Final Project Report for CIS 419/519 Introduction to Machine Learning

Lakshay Sharma LAKSHAYS@SEAS.UPENN.EDU
Mona Lee MONALEE@SEAS.UPENN.EDU
Sean Waxman-Lenz SWAXLENZ@SEAS.UPENN.EDU

Abstract

We created a machine learning algorithm that outputs an AI to play Super Mario Bros. This AI is a deep double Q-learning agent, with the goal to have Mario make it as far right in the first stage of the game as possible on a single life. The optimal training of such an agent is the focus of this project.

1. Problem and Motivation

For one of the most classic games Super Mario Bros, our goal is to create an AI Mario to play and make it as far in the game as possible on a single life. Super Mario Bros is a unique game, due to both its inherent simplicity that lends itself well to Reinforcement Learning (RL) gym environments, as well as its subtle difficulty that reveals more challenges the deeper one delves into actually implementing an RL agent for it. Such a setup is naturally ideal for the scope of this project, and hence the choice of the game is justified.

RL is a very active branch of machine learning research that focuses on application of learning methods to real-life problems where future predictions need to change based on the outcomes (or "rewards") of past predictions. Applications for reinforcement learning can be found everywhere in solving modern computer science problems. With the relevance of RL to solving real problems, our research into how an agent learns this game can give insight into what types of reward functions work best, how to adjust learning rates, or how 'adventurous' to make the agent. It also provides an interesting example with mass appeal to demonstrate the power of ML and potentially create greater interest across the population.

University of Pennsylvania, CIS 419/519 Course Project.

1.1. Application

This RL agent is very effective to act as a testbed for directly applying the recent insights made in this field by relevant research. In addition, we will gain a lot of skills related to practical reinforcement learning that we can then use directly to train much more advanced agents for novel problems and environments, like the recent agent for Super Smash Bros. Melee developed by MIT PhD students that beat some of the world's best human players. This project will therefore be a very powerful and effective augmentation to our machine learning education.

1.2. Related Work

The related work in this area has been mainly done by Google's DeepMind, which works on researching and developing AI systems and has created a deep learning model to successfully perform in seven Atari games (Mnih, Kavukcuoglu, et al. Nature 2015 [1]). Although they didn't develop a model for Super Mario Bros. specifically, they were the group that initially developed reinforcement learning for game play by training a deep neural network to act as a function approximator for the Q-function that is fundamental to Q-learning. (hence the term deep Q-learning)

2. Approach

We employ and combine various kinds of reinforcement learning algorithms such as Q-learning, double Q-learning and deep Q-learning, to formulate a single solution that works best for the given problem. In addition, we will experiment by varying our rewards system to identify the best reward scheme for training such an RL agent.

2.1. Reinforcement Learning

In this project, we consider the standard reinforcement learning setting where an agent interacts with an environment ϵ over a number of discrete time steps and at each time step *t*, the agent receives a state s_t and selects an action a_t from the set of all possible actions. The agent decides which action to take based on a policy π that maps

from states to actions. After performing the recommended action, the agent receives the next state s_{t+1} and a reward value r_t, which is what will be used to determine success. At the end of a run (when time steps reaches its designated limit or the agent reaches a terminal state), the utility or total reward is calculated as

$$U([r_1, ..., r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_{t+1} \tag{1}$$

where γ ranges from 0 to 1 applying a discount for rewards at later times. The goal of the agent is to maximize the expected return from each state s_t.

The value of a state s under policy π is calculated by the value function

$$V^\pi(s) = \mathbb{E}(\sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid S_0 = s) \tag{2}$$

and indicates the expected return for following policy. from state s_t. The Q-value of taking action a in state s then following policy π is calculated by the action-value function:

$$Q^\pi(s) = \mathbb{E}(\sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid S_0 = s, A_0 = a) \tag{3}$$

To obtain the maximum action value for state s and action a achievable by any policy, we use the optimal value function:

$$Q^*(s) = \max_a Q^*(s, a) \tag{4}$$

The optimal action is therefore the one that yields this maximum Q value from the current state. This serves as the foundation for Q-learning. However, since we do not know the exact value that any action will yield for the future, we need to use RL methods to train an agent to best approximate the Q function. There are two ways to do this: direct Q-leaning and deep Q-learning.

If the Q function is stored directly as a tensor with as many entries as there are possible actions for each possible state, then we can train the values in the tensor directly using the Bellman equation (below) for updates, and this is known as Q-learning.

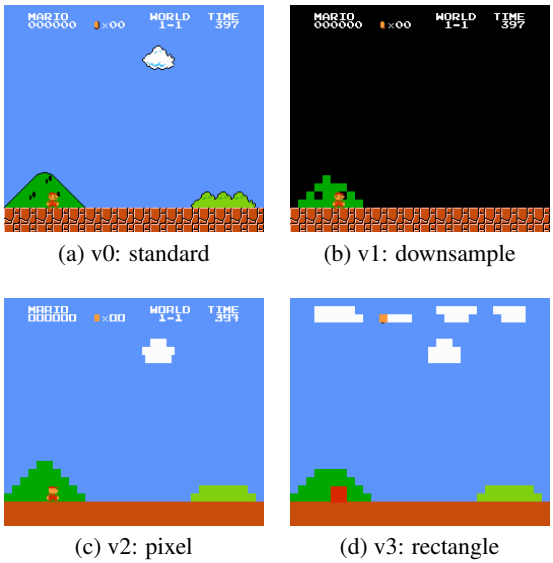
$$Q^\pi(s, a) = r + \gamma Q^\pi(s', \pi(s')) \tag{5}$$

However, sometimes the Q-value function can be too difficult to store. (mainly due to a large number of states, as is the case in this situation) Then, we invoke the rationale that neural networks work magnificently as universal function approximators, and we arrive at the concept of deep Q-learning. Here, the values are determined via deep neural networks, and it is the weights of the layers that are trained to approximate the true value function as closely as possible. Here, a convolutional neural network (CNN) will be used due to its clear advantages in dealing with image frames that we use as the states for the gym.

There are a couple final touches needed before this method can be called perfect for this problem: first off, instead of updating the CNN weights after every state update, it is essential to have a buffer known as experience replay memory that is randomly sampled from every few steps to update the weights of the CNN. This is done to prevent biases entering the agent due to correlations between successive state transitions. Secondly, we will be using deep double Q-learning, as we will have two CNNs: one target network and one online network. The online network will have its weights regularly updated, whereas the target network will copy over the online network's weights at regular intervals. This prevents local optimum solutions from being selected over and over again due to overestimation of the Q values.

2.2. Environment

Our environment for this project is centered around OpenAI Gym. We use a custom gym created to play Super Mario Bros., and the gym itself gives us a lot of options to experiment with to maximize training speed and efficiency. We restricted the gym to only work with stage 1-1 of Super Mario Bros. on a single life, to suit the goals described above. We then made use of the following different image output types provided by the environment to discover the differences between them with respect to out training regimen.



(a) v0: standard (b) v1: downsample

(c) v2: pixel (d) v3: rectangle

Figure 1. The various environments used for training.

2.3. Algorithms

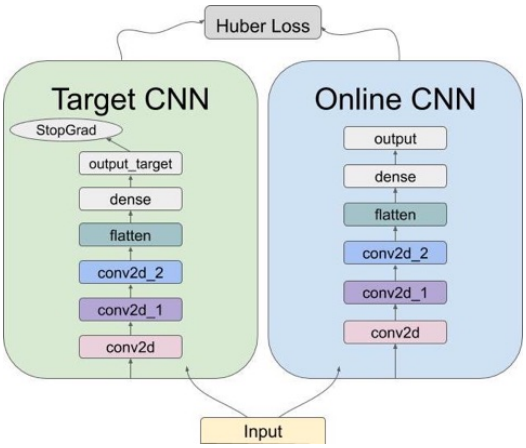
We start with an Epsilon-Greedy Deep Double Q-Learning model. As the state/observation output of the gym at each step, we obtain a 240x256x3 tensor, with each value going from 0-255. This tensor represents the image, pixel by pixel, at the current frame of the game, and is processed as described into the next section and fed into the model CNN. Our goal for the model is to predict the Q-values for the current state. We then use the methods taught to us (and described in other sections) for both CNNs and DQNs to train our model effectively.

Algorithm 1 Q-learning: Mario

Input: states X, actions A, Reward function R, learning rate α, discounting factor γ
function QLEARNING(X, A, R, α, γ)
  Initialize $Q : X \times A \leftarrow \mathbb{R}$ arbitrarily
  while Q is not converged **do**
    Start in state s $\in X$
    while s is not terminal **do**
      Calculate π according to Q and exploration strategy (e.g. $\pi(x) \leftarrow argmax_a Q(x, a)$)
      $a \leftarrow \pi(s)$
      $r \leftarrow R(s, a)$
      $s' \leftarrow T(s, a)$
      $Q(s', a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \gamma max_a Q(s', a'))$
      $s \leftarrow s'$
    end while
  end while
  return Q
end function

2.4. Model Architecture

The policy function approximator is a simple feed forward neural network. We have two copies with the same structure to represent the online and the target network, but the target network is not allowed to backpropagate the gradient since it does not directly update to optimize the loss. The online network has the usual loss and gradient propagation characteristics. The input to each of the networks is a simple [84x84x4] tensor, since we scale each frame of the game to an 84x84 size to speed up training, and then stack the grayscale versions of the frames from 4 consecutive states to get a single input tensor for the neural network. The CNN consists of 3 convolutional layers with kernel strides 8, 4 and 3, strides 4, 2 and 1 and filters 32, 64 and 64 respectively. All three have relu activations. Then we have a flatten followed by a fully connected layer with 512 neurons and relu activation. This leads to the output layer, with 6 neurons for each of the 6 actions in the action space chosen. (RIGHT_ONLY) This structure is outlined below:



2.5. Optimization

The loss function used in our optimization process is the Huber loss, which acts as a mean square loss for small deviations, but keeps outliers from interfering by acting as an absolute value error for large deviations. We run gradient descent on the online network to tweak the weights so as to minimize this loss so as to best approximate the true value function for the environment. The value of γ used is 0.9, and the learning rate is 0.00025. We also make use of adaptive moment estimation (ADAM) in our optimization logic.

2.6. Reward Function

The reward function provided by the environment was found to be detailed enough that the sign itself was enough information to train the agent, and the value itself was found to slow the training process much more than it helped with the accuracy. Therefore, we clipped the reward using the sign function in a reward wrapper for the environment, and ensured that only rewards of -1, 0 or 1 are possible.

2.7. Training

Training is usually required to be ran for atleast a minimum of 500 episodes, and ideally 1000 or more to see useful results. The starting value used for ϵ is 1, and it decays muliplicatively by a factor whose ideal value was experimentally found to be 0.999975. The batch size for retrieval from experience replay memory was taken to be 32 and the target network copies over the online network's weights every 10000 steps. These values together make an ideal starting set of hyperparameters that can be tweaked.

3. Results

We found the following results during our experiments:

- First off, we discover that the most optimal environment to train in is clearly v1 (downsample). v1 not only is faster but also more accurate to train in than the other environments, and the average reward increases the fastest and reaches the highest eventual value for this environment in particular. This is contrary to our original idea that v3 (rectangle) would be the fastest environment to train in. We think that this could be due to the unique black background provided by v1 in stage 1-1, which likely causes the CNN to be more effective at detecting changes between state inputs and responding appropriately, due tohigher contrast.

- Secondly, we were not able to get Mario to finish stage 1-1 on a single life, but we came very close and made quite a lot of progress in the map. With all the insights we achieved about the problem, we would certainly have completed the stage as well as the game if we had more time.

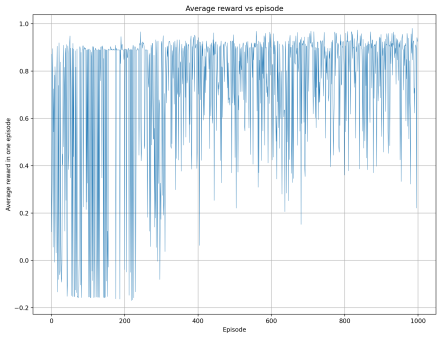
- Third, we noticed a very interesting trend with respect to the rewards function. The default environment comes with a detailed rewards scheme that ranges from -15 to 15 as a result of being a sum of three separate numbers. However, we found that clipping this information about the reward using the sign function on the reward and making it so that all rewards are either -1, 0 or 1 does not affect accuracy too much while speeding up training noticeably.

- Fourth, we noticed that a big hurdle that the agent was having was not being able to jump over pipes. We tried a lot of different methods from changing the rewards scheme to be very punishing for zero x-velocity between states to scaling up the difference between highest and lowest reward attainable, but not much improvement was seen. However, we were able to greatly improve the success rate of the jumps over pipes by fine-tuning the decay rate of ϵ and increasing the number of training episodes, and managed to overcome this, as stated in the last point.

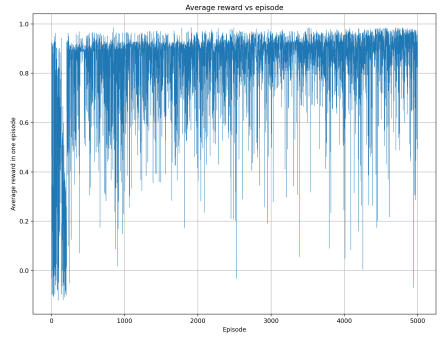
- Fifth, we determined that our final method of sampling the frames after 4 steps, converting them to grayscale and collecting them into a single tensor state was very effective at not only speeding up the training hugely, but also helped prevent correlation bias, since two immediately consecutive frames very rarely have much new information between the two of them.

- Lastly, we found that in the most trained versions of the agent, it started showing very intelligent behavior. First, we noticed that it had learned to regularly slow down and jump carefully whenever it saw a gap in the floor where it could fall and die. Next, the agent managed to hit a hidden question block in a couple of its runs, surprising us all. (a hidden question block is one which is transparent until Mario hits it by jumping, and it usually gives a powerful powerup) In addition, this smart Mario almost never hits an enemy that is approaching him head-on, almost all his enemy deaths are due to him landing next to one after a jump and inevitably dying in the next frame. In addition, the pipe stalls occur very rarely if at all. This is proof that the agent has learned a lot from its mistakes. This version of the agent took 5000 episodes and was trained in v0 (standard) for visual fidelity.

Some plots we obtained for these results are as follows:



(a) 1000 episodes on v1



(b) 5000 episodes on v0

Figure 2. Growth of reward obtained in each episode.

4. Conclusion

We learned a lot from this project, and are determined to keep researching this problem in particular and DQN in general to test our limits and break past them!

Acknowledgments

A huge thank you to the professors and the teaching staff for making this possible!

References

[1] Vlad Mnih, Koray Kavukcuoglu, et al. Nature (2013, January 1). Playing Atari with Deep Reinforcement Learning. Retrieved from https://deepmind.com/research/publications/playing-atari-deep-reinforcement-learning

[2] Kauten, C. (2018). gym-super-mario-bros. Retrieved from https://pypi.org/project/gym-super-mario-bros