Learning to Play Atari Games

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

Abstract—**Teaching computers to play video games is a complex learning problem that has recently seen increased attention. In this paper, we develop a system that, using constant model and hyperparameter settings, learns to play a variety of Atari games. In order to accomplish this task, we extract object features from the game screen, and provide these features as input into reinforcement learning algorithms. We detail our testing of different hyperparameter settings for two reinforcement learning algorithms, using both linear and neural network function approximators, in order to compare the performance of these approaches. We also consider learning techniques such as the use of replay memory with Q-learning, finding that even on simple MDPs this method significantly improves performance. Finally, we evaluate our model, selected through validation on the game Breakout, on a test set of different video games.**

I. Introduction

Video games pose a challenging sequential decision making problem for computers. Video games have a straight-forward visual interface, with a large state-space. They 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. In this project, we consider this problem on the Atari 2600 platform, which provides a set of relatively simple video games. We attempt to lower the complexity of the problem by applying intuition into the human solution to video games. By detecting objects on the screen, we allow the learning algorithm to operate in a reduced state-space. We then experiment with multiple reinforcement learning algorithms and function approximators in order to determine the algorithm and hyperparameter settings that could make best use of these features. The only inputs into our learning system are a 192x160 grid of RGB pixels representing the game screen and the current score of the game. This screen is then processed into classified objects by a combination of computer vision techniques and the DBSCAN algorithm. These objects are then fed into one of two learning agents, either Q-Learning with replay memory or SARSA(λ), which output an action according to an ϵ-greedy policy, which is then used to advance the game to the next state. We evaluate the model selected during validation on a test set of Atari games to assess the models ability to generalize. This report details the results of these experiments, and the modifications to these algorithms that increase their performance.

We also used this project for CS221. 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 as well as the reinforcement learning implementations themselves, namely development of both the linear and neural network-based SARSA(λ) and Q-learning models.

II. Related 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 [11]. 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 [12]. Development of a general video game playing agent specifically for the Atari 2600 was introduced in 2006 by Naddaf [9]. 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 [5]; 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. [4] 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. [8][7] 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.

III. Dataset and Features

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, preprocessing these pixels provides 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.

A. The Game State

We extract the game screen at each frame of the game using the Arcade Learning Environment (ALE) [1], an open source emulation toolkit with a machine learning interface. This toolkit provides both the raw pixel data of the screen and the reward associated with each state. The screen is output in the form of a 192x160 grid of RGB pixels. The reward function is based upon the ”score” of a game, and is extracted from a games RAM contents.

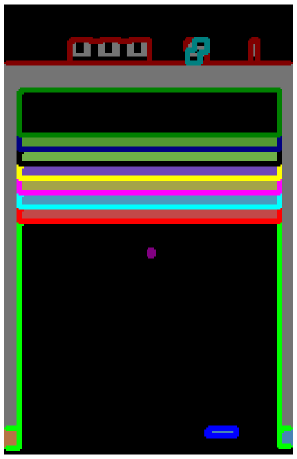

Fig. 1. Output of DBSCAN clustering, where each outline color represents a unique label.

B. 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 through OpenCV [3], 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.

C. Feature Clustering

We use 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 [10], 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.

D. 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.

E. Feature Output

The extracted and classified object features are processed into a format easily usable by a linear function approximator. First, we reduce the state-space of the screen into a smaller number of tiles in order to increase the learning speed of the algorithm. We also reduce the derivatives from a numerical derivative to a boolean indicating the direction of motion in the x and y directions. We additionally include ”cross features” that include the relative positions and derivatives between each object pair, as the linear function approximator is not able evaluate object interactions without cross features.

IV. METHODS

A. Reinforcement Learning Algorithms

Our review of past research [4] indicated that model-based reinforcement learning algorithms would likely not achieve high performance. For example, [6] 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 and only linear approximators for the second.

B. Q-Learning with Replay Memory

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

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

With the following parameter update equation:

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

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 more sophisticated replacement policy.

C. 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 eligibile to be assigned credit for a given reward encountered by the agent. Each τ value is updated per timestep 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:

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

D. Function Approximation

1) Linear Approximation: For linear function approximation, the models are parameterized with one value for each feature. The output of the value approximation is then a linear combination of the present features multiplied by their corresponding values. The parameter updates for each model are provided above in the sections for the corresponding algorithm.

2) Feed-Forward Neural Network: 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. [7] 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 assume a constant ordering of the input objects, ensured using the extracted class and within-class ids of each object.

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 outputs of other actions are not considered because they do not contribute to the loss value.

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

E. Training Methods

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 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. We evaluate these claims on a simple test MDP in section 5.

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.

V. Experiments and Results

A. Replay Memory

Prior to training the different algorithms on Atari games, we compared their performance on a simple test Markov Decision Process (MDP), as well as evaluated the impact of a replay memory and static target function for the case of Q-learing specifically. We used a simple grid MDP with positive and negative rewards on the edges. We also incorporated a trigger location that when encountered by the agent prior to reaching an edge would significantly increase the final reward with the hope that this would approximate the delayed rewards present in many Atari games.

We first compared performance using 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, and the run that achieved the maximum reward from its group was selected for visualization Figures 2 and 3.

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 exhibit 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.

These findings align with the assertions of Minh et al. that a replay memory helps improve learning. They further assert that this is the case because the replay memory eliminates correlations between consecutive updates thereby making each

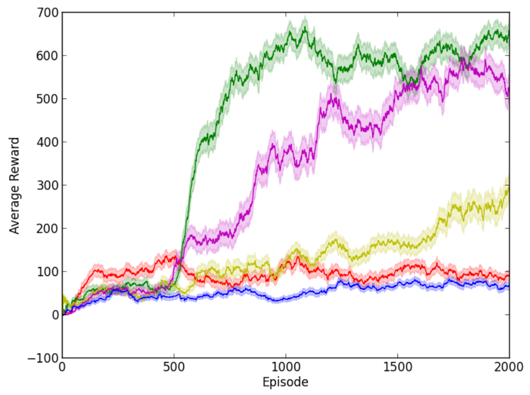

Fig. 2. Effect of replay memory sample size on average reward with constant number of trials. The 1000-capacity, 20-sample memory shown in green provides the best performance, but also executes the most updates.

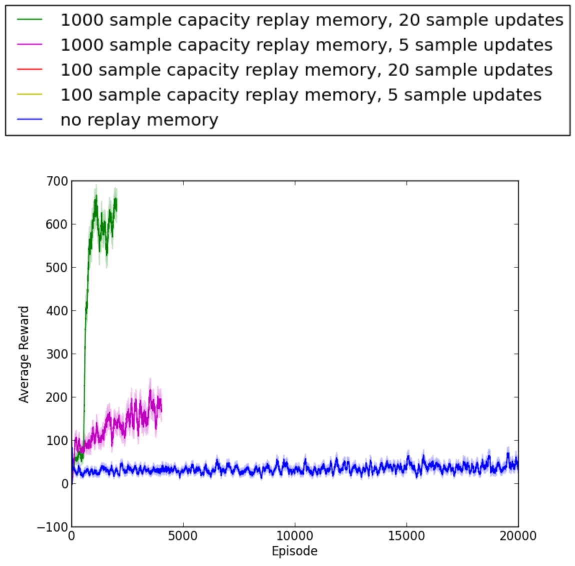

Fig. 3. Effect of replay memory sample size on average reward with constant number of updates. The 1000-capacity, 20-sample memory shown in green again provides the best performance even though it performs the same number of updates as the other two memories shown.

update more informative. To evaluate this claim, we collected both the samples used in making parameter updates and those encountered sequentially during the execution of a Q-learing model (i.e., those that would be used to update parameters when not using a replay memory) with a 10000-capacity replay memory on the Atari game Breakout. We then performed Principal Component Analysis (PCA) over both sets in fifty-item batches and compared the fraction of the variance explained by the first principal component for each set. Averaging over 10,000 episodes, we found that the first principal component of the sequential updates could explain 44.0% of the variance in the fifty-item batches, whereas the first principal component of the sampled updates could explain only 26.2% of the variance in the fifty-item batches. These values indicate that the replay memory samples are less correlated than those used in sequential updates.

B. Model Validation on Breakout

We use the Atari game Breakout as the validation game on which we would select the best hyperparameters for the model. Breakout was selected for its relatively simple gameplay, delayed rewards, and demonstrated high performance in related work [7].

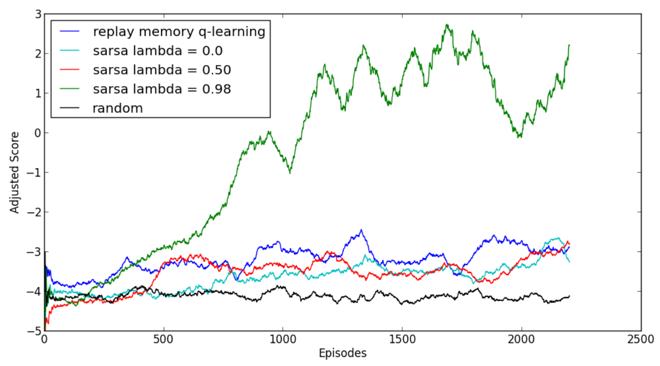

Fig. 4. Learning curves for different leraning models playing breakout.

We have tested three different values of λ in SARSA(λ), as well as Q-Learning with replay memory.

The learning curves displayed in Figure 4 clearly show that SARSA(λ) learning with $\lambda = 0.98$ has the best performance on breakout. SARSA(λ) is a well suited model for breakout, as rewards for breakout are heavily delayed from the actions that create the rewards. A high value of λ successfully assigns credit to actions that preceded rewards.

One can gain insight into the learned solution by looking at the weights associated with different feature combinations. Figure 5 shows a visualization of these weights in the form of a heatmap. Based upon these weights the ball being in the same x location as the paddle is desirable, as well as the ball being higher on the screen. This closely corresponds to human intuition for the game breakout.

C. Model Evaluation on General Atari Games

In order to test our hyper-parameter turning, we conducted learning experiments on three other Atari games: Asterix, Chopper Command, and Space Invaders. The learning performance of these games can be seen in Figure 6.

The learning algorithm did not perform as well on the test set as it did when training on Breakout. This is most likely caused by the increased complexity of the test set, as every game in the test set has a larger state and action space than Breakout. There is evidence of learning only in the game chopper command, which has easy to extract features and fairly simple interactions between features. Asterix has a very busy screen state, which makes feature extraction difficult and this causes slow and suboptimal learning. Space invaders is a more complex game than breakout, and the resulting state-space is far harder to explore fully.

D. Neural Network Function Evaluation

Despite numerous experiments with different network structures and hyperparameter settings, we were largely unable to achieve above random results with the neural network function approximator. We significantly departed in setup from the extant literature, and this made it more difficult to troubleshoot the issues we encountered. The opaque nature of neural network hidden layer features also made it difficult to assess whether any progress was or could be made in learning directly from the extracted object features.

[2]Heatmap images generated by Rush Moody

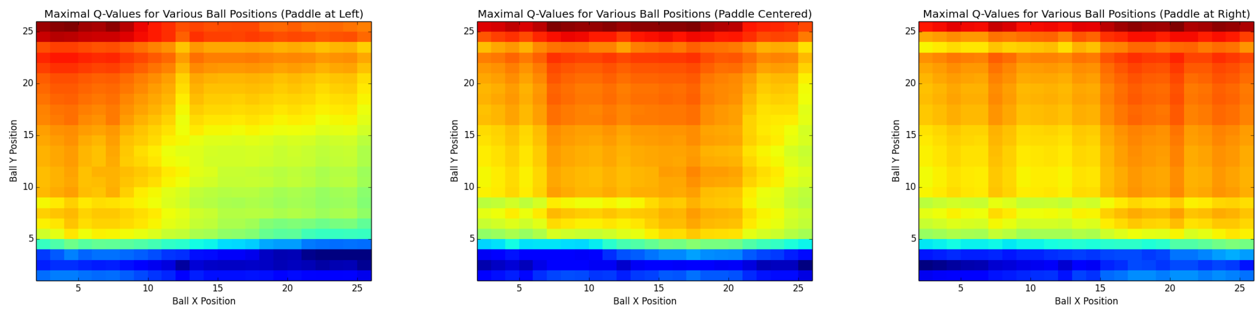

Fig. 5. Heatmap showing optimal Q value attainable for each position of the ball holding position of the paddle fixed. Dark red is associated with the largest Q values and dark blue with the lowest. The assumed paddle position in the above image is left, center, right respectively[2].

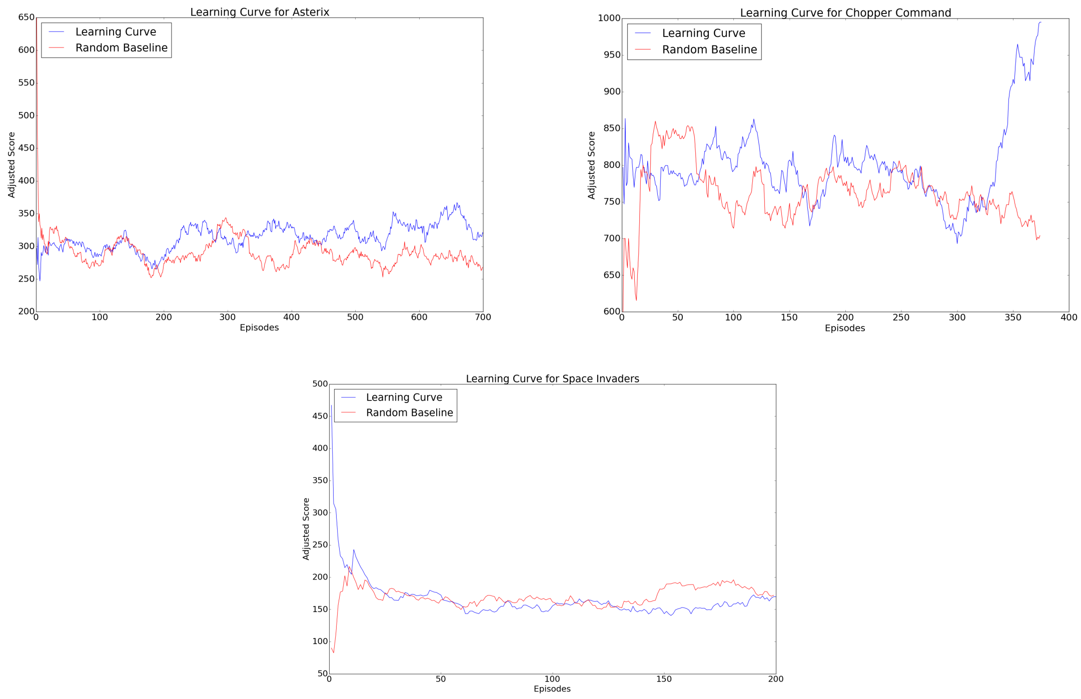

Fig. 6. Learning curves for Asterix, Chopper Command, and Space Invaders.

VI. 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] Gary Bradski et al. “The opencv library”. In: *Doctor Dobbs Journal* 25.11 (2000), pp. 120–126.

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

[5] 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.

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

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

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

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

[10] Fabian Pedregosa et al. “Scikit-learn: Machine learning in Python”. In: *The Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

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

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