Deep Reinforcement Learning and Transfer Learning with FlappyBird

Cedrick Argueta, Austin Chow, Cristian Lomeli
Department of Computer Science
Stanford University
Stanford, CA 94305
{cedrick, archow, clomeli}@cs.stanford.edu

Abstract

Reinforcement learning's growth in popularity in recent years is partly due to its ability to play some video games with a level of mastery that no human can reach. Transfer learning is popular in the field of deep learning, and using pre-trained models on certain tasks speeds up training time and increases performance significantly. In this project we aim to apply transfer learning to the popular video game *FlappyBird* and analyze its performance compared to traditional reinforcement learning algorithms.

1 Introduction

Reinforcement learning is a technique for solving certain decision tasks, where an agent attempts to maximize a long-term reward through trial and error. This trial and error paradigm is better explained through the lens of exploration and exploitation: the agent must decide whether it explores and learns new information, or exploits the current policy to maximize reward. We will primarily explore the applicability of deep Q-learning, a deep reinforcement learning algorithm, to the FlappyBird game. In addition, we intend to explore the impact of transfer learning, a machine learning technique where a model developed for a specific task is reused as the starting point for a model in a second task. Being able to transfer learn will allow us to develop a more general model for different tasks, saving time and effort. In particular, we aim to see whether transfer learning will speed up training time to reach the same performance level as the DQN model.

2 Related Work

The impetus behind our project is DeepMind's paper, "Playing Atari with Deep Reinforcement Learning". Mnih et al. were able to show how a Deep Q-Network could take raw image pixels from an Atari game and estimate the Q function. [6] Their model also takes advantage of recent advances in convolutional neural networks for image processing, a target network for stabilizing policy updates, and experience replay for a more efficient use of previous experience. Transfer learning is the idea that generalization occurs both within tasks and across tasks. [9] Deep reinforcement learning may benefit from transfer learning, especially since convolutional neural networks are often used for playing games. Since convolutional neural networks often learn similar features in the first, second, etc. layers across a range of image classification tasks, it's possible that transfer learning in the context of reinforcement learning for video games can exploit this.

Preprint. Work in progress.

3 Game Mechanics

The game of FlappyBird can be described as follows: a bird flies at a constant horizontal velocity v_x and a variable veritcal velocity v_y. The bird can flap its wings, giving it a boost in altitude and velocity v_y. The aim of the game is to avoid randomly generated pipes that restrict the flying area, leaving only a small gap for the bird to fly through.

We model the problem as a Markov decision process with no knowledge of the transition probabilities or reward function at every state. The transition probabilities are unknown, since each state consists of the deterministic bird position and velocity, along with the non-deterministic pipe positions. The only reward signal received from the game in its standard implementation is when the bird flies past a pipe, giving us a reward of 1. This sparse reward makes it impossible for us to get an explicit reward for each (s, a, s') tuple. The start state s_{start} is the bird at some constant height, with no pipes on the screen. The actions for every state are the same: the bird can flap its wings or not. The only exception to this are the end states s_{end}, where the bird collides with a pipe or the ground. In these cases, there are no actions to take, and the episode ends.

We have a similar model for *PixelCopter*, the game we plan on transfer learning with. These games are similar in terms of objective and action/observation spaces, so we hope that by applying transfer learning we are able to speed up training time and perhaps maximum absolute reward.

4 Approach

The goal of our project is twofold: we aim to evaluate deep reinforcement learning algorithms on the FlappyBird game, and also experiment with transfer learning to analyze the impact it makes on the training process.

Vanilla learning methods like Q-learning are not well suited to this problem, since the state space of the game is very large. The position of the bird and pipes are continuous values, and as such we have an almost-zero probability of reaching a particular state that we've seen before. Thus, it will be necessary to use either function approximation to generalize to unseen states or some form of deep learning that can extract features automatically. We envision using Deep Q-networks as a nice foray into deep learning, given our experience with Q-learning. The DQN model uses a neural network to approximate the Q action-value function, and allows for better generalization than the standard Q-learning approach.

Furthermore, we will demonstrate the ability for policies learned through deep reinforcement learning on other games to transfer to FlappyBird. This has been demonstrated before, particularly through the use of convolutional neural networks for playing games directly from pixels. [6] In our case specifically, we will develop a Q-network parameterized by weights $\theta_{FlappyBird}$ that performs 'well enough' on FlappyBird, along with Q-networks parameterized by $\theta_{PixelCopter}$ that perform 'well enough' on the game PixelCopter. These networks will be initialized through Xavier initialization. [5] Once we have this, we will begin training new Q-networks parameterized by $\theta'_{PixelCopter}$ that were initialized with $\theta_{FlappyBird}$. We do this by fixing weights up until the final hidden layer, and solely training those weights. This gives the effect of reducing the number of parameters needed for optimization, making our training faster and more efficient. We will then compare the training times and absolute performance of the θ and θ' models, demonstrating the increase in efficiency or performance that transfer learning entails.

4.1 Infrastructure

The infrastructure for the game comes mostly from the PyGame Learning Environment, OpenAI gym, and keras-rl packages. The PyGame Learning Environment provides a nicely wrapped implementation of FlappyBird, complete with sprites and the relevant game mechanics built in. [8] [3] [2] Keras-rl provides a deep reinforcement learning framework that gives us a simple interface for training agents. We take advantage of these two here, writing simple wrappers for the PLE game instance so that it is compatible with keras-rl. [7] The keras-rl package additionally provides an implementation of several algorithms that are applicable to FlappyBird, particularly Deep Q-networks and the SARSA algorithm. For the transfer learning portion of the project, we saved weights info `hdf5` files with the Keras package. The neural networks and much of the other deep learning architecture was also created in Keras, with a TensorFlow backend. [4] [1] This allowed us to monitor training time and other metrics inside TensorBoard.

Code for our project can be found at this link: https://github.com/cdrckrgt/cs221-project.

5 Model

In this section, we describe the DQN algorithm and how it applies to our task. We use a modified version of the algorithm described in the 2013 DeepMind paper.

Again, we model the environment $\mathcal{E}$ as a Markov decision process with no knowledge of the transition probabilities or reward function at every state. There is a set of actions that may be taken at every time-step, designated $\mathcal{A}$. Each interaction between the agent and its environment at time t produces an observation $x_t \in \mathbb{R}^d$ and reward r_t, where d is the dimensionality of the environment.

As in the standard Q-learning algorithm, we consider sequences of observations, actions, and rewards $x_1, a_1, r_1, \ldots, x_t, a_t, r_t$ as the state of $\mathcal{E}$ at time t. The objective of the agent is to maximize its future reward, defined as:

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \tag{1}$$

such that T is the termination time-step of the game. Then the optimal Q-value function is defined as follows:

$$Q^*(s, a) = \max_{\pi} \mathbb{E}[R_t \mid s_t = s, a_t = a, \pi] \tag{2}$$

such that π is a policy that maps states to actions. So far we have described steps to arriving at the Q-learning algorithm described by Dayans and Watkins. [10] The DQN algorithm presented by DeepMind makes numerous improvement on this algorithm by introducing a non-linear function approximator and experience replay.

5.1 The Q-network

Since the vanilla Q-learning algorithm does not generalize well to state spaces as large as those encountered in video games, we use a neural network to approximate the Q-value function. Then we have a neural network parametrized by weights θ such that $Q(s, a; \theta) \approx Q^*(s, a)$. At every iteration of training i, we minimize the squared error between the current Q-function and the optimal Q-function, according to the loss function:

$$L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)}[(y_i - Q(s, a; \theta_i))^2] \tag{3}$$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}}[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1} \mid s, a]$ and $\rho(s, a)$ is a probability distribution over states and actions. The intuition behind this loss function is that we are minimizing the distance between our target, which is the maximum expected future reward for this state and the previous iteration's weights, and our current Q value. We are effectively getting closer and closer to the true Q values every training iteration. To minimize this objective function, we update the weights according to the gradient:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)}[(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i))\nabla_{\theta_i} Q(s, a; \theta_i)] \tag{4}$$

And we've now arrived at the Deep Q-network algorithm that we can apply to our task. It's important to note that the original DQN paper applies this algorithm to raw pixel vectors as each x_t, whereas we use the internal game state for each x_t. This allows us to achieve the same level of performance in a much quicker fashion, since we don't need to spend the extra compute power on training a convolutional neural network to learn features for each game. The algorithm, lifted from Mnih's paper, is described in 1.

5.2 Experience Replay

Additionally, the algorithm presented in Mnih's paper makes use of experience replay to smooth learning and prevent divergence. Rather than performing gradient updates on each consecutive sample, samples are stored in a ring buffer and sampled randomly during the learning phase of the algorithm. This has the effect of improving sample efficiency and removing the dependence of each training update on the previous training update.

Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and observation $x_{t+1}$
        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 $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 4
    end for
end for
```

5.3 Target Network

Finally, we make use of a target network to further stabilize the training process. Note that during our gradient descent, the weights that produce our target values shift every iteration. Instead of using these rapidly shifting weights, we copy the parameters of the training network $\theta_{training}$ to a second, identical model θ_{target}. With this identical model, we may compute the same targets, albeit with less variance by only updating the θ_{target} to match the $\theta_{training}$ with some fixed, low probability at every iteration.

5.4 Annealed ϵ-greedy Policy

Rather than use the standard ϵ-greedy policy, we opted to use a policy that annealed ϵ from some value α to some value β over training. Our goal was to incentivize exploration early on, but revert back to the usually ϵ-greedy policy after some about of training.

5.5 Model Architecture

Since we use as input to the Q-network the internal game state representation as opposed to the raw pixel inputs, we don't make use of a convolutional neural network to learn features directly from the game's pizels. We instead make use of a multi-layered perceptron that takes as input the game's state in vector form. In the case of FlappyBird, the state vector is a vector in $\mathbb{R}^8$, and contains the bird's position, velocity, and the distance, top, and bottom positions of the next two pipes. The dimensionality of the input differed for each game we worked with, but we kept the dimensionality of the hidden layers the same for each game so that we could experiment with transfer learning. We have three fully connected hidden layers, with dimensionality 256, 128, and 16. The outputs of the Q-network for each game were the Q-values of all actions in the action space for that game. In the case of FlappyBird, the output was two Q-values in $\mathbb{R}$, one for the action to jump, and the null action.

6 Preliminary Results

6.1 Vanilla DQN

After finalizing the DQN algorithm, we experimented with different hyperparameters and had to define performance metrics to compare the base DQN models with the transfer learned models. The hyperparameters for the models trained from scratch are shown in table 1. We see in the graph 6.1 the variance in episode reward over different random seeds for the FlappyBird game. Each run was trained for 30,000 iterations, and we see the ability for DQN to learn to play the game sometimes but fail other times. We aim to decrease this variance soon through more hyperparameter optimization and tinkering with the model architecture more. The DQN performs well above the ability of the average human when it does learn to play the game, since it can sometimes average scores in the thousands while testing.

Hyperparameters for Xavier-initialized networks

FlappyBird Hyperparameters	
Target Model Update	1e-2
Learning Rate	1e-3
γ	0.99
Annealed ϵ	$0.4 \to 0.1$
Reward Profile	
tick	0.1
passed pipe	1.0
collided	-10.0

PixelCopter Hyperparameters	
Target Model Update	1e-1
Learning Rate	1e-3
...	...
tick	0.1
passed block	1.0
collided	-10.0

Table 1: These hyperparameters were obtained with lots of trial and error. If not noted here, the hyperparameter was left as the default value in Keras.

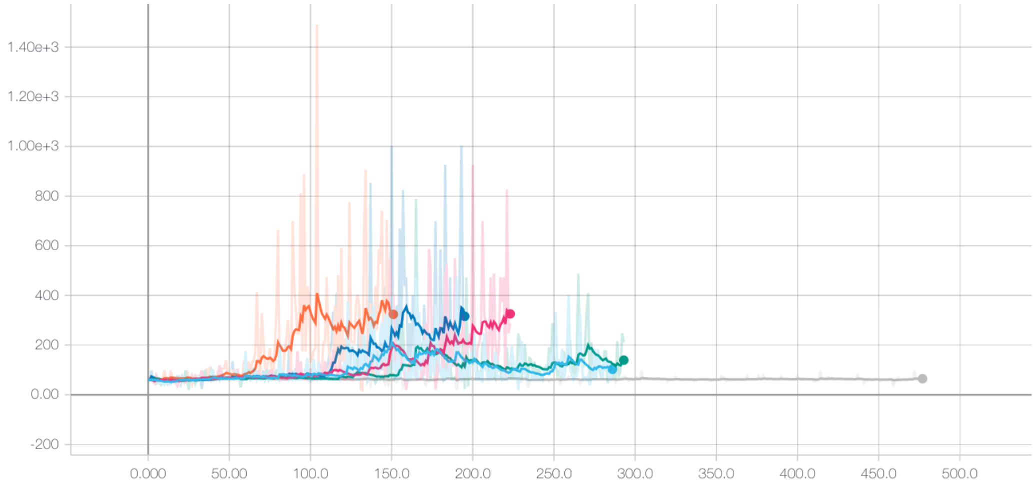

Figure 1: 6 different runs for DQN on FlappyBird. The x-axis shows episode number, while the y-axis shows episode length.

We show the performance of the DQN trained on the PixelCopter training in 2. For both games, we show the episode length as a proxy for how good the agent performs on the episode. The performance of DQN on PixelCopter without transfer learning plateaus very early on, and has a very poor performanc eoverall.

6.2 Transfer Learning

We show in 6.2 the effectiveness of using pre-trained weights from FlappyBird on the game PixelCopter.

6.3 Analysis

We are still working out the kinks from transfer learning with solely dense layers. One of the approaches we may try is just changing the input from the state vector representation to the image, since convolutional neural nets have been shown to transfer learn well with CNNs. The traditional method of freezing some subset of middle layers and only training the outer layers hasn't proved effective so far.

Additionally, we have yet to begin training on machines stronger than our laptops. Because of this, we've haven't been able to practically train for periods of longer than 50,000 iterations. Because of

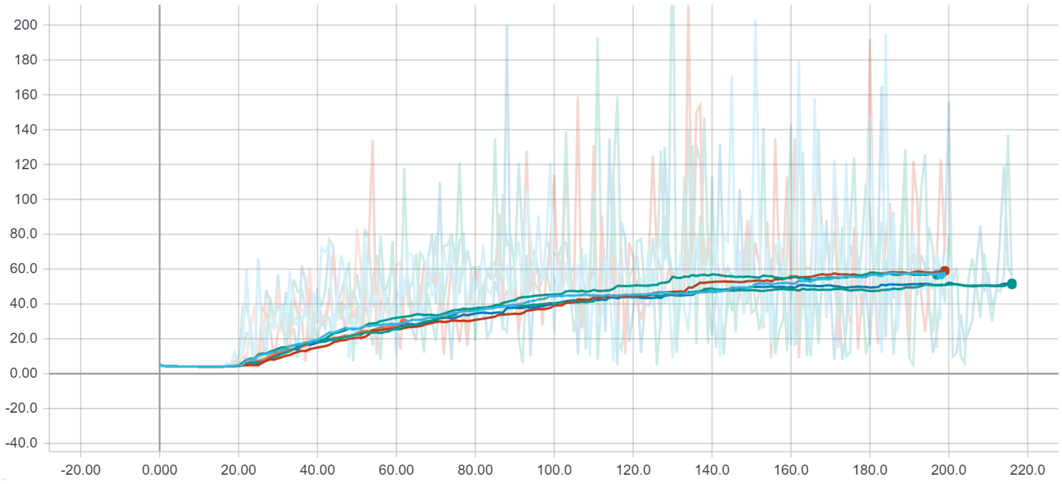

Figure 2: 5 different runs for DQN on PixelCopter. The x-axis shows episode number, while the y-axis shows episode length.

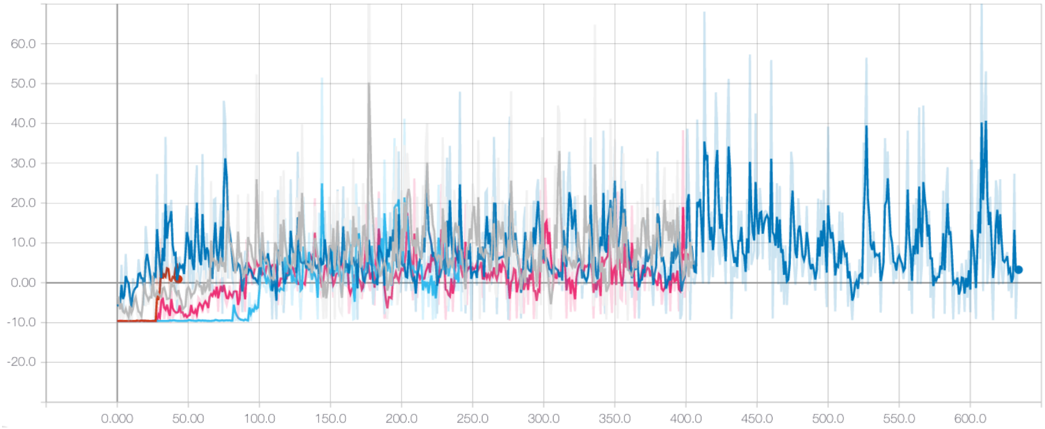

Figure 3: 6 different runs for the DQN on PixelCopter, using weights pretrained on FlappyBird.

the nature of some of these games, it might take longer than that to see some improvement, especially if we were to generalize the DQN to receieve images as input. Despite this, we were still able to get some good results on the vanilla DQN and FlappyBird.

The vanilla DQN model trained on PixelCopter doesn't generally perform well, even with long training times and hyperparameter tuning. Our hypothesis is that the state vector that the game produces doesn't yield enough information about the game for the model to learn how to play. Only the wall directly above and below the Pixelcopter is shown to the model, leaving the rest of the terrain unknown until the PixelCopter flies past it. One way to approach this issue is to simply feed the pixel representation of the game's state as image input to the model, which would necessitate the use of CNNs.

Because the performance of DQN on PixelCopter was poor to begin with, we hoped to achieve better results through transfer learning. After running tests with transfer learning, we see similar performance to the non-transfer-learned DQN model on PixelCopter, leading us to believe that the transfer learning was ineffectual. We don't know the effectiveness of transfer learning on solely fully-connected layers, and hopefully this goes to show us that we might want to continue with CNNs and learning from pixel input in the future.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *CoRR*, abs/1207.4708, 2012.

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

[4] François Chollet et al. Keras, 2015.

[5] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

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

[7] Matthias Plappert. keras-rl, 2016.

[8] Norman Tasfi. Pygame learning environment. `https://github.com/ntasfi/PyGame-Learning-Environment`, 2016.

[9] Matthew E. Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *JMLR*, 10, 2009.

[10] Christopher Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8, 1992.