Playing CHIP-8 Games with Reinforcement Learning

Niven Achenjang, Patrick DeMichele, Sam Rogers
Stanford University

Abstract

We begin with some background in the history of CHIP-8 games and the use of Deep Q-Learning for game playing from direct sensory input. We then outline our methodology for adapting Deep Q-Learning for playing CHIP-8 games, with *Pong* as the primary example. We then give results for three experiments training an agent to play *Pong*: one baseline using discretized screen states and value iteration, one using Deep Q-Learning on a feedforward neural network, and one using Deep Q-Learning on a convolution neural network.

1 Introduction

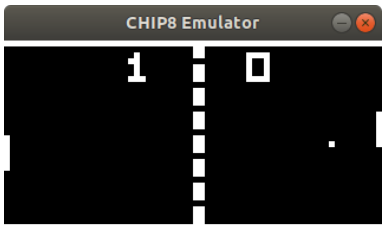

Figure 1: A CHIP-8 emulator running *Pong*

1.1 Background

In a paper published at the 2013 NIPS Deep Learning Workshop titled "Playing Atari with Deep Reinforcement Learning", DeepMind Technologies presented what they described as "the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning" [1]. In their paper, DeepMind created an agent that successfully learned to play seven arcade games for the Atari 2600 from screen data and game score alone. The method developed for this project was termed Deep Q-Learning. Our goal was to reproduce some of DeepMind's results in a scaled back setting: CHIP-8 games.

CHIP-8 is an 8-bit programming language developed in the 1970s for creating videogames on microcomputers [2]. Many classic video games, such as *Pong, Pac-Man,* and *Space Invaders* had ports for CHIP-8. While computers that actually run CHIP-8 are hard to find today, open-source CHIP-8 emulators are widely available from the internet.

1.2 Motivation

Applying Deep Q-Learning to learn CHIP-8 games offers an opportunity to reproducing DeepMind's results in a significantly lower-dimensional (and thereby less computationally expensive) setting. Atari 2600 screens are large and in color, being represented as $84 \times 84 \times 4$ float arrays. CHIP-8 screens are smaller and monochromatic, and were able to be represented as 32×64 boolean arrays. Using much smaller dimensional input allowed us to assess Deep Q-Learning's efficacy compared to

more traditional reinforcement techniques, such as value iteration. While Deep Q-Learning has been shown to be much more effective in the case of Atari 2600 games, we were interested in seeing if this was also the case in a downscaled setting.

2 Methodology

2.1 Emulator and Reinforcement Learning Environment

We used an open source Python-based CHIP-8 emulator [5] to perform our experiments. The original emulator drew pixels directly to the screen, and this was modified to keeping track of the screen as a boolean array to allow games to be played without actually rendering the games in an on-screen window.

All experiments were conducted on a CHIP-8 port of the game *Pong* (technically *Pong 2*, which differs from the original only in that it has a dividing line down the center of the screen). While the game is normally played by two human opponents each controlling one paddle, we hard-coded a computer opponent using known data about the paddles and ball for the agent to play against.

The basic setup for reinforcement learning was the same across experiments. We set each time-step to be equivalent to six frames of game-play, and each time-step the two consecutive screen arrays were sent to the RL algorithm as a single state. The algorithm would then respond with one of three actions: move the paddle up, move the paddle down, or stay still. Reward was then given based on the score of the game.

2.2 Discretized Screens and Value Iteration

In order to gauge the relative efficacy of a Deep Q-Network (from now on DQN) for learning CHIP-8 games, we chose to use value iteration on transition probabilities as a baseline test for reinforcement learning. Since there are 2^{512} possible states, it was impossible to use this technique on the states as is. Because of this we decided to discretize the screen into 32 possible states based on the region of the screen which contains the centroid of the activated pixels (that is, the white pixels on the monochrome screen). We then perform value iteration to calculate optimal transition probabilities on the discretized states.

2.3 Implementing the DQN

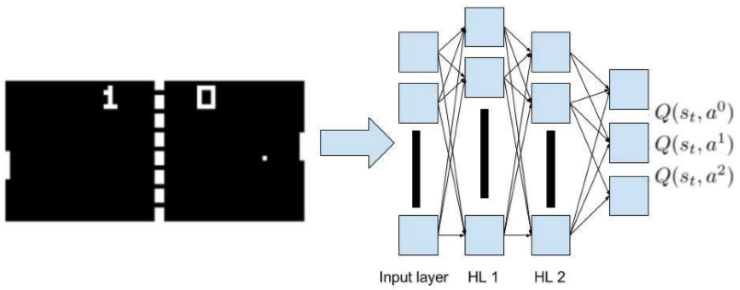

Figure 2: Architecture of a DQN

We performed experiments using two different neural network architectures for our DQN, one feed-forward network and one convolutional network. All neural network architecture was implemented using PyTorch.

In both cases, policies were updated according to the standard Q-Learning update rule:

$$Q(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} \right)$$

Where the neural network was used to estimate the value of the function Q for given state action pair, which allows the network to be updated with standard backpropagation. The feedforward network used ReLU activations with two hidden layers, whereas the convolutional network used two batch norm layers. Both used smooth $L1$ loss for loss functions.

2.4 Memory Replay and Epsilon-greedy Exploration

In addition to the DQN architecture outlined above, we also implemented two techniques known as memory replay and ϵ-greedy exploration. Both techniques have been shown to increase stability in reinforcement learning applications where reward is sparse [3] (in *Pong* for example, upwards of a hundred updates may go by before the efficacy of an action with respect to reward is made apparent).

Memory replay involves saving each state s_i, action a, successive state s_{i+1}, and reward r in a tuple $T = (s_i, a, s_{i+1}, r)$. We then have a fixed length array *memory* which stores the last N transitions for some fixed N. The DQN is then updated using mini-batch gradient descent on randomly sampled transitions from memory. While performing updates on each transition sequentially would cause each state to be highly similar to the last, memory replay decorrelates the states, and allows each state to potentially be learned more than once [4].

ϵ-greedy exploration sets a threshold where it will act randomly with probability ϵ, and will take an action from the RL algorithm with probability $1 - \epsilon$. At the beginning of testing, ϵ will be very high, causing the agent to act randomly the majority of the time, and is then slowly annealed as testing continues. This random exploration helps prevent the agent from getting caught in local maximums.

2.5 Other Testing Considerations

In order to improve the performance of the DQN, we implemented several additional features, which included:

- A separate target network which updates every three episode
- $L2$ regularization of network parameters
- Addition of Gaussian noise to states
- Extra reward or penalty for winning or losing an episode

3 Experiments and Discussion

All experiments were conducted with a learning rate of $\alpha = 0.001$ and a discount value of $\gamma = 0.999$. Each episode is one game of *Pong*, which ends when either the agent or opponent reaches 10 points. Tests were run for roughly 1500 episodes each, although there was some variation due to variable episode length. Memory length was set to 2^{13} transitions, and gradient descent for the DQN was performed on mini-batches of size 256 each time-step, sampled randomly from memory. For ϵ-greedy exploration, ϵ was initialized to 0.9 and annealed to 0.05 with a decay rate of $n/100$ where n is the number of episodes played.

The agent was awared a reward of $+1$ after scoring a point, and received a penalty of -2 after being scored upon. It was also given a reward of $+20$ if it won a game, to ensure its net reward was positive for that episode, and a penalty of -20 if it lost to ensure its net reward was negative.

3.1 Baselines

The average value of the "Q function," average reward, and squared norm of parameters across episodes is given below. We put Q function in quotes since the baseline did not have a Q function in the same sense as the DQN's did, but rather gave a value for each discrete state.

We see that the expected reward from action quickly drops to around -100 and stays there, and the average reward across episodes does not significantly increase throughout training, although it does appear to have won a few games. From this data we can surmise that the baseline did little better than random chance after learning.

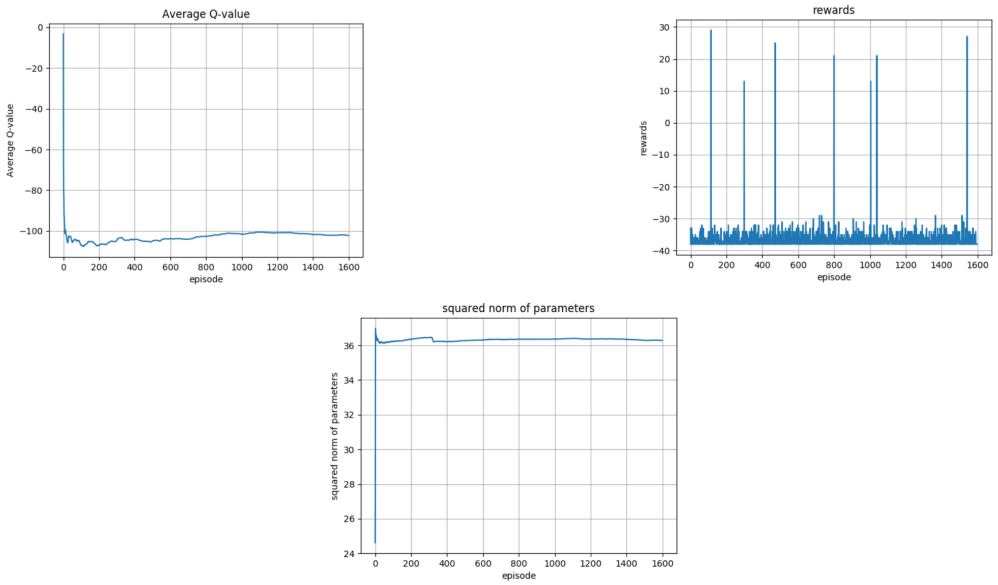

Figure 3: Baseline Value Iteration Results

3.2 Feedforward DQN

The average value of the Q function, average reward, and squared norm of parameters across episodes is given below.

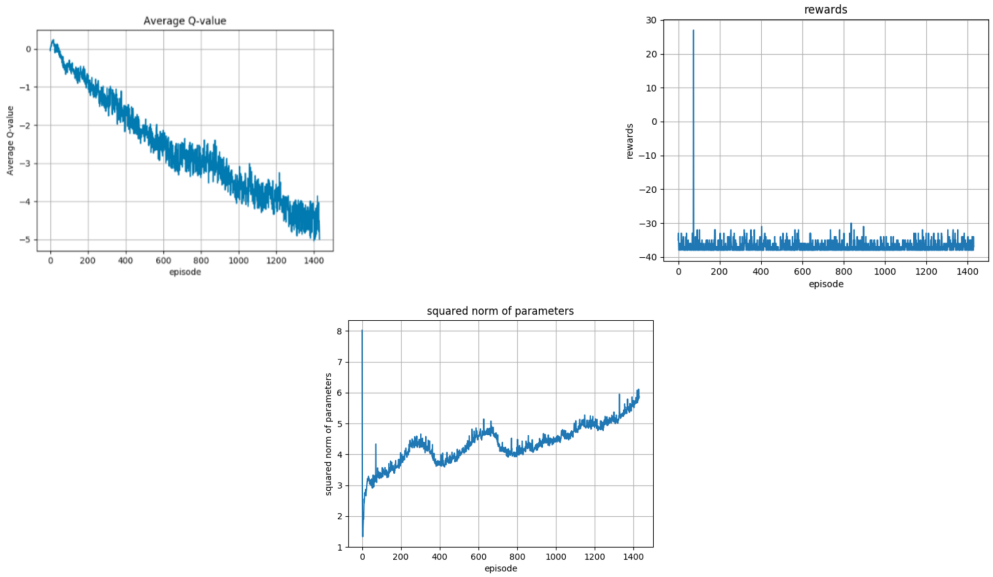

Figure 4: Feedforward Network Results

As we see above, the average Q value was almost exclusively negative and decreasing throughout testing (the short positive values near the beginning of testing was likely due to random initialization of network parameters). Moreover, reward showed no improvement, and the feedforward net only won a single game, likely due to chance, near the beginning of training.

Overall, the DQN with a feedforward network showed little improvement over the baseline testing and therefore performed not much better than random chance.

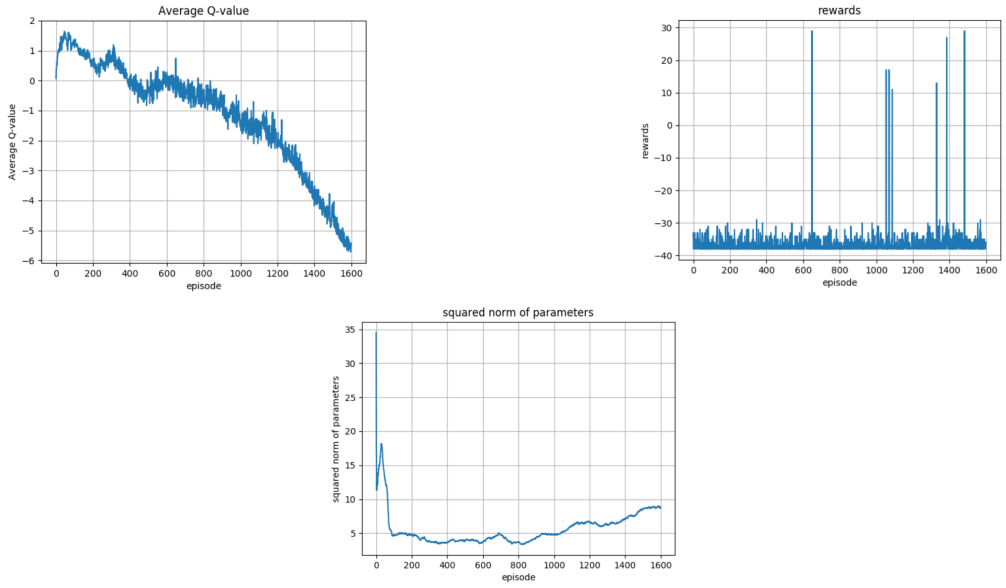

Figure 5: Convolutional Network Results

3.3 Convolutional DQN

The average value of the Q function, average reward, and squared norm of network parameters over each episode is given above.

Here see the loss function stabilize better than with the feedforward network, however the Q values still trend monotonically negatively. We see that while the convolutional net didn't win any games near the beginning of games, it won games more often toward the end of training.

4 Conclusions

As evidenced in the data above, the baseline testing and feedforward DQN were unfavorable options in teaching the agent effective policies for *Pong*-playing. However, the convolutional DQN, despite being significantly more computationally expensive than the other techniques, only showed marginal signs of improvement throughout testing. This indicates it had at least begun to learn partial policy controls for playing the game, but significant increases in reward were not observed.

In general, the test with the convolutional network did not meet expectations. One reason for this could simply be not enough computation time: the convolutional net requires significant computation time, so that we were constrained in the number and length of tests we could perform. Another issue we noticed after finishing testing was that our update rule for the Q function was giving the incorrect value on terminal states (those at the end of an episode). Given more time, we would also try varying values for replay memory and ϵ-greedy exploration, as small changes in these parameters have been shown to significantly affect learning [3].

5 References

[1] Volodymyr Mnih *et al.*, "Playing Atari with Deep Reinforcement Learning," DeepMind Technologies Ltd. arXiv:1312.5602

[2] Laurence Muller, "How to Write a CHIP-8 Emulator," http://www.multigesture.net/articles/how-to-write-an-emulator-chip-8-interpreter/

[3] Michel Tokic and Gunther Palm, "Value-Difference based Exploration: Adaptive Control between epsilon-Greedy and Softmax," *Proceedings of the 34th Annual German conference on Advances in artificial intelligence*

[4] Ruishan Liu, James Zou, "The Effects of Memory Replay in Reinforcement Learning" arXiv:1710.06574 [cs.AI]

[5] https://github.com/craigthomas/Chip8Python

6 Contributions

Niven Achenjang: Helped plan, organize, and perform experiments. Implemented the RL environment that the agent interacts with. Used PyTorch to implement the convolutional DQN. Implemented a collection of flags to turn on and off certain paramters ($L2$ regularization, noise, convolutional vs. feedforward NN, and so on), as well collection of testing data into .csv files for analysis.

Patrick DeMichele: Helped plan, organize, and perform experiments. Created the computer opponent for the agent to play against. Used PyTorch to implement the feedforward DQN. Implemented the value iteration baseline test.

Sam Rogers: Helped plan, organize, and perform experiments. Modified the CHIP-8 emulator to run without rendering the screen. Implemented memory replay and ϵ-greedy exploration. Compiled results into this report.