greatly reduce the random fluctuations in the agent's scores.

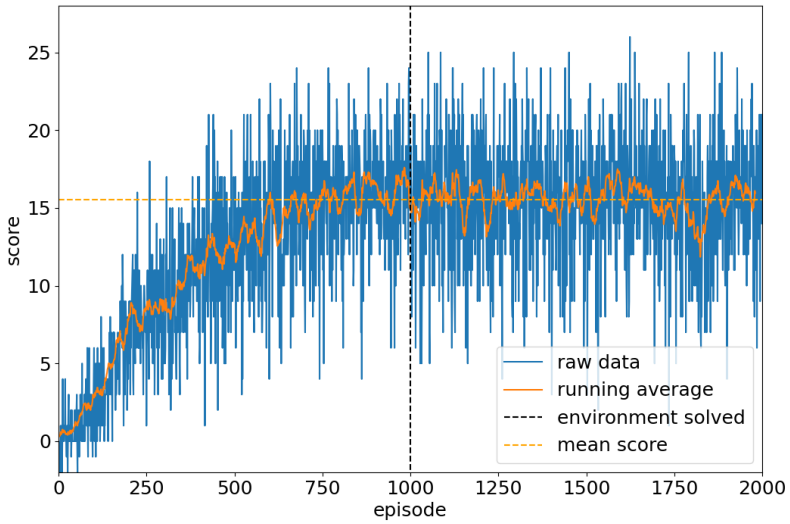


Figure 3: Scores for run zero of the DQN agent.

# 4 Conclusion & Future Work

For environments with discrete action spaces, temporal-difference learning methods like deep Q-learning, are typically able to learn the state-action value function, which could then be used to derive a good policy. For the deep Q-learning model, the policy would be to always take the best action instead of occasionally exploring by taking a random action, once the agent is trained. The Q-learning agent trained in this project performed well for the Unity Banana Collector environment, learning the state-action value function well enough to achieve and maintain a score of +15 for more than 1000 episodes. Several modification which could be made in the future to improve various aspects of the performance of the agent are as follows. The experience replay buffer could be modified to be a prioritized experience replay buffer [8]. The prioritized experience replay buffer replaces the random sampling with a sampling technique that takes into account the temporal-difference error associated with each saved transition tuple. This allows transitions with higher expected learning potential to be sampled more frequently, causing the agent to learn more quickly than it would from a random selection of experienced transitions. This would certainly speed up the convergence of the learning algorithm, and could potentially result in a larger average final score.

Another variation on the original deep Q-network model is the double deep Q-network (DDQN) or Double Q-learning [9]. The DDQN model was designed to address the problem of overestimation of the value of a state present in deep Q-networks that use 1-step bootstrapping to perform their estimates. The method requires minimal changes to the deep Q-network architecture. The general idea of Double Q-learning is to reduce overestimations by decomposing the max operation in the target into action selection and action evaluation. To achieve this, the target network is used to estimate the value of the state while the local online network is used to select the next action. Though not entirely decoupled, the parameters of the two networks are independent enough that the value estimations calculated from the target network are significantly more accurate than when simply using deep Q-learning. This modification results in the agent learning policies that, on average, provide higher returns. As was mentioned previously, another possible improvement to the model would be to use weight averaging to stabilize returns [7]. This would lead to an overall more stable policy and more consistent

performance from the agent once trained.

## References


[1] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A general platform for intelligent agents," *arXiv preprint arXiv:1809.02627*, 2020.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.

[3] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2015.

[4] D.-J. Shin and J.-J. Kim, "Deep reinforcement learning-based network routing technology for data recovery in exa-scale cloud distributed clustering systems," *Applied Sciences*, vol. 11, no. 18, 2021.

[5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[6] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.

[7] E. Nikishin, P. Izmailov, B. Athiwaratkun, D. Podoprikhin, T. Garipov, P. Shvechikov, D. P. Vetrov, and A. G. Wilson, "Improving stability in deep reinforcement learning with weight averaging," 2018.

[8] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2015.

[9] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 2015.