Review and Evaluation of Tree-To-Sequence Learning

Report by Sharvin Shah

Introduction:

Natural-language processing (NLP) is an area of computer science and artificial intelligence concerned with the interactions between computers and human (natural) languages, how to program computers to fruitfully process large amounts of natural language data[i]. With the continuous rise in text data, understanding it is one of the most important problems of this information age as people communicate with each other using language through different media like speech, text, web search, advertisements, emails, language translation, medical reports, customer services etc. Most of the natural language processing challenges like machine translation, speech recognition, language generation etc. are increasingly dealt with using machine learning models where data is perceived as a sequence of tokens. More recent models involve using neural network architectures specialized for processing sequential data like Recurrent Neural Networks (RNNs), Long Short-Term Memory Networks (LSTM) and Gated Recurrent Unit (GRU) along with its variations as they can deal with memory-based dependencies between tokens by using shared parameters. Encoder-Decoder Models specialize in tasks where a mode of data is encoded as memory cell or context by the encoder only to be decoded appropriately by the decoder depending on the application. Let us brief a bit on the components that would be comprising the components of this architecture.

The LSTM (Decoder) which is used for our implementation of tree-to-sequence model is governed by the following equations:

$$
\begin{aligned}
i_t &= \sigma\left(W^{(i)}x_t + U^{(i)}h_{t-1} + b^{(i)}\right), \qquad (1)\\
f_t &= \sigma\left(W^{(f)}x_t + U^{(f)}h_{t-1} + b^{(f)}\right),\\
o_t &= \sigma\left(W^{(o)}x_t + U^{(o)}h_{t-1} + b^{(o)}\right),\\
u_t &= \tanh\left(W^{(u)}x_t + U^{(u)}h_{t-1} + b^{(u)}\right),\\
c_t &= i_t \odot u_t + f_t \odot c_{t-1},\\
h_t &= o_t \odot \tanh(c_t),
\end{aligned}
$$

The LSTM unit at each time step t is a collection of vectors in R^d : an *input gate* i_t a *forget gate* f_t an *output gate* o_t, a *memory cell* c_t and a *hidden state* h_t. The gating vectors like input, forget and output have entries in [0, 1]. X_t is the input at current time step, σ denotes the logistic sigmoid function and $\odot$ denotes elementwise multiplication. Intuitively, the forget gate controls the extent to which the previous memory cell is forgotten, the input gate controls how much each unit is updated, and the output gate controls the exposure of the internal memory state. The hidden state vector in an LSTM unit is therefore a gated, partial view of the state of the unit's internal memory cell.

The difference between the standard LSTM unit and Tree-LSTM units is that gating vectors and memory cell updates are dependent on the states of possibly many child units. Additionally, instead of a single forget gate, the Tree-LSTM unit contains one *forget gate* f_{jk} for each child k. This allows the Tree-LSTM unit to selectively incorporate information from each child.

Given a tree, let *C(j)* denote the set of children of node *j*. The Child-Sum Tree-LSTM (encoder) transition equations are the following:

$$\tilde{h}_j = \sum_{k \in C(j)} h_k, \tag{2}$$

$$i_j = \sigma\left(W^{(i)} x_j + U^{(i)} \tilde{h}_j + b^{(i)}\right), \tag{3}$$

$$f_{jk} = \sigma\left(W^{(f)} x_j + U^{(f)} h_k + b^{(f)}\right), \tag{4}$$

$$o_j = \sigma\left(W^{(o)} x_j + U^{(o)} \tilde{h}_j + b^{(o)}\right), \tag{5}$$

$$u_j = \tanh\left(W^{(u)} x_j + U^{(u)} \tilde{h}_j + b^{(u)}\right), \tag{6}$$

$$c_j = i_j \odot u_j + \sum_{k \in C(j)} f_{jk} \odot c_k, \tag{7}$$

$$h_j = o_j \odot \tanh(c_j), \tag{8}$$

where in Eq. 4, $k \in C(j)$.

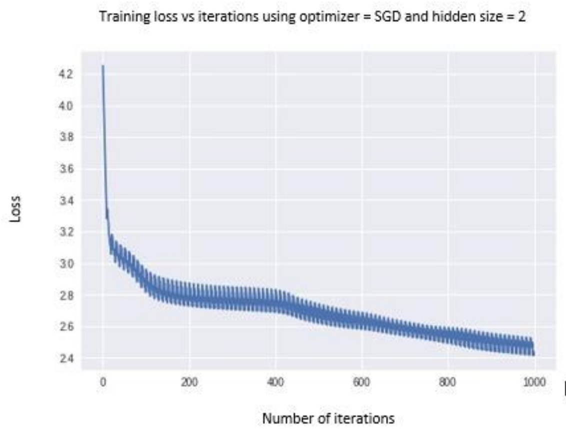
Intuitively, we can interpret each parameter matrix in these equations as encoding correlations between the component vectors of the Tree-LSTM unit, the input x_j , and the hidden states h_k of the unit's children. For example, in a dependency tree application, the model can learn parameters *W(i)* such that the components of the input gate i_j have values close to 1 (i.e., "open") when a semantically important content word (such as a verb) is given as input, and values close to 0 (i.e., "closed") when the input is a relatively unimportant word (such as a determiner).[ii]

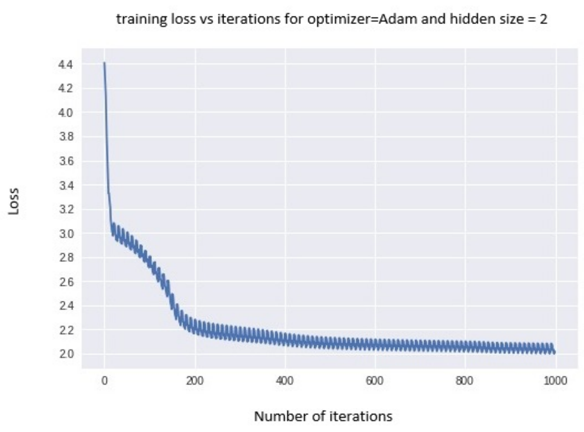
The way the above 2 components would interact is the deserialization of sequence done by the tree while encoding and serialization carried out by the sequential decoder to generate sequence from the context vector. The loss would be calculated, followed by the backpropagation step and the update of parameters to reduce the loss over multiple iterations. Let's discuss how we went about the project and what we found in the process in the following section.

Methodology and Results:

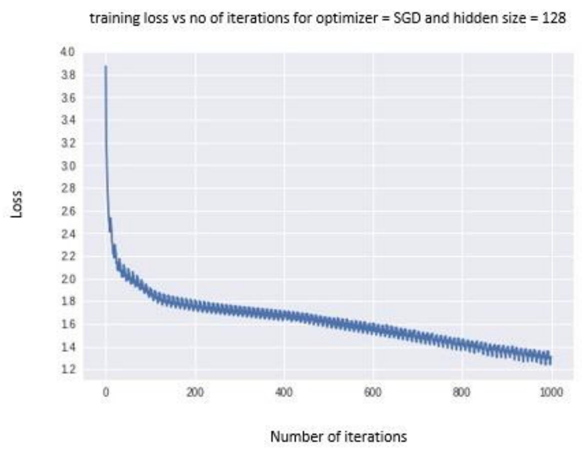
The project involved implementation of a Sequence-to-Sequence model which was adapted from its PyTorch implementation used for Machine Translation between English and French languages to understand the role of encoder, decoder along with the effect of attention on the performance of the generated translations. It was followed by implementing Tree-to-Sequence based model which was implemented in PyTorch where the Tree-LSTM (Tree-structured Long Short-Term Memory) variant Child-Sum Tree-LSTM (Dependency Tree-LSTM) was used as the encoder and a normal RNN (Recurrent Neural Network) was used as the decoder. The idea behind it was to generate a context vector with the tree encoding and the sequence-based decoder would learn the serialization process generating equivalent sequences after training on the dataset for multiple epochs. Our input sequences were deserialized into tree-based structures using NLTK (Natural Language Toolkit) library in and was encoded into the context vector using Tree-LSTM and its last hidden state was used to initialize the first hidden state for the decoder. The input for the decoder was initialized with a simple zero vector, and by using teacher forcing we gave the target variable for the previous state as the input for the current state. We chose negative likelihood loss as our loss parameter and all the associated parameters were updated accordingly after backpropagation to reduce the loss (to reflect learning and/or memorization). Shout out to Riddhiman Dasgupta[iii] for his insightful implementation on Tree-LSTM which helped in improving its understanding.

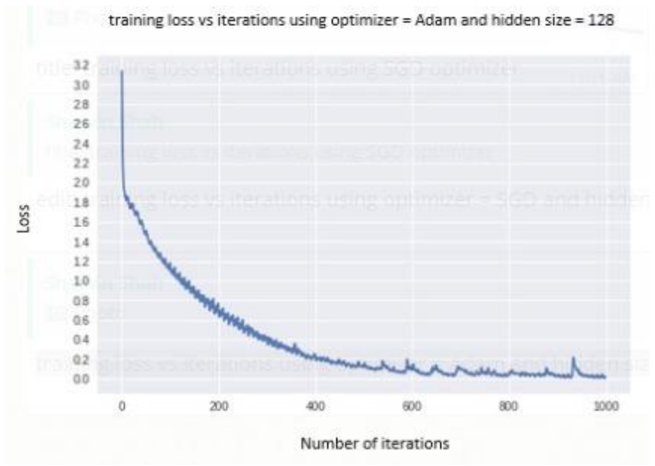
After setting up the model to train for multiple (1000) epochs, we experimented with some of the parameters like the hidden size for the encoder and optimizers with its parameters like learning rate etc. While we started with simple stochastic gradient descent (SGD) for the optimizer, we found that the training loss fluctuated a lot more. Increasing the learning rate made the loss overshoot and miss the optimum and decreasing the learning rate would prevent the reaching of optimum. We therefore went with more adaptive learning-based methods for optimization like Adam to resolve this issue. For the hidden size 2 with SGD optimization (left) and Adam optimization (right), please find the images below with average loss as the Y-axis and no of epochs as the X-axis.





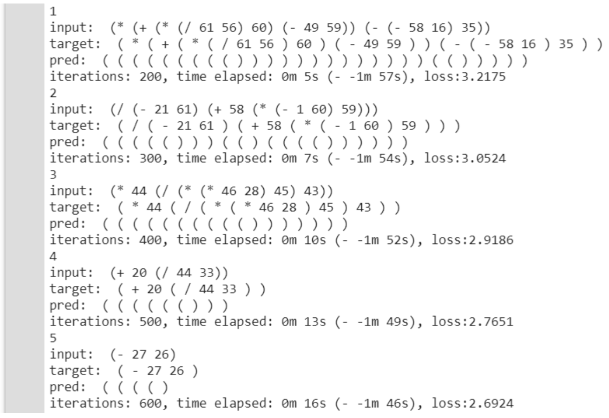
As we can see, there are a lot more fluctuations on the left than on the right. Having said that, neither of the optimizers do a good job because of the bottleneck being the hidden size. Therefore, we increase the hidden size to 128 for these 2 optimizers now and compare the results below. Please find the relevant images below with average loss as the Y-axis and the no. of epochs as the X-axis with the one with the SGD optimizer to the left and the Adam optimizer to the right.





As you can see, the fluctuations are still prevalent with SGD (left) and it still has not converged at all. In comparison Adam optimizer has almost perfectly memorized the dataset and the training loss almost reached zero. While one may be quick to say that clearly Adam is better, it can also overfit and not perform similarly on the testing set.

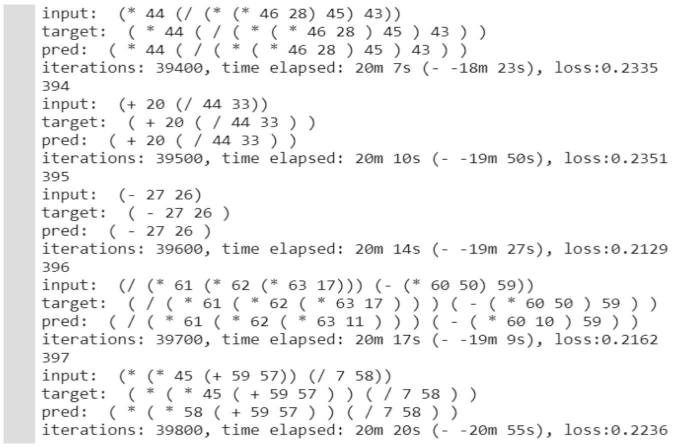
Also, we printed out predicted results and compared them with the input and true output (target variable) during different stages of iteration to gain an insight as to how the model memorizes. The results are insightful and as listed below:



```
1
input:  (* (+ (* (/ 61 56) 60) (- 49 59)) (- (- 58 16) 35))
target:  ( * ( + ( * ( / 61 56 ) 60 ) ( - 49 59 ) ) ( - ( - 58 16 ) 35 ) )
pred:  ( ( ( ( ( ( ( ( ( ) ) ) ) ) ) ) ) ) ) ) ) ) ( ( ) ) ) ) )
iterations: 200, time elapsed: 0m 5s (- -1m 57s), loss:3.2175
2
input:  (/ (- 21 61) (+ 58 (* (- 1 60) 59)))
target:  ( / ( - 21 61 ) ( + 58 ( * ( - 1 60 ) 59 ) ) )
pred:  ( ( ( ( ( ) ) ) ( ( ) ( ( ( ( ) ) ) ) ) )
iterations: 300, time elapsed: 0m 7s (- -1m 54s), loss:3.0524
3
input:  (* 44 (/ (* (* 46 28) 45) 43))
target:  ( * 44 ( / ( * ( * 46 28 ) 45 ) 43 ) )
pred:  ( ( ( ( ( ( ( ( ( ( ) ) ) ) ) ) )
iterations: 400, time elapsed: 0m 10s (- -1m 52s), loss:2.9186
4
input:  (+ 20 (/ 44 33))
target:  ( + 20 ( / 44 33 ) )
pred:  ( ( ( ( ( ( ) ) )
iterations: 500, time elapsed: 0m 13s (- -1m 49s), loss:2.7651
5
input:  (- 27 26)
target:  ( - 27 26 )
pred:  ( ( ( ( )
iterations: 600, time elapsed: 0m 16s (- -1m 46s), loss:2.6924
```

From the predictive results, you can see that initially the prediction involves parentheses and operators (seen from 10-11th epoch) as those are one of the first characters observed. The loss as a quantifiable metric needs a little justification.

From the observations, we observed that for predictions to be more visibly correct in the epochs we see printed (10 times per epoch), the loss (using negative log loss criterion) was well below 1 (around 0.65 for the simpler expressions to be accurately generated and correctly identifying positions of parenthesis and operators in the process for the larger expressions), and close to 0.25 where either there were no errors or just a token mismatch. One such example where the errors are lesser is as below:



```
input:  (* 44 (/ (* (* 46 28) 45) 43))
target:  ( * 44 ( / ( * ( * 46 28 ) 45 ) 43 ) )
pred:  ( * 44 ( / ( * ( * 46 28 ) 45 ) 43 ) )
iterations: 39400, time elapsed: 20m 7s (- -18m 23s), loss:0.2335
394
input:  (+ 20 (/ 44 33))
target:  ( + 20 ( / 44 33 ) )
pred:  ( + 20 ( / 44 33 ) )
iterations: 39500, time elapsed: 20m 10s (- -19m 50s), loss:0.2351
395
input:  (- 27 26)
target:  ( - 27 26 )
pred:  ( - 27 26 )
iterations: 39600, time elapsed: 20m 14s (- -19m 27s), loss:0.2129
396
input:  (/ (* 61 (* 62 (* 63 17))) (- (* 60 50) 59))
target:  ( / ( * 61 ( * 62 ( * 63 17 ) ) ) ( - ( * 60 50 ) 59 ) )
pred:  ( / ( * 61 ( * 62 ( * 63 11 ) ) ) ( - ( * 60 10 ) 59 ) )
iterations: 39700, time elapsed: 20m 17s (- -19m 9s), loss:0.2162
397
input:  (* (* 45 (+ 59 57)) (/ 7 58))
target:  ( * ( * 45 ( + 59 57 ) ) ( / 7 58 ) )
pred:  ( * ( * 58 ( + 59 57 ) ) ( / 7 58 ) )
iterations: 39800, time elapsed: 20m 20s (- -20m 55s), loss:0.2236
```

The generated sequence is visibly like the target sequence or has some token mismatches in numbers, which is a clear indication of learning (memorization) success of the serializer.

Discussion and Future Directions:

While the training was successful for the small dataset that we worked with to evaluate our implementation, the training time was unexpectedly high for the GPU and we found CPU to be quicker in evaluating the loss. This leaves an obvious extension of speeding up the training. One of the methods that can be tried is dynamic batching of trees to fasten learning. Hybrid models tree sequence architectures like SPINN (Stack-augmented Parser-Interpreter Neural Network) which blends recursive neural networks and recurrent neural networks, or maybe exploring to adapt Hogwild training into the implementation which is a method to parallelize Stochastic Gradient Descent.

One other obvious avenue could be explored which we originally intended to but couldn't due to time constraints was incorporating attention into it. Attention allows the decoder to focus on the most relevant portions of the sequence to ensure greater semantic similarity between the predicted sequence and the target sequence and can help in improving performance on the dataset.

Also, after incorporating the above extensions we would like to test it on larger datasets to test its efficacy and draw comparisons with the corresponding results generated by the sequence-to-sequence models.

I would like to take a moment to thank João for his endless support, understanding and patience. I was able to learn a lot by discussing ideas and issues with him, and he ensured that smooth learning was achieved (both by the machine and me ☺). In the end, I would like to thank Prof. Lyle Ungar for providing me the opportunity to work on an interesting problem and be able to learn from highly competent people in it.

References:

---

[i] Nadkarni PM, Ohno-Machado L, Chapman WW. Natural language processing: an introduction. Journal of the American Medical Informatics Association : JAMIA. 2011;18(5):544-551. doi:10.1136/amiajnl-2011-000464.

[ii] Tai, et al. "Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks." [1402.1128] Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition, 30 May 2015, arxiv.org/abs/1503.00075.

[iii] Dasguptar. "Dasguptar/Treelstm.pytorch." GitHub, github.com/dasguptar/treelstm.pytorch