Constructive Function Approximation

Paul E. Utgoff
Doina Precup

Technical Report 97-04
January 21, 1997

Department of Computer Science
University of Massachusetts
Amherst, MA 01003

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

Abstract

Algorithm ELF is a nonlinear function approximator that constructs features as necessary while it updates its representation of the function that it is learning to approximate. Feature construction is guided by representational inadequacy that is associated with particular values of the base-level variables. The features have a logical form, making them easy to interpret.

Contents

1 Introduction **1**

2 Algorithm ELF **1**

2.1 Representation 1

2.2 Operation 2

3 An Artificial Task **6**

4 Checkers **7**

5 Related Work **10**

6 Convergence **11**

7 Summary **12**

1 Introduction

Numerical evaluation functions have become a mainstay in building systems that make choices. Many approaches have been devised that enable such systems to learn an evaluation function by representing it as a parameterized model, and then by adjusting those parameters, based on observed error. Many researchers are now addressing the problem of how to search the space of parameterized models automatically. A parameterized model is good to the extent that it facilitates finding quickly an evaluation function of high accuracy and low memory consumption.

An evaluation function v maps every element in its domain $\mathbf{X}$ to a real value. One needs a base level representation for the elements of $\mathbf{X}$, and a procedure for computing $\hat{v}(\mathbf{x})$, where $\mathbf{x} \in \mathbf{X}$. As always, the choice of base level representation has a profound effect on how well v can be approximated, as does the choice of a parameterized model. For example, if every element $\mathbf{x}$ is described by a single integer component, and v is related linearly to the integer value, and one chooses a model of the form $\hat{v}(\mathbf{x}) = mx_1 + b$, then one can expect to find values for m and b quite easily. With a different base level representation, or a different parameterized model, the problem of finding a good $\hat{v}$ may become difficult or impossible.

It would be convenient to be able to pick an obvious and easy-to-implement base level representation for the elements of $\mathbf{X}$. For practical reasons, a second level of representation is typically present, taking the form of a feature vector. Instead of computing $\hat{v}(\mathbf{x})$, one computes $\hat{v}(\mathbf{f}(\mathbf{x}))$. There is no loss of generality, as one could choose to define $\mathbf{f}(\mathbf{x}) = \mathbf{x}$. On the contrary, there is great advantage in this approach because one can map the easy-to-implement base-level representation to another that is better suited to the parameterized model under consideration. Furthermore, the representation for $\mathbf{f}$ can be manipulated by the algorithm, whereas it may be quite difficult to manipulate the base level representation directly.

One would like to have a function approximator that constructs its own features as necessary to represent an accurate approximation of the function to be learned. One would like to find an accurate approximation as quickly as possible, measured both in terms of human and machine time. Ideally, the algorithm would require little setup, and only a single run. Though not necessary, it would be nice to be able to interpret the features constructed by the approximator.

2 Algorithm ELF

The algorithm ELF (Evaluation Function Learner, with a twist) represents its approximation as a linear combination of its feature values and a set of corresponding adjustable scalar parameters, called weights. It adjusts the weights and it also varies the set of component functions in $\mathbf{f}$. Although $\hat{v}$ is linear in the features, each feature is nonlinear in the base-level variables.

2.1 Representation

The base-level representation consists of a set of Boolean variables. A domain element $\mathbf{x}$ is represented as a conjunction of particular variable values, and is therefore in $\{0, 1\}^n$. An application that uses the algorithm may need to apply a fixed transformation to map

non-Boolean variables to Boolean, but such a step is not part of the ELF algorithm itself. A discrete variable can be mapped to a set of propositional variables, and a numeric variable can be mapped to a set of discrete intervals, each one comprising a Boolean variable.

Table 1. Matching of Pattern and State

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

A feature consists of a set cover over the domain elements $\mathbf{X}$, and an associated weight. As features are constructed or deleted during learning, one needs to construct or delete the corresponding adjustable weights of the linear combination. For this reason, it is convenient to think of each feature's weight as being part of the features. When a feature is constructed, so is its weight, When a feature is deleted, so is its weight. Each feature evaluates to 1 if it covers the domain element $\mathbf{x}$ being evaluated, and 0 otherwise. Since the value of the function approximation is $\mathbf{W}^T\mathbf{f}(\mathbf{x})$, it can be computed simply by summing the weights for those features that cover the $\mathbf{x}$.

Whether a particular feature covers a particular instance is determined by a set definition, also called a pattern, that is associated with the feature. This set cover is defined by a pattern vector with as many components as there are base-level 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 base-level variable, while a '0' in the pattern matches only a '0' value, as shown in Table 1. The pattern of all '#' covers every domain element because the pattern matches any domain element at every component. The pattern of all '0' covers the one element in which all the base-level variables have value '0'. One pattern is strictly more general than another if and only if it covers all the domain elements covered by the other.

As the list of features grows and shrinks during learning, the parameterized model changes in the number of features and corresponding adjustable parameters. No other model classes are considered, but there is no loss of generality in terms of representation. One can show that when all possible features are present in the set of features $\mathbf{f}$, any possible assignment of values to domain elements can be approximated exactly. Although the approximated function is linear in the feature weights, the binary features are non-linear in the base-level variables because a feature has value 1 for some variables and value 0 for others.

2.2 Operation

The ELF algorithm is driven by a stream of point-wise errors, provided by the application or environment in which the algorithm is embedded. Each point-wise error consists of a particular domain element $\mathbf{x}$, and an error in the current approximation $\hat{v}(\mathbf{f}(\mathbf{x}))$. Any application that can infer an improved estimate for $\mathbf{x}$ can deliver such a point-wise error. When an error is received by ELF, it takes two steps. First it updates the adjustable weights (one per feature), and then it updates the features in $\mathbf{f}$ as necessary.

The weights are upated using the well known Widrow-Hoff rule (Widrow & Hoff, 1960) for iteratively minimizing the mean-squared error of the evaluation function. One computes an amount by which to alter the weights of those features that matched the instance. The update rule takes this form here because the features evaluate individually to 0 or 1. The amount by which to adjust each weight is the error divided by the number of features that matched, multiplied by a stepsize parameter α that is typically a fixed value less than 1.

Initially, the function approximation consists of the one most general feature, with a weight of 0. This initial approximation returns 0 for every element. As point-wise errors start to come in, the single weight of the single feature is adjusted in an attempt to minimize the mean-squared error. This amounts to trying to fit the points with a constant function. By itself, this would not be very good, but one can gather useful information while this vain attempt at fitting is taking place, and then use that information to add a useful feature that will help subsequent attempts.

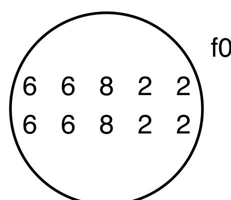

Figure 1. Target Values

Consider a simple illustration. Imagine that the space of domain elements is as depicted by the Venn diagram shown in Figure 1. The points are not shown, but some of their target values are. Some points should evaluate to 6, while others should evaluate to 8, while still others should evaluate to 2. As the weight of the most general feature f_0 is repeatedly adjusted, suppose that it settles to the value 5. Then the error associated with each of these points would be as shown in Figure 2.

While ELF is attempting to adjust the feature weights, initially just the one weight, it accumulates the error that is associated with each bit being on in an instance. With each feature, ELF maintains a separate vector of bit-errors that are accumulated by attributing the amount of correction that has been applied to the feature weight to each of the bits in the feature pattern whose corresponding bit was set in the domain element $\mathbf{x}$. This information is of central importance to the procedure for updating the features, but plays no role in updating the feature weights. This is much like taking an X-ray picture. One bombards the $\mathbf{X}$ space with error, which paints a picture as defined by the accumulated bit errors. These errors provide very useful information about those subsets of the domain to which one would like to be able to assign different values.

To accomplish assigning one value to some instances and not others, one constructs a new feature that covers just those instances. The procedure for doing this is described below. For now, assume that a new feature has been constructed that covers the 6s and the 8s. Through subsequent point-wise error adjustments, suppose that the original feature f_0 takes on a weight of 2, and the new feature f_1 takes on a weight of 7. Now the new X-ray paints the error picture shown in Figure 3. These errors are smaller than before.

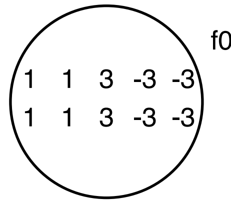

Figure 2. Error with One Feature

Finally, assume that a new feature f_2 has been constructed that covers the 8s and the 2s. Through more point-wise error adjustments, suppose that f_0 takes on a weight of 0, f_1 takes on a weight of 4, and f_2 takes on a weight of 2. Now a new X-ray would be transparent because the error is uniformly 0.

It is not possible in the pattern language for one feature to describe an arbitrary subset of the domain. One must define the set cover of the feature with a conjunctive pattern of '#' and '0' as described above. To the extent possible, one would like ELF to identify intrinsic properties shared by many instances, and associate an additive value with each. For ELF, each feature represents such an intrinsic property and its value.

To add a new feature, ELF identifies the feature that is least able to reduce the error that is attributed to it. In the initial approximation, there is the single most general feature that is trying to fit all the points with a constant value. Even after other features come into existence, the problem is still the same with respect to any one feature. The other features simply transform the values of the instances they cover. Every feature tries to fit the points it covers with a constant function. The feature has value 0 for those instances it does not cover, and the value of its weight for all others. One needs to find the feature that is fitting its points least well, given all the other features.

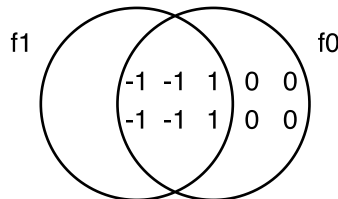

Figure 3. Error with Two Features

To identify the feature that fits its points least well, given the other features, one looks for the feature with the greatest spread between the maximum and minimum bit-wise error. This value is called the *warp* of a feature, because the bitwise errors indicate a desired direction in which to move the weight of the feature. The feature is not actually warped, as it has only its single scalar weight, but the bitwise errors can be seen as stresses on the feature weight related to different segments of the domain.

To change the set of features, ELF identifies the feature with the largest warp value, then identifies the '#' in the feature whose accumulated error is most different from the mean accumulated error for that feature. ELF then makes a copy of the feature, changes the '#' to a '0' in the copy, initializes the weight of the new feature to 0, and adds it to the set of

features in $\mathbf{f}$. By initializing the weight to 0, adding the new feature has no immediate effect on the approximated function. However, subsequent weight adjustments will use the new feature like any other other, moving its weight to a more useful value.

Every feature definition can be reached in principle, even though the only method for revising a feature definition is to specialize it. Whenever a new feature is added, the accumulated bit errors and several other bookkeeping variables are reset for all the features. However, the feature weights $\mathbf{w}$ are left untouched. It would be quite useless to allow features with identical patterns, so ELF prevents this by not considering any specialization that would produce a duplicate pattern.

The weights need enough point-wise error adjustments to reach the best attainable values, given the current features $\mathbf{f}$. How does one distinguish useful weight adjustments from useless? For ELF, weight adjustments are useful if new minimum or maximum weight values are produced sufficiently often or if a new minimum in a moving average of the absolute observed errors is produced sufficiently often. Otherwise the weight adjustments are useless.

One can expect interaction in adjusting of the weights due to overlap of the features. Until the weights and the error stabilize, one does not really know which feature is most warped for the current set of features. There are extra bookkeeping variables that are maintained in order to determine when the weights have stabilized and when the error has stabilized. If no new minimum or maximum value of weight has been established during a long sequence of weight updates, then the weights are assumed to be relatively stable. Similarly, if no new minimum error has been establised during a long sequence of weight updates, then the error is assumed to be relatively stable. If the weights and the error are all relatively stable, then ELF will take action to specialize the most warped feature.

Finally, one deletes a feature whose weight has been near to 0 for an extended length of time. This is accomplished by considering features for deletion only at the same time that one might specialize a feature. For a feature that has both its minimum and maximum observed weight near 0, the feature is deleted. However, the most general feature is never deleted. This is critical, so that any feature definition remains potentially reachable through specialization. Deletion is not an important part of the algorithm. It is included to weed out useless features for the sake of efficiency. The ELF algorithm would work without deletion, though somewhat less efficiently.

Because ELF specializes bits that are associated with the largest errors, those features that need high magnitude weights tend to be identified earliest. This has the desirable effect of reducing error early in the learning process. As features are found that reduce error, the new errors that emerge tend to be related to features that have smaller magnitude weights. The effect is to keep reducing residual error.

There is no random element of the algorithm itself. Given the same stream of point-wise errors, the algorithm will do the same thing every time. The algorithm decides when and where to add new features based on need.

Table 2. Artificial 4^{10} Domain

Target v

Weight	Feature
-93.9527350	ffffedffff
-76.8096050	feeedfedde
74.4897340	ffef79d3fe
74.0239410	fefbefff5b
45.6317710	edfef7efdf
-30.6775870	ffffffffff
14.2437830	bfbfdfe6de
-10.0314590	ff3f3ff7fd
1.9227300	fffffdfffe

ELF $\hat{v}$

Weight	Feature
-93.9527350	ffffedffff
-76.8096050	feeedfedde
74.4897331	ffef79dffe
-74.4897259	ffef79dcfe
74.0239411	fefbeffffb
-74.0239410	fefbefffab
45.6317710	edfef7efdf
-30.6775867	ffffffffff
14.2437743	bfbfdfe6de
-10.0314592	ff3ffff7fd
10.0314589	ff3fcff7fd
1.9227299	fffffdfffe

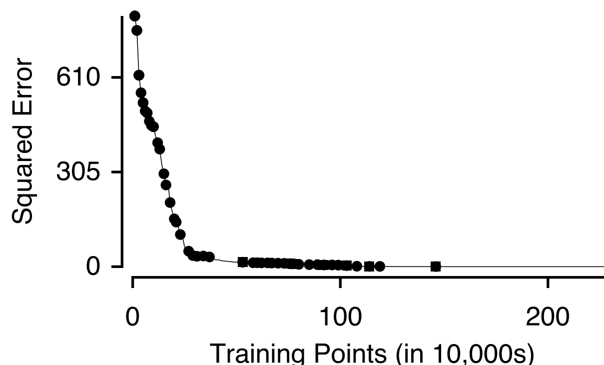

Figure 4. Squared Error for Artificial

3 An Artificial Task

To illustrate ELF, the algorithm was embedded in a program that includes a function v represented in the same form that ELF uses for its approximation. The target v is not available to the algorithm. The program repeatedly samples a point at random from $\mathbf{X}$, evaluates it with each of the target v and the approximated $\hat{v}$ to determine the error, and then delivers the point-wise error to ELF. This loop continues to execute until a moving average of the absolute errors drops below 10^{-6}.

The target v and the final approximation $\hat{v}$ are shown in Table 2, though 31 features all with magnitudes below 10^{-4} have been omitted. A total of 63 features were created, with 20 of them deleted, leaving 43 total features, 12 of which appear in the figure. Each feature is shown as its weight and its pattern coded in hexadecimal, where '#' is coded as bit-value 1 and '0' is coded as bit-value 0. The target function consists of 256 regions over a domain of 1,048,576 elements. Comparing the features of $\hat{v}$ with those of v, one sees that ELF found an approximation in terms of the intrinsic properties in the target. There are three cases of a pair of features that collectively match a single feature in the target. For example, `ffef79dffe` and `ffef79dcfe` in the approximation collectively represent `ffef79d3fe` in the target.

A moving average of the mean squared error is plotted in Figure 4, with the number of training points shown in ten-thousands. The error comes down quickly, with revision of $\mathbf{f}$ continuing longer after the squared error has attained a low value. Each dot along the curve shows when some feature was (copied and) specialized by one bit (changing a '#' to a '0'). Each square shows when some feature was deleted.

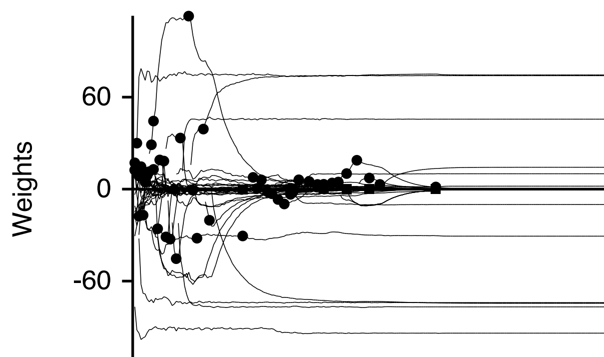

Figure 5. Feature Weights for Artificial

The components of the evaluation function (the feature weights) are shown in Figure 5. The x-axis represents the number of training elements observed, as it does for the previous Figure 4. One can see how the weight of each feature changes over time. Dots along the same curve indicate that the feature has spawned multiple specializations, not that it has become very specialized. After a feature has been copied and specialized, the weights of all the features continue to adjust through subsequent training.

One sees that upon specialization, feature weights 'travel' until a new set of weights that minimize the squared error are attained. Often, many weights change at one time, due to sympathetic movement of overlapping features. When a feature is (copied and) specialized, the new specialized feature has an initial weight of 0. These new features are clearly visible in the graph. By looking at the dots on the curves, one can find the new feature that is born at that time.

Although this figure is easier to interpret in its color version, one can still observe two characterstics. First, the features with larger magnitude weights tend to be found earlier than those with smaller magnitude weights. Second, specialization often occurs repeatedly on the same feature. One can see several consecutive specializations occuring on the same feature, by finding a succession of feature creations. One can speculate that as the feature begins to cover instances that share an intrinsic property, the bit errors are more clearly attributable to certain bits, making those features least adequate under the warp measure described above.

4 Checkers

The ELF algorithm has been embedded in a checkers playing program that uses temporal difference learning (Sutton, 1988), specifically TD(0), to infer point-wise errors during play. The program plays against itself using stochastic move selection. The probability of selecting a particular move is a function of how well its approximated $\hat{v}$ compares to the other choices. It is important to select moves from time to time that may appear suboptimal now, so that there is an opportunity to improve the estimate of the expected value of the point (game state). The stochastic move selection at ply 1 is based on values backed up from searching ply 2.

The base-level representation is propositional, with one Boolean variable for each square-

Figure 6. Checkers Square Names

Table 3. Checkers

Weight	Feature
70.228	a2≠O
69.490	b7≠o
-59.596	f1≠x
59.084	c8≠o,f1≠x
-48.215	a4≠x
45.167	a6≠O
-45.010	c8≠x
-43.109	a8≠x
42.129	g8≠o
41.885	g6≠o
-39.940	e8≠x,f1≠x
39.336	f5≠o
-38.503	b7≠o,c6≠x
37.615	h1≠o
-37.078	b7≠x
-35.986	e4≠x,f3≠x
-34.309	a2≠x,a2≠O
33.534	g2≠o
-31.840	b5≠x,b7≠o
-30.746	a2≠O,b3≠x
-29.495	b7≠o,d5≠x
28.847	f1≠x,h3≠o
-27.820	e6≠x
-26.407	b7≠o,c4≠x
25.451	f1≠x,g4≠o

value combination. The squares are named as shown in Figure 6. Any square can have the value 'e' for empty, 'x' for black pawn (single checker), 'X' for black king, 'o' for red pawn, or 'O' for red king. The 32 discrete variables of five possible values each makes 160 base-level Boolean variables. One could be slightly more compact by noticing that a pawn cannot exist in the last row across the board because it is immediately converted to a king when reaching such a square.

Table 3 shows the 25 features constructed by the ELF algorithm that currently have the largest magnitudes. A total of 189 features have been constructed, with 14 deleted, accounting for the 175 extant features, from which these 25 are taken. Although it is not worthwhile to display all of the features here, they are normally kept in the order in which they were created, making it possible to see how the feature set has evolved. Inspection of these 25 features shows that the program has learned to assess game states in terms of material and in terms of certain combinations. For example, the feature with the largest magnitude weight ($a2 \neq O$) covers all states in which there is not a red king at square a2. The large positive weight indicates that this is a valuable intrinsic property of a board state for the black player (the learner). One might expect to see 32 such features, representing the intrinsic worth of the opponent (red) owning kings at different squares, and owning kings collectively, but far fewer are present in the set of 175. This may well indicate that the

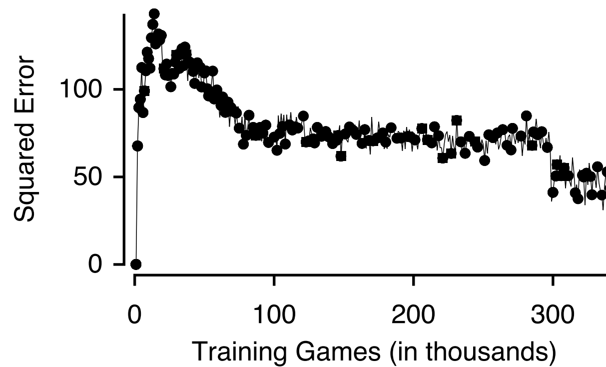

Figure 7. Squared Error for Checkers

program is not exploring (choosing apparently suboptimal moves) enough. A second run is now underway.

The features with more than one term (more than one '0' in the pattern) often appear to be somewhat strange. Each one represents a hypothesis of an instrinsic property, with a particular instrinsic worth, given the other features. Many of these features will probably eventually be superceded, but they have come about, and are observable now, due to the point-wise errors observed in the portions of the state space visited.

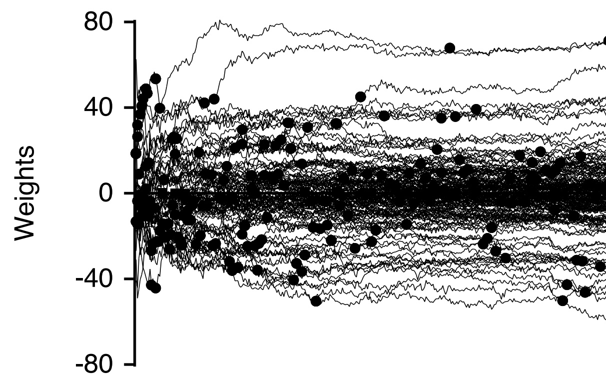

Figure 8. Feature Weights for Checkers

Figure 7 shows a moving average of the squared point-wise error values. The program receives a reward of $100 + \frac{1}{n}$ when it wins, minus that amount when it loses, and 0 when it draws, where n is the number of moves in the game. A game is drawn when 200 consecutive moves without a jump have occured. The program approximates the value of a position poorly early its learning, then improves and appears to level for almost 200,000 games until its features evolve sufficiently to enable further improvement. It remains a conjecture that the improvement will continue, albeit slowly.

Figure 8 shows the trace of the feature weights. There are no apparent cycles, and new features continue to be constructed. No one knows how complex the true value function v for checkers is in this base-level representation (or any adequate representation for that matter). The program beats a random player handily, but its level of play cannot yet be considered good. A version exists that imediately adds 120 useful features to the initial

ELF approximation, but it too reaches a point after which progress comes slowly. While it is tempting to add hand-crafted features, the objective is to study how a function approximator can find useful features on its own, as suggested by Samuel (1963).

5 Related Work

There is much research on the problem of constructive function approximation, notably in the study of artificial neural networks. For example, Ash's (1989) Dynamic Node Creation detects when the error of multi-layer feed-forward network asymptotes, at which point if that error is unacceptably high, the algorithm adds a new hidden unit to the network, and training resumes. For this common kind of network, each hidden unit is a feature that maps its inputs (the base-level variables) to a squashed output between 0 and 1, or some other defined range. The feature itself computes a linear combination of the base-level variables and a vector of adjustable weights specific to the feature. The squashed output defines a differentiable set boundary, and the feature-specific weights combined with the base-level variables effect a rotation of the axes in the base-level space. Such a rotation can be quite useful, and amounts to a method for defining a kind of set cover for the feature that is different from that available to the ELF approximator. Two known problems with artificial neural networks (not specific to Ash's DNC) are that they can become trapped in weight space, such that observing point-wise errors becomes useless, and secondly that multidimensional rotated squashed features are difficult to interpret.

A meiosis network (Hanson, 1990) is a feed-forward network in which the variance of each weight is maintained. For a unit that has one or more weights of high variance, the unit is split into two. The input and output connections are duplicated and the corresponding weights of the two units are moved away from their means. Meiosis networks have been tested on classification tasks.

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. The system detects when the hyperplane of a hidden unit is oscillating, indicating that the unit is being pushed in conflicting directions in feature space. Such a unit is split into two units, and the weights are set so that the units are moved apart from each other along an advantageous axis. Although this approach sometimes works well, Wynne-Jones observes that the units often work back toward each other instead of diverging.

Fahlman and Lebiere's (1990) cascade correlation method constructs a new hidden unit and freezes its defining weights. The original input variables and the newly constructed unit become the input variables for the next layer. Thus, one adds a new feature and a new layer of mapping at the same time. The algorithm alternates between adding a new unit/layer, and adjusting the weights for the output units. The algorithm has produced good results when applied to classification tasks.

Fritske (1993) has developed growing-cell-structures, an improvement to self-organizing feature maps. The algorithm builds an approximation in the form of an interpolator, adding refinement and precision when and where needed.

From the study of genetic algorithms (Goldberg, 1989), one might expect a method using feature recombination (crossover) and mutation to do well. Our earlier work with

a randomized (though not GA) approach did not do nearly as well as the ELF algorithm. Using error to indicate where and when a new feature is needed is a more direct approach. In both illustrations discussed above, one sees little turnover in the features, i.e. few deletions. While it is possible that a more aggressive approach to feature construction may prove useful, one must keep in mind that each '#' changed to a '0' in a pattern cuts its coverage in half. If one is hoping to identify intrinsic properties shared by a large number of domain elements, then one needs to specialize somewhat methodically.

6 Convergence

In the two illustrations of ELF above, ELF clearly converges for the artificial task, which is a stationary function approximation problem. Convergence can be guaranteed for such problems if ELF does not delete features. Deletion is not an essential part of the algorithm, so one could as well consider a version of ELF that does not delete.

As ELF receives point-wise errors and adjusts the weights of its features, it is fitting a linear combination of fixed nonlinear features. Because it is using the LMS rule, the approximation is guaranteed to minimize the error, given the current set of features. We are concerned however with stepping from feature set to feature set in a way that enables an unconditional global minimum to be reached, instead of a global minimum given a feature set. At each conditional global minimum, either it is in fact an unconditional global minimum or it is not, which can be determined by whether the errors are nearly 0. If the conditional global minimum is not unconditionally at a minimum, then the ELF approximator will add a new specialized feature. Given the new feature set, ELF has the opportunity to achieve a new conditional minimum that is lower than was achieved with the previous set of features.

In a version of ELF that adds features, but does not delete features, in the limit, all possible features might be added, at which point any function can be approximated, and the LMS rule will find a global minimum, which in that feature space must also be an unconditional minimum.

For a problem in which there is noise in the base-level variable encoding, or the target value for a particular domain element, how will the ELF approximator behave? To gain at least some insight on this matter, a program was set up in which ELF was trained on points from the 11-bit multiplexor problem. However, the target value of 0.0 or 1.0 had a random value uniformly between ± 0.5 added to it. The program reduced the error to near 0.25, the expected amount. While the algorithm continued (vainly) to add new features and delete old ones, the total number of features remained small.

ELF is also a well-suited function approximator for temporal difference algorithms. Tsitsiklis and Van Roy (1996) proved that TD algorithms converge in the limit with probability 1 when used with function approximators that are linear in the parameters, if the basis functions of the approximator satisfy certain assumptions. Firstly, the basis functions need to be bounded, which is true for the features that ELF constructs. Secondly, the basis functions should be linearly independent. The proof that the complete set of features for a given number of input bits is linearly independent can be done by induction, but is omitted here. This means that any subset of features will be linearly independent as well. There is one additional assumption involving the basis functions, that is always satisfied for problems

with finite state spaces, which must be true given the Boolean base-level encoding required by ELF.

In the case when only feature addition is allowed, ELF can be viewed as constructing a sequence of function approximators that are linear in the parameters. If TD algorithms are used, each such approximator will converge to a set of weights that provides a certain quality of approximation. If the error is too big, ELF will add another feature and resume the process of tuning the weights. Based on results from the cited paper, this process will converge in the limit for tasks that satisfy the Markov chain assumptions.

7 Summary

The ELF algorithm can learn to approximate a real-valued function over a space of boolean-valued variables. The algorithm constructs features that it needs to enable its weight adjustments to reduce error. The construction of new features is driven by associating point-wise errors with the domain variables. The principal operator is to specialize an existing feature that is least able to fit the values for the elements that it covers. A proof was sketched that ELF converges to a global minumum, but no formal proof yet exists. The features found by ELF represent intrinsic properties. The pattern language for a feature's set cover has a logical form that is easy to interpret.

Acknowledgements

Rich Sutton provided several useful suggestions during the early stages of this work. We thank Andy Barto, Gunnar Blix, David Jensen, and Margie Connell for providing helpful comments.

References

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

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.

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

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

Samuel, A. (1963). Some studies in machine learning using the game of Checkers. In Feigenbaum & Feldman (Eds.), *Computers and Thought.* New York: McGraw-Hill.

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

Tsitsiklis, J. N. , & Van Roy, B. (1996). *An analysis of temporal-difference learning with function approximation*, (Technical report LIDS-P-2322), Cambridge, MA: MIT.

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).