Toy Helicopter Control via Deep Reinforcement Learning

Dylan Rhodes
dylanr@cs.stanford.edu

Abstract

This paper investigates the efficacy of various agents at a simplified aircraft collision avoidance task. The agents are trained to approximate the infinite horizon discounted reward (Q) function for states and actions drawn from a helicopter flying simulation. Two neural network architectures motivated by recent research in deep reinforcement learning for arcade games are implemented and benchmarked against a naive discretization strategy, a Bernoulli random agent, and human performance on the task. Source code is available here.

1 Introduction

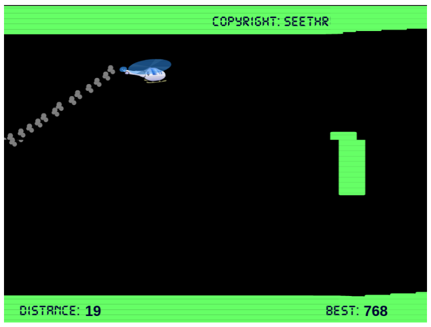

Figure 1: Screenshot depicting the original Helicopter Game.

The motivation for this project came both from interest in recent research on the application of convolutional neural network (CNN) architectures to the arcade domain as well as recognition of Professor Kochenderfer's work on aircraft collision avoidance systems. The focus of the project is the Helicopter Game, a simple web game in which the player pilots a small helicopter through a narrow cave, avoiding randomly generated obstacles. This game is non-trivial for human players, given the small margin for error and necessity of quick reaction speeds.

The goal was to benchmark a variety of strategies for approximating optimal control policies for the helicopter with the ultimate hope of creating a 'superhuman' pilot. To this end, a clone of the game was implemented in the pyGame framework which is playable by both humans and software-controlled agents. A player's score is equivalent to the elapsed time of survival, so agents must learn to compensate for the bounds of the cave, the random obstacles, as well as the helicopter's current position and velocity. Four agents were implemented, trained, and evaluated on the simulator and their performance compared to that of a human agent.

2 Related Work

Deep learning is a booming area of research within the machine learning community, especially among computer vision and natural language processing experts. Since Alex Krizhevsky's resounding success at the 2012 ILSVRC[1], work on novel network architectures, gradient-based optimizers, and efficient batch computation has become mainstream. This rush has resulted in the development of more efficient optimization methods, a proliferation of GPU-backed libraries with accelerated performance, and a multitude of cross-applications of CNN's to open problems and benchmarks across artificial intelligence.

This project derives a great deal of inspiration from some cross-domain research done at Google DeepMind on the use of CNN's as arbitrary feature transforms for image patches as input to reinforcement learning algorithms. Specifically, a team found that human level control policies could be learned for many complicated Atari 2600 games from raw pixel data inputs coupled with the use of the existing score function as reward[2]. Using a relatively shallow, generic architecture and identical hyperparameter settings, Mnih et al. were able to train effective agents for forty-nine classic Atari games with the majority performing at or above the level of a human games-tester. The policies were derived from a linear Q approximation function with learned parameters over a feature representation provided by several convolutional layers. Prior similar work had focused on the use of autoencoders to learn feature representations yet suffered from the tendency for network parameters to diverge and generally poor performance. Mnih et al propose a novel training data sampling strategy coupled with a memory bank of finite size as well as some other minor modifications which improve network convergence and performance on the final task. These training strategies, network architectures, and other modifications as they pertain to the experiments performed here are described in detail in section 4.

3 Simulation Environment

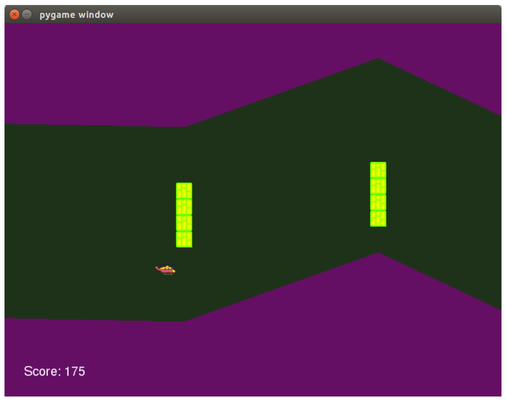

Figure 2: A CNN based agent acts in the simulator.

Although the simulator is not terribly interesting from a technical standpoint, it constituted the backbone of the training and experimental procedures as the transition and reward function of the POMDP, and its implementation formed a not-insignificant portion of work on the project. Most importantly, the simulator can be directly controlled by both learned agents and humans, which allows for experimental comparison between the two.

The simulator faithfully copies the fundamental design of the original Helicopter game depicted in Figure 1. The user controls a small helicopter avatar which begins in the center of a narrow, flat cave. At any individual frame, the user can leave the throttle on or off, which will cause positive or negative vertical acceleration respectively. Obstacles are randomly sampled at a constant rate and cause the cave walls to slope between them. Users receive a point for each frame until the helicopter crashes into the floor, ceiling, or obstacles in the cave, at which point the game terminates.

The simulator represents the transition and reward functions of a complex POMDP. A large portion of the game's difficulty for software-based agents can be attributed to the large size of the state space, given the many possible configurations of the cave walls and obstacles as well as the helicopter's real-valued position and velocity. Even human and convolutional players do not have full information of the game state, since the generated cave extends beyond the visible screen. In addition, the obstacle and cave ceiling sampling procedures are stochastic and cannot be effectively predicted ahead of time.

4 Agents

During this project, the performance of four computer-controlled agents in the simulator was compared. Three of the agents learned policies via maximization of a learned Q function estimated during training via reinforcement learning, whereas the other was a benchmark Bernoulli agent.

4.1 Bernoulli Agent

The Bernoulli Agent was used as a naive baseline for comparison with the reinforcement learning based methods. It is parameterized by a single hyperparameter $0 \leq p \leq 1$, which represents the probability of leaving the throttle on. At each time step, the agent samples from a Bernoulli random variable parameterized by p and acts accordingly. For the purposes of this project, p was set to 0.625 following an informal, manual grid search.

4.2 Discrete Q Agent

The Discrete Q Agent was the simplest of the three reinforcement learning methods applied to the control problem. It takes advantage of a drastic simplification of the state space to a discrete set of twenty thousand values in order to approximate the true Q function. The space is discretized along five axes: the helicopter's vertical position and vertical velocity, the vertical position of the upcoming obstacle, and the height of the cave ceiling over the helicopter and upcoming obstacle. Iterative updates are applied according to the standard Q learning function given below.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

4.3 Dense Q Agent

The Dense Q Agent was the first of the deep architectures implemented and applied to the collision avoidance task. It employs a fairly standard architecture along with a few training speedups which stem from the work of Mnih et al.

The agent takes as input a stack of state vectors, each of which consists of nine scalar values between zero and one and corresponds to information about the state. The values included in the state vector for this agent include the helicopter's vertical position and velocity, the height of the cave ceiling above the helicopter, the vertical and horizontal positions of the next two upcoming obstacles, and the height of the cave ceiling over them. Mnih et al. claim that concatenating the state vectors into stacks of m frames improves convergence rates over decision making at each time step. This claim is experimentally validated in section 5.1, and the final dense Q agent employed stack traces of depth $m = 4$.

The dense agent uses a shallow, fully connected neural network to approximate the Q function. It contains four linear layers which produce activations of length 256. These linear layers are parameterized by weight matrices and bias vectors and interspersed with rectified linear unit nonlinearities.

Training proceeds via batch gradient descent using Geoff Hinton's RMSProp accelerated gradient formulation[3]. The loss function combines elements of the Bellman equation and temporal difference learning:

$$L_i(\theta_i) = \mathbf{E}_{(s,a,r,s') \sim U(D)}[(r + \gamma \text{max}_{a'} Q(s', a'; \theta_i^{-}) - Q(s, a; \theta_i))^2]$$

There are two significant innovations which are worth explaining. $U(D)$ represents a memory bank of finite size in which previous (s, a, r, s') tuples are stored and evacuated at random. Sampling minibatches uniformly at random from this memory bank breaks the correlations between successive samples and is vital to the network's convergence. The dense agent's memory bank held a maximum of fifty thousand samples throughout the experiments. $Q(s', a'; \theta_i^-)$ is what Mnih refers to as the target model; namely, it is a stale copy of the network which is replaced once every C iterations. Somewhat like the memory bank, this helps to break correlations between recent gradient updates and the value function for immediately upcoming states, which could create a feedback loop inducing oscillations or divergence[2]. For the purposes of this project, C was set to $10,000$.

4.4 Convolutional Q Agent

The final agent evaluated in the simulator takes advantage of the convolutional architecture proposed by Mnih et al. It differs from the dense agent only in the network architecture of the feature transform and in the type of input. The training strategy, loss function, gradient optimization method, and all shared metaparameters were equivalent between the two deep reinforcement learning agents with the exception of the memory bank size, which was reduced to $10,000$ samples to fit in RAM.

The network architecture contains three convolutional layers followed by a densely connected linear layer and a final linear scoring function. The first convolutional layer contains 32 (8×8) filters with a stride of four pixels each. The second holds 64 filters of size (4×4) with a stride of two pixels, and the last convolution has 64 (3×3) filters with a stride of one. The first linear layer produces activations of length 512. All of the intermediate layers are followed by a rectified linear unit nonlinearity function.

It accepts stacks of pixel inputs with $m = 4$ as with the dense agent. The individual frames in these stacks are produced by a preprocessing function ϕ which resizes them to a constant width and height of eighty-four pixels and then compresses each pixel to a flat luminance value by taking a weighted sum of the red, green, and blue channels. Thus, the dimensionality of the input stack is $(4 \times 84 \times 84)$. As a final step, these luminance values are scaled between zero and one and transformed to four byte floats.

5 Experimental Results

Over the course of this project, two main experiments were performed. The first sought to assess the importance of stacking states to the convergence rate and policy quality in order to properly parameterize the deep models. The second experiment was a comparison between the performance of the policies learned by the four software agents and a human actor.

5.1 State trace depth

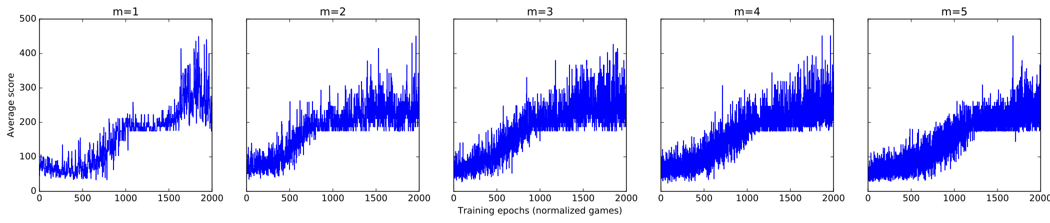

Figure 3: Training performance for dense agents with various state stack depths.

	m=1	m=2	m=3	m=4	m=5
Test Score	227.14 ± 9.51	248.77 ± 10.31	263.06 ± 8.91	271.11 ± 10.04	242.87 ± 8.45

Table 1: Test performance for the various dense agents with 95% confidence intervals.

Mnih et al. do not discuss the importance of state stacking in detail. They claim that the use of state histories improves their networks' performance and claim that their algorithm is robust to other values of m, yet they offer no real justification for the choice of $m = 4$. It is clear that stacking states allows the agent to encounter roughly m times as many games in the same amount of processing time, yet it is unclear whether this is the root cause of the increased performance and whether $m = 4$ is optimal for the helicopter problem.

To sort out these issues, dense agents were parameterized with different m values where $1 \leq m \leq 5$ and trained for an identical number of iterations. The performance during training is illustrated in Figure 3, and the performance of the strategies extracted from the final learned Q functions is given in Table 1.

There are several features of Figure 3 which are worth noting. Of immediate interest is the shelf which all models manage to attain. This base cutoff value represents the point at which the agents learn to successfully navigate the cave up until the first obstacle, crashing into which earns a score of 175. Interestingly, the $m = 2$ model reaches this stage first, although its final score is lower than that of the $m = 3$ and $m = 4$ agents. Learning effectively after the first obstacle has been reached is crucial to overall performance, since the rest of the game follows roughly the same distribution of states from then on. Based on its comparatively quick optimization past this point and overall strongest test performance, m was set to 4 for both final deep agents.

5.2 Simulator Performance

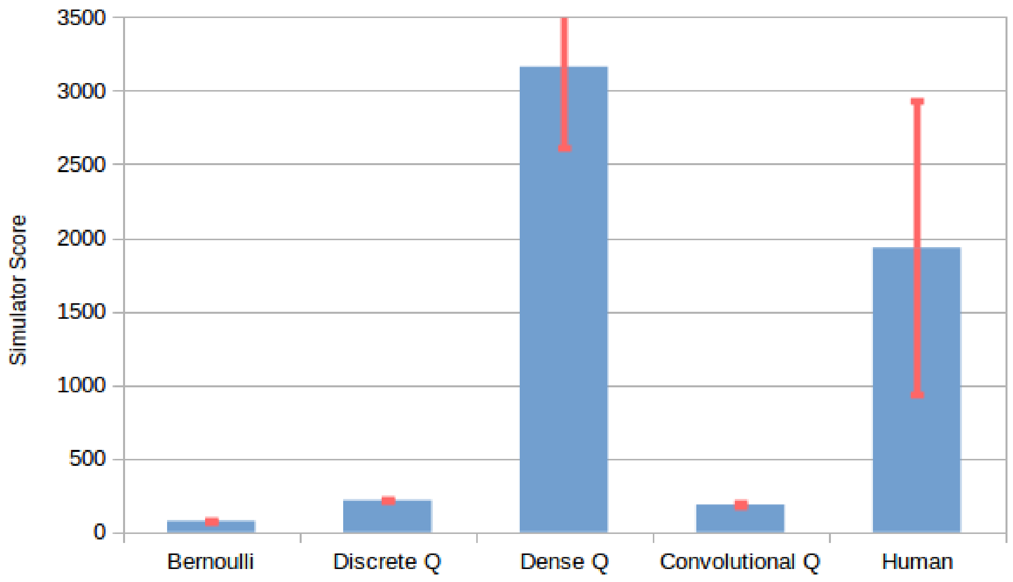

Figure 4: Final results of trained models in simulator; error bars represent 95% confidence intervals.

The final experiment performed was to benchmark each trained agent over a set of $n = 500$ runs following the extracted optimal policy. Unfortunately, due to runtime constraints, it was not possible to train all of the agents for an identical number of games. Both the discrete Q agent and the dense Q agent were trained for 30,000 games apiece. This training procedure took about six hours for the discrete agent and twenty-five for the dense Q agent. The convolutional Q agent proved still more costly to train, even after the expensive matrix operations had been moved to the GPU. Further efforts with a profiler and some cache optimizations reduced the runtime by around thirty percent, but the bottleneck of performing backpropagation was not possible to avoid. It did not seem acceptable to reduce the size of the input image further, and it seemed worthwhile to try to maintain the network architecture from Mnih's work, so this agent was trained for only 15,000 games. The training procedure still took over thirty-one hours to run to completion.

As one can see in Figure 4, the dense Q agent ultimately performed the best in the simulator. The final trained agent posted an impressive maximum score of 14,476 which represents over eight minutes of flying time without a crash at human speed. The performance of the convolutional agent was disappointing, but it can likely be attributed to an inadequate amount of training iterations. All three reinforcement agents surpassed the Bernoulli baseline by a significant margin.

The human trials were performed by two separate people. They were each given a few games to train on until they felt comfortable, after which their performance was recorded. Both users had some previous experience with the classic Helicopter game, although they had not interacted with this particular incarnation. The game was run at a relatively laconic 30 frames per second, which made it significantly easier than the more well-known version. Still, the human players were drastically outperformed by the dense Q agent. Only twenty-five samples were collected from the test players, but they proved sufficient to illustrate the superiority of the dense Q agent.

6 Conclusion and Future Work

There were several lessons to be gleaned from this project. Above all, it illustrates the pitfalls of working with computationally costly models. Both the developmental and experimental stages of the project were slowed considerably by the necessity to occasionally wait extended periods of time for networks to train and be evaluated. To put the performance of these models in context, Mnih et al trained their agents over fifty million frames apiece, whereas the most well-trained agent in these experiments - the final dense Q agent - only ever saw slightly over one million.

It may be that the performance issues can be traced to deficiencies in the code base, although profiling shows that the vast majority of time is spent in backpropagation. On the advice of Professor Kochenderfer, both deep agents were implemented using the Chainer library for Python. The matrix calculations for both deep agents were performed via calls to CuDNN, but it may be that the available GPU, an Nvidia 750M with just 1GB of dedicated memory, was not up to the task. It would be interesting to evaluate the performance of the agents on more powerful hardware or after a longer training period. None of the agents had reached convergence by the time training was stopped, so it would be reasonable to expect better test performance.

There are also other techniques which could be applied to improve the performance of the agents. More extensive preprocessing on the input vectors such as zero-centering, PCA, or even whitening would be likely to improve performance. Another possible improvement would be the application of dropout regularization to the training period. This was not possible in the context of this project, since dropout generally increases the time to convergence.

Overall, the agents' performance was satisfactory. The dense Q agent learned a surprisingly strong policy which was able to beat the two human game testers even playing at a low frame rate. In addition, the other two agents handily surpassed the performance of the Bernoulli baseline. In this regard, the project shows that reinforcement learning is an effective, though computationally costly, family of algorithms.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[2] 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.

[3] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4, 2012.