Journal of Artificial Intelligence Research 47 (2013) 253–279 Submitted 02/13; published 06/13

The Arcade Learning Environment: An Evaluation Platform for General Agents

Marc G. Bellemare MG17@CS.UALBERTA.CA
University of Alberta, Edmonton, Alberta, Canada

Yavar Naddaf YAVAR@EMPIRICALRESULTS.CA
Empirical Results Inc., Vancouver, British Columbia, Canada

Joel Veneness VENESS@CS.UALBERTA.CA
Michael Bowling BOWLING@CS.UALBERTA.CA
University of Alberta, Edmonton, Alberta, Canada

Abstract

In this article we introduce the Arcade Learning Environment (ALE): both a challenge problem and a platform and methodology for evaluating the development of general, domain-independent AI technology. ALE provides an interface to hundreds of Atari 2600 game environments, each one different, interesting, and designed to be a challenge for human players. ALE presents significant research challenges for reinforcement learning, model learning, model-based planning, imitation learning, transfer learning, and intrinsic motivation. Most importantly, it provides a rigorous testbed for evaluating and comparing approaches to these problems. We illustrate the promise of ALE by developing and benchmarking domain-independent agents designed using well-established AI techniques for both reinforcement learning and planning. In doing so, we also propose an evaluation methodology made possible by ALE, reporting empirical results on over 55 different games. All of the software, including the benchmark agents, is publicly available.

1. Introduction

A longstanding goal of artificial intelligence is the development of algorithms capable of general competency in a variety of tasks and domains without the need for domain-specific tailoring. To this end, different theoretical frameworks have been proposed to formalize the notion of "big" artificial intelligence (e.g., Russell, 1997; Hutter, 2005; Legg, 2008). Similar ideas have been developed around the theme of *lifelong learning*: learning a reusable, high-level understanding of the world from raw sensory data (Thrun & Mitchell, 1995; Pierce & Kuipers, 1997; Stober & Kuipers, 2008; Sutton et al., 2011). The growing interest in competitions such as the General Game Playing competition (Genesereth, Love, & Pell, 2005), Reinforcement Learning competition (Whiteson, Tanner, & White, 2010), and the International Planning competition (Coles et al., 2012) also suggests the artificial intelligence community's desire for the emergence of algorithms that provide general competency.

Designing generally competent agents raises the question of how to best evaluate them. Empirically evaluating general competency on a handful of parametrized benchmark problems is, by definition, flawed. Such an evaluation is prone to method overfitting (Whiteson, Tanner, Taylor, & Stone, 2011) and discounts the amount of expert effort necessary to transfer the algorithm to new domains. Ideally, the algorithm should be compared across domains that are (i) *varied* enough to claim generality, (ii) each *interesting* enough to be representative of settings that might be faced in practice, and (iii) each created by an *independent* party to be free of experimenter's bias.

In this article, we introduce the Arcade Learning Environment (ALE): a new challenge problem, platform, and experimental methodology for empirically assessing agents designed for general competency. ALE is a software framework for interfacing with emulated Atari 2600 game environments. The Atari 2600, a second generation game console, was originally released in 1977 and remained massively popular for over a decade. Over 500 games were developed for the Atari 2600, spanning a diverse range of genres such as shooters, beat'em ups, puzzle, sports, and action-adventure games; many game genres were pioneered on the console. While modern game consoles involve visuals, controls, and a general complexity that rivals the real world, Atari 2600 games are far simpler. In spite of this, they still pose a variety of challenging and interesting situations for human players.

ALE is both an experimental methodology and a challenge problem for general AI competency. In machine learning, it is considered poor experimental practice to both train and evaluate an algorithm on the same data set, as it can grossly over-estimate the algorithm's performance. The typical practice is instead to train on a *training set* then evaluate on a disjoint *test set*. With the large number of available games in ALE, we propose that a similar methodology can be used to the same effect: an approach's domain representation and parametrization should be first tuned on a small number of *training games*, before testing the approach on unseen *testing games*. Ideally, agents designed in this fashion are evaluated on the testing games only once, with no possibility for subsequent modifications to the algorithm. While general competency remains the long-term goal for artificial intelligence, ALE proposes an achievable stepping stone: techniques for general competency across the gamut of Atari 2600 games. We believe this represents a goal that is attainable in a short time-frame yet formidable enough to require new technological breakthroughs.

2. Arcade Learning Environment

We begin by describing our main contribution, the Arcade Learning Environment (ALE). ALE is a software framework designed to make it easy to develop agents that play arbitrary Atari 2600 games.

2.1 The Atari 2600

The Atari 2600 is a home video game console developed in 1977 and sold for over a decade (Montfort & Bogost, 2009). It popularized the use of general purpose CPUs in game console hardware, with game code distributed through cartridges. Over 500 original games were released for the console; "homebrew" games continue to be developed today, over thirty years later. The console's joystick, as well as some of the original games such as Adventure and Pitfall!, are iconic symbols of early video games. Nearly all arcade games of the time – Pac-Man and Space Invaders are two well-known examples – were ported to the console.

Despite the number and variety of games developed for the Atari 2600, the hardware is relatively simple. It has a 1.19Mhz CPU and can be emulated much faster than real-time on modern hardware. The cartridge ROM (typically 2–4kB) holds the game code, while the console RAM itself only holds 128 bytes (1024 bits). A single game screen is 160 pixels wide

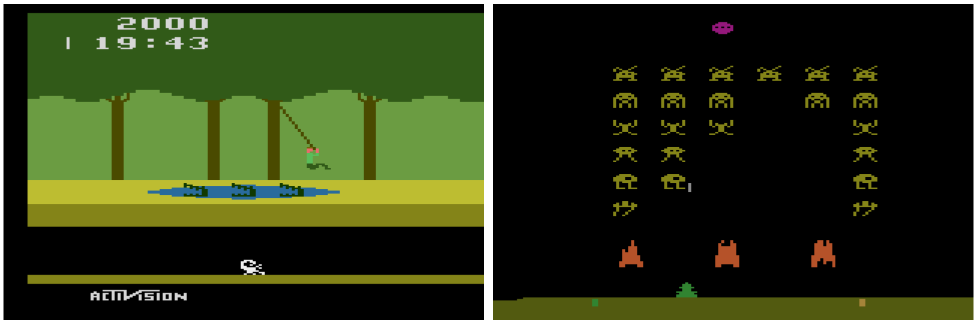

Figure 1: Screenshots of Pitfall! and Space Invaders.

and 210 pixels high, with a 128-colour palette; 18 "actions" can be input to the game via a digital joystick: three positions of the joystick for each axis, plus a single button. The Atari 2600 hardware limits the possible complexity of games, which we believe strikes the perfect balance: a challenging platform offering conceivable near-term advancements in learning, modelling, and planning.

2.2 Interface

ALE is built on top of Stella[1], an open-source Atari 2600 emulator. It allows the user to interface with the Atari 2600 by receiving joystick motions, sending screen and/or RAM information, and emulating the platform. ALE also provides a game-handling layer which transforms each game into a standard reinforcement learning problem by identifying the accumulated score and whether the game has ended. By default, each observation consists of a single game screen (frame): a 2D array of 7-bit pixels, 160 pixels wide by 210 pixels high. The action space consists of the 18 discrete actions defined by the joystick controller. The game-handling layer also specifies the minimal set of actions needed to play a particular game, although none of the results in this paper make use of this information. When running in real-time, the simulator generates 60 frames per second, and at full speed emulates up to 6000 frames per second. The reward at each time-step is defined on a game by game basis, typically by taking the difference in score or points between frames. An episode begins on the first frame after a reset command is issued, and terminates when the game ends. The game-handling layer also offers the ability to end the episode after a predefined number of frames[2]. The user therefore has access to several dozen games through a single common interface, and adding support for new games is relatively straightforward.

ALE further provides the functionality to save and restore the state of the emulator. When issued a *save-state* command, ALE saves all the relevant data about the current game, including the contents of the RAM, registers, and address counters. The *restore-state* command similarly resets the game to a previously saved state. This allows the use of ALE as a generative model to study topics such as planning and model-based reinforcement learning.

1. `http://stella.sourceforge.net/`
2. This functionality is needed for a small number of games to ensure that they always terminate. This prevents situations such as in Tennis, where a degenerate agent could choose to play indefinitely by refusing to serve.

2.3 Source Code

ALE is released as free, open-source software under the terms of the GNU General Public License. The latest version of the source code is publicly available at:

`http://arcadelearningenvironment.org`

The source code for the agents used in the benchmark experiments below is also available on the publication page for this article on the same website. While ALE itself is written in C++, a variety of interfaces are available that allow users to interact with ALE in the programming language of their choice. Support for new games is easily added by implementing a derived class representing the game's particular reward and termination functions.

3. Benchmark Results

Planning and reinforcement learning are two different AI problem formulations that can naturally be investigated within the ALE framework. Our purpose in presenting benchmark results for both of these formulations is two-fold. First, these results provide a baseline performance for traditional techniques, establishing a point of comparison with future, more advanced, approaches. Second, in describing these results we illustrate our proposed methodology for doing empirical validation with ALE.

3.1 Reinforcement Learning

We begin by providing benchmark results using SARSA(λ), a traditional technique for model-free reinforcement learning. Note that in the reinforcement learning setting, the agent does not have access to a model of the game dynamics. At each time step, the agent selects an action and receives a reward and an observation, and the agent's aim is to maximize its accumulated reward. In these experiments, we augmented the SARSA(λ) algorithm with linear function approximation, replacing traces, and ϵ-greedy exploration. A detailed explanation of SARSA(λ) and its extensions can be found in the work of Sutton and Barto (1998).

3.1.1 Feature Construction

In our approach to the reinforcement learning setting, the most important design issue is the choice of features to use with linear function approximation. We ran experiments using five different sets of features, which we now briefly explain; a complete description of these feature sets is given in Appendix A. Of these sets of features, BASS, DISCO and RAM were originally introduced by Naddaf (2010), while the rest are novel.

Basic. The *Basic* method, derived from Naddaf's BASS (2010), encodes the presence of colours on the Atari 2600 screen. The Basic method first removes the image background by storing the frequency of colours at each pixel location within a histogram. Each game background is precomputed offline, using 18,000 observations collected from sample trajectories. The sample trajectories are generated by following a human-provided trajectory for a random number of steps and subsequently selecting actions uniformly at random. The screen is then divided into 16×14 tiles. Basic generates one binary feature for each of the 128 colours and each of the tiles, giving a total of 28,672 features.

BASS. The *BASS* method behaves identically to the Basic method save in two respects. First, BASS augments the Basic feature set with pairwise combinations of its features. Second, BASS uses a smaller, 8-colour encoding to ensure that the number of pairwise combinations remains tractable.

DISCO. The *DISCO* method aims to detect objects within the Atari 2600 screen. To do so, it first preprocesses 36,000 observations from sample trajectories generated as in the *Basic* method. DISCO also performs the background subtraction steps as in Basic and BASS. Extracted objects are then labelled into classes. During the actual training, DISCO infers the class label of detected objects and encodes their position and velocity using tile coding (Sutton & Barto, 1998).

LSH. The *LSH* method maps raw Atari 2600 screens into a small set of binary features using Locally Sensitive Hashing (Gionis, Indyk, & Motwani, 1999). The screens are mapped using random projections, such that visually similar screens are more likely to generate the same features.

RAM. The *RAM* method works on an entirely different observation space than the other four methods. Rather than receiving in Atari 2600 screen as an observation, it directly observes the Atari 2600's 1024 bits of memory. Each bit of RAM is provided as a binary feature together with the pairwise logical-AND of every pair of bits.

3.1.2 Evaluation Methodology

We first constructed two sets of games, one for training and the other for testing. We used the training games for parameter tuning as well as design refinements, and the testing games for the final evaluation of our methods. Our training set consisted of five games: Asterix, Beam Rider, Freeway, Seaquest and Space Invaders. The parameter search involved finding suitable values for the parameters to the SARSA(λ) algorithm, i.e. the learning rate, exploration rate, discount factor, and the decay rate λ. We also searched the space of feature generation parameters, for example the abstraction level for the BASS agent. The results of our parameter search are summarized in Appendix C. Our testing set was constructed by choosing semi-randomly from the 381 games listed on Wikipedia[3] at the time of writing. Of these games, 123 games have their own Wikipedia page, have a single player mode, are not adult-themed or prototypes, and can be emulated in ALE. From this list, 50 games were chosen at random to form the test set.

Evaluation of each method on each game was performed as follows. An *episode* starts on the frame that follows the reset command, and terminates when the end-of-game condition is detected or after 5 minutes of real-time play (18,000 frames), whichever comes first. During an episode, the agent acts every 5 frames, or equivalently 12 times per second of gameplay. A reinforcement learning *trial* consists of 5,000 training episodes, followed by 500 evaluation episodes during which no learning takes place. The agent's performance is

3. http://en.wikipedia.org/wiki/List_of_Atari_2600_games (July 12, 2012)

Game	Basic	BASS	DISCO	LSH	RAM	Random	Const	Perturb	Human
Asterix	862	860	755	**987**	943	288	650	338	620
Seaquest	579	**665**	422	509	594	108	160	451	156
Boxing	-3	16	12	10	**44**	-1	-25	-10	-2
H.E.R.O.	6053	**6459**	2720	3836	3281	712	0	148	6087
Zaxxon	1392	2069	70	**3365**	304	0	0	2	820

Table 1: Reinforcement Learning results for selected games. Asterix and Seaquest are part of the training set.

measured as the average score achieved during the evaluation episodes. For each game, we report our methods' average performance across 30 trials.

For purposes of comparison, we also provide performance measures for three simple baseline agents – *Random*, *Const* and *Perturb* – as well as the performance of a non-expert human player. The Random agent picks a random action on every frame. The Const agent selects a single fixed action throughout an episode; our results reflect the highest score achieved by any single action within each game. The Perturb agent selects a fixed action with probability 0.95 and otherwise acts uniformly randomly; for each game, we report the performance of the best policy of this type. Additionally, we provide human player results that report the five-episode average score obtained by a beginner (who had never previously played Atari 2600 games) playing selected games. Our aim is not to provide exhaustive or accurate human-level benchmarks, which would be beyond the scope of this paper, but rather to offer insight into the performance level achieved by our agents.

3.1.3 Results

A complete report of our reinforcement learning results is given in Appendix D. Table 1 shows a small subset of results from two training games and three test games. In 40 games out of 55, learning agents perform better than the baseline agents. In some games, e.g., Double Dunk, Journey Escape and Tennis, the no-action baseline policy performs the best by essentially refusing to play and thus incurring no negative reward. Within the 40 games for which learning occurs, the BASS method generally performs best. DISCO performed particularly poorly compared to the other learning methods. The RAM-based agent, surprisingly, did not outperform image-based methods, despite building its representation from raw game *state*. It appears the screen image carries structural information that is not easily extracted from the RAM bits.

Our reinforcement learning results show that while some learning progress is already possible in Atari 2600 games, much more work remains to be done. Different methods perform well on different games, and no single method performs well on all games. Some games are particularly challenging. For example, platformers such as Montezuma's Revenge seem to require high-level planning far beyond what our current, domain-independent methods provide. Tennis requires fairly elaborate behaviour before observing any positive reward, but simple behaviour can avoid negative rewards. Our results also highlight the value of ALE as an experimental methodology. For example, the DISCO approach performs reasonably well on the training set, but suffers a dramatic reduction in performance when applied to unseen games. This suggests the method is less robust than the other methods we studied. After a quick glance at the full table of results in Appendix D, it is clear that summarizing results across such varied domains needs further attention; we explore this issue further in Section 4.

3.2 Planning

The Arcade Learning Environment can naturally be used to study planning techniques by using the emulator itself as a generative model. Initially it may seem that allowing the agent to plan into the future with a perfect model trivializes the problem. However, this is not the case: the size of state space in Atari 2600 games prohibits exhaustive search. Eighteen different actions are available at every frame; at 60 frames per second, looking ahead one second requires $18^{60} \approx 10^{75}$ simulation steps. Furthermore, rewards are often sparsely distributed, which causes significant horizon effects in many search algorithms.

3.2.1 Search Methods

We now provide benchmark ALE results for two traditional search methods. Each method was applied online to select an action at every time step (every five frames) until the game was over.

Breadth-first Search. Our first approach builds a search tree in a breadth-first fashion until a node limit is reached. Once the tree is expanded, node values are updated recursively from the bottom of the tree to the root. The agent then selects the action corresponding to the branch with the highest discounted sum of rewards. Expanding the full search tree requires a large number of simulation steps. For instance, selecting an action every 5 frames and allowing a maximum of 100,000 simulation steps per frame, the agent can only look ahead about a third of a second. In many games, this allows the agent to collect immediate rewards and avoid death but little else. For example, in Seaquest the agent must collect a swimmer and return to the surface before running out of air, which involves planning far beyond one second.

UCT: Upper Confidence Bounds Applied to Trees. A preferable alternative to exhaustively expanding the tree is to simulate deeper into the more promising branches. To do this, we need to find a balance between expanding the higher-valued branches and spending simulation steps on the lower-valued branches to get a better estimate of their values. The UCT algorithm, developed by Kocsis and Szepesvári (2006), deals with the exploration-exploitation dilemma by treating each node of a search tree as a multi-armed bandit problem. UCT uses a variation of UCB1, a bandit algorithm, to choose which child node to visit next. A common practice is to apply a t-step random simulation at the end of each leaf node to obtain an estimate from a longer trajectory. By expanding the more valuable branches of the tree and carrying out a random simulation at the leaf nodes, UCT is known to perform well in many different settings (Browne et al., 2012).

Our UCT implementation was entirely standard, except for one optimization. Few Atari games actually distinguish between all 18 actions at every time step. In Beam Rider, for example, the down action does nothing, and pressing the button when a bullet has already

Game	Full Tree	UCT	Best Learner	Best Baseline
Asterix	2136	**290700**	987	650
Seaquest	288	**5132**	665	451
Boxing	**100**	100	44	-1
H.E.R.O.	1324	**12860**	6459	712
Zaxxon	0	**22610**	3365	2

Table 2: Results for selected games. Asterix and Seaquest are part of the training set.

been shot has no effect. We exploit this fact as follows: after expanding the children of a node in the search tree, we compare the resulting emulator states. Actions that result in the same state are treated as duplicates and only one of the actions is considered in the search tree. This reduces the branching factor, thus allowing deeper search. At every step, we also reuse the part of our search tree corresponding to the selected action. Pseudocode for our implementation of the UCT algorithm is given in Appendix B.

3.2.2 Experimental Setup

We designed and tuned our algorithms based on the same five training games used in Section 3.1, and subsequently evaluated the methods on the fifty games of the testing set. The training games were used to determine the length of the search horizon as well as the constant controlling the amount of exploration at internal nodes of the tree. Each episode was set to last up to 5 minutes of real-time play (18,000 frames), with actions selected every 5 frames, matching our settings in Section 3.1.2. On average, each action selection step took on the order of 15 seconds. We also used the same discount factor as in Section 3.1. We ran our algorithms for 10 episodes per game. Details of the algorithmic parameters can be found in Appendix C.

3.2.3 Results

A complete report of our search results is given in Appendix D. Table 2 shows results on a selected subset of games. For reference purposes, we also include the performance of the best learning agent and the best baseline policy from Table 1. Together, our two search methods performed better than both learning agents and the baseline policies on 49 of 55 games. In most cases, UCT performs significantly better than breadth-first search. Four of the six games for which search methods do not perform best are games where rewards are sparse and require long-term planning. These are Freeway, Private Eye, Montezuma's Revenge and Venture.

4. Evaluation Metrics for General Atari 2600 Agents

Applying algorithms to a large set of games as we did in Sections 3.1 and 3.2 presents difficulties when interpreting the results. While the agent's goal in all games is to maximize its score, scores for two different games cannot be easily compared. Each game uses its own scale for scores, and different game mechanics make some games harder to learn than others. The challenges associated with comparing general agents has been previously highlighted

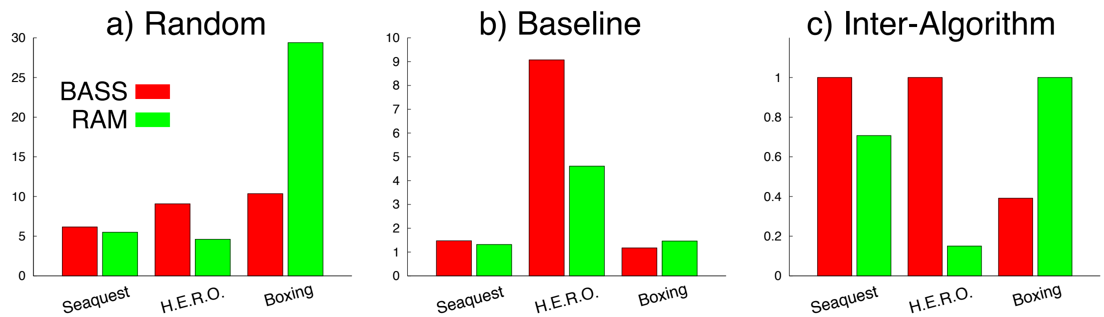

Figure 2: Left to right: random-normalized, baseline and inter-algorithm scores.

by Whiteson et al. (2011). Although we can always report full performance tables, as we did in Appendix D, some more compact summary statistics are also desirable. We now introduce some simple metrics that help compare agents across a diverse set of domains, such as our test set of Atari 2600 games.

4.1 Normalized Scores

Consider the scores $s_{g,1}$ and $s_{g,2}$ achieved by two algorithms in game g. Our goal here is to explore methods that allow us to compare two sets of scores $S_1 = \{s_{g_1,1}, \dots, s_{g_n,1}\}$ and $S_2 = \{s_{g_1,2}, \dots, s_{g_n,2}\}$. The approach we take is to transform $s_{g,i}$ into a normalized score $z_{g,i}$ with the aim of comparing normalized scores across games; in the ideal case, $z_{g,i} = z_{g',i}$ implies that algorithm i performs as well on game g as on game g'. In order to compare algorithms over a set of games, we aggregate normalized scores for each game and each algorithm.

The most natural way to compare games with different scoring scales is to normalize scores so that the numerical values become comparable. All of our normalization methods are defined using the notion of a *score range* $[r_{g,\min}, r_{g,\max}]$ computed for each game. Given such a score range, score $s_{g,i}$ is normalized by computing $z_{g,i} := (s_{g,i} - r_{g,\min}) \;/\; (r_{g,\max} - r_{g,\min})$.

4.1.1 Normalization to a Reference Score

One straightforward method is to normalize to a score range defined by repeated runs of a random agent across each game. Here, $r_{g,\max}$ is the absolute value of the average score achieved by the random agent, and $r_{g,\min} = 0$. Figure 2a depicts the *random-normalized scores* achieved by BASS and RAM on three games. Two issues arise with this approach: the scale of normalized scores may be excessively large and normalized scores are generally not translation invariant. The issue of scale is best seen in a game such as Freeway, for which the random agent achieves a score close to 0: scores achieved by learning agents, in the 10-20 range, are normalized into thousands. By contrast, no learning agent achieves a random-normalized score greater than 1 in Asteroids.

4.1.2 Normalizing to a Baseline Set

Rather than normalizing to a single reference we may normalize to the score range implied by a set of references. Let $b_{g,1}, \ldots, b_{g,k}$ be a set of reference scores. A method's *baseline score* is computed using the score range $[\min_{i\in\{1,\ldots,k\}} b_{g,i}, \max_{i\in\{1,\ldots,k\}} b_{g,i}]$.

Given a sufficiently rich set of reference scores, baseline normalization allows us to reduce the scores for most games to comparable quantities, and lets us know whether meaningful performance was obtained. Figure 2b shows example baseline scores. The score range for these scores corresponds to the scores achieved by 37 baseline agents (Section 3.1.2): *Random*, *Const* (one policy per action), and *Perturb* (one policy per action).

A natural idea is to also include scores achieved by human players into the baseline set. For example, one may include the score achieved by an expert as well as the score achieved by a beginner. However, using human scores raises its own set of issues. For example, humans often play games without seeking to maximize score; humans also benefit from prior knowledge that is difficult to incorporate into domain-independent agents.

4.1.3 Inter-Algorithm Normalization

A third alternative is to normalize using the scores achieved by the algorithms themselves. Given n algorithms, each achieving score $s_{g,i}$ on game g, we define the *inter-algorithm score* using the score range $[\min_{i\in\{1,\ldots,n\}} s_{g,i}, \max_{i\in\{1,\ldots,n\}} s_{g,i}]$. By definition, $z_{g,i} \in [0, 1]$. A special case of this is when n=2, where $z_{g,i} \in \{0, 1\}$ indicates which algorithm is better than the other. Figure 2c shows example inter-algorithm scores; the relevant score ranges are constructed from the performance of all five learning agents.

Because inter-algorithm scores are bounded, this type of normalization is an appealing solution to compare the relative performance of different methods. Its main drawback is that it gives no indication of the objective performance of the best algorithm. A good example of this is Venture: the inter-algorithm score of 1.0 achieved by BASS does not reflect the fact that none of our agents achieved a score remotely comparable to a human's performance. The lack of objective reference in inter-algorithm normalization suggests that it should be used to complement other scoring metrics.

4.2 Aggregating Scores

Once normalized scores are obtained for each game, the next step is to produce a measure that reflects how well each agent performs across the set of games. As illustrated by Table 4, a large table of numbers does not easily permit comparison between algorithms. We now describe three methods to aggregate normalized scores.

4.2.1 Average Score

The most straightforward method of aggregating normalized scores is to compute their average. Without perfect score normalization, however, score averages tend to be heavily influenced by games such as Zaxxon for which baseline scores are high. Averaging inter-algorithm scores obviates this issue as all scores are bounded between 0 and 1. Figure 3 displays average baseline and inter-algorithm scores for our learning agents.

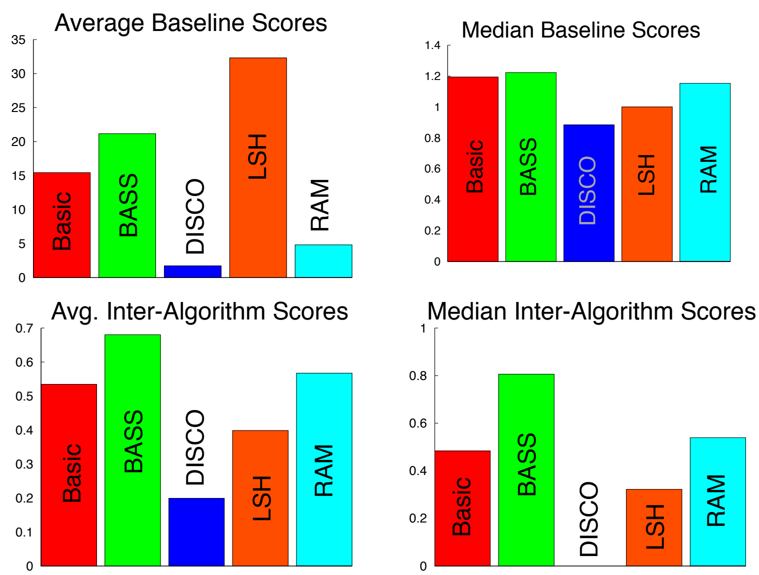

Figure 3: Aggregate normalized scores for the five reinforcement learning agents.

4.2.2 Median Score

Median scores are generally more robust to outliers than average scores. The median is obtained by sorting all normalized scores and selecting the middle element (the average of the two middle elements is used if the number of scores is even). Figure 3 shows median baseline and inter-algorithm scores for our learning agents. Comparing medians and averages in the baseline score (upper two graphs) illustrates exactly the outlier sensitivity of the average score, where the LSH method appears dramatically superior due entirely to its performance in Zaxxon.

4.2.3 Score Distribution

The *score distribution* aggregate is a natural generalization of the median score: it shows the fraction of games on which an algorithm achieves a certain normalized score or better. It is essentially a quantile plot or inverse empirical CDF. Unlike the average and median scores, the score distribution accurately represents the performance of an agent irrespective of how individual scores are distributed. Figure 4 shows baseline and inter-algorithm score distributions. Score distributions allow us to compare different algorithms at a glance – if one curve is above another, the corresponding method generally obtains higher scores.

Using the baseline score distribution, we can easily determine the proportion of games for which methods perform better than the baseline policies (scores above 1). The inter-algorithm score distribution, on the other hand, effectively conveys the relative performance of each method. In particular, it allows us to conclude that BASS performs slightly better than Basic and RAM, and that DISCO performs significantly worse than the other methods.

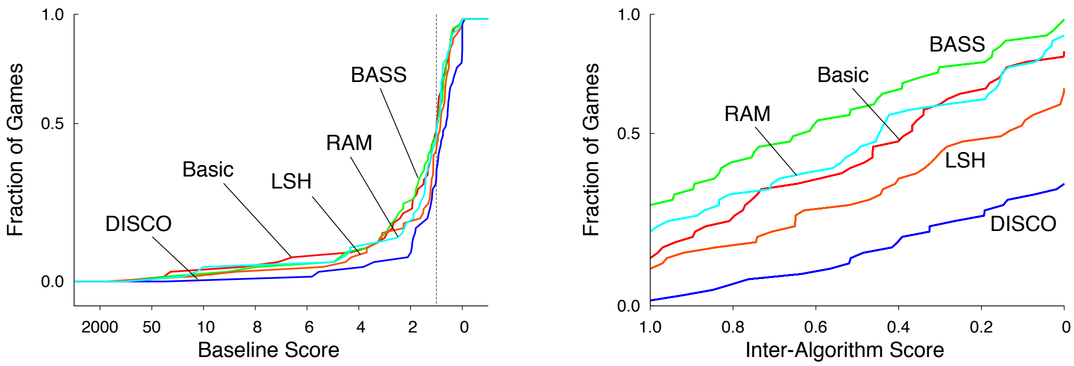

Figure 4: Score distribution over all games.

	Basic	BASS	DISCO	LSH	RAM
Basic	—	18–32	39–13	34–18	22–25
BASS	32–18	—	48–5	36–17	29–20
DISCO	13–39	5–48	—	17–33	9–41
LSH	18–34	17–36	33–17	—	15–36
RAM	25–22	20–29	41–9	36–15	—

Table 3: Paired tests over all games. Each entry shows the number of games for which the performance of the first algorithm (left) is better (–worse) than the second algorithm's.

4.3 Paired Tests

An alternate evaluation metric, especially useful when comparing only a few algorithms, is to perform paired tests over the raw scores. For each game, we performed a two-tailed Welsh's t-test with 99% confidence intervals to determine whether one algorithm's score was statistically different than the other's. Table 3 provides, for each pair of algorithms, the number of games for which one algorithm performs statistically better or worse than the other. Because of their ternary nature, paired tests tend to magnify small but significant differences in scores.

5. Related Work

We now briefly survey recent research related to Atari 2600 games and some prior work on the construction of empirical benchmarks for measuring general competency.

5.1 Atari Games

There has been some attention devoted to Atari 2600 game playing within the reinforcement learning community. For the most part, prior work has focused on the challenge of finding good state features for this domain. Diuk, Cohen, and Littman (2008) applied their DOORMAX algorithm to a restricted version of the game of PITFALL!. Their method extracts objects from the displayed image with game-specific object detection. These objects are then converted into a first-order logic representation of the world, the Object-Oriented

Markov Decision Process (OO-MDP). Their results show that DOORMAX can discover the optimal behaviour for this OO-MDP within one episode. Wintermute (2010) proposed a method that also extracts objects from the displayed image and embeds them into a logic-based architecture, SOAR. Their method uses a forward model of the scene to improve the performance of the Q-Learning algorithm (Watkins & Dayan, 1992). They showed that by using such a model, a reinforcement learning agent could learn to play a restricted version of the game of Frogger. Cobo, Zang, Isbell, and Thomaz (2011) investigated automatic feature discovery in the games of Pong and Frogger, using their own simulator. Their proposed method takes advantage of human trajectories to identify state features that are important for playing console games. Recently, Hausknecht, Khandelwal, Miikkulainen, and Stone (2012) proposed HyperNEAT-GGP, an evolutionary approach for finding policies to play Atari 2600 games. Although HyperNEAT-GGP is presented as a general game playing approach, it is currently difficult to assess its general performance as the reported results were limited to only two games. Finally, some of the authors of this paper (Bellemare, Veness, & Bowling, 2012) recently presented a domain-independent feature generation technique that attempts to focus its effort around the location of the player avatar. This work used the evaluation methodology advocated here and is the only one to demonstrate the technique across a large set of testing games.

5.2 Evaluation Frameworks for General Agents

Although the idea of using games to evaluate the performance of agents has a long history in artificial intelligence, it is only more recently that an emphasis on generality has assumed a more prominent role. Pell (1993) advocated the design of agents that, given an abstract description of a game, could automatically play them. His work strongly influenced the design of the now annual General Game Playing competition (Genesereth et al., 2005). Our framework differs in that we do not assume to have access to a compact logical description of the game semantics. Schaul, Togelius, and Schmidhuber (2011) also recently presented an interesting proposal for using games to measure the general capabilities of an agent. Whiteson et al. (2011) discuss a number of challenges in designing empirical tests to measure general reinforcement learning performance; this work can be seen as attempting to address their important concerns.

Starting in 2004 as a conference workshop, the Reinforcement Learning competition (Whiteson et al., 2010) was held until 2009 (a new iteration of the competition has been announced for 2013[4]). Each year new domains are proposed, including standard RL benchmarks, Tetris, and Infinite Mario (Mohan & Laird, 2009). In a typical competition domain, the agent's state information is summarized through a series of high-level state variables rather than direct sensory information. Infinite Mario, for example, provides the agent with an object-oriented observation space. In the past, organizers have provided a special 'Polyathlon' track in which agents must behave in a medley of continuous-observation, discrete-action domains.

Another longstanding competition, the International Planning Competition (IPC)[5], has been organized since 1998, and aims to "produce new benchmarks, and to gather and dis-

4. http://www.rl-competition.org
5. http://ipc.icaps-conference.org

seminate data about the current state-of-the-art" (Coles et al., 2012). The IPC is composed of different tracks corresponding to different types of planning problems, including factory optimization, elevator control and agent coordination. For example, one of the problems in the 2011 competition consists in coordinating a set of robots around a two-dimensional gridworld so that every tile is painted with a specific colour. Domains are described using either relational reinforcement learning, yielding parametrized Markov Decision Processes (MDPs) and Partially Observable MDPs, or using logic predicates, e.g. in STRIPS notation.

One indication of how much these competitions value domain variety can be seen in the time spent on finding a good specification language. The 2008-2009 RL competitions, for example, used RL-Glue[6] specifically for this purpose; the 2011 planning under uncertainty track of the IPC similar employed the Relation Dynamic Influence Diagram Language. While competitions seek to spur new research and evaluate existing algorithms through a standardized set of benchmarks, they are *not* independently developed, in the sense that the vast majority of domains are provided by the research community. Thus a typical competition domain reflects existing research directions: Mountain Car and Acrobot remain staples of the RL competition. These competitions also focus their research effort on domains that provide high-level state variables, for example the location of robots in the floor-painting domain described above. By contrast, the Arcade Learning Environment and the domain-independent setting force us to consider the question of perceptual grounding: how to extract meaningful state information from raw game screens (or RAM information). In turn, this emphasizes the design of algorithms that can be applied to sensor-rich domains without significant expert knowledge.

There have also been a number of attempts to define formal agent performance metrics based on algorithmic information theory. The first such attempts were due to Hernández-Orallo and Minaya-Collado (1998) and to Dowe and Hajek (1998). More recently, the approaches of Hernández-Orallo and Dowe (2010) and of Legg and Veness (2011) appear to have some potential. Although these frameworks are general and conceptually clean, the key challenge remains how to specify sufficiently interesting classes of environments. In our opinion, much more work is required before these approaches can claim to rival the practicality of using a large set of existing human-designed environments for agent evaluation.

6. Final Remarks

The Atari 2600 games were developed for humans and as such exhibit many idiosyncrasies that make them both challenging and exciting. Consider, for example, the game Pong. Pong has been studied in a variety of contexts as an interesting reinforcement learning domain (Cobo et al., 2011; Stober & Kuipers, 2008; Monroy, Stanley, & Miikkulainen, 2006). The Atari 2600 Pong, however, is significantly more complex than Pong domains developed for research. Games can easily last 10,000 time steps (compared to 200–1000 in other domains); observations are composed of 7-bit 160×210 images (compared to 300×200 black and white images in the work of Stober and Kuipers (2008), or 5-6 input features elsewhere); observations are also more complex, containing the two players' score and side walls. In sheer size, the Atari 2600 Pong is thus a larger domain. Its dynamics are also

6. `http://glue.rl-community.org`

more complicated. In research implementations of Pong object motion is implemented using first-order mechanics. However, in Atari 2600 Pong paddle control is nonlinear: simple experimentation shows that fully predicting the player's paddle requires knowledge of the last 18 actions. As with many other Atari games, the player paddle also moves every other frame, adding a degree of temporal aliasing to the domain.

While Atari 2600 Pong may appear unnecessarily contrived, it in fact reflects the unexpected complexity of the problems with which humans are faced. Most, if not all Atari 2600 games are subject to similar programming artifacts: in Space Invaders, for example, the invaders' velocity increases nonlinearly with the number of remaining invaders. In this way the Atari 2600 platform provides AI researchers with something unique: clean, easily-emulated domains which nevertheless provide many of the challenges typically associated with real-world applications.

Should technology advance so as to render general Atari 2600 game playing achievable, our challenge problem can always be extended to use more recent video game platforms. A natural progression, for example, would be to move on to the Commodore 64, then to the Nintendo, and so forth towards current generation consoles. All of these consoles have hundreds of released games, and older platforms have readily available emulators. With the ultra-realism of current generation consoles, each console represents a natural stepping stone toward general real-world competency. Our hope is that by using the methodology advocated in this paper, we can work in a bottom-up fashion towards developing more sophisticated AI technology while still maintaining empirical rigor.

7. Conclusion

This article has introduced the Arcade Learning Environment, a platform for evaluating the development of general, domain-independent agents. ALE provides an interface to hundreds of Atari 2600 game environments, each one different, interesting, and designed to be a challenge for human players. We illustrate the promise of ALE as a challenge problem by benchmarking several domain-independent agents that use well-established reinforcement learning and planning techniques. Our results suggest that general Atari game playing is a challenging but not intractable problem domain with the potential to aid the development and evaluation of general agents.

Acknowledgments

We would like to thank Marc Lanctot, Erik Talvitie, and Matthew Hausknecht for providing suggestions on helping debug and improving the Arcade Learning Environment source code. We would also like to thank our reviewers for their helpful feedback and enthusiasm about the Atari 2600 as a research platform. The work presented here was supported by the Alberta Innovates Technology Futures, the Alberta Innovates Centre for Machine Learning at the University of Alberta, and the Natural Science and Engineering Research Council of Canada. Invaluable computational resources were provided by Compute/Calcul Canada.

Appendix A. Feature Set Construction

This section gives a detailed description of the five feature generation techniques from Section 3.1.

A.1 Basic Abstraction of the ScreenShots (BASS)

The idea behind BASS is to directly encode colours present on the screen. This method is motivated by three observations on the Atari 2600 hardware and games:

1. While the Atari 2600 hardware supports a screen resolution of 160×210, game objects are usually larger than a few pixels. Overall, important game events happen at a much lower resolution.

2. Many Atari 2600 games have a static background, with a few important objects moving on the screen. While the screen matrix is densely populated, the actual interesting features on the screen are often sparse.

3. While the hardware can show up to 128 colours in the NTSC mode, it is limited to only 8 colours in the SECAM mode. Consequently, most games use a few number of colours to distinguish important objects on the screen.

The game screen is first preprocessed by subtracting its background, detected using a simple histogram method. BASS then encodes the presence of each of the eight SECAM palette colours at a low resolution, as depicted in Figure 5. Intuitively, BASS seeks to capture the presence of objects of certain colours at different screen locations. BASS also encodes relations between objects by constructing all pairwise combinations of its encoded colour features. In ASTERIX, for example, it is important to know if there is a green object (player character) *and* a red object (collectable object) in its vicinity. Pairwise features allow us to capture such object relations.

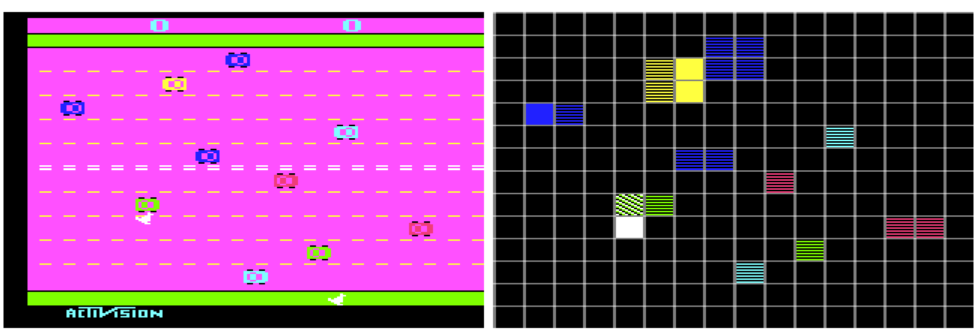

Figure 5: **Left:** Freeway in SECAM colours. **Right:** BASS colour encoding for the same screen.

A.2 Basic

The Basic method generates the same set of features as BASS, but omits the pairwise combinations. This allows us to study whether the additional features are beneficial or harmful to learning. Because the Basic method has fewer features than BASS, it encodes the presence of each of the 128 colours. In comparison to BASS, Basic therefore represents colour more accurately, but cannot represent object interactions.

A.3 Detecting Instances of Classes of Objects (DISCO)

This feature generation method is based on detecting a set of classes representing game entities and locating instances of these classes on the screen. DISCO is motivated by the following additional observations on Atari 2600 games:

1. The game entities are often instances of a few *classes* of objects. For instance, as Figure 6 shows, while there are many objects in a sample screen of the game Freeway, all of these objects are instances of only two classes: *Chicken* and *Car*. Similarly, all the objects on a sample screen of the game Seaquest are instances of one of these six classes: *Fish, Swimmer, Player Submarine, Enemy Submarine, Player Bullet,* and *Enemy Bullet*.

2. The interaction between two objects can often be generalized to all instances of their respective classes. As an example, consider *Car-Chicken* object interactions in Freeway: learning that there is lower value associated with one *Chicken* instance hitting a *Car* instance can be generalized to all instances of those two classes.

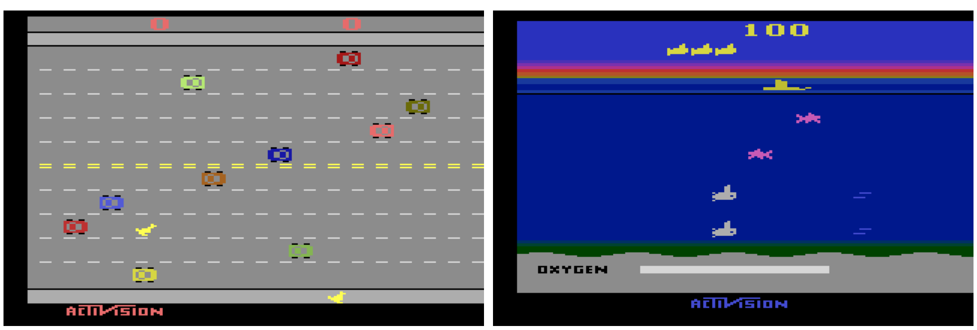

Figure 6: **Left:** Screenshot of the game Freeway. Although there are ten different cars, they can all be considered as instances of a single class. **Right:** Screenshot of the game Seaquest depicting four different object classes.

DISCO first performs a series of preprocessing steps to discover classes, during which no value function learning is performed. When the agent subsequently learns to play the game, DISCO generates features by detecting objects on the screen and classifying them. The DISCO process is summarized by the following steps:

- Preprocessing:
  - **Background detection:** The static background matrix is extracted using a histogram method, as with BASS.

Algorithm 1 Locally Sensitive Hashing (LSH) Feature Generation

Constants. M (hash table size), n (screen bit vector size)
l (number of random bit vectors), k (number of non-zero entries)
Initialization (once).
$\{v_1 \dots v_l\} \leftarrow$ generateRandomVectors(l, k, n)
$\{\text{hash}_1 \dots \text{hash}_l\} \leftarrow$ generateHashFunctions(l, M, n)

Input. A screen matrix I with elements $I_{xy} \in \{0, \dots, 127\}$

LSH(I)
$s \leftarrow$ binarizeScreen(I) (s has length n)
Initialize $\phi \in \mathbb{R}^{lM} = 0$
for $i = 1 \dots l$ **do**
$h = 0$
for $j = 1 \dots n$ **do**
$h \leftarrow h + \mathbb{I}_{[s_j = v_{ij}]}\text{hash}_i[j] \mod M$ (hash the projection of s onto v_i)
end for
$\phi[M(i-1) + h] = 1$ (one binary feature per random bit vector)
end for

binarizeScreen(I)
Initialize $s \in \mathbb{R}^n = 0$
for $y = 1 \dots h$, $x = 1 \dots w$ ($h = 210, w = 160$) **do**
$s[x + y * h + I_{xy}] = 1$
end for
return s

generateRandomVectors(l, k, n)
Initialize $v_1 \dots v_l \in \mathbb{R}^n = 0$
for $i = 1 \dots l$ **do**
Select $x_1, x_2, \dots, x_k$ distinct coordinates between 1 and n uniformly at random
$v_i[x_1] = 1$; $v_i[x_2] = 1$; $\dots$; $v_i[x_k] = 1$
end for
return $\{v_1, \dots v_l\}$

generateHashFunctions(l, M, n) (hash functions are vectors of random coordinates)
Initialize $\text{hash}_1 \dots \text{hash}_l \in \mathbb{R}^n = 0$
for $i = 1 \dots l$, $j = 1 \dots n$ **do**
$\text{hash}_i[j] \leftarrow$ random$(1, M)$ (uniformly random coordinate between 1 and M)
end for
return $\{\text{hash}_1, \dots \text{hash}_l\}$

Remark. With sparse vector operations, LSH has a $O(lk + n)$ cost per step.

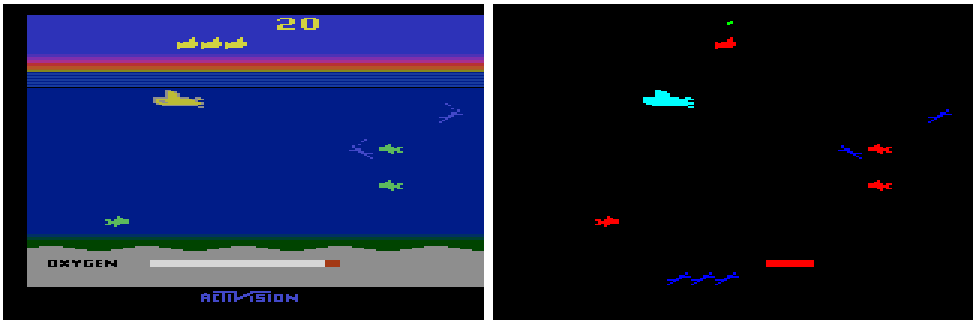

Figure 7: **Left:** Screenshot of the game Seaquest. **Right:** Objects detected by DISCO in the game Seaquest. Each colour represents a different class.

- **Blob extraction:** A list of moving blob (foreground) objects is detected in each game screen.
- **Class discovery:** A set of classes is detected from the extracted blob objects.
- **Class filtering:** Classes that appear infrequently or are restricted to small region of the screen are removed from the set.
- **Class merging:** Classes that have similar shapes are merged together.

- Feature generation:
  - **Class instance detection:** At each time step, class instances are detected from the current screen matrix.
  - **Feature vector generation:** A feature vector is generated from the detected instances by tile-coding their absolute position as well as the relative position and velocity of every pair of instances from different classes. Multiple instances of the same objects are combined additively.

Figure 7 shows discovered objects in a Seaquest frame. This image illustrates the difficulties in detecting objects: although DISCO correctly classifies the different fish as part of the same class, it also detects a life icon and the oxygen bar as part of that class.

A.4 Locality Sensitive Hashing (LSH)

An alternative approach to BASS and DISCO is to use well-established feature generation methods that are agnostic about the type of input they receive. Such methods include polynomial bases (Schweitzer & Seidmann, 1985), sparse distributed memories (Kanerva, 1988) and locality sensitive hashing (LSH) (Gionis et al., 1999). In this paper we consider the latter as a simple mean of reducing the large image space to a smaller, more manageable set of features. The input – here, a game screen – is first mapped to a bit vector of size $7 \times 210 \times 160$. The resulting vector is then hashed down into a smaller set of features. LSH performs an additional random projection step to ensure that similar screens are more likely to be binned together. The LSH generation method is detailed in Algorithm 1.

A.5 RAM-based Feature Generation

Unlike the previous three methods, which generate feature vectors based on the game screen, the RAM-based feature generation method relies on the contents of the console memory. The Atari 2600 has only $128 \times 8 = 1024$ bits of random access memory[7], which must hold the complete internal state of a game: location of game entities, timers, health indicators, etc. The RAM is therefore a relatively compact representation of the game state, and in contrast to the game screen, it is also Markovian. The purpose of our RAM-based agent is to investigate whether features generated from the RAM affect performance differently from features generated from game screens.

The first part of the generated feature vector simply includes the 1024 bits of RAM. Atari 2600 game programmers often used these bits not as individual values, but as part of 4-bit or 8-bit words. Linear function approximation on the individual bits can capture the value of these multi-bit words. We are also interested in the relation between pairs of values in memory. To capture these relations, the logical-AND of all possible bit pairs is appended to the feature vector. Note that a linear function on the pairwise AND's can capture products of both 4-bit and 8-bit words. This is because the product of two n-bit words can be expressed as a weighted sum of the pairwise products of their bits.

7. Some games provided more RAM on the game cartridge: the *Atari Super Chip*, for example, offered an additional 128 bytes of memory. The current approach only considers the main memory included in the Atari 2600 console.

Appendix B. UCT Pseudocode

Algorithm 2 UCT

Constants. m (search horizon), k (simulations per step)
Variables. Ψ (search tree)
Input. s (current state)

```
UCT(s)
  if Ψ is empty or root(Ψ) ≠ s then
    Ψ ← empty search tree
    root(Ψ) ← s
  end if
  repeat
    sample(Ψ, m)
  until visits(root(Ψ)) = k
  a ← bestAction(Ψ)
  prune(Ψ, a)        (optional)
  return a

sample(Ψ, m)
  n ← root(Ψ)
  while n is not a leaf, m > depth(n) do
    if some action a was never taken in n then
      (c, reward) ← emulate(n, a)        (run model for one step)
      immediate-return(c) ← reward
      child(n, a) ← c
      n ← c        (c is necessarily a leaf)
    else
      a ← selectAction(n)
      n ← child(n, a)
    end if
  end while
  R = rollout(n, m − depth(n))
  update-value(n, R)        (propagate values back up)

bestAction(Ψ)
  return arg max_a [visits(child(root(Ψ), a))]        (action most frequently taken at root)

prune(Ψ, a)
  root(Ψ) ← child(root(Ψ), a)
```

Algorithm 3 UCT Routines

Constants. γ : discount factor

selectAction(n)
  for all c children of n **do**
    $V(c) \leftarrow \text{average-return}(c) + \sqrt{\frac{\log[\text{visits}(c)]}{\text{visits}(n)}}$
  end for
  return $\arg\max_a V(\text{child}(n, a))$

rollout(n, m)
  $R = 0$ (Initialize Monte-Carlo return to 0)
  $g = 1$
  while $m > 0$ **do**
    Select a according to some rollout policy (e.g. uniformly randomly)
    $(n, \text{reward}) \leftarrow \text{emulate}(n, a)$
    $R \leftarrow R + g \times \text{reward}$
    $m \leftarrow m - 1$
    $g \leftarrow g \times \gamma$
  end while
  return R

update-value(n, R)
  $R \leftarrow R + \text{immediate-reward(n)}$
  $\text{average-return}(n) \leftarrow \text{average-return}(n)\frac{\text{visits}(n)}{\text{visits}(n)+1} + \frac{R}{\text{visits}(n)+1}$
  $\text{visits}(n) \leftarrow \text{visits}(n) + 1$
  if n is not the root of Ψ, i.e. $\text{parent}(n) \neq \text{null}$ **then**
    $\text{update-value}(\text{parent}(n), \gamma \times R)$
  end if

Appendix C. Experimental Parameters

General	All experiments	Maximum frames per episode	18,000
		Frames per action	5
	Reinforcement learning	Training episodes per trial	5,000
		Evaluation episodes per trial	500
		Number of trials per result	30
Preprocessing	Background detection	Sample screens per game	18,000
	Class discovery	Sample screens per game	36,000
		Maximum number of classes	10
		Maximum object velocity (pixels)	8
		Minimum frequency of class appearance	20%
Reinforcement learning	All agents	Discount factor γ	0.999
		Exploration rate ϵ	0.05
	BASS and Basic	Learning rate α	0.5
		Eligibility traces decay rate λ	0.9
		Grid width	16
		Grid height	14
	BASS only	Number of different colours	8
	Basic only	Number of different colours	128
	DISCO	Learning rate α	0.1
		Eligibility traces decay rate λ	0.9
		Tile coding, number of tilings	8
		Tile coding, grid size	8
	RAM-based	Learning rate α	0.2
		Eligibility traces decay rate λ	0.5
	LSH	Learning rate α	0.5
		Eligibility traces decay rate λ	0.5
		Number of random vectors l	2000
		Number of non-zero vector entries k	1000
		Per-vector hash table size M	50
Planning	UCT	Simulations per action	500
		Maximum search depth (frames)	300
		Exploration constant	0.1
	Full-tree search	Maximum frames emulated per action	133,000

Appendix D. Detailed Results

D.1 Reinforcement Learning

Game	Basic	BASS	DISCO	LSH	RAM	Random	Const	Perturb
Asterix	862.3	859.8	754.6	**987.3**	943.0	288.1	650.0	337.8
Beam Rider	929.4	872.7	563.0	793.6	729.8	434.7	**996.0**	754.8
Freeway	11.3	16.4	12.8	15.4	19.1	0.0	21.0	**22.5**
Seaquest	579.0	**664.8**	421.9	508.5	593.7	107.9	160.0	451.1
Space Invaders	203.6	250.1	239.1	222.2	226.5	156.1	245.0	**270.5**
Alien	**939.2**	893.4	623.6	510.2	726.4	102.0	140.0	313.9
Amidar	64.9	**103.4**	67.9	45.1	71.4	0.8	31.0	37.8
Assault	465.8	378.4	371.7	**628.0**	383.6	334.3	357.0	497.8
Asteroids	829.7	800.3	744.5	590.7	907.3	**1526.7**	140.0	539.9
Atlantis	**62687.0**	25375.0	20857.3	17593.9	19932.7	33058.4	1500.0	12089.1
Bank Heist	98.8	71.1	51.4	64.6	**190.8**	15.0	0.0	13.5
Battle Zone	15534.3	12750.8	0.0	14548.1	**15819.7**	2920.0	13000.0	5772.0
Berzerk	329.2	491.3	329.0	441.0	501.3	233.8	**670.0**	552.9
Bowling	28.5	**43.9**	35.2	26.1	29.3	24.6	30.0	30.0
Boxing	-2.8	15.5	12.4	10.5	**44.0**	-1.5	-25.0	-10.1
Breakout	3.3	**5.2**	3.9	2.5	4.0	1.5	3.0	2.9
Carnival	**2323.9**	1574.2	1646.3	1147.2	765.4	869.2	0.0	485.4
Centipede	7725.5	8803.8	6210.6	6161.6	7555.4	2805.1	**16527.0**	8937.2
Chopper Command	1191.4	**1581.5**	1349.0	943.0	1397.8	698.2	1000.0	973.7
Crazy Climber	6303.1	7455.6	4552.9	20453.7	**23410.6**	2335.4	0.0	2235.0
Demon Attack	520.5	318.5	208.8	355.8	324.8	289.3	130.0	**776.2**
Double Dunk	-15.8	-13.1	-23.2	-21.6	-20.3	-15.6	**0.0**	-20.3
Elevator Action	3025.2	2377.6	4.6	**3220.6**	507.9	1040.9	0.0	562.9
Enduro	111.8	**129.1**	0.0	95.8	112.3	0.0	9.0	25.9
Fishing Derby	-92.6	-92.1	**-89.5**	-93.2	-91.6	-93.8	-99.0	-97.2
Frostbite	161.0	161.1	176.6	**216.9**	147.9	70.3	160.0	175.2
Gopher	545.8	**1288.3**	295.7	941.8	722.5	243.7	0.0	286.8
Gravitar	185.3	251.1	197.4	105.9	**387.7**	205.4	0.0	106.0
H.E.R.O.	6053.1	**6458.8**	2719.8	3835.8	3281.1	712.0	0.0	147.5
Ice Hockey	-13.9	-14.8	-18.9	-15.1	-9.5	-14.8	**-1.0**	-6.5
James Bond	197.3	**202.8**	17.3	77.1	133.8	23.3	0.0	82.0
Journey Escape	-8441.0	-14730.7	-9392.2	-13898.9	-8713.5	-18201.7	**0.0**	-10693.9
Kangaroo	962.4	**1622.1**	457.9	256.4	481.7	44.4	200.0	498.4
Krull	2823.3	**3371.5**	2350.9	2798.1	2901.3	1880.1	0.0	1690.1
Kung-Fu Master	16416.2	**19544.0**	3207.0	8715.6	10361.1	488.2	0.0	578.4
Montezuma's Revenge	**10.7**	0.1	0.0	0.1	0.3	0.3	0.0	0.0
Ms. Pac-Man	1537.2	**1691.8**	999.6	1070.8	1021.1	163.3	210.0	505.5
Name This Game	1818.9	2386.8	1951.0	2029.8	2500.1	2012.3	**3080.0**	1854.3
Pooyan	800.3	1018.9	402.7	**1225.3**	1210.9	501.1	30.0	540.8
Pong	-19.2	**-19.0**	-19.6	-19.9	-19.9	-20.9	-21.0	-20.8
Private Eye	81.9	100.7	-23.0	684.3	111.9	-754.0	0.0	**1947.3**
Q*Bert	**613.5**	497.2	326.3	529.1	565.8	169.0	150.0	157.4
River Raid	1708.9	1438.0	0.0	**1904.3**	1309.9	1608.6	1070.0	1455.5
Road Runner	67.7	65.2	21.4	42.0	41.0	36.2	**900.0**	857.9
Robotank	12.8	10.1	9.3	10.8	**28.7**	1.6	17.0	11.3
Skiing	-1.1	-0.7	-0.1	**-0.0**	0.0	0.0	0.0	0.0
Star Gunner	850.2	**1069.5**	1002.2	722.9	769.3	638.1	600.0	509.8
Tennis	-0.2	-0.1	-0.1	-0.1	-0.1	-24.0	**0.0**	-0.3
Time Pilot	1728.2	2299.5	0.0	2429.2	**3741.2**	3458.8	500.0	718.7
Tutankham	40.7	52.6	0.0	85.2	**114.3**	23.1	0.0	17.3
Up and Down	**3532.7**	3351.0	2473.4	2475.1	3412.6	131.6	550.0	2962.9
Venture	0.0	**66.0**	0.0	0.0	0.0	0.0	0.0	0.0
Video Pinball	15046.8	12574.2	10779.5	9813.9	16871.3	**20021.1**	705.0	9527.9
Wizard of Wor	1768.8	**1981.3**	935.6	945.5	1096.2	772.4	300.0	470.3
Zaxxon	1392.0	2069.1	69.8	**3365.1**	304.3	0.0	0.0	2.0
Times Best	6	17	1	8	8	2	9	4

Table 4: Reinforcement Learning results. The first five games constitute our training set. See Section 3.1 for details.

D.2 Planning

Game	Full Tree	UCT	Best Learner	Best Baseline
Asterix	2135.7	**290700.0**	987.3	650.0
Beam Rider	693.5	**6624.6**	929.4	996.0
Freeway	0.0	0.4	19.1	**22.5**
Seaquest	288.0	**5132.4**	664.8	451.1
Space Invaders	112.2	**2718.0**	250.1	270.5
Alien	784.0	**7785.0**	939.2	313.9
Amidar	5.2	**180.3**	103.4	37.8
Assault	413.7	**1512.2**	628.0	497.8
Asteroids	3127.4	**4660.6**	907.3	1526.7
Atlantis	30460.0	**193858.0**	62687.0	33058.4
Bank Heist	21.5	**497.8**	190.8	15.0
Battle Zone	6312.5	**70333.3**	15819.7	13000.0
Berzerk	195.0	553.5	501.3	**670.0**
Bowling	25.5	25.1	**43.9**	30.0
Boxing	**100.0**	**100.0**	44.0	-1.5
Breakout	1.1	**364.4**	5.2	3.0
Carnival	950.0	**5132.0**	2323.9	869.2
Centipede	**125123.0**	110422.0	8803.8	16527.0
Chopper Command	1827.3	**34018.8**	1581.5	1000.0
Crazy Climber	37110.0	**98172.2**	23410.6	2335.4
Demon Attack	442.6	**28158.8**	520.5	776.2
Double Dunk	-18.5	**24.0**	-13.1	0.0
Elevator Action	730.0	**18100.0**	3220.6	1040.9
Enduro	0.6	**286.3**	129.1	25.9
Fishing Derby	-91.6	**37.8**	-89.5	-93.8
Frostbite	137.2	**270.5**	216.9	175.2
Gopher	1019.0	**20560.0**	1288.3	286.8
Gravitar	395.0	**2850.0**	387.7	205.4
H.E.R.O.	1323.8	**12859.5**	6458.8	712.0
Ice Hockey	-9.2	**39.4**	-9.5	-1.0
James Bond	25.0	**330.0**	202.8	82.0
Journey Escape	1327.3	**7683.3**	-8441.0	0.0
Kangaroo	90.0	**1990.0**	1622.1	498.4
Krull	3089.2	**5037.0**	3371.5	1880.1
Kung-Fu Master	12127.3	**48854.5**	19544.0	578.4
Montezuma's Revenge	0.0	0.0	**10.7**	0.3
Ms. Pacman	1708.5	**22336.0**	1691.8	505.5
Name This Game	5699.0	**15410.0**	2500.1	3080.0
Pooyan	909.7	**17763.4**	1225.3	540.8
Pong	-20.7	**21.0**	-19.0	-20.8
Private Eye	57.9	100.0	684.3	**1947.3**
Q*Bert	132.8	**17343.4**	613.5	169.0
River Raid	2178.5	**4449.0**	1904.3	1608.6
Road Runner	245.0	**38725.0**	67.7	900.0
Robotank	1.5	**50.4**	28.7	17.0
Skiing	**0.0**	-0.8	0.0	0.0
Star Gunner	**1345.0**	1207.1	1069.5	638.1
Tennis	-23.8	**2.8**	-0.1	0.0
Time Pilot	4063.6	**63854.5**	3741.2	3458.8
Tutankham	64.1	**225.5**	114.3	23.1
Up and Down	746.0	**74473.6**	3532.7	2962.9
Venture	0.0	0.0	**66.0**	0.0
Video Pinball	55567.3	**254748.0**	16871.3	20021.1
Wizard of Wor	3309.1	**105500.0**	1981.3	772.4
Zaxxon	0.0	**22610.0**	3365.1	2.0
Times Best	4	45	3	3

Table 5: Search results. The first five games constitute our training set. See Section 3.2 for details.

References

Bellemare, M., Veness, J., & Bowling, M. (2012). Investigating contingency awareness using Atari 2600 games. In *Proceedings of the the 26th Conference on Artificial Intelligence (AAAI)*.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., & Colton, S. (2012). A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(1), 1 –43.

Cobo, L. C., Zang, P., Isbell, C. L., & Thomaz, A. L. (2011). Automatic state abstraction from demonstration. In *Proceedings of the 22nd Second International Joint Conference on Articial Intelligence (IJCAI)*.

Coles, A., Coles, A., Olaya, A., Jiménez, S., López, C., Sanner, S., & Yoon, S. (2012). A survey of the seventh international planning competition. *AI Magazine*, *33*(1), 83–88.

Diuk, C., Cohen, A., & Littman, M. L. (2008). An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th International Conference on Machine learning (ICML)*.

Dowe, D. L., & Hajek, A. R. (1998). A non-behavioural, computational extension to the Turing Test. In *Proceedings of the International Conference on Computational Intelligence and Multimedia Applications (ICCIMA)*.

Genesereth, M. R., Love, N., & Pell, B. (2005). General Game Playing: Overview of the AAAI competition. *AI Magazine*, *26*(2), 62–72.

Gionis, A., Indyk, P., & Motwani, R. (1999). Similarity search in high dimensions via hashing. In *Proceedings of the International Conference on Very Large Databases*.

Hausknecht, M., Khandelwal, P., Miikkulainen, R., & Stone, P. (2012). HyperNEAT-GGP: A HyperNEAT-based Atari general game player. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*.

Hernández-Orallo, J., & Dowe, D. L. (2010). Measuring universal intelligence: Towards an anytime intelligence test. *Artificial Intelligence*, *174*(18), 1508 – 1539.

Hernández-Orallo, J., & Minaya-Collado, N. (1998). A formal definition of intelligence based on an intensional variant of Kolmogorov complexity. In *Proceedings of the International Symposium of Engineering of Intelligent Systems (EIS)*.

Hutter, M. (2005). *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability*. Springer, Berlin.

Kanerva, P. (1988). *Sparse Distributed Memory*. The MIT Press.

Kocsis, L., & Szepesvári, C. (2006). Bandit based Monte-Carlo planning. In *Proceedings of the 15th European Conference on Machine Learning (ECML)*.

Legg, S. (2008). *Machine Super Intelligence*. Ph.D. thesis, University of Lugano.

Legg, S., & Veness, J. (2011). An approximation of the universal intelligence measure. In *Proceedings of the Ray Solomonoff Memorial Conference*.

Mohan, S., & Laird, J. E. (2009). Learning to play Mario. Tech. rep. CCA-TR-2009-03, Center for Cognitive Architecture, University of Michigan.

Monroy, G. A., Stanley, K. O., & Miikkulainen, R. (2006). Coevolution of neural networks using a layered pareto archive. In *Proceedings of the 8th Genetic and Evolutionary Computation Conference (GECCO)*.

Montfort, N., & Bogost, I. (2009). *Racing the Beam: The Atari Video Computer System*. MIT Press.

Naddaf, Y. (2010). Game-Independent AI Agents for Playing Atari 2600 Console Games. Master's thesis, University of Alberta.

Pell, B. (1993). *Strategy Generation and Evaluation for Meta-Game Playing*. Ph.D. thesis, University of Cambridge.

Pierce, D., & Kuipers, B. (1997). Map learning with uninterpreted sensors and effectors. *Artificial Intelligence*, *92*(1-2), 169–227.

Russell, S. J. (1997). Rationality and intelligence. *Artificial intelligence*, *94*(1), 57–77.

Schaul, T., Togelius, J., & Schmidhuber, J. (2011). Measuring intelligence through games. *CoRR*, *abs/1109.1314*.

Schweitzer, P. J., & Seidmann, A. (1985). Generalized polynomial approximations in Markovian decision processes. *Journal of mathematical analysis and applications*, *110*(2), 568–582.

Stober, J., & Kuipers, B. (2008). From pixels to policies: A bootstrapping agent. In *Proceedings of the 7th IEEE International Conference on Development and Learning (ICDL)*.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. The MIT Press.

Sutton, R., Modayil, J., Delp, M., Degris, T., Pilarski, P., White, A., & Precup, D. (2011). Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *Proceedings of the 10th International Conference on Autonomous Agents and Multiagents Systems (AAMAS)*.

Thrun, S., & Mitchell, T. M. (1995). Lifelong robot learning. *Robotics and Autonomous Systems*, *15*(1), 25–46.

Watkins, C., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*, 279–292.

Whiteson, S., Tanner, B., Taylor, M. E., & Stone, P. (2011). Protecting against evaluation overfitting in empirical reinforcement learning. In *Proceedings of the IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*.

Whiteson, S., Tanner, B., & White, A. (2010). The reinforcement learning competitions. *AI Magazine*, *31*(2), 81–94.

Wintermute, S. (2010). Using imagery to simplify perceptual abstraction in reinforcement learning agents. In *Proceedings of the the 24th Conference on Artificial Intelligence (AAAI)*.