FlappyBird

Fargo: Team 5

April 2019

Introduction

Learning to play games has been one among of the popular topics researched in AI today. Solving such problems using game theory/ search algorithms require careful domain specific feature definitions, making them averse to scalability. The goal here is to develop a more general framework to learn game specific features and solve the problem. The game we are considering for this project is the popular mobile game - Flappy Bird. It involves navigating a bird through a bunch of obstacles. Though, this problem can be solved using naive RL implementation, it requires good feature definitions to set up the problem. Our goal is to develop a CNN model to learn features from just snapshots of the game and train the agent to take the right actions at each game instance.

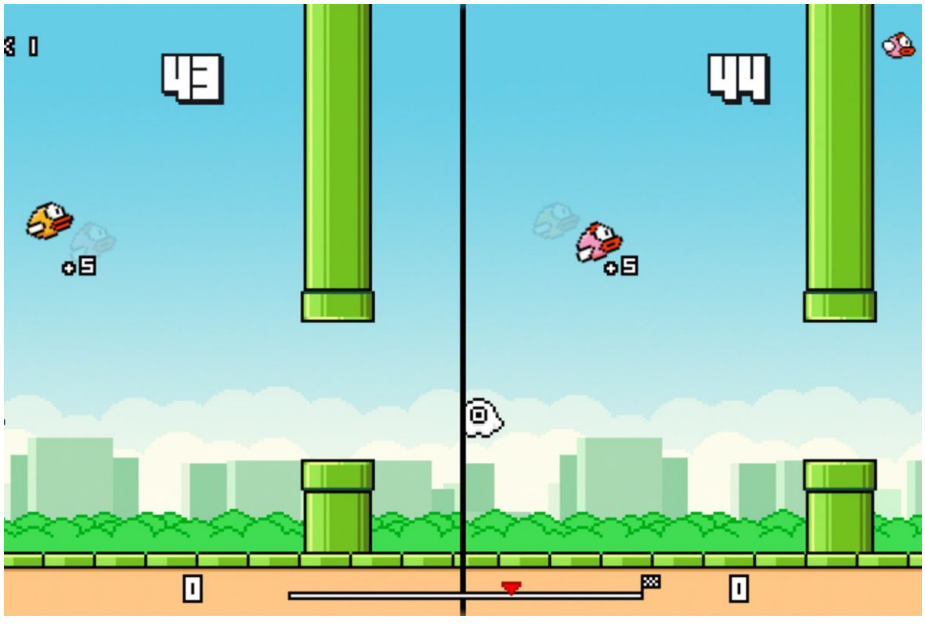


Figure 1: Flappy Bird Game

Goal

This project is an implementation of the research paper Deep Reinforcement Learning for Flappy Bird by Kevin Chen. The goal of the project is to learn a policy to have an agent successfully play the game Flappy Bird. This will be done by replicating the paper results through demonstrable code.

Problem Description

Training an agent to successfully play the game is especially challenging because our goal is to provide the agent with only pixel information and the score. The agent is not provided with information regarding what the bird looks like, what the pipes look like, or where the bird and pipes are. Instead, it must learn these representations and interactions and be able to generalize due to the very large state space i.e. directly use the input and score to develop an opti

Github link to our repo

https://github.com/neerajBarthwal/FlappyBirds-DQNmal strategy for the game Flappy Bird.

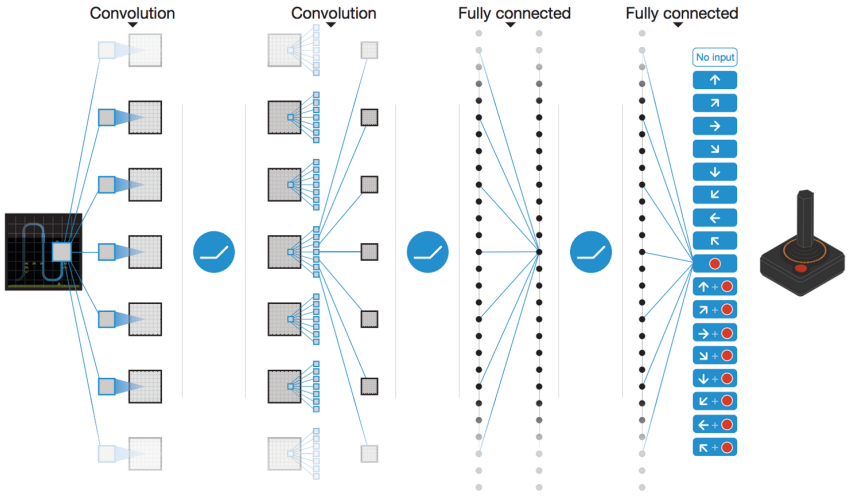


Figure 2: Using CNN to learn features from snapshots

Solving Flappy Birds and Problem Design: Reinforcement Learning

Reinforcement Learning

Reinfocement Learning is an aspect of Artificial Intelligence where an agent learns how to behave in an environment by performing actions and learning from the corresponding rewards.

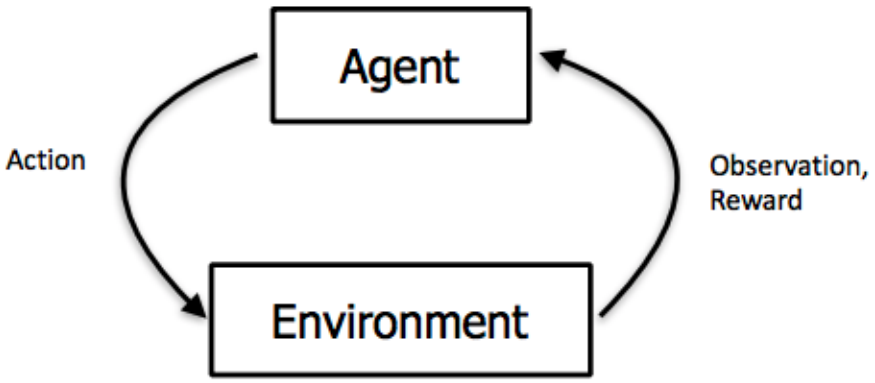


Figure 3: The action reward model

Reinforcement learning differs from standard supervised learning, in the sense that correct input/output pairs need not be presented, and sub-optimal actions need not be explicitly corrected. Instead the focus is on sequential decision making. What that means is, given the current input, you make a decision, and the next input depends on your this decision. In supervised learning, the decisions you make is either in a batch setting, or in an online setting, do not

affect what you see in the future.

Formulating RL problem: Markov Decision Process

The most common method to formulate a RL problem so that we can reason about it is to represent it as a Markov decision process. Suppose you are an agent, situated in an environment (e.g. Flappy Bird game). The environment is in a certain state (e.g. location of the pipe, location and direction of the bird, existence of gaps between the pipe). The agent can perform certain actions in the environment (e.g. flap or don't flap). These actions sometimes result in a reward (e.g. increase in score). Actions transform the environment and lead to a new state, where the agent can perform another action, and so on. The rules for how you choose those actions are called policy. The environment in general is stochastic, which means the next state may be somewhat random (e.g. the bird encounters a pipe and dies or passes through it).

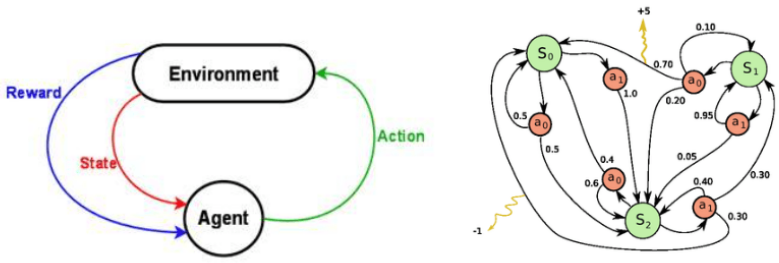


Figure 4: Left:Reinforcement Learning problem. Right:Markov Decision Process

The set of stat

Github link to our repo

https://github.com/neerajBarthwal/FlappyBirds-DQNes and actions, together with rules for transitioning from one state to another, make up a Markov decision process. One episode of this process (e.g. one game) forms a finite sequence of states, actions and rewards:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots, s_{n-1}, a_{n-1}, r_n, s_n$$

Here s_i represents the state, a_i is the action and r_{i+1} is the reward after performing the action. The episode ends with terminal state s_n (e.g. "game

over" screen). A Markov decision process relies on the Markov assumption, that the probability of the next state s_{i+1} depends only on current state s_i and action a_i, but not on preceding states or actions.

Discounted Future Reward

To perform well in the long-term, we need to take into account not only the immediate rewards, but also the future rewards we are going to get. How should we go about that?

Given one run of the Markov decision process, we can easily calculate the total reward for one episode:

$$R = r_1 + r_2 + r_3 + \ldots + r_n$$

Given that, the total future reward from time point t onward can be expressed as:

$$R_t = r_t + r_{t+1} + r_{t+2} + \ldots + r_n$$

But because our environment is stochastic, we can never be sure, if we will get the same rewards the next time we perform

Github link to our repo

https://github.com/neerajBarthwal/FlappyBirds-DQN the same actions. The more into the future we go, the more it may diverge. For that reason it is common to use discounted future reward instead:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \ldots + \gamma^{n-t} r_n$$

Here γ is the discount factor between 0 and 1 – the more into the future the reward is, the less we take it into consideration. It is easy to see, that discounted future reward at time step t can be expressed in terms of the same thing at time step t+1:

$$R_t = r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + \ldots)) = r_t + \gamma R_{t+1}$$

If we set the discount factor γ=0, then our strategy will be short-sighted and we rely only on the immediate rewards. If we want to balance between

immediate and future rewards, we should set discount factor to something like γ=0.9. If our environment is deterministic and the same actions always result in same rewards, then we can set discount factor γ=1.

A good strategy for an agent would be to **always choose an action that maximizes the (discounted) future reward.**

Q-Learning

In Q-learning we define a function Q(s, a) representing the maximum discounted future reward when we perform action a in state s, and continue optimally from that point on.

$$Q(s_t, a_t) = max\, R_{t+1}$$

Let's focus on just one transition ⟨s, a, r, s'⟩. Just like with discounted future rewards in the previous section, we can express the Q-value of state s and action a in terms of the Q-value of the next state s'.

$$Q(s, a) = r + \gamma max_{a'} Q(s', a')$$

This is called the **Bellman equation**. The maximum future reward for this state and action is the immediate reward plus maximum future reward for the next state.

The main idea in Q-learning is that we can iteratively approximate the Q-function using the Bellman equation. In the simplest case the Q-function is implemented as a table, with states as rows and actions as columns. **However, due to our large state space we use a neural network to approximate it.**

Approximating Q-function: Deep Q-Network (DQN):

This is the point where deep learning steps in. Neural networks are exceptionally good at coming up with good features for highly structured data. We could represent our Q-function with a neural network, that takes the state (four game screens) and action as input and outputs the corresponding Q-value. Alternatively we could take o

Github link to our repo

https://github.com/neerajBarthwal/FlappyBirds-DQNnly game screens as input and output the Q-value for each possible action. This approach has the

advantage, that if we want to perform a Q-value update or pick the action with the highest Q-value, we only have to do one forward pass through the network and have all Q-values for all actions available immediately.

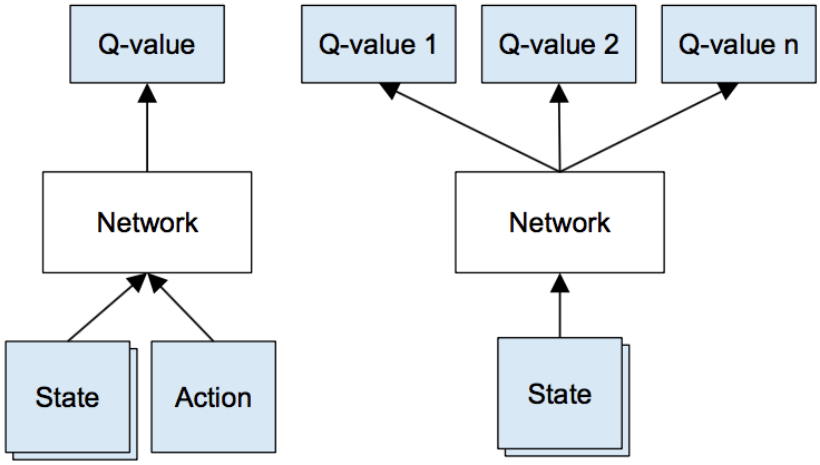


Figure 5: Left: Naive formulation of deep Q-network. Right: More optimized architecture of deep Q-network, used in DeepMind paper.

Here, the Q function can be represented with a neural network (since it is exceptionally good for learning features for highly structured data) that takes the state (last four game screens stacked) and action as input, returning the corresponding Q-value. This approach has the advantage where if we want to perform Q-value update or pick an action with highest Q-value we just need to do one forward pass through the network and have all Q-values for all the actions immediately.

Type	Filters	Filter Size	Strides	Activation
Conv-1	32	8x8	4	ReLU
Conv-2	64	4x4	2	ReLU
Conv-3	64	3x3	1	ReLU
Fully Connected-1	512			ReLU
Fully Connected-2	# actions (2)			Linear

Figure 6: Our CNN (inspired by the DeepMind Paper's architecture)

Input to the network are four 84×84 grayscale game screens. Outputs of the

network are Q-values for each possible action (flap or not). Q-values can be any real values, which makes it a regression task, that can be optimized with simple squared error loss.

$$L = \frac{1}{2}[\underbrace{r + max_{a'}Q(s', a')}_{\text{target}} - \underbrace{Q(s, a)}_{\text{prediction}}]^2$$

Figure 7: Loss function

Other techniques used to train the network and make it more stable

Exploration-Exploitation Tradeoff

Q-learning attempts to solve the credit assignment problem – it propagates rewards back in time, until it reaches the crucial decision point which was the actual cause for the obtained reward.

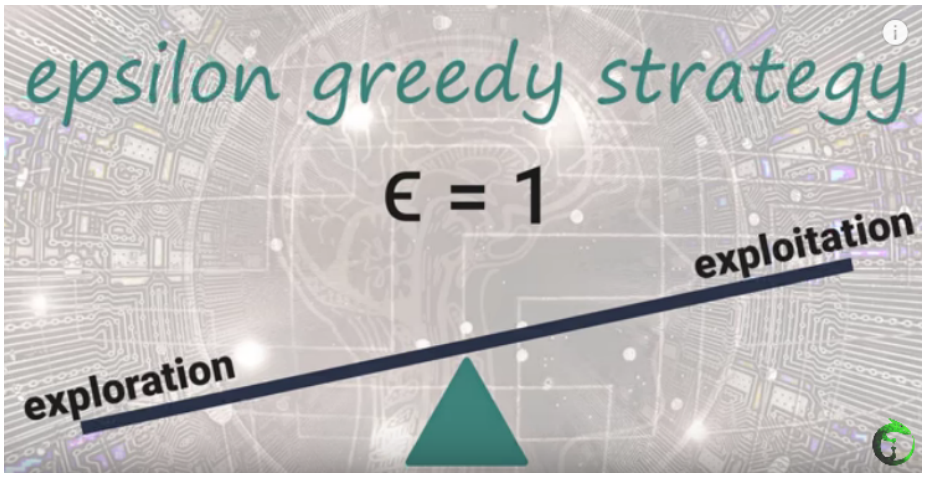


Figure 8: Exploration Exploitation Tradeoff

When a Q-network is initialized randomly, then its predictions are initially random as well. If we pick an action with the highest Q-value, the action will be random and the agent performs crude "exploration". As a Q-function converges, it returns more consistent Q-values and the amount of exploration decreases. So one could say, that Q-learning incorporates the exploration as part of the algorithm. But this exploration is "greedy", it settles with the first effective

strategy it finds. A simple and effective fix for the above problem is ϵ-greedy exploration – with probability ϵ choose a random action, otherwise go with the "greedy" action with the highest Q-value. We decrease ϵ over time from 0.9 to 0.0001 – in the beginning the system makes completely random moves to explore the state space maximally, and then it settles down to a fixed exploration rate.

Fixed Q-taregts

In the Deep Q Learning, we calculate the difference between the target (Q_target) and the current Q value (estimation of Q) as the loss.

But we don't have any idea of the real target Q-value. We need to estimate it. Using the Bellman equation, we saw that thetarget is just the reward of taking that action at that state plus the discounted highest Q value for the next state.

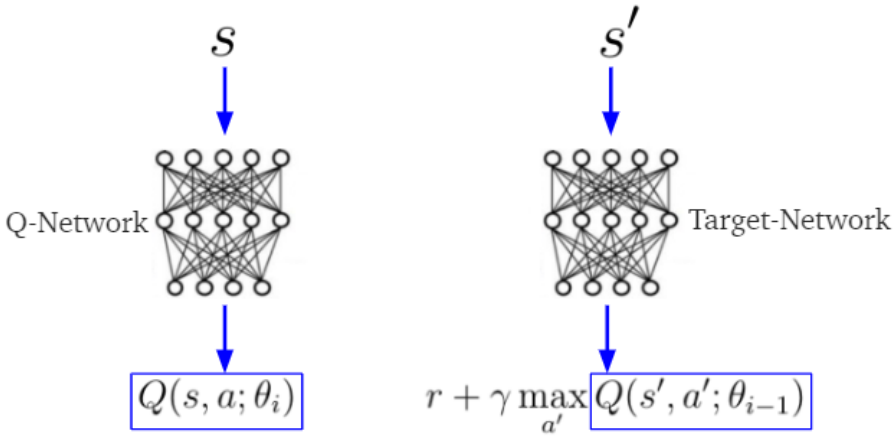
However, the problem is that we using the same parameters (weights) for estimating the target and the Q value. As a consequence, there is a big correlation between the target and the parameters (w) we are changing.

Therefore, it means that at every step of training, **our Q values shift but also the target value shifts**. So, we're getting closer to our target but the target is also moving. It's like chasing a moving target! This lead to a big oscillation in training.

Solution: Using a separate network with a fixed parameter for estimating the target Q-value.

Double DQNs

By calculating the target Q-value, we face a simple problem: **how are we sure that the best action for the next state is the action with the highest Q-value?** We know that the accuracy of q values depends on what action we tried and what neighboring states we explored.



As a consequence, at the beginning of the training we don't have enough information about the best action to take. Therefore, taking the maximum q value (which is noisy) as the best action to take can lead to false positives. If non-optimal actions are regularly given a higher Q value than the optimal best action, the learning will be complicated.

The solution is: when we compute the Q target, we use two networks to decouple the action selection from the target Q value generation. We:

- use our DQN network to select what is the best action to take for the next state (the action with the highest Q value).
- use our target network to calculate the target Q value of taking that action at the next state.

Game parameters changed for faster convergence

- Removed (blackened) the dynamic colourful background
- Removed the score readout, keeping each frame simple and clear
- Removed sound the effects
- Removed the base of the pipes (same for every frame, thus redundant)
- Fixed the bird colour (as opposed to randomly assigned colour previously)
- Removed display used by gameplay and trained just by seeing the cumulative rewards increase per episode over the course of training.

Game specific parameters

In our context, each state can be defined using 3 independent parameters:

- horizontal distance of the bird from the pipe
- vertical distance of the bird from the pipe
- vertical speed of the bird
- Only two possible legal actions to Flap or Not.

Intuitive Rewards

- Crashing: Negative (harsh)
- Surviving: Minimally positive
- Crossing a pipe: Positive

Goal: To maximise the long term reward

Details of assigned research paper

The research paper aims at explaining the model formulation , Q-learning , Dqn architecture amd experience replay and stability.

- In Q-learning bellman equation is used as an iterative update

$$Q_{i+1}(s,a) = r + \gamma \max_{a'} Q_i\left(s', a'\right) \tag{1}$$

- The task is to model the function as a Convolutional Neural Network and update its parameters using the update rule in above equation. Below equations would be the loss function and its gradient to model this function.

$$L = \sum_{s,a,r,s'} \left(Q(s,a;\theta) - \left(r + \gamma \max_{a'} Q\left(s', a'; \theta^{-}\right)\right)\right)^2 \tag{2}$$

$$\nabla_\theta L = \sum_{s,a,r,s^s} -2\left(Q(s,a;\theta) - \left(r + \gamma \max_{a'} Q\left(s', a'; \theta^{-}\right)\right)\right) \nabla_\theta Q(s,a;\theta) \tag{3}$$

Challenges

- The initial challenge was understanding the research paper. We read through Atari games paper as it made probem formulation easy to understand.
- Understanding reinforcement learning took some time initially.
- The main issue was availability of Gpu.We tried running on google cloud but after some iterations, we were getting ssh connection refused from Vm side. It takes 1 lakh iteration just to save replay memory. We tried contacting google cloud support, but they were quite reluctant to help as it is a trial account.

- Another issue was , cannot execute code in other mediums like collab as pygame documentation mandates the use of a video driver.
- Pygame uses display and sound drivers , and when we were running it on Gpu it was throwing No sound driver error , so we had to debug and comment out the sound driver part after understanding the Api.

Work Distribution

- **2018201069** Understood and implemented the training algorithm for DDQN.
- **2018201101** Created initial POC for model and tested feasibility of the network for training.
- **2018202017** Understood and decided the hyperparameters of model which helped in fast convergence.
- **2018201099** Implemented target network feature and fixed Q-values

Common Work:
Complete understanding of both the research papers and other improvement techniques.
Debugging and deploying the model on various VMs for training.
Testing different platforms for model training.

Literature Survey

Referred research papers -

- cs229.stanford.edu/proj2015/362_report.pdf
- cs231n.stanford.edu/reports/2016/pdfs/111_Report.pdf

Github link to our repo

https://github.com/neerajBarthwal/FlappyBirds-DQN