Reinforcement Learning for Atari Breakout

Vincent-Pierre Berges
vpberges@stanford.edu

Priyanka Rao
prao96@stanford.edu

Reid Pryzant
rpryzant@stanford.edu

Stanford University
CS 221 Project Paper

ABSTRACT

The challenges of applying reinforcement learning to modern AI applications are interesting, particularly in unknown environments in which there are delayed rewards. Classic arcade games have garnered considerable interest recently as a test bed for these kinds of algorithms. In this paper, we develop a system that is capable of teaching a computer agent how to play a custom implementation of the popular classic arcade game Breakout with variants of Q-learning and SARSA. We detail our implementation, testing, feature generation, and hyperparameter search, and eventually achieve significantly above-baseline control with SARSA. Furthermore, we investigate state-of-the-art learning techniques such as replay memory and policy gradients, finding that these techniques fail to outperform "simpler" methods. Finally, we evaluate our models through repeated gameplay, aggregate performance statistics, and parameter visualization.

1. INTRODUCTION

Reinforcement learning (RL) is currently one of the most active areas in Artificial Intelligence research. It is the technique by which an agent learns how to achieve rewards r through interactions with its environment. Many real-word applications such as robotics and autonomous cars are particularly well-suited for a RL approach as the environment is unknown and the consequences of actions are uncertain. Through trial-and-error and experience over time, an RL agent learns a mapping of states s to optimal actions a and develops a policy to achieve long-term rewards.

However, several challenges face the agent as it tries to learn this policy. The environment often provides delayed rewards for actions, making it difficult for the agent to learn which actions correspond to which rewards. Furthermore each action the agent takes can impact whats optimal later on, sometimes unpredictably. Even if the agent does learn a policy that allows it to achieve rewards, there is still the question of whether the policy is an optimal one. Thus, the agent must make a trade-off between exploring possibly sub-optimal actions with the hope that it may find a more optimal strategy and exploiting its existing policy.

In this project, we sought to understand how different RL algorithms, particularly SARSA, SARSA(λ), and Q-learning, and Policy Gradients, tackle these challenges and compare in terms of performance for Atari Breakout. By constraining the vast problem space to Breakout, we lower the computational complexity of the learning problem while still retaining the key qualitative aspects - an unknown environment and delayed rewards - that characterize most real-world applications. We defined the agent as the paddle and the goal as achieving a high game score. We also limited the action space to left and right movements of the paddle in order to further lower the computational complexity of the problem. By implementing and training the agent on various off-policy algorithms, we sought to determine the algorithms that generalize best and lead most frequently to high game scores.

2. RELATED WORK

Several research groups have explored the application of reinforcement learning to arcade games such as Flappy Bird, Tetris, Pacman, and Breakout. Undoubtedly, the most relevant to our project and well-known is the paper released by by Google DeepMind in 2015, in which an agent was taught to play Atari games purely based on sensory video input [**7**]. While previous applications of reinforcement learning to arcade games heavily relied on hand-crafted feature sets, DeepMind chose to leverage convolutional neural networks to extract relevant features from the game's video input [**4**]. The key contribution of the DeepMind project to the field of reinforcement learning was the novel concept of a deep Q-network, which combines Q-learning in with neural networks and experience replay to decorrelate states and update the action-value function. After being trained with a deep Q-network, the DeepMind agent was able to outperform humans on nearly 85% Breakout games [**4**]. However, despite achieving superhuman performance on several other Atari games, deep Q-networks severely under-performed on games such as Space Invaders that required extended long-term strategic planning [**7**].

Another well-known application of reinforcement learning is TD-Gammon. In 1995, an agent learned to play this highly stochastic game without any knowledge of the game rules [**13**]. Prior to TD-Gammon, other researchers had taught agents to play backgammon with knowledge of the game rules and one of either a neural network or the TD algorithm, but not both [**12**]. TD-Gammon used a unique encoding to represent the state of the backgammon board and combined both TD-learning and a multilayer neural network to estimate the value function. However, this approach did not generalize well to other games, perhaps because the randomness embedded in the rules of backgammon automatically leads to sufficient exploration of the state space [**12**]. In other arcade games, finding the balance between exploration and exploitation proved much more difficult.

Our selection of Breakout as our problem was inspired from the work done by DeepMind and TD-Gammon. Both DeepMind and TD-Gammon combined a non-linear function approximator with an RL algorithm to train an agent. We wanted to try different combinations of RL algorithms and non-linear function approximators and understand which combinations work best for Breakout and why. In this paper, we present an in-depth investigation of the performance of different RL algorithms for Breakout, which has not previously been done. Unlike DeepMind, however, we chose not to represent our state space as a vector of 210 x 160 x 128 pixels. We implemented the game of Breakout ourselves, so we had direct access to all of the game variables. We decided to leverage this programmatic access to craft feature sets instead of using convolutional neural networks.

3. IMPLEMENTATION

Prior to implementing different RL algorithms, we first implemented the game of Breakout from scratch using Pygame, a popular Python library for coding video games. We based our code for the game implementation off the Bricka module in Pygame, but modified it significantly (90.7% of our Breakout code was original). We deviated from the original Atari game in several ways. We limited the maximum ball speed, allowed only one life per game, did not award points for the first broken brick, and injected some randomness into return angles. The initial ball direction was random, and balls bouncing off the left side of the paddle speed up in the left direction (visa versa for right). Furthermore, we introduced an event loop so that we could operate the paddle progammatically with RL algorithms.

Our overall implementation for the project was in Python and was carefully structured into modularized files:

- `game_engine.py`. This file contains a superclass, Breakout, which had the implementation of the Breakout game and an abstract method to repeatedly receive input used to execute game turns. Subclasses in this file include:
  - `HumanControlledBreakout`. This is a game object which takes input from the keyboard in its `run()` method.
  - `BotControlledBreakout`. This is a game object which contains an Agent as an instance variable. On each turn it presents the current state and reward to the agent. The agent updates the Q-values or weights, takes the decision for the next action using the Q-values, updates the past experience. The game then decides whether to explore (using a ϵ-greedy acting policy) or follow this optimal movement.
  - `BaselineBreakout`. This is a game object which took actions randomly and served as our baseline.
  - `OracleControlledBreakout`. This is a game object which breaks the rules of the game to win every time. It follows the direction of the ball.
- `feature_extractors.py`. This file contains a base class, from which other subclasses that define feature sets can inherit from. All our simple and continuous feature sets are defined in this file.
- `agents.py`. This file contains the base class Agent, which several agents inherit from. All agents are in this file, such as the agent that plays with the SARSA algorithm, the Q-learning with replay memory algorithm, etc. Agents are initialized and called in `main.py` with specific hyper-parameter values and a feature extractor.
- `eligibility_tracer.py`. This file contains the logic for SARSA eligibility traces.
- `replay_memory.py`. This file allows any agent to make use of replay memory through a ReplayMemory class and methods to store a SARS tuple or sample from experience.
- `constants.py`. This file contained all the constants and hyperparameters, to simplify training the agents with different values.
- `utils.py`. This file contained several utility functions, some taken from the 221 homework code base.

We used git for version control and also created several test scripts to store models and capture and graph scores achieved over time for different agents.

4. STATES, REWARDS, AND FEATURES

4.1 The State

At each game frame, the game engine provides information of the state. This state is then used to compute the features the agent will use. The state includes :

Key	Information
game_state	Is the game ongoing / lost / won or just started
ball	The x and y coordinates of the ball
ball_vel	The x and y components of the ball's velocity
paddle	The x position of the paddle
bricks	Array of the coordinates of the remaining bricks
time	The number of frames since the game started
score	The current score of the game
lives	The number of lives left%

4.2 Discrete Features

We used a variety of feature extractors $\phi_i(s, a)$ to featurize states. In our first version, the features used were discrete. For instance, there are 4 possible "game_state" values in a state : STATE_BALL_IN_PADDLE, STATE_PLAYING, STATE_WON, STATE_GAME_OVER. Each of these values is a feature with the value 1 if it corresponds to the actual game state and 0 otherwise.

Similarly, for our first feature set we discretized the ball's and paddle's coordinates as well as the magnitude of the ball speed and it's angle with the vertical axis using one-hot encoding. The level of discretization was adjustable through the step size of the grid.

4.3 Continuous features

We quickly switched to continuous feature sets in order to use function approximation methods. In our first continuous feature implementation ($v1$, $\phi_1(s,a)$), we included all the possible information from the state about the paddle and the ball. It was apparent, though, that these features did not perform well. This may have been due to the lack of interaction terms or due to providing too much information. It is possible that the agents were not able to identify relevant features.

We implemented six continuous feature sets, but after training the agents, we realized that the feature sets that worked best were the simplest. For example, in feature set $v2$, there are only two indicator variables: one for relative position, and one for horizontal movement direction.

The third version of the continuous features $v3$ was very similar to the previous two but the position of the ball relative to the paddle was fed through a *tanh* to produce a continuous number between -1 and 1 (-1 when the ball is far left of the paddle, 0 when the ball is above the paddle and 1 when the ball is far right of the paddle).

Other examples of features we tried to include iteration terms, distance ball to paddle, distance ball to wall etc...

The fourth version $v4$ was the same as $v2$ but we included discretization of the absolute position of the ball and interaction terms between ball & paddle position.

In the fifth version $v5$, we included most of the information from the state in continuous form. Some things we included were the distance from the ball to the bricks, different walls, and the paddle.

Finally, $v6$ was built on top of $v2$ and $v4$ but also included an indicator of whether the ball was moving up or down, as well as interaction terms between that variable and other positional variables.

4.4 Rewards

As the agent plays the game, the game engine provides rewards when bricks are hit. We tried multiple approaches and started with the simplest - rewards that were simply the difference of score between two frames. The issue with this implementation was that most of the reward points come from breaking bricks but there is a large temporal gap between performing an action worth rewarding (the ball bouncing on the paddle) and the moment the ball breaks a brick. For this reason, we decided not to reward the agent when it breaks the first brick and instead reward it when it successfully bounces the ball on the paddle.

In order to help the agent learn quickly, we also provided intermediate rewards by rewarding a small amount of points when the x position of the paddle is close to the x position of the ball. The closer the two positions are at any point in time, the larger the reward. When the game ends, the reward is negative and there is a penalty proportional to the distance between the ball and paddle.

5. METHODS

5.1 Reinforcement Learning Algorithms

We decided to take a model-free, off-policy RL approach because any breakout-playing agent lacks explicit access to the game's reward and transition functions. The agent is tasked with maximizing its expected utility by taking actions but has no clue as to the implications of these actions and lacks any representation of the environment it inhabits. We chose model-free because our environment is known (we implemented the game so we know all its rules), but is too large to model directly. Furthermore, recent research indicates that model-based methods under-perform on the Breakout task [**2**]. As such, we directed our efforts to three model-free methods: Q-learning, SARSA(λ), and Policy Gradients (PG). We used function approximation for all three algorithms: linear and neural network for the first, linear only for the second, and neural network only for the third.

5.2 SARSA and SARSA(λ)

5.2.1 *SARSA*

SARSA is an on-policy bootstrapping algorithm that learns a mapping between state-action pairs and the Q-values of the control/evaluation policy π. In our case, we used an ϵ-greedy policy for π: with probability ϵ the action is chosen randomly, and with probability $1-\epsilon$ the action is chosen to satisfy $a = arg\ max_a Q_\pi(s,a)$. ϵ, then, provides some degree of control over the exploitation/exploration tradeoff.

On each (s,a,r,s',a') tuple, $Q_\pi(s,a)$ moves towards an estimate of the true value of being in state s and taking action a. This value is the sum of discounted rewards the agent can expect to receive, i.e. the 1-*step return* $q_t^{(1)}$. Whereas monte carlo (MC) methods take the expectation of this return over many episodes of gameplay, SARSA bootstraps on its current representation of Q_π to obtain its estimate.

The algorithm attempts to minimize the following objective in a function approximation setting:

$$\begin{aligned} min_w \sum_{(s,a,r,s',a')} &\Big(Q_\pi(s,a;w) - (r + \gamma Q_\pi(s',a';w))\Big) \\ = \sum_{(s,a,r,s',a')} &\Big(Q_\pi(s,a;w) - q_t^{(1)})\Big) \end{aligned}$$

It does so with the following SGD update:

$$w_{t+1} = w_t - (Q_\pi(s,a) - q_t^{(1)})\phi(s,a)$$

5.2.2 *SARSA(λ)*

Consider the following n-step returns at some time step t:

$$\begin{aligned} q_t^{(1)} &= r_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1}) \qquad\qquad \text{(SARSA)} \\ q_t^{(2)} &= r_{t+1} + \gamma r_{t+2} + \gamma^2 Q_\pi(s_{t+2}, a_{t+2}) \\ &\dots \\ q_t^{(k)} &= r_{t+1} + \gamma r_{t+2} + \dots + \gamma^{k-1} r_{t+k} + \gamma^k Q(s_{t+k}, a_{t+k}) \\ &\dots \\ q_t^{(\infty)} &= r_{t+1} + \gamma r_{t+2} + \dots + \gamma^{T-1} r_T \qquad \text{(MC)} \end{aligned}$$

SARSA(λ) is a method of letting us combine these n-step returns to reach a middle ground between SARSA and MC. This provides some degree of control over the bias/variance tradeoff. To this end we define a new type of return q^λ, that averages over multiple n-step returns. q^λ gives closer trajectories more weight:

$$q_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}q_t^{(n)}$$

λ is a knob that lets us control how far-sighted the algorithm is. If $\lambda = 1$, $q_t^\lambda = q_t^\infty$, it is the MC estimate. Likewise, if $\lambda = 0$, $q_t^\lambda = q_t^{(1)}$, it is the SARSA estimate.

There is a practical problem with SARSA(λ). We want to be able to do updates in real-time, and q^λ requires that we run out entire episodes of gameplay much like MC. The solution is to make use of a backward view of the same algorithm. Each past experience is given a weight, and those weights are used for updates. This record is called an *eligibility trace*. More concretely, the algorithm maintains a set of eligibility traces τ, one for each function approximator parameter. Each trace has a weight which determines the extent to which its parameter is eligible for credit for the current reward r_t. Each value of τ is updated per time step by a factor of λ. This results in an exponential decay of the impact of rewards over time (just like in the forward view of things). In our implementation, we used a *replacing trace*, which sets a default trace of 1, drops a trace if it falls below some threshold α, and otherwise decays the trace by a value of $c = \gamma\lambda$:

$$\tau_t[s] = \begin{cases} \gamma\lambda\tau_t[s] & \text{if } \gamma\lambda\tau_t[s] < \alpha \wedge s \neq s_t \\ 0 & \text{if } \gamma\lambda\tau_t[s] \geq \alpha \wedge s \neq s_t \\ 1 & \text{if } s = s_t \end{cases}$$

Note that in our implementation, we dropped γ and let λ be the sole controller of trace decay.

5.3 Q-Learning with Experience Replay

5.3.1 Q-Learning

Q-learning is a model-free, off-policy approach in which the estimated utility of being in a state and taking an action is updated towards a bootstrap estimate of the true return. The primary difference between Q-learning and SARSA is the former's off-policy nature. At each time step, the next action is selected using the acting policy π (in our case, ϵ-greedy). Q-values, however, are updated towards an alternative action suggested by an optimal policy. More specifically, Q-learning attempts to minimize the following objective under function approximation:

$$min_w \sum_{(s,a,r,s')}\Big(Q_{opt}(s,a;w) - (r + \gamma\ max_{a'}Q_{opt}(s',a')))\Big)$$

It does so with the following SGD update step:

$$w_{t+1} = w_t - \Big(Q_{opt}(s,a) - (r+\gamma\ max_{a'}Q_{opt}(s',a'))\Big)\phi(s,a)$$

5.3.2 Experience Replay

As formulated, there are several problems with Q-learning:

- Q-learning is a special case of off-policy learning: both the target and behavior policies are allowed to improve over time. This injects instability into each update because we are moving Q towards a shifting target.
- Gradient descent is not sample efficient. We throw away each experience tuple after a single update.
- Adjacent experience tuples are highly correlated, leading to serious multicolinearity in our training data.

We can correct for these problems by introducing a pair of modifications to Q-learning: *replay memory* **[1]** and *fixed Q targets* **[7]**. A replay memory is a cache of agent experience ((s, a, r, s') tuples) acquired by the acting policy. During learning, the algorithm samples from this experience to make its parameter updates. Our replay memory is a limited-memory system. When the memory has reached capacity, past experiences are replaced randomly. Fixed Q targets refers to the process of periodically freezing the parameters of our Q_{opt} function approximator, and using this static set of weights to compute targets. These fixed targets have been shown to yield in more stable updates **[8]**.

5.4 Policy Gradients

Recall that SARSA(λ) and Q-learning seek to update the action-value function Q using a policy generated directly from Q (i.e. ϵ-greedy). Policy gradient methods, on the other hand, seek to directly parameterize the policy itself: $\pi_\theta = p(a|s;\theta)$. In practice, policy gradients have been shown to converge quickly. Recent work suggests these methods might be the current state-of-the-art in the 2d game-playing domain **[9]**. As Karpathy said, "Q-learning is *so* 2013" **[5]**.

The implied PG objective is the average reward per timestep:

$$J(\theta) = \sum_s d^{\pi_\theta}(s)\sum_a \pi_\theta(s,a)r_s^a$$

where r_s^a is the reward from being in state s and taking action a, and $d^{\pi_\theta}(s)$ is the Laplace-smoothed asymptotic distribution of the Markov chain generated by π_θ. What this means is the following: run out many games $g = 1\cdots k$ and count the number of times you were in state s with $c_g(s)$; then normalize through to get probabilities; finally, smooth by layering on a count of λ to every possible s:

$$d^{\pi_\theta}(s) = \lim_{g\to\infty}\frac{c_g(s)+\lambda}{\sum_{s'}c_g(s')+\lambda}$$

The process of finding the gradient of the policy ($\nabla_\theta J(\theta)$) and perform stochastic gradient descent is a totally tubular field of research. Direct estimates of $J(\theta)$ would require that we run a huge number of games, which would be computationally expensive. There are several methods for calculating looser approximations of that gradient. We chose to implement that of likelihood ratios. This method is the most conceptually simple, and has been shown to converge nicely in several domains **[10]** **[14]**.

With this method, the policy gradient is defined as

$$\nabla_\theta J(\theta) = \nabla_\theta\Big(\sum_s d^{\pi_\theta}(s)\sum_a \pi_\theta(s,a)r_s^a\Big)$$
$$\approx E_{\pi_\theta}\Big[\nabla_\theta log\ \pi_\theta(s,a)q_t(s)\Big]$$

Where $q_t(s)$ is an unbiased sample of the eventual return at state s. Computing this leads to the following high-level algorithm:

1. Use θ to generate a log distribution over actions (log $p(a|s;\theta)$). In our case, we used a deep neural network.

2. Sample and execute an action from this distribution. Record the log probabilities, state, action, as well as any other factors (e.g. hidden unit activations) needed to backpropagate at this step.

3. Repeat steps 1 & 2 until the environment expels an eventual reward r (until the agent has won, lost, broken a brick, or returned a ball).

4. At this point, fill in the reward received for all the $q_t(s)$'s while traversing backwards through cached history.

5. When a batch's worth of eventual rewards have been filled in, fill in gradients and backpropagate by turning counts into log probabilities.

A cursory analysis may find this algorithm flawed. An agent could make many competent actions that should be encouraged before loosing a game (thus filling in a negative gradient for that point in time, discouraging those actions). In practice, however, positive rewards succeed a net majority of adept actions so on average, competency is encouraged.

5.5 Function Approximation

5.5.1 Linear Approximation

Models are parameterized with one value for each feature. The output of value approximation is a linear combination of features and weights:

$$\hat{Q}(s,a;\theta) = \theta^{\intercal}\phi(s,a)$$

We attached this function approximator to SARSA(λ) and Q-learning.

5.5.2 Deep Feed-Forward Neural Network

DNNs are the basic form of feed-forward neural network. They are composed of a series of fully connected layers where each layer takes on the form

$$\boldsymbol{y} = f(\boldsymbol{\theta x} + \boldsymbol{b})$$

Where

- $\boldsymbol{x} \in \mathbb{R}^n$ is a vector of inputs (e.g. from a previous layer).
- $\boldsymbol{\theta} \in \mathbb{R}^{k\times n}$ is a matrix of parameters.
- $\boldsymbol{b} \in \mathbb{R}^y$ is a vector of biases.
- $\boldsymbol{y} \in \mathbb{R}^y$ is an output vector.
- $f(\cdot)$ is some nonlinear activation function, e.g. *tanh*, *ReLU* or the sigmoid function.

The output of value approximation is set to that of the last layer of the network. We attached this function approximator to Q-learning, and used it as a policy network to generate distributions in our policy gradients implementation.

6. RESULTS & DISCUSSION

6.1 Agent Performance

Note that for all experiments, "performance" is measured by average points per game (ppg) over many games. Furthermore, for statistical power, we replicated our experiments and computed mean ppg (μ_{ppg}).

We first investigated the effect of modulating the parameters of our experience replay and SARSA(λ) implementations with the hope that this would yield insight into algorithm behavior. From Figure 1, it is apparent that increasing memory size and sample size slightly improves performance, albeit by largely insignificant margins. The biggest jump is between groups with a memory size of 100 and all others. 100 steps corresponds to approximately 3.5 seconds of gameplay. This suggests that the algorithm is still bootstrapping on the most recent inputs but in shuffled order and explains why this setting underperformed. A size-100 memory bypasses one of the primary advantages of experience replay we seek: data efficiency and reusability. These findings align with that of Karpathy [**5**], who found that larger memories and sample sizes are generally better than smaller.

An additional analysis we performed on our replay memory was principal components analysis (PCA) over 100-item batches of feature vectors sampled from a 10000-capacity memory and sequential gameplay. Averaging over 5,000 episodes, we found that the first principal component explained 53% of the variance of the standard, sequential Q-learning data but only 24% of the replay data. This indicates that our replay memory samples were indeed less correlated than those in sequential approaches.

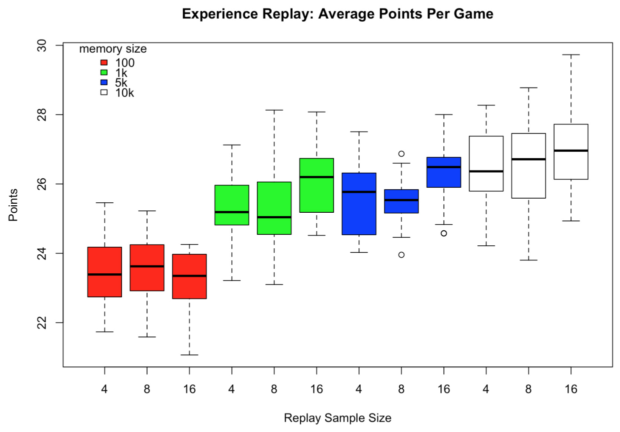

Figure 1. Effect of replay memory size and experience sample size on Q-learning performance. These data are from 24 replicates of 1000 test games following 2000 training games. The 864,000 diagnostic games were split between four concurrently running threads and took 23 hours to complete.

We conducted another set of experiments by varying two parameters of SARSA(λ): trace decay rate (λ) and trace eligibility threshold (α). We tested 12 combinations of these two parameters over 24 2000-train-game, 1000-test-game replicates. Interestingly, no μ_{ppg} measurements were statistically different (ANOVA test, $p = 0.82 > 0.05$). However, the results did not indicate homogeneity of variance (Bartlett test, $p = 2.2e^{16} < 0.05$). These data make sense considering the long delay between action and reward in the breakout setting: even with large λ, the eligibility trace may not stretch back far enough to pick up on multiple rewards. Regardless of the trace update mechanism, agents still behave in a myopic manner; only the most recent reward governs the direction and magnitude of the gradient. Furthermore, it makes sense that all the performance distributions had unique shapes, because trace values are modulated differently with each parameter setting. Since trace values don't change consistently, their values vary at varying amounts. Thus, gradients and downstream computations like Q-values and actions have varied variance as well.

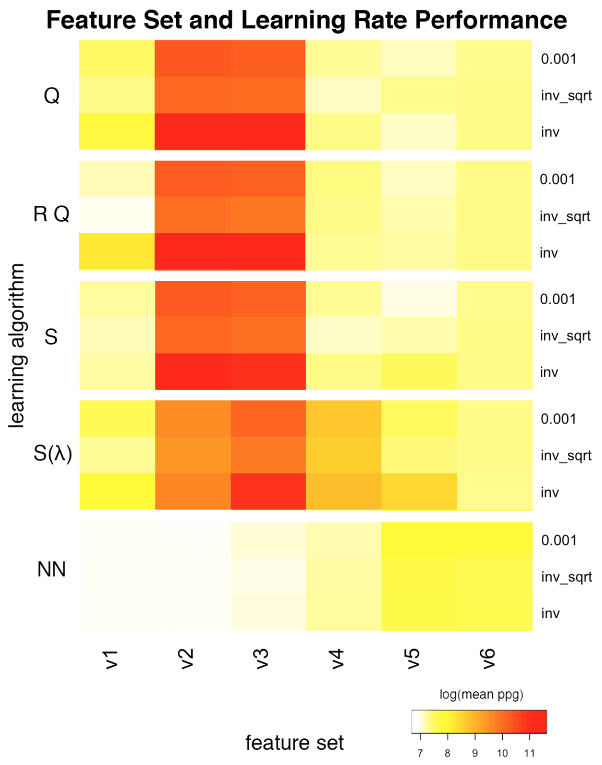

Figure 2. Performance of Q-learning (Q), Replay Q-learning (R Q), Neural Network Q-learning (NN), SARSA (S), and SARSA(λ) on each feature set and learning rate function. Performance is measured by $log(\mu_{ppg})$ over 8 replicates of 2000 training games ($\epsilon = 0.3$), followed by 1000 test games ($\epsilon = 0$) for each experimental group. These 2.1 million diagnostic games were run concurrently across 4 threads and took 34 hours to complete.

Our next group of experiments was concerned with hyperparameters, specifically feature sets and learning rates. As discussed in section 4, we implemented a variety of continuous and discrete feature sets with varying degrees of variable expansion and interaction. Furthermore, we implemented several learning rate functions:

- *Constant* learning rates which have been shown to work well in practice **[6]**.
- $1/\sqrt{\#\ iterations}$, which was described in lecture.
- $1/\#iterations$, which has nice theoretical properties like GLIE (greedy in the limit with infinite exploration) **[11]**

From Figure 2, it appears that among all the non-neural network learners, our $v2$ and $v3$ feature sets significantly outperformed all others. This makes sense, because those feature sets provided highly distilled information to the agent, e.g. "is the ball to the right or left of the paddle?". Since each feature is highly informative and largely discrete, linear combinations of their associated weights correspond to simple, yet powerful, game-playing rules. See Section 6.2 for a discussion of weights for these feature sets. Neural networks performed the best on $v5$ and $v6$. This makes sense, because the neural network we implemented expected a continuous input space. $v5$ and $v6$ were the only continuous feature sets with significant interaction terms and contained the largest amount of information from the game state.

Between the $v2$ and $v3$ (and excluding neural networks), the $v2$ outperformed $v3$ in 8 of the 12 experimental groups. Recall that $v2$ has binary indicator variables on the balls relative position (left / right) and direction of movement (right / left). $v3$ swaps $v2$'s positional flag for a smooth *tanh* function, which encodes both left/right (with sign) *and* distance (with absolute value). We hypothesize that this extra information encourages planning when the ball is far away, thereby increasing agent performance. It appears, however, that this real-valued input does little to change agent behavior. In retrospect, this flat-lining performance makes sense because we didn't include basis expansions of $v3$'s *tanh* feature. For example, consider two situations where the agent has a linear function approximator. $v3$ sends a strong *tanh* signal for distant left at this moment. For $v2$, the binary flag for left is raised. Both cases trigger the same agent behavior because a single weight is activated.

The notable exception is SARSA(λ), where $v3$ outperformed $v2$ across all learning rates. This may be due to SARSA(λ)'s farsightedness: the algorithm can indirectly make use of the higher resolution positional information through the historical rewards buried in its eligibility traces.

Examining Figure 2 on a per-row basis, we see that a learning rate of $1/\#$ outperformed its counterparts in every case (SARSA(λ) seems close in log-space but $1/\#$ outperformed 0.001 by a margin of 2192 ppg).

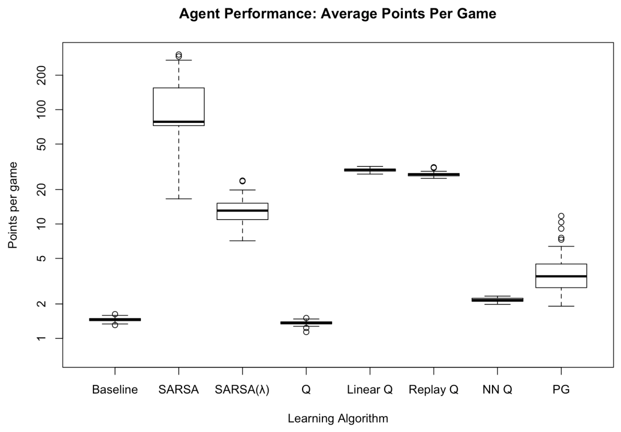

Figure 3. Performance of each algorithm, as measured by average points per game over 64 replicates of 2000 test games ($\epsilon = 0$) after 4000 training games ($\epsilon = 0.3$). These 3,584,000 diagnostic games were split between four concurrently running threads and took 37 hours to complete.

For our next set of experiments, we took the highest-performing feature set/learning rate combinations for each learning algorithm and compared their performance on a global basis (Figure 3). We included a random baseline and policy gradients agent in this analysis. It is clear the SARSA performed the best, followed by Q-learning and replay Q, then SARSA(λ), policy gradients, neural network Q, and finally discrete Q-learning, which was the only agent that failed to outperform the baseline.

It is immediately evident that μ_{ppg} distributions follow two patterns: tightly packed and diffuse. The reason for this discrepancy in variance is because those agents with wider distributions won games, but winning games is a rare occurrence. Recall that 1000 points are awarded for winning a game. SARSA, the top performer, won the most games during test time (123), so its μ_{ppg} distribution had the largest variance.

SARSA outperformed all other algorithms by a wide margin. This may be due to the close relationship between the value function and bootstrap updates. It is likely that SARSA(λ) under-performed because of the same reasons as discussed in the λ-varying experiments: due to the nature of the problem setting, SARSA(λ) behaves myopically and is essentially a diluted version of SARSA.

It is not surprising that discrete Q-learning failed to outperform the baseline; the state spaces of the breakout game are huge. What is more interesting is that linear and replay Q-learning performed very similarly, with linear Q-learning slightly edging out its counterpart. Though these data go against the general results of RL approaches applied to Atari games [**8**], they make intuitive sense for the Breakout game specifically. Competent gameplay results from sequences of good actions, and the task of generating such sequences may benefit from sequential game steps, even though they are correlated. Surprisingly, the neural network Q-learning agent did not fair as well as SARSA. Though it outperformed the baseline, it did so by a relatively small margin. This is likely because of insufficient signal in the feature space or multi-colinearity among inputs (this is a more serious problem for backpropagation than simple linear regression [**3**]).

It should be mentioned that policy gradients, though recognized as the current "state-of-the-art" for many reinforcement learning tasks [**9**], failed to deliver when applied to our Breakout game. Though policy gradient agents won a total of 22 games in our test runs, they generally underperformed. This may be due to similar reasons as the neural network Q agent, because we used a neural network to represent $p(a|s;\theta)$.

Our final investigation (Figure 4) involved elucidating the speed at which each learning algorithm "converges" to the center of each μ_{ppg} distribution observed in Figure 3. All curves exhibit increasing variance with time because the variance of previous estimates compounds as agents adjust their parameters.. In (A) we see the effect of static Q-targets on replay memory. The learning curve for this algorithm is rough, spiking whenever the system switches to a new set of high performance static weights. Q-learning appears to "converge" the fastest, perhaps because it is learning the values of its acting policy. This kind of target policy/acting policy synergy may have made the error surface shallower for our breakout implementation. SARSA and SARSA(λ) appear to behave similarly, likely because they chase the same objective function with identically formulated update steps.

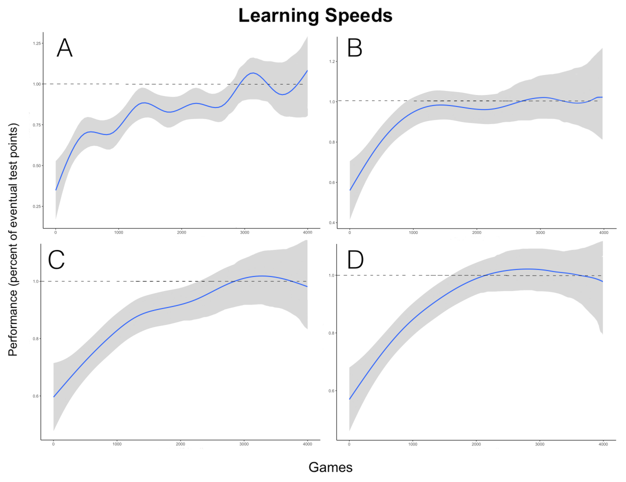

Figure 4. Analysis of the speed at which (A) replay Q-learning, (B) linear Q-learning, (C) SARSA, and (D) SARSA(λ) learn. These trajectories were obtained by training 10 replicates of each algorithm over 2000 games ($\epsilon = 0.3$). Every 50 games we spawned a thread and copied the current parameters into a concurrent series of 200 test games where $\epsilon = 0$. The y-axis depicts test performance of each test series as a percentage of the "goal" performance (mean observed performance from the experiment in Figure 1, indicated with dashed lines). Standard error of each estimate is shown in gray. The x-axis of each plot depicts training game index.

6.2 Model Evaluation

When looking into the weights of the models we made, we can understand how the agent learned how to play the game. We will use the example of SARSA using ϕ_2, because this feature set is high-performance and easily interpretable. The weights after 2000 training games are as follows. The numbers are not always the same but the differences between values are close to each other in all the SARSA models:

Position	**Ball Movt.**	**Action**	**Q-value**
Left	Left	Left	**0.01680**
Left	Left	Right	0.00976
Left	Left		0.00665
Left	Right	Right	**-0.03902**
Left	Right	Left	-0.03903
Left	Right		-0.04807
Right	Left	Left	**-0.01773**
Right	Left	Right	-0.01774
Right	Left		-0.02507
Right	Right	Right	**-0.05014**
Right	Right	Left	-0.05425
Right	Right		-0.05981

Position is the relative position of the ball relative to the paddle, Ball movt. is the current direction the ball is going towards and Action is the action that is taken.

We notice that when the ball is moving away from the paddle (position and movement are both same direction), the optimal action is to follow the ball and by a huge margin. On the other hand, when the ball is moving towards the paddle, the agent is a little more confused: the values for left and right actions are very close together but it seems it is better not to try to go under the ball in this case but rather to follow the movement of the ball.

Using this reduced set of features was insightful for debugging and understanding how the agent learns from the game.

7. CONCLUSION

In this paper, we implemented a variety of reinforcement learning algorithms and evaluated their performance on a custom implementation of Breakout, a classic Atari 2600 game. We found that while experience replay and eligibility trace mechanisms offer tantalizing theoretical benefits, they generally decrease performance on average. Furthermore, we demonstrated that highly informative features aid the ability of the agent to learn from these algorithms.

Neural networks were not as effective as simple linear function approximators. Future efforts could investigate additional feature sets and neural network topologies.

In the end, All of our algorithms except for discrete Q-learning outperformed the random baseline. Simple learning algorithms like SARSA paired with parsimonious feature sets did the best, outperforming their more complex cousins. As Da Vinci said, “simplicity is the ultimate sophistication”.

8. REFERENCES

[1] Adam, Sander, Lucian Busoniu, and Robert Babuska. “Experience replay for real-time reinforcement learning control.” *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.2 (2012): 201-212.

[2] Defazio, Aaron, and Thore Graepel. “A comparison of learning algorithms on the arcade learning environment.” *arXiv preprint* arXiv:1410.8620 (2014).

[3] De Veaux, Richard D., and Lyle H. Ungar. “Multicollinearity: A tale of two nonparametric regressions.” *Selecting Models from Data.* Springer New York, 1994. 393-402.

[4] ”Human-level Control through Deep Reinforcement Learning.” DeepMind. N.p., n.d. Web. 12 Dec. 2016.

[5] Karpathy, Andrej, “Deep Reinforcement Learning: Pong From Pixels”. *karpathy.github.io.* N.p., 2016. Web. 7 Dec. 2016.

[6] Kuan, C-M., and Kurt Hornik. “Convergence of learning algorithms with constant learning rates.” *IEEE Transactions on Neural Networks* 2.5 (1991): 484-489.

[7] Mnih, Volodymyr, et al. “Playing atari with deep reinforcement learning.” *arXiv preprint* arXiv:1312.5602 (2013).

[8] Mnih, Volodymyr, et al. “Human-level control through deep reinforcement learning.” *Nature* 518.7540 (2015): 529-533.

[9] Mnih, Volodymyr, et al. “Asynchronous methods for deep reinforcement learning.” arXiv preprint arXiv:1602.01783 (2016).

[10] Peters, Jan, and Stefan Schaal. “Reinforcement learning of motor skills with policy gradients.” *Neural networks* 21.4 (2008): 682-697. APA

[11] Silver, David. “Reinforcement Learning Course: COMPM050/COMPGI13”. 2015. Lecture.

[12] Sutton, Richard S., and Andrew G. Barto. “Reinforcement Learning: An Introduction”. Cambridge, MA: MIT, 1998. Print.

[13] Tesauro, Gerald. ”TD-Gammon: A Self-Teaching Backgammon Program.” Applications of Neural Networks (1995): 267-85. Web.

[14] Williams, Ronald J. “Simple statistical gradient-following algorithms for connectionist reinforcement learning.” Machine learning 8.3-4 (1992): 229-256.