Towards Using Neural Networks to Perform Object-Oriented Function Approximation

Dennis Taylor, Brett Bojduj, and Franz Kurfess

Abstract— **Many computational methods are based on the manipulation of entities with internal structure, such as objects, records, or data structures. Most conventional approaches based on neural networks have problems dealing with such structured entities. The algorithms presented in this paper represent a novel approach to neural-symbolic integration that allows for symbolic data in the form of objects to be translated to a scalar representation that can then be used by connectionist systems. We present the implementation of two translation algorithms that aid in performing object-oriented function approximation. We argue that objects provide an abstract representation of data that is well suited for the input and output of neural networks, as well as other statistical learning techniques. By examining the results of a simple sorting example, we illustrate the efficacy of these techniques.**

I. Introduction

The objective of this work is to demonstrate the feasibility of using neural networks for the manipulation of structured computational entities, such as objects. One specific example of such a method is object-oriented function approximation: if given a collection of three unsorted integers, is it possible to create a statistical model that can accurately approximate the sorted collection of those same integers? While this is clearly a simplified case of manipulating such structures, it will allow us to examine the basic principles underlying our method, and to demonstrate its feasibility. It is clear that further work is needed to examine other aspects, such as scalability (more complex structures, larger data sets), performance (time, space), and the use of object-orient function approximation for industrial applications.

In considering the answer to the above question, let us consider the following training data of input and output pairs.

```
{1,5,3}->{1,3,5}
{7,5,3}->{3,5,7}
{2,5,5}->{2,5,5}
{8,4,6}->{4,6,8}
{9,0,3}->{0,3,9}
...
```

Given an input within the same range of training data, but with a previously unobserved collection of integers, such as {4,5,3}, the model should be able to approximate the output to {3,4,5}. Although this is a rather trivial problem and fast algorithms exist to solve sorting collections of arbitrary sizes, it remains a difficult problem for neural networks to solve [1]. This work presents an approach to simplify this and other problems by representing the data as objects. In object-oriented design and programming, an object is a concrete instance of a type of class that consists of member data and is associated with a set of operations on that data [2]. Throughout this paper, object-oriented (OO) function approximation will have the following definition:

> Given a set of input and output object pairs, train a model so that it can quickly approximate the output object from a given, unseen input object.

In order to use a traditional form of approximation such as a neural network to approximate the sorting problem mentioned, a first step would be to find a suitable representation for the three integers. Typically this representation would be a sequence of vectors, but for less trivial examples the representation may not be immediately obvious. Our work began because of a need to simplify the representation of approximated data inputs and outputs. There are many current OO systems that could benefit from approximation techniques but lack the scalar representation necessary to explore this option. The problem of using symbolic data with connectionist systems has been studied a great deal in the past [3], [4], [5]. These approaches typically focus on logic programs and not at more complex data structures, but some research has focused more recently on complex structures which will be discussed in detail later in this document. Objects allow a problem's representation to be handled at a level of abstraction which is more readily understandable to a human. The advantages of using objects for the representation of data is evident in the widespread use of OO languages in present day software development, but has also been studied academically. A paper by Bennedsen et al. [6] presents findings that claim general abstraction has a positive impact on programming ability. Also, by preserving object identity we show that data can be automatically used in a categorical manner resulting in improved approximation.

One broad class of operations that could benefit from the use of the techniques described here is to determine the similarity of structures. While this can be solved with traditional methods (assuming that a similarity measure is defined), it can be beneficial in some situations to do a quick initial comparison of the structures, and only investigate more carefully if they are "reasonably similar." Instances of such similarity-based problems are the merging of database and ontology schemata, mapping of specific objects with certain attributes to other objects or classes (such as in case-based reasoning), or the comparison of structured entities, such as drawings.

Dennis Taylor, Brett Bojduj, and Franz Kurfess are with the Department of Computer Science at the California Polytechnic State University in San Luis Obispo. (email: {djtaylor, bbojduj, fkurfess}@calpoly.edu).

978-1-4244-1821-3/08/$25.00

II. Compositional Functions

The problem of translating structured data into a suitable representation for a neural network to use is not new. A growing body of work on this topic exists.

The simplest form of translation consists of concatenating Huffman encoded sets together to produce a binary or scalar representation of the various parts of the structure in a systematic way [7], [8]. This can be done using a depth-first traversal of the structure that visits each node and then concatenates the Huffman encoding for each node to the end of a representative sequence of numeric values. The examples used later in this document are of this particular flavor. In order to compensate for scaling problems with this technique and to address other possible concerns, several approaches which do not use concatenation have been considered.

One solution to this problem which comes from cognitive science is Sparse Distributed Representations (SDR). Categorical symbols (or concepts) are represented by random vectors or sets of random vectors. By using random vectors, SDR provides a programmatic encoding scheme which is well-suited for representing concepts, due to the fact that SDR does not inherently assign biased meaning to the vectors. The meaning for each vector is provided in the form of a cleanup memory that maps vectors to their assigned concept. SDR can be used for not only unbiased representation of concepts, but also for encoding the relations between concepts [9]. Two types of relations that are easy to encode using SDR representations include variable binding [10] and chunking [11]. Variable binding allows two concepts to be bound to each other. For example, the concept "EYE" can be bound to the concept "BLUE" to indicate that the variable 'eye' is bound to the color 'blue'. Chunking refers to the relationship where a list of concepts are grouped together to indicate that they belong to the same set. If a function can be used to combine two vectors into a third vector while maintaining all of the information about the first and second vectors, it can be considered a compositional function [12].

One implementation of an SDR is provided by Plate's [13] Holographic Reduced Representation (HRR), which uses a compositional function called a circular convolution, to encode recursive structures. The following mathematical definition is paraphrased from Jane Von-Neumann's paper [14] on learning HRR transformations, which provides a clear and well defined background description of HRRs.

Circular convolution binds two n-dimensional vectors:

$$X = (x_0, ..., x_{n-1}) \tag{1}$$

and

$$Y = (y_0, ..., y_{n-1}) \tag{2}$$

to a n-dimensional vector,

$$Z = (z_0, ..., z_{n-1}) \tag{3}$$

such that Z = X (circular convolution) Y with

$$z_j = \sum_{k=0}^{n-1} x_k y_{j-k} \tag{4}$$

for j = 0,....,n-1 (Note: The subscripts are computed modulo n.)

If the elements in each n-dimensional vector are independently and identically distributed over N(0,1/n), then decoding elements from bindings is possible using the approximate inverse to circular convolution, circular correlation, which is defined as:

(approximate X) = Z (circular correlation) Y with

$$x_j = \sum_{k=0}^{n-1} z_k y_{j+k} \tag{5}$$

for j = 0,...,n-1 (Note: The subscripts are computed modulo n.)

The approximate decoding of X provides the involution of X and is not exact; therefore, it is required to have clean-up memory that contains all possible elements of the domain to fully restore the value of X. High dimensional vectors are used to attempt to maintain the independence and even distribution of individual domain values. The larger the dimension of the vector is, the less likely it is that a random vector from the sample will be closely related to another random vector. The use of high dimensional vectors does not immediately provide a high cost since Plate [13] showed that by using Fast Fourier Transforms, circular convolution and correlation can be computed in O(n log n) time. Often this results in huge vectors being used, around the order of 2^{12} (2^{12} = 4,096). Using HRR this results in random vectors generated over N(0, 1/4096).

Another technique for computing SDR representations which is very similar to HRRs is binary spatter codes [9]. Instead of using numeric vectors, concepts are encoded as random binary sets. Again, variable binding is used to group key value pairs. The compositional function used for binding is the binary operator XOR (exclusive-or). Record chunking is used to group a set of symbols or key value pairs into a single set of bits. This operation is done by using the following rules.

Chunking is done by summing the number of ones in any bit-wise position. If the sum is greater than half of the number of chunked items, then the value is one. If the sum is less than half of the number of chunked items, then the value is zero. If the sum is exactly half, then the value is either one or zero, determined by a random 0.5 probability.

Recall (i.e., retrieving the value) is performed by XOR'ing the reduced representation with the desired key bit set, since XOR is its own inverse function. This will return another bit set which represents the value bound to that key, plus some small amount of noise. The noise is reduced by using a cleanup memory similar to that of HRRs. By using this approach with a large bit set representing each symbolic entity, an immense number of symbols can be represented. If the bit set has n bits then the number of possible unique

combinations is 2^n, as we have already seen with the numeric vectors of HRRs.

Although research into using distributed representations is still underway, the specific algorithms used in this paper here use a concatenating composition. Currently this is done because the distributed representations tested so far have not provided desirable arithmetic approximation results for continuous domains such as Real numbers or their computer equivalent floating point values.

III. Translation Algorithms

In order to leverage the huge body of work on machine learning and connectionist systems, this paper introduces an algorithm for object-to-scalar translation that will allow any statistical model which uses a sequence of vectors as input and output to be used for the underlying approximation approach. The algorithm must first perform pre-processing on a set of training data to gather information about the structure and distribution of variable values within the objects. It will then use this information to examine each object and translate it into a set of numeric values between 0 and 1. The next section will elaborate on this process.

IV. Pre-Processing

Every input and output object pair is examined by the pre-processor. During this stage the possible types and values of each member data variable is looked at recursively. The end result of this phase is a mapping from object structural positions, to observed variable domain values. Special care should be taken for certain language-specific variable types like arrays, maps, and collections, but for the sake of brevity they are not mentioned here.

The psuedo-code listed illustrates here shows how pre-processing is done. There are several terms used in this code which need to be described first. The term label refers to a sequence of characters or a string that represents the structural position of an object or a primitive member data field. The term VariableDomain refers to a class Object that has one method that is used in pre-processing called examineValue which takes an observed data value. The VariableDomain also contains two other methods to be used in the later stages after pre-processing which translate an observed object value into a numeric value between 0 and 1, and a method for translating numeric values between 0 and 1 back into observed objects. The getTypeDomain method can be implemented with common a map structure which maps a String/Label to a VariableDomain which holds information about the Types observed in each structural position. The getDomainSet can be implemented in a similar fasion with a map from a String/Label to another map containing Type to VariableDomain maps which hold the actual observed value variable domains.

So in this way the pre-processing stage is responsible for mapping each structural label observed in the training samples to a VariableDomain object which will record the observed values and create a Huffman encoding for them. In the analysis of this approach presented later we used a linear interpolation for each VariableDomain. For example the integer value 3, found between possible values of 0 and 6, would be translated to .5 since it is half way between the highest and lowest value observed. The next sub-section will present a more detailed example of this generic pre-processing phase.

```
// pre-processing
foreach (pair(Input, Output) in TrainingSamples) {
   examineAndLabel(Input, "in");
   examineAndLabel(Output, "out");
}

examineAndLabel(Object parent, String parentLabel) {
   VariableDomain domain = getTypeDomain(parentLabel);
   domain.examineValue(typeof(parent));
   if (typeof(parent) is primitive) {
      Map<Type, VariableDomain> domains
        = getDomainSet(parentLabel);
      VariableDomain domain = domains.get(typeof(parent));
      if (domain is null) {
         domain = createDomain(typeof(parent));
         domains.put(typeof(typeof(parent), domain);
      }
      domain.examineValue(parent);
   }
   else {
      foreach (Field child in parent.memberData) {
         examineAndLabel(child.objectData,
           parentLabel + child.label);
      }
   }
}
```

Fig. 1. Pseudo Code

A. XOR Example

The classic XOR (exclusive-or) example will be used to illustrate how this algorithm works. This example was chosen for its simplicity and notoriety and should not be considered a representative sample of the type of problems that would actually be approximated using this approach.

In the XOR example, there is one possible input type, named Case. This object has two boolean member data variables, named "b1" and "b2." The output variable is a boolean representing the result as the XOR of b1 and b2. Four objects are created for the training data for the XOR problem. The output has a "true" boolean value only when "b1" and "b2" are different. Otherwise the output is the boolean value, "false."

From this very simple example, the type of data that can be gathered for each variable's possible domain is somewhat limited, but still provides some insight into how the gathering of data is done. The pre-processing of each of the Case object instances will result in the following domain mappings for each variable:

"in1"↦TypeDomain with the Case type as the only observed type.

"in1_Case_b1"↦TypeDomain with boolean as the only observed type

"in1_Case_b1_boolean"↦BooleanDomain with observed values 0 and 1.

"in1_Case_b2"↦TypeDomain with boolean as the only observed type

"in1_Case_b2_boolean"↦BooleanDomain with observed values 0 and 1.
"out"↦TypeDomain with the boolean type as the only observed type.
"out_boolean"↦BooleanDomain with observed values 0 and 1.

The variable domain in each of these cases is a BooleanDomain, which can only have the binary values 0 or 1. In our implementation there exists a default variable domain for common language-specific (i.e., C, C++, Java, etc.) primitive types, including: integers, longs, floats, doubles, booleans, chars, strings, enums, and types. These domains keep track of the observed values or ranges of values so that they can provide the appropriate translation later. Currently they all work by applying only a linear interpolation between the values, which gives every value an equal chance. More complicated domains could be created that favor values that have been seen more often and therefore provide different distributions. So far this has not been a problem since the learning algorithm has mitigated the need for more complicated distributions.

If the variable "b1" was not limited to only a boolean, for example, but if it could be any object in the variable domain, then the mapping would be more elaborate. If there were several other cases where "b1" was an integer ranging from 2 to 7, another variable domain would be added to the previous list and the type domain would also have another value, such as:

"in1_Case_b1"↦TypeDomain with boolean and integer observed
"in1_Case_b1_integer"↦IntegerDomain with observed values from 2 to 7

Creating a mapping from each object's structural position to a variable domain was done to preserve as much structural information as possible and limit each structural position to defining only the possible domains observed in that position. This increases the chances for successful approximation for certain problems, because creating a mapping from the structural position to the variable domain limits the amount of variance in the possible objects that can be generated. It also makes it so that some functions cannot be approximated using this technique. This limitation and possible alternative solutions will be discussed in more detail later in this contribution.

V. Object to Scalar

Figure 2 illustrates how an object is translated to a scalar using the table created during the pre-processing phase.

If the Case object instance that contains the variable values "b1" = true and "b2" = false is translated using the algorithm from Figure 2, then we would get the following scalar values:

0.5 - three instances exist in the input: the input case, "b1," and "b2." The input case was the first observed instance.
1 - Case is the only possible type in the TypeDomain for "in1"

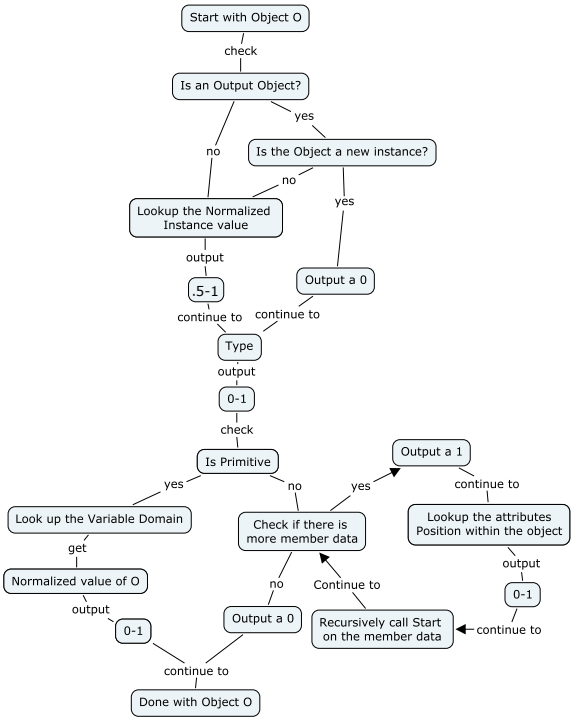

Fig. 2. Object to Scalar Conversion

1 - there is more member data to output
0 - the variable "b1" is the first settable attribute on a Case object.
0.75 - the variable "b1" is the second observed instance from the input.
1 - the variable "b1" has the value true so its boolean domain returns 1
1 - there is more member data to output
1 - the variable "b2" is the last settable attribute on a Case object.
1 - the variable "b2" is the third observed instance from the input.
0 - the variable "b2" has the value false so, the domain returns 0
0 - all of the member data values have been output

So the ending vector representation for this input instance would be $\{0.5, 1.0, 1.0, 0.0, .75, 1.0, 1.0, 1.0, 1.0, 0, 0\}$. The equivalent set of vectors from classical examples of the XOR problem would only be $\{1.0, 0\}$, representing only the boolean true and false. It may seem that there is far more data than is necessary to state the actual problem, and for this trivial example that is true, but for more complicated examples, the extra data may help preserve context as will be shown later. The translated output object for this example instance would be:

0 - the object is a new instance
1.0 - boolean is the only possible type in the

TypeDomain for “out”
1.0 - the boolean domain for “out_boolean” returns a 1 for the true value

So the vectors $\{0, 1.0, 1.0\}$ indicate that the output object is a new instance of type boolean with the value true. Again, if we compare this to the classical XOR example we would only have the value $\{1\}$. Out of all of the four training examples, the only value that changes will be the last value which will be either 0 or 1. So the amount of variance in the output object is the same as the classical example. In order to recognize this, the neural network must weed out the extra data, or it must be done in a secondary pre-processing step which can lead to more time-consuming training. This is one of the trade-offs that should be considered when using this approach. In the analysis section of this paper we show in the sorting example how this extra data can become a benefit.

VI. Choosing an Approximation Approach

There are many pattern recognition and machine learning techniques that could be used for the approximation method. One common trend that appears to be consistent amongst most machine learning techniques is that they all can be explained using a statistical model. For instance, depending on the type of neural network one chooses, it ”uses different ways to estimate or approximate the posterior probabilities from the training set” [15]. This is important to note, since throughout this paper, neural networks are used as the example of connectionist learning, though they are not the only possible choice. Once the Objects have been translated into a scalar representation a wide variety of learning algorithms could be used to perform the approximation. Choosing an appropriate approximation approach will make a significant difference in the results of this technique.

Our tests so far, written in Java with the JOONE toolkit, have used a standard feed-forward neural network trained using back-propagation with a sigmoid output layer. During these tests, the size of the hidden layer, learning rate, and momentum have been changed to provide varying behavior. Ideally, a self-configuring network would be used so that no user involvement would be needed to adjust these parameters. Future plans for this work include integrating something like the MLEANN [16] evolutionary neural network system or other techniques to reduce the amount of parametrization and guessing of structure required.

VII. Scalar to Object

The creation of objects from a scalar representation works similarly to the translation to scalar, but backwards. Each numeric value from 0 to 1 is read from the beginning of the set and has a predefined meaning based on its position according to Figure 3, as well as the pre-processing lookup variable domains. Objects are either created because they were not previously observed in the input objects, or they are attached directly from the input instances. Their member data is then translated recursively until the full output object has been constructed. Figure 3 illustrates how an object is created from a series of numeric values.

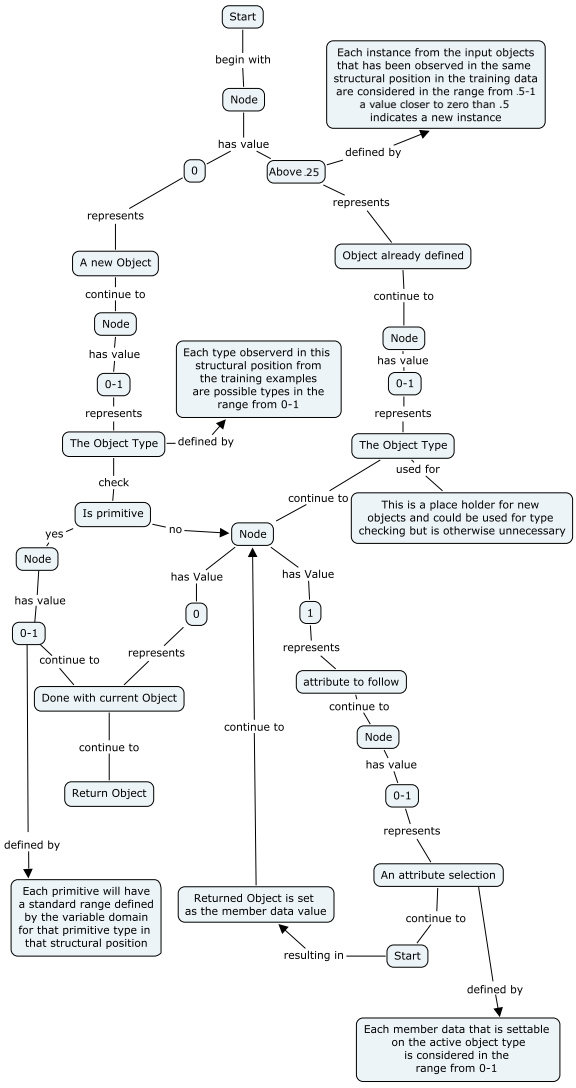

Fig. 3. Scalar to Object Conversion

VIII. Sorting Analysis

The sorting problem was chosen as an example to show that using this technique for generic OO function approximation is possible in principle. The first experiment shows a comparison of using the same neural network for sorting sets of integers with varying sizes while preserving object identities. The second experiment did not preserve object identities. Each experiment was run by creating sets of integers of size n. Each integer was chosen at random from the range of integers between 0 and 100. The network was trained on 500 sets and their sorted counterparts. In the first experiment, the same integer objects were used in both sets. In the second experiment, new integer object instances with the same primitive value where created. This was done so

TABLE I
SORTING IN PRINCIPLE

Set Size	% Correct	Random Chance	Relative Diff
2	99.3	50	1.986
3	83.2	16.666	4.992
4	33.5	4.166	9.041
5	4.9	0.833	5.88
6	2.3	0.138	16.66
7	0.4	0.019	21.05

the translation algorithm would not preserve their identities. After the networks had been trained on the training sets, they were tested on another 1,000 sets of random integers to see how many of the sets could be properly sorted. In the results of the first experiment (Table I) it is clear that the percentage of correct results for each set size is well above the percentage of correct sorting that would be expected from permuting the integers in any random order. This probability was calculated with the equation 1.0/n!. The probability represents the chance that any random permutation of the set of n integers is the correctly sorted set. From this comparison, it is clear to see that the network consistently performs better than random, indicating that it has learned to approximate sorting in principle.

TABLE II
RESULTS WITH INSTANCE PRESERVATION

Set Size	% correct w inst	Avg Int Off w inst
2	99.3	1.705
3	83.2	19.272
4	33.5	23.659
5	4.9	27.943
6	2.3	28.189
7	0.4	28.389
8	0	29.134
9	0	29.153

TABLE III
RESULTS WITHOUT INSTANCE PRESERVATION

Set Size	% correct w/o inst	Avg Int Off w/o inst
2	1	3.745
3	0.1	3.55
4	0	4.253
5	0	4.83
6	0	4.78
7	0	4.80
8	0	4.95
9	0	5.13

The second experiment compares the results presented in the first experiment to a trial run where integer instances were not preserved. Tables II and III show this comparison. The results show that without considering instances, the network is rarely exactly correct. When an integer was incorrect we checked how far off from the correct value it was. Without considering the instances, the neural network will often be close to correct, but will make up new integers. The values {96, 53, 12}, could result in a collection of three new integers, like {13, 51, 98}. Although this result is close to correct, none of the initial integers are in the resulting set. This can make the data hard to use. One trade off is that if the instances are preserved it can increase the average number of integers the values can be off. The values {16,12,15}, could result in {12,15,12}, here the average incorrect integer is off by 4 where as in the previous sets the average incorrect integer was off by just under 2. Although the average incorrect integer is further off, it may be more desirable to use this approach since the objects in the output are instances taken directly from the input. It is interesting to note that we tried several more experiments after getting zero percent of the sorting perfectly correct and noticed that without instance preservation when the set size was increased to 8 and 9 the average number of integers off grows rather slowly as the set size is increased, so when exact values are not necessary, closer values can be obtained by ignoring instance preservation.

The preservation of object identity allows variables with possibly wide-varying ranges to be chosen from only the set of available input instances. This allows the translation to provide an automated categorical interpretation of these values. In some problems this could be a huge benefit, since modern object-oriented languages rely heavily on object identity to test for true equality. If two objects are actually the exact same instance, then they will undoubtedly be equal. This knowledge can facilitate more efficient algorithms by doing things such as not having to duplicate the work of creating new objects when preserved instances could be readily available. Many algorithms rely on this to perform certain operations, such as looking up objects in a hash map to guarantee object equality after the hashing has been done.

IX. FUTURE WORK

As mentioned previously, the pre-processing technique that is currently being used is somewhat limited. If you consider a graph problem or something that similarly has highly irregular structure, this approach would not be able to account for all of the various domain values that could exist in conceivably many structural positions, without a training example for every case. Work on integrating structured data, such as graphs, has been done using recurrent neural networks and has shown reasonable results [17], [18]. The pre-processing stage could be modified to support more varied data structures by using a more flexible variable domain mapping. This would make the position of the variable within a specific input or output object to be inconsequential and variable domains would be mapped only to their position within the immediate parent object. Doing this will make the training more difficult since more values can exist at any given structural position within the object [19]. However, it would allow structures to be created that have yet to be observed from the training data. More experimentation with this approach would be needed to evaluate its possible

downfalls as well as potential benefits. If the extra data for object identity and type preservation increases training time to an unreasonable amount it may be necessary to look into some of the techniques for input variable selection, like those used by Herrera et al. [20].

X. Conclusions

Our results from sorting have shown that the translation algorithms presented can be used as a step towards Object Oriented function approximation in principle. As mentioned, the choice of statistical model and network structure used for the approximation step will be paramount to the future success of this work. The presented algorithms successfully provide the ability to translate objects into a usable scalar form and allow the creation of objects from the scalar output of a statistical model. However, it does not currently allow all algorithms to be approximated using this technique. The hope of this work is that by allowing the statistical models to perform both numeric and structural approximation within the same context, it may be possible to approximate a subset of object-oriented functions quickly, with an acceptable degree of error. The authors of this paper feel that the integration between object-oriented programming and statistical modeling is a possible step towards neural symbolic integration and could rapidly increase the production of systems that can benefit from sacrificing precision for performance.

References

[1] Y. Takefuji and K. Lee, "A super-parallel sorting algorithm based on neural networks," *Circuits and Systems, IEEE Transactions on*, vol. 37, no. 11, pp. 1425–1429, 1990.

[2] IBM, "Ibm reference glossary," `http://www-03.ibm.com/ibm/history/reference/glossary_o.html`, 2007.

[3] F. Kurfess, "Hybrid systems: Integrating ai and neural network techniques," in *From Synapses to Rules: Discovering Symbolic Rules from Neural Processed Data*, B. Apolloni and F. J. Kurfess, Eds. International School on Neural Networks series, Kluwer Academic Publishers, 2002, pp. 275–292.

[4] F. J. Kurfess, "Neural networks and structured knowledge: Representation and reasoning (part. 1)," *(guest editor) Applied Intelligence*, vol. 11, no. 1, 1999.

[5] ——, "Neural networks and structured knowledge: Rule extraction and applications (part. 2)," *(guest editor) Applied Intelligence*, vol. 12, no. 1/2, 2000.

[6] J. Bennedsen and M. Caspersen, "Abstraction ability as an indicator of success for learning object-oriented programming?" *ACM SIGCSE Bulletin*, vol. 38, no. 2, pp. 39–43, 2006.

[7] D. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[8] S. Klein, "Skeleton Trees for the Efficient Decoding of Huffman Encoded Texts," *Information Retrieval*, vol. 3, no. 1, pp. 7–23, 2000.

[9] P. Kanerva, "The spatter code for encoding concepts at many levels," *ICANN*, vol. 94, pp. 226–29, 1994.

[10] A. Browne and R. Sun, "Connectionist variable binding," *Expert Systems*, vol. 16, no. 3, pp. 189–207, 1999.

[11] T. A. Plate, "Holographic reduced Representations: Convolution algebra for compositional distributed representations," in *Proceedings of the 12th International Joint Conference on Artificial Intelligence, Sydney, Australia, August 1991*, J. Mylopoulos and R. Reiter, Eds. Morgan Kauffman, 1991, pp. 30–35.

[12] T. van Gelder, "Compositionality: A connectionist variation on a classical theme," *Cognitive Science*, vol. 14, no. 3, pp. 355–384, 1990.

[13] T. Plate, "Distributed Representations and Nested Compositional Structure," *Ph.D. Thesis, Graduate Department of Computer Science*, 1994.

[14] J. Neumann, "Learning the systematic transformation of holographic reduced representations," *Cognitive Systems Research*, vol. 3, no. 2, pp. 227–235, 2002.

[15] B. Ripley, *Pattern Recognition and Neural Networks*. Cambridge University Press, 1996.

[16] A. Abraham, "Meta-learning evolutionary artificial neural networks," *Elsevier Science, Netherlands*, vol. 56c, pp. 1–38, 2004.

[17] M. Bianchini, M. Maggini, L. Sarti, and F. Scarselli, "Recursive neural networks for object detection," *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, vol. 3, 2004.

[18] A. Sperduti and A. Starita, "Supervised neural networks for the classification of structures," *Neural Networks, IEEE Transactions on*, vol. 8, no. 3, pp. 714–735, 1997.

[19] A. Chortaras, G. Stamou, and A. Stafylopatis, "Learning ontology alignments using recursive neural networks," *Lecture Notes in Computer Science*, vol. 3697, p. 811, 2005.

[20] L. Herrera, H. Pomares, I. Rojas, M. Verleysen, and A. Guilen, "Effective input variable selection for function approximation," *Lecture Notes in Computer Science*, vol. 4131, pp. 41–50, 2005.