Feature Function Learning for Value Function Approximation

Paul E. Utgoff

Technical Report 96-09
January 20, 1996

Department of Computer Science
University of Massachusetts
Amherst, MA 01003

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

Keywords: value function approximation, automatic feature construction, feature function

Contents

1 Introduction **1**

2 Value Function Approximation **1**

3 Existing Approaches **3**

3.1 Back Propagation 3

3.2 Random 4

3.3 Constructive 4

3.4 Evolutionary 5

4 A Specific-To-General Search **5**

4.1 Agent Environment 5

4.2 Representing States and Features 6

4.3 Feature Function Search 7

4.4 Empirical Results 8

5 Discussion **9**

Abstract

To represent and learn a value function, one needs a set of features that facilitates the process by describing sets of states that share intrinsic properties. We visit existing approaches to automatic feature construction for value function approximation, and note important strengths and weakness of each. We offer an alternative approach that addresses some of these known weaknesses. Finally, we observe that searching for a good set of features is yet another form of biased search.

1 Introduction

A fundamental paradigm for implementing problem solvers is the form of heuristic search in which an evaluation function is used to estimate the payoff or cost that will ultimately be attained from a given state. With a perfect evaluation function, greedy search in the state space will produce an optimal solution with respect to what the evaluation function is measuring. To the degree that the evaluation function is imperfect with respect to identifying an optimal state, lookahead search is required to ensure optimality. When the required amount of search is prohibitively expensive, one searches less than is otherwise required, obtaining a good but not necessarily optimal solution. When building a problem solver in this paradigm, the implementer is faced with the problem of constructing a good evaluation function. It is well known that the representation of problem states is a critical factor that affects the ease of constructing an evaluation function and the accuracy it can attain as an estimator of ultimate payoff.

The study of methods for automatic construction of good evaluation functions receives much attention in the reinforcement learning community, where the construction task is known as *value function approximation.* A variety of methods are being actively pursued, including TD(λ) learning (Sutton, 1988) and Q-learning (Watkins, 1989). These two methods have been shown to find optimal value functions under certain conditions, and the processes modelled in these methods have been shown to be iterative Monte Carlo versions of dynamic programming (Barto, Bradtke & Singh, 1995). Although we have working methods for acquiring the true optimal values of states, we know much less about how to represent states and how to construct features in order to obtain good generalization and compactness.

The problem of how to map the primitive state representation to another representation that facilitates value function approximation is the focus here. This is a fundamental problem that remains largely unsolved. We visit previous approaches, point out their strengths and weaknesses, and then offer a new approach that attempts to draw on the strengths while reducing the weaknesses. The change of representation is accomplished in the usual way by mapping the state representation to a feature representation. We are concerned with how a good feature representation can be created automatically. We make a number of assumptions along the way so that we can concentrate directly on this problem.

2 Value Function Approximation

The goal of value function approximation is to produce a function V that maps each problem state $\mathbf{s}_i$ to its correct value with respect to what is being measured. Often such a V is not attainable but, as is well known, one does not need a perfectly accurate V when its

sole use is in a heuristic problem solver as described above. If V assigns a more desirable value to a preferrable state, then that V is sufficient for the problem solver to be able to distinguish which states are better than others, which is all that is needed for identifying an optimal next state. One can see that there are many versions of V that cause the same relative rankings among the states. One can expect to find a V that is optimal with respect to state selection long before, and perhaps without, attaining a highly accurate V.

We shall assume that a primitive representation of each state is given, and that this is the only primitive representation of state that is available. It is up to the problem solver, hereafter called the *agent*, to produce the best value function $V(\mathbf{s})$ that it can. We shall also assume that the agent is repeatedly solving problems, thereby obtaining information about which state sequences lead to which payoffs. Using temporal difference learning, or some other value inference method, the agent ascribes values to certain states, thereby providing points and function value estimates (point estimates) in state space. We assume that the agent is in an online setting, in which old point estimates are used just once and then discarded, and new point estimates are produced during future problem solving and value inferencing.

The primitive state representation s_i for state i can be viewed as a collection of measurable values. Typically each primitive component is either a real-value, or a binary value represented by two distinct real values. We shall refer to the set of primitive state values as the *inputs*, and we shall refer to the value of function V for a given set of inputs as the *output*. How can we represent a mapping from the inputs to the output value? We exclude the option of using a fine-grained lookup table because it does not scale well to large state spaces. If a weight is to be learned for each state, then the possibility of generalization is eliminated because learning the weight for one state does not help with learning the weight for another. Furthermore, a large state-space would require a large table. If one uses a lookup table with some amount of coarseness, i.e. in which more than one state is mapped to a given element in the table, then it is really just an alternative grid-like scheme for defining features. We are interested here in identifying features that describe intrinsic properties of set of states.

Instead, one typically maps the inputs to an intermediate representation consisting of one or more *features*, and then maps the feature values to the output. We shall refer to the set of features as the feature function, and shall denote it by $\mathbf{F}$. Thus, the value function V consists of two levels of mapping, indicated by $V(\mathbf{F}(\mathbf{s}_i))$. One could introduce additional levels of mapping, but we shall focus on this basic form. This kind of mapping to features is quite common. For example, consider the Manhattan-Distance evaluation function for the 8-puzzle. The primitive state representation consists of the tile locations, the feature function consists of the eight features in which each feature computes the city-block distance of its tile in its present location from its goal location, and the value function consists of the sum of the eight feature values.

We now have the problem of learning two mappings, $\mathbf{F}(\mathbf{s}_i)$ and $V(\mathbf{F}_i)$, where $\mathbf{F}_i$ is shorthand for $\mathbf{F}(\mathbf{s}_i)$. Because we are primarily concerned with the representation problem, we focus on the problem of finding $\mathbf{F}(\mathbf{s}_i)$ automatically. We shall assume that V is a linear combination of its inputs, which are now the feature values or feature outputs, and that the weights of this linear combination are updated by a simple error-correction rule associated with TD learning. Note that although V is a scalar output, $\mathbf{F}$ has multiple unordered

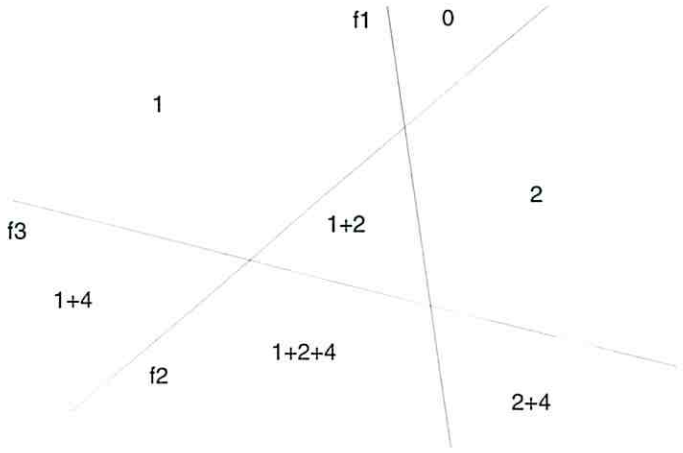

Figure 1: Seven Regions from Three Features: Feature $f1$ has weight 1, feature $f2$ has weight 2, and feature $f3$ has weight 4.

components.

This leaves us with the problem of finding a good feature function $\mathbf{F}$ that facilitates learning a good V under the stated assumptions. Thus we find ourselves in a space of feature functions, not knowing ahead of time which may be better than another, giving us a large search problem to solve.

3 Existing Approaches

Several approaches already exist for finding a feature function $\mathbf{F}$ automatically, and we visit these in order to see their strengths and weaknesses.

3.1 Back Propagation

The most notable method for automatically finding an $\mathbf{F}$, while simultaneously learning V, is back propagation of error through a feed-forward artificial neural network (Rumelhart & McClelland, 1986). In a two-layer network, the hidden layer of units corresponds precisely to the feature function $\mathbf{F}$, and the output unit corresponds to the value function V. Upon adjusting the weights of the output unit, the error is then apportioned to the hidden units, and each is updated individually in a similar manner.

Each hidden unit describes a subset of the inputs by placing a hyperplane through the state space. The unit outputs a high value (near 1.0) for states on one side, and a low value (near 0.0) for the remaining states. For example, consider an x-y space with three hidden units (features), as depicted in Figure 1. Each combination of feature values forms a distinct set of inputs to the output unit, which means in this case that seven different regions are describable, each having its own output value. In general, for n inputs, the number of distinct regions increases by a factor of n for each new feature that is added (assuming non-parallel feature boundaries). Since the value of each region is determined by combining the weights of the features that are present, one cannot expect the seven different regions to have unrelated values. Only the values available through feature combination are possible,

but the number of distinct values is exponential in the number of features.

The weight adjustment process for the units is driven by gradient information. One naturally questions whether hill-descending is sufficient for finding a good feature function. Although back propagation often does well, we know that it has shortcomings. First, it is not known how to choose a suitable number of hidden units. Second, one must initialize the hidden unit weights randomly so that there is initially a rich set of features. It is assumed that training will cause these feature definitions to change to better regions, as guided by the negative of the gradient of the error, which is not always the case. Finally, one still must often resort to hand-crafting of features in order to enable learning of V suitable well (Tesauro, 1992). Though it may be practical at times to accelerate the learning process by providing good features manually, our goal is to automate this process, so impediments need to be noted. One must start the process somewhere, and we have assumed that the state representation is given. If the information needed for defining useful features is present in the state representation, then we should hope that the feature construction method will find those useful features.

3.2 Random

An approach that is remarkably simple was shown by Sutton and Whitehead (1993) to do very well, outperforming back propagation. Their search for $\mathbf{F}$ consists of generating it at random in one step, and then keeping it. In terms of a neural network, they generate a fixed number of hidden units with random weights, and never let these weights change. Learning the value function V consists entirely of adjusting the weights of the output unit. With a large set of hidden units, one can expect that some of them will be useful. Recall that it is the combination of features that enables the value function to assign so many distinct values. Sutton and Whitehead report that their random representation approach using 100 features learned a more accurate value function than back propagation nets using 4, 40, and 400 adjustable hidden units. This is additional evidence that back propagation gets stuck during learning.

3.3 Constructive

Wynne-Jones (1992) presents an approach called *node splitting* that detects and attempts to repair an inadequate hidden layer of a feed-forward artificial neural network. His system detects when the hyperlane of a hidden unit is oscillating, indicating that the unit is being pushed in conflicting directions in feature space. His method splits such a unit into two, and initally sets them apart from each other by an explicit alterning of the weights. The goal is to set the units apart along the most advantageous axis. Although this approach sometimes works well, Wynne-Jones observes that the units often work back toward each other instead of diverging. He reports promising results when this technique is applied to a gaussian mixture model.

Other constructive approaches have been devised, including cascade correlation (Fahlman & Lebiere, 1990), and dynamic node creation (Ash, 1989). Each addresses the problem that weight adjustment of a fixed hidden layer is not generally sufficient for finding an adequate mapping of inputs to desired output.

3.4 Evolutionary

An evolutionary approach that draws heavily on domain knowledge is Fawcett's (1993) Zenith system. The programmer provides a declarative definition of the problem-solving domain, consisting of a theory that defines the state space, the move operators, and the goal that the agent endeavors to achieve. The Zenith system then applies its own transformation operators that synthesize features from this declarative knowledge. Fawcett maps a logical term or clause to a numeric value by computing the number of bindings for which the logical expression can match the state. The main loop consists of generating feature definitions, letting each one 'live' for a period of time while a coefficient is learned for it in a linear combination, and keeping the good features and discarding the bad. There is considerable overhead in specifying a complete and correct domain theory. In the game of Othello and in a telephone switching application, Zenith created and retained features that were previously known to be good, and it created new useful features that were entirely original to Zenith.

A different evolutionary approach is exemplified by Levinson and Snyder's (1991) MORPH system for learning a value function for the game of chess. The feature function is represented as a set of patterns, each with an associated weight. Each pattern is an abstraction of a portion of a state, and is represented in a high-level hand-engineered pattern language based on attacking and guarding relationships among the chess pieces in the pattern. Their system produces a pattern for the state before a move and the state after a move, using a form of explanation-based learning to account for the differences. Their weight updating method is loosely based on temporal difference learning. The authors report disappointing but intriguing results for their system when learning with GNU-Chess as its opponent.

MORPH and Zenith have in common the idea that the feature function is a collection of weighted patterns, each of which survives or dies based on its utility to the value function. Search for the feature function is accomplished by modifying the set of features.

4 A Specific-To-General Search

We present a value approximation method in which V is updated by TD(λ), and $\mathbf{F}$ is updated by adding, deleting, or generalizing individual features that collectively constitute $\mathbf{F}$. To ground the discussion, we explain the method with respect to an implemented prototype called TQ, which learns a value function for the game of TicTacToe. This domain requires a non-trivial feature function, but has a relatively small state space.

4.1 Agent Environment

The TQ program repeatedly conducts trials, selecting moves for each player. A trial consists of playing a single game and receiving a payoff of 1 for a win, 0 for a draw, and -1 for a loss. Since TQ models two agents, it learns from each agent's move sequence and payoff. Although this is not an essential part of the design, one does obtain twice as much experience per game. A game provides an environment in which an agent must solve a problem, and the adversary is simply part of the environment. Each agent learns from the sequence of moves that it makes in its environment. There is no initial knowledge of the three-in-a-row concept. Instead, the agent simply makes moves, and is told when the game is over and what the payoff is.

Feature	State	Match
1	1	True
1	0	True
0	1	False
0	0	True

Figure 2. Truth Table for Match Predicate

Each agent uses a 1-ply search to enumerate the possible successor states and evaluate each one using the current feature function **F** and value function V. With probability $\frac{29}{30}$, the agent selects the move that produces the state with the best value according to V. Otherwise, the agent selects its move at random. This probability was chosen in a way that causes an average of one random move in every three games played. The value V of a state is an estimate of the ultimate payoff that will be achieved by following this mostly greedy move selection policy. One needs a certain amount of exploration in order to ensure that better alternatives can be noticed.

The value function is updated using linear TD(0.9). This is done for each agent by maintaining an eligibility trace of the feature values. The trace decays by a factor of 0.9 at each time step. Then, the difference between the 1-ply backed-up value and the value of the current state is computed. The weight changes that would be made at each step are accumulated until the trial is over. Then V is updated by adjusting its weights according to the accumulated changes. Finally, the feature function **F** is adjusted, as described below. The cycle then begins anew with a fresh trial.

4.2 Representing States and Features

One needs to choose a representation for the inputs and the features. The method we describe depends on a bit-vector representation, which we describe for the TicTacToe case. For each of the nine squares of the board, represent the contents of the square by a 3-bit value. If the square contains an 'X', then the X-bit is 1 and the other two bits are each 0. Similarly, define an O-bit and an empty-bit. For any particular state, exactly one bit in each group will be set, for a total of nine set bits. This representation of 27 bits has excess capacity, but the ability to represent combinations is important. More generally, one needs as many bits per cell as there are possible distinct contents of that cell. For example, in checkers one would need 32 cells of five bits each.

A feature is also represented as a bit vector of the same length. For matching purposes, when a bit is set in a feature's bit vector, it means that the corresponding bit in a state's bit vector is permitted to be set. Any state for which all of its set bits are permitted in the feature's bit vector *matches* the feature. Thus a feature's bit vector describes a set of state bit vectors that will match it. In this way a feature defines that set of states that it covers. A feature describes any state that matches it. Figure 2 shows the truth table for the match predicate.

The bit encoding provides an efficient matching procedure. Notice that a state matches a feature unless at some bit position(s) the feature's bit is 0 (not set) and the state' bit is

1 (set). The value of the feature is 1.0 if the state matches it, and 0.0 otherwise. The bit-vector representation for states and features, and the associated matching predicate, define an implicit partial ordering over the space of features. A state description is identical to a most specific but non-empty feature description, and the bit vector in which every bit is set is the most general feature description. One feature is more specific than or equal to another if and only if it matches it.

4.3 Feature Function Search

TQ starts with the empty feature function, which means that the value function is defined solely by its bias weight (the constant term of a linear combination). The value function is a constant function, whose most accurate single value comes to be approximated as well as possible through the TD updating that occurs. The program needs to notice when its feature function is inadequate for learning a sufficiently accurate value function, and it needs to take action to improve it. The feature function must change, and this is done by changing the set of features that constitute it. We need to add, delete, and revise individual features in a way that leads to an adequate feature function.

A new feature is added to **F** whenever the desired value of a state is non-zero and the state does not match any feature in **F**. The corresponding weight in V for the new feature is initialized to 0. The added feature has its bit vector initialized to the state's bit vector. We shall refer to such a feature as a singleton. We also add a new singleton to **F** whenever it is not already in **F** and the difference between the backed-up value of the state (from the 1-ply lookahead) and the local value of the state (from V) is too large in magnitude. For TQ, this occurs when the difference has magnitude greater than 0.01. Such a condition indicates that the features currently present in **F** are inadequate.

An old feature is deleted from **F** whenever it becomes useless, which occurs when it is only seldom matched by a state, or when the magnitude of its associated weight for V is near 0. In these cases, the feature contributes little to the value of V. For TQ, a feature is considered to be seldom used if it has not been matched by a state in the previous 500,000 attempts. Using 1-ply search in the game of TicTacToe, the largest number of match attempts per game is 9+8+...+1, which is 45 for both players combined. Thus 500,000 match attempts is on the order of 10,000 games. A weight for a feature that has magnitude less than 0.0001 is deemed to be too small to be useful.

We have discussed when and where to add a new singleton feature to **F**, and we have discussed when to eliminate a feature from **F**. When one lets TQ run with just these operations on **F**, it finds a feature function for which a suitable value function is also learned. However, the features in **F** are all singeltons, making **F** much like a fine-grained lookup table, which we wish to avoid. The number of features in the feature function is far less than the size of the state space because many suboptimal states are easily avoided. Nevertheless, we would still like to have features that cover larger sets of states. This is important for generalizing beyond the data and for reducing memory requirements.

One would like to generalize a feature whenever a single value for its weight is appropriate for each of the states that it covers. With weight adjustments occuring during learning of the value function V, and with features being created, deleted, and generalized, it is not clear

how to say when generalizing is a good idea. However, we do know that we do not want to generalize a relatively new feature whose weight is unlikely to be near its best value, and we do not want to generalize a feature whose weight continues to change repeatedly by a large amount. So, one may surmise that it may be productive to generalize a feature whose weight has been far from 0.0 for a long period of time. For TQ, a weight with a relatively large magnitude occurs only after a long period of time, so this can serve as a test for this condition. For TQ, any feature whose associated weight for V has a magnitude greater than 0.4 is generalized. However, a generalized feature is never added when a feature with the identical bit vector already exists in the feature function.

A potential problem with generalizing a feature with a large weight is that the generalization may be a poor one, introducing instability to its weight, and secondarily to other features that cover some of the same instances. This can be overcome by generalizing a copy of the feature, and leaving the original feature in the feature function, similar in spirit to node splitting discussed above. One then sets the weight of the original feature to half its value, and initializes the weight of the generalized feature to the same value. In this way, each state covered by the original feature is now also covered by the generalized feature, and the total contribution to the value function remains unaltered. However, if the generalized feature is good, then its weight will grow and the original's will shrink. If the generalized feature is bad, then its weight will shrink and the original's will grow. A feature with a low magnitude weight will eventually qualify for deletion and be removed. Thus, this mechanism implements a specific-to-general search of the feature space with pruning of unproductive paths. Only one of these two features can expected to survive.

A feature is generalized by randomly setting one of its bits that was not already set. Some choices may be better than others, but this is not known ahead of time. If the choice of bit was a bad one, then the generalization will eventually be removed. Subsequently a new one may be generated, again picking an unset bit at random. However, if the choice of bit was a good one, then the generalization will survive, and its predecessor will eventually be removed. The generalized feature might eventually become a candidate for further generalization. When a feature is generalized, and its weight is halved, it no longer immediately meets the criteria for generalization. The magnitude of its weight needs to grow large enough again in order for it to become a candidate for generalization again.

4.4 Empirical Results

Figure 3 shows the activity of TQ over a large number of tournaments, where each tournament consists of 500 trials. One observes that the total number of features in the feature function grows rapidly to over 3500, and then decreases to a number that fluctuates in the low 1000s. One also observes that TQ learns a reasonable value function quite early, but then compresses its representation over time. The peaks in the plot of games-not-tied correspond to periods in which TQ has a misleading value function for its 1-ply search. It takes time to improve the feature function and value function so that the mostly greedy policy performs well. The typical value near zero illustrates that TQ usually obtains a draw, which is the expected outcome for optimal play.

It is also informative to examine the features that TQ found. The minimum number of bits that can be set in a pattern is nine, and the maximum is 27. Inspection of the features

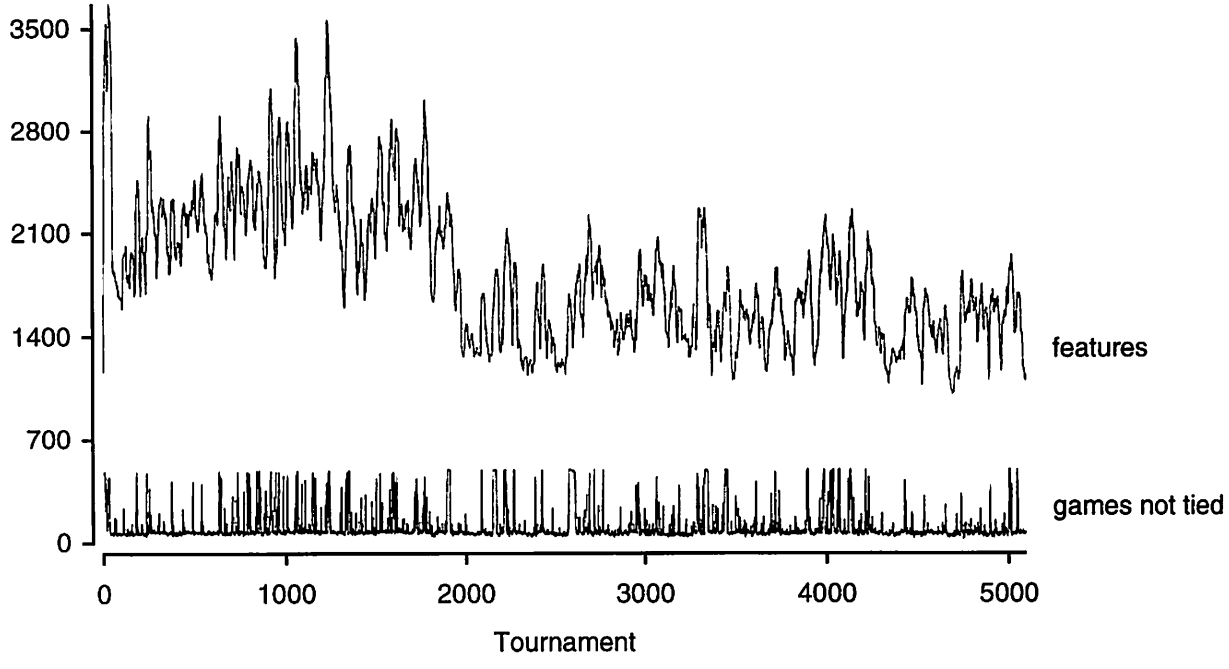

Figure 3. Features and number-of-games-not-tied vs. tournaments played

shows many different levels of generalization, including the single pattern of all 27 bits. Of course the single pattern of all 27 bits set covers all states and is therefore functionally equivalent to the bias weight of the value function V. This duplication is of no consequence. The 21-bit patterns that correspond to each of the three-in-a-line configurations are all present. In addition, many 9-bit patterns that correspond to mistakes in early play are also present. In short, TQ found the range of features that one would hope to see.

5 Discussion

TQ suffers from its need to revisit parts of the state space to promote generalization. The weight associated with a feature can grow to a large magnitude only by having its weight adjusted many times. In large state spaces, or with aggressive exploration, this will not occur because new states will be encountered a large part of the time. This causes features to be matched rarely, earn a low utility, and be deleted from the feature function. We plan to explore a general-to-specific approach much like TQ that will specialize where there appears to be potential for improvement. We have moved to the game of checkers so that we can experiment with a large state space.

When one sees a set of individuals represented as bit-vectors that are evolving over time, one is reminded of the work on genetic algorithms (Goldberg, 1989). It may well be such an approach can work here, but the crossover operator may not make sense in this setting. For TQ, the generalization operator causes two similar features (one has one additional bit set that the other does not) to be launched at the same time. If the generalization is good, the more specific feature will eventually be deleted, and if the generalization is bad, the more general feature will instead suffer this fate. A crossover operation would

produce a feature that may or may not prove useful. Given that the combination of bits is important collectively, it seems unlikely that a recombination of independently chosen bit vector segments will produce something useful. An important consideration is that a set of features has its weights adjusted collectively, and one needs to be somewhat wary of making huge steps in feature function space (Levinson & Snyder, 1991). Nevertheless, it would be quite simple to add this to TQ and conduct such an experiment. The search would of course no longer be specific-to-general in feature space.

Back propagation and other gradient-following methods in non-monotonic spaces can become trapped at local minima. Although back propagation makes good use of gradient information, we cannot expect back propagation to find the most useful features that are possible under the given the state representation and the rest of the learning context. Although back propagation is often described as a sufficient solution to the feature construction problem, that is a myth. Gradient information is a useful source for guiding the search, but it is just one such source, and there are well known problems associated with relying on it alone.

If one looks at how each feature is defined, and how the features are ordered in feature space, then one sees that a feature function learning method is yet another instance of biased search. When back propagation adjusts the weights of a hidden unit, it changes the set of instances that match that feature. The relationship between weight adjustment and state coverage of the feature affects the search in a fundamental way. The method we presented for TQ organizes the state space differently. The relationship between bit vector adjustment and state coverage of the feature also affects the search fundamentally. These adjustment/coverage relationships are different, and each one represents a bias for searching feature function space. It is unlikely that one is strictly better than the other in general.

The most distinctive aspect of the approach exemplified in the TQ approach is that it creates and adds features in those parts of the state space where additional descriptive resolution is required. This is accomplished in two ways. First, the method detects where the value function is too inaccurate and adds a most specific feature that modifies the value function for that particular state. Second, the method copies and generalizes features that have proved useful by virtue of having obtained a high magnitude weight. This approach effects a specific-to-general search in feature space that is controlled by a competition between the generalized feature and its predecessor.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. IRI-9222766. I thank Rich Sutton for many invaluable discussions and comments. I also thank Jeffery Clouse and Gunnar Blix for helpful comments.

References

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

Barto, A. G., Bradtke, S. J., & Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence, 72*, 81-138.

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

Fawcett, Tom E. (1993). *Feature discovery for problem solving systems.* Doctoral dissertation, Department of Computer Science, University of Massachusetts, Amherst, MA.

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley.

Levinson, R., & Snyder, R. (1991). Adaptive pattern-oriented chess. *Proceedings of the Ninth National Conference on Artificial Intelligence* (pp. 601-606). Anaheim, CA: MIT Press.

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.

Sutton, Richard S., & Whitehead, Steven D. (1993). Online learning with random representations. *Machine Learning: Proceedings of the Tenth International Conference* (pp. 314-321). Amherst, MA: Morgan Kaufmann.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning, 8*, 257-277.

Watkins, C.J.C.H. (1989). *Learning with delayed rewards.* Doctoral dissertation, Psychology Department, Cambridge University.

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