Assisted Feature Engineering and Feature Learning to Build Knowledge-Based Agents for Arcade Games

Bastian Andelefski and Stefan Schiffer

Knowledge-Based Systems Group
RWTH Aachen University, Aachen, Germany
{andelefski, schiffer}@kbsg.rwth-aachen.de

Abstract: Human knowledge can greatly increase the performance of autonomous agents. Leveraging this knowledge is sometimes neither straightforward nor easy. In this paper, we present an approach for assisted feature engineering and feature learning to build knowledge-based agents for three arcade games within the Arcade Learning Environment. While existing approaches mostly use model-free approaches we aim at creating a descriptive set of features for world modelling and building agents. To this end, we provide (visual) assistance in identifying and modelling features from RAM, we allow for learning features based on labeled game data, and we allow for creating basic agents using the above features. In our evaluation, we compare different methods to learn features from the RAM. We then compare several agents using different sets of manual and learned features with one another and with the state-of-the-art.

1 INTRODUCTION

In many domains, the knowledge of domain experts can greatly help in constructing agents that perform well in a certain task. However, how to transfer such knowledge from the expert into an agent formulation is often neither obvious nor easy to do. In this paper, we tackle the creation of knowledge-based agents for arcade style games. We do so by means of assisted feature engineering and feature learning. The resulting features are then used to construct knowledge-based agents. The goal is to make use of expert knowledge as conveniently as possible. We leverage human pattern recognition skills and modelling competence and combine it with machine learning techniques. With our approach we maximize the impact of training data in the learning and keep the agent's decision making traceable.

For our application domain, we chose the Arcade Learning Environment (ALE) introduced by Bellemare (Bellemare et al., 2013). Based on the Atari 2600 emulator Stella, it provides a unified interface for a large number of arcade games. Notably, this interface only offers raw data, specifically screen and RAM content as well as the current score. As a result, most agents have so far sidestepped the absence of an explicit world model by using model-free approaches. An early ALE paper by Naddaf (Naddaf,

2010) is an interesting exception to this trend. Naddaf compares several approaches of reinforcement learning and planning, including a few attempts at creating better features by image recognition. While this preprocessing does increase performance compared to the raw image data, it does not result in significant improvements over the raw RAM data. These results seem to indicate, that the RAM contains all relevant information in a format that is well suited to machine learning algorithms. It should therefore be possible to isolate important features and turn them into a simple world model.

We investigate this by building a toolkit and framework that makes it simple and fast to analyze the data and create appropriate world models from information contained in the RAM. Our goal is to use these tools to explore possible benefits of using experts and a model-based approach. We do this by using domain knowledge and human pattern recognition skills to define highly relevant features. These features are then used to construct agents using decision tree learning.

The outline of the paper is as follows. We start by introducing the background of our work, namely the Arcade Learning Environment and the games we use as well as the scikit learning toolkit and the learning algorithms we use. After reviewing related work on arcade game agents and feature learning we present our approach. We detail our assistance with analysing

the RAM for manual feature creation and elaborate on our feature learning. A brief view on our agent design is followed by an comprehensive evaluation of the feature learning. We then compare the agents' performance in playing arcade games with different sets of features and with the state-of-the-art.

2 BACKGROUND

We introduce the frameworks and libraries that the work described in this paper is based on. Further, we present the foundations that underlie the learning methods we use in our toolkit.

2.1 The Arcade Learning Environment

The Arcade Learning Environment was introduced by Bellemare as an engaging way to compare game playing agents. The benchmark is based on the classic gaming console Atari 2600[1] and offers a wide selection of highly diverse games with different control schemes and goals. One of the most engaging qualities of ALE is the extremely restricted nature of the system. While the games feature non-essential information to make them more engaging to players, they are running on a platform with only 128 bytes of RAM and 18 possible actions. These actions are listed in Table 1.

This dichotomy creates interesting AI domains that are quite complicated, but remain within the computational scope of modern hardware. The number of actions is reasonably small which makes it interesting also for planning approaches. But even in learning-based approaches it is helpful, because the training data for a game playing agent is spread across fewer actions and therefore more likely to be decisive.

2.2 Scikit-learn

Both, our feature learning and our game playing agent are based on machine learning methods. To ensure reliable performance and save time on our implementation, we use the machine learning library scikit-learn (Pedregosa et al., 2011) for all machine learning methods in this paper.

Scikit-learn[2] was first introduced in 2007 and has grown into an extensive resource for state-of-the-art implementations of machine learning algorithms. Its high performance, active maintenance and liberal BSD license make it an obvious choice among the long list of machine learning libraries.

---

[1] https://en.wikipedia.org/wiki/Atari_2600
[2] http://scikit-learn.org/

2.3 Learning Methods

We now give a short introduction into the classifiers used in our feature learning approach. A discussion of why each classifier was chosen can be found in Section 4.3.1.

Random trees are a way to counteract overfitting with decision tree learning. With classical approaches to decision trees it is not possible to increase generalization and accuracy at the same time. Random trees achieve this by training multiple decision trees with random sub-sets of the feature vector and training data. A collection of such trees then yields a *random decision forest* (Ho, 1998). Since the trees in the forest are trained independently, the generalization capabilities can be raised without sacrificing the accuracy. The individual trees classification results are combined, for instance, by a majority vote (Breiman, 2001; Friedman et al., 2001). With an increasing number of trees in the forest very high accuracy results can be achieved (Ho, 1998).

Support vector machines (SVMs) is a type of linear classifier, meaning that it tries to divide the input space with a hyperplane. This creates a binary classification, with the hyperplane as the decision boundary. The defining property of SVMs is, that they create this hyperplane such that it maximizes the distance between the two classes.

To avoid the linear restrictions of SVMs, the input space can be mapped to a higher-dimensional feature space through the 'kernel trick'. There are also several methods like 'one-versus-all' and 'one-versus-one' to allow multiple classes by splitting up the problem (Boser et al., 1992; Bishop, 2006).

Nearest neighbor classifiers are the only entirely model-free classifier in our selection. This means that they do not assume an underlying structure. Instead, they retain the entire training data as it was provided to them. New inputs are then categorized by finding the closest known data points, called the 'nearest neighbors'. The classifier votes among the selected neighbors and then outputs the winning class (Fix and Hodges Jr, 1951; Bishop, 2006).

2.4 Game Descriptions

This section gives a brief explanation of the games that we chose to focus on, namely *Space Invaders*, *Bowling*, and *Private Eye*. A basic understanding of the game mechanics is required to understand some of the examples throughout this paper.

Table 1: List of possible actions in ALE

noop (0)	fire (1)		
up (2)	right (3)	left (4)	down (5)
up-right (6)	up-left (7)	down-right (8)	down-left (9)
up-fire (10)	right-fire (11)	left-fire (12)	down-fire (13)
up-right-fire (14)	up-left-fire (15)	down-right-fire (16)	down-left-fire (17)
reset* (40)			

Space Invaders is a vertical shooter with the goal of defeating slowly approaching waves of alien space-ships. These waves move from one side to the other before moving down a single step. The controls are very simple, consisting only of moving left or right and shooting. In the early stages of the game three shields can be used as protection against the shots emitted by the grid of enemies. If the player is hit three times or an enemy reaches the bottom of the screen the game is over. Occasionally, a 'mothership' appears at the top of the screen. Defeating it gives a significant point bonus.

Bowling is an extremely simple video game version of the popular sport. The game is divided into position-ing, essentially moving your character up and down to align with your target, and ball guiding, where you influence the balls curve on its way down the lane. This influence defines the curve of the balls trajectory. Other than that, the standard rules of bowling apply.

Private Eye is an adventure game that revolves around finding items and returning them to a specific location on the map. The paths are filled with obsta-cles like potholes, animals and villains. In addition, there is a time limit of three minutes to collect and return all items. Movement is restricted to left, right and occasional up commands as well as jumping.

The three games represent different types of arcade games so that we can examine the feasibility of our approach for these different types. For a more detailed discussion on why each game was chosen we refer to Section 5.2.2. Screenshots of each of the three games can be found in Figure 1.

3 RELATED WORK

We discuss related work with regard to agent de-sign and feature learning.

3.1 Agents

A wide range of agents have been written for ALE with varying success. The most prevalent type are learning-based. Among the most advanced are Hausknecht's HyperNEAT agent (Hausknecht et al., 2012) and Mnih's deep learning approach (Mnih et al., 2013). Nair's massively parallel implementation of Mnih's ap-proach (Nair et al., 2015) is likely the most successful learning-based agent to date. It achieves superhuman scores in most games.

The most consistent performance however is deliv-ered by Lipovetzky's planning-based agents (Lipovet-zky et al., 2015). While there are comparably few planners targeted at ALE, Lipovetzky's iterated width (IW) and two-queue best-first search (2BFS) beat their learning-based rivals in most games.

This result is likely due to the fact, that action and reward are temporally divided in most games. While planning manages to solve this problem for small gaps, it still performs horribly in adventure games like Mon-tezumas Revenge and Private Eye, where rewards are offset by several seconds. A discussion of why deep learning does not work well in these games can be found in (Mnih et al., 2015).

3.2 Feature Learning

Feature learning describes the idea of distributing a classification task among several classifiers that are organized in a hierarchical structure. Each classifier is trained to recognize a specific feature.

While the field of feature learning is still young, there are already encouraging results. For one, Coates (Coates and Ng, 2012) achieved some of the highest current scores on the CIFAR-10 dataset with a k-means feature learning structure. A broader comparison of different methods can be found in the results of a fea-ture learning competition created by Google (Sculley et al., 2011). The challenge revolved around learning at most 100 features from 5.000 labeled examples of malicious URLs that could then be used for supervised prediction. While a lot of methods achieved results that essentially solved the problem, the pack was led by three different structures of random trees.

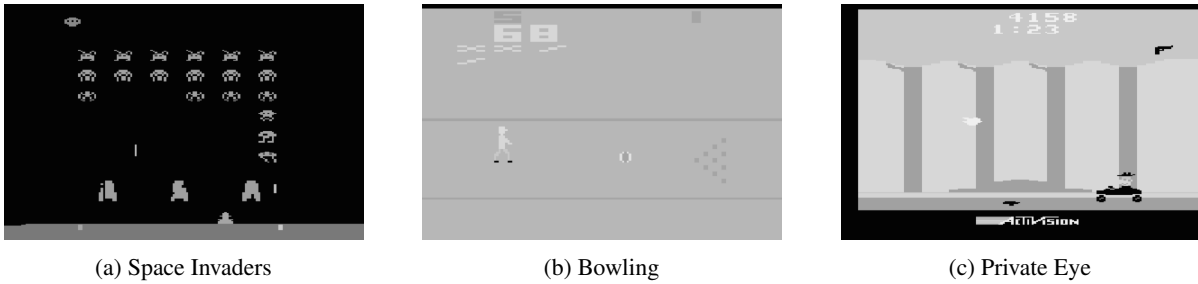(a) Space Invaders   (b) Bowling   (c) Private Eye

Figure 1: Screenshots from three games available in the Arcade Learning Environment.
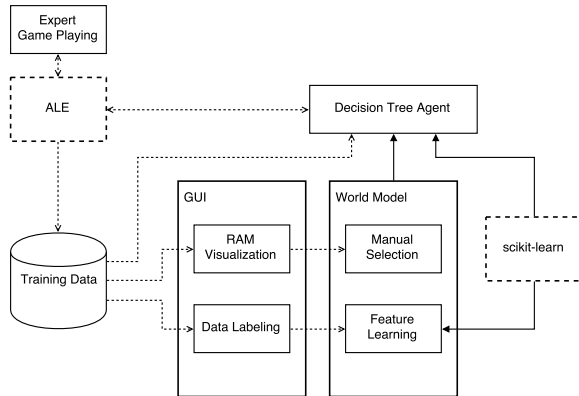

Figure 2: Structural overview of the approach. Dashed lines represent data flow, solid ones express a component relationship

4 APPROACH

As mentioned in the introduction, the main focus of this work is to involve human experts in producing relevant data for agent development. This section outlines our approach to this challenge.

4.1 Overview

The general idea is to use a human expert with knowledge of a given game to create training data and an appropriate world model of the game. We then create a game playing agent based on this information. A visual overview of the approach is given in Figure 2.

Before any analysis can occur, we need to **collect game data**. These usually consist of RAM states, screen images and the corresponding inputs of the player. In a fully automated setup, this could be achieved by using another agent, or even just randomly choosing actions. There are however two important benefits to using a human expert. First, we can ensure a much more efficient exploration of the game at hand. Situations that never occur unless the player fulfills specific requirements are abundant. Some of these require significant skill that an automatic approach might

not provide. Second, the expert can intentionally create training data that is relevant in fewer time. While playing the game normally should, in theory, provide all necessary data, it can be hard to find sufficient quantities of specific situations.

Once we have collected enough relevant data, we can begin the process of **feature selection**. We do this through means of manual selection and machine learning. It should be noted, that our approach is entirely based on RAM data. While image data could be included, we are exploring the notion, that all relevant data is contained in the RAM.

For the **manual analysis**, the expert chooses a few states that share a feature value, like the number of enemies on screen. This selection is done based on the corresponding game images. The toolkit then indicates the RAM sections that change the least among the chosen states, leaving out unremarkable consistencies. The idea behind this method is to highlight sections that likely encode the feature we are looking for.

Because only basic features of a game can be easily identified in the RAM, we also implement the more general approach of **feature learning**. Instead of reading values directly from the RAM, we train classifiers with supervised learning to create high-level features. In this case, the expert simply selects suitable example states by looking at the corresponding images and then labels them by hand.

The last step of our workflow is **autonomous play**. The agent design is intentionally kept simple. To focus on feature quality and the advantages of a precise world model, we use a decision tree agent. Observing an agent play can help in finding problems of the underlying world model.

4.2 Assisted RAM Analysis

Because it would not be feasible to extract RAM portions that correspond to specific features by simply looking at the raw data, we implement methods to highlight likely byte candidates. This section describes these methods and gives a brief introduction into the related parts of our toolkit interface.
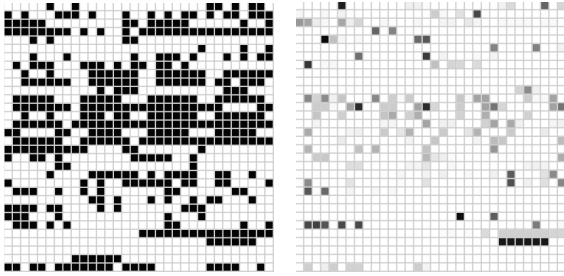
Figure 3: Raw and shaded RAM data as 32 by 32 grid

4.2.1 Visualizing Relevance

The central aspect of assisting the manual selection is visualization. In a first step, this means displaying the RAM data in a pattern oriented fashion. Comparing sets with 128 byte values each is quite demanding and quickly becomes confusing. To work around this limitation, we display these values as bits in a 32 by 32 black and white grid. As a result, an expert can simply regard them as patterns.

But we can do more. By selecting RAM states that share a given feature value, say the player stays at the same spot, we can introduce some statistical assistance. Since the states share a value, we want to highlight the RAM sections that are the same in all of them.

$$rel_p(S) = \frac{\sum_{s \in S} v_{p,s}}{|S|} \qquad (1)$$

Equation 1 does this by applying a bitwise calculation of the mean across the selected states given as set S. The bit position is indicated by p, so that rel_p is the relevance of the bit at position p and $v_{p,s}$ is the value of the bit in state s at position p.

We also find, that most of the RAM very rarely changes throughout the entire game. Most states that are highlighted are not uniquely consistent among the selected states, but are consistent across all recorded data. To counter this effect, we modify our approach to remove all unremarkable similarities.

$$\widetilde{rel}_p(S) = max(0, rel_p(S) - rel_p(R)) \qquad (2)$$

Equation 2 removes states that rarely ever change by simply subtracting the relevance across all recorded states, given as set R, from the relevance across the selected states. To preserve the property, that $\widetilde{rel}_p \in [0, 1]$, we also cut off all negative results. These negative values indicate that a bit is less consistent in the selected set. This is of no particular interest to us and breaks the possibility of using the result as a simple scalar later on.

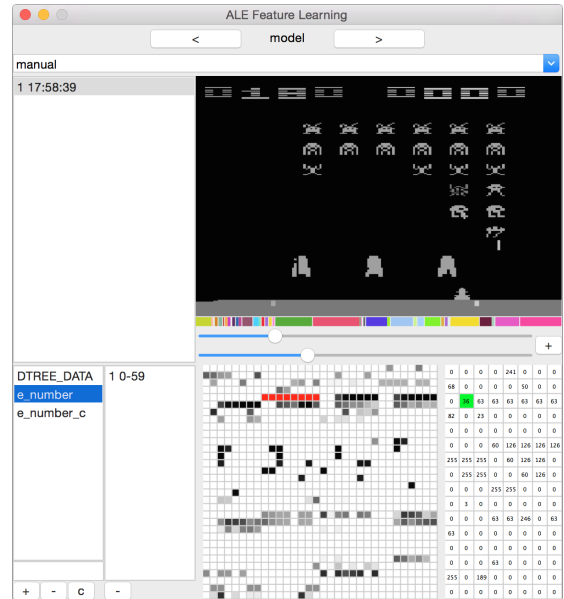Using this measure of relevance we can shade the grid to better reflect how important specific parts of

Figure 4: Visualization a candidate region in the RAM for the container 'e_number'. The colors below the game-screen indicate the changing value. The green mark in the lower right window indicates the corresponding numeric value of the RAM region.

the pattern seem to be. An example application can be seen in Figure 3. It should be noted, that the relevance rating is slightly modified by applying an exponent to increase visual distinction.

4.2.2 Sample Procedure

To give an impression of the manual feature engineering, let us consider the game *Space Invaders*. One of the descriptive attributes that the domain expert might identify is the number of enemies. An expert user can create a container for this attribute and associate recorded game data with certain values of that attribute. The interface to record data and label them is shown in Figure 4. The image shows selecting a candidate for a RAM portion encoding the number of enemies (modelled as 'e_number').

By browsing through the different sets of frames with the same value, the user might spot similarities in the RAM. This allows for identifying the portions of RAM encoding a particular feature. The mapping of state of a region in the RAM and the value of an attribute can then be used a feature.

4.3 Feature Learning

In this section, we will present and discuss our choices of feature classifiers and demonstrate how the toolkit

can be used to efficiently construct world models with feature learning.

4.3.1 Methods

To be effective, classifiers do not only have to be precise, but also fast, since the games are being played in real-time. We decided to use the following three classifiers to cover a spectrum of possible advantages.

Random trees perform well with the inclusion of irrelevant features. Every decision tree gets a random subset of features as its input. Features that are good indicators will dominate the decision process in almost any tree they are included in. Trees that lack these features should create predictions that are essentially randomly distributed. As a result, the ensuing majority vote is heavily influenced by relevant features. This property is very desirable in our application, since the systems RAM is likely to only include a few features that are truly decent indicators.

Support vector machines deal very well with high-dimensional data. This property can be observed in some upper bounds of the classification error for SVMs. Equation 3 shows one such upper bound, discussed by Vapnik (Vapnik and Chapelle, 2000).

$$E p_{error}^{l-1} \leq E\left(\frac{SD}{l\rho^2}\right) \quad (3)$$

Here, S denotes the span of all support vectors (the intersection of all sub-spaces containing the set of support vectors), D is the diameter of the smallest sphere containing all training points, ρ is the margin and l is the size of the training set. Most importantly, the upper bound is independent of the dimensionality of the problem. To us, this property could matter, since we are dealing with 128-dimensional data if we take the full RAM content as our input.

Nearest neighbor classifiers are suitable for problems that have arbitrarily irregular decision boundaries. Instead of calculating an explicit model, they simply vote on an input's class among a few of the nearest existing data points. We use this method as an indicator of how well explicit models can be formed, since we have little information about the decision boundaries. If the typically subpar nearest neighbor classifier surpasses the more advanced methods, there are likely to be problems that relate to the underlying models.

Our selection of the three classifiers described above is also supported by a comparative study in (Fernández-Delgado et al., 2014). Fernandez-Delgado
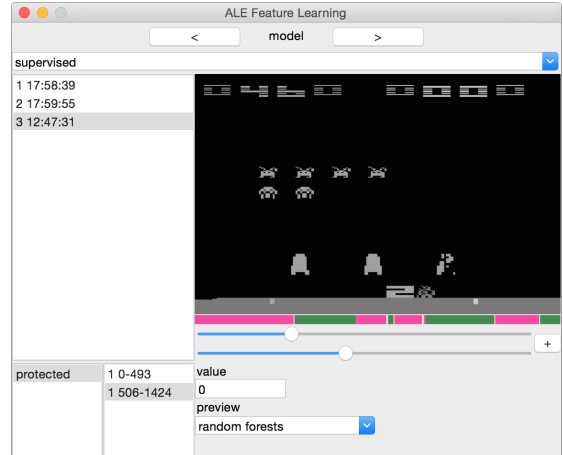

Figure 5: Feature learning procedure example: Visualizing classification results by a random forest on test data.

tested 179 classifiers on 121 datasets and found random trees to be the overall best, closely followed by support vector machines.

4.3.2 Sample Procedure

Similarly to the manual feature construction we also give a brief example of feature learning. Just like before, we can create a container for an attribute and label game situations with a value. In our example, let us consider the *Space Invaders* game again. This time, we are interested in whether our agent is protected by a shield. After collecting a set of training data, we can select a learning method and preview the performance of the same.

Figure 5 shows a situation in feature learning for the 'protected' attribute. The user can slide through different frames and check whether or not the classification is correct. The green and red areas in the bar below the game-screen indicate value false and true for 'protected', respectively.

4.4 Agent Design

We use a hierarchical structure to construct our agent. In this structure, classifiers identify individual features, which are then routed into higher-level classifiers. We first model a game world and then learn to play based on the resulting features. Figure 6 illustrates the hierarchy with the features used for the game Bowling. The left side utilizes manually selected features (see Section 4.2), while the right side uses feature learning to identify the features classes.

It is, of course, possible to create more classification layers, but because we were able to achieve very good results with a comparably flat structure there was
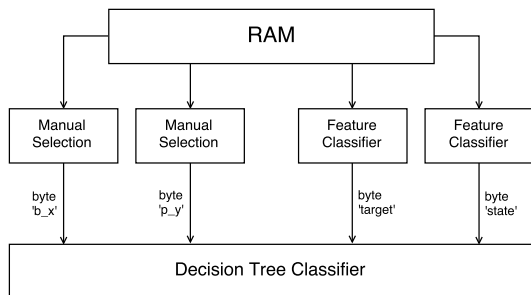
Figure 6: Bowling agent with feature learning structure. The classifiers feature vector includes both manually selected and learned features

no need for a more complex hierarchy.

4.4.1 Different Feature Sets

In order to measure the effects of an improved world model, we need to create an agent that uses it. In our case, we have a clear focus on the quality of features and therefore need an agent that is simple and transparent. Decision trees fit perfectly, because their structure can quickly be analyzed and carries obvious meaning, since each node corresponds to a specific feature.

Our baseline agents use the entire RAM as their input. The raw data is simply formatted into byte chunks and then routed into the decision tree classifier. Agents based on manual selection reduce this full byte array to only a few previously marked sections before routing the input into the decision tree classifier. For the feature learning approach, we apply the previously trained feature classifiers to the entire RAM, outputting a single byte that corresponds to the predicted class. This byte is then added to the feature vector of manually selected bytes.

5 EVALUATION

The evaluation comprises two main points. The first point is assessing the performance of the feature learning in ALE. We compare their scores and behavior across several features in all tested games. The second point is assessing the game playing performance achieved by our approach. This includes comparisons to both simple and state-of-the-art methods.

5.1 Learning Methods

In this section we compare the results of three classifiers in the context of ALE feature learning. We also

Table 2: Feature learning results based on their F_1 score. 'dnf' marks training processes that did not finish. Random trees delivers the best results across all features

	Space Invaders		Bowling		Private Eye	
	shot	bottom	target	state	object	clue
RT	0.968	0.997	1.000	0.968	1.000	0.998
SVM	dnf	0.993	1.000	0.963	0.999	0.991
NN	0.910	0.946	1.000	0.959	0.983	0.980

Figure 7: Visual comparison of classification results. The white lines indicate quickly changing classes. *From top to bottom: random trees, support vector machines, nearest neighbor*

introduce the methodology that was used in training and testing these classifiers.

5.1.1 Methodology

To measure the performance of each classifier used in feature and agent learning, both their precision and recall are calculated. Since these measures never differed by more than one percent in our tests, we decided to reduce them to their F_1 score. The F_1 score is the weighted average of precision and recall. Cross-validation is performed with randomly split training and test data sets of equal size. A grid search with a range of hand selected values is used to estimate appropriate parameters for all classifiers.

Each feature is trained with around 2000 frames of data and the resulting scores are averages of six runs for each configuration.

5.1.2 Results

Looking at the feature learning results in Table 2, the clear winner by scores is the random trees classifier. It equals or surpasses the alternatives in all tests and delivers great results of above 96% precision and recall.

But the scores do not tell the entire picture. In addition to achieving the highest scores, random trees are also the fastest to train and the fastest to evaluate. While the differences are usually marginal, they grow significantly with larger amounts of training data. Especially support vector machines struggle in this regard, as too much data causes excessive training times (the process was stopped after three hours of continuous computation). Since the scikit-learn implementation that is used in this paper is based on the very popular LIBSVM (Chang and Lin, 2011), it is highly unlikely to be the result of a coding error.

A different problem arises with the nearest neighbor classifier. While the overall scores indicate that its performance is stellar, the distribution of misclassifications is far more erratic. An example of this can be seen in Figure 7, where red and blue indicate two different classes and white represents high-frequency class changes. While slight errors around the optimal decision boundaries are likely inconsequential, stray misclassifications could significantly impact play performance.

5.2 Play Performance

Towards the overall goal of creating knowledge-based agents, the playing performance is important. In the following section we introduce the methodology that was used in testing the performance of our agents. We then analyze the results for each of the three tested games.

5.2.1 Methodology

The basis of all agents used in this paper are decision trees that are trained with data collected by human experts. In order to get a rough understanding of how much training data is required to adequately train the classifier, we ran tests with an increasing number of training frames. While around 3000 frames deliver some of the best results, it became quite apparent that this is due to a sort of beneficial overfitting. The agent is mimicking the experts example game very closely. Increasing the number of frames to around 10000 yields worse results, but also shows first signs of abstraction. The amount we settled on, around 20000 frames, delivers considerably better results, while also increasing the agents capability of handling new situations. Beyond that point gains were minimal.

Our underlying learning method differs from the, usually image-based, reinforcement learning that is used by most other researchers. This is why we decided to measure three configurations that allow for a self-contained comparison: The first one simply divides the RAM into 128 bytes and it is referred to with *full ram* in the tables below. The second one uses only manually selected parts of the RAM and it is referred to with *manually*. The third one adds learned features to the manually selected ones and it is referred to with *learned*. Because random trees dominate our classifier comparison, it is the only feature classifier we use from here on out. In addition to our own reference values, we include the closely related score of Bellemare's RAM approach (Bellemare et al., 2013) (referred to with *bellemare*), as well as the highest score we could find in related papers (referred to with *state-of-the-art*).

Table 3: Space Invaders F_1 and game scores over ten episodes

full ram										
0.59	0.59	0.61	0.56	0.63	0.61	0.63	0.64	0.61	0.60	**0.607**
310	260	160	180	445	130	255	70	230	115	**215.5**
manual selection										
0.76	0.75	0.75	0.75	0.74	0.73	0.75	0.75	0.74	0.75	**0.747**
305	370	275	235	320	185	785	175	295	320	**326.5**
learned										
0.70	0.73	0.74	0.73	0.73	0.72	0.72	0.73	0.72	0.73	**0.725**
210	555	230	90	510	285	505	315	230	430	**336.0**

Table 4: Space Invaders game score comparison

full ram	manual	learned	bellemare	state-of-the-art
215.5	326.5	336.0	226.5	**3974.0**

Each of our own configurations is trained and then run ten times per game. In addition to the scores reported by ALE, we also include the F_1 score for each decision tree. The latter is however not necessarily related to game performance, since the input data does not represent perfect play.

5.2.2 Results

We now look at the resulting gameplay performance for the three games *Space Invaders*, *Bowling*, and *Private Eye* introduced earlier. For each game, we include a short rationale for its inclusion.

Space Invaders is included in our game selection because it is a great example of the arcade-style gaming that was dominant on the Atari 2600. It is mainly reactionary and open ended. There are no complicated objectives and the entire necessary information is accessible at all times. *Manually selected features* include the x position of the player, x/y for the enemies and x/y for two shots, as well as the current number of enemies. The two *learned features* recognize danger from incoming shots and enemies reaching the bottom.

Table 3 shows that there are significant benefits to manually selecting relevant features. Not only does the classifier achieve a much higher F_1 score, but the play performance also increases by around 50%. The gains achieved by adding learned features are much more modest, with only a marginal increase of around 3% in the game score. The observed play style is however changed. The agent avoids enemies reaching the bottom more effectively. It thereby manages to exceed 400 points in four out of ten games, while manual selection only does so in one.

As can be seen in Table 4, our best score exceeds

Table 5: Bowling F_1 and game scores over ten episodes

full ram

0.94	0.93	0.93	0.93	0.94	0.93	0.94	0.93	0.94	0.94	**0.935**
dnf	dnf	111	127	177	102	dnf	dnf	99	84	**116.7**

manual selection

0.87	0.87	0.87	0.87	0.87	0.87	0.87	0.87	0.87	0.87	**0.870**
80	80	80	80	80	80	80	80	80	80	**80.0**

learned

0.90	0.90	0.89	0.90	0.89	0.91	0.90	0.90	0.90	0.90	**0.899**
217	190	217	219	174	218	198	216	219	219	**208.7**

Table 6: Bowling game score comparison

full ram	manual	learned	bellemare	state-of-the-art
116.7	80.0	**208.7**	29.3	69.0

Table 7: Private Eye F_1 and game scores over ten episodes

full ram

0.93	0.93	0.93	0.92	0.92	0.92	0.92	0.93	0.92	0.92	**0.924**
-882	26113	3377	26183	3463	3536	-1000	4071	-342	13738	**7825.7**

manual selection

0.89	0.90	0.89	0.90	0.89	0.89	0.88	0.89	0.89	0.90	**0.892**
26352	49761	-1000	18692	10647	25449	25868	3300	-1000	4532	**16260.1**

learned

0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	**0.850**
96960	96356	96456	96456	96156	96056	96556	96456	96556	95760	**96376.8**

Table 8: Private Eye game score comparison

full ram	manual	learned	bellemare	state-of-the-art
7825.7	16260.1	**96376.8**	111.9	2544.0

Bellemare's approach by around 50%. However, it falls very short of the current state-of-the-art.

Bowling stands out among the available games in ALE. Even though its premise and game mechanics are incredibly simple, no agent has so far achieved results that approach even a human layman. It is therefore interesting to see if human involvement in the modeling and training stages of an agent can remedy the underlying issues. The *manual features* that were easily identified in the RAM are the ball's x and the player's y position. Notably, the remaining pins proved too hard to isolate and were therefore not manually selected. To compensate, the *learned features* include the current target alongside a state feature to identify waiting periods.

The advantages of collecting training samples with an expert are immediately visible in Bowling. The full RAM approach beats Bellemare's control value by 300% and even outclasses the state-of-the art approach, IW(1) by Lipovetsky, by around 70%. However, the agent tends to get stuck waiting and fails to finish the game four out of ten times. The sparse manual selection fails to improve upon the full RAM input. Both the F_1 and the game score drop significantly. Adding learned features makes a world of difference. The additional context awareness prevents waiting loops and being able to identify the current target allows the agent to play a near perfect games. While most throws result in strikes, even the occasional spare is handled gracefully. All of this results in scores that trump the current best by more than 200%.

Looking at the comparison scores in Table 6, it is unclear why an extremely simple game like Bowling proves too hard for even the best planning algorithms. One possible explanation might be the significant delay between input and scoring that results from the ball traveling down the lane.

Private Eye is easily the most ambitious of the selected games. Its large world is only partially observable at any given time and solving the changing objectives requires many steps. Unsurprisingly, even the best agents fail to achieve any notable progress. Through our custom process of *manual feature* selection, we are able to isolate the player's x position, the current screen number and the id of the currently held item, as well as the current time. The first, and most important, *learned feature* is the current 'objective'. It encodes the target destination as a number from one to seven. The second learned feature indicates whether the current room still contains a 'clue'.

Table 7 shows a significantly increased score due to using expert training data. The initial score on the entire RAM is 70 times as high as the comparison value from Bellemare. It still even triples the current state-of-the-art, 2BFS by Lipovetsky.

Manual selection doubles the score of our own full RAM approach, but still manages to get lost on the way. This is shown by the low -1000 scores in two of the ten runs. In these instances, the agent gets stuck in a dead end and is eventually terminated by the running clock. The 'objective' feature solves this problem, with all runs resulting in a score of above 95000, or 37 times the current state-of-the-art (see Table 8). This score is also very close to the theoretical maximum of 101600 and includes finding and returning all items.

Where Bowling has a somewhat noticeable delay between action and reward, Private Eye separates these events by seconds or even minutes. It is therefore no surprise, that planning-based approaches fail to achieve high marks. The same applies to reinforcement learning, as the sparse rewards provide little guidance as to what constitutes a beneficial action. An expert's domain knowledge solves these problems very efficiently and allows for scores at a human level.
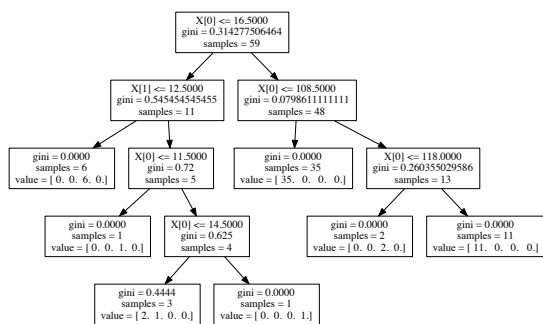
Figure 8: Simple Bowling agent. The structure automates bowling a strike.

5.3 Further Investigation

In this section we discuss some additional work that does not fit within the main evaluation of the paper. These are mainly extensions of our approaches and could serve as starting points for future work.

5.3.1 Clustering

In addition to supervised feature learning we tested an unsupervised approach based on a few popular clustering methods (namely k-means, mean shift and DB-SCAN). We did this to explore whether it would be possible to fully automate constructing world models within our toolkit. However, during our limited tests we were not able to create meaningful features through this process. Visualizing the results, we found, that clustering tends to create equally sized classes that are defined by the temporal proximity of their members.

Because unsupervised feature learning was not a core goal of this paper, we decided to stop our investigation at this point. Future research might focus on advancing these preliminary tests by isolating notable RAM sections.

5.3.2 Simplified Agents

One overarching idea behind this paper is to create agents that require less computational resources by leveraging additional domain knowledge. We tried to push the boundaries of expert involvement within our toolkit by selecting specific moments of perfect play to create a minimal agent. The ideal candidate for this approach is Bowling, because a 'strike' is always the best possible outcome and examples can easily be identified. Other games, such as Space Invaders are a lot more complicated, making it virtually impossible to choose perfect examples.

Choosing only one example sequence of a perfect throw created the decision tree shown in Figure 8. The only two features that are used are the player's x and the ball's y position. To give a point of comparison, the full RAM agents in our main evaluation of Bowling have around 300 nodes, while this simplified version only has 13. In addition, the score is much improved. The decision tree in Figure 8 achieves a score of 230, surpassing even our previous best agents with a near-perfect game.

While these results are interesting, they should be treated carefully. This reduced agent is essentially just a 'strike' automation. It can only deal with ideal conditions and has no capability of identifying and managing varying situations. Nevertheless, it is interesting to see, that a little additional domain knowledge can create an extremely simple agent that performs nearly perfectly.

6 CONCLUSION

In this paper we presented a toolkit for assisting developers of knowledge-based agents with feature engineering and feature learning. Our application domain are arcade games where we took three exemplary games from the Arcade Learning Environment. We provide assistance in creating features by two mean. For one, we assist in manually creating features by helping to identify relevant portions of the RAM. For another, we offer means to learn features from domain expert knowledge. We then also help to create simple agents for the games using the developed features. In our evaluation, we first compare different learning techniques for the feature learning. Then we assess the game playing performance of different agents using different (sub)sets of features available. We show that while the knowledge-based approach cannot keep up with state-of-the-art approaches using deep learning in rather reactive kind of games it can clearly outperform those methods in games that have a large delay between actions and their final effect.

In our research we found that there are significant gains in all tested games when preselecting features. Comparing the naive full RAM approach to a carefully crafted world model with everything else being equal, we saw scores increase by anywhere from 50% to 1100%. While these scores even surpass state-of-the-art methods in two out of three games, the effect cannot be entirely attributed to world modeling. The high-quality training data recorded by an expert is clearly also a notable factor, as even the naïve approach often outclasses the current best methods. Judging from the results, our toolkit can quickly create world mod-

els that meaningfully impact the overall game score. While it lacks the generality of game independent approaches, it shows that minimal expert involvement can enable even simple game playing agents to perform very well.

Because in this paper we focused on the feature extraction, we investigated the benefits on a limited set of agents only. Future work might revolve around applying our findings to a broader range of agents and games. Another interesting addition could be to further automate the manual feature selection. The methods presented in this paper often reduce the possible byte candidates to just a handful and it might be feasible to reduce the number even more. If nothing else, this work should demonstrate that feature quality can have a very meaningful impact in ALE. Taking this into account, it would be interesting to investigate how fully automatic dimensionality reduction methods can influence game playing performance for different agents.

Overall, we have shown that enabling descriptive features to build knowledge-based agents is a very promising route. It yields agents that are not only comprehensible but that are also able to outperform state-of-the-art solutions in difficult situations.

REFERENCES

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.

Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM.

Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

Chang, C.-C. and Lin, C.-J. (2011). Libsvm: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3):27:1–27:27.

Coates, A. and Ng, A. Y. (2012). Learning feature representations with k-means. In *Neural Networks: Tricks of the Trade*, pages 561–580. Springer.

Fernández-Delgado, M., Cernadas, E., Barro, S., and Amorim, D. (2014). Do we need hundreds of classifiers to solve real world classification problems? *The Journal of Machine Learning Research*, 15(1):3133–3181.

Fix, E. and Hodges Jr, J. L. (1951). Discriminatory analysis-nonparametric discrimination: consistency properties. Technical report, DTIC Document.

Friedman, J., Hastie, T., and Tibshirani, R. (2001). *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin.

Hausknecht, M., Khandelwal, P., Miikkulainen, R., and Stone, P. (2012). Hyperneat-ggp: A hyperneat-based atari general game player. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 217–224.

Ho, T. K. (1998). The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844.

Lipovetzky, N., Ramirez, M., and Geffner, H. (2015). Classical planning with simulators: Results on the atari video games. *Proc. IJCAI 2015*.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *NIPS Deep Learning Workshop 2013*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Naddaf, Y. (2010). *Game-independent ai agents for playing atari 2600 console games*. University of Alberta.

Nair, A., Srinivasan, P., and Blackwell, S. (2015). Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830.

Sculley, D. et al. (2011). Results from a semi-supervised feature learning competition. *NIPS 2011 Workshop on Deep Learning and Unsupervised Feature Learning*.

Vapnik, V. and Chapelle, O. (2000). Bounds on error expectation for support vector machines. *Neural computation*, 12(9):2013–2036.