Testing the Effectiveness of Various Reinforcement Learning Algorithms on a Game of Infinite Frogger

Gregory Luppescu
Department of Electrical Engineering
Stanford University
gluppes@stanford.edu

Hershed Tilak
Department of Electrical Engineering
Stanford University
htilak@stanford.edu

Abstract—**For this project, we implement a custom version of a game of "infinite" Frogger and test how well different learning algorithms can learn our game. The three algorithms tested are Q-learning with function approximation, SARSA with function approximation, and a genetic algorithm. Two feature extraction schemes are implemented and tested using Q-learning and SARSA, each trained for a total of 150,000 games, and the genetic algorithm is trained for 1000 generations. Results indicate that more expressive feature extractors allow for a more elaborate and intelligent policy, and that given the same feature extractor, Q-learning seems to do slightly better than SARSA. The genetic algorithm also learns to play the game reasonably well in the training time allotted.**

I. Introduction

The game Frogger is a simple 2-D game where a player must guide a frog across roads and rivers to get to its goal. While a good amount of work has been done using reinforcement learning and other techniques to learn to play conventional Frogger, we decided to try a novel approach and make a game of infinite Frogger, where the only goal of the player is to stay alive. This begs the question, which methods are most effective for learning how to play infinite scrolling games? In this paper, we attempt to answer this question by implementing and analyzing various methods for learning how to play our custom game.

II. Related Work

There has been a large amount of work done regarding teaching computers how to play video games. [1] utilizes a modification of Q-learning to use nearest neighbor states to exploit previous experience in the early stages of learning, with the hope that some of the effects of the curse of dimensionality can be mitigated. The work in [2] focuses on using the SARSA technique to learn to play simple 2-D video games, where they focus on creating a system that is capable of general game playing rather than optimizing to play one specific game. One of the games they attempt to learn is conventional Frogger, where they compare their learned policy to a simple baseline policy and random movement. The authors of [3] implement a genetic algorithm to learn the popular game Tetris. [4] implements a NeuroEvolution of Augmenting Topologies (NEAT) based solution to learn to play conventional Frogger. [5] examines the effects of fitness functions on the ability of a robot evolved with genetic algorithms to achieve its task. While we did not plan on using neural nets to implement our genetic algorithm, all three papers provide some insight into other general aspects of genetic algorithms that we use as inspiration.

III. Game Overview

A screenshot of the game can be seen in Figure 1. The possible moves for the player are UP, LEFT, RIGHT, DOWN, and STAY, and the score is proportional to how long the player has been alive. The score increments by one every 100 milliseconds. Block types consist of safe spots and death spots. Safe spots are grass (green), road (gray), and logs (brown if normal, black if sinking). Death spots are cars (red), and water (blue). Sinking logs are considered safe while still black. The frog is pink. Logs and cars spawn randomly, and logs sink randomly. The speed of movement of the logs and cars in a given row is also random, but the maximum possible value increases as the player score increases. Since

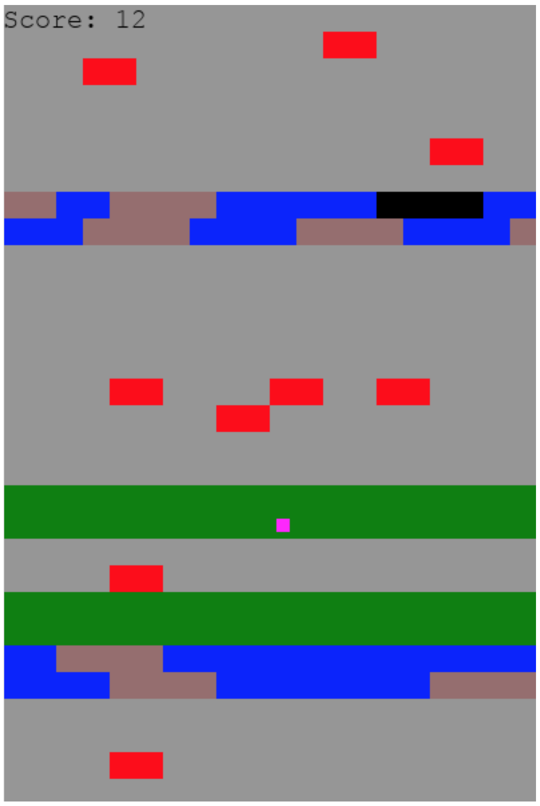

Fig. 1. Example of possible game state

the game is an infinite scrolling game, the screen periodically moves up, preventing the user from staying in a safe spot (grass row) forever. The speed at which the screen advances upwards is also proportional to the player score.

Our build of infinite Frogger was created using Pygame. Also, we used the Q-learning implementation from the Blackjack assignment (homework 4) as a template for our learning algorithms.

IV. Baseline and Oracle

To establish reasonable expectations for our algorithms, it is imperative we establish lower and upper bounds for our performance to determine how well each algorithm is learning the game.

A. Baseline

A logical baseline for our Frogger game is the simple greedy approach presented in Algorithm 1.

While simple and naive, this algorithm makes sense for a baseline, as it prioritizes moving forward the most. The priorities for moving right or left are equal, so whichever one goes first is randomized

Algorithm 1: Baseline Algorithm

```
if block in up direction is open then
    - move up
else if blocks on left and right are open then
    - move right or left at random
else if only one of right or left is open then
    - move to the open position
else
    - move backward
end
```

per frame. Finally, moving backward is reserved as a last resort action. The baseline controller actually performs reasonably well, achieving an average score of 69.1 points. This is the lower bound we strived to surpass with our learning algorithms.

B. Oracle

Since this is a one-player survival game, we surmised the oracle would be a perfect player that never dies. This oracle is not actually implementable, as the generation of various hazards is random, sometimes leading to situations where death is inevitable (a river with no logs). This oracle is more of a theoretical upper bound that we would like to get as close to as possible through the implementation of various learning algorithms.

V. Learning Algorithms

In this section, we give an overview of the various learning algorithms we employ to learn our game of infinite Frogger. We experiment with three different algorithms: the off-policy learning algorithm Q-learning, the on-policy learning algorithm SARSA, and a genetic algorithm.

A. TD Learning Methods

For the TD learning methods, we model our system as a Markov Decision Process. We use binary valued features consisting of (state, action) tuples, with the state dependent on the feature extractor being used. The possible actions for our MDP are UP, LEFT, DOWN, RIGHT, and STAY. Features and rewards will be described in more detail in the next section.

1) Q-learning: On each (s, a, r, s'), the weight update rule for Q-learning with function approximation is given as

$$\mathbf{w} \leftarrow \mathbf{w} - \eta[\hat{Q}_{opt}(s, a; \mathbf{w}) - (r + \gamma \hat{V}_{opt}(s'))]\phi(s, a) \tag{1}$$

where $\mathbf{w}$ is the feature weight vector, η is step size, r is the reward for taking action a in state s, γ is the discount factor, $\phi(s, a)$ is the feature vector corresponding to taking action a in state s,

$$\hat{Q}_{opt}(s, a; \mathbf{w}) = \mathbf{w}^T \phi(s, a) \tag{2}$$

and

$$\hat{V}_{opt}(s') = \max_{a' \in actions(s')} \hat{Q}_{opt}(s', a'; \mathbf{w}) \tag{3}$$

2) SARSA: On each (s, a, r, s', a'), the weight update rule for SARSA with function approximation is given as

$$\mathbf{w} \leftarrow \mathbf{w} - \eta[\hat{Q}_{\pi}(s, a; \mathbf{w}) - (r + \gamma \hat{Q}_{\pi}(s', a'; \mathbf{w}))]\phi(s, a) \tag{4}$$

where $\hat{Q}_{\pi}(s, a; \mathbf{w})$ is the current estimate of the Q-value of taking action a in state s, and all the other parameters are defined identically as in Equation 1.

B. Epsilon Greedy

For Q-learning and SARSA, it is imperative that we balance exploration and exploitation for efficient and effective learning. To do so, we utilize the epsilon greedy algorithm. For Q-learning, the greedy algorithm is defined as

$$\pi_{act}(s) = \begin{cases} \arg\max_{a \in Actions} \hat{Q}_{opt}(s, a; \mathbf{w}) & \text{probability } 1 - \epsilon \\ \text{random from Actions(s)} & \text{probability } \epsilon \end{cases} \tag{5}$$

Thus, for any state s a random action is taken with probability ϵ so that our learning algorithms can explore the state space but also exploit what it learns to gain a high utility. For SARSA, the equation for epsilon greedy is identical, except $\hat{Q}_{opt}(s, a; \mathbf{w})$ is replaced with $\hat{Q}_{\pi}(s, a; \mathbf{w})$.

C. Genetic Algorithm

An alternative approach to the reinforcement learning algorithms described earlier is to employ a genetic algorithm. In general a genetic algorithm is made up of genes, which then make up a genome, where every possible genome makes up the genome pool. In other words, each gene corresponds to a feature weight, a genome is a feature weight vector, and the genome pool is the domain of all possible feature weight vectors. The optimality of the feature vector weights can be quantified by a fitness function. Essentially, the higher the fitness of an individual genome, the more likely it is to be the optimal solution. In our scenario, the fitness is proportional to the score, i.e. how long the agent stays alive. For each generation, the genomes with the top fitness scores "reproduce" to create child genomes that are hopefully fitter than their parents. Given two parent genomes, a child genome is generated by

$$c_{ij} = \frac{f_i p_i + f_j p_j}{f_i + f_j} \tag{6}$$

where p_i is the genome of parent i, f_i is the fitness of parent i, and c_{ij} is the child genome of parents i and j. Like in evolutionary biology, there is a chance that genes in the child genome can "mutate" by some amount. We model mutation as a Bernoulli random variable, where α is the probability of mutation. If a gene is to be mutated the amount it changes by is modeled as a Gaussian random variable X with mean μ and variance σ^2. Once all of the children have been formed, the final step is to form a new generation of n genomes by making a new genome pool consisting of the child genomes and the top $n - c$ parent genomes, where c is the number of child genomes. This process repeats over many generations, and at the end of training, the genome with the highest fitness will be the optimal weight vector to use. A summary of the genetic algorithm can be seen in Algorithm 2.

This method is an interesting comparison to the TD learning methods because it seems both algorithms share a lot of commonality. The mutation element in the genetic algorithm is analogous to the exploration probability in Q-learning and SARSA,

Algorithm 2: Genetic Algorithm

```
- Initialize $n$ genomes (can initialize randomly
  or using pre-learned weights for a warm start)
while not converged or the max number of
  iterations has not been reached do
    for every genome in the pool do
        - calculate the average fitness of the
          genome over $m$ runs
    end
    - Get parent genomes that will breed by
      taking the top $\lfloor \sqrt{n} \rfloor$ genomes with the
      highest average fitness
    - Cross breed all combinations of parent
      genomes as a weighted average of the two
      parents (Equation 6)
    for each child genome do
        - Pick a gene uniformly at random
        - Mutate gene by amount $x$ with
          probability $\alpha$
    end
    - Form a new generation of $n$ genomes by
      making a new genome pool consisting of
      the child genomes and the top $n - c$
      parent genomes, where $c$ is the number of
      child genomes.
end
```

as both seem to balance the trade-off between exploration and exploitation.

VI. Feature Extraction and Rewards

All of our feature extractors return (state, action) tuples as features for non-death states, and return a special token feature if the player is dead. The value associated with a feature is 1 if the game is in the given state and the player takes the given action. It is zero otherwise. The utility of a feature extractor is limited by the complexity of the states that can be represented in its tuple. We explored using both simple and complex states.

A. Simple Feature Extractor

The simple feature extractor represented state using the following features:

1) is the block directly in front of the player currently safe? (yes / no)
2) is the block directly to the left of the player currently safe? (yes / no)
3) is the block directly to the right of the player currently safe? (yes / no)
4) is the block directly behind the player currently safe? (yes / no)
5) is there currently a river in front of the player, and if so, in which direction is the closest log? (not a river / up / left / right / no log in river)

B. Complex Feature Extractor

The complex feature extractor represented state using the following features:

1) what are the values of the blocks in the following locations relative to the player: front left, front center, front right, two to the left, one to the left, directly on top of, one to the right, two to the right, back left, back center, back right? (safe / unsafe / safe but will sink soon)
2) what is the direction of movement of the rows in the following locations relative to the player: in front of, centered on, behind? (left / right)
3) what is the direction type of the rows in the following locations relative to the player: in front of, centered on, behind? (safe / river / road)
4) will anything sink in the next time frame for the rows in the following locations relative to the player: in front of, centered on, behind? (yes / no)
5) is there currently a river in front of the player, and if so, in which direction is the closest log? (not a river / up / left / right / no logs in river)

C. Genetic Algorithm Feature Extractor

For the genetic algorithm, we employed a feature extractor consisting of the the first four features listed in the simple feature extractor section. This was done to reduce the state space used for the genetic algorithm so that it could learn faster.

D. Rewards

To expedite the learning process, we tailored rewards such that they correlated with the features extracted during training. The rewards included:

1) +8 if the player goes forward
2) -9 if the player moves backwards

3) +5 if the row directly behind the player is a river
4) -d if the row directly in front of the player is a river (where d is the distance to the closest log)
5) -500 if the player is dead

We included rewards 1 and 2 to encourage forward movement and to discourage backward movement. Because this game has no end, reward 3 was included as an intermediate goal to encourage learning to get across rivers. Reward 4 was included to encourage movement towards logs when the player approached a river, and reward 5 was included to greatly penalize dying.

VII. Experimental Procedure

For the two TD learning methods described in sections V-A1 and V-A2 we performed four experiments:

- Q-learning with the simple feature extractor
- Q-learning with the complex feature extractor
- SARSA with the simple feature extractor
- SARSA with the complex feature extractor

For each experiment we trained for $150,000$ play throughs of the game, with an exploration policy probability, or ϵ, of 0.4. Also, for every 1000 iterations of training, we would set ϵ to 0 and take the average score of the currently trained model over the course of 15 games. The discount factor γ was set to 0.8 and the step size η was set to 0.1.

For the genetic algorithm described in section V-C, we used the genetic feature extractor described in section VI-C. We did this to reduce the size of the state space, so that our algorithm could learn faster. We initialized $n = 10$ genomes to weight vectors containing zeroes for every feature weight. We ran the genetic algorithm for 1000 generations. For each generation, we calculated the average fitness of each genome over $m = 5$ runs and chose the top $\lfloor\sqrt{n}\rfloor = 3$ genomes to reproduce. To expedite learning, we set the mutation probability to $\alpha = 0.7$ and the amount mutated was set to the Gaussian random variable X, where $X \sim \;(\mu = 0, \sigma^2 = 4)$.

VIII. Results

A. Learning Curves

The learning curves for the TD learning experiments can be seen in figure 2, where the curves were formed by taking a moving average over the previous $15,000$ iterations. The learning curve for the genetic algorithm can be seen in figure 3, where the curve was formed by taking a moving average over the maximum fitness of the previous 10 generations. Note that both figures compare the results to the baseline algorithm and an algorithm of completely random movement.

B. Analysis

Looking at the TD learning curves in figure 2, the first observation to note is that the models using the simple feature extractors initially learned faster than the models using the complex feature extractors. These results make sense, as the feature space of the simple feature extractor is much smaller than the feature space of the complex feature extractor, and so it will take less time for our exploration policy to visit all of the states. While learning does indeed happen at a faster rate, the lack of information encoded in the simple feature extractor restricts the max score to lower than that of the complex feature extractor.

Further, comparing Q-learning and SARSA for both feature extractors, it seems that Q-learning learns at a faster rate overall than SARSA. This makes sense because SARSA updates the utility of a state action pair based on the next state and action actually taken. This means that if our exploration policy causes us to take a non-optimal policy in the next state, leading us to a death that should have been avoided, SARSA will penalize the current state action pair. On the other hand, Q-learning assumes we take the best possible action in the next state, so it does not penalize the current state action pair when the exploration policy forces our player to take a non-optimal action in the next state.

Due to time constraints, we were unable to run the models using the complex feature extractor to convergence. However, for the purposes of the project, the results unequivocally indicate the complex feature extractor learns more elaborate and effective policies than those of the simple feature extractor.

The genetic algorithm also yielded some very interesting and enlightening results. The learning curve seems to have two main components: ranges of slow increase in fitness and ranges of sudden

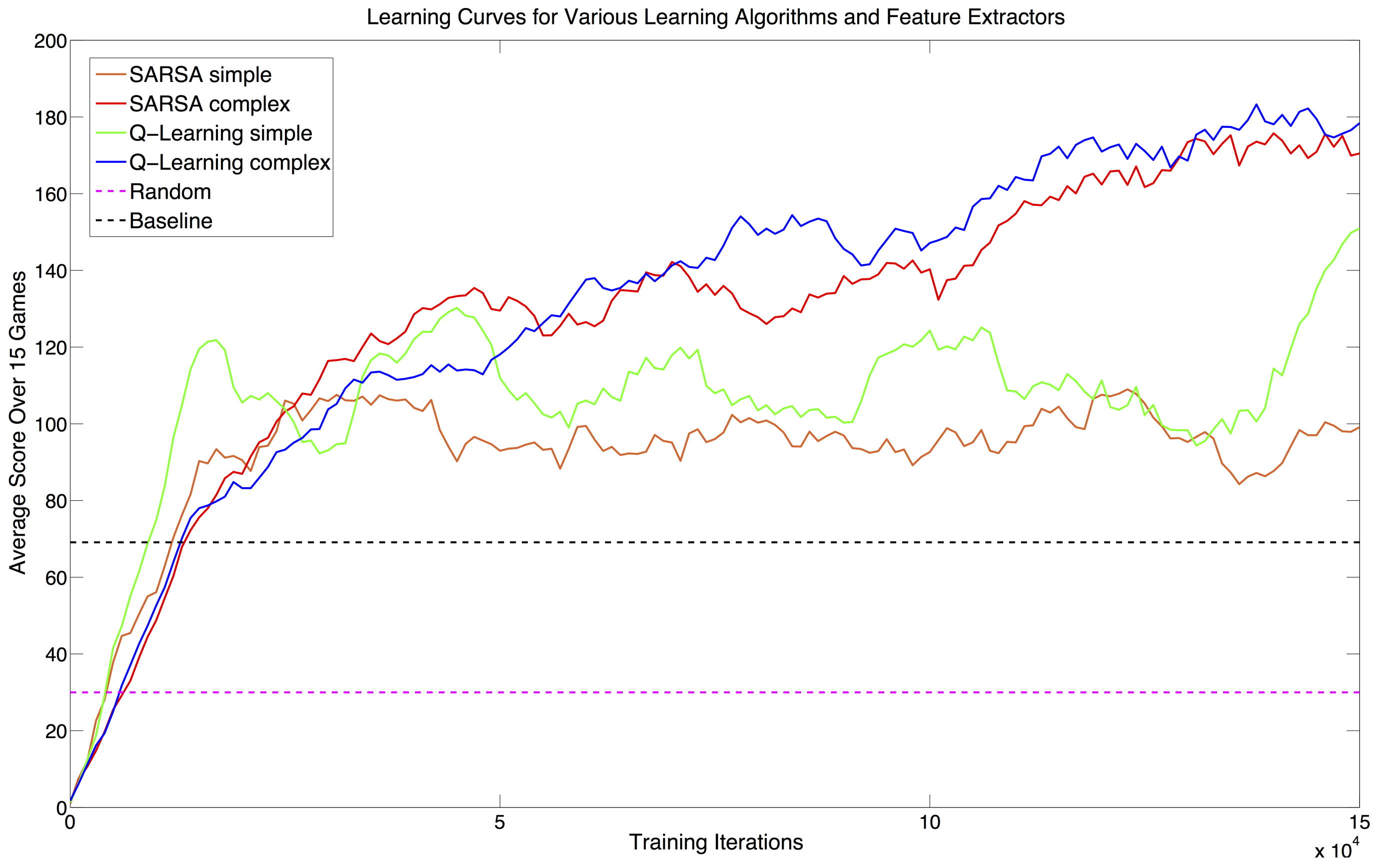

Fig. 2. Learning Curves for SARSA and Q-learning using the simple and complex feature extractors compared to baseline algorithm and random movement.

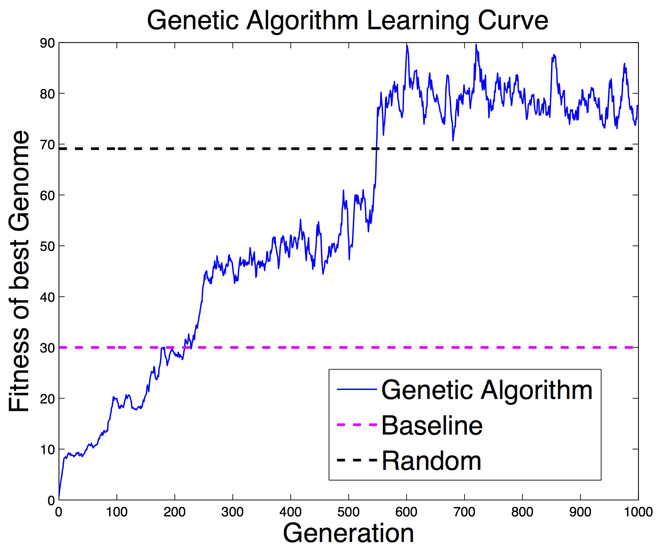

Fig. 3. Learning curve for genetic algorithm compared to baseline algorithm and random movement.

increase in fitness. We think the areas of sudden increase in fitness (in between generations 230 and 270, and in between generations 530 and 570) can be attributed to favorable mutations. The areas of slower increase in fitness are most likely due to the child genomes being slightly better variations of their respective parent genomes. Again, due to time constraints, we were unable to run the genetic algorithm to convergence. However, the shape of the learning curve suggests the algorithm was successful in learning how to play our game, as its performance still passed the baseline threshold. Given a more elaborate feature extractor and more time to train, we expect the results would be even better.

IX. Conclusions and Future Work

In this project, we compared the effectiveness of various learning algorithms for learning a game of infinite Frogger, and we also compared the effects of different types of feature extractors on the ability of our agent to learn how to play the game. While all of the implemented methods beat our baseline score, it seems that Q-learning coupled with a complex feature extractor yields the best results. Overall, we were very pleased with how well each method learned to play the game.

The most challenging aspect of the project was formulating a system of rewards and feature extractors such that learning was both efficient and effective. We experimented with many different types of feature extractors, and finally created two feature extractors to perform our experiments in a timely fashion.

Future work on this project could include designing other types of feature extractors and to compare them to the ones presented in this paper. It would also be interesting to train these various models for much longer than we did to see how good they can actually become at the game. Running each training process to convergence would glean more information as to how effective each of these learning methods can be. It would also be interesting to test other parameters for the genetic algorithm, other types of fitness functions, other reproduction schemes, and to simply run our presented algorithm for more generations.

ACKNOWLEDGMENT

We would like to thank Professor Percy Liang and our mentor Irving Hsu for providing guidance throughout this project.

REFERENCES

[1] Emigh, Matthew S.; Kriminger, Evan G.; Brockmeier, Austin J. *Reinforcement Learning in Video Games Using Nearest Neighbor Interpolation and Metric Learning*, IEEE Transactions on Computation Intelligence and AI in Games, 2013, vol 8, pp. 56-66.

[2] Johnson, Justin; Roberts, Mike; Fisher, Matt. *Learning to Play 2D Video Games*, (2012).

[3] Bohm, Niko; Gabriella Kokail; Mandl, Stegan. *An Evolutionary Approach to Tetris*, MIC2005: The Sixth Metaheuristics International Conference (2005).

[4] Ancona, Davis; Weiner, Jake. *Developing Frogger Player Intelligence Using NEAT and a SCORE Driven Fitness Function*

[5] Chasins, Sarah; Ng, Ivana. *Fitness Functions in NEAT-Evolved Maze Solving Robots*, Tech Report (2010).