Constructive Function Approximation[1]

Paul E. Utgoff utgoff@cs.umass.edu
Department of Computer Science, University of Massachusetts, Amherst, MA 01003

Doina Precup dprecup@cs.umass.edu
Department of Computer Science, University of Massachusetts, Amherst, MA 01003

Abstract: The problem of automatically constructing features for use in a learned evaluation function is visited. Issues of feature overlap, independence, and coverage are addressed. Three algorithms are applied to two tasks, measuring the error in the approximated function as learning proceeds. The issues are discussed in the context of their apparent effects on the function approximation process.

1 Introduction

The use of an evaluation function has become a mainstay in building decision-making components of larger systems. Many approaches have been devised for learning such an evaluation function while the system performs its tasks. As learning progresses, the evaluation function presumably improves at estimating the value of each element that it evaluates. Thus, the learned evaluation function approximates the true function that one would like to have in hand. Of central interest is how to formulate this approximation improvement process in a way that attempts to optimize several important criteria. A critical part of this process is to manage the set of features that are used to express the approximation.

The context in which the evaluation function is used and trained is largely immaterial for the discussion here. Typically, one would expect that the evaluation function learning is embedded within a system that is gaining experience through actual or simulated use. We can safely ignore this outer embedding here. However, one must note that there can be important interactions between function approximation and behavior, when behavior depends on the current evaluation function. For example, in reinforcement learning, the evaluation function, also known as a value function, often represents the discounted reward that one can expect to receive when proceeding from a certain state. However, if one makes each decision in a greedy manner, by making the apparently optimal choice, then certain states (elements of the domain of the evaluation function) are seen more often than others. As the evaluation function changes over time, the probability distribution over the domain will likely change with it.

Various methods for balancing the need to explore suboptimal states and exploit optimal states have been studied at length in the reinforcement learning community. With respect to the function approximation process, the essence of the interaction between the current approximation and the behavior is that the target function to be approximated appears to be non-stationary. Thus, one must ensure that a function approximation process can track a changing target function, which

[1]The correct citation for this article, (C) 1998 Copyright Kluwer Academic Publishers, is: Utgoff, P. E., & Precup, D. (1998). Constructive function approximation (pp. 219-235). In Motoda & Liu (Eds.), *Feature extraction, construction, and selection: A data-mining perspective*. Kluwer.

means that all representable approximations must remain reachable during learning. The discussion focuses on the stationary function approximation problem, taking this proviso into account.

The function approximation problem considered here is the following: given a finite domain $\mathbf{X}$, where each $\mathbf{x} \in \mathbf{X}$ is a vector of Boolean values, and given a stream of point estimates point estimate $(\mathbf{x}_i, v_i)$, find a function $\hat{v} : \mathbf{X} \rightarrow \Re$ that minimizes mean-squared error, learning time, and space consumption. The primary concern is to minimize error, but for a given level of error, one wants to use as few resources as possible. The Boolean domain admits many difficult functions. For example, the game of Checkers has 32 squares of no more than five possible values each, which can be encoded in a straightforward manner using 152 Boolean variables. More generally, a discrete variable with a value set of size n can be recoded as n propositional variables. It is also often practical to convert a real-valued variable to a set of discrete intervals, each of which can be converted to a Boolean variable. CMACs (Albus, 1981) and other discretization techniques have been used frequently to provide Boolean encodings for real-valued variables. There are many techniques that operate in the Real domain, but our interest here is restricted to the Boolean domain, which nevertheless contains many useful target functions that one would like to be able to learn.

2 The Need for Features

It is well known that it is typically very difficult to find an accurate approximation directly in terms of the base-level Boolean variables. The functions to be learned over such a domain are typically highly irregular with respect to the models that are often considered. For small domains, a lookup table with one cell per domain element can work very well. However, for d Boolean variables, one needs a table of 2^d cells, so this approach does not scale well to large numbers of Boolean variables. One can think of a lookup table of this kind as an unbiased function approximator. However, even when one can afford the space for a table, one must visit each cell often enough to attain a good estimated for the value of the function at that point (cell). Using a lookup table in this manner amounts to rote learning, which is devoid of generalization. Without a lookup table, how can one construct an approximation of high accuracy?

A common approach is to introduce an intermediate representation, specified as a set of features. In the discussion here, the term *feature* refers to a computable function of the base-level variables. In order to avoid confusion, the term *input feature*, frequently used in the literature, is replaced by *base level variable*. Given a vector of features $\mathbf{f}$, a domain element $\mathbf{x}$ is mapped to a vector of feature values, and $\hat{v}$ maps a vector of feature values to a real value. The function approximator needs to construct features to include in $\mathbf{f}$, and it needs to delete features from $\mathbf{f}$. Given a particular extant $\mathbf{f}$, the approximator needs to combine the feature values numerically so that $\hat{v}$ is computable. The features of $\mathbf{f}$ recode the base-level variables so that fitting a model to the instances as mapped by $\mathbf{f}$ is easier to do.

Feature construction and feature selection are dual problems. In principle, one could construct an approximation in which all possible features were represented explicitly. Then one could either remove features selectively, or mark each feature as being ‘in’ or ‘out’. This would have the flavor of a feature selection problem. Alternatively, one could start with no features, and then subsequently construct any needed feature or delete any existing feature not currently needed. This would have the flavor of a feature construction problem. However, these are just two views of the same problem: deciding which features to use in the approximation. For simplicity, the feature construction view is adopted here.

3 Feature Spaces

There are many ways in which the computed feature values could be combined to compute the estimated value of the function for a domain element. Many forms of combination can be restated as a linear combination of a vector **W** of real-valued weights and the vector of feature values, and this is the form that is discussed below. Although this form may appear to limit the space of describable evaluation functions, it is quite useful, and should be seen as providing a context in which to find a good set of features for representing the approximation of the target function. The function approximation problem becomes one of finding a good set of features, given that their values will be combined linearly with a weight vector to estimate the value of a domain element.

The domain of each feature is the set of base-level Boolean variables, and the range of each feature is the set $\{0,1\}$, for Boolean features, or the interval $(0,1)$ for sigmoid features. Given that sigmoid function values are close to zero or to one at all points, except near the transition, the range can be considered to be $\{0,1\}$ for all the feature types under consideration here. For each feature, its value is 1 if the domain element is covered by the feature, and its value is 0 otherwise. This kind of feature is itself a nonlinear function over the Boolean domain. When the linear combination is computed, a scalar is multiplied with the feature value, thus scaling it. Hence, each feature can be seen as describing a set of domain elements that share a common intrinsic property, contributing additively to the value of a domain element. The problem of finding good features becomes the problem of finding useful intrinsic properties of the domain elements.

Several issues arise with respect to the set of features in the approximation. First, are the features linearly independent? This has a direct impact on whether the weight updating algorithm can find a weight vector **W** that globally minimizes the mean-squared error in the approximation. Without linear independence, a weight updating algorithm is likely to become trapped at a local minimum that is not a global minimum. Though becoming trapped at a local minimum may still produce a useful evaluation function, one would rather avoid such a situation.

Second, are the features orthogonal? Orthogonal features form a basis for the instance space. Consequently, the weights of such features adjust independently of each other, and learning becomes significantly faster. Given the range of the features described above, the orthogonality condition implies the restriction that features should not overlap. When two features overlap, and point estimates are received (serially) from within the domain of each feature individually, and from within the intersection of the two features, then the weight corrections for the two features interact. While this is often workable, learning is slowed compared to updating weights for two features that do not overlap. This slowing is compounded as the number of overlapping features increases. For d distinct Boolean overlapping features, there are 2^d distinct regions of the domain. The number of distinct regions, and the amount of overlap in each have a large effect on the number of weight adjustments that are needed to minimize the function error, given those features.

Third, how large is the domain of each feature? This size has an effect on how quickly the weight that corresponds to that feature will be adjusted. This is because the weight associated with the feature is adjusted when the feature covers the observed domain element, and left alone otherwise (by virtue of having value 0). For the moment, assume that every domain element of the function is equally likely to appear in the observed point estimates. Then a feature with a larger domain is more likely to have its weight adjusted than a feature with a smaller domain. The feature with the smaller domain will probably take longer to reach its final value, given the influence of the other features. While the uniform distribution over domain elements is unlikely, one would still expect to see a large number of the domain elements, and hence notice this effect.

Finally, what sets of domain elements can a feature be defined to cover? Thinking of the function domain as a Boolean hypercube, one can define a feature as a hyperplane through the cube, with domain elements on the positive side of the plane being covered by the feature, and the rest being not covered. Any feature defined by a hyperplane that is either orthogonal or parallel to all the axes is called an *axis-parallel* feature. Any feature that is defined by a hyperplane, whether or not it is axis-parallel, is called an *oblique* feature. One could devise other methods to specify a particular set of domain elements for a feature definition, but no such additional methods are considered here.

4 Comparison of Three Function Approximation Algorithms

In order to gain some insight into how different algorithms for constructing features affect learning, three such algorithms are compared here on two stationary function approximation problems. The primary concern is how quickly mean-squared error falls as training points are received and used to update the approximation in a serial manner. A secondary concern is the size of the approximation. How many features are there, and how complex is each one? In each case, the number of bits needed to encode the approximation is estimated, providing a common basis for the size comparison.

4.1 DNC

Ash's (1989) Dynamic Node Creation (DNC) adds a new feature in the form of sigmoid function of a linear combination of the base level variables. The sigmoid function serves the purpose of providing a threshold, but it is differentiable, which helps the feature weight adjustment process. A feature covers those domain elements for which the feature has a value near one, and excludes those elements for which the feature has a value near zero. These features are also known as the hidden units of an artificial feed-forward network. For the problem considered here, the network output value is a linear combination of the feature values.

Each feature is defined by an oblique boundary. The features also overlap. DNC uses gradient descent to propagate the error in the approximation to the individual features, and again to the weights of each feature. Adjusting the weights of the linear combination that defines the feature corresponds to changing the definition of that feature, which can also be seen as deleting one feature and adding another. The features that are created are dependent on the output weights. As a result, the error surface for the function being approximated has local minima, in which the function approximation algorithm can become trapped.

When the mean-squared error in the approximation asymptotes at too high a level, the algorithm adds a new feature. In our implementation, the weights within a new feature that correspond to the base-level variables are initialized to 0. Therefore, each new feature value is initially 0.5 on the threshold of the sigmoid feature. The weight for this feature that is used in the linear combination of the approximation is also initialized to 0, which means that the new feature makes no initial contribution to the approximation. This has the nice property that adding a new feature has no immediate effect, and the new feature comes to contribute to the approximation smoothly over time as it is tuned in. Note that the weight initializations to 0 are different from Ash's use of small random values. Random values are not needed here, and the algorithm produces repeatable results for the same stream of observations.

4.2 ELF

The ELF algorithm (Utgoff, 1996) constructs each feature as a Boolean function of the base-level variables. The Boolean function is represented as a pattern, and it covers only those domain elements that match the pattern. Each component of a pattern has either the value '#' or the value '0'. A '#' matches either of the possible values of the corresponding base-level variable, while a '0' in the pattern matches only a '0' (False) value. 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.

This pattern language is somewhat unusual because it can specify that a base-level Boolean variable must have value '0', but not that it must have value '1'. Nevertheless, the pattern language is sufficient to describe any evaluation function over the domain. Consider a trivial domain of one Boolean variable. The pair of features '#' and '0' is sufficient because the feature '0' covers just one element, and the feature '#' covers both elements. One can assign a weight to the '#' feature that is correct for the '1' element, and one can assign a weight to the '0' feature so that it is correct for the '0' instance, given that the '#' feature also covers it, and has its own weight. This argument extends to any number of Boolean variables. There is no need for a '1' in this pattern language. The boundaries of an ELF feature are axis-parallel.

When a training point is presented to the ELF algorithm, it takes two steps. First it updates the adjustable weights of the approximation, and then it updates the features in $\mathbf{f}$ as necessary. The weights are updated using the 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 fixed at 0.1.

Initially, the function approximation consists of the one most general feature, with a weight of 0. This initial approximation returns 0.0 for every element. As point-estimates are observed, 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.

While ELF is attempting to adjust the feature weights, initially just the one weight, it accumulates the error that is associated with each bit (base-level Boolean variable) being on (true) 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 '#'s in the feature pattern whose corresponding base-level variable was true. 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.

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

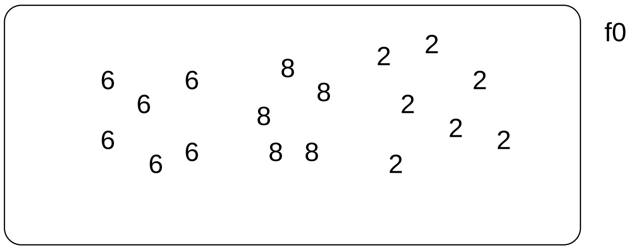

Figure 1. Target Values

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.

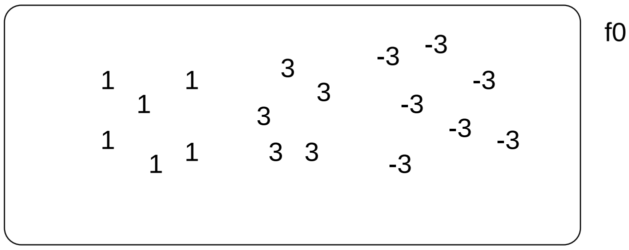

Figure 2. Error with One Feature

Assume that a new feature f_1 has been constructed that covers the 6s and the 8s. Through subsequent point-wise error adjustments, suppose that the most general feature f_0 takes on a weight of 2, and the new feature f_1 takes on a weight of 5. Now the new X-ray paints the error picture shown in Figure 3. These errors are smaller than before.

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 6, and f_2 takes on a weight of 2. Now a new X-ray would be transparent, as shown in Figure 4 because the error is uniformly 0.

To accomplish assigning one value to some instances and not others, one constructs a new feature that covers just those instances. This is done by identifying 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 contributes 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.

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

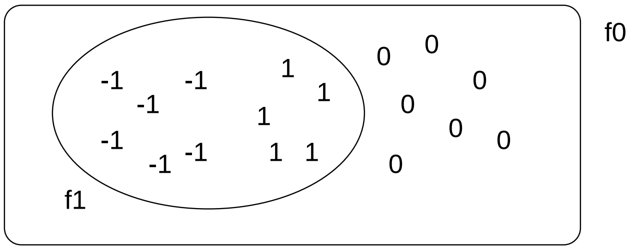

Figure 3. Error with Two Features

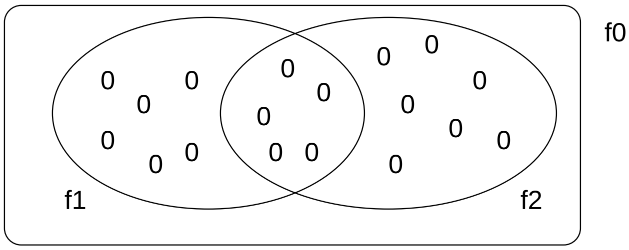

Figure 4. Error with Three Features

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 **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, 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 **w** are left untouched. In addition, it would be quite useless to allow two or more features with identical patterns, so ELF prevents this by not considering any specialization that would produce a duplicate pattern.

The algorithm 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 slightly less efficiently. One would need to omit this deletion mechanism to provide a guarantee that the algorithm will converge to 0.0 error in the limit.

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

Table 1. Artificial 4^{10} Task

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`

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 observed point-wise errors, the algorithm will produce the same result every time, which is highly desirable in terms of understandability and reproducibility. The algorithm decides when and where to add new features, based on need.

To illustrate ELF, the algorithm was embedded in a program that also includes a target function v represented separately from the approximated function $\hat{v}$. The target is in the same form that ELF uses for its approximation. The program repeatedly samples a point at random from $\mathbf{X}$, evaluates it with the target v and then delivers the point estimate to ELF. This loop continues to execute until a decaying average of the absolute errors drops below 10^{-6}.

The target v and the final approximation $\hat{v}$ are shown in Table 1, 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 decaying average of the mean squared error is plotted in Figure 5, with the number of training points shown in tens of thousands. The error diminishes quickly, with revision of $\mathbf{f}$ continuing long 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.

The components of the evaluation function (the feature weights) are shown in Figure 6. The x-axis represents the number of training elements observed (in tens of thsousands), as it does for the previous Figure 5. One can see how the weight of each feature changes over time. Dots along the

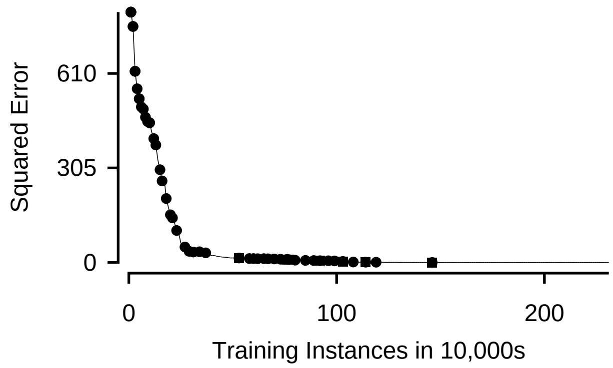

Figure 5. Squared Error for Artificial Task

same curve indicate that the feature has spawned multiple specializations, and not that the feature itself has become very specialized. After a feature has been copied and specialized, the weights of all the features continue to adjust through subsequent training.

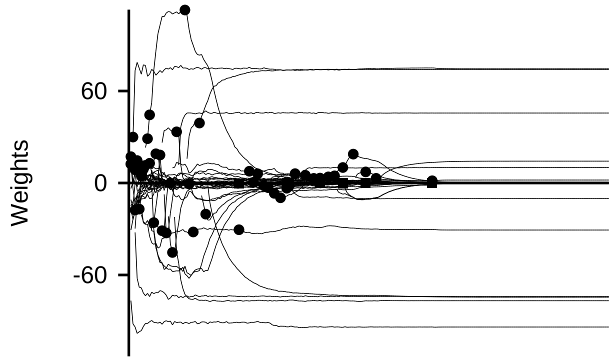

Figure 6. Feature Weights for Artificial Task

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.

One can observe two characterstics in the figure. 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 stress measure described above.

4.3 RT

The RT algorithm is somewhat similar to the ELF algorithm. One major difference is that RT instead constructs features that do not overlap. This is done by using a slightly different pattern language. A '#' matches either True or False in the corresponding Boolean variable. A '0' matches only False, and a '1' matches only True. Presence of a '1' in the pattern language differs from ELF. The set of features is initialized with the most general feature. When it becomes time to specialize a feature, two new features are constructed, and the parent feature is deleted. In one of the new features, the pattern is that of the parent, except that the identified '#' has been changed to a '0'. Similarly, in the other new feature, the '#' has been changed to a '1'. The weights for the two new features are initialized to that of the parent. These two features that have replaced the parent are now free to have their weights adjusted independently. Indeed, every element of the domain is always covered by exactly one feature. This is good for orthogonality, but it loses the idea of identifying intrinsic properties.

This strategy for modifying the set of features used in the approximation can be seen as a method for building a variable-resolution lookup table. Variable resolution methods for discretizing continuous variables have been used successfully in reinforcement learning tasks (Chapman & Kaelbling, 1991; Moore & Atkeson, 1995). This strategy can also be seen as forming a regression tree whose leaf values adjust continually to observed errors at that leaf (Breiman, Friedman, Olshen & Stone, 1984).

Because the features do not overlap, the RT algorithm cannot produce duplicate features. Because there are no feature interactions, the bit-errors for each feature are not cleared when a feature is split into two, except that the bit-errors for the two new features are initialized to 0.0.

4.4 TicTacToe

The TicTacToe function is defined over the 4,520 states that are reachable during legal play. Every position is expressed from the point of view of the player that is about to move. A position in which a win has been achieved or is attainable has value 10.0, a draw has value 0.0, and a position that is lost has value -10.0. The fact that the function comes from a sequential problem solving task is irrelevant here because it is treated simply as a stationary function approximation problem. This is done by uniformly sampling the 4,520 points (with known target values) repeatedly in an offline manner.

The concern is how various aspects of the feature construction process affect error reduction in the approximation. A regression tree based on minimizing an information criterion contains 1,467 leaves, which suggests an upper bound on the number of features needed for the approximation. As represented here, there are 27 base level variables for the TicTacToe function. This is because each of the nine squares can have one of three possible values. More compact codings are possible, but this one is straightforward.

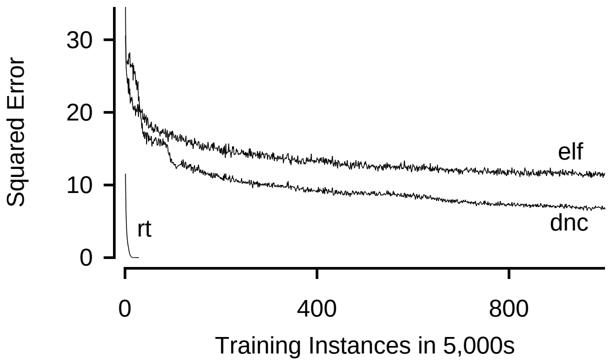

Figure 7. TicTacToe Error Graph

The number of features constructed by each algorithm, and the number of bits necessary to encode each feature are quite different. DNC constructed a function with 37 features. The representation for each feature contains 27 weights and there is also one more weight for the final linear combination. Assuming that each weight can be represented as a 32-bit floating point, the approximated function can be encoded in $(27 \times 32 + 32) \times 37 = 33,152$ bits.

ELF constructed 282 features, and deleted 26, for a net total of 256 features. Each feature can be encoded using one bit for each base level variable, and a real-valued weight. Using the same convention as above, the approximated function can be encoded in $(27 \times 1 + 32) \times 256 = 15,104$ bits.

The function constructed by RT was much larger, consisting of 5,245 features. Features are similar in size to those of ELF, except that a pattern for RT requires twice as many bits as a pattern for ELF. The encoding for the RT approximation takes $(27 \times 2 + 32) \times 5,245 = 451,070$ bits. Note also that there are 5,245 features over a domain in which only 4,520 different states (points) can occur. Clearly RT is overspecializing, which means the criterion for when to split a feature is too aggressive.

4.5 Eight Puzzle

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

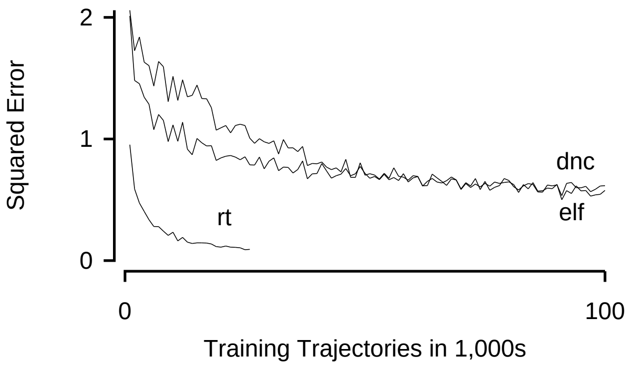

Figure 8. 8-puzzle Error Graph

The number of base-level variables for this function is 81, because each of the nine locations can have one of nine possible values. More compact encodings are possible, but this one is straightforward. DNC constructs a function of 15 features, which needs $(81 \times 32 + 32) \times 15 = 39,360$ bits for the encoding. ELF constructs 222 features and deletes 25, for a net total of 197 features. This approximation can be encoded in $(81 \times 1 + 32) \times 197 = 22,261$ bits. RT constructs a function of 14,065 features, and the encoding requires $(81 \times 2 + 32) \times 14,065 = 2,728,610$ bits.

4.6 Discussion

The three algorithms were each run on both tasks. Within each problem, the sequence of sampled points was identical for each algorithm. For the TicTacToe function, the reachable points in the domain were sampled randomly according to a uniform probability distribution. For the 8-puzzle function, the points were sampled from optimal trajectories, with the correct target value being used for each point. In this formulation, the 8-puzzle is a stationary function approximation task, but with a non-uniform probability distribution over the domain. Only the starting point of each trajectory is sampled randomly and uniformly from the domain. These differences in the sampling distributions (for TicTacToe versus 8-puzzle) occur due to the way these applications were implemented previously, not because of a deliberate design choice made here.

For TicTacToe, Figure 7 shows that DNC and ELF become slow at reducing the error. Both algorithms produce features that overlap. DNC has features that are not linearly independent, so an additional cause of the slowed rate of error reduction may be due to reaching a local minimum with the current feature set. ELF has features whose coverage is individually decreasing as points are seen. Although one can expect ELF to converge to 0.0 error in the limit, the feature overlaps and shrinking feature coverage work against that, causing the rate of error reduction to dwindle. DNC suffers the same fate, due to feature overlap and linear dependence. DNC does a better job of error reduction, suggesting that oblique features are more useful than axis-parallel features for this function. The same figure shows that the RT algorithm reduces error very quickly. It does however create a large number of features, which slows execution. When each of the runs was halted, DNC had 37 features, ELF had 256 features, and RT had 5,245 features.

For the 8-puzzle, Figure 8 shows that all three algorithms reduce the approximation error quite quickly. RT reduces error most quickly, but creates 14,065 features. DNC and ELF come much closer to RT as learning progresses, but still trail. However, each of DNC and ELF use many fewer features than RT in the process. One can surmise that DNC and ELF have each identified intrinsic properties. When halted, DNC had 15 features, and ELF had 197 features. Given the similar accuracies of DNC and ELF, it appears that oblique features have little apparent advantage over axis-parallel features for this particular function. As was observed for the TicTacToe function, the rate at which DNC and ELF reduce error becomes very slow.

One real-world task that we have studied is the instruction scheduling task. The goal is to learn a greedy evaluation function for picking the instruction that should be scheduled next for a basic block during the code-generation phase of code compilation. The task is known to be hard, and it is becoming increasingly important in the context of highly pipelined architectures. It was possible to learn to produce instruction schedules with running times within 10% of optimal on a DEC Alpha architecture. The function that needs to be learned in this case is much simpler than either of the tasks chosen as illustrations above.

5 Related Work

A variety of constructive methods have been devised for classification problems. Several algorithms have been designed for constructing networks of thresholded logic units, by adding boundaries that correct for misclassified examples (Parekh, Yang & Honavar, 1997). It should be possible to extend these techniques to function approximation problems.

A meiosis network (Hanson, 1990) is a feed-forward network in which the variance of each weight is maintained. For a hidden unit (feature) that has one or more weights of high variance, the unit is split into two. The input weights that define the feature, and the output weight for the linear combination are altered so that the two units are moved away from their means in opposite directions.

Wynne-Jones (1992) presents an approach called *node splitting* that detects and attempts to repair an inadequate hidden layer (set of features) of a feed-forward artificial neural network. The system detects when the hyperplane of a hidden unit (weights of an oblique feature) 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 (feature) 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.

Fritzke (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.

The RT algorithm is highly similar to Chapman and Kaelbling's (1991) G algorithm. The G algorithm splits the domain on Boolean variables that are deemed relevant to the function value by a statistical test. Chapman and Kaelbling note that the G algorithm will fail when bits are irrelevant in isolation, but relevant collectively. The RT algorithm will continue to split the space whenever the error for a feature is too high. Chapman and Kaelbling suggest that perhaps good features should be orthogonal, and therefore individually relevant.

5.1 Conclusions

There are several issues to consider when building a function approximator. The accuracy of the approximation is typically of paramount importance, though it is worth noting that when using an approximation to implement a decision-making component, one can tolerate error in the approximation when it does not lead to errors in decision making.

With regard to oblique or axis-parallel features, one can see here, and in many other examples, that oblique features can be very useful in building the approximation. Given that the set of axis-parallel features is properly contained in the set of oblique features, it would seem obvious to use oblique features. However, current methods for finding such features do not guarantee that those features are linearly dependent, which is a significant disadvantage.

Whether the features overlap has an important effect on the rate at which error is reduced. The number of distinct regions over the domain grows exponentially with the number of features. One can conclude that feature overlap is detrimental. On the other hand, if one is attempting to identify intrinsic properties, then overlap is necessary. When overlap is precluded, the features become orthogonal, and can be adjusted independently. One can reduce error more quickly by avoiding overlap, but at the expense of not finding intrinsic properties, and at the expense of requiring more total features. It may well be that one should first find useful instrinsic properties (as with DNC or ELF), and only then proceed to partition the resulting function (as with RT).

In order for these algorithms to be useful for reinforcement learning techniques, they should be able to track concept drift. DNC and ELF theoretically have this property. Further investigation is needed in order to establish how well they behave in practice. RT commits to a partition of the domain, so it is less suitable for non-stationary tasks. It would track the drift, but with a potentially rigid and therefore poor partition of the domain.

Much more exploration of constructive function approximation methods is needed. It would be useful to design a constructive method that produces oblique features that are linearly independent. It would be useful if the amount of feature overlap could be controlled dynamically, so that slowed learning and creating a large number of features can be balanced automatically. None of the algorithms discussed here or elsewhere truly solves the feature construction problem. A better understanding of the tradeoffs is needed, as well as better control of these tradeoffs in the constructive function approximation algorithms that are to be used in practice.

References

Albus, J. S. (1981). *Brain, behaviour and robotics*. Byte books.

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

Breiman, L., Friedman, J. H., Olshen, R. A., & Stone, C. J. (1984). *Classification and regression trees*. Belmont, CA: Wadsworth International Group.

Chapman, D., & Kaelbling, L. P. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparison. *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence* (pp. 726-731). Sidney, Australia: Morgan Kaufmann.

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

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

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

Moore, A. W., & Atkeson, C. G. (1995). The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning, 21*, 199-233.

Parekh, R., Yang, J., & Honavar, V. (1997). *Constructive neural network learning algorithms for multi-category real-valued pattern classification*, Iowa State University, Computer Science.

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

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

Wynne-Jones, M. (1991). Node splitting: A constructive algorithm for feed-forward neural networks. *Advances in Neural Information Processing Systems* (pp. 1072-1079).

Paul Utgoff is Associate Professor in the Department of Computer Science at the University of Massachusetts at Amherst, where he has taught since 1985. He is a former Action Editor of Machine Learning, and in 1993 served as Conference Chair for the Tenth International Conference on Machine Learning. His principal research interest is in developing methods for automatic construction of representations that facilitate machine learning.

Doina Precup is a third year PhD student in the Department of Computer Science at the University of Massachusetts at Amherst. In 1994, she earned her BSc from the Technical University Cluj-Napoca in Romania, and in the following year she received a Fulbright scholarship. Her current PhD research, under the direction of Rich Sutton, focuses on using spatial and temporal abstraction to enhance reinforcement learning algorithms.