ML Methods to Learn Logical Semantics

Jiageng Zheng

https://github.com/jiz322/Neural-Network-to-Learn-Logical-Semantics

Abstract

We implemented recursive neural networks for predicting satisfiability of predicates. On test data, our TreeRNN implementation achieves 93.4% accuracy. Other neural network models are used as baselines, as we try to account for their relatively low performance compared to the TreeRNN based on theoretical reasoning and experimental results.

1. Motivation

As we learned in class, correctly recognizing the entailment relationship is crucial for knowledge base systems. We have learned methods like forward chaining and backward chaining algorithms. These algorithms provide the best accuracy, but they may not have a good performance in terms of speed. The ML models will make faster predictions but lower accuracy. Since the accuracy of supervised learning is dependent on the abundance of data, in this experiment, we do not expect any model to provide a perfect accuracy. We will implement different ML methods and compare their accuracy. The RNN model is our focus.

2. Goal

We want to train RNN models which can predict whether a logical statement is satisfiable. We will also train ML models based on summation of embeddings as our baseline for comparison.

3. Related Papers

[1] RNN can Learn Logical Semantics.
[2] Tree-Structured Composition in Neural Networks without Tree-Structured Architecture

4. Related Git Repository

[3] https://github.com/sleepinyourhat/vector-entailment

5. Dataset: ALCIR-data.txt

This dataset is found in the related repository [3], the repository used for both paper [1] and [2]. This data set is not mentioned in any of these papers. We have not yet found its accuracy evaluation anywhere for this ALCIR data.
It has 10,000 predicates labeled with" satisfiable" or "unsatisfiable." If satisfiable, there has to exist at least one truth value combination of p and q that makes this predicate true.
Each entry is of the form as below.

satisfiable *(p (<=> ((q (| p)) (& p))))*

We split this into a training set (8,000 data) and a test set (2000 data). One weakness of our experiment is that different training sets and test sets are used across models due to shuffling. This should have been easily avoided by setting a random seed. The impact should not be huge since the amount of entries is not too tiny, and it does not make any model more preferable.

6. Methodologies: Neural Network Models

6.1. Sum of Embeddings

The implementation of the same type of baseline is discussed in the paper [1]. We embed each character including parentheses to a vector length of 70. Passing through two linear layers, we get a vector length of 2. The final softmax layer gives the possibility of both predictions. The cross-entropy loss is used as the loss function during training. Since this method lacks explainability, nothing can be explained more.

6.2. Neural Network with One LSTM Layer

Intuitively, the LSTM layer captures sequential relationships. But compared to TreeRNN, it has received less help from human knowledge. In this implementation, the parentheses are embedded. We fixed the length of embedding to 70. After embeddings are ready, they pass through one LSTM layer, and the final hidden state is retrieved. Then, the hidden state is passed through multiple linear layers, dropout layers, and the tanh layers, ending up with a vector length of 2. In the Sum model, the same activation and loss functions are used. It is interesting to see whether the LSTM algorithm can learn the semantics of parentheses throughout the training.

6.3. Tree RNN

We implement the TreeRNN discussed in paper [1] with a few variations. Different from the previous two models, this one does not embed parentheses anymore: the parentheses only help to build binary trees. We used binary trees to store each predicate, where leaf nodes are character embeddings, and the parent nodes are vectors computed using a formula from paper [1].

$$(1) \quad \vec{y}_{TreeRNN} = f(\mathbf{M} \begin{bmatrix} \vec{x}^{(l)} \\ \vec{x}^{(r)} \end{bmatrix} + \vec{b})$$

The two x vectors are the embeddings of the left and right child. The M matrix and b vector can be easily expressed using a linear layer from PyTorch. In the original paper, the activation function f is the tanh, but we decided to change it to sigmoid. The root node gives an output of this Tree RNN. Then, we pass this output to a linear layer, which gives a scalar value. The sigmoid of this scalar value is returned (a scalar from 0 to 1). Different from previous models, this is a task of logistic regression, so the BCE loss is used.

7. Evaluation Metrics

7.1. Loss value with Respect to Epoch Numbers

Throughout the training process, the loss values for each epoch are recorded as in the plots. These plots reflect the performance of the model changing with epoch numbers. Moreover, the loss values of the Sum Model (section 6.1) and the LSTM Model (section 6.2) are comparable since they use the same loss function.

7.2. Prediction accuracy, precision, recall, and f1 scores

$$accuracy = \frac{number\ of\ satisfiability\ correctly\ predicted}{total\ number\ of\ entries\ in\ test\ set}$$

The i below can only have 2 classes: satisfiable or unsatisfiable

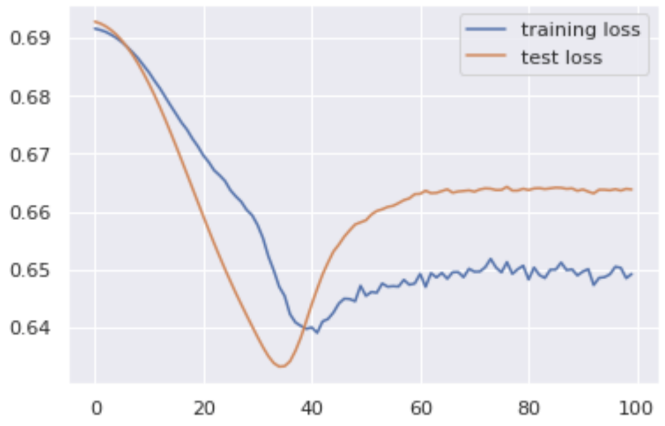
$$recall_i = \frac{number\ of\ class_i\ predicates\ correctly\ classified}{number\ of\ predicates\ classified\ as\ class_i}$$

$$precision_i = \frac{number\ of\ class_i\ predicates\ correctly\ classified}{number\ of\ class_i\ predicates}$$

$$F1_i = 2\frac{precision_i \cdot recall_i}{precision_i + recall_i}$$

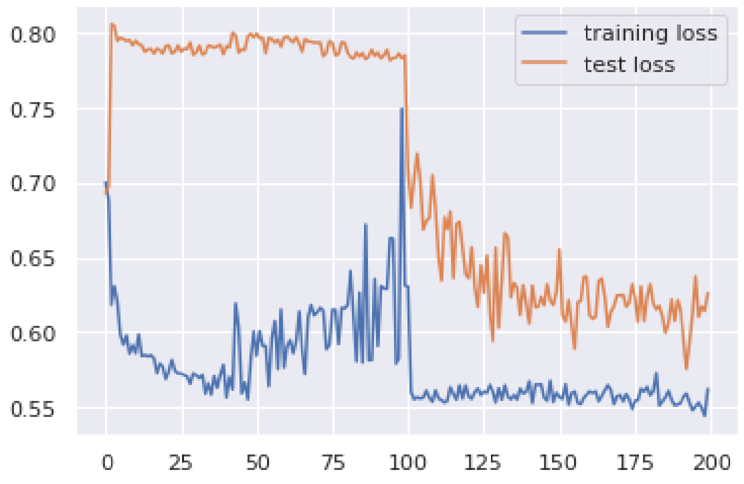
8. Result Analysis and Observations

8.1. Training Process



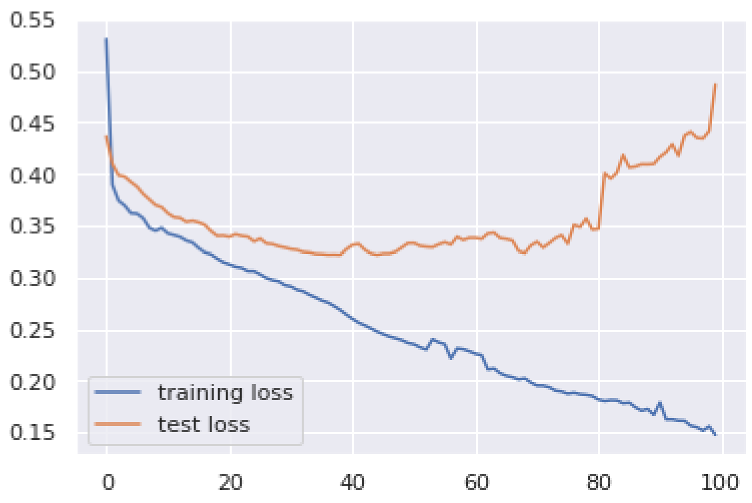
The Learning Curve of the Sum model (section 6.1)

Above is the learning curve of the sum model. After some tuning of learning rate, our minimum test loss is at 0.633. The overfitting problem occurred at epoch 40.



The Learning Curve of the LSTM model (section 6.2)

Above is the learning curve of the LSTM model. In the first 100 epochs, the test loss was maintained at roughly 0.78. Then, we adjusted the learning rate to lr= 0.00001/(1.5*(num_epochs_train+1)) and run 100 more epochs. The Best loss pair is Train Loss: 0.559 | Test Loss: 0.576. We observed that both training and test loss have larger fluctuations than the sum model. This may be due to its higher complexity (more layers and parameters). The final loss is less than the sum model, indicating this method has better performance.

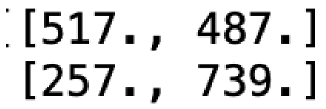


The Learning Curve of the TreeRNN model (section 6.3)

Above is the learning curve of the TreeRNN model. The training loss keeps decaying, but the overfitting occurs around the 80th epoch. The value of the loss is not comparable with the previous two models since they are using different loss functions.

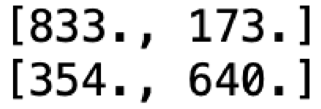
8.2. Confusion Matrix

For each epoch of each model, we built a confusion matrix for predictions on the test set. The diagonal of the confusion matrix represents the correct predictions. The row represents real labels, the column represents the predictions (index 0 means unsatisfiable, 1 means satisfiable). The graphs below are the best results for each model.



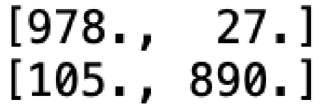
```
[517., 487.]
[257., 739.]
```

Best Confusion Matrix for the Sum Model



```
[833., 173.]
[354., 640.]
```

Best Confusion Matrix for the LSTM Model



```
[978.,  27.]
[105., 890.]
```

Best Confusion Matrix for the TreeRNN Mode

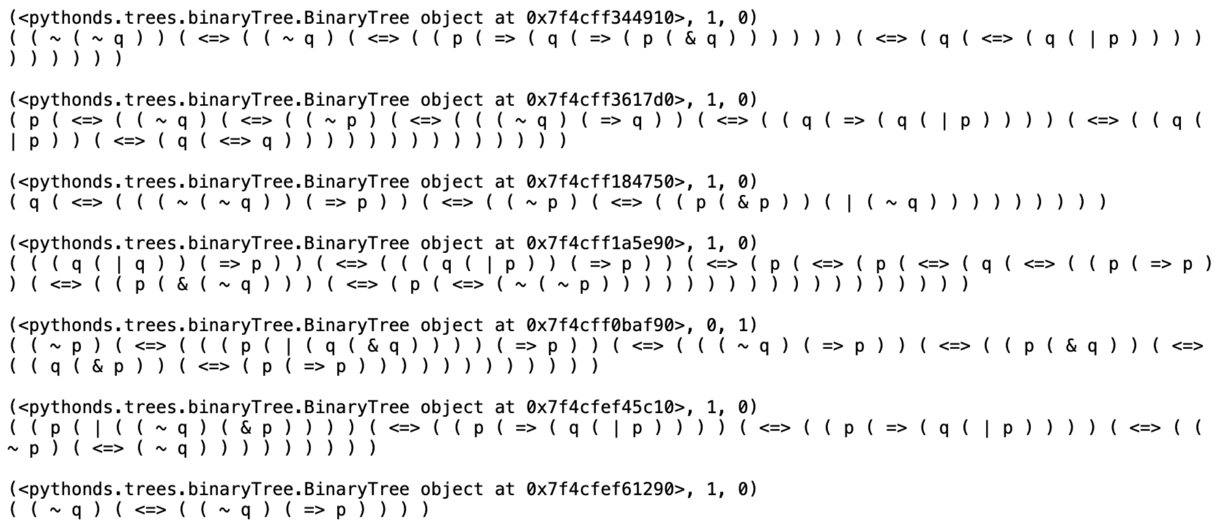
From the confusion matrix, we computed precision, recall, F1 score and accuracy collected in the chart below.

	Pcs_sat	Pcs_uns	rc_sat	rc_uns	F1_sat	F1_uns	Accuracy
Sum	0.742	0.515	0.602	0.668	0.665	0.582	0.628
LSTM	0.644	0.836	0.787	0.702	0.708	0.763	0.737
TRNN	**0.894**	**0.993**	**0.971**	**0.903**	**0.931**	**0.946**	**0.934**

Reflected by their values of precision and recall, we can see the Sum model tends to make satisfiable predictions, whereas LSTM and TreeRNN prefer to say a predicate is unsatisfiable. For all evaluations, the TreeRNN is always the best. This is reasonable because both Sum and LSTM need to learn the semantics of parentheses but TreeRNN does not.

8.3. Error Analysis

We printed out a list of errors made by our best model (TreeRNN). Here are some examples.



```
(<pythonds.trees.binaryTree.BinaryTree object at 0x7f4cff344910>, 1, 0)
( ( ~ ( ~ q ) ) ( <=> ( ( ~ q ) ( <=> ( ( p ( => ( q ( => ( p ( & q ) ) ) ) ) ) ( <=> ( q ( <=> ( q ( | p ) ) ) )
) ) ) ) ) )

(<pythonds.trees.binaryTree.BinaryTree object at 0x7f4cff3617d0>, 1, 0)
( p ( <=> ( ( ~ q ) ( <=> ( ( ~ p ) ( <=> ( ( ( ~ q ) ( => q ) ) ( <=> ( ( q ( => ( q ( | p ) ) ) ) ( <=> ( ( q (
| p ) ) ( <=> ( q ( <=> q ) ) ) ) ) ) ) ) ) ) ) ) ) )

(<pythonds.trees.binaryTree.BinaryTree object at 0x7f4cff184750>, 1, 0)
( q ( <=> ( ( ( ~ ( ~ q ) ) ( => p ) ) ( <=> ( ( ~ p ) ( <=> ( ( p ( & p ) ) ( | ( ~ q ) ) ) ) ) ) ) ) )

(<pythonds.trees.binaryTree.BinaryTree object at 0x7f4cff1a5e90>, 1, 0)
( ( ( q ( | q ) ) ( => p ) ) ( <=> ( ( ( q ( | p ) ) ( => p ) ) ( <=> ( p ( <=> ( p ( <=> ( q ( <=> ( ( p ( => p )
) ( <=> ( ( p ( & ( ~ q ) ) ) ( <=> ( p ( <=> ( ~ ( ~ p ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) )

(<pythonds.trees.binaryTree.BinaryTree object at 0x7f4cff0baf90>, 0, 1)
( ( ~ p ) ( <=> ( ( ( p ( | ( q ( & q ) ) ) ) ( => p ) ) ( <=> ( ( ( ~ q ) ( => p ) ) ( <=> ( ( p ( & q ) ) ( <=>
( ( q ( & p ) ) ( <=> ( p ( => p ) ) ) ) ) ) ) ) ) ) ) )

(<pythonds.trees.binaryTree.BinaryTree object at 0x7f4cfef45c10>, 1, 0)
( ( p ( | ( ( ~ q ) ( & p ) ) ) ) ( <=> ( ( p ( => ( q ( | p ) ) ) ) ( <=> ( ( p ( => ( q ( | p ) ) ) ) ( <=> ( (
~ p ) ( <=> ( ~ q ) ) ) ) ) ) ) ) )

(<pythonds.trees.binaryTree.BinaryTree object at 0x7f4cfef61290>, 1, 0)
( ( ~ q ) ( <=> ( ( ~ q ) ( => p ) ) ) )
```

Mistakes of TreeRNN

The tuple on the first line is (tree of predicate, label, prediction). The second line is the predicate retrieved from the tree after traversing. 1 represents satisfiable, and 0 represents unsatisfiable. From this graph, you can get some sense of the insanity of this dataset. The average height of the tree of all errors is 9.13, higher than that of all test sets at 8.10. It weakly suggests that the deeper the tree, the less accurate the prediction is. In the graph, the last predicate is short. Here we take a closer look.

```
( ( ~ q ) ( <=> ( ( ~ q ) ( => p ) ) ) )
```

This is not an easy one, since the only case to satisfy this predicate happens only when q is false and p is true. The "unsatisfiable" is a wrong prediction, but it seems to be a reasonable guess. It will be interesting to count how many mistakes are of this type.

9. Conclusion

In this experiment, we implemented the Sum model, LSTM, and TreeRNN. Being aware of the order of operation, the TreeRNN achieved 93.4% accuracy. The LSTM reaches 73.7% accuracy, Sum model gives 62.8% indicating both of them can find some patterns. The pattern found by these models is unclear, and it is interesting for further research.

Determining the satisfiability of a predicate is difficult for humans, but the computer can give 100 percent accuracy after parsing the predicate as a tree and enumerating all possible combinations of truth values. Let n represent the number of variables, the complexity of this problem is 2 to the power of n, which is a non-polynomial problem. In the sacrifice of some accuracy, the RNN method makes the complexity of this problem linear to the length of the predicate. A good accuracy can be achieved by implementing TreeRNN using the PyTorch library. It is still interesting to see the accuracy of this method when more variables are involved.