ELF: An Evaluation Function Learner That Constructs Its Own Features

Paul E. Utgoff

Technical Report 96-65
October 7, 1996

Department of Computer Science
University of Massachusetts
Amherst, MA 01003

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

Abstract

The ELF algorithm learns an evaluation function that maps an object that is described by a conjunction of discrete variables to a real value. This is accomplished by simultaneously constructing binary features and adjusting a real-valued weight for each of the constructed features. The algorithm determines how many features it needs and what they should be in a deterministic manner.

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 feedback. Many researchers are now addressing the problem of how to search the space of parameterized models automatically. To find a good parameterized model is to find a good representation, and a representation is good to the extent that it facilitates finding quickly an evaluation function of high accuracy and low memory cost.

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 $\mathbf{x} \in \mathbf{X}$. 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 $\hat{v}$ becomes difficult or impossible.

We have noted three degrees of freedom for systems that learn a good $\hat{v}$: vary the base level representation, vary the model class (parameterized combination of features), and vary the parameters of the current parameterized model. One often encounters systems in which the parameters are adjusted automatically, while the representation and parameterized model are held fixed. More recent work addresses how to vary all three, and that is the focus here.

One would like to be able to pick an obvious and easy-to-implement base level representation for the elements of $\mathbf{X}$, and this is commonly done. 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 would be quite difficult to manipulate the base level representation directly. Hereafter, we assume that the base-level representation is held fixed, and that the vector-valued function $\mathbf{f}$ is varied instead.

2 Algorithm ELF

We present an algorithm, which for convenience is called ELF (Evaluation Function Learner, with a twist). It varies the vector-valued function $\mathbf{f}$ by changing its component

functions and the number of them, the model class, and the parameters of the model.

2.1 Representation

The base-level representation consists of an encoded set of discrete variables. To simply the presentation, we shall assume that every discrete variable has the same number of possible values. There is excess capacity in this representation, but it is easy to implement. Consider a few examples. In the game of Checkers, there are 32 squares, any one of which has five possible contents: empty, red pawn, red king, black pawn, black king. Each of the 32 squares can be seen as a discrete variable with five possible values. Each value is encoded as a proposition (bit) that is true if the square has that value and false otherwise. Thus, each of the squares would be represented with five bits, for a total of 160 bits. A second example is the eight-puzzle, in which there are 9 locations, each of which can take on one of nine values, making an 81-bit representation.

An element of $\mathbf{X}$ is thus encoded as a boolean matrix, with one column for each discrete variable, and one row for each possible value of the variable. A matrix entry is 1 if the variable has that value in the domain element, and is 0 otherwise. Every column contains exactly one 1. For Checkers, a board is represented by a 5x32 matrix, and for the eight-puzzle, a board is represented by a 9x9 matrix.

Table 1. Feature Matching

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Feature f_6 | Element $\mathbf{x_4}$ | Element $\mathbf{x_7}$

The vector-valued $\mathbf{f}$ is a vector of feature definitions that initially consists of one component feature that is defined to be the most general feature. The vector of feature definitions subsequently changes due to addition of new feature definitions, deletion of existing feature definitions, and revision of feature definitions. Each feature is represented by a boolean matrix with exactly the same dimensionality as an element of $\mathbf{X}$. A 1 in the feature matrix means that, for matching purposes, a 0 or 1 is permitted in the corresponding cell of the element's matrix, and a 0 in the feature matrix means that only a 0 is permitted in the element's matrix cell. Consider a column of a feature matrix. If all entries are 1, then any domain element will match in that column. Thus, this representation provides a conjunctive normal form description of a set of domain elements, with one conjunctive term for each discrete variable. For example, Table 1 shows a feature matrix for a feature f_6, and the matrices for two different elements $\mathbf{x_4}$ and $\mathbf{x_7}$. The feature matrix f_6 covers the element $\mathbf{x_4}$ but it does not cover the element $\mathbf{x_7}$.

The most general feature consists of a matrix in which every cell has a value of 1. The most general feature matches (covers) every domain element because every value in any element's matrix is permitted according to the feature matrix. A feature covers a domain element if and only if every every bit value in the element's matrix is permitted according to the feature's matrix. One can implement the match predicate efficiently with a bitwise formula. By changing a bit in a feature matrix from 1 to 0, one specializes the feature because the set of domain elements that it covers is reduced, except when the set is already

empty. A feature maps a domain element $\mathbf{x}$ to the value 1 if it covers the element, and the value 0 otherwise.

The feature function $\mathbf{f}(\mathbf{x})$ is a vector of 0s and 1s, and the evaluation function $\hat{\mathbf{v}}$ is defined to be the linear combination $\mathbf{w^T f}(\mathbf{x})$, where $\mathbf{w}$ is a vector of real-valued weights. It is sometimes convenient to think of $\mathbf{f}$ as a list instead of a vector. The feature values are combined linearly with the corresponding weights, so one needs to ensure that each feature value is multiplied by the corresponding weight. The ordering of the component feature definitions in $\mathbf{f}$ is unimportant as long as the correspondence with the weights in $\mathbf{w}$ is maintained. Indeed, in the implementation each weight w_i is kept as a field of the feature definition to which it corresponds. The list of features that defines $\mathbf{f}$ changes during learning, including growing and shrinking, so this is a convenient way to maintain the correspondence between weights and features.

Because the list of features is growing and shrinking during learning, one can see that the parameterized model is changing in the number of components and in the adjustable parameters. No other model classes are considered, but there is no loss of generality in terms of representation. For example, one could have a list of features in which each feature covers exactly one domain element, with each domain element covered by exactly one feature. Then each feature can have a weight that is the value of the instance that it covers, providing a precise form of lookup table. So, the design choice of representing the evaluation function as a linear combination of binary features allows representing any real-valued function over the inputs. Although the function is linear in the weights, the binary features are non-linear over the inputs.

2.2 Operation

The top-level of algorithm ELF is data driven, and consists of three steps. First, upon receiving a training element (a domain element with a target value for $\hat{v}$), one computes the error for that element as the difference between the provided target value and the current value that $\hat{v}$ produces for that element. Second, one updates the weights $\mathbf{w}$ to reduce the error for that element. Finally, the feature function $\mathbf{f}$ is updated as needed. With respect to learning a good $\hat{v}$, it is assumed that training elements are supplied serially. The source of a training element is a surrounding environment, which may provide a perfect target value or a noisy value, or a value that is a current estimate that may be revised later by presenting the same domain element with a new target value. We shall not concern ourselves further with the source, but instead simply adopt the view that it is beneficial to make use of the training element.

The procedure for updating the weights uses the well known Widrow-Hoff rule (Duda & Hart, 1973) for updating the weights in a way that attempts to minimize the mean squared error of the evaluation function. One computes a fixed 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 stepsize parameter α is normalized dynamically. In addition to updating the weights, a separate matrix of bit-errors is maintained for each feature. These bit errors are accumulated by attributing the amount of correction that has been applied to the feature weight to each of the bits in the feature matrix whose

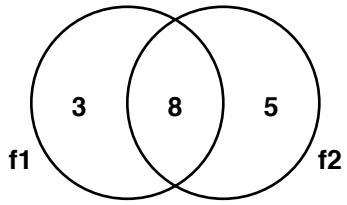

Figure 1. Four Regions from Two Features

corresponding bit was set in the domain element matrix. This information is of central importance to the procedure for updating the features, but plays no role in updating the feature weights.

The procedure for updating the features involves three kinds of operations: specializing an existing feature, adding the most general feature, and deleting a feature. The rest of this section describes these three operations.

Because every feature is binary, it is convenient to think of each feature in terms of the set of instances that it covers, and to depict the features in a Venn diagram. Each feature covers a set of elements, and each feature has an associated weight. As shown in Figure 1, the value to which v maps a domain element is the sum of the weights for those features that cover the element. An element covered by f_1 alone has value 3, an element covered by f_2 alone has value 5, and an element covered by both f_1 and f_2 has value $3 + 5 = 8$. In general, n features will define 2^n regions of the domain that can be mapped in this additive way. For ELF, the most general feature is always included in $\mathbf{f}$, so the number of regions is 2^{n-1}. The most general feature is not included in the Figure.

For a given $\mathbf{f}$, the process of adjusting the weights $\mathbf{w}$ is one of adjusting the additive values of the regions defined by the features. If one were to adjust w_1 to be 2, then all regions that include the instances covered by f_1 would be reduced simultaneously by 1. This kind of interaction is well known, and it is of central importance in finding a good feature function $\mathbf{f}$. Consider what happens when some of the domain elements covered by f_1 alone should take on one value for $\hat{v}$ while still other domain elements covered by f_1 alone should take on some different value. Clearly the single feature f_1 is not sufficient for representing $\hat{v}$ accurately. As training points are observed, and the weight adjusting procedure repeatedly modifies w_1, the weight vascillates because no single value is correct for all points. This of course has an effect on those elements that are covered by both f_1 and f_2, causing weight w_2 to vascillate in sympathy. If we could detect which of these features was less able to find a good weight, we could attempt to improve the feature by changing its coverage, which is what ELF does.

As mentioned above, while the weights $\mathbf{w}$ are being tuned, the error that is attributed to the bits of each feature matrix collect valuable information about what the features are being asked to do. Presumably, a feature is associated with an instrinsic property of the elements that it covers. This property contributes a certain amount to the value of the element. For now, assume that we have accumulated enough bit error information that we

are ready to act. For each feature, ELF computes the difference between the highest and lowest bit errors. A feature that has very different bit errors is trying to have very different values for its one weight, and in a way that can be associated with the bits of the instances that the feature covers. We refer to this difference between highest and lowest bit errors as the *warp* of the feature. The feature is not actually warped, but the bit errors indicate forces that could be allowed to change the feature definition. The feature with the largest warp value is doing the least effective job at identifying an intrinsic property of the instances it covers, and is the feature that will be specialized.

Having chosen the feature to specialize, one then needs to identify the best bit to set to 0 (disallow) in the feature's bit matrix. ELF selects the eligible bit whose bit error is most different from the mean of the bit errors. However, a bit is not eligible if clearing it would produce an empty feature or a feature that duplicates one that already exists. ELF adds a copy of the selected feature to $\mathbf{f}$, sets its associated weight to 0, and clears the selected bit in the newly copied feature. Every feature definition can be reached in principle, even though the only method for revising a feature definition is to specialize it. Finally, for every feature, its bit errors and several other bookkeeping variables are reset. However, the feature weights $\mathbf{w}$ are left untouched.

We know which feature and which bit of its matrix to specialize, but we have not yet said when to take such action. As the weights adjust, bit errors accumulate. When does one know that a bit should be specialized? For ELF, the weights must become relatively stable and the error in $\hat{v}$ must become relatively stable before taking action. 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.

We consider the weights to have stabilized if they are all varying within a fixed range. Of course the weights may continue to change, and potentially by large amounts, but all the activity comes to be recognized as occuring within fixed bounds. With each feature, in addition to storing its weight, the minimum observed value of the weight and the maximum observed value of the weight are maintained. When updating the weights, if a new min or max is established for any weight in $\mathbf{w}$, then the counter nsnmm is set to 0. Otherwise it is incremented. If this counter were to become sufficiently large, one would consider the weights to be varying within a limited range, and hence to be relatively stable. If the counter nsnmm exceeds five times the number of features in $\mathbf{f}$ times the number of matrix elements, then the counter is sufficiently large to infer relatively stable weights.

The method for determining whether the error has stabilized is similar in spirit. It is relatively stable if a new minimum magnitude of the error has not been established recently. The counter nsnme is set to 0 whenever a new minimum error magnitude is established, and incremented by 1 otherwise. When this counter exceeds the same threshold as stated above for the weights, then the error is considered 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

Table 2. Four Feature Target Function For 3^8 Domain

Target v

Weight	Feature
-783.8091600	76575527
699.8133500	77777777
-347.4764900	47677356
-30.2376300	77377777
-18.9269400	67657367

of time. This is accomplished by considering features for deletion only at the same time that one would 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.

2.3 Discussion

Because ELF specializes bits that are associated with the largest errors, those features that have 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 will have smaller magnitude weights. The effect is to keep reducing residual error.

With regard to using the ELF algorithm, it has several useful properties. First, the number of features grows or shrinks as necessary. One does not need to search for a good number of features by making multiple runs. Second, there is no random element of the algorithm itself. Given the same stream of training points, the algorithm will do the same thing every time. Third, the representation for learning an evaluation function $\hat{v}$ is adequate for describing any evaluation function over the discrete inputs of the base-level representation. Finally, the $\hat{v}$ approximation of v is highly accurate for the problems tested so far, converging to the stationary target v in every case. It is not known however whether ELF will always converge for noise-free stationary targets.

3 Illustrations

The ELF algorithm has been embedded in a program that permits a simple form of testing. The program repeatedly generates a domain element at random, evaluates it using the correct evaluation function v, and then provides the training element to the ELF algorithm so that it can update its version of $\hat{v}$. The program continues in this manner until a moving average of the observed errors drops below 10^{-10}. This provides a rudimentary method for seeing how well ELF can learn a stationary noise-free evaluation function.

The target function v is loaded from a file, or generated at random. It is of the same form that ELF uses for the $\hat{v}$ that it learns, i.e. a set of features, each described by a bit matrix and a single associated real-valued weight. One should question whether a target function of this form is in some way making the task simpler for the ELF algorithm. For the moment however, it is a sufficient means for testing whether ELF can find the intrinsic properties that one can ascertain by inspection of the target function.

Table 3. Function Learned by ELF For 3^8 Domain

ELF $\hat{v}$

Weight	Feature
783.8091600	76575557
-783.8091600	76575577
699.8133499	77777777
-347.4764899	77677356
347.4764899	37677356
-30.2376300	77377777
-18.9269397	67657367
-0.0000007	77677377
0.0000003	37677377
0.0000003	57677377
0.0000002	77677357
0.0000002	77677337
-0.0000001	67757767
-0.0000001	77657377
0.0000001	67777777
0.0000001	77757767
0.0000000	37777776
-0.0000000	77777776
-0.0000000	37777777
-0.0000000	77775757
0.0000000	76777577
0.0000000	77777377
0.0000000	77775777
-0.0000000	76777557
-0.0000000	67657767
-0.0000000	76577577
-0.0000000	77757777
0.0000000	76777777

A variety of target functions have been found by ELF, and one is shown here to illustrate some of the characteristics of the ELF algorithm in this supervised setting. The target function consists of four (4) features over a domain of elements where each is described by a conjunction of eight three-valued discrete variables. The size of the domain of elements is $3^8 = 6561$. In addition to the four features, there is also the most general feature, which is functionally equivalent to the customary bias weight because it associates a fundamental weight with all instances. Thus there are $2^4 = 16$ different regions of the target v. The target function is shown in Table 2, with the columns of each feature matrix collapsed into a single octal digit. Each column of matrix entries is encoded in a single octal digit.

The supervised learner using the ELF algorithm found the evaluation function shown in Figure 3. Feature deletion was disabled during the run in order to show the maximum number of features that were created during the run. Comparing the features of $\hat{v}$ with those of v, one sees three that appear identically in each, and several others that collectively match. Specifically, the features 77777777, 77377777, and 67657367 appear in each. However, for the feature 76575527 in the target, one sees the two features 76575577 and 76575557 in the learned function. These two features in the learned function differ by one bit, and have weights that are additive inverses. When both features match an instance, the weights cancel each other. Only when 76575577 matches, and 76575557 does not, is there a non-zero contribution to the linear combination. This occurs precisely when an instance matches 76575527, as in the target. One also sees that the features 77677356 and 37677356 in the learned function collectively match the 47677356 feature of the target function. Thus,

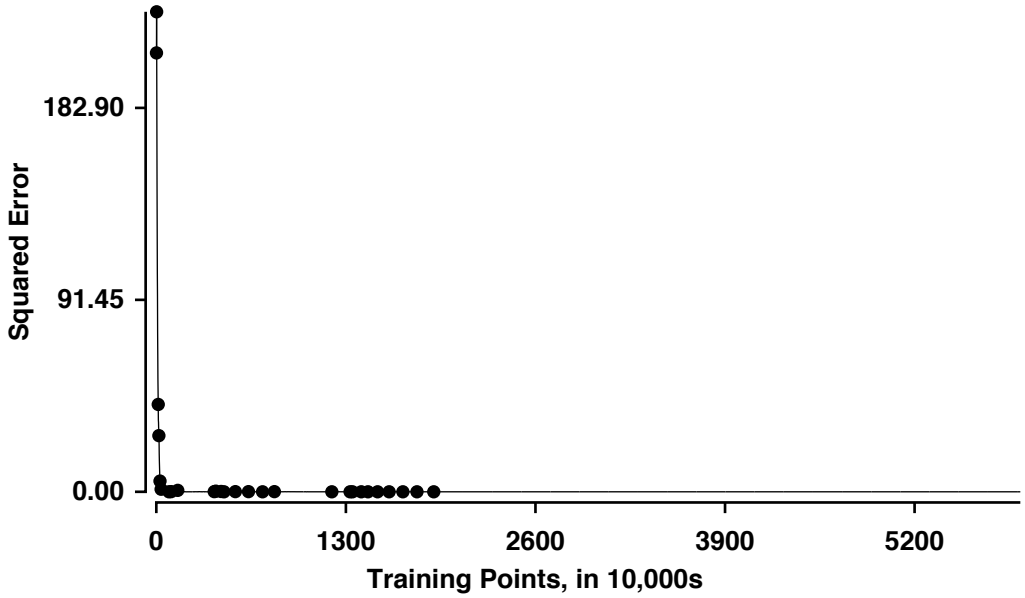

Figure 2. Squared Error of ELF for Five Feature Function Over 3^8 Domain

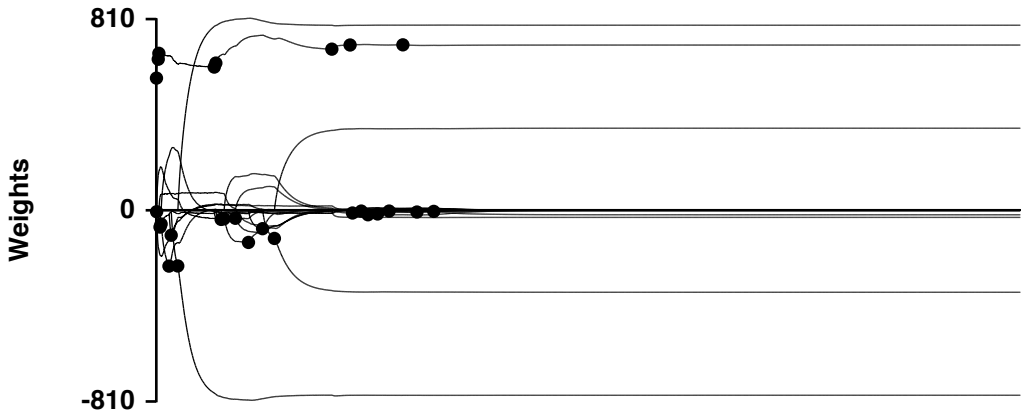

Figure 3. Weights of ELF for Five Feature Function Over 3^8 Domain

although ELF has learned the evaluation function, the feature function that it learned is not optimal with respect to space.

The mean squared error is plotted in Figure 2, with the number of training points shown in ten-thousands. The error comes down quickly, with revision of **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.

The components of the evaluation function (the feature weights) are shown in Figure 3. The x-axis represents the number of training elements observed, as it does for the previous figure. One can see how the weight of each feature changes over time. Each dot depicts a moment when that feature was (copied and) specialized by one bit. 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 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 Related Work

There are several similarities between ELF and the error back propagation algorithm (Rumelhart & McClelland, 1986), particularly when considering the same form of base-level representation. With backprop, the error is computed in the forward direction by evaluating the training point and comparing the target value to the computed value. Then the error is used to update the output unit by adjusting the weights of its inputs. Whereas backprop would then update its hidden units (features), ELF instead updates the bit-errors for each feature. The bit-errors are not part of the parameterized model that represents the evaluation function. Instead, ELF accumulates bit errors, and then periodically adjusts a single feature definition in one step. In contrast, backprop tunes its features at every weight update.

For backprop, one picks a fixed number of hidden units, and initializes their weights randomly. This initialization is needed so that hidden units will tune differently from each other, and come to represent different features. However, by chance some initializations are better than others, leading to variability and uncertainty in the performance of backprop. When the feature definitions cannot be improved further, backprop becomes stuck. In contrast, ELF controls the number of features (hidden units), and no random initialization is employed. Instead, the most inadequate feature is specialized by one bit, and the feature weights are retuned until they restabilize. ELF is somewhat conservative in changing one feature by one bit at a time. However, the selection of a feature and a bit to specialize is guided by errors accumulated since the last time the feature function $\mathbf{f}$ was modified.

A variety of constructive methods have been devised for revising the hidden layer of an artificial neural network. Ash's (1989) Dynamic Node Creation measures when the error of the network asymptotes, at which point if that error is unacceptably high, the algorithm adds a new hidden unit to the network, and training resumes.

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. 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 altering 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.

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.

5 Summary

The ELF algorithm came into being during August 1996. Work is continuing, and much remains to be done in exploring its characteristics and capabilities. For the problems on which ELF has been tried so far, ELF has found the target evaluation function. It has reduced error quickly, and has continued to reduce it asymptotically. The number of features has never grown impractically large, nor to much more than several times the optimal number. Multiple runs have not been needed because ELF determines how many features it needs and what those features are.

Many questions need to be answered. Will ELF converge to the correct v for noise-free stationary value functions? Although this would be desirable in principle, it may not be necessary in practice. How well does ELF tolerate noise? If one encounters noise on the desired value, then the instrinsic properties of the instances may not be obscured. If one encounters noise in the base level representation, the problem may be more difficult. How well does ELF learn for non-stationary problems? How well does ELF work for reinforcement learning methods such as TD learning? How does ELF compare to existing methods such as back-propagation with respect to short term and long term error reduction? Will ELF work as well when the target function is not so clearly related to the features that need to be found? These and other questions are being explored.

There is considerable interest in the problem of learning an evaluation function while finding features that facilitate the task. The ELF algorithm represents a new way to handle the problem when the inputs are discrete. Whether ELF has computational properties that would make it the method of choice for a useful class of problems remains to be seen.

Acknowledgements

I have had many lengthy and helpful discussions with Rich Sutton and Doina Precup. I thank Andy Barto, Doina Precup, 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.

Duda, R. O., & Hart, P. E. (1973). *Pattern classification and scene analysis.* New York: Wiley & Sons.

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

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.

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