Playing Flappy Bird using reinforcement learning with neural network

CS263C Project Report

By Tao Zhou & Han Wang

Contents

1. Introduction

The game, Flappy Bird (Figure 1), is simple at first glance. Just tap the screen to make a little bird "flap" and have some vertical velocity. After that, the bird falls (accelerates down). The goal is to get the bird to "fly-fall" through some openings in pipes. However, Flappy Bird was a hard game according to the players' evaluation and the brutal challenge is part of what made the game so infamously additive.

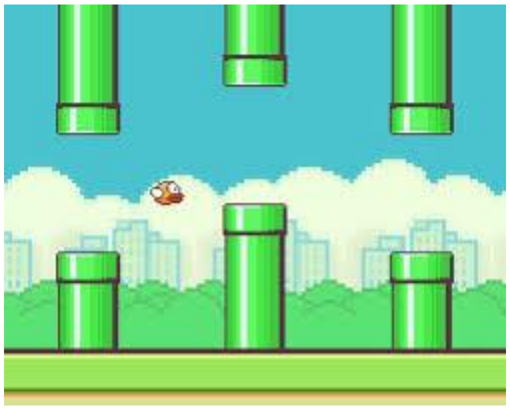


Figure 1. Flappy bird game

Since the game is extremely hard for human players, people saw the opportunity to train the bird to learn how to play the game by itself via reinforcement learning. Although the game is no longer available on Google Play or the App Store, it did not stop folks from creating very good replicas for the web. Sarvagyavaish found an alternative engine online and ripped out the typing component and added some javascript code to it [1].

The objective of the game was to direct a flying bird who moves continuously to the right between pipe pairs. If the bird touches the pipes, they lose. The bird briefly flaps upward each time once the player taps the screen ("click"); if the screen is not tapped, the bird falls because of gravity ("do nothing"). Thus, the bird has only two actions to choose. Each pair of pipes that he navigates between earns the player a single point and the distance between the lower and upper pipe is a constant across pipes pairs [2].

2. Components of the system

2.1 Reinforcement learning

In reinforcement learning (Q learning), an agent takes actions to maximize a cumulative reward. We try to create a “ghost” agent that plays “Flappy Bird” in an emulator and maximizes its performance measured by its score. The basic idea of Q learning is that in different environmental states you may take several possible actions and for different combination of actions and states, you may get different outcomes. Regarding the goodness of different outcomes, you will have different rewards. That is, after you try an action at a given state, you first evaluate the next state it will lead to and judge whether the next state is good or not. If it has led to a bad state, you will reduce the Q value of that action from that state. So in the next running time, other actions are more likely to be taken in the Q same state. The reward of each of those actions in each of those states can be accumulated and changed by learning in each step. The basic formulation of reinforcement learning can be shown as following:

$$Q_{t+1}(s_t, a_t) = R_t + \gamma \cdot \max Q_t (s_{t+1}, a_{t+1})$$

Here, R_t refers to the immediate reward you get, γ means the learning rate and $\max Q_t(s_{t+1}, a_{t+1})$ is determined by the biggest reward you can earned compared multiple combinations of states and action. In general, the Q value is determined by three representations - immediate reward, supposed state and action. This value is equal to the immediate reward plus the discounted value of the biggest reward you can earn in next step. The pseudocode of the algorithm can be shown as the following:

Algorithm 1 Q-learning Algorithm

```
Initialize matrix Q to zero
Initialize matrix R of states and actions; Set the gamma parameter
Set environment rewards in R; For each episode:
  Select initial state ”running” While state = ”running” do
    Select action from ”click” or ”do nothing” Get maximum Q value based on all actions
    Compute Q value = R(state, action) +
      gamma * Max(Q(next state, all actions)) Set next state as current state
  End Do
```

In fact, Q function could be absolutely anything besides a simple look-up table. It could be a linear, a polynomial or randomized trees. Here we use a neural-network to store q-value instead of look-up table for Q-values (Q-network). The Q-network can learn successful policies directly using end-to-end reinforcement learning.

However, reinforcement learning is known to be unstable or even to diverge when a neural network is used to represent Q (also known as action-value) function. This instability has several causes: the correlations present in the sequence of observations, the fact that small updates to Q may significantly change the policy and therefore change the data distribution, and the correlations between the action-values (Q) and the target values. To use reinforcement learning successfully in situations approaching real-world complexity, however, agents are confronted with a difficult task: they must derive efficient representations of the environment from high-dimensional sensory inputs, and use these to generalize past experience to new situations.

The instabilities mentioned above are addressed using two key ideas. First, we used a biologically inspired mechanism termed experience replay that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution. Second, we used an iterative update that adjusts the action-values (Q) towards target values that are only periodically updated, thereby reducing correlations with the target [3][4].

As for the action selection, we used ε-greedy algorithm -- the algorithm chooses a "learned" action with probability 1 - ε and a random action with probability ε. The value of ε is gradually decreased as the algorithm learns to play better. In fact, it is necessary to sometimes pick a random action to not get stuck in local reward maxima. At the beginning, the value of ε is relatively big and the agent takes random actions most of the time. In this way, the agent should be able to collect information about the environment. When ε starts to decrease, the agent starts to apply its learned behavior. It is worth mentioning that ε never reaches zero and therefore it still does some exploration even when performing well in the game.

2.2 Neural Network

The neural Q learning was implemented in the ConvNetJS library to train our bird to fly. The ConvNetJS Q learning algorithm is based on the Q learning with experience replay algorithm described in the paper Playing Atari with Deep Reinforcement Learning. ConvNetJS is a Javascript library for training Deep Learning models (mainly Neural Networks) entirely in your browser. Open a tab and you're training.

However, there are several differences between our implementation and the paper. On the one hand, we did not use deep learning. We do not have advanced GPU and it takes tremendous time to learn deep learning tools, like Caffee and Theano. Thus, the hidden layer only has two layers. On the other hand, we did not use pixels as input to the neural network. If so, the dimension of state space will be huge. Instead, a smaller state space (2 or 3 dimensions) was utilized in the training. The library provided the basic setting for Convolution Neural Network (CNN); however, the library is not restricted to CNN. The library also has Recurrent Neural Network (RNN) or common fully connected neural network.

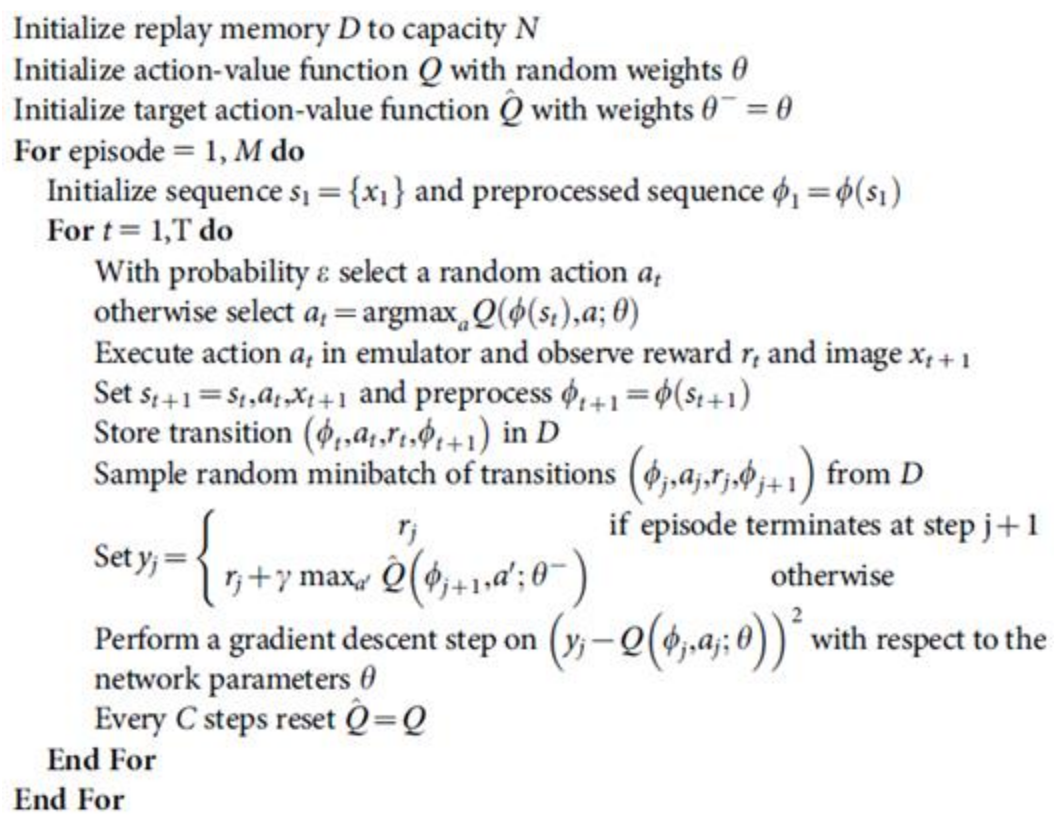
In this project, we specify a three layer neural network with two hidden layers of fifty neurons. The input layer declares the size of input, which is determined by the number of states, number of actions, number of rewards and number of their combinations. Then two layers of fifty neurons are declared followed by ReLU (Rectified Linear Unit). Finally, a linear classifier on top of previous hidden layers is declared. We design the system using a neural network to assign an expected reward value to each possible action.

2.3 Root Mean Squares of gradients (RMSProp)

The training of NN is based on gradient descent. There are problems with the approach -- the gradient does not change with the same rate along different direction. It might be not very reasonable to use the same learning rate for all components. The idea of RMSProp is to come up with a separate learning rate for different directions. If you can use all the data to estimate the direction of the gradient, then you can cope with this problem by using sign of the gradient sign. This will ensure that all components of the gradient are treated equally. This approach is called

rprop. Due to time limitation, we cannot have all the data for estimation and usually small sets of subsamples are utilized. Unfortunately rprop does not work with small sets of subsamples (minibatches). Thus, we use a variant of Prop method – RMSProp. After the result is obtained using rprop method, we then divide the data by the magnitude of the gradient. RMSProp proposes to keep track of previous gradients and divide updates not by the current magnitude, but also by the magnitude averaged across the last several updates. This will allow to modify the gradient according to its previous magnitudes, preventing from taking it into account with those extreme weights.
The full algorithm for training neural Q-networks is presented in Algorithm 2.
At the beginning, the system initializes the parameters, like replay memory D, action-value function Q with random weights θ and target action-value function, etc. Within each episode (=1,…, M, note: one episode is defined as the start to the end of the game with T iterations), the vertical/horizontal distance between the bird and the closet lower pipe are set as the current state ϕ_t.
The agent selects and executes actions according to an ε-greedy policy based on Q. According to this policy, the agent selects $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$ with probability 1-ε and selects a random action with probability ε. The action is executed and the agent observes reward r_t and image x_{t+1}. The transitions $(\phi_t, a_t, r_t, \phi_{t+1})$ will be stored in replay memory.
Then, the system will learn a new policy – random minibatch samples of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ are sampled from replay memory D and approximate target value y_j is calculated based on bellman equation. The data y_j is then fed into neural netwrok optimized via RMSProp.

Algorithm 2: neural Q-learning with experience replay



```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t$ = 1,T do
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the
    network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  End For
End For
```

3. Implementation details

As for an initial try, we chose the vertical distance (the vertical red arrow) and the horizontal distance (the horizontal red arrow) between the bird and the lower pipe as the states of the bird (Figure 2).

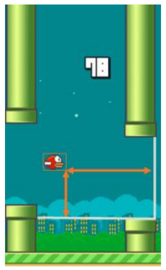


Figure 2. Flappy bird states

These states are then passed through two successive fully connected hidden layers to the output layer. The output layer has one node for each possible action (two actions: “do nothing” and “click”) and the activation of those nodes indicates the expected reward from each of the possible actions. Finally, the action with the highest expected reward is selected for execution.

We set up the neural network in ConvNetJS library for training. At the beginning, we used deeqlearn.Brain: to initialize the network. Then we trained the neural network using the following routines: 1) var action = brain.forward(array_with_num_inputs_numbers); action is a number telling index of the action the agent chooses. After that, we applied the action on environment and observe some reward. 2) brain.backward(reward); We fed the reward back into the neural network.

We started out using the default options set by the ConvNetJS library. The properties are outlined below:

temporal_window: 1
total_learning_steps: 100000
start_learn_threshold: 1000
learning_steps_burnin: 3000
learning_rate: 0.01
epsilon_min: 0.05
gamma: 0.8

But the bird could not get through the first pipe often enough to realize that this was a good thing. Because the total learning steps was 100k, the epsilon dropped and the bird was only able to learn that not flapping would cause it to die. Therefore, the bird learned to always flap to the top, and then died once it hit the first pipe. Therefore, the bird always learned to flap to the top and died once it hit the first pipe. The reward model we were using was: 1) dead: -100; 2) alive: 1.

Thus, we also tried to reward the bird heavily for flying through a pipe and not rewarding for just being alive and follow the some other brain setting as shown below.

temporal_window: 3
total_learning_steps: 1000000
start_learn_threshold: 1000
learning_steps_burnin: 50000
learning_rate: 0.01
epsilon_min: 0.05
gamma: 0.7

In this experiment, we tested different combination of state spaces into the neural network:

1) Vertical distance and horizontal distance from the lower pipe
2) Vertical position of the bird, vertical position of the lower pipe and horizontal distance from the lower pipe (Figure 3)
3) We also tried using position or distance normalized by pixel numbers along two directions

We also tested different types of reward:

1) Running (alive): +1, dead: -1000.
2) Running (alive):0, dead: -1, past the pipe: +100.
3) We also tried to make the reward smaller. If there are two reward situations, we set running (alive) as +0.01 and dead as -1; for three reward situations, we set running (alive) as 0, deal as -0.1 and pass the pipe as +1.

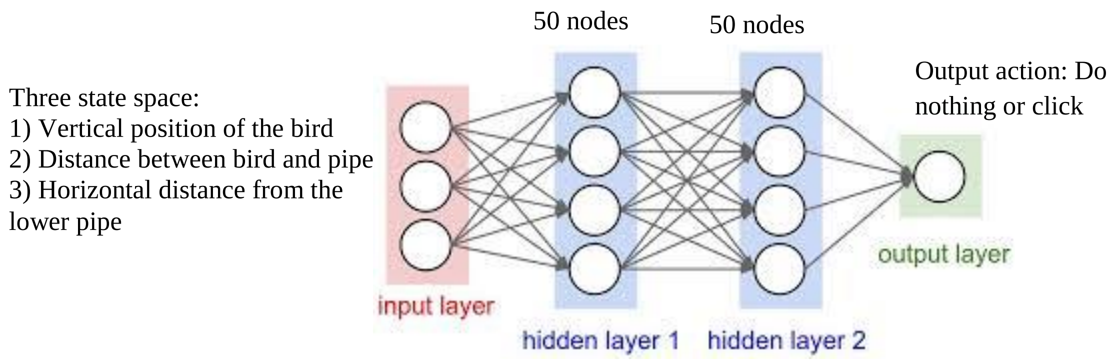

50 nodes
50 nodes
Three state space:
1) Vertical position of the bird
2) Distance between bird and pipe
3) Horizontal distance from the lower pipe
Output action: Do nothing or click
input layer
hidden layer 1
hidden layer 2
output layer

Figure 3. The neural network diagram for three state spaces

However, the bird could only flap so far. The bird passed the first few pipes with mild change of pipes height and often failed if the pipe lowered or rose suddenly. The fine tuning of NN parameters are needed for high level performance of the bird.

In this project, Tao Zhou is mainly responsible for coding and the tuning of neural network parameters. He combines the existing flappy bird game engine and the ConvNetJS library together. The code was written in Javascript and executed in Firefox. Han Wang is mainly responsible for understanding Q-learning and figure out how to replace look-up table with neural network.

Reference

[1] http://sarvagyavaish.github.io/FlappyBirdRL/

[2] https://en.wikipedia.org/wiki/Flappy_Bird

[3] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (Dec 2013). Playing Atari with deep reinforcement learning. Technical Report arXiv: 1312.5602, 2013.

[4] V Mnih, K Kavukcuoglu, D Silver, A A Rusu, J Veness, et al. Human-level control through deep reinforcement learning. Nature, 518(7540):529–533, 2015.

[5] http://cs.stanford.edu/people/karpathy/convnetjs/demo/rldemo.html

[6] http://rdrapeau.github.io/neural-net-nes/2015/11/15/teaching-flappy-to-win.html

[7]https://en.wikipedia.org/wiki/Q-learning

[8]http://mnemstudio.org/path-finding-q-learning.html

The mechanism code:

1) Code for “brain” (Neural Network) initialization:

```
function initilizeBrain()
{
        var num_inputs = 3; // horizontal, vertical
        var num_actions = 2; // do nothing, tick
        var temporal_window = 3; // amount of temporal memory. 0 = agent lives in-the-moment :)
        var network_size = num_inputs*temporal_window + num_actions*temporal_window + num_inputs;
        // the value function network computes a value of taking any of the possible actions
        // given an input state. Here we specify one explicitly the hard way
        // but user could also equivalently instead use opt.hidden_layer_sizes = [20,20]
        // to just insert simple relu hidden layers.
        var layer_defs = [];
        layer_defs.push({type:'input', out_sx:1, out_sy:1, out_depth:network_size});
        layer_defs.push({type:'fc', num_neurons: 50, activation:'relu'});
        layer_defs.push({type:'fc', num_neurons: 50, activation:'relu'});
        layer_defs.push({type:'regression', num_neurons:num_actions});
        // options for the Temporal Difference learner that trains the above net
        // by backpropping the temporal difference learning rule.
        var tdtrainer_options = {learning_rate:0.01, momentum:0.0, batch_size:64, l2_decay:0.01};
        var opt = {};
        opt.temporal_window = temporal_window;
        opt.experience_size = 30000;
        opt.start_learn_threshold = 1000;
        opt.gamma = 0.7;
        opt.learning_steps_total = 1000000;
        opt.learning_steps_burnin = 50000;
        opt.epsilon_min = 0.05;
        opt.epsilon_test_time = 0.05;
        opt.layer_defs = layer_defs;
        opt.tdtrainer_options = tdtrainer_options;
        return new deepqlearn.Brain(num_inputs, num_actions, opt);
}
```

2) Code for action specification based on the input states to the neural network

```
function getActionByIndex(action_index)
{
        var bird_action = "";
        if( action_index === 0 ){bird_action = "do_nothing"}
        else{bird_action = "click";}
        return bird_action;
}
```

3) Code for reward based on the selected action

```
function getReward(my_bird)
{
        var reward = 0;
        if( my_bird !== undefined && my_bird.state !== undefined )
        {
                var current_state = my_bird.state.get();
                switch (current_state)
                {
                        case "RUNNING":
                                reward = 0;
                                break;
```

```
                    case "DYING":
                        reward = -1;
                        break;
                    case "SCORE":
                        reward = 100;
                        my_bird.state.set("RUNNING");
                        break;
                }
        }
        return reward;
}
```

4) Code for controlling the bird

```
tick: function () {
        this.state.tick();
        this.bird.tick();
        var valid = false;
        var reward = 0;
        switch (this.state.get()) {
                case "BORN":
                        this.state.set("RUNNING");
                        this.bird.state.set("CRUSING");
                break;
                case "RUNNING":
                        if (this.state.first()) {
                                this.bird.state.set("RUNNING");
                        }
                        this.tick_RUNNING();
                        valid = true;
                        break;
                case "SCORE":
                        valid = true;
                        break;
                case "DYING":
                        this.state.set("GAMEOVER");
                        valid = true;
                        break;
                case "GAMEOVER":
                        if (this.state.first()) {
                                if (this.score > window.game.best) {
                                        window.game.best = this.score;
                                }
                        }
                        age ++;
                        this.reset();
                        this.state.set("BORN");
                        break;
                }
                if (valid) {
                        // Step 2: Observe State S'
                        var horizontal_distance = 9999;
                        var vertical_distance = 9999;
                        var bird_pos = 9999;
                        var pipe_pos = 9999;
                        var pipe2_pos = 9999;
                        var pipex1 = 9999;
                        var pipex2 = 9999;
                        for (var i = 0; i < 6; i++)
                        {
                                if (this.pipes[i].dir == "up" && this.pipes[i].x + this.pipes[i].w >= this.bird.x)
                                {
                                        var diff = (this.pipes[i].x + this.pipes[i].w - this.bird.x);
                                        if (horizontal_distance > diff) {
                                                horizontal_distance = diff;
                                                bird_pos = this.bird.y;
                                                pipe_pos = this.pipes[i].y;
                                        }
                                }
                        }
                        var input_array = new Array( 3 )
                        input_array[0] = bird_pos;
                        input_array[1] = pipe_pos;
                        input_array[2] = horizontal_distance;
                        var action_index = this.brain.forward(input_array);
                        var action_to_do = getActionByIndex(action_index);
                        this.action_to_perform =  action_to_do;
                        var reward = getReward(this);
                        this.brain.backward(reward);
                        if (this.action_to_perform == "click") {
                                this.bird.performJump();
                        }
                }
                if (this.shake && !this.shake.tick()) {
                        this.shake = null;
                }
                if (this.flash && !this.flash.tick()) {
                        this.flash = null;
                }
        },
```