A Deep RL-based Agent for Bug on a Wire

Nihal Singh
Georgia Institute of Technology
Atlanta, USA
nihal111@gatech.edu

Roshan Pati
Georgia Institute of Technology
Atlanta, USA
roshanpati@gatech.edu

Arindum Roy
Georgia Institute of Technology
Atlanta, USA
arindumroy@gatech.edu

Monica Gupta
Georgia Institute of Technology
Atlanta, USA
mgupta334@gatech.edu

Abstract

Reinforcement learning (RL) provides a natural human-like approach wherein positive and negative cues guide an RL agent's behavior. Specifically, these cues are used to train action-value function approximators using which the agent attempts to achieve a high score. In this project, we apply the Reinforcement Learning framework to beat the popular miniclip game- Bug on a Wire to achieve a score comparable to humans. Our agent is based on Deep-Q Networks (DQN), an approach commonly employed in designing RL agents for Atari 2600 games. Further, we show that our setup can be easily configured to other simple environments with minimal modifications. We show this by extending our setup to another popular game- Flappy Bird.
`Our code repository can be found here`[1].

1. Introduction

1.1. Objective

In the course of this project we developed a general purpose deep Reinforcement Learning agent setup, that can be trained to play a host of flash games. We tested our model on the popular miniclip game, Bug on a Wire. Our setup performs the requisite environment interfacing to handle the pre-processing pipeline from raw screenshot images as input to the network. This way, our agent is able to train itself directly using sensory inputs, in our case, images. We test this by extending our environment+agent setup with minimal changes to play the game Flappy Bird.

Bug on a Wire is an endless runner game that is procedurally-generated. The objective of the game is to move the bug using keyboard inputs to help make its way past crows and survive for the longest duration possible without being eaten. The bug is allowed dodging movement on four wires, by moving LEFT, RIGHT and JUMPING. Getting too close to a crow results in the bug being eaten and a subsequent GAME-OVER screen.

[1] https://github.com/nihal111/bugOnAWireRL

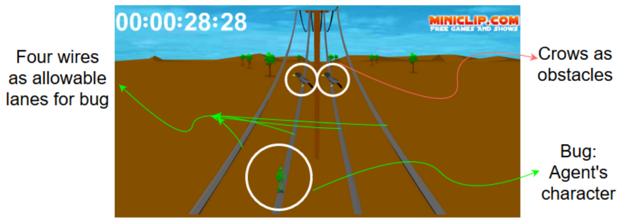

Figure 1. An illustration highlighting the various facets of the game- Bug on a Wire

Reinforcement learning becomes imperative when it comes playing games, as rule-based agents cannot be modeled to make decisions when there is uncertainty involved. The adaptive behaviour displayed by the agent is learnt over successive iterations of game play, incorporating positive and negative rewards attached with each iteration.

In our project, we explore two different strategies, namely DQN [6] and DQfD [5], to train our agent. Initially we applied an approach which uses an epsilon-greedy DQN based agent. DQN requires a huge training time to achieve good performance, so we explored DQfD, where the agent additionally learns from a set of human-performed demonstrations, to quickly learn optimal moves. We perform feature engineering, to generate the state from the raw frames obtained from the flash game. We then

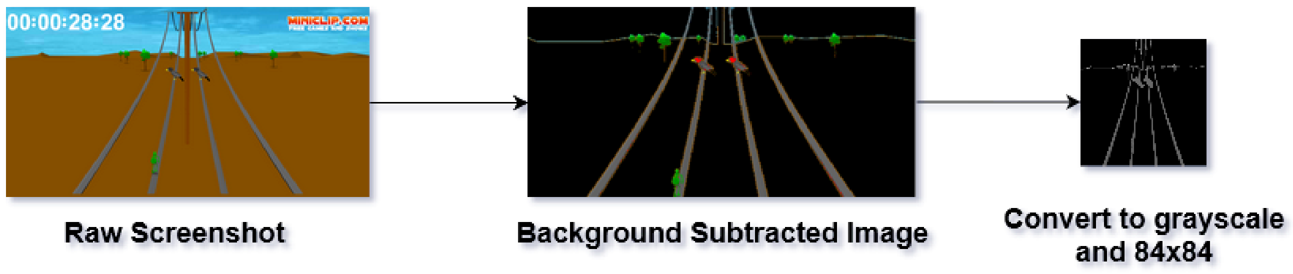

Figure 2. Schematic of the pre-processing pipeline for Bug on a Wire. First removes background from raw RGB screenshot and then converts to Grayscale and resizes to 84x84.

experiment with different reward assignments and state parameterizations to bring the model to convergence in a time efficient fashion. Training a reinforcement learning model can prove to be a difficult task as the network is prone to getting stuck at local optima.

1.2. Literature Survey

In the recent past, (DQN) [6], (OpenAI gym) [3] have been successful in coming up with agents for playing a variety of Atari 2600 games using deep reinforcement learning, surpassing human expert-level on multiple games. The deep neural networks act as function approximators to represent the Q-value (action-value) in Q-learning. They also introduce methods to stabilize and bring the network to quicker convergence by using ”experience replay”, that facilitates learning from previously recorded experiences from which batches are randomly sampled to update the network so as to un-correlate experiences and delay updates for the target network, improving stability.

However, their setup is specifically built only to handle a small set of game environments. Furthermore, it has been shown [5] that vanilla DQN setup suffers from very slow training, requiring the agent to play a very large number of game episodes before the model learns anything meaningful.

Setups like DQfD [5] can effectively boost performance, pushing the network out of local minima, by training on exemplar human generated (or otherwise rule-based agent generated) episodes. However, this approach has only been tested for a small set of 11 game environments.

Every environment is different, in that, approaches that work [1] for simple games like Flappy Birds (essentially a 1-D game, since bird only moves vertically along a fixed x), which have a small action space (only two) may not work for more complex games.

1.3. Impact

The novelty and hence the impact of our project derives from the fact that there has been no work done on Miniclip games like the environment of Bug on a Wire. Not only does our setup play this particular game, but most of the environment setup is reusable and can be easily applied to play any flash game with minor modifications in the set of permitted actions and rewards. We demonstrate this by extending our setup to another game - Flappy bird.

1.4. Data

For the purpose of collecting training data, we capture frames of manually demonstrated as well as live game play as the agent learns to play the games. Each input is a set of 4 grey-scale images of size 84x84, which is fed to the network to learn the optimal policy.

2. Approach

We apply the known method of DQN that has been tested on arcade-game environments like Atari 2600. We build on this approach by adding augmentations like DQfD based exemplar samples and prioritized experience replay to boost our performance. While these approaches have been proven to show significant results for a small set of environments, we attempt to integrate our new environment of Bug on a Wire. We find that deep reinforcement learning on any new environment is a difficult task. We experiment with different techniques, combining different approaches to boost our scores. Our experiments and results are highlighted in Section 3.

2.1. Environment

We first set up an environment to interact with the game. We sample frames at a rate of 0.17 seconds per frame, which is the fastest we could achieve taking into account the time taken by the screen-grabbing tool used.

2.1.1 State Representation

We sample 4 contiguous frames of size $84 * 84$ which are passed into a mask filter that assigns the pixel values of the pixels with non-relevant images (e.g. sky, mountains, background etc.) to zero as to produce the state representation. The game_over state is assigned None and is detected by simply parsing a patch of pixel values that appear after the game gets over. The pre-processing pipeline we use for generating the state representation is illustrated in Figure 2.

2.1.2 Actions

Our action set consists of 4 actions namely *UP*, *LEFT*, *RIGHT* and *NO ACTION*.

For taking actions we use *pymouse* and *pykeyboard* packages to bind keyboard and mouse presses to interact with the game.

2.1.3 Rewards

For credit assignment, since the aim of the game is to survive as long as possible we assign rewards in a manner that imitates the cumulative reward of the actual game i.e. the total running time which is linear in the number of frames.

We experiment with several credit-assignment schema. Our first schema used only a 0 / -1 reward assignment, with -1 corresponding to a transition that ends the game and 0 corresponding to all other transitions. We realized that the reward signals from this schema are too sparse, making it difficult for the agent to learn.

The reward assignment we converge to is expressed as follows

$$reward(s,a) = \begin{cases} -5, & \text{if } \textit{is\_game\_over} \\ \mathrm{N}_c - A_p + 0.1, & \text{otherwise} \end{cases}$$

where N_c is *num_crows_crossed*, which is calculated by parsing image patches of the 4 wires where the bug is not present. This is used to incentivize the model with the idea of avoiding crows which is the primary motive of the game. A_p is *action_penalty*, that occurs every time the agent tries to jump left from the left most rope or jumps right for the right most rope i.e. for every action having no consequence.

2.2. Deep Q-Network

A deep Q network (DQN) is a multi-layered neural network that for a given state s outputs a vector of action values $Q(s, \cdot; \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ are the parameters of the network. For an state space of n-dimensions and an action space containing m actions, the neural network is a function from $\mathbb{R}^n$ to $\mathbb{R}^m$. Two crucial aspects of the DQN algorithm as proposed by [6] are the use of a target network, and the use of experience replay. The target network, with parameters $\boldsymbol{\theta}^-$, is the same as the online network except that its parameters are copied every τ steps from the online network, so that then $\boldsymbol{\theta}_t^- = \boldsymbol{\theta}_t$, and kept fixed on all other steps. The target used by DQN is then

$$Y_t^{\mathrm{DQN}} \equiv R_{t+1} + \gamma \max Q\left(S_{t+1}, a; \boldsymbol{\theta}_t^-\right)$$

Our DQN architecture was inspired from [6] consisting of 3 convolution layers each followed by batch normalization and two fully connected layers for the classifier. We did not use any pooling layers in the network since pooling incorporates the translation invariant property of an image to an extent that we would like to avoid since exact entity (bug, crow) positions are crucial. We applied ReLU activation on top of all but the output layer. We use the same network for both the policy net and target net as well as at the end of every episode wherein we copy the policy net into the target net. The code repository referred for DQN implementation can be found here (Official PyTorch tutorials) [8].

2.3. Experience Replay

For experience replay [9], observed transitions are stored for some time and sampled uniformly from this memory bank to update the network. The memory bank is a fixed queue which keeps getting over-written with new experiences observed from the environment.

Experience transitions are usually uniformly sampled from a replay memory. However, this approach simply replays transitions at the same frequency that they were originally experienced, regardless of their significance. Schaul et. al [10] develop a framework for prioritizing experience, so as to replay important transitions more frequently, and therefore learn more efficiently. They used prioritized experience replay in Deep Q-Networks (DQN) by assigning a priority proportional to the TD error as given by the difference between the policy and the target net, to achieve human-level performance across several Atari games.

The primary intuition behind our approach is the fact that since the inception DQN has been applied to many games and there have been empirical evidence suggested by the work of [7] of the fact that pure RL tends to work better for reflex based games rather than memory/logic based game and since our objective was to design an RL framework that can be plugged for any flash game with the game screens as state inputs DQN seemed like a good starting point for us. The novelty lies in the fact that our platform is robust enough to be quickly plugged

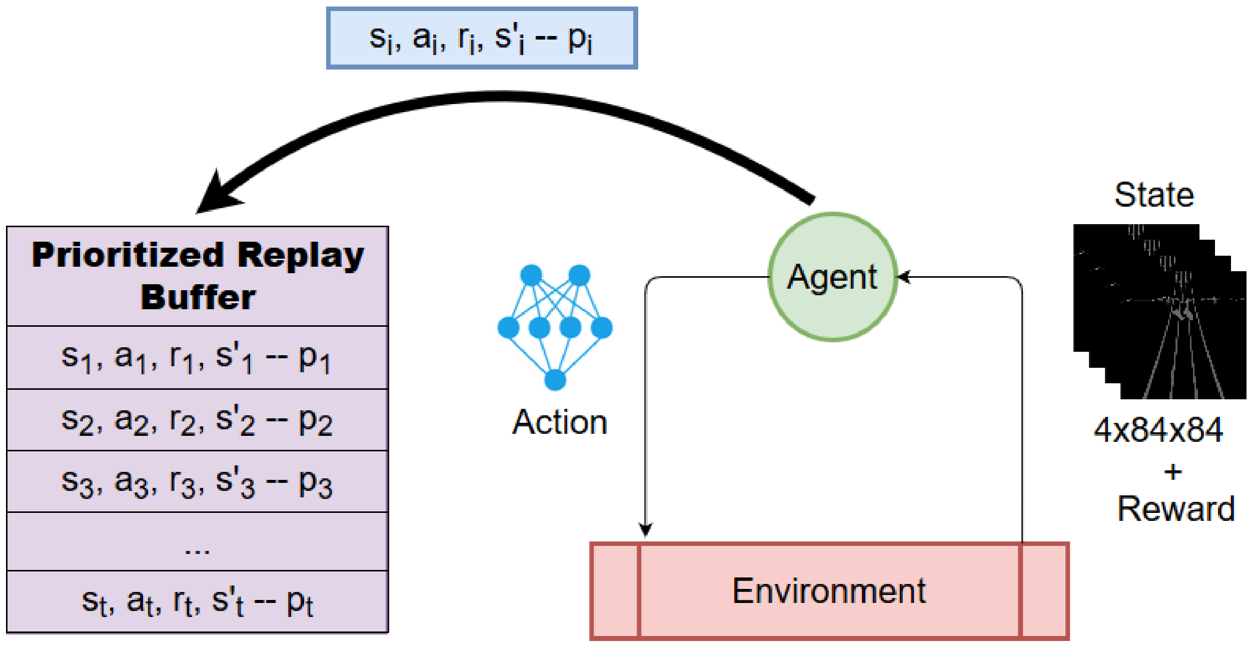
s_i, a_i, r_i, s'_i -- p_i
Prioritized Replay Buffer
s_1, a_1, r_1, s'_1 -- p_1
s_2, a_2, r_2, s'_2 -- p_2
s_3, a_3, r_3, s'_3 -- p_3
...
s_t, a_t, r_t, s'_t -- p_t
Action
Agent
State
4x84x84
+
Reward
Environment

Figure 3. Agent-environment interaction and storing transition experiences in prioritized replay buffer.

into a different game environment and start learning from it.

Building on this prior work, we experiment with both techniques to analyze how it affects training for the agent in our new environment. This is illustrated in Figure 3.

2.4. Deep Q-Learning from Demonstrations (DQfD)

Recently, [5] proposed Deep Q-learning from Demonstrations (DQfD), that leverages small sets of demonstration data to massively accelerate the learning process even from relatively small amounts of demonstration data. For the purpose of our project, we manually played the game until a sizeable replay buffer is saved which we call the *DQfD memory*. In addition, we added a *DQN memory* sampled from live game play during training which is updated each time we collect enough samples to fill up the memory. The idea behind this is to let the model explore in the state-action space and learn from its mistakes instead of only over-fitting on the DQfD memory.

3. Experiments and Results

The aim of our task is for the agent to survive in the game as long as possible to create a high score. To this end, we experiment with several techniques, incorporating different state parameterization and creating new reward schema to induce our agent to derive meaningful signals from the environment to facilitate its learning. We improve our performance by training on both the DQN and DQfD memory. To measure our progress, we plot the variation of the episode duration with training time, also evaluating 5 times every k episodes to get the average evaluation episode duration. The experiments can be broadly classified into following 3 categories.

3.1. Feature Engineering Experiments

1. Using raw RGB image directly with only resizing (3x84x84) and no pre-processing and throwing a deep neural network at it to learn the requisite information.

2. Pre-process the raw RGB image to remove the background, retaining only the wires, the bug and the crows as non-zero values in the image (3x84x84).

3. Using 4 contiguous frames as the state representation and changing the pre-processed image from 2) to grayscale (4x84x84).

Approach 1) of sampling the raw image directly did not work at all as anticipated. With a constantly moving target to chase, the model does not learn anything meaningful and it is difficult to train the CNN layer on a continuously moving image with lot of moving background, increasing the state space significantly.

Approach 2) yielded slightly better results, but not very satisfactory. Since the speed of the game (correlated with the difficulty) increases as time passes, we hypothesized

Hyper-parameter	Value
batch size	512
train episodes	5000
optimizer	ADAM
learning rate	$1e-4$
γ	0.99
eps_start	0.1
eps_end	0.05
eps_decay	500
optimizer	ADAM

Table 1. Hyper-parameter settings

that the agent can learn the positions of the bug and the crows from the state, but it fails to take into consideration the speed of approaching crows and also the perspective change of the positions of the crows (as the bird trajectories are not strictly vertical) which can be crucial information when learning how to time the evading actions.

Hence, to incorporate speed and perspective change we concatenated the last 4 frames sampled from the environment and resized it to get the input state representation of 4x84x84 as explained in approach 3).

3.2. Model Experiments

To utilize both the DQN and DQfD replay buffers we trained using both alternatively i.e. after each episode the policy net was updated based on the gradients with respect to both the DQN and DQfD loss. The network architecture was limited to a 3 layer CNN with ReLU activation because of training time constraints. The best hyper-parameters tuned for the model are enlisted in Table 1.

3.3. Reward Assignment Experiments

Ideally while assigning rewards, given a particular action a and next state $s^{'}$ we'd like to aim the credit assignment mechanism of the game which can basically be summarized as the amount of time survived. We experiment with the enlisted reward schema and analyze their results.

1. Reward of -1 for GAME_OVER, 0 otherwise.
2. Added reward N_c = number of crows the bug passes.
3. Deducted reward corresponding to A_p = action penalty.
4. Added 0.1 reward for every non terminal transition.

Initially with approach 1), we assigned reward 0 for non-final states and -1 for final states. To fill up the glaring lacuna of sparse reward signals we added intermediary rewards which is equivalent to the number of crows avoided in the next state(N_c) as described in 2).

Model	Avg Eval Durations (frames)
single frame raw RGB states + binary reward assignment	143
single frame raw RGB states + intermediate reward assignment	117
4 frame preprocessed grayscale states + binary reward assignment	231
4 frame preprocessed grayscale states + intermediate reward assignment with N_c and A_p	264

Table 2. Results Comparison on BugOnAWire

Further, to introduce the environment constraint of disallowed actions (i.e. attempting to move beyond the fringes– inconsequential actions) we introduced an Action Penalty, A_p for such actions as explained in 3).

Finally, to avoid vanishing gradients because of 0 reward we changed it to 0.1 for every non-final next state along with N_c and A_p.

Our final reward schema that we converged on is expressed in Section 2.1.3.

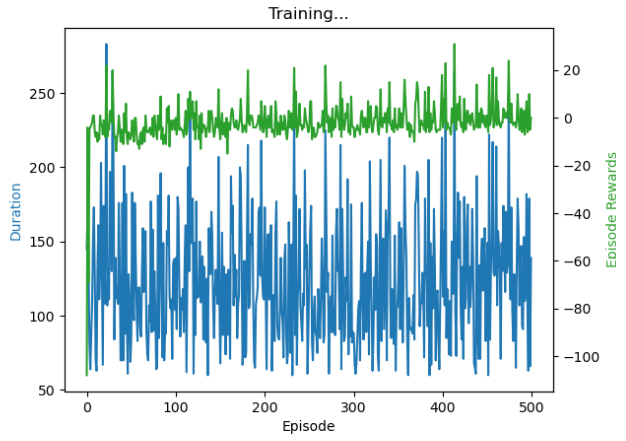

Figure 4. Plot of average episode durations and total episode rewards with training episodes. Results from the last 500 episodes of training, it can be seen that the agent often gets stuck in a local minima.

Student Name	**Contributed Aspects**	**Details**
Nihal Kumar Singh	Environment interfacing, Feature engineering, DQN experiments	Interfacing the environment for Bug on a Wire, built pre-processing pipeline. Ran DQN experiments.
Monica Gupta	Environment interfacing, DQfD experiments	Ran experiments for DQfD and dealt with environment interfacing for Flappy Bird.
Arindum Roy	Reward Assignment, DQfD implementation, Result analysis	Experimented with different reward assignment schema for faster convergence and DQfD implementation to record human played games.
Roshan Pati	DQN experiments, hyperparam tuning, Prioritized Experience Replay	Replicated DQN and implemented prioritized experience replay buffer. Extensive hyperparameter tuning.

Table 3. Work distribution

3.4. Analyzing Results

3.4.1 Kamikaze Bug

We observe that the agent learns to extract information about the approaching crows but only partly. By observing the bug's action, it is clear that the bug reliably performs an evading action when the crow approaches a certain close-by distance. However, the bug does not learn complete information about which wire the approaching crow is on. This is observed as often times the bug executes an action that lands it into the mouth of a bug and this effectively leads to game-over. We hypothesize that the reason for this behavior can be attributed to the 3-D perspective of the game, where the approaching crows seem to converge from the same central point on the screen and diverge only upon closing up on the bug. This makes it difficult for the agent to correlate its position (position of the bug) along with the position of the approaching crows, hence often leading to suicidal tendencies for the agent.

3.4.2 In Bellman we trust

On the basis of the Bellman Fixed Point theorem, from any starting point, Q-value iteration or dynamic programming approaches can be said to converge to a unique fixed point in the Bellman space. However, when applying a deep neural network based function approximator that handles the state representation, there is a chance of getting stuck into local optima. In particular, Fan et. al [11] have shown properties about the convergence of the statistical error from approximating the action-value function using deep neural network, while the algorithmic error converges to zero at a geometric rate.

We believe that a combination of DQfD and DQN with an intelligent reward assignment schema and ample training time can be applied to a simple environment to create an RL agent that learns to navigate well.

To summarize, our task suffers from similar challenges as that of pure RL, like Bellman convergence, sparse rewards, visual information extraction (perspective fluctuation, game tempo variation etc.) and we discussed several intuitive approaches above to abate these limitations to some extent.

3.4.3 Multi-task objectives

In addition to just assigning the reward schema to induce the model to learn desired behaviors, another technique that can be used is to use a multi-task objective that forces the module to learn features from the image that we are interested in. For example, it can be easily checked which wire the bug occupies by checking the respective region of interest. With this it is easy to obtain a 4-dimensional one-hot vector that corresponds to the bug's position, *original_position*. The fully-connected layers that act after the CNN can reduce the state space to a 4 dimensional vector, *predicted_position*. Another component can then be added to the loss term corresponding to the KL-Divergence between *original_position* and *predicted_position*. This would induce the model to quicker learn features from the raw image that we are interested in, and that are crucial towards the game. Similarly, we can apply the same technique to detecting crow positions. However, we do not completely experiment with this idea due to paucity of time.

4. Extension to other environments- Flappy Bird

We extend our setup to the Flappy Bird environment with minimal modifications. Our pre-processing effectively remains the same, as illustrated in Figure 4. As can be observed from Figure 6 which is referred from [4] duration running average shoots up at around iteration number 2000 is almost constant roughly till episode 6000 which is the period when the model has learnt to get past the first pipe. In our training the maximum score the model could reach

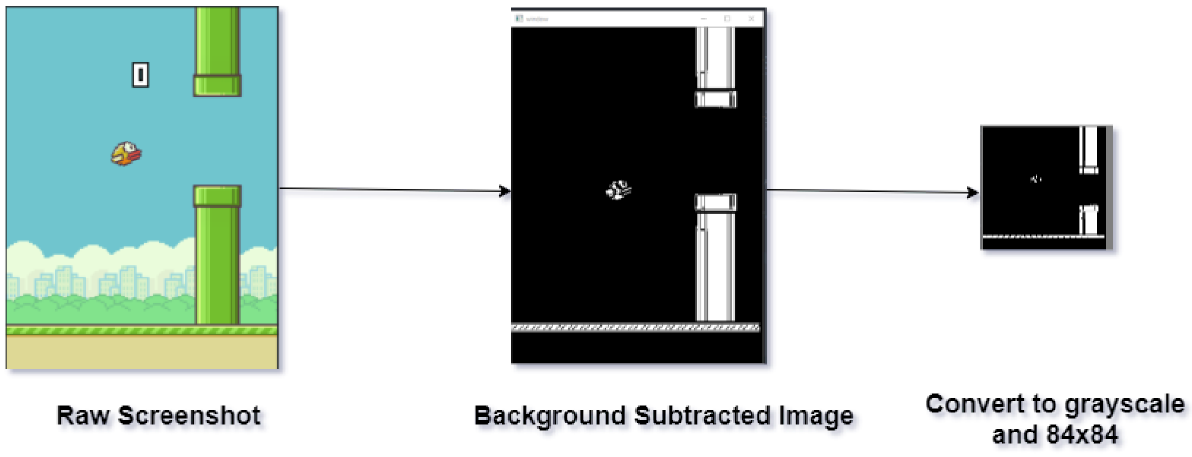

Figure 5. Schematic of the pre-processing pipeline for Flappy Bird. First removes background from raw RGB screenshot and then converts to Grayscale and resizes to 84x84.

during training is 3 in the first 500 iterations. By drawing a parallel between both we can see that our model not only tends to follow the learning curve of the referred model, but also converges faster. Due to time limitations we weren't able to test for more iterations which can be addressed in future course of actions.

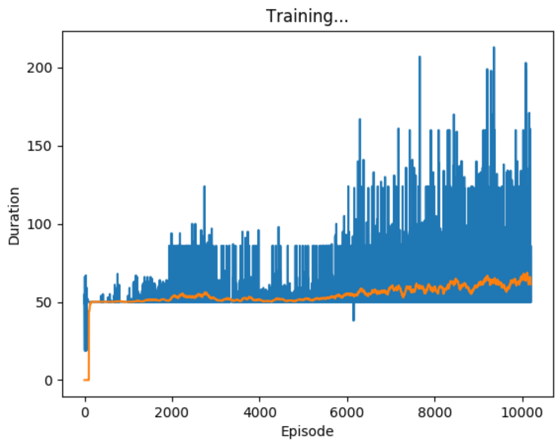

Figure 6. Training plot on a typical Flappy Bird environment.

5. Conclusion

In this project we extend the established DQN network and demonstrated its ability to master difficult control policies for the game "Bug on a wire" using only raw pixel based input coming directly from a screen capture. We show that our setup can be easily repurposed to fit a new environment with minimal modifications, by extending to another game- Flappy Bird. We experiment with a bunch of different techniques like DQN, DQfD, Prioritized Experience Replay and several feature engineering and reward-assignment schema. We obtain decent performance from our agent on the Bug on a Wire environment, getting a best eval score of average episode duration as 45 seconds. Given the difficulty of the game, we find this as an acceptable outcome.

A lot of the biggest challenges in RL revolve around how we interact with the environment effectively (e.g. exploration vs. exploitation, sample efficiency), and how we learn from experience effectively (e.g. long-term credit assignment, sparse reward signals). Furthermore, we can use other techniques like A3C [2] and expand on multi-task objectives as explained in Section 3.4.3 to make learning more instructive.

6. Work Division

For this project, we collaborated extensively online and our work distribution is summarized in Table 3.

References

[1] Naveen Appiah and Sagar Vare. Playing flappybird with deep reinforcement learning.

[2] Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz. Reinforcement learning through asynchronous advantage actor-critic on a gpu. *arXiv preprint arXiv:1611.06256*, 2016.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[4] hardlyrichie/pytorch-flappy bird. Github: https://github.com/hardlyrichie/pytorch-flappy-bird.

[5] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, et al. Deep q-learning from demonstrations. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[8] Adam Paszke. Torchdqn: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.

[9] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[10] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[11] Zhuoran Yang, Yuchen Xie, and Zhaoran Wang. A theoretical analysis of deep q-learning. *CoRR*, abs/1901.00137, 2019.