Feature Construction for Game Playing[1]

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

Abstract: To build an evaluation function for game-playing, one needs to construct informative features that enable accurate relative assessment of a game state. This chapter describes the feature construction problem, and suggests directions for dealing with shortcomings in the present state of the art.

1 Introduction

An essential ingredient of a good game-playing program is the ability to approximate well the relative utility of the game states that it may encounter in actual play or look-ahead search. To approximate well, the program requires features that measure strengths and weaknesses of a position, and a method of combining these feature values into a single number that indicates the overall rating of the state relative to others. Features are needed whenever it too difficult to define a good evaluation over the input variables, as is typical. Features are also needed whenever search cannot reach terminal positions from which to back up true payoffs. The state variables, also know as input variables, form the base-level representation of state. A feature is a function of state variables or other features.

When in a particular state, and confronted with a choice of successor states, one evaluates each successor and selects one with the best value, with the possible exception of electing to explore some apparently suboptimal state. In addition, one may look further ahead than the immediate successors, presumably producing a better estimate of the value of the node than one would obtain without looking ahead. For simplicity, we assume that one has a model of the domain, so that modeling the successor states is possible. This is not strictly necessary, as one can represent a successor state implicitly as the current state conjoined with the name of the action that would be used to produce the successor, as in Q learning (Watkins & Dayan, 1992).

2 Evaluation Functions

There are four major aspects of designing a good evaluation function. These four processes interact, and do not necessarily occur in any fixed order. One needs to:

1. create a set of features that can measure the presence or absence of one or more intrinsic properties of a state,

2. choose a functional form (model class) that maps the various feature values to an overall scalar value,

3. provide game payoffs, for the purpose of inferring target values for the states encountered, and

[1]The correct citation for this article, (C) 2001 Copyright Nova Science Publishers, is: Utgoff, P. E. (2001). Feature construction for game playing (pp. 131-152). In Fuerenkranz & Kubat (Eds.), *Machines that learn to play games*. Nova Science Publishers.

4. select a method for adjusting parameters that are part of the functional form.

Our focus is on the first of these four design aspects, the problem of creating useful features. To a large degree, it is possible to make simplifying assumptions or rely on existing methods for the other three. Due to the interactions, one needs to consider them all, but our interest is primarily in feature construction. For example, for the functional form, one can use a linear combination of a particular set of features. For the training signal, one can use temporal difference learning (Sutton, 1988) to infer expected utility of a state. Finally, to adjust the parameters of the functional form, one can descend the gradient of an error function with respect to the adjustable parameters (Press, Flannery, Teukolsky & Vetterling, 1988; Freeman & Skapura, 1991). Other choices for these three items may be preferable in some settings, but those can be determined in the context of a particular scheme for feature construction.

For the purpose of selecting a best move, accuracy is important only to the extent that it affects move selection. It may be acceptable for the absolute assessment of each state to be flawed if the relative comparison of such values nevertheless gives the best value to the best move (Berliner, 1979; Utgoff & Clouse, 1991). The problem of finding a useful set of features may be somewhat simplified by taking relative assessment into account, but the fundamental problem remains.

3 Feature Overlap

A critical design choice is whether to construct features that overlap or that are disjoint. Suppose that Boolean feature f_1 is true for some set of its inputs, and similarly that Boolean feature f_2 is true for some set of its inputs. If the sets covered by these two features intersect, then these features overlap, and otherwise they are disjoint. Suppose that f_1 corresponds to *color(x)=red* and that f_2 corresponds to *shape(x)=square*. In this case it is natural to assume that the two features take on values independently of one another. If the presence of f_1 contributes w_1 to the evaluation, and the presence of f_2 contributes w_2 to the evaluation, then in this overlapping model, joint presence of f_1 and f_2 contributes $w_1 + w_2$ to the evaluation. In the alternative model of disjoint features, each of the f_1xf_2 combinations would be a separate feature with its own individual contribution. For example, say $g_3 = f_1 \wedge f_2$. With disjoint features, only one can be true at any time, and its contribution to the evaluation is the sole contribution, and hence is the evaluation. When g_3 is true, the evaluation is w_3. A common representation for an evaluation function based on disjoint features is a cellular lookup table or a tree-structured lookup table (Breiman, Friedman, Olshen & Stone, 1984).

Our interest here is in overlapping features. One can compensate for dependencies by including higher order terms as needed. For example, if $f_1 \wedge f_2$ should evaluate to 7 instead of 8, then one can include another feature $f_3 = f_1 \wedge f_2$ with weight -1 We would like to detect the presence or absence of each intrinsic property of a state by evaluating a Boolean feature. Target functions (true relative utility) for game playing are typically highly irregular over the state variables. One can only hope to turn the combinatorics of overlapping features to one's advantage. The essential aspect of a good feature is that it applies to a set of states that share some intrinsic property. It may also be the case that a feature is defined with some amount of error in its coverage. The ultimate test of a feature's worthiness is whether the game-playing program can achieve higher expected payoff with the feature than without (Berliner, 1984).

Before putting aside disjoint features, it is worth noting that work on Othello has produced a practical hybrid approach. For Othello, it has become common to use tables for subsets of the

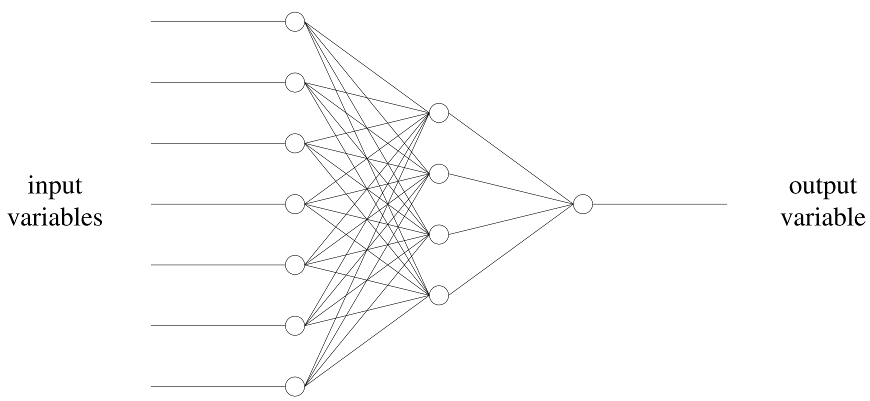

Figure 1. Single Layer of Features

input space, and to combine the individual values linearly. For example, Rosenbloom's (1982) edge-tables represent each of the possible $3^8 = 6,561$ edge configurations individually. Each edge configuration indexes a table in which its associated weight is stored. This is equivalent to treating each edge as $6,561$ boolean features, exactly one of which can be true at any time. The table stores the associated weight to contribute to the evaluation of the position. By using the same table for each of the four edges, one has in effect $4 \cdot 6,561 = 26,244$ boolean variables. These boolean edge features are included in the set of features along with others that are not table-based. BILL (Lee & Mahajan, 1988) extends the edge table to include the two neighboring X-squares. LOGISTELLO (Buro, 1998) extends this idea to other informative subsets of the instance space.

Nonoverlapping features can be very effective in discriminating useful subsets of instances. We suspect that in the longer run, it will be more useful to be able to identify overlapping features that afford the same discriminations. This might save space, but more importantly it would provide some insight on *why* a particular instance has the value it does.

4 Constructing Overlapping Features

Devising and understanding methods for automatic feature construction is a longstanding research quest. Constructing good features is the major developmental bottleneck in building a system that will make high quality decisions. We must continue to study how to enhance our ability to automate this process. This section briefly reviews the state of the art. At present, there are four broad categories of methods for constructing features, which are discussed below. First is parameter tuning, second is higher-order term expansion, third are quasi-random methods, and fourth is knowledge-derivation.

4.1 Parameter Tuning

The most popular method for constructing features automatically is to use a layered feed-forward artificial neural network with at least one layer of hidden units, as suggested in Figure 1. Each unit is an element that computes a simple nonlinear function, e.g. sigmoid, of its inputs. By treating the output of a unit as the input to one or more other units, one obtains a network of computing elements that can in principle compute an arbitrarily complicated function of its inputs. The power of this approach is that the hidden units are defined parametrically, and that the parameter values can be learned by descending the error surface with respect to these adjustable parameters (Werbos, 1977; Rumelhart & McClelland, 1986). Error in the evaluation function is distributed to the hidden units, which are then each updated to reduce hidden unit (feature) error.

While this kind of approach is quite useful for many problems, it suffers several drawbacks. First, such an algorithm can become trapped at a local minimum when descending the error gradient, though for some applications a local minimum may be useful nevertheless. Second, such an algorithm can be very slow, sometimes requiring weeks to months, or longer, to reach its best performance level. Speed would be less of an issue if one could be sure the algorithm would not become trapped at a local minimum. That is the main problem, that the algorithm can fail to find features that it would need to achieve sufficient accuracy. To become trapped at a local minimum is to have a feature definition settle at a local decision boundary that is not as useful as another boundary. The algorithm cannot revise the feature definition because any such attempt would introduce error (temporarily), which the algorithm is designed to minimize. To overcome this, one must repeatedly re-engineer the input variables carefully until the program can learn adequately.

Parameter adjustment of hidden unit input weights is the fundamental aspect of this approach to feature construction. An associated task is to determine the network architecture. How many hidden units shall there be, and how shall the units be connected? Methods have been devised for modifying the network architecture during learning. This makes adding (or deleting) hidden units an explicit part of the procedure, instead of having the system builder implement a static architecture. Despite the appeal of automating construction of the network topology, the problem of becoming ensnared at a local minimum during gradient descent is not eliminated.

Ash's (1989) dynamic-node-creation algorithm DNC inserts a new hidden unit whenever network error asymptotes at an unacceptably high level. Hanson's (1990) meiosis network tracks variance for each input weight, and splits a unit that has the highest variance input weight into two, moving each away from the other at that moment. Similarly, the node-splitting method of Wynne-Jones (1991) detects when the weights of a hidden unit are oscillating, and splits the most oscillatory unit into two. Fahlman and Lebiere's (1990) cascade correlation method learns the weights of just the single output unit. When it stabilizes, if the error is too high, its weights are frozen, and a new output unit is created, with inputs from all of the existing frozen units. Frean's (1990) UPSTART algorithm finds two-class convex boundaries by repeatedly adding new linear threshold units (ltu) in such a way as to correct classification errors of frozen ltus.

In reinforcement learning, it is common to set up a problem so that the program (agent) learns from a single reward or punishment. Suddarth (1991) and Caruana (1997) have each shown that a richer training signal greatly facilitates the learning task. For example, in a game if one provides just the game value for a given state, one has much less information than if one also provides the locations of the legal moves. Knowledge of the legal moves is typically implicit in the problem, but by making it an explicit part of the training signal, the agent can and must learn much more. This is very powerful when, for example, the assortment of legal moves bears on the game value. This indicates more generally that learning features for one task may aid learning another, which is discussed below.

4.2 Higher Order Expansion

One can construct a feature as a conjunction of a subset of the Boolean input variables. The empty conjunction is always true, and therefore covers (is true for) all input examples. Each conjunction of just one input variable is true if and only if the input variable is true in the example. One can create a feature for each distinct pair of input variables. Generally, one can produce terms of increasingly higher order, each one defining a feature. If one includes all possible conjunctive features, then one can represent any evaluation function. Although this kind of completeness seems appealing, an accurate function may require exponentially many features, making it intractable in general. One can use a connective other than AND. For example, using EXCLUSIVE-OR produces a Walsh expansion (Duda & Hart, 1973).

One can expand selectively, by constructing just those features (terms) that one predicts will add accuracy to the evaluation function. For example, consider the ELF algorithm (Utgoff & Precup, 1998). The input variables must be Boolean, which means that discrete many-valued variables or continuous variables must be mapped to Boolean beforehand. The approximator starts with the most general feature (true for all inputs), and tries to learn a weight for it, which amounts to approximating the target evaluation function by a constant. The algorithm responds to a stream of `(example,value)` pairs by adjusting its weights to reduce error for each pair encountered. During this series of corrections, it accumulates for each individual feature an association of errors with its input values. When the weights stabilize, i.e. reach a locally optimal neighborhood (Utgoff, 1989), ELF constructs a new feature that is a specialization of the one with the strongest error/input association. The old feature and the new specialized feature are able to discriminate instances that differ in the input variable of interest, enabling refinement of the evaluation function. The ELF approximator can work well for some problems, but it tends to bog down as training proceeds. There are two problems. The first is that ELF removes the largest errors first. As smaller error residuals occur, it becomes difficult to differentiate their causes. Second, as new features are constructed, the dimensionality of the problem rises, requiring longer training periods for the weights to stabilize.

A second example of an algorithm the expands selectively is the VU (value unification) algorithm (Utgoff & Stracuzzi, 1999). Through a series of best-first searches, the algorithm repeatedly identifies a feature that covers the largest subset of the training examples that does not include any correctly evaluated example. The conjunctive term that defines that subset is adopted as a new feature. This is useful because a weight can be associated with the new feature that will eliminate the error for at least one example. The weight is computed directly in a manner that eliminates apparent error for a large subset of the examples covered by this new feature. The algorithm halts when all examples evaluate correctly. This is an unusual algorithm, combining mechanics of covering algorithms with the manipulation of additive values needed to produce an evaluation. It reduces error in a concentrated manner, rather than being guided by a global error measure. It can find large feature sets quite quickly. Nevertheless, it is brittle and it relies on searching a space of conjunctively defined features.

An important lesson from the work on VU is that one should not be content with a sufficient number of layers of conjunctively defined features. Although it is possible to prove representational adequacy, this does not mean that one has achieved the best possible compression. For example, a VU variant compressed the 5,477 possible positions of tic-tac-toe into a single layer of 2,779 conjunctive features, which is only about 2:1. As discussed below, one can do much better.

Series expansions are generally unwieldy, producing an explosion of features, many of which may be useless. One can attempt to produce the terms of an expansion more selectively, but such approaches eventually bog down.

4.3 Quasi-Random Methods

From the study of genetic algorithms (Goldberg, 1989), one might expect a method using feature recombination (crossover) and mutation to do well. Our earlier attempts to apply this kind of approach (GA and non-GA variants) 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. If one is hoping to identify intrinsic properties shared by a large number of domain elements, then it appears that one needs to specialize more methodically.

4.4 Knowledge Derivation

If one adopts the view that all the knowledge important to state evaluation is entailed in the task definition, then one can start with a formal statement of the task domain, and attempt to decompose it into useful measurable elements (Fawcett, 1993; Callan, 1993). Fawcett's (1993) ZENITH system uses a declarative theory that defines the problem-solving state space, the move operators, and the goal that the learner endeavors to achieve. The system then applies its own transformation operators that synthesize features from this declarative knowledge. ZENITH maps a logical term or clause to a numeric value by computing the number of bindings for which the logical expression can match the state. ZENITH reconstructed most of the well-accepted features for Othello, and it also constructed a useful original feature humanly dubbed *frontier-directions*.

Levinson and Snyder's (1991) MORPH system learned an evaluation function for the game of chess. The set of features is represented as a set of patterns, each with an associated weight. Each pattern is an abstraction of a portion of a state, and is represented in a high-level hand-engineered pattern language based on attacking and guarding relationships among the chess pieces in the pattern. The system produces a pattern for the state before a move and the state after a move, using a form of explanation-based learning to account for the differences.

5 Directions for Constructing Overlapping Features

Our primary goal is to develop methods that enable automatic construction of features for use in an evaluation function that is used to drive move selection. As stated at the outset, the system obtains feedback from its environment regarding the utility of the various outcomes. From this feedback, the system adjusts the parameters of its functional form to minimize evaluation error. How can features be constructed automatically in a manner that avoids the traps and weaknesses already described?

We would like to be able to decompose the state space into subsets that share intrinsic properties. Each such subset (concept) needs to be circumscribed by a feature definition that is true for elements within the subset and false for those outside. These subsets can overlap because a state may possess multiple properties simultaneously. The set of features should be compact and discriminating to the extent possible. The feature definitions should be found quickly with a minimum of human effort. Each feature should evaluate efficiently (without search or expensive matching costs) when applied to its inputs. Ideally, the features should be intelligible to humans, though this is not strictly necessary.

The next three sections discuss layered learning, compression afforded by individual computing elements, systems teachable through supervised problem decomposition and training, and feature compilation.

5.1 Layered Learning

It is well accepted that most concepts learned by a human are based on existing concepts that were learned earlier. Humans are gifted in their ability to create new knowledge from old knowledge, new experiences, or thinking. It is very efficient to layer knowledge in this way.

Otherwise, a person would need to learn each new concept from first principles. This would be a disastrous approach, with respect to efficiency of learning (survivability), storage efficiency, and with respect to understanding how particular elements of knowledge relate to one another.

Remarkably, most approaches to evaluation function learning attempt to learn while using just one (or sometimes two) layers of features (knowledge). This may be motivated by the fact that two hidden layers of features (units), with 'enough' features per layer, are sufficient to represent any evaluation function. However, sufficiency does not imply efficiency either in terms of learning time or space consumption. It implies efficiency only in terms of layers. Consider a simple analogous example from propositional logic. The sentence $((a \vee b) \wedge (c \vee d)) \vee ((e \vee f) \wedge (g \vee h))$ requires seven (7) binary operators and three (3) layers to evaluate, whereas the logically equivalent sentence $((a \wedge c) \vee (a \wedge d) \vee (b \wedge c) \vee (b \wedge d) \vee (e \wedge g) \vee (e \wedge h) \vee (f \wedge g) \vee (f \wedge h))$ requires fifteen (15) binary operators and two (2) layers to evaluate.

One possible motivation for minimizing the number of layers would be to facilitate the parameter adjustment scheme. However, layers of features model a problem decomposition, and gradient descent does not seem to do well when using many layers. Back-propagation of error in a many-layered network would model the process of learning all the layered knowledge at one time. If this were easy to do, we would expect to see humans do it, but that is not the case. We do not teach quantum mechanics to kindergarteners. Humans can build a new concept from the old when the new concept is not too different from the old. This is why we have curricula and teachers and parents, who help us learn more quickly by arranging new knowledge into a palatable order by accounting for the various dependencies known to the teacher but not the student. One important lesson that people learn is exactly the one just described; take on something new only if it requires learning a small amount, and go back to fill in background if that something new would require learning too much at once.

Consider a very simple example from the game of Othello. To play well, one must be able to recognize each and every available legal move for either of the two players in any state, because it is advantageous to minimize the number of legal moves for the opponent. A *legal move* exists at a square of the 8x8 board if and only if a bracketed span exists in any one or more of the eight lines emanating in one of the compass directions from the square. A *bracketed span* exists if and only if the starting square is empty, followed by one or more of the opponent's discs, followed by one of the player's discs. So, one must be able to recognize each and every available bracketed span for either of the two players in any state. The set of possible bracketed spans forms one feature layer, the set of legal move squares forms a second feature layer, and the number of moves available to each player forms a third feature layer.

One might attempt to capture this knowledge in fewer layers, but to what avail? Programs that use back-propagation of error to learn just a win/loss or disc-differential evaluation function do not find this feature decomposition. Some programs find features that are somewhat correlated with mobility, but they do not find features for the legal moves and the bracketed spans. Programs should not be expected to do well at learning many layers of features from scratch. People are poor at this as well. The rules of Othello explain bracketed span in order to explain legal move. One feature is used to define another because it is efficient to do so.

Present-day practice (Tesauro, 1992), dating back several decades (Samuel, 1963), is to engineer a set of input variables that enables a one or two layer learning algorithm to learn a good evaluation function. This is quite reasonable, given the limited capabilities of our parameter adjustment learning algorithms. It amounts to hand-coding the layers of features that need to have

Table 1. LTU for Spans in Same Direction

	s_0	s_1	s_2	s_3	s_4	s_5
empty	0	−8	−4	−2	−1	0
black	−8	−8	4	2	1	1
white	−8	0	0	0	0	0

been learned before being ready to learn the task set up by the program designer. We need to move beyond these 'last-layer' learning approaches. Instead of hand-coding the required prior features as input variables, we need to learn these too, to the extent possible. Instead of setting up a one-shot learning program, we need to set up an agent that can learn an assortment of related concepts (features) over an extended period.

We suspect that formulating an algorithm to construct a sufficient set of features automatically has been elusive for so many decades for two fundamental reasons. First, aiming for one or at most two feature layers is counterproductive. Second, one cannot expect a program to infer all the required knowledge that would be represented in the various layers from just the outcomes for the task of interest to the program designer. It is simply unrealistic to expect programs to learn all that is needed from scratch.

In the same vein, the learning task of learning to play a game is often formulated in an impoverished manner from the outset. For example, the entire notion of move availability is hidden in the game mechanics, and is therefore only implicit in the the play. This is quite different from human play; if a human makes an illegal move, the opponent or referee identifies it as such. How can we expect a learning program to uncover facts related to available legal moves if it is only ever presented with a list of available moves? One might be able to add some appropriate inference rules, but these kinds of fixes merely change the built-in mechanics of the game. It would be better to treat the concept of move legality as a subconcept for the program to learn. Indeed, as discussed above, the rules of the game state this concept explicitly, suggesting that it must be mastered independently of tactics and strategy for the game itself.

5.2 Compression

An important question is how to represent a feature. Some computing elements are better than others for a given set of inputs that need to be covered (grouped) by a feature definition. Consider again the problem of representing all the possible bracketed spans that emanate from a particular square of an Othello board. It happens that it is possible to represent all the bracketed spans that emanate in the same direction with just one linear threshold unit (ltu). Yet, it is impossible to represent with only one ltu even two spans that emanate in different directions.

To understand this apparent curiosity, consider a ltu as depicted numerically in Table 1. In this example, the threshold is assumed to be 0, so any sum of weights above 0 indicates presence of a span, and any sum of 0 or less indicates absence of a span. In the table, square s_0 is the square from which the bracketed span may emanate. If square s_0 (see column) is empty (see row), then the contributing weight for the linear combination is 0 (see cell entry), meaning that a bracketed span is still possible, to be determined by the rest of the pattern. Notice that if square s_0 contains either a black disc or a white disc, then the contribution is -8, indicating that bracketed span is impossible. Square s_1 must be white for black to have a span, so one sees a weight of 0 for white, and weights of -8 for empty or black. If square s_2 contains a black disc, then a span is present, and the weight of 4 causes the sum to be positive regardless of the remaining values. If square s_2 is empty, then a span must be absent, and if the square holds a white disc, a span is still possible. If one considers the various combinations, one sees that the sum is positive for just those patterns in which a span is present.

For spans in one direction, only one span can exist at a time, whereas for spans in differing directions, one can exist independently of another. A negative sum that would indicate lack of a span in *one* direction could unfortunately indicate lack of a span in *any* direction.

Given the large set of possible spans, it is noteworthy that they cluster into eight groups. Using one ltu for each direction, one needs eight ltus for those groups, and a ninth to represent a disjunction of the eight. This hints that a good teacher would offer this decomposition to the learner. Of course, a learner should not rely solely on the teacher, and should attempt to cluster inputs when possible. However, a good teacher facilitates learning, and we need to know how to integrate supervised and unsupervised compression of useful sets of inputs into feature definitions.

Unlike a linear threshold unit, a sigmoid threshold unit cannot necessarily represent all the spans emanating from a square in a single direction. One would need to tune the shape of the sigmoid explicitly. It is well known that fitting data points with a threshold-shaped function (sigmoid) can fail to find a separating plan that might exist (Duda & Hart, 1973; Freeman & Skapura, 1991). Using a poorly suited computing element for a feature definition can complicate the structure of the feature network, and slow learning.

5.3 Teachable Systems

We need a methodology for constructing systems that can learn a multitude of layered features individually and collectively. The notion of training a single output unit or output layer needs to give way to the idea that one can train features through means not limited to back-propagation of error. One needs to be able to train any element (feature) at any time. Returning to the Othello example, expert play depends on move presence, which depends on bracketed-span presence. The system needs to be able to accept supervised training of the various bracketed spans, or the various move presences at any time. Such capabilities can complement, rather than displace, methods that have been developed for last-layer learning.

The notion of layered learning may suggest that the layers be learned one at a time. This need not be the case as some layered knowledge may become advanced while other knowledge remains relatively undeveloped. Furthermore, inputs to a feature can come from any existing feature, so the layers may not be so tidy. Finally, one can always define new state variables later, and build new features at various levels above, long after development of other features. We need to know how layers of features can be constructed and modified incrementally.

The decision-making program (agent) should be able to interact with a human expert teacher, who will invent features that correspond to useful decompositions or views of the task domain. The evaluation function learner must automatically draw on the pool of features that it has learned over time. Several issues arise. One set of these surrounds the data structures and mechanisms for making it easy for the trainer to offer features and lessons at a comfortably high level. These same structures need to be accessible to the evaluation function learner.

The second set of issues surrounds the problem of specifying generalized statements about feature definitions. Stated another way, sometimes the trainer will want to provide training examples for a particular feature, and at other times the trainer will wish instead to provide a definition of the feature. Such a definition will be at a high level, and will need to be converted as discussed

Table 2. Application Interface Functions

```
double
  probe(char *unit_name);

fmem
  *create_fmem(),
  *load_fmem(char *file_name);

int
  save_fmem(fmem *f, char *file_name),
  save_unit(unit *u, FILE *fp),
  save_wire(wire *w, FILE *fp),
  set_value(fmem *f, char *unit_name, double v),
  train_value(fmem *f, char *unit_name, double v);

unit
  *lookup_unit(fmem *f, char *name);

void
  create_template(char *template_name,
                  char *input_unit_1, ..., char *input_unit_n),
  create_unit(fmem *f, function ft, char *name),
  create_wire(fmem *f, char *input_unit_name, char *output_unit_name,
              double weight),
  eval(fmem *f),
  eval_unit(fmem *f, unit *u),
  insert_unit(fmem *f, unit *u),
  load_units(fmem *f, FILE *fp),
  load_wires(fmem *f, FILE *fp),
  sort_units(fmem *f),
  update_layer_nums(unit *u);
  set_template_values(char *template_name,
              double *value_1, ..., double *value_n),
```

below in Section 5.3.2.

5.3.1 A Persistent Network of Layered Features

We have begun work on a data type for training and maintaining a persistent layered feature memory. This work is not yet well developed, but it is described here because it illustrates a path toward longer term learning of many-layered features. Each input variable is named symbolically. A new feature is created by giving it a symbolic name, by specifying the type of computing element it uses, and by specifying the symbolic names of its inputs. By using symbolic names, the underlying structures need not be known to the expert teacher or any other external entity. We assume throughout the discussion that a feature accepts Boolean input values and produces a Boolean output value. It may be desirable to accept real-valued input variables, but our emphasis is on features as intrinsic properties that are present or absent.

Table 2 sketches an incomplete set of functions that would be used to build, manage, train, and use a growing library of interconnected features. Because the inputs to a feature are state variables or other features, the set of features and input variables form a network of features, maintained as a data type. The data structures are loaded and saved as necessary so that the feature definitions persist and develop over time through use with a variety of applications. For example, if in one application a teacher has undertaken to educate the agent about the properties of a pair of six-sided dice, then the features associated with and trained for dice become available to other applications. This is one approach to effecting longer term learning and transfer.

To evaluate a feature, one specifies the values of its input variables on which it depends, and the output value of the desired feature is then computed by applying the function of the feature's computing element to its inputs, recursively as necessary. To train a feature, one provides additionally a target output value that the feature should produce for the given inputs. The update rule appropriate for the function of the computing element is applied.

Although different program can use the network in different ways, it will be common for a particular program to have repeated interactions of a particular form. For example, a training program may provide a sequence of training instances for refining the definition of a particular feature. It could become expensive to make all the necessary symbolic associations (lookups) for each training event. To circumvent this expense, one defines a template for such a series of interactions. The associations are found just once at the beginning of the sequence. Then the values of a training instance can be loaded into relevant elements of the network at once via the template in which the pointers to the elements have been stored when the template was defined.

One needs to forbid or handle carefully recurrent structures. We shall avoid recurrent structures for the near term. The evaluation function learner can use any input variable or any feature as an input. These values can be inputs to a single linear combination unit, or to a layer of sigmoid-threshold units, as in the customary approaches to last-layer learning. For learning an evaluation function, the connections from existing features to feature modeling the overall evaluation will be created implicitly.

5.3.2 Feature Compilation

There is little point in training a computing element empirically when its definition is well understood by a teacher and can be communicated. Instead of providing a stream of `(example,value)` pairs, one would like to be able to provide a definition in a high-level specification language. To do this, one needs a procedure for compiling each such specification into the data structures in which features are represented internally. A feature compiler has been constructed for this purpose (Stracuzzi & Utgoff, 2000). Consider the game of tic-tac-toe. Why learn from data to recognize the eight ways of forming three X marks in a line when one can specify such features relatively easily?

The compiler accepts statements in a somewhat extended first-order logic, and converts these to specific features modeled by simple computing elements. At present, a computing element (feature) can be one of AND, OR, NOT, or LTU. The main compilation mechanism is based on the presumption that quantified statements can be enumerated because the domain of each quantified logic variable (not state variable) is finite and small. The logic specification in Figure 2 shows how to specify the eight possible lines of three Xs that individually represent the presence of any of the eight simplest winning configurations. This specification abstracts the essence of three in a row.

The compiler is told the domain of each quantified logic variable, and is told the quantified

```
; Constants
Define: X = 0;
Define: O = 1;
Define: E = 2;

; Variables
Define: p1 = 0..8;
Define: p2 = 0..8;
Define: p3 = 0..8;

; Constraints
Set: p2>p1;
Set: p3>p2;

; Feature
Feature: XWON( p1, p2, p3) ←
   OR(
         ; Horizontal lines
         AND( p3=p2+1, p2=p1+1, p1 mod 3=0, S(p1, X), S(p2, X), S(p3, X)),
         ; Vertical lines
         AND( p3=p2+3, p2=p1+3, S(p1, X), S(p2, X), S(p3, X)),
         ; Diagonal lines
         AND( p3=p2+4, p2=p1+4, S(p1, X), S(p2, X), S(p3, X)),
         AND( p3=p2+2, p2=p1+2, p1=2, S(p1, X), S(p2, X), S(p3, X)));

; Feature
Feature: HASROW(X) ← EXISTS(p1,p2,p3)[ XWON(p1,p2,p3) ];
```

Figure 2. Specification X-has-three-in-a-row

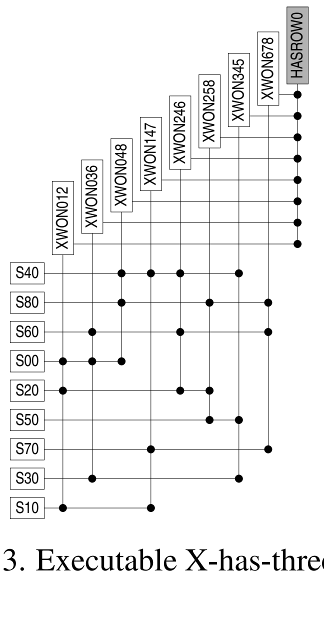

Figure 3. Executable X-has-three-in-a-row

definitions for the lines. The compiler then enumerates the possibilities, and produces low-level executable feature definitions that can evaluate quickly for a state. Figure 3 shows the features that were created automatically by the feature compiler for the specification of three-in-a-row given in Figure 2. The left-hand boxes of the figure indicate the input state variables, which are Boolean-valued, corresponding to presence or absence of X, O or Empty in each square of the board. The top boxes indicate feature definitions. In this figure, the label in each box was formed by copying the text of the logic statement and removing all commas. A shaded box indicates that the feature's computing element is the OR function, and a non-shaded box indicates the AND function. A solid dot indicates a positive connection from an input variable or feature to a feature. There are no NOT or LTU features in the figure.

For tic-tac-toe, one can specify all the high level features that are needed to play perfectly using a simple one-ply search. At the very least, a decision-making program must enumerate its choices and select the best. For the specification that includes all the needed high level features (not shown), the compiler creates 1,064 features in six layers in just seven seconds. These features were verified against a database of all tic-tac-toe positions. Recall that VU found 2,779 features for a single layer of features. Compared to the 5,477 possible positions, the six layers of 1,064 achieve about a 5:1 compression, which is much better. This is strong evidence in favor of using more than the minimum number of layers of features.

The particular approach we took for this application is a form of dynamic programming (Dijkstra, 1976), a form of goal regression (Waldinger, 1976), and a kind of static analysis of the domain (Etzioni, 1993). Having the features for the immediate wins, one can then generate the features for states in which an immediate can be produced in one move. This kind of feature bootstrapping leads one to features that recognize positions that will be won under perfect play. For other applications, it may very well be that one cannot construct features that enable perfect one-step decision making. Our goal is to provide a practical means of specifying useful features. For example, in Chess, one could specify a feature that is true when an opponent's piece is pinned by a bishop, rook, or queen. This feature enables learning a more accurate evaluation function, but it is just one aspect of position evaluation.

As discussed above, we have already begun to look at how to quantify logic variables over atomic objects in the domain. More recently, we have extended the notion of object to composites of objects. For example, consider Qubic, which is 4x4x4 tic-tac-toe. There are 76 lines of four cells. It is highly inefficient to quantify four logic variables over the 64 cells, test the 635,376 combinations for those that form one of the 76 lines, and then test whether some condition holds for that line. It is very much more efficient to define the 76 lines as 76 composite objects, and then quantify over that finite domain of lines.

5.3.3 Decomposition, Compression, and Intrinsic Properties

Features that make sense to an expert teacher may not be the most efficient means of decomposing the state space for the purpose of learning an accurate evaluation function. For example, as described above, all the Othello spans emanating from a particular square in a particular direction can be covered by just one linear threshold unit. It is not necessary to use a separate element for each span. Is this important, and if so, how is a teacher to know this? It is asking too much, given our present knowledge, but it would be highly useful to understand which sets of patterns are compressible into a single computing element such as a linear threshold unit. We need to study this.

To be able to experiment along these lines, we have developed an algorithm called PLM that finds a piecewise linear fit of the examples for each class. It is based on an older algorithm by Duda and Fossum (1966) that has been augmented to add new linear threshold units dynamically as needed during learning (Utgoff, 1989). The algorithm has some attractive properties that we are investigating separately in the context of classification algorithms.

The PLM algorithm was applied to a database of the 5,477 states of tic-tac-toe. Each state is labeled either as a member of class 'W' (win), class 'D' (draw), or class 'L' (lose). The database of labeled positions was produced in less than one second by a brute force searcher that assumed perfect play at all times. All states use the negamax representation, which means that X represents the player that has just moved. If in fact O has just moved, then in the state representation, all O will have been changed to X and all X will have been changed to O. This removes the need to record which player is on move.

After approximately 18 cpu hours on a DEC Alpha, PLM produced a perfect three-class piecewise linear fit of all the data. There were 46 ltus for class 'W', 56 ltus for class 'D', and 36 ltus for class 'L', for a total of 138 ltus (features). There is no guarantee that this is a minimum number of ltus. Nevertheless, this is a significant improvement over the 1,064 features that were produced using the feature compiler described above. Compared to the 5,477 positions, this is almost a 40:1 compression. This does not mean that compiling is a bad idea, but only that a better target set of computing elements would have been linear threshold units. The compiler ran in seven seconds, but PLM ran in eighteen hours. Compiling directly into LTU features would produce much stronger compression than Boolean features, at low cost. Consider an example from tic-tac-toe. One useful feature is whether a particular square is empty, and whether the remaining two squares in a particular line hold 1 X and 1 empty. A single linear threshold element can represent both cases, whereas one would need to recognize each case individually and OR them in Boolean logic.

Inspection of the 46 ltus for class 'W', and the set of examples covered by each, is informative. Many instances are covered by more than one ltu. An instance is a member of class 'W' if any of the ltus for class 'W' evaluates to a value above a small margin above 0. Of the 46 ltus, one finds an ltu for each of the eight lines of three in a row. Each cluster of instances covered by an ltu has certain commonalities. Given that an ltu represents a boundary in the Boolean hypercube, clustered instances represent hamming neighborhoods. We need to know much more about which sets of instances can be covered by a single ltu, and how to use a set of ltus to cover a set of instances in a piecewise manner. We need to identify whether there are any implications for how a teacher of features should decompose the space.

PLM finds a single layer of ltus that circumscribe a convex region. This may not be a good target either. By building useful layers of features, non-convex regions become describable. It is important to think about decompositions comprehensively with respect to layers of useful features.

6 Discussion

It appears important to investigate methods that enable construction of many layers of features. This follows from the observation that useful features are defined in terms of others, and from our desire to move beyond last-layer learning systems. Humans are not able to learn all features at once. Learning provides small incremental advances over what one already knows.

If learning of many-layered features is to occur over an extended period of time, with a variety of interactions and agents, then it will be important to develop a facilitating data type. We need an interface whereby a human or separate program can define features by specifying a symbolic name for it, its computing element, and the symbolic names of its inputs. These features can be trained independently at any time, leaving alone other connected features in the network of features. All features can serve as inputs to the last element or layer of elements that compute the evaluation of the state.

In addition to facilitating supervised construction and training of features for a many-layered network, it will useful to study feature compilation methods that convert high level feature specifications into low level features or feature sets represented in the underlying data structures. Although we have a preliminary version, there is much exploration yet to be done regarding more powerful specification languages and more succinct target representations. Several issues regarding task decomposition and information compression need to be addressed.

One should consider whether proposing to build a long-lived network of feature definitions is just pushing the hand-coding of features out of one programming process (application program) and into another (constructing and training features). On the surface this may be so, but there is much more to it than that. Many layers of features offer significant representational advantages. Furthermore, an agent should be able to benefit from an organized curriculum of instruction, interleaved with learning from experience.

It is quite common to employ artificial neural networks for problems in which a last layer (or two) of features can be learned adequately. Artificial neural networks can be highly useful in this role, and they perform magnificently when the inputs have been properly engineered. However, they can become trapped at local minima, and they can fail to find features that would provide improved evaluation accuracy. We need to have a broader scope when addressing the problem of constructing useful features for evaluation function learning. Several ideas have been proposed here to this end. It will be fruitful to consider many layers of features, supervised teaching of features, feature compilation, problem decomposition, and unsupervised reorganization and compression of existing features.

Acknowledgments

This work was supported by National Science Foundation Grant IRI-9711239. The discussion at the 1999 ICML Workshop on Learning and Game-Playing was most helpful, particularly Donald Michie's remarks about learning in stages. David Stracuzzi built the feature compiler. Margaret Connell, Gang Ding, and Richard Cochran provided helpful comments.

References

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

Berliner, H. (1979). The B* tree search algorithm: A best-first proof procedure. *Artificial Intelligence, 12*, 23-40.

Berliner, H. J. (1984). Search vs knowledge: An analysis from the domain of games (pp. 105-117). In Elithorn & Banerji (Eds.), *Artificial and Human Intelligence*. New York: Elsevier Science Publishers.

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

Buro, M. (1998). From simple features to sophisticated evaluation functions. *The First International Conference on Computers and Games*. Tsukuba, Japan.

Callan, J. P. (1993). *Knowledge-based feature generation for inductive learning*. Doctoral dissertation, Department of Computer Science, University of Massachusetts, Amherst, MA.

Caruana, R. (1997). Multitask learning. *Machine Learning, 28*, 41-75.

Dijkstra, E. W. (1976). *A discipline of programming*. Prentice-Hall.

Duda, R. O., & Fossum, H. (1966). Pattern classification by iteratively determined linear and piecewise linear discriminant functions. *IEEE Transactions on Electronic Computers, EC-15*, 220-232.

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

Etzioni, O. (1993). Acquiring search-control knowledge via static analysis. *Artificial Intelligence, 62*, 255-301.

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

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

Frean, M. (1990). The Upstart algorithm: A method for constructing and training feedforward neural networks. *Neural Computation, 2*, 198-209.

Freeman, J. A., & Skapura, D. M. (1991). *Neural networks: Algorithms, applications, and programming techniques*. Addison-Wesley.

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.

Lee, K. F., & Mahajan, S. (1988). A pattern classification approach to evaluation function learning. *Artificial Intelligence, 36*, 1-25.

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

Press, W. H., Flannery, B. P., Teukolsky, S. A., & Vetterling, W. T. (1988). *Numerical recipies in C: The art of scientific computing*. New York: Cambridge University Press.

Rosenbloom, P. (1982). A world-championship-level Othello program. *Artificial Intelligence, 19*, 279-320.

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

Samuel, A. (1963). Some studies in machine learning using the game of Checkers (pp. 71-105). In Feigenbaum & Feldman (Eds.), *Computers and Thought*. New York: McGraw-Hill.

Suddarth, S. C., & Holden, A. D. C. (1991). Symbolic-neural systems and the use of hints for developing complex systems. *International Journal of Man-Machine Studies, 35*, 291-311.

Stracuzzi, D. J., & Utgoff, P. E. (2000). *Feature compilation*, (TR-00-18), Amherst, MA: University of Massachusetts, Computer Science Department.

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

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

Utgoff, P. E. (1989). Perceptron trees: A case study in hybrid concept representations. *Connection Science, 1*, 377-391.

Utgoff, P. E., & Clouse, J. A. (1991). Two kinds of training information for evaluation function learning. *Proceedings of the Ninth National Conference on Artificial Intelligence* (pp. 596-600). Anaheim, CA: MIT Press.

Utgoff, P. E., & Precup, D. (1998). Constructive function approximation (pp. 219-235). In Liu & Motoda (Eds.), *Feature extraction, construction, and selection: A data-mining perspective*. Kluwer.

Utgoff, P. E., & Stracuzzi, D. J. (1999). Approximation via value unification. *Proceedings of the Sixteenth International Conference on Machine Learning* (pp. 425-432). Ljubljana: Morgan Kaufmann.

Waldinger, R. (1976). Achieving several goals simultaneously (pp. 94-136). In Elcock & Michie (Eds.), *Machine Intelligence*. New York: Wiley & Sons.

Watkins, C.J.C.H., & Dayan, P. (1992). Q-Learning. *Machine Learning, 8*, 279-292.

Werbos, P. J. (1977). Advanced forecasting methods for global crisis warning and models of intelligence. *General Systems Yearbook, 22*, 25-38.

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