AI Agent for NES Super Mario Brothers

Denis Barkar
Stanford University
dbarkar@stanford.edu

Abstract

In this work, I used the deep learning model known as DDQN to learn control policies for a NES game Super Mario Brothers. The input given to the model are raw pixels of the game screen and the output is a value function estimating future rewards. The model is a convolutional neural network that is trained through only raw frames of the game and basic info such as score and motion without no adjustments to a specific NES game.

1. Introduction

The main objective of this work is to fully adapt the original DQN's Atari 2600 code to work with any NES game. With this algorithm in hand, I aim at showing that the algorithms can be trained for different games.

The presented approach uses raw pixel data only, screen frames are downscaled and converted to grayscale.

The success of this model 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. Despite these challenges, a convolutional neural network, trained with a Double Q-learning algorithm and updated with stochastic gradient descent can work reasonably well compared to a new human player.

In this project, I used the OpenAI Gym Retro toolkit which integrates NES emulator into learning environment and allows us to focus on writing the reinforcement learning algorithms together with TensorFlow library [1].

2. Related work

Progress in Artificial Intelligence (AI) has been continuously increasing over the last years. Among the trends followed in the field, artificial neural networks are currently a top trend, mainly because of advancements in large data handling, computing power and the arrival of techniques such as backpropagation and deep learning neural networks.

Reinforcement Learning (RL) tries to emulate human behavior and the fact that humans can learn from a completely clean slate, that is, acquiring knowledge from experience and perception without depending on inherited expertise or memory. Mainly, it programs the machine (also known as learning agent) to take actions in an environment to maximize some notion of cumulative reward. The environment is mainly treated as a Markov decision process (MDP) [7] and although similar to a dynamic programming approach, a reinforcement learning approach does not assume knowledge of an exact mathematical model of the MDP and target large MDPs where exact methods become unfeasible. A direct consequence of this approach is that the training data set for the learning agent is entirely generated by the training agent itself and its actions on the environment and hence does not need a human expert to label it; a virtually infinite training data set is generated on demand and tagged according to the agent's reward system in order for the agent to progress further in its learning process, always looking for a greater reward.

A game (and moreover a computer game) is an excellent candidate for a reinforcement learning approach since every game has a set of rules, a set of available commands and a victory condition or some score system, which the player wants to achieve or maximize. Google's Deepmind group used these concepts to create the first deep learning model (namely Deep Q-Networks, also knows as DQNs) to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. Their model was trained in a set of Atari 2600 games and was able to deliver state-of-the-art results without any adjustment of the architecture or the learning algorithm [3].

Many improvements were presented in the literature since DQN has been released, including Double DQN (DDQN) [6], a DQN algorithm that decouples the action selection and evaluation in the target computation step to avoid the overoptimism of DQN, what demonstrated to lead to more stable results.

Given the state-of-the-art results observed for Atari 2600 games, the primary purpose of this work is to adapt the DDQN to be used on Nintendo Entertainment System (NES) games and evaluate its performance. The Atari games are known for their simplicity and almost complete absence of information unrelated to the play-ability (that is, graphics that only appeal to the player eye and whose lack would not affect the player's performance), while NES games have a bigger screen size, broader color palette and commands and are, in general, far more complex and have far more "irrelevant" graphics.

3. Methods

3.1. 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 the set of legal game actions, $A = \{1, \ldots, \mathcal{K}\}$. 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. Additionally, the agent receives a reward r_t which is determined by a linear combination of the change in the total game score, the distance the agent moved to the right, number of lifes, number of coins and current Mario state (small, big, with fireball). The primary objective of the game is to reach the flag post at the end of the stage without Mario losing all of his lives.

Action space is discrete and narrowed to 6 available actions (Left, Right, Up, Down, A, B).

3.2. 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 t_n [5].

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

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

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 resulting state s_{t+1} is then

$$\theta_{t+1} = \theta_t + \alpha(y_t^Q - Q(s_t,a_t;\theta_t))\nabla_\theta Q(s_t,a_t;\theta_t)$$

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

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

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

3.3. 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-network, 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.

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

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

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 action space.

3.4. 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 [6]. To mitigate this, we follow an approach proposed by van Hasselt, by assigning experiences randomly to update on of two value functions, which

Figure 1. Preprocessed frame

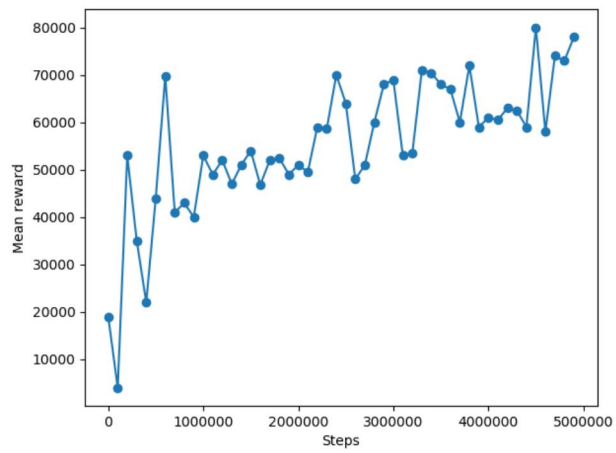

Figure 2. Mean reward

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 becomes

$$y_t^{\mathrm{DDQN}} = R_{t+1} + \gamma Q(S_{t+1}, \mathrm{argmax}_a Q(S_{t+1}, a; \theta_t), \theta_t^-).$$

3.5. Convolutional Neural Network Model

The observation is an RGB image of the screen, which is an array of shape $(240, 224, 3)$. The image frame is cropped, downscaled, converted to grayscale and normalized to an 84×84 black and white image. A square input image was needed to use GPU-based 2D convolution

The convolutional neural network architechture is described below

1. Input: Four grayscale frames with resolution of 84×84 pixels.
2. Hidden layer: Convolves 8 7×7 filters of stride 2 with the input image and applies a rectifier nonlinearity
3. Hidden layer: Max pooling 3×3 of stride 2
4. Hidden layer: Convolves 16 5×5 filters of stride 2 and applies a rectifier nonlinearity
5. Hidden layer: Max pooling 3×3 of stride 2
6. Hidden layer: Convolves 32 3×3 filters of stride 2 and applies a rectifier nonlinearity
7. Hidden layer: Fully connected layer that consists of 256 rectifier unit
8. Output: Fully connected linear layer which outputs Q-values of each valid action (6 actions in total)

3.6. Evaluation over the training process

To be able to check the progress of the agent over time during the training process, an evaluation routine (much like the test routine) is called frequently to evaluate the progress of the agent. Upon the routine's end, a few relevant parameters that are related to the agent's development are stored:

- the loss of the network, which is defined as how much the network has deviated from the start of the evaluation to its end. This parameter should ideally decrease over time, since the network adapts over time to better predict which actions will yield more rewards and the more the network is trained the better at predicting it is and the lesser it should deviate from a previous state;
- the mean Q-value, defined as the average maximum value of Q for all states in the network.This should ideally increase over time, since the network should select a speci

  c action for each instance of a game, in contrast with a random approach, which is the behavior of an untrained network;
- the mean score in the game achieved during the evaluation process, which may take multiple gameplays. This value is a much more practical way to evaluate the performance.

Now, besides the qualitative result of testing the final trained network by using it to play the game, the data above shares meaningful information about the whole training process.

4. Results

The initial results were not very promising. Although the network did indeed learn, the learning rate was prolonged

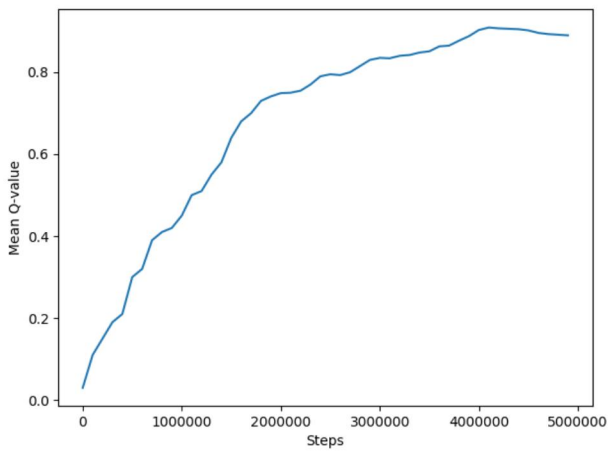

Figure 3. Mean Q-value

and very inferior to that of a human player, even a beginner.

Since the training process takes quite some time to be done and the fact that the agent indicated that it could learn further with more training time, some modification to the network was considered to accelerate the training process. Considering hyper-parameters, an adaptive learning rate α which decreased at each epoch k and a discount rate γ which increased at each epoch k until the value of 0.99 was found to achieve better results for Atari games [2].

The initial values for α and γ were set, respectively, at 0.0005 (double the original value of 0.00025) and 0.95 (the original value was fixed at 0.99). An epoch was set at 200.000 learning steps of the algorithm, so the values above would be updated after these many steps.

After the modification above the network achieved better learning rates quicker than before. Below we may see the results in figures 2, and 3.

I also implemented prioritized experience replay, which could be sampled non-uniformly.

The training was done for a total of 5000000 steps using NVIDIA RTX 2080 Ti video card. After the training, learned agent could finish the first level in most cases.

The code can be seen here: `https://github.com/mastermind-/super-mario-bros`.

5. Conclusion

In this project, I designed an AI agent using Double Deep Q-learning in order to play NES Super Mario Bros which was able to successfully complete first level of the game.

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 exploration. 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.

Although the pure RL approach worked, its results were not very encouraging, probably due to NES games superior complexity against Atari 2600 games.

For future work, it is possible to:

- increase the number of the network's layers
- train the network for much more time
- try Actor-Critic algorithm and its modifications
- try exploration based algorithm.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] V. François-Lavet, R. Fonteneau, and D. Ernst. How to discount deep reinforcement learning: Towards new dynamic strategies. *CoRR*, abs/1512.02011, 2015.

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

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015.

[5] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

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

[7] G. Weiss. Dynamic programming and markov processes. ronald a. howard. technology press and wiley, new york, 1960. viii + 136 pp. illus. $5.75. *Science*, 132(3428):667–667, 1960.