Programming Language Support for Autoassociative Memory

Dept. of CIS - Senior Design 2014-2015

ABSTRACT

Data structures such as arrays or hasmaps use exact key lookups and memory addresses for retrieving entries. This differs from a more relational model of memory such as human memory, which learns from heuristics and pattern matching as it receives input from its surrounding environment. Analogously, an autoassociative memory receives an input and returns a output value that may only be similar to the input and to the other memory values according to learned patterns. Our programming language library seeks to introduce a novel interface to this autoassociative memory in order to facilitate programming efficiency for the end user. Specifically, the interface will provide an easier way to handle structured input data for the user, as well as managing user-defined types in the autoassociative memory.

1. INTRODUCTION

Autoassociative memories are frequently used in machine learning applications, and while these applications generally handle data in the form of plain, flat vectors, there are certain applications where programmers would want to preserve their highly structured, rich representations of data. For instance, machine learning tasks related to NLP (Natural Language Processing) often employ n-ary parse trees, and these parse trees can become quite complicated due to their recursive nature and non-binary branching. Moreover, sentences or phrases take on different semantic meaning due to their structure, so it would clearly behoove programmers to somehow preserve structure in their applications. In response, this programming language library offers the end user with an intuitive interface to autoassociative memory that preserves the structure of data and gives them the capability to define their own custom data types.

First, this paper will discuss background information relevant to autoassociative memory, structured data, and machine learning in order to contextualize the more technical aspects of the library. Next, the paper will delineate research related to autoassociative memories and give an overview of specific methods that were used in the library. Then, it will establish a model or workflow of the interface, and the process of using this interface for autoassociative memories in machine learning applications. Afterwards, it will delve into the implementation details, and discuss the language and third-party libraries that were used. Finally, the performance and functionality of the library will be assessed, and possible improvements and new features will be discussed.

2. BACKGROUND

2.1 Machine Learning

Autoassociative memory is part of the much broader topic of machine learning. Machine learning is widely used today to solve problems that lack easily definable patterns (i.e. natural language processing, computer vision, classification). The goal of machine learning algorithms is to build models based on an abundant amount of training data. Since the types of data vary wildly from one problem to the next, from sentences in different languages to video data from a camera, a method of reconciling the data with mathematical models is required. In this respect, researchers usually convert elements of the data set into a list of features, represented using a vector of real numbers. Each feature is a measurable value that encapsulates some aspect of the original data. Mathematical models are built to take in these feature vectors and produce results in some fashion.

2.2 Autoassociative Memory

Autoassociative memories use learned patterns about the data to retrieve information, whereas traditional memories rely on a unique address to recall data. Autoassociative memory mirrors the memory of human minds, which use heuristics to interpret new observations using previous experiences. In essence, human minds also implicitly perform feature extraction. Since most humans do not have photographic memories, it is important to commit the specific features of observations to memory. For example, a dog can be visually described by its size, fur length, color, etc. Using those features, one could vaguely reconstruct what the original dog looked like. An autoassociative memory emulates these aspects of human memory using mathematical models, and is frequently implemented via neural networks. A neural network consists of a series of functions acting on the features of given input values. Using these functions, it can find relationships or make predictions based on its inputs. It also reacts and changes whenever it sees new input to enhance accuracy for future input. Autoassociative memories do not maintain a one-to-one relationship between inputs and stored entries. Outputs are constructed based on knowledge obtained from all of the stored entries.

2.3 Autoencoders

Autoassociative memory can be implemented using a type
of neural network called autoencoders. An autoencoder finds relationships between data by removing noise. Often, there are a lot of features associated with a particular piece of data, but they are not all equally useful in every application. Autoencoders seek to find the “essential” features of a given input by encoding the input vectors into a lower dimension via an encoding matrix. This encoded representation offers a more direct and accurate way to determine relationships between data.

\[ y = f(W_e x + b), \]

where \( f \) is a non-linear transformation such as a sigmoid function and \( b \) is a bias factor. The encoded representation \( y \) can then be decoded using a different decoding matrix to produce an output with the original dimension. This output is essentially what the memory “remembers” the original input to be. Through training, the values of the encoding and decoding matrices as well as the bias values will be optimized for finding relevant combinations of features in the encoding as well as reconstruction of the original input.

The work presented here heavily relies upon the ideas in autoencoders, and these ideas are extended to help work with recursively structured data.

3. RELATED WORK

Machine learning applications generally deal with linear, flat data in the form of vectors, and cannot process structured data as easily. There have been some strides in machine learning to deal with this issue; Sperduti [8] attempted to study the training of linear autoencoders for modeling structured input. There has also been research that involves autoencoders with NLP applications. In particular, Richard Socher [7] applied semi-supervised recursive autoencoders to sentiment analysis at the sentence level.

In addition, Socher [6] used the method of recursive autoencoders to perform paraphrase detection by measuring word and phrase similarity. These recursive autoencoders can function as a model of autoassociative memory, since they take in input data, recombine and apply patterns upon them, and output a value of the same type and format as the input data. However, Socher’s autoencoders only worked on binary trees, so his parse trees had to first be binarized. Thus, the input data still had to be transformed in a way that did not preserve its structure. Our programming language library improves upon Socher’s recursive autoencoders by supporting ways to encode data without sacrificing their complex structure.

4. SYSTEM MODEL

4.1 Interface

An overview of the library workflow is presented in Figure 2. As shown in the figure, the central component of this library interface is a memory, which requires a data type that is specified by the user. This type will correspond to the format of the training data, which the user also provides to the memory.

After the memory trains on the data, the user obtains an output from the memory based on the training from the previous step and the input value given to the memory.

Finally, it is likely that the user will have to apply some post-processing to the output values, for whatever purposes he desires.

4.2 Recursive Autoencoder

The main component of our library requires an autoassociative memory implementation that can handle arbitrary structured data. We begin by describing a traditional recursive autoencoder that only works on binary trees with vector-valued leaves, and discuss the changes upon this model that allow us to use any given recursive structure with the autoencoder.

A traditional recursive autoencoder works in the following fashion: Suppose we have a binary tree with a node \( y \) and children \( x_1 \) and \( x_2 \) that are leaves holding \( n \)-dimensional data vectors. We can encode the two children into a single node by concatenating the two vectors and encoding them using a \( n \times 2n \) encoding matrix \( W_e \). The equation for this encoding is fairly similar to a standard autoencoder

\[ y = f(W_e [x_1; x_2] + b), \]
where $f$ is a sigmoid function and $b$ is a bias value. Notice that $y$ will have dimension $n$, so that the parent of $y$ will be able to encode $y$ recursively (refer to Figure 3). As long as the binary tree is full and only has vector values at the leaves, this autoencoder will be able to encode a whole tree as a single vector of dimension $n$.

As a single vector of dimension $n$, this autoencoder will be able to encode a whole tree because the binary tree is full and only has vector values at the leaves.

Figures 3: Recursive Autoencoder

Decoding works in an intuitive fashion as well. Given an encoded tree represented as a single vector $y \in \mathbb{R}^n$, the autoencoder uses a decoding matrix $W_d \in \mathbb{R}^{2n \times n}$ to decode $y$ into a $2n$-dimensional vector, which can be split into two $n$-dimensional vectors $y_1$ and $y_2$ and decoded recursively,

$$[y_1; y_2] = f(W_d y + b).$$

The recursion ends when the decoded structure matches that of the input structure.

This model of recursive autoencoder works well for the purposes it was designed for, but there are two main drawbacks. First, the structure that it can work on is very rigidly defined, which necessitates that users convert their data types. Second, in the decoding step, the original structure must be supplied to create a sensible output. To address these issues, we propose augmenting the traditional recursive autoencoder with tags as well as a variable number of encoding/decoding matrices such that it better handles generic data structures.

4.3 Tagged Recursive Autoencoder

In the recursive autoencoder described above, it was sufficient to keep one encoding matrix of size $n \times 2n$. We can examine the type definition of the structure to confirm that this works. For the tree described above, there are only two possible identities for a given node in the tree: either it is a leaf or it has two children. A leaf does not need to be encoded, so only one matrix is needed to handle the recursive case. This obviously does not work in general, as one can define a tree to have two or three children, or to contain values at the internal nodes. For example, if a node had 3 children, the autoencoder would need to use a $n \times 3n$ matrix to encode it.

For the autoencoder to work with many different recursive cases of a given data type, it will need to maintain a different matrix for each recursive case. Although it is possible to generate these matrices using compiler introspection of the given data type, it is out of the scope of this paper, and we shall discuss it further in the Future Work section.

Since we cannot automatically generate the matrices based on the data type, the user must provide a “description” of his type definition, which is essentially a list of the entries in each recursive case. Upon initialization, the autoencoder will create the matrices required using the “description” given by the user. For encoding, the only change will be choosing which encoding matrix to used based on the recursive case at runtime. Using multiple encoding matrices, it is possible to get an encoding of the data structure as a single vector. Decoding can be done in the same way as a traditional recursive autoencoder, by providing the input structure and recursively decoding until the structure matches.

Now, we discuss the second aforementioned drawback of traditional recursive autoencoders: the structure of the input must be provided to decode. Since the autoencoder can learn how to encode the data vectors as a single vector, it is a natural extension to try to encode the structure itself. To incorporate the structure into the autoencoder, we can use a set of orthonormal vectors, or tags, to annotate which recursive case each piece of data belonged to. At construction, the autoencoder can create a set of random tags based on the number of recursive cases, much like the encoding/decoding matrices. Described below are two ways in which tags can be used to encode the structure of the data along with the data vectors.

4.3.1 Tags as Data

One possible method is to simply treat the tags as part of the data. Given a node $x \in \mathbb{R}^n$ in the tree, either a leaf or an encoded representation of its subtree, we can prepend a tag $t \in \mathbb{R}^k$ to $x$ (where $k$ is the number of recursive cases) to get a $n + k$ dimensional vector $y$. Then, $y$ can be encoded as a data vector $y \in \mathbb{R}^n$ using a special tag encoding matrix $T \in \mathbb{R}^{n \times (n+k)}$ (refer to Figure 4). Afterwards, the autoencoder can proceed as before by concatenating the children and applying the appropriate transformation matrix.

Figures 4: Tags as Data

The decoding step is where we can make use of the tags to recover the structure of the data. Starting with a single vector $y$, we apply the tag decoding matrix $T_d$ to get a vector
\( y_t \in \mathbb{R}^{n+k} \). Next, \( y_t \) can be split into a tag \( t \) and a data vector \( x \). Based on the tag \( t \), the autoencoder will choose the mostly likely structure that the original data had, and decode the data portion accordingly.

### 4.3.2 Parallel Tag Training

Another possible method is to train the tags in the same fashion that the data is trained. Given two nodes \( x \) and \( x_2 \), and their tags \( t \) and \( t_2 \) as defined in the previous section, we can concatenate the tags together with a tag denoting the "node" structure, \( t_3 \). The data is concatenated as in the original recursive autoencoder, and both the concatenated tag and data vectors are individually passed through the encoding matrices \( T \) and \( W \), respectively, where \( T \in \mathbb{R}^{k \times 3k} \).

In the end, a \( k + n \) sized vector is returned.

### 4.4 Optimization

In a similar fashion to standard machine learning applications, the user will start with a set of training data. The user will then need to specify the type and structure of the data he is inputting to the library, so it will know how to process it appropriately.

Moreover, the error function will need to be supplied by the user as well. While many machine learning packages that deal with vectors may offer some standard preset error functions such as the Euclidean norm, applying those error functions to data structures is not standardized. Thus, it will be a requisite input to this library.

To train, it will take the training data provided, and for each entry, the reconstitution error will be calculated. That is, the data will be encoded and decoded, and then compared to the original data using the provided error function. This sum of the reconstitution errors will be the function that this library attempts to optimize, and will be the benchmark for accuracy.

### 5. SYSTEM IMPLEMENTATION

#### 5.1 Choice of Language

In order to implement a generalized process for handling structured data, the programming language library has been written in OCaml. OCaml is a functional language, so it offers more fluidity when handling structured data. In addition, OCaml has procedural elements that facilitates the ease of implementing and prototyping the various algorithms and models that have been encountered so far.

An OCaml code snippet in Figure 6 is provided as an example of the interface delineated in Section 4.1.

```ocaml
(* memory containing objects of type 'a *')
type 'a t

(* the type of vectors *)
type vector

(* a sample type that can be used in the memory *)
(* custom types may only use vector as the internal data type *)
type tree =
  | Leaf of vector
  | Node of tree * vector * tree

(* create a memory of 'a objects containing vectors of a fixed size *)
(* provide an error function on 'a objects *)
val create : int -> ('a -> 'a -> float) -> 'a t

(* encode an object as a vector *)
val encode : 'a -> 'a t -> vector

(* decode a vector into an 'a object *)
val decode : vector -> 'a t -> 'a

(* train the memory on a list of 'a values *)
val train : 'a list -> 'a t -> unit
```

Figure 6: Example Interface Written in OCaml

As is clear from the comments, tree is just an example of a type that can be used with this library.

In addition, the encode and decode functions work together to obtain a value from the memory: After the training data is fed to the memory, an object can be encoded as a vector, and then decoded back into an object, where both operations take place in conjunction with the memory. This implements the output retrieval scheme outlined in Section 4.1.

#### 5.2 Third-party Libraries

We had to utilize certain third-party libraries in order to achieve performance and functionality goals. For adjusting the OCaml runtime itself, we used Jane Street’s Core library [1] for tuning the garbage collection to improve performance.

To perform the linear algebra operations involved in implementing the different autocoders delineated in Section 4, we used a library called Lacaml [3] that provides OCaml bindings for BLAS (Basic Linear Algebra Subprograms) [4] and LAPACK (Linear Algebra PACKage) [5]. These are two well-established, robust and performant linear algebra libraries that are commonly used for machine learning applications. Notably, BLAS at the low level is widely used in scientific computation in many more complicated higher level languages, such as MATLAB or R. This allows efficient performance with matrix operations and mathematical operations, while also allowing much of the higher-level logic to remain within within OCaml.

#### 5.3 Gradient Descent

Gradient descent is used to perform the optimizations on the encoding/decoding matrices of the autoencoders. It is a
standard, yet relatively simple method of minimizing a function. To do so, it requires the gradient of a function - that is, the partial derivative of the function with respect to all of its parameters. Since the partial derivative of a function with respect to a parameter represents the rate of change of that function with respect to a small positive change in the parameter, a positive partial derivative indicates that the parameter should be lower, while a negative partial indicates that the parameter should be higher.

However, it is possible that by increasing or decreasing the parameter values, it “overshoots”, and thus ends up with a higher function value. Therefore, the aggressiveness of this parameter adjustment must be adaptive in order to maintain a balance between efficiency and correctness. In this way, given starting parameters, gradient descent can eventually find slightly more optimal values from that starting point.

This library’s implementation of gradient descent is relatively simple. The structure of the routines were initially roughly based on example code posted online by the writers of *F# for Scientists*, and were adapted for this library [2]. It takes in a function, the derivative of that function, and starting parameters, and continually adjusts parameter values as described above until either a fixed point is reached, or a specified max number of iterations is reached. It utilizes the bindings in Lacaml to perform vector and matrix operations more efficiently, while the main loop is still inside OCaml. This way, the flexibility in addressing data structures in error calculation is maintained, while being able to obtain faster performance.

6. RESULTS

6.1 Experiments

Experiments for the programming library were performed on two datasets, one “small” and one “large”. The small dataset consisted of two trees with 10-dimensional vectors as the internal data type, and two trees with 20-dimensional vectors as the internal data type.

We encountered significant performance issues regarding certain portions of our code. A profiler was run, and there seems to be a large amount of time spent in a vector map function, which we call to apply a sigmoid function after matrix transformations. While the runtime was reduced by using a faster sigmoid function (we had been previously using \texttt{tanh}), the number of calls could not be reduced, and thus is still a bottleneck.

Two other issues were a large number of calls to methods that find matrix and vector dimensions, and a large amount of time being spent in the garbage collector. In order to fix these issues, we experimented with certain optimizations, which included compiling the library with an “unsafe” option to remove array bound checking and tuning the OCaml garbage collection parameters for better memory performance. Finally, we tested these datasets for both tagged and untagged implementations of our programming library. The graphs for these experiments\(^1\) are shown in Figure 7 and Figure 8.

\(^1\)These experiments were performed on a Quad-core Intel Core i7-4700MQ @ 2.4 GHz

\[\text{Figure 7: Graph with Small Data Sets}\]

\[\text{Figure 8: Graph with Large Data Sets}\]

6.2 Evaluation

These graphs compare the training error (with respect to the error function) to the number of iterations that it took to complete the test. The lower bound for training error is 0, and the closer the graphs converge to 0, the better the error.

For the untagged tests, we used the standard Euclidean metric in our tree error function. The error of two trees \(T_1\) and \(T_2\) is simply the sum of the distances of each leaf in \(T_1\) to its corresponding leaf in \(T_2\).

As can be seen from the graphs, the optimized versions of the untagged tests for both small and large datasets took fewer iterations to complete, so our optimizations indeed improved the performance of our library.

On the other hand, the tagged tests struggled to converge. While they appear to be better in earlier iterations, that is simply due to an adjustment that needed to be made with the error function to handle non-matching tree structures. To deal with trees where the structure is different, the norm of any vectors in non-matching areas will be added to the error. However, they are first multiplied by a factor that scales with the inverse of the depth of that vector in the
tree.

The lack of convergence was determined to likely be due to the non-continuous nature of the error function once the structure of the data was introduced in the encoding/decoding. Since mismatched structures are likely to be penalized more than mismatched values, and since structural decoding is done with strict thresholds, the optimization functions could not determine the direction of optimization unless the initial values were already on the cusp of changing the structure of the encoded/decoded data. Therefore, the training simply attempted to optimize for smaller values in the mismatched values, which resulted in convergence to a local minimum.

7. FUTURE WORK

7.1 Code Generation for Optimization

Our optimization routines currently accept functions with a fixed number of arguments to ensure an accurate compiler-level representation of the types involved. That is, if the error function takes in two matrices and two vectors as inputs, that should be reflected in the typing. However, this approach leads to difficulties when it comes to generalizing code to any supplied structure. A solution to this would simply be to accept functions that take lists of matrices and vectors. However, this removes the semantics behind the arguments. A better solution may be to dynamically generate type-safe code that encapsulates the types of the arguments.

7.2 Compiler Introspection

Currently, while our system model supports user-supplied data structures, it requires a substantial amount of user specification for the library to function. The end goal would be to achieve a level of flexibility where the user may simply provide data, and the runtime type would be determined and all of the memory operations would adapt to account for the type. However, this requires a higher level of introspection than OCaml provides, as variants cannot be enumerated programmatically. Thus, some degree of compiler-level introspection must be used in order to provide this functionality.

7.3 Error Functions for Tagged Structures

For the tagged structure experiments, our optimization routines struggled to find values that resulted in accurately encoded and decoded structures. As mentioned earlier, this is likely due to the error function no longer being differentiable with respect to the parameters. Since the gradient will likely be too short-sighted to accurately reflect the adjustments necessary, a possible route forward would be to annotate the data structures with the tag, resulting in a “soft” structure declaration as well as the actual structure. The error function, therefore, would be able to detect not only if two structures are different, but also how different they are, and could therefore optimize in the direction that causes the structures to be more similar.

8. ETHICS

Since this library interface is heavily research-based, there are no obvious ethical issues to consider. However, if the library were put into a production system, it would likely see use in machine learning applications, and there are plenty of ethical issues related to machine learning. In this context, our programming library interface would face similar ethical issues. For instance, machine learning applications work on large sets of data, and some of this data can be quite personal and private. In a certain sense, machine learning models that only work on flat vectors obfuscate this data because the data must be converted and transformed from their original structure into something different. This interface, however, would not perform this data obfuscation because it is able to accept the data in its original structure. This could potentially create some privacy issues, since the end user of this library would have an easier way – either intentionally or unintentionally – of gathering personal, private information from the collected datasets.

9. CONCLUSIONS

This library started as an endeavor to create a interface to offer programming support for autoassociative memories. There were two main goals - offer a flexible user interface, and allow support for structured data. Structured data was handled in a method similar to that in recursive autoencoders, with additional experimental methods explored as well. Unfortunately, this complicated the user interface, which currently requires substantial user specification of data structure-related information. However, the modularity of our methods implies that there are distinct avenues forward in order to provide a better user experience. Moreover, the exploration of tagged autoencoders also indicates potential ways forward with “soft structure” error functions.

10. REFERENCES