Deep Reinforcement Learning for Flappy Bird

Kevin Chen

***Abstract*—Reinforcement learning is essential for applications where there is no single correct way to solve a problem. In this project, we show that deep reinforcement learning is very effective at learning how to play the game Flappy Bird, despite the high-dimensional sensory input. The agent is not given information about what the bird or pipes look like - it must learn these representations and directly use the input and score to develop an optimal strategy. Our agent uses a convolutional neural network to evaluate the Q-function for a variant of Q-learning, and we show that it is able to achieve super-human performance. Furthermore, we discuss difficulties and potential improvements with deep reinforcement learning.**

I. Introduction

Reinforcement learning is useful when we need an agent to perform a task, but there is no single "correct" way of completing it. For example, how would one program a robot to travel from one place to another and bring back food? It would be unrealistic to program every move and step that it must take. Instead, it should learn to make decisions under uncertainty and with very high dimensional input (such as a camera) in order to reach the end goal. This project focuses on a first step in realizing this.

The goal of the project is to learn a policy to have an agent successfully play the game Flappy Bird. Flappy Bird is a game in which the player tries to keep the bird alive for as long as possible. The bird automatically falls towards the ground by due to gravity, and if it hits the ground, it dies and the game ends. The bird must also navigate through pipes. The pipes restrict the height of the bird to be within a certain specific range as the bird passes through them. If the bird is too high or too low, it will crash into the pipe and die. Therefore, the player must time flaps/jumps properly to keep the bird alive as it passes through these obstacles. The game score is measured by how many obstacles the bird successfully passes through. Therefore, to get a high score, the player must keep the bird alive for as long as possible as it encounters the pipes.

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.

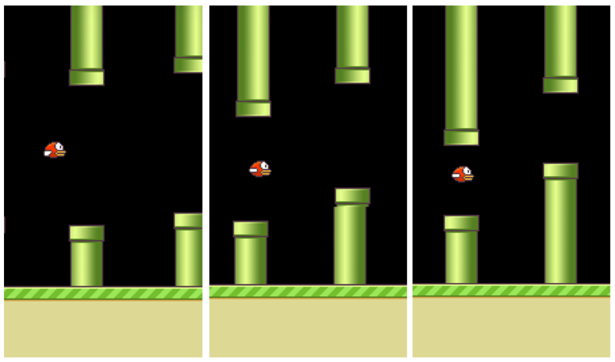

Fig. 1: Three screenshots of the game Flappy Bird at three different difficulties (easy, medium, hard) respectively.

II. Related Work

The related work in this area is primarily by Google Deepmind. Mnih et al. are able to successfully train agents to play the Atari 2600 games using deep reinforcement learning, surpassing human expert-level on multiple games [1], [2]. These works inspired this project, which is heavily modeled after their approach. They use a deep Q-network (DQN) to evaluate the Q-function for Q-learning and also use experience replay to de-correlate experiences. Their approach is essentially state-of-the-art and was the main catalyst for deep reinforcement learning, after which many papers tried to make improvements. The main strength is that they were able to train an agent despite extremely high dimensional input (pixels) and no specification about intrinsic game parameters. In fact, they are able to outperform a human expert on three out of seven Atari 2600 games. However, further improvements involve prioritizing experience replay, more efficient training, and better stability when training. [2] tried to address the stability issues by clipping the loss to $+1$ or -1, and by updating the target network once in every C updates to the DQN rather than updating the target network every iteration.

III. Method

In this section, we describe how the model is parameterized and the general algorithm.

A. MDP Formulation

The **actions** that the agent can take are to flap ($a = 1$) or to do nothing and let the bird drop ($a = 0$). The **state** is represented by a sequence of frames from the Flappy Bird game as well as the recent actions that the player took. Specifically, the state is the sequence shown in

$$s_t = (x_{t-histLen+1}, a_{t-histLen+1}, ..., x_{t-1}, a_{t-1}, x_t) \tag{1}$$

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\varepsilon} \left[(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i))\nabla_{\theta_i} Q(s, a; \theta_i)\right] \tag{2}$$

Equation 1 where s_t is the state at time t, x_t is the pixel input (or the frame or screen capture) at time t, and a_t is the action taken at time t. $historyLength$ (or $histLen$) is a hyperparameter that specifies how many of the most recent frames to keep track of. This is to reduce the storage and state space compared to saving all frames and actions starting from $t = 1$. The reason for storing multiple x's and a's rather than storing a single frame x is because the agent needs temporal information to play. For example, the agent cannot deduce the velocity of the bird from a single frame, but velocity is essential for making a decision.

The **discount factor** was set to $\gamma = 0.95$. The **transition probabilities** and the **rewards** are unknown to the agent. Since Q-learning is model-free, we do not explicitly estimate the transition probabilities and rewards, but instead directly try to estimate the optimal Q-function. This is described further in the Q-learning section.

However, we still must define the rewards intrinsic to the game. Ideally, the reward should essentially be the score of the game. It starts out as 0 and every time the bird passes a pipe, the score increases by 1. However, this is potentially problematic in that the rewards will be very sparse. Specifically, if the bird dies instantly at the start of the game, the reward would be similar to if the bird died right before reaching the pipe. The performance is clearly better if the bird survives up until the pipe compared to dying instantly. Therefore, adding a reward for staying alive encourages the agent to think similarly. Without this additional reward, the agent should eventually realize this, but adding the reward, called $rewardAlive$, speeds up the training process. In total, we have three rewards: $rewardAlive$, $rewardPipe$, and $rewardDead$. The agent gets $rewardAlive$ for every frame it stays alive, $rewardPipe$ for successfully passing a pipe, and $rewardDead$ for dying.

B. Q-learning

The goal in reinforcement learning is always to maximize the expected value of the total payoff (or expected return). In Q-learning, which is off-policy, we use the Bellman equation as an iterative update

$$Q_{i+1}(s, a) = \mathbb{E}_{s'\sim\varepsilon}[r + \gamma \max_{a'} Q_i(s', a')|s, a] \tag{3}$$

where s' is the next state, r is the reward, ε is the environment, and $Q_i(s, a)$ is the Q-function at the ith iteration. It can be shown that this iterative update converges to the optimal Q-function (the Q-function associated with the optimal policy). However, this is rote learning. To prevent rote learning, function approximations are used for the Q-function to allow generalization to unseen states. Our approach uses the deep Q-learning approach in which we use a neural network to approximate the Q-function. This neural network is a convolutional neural network which we call the Deep Q-Network (DQN).

A common loss used for training a Q-function approximator is

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)} \left[\frac{1}{2}(y_i - Q(s, a; \theta_i))^2\right] \tag{4}$$

where θ_i are the parameters of the Q-network at iteration i and y_i is the target at iteration i. The target y_i is defined as

$$y_i = \mathbb{E}_{s'\sim\varepsilon} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})|s, a\right] \tag{5}$$

for a given experience $e = (s, a, r, s')$. An experience is analogous to a datapoint such as in linear regression and the replay memory, a list of experiences, is analogous to a dataset such as in linear regression. The gradient of the loss function with respect to the weights is shown in equation 2. Thus, we can simply use stochastic gradient descent and backpropagation on the above loss function to update the weights of the network.

Additionally, we take an ϵ-greedy approach to handle the exploration-exploitation problem in Q-learning. That is, when we are training, we select a random action with probability ϵ and choose the optimal action $a_{opt} = \arg\max_{a'} Q(s, a')$. In our implementation, we linearly change the exploration probability ϵ from 1 to 0.1 as the agent trains. This is to encourage a lot of exploration in the beginning where the agent has no idea how to play the game and the state space is extremely large. It takes a large number of random actions and as it starts to figure out which actions are better in different situations/states, it exploits more and tries to narrow down what the optimal actions are.

C. Experience replay

A problem that arises in traditional Q-learning is that the experiences from consecutive frames of the same episode (a run from start to finish of a single game) are very correlated. This hinders the training process and leads to inefficient training. Therefore, to de-correlate these experiences, we use *experience replay*. In experience replay, we store an experience (s, a, r, s') at every frame into the replay memory. The replay memory has a certain size and contains the most recent $replayMemorySize$ experiences. It is constantly updated (like a queue) so that they are associated with the actions taken with the recent Q-functions. The batch used to update the DQN is composed by uniformly sampling experiences from the replay memory. As a result, our experiences are no longer likely to be correlated.

D. Stability

Moreover, to encourage more stability in decreasing the loss function, we use a target network $\hat{Q}(s, a)$. $\hat{Q}(s, a)$ is essentially the the same as $Q(s, a)$. The network has the same structure, but the parameters may be different. At every C updates to the DQN $Q(s, a)$, we update $\hat{Q}(s, a)$. This $\hat{Q}(s, a)$ is then used for computing the target y_i according to:

$$y_i = \mathbb{E}_{s' \sim \varepsilon}\left[r + \gamma \max_{a'} \hat{Q}(s', a'; \hat{\theta}_{i-1}) | s, a\right] \quad (6)$$

This leads to better stability when updating the DQN.

E. Pre-processing

Since we use a very high dimensional state, we actually perform pre-processing to reduce the dimensionality and state space. The pre-processing is done over the pixels, so we first extract the images from the state s_t. The original screen size is 512×288 pixels in three channels, but we convert the image captured from the screen to grayscale, crop it to 340×288 pixels, and downsample it by a factor of 0.3, resulting in a 102×86 pixel image. It is then rescaled to 84×84 pixels and normalized from [0, 255] to [0, 1]. I call this feature extractor $\phi(s)$.

F. Deep Q-Network

Our Q-function is approximated by a convolutional neural network. This network takes as input a $84 \times 84 \times historyLength$ image and has a single output for every possible action. The first layer is a convolution layer with 32 filters of size 8×8 with stride 4, followed by a rectified nonlinearity. The second layer is also a convolution layer of 64 filters of size 4×4 with stride 2, followed by another rectified linear unit. The third convolution layer has 64 filters of size 3×3 with stride 1 followed by a rectified linear unit. Following that is a fully connected layer with 512 outputs, and then the output layer (also fully connected) with a single output for each action. To choose the best action, we take the action with the highest output Q-value ($a_{opt} = \arg\max_{a'} Q(s, a')$).

G. Pipeline

Algorithm 1: Deep Q-learning algorithm for Flappy Bird

```
initialize replay memory
initialize DQN to random weights
repeat
    new episode (new game)
    initialize state $s_0$
    repeat
        extract $x_t$ from raw pixel data update state $s_t$ with $x_t$
        add experience $e_t = (\phi(s_{t-1}), a_{t-1}, r_{t-1}, \phi(s_t))$ to replay memory
        take best action $a_t = \arg\min_{a \in actions} Q(s_t, a)$ with exploration if training
        uniformly sample a batch of experiences from the replay memory
        backpropagate and update DQN with the minibatch
        update exploration probability $\epsilon$
        if C updates to DQN since last update to target network then
            update the target Q-network $\hat{Q}(s, a) \leftarrow Q(s, a)$
        end
        update state $s_t$ with $a_t$
        update current reward $r_t$ and total reward $totalReward$
        update game parameters (bird position, etc.)
        refresh screen
    until flappy bird crashes;
    restart Flappy Bird
until convergence or number of iterations reached;
```

The pipeline for the entire DQN training process is shown in **Algorithm 1**. It is as previously described earlier in this section. We apply Q-learning but use experience replay, storing every experience in the replay memory at every frame. When we perform an update to the DQN, we sample uniformly to get a batch of experiences and use that to update the DQN. This is analogous to sampling batches from a dataset using SGD/mini-batch gradient descent in convolutional neural networks for image classification or deep learning in general. Then we update the exploration probability as well as the target network $\hat{Q}(s, a)$ if necessary.

IV. Results

A video can be found at the following link: https://youtu.be/9WKBzTUsPKc. Our metric for evaluating the performance of the DQN is the game score (numper of pipes passed). The reported scores in the tables are the average scores over 10 games (unless otherwise specified).

A. Testing parameters

The Flappy Bird game was run at 30 frames per second, and $historyLength$ was set to 5. The discount factor was 0.95 and the rewards were the following: $rewardAlive = +0.1$, $rewardPipe = +1.0$, $rewardDead = -1.0$. The exploration probability ϵ decreased from 1 to 0.1 over 600000 updates to the DQN. The size of the replay memory was 20000 experiences.

For training, we used RMSProp with a learning rate of 1e-6, decay of 0.9, and momentum as 0.95. These were chosen similarly to that of [2]. To figure out better parameters, some were done by trial and error. For example, we noticed that the learning rate was too high when the neural network weights began exploding, and used a binary search algorithm to figure out the best learning rate. If the learning rate was too low, it would take longer to train. We did updates in mini-batches of size 32 (experiences). We only begin training after the replay memory has at least 3000 experiences and update the target network $\hat{Q}(s, a)$ once for every 1000 updates to the DQN. Our convolution weights are initialized to have a normal distribution with mean 0 and variance 0.1. This deep neural network was implemented using TensorFlow.

B. Overall performance

The trained DQN plays extremely well and even performs better than humans. We compare the results of the DQN with a baseline and humans. The baseline implementation flaps every z frames to keep the bird in the middle of the screen. This baseline was chosen because the pipe gaps locations are uniformly distributed with the expected location to be in the middle of the screen. These comparisons are shown in table I (average score) and table II (highest score).

training difficulty	**flap every n**	**human**	**DQN**
easy	Inf	Inf	Inf
medium	Inf	Inf	Inf
hard	0.5	17.625	82.2

TABLE I: Average score of DQN on varying difficulties compared to baseline and human performance

training difficulty	**flap every n**	**human**	**DQN**
easy	Inf	Inf	Inf
medium	11	Inf	Inf
hard	1	65	215

TABLE II: Highest score of DQN on varying difficulties compared to baseline and human scores

The performance of the DQN is much higher than the baseline and human performance. If the score was higher than 1000, the score was considered to be infinity (except for the human case where if they got a score above 100, this would be considered infinity). The human case was generalized to be infinity if the user could play for forever if he or she could focus and did not need to take breaks (eat, sleep, etc.). Although the scores for human and DQN are both infinity for the easy and medium difficulties, in reality the DQN is better because it does not have to take a break whereas the DQN can play for 10+ hours at a time.

In general, almost all of the failures in the hard difficulty are because the bird flaps upwards when it should be letting the bird drop, and then it dies. However, once in a while, the bird will just barely clip the top-right corner of the lower pipe as it is falling. Furthermore, I noticed that the agent seems to take riskier moves when it trains more. Thus, a follow-up test to resolve these problems could be to encourage the agent to take the moves with the lowest risk. To do this, we would have the agent make a random move a small probability of the time during training (even if the agent is supposed to be evaluating the optimal action). To maximize the expected return, the agent would have to play very safely.

C. Training time

In this section, we discuss how the number of training iterations affects the performance of the Flappy Bird agent. The number of training iterations refers to the number of updates to the DQN (there is no exact definition of epoch here). Our results (see Table III) show that more training does not necessarily lead to better scores. In fact there is some instability and the scores fluctuate

training iterations	easy	medium	hard
99000	1680.9	52.7	0.3
199000	1026.8	101.8	11.6
299000	351.06	42.7	65
399000	1006.11	2598	71.6

TABLE III: Average score of DQN as a function of learning rate

Game difficulty	DQN (easy)	DQN (medium)	DQN (hard)
Easy	Inf	Inf	Inf
Medium	0.7	Inf	Inf
Hard	0.1	0.6	82.2

TABLE IV: Performance of DQN on medium difficulty with weights initialized from DQN trained on easy

Game difficulty	DQN (easy)	DQN (medium)	DQN (hard)
Easy	Inf	99.3	1.5
Medium	1.11	Inf	1.5
Hard	0.0	0.6	71.6

TABLE V: Performance of DQN after training on the tested difficulty but initialized to random weights

# iterations	easy	w/ rewardAlive	medium	w/ rewardAlive
99000	28.1	1680.9	2.6	52.7
199000	128.4	1026.8	128.6	101.8
299000	617.11	351.06	58.7	42.7
399000	282.3	1006.11	299.1	2598

TABLE VI: Comparison of training with additional $rewardAlive$ and without it

with more training after a certain point. For example, the hard difficulty had not reached this point of training and consistently yields better results with more training. This instability is inherent to many reinforcement learning algorithms and could be further investigated in a follow-up project. One potential solution would be to decrease the learning rate as more training occurs or to increase model complexity (neural network architecture).

D. *Training with initial pre-trained network*

Here, we describe results of a network which is initialized to another pre-trained network. Specifically, when training the network to play on the medium difficulty, we initialize the DQN to have the same weights of a network that was previously trained on the easy difficulty. This yielded the best results performance-wise compared to any of our other trained networks. The network was trained on the medium difficulty for 209,000 updates after being initialized to the previously trained DQN on easy mode.

From Table IV, it is clear that not only does the DQN perform better on the difficulty it was trained on, but it also performs better on the easier difficulties. It remembers how to perform well on the easy mode while it modifies it weights to also perform well on the medium difficulty. The same could not be said about the networks which were directly trained on the easy/medium/hard difficulties (as shown in Table V), which is a very insightful observation.

After training for 199 iterations on each difficulty (directly) with random initialized weights, we got the results shown in Table V. These networks do not generalize well to different difficulties.

E. *Removing the reward for staying alive*

In this section, we test whether the $rewardAlive$ reward truly leads to faster convergence or better results as suspected. The results in Table VI show that indeed adding a $rewardAlive$ reward accelerates the training process since it provides an incentive which is directly correlated to the score/goal. More importantly, it prevents sparse rewards to encourage faster learning. Therefore, if a reward is directly correlated with the intended reward (such as score of the game), then it is beneficial to use this correlated reward in addition to the intended reward to speed up the training process.

V. Conclusion

We were able to successfully play the game Flappy Bird by learning straight from the pixels and the score, achieving super-human results. However, training was not consistent in that more training did not necessarily correlate with better score. The model could be overfitting or forgetting so future work could attempt to explore and resolve this issue. Another very important area that could be refined is the experience replay. We uniformly sampled from the replay memory, but some experiences have more impact on successfully training the DQN than other experiences. Being able to prioritize these experiences would lead to better performance, efficient training, and faster convergence. 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 performance. Overall, our results show that deep reinforcement learning is a step in the right direction and has a lot of potential for further application.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with Deep Reinforcement Learning. In *Deep Learning, Neural Information Processing Systems Workshop*, 2013.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattle, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529-533, 2015.

[3] T. Schaul, J. Quan, I. Antonoglou, D. Silver. Prioritized Experience Replay. *arXiv: http://arxiv.org/abs/1511.05952*