# Playing FlappyBird with Deep Reinforcement Learning

**Naveen Appiah**
Mechanical Engineering
nappiahb@stanford.edu

**Sagar Vare**
Stanford ICME
svare@stanford.edu

## Abstract

*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.*

## 1 Introduction- Problem Definition

Flappy bird (Figure 1) is a game in which the player guides the bird, which is the "hero" of the game through the space between pairs of pipes. At each instant there are two actions that the player can take: to press the 'up' key, which makes the bird jump upward or not pressing any key, which makes it descend at a constant rate.

Today, the recent advances in deep neural networks, in which several layers of nodes are used to build up progressively more abstract representations of the data, have made it possible for machine learning models to learn concepts such as object categories directly from raw sensory data. It is has also been observed that deep convolutional networks, which use hierarchical layers of tiled convolutional filters to mimic the effects of receptive fields produce promising results in solving computer vision problems such as classification and detection. The goal here is to develop a deep neural network to learn game specific features just from the raw pixels and decide on what actions to take. Inspired by [1] and [2], we propose a reinforcement learning set-up to learn and play this game..

Reinforcement learning is essential when it is not sufficient to tackle problems by programming the agent with just a few predetermined behaviors. It is a way to teach the agent to make the right decisions under uncertainty and with very high dimensional input (such as a camera) by making it experiencing scenarios. In this way, the learning can happen online and the agent can learn to react to even the rarest of scenarios which the brutal programming would never consider.

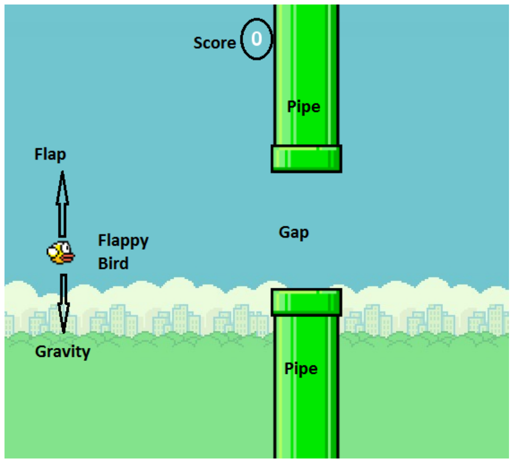


Figure 1: Flappybird Game - Schematics

## 2 Related Work

Google Deepmind's efforts to use Deep learning techniques to play games have paved way to looking at artificial Intelligence problems with a completely different lens. Their recent success, AlphaGo [4], the Go agent that has been giving a stiff competition to the experts in the game show clearly the potential of what Deep learning is

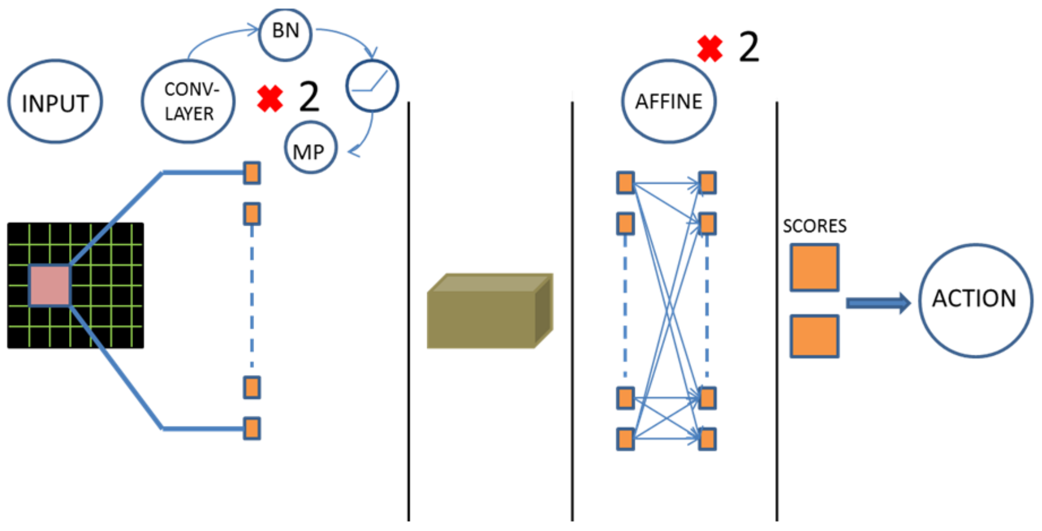


Figure 2: Schematic Architecture of the Convolutional Neural Network.

capable of. Deepmind's previous venture was to learn and play the Atari 2600 games just from the raw pixel data. Mnih et al. are able to successfully train agents to play these games using reinforcement learning, surpassing human expert-level on multiple games [1],[2]. Here, they have developed a novel agent, a deep Q-network (DQN) combining reinforcement learning with deep neural networks. The deep Neural Networks acts as the approximate function to represent the Q-value (action-value) in Q-learning. They also discuss a few techniques to improve the efficiency of training and better the stability. They use a "experience replay" of previous experiences from which mini-batches are randomly sampled to update the network so as to de-correlate experiences and delayed updates for the cloned model from which target values are obtained (explained in detail later) to better the stability. Another advantage of this pipeline is the complete absence of labeled data. The model learns by playing with the game emulator and learns to make good decisions over time. It is this simple learning framework and their stupendous results in playing the Atari games, inspired us to implement a similar algorithm for this project.

## 3 Methods

In the following section, we describe how the model is parameterized and the Q-learning algorithm. The task of the AI agent when the model gets deployed is to extract images of game instances and output the necessary action to be taken from the set of feasible actions that can be taken. This is similar to a classification problem. Unlike the common classification problem, we don't have labeled data to train the model on. Instead, a reinforcement learning setting tries to evaluate the actions at a given state based on the reward it observes by executing it.

### 3.1 Model formulation

The **actions** the bird can take are to flap ($a = 1$) or do nothing ($a = 0$). The state at time (frame) $t$ is derived by preprocessing the raw image of the current frame ($x_t$) with a finite number of previous frames ($x_{t-1}, x_{t-2}, ...$). This way, each state will uniquely recognize the trajectory the bird had followed to reach that position and thus provide temporal information to the agent. The number of previous frames stored becomes a hyper-parameter. Ideally, $s_t$ should be a function of all frames from $t = 1$ but to reduce the state-space, only a finite number of frames are used.

As we know, that the bird dies when it hits the pipe or the edges of the screen, we can associate a negative reward for bird crashing and a positive reward if it passes through the gap. This will be close to what a human player tries to do, i.e try to avoid dying and score as many points as possible. Therefore, there are two rewards, $rewardPass$ and $rewardDie$. A discount factor ($\gamma$) of 0.9 is used to discount the rewards propagated from the future action-values

## 4 Q-learning

The goal of reinforcement learning is to maximize the total pay-off (reward). In Q-learning, which is off-policy, we use the bellman equation as an iterative update

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

where, $s'$ and $a'$ are the state and action at the next frame, $r$ is the reward and $\gamma$ is the discount factor. $Q_i(s,a)$ is the Q-value for $(s,a)$ at the $i^{th}$ iteration. It can be shown that this iterative update converges to an optimal Q-function. To prevent rote learning, this action value function can be approximated with a function (deep network in this case) so that it can generalize to unseen states as well. This update neglects the stochasticity in state transitions which is the case in this game. A sequence of observations $s_t, a_t, r_t, s_{t+1}$ becomes an input point to the learning algorithm. Basically, the Q-function should fit through a bunch of such input points to create a model that favors actions that maximizes the overall reward. The task is to model this function as a Convolutional Neural Network and update its parameters using the update rule in equation 1. Equations 2 and 3 would be the loss function and its gradient to model this function.

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

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

Here, $\theta$ are the DQN parameters that get trained and $\theta^-$ (explained in later sections) are non updated parameters for the Q-value function. Thus, we can simply use stochastic gradient descent and backpropagation on the above loss function to update the weights $(\theta)$ of the network. The training methodology we plan for is inspired by the work in [3]. 1 is the algorithm we have devised for training. Training involves an $\epsilon$-greedy approach to increase exploration. That is, when we are training, we select a random action with probability $\epsilon$ or otherwise choose the optimal action $a_{opt} = argmax_{a'} Q(s,a';\theta)$. The $\epsilon$ anneals linearly to zero with increase in number of updates.

## 4.1 Pre-Processing

The direct output of the Flappy bird game is 284×512, but to save on memory we worked with down-sized images of 64×64. Each image is on a 0-255 color scale. Additionally to improve the accuracy of the convolutional network, the background layer was removed and substituted with a pure black image to remove noise. To process a set of images from a finite number of frames before the current frame into a state (as mentioned in model formulation section), the following approach is used. The current frame is overlapped with the previous frames with slightly reduced intensities and the intensity reduces as we move farther away from the most recent frame. Thus, the input image will give good information on the trajectory on which the bird is currently in.

## 4.2 DQN Architecture

In the current architecture we have 3 hidden layers as shown in Figure 2. First we have two Convolutional layers followed by two fully connected layers. The output of the final fully connected layer is the score for the two actions, which is given to the loss function. The modified loss function helps us learn in a Q-learning setting. We have a spatial batch norm, ReLu and max pooling layer after every convolutional layer. Additionally, there is a ReLu and batch norm after the first affine layer (the output of the batchnorm is fed into the final affine layer). For the convolutional layers we have used 32 Filters of size 3 and stride 1 with padding and max pooling with a 2×2 kernals. The input image is of the size 64 × 64. There are two possible output actions at each instant, and we get a score value for each of the actions for deciding on the best.

## 4.3 Experience Replay and Stability

In Q-learning, the experiences recorded in a sequential manner are highly co-related. If they are used in the same sequence to update the DQN parameters, the training process will be hindered. Similar to sampling a mini-batch from a labeled data set to train a classification model, we should bring in some randomness in the experiences that get selected for updating the DQN. To make this possible, a $replayMemory$ is setup which stores the experience $(s_t, a_t, r_t, s_{t+1})$ at every frame until the max size $numReplayMemory$ is reached. After the replay memory is filled to a certain number, a mini-batch of experiences is sampled randomly and used to run a gradient descent on the DQN parameters. This update to the DQN parameters happens at regular intervals. As a result of this randomness in the choice of the mini-batch, the data that goes in to update the DQN parameters are likely to be de-correlated.

To better the stability of the convergence of the loss functions, we use a clone of the DQN model with parameters $\theta^-$ as shown in equation 2 in the bellman update equation. The parameters $\theta^-$ are updated to $\theta$ after every C updates to the DQN. This $\theta^-$ is used to calculate $y_j$ as shown in algorithm 1

## 4.4 Training Setup

The pipeline for the entire DQN training process is shown in Algorithm 1. As mentioned in the previous parts of this section, the experiences are stored in a replay memory and at regular intervals, a random mini-batch of experiences are sampled from the memory and used to perform a gradient descent on the DQN parameters. Then we update the exploration probability as well as the target network parameters $\theta^-$ if necessary.

**Algorithm 1** Deep Reinforcement learning

```
 1: Initialize replay memory D to certain capacity
 2: Initialize the Q-value function with random weights $\theta$
 3: Initialize $\theta^- = \theta$
 4: for games = 1 → maxGames do
 5:     for snapShots = 1 → T do
 6:         With probability $\epsilon$ select a random action $a_t$
 7:         otherwise select $a_t = argmax_a Q(s_t, a; \theta)$
 8:         Execute $a_t$ and observe $r_t$ and next sate $s_{t+1}$
 9:         Store transition $s_t, a_t, r_t, s_{t+1}$ in D
10:         Sample mini-batch of transitions from D
11:         for j = 1 → size of minibatch do
12:             if game terminates at next state then
13:                 $y_j = r_f$
14:             else
15:                 $y_j = r_j + \gamma \max_{a'} Q(s', a'; \theta^-))$
16:             end if
17:         end for
18:         Perform gradient descent on the loss w.r.t $\theta$
19:         Every C steps reset $\theta^- = \theta$
20:     end for
21: end for
```

The score of the output game is the sole evaluation metric. To make the results robust, we take an average score over a few games rather than a single one. The $\epsilon$ factor is set to zero during test and while training, we use a decaying value. This is to model the surety of our decisions as we train and learn more.

## 5 Experiments and Results

### 5.1 Training parameters

**Model Parameters:** The Flappy bird is played at 10 frames per second, 3 recent frames are processed to generate a state, the discount factor $\gamma$ is set to 0.9 and the rewards are as follows: $rewardPass = +1.0$ and $rewardDie = -1.0$.

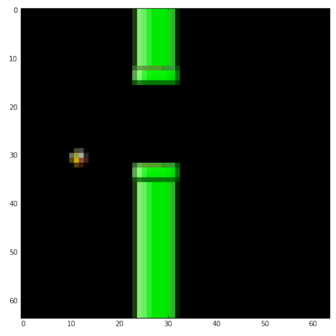
**DQN parameters:** The exploration probability ($\epsilon$) linearly decreased from 0.6 to 0 in 1000 updates. The size of the replay memory is set to 1000 and the mini-batches are sampled once it has 500 experiences. The parameters of the target model $\theta^-$ are updated every C=100 updates. A mini-batch of 32 is randomly sampled every 5 frames to update the DQN parameters.

**Training parameters:** The Gradient descent update rule used to update the DQN parameters is $Adam$ with a learning rate $1e-6$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$. These parameters were chosen on a trial and error basis observing the convergence of the loss value. Our convolution weights are initialized to have a normal distribution with mean 0 and variance $1e-2$.

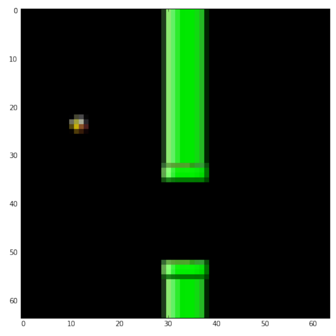
The whole DQN architecture and the Q-learning setup was developed in python using numpy and matplotlib libraries. The game emulator is also a python-pygame implementation found at https://github.com/TimoWilken/flappy-bird-pygame.git

### 5.2 Results and Analysis

After training, a few snapshots of the game were tested with the model to see if results made sense. Figure 3 shows some example snaps and their corresponding scores which make perfect sense.



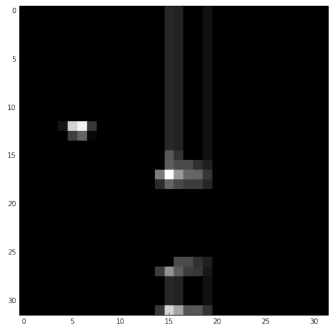
(a) Score: UP = **1.870**, DOWN = -1.830



(b) Score: UP = -1.999, DOWN = 1.983

Figure 3: Example snapshots with their corresponding scores. 3a is scenario where the bird has to jump up and 3b is a scenario where the bird has to go down

To understand more about the working of the trained CNN model, test image 3b was visualized after the convolution layers to notice the activation. It could be seen that most activation show clear patches on the edges of the gap and the bird (Figure 4). we can clearly infer that the network is learning to find the relative position of the bird with respect to the gap


(a) After 1st conv layer

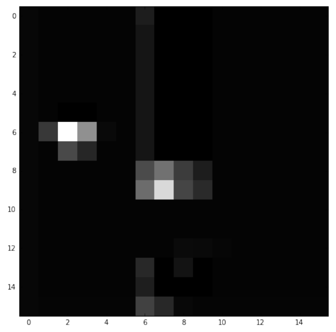
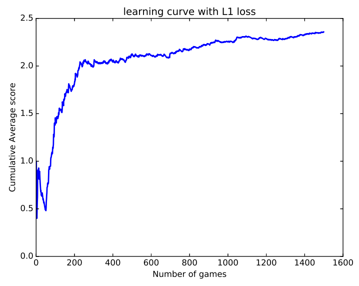

(b) After 2nd conv layer

Figure 4: Activation after convolution layers

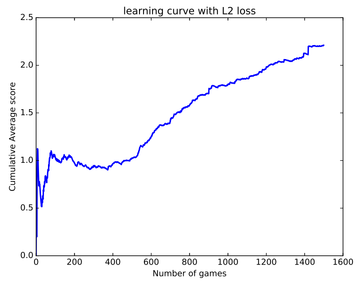
In most of the experiments done in [3] on Atari games, the loss function used is L2. We experimented with a L1 loss (equation 4) to introduce some regularization. This resulted in a very steep learning rate in the beginning as seen in Figure 5. In both cases, it can be seen that there is steady increase in the average score showing that the model is learning steadily.

$$L = \sum_{s,a,r,s'} |Q(s,a;\theta) - (r + \gamma \max_{a'} Q(s',a';\theta^{-}))| \tag{4}$$

This is a video of the agent playing the game https://www.youtube.com/watch?v=vLMpEx9lSuo. It can be noticed that even though the bird dies sometimes, it is seen that it tries hard to reach for the gaps and mostly crashes at the edges of the gap. Possible solution could be to use a different reward scheme that would make the bird take a path farthest from both the top and bottom pipes. Intuitively, humans playing it also try to keep the bird always at the center of the gap and something similar can be achieved through a carefully designed reward scheme. Improving the model capacity will also be a possible next step to try out. The more interesting observation is that the bird starts moving in a straight line. The implementation of the game dynamics in the emulator is a little strange in a way that it doesn't let the bird to have any $y$ displacement if the flap action is taken continuously at all frames. Surprisingly, the bird has learnt to do that through just reinforcement learning.



(a) L1 loss



(b) L2 loss

Figure 5: Learning curves for two different loss functions

| | Human | DQN with L1 | DQN with L2 |
|---|---|---|---|
| Avg score | 4.25 | 2.6 | 3.3 |
| Max score | 21 | 11 | 15 |

Table 1: Average scores calculated over 20 games. Human's score corresponding to a beginner

Though the results have not surpassed human level, the model has shown some promising results and it is just a matter of fine tuning the Q-learning parameters to improve the performance as discussed earlier.

### 5.3 PRE-TRAINING

Another experiment that we tried out was to pre-train the DQN with some labeled experience data. The flappy bird game is in essence an obstacle avoidance problem that can be solved using simple search algorithms to find a collision free path if we have access to the internal states of the game. So, we setup an A* solver to play the game and it solved for a path at every frame and let the agent do the first of the series of optimal actions thus derived. The agent playing with A* algorithm never dies and this is used to generate a data set of experiences (states $s_t$) and the corresponding right action ($a_t$). This is like a data set in any classification problem. The CNN model was then trained with a simple softmax loss to classify this data right to 96% accuracy (validation). The hope was that, by beginning reinforcement learning with the model initialized to parameters of this pre-trained model, the whole training process would accelerate. But it was observed from the learning curves that the model seems to unlearn the pre-trained abilities and tries to learn afresh. This could be explained from the fact that, the A* search results in an optimal path that is different from the path reinforcement learning considers optimal. Tweaking the reward scheme could be a possible way to let reinforcement learning generate paths close to that of the A* paths but this problem is left to be further investigated in a follow-up project.

## 6 Conclusion and Future Work

We were able to successfully implement a deep reinforcement learning framework to play the game FlappyBird close to human level. Though the results did not show super-human performance, it was definitely a step in the right direction. It can be seen from the way the bird plays, it tries to reach for gaps but unfortunately crashes at the corners. A possible fix could be to somehow find a way to train the model to fit more accurately to the experience data very close to the pipe. In the current design of experience replay we sample uniformly to obtain the mini-batch and update the model. Devising a way to sample more experience points close to the danger areas would help solving this problem, better the training rate and improve convergence. With respect to the model architecture, an RNN setup could capture the temporal correlations effectively as this game involves decision making with a well informed knowledge of the previous states. Moreover, in this game we removed the background and score to reduce clutter and increase likeliness of successful training. It would be interesting to see how restoring the background affects agent's performance. Overall, our results show the capacity of Deep neural networks and how a generic reinforcement learning setup such as this could learn and play the game with very minimal domain knowledge. This has thus opened up paths for a lot of potential applications.

## References

[1] C. Clark and A. Storkey. Teaching deep convolutional neural networks to play go. *arXiv preprint arXiv:1412.3409*, 2014. 1, 2

[2] 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. 1, 2

[3] 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. 3, 5

[4] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. 1