

Using Training Regimens to Teach Expanding Function Approximators

Peng Zang
pengzang@cc.gatech.edu

Arya J. Irani
arya@cc.gatech.edu

Peng Zhou
pzhou6@cc.gatech.edu

Charles L. Isbell Jr.
isbell@cc.gatech.edu

Andrea L. Thomaz
athomaz@cc.gatech.edu

Georgia Institute of Technology, College of Computing
801 Atlantic Dr., Atlanta, GA 30332

1. ABSTRACT

In complex real-world environments, traditional (tabular) techniques for solving Reinforcement Learning (RL) do not scale. Function approximation is needed, but unfortunately, existing approaches generally have poor convergence and optimality guarantees. Additionally, for the case of human environments, it is valuable to be able to leverage human input. In this paper we introduce Expanding Value Function Approximation (EVFA), a function approximation algorithm that returns the optimal value function given sufficient rounds. To leverage human input, we introduce a new human-agent interaction scheme, training regimens, which allow humans to interact with and improve agent learning in the setting of a machine learning game. In experiments, we show EVFA compares favorably to standard value approximation approaches. We also show that training regimens enable humans to further improve EVFA performance. In our user study, we find that non-experts are able to provide effective regimens and that they found the game fun.

Categories and Subject Descriptors

I.2.6 [Artificial Intelligence]: Learning—*Reinforcement learning*

General Terms

Algorithms, Human Factors

Keywords

Learning, Human-robot/agent interaction, Agents in games and virtual environments

2. INTRODUCTION

Our research goals center around deploying agents in human environments, *e.g.*, robotic assistants in homes, schools, or hospitals. It is intractable to pre-program a robot with every skill necessary in these domains, thus we focus on ways to allow agents like these to learn and adapt.

Cite as: Using Training Regimens to Teach Expanding Function Approximators, Peng Zang, Arya J. Irani, Peng Zhou, Andrea L. Thomaz, Charles L. Isbell Jr., *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, van der Hoek, Kaminka, Lespérance, Luck and Sen (eds.), May, 10–14, 2010, Toronto, Canada, pp. XXX-XXX.

Copyright © 2010, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

Reinforcement learning (RL) is a broad field that defines a particular framework for specifying how an agent can act to maximize its long-term utility in a stochastic (and perhaps only partially observable) environment. Approaches to RL typically focus on learning a *policy*, a reactive plan that accomplishes specific goals (expressed implicitly through a reward signal).

Traditional RL techniques depend on tabular representations. However, when faced with large state spaces, these techniques become intractable and it becomes necessary to approximate the value function, policy, or both. Additionally, we recognize that in many real-world environments, agents will be learning in close proximity to humans who can help improve their performance. Thus, learning to take advantage of human input also arises as a research goal.

Introducing approximations into RL techniques is, unfortunately, not straightforward. While tabular techniques have been shown to have many desirable properties such as convergence and optimality, the use of even simple approximations introduce errors which jeopardize these properties. This can occur even if the best approximation is found at each step before changing the policy, even over many different notions of “best”, from mean-squared-error to residual-gradient [12]. Subsequently, there has been much study in making effective use of approximations. Sutton *et al.* showed that if the policy representation is differentiable, gradient descent can be used to converge to a local optimum [12]. Gordon showed that specific types of approximators known as “averagers” (non-expansive in max norm, linear and monotone) maintain convergence [7]. Boyan presented a method for function approximation that does not rely upon intermediate value functions and showed that it terminates [4]. It is this last approach that we will focus on. In this paper, we introduce expanding value function approximation (EVFA), a function approximation scheme based on Boyan’s work that is guaranteed to terminate and achieve the optimal policy, given sufficient rounds, and any learner capable of representing the true value function.

While we will show that EVFA has good performance, we would like to be able to leverage available human input to further improve that performance. A variety of methods for applying human input have been explored. One common method is demonstration, where solutions to example problems are shown to the learner. Other methods have humans decompose problems so the learner need only solve a series of small problems (*e.g.*, hierarchical decompositions [5], goal-

based decomposition [8], or explicit training of skills [11]). Still other methods, like reward shaping [6] leverage human input to guide agent exploration. In this paper, we introduce a new interaction scheme whereby a human teacher helps the agent learn by providing a series of increasingly difficult problem instances, a *training regimen*. This is much like providing demonstrations, but without the burden of having to provide solutions. The agent must instead learn every step by itself. To explore this interaction mode, we built a video game for training agents much like the NERO video game [11] which similarly falls under the genre of machine learning games. Experiments show that human input improves learning performance over the baseline. In our user study, we find non-experts able to provide effective training that improves agent performance. Most participants also report agreeing that the game is fun.

In summary, to tackle complex domains, we introduce a new algorithm to improve function approximation. For the case when the target domain is a human environment, we present training regimens as a mechanism for leveraging human input to further boost performance.

3. PRELIMINARIES

We define a finite Markov Decision Problem (MDP) $M = (S, A, P_{ss'}^a, R_s^a, \gamma)$ by a finite set of states S , a finite set of actions A , a transition model $P_{ss'}^a = Pr(s'|s, a)$ specifying the probability of reaching state s' by taking action a in state s , a reward model $R_s^a = r(s, a)$ specifying the immediate reward received when taking action a in state s , and the discount factor $0 \leq \gamma \leq 1$.

In this work, we focus on episodic MDPs, *i.e.*, MDPs with absorbing or “goal” states. To ease discussion, we will assume (without loss of generality) that rewards are strictly negative. This will allow us to speak more intuitively in terms of cost, $c = -r$, rather than rewards.

A policy, $\pi : S \rightarrow A$, dictates which action an agent should perform in a particular state. The distance or *value* of a state $J^\pi(s)$ is the expected sum of discounted costs an agent receives, when following policy π from state s . $J^*(s)$ is the value of state s when an agent follows an optimal policy that minimizes long-term expected cost (maximizes long-term reward). Value iteration is an algorithm for computing J^* .

4. EVFA

Standard approaches to RL (such as value iteration) require intermediate value and policy functions. When using function approximation with these approaches, not only do we need to accurately approximate the final function, but all intermediate functions as well. This can be problematic as intermediate functions can vary wildly and are sensitive to initial values. In practice, this often results in erratic behavior. Often, one run can converge to a good policy while another run may fail completely. Sometimes, adding an additional feature to the feature set can trigger complete failure. This makes using RL with function approximation difficult.

While much progress has been done toward mitigating this effect, there is another line of work that avoids it altogether [3]. We call this approach “expanding” as opposed to “iterating” because it works by expanding an accurate approximation of limited size across the state space until it

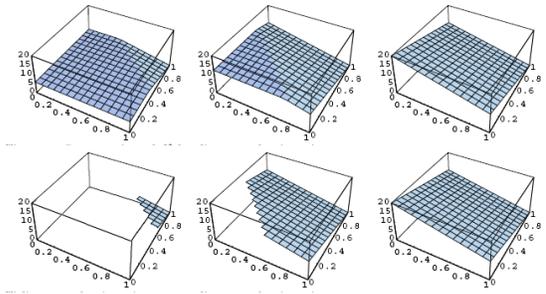


Figure 1: Iterating (top) vs Expanding (bottom) Value Function Approximation

covers the entire space, as opposed to starting with a poor approximation of the entire space and iteratively improving its accuracy.

Figure 1, based on illustrations in [4], shows how the two approaches differ. Notice how the expanding approach only ever attempts to model the optimal value function, albeit incompletely at first. We can summarize the benefits of the expanding approach over the iterative approach as follows:

- Eliminates intermediate functions as an error source.
- Offers potential computational benefits as we do not have to learn or represent intermediate value functions.
- Has better anytime behavior: the expanding approach always has an accurate (though partial) value function. By contrast, an intermediate value function of an iterative framework can be arbitrarily bad.

Our work is an extension of Boyan’s [3] to general, goal based MDPs. Note that although we focus on value function approximation, the same technique can be similarly applied to policy approximation as well.

4.1 Previous work

Boyan’s work is based on the idea of maintaining a “support” set of states whose final J^* values have been computed. When function approximation is performed, it is trained on just this set. Thus, in effect, the support set represents the region of space that has been approximated. The support set starts with just the terminal states and grows backwards until it covers the entire state space. Expansion or growth of the support set/approximated region is accomplished by computing, via one-step backup(s), the value of states whose next state(s) can be “accurately predicted” by the function approximator. These states and their computed values are then added into the support set. Whether a state’s value can be “accurately predicted” is decided by performing rollouts — simulated trajectories guided by the greedy policy of the approximated value function. If the rollouts verify the estimated value of the state, it is considered “accurately predicted”.

Boyan presents two algorithms, “Grow-Support” and “ROUT” [3], based on this basic workflow. Unfortunately, the algorithms have some limitations. “Grow-Support” requires deterministic domains. “ROUT” can handle stochastic transitions, but only in acyclic domains. Beyond these restrictions, the algorithms are also not guaranteed to return the optimal policy. Our approach resolves these issues.



Figure 2: Hallway with nondeterministic ice-patch



Figure 3: Hallway with teleporter to goal at far left. All actions have unit cost

4.2 Removing domain restrictions

Grow-Support and ROUT only work on deterministic/acyclic domains because they use one-step backups to expand the approximation. For nondeterministic domains, the algorithms may prematurely terminate because they have reached a point where all states on the fringe of the approximated region have values dependent upon states outside. Figure 2 illustrates this problem: from the ice-patch state, all actions have a chance of going left, making its value impossible to compute without first knowing the value of the state left of it. Since the function approximator grows from the right, it can never grow past this point.

To solve this problem, we rely not on single step backups to compute values, but full MDP solvers. For computational tractability we only use solvers that take advantage of the start state (or set of start states) such as Real-Time Dynamic Programming [2] (RTDP). We call these single source solvers (SSS). In general, solving the value of a single state can be as difficult as solving the value of all states (*e.g.*, fully connected domains). In practice however, domains are rarely fully connected. Typically, nondeterminism is localized. SSS are also relatively insensitive to the size of the state space; rather, they depend on the length of the problem. By taking advantage of these two attributes, we can make SSS efficient at computing the value of a state as long as the length of the problem is short.

4.3 Solving for the optimal policy

In order for EVFA to work, we must expand the approximated region. To do so in a tractable manner, we rely on the estimates of state values inside the region to compute state values outside. Since there may be errors in the approximation, we must verify any value estimates. “Grow-Support” and “ROUT” rely on rollouts and Bellman errors to verify value estimates. However, it is important to recognize that these procedures only verify an upper bound on the distance, not the actual distance. Figure 3 illustrates this problem. After adding $(state, distance)$ pairs $(A, 1)$ and $(B, 2)$ into the support set and training the function approximator, state C will appear correctly approximated and pass verification with distance 3 ($C \rightarrow B \rightarrow A \rightarrow Goal$). In fact however, state C is not correctly approximated as its true value is 2 ($C \rightarrow Teleport \rightarrow Goal$). Previous work does not address this problem. As a result, they are susceptible to over-estimation errors that may persist and spread over expansions.

In our work, we mitigate this effect by adding the ability to detect and correct errors. We do so in two ways. First, whenever the value of a state is queried, it is computed

by a local refinement over the output of the value function approximator. This is done even if the surrounding states have been verified. If the approximated value and the refined value differ, the lower value is returned¹. In our algorithm, we mark states within some distance of the queried state and do not use approximator estimates (even if verified) on these marked states. This ensures a local region surrounding the queried state is computed from scratch rather than taken from the function approximator. Second, whenever these discrepancies are encountered, we shrink the effective region of the approximator and invalidate all values in the support set greater than refined value. This mitigates error propagation.

4.4 Speeding up verification

Value estimate verification is frequently required and can be expensive in non-deterministic domains. We would like to reduce the number of times it must be called. To this end, we introduce the concept of assigning a maximum effective region of the predictor (MERP), to all trained approximators. The MERP describes, in terms of distance to the goal, the region within which the approximator is accurate. If the function approximator is asked to estimate the value of a state and returns a value greater than MERP, we can immediately conclude that our estimate is inaccurate without needing to perform verification. The verification procedure is only performed when an estimate is within MERP and thus may be accurate.

Introducing the MERP puts a hard limit on the effective region of our trained approximators. Unlike Boyan’s algorithms, in which the effective approximated region may be amorphous, our regions are always strictly spherical, centered at the goal.

4.5 EVFA algorithm

EVFA (see Algorithm 1) starts with the support set, $supp$, containing just the goal states, and expands it until we have an approximator that is accurate over the entire state space. To expand the approximated region, we sample a state, solve it, and add the resulting state value pair as a new training example into $supp$. Eventually $supp$ will grow large enough to enable supervised learner L to produce an accurate hypothesis (approximation of the value function), at which point we can update the approximation and increase its effective region.

Solving a state is performed by the SSS. In our work we target large discretized domains and so a modified version of RTDP is used. RTDP is modified in two ways. First, its value table (excepting immediate region of radius r) is initialized to $h(s)$ where verified and to the optimistic estimate $merp$ otherwise. This initialization allows RTDP to leverage the existing estimated region so it does not have to search all the way to the goal. As soon as it hits a known-valued state, we can act as if we have reached to goal, no search past that known state is needed. This makes RTDP fast. We do not use $h(s)$ in the immediate region around s in order to ensure local refinement, *i.e.*, to find and correct possible errors. The second modification we make to RTDP is to employ a distance cap of radius r . If r is ever exceeded we consider the state too far to solve and return NaN. Recall that SSS is only efficient at solving relatively short

¹Both values are upper bounds, thus the smaller of the two is the tighter upper bound.

Algorithm 1 *EVFA*

Require: MDP M , Learner L , $n \in \mathbb{N}$,
 $r \in \mathbb{R}_{>0}$, $\alpha \in (0, 1]$, $\epsilon \in \mathbb{R}$
 $supp \leftarrow \{(s, 0) \mid s \text{ is a goal of } M\}$
 $backup \leftarrow \emptyset$
 $h \leftarrow \text{NULL}$
 $merp \leftarrow 0$
for $i = 0$ to n **do**
 $s \leftarrow$ random state from M
 $j \leftarrow \text{SSS}(s, h, merp, r, \epsilon)$
 if $j = \text{NaN}$ **then**
 continue
 else if $(s, oldj) \in supp$ where $oldj - j > 2\epsilon$ **then**
 $merp \leftarrow \text{floor}(j/r)r$
 $backup \leftarrow \{(s, x) \in supp \mid x > merp\}$
 $supp \leftarrow \{(s, x) \in supp \mid x \leq merp\}$
 $supp \leftarrow \text{add}(supp, (s, j))$
 $\text{scourBackup}(backup, supp)$
 else
 $supp \leftarrow \text{add}(supp, (s, \min(j, oldj)))$
 $trainset, testset \leftarrow \text{randsplit}(supp)$
 $h' \leftarrow L(trainset)$
 if $\text{test}(h', testset, \alpha)$ **then**
 $h \leftarrow h'$
 $merp \leftarrow merp + r$
 $\text{scourBackup}(backup, supp)$
 end if
 end if
end for
return $h, merp$

problems. We would much rather solve long problems by repeatedly expanding the approximated region from solving short problems than solving them directly with SSS. The distance cap, r , allows us to control the expansion step-size. A large r allows distant problems to be solved and trained on. This means a larger expansion, but computing the solution to each problem will take longer. We can also view r as imposing a distance based decomposition on the MDP so that RTDP is only ever asked to provide solutions to incremental problems. In this view, r controls the granularity of that decomposition.

We do not train L on the whole of $supp$. Instead, some portion is held out to use as a test set. To test a hypothesis, we measure the percentage of states from the test set whose values can be verified via rollouts or similar procedure. If the percent verifiable exceeds α , our precision parameter, we consider the test passed and expand the approximated region.

Errors may exist in $supp$. They are found when we compute a value that is significantly (by twice the SSS precision ϵ) lower than the one stored in $supp^2$. When such an error is encountered, a multi-step process we call a “merp regression” occurs. First, we reset the $merp$ to the last radius multiple before the error. Second, state value pairs whose value are greater than $merp$ are removed from $supp$ as the error may have propagated to them. These pairs are backed up in $backup$. Finally, they may later be added back to $supp$

²To see why, consider that all values produced by SSS must be upper bound as SSS itself is sound and it relies on verified values which are guaranteed to be upper bounds

by `scourBackup`. `scourBackup` is a procedure which runs SSS on every state in $backup$. Those that can be solved are added back into $supp$.

In actual implementation, we make a few optimizing changes. First, because (re)training the function approximator can be time consuming, we only train after many states have been added to the support set. States whose values are already accurately predicted by the hypothesis do not count. This change not only makes training more efficient, but also means we can rapidly expand the effective region and only train when we bump up against errors in the approximator. Second, we modify RTDP such that it returns not just the value of the start state, but the values of all states along the optimal path to the goal. If the RTDP solution ends in an estimated region, we use the verification procedure to produce the values of states along the optimal path from the beginning of the region to the goal. We then add all these state-value pairs into the support set. This modification makes training example generation more efficient. Finally, having to specify the number of expansion rounds to run is troublesome. Instead, we use a stopping condition. Given a randomly chosen “bench” set of states, the algorithm is run until the percentage of states whose values can be verified exceeds α . This ensures, with high probability, the estimated region covers the entire space.

4.6 Optimality

Optimality is guaranteed probabilistically. For ease of discussion we will use unequivocal terms such as “perfect”, but we mean this in a probabilistic sense as in the probability of error(s) approaches zero.

Observe that $merp$ must always be a multiple of radius r . For convenience, we will refer to $merp$ by level $l = merp/r$ where l is a natural number.

We make the following assumptions: (1) PAC learner L , (2) hypothesis space of L contains J^* , and (3) radius r is large enough to guarantee SSS can always solve some states.

THEOREM 1. *Given sufficient rounds, EVFA will produce $(h, merp = kr)$ or simply (h, k) such that with high probability $h(s) = J^*(s)$ when $J^*(s) \leq kr$, for all natural numbers k .*

PROOF. Suppose there exists a k such that EVFA never produces a perfect (h, k) no matter how many rounds are run. Given our radius assumption, some states will be solvable by SSS. If the solution does not identify an error in $supp$, the state value pair will be added to $supp$. If we find no errors in $supp$ for many consecutive rounds, $supp$ can grow arbitrarily large. It also implies that $supp$ contains no errors. Given sufficient rounds under these conditions, learner L will produce highly accurate hypotheses. This means we can grow $merp$ to arbitrary sizes. As a result we must always periodically find errors in $supp$.

SSS guarantee computed values to be upper bounds and the state space is finite; as long as values are not forgotten, there can only be a finite number of (downward) corrections before $supp$ must be correct. In EVFA, if a state value pair must be removed from $supp$, it is archived in $backup$ and later added back in. Thus after a finite number of rounds $supp$ must be correct. However, this contradicts our previously derived result that we must periodically find errors in $supp$. Thus our assumption must be incorrect. \square

The proof requires that the hypothesis space of L contain

J^* . In practice, we only need L to contain a hypothesis reasonably close to J^* . This is controlled by the precision parameter α . If the hypothesis space cannot express any hypotheses reasonably close, EVFA will learn the largest partial value function it can express. For example, if limited to a linear model in a 2 room maze world, EVFA would expand the value function from the goal until it reached the entryway to the second room. At that point it would be unable to expand further, because the nonlinearity of values caused by the wall prevents some states from being verified, causing the algorithm to halt.

EVFA optimality does not require any specific form for the approximation algorithm. This means unlike iterative techniques that are limited to linear architectures for stability, we can use any PAC learner. This eases feature engineering.

5. LEVERAGING HUMAN INPUT

We introduce a new interaction scheme whereby a human teacher helps a learning agent by providing a series of increasingly difficult problem instances³, a *training regimen*.

We developed this interaction scheme when standard teaching techniques proved difficult to apply to EVFA. Learning by demonstration [1] for example, while rich in information, requires the teacher to provide solutions to problems (typically by performing the task). This becomes burdensome in complex domains that may require thousands of samples to learn. Training regimens are much like demonstrations but without the need to provide solutions. More importantly, this enables augmentation techniques where the system can help obtain sample problems so we need not literally specify thousands of training problems. Training regimens also provide decomposition. However, unlike approaches that require specialized learners and frameworks (*e.g.*, MAXQ hierarchies [5]), regimens do so in a general fashion. This provides wider applicability and in particular, applies to EVFA. Training regimens can also be seen as an alternative way of guiding exploration. Unlike reward shaping where one guides exploration indirectly by authoring a potential-based shaping function [10], training regimens allows one to guide exploration directly by giving samples of where the learner should focus.

Training regimens provide many advantages to an automated solver:

Focus: Algorithms typically assume the goal is to solve all instances (to find a policy over all states), and that all instances are equally important. These assumptions are often false. For example, we do not care about solving all chess board positions, just those reachable. A properly-tailored regimen increases learning efficiency by maximizing generalization while minimizing the number of instances the learner must see and solve. For example one may provide a higher density of examples in complex or important regions and fewer examples elsewhere. In an interactive setting, the instance chosen can be tailored to learner performance, *e.g.*, highlight errors in the learned function. Finally, proper focus provides better measures of performance as it allows us to weight errors based on importance.

Sampling: Function approximation approaches to solving RL typically use random state sampling. Usually, the uni-

form distribution is assumed. While we often assume this sampling is inexpensive, domain constraints may make it otherwise. Sampling is also difficult if we need to sample according to some specific distribution. EVFA for example, requires that we sample problem instances of particular difficulty levels at different times because the SSS has limited range. To deal with these difficulties, rejection sampling is commonly used. Unfortunately, depending on how well the generative distribution matches the target distribution, it can be very expensive to perform. Providing a training regimen mitigates this need.

Decomposition: How one orders problem instances in a regimen guides the learner. By ordering instances such that more difficult ones build upon simpler ones, we can save the learner significant work. Solving each new problem instance will then only require incremental effort. We order instances by distance so that solutions to distant problems can take advantage of prior solutions to shorter ones. This approach improves learning efficiency by relying on stitching value functions together rather than on specifics of any particular learning algorithm used. As a result, it is more widely applicable.

We implement the training regimen interaction as a video game much like NERO. In the following sections we will give a description of our game, explain our methods of specifying training regimens, and discuss the results of our user study.

5.1 Description

Our game is based on a school metaphor. The human takes on the role of a teacher or parent whose job it is to advance the agent (or student) through successive grades until graduation where upon the agent can successfully perform the target task. They are provided an interface where they can give the agent a series of problem instances organized as “homeworks”. They are also prompted to give “tests”, sets of instances that are previously unseen by the agent to measure the agent’s learning. When they deem the agent’s performance sufficient, they can advance the agent into the next grade. Finally, the agent may “flunk” out of a grade if they are advanced prematurely. For EVFA, this corresponds to a “merp regression” (see Section 4.5).

Figure 4 shows our game in the homework creation screen. Our game uses the Wall-E domain. In this domain, the agent, Wall-E, is on a spacedock and tasked with moving a cube of trash from some location to the port (some other location). It must then return to a charger. The domain is discretized into a 12x12 grid. In the homework creation screen, the human teacher must create many problem instances which will, together, make up a homework given to Wall-E when the “submit” button is pressed. We use the game metaphor of a “level editor” for problem instance creation. Creating a level is simple. The human teacher simply drags and drops various level building elements such as Wall-E’s start and end positions and the cube’s start and end positions on to the board. The end positions correspond to the port and charger. Once the level is setup, the teacher must click the “Generate” button. This adds the level and many similar ones to the homework. The generation mechanism allows the teacher to create more than one homework problem per board setup. This process, which we call “regimen augmentation”, decreases the workload of the teacher.

Recall that problem instances must be ordered by increasing length. The “grade” that Wall-E is in controls the ap-

³Here, an *instance* refers to solving a RL problem *wrt.* a specific start and goal state pair, such as the starting and ending squares in a maze.

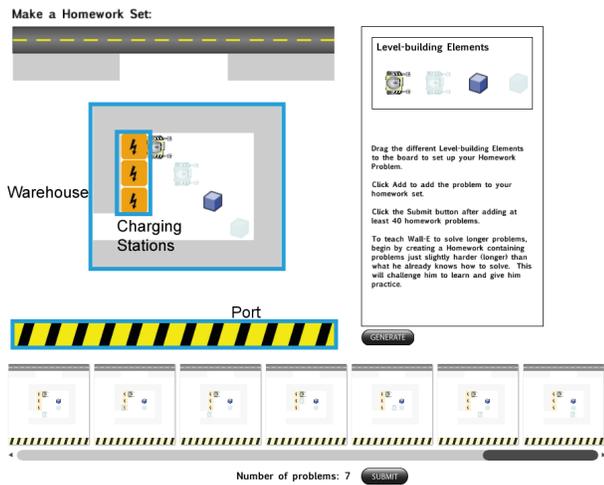


Figure 4: The homework creation screen of our game. The right panel holds our level building elements and instructions. The bottom panel collects the problems generated. The main element of the screen is a level map showing initial and end locations (ghosted icons)

proximate length of the problem. We enforce problem instances of increasing length by starting the game with Wall-E in the lowest grade, 1st Grade. In this grade, only the shortest problems are accepted so the human teacher must provide problems that are almost already fully solved. To advance Wall-E and gain access to longer problems, the teacher must give him homework(s) and evaluate his progress via tests. A test is just like a homework but instead of solved to generate training data, it is used to test the trained function approximator. If the function approximator fails the verification procedure on a test problem, we mark it as Wall-E having gotten the problem “wrong”. If Wall-E scores high on tests then it means the trained approximator is accurate. At this point, we can advance him to the next level. This corresponds to updating the h and increasing the *merp* in the EVFA algorithm.

The game ends when Wall-E can pass a “graduation” test consisting of randomly chosen problems from the target task. For our game the task is to teach Wall-E to move the cube to the port and return to the charger subject to the following conditions:

- Wall-E always starts in the warehouse
- The cube always starts in the warehouse
- The charger must be located somewhere against the left wall of the warehouse
- The port must be located inside the loading area at the bottom of the map

5.2 Regimen augmentation

Complex domains may require many training instances to learn correctly. To require a human teacher to enter each instance manually would be too burdensome. Instead, we provide a system to augment human input so that many instances can be generated from relatively little effort. We explore two different forms of augmentation.

The first form is called the “exemplar form”. The idea is for humans to provide a few exemplar problem instances and the system will fill in the rest. It works as follows. The human teacher sets up a level as normal. When they hit generate, we will add not only the specified level but many similar levels automatically generated based on that level. Similar levels are generated through two mechanisms. In the random walk mechanism, short random walks are performed from the start position to generate alternative start positions and from the end position to generate alternative end positions. The cross-product of these alternative positions form the resulting set of similar levels. The second mechanism, random perturbations, generates similar levels by directly perturbing the feature vector that encodes the level. Some generated levels may not be valid. These are detected and discarded.

The second form is called the “distribution form”. In this form instead of specifying exemplars, we ask the human teacher to draw or mark a region representing a distribution from which the system can utilize random sampling to generate the levels. Since the state space is multi-dimensional we ask the teacher to draw a region for each object in the level. The region specifies how that object and its feature encoding is distributed. The joint distribution is then computed by the system and randomly sampled. Again, some generated levels may not be valid, these are removed.

5.3 User study

We ran an user study on our game to see if non-expert humans could play it to successfully train Wall-E, and to see if they found the game fun. We hypothesized the affirmative to both questions.

In the study, participants were first given instructions to read and were left a few minutes alone to do so. The instructions begin by providing a back story to introduce the user to their role as a teacher and the school-based context of the game. It then explains that their objective is to teach Wall-E and advance him through the grades towards the target task (see above). To help ensure the participants are motivated, we hold a competition among the Wall-Es trained by the participants. The winner, as the instructions explain, wins a \$20 Amazon gift card. The rest of the instructions describe the operation of various game interfaces and briefly explain the expected progression. Namely, that Wall-E starts in 1st Grade where he cannot solve but the simplest (shortest) of problems but will be able to solve longer problems as he advances Grades. The instructions also warn that advancing too quickly may result in Wall-E learning incorrect concepts which may cause him to flunk out of later grades. To help ensure participants understood the instructions, the experimenter also provided a brief demonstration highlighting the more important interface elements. Participants were then left to play the game. At the end of the study, an exit survey (including demographic questions) and brief interview were performed. Due to time constraints, we did not require participants to complete the game. Instead, we allotted a maximum of one hour for each participant. Participants could also choose to stop early.

We had 10 participants for our study. Participants were drawn from the campus community. Their age ranged from 16 to 54. Education levels as reported on the exit survey ranged from high school to PhD students. The survey also asked participants to rate the level to which they agreed with

the statement “I found the game fun” on a Likert scale. 6 participants reported agreement or strong agreement while 3 reported some level of disagreement. Interviews revealed some participants experienced confusion over what to do which may have negatively influenced their experience.

To measure how well participants trained Wall-E, we looked at how well their trained Wall-E performed and the number of training examples users gave to reach that level of performance. While participants gave varying amounts of training to Wall-E, on average, Wall-E received 4,500 training examples whereupon he is able to solve approximately 15% of randomly selected problems. By contrast, after the author had given 5,800 training examples, Wall-E was able to solve roughly 7% of the same randomly selected problems. From this, we conclude that non-experts are capable of providing effective training regimens.

It may seem surprising that non-experts produced better performance than the authors. However, as authors, our training regimens reflected a greater awareness of potential “merp regressions”. To avoid this, we tended to give many examples before advancing to ensure a properly trained predictor and avoid any regression. By contrast, users tended to advance earlier and simply retrained when faced with regression. This more aggressive approach led to higher sample efficiency.

With respect to overall performance, success rates of 15% and 7% may seem low. However, these rates reflect training set sizes of around 5,000 examples. As we see in Figure 5, with 24,000 examples, we achieve optimal performance.

6. EXPERIMENTS

Our experiments are organized into two sections. In the first section we seek to verify empirically the correctness characteristics of EVFA. Experiments will be in simpler domains and we will compare against various baselines. In the second section, we will focus on how taking advantage of human input can further improve EVFA performance. Emphasis will be on comparing performance of EVFA with and without human input of various forms. To demonstrate scalability and robustness, we will also use a more complex domain. For all experiments, EVFA used $r = 7.0$, $\alpha = 0.98$, and the regression tree algorithm GUIDE [9] as the learner.

6.1 Correctness

We used three domains to verify EVFA performance. The first is a two-room, deterministic maze domain. It is a commonly used domain for evaluating function approximation schemes because to the discontinuity the room divider introduces into the value function. The second and third are randomly generated mazes with non-deterministic wind elements. In a wind square, the agent has a 30% chance of being blown by the wind and move in that direction. 30% and 70% percent of the squares in the second and third mazes, respectively, are wind or wall elements. Actions in the maze are North, South, East and West. Rewards are uniformly -1. A single goal is located at (0,0).

For baseline comparison, we used LSPI with varying numbers of RBFs. We use the RBF $K(s, c) = \exp(-\frac{ED(s, c)^2}{2\sigma^2})$ where $ED(\cdot, \cdot)$ is the Euclidean distance and c is the Gaussian center. To extend the RBF to state-action space we duplicate it for each action. Specifically for each action u and center c , we generate basis function $\phi(s, a) = I(a =$

Table 1: Return of EVFA and LSPI variants on maze domains, averaged over 10 runs

Algorithm	Two Room	Rand 30	Rand 70
EVFA	15	20	24
LSPI (10 RBFs)	37	41	61
LSPI (20 RBFs)	52	32	50
LSPI (40 RBFs)	40	25	45

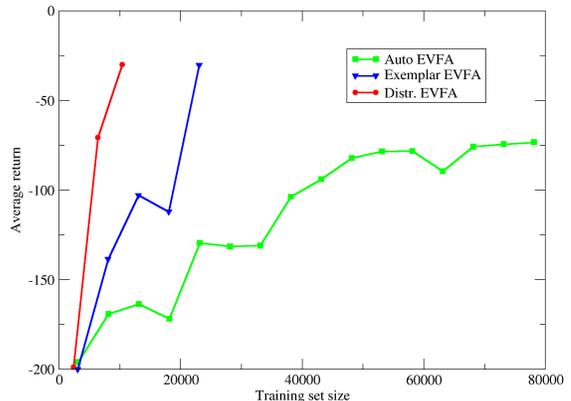


Figure 5: Sample complexity of different variants of EVFA

$u)K(s, c)$ where I is the indicator function. Centers for RBFs were randomly placed.

To obtain good performance from LSPI we had to try multiple configurations. Specifically, for each LSPI variant, we generated 20 center configurations. For each center configuration, we ran LSPI 20 times with maximum iterations set to 30. The best policy seen is returned. Table 1 shows our results. EVFA dominates LSPI. Of particular interest, is EVFA’s learning behavior in the Two-Room domain. It is able to reach optimal after using its learner just twice. This is because EVFA only (re)trains when it encounters state values it cannot correct predict. As a result, after the first few examples produce a hypothesis accurate in the first room, EVFA expands without (re)training until it enters the second room.

6.2 Interactive EVFA

In this section, we seek to empirically explore whether human interaction can further improve EVFA performance. We use the Wall-E domain for this series of experiments (see Section 5.1). State is represented as the vector [Wall-EX, Wall-EY, CubeX, CubeY, Holdingp, PortX, PortY, ChargerX, ChargerY]. Holdingp is a binary feature indicating whether Wall-E is currently holding the trash cube. Actions are North, South, East, West, Load, and Unload. While our user study suggests that non-experts are capable of providing effective training regimens, time constraints prevented them from training Wall-E with enough examples for our experiments. Thus for the experiments in this section, human interaction data was generated by the authors. In the following figures, automated EVFA results reflect an average over 10 runs, while interactive EVFA results reflect a single run.

Figure 5 shows our sample complexity results. Both aug-

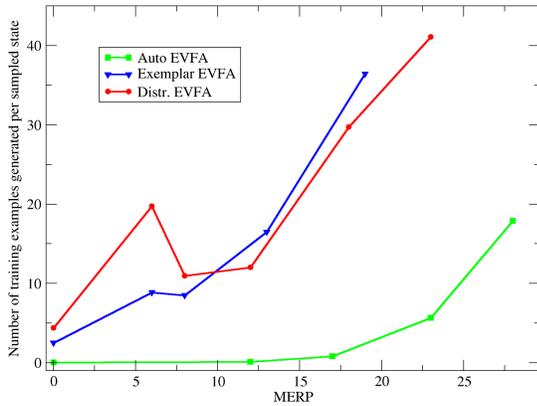


Figure 6: Rejection sampling efficiency of different variants of EVFA

mentation techniques outperform the automated baseline. This makes sense as the human provided training regimen can focus the learner on relevant areas of the state space and allow it to ignore, for example, states in which the Wall-E or the Cube is outside and above the warehouse entrance. Interestingly the distribution form of augmentation outperforms the exemplar form. Detailed analysis of the training examples generated shows that the exemplar form does not generate data as evenly distributed as the distribution form. As a result, the function approximator does not learn as well on the data generated by the exemplar form. This suggests two possible issues with the exemplar form. First, using random walks to generate similar states may bias the distribution. Second, while humans may be capable of specifying multi-dimensional regions of interest, they may be poor at performing random sampling of that region.

Beyond the ability to focus the learner, we expect human augmented EVFA to show improved performance from improved sampling efficiency. Recall that the automated algorithm must use rejection sampling to obtain problems of the appropriate difficulty. By contrast, humans can provide it almost directly. Some small amount of rejection sampling is necessary in performing augmentation of human input, but this should be insignificant compared to the amount required by the automated version. To explore this, we collected statistics on the number of training examples each sampled state generated. Figure 6 has our results. As expected, augmentation forms are far more efficient at generating samples. The automated version suffers the worst performance at the beginning when the region of interest is the immediate area surrounding the goal. This area is small relative to the size of the state space. As EVFA expands the coverage of the learned approximator, the region of interest also expands, resulting in improved efficiency. This suggests human guidance is more helpful at the beginning of learning.

7. CONCLUSION

In large, complex domains, tabular representations become intractable and it becomes necessary to approximate the value function. We introduced EVFA, an approach to function approximation that avoids intermediate value functions. EVFA has several desirable properties such as termi-

nation and optimality. It also has better anytime behavior and puts no limitations on the learning architecture used. EVFA relies on SSS to do the heavy lifting. As a result, it is not appropriate for some (*e.g.*, fully connected) MDPs. In practice however, we expect EVFA to perform well on most domains.

For domains in human environments, leveraging human input is important. We introduced a new human-agent interaction mode: training regimens. This mode was successfully applied to EVFA where we were able to obtain improvement over the purely automated baseline. By embedding this integration in a machine learning game we were able to show that non-experts are able to provide effective regimens. Training regimens are interesting because they allow a human to easily specify thousands of training examples with relatively little effort. This allows a human to easily focus agent learning on particular regions of a problem, and provide inexpensive sampling and decomposition.

8. REFERENCES

- [1] B. Argall, S. Chernova, M. Veloso, and B. Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, 2009.
- [2] A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artif. Intell.*, 72(1-2):81–138, 1995.
- [3] J. Boyan. *Learning Evaluation Functions for Global Optimization*. PhD thesis, Pittsburgh, PA, USA, 1998.
- [4] J. A. Boyan and A. W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in Neural Information Processing Systems 7*, pages 369–376. MIT Press, 1995.
- [5] T. Dietterich. The MAXQ method for hierarchical reinforcement learning. In *Proceedings of the fifteenth international conference on machine learning*, pages 118–126. Citeseer, 1998.
- [6] M. Dorigo and M. Colombetti. Robot shaping: developing autonomous agents through learning. *Artif. Intell.*, 71(2):321–370, 1994.
- [7] G. Gordon. Stable Function Approximation in Dynamic Programming. In *Proceedings of the Twelfth International Conference on Machine Learning*, page 261. Morgan Kaufmann, 1995.
- [8] J. Karlsson. *Learning to solve multiple goals*. PhD thesis, Rochester, NY, USA, 1997.
- [9] W. Loh. Regression trees with unbiased variable selection and interaction detection. *Statistica Sinica*, 12:361–386, 2002.
- [10] A. Y. Ng, D. Harada, and S. J. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 16, pages 278–287. Morgan Kaufmann, 1999.
- [11] K. O. Stanley, B. D. Bryant, and R. Miikkulainen. Real-time neuroevolution in the nero video game. *IEEE Transactions on Evolutionary Computation*, 9:653–668, 2005.
- [12] R. S. Sutton, D. Mcallester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, pages 1057–1063. MIT Press, 1999.