CS221 Final Project: Learning Atari

David Hershey, Rush Moody, Blake Wulfe
{dshersh, rmoody, wulfebw}@stanford

December 11, 2015

1 Introduction

1.1 Task Definition and Atari Learning Environment

Our goal for this project is to enable a computer to learn to play a set of Atari 2600 video games based solely on the visual output from those games. Video games pose a challenging sequential decision making problem for computers, which often reflect reduced versions of real-world problems. For this reason, algorithms capable of operating on this visual data and learning to make intelligent decisions are highly desirable, as they can potentially be generalized to more difficult, real-world settings. This task is modeled as a Markov Decision Process represented by the Atari Learning Environment (ALE)[1], a machine learning framework that simulates the video games and allows for interaction with the game state. We employ two model-free, bootstrapping reinforcement learning algorithms: Q-learning with replay memory and SARSA(λ). The state-space of the games is too large for the learning algorithms to explore directly, so we reduce its size by extracting objects from each game screen and classifying these objects with an unsupervised learning algorithm. We then generate features based upon these objects and their relationships with each other. We use both linear and neural network based function approximators to transform these features into value estimates used by the selected algorithms.

We also used this project for CS229. For CS229, we focused our efforts on the extraction of features from the game screen, the hyper-parameter tuning of each of our reinforcement learning algorithms, and the analysis of replay memory. For CS221, we considered additional feature extraction methods, the reinforcement learning implementations themselves, namely development of both the linear and neural network-based SARSA(λ) and Q-learning models, and the test MDPs to evaluate these models.

2 Existing Work

The idea of enabling computers to learn to play games has been around at least since Shannon proposed an algorithm for playing chess in 1949 [9]. TD-Gammon, introduced in 1992, helped popularize the use of reinforcement learning - specifically temporal-difference learning in conjunction with function approximation - in enabling computers to learn to play games [10]. Development of a general video game playing agent specifically for the Atari 2600 was introduced in 2006 by Naddaf [8]. Bellemare et al. [1] formally introduced the ALE framework and established a set of baselines using SARSA(λ) applied to tile-coded features as well as search-based methods that explore the game through saving and reloading its RAM contents. Planning-based methods relying on this ability have achieved the state-of-the-art results in playing Atari games [4]; however, we focus here on methods that use information available to human players and that can execute in real-time. Within these constraints, Defazio et al. [3] compared the performance of linear, model-free algorithms including SARSA(λ), Q(λ), next-step and per-time-step maximizing agents, Actor-Critic methods, and others for Atari game playing, finding that the best performing algorithm differed significantly between games. Recently, Minh et al. [7][6] have applied what they call Deep Q-Networks (DQN) to the task of learning to play Atari games. This work uses Q-learning with nonlinear, specifically deep neural network, function approximators to achieve state-of-the-art performance on a large variety of Atari games. The DQN avoids the potential instability of nonlinear models used to approximate action-value functions by randomly sampling from a replay memory in performing parameter updates and through the use of a semi-stationary target function. Using these methods, the researchers were able to effectively train a convolutional neural network function approximator to transform image input into estimated state-action

values. In this paper we consider an adapted approach to the DQN as well as the conventional approaches using SARSA(λ) models.

3 Methods and Models

3.1 Object Feature Extraction

The state-space of a game is the set of possible screens that the game can produce. Modeling this state-space as a grid of pixels is possible; however, this leads to an large and difficult to explore state-space. With sufficient computational power this is possible, however it is possible to pre-process these pixels with insight into what composes a typical game screen. In order to reduce the state-space of our game models, we elected to extract objects from the game screen rather than using raw pixel data as learning input. This follows from an intuitive understanding of how a human agent would play a game: identify the objects on the screen, their relative positions and velocities, and then decide on an action.

3.1.1 Blob-based

Our initial attempt at identifying objects performed similar-colored blob detection using a modified breadth-first search over the screen pixels. We attempted to identify the background color of the game based on the most common single pixel color, and then perform BFS starting at any seed pixels (a pixel which doesn't match the background color and hasn't already been included in an object), where we mark any adjacent pixels within a given color tolerance to be part of our current blob. This technique proved fairly accurate when tested on Breakout, and has some desirable qualities for object classification purposes (namely, that we can store and return the exact location and color of all pixels within an object during detection), but proved to be significantly slower and less general than our second attempt, where we leveraged the OpenCV image library to perform edge detection as the basis of our object identification.

3.1.2 OpenCV Contour Detection

The first step in extracting visual features from the raw pixel data is identifying regions on the screen that correspond to unique entities. In most Atari games, the screen has very distinct, often monocolored features. We've elected to use methods available OpenCV, an open source computer vision tool, to extract these features. OpenCV is used to detect edges in an image, then draw contours around distinct closed edges. We then use information about these contours, such as their length, enclosed area, and the average color contained in a contour as input to a clustering algorithm in order to classify features.

3.1.3 Feature Clustering

We are using DBSCAN clustering to classify the features on a given screen. DBSCAN models clusters as points with many nearby neighbors. We use the implementation of DBSCAN in scikit-learn, an open source machine learning library. We selected this algorithm as it estimates the number of clusters while it clusters the data. This allows it to discover new classifications while running, which is important as new entities could appear on the screen at any timestep.

DBSCAN relies on two parameters, the minimum cluster size and the euclidean size of the clusters. We've set the minimum clusters size to one, so that each outlier is classified into a new category. This is important as there is often only one instance of some entities in a game. The euclidean size of the clusters is automatically estimated by the algorithm.

It is critical for learning purposes that feature labels be consistent from frame to frame. Since DBSCAN is an unsupervised algorithm, it does not guarantee that labels are consistent. In order to address this issue, we've applied a consistency checking algorithm to the output of DBSCAN. This algorithm works by storing 10 examples of each label it has seen so far, and then checking the classification of these labels every time DBSCAN is run. A mapping between the most recent DBSCAN labels and the time-invariant labels is then constructed, which is used to re-map every label to a time-invariant label.

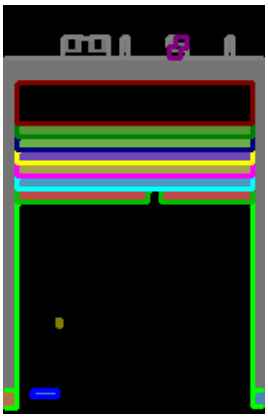


Figure 1: Output of DBSCAN clustering, where each outline color represents a unique label.

3.1.4 Object Tracking

The final step of the object extraction is to track objects from frame to frame. This tracking allows us to measure the derivatives of each object on the screen, which is an important feature for the learning agent. Tracking also allows us to differentiate between different instances of a single entity type on the screen, which can be useful for some features. The tracking algorithm is a simple greedy algorithm that matches each entity on the previous screen with the closest entity sharing the same label on the current screen.

3.2 Linear Function Approximation Models

Our object tracking and classification subsystem gives us a set of n different (object_id, object_position, object_direction) tuples, where each tuple contains the category ID of the object, the center (x,y) position of the object, and the current direction of the object (which has two components, x and y, each of which can take the value -1, 0, or 1). Although we could feed this state information directly into our learning algorithm, for any non trivial game the resulting state space is enormous, especially when there are more than 2 objects; for example, if the screen is only 50 pixels square (smaller than the Atari screen) and we have only two object categories, there are already more than $((50^2)^2 * (3^2)^2) = 506{,}250{,}000$ possible states, ensuring that we can't rote-memorize the state values within a reasonable time frame. Accordingly we turn to function approximation, with the idea that this will allow us to exploit similarities between states to learn about states that we have not actually reached while playing the game itself. Our choice of features here is critical, as we want to select features that 1) convey useful information about the state, 2) will provide a significant decrease in sparsity versus the identify feature extractor, 3) generalize well between different scenarios and games.

3.2.1 Feature Selection

Broadly speaking, the features we extract from the game state fall into two categories, but all of them follow the feature template paradigm we discussed in class and are parametrized by our identified objects. The first category simply contains information about each of the objects currently on the screen, combining the category ID with the x and y position as well as the direction values into several different templates. This allows us to capture the key basic information about the state, but may fail to capture more complex interactions between objects, creating the impetus for our second class of feature: pairwise feature template between different objects on the screen. These features combine one or more pieces of information from different objects into a single feature template, increasing the sparsity of the feature in order to hopefully improve its usefulness. The pairwise templates range in complexity from merely having the x and y difference between the two objects to combining the y position and y direction of the first object with the x position difference to another object. To help combat the sparsity of these templates, we also shrink the state space by essentially rounding the x and y values to some set precision.

3.2.2 Learning Agents: Q-Learning, SARSA, SARSA(λ)

Once we define our feature extractor the next task at hand is to pick a learning algorithm to apply to this setting. Our review of past research indicated that model-based reinforcement learning algorithms would likely not achieve high performance [3]. For example, [5] found fitted value iteration to perform poorly on Atari games due to a rank deficiency of the feature matrix. As such, we focus on model-free algorithms here, specifically Q-learning with memory replay and SARSA(λ). We use function approximation for both algorithms. For the first we use both linear and neural network state-action value approximators, while we only use linear approximators for the second.

3.2.3 Q-Learning with Replay Memory

Q-learning is an off-policy, bootstrapping algorithm that learns a mapping between state-action pairs to the optima Q values of those pairs. It specifically attempts to minimize the following objective when using function approximation:

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

With the following parameter update equation:

$$w(i+1) := w(i) - [Q_{opt}(s,a,;w) - (r + max_{a' \in actions(s')} Q_{opt}(s',a';w))](s,a)$$

Using a replay memory this algorithm stores a history of (state, action, reward, newState) tuples, and then during learning samples from this memory according to some distribution to make parameter updates. Data is collected according to some policy, commonly ϵ-Greedy. With a limited-capacity replay memory, past experiences can be replaced randomly or with a different replacement policy.

3.2.4 SARSA(λ)

SARSA is an on-policy, bootstrapping algorithm that learns a mapping between state-action pairs and the Q value of the policy on which it currently operates. It specifically attempts to minimize the following objective when using function approximation:

$$min_w \sum_{s,a,r,s',a'} Q_{pi}(s,a;w) - (r + Q_{pi}(s',a';w))$$

With the following parameter update equation:

$$w(i+1) := w(i) - (Q_{pi}(s,a,;w) - (r + Q_{pi}(s',a';w)) * (s,a)$$

Updating these parameters results in an update to the models policy. With SARSA(λ), the algorithm maintains a set of eligibility traces τ, one for each parameter of the function approximator. These traces determine the extent to which each parameter is eligible to be assigned credit for a given reward encountered by the agent. Each τ value is updated each step by a factor of λ resulting in an exponential decay of the impact of rewards on given parameters over time. For our experiments we use a replacing trace, which performs the following update at each timestep:

$$e_t(s) = \begin{cases} \gamma \lambda e_{t-1}(s) & \text{if } s \neq s_t \\ 1 & \text{if } s = s_t \end{cases}$$

3.3 Neural Network Function Approximation Models

Designing higher-order cross-feature templates is time-consuming and depends on domain-specific knowledge. It is possible to avoid these issues through the use of a neural network function approximator that automatically learns relevant, high-level features from the original input representation. For example, Minh et al. [6] used a Convolutional Neural Network (CNN) to transform image pixels directly into state-action value estimates. Their networks generally contained a set of convolutional layers followed by a set of densely connected, Rectified Linear Unit (ReLU) layers.

We hypothesized that we could replace the initial convolutional layers with traditional object detection and tracking methods in order to extract basic features, and then pass those features directly into a set of densely-connected ReLU layers. Manually extracted features are clearly less informative than those extracted by a CNN because they make certain assumptions about what information is actually important for playing a game, whereas the CNN features are derived directly from learning this information for each specific game. Therefore, by replacing the CNN with manually extracted features we expected to achieve diminished results, though at significantly reduced computational cost.

A complication of passing manually extracted features into a neural network is that this demands some manner of ensuring the neural network is invariant to the order of the input features. To address this we assumed a constant ordering of the input objects, ensured using the extracted class and within-class ids of each object.

3.3.1 Network Structure and Operation

We developed a neural network using Theano [2] that transforms object features representing each state into action-value estimates for each action. This is accomplished by associating each of the output values of the network with one of the possible actions of the agent. For a given state, taking the estimated optimal action is then accomplished through an argmax over the output values of the network. During backpropogation, the output units for other actions are not considered because they do not contribute to the loss, thereby allowing us to perform reinforcement only on the action taken.

We evaluated a number of different network structures and activation functions. Specifically, we considered networks with two, three, and four densely-connected layers both with diminishing and constant number of hidden units over layers. For activation functions we considered ReLU ($\max(x, 0)$), leaky ReLU ($\max(x, \alpha * x)$, with $0.01 < \alpha < 0.2$) and tanh activation functions.

3.3.2 Training Method and Hyperparameters

The DQN trained by Minh et al. used two primary adaptions to the training process that they claim enabled their network to learn effectively. First, the DQN used a replay memory. This means that at each training step, the agent randomly samples data from a history of (state, action, reward, newState) tuples, and performs parameter updates based on this data. This approach allegedly improves training by eliminating correlations between consecutive screens and by more efficiently using collected data. Notably, using this approach requires an off-policy model, which motivated the researchers choice of Q-learning. Second, during training, the network uses target values derived from static, previous values of its weights rather than its current weights. These previous values are then periodically updated to the current values. By doing this, the researchers claim learning proceeds in a more stable manner.

We evaluated memory replay on a test MDP, which is discussed more in the next section. We used different replay memory instantiations, varying memory capacity and sample size. Specifically, we compared two capacities - one hundred and one thousand - and two sample sizes - five and twenty - as well as a baseline model that has no replay memory. All models operated with the same exploration epsilon value (0.3), learning rate (0.01), and frozen target update period (500). Each such combination was run five times for 2000 trials, with the run that achieved the maximum reward from its group being selected for visualization in the graph.

Our results showed that, in the case of the test MDP, both capacity and sample size significantly impacted the performance of the learning algorithm, even when accounting for the increased number of updates performed with large-sample memories. Figure 2, showing performance over a constant number of *trials*, illustrates that the large capacity and sample size replay memory seems to benefit from a much steeper learning curve, even when compared to the smaller-capacity memory making equivalently-sized updates. Of course, this replay memory performs more updates per trail than the replay memories with smaller sample sizes, so we also compared the performance of the different memories holding the number of updates fixed. Figure 3, shows performance over a constant number of *updates*, and again illustrates that the Q-learning algorithm using the larger memory seems to learn much more quickly than both the five sample size update, and sequential update (i.e., no memory replay) approaches.

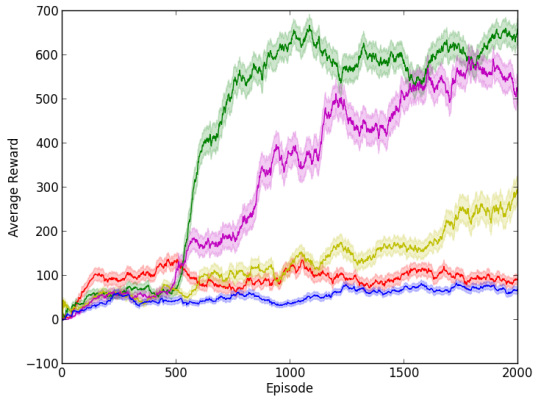


Figure 2: Effect of replay memory sample size on average reward with constant number of trials. The 1000-capacity, 20-sample memory shown in green provides the best performance, but also executes the most updates.

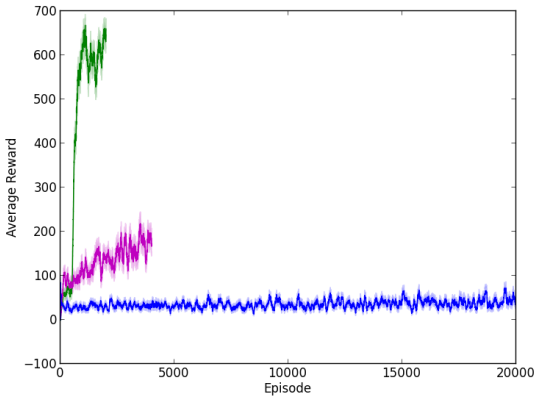
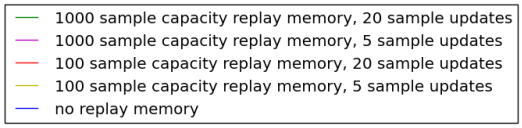


Figure 3: Effect of replay memory sample size on average reward with constant number of updates. The 1000-capacity, 20-sample memory shown in green again provides the best performance even though it performs the same number of updates.

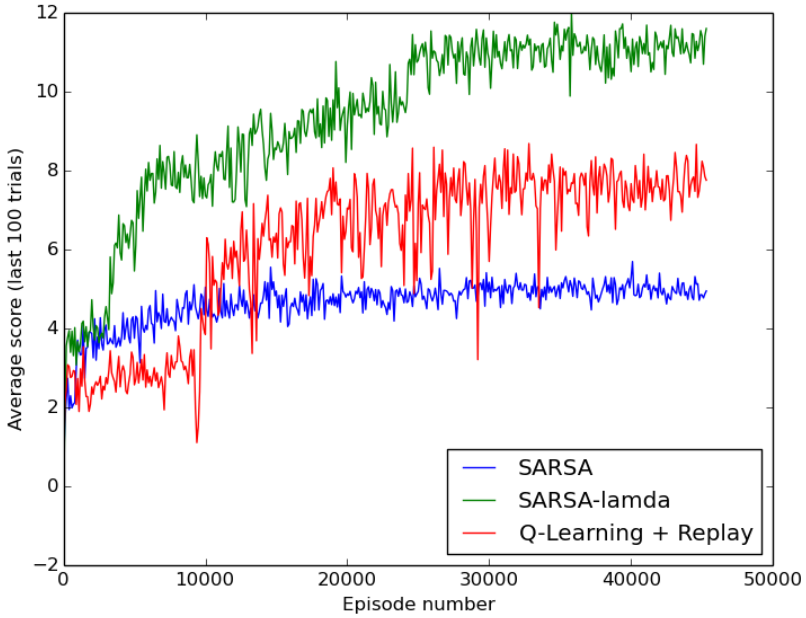


4 Results and Analysis

4.1 Linear Model

4.1.1 Test MDP Performance

To gain some insight into the behavior of our different learning agents and further explore hyperparameter tradeoffs, before testing our agents on the full Atari environment we first turned to some (much) simpler MDPs that approximated some of the challenges we expected to face on the Atari games. We first created a number line MDP, similar to the homework, where we can attempt to move left or right (with varying chances of success), and then generalized that MDP to two dimensions. Although the number of episodes required to learn differed for these MDPs, all of the agents (Q-learning, SARSA, SARSA(λ)) learned to play these games optimally. To then approximate Breakout more closely, we added a delayed gratification aspect to the game by designating a fixed, hidden "magic square" on the grid, where the agent gets a significantly higher reward at the end of the game if they have visited that magic square at any point before reaching the end. The performance of our agents on this simple game (using an exponentially decaying exploration and learning rate) over time is given in the graph below. Note that the maximum score for our MDP varies slightly (as we start in a random position and have a non-zero discount factor) but averages around 12.



As we can see from the graph, for this delayed-gratification setting the SARSA(λ) player significantly outperforms our other two agents. This result follows our intuition that such an agent is better-equipped to handle the delayed aspect of the problem by tracking a number of previous states and applying a (discounted) update to the Q-values for those states as well when we receive a reward later on. In a game such as Breakout, where the key strategic triumph (bouncing the ball back upwards off the paddle) occurs significantly before we actually see the reward for breaking a brick, we decided that such behavior would be valuable (or even essential) to adequately learning how to play the game.

4.1.2 Atari Game Results

Our model achieved significant improvement playing Breakout over time, although it seems to hit a ceiling after approximately 1250 episodes. Our results on other games were mixed; it failed to show much improvement on Asterix and Space Invaders, but did exhibit some progress learning to play Chopper Command. The plots for our performance on Breakout are shown below for various parameter value and agents, as well as our performance using the SARSA(λ) agent on other games. Our Oracle for each of these games is the performance of a human player, which for Breakout is a score of 25.8, for Asterix is a score of 8503, for Space Invaders a score of 1652, and for Chopper Command is a score of 9882 [6].

4.1.3 Feature Weight Analysis

To gain deeper insight into what our model is learning, we created a feature weight analysis tool that allows us to plot all of the weight values for a given feature template that is parameterized by some single variable, such as our position or position difference features. We then examined the weight values of several features for our Breakout-trained model that we thought would be useful for learning to play Breakout, to see whether our intuition matched the feature weights. The weights of the x-position difference feature template between the ball and the paddle are given below: Our intuition is that a valid strategy is to keep the paddle constantly positioned directly under the ball and to then track the ball's movement; as shown in the graph, there is a clear maximum around where the x difference is 0. Additionally, in the first graph we can see that there is a rapid decrease in the feature weight as the paddle moves further to the right of the ball; this matches our intuition that we would instead want to move left to correct the difference. Several other feature templates showed similar degrees of promise, but to get a more complete picture of our agent's learning, we created a more sophisticated tool for simulating game states and extracting the learned Q-Values and the resulting policy for our Breakout agent.

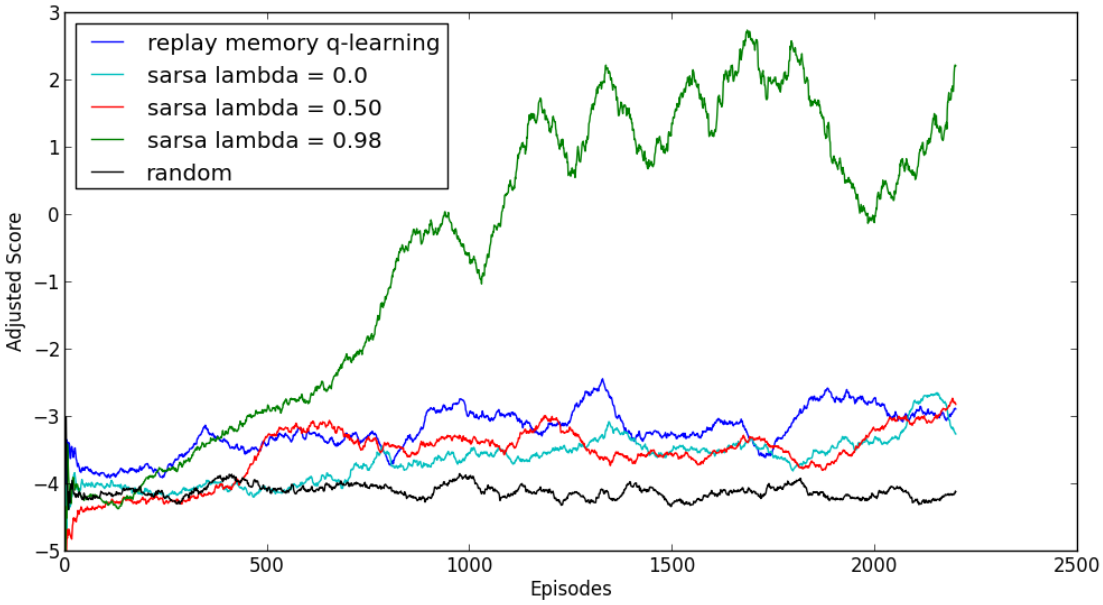


Figure 4: Breakout results for different agents and parameter settings

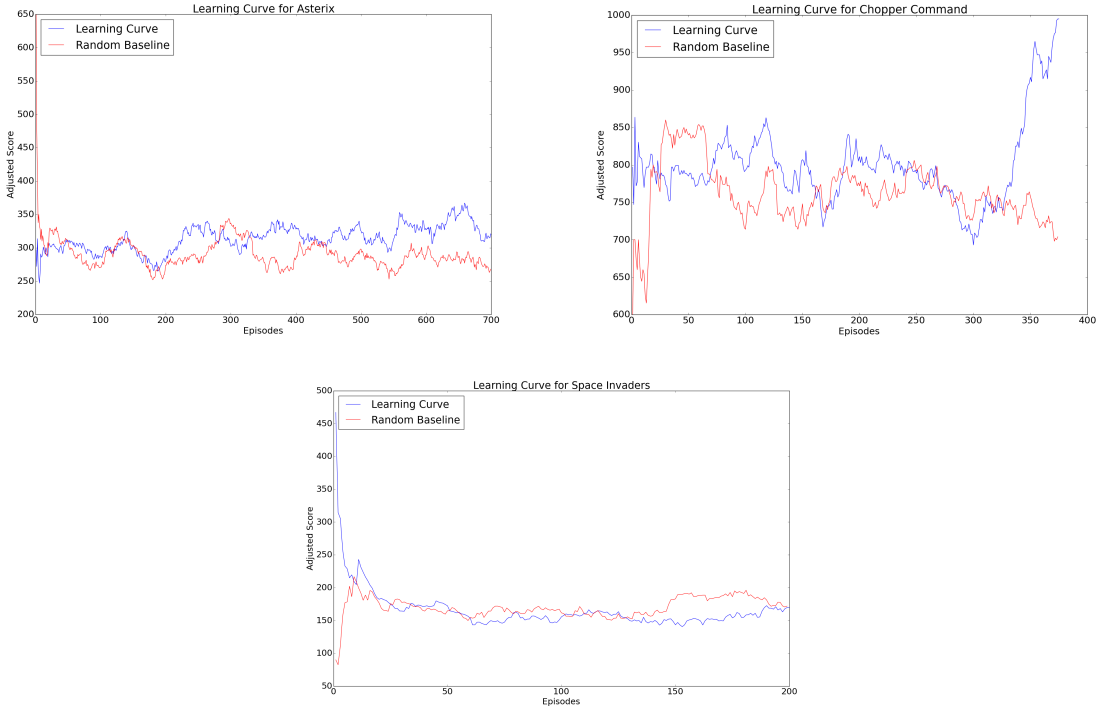
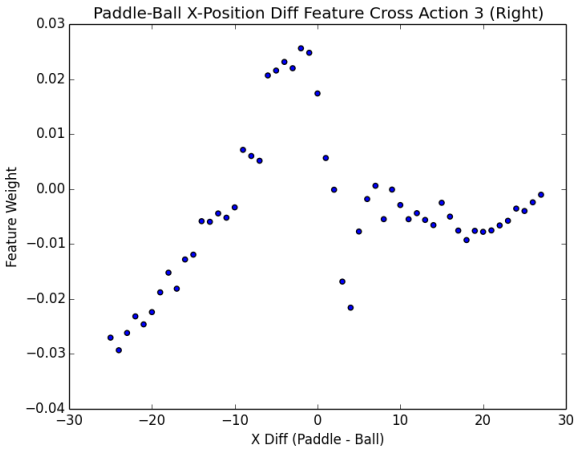
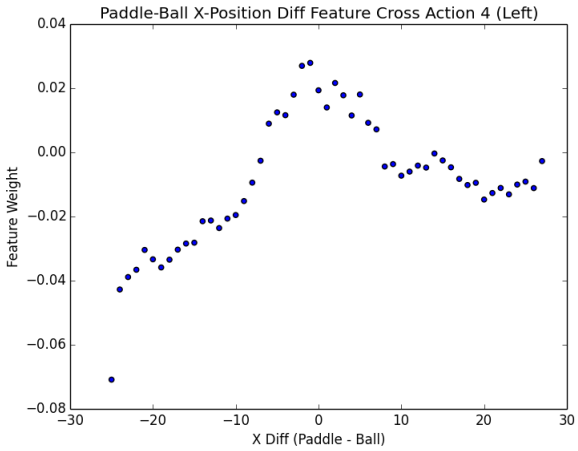


Figure 5: Learning curves for Asterix, Chopper Command, and Space Invaders.



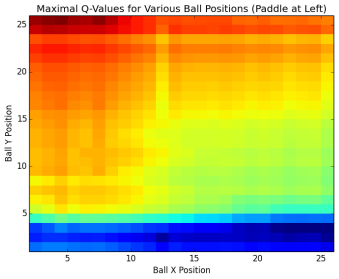
(a) X Diff. Feature Weight for Action 3 (Move Right)



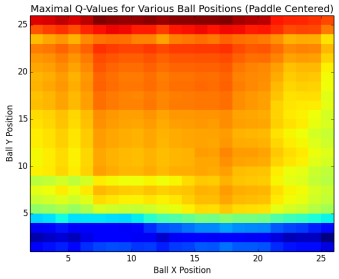
(b) X Diff. Feature Weight for Action 4 (Move Left)

4.1.4 Plots of State Values for Fixed Paddle, Moving Ball

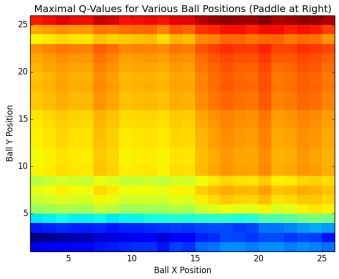
For this tool we fixed the position of the paddle and then examine the Q-values for various simulated states as we move the ball around the screen. The resulting Q-values are shown as a heat map below, where the color of a given x,y position is determined by the Q-value of the optimal action when the ball is at that x,y position, using a mock feature extractor and our learned weight vector. Red represents more favorable states, while blue is less favorable:



(a) Paddle at far left



(b) Paddle in middle



(c) Paddle at far right

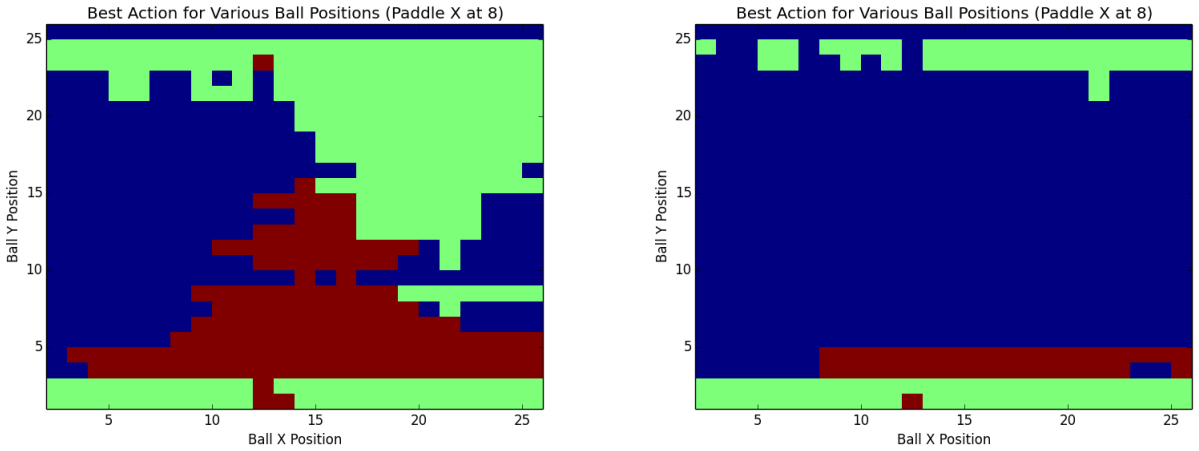
Here again our intuition is matched by the plots: the most favorable states are when the ball is directly above the paddle and up near the bricks at the top of the screen for all three of these paddle positions, while if the ball is ever below the paddle the state value is quite negative.

4.1.5 Plots of Best Action For Fixed Paddle, Moving Ball

Using the same tool as before we were also able to create a plot of the optimal action for our learned policy for a given simulated ball/paddle state. For the plots in figure 8 below, we fixed the paddle at $x = 8$ and plotted the best action choice for various ball positions. For the first graph we set the ball's y derivative feature to indicate the ball is moving downwards and to the left. We can see that our agent knows to move left or right depending on where the ball is positioned relative to the paddle; although there is some noise in the graph, it looks fairly close to what we expect. For the second graph, we set the ball's y derivative feature to indicate upwards movement; note the drastic difference in resulting actions. Although the agent seems to almost always favor leftwards movement in this case, it is encouraging that it has learned the importance of the ball's y derivative when selecting its action.

4.2 Neural Network

As it turns out, getting a neural network to output any useful results in this setting proved quite difficult despite numerous experiments with different network structures and hyperparameter settings; we significantly departed



(a) Best Action Map, Ball moving down

(b) Best Action Map, Ball moving up

Figure 8: Best Action Maps. Blue=Left, Green=No-Op, Red=Right

in setup from the extant literature, which also made it harder to correct our course. The opaque nature of neural network hidden layer features also made it difficult to align our intuition with the trained weights, so due to paper length consideration we are deliberately leaving this section mostly empty to focus on the more content-rich parts of the project.

5 Conclusion and Future Work

In this paper we evaluated a set of reinforcement learning models and corresponding hyperparameters on the challenging task of general video game playing. We first found that, testing on a simple MDP, replay memory sample size and capacity significantly impact the performance of Q-learning. We next showed that SARSA(λ), paired with simple linear function approximation and object extraction methods, can effectively learn a simple Atari game with significantly above random performance, and that Q-learning with replay memory and a static target function achieves worse performance. Furthermore, we have showed that higher λ values tend to correspond to faster learning for this application.

The algorithm was not able to to generalize effectively to other games with potentially more complicated screen states or objective functions. The method of extracting objects from the game screen is very dependent on stable, fast extraction of visual features, and for more complex games a more robust system would be needed to achieve success with this method.

Neural networks were not as effective for function approximation as a simple linear function approximator. The approach of extracting a variant number of objects from the screen is difficult to translate into an effective input to a neural network, and adds significant complexity to the computation. Future efforts could investigate different input and layering schemes for the neural network to test its effectiveness with different parameters.

References

[1] Marc G Bellemare et al. "The arcade learning environment: An evaluation platform for general agents". In: *arXiv preprint arXiv:1207.4708* (2012).

[2] James Bergstra et al. "Theano: Deep learning on gpus with python". In: *NIPS 2011, BigLearning Workshop, Granada, Spain*. 2011.

[3] Aaron Defazio and Thore Graepel. "A Comparison of learning algorithms on the Arcade Learning Environment". In: *arXiv preprint arXiv:1410.8620* (2014).

[4] Xiaoxiao Guo et al. "Deep learning for real-time Atari game play using offline Monte-Carlo tree search planning". In: *Advances in Neural Information Processing Systems*. 2014, pp. 3338–3346.

[5] Justin Johnson, Mike Roberts, and Matt Fisher. "Learning to Play 2D Video Games". In: ().

[6] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

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

[8] Yavar Naddaf et al. *Game-independent ai agents for playing atari 2600 console games*. University of Alberta, 2010.

[9] Claude E Shannon. *Programming a computer for playing chess*. Springer, 1988.

[10] Gerald Tesauro. "Temporal difference learning and TD-Gammon". In: *Communications of the ACM* 38.3 (1995), pp. 58–68.