Relative Value Function Approximation

Paul E. Utgoff
Doina Precup

Technical Report 97-03
January 7, 1997

Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Telephone: (413) 545-4843
Net: utgoff@cs.umass.edu

Abstract

A form of temporal difference learning is presented that learns the relative utility of states, instead of the absolute utility. This formulation backs up decisions instead of values, making it possible to learn a simpler function for defining a decision-making policy. A nonlinear relative value function can be learned without increasing the dimensionality of the inputs.

Contents

1 Introduction

One often needs to construct a decision-making component of a larger program. For example, in scheduling one needs to decide which requests for resources to satisfy. In diagnosis, one needs to decide which test or treatment to perform. In financial markets, one needs to decide which trades to make. One constructs a decision-making component by encoding a policy that maps every state of interest to a recommended action that presumably maximizes a utility measure. The policy may be hard-coded by a programmer, or it may be acquired automatically through experience, given a suitable representation and value inference mechanism.

A common approach to learning a good policy is to associate a value with each state in the decision-making domain. These values can be adjusted through a variety of reinforcement learning methods. As experience is gained, the values come to approximate the utility of being in a given state. One can define a policy in terms of this approximate value function V by stating that the decision maker is to select the action that produces the successor state with the highest utility according to V. In nontrivial domains, the function V may be highly irregular over the state representation, which will tend to make it more difficult to learn.

Does one need to learn such a V when the objective is to learn a good decision-making policy? For decision-making purposes, one needs to select an action. A value function V offers a means to an end, but is not itself an end. One does not need a value that approximates utility when the task at hand is to select an action. One needs only a policy that indicates a good action to select. It may well be that a simpler function, say R, will suffice if it causes at least equally good decisions to be made. The values in R approximate *relative* utility, while the values in V approximate *absolute* utility.

2 Approximating Absolute Utility

A variety of reinforcement learning methods exist for improving an approximate V over time. With temporal difference learning (Sutton, 1988), one can learn an optimal V using TD(0), subject to certain assumptions. The fundamental goal of temporal difference learning is to learn to predict the value of each state in the decision-making domain, which in this case forms the basis of a decision-making policy. For a goal state, one wants to predict the payoff for being in that state. For a non-goal state, one wants to predict the payoff that will ultimately be attained by obeying the policy. By approximating these payoffs well, one can judge which state should be selected.

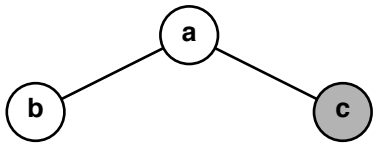

Figure 1. Value Backup

The TD(0) algorithm performs a 1-ply lookahead, evaluates each successor state according to V, and then chooses the best successor with high probability. One needs to make

apparently suboptimal choices from time to time in order to explore alternatives that may eventually come to be recognized as optimal. Knowing the V value of the best successor, the algorithm then adjusts the approximation of V so that evaluation of the parent state will return a value more like that of its best successor. For example, in Figure 1 if state c is best, then TD(0) would correct the value associated with state a to be closer to that of state c.

The TD(0) method is easy to program, requires no memory beyond that of the function approximator, and finds an optimal V, assuming that the state space is explored sufficiently often and that the function approximator can represent V with adequate precision. One needs to choose a suitable approximator. For a small state space, a lookup table is often practical. For a larger space, a linear combination of the state variables and a weight vector may work. For a nonlinear function, one may want to choose a multi-layer perceptron. The function approximator is an independent variable that is adjusted by the programmer. One must choose an approximator that will work well for the V that needs to be learned.

3 Approximating Relative Utility

When the objective is to learn a decision-making policy that is based on a value function, it may be possible to represent the same policy defined by a given V by using a simpler function R. If the simpler function can do the same job, it should be found more quickly, and it should require less memory.

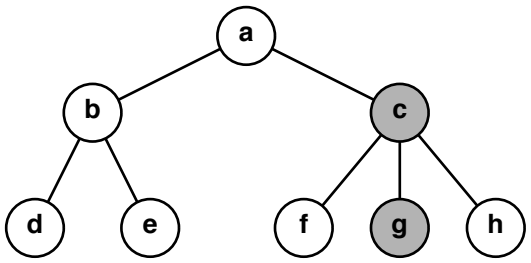

Figure 2. Preference Backup

Consider a formulation of temporal difference learning that backs up decisions instead of values. Instead of adjusting the approximation of V so that the value of the parent is more like the value of the best child, adjust R as needed so that its preference for successors among its children corresponds to its preferences among its grandchildren. This corresponds to a 1-ply lookahead in plies that correspond to decisions, instead of plies that correspond to values. For example, in Figure 2, a preference for state g among the grandchildren nodes implies a preference for state c among the children nodes.

How shall R be represented and updated? It can be represented as a numeric function, and approximated with any approximator that one might choose for a V. As before, one needs an approximator that is appropriate to the complexity of the function to be approximated. However, the error correction for R differs. Recall that for the V-learner, one associates the temporal difference error $V(a) - V(c)$ with state a. The state names a and c have been borrowed from Figure 1, but these each correspond to the vector of state variables that describes the state. This gives a positive error when $V(a)$ is too high.

However, for the R-learner, if the decision that would be made is currently wrong, then one associates an error with each of two states whose values have the incorrect relationship. One should have been higher than the other, and was not. One adjusts each one toward the other. Specifically, if a state c should be preferred to a state b, but is not, then one associates the error $\frac{R(b)-R(c)}{2} + \epsilon$ with state b, and the negative of that error with state c. These state names have been borrowed from Figure 2. This gives a positive error for state b when $R(b)$ is too high. Furthermore, even when the policy is correct according to the one-step lookahead, if the difference between the two relative values is closer than a margin β, then the two relative values are adjusted to move them farther apart. When one of the R-learner's immediate children is a goal state, no lookahead is needed. This is analogous to the V-learner not requiring lookahead when the environment supplies the payoff for a goal state.

4 Questions

One would like to know under what circumstances temporal difference learning of decisions works at all. Assuming it does work, one wonders whether optimality proofs that have been produced for learning a V with TD(0) can be extended to this case of learning an R (this question is not addressed here). A second set of questions concerns how the V-learner and R-learner compare. What advantages or disadvantages does the R-learner possess?

The next three sections describe two experiments that shed some light on these questions. The first experiment compares the two approaches for a simple grid-world task, in which the function approximator is a lookup table. The objective is to verify that backing up decisions during on-line decision-making works. For the second experiment, a different function approximator is employed, which is described below. The second experiment uses a task for which the true value function is highly nonlinear in the input variables. Several measurements are taken for each of the learners during the sequence of trials.

5 Grid World

The first experiment uses a simple task to illustrate that temporal difference learning using backed up preferences does indeed work. The task environment consists of an 8x8 grid world, as shown in Figure 3, with two goal states, and five cells containing obstructions that make them unvisitable. At the beginning of a trial, the learner is started at a randomly chosen empty cell, and can move north, south, east or west, one cell at a time. Any attempt to move outside the grid borders or into an obstructed cell has no effect. The trial is complete when the decision maker moves to one of the goal states, at which point the environment provides a payoff of 1. The task is to learn an optimal policy for this environment, i.e. a policy that leads the decision maker to a goal state in a minimal number of steps.

Two learners were set loose on this task. Each learner uses temporal difference learning, but as described above the V-learner backs up values, and the R-learner backs up preferences. The V-learner uses a discounting factor $\gamma = 0.9$ to give a greater payoff for a shorter path to a goal. The V-learner attained an optimal policy after thirty-eight trials, and the R-learner achieved an optimal policy after nine trials. Each learner used the same random-number seed, so the progression of potential trials is identical for each.

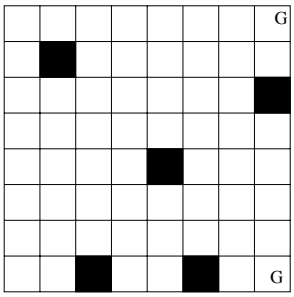

Figure 3. Grid World Task

One could make larger grids and do multiple runs, but the objective of this simple task was to indicate that backing up preferences in an on-line manner works. Whether this method will converge to an optimal policy in general is presently unknown.

6 The ELF Function Approximator

The next experiment uses a more sophisticated function approximator, described here. As mentioned above, the choice of function approximator is orthogonal to our focus here. Any function approximator with sufficient precision will do. The ELF function approximator (Utgoff, 1996) is chosen here principally because it can adjust the complexity of its model dynamically, eliminating the need for multiple runs. The features that it constructs are easy to interpret, and the number of them that are created provides a measure of the complexity of the function that is being approximated.

ELF is a nonlinear function approximator that constructs features as necessary while it updates its representation of the function that it is learning. The function is represented by two layers of mapping. The first layer maps a boolean vector representing the state to a set of boolean feature values. The second layer maps the features to a single scalar value by combining them linearly with a vector of real-valued coefficients called weights. Though the second layer is linear in the boolean features, the boolean features are nonlinear in the state (input) variables.

Table 1. Matching of Pattern and State

Pattern	State	Match
#	1	true
#	0	true
0	1	false
0	0	true

Each feature is a set cover over the state space. The cover is defined by a pattern vector with as many components as there are state variables. Each component of a pattern has either the value '#' or the value '0'. A '#' matches any (either) of the possible values of the corresponding state variable, while a '0' in the pattern matches only a '0' value. A table depicting whether a component matches is shown in Table 1. The pattern of all '#' covers every state in the decision-making domain because the pattern matches any instance at every

component. The pattern of all '0' covers the one state in which all the state variables have value '0'. One pattern is strictly more general than another if and only if it covers all the states covered by the other.

For every feature in the first layer, there is a corresponding weight in the weight vector of the second layer. It is convenient to think of each feature's corresponding weight as being a facet of the feature. This is because the set of features changes dynamically as ELF updates its approximation. When a feature is constructed or deleted, so too is the weight that corresponds to it.

Initially, the function approximation consists of the one most general feature, with a weight of 0. To evaluate a state, one computes the linear combination of the feature values and feature weights. A feature has value 1 if it covers the state, and value 0 otherwise. Thus, the initial approximation returns 0 for every state. To update the approximation, one provides ELF with a stream of pointwise errors, where a point is a state. ELF uses the LMS rule (Widrow & Hoff, 1960) to adjust the weights of those features that matched the state. Those that did not match have value 0, and those weights do not change.

Of course the approximation with the one most general feature is rarely adequate. One can repeatedly adjust its single weight, attempting in effect to fit a constant function to the points. ELF keeps track of which feature is having the greatest difficulty in fitting, and furthermore associates this difficulty with each of the state variables. When ELF determines that adjustment of the weights has ceased to be productive, it adds a new feature that is a specialized copy of the feature that has been having the greatest difficulty in fitting. The copy is specialized by changing a '#' in its pattern to a '0', thereby customizing the set of states covered by the feature. Note that every feature except the most general one is nonlinear in the input variables because it has value 0 for the states it does not cover, and the value of its weight for the states that it does cover.

ELF constructs features as needed and where needed, driven by the points and errors that it observes. Each feature has a logical interpretation as defined by its pattern and the input variables. ELF deletes features (but never the most general feature) whose weights have become relatively stable close to 0. ELF is quite direct in constructing useful features. Hence, the number of features that it constructs and retains provides a good indication of function complexity.

7 Eight Puzzle

The eight-puzzle is a sliding tile puzzle that contains eight square tiles in a flat space that could accomodate nine tiles in a 3x3 arrangement. By having just eight tiles, there is an empty location, into which one can slide one of the adjacent tiles. There are 181,440 states, including a single goal state consisting of all tiles in row order (the tiles are numbered 1-8), with the empty location in the lower right corner. The longest path to this goal state requires 31 steps, providing 32 different utility values. The exact value function contains 25,077 local maxima (states from which every decision is equally good) and one minimum. A regression tree based on minimizing variance of its blocks contains 114,114 leaves. This function is difficult to compress.

A V-learner and an R-learner were each turned loose on learning a good policy by way of

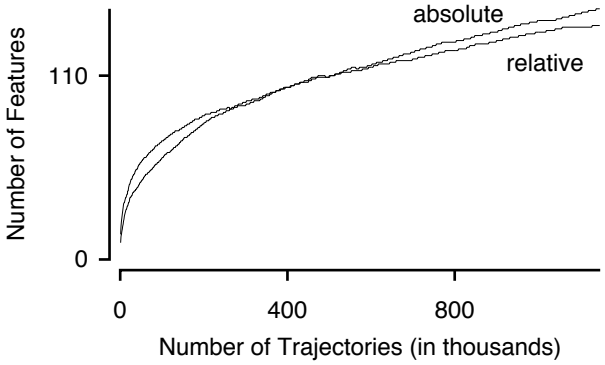

Figure 4. Function Complexity

making repeated trials in the task domain. To expedite the learning, each learner was told an optimal move for each state, meaning that the inferred error at each step was better than would be available otherwise. This use of error-free trials should not be necessary, e.g. the Grid-World task above, but was done to speed the process in order to estimate the longer term effects earlier. It is very unlikely that a learner will stumble onto the goal state early in its training, so some form of assistance would be needed anyway, such as a deeper search.

Table 2. Twenty Features of V

Weight	Feature
-11.489	
-5.258	∼at(1,9), ∼at(4,1), ∼at(4,4)
5.109	∼at(4,4), ∼at(7,4)
-4.937	∼at(4,4)
4.237	∼at(7,8), ∼at(8,7)
3.962	∼at(1,9), ∼at(3,9), ∼at(7,9), ∼at(8,9)
-3.712	∼at(1,9), ∼at(3,9), ∼at(5,9), ∼at(7,9), ∼at(8,9)
3.607	∼at(1,4), ∼at(1,9), ∼at(4,1), ∼at(4,4)
3.533	∼at(8,7)
3.326	∼at(b,8)
3.027	∼at(3,3), ∼at(6,3)
-2.981	∼at(b,2), ∼at(b,6)
-2.908	∼at(b,1), ∼at(b,8)
-2.839	∼at(3,7), ∼at(8,7)
2.836	∼at(3,3), ∼at(3,6)
2.758	∼at(1,1)
2.718	∼at(1,1), ∼at(2,4)
2.683	∼at(7,7), ∼at(8,4)
2.680	∼at(6,6)
2.677	∼at(6,8), ∼at(b,6)

Figure 4 shows the number of features in each of the approximated functions, with the curve labelled 'absolute' corresponding to the V-learner, and the curve labelled 'relative' corresponding to the R-learner. The two functions have similar complexity by this measure. One might expect the R-learner to have a smaller number of features, but in this case the learning is still in progress. It is likely that the R-learner is simply further along in acquiring a good policy.

Table 2 shows the twenty features of V that have the largest magnitude weights, as does Table 3 for R. The feature interpretation is determined by printing the negation of each binary state variable for which there is a '0' in the feature's pattern. This is shown in the table as a list of ∼at(tile,location) conjuncts for each feature, with the blank indicated by a 'b'. The weights for the V features indicate absolute utility, while the weights for the R features indicate relative utility, making their units different. One can see that these features of the two different learners are different, as are the entire sets (not shown here). This is one

indication that the learners are learning different functions.

Table 3. Twenty Features of R

Weight	Feature
-56.224	
10.557	∼at(3,3)
7.001	∼at(8,7)
6.400	∼at(1,1)
5.950	∼at(2,2)
4.314	∼at(7,7)
4.288	∼at(1,1), ∼at(1,4)
-4.179	∼at(5,5)
4.089	∼at(4,4)
4.047	∼at(2,6), ∼at(3,3)
3.872	∼at(6,8), ∼at(8,6), ∼at(b,6), ∼at(b,8)
-3.435	∼at(3,7), ∼at(8,7)
3.146	∼at(3,6), ∼at(6,3), ∼at(6,6), ∼at(b,3)
-2.856	∼at(3,3), ∼at(7,3)
2.667	∼at(8,8)
2.594	∼at(1,2)
2.523	∼at(7,8), ∼at(b,8)
-2.432	∼at(2,2), ∼at(2,6), ∼at(3,3)
-2.407	∼at(5,2), ∼at(5,9)
-2.403	∼at(2,9), ∼at(b,8)

The V-learner infers an error in V at each state, delivering an error correction to the function approximator each time. The R-learner detects whether the current relative value is wrong with respect to a one-step look ahead. If the policy is correct and the relative values differ by at least β ($\beta = 1$ here), then no error correction is delivered to the function approximator. Otherwise, a pair of error corrections is inferred. One could view this pair of corrections as a single composite correction, but for simplicity it is counted here as a pair of corrections. Which learner performs more total corrections? Figure 5 shows that the R-learner requires more corrections for a brief period, but then requires far fewer corrections thereafter. This is useful because one can spend less time updating the approximation by virtue of providing it with fewer error corrections.

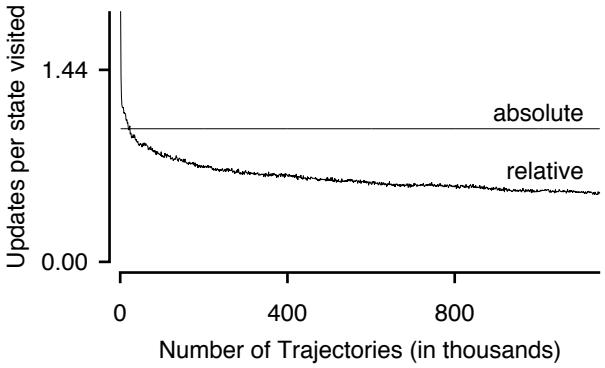

Figure 5. Error Corrections

Finally, with which policy is the probability of incorrect decision lower? For the 8-puzzle, there is a single goal-state, and a start-state is chosen at random from the entire state space. Thus states near the goal tend to be visited more often than those far away. A probability distribution over the state space was accumulated during training by counting the number of times each state was visited, and dividing each count by the total number of these counts for all states. To compute the probability of making an incorrect decision when following the policy, every state was checked. If any successor state with a best value (allowing for ties)

was not optimal, then the policy was scored with an error for that parent state. The overall probability of making a suboptimal decision (scored this way) is the sum of the probabilities for those states in which the policy could recommend a suboptimal move. Figure 6 shows this probability for the two learners. One can see that the policy of the R-learner is more accurate than that of the V-learner, and the accuracy advantage is obtained early and retained. The figure also includes the same measure (probability of error) when using the sum of the individual tile manhattan distances from tile goal as the value function.

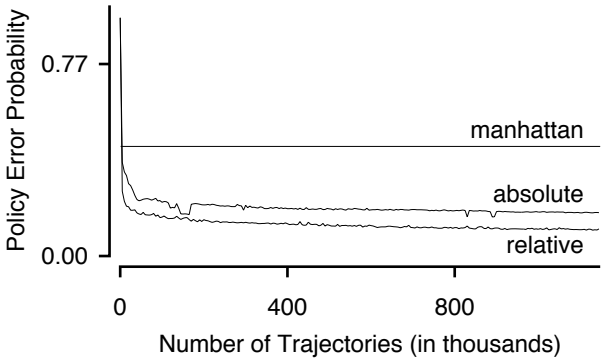

Figure 6. Policy Error

8 Discussion

Learning a relative value function can be applied easily to learning from passive observation, or to accepting an expert's advice. When an expert makes decisions, it is possible to infer that the choice made by the expert is to be prefered to those alternatives that the expert did not select. This can be useful when there is no particular goal state or payoff. One can infer the preference and update R at each step. For example, Broos and Branting (1994) employed a form of preference learning to capture telescope scheduling preferences among a set of regular telescope users. Tesauro (1989) implemented a backgammon program (prior to TD-Gammon) that learned from expert preferences in move selection. That program did not backup decisions, but instead learned in a supervised setting. It used a network in which two states were presented simultaneously, doubling the input dimensionality.

It is not necessary to learn either absolute or relative utility alone. One can learn from payoffs and preferences alike (Utgoff & Clouse, 1991), depending on which information is available at the moment. In effect, payoffs allow one to ground the units of measure of the approximated function in absolute utility, while preferences allow one to adjust the relationship among states that are available at a decision point. While Utgoff and Clouse considered only a linear function approximator, it has been shown here how to extend preference learning to the general case of temporal difference learning, and learning of a nonlinear relative utility function.

There is considerable on-going research on function approximation. A popular approach is to use a multi-layer perceptron (Rumelhart & McClelland, 1986), with sigmoid features and backpropagation of error. This approach adjusts its feature definitions by tuning their adjustable parameters. A constructive approach is available that dynamically adds new

features (hidden units) during learning (Ash, 1989). Other approaches for dynamically adding features include cascade correlation (Fahlman & Lebiere, 1990), meiosis networks (Hanson, 1990), node splitting (Wynne-Jones, 1992), and growing-cell-structures (Fritzke, 1993). ELF was chosen for the 8-puzzle experiment because it lends itself to inspecting the features, but a different approximator could have been chosen instead.

9 Conclusions

In principle, one does not need to approximate absolute utility in order to acquire a good policy. An alternative is to approximate relative utility. One can backup preferences (relative utility) via one-step lookahead in a manner analogous to backup of values (absolute utility). The approach of learning relative utility may be better suited to certain decision-making applications. A function that represents relative utility may be simpler than a function that represents absolute utility. For each of the experiments described above, a more accurate policy was learned more quickly when learning relative utility. More comparisons are needed, as are experiments that combine learning of both absolute and relative utility.

Acknowledgments

Nathan Sitkoff and Amy McGovern provided helpful comments.

References

Ash, T. (1989). Dynamic node creation in backpropagation networks. *Connection Science, 1*, 365-375.

Broos, P., & Branting, K. (1994). Compositional instance-based learning. *Proceedings of the Twelfth National Conference on Artificial Intelligence* (pp. 651-656). Seattle, WA: MIT Press.

Fahlman, S. E., & Lebiere, C. (1990). The cascade correlation architecture. *Advances in Neural Information Processing Systems, 2*, 524-532.

Fritzke, B. (1993). Kohonen feature maps and growing cell structures: A performance comparison. *Advances in Neural Information Processing Systems, 5*, 123-130.

Hanson, S. J. (1990). Meiosis networks. *Advances in Neural Information Processing Systems, 2*, 533-541.

Rumelhart, D. E., & McClelland, J. L. (1986). *Parallel distributed processing.* Cambridge, MA: MIT Press.

Sutton, R. S. (1988). Learning to predict by the method of temporal differences. *Machine Learning, 3*, 9-44.

Tesauro, G. (1989). Connectionist learning of expert preferences by comparison training. In Touretzky (Ed.), *Advances in Neural Information Processing Systems.* Morgan Kaufmann.

Utgoff, P. E., & Clouse, J. A. (1991). Two kinds of training information for evaluation function learning. *Proceedings of the Ninth National Conference on Artificial Intelligence*

(pp. 596-600). Anaheim, CA: MIT Press.

Utgoff, P. E. (1996). *ELF: An evaluation function learner that constructs its own features*, (Technical Report 96-65), Amherst, MA: University of Massachusetts, Department of Computer Science.

Widrow, B., & Hoff, M. E. (1960). Adaptive switching circuits. *Convention Records of the Western Conference of the Institute for Radio Engineers* (pp. 96-104).

Wynne-Jones, M. (1992). Node splitting: A constructive algorithm for feed-forward neural networks. *Advances in Neural Information Processing Systems* (pp. 1072-1079).