# Fast Learning on Jelly Bean World: COMP 579 Final Project Report

**Dragos Cristian Manta**
Université de Montréal, Mila
cristian-dragos.manta@mila.quebec
Group 011

**Konstantinos Christopher Tsiolis**
McGill University, Mila
kc.tsiolis@mail.mcgill.ca
Group 011

## Abstract

This work aims to create fast-learning and high-performing agents on Jelly Bean World (JBW) by exploring five different reinforcement learning algorithms with different function approximators and by leveraging different signals from the observation space, which is composed of scent, RGB images and custom features provided by the TA's.

## 1 Introduction

JBW [7] is a nonstationary infinite two-dimensional grid world which is used to assess the continual learning capabilities of agents. In this work, we tackle a simpler stationary version of JBW which is populated by four items. Apples give a reward of +1, bananas give +0.1, truffles give -0.1, and jelly beans -1. The goal of the agent is to navigate this environment so as to maximize its total reward during an episode of 5000 steps.

We approached this problem using a variety of standard reinforcement learning methods, which fall into two categories: value-based methods and policy-based methods. We detail these approaches in Section 2. We evaluate the performance (total reward per episode) and sample efficiency of agents using these methods.

## 2 Methods

Prior to discussing the approaches we use, we discuss some unifying threads in our methodology. First, given that the JBW grid is infinite, we cannot use tabular RL methods and must resort to function approximation in order to generalize across states. Though we also experiment with a linear function approximator, the convolutional neural network (CNN) is our function approximator of choice due to its inductive bias that is well-suited for working with two-dimensional grid and image-structured data. Second, all methods have the same set of inputs available to them which can be used to construct a state representation: scent input, visual input and TA-provided features.

### 2.1 Value-based Methods

In this problem, we are ultimately interested in learning a policy $\pi(\cdot|s)$, which gives a probability distribution over actions (up, down, left, right) for each state $s$ in the state space $\mathcal{S}$. In value-based methods, we achieve this indirectly by parametrizing a *value function* with our function approximator. In this work, we focus on the action value function [9]

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\middle| S_t = s, A_t = a\right]. \tag{1}$$

At evaluation time, we take our final policy to be $\varepsilon$-greedy with respect to the learned $Q$ function. Though these approaches have the disadvantage that the policy is not learned directly, they learn a predictor of the expected return that results from taking an action $a$ in a state $s$ and following a policy $\pi$. This leads to greater interpretability, as we can track the agent's learning via the evolution of the $Q$ values the model produces (see Section 4). Furthermore, value-based methods have had tremendous success in game environments with two-dimensional visual input, with the most notable example being the Deep Q Networks of [5].

**Sarsa.** Sarsa is the most natural on-policy value-based method when working with action values. Given a transition $(s, a, r, s')$ where $a$ is sampled from $\pi$ ($\varepsilon$-greedy with respect to $Q$ in our case), $Q(s, a; \theta)$ (where $\theta$ denotes the model parameters) is updated online using the sample-based semi-gradient Bellman update

$$\theta \leftarrow \theta + \alpha(R + \gamma Q(s', a'; \theta) - Q(s, a; \theta))\nabla_\theta Q(s, a; \theta), \tag{2}$$

where $a' \sim \pi$. This is a well-motivated algorithm due to its theoretical foundation in the Bellman equation and guarantee of convergence to a region near the true action value function (albeit only with linear function approximators) [9].

**Q-Learning.** The Q-learning update is based on the Bellman optimality equation and also takes a semi-gradient form, as follows:

$$\theta \leftarrow \theta + \alpha\big(R + \gamma \max_{a'} Q(s', a'; \theta) - Q(s, a; \theta)\big)\nabla_\theta Q(s, a; \theta). \tag{3}$$

Thus, while $a$ is sampled from the behaviour policy that is generating the transitions used for training (in our setup, $a$ is sampled $\varepsilon$-greedy with respect to $Q$), $a'$ is sampled from the greedy policy with respect to $Q$, thus making $Q$-learning off-policy. This method also comes with theoretical convergence guarantees, but only in the tabular case.

**DQN.** Introduced in [4] and [5], Deep Q Networks (DQN) improve upon Q-learning via the addition of a replay buffer and a target network. The former allows for the learner to be exposed to the same transitions multiple times during training and to break correlations between successive inputs (so as to be more akin to the i.i.d. setting of supervised learning). The target network provides greater stability to the algorithm which can easily succumb to the "deadly triad" of function approximation, bootstrapping, and off-policy learning [9] by delaying updates to the value function in the Q-learning target. DQN achieves superhuman performance on a suite of Atari games using only visual input [5]. Given that JBW is a two-dimensional grid environment with similar actions to those in an Atari game (up, down, left, right), we hypothesize that DQN can achieve high performance on our task. Furthermore, we hypothesize that the use of a replay buffer will aid with sample efficiency.

**DDQN.** Proposed in [10], double Q-learning makes use of two DQNs which take turns playing the role of the learner and the target. This is found to have the effect of reducing maximization bias that results from the $\max$ term in the Q-learning target. In their results, [10] show that DDQN outperforms DQN on the Atari suite.

### 2.2 Policy-based Methods

We also implemented a policy gradient method, A2C, to complement our suite of value-based methods. This has the advantage of directly learning a policy that seeks to maximize the expected reward per step.

**A2C.** Advantage Actor Critic, a special online case of the more general A3C [3], learns a state value function (the critic) to aid the learning of a policy (the actor). First, transitions are collected for $T$ steps. Then, for each step $t \in \{0, \dots, T-1\}$, we consider the $(T-t)$-step return $\hat{R}_t = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{T-t+1} r_T + \gamma^{T-t} V(s_t)$ and the *advantage* $\hat{A}_t = \hat{R}_t - V(s_t)$. We emphasize the use of multi-step returns and thus the lower reliance on bootstrapping, which serves as additional motivation for experimenting with this method. The loss as formulated in [8] is

$$\mathcal{L} = \sum_{t=1}^{T} \Big[ -\log\big(\pi_\theta(a_t|s_t)\big)\hat{A}_t + \hat{A}_t^2 \Big]. \tag{4}$$

For gradient descent, we take the gradient of the first term with respect to the policy parameters (giving us the usual policy gradient with the estimated value function as a baseline) and the gradient of the second term with respect to the value function parameters (giving us a multi-step TD update).

## 3 Experiments

An invite to our complete repository for experiment code[1] was sent to the TA GitHub account. The implementation of all methods relies on the PyTorch library [6]. All agents are trained for 500,000 timesteps and evaluated every 5000 timesteps from the same starting state. Optimization is done using the Adam optimizer [1]. Performance is reported at the best timestep, where the average is taken across three seeds[2]. Sample efficiency is measured as the area under the curve (AUC) over the first 100,000 timesteps only (with evaluation at every 10,000 timesteps), where the horizontal axis is rescaled to the interval $[0, 10]$. Taking inspiration from [4], for DQN and DDQN we also select a random subset of states by following a random policy before training and report the average $Q$ value of the greedy action at every 1000 steps in order to track the evolution of $Q$.

For each of the methods listed in Section 2, we performed an informal hyperparameter search and a few ablation experiments (for example: removing the replay buffer from DQN) and chose the best settings based on best performance. We omit the details of this search due to space considerations and instead only mention the final hyperparameter configuration for each method.

We experimented with a linear function approximator and a CNN for Sarsa and Q-learning. For the linear function approximator, input features are flattened into a 900-dimensional vector. Since we found CNN to be far superior, we only experiment with the CNN for the other methods. Our CNN architecture is inspired by the LeNet-5 architecture [2], which was designed for small-resolution images such as MNIST, because we are working with $15 \times 15 \times 4$ grid inputs. We also took inspiration from the CNN used in the DQN paper [5]. To process the TA-provided features, we use a convolutional layer with six $3 \times 3$ kernels and another with sixteen $3 \times 3$ kernels, with a $2 \times 2$ average pooling layer after each. Then, we have a fully-connected network with a hidden layer of size 32. For the value-based methods, the network produces four outputs (one for each action). For A2C, we have two heads, one which produces four outputs (the policy head) and one which produces one output (the state value head).

For Sarsa and vanilla Q-learning, we take $\alpha = 0.001$ and $\varepsilon = 0.1$ for the behaviour policy as well as for the policy at final evaluation. For DQN and DDQN, we take $\alpha = 0.00025$ and start with $\varepsilon = 1$ and linearly anneal to $\varepsilon = 0.1$ over the first 10,000 timesteps, in similar spirit to [5]. This allows for a gradual decrease in exploration as learning progresses. At final evaluation, we take $\varepsilon = 0.05$. We set the capacity of the replay buffer to 1000 and use a batch size of 16. We also update the target network at every 1000 iterations. For A2C, we take $\alpha = 0.001$ and $T = 5$.

Regarding the signals from the observation space that we used, we tried incorporating the scent with DQN and DDQN by concatenating the output of the penultimate layer with the scent input. Although this did improve the performance in general, the hyperparameters that we described above (and which produce the best performance and sample efficiency) yield better results without the scent. Therefore, we omit it for the rest of our experiments. Similarly, we found superior results with the provided features rather than with the visual inputs.

## 4 Results and Discussion

Experiments with Sarsa and Q-learning justify our choice of the CNN over a simpler linear function approximator. In Table 2, we observe that the CNN more than doubles the performance and sample efficiency of the linear model, which further validates the power of its nonlinearity and inductive bias.

[1] https://github.com/CristianManta/COMP579-project

[2] They are: 579, 911 and 2022.

Table 1, our main results table, shows the performance and sample efficiency scores of each algorithm, measured according to the previous section. One problem that we observed to be common to the DQN and DDQN algorithms is the instability during training, as shown in Figures 1 and 2 (left), where the mean accumulated reward in evaluation mode can vary wildly at different stages in training. Sometimes, we even observed a decrease from 700 to 20 after one episode. On the other hand, the training looks much more stable from the perspective of the learned Q-values, as seen in Figures 1 and 2 (right). This pattern is also consistent across all hyperparameters and ablation experiments that we performed. The steady increase in Q values means that the agents become more and more optimistic and always find an action with higher expected return on average (across states), which can be interpreted as "finding more and more promising moves in the average situation (state)" as they evolve.

On the other hand, A2C experiences far less fluctuations in the accumulated reward on its way to a higher performance and sample efficiency than all other methods (see Figure 3. This suggests that on this particular task, greater benefits can be reaped from learning the policy directly and making use of multi-step returns than learning action values and making use of a replay buffer.

Furthermore, in comparing DQN and DDQN to the other three algorithms, we remark that the use of a replay buffer seems to have little impact on both performance and sample efficiency. In fact, DQN is the worst-performing algorithm and DDQN's sample efficiency is lower than A2C. We conjecture that this is due to the relative simplicity of the problem: the input space is small compared to Atari games (which have $84 \times 84 \times 4$ images) [5] and there are only five possible rewards for each action (the reward for each of the four items along with zero).

| **Algorithm** | **Performance** | **Sample Efficiency** |
|---|---|---|
| Sarsa | 672.20 | 1033.72 |
| Q-Learning | 674.33 | 2011.18 |
| DQN | 665.83 | 1269.45 |
| DDQN | 698.67 | 2234.50 |
| A2C | 766.13 | 3046.77 |

Table 1: Performance metrics obtained with the best hyperparameter settings for all of the methods used. Note that the performance may not correspond to the leaderboard evaluation performance since we likely don't use the same seeds. Moreover, our sample efficiency metrics are not on the same scale as the leaderboard, so these numbers are only meant to rank the above methods.

## 5 Conclusion and Statement of Contributions

We train and evaluate five reinforcement learning methods: Sarsa, Q-learning, DQN, DDQN, and A2C on a simplified version of the Jelly Bean environment. We find that A2C, a policy gradient method, achieves the highest performance and sample efficiency.

One of the things that we did not explore a lot is the choice of the optimizer. We used the Adam optimizer for all of our algorithms because it was harder to tune SGD. Moreover, [4] and [5] use RMSProp, which could perhaps stabilize our training. Another consideration which could help our sample efficiency further is adding prioritized experience replay. Overall though, we were quite successful in beating the baseline agent presented in the leaderboard by a large margin.

KC wrote the introduction and methods sections of this report and Cristian proofread them. Both KC and Cristian contributed equally to the experiments, results and discussion, and conclusion sections of this report as well as to the creation of the video. Finally, both authors share equal credit for the code implementations.

## References

[1] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[2] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[3] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[6] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[7] E. A. Platanios, A. Saparov, and T. Mitchell. Jelly bean world: A testbed for never-ending learning. *arXiv preprint arXiv:2002.06306*, 2020.

[8] D. Precup. Actor-critic methods. *COMP 579: Reinforcement Learning*, 2022.

[9] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[10] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

# A Appendix

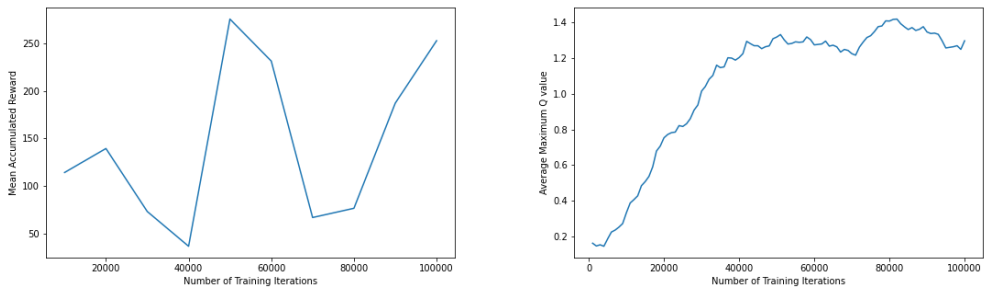


Figure 1: DQN sample efficiency averaged over three seeds (left) and evolution of greedy state-action values (right).

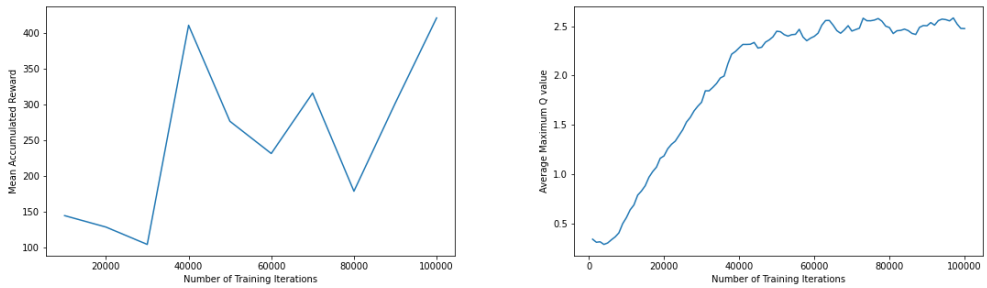


Figure 2: DDQN sample efficiency averaged over three seeds (left) and evolution of greedy state-action values (right). Note that, since the action choices are made according to $Q_1(S, A) + Q_2(S, A)$ for DDQN, we displayed the sum on the y-axis when reporting the greedy Q values, so it is expected that the values are higher in this plot than in the one for DQN.

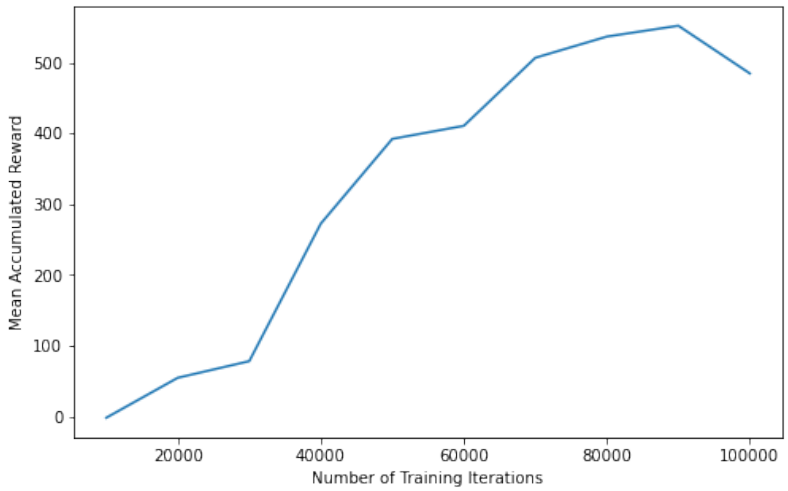


Figure 3: A2C sample efficiency averaged over 3 seeds.

| Algorithm | Performance | Sample Efficiency |
|---|---|---|
| Linear Sarsa | 168.13 | 401.23 |
| CNN Sarsa | 672.20 | 1033.72 |
| Linear Q-Learning | 203.13 | 488.65 |
| CNN Q-Learning | 674.33 | 2011.18 |

Table 2: Comparison between the linear function approximator and CNN for the Sarsa and Q-Learning methods.