Reinforcement Learning in Super Mario Bros.

CS 182 Final Project Report

Gabe Grand and Kevin Loughlin

December 12, 2016

1 Introduction

For our final project, we developed a reinforcement learning AI to play World 1-1 of the original Super Mario Bros (SMB). SMB is a classic platform video game released by Nintendo in 1985. From an AI-theoretic perspective, SMB poses several interesting challenges. First, the state space is vast; within a single level, Mario encounters a wide variety of obstacles, enemies, and terrain. Second, many elements of the state are not fully observable, given the state representation (which is discussed in Section 3). Finally, it is difficult to define and produce a singular optimal policy (if one even exists), since the game incorporates several interrelated but distinct goals. Mario must reach the end of the level as quickly as possible, but various actions, like collecting coins and defeating enemies, can contribute to the score.

Our project relies on two main resources. The first of these is OpenAI Gym, a toolkit for developing and comparing reinforcement learning algorithms [1]. Our second primary tool is developer Philip Paquette's Gym Super Mario, which serves as an interface between the OpenAI Gym and a SMB emulator [2]. The Gym Super Mario environment provides a state space representation, action space representation, and basic reward function for the SMB environment. As we have learned in this course, these three features comprise the fundamental aspects of a reinforcement learning (RL) problem.

Rather than frame the problem as a Markov Decision Process, we decided to use model-free RL algorithms. This decision was motivated by the fact that we did not have access to the underlying model of Mario's movement (namely, the transition probabilities between states). Without this information, we concluded that it would be difficult to frame SMB as an MDP. Model-free RL, on the other hand, provides a tractable, sample-based method of learning the values of states and/or actions online.

We applied three RL algorithms to SMB, adopted from the versions found in the course textbook [3]. We began with Exact Q-learning, which learns the value of state-action pairs based on the full state representation of the game. We quickly realized that this approach involved far too large of a state-action space to achieve learning in computationally-feasible time. We then turned to approximate Q and approximate SARSA learning, in which we approximated Mario's state as a collection of features.

Ultimately, the project proved much more complex than we initially expected. The emulator runs in close to real time, which has significantly constrained our ability to train our agents and tune the hyperparameters associated with their behavior. We attempted to mitigate this problem by using Harvard's Odyssey cluster, but unfortunately, we were unable to install all of the required software. We have also spent extensive time on feature engineering, but still feel that there is room for improvement in this area, as Mario does not converge to a policy that reliably beats World 1-1.

Despite these shortcomings, we are nevertheless satisfied by the progress that we have made. We were able to build on the RL algorithms used in class and bolster them with new material, including eligibility traces and SARSA. Promisingly, our algorithms reliably reach the checkpoint half way through the level. To our knowledge, ours is the first concerted effort by a development team to solve the OpenAI Gym Super Mario environment. Our project provides a substantial codebase for applying various RL techniques to SMB. It is our hope that our work will serve as a contribution and a model to other developers who approach this problem in the future.

2 Background and Related Work

Recently, work with retro video games, such as those of Nintendo [4] and Atari [5, 6], has been a major trend in RL literature. These games are simple enough to be computationally-tractable, yet often involve highly-intricate strategies.

In particular, Mario has proved to be of particular interest within the AI community. In 2009, in collaboration with IEEE, AI researchers held a Mario AI Competition [7, 8]. While competitors employed a variety of machine learning techniques, including RL and genetic algorithms [9], it is intriguing to note that the top team used an A* search [8].

The most useful source for our project has been the work of Yizheng Liao, Kun Yi, and Zhe Yang of Stanford on their CS229 Final Report [10]. The team developed an approximate Q-Learning agent for SMB that beats its level about 90% of the time. The descriptions of their model's features provided in their paper have informed our own feature-engineering decisions.

3 Problem Specification

The goal of Super Mario Bros is relatively simple: beginning on the left edge of a level, navigate to the flagpole without dying, while maximizing the score. Our project's goal is to create an AI agent that successfully navigates through a single level: World 1-1.

In order to achieve success, Mario generally must move right; however, only moving right would be a far cry from an optimal policy. In order to behave optimally, Mario must avoid and/or kill enemies, jump over cliffs and pipes, collect mushrooms and coins, and adhere to the time limit. Needless to say, these challenges significantly complicate the task at hand.

To formulate SMB as an RL problem, we must specify three main components, which were briefly discussed in the introduction: the state space, the action space, and the reward function.

3.1 State Representation

Mario's state is represented as a 13 by 16 tile grid of the numbers 0-3, where 0 represents empty space, 1 represents an object (such as a coin, ground, pipe, etc.), 2 represents an enemy, and 3 represents Mario.

3.2 Action Space

The action space is based off of the Nintendo controller, which has 6 buttons that either pressed (1) or not (0). This would result in $2^6 = 64$ possible actions. However, only 14 of these combinations make logical sense, and only 9 of the 14 have an effect on the game. For example, the combination left and right does nothing, as Mario simply remains in place. As another example, the up button theoretically performs an action in the game, but in practice has no effect. We therefore limited our action space to the following 9 actions, where A is jump and B is sprint:

```
actions = { Left, Left + A, Left + B, Left + A + B, Right, Right + A,
Right + B, Right + A + B, A }
```

3.3 Reward Function

The reward function is the principal instrument that we use to shape Mario's behavior. The Gym Super Mario environment provides a simple default reward function, which is proportional to Mario's distance δ in the level.

$$R(s) = \delta(s) - \delta(s-1)$$

Thus, Mario gets +1 for moving one step to the right, -1 for moving one step to the left, and 0 for standing still. Since we wanted to encourage Mario to consistently make forward progress, we slightly altered the reward function to penalize Mario for standing still.

$$R(s) = \begin{cases} \delta(s) - \delta(s-1) & \text{if } \delta(s) - \delta(s-1) > 0 \\ \delta(s) - \delta(s-1) - 1 & \text{if } \delta(s) - \delta(s-1) \leq 0 \end{cases}$$

In addition to calculating the reward based on distance, we also added large rewards based on the occurrence of specific events. The actual values of these rewards involve arbitrary hyperparameters, which we tuned subjectively. When Mario dies, we subtract 100 from the reward. When Mario's score increases by x (from collecting an item or killing an enemy), we add σx to the reward, where σ is the "score factor" hyperparameter. Finally, when Mario jumps over a particular large gap that occurs half-way through the level we add 500 to the reward.

4 Approach

We implemented three variants of Q-Learning: Exact Q-Learning, Approximate Q-Learning, and Approximate SARSA (State-Action-Reward-State-Action). We implemented these agents as Python classes. The approximate SARSA inherits from Approximate Q agent, and all three agents inherit from an abstract class that exposes a standardized interface. These algorithms were all implemented using standard data structures like lists and dictionaries.

The general idea of Q-learning is as follows: beginning in some state s, Mario takes action a to arrive at s', receiving reward r'. We then update the Q-value associated with (s, a). The main difference among these algorithms is how and when these updates are performed.

4.1 Exact Q-Learning

While Exact Q-Learning is guaranteed to converge, it is unsatisfactory for our problem because of the sheer size of the state-action space. A back-of-the-envelope calculation reveals that, with four possible values for each tile (0-3), 13 times 16 tiles, and 9 possible actions, the number of state-action pairs is extremely high:

$$4^{16*13} * 9 \approx 1.52 * 10^{126}$$

In practice, we found empirically that Mario only deals with about 100,000-500,000 unique states per level. However, given our training time limitations and the strain on the emulator environment, the size of the state space proves to be a significant limitation to the effectiveness of Exact Q-Learning.

Algorithm 1 Exact Q-Learning algorithm (Adapted from [3])

```
procedure EXACTQ($percept$)
inputs: $percept$, a percept indicating the current state $s'$ and reward signal $r'$
persistent: $g$, an exploration function
$\epsilon$, the probability of a random move, decreasing from 1 to 0.05 based on iteration
$Q$, a table of action values indexed by state and action, initially zero
$N_{sa}$, a table of frequencies for state-action pairs, initially zero
$s, a, r$, the previous state, action, and reward, initially null
    if Terminal?($s'$) then $Q[s', None] \leftarrow r'$
    if $s$ is not null then
        increment $N_{sa}[s, a]$
        $Q[s,a] \leftarrow Q[s,a] + \alpha(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$
    $s, a, r \leftarrow s', g(\epsilon_i, \arg\max_{a'} Q[s',a'] + \frac{10}{N_{sa}[s',a']+1}), r'$
    return $a$
```

4.2 Approximate Q-Learning

Given that the full space space is too large, we needed to find a way to generalize across different states in order to significantly reduce the number of states. Approximate Q-Learning does exactly this. In approximate Q-Learning, we engineer a set of features to approximate each state. Then, we can express the Q-value as a linear combination of these features:

$$Q(s,a) = w_1 f_1(s,a) + \ldots + w_n f_n(s,a)$$

Algorithm 2 Approximate Q-Learning algorithm

```
procedure APPROXQ(percept)
inputs: percept, a percept indicating the current state $s'$ and reward signal $r'$
persistent: $g$, an exploration function
$\epsilon$, the probability of a random move, decreasing from 1 to 0.05 based on iteration
$F$, a set of $n$ feature functions that returns a feature value for a state action pair
$W'$, a set of $n$ weights for each feature function, initially all zero $N_{sa}$, a table of frequencies for state-action pairs, initially zero
$s, a, r, W$, the previous state, action, reward, and weights, initially null
    if Terminal?($s'$) then $\max_{a'} W \cdot F(s', a') \leftarrow r'$
    if $s$ is not null then
        increment $N_{sa}[s, a]$
        for $i = 1$ to $n$ do
            $W'[i] \leftarrow W'[i] + \alpha(r + \gamma \max_{a'} W \cdot F(s', a') - W \cdot F(s, a))F[i](s, a)$
    $s, a, r, W \leftarrow s', g(\epsilon_i, \arg\max_{a'} Q[s', a'] + \frac{10}{N_{sa}[s', a'] + 1}), r', W'$
    return $a$
```

Here, we provide a description of the features we used, as well as our motivation for including each feature. Note that, besides horizontal velocity, all features are binary. We attempted many features not included in this list, but ultimately found that (a) binary features made life easier in terms of weights converging (which they are not guaranteed to do in approximate Q learning), and (b) yielded the most logical, comprehensible model — in some ways reflecting a decision tree.

- *Horizontal Velocity*: the number of tiles Mario has moved in the x direction from the previous state, where right is positive and left is negative. This feature thus serves the dual purposes of taking direction of movement and speed into account. Ideally, Mario will learn that moving quickly to the right (i.e. having a large, positive velocity) will do well for him.

- *Run Action*: whether the current action being taken involves pressing the B (sprint) button. We want Mario to learn what situations are best for running, such as when he must run to avoid an enemy or to be able to clear an osbtacle.

- *Jump Action*: whether the current action being taken involves pressing the A (jump) button. Knowing when to jump is critical for clearing obstacles, fetching coins, and stomping on enemies.

- *Right Action, Left Action*: whether the current action being taken involves pressing the right/left button. Our hope is that right will be associated a positive weight, and left will be associated with a negative weight, as only right is generally associated with progress in the game.

- *Stuck*: whether Mario is able to move right in his present state. This is useful to Mario for determining when he must jump — if he cannot move right, clearly he must change his vertical position.

- *Gap Below, Gap Right Far, Gap Left Far, Gap Right Near, Gap Left Near*: whether there is a gap below/right/left of Mario that is within 20% or 8% of screen distance away (where these numbers were most effective based from trial and error). For all of the gap features, it is useful to know the relative closeness of a gap for Mario, so that he may determine when is best to jump over the gap.

- *Enemy On Screen*: whether an enemy is presently on screen. Having enemies on screen presents a potential for death, but also for points (via killing them).
- *Can Stomp Enemy*: whether there is an enemy 2 spaces or fewer directly below Mario. This is Mario's primary method of eliminating enemies.
- *Enemy Danger Right, Enemy Danger Left*: whether an enemy is within 20% of screen distance to the right/left of Mario. Both enemy danger features are useful for Mario's planning as to where he needs to move to avoid or kill the enemies.

4.3 Approximate SARSA

While we believed Approximate Q-Learning to be a sufficient methodology, we also wanted to explore a slight variant. Approximate SARSA is almost identical to Approximate Q Learning, except for one small difference. While Q-Learning is an off-policy learning algorithm, SARSA is an on-policy learning algorithm. Q-Learning backs up the maximum Q-value from s', whereas SARSA waits until an action is actually taken from s' and backs up the Q-value for that state-action pair [3]. What this boils down to is that the exploration stages for the algorithms are quite different. The Q-Learning agent might be more flexible, but our hope is that a sufficiently good initial policy for the SARSA agent will allow the weights to converge more quickly, avoiding fruitless exploration.

Algorithm 3 Approximate SARSA algorithm

```
procedure APPROXSARSA(percept)
inputs: percept, a percept indicating the current state $s'$ and reward signal $r'$
persistent: $g$, an exploration function
$\epsilon$, the probability of a random move, decreasing from 1 to 0.05 based on iteration
$F$, a set of $n$ feature functions that returns a feature value for a state action pair
$W'$, a set of $n$ weights for each feature function, initially all zero $N_{sa}$, a table of frequencies for state-action pairs, initially zero
$s, a, r, W$, the previous state, action, reward, and weights, initially null
    $a' \leftarrow g(\epsilon_i, \arg\max_{a'} Q[s', a'] + \frac{10}{N_{sa}[s', a'] + 1})$        ▷ Choose next action
    if Terminal?($s'$) then $W \cdot F(s', a') \leftarrow r'$
    if $s$ is not null then
        increment $N_{sa}[s, a]$
        for $i = 1$ to $n$ do
            $W'[i] \leftarrow W'[i] + \alpha(r + \gamma(W \cdot F(s', a')) - W \cdot F(s, a))F[i](s, a)$        ▷ No max
    $s, a, r, W \leftarrow s', a', r', W'$
    return $a$
```

4.4 Exploration function

Across all of these algorithms, we employed an exploration function g. The purpose of g is to balance taking the greediest possible action (i.e. the action that maximizes our current estimate of $Q[s', a']$) versus an action that has not been (extensively) explored. Any exploration function should thus increase with Q and decrease with N. Perhaps the simplest example is

$$f(Q[s', a'], N_{sa}[s', a']) = Q[s', a'] + \frac{k}{N_{sa}[s', a'] + 1}$$

where k is a hyperparameter constant (which we set to 10 based on testing different values), and 1 is added to avoid division by 0. In other words, rather than simply choosing the action with highest Q value early on, Mario is more likely to choose actions that have not been taken from the current state, as this indicates a low N and thus larger second term.

However, in an effort to make Mario's Q-values converger faster, and to manipulate his behavior, we wanted to include random actions in his repetoire, based from a prior distribution that we believed could serve him well in the game (namely, go fast, go right, and jump, with more likelihood than other actions). Thus, with probability ϵ (where ϵ starts as 1, but decreases over time to a minimum value of MIN_EPSILON $= 0.05$ based on the function $\epsilon_i = \max(0.05, \frac{A}{A+i}$, and where $A = 500$), Mario will take a random action based from the following prior distribution (which is normalized in our code)

$$PRIOR = \{\textbf{left} : 2,\ \textbf{left-jump} : 2,\ \textbf{left-sprint} : 2,\ \textbf{left-jump-sprint} : 2,\ \textbf{right} : 5,$$
$$\textbf{right-jump} : 5,\ \textbf{right-sprint} : 20,\ \textbf{right-jump-sprint} : 10,\ \textbf{jump} : 2\}$$

Thus, our final exploration function is as follows:

$$g\left(\epsilon_i, \arg\max_{a'} Q[s', a'] + \frac{k}{N_{sa}[s', a'] + 1}\right) = \begin{cases} \arg\max_{a'} Q[s', a'] + \frac{k}{N_{sa}[s', a'] + 1} & \text{with probability } 1 - \epsilon_i \\ a' \in PRIOR & \text{with probability } \epsilon_i \end{cases}$$

4.5 Eligibility Traces

When Mario gets a reward, the approximate Q update modifies the weights of the features that are currently active. However, because the game states are causally-dependent and sequential in nature, it makes sense to assign credit for a reward (or equivalently, blame for a punishment) to previous states, in addition to the current state.

In order to address this credit-assignment problem, we incorporated eligibility traces into both of our approximate learning algorithms. With eligibility traces, our approximate Q algorithm becomes approximate Q(λ); likewise, approximate SARSA becomes approximate SARSA(λ). In addition to updating the weights based on the features for the current state, we also iteratively apply updates based on the features for each of the previous states. The magnitude of these updates is determined by a parameter λ. If $W'[i_t]$ is the normal weight update based on the features at timestep t, then the new weight update is given below. To eliminate inconsequential computations, we only calculate $W'[i_t]$ for the n most recent timesteps. The sequence of arrows represents iterative updates to $W'[i_t]$.

$$W'[i_t] \leftarrow W'[i_t] \leftarrow \lambda W'[i_{t-1}] \leftarrow \lambda^2 W'[i_{t-2}] \leftarrow \ldots \leftarrow \lambda^n W'[i_{t-n}]$$

The use of eligibility traces helps to solve the credit-assignment problem by taking into account recently-active features. It also has the additional benefit of increasing the rate at which the weights converge. However, the use of eligibility traces has a computational tradeoff, since we now must compute n weight updates at every timestep.

5 Experiments

We ran each of our three RL agents (Exact Q, Approximate Q, and Approximate SARSA), as well as three benchmark agents (described below), for 100 training iterations on World 1-1. We started each algorithm started from scratch (i.e. no learning), since we wanted to examine how the agent's learning evolved over time. The number of training iterations was selected to accommodate testing all six agents given our computational constraints. Although we had previously trained our RL agents for as many as 1000 iterations, we found that our algorithms generally converged within 100 iterations, and that (aside from the Exact Q Agent) the additional training did not contribute to any significant improvement in performance. Both of these results are discussed in the Results and Discussion sections.

Since we did not have the ability to systematically test different combinations of hyperparameters for each agent, we instead selected hyperparameters qualitatively, based on our experience running the model. The agents were initialized with the following (main) hyperparameters, which were selected with the following considerations in mind. In addition, a half-dozen other hyperparameters were used to control values like the penalty for Mario dying, and the proportion of the game score (e.g., due to collection of items) to incorporate into the reward function.

- α **(learning rate):** 0.1 for Exact Q, 0.01 for Approx Q and Approx SARSA. Both settings of α were deliberately low in order to prevent over-learning. The latter was an order of magnitude lower in order to prevent the weights from diverging.
- γ **(discount factor):** 0.95 for all agents. State sequences are causally-dependent, and the game runs in 30 FPS. Thus, a high γ is necessary to ensure that actions are selected with future states in consideration.
- ε**-min (minimum random move probability):** 0.05 for all agents. Regardless of the number of times that a state has been previously explored, it makes sense to take a random action 1/20 times, to ensure that Mario does not get stuck.
- λ **(eligibility trace decay):** 0.8 for Approx. Q and Approx. SARSA (N/A for Exact Q). Again, consecutive game states are highly dependent, so a high value of λ will ensure that credit is appropriately assigned to recent previous states.

In addition to the three algorithms described in Section 4, we also tested three simple benchmark algorithms. RandomAgent samples from a uniform prior over all 9 possible actions. WeightedRandomAgent samples from a weighted prior in Section 4.4. Finally, HeuristicAgent also samples from the weighted prior. However, HeuristicAgent follows several additional hard-coded rules which we engineered in order to get Mario to avoid gaps and enemies.

5.1 Results

We calculated the mean and maximum distances achieved by each agent over 100 iterations. Progress through the level was reset after every iteration. However, it should be noted that once Mario reached a checkpoint that was approximately half-way through the level, all subsequent iterations restarted at a checkpoint. Except for the RandomAgent, all agents managed to reach the checkpoint.

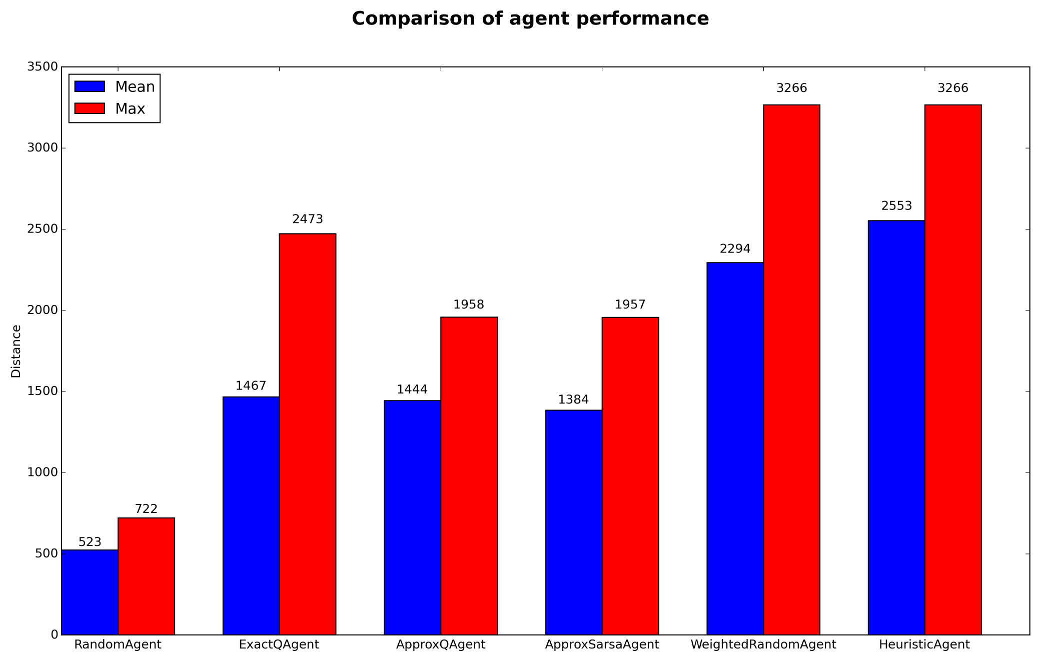

Figure 1: Comparison of agent performance, showing the mean (blue) and maximum (red) distance achieved in 100 iterations.

As Figure 1 reveals, all three Q-learning agents significantly outperformed the RandomAgent. However, none of the agents reached the end of the level. In general, the Q-learning agents had difficulty reliably clearing a large gap that occurred at distance 1400, soon after the half-way checkpoint. Meanwhile, both the WeightedRandomAgent and the HeuristicAgent each reached the end of the level exactly once. The HeuristicAgent, which had specific rules for avoiding enemies and jumping over gaps, slightly outperformed the WeightedRandomAgent.

Before running these tests, and on first examination of these results, we were concerned that the number of training iterations was not sufficient to allow the Q-learning algorithms to converge to a sufficient policy. As we have noted, 100 iterations is far fewer than we would have liked, but it was the best we could do in terms of computational time limits. We therefore set out to quantify the learning of each agent to assess convergence.

We began with the ExactQAgent, examining the number of unique states learned over the course

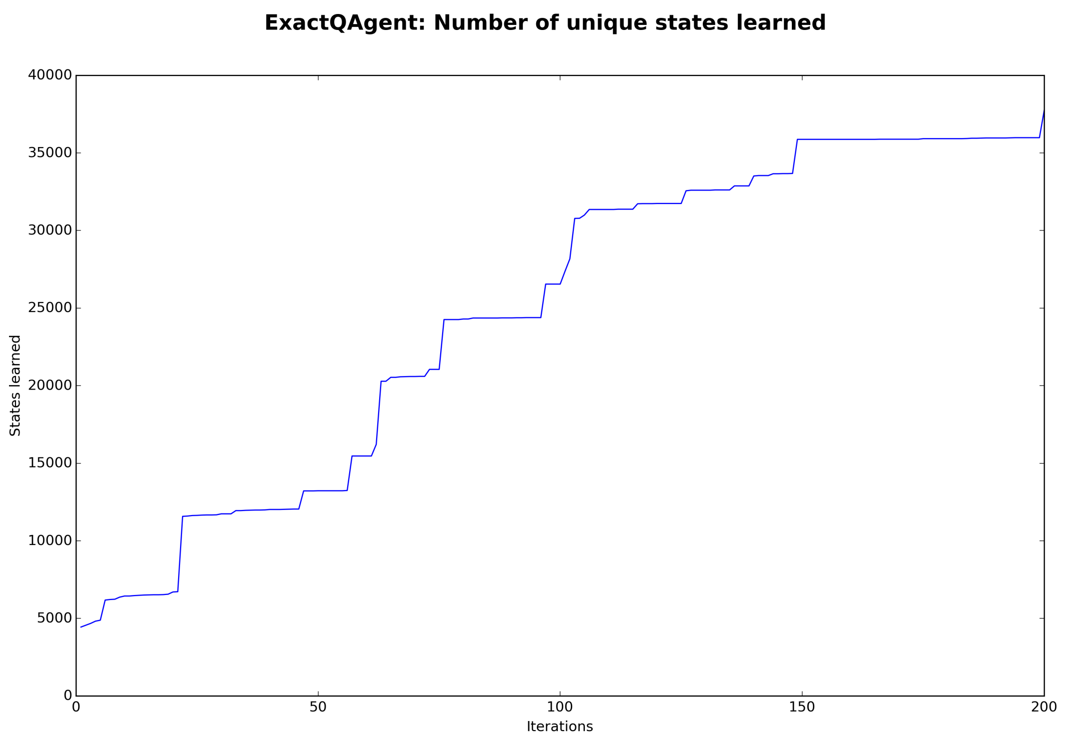

Figure 2: Number of unique states learned by the ExactQAgent in 200 iterations. Notably, learning appears to level off after iteration 150.

of 200 training iterations (Figure 2). We found that this number increased roughly linearly for approx. 150 iterations, after which point it leveled off. Therefore, we acknowledge that 100 iterations was indeed insufficient for Exact Q Learning.

However, in the case of the Approximate Q Agent, we observed that the weights assigned to each feature generally converged within 100 iterations (which admittedly, may have been influenced by Mario regularly dying at the midway checkpoint). The convergence graph (Figure 3) is characterized by an initial learning period, in which the weights fluctuate dramatically, followed by a long period of stabilization. (The graph of the weights for the Approximate SARSA agent was very similar to that of the ApproximateQAgent, so we did not include it in this report.) From these results, we concluded that additional training iterations would not have been likely to produce a significant increase in performance for the approximate Q agents.

As further proof of this theory, we ran the ApproximateQAgent for 1000 consecutive training iterations on World 1-1. As expected, the weights converged after fewer than 100 iterations. However, the performance did not significantly improve; approximately 9 times out of 10, Mario was unable to clear the gap that occurs after the checkpoint.

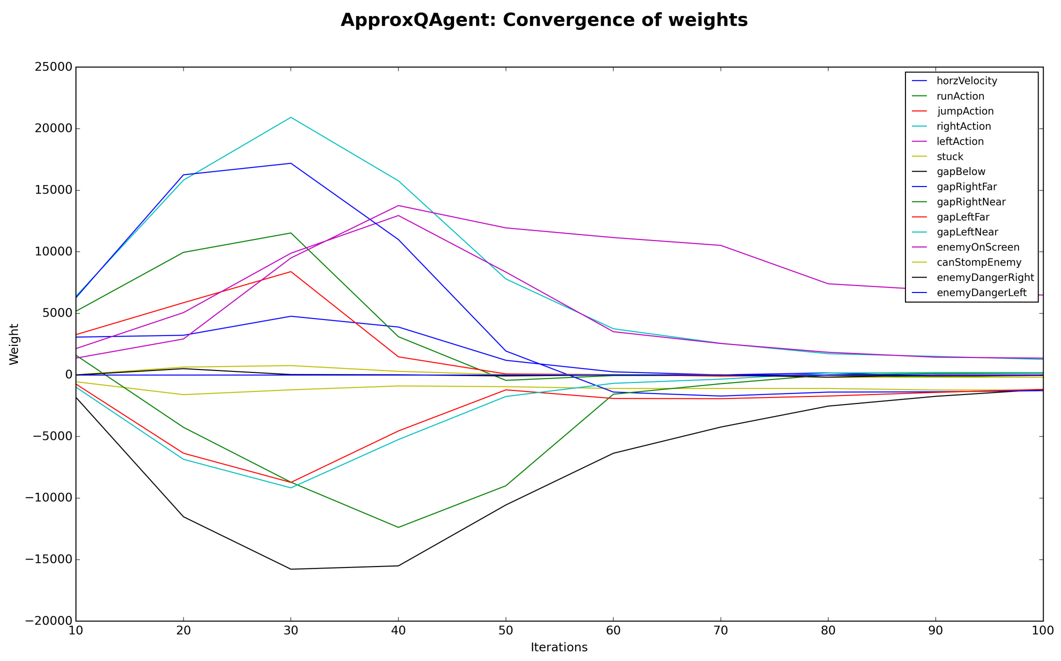

Figure 3: Convergence of weights in the ApproxQAgent. Weights undergo significant fluctuations early on before converging within 80 iterations.

5.2 A note on computational resources

Computation time presented a significant barrier to the evaluation of our model's performance. Due to software limitations in the speed of the FCEUX emulator, training currently must occur in approximately real-time. Though we worked directly with the developer of the environment, we were unable to increase the emulation speed. As a result, we found that a standard round of 1000 training iterations took up to several hours, depending on the level. Professor Kuindersma was kind enough to provide us access to the Harvard Odyssey computing cluster, but to install the necessary software (FCEUX emulator, Direct X11) would have required system privileges that we did not possess. Ultimately, we were unable to test our model as extensively, and under as diverse hyperparameter settings, as we would have wished.

6 Discussion

Given our result that the ApproximateQAgent converges in under 100 iterations, a central question for us was why the agent was unable to learn a better policy for the level. One major shortcoming of the approximate Q learning paradigm is the assumption of linearity: Q values are assumed to be some linear combination of the weights. In designing our features, we endeavored to pick out aspects of the state that would reliably be associated with positive or negative rewards. As an example, the `rightAction` feature, which is active when Mario takes an action that would lead him to make forward progress, is almost always associated with a positive reward. However, in practice, many features are context-dependent, so it is not realistic to assume that there exists some constant, optimal weight w^* that should be assigned to the feature. For instance, one might expect that the `enemyDangerRight` feature would be associated with a negative reward, since it is active right before Mario collides with an enemy. However, in the case when Mario is able to stomp and kill the enemy, `enemyDangerRight` may actually be associated with a positive reward. Clearly, the agent should take into account the fact that there is an enemy to the right of Mario. However, whether this should be considered “good” or “bad” is highly dependent on the broader context of the game state.

The high degree of variability associated with feature-based learning offers a potential explanation for why Exact Q learning appears to have outperformed the Approximate Q algorithms. Though it requires more computation time, Exact Q learning does not require features. In contrast, our approximate algorithms rely on the strength of the features. With a more comprehensive and consistent feature set, we predict that Approximate Q would vastly outperform Exact Q Learning.

In addition to feature-engineering, another significant challenge that we faced was how to design the reward function to encourage the kinds of complex behaviors necessary to make progress in the level. For instance, in order to clear tall pipes, Mario must either get a running start, or he must hold down the jump button for roughly 25 frames consecutive frames while moving right. Through trial-and-error, the agent is bound to eventually encounter the correct move sequence. Ideally, we would immediately provide the agent with a high reward for clearing the pipe in order to reinforce this behavior. However, we were unsure how to accomplish this without specifically accounting for the locations of various obstacles within the level. Ultimately, we resorted to a patchwork of small “hacks” to help clear tough obstacles – for instance, we hard-coded the exact distance of the notorious gap after the checkpoint, so that we could reward Mario for crossing it. As a last resort, we also added a subroutine to each agent’s action selection to detect when Mario is stuck and to mash the jump button when this occurs.

The process of engineering our features and reward function has made us acutely aware of the extent of the prior knowledge and abilities that we, as humans, bring to complex learning problems. Unlike reflex-based video games such as Flappy Bird or Pong, SMB requires significant pattern-recognition and reasoning skills. For instance, by watching it for a few seconds, we can recognize that a Goomba can only move laterally, and is incapable of jumping into the air. From this information, we can leverage our intuitive physics to plan a sequence of moves that will allow Mario to jump over the Goomba. RL algorithms, on the other hand, come with no prior knowledge about how physical objects behave and interact in the real world. In order to learn and reason about these considerations, the agent would need to be capable of building a model of the game world. Ironically, the very aspects of model-free RL that led us to apply it to SMB also serve as significant limitations on its performance in this domain.

As we mentioned in the introduction, our project was initially born out of the OpenAI Gym framework, which was created to facilitate the development of reinforcement learning algorithms. Just recently, however, OpenAI released a new framework, called Universe, with the stated goal of “measuring and training an AI’s general intelligence across the world’s supply of games, websites and other applications.”[1] Unlike the Gym library, which contains mostly narrow, focused tasks,

[1]https://openai.com/blog/universe/

Universe includes over a thousand environments ranging from browser tasks, to Flash games, to triple-A video game titles like Grand Theft Auto V. We feel that in order to meet the challenges of these new and complex environments, the field of AI will need to undergo a significant evolution. In order to exit the Gym and step out into the Universe, our agents must learn not just to react blindly to rewards and punishments; instead, they must actively build models that allow them to reason deeply about the world.

A System Description

All of the code used in our project is hosted on Github at: https://github.com/kevinloughlin/Super-Mario-RL.

In order to use our system, you must download and install the OpenAI Gym, the FCEUX Nintendo Entertainment System emulator, and the Gym Super Mario environment. We provide instructions for installation and setup on Mac OS X and Python 2.7. In a bash shell, run the following commands.

```
pip install gym
pip install gym-pull
brew upgrade
brew install homebrew/games/fceux
```

Once this is completed, open a python shell and run the following.

```
import gym
import gym_pull
gym_pull.pull('github.com/ppaquette/gym-super-mario@gabegrand')
```

To generate the output discussed in our writeup, navigate to src/. You can set the hyperparameters as you wish in hyperparameters.py, and then run test.py (which will launch the emulator and use our code for training). Finally, you can kill the processes via Control-C in the Python terminal, or running ./mario.sh in the src directory.

B Group Makeup

- **Gabe Grand**: (1) Read background work, (2) Installed Gym Super Mario and handled technical issues, (3) Implemented Q agent framework, (4) Analyzed data from experiments, (5) Wrote Experiments and Discussion sections
- **Kevin Loughlin**: (1) Read background work, (2) Created and tested approximation features, (3) Implemented approximate SARSA agent, (4) Wrote Problem Specification and Approach sections

The only major difference from our original work proposal is that we didn't end up doing POMDP's, since we realized we didn't have access to any sort of transition model. Instead, we added the approximate SARSA agent and went in-depth on improvements to these algorithms, such as better exploration functions and eligibility traces.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016.

[2] Philip Paquette. Gym super mario, 2016.

[3] Stuart Jonathan Russell and Peter Norvig. *Artificial intelligence: A Modern Approach*, volume 3. Pearson Education, Inc., 2010.

[4] Tomás Banzas Illa. Reinforcement learning in an emulated NES environment. 2015.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 2015.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[7] Julian Togelius, Sergey Karakovskiy, and Robin Baumgarten. The 2009 mario ai competition. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[8] Sergey Karakovskiy and Julian Togelius. The mario ai benchmark and competitions. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):55–67, 2012.

[9] Diego Perez, Miguel Nicolau, Michael ONeill, and Anthony Brabazon. Evolving behaviour trees for the mario ai competition using grammatical evolution. In *European Conference on the Applications of Evolutionary Computation*, pages 123–132. Springer, 2011.

[10] Yizheng Liao, Kun Yi, and Zhe Yang. CS229 final report: reinforcement learning to play Mario. Technical report, Technical report, Stanford University, 2012.