An Evolutionary Feature Discovery Method for Reinforcement Learning

Reza Mahjourian
Department of Computer Science
University of Texas at Austin
reza@cs.utexas.edu

Peter Stone
Department of Computer Science
University of Texas at Austin
pstone@cs.utexas.edu

ABSTRACT

Using linear methods for reinforcement learning problems requires designing efficient features. However, designing features often requires having ample knowledge about the problem domain. When dealing with complex problem domains, coming up with efficient feature sets often requires a trial and error process which can prove difficult or inefficient. We present an evolutionary algorithm for generating and evaluating candidate feature sets for learning a task using gradient descent Sarsa(λ) as a linear method. Our evaluations on three different problem domains show that our solution is effective.

1. INTRODUCTION

Most real-world reinforcement learning (RL) problems involve numerous state variables or state variables which are continuous. Such properties in state variables make application of tabular learning methods such as Q-learning impractical. For this reason, there are alternative learning methods which use function approximators to approximate the values of states or state-action pairs. Examples of widely-used function approximators are neural networks and linear architectures.

In order to employ function approximation, one needs to come up with a suitable design for a function approximator. Function approximators usually operate on features, which are high-level abstractions of important properties in the problem state. Designing suitable features for a problem requires sufficient domain knowledge. Often the designer has to try different sets of features and see if they perform well or not.

We present an evolutionary method for discovering effective features for learning a task using reinforcement learning. We focus our attention on discovering features which are suitable for linear architectures, because 1) linear architectures are easier to understand and analyze compared to some other types of learners like neural networks and 2) they are efficient in terms of speed of convergence and computational resources needed. The complexity of linear methods increases only linearly with the number of features used. However, one must come up with a set of features that would be suitable for learning the problem. In particular, in order for a linear function approximator to work well, the features need to be independent of each other as much as possible.

In this paper, we focus on problems with spatial state variables. However, we expect that with appropriate modifications this method can be extended to be applicable a wider range of state representations and feature types.

2. RELATED WORK

The work in [1] is one of the earliest publications which discuss the process of constructing features from lower level state variables via some set of transformations.

Most feature discovery solutions in the context of Reinforcement Learning try to create higher level features by construction basis functions [6], [2]. Most of this work is mathematically grounded and the features are less recognizable/understandable than those created by our method.

There are also feature discovery solutions that use genetic algorithms. An example is the work in [4]. These evolutionary methods typically use genetic algorithms to evolve features directly, rather than evolving agents who each have their own feature sets.

For example, Learning Classifier Systems like XCSF [11] operate over a space of binary features in the problem domain and employ genetic algorithms to learn a piece-wise linear approximation over the original state variables.

As another example, NEAT-Q [10] evolves neural network topologies that can learn a Reinforcement Learning task using Q-Learning. The features discovered by such a solution are implicitly represented in the network weights that are trained through back propagation based on the TD (Temporal Difference) errors.

3. THE PROBLEM DOMAINS

3.1 Knight Joust

The first domain that we use is Knight Joust [9]. The environment for the Knight Joust task is situated on a 25×25 grid. In this task, a player starts on the lowest row of the grid and an opponent starts on the highest row of the grid. The player and the opponent alternate moves. The player's goal is to reach the highest row without being captured by the opponent. In each time step, the player can choose from one of the following three actions: 1) **Forward**: The player moves one square north. 2) **Jump West**: The player moves one square north and two square west. 3) **Jump East**: The player moves one square north and two square east.

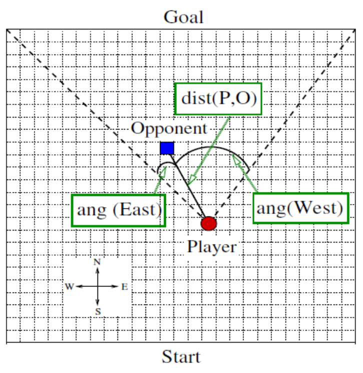

Figure 1: Hand-coded Features in Knight Joust (from [9])

The player receives a reward of +20 whenever it takes the Forward action and a reward of 0 whenever it performs a jump. Moreover, the player receives an additional reward of +20 when it reaches the highest row on the grid. The episode terminates when the player reaches the highest row or when it is captured by the opponent.

In each time step, the opponent can move one cell in each of the 8 possible directions. The opponent uses the fixed policy outlined in Figure 2.

```
if Opponent is East of Player:
    Move West with probability 0.9
else if Opponent is West of Player:
    Move East with probability 0.9

if Opponent is North of Player:
    Move South with probability 1.0
else if Opponent is South of Player:
    Move North with probability 0.8
```

Figure 2: Knight Joust Opponent Policy

Figure 1 (taken from [9]) shows a sample state on the Knight Joust grid. To make the problem more challenging, we set the starting column for the player and the opponent randomly in each episode.

3.2 Simplified Single-Agent Keepaway Soccer Simulation

The Keepaway Soccer game [7] is a relatively complicated multi-agent learning problem. In 3×2 Keepaway, a group of three keepers try to maintain possession of a ball in a $25m \times 25m$ square field while two takers try to take the ball by approaching the keeper with the ball or by intercepting their passes. The keepers should learn to keep the ball for longer periods. Figure 3 shows a sample configuration of the original Keepaway field.

We designed a simplified version of this domain to study our algorithm on a domain with continuous state variables. In our version, the problem is treated as a single-agent task. The agent is aware which keeper has possession of the ball and can choose from one of the three available actions: a) keep the ball, b) pass the ball to keeper 2, and c) pass the ball to keeper 3. Passes take exactly one time step to complete and can succeed or fail probabilistically depending on the positions of the players.

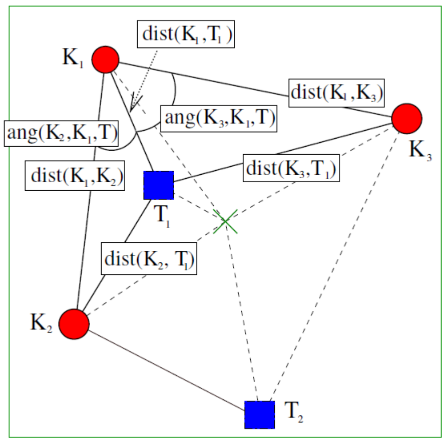

Figure 3: Hand-coded Features in the Original Keepaway Soccer Domain (from [9])

The probabilities with which a taker can steal from a keep action or intercept a pass action are determined by the following rules.

- $taker_1$ steals from a keep action with probability: $min(1/dist(keeper, taker_1), 1)$
- $taker_1$ intercepts a pass from $keeper_a$ to $keeper_b$ with probability: $min(1/(dist(taker_1, line_{a,b}) + 4.0), 1) + 0.1$

Similar probabilities apply to $taker_2$. Based on these probabilities, the optimal strategy for a keeper would be to try to keep the ball as long as no takers are close, and pass it before they get too close. Based on the probabilities, a pass is more likely to succeed if the takers are farther from the trajectory of the pass.

3.3 Mini-soccer

The third and last problem domain we study is the Mini-soccer domain, which is introduced in [5]. The original game is played in a 4×5 grid as shown in Figure 4 with two goalposts of size 2 cells. However, in our implementation we used a field size of 10×20 cells with goalposts of size 4 cells.

There are two players, A and B, which start at fixed positions on the grid on every episode. One of the players has possession of the ball. The objective for each player is to move the ball to the goal zone at the opposing side. Scoring a goal results in a reward of +1 at the end, and receiving a goal results in a reward of -1. In each step each players can choose one of the five available actions: North, South, East, West, and Stand. Once the players have selected their actions, their moves take place simultaneously. If the two players enter the same cell at the end of a move, the moves don't take place and possession of the ball is transferred to the other player.

Since this domain requires an opponent, we developed a simple agent with a fixed policy which can play against our learning agents.

4. BASELINE SOLUTIONS

In order to develop baselines for evaluation of our evolutionary feature discovery method, we first start by discussing some conventional solutions for learning these tasks.

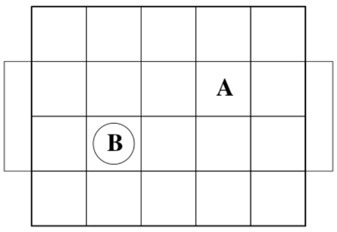

Figure 4: Initial configuration in Mini-soccer domain (from [5])

4.1 Tabular Sarsa(λ)

For the Knight Joust problem, one can learn the task using tabular Sarsa(λ) on the raw state variables. There are 25^4 potential states and, therefore, there would be $25^4 \times 3$ state-action-value entries. So using a tabular method to maintain the value estimates is feasible. Likewise, the number of states in the Mini-soccer game is small enough for a tabular method to be applicable. In the Simplified Keepaway Soccer problem the state variables are continuous and we can't use tabular methods, unless we employ tiling or some other form of state abstraction.

4.2 Linear Architectures with Gradient Descent Sarsa(λ)

Another solution for learning these tasks is using a linear method like Gradient Descent Sarsa(λ) [8] on a set of features.

For the Knight Joust problem, we used the following set of features, which were suggested by the original author of the problem: 1) **`Dist(P,O)`**: The distance between the player and the opponent. 2) **`Ang(West)`**: The angle between the lines connecting the player to the opponent and to the upper right corner of the grid. 3) **`Ang(East)`**: The angle between the lines connecting the player to the opponent and to the upper left corner of the grid. These features are shown in Figure 1 as well.

The solution discussed in [9] uses discretized values of the above distance and angle measurements to learn the task using tabular Sarsa(λ). However since we are interested in linear architectures, we use an encoding of the features similar to the solution for the Keepaway Soccer game given in [7]. We create 10 one-dimensional tilings of each feature to get 30 feature groups. Each tiling consists of 10 tiles. Also, each set of tilings are created using a different offset to provide a smooth coverage over the range of values. The features are encoded using a unary encoding. Therefore, our linear architecture consists of 300 binary features and there are 300 weights to be learned for each of the 3 possible actions.

For Simplified Keepaway Soccer, we learn the task using the 13 features defined in [7]. These features are shown in Figure 3. Since our simulation of the Keepaway Soccer game is very different from its original settings, we can't expect the same set of features to be optimal or even effective for learning this domain. On the other hand, they seem to be a reasonable set of features for learning this task, so we choose to use them as a representation of what an informed domain expert might come up with.

Similarly, for the Mini-soccer domain we hand designed the following set of features for learning the task: 1) Ball in possession flag 2) Distance between the agent and the opponent 3) Distance between the agent and center of the right goal 4) Distance between the agent and center of the left goal 5) Distance between the opponent and center of the right goal 6) Distance between the opponent and center of the left goal 7) Angle between the agent, opponent, and top right corner of the right goal 8) Angle between the agent, opponent, and bottom left corner of the left goal. Except for the first feature, which is a binary flag, each of the other features were discretized 10 times using different offset values to provide a smooth coverage over the range of possible values.

5. EVOLUTIONARY FEATURE DISCOVERY

Were we not given any suggestions on what features to use to learn the Knight Joust, Keepaway Soccer, and Mini-soccer tasks, how could we have come up with a set of features that would be efficient for learning these problems? Moreover, starting with a given set of features, how could we search for alternative and potentially more efficient features for these domains?

We are interested in an evolutionary method which can evolve a suitable set of features for learning a task using a linear architecture. There are two possible ways to approach this problem with an evolutionary method: The first approach is to evolve a set of features which would be efficient for learning the task from scratch. In other words, we would use evolution to discover a set of features that can learn the task more quickly from scratch. The second approach is to use an agent with a dynamic set of features, which continues to improve its performance by successively evolving its feature set.

The first option is akin to Darwinian evolution, while the second option is akin to Lamarckian evolution. Both of these approaches are viable and may have their own use cases. The first option may be more attractive if we are experimenting with a smaller version of a problem and hope to develop a set of features that can learn the original problem more efficiently. This option is also attractive from an analytical point of view, when we are interested in developing an understanding about the properties of a domain and the properties of the features that can be used for learning it.

Evaluating a Darwinian solution would be more time-consuming, since evaluating each candidate set of features requires testing it on an agent which has no prior learning experience in the domain. With the Lamarckian approach, however, a surviving agent wouldn't lose what it has already learned in the previous generations, and instead would continue its learning process with a modified set of features.

We chose to follow the second approach, as it is more suitable if we mainly care about learning the task as easily as possible. Moreover, a solution based on the second approach can be extended to develop an online solution, which would try to maximize the cumulative reward in every generation while still allowing for exploration [10].

5.1 The Algorithm Outline

Our evolutionary algorithm is based on the concept of "Featurizers." They are entities that can generate candidate features from the variables defined in the agent's state representation. The algorithm starts with an agent which has an empty feature set. Using the available featurizers, a Mutator creates mutated copies of the agent. The featurizers serve as the mutation operations available to the Mutator. Once the number of agents reaches the predefined population size, the agents in the population are trained in the environment for a predefined number of episodes. At the end of the training phase, the agents which receive the most reward are selected for survival and the other agents are discarded. In the subsequent generation, the Mutator again uses the available featurizers to mutate the feature sets of the surviving agents. Original unmutated versions of the the surviving agents are also kept in the next generation. The Mutator also copies over the trained weights from the original agents to the mutated agents.

5.2 State Representation

Even though we were mainly concerned with learning the three problems discussed earlier, we tried to create a framework which would be applicable to a wider range of problems and different kinds of featurizers. In particular, we designed a modular representation of the state. In this representation, the problem's state has to be encoded as a set of state components. For the Knight Joust problem and the Simplified Keepaway Soccer problem, the only type of state component that we needed was a two-dimensional point. For example, the state in the Knight Joust problem can be encoded as two points. For the Mini-soccer game, we needed two-dimensional points and a binary flag state component used to indicate whether the agent has possession of the ball or not. Some other state components that might be useful for other types of problem domains are: n-dimensional points, angles, and values from a fixed discrete set.

Depending on their type, the state components can be tagged with additional attributes. For example, points are tagged with ranges of the values that they can take. Also, points can be tagged as continuous or discrete. The values of these attributes would enable the featurizers to pick the appropriate types of components for creating candidate features.

5.3 Featurizers

A featurizer is an entity which can create candidate features from the set of components or variables in the state. Featurizers can also modify existing features. Since featurizers work only in terms of state components and current features, one can imagine a library of featurizers that can be used in different problem domains without any change. Each featurizer has some criteria for picking its input components. For example, some featurizer may only operate on state components with continuous values. Below is a list of featurizers that we have implemented so far.

- **Binary Flag Featurizer** (`Flag`) This feature is only relevant to state variables that are represented as a binary flag. It maps the binary-valued state variable to a feature with two cells. At any given point in time, one of these two cells is considered on, depending on the value of the state variable.
- **Point-Point Distance Featurizer** (`Dist`) Selects two points and creates a feature with a tile coding of the distance between them.
- **Point-Point Projected Distance Featurizer** (`Dist-Y` or `Dist-Y`) Similar to the previous featurizer, but computes the distance projected on one of the axes (for example X or Y.)
- **Angle Featurizer** (`Angle`) Selects three points and creates a feature with a tile coding of the angle between them.
- **Two-dimensional Point Featurizer** (`Point-XY`) Selects a non-stationary point and creates a feature based on a two-dimensional tile coding of it.
- **One-dimensional Point Featurizer** (`Point-X`) Selects a non-stationary point and creates a feature based on a one-dimensional tile coding of its x or y component.
- **Retile Featurizer** Selects an existing feature and creates a retiling of it using a random offset value.
- **Interaction Featurizer** (`Interaction`) Selects two existing features and creates a new feature by considering the possible combinations of the values for the two features. This featurizer can produce effective features in cases where the generated simple features are not independent and there are interactions between them in the problem domain.

Even though one can consider using every type of featurizer available in this library, in our implementation the Mutator accepts a list of desired featurizers as input. In addition, each selected featurizers is associated with a probability which specifies how often it should be tried during mutations. Section 8 discusses an alternative.

In addition to the components that come from the learner's state, the featurizers can also work with variables that are defined in the domain's environment. For example, in the Knight Joust problem, we considered the four corners of the board as four important points. Likewise, in the Simplified Keepaway Soccer game, the four corners and the center of the board were considered as important points. These extra components can either be defined explicitly in the environment, or be extracted using some heuristic methods.

5.4 Feature Set Costs

Our evolutionary algorithm can also factor in the computational costs of the candidate feature sets. For some problems, it might be desirable to discover feature sets that are not only efficient, but also light, in terms of the computational costs associated with training them. Different feature types generated by different featurizers typically have different computational costs. For example, the interaction featurizer and multi-dimensional tilings of points create more binary features compared to scalar distance and angle featurizers.

At the end of each generation, we discount the rewards received by feature sets based on their computational costs. To control how aggressively the algorithm tries to reduce the computational cost of the selected feature sets, we created a parameter named η. At the end of each generation, the rewards received by feature set f are multiplied by $\eta^{cost(f)/min(cost)}$, where $cost(f)$ represents the computational cost of feature set f and $min(cost)$ represents the cost of the lightest feature set in the population. A values of 1.0 for η amounts to ignoring the computational costs. Our experiments showed that choosing a value around 0.98 is enough to make the algorithm select lighter feature sets.

To estimate the cost of the feature sets, we first tried regressing an analytical model which would estimate the cost of a feature set from the types of features present in it. However, this proved to be difficult, since feature types performed differently across different problem domains. So, instead, we switched to recording and using the actual CPU seconds that the feature set uses during training and use that as the value of $cost(f)$.

In order to focus on the core behavior of the algorithm, for all the experiments reported in this paper we set the value of η to 1.0, making the algorithm try to optimize rewards and ignore the costs of the feature sets.

6. THE EXPERIMENT SETUP

We first ran the baseline implementations for all three domains. There are two implementations for the Knight Joust problem, one implementation for the Simplified Keepaway Soccer, and one for the Mini-soccer problem. For all baseline implementations we ran the programs for 20 of trials and averaged the results over those trials. In addition, out of those 20 trials, we selected the trial that produced the highest average rewards and extracted its final policy as the best policy that the hand-coded feature sets could produce. This policy was in turn evaluated over 3000 problems for 20 trials.

Then, we ran the evolutionary algorithm once for each of the three domains. During each generation, we recorded the rewards received by the champion of the generation. We also recorded the average rewards received by all the population in each generation. Once the algorithm terminated, we selected the generation that had produced the highest average rewards and extracted its final policy as the best policy that the evolutionary algorithm could produce. Similarly to the hand-coded agents, this policy was in turn evaluated over 3000 problems for 20 trials.

6.1 Basic Parameters

In all experiments, we set the value of λ to 0.95, ϵ to 0.05, and α to 0.1. The values of ϵ and α are kept constant and do not change over time. This choice was mainly made because we wanted to keep the parameters for the baseline implementations and the evolutionary implementations exactly the same at all times. In the Knight Joust and Simplified Keepaway Soccer domains, we set the value of γ, the discount factor in the environment, to 1.0. For the Mini-soccer domain, we set γ to 0.95 to encourage the agents to score more quickly.

In all implementations, we used replacing eligibility traces. The eligibility trace of the current state-action pair, or the trace for the present features and selected action are set to one. Also, the eligibility trace for all the other actions over the current state, or the present features, are set to zero.

For all three problem domains we used 15 generations with a population size of 100. Each generation was trained for 200 episodes.

Also, to make the comparison of evolving feature sets more reliable with fewer training episodes, we train all the agents in the population on the same set of starting states, which are generated randomly at the start of the generation.

In addition, we generate a set of seed values for the random number generator at the start of each generation and use the same seed values for the corresponding episodes for all the agents [3].

6.2 Initializing Weights

Even though in the Sarsa(λ) algorithm the initial weights could be set arbitrarily, we preferred to set the initial weights optimistically to speed up exploration in the early phases. Likewise, for tabular Sarsa(λ), we set the initial Q-values optimistically.

For the evolutionary algorithm, we experimented with different strategies for initializing weights. In the very first generation, where all the agents have only one feature group in their feature set, we set the weight optimistically.

As discussed earlier, at the end of every generation successful feature sets which survive are selected for mutation, whereby they are augmented with newly generated features. We tried three main strategies for setting the weights on the newly added features. The first strategy was to set the weights optimistically regardless of the current weights associated with the existing feature set. The second strategy was to scale down the weights on the existing features and then set the weights on the new feature group optimistically in proportion to the number of existing features. The last strategy we tried was to set all new weights to zero. In fact, we tried a parametric implementation which could vary the weights on the new feature groups from 0.0 to the maximum value of $1/f$, where f is the number of feature groups in the existing feature set.

The results from our experiments showed that out of these different strategies pessimistic initialization of weights (setting them all to zero) worked the best. One possible explanation for this is that setting the new weights optimistically disturbs the agent's existing state value estimates and forces the feature set to undergo some period of readjustment at the beginning of every new generation. On the other hand, setting the new weights to zero allows the mutated agents to retain their existing value estimates for the states. When a mutated agent starts operating in the new generation, the newly added feature starts with having no influence in evaluating the state values and just gets trained based on the decisions that the previous feature groups prefer. Then, provided the new feature is useful, at some point it could come into play by differentiating between states that would otherwise look identical to the existing features.

6.3 Selection Process

We use a soft-max method for determining which agents get to survive for the next generation and the probabilities by which they get selected for mutation and generation of new feature sets. At the end of each generation, the top 15% of the population are selected for survival. Then, a soft-max algorithm is used to randomly select one of the surviving agents and augment its feature set by applying one of the available featurizers to generate a new feature.

Using a soft-max method allows for maintaining some diversity in the population, and at the same time prioritization feature sets which have performed better. At the end of each generation, the rewards received by the different agents are normalized against the average and the soft-max algorithm is applied. We use a temperature value of $\tau = 2.0$. Normalizing against the average allows the algorithm to behave more consistently with different scales in reward values.

7. RESULTS AND ANALYSIS

7.1 Knight Joust

To get an idea how a straight-forward tabular implementation of Sarsa (λ) would do on the simple Knight Joust task, we trained a tabular Sarsa (λ) on the raw state variables over 20 trials. The rewards nearly plateaued at about 375 per episode after about 500000 episodes. In this domain, the agent can receive a theoretical maximum of 400, assuming it never has to use jumps.

Figure 5 shows the results of running Gradient Descent Sarsa(λ) for the Knight Joust problem with the 300 hand-coded binary features explained in Section 4.

As seen in Figure 5, after 3000 episodes of training the agent can achieve an average reward of about 340. This is the average over

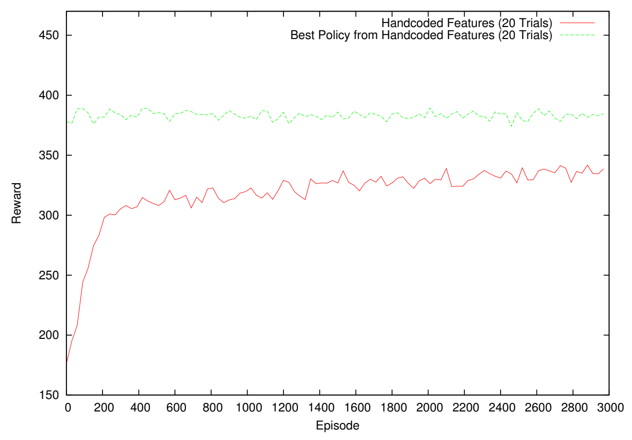

Figure 5: Knight Joust: Rewards received by the agent using hand-coded features

20 trials. Once the 20 trials were finished, we selected the trial that achieved the highest rewards. The weights learned in this trial represented the best policy that the hand-coded features learned over the 20 trials. We then fixed these weights and evaluated the agent over 3000 randomly generated Knight Joust problems. We repeated this evaluation 20 times with different sets of randomly generated problems. The flat line in Figure 5 shows the rewards achieved by this policy.

Next, we ran the evolutionary algorithm in this domain. Figure 6 shows the rewards received by the champions of each generation in the evolutionary algorithm. The graph shows 3000 episodes, corresponding to 15 generations with 200 episodes each. Note that this graph reflects a single run of the evolutionary algorithm. Therefore, the solid line shows the rewards received by an individual champion in each generation. This might explain why this plot is more noisy compared to Figure 5, where the rewards were averaged over 20 trials. Also, each 200 episodes in this graph correspond to a possibly different feature set, since it reflects the rewards received by the champion of that generation. The second line in the graph shows the average rewards received by all the agents in the population in each generation. As seen in Figure 5, there is not a big difference between the champion and the other agents in the population.

As shown in the Figure 6, the evolutionary solution is able to achieve an average reward of about 400 per episode, which is higher than the rewards achieved by the hand-coded feature set. The champion agent reaches this level after about 5 generations (1000 episodes.) At this point, the champion's feature set contains at most 5 feature groups, while the hand-coded feature set contains 30 feature groups as outlined in Section 4.2.

At the end of the 15 generations, we selected the generation whose champion achieved the highest reward. This was not necessarily the last generation champion. The algorithm was usually able to find an efficient feature set within just a few generations, such that adding extra features to this feature set did not improve the performance. We then extracted the policy learned by this champion by fixing the learned weights and evaluated this policy over 3000 randomly generated problems over 20 trials.

Figure 7 compares the rewards received by the champions in the evolutionary algorithm with the rewards received by the baseline solution using the hand-coded set of features. It also compares the rewards achieved by the best policies produced by each algorithm. As shown in the figure, the evolutionary algorithm is able to find better policies than the hand-coded features. For the evolutionary algorithm, the rewards achieved by the best policy are just slightly higher than the rewards achieved by the generation champions. However, it seems that in the case of hand-coded features, there is a wider gap between the best policy and the average rewards over 20 trials.

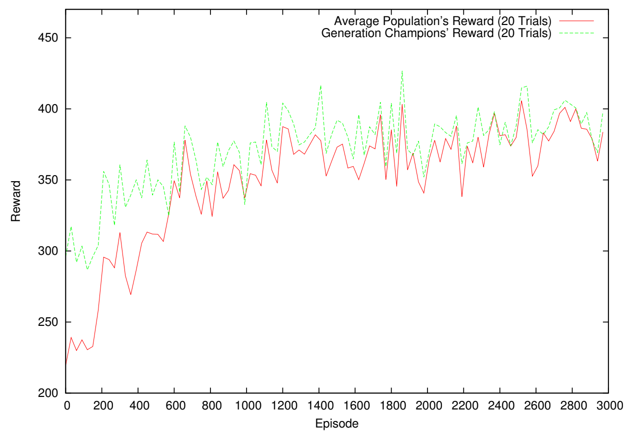

Figure 6: Knight Joust: Rewards received by the agents in the evolutionary algorithm

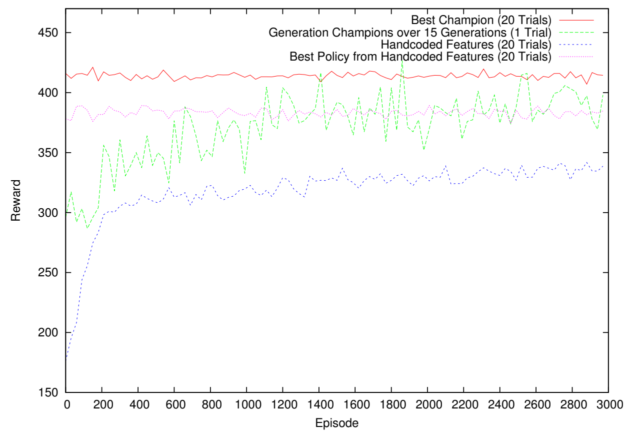

Figure 7: Knight Joust: Comparing the performance of the best policies obtained from hand-coded features and the evolutionary algorithm

Table 1 shows the feature set of the champions of the first few generations, as well as the feature set of the final champion at the end of the 15th generation. At the end of the second generation, the agent using the two features `Dist-X(player-opponent)` and `Dist-Y(player-opponent)` shows the best performance, which is somewhat expected due to the characteristics of the domain. The `Dist-X` feature is based on the distance of the two points projected on the X axis. Likewise, the The `Dist-Y` feature measures the distance projected on the Y axis. The numbers inside the square braces show the tiling offsets used by each feature group. The value of offset can range anywhere from 0.0 up to 1.0 and is generated randomly at the time of feature generation.

G.	Feature Set	R.
1	Dist-Y(opponent-lowerleft)[0.0]	297.90
2	Dist-X(player-opponent)[1.0], Dist-Y(player-opponent)[0.0]	341.20
3	Dist-X(opponent-player)[1.0], Point-XY(player)[0.4], Point-XY(opponent)[0.7]	344.50
4	Dist-Y(player-lowerleft)[0.0], Dist(player-opponent)[0.5], Angle(opponent-lowerright-player)[0.8]	364.00
5	Dist-X(opponent-player)[1.0], Point-XY(player)[0.4], Dist-X(opponent-lowerleft)[1.0], Dist-Y(opponent-player)[1.0], Dist-X(player-opponent)[1.0]	365.10
…	…	…
15	Dist-X(opponent-player)[1.0], Point-XY(player)[0.4], Dist-X(opponent-lowerleft)[1.0], Dist-Y(opponent-player)[1.0], Dist(opponent-player)[0.8], Interaction(Dist-Y(opponent-player)[1.0] * Dist(opponent-player)[0.8]), Dist-Y(opponent-player)[0.0], Angle(opponent-player-upperleft)[0.6], Point-XY(player)[0.4], Point-X(player)[0.1], Dist-X(player-lowerright)[1.0], Angle(player-upperleft-opponent)[0.2]	390.90

Table 1: Knight Joust: Champion feature sets at the end of some generations

Even though the evolutionary algorithm is able to surpass the agent with the hand-coded set of features, we should observe that it has a higher computational cost. For every training episode used by the baseline implementation, the evolutionary algorithm uses 100 episodes for training all the agents in the current population.

7.2 Simplified Keepaway Soccer

Figure 8 shows the result of running both algorithms in the Simplified Keepaway domain. As shown in the figure, at the end of 3000 episodes the agent with hand-coded features achieves an average award of about 10 per episode. Also, when we extracted the best policy out of the 20 trials, the hand-coded feature set could achieve an average reward of about 18 per episode over the 3000 randomly generated problems. This is while the champions in the evolutionary algorithm can reach an average reward of about 24 and the best policy discovered reaches about 30.

For Simplified Keepaway Soccer, the gap between the evolutionary algorithm and the hand-coded feature set is quite high. This is probably because the hand-coded feature set that we have selected is not well suited for our version of the keepaway game. This feature set was designed by the authors in [7] for a different version of the game. We could have tried finding other candidates for the hand-coded feature sets. This would have been a trial and error process; Exactly the kind of which the solution in this paper is designed to avoid.

It is also possible that hand-coded feature set needs many more episodes to learn the task. It is likely that the large number of features in the hand-coded agent combined with optimistic initialization of the weights does not allow the agent to exit the initial exploratory phase of learning by the end of 3000 episodes.

Table 2 shows the champion feature sets discovered in this domain at the end of some representative generations.

7.3 Mini-soccer

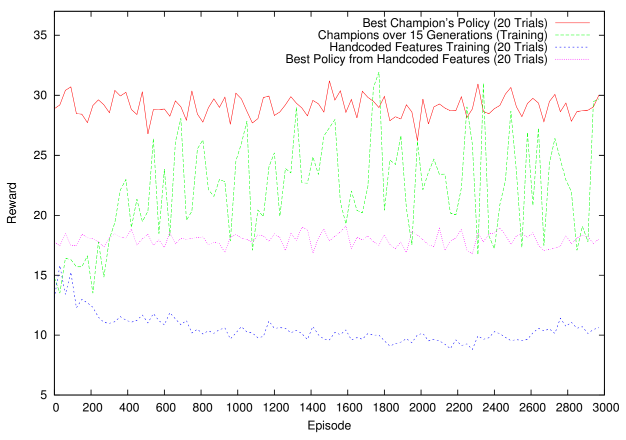

Figure 8: Simplified Keepaway: Comparing the performance of the best policies obtained from hand-coded features and the evolutionary algorithm

G.	Feature Set	R.
1	Dist-Y(keeper2-upperleft)[1.0]	15.46
2	Dist-Y(keeper2-upperleft)[1.0], Dist-Y(taker1-keeper1)[0.0]	17.91
3	Dist-Y(keeper2-upperleft)[1.0], Dist-Y(taker1-keeper1)[0.0], Dist-X(taker1-keeper2)[1.0]	22.71
4	Dist(taker2-upperright)[0.6], Dist-Y(taker2-keeper1)[0.0], Point-XY(taker2)[0.7], Interaction(Dist-Y(taker2-keeper1)[0.0] * Point-XY(taker2)[0.7])	22.71
…	…	…
15	Dist(taker2-upperright)[0.6], Dist-Y(taker2-keeper1)[0.0], Point-X(keeper2)[0.4], Dist(keeper3-keeper2)[0.8], Interaction(Dist-Y(taker2-keeper1)[0.0] * Dist(taker2-upperright)[0.6]), Point-X(taker1)[0.7], Dist-Y(taker2-keeper1)[1.0], Dist-X(keeper2-lowerleft)[1.0], Point-X(keeper2)[0.4], Point-X(taker1)[0.7], Dist-X(taker1-keeper2)[1.0], Interaction(Point-X(keeper2)[0.4] * Point-X(taker1)[0.7])	22.38

Table 2: Simplified Keepaway: Champion feature sets at the end of some generations

As in the previous domains, we present the results on performance of the hand-coded feature set and the evolutionary algorithm both during training, and after we have extracted their best policies.

Figure 9 shows the the results for running both algorithms. The average rewards received by the hand-coded feature set reaches about 0.1. On the other hand, the generation champions in the evolutionary algorithm are able to reach an average reward of 0.35 after about 10 generations. Because of the discounting factor, the best an agent can do in this domain is receiving a reward of about 0.41 and the worst it can do is to receive -0.41.

Figure 9 also compares the performance of the best policies extracted from each algorithm. In this domain, our hand-coded feature set was able to learn an efficient policy. However, our hand-coded feature set used 71 feature groups discretized from 8 original features. This is while the feature set learned by the evolutionary algorithm uses far fewer features groups.

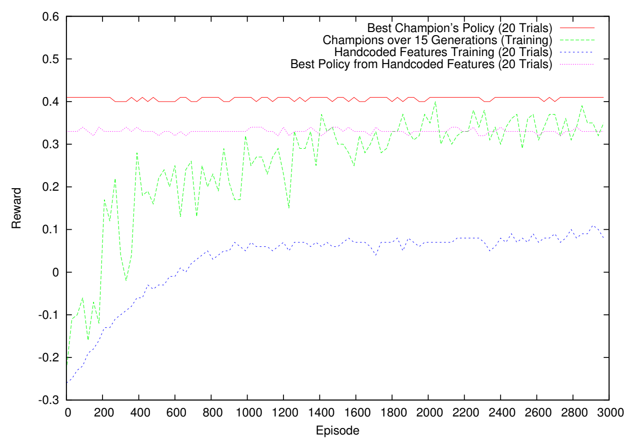

Figure 9: Mini-soccer: Comparing the performance of the best policies obtained from hand-coded features and the evolutionary algorithm

G.	Feature Set	R.
1	Angle(opponent-rightgoaltop-player)[0.6]	-0.12
2	Angle(player-opponent-rightgoalbottom)[0.8], Flag(player-has-ball)	0.10
3	Angle(opponent-rightgoalcenter-player)[0.1], Flag(player-has-ball), Dist(player-opponent)[0.6]	0.20
4	Angle(opponent-rightgoalcenter-player)[0.1], Flag(player-has-ball), Dist(opponent-player)[0.1], Angle(opponent-rightgoalcenter-player)[0.4]	0.21
5	Angle(opponent-rightgoalcenter-player)[0.1], Flag(player-has-ball), Dist(opponent-player)[0.1], Interaction(Flag(player-has-ball) * Angle(opponent-rightgoalcenter-player)[0.1]), Dist-Y(player-opponent)[1.0]	0.21
...	...	...

Table 3: Mini-soccer: Champion feature sets at the end of some generations

Table 3 shows the champion feature sets discovered in the Mini-soccer domain at the end of some representative generations. The Mini-soccer is a domain where there are interactions between the features: An efficient policy needs to make a distinction between the states where the player has the ball, and therefore need to go for the goal, and the states where the player does not have the ball, and therefore needs to block the opponent to gain possession of the ball. This is reflected in the champion feature sets discovered by the evolutionary algorithm; The algorithm selects many interaction type features which combine the ball possession flag with other features based on angles and distances.

8. CONCLUSIONS

We have introduced and evaluated an evolutionary algorithm for discovering efficient features for learning RL tasks using gradient descent Sarsa(λ) as an example linear method. Our evaluation results show that the algorithm has been able to evolve agents which are able to learn the tasks efficiently, at least as compared with our hand-coded feature sets. However, the evolutionary algorithm is able to identify the most important set of features for each problem. Inspecting the champion feature sets of the earlier generations reveals some minimal sets of features which have resulted in the biggest improvements in rewards.

8.1 Potential Improvements

The evolutionary algorithm can be extended to be used for online learning, where it is important to maximize the cumulative reward received over all episodes. To achieve this, we could use an ϵ-soft or a soft-max method to give a higher priority to training the agents that are more promising.

Another idea for improving the algorithm is designing a credit system which could guide the Mutator and the featurizers on the effectiveness of their previous choices. For example, if a feature based on the distance between two points S_1 and S_2 performs very well in training, the algorithm can assign some credit both to the distance featurizer, and to the points S_1 and S_2. The credit system would make the distance featurizer more likely to get selected in subsequent mutations. Also, any featurizers looking for points in the state would be more likely to select S_1 or S_2. One can also use an idea similar to eligibility traces to assign back credit to the components of features that were created in the previous generations.

In situations where the number of featurizers or state variables is very large, this form of credit assignment can be expected to increase the rate of progress in the evolutionary algorithm.

9. REFERENCES

[1] T. Fawcett. Feature discovery for problem solving systems. 1993.

[2] J. Johns and S. Mahadevan. Constructing basis functions from directed graphs for value function approximation. In *Proceedings of the 24th international conference on Machine learning*, pages 385–392. ACM, 2007.

[3] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3):462–466, 1952.

[4] K. Krawiec. Genetic programming-based construction of features for machine learning and knowledge discovery tasks. *Genetic Programming and Evolvable Machines*, 3(4):329–343, 2002.

[5] M. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the eleventh international conference on machine learning*, volume 157, page 163. Citeseer, 1994.

[6] R. Parr, C. Painter-Wakefield, L. Li, and M. Littman. Analyzing feature generation for value-function approximation. In *Proceedings of the 24th international conference on Machine learning*, pages 737–744. ACM, 2007.

[7] P. Stone, R. Sutton, and G. Kuhlmann. Reinforcement learning for robocup soccer keepaway. *Adaptive Behavior*, 13(3):165, 2005.

[8] R. Sutton and A. Barto. *Introduction to reinforcement learning*. MIT Press, 1998.

[9] M. E. Taylor and P. Stone. Cross-domain transfer for reinforcement learning. In *Proceedings of the Twenty-Fourth International Conference on Machine Learning (ICML)*, June 2007.

[10] S. Whiteson and P. Stone. Evolutionary function approximation for reinforcement learning. *The Journal of Machine Learning Research*, 7:877–917, 2006.

[11] S. Wilson. Classifiers that approximate functions. *Natural Computing*, 1(2):211–234, 2002.