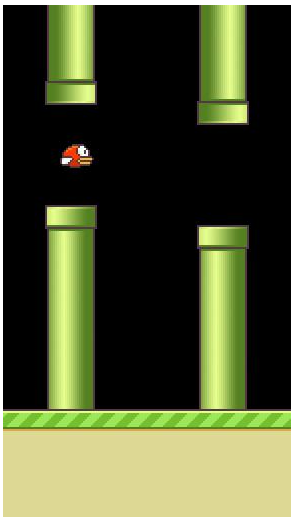
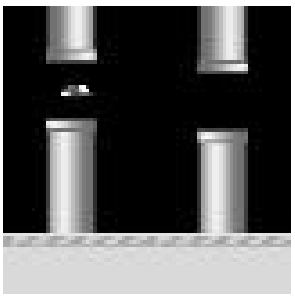
1 Introduction

Reinforcement learning is a trending subfield within AI nowadays. Typically, in reinforcement learning we model an agent to function in a given environment under some assumptions of the notion of reward and loss.





Typically game playing is used to explore and compare new techniques and algorithms in reinforcement learning. For this project, we try to train an agent to perform well on a popular android game, called Flappy Bird. Put simply, the game involves an agent (the bird) that has to cross obstacles without crashing.

2 Setup

For this project, we modify the pre-existing code available at [1], which demonstrates DQN on Keras and uses a pygame implementation of Flappy Bird. Our code is available at [2] (see branches other than master).

For preprocessing, we

- convert a colored image to grayscale
- resize to 80×80
- rescale intensities, so the brightest pixel is white and darkest is black

4 consecutive frames are stacked to create a shape of $(4, 80, 80)$. This is the input to our convolutional network.

For training the network, squared loss is used as the loss function and Adam optimizer with a learning rate of 10^{-4} is used. For the training, γ (discount parameter) is set to 0.99. For methods that use ϵ greedy, ϵ is set to anneal from 0.2 to 0.0001 over the period of training. For bootstrap methods, number of heads K is set to 10. Also, the replay buffer can hold 50000 transitions, and the training is done in mini-batches of size 32.

3 Methods

In Q learning, we try to learn a good approximation of the Q^* function, that tells us the expected discounted reward from the given state $s \in \mathbb{S}$, taking an action $a \in \mathbb{A}$ at the current time step. Q^* follows the Bellman equation:

$$Q^*(s, a) = \mathbb{E}[r_t + \gamma \max_{a'} Q^*(s', a')]$$

3.1 DQN

To make Q learning feasible, [3] tries to approximate Q^* by creating a network that just takes in s, and as output has $|\mathbb{A}|$ nodes, each of which gives an approximation of $Q^*(s, a)$ for $a \in \mathbb{A}$. They also use experience replay [4] with holdout weights. Backpropagation is done on

$$\text{loss} = \text{target} - \text{prediction} = (r + \max_{a'} Q^*(s', a'; \theta_-)) - Q^*(s, a; \theta)$$

3.2 Double DQN

Q learning is likely to choose overoptimistic values over time [5]. Double Q Learning separates the **selection** of action from the **evaluation** by using two sets of weights. Double DQN [6] uses current weights for selection and holdout weights for evaluation.

$$\text{target} = r + Q^*(s', \arg\max_{a'} Q^*(s', a'; \theta); \theta_-)$$

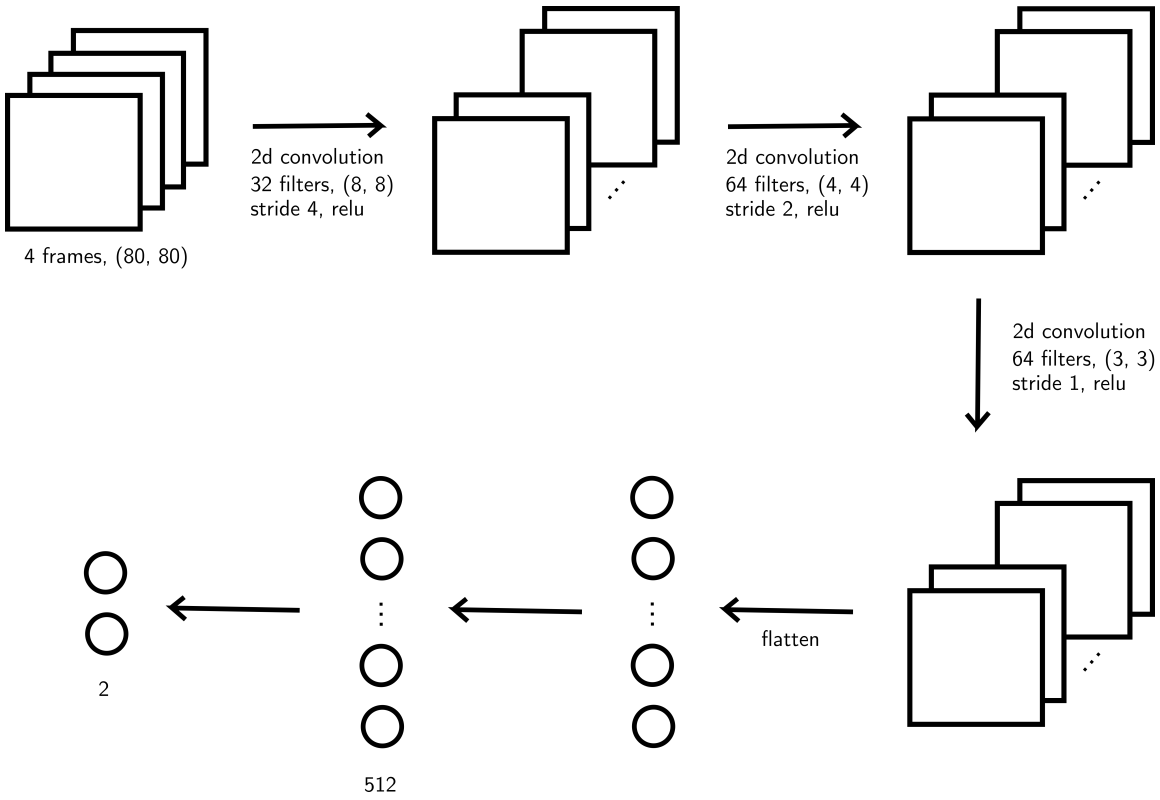


Figure: Network Architecture

3.3 DQN with UCB1

UCB1 [8], where UCB1 stands for upper confidence bound, is an OFU (optimism in face of uncertainty) approach. It deals with the exploration/exploitation dilemma by assigning a bonus term for actions that are not explored enough. Thus, instead of maximizing the action value function we maximize the following:

$$a_t = \arg\max_{a'} \left\{ Q(s, a') + \sqrt{\frac{2 \log t}{N_t(a')}} \right\}$$

3.4 Async RL

Instead of experience replay, in asynchronous RL [9] we execute multiple agents in parallel, on multiple instances of the environment. Multiple actors-learners running in parallel explore different parts of the environment. The overall changes being made to the parameters by multiple actor-learners applying online updates in parallel are likely to be less correlated in time than a single agent applying online updates. Hence the asynchronous method does not rely on experience replay, and the training time is reduced roughly proportional to the number of parallel actor-learners. Moreover, this method can also be used by on-policy algorithm, but it is out of the scope of this project. For this project, we just tried asynchronous one-step Q-learning.

3.5 Bootstrapped DQN

Bootstrapped DQN [7] works by having K neural networks (heads), which start with different random weights. Each of them learns on a bootstrapped sample of the experience buffer. At every training iteration, one head is chosen at random, and its decision is used to train some of the heads, based on the bootstrap mask. This leads to efficient exploration.

3.6 Bootstrapped DQN with UCB1

We modify the algorithm for Bootstrapped DQN [7] by changing the update rule to incorporate UCB1. This leads to faster convergence, as our experiments show.

4 Experiments and Results

Taking the existing implementation for flappy-bird DQN [1], we implemented multiple exploration and reward based methods such as double DQN, bootstrapped DQN, UCB1, bootstrapped DQN with UCB1 and asynchronous RL and ran experiments

to quantitatively compare their performances. For each experiment, we trained the network for 1 million time steps, that takes around 25 hours on a single GTX 1080 GPU. For double DQN and asynchronous RL we faced implementation challenges, hence results these aren't tabulated. For instance, for asynchronous RL, since pygame does not support multiple screens in one process, the display screen has to be locked very frequently. It makes the program very slow. So we can only train the model for about 0.03M time steps every 3 hours for each thread.

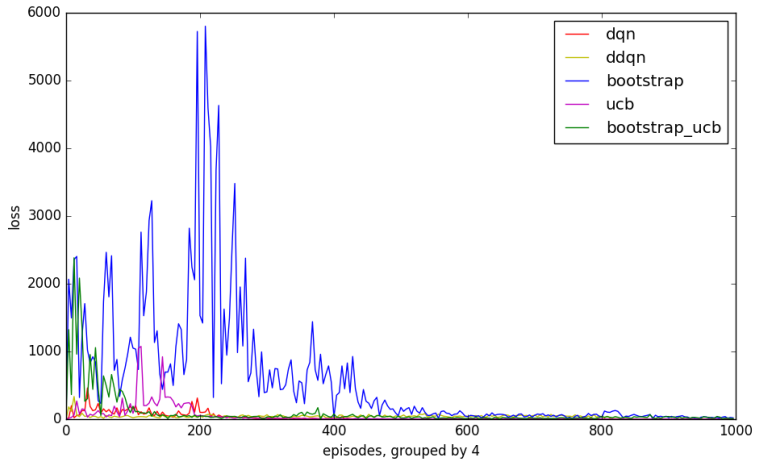


Figure 1: Plot of loss vs no. of training episodes (how fast they converge to 0?)

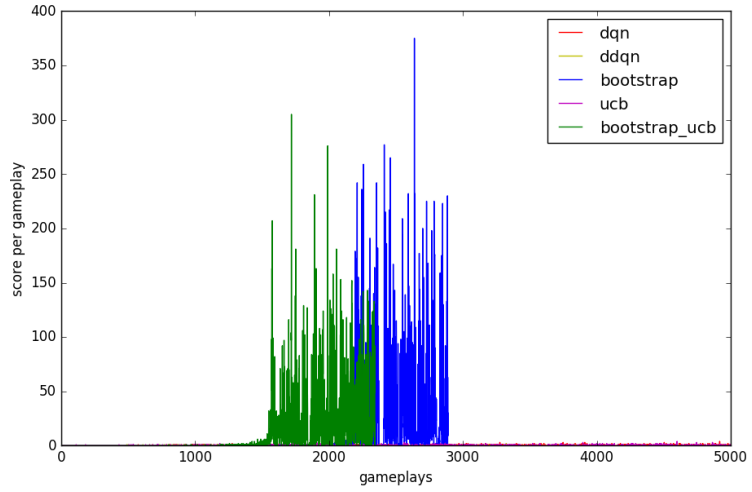


Figure 2: Plot of scores per gameplay in training

Figure 1 gives the plot of squared loss over time-steps for each experiment. Figure 2 gives the plot for scores over episodes during training. From the plots we can infer that bootstrapped DQN with UCB1 starts giving higher scores with fewer training time-steps or episodes narrowly followed by the pure bootstrapped DQN approach. Figure 3 gives plot for average scores over episodes, for a total of 50 episodes while testing. From Table 1, bootstrapped DQN with majority voting outperforms all other methods with average score over 50 episodes of 201 and max score of 740.

Specifically, for bootstrapped DQN (both vanilla and with UCB1), in test mode, we ran 2 different approaches in terms of choosing the action. In the first approach, out of the 10 heads (models), we randomly choose a head to get the best action. In the second approach, we decide the action based on the vote by all heads. We find from Figure 3 and Table 1 that voting approach scores the best. This follows from the point from [7] that at critical time steps, all the decision heads generally agree on the best decision.

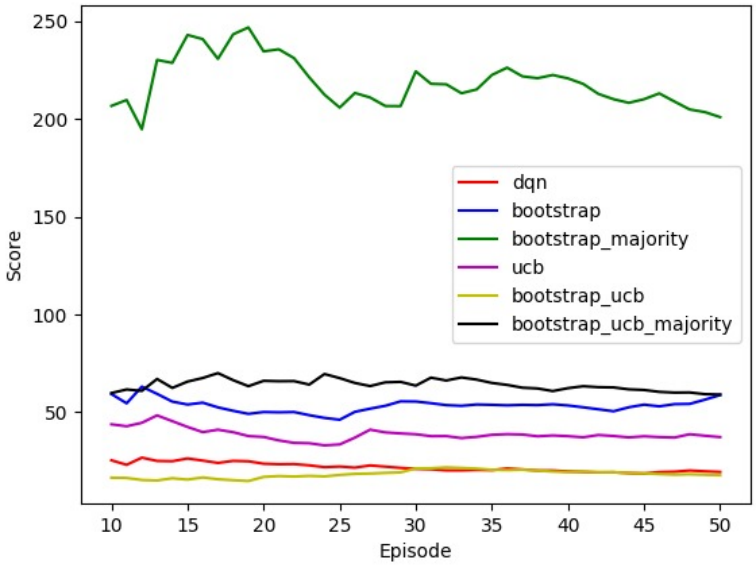


Figure 3: Plot of moving average scores for different RL approaches in test phase

Method	**Max score achieved**
DQN	67
bootstrap	243
bootstrap with majority voting	740
UCB1	147
bootstrap with UCB1	81
bootstrap with UCB1 and majority voting	197

Table 1: Max scores achieved for different methods

5 Conclusion

As expected, the experiments show that bootstrap with majority voting performs the best among all the techniques. An interesting observation is that, UCB1, even though improves the performance of DQN as far as average scores is concerned, doesn't go well with bootstrap, since the gameplay scores on average reduce if it is used. But, it also makes the game start playing good fairly quickly, even quicker than vanilla bootstrap. In conclusion, our experiments try to cover and corroborate most of the recent developments in the burgeoning field of RL.

References

[1] https://github.com/yanpanlau/Keras-FlappyBird

[2] https://github.com/djian2017/flappy-bird

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

[4] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, DTIC Document, 1993.

[5] S. Thrun and A. Schwartz. Issues in using function approximation for reinforcement learning, 1993.

[6] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

[7] Osband, Ian, et al. "Deep exploration via bootstrapped DQN." Advances in Neural Information Processing Systems. 2016.

[8] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multi-armed bandit problem. Machine Learning, 47:235256, 2002.

[9] Mnih, Volodymyr, et al. "Asynchronous methods for deep reinforcement learning." International Conference on Machine Learning. 2016.