CS229 Final Report
Deep Q-Learning to Play Mario

Sean Klein
Department of Computer Science
Stanford University
Email: saklein@stanford.edu

***Abstract*—In this paper, I study applying applying and adjusting DeepMind's Atari Deep Q-Learning model to train an automatic agent to play the 1985 Nintendo game Super Mario Bros. The agent learns control policies from raw pixel data using deep reinforcement learning. The model is a convolutional neural network that trained through only raw frames of the game and basic info such as score and motion.**

I. Introduction

Coming into this project, I initially intended to use traditional Q-learning with the Mario AI Competition benchmark software, as much game information is available to make hand-crafted features. However, the success of this model then depends heavily on the quality of the feature representation. Representing the all possible permutations of Mario's world, creates an exceedingly large state-action space.

Another noteworthy challenge in reinforcement learning stems from the fact that rewards are sparse and time-delayed. When the agent earns a reward, we must determine which of the preceding actions played a role in getting that reward, and to what extent.[1] Despite these challenges, a convolutional neural network, trained with a Double Q-learning algorithm and updated with stochastic gradient descent can reasonably well compared to a new human player.

In this project, I study how to adapt and improve the Atari Deep Q-Networks to train a Mario controller agent, which can learn from the game raw pixel data and in-game score. For my controller, I use a port of the Arcade/Atari Learning Environment (ALE) and that instead uses FCEUX, an open source NES emulator in order to implement an adjusted DQN strategy known as Double Deeq Q-learning as standard Q-learning seemes to struggle with long-horizon games.

II. Background

In this section, we briefly introduce the Mario game mechanics as well as the model and learning algorithm used.

A. Game Mechanics

We consider tasks in which our agent interacts with an environment $\mathcal{E}$, in this case the NES emulator, in a sequence of actions, observations, and rewards. At each time-step the agent selects an action a_t from teh set of legal game actions, $\mathcal{A} = \{1, ..., \mathcal{K}\}$. The action is passed to the emulator, which modifies its own internal state and the game score. This internal state is not observed by the agent, which, rather observes a vector of raw pixels that represents the current on-screen frame.[1] Additionally, the agent receives a reward r_t which is determined by a linear combination of the change in the total game score and the distance the agent moved to the right. The primary objective of the game is to reach the flag post at the end of each stage without Mario losing all of his lives. The secondary objective is to obtain the highest score possible, which is done through collecting items, killing enemies, and completing a level quickly.

B. Q-Learning

Because the agent only observes the current frame at any one time-step, the task can only be partitially observed and many of the states of $\mathcal{E}$ are pereptually aliased - meaning the current screen x_t

is not enough information to understand the entirety of the agent's in-game circumstance. Therefore we consider a sequence of actions and states and learn strategies from these. We assume that all sequences in $\mathcal{E}$ terminate in a finite number of time steps. Thus we can we can model Super Mario as a finite Markov decision process (MDP), in which each sequence is a distinct state. This allows us to use standard RL mthods for MDPs by using the ocmplete sequence as the state representation at time tn.[2]

To solve sequential decision problems, we can estimate for the function

$$Q^*(s,a) = \mathbb{E}_{s'}[r + \gamma max_{a'} Q^*(s',a')|s,a] \quad (1)$$

using Q-learning. Note here $\gamma \in [0,1]$ is the discount factor that determines the trade-off between short and long-term rewards. However, traditional Q-learning requires that we learn all action values in all states separately, which is impractical in a game as complex as Super Mario Bros. Instead we can learn a parametarized value function $Q(s,a;\theta_t)$. The standard Q-learning update for the parameters after taking action a_t in state s_t and observing the immediate reward R_{t+1} and rsulting state s_{t+1} is then

$$\theta_{t+1} = \theta_t + \alpha(y_t^Q - Q(s_t,a_t;\theta_t))\nabla_\theta Q(s_t,t_t;\theta_t).[3] \quad (2)$$

where α is a scalar learning rate and the target Y_t^Q is defined as

$$y_t^Q = r_{t+1} + \gamma max_a Q(s_{t+1},a,\theta_t) \quad (3)$$

This update resembles stochastic gradient descent and update the current value of $Q(S_t,A_t;\theta_t)$ toward a targetY_t^Q.

C. Deep Q Networks

A deep Q network (DQN) is a multi-layered convolutional neural network that outputs a vector of action values given state s and network parameters θ. It is a function from $\mathbb{R}^n$ to $\mathbb{R}^m$, where n is the dimension of the state space and m is the dimension of the action space. Three key elements of the Deep Q-network algorithm are experience replay, fixed target Q-networks, and limiting the range of rewards. Experience replay addresses the previously stated problem that rewards are often time-delayed. It helps break correlations in data and learn from all past policies. A bank of the most recent transitions are stored for some predetermined steps and sampled uniformly at random to update the network.[1]

We also fix the parameters used in the Q-learning target. We do this by making fixing the parameters θ^- in the target network, and only update them every τ time-steps so that $\theta_t^- \leftarrow \theta_t$ at designated intervals. Thus our objective function is

$$y_t^{DQN} = r_{t+1} + \gamma max_a Q(s_{t+1},a,\theta_t) \quad (4)$$

.

Finally, we clip the rewards at some determined threshold, often at $[-1,1]$ which prevents the Q-values from becoming too large and ensures that the gradients are well conditioned.[4]

It should be noted, that, for all of the strengths of Deep Q-learning, it does not perform as well on games with large

D. Double Deep Q-learning

In both traditional and deep Q-learning algorithm, the max operator uses the same values to choose and evaluate an action, which can lead to greater estimation error, and, as a result, overconfidence.[5] To mitigate this, we follow an approach proposed by van Hasselt, by assigning experiences randomly to update on of two value functions, which results in two sets of weights, θ and θ'. Each update, one is used to determine the greedy policy while the other determines its value. The target network in the deep Q-network model provides a a second value function without having to create another network. We evaluate the greedy policy with the online network, but then estimate its value with the target network. Thus our target ecomes

$$y_t^{DoubleDQN} = R_{t+1} + \gamma Q(S_{t+1}, argmax_a Q(S_{t+1},a;\theta_t),\theta_t^-). \quad (5)$$

[3]

Fig. 1. An example of a downsampled in game frame that is input into the neural network

III. Convolutional Neural Network Model

The model used worked directly with in-game frames as input, so some image preprocessing was done in order to reduce dimensionality and lessen the computational load. Each frame was downsampled from the original 256×240 pixels to an 84×84 black and white image. A square input image was needed to use GPU-based 2D convolution. We denote this process for an in game frame j as ϕ_j.[1]

The convolutional neural network architechture is described below. It mirrors the network used in the Atari Learning Environment, but with a few alterations to the size of each layer as well as minor tweaks of the filter itself.

1) Input: Four grayscale frames with a resolution of 84×84 pixels
2) Hidden Layer: Convolves 32 8×8 filters of stride 4 with the input image and applies a rectifier nonlinearity
3) Hidden Layer: Convolves 64 4×4 filters of stride 2 and applies a rectifier nonlinearity
4) Hidden Layer: Convolves 128 3×3 filters of stride 1 and applies a rectifier nonlinearity
5) Hidden Layer: Fully connected layer that consists of 512 rectifier units
6) Output: Fully connected linear layer which outputs Q-values of each valid action (15 actions in total)

[1]

IV. Results

Training was done in 10,000 step episodes (epochs) for a total of 5,000,000 steps. Each episode took 30-40 minutes using an NVidia GTX 1080. The required time to go through all 5,000,000 steps, paired with time constraints made thorough iteration difficult, and also constrained the size of the neural network itself.

A. Exploration vs. Exploitation

When training our agent, we used a linearly decaying ϵ-greedy approach. That is our agent selects the action a_{opt} which maximizes our estimated future value with probability $(1 - \epsilon)$ and uniformly selects from the valid actions otherwise.Each training epoch, we decrement ϵ by a fixed value. During training ϵ was initialized such that $\epsilon_i = 1$, as our agent had no knowledge of the world. Our target value was set to $\epsilon_f = .01$. During evaluation an testing, we set $\epsilon_{eval} = .01$ to limit exploration.

B. Learning Rate

We compute an adaptive learning rate using the RMSprop method to improve our mini-batch learning. For notational compactness, we will denote $g_{i,t}$ as the gradient of the objective function with respect to the parameter θ_i at time step t. We compute the running average of g at time t

$$\mathbb{E}[g^2]_t = \eta \mathbb{E}[g^2]_{t-1} + (1 - \eta) g_t^2 \tag{6}$$

for a decay-rate η. The root mean squared (RMS) error criterion of the gradient is thus

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\mathbb{E}[g^2]_t + \epsilon}} g_t \tag{7}$$

[6] where ϵ is a smoothing term with a small value to prevent division by zero in the parameter update. Thus our per-parameter learning rate is adjusted based on the moving average of the magnitude of its gradients. We do this in an attempt to stem the aggressively diminishing learning rates of other per-parameter algorithms.[7]

C. Discount Rate

Training and evaluation was done with a discount factor γ which denotes how much future state is taken into account during optimization. After evaluating several possible γ values I determined that the a value of $\gamma = .9$ yielded the best results.

D. Evaluation

As stated previously, evaluations were run at the end of every 5 episodes, which consisted of10,000 time steps each Unfortunately, the results, while acceptable, were a bit below expectations. However, training was stopped at 5,000,000 steps each time, but the agent was still learning and improving. A larger network and more training steps may have yielded more desirable results. During trainign with the best-found hyperparameters, due to technical difficulties, logging was overwritten. The following figures will correspond to $t = [0, 1,000, 000], [4, 000, 000, 5, 000, 000]$.

I found that, while the agent showed moments of high-level play, where it would put together an excellent sequence of moves, it would just as often make "unforced errors", such as falling into a pit. The most pronounced roadblock occurs on the second stage, which is significantly more difficult than the first and requires a precise sequence of moves, including standing still, in order to effectively navigate it.

In the first figure above, which shows the average total reward over each evaluation episode, the reward fluctuates greatly, as expected, but does not seem to trend positively as much as I had hoped. It is possible more training was simply needed in order for it to stabilize around a higher value.

In the second figure which shows the average Q value over each evaluation, notice that the value of Q is still trending upward in the final evaluation episodes. This seems to indicate that more training with the same hyperparameters could continue to improve results.

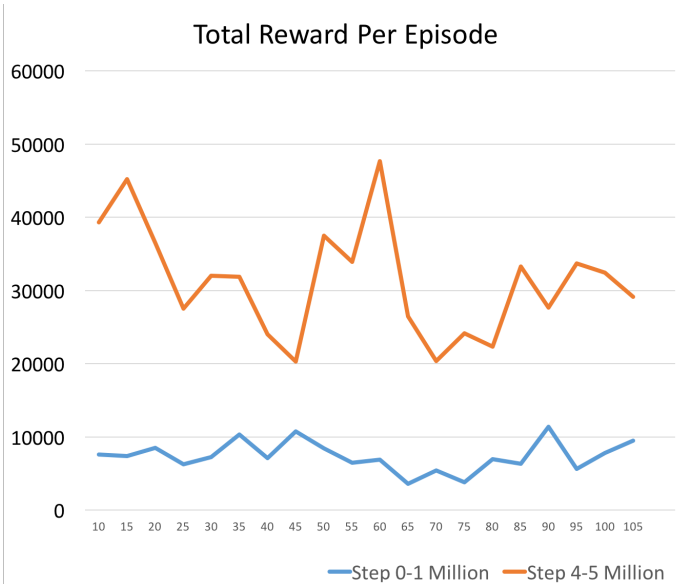

Fig. 2. Total reward accumulated during evaluation episode lasting 10,000 time steps

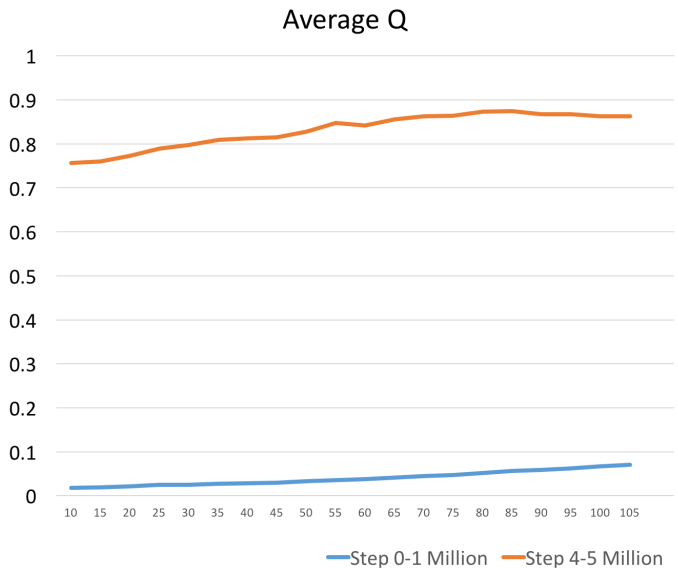

Fig. 3. Average Qvalue for each of the two logged training sessions with the best hyperparameters

V. Conclusion

In this project, I designed an automatic agent using Double Deep Q-learning based on the Atari Learnign Environment and other advances in deep RL in order to play Super Mario Bros. While the agent was able to successfully complete World 1-1 with a high success rate, it was unable to make significant progress in World 1-2 with any sort of consistency. According to a lecture from DeepMind, Deep Q-learning is an extremely effective technique when playing quick-moving, complex, short-horizon

games with fairly immediate rewards, but does not perform as well in long-horizon games that involve ”walking around”. Super Mario is a game of both short and long horizons, where the ultimate goal is fairly delayed, but there are immediate rewards and hazards in the environment. Part of the difficulty of Mario and other platformer games is that it requires precise timing and sequencing of actions in order to perform well. For further work on this project, I would first run a much longer training session to see the performance. I would also try more complex implementations of experience replay, such as prioritized experience replay, which could be sampled non-uniformly in some way.

REFERENCES

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, “Playing atari with deep reinforcement learning,” *CoRR*, vol. abs/1312.5602, 2013.

[2] R. Sutton and A. Barto, *Reinforcement learning: An introduction*, vol. 1. Cambridge Univ Press, 1998.

[3] H. van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double q-learning,” *CoRR*, vol. abs/1509.06461, 2015.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[5] H. Van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double q-learning,” *CoRR, abs/1509.06461*, 2015.

[6] Y. Bengio and M. CA, “Rmsprop and equilibrated adaptive learning rates for non-convex optimization,”

[7] T. Tieleman and G. Hinton, “Lecture 6.5-rmsprop,” *COURSERA: Neural networks for machine learning*, 2012.