Learning to Play 2D Video Games

Justin Johnson
jcjohns@stanford.edu

Mike Roberts
mlrobert@stanford.edu

Matt Fisher*
mdfisher@stanford.edu

Abstract

Our goal in this project is to implement a machine learning system which learns to play simple 2D video games. More specifically, we focus on the problem of building a system that is capable of learning to play a variety of different games well, rather than trying to build a system that can play a single game perfectly.

We begin by encoding individual video frames using features that capture the absolute and relative positions between visible objects. This feature transform: (1) generalizes across a wide class of 2D games; and (2) produces very sparse feature vectors, which we exploit to drastically reduce computation times. To learn an appropriate gameplay policy, we experiment with model-based and model-free reinforcement learning methods. We find that the SARSA(λ) algorithm for model-free reinforcement learning successfully learns to play PONG, FROGGER, DANCE-DANCE-REVOLUTION, *as well as several other games of comparable complexity.*

1. Introduction

AI systems are capable of playing specific video games, such as Super Mario World [4] and Starcraft [7], with comparable skill to expert human players. However, all such AI systems rely on a human to somehow perform the challenging and tedious task of specifying the game rules, objectives and entities.

For example, state-of-the-art AI systems for playing Mario and Starcraft can play these games effectively, even when faced with challenging and complex game states. However, these systems rely heavily on hand-crafted heuristics and search algorithms that are specific to the game they target, and are not readily generalizable to other games.

In contrast, systems for General Game Playing (GGP) [3], such as CadiaPlayer [2], can play novel games for which they were not specifically designed. However, GGP systems rely on a human to provide a complete formal specification of the game rules, objectives, and entities in a logical programming language similar to Prolog. Arriving at such a formal specification is very tedious even for the simplest games. This limitation significantly constrains the applicability of GGP systems.

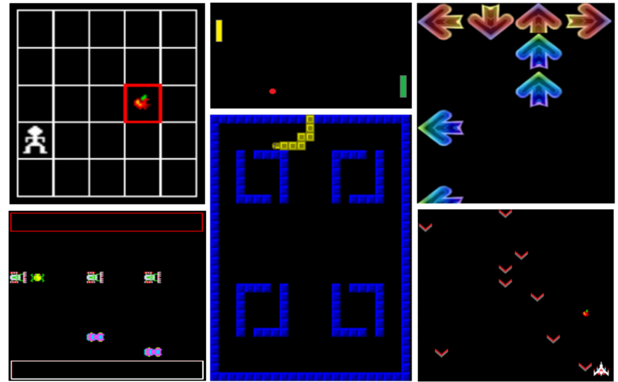

Figure 1. Our system successfully learns to play the games shown above: EAT-THE-FRUIT (top-left), PONG (top-middle), DANCE-DANCE-REVOLUTION (top-right), FROGGER (bottom-left), SNAKE (bottom-middle), DODGE-THE-MISSILE (bottom-right).

Very recently, Bellemare *et al*. [1] released the Arcade Learning Environment for evaluating the performance of AI agents on a large set of Atari 2600 games. In this work, Bellemare *et al*. evaluate a variety of feature transforms that generalize across 2D games, as well as evaluating the SARSA(λ) algorithm for online model-free reinforcement learning in this setting. Bellemare *et al*. demonstrate that the SARSA(λ) achieves reasonable performance on a large variety of games.

In this project, we aim for comparable performance and generality to that recently demonstrated by Bellemare *et al*. [1]. Indeed, our technical approach is directly inspired by their work. To our knowledge, the Arcade Learning Environment is the only system to implement AI agents that learn to play such a wide variety of non-trivial games. Therefore, it is worth emphasizing that the games we consider in this project are at the approximate boundary of what general AI agents are capable of learning. This is true despite the apparent simplicity of our games, even compared to classic 2D games like Super Mario World.

*Mike and Justin are enrolled in CS 229, but Matt is not. Matt is a senior PhD student in the Stanford Graphics Group, who will has advised and collaborated with Mike and Justin on this project. He wrote the game model learning algorithm mentioned in Section 4.

2. Games

To evaluate our system we implemented a number of games of complexity comparable to early arcade games. Each game contains a number of distinct object types, and game state consists of a fixed configuration of objects. The state space S of a game is the set of all possible object configurations. Unless otherwise noted, the action space of each game is $A = \{L, R, U, D, \emptyset\}$, and consists of one action corresponding to each of the four cardinal directions and the do-nothing action $\emptyset$.

Each game is played over a series of episodes, where an episode consists of many frames. To prevent a perfect player from playing indefinitely, we cap episode length where appropriate. In all situations, early termination of an episode due to capping earns the player zero reward in the final frame of the episode.

GRID-WORLD. In this game, the player controls a character on a 5×5 grid. During each frame of the game, the player may move in any direction or remain stationary. The player begins each episode in the lower left corner of the grid and must reach the upper right corner. When this goal is achieved, the player receives a positive reward and the episode ends. In addition, the player receives a negative reward for stepping on the central square. We evaluate the player's performance by counting the number of frames per episode. Fewer frames per episode indicates better performance, since it means that the player navigated to the goal square more quickly.

EAT-THE-FRUIT. Similar to GRIDWORLD, the player controls a character on a fixed sized grid. At the start of each episode, an apple appears on a randomly chosen square. The player begins in the lower left corner of the grid and must move to the apple. After eating the apple, the player receives a reward and the episode ends. As in GRIDWORLD, we measure a player's performance on this game by counting the number of frames per episode.

DODGE-THE-MISSILE. In this game the player controls a space ship which can move left or right across the bottom of the screen, so the action set is $A = \{L, R, \emptyset\}$. Missiles and powerups spawn at the top of the screen and fall toward the player. The player receives a positive reward for collecting powerups; being hit by a missile incurs a negative reward and causes the episode to end. We cap the episode length at 5000 frames. We evaluate the player's performance by counting both the number of frames per episode and the number of powerups collected per episode. Larger numbers for each metric indicate better performance.

FROGGER. In this game, the player controls a frog which can move in any direction or remain stationary. The player begins each episode at the bottom of the screen and must guide the frog to the top of the screen. This goal is made more challenging by cars that move horizontally across the screen. The episode ends when the frog either reaches the top of the screen or is hit by a car. The former earns a reward of $r_1 > 0$ and the latter receives a reward of $r_2 < 0$. We evaluate the player's performance by computing her average reward per episode.

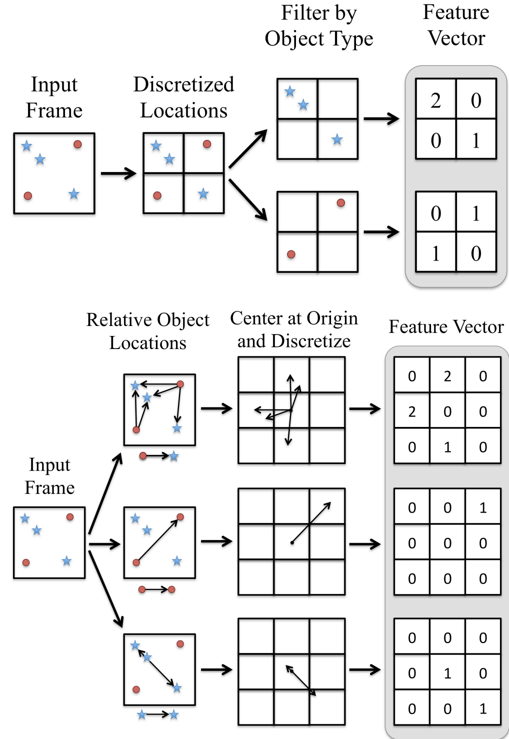

Figure 2. Our tile-coded feature representation. We encode the absolute positions of game objects (top) as well as relative positions of game objects (bottom) in spatial bins. Relative positions are computed separately for all pairs of object types. For any game state $s \in S$, this results in a feature vector $\phi(s)$ of dimension $d = O(k^2)$ where k is the number of distinct object types in the game. To be used in the SARSA learning algorithm, the feature transform must also encode the action $a_i \in A = \{a_0, \ldots, a_{|A|-1}\}$ that is to be taken from the current game state s. To this end, our final feature vector $\phi(s, a_i)$ is simply the vector $\phi(s)$ with all indices shifted by $i|A|$ and with zeros at all other positions.

PONG. In this game, two paddles move up and down across the left and right sides of the screen while volleying a ball back and forth. The player controls the left paddle, whereas the game controls the right paddle. The action space is $A = \{U, D, \emptyset\}$. Failing to bounce the ball yields a negative reward and ends the episode. We cap the episode length at 50 successful bounces. We evaluate the player's performance by counting the number of successful bounces per episode.

SNAKE. In this game, the player controls a snake of fixed length that moves around a maze. With no player in-

put, the snake moves forward at a constant rate; pressing a direction key changes the direction that the snake travels. The episode ends with a negative reward if the snake head intersects either a wall or the snake body. We cap the epsiode length at 180 frames. We evaluate the player's performance by counting the number of frames that it survives per episode.

DANCE-DANCE-REVOLUTION. In this game, arrows appear at the bottom of the screen and scroll toward targets at the top of the screen. Whenever an arrow overlaps its corresponding target, the player must press the direction key corresponding to the direction of the arrow. The trivial strategy of pressing every arrow at every frame is impossible, since the player can press at most one direction per frame. Each episode lasts for 1000 frames, and a player's performance is measured by the fraction of arrows that it successfully hits.

3. Feature Design

Since we want our learning system to generalize across games, we must avoid including any game-specific state in our features. For example, explicitly encoding the position of Mario, along with the positions of game entities that we know can harm Mario, into our features would run counter to our goal of generality. However, we must encode the observable game state with sufficient fidelity to make accurate predictions. On the other hand, we must carefully design features of sufficiently low dimensionality that our learning problems remain computationally tractable.

With these competing concerns in mind, we follow the approach of Bellemare *et al*. [1] and encode the game state using tile-coded features. This encoding allows us to efficiently encode the absolute and relative positions of objects within the game. See Figure 2 for details.

Although the resulting feature vector is very high dimensional (over 100,000 for several games), it is very sparse (see Figure 3). Storing the feature vector sparsely allows our algorithm to remain computationally efficient despite the high dimensionality of our feature transform.

4. Model-Based Reinforcement Learning

To learn an appropriate gameplay policy, we began with the following observation. Although the state spaces of the games we consider are very large, the action spaces of the games we consider are very small. For example, there are roughly $10^{32\times 32}$ different possible states in SNAKE, but only 5 actions. This observation motivated us to learn gameplay policies by performing fitted value iteration.

Since fitted value iteration requires access to a game model, we learned one from recorded examples of gameplay. More specifically, we formulated the program of learning a game model as a supervised learning problem, which we addressed by training a collection of decision trees. Very roughly speaking, our input features encoded the current observable game state at time t, as well as the input provided by the player at time t. Our target variables encoded the game state at time $t+1$. We then learned a game model by training a collection of decision trees. See our midterm progress report for details.

Using our learned game model, as well as the feature transform ϕ described in Section 3, we were equipped to apply fitted value iteration to learn a gameplay policy. At its core, fitted value iteration approximates the value function as $V(s) \approx \theta^T\phi(s)$. The weight vector θ is found by solving a linear regression problem with design matrix $\Phi(\vec{s}) = (\phi(s_1), \ldots, \phi(s_m))^T$ where $\vec{s} = (s_1, \ldots, s_m)$ is a vector of states. Unfortunately, we observed severe numeric instability using this approach for games as simple as GRID-WORLD. We speculate that this instability stems from a severe rank-deficiency of the feature matrix $\Phi(\vec{s})$.

In the case of GRID-WORLD, the rank-deficiency of $\Phi(\vec{s})$ occurs because there are a total of 25 unique states, assuming a 5×5 game area. Therefore, $\Phi(\vec{s})$ can have at most 25 unique rows no matter the dimension of $\vec{s}$, so $\Phi(\vec{s})$ has rank ≤ 25. However, using the feature transform described in Section 3, $\Phi(\vec{s})$ will much greater than 25 columns. The linear regression problem is therefore underconstrained and has an infinite number of solutions. We considered taking the minimum-norm solution, but it is not clear that this approach would sensibly approximate our unknown value function.

To make matters worse, this numeric instability becomes more pronounced for more complex games. The dimension of the feature transform $\phi(s)$ (and hence the number of columns of $\Phi(\vec{s})$) grows quadratically with a large con-

Algorithm 1 Learn to play a game using the SARSA(λ) algorithm with linear function approximation. See Section 5 for definitions of the variables.

```
function LEARN-GAME()
    τ ← 0, w ← 0
    s ← Initial state, a ← Initial action
    ε ← ε_0 // Initial exploration rate
    repeat
        Take action a, observe next state s′ and reward r
        s′ ← CHOOSE-ACTION(s, ε)
        δ ← r + γ w^T φ(s′, a′) − w^T φ(s, a)
        τ ← λτ
        for φ_i(s, a) ≠ 0 do
            τ_i = 1
        w ← w + wαδτ // Update weight vector
        ε ← ε_d ε // Decay exploration rate
    until termination
```

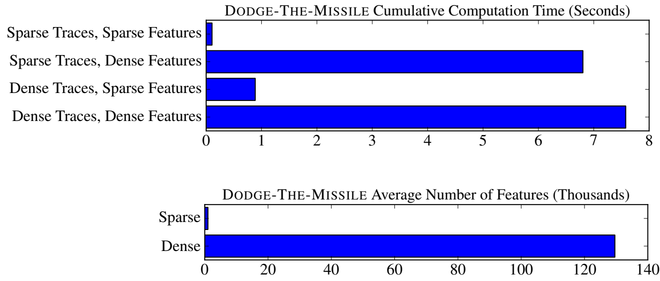

Figure 3. Top: Comparison of total computation time for learning DODGE-THE-MISSILE using different combinations of sparse and dense feature and trace vectors. Bottom: The average number of nonzero features for DODGE-THE-MISSILE. We observe that the average number of nonzero features is very small, which we exploit to drastically reduce computation times.

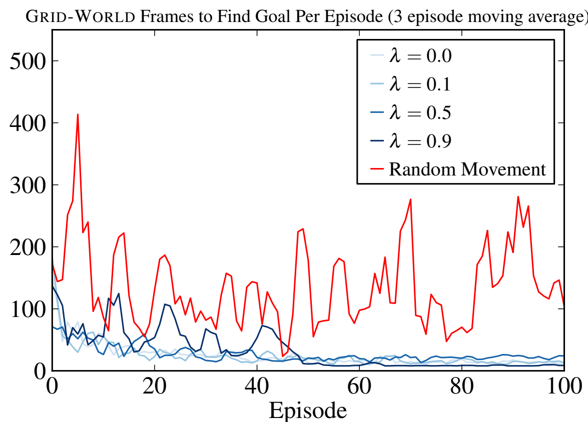

Figure 4. The effect of different values for λ. on the convergence of our algorithm when playing GRID-WORLD. We show convergence rates for various values of λ relative to random movement. Lower is better. We observe that our algorithm outperforms random play across a wide range of values for λ.

stant as the number of unique object types increases. Even if this is smaller than the number of sensible game states, a numerically stable regression step would require a large number of training samples, which could quickly become computationally infeasible.

The numerical stability problems associated with fitted value iteration prompted us to markedly reconsider our technical approach. This lead us to the model-free reinforcement learning algorithm we describe in the following section.

5. Model-Free Reinforcement Learning

Our system uses the SARSA(λ) algorithm with linear function approximation (see Algorithm 1) to learn a gamelay policy. SARSA [5] is an online model-free algorithm for reinforcement learning. The algorithm iteratively computes a state-action value function $Q : S \times A \rightarrow \mathbb{R}$ based on the rewards received in the two most recently observed states. SARSA(λ) [6] is a variant that updates the state-action value function based on the rewards received over a large window of recently observed states. In our case, the full state space S of an unknown game may be very large, so we approximate the state-value function as $Q(s, a) = w^T \phi(s, a)$, where $\phi : S \times A \rightarrow \mathbb{R}^n$ is the feature transform described in Section 3 and $w \in \mathbb{R}^n$ is a weight vector.

At each game state s, the algorithm chooses an action a using an ϵ-greedy policy: with probability ϵ the action is chosen randomly, and with probability $1 - \epsilon$ the action is chosen to satisfy $a = \arg\max_{a \in A} Q(s, a)$. The parameter $\epsilon \in [0, 1]$ controls the relative importance of exploration and exploitation, and as such is known as the *exploration rate*. In our implementation we decay ϵ exponentially over time. This encourages exploration near the beginning of the learning process, and exploitation near the end of the learning process.

The algorithm keeps track of recently seen states using a *trace vector* $\tau \in \mathbb{R}^n$. More specifically, τ records the recency with which each feature has been observed to be nonzero. The sparsity of typical feature vectors causes τ to be sparse as well. This sparsity can be exploited for computational efficiency (see Figure 3).

We update the trace vector using a parameter $\lambda \in [0, 1]$, which controls the extent to which recently seen features contribute to state-value function updates. Varying the value of λ can affect the rate at which the learning algorithm converges (see Figure 4). Admittedly, we found that different values of λ were required for each game in order to achieve the best possible performance. For example, we used $\lambda = 0.3$ for DANCE-DANCE-REVOLUTION and $\lambda = 0.8$ for DODGE-THE-MISSILE.

The algorithm also depends on a *learning rate* $\alpha \in [0, 1]$, which has similar meaning to the learning rate in gradient descent. We found that α required tuning for each game. For example, FROGGER performed best with $\alpha = 0.01$, whereas SNAKE performed best with $\alpha = 0.00001$.

6. Results

Our system successfully learns to play GRID-WORLD, EAT-THE-FRUIT, DODGE-THE-MISSILE, FROGGER, PONG, and DANCE-DANCE-REVOLUTION. We evaluate the performance of our system on these games by comparing with game agents that choose random actions at every frame to show that substantial learning takes place (see Figures 4 and 5). Our system learns to play SNAKE successfully only when we simplify the game by reducing the length of the snake body to 1 (see Figure 6).

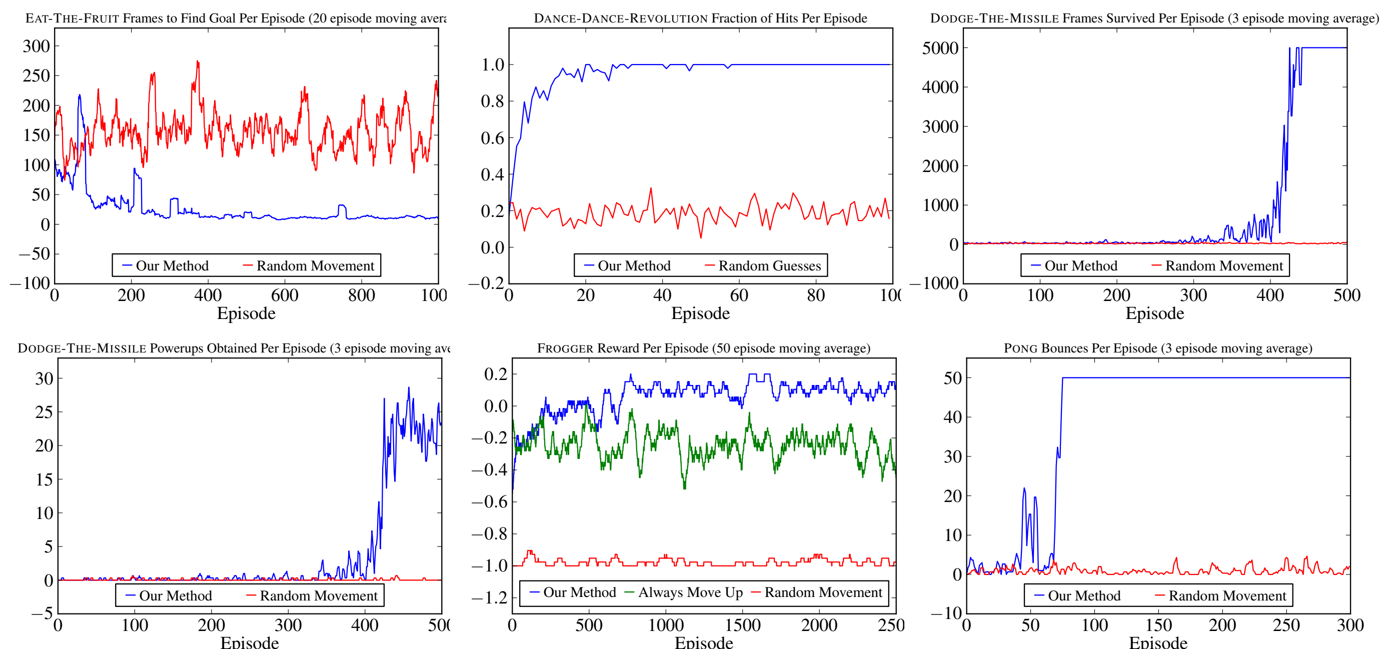

Figure 5. Performance of our algorithm relative to random play for EAT-THE-FRUIT (top-left, lower is better), DANCE-DANCE-REVOLUTION (top-middle), DODGE-THE-MISSILE (top-right and bottom-left), FROGGER (bottom-middle), and PONG (bottom-right). For DODGE-THE-MISSILE, we capped the episode length at 5000 frames. For PONG, we capped the episode length at 50 bounces. For FROGGER , we set $r_1 = 0.2$ and $r_2 = -1$. Note that after our algorithm has learned to play DODGE-THE-MISSILE effectively, it is capable of collecting powerups while simultaneously avoiding missiles. Note that since continuously trying to move upwards can be a viable strategy when playing FROGGER, we also compare the performance of our algorithm to an AI player that continuously tries to move upwards.

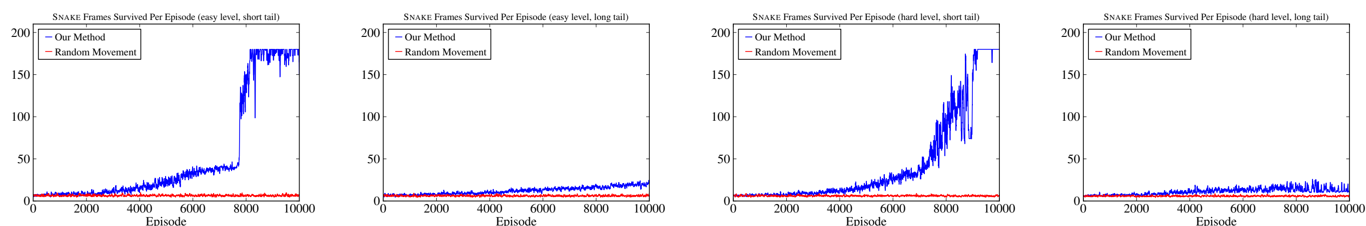

Figure 6. Performance of our algorithm on SNAKE relative to random play. We evaluate our algorithm's performance on the following four different game variations: empty game board with a snake body length of 1 (left), empty game board with a snake body length of 10 (left-middle), relatively cluttered game board with snake body length of 1 (right-middle), and relatively cluttered game board with a snake body body length of 10 (right). Our algorithm was able to learn effectively on the cluttered game board, but not with the longer body. This is because having a longer body requires longer-term decision making. On the other hand, a short body makes it possible to play according to a relatively greedy strategy, even on a relatively cluttered game board.

References

[1] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *ArXiv e-prints*, July 2012.

[2] Y. Björnsson and H. Finnsson. CadiaPlayer: A Simulation-Based General Game Player. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1), 2009.

[3] M. Genesereth, N. Love, and B. Pell. General Game Playing: Overview of the AAAI Competition. *AI magazine*, Spring, 2005.

[4] S. Karakovskiy and J. Togelius. The Mario AI Benchmark and Competitions. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1), 2012.

[5] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of Machine Learning*. Adaptive Computation and Machine Learning Series. Mit Press, 2012.

[6] M. Wiering and J. Schmidhuber. Fast online q (λ). *Machine Learning*, 33(1):105–115, 1998.

[7] J. Young, F. Smith, C. Atkinson, K. Poyner, and T. Chothia. SCAIL: An Integrated Starcraft AI System. In *IEEE Conference on Computational Intelligence and Games*, 2012.