Learning to play Flappy Bird Using Deep Q-Networks

Anup Shakya (ashakya@memphis.edu)

Abstract

Reinforcement learning (RL) is an emerging domain in machine learning where an agent learns to take actions in an environment in order to maximize the cumulative reward. In this project, we use a very popular algorithm in RL, called Deep Q-Networks and show that it is very effective at learning to play a game like Flappy bird despite having a high-dimensional sensory input. The agent is given no prior information about what the bird looks like and what the pipes do. The agent must learn these representations and use it to develop an optimal strategy to maximize the cumulative reward. We further show that the agent is able to achieve super-human performance and discuss on the problems and potential improvements with DQN.

INTRODUCTION

Reinforcement Learning is quickly becoming very popular in the machine learning community, mostly due to its generality. It is studied in multiple disciplines like game theory, control theory, operations research, multi-agent systems, statistics and many more. The craze around RL is also due to the fact that it does not need carefully crafted and labelled data. An agent just focuses on finding a balance between exploration and exploitation. The notion of exploration and exploitation is quite important in RL. Exploration helps the agent to try new actions in the environment, which is critical to find new knowledge. Exploitation helps the agent to act greedily by using the obtained knowledge to get a higher reward immediately.

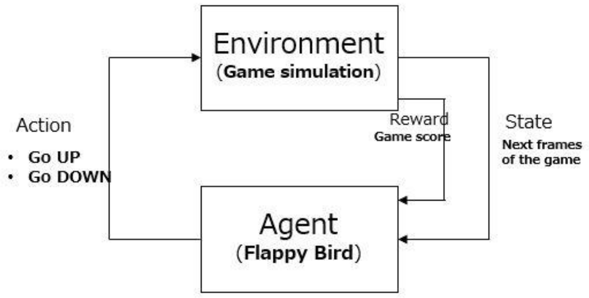
Reinforcement learning leverages the fact that there is no single correct way to complete a certain task. For example, we need to program a robot to travel from city A to city B. Clearly, there would be no single way to travel from city A to city B and it would be unrealistic to program every maneuver the robot must take to complete the task. Therefore, it must learn to take decisions under uncertainty with high dimensional input (like video output from camera) in order to complete the task. This project is a first step at realizing such implementation.

Flappy Bird is a game where the player needs to keep the bird alive by keeping it from hitting the pipe or the ground. The aim is to time the flaps so that the bird gets in between the pipes and stay alive for as long as possible. This game was originally launched as a mobile game and it was very popular but was removed from the Play Store in 2014 by the creator himself as he felt guilty that the game was too addictive.

Arcade Learning Environment (ALE) [1] was published in 2013 which proposed a general learning environment for AI. ALE provided an emulator for Atari 2600 games and proved to be a testbed for many reinforcement learning algorithms. Deep Q-Networks [2, 3] was a breakthrough algorithm that acted like “one algorithm to learn them all”. The implementation of DQN consists of a convolutional neural network which was fed frame pixels of the game and the agent learnt its behavior by just looking at these pixels. The network is trained with a variant of Q-learning algorithm, with stochastic gradient descent to update the weights. To solve the problem of correlated data and non-stationary distributions, DQN uses experience replay mechanism which randomly samples previous transitions, and thereby smoothens the training distribution over many past behaviors. DQN performed at super-human level for many of the Atari 2600 games.

MDP Formulation

As is the case for all reinforcement learning algorithms, we formulate the problem at hand as a Markov Decision Process (MDP). The MDP formulation for the flappy bird game is quite straight-forward. The MDP formulation is shown below:



Environment	→ The game world
Agent	→ The flappy bird
State	→ Multiple frames of the game (pixel values)
Action	→ Flap the wings (Go Up) Do nothing (Go Down)
Reward	→ is basically the game score +1 for passing between the vertical columns +0.1 for staying alive in each state -1 for crashing.
Goal	→ maximize the game score or reward.

The state here, is represented by the sequence of game frames. The state is defined as a tuple below:

<current state, action, reward, next state>

Deep Q-Networks

DQN is a state-of-the-art algorithm that was a breakthrough and a first successful implementation of deep neural networks in reinforcement learning. The most attractive thing about DQN was that it did not require any hand-crafted features and was generalizable over a wide range of applications. The original authors used the same network architecture and hyperparameter settings across 7 Atari 2600 games. And this performed well on all the games. This demonstrated the robustness of the algorithm and made it very famous.

Fig. 1 shows the network architecture implemented for this project. The original game frames have dimensions 288 x 512. A sequence of four consecutive game frames is picked and then converted to grayscale image. The gray-scaled image is then converted to image containing pixel value zeros and ones using thresholding. Then, the image size is converted to 80 x 80 in order to match the size of the input in the convolutional layer. The network consists of three convolutional layers, where each layer is followed by a max pooling layer. At the end of the convolutional layers are two fully connected dense layers with ReLu activation. The final output is given by a Dense

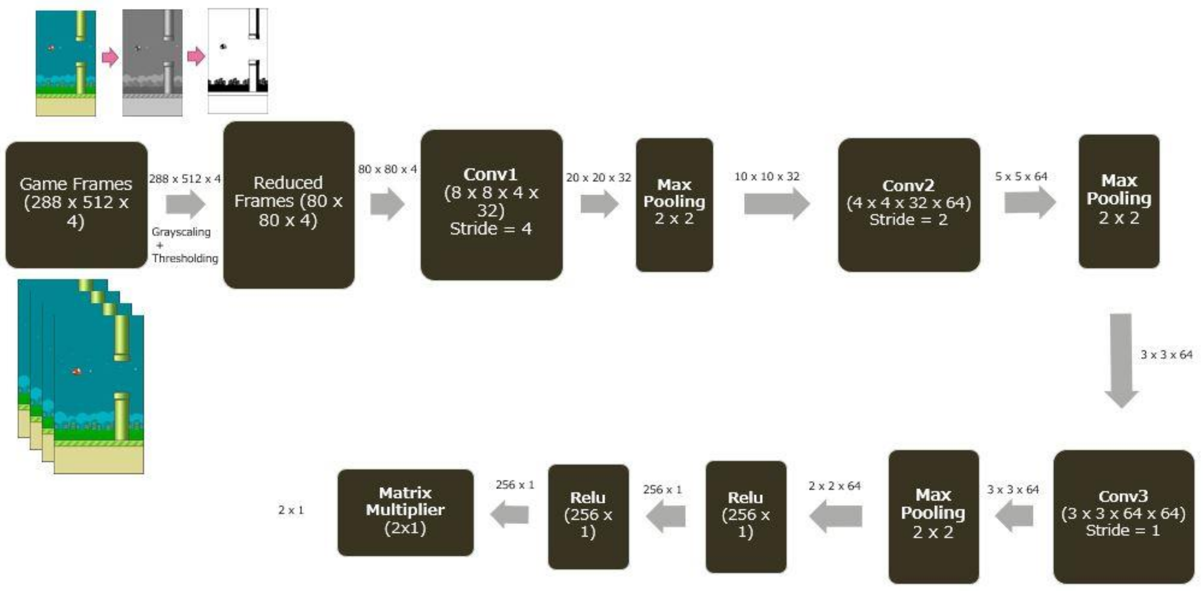


Fig. 1 Network Architecture for DQN implementation

layer with linear activation. The final output dimension is 2 x 1, which represents the action to take given the current state. The DQN algorithm is given below:

```
❑ Initialize Replay Memory D to size N
❑ Initialize action-value function Q with random values.
❑ For each episode 1 to M
        - Initialize state $s_1$
        - for t = 1 to T
              select random action at with probability $\epsilon$
              else select $a_t = \max_a Q(s_t, a; \Theta_i)$
              execute $a_t$ in emulator and observe $r_t$ and $s_{t+1}$
              Store $[s_t, a_t, r_t, s_{t+1}]$ in Replay Memory D
              Sample minibatch of transitions $[s_j, a_j, r_j, s_{j+1}]$ from D
              Set $y_j = \begin{cases} r_j & for\ terminal\ state\ s_{j+1} \\ r_j + \gamma * max_{a'} Q(s_{j+1}, a'; \Theta_i) & for\ non\ terminal\ state\ s_{j+1} \end{cases}$
        - Perform gradient step on $(y_j - Q(s_j, a_j; \Theta_i))^2$ with respect to $\Theta$
```

The training in DQN has three phases: observe, explore, and train. The training starts out with a high value of ϵ that permits proper exploration of the possible states in the game world. The first phase is ‘observe’, where the agent first collects states in the replay memory. The replay memory is, as the name suggests, the experience memory of the agent which it can use to know about the past actions and rewards. Once the experience replay memory is filled, the next phase is ‘explore’. In this phase, the ϵ value is gradually decreased and the agent takes a greedy action with probability (1- ϵ) using the replay memory to guide the greedy action. The expected behavior in this phase is that the replay memory eventually gets filled with more and more greedy actions that result in greater rewards. In a way, we can say that the algorithm generates its own labelled dataset to train the neural network. As the training progresses forward, the dataset becomes more and more refined. Once the ϵ value has dropped to a very small value, the final phase starts which is the 'train' phase. In this phase, the agent takes greedy actions with very high probability. This is the phase where the agent actually learns the optimal policy to play the game.

Despite being such a robust algorithm, DQN is not free of flaws. The most problematic thing about DQN is its convergence. We can take a look at the Bellman Equation below.

$$Q(s_t, a_t) = r_{t+1} + \gamma \max(Q(s_{t+1}))$$

Here, the Q(.) function depends on itself. When we are calculating the Q-value for the current state, it depends on the Q-value of the future states. However, in DQN, the Q-function is updated in every iteration. So, the target Q-function is just like a moving target, or we can say the algorithm would be chasing its own tail. This causes longer convergence time.

Double DQN

Double Deep-Q Networks [4] was introduced as an improvement to the vanilla DQN algorithm that solved the substantial overestimation problem. This specific adaptation to the DQN algorithm results in reduced overestimation and also much better performance on several games. The Double DQN uses a target network and fixes it for stabilizing the convergence. So, it has two Q networks: local and target. The local network parameters are updated every iteration, but the target network parameters are updated every C iterations. The target network is used to estimate the Q-value.

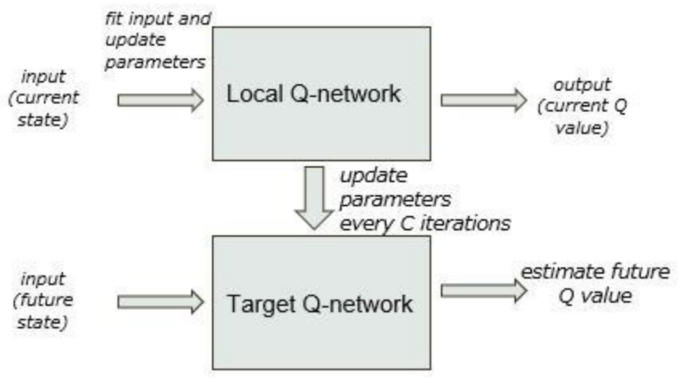


Fig. 2 Double DQN Architecture

As shown in Fig. 2, the local Q-network is trained and updated in every iteration. However, the target Q-network is kept fixed. After C iterations, the parameters of the local network is copied to the target network. Since, the target network is not moving around too much, it is much easier for the network to converge.

EXPERIMENTS

The experiments for this project were carried out in medium difficulty level of the game. For implementing the RL algorithms, I used the source code for the game flappy bird from GitHub [5]. The game consists of different background like the night scene and day scene. Also, the bird had three color variants. The change of background had prominent effect in the network and the change of bird's color had very little effect. So, I conducted three experiments. They are:

1. Black background with red bird using vanilla DQN
2. Night scene background with red bird using vanilla DQN
3. Night scene background with red bird using Double DQN

I used the same training hyper-parameters for all the experiments. The hyper-parameters are given below:

Discount Factor = 0.99
OBSERVE = 50,000
EXPLORE = 500,000
Initial ϵ = 0.2
Final ϵ = 0.001
Experience Memory Buffer = 50,000

RESULTS

The table below shows the training time for each of the experimental setup. One thing that was quite clearly observed was that the learning in DQN is not transferable from one setup to the other. The first experiment was performed with a black background using DQN algorithm. The final trained network was then fed with input frames from the game with a night scene background. The network failed to transfer anything that it learned from the previous experiment and I had to train the network all over from scratch. The second experiment with night scene background took a long time (more than 2 times) to train. This can be explained by the fact that with the colored and textured background there was additional complexities. And the network had to figure out which part of the image is the bird and which part is the background.

Experiment	**Total Training Time**
Black background with DQN	1.5 days
Night scene background with DQN	4 days
Night scene background with Double DQN	2 days

The problems associated with DQN was quite clear in the second experiment, where the network overestimated. As a work around to this problem, the third experimental setup involves the same night scene with Double DQN algorithm. With Double DQN at scene, the training improved a lot. We can see from the table above that it was able to train the network in half the time. Furthermore, with the Double DQN approach, the agent seemed to have a better control of the game which could be seen in the way the bird was maneuvered.

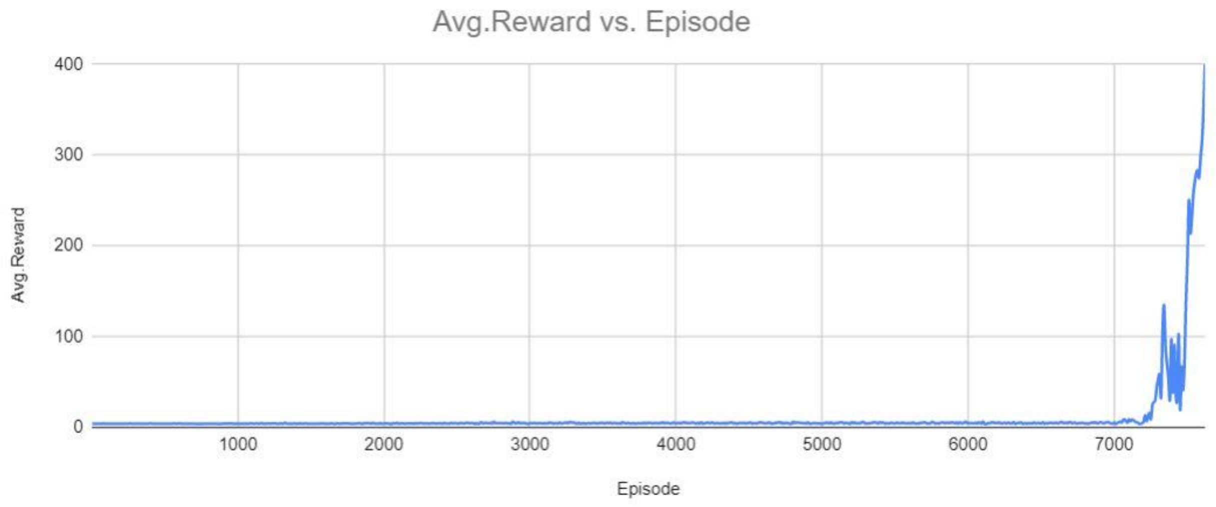


Fig. 3 Plot showing average reward earned during different phases of training for Double DQN. Episodes 1 to 1000 is the observe phase, episodes 1000 to 7000 is the explore phase, and 7000 to 8000 is the training phase.

The plot in Fig. 3 shows the average reward earned by the agent during the training process of the third experiment (Double DQN). Each episode here refers to a full game played by the agent until the bird dies. As we can see, the training starts with a very low average reward. The initial episodes till 7000th episode is where the agent is collecting proper data in its replay memory. After the 7000th iteration, the training phase starts where the agent starts taking more greedy actions. The average reward drastically increases and gets stuck in local minima for some time. After more training, in about 8000th iteration, the average reward reaches its maximum value. At this point, the bird just does not die, and the game goes on and on.

It is also to be noted that the staying alive reward was vital for the agent to converge much faster. This reward encouraged the agent to take some action rather than doing nothing very early on in the training process (observe phase). Without this reward, the network often got stuck in loops where the bird just did nothing and crashed into the ground.

CONCLUSIONS

Reinforcement learning is a really exciting domain in machine learning and its attractiveness is ever increasing as more and more researchers are starting to study it. With DQN as a breakthrough algorithm, and gameplay (ALE) as a testbed for newer algorithms, many algorithms have been developed that is a testament to the immense promise that Reinforcement Learning has. An RL agent mimics the way a human learner learns i.e., by the notion of reward, exploration, and exploitation. This project is my first step into the realm of reinforcement learning. And this project helped me explore the recent developments in RL in greater depth. I hope to continue following, studying, and working on this in the future.

References

1. Bellemare, M. G., Naddaf, Y., Veness, J. & Bowling, M. *The arcade learning environment: An evaluation platform for general agents*. J. Artif. Intell. Res. 47, 253–279 (2013).
2. Mnih, V. et al. *Human-level control through deep reinforcement learning*. Nature 518, 529–533 (2015).
3. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. *Playing Atari with deep reinforcement learning*. NIPS ’13 Workshop on Deep Learning, 2013.
4. Van Hasselt H, Guez A, Silver D. *Deep reinforcement learning with double q-learning*. In Proceedings of the AAAI Conference on Artificial Intelligence 2016 Mar 2 (Vol. 30, No.1), 2016.
5. Sourabh Verma, FlapPyBird, Github:*https://github.com/sourabhv/FlapPyBird*