# Cooperative Multi-agent Reinforcement Learning for Flappy Bird*

Corbin Rosset†, Caroline Cevallos†, Ian Mukherjee†

***Abstract*— The advent of Google's Deep Q-Learning Network ushered in a new generation of reinforcement learning systems that learn control policies directly from raw sensory data. Two important innovations– using a deep convolutional neural network to approximate the action-value function, and stateful "experience-replay" mechanisms–allowed agents to tractably learn successful strategies on Atari games, like Pong. Researchers are working to generalize these techniques to handle the complex challenges faced by human agents in the real world. To this end, we are the first to investigate multi-agent reinforcement learning to elicit cooperative play in the game Flappy Bird. In our experiments, each agent is independent and only observes the actions of the other player from the raw environment pixels. We report that two agents are able to achieve a score over 1000 after being trained on over 20,000 rounds of play.**

## I. INTRODUCTION

Reinforcement learning is a category of machine learning that instructs an agent to act rationally in complex environments using only rewards/punishments to guide its action in place of supervision. The environment is comprised of possibly infinite states $s_t \in \mathcal{S}$ in each of which an agent receives a reward $r_t = R(s_t) \in \mathcal{R}$ and performs an action $a_t$ from a set of actions $\mathcal{A}$ in order to reach another next state with hopefully higher reward. There is also a notion of a transition model, which specifies how an action changes one state to the next - this can be defined stochastically (where actions can be determined as unreliable), or it can be determined by physics[1]. The important assumption is that the state transitions adhere to the markov property, that is, the next state should only depend on the previous state and the action.

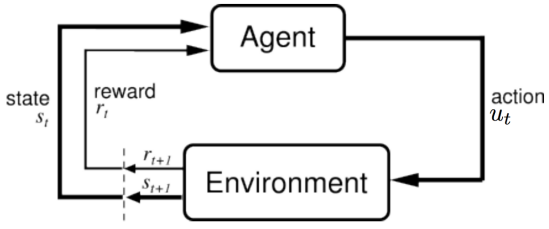


Fig. 1. Feedback system

One solution to a reinforcement learning problem is to learn a policy $\pi(s)$ that specifies the best action to take in any state (as the argmax of probability distribution over the possible actions at that state). It is understood that the best policy $\pi^*(s)$ yields the highest utility in the next state. The utility $U(s_t)$ is the expected sum of discounted future rewards over an infinite horizon of decision making; hence utility is a metric of the long-term quality of a policy [2]. The utility of each state is its own reward plus the expected reward of its successor states, so utilities of states are not independent.

$$\pi^*(s_t) = \arg\max_{a \in \mathcal{A}(s_t)} \sum_{s_{t+1}} P(s_{t+1}|s_t, a)U(s_{t+1}) \tag{1}$$

where

$$U^\pi(s_t) = \mathbb{E} \sum_{t'=t}^{\infty} (\gamma^{t'-t} * r_{t'})^3 \tag{2}$$

The expectation is taken with respect to all possible gameplay state sequences that can arise from the current state-action pair. There are two problems to calculating $\pi^*$. The first is that $U$ takes a sum over infinitely many items. This is addressed by the Bellman equation, which allows for an iterative approach to finding the optimal policy:

$$U(s_t) = R(s_t) + \gamma \max_{a \in \mathcal{A}(s_t)} \sum_{s'} P(s'|s_t, a)U(s') \tag{3}$$

It can be proven that the solution set of the Bellman equations over all states is unique and equal to the utilities defined in Equation 2. However, there are as many Bellman Equations as there are states, and they are nonlinear because of the "max" operator[4], so we must iterate the following update until convergence in an algorithm known as Value Iteration:

$$U(s_{t+1}) = \mathbb{E}\big[r_t + \gamma \max_{a' \in \mathcal{A}} U_t(s_t)|s_t, a_t\big] \tag{4}$$

Secondly, Equation 1, calls for knowledge of the transition model $p(s_{t+1}|s_t, a_t)$. If this is known, then this scenario is known as a Markov Decision Process and algorithms such as Value Iteration and Policy Iteration are guaranteed to converge to optimal policies.

Without knowing either the transition model or the reward function – "model free" – agents can still infer the

*Final Project for EN.600.661 Computer Vision at Johns Hopkins University. Instructor: Dr. Austin Reiter

†Johns Hopkins University Dept. of Computer Science

[1]If the transition model is fully known, then most reinforcement learning tasks reduce to path planning through the state space

[2]discounting refers to how much the agent prefers short term rewards to long term ones, and is a hyper-parameter $\gamma$ usually set to 0.99

[3]In practice, the summation terminates when the game terminates

[4]They become linear if the policy is fixed, but the policy is exactly what we are trying to learn

best policy by trial and error. In most challenges relevant to the real world, this is the only practical approach for the following reasons: humans cannot consistently specify rewards and transitions for all possible state-action pairs, the state-action space might be too large to even compute rewards or transitions for, and the exact outcome of actions cannot be determined in complex environments. Yet, humans can provide reliable feedback on a number of obvious events such as scoring, winning, crashing, dying, etc.

### A. Q-learning

There is a fundamental question at hand: to learn both a model for the environment (its transitions) and utility function as previously defined, or merely an action-utility function with no model. The answer seems to be that for less structured and more complex enviornments, learning an action-value function is favored. Instrumental for the latter approach is a strategy known as "temporal difference" learning. It disregards the transition probabilities altogether, but instead uses the observed $s, a, s'$ transitions to adjust the utilities of the observed states to match the utility estimates of their $observed\ successor$ states. With an appropriately decreasing learning rate $\alpha$, the utility estimates converge toward equilibrium values that agree with the Bellman constraints from Equation 4:

$$U^{\pi}(s) \leftarrow U^{\pi}(s) + \alpha\big(R(s) + \gamma U^{\pi}(s') - U^{\pi}(s)\big) \qquad (5)$$

Inside the factor with $\alpha$ as a coefficient, one can see how Equation 4 is used as a target value to adjust $U(s)$ for any fixed policy $\pi$.

A Q-learning agent seeks a similar action-utility function $Q(s, a)$ which gives the expected utility of action $a$ in state $s$ using one-step lookahead, where $U(s) = \max_a Q(s, a)$. A Q-learning agent does not need a model of the environment or transitions at any time, hence it is "model free". At equilibrium, Equation 3 still holds if $U(s_t)$ is replaced with $Q(s_t, a_t)$. But using the temporal difference approach in Equation 5, the Q-learning update equation becomes:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\big(R(s) + \gamma Q(s', a') - Q(s, a)\big) \quad (6)$$

which is calculated whenever an agent takes action $a$ in state $s$ leading to $s'$. Conveniently, Equation 5 gives rise to a natural loss function for the Q-function: the term with $\alpha$ as a coefficient is zero if the Q-function is a perfect predictor of the discounted utility of the next state. These updates also lead to an off-policy algorithm, since the value of the optimal action is learned independently of what action was actually taken.[5] The optimal Q-function is analogous to the optimal utility function from Equation 4, since the Bellman equation still applies:

$$\mathcal{Q}^*(s, a) = \mathbb{E}_{s'\tilde{\mathcal{E}}}\big[R(s) + \gamma \max_{a'} Q^*(s', a')|s, a\big] \qquad (7)$$

[5]As opposed to an on-policy algorithm, which learns the values only of the actions being taken

In active reinforcement learning, an agent must decide what actions to take. For model-free agents to understand the consequences of their actions, they must explore the state-action space by randomly choosing actions without regard to one it believes maximizes the Q-value. After training, and when the user believes the agent has gained significant understanding of the environment, then the agent can exploit a greedy policy to maximize its reward. Empirically, it is best to gradually transition from a curious exploratory agent to a greedy one over many iterations. This is known as the exploration-exploitation trade-off, and is manifested in an $\epsilon$-greedy strategy that follows the greedy strategy with probability $1 - \epsilon$ as $\epsilon$ decays over training iterations.

There is one glaring limitation however: learning the Q-function requires observing every state-action pair many times over for convergence. This is not possible even for toy problems, so the Q-function must be approximated so that it is no longer a lookup table. Often times a linear regression model is used as the function approximator, often with hand-crafted domain-specific features. Compressing a model into a finite number of parameters also generalizes performance to scenarios the agent hasn't encountered before. Neural networks have proven to be good universal function approximators and in that they represent a large space of functions, the optimal of which can be found expeditiously. While Q-functions represented as linear models are guaranteed to converge to the closest possible apprximation to the true Q-function, nonlinear functions enjoy no such theoretical guarantees and often diverge wildly even in simple environments. They have nonetheless been a key innovation for advancing reinforcement learning methods, leading to the rise of Deep Q-learning.

### B. The PyGame and Flappybird Environment

In our case, actions are reliable and the transition model is determined by simple projectile physics, but the agent does not have access to this information since the goal is to learn from only the raw pixel inputs. The state is the raw pixels of four sequential frames of game-play. If the reader is not familiar with Flappybird, a demo can be found here: `http://flappybird.io/`

## II. METHODS

### A. Deep Q-learning

Any neural network employed as a Q-function approximator is known as a Q-network [2]. Much work has been devoted to alleviating the instability of nonlinear Q-function approximators. It turns out that two of the largest impediments were actually quite obvious: many nonlinear function approximators including feed-forward neural networks treat examples as being 1) independent and 2) identically distributed across a stationary distribution. However, game-play is inherently time series data in which sequences of states of often high correlated, hence, the action-value $Q(s, a)$ and the target values $(r + \gamma \max_{a'} Q(s', a'))$ are correlated. As the agent learns, the distribution of states it observes is non-stationary as it chooses "smarter" actions; empirically, small

perturbations to the Q-function can significantly change the policy [4].

Instead of updating the Q-functions using only one temporal difference update as in Equation 6, using an "experience replay" mechanism to smooth the training distribution by sampling a subset of size $K < D$ of the most recent $e_t = (s_t, a_t, r_t, s_{t+1})$ experiences or transitions. A queue of the most recent $D$ transitions is kept. For experience replay, Bellman updates are applied sequentially to a subset or minibatch of the stored experiences chosen uniformly at random from the queue. Mnih et al. 2015 give a discussion of the benefits of experience replay. The network loss function for a minibatch at iteration $i$ is the mean-squared error in the Bellman equation with the optimal target values being the approximate ones:

$$L_i(\theta_i) = \mathbb{E}\big[\big(R(s) + \gamma_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\big)^2\big] \quad (8)$$

where the expectation is taken with respect to (s, a, r, s') experience tuples and reduces to arithmetic average over the $K$ experiences. Secondly, the network parameters are only updated every $C$ steps, so those used in the computation at iteration $i$ are $\theta_i^-$, and they are only updated to $\theta_i$ when $i$ $mod$ $C$ is zero. This is contrary to supervised machine learning settings involving neural networks, since the network's loss functions depends on the network's parameters from a previous iteration. Yet this is correct, since the Bellman equation seeks to adjust the estimate of the Q-value prior to executing action $a$ to that received in the state $s'$ following $a$.

$$\nabla\theta_i L(\theta_i) = \mathbb{E}\big[\big(R(s) + \gamma \max_{a'} Q(s', a'; \theta_i^-) \quad (9)$$

$$- Q(s, a; \theta_i)\big)\nabla_{\theta_i} Q(s, a)\big] \quad (10)$$

Stochastic Gradient Descent is used rather than computing the full expectation.

Another facet of learning the Q-function from raw sensory input is that the state becomes only partially observable, giving rise to a Partially Observed Markov Decision Process (POMDP), which has less theoretical guarantees, but in this case does not affect performance [5].

### B. Convolution Neural Network Architecture

Convolutional Neural Networks are adept at extracting low-dimensional, semantically meaningful feature vectors from images using relatively few parameters.

The architecture of the deep Q-network suited to raw image inputs was a three-layer convolutional neural network followed by two fully connected layers. The final layers involve reshaping the convolved layers and apply ReLU, an element-wise activation function written as follows:

$$f(x) = \min(0, x) \quad (11)$$

This activation function approximates the softplus function with a hardmax, which induces sparsity in the hidden layers of the convolutional neural network. The output of the CNN represents the action-value function, so sparsity is encouraged, and it is preferred that the best action has the highest value, while the other actions are near zero.

### C. Extension to Multi-player Game-play

The game code was modified to add a second player sprite independently controlled from the first. The game ends if either bird dies and scoring is incremented only when both birds successfully passing through a pipe. For simplicity the first player is always red and the second player is blue so as to not confuse the networks (even though images are gray-scaled).

### D. Cooperative Play and Rewards

In order for the system to score, both agents have to clear a pipe. We feared that if both agents are rewarded equally, the system would be vulnerable to getting stuck in an local optimum where the first agent always clears the pipe and the second agent would never be forced to play expertly. To preempt this, we established the following rewards scheme:

| Rewards | Agent 2 Clears | Agent 2 in Air | Agent 2 Crash |
|---|---|---|---|
| Agent 1 Clears | (0.5, 2) | (0.5, 0.1) | (0.5, -1) |
| Agent 1 in Air | (0.1, 2) | ( 0.1, 0.1) | ( 0.1, -1) |
| Agent 1 Crash | (-1, 2) | (-1, 0.1) | (-1, -1) |

The table shows the rewards received by each agent under each possible scenario. Agent 1 always leads Agent 2 in the world, and the tuple (X, Y) means that Agent 1 received X as a reward, and Agent 2 Y.

### E. Training Procedure

We trained our single- and multi-agent neural networks with the same hyperparameters as in [1]. The training schedule was comprised of three stages: the "observe" stage, which lasted one hundred thousand time steps; the "explore" stage, which lasted two million time steps; and the "train" stage, which continued until the program was terminated. In practice, the observe stage was adequate for training.

The discount rate, $\gamma$, was set to .99 to prevent overfitting the agent from becoming short-sighted; $\epsilon$, which determined how often an agent plays a random action, was set to 0.2 initially, and decayed linearly to 0.0001 over the "explore" phase. The experience replay queue was constrained to hold fifty thousand state-action-state-reward transitions.

Training lasted two and a half days for the single-agent neural network lasted, and over five days for the multi-agent network.

## III. Results

We have published Youtube videos of single agent[6] and two-agent flappy bird[7] trained with our methodology. The two-agent flappy bird took more than twice as long to train, but this was on a cloud-based virtual machine with 8GB Ram, 4 cores, and no GPU.

[6]single agent: `https://youtu.be/sZHn_10Mo_k`
[7]two-agent: `https://youtu.be/y9ZQDuxpK44`

### A. Single Player Results

The maximum score we saw after two and a half days of training was 527. By analyzing the trends in Figures 2 & 3, it is simple to see that more training could yield higher and more consistent scores.

| **Stage** | Games Played | $Q_{Max}$ | Game Score |
|---|---|---|---|
| Observe | 500 | 0.008 | 0 |
| Explore | 13,000 | 10.8 | 40 |
| Train | 14,000 | 12 | 209 |

TABLE I

SCORES TAKEN FROM EACH TRAINING STATE DURING SINGLE-AGENT TRAINING.

Table II illustrates increasing confidence and game score as the number of games played increases. Each game played is indicative of one or more time steps. The agent became exponentially more confident when transitioning from the observation state to the exploration state, achieving a Q-value of about 10.8. It is trivial to see that–if time allowed for more training–the game score could surpass the level of an experienced Flappy Bird player. This trend is best represented in our multi-agent results.

### B. Multi-player Results

The maximum score we saw after six days of training was 1093, but it could achieve higher and more consistent scores with more training.

Figure 2 shows the score achieved by the system at the end of each new game it played, as well as the 100-moving average. Figure 3 shows the maximum Q-value for any action the agent considered at any time step in game-play; this is a measure of how confident the agent was in the best action. Because the rewards scheme in Table 1 rewards the second agent more for surviving, the second agent's confidence outpaces the first agent as the system learns to keep both agents alive.

By the end of training, the $Q_{max}$ for player 1 was 11.4123, and that for player 2 was 15.6436 as Figure 3 shows.

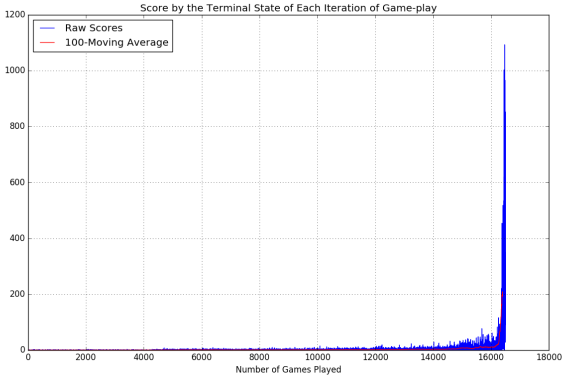


Fig. 2. The score achieved by the agent by the end of each game-play session.

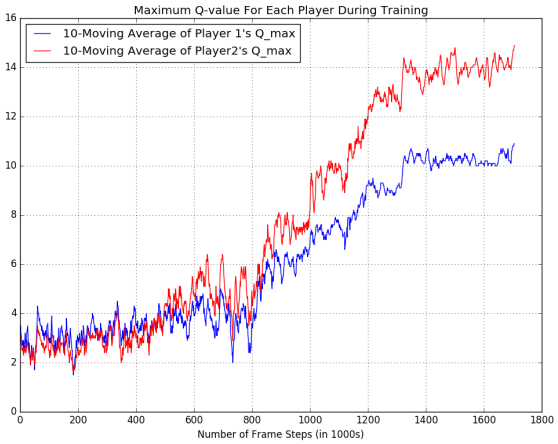


Fig. 3. The maximum Q-value observed by each agent at every frame-step (each game is comprised of a sequence of frame-steps).

## APPENDIX

Our code is published at `https://github.com/iankm/FlappyDQL-MultiAgent`. The master branch is for single player Flappy Bird, the FlaPyBirdz branch contains code for multiplayer. The Deep Q agents are implemented as classes for modularity and scability.

## REFERENCES


[1] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).

[2] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529-533.

[3] Tampuu, Ardi, et al. "Multiagent cooperation and competition with deep reinforcement learning." arXiv preprint arXiv:1511.08779 (2015).

[4] Russell, Stuart Jonathan, et al. Artificial intelligence: a modern approach. Vol. 2. Upper Saddle River: Prentice hall, 2003.

[5] Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. Vol. 1. No. 1. Cambridge: MIT press, 1998.